Time-Dependent SHARC-Routing

Daniel Delling[1]

[1] *Universität Karlsruhe (TH), 76128 Karlsruhe, Germany*
Email:`delling@ira.uka.de`

Abstract

In recent years, many speed-up techniques for Dijkstra's algorithm have been developed that make the computation of shortest paths in *static* road networks a matter of microseconds. However, only few of those techniques work in *time-dependent* networks which, unfortunately, appear quite frequently in reality: Roads are predictably congested by traffic jams, and efficient timetable information systems rely on time-dependent networks. Hence, a fast technique for routing in such networks is needed.

In this work, we present an efficient time-dependent route planning algorithm. It is based on our recently introduced SHARC algorithm, which we adapt by augmenting its basic ingredients such that correctness can still be guaranteed in a time-dependent scenario. As a result, we are able to efficiently compute exact shortest paths in time-dependent continental-sized transportation networks, both of roads and of railways. It should be noted that time-dependent SHARC was the first efficient algorithm for time-dependent route planning.

1 Introduction

Computing shortest paths in graphs is used in many real-world applications like route planning in road networks, timetable information for railways, or scheduling for airplanes. In general, Dijkstra's algorithm [12] finds a shortest path between a given source s and target t. Unfortunately, the algorithm is far too slow to be used on huge datasets. Thus, several speed-up techniques have been developed (see [9] for an overview) yielding faster query times for typical instances, e.g., road or railway networks. A major drawback of most existing speed-up techniques is that their correctness depends on the fact that the network is static, i.e., the network does *not* change between queries. Only [11, 37] showed how preprocessing can be updated if a road network is perturbed by a relatively small number of traffic jams.

However, in real-world road networks, many traffic jams are predictable. This can be modeled by a time-dependent network, where the travel time depends on the departure time τ. Moreover, a very efficient model for timetable information relies on time-dependent networks (cf. [32] for details) as well. Unfortunately, none of the recent high-performance speed-up techniques can be used in a time-dependent network in a straightforward manner. Moreover, possible problem statements for shortest paths become even more complex in such networks. For instance, a user could ask at what time she should depart in order to spend as little time traveling as possible.

In this work, we present a speed-up technique for exact time-dependent routing in road and railway networks. As a starting point, we decided to use our recently developed SHARC algorithm [2, 3] as it is a fast unidirectional technique. Hence, we avoid the problem of bidirectional search other techniques rely on. Note that if bidirectional search is to be used in time-dependent networks, the arrival time would have to be known in advance. It turns out that time-dependent SHARC is able to efficiently compute shortest paths in time-dependent continental-sized road and timetable networks. Moreover, we are able to compute the shortest paths between two points not only for a specific departure time, but for *all* possible departure times.

Related Work

As already mentioned, a lot of speed-up techniques for static scenarios have been developed in recent years. Since we here focus on *time-dependent* route planning, we direct the interested reader to [9], which gives a recent overview over static routing techniques.

Much less work has been done on time-dependent route planning. In [5], Dijkstra's algorithm is extended to the time-dependent case based on the assumption that the network fulfills the FIFO property. A network is called FIFO if all of its edges fulfill the FIFO property. A FIFO edge (u, v) ensures that if a person A traverses (u, v) before a person B, B cannot arrive at the node v before A. Computation of shortest paths in FIFO networks is polynomially solvable [22]. In non-FIFO networks, complexity depends on the restriction whether waiting at nodes is allowed. If waiting is allowed, the problems stays polynomially solvable; if it is not allowed, the problem is NP-hard [29]. Fortunately, transportation networks can be modeled in such a way that the FIFO property holds.

Goal-directed search, also called A^* [20], has been adapted to the time-dependent scenario [14]; an efficient version (called ALT) for the static case has been presented in [16, 19]. In [11], unidirectional ALT is evaluated on time-dependent graphs (fulfilling the FIFO property) yielding mild speed-ups of a factor between 3 and 5, depending on the degree of time-dependency. Goal-directed search has also been successfully applied to time-dependent timetable networks [31, 32, 13]. In [28], it has been shown that time-dependent ALT can be used in a bidirectional manner. The key idea of bidirectional search in time-dependent networks is that the backward search is only used to bound the nodes the forward search has to visit. This approach can be further accelerated by limiting ALT search to a core extracted during preprocessing [8]. Moreover, our old implementation of static SHARC [2] already allowed fast *approximate* queries in a time-dependent scenario.

Recently, a *hierarchical* speed-up technique, Contraction Hierarchies [15], has been extended to the time-dependent scenario [1] as well. It turns out that the performance of time-dependent Contraction Hierarchies is comparable to the approach presented in this work but for the price of a very high amount of preprocessed data. The main reason for this is the fact that shortcuts are much more expensive in terms of space consumption in time-dependent scenarios (cf. Section 3).

Our Contribution

In this work, we show how SHARC can be generalized in such a way that we are able to perform exact shortest-path queries in time-dependent networks. The key observation is that the concept of SHARC stays untouched. However, at certain points we augment static routines to time-dependent ones. Moreover, we slightly adapt the intuition of Arc-Flags [24, 21]. And finally, we deal with the problem that adding shortcuts to the graph is more expensive than in static scenarios. As a result, we are able to perform *exact* time-dependent queries in road and railway networks.

We start describing our work on time-dependent route planning in Section 2 by introducing basic definitions and a short review of Arc-Flags and SHARC in static scenarios. Basic work on modeling time-dependency in road and railway networks is located in Section 3. We augment the main ingredients of SHARC, namely (local) Dijkstra-searches, contraction, and Arc-Flags, in Section 4. It turns out that the adaption of Dijkstra and contraction is straightforward, while arc-flags computation gets more expensive: The key observation is that we have to alter the intuition of arc-flags slightly for correct routing in time-dependent networks. The preprocessing routine itself and the query algorithms of time-dependent SHARC are located in Section 5. We also provide a detailed proof of correctness and present several optimization techniques to reduce the preprocessing effort and to increase the query performance.

In order to show that time-dependent SHARC performs well in real-world environments, we present an extensive experimental evaluation in Section 6. As inputs we use continental-sized transportation networks. It turns out that SHARC is more than 20 times faster than plain Dijkstra for timetable information. The corresponding figure for road networks is between 60 and > 5000, depending on the degree of time-dependency of the network and preprocessing effort. Section 7 concludes our work with a summary and possible future research.

A preliminary version of this work has been published in [7]. However, we here give detailed proofs of correctness, present a new variant of time-dependent SHARC, and finally, we now have access to real-world time-dependent road networks. An extended computational study reveals further interesting details of time-dependent SHARC.

2 Preliminaries

An (undirected) graph $G = (V, E)$ consists of a finite set V of nodes and a finite set E of *edges*. An edge is an unordered pair $\{u, v\}$ of nodes $u, v \in V$. If the edges are ordered pairs (u, v), we call the graph *directed*. In this case, the node u is called the *tail* of the edge, v the *head*. The number of nodes $|V|$ is denoted by n, the number of edges by m. We say a graph is *sparse* if $m \in O(n)$. Throughout this paper we restrict ourselves to directed sparse graphs which are weighted by a length function len, with $len(u, v)$ depicting the *travel time* from u to v. Given a set of edges H, tail(H) / head(H) denotes the set of all tails / heads in H. By $\deg_{in}(v)$ / $\deg_{out}(v)$ we denote the number of edges whose head / tail is v. The reverse graph $\overleftarrow{G} = (V, \overleftarrow{E})$ is the graph obtained from G by substituting each $(u, v) \in E$ by (v, u). The 2-core of an undirected graph is the maximal node-induced subgraph of minimum node degree 2. The 2-core of a directed graph is the 2-core of the corresponding simple, unweighted, undirected graph. All nodes not being part of the 2-core are called 1-shell nodes. A tree on a graph for which only the root is located in the 2-core is called an *attached tree*.

Time-Dependency. The main difference between time-independent and time-dependent route planning is the travel time function assigned to the edges. In the time-independent scenario, we normally use a positive length function $len : E \to \mathbb{R}^+$, while in a time-dependent scenario, we use travel functions instead of constants for specifying edge weights. Throughout the whole work, we restrict ourselves to a function space $\mathbb{F}$ consisting of positive *periodic* functions $f : \Pi \to \mathbb{R}^+, \Pi = [0, p], p \in \mathbb{N}$ such that $f(0) = f(p)$ and $f(x) + x \leq f(y) + y$ for any $x, y \in \Pi, x \leq y$. Note that these functions respect the FIFO property. In the following, we call Π the *period* of the input. We restrict ourselves to directed graphs $G = (V, E)$ with time-dependent length functions $len : E \to \mathbb{F}$. We use $len : E \times [0, p] \to \mathbb{R}^+$ to evaluate an edge for a specific departure time. Note that our networks fulfill the FIFO property due to our choice of $\mathbb{F}$.

Let $f, g \in \mathbb{F}$ be two travel functions. We say $f < g$ if $f(x) < g(x)$ holds for all $x \in \Pi$. The *composition* of f and g, depicting a travel function for traversing g directly after f, is defined by $f \oplus g := f + (g \circ (f + id))$ with $id(x) = x, x \in \Pi$. Moreover, we need to *merge* functions, which we define by $\min(f, g)$. For more details on these operations, see Section 3. The upper bound of f is noted by $\overline{f} = \max_{x \in \Pi} f(x)$, the lower by $\underline{f} = \min_{x \in \Pi} f(x)$. An underapproximation $\downarrow f$ of a function f is a function such that $\downarrow f(x) \leq f(x)$ holds for all $x \in \Pi$. An overapproximation $\uparrow f$ is defined analogously. Bounds and approximations of our time-dependent edge function len is given by analogous notations. Obviously, one can obtain a time-independent graph $\underline{G}$ from a time-dependent graph G by substituting the time-dependent length function by $\underline{len}$. We call $\underline{G}$ the *lower bound graph* of G.

Paths. A path P in G is a sequence of nodes $(u_1, \ldots, u_k)$ such that $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$. In time-dependent scenarios, the length $\gamma_\tau(P)$ of a path P departing from u_1 at time τ is recursively given by

$$\begin{aligned}\gamma_\tau\big((u_1,u_2)\big) &= len\big((u_1,u_2),\tau\big)\\ \gamma_\tau\big((u_1,\dots,u_j)\big) &= \gamma_\tau\big((u_1,\dots,u_{j-1})\big) + len\Big((u_{j-1},u_j),\gamma_\tau\big((u_1,\dots,u_{j-1})\big)\Big)\end{aligned}$$

In other words, the length of the path depends on a departure time τ from u_1. In a time-dependent scenario, we are interested in two types of distances. On the one hand, we want to compute the shortest path between two nodes for a given departure time. On the other hand, we are also interested in retrieving the distance between two nodes for *all* possible departure times $\in \Pi$. By $d(s,t,\tau)$ we denote the length of a shortest s–t path $\in V$ if departing from s at time τ. The distance label, i.e., the distance between s and t for all possible departure times $\tau \in \Pi$, is given by $d_*(s,t)$. Note that the distance label is a function $\in \mathbb{F}$. In this work, we call a query for determining $d(s,t,\tau)$ an *s-t time-query*, while a query for computing $d_*(s,t)$ is denoted by *s-t profile-query*.

Partitions. A *partition* of V is a family $\mathcal{C} = \{C_0, C_1, \dots, C_k\}$ of sets $C_i \subseteq V$ such that each node $v \in V$ is contained in exactly one set C_i. An element of a partition is called a *cell*. We denote by $\mathbb{c}(u)$ the cell u is assigned to. A *multilevel partition* of V is a family of partitions $\{\mathcal{C}^0, \mathcal{C}^1, \dots, \mathcal{C}^l\}$ such that for each $i < l$ and each $C^i_n \in \mathcal{C}^i$ a cell $C^{i+1}_m \in \mathcal{C}^{i+1}$ exists with $C^i_n \subseteq C^{i+1}_m$. In that case the cell C^{i+1}_m is called the *supercell* of C^i_n. The supercell of a level-l cell is V. We denote by $\mathbb{c}_j(u)$ the level-j cell u is assigned to. The *boundary nodes* B_C of a cell C are all nodes $u \in C$ for which at least one node $v \in V \setminus C$ exists such that $(v,u) \in E$.

2.1 Arc-Flags

The classic arc-flag approach [24, 21] first computes a partition $\mathcal{C}$ of the graph and then attaches a *label* to each edge e. A label contains, for each cell $C_i \in \mathcal{C}$, a flag $AF_{C_i}(e)$ which is `true` if a shortest path to a node in C_i starts with e. A modified DIJKSTRA then only considers those edges for which the flag of the target node's cell is `true`.

Computation of Arc-Flags. Throughout the years, several approaches have been introduced (see e.g. [24, 23, 26, 21, 25]) for computing arc-flags. We here concentrate on an approach which turns out to be the most suited for augmentation. First, the own-cell flags of all edges not crossing borders have to be set to `true`. The own-cell flag of an edge (u,v) is the flag for the region of u and v. If u and v are in different cells, the edge does not have an own-cell flag. Next, a shortest path tree in $\overleftarrow{G}$ is grown from all boundary nodes $b \in B_C$ of all cells C. Then $AF_C(u,v)$ is set to `true` if (u,v) is a tree edge for at least one tree grown from all boundary nodes $b \in B_C$. Note that [21] introduces a faster algorithm for setting flags. It propergates labels of size $|B_C|$ through the network depicting the distances to all boundary nodes of the cell. However, this approach tends to consume a lot of memory making its adaption to a time-dependent scenario impractical.

Multi-Level Arc-Flags. One main disadvantage of uni-directional Arc-Flags is that as soon as the query reaches the target's cell, almost all edges are relaxed as all edges have their own-cell flag set. In huge transportation networks cells can get quite big yielding bad query performance. An approach to remedy this drawback is introduced in [27]: A second layer of arc-flags is computed for each cell. Therefore, each cell is again partitioned into several subcells and arc-flags are computed for each. A Multi-Level Arc-Flags query then first uses the flags on the topmost level and as soon as the query enters the target's cell on the topmost level, the low-level arc-flags are used for pruning. This idea can be extended to a multi-level setup in a straightforward manner.

2.2 Static SHARC-Routing

The main drawback of Arc-Flags is the time-consuming preprocessing which is remedied by SHARC [2, 3] via the integration of a graph contraction scheme [34, 35, 17, 18] into preprocessing. A graph contraction routine removes unimportant nodes and edges from the graph and adds shortcuts to preserve distances between non-removed nodes. The key idea of SHARC is now that we

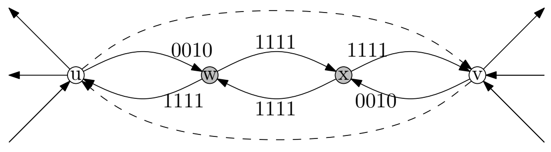


Figure 1: Example for assigning arc-flags during contraction for a partition having four cells. All nodes are in cell 3. The gray nodes (w and x) are removed, the dashed shortcuts are added by the contraction. Arc-flags (edge labels) are indicated by a 1 for `true` and 0 for `false`. The edges directing into the shell get *only* their own-cell flag set `true`. All edges in and out of the shell get all flags assigned to `true`.

can set arc-flags for removed edges automatically. See Figure 1 for an example. Then the expensive computation of arc-flags can be restricted to a much smaller network reducing preprocessing times significantly. Note that we assign suboptimal arc-flags to removed edges, which we can fortunately *refine* as very last step of preprocessing. Here, we propagate arc-flags from important edges to unimportant ones. See Figure 2 for an example. Summarizing, preprocessing of static SHARC is divided into three sections. During the *initialization* phase, we perform a multi-level partition of G. Then, an *iterative* process starts. At each step i we first contract the graph by bypassing unimportant nodes and set the arc-flags automatically for each removed edge. On the contracted graph we compute the arc-flags. In the *finalization* phase, we assemble the output graph and refine arc-flags of edges removed during contraction.

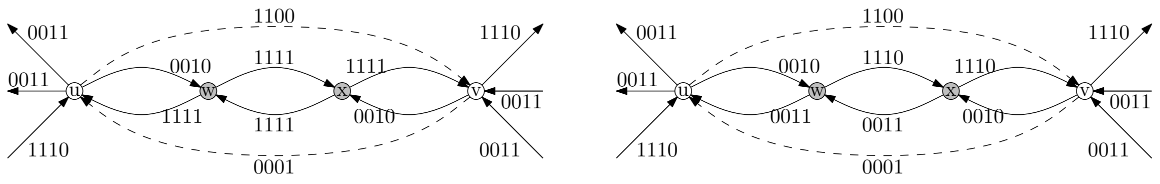


Figure 2: Example for refining arc-flags. The figure in the left shows the graph from Figure 1 after the last iteration step. The figure on the right shows the resulting arc-flags of our refinement routine: the edges (w, x) and (x, v) inherit the flags of the outgoing edge from v, while (x, w) and (w, u) inherit the OR of the flags of the outgoing edges from u.

SHARC adopts the Multi-Level Arc-Flags query. Compared to plain Dijkstra, the modifications are as follows: When settling a node u, we compute the lowest level i on which u and the target node t are in the same supercell. When relaxing the edges outgoing from u, we consider only those edges having a set arc-flag on level i for the corresponding cell of t. The advantages of SHARC over Arc-Flags is two-fold. On the one hand, by limiting computations of arc-flags to important parts of the graph, preprocessing times decrease significantly. On the other hand, by the introduction of shortcuts the number of relaxed edges is reduced. Summarizing, SHARC improves on Arc-Flags with respect to preprocessing effort *and* query performance.

3 Models and Basic Operations

In this work, we focus on route planning in transportation networks. Hence, we here briefly present how to efficiently model road and train networks as graphs. Moreover, we show how to realize the operations link and merge (cf. Section 2) on the travel time functions we use for these models.

3.1 Road Networks

In road networks, we have roads and junctions. We intoduce a node for each junction being connected by a directed edge if there is a direct connection between them. In this work, we use periodic piecewise linear functions for modeling time-dependency in road networks. Each edge gets assigned a number of sample points that depict the travel time on this road at the specific time. Evaluating a function at time τ is then done by linear interpolation between the points left and right to τ.

Composition. In the following, we need to link two piecewise linear functions f, g to $f \oplus g$, modeling the duration for traversing g directly after f. This is done as follows. Let $I(f) = \{(t_1^f, w_1^f), \dots, (t_l^f, w_l^f)\}$ with $t_i^f \in \Pi, w_i^f \in \mathbb{R}^+, 1 \le i \le l$ be the interpolation points of f and $I(g) = \{(t_1^g, w_1^g), \dots, (t_k^g, w_k^g)\}$ those of g. Then the interpolation points $\{\big(t_1^f, w_1^f + g(t_1^f + w_1^f)\big), \dots, \big(t_l^f, w_l^f + g(t_l^f + w_l^f)\big)\}$ are obviously included in $I(f \oplus g)$. However, we also have to add some more interpolation points, namely those from arriving at the timestamps t_j^g of g. More precisely, let $t_1^{-1}, \dots, t_k^{-1}$ be chosen such that $f(t_j^{-1}) + t_j^{-1} = t_j^g$ holds for all $1 \le j \le k$. Then, we also have to add $\{(t_1^{-1} \mod \Pi, f(t_1^{-1}) + w_1^g), \dots, (t_k^{-1} \mod \Pi, f(t_k^{-1}) + w_k^g)\}$ to $I(f \oplus g)$. See Figure 3 for an example. Note that the composed function $f \oplus g$ may have up to $P(f) + P(g)$ number of interpolation points in the worst case, with $P(f) := |I(f)|$ denoting the number of interpolation points of f. Since the linking of functions can be implemented by a sweeping algorithm, the worst-case running time of this operation is in $O(P(f) + P(g))$.

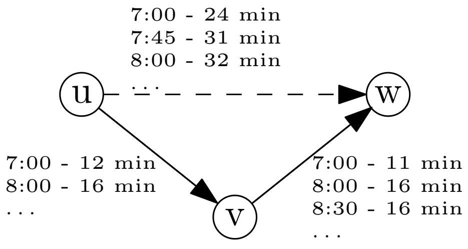


Figure 3: Time-dependent composition in road networks. A function depicting the travel time from u to w via v yields an additional interpolation point at 7:45 because otherwise the interpolated travel time at 7:45 would be 30 minutes instead of 31.

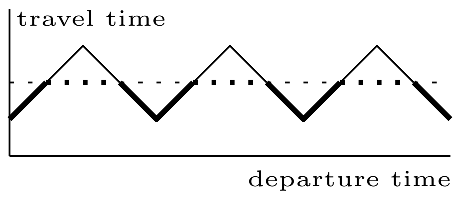
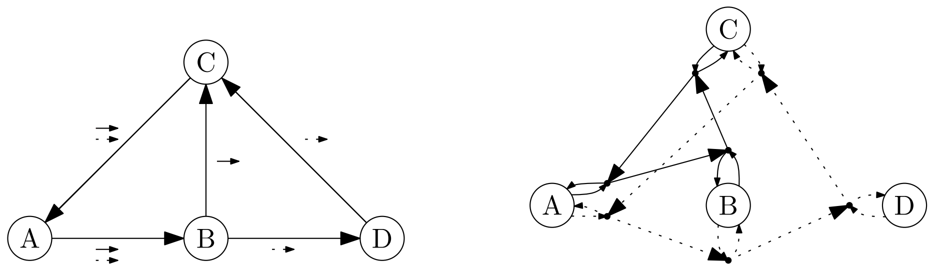


Figure 4: Time-dependent merging of two piecewise linear functions f and g in road networks. f is drawn solid, g dotted, the merged function is drawn thicker. Note that $P(\min(f,g)) > P(f) + P(g)$ holds.

Merging. We also have to merge two piecewise linear functions f, g to $\min(f, g)$. Like for linking, this may increase the number of breakpoints. More precisely, we have to check for all timestamps t_i^f of f whether $w_i^f < g(t_i^f)$ holds. If it holds, we need to keep the interpolation point (t_i^f, w_i^f), otherwise we do not need it. Analogously, we proceed for all timestamps t_j^g of g. However, additional interpolation points have to be added for all intersection points of f and g. Figure 4 gives an example. Note that the worst-case running time of this operation is in $O(P(f) + P(g))$.

3.2 Railways

The straightforward approach [4] for railway networks is to model each station by a single node, a time-dependent edge is inserted if a direct connection between two stations exist. See Figure 5(a) for a small example. Then, several weights are assigned to each edge. For each train, we add an interpolation point to the corresponding edge. A problem of this approach is that transfer times cannot be incorporated properly. To do so, a time-dependent *train-route model [31]* has to be applied. For each train route, a node is introduced at each station the trains of this route stop. These nodes are connected by time-dependent edges modeling the trains running on this route. For each station, a supernode is introduced which is connected to all route nodes of this station modeling the transfer from one train to another. See Figure 5(b) for an example. In this work, we use the train-route model and guarantee the FIFO property of such graphs by introducing multi-edges.



(a) Simple time-dependent model. (b) Realistic time-dependent model.

Figure 5: Time-dependent railway graphs. For both models, we have four stations and two train routes. The first one runs from A to B to C and back to A, while the second one runs A→B→C→D→A. In the simple model, switching from one train to another can be done in 0 minutes, while transfer times are incorporated correctly in the realistic model.

When we want to evaluate a time-dependent edge at a specific time τ, we identify the interpolation point (t_i, w_i) with minimum $t_i - \tau \geq 0$. Then the resulting traveltime is $w_i + t_i - \tau$, i.e., the waiting time for the next connection plus its travel duration.

Composition. Interestingly, the composition of two timetable edge-functions f, g is less expensive than in road networks. More precisely, $P(f \oplus g) \leq \min\{P(f), P(g)\}$ holds as the number of relevant departure times is dominated by the edge with less connections. More precisely, we determine for each interpolation point $(t_i^f, w_i^f) \in I(f)$ the so called *connection interpolation point* of g, which is the point $(t_j^g, w_j^g) \in I(g)$ with $t_j^g - t_i^f - w_i^f \geq 0$ minimal. In other words, this is the first connection of g we can catch when taking the connection departing at timestamp t_i^f. Then, we add the interpolation point $(t_i^f, (t_j^g - t_i^f + w_j^g))$ to $I(f \oplus g)$. Since f and g are FIFO functions this ensures correctness. However, it may happen that two interpolation points $\in I(f)$ yield the same connection point of g. In such a situation we only need to keep the point with maximal timestamp since an earlier departure does not pay off. See Figure 6 for an example. Although the number of interpolation points may decrease, this operation takes $O(P(f) + P(g))$ time in the worst case.

Merging. The merging of two public transportation functions is straightforward. For each timestamp t_i^f of f we check whether $w_i^f < g(t_i^f)$ holds. If it holds, we add (t_i^f, w_i^f) to $I(\min(f, g))$. We do the same for all timestamp of g. Note that in the worst case, $\min(f, g)$ may have up to $P(f) + P(g)$ interpolation points. See Figure 7 for an example. Note that the merging of timetable functions is also cheaper than the merging of functions used for road networks, but still in $O(P(f) + P(g))$.

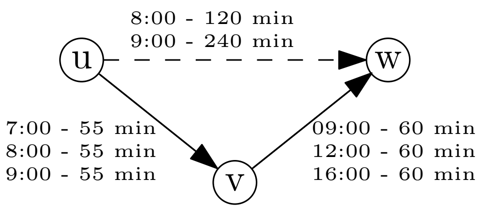


Figure 6: Time-dependent composition in public transportation networks. A function depicting the travel time from u to w via v yields less interpolation points than both functions constructed it is from.

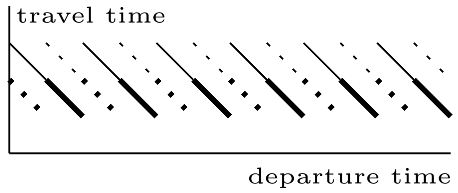


Figure 7: Time-dependent merging of two public transportation functions f and g, f is drawn solid, g dotted, the merged function is drawn thicker.

4 Augmenting Ingredients

The main ingredients of the static SHARC preprocessing are (local) Dijkstra-searches, Arc-Flags and contraction (cf. Section 2). So, in order to augment SHARC to time-dependency, we first need to augment all these ingredients such that correctness is guaranteed in a time-dependent scenario, which we do in this section.

4.1 Dijkstra

Computing $d(s,t,\tau)$ can be solved by a modified Dijkstra [5]: when relaxing an edge (u,v) we have to evaluate its weight for departure time $\tau + d(s,u,\tau)$. In our scenario, the running time for evaluating functions is negligible, hence the additional effort for respecting the departure time is negligible as well. However, computing $d_*(s,t)$ is more expensive but can to be computed by a label-correcting algorithm [6]. Such an algorithm can be implemented very similarly to Dijkstra. The source node s is initialized with a constant label $d_*(s,s) \equiv 0$, any other node u with a constant label $d_*(s,u) \equiv \infty$. Then, in each iteration step, a node u with minimum $\underline{d_*}(s,u)$ is removed from the priority queue. Then for all outgoing edges (u,v) a temporary label $l(v) = d_*(s,u) \oplus len(u,v)$ is created. If $l(v) \geq d_*(s,v)$ does *not* hold, $l(v)$ yields an improvement. Hence, $d_*(s,v)$ is updated to $\min\{l(v), d_*(s,v)\}$ and v is inserted into the queue. We may stop the routine if we remove a node u from the queue with $\underline{d}(s,u) \geq \overline{d}(s,t)$. If we want to compute $d_*(s,t)$ for many nodes $t \in V$, we apply a label-correcting algorithm and stop the routine as soon as our stopping criterion holds for all t. Note that we may reinsert nodes into the queue that have already been removed by this procedure. Also note that when applied to a graph with constant edge-functions, this algorithm equals a normal Dijkstra.

An interesting result from [6] is the fact that the worst-case runtime of such a label-correcting algorithm is $O(nmf^*)$, where f^* denotes $\max_{u \in V} = P(d_*(s,u))$. Note that the increase from $\log n$ to n (compared to a static Dijkstra) derives from the fact that we lose the label-setting property. Fortunately, the number of multiple insertions of nodes stays small in road networks (cf. Table 3 in Section 6). Still, the runtime of this algorithm is dominated by the complexity of the constructed distance labels. In fact, these labels can even grow exponentially for worst-case inputs [6]. Such exponential growth may happen if the number of different shortest paths (for varying τ) between two nodes is exponential. Fortunately, it turns out that in transportation networks, such situations do not arise. Hence, the usage of this algorithm is still practical as our experiments confirm.

In the following, we construct *profile graphs (PG)*, i.e., compute $d_*(s,u)$ for a given source s and all nodes $u \in V$, with our label-correcting algorithm. We call an edge (u,v) a *PG-edge* if $d_*(s,u) \oplus (u,v) > d_*(s,v)$ does *not* hold. In other words, an edge (u,v) is a PG-edge (with respect to s) if it is part of a shortest path from s to v for at least one departure time.

4.2 Arc-Flags

In time-independent scenarios, a set arc-flag $AF_C(e)$ denotes whether e has to be considered for a shortest-path query targeting a node within C. In other words, the flag is set if e is important for (at least one target node in) C. In a time-dependent scenario, we use the following intuition to set arc-flags: an arc-flag $AF_C(e)$ is set to `true`, if e is important for C at least once during Π. A straightforward adaption of computing arc-flags in a time-dependent graph is to construct a profile graph in $\overleftarrow{G}$ for all boundary nodes $b \in B_C$ of all cells C. Then we set $AF_C(u,v) =$ `true` if (u,v) is a PG-edge for at least one PG built from all boundary nodes $b \in B_C$. See Figure 8 for an example. In addition, we also set all own-cell flags to `true`. The time-dependent query is a normal time-dependent DIJKSTRA only relaxing edges with set flag for the target's region.

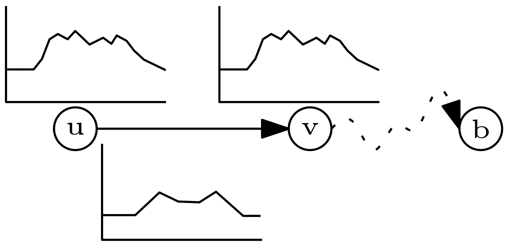


Figure 8: Computation of time-dependent arc-flags. By construction of a profile graph from the boundary node b, we end up in two distance labels $d_*(u,b)$ and $d_*(v,b)$. If $len(u,v) \oplus d_*(v,b) > d_*(u,b)$ does not hold, (u,v) is a PG edge (with respect to b) and hence gets the arc-flag for $\mathbb{c}(b)$ assigned `true`.

Lemma 1 *Time-dependent Arc-Flags is correct.*

Proof. To show correctness of time-dependent Arc-Flags, we have to prove that for each shortest s–t path $p^\tau_{st} = (e_0,\dots,e_k), \tau \in \Pi$ the following condition holds: $AF_T(e_i) =$ `true`$, 0 \le i \le k$ with $T = \mathbb{c}(t)$. For all edges $e_i = (u_i,v_i)$ with $\mathbb{c}(u_i) = \mathbb{c}(v_i) = \mathbb{c}(t)$ this holds because we set own-cell flags to `true`.

Let s and t be arbitrary nodes, and let τ be an arbitrary departure time. In addition, let $e_i = (u_i,v_i) \in p^\tau_{st}, \mathbb{c}(u_i) \ne \mathbb{c}(t), \mathbb{c}(v_i) \ne \mathbb{c}(t)$, and b_T be the last boundary node of region T on p^τ_{st}. We know that the subpath from s to b_T is a shortest path (for departure time τ). Assume $AF_T(e_i) =$ `false`. Since $AF_T(e_i) =$ `false` holds, $d_*(u_i,b_i) \oplus (u_i,v_i) > d_*(v_i,b_i)$ must hold as well. This is a contradiction since e_i is part of the shortest path from s to b_T. □

Approximation. Computing arc-flags as described above requires building a complete profile graph on the backward graph from each boundary node, which yields too long preprocessing times for large networks. Recall that the running time of building PGs highly depends on the complexity of the edge functions. Hence, we may construct two PGs for each boundary node; the first uses $\uparrow len$ as length functions, the second $\downarrow len$. As we use approximations with a constant number of interpolation points, constructing two such PGs is faster than building a single exact one. We end up in two distance labels per node u, one being an overapproximation, the other one being an underapproximation of the correct label. Then, for each $(u,v) \in E$, we set $\overline{AF}_C(u,v) =$ `true` if $\downarrow len(u,v) \oplus \downarrow d_*(v,b_C) > \uparrow d_*(u,b_C)$ does not hold. See Figure 9 for an example.

If networks get so big that even computing approximate labels is prohibitive due to running times, one can even use upper and lower bounds for the labels. This has the advantage that building two shortest-path trees per boundary node is sufficient for setting correct arc-flags. The first uses $\overline{len}$ as length function, the other $\underline{len}$. See Figure 10 for an example. Note that by approximating arc-flags (denoted by $\overline{AF}$), their quality may decrease but correctness is untouched. Thus, queries remain correct by may become slower.

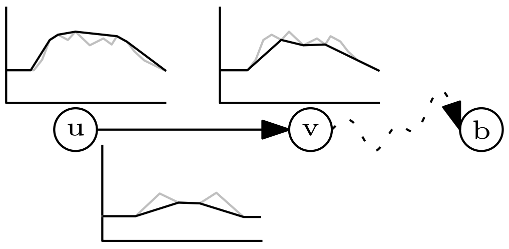


Figure 9: Approximation of time-dependent arc-flags via functions. The original functions are drawn in gray. By using over- and underapproximations of *len* during construction of the profile graphs, we end up in approximated distance labels for u and v. Then, we set the arc-flag of (u,v) to `true` if $\downarrow len(u,v) \oplus \downarrow d_*(v,b_C) > \uparrow d_*(u,b_C)$ does not hold.

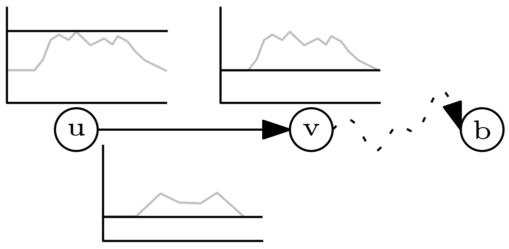


Figure 10: Approximation of time-dependent arc-flags via bounds. The original functions are drawn in gray. Unlike for approximation via functions, we use bounds for approximations. We set the arc-flag of (u,v) to `true` if $\underline{len(u,v)} + \underline{d_*(v,b_C)} \le \overline{d_*(u,b_C)}$ holds.

Lemma 2 *Approximate Arc-Flags is correct.*

Proof. We have to show that $AF(e) = \texttt{true} \Rightarrow \overline{AF}(e) = \texttt{true}$ holds for all edges $e = (u,v)$. Assume $AF(e) = \texttt{true}$ and $\overline{AF}(e) = \texttt{false}$. Let b be the boundary node for which e is a PG-edge. Since $AF(e) = \texttt{true}$, we known that $len_\tau(u,v) + d_{\tau+len_\tau(u,v)}(v,b) = d_\tau(u,b)$ holds for at least one departure time. Since $\overline{AF}((u,v)) = \texttt{false}$, $\downarrow len(u,v) \oplus \downarrow d_*(v,b) > \uparrow d_*(u,b)$ must hold as well. From this follows that τ, $len_\tau(v,u) + d_{\tau+len_\tau(v,u)}(u,b) > d_\tau(v,b)$ must hold, which is a contradiction to $AF(e) = \texttt{true}$. □

Heuristic Arc-Flags. Analyzing both approaches for computing arc-flags, exact and approximated, we observe the following. Exact flags yield excellent query times (cf. Section 6) but preprocessing is time-consuming, while approximated flags yield lower preprocessing times but query performance is much worse than for exact flags.

Hence, we propose a third approach for computing flags. Unfortunately, we cannot guarantee correctness but experiments show that in road networks, errors are very small. The preprocessing is as follows: We grow $K+2$ shortest-path trees from each boundary node, the first uses $\underline{len}$ as metric, the second one $\overline{len}$. The remaining K trees are time-queries in $\overleftarrow{G}$ using a fixed arrival time at the boundary node. We set a flag of an edge for a region C if the edge is part of at least one shortest path tree grown from the boundary nodes of C.

As already mentioned, this approach may yield incorrect queries as a shortest path for a specific departure time may have been missed. However, it is obvious that a path is found since at least for one departure time, flags are set to `true` for a shortest path to the target's region. We evaluate the error-rate in Section 6.

Multi-Level Arc-Flags

Preprocessing the multi-level extension of Arc-Flags in a time-dependent scenario is done as follows. Arc-flags on the upper level are computed as described above. For the lower flags, we construct a PG for all boundary nodes b on the lower level. We may stop the construction as soon as $\underline{d_*}(u,b) \geq \overline{d_*}(v,b)$ holds for all nodes v in the supercell of C and all nodes u in the priority queue. Then, we set an arc-flag to `true` if the edge is a PG-edge of at least one PG. Note that we can apply our alternative arc-flag setting strategies (approximate and heuristic) to low-level flags as well.

Lemma 3 *Time-Dependent Multi-Level Arc-Flags is correct.*

Proof. In the following, we show the correctness of a two-level setup. The generalization to a multi-level scenario is straightforward. Let $p_{st}^{\tau} = (e_0, \dots, e_k), \tau \in \Pi$ be an arbitrary s–t shortest path with arbitrary departure time τ. Let $\mathbb{c}_i(u)$ be the cell of u in level i, where 0 denotes the lower, 1 the upper level. An edge (u,v) is part of the upper level if $\mathbb{c}_1(u) \neq \mathbb{c}_1(t)$ and $\mathbb{c}_1(v) \neq \mathbb{c}_1(t)$. According to Lemma 1, we know that all edges being part of the upper level have $AF_{\mathbb{c}_1(t)} = \texttt{true}$. Let b be the last boundary node of $\mathbb{c}_0(t)$ on p_{st}^{τ}. Since we have built a profile graph from b during preprocessing until all nodes in $\mathbb{c}_1(t)$ have their final label assigned, edges being part of the lower level have proper arc-flags assigned. □

4.3 Contraction

Our time-dependent contraction routine is very similar to a static one [34, 35, 18, 2]. First we reduce the number of nodes by removing unimportant ones and—in order to preserve distances between non-removed nodes—add time-dependent shortcuts to the graph. Then, we apply an edge-reduction step that removes unneeded shortcuts.

Node-Reduction. We reduce the number of nodes by iteratively *bypassing* nodes until no node is *bypassable* any more. To bypass a node u we first remove u, its incoming edges I and its outgoing edges O from the graph. Then, for each $v \in \text{tail}(I)$ and for each $w \in \text{head}(O) \setminus \{v\}$ we introduce a new edge of the length $len(v,u) \oplus len(u,w)$. In the following, we will see that allowing multi-edges eases unpacking shortcuts since each shortcut represents exactly one path. We call the number of edges of the path that a shortcut represents on the graph before the node-reduction the *hop number* of the shortcut.

The order in which nodes are bypassed changes the resulting contracted graph. Hence, we use a heap to determine the next bypassable node. Therefore, we first determine the number *#shortcut* of edges that would be inserted into the graph if u was bypassed. However, we do not count existing edges connecting nodes in tail(I) with nodes in head(O). Let $\zeta(u) = \#shortcut / (\deg_{in}(u) + \deg_{out}(u))$ be the *expansion* [18] of a node u. Furthermore, let $h(u)$ be the hop number of the hop-maximal shortcut, and let $p(u)$ be the number of interpolation points of the shortcut with most interpolation points, that would be added if u was bypassed. Then we set the key of a node u within the heap to $h(u) + p(u) + 10 \cdot \zeta(u)$, smaller keys having higher priority. By this ordering for bypassing nodes we prefer nodes whose removal yield few additional shortcuts with a small hop number and few interpolation points.

We *stop* the node-reduction as soon as we would bypass a node u with an expansion $\zeta(u) > C$, with C called the expansion threshold. Moreover, to keep the costs of shortcuts limited we do not bypass a node if its removal would either result in a shortcut with more than I interpolation points or a hop number greater than H. We say that the nodes that have been bypassed belong to the *shell*, while the remaining nodes are called *core nodes.*

Corollary 1 *Time-dependent node-reduction keeps distances (for all departure times) between core nodes correct.*

Proof. Correctness follows directly from our rules of adding shortcuts. □

Edge-Reduction. The second edge-reduction step for time-dependent networks is also similar to a the static one: We build a profile graph (instead of a shortest path tree) from each node u of the core. We stop the construction as soon as all neighbors v of u have their final label assigned. Then we check for all neighbors whether $d_*(u, v) < len(u, v)$ holds. If it holds, we can remove (u, v) from the graph because for all possible departure times, the path from u to v does not include (u, v). As already mentioned, building profile graphs is very time-consuming. Hence, we limit the running time of this procedure by restricting the number of priority-queue removals to 20.

Since building even such very limited profile graphs can get very time-consuming, we apply a *bounded* edge-reduction step directly before. We grow two shortest path trees from u, one uses $\overline{len}$ as length function, the other one $\underline{len}$. Here, we may stop the growth as soon as all outgoing neighbors v of u have been settled. If $\overline{d_*(u, v)} < \underline{len(u, v)}$ holds, we can safely remove (u, v) from the graph.

Corollary 2 *Time-dependent edge-reduction keeps distances (for all departure times) between core nodes correct.*

Proof. Correctness follows directly from our rules of removal. □

Discussion. Time-dependent contraction in road networks is space-consuming. Each added shortcut increases the total number of interpolation points of the graph (cf. Section 3). It turns out that this increase in number of interpolation points is one of the main problems when routing in time-dependent road networks: Almost all speed-up techniques developed for static scenarios rely on adding long shortcuts to the graph. While this is "cheap" for static scenarios, the insertion of time-dependent shortcuts yields a high amount of preprocessed data. So, unlike in time-independent scenarios, the degree of contraction must be chosen carefully to keep the number of shortcuts as low as possible.

5 Time-Dependent SHARC

With our ingredients augmented, we are now ready to augment SHARC itself. It turns out that the additional effort for augmentation is quite small since we may leave the basic concept of SHARC untouched.

5.1 Preprocessing

During the *initialization* phase, we remove the 1-shell nodes from the graph since we can directly assign correct arc-flags to all edges adjacent to 1-shell nodes: Edges targeting the 2-core get full flags assigned, those directing away from the 2-core get only the own-cell flag set to `true`. Note that this procedure is independent from edge weights. After extracting the 2-core, we perform a multi-level partitioning of the *unweighted* graph. The partition has to fulfill several requirements: cells should be connected, the size of cells should be balanced, and the number of boundary nodes should be as low as possible. Like in [2], we obtain such a partition by local optimization of a partition obtained from SCOTCH [30].

After the initialization, an *iterative* process starts, consisting of two phases: contraction and arc-flag computation. We apply a contraction step according to Section 4. However, in order to guarantee correctness, we have to use *cell-aware* contraction, i.e., a node u is never marked as bypassable if any of its neighboring nodes is *not* in the same cell as u. Next, we assign arc-flags to all edges of our output graph, including those which we remove during contraction. Like for static

SHARC, we can set arc-flags for all removed edges automatically. We set the arc-flags of the current and all higher levels depending on the tail v of the deleted edge. If v is a core node, we only set the own-cell flag to `true` (and others to `false`) because this edge can only be relevant for a query targeting a node in this cell. If v belongs to the shell, all arc-flags are set to `true` as a query has to leave the shell in order to reach a node outside this cell. See Figure 1 for an example. Setting arc-flags of those edges not removed from the graph is more expensive since we apply one of the preprocessing techniques for Multi-Level Arc-Flags from Section 4.

The *final* phase of our preprocessing-routine assembles the output graph. It contains the original graph, shortcuts added during preprocessing and arc-flags for all edges of the output graph. However, some edges may have no arc-flag set to `true`. Since these edges are never relaxed by our query algorithm, we can remove them from the output graph.

5.2 Query

Time-dependent SHARC allows time- and profile-queries. For computing $d(s,t,\tau)$, we use a modified DIJKSTRA that operates on the output graph. The modifications are as follows: When settling a node u, we compute the lowest level i on which u and the target node t are in the same supercell. Note that level i is the lowest level in which u and t are in different cells. Moreover, we consider only those edges outgoing from u having a set arc-flag on level i for the corresponding cell of t. In other words, we *prune* edges that are not important for the current query. We stop the query as soon as we settle t. See Figure 11 for an example query.

For computing $d_*(s,t)$, we use a modified variant of our label-correcting algorithm (see Section 4) that also operates on the output graph. The modifications are the same as for time-queries and the stopping criterion is the standard one explained in Section 4.

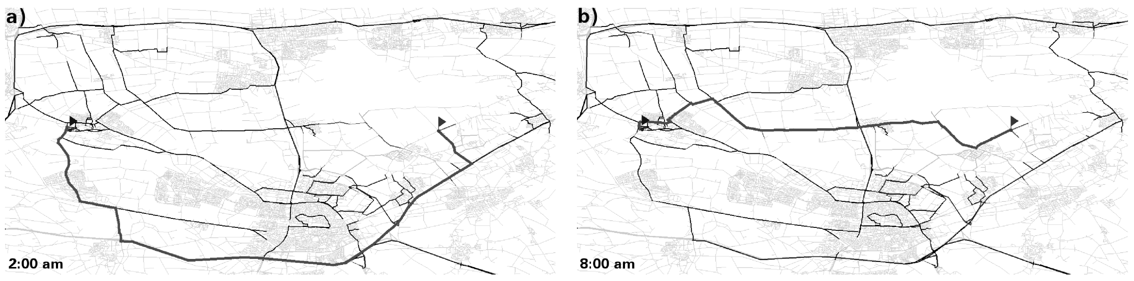


Figure 11: Two examples for time-dependent queries with different departure times (but identical source and target) yielding different quickest paths. The input is a road network with congested motorways. The source of the query is marked by the flag on the left, the target by the one on the right. The quickest paths are drawn thicker. Roads touched by SHARC are black. Note that the edges touched are almost independent of the departure time. Also note that for a nighttime departure (left), it pays off to use the highway (lower route) while for a departure during rush hours, the quickest path is across fields without using highways.

Outputting Shortest Paths. SHARC adds shortcuts to the graph in order to accelerate queries. If the complete description of the path is needed, the shortcuts have to be unpacked. Since we allow multi-edges during contraction, each shortcut represents exactly one path in the network, and hence, we can directly apply our unpacking routine for static SHARC, which, in turn, is adapted from Highway Hierarchies [10].

5.3 Correctness

Before proving correctness of SHARC, we first have to show that the following lemma holds. We denote by G_i the graph after iteration step i, $i = 1, \ldots, L$. By G_0 we denote the input graph G. The level $l(u)$ of a node u is defined to be the integer i such that u is contained in G_i but not in G_{i+1}. We further define the level of a node contained in G_L to be L. Note that the proof of correctness is very similar to the one presented in [3]. Still, we include it here for self-containedness.

Lemma 4 *Let s and t be arbitrary nodes in G such that there is a path from s to t in G_0. Let τ be an arbitrary departure time. At each step i of the SHARC preprocessing there exists a shortest s-t-path $p^\tau_{st} = (v_1, \ldots, v_{j_1}; u_1, \ldots, u_{j_2}; w_1, \ldots, w_{j_3})$, $j_1, j_2, j_3 \in \mathbb{N}_0$, in $\bigcup_{k=0}^{i} G_k$, such that*

- $l(v_1), \ldots, l(v_{j_1}), l(w_1), \ldots, l(w_{j_3}) < i$,
- $l(u_1), \ldots, l(u_{j_2}) \geq i$
- $\mathbb{c}_i(u_{j_2}) = \mathbb{c}_i(t)$
- *for each edge e of p^τ_{st}, the arc-flags assigned to e until iteration i allow the path p^τ_{st} to t.*

We use the convention that $j_r = 0,\ r \in \{1, 2, 3\}$ means that the corresponding subpath has no edges.

In other words, during each step of the preprocessing, the lemma assures that there exists a path in the output graph such that at the beginning, the levels of nodes monotonically increase and the end, levels monotonically decrease again. Moreover, arc-flags are set in such a way that the according flag is set to true.

Proof. We prove Lemma 4 by induction on the iteration steps. Since Lemma 1 holds, the claim holds trivially for $i = 0$. The inductive step works as follows: Assume the claim holds for step i. Let s and t be arbitrary nodes, for which there is a path from s to t in G_0. We denote by $p^\tau_{st} = (v_1, \ldots, v_{j_1}; u_1, \ldots, u_{j_2}; w_1, \ldots, w_{j_3})$ the s-t-path according to the lemma for step i.

Let $p_m = (u_1, \ldots, u_k)$ be the corresponding sub-path of p^τ_{st} at iteration step i. The iteration step $i + 1$ consists of the contraction phase and the arc-flag computation. Since the latter does not violate the lemma, it remains to be shown that after each contraction step, Lemma 4 holds.
There exists a maximal path $(u_{\ell_1}, u_{\ell_2}, \ldots, u_{\ell_d})$ with $1 \leq \ell_1 \leq \ldots \leq \ell_d \leq k$ for which

- for each $f = 1, \ldots, d - 1$ either $\ell_f + 1 = \ell_{f+1}$ or the subpath $(u_{\ell_f}, u_{\ell_f+1}, \ldots u_{\ell_{f+1}})$ has been replaced by a shortcut,
- the nodes $u_1, \ldots, u_{\ell_1-1}$ have been deleted, if $\ell_1 \neq 1$ and
- the nodes $u_{\ell_d+1}, \ldots, u_k$ have been deleted, if $\ell_d \neq k$.

The contraction routine guarantees that:

- $(u_{\ell_1}, u_{\ell_2}, \ldots, u_{\ell_d})$ is also a shortest path, see Corollary 1.
- u_{ℓ_d} is in the same shell as u_k in all levels greater than i (because of cell aware contraction)
- the deleted edges in $(u_1, \ldots, u_{\ell_1-1})$ either already have their arc-flags for the path p^τ_{st} assigned (case `a`) or the nodes $u_1, \ldots, u_{\ell_1-1}$ are in the shell (case `b`). In case `a`, the arc-flags are correct because of the induction hypothesis. In case `b`, all arc-flags for all higher levels are assigned `true`.
- the deleted edges in $(u_{\ell_d+1}, \ldots, u_k)$ either already have their arc-flags for the path p^τ_{st} assigned or $u_{\ell_d+1}, \ldots, u_k$ are in the same shell (due to cell-aware contraction) as t for all levels $\geq i$. Since the own-cell flag always is set `true` for deleted edges the path stays valid.

According to Corollaries 1 and 2, distances are preserved during preprocessing. Hence, for arbitrary i, $0 \leq i \leq L$ a shortest path in G_i is also a shortest path in $\bigcup_{k=0}^{L} G_k$. Concluding, the path $\hat{p}_{st}^{\tau} = (v_1, \ldots, v_{j_1}, u_1, \ldots, u_{\ell_1 - 1};\ u_{\ell_1}, u_{\ell_2}, \ldots, u_{\ell_d};\ u_{\ell_d + 1}, \ldots, u_k, w_1, \ldots, w_{j_3})$ fulfills all claims of the lemma for iteration step $i + 1$. □

The lemma guarantees that arc-flags are set properly at each iteration. With this lemma at hand, we are finally ready to prove correctness of time-dependent SHARC.

Theorem 1 *The distances computed by time-dependent SHARC are correct.*

Proof. Lemma 4 holds during all phases of all iteration steps of SHARC preprocessing. So, together with Lemma 3 the preprocessing algorithm is correct. □

5.4 Optimizations

Although SHARC as described above already yields a low preprocessing effort combined with good query performance, we use some optimization techniques to reduce preprocessing effort (time and space consumption) and to increase query performance.

Refinement of Arc-Flags. During the iteration-phase we set sub-optimal arc-flags to edges originating from shell nodes. However, we can do better: we are able to *refine* arc-flags by *propagation* of arc-flags from higher to lower levels. Recall that the level $l(u)$ of a node u is determined by the iteration step it is removed from the graph. All nodes removed during iteration step i belong to level i. Those nodes which are part of the core-graph after the last iteration step belong to level L. In the following, we explain our propagation routine for a given node u.

First, we construct a partial profile graph T starting from u. When settling a node v, we do not relax edges whose heads are on a level smaller than $l(v)$. Moreover, we add a settled node v to a set $\vec{N}(u)$ if $l(v) > l(u)$ holds *and* a predecessor—with respect to T—of the node v is in level $l(u)$. In the following, we call $\vec{N}(u)$ the *exit nodes* of u. For the stopping criterion we need the notion of covered nodes. A node v is called covered as soon as all its predecessors—with respect to T—belong to a level $> l(u)$. We stop the construction of T as soon as all nodes in the priority queue are covered and the minimum key in the priority queue is greater than $\max_{v \in \vec{N}(u)} \overline{d_*}(u, v)$. This ensures that all exit nodes of u have been discovered and that they have their final distance labels assigned.

Once this partial profile graph is built we are ready to refine the arc-flags of all edges outgoing from u. Therefore, we assign exit nodes to outgoing edges from u. Starting at an exit node n_E we follow the predecessors in T until we finally end up in a node x whose predecessor is u. The edge (u, x) now inherits the flags from n_E. Every edge outgoing from n_E whose head v is *not* an exit node of u and *not* in a level $< l(u)$ propagates all `true` flags of all levels $\geq l(u)$ to (u, x). See Figure 2 for an example. In order to propagate flags from higher to lower levels we perform our propagation-routine in $L - 1$ refinement steps, starting at level $L - 1$ and in descending order. Note that during refinement step i we only refine arc-flags of edges outgoing from nodes belonging to level i. Also note that in profile graphs, a node may have more than one predecessor. In order to preserve correctness, we have to follow each predecessor until we reach u.

As already mentioned, constructing PGs is time-consuming. Hence, we limit the growth of those graphs to $n \log(n)/|V_l|$ priority-queue removals, where V_l denotes the nodes in level l. In order to preserve correctness, we then may only propagate the flags from the exit nodes to u if the stopping criterion is fulfilled before this number of removals.

Lemma 5 *Refinement of Arc-Flags is correct.*

Proof. Recall that the own-cell flag does not get altered by the refinement routine. Hence, we only have to consider flags for other cells. Assume we perform the propagation routine at a level l to a level-l node u. A shortest path p_{ut}^{τ} from u to a node t in another cell on level $\geq l$ needs to contain a level $> l$ node that is in the same cell as u because of the cell-aware contraction. Moreover, with iterated application of Lemma 4 we know that there must be an (arc-flag valid) shortest s-t-path for which the sequence of the levels of the nodes first is monotonically ascending and then monotonically descending. In fact, to cross a border of the current cell at level l, at least two level $> l$ nodes are on p_{ut}^{τ}. We consider the first level $> l$ node u_1 on p_{ut}^{τ}. This must be an exit node of u. The node u_2 after u_1 on p_{ut}^{τ} is covered and therefore it is no exit node. Furthermore, it is in a level $> l$. Hence, the flags of the edge (u_1, u_2) are propagated to the first edge on p_{ut}^{τ} and thus, Lemma 4 holds also after refinement which proves that the refinement phase is correct. □

Shortcut-Removal. As already discussed in Section 4, time-dependent shortcuts are very space-consuming. Hence, we try to remove shortcuts as the very last step of preprocessing. The routine works as follows. For each added shortcut (u, v) we analyze the path $p_{uv} = (u, u_0, \ldots, u_k, v)$ it represents. If $\deg_{out}(u_i) \leq 3$ holds for all $0 \leq i \leq k$, we remove (u, v) from the graph and the edge (u, u_0) additionally inherits the arc-flags from (u, v).

Arc-Flag Compression. In order to reduce space consumption of SHARC, we compress the arc-flag information. During our studies, we observed that the number of unique arc-flags sets is much less than the number of edges. Thus, instead of storing the arc-flags directly at each edge, we use a separate table containing all possible unique arc-flags sets. In order to access the flags efficiently, we assign an additional pointer to each edge indexing the correct arc-flags set in the table. This reduces the space consumption for storing the arc-flags data by a factor of ≈ 5. Note however, that the main space overhead for time-dependent SHARC stems from time-dependent shortcuts.

Improved Locality. In order to improve query performance, we increase—similar to [18]—cache efficiency of the output graph by reordering nodes according to the level they have been removed from the graph. As a consequence, the number of cache misses is reduced yielding lower query times.

Landmarks. SHARC harmonizes with A^* search. The idea of goal-directed or A^* search is to push the search towards the target. By adding a potential $\pi : V \to \mathbb{R}$ to the priority of each node, the order in which nodes are removed from the priority queue is altered. In order to preserve correctness, the potential has to be *feasible* [20], i.e., $\mathit{len}(u, v) - \pi(u) + \pi(v) \geq 0$ holds for all $(u, v) \in E$. The ALT algorithm [16, 19] uses a small number of nodes—so called *landmarks*—and the triangle inequality to compute *lower bounds* on the distance to the target. Given a set $L \subseteq V$ of landmarks and distances $d(l_i, v), d(v, l_i)$ for all nodes $v \in V$ and landmarks $l_i \in L$, the following triangle inequalities hold:

$$d(l_1, u) + d(u, v) \geq d(l_1, v) \quad \text{and} \quad d(u, v) + d(v, l_2) \geq d(u, l_2)$$

See Figure 12 for an illustration. Therefore, $lb(u, v) := \max_{l \in L} \max\{d(u, l) - d(v, l), d(l, v) - d(l, u)\}$ provides a lower bound for the distance $d(u, v)$. It turns out that these lower bounds are feasible potentials. The quality of the lower bounds highly depends on the quality of the selected landmarks. Interestingly, undirectional ALT can be used without further adaption if landmark-distances are computed using $\underline{\mathit{len}}$ as metric [11].

Finding good landmarks is difficult. In this work, we use landmark selection with *avoid* [16]. This heuristic tries to identify regions of the graph that are not well covered by the current landmark

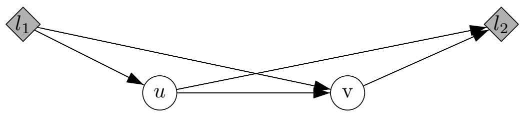


Figure 12: Triangle inequalities for landmarks. The landmarks are l_1 and l_2.

set L. The routine first grows a shortest-path tree T_r from a random node r. The *weight* of each node v is the difference between $d(v, r)$ and the lower bound $\underline{d}(v, r)$ obtained by the given landmarks. The *size* of a node v is defined by the sum of its weight and the size of its children in T_r. If the subtree of T_r rooted at v contains a landmark, the size of v is set to zero. Starting from the node with maximum size, T_r is traversed following the child with highest size. The leaf obtained by this traversal is added to L.

We can combine ALT with SHARC easily. We run a time-dependent ALT preprocessing consisting of selecting landmarks $L \subseteq V$ and computing $\underline{d_*(l, v), d_*(v, l)}$ (in $\underline{G}$) for all $v \in V, l \in L$. Then, we apply a normal SHARC query but use $d(s, \overline{u, \tau) + l}\underline{b(u)}$ instead of $d(s, u, \tau)$ as priority key. We call this combination L-SHARC (**L**andmarks and **SHARC**).

5.5 Comparison to Static SHARC

Comparing our new time-dependent preprocessing routine with the one for static SHARC [2, 3], one may notice that the concept itself stays untouched. We still apply three phases: initialization, iteration, and finalization. The initialization stays untouched as both removal of 1-shell nodes and partitioning are performed on the unweighted graph. However, the iteration process gets more expensive both in terms of memory consumption and preprocessing times: The higher memory consumption is due to the fact that time-dependent shortcuts use more space than static ones, while the longer preprocessing times are due to the more complex algorithms for setting arc-flags.

The finalization is also altered slightly. On the one hand, we again use a label-correcting algorithm instead of Dijkstra for arc-flag refinement, which makes this procedure more time-consuming. Unlike in [3], we limit the effort for refinement by limiting the number of heap operations. This yields faster preprocessing times but the quality of arc-flags is worse than it could be. The high memory consumption of shortcuts is the reason why we introduce a new routine for removing shortcuts from the output graph. Note that we could directly use our time-dependent variant for time-independent networks. However, preprocessing times increase by a factor of 4 when using our time-dependent preprocessing instead of our static one. This is mainly due to two facts: On the one hand, we use more complex data structures for storing distance labels during arc-flags computation. On the other hand, we use a faster algorithm for setting arc-flags in a static scenario.

The query algorithm stays almost untouched. The only difference between a static and time-dependent SHARC query is the same as for plain Dijkstra: The key of a node depends on the departure time. Due to these very minor changes, the slow-down deriving from using a time-dependent query for a time-independent network is almost negligible.

6 Experiments

In this section, we present our experimental evaluation. To this end, we evaluate the performance of time-dependent SHARC for road and railway networks. Our implementation is written in C++. As priority queue we use a binary heap. For implementation details on graph data structures and multi-level partitions, see Appendix A. 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 3.

Methodology. In the following, we report preprocessing times and the overhead of the preprocessed data in terms of *additional* bytes per node. Moreover, we report the increase in number of edges and interpolation points of the output graph compared to the input. We report two types of queries: *time-queries*, i.e., queries for a specific departure time, and *profile-queries*, i.e., queries for computing $d_*(s,t)$. For each type we provide the average number of settled nodes, i.e., the number of nodes taken from the priority queue, the average number of relaxed edges, and the average query time. For s-t profile-queries, the nodes s and t are picked uniformly at random. Time-queries additionally need a departure time τ, which we pick uniformly at random as well. All figures in this paper are based on 100 000 random s-t queries and refer to the scenario that only the lengths of the shortest paths have to be determined, without outputting a complete description of the paths. However, our shortcut expansion routine needs less than 1 ms to output the whole path; the additional space overhead is ≈ 4 bytes per node.

Since picking s and t uniformly at random is not a very realistic assumption in continental-sized networks, we also evaluate *local queries* [34]. This setup allows insights into the performance of SHARC depending on the length of a query. This is achieved by choosing 1 000 (s,t) pairs for each Dijkstra rank: Starting a query from s (with random departure time), the rank of t is denoted by the number of settled nodes before t is settled. It is given for $2^0, 2^1, \ldots, 2^{\log|V|}$. The results are then presented in the form of a box-and-whisker plot [33].

Default Setting. Unless otherwise stated, we use $C = 2.5$ as expansion threshold during contraction for the all levels. The hop-bound of our contraction is set to 10, the interpolation-bound to 300. Note that the hop counter of all edges is set to 1 before each contraction step. Since we apply node-reduction during each iteration step, shortcuts may eventually represent more than 10 edges of the input graph. If we use landmarks, we select 8 nodes with *avoid*.

In the following, we evaluate four variants of time-dependent SHARC. The only difference between them is the way we compute arc-flags during preprocessing. Refinement of arc-flags is the same for all variants. Our *economical* variant sets arc-flags via DIJKSTRA-based approximation of labels, the *generous* version uses approximation with a fixed number (24, one for each hour of the day) of interpolation points, while the *aggressive* variant uses exact label-correcting algorithms on the topmost level. Finally, our *heuristic* variant sets heuristic (and potentially false negative) flags on all levels. For the latter, we construct 14 shortest path trees per node, i.e., we set K to 12. We hereby use arrival times for every two hours of the day. To keep preprocessing times limited, we compute arc-flags only for the topmost three levels and do not refine arc-flags for the lowest two levels. For static SHARC on road networks, this reduces preprocessing times by a factor of 3, but query performance decreases only by $\approx 30\%$.

6.1 Time-Dependent Road Networks

Inputs. Unfortunately, we only have access to a real-world time-dependent road network of Germany. Hence, we use two types of inputs. Besides the German road network, we also use the available real-world *time-independent* network of Western Europe and generate *synthetic* rush hours. All data has been provided by PTV AG for scientific use. The German road network has approximately 4.7 million nodes and 10.8 million edges. The corresponding figures for Europe are 18 million and 42.6 million, respectively.

6.1.1 Germany

We have access to five different traffic scenarios, collected from historical data: Monday, midweek (Tuesday till Thursday), Friday, Saturday, and Sunday. As expected, congestion of the roads is higher during the week than on the weekend: $\approx 8\%$ of the edges are time-dependent for Monday, midweek, and Friday. The corresponding figures for Saturday and Sunday are $\approx 5\%$ and $\approx 3\%$, respectively. We define the delay of a time-query by $1 - d(s,t,\tau)/d(s,t)$ with $d(s,t)$ depicting the length of a shortest s–t path in $\underline{G}$. For our inputs, the average delay over 100 000 random queries is 2.4% for Monday, 2.7% for midweek, 2.6% for Friday, 0.7% for Saturday, and 0.4% for Sunday. This confirms our assumption that traffic is higher during the week than on the weekend. For all German inputs, we apply a 5-level partition, with 112 cells on level 4 and 4 cells per supercell on levels 0 to 3. For this setup, we analyze the impact of the degree of perturbation and the quality of different arc-flag computations. Note that in the following, we use the term of perturbation depicting the degree of time-dependency, i.e., the percentage of time-dependent edges and the average delay of an edge during rush hours.

Traffic Days. Table 1 reports the performance of time-dependent SHARC with and without landmarks for all profiles we have access to. We use our economical and generous variant for all traffic days and our aggressive version for Saturday and Sunday. Unfortunately, preprocessing of our aggressive variant is too long for the remaining traffic days. For comparison, we also report the performance of static SHARC in a "no traffic" scenario. We also report the speed-up over

Table 1: Performance of SHARC on the German road network instance. Column *scenario* depicts the traffic day. Preprocessing times are given in hours and minutes, the overhead in *additional* bytes per node, increase in number of edges and increase in number of interpolation points. For queries, we report the average number of nodes taken from the priority queue, average number of relaxed edges, and average execution times of a query. We also report the speed-up over DIJKSTRA's algorithm.

		PREPROCESSING				TIME-QUERIES					
		time	space	edge	points	#settled	speed	#relaxed	speed	time	speed
scenario	algorithm	[h:m]	[B/n]	inc.	inc.	nodes	up	edges	up	[ms]	up
Monday	eco SHARC	1:16	156.6	25.4%	366.8%	19 136	124	101 176	54	24.55	63
	eco L-SHARC	1:18	220.6	25.4%	366.8%	2 681	887	18 071	303	6.10	255
	gen SHARC	20:47	155.9	25.2%	362.1%	16 472	144	87 092	63	21.13	74
	gen L-SHARC	20:49	219.9	25.2%	362.1%	2 308	1 030	15 555	352	5.25	296
midweek	eco SHARC	1:16	154.9	25.4%	363.8%	19 425	119	104 947	51	25.06	60
	eco L-SHARC	1:18	218.9	25.4%	363.8%	2 776	831	19 005	279	6.31	238
	gen SHARC	20:45	154.2	25.2%	359.2%	16 954	136	91 596	58	21.87	69
	gen L-SHARC	20:47	218.2	25.2%	359.2%	2 423	952	16 587	320	5.51	273
Friday	eco SHARC	1:10	142.0	25.4%	358.0%	17 412	134	92 473	58	22.07	69
	eco L-SHARC	1:12	206.0	25.4%	358.0%	2 500	936	16 895	319	5.59	271
	gen SHARC	19:31	141.7	25.2%	356.1%	15 308	153	81 298	66	19.40	78
	gen L-SHARC	19:33	205.7	25.2%	356.1%	2 198	1 065	14 853	363	4.92	309
Saturday	eco SHARC	0:42	90.3	25.0%	283.6%	5 284	441	19 991	269	5.34	276
	eco L-SHARC	0:44	154.3	25.0%	283.6%	940	2 478	4 867	1 103	1.50	978
	gen SHARC	6:54	88.9	24.9%	278.1%	4 842	481	18 319	293	4.89	301
	gen L-SHARC	6:56	152.9	24.9%	278.1%	861	2 705	4 460	1 204	1.38	1 067
	agg SHARC	48:57	84.3	24.5%	264.4%	721	3 229	1 603	3 349	0.58	2 554
	agg L-SHARC	48:59	148.3	24.5%	264.4%	295	7 905	1 036	5 182	0.32	4 589
Sunday	eco SHARC	0:30	64.6	24.6%	215.8%	2 142	1 097	6 549	826	1.86	787
	eco L-SHARC	0:32	128.6	24.6%	215.8%	576	4 076	2 460	2 200	0.73	2 011
	gen SHARC	5:27	62.9	24.5%	211.2%	1 737	1 352	5 311	1 019	1.51	970
	gen L-SHARC	5:29	126.9	24.5%	211.2%	467	5 026	1 995	2 712	0.59	2 480
	agg SHARC	27:20	60.7	24.1%	202.6%	670	3 504	1 439	3 759	0.50	2 904
	agg L-SHARC	27:22	124.7	24.1%	202.6%	283	8 300	978	5 535	0.29	5 045
no traffic	static SHARC	0:06	13.5	23.9%	23.9%	591	3 790	1 837	2 810	0.30	4 075

DIJKSTRA's algorithm, which settles ≈ 2.3 million nodes in 1.5 seconds on average. These values are independent of the applied traffic scenario.

We observe that the degree of perturbation has a high influence on both preprocessing and query performance of economical SHARC. Preprocessing times increase if perturbation is higher. This is mainly due to our refinement phase that uses partial label-correcting algorithms in order to improve the quality of arc-flags. The increase in overhead derives from the fact that the number of additional interpolation points for shortcuts increases. Analyzing query performance of SHARC, we observe that in a Sunday scenario, SHARC provides speed-ups of up to 787 over DIJKSTRA. However, this values drops to 60 if a high traffic scenario is applied. The reason for this loss in query performance is the bad quality of our DIJKSTRA-based approximation. If perturbation is higher, upper and lower bounds are less tight than in a scenario with only few time-dependent edges. Interestingly, adding landmarks yields an additional speed-up of up to 4. This is especially useful in high traffic scenarios as query performance is now down to 6.31 ms, which seems to be sufficient for most applications.

Switching to arc-flags approximation via functions during preprocessing (generous SHARC) hardly pays off. Preprocessing times increase by a factor between 10 (Sunday) and 20 (midweek) but this tremendous increase only yields an increase in query performance by $\approx 20\%$. We conclude that it is sufficient to settle for arc-flags approximation via bounds.

However, investing more preprocessing time pays off: Query performance of aggressive SHARC is almost independent of the traffic day: For both Saturday and Sunday, we observe query times of 0.5 ms, a speed-up of about 3 000 over DIJKSTRA's algorithm. By adding landmarks, we get down to 0.3 ms and the speed-up is now $\approx 5\,000$. Compared to static SHARC, we observe that aggressive time-dependent SHARC yields almost the same speed-up in terms of settled nodes. However, the number of relaxed edges is lower in static scenarios and, thus, query performance is slightly better. Still, we pay a high price in terms of preprocessing times for switching to aggressive SHARC. It seems as the best trade-off between preprocessing effort and query performance is an economical variant combined with landmarks. Here, speed-ups over DIJKSTRA's algorithm vary between 238 and 2 011, depending on the traffic day.

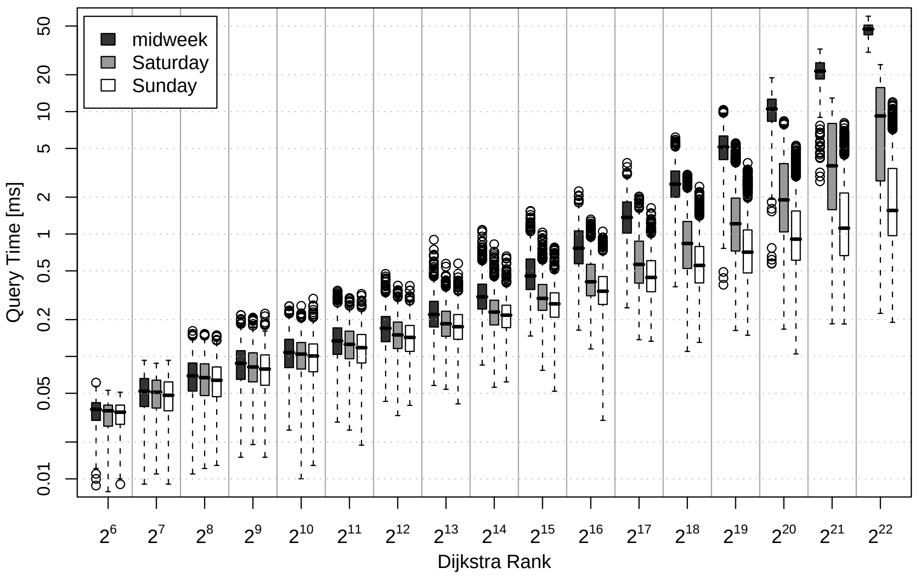


Figure 13: Comparison of time-dependent economical SHARC applying a Wednesday, Saturday, and Sunday traffic scenario using the Dijkstra rank methodology [34]. The results are represented as a box-and-whisker plot [33]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

Local Queries. In order to gain further insights into the impact of traffic days on query performance of economical SHARC, Figure 13 reports the query times of SHARC with respect to the Dijkstra rank. As inputs we use our German road network applying traffic data for midweek, Saturday, and Sunday. Note that we use a logarithmic scale due to outliers. We observe that up to a rank of 2^{12}, query performance is almost independent of the traffic day. However, beyond this rank, high traffic queries (midweek) get slower. The same holds for medium traffic queries (Saturday) beyond ranks of 2^{15}. The reason for this is that for long-range queries the quality of DIJKSTRA-based arc-flags fades since upper- and lower bounds get worse with increasing distance. Another interesting observation is that queries for a given rank vary by up to 2 orders of magnitude. Still, all queries are executed in less than 55 *ms*.

Heuristic SHARC. Table 2 reports query performance of SHARC if suboptimal paths are allowed. For approximate queries, the quality of the computed path is very important. Let μ be the length of the shortest path and σ be the length of the path found by heuristic SHARC. We report three types of errors: The error-rate (which depicts which fraction of the queries are non-optimal), the maximal relative error ($\sigma/\mu - 1$), and maximal absolute error ($\sigma - \mu$). Obviously, the path found by heuristic SHARC can only be longer than the shortest. Note that preprocessing times of heuristic SHARC are higher than for our (economical) exact variant since we need to grow 14 instead of 2 shortest path trees per boundary node. We observe excellent query performance *and*

Table 2: Performance of heuristic SHARC on the German road network instance. Since heuristic SHARC may yield suboptimal paths, we report the error-rate, the maximal relative, and maximal absolute error.

		PREPRO		ERROR			TIME-QUERIES					
		time	space	error	max	max	#settled	speed	#rel.	speed	time	speed
scenario	algorithm	[h:m]	[B/n]	-rate	rel.	ab.[s]	nodes	up	edges	up	[ms]	up
Monday	heu SHARC	3:30	138.2	0.46%	0.54%	39.3	810	2 935	1 593	3 439	0.69	2 253
	heu L-SHARC	3:32	202.2	0.46%	0.54%	39.3	330	7 213	1 076	5 090	0.38	4 104
midweek	heu SHARC	3:26	137.2	0.82%	0.61%	48.3	818	2 820	1 611	3 297	0.69	2 164
	heu L-SHARC	3:28	201.2	0.82%	0.61%	48.3	334	6 900	1 092	4 866	0.38	3 915
Friday	heu SHARC	3:14	125.2	0.50%	0.50%	50.3	769	3 044	1 522	3 543	0.64	2 358
	heu L-SHARC	3:16	189.2	0.50%	0.50%	50.3	322	7 266	1 054	5 118	0.36	4 168
Saturday	heu SHARC	2:13	80.4	0.18%	0.23%	16.9	666	3 499	1 336	4 018	0.51	2 887
	heu L-SHARC	2:15	144.4	0.18%	0.23%	16.9	278	8 369	927	5 788	0.29	5 097
Sunday	heu SHARC	1:48	58.8	0.09%	0.36%	14.9	635	3 699	1 271	4 255	0.46	3 163
	heu L-SHARC	1:50	122.8	0.09%	0.36%	14.9	272	8 639	908	5 960	0.27	5 420
no traffic	static SHARC	0:06	13.5	0.00%	0.00%	0.0	591	3 790	1 837	2 810	0.30	4 075

preprocessing effort of heuristic SHARC. Without landmarks, queries are up to 3 163 times faster than DIJKSTRA. If landmarks are added, this value increases to above 5 400. These values are achieved by a preprocessing effort of not more than 3.5 hours. More importantly, the impact of perturbation fades. For high traffic scenarios, queries are below 0.38 ms, for low traffic scenarios below 0.27 ms. This very good performance comes with paths of very good quality. Less than 0.9% of the queries are suboptimal with a maximal relative error of 0.61% and maximal absolute error of 50.3 seconds (the average path length is $\approx$ 12 hours). Since time-dependent road networks are based on historical data anyway, such low errors seem reasonable for real-world applications. One could even think of the following approach. We compute economical and heuristic SHARC, which only differ in arc-flags, not in shortcuts: As long as the server load is low, we use economical SHARC and switch to heuristic SHARC only during peak hours. Comparing heuristic and aggressive SHARC (cf. Table 1), we observe that query performance is almost the same for both variants. However, the former yields much lower preprocessing times, while the latter guarantees correctness.

Table 3: Performance of SHARC profile queries on Germany. Column *#nodes reinserted* depicts how many nodes have been reinserted in the queue after removal. Column *profile/time* shows the quotient of the corresponding figure for profile and time queries. Hence, it shows the slow-down when switching from time to profile queries.

		TIME-QUERIES			PROFILE-QUERIES					
		#settled	#relaxed	time	#settled	prof.	#nodes	#relaxed	time	prof.
traffic day	variant	nodes	edges	[ms]	nodes	/time	reins.	edges	[ms]	/time
Monday	eco	19 136	101 176	24.55	19 768	1.03	402	208 942	51 122	2 083
	heu	810	1 593	0.69	1 071	1.32	24	3 597	1 008	1 461
midweek	eco	19 425	104 947	25.06	20 538	1.06	432	222 066	60 147	2 400
	heu	818	1 611	0.69	1 100	1.35	27	3 731	1 075	1 548
Friday	eco	17 412	92 473	22.07	19 530	1.12	346	204 545	52 780	2 392
	heu	769	1 522	0.64	1 049	1.36	21	3 551	832	1 293
Saturday	eco	5 284	19 991	5.34	5 495	1.04	44	41 956	3 330	624
	agg	721	1 603	0.58	865	1.20	9	3 269	134	233
	heu	666	1 336	0.51	798	1.20	8	2 665	98	192
Sunday	eco	2 142	6 549	1.86	2 294	1.07	12	13 563	536	288
	agg	670	1 439	0.50	781	1.17	5	2 824	57	114
	heu	635	1 271	0.46	738	1.16	5	2 449	45	98

Profile Queries. Up to this point we only reported performance of time-queries, Table 3 now reports profile-query performance of SHARC. Note that profile figures are based on 1 000 random queries and that we also report time-query performance for comparison.

We observe that the perturbation has an even higher impact on profile-queries. While profiles can be computed by economical SHARC within 1 second on the weekend, profile queries take up to 1 minute during high traffic days. Our heuristic version, however, yields acceptable query times. For all traffic scenarios, a complete profile can be computed in ≈ 1 second. Comparing time- and profile-queries, we observe that the search-space only increases at most by 35% when running profile- instead of the time-queries. However, due to the high number of interpolation points of the labels propagated through the network, profile-queries are up to 2 400 times slower than time-queries. The slow-down is much less for a low traffic scenario. This is due to the fact that less edges are time-dependent and thus, labels get less complex in low traffic scenarios than in high traffic situations. Summarizing, switching from time to profile queries is expensive in terms of query times but at least for heuristic SHARC, computing a complete profile is practical.

6.1.2 Europe

Each edge of our European road network belongs to one of five main categories representing motorways, national roads, local streets, urban streets, and rural roads. In order to generate synthetic time-dependent edge costs, we use the generator introduced in [28]. The methods developed there are based on statistics gathered using real-world data on a limited-size road network. The period is set to 24 hours. Moreover, two traffic jams are assigned to each edge, one in the morning, one in the evening. For details, see [28]. We additionally adjust the degree of perturbation by assigning time-dependent edge-costs only to specific categories of edges. In our *no traffic* scenario, all edges are time-independent, i.e., the graph is static. In a *low traffic* scenario, all motorways are time-dependent, other roads are time-independent. The *medium traffic* scenario additionally includes congested national roads, and for the *high traffic* scenario, we perturb all edges except local and rural roads.

Table 4: Performance of SHARC on our time-dependent European road network instance. *Scenario* depicts the degree of perturbation, as described above.

		PREPRO		ERROR			TIME-QUERIES					
		time	space	error	max	max	#sett.	speed	#rel.	speed	time	speed
scen.	algorithm	[h:m]	[B/n]	-rate	rel.	[s]	nodes	up	edges	up	[ms]	up
	eco SHARC	1:45	21.9	0.00%	0.00%	0	36 063	247	238 467	88	31.55	177
low	eco L-SHARC	1:50	85.9	0.00%	0.00%	0	9 506	939	74 890	281	11.45	487
traffic	heu SHARC	6:31	21.1	35.25%	0.68%	285	2 827	3 156	4 333	4 849	1.26	4 410
	heu L-SHARC	6:36	85.1	35.25%	0.68%	285	1 550	5 758	4 081	5 149	0.91	6 097
	eco SHARC	4:37	42.6	0.00%	0.00%	0	42 776	210	296 845	75	42.75	132
med	eco L-SHARC	4:42	106.6	0.00%	0.00%	0	11 977	749	98 049	226	18.72	301
traffic	heu SHARC	10:55	39.1	36.11%	1.28%	431	3 920	2 289	6 238	3 550	1.78	3 154
	heu L-SHARC	11:00	103.1	36.11%	1.28%	431	2 308	3 888	6 016	3 681	1.33	4 240
	eco SHARC	6:44	133.8	0.00%	0.00%	0	66 908	133	480 768	44	82.12	70
high	eco L-SHARC	6:49	197.8	0.00%	0.00%	0	18 289	485	165 382	127	38.29	150
traffic	heu SHARC	22:12	127.2	39.56%	1.60%	541	5 031	1 764	8 411	2 498	2.94	1 958
	heu L-SHARC	22:17	191.2	39.56%	1.60%	541	3 873	2 292	8 103	2 592	2.13	2 703
no	static SHARC	0:35	13.7	0.00%	0.00%	0	779	11 301	3 335	6 299	0.35	15 831

Table 4 reports the results of economical and heuristic SHARC for the European inputs. Unfortunately, it turned out that this input is too big to apply generous or aggressive SHARC. Note that we again report the performance of static SHARC for comparison. Like for Germany, we observe that the degree of perturbation has a high influence on both preprocessing and query performance of SHARC. Again, preprocessing times increase if more edges are time-dependent. Query performance of economical SHARC on Europe is similar—with respect to speed-up over DIJKSTRA's algorithm—to Germany. Combined with landmarks, queries times are below 40 ms for all scenarios. These query times can be achieved by investing up to 7 hours of preprocessing, which still seems reasonable for most applications.

Comparing heuristic SHARC on Germany (cf. Table 2) and Europe, we observe that speed-ups over DIJKSTRA's algorithm are almost identical in both cases. However, the quality of paths is worse for Europe than for Germany: Up to 40% of the queries are incorrect and the maximal error increases to 1.6%. A reason for this is that for Europe, shortcuts get more complex than for Germany. Hence, the shortest path may change more often during the day than for Germany. Still, with respect to travel times within Europe, these errors still seem reasonable.

6.1.3 Comparison

Next, we compare time-dependent SHARC to other recent time-dependent speed-up techniques, including those published after the preliminary version of this work. We hereby split our comparison in two parts: exact queries and approximation. Table 5 reports query performance of time-dependent DIJKSTRA, uni-directional ALT [11], bidirectional ALT [28], Core-ALT [8], and Contraction Hierarchies [1] compared to SHARC in an exact setup, while Table 6 depicts performance if suboptimal paths are allowed. As input we use our time-dependent road networks of Europe (high traffic) and Germany (midweek and Sunday). Note that no approximate variant of Contraction Hierarchies exists yet and that no results for Europe (high traffic) have been published. The reason for the latter is the high memory consumption making Contraction Hierarchies impractical for this input. Also note that the time-dependent variants of Contraction Hierarchies and Core-ALT have been published after [7].

Table 5: Performance of DIJKSTRA, uni- and bidirectional ALT, Core-ALT, SHARC, and Contraction Hierarchies (CH) in an exact setup. Note that no figures for the number of relaxed edges of CH is given in [1].

		PREPRO		QUERIES					
		time	space	#settled	speed	#relaxed	speed	time	speed
input	algorithm	[h:m]	[B/n]	nodes	up	edges	up	[ms]	up
Ger midweek	Dijkstra	0:00	0	2 305 440	1	5 311 600	1	1 502.88	1
	uni-ALT	0:23	128	200 236	12	239 112	22	148.36	10
	ALT	0:23	128	110 134	21	131 090	41	94.26	16
	Core-ALT	0:09	50	3 190	723	12 255	433	5.36	280
	eco SHARC	1:16	155	19 425	119	104 947	51	25.06	60
	eco L-SHARC	1:18	219	2 776	831	19 005	279	6.31	238
	CH	0:25	1 019	528	4 366	–	–	1.22	1231
Ger Sunday	Dijkstra	0:00	0	2 348 470	1	5 410 600	1	1 464.41	1
	uni-ALT	0:23	128	142 631	16	170 670	32	92.79	16
	ALT	0:23	128	58 956	40	70 333	77	42.96	34
	Core-ALT	0:05	19	1 773	1 325	6 712	806	2.13	688
	eco SHARC	0:30	65	2 142	1 097	6 549	826	1.86	787
	eco L-SHARC	0:32	129	576	4 076	2 460	2 200	0.73	2 011
	agg SHARC	27:20	61	670	3 504	1 439	3 759	0.50	2 904
	agg L-SHARC	27:22	125	283	8 300	978	5 535	0.29	5 045
	CH	0:11	248	407	5 770	–	–	0.71	2 061
Europe high traffic	Dijkstra	0:00	0	8 877 158	1	21 006 800	1	5 757.45	1
	uni-ALT	1:15	128	2 143 160	4	2 613 994	8	1 520.83	4
	ALT	1:15	128	3 009 320	3	3 799 112	6	1 379.21	4
	Core-ALT	1:00	61	60 961	146	356 527	59	121.47	47
	eco SHARC	6:44	134	66 908	133	480 768	44	82.12	70
	eco L-SHARC	6:49	198	18 289	485	165 382	127	38.29	150

Table 6: Performance of DIJKSTRA, uni- and bidirectional ALT, Core-ALT, and SHARC in an approximation setup.

		PREPRO		ERROR			TIME-QUERIES					
		time	space	error	max	max	#sett.	spd	#rel.	speed	time	spd
input	algorithm	[h:m]	[B/n]	-rate	rel.	abs[s]	nodes	up	edges	up	[ms]	up
Ger mid	ALT	0:23	128	12.4%	14.32%	1 892	50 764	45	60 398	88	36.92	41
	Core-ALT	0:09	50	8.2%	13.84%	2 408	1 593	1 447	5 339	995	1.87	804
	heu SHARC	3:26	137	0.8%	0.61%	48	818	2 820	1 611	3 297	0.69	2 164
	heu L-SHARC	3:28	201	0.8%	0.61%	48	334	6 900	1 092	4 866	0.38	3 915
Ger Sun	ALT	0:23	128	10.4%	14.28%	1 753	50 349	47	59 994	90	36.04	41
	Core-ALT	0:05	19	4.0%	12.72%	1 400	1 551	1 514	5 541	976	1.71	856
	heu SHARC	1:48	59	0.1%	0.36%	15	635	3 699	1 271	4 255	0.46	3 163
	heu L-SHARC	1:50	123	0.1%	0.36%	15	272	8 639	908	5 960	0.27	5 420
Eur high	ALT	1:15	128	35.4%	10.57%	5 789	311 209	29	382 061	55	214.24	27
	Core-ALT	1:00	61	33.0%	8.69%	6 643	6 365	1 395	32 719	642	9.22	624
	heu SHARC	22:12	127	39.6%	1.60%	541	5 031	1 764	8 411	2 498	2.94	1 958
	heu L-SHARC	22:17	191	39.6%	1.60%	541	3 873	2 292	8 103	2 592	2.13	2 703

Exact Setup. Depending on the scenario, different algorithms perform best. While Core-ALT is the technique with lowest preprocessing effort (both time and overhead), Contraction Hierarchies (CH) or SHARC win with respect to query performance. While CH tend to have fast query times, the space consumption is up to 1 000 bytes per node. For this reason, CH cannot be used for Europe (high traffic). Aggressive SHARC however, has the lowest query times but for the price of high preprocessing times. In fact, preprocessing times for aggressive SHARC are only practical for Germany Sunday. As soon as the graph gets bigger or more edges are time-dependent, preprocessing takes more than 2 days. So, it seems as if economical L-SHARC is the favorable technique since it is the one most robust to the input. Moreover, query performance is always within reasonable bounds.

Approximation. Things are even clearer if we allow suboptimal paths. Performance of SHARC is boosted by more than an order of magnitude if we drop correctness combined with a reasonable preprocessing effort. Although ALT and Core-ALT also gain from allowing suboptimal paths, both query performance and the quality of the computed paths is (much) worse than for heuristic SHARC. We conclude that SHARC is (clearly) superior if we allow slightly suboptimal paths.

6.2 Time-Dependent Timetable Information

Our timetable data—provided by HaCon for scientific use—of Europe consists of 30 516 train stations and 1 775 482 elementary connections. The period is 24 hours. The resulting realistic, i.e., including transfer times, time-dependent network has about 0.5 million nodes and 1.4 million edges, and fulfills the FIFO property (cf. Section 3). Table 7 reports the performance of time-dependent SHARC using this input. We report the performance of economical and aggressive SHARC. Recall that the economical version computes approximate arc-flags on all levels, while our aggressive variant computes exact flags during preprocessing. For comparison, we also report the results for plain Dijkstra and unidirectional ALT.

We observe a good performance of SHARC in general. Queries for specific departure times are up to 29.7 times faster than plain Dijkstra in terms of search space. This lower search space yields a speed-up of a factor of 26.6. This gap originates from the fact that SHARC operates on a graph enriched by shortcuts. As shortcuts tend to have many interpolation points, evaluating them is more expensive than original edges. As expected, our economical variant is slower than the aggressive version but preprocessing is about 8 times faster. Recall that the only difference between both versions is the way arc-flags are computed during the last iteration step. However, since composing and merging functions is more expensive than adding and comparing integers, preprocessing times increase significantly.

Table 7: Performance of time-dependent Dijkstra, uni-directional ALT and SHARC using our timetable data as input. Preprocessing times are given in hours and minutes, the overhead in bytes per node. Moreover, we report the increase in edge count over the input. Column *#settled nodes* denotes the number of nodes removed from the priority queue, query *times* are given in milliseconds. Column *speed-up* reports the speed-up over the corresponding value for plain Dijkstra.

	Prepro			Time-Queries				Profile-Queries			
	time	space	edge	#settled	speed	time	speed	#settled	speed	time	speed
technique	[h:m]	[B/n]	inc.	nodes	up	[ms]	up	nodes	up	[ms]	up
Dijkstra	0:00	0	0%	260 095	1.0	125.2	1.0	1 919 662	1.0	5 327	1.0
uni-ALT	0:02	128	0%	127 103	2.0	75.3	1.7	1 434 112	1.3	4 384	1.2
eco SHARC	1:30	113	74%	32 575	8.0	17.5	7.2	181 782	10.6	988	5.4
agg SHARC	12:15	120	74%	8 771	29.7	4.7	26.6	55 306	34.7	273	19.5

Comparing time- and profile-queries, we observe that the slow-down in terms of the number of heap operations is only of a factor of 4 to 7. Again, as composing and merging functions is more expensive than adding and comparing integers, the loss in terms of running times is much higher. Still, both SHARC variants are capable of computing a travel time function between two random train stations in less than 1 second.

Comparison to Road Networks. Comparing the figures from Tables 3 and 7, we observe that speed-ups for time-queries in road networks are higher than in railway networks. To some extend, this stems from the fact that the network is smaller. On a road network with 0.5 million nodes, aggressive SHARC yields speed-ups of about 800. The remaining gap mainly stems from the fact that road networks include a stronger hierarchy than railway networks. However, switching from time to profile queries is cheaper for timetable information. The reason for this is that it is cheaper to compose and merge functions needed for timetables than those needed for road networks (cf. Section 3).

7 Conclusion

In this work, we presented the first efficient speed-up technique for exact routing in large time-dependent transportation networks. We generalized the recently introduced SHARC algorithm by augmenting several static routines of the preprocessing to time-dependent variants. In addition, we introduced routines to handle the problem of expensive shortcuts. As a result, we are able to run fast queries on continental-sized transportation networks of both roads and railways. Moreover, we are able to compute the distances between two nodes for all possible departure times. By dropping optimality, we compute time-dependent paths with minor errors in road networks up to 5 000 times faster than plain Dijkstra.

Regarding future work, one could think of faster ways of composing, merging, and approximating piecewise linear functions as this would directly accelerate preprocessing and, more importantly, profile-queries significantly. Preprocessing of aggressive SHARC is based on constructing multiple profile graphs, which are independent of each other. Hence, it seems as if massive parallelization might help to preprocess aggressive SHARC within reasonable time for all inputs. Another interesting question is whether we can somehow reduce the space consumption of SHARC. Here, one should focus on shortcuts since they are the main reason for the increase in space consumption when switching from static to time-dependent route planning.

Acknowledgments. The author would like to thank Veit Batz, Reinhard Bauer, Andrew Goldberg, Robert Görke, Riko Jacob, Bastian Katz, Ignaz Rutter, Peter Sanders, Heiko Schilling, Dominik Schultes, Dorothea Wagner, and Renato Werneck for interesting discussions on time-dependent route planning. Moreover, I thank the anonymous referees for their very detailed reviews and many valuable suggestions for improvement. I also thank Giacomo Nannicini for providing me with his time-dependent edge-cost generator, and Thomas Pajor for his work on parsers for the HAFAS timetable data. Finally, I thank PTV and HaCon for providing me with real-world data for scientific use.

References

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

[2] 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, April 2008.

[3] R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14:2.4–2.29, May 2009. Special Section on Selected Papers from ALENEX 2008.

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

[5] K. Cooke and E. Halsey. The Shortest Route Through a Network with Time-Dependent Intermodal Transit Times. *Journal of Mathematical Analysis and Applications*, (14):493–498, 1966.

[6] B. C. Dean. Continuous-Time Dynamic Shortest Path Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.

[7] D. Delling. Time-Dependent SHARC-Routing. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 332–343. Springer, September 2008. Best Student Paper Award - ESA Track B.

[8] D. Delling and G. Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of *Lecture Notes in Computer Science*, pages 813–824. Springer, December 2008.

[9] 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*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[10] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway Hierarchies Star. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *Shortest Path Computations: Ninth DIMACS Challenge*, volume 24 of *DIMACS Book*. American Mathematical Society, 2009. Accepted for publication, to appear.

[11] D. Delling and D. Wagner. Landmark-Based Routing in Dynamic Graphs. In C. Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer, June 2007.

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

[13] Y. Disser, M. Müller–Hannemann, and M. Schnee. Multi-Criteria Shortest Paths in Time-Dependent Train 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 347–361. Springer, June 2008.

[14] I. C. Flinsenberg. *Route Planning Algorithms for Car Navigation.* PhD thesis, Technische Universiteit Eindhoven, 2004.

[15] 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.

[16] 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.

[17] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*, pages 129–143. SIAM, 2006.

[18] 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.

[19] 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.

[20] 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.

[21] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *Shortest Path Computations: Ninth DIMACS Challenge*, volume 24 of *DIMACS Book*. American Mathematical Society, 2009. To appear.

[22] D. E. Kaufman and R. L. Smith. Fastest Paths in Time-Dependent Networks for Intelligent Vehicle-Highway Systems Application. *Journal of Intelligent Transportation Systems*, 1(1):1–11, 1993.

[23] 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.

[24] U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.

[25] U. Lauther. An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Road-networks with Precalculated Edge-Flags. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *Shortest Path Computations: Ninth DIMACS Challenge*, volume 24 of *DIMACS Book*. American Mathematical Society, 2009. To appear.

[26] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning Graphs to Speed Up Dijkstra's Algorithm. In *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, Lecture Notes in Computer Science, pages 189–202. Springer, 2005.

[27] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11:2.8, 2006.

[28] G. Nannicini, D. Delling, L. Liberti, and D. Schultes. Bidirectional A* Search for Time-Dependent Fast Paths. In C. C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 334–346. Springer, June 2008.

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

[30] F. Pellegrini. SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package, 2007.

[31] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Towards Realistic Modeling of Time-Table Information through the Time-Dependent Approach. In *Proceedings of ATMOS Workshop 2003*, pages 85–103, 2004.

[32] 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.

[33] R Development Core Team. R: A Language and Environment for Statistical Computing, 2004.

[34] P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[35] 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.

[36] D. Schultes. *Route Planning in Road Networks*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, February 2008.

[37] D. Schultes and P. Sanders. Dynamic Highway-Node Routing. In C. Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 66–79. Springer, June 2007.

A Implementation Details

For an efficient implementation of time-dependent SHARC, we need a fast time-dependent graph datastructure, a tailored data structure for maintaining a multi-level partition, and an efficient priority queue. For the latter, Dominik Schultes provided us with his implementation of a binary heap [36]. In the following, we describe our graph and multi-level partition data structures in more detail.

Time-Dependent Graphs. SHARC is a unidirectional technique allowing a very simple data structure. We use two arrays of structs, one representing nodes, the other edges. Enumeration is started at zero. The edge entries are ordered by their source nodes; thus, all outgoing edges of a node are stored in succession. Each node stores the index to its first outgoing edge, providing an easy access to them. Each edge stores its head node. A dummy node is also saved at the end of the node-array to provide a pointer to the first invalid element of the edge-array. A number of interpolation points is stored to each edge depicting the travel time at different departure times. Since the number of such points is not the same for each edge we introduce a third layer storing all interpolation points of the graph. Each edge stores an additional pointer to the first interpolation point in the third layer. For each edge, the interpolation points are sorted by their time value. Note that we have to introduce a dummy edge for iteration over the points of the last edges. Figure 14 gives an example. For accessing the correct interpolation points for a specific departure time τ, we access the point $p = \lfloor \tau/\Pi \rfloor \cdot |P(e)| + \texttt{firstPoint}(e)$ where $|P(e)|$ denotes the number of points assigned to edge e. In most cases, this access point is close to the one we seek. By linear search we finally retrieve the correct entries.

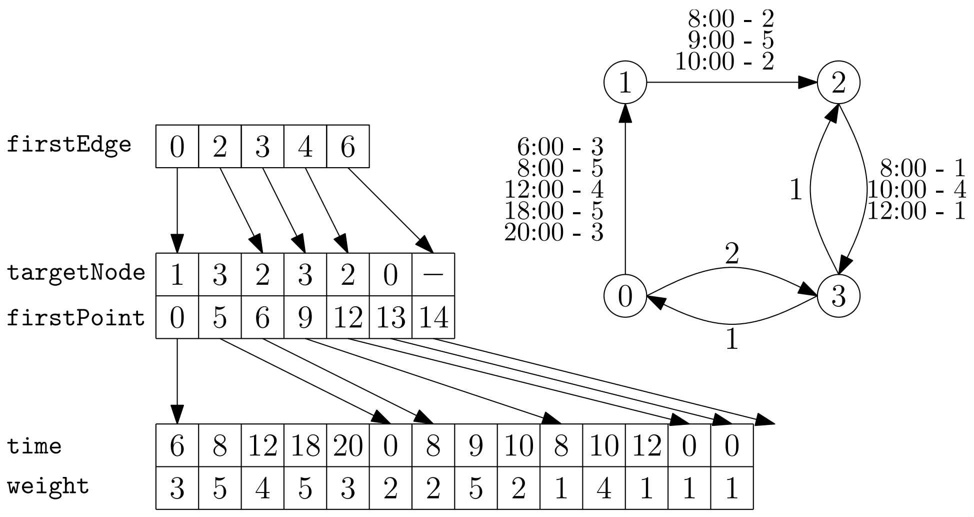


Figure 14: Adjacency representation of a time-dependent graph. The right hand figure shows the representation of the graph including three time-dependent edges. The resulting data structure is shown by the left hand figure.

Multi-Level Partition. During a SHARC query, we need to access the cell number of a node on a certain level very efficiently. In fact, this operation is executed whenever we settle a node. So, access should be as fast as possible. For each node, we store the cell numbers of all levels in one integer. The cell number on the lowest level is stored in the lowest bits, the number on the highest level in the highest bits. An example if given in Fig. 15. Then, the cell number can be accessed by one bitshift and an additional AND operation.

	1	1	0	1	0	0	1	1	1	0	1	0	0	1	0	0
level	4							3			2		1		0	
cell number	105							6			2		1		0	

Figure 15: Example for storing a multi-level partition with 4 cells on the lower 3 levels, 8 cells on the fourth level, and 108 cells on the topmost level.