Space-Efficient SHARC-Routing*

Edith Brunel[1], Daniel Delling[1,2], Andreas Gemsa[1], and Dorothea Wagner[1]

[1] Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany. edith@brunel-online.de, {gemsa,dorothea.wagner}@kit.edu
[2] Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043. dadellin@microsoft.com

Abstract. Accelerating the computation of quickest paths in road networks has been undergoing a rapid development during the last years. The breakthrough idea for handling road networks with tens of millions of nodes was the concept of *shortcuts*, i.e., additional arcs that represent long paths in the input. Very recently, this concept has been transferred to *time-dependent* road networks where travel times on arcs are given by functions. Unfortunately, the concept of shortcuts is very space-consuming in time-dependent road networks since the travel time functions assigned to the shortcuts may become quite complex.
In this work, we present how the space overhead induced by time-dependent SHARC, a technique relying on shortcuts as well, can be reduced significantly. We are able to reduce the overhead stemming from SHARC by a factor of up to 11.5 for the price of a loss in query performance of a factor of 4. The methods we present allow a trade-off between space consumption and query performance.

1 Introduction

Route Planning is a prime example of algorithm engineering. Modeling the network as graph G with arc weights depicting travel times, the shortest path in G equals the quickest connection in the transportation network. In general, DIJKSTRA's algorithm [12] solves this task, but unfortunately, the algorithm is way to slow to be used in transportation networks with tens of millions of nodes. Therefore, so called *speed-up techniques* split the work into two parts. During an *offline* phase, called preprocessing, additional data is computed that accelerates queries during the *online* phase. The main concept for route planning in road networks was the introduction of so called *shortcuts*, i.e., arcs representing long paths, to the graph. A speed-up technique then relaxes the shortcut instead of the whole path if the target is "sufficiently far away".

However, adapting the concept of shortcuts to *time-dependent* road networks yields several problems. A travel time function assigned to the shortcut is as complex as all arc functions the shortcut represents. The reason for this is that we need to *link* piecewise linear functions (cf. [8] for details). For example, a straightforward adaption of Contraction Hierarchies [13], a technique relying solely on shortcuts yields an overhead of $\approx 1\,000$ bytes per node [1] in a time-dependent scenario whereas the overhead in a time-independent scenario is almost negligible.

* Partially supported by the DFG (project WA 654/16-1).

[9] gives an overview over time-independent speed-up techniques, while [11] summarizes the recent work on time-dependent speed-up techniques. As already mentioned, all efficient speed-up techniques for road networks rely on adding shortcuts to the graph making the usage of them in a limited time-dependent environment complicated. Memory efficient variants of *time-independent* speed-up techniques however exist. For example, Contraction Hierarchies [13] have been implemented on a mobile device [21]. The straightfoward time-dependent variant [1] is very space-consuming. The ALT [14] algorithm, which works in a time-dependent scenario as well [19], has been implemented on an external device [15] as well. However, space consumption of ALT is rather high and performance is clearly inferior to SHARC. Work on the compression of time-independent graph data structures can also be found in [4, 5]. To the best of our knowledge, we were the first who studied the problem of compressing a high-performance time-dependent speed-up technique [7]. However, since the publication of [7], the memory consumption of time-dependent Contraction Hierarchies has been reduced [2]. Still, our approach yields a factor of 1.6 less overhead.

In this work, we present how to compress the preprocessing of SHARC, introduced in [3] and augmented to the time-dependent scenario in [8], without too high of a loss in query performance. The key idea is to identify unimportant parts of the preprocessing and remove them in such a way that correctness of SHARC can still be guaranteed. After settling preliminaries and recalling SHARC in Section 2, we present our main contribution in Section 3. There, we show how to reduce the overhead stemming from arc-flags stored to the graph, by mapping unimportant arc-flag vectors to important ones. The advantage of this approach over other compression schemes such as bloom filters [6] is that we do not need to change the query algorithm of SHARC. Due to this fact, we keep the additional computational effort limited. Moreover, we show that we can remove shortcuts from SHARC, again without changing the query algorithm. Finally, we may even remove the complex travel time functions from the shortcuts by reproducing the length function on-the-fly. In Section 4 we run extensive tests in order to show the feasibility of our compression schemes. It turns out that we can safely remove 40% of the arc-flag information *without* any loss in query performance. Moreover, about 30% of the shortcuts added by SHARC are of limited use as well. So, we may also remove them. Finally, it turns out that by removing the travel time functions from the remaining shortcuts, we can reduce the overall overhead of SHARC significantely. As a result, we are able to reduce the overhead induced by SHARC by a factor of up to 11.5. The resulting memory efficient variant of SHARC yields an overhead of 13.5 (instead of 156) bytes per node combined with average query times of about 3 *ms* (on the German road network with realistic time-dependent traffic), around 500 times faster than DIJKSTRA's algorithm.

2 Preliminaries

A (directed) graph $G = (V,A)$ consists of a finite set V of nodes and a finite set E of arcs. An arc is an ordered pair (u,v) of nodes $u,v \in V$, the node u is called the *tail* of the arc, v the *head*. The number of nodes $|V|$ is denoted by n, the number of arcs by m. Throughout the whole work we restrict ourselves to directed graphs which are

weighted by a piece-wise linear periodic travel time function *len*. A travel time function $len(e)$ is defined by several *interpolation points*, each consisting of a timestamp t and a travel time $w > 0$, depicting the travel time on e at time t. The travel time between two interpolation points is done by *linear interpolation*. The composition of two travel time functions f, g is defined by $f \oplus g := f + (g \circ (f + id))$.

A *partition* of V is a family $\mathscr{C} = \{C_0, C_1, \ldots, C_k\}$ 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 $\{\mathscr{C}^0, \mathscr{C}^1, \ldots, \mathscr{C}^l\}$ such that for each $i < l$ and each $C_n^i \in \mathscr{C}^i$ a cell $C_m^{i+1} \in \mathscr{C}^{i+1}$ 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 original arc-flag approach [17, 16] first computes a partition $\mathscr{C}$ of the graph and then attaches a *label* AF to each arc e. A label contains, for each cell $C_i \in \mathscr{C}$, a flag $AF_{C_i}(e)$ which is `true` if a shortest path to a node in C_i starts with e. A modified DIJKSTRA then only considers those arcs for which the flag of the target node's cell is `true`. Given two arc-flag vectors AF_1, AF_2. The OR arc-flags vector $AF_1 \vee AF_2$ has all arc-flags set to `true` that are `true` in AF_1 or AF_1. The AND of two arc-flags vectors is defined analogously and is denoted by $AF_1 \wedge AF_2$.

Note that more and more arcs have a flag set for the target's cell when approaching the target cell (called the *coning effect*) and finally, all arcs are considered as soon as the search enters the target cell. Hence, [18] introduces a second layer of arc-flags for each cell. Therefore, each cell is again partitioned into several subcells and arc-flags are computed for each. This approach can be extended to a multi-level arc-flags scenario easily. A multi-level arc-flags query then first uses the flags on the topmost level and as soon as the query enters the target's cell on the topmost level, the lower-level arc-flags are used for pruning. In the following we denote by the level of an arc-flag the level of layer it is responsible for.

SHARC [3]. The main disadvantage of a multi-level arc-flags approach is the time-consuming preprocessing [16]. SHARC improves on this by the integrating of contraction, i.e., a routine iteratively removing unimportant nodes and adding shortcuts in order to preserve distances between non-removed nodes. Preprocessing of SHARC is an iterative process: during each iteration step i, we contract the graph and then compute the level i arc-flags. One key observation of SHARC is that we are able to assign arc-flags to all bypassed arcs during contraction. More precisely, any arc (u, v) outgoing from a non-removed node and heading to a removed one gets only one flag set to `true`, namely, for the region v is assigned to. Any other bypassed arc gets all flags set to `true`. By this procedure, unimportant arcs are only relaxed at the beginning and end of a query. Although these suboptimal arc-flags already yield a good query performance, SHARC improves on this by a (very local) arc-flag refinement routine. The key observation here is that bypassed arcs may inherit flags from arcs not bypassed during contraction (cf. [3] for details). It should be noted SHARC integrates contraction in such a natural way that the multi-level arc-flags query can be applied to SHARC without modification.

Due to its unidirectional query algorithm, SHARC was a natural choice for augmenting it to a time-dependent [8] and a multi-criteria scenario [10]. The idea is the same for both augmentations: adapt the basic ingredients of the preprocessing, i.e.,

arc-flags, contraction, and arc-flags refinement, such that correctness of them can still be guaranteed and leave the basic concept untouched. It turns out that SHARC performs pretty well in both augmented scenarios. However, a crucial problem for time-dependent route planning are shortcuts representing paths in the original graph. While this is "cheap" in time-independent networks, the travel time functions assigned to time-dependent shortcuts may become quite complex. In fact, the number of interpolation points defining the shortcut is approximately the sum of all interpolation points assigned to the arcs the shortcut represents. See [8] for details. In fact, the overhead of SHARC increases by a factor of up to 10 when switching from a time-independent to a time-dependent scenario exactly because of these complex travel time functions.

SHARC adds auxiliary data to the graph. More precisely, the overhead stems from several ingredients: Region information, arc-flags, topological information of shortcuts, the travel time functions of shortcuts and shortcut unpacking information. We call the graph enriched by shortcuts and any other auxiliary data the *output graph*.

The first overhead, i.e., the region information, is used for determining which arc-flag to evaluate during query times. This information is encoded by an integer and cannot be compressed without a significant performance penalty. We have a arc-flags vector for each arc. However, the number of unique arc-flags vectors is much smaller than the number of arcs. So, instead of storing the arc-flags directly at each arc, we use a separate table containing all possible unique arc-flags sets. In order to access the flags efficiently, we assign an additional pointer to each arc indexing the correct arc-flags set in the table. The main overhead, however, stems from the shortcuts we add to the graph. For each added shortcut, we need to store the topological information, i.e., the head and tail of the arc, and the travel time function depicting the travel time on the path the shortcut represents. Moreover, we need to store the arcs the shortcut represents in order to retrieve the complete description of a computed path.

3 Preprocessing Compression

In this section, we show how to reduce the space consumption of SHARC by removing unimportant arc-flags, shortcuts, and functions without violating correctness.

Arc-Flags. The first source of overhead for SHARC is storing the arc-flags for each arc. As already mentioned, our original SHARC implementation already compresses the arc-flag information by storing each unique arc-flag set separately in a table (called the *arc-flags table*) and each arc stores an index to the arc-flag table. Figure 1 gives a small example.

Lemma 1. *Given a correct SHARC preprocessing. Flipping (arbitrary) arc-flags from* `false` *to* `true` *does not violate the correctness of SHARC.*

Proof. Let $P = (u_1, \ldots, u_k)$ be an arbitrary shortest path in G. Since SHARC-Routing is correct, we know that each arc (u_i, u_{i+1}) has the arc-flag being responsible for t set to `true`. Since we only flip bits from `false` to `true`, this also holds after bit-flipping. □

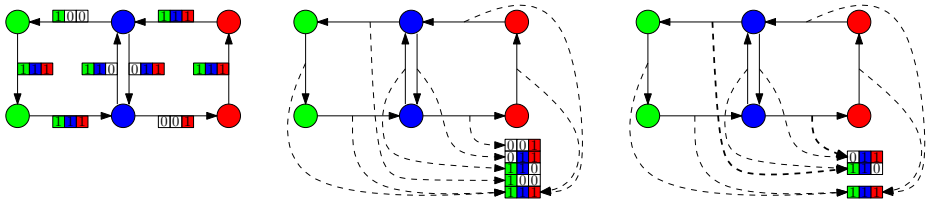

Fig. 1. Compression of Arc-Flags. The input is partitioned into three cells, indicated by coloring. A set arc-flag is indicated by color and a one. Instead of storing the flags directly at the arcs (left figure), we store each unique arc-flags vector in a separate table with arcs indexing the right arc-flags vector (middle figure). We additionally compress the table by flipping bits from `false` to `true` (right figure) and thus, we reduce the number of entries in the table. The remapped indices are drawn thicker.

This lemma allows us to flip bits in the arc-flag table from `false` to `true`. Hence, we compress the arc-flag table by *bit-flipping*. Let AF_r, AF_m be two unique arc-flag vectors such that $AF_r \subseteq AF_m$, i.e., all arc-flags set in AF_r are also set in AF_m. Then, we may remove AF_r from our arc-flag table and all arcs indexing AF_r are remapped to AF_m. Note that this compression scheme has no impact on the query algorithm.

The compression rate achieved by bit-flipping highly depends on which arc-flag vectors to remove and to which arc-flag vectors they are mapped. We introduce an *arc-flag costs* function cost assigning an importance value to each arc-flags vector. The idea is as follows: for each layer i of the multi-level partition, we introduce a value cost_i. Let $|AF_i|$ be the number of flags set to `true` on level i. Then we define $\text{cost}(AF) = \sum_{i=0}^{l} \text{cost}_i \cdot |AF_i|$. The higher the costs of an arc-flags vector scores, the more important it is. So, a good candidate for removing it from the table should have low costs. The remaining question is what a good candidate for mapping is. Therefore, we define the *flipping costs* between two arc-flag vectors AF_r, AF_m with $AF_r \subseteq AF_m$ as $\text{cost}(AF_r \wedge AF_m)$. A good mapping candidate AF_m for a vector AF_r to be removed is the arc-flags vector with minimal flipping costs. It is easy to see that $\text{cost}(AF_r \wedge AF_m) = \text{cost}(AF_m) - \text{cost}(AF_r)$ holds. We reduce the overhead induced by arc-flags by iteratively removing arc-flags vectors from the table. Therefore, we order the unique arc-flags vectors non-decreasing by their costs. Then, we remove the arc-flags vector AF_r with lowest costs from the table and remap all arcs indexing AF_r to the arc-flags vector AF_m with minimal costs and for which $AF_r \subseteq AF_m$ holds.

Shortcuts. In Section 2, we discussed that, at least in the time-dependent scenario, the main source of overhead derives from the (time-dependent) shortcuts added to the graph. In [3], we already presented a subroutine to remove *all* shortcuts from SHARC. However, this yielded a high penalty in query performance. The following lemma recaps the main idea.

Lemma 2. *Given a correct SHARC preprocessing. Let (u,v) be an added shortcut during preprocessing and let $P_{uv} = (u, u_0, \ldots, u_k, v)$ be the path it represents. By removing (u,v) from the graph and setting $AF(u,u_0) = AF(u,u_0) \vee AF(u,v)$, correctness of SHARC is not violated.*

Proof. Let $P = (s, \dots, u, v, \dots, t)$ be an arbitrary shortest path using shortcut (u,v) in the output graph of SHARC. Let also (u,u_0) be the first edge of the path (u,v) represents in the original graph. We need to show that after removing (u,v), the path $P' = (s, \dots, u, u_0, \dots, t)$ has all flags for t set to `true`. Since SHARC is correct, we know that the subpath $(s, \dots, u)$ has correct flags set. Moreover, (u,u_0) has proper flag set as well, since we propagate all flags from (u,v) to (u,u_0). We also know that the shortest path from u_0 to t must not contain (u,v) since we restrict ourselves to positive length functions (cf. Section 2). Due to correctness of SHARC, the shortest path from u_0 to t must have proper flags set. Hence, P' has proper flags set as well. □

In other words, we reroute any shortest path query using a removed shortcut (u,v) to its path it represents, Figure 2 gives an example. The lemma allows us to remove arbitrary shortcuts from the output graph. Note that we may again leave the SHARC query untouched. Some shortcuts are more important than others and the ordering in which we remove shortcuts has a high impact on the resulting query performance. Generally, a removed shortcut should only have low arc-flags costs (cf. Section 4). Furthermore, let $l(u)$ be the level of an arbitrary node u, given by iteration u was removed during the original preprocessing of SHARC (cf. Section 2). We define the *tail-level* of a shortcut (u,v) by $l(u)$, while $l(v)$ is the *head-level*. Presumably, shortcuts with low head and tail levels are less important than those with high ones. A fourth indicator for the importance of a shortcut is the so-called *search space coning coefficient*. Let (u,v) be a shortcut and let $P_{uv} = (u, u_0, \dots, u_k, v)$ be the path it represents. Then the search space coning coefficient of (u,v) is given by $\mathrm{sscc}(u,v) = \sum_{u_i \in P_{uv}} \sum_{(u_i,w) \in E, w \neq u_{i+1}} \mathrm{cost}\big(AF(u,v) \wedge AF(u_i,w)\big)$. In other words, the search space coning coefficient depicts how many arcs may be relaxed additionally if (u,v) was removed, i.e., the search cones. Therefore, the arc-flags of any outgoing arc from $u_i \in P_{uv}$ is examined and whenever a flag is set that is also set for the shortcut, the search space coning coefficient increases.

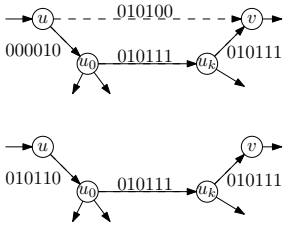

Fig. 2. Example for removing the shortcut (u,v), representing (u,u_0,u_k,v), by propagating flags from (u,v) to (u,u_0).

Our *shortcut-removal* compression scheme iteratively removes shortcuts from the graph and sets arc-flags according to Lemma 2. We use a priority queue to determine which shortcut to remove next. The priority of a shortcut is given by a linear combination of its head level, its tail level, its arc-flag costs, and the search space coning coefficient. We normalize these values by their maximal values during initialization.

Removing Travel Time Functions. Removing shortcuts from the output graph increases the search space since unnecessary arcs may be relaxed during traversing the path the shortcut represents. However, the main problem of time-dependent shortcuts are their complex travel time functions. Another possibility to remedy this space consumption is to remove the travel time functions but to keep the shortcut itself. Now, when a shortcut is relaxed, we compute the weight of it by unpacking the shortcut on-the-fly. The advantage of this over complete removal of the shortcut is that the search space does not increase. However, due to on-the-fly unpacking query times may increase.

Again, like for removing shortcuts and arc-flags compression, we are able to remove the travel time functions only from some of the shortcuts. On the one hand, we want to remove a shortcut with a complex function, and on the other hand, it should not be relaxed too frequentely. Hence, we use a priority queue in order to determine which function to delete next. As key for (u,v) we use a linear combination of the number of interpolation points of $len(u,v)$ and the arc-flag costs of $AF(u,v)$.

4 Experiments

Our experimental evaluation has been done on one core of an AMD Operon 2218 running SUSE Linux 11.1. 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 4.3, using optimization level 3. Our implementation is written in C++. As priority queue we use a binary heap.

We use the German road network as input, it has approximately 4.7 million nodes and 10.8 million arcs. We have access to five different traffic scenarios, generated from realistic traffic simulations: Monday, midweek (Tuesday till Thursday), Friday, Saturday, and Sunday. All data has been provided by PTV AG [20] for scientific use.

For testing our compression schemes, we run a complete time-dependent SHARC preprocessing (heuristic variant [8]) on our Monday instance, we use our default parameters from [8]. The input has a space consumption of 44.2 bytes per node. SHARC adds about 2.7 million shortcuts and increases the number of interpolation points (for time-dependent arcs) from 12.7 million to about 92.1 million, yielding a total overhead of 156.94 additional bytes per node. Form those 156.94 bytes per node, 11.55 are stemming from shortcuts (topology and unpacking information), 8.2 from the arc-flag information, 2.0 from the region information, while 135.31 bytes per node stem from the additional interpolation points added to the graph. Hence, the main overhead stems from the latter. Note that the total overhead is slightly higher than reported in [8]. The reason for this is a change (we now store no interpolation point for a time-independent edge) to a more space-efficient graph data structure. Note that the heuristic variant of SHARC may compute a path that is slightly longer than the shortest in very few occasions [8]. However, all insights gained her also hold for any other variant.

In order to evaluate how well our schemes work, we evaluate the query performance of this SHARC preprocessing after compression. Therefore, we run 100 000 *s*-*t* queries for which we pick s, t, and the departure time uniformly at random. Then, we provide the average query time. Note that we do not report the time for unpacking the whole path. However, this can be done in less than 0.1 ms.

Arc-Flags. Figure 3 depicts the performance of time-dependent SHARC in our Monday scenario after removing a varying amount of arc-flags vectors for different cost functions. Note that we do not provide running times for compression since it takes less than one minute to compress the arc-flags. This is only a small fraction of the time the SHARC preprocessing takes (3-4 hours). We observe that the choice of the cost function has a high impact on the success of our flag compression scheme. As expected, a cost function that prefers flipping of low-level flags (cost function 1,3,9,27,243) performs better than one that prefers high-level flags (cost function 16,8,4,2,1). Interestingly, we

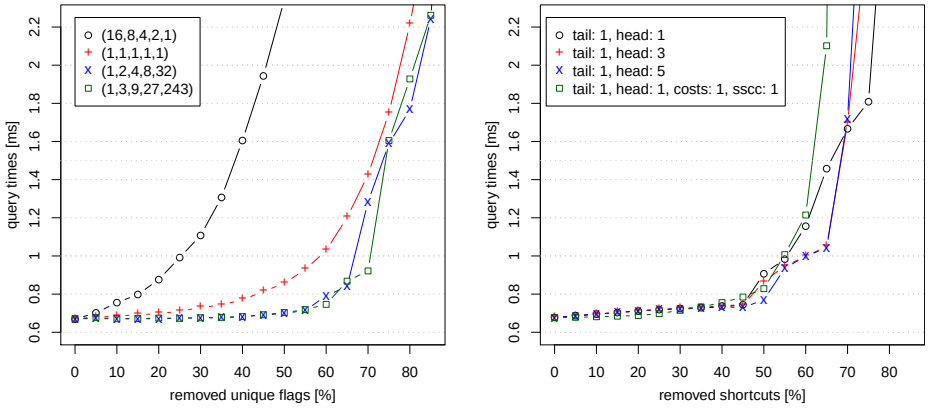

Fig. 3. Removing arc-flags (left) and shortcuts (right) from the output graph with cost functions. For arc-flags, the entry on the left indicates the costs for flipping a low level flag, while the right most entry shows the costs for flipping the highest level.

may remove up to 40% of the arc-flags vectors without any loss in performance. This reduces the overhead induced by arc-flags from 8.2 bytes per node to 7.2. By removing 60% of the vectors (resulting overhead: 6.9), the query performance decreases only by 10%. However, removing more than 70% of the flags yields a significant penalty in performance, although the overhead (for arc-flags) is only reduced to 6.3 bytes per node. So, since arc-flags contribute only a small fraction to the overhead, it seems reasonable to settle for a arc-flag compression rate of 40%.

Shortcuts. Figure 3 depicts the performance of SHARC after removing a varying amount of shortcuts for different linear coefficients introduced in Section 3. Note again that we do not report running times for compression since it takes less than one minute to obtain the reduced preprocessed data. We observe that we can remove up to 45% of the shortcuts yielding a mild increase in query performance ($\approx$ 10%). This reduces the overhead of the shortcuts from 11.55 bytes per node to 7.51. Moreover, the overhead induced by travel time functions is reduced from 135.31 to 118.72 bytes per node since some of the removed shortcuts are time-dependent. Up to 65%, the loss in query performance is still acceptable (a factor of 2), especially when keeping the gain in mind: the overhead for shortcuts reduces to 5.3 bytes per node and 97.7 bytes per node for additional interpolation points. Analyzing the impact of ordering, we observe that the level of head and tail seem to be the most important parameters. Interestingly, the head level seems to be more important than the tail level. The reason for this is that shortcuts from lower to higher levels are relaxed at the beginning of a query, while shortcuts from higher to lower levels are relaxed at the end. Since removing shortcuts cones the query (cf. Section 3), the latter shortcuts are less important. The influence of the search space coning coefficient is minor and only observable for very low compression rates: at 20%, the loss in performance is almost negligible. In the following, we will use values of 20%, 45%, and 65% as default parameters.

Travel Time Functions. Figure 4 indicates the query performance of SHARC after removing travel time functions from time-dependent shortcuts. We here evaluate different orderings given by different weights for the coefficients arc-flag costs and number of interpolation points, as explained in Section 3. We observe that for low compression rates, the arc-flag costs are more important than the number of points on the shortcut. However, the situation is vice versa between compression rates between 60% and 85%: here an ordering based on the number of points performs better than the order based on arc-flag costs. However, the differences are marginal, hence we use arc-flag costs as default for determing the ordering. In general, we may remove up to 40% of the additional points for a loss of query performance of about 25%. This already reduces the overhead induced by additional interpolation points to 81.2 bytes per node. The corresponding figures for a compression rate of 60% are a query performance penalty of factor 2 and a resulting overhead of 63.1. Most remarkably, we may even remove *all* additional interpolation points from the output graph with paying "only" a loss of performance of a factor of 3.2. This yields a *total* overhead of 21.6 bytes per node, a reduction of factor of 7.5 over the uncompressed preprocessing. Still the average query performance of 2.3 ms is still a speedup of a factor of 678 over DIJKSTRA's algorithm.

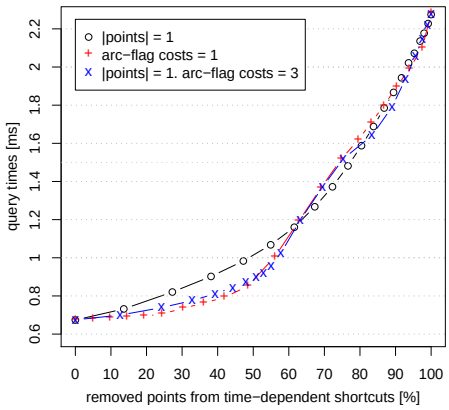

Fig. 4. Removing travel time functions. The x-axis indicates how many of the additional interpolation points are removed by the compression scheme. Coefficients not indicated are set to zero.

Combinations. Up to now, we evaluated each compression scheme separately. Table 1 gives an overview if we combine all three schemes among each other. We here report the overhead of the preprocessed data in terms of *additional* bytes per node. For evaluating the query performance, we not only provide query times but also the average number of settled nodes and relaxed arcs for 100 000 random *s*-*t* queries. Moreover, we report the speed-up over our (efficient) implementation of time-dependent Dijkstra. On this input, the latter settles about 2.2 million arcs in about 1.5 seconds on average. We observe that we may vary the compression rate yielding different total overhead and query performance. A good trade-off seems to be achieved for compressing shortcuts by 20%, interpolation points by 60%, and flags by 40%. This reduces space overhead in total by a factor of 2 and yields a loss in query performance by a factor of 1.85. For this is reason, we call these values a *medium* compression setup. Our *high* compression setup removes 65% of all shortcuts, removes all interpolation points from remaining time-dependent shortcuts and reduces the flag-table by 40%. This reduces the overhead induced by SHARC by a factor of almost 11.5 while query performance is ≈ 4 times slower than without compression. Any compression beyond this point yields a big performance loss without a significant reduction in preprocessed data.

Table 1. Query performance for different combinations of our compression schemes. As input, we use the German road network with traffic scenario Monday.

SHORTCUTS		POINTS		FLAGS		TOTAL		QUERIES			
rem.	overhead [bytes/n]	rem. [%]	overhead [bytes/n]	rem. [%]	overhead [bytes/n]	overhead [bytes/n]	comp. [%]	#sett. nodes	#rel. arcs	time [ms]	speed up
0	11.55	0	135.31	0	8.20	156.94	0.0	775	984	0.68	2 288
20	9.72	40	75.67	40	7.03	94.29	39.9	876	1 095	0.87	1 786
20	**9.72**	**60**	**50.22**	**40**	**7.03**	**68.84**	**56.1**	**876**	**1 095**	**1.26**	**1 238**
20	9.72	100	0.00	40	7.03	18.62	88.1	876	1 095	2.44	636
45	7.51	40	67.19	40	6.67	83.24	47.0	949	1 167	0.94	1 654
45	7.51	60	44.78	40	6.67	60.83	61.2	949	1 167	1.43	1 087
45	7.51	100	0.00	40	6.67	16.05	89.8	949	1 167	2.56	606
65	5.30	40	54.66	40	6.34	68.17	56.6	1 717	1 971	1.39	1 118
65	5.30	60	37.97	40	6.34	51.48	67.2	1 717	1 971	1.88	827
65	**5.30**	**100**	**0.00**	**40**	**6.34**	**13.52**	**91.4**	**1 717**	**1 971**	**2.98**	**521**
65	5.30	100	0.00	60	6.10	13.27	91.5	1 811	2 085	3.07	506

Traffic Scenarios. Our final testset evaluates the impact of different traffic scenarios on our compression schemes. Besides our Monday scenario which we evaluated up to this point, we now also apply a midweek (Tuesday to Thursday), Friday, Saturday, Sunday, and "no traffic" scenario. Note that the latter is a time-independent network. Our graph data structures occupy 44.2, 44.1, 41.0, 31.4, 27.8, and 22.4 bytes per node, respectively. We here also report the *additional* overhead induced by SHARC, as well as the total time of preprocessing (including compression). The resulting figures can be found in Tab. 2.

We observe that for all time-dependent inputs, our medium compression setup reduces the space consumption by a factor of 2 combined with a performance penalty between 1.5 and 1.9, depending on the degree of time-dependency in the network. Our high compression rate yields an overhead of ≈ 13.5 bytes per node, independently of the applied traffic scenario. This even holds for our "no traffic scenario". The query performance however, varies between 0.37 ms (no traffic) and 3.06 ms (midweek). The reason for this is that in a high traffic scenario, more are arcs are time-dependent and hence, more arcs need to be evaluated when unpacking a (time-dependent) shortcut on-the-fly. Since the no traffic input contains no time-dependent arcs, no shortcut has a travel time function assigned. Hence, the costly on-the-fly unpacking needs not to be done during query times.

Comparison. Finally, we compare our memory-efficient version of SHARC (high compression) with the most recent variant of Contraction Hierachies (CH) [2]. The input is Germany midweek. CH achieves a speed-up of 714 over Dijkstra's algorithm, assembling 23 additional bytes per node in 37 minutes. SHARC yields a slightly lower speed-up (491) with less overhead (13.5 bytes per node) for the price of a longer preprocessing time (around 4 hours). However, our heuristic variant of SHARC (which we use in this paper) drops correctness while CH is provably correct. Still, in (very) space-limited time-dependent environment, SHARC seems to be the first choice.

Table 2. Query performance of heuristic time-dependent SHARC applying different traffic scenarios for our German road network. Column *compression rate* indicates our default rates from Section 4. Columns *increase edges, points* indicate the increase in number of edges and points compared to the input.

	PREPROCESSING						QUERIES			
scenario	comp. rate	time [h:m]	inc. edges	inc. points	space [bytes/n]	comp [%]	#sett. nodes	#rel. arcs	time [ms]	speed up
Monday	none	3:52	25.2%	621.1%	156.94	0.0	775	984	0.68	2 288
	med	3:54	19.9%	230.5%	68.84	56.1	876	1 095	1.26	1 238
	high	3:54	8.2%	0.0%	13.52	91.4	1 717	1 971	2.98	521
midweek	none	3:46	25.2%	621.8%	156.45	0.0	777	990	0.68	2 203
	med	3:48	20.0%	229.3%	68.35	56.3	880	1 102	1.28	1 177
	high	3:48	8.2%	0.0%	13.54	91.3	1 715	1 971	3.06	491
Friday	none	3:22	25.1%	654.4%	142.90	0.0	733	930	0.63	2 400
	med	3:24	19.9%	240.4%	63.22	55.8	837	1 043	1.17	1 302
	high	3:24	8.2%	0.0%	13.59	90.5	1 691	1 932	2.79	543
Saturday	none	2:24	24.7%	784.2%	92.37	0.0	624	782	0.49	3 001
	med	2:26	19.6%	286.8%	44.58	51.7	726	892	0.81	1 825
	high	2:26	8.0%	0.0%	13.69	85.2	1 615	1 817	1.95	752
Sunday	none	1:53	24.3%	859.4%	67.43	0.0	593	735	0.44	3 299
	med	1:55	19.2%	317.8%	35.58	47.2	693	844	0.68	2 159
	high	1:55	7.9%	0.0%	13.64	79.8	1 576	1 762	1.64	893
no	none	0:10	23.3%	0.0%	20.90	0.0	277	336	0.18	6 784
	med	0:12	18.4%	0.0%	17.80	14.8	327	390	0.20	5 990
	high	0:12	7.5%	0.0%	13.12	37.2	758	838	0.37	3 332

5 Conclusion

In this work, we showed how to reduce the space consumption of SHARC without too high of a loss in query performance. The key idea is to identify unimportant parts of the preprocessing and remove them in such a way that correctness of SHARC can still be guaranteed. More precisely, we showed how to reduce the overhead stemming from arc-flags stored to the graph, how to remove shortcuts and how to remove complex travel time functions assigned to shortcuts. As a result, we were able to reduce the overhead induced by SHARC by a factor of up to 11. We thereby solved the problem of high space consumption of time-dependent route planning: SHARC does not yield a space-consumption penalty for switching from time-independent to time-dependent route planning, making it an interesting candidate for mobile devices.

Regarding future work, it would be interesting to compress the time-dependent input graphs. Techniques from [4, 5, 21] show how to compress the topology information of a graph. The main challenge, however, seems to be the reduction of the space consumption needed for storing the travel time functions. A possible approach could be the following: Real-world networks often assign a so called delay-profile to each edge. So, instead of storing the travel functions at the edges, one could use an index pointing to the (small number of) delay profiles. Note that this approach is similar to the arc-flags compression scheme.

References

1. G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-Dependent Contraction Hierarchies. In *ALENEX*, pages 97–105. SIAM, April 2009.
2. G. V. Batz, R. Geisberger, S. Neubauer, and P. Sanders. Time-Dependent Contraction Hierarchies and Approximation. In *SEA'10*, *LNCS* 6049. Springer, May 2010.
3. R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14:2.4, May 2009.
4. D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact Representation of Separable Graphs. In *SODA*, pages 679–688. SIAM 2003.
5. D. K. Blandford, G. E. Blelloch, and I. A. Kash. An Experimental Analysis of a Compact Graph Representation. In *ALENEX*, pages 49–61. SIAM, 2004.
6. B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.
7. E. Brunel, D. Delling, A. Gemsa, and D. Wagner. Space-Efficient SHARC-Routing. Technical Report 13, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), 2009.
8. D. Delling. Time-Dependent SHARC-Routing. *Algorithmica*, July 2009.
9. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In *Algo. of Large and Complex Networks*, *LNCS* 5515, pages 117–139. Springer, 2009.
10. D. Delling and D. Wagner. Pareto Paths with SHARC. In *SEA'09*, *LNCS* 5526, pages 125–136. Springer, June 2009.
11. D. Delling and D. Wagner. Time-Dependent Route Planning. In *Robust and Online Large-Scale Optimization*, *LNCS* 5868, pages 207–230. Springer, 2009.
12. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
13. R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *WEA'08*, *LNCS* 5038, pages 319–333. Springer, June 2008.
14. A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *SODA*, pages 156–165, 2005.
15. A. V. Goldberg and R. F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *ALENEX*, pages 26–40. SIAM, 2005.
16. M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 41–72. American Mathematical Society, 2009.
17. U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.
18. R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11:2.8, 2006.
19. G. Nannicini, D. Delling, L. Liberti, and D. Schultes. Bidirectional A* Search for Time-Dependent Fast Paths. In *WEA'08*, *LNCS 5038*, pages 334–346. Springer, June 2008.
20. PTV AG - Planung Transport Verkehr. http://www.ptv.de, 2008.
21. P. Sanders, D. Schultes, and C. Vetter. Mobile Route Planning. In *ESA'08*, *LNCS 5193*, pages 732–743. Springer, September 2008.