Master thesis

Improving Vehicle Detour in Dynamic Ridesharing using Transfer Stops

Max Willich

Date: 2. Oktober 2023

Supervisors: Prof. Dr. Peter Sanders
M.Sc. Moritz Laupichler

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

Abstract

Ridesharing companies like *Uber* or *Lyft* provide an alternative to classic public transport, in which you can order a ride from anywhere to anywhere within the service area via an app. A vehicle will be assigned to you, pick you up and drive you to your destination. In services like UberXShare or LyftShared, other passengers might also be picked up on the way, causing a small detour. The process of matching your ridesharing request to some vehicle in the set of available vehicles is resolved using a *dispatching algorithm*.

Buchhold et al. in [3] developed the dispatching algorithm LOUD for the dynamic ridesharing scenario. It is able to find the optimal solution to a ridesharing request (based on a given cost function) in only a few milliseconds on realistically sized scenarios. In this thesis, we aim to add transfer stops to ridesharing by extending the LOUD-algorithm. We present multiple transfer-dispatching algorithms and implement and evaluate each of these algorithms thoroughly, analyzing their runtime performance and dispatching quality on various different scenarios.

Our best algorithm, where transfer vertices are sampled by *betweenness*, is able to find a better solution to a ridesharing request than LOUD in up to 20% of all requests, while only increasing the required runtime by about one order of magnitude. Furthermore, we are able to show that multi-transfer routes can bring substantial improvements over no- or one-transfer-routes on long-distance requests, and we show that runtime performance scales well with the number of requested transfer stops.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 29. September 2023

Contents

Abstract iii

1 Introduction 1

2 Related Work 3

3 Preliminaries 5
3.1 Dynamic Ridesharing Problem . . . 5
3.2 Dijkstra's Algorithm . . . 7
3.3 Contraction Hierarchies . . . 9
3.4 Bucket-CHs . . . 10
3.5 LOUD . . . 11

4 Ridesharing with Transfers 15
4.1 Changes to the ridesharing problem . . . 15
4.2 Transfer-At-Stop Algorithm . . . 16
4.3 Sampled-Transfer Algorithm . . . 17
4.4 Multi-Transfer Dispatching . . . 20

5 Evaluation 27
5.1 Setup . . . 27
5.2 Analysis of No-Transfer-Routes . . . 28
5.3 Analysis of the test scenarios . . . 30
5.4 Runtime Performance of single-transfer dispatching . . . 33
5.5 Dispatch Quality . . . 34
5.6 Algorithm-Specific Evaluation . . . 36
5.7 Vehicle Utilization . . . 44
5.8 Varying vehicle capacities . . . 46
5.9 Two-Transfer Dispatching . . . 49
5.10 Running on a larger scenario . . . 52
5.11 Combined single- and two-transfer dispatching . . . 55
5.12 More than two transfer stops . . . 57

6 Conclusion 63

7 Appendix **65**

Bibliography **69**

1 Introduction

Ridesharing services like Uber or Lyft serve as a cheaper and modern alternative to traditional taxis. Instead of having to call the taxi company to order a ride, you can order your ride via an app by simply entering your desired pickup- and dropoff-location. When ordering a ride, you will then be assigned a vehicle/driver that will pick you up at your requested pickup-location and drive you to your destination. Furthermore, with some services like UberXShare or LyftShared, as the name *ridesharing* suggests, it is possible that while en-route to your destination, your vehicle may pick up a second set of passengers on the way. This way, the fleet of vehicles is used more efficiently, reducing wait times and costs for passengers.

When a passenger request is made, a *dispatching algorithm* determines which vehicle picks up a passenger and drives them to their destination. It runs on a server owned by the ridesharing company and has to answer many requests each second. One recently developed dispatching algorithm is called LOUD (short for *Local Buckets Dispatching*) and can answer a ridesharing request on a road network the size of Berlin and a fleet of a thousand vehicles within only about two milliseconds on average on consumer hardware. It minimizes the detour required by all vehicles to serve the requests while keeping constraints on the passenger's wait- and travel-time.

Up until now, LOUD and many other dispatching algorithms are only able to resolve requests using direct trips. Transfer stops however could potentially reduce the total detour required to resolve a ridesharing request significantly. For example, if a vehicle exists that passes very close to the requests' pickup location on its route already, but doesn't get close to the requested dropoff location, the vehicle may not be viable for serving the passenger's request, as the detour needed to drive to the dropoff location would be too high. But with transfer stops, that vehicle could at least pick the passenger up and drive them half-way to a transfer location, where a second vehicle, which passes closely by the transfer and dropoff location, then drives the passenger to their destination. Increasing the utilization of the available vehicles means that fewer vehicles are needed to serve all customers, which reduces costs for the ridesharing provider and for the passengers.

In chapter chapter 4 we present three different dispatching algorithms for resolving ridesharing requests using a single transfer stop: One algorithm that builds directly up on LOUD and uses its internal data to quickly find viable transfer trips, and two algorithms that sample good transfer stops using some heuristic. The first sampling algorithm uses a geometric heuristic while the second one uses a heuristic based on *vertex betweenness*. We then expand each of these algorithms for multi-transfer dispatching to allow for more than one

transfer stop.

In chapter 5, we comprehensively evaluate each of the presented single- and multi-transfer algorithms for dispatching quality and runtime performance. For the evaluation we use realistic ridesharing scenarios based on road networks of Berlin and the German Ruhr-area. Our evaluation shows that our algorithms can find lower-cost solutions to ridesharing requests than LOUD for up to 20% of all requests while still maintaining viable runtime performance. We are also able to show that multi-transfer routes can bring a significant improvement to larger ride-sharing scenarios, and that our algorithms scale well with the number of requested transfer stops per route.

2 Related Work

City-wide shared taxis: a simulation study in Berlin [2]

Bischoff et al. present a simple dispatching algorithm, which resolves ridesharing requests using a brute-force approach: When a ridesharing request is made, the algorithm first iterates over *all* possible resolutions to that request and estimates the increase in vehicle operation time using a geometric heuristic. It then filters all solutions that are not deemed feasible based on that geometric estimate. For all remaining solutions, exact detours are calculated using Dijkstra's algorithm and the best solution is selected. The authors provide a comprehensive evaluation of ridesharing services as an alternative to taxis by integrating their algorithm into the transport simulation software *MATSim* and running it on the road network of Berlin. The evaluation encompasses, among other things, the potential detour decrease through sharing rides in comparison to traditional taxi-services, the average occupancy of vehicles depending on location (e.g. the average occupancy is highest near Berlin's largest airport) and the amount of idle vehicles at any point in time throughout the simulation.

Fast, Exact and Scalable Dynamic Ridesharing [3]

In [3], Valentin Buchhold, Peter Sanders and Dorothea Wagner present the LOUD-Algorithm, an algorithm that finds the optimal solution the dynamic ridesharing problem. It improves on the algorithm presented in [2], offering faster runtime performance and removing the need for the inaccurate geometric heuristic. The algorithm is faster than comparable algorithms for the dynamic ridesharing problem by at least one order of magnitude. The key to this speedup is the use of *bucket-based contraction hierarchies* (BCH), a routing technique that extends the point-to-point routing technique *contraction hierarchies* (CH) to allow for very fast many-to-one queries. Furthermore, Buchhold et al. use a pruning technique called *elliptic pruning* to greatly reduce the search space required to find the optimal solution without affecting optimality. This allows for much faster distance calculations than competing algorithms, which mainly use Dijkstra's algorithm for many-to-one queries. The LOUD-Algorithm is described in detail in chapter 3 and serves as the basis for our work.

Real-time Transfers for Improving Efficiency of Ridesharing Services in the Environment with Connected and Self-driving Vehicles [4]

Chen et al. also investigate the possible benefits of transfer-routes for the ridesharing scenario. In their algorithm, a passenger is first assigned a route using a conventional no-

transfer algorithm. Every time the vehicle passes along an edge (i.e. road) that could serve as a potential transfer stop, a new no-transfer ridesharing request is made from the current location to the requested dropoff location. Based on some heuristics described in the paper, the passenger will then either transfer at that location or continue along the original route. Chen et al. only evaluate single-transfer routes by only allowing a single transfer in the heuristic, but multi-transfer routes are trivial to implement using their technique as well. Unlike our scenario, the objective function given in [4] optimizes for passenger travel time. The authors have found a significant decrease in passenger travel time by introducing single-transfer routes. Note that all passenger wait times are included in this travel time.

The authors do not evaluate the runtime performance of their algorithm. However, since a ridesharing request has to be made on every edge along the route of a vehicle, we conjecture that the performance of their algorithm is too restrictive for our use-case. Chen et al. evaluate their algorithm on a small road network graph of Manhattan, containing only about 1600 edges and 800 vertices, whereas our test road network contains more than 100000 vertices and edges.

Dynamic Carpooling

Dynamic Carpooling is a closely related problem to the dynamic ridesharing problem. Here, vehicles have a fixed source and destination that they drive from/to. Along the way, the vehicle might make a detour to pick up additional passengers, a sort of *on-demand hitchhiking*. While dynamic ridesharing is used mostly within the confines of one urban area, dynamic carpooling, while still viable for urban scenarios, is especially useful for long distance inter-city travel. Again, a dispatching algorithm is needed to find a vehicle to pick passengers up and take them along for the ride.

In [12], Pelzer et al. present an algorithm for resolving various different types of ridesharing optimization problems, and evaluate their algorithm by resolving carpooling requests in a transport simulation on the road network of Singapore. Their algorithm uses a partitioning algorithm to create partitions of the road network that match its topology, and matches passengers to potential rides that lie in the same partition.

3 Preliminaries

3.1 Dynamic Ridesharing Problem

To begin, we formulate the problem statement of the dynamic ridesharing problem, describing all the data that makes up the ridesharing scenario as well as the optimization problem that is to be solved.

Road Network:
The road network, on which requests are made and rides are shared, is modeled as a directed graph $G = (V, E)$ with non-negative integer edge weights. Each vertex represents an intersection and each edge a road between two intersections. The edge weight $w(e)$ of an edge describes the travel time from one intersection to the next.

Ridesharing Request:
A ridesharing request is a 3-tuple $r = (r_t, r_p, r_d)$.

- $r_t \in \mathbb{N}$ is the point in time at which the request is made.
- $r_p \in V$ is the pickup location of this request.
- $r_d \in V$ is the dropoff location of this request.

We call r_p and r_d the *pickup-* and *dropoff-vertex*. Prebooking is not allowed. A passenger is available for pickup the moment they make a request.

Request Constraints:
Each request r comes with two time constraints: t_{wait}^{max} and t_{trip}^{max}, describing the maximum wait- and trip-time of a passenger. t_{wait}^{max} is simply a constant. t_{trip}^{max} is equal to $\alpha \cdot dist(r_p, r_d) + \beta$, where $\alpha \in \mathbb{R}$ and $\beta \in \mathbb{N}$ are constant parameters and $dist(r_p, r_d)$ is the direct distance from the pickup location to the dropoff location on the graph G.

Vehicle:
A vehicle is a 4-tuple $v = (v_l, v_c, v_t^*, v_t^\dagger)$.

- v_l is the initial start location of the vehicle.
- v_c is the maximum passenger capacity of the vehicle, excluding the driver.
- v_t^* is the point in time at which the vehicle starts its service.

- $v_t^\dagger$ is the point in time at which the vehicle ends its service.

The time between v_t^* and $v_t^\dagger$ is called the *vehicle service time*. No passenger can be picked up or dropped off outside of this time range. Outside of a vehicles' service range, the vehicle can be assumed to simply not exist at all. The set of all vehicles available to the ridesharing provider is called the *fleet*.

Vehicle Route:
A vehicle stop is a 4-tuple $s = (s_{dep}^{min}, s_{arr}^{max}, s_{occup}, s_{loc})$.

- s_{dep}^{min} is the current departure time of the vehicle at that stop.
- s_{arr}^{max} is the latest possible arrival time without violating any request constraints by arriving too late at either this or any succeeding stop.
- $s_{occup} \in \mathbb{N}$ is the occupancy of the vehicle at that stop.
- $s_{loc} \in V$ is the location of the stop.

Each vehicle has a *route*, which is a list of vehicle stops. Each stop takes a constant time t_{stop}. Therefore, the arrival time at a stop can be calculated directly from the departure time by simply subtracting this constant stop duration. The first stop in a route is either the stop that the vehicle is currently stopping on or the last visited stop. Once a vehicle visits the second stop in a route, the first stop gets removed, making the old second stop the new first stop. Initially, every route has length 1 and matches the initial location and startup-time of the vehicle. Each vehicle starts with no passengers.

Insertion:
An *insertion* is the result of resolving a ridesharing request and is described by a 3-tuple $i = (i_v, i_p, i_d)$.

- i_v is the vehicle in which the passenger gets taken from the pickup to the dropoff.
- i_p is the index (zero-indexed) of the stop in the route of i_v after which the new pickup stop gets inserted.
- i_d is the index (zero-indexed) of the stop in the route of i_v after which the new dropoff stop gets inserted.

For example, if $i_p = 1$, then after the second stop of vehicle i_v, a new stop gets inserted into the route to pick up the new passenger at the pickup-location. If the location of the second stop is already equal to the pickup-location, no new stop is inserted since the new passenger can be picked up at the existing stop.

When an insertion into a route is performed, a detour is caused by the newly created stops. The detour gets propagated through the route, updating the stops minimum departure times accordingly. Furthermore, since each request comes with its own set of time constraints, these new constraints get propagated through the route as well, updating the maximum arrival times of each stop. Finally, the occupancy of a stop also gets increased by one for

all stops between the new pickup stop and the dropoff stop. To perform an insertion, the request and its time constraints are required in addition to the actual insertion.

Insertion Cost Function:
A cost function assigns each insertion a cost. It is defined as:

$$c(i) = \delta + c_{wait} \cdot max(0, t_{wait} - t_{wait}^{max}) + c_{trip} \cdot max(0, t_{trip} - t_{trip}^{max})$$

in [3], with

- δ being the caused detour in the route (i.e. the difference in arrival time at the routes last stop before and after insertion).
- t_{wait} being the time the passenger has to wait until the vehicle arrives.
- t_{trip} being the difference between the vehicles' arrival at the dropoff location and the departure time at the pickup.
- c_{wait}, c_{trip} being model parameters for the cost of violating the request constraints.

This insertion cost function optimized both for passenger comfort (wait- and trip-time) as well as total travel time of all vehicles (detour). It models the requirements on ridesharing in real-life scenarios. While other cost functions are also feasible, we continue using the presented cost function from [3] in order to maintain comparability to existing results.

Note that while the request constraints are soft for a new insertion, they are hard-constraints for any already-resolved request in the route. If the new insertion causes a constraint of any other already-matched request to be violated (enforced with the maximum arrival times of each stop), the cost of the insertion is ∞. Similarly, the cost is ∞ if the vehicle would be overfull or would have to drive outside its service time.

3.2 Dijkstra's Algorithm

Djkstra's algorithm [6] is an algorithm to find the shortest path from a single source vertex to all other vertices in a Graph $G = (V, E)$ with a non-negative edge weight function $w : E \to \mathbb{N}$. Pseudocode for dijkstra's algorithm is given in algorithm 1

Q is an initially-empty priority queue. d and par are arrays that map vertices to their best-found distance and parent vertex. While Dijkstra's Algorithm is running, note that the key of a vertex v in Q is always set to $d(v)$.

Initially, all vertices have infinite distance to the source vertex s, except for the source vertex itself, which has the distance 0. Starting from line 10 of the pseudocode, the vertex u with the smallest best found distance from the source gets popped from the queue. For every neighbour v of u, the algorithm tries to improve the shortest distance to v by comparing the length of the path to v via u to the currently best found path (Line 16). If the new path

Algorithm 1 Dijkstra's Algorithm

Input: $G = (V, E)$, $s \in \mathbb{V}$

1: **for** $v \in V$ **do** ▷ Initialize priority queue and arrays
2: $\quad d(v) \leftarrow \infty$
3: $\quad par(v) \leftarrow \perp$
4: $\quad Q.insert(v, \infty)$
5: **end for**
6: $d(s) \leftarrow 0$ ▷ Set s as source vertex
7: $Q.decreaseKey(s, 0)$
8: **while** $|Q| \geq 0$ **do** ▷ Run until priority queue is empty
9: $\quad u = Q.pop()$
10:
11: $\quad$ **for** $e = (u, v) \in E$ **do** ▷ Update neighbours of u
12: $\quad\quad d' \leftarrow d(u) + w(e)$
13:
14: $\quad\quad$ **if** $d' \geq d(v)$ **then**
15: $\quad\quad\quad$ **continue**
16: $\quad\quad$ **end if**
17:
18: $\quad\quad d(v) \leftarrow d'$ ▷ New shortest path to v via u found
19: $\quad\quad par(v) \leftarrow u$
20: $\quad\quad Q.decreaseKey(v, d')$
21: $\quad$ **end for**
22: **end while**

is better, the distance, parent and key in the queue of v gets updated accordingly (Line 20 and beyond).

Note that Dijkstra's Algorithm is *label-setting*, which means that once a vertex gets popped from the queue, it is guaranteed that the shortest path to that vertex was found. Therefore, once the queue is empty, all shortest paths to all popped vertices have been found. Nodes that were not visited are not reachable from the source vertex and have a shortest distance of ∞.

A popular extension to Dijkstra's Algorithm is to introduce a target vertex t. After the Node u gets popped in line 11, if $u = t$ the algorithm can stop since the shortest path to the target vertex t was found. This is an example of a *stopping criterion*. Other stopping criterions for Dijkstra's algorithm are feasible and will be used throughout this work.

3.3 Contraction Hierarchies

Contraction Hierarchies is a two-phase routing technique first presented in [8] by Geisberger et. al. that answers point-to-point shortest path queries on road networks many orders of magnitude faster than a simple dijkstra query would. The first phase is a preprocessing phase that might take up to an hour, but only has to be run once for any given graph. The results of the preprocessing phase can then be stored on disk. The second phase is the query phase, in which point-to-point shortest-path queries are performed on the preprocessed graph.

The foundation for the preprocessing phase is the *vertex contraction*, hence the name *Contraction Hierarchies*. The goal of a vertex contraction is to remove a vertex from the graph while maintaining the shortest paths/distances between all other vertices. To achieve this, a *shortcut* edge has to be inserted into the graph for every pair of incoming and outgoing edges of the contracted vertex. For example: Suppose $n \in V$ is the vertex to be contracted, and imagine there's an incoming edge $e_{in} = (u, n)$ and an outgoing edge $e_{out} = (n, v)$ adjacent to n. In this case, a shortcut edge (u, v) is created with weight $w(e_{in}) + w(e_{out})$. This way, any shortest path that previously contained the sequence $(\dots, u, n, v, \dots)$ can now "skip" over n without changing the length of that shortest path. If such a shortcut is inserted for every pair of incoming and outgoing edges, all shortest path distances are maintained.
A key insight here is that a shortcut does not need to be created for e_{in} and e_{out} if there is a shorter path from u to v than (u, n, v). In that case, there is no shortest path of the form $(\dots, u, n, v, \dots)$, since the path could be shortened by using the shorter path from u to v. Therefore, when contracting a vertex n, for every pair $e_{in} = (u, n)$ and $e_{out} = (n, v)$, Dijkstra's algorithm is run, starting from u and with target v to check if there's a shorter path from u to v than (u, n, v). A shortcut (u, v) gets created only when (u, n, v) is the shortest path from u to v.

For preprocessing a graph G, the graph first gets copied twice, we call these copies the *contracted graph* G' and the *augmented graph* G^*. On the contracted graph G', all vertices are contracted sequentially in an order that tries to minimize the amount of shortcuts created. Determining this order is NP-Hard (proven in [5]), therefore heuristics for this order are used. Every time a vertex gets contracted in G', all created shortcuts are added to G^*. This is done until all vertices in G' were contracted and the graph is empty. The order of contraction now forms a hierarchy on the vertices: A vertex is higher up in the hierarchy the later it was contracted. We say that if $n \in V$ was the i-th vertex contracted, then n has the *rank* i.

The preprocessing phase is now done. To figure out the shortest-path distance between two vertices $s \in V$ and $t \in V$ in the graph G, two Dijkstra queries are started: The *forward query* has the source vertex s and searches along the augmented graph G^*. And the *backward query*, which has the source vertex t and searches along the *reverse* augmented graph G^*_R. Furthermore, both queries only explore edges that go *up* in the hierarchy, i.e.

an edge (u, v) is only explored if u has a smaller rank than v. The queries are run in an alternating manner, first doing a step of the forward query, then the reverse query, then the forward query again etc. As the queries are running, the best found distance μ is stored. When one query explores an edge (u, v) and v has already been settled by the other query, a new path from s to t has been found, its length being $d_f(v) + d_r(v)$, with d_f and d_r being the tentative distance functions of the forward and reverse queries. If this new path is shorter, then μ is set to the length of the found path. The queries can be stopped once one of the minimum keys of both priority queues exceed μ, since it can be guranteed that no shorter path will be found.

This query works for finding the point-to-point shortest path *distances* between two vertices. If the actual shortest path along G is required, first the contracted path needs to be extracted. Let n be the vertex where the forward and reverse queries intersected and found the shortest path from s to t. To extract the full contracted path from s to t, simply extract the path from s to n of the forward query and concatenate it with the reverse of the path from t to n of the reverse query. Next, the path needs to be recursively *unpacked* by replacing every contained shortcut edge with the two edges that this shortcut replaced. Note that these two edges might be shortcut edges again, therefore recursive unpacking is necessary. To achieve this, in the preprocessing phase, this information needs to be stored for every created shortcut edge. The fully unpacked path is then the shortest path from s to t on the graph G.

3.4 Bucket-CHs

In [11], Knopp et. al present an approach to extend any hierarchy-based routing technique for many-to-one queries. Applying this to CHs results in *Bucket Contraction Hierarchies* (abbreviated BCH). BCHs are a routing technique to answer one-to-many shortest distance queries, i.e. from any given source vertex s to a set of target vertices $T = \{t_1, t_2, \ldots\}$. In Bucket-CHs, a *bucket* is stored on every vertex of the reverse graph of a contraction hierarchy. A bucket is simply a list of tuples $(t, d_t) \in T \times \mathbb{N}$, with each being initially empty. For every target vertex $t \in T$, a Dijkstra query is started on the reverse graph G_R of the contraction hierarchy and is run until its queue is empty, exploring the entire search space. For every settled vertex n, an entry (t, d_t) is inserted into the bucket of the settled vertex, where d_t is the distance from n to t found by the reverse Dijkstra query.

To query the distances from any source vertex $s \in V$ to all vertices in T, run a Dijkstra query on the forward graph G_F of the contraction hierarchy. Maintain a tentative distance μ_t for each target vertex in T. For every settled vertex u, scan the entries in the bucket of that vertex. For any entry (t, d_t) in that bucket, $d(s, u) + d_t$ is a new possible shortest distance from s to t. Update the tentative distance μ_t accordingly. The bucket entries essentially serve as a replacement for the reverse query that would be required in ordinary CH-Queries, since the distances to the target vertices are precomputed and stored in the

buckets. The search can be stopped once the queue's minimum key is larger than the largest current tentative distance of any target vertex, since no better path will ever be found.

If a new target vertex gets added to T, simply generate bucket entries for that target vertex as described before. If a target vertex gets removed from T, run a reverse dijkstra query along the reverse graph G_R again. For every settled vertex, scan the bucket and remove any entry that matches the removed target vertex.

Many-to-one queries that compute the distances from a set of source vertices $S = \{s_1, s_2, \ldots\}$ to a single target vertex t are possible with virtually the same algorithm: This time, buckets are generated not along the reverse graph, but the forward graph. To then query the shortest distances, simply run a Dijkstra query on the reverse graph with source t and scan the bucket entries of the forward graph.

3.5 LOUD

LOUD, presented in [3], is a dispatching algorithm that finds the optimal insertion (with regards to the cost function described in section 2.1) $i = (i_v, i_p, i_d)$ for any given ridesharing request $r = (r_t, r_p, r_d)$ in only a few milliseconds.

LOUD uses BCH-Queries to determine distances required for calculating the cost of an insertion (see below). To that end, bucket entries are generated along both the upward and downward graph of the contraction hierarchy for each stop of each vehicle. In addition to the vertex- and distance-data, each bucket entry contains the vehicle id as well as the stop index for the stop that the entry was generated for. LOUD uses a technique called *elliptic pruning* which works as follows:

Note that a maximum allowed detour results out of the minimum departure time and the maximum arrival time of each stop and its successor. This maximum detour is called *leeway*. A detour over a vertex that is too far away from those two stops would violate one of the hard-constraints in the insertion cost function immediately. Therefore bucket entries for a stop are only generated on vertices where a detour via that vertex does not necessarily violate hard-constraints. This set of viable vertices forms an ellipsis around the two stops in question, hence the name. When generating bucket entries along the up-graph of the CH from a stop s, the leeway is calculated from the departure time of s's predecessor and the maximum arrival time of s instead. The leeway value of each stop is stored in each of its bucket entries.

Elliptic pruning greatly reduces the number of bucket entries in each bucket and strongly contributes to the good performance of LOUD. Note that since the last stop in a route has no successor, no leeway can be calculated for the buckets along the down-graph. Therefore, no buckets are generated and the required distances are calculated with a Dijkstra query instead (see below). The same is true for the first stop in each route and the buckets generated along the up-graph.

We are now ready to present how LOUD resolves a single ridesharing request: To begin

with, LOUD runs four BCH-Queries: One upward and one downward BCH-query running from the pickup and the dropoff stop. For each scanned bucket entry, if the distance to the currently settled vertex exceeds the leeway value stored in the bucket entry, the bucket entry gets ignored, since the resulting insertion would immediately violate some hard-constraint. The BCH-Queries each keep a list of all the vehicle-ids that were seen in some scanned and not ignored bucket entry. All vehicles that are not in the intersection of these four sets are guranteed to not be able to serve the ridesharing request without violating some hard-constraint somewhere.

Now, LOUD begins to simply iterate over all left-over feasible insertions, calculates the cost for each of them, and remembers the insertion with minimum cost. Note that the cost of an insertion can be calculated in $\mathcal{O}(1)$ if the following four distances are given:

- Distance from stop before pickup to pickup (distance to pickup)
- Distance from pickup to stop after pickup (distance from pickup)
- Distance from stop before dropoff to dropoff (distance to dropoff)
- Distance from dropoff to stop after dropoff (distance from dropoff)

Each of these distances were already calulated from the BCH-queries that were executed earlier. In the special case that the inserted dropoff would come right after the pickup, the distance from pickup is set to 0 and the distance to dropoff is set to the direct distance between pickup and dropoff, which has already been calculated in order to determine the maximum trip time of the request.

With these distances, arrival times at pickup and dropoff as well as the stops after them can be calculated in constant time, which is enough information to determine total detour, wait- and trip-time and therefore the cost of the insertion.

All possible insertions can now be categorized as follows:

(i) Insertions where the pickup (and dropoff) is inserted after the end of the route

(ii) Insertions where the pickup (but not necessarily the dropoff) is inserted "somewhere in the middle"

Insertions of type 2 are called *ordinary* insertions and are iterated over by LOUD first, but skipping over all ordinary insertions where the dropoff would be inserted at the end of the route. For those insertions, the four previously mentioned distances can be determined using the results of the BCH-Queries alone. Note that if the pickup stop index of the insertion is 0, the vehicle is diverted while driving, therefore the vehicle route gets updated first by calculating the path between the vehicle's first and second stop and following it until the time of request is reached. After this update, the first stop of the vehicle then points to the exact vehicle location. LOUD optimizes performance here by first calculating the insertion cost with the old vehicle location first, which is a lower bound to the real cost. If that lower bound exceeds the cost of an insertion that was already found, the vehicle location does not get updated, saving a CH-Query.

If the dropoff insertion index points to the last stop in the vehicles' route, the distance to the dropoff cannot be determined by the BCH-Queries, since no bucket entries for that last

stop exist along the downward-graph. To resolve this, after all other ordinary insertions of all vehicles have been tested, a Dijkstra query is run from the dropoff location along the reverse graph. This Dijkstra query stops when the distance to the dropoff (aka the minimum queue key) alone gurantees that any resulting insertion would have a cost higher than the best currently found insertion cost. This keeps the Dijkstra query limited to only a small section of the graph, minimizing the impact on performance. All distances from the last stops of all vehicles reached by the Dijkstra query to the dropoff are now determined and the final few ordinary insertions get iterated over.

Next, LOUD checks all insertions where the pickup is inserted after the last stop of a route. The idea is the same as before: Run a Dijkstra query, limited by the resulting minimum insertion cost, and figure out all distances to the pickup from that Dijkstra query. Since the dropoff has to come right after the pickup and also marks the end of the route, all other distances are trivial: The distance from the pickup is 0, distance to dropoff is the direct distance and distance from dropoff is 0 as well.

At this point, LOUD has found the minimum cost insertion for the request. The insertion is performed, adding stops and propagating the delay as well as the new maximum arrival times caused by the hard-constraints of the new passenger up and down the route. Finally, bucket entries are generated for the inserted stops.

4 Ridesharing with Transfers

We present and evaluate two ways to expand the LOUD-Algorithm for transfer stops, i.e. a request might be resolved using not just one, but multiple vehicles. This has the potential to decrease the detour required for resolving a request while still staying inside the bounds of the soft-constraints for the wait- and trip-time of the cost function, leading to a decrease in total insertion cost.

4.1 Changes to the ridesharing problem

First we adapt the mathematical model of the ridesharing problem to be able to account for insertions with transfer stops. To begin with, a request is now a 4-tuple $r = (r_t^{req}, r_t^{dep}, r_p, r_d)$, with r_p and r_d being as before, r_t^{req} is the time at which the request was made, and r_t^{dep} being the earliest possible departure time of the passenger at the pickup. We will see why this change is necessary below.
Let $r = (r_t^{req}, r_t^{dep}, r_p, r_d)$ be the original ridesharing request to be dispatched. The result of the dispatching algorithm is now not just a simple insertion, but a list of insertion-request-constraint-tuples $((i_1, r_1, c_1), (i_2, r_2, c_2), \ldots)$, with each insertion-request-constraint-tuple corresponding to one single-vehicle trip of the whole transfer trip. We call such a list of tuples a *solution*.

Each request contains information on the location of the pickup / dropoff / transfer-stops, e.g. for $r_1 = (r_{t1}^{req}, r_{t1}^{dep}, r_{p1}, r_{d1})$, r_{p1} is the pickup-location of the original request while r_{d1} is the location of the first transfer stop. Furthermore, the requests contain information of the departure time at those stops. The minimum departure time r_{t2}^{dep} of the second request r_2 is the arrival time of the passenger at the first transfer stop r_{d1} plus the constant minimum stop time t_{stop}.

Finally, the constraint information is set so that the maximum wait time at a stop is the total maximum wait time t_{wait}^{max} minus the wait time of the passenger at the pickup location, and the maximum trip time starting from one stop is the total maximum trip time t_{trip}^{max} minus the trip time already spent. Wait times on transfer stops count towards the trip time of the passenger, not the wait time. Note that the departure times of the requests $r_2, r_3, \ldots$ are now different from the time the request was made. The original time of request is still needed to determine the current location of a vehicle though.

Furthermore, it needs to be ensured that the passenger does not "miss" a transfer due to the

delay caused by an insertion that resolved a later request. There are two ways to achieve this:

- After inserting a transfer trip, set the maximum arrival time at a transfer stop to the arrival time of the transfer-pickup-vehicle at that stop. The hard-constraints now guarantee that the passenger will always arrive before the pickup-vehicle.
- For each stop s where a passenger is dropped off for a transfer, there is a corresponding stop t, belonging to another vehicles' route, that is responsible for picking up that passenger at the same location. Maintain a list of these corresponding stops $(t_1, t_2, \ldots)$ for every stop s. After performing an insertion $i = (i_v, i_p, i_d)$ into a route, scan every stop s in the route of i_v. For each stop t_i in the list of corresponding stops of s, check if the delayed arrival time at s causes the passenger to miss their connection at t_i. If that is the case, delay the departure time at t_i and propagate that delay through the entire route of the vehicle corresponding to t_i. Do this recursively for all affected vehicle routes until each delay has been resolved.

We choose the first method for our implementation for two reasons: First, if an insertion can affect more than one vehicle route, that significantly complicates the cost of calculating an insertion, which is an operation that is performed very often. The cost of calculating an insertion would still technically be in $\mathcal{O}(1)$, but with a much higher constant factor. Second, the first method guarantees that transfer stops are never delayed (except for traffic or other complications, which we don't simulate), which improves the ridesharing experience for the passenger.

Finally, because the minimum time of departure can now come after the time of request, stop times are now no longer constant. For example, a vehicle might drive and arrive at a stop location, but still has to wait for the passenger since the stop is a transfer stop and the other vehicle hasn't arrived yet. Because of that, we need to store the arrival time t_{arr}^{min} for each stop in addition to the already-stored departure time t_{dep}^{min}. The stop time t_{stop} is now a lower bound of the stop time: A stop requires *at least* t_{stop} amount of time, but might take longer if the vehicle has to wait. Note that our changes essentially added prebooking to the ridesharing scenario.

4.2 Transfer-At-Stop Algorithm

Our first algorithm finds only a specific type of single-transfer-routes: The first vehicle of the route picks up the passenger at the pickup-location and drops them off at an already-existing stop, which serves as a transfer stop. The second vehicle then picks up the passenger from that stop and drives the passenger to the dropoff point. The first vehicle needs to have at least two stops in its route, since otherwise there would be no eligible transfer stops for the passenger. The algorithm is as follows:

(i) Run BCH-Queries from the pickup vertex in forward- and reverse-direction to determine the set of eligible vehicles like in LOUD.

(ii) For each eligible vehicle v, iterate over all viable first-trip-insertions $i_1 = (i_v, i_p, i_d)$. The i_d-th stop of vehicle v will be the passengers' transfer stop. Build a request tuple $r1 = (r_{t1}^{req}, r_{t1}^{dep}, r_{p1}, r_{d1})$ that matches this first trip: r_{t1}^{req}, r_{t1}^{dep} and r_{p1} are as in the original request. r_{d1} is the location of the transfer stop. Calculate the arrival time at the transfer stop in $\mathcal{O}(\infty)$ just like when calculating an insertion cost. Save all found first-trip insertions in a list.

(iii) For each of these first trips, run a LOUD-Request $r2 = (r_{t2}^{req}, r_{t2}^{dep}, r_{p2}, r_{d2})$, where r_{p2} is the location of the transfer stop, r_{t2}^{dep} is the arrival time at the transfer stop and r_{d2}, r_{t2}^{req} are the same as in the original request. The time constraints c_2 for this request are as described before: The wait/trip-time constraint is the wait/trip-time constraint of the original request minus the already spent wait/trip-time in the first trip. Finally, build time constraints c_1 for the first trip so that the transfer stops gets reached in time and the constraints from the original request are fulfilled.

(iv) Calculate the cost of the entire trip by summing the detours and calculating the passenger wait- and trip-time like in LOUD. As before, we can compute this combined cost in $\mathcal{O}(1)$.

(v) Remember the solution $((i_1, r_1, c_1), (i_2, r_2, c_2))$ with minimum cost.

Note that for the LOUD-Requests in step 3, we don't want the vehicle of the resulting insertion to be the same vehicle as the one from the first trip. In order to achieve that, we extend LOUD slightly to allow for blacklisting vehicles: For a LOUD-Request, you can now specify a set of blacklisted vehicles. When iterating over the possible insertions, LOUD skips all insertions whose vehicles are blacklisted. This gurantees that the resulting best insertion is served by a non-blacklisted vehicle. In theory you could also skip over all bucket entries that match a blacklisted vehicle during the BCH-Queries, but the performance difference is neglegible.

The set of eligible vehicles for the first trip is small in practice as seen in [3], and not many stops can serve as transfer stops. Therefore the performance of this algorithm is good, as we will see in the section 5.4. However, due to the fact that the transfer stop *has* to be at an existing stop, the algorithm might miss good transfer routes with transfer locations that are not already on existing stops. Therefore, we now present a second algorithm that can find a larger variety of transfer routes.

4.3 Sampled-Transfer Algorithm

In theory, to find the optimal single-transfer-route between the pickup and dropoff, simply iterate over all vertices in the graph (except the pickup- and dropoff-location) and run two LOUD-Requests to find the optimal route for each transfer stop. (the vehicle found by the first request is blacklisted for the second LOUD-Request). Since urban road networks usually consist of hundreds of thousands of vertices, and two LOUD-Requests certainly take more than one millisecond even on modern hardware, this approach is not feasible in

practice.

The approach of the sampled-transfer algorithm is simple: First, we sample a small set of "sensible" transfer stop vertices using some heuristic. Then we run the two LOUD-Requests described above for each of these sampled vertices, remembering only the solution with the smallest cost. Since the set of sampled vertices has to be very small for adequate performance, the challenge in this approach is finding a good heuristic for good transfer stops. In the following section, we show two possible ways to sample transfer vertices. Starting from now, let $N \in \mathbb{N}^+$ be the number of sampled transfer vertices.

Geometric Sampling:
For this approach, the latitude- and longitude-coordinates (which we simply call *coordinates* from now on) of each vertex need to be available. The idea of this approach is that transfer stops intuitively should be "somewhere in the middle" between the pickup and dropoff. First, sample N random LatLng-Coords along a 2D-Normal-Distribution with its mean μ being the mean of the coordinates of the pickup and dropoff, and the standard deviation σ being some value smaller than half of the straight-line distance between those two coordinates. Then, iterate over every vertex $v \in V$ of the graph and for every vertex, calculate the distance to all sampled coordinates. Maintain an array of vertices of size N, one for each sampled coordinate, that contains the vertex with the smallest distance to the corresponding coordinate. This process of matching the vertices to the coordinates takes time in $\mathcal{O}(N \cdot |V|)$, which seems restrictive. In practice, while there is a measurable impact by this linear sweep, the total runtime is dominated by the LOUD-Requests that come after the sampling of the vertices. However, to improve performance in this step, some spatial index like a quad-tree [13] or kd-tree [7] could be used to reduce the required time to $\mathcal{O}(N \cdot log(|V|)$.

There's a good chance that two sampled coordinates are very close to each other and match to the same vertex. This could be resolved in multiple ways:

- Remove all duplicates from the list of sampled vertices. This causes the list of sampled vertices to potentially be smaller than N, which is not desirable.
- When sampling a coordinate, check the distance to any other already sampled coordinate. If the distance is smaller than some lower bound, discard the sample and try again.
- Do not sample randomly, but rather using some quasi-random set of numbers like a *Halton Sequence* [9]. This assures that the uniformly randomly generated numbers that are then redistributed to align with a normal distribution do not form any "clusters" by themselves.

In our implementation, if either the pickup/dropoff-location or an already sampled location is sampled again, we redo the sample to avoid duplicates and malformed requests. For the standard deviation of the normal distribution, we choose a quarter of the straight-line distance from the pickup- to the dropoff-coordinate by default. In section 5.6, we vary this tuning parameter to evaluate its impact on dispatching quality and runtime performance.

Betweenness Sampling:
A concern with the previously presented sampling strategy is that geometric approaches usually don't work well on road networks. For example, the geometric distance between two vertices is usually a bad heuristic for the shortest distance between those vertices in the graph, whose weights are based on travel time. This is quite intuitive: Driving from one end of a city to another by driving through the city might take half an hour. At the same time, driving from one city to another using a highway might also only take half an hour, but the distance traveled is much higher.

The following sampling strategy follows the intuition that good transfer stops should optimally be on as many shortest-paths on the graph as possible, since the chance that a vehicle (which always follows the shortest path between two stops) passes one of these stops is very high. The number of times a vertex v is contained in a shortest path between any two vertices $s, t \in V$ is called the *betweenness* of v. The betweenness of v is at least $2 \cdot |V|$ since a vertex certainly lies on all shortest paths with itself as either the source or target.

We calculate the betweenness of each vertex in the graph as follows: We maintain an array of size $|V|$ of betweenness values, one entry for each vertex $s \in V$. The array is initialized with zeroes.

(i) Run a full Dijkstra-query with source vertex s.
(ii) Build the shortest-path-tree using the parent-array calculated by the Dijkstra-query.
(iii) Calculate the sizes of all subtrees of the shortest-path-tree
(iv) Add the size of the subtree rooted at v to the entry for v in the betweenness-array.

The resulting betweenness is correct since the size of the shortest-path-subtree rooted at v is exactly the number of shortest paths from s to some target that contain v. To calculate the subtree-sizes, maintain yet another array of size $|V|$. We call this array A in the following pseudocode. Then, run this recursive function with input s:

Algorithm 2 SUBTREESIZE

Input: $T = (V, E_T), p \in V$

```
1: A[n] ← 1
2: for (p, c) ∈ E_T do
3:     SUBTREESIZE(T, c)
4:     A[n] ← A[n] + A[c]
5: end for
```

E_T are the edges of the shortest-path tree that was calculated from the parent-array of the Dijkstra-query. The runtime of this algorithm is in $\mathcal{O}(n)$, since the function gets called exactly once per vertex and each call incurs only a constant time cost.

The total calculating runtime is in $\mathcal{O}(|V| \cdot ((|V|log|V| + |E|) + |V|)) = \mathcal{O}(|V|^2 log|V| + |V||E|)$. An alternative method of calculating the betweenness of each vertex efficiently is *Johnson's Algorithm* [10], which has the same asymptotic runtime. The calculation of

the betweenness is quite cost-intensive in practice, however it only needs to be done once per graph. Therefore you can simply save the betweenness of each vertex to disk. The calculation has to be redone every time the graph or its metric changes, e.g. because of traffic or construction sites.

Next, we sort the list of all vertices by betweenness, in descending order. We then iterate over the vertices in that order and create a list of *chosen* vertices in the following way:

For each vertex v:

(i) Run Dijkstra's algorithm starting from v and terminate after the minimum queue key exceeds d_{min}, where d_{min} is a tuning parameter that determines the minimum distance between each possible transfer location.

(ii) If the Dijkstra-query has not visited any vertex already in the list of chosen vertices, add v to the list of chosen vertices.

(iii) Otherwise, do nothing and continue with the next vertex.

In our implementation, we initially set d_{min} to five minutes. The resulting list of vertices is called the set of *eligible transfer vertices* T. The travel-time distance between each vertex in T is guranteed to be at least d_{min}. The vertices of T therefore are high-betweenness vertices that cover the entire graph, with an even distance between each of the vertices. In section 5.6, we evaluate the impact that different values of d_{min} make on the dispatching quality and runtime performance.

Calculating this list of chosen vertices takes a few seconds, but also only needs to be done once per graph, so the list of potential transfer spots can be saved to disk as well.

4.4 Multi-Transfer Dispatching

We now expand the presented single-transfer dispatching algorithms to resolve ridesharing requests with an arbitrary amount of transfers. From now on, M denotes the intended number of transfer stops in the route. We begin with the transfer-at-stop algorithm:

Multi-Transfer at Stop:

A simplistic representation of the multi-transfer-at-stop algorithm can be seen in algorithm 3. The algorithm searches for the set of viable first trips just like in the single-transfer version. Now, lets say we have found a single viable first trip from the pickup vertex p to some transfer vertex t_1. We now resolve the rest of the request from t_1 to d recursively by building a new ridesharing request and calling the multi-transfer-at-stop again, but with one less transfer stop requested. The maximum trip time for that request is the original maximum trip time minus the already spent trip time in the first-trip. If $M = 0$ for the recursive call, LOUD is executed to find the final trip from t_M to d. As before, LOUD is now configured to count any wait time as trip time. For each recursive call, the vehicle corresponding to the first-trip gets blacklisted and will not be used in LOUD and FINDALLFIRSTTRIPS.

Algorithm 3 An simplified pseudocode of the multi-transfer-at-stop algorithm. Data like the road network graph or the vehicle routes are globally available for this pseudocode.

Input: $M^*, N \in \mathbb{N}, p^*, d \in V$

```
1: procedure MULTI-XFERATSTOP(M, p, d)
2:     if M = 0 then
3:         return LOUD(p, d)
4:     end if
5:     T ← FINDALLFIRSTTRIPS(p)
6:     B ← MIN_⌈M√N⌉(T, t → KEY(t))
7:     for t ∈ B do
8:         t ← t join with MULTI-XFERATSTOP(M − 1, t, d)
9:     end for
10:    return MIN(B, t → COST(t))
11: end procedure
12: MULTI-XFERATSTOP(M*, p*, d*)                          ▷ Initial call
```

Every first-trip found in a call of MULTI-XFERATSTOP comes with a lower bound on the final insertion cost of the multi-trip route: The final vehicle detour is at least the detour made by the vehicle of the first-trip to drive to the pickup location. The final wait time is actually exactly the wait time of the passenger for first trip in the original call of MULTI-XFERATSTOP, since each wait-time at a transfer stop counts as trip-time instead. The final trip time is at least the trip time from the pickup location to the first transfer stop plus the direct distance from the transfer spot to the dropoff. We calculate this direct distance using a CH-query. We use this lower bound as a "cost" of the first trip, and only recurse on the best $\lceil \sqrt[M]{N} \rceil$ found first-trips When multiple first-trips for a pickup were found, we only recurse on the $\lceil \sqrt[M]{N} \rceil$ found first-trips, where $N \in \mathbb{N}$ is a tuning parameter that determines a lower bound on how many complete M-transfer-routes get evaluated by the algorithm, provided the algorithm finds enough first-trips. The route found by each recursion is appended to the corresponding first-trip, and the overall best found route is returned.

Sampling Algorithms:

We now expand the transfer vertex sampling algorithms for multi-transfer dispatching. Instead of sampling just one transfer stop, we now sample a list of transfer stops $(t_1, t_2, \ldots, t_M)$. Such a list is called *transfer vertex tuple*. We then call the process of resolving a request with a route that contains M transfer stops *M-Transfer Dispatching*. $N \in \mathbb{N}$ now is the number of M-tuples that are supposed to be sampled. The main difficulty of Sampled-M-Transfer Dispatching is the sampling of the transfer vertex tuples. After the tuples have been sampled, LOUD has to be run $M + 1$ times, from each vertex to the next and each time blacklisting all vehicles that were already previously used. This is analogous to sampled single-transfer dispatching.

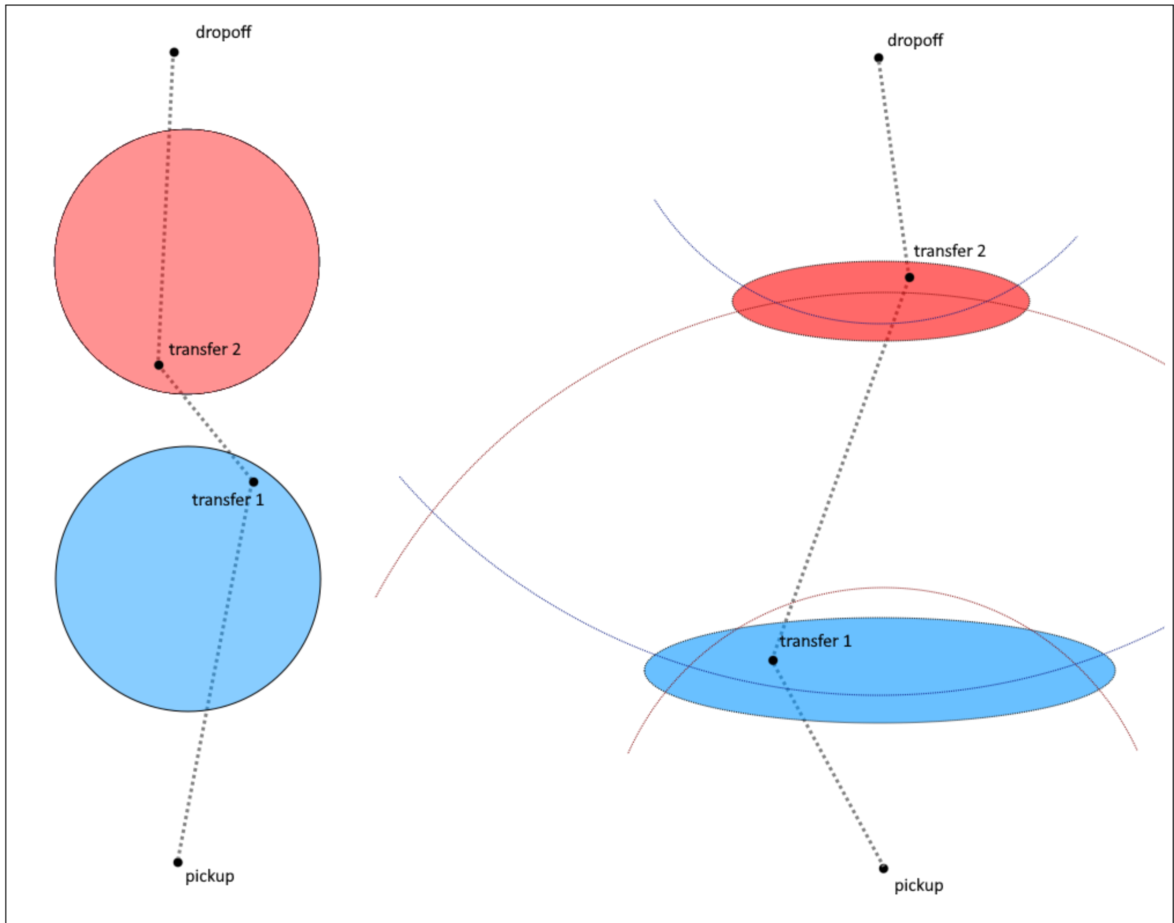


Figure 4.1: An example of a simple binomial geometric sampling and a slightly more intricate sampling along an ellipsis for 2-transfer dispatching. The circles represent the two-dimensional binomial distribution after which is sampled and roughly sketch out all points which lie within one standard deviation of the mean.

We now present how to modify both the geometric- and betweenness-sampling algorithm to find tuples of viable transfer stops.

Geometric:

In the single-transfer dispatching algorithm, the intuition for the sampling was to find vertices that are "somewhere in the middle of the pickup and dropoff". This intuition can seamlessly be extended to multi-transfer dispatching. For example, in 2-transfer dispatching, the first transfer stop should have a distance ratio from pickup and dropoff of roughly 1:2, and the second transfer stop a distance ratio of roughly 2:1, such that the resulting route is roughly divided into even thirds.

In fig. 4.1, on the left side, you can see an example of using a simple normal distribution

centered around the thirds of the direct line between pickup and dropoff, for use in 2-transfer sampling. You can see that in this example, the resulting transfer vertices do not evenly divide the route in thirds: The middle segment is much shorter. If you simply reduce the deviation of the normal distribution, it is easy to see that the sampling space would become very small, especially since you would need a smaller and smaller variance for higher M. This would lead to transfer vertices that are very close to each other, so not many truly different routes are found. To resolve this problem, we propose a new sampling approach: Instead of simply sampling along a two-dimensional normal distribution, which would result in a "circular" distribution of points around the mean, we try to sample points inside the ellipsis whose focal points are determined by the intersection of circles around the pickup- and dropoff-coordinates. This is better explained by example: In the case of 2-transfer-dispatching, like in fig. 4.1, to find the sampling space ellipsis focal points, draw a circle with a radius of a little more than the one-third of the straight-line distance from pickup to dropoff around the pickup, and another circle with a radius of a little more than two-thirds of the straight-line distance around the dropoff. Each point inbetween the two intersection points is roughly twice as far away from the dropoff as it is from the pickup (again, see fig. 4.1).

We now describe exactly how this sampling is done: From now on, we call the latitudinal component of a coordinate the *x-coordinate* and the longitudinal component the *y-coordinate*. For any point p, $x(p)$ is the x-coordinate of that point and $y(p)$ the y-coordinate. Let $dist_{geom}$ be the geometric straight-line distance between the pickup-coordinate p_p and dropoff-coordinate p_d. Let $\gamma > 1 \in \mathbb{R}$. We describe this algorithm for general M-transfer sampling. Let $r_1 = \gamma \cdot dist_{geom} \cdot \frac{i}{M+1}$ and $r_2 = \gamma \cdot dist_{geom} \cdot \frac{M-i}{M+1}$ be the radii of the circles, where $i \in \{1, \ldots, M\}$ is the index of the transfer stop vertex t_i to be sampled. t_1 is the transfer stop vertex closest to the pickup, t_M is the transfer stop vertex closest to the dropoff. The height h of the intersection area of the two circles described above is $h = r_1 + r_2 - dist_{geom} = dist_{geom} \cdot (\gamma - 1)$. The width w of this area is the height of the triangle spanned by the pickup-coordinate, the dropoff-coordinate and any intersection point. A sketch of all relevant line lengths is given in fig. 4.2.

To calculate w, we use *Heron's formula* [1] to calculate the area of the triangle with side-lengths r_1, r_2 and $dist_{geom}$ and divide by $\frac{dist_{geom}}{2}$ to get the height of that triangle, which is the width w we were looking for. The center point p_c of this area is on the straight-line between the pickup-coordinate and dropoff-coordinate with a distance of $\frac{i}{M+1} \cdot dist_{geom}$ from the pickup. Calculate the two-dimensional rotation matrix $M_R \in \mathbb{R}^{2\times 2}$ that rotates the coordinates so that the coordinate of the pickup and the coordinate of the dropoff have the same y-coordinate. Sample an x-coordinate using a 1D normal distribution with mean $x(M_R \cdot p_c)$ and deviation $\eta \cdot w$, and a y-coordinate in the same way with mean $y(M_R \cdot p_c)$ and deviation $\eta \cdot h$. Let this sampled point be called p'_i. Calculate the corresponding point p_i in the original coordinate system with $p_i = M_R^T \cdot p'_i$. The sampled point p_i will be sampled roughly around the intersection area of the two circles (depending on the chosen η, points may stray further from the intersection). For N requested transfer vertex tuples, sample

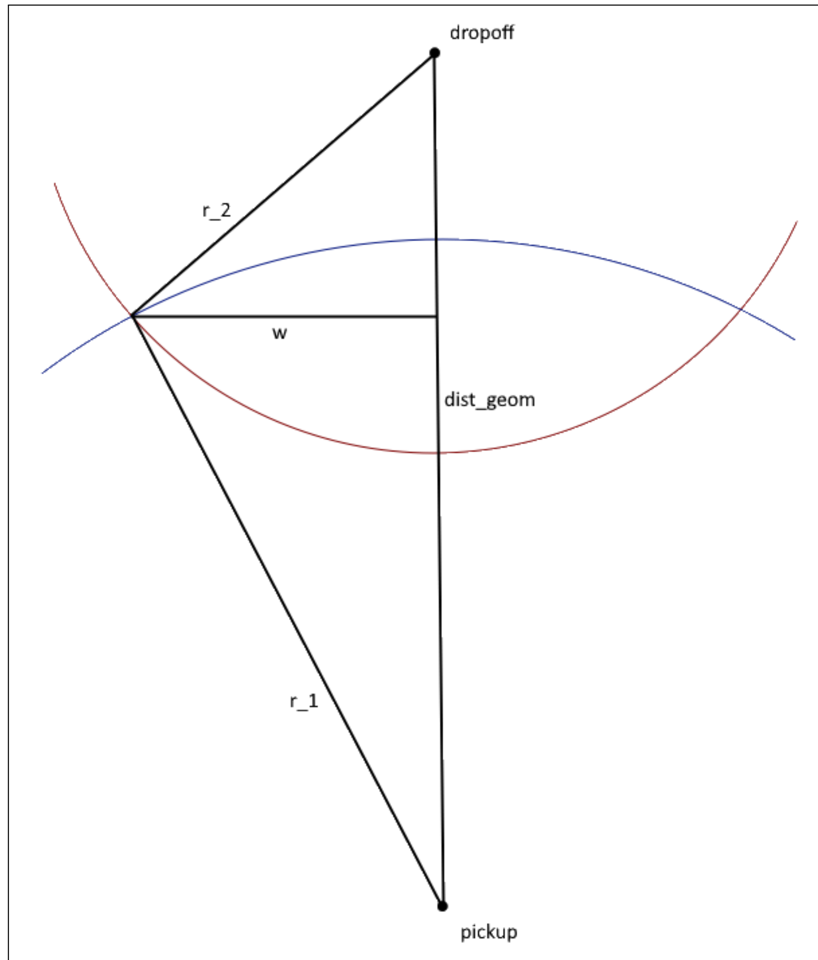


Figure 4.2: Geometric sketch for the calculation of the width of the area between the intersection points of the two circles around pickup and dropoff.

$\lceil \sqrt[M]{N} \rceil$ points for each $i \in \{1, \ldots, M\}$. Finally, match the sampled points to vertices in the graph and choose the transfer vertex combinations at random, avoiding duplicates.

Note that while this sampling algorithm seems more complicated, each additional step over the single-transfer coordinate sampling can be computed in $\mathcal{O}(1)$ and the rotation matrix M_R only has to be calculated once per request. Therefore the new sampling algorithm will most likely not have a strong impact on the performance.

Betweenness:

In the single-transfer betweenness-sampling algorithm, transfer-vertices were chosen so that the path from pickup p to dropoff d via the chosen transfer vertex is still short. We now expand that concept to M-transfer dispatching. The set of all eligible transfer vertices $T \subseteq V$ is computed just like in the single-transfer algorithm.

Algorithm 4 Pseudocode for the M-transfer dispatch betweenness-sampling algorithm.

Global: $G = (V, E), T \subseteq V, A \in (T^M \times \mathbb{N})^N$

```
 1: procedure BTWN(M, r, l_r, p, d)
 2:     SORT(T, t ∈ T → l_r + dist(p,t) + dist(t,d))
 3:     for t ∈ T do
 4:         if l_r + dist(p,t) + dist(t,d) ≥ A[N − 1] then
 5:             break
 6:         end if
 7:         APPEND(r, t)
 8:         l_r ← l_r + dist(t,d)
 9:         if M = 1 then
10:             SORTEDINSERT(A, (r, l_r))
11:         else
12:             BTWN(M − 1, r, l_r, t, d)
13:         end if
14:     end for
15: end procedure
16: BTWN(M, {−1, ..., −1}, 0, p, d                              ▷ Initial call
```

Maintain an array A of size N which contains the best found transfer vertex tuples as well as the distances of the route $p \to t_1 \to \ldots \to t_M \to d$ for each tuple. Initially, this array is filled with invalid tuples and all distances are set to ∞. Now, sort all eligible transfer vertices T by their distance to the pickup in ascending order. Iterate over each transfer vertex t in that order and recursively find the best $M - 1$ transfer stops for a trip from t to the dropoff. For each recursion, the current route gets passed as a function parameter. When the recursion gets down to $M = 1$, complete routes from p to d are found. Perform a sorted insertion of any found route into A if the found route is shorter than any route contained in A. Keep only the N shortest routes in A. Every time the recursive function is called, the function can return immediately if the distance of the current route plus the direct distance to the dropoff exceeds the length of the longest path in A. After the initial function call is done, A is filled with the N shortest routes from p to d via M transfer stops. Pseudocode for this algorithm can be seen in algorithm 4. It remains to be seen whether the performance of this algorithm is adequate. On one hand, the break condition should apply fairly quickly, since the best paths are found very quickly, thanks to the vertex ordering. On the other hand, especially for larger M, a fair amount of recursive calls, and therefore sorting needs to be done. Performance can be improved by customizing the sort function, for example using a custom quicksort: In quicksort, if the break condition seen in line 19 would apply to the pivot, the partition with keys larger than the pivot key does not need to be sorted. This reduces the amount of vertices to be sorted significantly. Furthermore there might be a problem with the variety of routes found, again especially for larger M: It is

possible that all N different found routes have the same $M - 1$ first transfer stops and the only difference in the routes is in the last transfer stop of the tuples. Therefore we present a second, much simpler algorithm for sampling M-transfer routes from the betweenness-sampled transfer vertices:

First, sample $\lceil \sqrt[M]{N} \rceil$ transfer stops for each i-th component ($i \in \{1, \ldots, M\}$) in a transfer vertex tuple by sorting all transfer stops by the key $dist(p, t) \cdot (M - i + 1) + dist(t, d) \cdot i$ and choosing the best ones. Sorting by this key causes the sampled transfer stops to be closer or futher away from the pickup/dropoff, depending on component. For example, when $M = 3$, for the key of the transfer stops sampled for the first component weighs the distance to the pickup three times more than the distance to the dropoff, causing transfer stops to be chosen rather close to the pickup. After a set of potential transfer vertices has been chosen for each component, choose combinations of these vertices at random to get a list of potential routes.

5 Evaluation

5.1 Setup

We extended the codebase written by Buchhold et. al for [3] with our presented algorithms. As in [3], all code was written in C++17, compiled using the GNU C++ Compiler with optimization level `-O3` and makes extensive use of modern C++ features to ensure optimal performance. Two computers were available for our experiments:

- `PC1`: 1x Intel i7 1170 processor, maximum clock @ 4.9GHz, 64GB DDR4-RAM, 16MB Cache.
- `PC2`: 2x 16-Core Intel Xeon E5-2683 v4 processors, clocked at 2.1 GHz, 512GiB of DDR4-RAM, 40MB Cache.

`PC1` is over twice as fast as `PC2` for our purposes. Unless stated otherwise, experiments are run on `PC2`. All simulations run only use a single CPU core, since none of the presented algorithms are parallel. All used default parameters can be seen in table 5.2.

The road network, vehicle fleet and list of requests is the same as in [3]: A road network of Berlin, which was extracted from *OpenStreetMap* and reduced to only the data required for routing, is used as the underlying graph. The vehicle and request data was generated using the transport simulation MATSim. Two sets of vehicle- and request-lists were generated: One for only one percent of the adult population of Berlin and Brandenburg, called the one-percent-scenario (abbreviated *1pct*), and one for ten percent of the population (ten-percent-scenario, abbreviated *10pct*). The 1pct-scenario contains 1000 Vehicles and 16569 Requests, whereas the 10pct-Scanario contains 10000 Vehicles and 149185 Requests. In section 5.10, we evaluate the presented algorithms on a similar, but larger scenario where the underlying road network represents the west German *Ruhr area*. As before, a one- and ten-percent scenario is available. We call these scenarios *ruhr-1pct* and *ruhr-10pct*. ruhr-1pct contains about 50000 requests and exactly 3000 vehicles, while ruhr-10pct contains just under 450000 requests and a fleet of 30000 vehicles. You can see a summary of these numbers in table 5.1.

The simulation runs over the course of one entire day, from early morning until late evening. The requests are realistically distributed in time and space, i.e. few requests are made in the early morning and most requests are made in the middle of the day. All vehicles serve requests the entire day and have a seating capacity of 4. Each request only contains a single passenger.

	$\|V\|$	$\|E\|$	#veh	#req
(berlin-)1pct	80922	169935	1000	16569
(berlin-)10pct	80922	169935	10000	149185
ruhr-1pct	420700	887790	3000	49708
ruhr-10pct	420700	887790	30000	447556

Table 5.1: Summary of the four different scenarios that we used as test data. The Berlin-scenarios are used for all evaluations except the ones in section 5.10.

variable	value	explanation
α	1.7	Parameter for t_{trip}^{max}
β	$2m$	Parameter for t_{trip}^{max}
t_{wait}^{max}	$5m$	Maximum passenger wait time for request constraints
t_{stop}	$1m$	Minimum stop time for picking up or dropping off passenger
N	8	Number of samples drawn for single/multi-transfer algorithms
d_{min}	$5m$	Min. distance between transfer vertices in betweenness-algorithms.
$\eta^{(1)}$	0.5	Standard deviation factor for single-transfer geometric algorithm.
$\eta^{(2)}$	0.25	Standard deviation factor for multi-transfer geometric algorithm.

Table 5.2: Default parameters used in evaluation. These are used unless stated otherwise.

5.2 Analysis of No-Transfer-Routes

To gain a better understanding of the ridesharing scenario, and to put the upcoming experimental results into context, we first present some stats about how unmodified LOUD resolves ridesharing-requests.

In table 5.3, basic data regarding the quality of the routes found by LOUD is presented. On average, a passenger that makes a ridesharing request embarks on a trip that takes a little over fifteen minutes in both scenarios. In the 1pct-scenario, the passenger has to wait about four minutes on average until the vehicle arrives, while in the 10pct-scenario the passenger only has to wait a little over two minutes for the vehicle. This is most likely because the increased amount of vehicles provide a better coverage of the road network, therefore the chance of a car being close to a pickup spot is higher. Two interesting properties about the ridesharing scenario emerge from this data: First, on average, both the constraint for the passenger wait-time and the passenger trip-time gets undershot significantly: The average passenger in the 1pct-scenario could wait for over a minute more and have a five minute longer trip without any increase in the cost of the corresponding insertion. In the 10pct-scenario, this is even more drastic, with the wait-time-constraint being undershot by over two minutes and the trip-time-constraint again by over five minutes. From this we can expect that eventual transfer-routes will probably show an increase in passenger wait- and trip-time (which will not increase the cost of the insertion, since the soft-constraints remain

Scenario	t_{detour}	t_{wait}	t_{trip}	t_{wait}^{max}	t_{trip}^{max}
1pct	967s	233s	1070s	300s	1374s
10pct	758s	141s	948s	300s	1276s

Table 5.3: Average actual and maximum wait- and trip-time for resolving a ridesharing-request using unmodified LOUD, as well as the detour (i.e. increase in vehicle operation time) caused by serving a new request.

fulfilled) but improve the detour of the individual vehicles, ultimately reducing the cost of the insertion.

Second, note that the average required detour for serving a request is very close to the total trip time for that request, suggesting that when a request is dispatched, the vehicle spends the majority of time driving passengers exclusively to their destination, with no actual ridesharing taking place. We take a closer look on just how close the current ridesharing scenario is to simple taxi calls:

Scenario	#pickup at end	#dropoff at end	#ordinary	avg. route len
1pct	80.23%	90.97%	9.03%	1.33
10pct	65.55%	85.89%	14.11%	1.56

Table 5.4: Data showing how often a certain type of insertion is performed by LOUD relative to the total number of requests, as well as the average route length of the vehicle before the insertion is performed.

In table 5.4 we show how often LOUD performs a certain type of insertion: A "pickup at end"-insertion is an insertion where the new passenger gets picked up after the vehicle has completed its current route. A "dropoff-at-end"-insertion is an insertion where the passenger gets driven to their destination after the vehicle has completed all other stops in its route. An ordinary insertion is an insertion where both the pickup and dropoff are inserted somewhere in the middle of the route, i.e. where the insertion is neither a "pickup-at-end"-insertion nor a "dropoff-at-end"-insertion. We also display the length of the vehicle route corresponding to the best found insertion *before* the insertion is performed. Note that each "pickup-at-end"-insertion" is automatically a "dropoff-at-end"-insertion as well, and therefore an insertion is either "dropoff-at-end" or ordinary, making the two percentages add up to 100%.

From the data in table 5.4, we can see that ordinary insertions are not all that ordinary: For the 1pct-scenario, over 80% of all insertions are appended to the end of the route, in the 10pct-scenario, this number is still over 65%. Furthermore the average route length of a vehicle before insertion is quite small at only 1.33 for the 1pct-scenario and 1.56 for the 10pct-scenario.

This means that for the majority of requests, the vehicle that gets matched to the passenger has already completed all previous stops and now sits idle somewhere on the road network.

When the request gets dispatched, that vehicle drives to the pickup location immediately, picks up the passenger and drives them directly to their destination, operating like a taxi. On the 10pct-scenario, more "real" ridesharing is performed than on the 1pct-scenario. This is most likely because the increased amount of vehicles driving along the road network increase the chance of finding a vehicle that is already passing close to the requested pickup location. We expect transfer dispatching to work better on the 10pct-scenario for the same reason: More vehicles on the road network means a higher likelyhood that a vehicle can serve a transfer stop without having to make a big detour.

For the passenger, this taxi-like type of insertion is ideal: The wait time is short, as shown in table 5.3 and the trip time is optimal since the direct path is taken. Detour-wise though these types of insertions are not optimal: The entire trip of the passenger as well as the trip to the pickup gets added to the vehicle operation time. For a better detour it would be more desireable to utilize vehicles that are already carrying some passengers and are driving roughly in the direction the new passenger wants.

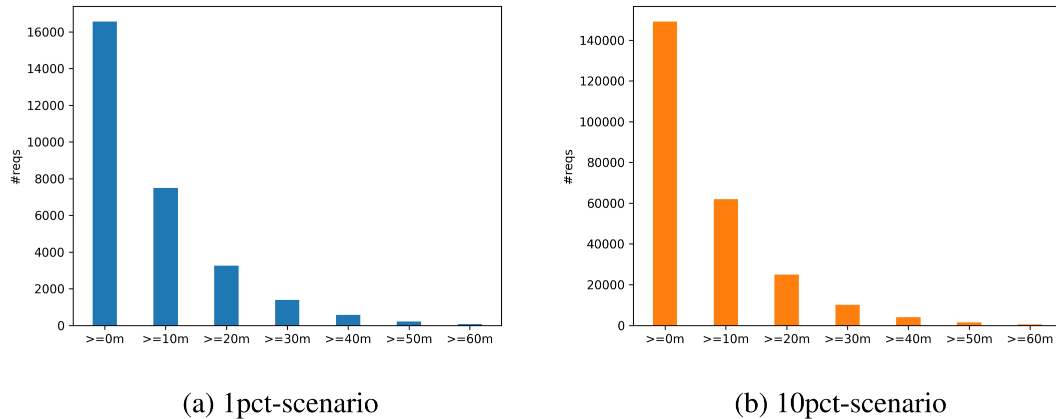
This also sheds some light on the very good performance of LOUD: As we've seen in table 5.3, most vehicle routes contain very few stops on average. Few vehicle stops lead to few bucket entries in each of the buckets, leading to fast BCH-queries, and furthermore the search space of all viable insertions is kept small, with only one possible insertion (the pickup-at-end insertion) existing for most vehicles.

This data has some consequences for transfer routes: Transfer routes will probably improve the vehicle utilization by reducing the detour required to resolve a single ridesharing request. This is an advantage for the ridesharing provider, since fewer vehicles are necessary to handle the request volume, saving money. For passengers however, the introduction of transfer routes will very likely be a downgrade: In addition to the stress caused by transfer stops, passenger wait- and trip-times will probably increase significantly. Furthermore the amount of taxi-like "pickup-at-end"-insertions, which are very comfortable for the passengers since no ridesharing actually takes place, will probably be replaced by more detour-optimal transfer routes.

To avoid this downgrade in ride quality, the cost function for the ridesharing problem could be modified. For example, a penalty for transfer stops could be introduced to take the stress of transfers into account, or a cost for wait- and trip-times could be introduced even when those values are under their respective constraints. We do not modify the cost function in any way, so that we can maintain comparability with [3]. Note though that the cost function can easily be modified in any arbitrary way, as long as it still contains the hard-constraints for wait- and trip-time, since those are required for elliptic pruning.

5.3 Analysis of the test scenarios

We look a little further into the scenarios that we are running on. As stated before, our underlying graph is an accurate representation of Berlin and its surrounding area. On this



(a) 1pct-scenario

(b) 10pct-scenario

Figure 5.1: Distribution of direct distances between pickup and dropoff for each ridesharing request on the 1pct- and 10pct-scenario. Each bar portrays how many requests have a direct distance from pickup to dropoff that are at least as long as the time given on the corresponding x-tick.

graph, the longest distances from one vertex to another are a little over an hour long. An example, routed by Google Maps, can be seen in fig. 5.2. Note that although the path through Berlin is much shorter geometrically speaking, the route around Berlin is actually the shortest path from source to destination. In the ridesharing scenario, this would mean that if this request were resolved using a taxi-like trip, the vehicle would drive around Berlin and therefore wouldn't be able to efficiently serve additional requests that are made inside Berlin.

We evaluated the direct distances of each request made by the transport simulation. The results can be seen in fig. 5.1. As you can see, half of all requests on both scenarios have a direct distance of less than ten minutes between pickup and dropoff. Only about 10% of all requests have a minimum driving time of over thirty minutes.

For ridesharing with transfer stops, this poses a problem: In practice, transfer stops for rides that are shorter than ten minutes are undesirable from a passengers' perspective. Furthermore, every extra transfer stop incurs more detour and trip time, as more vehicles are affected by the request and the wait time of the passenger at that transfer counts into the total trip time. Based on this data we predict that single-transfer dispatching might improve some of the longer distance queries, especially requests with a direct distance of >20m, since one transfer stop for ten minutes of travel time in a vehicle seems reasonable from an intuitive standpoint. We however also predict that multi-transfer dispatching, particularly with higher values of M, will most likely not improve very many requests, since the travel time between pickup and dropoff is simply too short to justify multiple transfer stops.

Finally, in fig. 5.3 you can see the distribution of the requests over the course of a day. You can see that most requests are, unsurprisingly, made during daytime. The graph shows

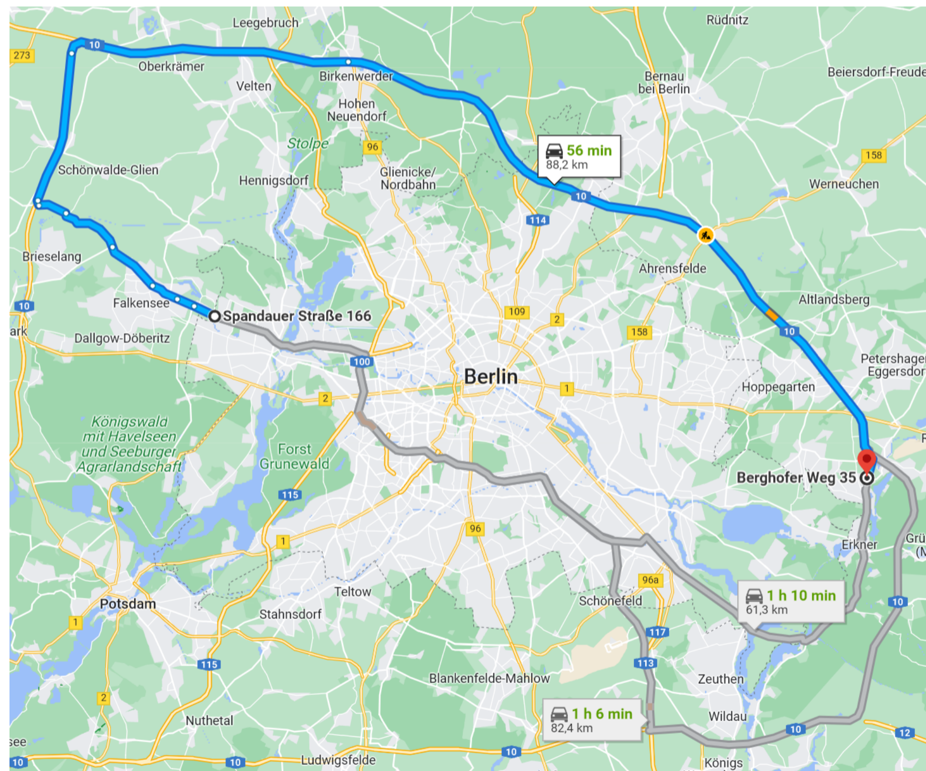
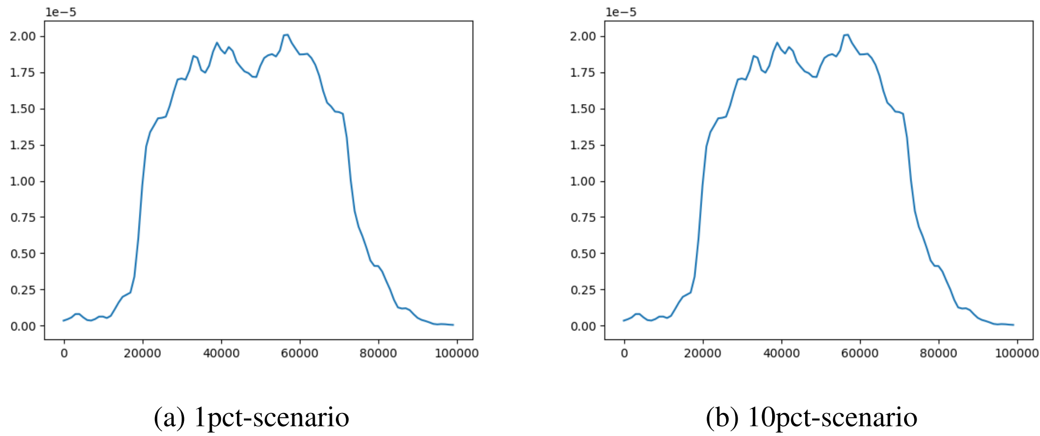


Figure 5.2: An example of what could be considered a "long distance" route through our road network, represented on Google Maps. Note that the direct route through Berlin is over ten minutes longer than the route around Berlin that uses the faster federal highways.

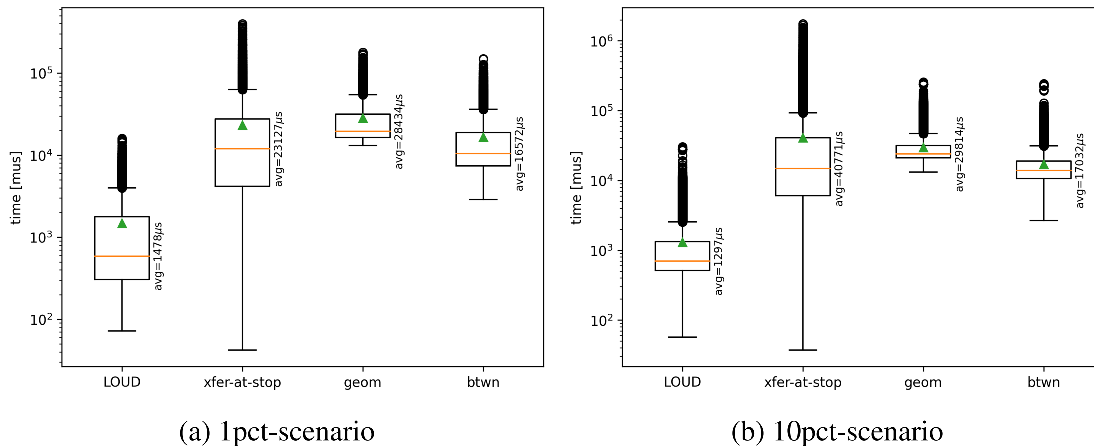
a steep increase in request density starting at around 7:00AM and a sharp dropoff again at roughly 7PM. We expect transfer dispatching to be most effective in this high-transfer-volume time period, as the chance of finding a viable second vehicle to make a transfer to is highest.



(a) 1pct-scenario (b) 10pct-scenario

Figure 5.3: Distribution of ridesharing requests on both scenarios across time

5.4 Runtime Performance of single-transfer dispatching



(a) 1pct-scenario (b) 10pct-scenario

Figure 5.4: Comparison of runtime for dispatching a single request between ordinary LOUD and each of the presented transfer dispatch algorithms on each scenario.

In fig. 5.4, we compare the required runtime to answer a single ridesharing request of each of the presented transfer dispatching algorithms to each other and also against the runtime of the original LOUD algorithm. The algorithms are evaluated on both the 1pct- and 10pct-scenario. Note the log scale on the y-axis. On our test computer, LOUD only requires a little over a millisecond on average to answer a ridesharing request, however the deviation is fairly high. LOUD takes slightly more time on the 10pct-scenario than the 1pct-scneario, since the longer vehicle routes lead to larger BCH-buckets and a larger search space.

The transfer-at-stop algorithm takes about 23ms on average, the geometric sampling algorithm 28ms and the betweenness-sampled algorithm 16ms. The deviation of the runtime of the transfer-at-stop algorithm is much higher than the runtime deviation of the sampling algorithms though. This is because while both sampling algorithms do exactly $2 \cdot N = 16$ LOUD-Requests each time, the transfer-at-stop algorithm performs one LOUD-Request per potential first trip that was found and not pruned. This number might be anywhere from 0 to a couple dozen requests. We evalute the number of LOUD-Requests made by the transfer-at-stop algorithm more extensively in section 5.6.

Furthermore, the runtime required by the transfer-at-stop algorithm is significantly higher on the 10pct-scenario than on the 1pct-scenario for the same reason: On the 10pct-scenario, vehicle routes are longer on average, resulting in more potential first trips. Therefore, while the transfer-at-stop algorithm is faster than the geometrically-sampled algorithm on the 1pct-scenario, this is no longer the case for the 10pct-scenario. The betweenness-sampled algortihm is the fastest on both scenarios.

Note that the difference in performance between the geometrically-sampled algorithm and the betweenness-sampled algorithm is caused by the linear search that maps sampled coordinates to edges in the graph, which takes far longer than the BCH-Queries that the betweenness-sampled algorithm has to execute. Since the runtime of both algorithms is dominated by the 16 performed LOUD-requests, the difference in runtime between the two algorithms is a constant summand.

In all the algorithms, the average required runtime is significantly higher than the median time, showing that the runtime is dominated by some requests where many different insertions are viable. This is especially noticeable for the transfer-at-stop algorithm, where the slowest requests take up to 500ms on the 1pct-scenario and even up to two seconds on the 10pct-scenario.

We can conclude that in regards to performance, the transfer-sampling algorithms are more viable for practical real-world use than the transfer-at-stop algorithm because the average runtime of the transfer-at-stop algorithm is going to increase further when switching to the full "100-percent"-scenario, while the runtime of the sampling algorithms remains almost the same. Even on the 10pct-scenario, the transfer-at-stop algorithm is the slowest of the three presented algorithms. Furthermore, the runtime of the sampling algorithms deviates much less, making performance more predictable.

5.5 Dispatch Quality

We now evaluate our main point of interest, the quality of the transfer routes found by the individual algorithms compared to the respective non-transfer route. We check how many of the approx. 16000 requests of the 1pct-scenario and the 150000 requests of the 10pct-scenario were improved by using a single-transfer-route. To that end, we run the transport simulation and answer each incoming request both with ordinary LOUD and with a single-

transfer dispatching algorithm. The insertion that has the lower cost gets inserted into the route. For every request where the single-transfer-route was better, we determine how the wait-time, trip-time and detour of it compares to the route that would have resulted from the insertion found by LOUD.

		xfer-at-stop	geometric sampled	betweenness sampled
1pct	hit-rate	7.65%	2.74%	8.87%
	Δ_{cost}	-4740	-3076	-3286
	Δ_{detour}	-219s	-236s	-316s
	Δ_{wait}	+1s	-98s	-74s
	Δ_{trip}	+203s	+483s	+435s
10pct	hit-rate	15.04%	6.94%	18.92%
	Δ_{cost}	-3177	-2285	-3148
	Δ_{detour}	-354s	-227s	-335s
	Δ_{wait}	+36s	-59s	-46s
	Δ_{trip}	+339s	+483s	+457s

Table 5.5: Comparison of dispatching quality between transfer algorithms and no-transfer routes on both scenarios. All Δ-values are averages and compare improving transfer-routes to the corresponding normal routes.

The results of our experiments can be seen in table 5.5. The hit-rate is the amount of times the single-transfer route was an improvement over the no-transfer-route, relative to the total number of requests. The wait time of a passenger is the duration between the time of request and the arrival of the first vehicle at the pickup location. The trip time is the difference between the arrival time of the passenger at the dropoff location and the departure time of the first vehicle from the pickup point. Note that all "wait times" at transfer stops count into the trip time, *not* into the wait time. The detour of a single-transfer-route is the sum of all the detours required by all affected vehicles.

For all transfer-dispatch-algorithms, the transfer route improves the detour on average by somewhere between three and five minutes, depending on the algorithm. Contrary to our expectations in section 5.2, while the trip time of a single-transfer trip increases significantly compared to the no-transfer trip, the wait time *decreases*. We assume that this is because for transfer-trips, more vehicles are viable for pickup since a pickup vehicle does not necessarily also have to pass close by the dropoff location to serve the request efficiently. Note that every vehicle that can serve the request efficiently with a no-transfer route is also a viable pickup vehicle for a single-transfer trip, since the wait time would be the same.

As we expected, transfer routes perform better on the 10pct-scenario than the 1pct-scenario: On the 10pct scenario, more than twice as many requests are resolved using transfer routes compared to the 1pct-scenario for all used algorithms. Each algorithm improves the cost

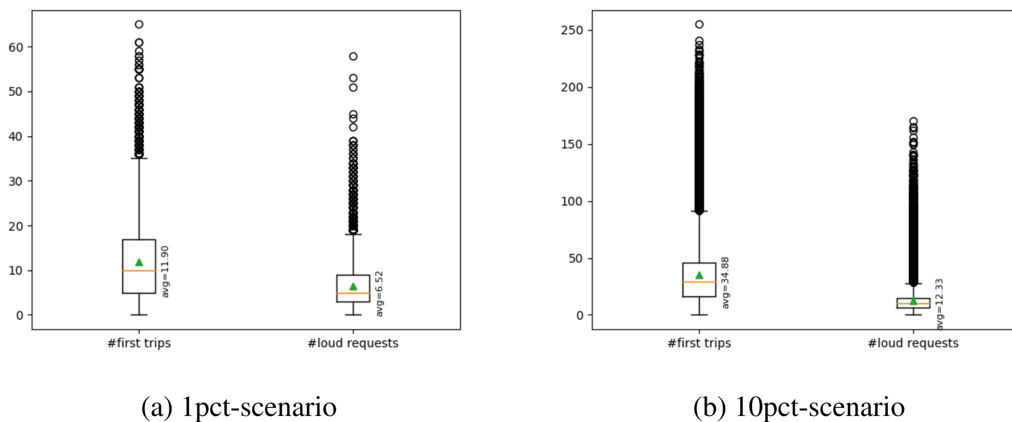
of the dispatching result by around 2000 to 4000 points. Note that each second of detour increases the cost by ten points. Each second of wait-time past the constraint increases the cost by ten points as well, while each second of excess trip-time increases the cost by 100 points. The greatest average improvement in insertion cost is made by the transfer-at-stop algorithm, which improves the dispatch result cost by almost 5000 points on the 1pct-scenario. The geometric sampling algorithm performs the worst in both scenarios, both in regard to hit-rate and to average insertion cost improvement.

Overall, the results seen in table 5.5 are promising. Even though insertion costs of no-transfer routes were already low because most requests were originally resolved using trip-time optimal "taxi-like" trips with very good wait times (as seen in section 5.2), two of the three presented algorithms manage to improve the cost of insertions for over 15% of all requests made on the 10pct-scenario. We expect this hit-rate to be even higher for a realistic 100pct-scenario.

5.6 Algorithm-Specific Evaluation

We now evaluate each of the presented transfer dispatch algorithms for interesting algorithm-specific data. We analyse the performance of the individual algorithms in detail, play with the various tuning parameters, etc.

Transfer-at-stop algorithm:



(a) 1pct-scenario (b) 10pct-scenario

Figure 5.5: Number of first trips found by the transfer-at-stop algorithm and number of actually performed LOUD-Requests on both scenarios.

We begin with the transfer-at-stop algorithm. The biggest point of interest for this algorithm is how many first trips are found, and how many of them are pruned. The number of non-

pruned first trips is the number of LOUD-requests made by the algorithm. We present the number of total and pruned first trips as boxplots in fig. 5.5. On the 1pct-scenario, about 12 first trips are found by the algorithm on average, with the median being slightly lower. Just under seven LOUD requests are executed on average, meaning that around five of the first trips are pruned on average for each request. To be more exact, 45% of all first-trips are pruned. We can also see that for some requests, up to 60 first trips are found, which then result in similarily many LOUD-requests.

When looking at the 10pct-scenario, the number of found first trips has increased drastically, with now almost 35 first trips being found on average, and more than 200 first trips being found on the largest outlier requests. This was to be expected though: In previous data we have already seen that since the road network is better covered with vehicles, more vehicles are able to pick up the passengers with low wait time and detour. Furthermore, as seen in table 5.3, vehicle routes tend to be longer on average on the 10pct-scenario, because the increased number of requests increases vehicle utilization. Since one of the conditions for finding a first trip is that a vehicle needs to have at least two stops, this leads to more viable first trips than in the 1pct-scenario, where many vehicles just sit idly on a completed stop. What we can also see though is that a larger proportion of first trips gets pruned: Out of the average 35 first trips, only 12 of them lead to a LOUD-Request, which evaluates to a pruning ratio of 64%. Comparing this data to the performance data in section 5.4, we can calculate that each LOUD-Request takes a bit over one millisecond, which is right in line with our previous findings.

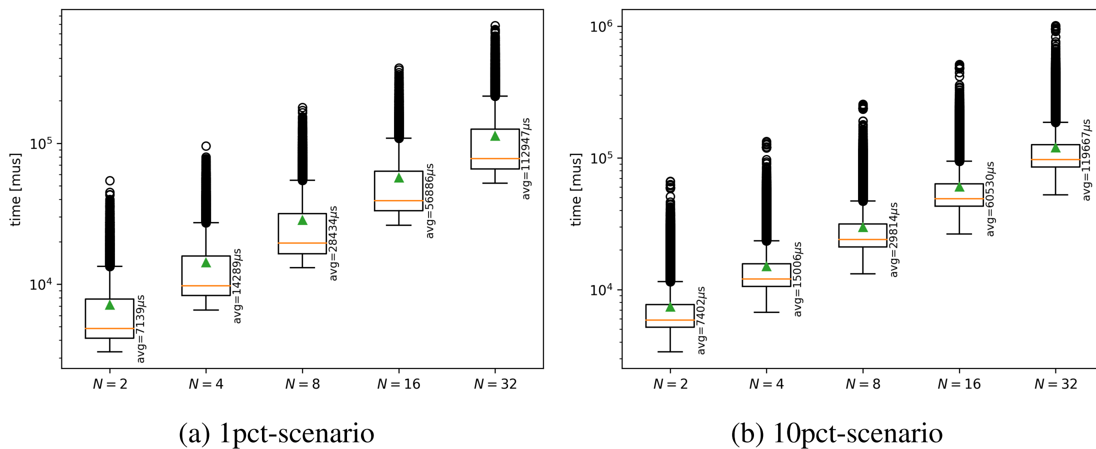
Overall, while the performance of the algorithm is adequate for the tested scenarios, we can assume that it doesn't scale well for scenarios with even more vehicles and higher request density, because many more first trips would be found. The pruning criterion works reasonably well, but its efficiency is dependent on some of the best first trips being present in the beginning of the list of all first trips, which is random. A possible optimization is to define some cost heuristic that estimates the total cost of the single-transfer trip based on only the first trip, and sort the list of first trips by that heuristic in ascending order.

We also evaluated how much the gathering of all first trips impacts the performance of the algorithm. However, the time it takes to create the list of all potential first trips consistently is in the single-digit microsecond range (excluding the BCH-Queries, but these need to be run anyway for LOUD) and therefore not worth a plot. The entire runtime of the algorithm is dominated by the LOUD-Requests.

Geometrically-sampled single-transfer dispatching:
We vary the tuning parameters N and σ for the geometrically-sampled single-transfer dispatch algorithm independently of each other and evaluate the impact on performance and dispatch quality on both scenarios.

		$N=2$	$N=4$	$N=8$	$N=16$	$N=32$
1pct	hit-rate	0.84%	1.65%	2.74%	4.07%	5.59%
	Δ_{cost}	-3220	-3157	-3076	-3086	-3077
	Δ_{detour}	-178s	-206s	-236s	-241s	-248s
	Δ_{wait}	-113s	-111s	-98s	-82s	-85s
	Δ_{trip}	437s	462s	483s	476s	479s
10pct	hit-rate	2.66%	4.34%	6.94%	10.06%	13.60%
	Δ_{cost}	-2187	-2291	-2285	-2356	-2373
	Δ_{detour}	-198s	-219s	-227s	-234s	-240s
	Δ_{wait}	-78s	-72s	-59s	-56s	-50s
	Δ_{trip}	+463s	+473s	+483s	+480s	+475s

Table 5.6: Quality values for the geometrically-sampled single-transfer dispatch algorithm for varying values of N (i.e. number of sampled transfer stops) on both scenarios.



(a) 1pct-scenario

(b) 10pct-scenario

Figure 5.6: Runtime required to resolve a single ridesharing request with the geometrically-sampled single-transfer dispatch algorithm for varying values of N on each scenario.

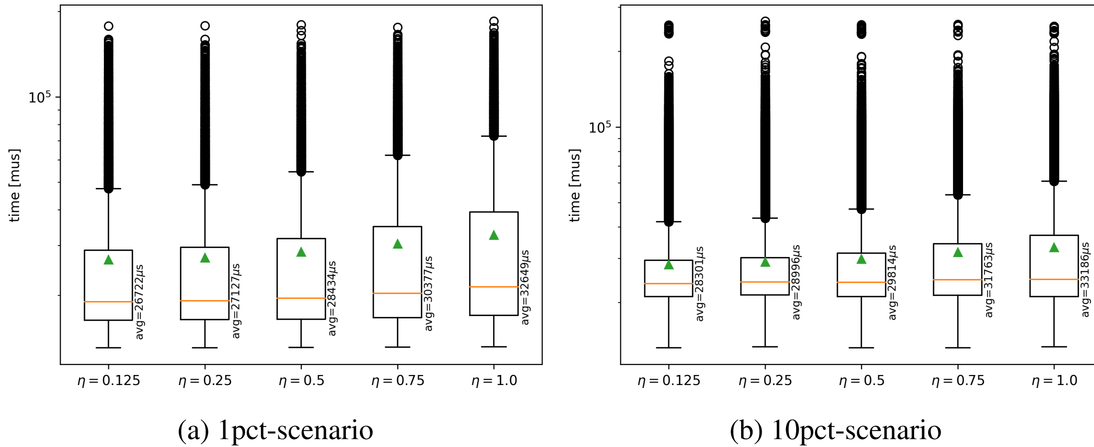
Our qualitative results are presented in table 5.6, while the runtime performance results are presented in fig. 5.6. The results are not surprising: A higher N results in a higher hit rate. For low N, doubling N results in a doubling of the transfer route hit rate. As N gets higher though, the curve flattens: Doubling N from 16 to 32 only shows a 37% increase in hit rate. This was to be expected, since most good routes inside the sampling radii will have been found when N is high enough. The runtime performance is also not surprising: When N is doubled, the required runtime is almost exactly doubled every single time since twice as many vertices have to be matched in the linear search and twice as many LOUD-Requests

have to be made.

Next, we vary the standard deviation σ in which the transfer coordinates are sampled. We set $\sigma = \eta \cdot \frac{dist_{geom}(p,d)}{2}$, where $dist_{geom}(p, d)$ is the geometric straight-line distance from the pickup to the dropoff, and $\eta \in (0, 1]$ is a tuning parameter. As stated earlier, our baseline value for η is $\frac{1}{2}$. We run all experiments with our earlier default value of $N = 8$.

		$\eta = 0.125$	$\eta = 0.25$	$\eta = 0.5$	$\eta = 0.75$	$\eta = 1.0$
1pct	hit-rate	2.40%	2.90%	2.74%	2.28%	1.68%
	Δ_{cost}	-3023	-2962	-3076	-2773	-2681
	Δ_{detour}	-222s	-226s	-236s	-197s	-177s
	Δ_{wait}	-88s	-83s	-98s	-119s	-129s
	Δ_{trip}	+428s	+440s	+483s	+445s	+445s
10pct	hit-rate	6.61%	7.37%	6.94%	5.60%	4.35%
	Δ_{cost}	-2289	-2259	-2285	-2288	-2243
	Δ_{detour}	-237s	-233s	-227s	-217s	-209s
	Δ_{wait}	-37s	-43s	-59s	-74s	-84s
	Δ_{trip}	+464s	+464s	+483s	+483s	+478s

Table 5.7: Quality values for the geometrically-sampled single-transfer dispatch algorithm for varying values of η (standard deviation factor) on both scenarios.



(a) 1pct-scenario

(b) 10pct-scenario

Figure 5.7: Runtime required to resolve a single ridesharing request on the 1pct-scenario with the geometrically-sampled single-transfer dispatch algorithm for varying values of η.

We present our results for the varying η in table 5.7 and fig. 5.7. When looking at the qualitative data, $\eta = 0.25$ seems to be a sweet spot, achieving the highest hit-rate on both scenarios. We already expected that a sweet spot has to exist: Choose η too small, and

the sampled points barely deviate, which causes good transfer spots that are further from the geometric center to be missed. Choose η too high and the transfer spots stray too far, causing long detours and trip times. The performance of the algorithm seems to be mostly unaffected by η. Larger values of η cause a very slight increase in runtime, most likely because the larger distances increase runtime of Dijkstra-, CH- and BCH-queries.

Comparing the hit rate of the geometric algorithm to the betweenness-sampled algorithm, the geometric algorithm seems to miss out on many good transfer spots that the betweenness-sampled algorithm finds. This shows that good transfer spots are not necessarily around the geometric center between the pickup- and dropoff-location. For high values of η, the sampled points move further away from the geometric center and the area in which vertices are sampled gets larger. However, since N does not increase with increasing values of η, the chance of finding some good transfer spot at all decreases, which can be seen in the decrease of the hit rate. As η gets larger and larger, the geometric sampling algorithm approaches a uniformly distributed random transfer vertex selection.

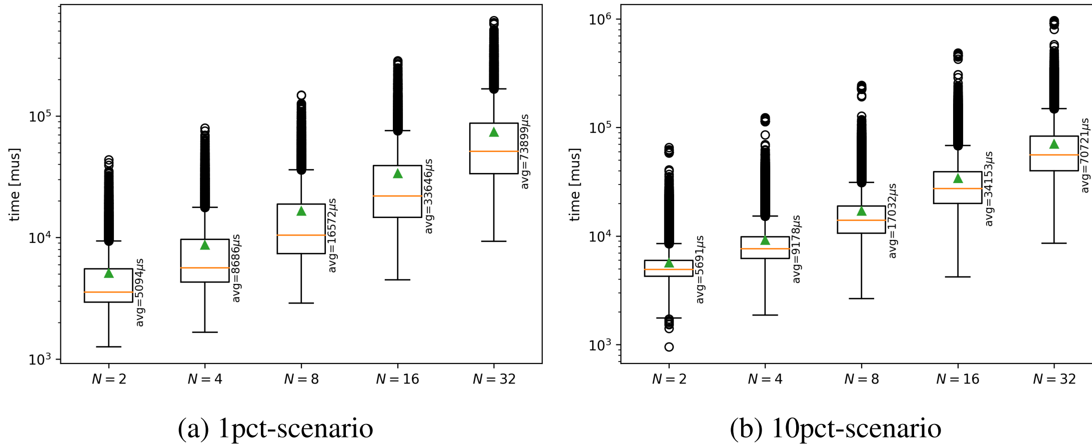
Wrapping up the single-transfer geometrically-sampled algorithm, we've shown that no combination of parameters matches the hit rate seen in the betweenness-sampled algorithm while still maintaining adequate runtime performance. Therefore we recommend the betweenness-sampled algorithm over the geometrically-sampled algorithm as long as the edge weights of the underlying graph do not follow geometric distances. The geometrically-sampled algorithm could be more interesting on road networks with similar vehicle speeds along every road, e.g. road networks of only inner cities with a constant speed limit. In those types of graphs, the geometric distance between two vertices is a better approximation of the actual travel time, since the weight of an edge is roughly the geometric distance between the vertices divided by that speed limit, assuming a mostly straight road, which is realistic in larger cities.

Betweenness-sampled single transfer dispatching:

We again evaluate the performance of the betweenness-sampled single-transfer dispatching algorithm for varying values of N. We furthermore vary the minimum required distance between potential transfer vertices d_{min}. Lower values of d_{min} lead to more potential transfer stops, increasing transfer vertex density, while higher values of d_{min} cause the potential transfer vertices to be more spread out. Therefore we expect a higher required runtime for lower values of d_{min}, since the BCH-buckets are larger.

		$N=2$	$N=4$	$N=8$	$N=16$	$N=32$
1pct	hit-rate	5.23%	7.26%	8.87%	9.80%	10.04%
	Δ_{cost}	-2930	-3110	-3286	-3429	-3474
	Δ_{detour}	-264s	-292s	-316s	-325s	-336s
	Δ_{wait}	-89s	-80s	-74s	-77s	-72s
	Δ_{trip}	+376s	+417s	+435s	+464s	+485s
10pct	hit-rate	13.74%	17.15%	18.92%	19.47%	19.66%
	Δ_{cost}	-2799	-2981	-3148	-3260	-3343
	Δ_{detour}	-297s	-321s	-335s	-345s	-349s
	Δ_{wait}	-48s	-45s	-46s	-46s	-46s
	Δ_{trip}	+418s	+441s	+457s	+480s	+487s

Table 5.8: Quality values for the betweenness-sampled single-transfer dispatch algorithm for varying values of N (i.e. number of sampled transfer stops) on both scenarios.



(a) 1pct-scenario

(b) 10pct-scenario

Figure 5.8: Runtime required to resolve a single ridesharing request with the betweenness-sampled single-transfer dispatch algorithm for varying values of N on each scenario.

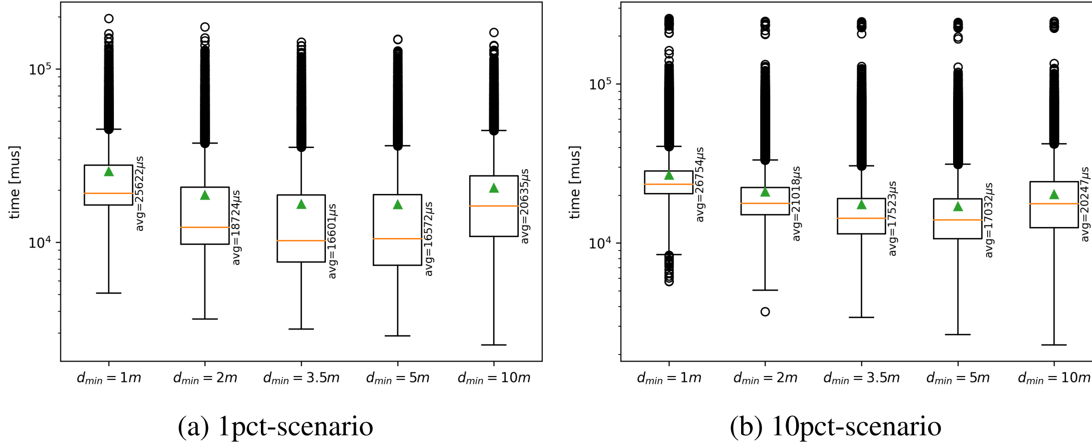
Again, we present our results in table 5.8 and fig. 5.8. Performance data is as expected: Doubling N roughly doubles the required runtime. The increase in runtime is slightly superlinear, probably since the transfer spots sampled for higher N have a higher distance to the pickup and dropoff, leading to slightly longer LOUD-Requests. The change in hit-rate as N gets larger is a little surprising: Increasing N past 8 shows no significant increase in hit-rate compared to the effect that N had on the geometrically-sampled algorithm. This is especially apparent on the 10pct-scenario. Even for $N = 4$, the hit rate on the 1pct-scenario is over 7% whereas for $N = 32$ it just barely manages to reach 10%, and on

the 10pct scenario the difference is even smaller. Considering the impact that N has on the runtime performance, $N = 4$ or $N = 8$ can be considered a sweetspot. The fact that the betweenness-sampled algorithm finds most good transfer vertices even with only four samples, and barely improves for higher values of N, means that the heuristic "High betweenness + Low via-distance" is really good at finding the best possible transfer locations very quickly.

We now vary the tuning parameter d_{min} which, as explained above, increases or decreases the total amount of potential transfer vertices. N is set back to the default value of $N = 8$.

		1min	2min	3.5min	5min	10min
1pct	$\|T\|$	19508	10066	5325	3389	1249
	hit-rate	9.01%	9.34%	9.42%	8.87%	6.26%
	Δ_{cost}	-3030	-3056	-3077	-3286	-3590
	Δ_{detour}	-272s	-299s	-299s	-316s	-355s
	Δ_{wait}	-87s	-69s	-75s	-74s	-82s
	Δ_{trip}	+383s	+414s	+414s	+435s	+517s
10pct	$\|T\|$	19508	10066	5325	3389	1249
	hit-rate	20.84%	21.10%	20.50%	18.92%	12.88%
	Δ_{cost}	-2674	-2795	-2953	-3148	-3550
	Δ_{detour}	-281s	-295s	-314s	-335s	-383s
	Δ_{wait}	-53s	-49s	-45s	-46s	-42s
	Δ_{trip}	+418s	+424s	+439s	+457s	+529s

Table 5.9: Quality values for the betweenness-sampled single-transfer dispatch algorithm for varying values of d_{min} (i.e. minimum distance between potential transfer vertices) on each scenario. Each value of d_{min} is given in minutes.



(a) 1pct-scenario (b) 10pct-scenario

Figure 5.9: Runtime required to resolve a single ridesharing request with the betweenness-sampled single-transfer dispatch algorithm for varying values of d_{min} on each scenario.

We present our results in table 5.9 and fig. 5.9. In addition to the usual quality attributes, we also present how many vertices were chosen as possible transfer vertices.

Higher values of d_{min} increase the distance between each potential transfer stop, spreading them out further across the graph. On the one hand, this might lead to the algorithm finding good transfer spots that are a bit further away from pickup and dropoff. On the other hand, large values of d_{min} might cause the algorithm to miss some good transfer spots that are close to the pickup- and dropoff. The number of possible transfer vertices initially decreases linearly for small values of d_{min} as d_{min} increases, but slightly superlinearly for larger values of d_{min}. $d_{min} = 3.5m$ achieves the highest hit-rate on the 1pct-scenario, while $d_{min} = 2m$ achieves the highest hit-rate on the 10pct-scenario. Good values for d_{min} are dependent on the average distance between pickup and dropoff locations of each request: Since in our scenario, most requests do not exceed 15min of direct distance, $d_{min} = 10m$ would mean that there's only one or two potential transfer stops that actually lie inbetween the pickup and dropoff, explaining the sharp drop in hit-rate. Higher values of d_{min} would be more applicable in scenarios where requests have a higher average direct distance between pickup and dropoff.

On both scenarios, $d_{min} = 5m$ requires the lowest amount of runtime, although just barely beating $d_{min} = 3.5m$. The increase in runtime for low values of d_{min} is due to the large amount of potential transfer stops filling up the BCH-Buckets, as well as increasing the cost of the required linear sweep to find the N best transfer vertices. The increase in runtime for large values of d_{min} can be explained the same way as in the sections about the geometric algorithm: Since the transfer vertices are further spread out, distances from pickup to transfer and from transfer to dropoff become longer, increasing the runtime of each individual LOUD-Request. Overall, for our scenario, we propose $d_{min} = 3.5m$ as a sweetspot, however this might very well be different for different road networks.

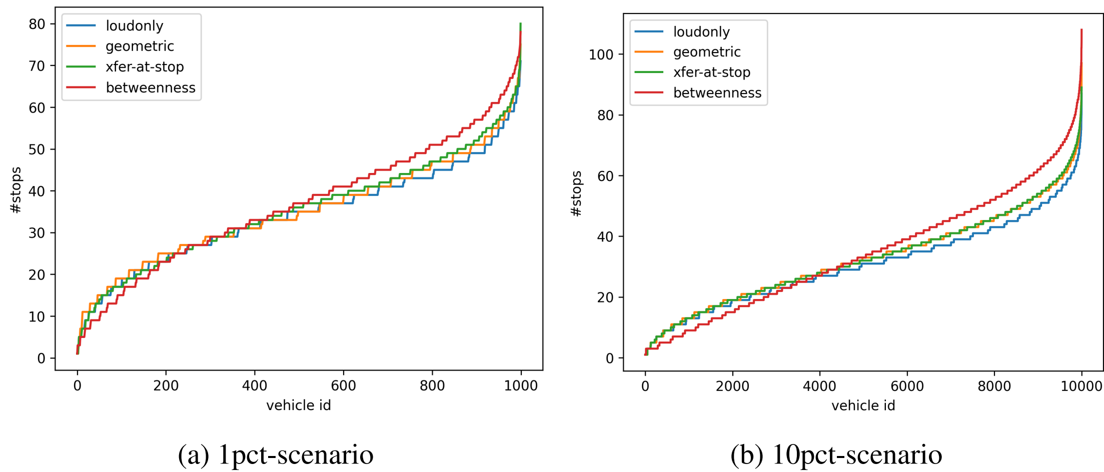
5.7 Vehicle Utilization

We evaluate how strongly the fleet is utilized when using various algorithms. To that end, we evaluate the average number of passengers (occupancy) that are inside each vehicle when driving. This average is weighted by the time driven: If a vehicle spends one hour driving with one passenger, and two hours with two, then its average occupancy is $\frac{1 \cdot 1 + 2 \cdot 2}{1+2} = \frac{5}{3} = 1.666\ldots$. Idle time, where the vehicle is empty, does not count into the average occupancy. We do log the total driving- and idling-time as well though. We also present how well the requests are distributed across the fleet by plotting the number of requests resolved by each vehicle. A perfect request distribution would result in a constant, flat plot, whereas a heavily skewed plot would mean that most requests are resolved by only a couple few central vehicles.

		occup	first dep	last arr	travel time	stops
1pct	LOUD	0.94	6:03AM	8:26PM	14:22	34.11
	XAS	1.13	6:03AM	8:18PM	14:14	35.27
	GEOM	0.96	6:01AM	8:24PM	14:23	35.01
	BTWN	0.97	5:59AM	7:40PM	13:41	36.63
10pct	LOUD	1.23	6:42AM	7:12PM	12:30	30.72
	XAS	2.03	6:42AM	7:01PM	12:19	32.78
	GEOM	1.25	6:40AM	7:06PM	12:26	32.75
	BTWN	1.28	6:30AM	5:36PM	11:06	34.58

Table 5.10: Evaluation results for the vehicle utilization in the two scenarios using various dispatching algorithms. As usual, LOUD is always used in parallel to the single-transfer dispatching algorithm and the best result of the two is used. The average occupancy of a vehicle is weighted by the time traveled with a certain occupancy. The "first dep"-column is the average of the time when a vehicle receives its first request, for each vehicle. Similarily, the "last arr" column is the average of the time when a vehicle resolves its last request, again for all vehicles. The "travel time" column is simply the average time of last arrival minus the average first departure time. The "stops"-column contains the avg. number of stops a vehicle route will have for the entirety of the simulation.

You can see the results of this evaluation in table 5.10. It is clear to see that the vehicle utilization is a little higher in the 10pct scenario than in the 1pct-scenario, averaging an occupancy of just over 1.2 in comparison to just under 1.0 for the 1pct-scenario for almost all algorithms. The only outlier is the transfer-at-stop-algorithm, which results in a significantly higher average vehicle occupancy than all other algorithms on both scenarios. To a degree, this was to be expected: The first trip of the transfer-at-stop algorithm always ends in an already existing stop, which means the vehicle doesn't have to do an almost-empty



(a) 1pct-scenario

(b) 10pct-scenario

Figure 5.10: Distribution of stops among the various vehicle routes. Vehicle-IDs are sorted by final number of stops in route, i.e. vehicle 1 is the vehicle with the lowest number of stops in its route after simulation completion.

ride to the dropoff destination. Furthermore the pickup detour will be very short since that is the main concern for the first trip.

For all other algorithms, the values do not change much compared to original LOUD. As expected, the number of average stops for each vehicle increases, and is highest for the betweenness-sampled algorithm, since its hit rate is the highest. Each hit of the single-transfer algorithm results in the potential creation of four new stops, instead of only two.

Compared to LOUD-only, the use of single-transfer dispatching reduces the time it takes to resolve all incoming ridesharing requests. With our best algorithm, the betweenness-sampled algorithm, this time gets reduced by 41 minutes on the 1pct-scenario and by over an hour on the 10pct-scenario. On average, using the transfer algorithms, vehicles depart a little earlier than without transfers, and also finish their service in less total time. This means that in addition to using the fleet of vehicles more efficiently to resolve the requests, dispatching with transfer routes causes vehicles in the fleet to be utilized earlier on average.

Last, we present the distribution of stops across the vehicle routes in fig. 5.10. As you can see, the distribution of stops is not flat, but rather follows a smooth s-curve. We can see that each vehicle makes around 35 stops on average, which fits the data seen in table 5.10. Some few vehicles get up to 70 or 80 stops into their route, while some other few vehicles are barely used at all. Most vehicles (the middle 60%) however fall neatly in the middle and get about 20 to 40 stops throughout their service time.

We can see that the curve is flattest for the LOUD-only dispatching, and steepest for dispatching with the betweenness-sampled algorithm. This means that while transfer routes might not distribute the requests as equally across the fleet as simple LOUD does, it shows that dispatching with transfer routes might require less total vehicles in the fleet than dispatching with only no-transfer routes to achieve similar dispatching quality. This could

be evaluated in more detail in future work by varying the number of available vehicles and comparing dispatch quality with and without single-transfer routes. We expect there to be some lower bound in the number of vehicles below which dispatch quality greatly decreases, since the capacity of the fleet is fully utilized, and we expect this bound to be lower when utilizing transfer routes in addition to no-transfer routes.

The transfer routes can make better use of already driving vehicles than no-transfer routes, since a driving vehicle only needs to pass close by either the pickup or dropoff point to be eligible for a good transfer route, as opposed to having to pass close to *both*. The curve is slightly steeper for the 10pct-scenario than for the 1pct-scenario, although not by much, especially when comparing the blue LOUD-only curves.

To summarize, the main benefit of transfer routes in ridesharing is that they manage to complete requests *faster* (i.e. with less extra driving time) on average than no-transfer routes for some requests. They do *not* increase the vehicle utilization significantly (except with the transfer-at-stop algorithm) and they cause a less even distribution of the requests on the fleet. For our best algorithm, the betweenness-sampled algorithm, on the best and most realistic available scenario, the 10pct-scenario, a vehicle with single-transfer dispatching manages to, on average, complete its requests 11% faster even though it has 13% more stops, on average, in total. From this we can conclude that a vehicle can, on average, in this scenario, complete requests around 25% faster than without single-transfer dispatching.

5.8 Varying vehicle capacities

Finally, we evaluate a slightly different ridesharing scenario: In this scenario, there are fixed transfer locations distributed across the road network. Large busses with high passenger capacities traverse between these locations, picking up passengers nearby that want to be picked up. After a passenger arrives at such a transfer spot, they get taken to their destination by a smaller vehicle with a lower passenger capacity. We call these small vehicles that drive passengers to the requested dropoff spot "smarts".

We modify the vehicle capacities of our previous scenarios to match the new scenario. We use the betweenness-sampled single-transfer algorithm with $d_{min} = 10m$ to create a list of these fixed transfer locations automatically. We then simply use the same algorithm to dispatch any ridesharing requests. Of course, for this new scenario, it would be best to design a new, specialized algorithm that only considers bus-vehicles for the first trip and only small vehicles for the second, to keep performance costs low and assert that each request is resolved by such a bus-smart-route. Our main interest is to see if this bus-smart behaviour occurs naturally even when not using a specialized algorithm.

In our new scenario, each bus has a capacity of 8 and each smart has a capacity of only 1. This might seem a little unrealistic, but remember that each ridesharing request also contains only a single passenger. Therefore the low smart capacity is still enough to serve a full request. For each of the previously seen scenarios, we create two new scenarios: One

	1pct-10%	1pct-25%	10pct-10%	10pct-25%
bus-only	11.76%	27.36%	14.37%	37.92%
smart-only	85.98%	67.69%	75.56%	46.52%
bus-bus	0.14%	1.47%	1.89%	7.98%
bus-smart	0.24%	0.30%	0.67%	0.64%
smart-bus	1.87%	3.19%	7.51%	6.94%
smart-smart	0.00%	0.00%	0.00%	0.00%

Table 5.11: Relative quantity of the types of routes dispatched by the betweenness-sampled single transfer dispatch algorithm on the various bus-smart-scenarios described in section 5.8. The algorithm was run with the parameters $N = 8$ and $d_{min} = 10m$. LOUD ran concurrently, the best of the two (no-transfer vs single-transfer) dispatching solutions was inserted.

where 25% of all vehicles are busses, and one where only 10% of all vehicles are busses. This results in four scenarios: *1pct-25%*, *1pct-10%*, *10pct-25%* and *10pct-10%*. For each transfer-route, we evaluate how often it falls into one of the categories *bus-bus*, *bus-smart*, *smart-bus* or *smart-smart*. We also log the average occupancy of the busses and compare against the values seen in section 5.7.

You can see the dispatching results of the various bus-smart-scenarios in table 5.11. As expected, most of the requests were answered using no-transfer routes. Because the fleet consists of many more smarts than busses in both the 10%- and 25%-scenarios, most no-transfer routes utilize a smart. However, on the 10pct-25% scenario, while smarts are still used more than busses for no-transfer routes, the gap isn't that large. This can easily be explained: On the 10pct-scenarios, as seen before, vehicle utilization is higher. Since smarts only have a capacity of 1, no ridesharing can take place and busses have to be used to cope with the larger number of requests.

When looking at the single-transfer routes, we can see that almost all of them are either bus-bus-routes or smart-bus routes. Only rarely are bus-smart-routes used. Smart-smart routes are not used at all, which makes sense: Resolving a request using a smart results in a taxi-style ride, since the smart can only pick up the passenger after dropping off any previous passengers and can't pick anyone up while driving the passenger to their destination. Using two smarts with a transfer strictly increases the wait- and trip-time of the passenger as well as the total detour of both vehicles, since the route via the transfer vertex is longer than the direct route taken by the no-transfer route and no ridesharing takes place. Therefore smart-smart routes always have a strictly higher insertion cost than if the first smart of that route would simply drive the passenger all the way to the dropoff.

Another interesting scenario would be a "smart-bus-smart"-scenario, where passengers first get driven to some transfer stop by a smart. A bus then rides from one transfer stop to another. A second smart-ride then takes the passenger from the transfer stop to their desti-

nation.

Designing and evaluating algorithms for these scenarios is out of the scope of this thesis. However, we present a couple of general ideas and thoughts on such algorithms and scenarios, which then might be fleshed out in future work.

We see two practical applications of the smart-bus-smart-scenario, which mostly differ in the behaviour of the busses:

(i) *On-Demand line busses*: The busses behave similarly to classic line busses, traveling from one transfer stop to the next within a single city. However unlike line busses, the route is not fixed but rather dynamically determined by the incoming requests. The bus makes its way to transfer stops *on-demand*.

(ii) *On-demand inter-city busses*: A bus picks up a large number of passengers within one city (or district) and then takes them to a different city (or district) that is far away. The source and target cities are determined on demand by the requests.

The first scenario can simply be solved by running three LOUD-requests: One for the first smart-route, one for finding a bus to get the passenger from transfer stop to transfer stop, and one for the final smart-route. Since the arrival time at the destination transfer stop is not known in advance, the final LOUD-request has to be made after the passenger arrives at the transfer stop. Alternatively, the last LOUD-request could be made with the minimum departure time simply set to the arrival time at the last transfer stop if no further requests are served by the bus. The smart that picks up the passenger might have to wait if the bus serves another request, however the maximum arrival time of the bus at that stop acts as an upper bound on the smarts wait time.

The second scenario is a little more challenging: The goal of a bus is to carry as many passengers as possible when making the trip from one city to the next, to save costs. If simple LOUD-requests were used, like in the first scenario, to determine the route of the bus, the bus would leave the city very quickly and wouldn't pick up many passengers at all.

One option would be to make prebooking of inter-city travel mandatory. Passengers would have to book their trip a day or a couple of hours in advance to have a gurantee of getting a good trip. An ideal bus route could then be calculated that picks up all passengers with the same source and destination cities from the transfer stops closest to the passengers pickup points. This method poses two issues though: First, while we think that for long distance travel mandatory prebooking would be acceptable, being able to book trips just-in-time is one of the major advantages of dynamic ridesharing; it would be better to find a non-prebooking solution. And second, it raises the question why transfer stops are even necessary and the why busses don't just pick up the passengers at their doorstep.

Therefore we present a second idea: In this scenario, only few or even just one transfer stop is available per district or city (depending on size of that district/city). At every transfer stop, busses stand ready, initially empty and without any assigned destination. When a passenger makes a request with pickup spot $p \in V$ and dropoff spot $d \in V$, the closest transfer spots t_1 and t_2 to p and d respectively will be determined (e.g. using a BCH-

query). If there is no bus at t_1 that is assigned the destination of t_2 yet, some bus will be assigned with that destination. After this assignment, any other request with source transfer stop t_1 and target transfer stop t_2 will be assigned to that bus. The bus departs either when it is full or a certain time t after the first passenger entered the bus. We call this time t the *departure delay*, and propose it to be a linear function in the direct distance between t_1 and t_2: $t = \alpha dist(t_1, t_2) + \beta$, with α, β being some constant parameters. A request can only be matched to that bus at t_1 if the passengers arrival time at t_1 is earlier than the departure time of the bus. LOUD-requests for getting the passengers from t_2 to their dropoff can be made as soon as the bus departs from t_1, as the bus will now drive the direct path from t_1 to t_2. After dropping of passengers at t_2, the bus has no assigned destination again and is ready to serve different requests.

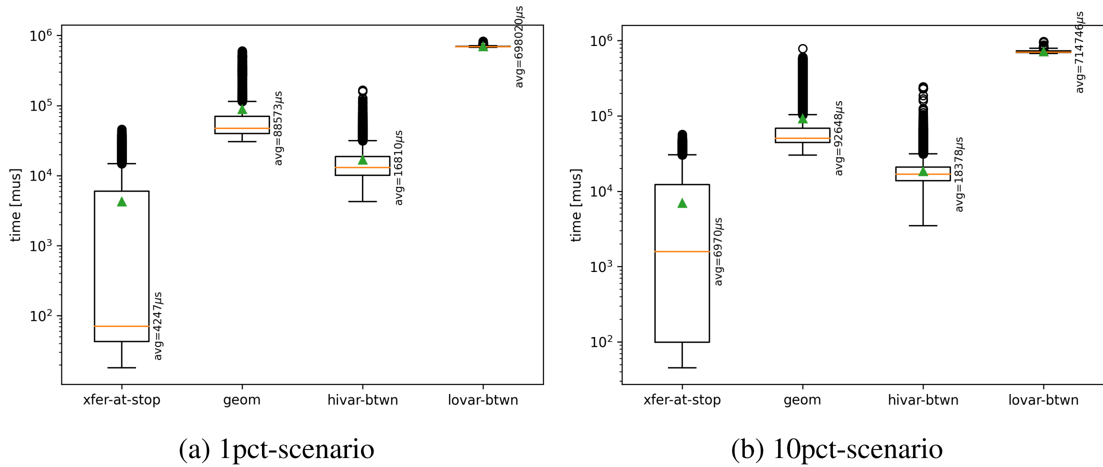
This algorithm makes on-demand long-distance trips possible with a high probability that those long-distance trips are made with a high bus occupancy. Future work here includes implementing the algorithm, fleshing out the details, generating realistic test data and evaluating the algorithm on that data. It would be interesting to see just how many transfer stops and busses per district/city are needed. If too few busses are available, it might not be possible to serve all requests or lead to very long trip times. If too many busses are available however, this could either cause a lot of busses to just idle at the transfer stops or, if the departure delay t is set very low, it would cause low occupancies on the long-distance trips. It would also be interesting to see how these dynamic long-distance busses compare against traditional long-distance line-busses in terms of passenger wait- and trip-times as well as bus occupancy.

5.9 Two-Transfer Dispatching

We now evaluate the four presented multi-transfer dispatching algorithms for their dispatching quality and runtime performance. The four algorithms are the multi-transfer-at-stop algorithm (abbreviated *xfer-at-stop* or XAS), the geometrically sampled algorithm (now abbreviated GEOM), the betweenness-sampled algorithm that only uses the best via-routes (abbreviated LOVAR-BTWN) and the betweenness-sampled algorithm with random transfer spot combinations (abbreviated HIVAR-BTWN). Each of the resulting routes gets compared with the no-transfer route found by LOUD, just like in previous quality evaluations. We evaluate each of these algorithms with parameters $N = 8$ and $M = 2$, i.e. each algorithm samples eight different transfer spot combinations and the resulting route will have two transfer spots. We do not expect higher M to achieve better results. In fact, because the distances of the ridesharing requests are small, only rarely exceeding half an hour of travel time, we expect the multi-transfer dispatching algorithms to perform worse than their single-transfer alternatives.

		XAS	GEOM	HIVAR-BTWN	LOVAR-BTWN
1pct	hit-rate	0.13%	0.16%	2.50%	3.75%
	Δ_{cost}	-8482	-1923	-3329	-3086
	Δ_{detour}	-1166s	-135s	-264s	-253s
	Δ_{wait}	+390s	-51s	-59s	-51s
	Δ_{trip}	+983s	+1246s	+789s	+788s
10pct	hit-rate	0.39%	0.45%	5.99%	9.00%
	Δ_{cost}	-9694	-2532	-3530	-3513
	Δ_{detour}	-1042s	-174s	-289s	-306s
	Δ_{wait}	+81s	-89s	-77s	-59s
	Δ_{trip}	+597s	+922s	+651s	+672s

Table 5.12: Comparison of quality values between the different multi-transfer dispatching algorithms on both scenarios, with $N = 8$ and $M = 2$.



(a) 1pct-scenario (b) 10pct-scenario

Figure 5.11: Runtime required to resolve a single ridesharing request with each of the presented multi-transfer dispatching algorithms on each scenario, using $N = 8$ and $M = 2$.

The results of the evaluation can be seen in table 5.12 and fig. 5.11. In both scenarios, the multi-transfer algorithms perform significantly worse than their respective single-transfer counterpart in regards to hit-rate. Furthermore, multi-transfer trips increase the passenger trip time much more than single-transfer trips do: While single-transfer trips required around five to seven minutes of extra trip time on average compared to the no-transfer route, the multi-transfer trips increase the passenger trip time by over ten minutes on average for each algorithm. Considering the soft-constraint t_{trip}^{max} on the trip time, this explains the low hit-rate: Only on few requests can you "afford" the extra trip time without the cost function blowing up. As before, while the hit-rate is still bad, multi-transfer routes perform better

on the 10pct-scenario than on the 1pct-scenario.

The runtime performance of each of the three algorithms is not affected much by the increase in number of vehicles and request density. This was to be expected, since the same was observable for the single-transfer sampling algorithms. Very surprisingly, while the runtime required by the geometrically sampled multi-transfer algorithm is about three to four times the runtime of the corresponding single-transfer algorithm, the runtime of the multi-transfer hivar-betweenness algorithm *matches* the runtime of the corresponding single-transfer algorithm, despite having to make 24 instead of 16 LOUD-Requests. Since the code for actually making the LOUD-Requests is the same for all three algorithms, and we have checked manually that the correct number of samples are drawn, this must mean that each individual LOUD-Request requires significantly less time on average for the betweenness-sampled algorithm, probably because distances between pickup and dropoff of each LOUD-Request are very small.

The performance of the low variance betweenness-sampled algorithm is bad, taking around 700ms per request on average. In fact, out of all presented algorithms in this thesis, this is the only algorithm that took longer than 24 hours to dispatch all requests in the ridesharing-simulation for the 10pct-scenario, meaning that this algorithm is not able to cope with that kind of request frequency. While the hit-rate of lowvar-btwn is best, it is not practical for real-life use.

Finally, the qualitative values for the multi-transfer transfer-at-stop algorithm seem out of place and much worse than all other algorithms. It achieves the lowest hit-rate of all algorithms, but when it hits, it reduces the insertion cost and vehicle detour drastically, at the expense of strongly increased wait- and trip-times. It is the only algorithm that shows an increase in wait time. We have thoroughly debugged the algorithm and can safely say that this is not an implementation error. We will explain the reason for the weird values: Let s_t be the stop of the vehicle that picks up the passenger from the first transfer stop, and s_{t+1} be the stop that comes after the transfer stop. The request to get the passenger from the first to the second transfer stop is a prebooking-request: $r_t^{dep} > r_t^{req}$, since the vehicle can only depart after the passenger has arrived. Therefore, after inserting the multi-transfer route calculated by the transfer-at-stop algorithm, s_{t+1} can only be reached after the timepoint r_t^{dep}. There are only two scenarios where this does not immediately violate the maximum arrival time of s_{t+1}:

(i) The departure time s_t is in the future, roughly around r_t^{dep}.

(ii) The distance traveled between s_t and s_{t+1} is very large.

The first case basically never happens, since routes are very short. All requests that are resolved using the transfer-at-stop algorithm fall into the second type of scenario. Therefore, the trip time increases drastically because the passenger has to ride along the full route from s_t to s_{t+1}, and then still has to get taken from the location of s_{t+1} to the dropoff location. Because excess trip-time is weighted ten times higher than excess wait time, the entire insertion cost is strongly dominated by the trip time and detour, making the cost for the excess wait time comparatively insignificant. This causes the drastic increase in wait

time. Only the cost of very few long distance requests which LOUD serves with taxi-like routes (i.e. very high detour) can be improved through the algorithm, since the algorithm is able to strongly improve the detour. For short- and medium-distance requests, most vehicles that could pick up the passenger with a low wait time go in a wrong direction, causing a big increase in trip time. The viable pickup vehicles that might be going in the right direction are far away, and the excess wait time cost alone is often larger than the total cost of the insertion found by LOUD. The sampled algorithms generally do a better job since they make sure that the passenger is traveling in "roughly the right direction" as they make their way from one transfer stop to the next. This is not guaranteed by the transfer-at-stop algorithm. The multi transfer-at-stop algorithm requires very little runtime, but this is because in almost all cases no viable trips are found from the first to the second transfer stop, resulting in no LOUD-requests being executed.

You can see an example of routes found by the high- and low-variance betweenness-sampled and geometrically-sampled algorithm in the Appendix (chapter 7). The transfer-at-stop-algorithm was not able to find a solution to that request. In these images, you can clearly see the difference in the routes found by each of the three algorithms: The high variance betweenness algorithm finds many different routes and deviates quite strongly from the geometric straight-line route in order to make its way to faster roads (e.g. highways). The low-variance betweenness algorithm, as predicted, does not find a large variety of routes; the only difference in the routes is in the very last stretch. The geometrically-sampled algorithm finds routes that follow the straight-line route between pickup and dropoff closely, with transfer vertices lying on the geometric thirds between pickup and dropoff. Sadly, this leads the vehicle right through the slow roads of the inner city.

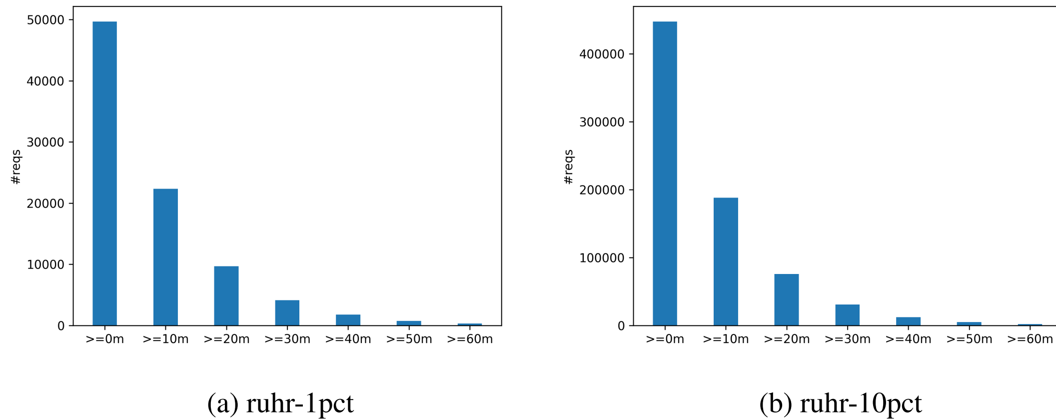
Overall, we can say that multi-transfer routes are not practical for our scenarios, i.e. urban inner-city ridesharing, since the requests simply do not cover a large enough distance to warrant multiple transfers. In more rural ridesharing scenarios, where longer distances are covered, multi-transfer dispatching might be more sensible. Sadly, we lack the data for running such a scenario.

For all upcoming evaluations, we will only evaluate the multi-transfer dispatching algorithms GEOM and *hivar-btwn* (which we now just call BTWN), since the other two algorithms are not practical for real-life use.

5.10 Running on a larger scenario

Note: Due to time constraints, experiments in this section were run on `PC1` to retrieve performance- and quality-data. In order to achieve comparability with performance data in other sections (e.g. section 5.4), we also ran the ruhr-1pct, but not ruhr-10pct, scenario on `PC2`. Running the 10pct-scenario on `PC2` would have exceeded our deadline.

We will now run all presented single-transfer algorithms and the two previously mentioned multi-transfer algorithms on a similar, but larger set of test data based on the Rhein-Ruhr



(a) ruhr-1pct

(b) ruhr-10pct

Figure 5.12: Distribution of direct distances between pickup and dropoff for each ridesharing request on the ruhr-1pct- and ruhr-10pct-scenario. Each bar portrays how many requests have a direct distance from pickup to dropoff that are at least as long as the time given on the corresponding x-tick.

metropolitan region in western Germany. As stated before, the *ruhr-1pct*-scenario contains around 50000 requests and 3000 vehicles, and the *ruhr-10pct*-scenario around 450000 requests and 30000 vehicles.

As you can see in fig. 5.12, even though the road network is bigger than the Berlin road network used in all other sections, the direct distances of requests are still short with more than half of all requests having a direct distance of less than ten minutes. Therefore, even though this scenario is larger, we don't expect our multi-transfer algorithms to perform much better than in the Berlin-scenario.

You can see the results in fig. 5.13 and table 5.13. We can see that the performance of the algorithms shows similarities to what we have seen before: XAS has a high variance in its runtime, because of the variable number of LOUD-requests, the sampling algorithms scale very well with the increase in scenario size, and BTWN has the best average runtime of all algorithms. It's noteworthy that the required runtime for the sampling algorithms actually *decrease* when switching from the 1pct- to the 10pct-scenario, suggesting that LOUD-requests on the larger scenario are slightly quicker. This could be due to shorter average request distances or due to lower costs for ordinary insertions, which tighten the bound on the dropoff-after-stop Dijkstra query.

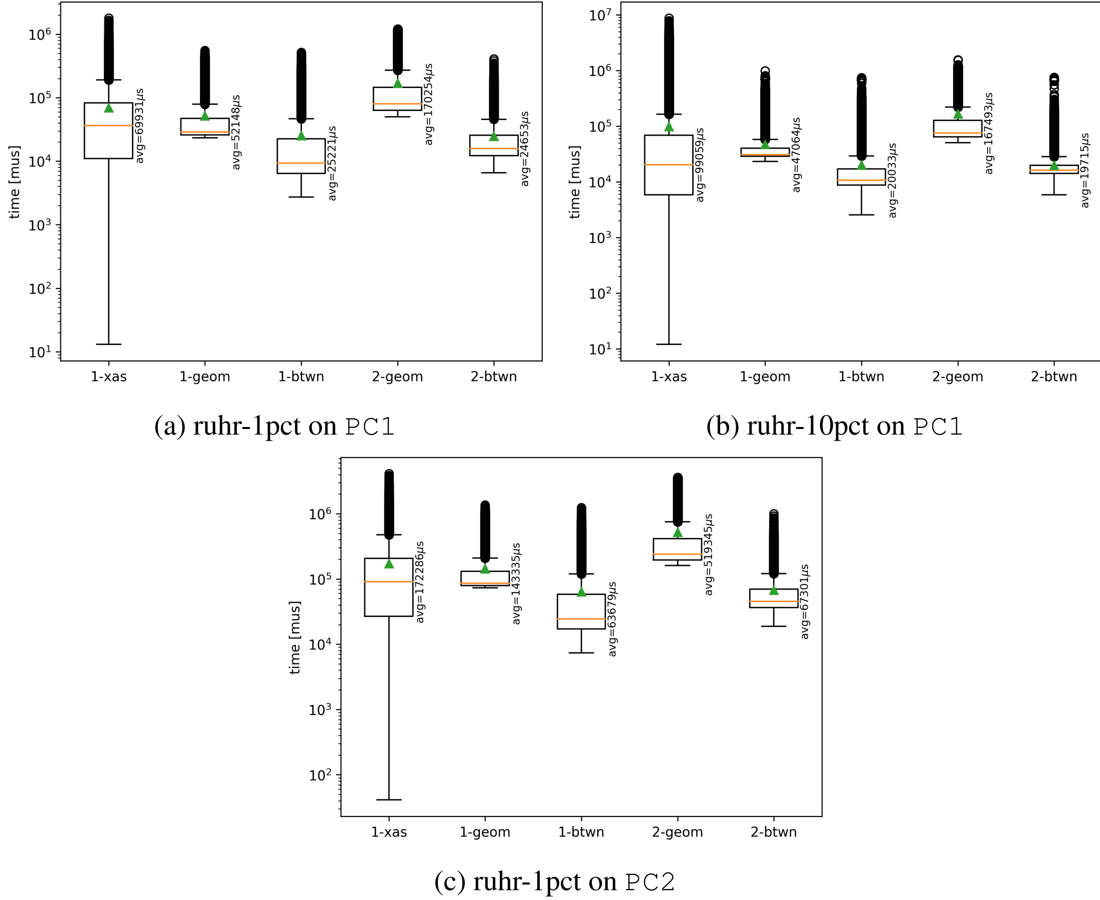
Comparing the performance data of the PC2-run on the 1pct-scenario to the data seen in section 5.4 and section 5.9, we can see that the increase in graph- and scenario size comes with an increase in required runtime by a factor of about four. This matches with the data seen in [3], where the authors have evaluated LOUD on both the Berlin- and Ruhr-scenarios and also found a roughly four-times increase in runtime per LOUD-request. Therefore, while we sadly didn't have the time to run the 10pct-scenario on PC2, we expect roughly

		1-XAS	1-GEOM	1-BTWN	2-GEOM	2-BTWN
ruhr-1pct	hit-rate	4.54%	1.00%	4.65%	0.01%	2.31%
	Δ_{cost}	-7639	-4721	-3432	-4350	-3432
	Δ_{detour}	+69s	-59s	-325s	-84s	-295s
	Δ_{wait}	+42s	-145s	-65s	-380s	-37s
	Δ_{trip}	+107s	+353s	+562s	+695s	+1000s
ruhr-10pct	hit-rate	8.51%	2.74%	10.98%	0.08%	7.49%
	Δ_{cost}	-3435	-2107	-2941	-1869	-3713
	Δ_{detour}	-221s	-189s	-324s	-157s	-340s
	Δ_{wait}	+12s	-38s	-26s	-29s	-31s
	Δ_{trip}	+378s	+476s	+558s	+801s	+794s

Table 5.13: Qualitative evaluation of single- and multi-transfer algorithms on the larger ruhr-1pct- and ruhr-10pct-scenarios. The single-transfer-algorithms are prefixed with "1-", multi-transfer algorithms with "2-" and $M = 2$ as well as all default values in table 5.2 are used.

the same slowdown factor for our algorithms on the ruhr-10pct scenario compared to the berlin-10pct scenario as the slowdown seen in [3], which is a little under four.

When looking at the qualitative values in table 5.13, as we expected, multi-transfer routes do not perform significantly better on the Ruhr-scenarios; in fact, they perform slightly *worse* than on the Berlin-scenarios. In general, the hit rate of each algorithm is roughly half of what we have seen on the Berlin-scenarios. We speculate that this decrease in hit-rate is due to the lower vehicle density on the Ruhr scenarios, since the graph is five times bigger while there are only three times more vehicles than on the Berlin scenarios. Still, a hit rate of almost 11% by the single-transfer betweenness algorithm on the 10pct-scenario, and likely a higher hitrate on a theoretical 100pct-scenario, is still a significant improvement to the dispatching of ridesharing requests.



(a) ruhr-1pct on `PC1`

(b) ruhr-10pct on `PC1`

(c) ruhr-1pct on `PC2`

Figure 5.13: Runtime required to resolve a single ridesharing request with single- and multi-transfer dispatching algorithms on the two large ruhr-scenarios *ruhr-1pct* and *ruhr-10pct*. The single-transfer-algorithms are prefixed with "1-", multi-transfer algorithms with "2-" and $M = 2$ as well as all default values in table 5.2 are used.

5.11 Combined single- and two-transfer dispatching

Up until now, we have only evaluated single-, two-, and multi-transfer dispatching algorithms by running one of them together with LOUD and always inserting the better of the two routes. We will now run LOUD, a single-transfer and a double-transfer dispatching algorithm concurrently, only ever inserting the best of the three results. We evaluate how often each algorithm supplied the best solution. All combinations of algorithms in $\{$*single-xfer-at-stop*, *single-geom*, *single-btwn*$\} \times \{$*multi-geom*, *multi-hivar-btwn*$\}$ will be tested on both scenarios.

Our results are presented in table 5.14. For each combination, it contains the hit-rate, i.e. how often the cost of the insertion produced by one algorithm beats the cost of the other

	AlgComb	$min = c_0$	$min = c_1$	$min = c_2$	$c_0 < c_1 < c_2$	$c_1 < c_2 < c_0$	$c_0 < c_2 < c_1$
1pct	XAS/GEOM	92.26%	7.64%	0.07%	75.16%	0.04%	17.00%
	GEOM/GEOM	97.06%	2.77%	0.16%	95.21%	0.02%	1.85%
	BTWN/GEOM	91.22%	8.70%	0.07%	90.27%	0.04%	0.95%
	XAS/BTWN	89.78%	7.88%	2.31%	70.72%	0.54%	19.07%
	GEOM/BTWN	94.66%	2.35%	2.99%	83.69%	0.22%	10.97%
	BTWN/BTWN	89.77%	7.79%	2.40%	85.31%	1.03%	4.46%
10pct	XAS/GEOM	84.85%	14.89%	0.19%	70.59%	0.15%	14.25%
	GEOM/GEOM	92.36%	7.35%	0.27%	89.44%	0.13%	2.92%
	BTWN/GEOM	80.96%	18.78%	0.09%	74.40%	0.19%	6.56%
	XAS/BTWN	80.31%	14.94%	4.66%	71.95%	2.08%	8.35%
	GEOM/BTWN	87.96%	5.34%	6.67%	79.18%	0.75%	8.78%
	BTWN/BTWN	78.74%	15.97%	5.08%	76.52%	3.72%	2.22%

Table 5.14: Comparison of hit-rates of no-, single- and multi-transfer routes when competing against each other. c_0 stands for the cost of the no-transfer-route, c_1 for the cost of the one-transfer-route and c_2 for the cost of the two-transfer-route. The first three columns state how often out of all requests a certain cost was the lowest. The last two rows give the relative quantities for the cost constellation in the label. The fourth column contains the number of occurrences of the "expected" request resolution, i.e. no-transfer is better than single-transfer is better than multi-transfer. The second-to-last column contains how often a two-transfer-route is better than the no-transfer-route, but worse than the single-transfer-route. The last column contains how often a two-transfer route was better than the one-transfer route, but not better than the no-transfer route, essentially "shadowing" the two-transfers-success to be registered in column 3.

algorithms. Furthermore, we are also interested in how often the multi-transfer-route is better than the no-transfer route, but worse than the single-transfer-route. Our initial assumption when designing the multi-transfer algorithms was that almost all requests where the multi-transfer route brings an improvement over LOUD would be better solved by the single-transfer algorithms anyway, due to the limited size of the road network and the low average distance between pickup and dropoff. We also check how often the multi-transfer-route is better than the single-transfer route, but worse than the no-transfer-route, which essentially "shadows" the achievement of the two-transfer-route.

The main conclusion to draw from the table is that both the single-transfer and multi-transfer betweenness-sampled algorithm always outperform the other transfer algorithms: First, unlike the geometric algorithm, the multi-transfer betweenness algorithm achieves the minimum insertion cost for a significant amount of requests. Second, when looking at the last column of the table, we can see that the multi-transfer betweenness algorithm regularly beats the insertion cost of the single-transfer algorithms GEOM and XAS, even when not achieving the minimum insertion cost due to LOUD finding a better no-transfer route. We can determine the total number of times where the two-transfer-route beat the one-transfer-route by adding the values from column 3 and 6. For example, we can see from the row *1pct-xas-btwn* that the two-transfer betweenness-algorithm achieves a better cost than the transfer-at-stop algorithm for 21.38% of request.

A major drawback of the transfer-at-stop algorithm is that it cannot make use of any idle vehicles for pickup, since a pickup-vehicle has to have at least one planned stop, which then serves as the transfer stop. The sampling algorithms on the other hand can make use of idle vehicles for each trip between pickup/transfer/dropoff-locations. In cases where this is useful, the transfer-at-stop algorithm gets beaten by the multi-transfer-stops, explaining the high values in the last column for the transfer-at-stop algorithm.

Lastly, the data shows that although multi-transfer routes don't seem particularly suited to our scenario, the single-transfer routes do not always improve on routes where the two-transfer-algorithm beats LOUD, contrary to our expectations. Even when running the betweenness-sampled multi-transfer algorithm against its single-transfer counterpart, which is the best transfer algorithm presented by us, it still finds a better route than the single-transfer route on 5% of all requests on the 10pct-scenario. An improvement of over 5% of all requests is equivalent to over 9000 total improved requests, which is far from negligible. This furthermore reinforces our belief that multi-transfer routes can bring substantial improvements to ridesharing routes on larger scenarios and road networks.

5.12 More than two transfer stops

Note: All experiments in this section were run on `PC1`, since even on that machine, they took over a week to complete. While this means that the performance data in this section is not comparable to other sections, our main interest is the change of runtime performance for growing values of M.

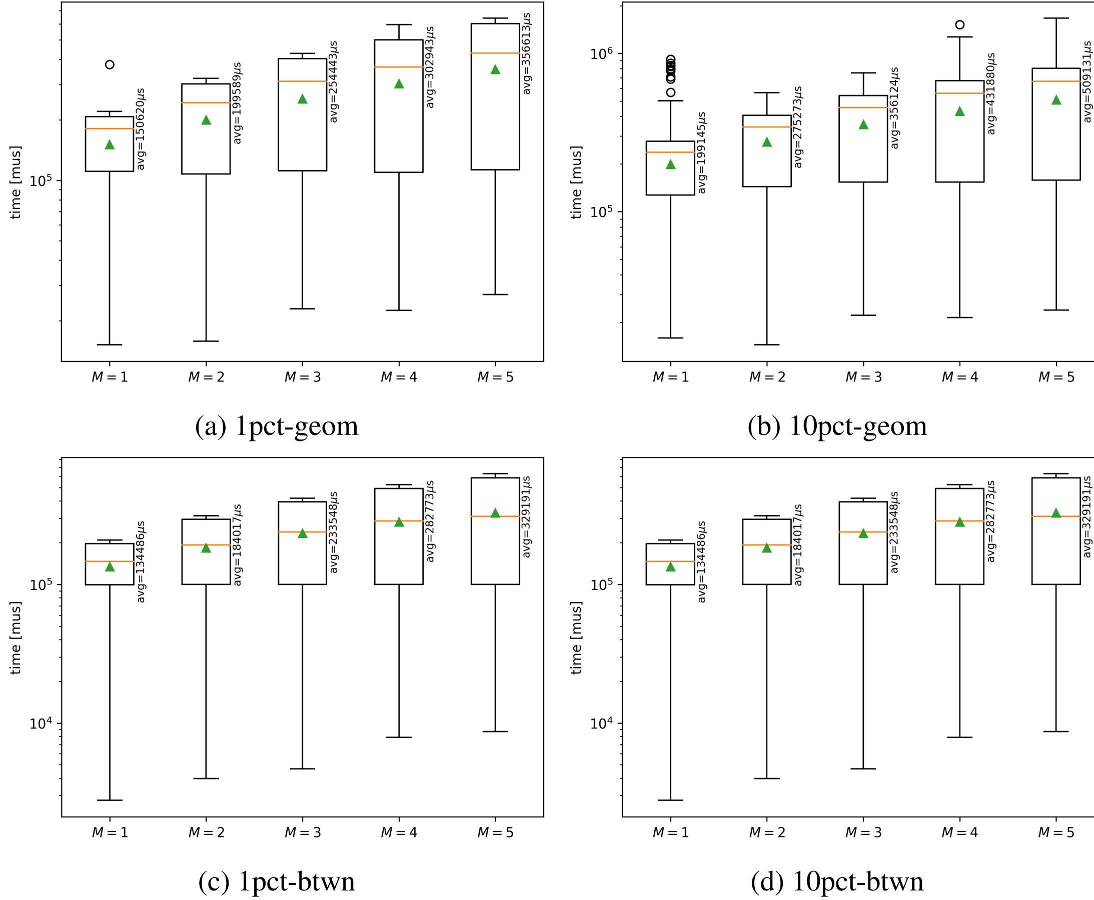
This relative data is mostly unaffected by the choice of machine.

Last but not least, we evaluate the impact of higher values for M (number of transfer stops) on the runtime performance and dispatching quality of our multi-transfer dispatching algorithms. As before, we will only evaluate the geometric and high-variance betweenness sampled algorithm, as the other two presented algorithms are not practical for real-life use. There is one problem though: Our multi-transfer algorithms perform very badly on on our scenarios and often do not work properly, even for just $M = 2$, on small- to medium distance requests. As we have seen in section 5.3, the average ridesharing request has a direct distance from pickup to dropoff of about ten minutes. For the betweenness-sampled algorithm, since there is a minimum distance d_{min} between possible transfer vertices, it is impossible for more than two or three transfer vertices to lie neatly "between" the pickup- and dropoff-vertex. This causes the transfer route to either sample the same transfer vertex for two consecutive stops, which is invalid, or make the passenger travel back and forth between the transfer vertices before driving to the dropoff point. For the geometric algorithm, the deviation of the normal distribution becomes very small, causing the same locations to be sampled over and over again. This triggers rejection sampling, requiring most vertices to be resampled over and over again, which drastically increases the runtime.

To resolve these issues, we modify the 1pct and 10pct scenario slightly. For each request in the scenarios, we modify the location of the dropoff vertex to be far away from the pickup vertex by choosing a new dropoff vertex with *Dijkstra-rank* of at least $\frac{|V|}{2}$ with respect to the pickup vertex. A vertex v has a Dijkstra-rank $d \in \mathbb{N}$ with respect to a source vertex s if it is the d-th vertex popped from the priority queue in a Dijkstra-query with source s. The vertex with Dijkstra-rank 1 is the source vertex s, and the vertex with Dijkstra-rank $|V|$ is the last vertex popped and thus the vertex furthest away from s. The dropoff vertex is a random vertex of all vertices with Dijkstra-rank $\geq \frac{|V|}{2}$.

All results of these experiments are presented in fig. 5.14 and table 5.15. We take a look at the performance data first: First, note that even though these experiments are run on a much more powerful machine, it still takes over 150ms on average to resolve a request with $M = 1$, much longer than in previous experiments. This is of course because the distances between pickup and dropoff are strongly increased, leading to many long-distance LOUD-queries. It shows that the performance of a single LOUD-request is strongly affected by the distance between pickup and dropoff. Second, as we can see from the long whiskers going down from the boxes into the single-millisecond range, some requests can still be answered very quickly. These requests are actually just the very last requests coming in: Since we didn't alter the vehicle service times, the final hundred-or-so requests can't be answered by any algorithm since the minimum arrival time at the dropoff (minimum pickup time + direct distance) exceeds the service time of all vehicles. Therefore vehicles get filtered out quickly, and these requests can be answered quickly (with a cost of ∞).

Next, lets take a look at how the runtime changes as M increases: While we can see, as expected, that the required runtime increases as M goes up, the increase is not linear. For example, for the 1pct-geometric experiment, increasing M from 1 to 3, which leads



(a) 1pct-geom

(b) 10pct-geom

(c) 1pct-btwn

(d) 10pct-btwn

Figure 5.14: Runtime required to resolve ridesharing requests on the modified long-distance scenarios using the and geometric- and betweenness-sampled multi-transfer dispatching algorithm with various values of $M \in \{1, \ldots, 5\}$. All other parameters are set to their default values $N = 8$, $\eta = 0.25$ and $d_{min} = 5m$.

to double the executed LOUD-Requests, "only" increases the runtime by about 70% on average, instead of doubling it. However, as noted in the last paragraph, the performance of LOUD-Requests is strongly affected by the distance between pickup and dropoff. The individual LOUD-Requests for higher values of M are naturally much shorter, since the total distance is effectively divided by M. As seen before in previous experiments, the betweenness-sampled algorithm shows the same behaviour in runtime, but is a little faster than the geometrically-sampled algorithm by a constant amount of time, since the BCH-queries are quicker than the linear search required for mapping the geometric coordinates to graph vertices. The runtime on the 10pct-scenario shows the same rate of increase as on the 1pct-scenario, each request just requires roughly a constant factor more time than in the smaller scenario, same as in previous evaluations.

		$M=1$	$M=2$	$M=3$	$M=4$	$M=5$
1pct-geom	hit-rate	3.82%	0.36%	0.00%	0.00%	0.00%
	Δ_{cost}	-13887	-5220	N/A	N/A	N/A
	Δ_{detour}	-1221s	-363s	N/A	N/A	N/A
	Δ_{wait}	-111s	-195s	N/A	N/A	N/A
	Δ_{trip}	+1348s	+1769s	N/A	N/A	N/A
10pct-geom	hit-rate	5.60%	1.00%	0.09%	0.00%*	0.00%
	Δ_{cost}	-13338	-9822	-6824	-3518	N/A
	Δ_{detour}	-1131s	-679s	-395s	-49s	N/A
	Δ_{wait}	-139s	-230s	-327s	-359s	N/A
	Δ_{trip}	+1261s	+1541s	+1664s	+1417s	N/A
1pct-btwn	hit-rate	9.91%	4.83%	2.45%	0.39%	0.03%
	Δ_{cost}	-16901	-16883	-14597	-15709	-13501
	Δ_{detour}	-1381s	-1286s	-991s	-1109s	-1460s
	Δ_{wait}	-174s	-336s	-449s	-564s	+54s
	Δ_{trip}	+1235s	+1512s	+1569s	+1767s	+3160s
10pct-btwn	hit-rate	13.36%	7.63%	4.57%	2.33%	1.14%
	Δ_{cost}	-15825	-16950	-16037	-16451	-16391
	Δ_{detour}	-1363s	-1351s	-1262s	-1186s	-1054s
	Δ_{wait}	-239s	-391s	-475s	-569s	-645s
	Δ_{trip}	+1219s	+1380s	+1479s	+1516s	+1564s

Table 5.15: Comparison of quality values of the geometric- and betweenness-sampled multi-transfer dispatching algorithm running on the modified long-distance scenarios with various values of $M \in \{1, \ldots, 5\}$. All other parameters are set to their default values $N = 8$, $\eta = 0.25$ and $d_{min} = 5m$.
*Not actually zero, but so small that it gets rounded to zero.

Next, we take a look at the quality evaluation. We can see that the geometric algorithm fails to answer *any* request better than LOUD can on the 1pct-scenario starting from $M = 3$ upwards. The betweenness-sampled algorithm fares a little better and is actually able to improve the cost of 0.03% (which is around 5 out of over 16000 requests) for $M = 5$, and more for smaller values of M. We can see that for each additional transfer stop, the trip time increases drastically compared to the no-transfer route found by LOUD. To compensate for this massive increase in trip-time, the multi-transfer-route has to show a dramatic improvement in vehicle detour, which the geometric algorithm simply fails to achieve. One reason why the geometric algorithm doesn't fare well at all is because especially on long-distance requests, the shortest route from pickup to dropoff does not follow the geometric straight-line between pickup and dropoff at all, as seen in fig. 5.2 earlier. Furthermore, when comparing table 5.15 to table 5.12, we can see that both multi-transfer algorithms work better on the modified long-distance scenario than on the original 1pct/10pct-scenarios, as ex-

pected. As seen in previous experiments, the transfer algorithms perform better on the 10pct scenario, achieving a higher hit rate than on the 1pct scenario and better passenger wait times while maintaining roughly the same trip time and detour.

Concluding, we find the results in this section promising. While our scenario clearly isn't well suited for multi-transfer dispatching, we have shown that the two presented algorithms scale well with an increase in the number of transfer stops. Furthermore, while the required runtime of over $150ms$ per request is restrictive for interactive applications, it is an acceptable runtime to provide potential end-users a pleasant experience, especially when considering that multi-transfer routes are only used for very long-distance requests. We have also shown that despite the bad match between our data and multi-transfer dispatching, we can still improve on some long-distance requests with up to $M = 4$ transfer stops using our high variance betweenness-sampled algorithm. On datasets that are better suited for multi-transfer ridesharing, e.g. with ridesharing between multiple cities, we expect the algorithms to perform even better.

6 Conclusion

In this work, we have designed and implemented multiple single- and multi-transfer dispatching algorithms for dynamic ridesharing on the basis of the no-transfer dispatching algorithm LOUD and evaluated these on a transport simulation with a realistic set of requests and vehicles on the road network of Berlin and the German Ruhr-area. We were able to show that single- and multi-transfer dispatching is able to significantly improve the dispatch quality of ridesharing requests with regards to detour and wait time, while only increasing the required runtime by about one order of magnitude compared to LOUD. Out of the three presented single-transfer dispatch algorithms, the betweenness-sampled algorithm shows the most promising results, achieving the highest hit-rate of all the algorithms while simultaneously having the lowest required runtime when run on the 10pct-scenario. We expect the sampling algorithms to even more strongly outperform the transfer-at-stop algorithm in the real-life "100pct"-scenario.

We have also shown four algorithms for multi-transfer routes, expanding on the single-transfer algorithms. Again, the multi-transfer algorithm based on betweenness sampling shows the most promising results. It is regularly able to find better solutions to long-distance requests than the no- and single-transfer alternatives, while still maintaining a reasonable runtime performance. We were furthermore able to show that the required runtime of multi-transfer dispatching scales well with the number of requested transfer stops, opening up possibilities for very long distance "cross-country" requests.

Future Work

There is some more work to be done with regards multi-transfer dispatching. Most importantly, it would be interesting to see multi-transfer ridesharing evaluated on a larger, realistic inter-city scenario with more long-distance requests. We have already shown that even on our intra-city scenario, long distance requests can be improved on significantly by utilizing multi-transfer routes. On an inter-city scenario, transfer stops might occur in each of the source and destination cities, with only few vehicles driving between cities but many vehicles serving request within the confines of a single city.

The smart-bus-smart scenario described in section 5.8 is of special interest as this mode of operation may provide a modern and more dynamic alternative to line-busses and trains for long-distance inter-city travel. Future work includes implementing the described algorithm or designing a new one, generating and collecting data for this scenario and evaluating the runtime performance and dispatching quality. It would be especially interesting to see how

the ride quality on such a scenario compares to traditional line-busses or trains in regards to trip-time and probability of missing a connection.

Furthermore, some improvements could be made to the presented algorithms: The performance of every presented algorithm could be improved by running all of the required LOUD-requests in parallel. Since no insertions take place while making these requests, synchronization overhead should be minimal. In the sampled-transfer algorithms, it might be possible to improve dispatching quality by doing some local optimization around the sampled vertices: After performing the LOUD-requests, iterate through vertices v in the neighbourhood of the sampled transfer vertex t and calculate the cost of a potential insertion with the vehicles and the stop indices found by the LOUD-requests, but with transfer location v instead. The runtime required to calculate this cost is very low, and it might be possible to find a transfer location that is slightly better than what was sampled previously.

Lastly, the test scenarios could be modified to represent transfer-dispatching more realistically. We believe that each transfer should incur a penalty in the insertion cost function, as transfers decrease customer satisfaction. Furthermore, while our algorithms can use any vertex on the road network as transfer stops, most vertices are unsuited for transfers in real life. For example, our algorithms might sample a transfer vertex somewhere in the middle of a federal highway where it is illegal to stop. To resolve this, the road network would need to be manually curated, labeling each vertex as transfer-viable or not-transfer-viable, or even giving a rating of each vertex between 0 and 1, where 0 means "not eligible for transfer" and 1 means "perfect for transfer". One interesting topic of research would be to automate this curation on the basis of OpenStreetMap-Data, perhaps using machine learning.

7 Appendix

All routes in the following images were found for the same request. The pickup location is on the top left of the image, the dropoff on the bottom right.

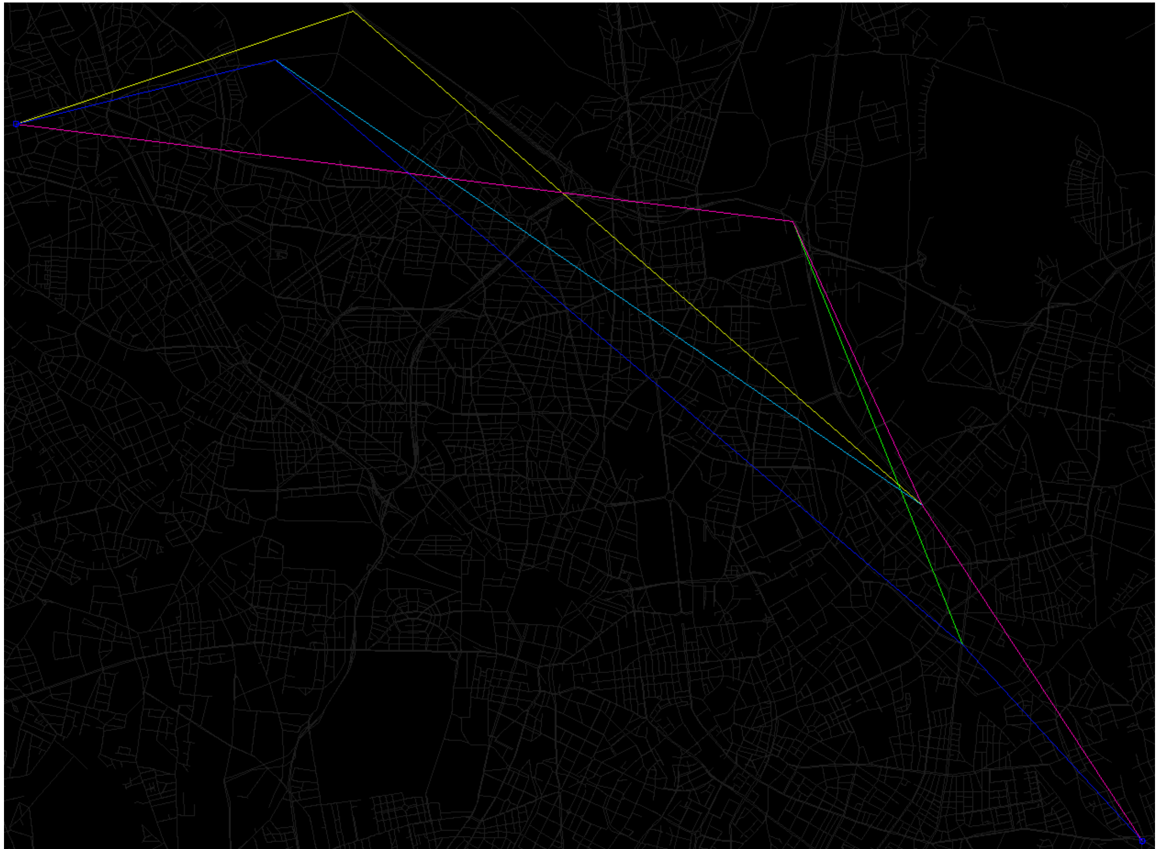


Figure 7.1: Example of multi-transfer routes found by the algorithm *hivar-btwn*.

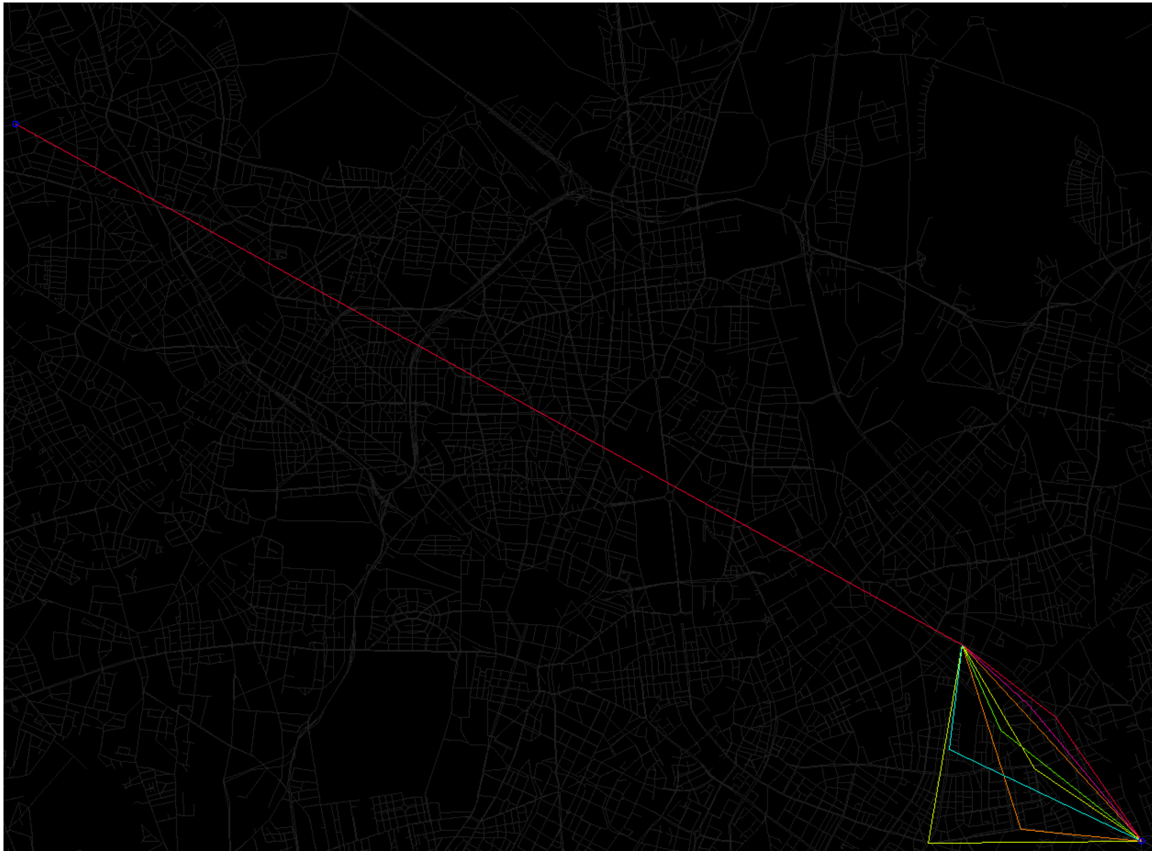


Figure 7.2: Example of multi-transfer routes found by the algorithm *lovar-btwn*.

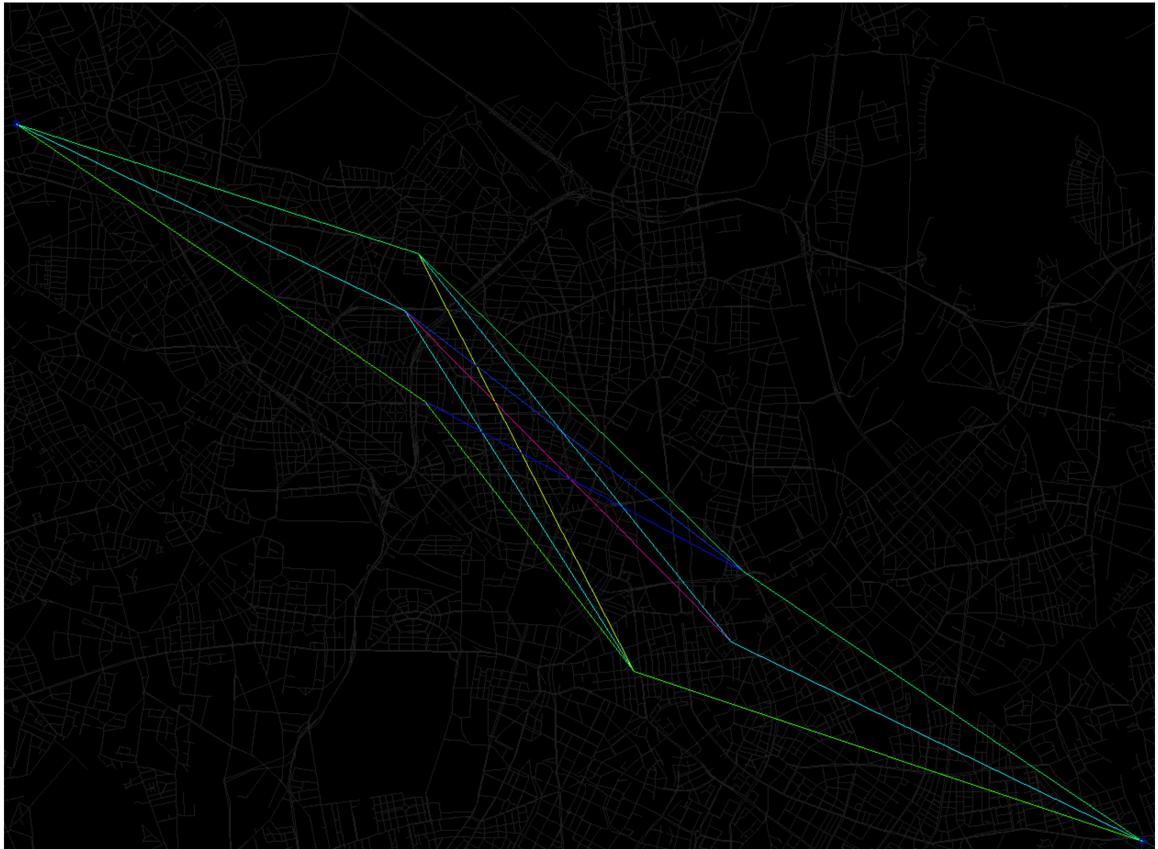


Figure 7.3: Example of multi-transfer routes found by the algorithm *geom*.

Bibliography

[1] Heron of Alexandria. *Metrica*. 60.

[2] Joschka Bischoff, Michal Maciejewski, and Kai Nagel. „City-wide shared taxis: A simulation study in Berlin“. In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. 2017, pp. 275–280. DOI: 10.1109/ITSC.2017.8317926.

[3] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. „Fast, Exact and Scalable Dynamic Ridesharing“. In: (2021). arXiv: 2011.02601 [cs.DS].

[4] Yen-Long Chen, Kuo-Feng Ssu, and Yu-Jung Chang. „Real-time Transfers for Improving Efficiency of Ridesharing Services in the Environment with Connected and Self-driving Vehicles“. In: *2020 International Computer Symposium (ICS)*. 2020, pp. 165–170. DOI: 10.1109/ICS51289.2020.00041.

[5] Tobias Columbus and Reinhard Bauer. „On the Complexity of Contraction Hierarchies“. In: 2009. URL: https://api.semanticscholar.org/CorpusID:14809386.

[6] Edsger W. Dijkstra. „A note on two problems in connexion with graphs“. In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[7] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. „An Algorithm for Finding Best Matches in Logarithmic Expected Time“. In: *ACM Trans. Math. Softw.* 3.3 (1977), pp. 209–226. ISSN: 0098-3500. DOI: 10.1145/355744.355745. URL: https://doi.org/10.1145/355744.355745.

[8] Robert Geisberger et al. „Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks“. In: *Experimental Algorithms*. Ed. by Catherine C. McGeoch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 319–333. ISBN: 978-3-540-68552-4.

[9] J. H. Halton. „Algorithm 247: Radical-Inverse Quasi-Random Point Sequence“. In: *Commun. ACM* 7.12 (1964), pp. 701–702. ISSN: 0001-0782. DOI: 10.1145/355588.365104. URL: https://doi.org/10.1145/355588.365104.

[10] Donald B. Johnson. „Efficient Algorithms for Shortest Paths in Sparse Networks“. In: *J. ACM* 24.1 (1977), pp. 1–13. ISSN: 0004-5411. DOI: 10.1145/321992.321993. URL: https://doi.org/10.1145/321992.321993.

[11] Sebastian Knopp et al. „Computing Many-to-Many Shortest Paths Using Highway Hierarchies". In: Jan. 2007. ISBN: 978-1-61197-287-0. DOI: 10.1137/1.9781611972870.4.

[12] Dominik Pelzer et al. „A Partition-Based Match Making Algorithm for Dynamic Ridesharing". In: *IEEE Transactions on Intelligent Transportation Systems* 16.5 (2015), pp. 2587–2598. DOI: 10.1109/TITS.2015.2413453.

[13] Hanan Samet. „The Quadtree and Related Hierarchical Data Structures". In: *ACM Comput. Surv.* 16.2 (1984), pp. 187–260. ISSN: 0360-0300. DOI: 10.1145/356924.356930. URL: https://doi.org/10.1145/356924.356930.