

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 naïve 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 6-8 address individual challenges. In Section 9, 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 ℓ_{veh} , δ_{veh} , ℓ_{psg} , and δ_{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 $\nu = (l_i, cap, t_{serv}^{min}, t_{serv}^{max})$ has an initial location l_i , a seating capacity cap and a service time interval $[t_{serv}^{min}, t_{serv}^{max}]$. The current *route* $R(\nu) = \langle s_0(\nu), \dots, s_{k(\nu)}(\nu) \rangle$ of a vehicle ν is a sequence of *stops* scheduled for the vehicle. The vehicle's current location $l_c(\nu)$ 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_{stop}^{min} 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 $t_{arr}^{min}(s)$ and departure time $t_{dep}^{min}(s)$ according to the current schedule.

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

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 ρ . 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 ρ . The sets of pickups and dropoffs for r and a radius ρ 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 ρ 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.t. the cost of that insertion according to a cost function is minimized. We formalize an insertion as a tuple (r, p, d, ν, i, j) indicating that vehicle ν picks up request r at pickup location $p \in P_\rho(r)$ 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 $0 \leq i \leq j \leq k(\nu)$.

2.1 Cost Function and Constraints

The cost of an insertion $\iota = (r, p, d, \nu, i, j)$ considers the added vehicle operation time $t_{detour}(\iota)$ of ν , the trip time $t_{trip}(\iota)$ of r , the sum of increased trip times $t_{trip}^+(\iota)$ of existing passengers of ν , and the walking time $t_{walk}(\iota)$ (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 ν 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 ν must still be picked up at their pickup stop within a *maximum wait time* t_{wait}^{max} . Fourth, every passenger \hat{r} already assigned to ν must still arrive at their destination within a *maximum trip time* $t_{trip}^{max}(\hat{r}) = \alpha \cdot \delta_{veh}(orig(\hat{r}), dest(\hat{r})) + \beta$. The values t_{wait}^{max} , α and β are model parameters.

All four constraints are hard constraints wrt. requests already assigned to ν . If ι breaks a hard constraint, we set the cost to ∞ . For the request r to be inserted, we treat the wait time and trip time constraints as soft constraints, i.e. violating them leads to cost penalties. Assume, the passenger is picked up at p at time t_{dep} . We define the cost penalties as

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

with model parameters γ_{wait} and γ_{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:

$$c(\iota) = t_{detour}(\iota) + \tau \cdot (t_{trip}(\iota) + t_{trip}^+(\iota)) + \omega \cdot t_{walk}(\iota) + c_{wait}^{vio}(\iota) + c_{trip}^{vio}(\iota)$$

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 τ and ω . 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* [8] 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 $\text{key}(v) = \delta(s, v)$ 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.

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, w) is created if $(u, v, w) \in E^2$ is the only shortest path between u and w .

Let E^+ contain all original edges E as well as all shortcut edges. The graph $G^+ = (V, E^+)$ constitutes the CH. The length $\ell^+(e)$ of a shortcut edge e is the sum of the lengths of replaced original edges while δ^+ is the according distance function. For the query phase, we partition E^+ into *up-edges* $E^\uparrow = \{(u, v) \in E^+ \mid \text{rank}(u) < \text{rank}(v)\}$ and *down-edges* $E^\downarrow = \{(u, v) \in E^+ \mid \text{rank}(u) > \text{rank}(v)\}$. We define an *upwards search graph* $G^\uparrow := (V, E^\uparrow)$ and a *downwards search graph* $G^\downarrow := (V, E^\downarrow)$. The distance functions δ^\uparrow and δ^\downarrow represent δ^+ constrained to G^\uparrow and G^\downarrow . The *upwards CH search space* $G_v^\uparrow = (V_v^\uparrow, E_v^\uparrow)$ rooted at a vertex $v \in V$ contains all vertices $V_v^\uparrow \subseteq V$ that can be reached from v in E^\uparrow and the according edges $E_v^\uparrow \subseteq E^\uparrow$. The *reverse downwards CH search space* $G_v^\downarrow = (V_v^\downarrow, E_v^\downarrow)$ rooted at a vertex $v \in V$ contains all vertices $V_v^\downarrow \subseteq V$ from which v is reachable in E^\downarrow and the according edges $E_v^\downarrow \subseteq E^\downarrow$.

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^\uparrow and a reverse Dijkstra search from t in G^\downarrow . Whenever the searches meet, they find an up-down-path from s to t , eventually finding a shortest path. The query can stop once the radius of either Dijkstra search 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* $B^\downarrow(v)$ 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 $B^\downarrow(v)$ for every settled $v \in V$. Then, a forward search from s in G^\uparrow can compute tentative shortest path distances as $\delta^\uparrow(s, v) + \delta^\downarrow(v, t)$ for every bucket entry $(t, \delta^\downarrow(v, t)) \in B^\downarrow(v)$ at every settled vertex $v \in V_s^\downarrow$.

BCH searches can analogously compute the distances from a set of sources to a single target. In that case, we speak of *source buckets* $B^\uparrow(v)$ 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 ∞ . 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* [11]. Since then, bundled searches have been used in a number of Dijkstra-based many-to-many shortest path algorithms [1, 17, 7, 5, 6]. More recently, the idea has been extended to point-to-point queries in CHs [3].

Instruction-Level Parallelism in Bundled Searches. Additionally, single-instruction multiple-data (SIMD) parallelism can be utilized for bundled searches [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 [3].

3.2 LOUD

Our algorithm is based on the dynamic ridesharing dispatching algorithm *LOUD* [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 [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 $\iota = (r, p, d, \nu, i, j)$ are inserted after the next stop of ν , 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 $l_c(\nu)$ and the location $l(s)$ of each stop $s \in R(\nu)$, the ignored cases could be integrated into the following definitions in a straight forward manner.

4.1 Walking Time and Walking to the Destination

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

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 ι_{psg} where the passenger is matched to no vehicle at all. Then, the cost of ι_{psg} depends only on the walking distance $t_{walk}(\iota_{psg}) = t_{trip}(\iota_{psg}) = \delta_{psg}(orig, dest)$. 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 $c(\iota_{psg}) = \tau \cdot t_{trip}(\iota_{psg}) + \omega \cdot t_{walk}(\iota_{psg}) + c_{trip}^{vio}(\iota_{psg})$. We explicitly allow pseudo-insertions for any distance $\delta_{psg}(orig, dest)$, i.e. the distance does not have to be found within the radius ρ around $orig$ or $dest$. Instead, we use a CH query in the passenger graph to find $\delta_{psg}(orig, dest)$. The cost $c(\iota_{psg})$ 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 $r = (orig, dest, t_{req})$ always waits to be picked up by the vehicle at the origin location $orig$. The request is issued at time t_{req} and the vehicle ν matched to the request can start making its way to the pickup location at the earliest at t_{req} . If the pickup location is $orig$ 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_{stop}^{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 $t_{arr}^{min}(s) = t_{dep}^{min}(s) - t_{stop}^{min}$. 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 $\delta_{veh}(s_i, s_{i+1}) = t_{dep}^{min}(s_{i+1}) - t_{stop}^{min} - t_{dep}^{min}(s_i)$.

Now, consider an insertion $\iota = (r, p, d, \nu, i, j)$ with $p \neq orig$. Then, both the vehicle and the passenger have to travel to p starting at $t_{dep}^{min}(s_i)$ and t_{req} , respectively. Consequently, the vehicle can arrive at p earlier than the passenger, precisely if $t_{dep}^{min}(s_i) + \delta_{veh}(s_i, p) < t_{req} + \delta_{psg}(orig, p)$. 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_{stop}^{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 ν can depart from p at the earliest after a stop time of t_{stop}^{min} and the passenger can depart as soon as they arrive at p leading us to

$$t_{dep}(\iota) = \max(t_{dep}^{min}(s_i) + \delta_{veh}(s_i, p) + t_{stop}^{min}, t_{req} + \delta_{psg}(orig, p)).$$

We later use $t_{dep}(\iota)$ in the definitions of the vehicle detour, passenger wait time and passenger trip time needed for the cost function (see Section 2.1). 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 $t_{arr}^{min}(s)$ from $t_{dep}^{min}(s)$ for a stop s , we have to now explicitly store $t_{arr}^{min}(s)$ and $t_{dep}^{min}(s)$. The distance between a pair of consecutive stops (s_i, s_{i+1}) can then be derived as $\delta_{veh}(s_i, s_{i+1}) = t_{arr}^{min}(s_{i+1}) - t_{dep}^{min}(s_i)$.

We denote the scheduled wait time of a vehicle at stop s with $t_{wait}^{veh}(s) = t_{dep}^{min}(s) - t_{arr}^{min}(s) - t_{stop}^{min}$.

4.3 Added Vehicle Operation Time and Passenger Trip Times

In the following, we define the added vehicle operation time $t_{detour}(\iota)$, the trip time for the new passenger $t_{trip}(\iota)$, as well as the sum of added trip times for existing passengers $t_{trip}^+(\iota)$ for an insertion $\iota = (r, p, d, \nu, i, j)$. Vehicle wait times as explained in Section 4.2 have an effect on all of these times.

In order to explain this effect, we first focus on the added vehicle operation time caused by the insertion ι . Conceptually, the added vehicle operation time is the time difference between the vehicle's arrival time at the last scheduled stop after the insertion and the arrival time at the last scheduled stop before the insertion.

Initial Detours. We start by explaining added vehicle operation times in a scenario without vehicle waiting times, i.e. $t_{wait}^{veh}(s) = 0$ for all $s \in R(\nu)$. In this case, the added vehicle operation time for performing a pickup at p and a dropoff at d is simply equal to the detour that vehicle ν has to take in its route after stop s_i and stop s_j , respectively. We call these detours the initial pickup detour and initial dropoff detour.

► **Definition 1.** *The initial pickup (dropoff) detour for an insertion $\iota = (r, p, d, \nu, i, j)$ 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

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

We define the initial dropoff detour as

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

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 $t_{dep}(\iota)$ at pickup p is the arrival time of ν at p plus the minimum stopping time t_{stop}^{min} , so $t_{dep}(\iota) = t_{dep}^{min}(s_i) + \delta(s_i, p) + t_{stop}^{min}$ (see Section 4.2). The first effect that vehicle wait times can have on the detour is the fact that $t_{dep}(\iota)$ 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 $t_{dep}(\iota) = \max(t_{dep}^{min}(s_i) + \delta(s_i, p) + t_{stop}^{min}, t_{req}(r) + \delta_{psg}(orig, p))$. Therefore, in a scenario with vehicle waiting times, the initial pickup detour $\Delta_p^{init}(\iota)$ 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 $j + 1 < a < k(\nu)$, $t_{wait}^{veh}(s_a) > 0$ and $t_{wait}^{veh}(s_b) = 0$ for $b \neq a$. Let $t_{arr}^{min}(s, \iota)$ and $t_{dep}^{min}(s, \iota)$ describe the scheduled arrival time and departure time

at stop $s \in R(\nu)$ after we perform insertion ι . After the insertion, vehicle ν will arrive at stop s_{j+1} with a delay of $\Delta_{j+1}^{res}(\iota) = \Delta_p^{init}(\iota) + \Delta_d^{init}(\iota)$ compared to before the insertion, so $t_{arr}^{min/\iota}(s_{j+1}) = t_{arr}^{min}(s_{j+1}) + \Delta_{j+1}^{res}(\iota)$. 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 $\Delta_a^{res}(\iota) = \Delta_{j+1}^{res}(\iota)$. However, before the insertion, the vehicle has to wait for a duration of $t_{wait}^{veh}(s_a)$ 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 ι . 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

$$\begin{aligned} t_{arr}^{min/\iota}(s_a, \iota) &= t_{arr}^{min}(s_a) + \Delta_a^{res}(\iota) \text{ but} \\ t_{dep}^{min/\iota}(s_a, \iota) &= t_{dep}^{min}(s_a) + \Delta_{a+1}^{res}(\iota) \text{ where } \Delta_{a+1}^{res}(\iota) = \max(\Delta_a^{res}(\iota) - t_{wait}^{veh}(s_a), 0) \end{aligned}$$

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 $\Delta_b^{res}(\iota) = \Delta_{a+1}^{res}(\iota)$ so $t_{arr}^{min/\iota}(s_b, \iota) = t_{arr}^{min}(s_b) + \Delta_{a+1}^{res}(\iota)$.

This leads us to the definition of residual detours which describe the actual delay that an insertion ι 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 $\Delta_a^{res}(\iota)$ for an insertion $\iota = (r, p, d, \nu, i, j)$ at stop $s_a \in R(\nu)$ describes how much later the vehicle ν will arrive at stop s_a after the insertion is performed. Formally, we define it as*

$$\Delta_a^{res}(\iota) := \begin{cases} 0 & \text{if } a \leq i \\ \Delta_p^{init}(\iota) & \text{if } i + 1 = a \leq j \\ \max(\Delta_j^{res}(\iota) - t_{wait}^{veh}(s_j), 0) + \Delta_d^{init}(\iota) & \text{if } a = j + 1 \text{ and } i \neq j \\ \Delta_p^{init}(\iota) + \Delta_d^{init}(\iota) & \text{if } a = j + 1 \text{ and } i = j \\ \max(\Delta_{a-1}^{res}(\iota) - t_{wait}^{veh}(s_{a-1}), 0) & \text{otherwise} \end{cases}$$

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

$$\begin{aligned} t_{arr}^{min/\iota}(s_a, \iota) &= t_{arr}^{min}(s_a) + \Delta_a^{res}(\iota) \text{ and} \\ t_{dep}^{min/\iota}(s_a, \iota) &= \max(t_{arr}^{min/\iota}(s_a) + t_{stop}^{min}, t_{dep}^{min}(s_a)). \end{aligned}$$

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

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

$$t_{detour}(\iota) := \begin{cases} \Delta_p^{init}(\iota) + \Delta_d^{init}(\iota) & \text{if } i = j = k(\nu) \\ t_{arr}^{min/\iota}(s_{k(\nu)}, \iota) - t_{arr}^{min}(s_{k(\nu)}) + \Delta_d^{init}(\iota) & \text{if } i < j = k(\nu) \\ t_{arr}^{min/\iota}(s_{k(\nu)}, \iota) - t_{arr}^{min}(s_{k(\nu)}) & \text{otherwise} \end{cases}$$

Trip Time, Added Trip Time For Existing Passengers. The previous definitions also enable us to define the trip time $t_{trip}(\iota)$ of the passenger associated with the request r . The trip time for r is simply the time between the request time $t_{req}(r)$ and the scheduled arrival of the passenger at their destination $dest(r)$. 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 $t_{trip}(\iota)$ for an insertion $\iota = (r, p, d, \nu, i, j)$ is defined as*

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

where

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

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 ν . Consider a request r' that has previously been assigned to vehicle ν . Assume r' is picked up by ν at stop $s_{i'}$ and dropped off at stop $s_{j'}$. By definition of residual detours, performing the insertion ι will delay the arrival of the vehicle at $s_{j'}$ by $\Delta_{j'}^{res}(\iota)$. 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 $t_{trip}^+(\iota)$ caused by an insertion $\iota = (r, p, d, \nu, i, j)$ is defined as*

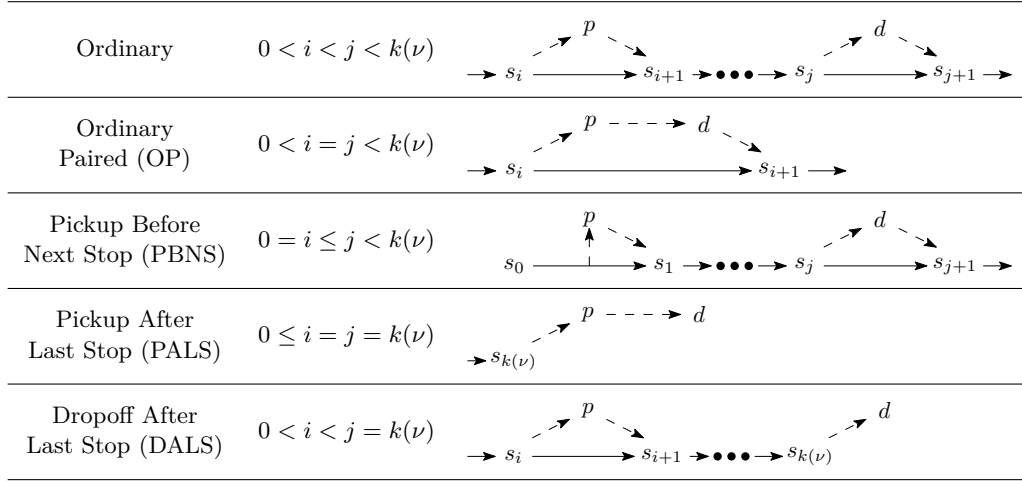
$$t_{trip}^+(\iota) := \sum_{a=i+1}^{k(\nu)} N_d(s_a) \cdot \Delta_a^{res}(\iota)$$

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 ρ 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 ι^* 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.



■ **Figure 1** Insertion types. Shows characterization of each type based on the pickup and dropoff insertion points i and j of an insertion $\iota = (r, p, d, \nu, i, j)$. Illustrations depict the current route of ν (solid arrows) with stops $s \in R(\nu)$ as well as the detours to and from p and d (dashed lines).

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 p and the dropoff d are inserted between two existing stops of the route $R(\nu)$. To compute the cost of any insertion of one of these types, we need to know the distances between existing vehicle stops and the PD-locations. BCH searches with elliptic pruning (see Section 3.2) have been shown to efficiently compute these distances [4]. We call these *elliptic BCH searches*. Additionally, cost calculations for paired insertions require the PD-distance $\delta_{veh}(p, d)$. In this section, we explain how we extend the required distance queries for multiple PD-locations.

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 ν 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 ν (see Section 3.2). When an elliptic BCH search to a pickup p scans a source bucket entry $(s, \delta^\uparrow(s_i, \nu)) \in B^\uparrow(v)$, the tentative distance $\delta^\uparrow(s_i, v) + \delta^\downarrow(v, p)$ can only lead to an insertion that holds all hard constraints if $\delta^\uparrow(s_i, v) + \delta^\downarrow(v, p) \leq \lambda(s_i, s_{i+1})$. This means the entry is only relevant for p if $\delta^\downarrow(v, p) \leq \lambda(s_i, s_{i+1}) - \delta^\uparrow(s_i, v)$. We call $\lambda_{res}(s_i, v) := \lambda(s_i, s_{i+1}) - \delta^\uparrow(s_i, v)$ the *remaining leeway* of a source bucket entry $(s_i, \delta^\uparrow(s_i, v))$ 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

v once an entry $(s, \delta^\uparrow(s, v))$ is scanned for which $\delta^\downarrow(v, p) > \lambda_{res}(s, v)$. In this case, we have $\delta^\downarrow(v, p) > \lambda_{res}(s, v) \geq \lambda_{res}(s', v)$ for any subsequent entry $(s', \delta^\uparrow(s', v))$, 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 δ_{PD}^{\max} be an upper bound on all PD-distances. Then, we only have to generate and scan bucket entries in a radius of δ_{PD}^{\max} . We use

$$\delta_{PD}^{\max} := \max_{p \in P_\rho} \delta_{veh}(p, orig) + \delta_{veh}(orig, dest) + \max_{d \in D_\rho} \delta_{veh}(dest, d).$$

We can compute $\delta_{veh}(p, orig)$ for all $p \in P_\rho$ and $\delta_{veh}(dest, d)$ for all $d \in D_\rho$ 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 $\iota = (r, p, d, \nu, i, j)$ with $0 < i \leq j < k(\nu)$ for a set of candidate vehicles found by the elliptic BCH queries [4]. We compute the cost $c(\iota)$ for each insertion and update ι^* to ι if $c(\iota) < c(\iota^*)$.

To compute the cost of a *pickup before next stop* (PBNS) insertion $\iota = (r, p, d, \nu, 0, j)$, we lack knowledge of the distance $\delta_{veh}(l_c(\nu), p)$ from the vehicle's current location $l_c(\nu)$ to

the pickup p . LOUD suggests filtering PBNS insertions based on a lower bound on the cost of ι obtained from considering $\delta_{veh}(s_0, p)$ instead of $\delta_{veh}(s_0, l_c(\nu)) + \delta_{veh}(l_c(\nu), p)$ for the pickup detour [4]. This filter discards almost all PBNS insertions. For the remaining PBNS insertions, we compute the missing distances $\delta_{veh}(l_c(\nu), p)$ 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 [4]. Instead, LOUD uses a reverse Dijkstra search rooted at *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{c} denote an upper bound on the best known insertion cost (initially $\hat{c} := c(\iota^*)$).

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

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

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* $B_{last}^\uparrow(v)$ for every $v \in V$. For every last stop $s_{k(\nu)}$, we generate an entry $(s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, v)) \in B_{last}^\uparrow(v)$ at each vertex v in the upward CH search space rooted at $s_{k(\nu)}$. Then, for every pickup $p \in P_\rho$, we run an *individual (last stop) BCH query* in G_p^\downarrow 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 $(s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, v)) \in B_{last}^\uparrow(v)$, it tries to improve the tentative distance $\delta(s_{k(\nu)}, p)$ with $\delta^\uparrow(s_{k(\nu)}, v) + \delta^\downarrow(v, p)$. Eventually, the shortest distance $\delta_{veh}(s_{k(\nu)}, p)$ 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

entries, especially at vertices that have a high rank in the CH. Therefore, the queries have to scan large numbers of bucket entries, rendering the last stop BCH approach ineffective.

The future work section of [4] suggests sorting the entries within each last stop bucket to address this issue. For each $v \in V$, we sort the entries in $B_{last}^\uparrow(v)$ by their distance $\delta^\uparrow(s_{k(\nu)}, v)$ in increasing order. Suppose a pickup query rooted at $p \in P_\rho$ scans the bucket $B_{last}^\uparrow(v)$. Let

$$c_{\min}(x) := c'(r, p, \delta_{pd}^{\min}, 0, t_{req}(r), x + \delta^\downarrow(v, p)) \quad \text{where } \delta_{pd}^{\min} := \min_{p \in P_\rho, d \in D_\rho} \delta_{veh}(p, d).$$

Then, for any entry $e = (s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, v)) \in B_{last}^\uparrow(v)$, the value $c_{\min}(\delta^\uparrow(s_{k(\nu)}, v))$ is a vehicle-independent lower bound for the cost of any PALS insertion where a vehicle drives a distance of at least $\delta^\uparrow(s_{k(\nu)}, v) + \delta^\downarrow(v, p)$ to p . Note that $c_{\min}(x)$ is linear in x . Thus, since the entries of $B_{last}^\uparrow(v)$ are sorted by $\delta^\uparrow(s_{k(\nu)}, v)$, we can stop the bucket scan if $c_{\min}(\delta^\uparrow(s_{k(\nu)}, v)) > \hat{c}$.

Updating the Upper Bound Cost. It is possible to simply use $c(\iota^*)$ for the upper bound cost \hat{c} needed for cost pruning. However, we can also improve \hat{c} during the search. Each tentative distance $\delta(s_{k(\nu)}, p)$ found acts as an upper bound on the actual shortest distance $\delta_{veh}(s_{k(\nu)}, p)$. Thus, whenever the tentative distance $\delta(s_{k(\nu)}, p)$ is updated, we can compute an upper bound

$$c_{\max} = c'(r, p, \delta_{veh}(p, dest), 0, t_{dep}^{min}(s_{k(\nu)}), \delta(s_{k(\nu)}, p))$$

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

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 $\iota_{pals}^* = (r, p, d, \nu, k(\nu), k(\nu))$ in advance, we would only need to find $\delta_{veh}(s_{k(\nu)}, p)$. Obviously, we do not know ι_{pals}^* in advance but we find that it is possible to prune the distance queries for individual pickups (or actually PD-pairs) by comparing them to each other. We introduce a collective BCH query that finds the best PALS insertion ι_{pals}^* as well as the last stop distance $\delta_{veh}(s_{k(\nu)}, p)$. In the following, we explain how labels representing PD-pairs are propagated through the CH search graph and how these labels can be pruned based on label domination.

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

$$c_{\min}(l) := c'(r, p, \delta_{veh}(p, d), \delta_{psg}(d, dest), t_{req}(r), \delta^\downarrow(v, p))$$

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_{\min} . Initially, at each pickup $p \in P_\rho$, an open label $(p, d, 0) \in open(p)$ is created for each $d \in D_\rho$. As long as Q contains a label l with $c_{\min}(l) \leq \hat{c}$ 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.

■ **Algorithm 1** Collective BCH search used to find distances from last stops to pickups.

```

1: procedure COLLECTIVEBCH( $P_\rho, D_\rho, G^\downarrow = (V^\downarrow, E^\downarrow)$ ) :  $(p^*, d^*)$  and  $\delta_{veh}(s_{k(\nu)}, p^*)$ 
2:    $Q :=$  PQ of labels with  $key_Q(l) = c_{\min}(l)$ ;  $open(v) := \emptyset, closed(v) := \emptyset$  for  $v \in V$ 
3:    $\hat{c} := c(\iota^*)$ 
4:   for each  $(p, d) \in P_\rho \times D_\rho$  do
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 SETTLELABEL( $l = (p, d, \delta^\downarrow(v, p))$ )
11:    $open(v).remove(l)$ ;  $closed(v).insert(l)$  ▷ mark  $l$  closed
12:   for each  $e = (s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, v)) \in B_{last}^\uparrow(v)$  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  $(u, v) \in E^\downarrow$  do ▷ propagate  $l$  to neighbors of  $v$ 
17:     insertLabelAtVertex( $u, (p, d, \ell^+(u, v) + \delta^\downarrow(v, p))$ )

18: procedure INSERTLABELATVERTEX( $v, l' = (p, d, \delta^\downarrow(v, p))$ )
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. Settling an open label $l = (p, d, \delta^\downarrow(v, p))$ 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 $B_{last}^\uparrow(v)$. For each entry $e = (s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, v)) \in B_{last}^\uparrow(v)$, we compute an upper bound cost

$$c_{\max}(l, e) := c'(r, p, \delta_{veh}(p, d), \delta_{psg}(d, dest), t_{dep}^{min}(s_{k(\nu)}), \delta^\uparrow(s_{k(\nu)}, v) + \delta^\downarrow(v, p)).$$

If $c_{\max}(l, e) < \hat{c}$, we mark $\iota = (r, p, d, \nu, k(\nu), k(\nu))$ as the best known PALS insertion, store the tentative distance $\delta(s_{k(\nu)}, p) = \delta^\uparrow(s_{k(\nu)}, v) + \delta^\downarrow(v, p)$, and update $\hat{c} := c_{\max}(l, e)$. Note that $c_{\max}(l, e)$ is the exact cost of the PALS insertion $\iota = (r, p, d, \nu, k(\nu), k(\nu))$ if $\delta(s_{k(\nu)}, p)$ 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

$$c_{\min}(l, e) := c'(r, p, \delta_{veh}(p, d), \delta_{psg}(d, dest), t_{req}(r), \delta^\uparrow(s_{k(\nu)}, v) + \delta^\downarrow(v, p)).$$

Third, we propagate l to all neighboring vertices of v . For each neighboring vertex $w \in V$ with $(w, v) \in G^\downarrow$, we create a new open label $l' = (p, d, \ell^+(w, v) + \delta^\downarrow(v, p))$ 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 to 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_v^\downarrow 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_v^\downarrow for a label l . Let $l = (p, d, \delta^\downarrow(v, p))$ be a label at v and $e = (s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, w)) \in B_{last}^\uparrow(w)$ a last stop bucket entry at a vertex $w \in V_v^\downarrow$. Then, we define an upper bound for the cost of an insertion where ν drives from $s_{k(\nu)}$ to p via w and v as

$$c_{\max}(l, v, e) := c'(r, p, \delta_{veh}(p, d), \delta_{psg}(d, dest), t_{dep}^{min}(s_{k(\nu)}), \delta^\uparrow(s_{k(\nu)}, w) + \delta^\downarrow(w, v) + \delta^\downarrow(v, p))$$

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 $w \in V_v^\downarrow$ and $e \in B_{last}^\uparrow(w)$.

► **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 $l = (p, d, \delta^\downarrow(v, p))$ dominates $l' = (p', d', \delta^\downarrow(v, p'))$ at v . Further, assume that $\iota = (r, p', d', \nu, k(\nu), k(\nu))$ is the best PALS insertion. Let π be a shortest path from $s_{k(\nu)}$ to p' . Wlog. π is an up-down-path in the CH consisting of an upwards prefix π^\uparrow and a downwards suffix π^\downarrow . If π^\downarrow does not contain v , then the collective search will not find π in G_v^\downarrow , and we do not have to settle l' at v .

Otherwise, $\pi^\downarrow = (w, \dots, v, \dots, p')$ with $w \in V_v^\downarrow$. Let $e = (s_{k(\nu)}, \delta^\uparrow(s_{k(\nu)}, w)) \in B_{last}^\uparrow(w)$. Since π is a shortest path, we know that

$$\begin{aligned} c_{\max}(l', v, e) &= c'(r, p', \delta_{veh}(p', d'), \delta_{psg}(d', dest), t_{dep}^{min}(s_{k(\nu)}), \delta_{veh}(s_{k(\nu)}, p')) \\ &= c((r, p', d', \nu, k(\nu), k(\nu))). \end{aligned}$$

However, l dominates l' which means that

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

This contradicts ι being the best PALS insertion. Hence, we do not have to settle label l' at v to find the best pair for ν . ◀

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 $l_i = (p_i, d_i, \delta^\downarrow(v, p_i))$ for $i = 1, 2$ and two PALS insertions $\iota_i = (r, p_i, d_i, \nu, k(\nu), k(\nu))$ found for these labels in G_v^\downarrow . Assume the vehicle ν arrives at v at some time $t = t_{dep}^{min}(s_{k(\nu)}) + \delta_{veh}(s_{k(\nu)}, v)$. If we know t , we can determine the cost difference $c(\iota_1) - c(\iota_2)$ irrespective of the actual vehicle ν . 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_v^\downarrow 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_v^\downarrow 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. $t + \delta^\downarrow(v, p_i) + t_{stop}^{min} \leq t_{req}(r) + \delta_{psg}(orig, p_i)$ (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^\downarrow , 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^\downarrow . 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 $t_{dep}^{max}(l_1) - t_{dep}^{min}(l_2)$ with

$$\begin{aligned} t_{dep}^{max}(l_1) &:= \max\{\delta^\downarrow(v, p_1) + t_{stop}^{min}, \delta_{psg}(p_1, orig)\} \\ t_{dep}^{min}(l_2) &:= \delta^\downarrow(v, p_2) + t_{stop}^{min} \end{aligned}$$

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

$$\begin{aligned} t_{arr}^{max}(l_1) &:= t_{dep}^{max}(l_1) + \delta_{veh}(p_1, d_1) + \delta_{psg}(d_1, dest) \\ t_{arr}^{min}(l_2) &:= t_{dep}^{min}(l_2) + \delta_{veh}(p_2, d_2) + \delta_{psg}(d_2, dest) \end{aligned}$$

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

$$\begin{aligned} \Delta_{wait}^{vio}(l_1, l_2) &:= \gamma_{wait} \max\{t_{dep}^{max}(l_1) - t_{dep}^{min}(l_2), 0\} \\ \Delta_{trip}^{vio}(l_1, l_2) &:= \gamma_{trip} \max\{\Delta_{trip}(l_1, l_2), 0\} \end{aligned}$$

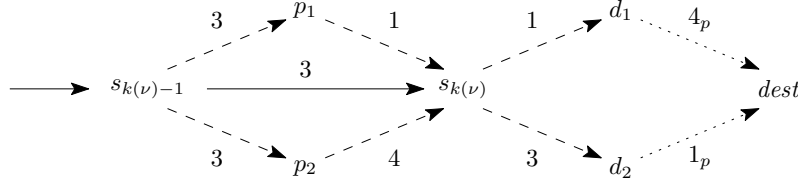
Note that the differences in detours and trip times are allowed to be negative to express a cost advantage for l_1 but the differences in penalties are not. Even if $t_{dep}^{max}(l_1) < t_{dep}^{min}(l_2)$ or $\Delta_{trip}(l_1, l_2) < 0$, we may find insertions in G_v^\downarrow where no penalties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

$$\Delta_c^{max}(l_1, l_2) := \Delta_{detour}(l_1, l_2) + \tau \Delta_{trip}(l_1, l_2) + \omega \Delta_{walk}(l_1, l_2) + \Delta_{wait}^{vio}(l_1, l_2) + \Delta_{trip}^{vio}(l_1, l_2)$$

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 that 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 ι found by the collective search is only guaranteed to be the best possible PALS insertion if ι 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 ι 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.



■ **Figure 2** Example *dropoff after last stop* insertions that illustrate why domination between dropoffs is a partial relation. Shows existing vehicle route (solid) with current last two stops ($s_{k(\nu)-1}, s_{k(\nu)}$), two pickups between these stops (p_1, p_2) and two dropoffs after the last stop (d_1, d_2) with necessary detours (dashed), as well as the walking distances from either dropoff to the destination (dotted). Edges are annotated with vehicle travel times or passenger travel times (marked with p).

8 Dropoff After Last Stop Insertions

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 ν if $c(\iota_1) < c(\iota_2)$ for any DALS insertions $\iota_1 = (r, p, d_1, \nu, i, k(\nu))$ and $\iota_2 = (r, p, d_2, \nu, i, k(\nu))$ 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 ι_1 and ι_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 ι_1 is better for some choice of p and i while ι_2 may be better for another.

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

Consider the insertions $\iota_{1i} = (r, p_1, d_i, \nu, k(\nu) - 1, k(\nu))$ for $i = 1, 2$. Neither insertion violates the trip time soft constraint as $t_{trip}(\iota_{11}) = 3 + 1 + 1 + 4 = 9$ and $t_{trip}(\iota_{12}) = 3 + 1 + 3 + 1 = 8$. The difference in detours is $t_{detour}(\iota_{11}) - t_{detour}(\iota_{12}) = 5 - 7 = -2$, 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 $\iota_{2i} = (r, p_2, d_i, \nu, k(\nu) - 1, k(\nu))$ for $i = 1, 2$. Here, both insertions violate the trip time soft constraint since $t_{trip}(\iota_{21}) = 12$ and $t_{trip}(\iota_{22}) = 11$. 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.

ρ	B-1%		B-10%		R-1%		R-10%	
	N_p	N_d	N_p	N_d	N_p	N_d	N_p	N_d
0s	1	1	1	1	1	1	1	1
150s	10	10	10	10	12	11	12	11
300s	33	31	33	31	35	33	35	33
450s	68	66	68	67	70	68	70	68
600s	115	113	115	114	114	112	114	112

penalties, we get $c(\iota_{21}) - c(\iota_{22}) = -1 + 10 = 9$. 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 $(d, \delta^\downarrow(v, d))$ at each vertex $v \in V$ with $d \in D_\rho$. Initially, an open label $(d, 0)$ is created at every $d \in D_\rho$. 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^\downarrow .

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 $l_i = (d_i, \delta^\downarrow(v, d_i))$ for $i = 1, 2$ be two dropoff labels at $v \in V$. Let

$$\Delta_{detour}(l_1, l_2) := \delta^\downarrow(v, d_1) - \delta^\downarrow(v, d_2)$$

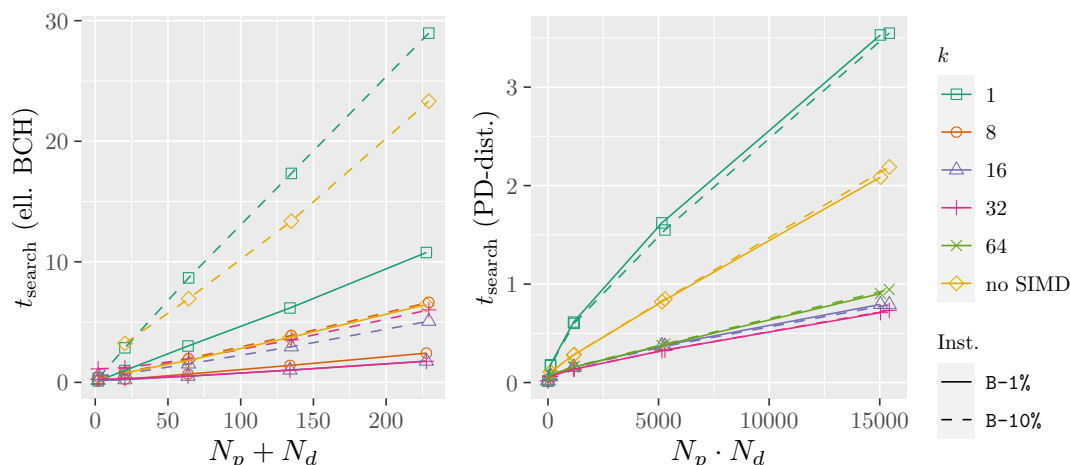
$$\Delta_{trip}(l_1, l_2) := \delta^\downarrow(v, d_1) + \delta_{psg}(d_1, dest) - \delta^\downarrow(v, d_2) - \delta_{psg}(d_2, dest)$$

Then d_1 dominates d_2 if

1. $\Delta_{detour}(l_1, l_2) + \tau \Delta_{trip}(l_1, l_2) < 0$ and
2. $\Delta_{detour}(l_1, l_2) + (\tau + \gamma_{trip}) \Delta_{trip}(l_1, l_2) < 0$

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

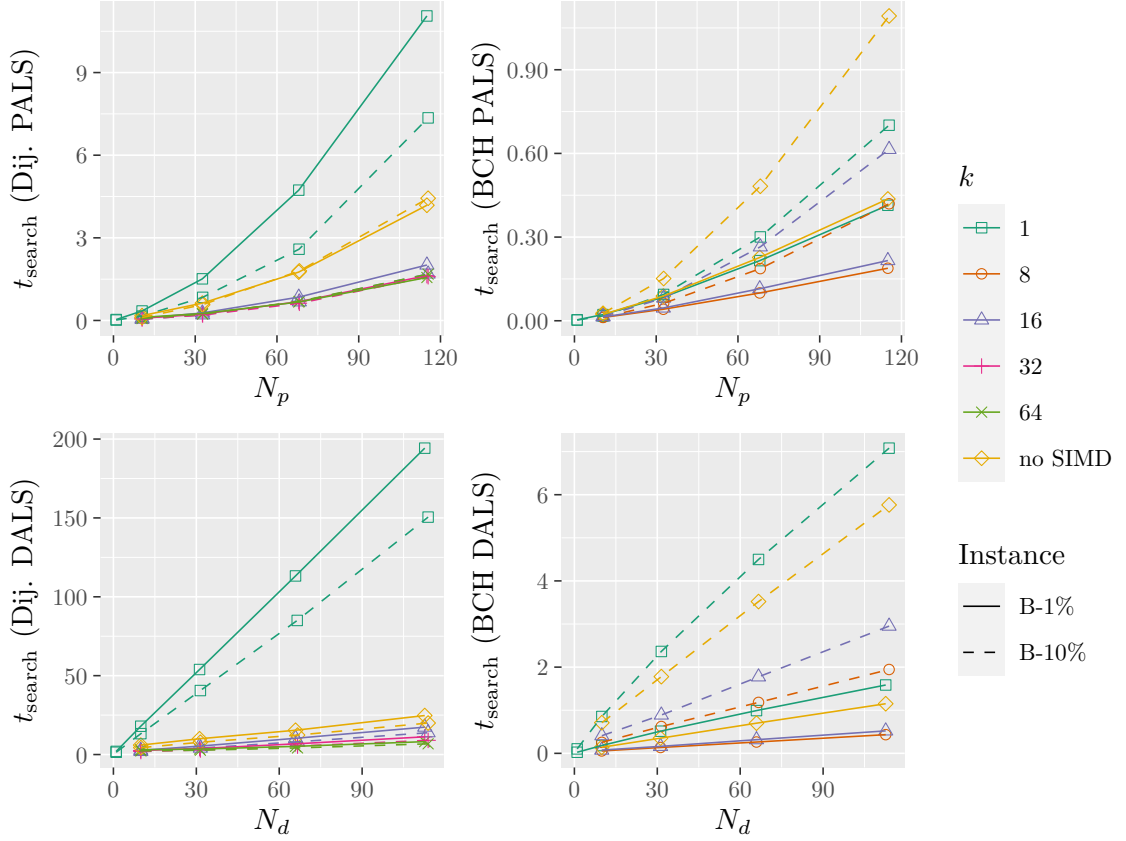


■ **Figure 3** Evaluation of bundled elliptic BCH searches (left) and PD-distance searches (right). Shows mean search times (in ms) for either search on the **Berlin-1pct** and **Berlin-10pct** instances with $\rho \in \{0s, 150s, 300s, 450s, 600s\}$. Considers $k \in \{1, 8, 16, 32\}$ for elliptic BCH 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.

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 data¹. 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 4.5km/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 ρ and report the average running times.

For our model parameters, we choose $t_{\text{wait}}^{\text{max}} = 600s$, $t_{\text{stop}}^{\text{min}} = 60s$, $\gamma_{\text{wait}} = 1$, $\gamma_{\text{trip}} = 10$, $\tau = 1$, and $\omega = 0$. We use the open-source library RoutingKit² 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 4** Evaluation of bundled last stop searches. Shows mean search times (in ms) for bundled Dijkstra searches (left) and individual BCH searches (right) in the PALS (top) and DALs (bottom) cases on the `Berlin-1pct` and `Berlin-10pct` instances with $\rho \in \{0s, 150s, 300s, 450s, 600s\}$. Considers $k \in \{1, 16, 32, 64\}$ for Dijkstra searches and $k \in \{1, 8, 16\}$ for BCH searches, using SIMD instructions for $k > 1$. Additionally shows running times without SIMD instructions for Dijkstra searches with $k = 64$ and BCH searches with $k = 8$. Note the different 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.

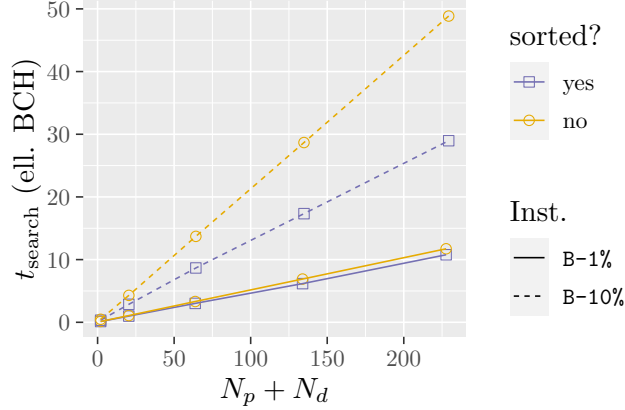
Bundled Last Stop Searches. We depict the running times of bundled Dijkstra searches and individual BCH searches for the PALS and DALs cases in Figure 4.

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



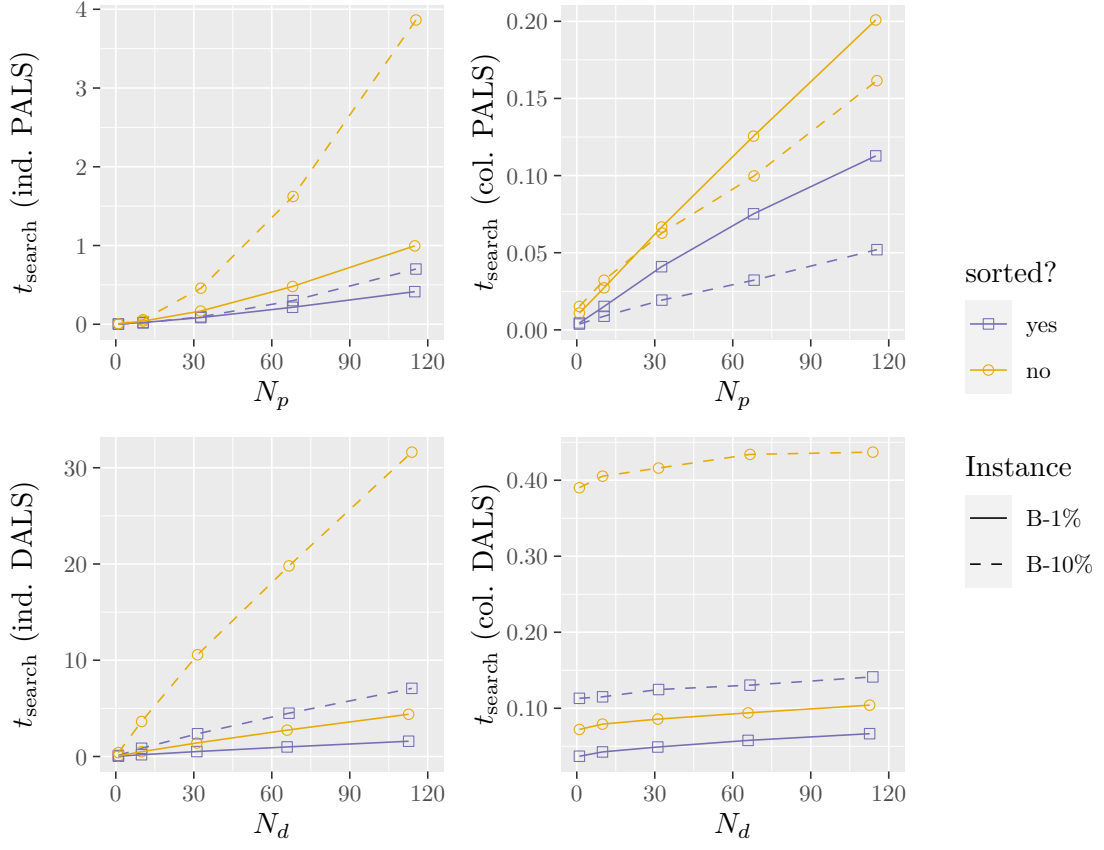
■ **Figure 5** Effectiveness of sorted buckets on running time of elliptic BCH queries. Shows mean running times (in ms) at $k = 1$ on the **Berlin-1pct** and **Berlin-10pct** instances for $\rho \in \{0s, 150s, 300s, 450s, 600s\}$.

of bucket entries scanned as well as the effects on the running time of the searches and the time for updating buckets. We experimentally compare all searches with sorted and unsorted buckets on the **Berlin-1pct** and **Berlin-10pct** instances with $\rho \in \{0s, 150s, 300s, 450s, 600s\}$. For elliptic BCH searches and individual last stop BCH searches, we use $k = 1$. We only run a single iteration with unsorted buckets for elliptic BCH searches and individual last stop BCH searches.

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 $10\mu s$. 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 $20\mu s$ per request while the reduction in running time is between one and three orders of magnitude larger.



■ **Figure 6** Effectiveness of sorted buckets on running time of last stop BCH queries. Shows mean running times (in ms) of individual BCH queries (left; $k = 1$) and collective BCH queries (right) in the PALS (top) and DALC (bottom) cases on the **Berlin-1pct** and **Berlin-10pct** instances for $\rho \in \{0s, 150s, 300s, 450s, 600s\}$ with and without sorted buckets. Note the different y -axes.

9.3 Collective BCH Searches

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 DALC 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 DALC 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

■ **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 DAL S case and three radii $\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$ on the **Berlin-1pct** and **Berlin-10pct** instances. Shows number of edge relaxations ($\#\text{rel.}$), number of bucket entries scanned ($\#\text{scans}$), mean search time (t_{search}) and the mean time for enumerating possible insertions (t_{enum}) per request. Times marked in bold are the smallest times per radius.

		Berlin-1pct					Berlin-10pct			
	ρ	Alg.	$\#\text{rel.}$	$\#\text{scans}$	t_{search}	t_{enum}	$\#\text{rel.}$	$\#\text{scans}$	t_{search}	t_{enum}
P A L S	0	Coll.	33	7	4.46	0.49	12	12	3.66	0.53
		BCH	30	7	3.16	0.37	12	7	2.31	0.42
		Dij.	459	–	33.26	0.30	179	–	14.38	0.34
	300	Coll.	225	34	40.98	0.65	68	33	19.47	0.68
		BCH	493	152	41.20	27.55	357	570	62.02	93.33
		Dij.	2125	–	255.21	24.64	1599	–	228.07	80.14
	600	Coll.	439	64	112.81	0.79	100	52	52.47	0.89
		BCH	2243	1135	191.32	408.40	1661	5530	420.25	1839.26
		Dij.	14821	–	1551.77	366.51	12852	–	1689.69	1681.56
D A L S	0	Coll.	177	1077	36.31	3.15	154	7895	112.76	6.89
		BCH	183	1128	26.14	3.12	159	8283	116.17	7.27
		Dij.	22317	–	1978.23	28.55	15780	–	1411.81	132.11
	300	Coll.	197	1053	48.74	7.46	173	7857	126.67	20.13
		BCH	1132	4792	134.84	97.13	1035	37684	605.15	363.65
		Dij.	24762	–	3444.91	174.46	18522	–	2667.37	694.29
	600	Coll.	213	1043	66.51	14.84	189	7837	141.38	40.57
		BCH	3868	15484	432.85	754.17	3509	122199	1967.98	2403.68
		Dij.	62087	–	8149.34	1639.57	49489	–	6807.04	5992.58

every PD-pair or dropoff. This applies to the stopping criteria for bucket scans and for the searches as a whole.

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 k searches 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 with the number of PD-locations ($N_p, N_d \sim \rho^2$) for individual BCH searches than for collective BCH searches. In fact, domination pruning works so well in the DAL S 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 ρ 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 ρ^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 ($\rho = \text{L} \dots$) and KaRRi with different radii ($\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$) on B-1%, B-10%, R-1%, and R-10%. Shows mean times for finding P_ρ and D_ρ , PD-distance searches, elliptic BCH searches, enumerating ordinary and PBNS insertions, PALS and DALIS searches, and updating routes and buckets as well as the mean total time per request. Numbers with asterisks are estimates.

Inst.	ρ	find P_ρ, D_ρ	PD	BCH	Ord.& PBNS	PALS	DALS	update	total
B-1%	L-0	0	17	124	147	43	1974	84	2388
	0	2	36	123	42	3	28	132	367
	300	95	155	497	119	43	56	138	1103
	600	316	735	1703	469	156	81	140	3601
B-10%	L-0	0	16	368	415	23	1474	80	2376
	0	3	35	362	353	3	119	200	1074
	L-300	*104	*18930	28675	736	918	44071	*80	*93514
	300	104	157	1502	641	24	140	203	2772
	L-600	*353	*246521	48856	1900	8998	166244	*80	*472952
	600	353	759	4804	1741	106	181	213	8158
R-1%	L-0	0	19	173	228	128	8572	82	9202
	0	3	18	163	106	5	34	138	481
	300	98	132	668	194	62	59	146	1375
	600	293	566	1989	494	216	81	149	3807
R-10%	L-0	0	18	703	944	89	6154	80	7988
	0	3	19	708	944	5	163	298	2155
	300	106	138	3360	1293	42	182	320	5462
	600	324	595	9396	2476	128	226	387	13554

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 $\rho \in \{0\text{s}, 300\text{s}, 600\text{s}\}$ 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 ($\rho = 0\text{s}$) 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 $\rho = 0\text{s}$, though. They are up to 27 and 250 times faster than LOUD’s Dijkstra searches in the PALS and DALIS 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 DALIS 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).

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

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 ρ , 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

- 1 Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithms (JEA)*, 14, 2010. doi:10.1145/1498698.1537599.
- 2 Joschka Bischoff, Michal Maciejewski, and Kai Nagel. City-Wide Shared Taxis: A Simulation Study in Berlin. In *IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, 2017. doi:10.1109/ITSC.2017.8317926.
- 3 Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Real-Time Traffic Assignment using Engineered Customizable Contraction Hierarchies. *ACM Journal of Experimental Algorithms (JEA)*, 24, 2019. doi:10.1145/3362693.
- 4 Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Fast, Exact and Scalable Dynamic Ridesharing. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, 2021. doi:10.1137/1.9781611976472.8.
- 5 Daniel Delling, Andrew V Goldberg, Andreas Nowatzyk, and Renato F Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. *Journal of Parallel and Distributed Computing*, 73, 2013. doi:10.1016/j.jpdc.2012.02.007.
- 6 Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Customizable Route Planning in Road Networks. *INFORMS Transportation Science*, 51(2), 2017. doi:10.1287/trsc.2014.0579.
- 7 Daniel Delling, Andrew V Goldberg, and Renato F Werneck. Faster Batched Shortest Paths in Road Networks. In *11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*. LIPIcs, 2011. doi:10.4230/OASIcs.ATMOS.2011.52.
- 8 Edsger W Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1, 1959.
- 9 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks using Contraction Hierarchies. *INFORMS Transportation Science*, 46(3), 2012. doi:10.1287/trsc.1110.0401.
- 10 Preeti Goel, Lars Kulik, and Kotagiri Ramamohanarao. Optimal Pick up Point Selection for Effective Ride Sharing. *IEEE Transactions on Big Data*, 3(2), 2017. doi:10.1109/TBDATA.2016.2599936.

- 11 Moritz Hilger, Ekkehard Köhler, Rolf H Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. *The Shortest Path Problem: 9th DIMACS Implementation Challenge*, 2009.
- 12 Andreas Horni, Kai Nagel, and Kay Axhausen, editors. *The Multi-Agent Transport Simulation MATSim*. Ubiquity Press, 2016. doi:10.5334/baw.
- 13 Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing Many-to-Many Shortest Paths using Highway Hierarchies. In *SIAM Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007. doi:10.1137/1.9781611972870.4.
- 14 Catherine Morency. The Ambivalence of Ridesharing. *Transportation*, 34(2), 2007. doi:10.1007/s11116-006-9101-9.
- 15 Changle Song, Julien Monteil, Jean-Luc Ygnace, and David Rey. Incentives for Ridesharing: A Case Study of Welfare and Traffic Congestion. *Journal of Advanced Transportation*, 2021. doi:10.1155/2021/6627660.
- 16 Mitja Stiglic, Niels Agatz, Martin Savelsbergh, and Mirko Gradisar. The Benefits of Meeting Points in Ride-Sharing Systems. *Transportation Research Part B: Methodological*, 82, 2015. doi:10.1016/j.trb.2015.07.025.
- 17 Hiroki Yanagisawa. A Multi-Source Label-Correcting Algorithm for the All-Pairs Shortest Paths Problem. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010. doi:10.1109/IPDPS.2010.5470362.
- 18 Biying Yu, Ye Ma, Meimei Xue, Baojun Tang, Bin Wang, Jinyue Yan, and Yi-Ming Wei. Environmental Benefits from Ridesharing: A Case of Beijing. *Applied Energy*, 191, 2017. doi:10.1016/j.apenergy.2017.01.052.
- 19 Dominik Ziemke, Ihab Kaddoura, and Kai Nagel. The MATSim Open Berlin Scenario: A Multimodal Agent-Based Transport Simulation Scenario Based on Synthetic Demand Modeling and Open Data. *Procedia Computer Science*, 151, 2019. doi:10.1016/j.procs.2019.04.120.