Contraction of Timetable Networks with Realistic Transfers*

Robert Geisberger

Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany
geisberger@kit.edu

Abstract. We contribute a fast routing algorithm for timetable networks with realistic transfer times. In this setting, our algorithm is the first one that successfully applies precomputation based on node contraction: gradually removing nodes from the graph and adding shortcuts to preserve shortest paths. This reduces query times to 0.5 ms with preprocessing times below 4 minutes on all tested instances, even on continental networks with 30 000 stations. 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. Also, our graph model does not require parallel edges.

Key words: route planning; public transit; algorithm engineering

1 Introduction

Route planning is one of the showpieces of algorithm engineering. Many hierarchical route planning algorithms have been developed over the past years and are very successful on static road networks (overview in [1]). Recently, Contraction Hierarchies (CH) [2] provided a particularly simple approach with fast preprocessing and query times. CH is solely based on the concept of *node contraction*: removing "unimportant" nodes and adding shortcuts to preserve shortest path distances. One year later, *time-dependent* CH (TCH) [3] was published and works well for time-dependent road networks, but *completely fails* for timetable networks of public transportation systems. In this paper, we show how to adapt CH successfully to timetable networks with realistic transfers, i. e. minimum transfer times. The positive outcome is partly due to our *station graph model*, where each station is a single node and no parallel edges between stations are necessary. Additionally, we change the contraction algorithm significantly and deal with special cases of timetable networks, e. g. loops.

* Partially supported by DFG grant SA 933/5-1, and the 'Concept for the Future' of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

Related Work

Public transportation networks have always been time-dependent, i. e. travel times depend on the availability of trains, buses or other vehicles. That makes them naturally harder than road networks, where simple models can be independent of the travel time and still achieve good results. There are two intensively studied models for modeling timetable information: the *time-expanded* [4,5,6], and the so-called *time-dependent* model[1] [7,8,9,10]. Both models answer queries by applying some shortest-path algorithm to a suitably constructed graph. In the time-expanded model, each node corresponds to a specific time event (departure or arrival), and each edge has a constant travel time. In the time-dependent model, 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 model, [7] propose to model each platform as a separate station and add walking links between them. [11] propose a similar extension for constant and variable transfer and describe it in more detail. Basically, the stations are 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. Those 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, [12,13] 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 connections per incoming edge instead per node. Their improvement over the time-dependent model was mainly that they compare all connections at a station and remove dominated ones.

Speed-up techniques are very successful when it comes to routing in time-dependent road networks, see [14] for an overview. However, timetable networks are very different from road networks [15] and there is only little work on speed-up techniques for them. Goal-directed search (A*) brings basic speed-up [16,17,11,18]. Time-dependent SHARC [19] brings better speed-up by using arc flags in the same scenario as we do and achieves query times of 2 ms but with preprocessing times of more than 5 hours[2]. Based on the station graph model, [13] also applied some speed-up techniques, namely arc flags that are valid on

[1] Note that the time-dependent model is a special technique to model the time-dependent information rather than an umbrella term for all these models.

[2] We scaled timings by a factor of 0.42 compared to [14] based on plain Dijkstra timings.

time periods and route contraction. They could not use node contraction because there were too many parallel edges between stations. Their preprocessing time is over 33 CPU hours resulting in a full day profile query time of more than 1 second (speed-up factor 5.2). These times are 2-3 orders of magnitude slower than ours but a comparison is not possible since they use a fully realistic bi-criteria scenario with footpaths, traffic days and graphs that are not available to us.

2 Preliminaries

We propose a model that is similar to the realistic time-dependent model introduced in [11], 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 stations $\mathcal{B}$, a set of *stop events* $\mathcal{Z}_S$ per station $S \in \mathcal{B}$, 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. A stop even is the consecutive arrival and departure of a train at a station, where no transfer is required. For the corresponding arriving elementary connection c_1 and the departing one c_2 holds $Z_2(c_1) = Z_1(c_2)$. If a transfer between some elementary connections c_1' and c_2' at station $S_2(c_1') = S_1(c_2')$ is required, $Z_2(c_1') \neq Z_1(c_2')$ must hold. We introduce additional stop events for the begin (no arrival) and 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 *cyclediffer-ence*(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 *cyclediffer-ence*$(t_d(c), t_a(c))$. We generally assume that trains operate daily but our model can be extended to work with traffic days. At a station $S \in \mathcal{B}$, it is possible to *transfer* from one train to another, 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).

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 $arr_i(P)$ also include day information to model trips that last longer than a day. Define $S_1(P) := S_1(c_1)$, $S_2(P) := S_2(c_k)$, $Z_1(P) := Z_1(c_1)$, $Z_2(P) := Z_2(c_k)$, $dep(P) := dep_1(P)$, and $arr(P) := arr_k(P)$. Such a sequence P is called a *consistent connection* from station $S_1(P)$ to $S_2(P)$ if it fulfills the following two consistency condi-

tions: (1) the departure station of c_{i+1} is the arrival station of c_i; (2) 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.

Given a timetable, we want to solve the earliest arrival problem (EAP), i.e. to compute the earliest arriving consistent connection between given stations A and B departing not earlier than a specified time t_0. We refer to the algorithm that solves the EAP as *time query*. In contrast, a *profile query* computes an optimal set of all consistent connections independent of the departure time.

3 Station Graph Model

We introduce a model that represents a timetable as a directed graph $G = (\mathcal{B}, E)$ with exactly one node per station. For a simplified model without transfer times, this is like the time-dependent model. The novelty is that even with positive transfer times, we keep one node per station and require no parallel edges. The attribute of an edge $e = (A, B) \in E$ is a set of consistent connections $fn(e)$ that depart at A and arrive at B, usually all elementary connections. Here and in the following we assume that all connections are consistent. 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. So we can avoid parallel edges, as this is important for CH preprocessing. However, even for time queries, we need to consider multiple dominant arrival events per station.

We say that a connection P *dominates* a connection Q if we can replace Q by P (Lemma 1). More formally, let Q be a connection. Define $parr(Q)$ as the (**p**revious) **arr**ival arrival time of the train at station $S_1(Q)$ before it departs at time $dep(Q)$, or $\perp$ if this train begins there. If $parr(Q) \neq \perp$ then we call $res_d(Q) := dep(Q) - parr(Q)$ the *residence time at departure*. We call Q a *critical departure* when $parr(Q) \neq \perp$ and $res_d(Q) < transfer(S_1(Q))$. Symmetrically, we define $ndep(Q)$ as the (**n**ext) **dep**arture time of the train at station $S_2(Q)$, or $\perp$ if the train ends there. When $ndep(Q) \neq \perp$ then we call $res_a(Q) := ndep(Q) - arr(Q)$ the *residence time at arrival*. And Q is a *critical arrival* when $ndep(Q) \neq \perp$ and $res_a(Q) < transfer(S_2(Q))$.

A connection P *dominates* 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) $Z_1(P) = Z_1(Q)$, or Q is not a critical departure, or $dep(P) - parr(Q) \geq transfer(S_1(P))$
(4) $Z_2(P) = Z_2(Q)$, or Q is not a critical arrival, or $ndep(Q) - arr(P) \geq transfer(S_2(P))$

Conditions (1),(2) are elementary conditions. Conditions (3),(4) are necessary to respect the minimum transfer times, when Q is a subconnection of a larger connection.

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

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' with $dep(R) \leq dep(R') \leq arr(R') \leq arr(R)$.*

3.1 Time Query

In this section we describe our baseline algorithm to answer a time query (A, B, t_0). We use a Dijkstra-like algorithm on our station graph that stores labels with each station and incrementally corrects them. A *label* is a connection P stored as a tuple (Z_2, arr)[3], where Z_2 is the arrival stop event and arr is the arrival time including days. The source station is always A, the target station $S_2(P)$ is implicitly given by the station that stores this label. Furthermore, we only consider connections departing not earlier than t_0 at A and want to minimize the arrival time. As we do not further care about the actual departure time at A, we call such a connection *arrival connection*. We say that an arrival connection P *dominates* Q iff all of the following conditions are fulfilled:
(1) $S_2(P) = S_2(Q)$
(2) $arr(P) \leq arr(Q)$
(3) $Z_2(P) = Z_2(Q)$, or Q is not a critical arrival, or $ndep(Q) - arr(P) \geq transfer(S_2(P))$

Lemma 2 shows that dominant arrival connections are sufficient for a time query.

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' with $arr(R') \leq arr(R)$.*

Our algorithm manages a set of dominant **a**rrival **c**onnections $ac(S)$ 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 $\perp$ and we start with the set $\{(\perp, t_0)\}$ at station A. Our query algorithm then knows that we are able to board all trains that depart not earlier than t_0. 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. We found a solution to the EAP once we extract a label of station B from the priority queue, as Theorem 1 proves.

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

[3] Such a label does not uniquely describe a connection but stores all relevant information for a time query.

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. Since the length of any connection is non-negative, and by the order in the priority queue, the first label of B extracted from the priority queue represents a solution to the EAP.

The link and minimum operation dominate the runtime of the query algorithm. The most important part is a suitable order of the connections, primarily ordered by arrival 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 speed-up by combining the link and minimum operation.

3.2 Profile Query

A profile query (A, B) is similar to a time query. However, we compute dominant connections $con(S)$ instead of dominant arrival connections. Also we cannot just stop the search when we remove a label of 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. For daily operating trains, we can compute a maximum length for a set of connections and can use it to prune the search. The efficient implementations of the minimum and link operation are also more complex. Similar to a time query, we use a suitable 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, which 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 Hierarchies (CH)

CH performs preprocessing based on node contraction to accelerate queries. Contracting a node (= station) v in the station graph removes v and all its adjacent edges from the graph and adds *shortcut edges* to preserve dominant connections between the remaining nodes. A shortcut edge bypasses node v and represents a set of whole connections. Practically, we contract one node at a time until the graph is empty. All original edges together with the shortcut edges form the result of the preprocessing, a *CH*.

4.1 Preprocessing

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? We answer this question usually by a one-to-many profile search from u omitting v (*witness search*). If we find for every connection of the path $\langle u, v, w\rangle$ a dominating connection (*witness*), we can omit a shortcut, otherwise we add a shortcut with all the connections that have not been dominated. To keep the number of profile searches small, we maintain a set of necessary shortcuts for each node v. They do not take a lot of space since timetable networks are much smaller than road networks. Then, the contraction of node v is reduced to just adding the stored shortcuts. Initially, we perform a one-to-many profile search from each node u and store with each neighbor v the necessary shortcuts (u, w) that bypass v. The search can be limited by the length of the longest potential shortcut connection from u. After the contraction, we need to update the stored shortcuts of the remaining nodes. The newly added shortcuts (u, w) may induce other shortcuts for the neighbors u and w. So we perform one forward profile search from u and add to w the necessary shortcuts (u, x) bypassing w. A backward profile search from w updates node u. To omit the case that two connections witness each other, we add a shortcut when the witness has the same length and is not faster. So at most two profile searches from each neighbor of v are necessary. When we add a new shortcut (u, w), but there is already an edge (u, w), we merge both edges using the minimum operation, so there are never parallel edges. Avoiding these parallel edges is important for the contraction, which performs worse on dense graphs. Thereby, we also ensure that we can uniquely identify an edge with its endpoints.

We also limit the number of hops and the number of transfers of a witness search. As observed in [2], this accelerates the witness search at the cost of potentially more shortcuts.

We could omit loops in static and time-dependent road networks. But for station graph timetable networks, loops are sometimes necessary when transfer times differ between stations. For example, assume there is a train $T1$: (station sequence) $A \to B \to C$ and another train $T2$: $C \to B \to D$. A large minimum transfer time at B and a small one at C can forbid the transfer from $T1$ to $T2$ at B but make it possible at C. Contracting station C requires a loop at station B to preserve the connection between A and D. These 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.

The order in which the nodes are contracted is deduced from a node priority consisting of: (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 depth(u) = max(depth(u),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. Nodes with higher priority are more 'important' and get contracted later. The nodes are contracted by computing independent node sets with a 2-neighborhood [20]. Also note that [2,3] perform a simulated contraction of a node to compute its edge quotient. [20] improves this by caching witnesses, but still needs to perform a simulated contraction when a shortcut is necessary. We can omit this due to our stored sets of necessary shortcuts.

Interestingly, we cannot directly use the algorithms used for time-dependent road networks [3]. We tried using the time-dependent model for the timetable networks, but too many shortcuts were added, especially a lot of shortcuts between the different train-route nodes of the same station pair occur.[4] Additionally, [3] strongly base their algorithm on min-max search that only uses the time-independent min./max. length of an edge to compute upper and lower bounds. However, in timetable networks, the max. travel time for an edge is very high, e. g. when there is no service during the night. So the computed upper bounds are too high to bring any exploitable advantages. Without min-max search, the algorithm of [3] is drastically less efficient, i. e. the preprocessing takes days instead of minutes.

4.2 Query

Our query is a bidirectional Dijkstra-like query in the CH. A directed edge (v, w), where w is contracted after v, is an *upward* edge, otherwise a *downward* edge. Our forward search only relaxes upward edges and our backward search only downward edges [2]. The node contraction ensures the correctness of the search.

For a CH time query, we do not know the arrival time at the target node. We solve this by marking all downward edges that are reachable from the target node. The standard time query algorithm, using only upward edges and the marked downward edges, solves the EAP. The CH profile query is based on the standard profile query algorithm. Note that using further optimizations that work for road networks (stall-on-demand, min-max search) [3] would even slowdown our query.

5 Experiments

Environment. The experimental evaluation was done on one core of a Intel Xeon X5550 processors (Quad-Core) clocked at 2.67 GHz with 48 GiB of RAM[5] and 2x8MiB 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.

[4] We tried to merge train-route nodes but this brought just small improvements.
[5] We never used more than 556 MiB of RAM, reported by the kernel.

Table 1. Network sizes and number of nodes and edges in the graph for each model.

network	stations	trains/ buses	elementary connections	time-dependent nodes	time-dependent edges	station based nodes	station based edges
`eur-longdist`	30 517	167 299	1 669 666	550 975	1 488 978	30 517	88 091
`ger-local1`	12 069	33 227	680 176	228 874	599 406	12 069	33 473
`ger-local2`	9 902	60 889	1 128 465	167 213	464 472	9 902	26 678

Table 2. Performance of the station graph model compared to the time-dependent model on plain Dijkstra queries. We report the total *space*, the *#delete mins* from the priority queue, query *times*, and the *speed-up* compared to the time-dependent model.

			TIME-QUERIES				PROFILE-QUERIES			
network	model	space [MiB]	#delete mins	spd up	time [ms]	spd up	#delete mins	spd up	time [ms]	spd up
`eur-longdist`	time-dep.	27.9	259 506	1.0	54.3	1.0	1 949 940	1.0	1 994	1.0
	station	48.3	14 504	17.9	9.4	5.8	48 216	40.4	242	8.2
`ger-local1`	time-dep.	11.3	112 683	1.0	20.9	1.0	1 167 630	1.0	1 263	1.0
	station	19.6	5 969	18.9	4.0	5.2	33 592	34.8	215	5.9
`ger-local2`	time-dep.	10.9	87 379	1.0	16.1	1.0	976 679	1.0	1 243	1.0
	station	29.3	5 091	17.2	3.5	4.6	27 675	35.3	258	4.8

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.

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 speed-up above 4.5 and even more when compared to the number of delete mins. However, since we do more work per delete min, this difference is expected. Profile queries have very good speed-up around 5 to 8 for all tested instances. Interestingly, our speed-up of the number of 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 train-route 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 in the different models different. So we give the absolute memory footprint: it is so small that we did not even try to reduce it, altough there is some potential.

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 with all of its route nodes at once, it did not work. It failed since the necessary shortcuts between all the train-route nodes multiplied quickly. So we developed the station graph model to fix these problems. Table 3 shows the resulting preprocessing and query performance. We get preprocessing times between 3 to 4 minutes using a hop limit of 7. The number of transfers is limited to the maximal number of transfers of a potential shortcut + 2. These timings are exceptional low (minutes instead of hours) compared to previous publications [19,13] and reduce time queries below 550 μs for all tested instances. CH work very well for `eur-longdist` where we get speed-ups of more than 37 for time queries and 65 for profile queries. When we multiply the speed-up of the comparison with the time-dependent model, we even get a speed-up of 218 (time) and 534 (profile) respectively. These speed-ups are one order of magnitude larger than previous speed-ups [19]. The network `ger-local2` is also suited for CH, the ratio between elementary connections and stations is however 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 15 to show further tradeoff between preprocessing and query time.) The smaller hop limit increases time query times by about 25%, whereas the other two networks just suffer an increase of about 16%. 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 [2], but there we can save space with bidirectional edges. But in timetable networks, we do not have bidirectional edges with the same weight, so we need to store them separately. CH on timetable networks are inherently space efficient as they are event-based, they increase the memory consumption by not more than a factor 2.4 (`ger-local1`: 19.6 MiB $\rightarrow$ 47.5 MiB). In contrast, CH time-dependent road networks are not event-based and get very complex travel time functions on shortcuts, leading to an increased memory consumption (Germany midweek: 0.4 GiB $\rightarrow$ 4.4 GiB). Recent work reduces the space consumption by using approximations to answer queries exactly [21].

6 Conclusions

Our work has two contributions. First of all the station graph model, which 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 speed-up 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 absence of parallel edges.

Table 3. Performance of CH. We report the preprocessing *time*, the *space* overhead and the increase in edge count. For query performance, we report the *#delete mins* from the priority queue, query *times*, and the *speed-up* over a plain Dijkstra (Table 2).

		PREPROCESSING			TIME-QUERIES				PROFILE-QUERIES			
	hop-	time	space	edge	#del.	spd	time	spd	#del.	spd	time	spd
network	limit	[s]	[MiB]	inc.	mins	up	[μs]	up	mins	up	[ms]	up
`eur-`	7	210	45.7	88%	192	75.7	251	37.5	260	186	3.7	65.1
`longdist`	15	619	45.3	86%	183	79.3	216	43.5	251	192	3.4	71.4
`ger-`	7	216	27.9	135%	207	28.8	544	7.3	441	76	27.0	8.0
`local1`	15	685	26.9	128%	186	32.1	434	9.2	426	79	24.2	8.9
`ger-`	7	167	36.0	123%	154	33.1	249	14.0	237	117	9.5	27.1
`local2`	15	459	35.0	117%	147	34.6	217	16.1	228	121	8.2	31.3

Our second contribution is the combination of the CH algorithm and the station graph model. With preprocessing times of a few minutes, we answer time queries in half a millisecond. Our algorithm is therefore suitable for web services with high load, where small query times are very important and can compensate for our restricted scenario.

In our opinion, our presented ideas build the algorithmic core to develop efficient algorithms in more realistic scenarios. Especially the successful demonstration of the contraction of timetable networks brings speed-up techniques to a new level. It allows to reduce network sizes and to apply other speed-up techniques only to a core of the hierarchy, even in case that the contraction of all nodes is infeasible.

References

1. 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. Volume 5515 of Lecture Notes in Computer Science. Springer (2009) 117–139
2. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. [22] 319–333
3. Batz, G.V., Delling, D., Sanders, P., Vetter, C.: Time-Dependent Contraction Hierarchies. In: Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09), SIAM (April 2009) 97–105
4. Müller–Hannemann, M., Weihe, K.: 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., Springer (2001) 185–197
5. Marcotte, P., Nguyen, S., eds.: Equilibrium and Advanced Transportation Modelling. Kluwer Academic Publishers Group (1998)
6. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. ACM Journal of Experimental Algorithmics **5** (2000) 12

7. Brodal, G., Jacob, R.: Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. [23] 3–15
8. Nachtigall, K.: Time depending shortest-path problems with applications to railway networks. European Journal of Operational Research **83**(1) (1995) 154–166
9. Orda, A., Rom, R.: Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. Journal of the ACM **37**(3) (1990) 607–625
10. Orda, A., Rom, R.: Minimum Weight Paths in Time-Dependent Networks. Networks **21** (1991) 295–319
11. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Efficient Models for Timetable Information in Public Transportation Systems. ACM Journal of Experimental Algorithmics **12** (2007) Article 2.4
12. Berger, A., Müller–Hannemann, M.: Subpath-Optimality of Multi-Criteria Shortest Paths in Time- and Event-Dependent Networks. Technical Report 1, University Halle-Wittenberg, Institute of Computer Science (2009)
13. Berger, A., Delling, D., Gebhardt, A., Müller–Hannemann, M.: 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)
14. Delling, D., Wagner, D.: Time-Dependent Route Planning. In Ahuja, R.K., Möhring, R.H., Zaroliagis, C., eds.: Robust and Online Large-Scale Optimization. Volume 5868 of Lecture Notes in Computer Science. Springer (2009) 207–230
15. Bast, H.: Car or Public Transport – Two Worlds. In Albers, S., Alt, H., Näher, S., eds.: Efficient Algorithms. Volume 5760 of Electronic Notes in Theoretical Computer Science. Springer (2009) 355–367
16. Hart, P.E., Nilsson, N., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics **4** (1968) 100–107
17. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Towards Realistic Modeling of Time-Table Information through the Time-Dependent Approach. [23] 85–103
18. Disser, Y., Müller–Hannemann, M., Schnee, M.: Multi-Criteria Shortest Paths in Time-Dependent Train Networks. [22] 347–361
19. Delling, D.: Time-Dependent SHARC-Routing. Algorithmica (July 2009) Special Issue: European Symposium on Algorithms 2008.
20. Vetter, C.: Parallel Time-Dependent Contraction Hierarchies (2009) Student Research Project. http://algo2.iti.kit.edu/documents/routeplanning/vetter_sa.pdf.
21. Batz, G.V., Geisberger, R., Neubauer, S., Sanders, P.: Time-Dependent Contraction Hierarchies and Approximation. In: Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10). Lecture Notes in Computer Science, Springer (2010)
22. McGeoch, C.C., ed.: Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08). In McGeoch, C.C., ed.: Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08). Volume 5038 of Lecture Notes in Computer Science., Springer (June 2008)
23. Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'03). In: Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'03). Volume 92 of Electronic Notes in Theoretical Computer Science. (2004)

A Examples

Example 1. This is an example timetable with elementary connections. {(1, 1, A, B, 23:05, 0:55), (1, 1, B, C, 1:02, 2:57), (1, 1, C, D, 3:00,4:20)} describe elementary connections of a train from station A via stations B, C to station D as shown in Figure **??**. The train departs at station A at 23:05 (hh:mm) and arrives at station B at 0:55 the next day. The length of this elementary connection is 1:50 = 110 minutes. {(2, 1, C, E, 3:00, 4:00), (3, 2, 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. We omitted the minimum transfer times of the stations.

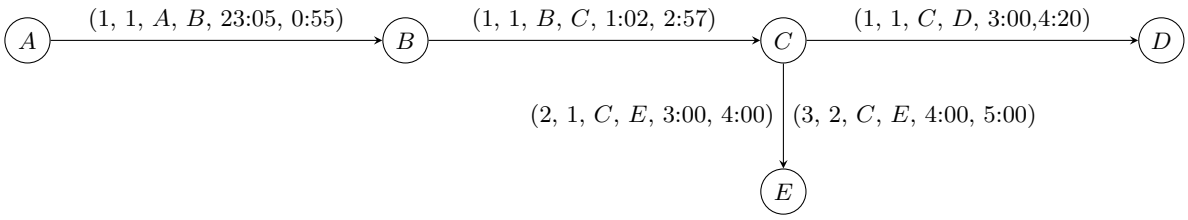

Fig. 1. Every node represents a station and every edge a set of elementary connections.

Example 2.

c_i	$(Z_1, Z_2, S_1, S_2, t_d, t_a)$	dep_i	arr_i
c_1	(1, 1, A, B, 23:05, 0:55)	23:05	24:55
c_2	(1, 1, B, C, 1:02, 2:57)	25:02	26:57
c_3	(3, 2, 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 **??**. Assume a transfer time at station C of 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 $3 < 5 = \mathit{transfer}(C)$ minutes between the arrival and the departure at station C.

Example 3. This example explains the motivation for the notion of a *critical departure/arrival*. Let our timetable be given by Figure **??**. As each edge represents just a single elementary connection, we can represent connections by their sequence of stations.

Let Q be the connection $B \to C \to E$ and let P be the connection $B \to D \to E$. Both connections are consistent and some of their attributes are summarized in Table **??**. Let us decide whether connection P dominates connection Q, i.e. we can replace connection Q by P. The conditions (1) and (2) from Section 3 are already fulfilled. Both depart at 9:10 and the length of Q is 10 minutes and the length of P is 9 minutes. At first glance, it might look like we can replace Q by P.

However, if Q appears as subconnection of a larger connection, the result may not be consistent. Let R be the connection $A \to B \to C \to E \to F$. Connection R is consistent as it is a single train from A to F without any transfers. Q is a subconnection of R. When we replace Q by P in R, we get a connection R' over $A \to B \to D \to E \to F$ with transfers at B and E. Connection R' is not consistent. The reason is that between the arrival of at B at 9:06 and the departure at 9:10 are only 4 minutes time, this is smaller than the minimum transfer time of 5 minutes. If R would arrive earlier, e.g. at 9:05, the transfer would be consistent, $res_d(Q)$ would increase to 5 minutes and Q would not be a critical departure.

Note that at station E we have a different situation as at station B altough Q is a critical arrival (at E). There, the transfer of R' is consistent since between the arrival at 9:19 and the departure at 9:24 are 5 minutes time to transfer. However, in general, conditions (1) and (2) are only sufficient if Q is neither a critical departure nor a critical arrival.

connection	$parr$	dep	res_d	critical departure	arr	$ndep$	res_a	critical arrival
Q	9:06	9:10	4	yes	9:20	9:24	4	yes
P	$\perp$	9:10	–	no	9:19	$\perp$	–	no

Table 4. Attributes of connections Q and P.

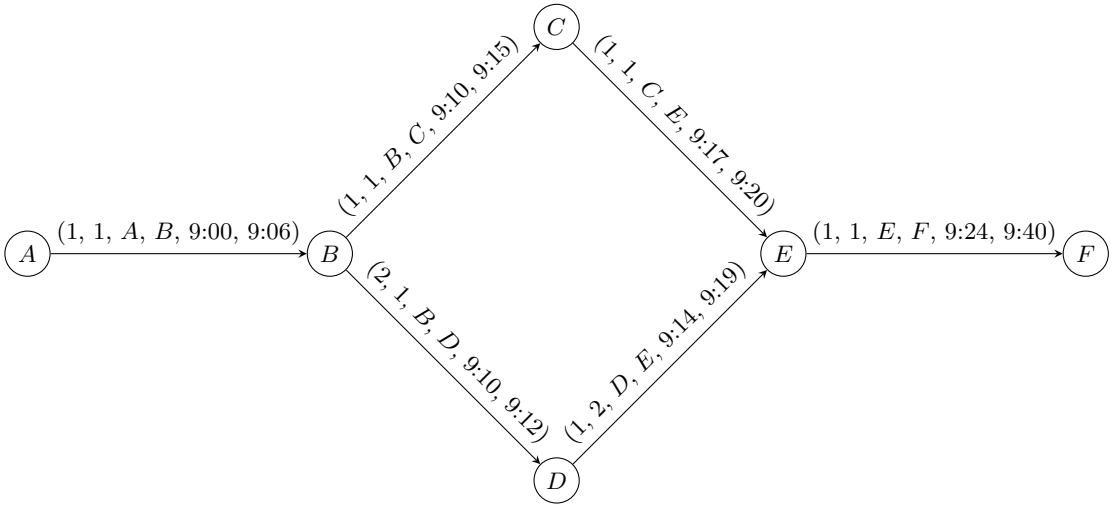

Fig. 2. Every node represents a station and every edge an elementary connection. The minimum transfer time at every station is 5 minutes.

Example 4. Consider train 1 that departs at station A at 12:00 and arrives/departs at station B at 12:01, then gets to C at 12:02. The train 2 departs at C at 12:03, at B at 12:04 and at D at 12:05. The set of elementary connections is {(1, 1, A, B, 12:00, 12:01), (1, 1, B, C, 12:01, 12:02), (2, 2, C, B, 12:03, 12:04), (2, 1, B, D, 12:04, 12:05)}, as shown in Figure **??**. Let the transfer time at station B be 5 minutes and the transfer time at station C be 1 minute. We want to go from A to D. It is not consistent to transfer from train 1 to train 2 at B since it would require a transfer time of 3 minutes or less. However, it is possible to transfer at C and then we get a consistent connection from A to D arriving at 12:05. Thus when we contract station C, we need to add a loop at station B. Technically speaking, the loop allows to transfer between certain connections at station B below the minimum transfer time.

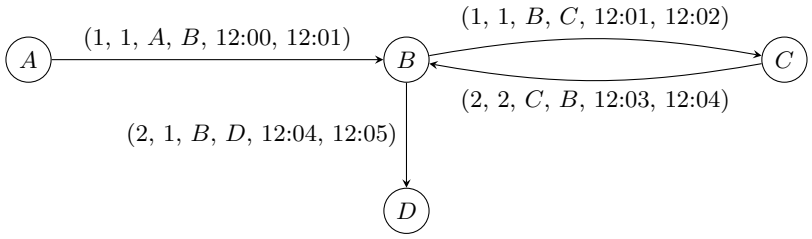

Fig. 3. Every node is a station and every edge an elementary connection. Let $\mathit{transfer}(B) = 5$ and $\mathit{transfer}(C) = 1$, then a loop at station B is necessary after the contraction of station C.

Example 5. This example should indicate why so many shortcuts are added during CH preprocessing in the time-dependent model and why the station graph model is a solution. Assume that we have a network with three stations A, B and C. Let there be three different train routes from A to B and from B to C. This is represented by 3 train-route nodes at A, $3 + 3 = 6$ train-route nodes at B and 3 train-route nodes at C. To allow transfers at stations, there is a dedicated station node and transfer edges from/to the train-route nodes. The graph is shown in Figure **??**. Now assume that we contract station B with all its nodes, and that we always need to add a shortcut. The resulting graph is shown in Figure **??**. So the initially 3 outgoing edges from the train-route nodes of station A and the 3 incoming edges to the train-route nodes of station C multiplied to $3 \cdot 3 = 9$ edges.

Figures **??**, **??** show the graph of the same network, but in the station graph model, before and after the contraction. There is just a single node per station and no parallel edges. During the contraction of station B, we just add a single shortcut from station A to station C representing a set of connections (these connections need not have the same train-route and the set also need not have the FIFO-property).

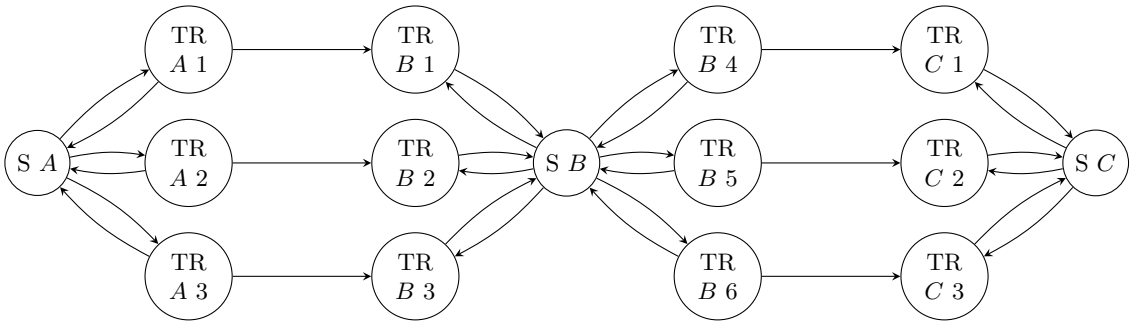

Fig. 4. Example graph in the time-dependent model before the contraction of station B, including the **S**tation node and all **T**rain-**R**oute nodes. The edges between station and train-route nodes are for transfers, the edges between train-route nodes are for connections.

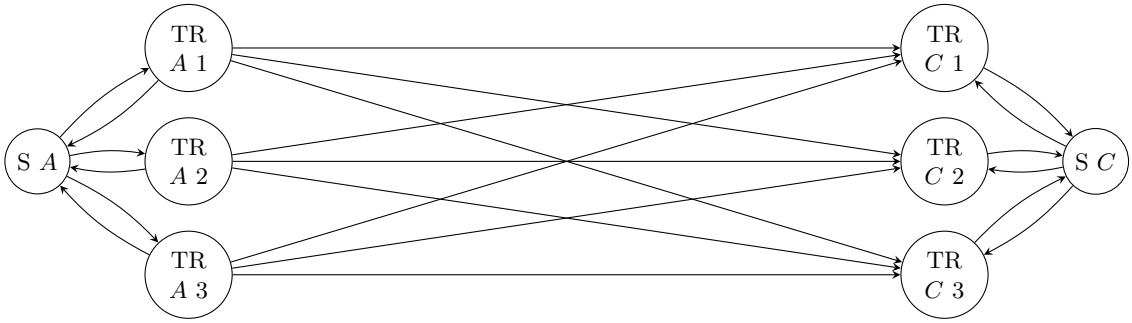

Fig. 5. Example graph in the time-dependent model. After the contraction of station B, there can be an edge from every train-route node of A to every train-route node of C.

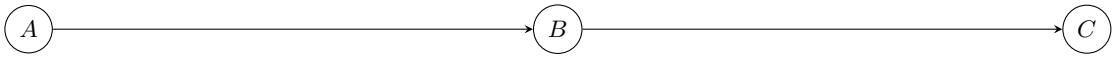

Fig. 6. Example graph in the station graph model before the contraction of station B. There is a single node per station.

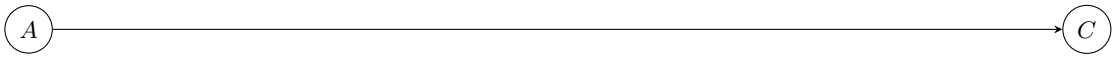

Fig. 7. Example graph in the station graph model after the contraction of station B.

B Proofs

Proof (of Lemma 1). $\Rightarrow$ 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 $Z_1(Q) \neq Z_1(P)$. Condition (3) is satisfied by choosing R as given by the critical departure of Q where Q is prefix of R. Let $Q = (c_1, \ldots, c_k)$ be a critical arrival and $Z_2(Q) \neq Z_2(P)$. Condition (4) is satisfied by choosing R as given by the critical arrival of Q where Q is suffix of R. $\square$

C Pseudo-Code

Algorithm 1: TimeQuery(A, B, t_0)

```
   // tentative dominant sets of arrival connections from A to S
 1 foreach S ∈ B \ A do ac(S) := ∅;
   // special value for A since we depart here at time t_0
 2 ac(A) := {(⊥, t_0)};
   // priority queue, key is earliest arrival time
 3 Q.insert(t_0, A);
 4 while Q ≠ ∅ do
 5     (t, S) := Q.deleteMin();
 6     if S = B then return t;
 7     foreach e := (S, T) ∈ E do
 8         N := minimum {ac(T), e.link(ac(S))};
 9         if N ≠ ac(T) then          // new connections not dominated
10             ac(T) := N;            // update arrival connections at T
11             k := min_{P∈N} arr(P);       // earliest arrival time at T
12             if T in Q then Q.decreaseKey(k, T) else Q.insert(k, T);
13 return ⊥;
```

Algorithm 2: ProfileQuery(A, B)

```
   // tentative dominant sets of connections from A to S
 1 foreach S ∈ B \ A do con(S) := ∅;
   // special value for A since we depart here
 2 con(A) := {⊥};
   // priority queue, key is minimum length from station A
 3 Q.insert(0, A);
 4 while Q ≠ ∅ do
 5     (t, S) := Q.deleteMin();
       // prune due to daily (1440 min. = 1 day) operating trains
 6     if min_{P∈con(B)} {arr(P) − dep(P)} + 1440 < t then return c(B);
 7     foreach e := (S, T) ∈ E do
 8         N := minimum {con(T), e.link(con(S))};
 9         if N ≠ con(T) then          // new connections not dominated
10             c(T) := N;              // update arrival connections at T
11             k := min_{P∈N} {arr(P) − dep(P)};    // min length from A to T
12             if T in Q then Q.decreaseKey(k, T) else Q.insert(k, T);
13 return ⊥;
```

D Figures

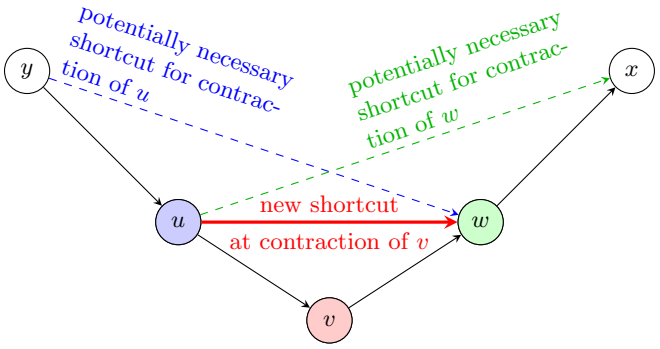

Fig. 8. If the contraction of node v adds a new shortcut (u, w), we need to update the necessary shortcuts stored with u and w. For the contraction of w (in the future), a potentially necessary shortcut (u, x) needs to be considered and for the contraction of u (in the future), a potentially necessary shortcut (y, w) needs to be considered.

E Implementation Details

The ordered set of connections for an edge is stored in an array. It is primarily ordered by departure time, secondarily by arrival time, and finally critical arrivals come 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 of multiple days and we get the connections of different days by shifting the times by 1440.

E.1 Operations on Arrival Connections

The ordered set of arrival connections is also stored in an array. It is primarily ordered by arrival time, and finally critical arrivals come 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.

We improve the basic algorithm. Given a connection $P \in g$, we identify the first connection $P_t \in h$ we can link to. This first connection is the beginning of a *dominant range*. Obviously, there will be a connection in h, so that after this connection, all connections linked with P will result in dominated connections. Therefor, such a connection marks the end of a dominant range. It is preferable to make the dominant range as small as possible, but also supersets of dominant ranges are dominant ranges. We could also 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 since we can precompute the dominant range for each P_t as it is independent of P. 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. $edt := \min_{P \in ac(S)} arr(P) \pmod{1440}$ // **e**arliest **d**eparture **t**ime, in $[0, 1439]$
2. $ett := edt + transfer(S) \pmod{1440}$ // **e**arliest departure with **t**ransfer **t**ime
3. Find first connection $P_n \in fn(e)$ with minimal $cycledifference(edt, dep(P_n))$ using buckets.
4. Find first connection $P_t \in fn(e)$ with minimal $cycledifference(ett, dep(P_t))$. 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 make 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.

Running time. The above link operation 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 5 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 $|ac(S)|$, r be the number of connections that depart within the range $[P_t, P_e)$. The runtime for computing a link is then $\mathrm{O}(b + c_d c_a + r)$. 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 $\mathrm{O}(c_d c_a)$ as summand in the runtime. We could improve the product down to $\mathrm{O}(c_d + c_a + u)$ 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. 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 $fn(e)$. $d + \mathit{transfer}(S)$ 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. To show that our link operation is empirically fast, we give the average of the parameters we used to bound our runtime in Table **??**. The average is the most relevant measure because we perform several hundreds link operations per query.

network	b	c_d	c_a	$c_d \cdot c_a$	r
`eur-longdist`	2.7	0.55	1.12	0.68	1.09
`ger-local1`	3.0	0.57	1.32	0.94	1.22
`ger-local2`	3.2	0.69	1.38	1.20	1.26

Table 5. Average of 1 000 random time queries.

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

Lemma 3. *Assume an array F of connections between two stations S_1 and S_2 that operate daily. The array is primarily ordered 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'.

E.2 Operations on Connections

Linking two edges for shortcuts and profile search uses the dominant range computation at link time. We change the order of the connections in the array for 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 scan to remove dominated connections. So we secondarily order by length descending, thirdly non-critical before critical departure, and finally 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 criterion 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 lengths. 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 length 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".

Running time. We give an idea why this link operation is very fast and may work in linear time in many cases. The experiments in Section 5 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 earliest 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 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 $\sum_{P \in fn(e_1)} (c_P + b_P) = O(g + h)$, we get a runtime of $O((g + h)s)$. As we already argued when we described the sweep buffer, s is small and in many cases constant, so our runtime should be $O(g + h)$ in many cases.

network	c_P	$\frac{\sum_{P \in fn(e_1)} c_P}{g+h}$	b_P	$\frac{\sum_{P \in fn(e_1)} b_P}{g+h}$	$\frac{\sum_{P \in fn(e_1)} (c_P + b_P)}{g+h}$	s
`eur-longdist`	0.93	0.29	3.25	0.84	1.14	2.34
`ger-local1`	1.26	0.29	3.97	0.93	1.22	3.85
`ger-local2`	1.44	0.30	4.30	1.08	1.38	5.30

Table 6. Average of 1 000 random profile queries.

Constructing the Minimum of two Sets of Connections. Query algorithms need two basic operations: link and minimum. For newly visited nodes only link is relevant, but usually a minimum follows a link, so it is efficient to integrate both. But first we will describe a standalone minimum operation, we use it to 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. Similar 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.

Running time. 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 $O((g + h)s)$. Since s is small and in many cases constant, the runtime should be $O(g + h)$ in many cases. To show that our link and minimum operation are empirically fast, we give the average of the parameters we used to bound our runtime in Table **??**.

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. 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 speed-ups.