Multi-Hop Ride Sharing

Florian Drews and **Dennis Luxen**
Karlsruhe Institute of Technology
Institute of Theoretical Informatics, Algorithmics II
76128 Karlsruhe, Germany
florian.drews@student.kit.edu, luxen@kit.edu

Abstract

We study the problem of ride sharing in road networks. Current approaches to this problem focus on simple bulletin board like services or on algorithms that do not allow to transfer. In this work, we present a solution with an arbitrary number of transfers that respects personal preferences of the users. We engineer the ride sharing problem by searching a graph that represents a timetable network similar to those used for train networks. Our experimental analysis shows that our solution provides good performance and that it is significantly faster than a naive search. The algorithm achieves about an order of magnitude higher speedups over Dijkstra's algorithm than what could be expected from previous work.

Introduction

Ride sharing is a popular way to share empty seats in a car for lower overall costs. *Drivers* offer a ride to the passenger, i.e. the *rider*. There exist a number of Internet services that provide the necessary matching between the two parties. Unfortunately, these systems are mostly simple solutions that mimic the behaviour of billboard systems, i.e, the feature set is small. For example, drivers *have* to fix routes in advance and riders are not able to make transfers between rides.

The paper is structured as follows. We give an overview on related literature as well as an introduction to single-hop ride sharing. Then, we explain the notion of *multi-hop ride sharing*. The modelling of our approach is explained and then assessed in an experimental evaluation. Finally, we provide a conclusion and an outlook into future work.

The contribution is two-fold. First, we show that the ride sharing problem can be solved by exploiting time-expanded graphs that are used for routing in transportation networks. Second, we give an efficient implementation that leverages Contraction Hierarchies, an efficient speedup technique to Dijkstra's algorithm, for certain subproblems. Both, our modelling and the application of Contraction Hierarchies, allow us to tackle the resulting graph search problem by more expensive methods.

Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Related Work

Computing point-to-point shortest (or fastest) path queries in a graph has been solved by Dijkstra's seminal algorithm [Dijkstra, 1959] since the early times of computer science. A road network is modelled as a graph $G = (V, E)$ with $|V| = n$ nodes and $|E| = m$ edges. Each edge $e \in E$ is associated with a cost $c(e)$ that is required to traverse that edge. A *path* P is a sequence of edges that connects a sequence of nodes. The *length* of a path is denoted by $l(P)$ and is the sum of its edge weights and $\mu(s, t)$ is the length of a shortest path between $s, t \in V$. For the sake of simplicity, consider that nodes are identified by their ID, i.e. $a \in V$ is treated as a number if appropriate. While the running time of Dijkstra's algorithm is clearly polynomial, it does not scale well to large instances, e.g. road networks of continental size. A well-tuned implementation still needs a few seconds for a single shortest path query on such a network even on today's hardware. This is not feasible for real-time services.

A^* [Hart, Nilsson, and Raphael, 1968] is a heuristic improvement to Dijkstra's algorithm for point-to-point queries. The search is guided towards the goal by following the path of lowest expected total cost. Each node v is associated with two cost functions:

- $g(v)$ is the (tentative) distance from the source.
- $h(v)$ is a lower bound of the distance to the target, also called *potential*.

Function $h(v)$ must be an *admissible* heuristic, i.e. a node's potential must not overestimate the distance to the target, i.e. $\mu(s, v) + h(v) \leq \mu(s, t), \forall v \in V$. The order of the priority queue is then given by the sum $f(v) = g(v) + h(v)$. An example for such an heuristic is the Euclidean distance which never overestimates the shortest path distance in a road network (with distance metric). A^* finds an optimal path if it exists, but it does not have strong worst-case guarantees on all graphs. Its performance depends on the accuracy of the heuristic. Goldberg and Harrelson [Goldberg and Harrelson, 2005] report on a number of variants of a sophisticated potential function based on landmarks and the triangle inequality. The observed performance on road networks is roughly an order of magnitude higher than Dijkstra's algorithm depending on the test instance.

Contraction Hierarchies (CH) [Geisberger et al., 2012] is an exact speedup technique. It has a convenient trade-off be-

tween preprocessing and query time by exploiting the inherent hierarchy of a road network. CH *shortcut* all nodes of the graph in some order. Contracting means that a node is (temporarily) removed from the network and replaced by as few shortcut edges as possible to preserve shortest path distances. The resulting data structure of the Contraction Hierarchies preprocessing is the union of the original graph and the set of shortcut edges. A shortest path computation on this data structure is essentially a bidirectional version of Dijkstra's algorithm that considers only edges to more important nodes, i.e. so-called upward edges. Note that this graph forms a directed acyclic graph (DAG) where edges only lead to more important nodes, i.e. nodes contracted later. The set of *forward* edges in this DAG is denoted by $G^{\uparrow}$ and the set of *reverse*, i.e. *backward*, edges by $G^{\downarrow}$. The length of a shortest path from node u to v in the forward (backward) search space is denoted by $d^{\uparrow}(u,v)$ ($d^{\downarrow}(u,v)$). The only crucial difference to bidirectional Dijkstra is the stopping criterion of the bidirectional search that adds nodes into the priority queues until the tentative distance of an added node exceeds the lower bound that may exist for a shortest path. A shortest path goes over a *middle node* that is settled in both (half-)searches and for which CH guarantee correct labelling in both search directions.

Computing a table of all pairwise shortest path distance between a set of nodes can be trivially done by running a quadratic number of queries. While this is already significantly faster with Contraction Hierarchies than with a naive implementation of Dijkstra's algorithm, this table can be computed much more efficiently with the algorithm of [Knopp et al., 2007]. The main observation is that when computing a quadratic number of pairwise distances a recurring set of important nodes gets settled in nearly all searches. The algorithm basically consists of two phases of *half searches* which we explain below. Consider a set of sources S and a set of targets T. Also, consider an initialized, but empty distance table D, i.e. all entries set to ∞. In the first phase, a simple CH backward search is conducted for each node $t \in T$. During each of these *half searches*, the pair $\left[t, d^{\downarrow}(v,t)\right]$ is noted for each settled node $v \in G^{\downarrow}$. Thus, the algorithm notes the distances to the target nodes at each and every potential middle node that is settled during the backward search phase. The information that is stored at each node v is called a *bucket* b_v, which is an unsorted, but dynamic array in practice.

In the second phase of the distance table computation, the forward searches are conducted for all nodes $s \in S$. Whenever a node v is settled at distance $d^{\uparrow}(s,v)$, its bucket b_v is scanned. Given a bucket element $\left[t, d^{\downarrow}(v,t)\right]$, the corresponding entry in the distance table $D[s,t]$ is updated if the following condition holds:

$$D[s,t] > \underbrace{d^{\uparrow}(s,v)}_{\text{forw. search}} + \underbrace{d^{\downarrow}(v,t)}_{\text{bucket entry}} .$$

Note that this condition is tested for each bucket and every entry encountered during the second phase of the algorithm.

The search spaces are small. Hence, the size of the buckets will be small, as will be the number of nodes that carry a non-empty bucket at all. Computing such a distance table with the above algorithm is a matter of mere seconds since only $O(|S| + |T|)$ half searches have to be conducted. The quadratic overhead to initialize and update the distance table entries is close to none in practice. It is easy to see, that computing a one-to-many (or many-to-one) query is a special case of the general algorithm to compute distance tables. We will use this algorithm in the description of single-hop ride sharing as was as our own method of multi-hop ride sharing from the subsequent section.

A related method is Transit Node Routing by [Bast, Funke, and Matijevic, 2009] which was recently simplified by [Arz, Luxen, and Sanders, 2013]. It exploits the observation that (almost) all long distance paths will enter an *arterial network* at some point. Queries are computed by getting to the entrance into this sub network quickly and combining it with information from a preprocessed distance table. *Local* routes that do not enter the arterial network are computed by a fallback algorithm. So, the preprocessing must compute this information. Somewhat similar is the method of Compressed Path Databases [Botea, 2011] that exploits similar properties in game graphs. Unfortunately, the preprocessing is super-linear and, thus, is unlikely to scale well in practice on graphs with millions of nodes and edges.

Single-Hop Ride Sharing

The following method by [Geisberger et al., 2010] computes offers with the smallest detours with respect to a request. We give a short recapitulation of the method here.

An offer *perfectly fits* a request only if origin and destination locations of driver and rider are identical, but requirements are more diverse in reality. For example, drivers and riders make detours when meeting at a specific location. Assume that network distances resemble costs. A matching that considers these kind of detours is said to compute *reasonable fits*:

Definition 1 *An offer $o = (s,t)$ and a request $g = (s',t')$ form a* reasonable fit *if there exists a path $P = \langle s, \ldots, s', \ldots, t', \ldots, t \rangle$ in G with $l(P) \leq \mu(s,t) + \varepsilon \cdot \mu(s',t')$, with $\varepsilon > 0$.*

In this setting, the driver is given an incentive to pick up the rider at their start location s' and to drop them off at their destination t'. While trivial choices for the tuning parameter, e.g. $\varepsilon = \infty$, make any match reasonable, a match is economically worthwhile if and only if there is an ε for which

$$\mu(s,s') + \mu(s',t') + \mu(s',t') + \mu(t',t) - \mu(s,t) \leq \varepsilon \cdot \mu(s',t'). \quad (1)$$

holds. Again, we are assuming that network weights resemble traveling costs. It is easy to see that reasonable passengers will pay at most $\frac{1}{2} \cdot \mu(s',t')$. Otherwise, it would be cheaper for the passenger not to join the ride at all. In other words, joining reasonable rides allows travelers to have costs lower than those associated with traveling alone.

Matching can be done in this setting by evaluating distances from a request to offers in the data base. For a dataset of k offers $o_i = (s_i, t_i)$, i=1..k, and a single request $g = (s',t')$, $2k+1$ shortest path distances $\mu(s',t')$,

$\mu(s_i, s')$ and $\mu(t', t_i)$ are evaluated. The detour for offer o_i is $\mu(s_i, s') + \mu(s', t') + \mu(t', t_i) - \mu(s_i, t_i)$.

The algorithm of [Knopp et al., 2007] is adapted to store preprocessed (half-)search spaces in buckets. For each s_i the forward search space $G^{\uparrow}(s_i)$ is computed in advance and stored in the *forward bucket* each potential meeting node u. Likewise, *backward buckets* are stored to speed up the computation of all $\mu(t', t_i)$. The shortest path distance $\mu(s', t')$ is computed separately. Queries are done by running a backward (forward) search from s' (t') and checking against the forward (backward) bucket of each encountered node in the search space. Queries for a data base of 10^5 entries run in the order of 5–50 milliseconds depending on values for ε. Unfortunately, the method is not easily generalizable to making transfers with efficient computation of minimum detours. Likewise, the offers are time-**in**dependent which makes it necessary to ignore bucket entries that are in the past or have too much waiting time to be reasonable. Also, it may be necessary to keep separate databases for each day. We refer the reader to the publication of [Geisberger et al., 2010].

Interestingly, [Abraham et al., 2012] provide a different variant of the single-hop ride sharing algorithm based on hub labeling using a technique called *double-hub indexing*. The query duration does not depend on the number of offers, but on the squared size of the label set of source and target nodes, which can be asymptotically less work.

Multi-Hop Ride Sharing

We like to give users of a ride sharing system even greater flexibility. Instead of allowing the riders to join one and only one driver, we allow them to transfer between drivers, i.e. ride with one driver for some time and then transfer to another one. The algorithm of Section can be extended to handle more than one hop by checking more distances, but this leads to a combinatorial explosion for all the pairwise distances that must be checked. But multiple hops may lead to connections that would have been impossible otherwise.

The paper is structured as follows. First, we present ideas on how to model multi-hop ride sharing in a merged setting of routing in a dynamic transport network with timetables *on top* of a road network. Furthermore, our model allows driver and rider specific preferences like maximum wait times and maximum detour. Second, we apply this model to develop an abstract representation of reasonable routes and describe the algorithms that add and remove offers, as well as the corresponding query algorithm to find best matches. Third and finally, we conduct an experimental study on the performance and quality of our solution.

Modelling Multiple Hops. As a first step to allow more realistic transfers, offers are now associated with a departure time and implicitly with an arrival time, too. To reduce complexity, we define a sufficiently high number of *stations* in the road network where riders switch the car, i.e. perform the *hop*. These are the start and endpoints of all trips. Therefore, *multi-hop offers* as well as *multi-hop requests* are represented by triples $\langle s, t, \tau \rangle$, where s is a start station, t a target station, and τ the time of departure. The earliest arrival time at t is given by $\tau' := \tau + \mu(s, t)$. Note that travel times on the road network are time-*in*dependent, whereas ride sharing on top of this network is modelled with time-dependency.

For now, we say that a driver has only one empty seat to share but that a rider can make several transfers. We note that a rider and driver may have to wait for some to actually join a ride. We denote the waiting time at station s_i for a given match m by $\omega_m(s_i)$ and the duration of path $P := \langle s_0, s_1, \ldots, s_t \rangle$ by

$$d(m) = d(P) := \sum_{i=0}^{t} \left(\omega_m(s_i) + \mu(s_i, s_{i+1}) \right) .$$

The raw travel time (without any waiting) $\mu(s_1, s_t) := \sum_i \mu(s_i, s_{i+1})$ is the sum of the individual travel times. As we are dealing with a scenario that accounts for departure and arrival times, we have to account for the fact that waiting periods must not be infinite. The rider should not have to wait too long for pick up. And the driver is usually already taking a detour to pick up the rider as argued previously. Thus, waiting times should be limited by a factor $\delta \geq 0$. It models the maximum percentage of time relative to the shortest path distance offer (request) that rider (or driver) are willing to wait in order to share the ride. We define reasonable delays for rider and driver:

Definition 2 (Reasonable Delay) *A match m is said to have* reasonable driver's delay *if the waiting does not exceed a relative threshold, i.e.*

$$\delta_d \geq d(P) - \frac{d(P) - \mu(s,t)}{\mu(s,t)} = \frac{d(P)}{\mu(s,t)} - 1. \qquad (2)$$

Let $r = (s', t', \tau, \delta_r)$ be a request for a set of offers O and $P := \langle s_1, \ldots, s_n \rangle$ a path where the potential (sub-)rides $m_i = (s_i, s_{i+1}, \tau_i)$, fit together such that $\tau_i + d(g_i) \leq \tau_{i+1}$ is called a fit*. Likewise, a ride m is said to have* reasonable rider's delay *δ_r if the waiting does not exceed a relative threshold, i.e.*

$$\delta_r \geq \frac{d(P) - \mu(s_1, s_t)}{\mu(s_1, s_t)} = \frac{d(P)}{\mu(s_1, s_t)} - 1. \qquad (3)$$

It is called a reasonable fit *if the driver and rider have reasonable delays and if the drivers detour is reasonable, too, as modelled in Equation 1 of the previous section.*

In other words, a fit is a sequence of rides, that allows the rider to travel from the origin to the destination via (potentially) multiple hops. A fit that minimizes the rider's delay δ_r is called *best fit*. Figure 1 gives a visualization. The rider waits 5 minutes at s_1 and reaches the destination with a relative delay of $\delta_r = 1/3$.

Slotted Time-Expanded Graph. Our approach to ridesharing and to finding a best fit has striking similarities to the problem of finding fastest train connections in a railway/-timetable network. The drivers provide time-dependent connections between station through their offers, but note that the underlying road network is assumed to have static travel times.

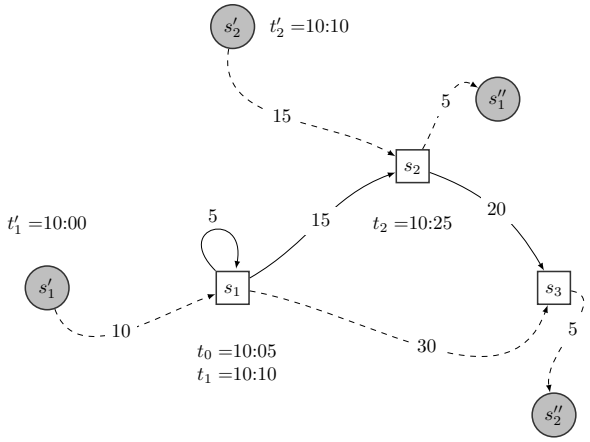

Figure 1: The Offers $o_1 = (s'_1, t'_1, \tau'_1)$ and $o_2 = (s'_2, t'_2, \tau'_2)$ for Request $r = (s_1, s_3, \tau_0)$ Yield a Two Hop Ride $g_1 = (s_1, t_1, \tau_1)$, $g_2 = (s_2, s_3, \tau_2)$ with Path $P = \langle s_1, s_2, s_3 \rangle$.

We adapt the common technique of (simplified) time-expanded graphs [Pyrga et al., 2004] (TEGs). Such a graph models the timetable information into its topology. There exists a node for every event in the timetable, i.e., one node for every *departure* or *arrival* of a train. For every connection, there exists a *train edge* which connects a *departure node* of station S_1 with an *arrival node* of station S_2 at time τ. Each edge is associated with a weight that is the travel time. Note that stations $S =: \{s_1, \ldots, s_k\}$ are represented by sets of nodes and not by single nodes and the set of nodes is sorted according to the time of the event they represent. So-called *transfer edges* connect the time-ordered nodes of a station by edges to model waiting within the station. Our idea is to build a TEG whose edges that is a super set to all *potentially* reasonable fits which are induced by the set of offers. By potentially we mean that we insert an edge if there *could* be some later request for which it might be reasonable. We will give a more detailed explanation in the following sections. A best multi-hop fit can then be computed by an earliest-arrival query in the resulting graph. A TEG is a directed and acyclic graph (DAG) since edges move forward in time. On one hand, it suffices to run a graph traversal like BFS on the graph to discover shortest paths. On the other hand, we prune the search space by a goal-directed graph search as we show in the following.

Travel itineraries are generally error-prone. Unexpected events like traffic conditions, accidents, severe weather, etc. make it hard to predict travel times with certainty. Shared rides are especially frail to delays as they are mostly organized between private partners that are all but bound to a service level agreement. Thus, one may not assume exact travel times for our ride sharing approach. For that reason, we adapt the TEG concept to what we call a *Slotted Time-Expanded Graph* (STEG). Our idea is to introduce time-slots which chop continuous time into discrete and equal-sized ranges. We postpone departure and arrival events until the end of any time-slot. As a consequence, we are inserting waiting periods at the stations. This waiting period is a backup to compensate for unexpected delays while still being able to make a transfer. We formalize this description by the following definition:

Definition 3 (Slotted Time-Expanded Graph (STEG)) *A STEG is a directed acyclic graph $G = (V, E)$ which maintains a time range of $t_r = s_l \cdot s_c$, where $s_l \in \mathbb{N}_+$ is the length of a time-slot and $s_c \in \mathbb{N}_+$ is the total number of slots. A node $u := (w, i) \in V$ resembles station w at time-slot i. Directed edges $e := (u, v, o) \in E$, where $u := (a, i)$ and $v := (b, j)$, resemble (potential) rides with offer o going from station a to b with respect to the corresponding time-slots, i.e. $i = \lfloor \tau_a / s_l \rfloor$ and $j = \lceil \tau_b / s_l \rceil$ for departure time τ_a and arrival time τ_b. Transfer edges model waiting at the station such that a time-slot node of a station is connected to the next slot in time.*

Adding an edge into a STEG implies that *some* waiting time may be added to an offer and we notice that rides that may have been reasonable without waiting may be rendered unreasonable, while the driven detour may still be perfectly fine. On one hand, this approach delays traveling by design, but on the other hand, it adds *some* reliability to making transfers and thus reflects an arguably more realistic scenario.

Adding and Removing Offers. Initially, we model an empty system with a set of stations $\mathcal{S}$ that may resemble good meeting places in reality, e.g. designated parking lots and related venues. The STEG holds a node for each time slot per station and all transfer edges are present. When an offer is added to the system, we insert a number of edges into the STEG. An edge $e = (u, v, o)$ has to be added if the driver of offer o *could* take some rider from u to v with reasonable detour and delay. While this could result in $|\mathcal{S}| - 1$ outgoing edges for each offer's departure node, we add only those edges that represent potentially reasonable routes. More precisely, we omit those edges that violate the driver's delay and detour constraints.

Therefore, we introduce (hopefully smaller) sets $\mathcal{S}_1$ and $\mathcal{S}_2$ which resemble candidate sets. The sets exploit the triangle inequality and are defined as follows:

Definition 4 (Reasonable Station Candidates) *Consider an offer $o = (s, t, \tau, \delta, \varepsilon)$ that specifies source, target, departure time, as well as upper bounds for delay and detour. Consider the set of all reasonable rides $G := \{(s', t', \tau) | s', t' \in \mathcal{S}\}$ for offer o. A super set of the source stations from which reasonable routes depart for an offer o is given by*

$$\mathcal{S}_1 := \{s' \in \mathcal{S} \mid \mu(s, s') + \mu(s', t) \leq (1 + \delta) \cdot \mu(s, t)\}.$$

A super set of the target stations at which reasonable routes arrive is given by

$$\mathcal{S}_2 := \{t' \in \mathcal{S} \mid \mu(s, t') + \mu(t', t) \leq (1 + \delta) \cdot \mu(s, t)\}.$$

The definitions for both sets are very similar and it is easy to see that this superset covers all reasonable fits. The

Listing 1: Adding a Multi-Hop Offer

```
function add_offer(o := (s, t, τ, δ, ε)) do
  S1, S2 = {}
  foreach s', t' ∈ S do
    if μ(s, s') + μ(s', t) ≤ (1 + δ) · μ(s, t) then
      S1 := S1 ∪ s'
    end
    if μ(s, t') + μ(t', t) ≤ (1 + δ) · μ(s, t) then
      S2 := S2 ∪ s'
    end
  end
  foreach Station s' ∈ S1 do
    foreach Station t' ∈ S2 do
      g := (s', t', τ)
      τ' := τ + μ(s', t')
      P := ⟨s, s', t', t⟩
      if is_reasonable(g, P, δ, ε) then
        u := (s', ⌊τ/s_l⌋)
        v := (t', ⌈τ'/s_l⌉
        insert_edge(u, v, o)
      end
    end
  end
end
```

needed distances can be computed easily by two one-to-many queries on the underlying road network. If there exists a distance table for all pairwise distances between the stations then a one-to-many query boils down to a column (line) scan. Note that not all combinations in $\mathcal{S}_1 \times \mathcal{S}_2$ are feasible for every offer. Therefore, we have to verify each candidate before actually adding the edge. The complete algorithm is given as pseudo-code in Listing 1.

Removing offers is necessary when a ride has been matched or in case a driver wishes to retract the offer. Scanning for edges to delete is cost intensive as it has to scan the outgoing edge lists of $|\mathcal{S}|$ stations. Instead we do a *lazy delete*, which means that we mark the offer itself as deleted and simply ignore its edges. The deletion of edges is done during queries where edges get scanned anyway and, thus, the cost is amortized over later operations. We note that mutual exclusion is necessary during deletion in a multi-threaded system.

Matching. We compute a *best fit* for a given request by running an earliest arrival query using Dijkstra's algorithm on the STEG as briefly mentioned before. Note that we can still use the label setting variant of Dijkstra's algorithm. Consider that distances on the underlying road network are available upon request. The query can be pruned at all nodes that violate delay constraints. First, for every edge that is relaxed during the search, we get a (tentative) value for the target node of the edge. We do not insert nodes into the priority queue that violate any threshold, called pruning rule 1. But we can also apply a second, more stricter pruning by using the underlying road network. For each settled node v, we compute the shortest path distance $\mu(v, t')$ in the road network graph. These distances can be looked up in constant time in a table of all pairwise distances between stations. We assume that we could leave v right away by a direct connection to the target. Obviously, this straight connection is a lower bound to the actual distance, i.e. an optimistic estimate. If this *lower bound path* violates our constraints, we do not relax any edges of v, called pruning rule 2.

It is easy to see that both pruning approaches only discard non-optimal nodes. Interestingly, the second approach is an admissible heuristic for goal-directed search with A^* as it never over-estimates the costs. We use this finding in our implementation and its experimental evaluation in the following section.

Experimental Evaluation. We implemented the above data structures and algorithms in C++ using the GCC Compiler version 4.3.2 with full optimizations. The experiments are done on a single core of Machine G. The test graph instance is *osm-germany-3*. We preprocess the road network using the shared-memory parallel Contraction Hierarchies processing of Vetter [Vetter, 2009]. A binary heap is used as the priority queue data structure. In addition we compute a distance table for all pairwise distances between the stations used for pruning. This table is computed with the algorithm of [Knopp et al., 2007].

In absence of realistic test data, we define rider and driver to split costs evenly and to have equal preferences for detour and delay, i.e. $\delta := \delta_r = \delta_d$. We generate 10 000 stations for our data set. The locations of stations are selected uniformly at random with the assumption that node density strongly correlates with population density. Source and target locations for our trips are then drawn uniformly at random from the stations. According to a case study conducted by the Federal Ministry of Transport, Building and Urban Development of Germany [Bundesministerium für Verkehr, 2008] the vast majority of trips take place during the day between 6 am and 8 pm. We pick the departure time of any trip to be from this interval to simulate a single day. The length of a time slot in the STEG is 30 minutes. This gives an expected waiting time of 15 minutes at the station which we see as reasonable. Since the starting time of all slots are synchronous, it suffices to represent edge weights by a multiple of the slot length.

We present three experimental results. First, we evaluate the number of edges inserted into the STEG depending on reasonable values for the delay factor δ and for the detour factor ε. We add 1 000 random offers and average over the number of inserted edges. Figure 2 gives a plot of the results. Note the logarithmic scale of the y-axis. We observe on the left hand side of Figure 2 that varying ε has a significant effect on the number of edges even when the allowed delay is high. On the right hand side, we observe that there is a point where further increasing the delay δ hardly increases the number of inserted edges. This appears to be true for all curves, i.e., all plotted maximum detours. As a rule of thumb we can say that it happens when maximum detour and delay thresholds are equal ($\delta \approx \varepsilon$).

		Dijkstra			A^*			
$\delta = \varepsilon$	Edges [10^9]	#sett. [10^3]	#scan. [10^6]	Query [s]	#sett. [10^3]	#scan. [10^6]	Query [s]	speedup
0.1	0.1	17	9	0.14	0.035	0.05	0.001	140
0.2	1.2	24	98	3.54	0.121	0.82	0.047	75
0.3	3.8	23	285	13.50	0.112	2.02	0.173	78
0.4	7.9	22	562	31.32	0.089	3.39	0.332	94
0.5	13.6	22	947	58.55	0.085	5.31	0.564	103

Table 1: Query Performance of Our Algorithm.

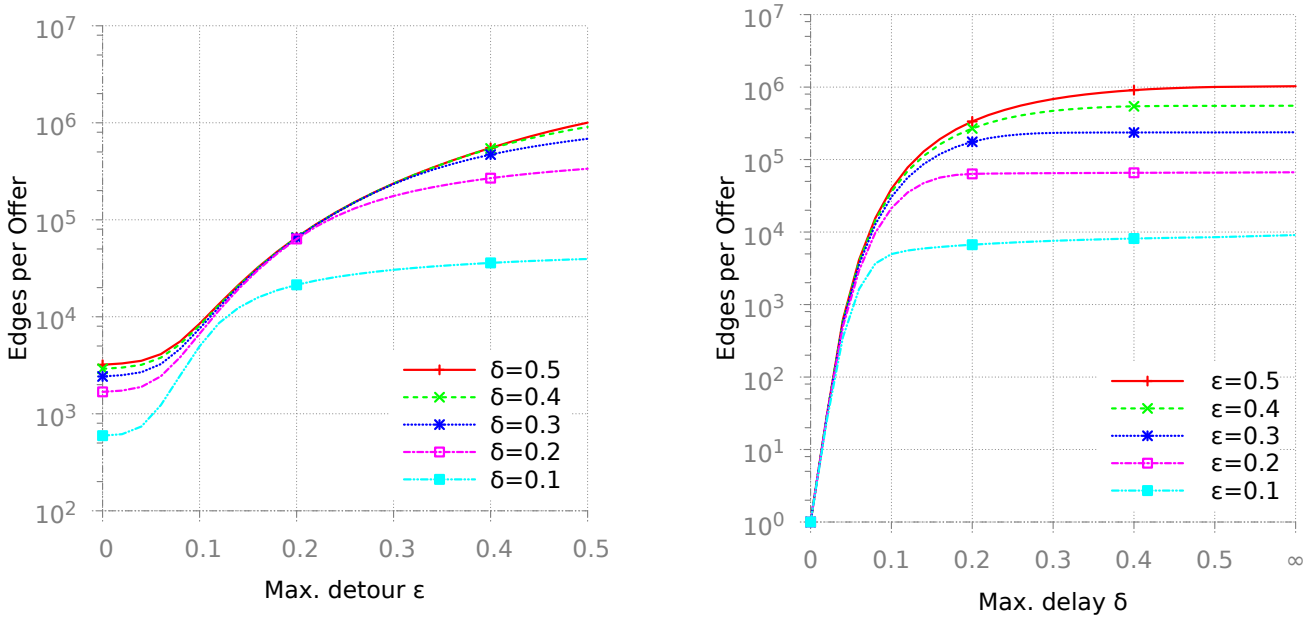

Figure 2: Number of Edges per Offer Depending on Tuning Parameters.

Second, we experiment on the matching rate against a given set of requests. We insert 10 000 random offers into the STEG. We vary the thresholds for allowed detour and delay, while we look into the impact of a threshold on the number of hops as well. Again, we assume that driver and rider have a common interest and share the same delay and detour factors. Figure 3 reports on these experiments. We observe that the matching rate depends on the maximum allowed delay as a larger threshold makes less attractive rides reasonable. The higher the threshold, the more edges are inserted into the STEG and finding a suitable path in a dense graph is more likely than in a sparse one. As expected, we see the best matching rates at $\delta = 0.5$. We see a steep increase in the fraction of matched rides when increasing from $\delta = 0.1$ to 0.2 and after that increases are not as significant.

On the right hand side of Figure 3 we fixed the delay to the reasonable value $\delta = 0.2$ and look at the matching rate depending on the number of allowed hops. Results for other values of δ are similar. We manage a hop counter that is increased only when sharing a ride. It is not increased for transfer edges. (Nodes are not inserted into the priority queue if their hop count exceeds $h \in \{1, 2, 3, 4, \infty\}$ in the respective experiment.) We see a significant increase in the matching rate when going from one hop to two hops, but we do not see any significant changes when further increasing the hop count. This means that searching for shared rides with three or more hops does not give significantly better results on average. We are surprised by the negligible improvements of more hops.

Third, we evaluate the performance of offer insertion as well as matching of queries. Removing an offer takes constant time as it just sets a flag in the list of offers and removes its edges in subsequent query operations, when they are encountered. We insert 10 000 randomly generated offers into an empty STEG and run the same amount of random queries against it without removing matched rides. Note that the space of the STEG has been preallocated to avoid costly resizes of the underlying basic data structures. The expected number of inserted edges per inserted offer is known from the previous experiment and we reserve 25% more space to be on the safe side. We implement Dijkstra's algorithm for comparison. Our implementation prunes the search such that it does not insert any nodes into the queue that violate the delay constraints by pruning rule 1. Our implementation

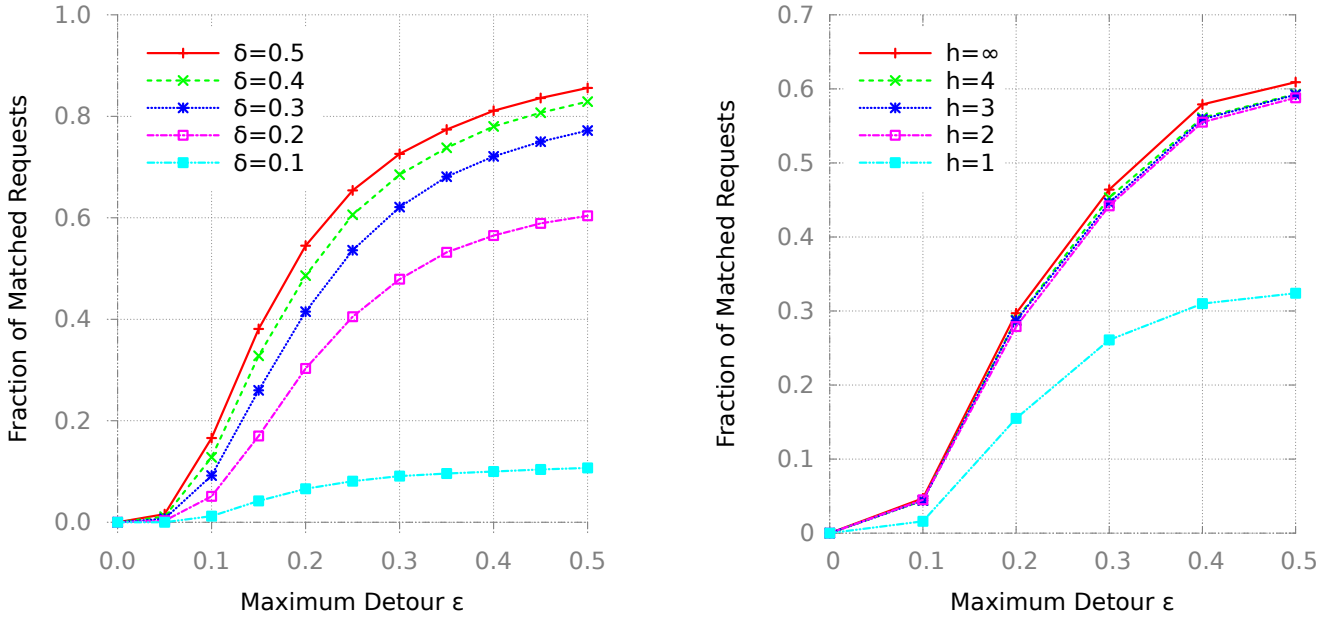

Figure 3: Matching Rates Depending on Tuning Parameters (left) and on Number of Hops for a Fixed Delay of $\delta = 0.2$ (right).

of A^* on the other hand uses a distance table of all stations to look up lower bounds for goal direction and also prunes the search when it reaches a node that can be proven as suboptimal by pruning rule 2.

Figure 4 reports on these experiments, where the query is run with A^*. On the left hand side of the figure, we observe that there is a significant difference between the results for $\delta = 0.1$ and $\delta = 0.2$ with respect to query time. The query time seems to be rather stable for lowest delay threshold independent of the detour, while the query time rises significantly for larger threshold values of detour and delay. But this is expected behavior as the search space of the shortest path query is small for $\delta = 0.1$. On the other hand, it is significantly larger for higher values of δ and we also see an amplification effect: The number of inserted edges is higher and the search is pruned at later distances. One effect adds to the other. On the right hand side of Figure 4, we see the time necessary to add an offer on average. We observe that the times are mostly independent of the maximum detour ε but that it depends on the value of δ. This is expected behavior the number of edges per offer depends on ε and most of the work that is done during insertion operations is to add edges into the graph.

Table 1 reports on the experiment that compares path searches with Dijkstra's algorithm to our A^* variant with pruning by lower bounds on the distance. As we preprocess the distance table beforehand, getting a lower bound for a distance to a target requires just one memory access. This single access is virtually for free during the search process compared to the overall number of memory accesses that are necessary. We observe speedups over Dijkstra's algorithm of 75–140. This is about an order of magnitude more than what we would expect from previous work, e.g. [Bauer, Delling, and Wagner, 2010; Delling, 2009] report factors of 7.0–8.5 when evaluating the performance of *uniALT* on a long distance timetable network. While it is not surprising to see a speedup of A^* over Dijkstra's algorithm, we note that it is significant. This is due to the good lower bounds that we achieve by preprocessing a large scale distance table on top of a road network. The observed speedups vary because the graph remains relatively sparse at first, i.e. $\delta = \varepsilon = 0.1$. Increasing to $\delta = \varepsilon = 0.2$ infers more than order of magnitude of additional scanned edges for our A^* variant. This is due to the amplification effect explained before. When further increasing the threshold value, the graph becomes even more dense and as a result the goal direction of our algorithm becomes more efficient. The query times of our search are at most just above half a second in all experiments, which can be arguably seen as fast enough for an online service in practice. This is especially true for tighter constraints. We see that the parameters give a convenient trade-off between query time, size of the data structure and matching rates.

Practical Considerations. Note that we expect users of such a system to attribute themselves to a number of nearby stations within their reach. Generally, fixed source and target stations are replaced by small sets of sources and targets that are close to the drivers real location(s). These sets are generated by one-to-many queries, e.g., to the next k stations, or more realistically by a multi-modal, multi-criteria one-to-many search, e.g., by adapting the algorithm of [Delling et al., 2013]. When generating sets $\mathcal{S}_1$ and $\mathcal{S}_2$ we adapt the pruning to include feasible stations and in the subsequent pairwise verification, we adapt the check if a pair is reasonable. For a rider, we adapt our path search by starting simultaneously from a set of sources and stop once a reason-

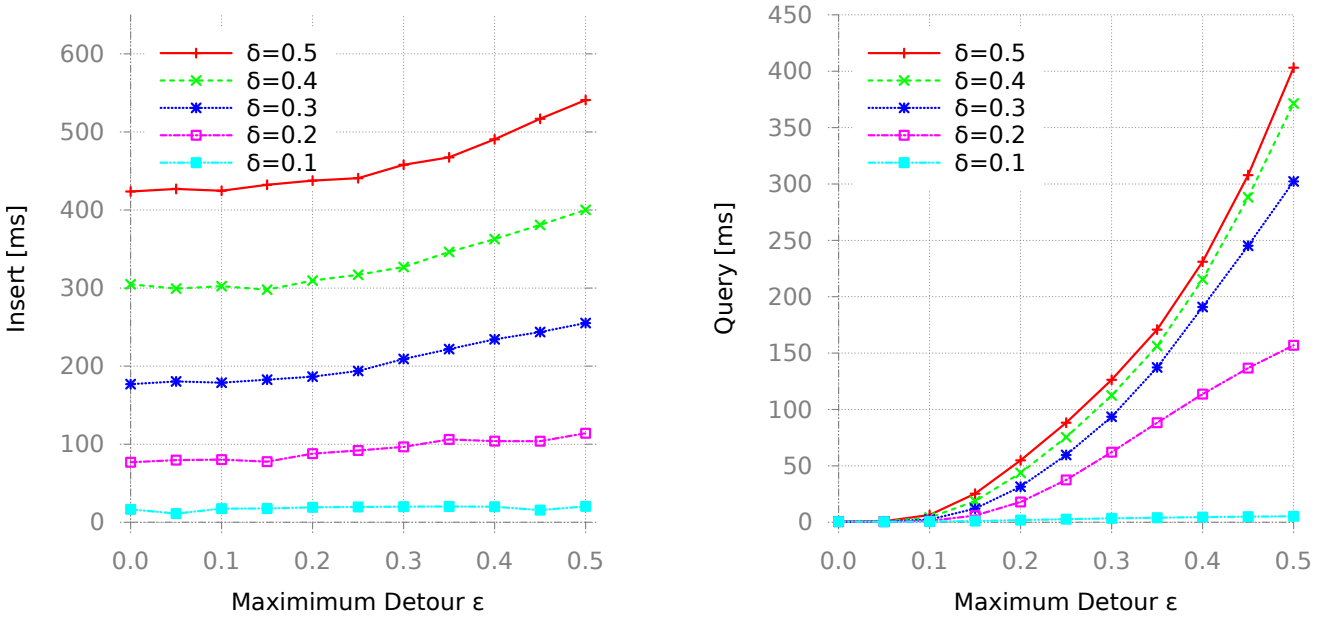

Figure 4: Time to Compute a Match (left) and Time to Add an Offer (right) Depending on Tuning Parameters.

able target station has been settled. The search is then started from these nodes simultaneously. The stopping criterion is adapted to stop the search once a *suitable* target station has been settled. Pruning the search is adapted likewise.

Time-dependency of the underlying road network can be handled as well. For example, the time-dependent variant of Contraction Hierarchies [Batz et al., 2013] is able to serve (nearly) as a drop-in replacement for the underlying speedup-technique. We note that edge weights between stations in the STEG must be computed with the time-dependent routing, but an approximated time-dependent distance table that only stores distances for the departure times, i.e. end of time-slots, is relatively easy to compute. This approximated and time-dependent distance table can then also serve to compute the lower bounds that are necessary for pruning the time-dependent path search.

It is also possible to model more than one rider per driver. In this case, a driver has a certain capacity that is associated with an offer. Once a rider joins in, we remove this offer from the system and (if capacity permits) add two new offers with adjusted thresholds for detour and delay, and with decreased capacity. The first offer resembles the path from the drivers source to the first pickup point, while the second offer resembles the ride from the drop-off location of the rider to the drivers target location. Note that the thresholds must be set relative to the already matched ride.

Mandatory waiting at a station is modelled for the driver by choosing a later target for inserted edges. It is easy to model such waiting times for riders by adapting the search.

Conclusion and Future Work

We presented an engineered graph search that solves the time-dependent multi-hop ride sharing problem with a fixed set of stations. The contribution is two-fold. First, we introduced a modelling of multi-hop ride sharing where users travel between a number of *stations* that is related to timetable networks for public transportation. Our approach allows a rider to reach a destination via a number of transfers. Second, we developed data structures and algorithms to efficiently compute matches. The application of Contraction Hierarchies to compute a distance table gives highly effective lower bounds to A^* search. This results in a speedup over Dijkstra's algorithm that is about an order of magnitude higher than what could be expected from previous work. A table with tight upper bounds to the distances can be used to further prune the search, but the impact, if any, needs to be evaluated in future work. It may proof useful when building on top of a time-dependent road network, though.

We evaluated the practicability of the algorithm with respect to the influence of tuning parameters and observe significant speedups as well as reasonable matching rates. These tuning parameters allow trade-offs between efficiency of insertion and query operations as well as the solution quality. One interesting result of our algorithm is that increasing the number of transfers to more than two does not lead to significantly superior results on average anymore. It will be interesting to see if this holds true for a substantially increased number of stations or not.

We would like to add more realistic modelling in the future. For example, we would like to explore an augmented multi-modal scenario that does not feature a single mode of transport but multiple ones, or even alternative routes [Luxen and Schieferdecker, 2012]. Additionally, we want to go beyond car sharing and integrate more sophisticated features such as car rental as well as public transportation, which are important means of transportation in metropolitan areas.

Acknowledgments

The authors would like to thank Peter Sanders and Dennis Schieferdecker for fruitful discussions on the subject at hand.

References

Abraham, I.; Delling, D.; Fiat, A.; Goldberg, A. V.; and Werneck, R. F. 2012. HLDB: Location-Based Services in Databases. In *ACM SIGSPATIAL Intern. Symp. on Adv. in Geographic Information Systems (GIS'12)*, 339–348. ACM.

Arz, J.; Luxen, D.; and Sanders, P. 2013. Transit Node Routing Reconsidered. In *International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*, 55–66. Springer.

Bast, H.; Funke, S.; and Matijevic, D. 2009. TRANSIT – Ultrafast Shortest-Path Queries with Linear-Time Preprocessing. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society.

Batz, G. V.; Geisberger, R.; Sanders, P.; and Vetter, C. 2013. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*. To appear.

Bauer, R.; Delling, D.; and Wagner, D. 2010. Experimental Study on Speed-Up Techniques for Timetable Information Systems. *Networks* 57:38–52.

Botea, A. 2011. Ultra-Fast Optimal Pathfinding without Runtime Search. In *Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'11)*, 122–127. AAAI.

Bundesministerium für Verkehr, B. u. S. 2008. Mobilität in Deutschland 2008. Technical report. http://www.mobilitaet-in-deutschland.de/pdf/MiD2008_Abschlussbericht_I.pdf.

Delling, D.; Dibbelt, J.; Pajor, T.; Wagner, D.; and Werneck, R. F. 2013. Computing Multimodal Journeys in Practice. In *International Symposium on Experimental Algorithms (SEA'13)*, LNCS. to appear.

Delling, D. 2009. *Engineering and Augmenting Route Planning Algorithms*. Ph.D. Dissertation, Universität Karlsruhe, Fakultät für Informatik.

Dijkstra, E. W. 1959. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* 1:269–271.

Geisberger, R.; Luxen, D.; Sanders, P.; Neubauer, S.; and Volker, L. 2010. Fast Detour Computation for Ride Sharing. In *Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10)*, volume 14 of *OASIcs*, 88–99.

Geisberger, R.; Sanders, P.; Schultes, D.; and Vetter, C. 2012. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transp. Sci.* 46(3):388–404.

Goldberg, A. V., and Harrelson, C. 2005. Computing the Shortest Path: A* Search Meets Graph Theory. In *ACM–SIAM Symp. on Discr. Algo. (SODA'05)*, 156–165. SIAM.

Hart, P. E.; Nilsson, N.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. on Sys. Sci. and Cyb.* 4:100–107.

Knopp, S.; Sanders, P.; Schultes, D.; Schulz, F.; and Wagner, D. 2007. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, 36–45. SIAM.

Luxen, D., and Schieferdecker, D. 2012. Candidate Sets for Alternative Routes in Road Networks. In *International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *LNCS*, 260–270. Springer.

Pyrga, E.; Schulz, F.; Wagner, D.; and Zaroliagis, C. 2004. Experimental Comparison of Shortest Path Approaches for Timetable Information. In *Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, 88–99. SIAM.

Vetter, C. 2009. Parallel Time-Dependent Contraction Hierarchies. Technical report, Karlsruhe Institute of Technology, Fakultät für Informatik. http://algo2.iti.kit.edu/download/vetter_sa.pdf.