

SHARC: Fast and Robust Unidirectional Routing ^{*}

Reinhard Bauer [†]

Daniel Delling[†]

Abstract

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.

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 [10] finds a shortest path between a given source s and target t . Unfortunately, the algorithm is far too slow to be used on huge datasets. Thus, several speed-up techniques have been developed (see [33, 29] for an overview) 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 [9]. The fastest known techniques [5, 1] were developed for road networks and use specific properties of those networks in order to gain their enormous speed-ups.

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 preprocessing—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 infor-

mation 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 [28] to Arc-Flags [21, 22, 23, 18]. 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. By using approximative arc-flags we are able to route very efficiently in *time-dependent* networks, increasing performance by one order of magnitude over previous time-dependent approaches. 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. 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 on a typical search space of uni- and bidirectional SHARC). Only Transit-Node Routing is faster than this variant of SHARC, but SHARC needs considerably less space. A side-effect of SHARC is that preprocessing takes much less time than for pure Arc-Flags.

Related Work. To our best knowledge, three approaches exist that iteratively contract and prune the graph during preprocessing. This idea was introduced in [27]. 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 [28] reducing preprocessing and query times drastically. The RE algorithm, introduced in [14, 15], also uses the contraction from [28] but pruning is based on reach values

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

[†]Universität Karlsruhe (TH), 76128 Karlsruhe, Germany, {rbauer, delling}@ira.uka.de.

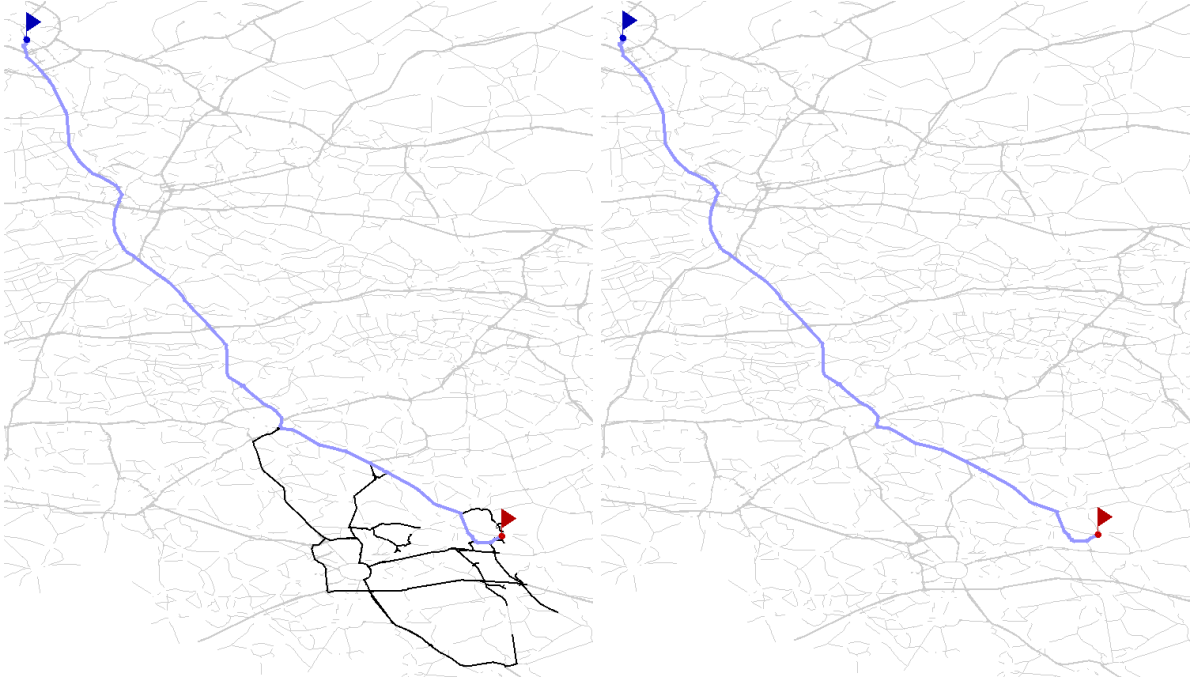


Figure 1: Search space of a typical uni-(left) and bidirectional(right) SHARC-query. The source of the query is the upper flag, the target the lower one. Relaxed edges are drawn in black. The shortest path is drawn thicker. Note that the bidirectional query *only* relaxes shortest-path edges.

for edges. A technique relying on contraction as well is Highway-Node Routing [31], which combines several ideas from other speed-up techniques. 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. Up to now, solely pure ALT [13] has been proven to work in such graphs [7]. Moreover, REAL [14, 15]—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. [26] for details).

Similar to Arc-Flags [21, 22, 23, 18], Geometric Containers [34] 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 *classic* Arc-Flags, see Section 2.

Overview. This paper is organized as follows. Section 2 introduces basic definitions and reviews the classic Arc-Flag approach. Preprocessing and the query al-

gorithm of our SHARC approach are presented in Section 3, while Section 4 shows how SHARC can be used in time-dependent scenarios. 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 $G = (V, E)$ with positive length function $len : E \rightarrow \mathbb{R}^+$. The reverse graph $\bar{G} = (V, \bar{E})$ is the graph obtained from G by substituting each $(u, v) \in E$ by (v, u) . Given a set of edges H , $source(H)$ / $target(H)$ denotes the set of all source / target nodes of edges in H . With $deg_{in}(v)$ / $deg_{out}(v)$ we denote the number of edges whose target / source node 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. A tree on a graph for which exactly the root lies in the 2-core is called an *attached tree*.

A *partition* of V is a family $\mathcal{C} = \{C_0, C_1, \dots, C_k\}$ of sets $C_i \subseteq V$ such that each node $v \in V$ is contained in exactly one set C_i . An element of a partition is

called a *cell*. A *multilevel partition* of V is a family of partitions $\{\mathcal{C}^0, \mathcal{C}^1, \dots, \mathcal{C}^l\}$ such that for each $i < l$ and each $C_n^i \in \mathcal{C}^i$ 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 $(u, v) \in E$. The distance according to len between two nodes u and v we denote by $d(u, v)$.

Classic Arc-Flags. The classic Arc-Flag approach, introduced in [21, 22], 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**. The big advantage of this approach is its easy query algorithm. Furthermore 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. The original approach grows a full shortest path tree from each boundary node yielding preprocessing times of several weeks for instances like the Western European road network. Recently, a new *centralized* approach has been introduced [17]. It grows a centralized tree from each cell keeping the distances to *all* boundary nodes of this cell in memory. This approach allows to preprocess the Western European road network within one day but for the price of high memory consumption during preprocessing.

Note that $AF_{C_i}(e)$ is **true** for almost all edges $e \in C_i$ (we call this flags the *own-cell-flag*). 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 (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 [23] 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 of SHARC is similar to 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_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 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 targeting the core 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 the “fully contained” property by assigning all nodes in an attached tree to the cell of its 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. As shown in [23], the classic 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 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 [25]. For 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 $u \in source(I)$ and for each $v \in target(I) \setminus \{u\}$ we introduce a new edge of the length $len(u, n) + len(n, v)$. If there already is an edge

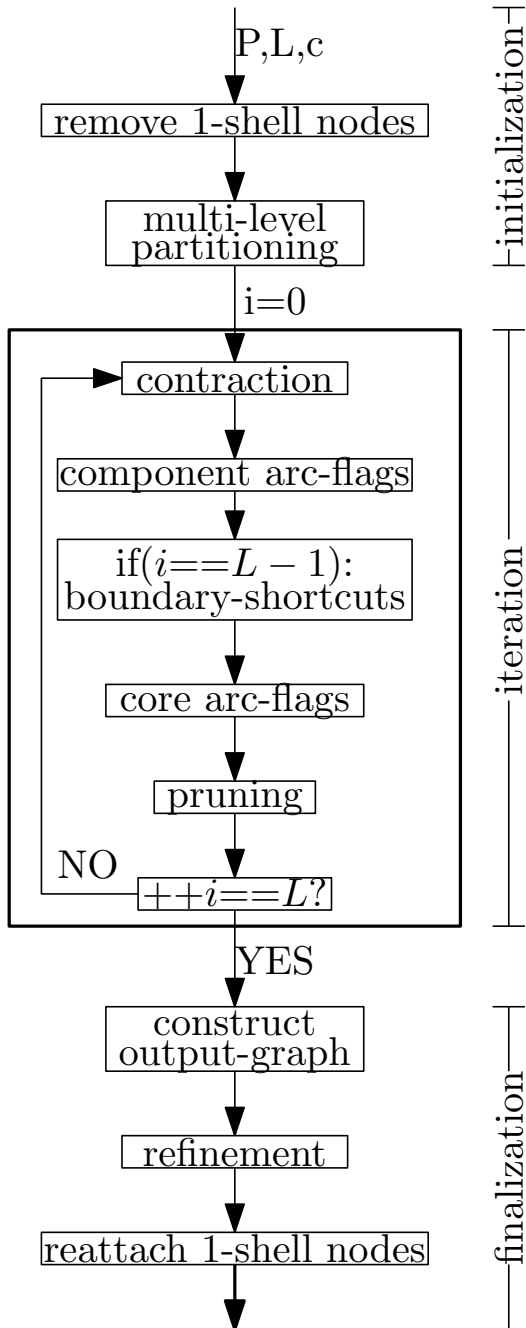


Figure 2: Schematic representation of the preprocessing. Input parameters are the partition parameters P , the number of levels L , and the contraction parameter c . During initialization, we remove the 1-shell nodes and partition the graph. Afterwards, an iterative process starts which contracts the graph, sets arc-flags, and prunes the graph. Moreover, during the last iteration step, boundary shortcuts are added to the graph. During the finalization, we construct the output-graph, refine arc-flags and reattach the 1-shell nodes to the graph.

connecting u and 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 $source(I)$ with nodes in $target(O)$ 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 [28]. The idea to control the order, in which the nodes are bypassed using a heap is due to [14]. In addition, we slightly altered the bypassing criterion, leading to significantly better results, e.g. on the road network of Western Europe, our routine bypasses twice the number of nodes with the same contraction parameter. The main difference to [28] is that we do not count existing edges for determining $\#shortcut$. Finally, the idea to bound the hop number of a shortcut is due to [6].

3.1.4 Boundary-Shortcuts. During our study, we observed that—at least for long-range queries on road networks—a classic bidirected 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 classic 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 [4] values c_B of the boundary nodes on the remaining core-graph. Each boundary node with a betweenness value higher than half the maximum gets $3 \cdot \sqrt{|B_C|}$ additional outgoing edges. The targets are those boundary nodes with highest $c_B \cdot h$ values, where h is the number of hops of the added shortcut.

3.1.5 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 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. Finally, shortcuts get their own-

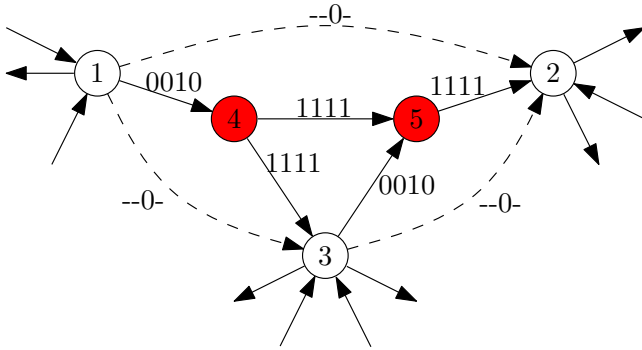


Figure 3: Example for assigning arc-flags during contraction for a partition having four cells. All nodes are in cell 3. The red nodes (4 and 5) are removed, the dashed shortcuts are added by the contraction. Arc-flags are indicated by a 1 for **true** and 0 for **false**. The edges directing into the 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 flag fixed to **false**.

cell flag *fixed* to **false** as relaxing shortcuts when the target cell is reached yields no speed-up. See Figure 3 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 [17], we grow, for each cell C , one centralized shortest path tree on the reverse 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 (see [17] for details). 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 $AF_C^i(e) = \mathbf{true}$.

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.

3.1.6 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.

3.1.7 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 in 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 .

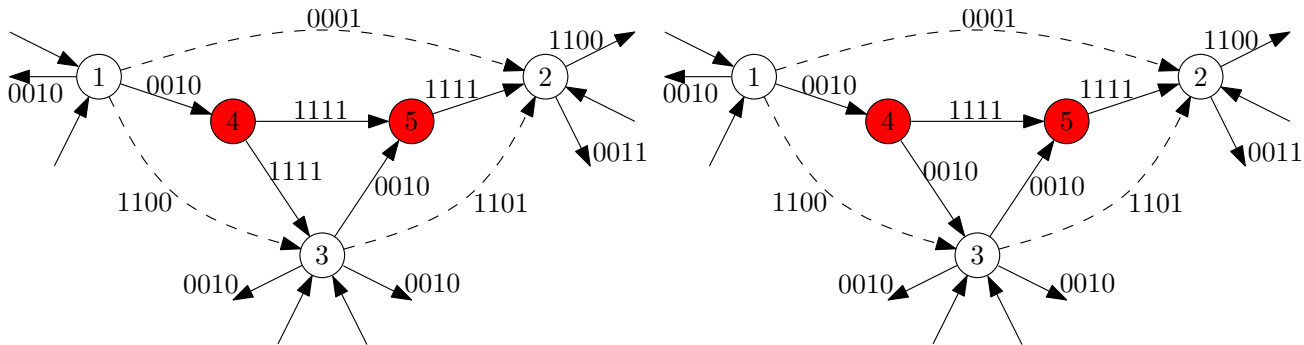


Figure 4: Example for refining the arc-flags of outgoing edges from node 4. The figure in the left shows the graph from Figure 3 after the last iteration step. The figure on the right shows the result of our refinement routine starting at node 4.

First, we build a partial shortest-path tree T starting at u , not relaxing edges that target nodes 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 *entry 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 entry nodes to outgoing edges from u . Starting at an entry node n_E we follow the predecessor in T until we finally end up in a node x whose predecessor is u . The edge (u, x) now inherits the flags from n_E . Every edge outgoing from n_E whose target t is *not* an entry 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 4 gives an example. Note that during refinement step i we only refine arc-flags of edges outgoing from nodes belonging to level i .

3.1.8 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**. Furthermore the multi-level partition and the computed arc-flags are given.

3.2 Query. Basically, our query is a multi-level Arc-Flags DIJKSTRA adapted from the two-level Arc-Flags DIJKSTRA presented in [23]. The query is 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. When relaxing the edges outgoing from n , we consider only those edges having a set arc-flag on level i for the corresponding cell of t . It is proven 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 additionally compute the common level of the current node and the target. 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 [6], as our contraction works similar to Highway Hierarchies.

Multi-Metric Query. In [3], 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 preprocess-

ing, 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.

Optimizations. In order to improve both performance and space efficiency, we use three optimizations. Firstly, we increase *locality* 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 check before running a query, whether the target is in the 1-shell of the graph. If this check holds we do not relax edges that target 1-shell nodes whenever we settle a node being part of the 2-core. Finally, we store each different arc-flag only once in a separate array. We assign an additional pointer to each edge indicating the correct arc-flags. This yields a lower space overhead.

4 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. The fastest known technique for such a scenario is ALT yielding only mild speed-ups of factor 3-5. In this section we present how to perform queries in time-dependent graphs with SHARC. In general, we assume that a time-dependent network $\overline{G} = (V, \overline{E})$ 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 \overline{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 $\overline{AF}_C((u, v)) = \text{true}$ 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.

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.

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 10.1. 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 4.1, 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 forward star implementation. As described in [30], we have to store each edge twice if we want to iterate efficiently over incoming and outgoing edges. Thus, the authors propose to compress edges if target and length of incoming and outgoing edges are equal. However, SHARC allows an even simpler implementation. During preprocessing we only operate on the reverse 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 target, for the query at its source). 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. Tab. 1).

Multi-Level Partition. As already mentioned, the performance of SHARC highly depends on the partition of the graph. Up to now [2], we used METIS [20] for partitioning a given graph. 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 [24] and SCOTCH [25] 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 used 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 3 minutes for all inputs used.

Default Setting. Unless otherwise stated, we use a *unidirectional* variant of SHARC with a 3-level partition with 16 cells per supercell on level 0 and 1 and 96 cells on level 2. Moreover, we use a value of $c = 2.5$ as contraction parameter. When performing random s - t queries, the source s and target t are picked uniformly at random and results are based on 10 000 queries.

Table 1: Performance of SHARC 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 milliseconds. Note that other techniques have been evaluated on slightly different computers. The results for Highway Hierarchies and Highway-Node Routing derive from [30]. Results for Arc-Flags are based on 200 PARTY cells and are taken from [17].

	Europe				USA			
	PREPRO		QUERY		PREPRO		QUERY	
	[h:m]	[B/n]	#settled	[ms]	[h:m]	[B/n]	#settled	[ms]
SHARC	2:17	13	1 114	0.39	1:57	16	1 770	0.68
bidirectional SHARC	3:12	20	145	0.091	2:38	21	350	0.18
Highway Hierarchies	0:19	48	709	0.61	0:17	34	925	0.67
Highway-Node	0:15	8	1 017	0.88	0:16	8	760	0.50
REAL-(64,16)	2:21	32	679	1.10	2:01	43	540	1.05
Arc-Flags	17:08	19	2 369	1.60	10:10	10	8 180	4.30
Grid-based Transit-Node	–	–	–	–	20:00	21	NA	0.063
HH-based Transit-Node	2:44	251	NA	0.006	3:25	244	NA	0.005

5.1 Static Environment. We start 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 [2].

5.1.1 Road Networks. As inputs we use the largest strongly connected component of the road networks of Western Europe, provided by PTV AG for scientific use, and of the US which is taken from the DIMACS homepage [9]. The former graph has approximately 18 million nodes and 42.6 million edges and edge lengths correspond to travel times. The corresponding figures for the USA are 23.9 million and 58.3 million, respectively.

Random Queries. Tab. 1 reports the results of SHARC with default settings compared to the most prominent speed-up techniques. In addition, we report the results of a variant of 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. On the one hand, this is due to the bigger size of the input yielding bigger cell sizes and on the other hand, 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. However, preprocessing does not take twice the time than for default SHARC as we use a 2-level

setup for the bidirectional variant and preprocessing the third level for default SHARC is quite expensive (around 40% of the total preprocessing time). Comparing query performance, bidirectional SHARC is clearly superior to the unidirectional variant. This is due to the known disadvantages of uni-directional classic Arc-Flags: the coning effect and no arc-flag information as soon as the search enters the target cell (cf. Section 2 for details).

Comparing SHARC with other techniques, we observe that SHARC can compete with any other technique except HH-based Transit Node Routing, which requires much more space than SHARC. Stunningly, for Europe, SHARC settles more nodes than Highway Node Routing 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-Node Routing. The bidirectional variant uses more space and has longer preprocessing times, but the performance of the query is very good. The number of nodes settled is smaller than for any other technique and due to the low computational overhead query times are clearly lower than for Highway Hierarchies, Highway-Node Routing or REAL. Compared to the classic Arc-Flags, SHARC significantly reduces preprocessing time and query performance is better.

Local Queries. Figure 5 reports the query times of uni- 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 inserted in the priority queue before v is reached. Thus, it is a kind of distance measure. As input we again use the European road network instance.

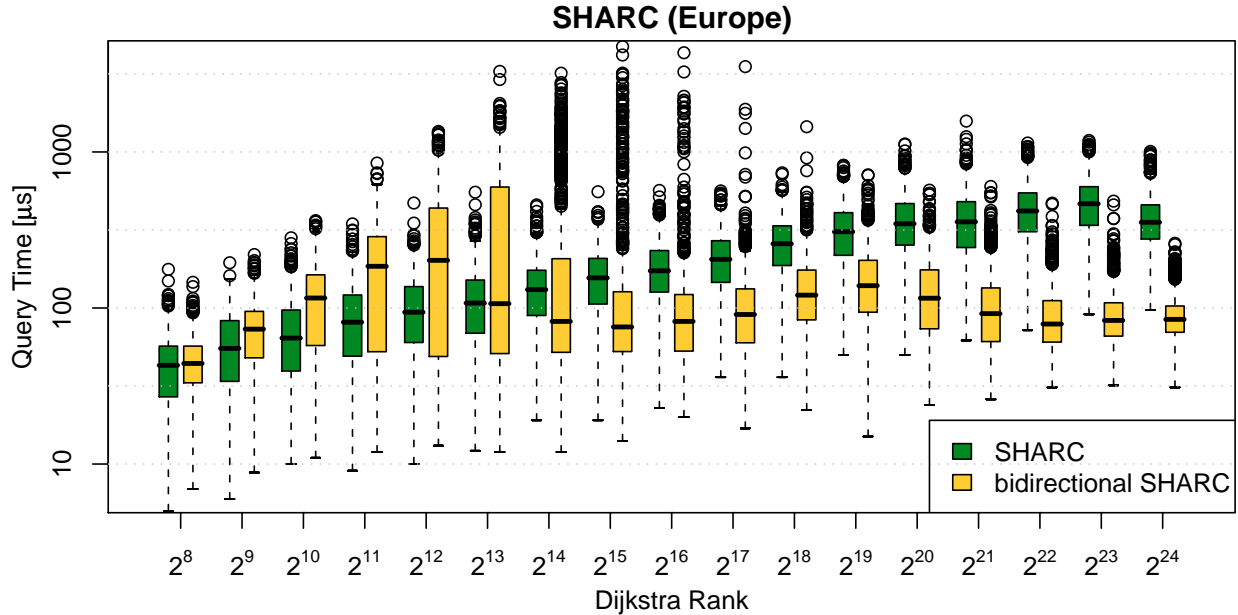


Figure 5: Comparison of uni- and bidirectional SHARC using the Dijkstra rank methodology [27]. The results are represented as box-and-whisker plot [32]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

Note that we use a logarithmic scale due to outliers. Unidirectional SHARC gets slower with increasing rank but the median stays below 0.6 ms while for bidirectional SHARC the median of the queries stays below 0.2 ms. However, for the latter, 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. Interestingly, 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 3 ms.

Table 2: 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.

profile	metric	PREPRO		QUERY	
		[h:m]	[B/n]	#settled	[ms]
linear	single	2:17	13	1 114	0.39
	multi	6:51	16	1 392	0.51
slow car	single	1:56	14	1 146	0.41
	multi	6:51	16	1 372	0.50
fast car	single	2:24	13	1 063	0.37
	multi	6:51	16	1 348	0.49

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. Tab. 2 gives an overview of the performance of SHARC when applied to metrics representing typical average speeds of slow/fast 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. Stunningly, 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.

5.1.2 Timetable Information Networks. Unlike 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. [26] for details). In such networks timetable information can be obtained by running a shortest path query. However, in

Table 3: 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.

graph	tech.	PREPRO		QUERY	
		[h:m]	[B/n]	#sett	[ms]
rail	Dijkstra	0:00	0	1 299 830	406.2
local	SHARC	10:02	9	11 006	3.8
rail	Dijkstra	0:00	0	609 352	221.2
long	SHARC	3:29	15	7 519	2.2
udg	Dijkstra	0:00	0	487 818	257.3
deg.5	SHARC	0:01	16	568	0.3
udg	Dijkstra	0:00	0	521 874	330.1
deg.7	SHARC	0:10	42	1 835	1.0
grid	Dijkstra	0:00	0	125 675	36.7
2-dim	SHARC	0:32	60	1 089	0.4
grid	Dijkstra	0:00	0	125 398	78.6
3-dim	SHARC	1:02	97	5 839	1.9

both models a backward search is prohibited as the time of arrival is unknown in advance. Tab. 3 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). For comparison, we also report results for plain DIJKSTRA.

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 [2]) SHARC (clearly) outperforms any other technique when applied to timetable information system.

5.1.3 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 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 (see e.g. [19]). 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 Tab. 3.

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.

5.2 Time-Dependency. Our final testset is performed on a time-dependent variant of the European road network instance. 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 [11]. 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.

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 stay small. Tab. 4 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.

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

Table 4: 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 [16].

	γ	ERROR			PREPRO		QUERY	
		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
ALT	-	0.0%	0.000%	0.00%	0:16	128	2 763 861	2 270.7
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

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 significantly: 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.

Comparing SHARC (with $\gamma = 1.1$) and ALT, we observe that SHARC queries are one order of magnitude faster but for the price of correctness. In addition, the overhead is much smaller than for ALT. Note that we do not have to store time-dependent edge weights for shortcuts due to our weaker bypassing criterion. Summarizing, SHARC allows to perform fast queries in time-dependent networks with moderate error-rates and small average relative errors.

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. Moreover, a bidirectional variant of SHARC clearly outperforms existing techniques except Transit Node Routing which needs much more space than SHARC.

Regarding future work, we are very optimistic that SHARC is very helpful when running multi-criteria queries due to the performance in multi-metric scenarios. In [15], 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. For the time-dependent scenario one could think of other ways to determine good approximation values. Moreover, it would be interesting how to perform *correct* time-dependent SHARC queries.

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, maybe by adapting the algorithms from [12] to approximate betweenness better. Another interesting question arising is whether we can further improve the contraction routine. And finally, finding partitions optimized for SHARC is an interesting question as well.

Summarizing, SHARC-Routing is a powerful, easy, fast and robust *unidirectional* technique for performing shortest-path queries in large networks.

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 classic Arc-Flags. Finally, we thank Moritz Hilger for running a preliminary experiment with his new centralized approach.

References

- [1] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.

- [2] R. Bauer, D. Delling, and D. Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. In C. Liebchen, R. K. Ahuja, and J. A. Mesa, editors, *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*. Schloss Dagstuhl, Germany, 2007.
- [3] R. Bauer, D. Delling, and D. Wagner. Shortest-Path Indices: Establishing a Methodology for Shortest-Path Problems. Technical Report 2007-14, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), 2007.
- [4] U. Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.
- [5] D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner. High-Performance Multi-Level Graphs. In Demetrescu et al. [9].
- [6] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway Hierarchies Star. In Demetrescu et al. [9].
- [7] D. Delling and D. Wagner. Landmark-Based Routing in Dynamic Graphs. In Demetrescu [8], pages 52–65.
- [8] C. Demetrescu, editor. *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*. Springer, June 2007.
- [9] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.
- [10] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [11] I. C. Flinzenberg. *Route Planning Algorithms for Car Navigation*. PhD thesis, Technische Universiteit Eindhoven, 2004.
- [12] R. Geisberger, P. Sanders, and D. Schultes. Better Approximation of Betweenness Centrality. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*. SIAM, 2008. to appear.
- [13] A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165, 2005.
- [14] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*, pages 129–143. SIAM, 2006.
- [15] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Better Landmarks Within Reach. In Demetrescu [8], pages 38–51.
- [16] A. V. Goldberg and R. F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.
- [17] M. Hilger. Accelerating Point-to-Point Shortest Path Computations in Large Scale Networks. Master's thesis, Technische Universität Berlin, 2007.
- [18] M. Hilger, E. Köhler, R. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Demetrescu et al. [9].
- [19] F. Kuhn, R. Wattenhofer, and A. Zollinger. Worst-Case Optimal and Average-Case Efficient Geometric Ad-Hoc Routing. In *Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC'03)*, 2003.
- [20] K. Lab. METIS - Family of Multilevel Partitioning Algorithms, 2007.
- [21] U. Lauther. Slow Preprocessing of Graphs for Extremely Fast Shortest Path Calculations, 1997. Lecture at the Workshop on Computational Integer Programming at ZIB.
- [22] U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. volume 22, pages 219–230. IfGI prints, 2004.
- [23] R. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11:2.8, 2006.
- [24] B. Monien and S. Schamberger. Graph Partitioning with the Party Library: Helpful-Sets in Practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 198–205. IEEE Computer Society, 2004.
- [25] F. Pellegrini. SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package, 2007.
- [26] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12:Article 2.4, 2007.
- [27] P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.
- [28] P. Sanders and D. Schultes. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.
- [29] P. Sanders and D. Schultes. Engineering Fast Route Planning Algorithms. In Demetrescu [8], pages 23–36.
- [30] P. Sanders and D. Schultes. Engineering Highway Hierarchies. submitted for publication, preliminary version at <http://algo2.iti.uka.de/schultes/hwy/>, 2007.
- [31] D. Schultes and P. Sanders. Dynamic Highway-Node Routing. In Demetrescu [8], pages 66–79.
- [32] R. D. Team. R: A Language and Environment for Statistical Computing, 2004.
- [33] D. Wagner and T. Willhalm. Speed-Up Techniques

for Shortest-Path Computations. In *Proceedings of the 24th International Symposium on Theoretical Aspects of Computer Science (STACS'07)*, Lecture Notes in Computer Science, pages 23–36. Springer, February 2007.

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

A Proof of Correctness

We here present a proof of correctness for SHARC-Routing. SHARC directly adapts the query from classic 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 classic 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 , $i = 1, \dots, L - 1$. 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 the handling of the 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. We remember 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 shortest paths are considered, that are hop minimal for given source and target, 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 $i = 1, \dots, L - 1$.

Hence, 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 $P = (v_1, \dots, v_{j_1}; u_1, \dots, u_{j_2}; w_1, \dots, w_{j_3})$, $j_1, j_2, j_3 \in \mathbb{N}_0$, in $\bigcup_{k=0}^i G_k$, such that*

- *the nodes v_1, \dots, v_{j_1} and w_1, \dots, w_{j_3} have level of at most i ,*
- *the nodes u_1, \dots, 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 reverse 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 $P = (v_1, \dots, v_{j_1}; u_1, \dots, u_{j_2}; w_1, \dots, w_{j_3})$ 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 $(u_{\ell_1}, u_{\ell_2}, \dots, u_{\ell_d})$ with $1 \leq \ell_1, \leq \dots \leq \ell_d \leq k$ for which

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

By the construction of the contraction routine we know

- $(u_{\ell_1}, u_{\ell_2}, \dots, u_{\ell_d})$ 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 $(u_1, \dots, u_{\ell_1-1})$ 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, \dots, u_{ℓ_1-1} are in the component. Hence, all arc-flags for all higher levels are assigned **true**.
- the deleted edges in $(u_{\ell_d+1}, \dots, u_k)$ 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_{ℓ_d+1}, \dots, 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 $\hat{P} = (v_1, \dots, v_{j_1}, u_1, \dots, u_{\ell_1-1}; u_{\ell_1}, u_{\ell_2}, \dots, u_{\ell_d}; u_{\ell_d+1}, \dots, u_k, w_1, \dots, w_{j_3})$ 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 $(v_1, \dots, v_{j_1}, u_1, \dots, u_{\ell_1-1}; u_{\ell_1}, u_{\ell_2}, \dots; u_{l_k}, \dots, u_{\ell_d}, u_{\ell_d+1}, \dots, u_k, w_1, \dots, w_{j_3})$ 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. \square

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 entry node of s . The node u_2 after u_1 on P is covered and therefore no entry 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.