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 interpolating 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 [8, 45, 2, 20] and real-world field tests [27, 44, 68, 66, 40, 65, 24, 70]. A widespread adaptation of taxi sharing is expected to coincide with an increased demand for sustainable personal transportation [63, 68] and the availability of autonomously piloted vehicles [20, 21, 55, 19, 6].

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 [64, 4], may be shared with other customers [64, 42], and may alleviate concerns of privacy for riders [29]. 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) [43, 26] 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 [11]. For multiple meeting points, our routing techniques are up to

---

[*] The full version of this paper [46] is available on arXiv.

[†] Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Algorithm Engineering.

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, 32]. 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 metaheuristics 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, 52], 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 [39, 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, 39, 53, 33, 37, 38, 57, 49]. 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, 36, 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 [1].

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 [43, 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 [25]. 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. [56] 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 euclidian 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. [56] 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 [46].

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, 64].

Li et al. [47] 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 variant.

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* $R(\nu) = \langle s_0(\nu), \dots, s_{k(\nu)}(\nu)\rangle$ of a vehicle ν 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 $orig(r)$ within a time radius ρ. Analogously, our default set of dropoff locations (or *dropoffs*) $D_\rho(r)$ contains all eligible vertices from which the rider can reach $dest(r)$ within ρ. 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 ρ is a model parameter. For the sake of simplicity, we use the same ρ for every request but the model also permits varying ρ 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, ν, i, j) indicating that vehicle ν 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 $c(\iota)$ of an insertion $\iota = (r, p, d, \nu, i, j)$ represents the associated vehicle operation cost and the rider service quality in a linear combination of the form

$$c(\iota) = t_{detour}(\iota) + \tau \cdot (t_{trip}(\iota) + t^{+}_{trip}(\iota)) + \omega \cdot t_{walk}(\iota) + c^{vio}_{wait}(\iota) + c^{vio}_{trip}(\iota). \tag{2.1}$$

Here, the *added vehicle operation time* $t_{detour}(\iota)$ describes the time that vehicle ν needs for the detour it makes to accommodate the pickup at p and dropoff at d in its route. The *trip time* $t_{trip}(\iota)$ denotes the time that passes between the issuing of request r ($t_{req}(r)$) and the arrival of the rider at their destination $dest(r)$, including waiting and walking times. The detours made by ν may increase this trip time for existing riders of ν. The *added trip time* $t^{+}_{trip}(\iota)$ is the sum of these increases for all affected riders. The *walking time* $t_{walk}(\iota)$ represents how long the rider needs to move from their origin to the pickup and from the dropoff to their destination using local transportation. We weight the importance of rider trip times and walking times relative to the vehicle operation time using the model parameters τ and ω.

The remaining terms $c^{vio}_{wait}(\iota)$ and $c^{vio}_{trip}(\iota)$ describe penalties for violating constraints on the service quality of the new rider. We consider a total of four constraints originally put forth in [11]. After the insertion, the following must hold: First, the *occupancy* of ν must never exceed a fixed capacity. Second, the vehicle must still reach its last stop before a fixed *end of its service time*. Third, every rider already assigned to ν must still be picked up at their pickup stop within a *maximum wait time* t^{max}_{wait}. Fourth, every rider $\hat{r}$ already assigned to ν must still arrive at their destination within a *maximum trip time* $t^{max}_{trip}(\hat{r}) = \alpha \cdot \delta(orig(\hat{r}), dest(\hat{r})) + \beta$ where $\delta(orig(\hat{r}), dest(\hat{r}))$ is the direct vehicle travel time from the origin to the destination of $\hat{r}$. The values t^{max}_{wait}, α and β are model parameters.

All four constraints are hard constraints w.r.t. requests already assigned to ν. If ι breaks a hard constraint, we set the cost to ∞. 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 $c^{vio}_{wait}(\iota)$ and $c^{vio}_{trip}(\iota)$. Assume, the rider is picked up at p at time t_{dep}. We define

$$\begin{aligned} c^{vio}_{wait}(\iota) &= \gamma_{wait} \cdot \max\{t_{dep} - t_{req}(r) - t^{max}_{wait}, 0\}, \\ c^{vio}_{trip}(\iota) &= \gamma_{trip} \cdot \max\{t_{trip}(\iota) - t^{max}_{trip}(r), 0\} \end{aligned}$$

with model parameters γ_{wait} and γ_{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. [11] 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* [18] 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 $\tilde{\delta}(s, v)$ for every $v \in V$. An addressable priority queue Q with $\text{key}(v) = \tilde{\delta}(s, v)$ contains *active* vertices. Initially, $Q := \{s\}$, $\tilde{\delta}(s, s) := 0$ and $\tilde{\delta}(s, v) := \infty$ 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 $\tilde{\delta}(s, v)$ with $\tilde{\delta}(s, u) + \ell(e)$. 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* [26]. 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, w) is created if $(u, v, w) \in E^2$ 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 δ^+ is the according distance function. 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 *upwards search graph* $G^{\uparrow} := (V, E^{\uparrow})$ and a *downwards search graph* $G^{\downarrow} := (V, E^{\downarrow})$. The distances $\delta^{\uparrow}$ and $\delta^{\downarrow}$ represent δ^+ constrained to $G^{\uparrow}$ and $G^{\downarrow}$.

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 [26]. A *CH query* from a source $s \in V$ to a target $t \in V$ runs a forward Dijkstra search from s in $G^{\uparrow}$ and a reverse Dijkstra search from t in $G^{\downarrow}$. 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

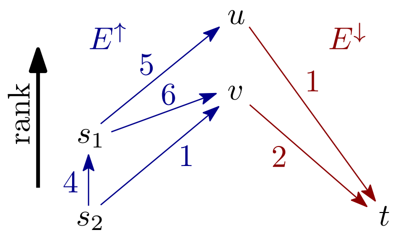

Figure 1: Example CH. Edges are annotated with weights. Vertical order of vertices indicates rank. Upward edges are blue and downward edges are red.

exceeds the best previously found distance from s to t.

Bucket Contraction Hierarchy Searches. *Bucket Contraction Hierarchy (BCH)* searches [43, 26] 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* $B^\uparrow(v)$ at each vertex $v \in V$. Conceptually, $B^\uparrow(v)$ 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 $G^\uparrow$ is run that adds an entry $(s, \delta^\uparrow(s, v))$ to $B^\uparrow(v)$ for every settled $v \in V$. Then, a reverse search from t in $G^\downarrow$ can compute tentative shortest path distances as $\delta^\uparrow(s, v) + \delta^\downarrow(v, t)$ for every bucket entry $(s, \delta^\uparrow(s, v)) \in B^\uparrow(v)$ at every settled vertex v.

Consider the example CH depicted in fig. 1. A BCH for $S = \{s_1, s_2\}$ would have the buckets $B^\uparrow(u) = \langle(s_1, 5), (s_2, 9)\rangle$ and $B^\uparrow(v) = \langle(s_1, 6), (s_2, 1)\rangle$. A reverse search from t traverses $G^\downarrow$ and finds shortest up-down paths between s_1, s_2 and t by scanning $B^\uparrow(u)$ and $B^\uparrow(v)$.

BCH searches can analogously compute the distances from a single source to a set of targets. In that case, we speak of *target buckets* $B^\downarrow(v)$ for every $v \in V$.

The advantage of BCH searches over point-to-point CH queries is that the search space of each source and each target is only traversed once, either to generate bucket entries or to scan bucket entries. However, BCH searches require more memory to store the bucket entries.

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 $\nu \in F$ define a *leeway* $\lambda(s_i, s_{i+1})$, i.e. a maximum permissible detour, between each pair of consecutive stops $(s_i, s_{i+1}) \in R(\nu)$. 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 ρ 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 ι^* 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 $\hat{c}$ denote an upper bound on the cost of the best insertion of r. For example, this can be the cost $c(\iota^*)$ of the best insertion ι^* 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 $\hat{c}$. 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 $B^{\uparrow}(v)$ for every $v \in V$. For every last stop $s_{k(\nu)}$, we generate an entry $(s_{k(\nu)}, \delta^{\uparrow}(s_{k(\nu)}, v)) \in B^{\uparrow}(v)$ at each vertex v in the upward CH search space rooted at $s_{k(\nu)}$. Then, for every pickup $p \in P_\rho$, 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 $(s_{k(\nu)}, \delta^{\uparrow}(s_{k(\nu)}, v)) \in B^{\uparrow}(v)$, it tries to improve the tentative distance $\tilde{\delta}(s_{k(\nu)}, p)$ with $\delta^{\uparrow}(s_{k(\nu)}, v) + \delta^{\downarrow}(v, p)$. Eventually, the shortest distance $\delta(s_{k(\nu)}, p)$ is found for every last stop $s_{k(\nu)}$. 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 $\hat{c}$, 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 $B^{\uparrow}(v)$ by their distance $\delta^{\uparrow}(s_{k(\nu)}, v)$ in increasing order. Suppose a pickup query rooted at $p \in P_\rho$ scans the bucket $B^{\uparrow}(v)$. For each entry $e = (s_{k(\nu)}, \delta^{\uparrow}(s_{k(\nu)}, v)) \in B^{\uparrow}(v)$, we can compute a vehicle-independent lower bound $c_{\min}(e)$ on the cost of any PALS insertion ι where the vehicle drives a distance of at least $\delta^{\uparrow}(s_{k(\nu)}, v) + \delta^{\downarrow}(v, p)$ to p. Since $c_{\min}(e)$ then increases monotonously with $\delta^{\uparrow}(s_{k(\nu)}, v)$, we can stop

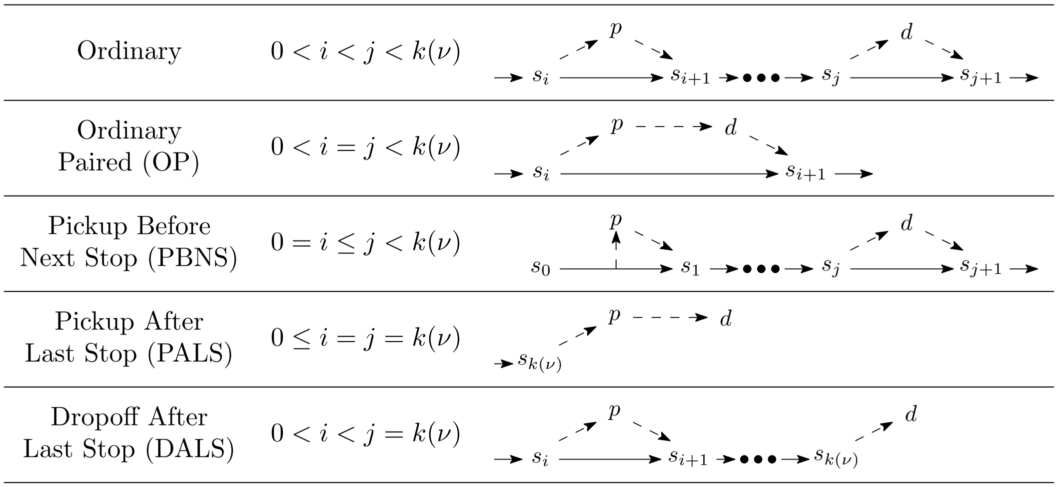

Figure 2: Insertion types. Shows characterization of each type based on the pickup and dropoff insertion points i and j of an insertion $\iota = (r, p, d, \nu, i, j)$. Illustrations depict the current route of ν (solid arrows) with stops $s \in R(\nu)$ as well as the detours to and from p and d (dashed lines).

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 $\iota^*_{\text{pals}} = (r, p, d, \nu, k(\nu), k(\nu))$ already, we would only need to find $\delta(s_{k(\nu)}, p)$. We propose a *collective BCH search* that directly finds ι^*_{pals} by propagating labels that represent PD-pairs through the search graph and pruning labels by comparing them to one another.

A label $(p, d, \delta^{\downarrow}(v, p))$ at a vertex $v \in V$ consists of the pickup $p \in P_\rho$, dropoff $d \in D_\rho$ and downwards distance $\delta^{\downarrow}(v, p)$. 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 $p \in P_\rho$, an open label $(p, d, 0)$ is created for each $d \in D_\rho$. 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 $l = (p, d, \delta^{\downarrow}(v, p))$ consists of two steps: First, we search for a new best insertion by traversing all entries in the last stop bucket $B^{\uparrow}(v)$ for l. For each entry $e = (s_{k(\nu)}, \delta^{\uparrow}(s_{k(\nu)}, v)) \in B^{\uparrow}(v)$, we get a tentative distance $\tilde{\delta}(s_{k(\nu)}, p) = \delta^{\uparrow}(s_{k(\nu)}, v) + \delta^{\downarrow}(v, p)$. With this tentative distance, we compute an upper bound $c_{\max}(l, e)$ on $c(\iota)$ for $\iota = (r, p, d, \nu, k(\nu), k(\nu))$. If $c_{\max}(l, e) < \hat{c}$, we mark ι 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 $c_{\max}(l, e) = c(\iota)$ 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 $l' = (p, d, \ell^{+}(w, v) + \delta^{\downarrow}(v, p))$. 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 $l = (p, d, \delta^{\downarrow}(v, p))$ in the search sub-tree rooted at v. Let $G^{\downarrow}_v = (V^{\downarrow}_v, E^{\downarrow}_v)$ denote the sub-graph of $G^{\downarrow}$ consisting of all paths to v. Consider a vertex $w \in V^{\downarrow}_v$ and an entry $e = (s_{k(\nu)}, \delta^{\uparrow}(s_{k(\nu)}, w)) \in B^{\uparrow}(w)$. Let $c_{\max}(l, v, e) := c((r, p, d, \nu, k(\nu), k(\nu)))$ under the assumption $\delta(s_{k(\nu)}, p) \overset{!}{=} \delta^{\uparrow}(s_{k(\nu)}, w) + \delta^{\downarrow}(w, v) + \delta^{\downarrow}(v, p)$.

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* $w \in V^{\downarrow}_v$ *and* $e \in B^{\uparrow}(w)$.

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]. ☐

Table 1: Key figures of our benchmark instances.

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

Since $c_{\max}(l, v, e)$ increases (almost) linearly with $\delta^{\uparrow}(s_{k(\nu)}, w) + \delta^{\downarrow}(w, v)$, we can (almost) test whether l dominates l' at v in constant time without having to look at any bucket entries in $G^{\downarrow}_{v}$. The cost function is not linear, though, due to the rider walking time and the potential penalties for violating soft constraints. However, we can still compute a very good under-approximation of the domination relation in constant time. For details on this approximation as well as a minor limitation of collective searches w.r.t. the service 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* $\lambda(s_i, s_{i+1}) - \delta^{\uparrow}(s_i, v)$ of each entry $(s_i, \delta^{\uparrow}(s_i, v))$ (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][2]. The underlying road networks are obtained from OpenStreetMap data[3] and CHs are computed using the open-source library RoutingKit[4]. 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_ρ and dropoffs N^d_ρ by using increasing walking radii $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$ 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 $\beta = 2\text{min}$ which means that each trip may take up to a maximum trip time of $1.7\delta(\mathit{orig}, \mathit{dest}) + 2\text{min}$. For the remaining parameters, we choose $t^{max}_{wait} = 600\text{s}$, $\gamma_{wait} = 1$, and $\gamma_{trip} = 10$.

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 $\rho \in \{150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$. We depict the speedups observed for bundled searches for the PALS and DALS cases in fig. 3. In preliminary experiments,

---

[1] Available at `https://github.com/molaupi/karri`.

[2] 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] `https://download.geofabrik.de/`, accessed Oct 30th 2023.

[4] `https://github.com/RoutingKit`, accessed Oct 30th 2023.

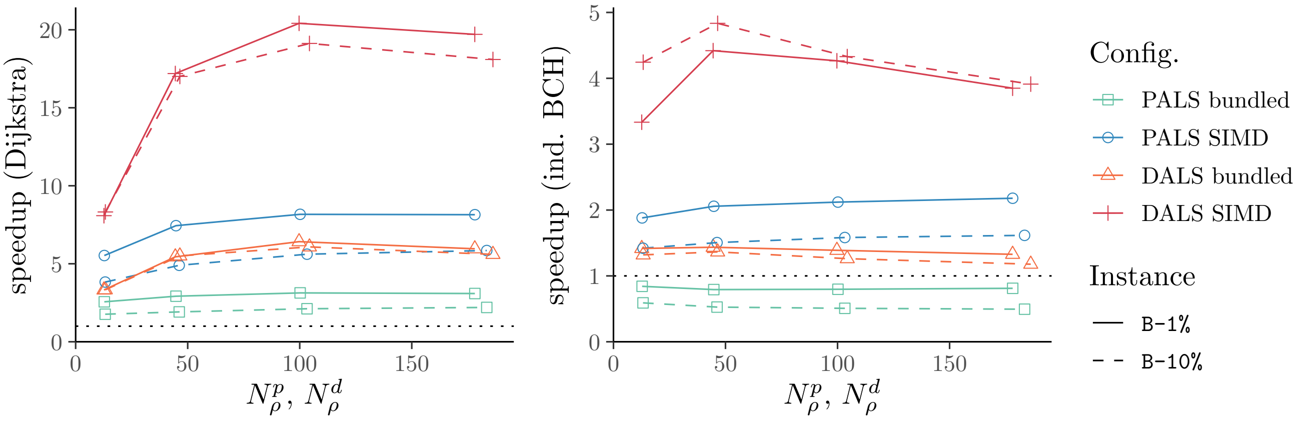

Figure 3: Mean speedups for bundling without SIMD instructions (*bundled*) and with SIMD instructions (*SIMD*) over non-bundled searches for Dijkstra searches (left) and individual BCH searches (right). Considers the PALS and DALS cases on the `B-1%` and `B-10%` instances with $\rho \in \{150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$. Note the different y-axes.

we found $k = 64$ to be best suited for Dijkstra searches. For individual BCH searches, $k = 8$, $k = 16$, or $k = 32$ may be 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 $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$ and $k = 1$. The speedups achieved with sorted buckets are shown in fig. 4.

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 ρ^4, this initialization can constitute up to 85% of the search time for larger values of ρ but sorted buckets have no effect on it.

Consequently, we observe speedups of only 1.96

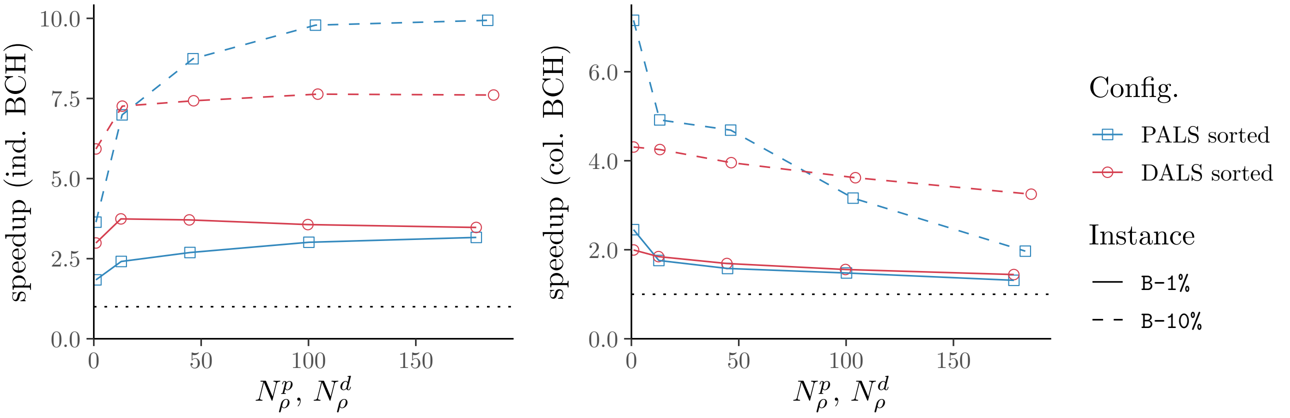

Figure 4: Mean speedups for individual (left, $k = 1$) and collective (right) BCH queries with sorted buckets over unsorted buckets. Considers the PALS and DALS cases on the `B-1%` and `B-10%` instances for $\rho \in \{0\text{s}, 150\text{s}, 300\text{s}, 450\text{s}, 600\text{s}\}$. Note the different y-axes.

(PALS) and 3.22 (DALS) for $\rho = 600$s on `Berlin-10pct`. If we disregard the overhead for initial labels, these speedups increase to 7.51 and 3.63.

Maintaining sorted last stop buckets incurs an average overhead per request of about 10µs for `Berlin-1pct` and about 35µs for `Berlin-10pct` while the reduction in search time is one to three orders of magnitude larger.

Collective BCH Searches. In table 2, we compare the search times and the times needed to enumerate candidate insertions for the three last stop search approaches. Additionally, we show the number of relaxed edges and scanned bucket entries. We report the results for $\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$ on the `B-1%` and `B-10%` instances. We use the optimal configuration for each combination of search type, insertion type, and radius.

At $\rho = 0$s, collective searches are slower than individual BCH searches as there is only a single pickup and dropoff so the overhead for explicitly maintaining labels instead of a single distance per vertex is unwarranted. At $\rho = 300$s and $\rho = 600$s, collective searches offer the best search times, though. In the PALS case, collective searches are up to 4 times faster than individual BCH searches. In the DALS case, this relative speedup is even larger at up to 14. We attribute the better scalability of collective searches to two main advantages:

Firstly, collective searches use more precise lower bounds on the cost of specific PD-pairs or dropoffs instead of a general lower bound that applies for every PD-pair or dropoff. Thus, both bucket scans and the search as a whole can be stopped earlier.

Secondly, collective searches consider all sources in one search, maximizing the amount of information available for domination pruning. Contrarily, bundled searches only consider k sources at once, so work may be repeated up to N_ρ^p/k times. Thus, the number of edge relaxations and bucket entry scans increases faster with the number of pickups ($N_\rho^p, N_\rho^d \sim \rho^2$) for individual BCH searches than for collective BCH searches.

In addition, the enumeration times remain small for collective searches while they increase massively with ρ for individual BCH searches. This is due to the fact that collective searches identify a single candidate insertion in the PALS case or a small set of candidate vehicles and dropoffs in the DALS case. Contrarily, individual BCH searches first find all distances and then enumerate an insertion for each combination of candidate vehicle, pickup and dropoff. As the number of PD-pairs is proportional to ρ^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 ρ in the PALS case. As stated before, with large ρ, 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. [11].

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 ($\rho = 0$s) 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 $\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$ on the `B-1%` and `B-10%` instances. Shows average number of edge relaxations ($\#_{\text{rel.}}$), number of bucket entries scanned ($\#_{\text{scans}}$), search time (t_{search}) and time for enumerating insertions (t_{enum}) per request. Bold numbers indicate smallest times per radius.

			`Berlin-1pct`				`Berlin-10pct`			
Type	ρ	Search	$\#_{\text{rel.}}$	$\#_{\text{scans}}$	t_{search}	t_{enum}	$\#_{\text{rel.}}$	$\#_{\text{scans}}$	t_{search}	t_{enum}
PALS	0	Coll.	40	8	4.96	**0.04**	19	11	3.89	**0.06**
		BCH	37	8	**3.63**	0.37	18	10	**3.11**	0.56
		Dij.	577	2	43.44	0.36	225	4	18.75	0.52
	300	Coll.	412	57	**70.16**	**0.08**	168	58	**41.95**	**1.49**
		BCH	967	274	73.95	44.12	797	1011	103.36	155.86
		Dij.	4302	17	497.94	40.4	3533	99	433.66	138.16
	600	Coll.	806	108	**286.11**	**0.09**	219	82	**213.9**	**23.90**
		BCH	5555	2514	424.65	812.64	4734	12214	823.72	3475.00
		Dij.	41092	137	4481.08	812.24	38102	960	4412.38	3098.09
DALS	0	Coll.	210	676	36.56	0.76	191	5066	95.37	**2.74**
		BCH	216	721	**23.78**	**0.72**	197	5419	**87.14**	2.96
		Dij.	19063	–	1665.24	15.60	14920	–	1344.48	58.98
	300	Coll.	253	662	**58.04**	**5.20**	235	5049	**117.04**	**20.97**
		BCH	2015	4116	182.81	145.01	1961	32487	623.84	596.46
		Dij.	26567	–	3561.61	232.62	22228	–	3014.30	984.84
	600	Coll.	296	656	**93.11**	**13.84**	277	5091	**157.07**	**53.61**
		BCH	8042	14602	685.23	1227.00	7683	115912	2214.15	4261.21
		Dij.	98143	–	12453.92	3063.71	88021	–	11307.36	12665.75

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 $N_\rho^p = N_\rho^d = 1$. Our last stop BCH searches are well suited for $\rho = 0$s, 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 naïve 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 $\rho = 300$s and $\rho = 600$s 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 naïve 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 naïve 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 [10]. 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 $\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$ on `B-1%` and `B-10%`. Shows mean times for finding P_ρ and D_ρ, 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.

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

Table 4: Solution quality of KaRRi with different radii ($\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$) 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).

Inst.	ρ	wait	trip	occ	op
`B-1%`	0	3:44	16:53	0.88	4:28
	300	3:15	15:54	0.93	4:00
	600	3:27	16:02	0.94	3:54
`B-10%`	0	2:40	15:34	1.06	3:14
	300	2:36	15:21	1.20	2:44
	600	2:53	15:40	1.24	2:38

In table 4, we compare the solution quality of KaRRi with $\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$. At $\rho = 300\text{s}$, 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 $\rho = 600\text{s}$, 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

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Lecture Notes in Computer Science*, volume 6630 LNCS, pages 230–241. Springer, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-20662-7_20.

[2] 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. ISSN 01912615. doi:10.1016/j.trb.2011.05.017.

[3] Niels Agatz, Alan Erera, Martin Savelsbergh, and Xing Wang. Optimization for dynamic ride-sharing: A review. *European Journal of Operational Research*, 223:295–303, 2012. ISSN 03772217. doi:10.1016/j.ejor.2012.05.028.

[4] Kamel Aissat and Ammar Oulamara. A priori approach of real-time ridesharing problem with intermediate meeting locations. *Journal of Artificial Intelligence and Soft Computing Research*, 4:287–299, 2014. ISSN 2083-2567. doi:10.1515/jaiscr-2015-0015.

[5] 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. ISSN 10916490. doi:10.1073/pnas.1611675114.

[6] Claudine Badue, Rânik Guidolini, Raphael Vivacqua Carneiro, Pedro Azevedo, Vinicius B. Cardoso, Avelino Forechi, Luan Jesus, Rodrigo Berriel, Thiago M. Paixão, Filipe Mutz, Lucas de Paula Veronese, Thiago Oliveira-Santos, and Alberto F. De Souza. Self-driving cars: A survey. *Expert Systems with Applications*, 165, 2021. ISSN 09574174. doi:10.1016/j.eswa.2020.113816.

[7] Reinhard Bauer and Daniel Delling. SHARC: Fast and robust unidirectional routing. *ACM Journal of Experimental Algorithms (JEA)*, 14, 2010. ISSN 1084-6654. doi:10.1145/1498698.1537599.

[8] Joschka Bischoff, Michal Maciejewski, and Kai Nagel. City-wide shared taxis: A simulation study in Berlin. In *IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, 2017. doi:10.1109/ITSC.2017.8317926.

[9] Filippo Bistaffa, Alessandro Farinelli, and Sarvapali Ramchurn. Sharing rides with friends: A coalition formation algorithm for ridesharing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 29, 2015. ISSN 2374-3468. doi:10.1609/aaai.v29i1.9242.

[10] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Real-time traffic assignment using engineered customizable contraction hierarchies. *ACM Journal of Experimental Algorithms (JEA)*, 24, 2019. ISSN 1084-6654. doi:10.1145/3362693.

[11] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Fast, exact and scalable dynamic ridesharing. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 98–112. Society for Industrial and Applied Mathematics, 2021. ISBN 9781611976472. doi:10.1137/1.9781611976472.8.

[12] Jean François Cordeau. A branch-and-cut algorithm for the dial-a-ride problem. *Operations Research*, 54:573–586, 2006. ISSN 0030364X. doi:10.1287/opre.1060.0283.

[13] Jean François Cordeau and Gilbert Laporte. The dial-a-ride problem: Models and algorithms. *Annals of Operations Research*, 153:29–46, 2007. ISSN 02545330. doi:10.1007/s10479-007-0170-8.

[14] 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)*. LIPIcs, 2011. ISBN 978-3-939897-33-0. doi:10.4230/OASIcs.ATMOS.2011.52.

[15] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73:940–952, 2013. ISSN 07437315. doi:10.1016/j.jpdc.2012.02.007.

[16] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *INFORMS Transportation Science*, 51, 2017. doi:10.1287/trsc.2014.0579.

[17] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 21:1–49, 2016. ISSN 1084-6654. doi:10.1145/2886843.

[18] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1, 1959.

[19] Fábio Duarte and Carlo Ratti. The impact of autonomous vehicles on cities: A review. *Journal of Urban Technology*, 25:3–18, 2018. ISSN 1063-0732. doi:10.1080/10630732.2018.1493883.

[20] 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. ISSN 0968090X. doi:10.1016/j.trc.2013.12.001.

[21] 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. ISSN 0049-4488. doi:10.1007/s11116-016-9729-z.

[22] Andres Fielbaum, Xiaoshan Bai, and Javier Alonso-Mora. On-demand ridesharing with optimized pick-up and drop-off walking locations. *Transportation Research Part C: Emerging Technologies*, 126:103061, 2021. ISSN 0968090X. doi:10.1016/j.trc.2021.103061.

[23] Masabumi Furuhata, Maged Dessouky, Fernando Ordóñez, Marc Etienne Brunet, Xiaoqing Wang, and Sven Koenig. Ridesharing: The state-of-the-art and future directions. *Transportation Research Part B: Methodological*, 57:28–46, 2013. ISSN 01912615. doi:10.1016/j.trb.2013.08.012.

[24] 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. ISSN 23519789. doi:10.1016/j.promfg.2015.07.329.

[25] Robert Geisberger, Dennis Luxen, Sabine Neubauer, Peter Sanders, and Lars Volker. Fast detour computation for ride sharing. In *OpenAccess Series in Informatics*, volume 14, pages 88–99. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010. ISBN 9783939897200. doi:10.4230/OASIcs.ATMOS.2010.88.

[26] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *INFORMS Transportation Science*, 46, 2012. doi:10.1287/trsc.1110.0401.

[27] 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 B.V., 2020. doi:10.1016/j.trpro.2020.03.105.

[28] Preeti Goel, Lars Kulik, and Kotagiri Ramamohanarao. Privacy-aware dynamic ride sharing. *ACM Transactions on Spatial Algorithms and Systems*, 2: 1–41, 2016. ISSN 2374-0353. doi:10.1145/2845080.

[29] Preeti Goel, Lars Kulik, and Kotagiri Ramamohanarao. Optimal pick up point selection for effective ride sharing. *IEEE Transactions on Big Data*, 3: 154–168, 2016. doi:10.1109/tbdata.2016.2599936.

[30] Wesam Herbawi and Michael Weber. A genetic and insertion heuristic algorithm for solving the dynamic ridematching problem with time windows. In *GECCO'12 - Proceedings of the 14th International Conference on Genetic and Evolutionary Computation*, pages 385–392, 2012. ISBN 9781450311779. doi:10.1145/2330163.2330219.

[31] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. *The Shortest Path Problem: 9th DIMACS Implementation Challenge*, 2009.

[32] 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. ISSN 01912615. doi:10.1016/j.trb.2018.02.001.

[33] Mark E.T. Horn. Fleet scheduling and dispatching for demand-responsive passenger services. *Transportation Research Part C: Emerging Technologies*, 10:35–63, 2002. ISSN 0968090X. doi:10.1016/S0968-090X(01)00003-1.

[34] Andreas Horni, Kai Nagel, and Kay W. Axhausen, editors. *The Multi-Agent Transport Simulation MATSim*. Ubiquity Press, 2016. ISBN 9781909188754. doi:10.5334/baw.

[35] Hadi Hosni, Joe Naoum-Sawaya, and Hassan Artail. The shared-taxi problem: Formulation and solution methods. *Transportation Research Part B: Methodological*, 70:303–318, 2014. ISSN 01912615. doi:10.1016/j.trb.2014.09.011.

[36] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. Large scale realtime ridesharing with service guarantee on road networks. In *Proceedings of the VLDB Endowment*, volume 7, pages 2017–2028. Association for Computing Machinery, 2014. doi:10.14778/2733085.2733106.

[37] Brady Hunsaker and Martin Savelsbergh. Efficient feasibility testing for dial-a-ride problems. *Operations Research Letters*, 30:169–173, 2002. ISSN 01676377. doi:10.1016/S0167-6377(02)00120-7.

[38] Carl H. Häll, Magdalena Högberg, and Jan T. Lundgren. A modeling system for simulation of dial-a-ride services. *Public Transport*, 4:17–37, 2012. ISSN 1866-749X. doi:10.1007/s12469-012-0052-6.

[39] 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*, 20:243–257, 1986. ISSN 01912615. doi:10.1016/0191-2615(86)90020-2.

[40] 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. ISSN 0967070X. doi:10.1016/j.tranpol.2017.12.004.

[41] Jaeyoung Jung, R. Jayakrishnan, and Ji Young Park. Dynamic shared-taxi dispatch algorithm with hybrid-simulated annealing. *Computer-Aided Civil and Infrastructure Engineering*, 31:275–291, 2016. ISSN 10939687. doi:10.1111/mice.12157.

[42] Levent Kaan and Eli V. Olinick. The vanpool assignment problem: Optimization models and solution algorithms. *Computers and Industrial Engineering*, 66:24–40, 2013. ISSN 03608352. doi:10.1016/j.cie.2013.05.020.

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

[44] Nadine Kostorz, Eva Fraedrich, and Martin Kagerbauer. Usage and user characteristics—insights from MOIA, europe's largest ridepooling service. *Sustainability*, 13:958, 2021. ISSN 2071-1050. doi:10.3390/su13020958.

[45] 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. ISSN 0361-1981. doi:10.1177/03611981231170624.

[46] Moritz Laupichler and Peter Sanders. Fast many-to-many routing for dynamic taxi sharing with meeting points. *arXiv*, 2023.

[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.

[48] 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. ISSN 18770428. doi:10.1016/j.sbspro.2012.04.122.

[49] Charlotte Lotze, Philip Marszal, Malte Schröder, and Marc Timme. Dynamic stop pooling for flexible and sustainable ride sharing. *New Journal of Physics*, 24:023034, 2022. ISSN 1367-2630. doi:10.1088/1367-2630/ac47c9.

[50] Shuo Ma, Yu Zheng, and Ouri Wolfson. T-Share: A large-scale dynamic taxi ridesharing service. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 410–421. IEEE, 2013. ISBN 978-1-4673-4910-9. doi:10.1109/ICDE.2013.6544843.

[51] Shuo Ma, Yu Zheng, and Ouri Wolfson. Real-time city-scale taxi ridesharing. *IEEE Transactions on Knowledge and Data Engineering*, 27:1782–1795, 2015. ISSN 1041-4347. doi:10.1109/TKDE.2014.2334313.

[52] Tai Yu Ma, Saeid Rasulkhani, Joseph Y.J. Chow, and Sylvain Klein. A dynamic ridesharing dispatch and idle vehicle repositioning strategy with integrated transit transfers. *Transportation Research Part E: Logistics and Transportation Review*, 128:417–442, 2019. ISSN 13665545. doi:10.1016/j.tre.2019.07.002.

[53] Oli B. G. Madsen, Hans F. Ravn, and Jens Moberg Rygaard. A heuristic algorithm for a dial-a-ride problem with time windows, multiple capacities, and multiple objectives. *Annals of Operations Research*, 60:193–208, 1995. ISSN 0254-5330. doi:10.1007/BF02031946.

[54] Carlo Manna and Steve Prestwich. Online stochastic planning for taxi and ridesharing. In *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI*, volume 2014-December, pages 906–913. IEEE Computer Society, 2014. ISBN 9781479965724. doi:10.1109/ICTAI.2014.138.

[55] 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*, 21:324–348, 2017. ISSN 1547-2450. doi:10.1080/15472450.2017.1291351.

[56] 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. 2021. doi:10.48550/arXiv.2105.00994.

[57] Masayo Ota, Huy Vo, Claudio Silva, and Juliana Freire. STaRS: Simulating taxi ride sharing at scale. *IEEE Transactions on Big Data*, 3:349–361, 2017. ISSN 2332-7790. doi:10.1109/TBDATA.2016.2627223.

[58] Dominik Pelzer, Jiajian Xiao, Daniel Zehe, Michael H. Lees, Alois C. Knoll, and Heiko Aydt. A partition-based match making algorithm for dynamic ridesharing. *IEEE Transactions on Intelligent Transportation Systems*, 16:2587–2598, 2015. ISSN 1524-9050. doi:10.1109/TITS.2015.2413453.

[59] Harilaos N. Psaraftis. A dynamic programming solution to the single vehicle many-to-many immediate request dial-a-ride problem. *Transportation Science*, 14:130–154, 1980. ISSN 0041-1655. doi:10.1287/trsc.14.2.130.

[60] 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*, 42:6728–6737, 2015. ISSN 09574174. doi:10.1016/j.eswa.2015.04.060.

[61] Martin Savelsbergh. Local search in routing problems with time windows. *Annals of Operations Research*, 4:285–305, 1985. ISSN 0254-5330. doi:10.1007/BF02022044.

[62] 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*, 38:1719–1730, 2011. ISSN 03050548. doi:10.1016/j.cor.2011.02.006.

[63] 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. ISSN 0197-6729. doi:10.1155/2021/6627660.

[64] Mitja Stiglic, Niels Agatz, Martin Savelsbergh, and Mirko Gradisar. The benefits of meeting points in ride-sharing systems. *Transportation Research Part B: Methodological*, 82:36–53, 2015. ISSN 01912615. doi:10.1016/j.trb.2015.07.025.

[65] Chichung Tao and Chungjung Wu. Behavioral responses to dynamic ridesharing services - the case of taxi-sharing project in Taipei. In *2008 IEEE International Conference on Service Operations and Logistics, and Informatics*, pages 1576–1581. IEEE, 2008. ISBN 978-1-4244-2012-4. doi:10.1109/SOLI.2008.4682777.

[66] 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. ISSN 22105395. doi:10.1016/j.rtbm.2018.06.003.

[67] Hiroki Yanagisawa. A multi-source label-correcting algorithm for the all-pairs shortest paths problem. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10. IEEE, 2010. ISBN 978-1-4244-6442-5. doi:10.1109/IPDPS.2010.5470362.

[68] 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. ISSN 03062619. doi:10.1016/j.apenergy.2017.01.052.

[69] Xingbin Zhan, W.Y. Szeto, and Xiqun Michael Chen. The dynamic ride-hailing sharing problem with multiple vehicle types and user classes. *Transportation Research Part E: Logistics and Transportation Review*, 168, 2022. ISSN 13665545. doi:10.1016/j.tre.2022.102891.

[70] 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. ISSN 0154-3229. doi:10.4000/rei.9984.

[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.