Fast Many-to-Many Routing for Ridesharing with Multiple Pickup and Dropoff Locations

Moritz Laupichler
Institute of Theoretical Informatics, Algorithm Engineering, Karlsruhe Institute of Technology, Kaiserstraße 12, 76131 Karlsruhe, Germany

Peter Sanders
Institute of Theoretical Informatics, Algorithm Engineering, Karlsruhe Institute of Technology, Kaiserstraße 12, 76131 Karlsruhe, Germany

Abstract
We introduce KaRRi, an improved algorithm for scheduling a fleet of shared vehicles 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 efficiently support a large number of possible pick-up and drop-off locations. This lays an algorithmic foundation for ridesharing 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 passenger wait and trip times.

2012 ACM Subject Classification Applied computing → Transportation; Theory of computation → Online algorithms; Theory of computation → Vector / streaming algorithms; Information systems → Geographic information systems

Keywords and phrases Algorithm Engineering, Route Planning, Ridesharing, Multi-Modal
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. Ridesharing systems that intelligently control large fleets of taxi-like vehicles have the potential to offer an alternative that is more convenient than public transportation and more economical and ecological than individually used cars [2, 15, 18]. This is particularly promising if these vehicles have electrical propulsion and autonomous piloting. However, current such systems do not deliver on these promises as the effectively usable capacity of the vehicles is quite small, even threatening to increase rather than decrease the total number of driven car-kilometers [14]. A main problem for larger capacity ridesharing vehicles is that picking up and dropping off customers introduces large delays for other passengers of the vehicle.

This paper lays algorithmic groundwork for a better integration of ridesharing fleets into a multi-modal transportation system. We focus on the question of how local transportation (e.g., walking, bicycles or scooters) can be used to reach a pickup or dropoff location (PD-location) that causes less delay for a vehicle, may be shared with other customers, and may alleviate concerns of privacy for riders [16, 10]. (A next, related step will be an analysis of how public transportation like trains or express buses may be used to cross large distances faster, and more economically/ecologically – overall resulting in an effective use of the hierarchy individual transportation, ridesharing, and public transit).

Our starting point is the LOUD system by Buchhold et al. [4] that comprises an online dispatching system for large ridesharing fleets. It uses one-to-many routing based on bucket contraction hierarchies (BCHs) [13, 9] to efficiently find the best insertion of a new customer into the current schedule of a vehicle. 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 LOUD with the possibility of performing the pickup and dropoff of a passenger not at fixed locations but at any location in the vicinity of the passenger’s origin and destination. The algorithm computes optimal assignments of passengers to vehicles including locations for the pickup and dropoff. We adapt LOUD’s objective function to this new scenario by incorporating passenger wait times, trip times, and overheads for individual transportation to the pickup location and from the dropoff location.

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 PD-locations. 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 the case of a single pickup and dropoff.

Our experimental evaluation uses realistic data sets to evaluate the efficiency of these measures. We find that our implementation is several times faster than LOUD in the case of a single pickup and dropoff. For multiple PD-locations, our routing techniques are up to three orders of magnitude faster than a naive extension of LOUD’s techniques. We also give first indications that allowing multiple PD-locations can reduce the operating costs of a taxi fleet by up to 15% without increasing passenger 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.
Paper Overview. After a more detailed problem statement in Section 2 we introduce basic notation and techniques in Section 3. We define the formal foundation for our cost model in the presence of multiple PD-locations in Section 4. Section 5 gives an overview on the KaRRI algorithm while Sections 6, 7 address individual challenges. In Section 8 we evaluate our approach experimentally.

2 Problem Statement

This section describes and gives the formal foundations for the dynamic ridesharing problem considered by our approach.

Road Network. We consider a road network to be a graph $G = (V_G, E_G)$ where edges represent road segments and vertices represent intersections. Every edge $e = (v, w) \in E_G$ 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 $\delta(v, w)$. Our algorithm uses two separate road networks $G_{veh}$ and $G_{psg}$ with associated $\ell_{veh}$, $\ell_{psg}$, and $\delta_{psg}$ to represent parts of the same road network accessible to vehicles and to pedestrians, respectively. Note that we only consider walking but $G_{psg}$ can also represent other modes of transportation, e.g. cycling. A passenger can board or alight a vehicle at a location $v$ if $v$ is accessible in both networks, i.e. $v \in V_{veh} \cap V_{psg}$.

Vehicle, Stop. Our algorithm has access to a fleet $F$ of vehicles. Each vehicle $\nu = \langle l_\nu, cap, t_{min}^{serv}, t_{max}^{serv} \rangle$ has an initial location $l_\nu$, a seating capacity $cap$ and a service time interval $[t^{min}_\nu, t^{max}_\nu]$. The current route $R(\nu) = \langle s_0(\nu), ..., s_{k(\nu)}(\nu) \rangle$ of a vehicle $\nu$ is a sequence of stops scheduled for the vehicle. The vehicle’s current location $l_\nu(\nu)$ is always somewhere between its previous (or current) stop $s_0(\nu)$ and its next stop $s_1(\nu)$. 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 $l(s) \in V$ in the graph. Abusing notation, we may write $s_i$ instead of $s_i(\nu)$ and only $s_i$ instead of $l(s_i)$. At each stop, a vehicle picks up and/or drops off one or more passengers, stopping for a minimum time of $t^{min}_\nu$, which is a model parameter. We denote the occupancy of a vehicle between stops $s_i$ and $s_{i+1}$ by $o(s_i)$. We update each vehicle’s route as new stops are introduced for newly assigned passengers. For each stop $s \in R(\nu)$ we maintain the earliest possible arrival time $t^{min}_\nu(s)$ and departure time $t^{max}_\nu(s)$ according to the current schedule.

Request. In our scenario, the dispatcher receives ride requests and immediately assigns them to vehicles. A request $r = \langle orig, dest, t_{req} \rangle$ 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.

Pickup (Location), Dropoff (Location). A possible pickup location (pickup for short) is a location $v \in V_{psg} \cap V_{veh}$ that is reachable from $orig(r)$ in $G_{psg}$ within a time radius $\rho$. Similarly, a possible dropoff location (dropoff) is a location $v \in V_{psg} \cap V_{veh}$ from which $dest(r)$ can be reached in $G_{psg}$ within $\rho$. The sets of pickups and dropoffs for $r$ and a radius $\rho$ are denoted by $P_\rho(r)$ and $D_\rho(r)$ Let $N_p(r) = |P_\rho(r)|$ and $N_d(r) = |D_\rho(r)|$. We collectively refer to pickups and dropoffs as PD-locations. We call a pair of pickup and dropoff a PD-pair. The radius $\rho$ is a model parameter.

Insertion. The goal of the dispatcher is to find an insertion of a pickup and dropoff of each request $r$ into any vehicle’s route $s$ 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 $p \in P_\rho(r)$ immediately after stop $s_i(\nu)$ and drops $r$ off at dropoff location $d \in D_\rho(r)$ immediately after stop $s_j(\nu)$ with $0 \leq i < j \leq k(\nu)$.
2.1 Cost Function and Constraints

The cost of an insertion $i = (r, p, d, \nu, i, j)$ considers the added vehicle operation time $t_{\text{detour}}(i)$ of $\nu$, the trip time $t_{\text{trip}}(i)$ of $r$, the sum of increased trip times $t_{\text{trip}}^{+}(i)$ of existing passengers of $\nu$, and the walking time $t_{\text{walk}}(i)$ (we defer the exact definitions of these terms to Section 4).

We consider a number of constraints for eligible insertions put forward in [4]. After the insertion, the following must hold: First, the occupancy of $\nu$ must never be exceeded by an insertion. Second, the vehicle must still reach its last stop before the end of its service time. Third, every passenger already assigned to $\nu$ must still be picked up at their pickup stop within a maximum wait time $t_{\text{wait}}^{\text{max}}$. Fourth, every passenger $\hat{r}$ already assigned to $\nu$ must still arrive at their destination within a maximum trip time $t_{\text{trip}}^{\text{max}}(\hat{r}) = \alpha \cdot \delta_{\text{veh}}(\text{orig}(\hat{r}), \text{dest}(\hat{r})) + \beta$.

The values $t_{\text{wait}}^{\text{max}}$, $\alpha$ and $\beta$ are model parameters.

All four constraints are hard constraints wrt. requests already assigned to $\nu$. If $i$ 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 cost penalties. Assume, the passenger is picked up at $p$ at time $t_{\text{dep}}$. We define the cost penalties as

\[
\begin{align*}
&c_{\text{wait}}^{\nu}(i) = \gamma_{\text{wait}} \cdot \max\{t_{\text{dep}} - t_{\text{req}}(r) - t_{\text{wait}}^{\text{max}}, 0\} \\
&c_{\text{trip}}^{\nu}(i) = \gamma_{\text{trip}} \cdot \max\{t_{\text{trip}}(i) - t_{\text{trip}}^{\text{max}}(r), 0\}
\end{align*}
\]

with model parameters $\gamma_{\text{wait}}$ and $\gamma_{\text{trip}}$ that scale the severity of the penalties.

For the total insertion cost, we use a linear combination of the vehicle detour times, passenger trip times, walking times, and soft constraint violation penalties:

\[
c(i) = t_{\text{detour}}(i) + \tau \cdot (t_{\text{trip}}(i) + t_{\text{trip}}^{+}(i)) + \omega \cdot t_{\text{walk}}(i) + c_{\text{wait}}^{\nu}(i) + c_{\text{trip}}^{\nu}(i)
\]

Note that we base our cost function on the one used in the LOUD algorithm [4]. However, the original cost function does not consider passenger trip times or walking times. We weight the importance of these times with the model parameters $\tau$ and $\omega$. Our cost function is equivalent to LOUD’s if $\tau = \omega = 0$.

3 Preliminaries

In this section, we describe several algorithms for the computation of shortest paths that are being used in this work. Furthermore, we summarize the LOUD algorithm for dynamic ridesharing [4] 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 [5] 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 $\delta(s, v)$ for every $v \in V$. An addressable priority queue (PQ) $Q$ with key($v$) = $\delta(s, v)$ contains active vertices. Initially, $Q := \{s\}$, $\delta(s, s) := 0$ and $\delta(s, v) := \infty$ for $v \neq s$. The algorithm proceeds by extracting the vertex with the smallest distance label from $Q$ and settling it. To settle $u \in V$, each outgoing edge $(u, v) \in E$ is relaxed. The relaxation of $e = (u, v)$ tries to improve the distance label $\delta(s, v)$ with $\delta(s, u) + \ell(e)$. If the distance is improved, $v$ is inserted into $Q$. The algorithm stops when $Q$ becomes empty.
Fast Many-to-Many Routing for Ridesharing with Multiple Pickup and Dropoff Loc.

Contraction Hierarchies. Contraction Hierarchies (CHs) \(^9\) are a speed-up technique for shortest path computations in road networks that exploits 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 \(^9\). 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 \((u,v)\) 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 \( \delta^+ \) is the according 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 \( E^\downarrow = \{(u,v) \in E^+ | \text{rank}(u) > \text{rank}(v)\} \). We define an upwards search graph \( G^+ := (V,E^+) \) and a downwards search graph \( G^\downarrow := (V,E^\downarrow) \). The distance functions \( \delta^+ \) and \( \delta^\downarrow \) represent \( \delta^+ \) constrained to \( G^+ \) and \( G^\downarrow \). The upwards CH search space \( G^+_v := (V^+_v,E^+_v) \) rooted at a vertex \( v \in V \) contains all vertices \( V^+_v \subseteq V \) that can be reached from \( v \) in \( E^+ \) and the according edges \( E^+_v \subseteq E^+ \). The reverse downwards CH search space \( G^\downarrow_v := (V^\downarrow_v,E^\downarrow_v) \) rooted at a vertex \( v \in V \) contains all vertices \( V^\downarrow_v \subseteq V \) from which \( v \) is reachable in \( E^\downarrow \) and the according edges \( E^\downarrow_v \subseteq E^\downarrow \).

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 \(^9\). 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 \( G^\downarrow \). 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 exceeds the best previously found distance from \( s \) to \( t \).

Bucket Contraction Hierarchy Searches. Bucket Contraction Hierarchy (BCH) \(^13,9\) searches find all shortest path distances from a source \( s \in V \) to a set of targets \( T \subseteq V \) 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 (target) bucket \( B^t(v) \) at each vertex \( v \in V \). For each target \( t \in T \), a reverse search in \( G^\downarrow \) is run that adds an entry \((t,\delta^\downarrow(v,t))\) to \( B^t(v) \) for every settled \( v \in V \). Then, a forward search from \( s \in G^+ \) can compute tentative shortest path distances as \( \delta^+(s,v) + \delta^\downarrow(v,t) \) for every bucket entry \((t,\delta^\downarrow(v,t))\) in \( B^t(v) \) at every settled vertex \( v \in V^+_v \).

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

The advantage of BCH searches is that the search space of each source and each target is only traversed once, either to compute bucket entries or to scan bucket entries. However, storing the bucket entries requires more memory than individual point-to-point CH queries.

Bundled Searches. Dijkstra-based shortest path algorithms for multiple sources can make use of bundled searches where the searches for \( k \) sources are advanced simultaneously. A bundled search maintains \( k \) tentative distance labels at each vertex. The search is rooted at each of the \( k \) sources. Initially, the \( j \)-th distance label at the \( j \)-th source is set to 0, and all other \( kn-k \) distance labels are set to \( \infty \). As usual, vertices are settled by relaxing each outgoing edge. 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.
The concept of bundled searches was first introduced for Dijkstra searches used for the computation of arc-flags under the name centralized searches \cite{11}. Since then, bundled searches have been used in a number of Dijkstra-based many-to-many shortest path algorithms \cite{1,17,7,5,6}. More recently, the idea has been extended to point-to-point queries in CHs \cite{3}.

**Instruction-Level Parallelism in Bundled Searches.** Additionally, single-instruction multiple-data (SIMD) parallelism can be utilized for bundled searches \cite{3}. Modern CPUs provide special vector registers and instructions that can store and manipulate multiple data items simultaneously. We can vectorize the computations needed during edge relaxations s.t. \( k \) computations are performed at the same time using a single vector instruction. SIMD instructions can substantially speed up bundled searches \cite{3}.

### 3.2 LOUD

Our algorithm is based on the dynamic ridesharing dispatching algorithm LOUD \cite{4}.

Given a fleet of vehicles and routes, the online algorithm matches incoming ridesharing 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 as 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 passengers 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 \cite{4}.

**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 the 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 distances from last stops to the destination of a request, constitute a significant part (at least 60% and up to more than 90%) of the total runtime of LOUD.

### 4 Conceptual Changes for Multiple Pickup and Dropoff Locations

We observe that introducing passenger movement for pickup and dropoff locations requires a careful consideration of its effects on vehicle detours and passenger trips. In the following, we describe these effects in detail, leading us to the formal foundation of our cost function.

Remark that we ignore two special cases in this section for the sake of simplicity: First, we assume that the pickup and dropoff for an insertion \( i = (r, p, d, \nu, i, j) \) are inserted after the next stop of \( \nu \), i.e. \( i \geq 1 \). Second, we do not consider the possibility of \( p \) or \( d \) coinciding with existing stops, i.e. we assume \( p \neq l(s_i) \) and \( d \neq l(s_j) \). We ignore these cases as they
would lead to bloated definitions. However, with knowledge of the vehicle’s current location \(l_v(\nu)\) and the location \(l(s)\) of each stop \(s \in R(\nu)\), the ignored cases could be integrated into the following definitions in a straightforward manner.

### 4.1 Walking Time and Walking to the Destination

The walking time of a regular insertion \(i = (r, p, d, \nu, i, j)\) is simply \(t_{walk}(i) := t_{psg}(\text{orig}, p) + \delta_{psg}(d, \text{dest})\).

We allow each passenger \(r\) to walk from their origin to their destination without ever boarding a vehicle. This requires a manner of pseudo-insertion \(t_{psg}\) where the passenger is matched to no vehicle at all. Then, the cost of \(t_{psg}\) depends only on the walking distance \(t_{walk}(t_{psg}) = t_{trip}(t_{psg}) = \delta_{psg}(\text{orig}, \text{dest})\). The pseudo-insertion affects no vehicle operation times or trip times of other passengers. We ignore the wait time soft constraint since the passenger does not wait for a vehicle. In effect, the total cost is \(c(t_{psg}) = \tau \cdot t_{trip}(t_{psg}) + \omega \cdot t_{walk}(t_{psg}) + c_{\text{trip}}(t_{psg})\). We explicitly allow pseudo-insertions for any distance \(\delta_{psg}(\text{orig}, \text{dest})\), i.e., the distance does not have to be found within the radius \(\rho\) around \(\text{orig}\) or \(\text{dest}\). Instead, we use a CH query in the passenger graph to find \(\delta_{psg}(\text{orig}, \text{dest})\). The cost \(c(t_{psg})\) can serve as a first upper bound on the cost of any insertion.

### 4.2 Vehicles Waiting for Passengers

In the traditional dynamic ridesharing model, a request \(r = (\text{orig}, \text{dest}, t_{req})\) always waits to be picked up by the vehicle at the origin location \(\text{orig}\). The request is issued at time \(t_{req}\) and the vehicle \(\nu\) matched to the request can start making its way to the pickup location at the earliest at \(t_{req}\). If the pickup location is \(\text{orig}\) this means that the vehicle will always arrive at the pickup location later than the passenger, i.e., only the passenger can wait for the vehicle, not the other way around. In that case, the time a vehicle is stopped at any stop is always exactly \(t_{\text{stop}}^{\text{min}}\) (unless the vehicle is idling, i.e., it currently has no other stops to get to). Therefore, for each stop \(s\), the scheduled arrival time can be inferred from the scheduled departure time as \(t_{\text{arr}}^{\text{min}}(s) = t_{\text{dep}}^{\text{min}}(s) - t_{\text{stop}}^{\text{min}}\). In the traditional dynamic ridesharing model, it suffices to store the departure time at each stop for the full vehicle schedule [4]. The shortest path distance between two consecutive stops \(s_i\) and \(s_{i+1}\) is then \(\delta_{veh}(s_i, s_{i+1}) = t_{\text{dep}}^{\text{min}}(s_{i+1}) - t_{\text{stop}}^{\text{min}} - t_{\text{dep}}^{\text{min}}(s_i)\).

Now, consider an insertion \(i = (r, p, d, \nu, i, j)\) with \(p \neq \text{orig}\). Then, both the vehicle and the passenger have to travel to \(p\) starting at \(t_{\text{dep}}(s_i)\) and \(t_{req}\), respectively. Consequently, the vehicle can arrive at \(p\) earlier than the passenger, precisely if \(t_{\text{dep}}^{\text{min}}(s_i) + \delta_{veh}(s_i, p) < t_{req} + \delta_{psg}(\text{orig}, p)\). In that case, the vehicle needs to wait for the passenger at \(p\). An inserted stop \(s\) at \(p\) can then take longer than \(t_{\text{stop}}^{\text{min}}\). We, therefore, define the actual earliest possible departure time at each pickup as the maximum of the earliest possible departure time of the vehicle and that of the passenger at the pickup. The vehicle \(\nu\) can depart from \(p\) at the earliest after a stop time of \(t_{\text{stop}}^{\text{min}}\) and the passenger can depart as soon as they arrive at \(p\) leading us to

\[
t_{\text{dep}}(i) = \max(t_{\text{dep}}^{\text{min}}(s_i) + \delta_{veh}(s_i, p) + t_{\text{stop}}^{\text{min}}, t_{req} + \delta_{psg}(\text{orig}, p)).
\]

We later use \(t_{\text{dep}}(i)\) in the definitions of the vehicle detour, passenger wait time and passenger trip time needed for the cost function (see Section 2.4). In particular, the wait times of a vehicle or of a passenger contribute to the vehicle operation times and passenger trip times.

Whereas in the traditional model we can infer \(t_{\text{arr}}^{\text{min}}(s)\) from \(t_{\text{dep}}^{\text{min}}(s)\) for a stop \(s\), we have to now explicitly store \(t_{\text{arr}}^{\text{min}}(s)\) and \(t_{\text{dep}}(s)\). The distance between a pair of consecutive stops \((s_i, s_{i+1})\) can then be derived as \(\delta_{veh}(s_i, s_{i+1}) = t_{\text{arr}}^{\text{min}}(s_{i+1}) - t_{\text{dep}}^{\text{min}}(s_i)\).
We define the initial dropoff detour as that results from the vehicle movement we have then to take in its route after stop \( s_i \) and stop \( s_j \), respectively. We call these the initial pickup detour and initial dropoff detour.

**Definition 1.** The initial pickup (dropoff) detour for an insertion \( i = (r, p, d, ν, i, j) \) is the detour that results from the vehicle \( ν \) first driving to \( p \) (\( d \)) after stop \( s_i \) (\( s_j \)) instead of driving to \( s_{i+1} \) (\( s_{j+1} \)) directly.

Formally, we define the initial pickup detour as

\[
\Delta_p^{\text{init}}(i) := \begin{cases} 
  t_{\text{dep}}(i) - t_{\text{min}}^{\text{dep}}(s_i) & \text{if } i = j \\
  t_{\text{dep}}(i) - t_{\text{min}}^{\text{dep}}(s_i) + d_{\text{veh}}(p, s_{i+1}) - d_{\text{veh}}(s_i, s_{i+1}) & \text{if } i \neq j 
\end{cases}
\]

We define the initial dropoff detour as

\[
\Delta_d^{\text{init}}(i) := \begin{cases} 
  \delta_{\text{veh}}(p, d) + t_{\text{min}}^{\text{stop}} & \text{if } i = j \text{ and } j = k(\nu) \\
  \delta_{\text{veh}}(s_j, d) + t_{\text{min}}^{\text{stop}} & \text{if } i \neq j \text{ and } j = k(\nu) \\
  \delta_{\text{veh}}(p, d) + t_{\text{min}}^{\text{stop}} + \delta_{\text{veh}}(d, s_{j+1}) - \delta_{\text{veh}}(s_j, s_{j+1}) & \text{if } i = j \text{ and } j \neq k(\nu) \\
  \delta_{\text{veh}}(s_j, d) + t_{\text{stop}} + \delta_{\text{veh}}(d, s_{j+1}) - \delta_{\text{veh}}(s_j, s_{j+1}) & \text{if } i \neq j \text{ and } j \neq k(\nu) 
\end{cases}
\]

Note that if \( i = j \), the vehicle has to make a combined detour to go from stop \( s_i \) to \( p \), then to \( d \) and finally to \( s_{i+1} \). In that case, only the leg between \( s_i \) and \( s_{i+1} \) of the existing route \( R(\nu) \) is replaced by the combined detour. The definition of initial pickup and dropoff detour account for this by subtracting the distance between \( s_i \) and \( s_{i+1} \) only once in the dropoff detour and not in the pickup detour if \( i = j \).

**Residual Detours, Added Vehicle Operation Time.** Without vehicle wait times, the departure time \( t_{\text{dep}}(p) \) at pickup \( p \) is the arrival time of \( ν \) at \( p \) plus the minimum stopping time \( t_{\text{stop}}^{\text{min}} \) so \( t_{\text{dep}}(p) = t_{\text{min}}^{\text{dep}}(s_i) + \delta(s_i, p) + t_{\text{stop}}^{\text{min}} \) (see Section 4.2). The first effect that vehicle wait times can have on the detour is the fact that \( t_{\text{dep}}(p) \) can depend on the passenger if they arrive at the pickup later than the vehicle. As defined in Section 4.2 with passenger movement we have \( t_{\text{dep}}(p) = \max(t_{\text{min}}^{\text{dep}}(s_i) + \delta(s_i, p) + t_{\text{stop}}^{\text{min}}(r) + \delta_{\text{pass}}(\text{arr}, p)) \). Therefore, in a scenario with vehicle waiting times, the initial pickup detour \( \Delta_p^{\text{init}}(i) \) can be larger than without them.

Furthermore, existing vehicle wait times at stops in \( R(\nu) \) can also factor into the added vehicle operation time. Assume \( j + 1 < a < k(\nu) \), \( t_{\text{veh}}(s_a) > 0 \) and \( t_{\text{wait}}(s_b) = 0 \) for \( b \neq a \). Let \( t_{\text{arr}}^{\text{min}}(s, t) \) and \( t_{\text{dep}}^{\text{min}}(s, t) \) describe the scheduled arrival time and departure time.
at stop \( s \in R(\nu) \) after we perform insertion \( \iota \). After the insertion, vehicle \( \nu \) will arrive at stop \( s_{j+1} \) with a delay of \( \Delta_{j+1}^{\text{res}}(\iota) = \Delta_{j}^{\text{pinit}}(\iota) + \Delta_{j+1}^{\text{dinit}}(\iota) \) compared to before the insertion, so \( t_{\text{arr}}^{\min}(s_{j+1}) = t_{\text{arr}}^{\min}(s_{j+1}) + \Delta_{j+1}^{\text{res}}(\iota) \). This delay in arrival times is propagated forward through the route until stop \( s_{a} \), so for the delay arriving at \( s_{a} \) we have \( \Delta_{a}^{\text{res}}(\iota) = \Delta_{a+1}^{\text{res}}(\iota) \).

However, before the insertion, the vehicle has to wait for a duration of \( t_{\text{wait}}^{\text{veh}}(s_{a}) \) at stop \( s_{a} \) for a passenger that is scheduled to be picked up here. Since the vehicle now arrives at \( s_{a} \) later, the waiting time may decrease. If the delay at \( s_{a} \) is larger than the previous wait time at \( s_{a} \), the vehicle may even now arrive at \( s_{a} \) later than the passenger.

Effectively, the vehicle now uses the time that was spent waiting for the passenger at \( s_{a} \) to perform part of the detour needed for the pickup and dropoff of \( \iota \). Thus, the vehicle wait times at stops after the pickup or dropoff of an insertion act as buffers to the added vehicle operation time. In our example, we have

\[
\begin{align*}
t_{\text{arr}}^{\min}(s_{a}, \iota) &= t_{\text{arr}}^{\min}(s_{a}) + \Delta_{a}^{\text{res}}(\iota) \quad \text{but} \\
t_{\text{dep}}^{\min}(s_{a}, \iota) &= t_{\text{dep}}^{\min}(s_{a}) + \Delta_{a+1}^{\text{res}}(\iota)
\end{align*}
\]

Note that the vehicle wait time at \( s_{a} \) reduces the delay for the departure time at \( s_{a} \) and also the arrival times at all following stops. Formally, for every stop \( s_{b} \) with \( a < b \) we get

\[ \Delta_{a}^{\text{res}}(\iota) \leq \Delta_{b}^{\text{res}}(\iota) \] so \( t_{\text{arr}}^{\min}(s_{b}, \iota) = t_{\text{arr}}^{\min}(s_{a}) + \Delta_{a+1}^{\text{res}}(\iota). \)

This leads us to the definition of residual detours which describe the actual delay that an insertion \( \iota \) causes for the arrival time at any existing stop of the route \( R(\nu) \). It takes into account the possibility that a vehicle wait time may exist at any stop that reduces the delay for all later stops.

**Definition 2.** The residual detour \( \Delta_{a}^{\text{res}}(\iota) \) for an insertion \( \iota = (r, p, d, \nu, i, j) \) at stop \( s_{a} \in R(\nu) \) describes how much later the vehicle \( \nu \) will arrive at stop \( s_{a} \) after the insertion is performed. Formally, we define it as

\[
\Delta_{a}^{\text{res}}(\iota) := \begin{cases} 
0 & \text{if } a \leq i \\
\Delta_{a}^{\text{pinit}}(\iota) & \text{if } i + 1 = a \leq j \\
\max(\Delta_{j}^{\text{res}}(\iota) - t_{\text{wait}}^{\text{veh}}(s_{j}), 0) + \Delta_{d}^{\text{dinit}}(\iota) & \text{if } a = j + 1 \text{ and } i \neq j \\
\Delta_{a}^{\text{pinit}}(\iota) + \Delta_{d}^{\text{dinit}}(\iota) & \text{if } a = j + 1 \text{ and } i = j \\
\max(\Delta_{a-1}^{\text{res}}(\iota) - t_{\text{wait}}^{\text{veh}}(s_{a-1}), 0) & \text{otherwise}
\end{cases}
\]

Residual detours allow us to easily define the new arrival times \( t_{\text{arr}}^{\min} \) and departure times \( t_{\text{dep}}^{\min} \) after an insertion \( \iota \) at each stop \( s_{a} \in R(\nu) \) as

\[
\begin{align*}
t_{\text{arr}}^{\min}(s_{a}, \iota) &= t_{\text{arr}}^{\min}(s_{a}) + \Delta_{a}^{\text{res}}(\iota) \\
t_{\text{dep}}^{\min}(s_{a}, \iota) &= \max(t_{\text{arr}}^{\min}(s_{a}) + t_{\text{stop}}^{\min}(s_{a}), t_{\text{dep}}^{\min}(s_{a})).
\end{align*}
\]

Then, the added vehicle operation time can be defined as:

**Definition 3.** The added vehicle operation time \( t_{\text{detour}}(\iota) \) caused by an insertion \( \iota = (r, p, d, \nu, i, j) \) is defined as

\[
t_{\text{detour}}(\iota) := \begin{cases} 
\Delta_{a}^{\text{pinit}}(\iota) + \Delta_{d}^{\text{dinit}}(\iota) & \text{if } i = j = k(\nu) \\
\max(t_{\text{arr}}^{\min}(s_{k(\nu), \iota}) - t_{\text{arr}}^{\min}(s_{k(\nu)}), 0) + \Delta_{d}^{\text{dinit}}(\iota) & \text{if } i < j = k(\nu) \\
\max(t_{\text{arr}}^{\min}(s_{k(\nu), \iota}) - t_{\text{arr}}^{\min}(s_{k(\nu)})) & \text{otherwise}
\end{cases}
\]
Trip Time, Added Trip Time For Existing Passengers. The previous definitions also enable us to define the trip time $t_{\text{trip}}(i)$ of the passenger associated with the request $r$. The trip time for $r$ is simply the time between the request time $t_{\text{req}}(r)$ and the scheduled arrival of the passenger at their destination $\text{dest}(r)$. Chronologically, a trip starts with the passenger moving to the pickup point $p$ and possibly waiting for the vehicle to arrive at $p$. This is followed by the actual ride in the vehicle from $p$ to $d$. Finally, the passenger moves from the dropoff $d$ to their destination.

Definition 4. The trip time $t_{\text{trip}}(i)$ for an insertion $i = (r, p, d, \nu, i, j)$ is defined as

$$t_{\text{trip}}(i) := (t_{\text{dep}}(i) - t_{\text{req}}(r)) + t_{\text{ride}}(i) + \delta_{\text{psg}}(d, \text{dest}(r))$$

where

$$t_{\text{ride}}(i) :=
\begin{cases}
\delta_{\text{veh}}(p, d) & \text{if } i = j \\
\delta_{\text{veh}}(p, s_{i+1}) + (t_{\text{dep}}^{\min}(s_j, i) - t_{\text{arr}}^{\min}(s_{i+1}, i)) + \delta_{\text{veh}}(s_j, d) & \text{if } i \neq j
\end{cases}$$

The final contribution to the total cost of an insertion is the trip time that is added for passengers that are already assigned to vehicle $\nu$. Consider a request $r'$ that has previously been assigned to vehicle $\nu$. Assume $r'$ is picked up by $\nu$ at stop $s_i'$ and dropped off at stop $s_j'$. By definition of residual detours, performing the insertion $i$ will delay the arrival of the vehicle at $s_j'$ by $\Delta_{s_j'}(i)$. Therefore, the trip time of $r'$ increases by that same amount of time.

Definition 5. Let $N_d(s)$ be the number of dropoffs currently scheduled to be performed at stop $s \in R(\nu)$ for a vehicle $\nu \in F$. The combined added trip time for existing passengers $t_{\text{trip}}^+(i)$ caused by an insertion $i = (r, p, d, \nu, i, j)$ is defined as

$$t_{\text{trip}}^+(i) := \sum_{a=i+1}^{k(\nu)} N_d(s_a) \cdot \Delta_{a}(\nu)$$

5 Algorithm Overview

We introduce the KaRRi (Karlsruhe Rapid Ridesharing) algorithm that efficiently answers ridesharing requests with multiple PD-locations using fast many-to-many routing. 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 PD-locations 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 Figure 1. For each insertion, KaRRi computes the cost according to the cost function (see Section 2.1). The insertion with the smallest cost $i^*$ is repeatedly updated and eventually returned.

Since we consider sets of possible PD-locations, 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 each PD-location to filter out infeasible insertions and to determine the cost of the remaining candidate insertions. In the following sections, we describe the bundling and filtering methods that we employ to limit the running time of the required many-to-many shortest path queries for each insertion type.
6 Ordinary, Ordinary Paired, and Pickup Before Next Stop Insertions

This section is concerned with ordinary, ordinary paired, and pickup before next stop insertions (see Figure 1). In all three insertion types mentioned, both the pickup and dropoff insertion points \(i\) and \(j\) of an insertion \(s = (r, p, d, \nu, i, j)\). Illustrations depict the current route of \(\nu\) (solid arrows) with stops \(s \in R(\nu)\) as well as the detours to and from \(p\) and \(d\) (dashed lines).

### 6.1 Elliptic BCH Searches

We can extend elliptic BCH queries to multiple PD-locations by simply repeating the queries for each PD-location. Even with elliptic pruning, this can lead to impractical running times for large numbers of PD-locations, though. Therefore, we supplement elliptic BCH searches with two techniques for better scalability to larger numbers of PD-locations. We describe these techniques only for pickups but they work analogously for dropoffs.

**Elliptic BCH Searches with Sorted Buckets.** First, we propose using sorted buckets to reduce the number of bucket entries scanned by BCH queries. We explain the principle only for source buckets but it can be analogously applied for target buckets.

Recall that the constraints for existing passengers of a vehicle \(\nu\) define a leeway \(\lambda(s_i, s_{i+1})\) for the detour between any two consecutive vehicle stops \(s_i\) and \(s_{i+1}\) of \(\nu\) (see Section 3.2). When an elliptic BCH search to a pickup \(p\) scans a source bucket entry \((s, \delta^+(s_i, v)) \in B^+(\nu)\), the tentative distance \(\delta^+(s_i, v) + \delta^+(v, p)\) can only lead to an insertion that holds all hard constraints if \(\delta^+(s_i, v) + \delta^+(v, p) \leq \lambda(s_i, s_{i+1})\). This means the entry is only relevant for \(p\) if \(\delta^+(v, p) \leq \lambda(s_i, s_{i+1}) - \delta^+(s_i, v)\). We call \(\lambda_{\text{res}}(s_i, v) := \lambda(s_i, s_{i+1}) - \delta^+(s_i, v)\) the remaining leeway of a source bucket entry \((s_i, \delta^+(s_i, v))\) at vertex \(v\).

We sort the entries of each source bucket at each vertex \(v\) by their remaining leeway in decreasing order. Then, an elliptic BCH search to a pickup \(p\) can stop scanning the entries at
once an entry \((s, \delta^I(s, v))\) is scanned for which \(\delta^I(v, p) > \lambda_{res}(s, v)\). In this case, we have \(\delta^I(v, p) > \lambda_{res}(s^*, v)\) for any subsequent entry \((s', \delta^I(s', v))\), so the remaining entries cannot lead to any insertions that adhere to the hard constraints. Maintaining the order of each bucket comprises a time overhead when inserting bucket entries. However, since bucket sizes are small, this overhead is limited. Note that sorted buckets can also be applied in the case of only a single pickup.

**Bundled Elliptic BCH Searches.** Second, we employ **bundled elliptic BCH searches** that exploit the locality of pickups.

Like any bundled search, a bundled elliptic BCH search is rooted at \(k\) pickups and updates \(k\) distances with each edge relaxation (see Section 3.1). Additionally, we can bundle bucket entry scans. Whenever a bucket entry for a stop \(s\) is scanned, the search tries to improve upon any of the \(k\) tentative distances between \(s\) and any of the \(k\) pickups. We can effectively bundle the edge relaxations and bucket entry scans of elliptic BCH searches because the localized pickups share similar CH search spaces. Moreover, we can use vectorized instructions to parallelize both edge relaxations and bucket entry scans. At the same time, elliptic pruning and sorted buckets can still be applied. To our knowledge, our algorithm is the first to explicitly use bundled BCH searches. The idea follows from the bundled CH searches used in [3].

### 6.2 PD-Distance Searches

Computing the PD-distances, i.e. the distances between pickups and dropoffs, is a many-to-many shortest path problem where the set of sources and the set of targets are localized.

Our algorithm uses a BCH approach to address this problem. We generate bucket entries for all dropoffs in their reverse CH search spaces. Then, we run queries in the upward CH search graph rooted at each pickup to find the PD-distances using the dropoff bucket entries. We propose two methods to improve these BCH searches.

Firstly, let \(\delta^\text{max}_{PD}\) be an upper bound on all PD-distances. Then, we only have to generate and scan bucket entries in a radius of \(\delta^\text{max}_{PD}\). We use

\[
\delta^\text{max}_{PD} := \max_{p \in P} \delta_{veh}(p, orig) + \delta_{veh}(orig, dest) + \max_{d \in D_p} \delta_{veh}(dest, d).
\]

We can compute \(\delta_{veh}(p, orig)\) for all \(p \in P\) and \(\delta_{veh}(dest, d)\) for all \(d \in D_p\) using two local Dijkstra searches rooted at \(orig\) and \(dest\), respectively. We obtain \(\delta_{veh}(orig, dest)\) with a single preliminary CH query.

Secondly, we can once again use bundled BCH searches. More specifically, we can generate bucket entries for batches of \(k\) dropoffs and then run queries for batches of \(k\) pickups where \(k\) is a configuration parameter. Again, bundled PD-distance searches utilize the locality of pickups and dropoffs and allow us to employ SIMD parallelism.

### 6.3 Enumerating Ordinary, Ordinary Paired, and Pickup Before Next Stop Insertions

After running our elliptic BCH queries and PD-distance searches, we know all distances that are required for ordinary and **ordinary paired** insertions. We enumerate the insertions \(i = (r, p, d, \nu, i, j)\) with \(0 < i < j < k(\nu)\) for a set of candidate vehicles found by the elliptic BCH queries [4]. We compute the cost \(c(i)\) for each insertion and update \(i^*\) to \(i\) if \(c(i) < c(i^*)\).

To compute the cost of a **pickup before next stop** (PBNS) insertion \(i = (r, p, d, \nu, 0, j)\), we lack knowledge of the distance \(\delta_{veh}(l_r(\nu), p)\) from the vehicle’s current location \(l_r(\nu)\) to
the pickup $p$. LOUD suggests filtering PBNS insertions based on a lower bound on the cost of $\ell$ obtained from considering $\delta_{ech}(s_{\ell}, p)$ instead of $\delta_{ech}(s_{\ell}, l_{\ell}(\nu)) + \delta_{ech}(l_{\ell}(\nu), p)$ for the pickup detour $\mathcal{H}$. This filter discards almost all PBNS insertions. For the remaining PBNS insertions, we compute the missing distances $\delta_{ech}(l_{\ell}(\nu), p)$ with a bucket based approach. We generate source bucket entries for the current location of every affected vehicle and run bundled queries from the pickups. The average number of such queries per request is less than 0.5.

## 7 Pickup After Last Stop Insertions

In this section, we consider pickup after last stop (PALS) insertions. The main challenge of PALS insertions is the computation of the distances from last stops to pickups. The authors of LOUD find that elliptic pruning is not applicable for the computation of these distances $\mathcal{H}$. Instead, LOUD uses a reverse Dijkstra search rooted at $\text{orig}$ that is stopped early when the search can no longer find an insertion better than the best known one. For multiple pickups, we can compute the required distances by analogously running reverse Dijkstra searches for each pickup. These Dijkstra searches may also be bundled to exploit the locality of the pickups.

However, even with a single pickup, this Dijkstra search takes up a significant part of the running time of the LOUD algorithm. Thus, for a large number of pickups, we expect infeasible running times. In this section, we introduce two new BCH based approaches for the computation of last stop distances. For the rest of this section, let $\hat{\ell}$ denote an upper bound on the best known insertion cost (initially $\hat{\ell} := c(\ell^*)$).

### Reformulation of Cost Function for PALS Insertions.

Note that the cost of any PALS insertion $\ell = (r, p, d, \nu, k(\nu), k(\nu))$ is fully characterized by the pickup $p$, the PD-distance $\delta_{ech}(p, d)$, the walking distance $\delta_{psg}(d, \text{dest})$, the departure time $t_{\text{dep}}^{\text{min}}(s_{k(\nu)})$ of $\nu$ at $s_{k(\nu)}$, and the last stop distance $\delta_{ech}(s_{k(\nu)}, p)$. Thus, we can write the cost of $\ell$ as

$$c(\ell) = c'(r, p, d, \delta_{ech}(p, d), \delta_{psg}(d, \text{dest}), t_{\text{dep}}^{\text{min}}(s_{k(\nu)}), \delta_{ech}(s_{k(\nu)}, p)).$$

### 7.1 Last Stop BCH Searches for PALS

Even though elliptic pruning is not applicable, we can still employ a BCH search approach for distances from last stops to pickups. For this, we maintain a last stop bucket $B^1_{\text{last}}(v)$ for every $v \in V$. For every last stop $s_{k(\nu)}$, we generate an entry $(s_{k(\nu)}, \delta^1(s_{k(\nu)}, v)) \in B^1_{\text{last}}(v)$ at each vertex $v$ in the upward CH search space rooted at $s_{k(\nu)}$. Then, for every pickup $p \in P_p$, we run an individual (last stop) BCH query in $G^2_p$ that scans the last stop bucket at each settled vertex to compute the shortest path distances from last stops to $p$. When the search scans an entry $(s_{k(\nu)}, \delta^1(s_{k(\nu)}, v)) \in B^1_{\text{last}}(v)$, it tries to improve the tentative distance $\delta(s_{k(\nu)}, p)$ with $\delta'(s_{k(\nu)}, v) + \delta^1(v, p)$. Eventually, the shortest distance $\delta_{ech}(s_{k(\nu)}, p)$ will be found for every last stop $s_{k(\nu)}$.

Whenever the last stop of a vehicle changes, we run two forward CH searches to remove the bucket entries of the old last stop and insert new entries for the new last stop. We stop the search early when the current distance no longer admits a PALS insertion with cost smaller than the best known cost. Furthermore, we can bundle the BCH queries and use SIMD parallelism in a similar manner to bundled elliptic BCH searches (see Section 6.1).

### Cost Pruning of Bucket Scans using Sorted Buckets.

A remaining issue of this approach is the size of the last stop buckets. Without elliptic pruning, buckets contain many more
Thus, since the entries of label \( l \) cost of any insertion, our search proceeds with a next step. In each step of the search, the \( d \) search maintains a priority queue that has not been settled yet. For each open label \( p \) \( Open \) and Closed Labels.

Based on label domination. \( PD-pairs \) are propagated through the \( CH \) search graph and how these labels can be pruned as the last stop distance other. We introduce a collective BCH query that finds the best PALS insertion

Obviously, we do not know the distance between every last stop and every pickup. If we knew the best PALS

Finally, we propose a search approach based on the idea that we do not actually need to

7.2 Collective Last Stop Searches for PALS

Finally, we propose a search approach based on the idea that we do not actually need to know the distance between every last stop and every pickup. If we knew the best PALS insertion \( \hat{c}^* \) in advance, we would only need to find \( \hat{c} \) in the search. Each tentative distance \( \delta(s_{k(v)}, p) \) found acts as an upper bound on the actual shortest distance \( \delta_{v,p}(s_{k(v)}, p) \). Thus, whenever the tentative distance \( \delta(s_{k(v)}, p) \) is updated, we can compute an upper bound

\[
c_{\text{max}} = c^*(r, p, \delta_{v,p}(p, \text{dest}), 0, t_{\text{dep}}(s_{k(v)}), \delta(s_{k(v)}, p))
\]

on the cost of the best PALS insertion with \( v \) and \( p \). We update \( \hat{c} \) to \( c_{\text{max}} \) if \( c_{\text{max}} < \hat{c} \). This technique finds inexact cost upper bounds early which is helpful for the stopping criterion of bucket scans.

Open and Closed Labels. A PD-pair label \( (p, d, \delta^*(v, p)) \) at a vertex \( v \in V \) consists of the pickup \( p \in P_p \), dropoff \( d \in D_p \) and downwards distance \( \delta^*(v, p) \). At each vertex \( v \in V \), there is a set of open labels \( \text{open}(v) \) and a set of closed labels \( \text{closed}(v) \). An open label is a label that has not been settled yet. For each open label \( l = (p, d, \delta^*(v, p)) \), we store a lower bound \( c_{\text{min}}(l) \) for the cost of a PALS insertion that can be found for \( l \) in \( G_v^* \)

\[
c_{\text{min}}(l) := c^*(r, p, \delta_{v,p}(p, d), \delta_{\text{psg}}(d, \text{dest}), t_{\text{req}}(r), \delta^*(v, p))
\]

Algorithm Outline. We give pseudocode for a collective BCH search in Algorithm \[1\]. Our search maintains a priority queue \( Q \) that contains all open labels ordered increasingly by \( c_{\text{min}} \). Initially, at each pickup \( p \in P_p \), an open label \( (p, d, 0) \in \text{open}(p) \) is created for each \( d \in D_p \). As long as \( Q \) contains a label \( l \) with \( c_{\text{min}}(l) \leq \hat{c} \) for a known upper bound \( \hat{c} \) on the cost of any insertion, our search proceeds with a next step. In each step of the search, the label \( l := \min(Q) \) is removed from \( Q \) and settled as described in the following.
1: **procedure** COLLECTIVE BCH($P_ρ$, $D_ρ$, $G^1 = (V^1, E^1)$) : $(p^*, d^*)$ and $δ_{veh}(s_κ(v), p^*)$
2: $Q :=$ PQ of labels with key$Q(l) = c_{min}(l)$; open($v) := \emptyset$, closed($v) := \emptyset$ for $v \in V$
3: $\hat{c} := c(l^*)$
4: for each $(p, d) \in P_ρ \times D_ρ$
5:   insertLabelAtVertex($p$, $(p, d, 0)$)
6: while $Q \neq \emptyset$ do
7:   $l := Q.deleteMin()$
8:   if $c_{min}(l) > \hat{c}$ then return
9:   settleLabel($l$)

10: **procedure** SETTLE LABEL($l = (p, d, δ^i(v, p))$
11:   open($v$).remove($l$); closed($v$).insert($l$) \> mark $l$ closed
12:   for each $e = (s_κ(v), δ^i(s_κ(v), v)) \in B^1_{last}(v)$ do \> scan last stop entries at $v$ for $l$
13:     if $c_{min}(l, e) > \hat{c}$ then break
14:     if $c_{max}(l, e) < \hat{c}$ then
15:       $(p^*, d^*) := (p, d)$; $\hat{c} := c_{max}(l, e)$
16:   for each $(u, v) \in E^i$ do \> propagate $l$ to neighbors of $v$
17:     insertLabelAtVertex($u$, $(p, d, t^+ (u, v) + δ^i(v, p))$

18: **procedure** INSERTLABELATVERTEX($v, l' = (p, d, δ^i(v, p))$
19:   if $c_{min}(l') > \hat{c}$ then return
20:   for each $l \in open(v) \cup closed(v)$ do
21:     if $l$ dominates $l'$ then return
22:   for each $l \in open(v)$ do
23:     if $l'$ dominates $l$ then open($v$).remove($l$)
24:     open($v$).insert($l'$); $Q$ insert($l'$)

**Settling Open Labels.** Setting an open label $l = (p, d, δ^i(v, p))$ consists of three steps: First, we mark $l$ closed at $v$, i.e. we move $l$ from open($v$) to closed($v$). Second, we search for a new best insertion by traversing all entries in the last stop bucket $B^1_{last}(v)$. For each entry $e = (s_κ(v), δ^i(s_κ(v), v)) \in B^1_{last}(v)$, we compute an upper bound cost

$$c_{max}(l, e) := c'(r, p, δ_{veh}(p, d), δ_{psg}(d, dest), t_{min}^{dest}(s_κ(v)), δ^i(s_κ(v), v) + δ^i(v, p)).$$

If $c_{max}(l, e) < \hat{c}$, we mark $l = (r, p, d, ν, k(v), k(ν))$ as the best known PALS insertion, store the tentative distance $δ(s_κ(v), p) = δ^i(s_κ(v), v) + δ^1(v, p)$, and update $\hat{c} := c_{max}(l, e)$. Note that $c_{max}(l, e)$ is the exact cost of the PALS insertion $l = (r, p, d, ν, k(v), k(ν))$ if $δ(s_κ(v), p)$ is a shortest path distance. Since the BCH search finds shortest up-down paths, we will thus eventually find the best PALS insertion. As before, we can stop each bucket scan early. For this purpose, we compute a vehicle-independent cost lower bound $c_{min}(l, e)$ s.t. we can stop the search early if $c_{min}(l, e) > \hat{c}$ using

$$c_{min}(l, e) := c'(r, p, δ_{veh}(p, d), δ_{psg}(d, dest), t_{req}(r), δ^i(s_κ(v), v) + δ^i(v, p)).$$

Third, we propagate $l$ to all neighboring vertices of $v$. For each neighboring vertex $w \in V$ with $(w, v) \in G^1$, we create a new open label $l' = (p, d, t^+ (w, v) + δ^i(v, p))$ at $w$. Here, we
employ cost pruning by discarding \( l' \) if the lower bound cost \( c_{\min}(l') \) for this PD-pair and this distance exceeds \( \hat{c} \). Furthermore, we may be able prune \( l' \) at \( v \) if it is dominated by an existing label at \( v \) as described in the following.

**Domination Pruning.** Propagating a label through the entire search space for every PD-pair is too expensive. However, we find that we can compare labels at the same vertex and prune dominated labels in a technique we call domination pruning. Intuitively, a label \( l \) dominates a label \( l' \) at a vertex \( v \) if we know that any insertion found in \( G_k^v \) that uses \( l' \) has higher costs than the equivalent insertion using \( l \).

To formalize this, we first define an upper bound for the cost of a PALS insertion found in \( G_k^v \) for a label \( l \). Let \( l = (p, d, \delta^i(v, p)) \) be a label at \( v \) and \( e = (s_{k(v)}, \delta^i(s_{k(v)}, w)) \in B_{last}^i(w) \) a last stop bucket entry at a vertex \( w \in V_k^i \). Then, we define an upper bound for the cost of an insertion where \( v \) drives from \( s_{k(v)} \) to \( p \) via \( w \) and \( v \) as

\[
c_{\max}(l, v, e) := c'(r, p, \delta_{veh}(p, d), \delta_{psg}(d, dest), t_{\min}^{\bull}(s_{k(v)}), \delta^i(s_{k(v)}, w) + \delta^i(w, v) + \delta^i(v, p))
\]

With this, we can formally define the domination relation between labels:

**Definition 6.** A PD-pair 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 \( w \in V_k^i \) and \( e \in B_{last}^i(w) \).

**Theorem 7.** If a label \( l \) dominates another label \( l' \) at \( v \), we do not need to settle \( l' \) at \( v \).

**Proof.** This can be shown by contradiction. Assume \( l = (p, d, \delta^i(v, p)) \) dominates \( l' = (p', d', \delta^i(v, p')) \) at \( v \). Further, assume that \( \nu = (r, p', d', \nu, k(v), k(v)) \) is the best PALS insertion. Let \( \pi \) be a shortest path from \( s_{k(v)} \) to \( p' \). Wlog, \( \pi \) is an up-down-path in the CH consisting of an upwards prefix \( \pi^+ \) and a downwards suffix \( \pi^- \). If \( \pi^- \) does not contain \( v \), then the collective search will not find \( \pi \) in \( G_k^v \), and we do not have to settle \( l' \) at \( v \).

Otherwise, \( \pi^- = (w, \ldots, v, \ldots, p') \) with \( w \in V_k^i \). Let \( e = (s_{k(v)}, \delta^i(s_{k(v)}, w)) \in B_{last}^i(w) \). Since \( \pi \) is a shortest path, we know that

\[
c_{\max}(l', v, e) = c'(r, p', \delta_{veh}(p', d'), \delta_{psg}(d', dest), t_{\min}^{\bull}(s_{k(v)}), \delta_{veh}(s_{k(v)}, p'))
\]

However, \( l \) dominates \( l' \) which means that

\[
c((r, p, d, \nu, k(v), k(v))) \leq c_{\max}(l, v, e) < c_{\max}(l', v, e) = c((r, p', d', \nu, k(v), k(v)))
\]

This contradicts \( \nu \) being the best PALS insertion. Hence, we do not have to settle label \( l' \) at \( v \) to find the best pair for \( \nu \).

**Efficiently Computing the Domination Relation.** We find that it is not trivial to compute the domination relation efficiently because of the non-linearity of the cost function.

Consider two labels \( l_i = (p_i, d_i, \delta^i(v, p_i)) \) for \( i = 1, 2 \) and two PALS insertions \( \nu_i = (r, p_i, d_i, \nu, k(v), k(v)) \) found for these labels in \( G_k^v \). Assume the vehicle \( \nu \) arrives at \( v \) at some time \( t = t_{\min}^{\bull}(s_{k(v)}) + \delta_{veh}(s_{k(v)}, v) \). If we know \( t \), we can determine the cost difference \( c(\nu_1) - c(\nu_2) \) irrespective of the actual vehicle \( \nu \). In other words, the difference \( \Delta_c(l_1, l_2, t) \) in costs between any insertions that can be found for \( l_1 \) and \( l_2 \) in \( G_k^v \) is a function of \( t \). Then, if \( \Delta_c(l_1, l_2, t) < 0 \) for all \( t \geq 0 \), every insertion found in \( G_k^v \) will be better with \( l_1 \) than with \( l_2 \), i.e. \( l_1 \) dominates \( l_2 \).

If the cost function for PALS insertions were to grow linearly with \( t \), then \( \Delta_c(l_1, l_2, t) \) would be constant wrt. \( t \). In that case, we could simply check for domination using \( \Delta_c(l_1, l_2, 0) < 0 \). However, the cost function does not increase linearly with \( t \): Firstly, the
cost is constant wrt. $t$ as long as the vehicle arrives at $p_i$ earlier than the passenger, i.e. $t + \delta(v, p_i) + \tau_{\text{stop}} \leq \tau_{\text{reg}}(r) + \delta_{\text{psg}}(\text{orig}, p_i)$ (see Section 4.2). Secondly, due to the wait time and trip time soft constraints, linear penalty terms are added to the cost function starting at a certain threshold for the wait time and trip time, both of which $t$ contributes to.

The passenger arrival time at $p_i$ and the thresholds for soft constraint penalties differ between labels. Thus, $\Delta_c(l_1, l_2, t)$ varies with $t$. Since we do not know which values of $t$ are possible for insertions found in $G^v$, we cannot trivially determine whether $l_1$ dominates $l_2$.

**Approximating the Domination Relation.** Instead, we under-approximate the domination relation by computing a sufficient precondition. For this purpose, we find an upper bound $\Delta_c^{\max}(l_1, l_2) \geq \max_{t \geq 0} \Delta_c(l_1, l_2, t)$ on the difference in insertion costs between any insertion that can be found for $l_1$ and $l_2$ in $G^v$. Then, $l_1$ dominates $l_2$ if $\Delta_c^{\max}(l_1, l_2) < 0$.

We now explain how to find this upper bound: The maximum difference in the departure times at $p_1$ and $p_2$ is $t_{\text{max}}^{\text{dep}}(l_1) - t_{\text{min}}^{\text{dep}}(l_2)$ with

$$t_{\text{max}}^{\text{dep}}(l_1) := \max\{\delta^v(p_1,\text{orig}) + t_{\text{stop}}^{\text{min}}\delta_{\text{psg}}(p_1,\text{orig})\}$$

$$t_{\text{min}}^{\text{dep}}(l_2) := \delta^v(p_2) + t_{\text{stop}}$$

Then, for every insertion found in $G^v$, the difference in detours between $l_1$ and $l_2$ is bounded by $\Delta_{\text{detour}}(l_1, l_2) := t_{\text{det}}^{\text{max}}(l_1) + \delta_{\text{veh}}(p_1, d_1) - t_{\text{det}}^{\text{min}}(l_2) - \delta_{\text{veh}}(p_2, d_2)$. For the difference in trip times, we first define:

$$t_{\text{arr}}^{\text{max}}(l_1) := t_{\text{dep}}^{\text{max}}(l_1) + \delta_{\text{veh}}(p_1, d_1) + \delta_{\text{psg}}(d_1, \text{dest})$$

$$t_{\text{arr}}^{\text{min}}(l_2) := t_{\text{dep}}^{\text{min}}(l_2) + \delta_{\text{veh}}(p_2, d_2) + \delta_{\text{psg}}(d_2, \text{dest})$$

Then, the difference in trip times is bounded by $\Delta_{\text{trip}}(l_1, l_2) := t_{\text{arr}}^{\text{max}}(l_1) - t_{\text{arr}}^{\text{min}}(l_2)$. We define upper bounds for the difference in penalties for the wait and trip time soft constraints as

$$\Delta_{\text{wait}}(l_1, l_2) := \gamma_{\text{wait}} \max\{t_{\text{dep}}^{\text{max}}(l_1) - t_{\text{dep}}^{\text{min}}(l_2), 0\}$$

$$\Delta_{\text{trip}}^{\text{max}}(l_1, l_2) := \gamma_{\text{trip}} \max\{\Delta_{\text{trip}}(l_1, l_2), 0\}$$

Note that the differences in detours and trip times are allowed to be negative to express a cost advantage for $l_1$ but the differences in penalties are not. Even if $t_{\text{dep}}^{\text{max}}(l_1) < t_{\text{dep}}^{\text{min}}(l_2)$ or $\Delta_{\text{trip}}(l_1, l_2) < 0$, we may find insertions in $G^v$ where no penalties apply for either label. In those cases, the penalty difference has to be zero. Let $\Delta_{\text{walk}}(l_1, l_2)$ be the fix difference in walking costs. Putting it all together, we get

$$\Delta_c^{\max}(l_1, l_2) := \Delta_{\text{detour}}(l_1, l_2) + \tau_{\text{trip}}(l_1, l_2) + \omega_{\text{walk}}(l_1, l_2) + \Delta_{\text{wait}}(l_1, l_2) + \Delta_{\text{trip}}^{\text{max}}(l_1, l_2)$$

We can compute $\Delta_c^{\max}(l_1, l_2)$ in constant time with information that is known at $v$ without looking ahead in the search tree. Since we under-approximate domination, it is possible for $l_1$ actually dominates $l_2$ but our condition does not hold. However, we find that our domination criterion still manages to prune the vast majority of labels early.

**Limitations.** We remark that the insertion $c$ found by the collective search is only guaranteed to be the best possible PALS insertion if $c$ holds the service time hard constraint. Since our search ignores the service time constraint, it may return an insertion that breaks the constraint even if there are other PALS insertions that do not.

Therefore, if $c$ breaks the service time constraint, we fall back to computing the distances from every last stop to every pickup using individual last stop BCH searches. The fallback individual BCH searches can make use of the good cost upper bounds found during the collective search. We find that this is only necessary in exceedingly rare cases.
The final insertion type are dropoff after last stop (DALS) insertions (see Figure 1). Similarly to PALS insertions, we face the problem of not knowing the distances from the vehicles’ last stops to dropoffs. We can use the same approaches of Dijkstra searches as well as individual and collective BCH searches to find these distances with some minor differences.

Firstly, cost pruning is less effective than in the PALS case since the lower bounds on costs cannot include the PD-distance. Secondly, we cannot update the global cost upper bound \( \hat{c} \) during the searches in the DALS case as we lack information about the cost of inserting the pickup earlier in the route. Thirdly, collective BCH searches have some more intricate differences between the PALS and DALS cases. We go into more detail about these differences in the rest of this section.

**Pareto-Best Dropoffs.** The largest difference of collective searches in the DALS case compared to the PALS case is the goal of the search. Whereas collective PALS searches always find a single best PD-pair, we cannot find a single best dropoff. Instead, we find a set of pareto-best dropoffs per vehicle.

To see why there cannot always be a single best dropoff for a vehicle \( \nu \in F \), assume that we would like to decide whether \( d_1 \in D_\rho \) always leads to better DALS insertions than \( d_2 \in D_\rho \). Dropoff \( d_1 \) is always better than \( d_2 \) for \( \nu \) if \( c(\iota_1) < c(\iota_2) \) for any DALS insertions \( \iota_1 = (\nu, d_1, 1, \nu, k(\nu)) \) and \( \iota_2 = (\nu, d_2, 0, i, k(\nu)) \) for any pickup \( p \in P_\rho \) and \( 0 \leq i < k(\nu) \).

Consider the cost difference \( c(\iota_1) - c(\iota_2) \). The vehicle detour and trip time is the same for both insertions up to \( s_{k(\nu)} \). Thus, the differences between \( \iota_1 \) and \( \iota_2 \) in the detours, trip times, added trip times for existing passengers, walking times, and penalties for the wait time constraint are all entirely independent of the choice of \( p \) and \( i \). However, for different choices of \( p \) and \( i \), either insertion may or may not violate the trip time soft constraint. Thus, it is possible that \( \iota_1 \) is better for some choice of \( p \) and \( i \) while \( \iota_2 \) may better for another.

To illustrate this issue, consider the example depicted in Figure 2. Suppose \( \tau = 1 \) and \( \gamma_{\text{trip}} = 10 \). Further, let the maximum trip time \( t_{\text{max}}(r) - t_{\text{req}}(r) = 10 \). Let \( \delta_{\text{p}}(\text{orig}, p_1) < 3 \) as well as \( \delta_{\text{p}}(\text{orig}, p_2) < 3 \).

Consider the insertions \( \iota_{1i} = (\nu, p_1, d_i, \nu, k(\nu)) \) for \( i = 1, 2 \). Neither insertion violates the trip time soft constraint as \( t_{\text{trip}}(\iota_{11}) = 3 + 1 + 4 = 9 \) and \( t_{\text{trip}}(\iota_{12}) = 3 + 1 + 3 + 1 = 8 \). The difference in detours is \( t_{\text{detour}}(\iota_{11}) - t_{\text{detour}}(\iota_{12}) = 5 - 7 = -2 \), leading to a difference in costs of \( c(\iota_{11}) - c(\iota_{12}) = -1 \). Therefore, \( d_1 \) is the better choice for \( p_1 \).

Now, consider the insertions \( \iota_{2i} = (\nu, p_2, d_i, \nu, k(\nu)) \) for \( i = 1, 2 \). Here, both insertions violate the trip time soft constraint since \( t_{\text{trip}}(\iota_{21}) = 12 \) and \( t_{\text{trip}}(\iota_{22}) = 11 \). The difference in detours and trip times add up like before but with the added difference in...
Table 1 Key figures of our benchmark instances.

| Instance    | |V| | |E| | #veh. | #req. |
|-------------|----|-----|-----|-----|-----|-----|-----|
| Berlin-1pct | 73689 | 159039 | 1000 | 16569 |
| Berlin-10pct| 73689 | 159039 | 10000 | 149185 |
| Ruhr-1pct   | 394049 | 840587 | 3000 | 49707 |
| Ruhr-10pct  | 394049 | 840587 | 30000 | 447555 |

Table 2 Average number of pickups (N_p) and dropoffs (N_d) for specific values of the walking radius ρ on the Berlin-1pct, Berlin-10pct, Ruhr-1pct, and Ruhr-10pct instances.

<table>
<thead>
<tr>
<th>ρ</th>
<th>B-1%</th>
<th>B-10%</th>
<th>R-1%</th>
<th>R-10%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N_p</td>
<td>N_d</td>
<td>N_p</td>
<td>N_d</td>
</tr>
<tr>
<td>0s</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td>1 1 1</td>
<td></td>
</tr>
<tr>
<td>150s</td>
<td>10 10 10 10</td>
<td>12 11 12 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>300s</td>
<td>33 33 33 33</td>
<td>35 33 35 33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>450s</td>
<td>68 66 68 67</td>
<td>70 68 70 68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>600s</td>
<td>115 113 115 114</td>
<td>114 112 114 112</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

penalties, we get c(t_{21}) − c(t_{22}) = −1 + 10 = 9. Thus, d_2 is the better choice for p_2.

This demonstrates that we may not be able to choose a single best dropoff for DALS insertions. Instead, we only know that a dropoff d_1 always leads to better insertions than a dropoff d_2 if d_1 is better for DALS insertions with and without a trip time violation. Thus, we can only ever obtain a set of pareto-best dropoffs for each vehicle.

Algorithm Outline and Partial Domination. Our collective search maintains open and closed dropoff labels of the form (d,δ↓(v,d)) at each vertex v ∈ V with d ∈ Dρ. Initially, an open label (d,0) is created at every d ∈ Dρ. As in the PALS case, we associate a lower bound cost c_{min}(l) with each open label l. In each step, the open label l with the smallest lower bound cost is settled by closing the label at its vertex v, scanning the last stop bucket at v, and propagating l to neighboring vertices in G↓v.

We use our cost based stopping criteria for bucket scans and the entire search. When a label is propagated to a new vertex, we apply domination pruning. For the domination relation, we test whether a dropoff will always be better than another dropoff with and without trip time penalties:

Definition 8. Let l_i = (d_i,δ↓(v,d_i)) for i = 1, 2 be two dropoff labels at v ∈ V. Let

Δ{detour}(l_1, l_2) := δ↓(v,d_1) − δ↓(v,d_2)

Δ{trip}(l_1, l_2) := δ↓(v,d_1) + δ_{psg}(d_1, dest) − δ↓(v,d_1) − δ_{psg}(d_2, dest)

Then d_1 dominates d_2 if
1. Δ{detour}(l_1, l_2) + τΔ{trip}(l_1, l_2) < 0 and
2. Δ{detour}(l_1, l_2) + (τ + γ_{trip})Δ{trip}(l_1, l_2) < 0

9 Experimental Evaluation

Our source code is written in C++17 and compiled with GCC 9.4 using -O3. We check the correctness of our implementation with a rigorous use of assertions (disabled for experiments). We conduct our experiments on a machine running Ubuntu 20.04 with 512 GiB of memory.
and two 16-core Intel Xeon E5-2670 v3 processors at 2.3GHz. 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 our implementation on the Berlin-1pct (B-1%), Berlin-10pct (B-10%), Ruhr-1pct (R-1%), and Ruhr-10pct (R-10%) request sets [4] that respectively represent 1% and 10% of ridesharing demand in the Berlin and Rhein-Ruhr metropolitan areas on a weekday. The request sets for Berlin were artificially generated based on the Open Berlin Scenario [19] for the MATSim transport simulation [12]. The request sets for the Rhein-Ruhr area were obtained by scaling up the Berlin request sets [4]. The underlying road networks are obtained from OpenStreetMap data [1]. We use the known speed limit of each road to determine the travel time of the according edge in the vehicle network. For the passenger network, we assume a constant walking speed of 4.5 km/h. We show the sizes of the networks and request sets in Table 1. We scale the number of pickups $N_p$ and dropoffs $N_d$ by using increasing walking radii $\rho \in \{0s, 150s, 300s, 450s, 600s\}$ which lead to the numbers of PD-locations given in Table 2. Unless stated otherwise, we run 5 iterations per combination of algorithm configuration, input, and radius $\rho$ and report the average running times.

For our model parameters, we choose $t_{\text{max wait}} = 600s$, $t_{\text{min stop}} = 60s$, $\gamma_{\text{wait}} = 1$, $\gamma_{\text{trip}} = 10$, $\tau = 1$, and $\omega = 0$. We use the open-source library RoutingKit [2] to compute the required CHs of the road networks which takes only a few seconds for our instances. This pre-processing step takes only a few seconds for each graph on any of our input instances.

**Figure 3** Evaluation of bundled elliptic BCH searches (left) and PD-distance searches (right). Shows mean search times (in ms) for either search on the Berlin-1pct and Berlin-10pct instances with $\rho \in \{0s, 150s, 300s, 450s, 600s\}$. Considers $k \in \{1, 8, 16, 32\}$ for elliptic BCH searches and $k \in \{1, 16, 32, 64\}$ for PD-distance searches, using SIMD instructions for $k > 1$. Additionally shows running times without SIMD instructions for elliptic searches with $k = 16$ and PD-distance searches with $k = 32$. Note the different x- and y-axes.
9.1 Bundled Searches

In this section, we experimentally evaluate bundled searches in each of the described applications and find the optimal value of $k$ for each of them. We conduct our experiments on the Berlin-1pct and Berlin-10pct instances with $\rho \in \{0s, 150s, 300s, 450s, 600s\}$. Due to time constraints, we first find the optimal value of $k$ with vector instructions and then evaluate the running time without vector instructions only for that value of $k$. We remark, though, that the optimal values of $k$ may differ depending on whether vector instructions are used at all and which SIMD instruction set is available. We only run a single iteration of Dijkstra searches because of their prohibitive running times.

Bundled Elliptic BCH Searches. We show experimental running times for bundled elliptic BCH searches with $k \in \{1, 8, 16, 32\}$ in Figure 3. We find that $k = 16$ is best suited here.

---

1. https://download.geofabrik.de/
2. https://github.com/RoutingKit/RoutingKit
Because of the use of shortcut edges, the search trees of each source in the CH only start to overlap at larger distances from the sources. Therefore, we can bundle edge relaxations and bucket scans in the periphery of the sources but not closer to each source. Sorted buckets additionally shift the focus of the work closer to the sources where large values of \( k \) are ineffective. The value of \( k = 16 \) strikes a balance between the two aspects of limiting the overhead closer to the sources while bundling operations further away.

The advantage of \( k = 16 \) over \( k = 32 \) is more pronounced on the Berlin-10pct instance. We attribute this to the fact that there are more vehicles in total which means that buckets contain more entries on average. Thus, with sorted buckets, the focus on work closer to the sources is more pronounced.

Bundled searches with \( k = 16 \) and without SIMD parallelism (no SIMD) lead to only small speedups. Again, bundling works better for the smaller instance due to a larger relative part of the searches' work being performed further away from the sources.

**Bundled PD-Distance Searches.** In Figure 3, we show the running times for bundled PD-distances searches with \( k \in \{1, 16, 32, 64\} \). Since our PD-distance BCH searches perform more work further away from the sources than elliptic BCH searches, the search trees of all searches overlap more. Thus, larger values of \( k \) allow effective bundling of edge relaxations and the generation and scan of bucket entries. Nonetheless, the potential for bundling is limited and larger values of \( k \) lead to additional overheads closer to the sources. In effect, \( k = 32 \) is the best choice for our PD-distance searches.

We additionally consider the running time at \( k = 32 \) without SIMD instructions (no SIMD). We observe speed-ups of up to 2.2 even without SIMD parallelism. This is again due to the larger amount of work performed in the periphery of the sources that can be bundled well.

**Bundled Last Stop Searches.** We depict the running times of bundled Dijkstra searches and individual BCH searches for the PALS and DALS cases in Figure 4.

We find that Dijkstra searches are well suited for bundling as we observe the smallest search times with \( k = 64 \). 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 we see good speedups for bundled Dijkstra searches even without SIMD instructions. Larger \( k \geq 128 \) 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.

The bundling of individual BCH searches is faced with the same issue as elliptic BCH searches. Because of the sorted buckets, most bucket entry scans are performed close to the sources of the BCH queries. In fact, this issue is even more pronounced for individual BCH queries as the total radius of each search is smaller due to the cost based stopping criterion. In the PALS case, the radius of each search is so small that bundling without SIMD instructions even at \( k = 8 \) actually increases the running time. Bundled searches work better in the DALS case since the stopping criterion is worse which leads to more work at larger distances. With SIMD parallelism, \( k = 8 \) has almost no drawbacks compared to \( k = 1 \). Thus, the optimal value in both cases is \( k = 8 \).

### 9.2 Sorted Buckets

In the following, we analyze the effect of sorted buckets on elliptic BCH searches as well as individual and collective last stop BCH searches. We consider the reduction in the number
Sorted Buckets for Elliptic BCH Searches. The buckets for elliptic BCH searches are already strongly pruned using elliptic pruning. Therefore, sorting these buckets only elicits a major effect with a sufficiently large number of vehicles. As we can see in Figure 5, sorted buckets only have a limited impact for the Berlin-1pct instances but a much larger one for the Berlin-10pct instance as the latter considers ten times more vehicles. On the larger input, sorted buckets reduce the number of entries scanned by about half, which leads to a decrease in the search time by up to 40% (20ms). At the same time, maintaining the order of bucket entries only increases the time for updating bucket entries by less than 10µs. In conclusion, sorted buckets are a valuable improvement for elliptic BCH searches, particularly with respect to the scalability to larger numbers of vehicles.

Sorted Buckets for Last Stop BCH Searches. For last stop BCH searches, sorted buckets are vital to reduce the number of bucket entries scanned since we cannot use elliptic pruning. We show the impact of sorted buckets on the last stop search times in Figure 6. For individual BCH searches, more than 95% and 75% fewer bucket entries are scanned with sorted buckets in the PALS and DALS cases, respectively. This reduces search times by up to 82% and 78%. For collective searches, the number of bucket entries scanned reduces by about the same relative factors. As collective searches scan fewer bucket entries in total, the impact on the search times is less pronounced with reductions of up to 70% in both cases.

Maintaining sorted last stop buckets incurs an average overhead of less than 20µs per request while the reduction in running time is between one and three orders of magnitude larger.
In Table 3, we compare the search times and the times needed for the enumeration of candidate insertions for the three search approaches used for the PALS and DALS cases. Additionally, we show the number of relaxed edges and scanned bucket entries. We show the results for $\rho \in \{0s, 300s, 600s\}$ on the Berlin-1pct and Berlin-10pct instances. We only conducted one iteration of the experiments for Dijkstra searches because of their large running time.

We find that collective searches are slower than individual BCH searches at $\rho = 0s$ (except for the DALS case on B-1%). This is due to the fact that there is only a single pickup and dropoff which means the overhead for propagating labels instead of only distances is unwarranted.

At $\rho = 300s$ and $\rho = 600s$, collective searches offer the best search times and by far the best enumeration times, though. The search times of collective searches are up to an order of magnitude smaller than those of individual BCH searches. We attribute this to two main advantages of collective searches.

Firstly, collective searches can be pruned more precisely because we use lower bounds on the cost of specific PD-pairs or dropoffs instead of a general lower bound on the cost of...
Thus, the number of edge relaxations and bucket entry scans increases much faster with an increasing number of dropoffs (cf. Figure 6).

In this section, we compare our approach with the LOUD algorithm [4].

9.4 Comparison with LOUD

We can see that enumeration times quickly become very large with tens of thousands of insertions tried.

In fact, domination pruning works so well in the DALS case that the search time is virtually constant with an increasing number of dropoffs (cf. Figure 6).

Furthermore, collective searches consider all sources in one search, maximizing the amount of information that can be used by domination pruning. Bundled searches can only consider one PD-pair or dropoff. This applies to the stopping criteria for bucket scans and for the searches as a whole.

Table 3: Comparison of the running times (in µs) of collective BCH searches (Coll.), individual BCH searches (BCH), and Dijkstra searches (Dij.) for the PALS and DALS case and three radii \( \rho \in \{0s,300s,600s\} \) on the Berlin-1pct and Berlin-10pct instances. Shows number of edge relaxations (#rel.), number of bucket entries scanned (#scans), mean search time (t\(\text{search}\)) and the mean time for enumerating possible insertions (t\(\text{enum}\)) per request. Times marked in bold are the smallest times per radius.

<table>
<thead>
<tr>
<th></th>
<th>Alg.</th>
<th>#(\text{rel.})</th>
<th>#scans</th>
<th>t(\text{search})</th>
<th>t(\text{enum})</th>
<th>Alg.</th>
<th>#(\text{rel.})</th>
<th>#scans</th>
<th>t(\text{search})</th>
<th>t(\text{enum})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Coll.</td>
<td>33</td>
<td>7</td>
<td>4.46</td>
<td>0.49</td>
<td>12</td>
<td>12</td>
<td>3.66</td>
<td>0.53</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCH</td>
<td>30</td>
<td>12</td>
<td>3.16</td>
<td>0.37</td>
<td>12</td>
<td>7</td>
<td>2.31</td>
<td>0.42</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dij.</td>
<td>386</td>
<td>–</td>
<td>33.26</td>
<td>0.30</td>
<td>179</td>
<td>–</td>
<td>14.38</td>
<td>0.34</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>Coll.</td>
<td>2125</td>
<td>–</td>
<td>40.98</td>
<td>0.65</td>
<td>68</td>
<td>33</td>
<td>19.47</td>
<td>0.68</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCH</td>
<td>493</td>
<td>152</td>
<td>41.20</td>
<td>27.55</td>
<td>357</td>
<td>57</td>
<td>62.02</td>
<td>93.33</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dij.</td>
<td>313</td>
<td>–</td>
<td>255.21</td>
<td>24.64</td>
<td>1599</td>
<td>–</td>
<td>228.07</td>
<td>80.14</td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>Coll.</td>
<td>439</td>
<td>64</td>
<td>112.81</td>
<td>0.79</td>
<td>100</td>
<td>52</td>
<td>52.47</td>
<td>0.89</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCH</td>
<td>2243</td>
<td>1135</td>
<td>191.32</td>
<td>408.40</td>
<td>1661</td>
<td>5530</td>
<td>420.25</td>
<td>1839.26</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dij.</td>
<td>1542</td>
<td>–</td>
<td>1551.77</td>
<td>366.51</td>
<td>12852</td>
<td>–</td>
<td>1689.69</td>
<td>1681.56</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Coll.</td>
<td>177</td>
<td>1077</td>
<td>36.31</td>
<td>3.15</td>
<td>154</td>
<td>7895</td>
<td>112.76</td>
<td>6.89</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCH</td>
<td>183</td>
<td>1128</td>
<td>26.14</td>
<td>3.12</td>
<td>159</td>
<td>8283</td>
<td>116.17</td>
<td>7.27</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dij.</td>
<td>22317</td>
<td>–</td>
<td>1978.23</td>
<td>28.55</td>
<td>15780</td>
<td>–</td>
<td>1411.81</td>
<td>132.11</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>Coll.</td>
<td>197</td>
<td>1053</td>
<td>48.74</td>
<td>7.46</td>
<td>173</td>
<td>7857</td>
<td>126.67</td>
<td>20.13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCH</td>
<td>1132</td>
<td>4792</td>
<td>134.84</td>
<td>97.13</td>
<td>1035</td>
<td>37684</td>
<td>605.15</td>
<td>363.65</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dij.</td>
<td>24762</td>
<td>–</td>
<td>3444.91</td>
<td>174.46</td>
<td>18522</td>
<td>–</td>
<td>2667.37</td>
<td>694.29</td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>Coll.</td>
<td>213</td>
<td>1043</td>
<td>66.51</td>
<td>14.84</td>
<td>189</td>
<td>7837</td>
<td>141.38</td>
<td>40.57</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCH</td>
<td>3868</td>
<td>15484</td>
<td>432.85</td>
<td>754.17</td>
<td>3509</td>
<td>122199</td>
<td>1967.98</td>
<td>2403.68</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dij.</td>
<td>62087</td>
<td>–</td>
<td>8149.34</td>
<td>1639.57</td>
<td>49489</td>
<td>–</td>
<td>6807.04</td>
<td>5992.58</td>
<td></td>
</tr>
</tbody>
</table>

Secondly, collective searches consider all sources in one search, maximizing the amount of information that can be used by domination pruning. Bundled searches can only consider one PD-pair or dropoff at once with time overheads for \( k > 8 \) (s.a.). Therefore, each edge may be scanned by up to \( N_p/8 \) or \( N_d/8 \) bundled searches with no way to bundle relaxations between searches. Thus, the number of edge relaxations and bucket entry scans increases much faster than for individual BCH searches. In fact, domination pruning works so well in the DALS case that the search time is virtually constant with an increasing number of dropoffs (cf. Figure 6).

In addition, the enumeration times for collective searches are almost constant, too, while they increase massively with \( \rho \) for individual BCH searches. This is due to the fact that collective searches identify a single candidate insertion during the search while individual BCH searches first find all distances and then enumerate an insertion for each combination of candidate vehicle and PD-pair. Since the number of PD-pairs is proportional to \( \rho^4 \), enumeration times quickly become very large with tens of thousands of insertions tried.

9.4 Comparison with LOUD

In this section, we compare our approach with the LOUD algorithm [4].
Table 4 Running times (in µs) of different phases of LOUD ($\rho = \text{L-...}$) and KaRRi with different radii ($\rho \in \{0s, 300s, 600s\}$) on B-1%, B-10%, R-1%, and R-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. Numbers with asterisks are estimates.

<table>
<thead>
<tr>
<th>Inst.</th>
<th>$\rho$</th>
<th>find $P_\rho, D_\rho$</th>
<th>PD</th>
<th>BCH</th>
<th>Ord.&amp; PBNS</th>
<th>PALS</th>
<th>DALS</th>
<th>update</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B-1%</td>
<td>L-0</td>
<td>0</td>
<td>17</td>
<td>124</td>
<td>147</td>
<td>43</td>
<td>1974</td>
<td>84</td>
<td>2388</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td>36</td>
<td>123</td>
<td>42</td>
<td>3</td>
<td>28</td>
<td>132</td>
<td>367</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>95</td>
<td>155</td>
<td>497</td>
<td>119</td>
<td>43</td>
<td>56</td>
<td>138</td>
<td>1103</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>316</td>
<td>735</td>
<td>1703</td>
<td>469</td>
<td>156</td>
<td>81</td>
<td>140</td>
<td>3601</td>
</tr>
<tr>
<td>B-10%</td>
<td>L-0</td>
<td>0</td>
<td>16</td>
<td>368</td>
<td>415</td>
<td>23</td>
<td>1474</td>
<td>80</td>
<td>2376</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>35</td>
<td>362</td>
<td>353</td>
<td>3</td>
<td>119</td>
<td>200</td>
<td>1074</td>
</tr>
<tr>
<td></td>
<td>L-300</td>
<td>*104</td>
<td>*18930</td>
<td>28675</td>
<td>736</td>
<td>918</td>
<td>44071</td>
<td>*80</td>
<td>93514</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>104</td>
<td>157</td>
<td>1502</td>
<td>641</td>
<td>24</td>
<td>140</td>
<td>203</td>
<td>2772</td>
</tr>
<tr>
<td></td>
<td>L-600</td>
<td>*353</td>
<td>*246521</td>
<td>48856</td>
<td>190</td>
<td>8998</td>
<td>166244</td>
<td>*80</td>
<td>472952</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>353</td>
<td>759</td>
<td>4804</td>
<td>1741</td>
<td>106</td>
<td>181</td>
<td>213</td>
<td>8158</td>
</tr>
<tr>
<td>R-1%</td>
<td>L-0</td>
<td>0</td>
<td>19</td>
<td>173</td>
<td>228</td>
<td>128</td>
<td>8572</td>
<td>82</td>
<td>9202</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>18</td>
<td>163</td>
<td>106</td>
<td>5</td>
<td>34</td>
<td>138</td>
<td>481</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>98</td>
<td>132</td>
<td>668</td>
<td>194</td>
<td>62</td>
<td>59</td>
<td>146</td>
<td>1375</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>293</td>
<td>566</td>
<td>1989</td>
<td>494</td>
<td>216</td>
<td>81</td>
<td>149</td>
<td>3807</td>
</tr>
<tr>
<td>R-10%</td>
<td>L-0</td>
<td>0</td>
<td>18</td>
<td>703</td>
<td>944</td>
<td>89</td>
<td>6154</td>
<td>80</td>
<td>7988</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>19</td>
<td>708</td>
<td>944</td>
<td>5</td>
<td>163</td>
<td>298</td>
<td>2155</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>106</td>
<td>138</td>
<td>3360</td>
<td>1293</td>
<td>42</td>
<td>182</td>
<td>320</td>
<td>5462</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>324</td>
<td>595</td>
<td>9396</td>
<td>2476</td>
<td>128</td>
<td>226</td>
<td>387</td>
<td>13554</td>
</tr>
</tbody>
</table>

Running Times. We give the running times for the different phases of both algorithms on the Berlin-1pct, Berlin-10pct, Ruhr-1pct, and Ruhr-10pct instances in Table 4. For KaRRi, we consider $\rho \in \{0s, 300s, 600s\}$ and use the optimal last stop search approach in each configuration. We also report the running times of the LOUD algorithm ($\rho = \text{L-0}$) on all four instances. Additionally, we consider an estimate for the running time of a naïve extension of LOUD to multiple PD-locations ($\rho = \text{L-300}$ and $\rho = \text{L-600}$) for the medium sized Berlin-10pct instance.

First, we consider the scenario with a single pickup and dropoff ($\rho = 0s$) which is the scenario considered by LOUD. Here, sorted buckets have little impact on the search times of elliptic BCH searches even though the number of bucket entries scanned is reduced. We attribute this to the fact that our implementation is meant to deal with any number of PD-locations while LOUD is specialized for the case of $N_p = N_d = 1$. Our last stop BCH searches are well suited for $\rho = 0s$, though. They are up to 27 and 250 times faster than LOUD’s Dijkstra searches in the PALS and DALS cases, respectively. Maintaining sorted last stop buckets does lead to increased update times, though, especially for the larger instance where buckets contain more entries. In total, we can reduce the average time per request by factors of 6.5 for Berlin-1pct and 2.2 for Berlin-10pct compared to LOUD. On the larger Ruhr instances, we achieve speedups of 19.1 and 3.7 for the Ruhr-1pct and Ruhr-10pct instances.

Next, we consider our estimate for an extension of LOUD that uses only the techniques of the original algorithm. For this, we configured KaRRi to use no bundled searches, no sorted buckets, and to use Dijkstra searches for the PALS and DALS cases. For the PD-distances,
Table 5: Solution quality of KaRRi with different radii ($\rho \in \{0s, 300s, 600s\}$) on B-1%, B-10%, R-1%, and R-10%. For requests, we report the average and 95%-quantile wait time, and the average ride and trip times (in mm:ss). For vehicles, we give the average times spent driving empty, driving occupied, and making stops, as well as the average total operation time (in hh:mm).

<table>
<thead>
<tr>
<th>Inst.</th>
<th>$\rho$</th>
<th>wait</th>
<th>w.-95%</th>
<th>ride</th>
<th>trip</th>
<th>empty</th>
<th>occ</th>
<th>stop</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-1%</td>
<td>0</td>
<td>3:49</td>
<td>9:38</td>
<td>12:32</td>
<td>17:18</td>
<td>0:41</td>
<td>3:10</td>
<td>0:28</td>
<td>4:19</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>3:22</td>
<td>8:24</td>
<td>11:56</td>
<td>16:13</td>
<td>0:30</td>
<td>2:56</td>
<td>0:30</td>
<td>3:56</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>3:27</td>
<td>8:34</td>
<td>11:47</td>
<td>16:11</td>
<td>0:29</td>
<td>2:51</td>
<td>0:30</td>
<td>3:51</td>
</tr>
<tr>
<td>B-10%</td>
<td>0</td>
<td>2:31</td>
<td>7:11</td>
<td>11:57</td>
<td>15:07</td>
<td>0:16</td>
<td>2:29</td>
<td>0:26</td>
<td>3:10</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>2:20</td>
<td>6:20</td>
<td>11:33</td>
<td>14:41</td>
<td>0:09</td>
<td>2:12</td>
<td>0:27</td>
<td>2:48</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>2:30</td>
<td>7:34</td>
<td>11:27</td>
<td>14:50</td>
<td>0:08</td>
<td>2:08</td>
<td>0:27</td>
<td>2:42</td>
</tr>
<tr>
<td>R-1%</td>
<td>0</td>
<td>4:49</td>
<td>11:48</td>
<td>12:25</td>
<td>18:26</td>
<td>0:56</td>
<td>3:15</td>
<td>0:27</td>
<td>4:39</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>4:15</td>
<td>10:20</td>
<td>11:48</td>
<td>17:07</td>
<td>0:44</td>
<td>3:03</td>
<td>0:29</td>
<td>4:16</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>4:17</td>
<td>9:52</td>
<td>11:33</td>
<td>16:52</td>
<td>0:42</td>
<td>2:58</td>
<td>0:29</td>
<td>4:09</td>
</tr>
<tr>
<td>R-10%</td>
<td>0</td>
<td>3:11</td>
<td>8:46</td>
<td>11:46</td>
<td>15:48</td>
<td>0:24</td>
<td>2:40</td>
<td>0:25</td>
<td>3:29</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>2:45</td>
<td>7:07</td>
<td>11:13</td>
<td>14:51</td>
<td>0:16</td>
<td>2:26</td>
<td>0:26</td>
<td>3:08</td>
</tr>
<tr>
<td></td>
<td>600</td>
<td>2:55</td>
<td>7:58</td>
<td>11:06</td>
<td>14:56</td>
<td>0:15</td>
<td>2:22</td>
<td>0:26</td>
<td>3:03</td>
</tr>
</tbody>
</table>

we obtained an estimate for the time of running one CH-query per PD-pair by multiplying LOUD’s PD-distance search time with $N_p \cdot N_d$. We find that bundling and sorted buckets make elliptic BCH searches about one order of magnitude faster than the naïve extension. For the PALS and DALS searches, our collective BCH approach beats the standard Dijkstra approach by two and three orders of magnitude, respectively. We assume that the CH-queries for PD-distances would in reality be faster than our estimate. Nonetheless, it is notable that our BCH based approach is hundreds of times faster than the estimate.

Solution Quality. In the following, we give a first idea of how trip times and vehicle operation times can be improved by extending ridesharing with walking.

In Table 5, we compare the solution quality of KaRRi with $\rho \in \{0s, 300s, 600s\}$. Note that we allow passengers to walk to their destinations if the resulting walking time leads to better cost than a ridesharing trip (see Section 4.1). The cost function used here equally weights the passenger trip times and vehicle operation times ($\tau = 1$). With larger values of $\rho$, we observe improvements in both criteria. At $\rho = 300s$, the average vehicle operations times and passenger wait times improve by up to 12% while trip times improve by up to 7%. At $\rho = 600s$, the vehicle operation times and passenger ride times reduce further compared to $\rho = 300s$.

There are a number of parameters not evaluated in these preliminary results. For instance, we have not considered the willingness of passengers to walk longer distances in order to reduce trip times in our cost function (since $\omega = 0$). Also, we only show results for a fixed number of vehicles and a density of requests representing a regular week day. In the future, we would like to include these parameters in our analysis of the effects of PD-locations on ridesharing and extend this evaluation to larger inputs.

10 Conclusions and Future Work

KaRRi develops efficient many-to-many routing with bucket contraction hierarchies that allows efficient scheduling of large vehicle fleets considering many pickup and dropoff locations. 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.

Next, we want to use KaRRi to evaluate different traffic scenarios. For example, we expect that even larger savings over single PD-locations are possible when using faster individual transport or when using fewer but larger vehicles.

Future work on the algorithmic side can achieve acceleration by clustering routing sources into batches of size $k$ by their proximity, by finding a collective approach for elliptic BCH searches, and by parallelizing the algorithm both over requests (that use different vehicles) and over different PD-locations. The overall cost could be further optimized by 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.

We expect that additional generalizations can integrate KaRRi with public transportation where pickup and dropoff locations can be stops of buses or trains and where the cost function has to take into account the public transportation schedule.

A longer term perspective is to allow switching vehicles during a trip. This opens up the possibility of more sharing using larger vehicles, eventually leading to a highly adaptive software defined public transportation system. This implies interesting algorithmic challenges as it leads to a combinatorial explosion of possible route options.

References


Fast Many-to-Many Routing for Ridesharing with Multiple Pickup and Dropoff Loc.


