Engineering and Augmenting Route Planning Algorithms

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Daniel Delling

aus Hamburg

Tag der mündlichen Prüfung:	10.02.2009
Erste Gutachterin:	Prof. Dr. Dorothea Wagner
Zweiter Gutachter:	Prof. Dr. Rolf Möhring

Acknowledgements

First of all I would like to thank Dorothea Wagner for the possibility to join her great group. During my studies, she greatly supported me in every way one can think of and her advice concerning all kinds of topics were of great help for me. Special thanks also go to the following persons. My office mate Sascha Meinert for the great atmosphere, Robert Görke for the burden of proof reading every single work of mine, including this thesis, Bastian Katz and Ignaz Rutter for helping me with great enthuasim when it came down to code tuning, Lillian Beckert for their perfect organization, Christos Zaroliagis for the opportunity to visit his group for one month, and Panos, Lina, and Apostolos for helping me to survive in Greece.

I enjoyed working on joint papers on route planning and graph clustering with my coauthors Veit Batz, Reinhard Bauer, Ulrik Brandes, Gianlorenzo D'Angelo, Marco Gaertler, Robert Geisberger, Kalliopi Giannakopoulou, Robert Görke, Martin Höfer, Martin Holzer, Leo Liberti, Kirill Müller, Giacomo Nannicini, Zoran Nikoloski, Thomas Pajor, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, Christian Schulz, Frank Schulz, Christian Vetter, Dorothea Wagner, and Christos Zaroliagis. Besides that, I had many interesting discussions with Andrew Goldberg, Riko Jacob, Matthias Müller-Hannemann, Mathias Schnee, and Renato Werneck about route planning in augmented scenarios.

Since there is a life outside university, I want to thank my girlfriend Isolde, my parents, and my grandmother for their love and support during the past three years. Without them, this work would not have been possible. In addition, I want to thank all members of the VSBW for many valuable discussions.

Finally, I thank Rolf Möhring for his willingness to review this thesis and the EU for granting my work.

Abstract

Algorithm engineering exhibited an impressive surge of interest during the last years, spearheaded by one of the showpieces of algorithm engineering: computation of shortest paths. In principle, Dijkstra's classical algorithm can solve this problem. However, for continental-sized transportation networks, Dijkstra's algorithm would take up to 10 seconds for finding a suitable connection, which is way too slow for practical applications. Hence, many speed-up techniques have been developed during the last years with the fastest ones yielding query times of few microseconds on road networks.

However, most developed techniques require the network to be *static* or only allow a small number of updates. In practice, however, travel duration often depends on the departure time. It turns out that efficient models for routing in almost all transportation systems, e.g., timetable information for railways or scheduling for airplanes, are based on *time-dependent* networks. Moreover, road networks are not static either: there is a growing body of data on travel times of important road segments stemming from road-side sensors, GPS systems inside cars, traffic simulations, etc. Using this data, we can assign *speed profiles* to roads. This yields a time-dependent road network.

Switching from a static to a time-dependent scenario is more challenging than one might expect: The input size increases drastically as travel times on congested motorways change during the day. On the technical side, most static techniques rely on bidirectional search, i.e., a second search is started from the target. This concept is complicated in time-dependent scenarios since the arrival time would have to be known in advance for such an approach. Moreover, possible problem statements for shortest paths become even more complex in such networks. A user could ask at what time she should depart in order to spend as little time traveling as possible. As a result, none of the existing high-performance techniques can be adapted to this realistic scenario easily.

Furthermore, the fastest route in transportation networks is often not the "best" one. For example, users traveling by train may be willing to accept longer travel times if the number of required transfers is lower or the cost of a journey with longer duration is cheaper. We end up in a multi-criteria scenario in which none of the high-performance approaches developed in the last years can be applied easily.

In this work, we introduce the first efficient, provably correct, algorithms for route planning in these augmented scenarios. Therefore, we follow the concept of algorithm engineering by designing, analyzing, implementing, and evaluating speed-up techniques for Dijkstra's algorithm. For an augmentation, we additionally pursue a very systematic approach. First we identify basic concepts for accelerating shortest path queries and analyze their drawbacks. By adding a hierarchical component, i.e., contraction, we are able to remedy those drawbacks and derive speed-up techniques with similar performance in time-independent networks as existing approaches. However, due to their foundation on basic concepts, their augmentations are easier than for existing approaches.

On top of the techniques for augmented scenarios, we accelerate the fastest techniques for time-independent route planning as well. We achieve this by adding goal-direction. The main challenge here is to keep the preprocessing effort limited. The key idea is that it is sufficient to apply goal-direction only on a small subgraph representing the upper part of the hierarchy.

Finally, we evaluate the *robustness* of speed-up techniques with respect to the input. It turns out that some techniques are more robust than others. The most robust techniques are introduced in this thesis.

Contents

Chapter 1

Introduction

Finding the quickest connection in transportation networks is a problem familiar to anybody who ever travelled. While in former times, route planning was done with maps at the kitchen's table, nowadays computer based route planning is established: Finding the best train connection is done via the internet while route planning in road networks is often done with mobile devices.

An efficient approach to tackle this problem derives from graph theory. We model the transportation network as a graph and apply travel times as a metric on the edges. Computing the shortest path in such a graph then yields the provably quickest route in the corresponding transportation network. In principle, Dijkstra's classical algorithm [Dij59] can solve this problem. However, for continental-sized transportation networks (consisting of up to 45 million road segments), Dijkstra's algorithm would take up to 10 seconds for finding a suitable connection, which is way too slow for practical applications. Roughly speaking, Dijkstra computes the distance to all possible locations in the network being closer than the target we are interested in. Clearly, it does not make sense to compute all these distances if we are only interested in the path between two points. Hence, many speed-up techniques have been developed within the last years. Such techniques split the work into two parts. During an *offline* phase, called preprocessing, we compute additional data that accelerates queries during the *online* phase. By exploiting several properties of a transportation network, the fastest techniques can obtain the quickest path in road networks within microseconds for the price of few hours of preprocessing. See Fig. 1.1 for an example of the search space of a speed-up technique compared to Dijkstra's algorithm.

However, all recently developed techniques require the network to be *static* which is an unrealistic assumption: road networks for example are predictably congested during rush hours. This realistic scenario is dealt with by *time-dependent* networks, i.e., the travel duration depends on the departure time. It turns out that efficient models for routing in almost all transportation systems, e.g., timetable information for railways or scheduling for airplanes, are based on time-dependent networks. Unfortunately, switching from a static to a time-dependent scenario is more challenging than one might expect: The input size increases drastically as travel times on congested motorways change during the day. Moreover, shortest paths heavily depend on the time of departure, e.g., during rush hours it might pay off to avoid highways. On the technical side, most static techniques rely on bidirectional search, i.e., a second search is started from the target. However, this concept is prohibited in time-dependent scenarios as the arrival time would have to be known in advance for such a procedure. As a result, none of the existing high-performance techniques can be adapted to this realistic scenario easily. This is one of the main reasons why commercial systems rely on approximative algorithms.

Another open problem for route planning is that the quickest route is often not the best one. A user might be willing to accept slightly longer travel times if the cost of

the journey is less. A common approach to cope with such a situation is to find *Pareto-optimal* (concerning other metrics than travel times) routes. Such routes have the property that each route is better than any other route with respect to at least one metric under consideration, e.g., travel costs or number of train changes.

In this work, we present efficient route planning algorithms for the augmented scenarios stated above. We hereby follow the concept of *algorithm engineering*: We design, analyze, implement, and experimentally evaluate augmented route planning algorithms. The evaluation is done on real-world transportation networks (roads and railways) and networks deriving from other fields of computer science.

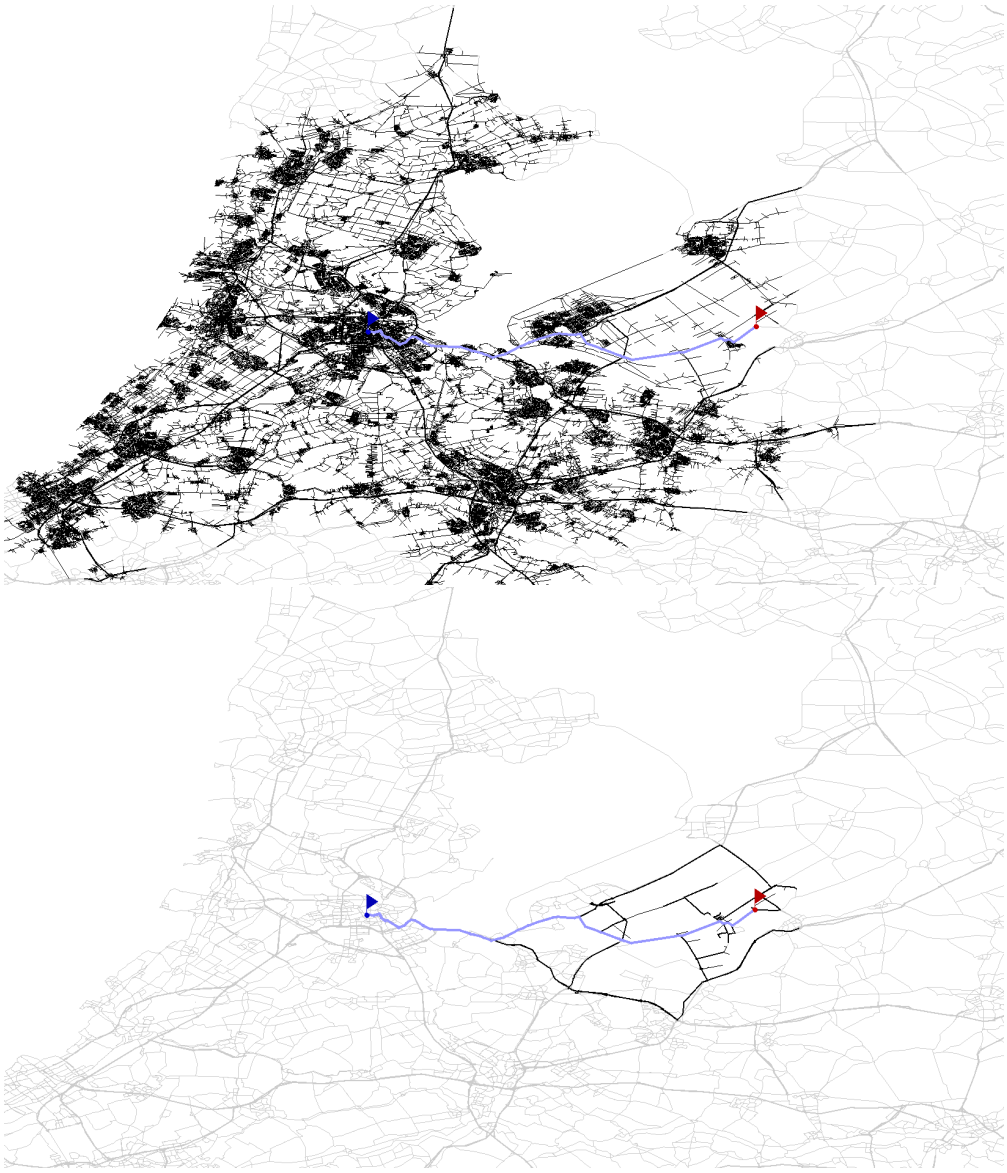


Figure 1.1: Search space of different algorithms for the same sample query in a road network. The upper figure depicts the search space of Dijkstra, the lower for a speed-up technique. Black edges are touched by algorithms, grey ones stay untouched. The shortest path is drawn thicker in light blue. We observe that the speed-up technique touches considerably fewer edges than Dijkstra.

1.1 Problem Definition

Given a directed graph $G = (V, E)$ with $n := |V|$ nodes and $m := |E|$ edges. In this work, we deal with the following three problems.

- *time-independent point-to-point shortest paths (TIPPSP)*:
  Compute the shortest path between a given source s and a given target t with respect to a length function $len : E \to \mathbb{R}^+$.
- *time-dependent point-to-point shortest paths (TDPPSP)*:
  Compute the shortest path between a given source s and given target t with respect to a *time-dependent* length function $len : E \times \mathbb{R}^+ \to \mathbb{R}^+$ for a given *departure time* τ.
- *point-to-point Pareto paths (PPPP)*:
  Compute all *Pareto* paths between a given source s and given target t with respect to a *k-dimensional* length function $len : E \to \mathbb{R}_+^k$.

Note that one crucial difference between the three problems is that the length function *len* assigned to the edges of the graph differs.

1.2 Related Work

We split related work into three logical parts: Time-independent, time-dependent, and multi-criteria route planning.

1.2.1 Time-Independent Route Planning

As already mentioned above, most recent research focused on this scenario. Hence, a bunch of speed-up techniques have been developed during the last years. Academic research on speed-up techniques for transportation networks started in 1999 on railway networks [SWW99] leading to several follow-up studies. However, a turning point was 2005 since at this point huge road networks were made available to the research community, immediately leading to a kind of "horse race" for the fastest technique on these inputs. The climax of the development was surely the DIMACS Challenge of shortest paths [DGJ06]. In the following, we shortly explain each technique separately, a chronological summary of the horse race can be found in [DSSW09a].

Two different approaches have been established during the last years: *hierarchical* techniques try to identify an important subgraph and carry out most of the time-consuming work on this subgraph, while *goal-directed* techniques work on the complete graph and try to "guide" the search to the target.

Basics

Dijkstra's Algorithm [Dij59] – the classical algorithm for route planning – maintains an array of *tentative distances* $D[u]$ for each node $u \in V$. The algorithm *visits* (or *settles*) the nodes of the network in non-descending order of their distance to the source node. As soon as the target t is settled, the search can be stopped since all nodes settled after t have a greater distance to s. The size of the search space, i.e., the number of settled

nodes, is $O(n)$ and $n/2$ (nodes) on the average. In this work, we assess the quality of route planning algorithms by looking at their *speed-up* compared to Dijkstra's algorithm, i.e., by which factor they can compute shortest paths faster.

Priority Queues. A naive implementation of Dijkstra's algorithm has a running time of $O(n^2)$ since finding the next node to settle takes $O(n)$ (linear search of candidates). However, the algorithm can be implemented using $O(n)$ priority queue operations. In the comparison based model this leads to $O(n \log n)$ execution time. In other models of computation (e.g. [Tho03]) and on the average [Mey01], better bounds exist. However, in practice the impact of priority queues on performance for large transportation networks is rather limited since cache faults for accessing the graph are usually the main bottleneck. In addition, our experiments indicate that the impact of priority queue implementations diminishes with advanced speed-up techniques that dramatically reduce the queue sizes.

Bidirectional Dijkstra [Dan62, GH05]. The most straightforward speed-up technique is bidirectional search. An additional (reverse) search is started from the target node and the query stops as soon as both searches meet. Note that most sophisticated methods are bidirectional approaches.

Geometric Goal Directed Search (A^*). The intuition behind goal directed search is that shortest paths 'should' lead in the general direction of the target. A^* search [HNR68] achieves this by modifying the weight of edge (u, v) to $len(u, v) - \pi(u) + \pi(v)$ where $\pi(v)$ is a lower bound on $d(v, t)$. Note that this manipulation shortens edges that lead towards the target. Since the added and subtracted *vertex potentials* $\pi(v)$ cancel out along any path, this modification of edge weights preserves shortest paths. Moreover, as long as all edge weights remain nonnegative, Dijkstra's algorithm can still be used. The classical way to use A^* for route planning in transportation networks estimates $d(v, t)$ based on the Euclidean distance between v and t and the average speed of the fastest connection anywhere in the network. Since this is a very conservative estimation, the speed-up for finding quickest routes is rather small. In [GH05], even a *slow-down* of more than a factor of two is reported since the search space is not significantly reduced but a considerable overhead is added.

Exploiting Hierarchy

Small Separators. Transportation networks are almost planar, i.e., most edges intersect only at nodes. Hence, techniques developed for planar graphs will often also work for transportation networks. Using $O(n \log^2 n)$ space and preprocessing time, query time $O(\sqrt{n} \log n)$ can be achieved [FR06, Kle05] for directed planar graphs without negative cycles. Queries accurate within a factor $(1+\epsilon)$ can be answered in near constant time using $O((n \log n)/\epsilon)$ space and preprocessing time [Tho01]. Most of these theoretical approaches look difficult to use in practice since they are complicated and need superlinear space. The approach from [Tho01] has recently been implemented and experimentally evaluated on a road network with one million nodes [MZ07].

Multi-Level Techniques. The first published practical approach for fast route planning [SWW99, SWW00] uses a set of nodes V_1 whose removal partitions the graph $G = G_0$ into small components. Now consider the *overlay graph* $G_1 = (V_1, E_1)$ where edges in

E_1 are *shortcuts* corresponding to shortest paths in G that do not contain nodes from V_1 in their interior. Routing can now be restricted to G_1 and the components containing s and t respectively. This process can be iterated yielding a multi-level method [SWZ02, HSW06, HSW08, Hol08]. A limitation of this approach is that the graphs at higher levels become much more dense than the input graphs thus limiting the benefits gained from the hierarchy. Also, computing small separators and shortcuts can become quite costly for large graphs.

Reach-Based Routing. Let $R(v) := \max_{s,t \in V} R_{st}(v)$ denote the *reach* of node v where $R_{st}(v) := \min(d(s,v), d(v,t))$. Gutman [Gut04] observed that a shortest-path search can be stopped at nodes with a reach too small to get to source or target from there. Variants of reach-based routing work with the reach of edges or characterize reach in terms of geometric distance rather than shortest-path distance. The first implementation had disappointing speed-ups (e.g., compared to [SWW99]) and preprocessing times that would be prohibitive for large networks.

Highway Hierarchies (HH) [SS05, SS06] group nodes and edges in a hierarchy of levels by alternating between two procedures. First, the network is contracted by removing low-degree nodes and then "important" edges—the highway edges—are identified. By rerunning those two steps, a natural hierarchy of the network is obtained. The contraction phase builds the *core* of a level and adds shortcuts to the graph. The identification of highway edges is done by local Dijkstra executions. The most advanced variant stops building the hierarchy at a certain point and computes a *distance* table containing all distances between the core-nodes of the highest level. The advantages of HH are very low preprocessing and query times on road networks with travel times.

Advanced Reach-Based Routing. It turns out that the preprocessing techniques developed for HHs can be adapted to preprocessing reach information [GKW06]. This makes reach computation faster and more accurate. More importantly, shortcuts make queries more effective by reducing the number of nodes traversed and by reducing the reach-values of the nodes bypassed by shortcuts. Reach-based routing is slower than HHs both with respect to preprocessing time and query time. However, the latter can be improved by a combination with goal-directed search to a point where both methods have similar performance.

Highway-Node Routing (HNR) [SS07] computes for a given sequence of node sets $V =: V_0 \supseteq V_1 \supseteq \ldots \supseteq V_L$ a hierarchy of *overlay graphs* [SWW99, SWZ02, HSW06, HSW08]: the level-ℓ overlay graph consists of the node set V_ℓ and an edge set E_ℓ that ensures the property that all distances between nodes in V_ℓ are equal to the corresponding distances in the underlying graph $G_{\ell-1}$. A bidirectional query algorithm takes advantage of the multi-level overlay graph by never moving downwards in the hierarchy—by that means, the search space size is greatly reduced. A node classification provided by the Highway Hierarchies approach was used to define the highway-node sets leading to a multi-level overlay graph with about ten levels. Hence, the performance of Highway-Node Routing is very similar to pure Highway Hierarchies but the advantage of the former is its capability to efficiently *update* the preprocessing and low memory consumption.

Contraction Hierarchies (CH) are a special case of Highway-Node Routing where we have n levels – one level for each node [GSSD08]. Such a fine hierarchy can improve

query performance by a considerable factor. During preprocessing, the input graph G is transferred to a search graph G' by storing only edges directing from unimportant to important nodes. As a remarkable result, G' is smaller than G yielding a negative overhead per node. Finally, by this transformation the query is simply a plain bidirectional Dijkstra search operating on G'.

Distance Tables. Once a hierarchical routing technique (e.g., HH, HNR, CH) has shrunk the size of the remaining network G' to $\Theta(\sqrt{n})$, one can afford to precompute and store a complete distance table for the remaining nodes [SS06]. Using this table, one can stop a query when it has reached G'. To compute the shortest-path distance, it then suffices to lookup all shortest-path distances between nodes entering G' in forward and backward search respectively. Since the number of entrance nodes is not very large, one can achieve a speed-up close to two compared to the underlying hierarchical technique.

Transit-Node Routing precomputes not only a distance table for important (*transit*) nodes but also all relevant connections between the remaining nodes and the transit nodes. Independently, three approaches proved successful for selecting transit nodes: separators [Mül06, DHM+09], border nodes of a partition [BFM+07, BFSS07, BFM09], and nodes categorized as important by other speed-up techniques [BFM+07, BFSS07, SS09]. It turns out that for route planning in road networks, the latter approach is the most promising one. Since only about 7–10 such *access connections* are needed per node one can 'almost' reduce routing in large road networks to about 100 table lookups. Interestingly, the difficult queries are now the local ones where the shortest path does not touch any transit node. This problem can be solved by introducing several *layers* of transit nodes. Between lower layer transit nodes, only those routes need to be stored that do not touch the higher layers.

Advanced Goal-Directed Search

Edge Labels. The idea behind edge labels is to precompute information for an edge e that specifies a set of nodes $M(e)$ with the property that $M(e)$ is a superset of all nodes that lie on a shortest path starting with e. In an s–t query, an edge e need not be relaxed if $t \notin M(e)$. In [SWW99], $M(e)$ is specified by an *angular range*. More effective is information that can distinguish between long range and short range edges. In [WW03, Wil05] many *geometric containers* are evaluated. Very good performance is observed for axis parallel rectangles. A disadvantage of geometric containers is that they require a complete all-pairs shortest-path computation. Faster precomputation is possible by partitioning the graph into k regions that have similar size and only a small number of boundary nodes. Now $M(e)$ is represented as a k-vector of *arc flags* [Lau04, KMS05, MSS+05, MSS+06, Sch06] where flag i indicates whether there is a shortest path containing e that leads to a node in region i. Arc flags can be computed using a single-source shortest-path computation from all boundary nodes of the regions. A further improvement gets by with only one (though comparatively expensive) search for each region [HKMS09]. Our work is largely based on the concept of arc flags, hence a more detailed introduction to this approach is given in Section 3.3.

Landmark A (ALT).* Using the triangle inequality, quite strong bounds on shortest-path distances can be obtained by precomputing distances to a set of *landmark* nodes

(≈ 16) that are well distributed over the far ends of the network [GH05, GW05]. Using reasonable space and much less preprocessing time than for edge labels, these lower bounds yield considerable speed-up for route planning. Since landmarks are extremely helpful in augmented scenarios, ALT is described in more detail in Section 3.2.

Precomputed Cluster Distances (PCD). In [MSM06], a different way to use precomputed distances for goal-directed search is given. The network is partitioned into clusters and then a shortest connection between any pair of clusters U and V, i.e., $\min_{u \in U, v \in V} d(u, v)$, is precomputed. PCDs cannot be used together with A^* search since reduced edge weights can become negative. However, PCDs yield upper and lower bounds for distances that can be used to prune search. This gives speed-up comparable to landmark-A^* using less space.

Combinations. Many speed-up techniques can be combined. In [SWW99, SWW00], a combination of a special kind of geometric container [WWZ05], the separator-based multi-level method, and A^* search yields a speed-up of 62 for a railway transportation problem. In [HSW04, HSWW06], combinations of A^* search, bidirectional search, the separator-based multi-level method, and geometric containers are studied: Depending on the graph type, different combinations turn out to be best. For real-world graphs, a combination of bidirectional search and geometric containers leads to the best running times. For public transportation however, a combination of Arc-Flags and ALT harmonizes well [DPW08].

REAL. In [GKW06], the advanced version of REach has successfully been combined with landmark-based A^* search (the ALt algorithm), obtaining the REAL algorithm. In the most recent version [GKW07], a variant is introduced where landmark distances are stored only with the more important nodes, i.e., nodes with high reach values. By this means, the memory consumption can be reduced significantly.

HH [DSSW09b]* combines Highway Hierarchies (HH) with landmark-based A* search. Similar to [GKW07], the landmarks are not chosen from the original graph, but for some level k of the highway hierarchy, which reduces the preprocessing time and memory consumption. As a result, the query works in two phases: in an initial phase, a non-goal-directed highway query is performed until all entrance points to level k have been discovered; for the remaining search, the landmark distances are available so that the combined algorithm can be used.

1.2.2 Time-Dependent Route Planning

Basics. In [CH66], Dijkstra's algorithm is extended to the time-dependent case based on the assumption that all edges of the network fulfill the FIFO property. The FIFO property is also called the *non-overtaking property*, because it states that if A leaves the node u of an edge (u, v) before B, B cannot arrive at node v before A. Computation of shortest paths in FIFO networks is polynomially solvable [KS93]. In non-FIFO networks, complexity depends on the restriction whether waiting at nodes is allowed. If waiting is allowed, the problems stays polynomially solvable, if it is not allowed, the problem is NP-hard [OR90].

Speed-up Techniques. Compared to the time-independent scenario, much less work has been done on speed-up techniques for time-dependent route planning. Goal-directed search based on A^* with Euclidean potentials has been adapted to time-dependent road networks in [Fli04], while its adaption to time-dependent railway networks can be found in [PSWZ04b, Sch05, PSWZ07, DMS08].

Basic algorithmic ideas how to augment a *hierarchical* technique, i.e., Contraction Hierarchies, can be found [BGS08]. However, it took quite some time until first experimental results were available [BDSV09]. It seems as if the performance of time-dependent Contraction Hierarchies is comparable to the techniques presented in this work but for the price of a very high amount of preprocessed data. The main reason for this is the fact that shortcuts are much more expensive in terms of space consumption in time-dependent scenarios (cf. Chapter 5 for details).

1.2.3 Pareto Route Planning

Basics. The straightforward approach to find all Pareto optimal paths is the generalization [Han79, Mar84, Möh99] of Dijkstra's algorithm: Each node $v \in V$ gets a number of multi-dimensional labels assigned, representing all Pareto paths to v. For the bicriteria case, [Han79] was the first presenting such generalization, while [The95] describes multi-criteria algorithms in detail. By this generalization, Dijkstra loses the label-setting property, i.e., now a node may be visited more than once. It turns out that a crucial problem for multi-criteria routing is the number of labels assigned to the nodes. The more labels are created, the more nodes are reinserted in the priority queue yielding considerably slow-downs compared to the single-criteria setup. In the worst case, the number of labels can be exponential in $|V|$ yielding impractical running times [Han79]. Hence, [Han79, War87] present an FPAS for the bicriteria shortest path problem.

Speed-up Techniques. Most of the work on speed-up techniques for multi-criteria scenarios was done on networks deriving from timetable information. Here, [MW01] observed that in such networks, the number of labels is often limited such that the brute force approach for finding *all* Pareto paths is often feasible. Experimental studies finding all Pareto paths in timetable graphs can be found in [PSWZ04a, PSWZ04b, Sch05, PSWZ07, MS07, GMS07, DMS08]. However, to the best of our knowledge, all previous work only uses basic speed-up techniques for accelerating the multi-criteria query. In most cases a special version of A^* is adapted to this scenario. Unfortunately, the resulting speed-ups only reach up to a factor of 5 which is much less than for the (single-criteria) speed-up techniques developed during the last years.

1.3 Main Contributions

In this work, we present provably correct speed-up techniques for efficient routing in time-dependent and multi-criteria scenarios. We therefore follow a very systematic approach. First, we identify basic ingredients in Chapter 3 and discuss known drawbacks of those approaches. More precisely, we recapture the concept of landmarks, Arc-Flags, and contraction. By combining these ingredients in Chapter 4 we obtain speed-up techniques which can compete with known approaches confirmed by an extensive experimental study. Besides that, we further accelerate known hierarchical techniques by adding goal-direction

via Arc-Flags. We obtain the fastest route planning algorithms for time-independent scenarios. However, the main contribution of this work is the augmentation of techniques to time-dependent scenarios. The advantages of our combinations over existing techniques is that due to their clear foundation on basic ingredients, adaption to augmented scenarios seems more promising than for other approaches. We support this theory in Chapter 5 where we first augment the ingredients and then obtain the first efficient techniques for route planning in time-dependent networks, again confirmed by an extensive experimental study. On top of that, we follow the same approach in order to augment techniques to multi-criteria scenarios. We present the first efficient speed-up technique for Pareto route planning, again supported by an experimental study.

Summarizing, this thesis provides four important contributions: two speed-up techniques working in augmented scenarios, the fastest known techniques for time-independent routing, and an experimental study on the robustness of speed-up techniques. In the following, we describe each contribution in more detail.

SHARC. In this thesis, we introduce SHARC-Routing, a fast and robust approach for *unidirectional* routing in large networks. The central idea of SHARC (Shortcuts + Arc-Flags) is to integrate the concept of contraction into Arc-Flags. 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. Furthermore, SHARC allows to perform very fast queries—*without* updating the preprocessing—in scenarios where metrics are changed frequently, e.g., different speed profiles for fast and slow cars. We also introduce an interesting variant of SHARC by removing all shortcuts from the graph after preprocessing. This variant may be very helpful in scenarios with very limited memory, e.g., portable navigation systems. In case a user needs even faster query times, our approach can also be used as a bidirectional algorithm boosting performance even more.

Augmenting SHARC. Although preprocessing is based on basic concepts, SHARC yields a low preprocessing effort combined with a unidirectional query algorithm being very similar to plain Dijkstra. In order to augment SHARC, we first extend the basic ingredients such that correctness can be guaranteed in time-dependent scenarios. It turns out that we can leave the general concept of SHARC untouched. An experimental evaluation with real world time-dependent transportation networks confirms the excellent performance of time-dependent SHARC: The speed-up over Dijkstra for road networks is between 60 and > 5000, depending on the degree of perturbation of the network and preprocessing effort. The corresponding value for railway networks is 26.5.

In Chapter 6 we show that this approach even works for a multi-criteria scenario. By augmenting the ingredients of SHARC to multi-criteria variants we again can assemble an augmented variant of SHARC yielding excellent query times in such a scenario. The speed-up over the generalized Dijkstra's algorithm is the same as in a single-criteria scenario, i.e., up to a factor of 15 000. However, it turns out that in road networks, multi-criteria searches yield too high a number of possible routes to the target. Hence, we introduce several reasonable constraints how to prune unattractive paths both during preprocessing

and queries. Here, the key observation is that we define a main metric (we use travel times) and only allow other paths if they do not yield too long a delay. Moreover, we also introduce a constraint called *pricing*. Paths with longer travel times are only accepted if they yield significant improvements in other metrics. With these additional constrains we are able to compute reasonable Pareto paths in continental-sized road networks.

CALT. The main disadvantage of SHARC is that updating the preprocessing in case of delays is complicated. Hence, we follow a second approach for routing in augmented scenarios, based on landmarks and contraction. First, we analyze the performance of landmarks in dynamic time-independent scenarios. It turns out that ALT works very well in dynamic scenarios but query performance is not competitive to known approaches. Besides that, pure ALT yields a high memory consumption prohibited in real-world applications. Hence, we introduce CALT remedying both drawback without violating the advantages of pure ALT. The key observation is that we extract an important subgraph, called the *core*, of the input graph and use only the core as input for the preprocessing-routine of ALT. As a result, we derive a two-phase query algorithm, similar to partial landmark REAL or HH*. During phase 1 we use plain Dijkstra to reach the core, while during phase 2, we use ALT in order to accelerate the search within the core. We obtain a very robust technique that unfolds its full potential in a dynamic time-dependent scenario.

Augmenting CALT. Adaption of landmarks and contraction to time-dependent scenarios is straightforward. However, CALT is only competitive if applied in a bidirectional manner. As already mentioned, a backward search is problematic since the arrival time is not known in advance. So, the main challenge of adaption is augmenting bidirectionally search. In Chapter 5, we present a general approach for searching bidirectionally in time-dependent networks. The key idea is to perform a *time-independent* backward search that bounds the set of nodes a time-dependent forward search has to visit. Experiments confirm that this approach yields very good results in time-dependent road networks. Next, we exploit the fact that landmarks work well in dynamic scenarios: We show how to update the preprocessing in case the cost functions change. As a result, we can efficiently compute quickest routes in *fully dynamic* time-dependent road networks.

Combinations. The third main contribution of this thesis is presenting the fastest known speed-up techniques for time-independent networks as well. By combining Arc-Flags with the fastest known hierarchical technique, i.e., Contraction Hierarchies and Transit-Node Routing, those concepts can further be accelerated while keeping preprocessing effort within reasonable time and space. The key idea is to use a purely hierarchical method until a specific point during the query. As soon as we have reached an 'important' subgraph, i.e., a high level within the hierarchy, we turn on arc-flags. Our most successful combination assembling from Contraction Hierarchies (CH) and Arc-Flags moderately increases preprocessing effort over pure CH but query performance is almost as good as Transit-Node Routing in road networks: On average, we settle only 45 nodes for computing the distance between two random nodes in a continental-sized road network. The advantage of this combination over Transit-Node Routing is its very low space consumption. However, we are also able to improve the performance of Transit-Node Routing by adding goal-direction via Arc-Flags to this approach. As a result, the number of required table lookups can be reduced by a factor of 13, resulting in average query times of less than 2 μs—more than three million times faster than Dijkstra's algorithm.

Study on Robustness. Up to now, due to the availability of huge road networks, much of recent research focused only on such networks [DSSW09a]. However, fast algorithms are needed for other applications as well. One might expect that all speed-up techniques can simply be used in any other application, yet several problems arise: several assumptions which hold for road networks may not hold for other networks, e.g., in timetable information bidirectional search is prohibited as the arrival time is unknown in advance. Performance is the other big issue. The fastest methods heavily exploit properties of road networks in order to gain their huge speed-ups. In Section 4.6, we evaluate the most prominent speed-up techniques developed in this thesis and during the last years on different types of input classes. It turns out that some techniques are more robust to the input than others.

1.4 Overview

This work is organized as follows:

Chapter 2 settles basic definitions and fundamentals of this thesis. In addition, models for route planning in road and train networks are recaptured from literature.

Chapter 3 introduces the basic ingredients our work is based on. More precisely, we discuss the concepts of landmarks, arc-flags, and contraction.

Chapter 4. Before working on time-dependent scenarios, we need to settle the question which technique to use for augmentation. This chapter introduces several interesting candidates for this task. We show that an approach based solely on landmarks is presumably the easiest one but for the price of bad query performance. We try to remedy this drawback by adding a contraction step yielding a very potential speed-up technique, called CALT. Still, this approach is based on bidirectional search making adaption to augmented scenarios complicated. Hence, we introduce SHARC, the fastest known techniques for *unidirectional* routing. Due to its unidirectional nature and its clear basis on known ingredients, this approach is the first choice for augmentation. Besides this, we show—allowing bidirectional search—how to further accelerate the fastest known hierarchical techniques. An experimental study on road networks reveals that these new combinations outperform any other speed-up technique. However, unidirectional SHARC can compete even with those techniques. An extensive experimental evaluation on different inputs reveals that some techniques highly depend on the type of input while others are very robust to the input.

Chapter 5 is probably the most innovative part of this thesis: augmenting speed-up techniques from Chapter 4 to time-dependent networks such that correctness can still be guaranteed. Therefore, we first augment the ingredients from Chapter 3 to time-dependent networks. Then, we follow two approaches to perform time-dependent queries. On the one hand, we augment the fastest unidirectional technique from Chapter 4, i.e., SHARC. On the other hand, we show how to solve the problem of bidirectional search. The backward search is executed on a *time-independent* graph and is only used to bound the nodes a *time-dependent* forward search has to visit. We also show how to efficiently route in time-dependent networks where cost functions change. An extensive experimental study reveals interesting facts on time-dependent routing general.

Chapter 6. Using a similar approach as for augmenting time-independent speed-up techniques to time-dependent ones, we present a multi-criteria variant of SHARC. Again, we first augment basic ingredients from Chapter 3 to multi-criteria variants such that correctness can still be guaranteed. An experimental evaluation confirms that multi-criteria SHARC provides similar speed-ups like as in a single-criteria setup.

Chapter 7 concludes our work with a discussion on the insights gained in this thesis and challenging open problems.

Chapter 2

Fundamentals

In this chapter we settle the very fundamentals of our work. This includes graphs, paths, partitions and approaches for modeling transportation networks as graphs.

2.1 Graphs

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

Edge Functions. The main difference between time-independent, time-dependent, and Pareto route planning is the length function assigned to the edges. In the time-independent scenario, we use a positive length function $len : E \to \mathbb{R}^+$.

Time-Dependency. In this setup we use functions instead of constants for specifying edge weights. Throughout the whole work, we restrict ourselves to a function space $\mathbb{F}$ consisting of positive *periodic* functions $f : \Pi \to \mathbb{R}^+, \Pi = [0, p], p \in \mathbb{N}$ such that $f(0) = f(p)$ and $f(x) + x \leq f(y) + y$ for any $x, y \in \Pi, x \leq y$. Note that these functions respect the FIFO property. In the following, we call Π the *period* of the input. We restrict ourselves to directed graphs $G = (V, E)$ with time-dependent length functions $len : E \to \mathbb{F}$. We use $len : E \times [0, p] \to \mathbb{R}^+$ to evaluate an edge for a specific departure time. Note that our networks fullfill the FIFO-property if we interpret the length of an edge as travel times due to our choice of $\mathbb{F}$. The composition of two functions $f, g \in \mathbb{F}$ is defined by $f \oplus g := f + (g \circ (f + id))$. Moreover, we need to *merge* functions, which we define by $\min(f, g)$. The upper bound of f is noted by $\overline{f} = \max_{x \in \Pi} f(x)$, the lower by $\underline{f} = \min_{x \in \Pi} f(x)$. An underapproximation $\downarrow f$ of a function f is a function such that $\downarrow f(x) \leq f(x)$ holds for all $x \in \Pi$. An overapproximation $\uparrow f$ is defined analogously. Bounds and approximations of our time-dependent edge function *len* is given by analogous notations. Obviously, one can obtain a time-independent graph $\underline{G}$ from a time-dependent graph G by substituting the time-dependent length function by $\underline{len}$. We call $\underline{G}$ the *lower bound graph* of G.

Pareto Scenario. Here, we assign more than one weight to each edge. In this work, we restrict ourselves to vectors in $\mathbb{R}_+^k$. For a k, let $L = (w_1, \ldots, w_k)$ and $L' = (w'_1, \ldots, w'_k)$ be two labels. We use the following notation and operations in $\mathbb{R}_+^k$: L *dominates* another label L' if $w_i < w'_i$ holds for one $1 \leq i \leq k$ and $w_i \leq w'_i$ holds for each $1 \leq j \leq k$. The sum of L and L' is defined by $L \oplus L' = (w_1 + w'_1, \ldots, w_k + w'_k)$. We call $\underline{L} = \min_{1 \leq i \leq k} w_i$ the component element of L, the maximum component $\overline{L}$ is defined analogously.

2.2 Paths

A path P in G is a sequence of nodes $(u_1, \ldots, u_k)$ such that $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$. In time-independent scenarios, the *length* of a path is given by $\sum_{i=1}^{k-1} len(u_i, u_{i+1})$. A path between two nodes s and t with minimum length is called a *shortest s–t path*. By $d(s,t)$ we denote the length of such a path.

Time-Dependency. In time-dependent scenarios, the length $\gamma_\tau(P)$ of a path P departing from u_1 at time τ is recursively given by

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

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

Pareto-Paths. In a multi-criteria scenario, the length (s,t) of an s–t path $P = (e_1, \ldots, e_r)$ is given by $len(e_1) \oplus \ldots \oplus len(e_r)$. In contrast to a single-criteria scenario, many paths exist between two nodes that do *not* dominate each other. In this work, we are interested in the *Pareto-set* $\mathcal{D}(s,t) = \{d_1(s,t) \ldots d_x(s,t)\}$ consisting of all non-dominated path-lengths $d_i(s,t)$ between s and t. We call $|\mathcal{D}(s,t)|$ the *size* of a Pareto-set. Note that by storing a predecessor for each d_i, we can compute all Pareto-paths as well.

2.3 Partitions

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

2.4 Models

One focus of this thesis is routing in transportation networks. In this section, we briefly present how to efficiently model road and train networks as graphs. For more details, we refer the interested reader to [Mül05, Vol08] and [Sch05, MSWZ07].

2.4.1 Road Networks

In road networks, we have roads and junctions. We intoduce a node for each junction being connected by a directed edge if there is a direct connection between them. The edge weight is set to the average travel time from one junction to another including waiting at traffic lights. Note that since roads may be one-way streets, some edges in this model are directed. However, most streets are two-way and thus, most of the graph is undirected.

Traffic Jams. In case of a traffic jam, we may increase the travel time on the edge modeling the perturbed edge. Note that for this kind of updates, travel times may increase and decrease afterwards but will not drop below the transit times of an 'empty' road.

Time-Dependency. In road networks, many traffic jams are predictable. We know that during rush hours, highways are crowded while at night they are empty. In order to cope with this scenario correctly, we assign travel time functions to each edge. In this work, we use piecewise linear functions for modeling time-dependency in road networks. Each edge gets assigned a number of sample points that depict the travel time on this road at the specific time. Evaluating a function at time τ is then done by linear interpolation between the points left and right to τ.

In the following, we need to *link* two piecewise linear functions f, g to $f \oplus g$, modeling the duration for traversing g directly after f. This is done as follows. Let $I(f) = \{(t_1^f, w_1^f), \ldots, (t_l^f, w_l^f)\}$ with $t_i^f \in \Pi, w_i^f \in \mathbb{R}^+, 1 \leq i \leq l$ be the interpolation points of f and $I(g) = \{(t_1^g, w_1^g), \ldots, (t_k^g, w_k^g)\}$ those of g. Then the interpolation points $\{\big(t_1^f, w_1^f + g(t_1^f + w_1^f)\big), \ldots, \big(t_l^f, w_l^f + g(t_l^f + w_l^f)\big)\}$ are obviously included in $I(f \oplus g)$. However, we also have to add some more interpolation points, namely those from arriving at the timestamps t_j^g of g. More precisely, let $t_1^{-1}, \ldots, t_k^{-1}$ be chosen such that $f(t_j^{-1}) + t_j^{-1} = t_j^g$ holds for all $1 \leq j \leq k$. Then, we also have to add $\{(t_1^{-1} \mod \Pi, f(t_1^{-1}) + w_1^g), \ldots, (t_k^{-1}$

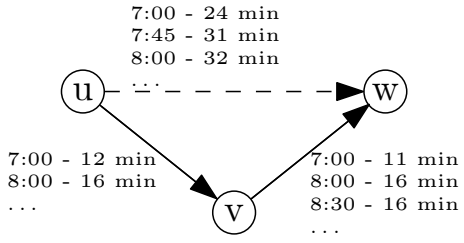


Figure 2.1: Time-dependent composition in road networks. A function depicting the travel time from u to w via v yields an additional interpolation point at 7:45 because otherwise the interpolated travel time at 7:45 would be 30 minutes instead of 31.

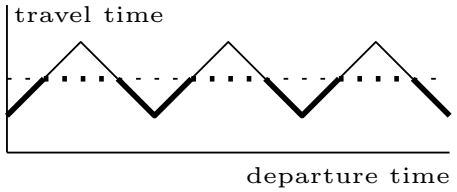


Figure 2.2: Time-dependent merging of two piecewise linear functions f and g in road networks. f is drawn solid, g dotted, the merged function is drawn thicker. Note that $P(\min(f, g)) > P(f) + P(g)$ holds.

mod $\Pi, f(t_k^{-1}) + w_k^g)\}$ to $I(f \oplus g)$. See Figure 2.1 for an example. Let $P(f)$ be the number of interpolation points of f. Note that the composed function $f \oplus g$ may have up to $P(f) + P(g)$ number of interpolation points in the worst case.

We also have to merge two piecewise linear functions f, g to $\min(f, g)$. Like for linking, this may increase the breakpoints. More precisely, we have to check for all timestamps t_i^f of f whether $w_i^f < g(t_i^f)$ holds. If it holds, we need to keep the interpolation point (t_i^f, w_i^f), otherwise we do not need it. Analogously, we proceed for all timestamps t_j^g of g. However, additional interpolation points have to be added for all intersection points of f and g. Figure 2.2 gives an example.

2.4.2 Timetable Information in Railway Networks

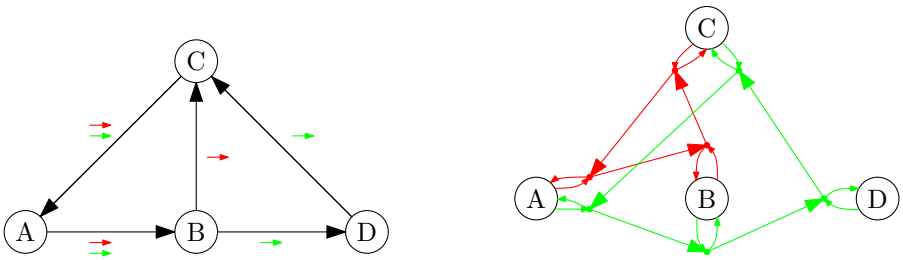
In railway networks, we want to solve the *earliest arrival problem*, i.e. compute the best connection for a given departure time. Three models exist that model this problem such that a shortest path computation solves this problem. Note that in this work, we restrict ourselves to *periodic* timetables.

Condensed Model [SWZ02]. The easiest model is the *condensed* model. Here, a node is introduced for each station and an edge is inserted iff a direct connection between two stations exists. The edge weight is set to be the minimum travel time over all possible connections between these two stations. The advantage of this model is that the resulting graphs are quite small and we are able to use existing time-independent speed-up techniques without modification. Unfortunately, several drawbacks exist. First of all, this model does not incorporate the actual departure time from a given station. Even worse, travel times highly depend on the time of the day and the time needed for changing trains is also not covered by this approach. As a result, the calculated travel time between two arbitrary stations in such a graph is only a *lower bound* of the real travel time. Hence, we do not use this model in this thesis.

Time-Dependent Model [BJ04]. This model tries to remedy the disadvantages of the condensed model. The main idea is to use *time-dependent* edges. Hence, each station is also modeled by a single node and an edge is again inserted iff a direct connection between two stations exist. See Fig. 2.3(a) for a small example. But unlike for the condensed model, several weights are assigned to each edge. For each train, we add an interpolation point to the corresponding edge. The timestamp σ of the interpolation point is the departure time, the weight w the travel time. When we want to evaluate a time-dependent edge at a specific time τ, we identify the interpolation point P with minimum $\sigma - \tau \geq 0$. Then the resulting traveltime is $w + \sigma - \tau$, i.e., the waiting time for the next connection plus its travel duration.

The advantage of this model is its still small size and the obtained travel time is feasible. Furthermore, delays can easily be incorporated: the corresponding weight—representing the delayed connection—of an edge can simply be increased. Note that the time-dependent model can be interpreted as an extension of the condensed model.

Realistic Transfer Times [PSWZ04b]. A problem of the approach described above is that transfer times cannot be incorporated properly. To do so, a time-dependent *train-route model* has to be applied. For each train route, a node is introduced at each station the trains of this route stop. These nodes are connected by time-dependent edges modeling



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

Figure 2.3: Time-dependent railway graphs. For both models, we have four stations and two train routes. The first runs from A to B to C and back to A, while the second runs A→B→C→D→A. In the simple model, switching from one train to another can be done in 0 minutes, while transfer times are incorporated correctly in the realistic model.

the trains running on this route. For each station, a supernode is introduced which is connected to all route nodes of this station modeling the transfer from one train to another. It turns out that this model increases the graph size by approximately a factor of 5. See Fig. 2.3(b) for an example. In this work, we use the train-route model. We guarantee the FIFO property of such graphs by introducing multi-edges.

Interestingly, the composition of two timetable edge-functions f, g is less expensive than in road networks. More precisely, $P(f \oplus g) \leq \min\{P(f), P(g)\}$ holds as the number of relevant departure times is dominated by the edge with less connections. More precisely, we determine for all interpolation points (t_i^f, w_i^f) the so called *connection interpolation point* of g, which is the point $(t_j^g, w_j^g) \in I(g)$ with $t_j^g - t_i^f - w_i^f \geq 0$ minimal. In other words, this is the first connection of g we can catch when taking the connection departing at timestamp t_i^f. Then, we add the interpolation point $(t_i^f, (t_j^g - t_i^f + w_j^g))$ to $I(f \oplus g)$. Since f and g are FIFO functions this ensures correctness. However, it may happen that two interpolation points $\in I(f)$ yield the same connection point of g. In such a situation we only need to keep the point with maximal timestamp since an earlier departure does not pay off. See Figure 2.4 for an example.

The merging of two public transportation functions is straightforward. For each timestamp t_i^f of f we check whether $w_i^f < g(t_i^f)$ holds. If it holds, we add (t_i^f, w_i^f) to $I(\min(f, g))$. We do the same for all timestamp of g. Note that in the worst case, $\min(f, g)$ may have up to $P(f) + P(g)$ interpolation points. See Figure 2.5 for an example. Note that the merging of timetable functions is also cheaper than the merging of functions used for road networks.

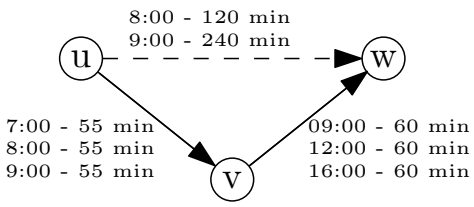


Figure 2.4: Time-dependent composition in public transportation networks. A function depicting the travel time from u to w via v yields less interpolation points than both functions constructed from.

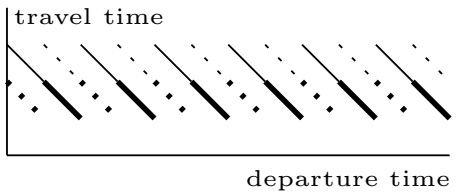
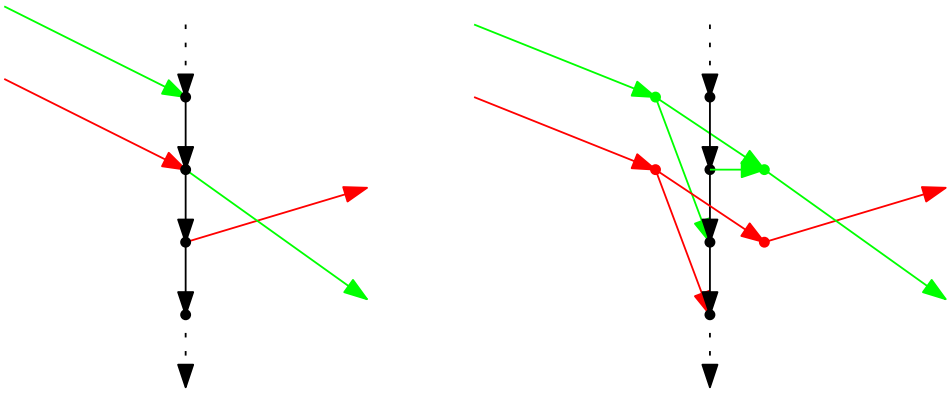


Figure 2.5: Time-dependent merging of two public transportation functions f and g, f is drawn solid, g dotted, the merged function is drawn thicker.

Time-Expanded Model [SWW99]. This model derives from the simple time-dependent model by "rolling out" time-dependency: A node is introduced for each arrival and departure *event* of train being connected among each other by time-independent edges. Each event at a station has a *timestamp*. The nodes at a station are ordered by timestamp. Two sequent events with timestamps t_i and t_j are connected by time-independent edges with a weight $t_j - t_i$. Finally, the last node with timestamp t_k is connected with the first one t_0. The weight is set to $(t_0 - t_k) + \Pi$ with Π being the period of the timetable. Figure 2.6(a) gives a small example.

Realistic Transfer Times [PSWZ04a]. Note that transfer times are again not covered correctly. For this reason, the model described above is called the *simple* time-expanded model. By introducing arrival, transfer, and departure nodes this can be remedied. Each arrival event is connected with the first transfer event that is reachable if transfer times are maintained. Unlike in the time-dependent model, switching to this realistic model increases the graph size only by a factor of ≈ 2.



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

Figure 2.6: Time-expanded railway graphs.

Discussion. For a long time, it seemed as if the time-expanded model cannot compete with the time-dependent model with respect to query performance: The increase in graph size yielded a greater search space. Recently, we were able to remedy this drawback by remodeling unimportant stations and pruning unnecessary connection during the query [DPW08]. Still, memory consumption of the time-dependent model is smaller.

At a glance, using speed-up techniques developed for static (time-independent) road networks on time-expanded graphs for timetable information seems promising. Since road networks seem to have similar properties as railway networks—both incorporate some kind of natural hierarchy and both are sparse—one might expect that speed-up techniques yield the same performance as on road networks. Unfortunately, our experimental study in Section 4.6 reveals that speed-up techniques perform significantely worse on time-expanded graphs than on road networks. Hence, the time-dependent model seems more promising.

Chapter 3

Basic Concepts

In this chapter, we revisit some of the techniques already mentioned in Chapter 1. More precisely, we explain Dijkstra's algorithm, Landmark-based routing, Arc-Flags, and the concept of contraction in more detail since those concepts are the most important foundations of the techniques introduced in Chapter 4. Note that the concepts described here only work in time-independent networks. Their augmentation to time-independent and multi-criteria scenarios can be found in Chapters 5 and 6.

3.1 Dijkstra

The classical algorithm for computing the shortest path from a given source to all other nodes in a directed graph with non-negative edge weights is due to Dijkstra [Dij59]. The algorithm maintains, for each node u, a label $distance[u]$ with the tentative distance from s to t. A priority queue Q contains all nodes that depict the current search horizon around s. At each step, the algorithm removes (or *settles*) the node u from Q with minimum distance to s. Then, all outgoing edges (u, v) of u are relaxed, i.e., we check whether $d(s, u) + len(u, v) < distance[v]$ holds. If it holds, a shorter path to v via u has been found. Hence, v is either inserted to the priority queue or its priority is decreased. The algorithm terminates if all nodes are settled. Algorithm 1 gives the algorithm in pseudo-code. The asymptotic running time of Dijkstra's algorithm depends on the choice of the priority queue. For general graphs, Fibonacci heaps [CLRS01] yield a worst-case time complexity of $O(m + n \log n)$, while using binary heaps [CLRS01] results in a worst-case time complexity of $O(m \log n)$. Since transportation networks are normally sparse, a

Algorithm 1: Dijkstra($G = (V, E)$,s)

```
1  Q.add(source, 0);                                          // priority queue
2  distance[];                          // array for distances of nodes from source s
3  while !Q.empty() do
4      u ← Q.deleteMin();                                     // settling node u
5      for all outgoing edges e from u do
6          v ← e.head;                                        // relaxing e
7          if distance[u] + e.weight() < distance[v] then
8              distance[v] ← distance[u] + e.weight()
9              key ← distance[v];                 // key depends on distance to s
10             Enqueue(v, key);                   // insert or decreaseKey
```

binary heap yields the same asymptotic time complexity in such networks. The advantage of binary over Fibonacci heaps is that the former are easier to implement and yield a lower computational overhead. Moreover, the search horizon of speed-up techniques is normally very low, so the impact of priority queues on the running times fades. In this work, we therefore use a binary heap as priority queue.

Point-to-Point Queries. If we are only interested in computing a shortest path from s to a given target t, we may stop the search as soon as t is settled by Dijkstra's algorithm. The priority of all nodes settled after t have a greater distance to s and since the graph contains only positive edge weights, the distance label of t cannot be improved further.

Shortest Path Tree. Running Dijkstra from a given source can be interpreted as growing a tree. If we store the predecessor of each node, we can easily identify the *shortest path tree* edges.

Dijkstra Rank. One may notice that Dijkstra settles the nodes of a graph in non-descending order of the distance to s. We call the position of a node u within this ordering the *Dijkstra rank* of u with respect to s.

3.2 A* Search Using Landmarks (ALT)

Next, we explain the known technique of A^* search [HNR68] in combination with landmarks, called ALT [GH05, GW05]. The search space of Dijkstra's algorithm can be visualized as a circle around the source. The idea of goal-directed or A^* search is to push the search towards the target. By adding a potential $\pi : V \to \mathbb{R}$ to the priority of each node, the order in which nodes are removed from the priority queue is altered. A 'good' potential lowers the priority of nodes that lie on a shortest path to the target. It is easy to see that A^* is equivalent to Dijkstra's algorithm on a graph with *reduced costs*, formally $len_\pi(u,v) = len(u,v) - \pi(u) + \pi(v)$. Since Dijkstra's algorithm works only on nonnegative edge costs, not all potentials are allowed. We call a potential π *feasible* if $len_\pi(u,v) \geq 0$ for all $(u,v) \in E$. The distance from each node v of G to the target t is the distance from v to t in the graph with reduced edge costs minus the potential of t plus the potential of v. So, if the potential $\pi(t)$ of the target t is zero, $\pi(v)$ provides a *lower bound* for the distance from v to the target t.

3.2.1 Preprocessing

There exist several techniques [SV86, WW05] to obtain feasible potentials using the layout of a graph. The ALT algorithm however, uses a small number of nodes—so called *landmarks*—and the triangle inequality to compute feasible potentials. Given a set $L \subseteq V$ of landmarks and distances $d(l,v), d(v,l)$ for all nodes $v \in V$ and landmarks $l \in L$, the following triangle inequalities hold:

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

Therefore, $\underline{d}(u,v) := \max_{l \in L} \max\{d(u,l) - d(v,l), d(l,v) - d(l,u)\}$ provides a feasible lower bound for the distance $d(u,v)$. See Figure 3.1 for an illustration. The quality of the lower bounds highly depends on the quality of the selected landmarks.

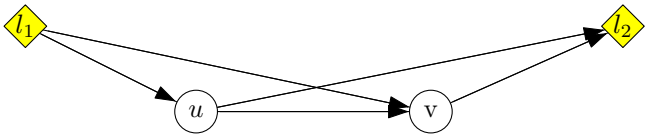


Figure 3.1: Triangle inequalities for landmarks. The landmarks are l_1 and l_2.

Landmark Selection. A crucial point in the success of a high speed-up when using ALT is the quality of landmarks. Since finding good landmarks is difficult, several heuristics [GH05, GW05] exist. We focus on the best known techniques: *avoid* and *maxCover*.

Avoid [GH05]. This heuristic tries to identify regions of the graph that are not well covered by the current landmark set S. Therefore, a shortest-path tree T_r is grown from a random node r. The *weight* of each node v is the difference between $d(v, r)$ and the lower bound $\underline{d}(v, r)$ obtained by the given landmarks. The *size* of a node v is defined by the sum of its weight and the size of its children in T_r. If the subtree of T_r rooted at v contains a landmark, the size of v is set to zero. Starting from the node with maximum size, T_r is traversed following the child with highest size. The leaf obtained by this traversal is added to S. In this strategy, the first root is picked uniformly at random. The following roots are picked with a probability proportional to the square of the distance to its nearest landmark.

MaxCover [GW05]. The main disadvantage of avoid is the starting phase of the heuristic. The first root is picked at random and the following landmarks are highly dependent on the starting landmark. MaxCover improves on this by first choosing a candidate set of landmarks (using avoid) that is about four times larger than needed. The landmarks actually used are selected from the candidates using several attempts with a local search routine. Each attempt starts with a random initial selection.

3.2.2 Query

The unidirectional ALT-query is a modified Dijkstra operating on the input graph, its pseudo-code is given in Algorithm 2. We observe that the only difference to plain Dijkstra is in Line 9, the key within the priority is not determined only by the distance to s but also by a lower bound of the distance to the target, given by the landmarks.

Bidirectional ALT. It turns out that undirectional ALT only provides mild speed-ups over Dijkstra's algorithm (cf. Section 4.5). The full potential of ALT is unleashed if applied bidirectional. At first glance, combining ALT and bidirectional search seems easy. Simply use a feasible potential π_f for the forward and a feasible potential π_b for the backward search. However, such an approach does not work due to the fact that the searches might work on different reduced costs, so that the shortest path might not have been found when both searches meet. This can only be guaranteed if π_f and π_b are *consistent*, meaning $len_{\pi_f}(u, v)$ in G is equal to $len_{\pi_b}(v, u)$ in the reverse graph. In this work, we use the variant of an average potential function [IHI+94] defined as $p_f(v) = (\pi_f(v) - \pi_b(v))/2$ for the forward and $p_b(v) = (\pi_b(v) - \pi_f(v))/2 = -p_f(v)$ for the backward search. By adding $\pi_b(t)/2$ to the forward and $\pi_f(s)/2$ to the backward search, p_f and p_b provide lower bounds to the target and source, respectively. Note that these potentials are feasible and consistent but provide worse lower bounds than the original ones.

Algorithm 2: ALT($G = (V, E)$, s, t)

```
1  Q.add(source, 0);                                          // priority queue
2  distance[];                    // array for distances of nodes from source s
3  while !Q.empty() or Q.minElement() ≠ t do
4      u ← Q.deleteMin();                                     // settling node u
5      for all outgoing edges e from u do
6          v ← e.head;                                        // relaxing e
7          if distance[u] + e.weight() < distance[v] then
8              distance[v] ← distance[u] + e.weight()
               // key depends on distance to s + lower bound to t
9              key ← distance[v] + d(v,t)
10             Enqueue(v, key)
11 return distance[t]
```

Optimizations. The bidirectional ALT implementation due to [GW05] uses several tuning techniques such as *active landmarks*, *pruning* and an enhanced stopping criterion. The search is stopped if the sum of minimum keys in the forward and the backward queue exceed $\mu + p_f(s)$, where μ represents the tentative shortest path length and is therefore an upper bound for the shortest path length from s to t. For each s-t query only two landmarks—one 'before' the source and one 'behind' the target—are initially used. At certain checkpoints a routine checks whether an additional landmark is added to the active set, with a maximal amount of six landmarks. Pruning means that before relaxing an arc (u, v) during the forward search we also check whether $d(s, u) + w(u, v) + \pi_f(v) < \mu$ holds. This technique may be applied to the backward search easily. Note that for pruning, the potential function need not be consistent.

Improved Efficiency. One downside of ALT seemed to be its low efficiency. In [GKW06], a reduction of factor 44 in search space only leads to a reduction in query times of factor 21. By storing landmark data more efficiently, this gap can be reduced. First, we sort the landmarks by ID in ascending order. The distances from and to a landmark are stored as one 64-bit integer for each node and landmark: The upper 32 bits refer to the 'to' distance and the lower to the 'from' distance. Thus, we initialize a 64-bit vector of size $|S| \cdot |V|$. Both distances of node number $i \in [0, |V| - 1]$ and landmark number $j \in [0, |S| - 1]$ are stored at position $|S| \cdot i + j$. As a consequence, when computing the potential for given node n, we only have one access to the main memory in most times.

3.2.3 Discusssion

The advantages of ALT are its easy adaption to dynamic scenarios (cf. Section 4.1), its robustness to the input (cf. Section 4.6) and its rather easy preprocessing algorithm. However, plain ALT cannot compete with hierarchical techniques in road networks, and even worse, ALT yields a high memory consumption: 16 landmarks already require an additional space of 128 Bytes per node. In Chapter 4, we will show how to remedy both disadvantages without losing the advantages.

3.3 Arc-Flags

The classic Arc-Flag approach, introduced in [Lau04, KMS05], first computes a partition $\mathcal{C}$ of the graph and then attaches a *label* to each edge e. A label contains, for each cell $C \in \mathcal{C}$, a flag $AF_C(e)$ which is `true` if a shortest path to at least one node in C starts with e. A modified Dijkstra—from now on called Arc-Flags Dijkstra—then only considers those edges for which the flag of the target node's cell is `true`. See Fig. 3.2 for an example. The big advantage of this approach is its easy query algorithm. Furthermore, we observed that for long-range queries in road networks, an Arc-Flags Dijkstra often is optimal in the sense that it *only* visits those edges that are on the shortest path. However, preprocessing is very extensive, either regarding preprocessing time or memory consumption.

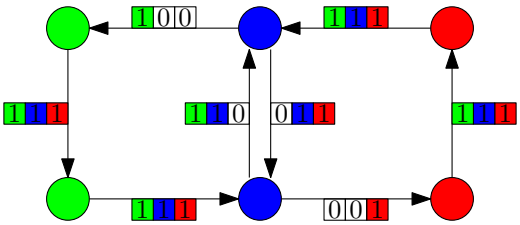


Figure 3.2: Example for Arc-Flags. The graph is partitioned into 3 regions.

3.3.1 Preprocessing

Preprocessing of Arc-Flags is divided into two parts. First, the graph is partitioned into k several cells. The second step then computes k flags for each edge.

Partition. The first approach for obtaining a partition based on a grid partition [Lau04]. It turns out that the performance of an Arc-Flags query heavily depends on the partition used. In order to achieve good speed-ups, several requirements have to be fulfilled: cells should be connected, the size of the cells should be balanced, and the number of boundary nodes has to be low. A systematical experimental study of the impact of partitions on Arc-Flags has been published in [MSS+06]. According to their work, the best results have been achieved if a METIS [Kar07] partition is applied.

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 [MS04] and SCOTCH [Pel07] for partitioning. The former produces connected cells but for the price of an even higher number of boundary nodes. SCOTCH has the lowest number of boundary cells, but connectivity of cells cannot be guaranteed. Due to this low number of boundary nodes, we use SCOTCH and improve the obtained partitioning by adding smaller pieces of disconnected cells to neighbor cells.

Setting Arc-Flags. The second step of preprocessing is the computation of arc-flags. Throughout the years, several approaches have been introduced (see e.g. [Lau04, KMS05, MSS+05, HKMS09, Lau09]). We here concentrate on two approaches which turned out to be the most efficient. For both approaches, own-cell flags of all edges not crossing borders have to be set to `true`. The own-cell flag of an edge (u, v) is the flag for the region of u and v. If u and v are in different cells, the edge does not have an own-cell flag.

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

Centralized Approach. The drawback of the first approach is that we have to grow $|B|$ shortest path trees yielding long preprocessing times for large transportation networks. [HKMS09] introduces a new approach to computing flags. A label-correcting algorithm (also called centralized tree) is performed for each cell C. The algorithm propergates labels of size $|B_C|$ through the network depicting the distances to all boundary nodes of the cell. The algorithm terminates if no label can be improved any more. Then, $AF_C((u, v))$ is set to true if $len(u, v) + d(v, b) = d(u, b)$ holds for at least one $b \in B_C$.

3.3.2 Query

A unidirectional Arc-Flags query is a modified Dijkstra operating on the input graph. For a random s–t query, it first determines the target cell T, and then relaxes only those edges with set flag for cell T. Note that compared to plain Dijkstra, an Arc-Flags query performs only one additional check. Algorithm 3 gives the query algorithm in pseudo code.

Algorithm 3: ARC-FLAGS DIJKSTRA($G = (V, E)$, s, t)

```
1  Q.add(source, 0);                                          // priority queue
2  distance[];                          // array for distances of nodes from source s
3  T ← region[t];                                             // target region
4  while !Q.empty() or Q.minElement() ≠ t do
5      u ← Q.deleteMin();                                     // settling node u
6      for all outgoing edges e from u do
7          if AF_T(e) = false then
8              continue;                                      // flag not set for region T
9          v ← e.head;                  // relaxing only edges with set flag for T
10         if distance[u] + e.weight() < distance[v] then
11             distance[v] ← distance[u] + e.weight()
12             key ← distance[v];                             // key depends on distance to s
13             ENQUEUE(v, key)
14 return distance[t]
```

3.3.3 Bidirectional Arc-Flags

Note that $AF_C(e)$ is true for almost all edges $e \in C$ due to 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, more and more edges become important when approaching the target cell (called the *coning effect*) and finally, all edges are considered as soon as the search enters the target cell. The coning effect can be weakened by a bidirectional query.

Switching from uni- to bidirectional Arc-Flags is easy but preprocessing effort is doubled. Additional *backward* flags need to be computed by growing reversed shortest path trees (or reversed centralized tree) in G. The bidirectional query algorithm is a modified bidirectional Dijkstra. The forward search is modified like a unidirectional Arc-Flags query. During initialization however, region S of source node s is determined as well. Then, the backward search only considers those incoming edges with a set backward flag for cell S.

3.3.4 Multi-Level Arc-Flags

While the coning effect can be weakened by a bidirectional approach, the problem of inner-cell queries persists also for bidirectional search. An approach to remedy this drawback is introduced in [MSS+06]: A second layer of arc-flags is computed for each cell. Therefore, each cell is again partitioned into several subcells and arc-flags are computed for each. A multi-level arc-flags query then first uses the flags on the topmost level and as soon as the query enters the target's cell on the topmost level, the low-level arc-flags are used for pruning.

Preprocessing in a time-independent scenario is done as follows. Arc-flags on the upper level are computed as described above. For the lower flags, grow a shortest path for all boundary nodes b on the lower level. Stop the growth as soon as all nodes in the supercell of C are settled. Then, we set a low-level arc-flag to `true` if the edge is a tree edge of at least one shortest path tree. Note that this approach can be extended to a multi-level approach in a straightforward manner. Also note that multi-level Arc-Flags can be applied bidirectionally as well.

3.3.5 Discusssion

The advantages of Arc-Flags is the easy concept combined with exceptional query performance: Preprocessing is based on Dijkstra-searches and the query algorithm performs only one additional check compared to plain Dijkstra. Stunningly, bidirectional Arc-Flags long-range queries are often optimal—at least in road networks—in that sense that *only* shortest path edges are relaxed.

However, the most crucial drawback of Arc-Flags is its time consuming preprocessing effort. Even the most advanced technique, i.e., the centralized approach, needs more than 17 hours to preprocess a continental-sized road network. Still, due to its superior undirectional query performance, Arc-Flags seemed to a good starting point for our work on time-independent (Chapter 4), time-dependent (Chapter 5), and multi-criteria (Chapter 6) shortest path computations.

3.4 Contraction

One reason for the success of hierarchical speed-up techniques like Highway-Hierarchies, Contraction Hierarchies, and the RE-algorithm is the iterative contraction of the input. It turns out that—at least in road networks—it is often sufficient to perform extensive preprocessing only on a small subgraph of the input, called the *core*. One main contribution of this work is the incorporation of such an approach to goal-directed methods as described above.

In the following we show how we extract a core of a given input graph G. The key idea is first to perform a node-reduction step that iteratively *bypasses* nodes until no node is *bypassable* any more. Shortcut edges are added to preserve distances between non-bypassed nodes, called *core* nodes. Our node-reduction potentially adds unneeded shortcuts which are identified and removed by our edge-reduction routine. In the following we explain each routine separately.

3.4.1 Node-Reduction

The number of nodes is reduced by iteratively *bypassing* nodes until no node is *bypassable* any more. To bypass a node x we first remove x, its incoming edges I and its outgoing edges O from the graph. Then, for each $u \in tails(I)$ and for each $v \in heads(I) \setminus \{u\}$ we introduce a new edge of the length $len(u, x) + len(x, v)$. If there already is an edge 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 x was bypassed, i.e., existing edges connecting nodes in $tails(I)$ with nodes in $heads(O)$ do not contribute to $\#shortcut$. Then we say a node is bypassable iff the *bypass criterion* $\#shortcut \leq c \cdot (\deg_{in}(x) + \deg_{out}(x))$ is fulfilled, where c is a tunable *contraction parameter*.

A node being bypassed influences the degree of its 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 x within the heap is $h(x) \cdot \#shortcut/(\deg_{in}(x) + \deg_{out}(x))$ where $h(x)$ is the hop number of the hop-maximal shortcut that would be added if x 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 h. We say that the nodes that have been bypassed belong to the *component*, while the remaining nodes are called *core-nodes*.

Corollary 3.1. *Node-reduction preserves distances between core nodes.*

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

3.4.2 Edge-Reduction

Note that our node-reduction routine potentially adds shortcuts not needed for keeping the distances in the core correct. See Figure 3.3 for an example. Hence, we perform an edge-reduction directly after node-reduction, similar to [SWW99]. We grow a shortest-path tree from each node u of the core. We stop the growth as soon as all neighbors t of u have been settled. Then we check for all neighbors t whether u is the predecessor of t in the grown partial shortest path tree. If u is not the predecessor, we can remove (u, t) from the graph because the shortest path from u to t does not include (u, t). In order to remove as many edges as possible we favor paths with more hops over those with few hops. In order to limit the running time of this procedure, we restrict the number of priority-queue removals to 10 000. Hence, we may leave some unneeded edges in the graph.

Corollary 3.2. *Edge-reduction preserves distances between core nodes.*

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

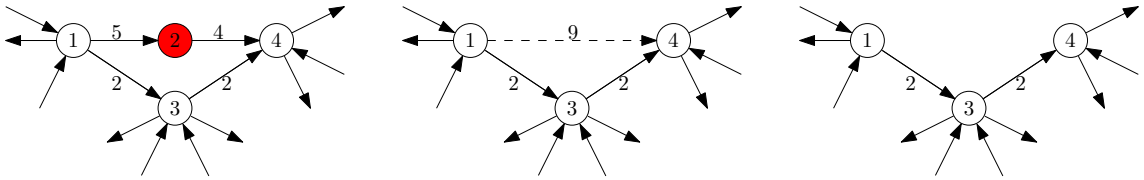


Figure 3.3: Example for contraction. The figure on the left depicts the input, edge labels indicate the weight of the edge. We contract, i.e., remove, node 2 and add an shortcut from node 1 to 4 with weight 9 (middle). However, the shortest path from 1 to 4 is via node 3 with length 4. Hence, we can safely remove the shortcut (1,4) from the core in order to preserve distances between core nodes. The resulting graph is shown on the right.

3.4.3 Comparison to Contraction Hierarchies

Contraction was introduced as an ingredient for Highway Hierarchies (HH) and was later adapted to Reach (RE). In both cases, node-reduction is similar as described above, while edge-reduction is somehow more complicated in both cases. For HH, highway-edges have to be identified, while for RE, edge-reduction is based on a rather complicated reach computation. It turned out that the simple edge-reduction routine from above is sufficient to prune the graph in order to efficiently construct a hierarchy in a road network [SS07]. Starting from this observation, Contraction Hierarchies [GSSD08] has been developed. Basically, a Contraction Hierarchy is built from an input by performing n node-reduction and edge-reduction steps, similar to the ones described above. Each node-reduction step contracts exactly one node u, resulting in a very limited edge-reduction routine as unneeded shortcuts may only be added between neighbors of u. However, CH tends to highly depend on the input and on chosen parameters. Hence, we refer to the routines described above if we construct a core.

Chapter 4

Time-Independent Route Planning

We start our work on route planning in time-independent networks. First, we show how landmarks can be used in dynamic scenarios and how the most crucial drawbacks of ALT, memory consumption and performance, can be remedied without losing the advantages of ALT. Moreover, keeping augmentation to time-dependency in mind, we develop a fast *unidirectional* speed-up technique competing with bidirectional approaches. With such a technique at hand, adaption to augmented scenarios is easier than for bidirectional approaches. Furthermore, we enhance the first natural choice for augmentation, i.e., landmarks, such that the performance gap to other techniques is almost closed. An extensive study confirms that our new combinations and techniques can compete with known approaches. Besides that, we further accelerate the fastest hierarchical speed-up techniques, i.e., Contraction Hierarchies and Transit-Node Routing, by adding goal-direction via arc-flags. It turns out that it is sufficient to compute arc-flags only for a small subset of important nodes and edges. These combinations yield excellent query times in road networks: Speed-ups are up to 3 million over Dijkstra's algorithm.

However, fast algorithms are also needed for other applications than route planning in road networks. One might expect that all speed-up techniques can simply be used in any other application, yet several problems arise: some inputs do not incorporate a hidden hierarchy, bidirectional search is forbidden, and in railway networks, delays occur frequently. In an experimental study on inputs other than road networks, it turns out that our combinations are very robust to the input.

Overview. This chapter is organized as follows. In Section 4.1 we shortly present how landmarks work in dynamic scenarios. The key observation is that ALT can be used in dynamic scenarios *without* any adaption. This is very helpful for time-dependent scenarios. However, it turns out that ALT cannot compete with other time-independent speed-up techniques. Hence, we introduce contraction to ALT in Section 4.2, resulting in a powerful and robust speed-up technique, called CALT. While CALT can be adapted to dynamic scenarios easily, the known problem of bidirectional search persists for this approach. Thus, we introduce SHARC, which has the big advantage that a unidirectional variant is as fast as most other bidirectional approaches. Since SHARC is augmented to time-dependent and multi-criteria scenarios in Chapters 5 and 6, we present this approach on a very detailed level. In Section 4.4, we show how to accelerate the fastest known hierarchical approaches, i.e., Contraction Hierarchies and Transit Node Routing, by computing additional arc-flags on a small core. This yields the fastest known techniques for routing many time-independent scenarios, including road networks. Our experimental evaluation in Section 4.5 also reveals that SHARC indeed can compete with bidirectional approaches. Other inputs are evaluated in Section 4.6 leading to the insight that some techniques are robust to the input than others. We conclude our work on time-independent route planning with a short summary and possible furture work in Section 4.7.

4.1 Dynamic ALT

In this section, we discuss how the preprocessing of the ALT algorithm can be updated efficiently. Furthermore, we discuss a variant where the preprocessing has to be updated only very few times. However, this approach may lead to a loss in performance.

4.1.1 Updating the Preprocessing

The preprocessing of ALT consists of two steps: the landmark selection and calculating the distance labels. As the selection of landmarks are heuristics, we settle for *static* landmarks, i.e., we do not reposition landmarks if the graph is altered. The update of the distance labels can be realized by dynamic shortest path trees. For each landmark, we store two trees: one for the forward edges, one for the backward edges. Whenever an edge is altered we update the tree structure including the distance labels of the nodes. In the following we discuss a memory efficient implementation of dynamic shortest path trees. The construction of a tree can be done by running a complete Dijkstra from the root of the tree.

Updating Shortest Path Trees. In [NST00] the update of a shortest path tree is discussed. The approach is based on a modified Dijkstra, trying to identify the regions that have to be updated after a change in edge weight. Therefore, a tree data structure is used in order to retrieve all successors and the parent of a node quickly. We briefly explain the routine for weight changes in a tree T, for details we refer to [NST00, FMSN00]. Moreover, we reduce deletions and insertions to weight changes.

Increase of an edge weight. In a first step, we check whether the updated edge e is a tree edge. If not, we are done as an increment of this edge cannot transform it into a tree edge. If it is a tree edge, we update the distance label of the head v by the weight change Δ. All descendants of t are updated by Δ and are added to a nodeset U. For all edges having a head in U, we have to check whether a shorter path via that edge exists yielding a lower distance label than after the update. If this is true for a node $u \in U$, we enqueue it to a priority Queue Q. For all nodes in Q we check whether an outgoing edge yields a lower distance label for another node. If this is true, we enqueue this node to Q. The routine stops if Q is empty.

Decrease of an edge weight. If an edge e is decreased by Δ, we check whether the decrease yields a lower distance label for the head v of e. If this is true, all descendants of t are decreased by Δ and added to U, including t itself. Similar to increments, we have to check all outgoing edges of U, whether the decreased label of a node $u \in U$ yields a shorter path to a neighbor node v of u. If this is true, we enqueue v to a priority Queue Q. The elements of Q are processed the same way they are processed for edge weight increments.

The insertion or deletion of an edge can be handled by weight changes. Deleting an edge may be interpreted as increasing the weight to infinity, while inserting an edge may be regarded as setting the weight from infinity to the desired value. Deleting a node can be handled by deleting all incoming and outgoing edges, while inserting a node n can be handled by edge insertions and deletions. Setting the label is straightforward then.

Data Structure. Analyzing our update routines from above, we realize that we need two operations implemented efficiently: Accessing all successors of a node u and retrieving its parent p. As transportation networks are normally sparse we do not need to store any additional information to implement these operations. The successors of u can be determined by checking for each head v of all outgoing edges e whether $d(u)+len(e) = d(v)$ holds. If it holds, v can be interpreted as successor of u. Analogously, we are able to determine the parent of a node u: Iterate all tails v of the incoming edges e of u. If $d(v) + len(e) = d(u)$ holds, v is the parent of i. This implementation allows to iterate all successors of u and accessing the parent of u in $O(\deg_{in}(u) + \deg_{out}(u))$. Note that we may obtain a different tree structure than rerunning a complete Dijkstra, but as we are only interested in distance labels, this approach is sufficient for the correctness of ALT.

The advantage of this approach is memory consumption. Keeping all distance labels for 16 landmarks on the road network of Western Europe in memory already requires about 2.2 GB of RAM (32 trees with 18 million nodes, 32 bit per node). Every additional pointer would need an additional amount of 2.2 GB. Thus, more advanced tree structures (1, 2, or 4 pointers) lead to an overhead that does not seem worth the effort.

4.1.2 Two Variants of the Dynamic ALT Algorithm

Eager Dynamic ALT. In the previous section we explained how to update the preprocessed data of ALT. Thus, we could use the update routine whenever the graph is altered. In the following, we call this variant of the dynamic ALT the *eager dynamic* version.

Lazy Dynamic ALT. However, analyzing our dynamic scenarios from Section 2.4 and the ALT algorithm from Section 3.2 we observe two important facts. On the one hand, ALT-queries only lose correctness if the potential of an edge results in a negative edge cost in the reduced graph. This can only happen if the cost of the edge drops below the value during preprocessing. On the other hand, for the most common update type in road networks—traffic jams—edge weights may increase and decrease but do not drop below the initial value of empty roads. Thus, a potential computed on the graph without any traffic stays feasible for those kinds of updates even when not updating the distances from

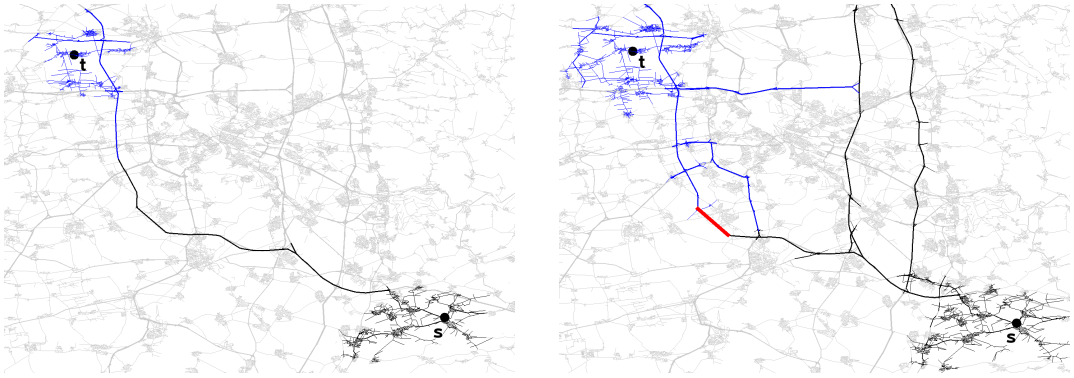


Figure 4.1: Search space of dynamic ALT for a fixed *s*-*t* query before and after jamming a motorway edge (marked in bold red). For the query with jammed motorway, the search space develops exactly like the normal query. At the point the normal query would stop, the perturbed query continues because the stopping criterion is not fulfilled yet.

and to all landmarks. Due to this observation, we may do the preprocessing for empty roads and use the obtained potentials even though an edge is perturbed. In [GH05], this idea was stated to be semi-dynamic, allowing only increases in edge weights. Nevertheless, as our update routine does not need any additional information, we are able to handle all kinds of updates. Our *lazy dynamic* variant of ALT leaves the preprocessing untouched unless the cost of an edge drops below its initial value.

This approach may lead to an increase in search space. If an edge e on the shortest path is increased without updating the preprocessing, the weight of e is also increased in G', the graph with reduced costs. Thus, the length of the shortest path increases in G'. So, the search stops later because more nodes are inserted in the priority queue. See Figure 4.1 for an example. However, as long as the edges are not on the shortest path of a requested query the search space does not increase. More precisely, the search space may even decrease because nodes 'behind' the updated edge are inserted later into the priority queue.

4.2 Core-Based Routing

As already discussed in Section 3.2.3, ALT suffers from two major drawbacks. Space consumption is rather high and—even more important—ALT cannot compete with hierarchical approaches—concerning query performance—in transportation networks. In this section, we show how to remedy both drawbacks without violating the advantages of pure ALT, i.e., easy adaption to dynamic scenarios and robustness to the input. The key idea is to perform an initial contraction step prior to ALT preprocessing. Landmarks are then chosen from the core and landmark distances are also only stored for core nodes. This yields a 2-phase query. During the first phase, a plain bidirectional Dijkstra is performed until the core is reached. Within the core, bidirectional ALT is applied. We call the resulting speed-up techniques *Core*-ALT (CALT). In the following we explain this approach in more detail. Therefore, we first introduce the generic concept of core-based routing (without ALT) and then focus on the combination with ALT.

4.2.1 Generic Approach

Preprocessing. At first, the input graph $G = (V, E)$ is contracted to a graph $G_C = (V_C, E_C)$, called the *core*. The key idea of *core-based routing* is not to use G as input for preprocessing but to use G_C instead. As a result, preprocessing of most techniques can be accelerated as the input can be shrunk. However, sophisticated methods like Highway Hierarchies, REAL, or SHARC already use contraction during preprocessing. Hence, this advantage especially holds for goal-directed techniques like ALT or Arc-Flags. After preprocessing the core, we store the preprocessed data and merge the core and the normal graph to a full graph $G_F = (V, E_F = E \cup E_C)$. Moreover, we mark the core-nodes with a flag.

Query. The s-t query is a modified bidirectional Dijkstra, consisting of two phases and performed on G_F. During phase 1, we run a bidirectional Dijkstra rooted at s and t *not* relaxing edges belonging to the core. We add each core node, called *entrance point*, settled by the forward search to a set S (T for the backward search). The first phase terminates if one of the following two conditions hold: (1) either both priority queues are empty or

(2) the distance to the closest entry points of s and t is larger than the length of the tentative shortest path. If case (2) holds, the whole query terminates. The second phase is initialized by refilling the queues with the nodes belonging to S and T. As key we use the distances computed during phase 1. Afterwards, we execute the query-algorithm of the applied speed-up technique which terminates according to its stopping condition. Note that when not using any speed-up technique for the second phase, we end up in a 2-level Highway-Node Routing setup (cf. [Sch08]).

Theorem 4.1. *Core-Based Routing is correct.*

Proof. Let $P = (s, u_1, \ldots, u_k, t)$ be an arbitrary shortest path in G. If no node on P is part of the core in G_F, core-based routing is correct since we find the path during phase 1. If only one node on P is part of the core, we also find the path during phase 1. Now, let more than one node on P be part of the core. Let u_i be the first and u_j be the last core node on P. During phase one, we obtain the subpaths from s to u_i and from u_j to t. Due to the fact that distances within the core are preserved by our contraction routine (Corollaries 3.1 and 3.2), we know that a path between u_i and u_j in G_C exists with the same length as the corresponding subpath of P. Hence, there exists a path in G_F with equal length that is found by core-based routing. □

4.2.2 CALT

Although we could use *any* speed-up techniques to instantiate our core-based approach we focus on a variant based on ALT due to the following reasons. First of all, ALT works well in *dynamic* scenarios. As contraction seems easy to augment to dynamic scenarios, it turns out that CALT (Core-ALT) also works well in dynamic scenarios (cf. Section 5.3). Second, pure ALT is a very robust technique with respect to the input and finally, ALT suffers from the critical drawback of high memory consumption—we have to store two distances per node and landmark—which can be reduced by switching to CALT. On top of the preprocessing of the generic approach, we compute landmarks on the core and store the distances to and from the landmarks for all core nodes. The second phase of core-based routing is replaced by ALT.

Proxy Nodes. Note that the ALT query requires lower bounds to s and t from every node within the core but both s and t need not be part of the core. In order to perform correct queries anyway, we adapt the ideas from [GKW07, DSSW09b] to overcome this problem. Let t'—called the *proxy node* of t—be the core node with minimum $d(t, t')$ and let l_1 and l_2 be two arbitrary landmarks $\in L \subset V_C$. Then the following equations hold for all $u \in V$. See Figure 4.2 for illustration.

$$\begin{aligned} d(u, t') &\leq d(u, t) + d(t, t') \\ d(u, l_2) &\leq d(u, t') + d(t', l_2) \\ d(l_1, t') &\leq d(l_1, u) + d(u, t') \end{aligned}$$

Hence,

$$\underline{d}(u, t) := \max_{l \in L} \max\{d(u, l) - d(t', l) - d(t, t'), d(l, t') - d(l, u) - d(t, t')\} \tag{4.1}$$

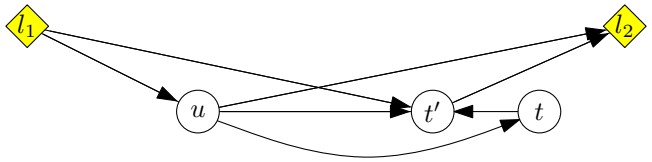


Figure 4.2: Proxy nodes for CALT.

provides a feasible lower bound for the distance $d(u,t)$. Analogously, we obtain $\underline{d}(s,u) := \max_{l \in L} \max\{d(s',l) - d(u,l) - d(s',s), d(l,u) - d(l,s') - d(s',s)\}$ as feasible lower bounds for the distance $d(s,u)$ with $s' \in V_C$ being the node with minimum $d(s',s)$.

We compute these proxy nodes of s and t for a given s–t query during the initialization phase of the second phase of the query by running Dijkstra-queries. Note that for CALT, the quality of the lower bounds not only depends on the quality of the selected landmarks but also on $d(t,t')$ and $d(s',s)$.

Improved Locality. We increase—similar to [GKW07]—cache efficiency of G_F by reordering nodes. As most of the query is performed on the core, we store the core nodes followed by the non-core nodes. As a consequence, the number of cache misses is reduced yielding lower query times. Furthermore, this eases accessing landmark distances since we can simply use an array with $|L| \cdot |V_C|$ 64-bit entries for storing these distances (cf. Section 3.2).

Corollary 4.2. CALT *is correct.*

Proof. According to Theorem 4.1, plain core-based routing is correct. Moreover, potentials as obtained from Equation (4.1) are feasible. Hence, applying ALT during phase 2 does not violate Theorem 4.1. □

4.3 SHARC

Due to its easy adaptability to dynamic scenarios, CALT seems to be a good candidate for augmenting to time-dependent scenarios. However, CALT can only compete with other time-independent techniques if applied bidirectional. As discussed earlier, this is problematic. For this reason, we here integrate contraction into Arc-Flags, yielding a fast and robust approach for *unidirectional* routing, which we call SHARC. However, our approach can also be used as a bidirectional algorithm yielding even faster query times. See Figure 4.3 for an example of a typical search space of uni- and bidirectional SHARC.

4.3.1 Preprocessing

Preprocessing of SHARC occurs in three phases: 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

Figure 4.3: 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.

output-graph, refine arc-flags of edges removed during contraction and finally reattach the 1-shell nodes removed at the beginning. Figure 4.4 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 straightforward.

Initialization

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.

Multi-Level Partition. As already mentioned in Section 3.3, the classic Arc-Flag method heavily depends on the partition used. The same holds for SHARC. In this work, we use a locally optimized partition obtained from SCOTCH [Pel07]. The number of levels L and the number of cells per level are tuning-parameters.

Iteration. Next, an iterative process starts. During each iteration step, we contract the graph, set arc-flags for removed edges and compute multi-level arc-flags.

Cell-Aware Contraction At the beginning of each iteration step we perform a node- and edge-reduction step according to the routines described in Chapter 3.4. However, in order to guarantee correctness, we use *cell-aware* node-reduction, i.e., a node u is never marked bypassable if any of its neighboring nodes is *not* in the same cell as u.

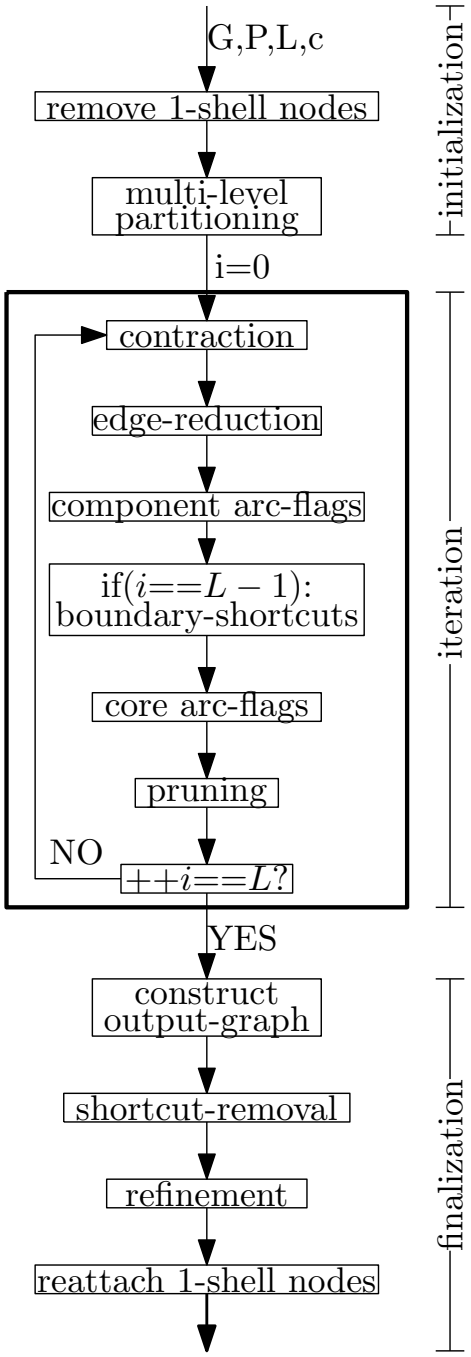


Figure 4.4: Schematic representation of SHARC-preprocessing. Input parameters are the partition parameters P, the number of levels L, and the contraction parameter c. During initialization, we remove the 1-shell nodes and partition the graph. Afterwards, an iterative process starts which contracts the graph, reduces edges, 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, remove unimportant shortcuts, and reattach the 1-shell nodes to the graph.

Component Arc-Flags. Our query algorithm is executed on the original graph enhanced by shortcuts added during the contraction phase. Thus, we have to assign arc-flags to each edge we remove during the contraction phase. One option would be to set every flag to true. However, we can do better. First of all, we keep all arc-flags that already have been computed for lower levels. We set the arc-flags of the current and all higher levels depending on the tail u of the deleted edge. If u is a core node, we only set the own-cell flag to true (and others to false) because this edge can only be relevant for a query targeting a node in this cell. If u belongs to the component, all arc-flags are set to true as a query has to leave the component in order to reach a node outside this cell. Finally, shortcuts get their own-cell flag *fixed* to false as relaxing shortcuts when the target cell is reached yields no speed-up. See Figure 4.5 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.

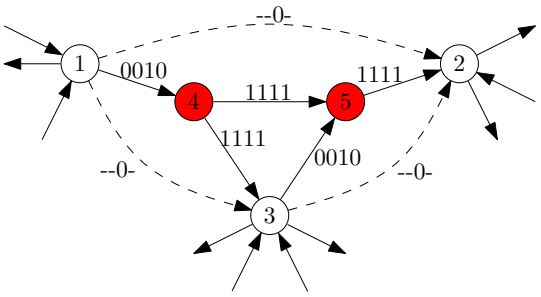


Figure 4.5: Example for assigning arc-flags during contraction for a partition having four cells. All nodes are in cell 3. The red nodes (4 and 5) are removed, the dashed shortcuts are added by the contraction. Arc-flags (edge labels) are indicated by a 1 for true and 0 for false. The edges directing into the component get *only* their own-cell flag set true. All edges in and out of the component get full flags. The added shortcuts get their own-cell flags fixed to false.

Core Arc-Flags. After the contraction phase and assigning arc-flags to removed edges, we compute the arc-flags of the core-edges of the current level i. As described in Section 3.3, we grow, for each cell C, one centralized shortest path tree on the reverse graph starting from every boundary node $b_C \in B_C$ of C. We stop growing the tree as soon as all nodes of C's supercell have a distance to each $b \in B_C$ greater than the smallest key in the priority queue used by the centralized shortest path tree algorithm. For any edge e that is in the supercell of C and that lies on a shortest path to at least one $b \in B_C$, we set $AF^i_C(e) = \text{true}$.

Finalization. After iteration, we assemble the *output graph* of the preprocessing consisting 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.

4.3.2 Query

Basically, our query is a multi-level Arc-Flags Dijkstra (cf. Section 3.3). The query is a modified Dijkstra that operates on the output graph. The modifications are as follows: When settling a node u, we compute the lowest level i on which u and the target node t are in the same supercell. When relaxing the edges outgoing from u, we consider only

those edges having a set arc-flag on level i for the corresponding cell of t. We want to point out that the SHARC query, compared to plain Dijkstra, only needs to perform two additional operations: computing the common level of the current node and the target and the arc-flags evaluation. Thus, our query is very efficient with a much smaller overhead compared to other hierarchical approaches.

Improved Locality. Like for CALT, we increase cache efficiency of the output graph by reordering nodes according to the level they have been removed at from the graph. As a consequence, the number of cache misses is reduced yielding lower query times.

Path-Expansion. During our experimental evaluation, we observed that many nodes have only one outgoing edge for which the arc-flag of the corresponding target is set to true for the current query (cf. Figure 4.3). We call this property the *no-choice* property and the specific edge with the flag set to true the *no-choice* edge. With this observation at hand, we can use the following optimization to speed-up the query. Whenever we insert a node u into the priority queue fulfilling the no-choice property, we skip this node and insert the head v of the no-choice edge into the queue. If the no-choice property also holds for node v, we also skip node v. We skip nodes until we either insert t, the target of the query, or insert a node for which the no-choice property does not hold. Note that *path-expansion* is especially helpful for our stripped variant of SHARC. Here, path-expansion partly remedies the drawback of lacking shortcuts.

Multi-Metric Query. In [BDW07b], we observed that the shortest path structure of a graph—as long as edge weights somehow correspond to travel times—hardly changes when we switch from one metric to another. Thus, one might expect that arc-flags are similar to each other for these metrics. We exploit this observation for our *multi-metric* variant of SHARC. During preprocessing, we compute arc-flags for all metrics and at the end we store only *one* arc-flag per edge by setting a flag true as soon as the flag is true for at least one metric. An important precondition for multi-metric SHARC is that we use the same partition for each metric. Note that the structure of the core computed by our contraction routine is independent of the applied metric.

Outputting Shortest Paths 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 [DSSW09b], as our contraction works similar to Highway Hierarchies: for each added shortcut, we store the IDs of the edges it represents.

4.3.3 Correctness

Before proving the correctness of SHARC, we first have to show that the following Lemma 4.3 holds. We denote by G_i the graph after iteration step i, $i = 1, \ldots, L-1$. By G_0 we denote the graph directly before iteration step 1 starts. The level $l(u)$ of a node u is defined to be the integer i such that u is contained in G_{i-1} but not in G_i. We further define the level of a node contained in G_{L-1} to be L.

Lemma 4.3. *Given arbitrary nodes s and t in G_0, for which there is a path from s to t in G_0. At each step i of the SHARC-preprocessing there exists a shortest s-t-path $P = (v_1, \ldots, v_{j_1}; u_1, \ldots, u_{j_2}; w_1, \ldots, w_{j_3})$, $j_1, j_2, j_3 \in \mathbb{N}_0$, in $\bigcup_{k=0}^{i} G_k$, such that*

- $l(v_1), \ldots, l(v_{j_1}), l(w_1), \ldots, l(w_{j_3}) \leq i$,
- $l(u_1), \ldots, l(u_{j_2}) \geq i+1$
- $\mathbb{c}_i(u_{j_2}) = \mathbb{c}_i(t)$
- *for each edge e of P, the arc-flags assigned to e until iteration 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.

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

Proof. We prove Lemma 4.3 by induction on the iteration steps. Since pure Arc-Flags is correct, the claim holds trivially for $i = 0$. The inductive step works as follows: Assume the claim holds for step i. Given arbitrary nodes s and t, for which there is a path from s to t in G_0. We denote by $P = (v_1, \ldots, v_{j_1}; u_1, \ldots, u_{j_2}; w_1, \ldots, w_{j_3})$ the s-t-path according to the lemma for step i.

The iteration step $i+1$ consists of the contraction phase and the arc-flag computation. Since the latter does not violate the lemma, it remains to be shown that after each contraction step, Lemma 4.3 holds.

There exists a maximal path $(u_{\ell_1}, u_{\ell_2}, \ldots, u_{\ell_d})$ with $1 \leq \ell_1, \leq \ldots \leq \ell_d \leq k$ for which

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

By the construction of the contraction routine we know

- $(u_{\ell_1}, u_{\ell_2}, \ldots, u_{\ell_d})$ is also a shortest path, see Corollary 3.1.
- u_{ℓ_d} is in the same component as u_k in all levels greater than i (because of cell aware contraction)
- the deleted edges in $(u_1, \ldots, u_{\ell_1 - 1})$ either already have their arc-flags for the path P assigned. Then the arc-flags are correct because of the induction hypothesis. Otherwise, We know that the nodes $u_1, \ldots, u_{\ell_1 - 1}$ are in the component. Hence, all arc-flags for all higher levels are assigned true.
- the deleted edges in $(u_{\ell_d+1}, \ldots, u_k)$ either already have their arc-flags for the path p^{τ}_{st} assigned, then arc-flags are correct because of the induction hypothesis.
- otherwise, by cell-aware contraction we know that $u_{\ell_d+1}, \ldots, u_k$ are in the same component as t for all levels at least i. As the own-cell flag is always set to true for deleted edges the path stays valid.

According to Corollaries 3.1 and 3.2, distances are preserved during preprocessing. Hence, for arbitrary i, $0 \leq i \leq L-1$ a shortest path in G_i is also a shortest path in $\bigcup_{k=0}^{L-1} G_k$. So, the path $\hat{P} = (v_1, \ldots, v_{j_1}, u_1, \ldots, u_{\ell_1-1};\ u_{\ell_1}, u_{\ell_2}, \ldots, u_{\ell_d};\ u_{\ell_d+1}, \ldots, u_k, w_1, \ldots, w_{j_3})$ fullfills all claims of the lemma for iteration step $i+1$. □

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

Theorem 4.4. *The distances computed by SHARC are correct.*

Proof. Lemma 4.3 holds during all phases of all iteration steps of SHARC-preprocessing. So, together with the correctness of multi-level Arc-Flags, the preprocessing algorithm is correct. □

4.3.4 Optimizations

Although SHARC as described above already yields a low preprocessing effort combined with good query performance, we use some optimization techniques to reduce both preprocessing time and space consumption of the preprocessed data.

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.

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 *exit nodes* $\vec{N}(u)$ of u. With this information we refine the arc-flags of all edges outgoing from u. First, we set all flags—except the own-cell flags—of all levels

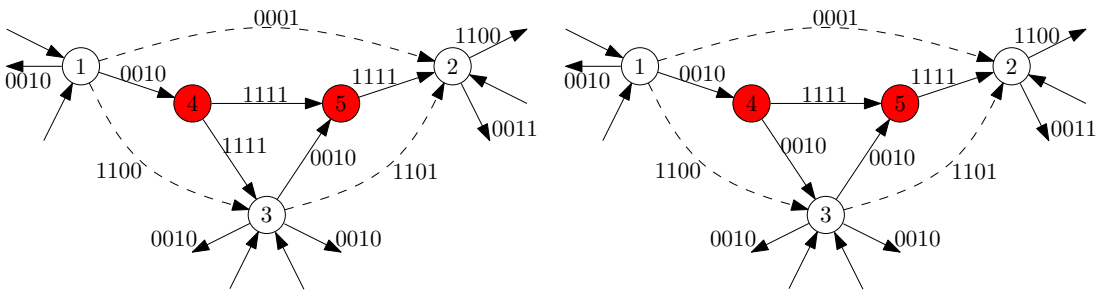


Figure 4.6: Example for refining the arc-flags of outgoing edges from node 4. The figure in the left shows the graph from Figure 4.5 after the last iteration step. The figure on the right shows the result of our refinement routine starting at node 4.

$\geq l(u)$ for all outgoing edges from u to false. Next, we assign exit nodes to outgoing edges from u. Starting at an exit node n_E we follow the predecessor in T until we finally end up in a node x whose predecessor is u. The edge (u, x) now inherits the flags from n_E. Every edge outgoing from n_E whose head v is *not* an exit node of u and *not* in a level $< l(u)$ propagates all true flags of all levels $\geq l(u)$ to (u, x). In order to propagate flags from higher to lower levels we perform our propagation-routine in $L - 1$ refinement steps, starting at level $L - 1$ and in descending order. Figure 4.6 gives an example. Note that during refinement step i we only refine arc-flags of edges outgoing from nodes belonging to level i.

Lemma 4.5. *Refinement of arc-flags is correct.*

Proof. Recall that the own-cell flag does not get altered by the refinement routine. Hence, we only have to consider flags for other cells. Assume we perform the propagation routine at a level l to a node u in level l.

A shortest 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 4.3 we know that there must be an (arc-flag valid) shortest s-t-path P for which the sequence of the levels of the nodes first is monotonically ascending and then monotonically descending. In fact, to cross a border of the current cell at level l, at least two level $> l$ nodes are on P. We consider the first level $> l$ node u_1 on P. This must be an exit node of u. The node u_2 after u_1 on P is covered and therefore no exit node. Furthermore it is of level $> l$. Hence, the flags of the edge (u_1, u_2) are propagated to the first edge on P and thus, Lemma 4.3 holds also after refinement which proves that the refinement phase is correct. □

Shortcut-Removal. Note that the insertion of shortcuts is one of the main reasons why the output graph is larger than the input. Hence, we try to remove shortcuts as the very last step of preprocessing. The routine works as follows. For each added shortcut (u, v) we analyze the shortest paths it represents. If all nodes on these shortest paths have less than 2 outgoing edges, we remove (u, v) from the graph and all edges being part of the shortest paths additionally inherit the arc-flags from (u, v). An example is given in Figure 4.7.

Stripped SHARC. Note that we could use our shortcut-removal routine to remove *all* shortcuts we added during preprocessing. Our output graph then equals the original input, with additional region information and arc-flags for each edge. As a result, such

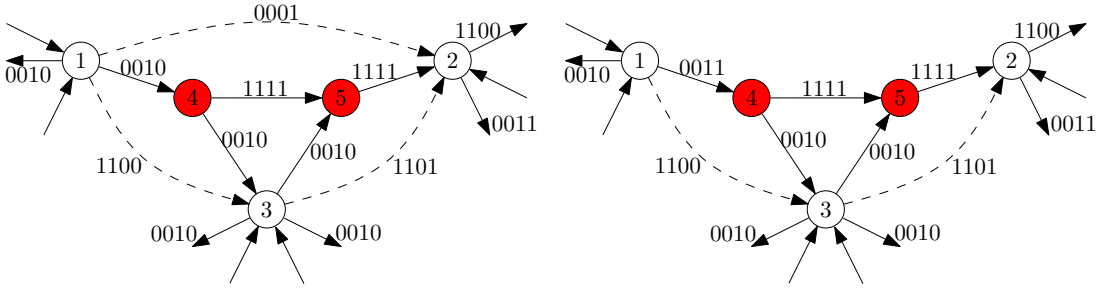


Figure 4.7: Example for removing shortcuts from the output graph. The graph on the left shows the output graph after refinement of arc-flags (cf. Figure 4.6). The shortcut from 1 to 2 is removed and the edges representing the shortcut inherit the flags from it.

a variant of SHARC can be interpreted as a faster preprocessing routine for multi-level arc-flags. However, we might set more flags to true than necessary.

The advantage of this variant is its easy adaptability to existing (commercial) systems. The existing core system may stay untouched; we simply add an arc-flag pointer to each edge, a region information to each node, and store the arc-flag array. Furthermore, the space consumption is very low, as shortcuts are one of the main reasons of space overhead. Finally, this variant needs no shortcut-unpacking routine if the complete path description is required. Summarizing, the variant may be very helpful for PDA-implementations where space is limited and users need the complete path. However, the disadvantage of this approach is its worse performance than SHARC with shortcuts (cf. Section 4.5).

Arc-Flag Compression. In order to reduce the space consumption of SHARC, we compress the arc-flag information. During our studies, we observed that the number of *different* arc-flags is much less than the number of edges. Thus, instead of storing arc-flags for each edge, we use a separate array containing all possible *unique* arc-flags. In order to access the flags efficiently, we assign an additional pointer to each edge indexing the correct arc-flags. This yields a lower space consumption of our preprocessed data.

Boundary-Shortcuts. During our experimental study, we observed that—at least for long-range queries in road networks—a classic bidirectional Arc-Flags Dijkstra often is optimal in the sense that it visits *only* the edges on the shortest path between two nodes. However, such shortest paths may become quite long in road networks. One advantage of SHARC over 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 if 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. Unfortunately, determining optimal shortcuts is complicated [BDDW09]. Hence, 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 [Bra01] 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 heads are those boundary nodes with highest $c_B \cdot h$ values, where h is the number of hops of the added shortcut.

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.

Pruning. If our edge-reduction routine during contraction does not yield a significant decrease in number of edges, we perform an additional pruning step. The idea is the following. Based on the already computed arc-flag information, we can determine edges

that already have their final flags assigned. Hence, we can safely remove them during preprocessing. However, we have to make sure that we add these edges to the output graph during finalization.

We perform our pruning 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. A drawback of this approach is that we can only remove edges from prunable cells.

Lemma 4.6. *Pruning is correct.*

Proof. We have to show that Lemma 4.3 holds at each iteration step. 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.

Let (u_{l_z}, u_{l_z+1}) be an edge of P deleted in the pruning step. Then, all edges on P after (u_{l_z}, u_{l_z+1}) are also deleted in that step. Summarizing, if no edge on $\hat{P}$ is deleted in the pruning step, then $\hat{P}$ fullfills all claims of the lemma for iteration step $i+1$. Otherwise, the path $(v_1, \ldots, v_{j_1}, u_1, \ldots, u_{\ell_1-1};\ u_{\ell_1}, u_{\ell_2}, \ldots; u_{l_k}, \ldots, u_{\ell_d}, u_{\ell_d+1}, \ldots, u_k, w_1, \ldots, w_{j_3})$ fullfills all claims of the Lemma 4.3 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. □

4.4 Hierarchy-Aware Arc-Flags

Two important goal-directed techniques have been established during the last years: ALT and Arc-Flags. The advantages of ALT are fast preprocessing and easy adaption to dynamic scenarios, while the latter is superior with respect to query-performance and space consumption (cf. Chapter 3). However, preprocessing of Arc-Flags is expensive (cf. Section 3.3). The central idea of *Hierarchy-Aware Arc-Flags* is to combine a hierarchical method with Arc-Flags. By computing arc-flags only for a subgraph containing all nodes in high levels of the hierarchy, we are able to reduce preprocessing times. In general, we could use any hierarchical approach but as Contraction Hierarchies (CH) and Transit-Node Routing are the best hierarchical methods with respect to space consumption and query performance, we focus on the combination of Arc-Flags with these techniques. However, we also present a combination of Reach and Arc-Flags.

4.4.1 Contraction Hierarchies + Arc-Flags (CHASE)

As already mentioned in Section 1.2, Contraction Hierarchies basically uses a plain bidirected Dijkstra on a search graph constructed during preprocessing. We are able to combine Arc-Flags and Contraction Hierarchies in a very natural way and name it the CHASE-algorithm (Contraction-Hierarchy + Arc-flagS + highway-nodE routing).

Preprocessing. First, we run a complete Contraction Hierarchies preprocessing which assembles the search graph G'. Next, we extract the subgraph H of G' containing the $|V_H|$ nodes of highest levels. The size of V_H is a tuning parameter. Recall that Contraction

Hierarchies uses $|V|$ levels with the most important node in level $|V|-1$. We partition H into k cells and compute arc-flags according to Section 3.3 for all edges in H. Summarizing, the preprocessing consists of constructing the search graph and computing arc-flags for H.

Query. Basically, the query is a two-phase algorithm. The first phase is a bidirected Dijkstra on G' with the following modification: When settling a node v belonging to H, we do *not* relax any outgoing edge from v. Instead, if v is settled by the forward search, we add v to a node set S, otherwise to T. Phase 1 ends if the search in both directions stops. The search stops in one direction, if either the respective priority queue is empty or if the minimum of the key values in that queue and the distance to the closest entrance point in that direction is equal or larger than the length of the tentative shortest path. The whole search can be stopped after the first phase, if either no entrance points have been found in one direction or if the tentative shortest-path distance is smaller than minimum over all distances to the entrance points and all key values remaining in the queues. Otherwise we switch to phase 2 of the query which we initialize by refilling the queues with the nodes from S and T. As keys we use the distances computed during phase 1. In phase 2, we use a bidirectional Arc-Flags Dijkstra. We identify the set C_S (C_T) of all cells that contain at least one node $u \in S$ ($u \in T$). The forward search only relaxes edges having a `true` arc-flag for any of the cells C_T. The backward search proceeds analogously.

Note that we have a trade-off between performance and preprocessing. If we use bigger subgraphs as input for preprocessing arc-flags, query-performance is better as arc-flags can be used earlier. However, preprocessing time increases as more arc-flags have to be computed.

Stall-On-Demand. Pure Contraction Hierarchies benefit from an optimization technique called stall-on-demand. During the query, a very local breadth-first search *stalls* nodes that cannot be part of the shortest path (cf. [Sch08] for details). However, during our experimental study, it turned out that this optimization technique does not pay off for CHASE. The search space decreases only slightly which cannot compensate the computational overhead of stall-on-demand. So, the resulting query of CHASE is a plain bidirectional Dijkstra operating on G' with arc-flags activated on high levels of the hierarchy.

Theorem 4.7. CHASE *is correct.*

Proof. The correctness of CH is known. If the query terminates during phase 1, then the correctness of the combinations directly follows from the correctness of CH. Otherwise, we know that a shortest s-t path must contain at least two entrance points $\hat{s} \in S$ and $\hat{t} \in T$. As the query relaxes all edges with `true` arc-flags for at least one cell in C_T and C_S, it is certain that the shortest path is found. □

Partial CHASE (pCHASE). Contraction Hierarchies yield excellent preprocessing and query times in road networks. The main reason is that the average degree of nodes with respect to the search graph G' stays low. However, for other inputs, the average degree may grow rapidly yielding bad preprocessing and query times. Our Partial CHASE algorithm is motivated from such inputs. Instead of computing a complete contraction hierarchy, we *stop* the contraction at a certain point. This yields a CH-core H with size $|V_H|$. We use the subgraph induced by V_H as input for arc-flags preprocessing. The idea is that the lacking hierarchy in the core is compensated by goal-direction.

4.4.2 Reach + Arc-Flags (ReachFlags)

Similar to CHASE, we can also combine Reach and Arc-Flags, called *ReachFlags*. We first run a complete Reach-preprocessing as described in [GKW07] and assemble the output graph. Next, we extract a subgraph H from the output graph containing all nodes with a reach value $\geq \ell$. Again, we compute arc-flags in H according to Section 3.3. Note we do not favor one path over another if both paths have the same length. The ReachFlags-query can easily by adapted from the CHASE-query in straightforward manner. Note that the input parameter ℓ adjusts the size of V_H. Thus, a similar trade-off in performance/preprocessing effort like for CHASE is given.

Theorem 4.8. *ReachFlags is correct.*

Proof. We know that pure reach-based routing is correct. With the same observations from the proof of Theorem 4.7, the correctness of ReachFlags follows. □

Partial ReachFlags (pReachFlags). Analogously to Partial CHASE, we can also define a partial variant of ReachFlags. Therefore, we slightly alter the reach preprocessing: Reach-computation according to [GKW07] is a process that iteratively contracts and prunes the input. After each iteration step, all nodes with final reach value assigned are removed from the graph. Starting from this observation, we are able to preprocess Partial ReachFlags. We first run ℓ iteration steps of Reach-preprocessing as described in [GKW07]. All nodes that do not have their final reach value set, get a reach value of ∞ assigned. Next, we assemble the output graph and extract a subgraph H from it containing all nodes with reach ∞. Again, we compute arc-flags in H according to [HKMS09]. Note that for Partial ReachFlags the input parameter ℓ adjusts the size of V_H.

4.4.3 Transit Node Routing + Arc-Flags

Transit Node Routing is based on the observation that when you start from a source node s and drive to somewhere 'far away', you will leave your current location via one of only a few 'important' traffic junctions, called (forward) *access nodes* $\overrightarrow{\mathrm{A}}(s)$. An analogous argument applies to the target t, i.e., the target is reached from one of only a few backward access nodes $\overleftarrow{\mathrm{A}}(t)$. Moreover, the union of all forward and backward access nodes of all nodes, called *transit-node set* $\mathcal{T}$, is rather small. This implies that for each node the distances to/from its forward/backward access nodes and for each transit-node pair (u, v) the distance between u and v can be stored. For given source and target nodes s and t, the length of the shortest path that passes at least one transit node is given by $d_{\mathcal{T}}(s,t) = \min\{d(s,u) + d(u,v) + d(v,t) \mid u \in \overrightarrow{\mathrm{A}}(s), v \in \overleftarrow{\mathrm{A}}(t)\}$. Note that all involved distances $d(s,u)$, $d(u,v)$, and $d(v,t)$ can be directly looked up in the precomputed data structures. As a final ingredient, a *locality filter* $\mathcal{L} : V \times V \to \{\mathsf{true}, \mathsf{false}\}$ is needed that decides whether given nodes s and t are too close to travel via a transit node. $\mathcal{L}$ has to fulfill the property that $\mathcal{L}(s,t) = \mathsf{false}$ implies $d(s,t) = d_{\mathcal{T}}(s,t)$. Then, the following algorithm can be used to compute the shortest-path length $d(s,t)$:

if $\mathcal{L}(s,t) = \mathsf{false}$ **then** compute and return $d_{\mathcal{T}}(s,t)$; **else** use any other routing algorithm.

For a given source-target pair (s,t), let $a := \max(|\overrightarrow{\mathrm{A}}(s)|, |\overleftarrow{\mathrm{A}}(t)|)$. Note that for a global query (i.e., $\mathcal{L}(s,t) = \mathsf{false}$), we need $O(a)$ time to lookup all access nodes, $O(a^2)$ to perform the table lookups, and $O(1)$ to check the locality filter.

Reducing the Number of Table Lookups. So, the most time-consuming part of a TNR-query are the table lookups. Hence, if we want to further improve the average query times, the first attempt should be to reduce the number of those lookups. This can be done by excluding certain access nodes at the outset, using an idea very similar to the arc-flag approach. We consider the minimal overlay graph $G_{\mathcal{T}} = (\mathcal{T}, E_{\mathcal{T}})$ of G, i.e., the graph with node set $\mathcal{T}$ and an edge set $E_{\mathcal{T}}$ such that $|E_{\mathcal{T}}|$ is minimal and for each node pair $(s,t) \in \mathcal{T} \times \mathcal{T}$, the distance from s to t in G corresponds to the distance from s to t in $G_{\mathcal{T}}$. We partition this graph $G_{\mathcal{T}}$ into k regions and store for each node $u \in \mathcal{T}$ its region $r(u) \in \{1, \ldots, k\}$. For each node s and each access node $u \in \overrightarrow{\mathrm{A}}(s)$, we manage a flag vector $f_{s,u}^{\rightarrow} : \{1, \ldots, k\} \to \{\mathsf{true}, \mathsf{false}\}$ such that $f_{s,u}^{\rightarrow}(x)$ is true iff there is a node $v \in \mathcal{T}$ with $r(v) = x$ such that $d(s,u) + d(u,v)$ is equal to $\min\{d(s,u') + d(u',v) \mid u' \in \overrightarrow{\mathrm{A}}(s)\}$. In other words, a flag of an access node u for a particular region x is set to true iff u is useful to get to some transit node in the region x when starting from the node s. Analogous flag vectors $f_{t,u}^{\leftarrow}$ are kept for the backward direction.

Preprocessing. The flag vectors can be precomputed in the following way, again using ideas similar to those used in the preprocessing of the arc-flag approach: Let $B \subseteq \mathcal{T}$ denote the set of border nodes, i.e., nodes that are adjacent to some node in $G_{\mathcal{T}}$ that belongs to a different region. For each node $s \in V$ and each border node $b \in B$, we determine the access nodes $u \in \overrightarrow{\mathrm{A}}(s)$ that minimize $d(s,u) + d(u,b)$; we set $f_{s,u}^{\rightarrow}(r(b))$ to true. In addition, $f_{s,u}^{\rightarrow}(r(u))$ is set to true for each $s \in V$ and each access node $u \in \overrightarrow{\mathrm{A}}(s)$ since each access node obviously minimizes the distance to itself. An analogous preprocessing step has to be done for the backward direction.

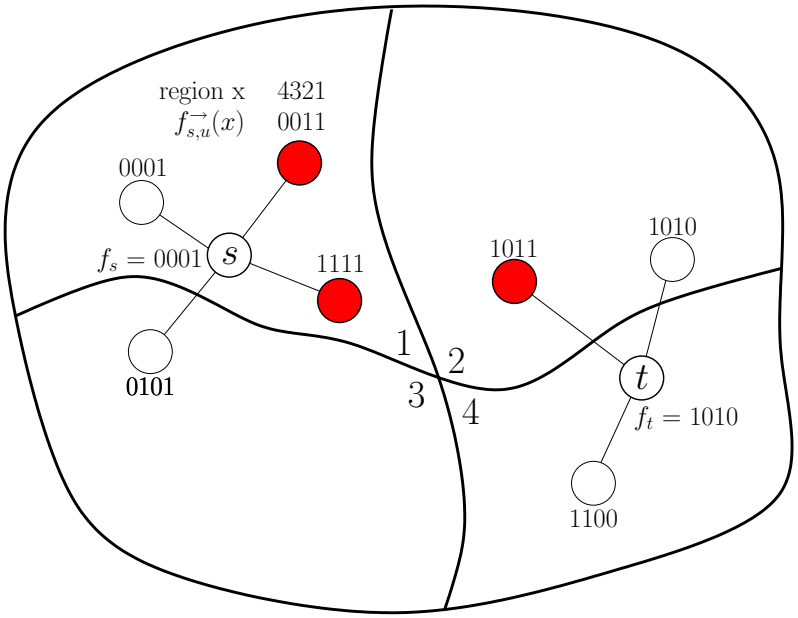


Figure 4.8: An example of a goal-directed transit-node query. Nodes selected for $\overrightarrow{\mathrm{A}}'(s)$ and $\overleftarrow{\mathrm{A}}'(t)$ are shaded. The number of required table lookups is reduced from 12 to 2.

Query. In a query from s to t, we can take advantage of the precomputed flag vectors. First, we consider all backward access nodes of t and build the flag vector f_t such that $f_t(r(u)) = \mathsf{true}$ for each $u \in \overleftarrow{\mathrm{A}}(t)$. Second, we consider only forward access nodes u of s with the property that the bitwise AND of $f^{\rightarrow}_{s,u}$ and f_t is not zero; we denote this set by $\overrightarrow{\mathrm{A}}'(s)$; during this step, we also build the vector f_s such that $f_s(r(u)) = \mathsf{true}$ for each $u \in \overrightarrow{\mathrm{A}}'(s)$. Third, we use f_s to determine the subset $\overleftarrow{\mathrm{A}}'(t) \subseteq \overleftarrow{\mathrm{A}}(t)$ analogously to the second step. Now, it is sufficient to perform only $|\overrightarrow{\mathrm{A}}'(s)| \times |\overleftarrow{\mathrm{A}}'(t)|$ table lookups. An example is given in Fig. 4.8. Note that determining $\overrightarrow{\mathrm{A}}'(s)$ and $\overleftarrow{\mathrm{A}}'(t)$ is in $O(a)$, in particular operations on the flag vectors can be considered as quite cheap.

Theorem 4.9. *TNR+AF is correct.*

Proof. We know that pure Transit Node Routing is correct. Hence, we have to show that $\min\{d(s,u) + d(u,v) + d(v,t) \mid u \in \overrightarrow{\mathrm{A}}(s), v \in \overleftarrow{\mathrm{A}}(t)\} = \min\{d(s,u) + d(u,v) + d(v,t) \mid u \in \overrightarrow{\mathrm{A}}'(s), v \in \overleftarrow{\mathrm{A}}'(t)\}$. Consider the nodes $u \in \overrightarrow{\mathrm{A}}(s)$ and $v \in \overleftarrow{\mathrm{A}}(t)$ that minimize the distance $d(s,u) + d(u,v) + d(v,t)$. In particular, we have $d(s,u) + d(u,v) = \min\{d(s,u') + d(u',v) \mid u' \in \overrightarrow{\mathrm{A}}(s)\}$, which implies that $f^{\rightarrow}_{s,u}(r(v)) = \mathsf{true}$. Furthermore, we have $f_t(r(v)) = \mathsf{true}$ since $v \in \overleftarrow{\mathrm{A}}(t)$. Hence, the bitwise AND of $f^{\rightarrow}_{s,u}$ and f_t is not zero and, consequently, $u \in \overrightarrow{\mathrm{A}}'(s)$. Analogously, we can show that $v \in \overleftarrow{\mathrm{A}}'(t)$. □

Optimizations. Presumably, it is a good idea to just store the bitwise OR of the forward and backward flag vectors in order to keep the memory consumption within reasonable bounds. The preprocessing of the flag vectors can be accelerated by rearranging the columns of the distance table so that all border nodes are stored consecutively, which reduces the number of cache misses.

4.5 Experiments on Road Networks

In this section, we present an extensive experimental evaluation of all techniques presented in this chapter. We here concentrate on road networks with travel times as metric, while we evaluate other networks in Section 4.6. Our implementation is written in C++ using solely the STL at some points. As priority queue we use a binary heap. See Appendix A for details. Our tests were executed on one core of an AMD Opteron 2218 running SUSE Linux 10.3. The machine is clocked at 2.6 GHz, has 16 GB of RAM and 2 x 1 MB of L2 cache, the DIMACS benchmark [DGJ06] on the full US road network with travel time metric takes 6 013.6 s. The program was compiled with GCC 4.2.1, using optimization level 3.

Inputs. As inputs for our test on road networks we use the largest strongly connected component[1] of the road networks of Western Europe, provided by PTV AG for scientific use, and of the US which is taken from the DIMACS Challenge homepage. The former graph has approximately 18 million nodes and 42.6 million edges. The corresponding figures for the USA are 23.9 million and 58.3 million, respectively. In both cases, edge lengths correspond to travel times. Each edge belongs to one of four main categories representing motorways, national roads, local streets, and urban streets. The European network has a fifth category representing rural roads.

[1]For historical reasons, some quoted results are based on the respective original network that contains a few additional nodes that are not connected to the largest strongly connected component.

Setup. In the following we report preprocessing effort and query performance of all speed-up techniques. For the former, we report the preprocessing time and the resulting *additional* space per node, while for the latter we report the average number of settled nodes, i.e., the number of nodes taken from the priority queues, and resulting query times. All figures in this paper refer to the scenario that only the lengths of the shortest paths have to be determined, without outputting a complete description of the paths.

We report two types of queries. For *random queries*, 10 000 random pairs of source and target are selected, while for *local queries* [SS05], 1 000 (s, t) pairs are chosen for each Dijkstra rank: Starting a query from s, the rank of t is denoted by the number of settled nodes before t is settled. It is given for $2^0, 2^1, \ldots, 2^{\log |V|}$. This setup is applied to some speed-up techniques in order to gain further insights into their performance on a particular graph depending on the length of a query. The results are presented in the form of a box-and-whisker plot [Tea04].

4.5.1 ALT

The Static ALT Algorithm. Before comparing our ALT implementations in a dynamic scenario, we report the performance of the bi- and unidirectional ALT algorithm in a static scenario. We evaluate different numbers of landmarks with respect to preprocessing, search space and query times performing 10 000 uniformly distributed random s-t queries. Due to memory requirements we used *avoid* for selecting 32 and 64 landmarks. For less landmarks, we used the superior *maxCover* heuristic. Table 4.1 gives an overview.

We see for bidirectional ALT that doubling the number of landmarks reduces search space and query times by factor 2, which does not hold for the unidirectional variant. This is due to the fact that goal direction works best on motorways as these roads mostly have reduced costs of 0 in the reduced graph. In the unidirectional search, one has to leave the motorway in order to reach the target. This drawback cannot be compensated by more landmarks. Comparing uni- and bidirectional ALT, one may notice that the time

Table 4.1: Preprocessing, search space, and query times of uni- and bidirectional ALT and Dijkstra based on 10 000 random s-t queries. The column *dist.* refers to the time needed to recompute all distance labels from scratch.

		Preprocessing			Query Unidir.		Query Bidir.	
		time	space	dist.	# settled	time	# settled	time
graph	algorithm	[min]	[B/n]	[min]	nodes	[ms]	nodes	[ms]
Europe	Dijkstra	0.0	0	0.0	9 114 385	5 591.6	4 764 110	2 713.2
	ALT-4	12.1	32	1.4	1 289 070	469.1	355 442	254.1
	ALT-8	26.1	64	2.8	1 019 843	391.6	163 776	127.8
	ALT-16	85.2	128	5.5	815 639	327.6	74 669	53.6
	ALT-24	145.0	192	8.3	742 958	303.7	56 338	44.2
	ALT-32	27.1	256	11.1	683 566	301.4	40 945	29.4
	ALT-64	68.2	512	22.1	604 968	288.5	25 324	19.6
USA	Dijkstra	0.0	0	0.0	11 847 523	6 780.7	7 345 846	3 751.4
	ALT-8	44.5	64	3.4	922 897	329.8	328 140	219.6
	ALT-16	103.2	128	6.8	762 390	308.6	180 804	129.3
	ALT-32	35.8	256	13.6	628 841	291.6	109 727	79.5
	ALT-64	92.9	512	27.2	520 710	268.8	68 861	48.9

spent per node is significantly smaller than for uni-ALT. The reason is the computational overhead for performing a bidirectional search. A reduction in search space of factor 44 (USA, ALT-16) yields a reduction in query time of factor 29. This is an overhead of factor 1.5 instead of 2.1, suggested by the figures in [GW05], deriving from our more efficient storage of landmark data (cf. Section 3.2).

Changing the Metric. As we do not consider repositioning landmarks, we only have to recompute all distance labels by rerunning a forward and backward Dijkstra from each landmark whenever the metric changes. With this strategy, we are able to change a metric in 5.5 minutes when using 16 landmarks on the European network.

Updating the Preprocessing. Before testing the lazy variant of dynamic ALT, we evaluate the time needed for updating all distance labels. Note, that even the lazy variant has to update the preprocessing sometimes. With the obtained figures we want to measure the trade-off for which types of perturbations the update of the preprocessing is worth the effort. Figure 4.9 shows the time needed for updating all 32 trees needed for 16 landmarks if an edge is increased or decreased by factor 2 (*low perturbation*) and 10 (*high perturbation*). We distinguish the different types of edges.

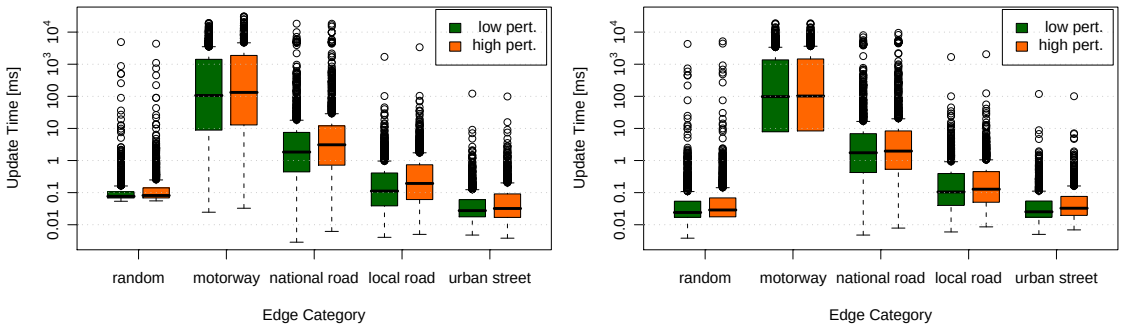


Figure 4.9: Required time for updating the 32 shortest path trees needed for 16 landmarks on the European instance. The figure on the left shows the average runtime of increasing one edge by factor 2 (grey) and by 10 (white) while the right reports the corresponding values for decrementing edges. For each category, the figures are based on 10 000 edge updates.

We observe that updating the preprocessing if an important edge is altered is more expensive than the perturbation of other road types. This is due to the fact that motorway edges have many descendants within the shortest path trees. Thus, more nodes are affected by such an update. However, the type of update has nearly no impact on the time spent for an update: neither how much an edge is increased nor whether an edge is increased or decreased. For almost all kind of updates we observe high fluctuations in update time. Very low update times are due to the fact that the routine is done if an increased edge is not a tree edge or a decreased edge does not yield a lower distance label. Outliers of high update times are due to the fact that not only the type of the edge has an impact on the importance for updates: altering a urban street being a tree edge near a landmark may lead to a dramatic change in the structure of the tree of this landmark.

Table 4.2: Search space and query times of lazy dynamic ALT algorithm performing 10 000 random s-t queries after 1 000 edges of a specific category have been perturbed by factor 2. The figures in parentheses refer to increases by factor 10. The percentage of affected queries (the shortest path contains an updated edge) is given in column number 3.

			lazy ALT-16				lazy ALT-32			
graph	road type	aff.[%]	# settled nodes		increase [%]		# settled nodes		increase [%]	
EUR	All roads	7.5	74 700	(77 759)	0.0	(4.1)	41 044	(43 919)	0.2	(7.3)
	urban	0.8	74 796	(74 859)	0.2	(0.3)	40 996	(41 120)	0.1	(0.4)
	local	1.5	74 659	(74 669)	0.0	(0.0)	40 949	(40 995)	0.0	(0.1)
	national	28.1	74 920	(75 777)	0.3	(1.5)	41 251	(42 279)	0.7	(3.3)
	motorway	95.3	97 249	(265 472)	30.2	(255.5)	59 550	(224 268)	45.4	(447.7)
USA	All roads	3.3	181 335	(181 768)	0.3	(0.5)	110 161	(110 254)	0.4	(0.5)
	urban	0.1	180 900	(180 776)	0.1	(0.0)	109 695	(110 108)	0.0	(0.3)
	local	2.6	180 962	(181 068)	0.1	(0.1)	109 873	(109 902)	0.1	(0.2)
	national	25.5	181 490	(184 375)	0.4	(2.0)	110 553	(112 881)	0.8	(2.9)
	motorway	94.3	207 908	(332 009)	15.0	(83.6)	130 466	(247 454)	18.9	(125.5)

Lazy Dynamic ALT. In the following, we evaluate the robustness of the lazy variant of ALT with respect to network changes. Therefore, we alter different types and number of edges by factor 2 and factor 10.

Edge Categories. First, we concentrate on different types of edge categories. Table 4.2 gives an overview of the performance for both dynamic ALT variants if 1 000 edges are perturbed before running random queries. We see that altering low-category edges has nearly no impact on the performance of lazy ALT. This is independent of the level of increase. As expected, altering motorway edges yields a loss in performance. We observe a loss of 30–45% for Europe and 15–19% for the US if the level of increase is moderate (factor 2). The situation changes for high perturbation. For Europe, queries are 3.5–5.5 times slower than in the static case (cf. Table 4.1), depending on the number of landmarks. The corresponding figures for the US are 1.8–2.3. Thus, lazy ALT is more robust on the US network than on the European. The loss in performance originates from the fact that for most queries, unperturbed motorways on the shortest path have costs of 0 in the reduced graph. Thus, the search stops later if these motorways are perturbed yielding a higher search space (cf. Section 4.1.2). Nevertheless, comparing the query times to a bidirectional Dijkstra, we still gain a speed-up of above 10. Combining the results from Figure 4.9 with the ones from Table 4.2, we conclude that updating the preprocessing has no advantage. For motorways, updating the preprocessing is expensive and altering other types of edges has no impact on the performace of lazy ALT.

Number of Updates. In Table 4.2, we observed that the perturbation of motorways has the highest impact on the lazy dynamic variant of ALT. Next, we change the number of perturbed motorways. Table 4.3 reports the performance of lazy dynamic ALT when different numbers of motorways are increased by factor 2 and factor 10, respectively, before running random queries on Europe. For perturbations by factor 2, we observe almost no loss in performance for less than 500 updates, although up to 87% of the queries are affected by the perturbation. Nevertheless, 2 000 or more perturbed edges lead to significant decreases in performance, resulting in query times of about 0.5 seconds for 10 000 updates. Note that the European network contains only about 175 000 motorway

Table 4.3: Search space and query times of the dynamic ALT algorithm performing 10 000 random s-t queries after a variable number of motorway edges have been increased by factor 2. The figures in parentheses refer to increases by factor 10. The percentage of affected queries (the shortest path contains an updated edge) is given in column 2.

		lazy ALT-16				lazy ALT-32			
#edges	aff.[%]	# settled nodes		increase [%]		# settled nodes		increase [%]	
100	39.9	75 691	(91 610)	1.4	(22.7)	41 725	(56 349)	1.9	(37.6)
200	64.7	78 533	(107 084)	5.2	(43.4)	44 220	(69 906)	8.0	(70.7)
500	87.1	86 284	(165 022)	15.6	(121.0)	50 007	(124 712)	22.1	(204.6)
1 000	95.3	97 249	(265 472)	30.2	(255.5)	59 550	(224 268)	45.4	(447.7)
2 000	97.8	154 112	(572 961)	106.4	(667.3)	115 111	(531 801)	181.1	(1 198.8)
5 000	99.1	320 624	(1 286 317)	329.4	(1622.7)	279 758	(1 247 628)	583.3	(2 947.1)
10 000	99.5	595 740	(2 048 455)	697.8	(2643.4)	553 590	(1 991 297)	1252.0	(4 763.3)

edges. As expected, the loss in performance is higher when motorway edges are increased by factor 10. For this case, up to 500 perturbations can be compensated well. Comparing slight and high increases we observe that the lazy variant can compensate four times more updates, e.g., 500 increases by factor 10 yield almost the same loss as 2 000 updates by factor 2.

The number of landmarks has almost no impact on the performance if more than 5 000 edges are perturbed. This is due to the fact that for almost all motorways the landmarks do not yield good reduced costs. We conclude that the lazy variant cannot compensate such a high degree of perturbation.

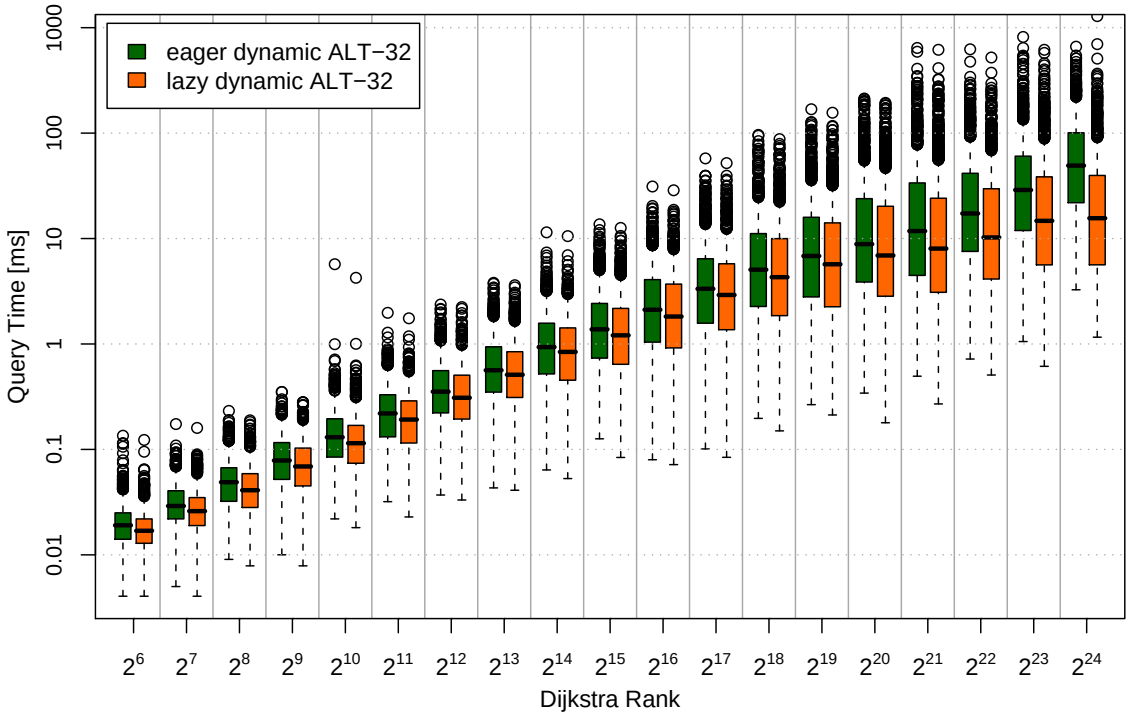


Figure 4.10: Comparison of query times of the lazy and eager dynamic variant of ALT using the Dijkstra rank methodology. The queries were run after 1 000 motorways were increased by a factor of 2.

Comparing Lazy and Eager Dynamic ALT. Table 4.2 shows that lazy ALT-32 yields an increase of 40% in search space for random queries on the European network with 1 000 low perturbed motorway edges. In order to obtain a more detailed insight for which types of queries these differences originates from, Figure 4.10 reports the query times of eager and lazy ALT-32 with respect to the Dijkstra rank in this scenario.

Query performance varies so heavily that we use a logarithmic scale. For each rank, we observe queries performing 20 times worse than the median. This is originated from the fact that for some queries no landmark provides very good lower bounds resulting in significantly higher search spaces. Comparing the eager and lazy dynamic version, we observe that query times differ only by a small factor for Dijkstra ranks below 2^{20}. However, for 2^{24}, the eager version is about factor 5 faster than the lazy one. This is due to the fact that those types of queries contain a lot of motorways and most of the jammed edges are used. The eager version yields a good potential for these edge while the lazy does not. We conclude that lazy ALT is more robust for small range queries than for long distance requests. Note that in a real-world scenario, you probably do not want to use traffic information that is more than one hour away from your current position. As the ALT algorithm provides lower bounds to *all* positions within the network, it is possible to ignore traffic jams efficiently without any additional information.

4.5.2 Core-ALT

For CALT, we first evaluate the impact of contraction on preprocessing effort and query performance. Table 4.4 reports the performance of CALT with 64 avoid landmarks [GW05] with varying contraction rate. Note that for $c = 0.0$ and $h = 0$, we end up in a plain ALT-setup. We observe that contraction has a very positive effect on ALT: Pre-

Table 4.4: Performance of CALT for varying contraction parameters c and h. Column *core nodes* depicts the percentage of core nodes in G_F, column *#add edges* reports the number of additional edges in G_F, and the resulting overhead (including landmark distances) is given in bytes per node. The preprocessing time is given in minutes. For queries, we report the size of the search space in the number of settled nodes, the number of entry nodes and the resulting average query times in milliseconds.

			Prepro.				Query		
			core	$\lvert E\rvert$	time	space	#settled	#entry	time
input	c	h	nodes	incr.	[min]	[B/n]	nodes	nodes	[ms]
	0.0	0	100.00%	0.00%	68	512.0	25 324	1.0	19.61
	0.5	10	35.48%	10.23%	21	187.7	10 925	3.2	8.02
	1.0	20	6.32%	14.24%	7	38.4	2 233	8.2	2.16
Europe	2.0	30	3.04%	11.41%	9	21.8	1 382	13.3	1.55
	2.5	50	1.88%	9.16%	11	15.4	1 394	18.6	1.34
	3.0	75	1.29%	7.80%	12	12.2	1 963	24.2	1.43
	5.0	100	0.86%	6.94%	18	9.8	3 126	34.0	1.67
	0.0	0	100.00%	0.00%	93	512.0	68 861	1.0	48.87
	0.5	10	28.90%	11.40%	20	154.2	21 544	3.4	16.61
	1.0	20	8.29%	12.68%	11	48.9	7 662	7.1	6.96
USA	2.0	30	3.21%	10.53%	12	22.5	3 338	12.6	4.11
	2.5	50	2.06%	8.00%	13	16.1	2 697	17.1	3.01
	3.0	75	1.45%	6.39%	14	12.6	2 863	22.0	2.85
	5.0	100	0.86%	4.50%	21	9.3	3 416	30.2	2.35

processing space and time decrease combined with better query performance. The latter is accelerated by more than one order of magnitude while the high memory consumption of ALT can be reduced to a reasonable amount. The number of additional edges in G_F first increases with increasing contraction rates but then decreases again. The reason for this is that the core shrinks rapidly. The few core nodes finally yield a high average degree but with respect to the total number of nodes, the impact of core edges fades. For Europe, we observe that at a certain point, higher contraction values yield worse query performance. It seems as if a good compromise is $c = 2.5$ and $h = 50$. Hence, we use these contraction values as default from now on.

Number of Landmarks. Next, we focus on the impact of the number of landmarks. More precisely, we evaluate 8, 16, 32, and 64 landmarks generated on cores obtained from different contraction rates. The results are given in Tab. 4.5. Note that for we use *maxCover* for 8 and 16 landmarks, while *avoid* was used to select 32 and 64 landmarks. Also note that a contraction of $c = 0.0$ and $h = 0$ again yields a pure ALT setup.

Table 4.5: Performance of CALT for different numbers of landmarks applying low contraction parameters.

	no cont. (c=0.0, h=0)				low cont. (c=1.0, h=20)			
	Prepro.		Query		Prepro.		Query	
	time	space	#settled	time	time	space	#settled	time
$\|L\|$	[min]	[B/n]	nodes	[ms]	[min]	[B/n]	nodes	[ms]
8	26.1	64	163 776	127.8	7.1	10.9	12 529	10.25
16	85.2	128	74 669	53.6	9.4	14.9	5 672	5.77
32	27.1	256	40 945	29.4	6.8	23.0	3 268	2.97
64	68.2	512	25 324	19.6	8.5	36.2	2 233	2.16

	med: c=2.5, h=50				high: c=5.0, h=100			
	Prepro.		Query		Prepro.		Query	
	time	space	#settled	time	time	space	#settled	time
$\|L\|$	[min]	[B/n]	nodes	[ms]	[min]	[B/n]	nodes	[ms]
8	10.1	7.0	4 431	3.98	17.8	5.9	4 106	2.51
16	11.0	8.2	2 456	2.33	18.3	6.5	3 500	2.23
32	10.0	10.6	1 704	1.66	17.7	7.6	3 264	2.01
64	10.5	15.4	1 394	1.34	18.0	9.8	3 126	1.67

We observe that with decreasing size of the core, the impact of number of landmarks fades: In a pure ALT setting, doubling the number of landmarks roughly yields an increase of a factor of 2 in query performance. On the contrary, in a high contraction scenario ($c = 5.0, h = 100$), the number of landmarks has nearly no influence on query performance. Using 64 instead of 8 landmarks decreases query times by only $\approx 33\%$. However, as memory consumption is still very low for 64 landmarks, we use this number as default for CALT.

Local Queries In order to gain deeper insights into the impact of contraction on query performance, Fig. 4.11 reports the query times of CALT for different contraction rates with respect to the Dijkstra rank. For ALT, we use 16 *maxCover* landmarks, for CALT 64 landmarks are selected by *avoid*. We observe that pure ALT is faster than CALT for ranks up to 2^8 if low contraction is applied. If the core gets smaller, pure ALT is faster than CALT for ranks up to 2^{10}. This is due to the fact that CALT has a two-phase query

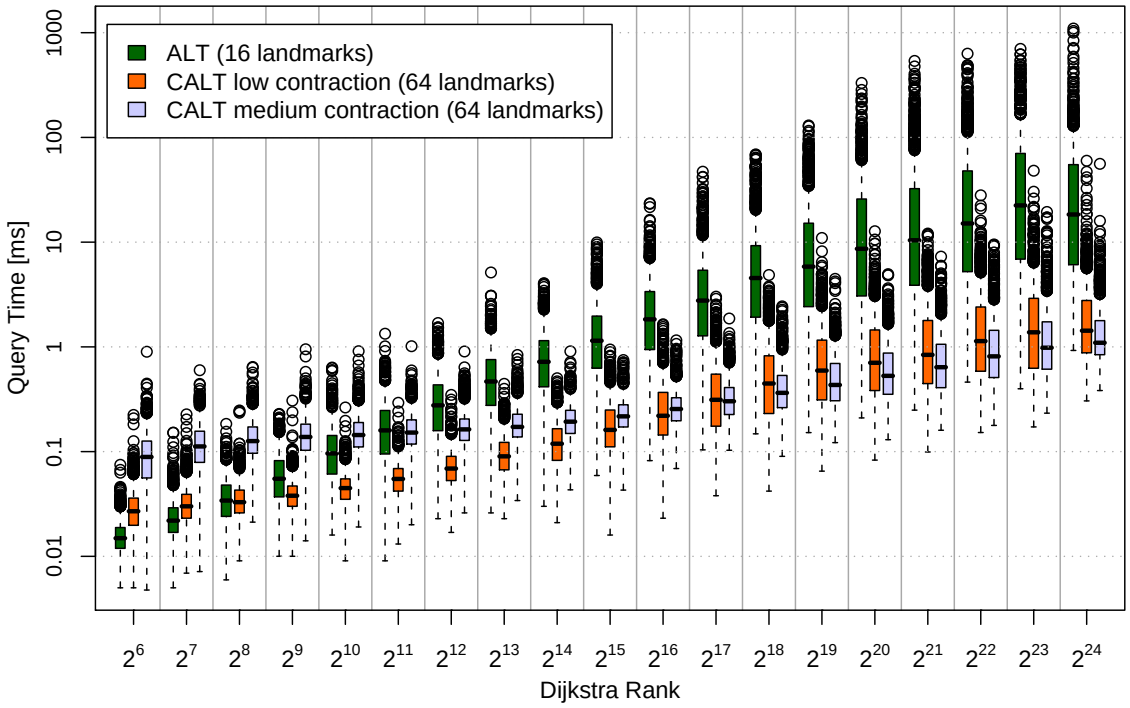


Figure 4.11: Comparison of ALT with 16, CALT with low contraction ($c = 1.0$, $h = 20$), and medium contraction ($c = 2.5$, $h = 50$) using the Dijkstra rank methodology.

yielding a higher overhead. It seems as if increasing the contraction rate has a negative effect for low-range queries while long-range queries seem to benefit from higher contraction rates. Still, low-range queries are executed in less than 1 ms for both contraction setups. Moreover, space consumption decreases with increasing contraction rates (cf. Tab 4.4). Hence, our choice of $c = 2.5$, $h = 50$ as default setting seems reasonable.

4.5.3 SHARC

Default Setting. Unless otherwise stated, we use a *unidirectional* variant of SHARC. We use $c = 2.5$ as contraction parameter and $h = 10$ as hop-bound. We use our path-expansion optimization only for our stripped variant of SHARC.

Multi-Level Partition. One main parameter of SHARC preprocessing is the multi-level partition. Table 4.6 reports the performance of SHARC if different types of SCOTCH-partitions are applied. We observe that the performance of SHARC highly depends on the partition of the graph. A classic 1-level setup yields query times of 23.6 ms. By increasing the number of levels, we achieve query times of down to 0.29 ms. Interestingly, the preprocessing time is almost the same for all applied partitions: We need roughly 1.5 hours for preprocessing. However, using more than 6 levels does not pay off: query times stay the same but the overhead increases. In general, it seems as if the best trade-off between preprocessing effort and query performance is achieved if the average number of nodes per cell is roughly 80. This value is achieved in a 6-level setup with 4,4,4,4,8,104 cells. Hence, we use this partition for both our continental-sized road networks.

Table 4.6: Performance of SHARC with different partitions. Column *prepro* shows the computation time of the preprocessing in hours and minutes and the eventual *additional* bytes per node needed for the preprocessed data. For queries, the search space is given in the number of settled nodes and the number of relaxed edges, execution times are given in milliseconds.

PARTITION										PREPRO		QUERY		
#cells per level									∅#nodes	time	space	#settled	#rel.	time
l_0	l_1	l_2	l_3	l_4	l_5	l_6	l_7	#total cells	per cell	[h:m]	[B/n]	nodes	edges	[μs]
128	-	-	-	-	-	-	-	128	140 704.5	1:52	6.0	78 429	178 103	23 306
8	120	-	-	-	-	-	-	960	18 760.6	1:14	9.8	11 362	26 323	3 049
4	4	120	-	-	-	-	-	1 920	9 380.3	1:24	10.6	5 982	14 128	1 637
4	8	116	-	-	-	-	-	3 712	4 851.9	1:25	10.8	3 459	8 372	983
8	8	112	-	-	-	-	-	7 168	2 512.6	1:36	11.5	2 182	5 389	667
16	16	96	-	-	-	-	-	24 576	732.8	2:12	13.1	1 217	3 169	428
4	4	4	116	-	-	-	-	7 424	2 425.9	1:20	11.2	2 025	5 219	625
4	4	8	112	-	-	-	-	14 336	1 256.3	1:14	11.6	1 320	3 544	441
4	8	8	108	-	-	-	-	27 648	651.4	1:15	12.4	984	2 755	358
4	8	16	100	-	-	-	-	51 200	351.8	1:17	13.1	819	2 357	319
4	4	4	4	112	-	-	-	28 672	628.1	1:12	12.0	957	2 827	360
4	4	4	8	108	-	-	-	55 296	325.7	1:13	13.0	774	2 337	309
4	4	8	8	104	-	-	-	106 496	169.1	1:18	13.7	700	2 153	294
4	4	4	16	100	-	-	-	102 400	175.9	1:16	13.7	703	2 162	295
4	4	8	16	96	-	-	-	196 608	91.6	1:24	15.0	671	2 066	287
4	8	8	16	92	-	-	-	376 832	47.8	1:30	16.0	663	2 046	288
4	4	4	4	4	108	-	-	110 592	162.9	1:15	13.6	695	2 263	299
4	**4**	**4**	**4**	**8**	**104**	**-**	**-**	**212 992**	**84.6**	**1:21**	**14.5**	**654**	**2 116**	**290**
4	4	4	8	8	100	-	-	409 600	44.0	1:28	16.1	645	2 087	290
4	4	4	8	16	92	-	-	753 664	23.9	1:31	17.7	646	2 028	289
4	4	8	8	16	88	-	-	1 441 792	12.5	1:50	19.8	663	2 085	296
4	4	4	4	4	4	104	-	425 984	42.3	1:27	15.6	649	2 209	299
4	4	4	4	4	8	100	-	819 200	22.0	1:31	17.6	628	2 094	289
4	4	4	4	8	8	96	-	1 572 864	11.5	1:46	19.3	637	2 092	294
4	4	4	4	8	16	88	-	2 883 584	6.2	1:54	20.7	663	2 100	303
4	4	4	8	8	16	84	-	5 505 024	3.3	2:00	21.5	655	2 035	294
4	4	4	4	4	4	4	100	1 638 400	11.0	1:43	19.2	650	2 247	308
4	4	4	4	4	4	8	96	3 145 728	5.7	1:56	20.0	627	2 113	293
4	4	4	4	4	8	8	92	6 029 312	3.0	1:51	21.1	649	2 121	300
4	4	4	4	4	8	16	84	11 010 048	1.6	2:03	21.5	648	2 035	296

Table 4.7: Performance of SHARC with varying contraction parameter.

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

Contraction Rate. Next, we check whether our choice of contraction parameter is useful. Table 4.7 shows the performance of SHARC with various contraction rates if our default 6-level partition with 4,4,4,4,8,104 cells is given. We observe a contraction rate other than 2.5 increases preprocessing space. While $c = 3.0$ increases query performance marginally, a lower contraction rate also yields worse query times. Hence, our choice of $c = 2.5$ is reasonable in this setup.

Reduction of Preprocessing Duration. SHARC exploits two aspects of a network in order to speed up the query: hierarchical properties by contraction, goal-direction by arc-flags. Table 4.8 shows the performance of SHARC if we do *not* compute arc-flags for all parts of the graph. This can be achieved by either *not* computing core arc-flags on lower levels or not refining low-level arc-flags. If we skip core arc-flags computation, we simply set all flags to true. Hence, we are able to reveal the main reasons for the good performance of SHARC. We observe that SHARC is already 65 times faster than pure Dijkstra if we do not compute any arc-flags at all. Note that this speed-up is achieved with a preprocessing lasting only 16 minutes. By computing arc-flags on different levels we can vary the trade-off between preprocessing effort and query performance: 34 minutes of preprocessing already yields query times of 355 μs. Hence, an additional preprocessing of 18 minutes (over a pure hierarchical setup) accelerates SHARC by an additional factor of 200. Computing arc-flags for the remaining levels costs another 47 minutes but query performance only increases by 20%. Summarizing, dropping goal-direction on lower levels of the hierarchy reduces preprocessing significantly without a dramatic decrease in query performance.

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

Table 4.8: Performance of SHARC for varying effort computing arc-flags. *Core levels* indicates during which iteration steps, core flags are computed. *Refinement levels* depict the levels on which arc-flags are refined.

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

Table 4.9: Performance of stripped SHARC with varying contraction rate during preprocessing.

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

Stripped SHARC. In Section 4.3 we discussed that we can remove *all* shortcuts during the last step of preprocessing. Table 4.9 reports the performance of stripped SHARC with different contraction parameters during preprocessing. Note that in contrast to the figures given in Table 5.6, we do *not* add boundary shortcuts, as they are removed anyway, at the end. Moreover, we do not use our locality optimization, but turn on path-expansion. It turns out that stripped SHARC requires a smaller contraction rate during preprocessing than normal SHARC. A contraction rate of 1.0 already yields very bad query performance for the stripped variant. However, applying a contraction rate of 0.5, the gap between normal and stripped SHARC almost closes. The disadvantage of such a low contraction rate is preprocessing time: it increases to almost 8 hours. However, as already mentioned, stripped SHARC should mainly be used in scenarios with limited memory, e.g., PDAs. Hence, preprocessing would be done once on a server and the preprocessed data would then be transfered to a PDA.

Random Queries. Table 4.10 reports the results of our different SHARC-variants. More precisely, we report the results of our economical, generous, and stripped version of SHARC. In addition, we report the results of bidirectional SHARC which uses bidirectional search in connection with a 2-level partition (16 cells per supercell at level 0, 112 at level 1). We observe excellent query times for SHARC in general. SHARC has a lower preprocessing time for the US than for Europe but for the price of worse query performance. This is due to the fact that the average hop number of shortest paths are bigger for the US than for Europe. However, the number of boundary nodes is smaller for the US yielding lower preprocessing effort. The bidirectional variant of SHARC has a more extensive preprocessing: both time and additional space increase, which is due to computing and storing forward and backward arc-flags. Comparing query performance, bidirectional

Table 4.10: Performance of different SHARC variants on the European and US road network with travel times. *Prepro* shows the computation time of the preprocessing in hours and minutes and the eventual *additional* bytes per node needed for the preprocessed data. For queries, the search space is given in the number of settled nodes, execution times are given in *microseconds*. Note that other techniques have been evaluated on slightly different computers.

	Europe				USA			
	PREPRO		QUERY		PREPRO		QUERY	
	time	space	#settled	time	time	space	#settled	time
	[h:m]	[B/n]	nodes	[μs]	[h:m]	[B/n]	nodes	[μs]
generous SHARC	1:21	14.5	654	290	0:58	18.1	865	376
economical SHARC	0:34	13.7	784	355	0:38	17.2	1 230	578
stripped SHARC	7:48	7.8	14 697	4 762	6:41	9.2	38 817	12 719
bidirectional SHARC	2:38	21.0	125	65	2:34	23.1	254	118

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 3.3). The stripped variant is more than one order of magnitude slower than SHARC with shortcuts, and preprocessing times are higher. However, the strength of this approach is its easy adaptability to existing commercial implementations. Still, stripped SHARC is about three orders of magnitude faster than plain Dijkstra.

Local Queries. Figure 4.12 reports the query times of generous, economical, and bidirectional SHARC with respect to the Dijkstra rank. For an s-t query, the Dijkstra rank of node v is the number of nodes removed from the priority queue by Dijkstra's algorithm before v is removed. Thus, it is a kind of distance measure. As input we again use the European road network instance. Note that we use a logarithmic scale. Both economical and generous SHARC get slower with increasing rank but the median stays below 0.4 ms for the economical variant. The corresponding figure for the generous variant is 0.23 ms. We observe that the gap between both unidirectional variants is almost the same for all ranks. Comparing uni- and bidirectional SHARC, we observe that the former is faster for low-range queries while the latter wins for long-range queries. This is mainly due to the lower number of levels of the bidirectional setup: query times increase up to ranks of 2^{13} which is roughly the size of cells at the lowest level. Above this rank query times decrease and increase again till the size of cells at level 1 is reached. As we use more levels in a unidirectional setup, this effect deriving from the partition cannot be observed for the unidirectional variant. Comparing uni- and bidirectional SHARC we observe more outliers for the latter which is mainly due to less levels. Still, all outliers are below 5.2 ms.

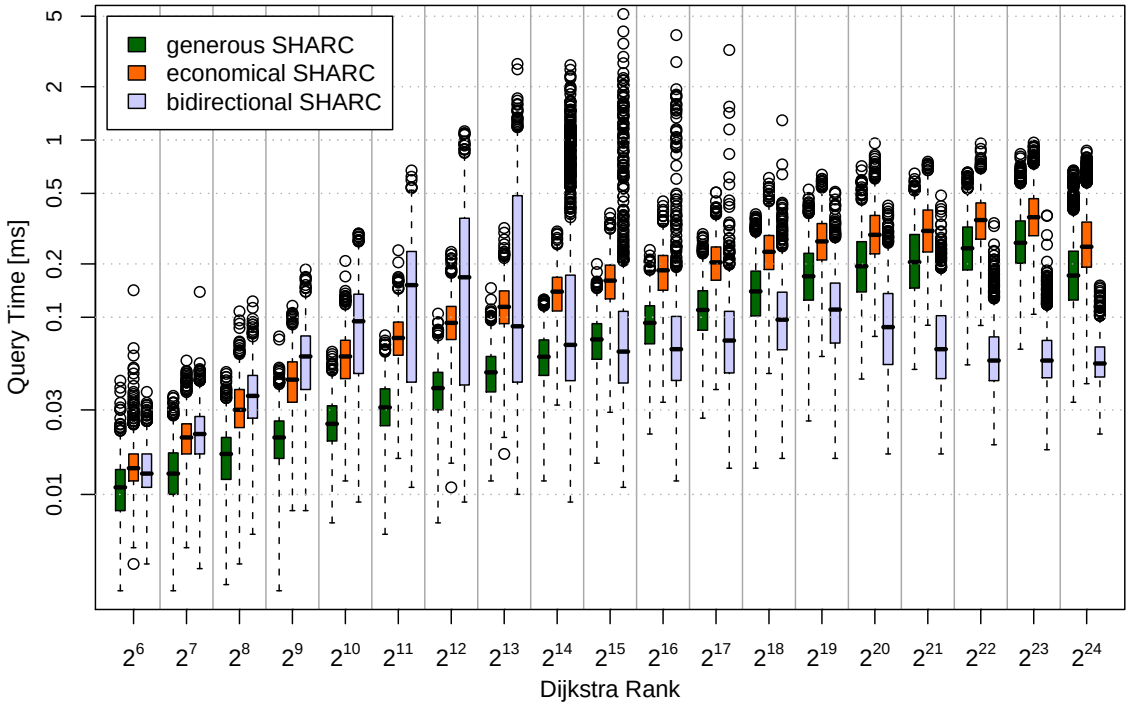


Figure 4.12: Comparison of generous, economical, and bidirectional SHARC using the Dijkstra rank methodology.

Table 4.11: Performance of SHARC on different metrics using the European road instance. *Multi-metric* refers to the variant with one arc-flag and three edge weights (one weight per metric) per edge, while *single* refers to running SHARC on the applied metric.

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

Multi-Metric Queries. The original dataset of Western Europe contains 13 different road categories. By applying different speed profiles to the categories we obtain different metrics. Table 4.11 gives an overview of the performance of (economical) 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. Strikingly, the loss in performance is only very little when storing only one arc-flag for all three metrics. However, the overhead increases due to storing more edge weights for shortcuts and the size of the arc-flags vector increases slightly. Due to the fact that we have to compute arc-flags for all metrics during preprocessing, the computational effort increases.

4.5.4 Hierarchy-Aware Arc-Flags

CHASE. The combination of Contraction Hierarchies and Arc-Flags allows a very flexible trade-off between preprocessing and query performance. The bigger the subgraph H used as input for Arc-Flags, the longer preprocessing takes but query performance improves. Table 4.12 reports the performance of CHASE for different sizes of H in percentage of the original graph. We partition H with SCOTCH [Pel07] into 128 cells. Two observations are remarkable: the effect of stall-on-demand (cf. Section 4.4.1) and the size of the subgraphs. While stall-on-demand pays off for pure CH, CHASE does not win from turning on this optimization. The number of settled nodes decreases but due to the

Table 4.12: Performance of CHASE for Europe with stall-on-demand turned on and off running 10 000 random queries. The search space is given as #settled nodes during phase 1 and in total. The number of entry points is given as well.

	size of H	0.0%	0.5%	1.0%	2.0%	5.0%	10.0%	20.0%
Prepro.	time [min]	25	31	41	62	99	244	536
	space [Byte/n]	-2.7	0.0	1.9	4.9	12.1	22.2	39.5
Query	#settled total	355	86	67	54	43	37	34
(with s-o-d)	#settled phase 1	–	60	41	29	18	13	8
	#entry points	–	21	14	10	7	5	4
	time [μs]	180.0	48.5	36.3	29.2	22.8	19.7	17.2
Query	#settled total	931	111	78	59	45	39	35
(without s-o-d)	#settled phase 1	–	76	47	31	19	13	8
	#entry points	–	30	18	12	8	6	4
	time [μs]	286.3	43.8	30.8	23.1	17.3	14.9	13.0

overhead query times increase. Another very interesting observation is the influence of the input size for arc-flags. Applying goal-direction on a very high level of the hierarchy speeds up the query significantly. A core size of 0.5% already yields an additional speed-up of a factor of 4 combined with an additional preprocessing effort of 6 minutes. Hence, we call this setup our *economical* variant of CHASE. Interestingly, further significant improvements—with respect to query times—are only observable for a core size of up to 5% of the input graph. Here, we achieve query times ≈ 10 times faster than plain CH. Still, 99 minutes of preprocessing is reasonable. Hence, we call this setup our *generous* variant of CHASE. Increasing the size of H to 10% or even 20% yields a much higher preprocessing effort (both space and time) but query performance increases only slightly, compared to 5%. However, our fastest variant settles only 35 nodes on the average having query times of 13.0 μs. Note that for this input, the average shortest path in its contracted form consists of 22 nodes, so only 13 unnecessary nodes are settled on the average.

Local Queries. Like for CALT, Fig. 4.13 reports the query times of economical and generous CHASE and plain CH with respect to the Dijkstra rank. We observe that up to a rank of 2^{14}, all three algorithms yield similar query times. This is expected since up to this rank, most of the queries do not touch the upmost part of the contraction hierarchy and hence, arc-flags do not contribute to the query. Above this rank, query performance gets better again. This effect has been observed for pure Arc-Flags as well [HKMS09]: Long-range queries often relax *only* the shortest path while for low-range queries, the advantage of arc-flags fades. Comparing economical and generous CHASE, we observe that above a rank of 2^{17} the latter is about 2.5 times faster than the former. For very high ranks, generous CHASE is more than an order of magnitude faster than pure CH.

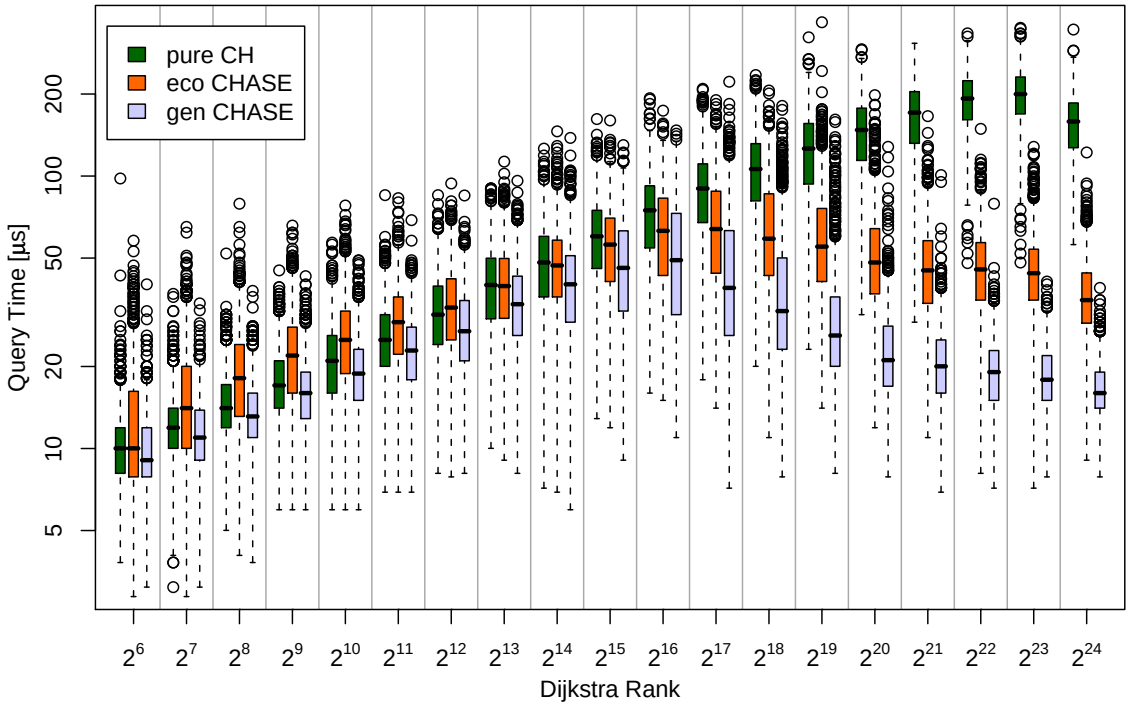


Figure 4.13: Comparison of pure CH, economical and generous CHASE using the Dijkstra rank methodology.

Partial CHASE. Up to now, we evaluated a setup where a complete CH is constructed. However, as discussed in Section 4.4.1, we may stop the construction at some point and compute arc-flags on a flat core. Table 4.13 reports the performance of partial CHASE if 0.5% and 5% of the graph is *not* contracted. For comparison, we report the figures of a partial variant of CH, called pCH. Similar to pCHASE, we stop contraction at some point and perform CH-queries in such a partial hierarchy. Moreover, we report the performance of plain CH and economical CHASE.

Table 4.13: Performance of pCHASE and partial CH. The input is Europe. Note that only 1 000 queries were computed for pCH.

	non-contracted	0.0%		0.5%		5.0%	
	algorithm	CH	eco CHASE	pCH	pCHASE	pCH	pCHASE
Prepro.	time [min]	25	31	19	21	15	31
	space [Byte/n]	-2.7	0	-2.8	-1.6	-2.9	3.6
Query	#settled total	355	86	97 913	2 544	965 018	12 782
	time [μs]	180	44	4 281	831	53 627	4 143

We observe that for road networks, partial variants of CHASE yield worse results than pure CH: With an uncontracted core of 0.5% preprocessing is a little bit faster but for the price of a slow-down of a factor of 4.6 in query performance. Higher uncontracted cores seem even more impractical. The reason for this rather bad performance stems from Contraction Hierarchies. The partial variant of CH yields a very bad query performance which cannot be compensated by arc-flags.

ReachFlags. Table 4.14 gives a similar overview to table 4.12 for ReachFlags, showing the effects of different sizes of the subgraph H. As expected, query times decrease with an increased subgraph. However, adding goal direction via Arc-Flags yields worse additional speed-ups than for CH. Computing flags on the topmost 0.5% of the graph accelerates queries only by a factor of 2. For CH, the corresponding figure is 4. Moreover, since CH is more than one order of magnitude faster than Reach, CHASE is superior to ReachFlags with respect to all relevant figures. Note however that our implementation of Reach that we also use as base for ReachFlags is roughly a factor of 2 slower than the implementation due to [GKW07]. Thus, a further speed-up of a factor of 2 might be possible.

Table 4.14: Performance of ReachFlags for Europe and the US. The results of our implementation of Reach correspond to a size of H of 0.0%.

input		size of H	0.0%	0.5%	1.0%	2.0%	5.0%	10.0%
Europe	Prepro.	time [min]	70	82	105	150	348	710
		space [Byte/n]	21.0	22.9	25.1	28.2	37.0	49.3
	Query	#settled total	7 387	5 454	3 754	2 763	1 101	638
		time [ms]	6.24	4.79	3.12	2.22	0.84	0.48
USA	Prepro.	time [min]	62	89	136	272	671	1 279
		space [Byte/n]	21	20	22	26	35	45
	Query	#settled total	4 261	2 563	1 719	1 339	693	450
		time [ms]	3.90	2.09	1.33	1.01	0.50	0.33

Partial ReachFlags. Table 4.15 reports the performance of partial ReachFlags. We stop reach computation after i iterations, set the reach of remaining nodes to infinity and compute arc-flags for the subgraph induced by these nodes.

Table 4.15: Performance of pReachFlags. The input is Europe. Core nodes indicates how many nodes have a reach value of infinity.

number of iterations		1	2	3	4	all
Prepro.	core-nodes	14.87%	5.32%	1.45%	0.20%	0.00%
	time [min]	400	229	107	69	70
	space [Byte/n]	36	30	25	22	21
Query	#settled total	1 149	1 168	2 797	5 718	7 387
	time [ms]	0.62	0.76	2.24	5.34	6.24

We observe that partial ReachFlags provides better results than pure reach: The less reach values we bound, the better the performance of the algorithm gets. This is due to the fact that we compute arc-flags for a bigger part of the graph. Comparing pReachFlags and pCHASE, it is interesting to note that for core sizes of $\approx 5\%$, pReachFlags outperforms pCHASE, while for core sizes of $\approx 0.5\%$, pCHASE outperforms pReachFlags. It seems as if the loss in performance for cutting a hierarchy based on reach is less than cutting a contraction hierarchy.

TNR+AF. The fastest variant of Transit-node Routing *without* using flag vectors is presented in [GSSD08]; the corresponding figures are quoted in Tab. 4.16. For this variant, we computed flag vectors according to Section 4.4.3 using $k = 48$ regions. This takes, in the case of Europe, about two additional hours and requires 117 additional bytes per node. Then, the average query time is reduced to as little as 1.9 μs, which is an improvement of almost factor 1.8 (factor 2.9 compared to the first publication in [BFM+07]) and a speed-up compared to Dijkstra's algorithm of more than 3 *million*. The results for the US are even better.

Table 4.16: Overview of the performance of Transit-Node Routing with and without additional arc-flags. For pure TNR, we report two figures. The first is due to [BFM+07] and based on Highway Hierarchies, while numbers for a TNR-implementation based on Contraction Hierarchies are given in [GSSD08]. Note that these results were conducted on a slightly different machine.

	Europe			USA		
	PREPRO.		QUERY	PREPRO.		QUERY
	time	overhead	time	time	overhead	time
method	[min]	[B/node]	[μs]	[min]	[B/node]	[μs]
HH-TNR	164	251	5.6	205	244	4.9
CH-TNR	112	204	3.4	90	220	3.0
CH-TNR+AF	229	321	1.9	157	263	1.7

The improved running times result from the reduced number of table accesses: in the case of Europe, on average only 3.1 entries have to be looked up instead of 40.9 when no flag vectors are used. Note that the runtime improvement is considerably less than a factor of 40.9 / 3.1 = 13.2 though. This is due to the fact that the average runtime also includes looking up the access nodes and dealing with local queries.

4.5.5 Comparison

Table 4.17 reports the performance of our new combinations in comparison to existing speed-up techniques.

CALT. By improving the organization of landmark data (cf. Section 3.2), we obtain a better query performance for ALT than reported in [GW05]. However, we do not compress landmark information and use a slightly better heuristic for landmark selection. Hence, we report both results. By adding contraction to ALT, we are able to reduce query times to 1.3 ms for Europe and to 3.0 ms for the US. This better performance is due to two facts. On the one hand, we may use more landmarks (we use 64) and on the other hand, the contraction reduces the number of hops of shortest paths. Moreover, the most crucial drawback of ALT—memory consumption—can be reduced to a reasonable amount, even when using 64 landmarks. Still, CALT cannot compete with REAL or pure hierarchical methods, but the main motivation for CALT is its easy adaptability to dynamic and time-dependent scenarios (cf. Section 5.3).

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

SHARC settles roughly the same number of nodes as Highway Hierarchies or REAL, but query times are smaller. This is due to the very low computational overhead of SHARC. Regarding preprocessing, SHARC uses less space than REAL or Highway Hierarchies. The computation time of the preprocessing is similar to REAL but longer than for Highway Hierarchies. The bidirectional variant uses more space and has longer preprocessing times, but the performance of the query is very good. Compared to the classic Arc-Flags, SHARC significantely reduces preprocessing time and query performance is better. Still, SHARC is the only high-performance unidirectional speed-up technique making usage in time-dependent networks easier (cf. Section 5.4).

CHASE. We report the figures for *economical* and *generous* CHASE. For Europe, the economical variant only needs 7 additional minutes of preprocessing over pure CH and the preprocessed data is still smaller than the input. Recall that a negative overhead derives from the fact that the search graph is smaller than the input, see Section 1.2. This economical variant is already roughly 4 times faster than pure CH. However, by increasing the size of the subgraph H used as input for arc-flags, we are able to almost close the gap to pure Transit-Node Routing. CHASE is only 5 times slower than TNR (and is even *faster* than the grid- and separator-based approach of TNR [BFM+07, DHM+09]). However, the preprocessed data is much smaller for CHASE, which makes it more practical in environments with limited memory. Moreover, it seems as if CHASE can be adapted to time-dependent scenarios easier than TNR [BDSV09].

Table 4.17: Overview of the performance of various speed-up techniques, grouped by (1.) hierarchical methods [Highway Hierarchies (HH), highway-node routing (HNR), Contraction Hierarchies (CH), Transit-Node Routing (TNR), High-Performance Multi-Level Routing (HPML)], (2.) goal-directed methods [landmark-based A^* search (ALT), Arc-Flags (AF)], (3.) previous combinations, and (4.) the new combinations introduced in this thesis. The additional overhead is given in bytes per node in comparison to *bidirectional* Dijkstra. Preprocessing times are given in minutes. Query performance is evaluated by the average number of settled nodes and the average running time of 10 000 (1 000 for pCH) random queries. Each column highlights the best result in bold. In addition, *Pareto optimal* speed-up techniques are also printed in bold. Note that the Pareto optima are the same for both road networks.

		Europe				USA			
		PREPRO.		QUERY		PREPRO.		QUERY	
method	source	time [min]	space [B/n]	#settled nodes	time [ms]	time [min]	space [B/n]	#settled nodes	time [ms]
Reach	[GKW07]	83	17	4 643	3.47	44	20	2 317	1.81
Reach	4.4.2	70	21	7 387	6.24	62	21	4 261	3.90
HH	[Sch08]	13	48	709	0.61	15	34	925	0.67
HNR	[Sch08]	15	2.4	981	0.85	16	1.6	784	0.45
CH	[GSSD08]	25	-2.7	355	0.18	27	-2.3	278	0.13
HPML	[DHM+09]	≈1 440	208	N/A	0.0188	≈2 160	260	N/A	0.0193
grid TNR	[BFM+07]	-	-	-	-	1 200	21	N/A	0.063
TNR	[BFM+07]	164	251	N/A	0.0056	205	244	N/A	0.0049
TNR	[GSSD08]	112	204	N/A	0.0034	90	220	N/A	0.0030
ALT-a16	[GKW07]	13	70	82 348	160.3	19	89	187 968	400.5
ALT-m16	4.1	85	128	74 669	53.6	103	128	180 804	129.3
ALT-a64	4.1	68	512	25 234	19.6	93	512	68 861	48.9
AF	[HKMS09]	2 156	25	1 593	1.1	1 419	21	5 522	3.3
REAL	[GKW07]	141	36	679	1.11	121	45	540	1.05
HH*	[DSSW09b]	14	72	511	0.49	18	56	627	0.55
SHARC	4.3	81	14.5	654	0.29	58	18.1	865	0.38
SHARC bidir.	4.3	158	21.0	125	0.065	158	21	350	0.18
CALT	4.2	**11**	15.4	1 394	1.34	**13**	16.1	2 697	3.01
pCH-0.5%	4.4.1	19	-2.8	97 913	4.28	21	-2.3	121 636	50.57
pCH-5.0%	4.4.1	15	**-2.9**	965 018	53.63	15	**-2.3**	1 209 290	667.80
eco CHASE	4.4.1	32	0.0	111	0.044	36	-0.8	127	0.049
gen CHASE	4.4.1	99	12	**45**	0.017	228	11	**49**	0.019
pCHASE-0.5%	4.4.1	21	-1.6	2 544	0.83	25	-1.5	4 693	1.40
pCHASE-5.0%	4.4.1	31	3.6	12 782	4.14	96	4.3	22 436	7.21
ReachFlags	4.4.2	348	37	1 101	0.84	671	35	693	0.50
pReachFlags	4.4.2	229	30	1 168	0.76	318	25	1 636	1.02
TNR+AF	4.4.3	229	321	N/A	**0.0019**	157	263	N/A	**0.0017**

ReachFlags. As already mentioned, our reach implementation yields worse results than the numbers reported in [GKW07]. Hence, we report both results. By adding arc-flags to reach we obtain query times comparable to REAL. However, preprocessing takes a little bit longer. Still, it seems as if ReachFlags is inferior to CHASE which is mainly due to the good performance of Contraction Hierarchies.

Summary. We observe that the best results for each measured performance criterion is obtained by one of our newly introduced speed-up techniques. In addition, we see that almost all of our techniques are *Pareto optimal*. A speed-up technique is called *Pareto optimal* if there is no other technique that is better in all measured variables. Thus, each of them is the optimal choice for a specific task with regards to the analyzed algorithms. Only our variants of ReachFlags and pCHASE with a larger core size fall short in this aspect.

4.6 Experimental Study on Robustness

In the last section we focused on the performance of our methods on road networks with travel times. However, speed-up technique should not be tailored to specific inputs. In order to evaluate the robustness of existing and new speed-up techniques, we here present an extensive experimental study on different types of inputs. More precisely, we evaluate different metrics for road networks, graphs deriving from railway optimization, sensor networks, and synthetic grid graphs.

Default Settings. Unless otherwise stated, we use the following settings for existing speed-up techniques. For ALT, we use 16 *maxCover* landmarks. In our Arc-Flag setup, we use 128 cells obtained from METIS. In addition, we evaluate the hierarchical RE algorithm, Highway Hierarchies (HH), and Contraction Hierarchies (CH). The performance of all approaches highly depends on the chosen preprocessing parameters which we here tune manually. For HH, we use a distance table as soon as the contracted graph has less than 10 000 nodes. CH uses its default aggressive parameters [GSSD08]. Moreover, we evaluate the combination of RE and ALT, named REAL, *without* reach-aware landmarks [GKW07]. For our new combinations, we apply the default parameter settings found in Section 4.5.

4.6.1 Metrics in Road Networks

First, we evaluate the impact of a metric change on speed-up techniques. As inputs we use our road networks from the last section but apply travel distances instead of travel times. We hereby want to test whether the speed-up techniques rely on the topology of the network or the speed-up derive from the used metric. Interestingly, plain Dijkstra settles the same number of nodes for each metric but query times vary heavily when switching metrics: Dijkstra's algorithm is two times faster on the distance metric than on travel times. This derives from the number of DECREASEKEY operations of the used priority queue. The results for speed-up techniques can be found in Tab. 4.18.

We observe that all hierarchical methods are slower on travel distances than on travel times. The reason is that the advantage of taking motorways fades when switching the metric. Often, a path across fields is shorter than the one via the highway. Hence, the incorporated hierarchy is less developed for travel distances than for travel times. While

Table 4.18: Overview of the performance of prominent speed-up techniques and combinations analogous to Tab. 4.17 but with travel distances as metric.

		Europe				USA			
		PREPRO.		QUERY		PREPRO.		QUERY	
		time	space	#settled	time	time	space	#settled	time
method		[min]	[B/n]	nodes	[ms]	[min]	[B/n]	nodes	[ms]
Reach	[GKW07]	49	15	7 045	5.53	70	22	7 104	5.97
HH	[Sch08]	32	36	3 261	3.53	38	66	3 512	3.73
CH	4.4.1	89	**-0.1**	1 650	4.19	57	**-1.2**	953	1.50
TNR	[Sch08]	162	301	N/A	**0.038**	217	281	N/A	0.086
ALT-a16	[GKW07]	**10**	70	240 750	430.0	**15**	89	276 195	530.4
ALT-m16	4.1	70	128	218 420	127.7	102	128	278 055	166.9
AF	[HKMS09]	1 874	33	7 139	5.0	1 311	37	12 209	8.8
REAL	[GKW07]	90	37	583	1.16	138	44	628	1.48
HH*	[DSSW09b]	33	92	1 449	1.51	40	89	1 372	1.37
SHARC	4.3	64	19	3 014	1.34	75	20	3 871	1.78
CALT	4.2.2	14	19	2 958	4.2	15	19	4 015	5.6
eco CHASE	4.4.1	224	7.0	175	0.156	185	2.5	148	0.103
gen CHASE	4.4.1	1 022	27	**67**	0.064	1 132	18	**63**	**0.043**
pCHASE-0.5%	4.4.1	40	1.9	5 957	2.61	43	0.3	8 276	3.17
pReachFlags	4.4.2	516	31	5 224	4.05	1 897	27	6 849	4.69

a performance loss is expected for hierarchical methods, Tab 4.18 clearly indicates that goal-directed techniques perform worse on travel distances as well. For Arc-Flags, this derives from the fact that flags for the paths across fields are set to true for this metric while they are set to false for travel times.

Combinations of techniques do not perform much worse on travel distances than on travel times. The REAL algorithm for example settles roughly the same number of nodes on both metrics and even has a *lower* preprocessing time. For CHASE, we observe that the combination yields excellent query times on this metric as well. This is especially interestingly since both pure Arc-Flags and CH are 5 and 20 times slower on travel distance whereas the combination of both approaches, yields similar query times. The reason for this is the following. For pure CH, edge reduction works worse on travel distances yielding higher degrees for high-level nodes. By applying Arc-Flags on this rather dense core, a lot of edge relaxations can be avoided. This also explains the highly increased preprocessing times of CHASE: The core is denser making arc-flags computation more time-consuming. Hence, partial CHASE is more promising on this input than on travel times as metric. We observe that preprocessing times are faster than for pure CH combined—at least for Europe—with better query times.

Concerning preprocessing times, CALT outperforms any other technique (except ALT) combined with reasonable query times. We conclude that CALT indeed is almost as robust as pure ALT with respect to metric changes.

4.6.2 Railway Networks

Our second set of experiments is executed on three simple time-expanded graphs (cf. Section 2.4). The first shows the local traffic of Berlin/Brandenburg, has 2 599 953 nodes and 3 899 807 edges, the second one represents local traffic of the Ruhrgebiet (2 277 812

nodes, 3 416 597 edges), and the last graph depicts long distance connections of Europe (1 192 736 nodes, 1 789 088 edges). Table 4.19 gives an overview of the performance of speed-up techniques on these instances.

Table 4.19: Performance of speed-up techniques on simple time-expanded railway networks.

	Berlin/Brandenburg				Ruhrgebiet				long distance			
	PREPRO		QUERY		PREPRO		QUERY		PREPRO		QUERY	
	time [min]	space [B/n]	#sett. nodes	time [ms]	time [min]	space [B/n]	#sett. nodes	time [ms]	time [min]	space [B/n]	#sett. nodes	time [ms]
Dijkstra	0	0	1299830	406.2	0	0	1134420	389.2	0	0	609352	221.2
BiDijkstra	0	0	496281	151.3	0	0	389577	122.8	0	0	143613	43.8
uni ALT	10	128	383921	133.6	10	128	171760	64.7	5	128	71194	26.0
ALT	10	128	47764	22.9	10	128	59516	30.5	5	128	31367	15.0
uni AF	2240	24	172362	72.2	2323	24	158174	66.4	1008	24	74737	32.4
AF	4479	48	24004	9.2	4646	48	28448	10.7	2016	48	10560	3.5
RE	182	39	27095	25.5	290	45	38397	39.8	63	43	8978	8.3
HH	38	263	5285	56.1	65	202	9528	196.2	12	386	1930	7.3
CH	27	0	416	0.4	43	4	546	0.6	8	3	376	0.3
uni REAL	192	167	20062	22.2	300	173	16649	21.1	68	171	6335	8.8
REAL	192	167	4159	6.6	300	173	7867	13.3	68	171	2479	4.5
CALT	2	45	2830	6.3	3	68	4247	11.3	1	63	1088	5.3
SHARC	602	9	11006	3.8	615	8	12412	4.2	209	15	7519	2.2
CHASE	33	2	125	0.1	48	7	244	0.2	9	5	299	0.2
pCHA-0.5%	22	2	4492	2.1	30	6	8209	4.5	6	4	14482	1.6
pCHA-5.0%	260	7	18698	7.2	169	13	20224	8.0	18	11	8985	3.2

Note that bidirectional approaches cannot be used out-of-the-box for time-expanded networks. In order to gain insights in the performance of these techniques, we also use bidirectional speed-up techniques by picking a random event at the target station. Thus, these bidirectional experiments are intended to give hints whether it is worth focusing on adapting bidirectional search to such graphs. Only SHARC and unidirectional Arc-Flags—with a partitioning by station—are applicable. Arc-Flags perform roughly 12-18 times faster than unidirectional Dijkstra. But when switching to bidirectional search we gain another speed-up of factor 6-10. Thus, it may be worth focusing on the question how to use bidirectional search in this scenario. However, we observe very long preprocessing times for Arc-Flags on these networks. The situation changes for CHASE, preprocessing times are within reasonable times and query performance is the best of all applied speed-up techniques. RE seems to have problems on the local traffic networks as preprocessing takes longer than 3 hours and speed-ups are only mild, while this does not hold for long distance connections. Regarding query times, HH has also problems with both local traffic networks: on Berlin/Brandenburg, HH is only 3 times faster than bidirectional Dijkstra, and on the Ruhrgebiet, HH is even slower. Interestingly, this does not derive from a missing hierarchy within the network: CH preforms very well on all three inputs. However, CH and CHASE rely on bidirectional which needs further adaption to provide feasible results for these inputs. Still, it may be expected that an adapted technique might work very well on these inputs.

4.6.3 Sensor Networks

At a glance, routing in sensor networks has similar properties as routing in road networks. Thus, we evaluate so called unit-disc graphs which are widely used for experimental evaluations in that field. Such graphs are obtained by arranging nodes on the plane and connecting nodes with a distance below a given threshold. It is obvious that the density can be varied by applying different threshold values. In our setup, we use graphs with about 1 000 000 nodes and an average degree of 5, 7, and 10, respectively. As metric, we use the distance between nodes according to their embedding. The results can be found in Tab. 4.20. Uni- and bidirectional Dijkstra settle roughly the same number of nodes independent of the average degree but query times again increase with higher density due to more relaxed edges. Analyzing ALT, the bidirectional variant is twice as fast as the unidirectional algorithm for the instance with degree 5 while for degree 10, both approaches are equal to each other with respect to query times. The decreasing search space of unidirectional ALT is due to the increasing number of edges. With more edges, the shortest path is very close to the flight distance between source and target. In such instances, the potentials deriving from landmarks are very good. Again adding contraction to ALT yields good results. Preprocessing times stay little and CALT outperforms ALT on all inputs. Arc-Flags yield very good query times but again for the price of high preprocessing times. Hierarchical methods work very good on average degrees of 5 and 7. For a degree of 10 preprocessing and query times increase drastically. For RE, a reason is that node-labels are used for pruning the search. With increasing density, many edges are never used by any shortest path. As these edges cannot be pruned by using node-labels, query times increase. CHASE outperforms any technique with less space and very small preprocessing times. However, the gap between CALT and CHASE shrinks the denser the graph gets. Summarizing, CHASE performs best on these inputs. Only for higher densities, CALT yields lower preprocessing times but still, CHASE has better query performance.

Table 4.20: Performance of speed-up techniques on unit-disc graphs with different average degree.

	average deg. 5				average deg. 7				average deg. 10			
	Prepro		Query		Prepro		Query		Prepro		Query	
	time [min]	space [B/n]	#sett. nodes	time [ms]	time [min]	space [B/n]	#sett. nodes	time [ms]	time [min]	space [B/n]	#sett. nodes	time [ms]
Dijkstra	0	0	487818	257.3	0	0	521874	330.1	0	0	502683	399.0
BiDijkstra	0	0	299077	164.4	0	0	340801	225.1	0	0	325803	269.4
uni ALT	8	128	22 476	17.1	8	128	16 634	15.1	10	128	14 561	16.0
ALT	8	128	9 222	8.5	8	128	10 565	11.8	10	128	11 749	15.6
uni Arc-Flags	53	80	8 556	7.9	299	112	16 445	16.8	801	160	21 413	24.2
Arc-Flags	105	160	2 091	1.8	598	224	4 761	4.6	1 602	320	7 019	7.5
RE	4	20	848	0.5	46	42	13 783	14.3	1 153	54	83 826	104.5
HH	2	251	203	0.2	12	549	5 068	8.5	71	690	23 756	49.1
CH	2	-13	236	0.13	21	-11	1 089	1.80	571	-4	2 475	11.5
uni REAL	12	148	307	0.4	54	170	2 072	3.2	1 163	182	8 780	13.6
REAL	12	148	291	0.4	54	170	2 394	4.1	1 163	182	11 449	21.7
CALT	1	7	689	0.5	2	29	670	1.0	9	137	992	2.6
SHARC	1	16	568	0.3	10	42	1 835	1.0	70	96	4 972	3.6
eco CHASE	2	-11	66	0.04	23	-3	424	0.57	578	15	1 457	4.7
gen CHASE	3	-2	43	0.02	52	27	112	0.12	924	85	293	0.53
pCHASE-10%	2	-2	36 473	1.2	12	10	5 677	2.7	69	35	7 529	2.7

4.6.4 Grid Graphs

Our last testset exploits the influence of graph diameter on the performance. Here, we vary the diameter of a graph by using multi-dimensional grid graphs with 2, 3, and 4 dimensions. The number of nodes is set to 250 000, and thus, the number of edges is 1, 1.5, and 2 million, respectively. Edge weights are picked uniformly at random from 1 to 1000. These results can be found in Tab. 4.21. Like for sensor networks, unidirectional Dijkstra settles the same number of nodes on all graphs. But due to more relaxed edges query times increase with an increasing number of dimensions. As the diameter shrinks with increasing number of dimensions, bidirectional Dijkstra settles less nodes on 4-dimensional grids than 2-dimensional grids. This analysis also holds for the performance of uni- and bidirectional ALT. For these inputs, it seems as if adding contraction to ALT does not really pay off. Query times are similar but preprocessing space increases. The reason for this is that the core is quite big. Storing distances to 64 landmarks on a big core requires a lot of space. The reason for the low speed-ups is that the number of entry points to the core is very high. Our hierarchical representatives RE/HH/CH perform very good on 2-dimensional grids but significantely lose performance when switching to higher dimensions. The main reason is that the contraction phase of the algorithms fail. Summarizing, ALT and CALT have the best trade-off with respect to preprocessing and query times on higher-dimensional grids. Only Arc-Flags is faster for the price of a much higher effort in preprocessing. Hierarchical methods can only compete with ALT and CALT on 2-dimensional grids.

Table 4.21: Performance of speed-up techniques on the grid graphs with different numbers of dimensions.

	2-dimensional				3-dimensional				4-dimensional			
	Prepro		Query		Prepro		Query		Prepro		Query	
	time [min]	space [B/n]	#sett. nodes	time [ms]	time [min]	space [B/n]	#sett. nodes	time [ms]	time [min]	space [B/n]	#sett. nodes	time [ms]
Dijkstra	0	0	125 675	36.7	0	0	125 398	78.6	0	0	122 796	137.5
BiDijkstra	0	0	79 962	24.2	0	0	45 269	28.2	0	0	21 763	20.3
uni ALT	1	128	5 452	2.5	2	128	4 223	3.8	3	128	5 031	7.5
ALT	1	128	2 381	1.5	2	128	1 807	2.2	3	128	1 329	2.5
uni AF	45	64	4 476	1.9	415	94	8 996	5.7	1 559	122	25 125	26.8
AF	89	128	1 340	0.6	830	189	1 685	1.0	3 117	244	2 800	2.3
RE	13	31	3 797	2.1	220	102	18 177	27.1	2 243	89	20 587	40.2
HH	2	1682	583	0.6	32	1954	17 243	95.8	680	662	61 715	343.0
CH	1	0	418	0.3	226	15	2 177	6.6	2 229	29	14 501	60.0
uni REAL	14	159	799	0.8	222	230	5 081	10.6	2 246	217	10 740	30.3
REAL	14	159	829	0.9	222	230	3 325	8.5	2 246	217	3 250	11.6
CALT	1	211	458	1.1	2	386	557	1.9	2	487	774	2.2
SHARC	32	60	1 089	0.4	62	97	5 839	1.9	292	13	20 115	11.5
eco CHASE	1	4	274	0.21	226	29	2 836	10.2	2 230	35	30 848	131.0
gen CHASE	2	24	101	0.07	244	113	772	1.3	2 260	106	29 811	124.8
pCHASE-10%	1	15	1 967	0.5	25	57	10 788	7.5	-	-	-	-
pCHASE-20%	2	26	3 063	0.8	31	69	10 052	5.2	482	208	31 384	52.1
pCHASE-50%	4	55	5 964	1.5	50	95	13 402	5.7	441	279	36 473	33.0

4.7 Concluding Remarks

Review. In this chapter, we showed that landmarks can be used out of the box in dynamic scenarios. By adding contraction to ALT, we remedied most of the drawbacks of pure ALT without losing the advantages. We also 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. Hence, it seems a good choice for augmentation.

Moreover, we systematically combined hierarchical techniques with arc-flags. As a result we are able to present the fastest algorithms for several (time-independent) scenarios and inputs. For sparse graphs, CHASE yields excellent speed-ups with low preprocessing effort. The algorithm is only overtaken by Transit-Node Routing in road networks with travel times, but the gap is almost closed. However, even Transit-Node Routing can be further accelerated by adding goal-direction. For denser inputs, we present a partial variant of CHASE which stops the hierarchy construction at a certain point and runs a pure goal-directed query on the core.

Concerning speed-up techniques in general, we gained further and interesting insights by our extensive experimental study. Hierarchical approaches seem to have problems with high-density networks, the chosen metric has a high impact on achieved speed-ups, edge-labels are somewhat superior to node-labels, and small diameters yield big speed-ups for bidirectional search. As a consequence, the choice of which technique to use highly depends on the scenario. However, of all examined speed-up techniques, ALT, CALT, and partial CHASE provide a reasonable trade-off on preprocessing time and space on the one hand and achieved speed-up on the other hand. Although these approaches are slower on hierarchical inputs they are more *robust* with respect to the input. Still, SHARC performs very well on most inputs, although it is a unidirectional technique.

Future Work. It seems as if route planning in time-independent networks has arrived at a final point. We have techniques with low memory requirements and excellent query times not only for road networks. However, some problems persists. Most importantly, updating preprocessing of SHARC is an interesting question. While updating the shortcuts seems possible by adapting ideas from [Sch08], updating arc-flags is non-trivial. The most challenging task however, i.e., the adaption of techniques to augmented scenarios, is dealt with in Chapters 5 and 6.

References. This chapter is based on [DW07, BDW07a, BD08, BDS+08] and the corresponding accepted [BDW09, BD09] and submitted [BDS+09] journal versions.

Chapter 5

Time-Dependent Route Planning

A major drawback of most existing speed-up techniques, including the ones from Chapter 4, is that their correctness depends on the fact that the network is static, i.e., the network does *not* change between queries. Only the preprocessed data of ALT (cf. Chapter 4.1) and Highway-Node Routing [SS07] can be updated if a road network is perturbed by a relatively small number of traffic jams. However, none of the techniques developed during the last years can be adapted to *time-dependent* networks easily.

In this chapter, we show how some of the speed-up techniques from Chapter 4 can be augmented for exact time-dependent routing in road and railway networks. However, in road networks, time-dependent data is based on historical data. As a result, travel times on roads for specific departure times only approximate the current traffic situations. So, it makes sense to evaluate speed-up techniques that find paths that are slightly longer than the shortest, which we do in this chapter as well.

Overview. This chapter is organized as follows. We augment the ingredients of our time-dependent speed-up techniques in Section 5.1. It turns out that landmark preprocessing stays correct and contraction can be augmented in a straightforward manner, but for a price of high memory consumption. Setting arc-flags in time-dependent networks gets more expensive in terms of preprocessing times. Hence, we introduce several ways to efficiently approximate arc-flags. Section 5.2 shows how bidirectional ALT can be applied in time-dependent networks anyway. We therefore perform a backward search that bounds the set of nodes that need to be explored by the forward search. Similar to Section 4.2, we enrich this approach by contraction in Section 5.3. Our augmented version of SHARC is introduced in Section 5.4 where we put the augmented ingredients together to an efficient time-dependent speed-up technique. All three approaches are extensively evaluated in Section 5.5 with real-world transportation networks of Germany and Europe. We conclude our work on time-dependent route planning by a summary and a discussion on future work in Section 5.6.

5.1 Augmenting Ingredients

Analyzing the speed-up techniques from Chapter 4, we observe that the preprocessing of most techniques relies on the following ingredients: Local Dijkstra-searches, arc-flags computation, landmarks, and contraction. In this section we show how to augment all these ingredients such that correctness is guaranteed in a time-dependent scenario.

5.1.1 Dijkstra

Computing $d(s,t,\tau)$ can be solved by a modified Dijkstra [CH66]: when relaxing an edge (u,v) we have to evaluate its weight for departure time $\tau + d(s,u,\tau)$. In our scenario, the running time for evaluating functions is negligible, hence the additional effort for respecting the departure time is negligible as well. However, computing $d_*(s,t)$ is more expensive but can be computed by a label-correcting algorithm [Dea99]. Such an algorithm can be implemented very similarly to Dijkstra. The source node s is initialized with a constant label $d_*(s,s) \equiv 0$, any other node u with a constant label $d_*(s,u) \equiv \infty$. Then, in each iteration step, a node u with minimum $\underline{d_*}(s,u)$ is removed from the priority queue. Then for all outgoing edges (u,v) a temporary label $l(v) = d_*(s,u) \oplus len(u,v)$ is created. If $l(v) \geq d_*(s,v)$ does *not* hold, $l(v)$ yields an improvement. Hence, $d_*(s,v)$ is updated to $\min\{l(v), d_*(s,v)\}$ and v is inserted into the queue. We may stop the routine if we remove a node u from the queue with $\underline{d}(s,u) \geq \overline{d}(s,t)$. If we want to compute $d_*(s,t)$ for many nodes $t \in V$, we apply a label-correcting algorithm and stop the routine as soon as our stopping criterion holds for all t. Note that we may reinsert nodes into the queue that have already been removed by this procedure. Also note that when applied to a graph with constant edge-functions, this algorithm equals a normal Dijkstra. An interesting result from [Dea99] is the fact that the runtime of label-correcting algorithms highly depends on the complexity of the edge-functions. This is confirmed by our experiments (see Section 5.5).

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

Bidirectional Profile Search. As already mentioned, bidirectional search is prohibited for time-queries as the arrival time is unknown. However, we can directly apply bidirectional search for profile-queries since we investigate *all* arrival times. Compared to a time-independent bidirectional Dijkstra, we only need to adjust the stopping criterion. Stop the search if the lower bound of the minimum label in the forward queue added to the lower bound of the minimum label in the backward queue is larger than the upper bound of the tentative distance label.

5.1.2 Landmarks

Based on the ideas from 4.1.2, we can adapt a unidirectional variant of the ALT algorithm to the time-dependent scenario: We perform both landmark selection and distance computation in the lower bound graph $\underline{G}$. It is obvious that we obtain a feasible potential. However, ALT implemented as bidirectional search is much faster than the unidirectional variant. As already mentioned, performing a bidirectional search in time-dependent network is non-trivial. A possible approach is the topic of Section 5.2.

Note that the time-dependent ALT algorithm also works in a dynamic time-dependent scenario. Using the same arguments from Section 4.1.2, the algorithm still performs accurate queries as long as edge weights do not drop below their lower bound. If this happens, the distance labels can be updated using the routine from Section 4.1.1.

5.1.3 Arc-Flags

In time-independent scenarios, a set arc-flag $AF_C(e)$ denotes whether e has to be considered for a shortest-path query targeting a node within C. In other words, the flag is set if e is important for (at least one target node) in C. In a time-dependent scenario, we use the following intuition to set arc-flags: an arc-flag $AF_C(e)$ is set to true, if e is important for C at least once during Π. A straightforward adaption of computing arc-flags in a time-dependent graph is to construct a profile graph in $\overleftarrow{G}$ for all boundary nodes $b \in B_C$ of all cells C. Then we set $AF_C(u, v) = \mathsf{true}$ if (u, v) is a PG-edge for at least one PG built from all boundary nodes $b \in B_C$. See Figure 5.1 for an example. In addition, we also set all own-cell flags to true as well. The time-dependent query is a normal time-dependent Dijkstra only relaxing edges with set flag for the target's region.

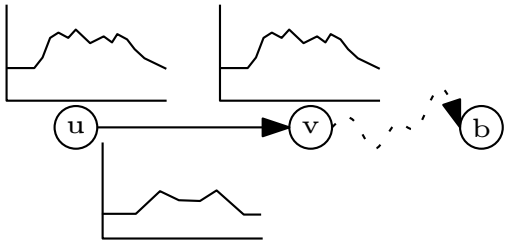


Figure 5.1: Computation of time-dependent arc-flags. By construction of a profile graph from the boundary node b, we end up in two distance labels $d_*(u, b)$ and $d_*(v, b)$. If $len(u, v) \oplus d_*(v, b) > d_*(u, b)$ does not hold, (u, v) is a PG edge (with respect to b) and hence gets the arc-flag for $\mathbb{c}(b)$ assigned true.

Lemma 5.1. *Time-dependent Arc-Flags is correct.*

Proof. To show correctness of time-dependent Arc-Flags, we have to prove that for each shortest s–t path $p^\tau_{st} = (e_0, \ldots, e_k), \tau \in \Pi$ the following condition holds: $AF_T(e_i) = \mathsf{true}, 0 \leq i \leq k$ with $T = \mathbb{c}(t)$. For all edges $e_i = (u_i, v_i)$ with $\mathbb{c}(u_i) = \mathbb{c}(v_i) = \mathbb{c}(t)$ this holds because we set own-cell flags to true. Let s and t be arbitrary nodes, and let τ be an arbitrary departure time. In addition, let $e_i = (u_i, v_i) \in p^\tau_{st}, \mathbb{c}(u_i) \neq \mathbb{c}(t), \mathbb{c}(v_i) \neq \mathbb{c}(t)$, and b_T be the last boundary node of region T on p^τ_{st}. We know that the subpath from s to b_T is a shortest path (for departure time τ). Assume $AF_T(e_i) = \mathsf{false}$. Since $AF_T(e_i) = \mathsf{false}$ holds, $d_*(u_i, b_T) \oplus (u_i, v_i) > d_*(v_i, b_T)$ must hold as well. This is a contradiction since e_i is part of the shortest path from s to b_T. □

Approximation. Computing arc-flags as described above requires to build a complete profile graph on the backward graph from each boundary node yielding too long preprocessing times for large networks. Recall that the running time of building PGs is dominated by the complexity of the function. Hence, we may construct two PGs for each boundary node, the first uses $\uparrow len$ as length functions, the latter $\downarrow len$. As we use approximations with a constant number of interpolation points, constructing two such PGs is faster than building one exact one. We end up in two distance labels per node u, one being an overapproximation, the other being an underapproximation of the correct label. Then, for each $(u, v) \in E$, we set $\overline{AF}_C(u, v) = \mathsf{true}$ if $len(u, v) \oplus \uparrow d_*(v, b_C) > \downarrow d_*(u, b_C)$ does not hold. See Figure 5.2 for an example.

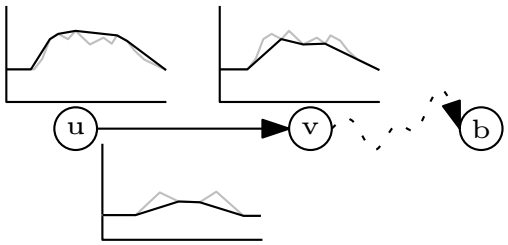


Figure 5.2: Approximation of time-dependent arc-flags via functions. The original functions are drawn in gray. By using over- and underapproximations of len during construction of the profile graphs, we end up in approximated distance labels for u and v. Then, we set the arc-flag of (u, v) to true if $\downarrow \overline{len}(u, v) \oplus \downarrow d_*(v, b_C) > \uparrow \underline{d_*}(u, b_C)$ does not hold.

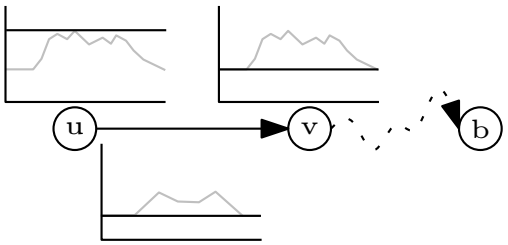


Figure 5.3: Approximation of time-dependent arc-flags via bounds. The original functions are drawn in gray. Unlike for approximation via functions, we use bounds for approximations. We set the arc-flag of (u, v) to true if $\underline{len}(u, v) + \underline{d_*}(v, b_C) \leq \overline{d_*}(u, b_C)$ holds.

If networks get so big that even setting approximate labels is prohibited due to running times, one can even use upper and lower bounds for the labels. This has the advantage that building two shortest-path trees per boundary node is sufficient for setting correct arc-flags. The first uses $\overline{len}$ as length function, the other $\underline{len}$. See Figure 5.3 for an example. Note that by approximating arc-flags (denoted by $\overline{AF}$), the quality of them may decrease but correctness is untouched. Thus, queries remain correct by may become slower.

Lemma 5.2. *Approximate Arc-Flags is correct.*

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

Heuristic Arc-Flags. Analyzing both approaches for computing arc-flags, exact and approximate, we observe the following. Exact flags yield excellent query times (cf. Section 5.5) but preprocessing is time consuming. On the contrary, approximate flags yield low preprocessing times but query performance is much worse than for exact flags.

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

As already mentioned, this approach may yield incorrect queries as a shortest path for a specific departure time may have been missed. However, it is obvious that a path is found since at least for one departure time, flags are set to true for a shortest path to the target's region. We evaluate the error-rate in Section 5.5.

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

Lemma 5.3. *Time-Dependent Multi-Level Arc-Flags is correct.*

Proof. In the following, we show the correctness of two-level Arc-Flags. The generalization to a multi-level scenario is straightforward.

Let $p^\tau_{st} = (e_0, \ldots, e_k), \tau \in \Pi$ be an arbitrary s–t shortest path with arbitrary departure time τ. Let $\mathbb{c}_i(u)$ be the cell of u in level i, where 0 denotes the lower, 1 the upper level. An edge (u, v) is part of the upper level if $\mathbb{c}_1(u) \neq \mathbb{c}_1(t)$ and $\mathbb{c}_1(v) \neq \mathbb{c}_1(t)$. According to Lemma 5.1, we know that all edges being part of the upper level have $AF_{\mathbb{c}_1(t)} = \mathsf{true}$. Let b be the last boundary node of $\mathbb{c}_0(t)$ on p^τ_{st}. Since we have built a profile graph from b during preprocessing until all nodes in $\mathbb{c}_1(t)$ have their final label assigned, edges being part of the lower level have proper arc-flags assigned. □

5.1.4 Contraction

Our time-dependent contraction routine is very similar to our time-independent one from Section 3.4. First we reduce the number of nodes by removing unimportant ones and—in order to preserve distances between non-removed nodes—add time-dependent shortcuts to the graph. Then, we apply an edge-reduction step that removes unneeded shortcuts.

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

The order in which nodes are bypassed changes the resulting contracted graph. Hence, we use a heap to determine the next bypassable node. Therefore, we first determine the number $\#\mathit{shortcut}$ of *new* edges that would be inserted into the graph if u was bypassed, i.e., existing edges connecting nodes in $\mathit{tails}(I)$ with nodes in $\mathit{heads}(O)$ do not contribute to $\#\mathit{shortcut}$. Let $\zeta(u) = \#\mathit{shortcut}/(\deg_{in}(u) + \deg_{out}(u))$ be the *expansion* [GKW07] of node u. Furthermore, let $h(u)$ be the hop number of the hop-maximal shortcut, and let $p(u)$ be the number of interpolation points of the shortcut with most interpolation points, that would be added if u was bypassed. Then we set the key of a node u within the heap to $h(u) + p(u) + 10 \cdot \zeta(u)$, smaller keys having higher priority. By this ordering for bypassing nodes we prefer nodes whose removal yield few additional shortcuts with a small hop number and few interpolation points.

We *stop* the node-reduction as soon as we would bypass a node u with an expansion $\zeta(u) > C$, with C called the expansion threshold. Moreover, to keep the costs of shortcuts limited we do not bypass a node if its removal would either result in a shortcut with more

than I interpolation points or a hop number greater than H. We say that the nodes that have been bypassed belong to the *component*, while the remaining nodes are called *core nodes*.

Corollary 5.4. *Time-dependent node-reduction keeps distances (for all departure times) between core nodes correct.*

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

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

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

Corollary 5.5. *Time-dependent edge-reduction keeps distances (for all departure times) between core nodes correct.*

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

Discussion. Time-dependent contraction in road networks is space-consuming. Each added shortcut increases the total number of interpolation points of the graph (cf. Fig. 2.1). So, unlike in time-independent scenarios, shortcuts must be carefully chosen and we have to keep the number of shortcuts as low as possible.

5.2 Bidirectional ALT

As already mentioned, unidirectional ALT can be used out of the box if preprocessing is done on the lower bound graph $\underline{G}$. However, unidirectional ALT yields only mild speed-ups even in time-independent Scenarios (cf. Tab. 4.1). Since potentials are based on $\underline{G}$, it is expected that query performance is even worse in time-dependent networks.

In this section, we show how *bidirectional* ALT can be used in time-dependent networks anyway. The idea is as follows: A backward search is performed in $\underline{G}$ and is only used to restrict nodes that need to be visited by the forward search. In the following, we explain the query algorithm in more detail, provide a proof of correctness, present several optimizations, and show how this approach can be used to approximate time-dependent shortest paths.

5.2.1 Query

The query algorithm is based on restricting the scope of a time-dependent A^* search from the source using a set of nodes defined by a time-*independent* A^* search from the

destination, i.e., the backward search is a reverse search in $\underline{G}$, which corresponds to the graph G weighted by the lower bounding function $\underline{len}$. More precisely, it works in in three phases:

1. A bidirectional ALT occurs on G, where the forward search is performed on the graph, and the backward search is run on the graph weighted by the lower bounding function $\underline{len}$. All nodes settled by the backward search are included in a set M. Phase 1 terminates as soon as the two search scopes meet.

2. Suppose that $v \in V$ is the first node in the intersection of the heaps of the forward and backward search; then the time dependent cost $\mu = \gamma_\tau(p_v)$ of the path p_v going from s to t passing through v is an upper bound to $d(s, t, \tau)$. In the second phase,

Algorithm 4: TDALT($G = (V, E)$, s, t, τ)

```
1  Q→.insert(s, 0); Q←.insert(t, 0); M := ∅; μ := +∞; done := false; phase := 1.
2  while ¬ done do
3      if (phase = 1) ∨ (phase = 2) then
4          ↔ ∈ {→, ←};        // phase 1 or 2:  alternate forward and backward
5      else
6          ↔ := →;                                  // phase 3:  only forward
7      u := Q↔.extractMin()
8      if (u = t) ∧ (↔ = →) then
9          done := true
10         continue
11     if (phase = 1) ∧ (u.dist→ + u.dist← < ∞) then
12         μ := u.dist→ + u.dist←;                          // update upper bound
13         phase := 2;                                    // switching to phase 2
14     if (phase = 2) ∧ (↔ = ←) ∧ (μ < u.key←) then
15         phase := 3;                                    // switching to phase 3
16         continue
17     for edges (u, v) ∈ E↔ do
18         if ↔ = ← then
19             M.insert(u)
20         else if (phase = 3) ∧ (v ∉ M) then
21             continue;   // only explore nodes explored by backward search
22         if (v ∈ Q↔) then
23             if u.dist↔ + len((u, v), u.dist↔) < v.dist↔ then
24                 Q↔.decreaseKey(v, u.dist↔ + len((u, v), u.dist↔) + π↔(v))
25             else
26                 Q↔.insert(v, u.dist↔ + len((u, v), u.dist↔) + π↔(v))
27 return μ
```

both search scopes are allowed to proceed until the backward search queue only contains nodes whose associated key exceeds μ. In other words: let β be the key of the minimum element of the backward search queue; phase 2 terminates as soon as $\beta > \mu$. Again, all nodes settled by the backward search are included in M.

3. Only the forward search continues, with the additional constraint that only nodes in M can be explored. The forward search terminates when t is settled.

The pseudocode for this algorithm is given in Algorithm 4. Note that we use the symbol $\leftrightarrow$ to indicate either the forward search ($\leftrightarrow = \rightarrow$) or the backward search ($\leftrightarrow = \leftarrow$). We denote by $\overrightarrow{E}$ the set of edges for the forward search, i.e., $\overrightarrow{E} = E$, and by $\overleftarrow{E}$ the set of edges for the backward search, i.e., $\overleftarrow{E} = \{(u, v) | (v, u) \in E\}$. A typical choice is to alternate between the forward and the backward search at each iteration of the algorithm during the first two phases. The typical search space of a TDALT query is given in Figure 5.4.

Profile Queries. Algorithm 4 computes the distance between two points for a specific departure time. If we want to compute the complete profile between two points, we can directly apply bidirectional search: The problem of unknown arrival no longer persists as we start the backward search for *all* possible arrival times. As a result, Algorithm 4 is not suitable for computing profiles. Thus, we do *not* analyze profile queries for ALT.

5.2.2 Correctness

Recall that we denote by $d(u, v, \tau)$ the length of the shortest path from u to v with departure time τ, by $\underline{d}(u, v)$ the length of the shortest path from u to v in the graph $\underline{G}$, and by $\pi_f(u)$ and $\pi_b(u)$ forward and backward potential of node u. We have the following theorems.

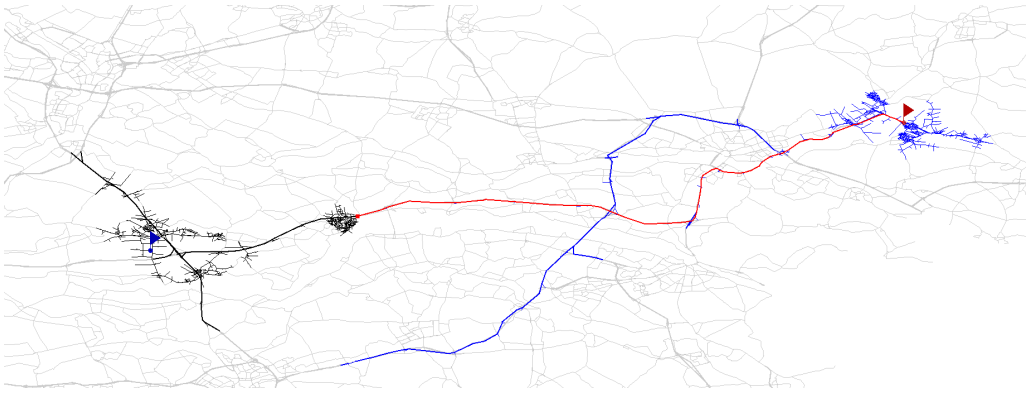
Theorem 5.6. *Algorithm 4 computes the shortest time-dependent path from s to t for a given departure time τ.*

Proof. The forward search of Algorithm 4 is exactly the same as the unidirectional version of the ALT algorithm during the first 2 phases, and thus it is correct; we have to prove that the restriction applied during phase 3 does not interfere with the correctness of the A^* algorithm.

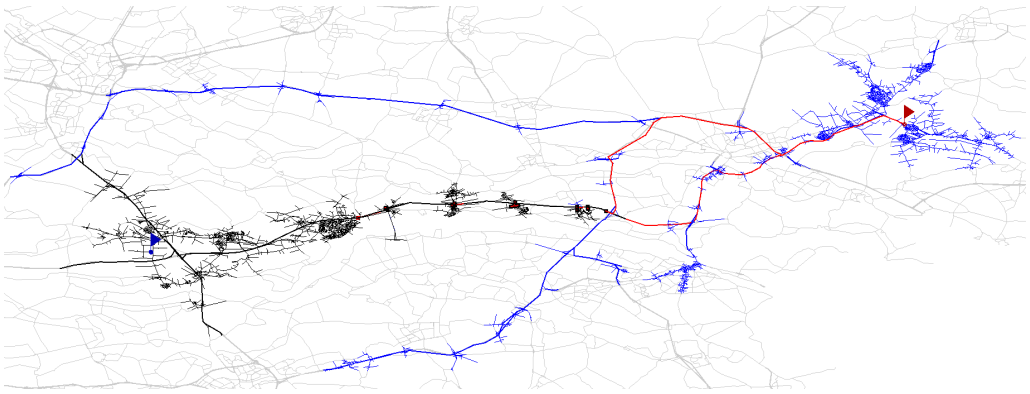
Let μ be an upper bound on the cost of the shortest path; in particular, this can be the cost $\gamma_\tau(p_v)$ of the $s \rightarrow t$ path passing through the first meeting point v of the forward and backward search. Let β be the smallest key of the backward search priority queue at the end of phase 2. Suppose that Algorithm 4 is not correct, i.e., it computes a sub-optimal path. Let p^* be the shortest path from s to t with departure time τ, and let u be the first node on p^* which is not explored by the forward search; by phase 3, this implies that $u \notin M$, i.e., u has not been settled by the backward search during the first 2 phases of Algorithm 4. Hence, we have that $\beta \leq \pi_b(u) + \underline{d}(u, t)$; then we have the chain $\gamma_\tau(p^*) \leq \mu < \beta \leq \pi_b(u) + \underline{d}(u, t) \leq \underline{d}(s, u) + \underline{d}(u, t) \leq d(s, u, \tau) + d(u, t, d(s, u, \tau)) = \gamma_\tau(p^*)$, which is a contradiction. □

5.2.3 Approximation

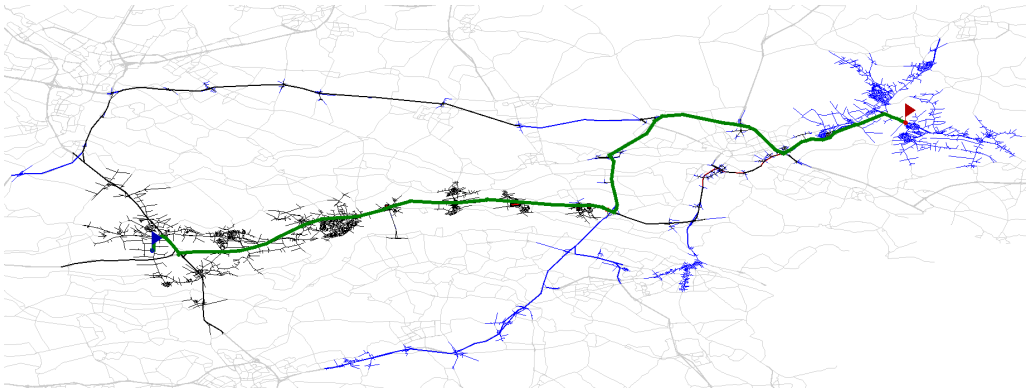
Analyzing Algorithm 4 and Theorem 5.6, we obtain a guaranteed approximation of the path found by switching to phase 3 earlier.



(a) Snapshot of the search space at the end of phase 1.



(b) Snapshot of the search space at the end of phase 2.



(c) Search space after termination.

Figure 5.4: Progress of an example TDALT query. The input is a road network with congested motorways. The source of the query is marked by the blue flag (left), the target by the red one (right). The quickest path is drawn in dark green. Roads touched by TDALT are colored: black depicts an edge relaxed by the forward search, blue depicts relaxation by the backward search, red are edges examined for determining an upper bound. Grey edges have not been touched. a) shows the search space at the end of phase 1, b) at end of phase 2, and c) the resulting search space. We observe that at the computed path differs from the path found at the end of phase 1. Moreover, it is clearly observable that only the forward search continues during phase 3.

Theorem 5.7. *Let p^* be the shortest path from s to t. If the condition to switch to phase 3 is $\mu < K\beta$ for a fixed parameter K, then Algorithm 4 computes a path p from s to t such that $\gamma_\tau(p) \leq K\gamma_\tau(p^*)$ for a given departure time τ.*

Proof. Suppose that $\gamma_\tau(p) > K\gamma_\tau(p^*)$. Let u be the first node on p^* which is not explored by the forward search; by phase 3, this implies that $u \notin M$, i.e., u has not been settled by the backward search during the first 2 phases of Algorithm 4. Hence, we have that $\beta \leq \pi_b(u) + \underline{d}(u,t)$; then we have the chain $\gamma_\tau(p) \leq \mu < K\beta \leq K(\pi_b(u) + \underline{d}(u,t)) \leq K(\underline{d}(s,u) + \underline{d}(u,t)) \leq K(d(s,u,\tau) + d(u,t,d(s,u,\tau))) = K(\gamma_\tau(p^*)) < \gamma_\tau(p)$, which is a contradiction. □

5.2.4 Optimizations

Performance of the basic version of the algorithm can be improved with the results that we describe in this section.

Theorem 5.8. *Let p be the shortest path from s to t with departure time τ. If all nodes u on p settled by the backward search are settled with a key smaller or equal to $d(s,u,\tau) + d(u,t,d(s,u,\tau))$, then Algorithm 4 is correct.*

Proof. Let Q be the backward search queue, let $\text{key}(u)$ be the key for the backward search of node u, let $\beta = \text{key}(v)$ be the smallest key in the backward search queue, which is attained at a node v, and let μ be the best upper bound on the cost of the solution currently known. To prove correctness, using the same arguments as in the proof of Thm. 5.6 we must make sure that, when the backward search stops at the end of phase 2, then all nodes on the shortest path from s to t that have not been explored by the forward search have been added to M. The backward search stops when $\mu < \beta$.

In an A^* search, the keys of settled nodes are non-decreasing. So every node u which at the current iteration has not been settled by the backward search will be settled with a key $\text{key}(u) \geq \text{key}(v)$, which yields $d(s,u,\tau) + d(u,t,d(s,u,\tau) \geq \text{key}(v) = \beta > \mu \; \forall u \in Q$. Thus, every node which has not been settled by the backward search cannot be on the shortest path from s to t, and Algorithm 4 is correct. □

This allows the use of larger lower bounds during the backward search: the backward A^* search does not have to compute shortest paths on the graph $\underline{G}$, but it should in any case guarantee that when a node u is settled then its key is an underestimation of the time-dependent cost of the time-dependent shortest path between s and t passing through u. The next proposition is of fundamental practical importance.

Proposition 5.9. *During phase 2 the backward search does not need to explore nodes that have already been settled by the forward search.*

Proof. Let $d_b(v)$ be the distance from a node v to node t computed by the backward search if we do not explore any node already explored by the forward search. We will prove that, when a node v on the shortest path from s to t with departure time τ is settled by the backward search, then $d_b(v) \leq d(v,t,d(s,v,\tau)) \; \forall \tau \in T$. By Thm. 5.8, this is enough to prove our statement.

Consider a node v settled by the backward search, but not by the forward search; let q be the shortest path from s to v with departure time τ, let q^* be the shortest path from

v to t with departure time $\tau_v = \gamma_\tau(q)$. Suppose that q^* does not pass through any node already settled by the forward search. Then $d_b(v) \leq \underline{\gamma}(q^*) \leq d(v, t, d(s, v, \tau))$.

Suppose now that q^* passes through a node w already settled by the forward search. Let p be the shortest path from s to w with departure time τ, and let p^* be the shortest path from w to t with departure time $\tau_w = \gamma_\tau(p)$; clearly v cannot be on p, because otherwise it would have been settled by the forward search. So we have, by the FIFO property and by optimality of p, that $\gamma_\tau(p) + \gamma_{\tau_w}(p') \leq \gamma_\tau(q) + \gamma_{\tau_v}(q')$, which means that v does not have to be explored and added to the set M by the backward search, because we already have a better path passing through w. Thus, even if $\text{key}(v) > d(s, v, \tau) + d(v, t, d(s, v, \tau))$ Algorithm 4 is correct. □

By Thm. 5.8, we can take advantage of the fact that the backward search is used only to bound the set of nodes explored by the forward search. This means that we can tighten the bounds used by the backward search, even if doing so resulted in an A^* backward search that computes suboptimal distances. To derive some valid lower bounds we need the following lemma and propositions.

Lemma 5.10. *Let v be a node, and u its parent node in the shortest path from s to v with departure time τ. Then $d(s, u, \tau) + \pi_f(u) \leq d(s, v, \tau) + \pi_f(v)$.*

Proof. Suppose that ℓ is the active landmark, i.e., the landmark in our landmarks set that currently gives the best bound; we have that either $\pi_f(u) = \underline{d}(u, \ell) - \underline{d}(t, \ell)$ or $\pi_f(u) = \underline{d}(\ell, t) - \underline{d}(\ell, u)$.

First case: $\pi_f(u) = \underline{d}(u, \ell) - \underline{d}(t, \ell)$. We have $d(s, u, \tau) + \pi_f(u) = d(s, u, \tau) + \underline{d}(u, \ell) - \underline{d}(t, \ell) \leq d(s, u, \tau) + \underline{d}(u, v) + \underline{d}(v, \ell) - \underline{d}(t, \ell) \leq d(s, u, \tau) + \underline{len}(u, v) + \underline{d}(v, \ell) - \underline{d}(t, \ell) \leq d(s, v, \tau) + \pi_f(v)$.

Second case: $\pi_f(u) = \underline{d}(\ell, t) - \underline{d}(\ell, u)$. We have $d(s, u, \tau) + \pi_f(u) = d(s, u, \tau) + \underline{d}(\ell, t) - \underline{d}(\ell, u)$; by triangular distance, $\underline{d}(\ell, v) \leq \underline{d}(\ell, u) + \underline{d}(u, v) \leq \underline{d}(\ell, u) + \underline{len}(u, v)$, which yields $-\underline{d}(\ell, u) \leq -\underline{d}(\ell, v) + \underline{len}(u, v)$. So $d(s, u, \tau) + \underline{d}(\ell, t) - \underline{d}(\ell, u) \leq d(s, u, \tau) + \underline{d}(\ell, t) - \underline{d}(\ell, v) + \underline{len}(u, v) \leq d(s, v, \tau) + \pi_f(v)$. □

Proposition 5.11. *At a given iteration, let v be the last node settled by the forward search. Then, for each node w which has not been settled by the forward search, $d(s, v, \tau) + \pi_f(v) - \pi_f(w) \leq d(s, w, \tau)$.*

Proof. There are two possibilities for w: either it has been explored (but not settled) by the forward search, or it has not been explored. Let Q be the set of nodes in the forward search queue. If w has been explored, then $w \in Q$, and clearly $d(s, v, \tau) + \pi_f(v) \leq d(s, w, \tau) + \pi_f(w)$ because v has been extracted before w, which proves our statement. Otherwise, there is a node $u \in Q$ on the shortest path from s to w with departure time τ which has been explored but not settled. We have that $d(s, v, \tau) + \pi_f(v) \leq d(s, u, \tau) + \pi_f(u)$ because v has been extracted while u is still in the queue, and by Lemma 5.10, if we examine the nodes $u = u_1, u_2, \ldots, u_k = w$ on the shortest path from s to w with departure time τ, we have that $d(s, u_1, \tau) + \pi_f(u_1) \leq \cdots \leq d(s, u_k, \tau) + \pi_f(u_k)$, from which our statement follows. □

Let v be as in Prop. 5.11, and w a node which has not been settled by the forward search. Prop. 5.11 suggests that we can use

$$\pi_b^*(w) = \max\{\pi_b(w), d(s, v, \tau) + \pi_f(v) - \pi_f(w)\} \tag{5.1}$$

as a lower bound to $d(s, w, \tau)$ during the backward search. However, we have to make sure that the bound is valid at each iteration of Algorithm 4.

Lemma 5.12. *If the key of the forward search used to compute the potential function π_b^* defined by (5.1) is fixed, then we have $\pi_b^*(v) \leq \pi_b^*(u) + \underline{len}(u, v)$ for each edge $(u, v) \in E$.*

Proof. By definition we have $\pi_b^*(v) = \max\{\pi_b(v), \alpha - \pi_f(v)\}$, where with α we denoted the key of a node settled by the forward search, which is fixed by hypothesis. Consider the case $\pi_b^*(v) = \pi_b(v)$; then, since the landmark potential functions π_b and π_f are consistent, we have $\pi_b^*(v) = \pi_b(v) \leq \pi_b(u) + \underline{len}(u, v) \leq \pi_b^*(u) + \underline{len}(u, v)$. Now consider the case $\pi_b^* = \alpha - \pi_f(v)$; then we have $\pi_b^*(v) = \alpha - \pi_f(v) \leq \alpha - \pi_f(u) + \underline{len}(u, v) \leq \pi_b^*(u) + \underline{len}(u, v)$, which completes the proof. □

This is enough to prove correctness of our algorithm with tightened bounds, as stated in the next theorem.

Theorem 5.13. *If we use the potential function π_b^* defined by (5.1) as potential function for the backward search, with a fixed value of the forward search key, then Algorithm 4 is correct.*

Proof. Let $d_b(u)$ be the distance from a node u to node t computed by the backward search. We will prove that, when a node u on the shortest path from s to t is settled by the backward search, $d_b(u) \leq d(u, t, d(s, u, \tau))\ \forall \tau \in T$. By Prop. 5.11 and Thm. 5.8, this is enough to prove our statement.

Let $q^* = (v_1 = u, \ldots, v_n = t)$ be the shortest path from u to t on $\underline{G}$. We proceed by induction on $i : n, \ldots, 1$ to prove that each node v_i is settled with the correct distance on $\underline{G}$, i.e., $d_b(v_i) = \underline{d}(v_i, t)$. It is trivial to see that the nodes v_n and v_{n-1} are settled with the correct distance on $\underline{G}$. For the induction step, suppose v_i is settled with the correct distance $d_b(v_i) = \underline{d}(v_i, t)$. By Lemma 5.12, we have $d_b(v_i) + \pi_b^*(v_i) \leq d_b(v_i) + \underline{len}(v_{i-1}, v_i) + \pi_b^*(v_{i-1}) = \underline{d}(v_{i-1}, t) + \pi_b^*(v_{i-1}) \leq d_b(v_{i-1}) + \pi_b^*(v_{i-1})$, hence v_i is extracted from the queue before v_{i-1}. This means that v_{i-1} will be settled with the correct distance $d_b(v_{i-1}) = \underline{d}(v_{i-1}, t)$, and the induction step is proven.

Thus, u will be settled with distance $d_b(u) = \underline{d}(u, t) \leq d(u, t, d(s, u, \tau))$, which proves our statement. □

By Thm. 5.13, Algorithm 4 is correct when using π_b^* only if we assume that the node v used in (5.1) is fixed at each backward search iteration. Thus, we do the following: we set up 10 checkpoints during the query; when a checkpoint is reached, the node v used to compute (5.1) is updated, and the backward search queue is flushed and filled again using the updated π_b^*. This is enough to guarantee correctness. The checkpoints are computed comparing the initial lower bound $\Delta = \pi_f(t)$ and the current distance from the source node, both for the forward search: the initial lower bound is divided by 10 and, whenever the current distance from the source node exceeds $k\Delta/10$ with $k \in \{1, \ldots, 10\}$, π_b^* is updated.

5.2.5 Dynamic Scenario

In Section 5.5, we will see that TDALT cannot compete with time-dependent SHARC. However, a main advantage of TDALT is that it is applicable in dynamic scenarios without any additional effort. The reason for this is the same as for dynamic ALT (cf. Section 4.1).

5.3 Core-ALT

Like for time-independent route planning, we can extend our bidirectional time-dependent ALT from the last section by contraction. It turns out that preprocessing is very similar to a time-independent scenario, while the query needs further adaption due to the problem of bidirectional search.

5.3.1 Preprocessing

As mentioned above, preprocessing is similar to the time-independent scenario: First, we extract a core $G_C = (V_C, E_C)$ with our time-dependent contraction routine from Section 5.1. Then, we merge the core with the original graph to obtain $G_F = G_C \cup G = (V, E \cup E_C)$ since $V_C \subset V$. Finally, we select landmarks from G_C and compute landmark distances in $\underline{G_C}$.

5.3.2 Query

The query algorithm consists of two phases, performed on G_F:

1. Initialization phase: start a Dijkstra search from both the source and the destination node on G_F, using the time-dependent costs for the forward search and the time-independent costs $\underline{len}$ for the backward search, pruning the search (i.e., not relaxing outgoing edges) at nodes $\in V_C$. Add each node settled by the forward search to a set S, and each node settled by the backward search to a set T. Iterate between the two searches until: (i) $S \cap T \neq \emptyset$ or (ii) the priority queues are empty.

2. Main phase: (i) If $S \cap T \neq \emptyset$, then start an unidirectional Dijkstra search from the source on G_F until the target is settled. (ii) If the priority queues are empty and we still have $S \cap T = \emptyset$, then start TDALT on the graph G_C, initializing the forward search queue with all leaves of S and the backward search queue with all leaves of T, using the distance labels computed during the initialization phase. The forward search is also allowed to explore any node $v \in T$, throughout the 3 phases of the algorithm. Stop when t is settled by the forward search.

In other words, the forward search "hops on" the core when it reaches a node $u \in S \cap V_C$, and "hops off" at all nodes $v \in T \cap V_C$. Again, since Dijkstra's algorithm is equivalent to A^* with a zero potential function, we can use Algorithm 4 in case (ii) during the main phase.

Proxy Nodes. Again, we need to compute lower bounds from any node to source and target of the query. We use the same methods as introduced for time-independent routing (cf. Section 4.2).

Unpacking Shortcuts. The path returned by our algorithm contains shortcuts added during contraction. If we want to retrieve the original corresponding path in G, we have to unpack shortcuts. Since we allow multi-edges during contraction (cf. Section 5.1), we can directly apply our contraction routine introduced for time-independent SHARC (cf. Section 4.3).

Profile Queries. Like for time-dependent ALT, the CALT query algorithm computes the distance between two points for a specific departure time. However, we could directly apply bidirectional search for profile queries. For the same reasons as ALT, we do *not* analyze profile queries for CALT.

5.3.3 Correctness

Theorem 5.14. *The core routing algorithm for time-dependent graphs is correct.*

Proof. Suppose that, during the initialization phase (i.e., when we build the two sets S and T), the two search scopes meet, thus $S \cap T \neq \emptyset$. In this case, we switch to unidirectional Dijkstra's algorithm on the original graph (plus added shortcuts), and correctness follows. Now suppose that the two search scopes do not meet: the two priority queues are empty and $S \cap T = \emptyset$, thus the shortest path p between s and t with departure time τ passes through at least one node belonging to the core V_C. Let $p = (s, \ldots, u, \ldots, v, \ldots, t)$, where u and v are, respectively, the first and the last node $\in V_C$ on the path. If $u = v$ then the proof is trivial; suppose $u \neq v$. Since the initialization phase explores all non-core nodes reachable from s and t, $u \in S$ and $v \in T$. By definition of v, $p_{|v \to t}$ passes only through non-core nodes; by the query algorithm, T contains all non-core nodes that can reach t passing only through non-core nodes. It follows that all nodes of $p_{|v \to t}$ are in T. Thus $p_{|u \to t}$ is entirely contained in $G_C \cup G[T] = (V_C \cup T, E_C \cup E[T])$. By correctness of Dijkstra's algorithm, the distance labels for nodes in S are exact with respect to the time-dependent cost function. Initializing the forward search queue with the leaves of S and applying A^* on $G_C \cup G[T]$ then yields the shortest path p by correctness of A^*. □

We immediately observe that for case (ii) of the main phase we can use any algorithm that guarantees correctness when applied on $G_C \cup G(T)$. In particular, the distance labels for nodes in T are correct distance labels for the backward search on the graph weighted by $\underline{len}$, so they fulfill the requirements for TDALT. Note that, in a typical core-routing setting for the ALT algorithm, landmark distances are computed and stored only for vertices in V_C (cf Section4.2), since the initialization phase on non-core nodes uses Dijkstra's algorithm only. This means that the landmark potential function cannot be used to apply the forward A^* search on the nodes in T. However, in order to combine TDALT with a core-routing framework we can use the backward distance labels computed with Dijkstra's algorithm during the initialization phase. Those are correct distance labels for the lower bounding function $\underline{len}$, thus they yield valid potentials for the forward search. We call this algorithm Time-Dependent Core-based ALT (TDCALT).

5.3.4 Updating the Core

Modifications in the cost functions can be easily taken into account under weak assumptions if shortcuts have not been added to the graph. However, a two-levels hierarchical setup is significantly more difficult to deal with, exactly because of shortcuts: since a shortcut represents the shortest path between its two endpoints for at least one departure time, if some edge costs change then the shortest path which is represented may also be subject to changes. Thus, a procedure to restore optimality of the core is needed. We first analyze the simple case of increasing breakpoint values, and then propose an algorithmic

framework to deal with general cost changes under some restrictive assumptions which are acceptable in practice.

Increases in Breakpoint Values. Let (V_C, E_C) be the core of G. Suppose that the cost function of one edge $e \in E$ is modified; the set of core nodes V_C need not change, as long as E_C is updated in order to preserve distances with respect to the uncontracted graph $G = (V, E)$ with the new cost function. There are two possible cases: either the new values of the modified breakpoints are smaller then the previous ones, or they are larger. In the first case, then all edges on the core E_C must be recomputed by running a label-correcting algorithm between the endpoints of each shortcut, as we do not know which shortcuts the updated edge may contribute to. In the second case, then the cost function for core edges may change for all those edges $e' \in E_C$ such that e' contains e in its decomposition for at least one time instant τ. In other words, if e contributed to a shortcut e', then the cost of e' has to be recomputed. As the cost of e has increased, then e cannot possibly contribute to other edges, thus we can restrict the update only to the shortcuts that contain the edge. To do so, we store for each $e \in E$ the set $S(e)$ of all shortcuts that e contributes to. Then, if one or more breakpoints of e have their value changed, we do the following.

Let $[\tau_1, \tau_{n-1}]$ be the smallest time interval that contains all modified breakpoints of edge e. If the breakpoints preceding and following $[\tau_1, \tau_{n-1}]$ are, respectively, at times τ and τ_n the cost function of e changes only in the interval $[\tau, \tau_n]$. For each shortcut $e' \in S(e)$, let $e'_0, \ldots, e'_d$, with $e'_i \in E \, \forall i$, be its decomposition in terms of the original edges, let $\lambda_j = \sum_{i=0}^{j-1} \underline{len}(e'_i)$ and $\rho_j = \sum_{i=0}^{j-1} \overline{len}(e'_i)$. If e is the edge with index j in the decomposition of e', then e' may be affected by the change in the cost function of e only if the departure time from the starting point of e' is in the interval $[\tau - \rho_j, \tau_n - \lambda_j]$. This is because e can be reached from the starting node of e' no sooner than λ_j, and no later than ρ_j. Thus, in order to update the shortcut e', we need to run a label-correcting algorithm between its two endpoints only in the time interval $[\tau - \rho_j, \tau_n - \lambda_j]$, as the rest of the cost function is not affected by the change. In practice, if the length of the time interval $[\tau, \tau_n]$ is larger than a given threshold we run a label-correcting algorithm between the shortcut's endpoints over the whole time period, as the gain obtained by running the algorithm over a smaller time interval does not offset the overhead due to updating only a part of the profile with respect to computing from scratch.

A Realistic Scenario. The procedure described above is valid only when the value of breakpoints increases. In a typical realistic scenario, this is often the case: the initial cost profiles are used to model normal traffic conditions, and cost updates occur only to add temporary slowdowns due to unexpected traffic jams. When the temporary slowdowns are no longer valid we would like to restore the initial cost profiles, i.e., lower breakpoints to their initial values, without recomputing the whole core. If we want to allow fast updates as long as the new breakpoint values are larger than the ones used for the initial core construction, without requiring that the values can only increase, then we have to manage the sets $S(e) \, \forall e \in E$ accordingly.

For example, given $e \in E$, suppose that the cost of its breakpoint at time $\tau \in \mathcal{T}$ increases, and all shortcuts $\in S(e)$ are updated. Suppose that, for a shortcut $e' \in S(e)$, e does not contribute to e' anymore due to the increased breakpoint value. If e' is removed from $S(e)$ and at a later time the value of the breakpoint at τ is restored to the original

value, then e' would not be updated because $e' \notin S(e)$, thus e' would not be optimal.

Our approach to tackle this problem is the following: for each edge $e \in E$, we update the sets $S(e)$ whenever a breakpoint value changes, with the additional constraint that elements of $S(e)$ after the initial core construction phase cannot be removed from the set. Thus, $S(e)$ contains all shortcuts that e contributes to with the current cost function, plus all shortcuts that e contributed to during the initial core construction. As a consequence we may update a shortcut $e' \in S(e)$ unnecessarily, if e contributed to e' during the initial core construction but ceased contributing after an update step; however, this guarantees correctness for all changes in the breakpoint values, as long as the new values are not strictly smaller than the values used during the initial graph contraction. From a practical point of view, this is a reasonable assumption.

Since the sets $S(e) \forall e \in E$ are stored in memory, the computational time required by the core update is largely dominated by the time required to run the label-correcting algorithm between the endpoints of shortcuts. Thus, we have a trade-off between query speed and update speed: if we allow the contraction routine to build long shortcuts (in terms of number of bypassed nodes, i.e., "hops", as well as traveling time) then we obtain a faster query algorithm, because we are able to skip more nodes during the shortest path computations. On the other hand, if we allow only limited-length shortcuts, then the query search space is larger, but the core update is significantly faster as the label-correcting algorithm takes less time. In Section 5.5.2 we provide an experimental evaluation for different scenarios.

5.4 SHARC

In this section, we show how SHARC, introduced in Section 4.3, can be generalized in such a way that we are able to perform exact shortest-path queries in time-dependent networks. The key observation is that the concept of SHARC stays untouched. However, at certain points we use the augmented routines from Section 5.1 instead of their static counterparts. As a result, we are able to perform *exact* time-dependent queries in road and railway networks, see Figures 5.5 for an example.

5.4.1 Preprocessing

Initialization. In a first step, we remove 1-shell nodes from the graph since we can directly assign correct arc-flags to all edges adjacent to 1-shell nodes: Edges targeting the 2-core get full flags assigned, those directing away from the 2-core get only the own-cell flag set to `true`. Note that this procedure is independent from edge weights. After extracting the 2-core, we perform a multi-level partitioning of the *unweighted* graph. The partition has to fulfill several requirements: cells should be connected, the size of cells should be balanced, and the number of boundary nodes should be as low as possible. Like for time-independent SHARC, we obtain such a partition by local optimization of a partition obtained from SCOTCH [Pel07].

Iteration. After the initialization, an iterative process starts. Each iteration step is divided into two parts, described in the following: contraction and arc-flag computation.

Contraction. First, we apply a contraction step according to Section 5.1. However, in order to guarantee correctness, we have to use *cell-aware* contraction, i.e., a node n is never marked as bypassable if any of its neighboring nodes is *not* in the same cell as n.

Arc-Flags. We have to set arc-flags for all edges of our output-graph, including those which we remove during contraction. Like for time-independent SHARC, we can set arc-flags for all removed edges automatically. We set the arc-flags of the current and all higher levels depending on the tail u of the deleted edge. If u is a core node, we only set the own-cell flag to true (and others to false) because this edge can only be relevant for a query targeting a node in this cell. If u belongs to the component, all arc-flags are set to true as a query has to leave the component in order to reach a node outside this cell.

Setting arc-flags of those edges not removed from the graph is more expensive since we apply one of the preprocessing techniques for multi-level Arc-Flags from 5.1. Note that we lose correctness of SHARC if we use heuristic arc-flags as preprocessing strategy.

Finalization. The last phase of our preprocessing-routine assembles the output graph. It contains the original graph, shortcuts added during preprocessing and arc-flags for all edges of the output graph. However, some edge may have no arc-flag set to true. As these edges are never relaxed by our query algorithm, we directly remove such edges from the output graph.

5.4.2 Query

Time-dependent SHARC allows time- and profile-queries. For computing $d(s, t, \tau)$, we use a modified Dijkstra that operates on the output graph. The modifications are as follows: When settling a node n, we compute the lowest level i on which n and the target node t are in the same supercell. Moreover, we consider only those edges outgoing from n having a set arc-flag on level i for the corresponding cell of t. In other words, we *prune* edges that are not important for the current query. We stop the query as soon as we settle t. See Fig. 5.5 for an example query.

For computing $d_*(s, t)$, we use a modified variant of our label-correcting algorithm (see Section 5.1) that also operates on the output graph. The modifications are the same as for time-queries and the stopping criterion is the standard one explained in Section 5.1.

Outputting Shortest Paths. SHARC adds shortcuts to the graph in order to accelerate queries. If the complete description of the path is needed, the shortcuts have to be unpacked. As we allow multi-edges during contraction, each shortcut represents exactly one path in the network, and hence, we can directly apply the unpacking routine from our time-independent variant of SHARC.

5.4.3 Correctness

Theorem 5.15. *Time-dependent SHARC is correct.*

Proof. The correctness of time-dependent SHARC can be shown equivalently to time-independent SHARC. The proof of Lemma 4.3 is based on two facts: Contraction preserves distances and multi-level Arc-Flags is correct. According to Corollaries 5.4 and 5.5, and Lemma 5.3, our augmented ingredients fulfill these requirements. Hence, our time-dependent variant of SHARC is correct as well. □

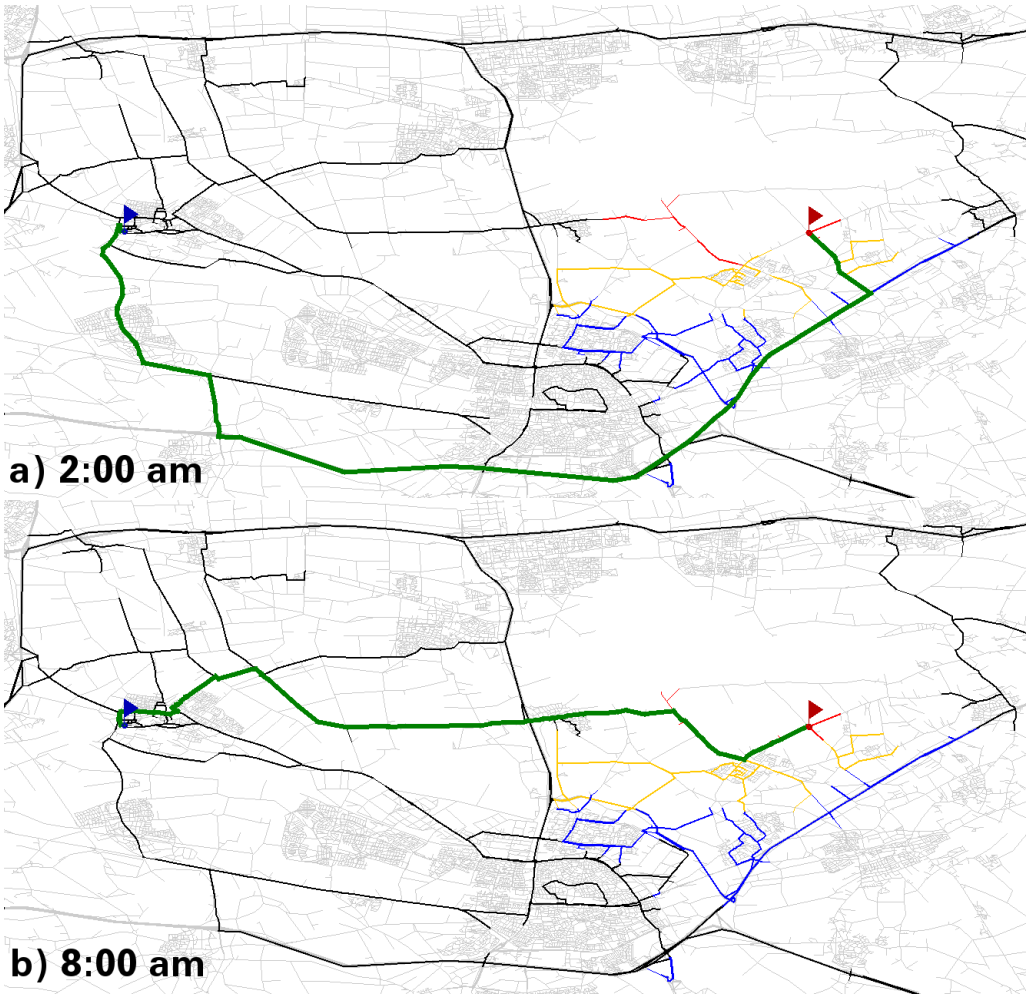


Figure 5.5: Two examples for time-dependent queries with different departure times (but identical source and target) yielding different quickest paths. The input is a road network with congested motorways. The source of the query is marked by the blue flag (left), the target by the red one (right). The quickest paths are drawn in dark green. Roads touched by SHARC are colored: black depicts an edge with a `true` arc-flag evaluated on the topmost level, blue depicts the second, yellow the third, and red the fourth level. Grey edges have not been touched. Note that the edges touched are almost independent of the departure time. Also note that for a nighttime departure (a), it pays off to use the highway (lower route) while for a departure during rush hours the quickest path is the direct way without using highways.

5.4.4 Optimizations

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

Refinement of arc-flags. Recall that refinement of arc-flags tries to improve those flags set to `true` during the contraction process. This is achieved by propagating flags of edges outgoing from high-level nodes to those outgoing from low-level nodes. In a time-independent scenario, we grow shortest path trees to find these so called exit nodes of each node. In a time-dependent scenario, we identify the exit nodes by constructing profile graphs. The propergation itself stays almost untouched: the only difference is that a node might have more than predecessor, which all have to be examined when identifying the corresponding outgoing edge. See Section 4.3.4 for details.

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

Lemma 5.16. *Refinement of Arc-Flags is correct.*

Proof. The only difference between time-dependent and time-independent refinement is that we grow PGs instead of shortest path trees in order to find exit nodes of a node. Hence, we can directly adapt the proof of Lemma 4.5 in order to prove Correctness of Lemma 5.16. □

Removing Shortcuts. As discussed in Section 2.4, time-dependent shortcuts are very space consuming. Hence, we try to remove shortcuts as the very last step of preprocessing. The routine works as follows. For each added shortcut (u, v) we analyze the path $p_{uv} = (u, u_0, \ldots, u_k, v)$ it represents. If all $\deg_{out}(u_i) \leq 3$ for $0 \leq i \leq k$, we remove (u, v) from the graph and the edge (u, u_0) additionally inherit the arc-flags from (u, v).

Improved Locality. Like for time-independent SHARC, we increase cache efficiency of the output graph by reordering nodes according to the level they have been removed at from the graph. As a consequence, the number of cache misses is reduced yielding lower query times.

Landmarks. Approximate arc-flags yield worse results than exact ones. In order to partly remedy this loss in performance, we can add landmarks to SHARC. We can combine ALT with SHARC easily. We run a time-dependent ALT preprocessing consisting of selecting landmarks $L \subseteq V$ and computing $d(l, v), d(v, l)$ for all $v \in V, l \in L$. Then, we apply a normal SHARC-query but use $d(s, u, \tau) + \pi(u)$ (cf. Section 3.2) instead of $d(s, u, \tau)$ as priority key. We call this combination L-SHARC (**L**andmarks and **SHARC**).

5.4.5 Comparison to Static SHARC

Comparing our new time-dependent preprocessing routine with the one for static SHARC due to Chapter 4.3, one may notice that the concept itself stays untouched. We still apply three phase: initialization, iteration, and finalization. The initialization stays untouched

as both 1-shell nodes removal and partitioning are performed on the unweighted graph. However, the iteration process gets more expensive both in terms of memory consumption and preprocessing times: The higher memory consumption is due to the fact that time-dependent shortcuts use more space than static ones, while the longer preprocessing times are due to the more complex algorithms for setting arc-flags.

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

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

5.5 Experiments

In this section, we present our experimental evaluation. To this end, we evaluate the performance of time-dependent ALT, CALT, and SHARC for road and railway networks. Our implementation is written in C++ using solely the STL. As priority queue we use a binary heap. Our tests were executed on one core of an AMD Opteron 2218 running SUSE Linux 10.3. The machine is clocked at 2.6 GHz, has 16 GB of RAM and 2 x 1 MB of L2 cache. The program was compiled with GCC 4.2.1, using optimization level 3.

Inputs. We apply two types of inputs. Road and railway networks. For the former, we have access to a real-world time-dependent road network of Germany. It has approximately 4.7 million nodes and 10.8 million edges. In order to analyze the scalability of our approaches, we additionally use the available real-world *time-independent* network of Western Europe (18 million nodes and 42.6 million edges) and generate *synthetic* rush hours. All data has been provided by PTV AG for scientific use.

Our German data contains five different traffic scenarios, collected from historical data: Monday, midweek (Tuesday till Thursday), Friday, Saturday, and Sunday. As expected, congesture of roads is higher during the week than on the weekend: $\approx 8\%$ of edges are time-dependent for Monday, midweek, and Friday. The corresponding figures for Saturday and Sunday are $\approx 5\%$ and $\approx 3\%$, respectively. We define the delay of a time-query by $1 - d(s,t,\tau)/d(s,t)$ with $d(s,t)$ depicting the length of the shortest s–t path in $\underline{G}$. For our inputs, the average delay over 100 000 random queries is 2.4% for Monday, 2.7% for midweek, 2.6% for Friday, 0.7% for Saturday, and 0.4% for Sunday. This confirms our assumption that traffic is higher during the week than on the weekend.

Our railways timetable data—provided by Hacon for scientific use—of Europe consists

of 30 516 stations and 1 775 482 elementary connections. The period is 24 hours. The resulting realistic, i.e., including transfer times, time-dependent network has about 0.5 million nodes and 1.4 million edges, and is fulfilling the FIFO-property.

Modeling Traffic. We do the following in order to model traffic for our European input. Each edge of this network belongs to one of five main categories representing motorways, national roads, local streets, urban streets, and rural roads. Synthetic time-dependent edge costs are generated assigning, at each node, several random values that represent peak hour (i.e., hour with maximum traffic increase), duration and speed of traffic increase/decrease for a traffic jam; for each node, two traffic jams are generated, one in the morning and one in the afternoon. Then, for each arc in a node's edge star, a *speed profile* is generated, using the traffic jam's characteristics of the corresponding node, and assigning a random increase factor between 1.5 and 3 to represent that edges's slowdown during peak hours with respect to uncongested hours. We do not assign speed profiles to edges that have both endpoints at nodes with level 0 in a pre-constructed Highway Hierarchy, and as a result those edges will have the same traveling time value throughout the day; for all other edges, we use the traffic jam values associated with the endpoint with smallest ID. This method was developed to ensure spatial coherency between traffic increases, i.e. if a certain arc is congested at a given time, then it is likely that adjacent arcs will be congested too. This is a basic principle of traffic analysis [Ker04].

We additionally adjust the degree of perturbation by assigning time-dependent edge-costs only to specific categories of edges. In our *no traffic* scenario, all edges are time-independent, i.e., the graph is static. In a *low traffic* scenario, all motorways are time-dependent, other roads are time-independent. The *medium traffic* scenario additionally includes congested national roads, and for the *high traffic* scenario, we perturbe all edges except local and rural roads.

Setup. In the following, we report preprocessing times and the overhead of the preprocessed data in terms of *additional* bytes per node. Moreover, we report two types of queries: *time-queries*, i.e., queries for a specific departure time, and *profile-queries*, i.e., queries for computing $d_*(s,t)$. For each type we provide the average number of settled nodes, i.e., the number of nodes taken from the priority queue, and the average query time. For s-t profile-queries, the nodes s and t are picked uniformly at random. Time-queries additionally need a departure time τ as well, which we pick uniformly at random as well. As all methods introduced in this chapter have approximate variants, we record four different statistics to characterize the solution quality: error rate, average relative error, maximum relative error, maximum absolute error. By *error rate* we denote the percentage of computed suboptimal paths over the total number of queries. By *relative error* on a particular query we denote the relative percentage increase of the approximated solution over the optimum, computed as $\omega/\omega^* - 1$, where ω is the cost of the approximated solution and ω^* is the cost of the optimum computed by Dijkstra's algorithm. We report *average* and *maximum* values of this quantity over the set of all queries. The maximum absolute error is given by $\omega - \omega^*$. All figures in this chapter are based on 100 000 random s-t queries and refer to the scenario that only the lengths of the shortest paths have to be determined, without outputting a complete description of the paths. Like in Section 4.5, we also evaluate local queries using the Dijkstra rank methodology [SS05].

5.5.1 ALT

Default Settings. Unless otherwise stated, we use 16 landmarks generated by maxCover yielding 128 Bytes of preprocessed space. For our European road network, preprocessing takes 75 minutes. The corresponding figure for Germany is 23 minutes.

Random Queries. Table 5.1 reports the results of our bidirectional ALT variant on time-dependent networks for different approximation values K using the European road network as input. Note that we also report the number of nodes settled at the *end* of each phase of our algorithm, denoting them with the labels phase 1, phase 2 and phase 3.

As expected, we observe a clear trade-off between the quality of the computed solution and query performance. If we are willing to accept an approximation factor of $K = 2.0$, on the European road network queries are on average 55 times faster than Dijkstra's algorithm, but almost 50% of the computed paths will be suboptimal and, although the average relative error is still small, in the worst case the approximated solution has a cost which is 30% larger than the optimal value. The reason for this poor solution quality is that, for such high approximation values, phase 2 is very short. As a consequence, nodes in the middle of the shortest path are not explored by our approach, and the meeting point of the two search scopes is far from being the optimal one. However, by decreasing the value of the approximation constant K we are able to obtain solutions that are very close to the optimum, and performance is significantly better than for unidirectional ALT or Dijkstra. In our experiments, it seems as if the best trade-off between quality and performance is achieved with an approximation value of $K = 1.15$, which yields average query times smaller than 215 ms with a maximum recorded relative error of 10.57%. As in road networks the speed profiles that weight edges cannot be completely accurate, settling for a slightly suboptimal solution (on average, less than 0.5% over the optimum for $K = 1.15$) usually is not a problem. By decreasing K to values < 1.05 it does not pay

Table 5.1: Performance of the time-dependent versions of Dijkstra, unidirectional ALT, and our bidirectional approach.

		Error			Query			
			relative		# settled nodes			time
method	K	rate	avg	max	phase 1	phase 2	phase 3	[ms]
Dijkstra	-	0.0%	0.000%	0.00%	-	-	8 877 158	5 757.4
uni ALT	-	0.0%	0.000%	0.00%	-	-	2 143 160	1 520.8
ALT	1.00	0.0%	0.000%	0.00%	132 129	2 556 840	3 009 320	2 842.0
	1.05	3.1%	0.012%	3.91%	132 129	1 244 050	1 574 750	1 379.2
	1.07	6.6%	0.034%	6.06%	132 129	849 171	1 098 470	915.4
	1.10	18.1%	0.106%	7.79%	132 129	473 414	622 466	481.9
	1.12	26.1%	0.182%	10.57%	132 129	337 353	444 991	325.0
	1.15	35.4%	0.292%	10.57%	132 129	236 108	311 209	214.2
	1.20	43.0%	0.485%	19.40%	132 129	171 154	225 557	145.3
	1.25	45.4%	0.589%	21.64%	132 129	148 856	196 581	122.3
	1.30	46.4%	0.656%	21.64%	132 129	139 089	184 143	111.6
	1.35	47.0%	0.704%	21.64%	132 129	134 582	178 410	107.4
	1.50	47.1%	0.722%	21.64%	132 129	132 299	175 468	105.4
	1.75	47.2%	0.726%	30.49%	132 129	132 131	175 248	105.4
	2.00	47.2%	0.726%	30.49%	132 129	132 130	175 247	105.4

off to use the bidirectional variant any more, as the unidirectional variant of ALT is faster and is always correct.

An interesting observation is that for $K = 2.0$ switching from a static to a time-dependent scenario increases query times only of a factor of ≈ 2: on the European road network, in a static scenario, ALT-16 has query times of 53.6 ms (see Tab. 4.1), while our time-dependent variant yields query times of 105 ms. We also note that for our bidirectional search there is an additional overhead which increases the time spent per node with respect to unidirectional ALT: on the European road network, using an approximation factor of $K = 1.05$ yields similar query times to unidirectional ALT, but the number of nodes settled by the bidirectional approach is almost 30% smaller. We suppose that this is due to the following facts: in the bidirectional approach, one has to check at each iteration if the current node has been settled in the opposite direction, and during phase 2 the upper bound has to be updated from time to time. The cost of these operations, added to the phase-switch checks, is probably not negligible.

Unmodified potential. We also report, for comparison, the results obtained on the European road network using the unmodified ALT potential function π_b for the backward search, instead of the tightened one π_b^* defined as in Equation (5.1). These can be found in Tab. 5.2, which has the same column labels as Tab. 5.1. Comparing query times with the same value of the approximation constant K, we see that using the potential function π_b^* yields a significant improvement over π_b. The difference in performance is larger as K increases. For $K = 1$ the difference is very small; for $K = 1.05$ the algorithm with π_b is 95% slower than the one with π_b^*, and the slowdown increases to 236% for $K = 1.10$ and to 293% for $K = 1.15$. With the largest approximation factor that we tested in our experiments, $K = 2$, the algorithm without the tightened potential function is more than 5 times slower. The same behavior is observed in terms of the number of settled nodes: while for $K = 1$ the number is very similar (only a 28% increase when not using π_b^*), the

Table 5.2: Performance of the time-dependent versions of Dijkstra, unidirectional ALT and our bidirectional approach *without* the tightened potential function π_b^* defined as in (5.1).

		Error			Query			
			relative		# settled nodes			time
method	K	rate	avg	max	phase 1	phase 2	phase 3	[ms]
Dijkstra	-	0.0%	0.000%	0.00%	-	-	8 877 158	5 757.4
uni ALT	-	0.0%	0.000%	0.00%	-	-	2 143 160	1 520.8
ALT	1.00	0.0%	0.000%	0.00%	719 650	3 763 990	3 862 070	3 291.6
	1.05	3.5%	0.023%	4.88%	719 650	2 996 940	3 238 120	2 683.5
	1.07	5.5%	0.046%	6.94%	719 650	2 519 750	2 874 500	2 290.7
	1.10	12.1%	0.123%	9.45%	719 650	1 810 340	2 201 870	1 619.2
	1.12	20.1%	0.237%	10.93%	719 650	1 416 240	1 772 080	1 218.4
	1.15	32.1%	0.474%	14.35%	719 650	1 049 750	1 345 930	842.0
	1.20	44.4%	0.788%	19.42%	719 650	824 331	1 079 290	618.3
	1.25	50.5%	0.994%	24.57%	719 650	755 262	996 631	553.3
	1.30	53.3%	1.104%	24.57%	719 650	735 524	972 294	531.5
	1.35	54.7%	1.166%	24.57%	719 650	727 843	962 950	526.5
	1.50	56.1%	1.248%	28.16%	719 650	720 359	953 704	524.7
	1.75	56.3%	1.261%	39.34%	719 650	719 661	952 947	519.0
	2.00	56.4%	1.262%	39.41%	719 650	719 650	952 933	518.2

ratio rapidly grows until it reaches a 444% increase for $K = 2$. Thus, a great deal of the significant improvement that we are able to obtain over Dijkstra's algorithm and unidirectional ALT with our bidirectional variant is due to the use of tightened bounds. If we use the standard ALT potential function π_b for the backward search then we do not manage to obtain a speed-up of more than a factor 3 with respect to unidirectional ALT, but this comes at the price of correctness. Summarizing, in our bidirectional approach one of the great advantages is that we are able to derive better lower bounds for the time-dependent search with respect to the original ALT bounds, and the new potential function accounts for a large computational improvement.

Local Queries. For random queries on the European road network, our bidirectional ALT algorithm with $K = 1.15$ is roughly 6.5 times faster than unidirectional ALT on average. In order to gain insight whether this speed-up derives from small- or large-range queries, Fig. 5.6 reports the query times with respect to the Dijkstra rank. These values were gathered on the European road network instance. Note that we use a logarithmic scale due to the fluctuating query times of bidirectional ALT. Comparing both ALT version, we observe that switching from uni- to bidirectional queries pays off especially for long-distance queries. This is not surprising, because for small distances the overhead for bidirectional routing is not counterbalanced by a significant decrease in the number of explored nodes: unidirectional ALT is faster for local queries. For ranks of 2^{24}, the median of the bidirectional variant is almost 2 orders of magnitude lower than for the unidirectional variant. Another interesting observation is the fact that some outliers of bidirectional ALT are almost as slow as the unidirectional variant. Comparing different approximation values, we observe that query times differ by roughly the same factor for all ranks less than 2^{23}.

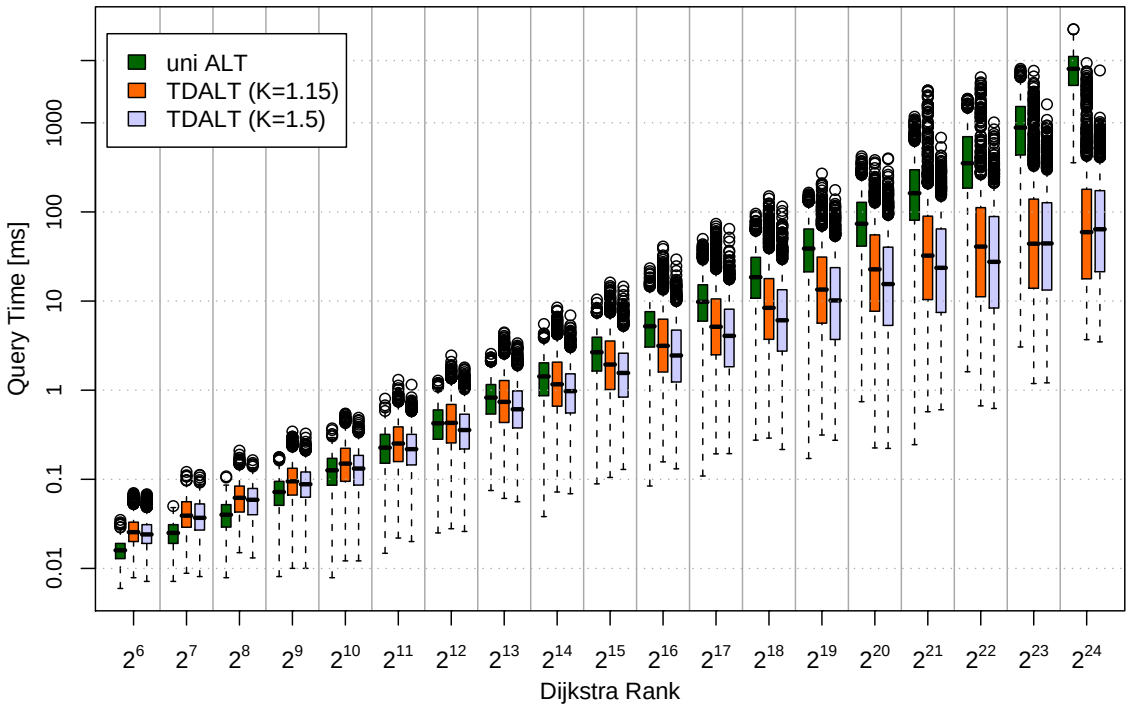


Figure 5.6: Comparison of uni- and bidirectional ALT using the Dijkstra rank methodology.

Number of Landmarks. In static scenarios, query times of bidirectional ALT can be significantly reduced by increasing the number of landmarks to 32 or even 64 (see Tab. 4.1). In order to check whether this also holds for our time-dependent variant, we recorded our algorithm's performance using different numbers of landmarks. Table 5.3 reports those results on the European road network. We evaluate 8 maxcover landmarks (yielding a preprocessing effort of 33 minutes and an overhead of 64 bytes per node), 16 maxcover landmarks (75 minutes, 128 bytes per node) and 32 avoid landmarks (29 minutes, 256 bytes per node). Note that we do not report error rates here, as it turned out that the number of landmarks has almost no impact on the quality of the computed paths. Surprisingly, the number of landmarks has a very small, at least for small K, influence on

Table 5.3: Performance of uni- and bidirectional ALT with different number of landmarks in a time-dependent scenario.

		8 landmarks		16 landmarks		32 landmarks	
	K	# settled	time [ms]	# settled	time [ms]	# settled	time [ms]
uni-ALT	-	2 280 420	1 446.4	2 143 160	1 520.8	2 056 190	1 623.3
ALT	1.00	3 147 440	2 745.5	3 009 320	2 842.0	2 931 080	2 953.3
	1.05	1 714 210	1 373.8	1 574 750	1 379.2	1 516 710	1 409.5
	1.10	768 368	540.2	622 466	481.9	561253	464.2
	1.15	461 259	293.5	311 209	214.2	250 248	184.4
	1.20	375 900	230.6	225 557	145.3	164 419	111.1
	1.50	326 076	195.8	175 468	105.3	113 040	68.1
	2.00	325 801	195.8	175 247	105.4	112 826	68.0

the performance of time-dependent ALT. Even worse, increasing the number of landmarks even yields larger average query times for unidirectional ALT and for bidirectional ALT with low K-values. This is due the fact that the search space decreases only slightly, but the additional overhead for accessing landmarks increases when there are more landmarks to take into account. However, when increasing K, a larger number of landmarks yields faster query times: with $K = 2.0$ and 32 landmarks we are able to perform time-dependent queries 84 times faster than plain Dijkstra, but the solution quality in this case is as poor as in the 16 landmarks case. Summarizing, for $K > 1.10$ increasing the number of landmarks has a positive effect on computational times, although switching from 16 to 32 landmarks does not yield the same benefits as from 8 to 16, and thus in our experiments is not worth the extra memory. On the other hand, for $K \leq 1.10$ and for unidirectional ALT increasing the number of landmarks has a negative effect on computational times, and thus is never a good choice in our experiments.

Traffic Days. Next, we focus on the impact of perturbation on TDALT. Therefore, Tab. 5.4 reports the performance of uni- and bidirectional time-dependent ALT for different traffic days on our German road network. Dijkstra settles 2.2 million nodes in ≈ 1.5 seconds in this setup, independent of the traffic day.

We observe that the degree of perturbation has only a mild impact on unidirectional ALT and exact bidirectional ALT. In a low traffic scenario, unidirectional ALT queries are up 16 times faster than plain Dijkstra, while this values drops to 10 if more edges are perturbed. Switching from exact to approximate queries does not pay off in low traffic scenarios: The gain in performance is only around 20% which seems rather low compared to the loss in quality of paths. However, this value increases to a factor of up to 3 in high

Table 5.4: Performance of TDALT on our German road network instance. *Scenario* depicts the degree of perturbation.

			Error				Query		
				relative		abs.	#settled	#relaxed	time
scenario	algorithm	K	rate	av.	max	max [s]	nodes	edges	[ms]
Monday	uni-ALT	–	0.0%	0.000%	0.00%	0	193 087	230 665	140.38
	TDALT	1.00	0.0%	0.000%	0.00%	0	106 743	127 190	88.53
		1.15	12.5%	0.094%	13.02%	1 811	51 137	60 838	37.23
		1.50	12.5%	0.096%	24.27%	1 811	51 119	60 816	37.12
midweek	uni-ALT	–	0.0%	0.000%	0.00%	0	200 236	239 112	147.20
	TDALT	1.00	0.0%	0.000%	0.00%	0	116 476	138 696	98.27
		1.15	12.4%	0.094%	14.32%	1 892	50 764	60 398	36.91
		1.50	12.5%	0.097%	27.59%	1 892	50 742	60 371	36.86
Friday	uni-ALT	–	0.0%	0.000%	0.00%	0	196 551	235 083	143.52
	TDALT	1.00	0.0%	0.000%	0.00%	0	116 857	139 175	98.28
		1.15	12.0%	0.096%	14.03%	1 490	50 891	60 550	36.92
		1.50	12.1%	0.098%	30.77%	1 490	50 874	60 531	36.82
Saturday	uni-ALT	–	0.0%	0.000%	0.00%	0	148 331	177 568	100.07
	TDALT	1.00	0.0%	0.000%	0.00%	0	63 717	76 001	47.41
		1.15	10.5%	0.088%	13.97%	2 613	50 042	59 607	36.00
		1.50	10.6%	0.089%	26.17%	2 613	50 036	59 600	35.63
Sunday	uni-ALT	–	0.0%	0.000%	0.00%	0	142 631	170 670	92.79
	TDALT	1.00	0.0%	0.000%	0.00%	0	58 956	70 333	42.96
		1.15	10.4%	0.088%	14.28%	1 753	50 349	59 994	36.04
		1.50	10.5%	0.089%	32.08%	1 753	50 345	59 988	35.74

traffic scenarios. Still, comparing Tabs. 5.1 and 5.4, the gain in performance for dropping correctness is much lower for Germany than for Europe. We assume that this derives from the size of the graph. With increasing graph size, lower bounds get worse as the gap between lower bound distance and time-dependent distance increases. This would also explain why speed-ups for unidirectional ALT are higher for Germany than for Europe.

Timetable Information. Our last tests for TDALT are executed on our timetable data. Table 5.5 depicts the performance of TDALT using this input. We observe lower speed-ups for timetable information than for road networks in general. Unidirectional ALT is about 66% faster than plain Dijkstra. Even worse, switching from uni- to bidirectional ALT does not pay off since we have to use $K > 2.5$ in order to obtain faster queries than for unidirectional ALT. However, such high approximation values yield unacceptable results: The obtained path is twice as long as the shortest. The bad performance in general derives from the fact that lower bounds are poor for railway networks as waiting times can be quite long at certain stations. The bad performance of bidirectional ALT derives from the fact that the second phase is long. Hence, we have to explore a great part of the graph after the first path has been found. That is why the speed-up over a unidirectional variant is—compared to road networks—rather low. We conclude that TDALT works well for road networks but fails on graphs deriving from timetable information for the railways.

Table 5.5: Performance of the time-dependent versions of Dijkstra, unidirectional ALT, and our bidirectional approach in a railways setting.

		Error				Query			
			relative		absolute	#settled nodes			time
method	K	rate	avg	max	max [min]	phase 1	phase 2	phase 3	[ms]
Dijkstra	–	0.0%	0.000%	0.00%	0	–	–	260 095	125.22
uni-ALT	–	0.0%	0.000%	0.00%	0	–	–	127 103	75.28
TDALT	1.0	0.0%	0.000%	0.00%	0	5 710	239 172	262 415	219.60
	1.5	3.2%	0.271%	29.11%	600	5 710	172 934	199 070	167.34
	2.0	12.6%	1.856%	70.55%	1885	5 710	123 274	143 389	120.96
	2.5	24.8%	5.677%	104.07%	3604	5 710	84 441	98 681	84.85
	3.0	37.7%	13.891%	151.46%	5177	5 710	55 452	69 275	60.67
	4.0	60.7%	33.808%	224.96%	7866	5 710	26 347	34 375	30.78
	5.0	72.7%	53.783%	314.58%	7866	5 710	13 516	18 296	16.65
	6.0	79.5%	69.967%	375.92%	7866	5 710	8 751	12 141	11.16
	7.0	82.5%	79.137%	482.27%	7866	5 710	7 014	9 817	8.96
	8.0	84.4%	84.528%	482.27%	7866	5 710	6 220	8 770	8.07
	9.0	85.3%	86.424%	557.60%	7866	5 710	5 880	8 308	7.69
	10.0	85.5%	88.106%	711.75%	7866	5 710	5 820	8 227	7.62
	20.0	86.6%	90.770%	938.97%	7866	5 710	5 711	8 083	7.48

5.5.2 CALT

Default Settings. Unless otherwise stated, we use 32 *avoid* landmarks selected from the extracted core. For CALT we do *not* evaluate timetable networks as Tab. 5.5 indicates that a bidirectional approach based on landmarks does not perform very well on timetable graphs.

Contraction Rates. Table 5.6 shows the performance of TDCALT for different contraction parameters. Note that contraction parameters of $c = 0.0$ and $h = 0$ yield a pure TDALT setup. In this setup, we fix the approximation value K to 1.15, which was found to be a good compromise between speed and quality of computed paths.

Table 5.6: Performance of TDCALT for different contraction rates. Column c denotes the maximum expansion of a bypassed node, column h the hop-limit of added shortcuts. The third column records how many nodes have *not* been bypassed applying the corresponding contraction parameters.

Core			Preprocessing				Exact Query		Approx. Query ($K = 1.15$)				
par.		core	time	space	increase in		#sett.	time	error	relative error		#sett.	time
c	h	nodes	[m]	[B/n]	edges	points	nodes	[ms]	-rate	avg.	max	nodes	[ms]
0.0	0	100.0%	28	256	0.0%	0.0%	2931080	2939.3	40.1%	0.303%	10.95%	250248	188.2
0.5	10	35.6%	15	99	9.8%	21.1%	1165840	1224.8	38.7%	0.302%	11.14%	99622	78.2
1.0	20	6.9%	18	41	12.6%	69.6%	233788	320.5	34.7%	0.288%	10.52%	19719	21.7
2.0	30	3.2%	30	45	9.9%	114.1%	108306	180.0	34.9%	0.287%	10.52%	9974	13.2
2.5	40	2.5%	39	50	9.1%	138.0%	84119	149.7	34.1%	0.275%	8.74%	8093	11.4
3.0	50	2.0%	50	56	8.7%	161.2%	70348	133.2	32.8%	0.267%	9.58%	7090	10.3
3.5	60	1.8%	60	61	8.5%	181.1%	60636	122.3	33.8%	0.280%	8.69%	6227	9.2
4.0	70	1.5%	88	74	8.5%	223.1%	52908	115.2	32.8%	0.265%	8.69%	5896	8.8
5.0	100	1.2%	134	89	8.6%	273.5%	45020	110.6	32.6%	0.266%	8.69%	5812	8.4

As expected, increasing the contraction parameters has a positive effect on query performance. The space overhead first decreases from 256 bytes per node to 41 ($c = 1.0$, $h = 20$), and then increases again. The reason for this is that the core shrinks very quickly, hence we store landmark distances only for 6.6% of the nodes. However, the number of interpolation points for shortcuts increases by up to a factor ≈ 4 with respect to the original graph. Storing these additional points is expensive and explains the increase in space consumption.

It is also interesting to note that the maximum error rate decreases when we allow more and longer shortcuts to be built. We believe that this is due to the fact that long shortcuts decrease the number of settled nodes and have large costs, so at each iteration of TDCALT the key of the backward search priority queue β increases by a large amount. As the algorithm switches from phase 2 to phase 3 when $\mu/\beta < K$, and β increases by large steps, phase 3 starts with a smaller maximum approximation value for the current query μ/β. This is especially true for short distance queries, where the value of μ is small.

Query speed. Table 5.7 reports the results of TDCALT for different approximation values K using the European road network as input. In this experiment we used contraction parameters $c = 3.0$ and $h = 50$, i.e., we allow long shortcuts to be built so to favor query speed. For comparison, we also report the results on the same road network for the time-dependent versions of Dijkstra, unidirectional ALT, and TDALT. Note that we use 32 avoid landmarks.

Table 5.7 shows that TDCALT yields a significant improvement over TDALT with respect to preprocessing space, size of the search space and query times. The latter two figures are improved by one order of magnitude; also, average and maximum error rate

Table 5.7: Performance of time-dependent Dijkstra, unidirectional ALT, TDALT and TDCALT with different approximation values K.

		Preproc.		Error			Query		
		time	space		relative		#settled	#relaxed	time
technique	K	[min]	[B/n]	rate	av.	max	nodes	edges	[ms]
Dijkstra	-	0	0	0.0%	0.000%	0.00%	8 877 158	21 010 600	5 757.4
uni ALT	-	28	256	0.0%	0.000%	0.00%	2 056 190	2,519,840	1 623.3
ALT	1.00	28	256	0.0%	0.000%	0.00%	2 931 080	3 674 870	2 953.3
	1.15	28	256	40.1%	0.303%	10.95%	250 248	301 355	184.4
	1.50	28	256	52.8%	0.734%	21.64%	113 040	136 887	68.1
CALT	1.00	60	61	0.0%	0.000%	0.00%	60 961	356 527	121.4
	1.05	60	61	2.7%	0.010%	3.94%	32 405	184 342	62.5
	1.07	60	61	6.5%	0.030%	4.29%	22 633	126 881	42.1
	1.10	60	61	16.6%	0.093%	7.88%	12 777	69 552	21.9
	1.12	60	61	24.5%	0.158%	7.88%	9 132	48 580	14.9
	1.15	60	61	33.0%	0.259%	8.69%	6 365	32 719	9.2
	1.20	60	61	39.8%	0.435%	12.37%	4 707	23 351	6.4
	1.25	60	61	42.0%	0.549%	15.52%	4 160	20 319	5.4
	1.30	60	61	43.0%	0.611%	16.97%	3 943	19 140	5.0
	1.35	60	61	43.4%	0.649%	18.78%	3 843	18 599	4.9
	1.50	60	61	43.7%	0.679%	20.73%	3 786	18 297	4.8
	1.75	60	61	43.7%	0.682%	27.61%	3 781	18 275	4.8
	2.00	60	61	43.7%	0.682%	27.61%	3 781	18 274	4.8

are smaller when using the same value of the approximation constant K. So, for high K values, TDCALT is more than three orders of magnitude faster than plain Dijkstra. For exact queries, this values is much lower: TDCALT is faster than unidirectional ALT by one order of magnitude, and the improvement over Dijkstra's algorithm is of a factor 30.

Local Queries. For random queries, TDCALT is one order of magnitude faster than TDALT on average. In order to gain insight whether this speed-up derives from small or large distance queries, Fig. 5.7 reports the query times with respect to the Dijkstra rank. These values were gathered on the European road network instance, using contraction parameters as in Tab. 5.7, i.e., $c = 3.5$ and $h = 60$.

Note that we use a logarithmic scale due to the fluctuating query times. Comparing both algorithms, we observe that the hierarchical approach used in TDCALT pays off especially for long distance queries. This is expected, since for small distances TDCALT may result in a simple application of Dijkstra's algorithm, with no speed-up techniques. For sufficiently long distances, however, the median of TDCALT is at least one order of magnitude faster than TDALT. Exact TDCALT is as fast as approximate TDCALT for ranks less than 2^{14}. For higher ranks however, the gap between both variants increases drastically. This is due to the fact that the lower bound quality is worse for long range queries as the gap between time-independent and -dependent path length increases with increasing path length. Comparing exact TDCALT and TDALT, we observe an interesting behavior. TDALT wins for low- and long-range queries while exact TDCALT outperforms approximate TDALT for mid-range queries.

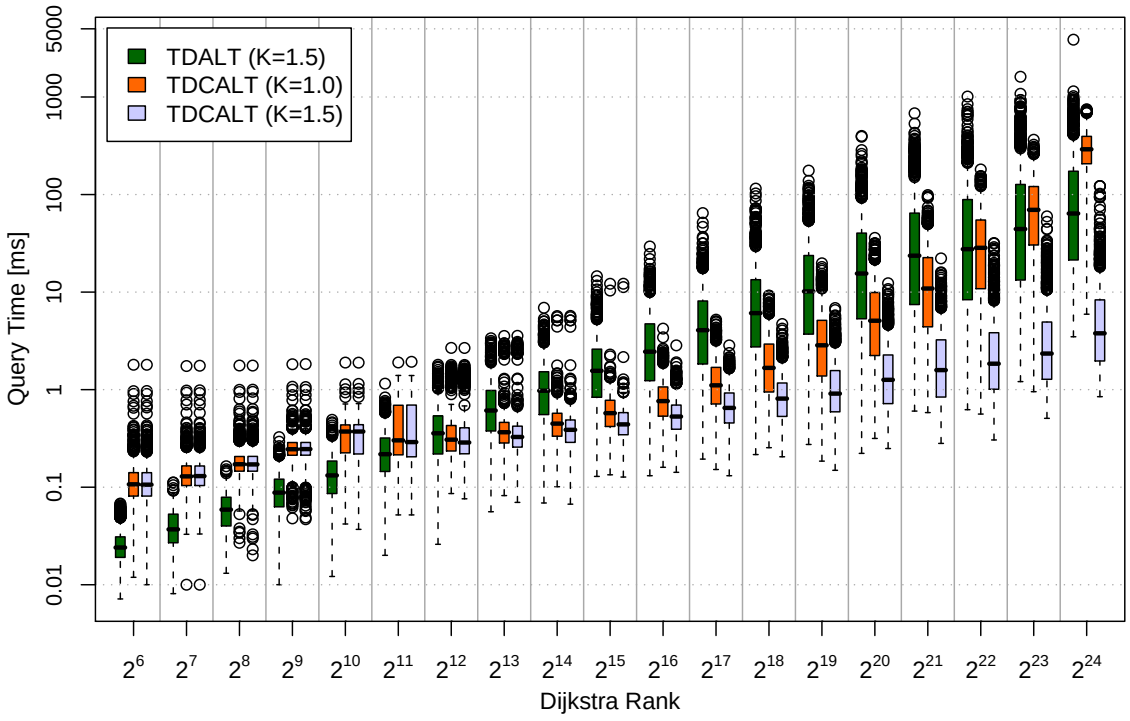


Figure 5.7: Comparison of TDALT and TDCALT using the Dijkstra rank.

Dynamic Updates. In order to evaluate the performance of the core update procedure we generated several traffic jams as follows: for each traffic jam, we select a path in the network covering 4 minutes of uncongested travel time on motorways. Then we randomly select a breakpoint between 6 AM and 9 PM, and for all edges on the path we multiply the corresponding breakpoint value by a factor 5. As also observed in Tab. 4.2, updates on motorway edges are the most difficult to deal with, since those edges are the most frequently used during the shortest path computations, thus they contribute to a large number of shortcuts. In Tab. 5.8 we report average and maximum required time over 1000 runs to update the core in case of a single traffic jam, applying different contraction parameters. We also report the corresponding figures for a batch update of 1000 traffic jams (computed over 100 runs), in order to reduce the fluctuations and give a clearer indication of required CPU time when performing multiple updates. Besides, we measured the average and maximum time required to update the core when modifying a single breakpoint on a motorway edge selected uniformly at random; we also record the corresponding values when modifying 1000 single breakpoints on random motorway edges (computed over 100 runs). As there is no spatial locality when updating a single breakpoint over random edges, this represents a worst-case scenario. Note that in this experiment we limit the length of shortcuts in terms of uncongested travel time (as reported in the third column). This is because in the dynamic scenario the length of shortcuts plays the most important role when determining the required CPU effort for an update operation, and if we allow the shortcuts length to grow indefinitely we may have unpractical update times. Hence, we also report preprocessing space in terms of additional bytes per node, and query times with $K = 1.15$. We remark that Tab. 5.8 only considers the CPU time required to update the core, and does not take into account the computational effort to modify the cost functions for edges at level 0 in the hierarchy, i.e. not belonging to the core. However, this effort is negligible in practice, because the modification of a breakpoint of an edge outside the core has an influence only on the edge itself. Therefore, the update is carried out by simply modifying the corresponding breakpoint value, whereas the core update is considerably more time-consuming.

Table 5.8: CPU time required to update the core in case of traffic jams for different contraction parameters and limits for the length of shortcuts.

cont.		limit	space	traffic jam				single breakpoint				query
				single[ms]		batch[ms]		single[ms]		batch[ms]		
c	h	[min]	[B/n]	av.	max	av.	max	av.	max	av.	max	[ms]
0.0	0	–	256	0.0	0	0	0	0.0	0	0	0	188.2
0.5	10	5	123	0.4	28	372	488	0.1	5	97	166	81.5
		10	121	0.7	49	619	799	0.1	12	183	383	85.2
		15	119	0.7	49	707	1 083	0.1	11	202	407	74.2
		20	119	0.7	49	820	1 200	0.2	59	291	459	73.8
1.0	20	5	82	7.8	229	7 144	8 090	1.8	78	1 853	2 041	34.5
		10	72	21.2	778	20 329	22 734	5.8	371	5 957	9 266	27.1
		15	68	32.1	2 226	27 327	33 313	7.2	427	7 291	11 522	25.4
		20	66	37.0	2 231	30 787	39 470	8.8	1 197	8 476	11 426	22.8
2.0	30	5	88	17.4	290	16 293	17 493	5.7	283	5 019	6 017	33.7
		10	82	90.5	3 868	79 092	85 259	27.6	1 894	24 943	27 501	22.8
		15	79	171.0	4 604	120 018	142 455	49.4	2 451	46 237	58 936	19.7
		20	77	219.7	5 073	187 595	206 569	63.3	5 510	60 940	65 954	16.4

As expected, the effort to update the core becomes more expensive with increasing contraction parameters. First, we consider the scenario where we generate 1000 traffic jams over motorway edges, and modify the cost functions accordingly. For $c = 0.5$, $h = 10$ the updates are very fast, even if we allow long shortcuts (i.e. 20 minutes of uncongested travel time). The average CPU time for an update of 1000 traffic jams is always smaller than 1 second, therefore we are able to deal with a large number of breakpoint modifications in a short time. This is confirmed by the very small average time required to update the core after modifying a random breakpoint on a random motorway edge, which is smaller than 0.2 milliseconds. As we increase the contraction parameters, dynamic updates take longer to deal with. A larger number of long shortcuts is created, therefore update times grow rapidly, requiring several seconds. The average time to update the core after adding 1000 traffic jams with contraction parameters $c = 1.0$, $h = 20$ is at least one order of magnitude larger than the respective values with parameters $c = 0.5$, $h = 10$. Very large updates are feasible in practice only if we limit the length of shortcuts to 5 minutes of uncongested travel time; for most practical applications, however, updates are not very frequent, therefore adding 1000 traffic jams in $\approx$ 30 seconds is reasonably fast. If we consider contraction parameters $c = 2.0$, $h = 30$, then the updates for this scenario may require several minutes; however, limiting the length of shortcuts helps.

Next, we analyze update times for modifications of a single breakpoint over random motorway edges. We observe that they confirm the analysis for the previous scenario (adding 1000 traffic jams). For small contraction parameters (or if we limit shortcuts to a small length in terms of uncongested traveling time), updating the core after modifying one breakpoint requires on average less than 10 milliseconds, whereas if we modify 1000 breakpoints we need less than 10 seconds. For $c = 0.5$, $h = 10$ we can carry out the updates in less than 0.5 seconds. If we allow shortcuts to grow, then updates may require several seconds.

If we compare the time required to update the core after adding 1000 traffic jams with respect to modifying 1000 breakpoints, we see that our update routine greatly benefits from spatial locality of the modified edges: the first scenario is only $\approx$3-4 times slower than the second, but the number of modified edges is larger, because each traffic jam extends over several motorway edges. However, this is expected: as each shortcut is updated only once, modifications on contiguous edges may require no additional effort, if all modified edges belong to the same shortcut. In real world applications, traffic jams typically occur on contiguous edges [Ker04], therefore our update routine should behave better in practice than in worst case scenarios.

Summarizing, we observe a clear trade off between query times and update times depending on the contraction parameters, so that for those applications which require frequent updates we can minimize update costs while keeping query times < 100 ms, and for applications which require very few or no updates we can minimize query times. If most of the edges have their cost changed we can rerun the core edges computation, i.e. recomputing all edges on the core from scratch, which only takes a few minutes.

Traffic days. Next, we evaluate the impact of traffic days on TDCALT. Table 5.9 reports the performance of TDCALT using our German road network with different traffic scenarios as input. We observe a similar behavior as for TDALT. Approximation values > 1.15 do not pay off in terms of query performance and switching from exact to approximate queries yields less improvement for Germany than for Europe. Moreover, it does not pay

Table 5.9: Performance of TDCALT on our German road network instance. *Scenario* depicts the degree of perturbation.

		Preproc.		Error				Query		
		time	space	relative			abs.	#settled	#relaxed	time
scenario	K	[min]	[B/n]	rate	av.	max	max [s]	nodes	edges	[ms]
	1.00	9	50.3	0.0%	0.000%	0.00%	0	2 984	11 316	4.84
Monday	1.15	9	50.3	8.3%	0.051%	11.00%	1 618	1 588	5 303	1.84
	1.50	9	50.3	8.3%	0.052%	17.25%	1 618	1 587	5 301	1.84
	1.00	9	50.3	0.0%	0.000%	0.00%	0	3 190	12 255	5.36
midweek	1.15	9	50.3	8.2%	0.051%	13.84%	2 408	1 593	5 339	1.87
	1.50	9	50.3	8.2%	0.052%	13.84%	2 408	1 592	5 337	1.86
	1.00	8	44.9	0.0%	0.000%	0.00%	0	3 097	12 162	5.21
Friday	1.15	8	44.9	7.8%	0.052%	11.29%	2 348	1 579	5 376	1.82
	1.50	8	44.9	7.8%	0.054%	21.19%	2 348	1 579	5 374	1.82
	1.00	6	27.8	0.0%	0.000%	0.00%	0	1 856	7 188	2.42
Saturday	1.15	6	27.8	4.4%	0.031%	11.50%	1 913	1 539	5 542	1.71
	1.50	6	27.8	4.4%	0.031%	24.17%	1 913	1 539	5 541	1.71
	1.00	5	19.1	0.0%	0.000%	0.00%	0	1 773	6 712	2.13
Sunday	1.15	5	19.1	4.0%	0.029%	12.72%	1 400	1 551	5 541	1.68
	1.50	5	19.1	4.1%	0.029%	17.84%	1 400	1 550	5 540	1.68

off to drop correctness in low traffic scenarios. Still, query performance of TDCALT is excellent. Exact queries are between 280 and 704 times faster—depending on the traffic situation—than plain Dijkstra. Similar to TDALT, the traffic scenario has almost no influence on approximate TDCALT: Less than 10% of the queries are incorrect with paths being up to 39 minutes longer than the shortest. Such paths can be computed 900 times faster than by Dijkstra.

5.5.3 SHARC

Default Settings. Unless otherwise stated, we use $c = 2.5$ as contraction parameter for the all levels. The hop-bound of our contraction is set to 10, the interpolation-bound to 300. If we use landmarks, we select 8 nodes with *avoid*. We do not use more landmarks as for unidirectional ALT, more landmarks do not yield improved query times (cf. Tab. 5.3).

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

Germany. We apply a 5-level partition 4 cells per supercell on levels 0 to 3 and 112 cells on level 4. For this setup, we analyze the impact of degree of perturbation and the quality of different arc-flags computations.

Traffic Days. Table 5.10 reports the performance of time-dependent SHARC with and without landmarks for all profiles we have access to. We use our economical and generous variant for all traffic days and our aggressive version for Saturday and Sunday. Unfortunately, preprocessing of our aggressive variant is too long for the remaining traffic days. For comparison, we also report the performance of static SHARC in a "no traffic" scenario. We also report the speed-up over Dijkstra's algorithm, which settles ≈ 2.2 million nodes in 1.5 seconds on average, independent of the applied traffic scenario.

We observe that the degree of perturbation has a high influence on both preprocessing and query performance of economical SHARC. Preprocessing times increase if perturbation is higher. This is mainly due to our refinement phase that uses partial label-correcting algorithms in order to improve the quality of arc-flags. The increase in overhead derives from the fact that the number of additional interpolation points for shortcuts increases. Analyzing query performance of SHARC, we observe that in a Sunday scenario, SHARC provides speed-ups of up to 787 over Dijkstra. However, this values drops to 60 if a high

Table 5.10: Performance of SHARC on German road network instance. *Scenario* depicts the degree of perturbation, as described above. We here also report the speed-up over Dijkstra.

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

traffic scenario is applied. The reason for this loss in query performance is the bad quality of our Dijkstra-based approximation. If perturbation is higher, upper- and lower-bounds are less tight than in a scenario with only few time-dependent edges. We observe that adding landmarks yields an additional speed-up of up to 4. This is especially useful in high traffic scenarios as query performance is now down to 6.31 ms, which seems to be sufficient for most applications.

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

The query performance of aggressive SHARC is almost independent of the traffic day: For both Saturday and Sunday, we observe query times of ≈ 0.55 ms, a speed-up of about 3 000 over Dijkstra's algorithm. By adding landmarks, we get down to ≈ 0.3 ms and the speed-up is now $\approx 5\,000$. Compared to static SHARC, we observe that aggressive time-dependent SHARC yields almost the same speed-up in terms of settled nodes. However, the number of relaxed edges is lower in static scenarios and, thus, query performance is slightly better. However, we pay a high price in terms of preprocessing times for switching to aggressive SHARC. It seems as the best trade-off between preprocessing effort and query performance is an economical variant combined with landmarks. Here, speed-ups over Dijkstra's algorithm vary between 238 and 2 011, depending on the traffic day.

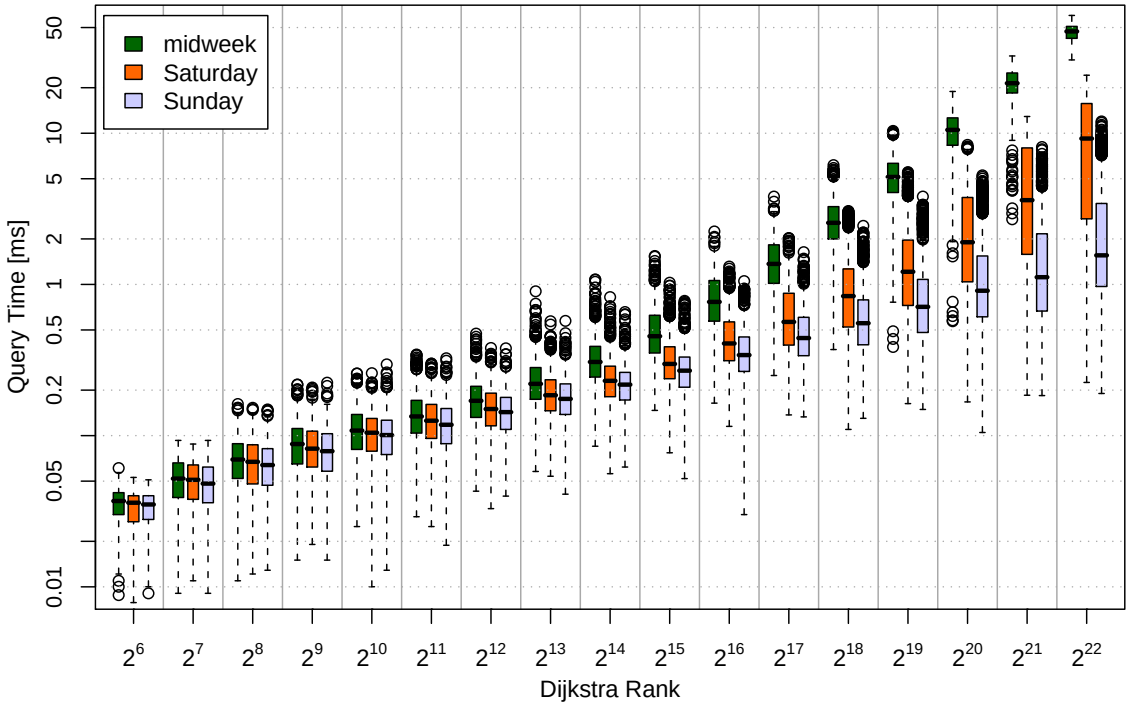


Figure 5.8: Comparison of time-dependent economical SHARC applying a Wednesday, Saturday, and Sunday traffic scenario using the Dijkstra rank methodology.

Local Queries. In order to gain further insights into the impact of traffic days on query performance of economical SHARC, Fig. 5.8 reports the query times of SHARC with respect to the Dijkstra rank. As inputs we use our German road network applying traffic data for midweek, Saturday, and Sunday. Note that we use a logarithmic scale due to outliers. We observe that up to a rank of 2^{12}, query performance is almost independent of the traffic day. However, beyond this rank, high traffic queries (midweek) get slower. The same holds for medium traffic queries (Saturday) beyond ranks of 2^{15}. The reason for this is that for long-range queries the quality of Dijkstra-based arc-flags is bad since upper and lower bounds get worse with increasing distance. Another interesting observation is that queries for a given rank vary by up to 2 orders of magnitudes. Still, all queries are executed in less than 55 ms.

Heuristic SHARC. Table 5.11 reports query performance of SHARC if suboptimal paths are allowed. We observe excellent query performance *and* preprocessing effort of approximate SHARC. Without landmarks, queries are up to 3 163 times faster than Dijkstra. If landmarks are added, this value increases to above 5 400. These values are achieved by a preprocessing effort of not more than 3.5 hours. More importantly, the impact of perturbation fades. For high traffic scenarios, queries are below 0.38 ms, for low traffic scenarios below 0.27 ms. This very good performance comes together with a very good quality of paths. Less than 0.9% of the queries are suboptimal and, more importantly, the found path is at most 50.3 seconds longer than the shortest. This is less than 0.61% of the shortest path length. As time-dependent road networks are based on historical data anyway, such low errors seem reasonable for real-world applications. One could even think of the following approach. We compute economical and approximate SHARC, both variants only differ in arc-flags. As long as the server load is low, we use economical SHARC and switch to approximate SHARC only during peek hours.

Comparing approximate and aggressive SHARC (cf. Tab. 5.10), we observe that query performance is almost the same for both variants. However, the former yields much lower preprocessing times, while the latter guarantees correctness of the found paths.

Table 5.11: Performance of approximate SHARC on German road network instance. Since approximate SHARC may yield suboptimal paths, we report the error-rate, the maximal relative, and maximal absolute error.

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

Table 5.12: Performance of SHARC profile queries. *#nodes reinserted* depicts how many nodes have been reinserted in the queue after removal. *profile/time* shows the quotient of the corresponding figure for profile and time queries. Hence, it shows the slow-down when switching from time to profile queries

		Time-Queries			Profile-Queries					
traffic day	variant	#delete mins	#relaxed edges	time [ms]	#delete mins	profile /time	#re-ins.	#relaxed edges	time [ms]	profile /time
Monday	eco	19 136	101 176	24.55	19 768	1.03	402	208 942	51 122	2 082.6
	heu	810	1 593	0.69	1 071	1.32	24	3 597	1 008	1 460.9
midweek	eco	19 425	104 947	25.06	20 538	1.06	432	222 066	60 147	2 400.3
	heu	818	1 611	0.69	1 100	1.35	27	3 731	1 075	1 548.4
Friday	eco	17 412	92 473	22.07	19 530	1.12	346	204 545	52 780	2 391.9
	heu	769	1 522	0.64	1 049	1.36	21	3 551	832	1 293.2
Saturday	eco	5 284	19 991	5.34	5 495	1.04	44	41 956	3 330	624.0
	agg	721	1 603	0.58	865	1.20	9	3 269	134	232.5
	heu	666	1 336	0.51	798	1.20	8	2 665	98	191.9
Sunday	eco	2 142	6 549	1.86	2 294	1.07	12	13 563	536	288.1
	agg	670	1 439	0.50	781	1.17	5	2 824	57	113.5
	heu	635	1 271	0.46	738	1.16	5	2 449	45	97.9

Profile Queries. Up to now we only reported time query performance. Table 5.12 reports profile query performance of SHARC. Note that profile figures are based on 1 000 random queries and that we also report time query performance for comparison.

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

Europe. Table 5.13 reports the results of economical and approximate SHARC for our European inputs. Unfortunately, it turned out that this input is too big to apply aggressive SHARC. Note that we again report the performance of static SHARC for comparison.

Like for Germany, we observe that the degree of perturbation has a high influence on both preprocessing and query performance of SHARC. Again, preprocessing times increase if more edges are time-dependent. Query performance of economical SHARC on Europe is similar—with respect to speed-up over Dijkstra's algorithm—to Germany. Combined with landmarks, queries times are below 40 ms for all scenarios. These query times can be achieved by investing up to 7 hours of preprocessing, which still seems reasonable for most applications.

Table 5.13: Performance of SHARC on our time-dependent European road network instance. *Scenario* depicts the degree of perturbation, as described above.

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

Comparing approximate SHARC on Germany (cf. Tab. 5.11) and Europe, we observe that speed-ups over Dijkstra's algorithm are almost identical in both cases. However, the quality of paths is worse for Europe than for Germany: Up to 40% of the queries are incorrect and the maximal error increases to 1.6%. A reason for this is that for Europe, shortcuts get more complex than for Germany. Hence, the shortest path may change more often during the day than for Germany. Still, with respect to travel times within Europe, these errors still seem reasonable.

Timetable Information. Table 5.14 shows the performance of time-dependent SHARC using our timetable input. We report the performance of two variants of SHARC: the economical version computes Dijkstra-based arc-flags on all levels, while our aggressive variant computes *exact* flags during the last iteration step. Note that we do *not* use additional techniques in order to improve query performance, e.g., the *avoid binary search* technique (cf. [PSWZ07] for details). For comparison, we also report the results for plain Dijkstra and unidirectional.

Table 5.14: Performance of time-dependent Dijkstra, uni-directional ALT and SHARC using our timetable data as input. Preprocessing times are given in hours and minutes, the overhead in bytes per node. Moreover, we report the increase in edge count over the input. *#delete mins* denotes the number of nodes removed from the priority queue, query *times* are given in milliseconds. *Speed-up* reports the speed-up over the corresponding value for plain Dijkstra.

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

We observe a good performance of SHARC in general. Queries for a specific departure times are up to 29.7 times faster than plain Dijkstra in terms of search space. This lower search space yields a speed-up of a factor of 26.6. This gap originates from the fact that SHARC operates on a graph enriched by shortcuts. As shortcuts tend to have many interpolation points, evaluating them is more expensive than original edges. As expected, our economical variant is slower than the aggressive version but preprocessing is almost 8 times faster. Recall that the only difference between both version is the way arc-flags are computed during the last iteration step. Although the number of heap operations is nearly the same for running one label-correcting algorithm per boundary node as for growing two Dijkstra-trees, the former has to use functions as labels. As composing and merging functions is more expensive than adding and comparing integers, preprocessing times increase significantely.

Comparing time- and profile-queries, we observe that computing $d_*(s,t)$ instead of $d(s,t,\tau)$ yields an increase of about factor $4-7$ in terms of heap operations. Again, as composing and merging functions is more expensive than adding and comparing integers, the loss in terms of running times is much higher. Still, both our SHARC-variants are capable of computing d_* for two random stations in less than 1 second.

Comparison to Road Networks. Comparing the figures from Tabs. 5.12 and 5.14, we observe that speed-ups for time-queries in road networks are higher than in railway networks. However, switching from time to profile queries is cheaper for timetable information. The reason for this is that composing functions needed for timetables is cheaper than those needed for road networks.

5.5.4 Comparison

Finally, we compare all time-dependent algorithms discussed in this chapter among each other. We hereby split our comparison in two parts. Exact queries and approximation. Table 5.15 reports query performance of time-dependent Dijkstra, uni-directional ALT, bidirectional ALT, Core-ALT, and SHARC for our exact setup, while Tab. 5.16 depicts performance if suboptimal paths are allowed. As input we use our time-dependent road networks of Europe (high traffic) and Germany (midweek and Sunday).

Exact Setup. Depending on the scenario, SHARC or TDCALT performs best. While TDCALT is the fastest technique for Germany midweek, SHARC wins for low perturbation scenarios and Europe. However, SHARC tends to have a higher preprocessing effort, regarding both space and time. As soon as costs functions change frequentely, TDCALT is our first choice since it works in dynamic scenarios, while SHARC does not. Summarizing, depending on the size of the graph and degree of perturbation, our presented speed-up techniques are 150 to 5 000 times faster than plain Dijkstra. For all evaluated networks, the query performance is sufficient for most real-world environments.

Approximation. In an approximate scenario, things are clearer. Performance of SHARC is boosted by more than an order of magnitude if we drop correctness. Although ALT and Core-ALT also gain from allowing suboptimal paths, both query performance and quality of paths is (much) worse than for approximate SHARC. We conclude that SHARC is superior if we allow slightly suboptimal paths. Summarizing, approximate SHARC yields speed-ups between 2 700 to 5 420 over Dijkstra's algorithm combined with very low errors.

Table 5.15: Performance of Dijkstra, uni- and bidirectional ALT, Core-ALT, and SHARC in an exact setup.

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

Table 5.16: Performance of Dijkstra, uni- and bidirectional ALT, Core-ALT, and SHARC in an approximation setup.

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

5.6 Concluding Remarks

Review. In this chapter, we presented the first efficient speed-up techniques for exact routing in large time-dependent transportation networks. On the one hand, we generalized our SHARC-algorithm by augmenting several static routines of the preprocessing to time-dependent variants. On the other hand, we showed how to route bidirectionally in time-dependent scenarios. The backward search operates on a time-independent graph and bounds the forward search. As a result, we are able to run fast queries on continental-sized transportation networks of both roads and of railways. Moreover, we are able to compute the distances between two nodes for all possible departure times.

Future Work. Regarding future work, one could think of faster ways of composing, merging, and approximating piece-wise linear functions as this would directly accelerate preprocessing and, more importantly, profile-queries significantly. Aggressive SHARC is the superior technique with respect to query performance. Unfortunately, preprocessing times are impractical in high perturbation scenarios. Since preprocessing is based on building profile graphs being independent of each other, massive parallelization might be an option to preprocess aggressive SHARC in reasonable time for such networks.

Preliminary results in [BDSV09] confirm that CHASE (cf. Section 4.4) can be augmented to time-dependent scenarios as well. However, since Contraction Hierarchies is solely based on shortcuts, the space consumption of this approach is rather high. A challenging task for the future is to reduce the space consumption of this approach.

References. This chapter is based on [NDLS08, Del08, DN08] and the corresponding accepted [Del09] and submitted journal version [NDLS09, DN09].

Chapter 6

Pareto Route Planning

Up to now, we focused on speed-up techniques for Dijkstra's algorithm in *single-criteria* scenarios. The goal was to find the quickest route within a transportation network. However, the quickest route is often not the best one. A user might be willing to accept slightly longer travel times if the costs of the journey are smaller. A common approach to cope with such a situation is to find all *Pareto-optimal* (concerning other metrics than travel times) routes.

In this chapter, we present a multi-criteria variant of SHARC. Unlike other speed-up techniques for Dijkstra's algorithm, SHARC uses a unidirectional query making it the first choice for adapting a single-criteria technique to a multi-criteria scenario. It turns out that multi-criteria SHARC yields speed-ups of a factor of up to 15 000 over a generalized version of Dijkstra's algorithm.

Overview. We start our work on multi-criteria routing with the generalization of our basic ingredients from Chapter 3, located in Section 6.1. Similar to our augmentation to the time-dependent scenario, it turns out that the adaption of contraction is straightforward, while for Arc-Flags, we have to alter the intuition of a set arc-flag slightly. In Section 6.2 we present the preprocessing and the query algorithm of multi-criteria SHARC. The last ingredient for SHARC, arc-flags refinement, is generalized by substituting local single-criteria Dijkstra-searches by multi-criteria ones. The experimental evaluation in Section 6.3 confirms the excellent speed-up achieved by our multi-criteria variant of SHARC. We conclude our work with a summary and possible future work in Section 6.4.

6.1 Augmenting Ingredients

From our augmentation of SHARC to a time-dependent scenario (cf. Chapter 5), we learned that it is sufficient to augment its ingredients, i.e., local Dijkstra-searches, arc-flags computation, and contraction. In this section we show how to augment all these ingredients such that correctness is guaranteed even in a multi-criteria scenario.

6.1.1 Dijkstra

Computing a Pareto set $\mathcal{D}(s,t)$ can be done by a straightforward generalization of Dijkstra's algorithm. For managing the different distance-vectors at each node v, we maintain a list of labels $\texttt{list}(v)$. The list at the source node s is initialized with a label $d(s,s) = (0,\ldots,0)$, any other list is empty. We insert $d(s,s)$ to a priority queue. Then, in each iteration step, we extract the label with the smallest minimum component. Then for all outgoing edges (u,v) a temporary label $d(s,v) = d(s,u) \oplus len(u,v)$ is created. If $d(s,v)$ is not dominated by any of the labels in $\texttt{list}(v)$, we add $d(s,v)$ to $\texttt{list}(v)$, add

$d(s, v)$ to the priority queue, and remove all labels from $\texttt{list}(v)$ that are dominated by $d(s, v)$. We may stop the query as soon as $\texttt{list}(t) \neq \emptyset$ and all labels in the priority queue are dominated by all labels in $\texttt{list}(t)$.

Pareto Path Graphs. In the following, we construct *Pareto path graphs (PPG)* by computing $\mathcal{D}(s, u)$ for a given source s and all nodes $u \in V$, with our generalized Dijkstra algorithm. We call an edge (u, v) a *PPG*-edge if $L \in \texttt{list}(u)$ and $L' \in \texttt{list}(v)$ exist such that $L \oplus len(u, v) = L'$. In other words, (u, v) is a PPG-edge if it is part of at least one Pareto-optimal path from s to v. Note that by this notion one of two parallel edges can be a PPG-edge while the other one is not.

6.1.2 Arc-Flags

In a single-criteria scenario, an arc-flag $AF_C(e)$ denotes whether e has to be considered for a shortest-path query targeting a node within C. In other words, the flag is set if e is important for (at least one target node) in C. In Section 5.1, we adapted Arc-Flags to a time-dependent scenario by setting a flag to true as soon as it is important for at least one departure time. The adaption to a multi-criteria scenario is very similar: we set an arc-flag $AF_C(e)$ to true, if e is important for at least one Pareto path targeting a node in C.

Unlike in the time-dependent scenario—where we needed approximations—we can settle for the straightforward approach for augmenting Arc-Flags. We build a Pareto path graph in $\overleftarrow{G}$ for all boundary nodes $b \in B_C$ of all cells C at level i. We stop the growth as soon as all labels in the priority queue are dominated by all labels $L(v, b)$ assigned to the nodes v in the supercell of C. Then we set $AF_C(u, v) = \text{true}$ if (u, v) is a PPG-edge for at least one PPG grown from all boundary nodes $b \in B_C$. Moreover, we set all own-cell flags to true.

Lemma 6.1. *Pareto Arc-Flags is correct.*

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

Let s and t be arbitrary nodes, and let b_T be the last boundary node of region T on p_{st}. We know that the subpath from s to b_T is a Pareto path. Hence, all edges on p_{st} are PPG-edges of the PPG built from b_T during preprocessing. Thus, all edges e on p_{st} have $AF_T(e) = \text{true}$. □

Multi-Level Arc-Flags. SHARC is based on multi-level Arc-Flags. Hence, we need to augment the concept of multi-level Arc-Flags to a multi-criteria scenario. Again, the augmentation is similar to the one to time-dependent networks. We describe a two-level setup which can be extended to a multi-level scenario easily.

Preprocessing is done as follows. Arc-flags on the upper level are computed as described above. For the lower flags, we grow a PPG in $\overleftarrow{G}$ for all boundary nodes b on the lower level. We may stop the growth as soon as all labels attached to the in the supercell of C dominate all labels in the priority queue. Then, we set an arc-flag to true if the edge is a PPG edge of at least one Pareto path graph.

Lemma 6.2. *Pareto multi-level Arc-Flags is correct.*

Proof. The proof is very similar to the proof of Lemma 5.3. Again, we show the correctness of two-level Arc-Flags as the generalization to a multi-level scenario is straightforward.

Let $p_{st} = (e_0, \ldots, e_k)$ be an arbitrary s–t Pareto path. Let $\mathbb{c}_i(u)$ be the cell of u in level i, where 0 denotes the lower, 1 the upper level. An edge (u, v) is part of the upper level if $\mathbb{c}_1(u) \neq \mathbb{c}_1(t)$ and $\mathbb{c}_1(v) \neq \mathbb{c}_1(t)$. According to Lemma 6.1, we know that all edges being part of the upper level have $AF_{\mathbb{c}_1(t)} = \mathsf{true}$. Let b be the last boundary node of $\mathbb{c}_0(t)$ on p^{τ}_{st}. Since we have grown a Pareto path graph from b during preprocessing until all nodes in $\mathbb{c}_1(t)$ have their final Pareto labels assigned, edges being part of the lower level have proper arc-flags assigned. □

6.1.3 Contraction

Our augmented Pareto contraction routine is very similar to our static one from Section 3.4. Again, we first reduce the number of nodes by removing unimportant ones and—in order to preserve Pareto sets between non-removed nodes—add shortcuts to the graph. Then, we apply an edge-reduction step that removes unneeded shortcuts.

Node-Reduction. We iteratively *bypass* nodes until no node is *bypassable* any more. To bypass a node u we first remove u, its incoming edges I and its outgoing edges O from the graph. Then, for each combination of $e_i \in I$ and $e_o \in O$, we introduce a new edge with label $len(e_i) \oplus len(e_o)$. Note that we explicitely allow multi-edges. Also note that contraction gets more expensive in a multi-criteria scenario due to multi-edges.

Like for time-independent and time-dependent node reduction, we use a heap to determine the next bypassable node. Let $\#shortcut$ of *new* edges that would be inserted into the graph if u was bypassed and let $\zeta(x) = \#\text{shortcut}/(|I| + |O|)$ be the *expansion* of node u. Furthermore, let $h(u)$ be the hop number of the hop-maximal shortcut- Then we set the key of a node u within the heap to $h(u) + 10 \cdot \zeta(u)$, smaller keys have higher priority.

To keep the costs of shortcuts limited we do not bypass a node if its removal results in a hop number greater than h. We say that the nodes that have been bypassed belong to the *component*, while the remaining nodes are called *core-nodes*.

Corollary 6.3. *Pareto node-reduction preserves Pareto sets between core nodes.*

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

Edge-Reduction. We identify unneeded shortcuts by growing a Pareto path graph from each node u of the core. We stop the growth as soon as all neighbors v of u have their final Pareto-set assigned. Then we may remove all edges from u to v whose label is dominated by at least one of the labels $\texttt{list}(v)$. In order to limit the running time of this procedure, we restrict the number of priority-queue removals to 1 000.

Corollary 6.4. *Pareto edge-reduction preserves Pareto sets between core nodes.*

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

6.2 Pareto SHARC

With the augmented ingredients, we are ready to augment SHARC. Remarkably, the augmentation is now very similar to time-dependent SHARC from Section 5.4. During perprocessing, we apply the augmented routines from Section 6.1 instead of their time-independent counterparts, while the query is again a modified multi-criteria Dijkstra pruning unimportant edges.

6.2.1 Preprocessing

Initialization. In a first step, we again apply our methods being independent of the applied metric: We remove 1-shell nodes from the graph since we can directly assign correct arc-flags to all edges adjacent to 1-shell nodes. Moreover, we perform a multi-level partitioning of the input.

Iteration. After the initialization, our iterative process starts. Each iteration step is again divided into two parts: contraction and arc-flag computation.

Contraction. First, we apply a contraction step according to Section 6.1. Like for time-independent and time-dependent SHARC, we have to use *cell-aware* contraction, i.e., a node u is never marked as bypassable if any of its neighboring nodes is *not* in the same cell as u.

Arc-Flags. We have to set arc-flags for all edges of our output-graph, including those which we remove during contraction. Like for static SHARC, we can set arc-flags for all removed edges automatically. We set the arc-flags of the current and all higher levels depending on the tail u of the deleted edge. If u is a core node, we only set the own-cell flag to true (and others to false) because this edge can only be relevant for a query targeting a node in this cell. If u belongs to the component, all arc-flags are set to true as a query has to leave the component in order to reach a node outside this cell. Setting arc-flags of those edges not removed from the graph is more time-consuming since we apply the preprocessing of multi-level Pareto Arc-Flags from 6.1.

Finalization. The last phase of our preprocessing-routine assembles the output graph. It contains the original graph, shortcuts added during preprocessing and arc-flags for all edges of the output graph. However, some edge may have no arc-flag set to true. As these edges are never relaxed by our query algorithm, we directly remove such edges from the output graph.

6.2.2 Query

Augmenting the SHARC-query is straightforward. For computing a Pareto-set $\mathcal{D}(s,t)$, we use a modified multi-criteria Dijkstra (Section 6.1) that operates on the output graph. The modifications are then the same as for the single-criteria variant of SHARC: When settling a node n, we compute the lowest level i on which n and the target node t are in the same supercell. Moreover, we consider only those edges outgoing from n having a set arc-flag on level i for the corresponding cell of t. In other words, we prune edges that are not important for the current query. The stopping criterion is the same as for a multi-criteria Dijkstra.

6.2.3 Correctness

Theorem 6.5. *Pareto SHARC is correct.*

Proof. Correctness of Pareto SHARC can be shown equivalent to time-independent and time-dependent SHARC. Again, we need to show that an equivalent variant of Lemma 4.3 also hold in our multi-criteria setting. Analyzing the proof of Lemma 4.3, one may notice that it is based on two facts: Contraction preserves distances and multi-level Arc-Flags is correct. According to Corollaries 6.3 and 6.4, and Lemma 6.2, our augmented ingredients fulfill these requirements. Hence, our multi-criteria variant of SHARC is correct as well. □

6.2.4 Optimizations

We also apply our optimizations for time-independent SHARC to our multi-criteria variant. The most important one is again refinement of arc-flags. However, we also reorder nodes to improve locality and compress arc-flags.

Refinement of Arc-Flags. Recall that refinement of arc-flags tries to improve those flags set to `true` during the contraction process. This is achieved by propagating flags of edges outgoing from high-level nodes to those outgoing from low-level nodes. In a time-independent scenario, we grow shortest path trees to find the so called exit nodes of each node, while in a time-dependent scenario, we use profile graphs to determine these nodes. Hence, we now grow Pareto path graphs from each node. The propergation itself stays untouched. See Section 5.4 for details.

Like for profile graphs, growing Pareto path graphs can get expensive. Hence, we limit the growth to $n \log(n)/|V_l|$, where V_l denotes the nodes in level l, priority-queue removals. In order to preserve correctness, we then may only propagate the flags from the exit nodes to u if the stopping criterion is fulfilled before this number of removals.

Lemma 6.6. *Refinement of arc-flags is correct.*

Proof. The only difference between time-dependent and multi-criteria refinement is that we grow Pareto path instead of profile graphs in order to find exit nodes of a node. Hence, we can directly adapt the proof of Lemma 5.16 in order to prove the correctness of Lemma 6.6. □

Improved Locality. In order to improve query performance of multi-criteria SHARC, we again increase cache efficiency of the output graph by reordering nodes according to the level they have been removed at from the graph.

6.3 Experiments

In this section, we present our experimental evaluation. To this end, we evaluate the performance of multi-criteria SHARC using several types of graphs with different types and numbers of metrics. Our implementation is written in C++ using solely the STL at some points. As priority queue we use a binary heap. Our tests were executed on one core of an AMD Opteron 2218 running SUSE Linux 10.3. The machine is clocked at 2.6 GHz, has 16 GB of RAM and 2 x 1 MB of L2 cache. The program was compiled with GCC 4.2, using optimization level 3.

6.3.1 Road Networks

Inputs. We use four real world road networks for our experimental evaluation. The first one is the largest strongly connected component of the road network of Western Europe, provided by PTV AG for scientific use. It has approximately 18 million nodes and 42.6 million edges. However, it turns out this input is too big for finding all Pareto routes. Hence, we also use three smaller network, namely the road network of Luxemburg consisting of 30 661 nodes and 71 619 edges, a road network of Karlsruhe and surrounding (77 740 nodes, 196 327 edges), and the road network of the Netherlands (892 392 nodes, 2 159 589 edges). As metrics we use travel times for fast cars/slow trucks, costs (toll + fuel consumption), travel distances, and unit lengths. Note that the last metric is a rather synthetic one. However, it is part of the DIMACS benchmark testsuite [DGJ06].

Default Setting. For Europe, we use a 6-level partition with 4 cells per supercell on levels 0 to 3, 8 cells per supercell on level 4, and 104 cells on level 5. A 3-level partition is applied when using Luxemburg and Karlsruhe as input, with 4 cells per supercell on levels 0 and 1, and 56 cells on level 2. For the netherlands, we apply a 4-level partition, with 4 cells per supercell on levels 0 and 1, 8 cells on level 2, and 112 cells on level 3. We use $c = 2.5$ as contraction parameter for the all levels for both inputs. The hop-bound of our contraction is set to 10. To keep preprocessing times limited, we use an economical variant (cf. Section 4.5), i.e., we compute arc-flags only for the topmost level and do not refine arc-flags for the lowest two levels. For static single-criteria SHARC, this reduces preprocessing times by a factor of 3 (81 minutes → 27 minutes), but query performance is still good enough (0.61 ms instead of 0.28 ms). In the following, we report preprocessing times and the overhead of the preprocessed data in terms of *additional* bytes per node. Moreover, we provide the average number of settled nodes, i.e., the number of nodes taken from the priority queue, and the average query time. For random s-t queries, the nodes s and t are picked uniformly at random. All figures in this paper are based on 1 000 random s-t queries and refer to the scenario that only distance labels of the Pareto paths have to be determined, without outputting a complete description of the paths.

Full Pareto-Setting. Table 6.1 depicts the performance of multi-criteria SHARC on our Luxemburg and Karlsruhe instance in a full Pareto bicriteria setting. For comparison, we also report the performance of single-criteria SHARC on all five metrics. We observe a good performance of multi-criteria SHARC in general. Preprecessing times are less than 15 minutes which is sufficient for most applications. Interestingly, the speed-up over Dijkstra's algorithm with respect to query times even increases when switching to multi-criteria SHARC. However, comparing single- and multi-criteria, we observe that query performance highly depends on the size of the Pareto set at the target node. For similar metrics (fast car and slow truck), bicriteria queries are only 3 times slower than a single-criteria queries. This stems from the fact that the average size of the Pareto-set is only 2. If more labels are created, like for fast car + costs, multi-criteria queries are up to 673 times slower. Even worse, this slow-down increases even further when we apply our Karlsruhe network. Here, the queries are up to 3 366 times slower. Summarizing, the number of labels created, and thus, the loss in query performance over single-criteria queries, is too high for using a full Pareto-setting for a big input like Western Europe. Hence, we show in the following how to reduce the number of labels such that "unimportant" Pareto-routes are pruned as early as possible.

Table 6.1: Performance of single- and multi-criteria SHARC applying different metrics for our Luxemburg and Karlsruhe inputs. *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, we report the number of labels created at the target node, the number of nodes removed from the priority queue, execution times in milliseconds, and speed-up over Dijkstra's algorithm.

		Prepro		Query				
		time	space	target	#delete	speed	time	speed
input	metrics	[h:m]	[B/n]	labels	mins	up	[ms]	up
Luxemburg	fast car	< 0:01	12.4	1.0	138	112	0.03	114
	slow truck	< 0:01	12.6	1.0	142	109	0.03	111
	costs	< 0:01	12.0	1.0	151	101	0.03	96
	distances	< 0:01	14.7	1.0	158	97	0.03	87
	unit	< 0:01	13.7	1.0	149	106	0.03	96
	fast car + slow truck	0:01	14.7	2.0	285	106	0.09	100
	fast car + costs	0:04	24.1	29.6	4 149	97	6.49	263
	fast car + dist.	0:14	22.3	49.9	8 348	51	20.21	78
	fast car + unit	0:06	23.7	25.7	4 923	57	5.13	112
	costs + dist.	0:02	20.4	29.6	3 947	78	4.87	119
Karlsruhe	fast car	<0:01	12.4	1.0	206	189	0.04	188
	slow truck	<0:01	12.7	1.0	212	187	0.04	178
	costs	<0:01	15.4	1.0	244	156	0.05	129
	distances	<0:01	15.7	1.0	261	146	0.06	119
	unit	<0:01	14.1	1.0	238	165	0.05	147
	fast car + slow truck	0:01	15.3	1.9	797	98	0.26	108
	fast car + costs	1:30	26.6	52.7	15 912	118	80.88	184
	fast car + dist.	3:58	23.6	99.4	31 279	79	202.15	153
	fast car + unit	0:17	26.6	27.0	11 319	91	16.04	200
	costs + dist.	1:11	21.9	67.2	19 775	84	67.75	160

Reduction of Labels. As observed in Tab 6.1, the number of labels assigned to a node increase with growing graph size. In order to efficiently compute Pareto-paths for our European road network, we need to reduce the number of labels both during preprocessing and queries. We achieve this by tightening the definition of dominance. Therefore, we define the travel time metric to be the dominating metric W. Then, our tightened definition of dominance is as follows: Besides the constraints from Section 2.2, we say a label $L = (W, w_1, \ldots, w_{k-1})$ dominates another label $L' = (W', w'_1, \ldots, w'_{k-1})$ if $W \cdot (1+\epsilon) > W'$ holds. In other words, we only allow Pareto-paths which are up to ϵ times longer (with respect to the dominating metric). Note that by this notion, this has to hold for all sub-paths as well.

Table 6.2 reports the performance of bicriteria SHARC using the tightened definition of dominance (with varying ϵ) during preprocessing *and* queries. As input, we use three networks: Karlsruhe, the Netherlands, and Europe. We here focus on the probably most important combination of metrics, namely fast car travel time and costs). We observe that our additional constraint works: Preprocessing times decrease and query performance gets much better. However, as expected, very small ϵ values yield on a small subset of the Pareto-set and high ϵ values yield high preprocessing times. For small and mid-size inputs, i.e., less than 1 million nodes, setting ϵ to 0.5 yields a reasonable amount of Pareto paths combined with good preprocessing times and good query performance. Unfortunately, for

Table 6.2: Performance of bi-criteria SHARC with varying ϵ using travel times and costs as metrics. The inputs are Karlsruhe, the Netherlands, and Europe.

	Karlsruhe					The Netherlands				
	Prepro		Query			Prepro		Query		
	time	space	target	#delete	time	time	space	target	#delete	time
ϵ	[h:m]	[B/n]	labels	mins	[ms]	[h:m]	[B/n]	labels	mins	[ms]
0.000	< 0:01	14.3	1.0	265	0.09	0:01	15.2	1.0	452	0.21
0.001	< 0:01	14.3	1.1	271	0.09	0:01	15.3	1.1	461	0.21
0.002	< 0:01	14.3	1.1	302	0.10	0:01	15.2	1.2	489	0.22
0.005	< 0:01	14.5	1.3	307	0.11	0:01	15.3	1.4	517	0.24
0.010	< 0:01	14.6	1.5	322	0.11	0:01	15.4	1.7	590	0.27
0.020	< 0:01	14.8	1.9	387	0.13	0:01	15.6	2.2	672	0.32
0.050	< 0:01	15.7	2.5	495	0.18	0:02	16.5	3.3	1 009	0.51
0.100	< 0:01	16.6	4.2	804	0.33	0:04	17.3	4.8	1 405	0.82
0.200	0:01	17.9	6.4	1,989	1.86	0:09	18.5	7.2	2 225	1.67
0.500	0:02	20.7	14.0	3 193	3.61	0:39	20.6	12.8	4 227	4.85
1.000	0:13	23.1	24.0	9 072	14.86	3:44	22.5	20.0	12 481	26.85
∞	1:30	26.6	52.7	15 912	80.88	>24:00	–	–	–	–

	Europe				
	Prepro		Query		
	time	space	target	#delete	time
ϵ	[h:m]	[B/n]	labels	mins	[ms]
0.000	0:53	18.4	1.0	3 299	2.64
0.001	1:00	18.5	1.1	3 644	4.12
0.002	1:03	18.5	1.2	4 340	7.12
0.005	1:18	18.6	1.4	5 012	11.34
0.010	1:58	18.9	2.4	9 861	19.20
0.020	4:10	19.3	5.0	24 540	48.05
0.050	14:12	20.1	23.4	137 092	412.74
0.100	>24:00	–	–	–	–

our European input, only $\epsilon \leq 0.02$ yields practical preprocessing and query times.

Further Reduction. As observable in Tab. 6.2, our approach for reducing the number of labels is only practical for very small ϵ if we use Europe as input. As we are interested in paths with bigger ϵ values as well, we add another constraint, called *pricing*, in order to define dominance. Besides the constraints from Section 2.2 and from above, we say a label $L = (W, w_1, \ldots, w_{k-1})$ dominates another label $L' = (W', w'_1, \ldots, w'_{k-1})$ if $\sum_i w'_i / \sum_i w_i < W/W' \cdot \gamma$ holds. In other words, we only accept labels with longer travel times if this results in a decrease in the other metrics under consideration.

Table 6.3: Performance of bi-criteria SHARC with varying γ. ϵ is fixed to 0.5.

	Prepro		Query		
	time	space	target	#delete	time
γ	[h:m]	[B/n]	labels	mins	[ms]
1.100	0:58	19.1	1.2	2 538	1.81
1.050	1:07	19.6	1.3	3 089	2.21
1.010	1:40	20.4	1.7	4 268	3.16
1.005	2:04	20.6	1.9	5 766	4.11
1.001	3:30	20.8	2.7	7 785	6.11
1.000	7:12	21.3	5.3	19 234	35.42
0.999	15:43	22.5	15.2	87 144	297.20
0.995	>24:00	–	–	–	–

With this further tightened definition of label dominance, we are finally ready to run multi-criteria queries on our European instance. Table 6.3 shows the performance of multi-

criteria SHARC with varying γ in a bicriteria scenario (travel times + costs) for Europe. Note that we *fix* $\epsilon = 0.5$. It turns out that our additional constraints work. With $\gamma = 1.0$, we create 5.3 labels in 35.42 ms on average at the target node, being sufficient for practical applications. Preprocessing times are still within reasonable times, i.e., less than 8 hours. If we want to generate more labels, we could set $\gamma = 0.999$. However, query times drop to almost 300 ms and preprocessing increases drastically. Summarizing, bicriteria queries for travel times and travel costs are possible if we use $\gamma = 1.0$ and $\epsilon = 1.5$.

Similar Metrics. Our last experiment for road networks deals with the following scenario. We are interested in the quickest route for different types of vehicles. Hence, we perform multi-criteria queries on metrics all based on travel times. More precisely, we use typical average speeds of fast cars, slow cars, fast trucks, and slow trucks. Due to the very limited size of the resulting Pareto-sets, we afford not to use our tightened definition of dominance for this experiment. Tab. 6.4 shows the performance of multi-criteria SHARC in such a single-, bi- and tri-, and quad-criteria scenario. We observe that a full Pareto-

Table 6.4: Performance of multi-criteria SHARC applying different travel time metrics. The inputs are the Netherlands and Europe.

		PREPRO		QUERY				
		time	space	target	#del.	speed	time	speed
input	metrics	[h:m]	[B/n]	labels	mins	up	[ms]	up
The Netherlands	fast car(fc)	0:01	13.7	1.0	364	1 215	0.11	1 490
	slow car(sc)	0:01	13.8	1.0	359	1 263	0.10	1 472
	fast truck(ft)	0:01	13.9	1.0	365	1 189	0.10	1 332
	slow truck(st)	0:01	13.9	1.0	363	1 214	0.10	1 306
	fc+st	0:05	16.2	2.2	850	1 223	0.33	2 532
	fc+ft	0:05	16.2	2.0	768	1 233	0.29	2 371
	fc+sc	0:05	15.5	1.2	520	1 163	0.19	1 896
	sc+st	0:05	16.2	1.9	742	1 182	0.29	2 009
	sc+ft	0:05	16.2	1.7	679	1 155	0.26	1 850
	ft+st	0:05	15.7	1.3	551	1 147	0.21	1 692
	fc+sc+st	0:06	19.0	2.3	867	1 244	0.37	2 580
	fc+sc+ft	0:06	18.9	2.0	764	1 231	0.32	2 385
	sc+ft+st	0:06	19.0	1.9	740	1 190	0.30	2 134
	fc+sc+ft+st	0:07	21.8	2.5	942	1 152	0.43	2 362
Europe	fast car(fc)	0:25	13.7	1.0	1,457	6 177	0.69	7 536
	slow car(sc)	0:24	13.8	1.0	1,367	6 584	0.67	7 761
	fast truck(ft)	0:23	13.9	1.0	1,486	6 057	0.71	7 324
	slow truck(st)	0:25	13.9	1.0	1,423	6 325	0.68	7 647
	fc+st	2:24	18.3	3.8	6 819	6 196	4.35	12 009
	fc+ft	1:30	18.3	3.2	5 466	6 162	3.91	11 349
	fc+sc	1:08	17.1	2.0	4 265	6 012	2.26	10 234
	sc+st	1:53	18.1	3.3	5 301	5 741	4.02	10 874
	sc+ft	1:49	16.2	3.2	5 412	5 488	3.65	10 663
	ft+st	1:28	17.4	3.0	5 157	5 669	3.73	12 818
	fc+sc+st	2:41	20.3	4.5	8 713	6 513	5.70	12 741
	fc+sc+ft	2:47	21.5	3.9	7 133	5 989	4.87	12 144
	sc+ft+st	2:59	22.0	4.2	7 962	6 348	5.12	13 412
	fc+sc+ft+st	4:41	24.5	6.2	12 766	6 415	7.85	15 281

setting is feasible if metrics are similar to each other, mainly because the number labels is very limited. Remarkably, the speed-up of multi-criteria SHARC over multi-criteria Dijkstra is even *higher* than in a single-criteria scenario. The slow-down in preprocessing times and query performance is quite high but still, especially the latter is fast enough for practical applications. Quadro-criteria queries need less than 8 ms for our European road networks, being sufficient for most applications. A generalized Dijkstra needs about 120 seconds on average for finding a Pareto-set in this quad-criteria scenario. This speed-up of more than 15 000 is achieved by a preprocessing taking less than 5 hours.

6.3.2 Synthetic Inputs

In order to show the good performance of multi-criteria SHARC in other networks than road graphs, we also run some tests on a synthetic input. More precisely, we evaluate unit disk graphs which are widely used for experimental evaluations in the field of sensor networks. Such graphs are obtained by arranging nodes uniformly at random on the plane and connecting nodes with a distance below a given threshold. In our setup, we use graphs with about 1 000 000 nodes and an average degree 5. We apply two metrics: the distance between nodes according to their embedding and unit lengths. The results can be found in Tab. 6.5.

Table 6.5: Performance of single- and bi-criteria SHARC applying different metrics for our synthetic input.

	Prepro		Query				
	time	space	target	#delete	speed	time	speed
metrics	[h:m]	[B/n]	labels	mins	up	[ms]	up
unit	0:01	16.8	1.0	324	1 495	0.13	1 590
distances	0:01	12.7	1.0	313	1 598	0.13	1 667
unit+dist	0:20	20.2	24.2	9 158	1 276	9.23	3 837

Like for road networks, SHARC performs well on these inputs. The preprocessing of the bicriteria is about 20 times slower than for singlecriteria SHARC, but still less than 30 minutes. Again, speed-ups over a bicriteria Dijkstra is even higher than in a single-criteria scenario.

6.4 Concluding Remarks

Review. In this chapter, we presented an efficient speed-up technique for computing multi-criteria paths in large-scale road networks. By augmenting single-criteria routines to multi-criteria versions, we were able to present a multi-criteria variant of SHARC. Several experiments confirm that speed-ups over a multi-criteria Dijkstra are at least the same as in a single-criteria scenario, in many cases the speed-up with respect to query times is even higher. However, if metrics differ strongly, the number of possible Pareto-routes increases drastically making preprocessing and query times impractical for large instances. By tightening the definition of dominance, we are able to prune unimportant Pareto-routes both during preprocessing and queries. As a result, SHARC provides a feasible subset of Pareto-routes in a continental-sized road network. Moreover, tests on synthetic data show the robustness of multi-criteria SHARC.

Future Work. Regarding future work, one can think of other ways for pruning the Pareto-set. Maybe other constraints yield better subsets of the Pareto-set computable in reasonable time as well. An open challenging problem is the adaption of multi-criteria SHARC to a fully realistic timetable information system like the ones presented in [MS07, DMS08]. Due to the experimental results presented here, we are optimistic that this should work pretty well.

References. This chapter is based on [DW09].

Chapter 7

Conclusion

In this work, we introduced the first efficient speed-up techniques for routing in augmented, i.e., time-dependent and multi-criteria, scenarios. Therefore, we followed the paradigm of algorithm engineering by designing, analyzing, implementing, and evaluating speed-up techniques for Dijkstra's algorithm.

For augmentation, we pursued a systematic approach. We identified basic ingredients and analyzed drawbacks of those. In a study on time-independent route planning we showed that by incorporating contraction, i.e., a hierarchical component, into goal-directed speed-up techniques, the known drawbacks of the latter diminish. In fact, we obtain speed-up techniques which can compete with the fastest known techniques. However, due to our paradigm of basic ingredients, their augmentations are easier than for other techniques. As a result, we were able to present augmented variants of SHARC and CALT. The former technique relies on unidirectional search, while CALT performs *bidirectional* search. The problem of unknown arrival times is solved by running a *time-independent* search from the target bounding the nodes the *time-dependent* forward search has to visit. With these new techniques, we are able to compute time-dependent shortest paths up to 5 000 times faster than plain Dijkstra. Moreover, CALT can even handle the scenario where cost functions change due to unexpected traffic jams.

We not only introduced the first efficient speed-up techniques for routing in augmented scenarios. In addition, we further accelerated the fastest known techniques for route planning in road networks. We achieved this additional speed-up by adding goal-direction via Arc-Flags to the fastest hierarchical techniques. The additional preprocessing effort remains limited since it turns out that it is sufficient to use goal-direction only on a small subgraph constituting the upper part the hierarchy.

Besides that, we presented an experimental study on the robustness of speed-up techniques. By evaluating networks other than road networks, we gained further insights into the performance of speed-up techniques in general. It turned out that some techniques are more robust than others. Two of those robust techniques are introduced in this thesis: CALT, a combination ALT and contraction, and CHASE, a combination of Contraction Hierarchies and Arc-Flags.

Future Work. An interesting question deriving from our study on robustness is whether we can somehow *predict* the performance of a speed-up technique on a network. A good starting point would be to develop *indices* that evaluate the network in terms of how well our basic ingredients from Chapter 3 can exploit certain characteristics of the network. Although we already did some work on such indices [BDW07b], it seems as if this preliminary study is more a starting than an end point for network analysis with respect to the performance of speed-up techniques.

One outcome of route planning in road networks is that adding shortcuts to the graph was the key idea in order to preprocess such huge networks fast. However, we observed that shortcuts are much more space-consuming in time-dependent networks. In fact, the adaption of Contraction Hierarchies [BDSV09] suffers from this fact: The space consumption increases tremendously when switching to time-dependent scenarios. On the long run, we need to develop a technique that does not rely on shortcuts at all. An interesting starting point would be a stripped variant of time-dependent SHARC. Moreover, query performance of time-dependent SHARC is excellent but growing profile graphs during preprocessing is time-consuming. However, the constructions of these graphs are almost independent from one another. Hence, it would be interesting whether massive parallelism might help here. Finally, developing routines for updating arc-flags, in case cost functions change, is a challenging task as well.

This work introduced a multi-criteria variant of SHARC. However, this work can more be seen as a starting point on multi-criteria routing. Is there a chance of finding better and more Pareto routes? How do we find good alternative routes in single-criteria scenarios and how much do such routes differ from Pareto routes? Moreover, what about multi-criteria search in time-dependent scenarios? Finally, the adaption of multi-criteria SHARC to a real-world timetable information system as presented in [DMS08] is a challenging task as well.

Up to now and including this thesis, research focused on fast route planning in road networks *or* public transportation. However, on the long run, we are interested in planning routes in a multi-modal scenario: We start by car to reach the nearest train station, ride the train to the airport, fly to an airport near our destination and finally take a taxi. In other words, we need to incorporate public transportation in road networks. We are optimistic that the time-dependent methods from Chapter 5 are helpful for this task.

Bibliography

[BD08] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. In Ian Munro and Dorothea Wagner, editors, *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, April 2008.
(Cited on page 70.)

[BD09] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 2009. Special Section devoted to selected best papers presented at ALENEX'08. To appear.
(Cited on page 70.)

[BDDW09] Reinhard Bauer, Gianlorenzo D'Angelo, Daniel Delling, and Dorothea Wagner. The Shortcut Problem – Complexity and Approximation. In *Proceedings of the 35th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'09)*, volume 5404 of *Lecture Notes in Computer Science*, pages 105–116. Springer, January 2009.
(Cited on page 42.)

[BDS+08] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 303–318. Springer, June 2008.
(Cited on page 70.)

[BDS+09] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. Invited to a special issue of the ACM Journal of Experimental Algorithmics devoted to the best papers of WEA 2008, 2009.
(Cited on page 70.)

[BDSV09] Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-Dependent Contraction Hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*, pages 97–105. SIAM, April 2009.
(Cited on pages 8, 63, 110, and 124.)

[BDW07a] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. In Christian Liebchen, Ravindra K. Ahuja, and Juan A. Mesa, editors, *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, pages 209–225. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
(Cited on page 70.)

[BDW07b] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Shortest-Path Indices: Establishing a Methodology for Shortest-Path Problems. Technical Report 2007-14, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), 2007.
(Cited on pages 38 and 123.)

[BDW09] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. *Networks*, 2009. Accepted for publication, to appear.
(Cited on page 70.)

[BFM+07] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In Transit to Constant Shortest-Path Queries in Road Networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 46–59. SIAM, 2007.
(Cited on pages 6, 62, 63, and 64.)

[BFM09] Holger Bast, Stefan Funke, and Domagoj Matijevic. TRANSIT Ultrafast Shortest-Path Queries with Linear-Time Preprocessing. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *Shortest Paths: Ninth DIMACS Implementation Challenge*, DIMACS Book. American Mathematical Society, 2009. Accepted for publication, to appear.
(Cited on page 6.)

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.
(Cited on page 6.)

[BGS08] Veit Batz, Robert Geisberger, and Peter Sanders. Time Dependent Contraction Hierarchies - Basic Algorithmic Ideas. Technical report, ITI Sanders, Faculty of Informatics, Universität Karlsruhe (TH), 2008.
(Cited on page 8.)

[BJ04] Gerth Brodal and Riko Jacob. Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. In *Proceedings of ATMOS Workshop 2003*, pages 3–15, 2004.
(Cited on page 16.)

[Bra01] Ulrik Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.
(Cited on page 42.)

[CH66] K. Cooke and E. Halsey. The Shortest Route Through a Network with Time-Dependent Intermodal Transit Times. *Journal of Mathematical Analysis and Applications*, (14):493–498, 1966.
(Cited on pages 7 and 72.)

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.
(Cited on pages 19, 141, and 144.)

[Dan62] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1962.
(Cited on page 4.)

[Dea99] Brian C. Dean. Continuous-Time Dynamic Shortest Path Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.
(Cited on page 72.)

[Del08] Daniel Delling. Time-Dependent SHARC-Routing. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 332–343. Springer, September 2008. Best Student Paper Award - ESA Track B.
(Cited on page 110.)

[Del09] Daniel Delling. Time-Dependent SHARC-Routing. *Algorithmica*, 2009. Special section devoted to selected best papers of ESA'08. to appear.
(Cited on page 110.)

[DGJ06] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.
(Cited on pages 3, 47, 116, and 141.)

[DHM+09] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. High-Performance Multi-Level Routing. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *Shortest Paths: Ninth DIMACS Implementation Challenge*, DIMACS Book. American Mathematical Society, 2009. Accepted for publication, to appear.
(Cited on pages 6, 63, and 64.)

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
(Cited on pages 1, 3, and 19.)

[DMS08] Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee. Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 347–361. Springer, June 2008.
(Cited on pages 8, 121, and 124.)

[DN08] Daniel Delling and Giacomo Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of *Lecture Notes in Computer Science*, pages 813–824. Springer, December 2008.
(Cited on page 110.)

[DN09] Daniel Delling and Giacomo Nannicini. Core Routing on Dynamic Time-Dependent Road Networks. Journal version of ISAAC'08, 2009.
(Cited on page 110.)

[DPW08] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Engineering Time-Expanded Graphs for Faster Timetable Information. In *Proceedings of the 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08)*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, September 2008.
(Cited on pages 7 and 18.)

[DSSW09a] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, Lecture Notes in Computer Science. Springer, 2009. To appear.
(Cited on pages 3 and 11.)

[DSSW09b] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Highway Hierarchies Star. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *Shortest Paths: Ninth DIMACS Implementation Challenge*, DIMACS Book. American Mathematical Society, 2009. Accepted for publication, to appear.
(Cited on pages 7, 33, 38, 64, and 66.)

[DW07] Daniel Delling and Dorothea Wagner. Landmark-Based Routing in Dynamic Graphs. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer, June 2007.
(Cited on page 70.)

[DW09] Daniel Delling and Dorothea Wagner. Pareto Paths with SHARC. In Jan Vahrenhold, editor, *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, Lecture Notes in Computer Science. Springer, June 2009. Accepted for publication, to appear.
(Cited on page 121.)

[Fli04] Ingrid C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation.* PhD thesis, Technische Universiteit Eindhoven, 2004.
(Cited on page 8.)

[FMSN00] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully Dynamic Algorithms for Maintaining Shortest Paths trees. *Journal of Algorithms*, 34(2), February 2000.
(Cited on page 30.)

[FR06] Jittat Fakcharoenphol and Satish Rao. Planar Graphs, Negative Weight Edges, Shortest Paths, and near Linear Time. *Journal of Computer and System Sciences*, 72(5):868–889, 2006.
(Cited on page 4.)

[GH05] Andrew V. Goldberg and Chris 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.
(Cited on pages 4, 7, 20, 21, and 32.)

[GKW06] Andrew V. Goldberg, Haim Kaplan, and Renato 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.
(Cited on pages 5, 7, and 22.)

[GKW07] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Better Landmarks Within Reach. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 38–51. Springer, June 2007.
(Cited on pages 7, 33, 34, 45, 61, 64, 65, 66, and 75.)

[GMS07] Thorsten Gunkel, Matthias Müller–Hannemann, and Mathias Schnee. Improved Search for Night Train Connections. In Christian Liebchen, Ravindra K. Ahuja, and Juan A. Mesa, editors, *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, pages 243–258. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
(Cited on page 8.)

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.
(Cited on pages 5, 27, 62, 64, and 65.)

[Gut04] Ronald J. Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 100–111. SIAM, 2004.
(Cited on page 5.)

[GW05] Andrew V. Goldberg and Renato 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.
(Cited on pages 7, 20, 21, 22, 49, 52, and 63.)

[Han79] P. Hansen. Bricriteria Path Problems. In Günter Fandel and T. Gal, editors, *Multiple Criteria Decision Making – Theory and Application –*, pages 109–127. Springer, 1979.
(Cited on page 8.)

[HKMS09] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *Shortest Paths: Ninth DIMACS Implementation Challenge*, DIMACS Book. American Mathematical Society, 2009. To appear.
(Cited on pages 6, 23, 24, 45, 60, 64, and 66.)

[HNR68] Peter E. Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
(Cited on pages 4 and 20.)

[Hol08] Martin Holzer. *Engineering Planar-Separator and Shortest-Path Algorithms*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008.
(Cited on page 5.)

[HSW04] Martin Holzer, Frank Schulz, and Thomas Willhalm. Combining Speed-up Techniques for Shortest-Path Computations. In *Proceedings of the 3rd Workshop on Experimental Algorithms (WEA'04)*, volume 3059 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2004.
(Cited on page 7.)

[HSW06] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*. SIAM, 2006.
(Cited on page 5.)

[HSW08] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13:2.5:1–2.5:26, December 2008.
(Cited on page 5.)

[HSWW06] Martin Holzer, Frank Schulz, Dorothea Wagner, and Thomas Willhalm. Combining Speed-up Techniques for Shortest-Path Computations. *ACM Journal of Experimental Algorithmics*, 10, 2006.
(Cited on page 7.)

[IHI+94] T. Ikeda, Min-Yao Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by AI search techniques. In *Proceedings of the Vehicle Navigation and Information*

Systems Conference (VNSI'94), pages 291–296. ACM Press, 1994.
(Cited on page 21.)

[Kar07] George Karypis. METIS - Family of Multilevel Partitioning Algorithms, 2007.
(Cited on page 23.)

[Ker04] Boris S. Kerner. *The Physics of Traffic*. Springer, 2004.
(Cited on pages 91 and 101.)

[Kle05] Philip N. Klein. Multiple-Source Shortest Paths in Planar Graphs. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 146–155, 2005.
(Cited on page 4.)

[KMS05] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, Lecture Notes in Computer Science, pages 126–138. Springer, 2005.
(Cited on pages 6 and 23.)

[KS93] David E. Kaufman and Robert L. Smith. Fastest Paths in Time-Dependent Networks for Intelligent Vehicle-Highway Systems Application. *Journal of Intelligent Transportation Systems*, 1(1):1–11, 1993.
(Cited on page 7.)

[Lau04] Ulrich 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.
(Cited on pages 6 and 23.)

[Lau09] Ulrich Lauther. An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Roadnetworks with Precalculated Edge-Flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *Shortest Paths: Ninth DIMACS Implementation Challenge*, DIMACS Book. American Mathematical Society, 2009. To appear.
(Cited on page 23.)

[Mar84] Ernesto Queiros Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 26(3):236–245, 1984.
(Cited on page 8.)

[Mey01] Ulrich Meyer. Single-Source Shortest-Paths on Arbitrary Directed Graphs in Linear Average-Case Time. In *Proceedings of the 12th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'01)*, pages 797–806, 2001.
(Cited on page 4.)

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA, A platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
(Cited on page 141.)

[Möh99] Rolf H. Möhring. Verteilte Verbindungssuche im öffentlichen Personenverkehr – Graphentheoretische Modelle und Algorithmen. In Patrick Horster, editor, *Angewandte Mathematik insbesondere Informatik, Beispiele erfolgreicher Wege zwischen Mathematik und Informatik*, pages 192–220. Vieweg, 1999.
(Cited on page 8.)

[MS04] Burkhard Monien and Stefan 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.
(Cited on page 23.)

[MS07] Matthias Müller–Hannemann and Mathias Schnee. Finding All Attractive Train Connections by Multi-Criteria Pareto Search. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*, pages 246–263. Springer, 2007.
(Cited on pages 8 and 121.)

[MSM06] Jens Maue, Peter Sanders, and Domagoj Matijevic. Goal Directed Shortest Path Queries Using Precomputed Cluster Distances. In *Proceedings of the 5th Workshop on Experimental Algorithms (WEA'06)*, volume 4007 of *Lecture Notes in Computer Science*, pages 316–328. Springer, 2006.
(Cited on page 7.)

[MSS+05] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speed Up Dijkstra's Algorithm. In *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, Lecture Notes in Computer Science, pages 189–202. Springer, 2005.
(Cited on pages 6 and 23.)

[MSS+06] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11:2.8, 2006.
(Cited on pages 6, 23, and 25.)

[MSWZ07] Matthias Müller–Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable Information: Models and Algorithms. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 2007.
(Cited on page 15.)

[Mül05] Kirill Müller. Berechnung kürzester Pfade unter Beachtung von Abbiegeverboten, 2005. Student Research Project.
(Cited on page 15.)

[Mül06] Kirill Müller. Design and Implementation of an Efficient Hierarchical Speedup Technique for Computation of Exact Shortest Paths in Graphs. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, June 2006.
(Cited on page 6.)

[MW01] Matthias Müller–Hannemann and Karsten Weihe. Pareto Shortest Paths is Often Feasible in Practice. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*, volume 2141 of *Lecture Notes in Computer Science*, pages 185–197. Springer, 2001.
(Cited on page 8.)

[MZ07] Laurent Flindt Muller and Martin Zachariasen. Fast and Compact Oracles for Approximate Distances in Planar Graphs. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'07)*, volume 4698 of *Lecture Notes in Computer Science*, pages 657–668. Springer, 2007.
(Cited on page 4.)

[NDLS08] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* Search for Time-Dependent Fast Paths. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 334–346. Springer, June 2008.
(Cited on page 110.)

[NDLS09] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* Search on Time-Dependent Road Networks. Journal version of WEA'08, 2009.
(Cited on page 110.)

[NST00] Paolo Narváez, Kai-Yeung Siu, and Hong-Yi Tzeng. New Dynamic Algorithms for Shortest Path Tree Computation. *IEEE/ACM Transactions on Networking*, 8(6):734–746, 2000.
(Cited on page 30.)

[OR90] Ariel Orda and Raphael Rom. Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. *Journal of the ACM*, 37(3):607–625, 1990.
(Cited on page 7.)

[Pel07] Francois Pellegrini. SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package, 2007.
(Cited on pages 23, 35, 59, and 86.)

[PSWZ04a] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Experimental Comparison of Shortest Path Approaches for Timetable Information. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 88–99. SIAM, 2004.
(Cited on pages 8 and 18.)

[PSWZ04b] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Towards Realistic Modeling of Time-Table Information through the Time-Dependent Approach. In *Proceedings of ATMOS Workshop 2003*, pages 85–103, 2004.
(Cited on pages 8 and 16.)

[PSWZ07] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12:Article 2.4, 2007.
(Cited on pages 8 and 107.)

[Sch05] Frank Schulz. *Timetable Information and Shortest Paths*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2005.
(Cited on pages 8 and 15.)

[Sch06] Heiko Schilling. *Route Assignment Problems in Large Networks*. PhD thesis, Technische Universität Berlin, 2006.
(Cited on page 6.)

[Sch08] Dominik Schultes. *Route Planning in Road Networks*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, February 2008.
(Cited on pages 33, 44, 64, 66, 70, and 141.)

[SLL02] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.
(Cited on page 141.)

[SS05] Peter Sanders and Dominik 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.
(Cited on pages 5, 48, and 91.)

[SS06] Peter Sanders and Dominik 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.
(Cited on pages 5 and 6.)

[SS07] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 66–79. Springer, June 2007.
(Cited on pages 5, 27, and 71.)

[SS09] Peter Sanders and Dominik Schultes. Robust, Almost Constant Time Shortest-Path Queries in Road Networks. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *Shortest Paths: Ninth DIMACS Implementation Challenge*, DIMACS Book. American Mathematical Society, 2009. Accepted for publication, to appear.
(Cited on page 6.)

[SV86] Robert Sedgewick and Jeffrey S. Vitter. Shortest Paths in Euclidean Graphs. *Algorithmica*, 1(1):31–48, 1986.
(Cited on page 20.)

[SWW99] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. In *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE'99)*, volume 1668 of *Lecture Notes in Computer Science*, pages 110–123. Springer, 1999.
(Cited on pages 3, 4, 5, 6, 7, 18, and 26.)

[SWW00] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5, 2000.
(Cited on pages 4 and 7.)

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002.
(Cited on pages 5 and 16.)

[Tea04] R Development Team. R: A Language and Environment for Statistical Computing, 2004.
(Cited on page 48.)

[The95] Dirk Theune. *Robuste und effiziente Methoden zur Lsung von Wegproblemen.* PhD thesis, 1995.
(Cited on page 8.)

[Tho01] Mikkel Thorup. Compact Oracles for Reachability and Approximate Distances in Planar Digraphs. In *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science (FOCS'01)*, pages 242–251. IEEE Computer Society Press, 2001.
(Cited on page 4.)

[Tho03] Mikkel Thorup. Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem. In *Proceedings of the 35th Annual ACM Symposium on the Theory of Computing (STOC'03)*, pages 149–158, June 2003.
(Cited on page 4.)

[Vol08] Lars Volker. Route Planning in Road Networks with Turn Costs, 2008. Student Research Project.
(Cited on page 15.)

[War87] Arthur Warburton. Approximation of Pareto Optima in Multiple-Objective Shortest-Path Problems. *Operations Research*, 35(1):70–79, 1987.
(Cited on page 8.)

[Wil05] Thomas Willhalm. *Engineering Shortest Paths and Layout Algorithms for Large Graphs.* PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2005.
(Cited on page 6.)

[WW03] Dorothea Wagner and Thomas Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03)*, volume 2832 of *Lecture Notes in Computer Science*, pages 776–787. Springer, 2003.
(Cited on page 6.)

[WW05] Dorothea Wagner and Thomas Willhalm. Drawing Graphs to Speed Up Shortest-Path Computations. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 15–24. SIAM, 2005.
(Cited on page 20.)

[WWZ05] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Geometric Containers for Efficient Shortest-Path Computation. *ACM Journal of Experimental Algorithmics*, 10:1.3, 2005.
(Cited on page 7.)

List of Figures

List of Tables

Appendix A

Implementation Details

When we started our work on route planning, we had to make a fundamental implementation choice. Do we want to use external libraries like LEDA [MN99] or BOOST [SLL02]? It turned out that those libraries either yield a tremendous overhead in space consumption or tend to be slower than tailored implementations [DGJ06]. Moreover, for efficient route planning algorithms, only basic datastructures are needed. More precisely, we need fast graph datastructures, a tailored datastructure for maintaining a multi-level partition, and an efficient priority queue. For the latter, Dominik Schultes provided us with his implementation of a binary heap [Sch08]. In the following, we describe our graph and multi-level partition datastructures in more detail. Our implementation is completely written in C++ using solely the STL at some points. Since virtual functions yield performance penalties, we use templates for providing a similar functionality as virtuality but *without* the known drawback.

A.1 Graphs

The most fundamental datastructure of our work is a graph. Since we assign different length functions for our different scenarios, we need tailored graphs for each scenario. While during query times, the graph is static in the sense that only edge weights may be updated but the topology stays untouched, the concept of contraction adds and removes nodes and edges from the graph. Hence, we also need a dynamic datastructure. Finally, we need to invest additional effort if we want to route efficiently in a bidirectional manner.

A.1.1 Static Graph

First, we focus on static graphs, i.e., the topology does not change after the graph has been constructed. In general, our graph datastructure is based on an *adjacency array representation* [CLRS01]. However, depending on the scenario and approach, small changes have to be incorporated.

Time-Independent Graphs. Our most prominent technique, SHARC, is a unidirectional technique. This allows a very simple datastructure. We use two arrays of structs, one representing nodes, the other edges. Enumeration is started at zero. The edge entries are ordered by their source nodes; thus, all outgoing edges of a node are stored in succession. Each node stores the index to its first outgoing edge, providing an easy access to them. A dummy node is also saved at the end of the node-array to provide a pointer to the first invalid element of the edge-array. Edges store their *weight* and their *targetNode*. Figure A.1 gives a small example.

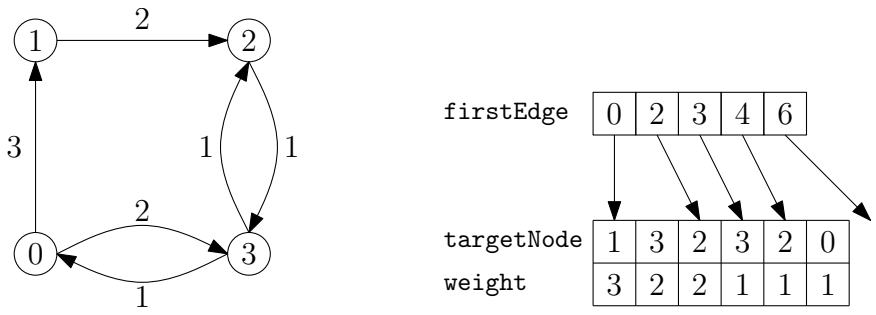


Figure A.1: Adjacency representation. The figure on right shows the representation of the graph from the left as adjacency representation.

Since different speed-up techniques use different additional data, the entries of the arrays are implemented by template structs. The basic data structures for this purpose are *basicNode* and *basicEdge*. If further information is needed at a node or at an edge, it can directly be added by extending the respective template. However, adding too much data to the structs has a negative impact on the performance. Smaller entries can be stored and retrieved faster. They also profit more from caching effects since more entries fit into the CPU cache.

Bidirectional Techniques. The above described representation has the disadvantage that no easy access to the incoming edges of a node exists. Since iterations over all incoming edges occur frequently when performing a *bidirectional* Dijkstra search, this shortcoming has to be remedied. Therefore, each edge is stored twice: Once at its tail and at its head. Additional Boolean flags indicate whether an edge is incoming or outgoing with respect to its target node. A small form of edge compression is used for undirected edges that would otherwise have to be stored four times (twice at both nodes, as incoming and as outgoing edge): Both directional flags are set, and the edge is only stored once at each node. This is extremely useful in road networks as here, most edges are undirected (cf. Section 2.4). Figure A.2 gives an example.

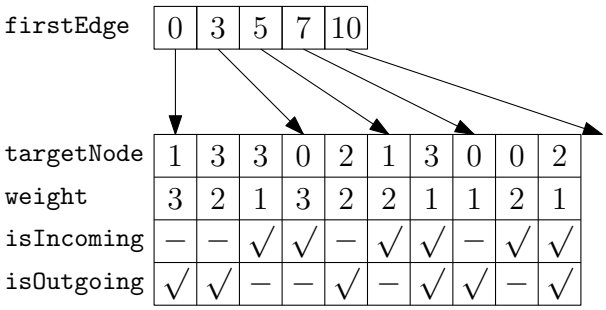


Figure A.2: Adjacency representation of the graph from Figure A.1. Incoming edges can now be efficiently accessed to as well.

Time-Dependent Graphs. The main difference to time-independent graphs is that more data is stored at edges. More precisely, a number of interpolation points is stored to each edge depicting the travel time at different departure times. Since the number of such

points is not the same for each edge we introduce a third layer storing *all* interpolation points of the graph. Each edge stores an additional pointer to the first interpolation point in the third layer. For each edge, the interpolation points are sorted by their time value. Note that we have to introduce a dummy edge for iteration over the points of the last edges. Figure A.3 gives an example. For accessing the correct interpolation points for a specific departure time τ, we access the point $p = \tau/\Pi \cdot |P(e)| + \texttt{firstPoint}(e)$ where $|P(e)|$ denotes the number of points assigned to edge e. In most cases, this access point is close to the one we seek. By linear search we finally retrieve the correct points.

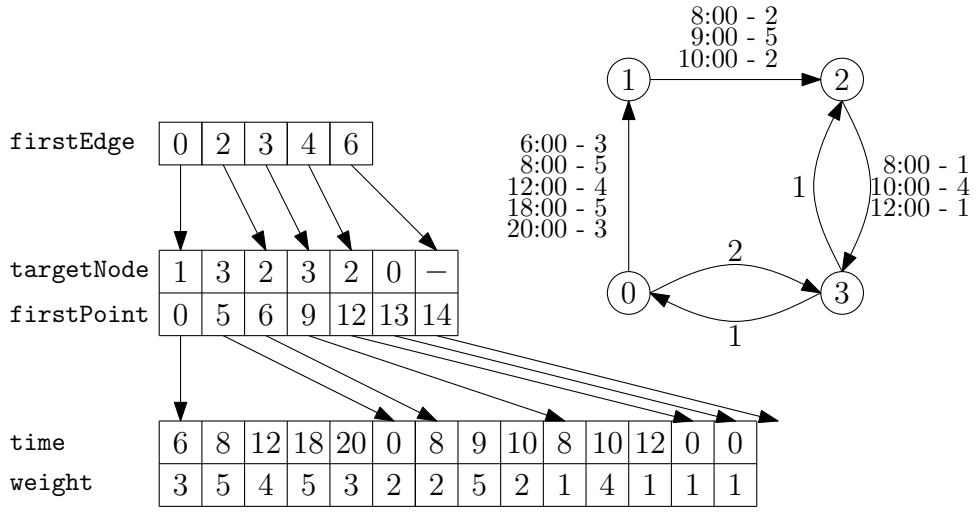


Figure A.3: Adjacency representation of a time-dependent graph. The right hand figure shows the representation of the graph including three time-dependent edges. The resulting datastructure is shown by the left hand figure.

Bidirectional Search. Some techniques introduced in this thesis rely on bidirectional search in time-dependent graphs. A straightforward adaption of the ideas used in time-independent scenarios would yield too high a space overhead, especially in completely directed graphs. Hence, we store incoming edges as *time-independent* edges separately in an array. Figure A.4 gives an example.

Multi-Criteria Graphs. Unlike for time-dependent graphs, modifications to our time-independent graph datastruture are only little. Since the number of weights assigned to one edge is fixed, we do not need a third layer like for time-dependent graphs. Instead, we simply store all edge weights directly in the corresponding edge array.

A.1.2 Dynamic Graph

During query times, the only updates we allow are based on edge weights which we can handle with the above introduced data structures: In the time-independent case, we can simply update the corresponding edge weight, while for the time-dependent case we need to update the breakpoint. Hence, it sufficient to settle for adjacency arrays during query times. During preprocessing however, we need to remove nodes and edges during contraction. Moreover, we add shortcuts to preserve correctness. Removing nodes and edges

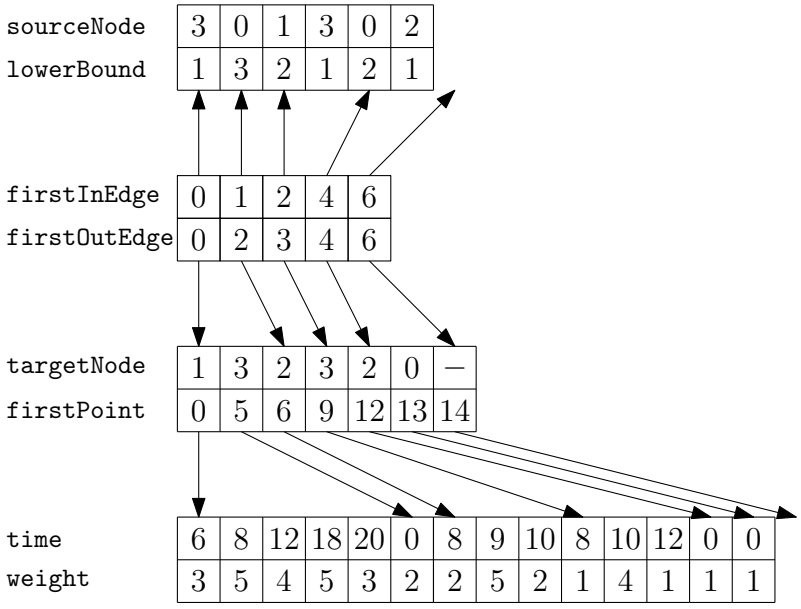


Figure A.4: Adjacency representation of a time-dependent graph including incoming edges. The graph represented is the one from Fig. A.3.

is easy. We introduce a Boolean for marking nodes and edges as valid. If we remove an node or edge, we simply mark the corresponding entry as invalid. Adding shortcuts however, cannot be handled easily in an adjacency array. Hence, we use a temporary dynamic datastructure based on *adjacency lists* [CLRS01]. Each node maintains a list of temporary edges. At the end of a contraction step, we rebuild the graph from still valid nodes and edges from the original graph and from the temporary edges.

A.2 Multi-Level Partition

During a SHARC query, we need to access the cell number of a node on a certain level very efficiently. In fact, this operation is executed whenever we settle a node. So, access should be as fast as possible. We store the cell numbers of all levels in own integer. The cell number on the lowest level is stored in the lowest bits, the number on the highest level in the highest bits. An example if given in Fig. A.5. Then, the cell number can be accessed by one bitshift and an additional AND operation.

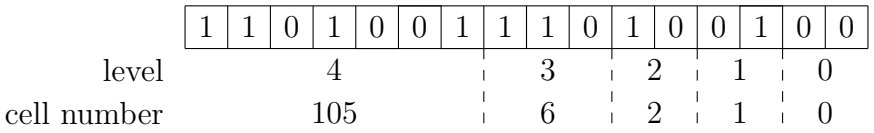


Figure A.5: Example for storing a multi-level partition with 4 cells on the lower 3 levels, 8 cells on the fourth level, and 108 cells on the topmost level.

Appendix B

List of Publications

Book Chapters:

- **Engineering Route Planning Algorithms.** In: *Algorithmics of Large and Complex Networks*, Lecture Notes in Computer Science. Springer, 2009. Accepted for publication, to appear. Joint work with Peter Sanders, Dominik Schultes, and Dorothea Wagner.

- **High-Performance Multi-Level Routing.** In: *Shortest Paths: Ninth DIMACS Implementation Challenge,* DIMACS Book. American Mathematical Society, 2008. Accepted for publication, to appear. Joint work with Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner.

- **Highway Hierarchies Star.** In: *Shortest Paths: Ninth DIMACS Implementation Challenge,* DIMACS Book. American Mathematical Society, 2008. Accepted for publication, to appear. Joint work with Peter Sanders, Dominik Schultes, and Dorothea Wagner.

Journals (accepted):

- **Time-Dependent SHARC-Routing.** *Algorithmica,* 2009. Special section devoted to selected best papers presented at ESA'08. Accepted for publication, to appear.

- **SHARC: Fast and Robust Unidirectional Routing.** *ACM Journal of Experimental Algorithmics,* 2009. Special section devoted to selected best papers presented at ALENEX'08. Accepted for publication, to appear. Joint work with Reinhard Bauer.

- **Experimental Study on Speed-Up Techniques for Timetable Information Systems.** *Networks,* 2009. Accepted for publication, to appear. Joint work with Reinhard Bauer and Dorothea Wagner.

- **On Modularity Clustering.** *IEEE Transactions on Knowledge and Data Engineering,* 20(2):172-188, February 2008. Joint work with Ulrik Brandes, Marco Gaertler, Robert Görke, Martin Höfer, Zoran Nikoloski, and Dorothea Wagner.

Journals (submitted):

- **Core Routing on Dynamic Time-Dependent Road Networks.** Journal version of ISAAC'08. Joint work with Giacomo Nannicini.

- **Engineering Time-Expanded Graphs for Faster Timetable Information.** Journal version of ATMOS'08. Joint work with Thomas Pajor and Dorothea Wagner.

- **Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm.** Invited submission to a special issue of the *ACM Journal of Experimental Algorithmics* devoted to the best papers of WEA'08. Joint work with Reinhard Bauer, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner.

- **Bidirectional A* Search on Time-Dependent Road Networks.** Journal version of WEA'08. Joint work with Giacomo Nannicini, Leo Liberti, and Dominik Schultes.

Conference Articles:

- **ORCA Reduction and ContrAction Graph Clustering.** In: *Proceedings of the 5th International Conference on Algorithmic Aspects in Information and Management (AAIM'09)*, Lecture Notes in Computer Science. Springer, June 2009. to appear. Joint work with Robert Grke, Christian Schulz, and Dorothea Wagner.

- **Pareto Paths with SHARC.** In: *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*. Springer, June 2009. Accepted for publication, to appear. Joint work with Dorothea Wagner.

- **The Shortcut Problem - Complexity and Approximation.** In: *Proceedings of the 35th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'09)*, Lecture Notes in Computer Science. Springer, 2009. Accepted for publication, to appear. Joint work with Reinhard Bauer, Gianlorenzo D'Angelo, and Dorothea Wagner.

- **Time-Dependent Contraction Hierarchies.** In: *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*. SIAM, 2009. Accepted for publication, to appear. Joint work with Veit Batz, Peter Sanders, and Christian Vetter.

- **Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks.** In: *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of Lecture Notes in Computer Science, pages 813-824. Springer, December 2008. Joint work with Giacomo Nannicini.

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

- **Engineering Time-Expanded Graphs for Faster Timetable Information.** In: *Proceedings of the 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08)*, Dagstuhl Seminar Proceedings. September 2008. Joint work with Thomas Pajor and Dorothea Wagner.

- **Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm.** In: *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of Lecture Notes in Computer Science, pages 303-318. Springer, June 2008. Joint work with Reinhard Bauer, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner.

- **Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks.** In: *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of Lecture Notes in Computer Science, pages 319-333. Springer, June 2008. Joint work with Robert Geisberger, Peter Sanders, and Dominik Schultes.

- **Bidirectional A* Search for Time-Dependent Fast Paths.** In: *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of Lecture Notes in Computer Science, pages 334-346. Springer, June 2008. Joint work with Giacomo Nannicini, Leo Liberti, and Dominik Schultes.

- **SHARC: Fast and Robust Unidirectional Routing.** In: *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13-26. SIAM, 2008. Joint work with Reinhard Bauer.

- **Engineering Comparators for Graph Clusterings.** In: *Proceedings of the 4rd International Conference on Algorithmic Aspects in Information and Management (AAIM'08)*, volume 5034 of Lecture Notes in Computer Science, pages 131-142. Springer, 2008. Joint work with Marco Gaertler, Robert Görke, and Dorothea Wagner.

- **On Finding Graph Clusterings with Maximum Modularity.** In: *Proceedings of the 33rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG'07)*, volume 4769 of Lecture Notes in Computer Science, pages 121-132. Springer, October 2007. Joint work with Ulrik Brandes, Martin Höfer, Marco Gaertler, Robert Görke, Zoran Nikoloski, and Dorothea Wagner.

- **Landmark-Based Routing in Dynamic Graphs.** In: *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of Lecture Notes in Computer Science, pages 52-65. Springer, June 2007. Joint work with Dorothea Wagner.

- **Experimental Study on Speed-Up Techniques for Timetable Information Systems.** In: *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, pages 209-225. Dagstuhl Seminar Proceedings. 2007. Joint work with Reinhard Bauer and Dorothea Wagner.

- **Generating Significant Graph Clusterings.** In: *Proceedings of the European Conference of Complex Systems (ECCS'06)*, September 2006. Joint work with Marco Gaertler and Dorothea Wagner.

Appendix C

Curriculum Vitae

Daniel Delling

born June 7, 1979 in Hamburg, Germany

Current status

since 03/06	Research and teaching assistant, chair Algorithmics I Universität Fridericiana zu Karlsruhe (TH)

Education

03/06–02/09	PhD student in Informatics, Universitat Fridericiana zu Karlsruhe (TH) Advisors: Prof. Dr. D. Wagner, Prof. Dr. R. Möhring
10/00–01/06	Diplom (German M.Sc.) in Informatics Universitat Fridericiana zu Karlsruhe (TH) Thesis: Analyse und Evaluierung von Vergleichsmaßen für Graphclusterungen.
08/89–07/98	Abitur (German A-level equivalents) Gynmasium Oberalster, Hamburg

Studies abroad

10/07	Visiting academic at the Department of Computer Science University of Patras, Greece Advisor: Prof. Dr. C. Zaroliagis

Teaching Activites

Winter 07/08	Implementation course “Development of a fast dynamic graph datatructure”
Winter 06/07	Teaching assistant “algorithms and datastructures”

Appendix D

Deutsche Zusammenfassung

Optimale Routen in Verkehrsnetzen zu bestimmen ist ein alltägliches Problem. Wurden früher Reiserouten mit Hilfe von Karten am Küchentisch geplant, ist heute die computergestützte Routenplanung in weiten Teilen der Bevölkerung etabliert: Die beste Eisenbahnverbindung ermittelt man im Internet, für Routenplanung in Straßennetzen benutzt man häufig mobile Navigationsgeräte.

Ein Ansatz, um die besten Verbindungen in solchen Netzen computergestützt zu finden, stammt aus der Graphentheorie. Man modelliert das Netzwerk als Graphen und berechnet darin einen kürzesten Weg, eine mögliche Route. Legt man Reisezeiten als Metrik zu Grunde, ist die so berechnete Route die beweisbar schnellste Verbindung. Dijkstra's Algorithmus aus dem Jahre 1959 löst dieses Problem zwar beweisbar optimal, allerdings sind Verkehrsnetze so groß (das Straßennetzwerk von West- und Mittel-Europa besteht aus ca. 45 Millionen Abschnitten), dass der klassische Ansatz von Dijsktra zu lange für eine Anfrage braucht. Aus diesem Grund ist die Entwicklung von *Beschleunigungstechniken* für Dijkstra's Algorithmus Gegenstand aktueller Forschung. Dabei handelt es sich um zweistufige Verfahren, die in einem Vorverarbeitungsschritt das Netzwerk mit Zusatzinformationen anreichern, um anschließend die Berechnung von kürzesten Wegen zu beschleunigen.

In den letzten Jahren konzentrierte man sich auf die Entwicklung von Beschleunigungstechniken in *statischen* Straßennetzwerken. Mittlerweile können in solchen Netzwerken kürzeste Wege innerhalb von Mikrosekunden berechnet werden. Allerdings berücksichtigen alle bisherigen Arbeiten einen wichtigen Aspekt nicht. Die optimale Route hängt häufig von dem Abfahrtszeitpunkt ab. Beispielsweise lohnt es sich, Autobahnen während Stoßzeiten zu meiden. Solch ein Szenario kann mittels einem *zeitabhängigem* Netzwerk modelliert werden, die Reisezeit einer Verbindung hängt nun von der Abfahrtszeit ab. Es stellt sich heraus, dass die Integration des Zeitaspektes in Beschleunigungstechniken nicht kanonisch dem statischen Fall folgt. Problematisch ist beispielsweise die deutlich angestiegene Eingabegröße. Schwerwiegender ist allerdings die Tatsache, dass die bisher schnellsten Techniken auf bidirektionaler Suche basieren. Hierbei wird eine zweite Suche vom Ziel gestartet. Dieser Ansatz stellt sich in zeitabhängigen Szenarien allerdings als schwierig dar, da die Ankunftszeit vorab unbekannt ist. Außerdem werden Anfragen auch komplexer: Beispielsweise könnte ein Benutzer den Abfahrtszeitpunkt anfragen, der seine Reisezeit minimiert.

Ein weiterer nicht betrachteter Aspekt bei der Routenplanung ist, dass nicht immer die schnellste Route auch die attraktivste ist. Man ist durchaus bereit eine etwas längere Reisezeit in Kauf zu nehmen, wenn die Kosten der Reise deutlich geringer sind oder wenn die Reiseroute landschaftlich besonders reizvoll ist. Anstatt nur die Reisezeit zu minimieren bezieht man auch andere Metriken wie beispielsweise Kosten in die Bewertung von Routen

mit ein. Solch besseren Routen können mittels *multikriterieller* Optimierung berechnet werden. Man berechnet nun die *Pareto-optimalen* Wege zwischen zwei Punkten. Diese kennzeichnen sich dadurch, dass jeder Weg bezüglich mindestens einer Metrik besser als die anderen ist. Wie im zeitabhängigen Fall ist die Adaption der unikriteriellen Verfahren an dieses Szenario nicht trivial.

Ergebnisse

Die vorliegende Arbeit "Engineering and Augmenting Route Planning Algorithms" stellt neue, beweisbar korrekte Verfahren vor, die es ermöglichen oben genannte Problemstellungen effizient zu lösen. Dabei verfolge ich den Ansatz des *Algorithm Engineering*: Nicht nur der Entwurf und die theoretische Analyse ist von wesentlicher Bedeutung bei der Entwicklung von Algorithmen, sondern auch deren Implementierung und experimentelle Evaluation. Dieser Prozess kann als Kreislauf aufgefasst werden, bei dem die Experimente neue Impulse für den Entwurf eines Algorithmus liefern können. Demzufolge evaluiere ich alle in dieser Arbeit entwickelten Techniken ausgiebig experimentell mit Real-Welt Daten. Dabei liegt der Hauptaugenmerk auf Straßen- und Eisenbahnnetzen. Um die Robustheit der vorgestellten Verfahren gegenüber der Eingabe zu demonstrieren, werden zusätzlich auch Daten aus anderen Bereichen der Informatik als Eingabe genutzt. Die Ergebnisse der Arbeit sind im Detail:

Zeitunabhängiges Szenario. Zuerst untersuche ich bekannte Techniken auf ihre mögliche Adaption an die neuen Herausforderungen des Zeitaspektes. Dabei liegt ein Hauptaugenmerk darauf, inwieweit Techniken mit Änderungen im Netzwerk zurecht kommen. Ferner konzentriere ich mich auf die Entwicklung eines schnellen unidirektionalen Verfahrens, das genauso effizient wie bidirektionale Ansätze ist und dabei den Vorteil hat, dass die Adaption an komplexere Szenarien deutlich einfacher ist. Außerdem kombiniere ich bekannte Techniken miteinander, so dass die Arbeit die schnellsten Techniken zur Berechnung kürzester Wege in statischen zeitunabhängigen Netzwerken vorstellt.

Zeitabhängiges Szenario. Der innovativste Teil meiner Arbeit ist die Entwicklung von Beschleunigungstechniken für das zeitabhängige Szenario. Hierbei ergänze ich Techniken aus dem vorherigen Teil um den Zeitaspekt. Dabei sind die berechneten Routen weiterhin beweisbar optimal. Ich verfolge zwei Ansätze: Zum einen erweitere ich den schnellsten unidirektionalen Ansatz aus dem vorherigen Abschnitt. Zum anderen untersuche ich, inwieweit man eine Rückwärtssuche vom Ziel starten kann, um die Vorwärtssuche sinnvoll einzuschränken.

Multikriterielles Szenario. Durch Erweiterung des schnellsten unidirektionalen Verfahrens aus dem zeitunabhängigen Abschnitt können ebenfalls Pareto-optimale Routen effizient berechnet werden. Allerdings muss die Anzahl sinnvoller Wege, zu mindestens in Straßengraphen, beschränkt werden. Beispielsweise betrachte ich bezüglich Reisezeit deutlich längere Routen nur dann als sinnvoll, wenn die Reisekosten signifikant geringer sind.