Fast Many-to-Many Routing for Dynamic Taxi Sharing with Meeting Points*

Moritz Laupichler†  Peter Sanders†

Abstract
We introduce an improved algorithm for the dynamic taxi sharing problem, i.e. a dispatcher that schedules a fleet of shared taxis as it is used by services like UberXShare and Lyft. Shared. We speed up the basic online algorithm that looks for all possible insertions of a new customer into a set of existing routes, we generalize the objective function, and we efficiently support a large number of possible pick-up and drop-off locations. This lays an algorithmic foundation for taxi sharing systems with higher vehicle occupancy – enabling greatly reduced cost and ecological impact at comparable service quality. We find that our algorithm computes assignments between vehicles and riders several times faster than a previous state-of-the-art approach. Further, we observe that allowing meeting points for vehicles and riders can reduce the operating cost of vehicle fleets by up to 15% while also reducing rider wait and trip times.

1 Introduction
Current transportation systems are largely based on a combination of individual transport (often with heavy, polluting cars that consume a lot of energy and space) and public transportation that is often slow, inconvenient, and underdeveloped. Recently, taxi sharing systems that intelligently control fleets of shared taxi-like vehicles have garnered a lot of attention as a promising means of interopolating between the economical and ecological benefits of public transportation and the convenience and flexibility of individually used cars. The traffic engineering community has extensively studied the possible advantages of such systems in a large number of simulation studies and real-world field tests and a widespread adaptation of taxi sharing is expected to coincide with an increased demand for sustainable personal transportation and the availability of autonomously piloted vehicles. A main issue of current such systems is that the potential for shared rides is usually limited as each additional stop made to pick up or drop off a rider causes delays for other riders. This makes taxi sharing less attractive and makes larger capacity vehicles infeasible.

We focus on the question of how riders can use local transportation (e.g., walking, bicycles or scooters) to reach a pickup or dropoff location (meeting point) that causes less delay for a vehicle, may be shared with other customers, and may alleviate concerns of privacy for riders. This acts as a first step towards a hierarchy of personal transportation consisting of local transportation, taxi sharing, and public transit, promising economical and ecological benefits compared to current transportation systems.

Our starting point is the dynamic taxi sharing dispatcher by Buchhold et al. [11]. It uses one-to-many routing based on bucket contraction hierarchies (BCHs) to efficiently compute the best feasible assignments of riders to vehicles. This is a crucial step for handling large fleets in real time and computing realistic simulations of such systems in transportation research.

We introduce the KaRRi (Karlsruhe Rapid Ridesharing) algorithm that extends the dispatcher with the possibility of performing the pickup and dropoff of a rider not at fixed locations but at meeting points which can be any location close to the rider’s origin and destination. The algorithm computes optimal assignments of riders to vehicles including locations for the pickup and dropoff. We adapt the dispatcher’s objective function to this new scenario by incorporating rider trip times and overheads for local travel to and from meeting points.

Finding not only the best vehicle for a request but an optimal combination of a vehicle, a pickup location, and a dropoff location leads to a much larger number of possible assignments. To determine the best assignment, we need to solve a number of many-to-many routing problems between vehicle locations and all possible meeting points. We use BCH queries to address this issue and propose novel speedup techniques both for general purpose bucket based queries and for the specific case of localized sources or targets. We find that these techniques are also applicable for faster routing in a scenario without meeting points.

Our experimental evaluation uses realistic data sets to evaluate the efficiency of these measures. In a scenario without meeting points, our implementation is several times faster than the state-of-the-art dispatcher. For multiple meeting points, our routing techniques are up to
three orders of magnitude faster than a naïve extension of previous techniques. We also give first indications that meeting points can reduce the operating costs of a taxi fleet by up to 15% without increasing rider wait times or trip times. A closer investigation of possible effects on the transport system is left to future work likely in cooperation with application experts.

1.1 Related Work. Taxi sharing and related problems are well studied in transportation research. We summarize existing solution approaches and research into the effect of meeting points on such systems.

Taxi Sharing. Taxi sharing (also called ride pooling) is the problem of dispatching rider requests asking to go from an origin to a destination location to a fleet of taxi-like vehicles while adhering to rider constraints like a latest possible arrival time. The goal is to find assignments of riders to vehicles that optimize an objective function such as the total vehicle operation time.

Taxi sharing can be seen as a special case of the well studied Dial-a-Ride problem (DARP) [13, 52]. Most research on taxi sharing deals with the static variant of the problem where all rider requests are known in advance, including their individual time constraints. The static problem is known to be NP-complete [50, 61]. Small problem instances can be solved optimally using integer programming [12, 5, 35, 13]. Other solutions sacrifice optimality for better performance using meta-heuristics like simulated annealing [48, 41], GRASP [60], or the artificial bee colony algorithm [69].

We study the dynamic taxi sharing problem. Here, the dispatcher is informed about requests as they come in and has to assign riders to vehicles in that order without knowing future requests. Though there is increasing interest in dynamic ridesharing with stochastic information about future rider demand [54, 62, 62], we stick to the traditional agnostic view [59]. Thus, we are concerned with local decision heuristics that try to find a best assignment for each request, attempting to minimize the negative impact on the global objective function or cost of the chosen assignment. Note that the routing techniques discussed in this paper are also applicable to static and stochastic dispatchers as they, too, need to compute many-to-many shortest path queries.

A lot of work on dynamic taxi sharing focuses on enumerating assignments and assessing their feasibility w.r.t. the riders’ time constraints [59, 53, 37]. For this, the dispatcher needs to know the extent of the vehicle detours made to service the new rider. Oftentimes, these detours are simply assumed to be known [59, 59, 53, 33, 32, 38, 31, 37, 35, 10]. However, finding the shortest paths that comprise the detours in the road network poses a major time overhead and can become a bottleneck for the performance of a taxi sharing dispatcher.

Some recent works acknowledge this overhead by first employing filtering heuristics (e.g., based on geodesic distances [8, 34] or spatial indices [50, 50, 51]) to find a small set of candidate assignments s.t. shortest path queries only have to be executed for these candidates. These heuristic dispatchers use varying shortest path algorithms as a black box, ranging in efficiency from Dijkstra’s algorithm [18] to hub labeling [17].

Buchhold et al. [11] employ a more involved approach by using the time constraints of already assigned riders to prune bucket contraction hierarchy (BCH) searches [23, 26], a state-of-the-art one-to-many shortest path algorithm. This allows the shortest path algorithm itself to act as a filter of feasible assignments, efficiently computing both a set of candidate vehicles that is guaranteed to contain the best assignment and the required shortest paths. The algorithm is also equipped to work with customizable contraction hierarchies [17] which allow for fast readjustment of travel times in the road network caused by changing traffic conditions.

Ride Matching. In taxi sharing, the vehicles’ only purpose is to service riders. In contrast, the closely related ride matching or ride sharing problem assumes that each driver is a private entity with their own origin, destination and time constraints [23, 3, 30].

Ride matching is largely faced with the same challenges as taxi sharing. Static solutions can be found with integer programming [2, 64] and branch-and-bound algorithms [9], or approximated with evolutionary algorithms [30]. Approaches for the dynamic variant include locality-constrained greedy matching algorithms [28, 58] and the application of static solutions for buffered sets of requests [30, 2]. As with taxi sharing, BCHs may be suited to compute vehicle detours [27]. For overviews on ride matching, we refer to [23] and [3].

Meeting Points in Taxi Sharing. Meeting points allow riders to be picked up and dropped off at locations close to their origin and destination, respectively. This requires the rider to walk a small distance but potentially reduces the cost of an assignment.

Taxi sharing with meeting points has started garnering attention only recently. Most works that we are aware of focus on the positive effects of meeting points on the operation costs and service quality of dispatchers.

For this purpose, Fielbaum et al. [22] and Mounesan et al. [50] independently extend a previous ILP formulation of the static taxi sharing problem [5] with meeting points and evaluate their impact in experiments on the road network of Manhattan. Meeting points are found to increase the rate of requests that can be serviced within certain wait and trip time limits, while simultaneously decreasing the total vehicle operation time.
Lotze et al. [49] explore stop pooling, a restricted form of meeting points, for a simple dynamic taxi sharing model using requests distributed in a euclidean plane. The authors find that stop pooling reduces both the vehicle operation times and the rider trip times, breaking a traditional trade-off between the two.

Mounesan et al. [50] focus not only on the impact of meeting points on the quality of a dispatcher but also consider the scalability for larger realistic inputs. The authors develop the dynamic taxi sharing dispatcher STaRS+ that extends STaRS [57] with meeting points. Using a distance cache for pre-computed all-pairs shortest path distances, STaRS+ is able to answer a request on the road network of all five boroughs of New York City in about 10ms with a fleet of 10000 vehicles.

All works mentioned here make significant simplifications to the taxi sharing model or the dispatching process with meeting points to maintain feasible running times. Most importantly, shortest path distances are generally pre-computed which hinders the scalability and flexibility of the dispatchers. We detail how these simplifications affect existing approaches, particularly our most direct competitor STaRS+, in the full paper [40].

Meeting Points in Ride Matching. Several publications have studied meeting points on the closely related ride matching problem and have found a positive impact on the quality of matches [4] [63].

Li et al. [37] show that it is NP-hard to find optimal meeting points for a set of ride matching requests even when considering only a single vehicle. The authors present multiple dynamic programming based solution algorithms for a slightly relaxed problem 10000.

Goel et al. [29] [28] show that meeting points can also be used to facilitate privacy-aware ride matching. They choose a set of potential meeting points that cover the road network in such a way that any rider can communicate a small subset of meeting points close to their origin location to the driver without allowing them to identify the rider’s true origin location.

1.2 Paper Overview. After a more detailed problem statement in section 2, we introduce basic notation and techniques in section 3. Section 4 describes the KaRRi algorithm and section 5 evaluates it experimentally.

2 Problem Statement

This section describes the formal foundations for the dynamic taxi sharing problem considered by our approach.

Road Network. We consider a road network to be a directed graph \( G = (V, E) \) where edges represent road segments and vertices represent intersections. Every edge \( e = (v, w) \in E \) has a travel time \( \ell(e) = \ell(v, w) \). We denote the shortest path distance (i.e. travel time) from a vertex \( v \) to a vertex \( w \) by \( \delta(v, w) \).

Vehicle, Stop. Our algorithm has access to a fleet \( F \) of vehicles. The current route \( R(\nu) = \langle s_0(\nu), \ldots, s_k(\nu) \rangle \) of a vehicle \( \nu \) is a sequence of stops scheduled for the vehicle. The vehicle’s current location is always somewhere between its previous (or current) stop \( s_0(\nu) \) and its next stop \( s_1(\nu) \). We update the routes accordingly as vehicles reach stops or are assigned new stops. Thus, \( k(\nu) = |R(\nu)| - 1 \) is the number of stops that the vehicle yet has to visit. Each stop \( s \) is mapped to a vertex \( loc(s) \in V \) in the graph. Given a sufficiently clear context, we may write \( s_i \) instead of \( s_i(\nu) \) and only \( s_i \) instead of \( loc(s_i) \).

Request. In our scenario, the dispatcher receives ride requests and immediately assigns them to vehicles. A request \( r = (orig, dest, t_{req}) \) has an origin location \( orig \in V \), a destination location \( dest \in V \) and a time \( t_{req} \) at which the request is issued. We do not allow pre-booking, i.e. the request time is also the earliest possible departure time.

Meeting Points. We assume that riders can reach meeting points in their vicinity using local transportation such as walking or cycling. We represent the paths accessible to this mode of transportation in a road network \( G_{psg} = (V_{psg}, E_{psg}) \). For any request \( r \), any two subsets of \( V_{psg} \cap V \) can be chosen as the sets of potential pickup and dropoff locations for \( r \).

We use a set of pickup locations (or pickups) \( P(\rho)(r) \) that contains all eligible vertices that the rider can reach in \( G_{psg} \) from \( orig(\nu) \) within a time radius \( \rho \). Analogously, our default set of dropoff locations (or dropoffs) \( D(\rho)(r) \) contains all eligible vertices from which the rider can reach \( dest(\nu) \) within \( \rho \). We collectively refer to the pickups and dropoffs of \( r \) as the meeting points of \( r \). Let \( N^p(\rho)(r) = |P(\rho)(r)| \) and \( N^d(\rho)(r) = |D(\rho)(r)| \). We call a pair of pickup and dropoff a PD-pair and the distance between a pickup and a dropoff a PD-distance. If the context allows it, we omit \( r \) in the notation of the terms defined above. The radius \( \rho \) is a model parameter. For the sake of simplicity, we use the same \( \rho \) for every request but the model also permits varying \( \rho \) with each request.

Insertion. For each request \( r \), our dispatcher finds an insertion of a pickup and dropoff of \( r \) into any vehicle’s route \( s.t. \) the cost of that insertion according to a cost function is minimized. We formalize an insertion as a tuple \( (r, p, d, \nu, i, j) \) indicating that vehicle \( \nu \) picks up request \( r \) at pickup location \( p \in P(\rho) \) immediately after stop \( s_i \) and drops off \( r \) at dropoff location \( d \in D(\rho) \) immediately after stop \( s_j \) with \( 0 \leq i \leq j \leq k(\nu) \).

2.1 Cost Function and Constraints. The cost \( c(\nu) \) of an insertion \( i = (r, p, d, \nu, i, j) \) represents the associated vehicle operation cost and the rider service
quality in a linear combination of the form
\[
c(t) = t_{\text{detour}}(t) + \tau \cdot (t_{\text{trip}}(t) + t_{\text{trip}}^+(t)) + \omega \cdot t_{\text{walk}}(t) + c_{\text{wait}}(t) + c_{\text{trip}}(t).
\] (2.1)

Here, the added vehicle operation time \( t_{\text{detour}}(t) \) describes the time that vehicle \( \nu \) needs for the detour it makes to accommodate the pickup at \( p \) and dropoff at \( d \) in its route. The trip time \( t_{\text{trip}}(t) \) denotes the time that passes between the issuing of request \( r \) (\( t_{\text{req}}(r) \)) and the arrival of the rider at their destination \( \text{dest}(r) \), including waiting and walking times. The detours made by \( \nu \) may increase this trip time for existing riders of \( \nu \). The added trip time \( t_{\text{trip}}^+(t) \) is the sum of these increases for all affected riders. The walking time \( t_{\text{walk}}(t) \) represents how long the rider needs to move from their origin to their destination. We consider a total of four constraints for all affected riders. The waiting time \( c_{\text{wait}}(t) \) and \( c_{\text{trip}}(t) \) describe the time that vehicle \( \nu \) needs for the detour to pick up the ride at its pickup stop within a maximum wait time \( t_{\text{wait}}^{\text{max}} \). Fourth, every rider \( \hat{r} \) already assigned to \( \nu \) must still reach its last stop before a fixed time \( \text{dep} \). The direct vehicle travel time from the origin to the destination of \( \hat{r} \). The values \( t_{\text{wait}}^{\text{max}}, \alpha \) and \( \beta \) are model parameters.

All four constraints are hard constraints w.r.t. requests already assigned to \( \nu \). If \( r \) breaks a hard constraint, we set the cost to \( \infty \). For the request \( r \) to be inserted, we treat the wait time and trip time constraints as soft constraints, i.e., violating them leads to the cost penalties \( c_{\text{wait}}(t) \) and \( c_{\text{trip}}(t) \). Assume, the rider is picked up at \( p \) at time \( t_{\text{dep}} \). We define
\[
c_{\text{wait}}(t) = \gamma_{\text{wait}} \cdot \max\{t_{\text{dep}} - t_{\text{req}}(r) - t_{\text{wait}}^{\text{max}}, 0\},
\]
\[
c_{\text{trip}}(t) = \gamma_{\text{trip}} \cdot \max\{t_{\text{trip}}(t) - t_{\text{wait}}^{\text{max}}(r), 0\}
\]
with model parameters \( \gamma_{\text{wait}} \) and \( \gamma_{\text{trip}} \) that scale the severity of the penalties.

For formal definitions of the terms used in our cost function, we refer to the full paper [46].

3 Preliminaries

In this section, we describe several shortest path algorithms used in this work. Furthermore, we summarize the dynamic taxi sharing dispatcher introduced by Buchhold et al. [1] that serves as the basis of our work.

3.1 Shortest Path Algorithms. In the following, we explain a number of algorithms that compute different variants of shortest path queries on road networks.

Dijkstra’s Shortest Path Algorithm. Dijkstra’s shortest path algorithm [13] computes the shortest path from a source \( s \in V \) to all other vertices in a weighted graph \( G = (V, E, \ell) \).

The algorithm stores a distance label \( \hat{\delta}(s, v) \) for every \( v \in V \). An addressable priority queue \( Q \) with key(\( v \)) = \( \hat{\delta}(s, v) \) contains active vertices. Initially, \( Q := \{s\} \), \( \hat{\delta}(s, s) := 0 \) and \( \hat{\delta}(s, v) := \infty \) for \( v \neq s \).

The algorithm repeatedly extracts the vertex with the smallest distance label from \( Q \) and settles it. To settle \( u \in V \), each outgoing edge \( (u, v) \in E \) is relaxed by trying to improve the distance label \( \hat{\delta}(s, v) \) with \( \hat{\delta}(s, u) + \ell(e) \). If the distance is improved, \( v \) is inserted into \( Q \). The algorithm stops when \( Q \) becomes empty.

Contraction Hierarchies. Contraction Hierarchies (CHs) [26] speed-up shortest path computations by exploiting the hierarchical nature of road networks. A CH is constructed in a pre-processing phase. Then, shortest path queries can be computed on the CH using restricted Dijkstra searches.

To construct a CH, all vertices in a road network \( G = (V, E) \) are ordered heuristically by their importance or rank [29]. Vertices are contracted in the order of increasing rank. The contraction of \( v \in V \) temporarily removes \( v \) from the graph. To preserve shortest paths, a shortcut edge \( (u, v, w) \) is created if \( (u, v, w) \in E \) is the only shortest path between \( u \) and \( w \).

Let \( E^+ \) contain all original edges \( E \) as well as all shortcut edges. The graph \( G^+ = (V, E^+) \) constitutes the CH. The length \( \ell^+(e) \) of a shortcut edge \( e \) is the sum of the lengths of replaced original edges while \( \delta^+ \) is the accounting distance function. For the query phase, we partition \( E^+ \) into up-edges \( E^+ = \{(u, v) \in E^+ | \text{rank}(u) < \text{rank}(v)\} \) and down-edges \( E^+_d = \{(u, v) \in E^+ | \text{rank}(u) > \text{rank}(v)\} \).

We define an upwards search graph \( G^+_u := (V, E^+_u) \) and a downwards search graph \( G^+_d := (V, E^+_d) \). The distances \( \delta^+ \) and \( \delta^d \) represent \( \delta^+ \) constrained to \( G^+ \) and \( G^d \).

For any two vertices \( s, t \in V \), it can be shown that there is a shortest path from \( s \) to \( t \) that is an up-down path in the CH, i.e. consists of only up-edges followed by only down-edges [29]. A CH query from a source \( s \in V \) to a target \( t \in V \) runs a forward Dijkstra search from \( s \) in \( G^+ \) and a reverse Dijkstra search from \( t \) in \( G^d \). Whenever the searches meet, they find an up-down-path from \( s \) to \( t \), eventually finding a shortest path. The query can stop once the radius of either Dijkstra search
exceeds the best previously found distance from \( s \) to \( t \).

**Bucket Contraction Hierarchy Searches.** Bucket Contraction Hierarchy (BCH) searches [13, 20] find all shortest path distances from a set of sources \( S \subseteq V \) to a target \( t \in V \) in a road network \( G = (V, E) \). A CH \( G^+ \) of \( G \) is used as the basis of the algorithm.

The idea is to construct a *(source)* bucket \( B^+(v) \) at each vertex \( v \in V \). Conceptually, \( B^+(v) \) is a list of entries, each one of which stores the upwards distance from one of the sources to \( v \). For each source \( s \in S \), a forward search in \( G^+ \) is run that adds an entry \((s, \delta^+(s, v))\) to \( B^+(v) \) for every settled \( v \in V \). Then, a reverse search from \( t \) traverses \( G^+ \) and finds shortest up-down paths between \( s, s_2 \) and \( t \) by scanning \( B^+(u) \) and \( B^+(v) \).

**Bundled Searches.** Dijkstra-based shortest path algorithms for multiple sources can be bundled s.t. the searches for \( k \) sources are advanced simultaneously. A bundled search maintains \( k \) tentative distance labels at each vertex. When the search relaxes an edge \((u, v) \in E\), it tries to update all \( k \) distance labels at \( v \).

A bundled relaxation can be more cache efficient than \( k \) individual relaxations as all \( k \) distances are stored in consecutive memory. However, the relaxation of \((u, v) \in E\) may perform unproductive work if not all \( k \) searches have reached \( u \) yet. Thus, bundling is effective if all \( k \) searches relax largely the same edges. The value of \( k \) is a tuning parameter.

Bundled searches were first introduced for Dijkstra searches used for the computation of arc-flags under the name *centralized searches* [31]. Since then, bundled searches have been used in a number of Dijkstra-based shortest path algorithms [7, 67, 14, 15, 16], including point-to-point queries in CHs [10].

**SIMD Parallelism in Bundled Searches.** Bundled searches can be sped up substantially using single-instruction multiple-data (SIMD) parallelism [10]. Modern CPUs provide special vector registers and instructions that can store and manipulate multiple data items at once. We can vectorize the computations needed during edge relaxations s.t. \( k \) computations are performed at the same time using a single vector instruction.

### 3.2 LOUD

Our algorithm is based on the dynamic taxi sharing dispatching algorithm LOUD [11].

Given a fleet of vehicles and routes, the online algorithm matches incoming taxi sharing requests to vehicles. For each request, a feasible insertion of the request’s origin \( o \) and destination \( d \) into a vehicle’s route is found s.t. the detour of the vehicle is minimized.

**Elliptic Pruning.** To compute the costs of possible insertions, the algorithm requires the distances between existing vehicle stops and \( o \) and \( d \). LOUD computes these distances using BCHs with bucket entries for each vehicle stop and queries run from \( o \) and \( d \).

We refer to these BCH searches as *elliptic BCH searches* since they utilize a pruning technique for these buckets called *elliptic pruning*: Each insertion is subject to the same soft and hard constraints that we describe in section 2.1. The wait time and trip time hard constraints of riders already assigned to a vehicle \( v \in F \) define a *leeway* \( \lambda(s_i, s_{i+1}) \), i.e., a maximum permissible detour, between each pair of consecutive stops \((s_i, s_{i+1}) \in R(v)\). Any detour that exceeds \( \lambda(s_i, s_{i+1}) \) breaks some hard constraint and is infeasible. The leeway \( \lambda(s_i, s_{i+1}) \) defines a detour ellipse that contains all vertices at which a pickup or dropoff may be made between \( s_i \) and \( s_{i+1} \) without breaking a hard constraint. Thus, bucket entries for \( s_i \) and \( s_{i+1} \) only need to be generated at vertices within the ellipse. Elliptic pruning vastly reduces the number of bucket entries that need to be scanned by the BCH searches and limits the number of candidate vehicles for insertions [11].

**Last Stop Distances.** LOUD also allows the insertion of the origin and/or destination after the last stop of a vehicle’s route. Here, elliptic pruning is not applicable since the leeway of any vehicle is unbounded after its last stop. Instead, LOUD uses reverse Dijkstra queries in the road network rooted at \( o \) or \( d \) to find the distances from last stops to \( o \) or \( d \). These Dijkstra queries, particularly for the destination of a request,
constitute a significant part (at least 60% and up to more than 90%) of the total running time of LOUD.

4 The Algorithm

We introduce the KaRRi algorithm that efficiently answers taxi sharing requests with multiple meeting points using fast many-to-many routing.

4.1 Algorithm Outline. The KaRRi algorithm dynamically accepts requests and finds an insertion for each request that has optimal cost according to the cost function and current system state.

For a request \( r \), the algorithm first finds the possible meeting points in a walking radius \( \rho \) around the origin and destination using bounded Dijkstra searches. Then, the algorithm evaluates all insertions in the order of types illustrated in fig. 2. For each insertion, KaRRi computes the cost according to the cost function (see section 2.1). The insertion with the smallest cost \( c^* \) is repeatedly updated and eventually returned.

Since we consider sets of possible meeting points, the number of potential insertions becomes the main challenge of the algorithm. In particular, we face the issue of computing the shortest paths between existing vehicle stops and every meeting point to filter out infeasible insertions and to determine the cost of the remaining candidate insertions. In the following, we describe the bundling and filtering methods that we employ to limit the running time of the required many-to-many shortest path queries.

4.2 Many-to-Many Routing Techniques. We illustrate our contributions for pickup after last stop (PALS) searches, i.e., the problem of finding the distances between last stops and pickups. This particular many-to-many shortest path problem serves as a good overview since all of our techniques can be applied. We explain how we can apply the techniques for the remaining shortest path computations in the next section. For the rest of this section, let \( c \) denote an upper bound on the cost of the best insertion of \( r \). For example, this can be the cost \( c(\tau^*) \) of the best insertion \( \tau^* \) seen so far.

Bundled Searches with Localized Sources. Buchhold et al. [11] find the distances from last stops to pickups using a reverse Dijkstra search rooted at the request’s origin. The search is stopped when the smallest distance to any active vertex along with a lower bound on the PD-distance no longer admit a PALS insertion with smaller cost than \( c \). We can extend this technique to multiple pickups by analogously running reverse Dijkstra searches for each pickup.

We find that bundled searches (see section 3.1) work particularly well here. Since the pickups are localized, every individual search has highly similar search trees. This is further reinforced due to the hierarchical nature of road networks. Thus, each individual search performs largely the same edge relaxations which allows effective bundling of these relaxations. Additionally, bundled searches can parallelize edge relaxations with SIMD instructions. These advantages of bundled searches are not limited to Dijkstra searches for PALS distances but hold true for any shortest path queries that need to be repeated for every meeting point.

BCH Searches with Sorted Buckets. Since Dijkstra’s algorithm is inefficient for road networks, we introduce an approach for the computation of PALS distances with bucket contraction hierarchies (BCHs).

For this, we maintain a bucket \( B^T(v) \) for every \( v \in V \). For every last stop \( s_k(v) \), we generate an entry \( (s_k(v), \delta^T(s_k(v), v)) \in B^T(v) \) at each vertex \( v \) in the upward CH search space rooted at \( s_k(v) \). Then, for every pickup \( p \in P_r \), we run an individual (last stop) BCH query in the reverse CH search space rooted at \( p \) that scans the bucket at each settled vertex to compute the shortest path distances from last stops to \( p \). When the search scans an entry \( (s_k(v), \delta^T(s_k(v), v)) \), it tries to improve the tentative distance \( \delta(s_k(v), p) \) with \( \delta^T(s_k(v), v) + \delta(v, p) \). Eventually, the shortest distance \( \delta(s_k(v), p) \) is found for every last stop \( s_k(v) \). Both edge relaxations and bucket scans can be bundled to run \( k \) searches simultaneously. Similarly to Dijkstra searches, we apply a cost-based stopping criterion: As soon as the smallest distance to any active vertex along with a lower bound on the PD-distance no longer admit an insertion with smaller cost than \( c \), the search is stopped.

A yet-unmentioned issue of this approach is the fact that elliptic pruning is not applicable since there are no time constraints of existing riders after the last stop that would define a detour leeway (cf. section 3.2). Thus, the number of entries per bucket may grow very large, especially at vertices that have a high rank in the CH. In effect, a lot of time may be spent on scanning bucket entries and every vehicle has to be considered when enumerating PALS insertions after the queries.

To address this issue, the future work section of [11] suggests sorting the entries within each last stop bucket to be able to stop each bucket scan early. For each \( v \in V \), we sort the entries in \( B^T(v) \) by their distance \( \delta^T(s_k(v), v) \) in increasing order. Suppose a pickup query rooted at \( p \in P_r \) scans the bucket \( B^T(v) \). For each entry \( e = (s_k(v), \delta^T(s_k(v), v)) \in B^T(v) \), we can compute a vehicle-independent lower bound \( c_{\min}(e) \) on the cost of any PALS insertion \( \tau \) where the vehicle drives a distance of at least \( \delta^T(s_k(v), v) + \delta(v, p) \) to \( p \). Since \( c_{\min}(e) \) then increases monotonously with \( \delta^T(s_k(v), v) \), we can stop...
scanning the sorted bucket as soon as we read an entry with \(c_{\min}(e) > \hat{c}\).

Sorted buckets can reduce the number of entries scanned for any BCH searches where an upper bound on the required distance is known.

**Collective Last Stop Searches.** Note that we do not actually need to know the distance between every last stop and every pickup. If we knew the best PALS insertion \(i^*_\text{pals} = (r, p, d, \nu, k, \nu, k(\nu))\) already, we would only need to find \(\delta(s_{k(\nu)}, p)\). We propose a collective BCH search that directly finds \(i^*_\text{pals}\) by propagating labels that represent PD-pairs through the search graph and pruning labels by comparing them to one another.

A label \((p, d, \delta^i(v, p))\) at a vertex \(v \in V\) consists of the pickup \(p \in P_v\), dropoff \(d \in D_v\) and downwards distance \(\delta^i(v, p)\). For each label \(l\), we store a lower bound \(c_{\min}(l)\) for the cost of any PALS insertion that can be found for \(l\) in the search sub-tree rooted at \(v\).

Our search maintains a priority queue \(Q\) that contains all active labels ordered increasingly by \(c_{\min}\). Initially, at each pickup \(p \in P_v\), an open label \((p, d, 0)\) is created for each \(d \in D_v\). As long as \(Q\) contains a label \(l\) with \(c_{\min}(l) < \hat{c}\), our search proceeds with a next step. In each step of the search, the label \(l := \min(Q)\) is removed from \(Q\) and settled.

Settling a label \(l = (p, d, \delta^i(v, p))\) consists of two steps: First, we search for a new best insertion by traversing all entries in the last stop bucket \(B^i(v)\) for \(l\). For each entry \(e = (s_{k(\nu)}, \delta^i(s_{k(\nu)}, v)) \in B^i(v)\), we get a tentative distance \(\delta(s_{k(\nu)}, p) = \delta^i(s_{k(\nu)}, v) + \delta^i(v, p)\).

With this tentative distance, we compute an upper bound \(c_{\max}(l, e)\) on \(c(i)\) for \(i = (r, p, d, \nu, k(\nu), k(\nu))\). If \(c_{\max}(l, e) < \hat{c}\), we mark \(i\) as the best seen PALS insertion and update \(\hat{c} := c_{\max}(l, e)\). As we have full information except for the last stop distance, we can ensure that \(c_{\max}(l, e) = c(i)\) if the tentative distance is exact. Thus, we eventually find the best PALS insertion since our BCH search finds shortest up-down-paths. We still use sorted buckets so bucket scans may be stopped early. Second, we propagate \(l\) to each neighboring vertex \(w\) of \(v\) where we create a new open label \(l' = (p, d, \delta^i(w, v) + \delta^i(v, p))\). We discard \(l'\) if \(c_{\min}(l') > \hat{c}\).

The central idea of collective searches is that we can additionally prune \(l'\) if it is dominated by any of the open or already settled labels at \(w\). Intuitively, a label \(l\) dominates a label \(l'\) at a vertex \(v\) if any vehicle that drives via \(v\) for a PALS insertion should perform the pickup and dropoff at \(p\) and \(d\) instead of \(p'\) and \(d'\) to minimize the cost of the insertion.

To formalize this, we define an upper bound for the cost of a PALS insertion that can be found for \(l = (p, d, \delta^i(v, p))\) in the search sub-tree rooted at \(v\). Let \(G^i_v = (V^i_v, E^i_v)\) denote the sub-graph of \(G^i\) consisting of all paths to \(v\). Consider a vertex \(w \in V^i_v\) and an entry \(e = (s_{k(\nu)}, s_{k(\nu)}(v), w) \in B^i(w)\). Let \(c_{\max}(l, v, e) := c((r, p, d, \nu, k(\nu), k(\nu)))\) under the assumption \(\delta(s_{k(\nu)}, p) = \delta^i(s_{k(\nu)}, v) + \delta^i(v, p)\).

**Definition 4.1.** A label \(l\) dominates another label \(l'\) at a vertex \(v \in V\) exactly if \(c_{\max}(l, v, e) < c_{\max}(l', v, e)\) for every \(w \in V^i_v\) and \(e \in B^i(w)\).

**Theorem 4.1.** If a label \(l\) dominates another label \(l'\) at \(v\), we do not need to settle \(l'\) at \(v\).

Proof. Omitted for brevity. See full paper [46].
we can bundle elliptic BCH searches.
bundled Dijkstra or BCH searches, or a collective BCH
every dropoff. Similar to the PALS case, we can use
BCHs to find these distances for the remaining insertions.
introduced by Buchhold et al. Then, we use bundled
location to the pickup. We filter out the vast majority
need to know the distance from the vehicle’s current
the searches for generating and scanning entries.
remaining leeway
bucket by the
number of scanned bucket entries by sorting each source
approaches can be found in the full paper [46].
approximation of the domination relation in constant
time constraint, we refer to the full paper [46].

4.3 Applying Routing Techniques to Other Insertion Types. We briefly explain how bundled searches, sorted buckets, and collective BCH searches can be applied for shortest path queries needed for the other insertion types (see fig. 2). More details on these approaches can be found in the full paper [46].

Elliptic pruning enables BCH searches to efficiently compute the shortest paths to and from stops that are not last stops (see section 3.2). We can further reduce the number of scanned bucket entries by sorting each source bucket by the remaining leeway \( \lambda(s_i, s_{i+1}) - \delta^T(s_i, v) \) of each entry \((s_i, \delta^T(s_i, v))\) (analogous for target buckets), allowing us to stop each bucket scan early. Additionally, we can bundle elliptic BCH searches.

For paired insertions, we compute the distance between every pickup and dropoff using BCH searches. We generate bucket entries for dropoffs and scan the entries in queries rooted at the pickups. We can bundle the searches for generating and scanning entries.

For a pickup before next stop (PBNS) insertion, we need to know the distance from the vehicle’s current location to the pickup. We filter out the vast majority of PBNS insertions based on a cost lower bound as introduced by Buchhold et al. [11]. Then, we use bundled BCHs to find these distances for the remaining insertions.

For dropoff after last stop (DALS) insertions, we need to compute the distances from every last stop to every dropoff. Similar to the PALS case, we can use bundled Dijkstra or BCH searches, or a collective BCH search with domination pruning.

5 Experimental Evaluation
Our source code\(^1\) is written in C++17 and compiled with GCC 9.4 using -O3. We run our experiments on a machine with Ubuntu 20.04, 512 GiB of memory and two 16-core Intel Xeon E5-2683 v4 processors at 2.1GHz. 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 KaRRi on the Berlin-1pct (B-1%) and Berlin-10pct (B-10%) request sets \([11]\) that represent 1% and 10% of taxi sharing demand in the Berlin metropolitan area on a weekday. The request sets were artificially generated based on the Open Berlin Scenario \([71]\) for the MATSim transport simulation \([34]\). The underlying road networks are obtained from OpenStreetMap data\(^2\) and CHs are computed using the open-source library RoutingKit\(^3\). The sizes of the instances are shown in table \(1\). We consider walking as a mode of local transportation for riders. We scale the number of pickups \(N_p^\rho\) and dropoffs \(N_d^\rho\) by using increasing walking radii \(\rho \in \{0s, 150s, 300s, 450s, 600s\}\) which lead to rough averages of \(N_p^\rho \approx N_d^\rho \in \{1, 12, 44, 100, 180\}\) for our instances. We run five iterations of every experiment, and report average running times.

For our cost function (see eq. (2.1)), we adopt a basic “time is money” approach. We use \(\tau = 1\) to weight the time of a driver and a rider equally. By setting \(\omega = 0\) we do not penalize walking over driving. This choice maximizes the effect of meeting points on vehicle detours. In accordance with the MATSim transport simulation, we choose \(\alpha = 1.7\) and \(\beta = 2\) min which means that each trip may take up to a maximum trip time of \(1.7\delta(\text{orig, dest}) + 2\) min. For the remaining parameters, we choose \(t_{\text{wait}}^{\text{max}} = 600\) s, \(\gamma_{\text{wait}} = 1\), and \(\gamma_{\text{trip}} = 10\).

5.1 Effectiveness of Many-to-Many Routing Techniques. We evaluate the effectiveness of the proposed routing techniques for the last stop searches used for the PALS and DALS insertion types. We show the results for elliptic BCH searches and PD-distance searches in the full paper \([46]\).

**Bundled Searches.** We experimentally evaluate the impact of bundling Dijkstra and individual BCH searches on the Berlin-1pct and Berlin-10pct instances with \(\rho \in \{150s, 300s, 450s, 600s\}\). We depict the speedups observed for bundled searches for the PALS and DALS cases in fig. \(3\). In preliminary experiments,

\[c_{\text{max}}(l, v, e) = \delta^T(s_{\text{pick}}, w) + \delta^T(w, v)\]

| Instance | \(|V|\) | \(|E|\) | #veh. | #req. |
|----------|--------|--------|-------|-------|
| B-1%     | 94422  | 193212 | 1000  | 16569 |
| B-10%    | 94422  | 193212 | 10000 | 149185 |

\[^{1}\text{Available at https://github.com/molaupi/karri}\]
\[^{2}\text{MATSim generates realistic demand data but considering more than 10% of taxi sharing demand would take processing times in the order of multiple months. For details, see [11].}\]
\[^{3}\text{https://download.geofabrik.de/} \text{accessed Oct 30th 2023.}\]
\[^{4}\text{https://github.com/RoutingKit/} \text{accessed Oct 30th 2023.}\]
With sorted buckets, more bucket entries are scanned without vector instructions are slower than non-bundled well suited for bundling.

In effect, most work will run into cache limitations as hundreds of bytes of whether SIMD instructions are used. For details, we performed by individual last stop BCH searches is not optimal, depending on the input instance and whether SIMD instructions are used. For details, we refer to the full paper [46].

We find that Dijkstra searches are well suited for bundling. Since Dijkstra searches do not use shortcut edges, the searches for each individual source meet much earlier than BCH searches. Thus, the vast majority of the large number of edge relaxations of Dijkstra searches can be bundled well. This is evidenced by the fact that bundled Dijkstra searches experience good speedups of up to 3.09 in the PALS case and 5.96 in the DALS case even without SIMD instructions. Using SIMD instructions, we can improve these speedups to up to 7.60 and 19.71. Even larger values of $k > 64$ may be useful for larger numbers of sources but eventually we will run into cache limitations as hundreds of bytes of distance labels need to be handled per vertex.

Contrarily, individual last stop BCH searches cannot be bundled as well due to two opposing properties: Firstly, most work is performed close to the sources. With sorted buckets, more bucket entries are scanned at vertices closer to the sources. Additionally, the cost based stopping criterion of last stop BCH searches limits the search radius. Secondly, due to the usage of shortcut edges, the search trees of individual searches only overlap at larger distances from the sources. Thus, edge relaxations and bucket entry scans cannot be bundled well in the proximity of the sources. In effect, most work performed by individual last stop BCH searches is not well suited for bundling.

In the PALS case, we see speedups of only up to 2.04 with SIMD instructions. In fact, bundled searches without vector instructions are slower than non-bundled searches in the PALS case. Speedups are better in the DALS case as the searches explore a larger search radius due to the fact that the cost based stopping criterion cannot take any information about the pickup into account. In the DALS case, bundling with SIMD instructions achieves speedups of up to 3.90.

Sorted Buckets. In the following, we analyze the effect of sorted buckets on individual and collective last stop BCH searches. We experimentally evaluate both searches with sorted and unsorted buckets on the Berlin-1pct and Berlin-10pct instances with $\rho \in \{0s, 150s, 300s, 450s, 600s\}$ and $k = 1$. The speedups achieved with sorted buckets are shown in fig. [14].

For last stop BCH searches, sorted buckets are vital to reduce the number of bucket entries scanned since we cannot use elliptic pruning. For individual BCH searches, more than 97% and 89% fewer bucket entries are scanned with sorted buckets in the PALS and DALS cases, respectively. This reduces search times by factors of up to 9.09 and 7.14.

For collective searches, the number of bucket entries scanned decreases by similar rates of 97% and 87%. However, the resulting speedups are less pronounced, particularly for larger numbers of meeting points in the PALS case. We attribute this to the fact that collective searches spend comparatively more time on pruning the searches. This means that the searches need to spend less time scanning bucket entries, which limits the impact of sorted buckets. Notably, collective PALS searches generate initial labels for every PD-pair but prune almost all of them immediately. As the number of PD-pairs is proportional to $\rho^4$, this initialization can constitute up to 85% of the search time for larger values of $\rho$ but sorted buckets have no effect on it.

Consequently, we observe speedups of only 1.96
We use the optimal configuration for each combination of candidate vehicle, pickup, and dropoff. As the number of PD-pairs is proportional to \(\rho^4\), enumeration times of individual BCH searches quickly become very large with tens to hundreds of thousands of insertions tried.

For a similar reason, collective searches scale worse in the PALS case than in the DALS case. In the PALS and DALS cases, one initial label is generated for every PD-pair and every dropoff, respectively. Thus, the number of initial labels increases much stronger with growing \(\rho\) in the PALS case. As stated before, with large \(\rho\), the majority of the running time of collective PALS searches is spent on generating and pruning initial labels.

5.2 Comparison with Baseline Dispatcher. In this section, we compare KaRRi with the baseline dispatcher by Buchhold et al. [1].

Running Times. We give the running times for the different phases of both our algorithm (K) and the baseline (B) on B-1\% and B-10\% in Table 3.

First, we consider the scenario without meeting points (\(\rho = 0s\)) and compare KaRRi with the baseline dispatcher. Here, sorted buckets have no positive impact on the search times of elliptic BCH searches even though the number of bucket entries scanned is reduced. We
Table 2: Comparison of the PALS and DALS running times (in µs) of collective BCH searches (Coll.), individual BCH searches (BCH), and Dijkstra searches (Dij.) in their optimal configurations for three radii $\rho \in \{600s, 300s, 600s\}$ on the B-1% and B-10% instances. Shows average number of edge relaxations ($#_{rel}$), number of bucket entries scanned ($#_{scans}$), search time ($t_{search}$) and time for enumerating insertions ($t_{enum}$) per request. Bold numbers indicate smallest times per radius.

<table>
<thead>
<tr>
<th>Type</th>
<th>$\rho$</th>
<th>Search</th>
<th>$#_{rel}$</th>
<th>$#_{scans}$</th>
<th>$t_{search}$</th>
<th>$t_{enum}$</th>
<th>$#_{rel}$</th>
<th>$#_{scans}$</th>
<th>$t_{search}$</th>
<th>$t_{enum}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>BCH</td>
<td>40</td>
<td>8</td>
<td>4.96</td>
<td><strong>0.04</strong></td>
<td>19</td>
<td>11</td>
<td>3.89</td>
<td><strong>0.06</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Coll.</td>
<td>412</td>
<td>57</td>
<td><strong>70.16</strong></td>
<td>0.08</td>
<td>168</td>
<td>58</td>
<td><strong>41.95</strong></td>
<td><strong>1.49</strong></td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>BCH</td>
<td>967</td>
<td>274</td>
<td>73.95</td>
<td>44.12</td>
<td>797</td>
<td>1011</td>
<td>103.36</td>
<td>155.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Coll.</td>
<td>4302</td>
<td>17</td>
<td>497.94</td>
<td>40.4</td>
<td>3533</td>
<td>99</td>
<td>433.66</td>
<td>138.16</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>BCH</td>
<td>5555</td>
<td>2514</td>
<td>424.65</td>
<td>812.64</td>
<td>4734</td>
<td>12214</td>
<td>823.72</td>
<td>3475.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Coll.</td>
<td>806</td>
<td>108</td>
<td><strong>286.11</strong></td>
<td>0.09</td>
<td>219</td>
<td>82</td>
<td><strong>213.9</strong></td>
<td><strong>23.90</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Dij.</td>
<td>41092</td>
<td>137</td>
<td>4481.08</td>
<td>812.24</td>
<td>38102</td>
<td>960</td>
<td>4412.38</td>
<td>3098.09</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BCH</td>
<td>967</td>
<td>274</td>
<td>73.95</td>
<td>44.12</td>
<td>797</td>
<td>1011</td>
<td>103.36</td>
<td>155.86</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>Coll.</td>
<td>4302</td>
<td>17</td>
<td>497.94</td>
<td>40.4</td>
<td>3533</td>
<td>99</td>
<td>433.66</td>
<td>138.16</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>BCH</td>
<td>5555</td>
<td>2514</td>
<td>424.65</td>
<td>812.64</td>
<td>4734</td>
<td>12214</td>
<td>823.72</td>
<td>3475.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Coll.</td>
<td>806</td>
<td>108</td>
<td><strong>286.11</strong></td>
<td>0.09</td>
<td>219</td>
<td>82</td>
<td><strong>213.9</strong></td>
<td><strong>23.90</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Dij.</td>
<td>41092</td>
<td>137</td>
<td>4481.08</td>
<td>812.24</td>
<td>38102</td>
<td>960</td>
<td>4412.38</td>
<td>3098.09</td>
</tr>
</tbody>
</table>

attribute this to the fact that our implementation is meant to deal with any number of meeting points while the baseline is specialized for the case of $N^p_0 = N^d_0 = 1$. Our last stop BCH searches are well suited for $\rho = 6$ ms, though. They are up to 14 and 70 times faster than the baseline Dijkstra searches in the PALS and DALS cases. Note that maintaining sorted buckets does lead to increased update times, though. In total, we can reduce the average time per request by factors of almost 5 and 2 for B-1% and B-10% compared to the baseline. In the full paper [46], we show that KaRRi compares even more favorably on other instances that are three times larger.

Unfortunately, the source code of KaRRi’s closest existing competitor STaRS+ [56] is currently not publicly available, making an experimental comparison of both approaches difficult. Instead, we validate the effectiveness of our approach by comparing it with a naive extension of the techniques used by our baseline algorithm. For this, we configured KaRRi to use no bundled searches or sorted buckets, to use Dijkstra searches for the PALS and DALS cases, and to use point-to-point CH queries to compute PD-distances. We report the running times of this extension for $\rho = 300s$ and $\rho = 600s$ on the B-10% instance and compare them to KaRRi.

We find that bundling the elliptic BCH searches and using sorted buckets make them about one order of magnitude faster than the naive extension. PD-distances can be computed around two orders of magnitude faster with our bucket based approach than with individual CH queries. Our collective searches for the PALS and DALS cases beat the naive approach by two and three orders of magnitude, respectively.

We also equipped KaRRi with the possibility to use customizable CHs (CCHs) [17] with elimination tree searches that are well suited for bundling [19]. This allows us to adapt our CH to changed travel times in the road network in less than 100ms at the cost of an increase in KaRRi’s running time of less than 1ms per request on average. Thus, KaRRi with CCHs combines fast dispatching with the ability to react to changing traffic conditions in real time. For a full evaluation of KaRRi with CCHs, we refer to the full paper [46].

Solution Quality. In the following, we summarize how meeting points affect the quality of assignments.
Table 3: Running times (in µs) of the baseline (B), naïvely extended baseline (B*), and KaRRi (K) with \( \rho \in \{0s, 300s, 600s\} \) on B-1% and B-10%. Shows mean times for finding \( P_\rho \) and \( D_\rho \), PD-distance searches, elliptic BCH searches, enumerating ordinary and PBNS insertions, PALS and DALS searches, and updating routes and buckets as well as the mean total time per request.

<table>
<thead>
<tr>
<th>Inst.</th>
<th>( \rho )</th>
<th>Alg.</th>
<th>find ( P_\rho, D_\rho )</th>
<th>PD</th>
<th>Ell. BCH</th>
<th>Ord. &amp; PBNS</th>
<th>PALS</th>
<th>DALS</th>
<th>update</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-1%</td>
<td>0</td>
<td>B</td>
<td>0</td>
<td>21</td>
<td>113</td>
<td>68</td>
<td>55</td>
<td>1682</td>
<td>97</td>
<td>2036</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>K</td>
<td>2</td>
<td>74</td>
<td>115</td>
<td>41</td>
<td>4</td>
<td>24</td>
<td>151</td>
<td>413</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>K</td>
<td>173</td>
<td>300</td>
<td>617</td>
<td>151</td>
<td>72</td>
<td>63</td>
<td>154</td>
<td>1530</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>K</td>
<td>617</td>
<td>1536</td>
<td>2536</td>
<td>881</td>
<td>298</td>
<td>107</td>
<td>155</td>
<td>6129</td>
</tr>
<tr>
<td>B-10%</td>
<td>0</td>
<td>B</td>
<td>0</td>
<td>19</td>
<td>328</td>
<td>247</td>
<td>27</td>
<td>1361</td>
<td>94</td>
<td>2076</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K</td>
<td>3</td>
<td>73</td>
<td>351</td>
<td>346</td>
<td>4</td>
<td>88</td>
<td>245</td>
<td>1111</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>B*</td>
<td>194</td>
<td>30600</td>
<td>20196</td>
<td>905</td>
<td>2275</td>
<td>54270</td>
<td>119</td>
<td>108559</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K</td>
<td>195</td>
<td>308</td>
<td>1770</td>
<td>783</td>
<td>51</td>
<td>138</td>
<td>246</td>
<td>3490</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>B*</td>
<td>716</td>
<td>513788</td>
<td>77813</td>
<td>3313</td>
<td>28901</td>
<td>220555</td>
<td>118</td>
<td>845204</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K</td>
<td>708</td>
<td>1662</td>
<td>6891</td>
<td>3227</td>
<td>312</td>
<td>211</td>
<td>249</td>
<td>13260</td>
</tr>
</tbody>
</table>

Table 4: Solution quality of KaRRi with different radii \( \rho \in \{0s, 300s, 600s\} \) on B-1% and B-10%. For riders, we report the average wait and trip times (in mm:ss). For vehicles, we give the average occupancy while driving and average total operation time (in hh:mm).

<table>
<thead>
<tr>
<th>Inst.</th>
<th>( \rho )</th>
<th>wait</th>
<th>trip</th>
<th>occ</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-1%</td>
<td>0</td>
<td>3:44</td>
<td>16:53</td>
<td>0.88</td>
<td>4:28</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>3:15</td>
<td>15:54</td>
<td>0.93</td>
<td>4:00</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>3:27</td>
<td>16:02</td>
<td>0.94</td>
<td>3:54</td>
</tr>
<tr>
<td>B-10%</td>
<td>0</td>
<td>2:40</td>
<td>15:34</td>
<td>1.06</td>
<td>3:14</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>2:36</td>
<td>15:21</td>
<td>1.20</td>
<td>2:44</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>2:53</td>
<td>15:40</td>
<td>1.24</td>
<td>2:38</td>
</tr>
</tbody>
</table>

In table 4 we compare the solution quality of KaRRi with \( \rho \in \{0s, 300s, 600s\} \). At \( \rho = 300s \), we observe improvements for both riders and vehicles. Here, existing wait times are replaced with walking which leads to benefits for all agents. By allowing longer walking distances with \( \rho = 600s \), we can further improve the vehicle operation times. However, since we equally weight vehicle and rider times \( \tau = 1 \), riders are often required to walk further to save time for vehicles, increasing the average wait and trip times. Different values for the cost function parameters may be better suited to reflect the needs of riders, particularly in a future of autonomously piloted taxis. We defer an according analysis to future work.

6 Conclusions and Future Work

KaRRi develops efficient many-to-many routing with bucket contraction hierarchies for dynamic taxi sharing. This allows real-time dispatching systems to enjoy benefits like a reduction in operating costs and air pollution even with large vehicle fleets and many meeting points. A flexible cost function allows configuration to many situations, e.g. using walking, bicycles or scooters. We expect that the new techniques like sorted buckets can also be applied for other problems that use many-to-many routing with correlated sources and targets.

KaRRi’s small running times open dynamic taxi sharing up to a variety of extensions that promise to improve the quality of service. We are particularly interested in going away from greedy online scheduling, instead taking into account pre-booked trips and opportunities to transparently change existing trips for local search style optimizations. Additionally, we expect that we can generalize KaRRi to integrate it with public transportation s.t. meeting points can be stops of buses or trains and the cost function has to take into account the public transportation schedule. A longer term perspective is to allow transfers between vehicles during a trip. This may increase the number of shared rides, eventually leading to a highly adaptive software defined public transportation system. These extensions imply interesting algorithmic challenges as they lead to a combinatorial explosion of possible route options.

Future parallelization both over over different meeting points and over entire requests can improve scalability to even larger metropolitan regions.
References


[47] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. Optimal multi-meeting-point route search. IEEE Transactions on Knowledge and Data Engineering, 28:770–784, 2016. ISSN 1041-4347. doi:10.1109/TKDE.2015.2492554


[71] Dominik Ziemke, Ihab Kaddoura, and Kai Nagel. The MATSim Open Berlin scenario: A multimodal agent-based transport simulation scenario based on
synthetic demand modeling and open data. *Procedia Computer Science*, 151, 2019. ISSN 1877-0509. doi[DOI.org/10.1016/j.procs.2019.04.120]