Time-Dependent SHARC-Routing*

Daniel Delling

Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
delling@ira.uka.de

Abstract. During the last years, many speed-up techniques for DIJKSTRA's algorithm have been developed. As a result, computing a shortest path in a *static* road network is a matter of microseconds. However, only few of those techniques work in *time-dependent* networks. Unfortunately, such networks appear frequentely in reality: Roads are predictably congestured 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 *exact* time-dependent speed-up technique based on our recent SHARC-algorithm. As a result, we are able to efficiently compute shortest paths in time-dependent continental-sized transportation networks, both of roads and of railways.

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 [1] 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 yielding faster query times for typical instances, e.g., road or railway networks. See [2] for an overview. 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 [3,4] 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. [5] for details) as well. Unfortunately, none of the speed-up techniques yielding high speed-ups can be used in a time-dependent network in a straight-forward manner. Moreover, possible problem statements for shortest paths become even more complex in such networks. A user could ask at what time she should depart in order to spend as little time traveling as possible.

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

D. Halperin and K. Mehlhorn (Eds.): ESA 2008, LNCS 5193, pp. 332–343, 2008.

Related Work. As already mentioned, a lot of speed-up techniques for static scenarios have been developed during the last years. Due to space limitations, we direct the interested reader to [2], which gives a good overview over static routing techniques. Much less work has been done on time-dependent routing. In [6], DIJKSTRA's algorithm is extended to the time-dependent case based on the assumption that the network fulfills the FIFO property. The FIFO property is also called the *non-overtaking property*, because it basically states that if A leaves an arbitrary node s before B, B cannot arrive at any node t before A. Computation of shortest paths in FIFO networks is polynomially solvable [7], while it is NP-hard in non-FIFO networks [8].

Goal-directed search, also called A^* [9], has been adapted to the previously described scenario; an efficient version for the static case has been presented in [10]. In [3], 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 perturbation. Goal-directed search has also successfully been applied to time-dependent timetable networks [5,11]. Recently, it has been shown that time-dependent ALT can be used in a bidirectional manner [12]. However, in order to obtain faster queries than in the unidirectional case, the user has to accept approximative solutions. Moreover, our old implementation of static SHARC [13] already allowed fast *approximative* queries in a time-dependent scenario.

Our Contribution. In this work, we show how our recently developed SHARC-algorithm [13] 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 [14]. 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 our work on time-dependent routing in Section 2 by introducing basic definitions and a short review of SHARC in static scenarios. Basic work on modeling time-dependency in road and railway networks is located in Section 3. Furthermore, we introduce basic algorithms that our preprocessing routines rely on. The preprocessing routine itself and the query algorithms of time-dependent SHARC are located in Section 4. We hereby show how the two main ingredients of SHARC, i.e., graph contraction and arc-flags computation, have to be altered for time-dependent networks. It turns out that the adaption of contraction is straight-forward, 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. We also show how SHARC can be used to compute a shortest path between two points for all possible departure times.

In order to show that time-dependent SHARC performs well in real-world environments, we present an extensive experimental evaluation in Section 5. Section 6 concludes our work with a summary and possible future research.

2 Preliminaries

The major difference between static and time-dependent routing is the usage of 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 $x, y \in \Pi, x \leq y$. In the following, we call Π the *period* of the input. The composition of two functions $f, g \in \mathbb{F}$ is defined by $(f \oplus g)(x) := f(x) + g((f(x) + x) \, mod \, p)$. Moreover, we need to *merge* functions. The merged function h of two functions f, g is defined by $h(x) = \min\{f(x), g(x)\}$. Comparison of functions is defined as follows: $f < g$ means that $f(x) < g(x)$ holds for all $x \in \Pi$. 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. We also restrict ourselves to simple, directed graphs $G = (V, E)$ with time-dependent length functions $len : E \to \mathbb{F}$. Note that our networks fullfill the FIFO-property if we interpret the length of an edge as travel times due to our choice of $\mathbb{F}$. The reverse graph $\overline{G} = (V, \overline{E})$ 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. A *partition* of V is a family $\mathcal{C} = \{C_0, C_1, \ldots, C_k\}$ 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*. A *multilevel partition* of V is a family of partitions $\{\mathcal{C}^0, \mathcal{C}^1, \ldots, \mathcal{C}^l\}$ such that for each $i < l$ and each $C_n^i \in \mathcal{C}^i$ a cell $C_m^{i+1} \in \mathcal{C}^{i+1}$ exists with $C_n^i \subseteq C_m^{i+1}$. In that case the cell C_m^{i+1} is called the *supercell* of C_n^i. The supercell of a level-l cell is V. 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 $(v, u) \in E$ or $(u, v) \in E$.

By $d(s, t, \tau)$ we denote the distance between $s, t \in V$ if departing from s at time τ. The distance-label, i.e., the distance between s and t for all possible departure times $\in \Pi$, is given by $d_*(s, t)$. Note that the distance-label is a function $\in \mathbb{F}$. In the following, 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*.

Static SHARC-Routing. The classic arc-flag approach [14] 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* iff 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*. SHARC [13] extends and combines ideas of Arc-Flags and hierarchical approaches [18,17]. Preprocessing of static SHARC is divided into three sections. During the *initialization* phase, we extract the 2-core of the graph and perform a *multi-level* partition of G. Then, an *iterative* process starts. At each step i we first *contract* the graph by *bypassing* low-degree nodes and set the arc-flags *automatically* for each removed edge. On the contracted graph we compute the arc-flags of level i by growing a

partial centralized shortest-path tree from each cell C_j^i. At the end of each step we *prune* the input by detecting those edges that already have their final arc-flags assigned. In the *finalization* phase, we assemble the output-graph, refine arc-flags of edges removed during contraction and finally reattach the nodes removed at the beginning. The query of static SHARC is a multi-level Arc-Flags DIJKSTRA adapted from a two-level Arc-Flags setup [14].

3 Models and Basic Algorithms

In this section, we introduce our approach how to model time-dependency in road and railway networks efficiently. In particular, we present our label-correcting algorithm, which is a main ingredient of time-dependent SHARC-preprocessing.

Modeling Time-Dependency. We apply two types of edge-functions, one for road networks, the other one for timetable information.

In *road networks*, we use a piece-wise linear function for modeling time-dependency. 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. Let $P(f)$ be the number of interpolation points of f. Then the composed function $f \oplus g$, modeling the duration for traversing g after f, may have up to $P(f)+P(g)$ number of interpolation points in worst case. This is one of the main problems when routing in time-dependent graphs: 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. Even worse, merging travel-functions, modeling the merge of two parallel edges into one, increases $|P|$ as well.

In *timetable graphs*, time-dependent edges model several trains running on the same route from one station to another. For each such connection, we add an interpolation point to the corresponding edge. The timestamp σ of the interpolation point is the departure time, the weight w the travel time. When we want to evaluate a time-dependent edge at a specific time τ, we identify the interpolation point with minimum $\sigma - \tau \geq 0$. Then the resulting traveltime is $w + \sigma - \tau$, i.e., the waiting time for the next connection plus its travel duration.

Note that composing two timetable edge-functions f, g is less expensive than in road networks. More precisely, $P(f \oplus g) = \min\{P(f), P(g)\}$ holds as the number of relevant departure times is dominated by the edge with less connections. Merging functions, however, may increase the number of interpolation points.

Label-Correcting Algorithms. As already mentioned, computing $d(s, t, \tau)$ can be solved by a modified DIJKSTRA [6]. However, computing $d_*(s, t)$ is more expensive but can be computed by a label-correcting algorithm [15]. Such an algorithm can be implemented very similarly to DIJKSTRA. The source node s is initialized with a constant label $d_*(s, s) = 0$, any other node u with a constant label $d_*(s, u) = \infty$. 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 [15] is the fact that the runtime of label-correcting algorithms highly depends on the complexity of the edge-functions.

In the following, we construct *shortest-path DAGs*, 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 (v,u) a *DAG*-edge if $d_*(s,v) \oplus (v,u) > d_*(s,u)$ does *not* hold. In other words, (u,v) is a DAG-edge iff it is part of a shortest path from s to v for at least one departure time.

4 Exact Time-Dependent SHARC

In static scenarios, a true 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. In the following, we show how to adapt preprocessing of SHARC in order to reflect this intuition correctly. Moreover, we present the time-dependent query algorithm.

4.1 Time-Dependent Preprocessing

Initialization. In a first step, we extract the 2-core from the graph since we can directly assign correct arc-flags to all edges outside the 2-core: Edges targeting the 2-core get full flags assigned, others only get the own-cell flag set to true. Note that this procedure is independent from edge weights. Next, we perform a multi-level partitioning—using SCOTCH [16]—of the *unweighted* graph.

Iteration. Next, an iterative process starts. Each iteration step is divided into three parts, described in the following: contraction, edge reduction, arc-flag computation.

Contraction. Our time-dependent contraction routine works very similar to a static one [2,17]: We iteratively *bypass* nodes until no node is *bypassable* any more. To bypass a node n we first remove n, its incoming edges I and its outgoing edges O from the graph. Then, for each combination of $e_i \in I$ and $e_o \in O$, we introduce a new edge of the length $len(e_i) \oplus len(e_o)$. Note that we explicitely allow multi-edges. Hence, each shortcut represents exactly one path in the graph,

making shortcut unpacking easier. We call the number of edges of the path that a shortcut represents on the graph at the beginning of the current iteration step the *hop number* of the shortcut. Note that in road networks, contraction gets more expensive in terms of memory consumption because the number of interpolation points of an added shortcut is roughly the sum of the number of interpolation points of the arcs the shortcuts is assembled from. Moreover, merging a shortcut with an existing edge may increase the number of interpolation points even further. Hence, the choice of which node to bypass next is even more important for the time-dependent scenario than for the static one. We use a heap to determine the next bypassable node [17]. Let $\#shortcut$ of *new* edges that would be inserted into the graph if n is bypassed and let $\zeta(n) = \#\text{shortcut}/(|I| + |O|)$ be the *expansion* [17] of node n. Furthermore, let $h(n)$ be the hop number of the hop-maximal shortcut, and let $p(n)$ be the number of interpolation points of the shortcut with most interpolation points, that would be added if n was bypassed. Then we set the key of a node n within the heap to $h(n)+p(n)+10\cdot\zeta(n)$, smaller keys have 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.

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 200 interpolation points or a hop number greater than 10. We say that the nodes that have been bypassed belong to the *component*, while the remaining nodes are called *core-nodes*. Note that in order to guarantee correctness, we have to use *cell-aware* contraction, i.e., a node n is never marked bypassable if any of its neighboring nodes is *not* in the same cell as n.

Edge-Reduction [4]. Note that our contraction potentially adds shortcuts not needed for keeping the distances in the core correct. Hence, we perform an edge reduction directly after each contraction. We grow a shortest-path DAG from each node u of the core. We stop the growth as soon as all neighbors t of u have their final label assigned. Then we check all neighbors whether $d_*(u,t) < len(u,t)$ holds. If it holds, we can remove (u,t) from the graph because for all possible departure times, the path from u to t does not include (u,t). In order to limit the running time of this procedure, we restrict the number of priority-queue removals to 100. Hence, we may leave some unneeded edges in the graph.

Arc-Flags. We have to set arc-flags for 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 source node s of the deleted edge. If s 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 s belongs to the component, all arc-flags are set to *true* as a query has to leave the component in order to reach a node outside this cell.

Setting arc-flags of those edges not removed from the graph is more expensive. A straight-forward adaption of computing arc-flags in a time-dependent graph

is to grow a shortest-path DAG in $\overline{G}$ for all boundary nodes $b \in B_C$ of all cells C at level i. We stop the growth 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 $AF_C(u,v) = \mathit{true}$ if (u,v) is a DAG-edge for at least one DAG grown from all boundary nodes $b \in B_C$.

However, this approach would require to compute a full label-correcting algorithm on the backward graph from each boundary node yielding too long preprocessing times for large networks. Recall that the running time of growing DAGs is dominated by the complexity of the function. Hence, we may grow two DAGs for each boundary node, the first uses $\uparrow len$ as length functions, the latter $\downarrow len$. As we use approximations with a constant number of interpolation points (we use 48), growing two such DAGs is faster than growing one exact one. We end up in two distance labels per node u, one being an overapproximation, the other being an underapproximation of the correct label. Then, we set $AF_C(u,v) = \mathit{true}$ if $len(u,v) \oplus \uparrow d_*(v,b_C) >\downarrow d_*(u,b_C)$ does not hold. If networks get so big that even setting approximative labels is prohibited 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 $\underline{len}$. Note that by approximating arc-flags the quality of them may decrease but correctness is untouched. Thus, queries remain correct but may become slower.

Finalization. The last phase of our preprocessing-routine first 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 edge may have no arc-flag set to true, which we can safely remove from the output graph.

Arc-Flags Refinement. During the iteration-phase we set sub-optimal arc-flags to edges originating from component nodes. However, a query starting from a node u being part of the a component has to leave the component via core-nodes. We call those nodes the exit nodes of u. The idea of arc-flags refinement is to propagate the flags from the exit nodes to edges outgoing from u. For details, see [13]. This routine can directly be adapted to a time-dependent scenario by growing shortest-paths DAGs from each u. However, we limit the growth of those DAGs to 1000 priority-queue removals due to performance. 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.

Shortcut-Removal. As already mentioned, 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 shortest path it represents. If all nodes on this shortest path have a degree less than 3, we remove (u,v) from the graph and all edges being part of the shortest path additionally inherit the arc-flags from (u,v).

4.2 Query

Time-dependent SHARC allows time- and profile-queries. For computing $d_\tau(s,t)$, we use a modified DIJKSTRA that operates on the output graph. The modifications are as follows: When settling a node n, we compute the lowest level i on which n and the target node t are in the same supercell. In order to keep the effort for this operation as small as possible we use such a numbering of cells such that the common level can be computed by the most significant bit of current and target cell. Moreover, we consider only those edges outgoing from n 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. For computing $d_*(s,t)$, we use a modified variant of our label-correcting algorithm (see Section 3) 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 3.

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. As we allow multi-edges during contraction, each shortcut represents exactly one path in the network, and hence, we can directly apply the unpacking routine from Highway Hierarchies [18].

5 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++ using solely the STL. As priority queue we use a binary heap. Our tests were executed on one core of an AMD Opteron 2218 running SUSE Linux 10.3. The machine is clocked at 2.6 GHz, has 16 GB of RAM and 2 x 1 MB of L2 cache. The program was compiled with GCC 4.2, using optimization level 3.

Default Setting. Unless otherwise stated, we use a 7-level partition with 4 cells per supercell on levels 0 to 5 and 104 cells on level 6 for road networks. As railway networks are smaller, we use a 3-level setup with 8 cells per supercell on levels 0 to 1 and 112 cells on level 2 for such networks. We use $c = 2.5$ as contraction parameter for the all levels. The hop-bound of our contraction is set to 10, the interpolation-bound to 200.

In the following, we report preprocessing times and the overhead of the preprocessed data in terms of *additional* bytes per node. Moreover, 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, 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 as well, which we pick uniformly at random as well. All figures in this paper are based on 1 000 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.

5.1 Time-Dependent Timetable Information

Our timetable data—provided by Hacon for scientific use—of Europe consists of 30 516 stations and 1 775 482 elementary connections. The period is 24 hours. The resulting realistic, i.e., including transfer times, time-dependent network (cf. [5] for details on modeling issues) has about 0.5 million nodes and 1.4 million edges. Table 1 reports the performance of time-dependent SHARC using this input. We report the performance of two variants of SHARC: the economical version computes DIJKSTRA-based arc-flags on all levels, while our generous variant computes *exact* flags during the last iteration step. Note that we do *not* use additional techniques in order to improve query performance, e.g. the *avoid binary search* technique (cf. [5] for details). For comparison, we also report the results for plain DIJKSTRA and unidirectional ALT [3].

We observe a good performance of SHARC in general. Queries for a 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 generous version but preprocessing is almost 8 times faster. Recall that the only difference between both version is the way arc-flags are computed during the last iteration step. Although the number of heap operations is nearly the same for running one label-correcting algorithm per boundary node as for growing two DIJKSTRA-trees, the former has to use functions as labels. As composing and merging functions is more expensive than adding and comparing integers, preprocessing times increase significantely.

Table 1. 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. *#delete mins* denotes the number of nodes removed from the priority queue, query *times* are given in milliseconds. Moreover, *speed-up* reports the speed-up over the corresponding value for plain DIJKSTRA.

	PREPRO			TIME-QUERIES				PROFILE-QUERIES			
	time	space	edge	#delete	speed	time	speed	#delete	speed	time	speed
technique	[h:m]	[B/n]	inc.	mins	up	[ms]	up	mins	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
gen 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 computing d_* instead of d yields an increase of about factor $4 - 7$ in terms of heap operations. 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 our SHARC-variants are capable of computing d_* for two random stations in less than 1 second.

5.2 Time-Dependent Road Networks

Unfortunately, we have no access to *large* real-world time-dependent road networks. Hence, we use available *real-world* time-independent networks and generate synthetic rush hours. As network, we use the largest strongly connected component of the road network of Western Europe, provided by PTV AG for scientific use. The graph has approximately 18 million nodes and 42.6 million edges and edge lengths correspond to (time-independent) travel times. Each edge 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 [12]. 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. For details, see [12]. We additionally adjust the degree of perturbation by assigning time-dependent edge-costs only to specific categories of edges. In a *low traffic* scenario, only motorways are time-dependent. The *medium traffic* scenario additionally includes congested national roads, and for the *high traffic* scenario, we pertube all edges except local and rural roads. For comparison, we also report the performance of *static* SHARC in a *no traffic* scenario, i.e., all edges are time-independent.

Table 2 reports the results of SHARC in our different scenarios. Note that we use the same parameters for all inputs and also report the speed-up over Dijkstra's algorithm in terms of query performance. Unfortunately, it turned out that this input is too big to use a label-correcting algorithm for computing arc-flags. Hence, we use Dijkstra-based approximation of arc-flags for all levels. Note that this type of preprocessing equates our economical variant from the last section. We observe that the degree of perturbation has a high influence on both preprocessing and query performance of SHARC. Preprocessing times increase if more edges are time-dependent. This is mainly due to our

Table 2. Performance of SHARC on our time-dependent European road network instance. Note that profile-queries are reported in *seconds*, while time-queries are given in *milliseconds*. Also note that we apply static SHARC for the no traffic scenario.

	Prepro			Time-Queries				Profile-Queries	
	time	space	edge	#delete	speed	time	speed	#delete	time
scenario	[h:m]	[B/n]	inc.	mins	up	[ms]	up	mins	[s]
no traffic	0:41	13.7	27%	997	8 830	0.42	13 369	-	-
low traffic	4:03	27.2	31%	34 123	261	26.12	214	37 980	35.28
medium traffic	6:10	45.6	32%	51 738	173	38.05	148	57 761	61.05
high traffic	8:31	112.4	34%	84 234	105	75.33	76	92 413	154.32

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 time-query performance of SHARC, we observe that in a our scenario where only motorways are time-dependent, SHARC provides speed-ups of up to 214 over DIJKSTRA. However, this values drops to 76 if more and more edges become time-dependent. The reason for this loss in query performance is the bad quality of our DIJKSTRA-based approximation. If more edges are time-dependent, upper- and lower-bounds are less tight than in a scenario with only few time-dependent edges. Comparing time- and profile-queries, we observe that the search-space only increases by $\approx 10\%$ 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 1200 times slower than time-queries. Comparing the figures from Tab. 1 and 2, we observe that speed-ups for time-queries in road networks are higher than in railway networks. However, the situation changes when running profile-queries. Here, timetable queries are much faster than queries in road networks. The reason for this is that composing functions needed for timetables is cheaper than those needed for road networks.

Summarizing, average time-query times are below 100 ms for all scenarios, while plain DIJKSTRA has query times of about 5.6 seconds. Moreover, for the probably most important scenario, i.e., the medium traffic scenario, SHARC provides query times of about 38 ms being sufficient for many applications. Moreover, SHARC allows profile-queries that cannot be handled by a plain label-correcting algorithm due to memory consumption.

6 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 of railways. Moreover, we are able to compute the distances between two nodes for all possible departure times.

Regarding future work, one could think of faster ways of composing, merging, and approximating piece-wise linear functions as this would directly accelerate preprocessing and, more importantly, profile-queries significantly. Another interesting question is, whether SHARC is helpful to run multi-criteria queries in time-dependent graphs. The good performance of the multi-metric variant of static SHARC [13] might be a good indicator that this works well. This is very interesting for timetable information systems as users may be willing to accept longer travel times if the required number of transfers is smaller.

Acknowledgments. The author thanks Veit Batz, Reinhard Bauer, Robert Görke, Riko Jacob, Bastian Katz, Ignaz Rutter, Peter Sanders, Dominik Schultes

and Dorothea Wagner for interesting discussions on time-dependent routing. Moreover, I thank Giacomo Nannicini for providing me with his time-dependent edge-cost generator, and Thomas Pajor for his work on timetable data parsers.

References

1. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1, 269–271 (1959)
2. Schultes, D.: Route Planning in Road Networks. PhD thesis, Uni. Karlsruhe (2008)
3. Delling, D., Wagner, D.: Landmark-Based Routing in Dynamic Graphs. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 52–65. Springer, Heidelberg (2007)
4. Schultes, D., Sanders, P.: Dynamic Highway-Node Routing. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 66–79. Springer, Heidelberg (2007)
5. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Efficient Models for Timetable Information in Public Transportation Systems. ACM J. of Exp. Al. 12, 2–4 (2007)
6. Cooke, K., Halsey, E.: The Shortest Route Through a Network with Time-Dependent Intermodal Transit Times. Journal of Mathematical Analysis and Applications, 493–498 (1966)
7. Kaufman, D.E., Smith, R.L.: Fastest Paths in Time-Dependent Networks for Intelligent Vehicle-Highway Systems Application. Journal of Intelligent Transportation Systems 1, 1–11 (1993)
8. Orda, A., Rom, R.: Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. Journal of the ACM 37, 607–625 (1990)
9. Hart, P.E., Nilsson, N., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics 4, 100–107 (1968)
10. Goldberg, A.V., Harrelson, C.: Computing the Shortest Path: A* Search Meets Graph Theory. In: 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), pp. 156–165 (2005)
11. Disser, Y., Müller-Hannemann, M., Schnee, M.: Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 347–361. Springer, Heidelberg (2008)
12. Nannicini, G., Delling, D., Liberti, L., Schultes, D.: Bidirectional A* Search for Time-Dependent Fast Paths. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 334–346. Springer, Heidelberg (2008)
13. Bauer, R., Delling, D.: SHARC: Fast and Robust Unidirectional Routing. In: 10th Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 13–26 (2008)
14. Hilger, M., Köhler, E., Möhring, R.H., Schilling, H.: Fast Point-to-Point Shortest Path Computations with Arc-Flags. In: 9th DIMACS Implementation Challenge - Shortest Paths (2006)
15. Dean, B.C.: Continuous-Time Dynamic Shortest Path Algorithms. Master's thesis, Massachusetts Institute of Technology (1999)
16. Pellegrini, F.: SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package (2007)
17. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Better Landmarks Within Reach. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 38–51. Springer, Heidelberg (2007)
18. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway Hierarchies Star. In: 9th DIMACS Implementation Challenge - Shortest Paths (2006)