SHARC: Fast and Robust Unidirectional Routing

REINHARD BAUER
Universität Karlsruhe (TH)
and
DANIEL DELLING
Universität Karlsruhe (TH)

During the last years, impressive speed-up techniques for DIJKSTRA's algorithm have been developed. Unfortunately, the most advanced techniques use *bidirectional* search which makes it hard to use them in scenarios where a backward search is prohibited. Even worse, such scenarios are widely spread, e.g., timetable-information systems or *time-dependent* networks.

In this work, we present a *unidirectional* speed-up technique which competes with bidirectional approaches. Moreover, we show how to exploit the advantage of unidirectional routing for fast exact queries in timetable information systems and for fast approximative queries in time-dependent scenarios. By running experiments on several inputs other than road networks, we show that our approach is very robust to the input.

Categories and Subject Descriptors: G.2.2 [**Graph Theory**]: Graph algorithms

General Terms: Algorithms, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Dijkstra's algorithm, speed-up technique

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 [Dijkstra 1959] 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. Due to the availability of huge road networks, recent research on shortest paths speed-up techniques solely concentrated on those networks [Demetrescu et al. 2006]. The fastest known techniques [Bauer et al. 2008] were developed for road networks and use specific properties of those networks in order to gain their enormous speed-ups.

A previous version appeared at the Workshop on Algorithm Engineering and Experiments (ALENEX'08).
This work was partially supported by the Future and Emerging Technologies Unit of EC (IST priority – 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).
Authors' addresses: Reinhard Bauer and Daniel Delling, Department of Computer Sciences, Universität Karlsruhe (TH), Box 6980, 76128 Karlsruhe, Germany; email: {rbauer,delling}@informatik.uni-karlsruhe.de.

However, these techniques perform a bidirectional query or at least need to know the exact target node of a query. In general, these hierarchical techniques step up a hierarchy—built during a preprocessing phase—starting both from source and target and perform a fast query on a very small graph. Unfortunately, in certain scenarios a backward search is prohibited, e.g. in timetable information systems and time-dependent graphs the time of arrival is unknown. One option would be to guess the arrival time and then to adjust the arrival time after forward and backward search have met. Another option is to develop a fast *unidirectional* algorithm.

In this work, we introduce SHARC-Routing, a fast and robust approach for *unidirectional* routing in large networks. The central idea of SHARC (Shortcuts + Arc-Flags) is the adaptation of techniques developed for Highway Hierarchies [Sanders and Schultes 2006] to Arc-Flags [Lauther 2004; Möhring et al. 2006; Hilger et al. 2006]. In general, SHARC-Routing iteratively constructs a contraction-based hierarchy during preprocessing and automatically sets *arc-flags* for edges removed during contraction. More precisely, arc-flags are set in such a way that a unidirectional query considers these removed *component*-edges only at the beginning and the end of a query. As a result, we are able to route very efficiently in scenarios where other techniques fail due to their bidirectional nature. It turned out that SHARC was a promising candidate for routing in time-dependent networks [Delling 2008]. Furthermore, SHARC allows to perform very fast queries—*without* updating the preprocessing—in scenarios where metrics are changed frequently, e.g. different speed profiles for fast and slow cars. We also introduce an interesting variant of SHARC by removing all shortcuts from the graph after preprocessing. This variant may be very helpful in scenarios with very limited memory, e.g., portable navigation systems. In case a user needs even faster query times, our approach can also be used as a bidirectional algorithm that outperforms the most prominent techniques. See Figure 1 for an example of a typical search space of uni- and bidirectional SHARC.

Related Work. Many speed-up techniques have been developed during the last years (see [Delling et al. 2009a] for an overview). Hence, we here focus on work that is directly connected to SHARC. To our best knowledge, two approaches exist that iteratively contract and prune the graph during preprocessing. This idea was introduced in [Sanders and Schultes 2005]. First, the graph is contracted and afterwards partial trees are built in order to determine highway edges. Non-highway edges are removed from the graph. The contraction was significantly enhanced in [Sanders and Schultes 2006] reducing preprocessing and query times drastically. The RE algorithm, introduced in [Goldberg et al. 2006; 2007], also uses the contraction from [Sanders and Schultes 2006] but pruning is based on reach values for edges. A technique relying solely on contraction is Contraction Hierarchies [Geisberger et al. 2008]. All those techniques build a hierarchy during the preprocessing and the query exploits this hierarchy. Moreover, these techniques gain their impressive speed-ups from using a bidirectional query, which—among other problems—makes it hard to use them in time-dependent graphs. Moreover, REAL [Goldberg et al. 2006; 2007]—a combination of RE and ALT—can be used in a unidirectional sense but still, the exact target node has to be known for ALT, which is unknown in timetable information systems (cf. [Pyrga et al. 2007] for details). Similar to Arc-Flags [Lauther 2004; Möhring et al. 2006; Hilger et al. 2006], Geometric Containers [Wagner

Fig. 1. Search space of a typical uni-(left) and bidirectional(right) SHARC-query. The source of the query is the upper flag, the target the lower one. Relaxed edges are drawn in black. The shortest path is drawn thicker. Note that the bidirectional query *only* relaxes shortest-path edges.

et al. 2005] attaches a label to each edge indicating whether this edge is important for the current query. However, Geometric Containers has a worse performance than Arc-Flags and preprocessing is based on computing a full shortest path tree from every node within the graph. For more details on Arc-Flags, see Section 2.

An extended abstract of SHARC has been published in [Bauer and Delling 2008]. Since its publication we were able to improve preprocessing, both with respect to running times and space consumption. We achieve these improvements by introducing additional preprocessing routines. Furthermore, we present a variant of SHARC which can be adapted to existing (commercial) systems easily and has a very low space consumption. Finally, we provide extensive parameter tests for SHARC yielding a better insight in the impact of parameter choice on the performance of SHARC.

Meanwhile [Delling 2008], we managed to augment SHARC to time-dependent networks. Note that this extension is *not* included in this work. We here focus on the time-independent variant of SHARC. However, we include first ideas how to use SHARC in time-dependent networks in a straightforward and approximate manner. We include these results for historical reasons since they were also published in [Bauer and Delling 2008].

Overview. This paper is organized as follows. Section 2 introduces basic definitions and reviews the Arc-Flag approach. Preprocessing and the query algorithm of our SHARC approach are presented in Section 3, while Section 4 presents initial ideas how SHARC can be used in an approximate time-dependent scenario. Our experimental study on real-world and synthetic datasets is located in Section 5 showing the excellent performance of SHARC on various instances. Our work is concluded by a summary and possible future work in Section 6.

2. PRELIMINARIES

Throughout the whole work we restrict ourselves to simple, directed graphs $G = (V, E, len)$ with positive length function $len : E \to \mathbb{R}^+$. The transpose graph $\overleftarrow{G} = (V, \overleftarrow{E})$ is the graph obtained from G by substituting each $(u, v) \in E$ by (v, u). Given a set of edges H, $tail(H)$ / $head(H)$ denotes the set of all tails / heads of edges in H. With $\deg_{in}(v)$ / $\deg_{out}(v)$ we denote the number of edges whose head / tail is v. 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. Note that connected components within the 1-shell are trees. Since each tree is attached to the 2-core, we call these trees *attached trees*.

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 $\{\mathcal{C}^0, \mathcal{C}^1, \ldots, \mathcal{C}^{L-1}\}$ such that for each $l < L-1$ and each $C_i^l \in \mathcal{C}^l$ a cell $C_j^{l+1} \in \mathcal{C}^{l+1}$ exists with $C_i^l \subseteq C_j^{l+1}$. In that case the cell C_j^{l+1} is called the *supercell* of C_i^l. The supercell of a level-$L-1$ cell is V. Note that the number of levels is denoted by L. 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 $(u, v) \in E$. We denote by $d(u, v)$ the distance according to len between two nodes u and v.

Arc-Flags

The original Arc-Flag approach, introduced in [Lauther 2004; Köhler et al. 2005], first computes a partition $\mathcal{C}$ of the graph and then attaches a *label* to each edge e. A label contains, for each cell $C \in \mathcal{C}$, a flag $AF_C(e)$ which is true if a shortest path to at least one node in C starts with e. A modified DIJKSTRA—from now on called Arc-Flags DIJKSTRA—then only considers those edges for which the flag of the target node's cell is true. The big advantage of this approach is its easy query algorithm. Furthermore, we observed that for long-range queries in road networks, an Arc-Flags DIJKSTRA often is optimal in the sense that it *only* visits those edges that are on the shortest path. However, preprocessing is very extensive, either regarding preprocessing time or memory consumption.

Preprocessing. of Arc-Flags is divided into two parts. First, the graph is partitioned into k several cells. The second step then computes k flags for each edge. The first approach for obtaining a partition based on a grid partition [Lauther 2004]. It turns out that the performance of an Arc-Flags query heavily depends on the partition used. In order to achieve good speed-ups, several requirements have to be fulfilled: cells should be connected, the size of the cells should be balanced, and the number of boundary nodes has to be low. A systematical experimental study of the impact of partitions on Arc-Flags has been published in [Möhring et al. 2006]. According to their work, the best results have been achieved if a METIS [Karypis 2007] partition is applied.

The second step of preprocessing is the computation of arc-flags. Throughout the years, several approaches have been introduced (see e.g. [Lauther 2004; Köhler et al.

2005; Möhring et al. 2005; Hilger et al. 2009; Lauther 2009]). We here concentrate on two approaches which turned out to be the most efficient. For both approaches, 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.

Boundary Shortest Path Trees. A true arc-flag $AF_C(e)$ denotes whether e has to be considered for a shortest-path query targeting a node within C. This can be computed as follows: Grow a shortest path tree in $\overleftarrow{G}$ from all boundary nodes $b \in B_C$ of all cells C. Then set $AF_C(u, v) =$ true if (u, v) is a tree edge for at least one tree grown from all boundary nodes $b \in B_C$.

Centralized Approach. The drawback of the first approach is that we have to grow $|B|$ shortest path trees yielding long preprocessing times for large transportation networks. [Hilger et al. 2009] introduces a new approach to computing flags. A label-correcting algorithm (also called centralized tree) is performed for each cell C. The algorithm propergates labels of size $|B_C|$ through the network depicting the distances to all boundary nodes of the cell. The algorithm terminates if no label can be improved any more. Then, $AF_C((u, v))$ is set to true if $len(u, v)+d(v, b) = d(u, b)$ holds for at least one $b \in B_C$.

Query. A unidirectional Arc-Flags query is a modified Dijkstra operating on the input graph. For a random s–t query, it first determines the target cell T, and then relaxes only those edges with set flag for cell T. Note that compared to plain Dijkstra, an Arc-Flags query performs only one additional check.

Note that $AF_{C_i}(e)$ is true for almost all edges $e \in C_i$. Due to these own-cell-flags an Arc-Flags Dijkstra yields no speed-up for queries within the same cell. Even worse, using a unidirectional query, more and more edges become important when approaching the target cell (called the *coning effect*) and finally, all edges are considered as soon as the search enters the target cell. While the coning effect can be weakened by a bidirectional query, the former also holds for such queries. Thus, a two-level approach is introduced in [Möhring et al. 2006] which weakens these drawbacks as cells become quite small on the lower level. It is obvious that this approach can be extended to a multi-level approach.

3. STATIC SHARC

In this section, we explain SHARC-Routing in *static* scenarios, i.e., the graph remains untouched between two queries. In general, the SHARC query is a standard multi-level Arc-Flags Dijkstra, while the preprocessing incorporates ideas from hierarchical approaches.

3.1 Preprocessing

Preprocessing of SHARC adopts ideas from hierarchical approaches like Highway Hierarchies and REAL. During the *initialization* phase, we extract the 2-core of the graph and perform a multi-level partition of G according to an input parameter P. The number of levels L is an input parameter as well. 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 of level i by growing a *partial* centralized shortest-path tree from each cell C_i^j. 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 1-shell nodes removed at the beginning. Figure 2 shows a scheme of the SHARC-preprocessing. In the following we explain each phase separately. We hereby restrict ourselves to arc-flags for the unidirectional variant of SHARC. However, the extension to computing bidirectional arc-flags is straight-forward.

3.1.1 *1-Shell Nodes.* First of all, we extract the 2-core of the graph as we can directly assign correct arc-flags to attached trees that are fully contained in a cell: Each edge whose head is a core node gets all flags assigned true while those directing away from the core only get their own-cell flag set true. By removing 1-shell nodes *before* computing the partition we ensure that an attached tree is fully contained in a cell by assigning all nodes in an to the cell of its 2-core root. After the last step of our preprocessing we simply reattach the nodes and edges of the 1-shell to the output graph.

3.1.2 *Multi-Level Partition.* According to [Möhring et al. 2006], the Arc-Flag method heavily depends on the partition used. The same holds for SHARC. In order to achieve good speed-ups, several requirements have to be fulfilled: cells should be connected, the size of the cells should be balanced, and the number of boundary nodes has to be low. In this work, we use a locally optimized partition obtained from SCOTCH [Pellegrini 2007]. For further details, see Section 5. The number of levels L and the number of cells per level are tuning-parameters.

3.1.3 *Contraction.* The graph is contracted by iteratively *bypassing* 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 $u \in tail(I)$ and for each $v \in head(O) \setminus \{u\}$ we introduce a new edge of the length $len(u, n) + len(n, v)$. If there already is an edge connecting u and

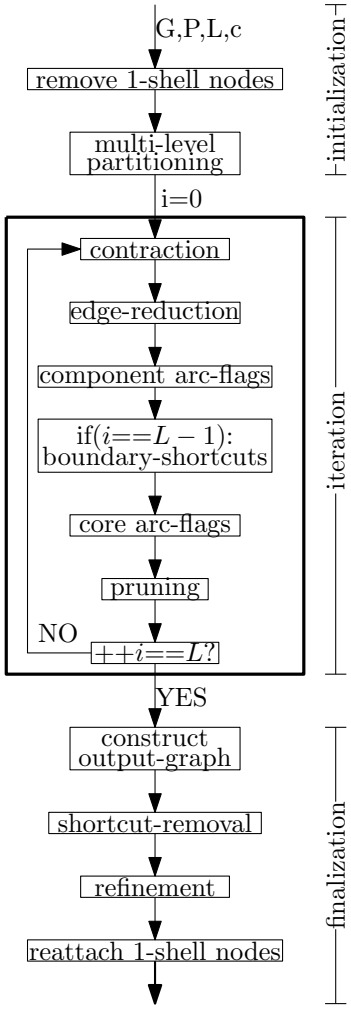

Fig. 2. Schematic representation of the preprocessing. Input parameters are the partition parameters P, the number of levels L, and the contraction parameter c.

v in the graph, we only keep the one with smaller length. 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. To check whether a node is bypassable we first determine the number $\#shortcut$ of *new* edges that would be inserted into the graph if n is bypassed, i.e., existing edges connecting nodes in $tail(I)$ with nodes in $head(O)$ do not contribute to $\#shortcut$. Then we say a node is bypassable iff the *bypass criterion* $\#shortcut \leq c \cdot (\deg_{in}(n) + \deg_{out}(n))$ is fulfilled, where c is a tunable *contraction parameter*.

A node being bypassed influences the degree of their neighbors and thus, their bypassability. Therefore, the order in which nodes are bypassed changes the resulting contracted graph. We use a heap to determine the next bypassable node. The key of a node n within the heap is $h \cdot \#shortcut/(\deg_{in}(n) + \deg_{out}(n))$ where h is the hop number of the hop-maximal shortcut that would be added if n was bypassed, smaller keys have higher priority. To keep the length of shortcuts limited we do not bypass a node if that results in adding a shortcut with 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*. In order to guarantee correctness, we 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.

Our contraction routine mainly follows the ideas introduced in [Sanders and Schultes 2006]. The idea to control the order, in which the nodes are bypassed using a heap is due to [Goldberg et al. 2006]. Finally, the idea to bound the hop number of a shortcut is due to [Delling et al. 2009b].

3.1.4 *Edge-Reduction.* Note that our contraction routine potentially adds shortcuts not needed for keeping the distances in the core correct. See Figure 3 for an example. Hence, we perform an edge reduction directly after contraction, similar to [Schultes and Sanders 2007]. We grow a shortest-path tree from each node u of the core. We stop the growth as soon as all neighbors v of u have been settled. Then we check for all neighbors whether $d(v) < len(u, v)$ holds. If it holds, we can remove (u, v) from the graph because the shortest path from u to v does not include (u, v). Note that if (u, v) is a very long edge, this routine explores almost the full graph since v is settled at a very late point. In order to solve this problem, we restrict the number of priority-queue removals to a fixed value k. In road networks, experiments indicate that $k = 10\,000$ is a reasonable choice. Note that we then may leave some unneeded edges in the graph.

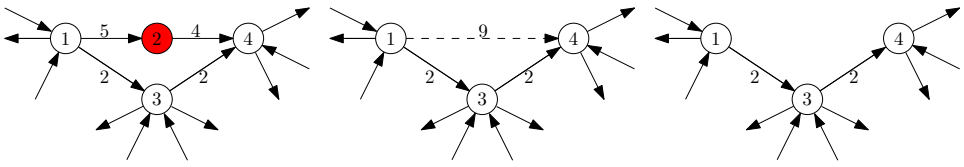

Fig. 3. Example for edge-reduction. The figure on the left depicts the input, edge labels indicate the weight of the edge. We contract, i.e., remove, node 2 and add an shortcut from node 1 to 4 with weight 9 (middle). However, the shortest path from 1 to 4 is via node 3 with length 4. Hence, we can safely remove the shortcut (1,4) from the core in order to preserve distances between core nodes. The resulting graph is shown on the right.

3.1.5 *Boundary-Shortcuts.* During our experimental study, we observed that—at least for long-range queries in road networks—a plain bidirectional Arc-Flags DIJKSTRA often is optimal in the sense that it visits *only* the edges on the shortest path between two nodes. However, such shortest paths may become quite long in road networks. One advantage of SHARC over Arc-Flags is that the contraction routine reduces the number of hops of shortest paths in the network yielding smaller search spaces. In order to further reduce this hop number we enrich the graph by additional shortcuts. In general we could try any shortcuts as our preprocessing favors paths with less hops over those with more hops and thus, added shortcuts are used for long range queries. However, adding shortcuts crossing cell-borders can increase the number of boundary nodes and hence, increase preprocessing time. Therefore, we use the following heuristic to determine good shortcuts: we add *boundary shortcuts* between some boundary nodes belonging to the same cell C at level $L-1$. In order to keep the number of added edges small we compute the betweenness [Brandes 2001] values c_B of the boundary nodes on the remaining core-graph. Recall that betweenness is a centrality measure: nodes that occur on many shortest paths between other nodes have higher betweenness than those that do not. Each boundary node with a betweenness value higher than half the maximum gets $3 \cdot \sqrt{|B_C|}$ additional outgoing edges. The heads are those boundary nodes with highest $c_B \cdot h$ values, where h is the number of hops of the added shortcut.

3.1.6 *Arc-Flags.* Our query algorithm is executed on the original graph enhanced by shortcuts added during the contraction phase. Thus, we have to assign arc-flags to each edge we remove during the contraction phase. One option would be to set every flag to true. However, we can do better. First of all, we keep all arc-flags that already have been computed for lower levels. We set the arc-flags of the current and all higher levels depending on the tail u of the deleted edge. If u 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 u 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. Finally, shortcuts get their own-cell flag *fixed* to false as relaxing shortcuts when the target cell is reached yields no speed-

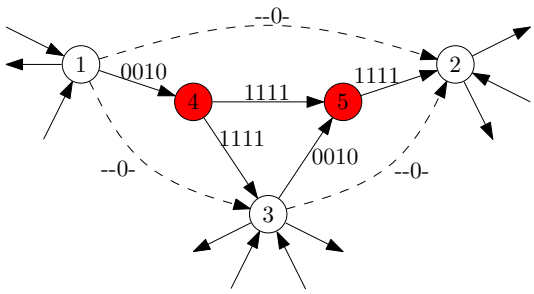

Fig. 4. Example for assigning arc-flags during contraction for a partition having four cells. All nodes are in cell 3. The red nodes (4 and 5) are removed, the dashed shortcuts are added by the contraction. Arc-flags (edge labels) are indicated by a 1 for true and 0 for false. The edges directing into the component get *only* their own-cell flag set true. All edges in and out of the component get full flags. The added shortcuts get their own-cell flags fixed to false.

up. See Figure 4 for an example. As a result, an Arc-Flags query only considers components at the beginning and the end of a query. Moreover, we reduce the search space.

Assigning Arc-Flags to Core-Edges. After the contraction phase and assigning arc-flags to removed edges, we compute the arc-flags of the core-edges of the current level i. As described in Section 2, we grow for each cell C one centralized shortest path tree on the transpose graph starting from every boundary node $n \in B_C$ of C. We stop growing the tree as soon as all nodes of C's supercell have a distance to each $b \in B_C$ greater than the smallest key in the priority queue used by the centralized shortest path tree algorithm. For any edge e that is in the supercell of C and that lies on a shortest path to at least one $b \in B_C$, we set $AF^i_C(e) = \mathsf{true}$.

Note that the centralized approach sets arc-flags to true for *all* possible shortest paths between two nodes. In order to favor boundary shortcuts, we extend the centralized approach by introducing a second matrix that stores the number of hops to every boundary node. With the help of this second matrix we are able to assign true arc-flags only to *hop-minimal* shortest paths. However, using a second matrix increases the high memory consumption of the centralized approach even further. Thus, we use this extension only during the last iteration step where the core is small.

3.1.7 *Pruning.* After computing arc-flags at the current level, we prune the input. We remove unimportant edges from the graph by running two steps. First, we identify *prunable* cells. A cell C is called prunable if all neighboring cells are assigned to the same supercell. Then we remove all edges from a prunable cell that have at most their own-cell bit set. For those edges no flag can be assigned true in higher levels as then at least one flag for the surrounding cells must have been set before.

3.1.8 *Refinement of Arc-Flags.* Our contraction routine described above sets all flags to true for almost all edges removed by our contraction routine. However, we can do better: we are able to *refine* arc-flags by *propagation* of arc-flags from higher to lower levels. Before explaining our propagation routine we need the notion of level. 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 build a partial shortest-path tree T starting at u, not relaxing edges with heads on a level smaller than $l(u)$. We stop the growth as soon as all nodes in the priority queue are *covered*. A node v is called covered as soon as a node between u and v—with respect to T—belongs to a level $> l(u)$. After the termination of the growth we remove all covered nodes from T resulting in a tree rooted at u and with leaves either in $l(u)$ or in a level higher than $l(u)$. Those leaves of the built tree belonging to a level higher than $l(u)$ we call *exit nodes* $\vec{N}(u)$ of u.

With this information we refine the arc-flags of all edges outgoing from u. First, we set all flags—except the own-cell flags—of all levels $\geq l(u)$ for all outgoing edges from u to false. Next, we assign exit nodes to outgoing edges from u. Starting at an exit node n_E we follow the predecessor in T until we finally end up in a node x

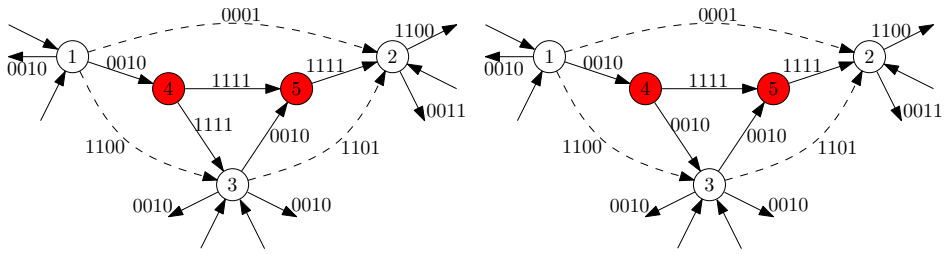

Fig. 5. Example for refining the arc-flags of outgoing edges from node 4. The figure in the left shows the graph from Figure 4 after the last iteration step. The figure on the right shows the result of our refinement routine starting at node 4.

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).

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. Figure 5 gives an example. Note that during refinement step i we only refine arc-flags of edges outgoing from nodes belonging to level i.

3.1.9 *Output Graph.* The *output graph* of the preprocessing consists of the original graph enhanced by all shortcuts that are in the contracted graph at the end of at least one iteration step. Note that an edge (u, v) may be contained in no shortest path because a shorter path from u to v already exists. This especially holds for the shortcuts we added to the graph. As a consequence, such edges have no flag set true after the last step. Thus, we can remove all edges from the output graph with no flag set true.

3.1.10 *Shortcut-Removal.* Note that the insertion of shortcuts is one of the main reasons why the output graph is larger than the input. 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 paths it represents. If all nodes on these shortest paths have less than 3 outgoing edges, we remove (u, v) from the graph and all edges being part of the shortest paths additionally inherit the arc-flags from (u, v). An example is given in Figure 6.

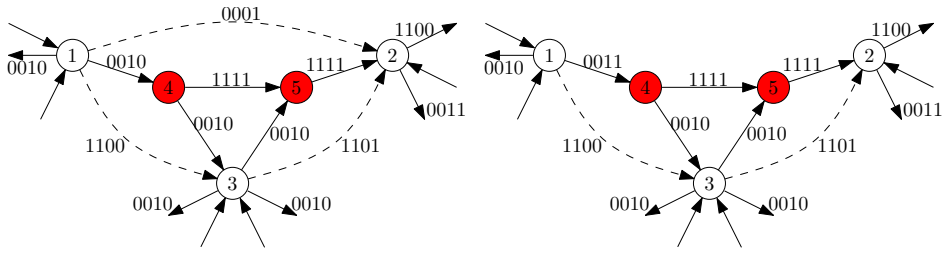

Fig. 6. Example for removing a shortcut from the output graph. The graph on the left shows the output graph after refinement of arc-flags (cf. Figure 5). The shortcut from 1 to 2 is removed and the edges representing the shortcut inherit the flags from it.

3.1.11 *Optimizations.* In order to improve both performance and space efficiency, we use two optimizations. Firstly, similar to [Goldberg et al. 2007], we increase cache efficiency of the output graph by reordering nodes according to the level they have been removed at from the graph. As a consequence, the number of cache misses is reduced yielding lower query times. Secondly, we compress the arc-flag information. During our studies, we observed that the number of *different* arc-flags is much less than the number of edges. Thus, instead of storing arc-flags for each edge, we use a separate array containing all possible *unique* arc-flags. In order to access the flags efficiently, we assign an additional pointer to each edge indexing the correct arc-flags. This yields a lower space consumption of our preprocessed data.

Stripped SHARC. Note that we could use our shortcut-removal routine to remove *all* shortcuts we added during preprocessing. Our output graph then equals the original input, with additional region information and arc-flags for each edge. As a result, such a variant of SHARC can be interpreted as a faster preprocessing routine for multi-level arc-flags. However, we might set more flags to `true` than necessary.

The advantage of this variant is its easy adaptability to existing (commercial) systems. The existing core system may stay untouched; we simply add an arc-flag pointer to each edge, a region information to each node, and store the arc-flag array. Furthermore, the space consumption is very low, as shortcuts are one of the main reasons of space overhead. Finally, this variant needs no shortcut-unpacking routine if the complete path description is required. Summarizing, the variant may be very helpful for PDA-implementations where space is limited and users need the complete path. However, the disadvantage of this approach is its worse performance than SHARC with shortcuts (cf. Section 5).

3.2 Query

Basically, our query is a multi-level Arc-Flags Dijkstra adapted from the two-level Arc-Flags Dijkstra presented in [Möhring et al. 2006]. The query is 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. When relaxing the edges outgoing from n, we consider only those edges having an arc-flag set on level i for the corresponding cell of t. It is proven [Möhring et al. 2006] that Arc-Flags performs correct queries. However, as our preprocessing is different, we have to prove Theorem 3.1.

Theorem 3.1. *The distances computed by SHARC are correct with respect to the original graph.*

The proof can be found in Appendix A. We want to point out that the SHARC query, compared to plain Dijkstra, only needs to perform two additional operation: computing the common level of the current node and the target and the arc-flags evaluation. Thus, our query is very efficient with a much smaller overhead compared to other hierarchical approaches. Note that SHARC uses shortcuts which have to be unpacked for determining the shortest path (if not only the distance is queried). However, we can directly use the methods from [Delling et al. 2009b], as our contraction works similar to Highway Hierarchies.

Path-Expansion. During our experimental evaluation, we observed that many nodes have only one outgoing edge for which the arc-flag of the corresponding target is set to true for the current query (cf. Figure 1). We call this property the *no-choice* property and the specific edge with the flag set to true the *no-choice* edge. With this observation at hand, we can use the following optimization to speedup the query. Whenever we insert a node u into the priority queue fulfilling the no-choice property, we skip this node and insert the head v of the no-choice edge into the queue. If the no-choice property also holds for v, we also skip v. We skip nodes until we either insert t, the target of the query, or insert a node for which the no-choice property does not hold. Note that *path-expansion* is especially helpful for our stripped variant of SHARC. Here, path-expansion partly remedies the drawback of lacking shortcuts.

Multi-Metric Query. In [Bauer et al. 2007b], we observed that the shortest path structure of a graph—as long as edge weights somehow correspond to travel times—hardly changes when we switch from one metric to another. Thus, one might expect that arc-flags are similar to each other for these metrics. We exploit this observation for our *multi-metric* variant of SHARC. During preprocessing, we compute arc-flags for all metrics and at the end we store only *one* arc-flag per edge by setting a flag true as soon as the flag is true for at least one metric. An important precondition for multi-metric SHARC is that we use the same partition for each metric. Note that the structure of the core computed by our contraction routine is independent of the applied metric.

4. APPROXIMATE TIME-DEPENDENT SHARC

Up to this point, we have shown how preprocessing works in a *static* scenario. As our query is unidirectional it seems promising to use SHARC in a *time-dependent* scenario. In this section we present how to perform approximate queries in time-dependent graphs with SHARC. In general, we assume that a time-dependent network $\overrightarrow{G} = (V, \overrightarrow{E})$ derives from an independent network $G = (V, E)$ by *increasing* edge weights at certain times of the day. For road networks these increases represent rush hours. The idea is to compute *approximative* arc-flags in G and to use these flags for routing in $\overrightarrow{G}$. In order to compute approximative arc-flags, we relax our criterion for setting arc-flags. Recall that for exact flags, $AF_C((u,v))$ is set true if $d(u,b) + len(u,v) = d(v,b)$ holds for at least one $b \in B_C$. For γ-approximate flags (indicated by $\overline{AF}$), we set $\overline{AF}_C((u,v)) =$ true if equation $d(u,b) + len(u,v) \leq \gamma \cdot d(v,b)$ holds for at least one $b \in B_C$. Note that we *only* have to change this criterion in order to compute approximative arc-flags instead of exact ones by our preprocessing. However, we do *not* add boundary shortcuts as this relaxed criterion does not favor those shortcuts. In order to perform queries in $\overrightarrow{G}$, we apply our modifications from Section 3.2 to a time-dependent Dijkstra [Cooke and Halsey 1966] in a straight-forward manner. Note that by computing arc-flags in G instead of $\overrightarrow{G}$ we loose correctness for time-dependent queries. It is easy to see that there exists a trade-off between performance and quality. Low γ-values yield low query times but the error-rate may increase, while a large γ reduces the error rate of γ-SHARC but yields worse query performance, as much more edges are relaxed during the query than necessary.

Exact Time-Dependent SHARC. Note that we include the above results for historical reasons. Meanwhile [Delling 2008], we presented an exact variant of time-dependent SHARC. Recall that in a static, i.e., time-independent, setup a set arc-flag denotes that the edge is important for the corresponding cell. The key idea for exact time-dependent routing is to set an arc-flag to `true` as soon as it is important for *at least one departure time*. For details how to incorporate this intuition correctly, we refer the interested reader to [Delling 2008; 2009].

5. EXPERIMENTS

In this section, we present an extensive experimental evaluation of our SHARC-Routing approach. To this end, we evaluate the performance of SHARC in various scenarios and inputs. 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 4.2.1, using optimization level 3.

Implementation Details. Our implementation is written in C++ using solely the STL. As priority queue we use a binary heap. Our graph is represented as adjacency array implementation. As described in [Schultes 2008], we have to store each edge twice if we want to iterate efficiently over incoming and outgoing edges. However, especially in road networks, many edges are undirected. Thus, the authors propose to compress edges if head and length of incoming and outgoing edges are equal. However, SHARC allows an even simpler implementation. During preprocessing we only operate on the transpose graph and thus do *not* iterate over outgoing edges while during the query we *only* iterate over outgoing edges. As a consequence, we only have to store each edge once (for preprocessing at its head, for the query at its tail). Thus, another advantage of our unidirectional SHARC approach is that we can reduce the memory consumption of the graph. Note that this does not hold for our bidirectional SHARC variant which needs considerably more space (cf. Table V).

Setup. Unless otherwise stated, we use a *unidirectional* variant of SHARC. We use $c = 2.5$ as contraction parameter and $h = 10$ as hop-bound. We use our path-expansion optimization only for our stripped variant of SHARC. In the following we report preprocessing effort and query performance of all speed-up techniques. For the former we report the preprocessing time, the increase in number of edges of the output graph compared to the input, and the resulting *additional* space per node. For query performance, we report the average number of settled nodes, i.e., the number of nodes taken from the priority queues, and resulting query times. At certain points, we also report the number of edges our algorithm relaxes. All figures in this paper 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 efficient implementation for unpacking shortcuts due to [Delling et al. 2009b] needs about 4 additional bytes per node of preprocessed data. Then it takes less than 0.5 ms to unpack a shortest path. This value can be reduced to less than 0.2 ms for the price of a slight increase in preprocessed data. For details, see [Delling et al. 2009b].

We report two types of queries. For *random queries*, 10 000 random pairs of source and target are selected, while for *local queries* [Sanders and Schultes 2005], 1 000 (s, t) pairs are chosen for each Dijkstra rank: Starting a query from s, the rank of t is denoted by the number of settled nodes before t is settled. It is given for $2^0, 2^1, \ldots, 2^{\log |V|}$. This setup is applied to some speed-up techniques in order to gain further insights into their performance on a particular graph depending on the length of a query. The results are presented in the form of a box-and-whisker plot.

5.1 Parameter Tests

We start our experimental evaluation with various parameter tests. As input we use the largest strongly connected component of the road network of Western Europe, provided by PTV AG for scientific use. It has approximately 18 million nodes and 42.6 million edges and edge lengths correspond to travel times. On this input, a plain DIJKSTRA settles $\approx$ 9 millions nodes and relaxes $\approx$ 21 million edges in 5.1 secondes on average when running random queries.

Multi-Level Partition. In [Möhring et al. 2005; 2006], the best results for Arc-Flags were achieved by applying a graph partitioning obtained by METIS [Karypis and Kumar 1998]. However, in our experimental study we observed two downsides of METIS: On the one hand, cells are sometimes disconnected and the number of boundary nodes is quite high. Thus, we also tested PARTY [Monien and Schamberger 2004] and SCOTCH [Pellegrini 2007] for partitioning. The former produces connected cells but for the price of an even higher number of boundary nodes. SCOTCH has the lowest number of boundary cells, but connectivity of cells cannot be guaranteed. Due to this low number of boundary nodes, we use SCOTCH and improve the obtained partitioning by adding smaller pieces of disconnected cells to neighbor cells. As a result, constructing and optimizing a partition can be done in less than 5 minutes for all inputs used. Table I reports the performance of SHARC if different types of SCOTCH-partitions are applied.

We observe that the performance of SHARC highly depends on the partition of the graph. A classic 1-level setup yields query times of 23.6 ms. By increasing the number of levels, we achieve query times of down to 0.29 ms. Interestingly, the preprocessing time is almost the same for all applied partitions: We need roughly 1.5 hours for preprocessing. However, using more than 6 levels does not pay off: query times stay the same but the overhead increases, mainly due to more shortcuts added to the graph. In general, it seems as if the best trade-off between preprocessing effort and query performance is achieved if the average number of nodes per cell is roughly 80. This value is achieved in a 6-level setup with 4,4,4,4,8,104 cells. Hence, we use this partition for our continental-size road networks for the rest of this work.

Contraction Rate. Next, we check whether our choice of contraction parameter is useful. Table II shows the performance of SHARC with various contraction rates if our default 6-level partition with 4,4,4,4,8,104 cells is given. We observe a contraction rate other than 2.5 increases preprocessing space. While $c = 3.0$ increases query performance marginally, a lower contraction rate also yields worse query times. Hence, our choice of $c = 2.5$ is reasonable in this setup.

Table I. Performance of SHARC with different partitions. Column *prepro* shows the computation time of the preprocessing in hours and minutes and the eventual *additional* bytes per node needed for the preprocessed data. In addition, the increase in number of edges over the input is given. For queries, the search space is given in the number of settled nodes and the number of relaxed edges, execution times are given in milliseconds.

PARTITION										PREPRO			QUERY		
#cells per level									⌀nodes	time	space	edge	#sett.	#rel.	time
l_0	l_1	l_2	l_3	l_4	l_5	l_6	l_7	#cells	per cell	[h:m]	[B/n]	inc.	nodes	edges	[μs]
128	-	-	-	-	-	-	-	128	140 705	1:52	6.0	4.2%	78 429	178 103	23 306
8	120	-	-	-	-	-	-	960	18 761	1:14	9.8	14.7%	11 362	26 323	3 049
4	4	120	-	-	-	-	-	1 920	9 380	1:24	10.6	17.8%	5 982	14 128	1 637
4	8	116	-	-	-	-	-	3 712	4 852	1:25	10.8	17.9%	3 459	8 372	983
8	8	112	-	-	-	-	-	7 168	2 513	1:36	11.5	18.0%	2 182	5 389	667
16	16	96	-	-	-	-	-	24 576	733	2:12	13.1	18.4%	1 217	3 169	428
4	4	4	116	-	-	-	-	7 424	2 426	1:20	11.2	19.4%	2 025	5 219	625
4	4	8	112	-	-	-	-	14 336	1 256	1:14	11.6	19.9%	1 320	3 544	441
4	8	8	108	-	-	-	-	27 648	651	1:15	12.4	20.4%	984	2 755	358
4	8	16	100	-	-	-	-	51 200	352	1:17	13.1	21.2%	819	2 357	319
4	4	4	4	112	-	-	-	28 672	628	1:12	12.0	21.2%	957	2 827	360
4	4	4	8	108	-	-	-	55 296	326	1:13	13.0	22.3%	774	2 337	309
4	4	8	8	104	-	-	-	106 496	169	1:18	13.7	23.7%	700	2 153	294
4	4	4	16	100	-	-	-	102 400	176	1:16	13.7	23.6%	703	2 162	295
4	4	8	16	96	-	-	-	196 608	92	1:24	15.0	25.5%	671	2 066	287
4	8	8	16	92	-	-	-	376 832	48	1:30	16.0	27.7%	663	2 046	288
4	4	4	4	4	108	-	-	110 592	163	1:15	13.6	24.3%	695	2 263	299
4	**4**	**4**	**4**	**8**	**104**	**-**	**-**	**212 992**	**85**	**1:21**	**14.5**	**26.5%**	**654**	**2 116**	**290**
4	4	4	8	8	100	-	-	409 600	44	1:28	16.1	29.5%	645	2 087	290
4	4	4	8	16	92	-	-	753 664	24	1:31	17.7	33.9%	646	2 028	289
4	4	8	8	16	88	-	-	1 441 792	13	1:50	19.8	41.7%	663	2 085	296
4	4	4	4	4	4	104	-	425 984	42	1:27	15.6	30.3%	649	2 209	299
4	4	4	4	4	8	100	-	819 200	22	1:31	17.6	35.3%	628	2 094	289
4	4	4	4	8	8	96	-	1 572 864	12	1:46	19.3	40.1%	637	2 092	294
4	4	4	4	8	16	88	-	2 883 584	6	1:54	20.7	41.7%	663	2 100	303
4	4	4	8	8	16	84	-	5 505 024	3	2:00	21.5	41.7%	655	2 035	294
4	4	4	4	4	4	4	100	1 638 400	11	1:43	19.2	40.7%	650	2 247	308
4	4	4	4	4	4	8	96	3 145 728	6	1:56	20.0	42.4%	627	2 113	293
4	4	4	4	4	8	8	92	6 029 312	3	1:51	21.1	43.1%	649	2 121	300
4	4	4	4	4	8	16	84	11 010 048	2	2:03	21.5	42.6%	648	2 035	296

Table II. Performance of SHARC with varying contraction parameter.

	PREPRO			QUERY		
	time	space	edge	#settled	#relaxed	time
c	[h:m]	[B/n]	inc.	nodes	edges	[μs]
1.0	1:40	15.1	21.3%	1 572	3 705	578
1.5	1:20	14.8	23.8%	886	2 464	348
2.0	1:20	14.7	25.5%	714	2 171	301
2.5	1:21	14.5	26.5%	654	2 116	290
3.0	1:23	14.6	27.2%	622	2 109	286

Table III. Performance of SHARC for varying effort computing arc-flags. *Core levels* indicates during which iteration steps, core flags are computed. *Refinement levels* depict the levels on which arc-flags are refined.

ARC-FLAGS		PREPRO			QUERY		
core levels	refinement levels	time [h:m]	space [B/n]	edge inc.	#settled nodes	#relaxed edges	time [μs]
-	-	0:16	12.8	27.1%	204 518	960 653	76 640
5	5	0:24	13.2	26.9%	23 313	70 225	6 021
5	4-5	0:24	13.2	26.9%	6 583	23 038	1 843
5	3-5	0:25	13.3	26.9%	2 394	11 547	856
5	2-5	0:27	13.6	26.9%	1 350	8 721	611
5	1-5	0:29	13.7	26.9%	1 127	8 091	553
4-5	4-5	0:30	13.7	26.7%	6 186	18 170	1 626
4-5	3-5	0:30	13.7	26.7%	2 042	6 683	648
4-5	2-5	0:31	13.7	26.7%	993	3 883	405
4-5	**1-5**	**0:34**	**13.7**	**26.7%**	**784**	**3 338**	**355**
3-5	3-5	0:35	13.8	26.6%	1 974	5 962	615
3-5	2-5	0:37	14.2	26.5%	933	3 161	371
3-5	1-5	0:39	14.2	26.5%	729	2 629	323
2-5	2-5	0:44	14.3	26.5%	900	2 862	354
2-5	1-5	0:46	14.3	26.5%	696	2 335	305
1-5	1-5	0:54	14.5	26.5%	684	2 236	300
0-5	0-5	1:21	14.5	26.5%	654	2 116	290

Reduction of Preprocessing Duration. SHARC exploits two aspects of a network in order to speed up the query: hierarchical properties by contraction, goal-direction by arc-flags. Table III shows the performance of SHARC if we do *not* compute arc-flags for all parts of the graph. This can be achieved by either *not* computing core arc-flags on lower levels or not refining low-level arc-flags. If we skip core arc-flags computation, we simply set all flags to true. Hence, we are able to reveal the main reasons for the good performance of SHARC.

We observe that SHARC is already 65 times faster than pure DIJKSTRA if we do not compute any arc-flags at all. Note that this speed-up is achieved with a preprocessing lasting only 16 minutes. By computing arc-flags on different levels we can vary the trade-off between preprocessing effort and query performance: 34 minutes of preprocessing already yields query times of 355 μs. Hence, an additional preprocessing of 18 minutes (over a pure hierarchical setup) accelerates SHARC by an additional factor of 200. Computing arc-flags for the remaining levels costs another 47 minutes but query performance only increases by 20%. Summarizing, dropping goal-direction on lower levels of the hierarchy reduces preprocessing significantly without a dramatic decrease in query performance.

In the following, we call SHARC with arc-flags computation on all levels the *generous* variant. Our *economical* variant sets core arc-flags only on the two topmost levels and refines flags for all levels except the lowest one.

Stripped SHARC. In Section 3 we discussed that we can remove *all* shortcuts during the last step of preprocessing. Table IV reports the performance of stripped SHARC with different contraction parameters during preprocessing. Note that in contrast to the figures given in Table II, we do *not* add boundary shortcuts since they are removed anyway at the end. Moreover, we do not use our locality opti-

Table IV. Performance of stripped SHARC with varying contraction rate during preprocessing.

	SHARC				stripped SHARC			
	Prepro		Query		Prepro		Query	
	time	space	#settled	time	time	space	#settled	time
c	[h:m]	[B/n]	nodes	[ms]	[h:m]	[B/n]	nodes	[ms]
0.50	7:32	13.8	10 876	3.38	7:48	7.8	14 697	4.76
0.75	4:29	14.3	5 420	1.99	4:43	7.7	62 303	26.03
1.00	1:45	15.1	1 997	0.90	2:03	7.5	1 891 320	1 096.48

mization but turn on path-expansion. It turns out that stripped SHARC requires a smaller contraction rate during preprocessing than normal SHARC. A contraction rate of 1.0 already yields very bad query performance for the stripped variant. However, applying a contraction rate of 0.5, the gap between normal and stripped SHARC almost closes. The disadvantage of such a low contraction rate is preprocessing time: it increases to almost 8 hours. However, as already mentioned, stripped SHARC should mainly be used in scenarios with limited memory, e.g., PDAs. Hence, preprocessing would be done once on a server and the preprocessed data would then be transfered to a PDA.

5.2 Static Environment

We continue our experimental evaluation with various tests for the *static* scenario. We hereby focus on road networks but also evaluate graphs derived from timetable information systems and synthetic datasets that have been evaluated in [Bauer et al. 2007a].

5.2.1 *Road Networks.* As inputs we again use the largest strongly connected component of the road network of Western Europe. Moreover, we use the US road network which is taken from the DIMACS homepage [Demetrescu et al. 2006]. It has approximately 23.9 million nodes and 58.3 million edges and edge lengths correspond to travel times.

Random Queries. Table V reports the results of our different SHARC-variants (cf. Section 5.1) compared to the most prominent speed-up techniques. More precisely, we report the results of our economical, generous, and stripped version of SHARC compared to Highway Hierarchies (results taken from [Schultes 2008]), Contraction Hierarchies [Geisberger et al. 2008], REAL [Goldberg et al. 2007]. Arc-Flags [Hilger 2007], CHASE [Bauer et al. 2008], and Transit Node Routing [Geisberger et al. 2008]. In addition, we report the results of bidirectional SHARC which uses bidirectional search in connection with a 2-level partition (16 cells per supercell at level 0, 112 at level 1).

We observe excellent query times for SHARC in general. Interestingly, SHARC has a lower preprocessing time for the US than for Europe but for the price of worse query performance. This is due to the fact that the average hop number of shortest paths are bigger for the US than for Europe. However, the number of boundary nodes is smaller for the US yielding lower preprocessing effort. The bidirectional variant of SHARC has a more extensive preprocessing: both time and additional space increase, which is due to computing and storing forward and backward arc-flags. Comparing query performance, bidirectional SHARC is clearly superior to

the unidirectional variant. This is due to the known disadvantages of uni-directional Arc-Flags: the coning effect and no arc-flag information as soon as the search enters the target cell (cf. Section 2 for details). The stripped variant is more than one order of magnitude slower than SHARC with shortcuts, and preprocessing times are higher. However, the strength of this approach is its easy adaptability to existing routing implementation. Still, stripped SHARC is about three orders of magnitude faster than plain DIJKSTRA.

Table V. Performance of different SHARC variants and the most prominent speed-up techniques on the European and US road network with travel times. *Prepro* shows the computation time of the preprocessing in hours and minutes and the eventual *additional* bytes per node needed for the preprocessed data. For queries, the search space is given in the number of settled nodes, execution times are given in *microseconds*. Note that other techniques have been evaluated on slightly different computers.

	Europe				USA			
	PREPRO		QUERY		PREPRO		QUERY	
	time [h:m]	space [B/n]	#settled nodes	time [μs]	time [h:m]	space [B/n]	#settled nodes	time [μs]
generous SHARC	1:21	14.5	654	290	0:58	18.1	865	376
economical SHARC	0:34	13.7	784	355	0:38	17.2	1 230	578
stripped SHARC	7:48	7.8	14 697	4 762	6:41	9.2	38 817	12 719
bidirectional SHARC	2:38	21.0	125	65	2:34	23.1	254	118
Highway Hierarchies	0:19	48	709	610	0:17	34	925	670
Contraction Hierarchies	0:32	-3.0	359	154	0:27	-2.3	278	132
REAL-(64,16)	2:21	32	679	1 110	2:01	43	540	1 050
Arc-Flags	17:08	18.9	2 369	1 600	10:10	9.9	8 180	4 300
CHASE (Arc-Flags+CH)	1:39	12	45	17	3:48	11	49	19
Transit Node	2:44	251	NA	3.4	1:30	220	NA	3.0

Comparing SHARC with other techniques, we observe that SHARC can compete with almost all bidirectional approaches. Unidirectional SHARC is only surpassed by Contraction Hierarchies(CH), CHASE (a combination of CH and Arc-Flags), and Transit Node Routing. However, the latter requires much more space than SHARC, and the other approaches cannot be used in a unidirectional manner easily. Bidirectional SHARC is faster than CH, but slower than CHASE. SHARC and CHASE are similar to each other, both exploit hierarchical properties of the network by contraction and goal-direction by arc-flags. However, CHASE focuses on hierarchical properties, SHARC on goal-direction. It seems as if in this setup, CHASE is superior due to its more sophisticated hierarchical properties.

Interestingly, for Europe, SHARC settles roughly the same number of nodes as Highway Hierarchies or REAL, but query times are smaller. This is due to the very low computational overhead of SHARC. Regarding preprocessing, SHARC uses less space than REAL or Highway Hierarchies. The computation time of the preprocessing is similar to REAL but longer than for Highway Hierarchies. The bidirectional variant uses more space and has longer preprocessing times, but the performance of the query is very good. Compared to the Arc-Flags, SHARC significantely reduces preprocessing time and query performance is better.

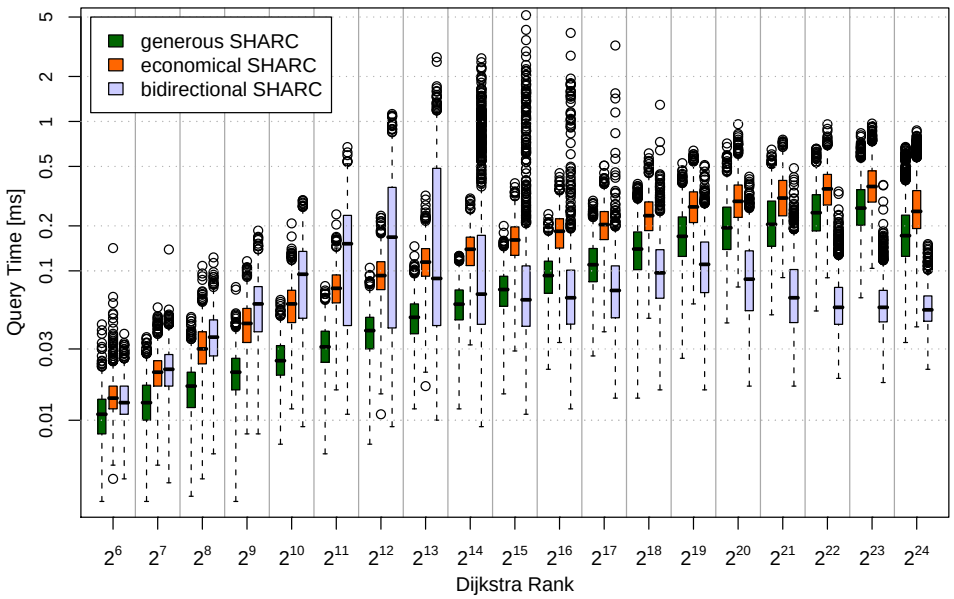

Fig. 7. Comparison of generous, economical, and bidirectional SHARC using the Dijkstra rank methodology. The results are represented as box-and-whisker plot: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

Local Queries. Figure 7 reports the query times of generous, economical, and bidirectional SHARC with respect to the Dijkstra rank. For an s-t query, the Dijkstra rank of node v is the number of nodes removed from the priority queue by DIJKSTRA's algorithm before v is removed. Thus, it is a kind of distance measure. As input we again use the European road network instance. Note that we use a logarithmic scale. Both economical and generous SHARC get slower with increasing rank but the median stays below 0.4 ms for the economical variant. The corresponding figure for the generous variant is 0.23 ms. We observe that the gap between both unidirectional variants is almost the same for all ranks. Comparing uni- and bidirectional SHARC, we observe that the former is faster for low-range queries while the latter wins for long-range queries. This is mainly due to the lower number of levels of the bidirectional setup: query times increase up to ranks of 2^{13} which is roughly the size of cells at the lowest level. Above this rank query times decrease and increase again till the size of cells at level 1 is reached. As we use more levels in a unidirectional setup, this effect deriving from the partition cannot be observed for the unidirectional variant. Comparing uni- and bidirectional SHARC we observe more outliers for the latter which is mainly due to less levels. Still, all outliers are below 5.2 ms.

Multi-Metric Queries. The original dataset of Western Europe contains 13 different road categories. By applying different speed profiles to the categories we obtain different metrics. Table VI gives an overview of the performance of (economical) SHARC when applied to metrics representing typical average speeds of slow/fast

Table VI. Performance of SHARC on different metrics using the European road instance. *Multi-metric* refers to the variant with one arc-flag and three edge weights (one weight per metric) per edge, while *single* refers to running SHARC on the applied metric.

	linear				fast car				slow car			
	Prepro		Query		Prepro		Query		Prepro		Query	
metric	time [h:m]	space [B/n]	#sett. nodes	time [μs]	time [h:m]	space [B/n]	#sett. nodes	time [μs]	time [h:m]	space [B/n]	#sett. nodes	time [μs]
single	0:34	13	784	355	0:28	14	804	364	0:35	13	779	349
multi	1:38	16	976	469	1:38	16	964	464	1:38	16	948	455

cars. Moreover, we report results for the *linear* profile which is most often used in other publications and is obtained by assigning average speeds of 10, 20, ..., 130 to the 13 categories. Finally, results are given for multi-metric SHARC, which stores only *one* arc-flag for each edge.

As expected, SHARC performs very well on other metrics based on travel times. Strikingly, the loss in performance is only very little when storing only one arc-flag for all three metrics. However, the overhead increases due to storing more edge weights for shortcuts and the size of the arc-flags vector increases slightly. Due to the fact that we have to compute arc-flags for all metrics during preprocessing, the computational effort increases.

5.2.2 *Timetable Information Networks.* Unlike former bidirectional approaches, SHARC-Routing can be used for timetable information. In general, two approaches exist to model timetable information as graphs: time-dependent and time-expanded networks (cf. [Pyrga et al. 2007] for details). In such networks timetable information can be obtained by running a shortest path query. However, in both models a backward search is prohibited as the time of arrival is unknown in advance. Table VII reports the results of SHARC on 2 time-expanded networks: The first represents the local traffic of Berlin/Brandenburg, has 2 599 953 nodes and 3 899 807 edges, the other graph depicts long distance connections of Europe (1 192 736 nodes, 1 789 088 edges). The networks are based on real-world data provided by HAFAS AG for scientific use. For comparison, we also report results for plain Dijkstra.

Table VII. Performance of plain Dijkstra and SHARC on a local and long-distance time-expanded timetable networks, unit disk graphs (udg) with average degree 5 and 7, and grid graphs with 2 and 3 number of dimensions. Due to the smaller size of the input, we use a 2-level partition with 16,112 cells.

	Prepro			Query		Prepro			Query	
	time [h:m]	space [B/n]	edge inc.	#settled nodes	time [ms]	time [h:m]	space [B/n]	edge inc.	#settled nodes	time [ms]
railways	local traffic					long distance				
Dijkstra	0:00	0	0.0%	1 299 830	406.2	0:00	0	0.0%	609 352	221.2
SHARC	10:02	9	24.5%	11 006	3.8	3:29	15	18.5%	7 519	2.2
unit disk	average deg. 5					average deg. 7				
Dijkstra	0:00	0	0.0%	487 818	257.3	0:00	0	0.0%	521 874	330.1
SHARC	0:01	16	3.1%	568	0.3	0:10	42	16.7%	1 835	1.0
grids	2 dimensional					3 dimensional				
Dijkstra	0:00	0	0.0%	125 675	36.7	0:00	0	0.0%	125 398	78.6
SHARC	0:32	60	55.9%	1 089	0.4	1:02	97	35.7%	5 839	1.9

For time-expanded railway graphs we observe an increase in performance of factor 100 over plain DIJKSTRA but preprocessing is still quite high which is mainly due to the partition. The number of boundary nodes is very high yielding high preprocessing times. However, compared to other techniques (see [Bauer et al. 2007a]) SHARC (clearly) outperforms any other technique when applied to timetable information system.

5.2.3 *Other inputs.* In order to show the robustness of SHARC-Routing we also present results on synthetic data. On the one hand, 2- and 3-dimensional grids [Goldberg et al. 2006] are evaluated. The number of nodes is set to 250 000, and thus, the number of edges is 1 and 1.5 million, respectively. Edge weights are picked uniformly at random from 1 to 1000. On the other hand, we evaluate random geometric graphs—so called unit disk graphs—which are widely used for experimental evaluations in the field of sensor networks. Such graphs are obtained by arranging nodes uniformly at random on the plane and connecting nodes with a distance below a given threshold. By applying different threshold values we vary the density of the graph. In our setup, we use graphs with about 1 000 000 nodes and an average degree of 5 and 7, respectively. As metric, we use the distance between nodes according to their embedding. The results can be found in Table VII.

We observe that SHARC provides very good results for all inputs. For unit disk graphs, performance gets worse with increasing degree as the graph gets denser. The same holds for grid graphs when increasing the number of dimensions.

5.3 Time-Dependency

Our final testset is performed on a time-dependent variant of the European road network instance. Note that these tests were conducted before the publication of [Delling 2008]. The implementation these results are based on is clearly inferior to the one used in [Delling 2008]: we here cannot bypass nodes incident to time-dependent edges and evaluating travel times is more time-consuming. However, the variant presented here was the first efficient approach to time-dependent routing. Hence, these results are interesting from a historical point of view.

Setup. We interpret the initial values as empty roads and add transit times according to rush hours. Due to the lack of data we increase *all* motorways by a factor of two and all national roads by a factor of 1.5 during rush hours. Our model is inspired by [Flinsenberg 2004]. Our time-dependent implementation assigns 24 different weights to edges, each representing the edge weight at one hour of the day. Between two full hours, we interpolate the real edge weight *linearly*. An easy approach would be to store 24 edge weights separately. As this consumes a lot of memory, we reduce this overhead by storing factors for each hour between 5:00 and 22:00 of the day and the edge weight representing the empty road. Then we compute the travel time of the day by multiplying the initial edge weight with the factor (afterwards, we still have to interpolate). For each factor at the day, we store 7 bits resulting in 128 additional bits for each time-dependent edge. Note that we assume that roads are empty between 23:00 and 4:00.

Time-Dependent Contraction. Another problem for time-dependency is shortcutting time-dependent edges. We avoid this problem by *not* bypassing nodes which

are incident to a time-dependent edge which has the advantage that the space-overhead for additional shortcuts stays small.

Random Queries. Table VIII shows the performance of γ-SHARC for different approximation values. Like in the static scenario we use our default settings. For comparison, the values of time-dependent DIJKSTRA and ALT are also given. As we perform approximative SHARC-queries, we report three types of errors: By *error-rate* we denote the percentage of inaccurate queries. Besides the number of inaccurate queries it is also important to know the quality of a found path. Thus, we report the maximum and average *relative* error of all queries, computed by $1 - \mu_s/\mu_D$, where μ_s and μ_D depict the lengths of the paths found by SHARC and plain DIJKSTRA, respectively.

Table VIII. Performance of the time-dependent versions of DIJKSTRA, ALT, and SHARC on the Western European road network with time-dependent edge weights. For ALT, we use 16 *avoid* landmarks.

		ERROR			PREPRO		QUERY	
	γ	rate	rel. avg.	rel. max	[h:m]	[B/n]	#settled	[ms]
Dijkstra	-	0.0%	0.000%	0.00%	0:00	0	9 016 965	8 890.1
SHARC	1.000	61.5%	0.242%	15.90%	2:51	13	9 804	3.8
	1.005	39.9%	0.096%	15.90%	2:53	13	113 993	61.2
	1.010	32.9%	0.046%	15.90%	2:51	13	221 074	131.3
	1.020	29.5%	0.024%	14.37%	2:50	13	285 971	182.7
	1.050	27.4%	0.013%	2.19%	2:51	13	312 593	210.9
	1.100	26.5%	0.009%	0.56%	2:52	12	321 501	220.8

We observe that using γ values higher than 1.0 drastically reduces query performance. While error-rates are quite high for low γ values, the relative error is still quite low. Thus, the quality of the computed paths is good, although in the worst-case the found path is 15.9% longer than the shortest. However, by increasing γ we are able to reduce the error-rate and the relative error significantely: The error-rate drops below 27%, the average error is below 0.01%, and in worst case the found path is only 0.56% longer than optimal. Generally speaking, SHARC routing allows a trade-off between quality and performance. Allowing moderate errors, we are able to perform queries 2 000 times faster than plain DIJKSTRA, while queries are still 40 times faster when allowing only very small errors.

Comparison. Comparing the figures from Tab. VIII to the values published for the exact variant [Delling 2008], we notice that approximate SHARC is clearly inferior to the new exact variant. Only for $\gamma = 1.000$, queries are faster but for the price of correctness.

6. CONCLUSION

In this work, we introduced SHARC-Routing which combines several ideas from Highway Hierarchies, Arc-Flags, and the REAL-algorithm. More precisely, our approach can be interpreted as a unidirectional hierarchical approach: SHARC steps up the hierarchy at the beginning of the query, runs a strongly *goal-directed* query on the highest level and *automatically* steps down the hierarchy as soon as the search is approaching the target cell. As a result we are able to perform queries as

fast as bidirectional approaches but SHARC can be used in scenarios where former techniques fail due to their bidirectional nature. Due to the unidirectional nature of SHARC, this technique was a promising starting point for the development of an *exact* time-dependent speed-up technique [Delling 2008]. Besides [Batz et al. 2009], time-dependent SHARC is currently the best speed-up technique for time-dependent route planning.

Regarding future work, it would be interesting to compute *reach* values [Gutman 2004] with SHARC. In [Goldberg et al. 2007], an algorithm is introduced for computing exact reach values which is based on partitioning the graph. As our pruning rule would also hold for reach values, we are optimistic that we can compute *exact* reach values for our output graph with our SHARC preprocessing. SHARC-Routing itself also leaves room for improvement. The pruning rule could be enhanced in such a way that we can prune all cells. Moreover, it would be interesting to find better additional shortcuts. Another interesting question arising is whether we can adapt the contraction routine from [Geisberger et al. 2008] to SHARC. And finally, finding partitions optimized for SHARC is an interesting question as well.

APPENDIX

A. PROOF OF CORRECTNESS

We here present a proof of correctness for SHARC-Routing. SHARC directly adapts the query from Arc-Flags, which is proved to be correct. Hence, we only have to show the correctness for all techniques that are used for SHARC-Routing but not for Arc-Flags.

The proof is logically split into two parts. First, we prove the correctness of the preprocessing without the refinement phase. Afterwards, we show that the refinement phase is correct as well.

A.1 Initialization and Main Phase

We denote by G_i the graph after iteration step i, $i = 1, \ldots, L-1$. By G_0 we denote the graph directly before iteration step 1 starts. The level $l(u)$ of a node u is defined to be the integer i such that u is contained in G_{i-1} but not in G_i. We further define the level of a node contained in G_{L-1} to be L.

The correctness of the multi-level arc-flag approach is known. The correctness of handling 1-shell nodes is due to the fact that a shortest path starting from or ending at a 1-shell node u is either completely included in the attached tree T in which also u is contained, or has to leave or enter T via the corresponding core-node.

We want to stress that, when computing arc-flags, shortest paths do not have to be unique. Recall how SHARC handles that: In each level $l < L-1$ all shortest paths are considered, i.e., a shortest path directed acyclic graph is grown instead of a shortest paths tree and a flag for a cell C and an edge (u, v) is set true, if at least one shortest path to C containing (u, v) exists. In level $L-1$, all hop minimal shortest paths are considered, i.e., a flag for a cell C and an edge (u, v) is set true, if at least one shortest path to C containing (u, v) exists that is hop minimal among all shortest paths with same source and target. We observe that the distances between two arbitrary nodes u and v are the same in the graph G_0 and $\bigcup_{k=0}^{i} G_k$ for any $i = 1, \ldots, L-1$.

Hence, in order to proof the correctness of unidirectional SHARC-Routing without the refinement phase and without 1-shell nodes we additionally have to proof the following lemma:

LEMMA A.1. *Given arbitrary nodes s and t in G_0, for which there is a path from s to t in G_0. At each step i of the SHARC-preprocessing there exists a shortest s-t-path $P = (v_1, \ldots, v_{j_1}; u_1, \ldots, u_{j_2}; w_1, \ldots, w_{j_3})$, $j_1, j_2, j_3 \in \mathbb{N}_0$, in $\bigcup_{k=0}^{i} G_k$, such that*

—*the nodes $v_1, \ldots, v_{j_1}$ and $w_1, \ldots, w_{j_3}$ have level of at most i,*

—*the nodes $u_1, \ldots, u_{j_2}$ have level of at least $i+1$*

—*u_{j_2} and t are in the same cell at level i*

—*for each edge e of P, the arc-flags assigned to e until step i allow the path P to t.*

We use the convention that $j_k = 0$, $k \in \{1,2,3\}$ means that the according subpath is void.

The lemma guarantees that, at each iteration step, arc-flags are set properly. The correctness of the bidirectional variant follows from the observation that a hop-minimal shortest path on a graph is also a hop-minimal shortest path on the transpose graph.

PROOF. We show the claim by induction on the iteration steps. The claim holds trivially for $i = 0$. The inductive step works as follows: Assume the claim holds for step i. Given arbitrary nodes s and t, for which there is a path from s to t in G_0. We denote by $P = (v_1, \ldots, v_{j_1}; u_1, \ldots, u_{j_2}; w_1, \ldots, w_{j_3})$ the s-t-path according to the lemma for step i.

The iteration step $i+1$ consists of the contraction phase, the insertion of boundary shortcuts in case $i + 1 = L - 1$, the arc-flag computation and the pruning phase. We consider the phases one after another:

After the Contraction Phase. 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 subpaths $(u_{\ell_f}, u_{\ell_f+1}, \ldots u_{\ell_{f+1}})$ have 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 $\ell_d \neq k$.

By the construction of the contraction routine we know

—$(u_{\ell_1}, u_{\ell_2}, \ldots, u_{\ell_d})$ is also a shortest path

—u_{ℓ_d} is in the same component 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 assigned. Then the arc-flags are correct because of the inductive hypothesis. Otherwise, We know that the nodes $u_1, \ldots, u_{\ell_1 - 1}$ are in the component. Hence, 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 assigned, then arc-flags are correct because of the inductive hypothesis. Otherwise, by cell-aware contraction we know that $u_{\ell_d+1}, \ldots, u_k$ are in the same component as t for all levels at least i. As the own-cell flag always is set true for deleted edges the path stays valid.

As distances do not change during preprocessing we know that, for arbitrary i, $0 \leq i \leq L-1$ a shortest path in G_i is also a shortest path in $\bigcup_{k=0}^{L-1} G_k$. Concluding, the path $\hat{P} = (v_1, \ldots, v_{j_1}, u_1, \ldots, u_{\ell_1-1}; u_{\ell_1}, u_{\ell_2}, \ldots, u_{\ell_d}; u_{\ell_d+1}, \ldots, u_k, w_1, \ldots, w_{j_3})$ fullfills all claims of the lemma for iteration step $i+1$.

After Insertion of Boundary Shortcuts. Here, the claim holds trivially.

After Arc-Flags Computation. Here, the claim also holds trivially.

After Pruning. We consider the path $\hat{P}$ obtained from the contraction step. Let (u_{l_r}, u_{l_r+1}) be an edge of $\hat{P}$ deleted in the pruning step, for which u_{l_r} is not in the same cell as u_{l_d} at level $i+1$. As there exists a shortest path to u_{l_d} not only the own-cell flag of (u_{l_r}, u_{l_r+1}) is set, which is a contradiction to the assumption that (u_{l_r}, u_{l_r+1}) has been deleted in the pruning step.

Furthermore, let (u_{l_z}, u_{l_z+1}) be an edge of P deleted in the pruning step. Then, all edges on P after (u_{l_z}, u_{l_z+1}) are also deleted in that step. Summarizing, if no edge on $\hat{P}$ is deleted in the pruning step, then $\hat{P}$ fullfills all claims of the lemma for iteration step $i+1$. Otherwise, the path $(v_1, \ldots, v_{j_1}, u_1, \ldots, u_{\ell_1-1}; u_{\ell_1}, u_{\ell_2}, \ldots; u_{l_k}, \ldots, u_{\ell_d}, u_{\ell_d+1}, \ldots, u_k, w_1, \ldots, w_{j_3})$ fullfills all claims of the lemma for iteration step $i+1$ where u_{l_k}, u_{l_k+1} is the first edge on P that has been deleted in the pruning step.

Summarizing, Lemma A.1 holds during all phases of all iteration steps of SHARC-preprocessing. So, the preprocessing algorithm (without the refinement phase) is correct. □

A.2 Refinement Phase

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 s.

A path P from s 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 A.1 we know that there must be an (arc-flag valid) shortest s-t-path P 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. We consider the first level $> l$ node u_1 on P. This must be an exit node of s. The node u_2 after u_1 on P is covered and therefore no exit node. Furthermore, it is of level $> l$. Hence, the flags of the edge (u_1, u_2) are propagated to the first edge on P and the claim holds which proves that the refinement phase is correct. Together with Lemma A.1 and the correctness of the multi-level Arc-Flags query, SHARC-Routing is correct.

Acknowledgments. We would like to thank Peter Sanders and Dominik Schultes for interesting discussions on contraction and Arc-Flags. We also thank Daniel Karch for implementing Arc-Flags. Finally, we thank Moritz Hilger for running a preliminary experiment with his new centralized approach.

REFERENCES

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

Bauer, R. and Delling, D. 2008. SHARC: Fast and Robust Unidirectional Routing. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, I. Munro and D. Wagner, Eds. SIAM, 13–26.

Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., and Wagner, D. 2008. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, C. C. McGeoch, Ed. Lecture Notes in Computer Science, vol. 5038. Springer, 303–318.

Bauer, R., Delling, D., and Wagner, D. 2007a. Experimental Study on Speed-Up Techniques for Timetable Information Systems. In *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, C. Liebchen, R. K. Ahuja, and J. A. Mesa, Eds. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 209–225.

Bauer, R., Delling, D., and Wagner, D. 2007b. Shortest-Path Indices: Establishing a Methodology for Shortest-Path Problems. Tech. Rep. 2007-14, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH).

Brandes, U. 2001. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology 25,* 2, 163–177.

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

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

Delling, D. 2009. Time-Dependent SHARC-Routing. *Algorithmica*. Special section devoted to selected best papers of ESA'08. to appear.

Delling, D., Sanders, P., Schultes, D., and Wagner, D. 2009a. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*, J. Lerner, D. Wagner, and K. A. Zweig, Eds. Lecture Notes in Computer Science. Springer. To appear.

Delling, D., Sanders, P., Schultes, D., and Wagner, D. 2009b. Highway Hierarchies Star. In *Shortest Paths: Ninth DIMACS Implementation Challenge*, C. Demetrescu, A. V. Goldberg, and D. S. Johnson, Eds. DIMACS Book. American Mathematical Society. Accepted for publication, to appear.

Demetrescu, C., Goldberg, A. V., and Johnson, D. S., Eds. 2006. *9th DIMACS Implementation Challenge - Shortest Paths*.

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

Flinsenberg, I. C. 2004. Route Planning Algorithms for Car Navigation. Ph.D. thesis, Technische Universiteit Eindhoven.

Geisberger, R., Sanders, P., Schultes, D., and Delling, D. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, C. C. McGeoch, Ed. Lecture Notes in Computer Science, vol. 5038. Springer, 319–333.

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

Goldberg, A. V., Kaplan, H., and Werneck, R. F. 2007. Better Landmarks Within Reach. In *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, C. Demetrescu, Ed. Lecture Notes in Computer Science, vol. 4525. Springer, 38–51.

Gutman, R. J. 2004. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*. SIAM, 100–111.

Hilger, M. 2007. Accelerating Point-to-Point Shortest Path Computations in Large Scale Networks. M.S. thesis, Technische Universität Berlin.

Hilger, M., Köhler, E., Möhring, R. H., and Schilling, H. 2006. Fast Point-to-Point Shortest Path Computations with Arc-Flags. See Demetrescu et al. [2006].

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

Karypis, G. 2007. METIS - Family of Multilevel Partitioning Algorithms.

Karypis, G. and Kumar, V. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing 20,* 1, 359–392.

Köhler, E., Möhring, R. H., and Schilling, H. 2005. 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. Springer, 126–138.

Lauther, U. 2004. 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*. Vol. 22. IfGI prints, 219–230.

Lauther, U. 2009. An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Roadnetworks with Precalculated Edge-Flags. In *Shortest Paths: Ninth DIMACS Implementation Challenge*, C. Demetrescu, A. V. Goldberg, and D. S. Johnson, Eds. DIMACS Book. American Mathematical Society. To appear.

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

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

Monien, B. and Schamberger, S. 2004. Graph Partitioning with the Party Library: HelpfulSets in Practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*. IEEE Computer Society, 198–205.

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

Pyrga, E., Schulz, F., Wagner, D., and Zaroliagis, C. 2007. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics 12*, Article 2.4.

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

Sanders, P. and Schultes, D. 2006. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*. Lecture Notes in Computer Science, vol. 4168. Springer, 804–816.

Schultes, D. 2008. Route Planning in Road Networks. Ph.D. thesis, Universität Karlsruhe (TH), Fakultät für Informatik.

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

Wagner, D., Willhalm, T., and Zaroliagis, C. 2005. Geometric Containers for Efficient Shortest-Path Computation. *ACM Journal of Experimental Algorithmics 10*, 1.3.