Contraction of Timetable Networks with Realistic Transfers

Robert Geisberger
Universität Karlsruhe (TH), 76128 Karlsruhe, Germany, geisberger@ira.uka.de

December 1, 2018

Abstract

We successfully contract timetable networks with realistic transfer times. Contraction gradually removes nodes from the graph and adds shortcuts to preserve shortest paths. This reduces query times to 1 ms with preprocessing times around 6 minutes on all tested instances. We achieve this by an improved contraction algorithm and by using a station graph model. Every node in our graph has a one-to-one correspondence to a station and every edge has an assigned collection of connections. Our graph model does not need parallel edges. The query algorithm does not compute a single earliest arrival time at a station but a set of arriving connections that allow best transfer opportunities.

1 Introduction

Compared to road networks, query algorithms for timetable information in public transportation systems are still slow. As multi-modal routing becomes more and more important, it is necessary to develop speedup techniques that work well. On the one hand, previous research focused on modelling more and more features instead of fast algorithms. On the other hand, current speedup techniques for time-dependent routing in road networks are very fast, e.g. [5, 1], but they have some problems to process transportation networks. So more research on fast timetable routing is necessary. We successfully adapt a fast speedup technique for road networks. The positive outcome is mainly due to the station graph model that looks much more natural and lacks a lot of problems that other models have.

Related Work

Public transportation networks have always been time-dependent, i.e. travel times depend on the availability of trains, buses or other vehicles. So they are naturally harder than road networks, where simple models can be independent of the travel time and still achieve good results. There are two intensively investigated approaches for modeling timetable information: the *time-expanded* [11, 9, 16, 17], and the so called *time-dependent* approach [4, 12, 13, 14]. Note that the time-dependent approach is a special approach to model the time-dependent information and is no umbrella term for all these approaches. Both approaches answer queries by applying some shortest-path algorithm to a suitably constructed graph. In the time-expanded approach, each node corresponds to a specific time event (departure or arrival), and each edge has a constant travel time. In the time-dependent approach each node corresponds to a station, and the costs on an edge are assigned depending on the time in which the particular edge will be used by the shortest-path algorithm.

To model more realistic transfers in the time-dependent approach, [4] propose to model each platform as a separate station and add walking links between them. In [15], a similar extension for constant and variable transfer times is proposed and described in more detail. Basically, a station is expanded to a *train-route graph* where no one-to-one correspondence between nodes and stations exists anymore. A *train route* is the maximal subset of trains that follow the exact same route, at possibly different times and do not overtake each other. Each train route has its own node at each station and they are interconnected within a station with the given transfer times. This results in a significant blowup in the number of nodes and creates

a lot of redundancy information that is collected during a query. Recently, another research group [3, 2] independently proposed a model that is similar to ours. They call it the *station graph* model and mainly use it to compute all Pareto-optimal paths in a fully realistic scenario. For unification, we will give our model the same name although there are some important differences in the details. The most significant differences are that (1) they require parallel edges, one for each train route and (2) their query algorithm computes a label for each edge instead of each node. Their improvement over the time-dependent model was mainly that they compare all labels at a station and remove dominated ones.

Speedup techniques are very successful when it comes to routing in time-dependent road networks, see [6] for an overview. However, there is only little work on speedup techniques for timetable networks. Time-dependent SHARC [6] uses the same scenario as we do and achieves query times of 2.4 ms but with preprocessing times of more than 6 hours (we scaled by a factor of 0.5 compared to [6] based on plain Dijkstra performance of our and their hardware). Based on the station graph model, [2] also applied some speedup techniques, namely arc flags that are valid on time periods and route contraction. They could not use node contraction because they were too many parallel edges between stations. Their preprocessing time (not scaled due to lack of comparable figures) is over 33 CPU hours resulting in a full day profile query time of more than 1 second (speedup factor 5.2).

We will show that a modified version contraction hierarchies (CH) [1, 8], a very successful speedup technique for road networks, will work in our scenario with realistic transfers. It is solely based on node contraction: removing "unimportant" nodes and adding shortcuts to preserve shortest path distances. A bidirectional query can then find shortest paths looking only at a few hundred nodes.

2 Formal Description

We propose a model that is similar to the realistic time-dependent model introduced by [15] but we keep a one-to-one mapping between nodes in the graph and real stations.

A *timetable* consists of data concerning: *stations* (or bus stops, ports, etc), *trains* (or buses, ferries, etc), connecting stations, *departure* and *arrival times* of trains at stations, and *traffic days*. More formally, we are given a set of *stop events* $\mathcal{Z}$, a set of stations $\mathcal{B}$, and a set of *elementary connections* $\mathcal{C}$, whose elements c are 6-tuples of the form $c = (Z_1, Z_2, S_1, S_2, t_d, t_a)$. Such a tuple (elementary connection) is interpreted as train that leaves station S_1 at time t_d after stop Z_1 and the *immediately next* stop is Z_2 at station S_2 at time t_a. If x denotes a tuple's field, then the notation of $x(c)$ specifies the value of x in the elementary connection c. Two consecutive elementary connections, c_1 followed by c_2, where no transfer is required, share a stop event, i.e. $Z_2(c_1) = Z_1(c_2)$. A stop event can not only be a consecutive arrival and departure of a train, but also the begin (no arrival) or the end (no departure) of a train.

The *departure* and *arrival times* $t_d(c)$ and $t_a(c)$ of an elementary connection $c \in \mathcal{C}$ within a day are integers in the interval $[0, 1439]$ representing time in minutes after midnight. Given two time values t and t', $t \leq t'$, the *cycle difference*(t, t') is the smallest nonnegative integer ℓ such that $\ell \equiv t' - t \pmod{1440}$. The *length* of an elementary connection c, denoted by *length*(c), is *cycle difference*$(t_d(c), t_a(c))$. A timetable is valid for a number of N *traffic days*, and every train is assigned a bit-field of N bits determining of which traffic days the train operates (for overnight trains the departure of the first elementary connection counts). We will generally assume that trains operate daily. At a station $S \in \mathcal{B}$, it is possible to *transfer* from one train to another. Such a transfer is only possible if the time between the arrival and the departure at the station S is larger than or equal to a given, station-specific, *minimum transfer time*, denoted by *transfer*(S).

Example 1 *{(1, 2, A, B, 23:05, 0:55), (2, 3, B, C, 1:02, 2:57), (3, 4, C, D, 3:00,4:20)} describe elementary connections of a train from station A (stop event 1) via stations B (stop event 2), C (stop event 3) to station D (stop event 4) as shown in Figure 1. E.g. the train departs at station A at 23:05 (hh:mm) and arrives at station B at 0:55 at the next day. The length of this elementary connection is 1:50 = 110 minutes. {(5, 6, C, E, 3:00, 4:00), (7, 8, C, E, 4:00, 5:00)} describe elementary connections of two trains from station C to E, the first train departs at 3:00 and arrives at 4:00, the second train one hour later.*

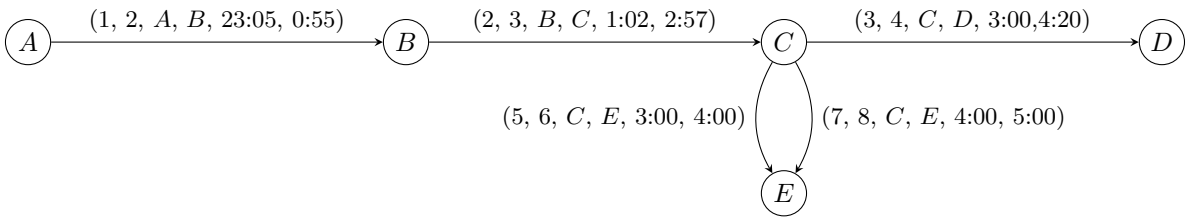

Figure 1: Every node is a station and every edge an elementary connection.

Let $P = (c_1, \ldots, c_k)$ be a sequence of elementary connections together with departure times $dep_i(P)$ and arrival times $arr_i(P)$ for each elementary connection c_i, $1 \leq i \leq k$. We assume that the times $dep_i(P)$ and $arr_i(P)$ include data regarding also the departure/arrival day by counting time in minutes from the first day of the timetable. Such a time t is of the form $t = a1440 + b$, where $a \in [0, N-1]$ and $b \in [0, 1439]$. Hence, the actual time within a day is t (mod 1440) and the actual day is $\lfloor t/1440 \rfloor$. Such a sequence P is called a *consistent connection* from station $A = S_1(c_1)$ to station $B = S_2(c_k)$ if it fulfills some consistency conditions: (a) the departure station of c_{i+1} is the arrival station of c_i; (b) the time values $dep_i(P)$ and $arr_i(P)$ correspond to the time values t_d and t_a, resp., of the elementary connections (modulo 1440) and respect the transfer times at stations. More formally, P is a *consistent connection* if the following conditions are satisfied:

$$
\begin{aligned}
c_i & \quad \text{is valid on day } \lfloor dep_i(P)/1440 \rfloor \\
S_2(c_i) &= S_1(c_{i+1}) \\
dep_i(P) &\equiv t_d(c_i) \pmod{1440} \\
arr_i(P) &= length(c_i) + dep_i(P) \\
dep_{i+1}(P) - arr_i(P) &\geq \begin{cases} 0 & \text{if } Z_1(c_{i+1}) = Z_2(c_i) \\ transfer(S_2(c_i)) & \text{otherwise} \end{cases}
\end{aligned}
$$

Example 2

c_i	$(Z_1, Z_2, S_1, S_2, t_d, t_a)$	dep_i	arr_i
c_1	*(1, 2, A, B, 23:05, 0:55)*	*23:05*	*24:55*
c_2	*(2, 3, B, C, 1:02, 2:57)*	*25:02*	*26:57*
c_3	*(7, 8, C, E, 4:00, 5:00)*	*28:00*	*29:00*

$P = (c_1, c_2, c_3)$ is a consistent connection with one transfer. The elementary connections are from Example 1. Assume a transfer time at station C of $transfer(C) = 5$ minutes. It would not be consistent to replace c_3 with the train that arrives at $arr_3(P) = 28{:}00$ since there are only $3 < 5 = transfer(C)$ minutes between the arrival and the departure at station C.

2.1 Time Query

A time query (A, B, t_0) consists of a departure station A, an arrival station B and a departure time t_0 (including the departure day). Connections are *valid* if they depart not before the given departure time t_0. A time query solves the earliest arrival problem (EAP) by minimizing the difference between the arrival time and the given departure time.

2.2 Profile Query

A profile query (A, B) consists of a departure station A and an arrival station B. It computes a dominant set of all consistent connections between A and B.

3 Station Graph Model

We introduce a model that represents a timetable as a digraph with exactly one node per station. For a simplified model without transfer times, this is like the time-dependent model. New is that even with positive transfer times, we keep one node per station and do not have parallel edges. We got the idea while trying to apply node contraction to time-dependent networks. Our observation was, that too many shortcuts where added, and there were a lot of edges between the nodes of the same station pair. In a first step, we tried to reduce the number of nodes by merging route nodes. We can do this when it is never necessary to transfer between the two routes at this station. However, we were not successful with this step and eventually came up with the station graph model.

The attribute of an edge $e = (A, B)$ is a set of consistent connections $fn(e)$ that depart in A and arrive in B. In this section, all connections are consistent, so we omit to mention it again. Previous models required that all connections of a single edge fulfill the FIFO-property, i.e. they do not overtake each other. In contrast, we do not require this property and we will see that even for time queries, we can have more than one dominant arrival event per station.

To link two edges $e_1 = (A, B)$ and $e_2 = (B, C)$ to an edge $e_3 = (A, C)$ we need to pairwise link the connections in $fn(e_1)$ and $fn(e_2)$. We need to drop the new connections that are not consistent. Also some other new connections might not be necessary if they can always be replaced by other new connections without worsening the objective value. We call such replaceable connections *dominated* in this set of connections and all other connections *dominant*. Note that the exact definition of domination depends on the objective function.

3.1 Time Query

We solve the EAP with a label correcting algorithm based on the Dijkstra algorithm. A label is a *connection* P stored as a tuple (Z_1, Z_2, dep, arr) where Z_1 is the stop event for departure, Z_2 is the stop event for arrival and dep and arr are the departure/arrival time including days. The source station $S_1(P)$ and the target station $S_2(P)$ are implicitly given by the node/edge that stores this connection.

Not all valid connections are relevant for a time query. We need not to store connections that can be dominated (replaced) by another stored connection. Before we can specify when two connections dominate each other, we need some more definitions. Let P be a connection. Define $parr(P)$ as the previous arrival time of the stop event $Z_1(P)$ at station $S_1(P)$ or $\perp$ when the train begins there. And define $ndep(P)$ as the next departure time of the stop event $Z_2(P)$ at station $S_2(P)$ or $\perp$ when the train ends there. When $parr(P) \neq \perp$ then we call $res_d(P) := dep(P) - parr(P)$ the *residence time at departure*. Consequently, when $ndep(P) \neq \perp$ then we call $res_a(P) := ndep(P) - arr(P)$ the *residence time at arrival*. We call it a *critical departure* when $parr(P) \neq \perp$ and $res_d(P) < transfer(S_1(P))$ and a *critical arrival* when $ndep(P) \neq \perp$ and $res_a(P) < transfer(S_2(P))$.

A connection P dominates a connection Q iff all of the following conditions are fulfilled:

(1) $S_1(P) = S_1(Q)$ and $S_2(P) = S_2(Q)$

(2) $dep(Q) \leq dep(P)$ and $arr(P) \leq arr(Q)$

(3) $Z_1(Q) = Z_1(P)$ or Q is not a critical departure or $dep(P) - parr(Q) \geq transfer(S_1(P))$

(4) $Z_2(Q) = Z_2(P)$ or Q is not a critical arrival or $ndep(Q) - arr(P) \geq transfer(S_2(P))$

We could see the query algorithm also a multi-criteria shortest path problem with relaxed Pareto dominance [10]. The length of a connection is the cost criterion (2), but it is relaxed by the additional train and transfer information (3),(4).

Given an connection $R = (c_1, \ldots, c_k)$, we call a connection $(c_1, \ldots, c_i)$ with $1 \leq i \leq k$ a *prefix* of R, a connection $(c_j, \ldots, c_k)$ with $1 \leq j \leq k$ a *suffix* of R and a connection $(c_i, \ldots, c_j)$ with $1 \leq i \leq j \leq k$ a *subconnection* of R.

Lemma 1 formalizes the intuition of the domination relation.

Lemma 1 *A consistent connection P dominates a consistent connection Q iff for all consistent connections R with subconnection Q, we can replace Q by P to get a consistent connection R'. Also $dep(R) \leq dep(R') \leq arr(R') \leq arr(R)$.*

Proof. $\Rightarrow$ P dominates Q: Condition (1) locates P and Q at the same stations. Condition (2) ensures that we can replace $R = Q$ by P directly or when transfer times are irrelevant. The last two conditions (3) and (4) ensure that we can replace Q by P even when Q is just a part of a bigger connection and we need to consider transfer times. The prefix of this bigger connection w.r.t. Q may arrive in $S_1(Q)$ with stop event $Z_1(Q)$ and condition (3) ensures that it is consistent to transfer to $Z_1(P)$. Consequently the suffix of this bigger connection w.r.t. Q may depart in $S_2(Q)$ with stop event $Z_2(Q)$ and condition (4) ensures that it is consistent to transfer to $Z_2(P)$.

$\Leftarrow$ $\forall R$ we can replace Q by P: Condition (1) holds trivially. We get condition (2) with $R = Q$. Let $Q = (c_1, \ldots, c_k)$ be a critical departure and $Z_1(Q) \neq Z_1(P)$. Then we get condition (3) when we choose R as the extension of Q that is given by the critical departure. Let $Q = (c_1, \ldots, c_k)$ be a critical arrival and $Z_2(Q) \neq Z_2(P)$. Then we get condition (4) when we choose R as the extension of Q that is given by the critical arrival. □

With the given model of a connection, we can model waiting times implicitly as the time between the arrival with one train and the departure of the next train. However initial waiting as it can appear for a time query (A, B, t_0) is currently not possible. So we introduce an *arrival connection* P that is represented by a tuple (arr, Z_2) and defines a connection from A to $S_2(P)$. We perform the query as a label correcting algorithm and store a dominant set of arrival connections with a station so that $S_2(P)$ is implicitly given. An arrival connection is called *consistent* w.r.t. the query if it is a consistent connection Q with $dep_1(Q) \geq t_0$ and $S_1(Q) = A$. All arrival connections in this section are consistent w.r.t. the query so we do not mention it again. Linking of a set of arrival connections at station C with an edge (C, D) will result in a set of arrival connections at station D.

An arrival connection P dominates an arrival connection Q iff all of the following conditions are fulfilled:

(1) $S_2(P) = S_2(Q)$

(2) $arr(P) \leq arr(Q)$

(3) $Z_2(Q) = Z_2(P)$ or Q is not a critical arrival or $ndep(Q) - arr(P) \geq transfer(S_2(P))$

A result of Lemma 1 is Lemma 2.

Lemma 2 *Let (A, B, t_0) be a time query. A consistent arrival connection P dominates a consistent arrival connection Q iff for all consistent arrival connections R with prefix Q, we can replace Q by P to get a consistent arrival connection R'. Also $arr(R') \leq arr(R)$.*

To solve the EAP, we manage a set of dominant arrival connections $ac(S)$ for each station S. The initialization of $ac(A)$ at the departure station A is a special case since we have no real connection to station A. That is why we introduce a special stop event $\forall$ and we start with the set $\{(t_0, \forall)\}$ at station A. Our query algorithm then knows that we are able to board all trains that depart at t_0 or later. We perform a label correcting query that uses the minimum arrival time of the (new) connections as key of a priority queue. This algorithm needs two elementary operations: (1) *link*: We need to traverse an edge $e = (S, T)$ by linking a given set of arrival connections $ac(S)$ with the connections $fn(e)$ to get a new set of arrival connections to station T. (2) *minimum*: We need to combine the already existing arrival connections at T with the new ones to a dominant set. These two operations also dominate the runtime of the query algorithm and we describe in Appendix A efficient implementations. The most important part is a handy order of the connections, primarily ordered by departure time. The minimum operation is then mainly a linear merge operation and the link operation uses precomputed intervals to look only at a small relevant subset of $fn(e)$. We gain additional speedup by combining the link and minimum operation.

We found a solution to the EAP once the key of the priority queue is $\geq$ the minimum arrival time at B.

```
Function timeQuery(A, B, t_0)
    foreach S ∈ B \ A do ac(S) := ∅
    ac(A) := {(t_0, ∀)}
    Q.insert(t_0, A)
    while Q ≠ ∅
        (t, S) := Q.deleteMin()
        if S = B then return t
        foreach edge e := (S, T)
            N := minimum {ac(T), e.link(ac(S))}
            if N ≠ ac(T)
                ac(T) := N
                k := min_{P∈N} arr(P)
                if T in Q then Q.decreaseKey(k, T) else Q.insert(k, T)
    return ⊥
```

Theorem 1 *The time query in the station graph model solves the EAP.*

Proof. The query algorithm only creates consistent connections because link and minimum do so. Lemma 2 ensures that there is never a connection with earlier arrival time. The connections depart from station A not before t_0 by initialization. And since the length of any connection is non-negative, the minimum length arrival connection P at station B after the first deleteMin() of B is a solution to the EAP. □

3.2 Profile Query

A profile query (A, B) is similar to a time query. However, we compute dominant connections instead of dominant arrival connections. Also we cannot just stop the search when we remove B from the priority queue for the first time. We are only allowed to stop the search when we know that we have a dominant set of all consistent connections between A and B. E.g. for daily operating trains, we can compute a maximum duration for a set of connections and can use it to prune the search. The efficient implementations of the minimum and link operation, see Appendix A, are also more complex. Similar to a time query, we use a handy order of the connections, primarily ordered by departure time. The minimum operation is an almost linear merge: we merge the connections in descending order and remove dominated ones. This is done with a sweep buffer that keeps all previous dominant connections that are relevant for the current departure time. The link operation, it links connections from station A to S with connections from station S to T, is more complex: in a nutshell, we process the sorted connections from A to S one by one, compute a relevant interval of connections from S to T as for the time query, and remove dominated connections using a sweep buffer like for the minimum operation.

4 Contraction

In this section, we describe how we contract a station graph timetable network. Interestingly, we cannot directly use the algorithms used for time-dependent road networks [1]. The most time consuming part of the contraction is the witness search: given a node v and an incoming edge (u, v) and an outgoing edge (v, w), is a shortcut between u and w necessary when we contract v? For time-dependent road networks, a min-max search from u to w is performed. It computes a small corridor; this corridor is used by a profile search to decide the necessity of a shortcut. However, in timetable networks, min-max search is not successful, especially the maximum travel time for an edge is very high, e.g. when there is no service during the night. So we need another way to improve the computation of shortcuts. Our idea is to use a one-to-many search from u and use it to identify necessary shortcuts for the contraction of all its neighbors v. Before we start contracting nodes, we do this for all nodes u and store the necessary shortcuts (u, w) with each node v. These stored shortcuts do not take a lot of space since timetable networks are much smaller than road networks. When we contract a node v, we just add the necessary shortcuts. We add shortcuts even in case of equality,

so only the endpoints of the newly introduced shortcuts are affected and we need to update their stored shortcuts. Consider a shortcut (u, w) that is added during the contraction of v. Then we currently do not know which pair of edges (u, w), (w, x) needs a shortcut when w is contracted. A one-to-many forward search from u will give all necessary information to update the stored shortcuts at w. And analogously, a one-to-many backward search from w will give all necessary information to update the stored shortcuts at u. So at most one one-to-many forward search and one one-to-many backward search from each neighbor of v is necessary. When we add a new shortcut (u, w) but there is already an edge (u, w), we merge both edges so there are never parallel edges. Avoiding these parallel edges is important for the contraction, as it performs worse on dense graphs. Thereby, we also ensure that we can uniquely identify an edge with its endpoints. The one-to-many searches can be limited by the duration of the longest shortcut between u and w. We also limit the number of hops and the number of transfers. As observed in [8], this accelerates the witness search at the cost of missed witnesses and potentially more shortcuts.

We could omit loops in static and time-dependent road networks. But for station graph timetable networks, loops are sometimes necessary. We will give an example:

Example 3 *Consider a train that departs at station A at 12:00 then arrives/departs at station B at 12:01, then C at 12:02, then again B at 12:03 and then D at 12:04. Let the transfer time at station B be 5 minutes. We want to go from A to D. In our model, it is not possible to transfer from the train at B at 12:01 to the same train at B at 12:03. However, it is possible to stay at the train via C and then we get a consistent connection from A to D arriving at 12:04. Thus when we contract station C, we need to add a loop at station B.*

Loops also make the witness computation and the update of the stored shortcuts more complex. A shortcut (u, w) for node v with loop (v, v) must not only represent the path $\langle u, v, w\rangle$ but also $\langle u, v, v, w\rangle$. So when we add a shortcut (v, v) during the contraction of another node, we need to recompute all stored shortcuts of node v.

We use two terms for the node ordering: (a) The edge quotient, the quotient between the amount of shortcuts added and the amount of edge removed from the remaining graph. (b) The hierarchy depth, an upper bound on the amount of hops that can be performed in the resulting hierarchy. Initially, we set $\text{depth}(u) = 0$ and when a node v is contracted, we set $\text{depth}(u) = \max(\text{depth}(u), \text{depth}(v)+1)$ for all neighbors u. We weight (a) with 10 and (b) with 1 in a linear combination to compute the node priorities. The nodes are contracted by computing independent node sets with with a 2-neighborhood as described in [18].

5 Query

As any hierarchical time-dependent time query, we suffer from the unknown arrival time. This problem is solved in [1] by computing a corridor from the target node using a min-max backward search in the hierarchy. We replace this min-max search by a breath first search since min-max search does not work so well. Also we do not use the stall-on-demand technique since the additional overhead does not pay off. Our forward and backward search are not interleaved; we first perform the backward and then the forward search. In road networks [1], even for the profile query, the computation of a forward and backward corridor using min-max searches is beneficial. Again, we omit the min-max searches and just use a bidirectional interleaved profile query.

Theorem 2 *The time query in a station graph CH solves the EAP.*

Proof. The proof is similar to the one given in [7]. Here, we will only present a short outline of the proof. From a given time query (A, B, t_0) and a shortest path in the original graph, we construct a path that is found by our query algorithm. Since we use profile searches for the witness computation, Lemma 1 ensures that we can recursively remove a node from the path by replacing the neighboring edges either with a shortcut or a witness path. Different to [7], a shortest path may contain a station more than once. Thus when we

remove a node with loop, we remove not only two but three edges from the path. We cover this case with our special treatment of nodes with loops during the contraction. □

6 Experiments

Environment. Experiments have been done on one core of a dual Xeon 5345 processor clocked at 2.33 GHz with 16 GB main memory and 2 × 2× 4 MB of cache, running SuSE Linux 11.1 (kernel 2.6.27). The program was compiled by the GNU C++ compiler 4.3.2 using optimization level 3.

Test Instances. We have used real-world data from the European railways. The network of the long distance connections of Europe (`eur-longdist`) is from the winter period 1996/97. The network of the local traffic in Berlin/Brandenburg (`ger-local1`) and of the Rhein/Main region in Germany (`ger-local2`) are from the winter period 2000/01. The sizes of all networks are listed in Table 1.

Table 1: Network sizes. We give the the number of nodes and edges in the resulting graph for both models. Note that in the station graph model, the number of nodes corresponds to the number of stations.

network	stations	trains/ buses	elementary connections	time-dependent nodes	time-dependent edges	station edges
`eur-longdist`	30 517	167 299	1 669 666	535 963	1 456 904	88 091
`ger-local1`	12 069	33 227	680 176	225 797	600 690	33 473
`ger-local2`	9 902	60 889	1 128 465	183 207	704 673	26 678

Results. We selected 1 000 random queries and give average performance measures. We compare the time-dependent model and our new station model using a simple unidirectional Dijkstra algorithm in Table 2. Time queries have a good query time speedup above 4 and even more when compared to the #delete mins. However, since we do more work per delete min, this difference is expected. Profile queries have very good speedup around 6 to 8 for all tested instances. Interestingly, our speedup of the #delete mins is even better than for time queries. We assume that more re-visits occur since there are often parallel edges between a pair of stations represented by its onboard nodes. Our model does not have this problem since we have no parallel edges and each station is represented by just one node. It is not possible to compare the space consumption per node since the number of nodes is different in the different models. So we give the the absolute memory footprint: it is so small, we did not even try to reduce it, altough there is potential for that.

Table 2: Performance of the station graph model compared to the time-dependent model on plain Dijkstra queries. *#delete mins* denotes the number of nodes removed from the priority queue, query *times* are given in milliseconds. Moreover, we report the *speedup* over the corresponding value from the time-dependent model.

			TIME-QUERIES				PROFILE-QUERIES			
network	model	space [MB]	#delete mins	speed up	time [ms]	speed up	#delete mins	speed up	time [ms]	speed up
`eur-longdist`	Dependent	27.5	253 349	1.0	68.0	1.0	1 900 520	1.0	2 482	1.0
	Station	48.3	14 519	17.4	14.3	4.8	48 420	39.3	333	7.5
`ger-local1`	Dependent	11.6	110 864	1.0	26.0	1.0	1 172 320	1.0	1 642	1.0
	Station	19.6	5 977	18.5	6.2	4.2	33 647	34.8	289	5.7
`ger-local2`	Dependent	16.1	98 684	1.0	25.6	1.0	1 145 560	1.0	2 822	1.0
	Station	29.4	5 097	19.4	5.5	4.7	27 701	41.3	345	8.2

Before we present our results for CH, we would like to mention that we were unable to contract the same networks in the time-dependent model. The contraction took days and the average degree in the remaining graph exploded. Even when we contracted whole stations at once, including onboard and station nodes, it did not work. It failed since the necessary shortcuts between all the onboard nodes multiplied quickly. So we developed the station graph model to fix these problems. Table 3 shows the the resulting preprocessing and query performance. We get preprocessing times between 4 to 6 minutes using a hop limit of 7, these times are exceptional low compared to previous publications [5, 2]. This is sufficient to reduce time queries below 1 ms for all tested instances. CHs work very well for `eur-longdist` where we get speedups of more than 35 for time queries and 54 for profile queries. When we multiply the speedup of the time-dependent model, we get even a speedup of 172 (time) and 409 (profile) respectively. The network `ger-local2` is also suited for CH, the ratio between elementary connections and stations is just very high so there is more work per settled node. More difficult is `ger-local1`; in our opinion, this network is less hierarchically structured. We see that on the effect of different hop limits for precomputation: we chose 7 as a hop limit for fast preprocessing and then selected 18 as it achieves the best query times for `ger-local1`. The smaller hop limit increases time query times by about 50%, whereas the other two networks just suffer an increase of about 20%. So important witnesses in `ger-local1` contain more edges indicating a lack of hierarchy.

We do not really have to worry about preprocessing space since those networks are very small. The number of edges roughly doubles for all instances. We observe similar results for static road networks [8], but there we can save space with bidirectional edges. But in timetable networks, we do not have bidirectional edges with same weight, so we need to store them separately. In contrast to time-dependent road networks [1] (Germany midweek: 0.4 GB → 4.4 GB), CHs increase the memory consumption by not more than a factor 2.8 (`ger-local1`: 19.6 MB → 54.6 MB). So in our model, CHs have not only fast preprocessing and query times but are even space efficient.

It is hard to compare ourselves to [2] since we do not use the same graph and also not the same scenario. When we just look at the numbers, we are about 2-3 orders of magnitude faster, both for preprocessing and for query times, but our scenario lacks a lot of features.

Table 3: Performance of CH. Preprocessing times are given in seconds, the overhead in megabytes. Moreover, we report the increase in edge count over the input. *#delete mins* denotes the number of nodes removed from the priority queue, query *times* and *speed-up* over a plain Dijkstra are given.

		PREPROCESSING			TIME-QUERIES				PROFILE-QUERIES			
network	hop-limit	time [s]	space [MB]	edge inc.	#del. mins	speed up	time [μs]	speed up	#del. mins	speed up	time [ms]	speed up
`eur-longdist`	7	277	48.6	88%	194	75	399	35.8	266	182	6.1	54.6
	18	1 000	48.0	85%	186	78	331	43.2	260	186	5.4	61.7
`ger-local1`	7	244	35.0	134%	203	29	968	6.4	431	78	50.2	5.8
	18	1 348	33.4	125%	186	32	658	9.5	389	86	40.7	7.1
`ger-local2`	7	367	41.2	122%	155	33	424	13.0	251	110	17.8	19.4
	18	746	39.8	115%	151	34	358	15.4	258	107	15.7	22.0

7 Conclusions

Our model, that has just one node per station, is clearly superior to the time-dependent model for the given scenario. Although the link and minimum operations are more expensive, we are still faster than in the time-dependent model since we need to execute them less often. Also all known speedup techniques that work for the time-dependent model should work for our new model. Most likely, they even work better since the hierarchy of the network is more visible because of the one-to-one mapping of stations to nodes and the lack of parallel edges. We demonstrate that with CHs, they work very well. We achieve query times below 1 ms with preprocessing times around 6 minutes. Our algorithm is therefore suitable for simple web services,

where small query times are very important and can compensate for the quality of the results.

8 Future Work

Our tested instances are very small, especially when we compare it to road networks. However, it is very hard to get more data from the travel agencies. Moreover, combining data from several agencies is hard because of proprietary formats, different levels of granularity, etc. The experience for road networks shows that speedups increase with the size of the network. So we can hope for even better speedups in larger networks. Not only larger, but more realistic scenarios are also important. Our scenario is very basic and the one from [2] has a lot of features. It would be interesting to see the effects of the single features on the speedup techniques, i.e. where do the 2-3 orders of magnitude in performance come from? What are the performance costs of multi-criteria, traffic days, footpaths, etc. and what can we learn to develop faster algorithms? Or is our implementation just highly tuned and the implementation of [2] could be improved with our ideas?

We already saw in Section 6 that CHs work worse when the network is not very hierarchical. Public transportation in Europe is very good and also has a lot of hierarchy. But in other countries, e.g. the United States or Brazil, these networks are worse. There exists e.g. large bus networks with little structure; most likely improved goal directed speedup techniques are necessary for such networks. In some networks, there is no continuous service throughout the day and stations are only important for a few hours. Also the station placement is not very good in every network, e.g. there is a separate station for each bus, and transfers often require a lot of walking, or there are dozens of stops in a very small area. We conjecture, that even in a network with hierarchy, contraction fails when there are too many important footpaths.

But there are not only difficult but also different public transportation timetables, e.g. in London, they specify how frequent a bus will arrive at a station instead of fixed arrival times. We can integrate this frequency based routing in our station graph model. Technically, an edge stores not only a set of timetable connections but also a set of frequency intervals. Of course, in most cases, one of the sets will be empty. When we link an timetable connection and a frequency based connection, we get a connection with a specific departure and arrival time and thus we can handle it as a normal timetable connection.

We should also evaluate different speedup techniques in combination with the station graph model. We only tried CHs, since our original motivation was to proof that they work in timetable networks. But combinations with goal-directed techniques should accelerate the query even more.

References

[1] Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-Dependent Contraction Hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*, pages 97–105. SIAM, April 2009.

[2] Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller–Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*, Dagstuhl Seminar Proceedings, 2009. To appear.

[3] Annabell Berger and Matthias Müller–Hannemann. Subpath-Optimality of Multi-Criteria Shortest Paths in Time- and Event-Dependent Networks. Technical Report 1, University Halle-Wittenberg, Institute of Computer Science, 2009.

[4] Gerth Brodal and Riko Jacob. Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. In *Proceedings of ATMOS Workshop 2003*, pages 3–15, 2004.

[5] Daniel Delling. Time-Dependent SHARC-Routing. *Algorithmica*, July 2009. Special Issue: European Symposium on Algorithms 2008.

[6] Daniel Delling and Dorothea Wagner. Time-Dependent Route Planning. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, Lecture Notes in Computer Science. Springer, 2009. Accepted for publication, to appear.

[7] Robert Geisberger. Contraction Hierarchies. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008.

[8] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

[9] Patrice Marcotte and Sang Nguyen, editors. *Equilibrium and Advanced Transportation Modelling*. Kluwer Academic Publishers Group, 1998.

[10] Matthias Müller–Hannemann and Mathias Schnee. Finding All Attractive Train Connections by Multi-Criteria Pareto Search. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*, pages 246–263. Springer, 2007.

[11] Matthias Müller–Hannemann and Karsten Weihe. Pareto Shortest Paths is Often Feasible in Practice. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*, volume 2141 of *Lecture Notes in Computer Science*, pages 185–197. Springer, 2001.

[12] Karl Nachtigall. Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research*, 83(1):154–166, 1995.

[13] Ariel Orda and Raphael Rom. Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. *Journal of the ACM*, 37(3):607–625, 1990.

[14] Ariel Orda and Raphael Rom. Minimum Weight Paths in Time-Dependent Networks. *Networks*, 21:295–319, 1991.

[15] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12:Article 2.4, 2007.

[16] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[17] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002.

[18] Christian Vetter. Parallel Time-Dependent Contraction Hierarchies. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009.

A Implementation Details

The set of connections for each edge are stored as an ordered array. They are primarily ordered by departure time and then secondarily ordered by arrival time and then critical arrival before non-critical arrivals. For each connection, we only store a representative with departure time in $[0, 1439]$. So the array actually represents a larger *outrolled* array by simply concatenating and shifting the times by 1440. The first piece is at day 0, the second piece at day 1, etc. The set of arrival connections is also stored by an ordered array. They are primarily ordered by arrival time and then critical arrival before non-critical arrivals. This ensures that no arrival connection in the array dominates an arrival connection with lower index.

A basic link algorithm for the time query would link to all connections and afterwards remove the dominated arrival connections. Let $g := |ac(S)|$ and $h := |fn(e = (S, T))|$, the basic algorithm would create up to $\Theta(g \cdot h)$ connections. Especially h can be very large even though usually only a small range in $fn(e)$ is relevant for the link operation. When we identified the first connection we can link to, we can store a *dominant range* with it so that all connections after this range result in dominated arrival connections. We could distinguish between linking to a certain connection with and without transfer but we restrict ourselves only to the case with transfer. This results in a practically very efficient link operation. So given an array of arrival connections $ac(S)$ and an array of connections of an edge $fn(e)$ to relax, the link will work as follows:

1. $edt := \min_{P \in ac(S)} arr(P) \pmod{1440}$ // earliest departure time, in $[0, 1439]$
2. $ett := edt + transfer(S) \pmod{1440}$ // earliest departure with transfer time
3. Find first connection $P_n \in fn(e)$ with minimal *cycle difference*$(edt, dep(P_n))$ using buckets.
4. Find first connection $P_t \in fn(e)$ with minimal *cycle difference*$(ett, dep(P_t))$. Connection P_t gives a dominant range that is identified by the first connection P_e outside the range This partitions the outrolled array of $fn(e)$:

	P_n	P_t	P_e
⋯	link w/o transfer	link w/ transfer	⋯

   We may only link to a connection in $[P_n, P_t)$ without (w/o) transfers and thus all arrival connections in $ac(S)$ are relevant to decide which consistent arrival connections we can create there. It is consistent to link to all connections with transfers from P_t on.
5. While we link, we remember the minimal arrival time and use it to skip dominated arrival connections.
6. Finally we sort the resulting connections and remove the dominated ones. This step is necessary because the minimum arrival time may decrease while we link and we may have to remove duplicates, too.

Given two sets of arrival connections at a node, we want to build the dominant set of the union, the *minimum*. This can be done in linear time by just merging them. Sometimes arrival connections are equivalent but not identical. Two arrival connections are *equivalent* if they are identical or have the same arrival time and neither of them has a critical arrival. In this case we must keep just one of them. We base this decision so that we minimize the number of queue inserts in the query algorithm, e.g. prefer the one from $ac(T)$ if available.

Runtime. The above link operation is is more complex than a usual link operation that maps departure time to arrival time. However, we give an idea why this link operation is very fast and may work in constant time in many cases. The experiments in Section 6 show that it is indeed very efficient. Let b be the number of connections in the bucket. Let c_d be the number of connections that depart within the transfer time window $[P_n, P_t)$ at the station. Let c_a be the number of arrival connections $|ac(S)|$. Let r be the number of connections that depart within the range $[P_t, P_e)$. The runtime of link is then $O(b + c_d c_a + r)$. We choose the number of buckets proportional to the number of connections, so b is in many cases constant. For linking

connections without transfer, we have the product $O(c_d c_a)$ as summand in the runtime. We could improve the product down to $O(c_d + c_a + u)$ with hashing, where u is the number of linked connections. But this is slower in practice since c_d and c_a are usually very small. That is because the station-dependent transfer time window is usually very small, and also only very few connections depart and arrive within a single window. It is harder to give a feeling of the size of the range $[P_t, P_e)$. Assume that every connection operates daily. Let be d the difference between the length of P_t and the minimum length of any connection in $fn(e)$. $d + transfer(S)$ is an upper bound on the size of the time window of this range. So when d is small, and this should be in many cases, also r is small.

Computing the Dominant Range. Besides the buckets we also need to compute the dominant ranges. Lemma 3 gives the instructions how to efficiently compute them using a sweep algorithm approach.

Lemma 3 *Given an array F of connections between two stations S_1 and S_2 that operate daily. The array is primarily orderd by departure time and the secondarily ordered by arrival time and then critical arrival before non-critical arrivals. Let P be a arrival connection at station S_1 that can link with a connection Q in the outrolled array with a transfer. Then all connections that are later in this outrolled array and may not be dominated by the new arrival connection, created by the link of P and Q, depart earlier than $dep(Q) + d + transfer(S_2)$. d is the difference between the length of Q and the minimum length of any connection in F.*

Proof. Let Q' be a connection that does not depart earlier than $dep(Q) + d + transfer(S_2)$. Since $arr(P) + transfer(S_1) \leq dep(Q)$, and $dep(Q) \leq dep(Q')$, we can link P with Q'. By definition of d is $length(Q) \leq length(Q') + d$ and thus $arr(Q) = dep(Q) + length(Q) \leq dep(Q) + length(Q') + d \leq (dep(Q') - d - transfer(S_2)) + length(Q') + d \leq arr(Q') - transfer(S_2)$. When Q' has a critical arrival, then $arr(Q') \leq ndep(Q')$ so that P linked with Q will dominate P linked with Q'. □

Linking two edges for shortcuts and profile search is done by doing the dominant range computation at link time. We change the order of the connections in the array of connections when supporting this operation. They are still primarily ordered by departure. But within the same departure time, the dominant connection should be after the dominated one. That allows for an efficient backward sweep to remove dominated connections. Namely we secondarily order by length descending, then non-critical before critical departure, then non-critical before critical arrival. Finally, we order by the first and last stop event, preferring a stop event with critical departure or arrival. The last order is necessary for an efficient building of a dominant union (minimum) of two connection sets where the preference is on one set.

Given two edges $e_1 = (S_1, S_2)$ and $e_2 = (S_2, S_3)$, we want to link all consistent connections to create $fn(e_3)$ for an an edge $e_3 = (S_1, S_3)$. A trivial algorithm would link each consistent pair of connections in $fn(e_1)$ and $fn(e_2)$ and then compare each of the resulting connections with all other connections to find a dominant set of connections. However, this is impractical for large $g = |fn(e_1)|$ and $h = |fn(e_2)|$. We would create $\Theta(g \cdot h)$ new connections and do up to $\Theta\big((gh)^2\big)$ comparisons.

So we propose a different strategy for linking that is considerably faster for practical instances. We process the connections in $fn(e_1)$ in descending order. Given a connection P, we want to find a connection Q that dominates P at the departure at S_1. So we only need to link P to connections in $fn(e_2)$ that depart in S_2 after the arrival of P but before the arrival of Q. Preferably we want to find the Q with the earliest arrival time. However, we find the Q with the earliest arrival time in S_2 with $dep(Q) \geq dep(P) + transfer(S_2)$. Then Q will not only dominate P at the departure but also any connection departing not later than P. So we can use a simple finger search to find Q. Now we link P only to connections in $fn(e_2)$ departing between the arrival of P and Q. We use finger search to find the first connection that departs in $fn(e_2)$ after the arrival of P. Of course, we need to take the transfer time at S_2 into account when we link. It is not always necessary to link to all connections that depart before Q arrives; we can use the knowledge of the minimum length in $fn(e_2)$ to stop linking when we cannot expect any new dominant connections. The newly linked connections may (1) not be dominant and also may (2) not be in order.

(1) To remove dominated connections, we use a sweep buffer that has as state the current departure time and holds all relevant connections with higher order to dominate a connection with the current departure time. The number of relevant connections is usually small. We need at most all the connections that depart less than $transfer(S_1)$ later than the current departure time and also all connections that depart at least $transfer(S_1)$ later than the current departure time but their arrival time is not more than $transfer(S_3)$ later than the current earliest arrival time. Assuming that only few connections depart in S_1 within $transfer(S_1)$ minutes, and only few connections arrive in S_3 within $transfer(S_3)$ minutes, the sweep buffer has only a few entries.

(2) Connections can only be unordered within a range with same departure time, e.g. when they have ascending durations. So we use the idea of insertion sort to reposition a connection that is not in order. While we reposition a new connection, we must check whether it dominates the connections that are now positioned before it. E.g. a new connection with same departure than the previous one but smaller duration may dominate the previous one if the departure is not critical.

After we processed all connections in $fn(e_1)$, we have a superset of $fn(e_3)$ that is already ordered, but some connections may be dominated. This happens when the dominant connection departs after midnight and the dominated connection before, so the periodic border is between them. To remove all dominated connections, we continue scanning backwards through the new connections but now on "day -1" using the sweep buffer. We can stop when no connection in the sweep buffer is of "day 0".

Runtime. We give an idea why this link operation is very fast and may work in linear time in many cases. The experiments in Section 6 show that it is indeed very efficient. Let c_P be the size of the range in $fn(e_3)$ that departs between the arrival of P and Q. Let b_P be the runtime of the finger search to find the erliest connection in $fn(e_2)$ that departs after the arrival of P Let s be the maximum number of relevant connections in the sweep buffer. The runtime of link is then $\mathrm{O}\left(\sum_{P \in fn(e_1)} (c_P s + b_P)\right)$. This upper bound reflects the linking and usage of the sweep buffer. The backward scanning on "day -1" is also included, since it just adds a constant factor to the runtime. The finger search for Q is amortized in $\mathrm{O}(1)$, so it is also included in the runtime above. It is hard to get a feeling for c_P and b_P, they can be large when $h = |fn(e_2)|$ is much larger than $g = |fn(e_1)|$. Under the practical assumption that $\sum_{P \in fn(e_1)} (c_P + b_P) = \mathrm{O}(g + h)$, we get a runtime of $\mathrm{O}((g + h)s)$. As we already argued when we described the sweep buffer, s is small and in many cases constant, so our runtime should be $\mathrm{O}(g + h)$ in many cases.

Building the Minimum of two Sets of Connections. Query algorithms need two basic operations: link and minimum. For newly visited nodes only link is relevant, otherwise a minimum follows a link so it is efficient to integrate both. But first we will describe a standalone minimum operation, we use it compare witness paths and possible shortcuts. It is basically a backwards merge of the ordered arrays of connections and uses a sweep buffer as for the link operation. Also like the link operation, we continue backward scanning on "day -1" to get rid of dominated connections over the periodic border.

Like for an arrival connection, two connections are *equivalent* when they have the same length, an equivalent departure and equivalent arrival. Two connections P and Q have an *equivalent departure* when their departure is identical or when the departure is not critical and they have the same departure time. Analogously, two connections P and Q have an *equivalent arrival* when their arrival is identical or when the arrival is not critical and they have the same arrival time. Because of the order, equivalent connections are next to each other. So we can easily detect them during the merge. Tie breaking is done in a way to reduce the number of priority queue operations.

Runtime. Let g and h be the cardinalities of the two sets we merge. Let s be the maximum size of the sweep buffer. Then the runtime of the minimum operation is $\mathrm{O}((g + h)s)$. Since s is small and in many cases constant, the runtime should be $\mathrm{O}(g + h)$ in many cases.

Integrating Link and Minimum. A minimum operation always follows a link operation when we relax an edge to an already reached station S. This happens quite often for profile queries, so we can exploit this to tune our algorithm. It is quite simple, we directly process the newly linked connections one by one and directly merge them with the current connections at S. When a new connection is not in order, we fix this with the insertion sort idea. The rest is like in the stand-alone minimum operation. This integration reduces required memory allocations and gives significant speedups.

More Features. Our implementations of *link* and *minimum* are tuned for the EAP. But we believe that we can extend our implementation to more features, e.g. multi-criteria, without loosing too much efficiency. The idea with the dominant ranges can be generalized to support different dominance relations. Also, since we already work with sets of connections, we do not have to pay an additional penalty for memory management when we switch to multi-criteria. In a sense, we already have a multi-criteria optimization problem, we optimize for travel time but relax it with the train and transfer information. However, for traffic days, the dominant ranges may work not well. In this case we can e.g. split the connections by day or introduce a dominant linked list of connections, that are relevant from a given earliest departing transfer connection.