Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns

Hannah Bast[1,2], Erik Carlsson[2], Arno Eigenwillig[2], Robert Geisberger[3,2], Chris Harrelson[2], Veselin Raychev[2], and Fabien Viger[2]

[1] Albert-Ludwigs-Universität Freiburg, 79110 Freiburg, Germany
[2] Google, Brandschenkestrasse 110, 8002 Zürich, Switzerland
[3] Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany

Abstract. We show how to route on very large public transportation networks (up to half a billion arcs) with average query times of a few milliseconds. We take into account many realistic features like: traffic days, walking between stations, queries between geographic locations instead of a source and a target station, and multi-criteria cost functions. Our algorithm is based on two key observations: (1) many shortest paths share the same *transfer pattern*, i.e., the sequence of stations where a change of vehicle occurs; (2) *direct connections* without change of vehicle can be looked up quickly. We precompute the respective data; in practice, this can be done in time linear in the network size, at the expense of a small fraction of non-optimal results. We have accelerated public transportation routing on Google Maps with a system based on our ideas. We report experimental results for three data sets of various kinds and sizes.

1 Introduction

In recent years, several algorithms have been developed that, after a precomputation, find shortest paths on the road network of a whole continent in a few microseconds, which is a million times faster than Dijkstra's algorithm. However, none of the tricks behind these algorithms yields similar speed-ups for public transportation networks of comparable sizes, especially when they are realistically modeled and show poor structure, like bus-only networks in big metropolitan areas. In this paper we present a new algorithm for routing on public transportation networks that is fast even when the network is realistically modeled, very large and poorly structured. These are the challenges faced by public transportation routing on Google Maps (`http://www.google.com/transit`), and our algorithm has successfully addressed them. It is based on the following new idea.

Think of the query $A@t \rightarrow B$, with source station $A =$ Freiburg, target station $B =$ Zürich, and departure time $t = 10{:}00$. Without assuming anything about the nature of the network and without any precomputation, we would have to do a Dijkstra-like search and explore many nodes to compute an optimal path. Now let us assume that we are given the following additional information: each and every optimal path from Freiburg to Zürich, no matter on which day and at which time of the day, either is a direct connection (with no transfer in between) or it is a trip with exactly one transfer at Basel. We call *Freiburg – Zürich* and *Freiburg – Basel – Zürich* the optimal *transfer patterns* between Freiburg and Zürich (for each optimal path, take the source station, the sequence of transfers, and the target station). Note how little information the set of optimal transfer patterns for this station pair is. Additionally, let us assume that we have timetable information for each station that allows us to very quickly determine the next *direct* connection from a given station to some other station.

With this information, it becomes very easy to answer the query $A@t \rightarrow B$ for an arbitrary given time t. Say $t =$ 10:00. Find the next direct connection from Freiburg to Zürich after t. Say it leaves Freiburg at 12:55 and arrives in Zürich at 14:52. (There are only few direct trains between these two stations over the day.) Also find the next direct connection from Freiburg to Basel after t. Say it leaves Freiburg at 10:02 and arrives in Basel at 10:47. Then find the next direct connection from Basel to Zürich after 10:47. Say it leaves Basel at 11:07 and arrives in Zürich at 12:00. In our cost model (see Section 3) these two connections are incomparable (one is faster, and the other has less transfers), and thus we would report both. Since the two given transfer patterns were the only optimal ones, we can be sure to have found all optimal connections. And we needed only three direct-connection queries to compute them.

Conceptually, our whole scheme goes as follows. The set of all optimal transfer patterns between all station pairs is too large to precompute and store. We therefore precompute a set of *parts of* transfer patterns such that all optimal transfer patterns can be combined from these parts. For our three datasets, we can precompute such parts in 20–40 core hours per 1 million departure/arrival events and store them in 10–50 MB per 1000 stations. From these parts, also non-optimal transfer patterns can be combined, but this only means additional work at query time; it will not let us miss any optimal connections. Think of storing parts of transfer patterns, to be recombined at query time, as a lossy compression of the set of all optimal transfer patterns. We also precompute a data structure for fast direct-connection queries, which, for our three datasets, needs 3–10 MB per 1 000 stations and has a query time of 2–10 μs.

Having this information precomputed, we then proceed as follows for a given query $A@t \rightarrow B$. From the precomputed parts, we compute all combinations that yield a transfer pattern between A and B. We overlay all these patterns to form what we call the *query graph*. Finding the optimal connection(s) amounts to a shortest-path computation on the query graph with source A and target B, where each arc evaluation is a direct-connection query. The query graph for our simple example from above has three nodes ($A =$ Freiburg, $B =$ Zürich, and $C =$ Basel) and three arcs (A $\rightarrow$ B, A $\rightarrow$ C, C $\rightarrow$ B). Due to the non-optimal transfer patterns that come from the recombination of the precomputed parts, our actual query graphs typically have several hundreds of arcs. However, since direct-connection queries can be computed in about 10 μs, this will still give us query times on the order of a few milliseconds, and by the way our approach works, these times are independent of the size of the network.

2 Related Work

The most successful "tricks of the trade" for fast routing on transportation networks can be summarized under the following five headings: *bidirectional search*, *exploiting hierarchy*, *graph contraction*, *goal direction*, and *distance tables.* The recent overview article [2] describes each of these and provides evidence and explanations why they give excellent speed-ups on road networks, but fail to do so on public transportation networks, especially such with poor structure. Two recent surveys on fast routing on road networks and on public transportation networks, respectively, are [5] and [10].

A fully realistic model like ours was recently considered in [6] and [3]. However, the network considered in those papers is relatively small (about 8900 stations) and very well-structured (German trains, almost no local transport). Also, there are only very few walking arcs, as walking between stations is rarely an issue for pure train networks. Reported query times are about one second for [6] and a few hundred milliseconds for [3]. The title of the latter paper aptly states that obtaining speed-ups for routing on public transportation networks in a realistic model "is harder than expected".

The best query times so far, of around 1 ms, were achieved in [4] and [7]. However, their model does support neither walking between stations, nor traffic days, nor multi-criteria costs, and, especially for the latter, it looks unlikely that their algorithms can be suitably extended. These two papers considered a graph of the European long-distance trains, and also two local networks (Berlin and Frankfurt), of size about 10 000 stations each.

Our algorithm is the first to yield fast query times (on the order of a few milliseconds) on a fully realistic model for public transportation networks also with poor structure (like bus-only networks) and of sizes more than an order of magnitude larger than what was considered so far.

3 Problem Formalization

A timetable describes the available trips of vehicles (buses, trains, ferries etc.) along sequences of stations (bus stops, train stations, ports etc.), including the times of day at which they depart and arrive. For routing, it is represented as a graph, see [11] for a comparison of various graph models. We use a *time-expanded graph* with three kinds of nodes, each carries a time and belongs to a station. For every *elementary connection* from station A to the next station B on the same trip, we put a *departure node* $A\mathrm{d}@t_1$ at A with the departure time t_1, an *arrival node* $B\mathrm{a}@t_2$ at B with the arrival time t_2 and an arc $A\mathrm{d}@t_1 \to B\mathrm{a}@t_2$ to model riding this vehicle from A to B. If the vehicle continues from B at time t_3, we put an arc $B\mathrm{a}@t_2 \to B\mathrm{d}@t_3$ that represents staying on the vehicle at B. This is possible no matter how small the difference $t_3 - t_2$ is.

Transfers at B shall be possible only to departures after a *minimum transfer duration* Δt_B. For each departure node $B\mathrm{d}@t$ we put a *transfer node* $B\mathrm{t}@t$ at the same time and an arc $B\mathrm{t}@t \to B\mathrm{d}@t$ between them. Also, we put an arc $B\mathrm{t}@t \to B\mathrm{t}@t'$ to the transfer node at B that comes next in the ascending order of departure times (with ties broken arbitrarily); these arcs form the *waiting chain* at B. Now, to allow a transfer after having reached $B\text{a}@t_2$, we put an arc to the first transfer node $B\text{t}@t$ with $t \geq t_2 + \Delta t_B$. This gives the opportunity to transfer to that and all later departures from B.

For exposition, we regard the graph as fully time-expanded, meaning times increase unbounded from time 0 (midnight of day 0). In practice, we exploit the periodicity of timetables by using times modulo 24 hours and bit masks to indicate a trip's traffic days. Also, we allow additional transfers by walking to nearby stations. See Section 6 for details. In our implementation, the resulting graphs can be stored in about 35 MB of memory per 1 million elementary connections.

Our scheme supports a fairly general class of multi-criteria cost functions and optimality notions. In our implementation, a cost is a pair (d, p) of non-negative duration and penalty. Duration of an arc is the difference in time between its endpoints. Penalty applies mostly to transfers: each station B defines a fixed penalty score for transferring, and that is the penalty component of the cost of arcs $B\text{a}@t \to B\text{t}@t'$. The arcs from departure to arrival nodes may be given a small penalty score for using that elementary connection. Other arcs, in particular waiting arcs, have penalty zero. The cost of a path in the graph is the component-wise sum of the costs of the arcs.

We say cost (d_1, p_1) *dominates* or *is better than* cost (d_2, p_2) *in the Pareto sense* if $d_1 \leq d_2$ and $p_1 \leq p_2$ and one of the inequalities is strict. Each finite set of costs has a unique subset of *(Pareto-)optimal* costs that are pairwise non-dominating but dominate any other cost in the set (in the Pareto sense).

Definition 1. *Consider a* station-to-station query *$A@t \to B$. Take the first transfer node $A\text{t}@t'$ with $t' \geq t$. For this query, we extend the graph by a source node S with an arc of duration $t' - t$ and penalty 0 that leads to $A\text{t}@t'$ and by a target node T with incoming arcs of zero cost from all arrival nodes of B.*

The paths from S to T are the feasible connections *for the query. If the cost of a feasible connection is not dominated by any other, we call them* optimal cost *and* optimal connection*, respectively, for the query.*

The query's result is an optimal set of connections*, that is, a set of optimal connections containing exactly one for each optimal cost.*

Note that the waiting chain at A makes paths from S through *all* departure nodes after time t feasible. We exclude multiple connections for the same optimal cost. They do occur (even for a single-criterion cost function) but add little value.[1]

4 Basic Algorithm

In this section we present a first simple algorithm that illustrates the key ideas. It has very fast query times but a quadratic precomputation complexity.

[1] Connections of equal cost, relative to query time t, arrive at the same time. It is preferable to choose one that departs as late as possible from A; we will return to that in Section 6. Those that depart latest often differ in trivial ways (e.g., using this or that tram between two train stations), so returning just one is fine.

4.1 Fast Direct-Connection Queries

Definition 2. *For a* direct-connection query $A@t \rightarrow B$*, the feasible connections are defined as in Definition 1, except that only* connections without transfers *are permitted. The result of the query are the optimal costs in this restricted set.*

The following data structure answers direct-connection queries in about $10\,\mu$s.

1. Precompute all *trips* (maximal paths in the graph without transfer nodes) and group them into *lines* L1, L2, ... such that all trips on a line share the same sequence of stations, do not overtake each other (FIFO property, like the *train routes* in [11]), and have the same penalty score between any two stations.
The trips of a line are sorted by time and stored in a 2D array like this:

line L17	S154	S097		S987		S111		...
trip 1	8:15	8:22	8:23	8:27	8:29	8:38	8:39	...
trip 2	9:14	9:21	9:22	9:28	9:28	9:37	9:38	...
...	...	...	...	...	...	...	...	...

2. For each station, precompute the sorted list of lines incident to it and its position(s) on each line. For example:
S097: (L8, 4) (L17, 2) (L34, 5) (L87, 17) ...
S111: (L9, 1) (L13, 5) (L17, 4) (L55, 16) ...

3. To answer a direct-connection query $A@t \rightarrow B$, intersect the incidence lists of A and B. For each occurrence of A before B on a line, read off the cost of the earliest feasible trip, then choose the optimal costs among all these.

In our example, the query S097@9:03 → S111 finds positions 2 and 4 on L17 and the trip that reaches S111 at 9:37.

Lemma 1. *A query $A@t \rightarrow B$ to the direct-connection data structure returns all optimal costs of direct connections.*

Proof. The straightforward proof can be found in the extended version [1].

4.2 Transfer patterns precomputation

Definition 3. *For any path, consider the subsequence of nodes formed by the first node, each arrival node whose successor is a transfer node, and the last node. The sequence of stations of these nodes is the* transfer pattern *of the path.*

An optimal set of transfer patterns *for a pair (A, B) of stations is a set S of transfer patterns such that for all queries $A@t \rightarrow B$ there is an optimal set of connections whose transfer patterns are contained in S, and such that each element in S is the transfer pattern of an optimal connection for a query $A@t \rightarrow B$ at some time t.*

For every source station A, we compute optimal sets of transfer patterns to all stations B reachable from it and store them in one DAG for A. This DAG has three different types of nodes: one *root node* labeled A, for each reachable station B a *target node* labeled B, and for each transfer pattern prefix

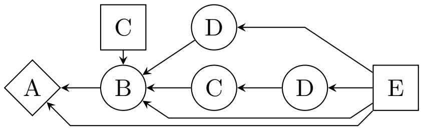

Fig. 1. DAG for the transfer patterns 'AE', 'ABE', 'ABC', 'ABDE' and 'ABCDE'. The root node is the diamond, prefix nodes are circles and target stations are rectangles. There are potentially several prefix nodes with the same label: In our example, 'D' occurs twice, the top one representing the prefix 'ABD' and the bottom one 'ABCD'.

$AC_1 \dots C_i$, occurring in at least one transfer pattern $AC_1 \dots C_i \dots B$, a *prefix node* labeled C_i. They are connected in the natural way such that precisely the transfer patterns we want to store are represented by a path from their target stations to the root node, labeled in reverse order; Figure 1 shows an example. We use the following algorithm *transfer_patterns(A)*.

1. Run a multi-criteria variant of Dijkstra's algorithm [9,13,8] starting from labels of cost zero at all *transfer* nodes of station A.

2. For every station B, choose optimal connections with the *arrival chain algorithm*: For all distinct arrival times $t_1 < t_2 < \dots$ at B, select a dominant subset in the set of labels consisting of (i) those settled at the arrival node(s) at time t_i and (ii) those selected for time t_{i-1}, with duration increased by $t_i - t_{i-1}$; ties to be broken in preference of (ii).

3. Trace back the paths of all labels selected in Step 2. Create the DAG of transfer patterns of these paths by traversing them from the source A.

Lemma 2. *If c is an optimal cost for the query $A@t_0 \to B$,* transfer_patterns(A) *computes the transfer pattern of a feasible connection for the query that realizes cost c.*

Proof. Let $c = (d, p)$. The label set for time $t_0 + d$ keeps a label with penalty p that departs at or after t_0. This needs that duration and penalty are optimized independently (i.e., Pareto-style). For details, see the extended version [1].

Running *transfer_patterns(A)* for all stations A is easy to parallelize by splitting the set of source stations A between machines, but even so, the total running time remains an issue. We can estimate it as follows. Let s be the number of stations, let n be the average number of nodes per station (< 569 for all our graphs, with the optimizations of Section 6), and let ℓ be the average number of settled labels per node and per run of *transfer_patterns(A)* (< 16 in all our experiments, in the setting of Sections 6 and 7). Then the total number of labels settled by *transfer_patterns(A)* for all stations A is $L = \ell \cdot n \cdot s^2$.

Steps 1–3 have running time essentially linear in L, with logarithmic factors for maintaining various sets. (For Step 1, this includes the priority queue of Dijkstra's algorithm. The bounded out-degree of our graphs bounds the number of unsettled labels linearly in L.) Since L is quadratic in s, this precomputation is infeasible in practice for large networks, despite parallelization. We will address this issue in Sections 5 and 7.

4.3 Query Graph Construction and Evaluation

For a query $A@t \to B$, we build the *query graph* as follows, independent of t:

1. Fetch the precomputed transfer patterns DAG for station A.
2. Search target node B in the DAG. Assume it has ℓ successor nodes with labels $C_1, \ldots, C_\ell$. Add the arcs $(C_1, B), \ldots, (C_\ell, B)$ to the query graph.
3. Recursively perform Step 2 for each successor node with a label $C_i \neq A$.

Figure 2 shows the query graph from A to E built from the DAG in Figure 1.

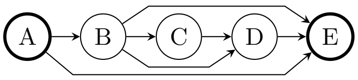

Fig. 2. Query graph A→ E from transfer patterns 'AE', 'ABE', 'ABDE' and 'ABCDE'.

Lemma 3. *For each transfer pattern $AC_1 \ldots C_k B$ in the DAG there exists the path $\langle A, C_1, \ldots, C_k, B\rangle$ in the constructed query graph.*

Proof. The straightforward proof can be found in the extended version [1].

Given the query graph, evaluating the query is simply a matter of a time-dependent multi-criteria Dijkstra search [6] on that graph. Labels in the queue are tuples of station and cost (time and penalty). Relaxing an arc $C \to D$ for a label with time t amounts to a direct-connection query $C@t \to D$.

By storing parent pointers from each label to its predecessor on the shortest path, we eventually obtain, for an optimal label at the target station, the sequence of transfers on an optimal path from the source to the target, as well as the times at which we arrive at each of these transfers. More details on the optimal paths can be provided by augmenting the direct-connection data structure.

Theorem 1. *For a given query $A@t \to B$, the described search on the query graph from A to B returns the set of optimal costs and for each such cost a corresponding path.*

Proof. We precomputed transfer patterns for each optimal cost of the query (Lemma 2) and the paths connecting these transfer stations are in our query graph (Lemma 3). From the correctly answered direct-connection queries (Lemma 1), the time-dependent Dijkstra algorithm on the query graph computes all optimal costs including matching paths.

5 Hub Stations

The preprocessing described in Section 4.2 uses quadratic time and produces a result of quadratic size. To reduce this, we do these expensive *global* searches only from a suitably preselected set of *hubs* and compute transfer patterns from hubs to all other stations.[2] For all non-hub stations, we do *local* searches computing

[2] Computing transfer patterns only to other hubs is not faster.

only those transfer patterns without hubs or their parts up to the first hub. More precisely, let $AC_1 \ldots C_k B$ be a transfer pattern from a non-hub A that we would have stored in Section 4.2. If any of the C_i is a hub, we do not store this pattern any more. The hub C_i with minimal i is called an *access station* of A. We still store transfer patterns $A \ldots C_i$ and $C_i \ldots B$ into and out of the access station. This shrinks transfer patterns enough to allow query processing entirely from main memory on a single machine, even for large networks (see Section 8).[3]

Selecting the hubs. We create a time-independent graph by overlaying the nodes and arcs of each line (as computed in Section 4.1), using the minimum of arc costs. Then, we perform cost-limited Dijkstra searches from a random sample of source stations. The stations being on the largest number of shortest paths are chosen as hubs.[4]

Transfer patterns computation. The *global* search remains as described in Section 4.2. The *local* search additionally marks labels stemming from labels at transfer nodes of hubs as *inactive*, and stops as soon as all unsettled labels are inactive [12]. Inactive labels are ignored when the transfer patterns are read off.[5]

Query graph. Processing a query $A@t \to B$ looks up the set $\mathcal{X}$ of access stations of A and constructs the query graph from the transfer patterns between the station pairs $\{(A,B)\} \cup (\{A\} \times \mathcal{X}) \cup (\mathcal{X} \times \{B\})$. The evaluation of the query graph remains unchanged.

Lemma 4. *If c is an optimal cost for the query $A@t_0 \to B$, then the query graph from A to B contains the transfer pattern of a feasible connection for the query that realizes cost c.*

Proof. If no suitable transfer pattern $A \ldots B$ was computed, then A is a non-hub with an access station X such that a suitable transfer pattern has been computed in two parts $A \ldots X$ and $X \ldots B$, as we show in the extended version [1]. This needs that duration and penalty are optimized independently (i.e., Pareto-style).

6 Further Refinements

Above, we simplified the presentation of our algorithm. Our actual implementation includes the following refinements.

Location-to-location. Our implementation answers location-to-location queries. In the model from Definition 1, the source and target node now stand for the source and target locations, and they are connected to a selection of source and

[3] The number of global searches could be reduced further by introducing several levels of hubs, but in our implementation the total cost for the global searches is below the total cost for the local searches already with one level of hubs; see Section 8.

[4] We experimented with a variety of hub selection strategies, but they showed only little difference with respect to preprocessing time and query graph sizes, and so we stuck with the simplest strategy.

[5] Before that, inactive labels are needed to dominate non-optimal paths around hubs.

target stations nearby with arcs that take the walking cost into consideration. The query graph is built from transfer patterns for all pairs of source and target stations. This feature is of high practical value for dense metropolitan networks.

Walking arcs for transfers. Likewise, transfers by walking from stations to nearby stations are very important in metropolitan networks. We add arcs from arrival nodes to transfer nodes of useful nearby stations whose costs reflect the additional walking. This results in about twice more arcs in the graph and duplicates certain entities in the algorithm: transfer patterns now alternate between riding a vehicle and walking, the query graph requires two nodes per station, and two global searches from each hub are necessary, as the hub can either be the arrival or departure station of a transfer.

More compact graph model. In the precomputation, we optimize the representation of the graph from Section 3 in two ways. Departure nodes are removed and their predecessors (transfer node and maybe arrival node) are linked directly to their successor (the next arrival node), cf. [11, §8.1.2]. To exploit the periodicity of timetables, we *roll up* the graph modulo one day, that is, we label nodes with times modulo 24 hours and use bit masks to indicate each trip's traffic days.

Query graph search. After we have determined the earliest arrival time at the target station, we execute a backward search to find the optimal connection that *departs latest*, see footnote 1 on page 293. Furthermore, we apply the A^* heuristic to goal-direct the searches, using minimal costs between station pairs (computed along with the direct-connection data structure) as lower bounds.

7 Heuristic Optimizations

The system described so far gives exact results, that is, for each query we get an optimal connection for every optimal cost. However, despite the use of hubs (Section 5), the precomputation is not significantly faster than the quadratic precomputation described in Section 4. The reason is that, although the results of the local searches (the local transfer patterns) are reasonably small, almost every local search has a local path of very large cost and hence has to visit a large portion of the whole network before it can stop. Indeed, this *15 hours to the nearby village problem* is at the core of what makes routing in public transportation networks so hard [2].

The good news is that with our transfer patterns approach we don't have this problem at query time but only in the precomputation. Note here that our approach is unique in that it precomputes information for *all* queries, not just for queries where source and target are sufficiently "far apart". The bad news is that, despite intensive thought, we did not find a solution that is both fast and exact. We eventually resorted to the following simple but approximate solution: limit the local searches to at most two transfers, that is, using at most three vehicles. We call this the *3-legs heuristic*, and as we will see in Section 8, it indeed makes the local searches reasonably fast. Theoretically, we may now miss some optimal transfer patterns, but we found this to play no role in the practical use of our algorithm. For example, on our CH graph (Section 8), on 10 000 random queries the 3-leg heuristic gave only three non-optimal results, and all three of these were only a few percent off the optimum. We remark that errors in the input data are a much bigger issue in practice. More details on the quality of our approximation are given in the extended version [1].

Having accepted a small fraction of non-optimal results, we also developed and apply various other heuristics, which may lead to a non-optimal solution at query time, but whose measured effect in practice is again tolerable. The following heuristics in combination speed up our query times by a factor of 3–5.

1. In local searches, mark labels as inactive that travel through hubs without transfer beyond a distance threshold. (Requires fixup in query graph building.)
2. Do only one global search per hub, starting at transfer *and* arrival nodes.
3. Optimize duration relaxed by penalty [10], thus discarding Pareto-optimal trips whose improvement in penalty is small in relation to the longer duration.
4. Drop rare transfer patterns if optima on other patterns are almost as good.

In precomputation time, these additional heuristics (esp. reducing the number of labels with 3.) save roughly another factor of 2. Unlike the 3-leg heuristic, they are not essential for the feasibility of our approach.

8 Experiments

The experimental results we provide in this section are for a fully-fledged C++ implementation, with all the refinements from Section 6 and all the tricks from Section 7 included. For precomputation, our experiments were run on a compute cluster of Opteron and Xeon-based 64-bit servers. Queries were answered by a single machine of the cluster, with all data in main memory.

Graphs. We ran our experiments on three different graphs: the train + local transport network of most of Switzerland (CH), the complete transport network of the larger New York area (NY), and the train + local transport network of much of North America (NA). Table 1 summarizes the different sizes and types.

Table 1. The three public transportation graphs from our experiments

name	#stations	#nodes	#arcs	space	type
CH	20.6 K	3.5 M	11.9 M	64 MB	trains + local, well-structured
NY	29.4 K	16.7 M	79.8 M	301 MB	mostly local, poor structure
NA	338.1 K	113.2 M	449.1 M	2 038 MB	trains + local, poor structure

Direct-connection queries. Table 2 shows that the preprocessing time for the direct-connection data structure is negligible compared to the transfer patterns precomputation time. The space requirement is from 3 MB per 1000 stations for CH to 10 MB per 1000 stations for NY and NA. A query takes from 2 μs for CH to around 10 μs for NY and NA. Note that the larger direct-connection query time for NY and NA is a yardstick for their poor structure (not for their size).

Table 2. Direct-connection data structure: construction time and size. The query time range is from getting the fastest to all Pareto-optimal connections.

name	precomp. time	output size	query time
CH	< 1 min	68 MB	2 μs
NY	4 min	335 MB	5–9 μs
NA	49 min	3 399 MB	9–14 μs

Transfer patterns precomputation. Our precomputation time (Table 3) is 20–40 (CPU core) hours per 1 million nodes and the resulting (parts of) transfer patterns can be stored in 10–50 MB per 1000 stations. Again, these ratios depend mostly on the structure of the network (best for CH, worst for NY and NA), and not on its size.

Table 3. Transfer patterns precomputation times and results

name	precomp. time		output size		#TP/station pair	
	local	global	local	global	local	global
CH (w/o hubs)	–	635 h	–	18 562 MB	–	11.0
CH (w/ hubs)	562 h	24 h	229 MB	590 MB	2.6	25.8
CH (heuristic)	57 h	4 h	60 MB	154 MB	2.0	6.8
NY (heuristic)	724 h	64 h	787 MB	786 MB	3.7	16.4
NA (heuristic)	2 632 h	571 h	6 849 MB	7 151 MB	3.4	10.5

Query graph construction and evaluation. Table 4 shows that, on average, query graph construction and evaluation take 5 μs and 15 μs per arc, respectively. The typical number of arcs in a query graph for a station-to-station query (1:1) is below 1000 and the typical query time is below 10 ms. Location-to-location queries with 50 source and 50 target stations (50:50) take about 50 ms.

Table 4. Average query graph construction time, size, and evaluation time. The third column also provides the median, 90%-ile and 99%-ile. Column 'D' is the domination, either **P**areto or **S**ingle-criterion (sum of duration and penalty).

name			constr.	#arcs (50/mean/90/99)				D	eval.	#arc ev.
CH	w/o hubs	1:1	< 1 ms	32	34	56	86	P	< 1 ms	89
CH	w/ hubs	1:1	1 ms	189	264	569	1286	P	3 ms	540
CH	heuristic	1:1	< 1 ms	80	102	184	560	P	< 1 ms	194
NY	heuristic	1:1	2 ms	433	741	1 917	3 597	P	6 ms	721
NY	heuristic	1:1	2 ms	433	741	1 917	3 597	S	3 ms	248
NY	heuristic	50:50	32 ms	3 214	6 060	15 878	35 382	S	18 ms	1 413
NA	heuristic	1:1	2 ms	261	536	1 277	3 934	P	10 ms	705
NA	heuristic	1:1	2 ms	261	536	1 277	3 934	S	5 ms	321
NA	heuristic	50:50	22 ms	2 005	3 484	7 240	25 775	S	21 ms	1 596

9 Conclusions

We believe that the transfer patterns idea has great potential, and we have shown some of its potential in this paper. Obvious directions for future research are: (1) get exact local searches with a feasible precomputation time; (2) make the precomputation faster; (3) reduce the size of the query graphs; (4) speed up the (already very fast) direct-connection queries. Transfer patterns are, by their very nature, also well-suited for so-called *profile queries* (compute all paths from A to B over a large time window). We want to explore this potential further.

Acknowledgements. The authors acknowledge inspiring conversations with Alex Hall as well as helpful comments by Alex Gontmakher and an anonymous reviewer of the conference submission.

References

1. Extended version of this paper, `http://ad.informatik.uni-freiburg.de/papers`
2. Bast, H.: Car or public transport – two worlds. In: Albers, S., Alt, H., Näher, S. (eds.) Efficient Algorithms. LNCS, vol. 5760, pp. 355–367. Springer, Heidelberg (2009)
3. Berger, A., Delling, D., Gebhardt, A., Müller-Hannemann, M.: Accelerating time-dependent multi-criteria timetable information is harder than expected. In: ATMOS 2009. Dagstuhl Seminar Proceedings (2009)
4. Delling, D.: Time-dependent SHARC-routing. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 332–343. Springer, Heidelberg (2008)
5. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering route planning algorithms. In: Lerner, J., Wagner, D., Zweig, K.A. (eds.) Algorithmics of Large and Complex Networks. LNCS, vol. 5515, pp. 117–139. Springer, Heidelberg (2009)
6. Disser, Y., Müller-Hannemann, M., Schnee, M.: Multi-criteria shortest paths in time-dependent train networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 347–361. Springer, Heidelberg (2008)
7. Geisberger, R.: Contraction of timetable networks with realistic transfers. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 71–82. Springer, Heidelberg (2010)
8. Loui, R.P.: Optimal paths in graphs with stochastic or multidimensional weights. Commun. ACM 26, 670–676 (1983)
9. Möhring, R.H.: Verteilte Verbindungssuche im öffentlichen Personenverkehr: Graphentheoretische Modelle und Algorithmen. In: Horster, P. (ed.) Angewandte Mathematik – insbesondere Informatik, pp. 192–220. Vieweg (1999)
10. Müller-Hannemann, M., Schulz, F., Wagner, D., Zaroliagis, C.: Timetable information: Models and algorithms. In: Geraets, F., Kroon, L.G., Schoebel, A., Wagner, D., Zaroliagis, C.D. (eds.) Railway Optimization 2004. LNCS, vol. 4359, pp. 67–90. Springer, Heidelberg (2007)
11. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.D.: Efficient models for timetable information in public transportation systems. J. Exp. Algorithmics 12 (2007)
12. Schultes, D., Sanders, P.: Dynamic highway-node routing. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 66–79. Springer, Heidelberg (2007)
13. Theune, D.: Robuste und effiziente Methoden zur Lösung von Wegproblemen. Teubner (1995)