# Engineering Time-Expanded Graphs for Faster Timetable Information*

Daniel Delling, Thomas Pajor, and Dorothea Wagner

Department of Computer Science, University of Karlsruhe, P.O. Box 6980, 76128 Karlsruhe, Germany.
{delling,pajor,wagner}@informatik.uni-karlsruhe.de

**Abstract.** We present an extension of the well-known time-expanded approach for timetable information. By remodeling unimportant stations, we are able to obtain faster query times with less space consumption than the original model. Moreover, we show that our extensions harmonize well with speed-up techniques whose adaption to timetable networks is more challenging than one might expect.

## 1 Introduction

During the last years, many speed-up techniques for computing a shortest path between a given source $s$ and target $t$ have been developed. The main motivation is that computing shortest paths in graphs is used in many real-world applications like route planning in road networks or timetable information for railways. Although DIJKSTRA's algorithm [6] can solve this problem, it is far too slow to be used on huge datasets. Thus, several speed-up techniques have been developed (see [5] for an overview) yielding faster query times for typical instances. However, recent research focused on developing speed-up techniques for road networks, while only few work has been done on adapting techniques to graphs deriving from timetable information systems. In general, two approaches exist for modeling timetable information: The time-dependent and time-expanded approach. While the former yields smaller inputs (and hence, smaller query times), the latter allows a more flexible modeling of additional constraints. It turns out that adaption of speed-up techniques to each of these models is more challenging than one might expect.

In this work, we use a different approach for obtaining faster query times. Instead of applying a routing algorithm, e.g., plain DIJKSTRA, on the original model, we improve the *time-expanded* model itself in such a way that a routing algorithm does not exploit parts of the graph not necessary for solving the earliest arrival problem (EAP). Interestingly, it turns out that those optimizations are included in the time-dependent approach implicitely. By introducing those techniques to the time-expanded approach, query times for the time-expanded approach are comparable to the time-dependent approach.

### 1.1 Related Work

The simple, i.e., without realistic transfers, time-expanded model has been introduced in [22]. The model has been generalized in [19] in order to deal with realistic transfers. Since then, this realistic model has been used for many experimental studies, e.g., [15, 20, 2]; most of them focusing on faster speed-up techniques or multi-criteria optimization for timetable information. However, [22] enriched the simple time-expanded graph by shortcuts and [20] introduced minor changes to the time-expanded model itself by removing unnecessary nodes with outgoing degree 1.

* Partially supported by the Future and Emerging Technologies Unit of EC (IST priority – 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

### 1.2 Our Contributions

This paper is organized as follows. Section 2 includes formal definitions and a review of the time-expanded model for timetable information. Our main contribution is Section 3. We show how the main ingredient for high-performance speed-up techniques in road networks, i.e., *contraction*, can be adapted to time-expanded graphs. Unfortunately, it turned out that this contraction yields a tremendous growth in number of edges (unlike in road networks). However, by changing the modeling of unimportant stations, a DIJKSTRA does not exploit unnecessary parts of the network. The key observation is the following. Assume $T$ is a station with only one line stopping. A passenger traveling via $T$ only leaves the train if $T$ is her target station, otherwise it never pays off to leave the train. Moreover, we are able to generalize this approach to stations with more lines stopping at that station. In Section 4 we introduce a new speed-up technique tailored to time-expanded graphs based on blocking certain connections. Furthermore, we show how existing techniques have to be adapted to timetable graphs. It turns out that certain pitfalls exist that one might not expect. However, those adapted techniques harmonize well with our new approaches, which we confirm by an experimental evaluation in Section 5. We conclude our work in Section 6 with a summary and future work.

A preliminary version of this paper has been published in [4]. Besides some minor improvements, we here provide detailed proofs of correctness.

## 2 Preliminaries

Throughout the whole work, we restrict ourselves to the earliest arrival problem (EAP), i.e., find a connection in a timetable network with lowest travel time. In the following we often call this single-criteria search in contrast to multi-criteria search that also minimizes number of transfers and further criteria [15, 20].

Moreover, we restrict ourselves to simple, directed graphs $G = (V, E, \text{length})$ with positive length function $\text{length} : E \to \mathbb{R}^+$. The reverse graph $\overline{G} = (V, \overline{E})$ is the graph obtained from $G$ by substituting each $(u, v) \in E$ by $(v, u)$. A *partition* of $V$ is a family $\mathcal{P} = \{P_0, P_1, \ldots, P_k\}$ of sets $P_i \subseteq V$ such that each node $v \in V$ is contained in exactly one set $P_i$. An element of a partition is called a *cell*. The *boundary nodes* $B_P$ of a cell $P$ are all nodes $u \in P$ for which at least one node $v \in V \setminus P$ exists such that $(v, u) \in E$ or $(u, v) \in E$.

**The Condensed Model** is the easiest approach for modeling timetable information. Here, a node is introduced for each station and an edge is inserted iff a direct connection between two stations exists. The edge weight is set to be the minimum travel time over all possible connections between these two stations. Unfortunately, several drawbacks exist. First of all, this model does not incorporate the actual departure time from a given station. Even worse, travel times highly depend on the time of the day and the time needed for changing trains is also not covered by this approach. As a result, the calculated travel time between two arbitrary stations in such a graph is only a *lower bound* of the real travel time. However, in Section 4 we show that the condensed model is helpful for certain speed-up techniques.

**The (Realistic) Time-Expanded Model.** Throughout this work, we use the realistic time-expanded model allowing realistic queries. Therefore, three types of nodes are

used to represent certain events in the timetable. *Departure* and *arrival nodes* are used to model elementary connections in the timetable. Thus, for each elementary connection $c \in \mathcal{C}$ one arrival and departure node is created and an edge is inserted between them. To model transfers, *transfer nodes* are introduced. For each departure event one transfer node is created which connects to the respective departure node having weight 0. To ensure a minimum transfer time $\textsc{transfer}(S)$ at a specific station $S$, an edge from each arrival node $u$ is inserted to the smallest (considering time) transfer node $v$ where $\Delta(\textsc{time}(u), \textsc{time}(v)) \geq \textsc{transfer}(S)$. Here $\Delta(\cdot, \cdot)$ denotes the time difference between two points in time and $\textsc{time} : V \to \mathcal{T}$ maps each node to its timestamp with respect to the timetable. Due to the periodic nature of our timetables $\Delta$ is defined by

$$\Delta(t_1, t_2) := \begin{cases} t_2 - t_1 & \text{if } t_2 \geq t_1, \\ t_2 + 1440 - t_1 & \text{otherwise.} \end{cases}$$

To ensure the possibility to stay in the same train when passing through a station, an additional edge is created which connects the arrival node with the appropriate departure node belonging to this same train. Further to allow transfers to an arbitrary train, transfer nodes are ordered non-decreasing. Two adjacent nodes (w.r.t. the order) are connected by an edge from the smaller to the bigger node. Furthermore, to allow transfers over midnight, an overnight-edge from the biggest to the smallest node is created. For further details, see [20].

For each edge $e = (u, v)$ in the expanded graph the weight $w(e)$ is defined as the time difference $\Delta(\textsc{time}(u), \textsc{time}(v))$ of the nodes the edge connects. Hence, we call the graph consistent in time, meaning for each path from $u$ to $v$ in the graph, the sum of the edge weights along the paths is equal to the time difference $\Delta(\textsc{time}(u), \textsc{time}(v))$.

For future considerations the following notation will be helpful. Let $\prec \subseteq V \times V$ be a relation which compares two events in time. Since in the expanded model nodes correspond to events with a certain timestamp, our relation is defined on the set of nodes of the graph. We say for two nodes $u, v \in V$ that $u \prec v$ if the event of $u$ is happening *before* the event of $v$. Please note that it cannot be determined for $u$ and $v$ if $u \prec v$ just by comparing $\textsc{time}(u)$ and $\textsc{time}(v)$ due to the periodic nature of the timetable and the fact that times are always expressed in minutes after midnight. If for example $\textsc{time}(u) = 400$ and $\textsc{time}(v) = 600$ there are two possibilities. Either $u \prec v$ with $\Delta(u, v) = 200$ or $v \prec u$ with $\Delta(v, u) = 1640$. As a consequence, the $\Delta$ function applied to a tuple $(u, v)$ is *only* valid if $u \prec v$.

# 3 Engineering the Time-Expanded Model

In this section, we present approaches how to enhance the classical time-expanded model. Our first attempt applies a technique deriving from road networks, i.e., contraction, to railway graphs. However, it turns out that this approach yields a too high number of edges. Hence, we also introduce the *Route-Model* which changes the modeling of "unimportant" stations.

## 3.1 Basic Contraction

All speed-up techniques developed during the last years have one thing in common. During preprocessing they apply a contraction routine, i.e., a process that removes unimportant

nodes from the graph and adds shortcuts to the graph to keep the distances between the remaining nodes correct. Interestingly, the fastest hierarchical technique for routing in road networks, Contraction Hierarchies [7], relies *only* on such a routine. The key observation is that in road networks, the average degree of remaining nodes does *not* explode.

At a glance, one could be optimistic that contraction also works well in railway networks. Like in road networks, some nodes in time-expanded graphs are more important than others. However, contraction does not exploit the special structure of time-expanded timetable graphs. For example, departure nodes have an outgoing degree of 1. Thus, we can safely remove such nodes and add a shortcut between the corresponding transfer and arrival node. More precisely, we propose a new contraction routine consisting of three steps. In the following we explain each step separately.

**Omitting Departure Nodes** The first step of our contraction routing bypasses *all* departure nodes. In [20], the authors state that departure nodes can be omitted in time-expanded graphs which can be interpreted as bypassing those nodes.

**Omitting Arrival Nodes** In a second step, we bypass *all* arrival nodes within the network. As a consequence, the degree of transfer nodes highly increases. By these two steps we reduce the number of nodes by approximately a factor of 3. However, the graph still contains all original transfer nodes of which some are more important than others.

**Bypass Transfer Nodes** The final step of our contraction bypasses nodes according to their degree. We bypass nodes with low degree first yielding changes in the degree of its neighbors. Our contraction ends if all transfer nodes have a total degree at least of $\delta$, which is a tuning parameter. We suggest to use a min-heap to determine the next node to be bypassed. The key of a node $x$ shall be $\deg_{in}(x) + \deg_{out}(x)$.

Note that we need not apply all three steps. While the first step reduces both number of nodes and edges, the following two steps yield higher edge counts. In the following, we call a time-expanded model with shortcut departure nodes, the *phase 1* model. The *phase 2* model has neither arrival nor departure nodes. If we also remove (some) transfer nodes, we call the resulting graph a *phase 3* graph. For an experimental evaluation of this contraction routine, see Section 5.

### 3.2 Route-Model

In our experimental studies, it turned out that our contraction routine from the last section suffers from a dramatic growth in number of edges. Already our phase 2 model has up to 3.6 times more edges than the original graph (cf. Section 5). Hence, we here introduce a different approach, called the *route model*. In contrast to contraction, we exploit certain semantic properties of the time expanded graph regarding transferring which eventually leads to a reduction of the number of shortest paths. The classic time-expanded model allows transfers at a station from each arriving train to *all* subsequent departing trains. However, when planning an itinerary by hand, we would probably do the following intuitive pruning: During the way from the source to the target station assume we find a route which leads to some station $S$ on the way, arriving there at time $t_S$. Then, we would not need to examine paths toward station $S$ with an arrival time $t'_S > t_S$, since computing these paths is redundant as we already arrived at $S$ earlier, and we could achieve the same result by taking the earlier computed path arriving at $S$ at $t_S$ and then waiting at $S$ until $t'_S$. This observation is the basic idea behind the route model.

*Remodeling of Stations.* The modifications to the (original realistic) time-expanded graph are done locally and independently for each station $S$, and involve the following three steps:

1. Remove all outgoing edges from all arrival nodes. This includes edges to transfer nodes as well as edges to the departure node of the same train.
2. Insert a minimal number of new transfer-edges directly from the arrival nodes to departure nodes. This allows us to model transfers more specific without losing any optimal shortest paths in comparison to the original time expanded model.
3. Keep the transfer nodes and their interconnecting edges as well as departure-edges from transfer to departure nodes. Although, there are no more edges in the graph to get from an arrival node to a transfer node, the transfer nodes are still used as source nodes for the actual DIJKSTRA query.

The only non-trivial modification is the second one, where for each arrival node we need to find a minimal set of departure nodes which shall become reachable from the particular arrival node. For that reason let $S$ be the currently considered station and $\mathcal{N}_S$ all *neighbors* of $S$. A station $T \in \mathcal{N}_S$ is called a neighbor of $S$ if at least one elementary connection from $S$ to $T$ exists. Thus, we can speak of *routes* between $S$ and each neighbor from $\mathcal{N}_S$. We now use the following notation. $u$ denotes an arbitrary but fixed arrival node of $S$ from which outgoing edges are inserted. $v$ denotes the departure node toward which the edges $(u, v)$ are inserted. Furthermore, $w$ denotes the arrival node corresponding to the elementary connection to which the departure node $v$ belongs. The basic idea is to insert (at least) one edge per route toward a departure node belonging to the the particular route. So, let us consider some fixed station $T \in \mathcal{N}_S$ with $T \neq R$ where $R$ is the station where we just came from through $u$. Of all departure nodes $v$ belonging to an elementary connection $(v, w)$ from $S$ to $T$ we insert an edge $(u, v)$ in $S$ according to the following criteria.

1. The node $w$ is the smallest (regarding time) possible (meaning it is not in violation with the second criterion) arrival node at $T$ that is after $u$, i.e. $w \succ u$.
2. The node $v$ respects the transfer time criterion at $S$. For that reason it has to hold that $v \succ u + \text{TRANSFER}(S)$ if $u$ and $v$ belong to different trains, or $v \succ u$ if they share the same train.

Obviously, by this strategy we select the edge $(u, v)$ according to the earliest possible arrival event at the *target station* $T$. This yields a transfer to a train which arrives at $T$ by the earliest possible time. Note that if we instead would have chosen $v$ according to the earliest possible departure node at $S$, we could have missed a different train that departs at $S$ later, but arrives at $T$ earlier. Such a scenario is called overtaking of trains. Also note, that if the train belonging to $u$ utilizes the route toward station $T$, it does not necessarily have to be the case, that the inserted edge $(u, v)$ corresponds to the departure event of that specific train. It simply corresponds to the train arriving at $T$ first, which may well be a different train.

*Transfer Times at Neighboring Stations.* While we did respect the transfer time criterion of $S$, we also have to respect the transfer time criterion at $T$. Figure 1 shows why this is important.

On the left side the train $Z_2$ arriving at $T$ just slightly after $Z_1$ is the optimal path, but it can not be transferred to, because at $S$ we only chose $Z_1$ and at $T$ the transfer time is

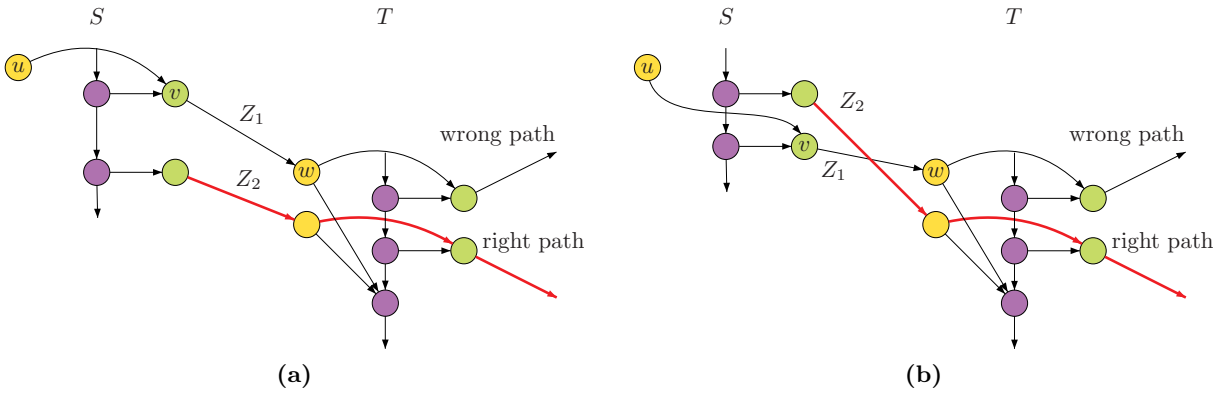


**Fig. 1:** Two problems concerning the transfer time criterion at station $T$.

too big to reach it from $Z_1$. On the right picture the scenario is even worse. While the train $Z_1$ is the earliest train regarding the arrival time at $T$, the optimal route again contains $Z_2$ which departs at $S$ earlier than $Z_2$, but it is not reachable because it arrives at $T$ slightly after $Z_1$. Again the transfer time at $T$ is too big to enter $Z_2$ at $T$. In both cases we have to ensure that $Z_2$ can be entered somewhere. Since our modifications should remain local in the sense that modifications at $S$ should not involve modifications at some other stations, we ensure that $Z_2$ can be reached at $S$.

By adding some more edges to the graph, we are able to allow those connections as well. Let $w_{\text{earl}}$ denote the earliest arrival node at $T$ as computed before. Then, we insert edges $(u, v)$ (belonging to connections $(v, w)$) satisfying the following properties.

1. Consider all trains arriving after $w_{\text{earl}}$ but no later than the transfer time at $T$, meaning $w \succ w_{\text{earl}}$ and $w \prec w_{\text{earl}} + \textsc{transfer}(T)$.
2. Still respect the transfer time criterion at $S$, i.e. $v \succ u + \textsc{transfer}(S)$ if $u$ and $v$ belong to different trains and $v \succ u$ otherwise.

This routine ensures that (a) it is possible to arrive at $T$ as early as possible and (b) all trains that go through $T$ within the margin between the earliest arrival time and the transfer time at $T$ can be reached by entering them at $S$.

*Uncommon Routes.* Despite these modifications, we additionally have to deal with another phenomenom in railway networks. In very few cases, it might pay off to use an itinerary with a sequence of stations $R \to S \to T \to S \to R'$ instead of $R \to T \to R'$. This odd situation may arise if $T$ and $S$ are close to each other, a train runs from $R$ to $T$, another from $T$ to $R'$, and $\textsc{transfer}(S) < \textsc{transfer}(T)$ holds. Figure 2 gives an example.

Our Route-Model does not allow such connections. However, we may overcome this problem by introducing edges at arrival nodes $u$ of $S$ toward departure nodes leading back to $R$ if and only if the following inequation holds:

$$\kappa_{R,S} + \kappa_{S,R} + \textsc{transfer}(S) < \textsc{transfer}(R).$$

Here $\kappa_{R,S}$ denotes the best lower bound regarding travel time from $R$ to $S$. By this we ensure that no shortest paths get lost while in most cases we still get the advantage of prohibiting cycles along the same route. Please note, that we can not rule out cycles such as $\cdots \to R \to S \to T \to R \to \cdots$, however cycles of this type occur less often in general timetable networks.

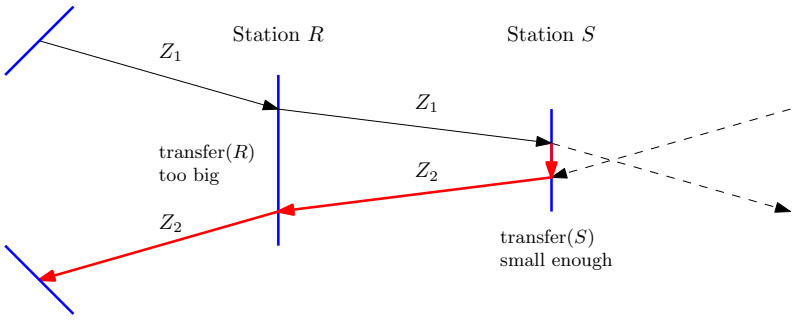


**Fig. 2:** Situation where it is necessary to go forth and back along the same route in order to transfer to train $Z_2$.

*Leaving Big Stations Untouched.* It turns out that remodeling of stations with many neighbors, e.g., major train hubs, lead to a disproportionately high increase in additional edges, since for each neighbor (route) at least one edge must be inserted for each arriving train. In the original time expanded model, however, at most two edges existed for each arrival node (arrival-transfer and arrival-departure). Since our modifications are only local we can choose for each station individually whether we want to convert it to the Route-Model or not. For that reason we introduce a tuning parameter $\gamma$ indicating that stations with more neighbors than $\gamma$ should be left untouched. Hence, changing $\gamma$ yields a trade-off between a speed-up regarding the number of touched nodes against an increasing size of the edge set of the graph.

A problem that arises when mixing Route-Model stations with classic stations is that the main advantage of the Route-Model—subsequent connections on the same route are not visited during the DIJKSTRA search—may fade. Analyzing the example in Figure 3, we observe a big station which has not been converted followed by a route containing a few small stations. While at the small stations no connections exist between connections of the same route, they are nevertheless visited, because they are all accessible through the big station. Hence, we developed *Node-Blocking* which adopts the idea behind the Route-Model

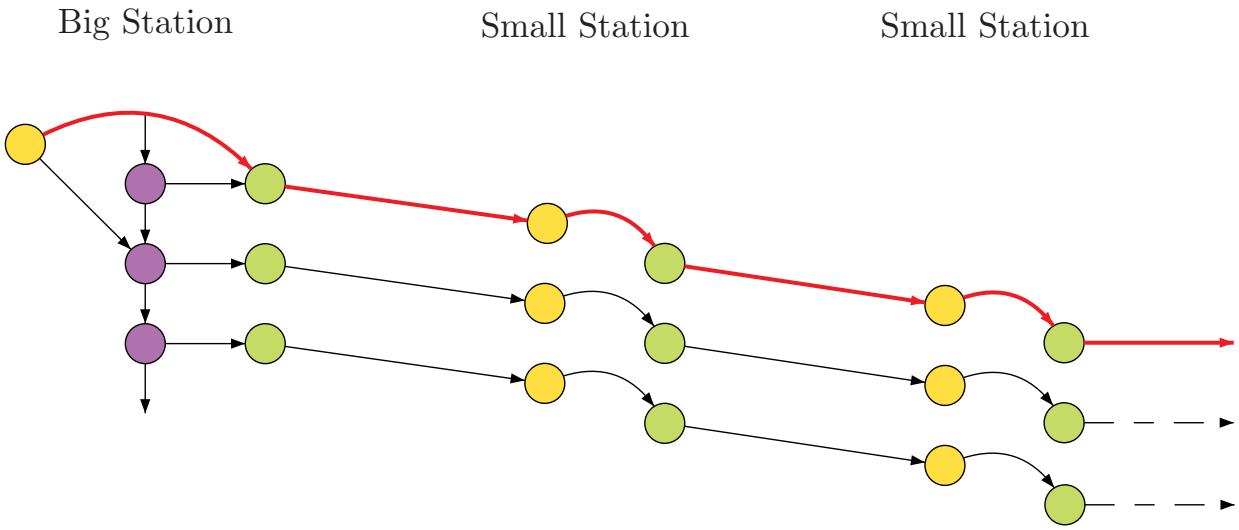


**Fig. 3:** When a big station which is not converted is visited during a DIJKSTRA query, all subsequent connections are visited as well, while only the red path should be relevant. Unimportant nodes are omitted in the figure.

as a speed-up technique, and blocks redundant connections of the same route, so they are not visited. This technique is explained in Section 4.

### 3.3 Correctness of the Route-Model

In this section we provide an extensive correctness proof of our Route-Model, i.e. we show that applying DIJKSTRA on the Route-Model still yields correct solutions to the earliest arrival problem.

In order to conduct our proof we need to introduce some notions first. Let $\Pi$ be a path in a time-expanded railway graph. Then $\Pi$ covers a *sequence of stations* $\mathbf{S} = S_1 \to S_2 \to \cdots \to S_n$. A sequence of the form $S_1 \to S_2 \to \cdots \to S_{k-1} \to S_k \to S_{k-1} \to \cdots S_2 \to S_1$ is called a *cycle*. Note, that there might be more complicated "cycles" like for example $S_1 \to S_2 \to S_3 \to S_1$, but we restrict ourselves to the simple cycles as defined above. A sequence $\mathbf{S}'$ is said to be *contained* in a sequence $\mathbf{S}$, if $\mathbf{S}'$ is part of $\mathbf{S}$, i.e. the sequence $\mathbf{S}'$ occurs at some place in $\mathbf{S}$. A cycle $\mathbf{S}$ is called *dispensable* if it holds that

$$\sum_{i=1}^{k-1} \kappa_{S_i,S_{i+1}} + \text{TRANSFER}(S_k) + \sum_{i=1}^{k-1} \kappa_{S_{i+1},S_i} \quad \geq \quad \text{TRANSFER}(S_1).$$

Here $\kappa_{R,S}$ for two stations $R$ and $S$, again, denotes the minimal travel time from $R$ to $S$. Now, a sequence $\mathbf{S}$ is called *minimal*, if it does not contain any dispensable cycles. A minimal sequence can be constructed from any (non-minimal) sequence by removing every dispensable cycle from it. Think of it as continuing the journey at $S_1$ (of the cycle) directly instead of going through the cycle first. Since the minimal travel time of the cycle is longer than the transfer time at $S_1$, this is always possible.

First, we now prove the following lemma, which is essential to the proof of correctness.

**Lemma 1.** *Each minimal sequence of stations in the realistic time expanded graph is also contained in the Route-Model graph.*

*Proof.* Let $\mathbf{S}$ be a minimal sequence in the time expanded graph. If $\mathbf{S}$ does not contain any cycles, then it is trivially contained in the Route-Model by its construction rules: At each station $S_i$ for each neighbor edges are introduced to connect to them, just as well toward $S_{i+1}$.

If there is a non-dispensable cycle in the sequence $\mathbf{S}$, then the only place where no edges might be in the Route-Model graph is at the turning point $S_k$ of the cycle. The sub-cycle $S_{k-1} \to S_k \to S_{k-1}$ must be non-dispensable itself, otherwise it would not be contained in $\mathbf{S}$. For that reason, it must hold that

$$\kappa_{S_{k-1},S_k} + \text{TRANSFER}(S_k) + \kappa_{S_k,S_{k-1}} \quad < \quad \text{TRANSFER}(S_{k-1}).$$

But this is exactly the criterion for which edges back to $S_{k-1}$ are inserted in the Route-Model. Hence, the path $\mathbf{S}$ is also contained in the Route-Model. □

We can now deduce the main correctness theorem.

**Theorem 1.** *Applying* DIJKSTRA *on the Route-Model yields correct solutions to the earliest arrival problem.*

*Proof.* We prove the theorem in two steps. First, we show that each shortest path in the Route-Model is also contained in the original time expanded model and second, we show the reverse, that for each shortest path in the expanded model there is an equivalently long shortest path in the Route-Model.

*Route Model → Classic Model.* Let $\Pi$ be an arbitrary (shortest) path in the Route-Model covering a sequence $\mathbf{S}$ of stations. The first construction step, namely the removal of edges does not create any new paths in the Route-Model, so by that argument $\Pi$ is also contained in the classic expanded graph. For the second construction step (the appropriate insertion of new outgoing edges from the arrival nodes) does not lead to any new shortest paths either. Since an edge $e = (u, v)$ at some station $S_i$ is only inserted if it does not violate the transfer time criterion, it always corresponds to a valid path in the classic time expanded graph. If no trains are changed through $e$, then $e$ is exactly the *train-edge* from the arrival to the departure node of that train. If trains are changed through $e$, then by the construction rules it holds that $u + \textsc{transfer}(S_i) \prec v$. But, in this case there is also a path (through some transfer nodes) from $u$ to $v$ in the classic graph. By that reason, there are no shorter paths in the Route-Model than in the classic model.

*Classic Model → Route Model.* We now show that no shortest paths get lost by the Route-Model. We prove this by contradiction. Let $\Pi$ be a shortest path of length $\lambda$ retrieved by some query from $S_1$ to $S_n$ at departure time $t_d(S_1)$. Assume that the shortest path $\Pi'$ computed in the Route-Model for the same query has length $\lambda' > \lambda$. Then there are two possibilities.

1. The sequence of stations covered by $\Pi$ and $\Pi'$ are identical.
   Then it must hold that at some station $S_i$ in the classic model we entered a train $Z_f$ that arrives at $S_n$ earlier. We assume without the loss of generality that $S_i$ is the latest possible station (meaning the nearest from the target station) where we entered the faster train $Z_f$. Because there is no possibility to enter $Z_f$ at a later point, for all subsequent stations $S_{i+1}, \ldots, S_n$ it must hold that either $Z_f$ arrives there before the slower train $Z_s$ (computed by the Route-Model), or that it arrives after $Z_s$ but within the margin of the transfer time at the particular station (otherwise $S_i$ would not be the latest possible station to switch to $Z_f$).
   Let $w_f$ be the arrival node of $Z_f$ and $w_s$ be the arrival node of $Z_s$ at the next station $S_{i+1}$. In the first case if $w_f \prec w_s$ there must have been an edge inserted in the Route-Model to board $Z_f$ at $S_i$, because there is always an edge inserted to the train arriving at $S_{i+1}$ earliest. In the second case if $w_s + \textsc{transfer}(S_{i+1}) \succ w_f$, there is also an edge inserted at $S_i$ to board $Z_f$, because edges to all trains along the route are inserted that arrive at $S_{i+1}$ within the margin of $\textsc{transfer}(S_{i+1})$. Hence it is possible to board $Z_f$ at $S_i$ in the Route-Model which is the desired contradiction.
2. The sequence of stations covered by $\Pi$ and $\Pi'$ are not identical.
   Let us call the sequences $\mathbf{S}$ and $\mathbf{S}'$. Because of Lemma 1 there must also exist a (potentially longer) path along $\mathbf{S}$ in the Route-Model. If we substitute $\Pi'$ for that path, this case can be reduced to the first one leading to the desired contradiction.

Thus, the two models are equivalent in the sense that (a) no shorter itineraries can be computed in the Route-Model and (b) for each (shortest) itinerary computed in the classic model there is an equally short itinerary computable in the Route-Model. □

# 4 Speedup Techniques

In principle, we could use DIJKSTRA's algorithm for solving EAP. However, plain DIJKSTRA visits unnecessary parts of the graph, even if we use our Route-Model. Hence, we introduce two approaches for obtaining faster query times. We adapt existing techniques—developed for road networks—to timetable graphs and introduce a new speed-up technique following the ideas from our Route-Model.

## 4.1 Tailored Speed-Up Techniques

**Node-Blocking** is a speed-up technique tailored to time-expanded networks. It basically incorporates the ideas behind the Route-Model as described in Section 3.2: if we can reach a station $S$ at some time $t_S$ we try to prune paths reaching $S$ at a later time $t'_S > t_S$. Recall that the Route-Model prunes the search by removing certain edges from the graph. Node-Blocking, on the contrary, achieves a similar result by dynamically blocking departure nodes during the DIJKSTRA query. The idea is as follows. If we visit a departure node $v$ belonging to an elementary connection targeting some station $T$, we can *prune* all future departure nodes $b$ targeting $T$.

*Preprocessing.* Formally, each departure node $v$ of an elementary connection between two stations $S$ and $T$ induces a set $B_v$ of blocked nodes. A node $b$ is contained in $B_v$ if and only if the following conditions hold.

1. $b$ is a departure node at $S$ belonging to an elementary connection targeting the same station $T$ as $v$.
2. $b \succ v$ holds.
3. If $w$ and $c$ are the arrival nodes at $T$ of the connections associated with $v$ and $b$, respectively, then $w + \text{TRANSFER}(T) \prec c$ must hold, i.e., we respect the transfer time criterion at $T$.

Although the "blocked state" of each node is dynamic in the sense that it depends on the shortest path query, and therefore must be computed during the query, the set $B_v$ of inducing blocked nodes can be precomputed for each node $v$ by iterating through all departure nodes of the station and checking whether the above criteria apply to them.

Note that in contrast to the Route-Model, we do not have to deal with the transfer time criterion at $S$, since we only *block* nodes, and hence never allow a path to be taken which was forbidden by the transfer time criterion at $S$. In worst case, we block departure nodes which cannot be reached anyway due to the transfer time criterion of $S$. Moreover, all special cases are covered by our third condition.

*Query.* The modifications to standard DIJKSTRA algorithm are simple. We introduce an additional flag blocked($v$) to all nodes of the graph, which is initialized to false. Then, whenever we try to insert a node $v$ into the queue, we mark all nodes $B_v$ as blocked. If $v$ is marked as blocked, we prune the search.

*Combination with Route Model.* Although our Route-Model and Node-Blocking follow the same ideas, the advantage of the Route-Model is the lower computation-overhead during the query. However, as discussed in Section 3.2, it does not pay off to remodel major hubs. Hence, Node-Blocking harmonizes well with the Route-Model as we use Node-Blocking for pruning paths at such hubs.

*Combination with Phase 1+ Models.* Since from the Phase 1 model onwards departure nodes are removed, Node-Blocking has to be altered slightly to conform with these models. Instead of departure nodes blocking future departure nodes, we simply let the corresponding arrival nodes (belonging to the respective departure nodes) block each other. In this case, the arrival nodes assume the role of the previous departure nodes regarding blocking, which allows us to continue using the same query algorithm.

**Correctness of Node-Blocking.** We assume at this point that the reader is familiar with the notions introduced during the correctness proof of the Route-Model in Section 3.3, in particular with the terms sequence and cycle. However, we do not restrict ourselves to *simple cycles* here. The term *dispensable cycle* can be generalized to any cycle $\mathbf{S} = S_1 \rightarrow \cdots \rightarrow S_n \rightarrow S_1$ if it holds that

$$\sum_{i=1}^{n-1} \kappa_{S_i,S_{i+1}} + \kappa_{S_n,S_1} \quad \geq \quad \text{TRANSFER}(S_1).$$

Please note, that this condition does not contain transfers along the cycle, because we do not necessarily have a unique "turning point" that induces a transfer.

**Theorem 2.** *Applying Node-Blocking to* Dijkstra*'s algorithm yields correct solutions to the earliest arrival problem.*

*Proof.* We conduct this proof in two steps. First, we show that each shortest path with Node-Blocking enabled is also a path without Node-Blocking. Second, we show that a shortest path without Node-Blocking due to a minimal sequence of stations is also computable with Node-Blocking enabled.

*Node-Blocking → Without Node-Blocking.* This is trivially true. Since Node-Blocking blocks nodes when they are about to be inserted into the priority queue, we can see this as dynamically deleting edges from the graph (namely the edges pointing to blocked nodes) during the query. Obviously, the graph emerging at the end of the query is a subgraph of the original graph, hence the computed path is also contained in the original graph without Node-Blocking enabled.

*Without Node-Blocking → Node-Blocking.* Let $\Pi$ be a shortest path covering a minimal sequence of stations $\mathbf{S}$ computed by Dijkstra. Note again, that for *any* shortest path $\Pi'$ covering a non-minimal sequence $\mathbf{S}'$ we can construct a minimal sequence $\mathbf{S}$ by removing each dispensable cycle. This directly induces the desired path $\Pi$. Then the following two statements hold.

1. *A path $\Pi_B$ with Node-Blocking enabled covering the same sequence exists.*
   The default blocked-state of all departure nodes is $false$. Therefore, when we arrive at some station $S_i$ along the sequence $\mathbf{S}$ on our path, the first departure node leading to $S_{i+1}$ is not blocked when it is inserted into the priority queue (Note, a node never blocks itself, so this is even true if only one connection toward $S_{i+1}$ existed). For that reason, there exists a path from $S_i$ to $S_{i+1}$. Note, that due to the minimal nature of the sequence $\mathbf{S}$ the subsequence $S_i \rightarrow S_{i+1}$ is not contained again in $\mathbf{S}$ at a future point with one exception: The travel time of the cycle (beginning with $S_{i+1}$) is longer

than TRANSFER($S_{i+1}$), but in this case the departure node belonging to the respective connection arriving within the margin of TRANSFER($S_{i+1}$) is not blocked.

2. *$\Pi_B$ is a shortest path.* Assume $t_d(S_i)$ is the first time the DIJKSTRA algorithm discovers a departure node $u$ along the route $S_i \rightarrow S_{i+1}$ in our sequence. Let furthermore $t_a(S_i)$ be the arrival time at $S_{i+1}$ of the connection belonging to that departure node. Now assume further, that the optimal route continues at $S_{i+1}$ at some point $t_d(S_{i+1}) \succ t_a(S_{i+1}) +$ TRANSFER($S_{i+1}$). Then taking the non-blocked connection through $u$ and waiting at $S_{i+1}$ yields an optimal subpath from $S_i$ to $S_{i+1}$. If the optimal journey continues at $S_{i+1}$ within the margin of transfer time, i.e. $t_d(S_{i+1}) \prec t_a(S_{i+1}) +$ TRANSFER($S_{i+1}$) then we ensure that the respective connections arriving within that margin are not blocked by $u$, hence the optimal subpath from $S_i$ to $S_{i+1}$ is prevailed as well. Since $S_i$ and $S_{i+1}$ were arbitrary sections along the (optimal) sequence $\mathbf{S}$, the computed path $\Pi_B$ is a shortest path.

From this follows that Node-Blocking yields correct shortest path queries w.r.t. the earliest arrival problem. □

### 4.2 Adapting Speed-Up Techniques

Although the adaption of many techniques may be promising, we choose basic goal-directed techniques for adaption. It turned out that adaption of more sophisticated techniques, e.g., Highway Hierarchies [21], Contraction Hierarchies [7], REAL [9], SHARC [1], is much more challenging than expected. The main reason are either the need of a bidirectional query algorithm or the bad performance of the contraction routine.

**Arc-Flags.** The classic Arc-Flag approach, introduced in [14, 13], first computes a partition $\mathcal{P}$ of the graph and then attaches a *label* to each edge $e$. A label contains, for each cell $P_i \in \mathcal{P}$, a flag $AF_{P_i}(e)$ which is *true* if a shortest path to at least one node in $P_i$ starts with $e$. A modified DIJKSTRA—from now on called Arc-Flags DIJKSTRA—then only considers those edges for which the flag of the target node's cell is *true*. The big advantage of this approach is its easy query algorithm. However, preprocessing is very extensive. The original approach grows a full shortest path tree from each boundary node yielding preprocessing times of several weeks for instances like the Western European road network. Recently, a new *centralized* approach has been introduced [12]. However, it turns out that this centralized cannot be used in time-expanded transportation networks due to memory consumption. Hence, we use the original approach of growing full shortest path trees from each node.

*Adaption.* The query algorithm can be adapted to time expanded railway graphs very easily. We only have to consider that the exact target node is unknown (just the target station is known). For that reason we simply abort the DIJKSTRA algorithm as soon as a node belonging to the target station is settled. The preprocessing of Arc-Flags, however, needs some extra attention. Since we do not know the exact target node in advance, we have to ensure that all nodes belonging to the same station also get the same cell-id of the partition assigned. For that reason, we simply compute the partition on the condensed graph and map it to the expanded graph by assigning for each node $v \in V$ the cell-id due to cell($v$) := cell(STATION($v$)).

Computing the backwards-shortest path trees from each boundary node of each cell can then be done as described in [14]. However, this approach yields a problem specific on time expanded graphs. Since the length of any path in the graph always corresponds to the time needed to travel between the beginning and ending event (node) of that particular path, any two different paths between the same nodes *always* have the same length. Therefore, the number of shortest paths (in fact, there are *only* shortest paths in time expanded graphs) is tremendous. Unfortunately, if we set flags to true for every path, we do not observe any speed-up (cf. Section 5). In order to achieve a speed-up we have to prefer some paths over others. We examine the following four reasonable strategies for prefering paths:

**Hop Minimization.** For two paths of equal length, choose the one that has less hops (nodes) on it. This approach is often used in road networks [1].

**Transfer Minimization.** Choose the path that has less transfers between trains. While this is a good strategy for querying, it sets too many arc-flags to true, since for different boundary nodes too many different paths lead a transfer-minimal connection.

**Distance Minimization.** Choose the path that is shorter (geographically).

**Direct Geographical Distance.** Choose the path whose direct geographical distance is closer to the source node of the shortest path tree, formally for some node $v$ that is reached from $u$ we choose the new predecessor according to

$$\mathrm{pre}(v)_{\mathrm{new}} := \underset{w \in \{u, \mathrm{pre}(v)\}}{\operatorname{argmin}} \{\sqrt{\big(\mathrm{coord}_x(w) - \mathrm{coord}_x(s)\big)^2 + \big(\mathrm{coord}_y(w) - \mathrm{coord}_y(s)\big)^2}\},$$

where $s$ is the source node of the shortest path tree. This optimization is very aggressive, as it leads to the same result for different boundary nodes of the same cell as often as possible.

Section 5 shows the huge difference in the query performance when the arc-flags are computed with different strategies. Note that we can optimize query times by setting as many flags as possible to false. However, we also loose the ability to choose the "best" path during the query (e.g. due to a minimal number of transfers, costs, etc.). This yields a trade-off between query time and the quality of the computed itineraries.

*Arc-Flags and Node-Blocking.* Unfortunately, Node-Blocking does not harmonize with Arc-Flags. This is due to the fact of Node-Blocking being a very aggressive technique, leaving only very few connection arcs per station and route accessible. The optimization criterion hereby, namely arriving as early as possible at the next station does not necessarily match with our path selection during Arc-Flags preprocessing. As a result, both techniques prune different shortest paths. A possible solution would be to adapt the path selection for Arc-Flags according to Node-Blocking. However, this turns out to be complicated as we have to grow shortest path trees on the reverse graph. Hence, this path selection strategy is not implemented yet.

**ALT.** Goal directed search, also called $A^*$ [11], pushes the search towards a target by adding a *potential* to the priority of each node. The ALT algorithm, introduced in [8], uses a small number of nodes—so called *landmarks*—and the triangle inequality to compute such feasible potentials. Given a set $L \subseteq V$ of landmarks and distances $d(\ell, v), d(v, \ell)$ for all nodes $v \in V$ and landmarks $\ell \in L$, the following triangle inequations hold: $d(u, v) + d(v, \ell) \geq d(u, \ell)$ and

$d(\ell, u) + d(u, v) \geq d(\ell, v)$. Therefore, $\pi(u, t) := \max_{\ell \in L} \max\{d(u, \ell) - d(t, \ell), d(\ell, t) - d(\ell, u)\}$ provides a lower bound for the distance $d(u, t)$ and, thus, can be used as a potential for $u$.

*Adaption.* The query algorithm is, again, straight forward to adapt to time-expanded railway graphs. Since the only difference to the standard DIJKSTRA algorithm is the key which is inserted into the priority queue, we can still simply abort the search as soon as a node of the target station gets settled. However, we cannot compute the landmarks on the expanded graph directly since then we would have to know the target node $t$ in advance. Hence, we compute the landmarks on the much smaller condensed graph which still yields feasible potentials because the edge weights in the condensed graph are defined as the lower bounds regarding travel time. The potential function $\pi$ during the query is then computed as follows:

$$\pi(v) = \max_{\ell \in L} \max\{\text{dist}(\text{STATION}(v), \ell) - \text{dist}(T, \ell), \text{dist}(\ell, T) - \text{dist}(\ell, \text{STATION}(v))\},$$

where $T$ is the target station of the query. We can think of this as using a "lower bound of a lower bound" of the shortest path.

Former studies revealed that the selection of landmark nodes is crucial to the performance of ALT. The quality of the lower bounds highly depends on the quality of the selected landmarks. Thus, several selection strategies exist. To this point, no technique is known how to pick landmarks yielding the smallest search space for random queries. Thus, several heuristics exist. The best are *avoid* and *maxCover*. The first tries to identify regions that are not well covered by landmarks while the latter is basically the avoid routine followed by a local optimization. For details, we refer to [10].

Due to the small size of the condensed networks, another strategy for obtaining potentials seems promising. For each query, we use the target station $T$ as landmark and compute the distances of all stations to $T$ on-the-fly. The advantage of this *dynamic-landmark-selection* is a tighter lower bound. However, we have to run a complete DIJKSTRA in the condensed graph for each query which can take more time than using worse lower bounds from landmarks during the query. Note that this approach for obtaining lower bounds for A* was already proposed in [15].

**Combining Arc-Flags and ALT.** In [17], we observed that Arc-Flags (with the direct geographical distance strategy) and ALT optimize in two different ways. While Arc-Flags prunes paths that lead to the wrong direction geographically, ALT optimizes in time in the sense that fast trains are preferred over slow trains. Fast trains (having less stops in between) tend to get near the target station faster, yielding a lower key in the priority queue regarding the lower bound function. For that reason, it is suggestive to examine the combination of the two speed-up techniques. The implementation is straight-forward, since Arc-Flags does not interfere with ALT—Arc-Flags simply ignores edges that do not have their appropriate flag set, and ALT just alters the key in the priority queue.

## 5 Experiments

In this section, we present our experimental evaluation. Our implementation is written in C++ using solely the STL. As priority queue we use a binary heap. Our tests were executed

on one core of an AMD Opteron 2218 running SUSE Linux 10.3. The machine is clocked at 2.6 GHz, has 16 GB of RAM and 2 x 1 MB of L2 cache. The program was compiled with GCC 4.2, using optimization level 4.

*Inputs.* We use two inputs for our evaluation. The *railway* network of Central Europe and a local *bus* network of greater Berlin. Both networks have been provided by HAFAS for scientific use; the former network consists of 30,517 stations and 1,775,552 elementary connections. The corresponding figures for the latter are 2,874 and 744,005, respectively. While the network of Europe provides a good average structure for a railway network mixed of long-distance trains supported by short-distance trains, the bus network of Berlin consists of a very homogeneous structure, since there are almost no "long-distance" buses. Because of this and the very dense operations of buses with their short travel times between stations, it has already been shown [17] that this network seems to be a very hard instance for timetable information queries.

It should be noted that, while our timetable data is realistic, the transfer times at the stations were not available to us. Hence, we generated them at random and chose between 5 and 10 minutes for the railway and between 3 and 5 minutes for the bus network.

*Default Settings.* In the following, we report preprocessing times and the overhead of the preprocessed data in terms of *additional* bytes per node. We evaluate query performance by running 1 000 random $s$–$t$ queries with source and target station picked uniformly at random. We *fix* the departure time to 7:00 am. We report the average number of settled nodes during the query as well as the average query time. The speed-up refers to the query time and is computed in reference to the classic time expanded model without any speed-up technique applied.

### 5.1 Models

*Parameters.* We start our experimental evaluation with parameter tests for our Route-Model. Recall that in the Route-Model we may affect the conversion process by the selection of $\gamma$ which controls the maximum number of neighbors a station may have in order to become a Route-Model station. In the following we use values between 2 and 10 for $\gamma$. Table 1 reports for both our inputs: the resulting size (in terms of number of edges) and query performance. Note that we do not report number of nodes, as the remodeling routine does not add or remove any nodes. We also enabled Node-Blocking (see Section 4.1).

We observe that for both instances the Route-Model yields a speed-up. Increasing $\gamma$ up to 5 increases performance, while values $> 5$ do not pay off. This is mostly due to the fact that for both graphs the majority of stations has less or equal than 5 neighbors (91% for the Europe and even 99% for the Berlin network).

Concerning Europe with $\gamma < 5$, we observe that the resulting graph has *less* edges than originally. Recall in the original graph the number of outgoing edges per arrival node is at most 2 (one toward the nearest transfer node and one toward the departure node of the same train). Hence, a decrease in number of the edges can only result from merely one edge being inserted for many arrival nodes at stations of degree 2. Interestingly, this observation of decreasing edges does not hold for our bus network which is due to the high density of the network: Because the stations are very close to each other, it often holds that the travel time to go forth and back between some stations $S_1$ and $S_2$ is less than $\text{TRANSFER}(S_1)$,

**Table 1:** The effect of $\gamma$ on the performance of the Route-Model with Node-Blocking enabled.

| | *europe* | | | | *bvb* | | | |
|---|---|---|---|---|---|---|---|---|
| | Size | Query | | | Size | Query | | |
| $\gamma$-value | #edges | #settled | [ms] | speed-up | #edges | #settled | [ms] | speed-up |
| reference | 8,505,951 | 1,161,696 | 534.7 | 1.00 | 3,694,253 | 151,379 | 37.6 | 1.00 |
| 2 | 7,912,584 | 411,836 | 202.4 | 2.64 | 3,785,680 | 91,591 | 27.4 | 1.37 |
| 3 | 8,035,324 | 359,294 | 171.7 | 3.11 | 4,292,849 | 74,963 | 25.2 | 1.49 |
| 4 | 8,332,816 | 329,413 | 158.3 | 3.38 | 5,059,228 | 63,438 | 25.1 | 1.50 |
| 5 | 8,729,619 | 313,046 | 154.1 | 3.47 | 5,437,647 | 59,670 | 25.4 | 1.48 |
| 6 | 9,071,974 | 303,460 | 153.9 | 3.47 | 5,625,277 | 57,990 | 25.6 | 1.47 |
| 7 | 9,396,276 | 297,831 | 155.1 | 3.45 | 5,768,926 | 56,994 | 25.8 | 1.46 |
| 8 | 9,712,940 | 292,482 | 156.4 | 3.42 | 5,782,375 | 56,921 | 25.7 | 1.46 |
| 9 | 9,936,119 | 289,036 | 158.7 | 3.37 | 5,782,375 | 56,921 | 25.8 | 1.46 |
| 10 | 10,195,050 | 285,103 | 159.3 | 3.36 | 5,782,375 | 56,921 | 25.8 | 1.46 |

which results in back-edges being inserted for arrival nodes at $S_2$ (coming from $S_1$). Second, the operation frequency of the buses is very high, such that it may occur that edges toward more than the first bus of the route are inserted, when they arrive at the next station within the margin of its transfer time.

Summarizing, a value of $\gamma = 5$ yields the best results for railway input. The corresponding figure for the bus networks is 4.

*Comparison to the Classic Time-Expanded Model.* Next, we compare different contraction steps (Section 3) and our route model with the classic time expanded model. Table 2 shows the differences in graph size and query performance. While the overall graph size decreases when switching from the classic expanded to the phase 1 model, the number of edges significantely increases if applying our phase 2 model. Although the number of nodes decreases about 50%, this increase in number of edges leads to an *worse* query performance, since more edges are relaxed during the query. We hence conclude that the phase 2 model—and therefore the phase 3 model as well—is not the preferred choice for fast timetable queries.

**Table 2:** Comparison of the different models. The Route-Model is computed with $\gamma = 5$ for *europe* and $\gamma = 4$ for *bvb*.

| | | Size | | Query | | |
|---|---|---|---|---|---|---|
| input | Model | #nodes | #edges | #settled | [ms] | spd-up |
| | Classic expanded | 5,207,980 | 8,505,951 | 1,161,696 | 534.7 | 1.00 |
| | Phase 1 | 3,472,022 | 6,769,991 | 768,181 | 426.5 | 1.25 |
| | Phase 2 | 1,736,064 | 15,571,190 | 431,274 | 631.1 | 0.85 |
| *europe* | Route | 5,207,980 | 8,729,619 | 793,462 | 360.6 | 1.48 |
| | Route w/ blocking | 5,207,980 | 8,729,619 | 313,046 | 154.1 | 3.47 |
| | Route + Phase 1 | 3,472,018 | 6,821,337 | 439,024 | 256.3 | 2.09 |
| | Route + Phase 1 w/ blocking | 3,472,018 | 6,821,337 | 200,213 | 122.8 | 4.35 |
| | Classic expanded | 2,232,016 | 3,694,253 | 151,379 | 37.6 | 1.00 |
| | Phase 1 | 1,488,011 | 2,950,248 | 99,253 | 29.1 | 1.29 |
| | Phase 2 | 744,006 | 13,229,482 | 60,218 | 56.8 | 0.66 |
| *bvb* | Route | 2,232,016 | 5,059,228 | 97,978 | 32.6 | 1.15 |
| | Route w/ blocking | 2,232,016 | 5,059,228 | 63,438 | 25.1 | 1.50 |
| | Route + Phase 1 | 1,488,011 | 3,918,788 | 51,210 | 22.7 | 1.66 |
| | Route + Phase 1 w/ blocking | 1,488,011 | 3,918,788 | 34,032 | 18.6 | 2.02 |

Regarding the Route-Model, the increase in graph size is still reasonable while the query time decreases. However, we see, that the query performance benefits from Node-Blocking as the speed-up more than doubles in the Europe network with Node-Blocking enabled. The reason for the weak performance without Node-Blocking is that paths through the graph, that should be pruned by the Route-Model approach, are still relaxed when they are not blocked in non-converted big traffic hubs. In the bus network the general performance gain is not as big as with the railway network. Even Node-Blocking does not have such a great impact, which is mostly due to the dense structure of this network.

Because the Route-Model can be combined well with the phase 1 model (departure nodes are simply removed after the conversion to the Route-Model), this gives us a gain in graph size while still keeping the advantages of the Route-Model. The query performance behaves as expected and increases by approximately one third compared to the Route-Model alone. If we then additionally apply Node-Blocking on the route + phase1 model, we get the best query performance of all the models which yields a speed-up of 4.35 in the railway network of Europe and 2.02 in the Berlin bus network.

### 5.2 Speedup Techniques

Up to now, we showed that by remodeling stations and using additional pruning techniques, we already achieve a speed-up of 4.35 over plain DIJKSTRA. Here, we now show that this approach harmonizes well with other speed-up techniques deriving from road networks.

*Path-Selection during Arc-Flags Preprocessing.* We already mentioned in Section 4.2 that in expanded timetable networks the number of shortest paths between two nodes is enormously high. It turns out that setting arc-flags for all paths yields a bad query performance. Hence, we have to favor some paths over the others. We proposed four different reasonable strategies: Minimize hops, minimize transfers, minimize accumulated geographic distance along the path and finally minimize the direct geographic distance from the preceding node to the source of the shortest path tree (see Section 4.2). Table 3 shows the impact of each strategy on the performance of Arc-Flags. Note that due to the long preprocessing times of Arc-Flags, we use a subnetwork of our European instance, namely the German railway network called *de_fern* (6822 stations and 554996 connections).

**Table 3:** Arc-Flags. Evaluation of different path-selection strategies. For each strategy we apply a partition with 64 cells.

| | PREPRO | | QUERY | | |
|---|---|---|---|---|---|
| Strategy | [h:m] | [B/n] | #settled | [ms] | speed-up |
| reference | — | 0 | 152,998 | 58.1 | 1.00 |
| hops | 17:00 | 26.2 | 149,931 | 70.3 | 0.83 |
| transfers | 16:26 | 26.2 | 152,307 | 71.7 | 0.81 |
| distance | 20:53 | 26.2 | 134,462 | 61.8 | 0.94 |
| geo. dist. to target | 16:08 | 26.2 | 38,511 | 15.0 | 3.87 |

While minimizing hops is useful in road networks [1] (which can be interpreted there as preferring a route that has less road crossings) this results in a poor performance in railway network. Almost all flags are opened during preprocessing, thus the overhead of the Arc-Flags query algorithm outweighs the benefit from the few remaining pruned arcs.

Interestingly, using minimal transfer or minimal distance strategies as path selection yields a poor query performance as well. This is mostly due to too many different paths of boundary nodes of the same cell being optimal, thus too many flags are set to *true*. Recall that the partition is computed on the condensed graph, hence for one station that is at the border of a cell, nodes belonging to all times of day are boundary nodes which may lead to very different transfer or distance minimal routes in the graph.

The minimal direct geographic distance strategy overcomes this issue by *always* choosing the same preceding node for *all* times of the day. For that reason, as many arc-flags as possible are kept $false$, which eventually yields a speed-up of 3.87 on the German railway network. Since all other strategies actually worsen the query performance, we choose the direct geographic distance strategy for further experiments involving Arc-Flags on time expanded railway networks.

*Speed-Up Techniques on our Models.* In the next experiment we compare the performance of the adapted speed-up techniques on the different models from Section 3. Because of the bad performance of the phase 2 model, we only compare the classic expanded model, the phase 1 model, the Route-Model and the combination of the route and phase 1 models.

Furthermore, we tested the effect of dynamic-landmark-selection against a precomputed set of landmarks. Table 4 shows our results. We show the query performance as well as preprocessing-costs by preprocessing time and additionally bytes per node required to store the preprocessed data. For each model we tested the following speed-up techniques:

- **BA**: Node-Blocking with ALT.
- **BdA**: Node-Blocking with ALT and dynamic-landmark-selection.
- **uFA**: Unidirectional Arc-Flags with ALT.
- **uFdA**: Unidirectional Arc-Flags with ALT and dynamic-landmark-selection.

Regarding classic ALT we always used a set of 8 precomputed landmarks by the *max-Cover* [10] method. Arc-Flags were computed using a partition of 128 cells obtained from *SCOTCH* [18]. The strategy for path-selection was *geographic distance to target*. Note that for Arc-Flags, we turn off Node-Blocking (cf. Section 4.2).

We observe, that for all speed-up technique our modifications to the classic expanded model yield improvements regarding both query performance and preprocessing time. While the transition from the classic to the phase 1 model is more beneficial for Arc-Flags than ALT with Node-Blocking, the latter performs better on the Route-Model where Node-Blocking fits the model considerably better. The combination "Route + Phase 1" unifies the advantages of each model yielding the best speed-ups.

In general, Arc-Flags has a higher impact on query time than ALT together with Node-Blocking (about 5.5 times faster on both networks) which is being paid for with very high preprocessing time and roughly 30 times more required space per node. Note, that the dynamic ALT comes for free, as it does not require any preprocessing at all. With our modified models we can, however, still achieve a speed-up of 10.13 in Europe and 2.54 in Berlin with dynamic ALT and Node-Blocking, which is useful in a scenario where preprocessing is limited or not allowed.

Comparing the standard ALT against ALT with dynamic landmarks, we observe, that regarding query time dynamic ALT only pays off as long as the general speed-up (achieved through some other speed-up technique or model) does not exceed the cost we pay for

**Table 4:** Comparing different models in conjunction with the classic speed-up techniques. The parameter set used throughout: 128 cells, *geographic distance to target* path-selection-strategy for Arc-Flags and 8 landmarks using *maxCover* for the classic ALT.

| | *europe* | | | | | *bvb* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Prepro | | Query | | | Prepro | | Query | | |
| Model/Algorithm | [h:m] | [B/n] | #settled | [ms] | spd | [h:m] | [B/n] | #settled | [ms] | spd |
| Reference | — | 0 | 1,161,696 | 534.7 | 1.00 | — | 0 | 151,379 | 37.6 | 1.00 |
| Classic Exp. (BA) | ≈ 4 s | 4.0 | 261,151 | 162.7 | 3.29 | ≈ 2 s | 4.1 | 96,533 | 33.6 | 1.12 |
| Classic Exp. (BdA) | ≈ 1 s | 4.0 | 233,280 | 130.8 | 4.09 | ≈ 1 s | 4.0 | 94,345 | 29.1 | 1.29 |
| Classic Exp. (uFA) | 106:11 | 106.5 | 71,937 | 32.7 | 16.35 | 45:30 | 108.0 | 49,921 | 17.0 | 2.21 |
| Classic Exp. (uFdA) | 106:11 | 106.5 | 65,143 | 33.9 | 15.77 | 45:30 | 107.9 | 49,014 | 15.2 | 2.47 |
| Phase 1 (BA) | ≈ 5 s | 4.5 | 208,579 | 145.5 | 3.67 | ≈ 2 s | 4.1 | 67,019 | 26.1 | 1.44 |
| Phase 1 (BdA) | ≈ 1 s | 4.0 | 185,996 | 116.4 | 4.59 | ≈ 1 s | 4.0 | 65,488 | 22.8 | 1.65 |
| Phase 1 (uFA) | 77:52 | 127.2 | 30,583 | 14.0 | 38.19 | 31:59 | 129.0 | 15,004 | 5.4 | 6.96 |
| Phase 1 (uFdA) | 77:52 | 126.7 | 27,310 | 18.5 | 29.06 | 31:59 | 128.9 | 14,713 | 5.1 | 7.37 |
| Route (BA) | < 4 s | 4.4 | 140,826 | 73.2 | 7.30 | ≈ 2 s | 4.1 | 49,591 | 22.3 | 1.69 |
| Route (BdA) | ≈ 1 s | 4.0 | 127,444 | 65.4 | 8.18 | ≈ 1 s | 4.0 | 48,390 | 19.8 | 1.90 |
| Route (uFA) | 85:49 | 109.7 | 50,050 | 22.1 | 24.19 | 50:58 | 147.1 | 25,289 | 10.2 | 3.69 |
| Route (uFdA) | 85:49 | 109.3 | 45,180 | 25.3 | 21.13 | 50:58 | 147.0 | 24,785 | 9.3 | 4.04 |
| Route + Ph. 1 (BA) | ≈ 4 s | 4.5 | 89,524 | 58.7 | 9.11 | < 2 s | 4.1 | 26,653 | 16.0 | 2.35 |
| Route + Ph. 1 (BdA) | ≈ 1 s | 4.0 | 80,665 | 52.8 | 10.13 | ≈ 1 s | 4.0 | 26,007 | 14.8 | 2.54 |
| Route + Ph. 1 (uFA) | 83:58 | 128.2 | 20,044 | 9.5 | 56.28 | 34:56 | 170.6 | 6,195 | 2.6 | 14.46 |
| Route + Ph. 1 (uFdA) | 83:58 | 127.7 | 17,805 | 15.2 | 35.18 | 34:56 | 170.5 | 6,053 | 2.8 | 13.43 |

computing the distance table on-the-fly. Since the condensed graph of Europe has about 11 times more stations than the Berlin graph, the cost for computing the dynamic distance table carries much more weight there—A one-to-all Dijkstra takes about 7 ms on the condensed graph of Europe. Hence, it never pays off using dynamic landmarks together with Arc-Flags here. The same effect can be observed in the Berlin network, however, only with the combination of the route and phase 1 models due to the much smaller condensed graph.

Summarizing, our modifications yield a speed-up of 3.5 if we apply ALT and Arc-Flags to both time-expanded graphs. The corresponding figure for our bus network is 5.5. This yields an overall speed-up of 56.28 for Europe and 14.46 for Berlin when compared to the classic model without any speed-up technique applied.

### 5.3 Comparison to the Time-Dependent Model

Table 5 compares the performance of Dijkstra's algorithm and ALT applied to our route+phase 1 time-expanded model and the time-dependent model. We observe that by the introduction of our Route-Model (and Node-Blocking) query performance of time-expanded queries are faster than for the time-dependent approach. Hence, we are able to close the performance-gap between both models. Analyzing the time-dependent approach, we notice that Node-Blocking is included implicitly: During a query we do not relax an edge more than once although it represents several connections running from one station to another. Hence, early connections *block* later ones. Our remodeling and Node-Blocking technique introduces these optimizations to the time-expanded approach. As a result the performance advantage of the time-dependent approach fades.

**Table 5:** Performance of DIJKSTRAand uni-directional ALT using a *time-dependent* variant of our European input. For comparison, the corresponding figure for the time-expanded approach (route-model with phase 1) are given as well.

| | time-dependent | | | | | time-expanded | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | PREPRO | QUERIES | | | | PREPRO | QUERIES | | | |
| | time | #settled | speed | time | speed | time | #settled | speed | time | speed |
| technique | [h:m] | nodes | up | [ms] | up | [h:m] | nodes | up | [ms] | up |
| Dijkstra | 0:00 | 260 095 | 1.0 | 125.2 | 1.0 | 0:00 | 200 213 | 1.0 | 122.8 | 1.0 |
| uni-ALT | 0:02 | 127 103 | 2.0 | 75.3 | 1.7 | 0:01 | 89 524 | 2.2 | 58.7 | 2.1 |

# 6 Conclusion

In this work, we introduced a local remodeling routine for the time-expanded approach based on the intuition that at many stations in a network, the number of reasonable choices is little. It turns out that this approach leads to a closely related speed-up technique harmonizing well with our remodeling. Moreover, we adapted speed-up techniques to the time-expanded model and show that they harmonize well with our new approach. Altogether, our approach yields query times up to 56.28 times faster than pure DIJKSTRA.

Regarding future work, we are optimistic that our approach would also work well for multi-criteria optimization. Although our pruning techniques may not work as strict as for single-criteria search, the number of reasonable choices is little in this scenario as well. Another very important problem is how to handle updates in case of delays. It seems as if updating a time-expanded graph is rather expensive, though possible [3, 16].

# References


1. R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. In I. Munro and D. Wagner, editors, *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, 2008.
2. R. Bauer, D. Delling, and D. Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. In C. Liebchen, R. K. Ahuja, and J. A. Mesa, editors, *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, pages 209–225. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
3. D. Delling, K. Giannakopoulou, D. Wagner, and C. Zaroliagis. Timetable Information Updating in Case of Delays: Modeling Issues. Technical Report 133, Arrival Technical Report, 2008.
4. D. Delling, T. Pajor, and D. Wagner. Engineering Time-Expanded Graphs for Faster Timetable Information. In *Proceedings of the 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08)*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, September 2008.
5. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. A. Zweig, editors, *Algorithmics of Large and Complex Networks*, Lecture Notes in Computer Science. Springer, 2009. To appear.
6. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
7. R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In C. 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.
8. A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165, 2005.
9. A. V. Goldberg, H. Kaplan, and R. F. Werneck. Better Landmarks Within Reach. In C. Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 38–51. Springer, June 2007.
10. A. V. Goldberg and R. F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.

11. P. E. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
12. M. Hilger. Accelerating Point-to-Point Shortest Path Computations in Large Scale Networks. Master's thesis, Technische Universität Berlin, 2007.
13. E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, Lecture Notes in Computer Science, pages 126–138. Springer, 2005.
14. U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. volume 22, pages 219–230. IfGI prints, 2004.
15. M. Müller–Hannemann and M. 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.
16. M. Müller–Hannemann, M. Schnee, and L. Frede. Efficient On-Trip Timetable Information in the Presence of Delays. In *Proceedings of the 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08)*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, September 2008.
17. T. Pajor. Goal Directed Speed-Up Techniques for Shortest Path Queries in Timetable Networks, January 2008. Student Research Project.
18. F. Pellegrini. SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package, 2007.
19. E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Experimental Comparison of Shortest Path Approaches for Timetable Information. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 88–99. SIAM, 2004.
20. E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12:Article 2.4, 2007.
21. P. Sanders and D. Schultes. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.
22. F. Schulz, D. Wagner, and K. Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. In *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE'99)*, volume 1668 of *Lecture Notes in Computer Science*, pages 110–123. Springer, 1999.