Universität Karlsruhe (TH)

Systematic Combination of Speed-Up Techniques for exact Shortest-Path Queries

Dennis Schieferdecker

Diploma Thesis

in the Department of Informatics
at the University of Karlsruhe

Referee: *Prof. Dr. Dorothea Wagner*
(Institute for Theoretical Informatics)

Advisors: *Dipl. Inf. Daniel Delling*
Dipl. Math. Reinhard Bauer
Dominik Schultes, M. Sc.

January 31, 2008

"I hear and I forget,
I see and I remember,
I do and I understand."
(accredited to Confucius)

Contents

Introduction

Arriving from one's location at the desired destination has never been easier than it is today. Navigation systems, often integrated into a body as small as a mobile phone, provide the means to find shortest paths between two positions according to some metric like distance or travel times with virtually no effort. These devices usually still use heuristics to compensate for their lack of computational power and memory. Thus, they only provide very good but not necessarily exact results. This suboptimality might be sufficient for simple route planning tasks in a road network, but there are other applications, also applying shortest-path queries, that either need exact results, or for which the improvements provided by the use of heuristics are still not enough. For example, there are queries on timetable information systems, Internet route planners, and optimization problems in logistics, just to name a few. All of them have to operate on more complex graphs than a typical road network, and thus, are in need of either more computational power and memory, or better algorithms.

A well-known solution for finding exact shortest paths in a static graph is provided by Dijkstra in [Dij59] and subsequently named after him. Many newer techniques built on Dijkstra's algorithm and heuristically improve upon its performance while maintaining its correctness. They usually exchange a shorter runtime for additional required preprocessing time and larger space consumption. Applying the fastest of these techniques, *Transit Node Routing*, query times on static road networks can be lowered by a factor of up to one million. But there is still room for further improvements, especially since this speed-up comes at the cost a huge memory overhead. Combining several techniques and trying to exploit their respective characteristics is a logical next step. After some initial combinatorial studies in [SWW99, SWW00], Holzer et al. provided a systematic overview of the possible combinations of speed-up techniques in [HSW04, HSWW06]. They observed that combinations of two different approaches, *goal-direction* and *hierarchies*, are most promising. They are usually fast and also robust with regards to the applied graphs. Since then, many of the existing techniques have been improved and new ones have been devised. In the recent years, only some isolated combinations of techniques have been introduced, like REAL [GKW06a] and Highway Hierarchies* [DSSW06]. Now, this thesis revisits the idea of combining speed-up techniques and builds upon the results by Holzer et al. The recent developments in this field of research are accounted for and combinations of goal-directed and hierarchical techniques are systematically analyzed. In particular, recent bidirectional techniques for routing on static graphs are considered. Extensive experiments on multiple different types of graphs are performed for the introduced combinations of techniques and the obtained results are analyzed.

1.1 Related Work

In the field of shortest-path techniques, there is a large number of contributors. Therefore, only several recent works are highlighted here. The results mentioned below are based on the European road network with travel times and have been taken from [BD08], in general. Note that the techniques have been evaluated on slightly different machines; thus, a small deviation from the results presented in this thesis is inevitable.
The term *speed-up* refers to the differences in query times between the presented technique to Dijkstra's algorithm, and *overhead* denotes the additional memory required by the technique.

a) Goal-Directed Techniques:

ALT. Goldberg et al. revisited the A*-search in 2004 [GH04]. This technique superimposes a potential on the graph, yielding lower bounds for the distance from each node to the target, and thus, giving a good sense of goal-direction for the query. They showed that its performance can be improved significantly by using landmark-based lower bounds instead of Euclidean ones. The new technique was abbreviated with the term ALT (A*-search, Landmarks, Triangle inequality). With about 0:13 (h:min), it features a very short preprocessing time and yields speed-ups of about 30 but at the expense of an overhead of 60 bytes/node.

ArcFlags. The original idea goes back to Lauther in 1997 [Lau97] and 2004 [Lau04]. It has been improved later in [KMS05, MSS$^+$05, MSS$^+$06, HKMS06, Hil07], with special regards to the performance of the preprocessing and the memory usage. The graph is partitioned into several regions. Each arc has a flag for every region that denotes, whether the arc is part of a shortest path into that region. This information provides an excellent sense of direction during the query and allows for pruning of large parts of the search space. With 4 000, speed-ups are in the same general area as Highway Hierarchies and with 25 bytes/node, the overhead is even smaller but the preprocessing times of over 36:00 are much higher.

b) Hierarchical Techniques:

Reach. Gutman first defined the notion of a vertex reach in 2004 [Gut04]. Figuratively speaking, a node has a large reach if it is located near the middle of a long shortest path. This knowledge can be used during a query to prune the search and to reduce the search space. This approach was later improved by Goldberg et al. in [GKW06a] by providing a bidirectional version of the query that uses implicit lower bounds for pruning, and by adding shortcuts to the graph, reducing the vertex reaches. The improved Reach algorithm performs about 1 200 times faster than Dijkstra's algorithm with an overhead of 12 bytes/node but also needs 1:20 for its preprocessing.

Highway Hierarchies. Sanders and Schultes introduced Highway Hierarchies in 2005 [SS05]. This method is based on the observation that only a highway network, i. e. a very spare part of the original graph, needs to be searched outside of a small neighbourhood around the source and the target. This approach can be iterated to generate a hierarchy of highway networks. The technique features very short preprocessing times of only 0:13 with an overhead of 48 bytes/node and speed-ups of about 7 000.

Highway Node Routing. This approach was developed by Schultes and Sanders in 2007 [SS07]. It generalizes former multi-level methods like in [SWW00, SWZ02, HSW06]. These methods all require graph separators to construct a hierarchy of overlay graphs. Highway Node Routing only needs an arbitrary classification of nodes, which can be computed e. g. by Highway Hierarchies [SS05] or Contraction Hierarchies [Gei08]. A new graph with additional shortcuts is built in such a way that, after entering a certain hierarchy level, the query will never encounter an arc to a lower level. Thus, the search space is effectively reduced by entering each higher level. This approach yields speed-ups of 5 000, while it usually has little to no effective overhead. Furthermore, it can be easily adapted for the use with dynamic graphs.

Transit Node Routing. In 2007, Bast et al. presented a fundamentally new approach for finding shortest paths [BFSS07]. They noted that each shortest path from about the same general area to some distant location leaves this area via one of only a few so-called transit nodes. These are interconnected by a sparse network that is only relevant for long-distance travel. Thus, it is sufficient to precompute the distances between all possible sources and targets and their respective transit nodes as well as the distances between the transit nodes themselves. Finding a shortest path is then reduced to only a few table lookups. This method features huge speed-ups of up to 700 000 with a moderate preprocessing time of 3:00, but unfortunately, it also has huge space requirements with an overhead of 251 bytes/node. This large memory consumption can be reduced by introducing a hierarchy of transit nodes. But this leads to more table lookups, and thus, a longer query time.
Note that although Bast et al. were the first to explicitly formulate the central observations and concepts of Transit Node Routing, Müller principally developed this type of algorithms before in [Mül06]. He extended the separator-based multi-level method and used the already available separator nodes as transit nodes, even though he did not use the same nomenclature.

c) Combinations:

Early Contributions. Initial studies in [SWW00] combined a special kind of geometric containers [WWZ05], the separator-based multi-level method [SWZ02], with the A* search [HNR68] to speed-up a railway transport problem. Later, Holzer et al. systematically combined basic speed-up techniques in [HSW04, HSWW06]. In particular, the A* search, the bidirectional search

[Dan62], the separator-based multi-level method, and normal geometric containers have been studied. One of their key observations was that it is most promising to combine hierarchical and goal-directed techniques. However, since their initial publication in 2004, many powerful hierarchical speed-up techniques have been developed, goal-directed techniques have been improved, and huge data sets have been made available to the community.

REAL. In 2005, Goldberg et al. improved the original Reach technique by Gutman, as already mentioned above, and also integrated goal-direction in the form of the ALT algorithm. The new technique was labeled REAL (Reach and ALT) [GKW06a]. Since then, it has been improved several times, e. g. by the introduction of reach-aware landmarks, better utilization of caching effects, and improved algorithms for the reach computation [GKW06b, GKW07]. The REAL algorithm now achieves speed-ups of up to 4 000 with moderate preprocessing times of 2:21 and space requirements of 32 bytes/node.

Highway Hierarchies*. Inspired by the REAL algorithm, Delling et al. combined Highway Hierarchies with the ALT algorithm in 2006 [DSSW06]. They showed that these techniques have some useful synergies. Landmark nodes can be selected, and landmark distances can be stored more efficiently by considering only a higher level of the highway hierarchy. Also, the ALT algorithm provides an additional sense of direction for the query, so that the search space can be pruned appropriately. But unfortunately, this algorithm performs only slightly better than normal Highway Hierarchies due to its complex stopping condition, yielding a speed-up of 8 500 with similar preprocessing times and an increased overhead of 69 bytes/node.

SHARC. Bauer and Delling introduced a new variant of the ArcFlags algorithm in 2007, which they labeled SHARC Routing (Shortcuts and ArcFlags) [BD08]. It combines ArcFlags with contraction approaches taken from other hierarchical methods. Basically, the SHARC query is a normal multi-level ArcFlags query, generalized from the two-level ArcFlags algorithm presented in [MSS$^+$06]. The preprocessing itself incorporates ideas from hierarchical approaches. The graph is partitioned into regions on several levels. On each level, it is contracted by bypassing unimportant nodes, and then, ArcFlags are computed. Subsequently, the graph is well sparsificated by removing arcs for which all flags have been computed. With 3:12 the bidirectional variant of this technique has a preprocessing time comparable to the other methods presented above. Its speed-up of 48 500 is only surpassed by Transit Node Routing, which in turn has a much higher memory overhead than the 20 bytes/node needed by the SHARC query.

1.2 Overview

Outline of the contents of each chapter in this thesis:

Chapter 1. This chapter. At first, a motivation is given for the compilation of thesis at hand. Then, various related work is presented. Following this, a short overview of the contents of the thesis is given.

Chapter 2. Some information on the basics of this field of research is provided in this chapter. At first, fundamental graph terms and definitions are introduced. Then, Dijkstra's algorithm for exact shortest-path queries on static graphs is explained. Several previous speed-up techniques are described subsequently. Promising combinations of these techniques are pointed out at the end of the chapter and are noted for further analysis.

Chapter 3. Some basic and already existing combinations of speed-up techniques are discussed here. The AALT algorithm is introduced as a new and simple technique, combining two goal-directed approaches, ArcFlags and ALT. In addition, an outline is given on two recent techniques, REAL and Highway Hierarchies*. Both take the ALT algorithm and add it to a hierarchical query, Reach or, respectively, Highway Hierarchies.

Chapter 4. This chapter deals with the contraction of graphs. The principles of contracting a graph are outlined, and a basic query, Contracted Dijkstra, is described for routing on a graph that has been altered in this way. The main part of the chapter concentrates on adapting the ALT algorithm for these contracted graphs. The resulting CALT technique shows similarities to REAL and Highway Hierarchies*, described in the last chapter.

Chapter 5. The fifth chapter focuses on the application of ArcFlags to hierarchical techniques. The Partial ArcFlags approach is introduced that applies ArcFlags only to the higher levels of an existing hierarchy. Afterwards, two concrete implementations of this approach are presented. At first, a combination with the Reach algorithm, called Reach-aware ArcFlags, is discussed. Then, it is shown how ArcFlags can be added to the query of Highway Node Routing. The resulting technique is labeled Hierarchy-aware ArcFlags. The queries of both of these techniques resemble the query of the CALT algorithm.

Chapter 6. The experimental studies are elaborated on in this chapter. At first, the setup used for performing the studies is explained, followed by the description of the different types of graphs that have been analyzed. Then, the results that have been obtained for the different combinations of speed-up techniques are presented and discussed thoroughly. Also, an overview is given, comparing the introduced combinations to other recent techniques.

Conclusion. The final chapter summarizes the results gained by the experimental studies. The viability of the new speed-up techniques introduced in this thesis is also discussed, and an outlook on possible future work in this field of research is given.

Appendices. Additional information, like an overview of the applied data structures, several proofs and further results, are presented in the appendices. They are followed by the list of figure, the list of tables, and the bibliography. The German abstract and the acknowledgements conclude the thesis.

Fundamentals

This chapter introduces the basic concepts and algorithms of graph theory that are used throughout this thesis. At first, some basic graph terms are defined with more details being available in virtually any textbook, e. g. [CLRS01]. Then, Dijkstra's algorithm for finding exact shortest paths in a static graph is explained, followed by a description of several recent speed-up techniques for this task. At the end of the chapter, possible and existing combinations of the presented algorithms are highlighted.

2.1 Graph Definitions

Let $G = (V, E)$ be a graph with a set of vertices or nodes V and a set of edges or arcs with $E = \{(u, v) | u, v \in V\}$. If G is a directed graph, (u, v) and (v, u) represent different arcs. A *weight function* $w : E \rightarrow \mathbb{R}$ assigns values to each arc. A *graph layout* $L : N \rightarrow \mathbb{R}^2$ maps each node to a set of coordinates. Fig. 2.1 illustrates a small sample graph.
For a directed graph G the reverse graph $\overline{G} = (V, \overline{E})$ is defined by the same set of nodes and edges as the original graph, but all facing in the opposite direction: $\overline{E} = \{(v, u) | (u, v) \in E\}$. A sequence of connected edges in the form $e_1 = (n_1, n_2),\ e_2 = (n_2, n_3),\ \ldots,\ e_k = (n_k, n_{k+1})$ is

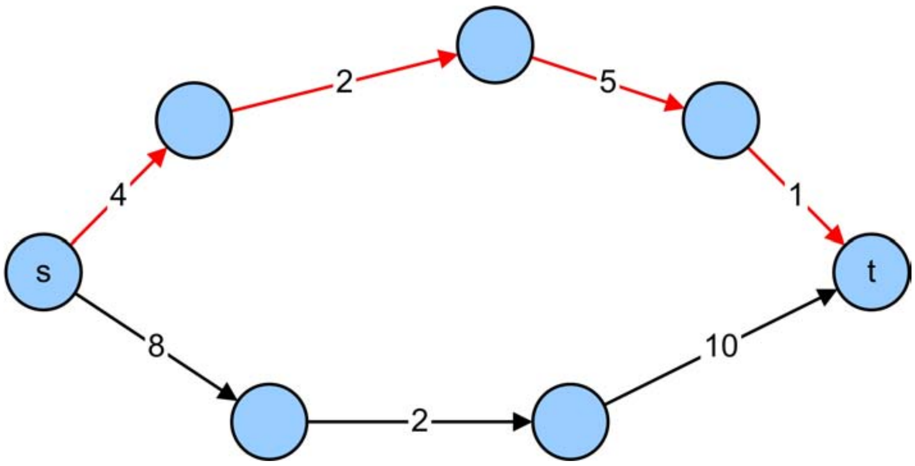


Figure 2.1: *Illustration of a small directed graph. It consists of seven nodes and seven edges. There are two paths from s to t, the upper one (red) has a length of 12, whereas the lower one (black) measures 20 units.*

called a path $P = \langle n_1, n_{k+1} \rangle$. The length of a path is defined as the sum of all edge weights $w(P) = \sum_{i=1}^{k} e_i$. Fig. 2.1 shows two paths from s to t, one of length 12 and one of length 20. The distance between two nodes $d(s,t)$ is defined to be the length of a shortest path between those nodes. In the sample graph, $d(s,t) = 12$ holds and the corresponding path is drawn in red.

2.2 Shortest-Path Search

One basic solution for finding shortest paths in a graph is called Dijkstra's algorithm [Dij59]. It computes the distances from a specified source to all other nodes in the graph.
For each node u the algorithm stores a tentative distance $d(u)$ from the source as an additional value. During a query all nodes are either unreached, reached, or settled. Unreached nodes have yet to be encountered by the algorithm; reached ones have already been visited and are managed in a priority queue Q according to their distance from the source. For settled nodes, the correct distance has already been computed.
The algorithm is initialized by setting all distances to infinity. The source node s is then assigned a distance of zero and inserted into the priority queue. In each step, the minimal element u of the queue is removed and becomes settled. If the condition $d(u) + w(u,v) \leq d(v)$ is true for one of its neighbours v, their tentative distance is updated to $d(v) = d(u) + w(u,v)$ and they are inserted into the priority queue. If they are already in the queue, their priority key just gets updated. The nodes v and their associated edges (u,v) are called touched.
The algorithm terminates if either the priority queue is empty, or the target node is about to become settled if one was provided by the initial query. The distances from the source to all settled nodes u can be accessed subsequently via $d(u)$.

Algorithm 1: Dijkstra's algorithm

```
1  begin
2      for $v \in V \setminus \{s\}$ do $d(v) = \infty$;
3      $u = \infty$;
4      $d(s) = 0$;
5      insert s in Q;
6      while $Q \neq 0$ and $u \neq t$ do
7          remove minimal element u from Q;
8          for $e = (u,v) \in E$ do
9              if $d(u) + w(u,v) \leq d(v)$ then
10                 $d(v) = d(u) + w(u,v)$;
11                 if $v \notin Q$ then insert v in Q;
12                 else update $d(v)$ in Q;
13 end
```

Note that Dijkstra's algorithm only produces correct results for graphs without negative edge weights. It has been shown in [Joh77] that a graph with negative edge weights can be converted appropriately in polynomial time if it does not contain negative loops, i. e. paths with the same source and target and a path length smaller than zero.
In general, the algorithm performs in $O(|V|^2)$. Depending on the graph and the data structure of the priority queue, this can be improved. For example on sparse graphs with $V \ll E$, runtimes are reduced to $O(|E| \log |V|)$ using a binary heap as priority queue, or $O(|E| + |V| \log |V|)$ using Fibonacci heaps. More on the runtimes of Dijkstra's algorithm can be found, e. g. in [CLRS01].

The algorithm can be easily modified to not only compute distances from the source to other nodes, but to also produce one associated shortest path tree. Every node needs to store its predecessor in the shortest path. At the start of the query, the predecessors of all nodes are undefined. If the tentative distance of a node v gets changed during the query, according to $d(v) = d(u) + w(u, v)$, its predecessor is set to u. After the search has ended, the shortest path tree can be computed in reverse, starting from the leaf nodees (see Fig. 2.2 for a shortest path).

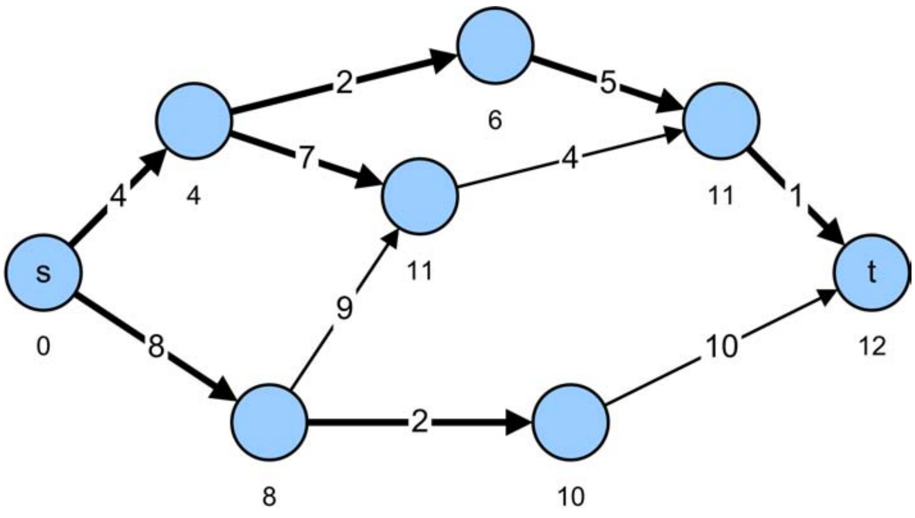


Figure 2.2: *A more complex sample graph is shown. A shortest path tree, rooted at s, is highlighted in bold. In addition, the distances from the source to each node are written below the respective nodes.*

All nodes that are reached by a shortest-path query form the so-called *search space*. In case of Dijkstra's algorithm, this can be illustrated by a sphere around the source that gets blown up as more and more nodes are visited. Before a node with a certain distance from the source is settled, all nodes closer to the source have to be settled. This is the worst-case scenario. Later, it is shown that minimizing the search space is one of two basic ideas that is applied by the speed-up techniques presented in this thesis. The other being the introduction of shortcuts so that fewer nodes have to be settled, overall, to arrive at a certain target.
The nodes currently in the priority queue form the *search front*. It can be illustrated by the surface of the sphere that represents the search space. The smaller it is, the smaller the impact of using a slow data structure for the priority queue becomes.

2.3 Speed-Up Techniques

Many techniques exist to enhance the performance of Dijkstra's algorithm. Here, five recent methods are described in more detail. They are chosen in particular, because combinations of them are studied later. The bidirectional search is also presented here, since it is a fundamental technique used by all of the other algorithms in this thesis.

2.3.1 Bidirectional Search

The bidirectional approach is probably the most evident idea to speed-up a shortest-path query from one source to one target [Dan62]. The query is performed by starting two Dijkstra searches, one from the source and one from the target. The latter is performed on the reverse graph and is labeled backward search in contrast to the forward search from the source. Both search directions are alternated according to some policy. Usually, they are either just alternated after each step or the direction with the smallest queue element is chosen.
The algorithm terminates if a node is about to become settled in one direction that has already been settled in the other direction. This node is called the *meeting node*. The resulting distance from the source to the target is the sum of the distance from the source to the meeting node in the forward direction and from the target to the meeting node in the backward direction.
This approach is applied by all other techniques presented in this thesis. So, strictly speaking, all of them already are a combination, of the bidirectional search and their respective approach.

Regarding the illustration of the search space in Fig. 2.3, it can be perceived why a bidirectional search is usually faster than a normal Dijkstra query. Instead of one huge sphere around the source, there are now two smaller spheres, one around the target and one around the source.

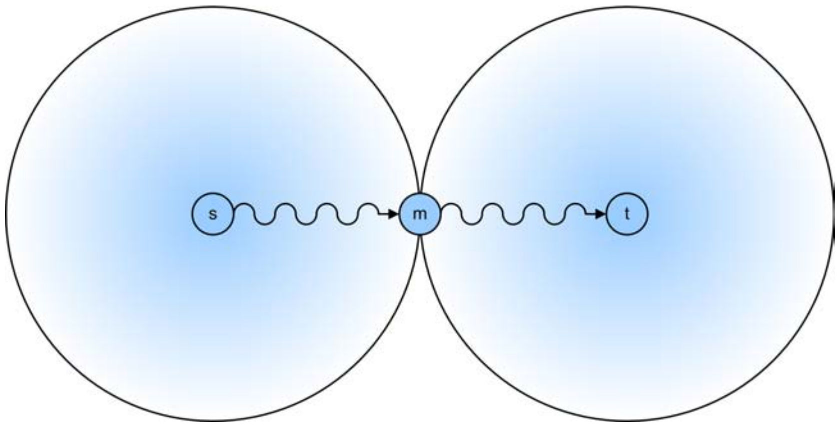


Figure 2.3: *Search space of a bidirectional Dijkstra query. The respective search spaces of both directions are illustrated by a sphere around the source and the target. Only the source, the target, and the meeting node are depicted for clarity. Note that the meeting node does not have to be located exactly in the middle.*

2.3.2 ALT

The ALT algorithm is based on the A* search, which originated from artificial intelligence studies in [HNR68]. The basic concepts of this search and two bidirectional variants are explained at first, before going into the particular details of the ALT algorithm.

A* search

The A* search is a modified Dijkstra's algorithm that applies additional information to improve the performance of the query. Let $G = (V, E)$ be an arbitrary graph and $\pi : V \rightarrow \mathbb{R}$ an arbitrary potential function on the graph-nodes. Then, a reduced weight function can be defined by $w_\pi(u, v) = w(u, v) + \pi(v) - \pi(u)$. Note that the length of an arbitrary u-v path is only changed by a constant value $\pi(v) - \pi(u)$ when $w_\pi(\cdot)$ is applied. The potential function $\pi(\cdot)$ is called *feasible* if the reduced edge weights $w_\pi(\cdot)$ are non-negative for all edges $e \in E$. The potential $\pi(u)$ is a lower bound on the distance $d(u, t)$, if $\pi(t) \leq 0$ holds and $w_\pi(\cdot)$ is feasible.
The A* search applies a feasible potential function $\pi(\cdot)$ to speed-up s-t queries. The basic structure of Dijkstra's algorithm is retained but priority keys are changed to $key(u) = d(s, u) + \pi(u)$. Thus, in each step the node u is settled with the shortest estimated path from the source to the target via u. Figuratively speaking, nodes that potentially lead closer to the target are preferred, whereas the other nodes are ignored.
This approach is equivalent to performing a normal Dijkstra search on a graph with the reduced weight function $w_\pi(u, v)$. Since the length of arbitrary u-v paths is only changed by a constant value $\pi(v) - \pi(u)$, the priority keys become $key(u) = d(s, u) + \pi(u) - \pi(s)$. This yields the same sorting as in priority queue above, since $\pi(s)$ is a constant value. Furthermore, this implies that shortest paths are maintained by this transformation. Therefore, running a shortest-path search on the normal graph is equivalent to running one on the graph with reduced edge weights.

Note that if a graph layout is given, the Euclidian distance to the target node is a suitable choice for a feasible potential function (see Fig. 2.4). However, this approach only yields viable lower bounds if the weight function also applies a distance metric, as experimentally shown.

Using a bidirectional A* search, several particularities have to be addressed. Let π_f be the potential in the forward direction and π_b the potential in the backward direction. The two potentials are called *consistent* if the reduced weight function is the same for both. This is true if $\pi_f + \pi_b = const.$ holds. If they are not consistent, the search cannot be stopped when the two search spaces meet since both directions use different weight functions[1]. There are two approaches to implement the bidirectional A* search. The *symmetric approach* proposed by Pohl [Poh71] and the *consistent approach* used by Ikeda et al. [IHI+94]:

[1]Although the path from the source to the meeting node and from the meeting node to the target are shortest paths on their own, a shortest path from the source to the target does not have to run via this meeting node.

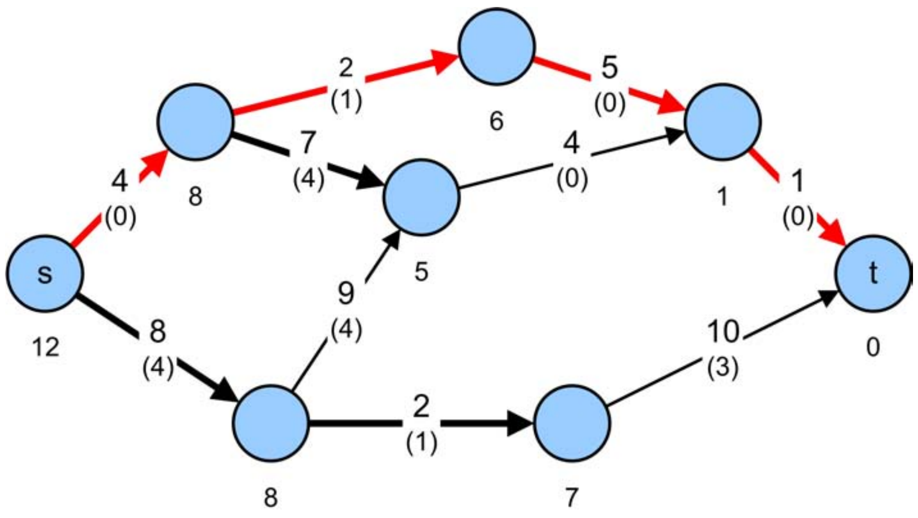


Figure 2.4: *Graph with a feasible potential function. Potential values are written below the nodes and reduced weights are given in brackets below the normal edge weights. The shortest path from s to t is shown in red and the associated search space is drawn in bold.*

Symmetric Approach. This approach uses two non-consistent potential functions. Thus, a new stopping condition is required. Every time an edge $e = (u, v)$ is touched by the query algorithm with v already settled in the other direction, the length of the path from the source to the target via e is computed. This is the tentative shortest path length, if no shorter one has been encountered so far. The search is stopped if one node is about to be settled with a priority key larger or equal to the tentative shortest path distance. Since the priority key is a lower bound on the shortest path length from the source to the target, a shortest path has already been encountered and no shorter one exists.
The query can be improved by pruning the search on nodes v that have already been settled in the other direction [Kwa89]. Since a shortest path from v to the target is already known in the other direction, no further information is gained by continuing the search from v.
The symmetric approach uses potential functions particularly suitable for each search direction. Thus, although the search has to continue well beyond the meeting node of both search spaces, usually few nodes have to settled to arrive at this point, which improves the performance.

Consistent Approach. This approach uses consistent potential functions to avoid having to use a more complicated stopping condition. Let π_f and π_b be two feasible potential functions. Then, Ikeda et al. [IHI+94] propose to use their differences as consistent potentials according to $p_f(u) = \frac{1}{2}(\pi_f(u) - \pi_b(u))$ and $p_b(u) = \frac{1}{2}(\pi_b(u) - \pi_f(u)) = -p_f(u)$. These are both feasible and consistent even though their lower bounds are not very good compared to the single potential functions. Usually, $\frac{1}{2}\pi_b(t)$ is added to the forward potential and, respectively, $\frac{1}{2}\pi_f(s)$ to the backward potential to make them more comprehensible[2]. This does not change the feasibility or the consistency of the potentials since only a constant value was added.

[2]The forward potential becomes zero at the target node $p_f(t) = 0$ and, respectively, the backward potential becomes zero at the source node $p_b(s) = 0$. Note that these are the exact lower bounds for both directions.

The consistent approach applies inferior potential functions compared to the symmetric approach. Thus, usually more nodes have to be settled before the search spaces meet. But due the simpler stopping criterion the search can be directly terminated. Thus, it is often the faster approach.

ALT algorithm

The ALT algorithm is an A* search with a far simpler potential function proposed in [GH04]. It uses landmarks and the triangle inequality of graphs to produce feasible lower bounds. Furthermore it is not dependent on the availability of additional layout information.

In a preprocessing step, a small set of nodes is chosen and labeled as landmarks. Distances from and to all other nodes are computed for them. During the query, a lower bound is computed according to $\mathrm{d}(\mathrm{u},\mathrm{t}) \geq \mathrm{d}_\mathrm{L}(\mathrm{u}) - \mathrm{d}_\mathrm{L}(\mathrm{t})$ with $\mathrm{d}_\mathrm{L}(\cdot)$ denoting distances to L or $\mathrm{d}(\mathrm{u},\mathrm{t}) \geq \mathrm{d}_\mathrm{L}(\mathrm{t}) - \mathrm{d}_\mathrm{L}(\mathrm{u})$ with $\mathrm{d}_\mathrm{L}(\cdot)$ denoting distances from L (see Fig. 2.5), both with respect to the forward search. The backward search is handled respectively. These bounds yield a feasible potential as proven in App. B. To obtain better lower bounds, the maximum of these two values over all landmarks can be used. The potential stays feasible, as shown by Goldberg and Harrelson in [GH04].

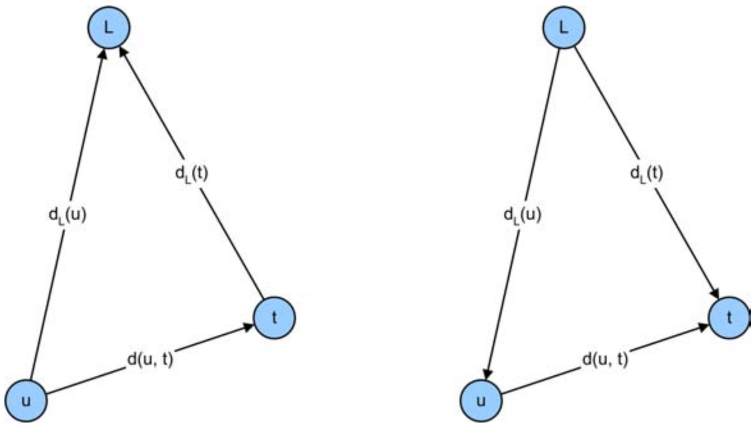


Figure 2.5: *Illustration to clarify the relations for finding lower bounds with the aid of landmarks. Here, the forward search is depicted. On the left side, distances to the landmark are used: $d(u,t) + d_L(t) \geq d_L(u)$. On the right side, distances from the landmark are used: $d_L(u) + d(u,t) \geq d_L(t)$.*

Active Landmarks. Instead of using all landmarks, only a subset is used at the start of the query and adapted as necessary. In general, landmarks located behind the target node of the respective search direction produce good lower bounds. This approach was introduced in [GW05].

Landmark Selection. The choice of the set of landmarks is crucial for the performance of the ALT algorithm. Several approaches are discussed in [GW05]. The *maxCover* and the *avoid* algorithms have shown to be two viable approaches with the former usually producing better and more reliable results.

2.3.3 Reach

The notion of reach in the context of graphs was first introduced by Gutman in [Gut04]. It can be applied either to nodes or to edges, with the latter being more effective but also needing more space. Here, the Reach algorithm is described using node-reaches, but most of the explanation can be easily adapted for edge-reaches.

Definition 1 *The reach $r_P(u)$ of a node u with respect to a path $P = \langle s, t \rangle$ through u is the minimum of the length of the sub-path $\langle s, u \rangle$ and the sub-path $\langle u, t \rangle$. The reach $r(u)$ of u is defined as the maximum of $r_P(u)$ over all shortest paths P containing u.*

Thus, the reach value is a measure for the centrality of the node. A node with a high reach value is more central to the graph in such a way that it is usually close to the middle of a long shortest path, whereas low-reach nodes are located rather near the end of shortest paths. On a road network this differentiation corresponds to important highway roads and non-relevant local roads. In Fig. 2.6, a sample graph with precomputed reach values is depicted.

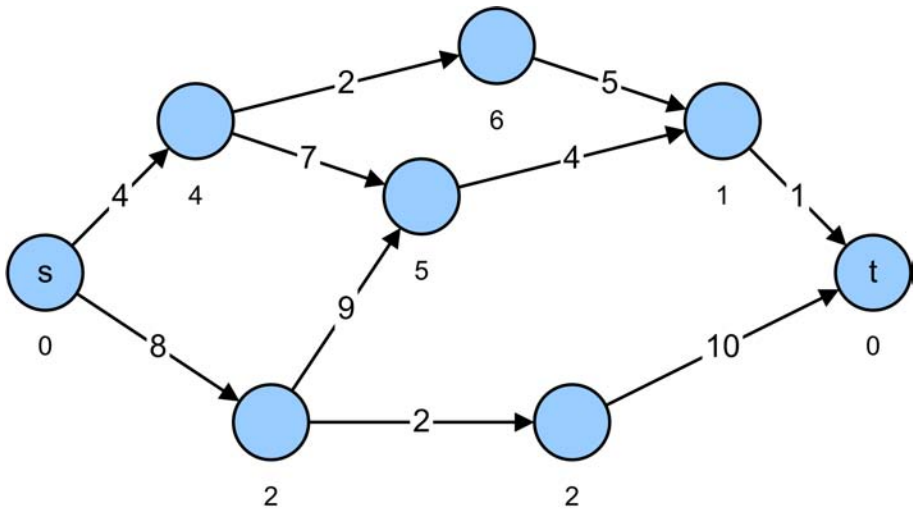


Figure 2.6: *Graph with exact reach values written below the nodes. Note that reach values are larger in the middle of the graph where the associated nodes are more likely to be part of a long shortest path.*

Reach values for each node are computed in a preprocessing step and can be used during the query to prune the search space. If $r(u) < d(s, u)$ and $r(u) < d(u, t)$ holds, v cannot be on a shortest path from s to t and does not need to be touched or settled by the query algorithm. Note that upper bounds on reaches $\bar{r}(u)$ and lower bounds on distances $\underline{d}(u, v)$ suffice for the condition to hold. This fact is utilized later.
Whereas reach values are precomputed and available during the query, and distances from the source to a node v are automatically given by the query, the distance from v to the target is more difficult to be obtained. Gutman [Gut04] suggested using Euclidean distances to compute lower bounds as is done by the A* search. But exactly the same limitations, as described previously in Sect. 2.3.2, also apply here. A graph layout is required to compute the Euclidean distances

and if the weight function is not based on a distance metric, this approach usually does not produce good lower bounds. Goldberg et al. found a more promising way to obtain lower bounds implicitly by using a bidirectional query together with reach-based pruning. They describe three different possible approaches for this purpose in [GKW06a], the *bidirectional bound*, the *self-bounding* and the *distance-balanced* algorithm. They are explained below with regards to the forward direction of a bidirectional s-t query (the backward distance works analogous):

Bidirectional Bound Algorithm. If a node u has already been settled in the backward direction, its distance to the target $d(u,t)$ is automatically given by that direction. If not, the smallest priority key in the backward queue can be used a lower bound on $d(u,t)$ instead. Note that the priority key of a node is equal to its tentative distance from the source of its search direction. Thus, if u has not been settled in the backward direction, the distance of u to the target is at least as large as the smallest priority key of the backward queue.

Self-Bounding Algorithm. The algorithm ignores the pruning condition on the lower bound of $d(u,t)$ and prunes the search at u just on behalf of $\bar{r}(u) < d(s,u)$. The node u can still be settled by the backward search if $d(u,t) \leq \bar{r}(u)$ holds. In other words, if a node is settled, it is done so by the direction to whose source it is closest. This requires a new stopping condition: The search in one direction is stopped if its priority queue is empty or if the smallest priority key in its queue is at least half the length of the shortest path encountered so far.

Distance-Balanced Algorithm. The query alternates between the forward and the backward search by selecting the node u with the smallest priority key, considering both directions. Equivalent to the bidirectional bound algorithm, this key is a lower bound on the distance of u to the target of its search direction, since v has not been settled in the opposite direction. Thus, the search is only terminated if either one of the priority queues is empty, or the sum of the smallest priority keys in each queue is equal to or greater than the shortest path encountered so far.

Pruning of a node is performed right before it is about to be settled. But this can already be done much earlier when considering whether to insert the node into the priority queue or not. Goldberg et al. call this approach *early pruning* [GKW06b]. In the forward direction after settling u and before touching (u,v), the search can be pruned at v if $\bar{r}(v) < d(s,u) + w(u,v)$ and $\bar{r}(v) < \underline{d}(v,t)$ hold. Note that this is the same condition as for the basic case, only adapted for the node v. One of the methods, described above, can be applied to deal with the lower bound on the distance to the target $\underline{d}(v,t)$. Pruning in the backward search is done accordingly. This approach can be further improved by sorting the edges (u,v) belonging to each node u in decreasing order of upper bounds on reaches $\bar{r}(v)$. Thus, if the conditions $\bar{r}(v) < d(s,u)$ and $\bar{r}(v) < \underline{d}(v,t)$ hold for one node v, they are also true for all following neighbours of u, which can be implicitly pruned in turn.

Preprocessing times are the biggest drawback of the Reach algorithm on large graphs. The trivial approach for computing reach values builds full shortest path trees for each node in the graph and applies the reach definition on them to get exact reach values: The reach of u with regards to all shortest paths starting at s can be directly computed as the minimum of the depth and height of u in the shortest path tree rooted at s. Then, the reach of u is the maximum of this value over all shortest path trees. Evidently, this approach is very suitable for large graphs since the preprocessing times quickly increase for larger graphs.

As mentioned earlier, upper bounds on reach values are sufficient for the Reach algorithm. Therefore, Gutman proposed an iterative approach that yields good upper bounds on node-reaches [Gut04]. Goldberg et al. improved this algorithm by introducing shortcuts and using edge-reaches during the preprocessing [GKW06a]. Later, they also introduced a new approach to compute exact reach values [GKW07], which shows similarities to the boundary nodes approach for computing ArcFlags by Möhring et al. [MSS$^+$06], discussed in Section 2.3.4.
The approximate preprocessing algorithm by Goldberg tries to iteratively find upper bounds on reaches. In each step, it tries to find upper bounds that are smaller than a certain threshold ε by growing partial shortest path trees of a depth greater than 2ε. Reach values are computed on the partial shortest path trees as described above for the trivial approach. The trees are grown far enough to ensure that no false positives are found, i. e. reporting reach values of less than ε even though the true value is higher. The algorithm can still produce false negatives but this only degrades the quality of the approximation and does not invalidate it. Nodes with reaches smaller than the threshold ε get bound and subsequently removed from the graph. To accommodate for the deleted nodes and to be able to still compute correct upper bounds on reaches, the neighbours of a deleted node are assigned penalty values, which are used during the reach computation. Before the next step, the threshold is increased by a certain factor and the iteration continues until all reaches are bound or until the iteration is stopped. In the latter case, unbound reaches have to be set to infinity.

Other than using edge-reaches during the preprocessing step and several further contributions, the most notable improvement by Goldberg et al. to Gutman's preprocessing algorithm was the introduction of shortcuts. In each step before starting to grow partial shortest path trees all nodes are checked, whether they are bypassable or not. If a node u gets bypassed, shortcuts are inserted leading around it and u and its adjacent edges are deleted from the graph. Since this removal might alter the bypassability of the node's neighbours, they have to be rechecked afterwards. Note that the order in which nodes are bypassed matters. The bypassability of a node depends on several factors: The ratio of added edges to deleted ones when bypassing the node, the in-degree and the out-degree of the node, the value of the largest reach of an adjacent edge of the node, and the length of the longest shortcut that would be inserted. A more thorough explanation of these criteria can be found in [GKW06b]. The basic notion of bypassing nodes and adding shortcuts according to some criteria is also applied by several other speed-up techniques like the highway networks by Sanders and Schultes [SS05, SS07, BFSS07] and the contraction approach of this thesis.

2.3.4 ArcFlags

ArcFlags, or edge labels as they are also called, have been first introduced several years ago by Lauther [Lau97, Lau04]. They feature a basic concept that is pretty simple:

In a preprocessing step, the graph $G = (V, E)$ is partitioned into several regions $\mathrm{R} = \{R_1, \ldots, R_n\}$ according to some strategy. For each region R_i and each edge $e \in E$, a flag $A_i(e)$ is added, denoting whether e is part of any shortest path leading into R_i. If an edge $e = (u, v)$ completely lies in one region R_i (i. e. $u \in R_i, v \in R_i$), the flag $A_i(e)$ is also set (see Fig. 2.7). All ArcFlags $A_i(e)$ of an edge e together are also referred to as the ArcFlag label $A(e)$ of e.
In addition to the preprocessing, Dijkstra's algorithm has to be modified to make use of the directional information provided by the ArcFlags. After settling a node u and before touching each of its neighbours v, the flag of the associated edge $A_T((u, v))$ is checked with T denoting the region of the target. If the edge is not on a shortest path into the region containing the target of the query, the flag is not set and the search space can be pruned at v.

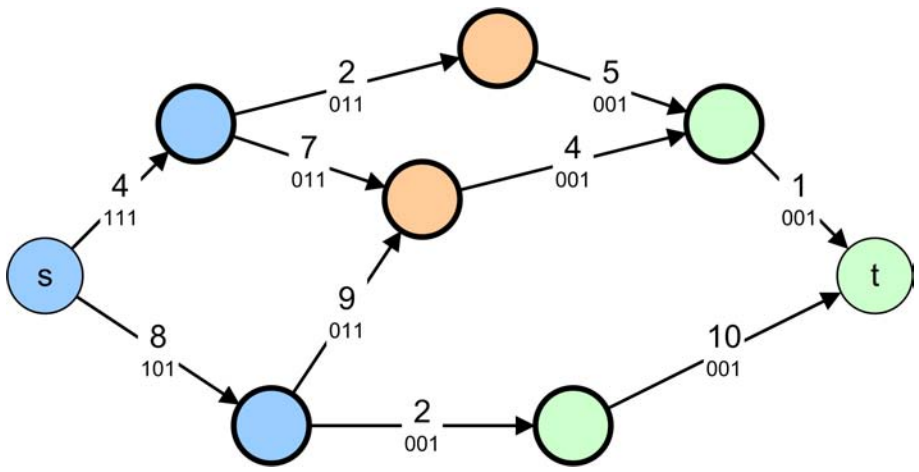


Figure 2.7: *The graph is divided into several regions denoted by different colours. The boundary nodes of each region are drawn in bold. ArcFlag labels for the forward direction are written below each edge in a binary notation (digits denote regions, from right to left: blue, orange, green).*

Since a query has to be only slightly modified to encompass ArcFlags, this method can be easily added to almost any other speed-up technique. In case of a bidirectional query, two sets of ArcFlags are needed, one for the forward direction and one for the backward direction. In addition, the search in the backward direction has to check flags for the source region R_S instead for the target region R_T. The bidirectional approach also decreases a problem of the original ArcFlags implementation called *coning*: When the search gets closer to the target region R_T, the search space starts to fan out, forming a cone. This effect occurs since near R_T usually there are usually more edges that lie on a shortest path leading into that region, and after entering the target region itself, no further directional information is available.

Just as for the Reach algorithm, preprocessing times are the largest drawback of the ArcFlags approach. Furthermore, ArcFlag labels also need a lot of additional space. But because of its performance and of its ease of use, several improvements to the mentioned shortcomings have been devised in the recent years: [KMS05, MSS$^+$05, MSS$^+$06, HKMS06, Hil07].

The basic approach to compute ArcFlags grows two shortest paths trees for each edge $e = (u, v)$, one rooted at u and the other one starting from v. For each node $n \in V$ the length of e is compared to the difference $|d(u, n) - d(v, n)|$. If these distances are equal, e lies on a shortest path to n and the corresponding ArcFlag $A_N(e)$ with $n \in R_N$ can be set.
This approach is very slow since it has to grow two full shortest path trees for each edge of the graph. But it can be improved significantly by exploiting the following observation [MSS$^+$06]: Every shortest path into a region R_i has to enter this region via one of its *boundary nodes*. A boundary node $u \in R_i$ of a region R_i is defined as having at least one neighbour v not belonging to this region $v \notin R_i$. Therefore, it is sufficient to grow shortest path trees from each boundary node of a region R_i in the reverse graph to obtain the corresponding ArcFlags for this region. Since these trees encompass all shortest paths leading into R_i, regarding the forward direction, $A_i(e)$ can be set for each edge e belonging to them.
Hilger further improved this approach by using a *centralized shortest-path search* [Hil07]. As above, only boundary nodes are considered when growing shortest path trees. But here, all trees rooted at the boundary nodes of one region are grown at once. The centralized shortest-path search capitalizes on the observation that shortest path trees grown from close neighbours often share identical sub-trees to significantly speed-up the preprocessing. But this also leads to several new problems. One of them is the huge space consumption of the algorithm. An easy way to avoid memory problems is to only use a part of the boundary nodes of a region at once. Thus, only a smaller shortest path tree has to be kept in memory, but at the cost of some speed-up. Further particularities of this preprocessing algorithm are elaborated in [Hil07].

To counteract the huge memory consumption of the algorithm, ArcFlags can be stored in a compressed way. Instead of adding them directly to all edges, each unique ArcFlag label $A(e)$ is put into a separate array and at each edge only a reference to this array is stored, which takes up less space for a sufficiently large number of used regions. This approach is already applied by [Hil07]. Compressing can already be done during the preprocessing step. Thus, the only additional overhead during the query is an extra table lookup. The query times might even profit from this setup. If a large part of the ArcFlags table can be stored in the CPU cache, access times are reduced significantly in addition to the already accelerated priority queue operations due to the smaller edge objects.

Finding a good partition is crucial for the performance of both, the preprocessing and the query, as shown in [MSS$^+$06]. The algorithm produces correct results independently of the partition used. But in order to obtain good speed-ups, several requirements should be fulfilled by it:

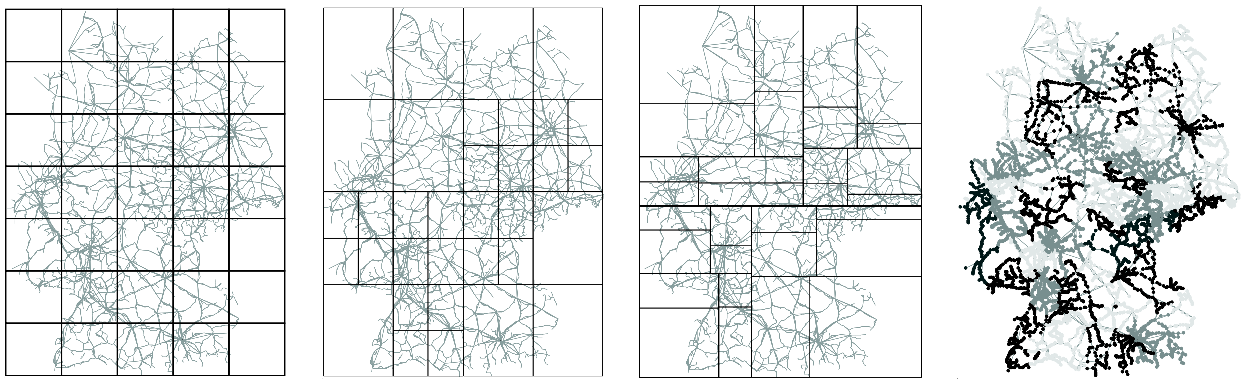

(a) rectangular partition (35 regions) **(b)** quad-tree partition (34 regions) **(c)** kd-tree partition (32 regions) **(d)** Metis partition (32 regions)

Figure 2.8: *Four possible graph partitions of the German road network. [Images by courtesy of Moritz Hilger].*

Each region should be connected so that a search going into this region does not split up into two different directions. The size of the regions should be balanced so that each ArcFlag $A_i(\cdot)$ is responsible for about the same amount of target nodes. The number of boundary nodes should be low, overall as well as for each region, since this directly affects the preprocessing times. More details on the partitioning of graphs can be found in [MSS$^+$06]. Some basic techniques are outlined below and the resulting partitions of the German road network are illustrated in Fig. 2.8:

Rectangular Partitions. This partitioning scheme requires a graph layout. From here on, it is quite simple: The graph is just divided into $n \times m$ individual regions by an equidistant grid.

Quad-Trees. This approach also requires a graph layout. It starts much like the rectangular partition by dividing the graph into four equally sized sub-regions. Now, grid cells containing more than a certain threshold β of nodes are divided again. This is repeated recursively until every cell has no more than β nodes. Thus, the actual graph geometry is reflected better.

kd-Trees. This scheme is a generalization of quad-trees. Here, each division does not have to yield equally sized sub-regions. It is performed alternately in horizontal or vertical direction. The exact position of the cut is determined e. g. by the average of the coordinates of the nodes.

Multi-Way Arc Separators. A partition with all of the desired characteristics can be computed by the multi-way arc separator algorithms presented in [KK98]. Several free and efficient implementations are available: METIS [Lab07], PARTY [MS04] and SCOTCH [Pel07].

Other recent improvements include the use of multi-layer partitions to save space, when storing ArcFlag labels and to improve query times within otherwise much larger regions [HKMS06], the introduction of shortcuts to speed-up queries [BD08], and the compression of ArcFlags to reduce the memory consumption of the algorithm [Hil07].

2.3.5 Highway Node Routing

The Highway Node Routing approach, presented by Schultes and Sanders in [SS07], takes ideas from Highway Hierarchies [SS05] and Multi-Level Overlay Graphs [SWW00, SWZ02, HSW06] and combines them into a new and efficient shortest path algorithm. Basically, it uses the hierarchical levels provided by Highway Hierarchies to construct multiple overlay graphs on which a query subsequently can be run.

For a given graph $G_i = (V_i, E_i)$ and a node set $V_{i+1} \subseteq V_i$ an *overlay graph* $G_{i+1} = (V_{i+1}, E_{i+1})$ of G_i can be implicitly defined by the condition $d_{i+1}(u,v) = d_i(u,v)$ with $u, v \in V_{i+1}$, meaning that all shortest paths between any two nodes $u, v \in V_{i+1}$ have the same distance in the original graph G_i and in the overlay graph G_{i+1}.
Now, a *multi-level overlay graph* denoted by $\mathcal{G} = (G_0, G_1, \dots, G_n)$ can be defined iteratively: For a given hierarchy of node sets $V_0 \supseteq V_1 \supseteq \dots \supseteq V_n$ the overlay graph of $G_i = (V_i, E_i)$ is given by $G_{i+1} = (V_{i+1}, E_{i+1})$, with $G_0 = (V_0, E_0)$ denoting the original graph $G = (V, E)$. The level of a node v is defined by $l(v) = \max\{i \mid v \in V_i\}$, and of an edge e by $l(e) = \max\{i \mid e \in E_i\}$.

A set of edges E_{i+1} that meets the demands made by the definition of overlay graphs, can be computed in many ways. Schultes and Sanders propose the application of *covering paths* for this purpose. By computing a covering-paths set $\mathcal{C}_u$ of a node $u \in V_{i+1}$ with respect to the node set $V_{i+1} \setminus \{u\}$, one also receives a set of covering nodes $C_u \subseteq V_{i+1} \setminus \{u\}$, formed by the endpoints v of the paths $\langle u, \dots, v \rangle \in \mathcal{C}_u$. By definition, every node $n \in V_{i+1}$ can be reached from u via a shortest path in G_i containing at least one node $v \in C_u$ (see Fig. 2.9). Therefore, it is sufficient to compute a covering-node set C_u for each node $u \in V_{i+1}$. Then, the edge set of the overlay graph G_{i+1} is defined by $E_{i+1} = \{(u,v) \mid u \in V_{i+1}, v \in C_u\}$. The corresponding edge weights are given by $w((u,v)) = d(u,v)$, with $d(u,v)$ measured in G_i.

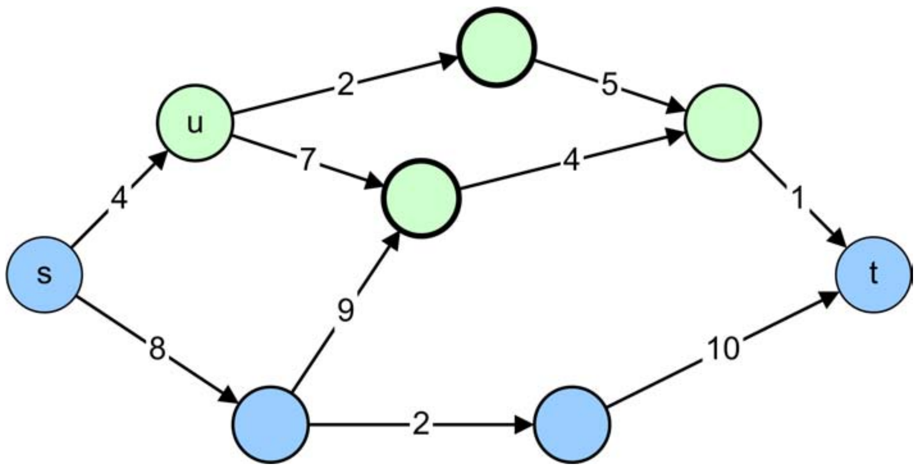


Figure 2.9: *A sample graph is depicted with two hierarchical levels that are differentiated by the colour of their nodes (level 0: blue, level 1: green). The covering-nodes set of* u *with respect to level 1 is highlighted in bold.*

The set of covering nodes C_u should be chosen as small as possible to increase the performance of the algorithm and to reduce its memory consumption. On the other hand, the computation of C_u should also be as fast as possible. These are two conflicting objectives that cannot be optimized at the same time. Schultes and Sanders presented a generic local search algorithm to compute covering-nodes sets and four different implementations with varying trade-offs between the two objectives: The *conservative approach*, the *aggressive approach*, the *stall-in-advance technique* and the *stall-on-demand technique*, with the latter apparently yielding the best trade-offs. They are elaborated more closely in [Sch07].
The generic local search algorithm computes a covering-paths set $\mathcal{C}_u$ of a node $u \in V_{i+1}$ with regards to a node set $V_{i+1} \setminus \{u\}$ and then derives the associated covering-nodes set C_u. The algorithm itself is a modified Dijkstra search on G_i yielding canonical shortest paths[3] with an additional pruning condition: The search is stopped prematurely at covered nodes and at nodes settled on a suboptimal path. Here, pruning implies that after the node is settled the search is not continued from it. The search is terminated as soon as the current search tree is covered, i. e. each tentative shortest paths to every leaf of the tree contains at least one node $v \in V_{i+1}$. Then, the first node $v_c \in V_{i+1}$ encountered on each of these shortest paths is the endpoint of a covering-path $\langle u, \ldots, v_c \rangle \in \mathcal{C}_u$. Thus, the set of covering nodes C_u is given by the nodes v_c.

The choice of the node sets $V = V_0 \supseteq V_1 \supseteq \ldots \supseteq V_n$ for each highway level is arbitrary in the sense that it always produces correct queries. The actual choice, however, has a large impact on the performance of the preprocessing and the query. Figuratively speaking, nodes that are more important for the graph (like motorways are more important than rural roads in a road network) should belong to a higher level. To obtain a good classification of the importance of the nodes, Schultes and Sanders applied their own Highway Hierarchies approach: The set of level-i core nodes of the highway hierarchy of G is used as highway-node set V_i. More about the construction of Highway Hierarchies can be found in [SS05, Sch07].
Recently, the newly developed Contraction Hierarchies [Gei08] have also been applied for classifying nodes according to their importance. So far, this approach seems to yield even better results.

The *highway nodes graph* is given by $H = (V, E_0 \cup \ldots \cup E_n)$. To perform the query, this graph can be reduced to $\overline{H}$ by removing all edges not belonging to the forward or backward search graph. In particular, this includes all edges leading from one highway level to a lower level. The reduced highway nodes graph not only helps to save space, the query also gets simpler, and thus, faster since highway levels no longer have to be verified explicitly.

The query on the highway nodes graph is done bidirectionally, alternating between the forward and the backward search after each step. When using the reduced graph $\overline{H}$ a simple Dijkstra's

[3]Consider two shortest paths $P = \langle s, \ldots, s', \ldots, t', \ldots, t \rangle$ and $P' = \langle s', \ldots, t' \rangle$ found by an algorithm that computes *canonical shortest paths*. If there are multiple shortest paths from s' to t', the algorithm always yields the same one, independent of the actual choice of s and t for the enclosing path P.

algorithm can be performed in each direction. If the full graph H is used instead, it has to be guaranteed that the search only moves to higher levels of the hierarchy and does not descend to lower ones: For each node v that is about to be reached via u, $l(v) \geq l(u)$ has to be true or the search is pruned at v. The query cannot be stopped when both search spaces meet, since a shorter path might still be found (see Fig. 2.10). The search is continued in each direction until the priority queue of the respective direction either is empty, or its smallest element has a tentative distance from the source larger than the shortest path encountered so far. This approach is called *asynchronous, aggressive variant* by Schultes in contrast to the *level-synchronized variant*, he also proposed in [SS07].

The *stall-on-demand technique* that has already been used for the preprocessing can also be applied to the query. It identifies nodes in the priority queue that will not be part of the final shortest path tree, according to the current information, and marks them as *stalled*. If a stalled node is settled, the search is not continued from it. On the other hand, if it is first reached via another node, it is unstalled again. The stalling process works as follows:
When a node v is settled and one of its neighbours u has already been stalled or settled before, the condition $d(s,u) + w(u,v) < d(s,v)$ is checked. If true, v has not been settled on a shortest path, since the path via u would have been shorter. Now, a stalling process is initiated to identify other nodes, that would be settled via v and therefore would not be on a shortest path from s. A breath first search on all reached nodes w is started from v. If their distance from the source via the edge (u,v) is shorter than their tentative distance $d(s,w)$, they are marked as stalled and the search is continued from them. After the stalling process has finished, the search is continued and pruned at v, since it has been stalled, too.
Note that instead of resorting to the tentative distances $d(s,\cdot)$ provided by the query, shorter ones that have been encountered by a stalling process can be stored and used by subsequent stalling processes to improve their performance.

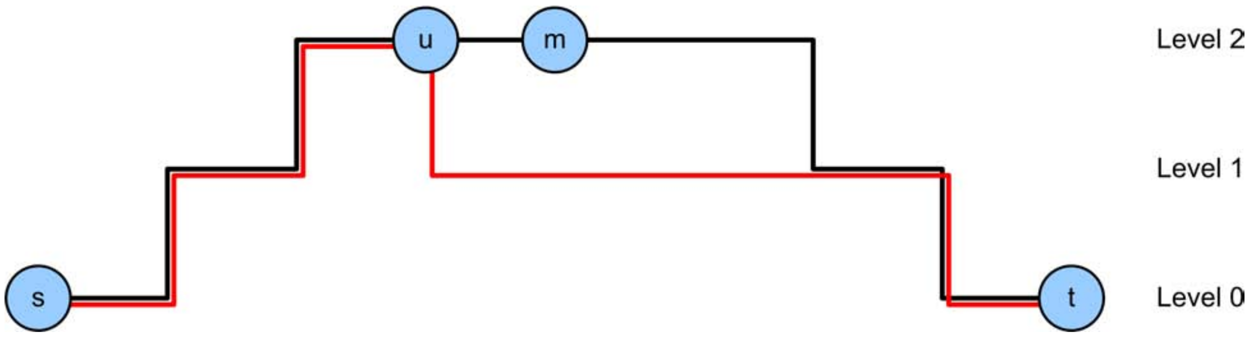


Figure 2.10: *A schematic view of an arbitrary path from s to t (black) and of a shortest path (red) is shown. At* u*, the forward search has to continue on the upper level and cannot descend to the lower one, leading to a suboptimal path via the first meeting node* m*. Thus, the query is dependent on the backward search to reach* u *on the lower level even after a path via* m *has already been found.*

2.4 Combinations

The speed-up techniques described so far can be divided primarily into two different approaches. Some apply goal-directed strategies to push the search in the direction of the target, whereas others try to exploit a hierarchical structure inherent to the graph to decrease the search space. Several combinations of these techniques are studied in the thesis (see Fig. 2.11 for an overview).

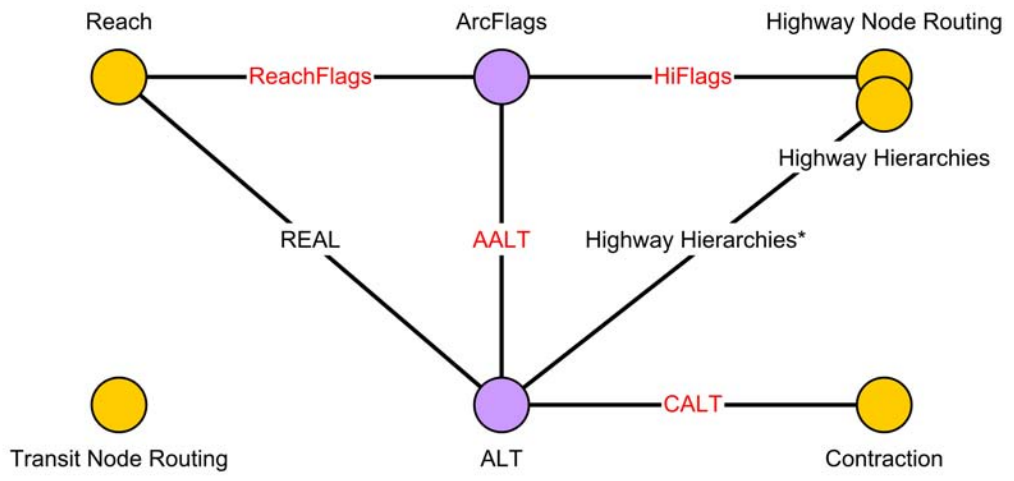


Figure 2.11: *Overview of the techniques and combinations presented in this thesis. Goal-directed approaches are marked purple and hierarchical ones are drawn in yellow. The bidirectional search is omitted for clarity. Edges denote combinations of techniques, black ones are already established and red ones are introduced in this thesis.*

In general, any techniques could be combined with each other. This has already been studied systematically in [HSW04, HSWW06] for the A* search, the bidirectional search, the separator-based multi-level method, and geometric containers. It turns out that the combinations of some techniques are more useful than of others. In particular, it has been observed that combing two different approaches is most promising whereas a combination of two similar techniques usually does not yield viable results since they tend to exploit the same aspects of the graph. Considering these results, a combination of the two hierarchical techniques Reach and Highway Hierarchies is not very promising. Both apply similar preprocessing steps to gain further information of the graph. Thus, the hierarchical structures obtained are not very distinct and do not support each other much. These techniques have also become quite complex and combining them would not only be unreasonably intricate, but probably yield a query algorithm with a large overhead.

Considering these thoughts, the thesis at hand focuses on studying the following combinations:

REAL, Highway Hierarchies*. Two combinations of ALT with hierarchical techniques have already been studied previously, REAL and Highway Hierarchies*. They are presented briefly in Chap. 3 to draw comparisons with the newly introduced techniques and to round off the review of the combinatorial techniques.

AALT. Even though the expectations are small for a combination of two goal-directed approach, the combination of ALT and ArcFlags has still been studied in Chap. 3. And surprisingly, they seem to exploit different directional information of the graph since a measurable speed-up is obtained, as the experiments have shown.

CALT. The basic contraction approach is a simple, recent hierarchical method that was inspired by the preprocessing steps of other hierarchical techniques. A combination with ALT rather than ArcFlags is studied with regard to a possible future application on dynamic graphs, for which the ALT technique would be more suitable than ArcFlags. The resulting combination, CALT, is presented in Chap. 4.

ReachFlags, HiFlags. After both, Reach and Highway Hierarchies, have been combined with ALT, a logical next step is the study of their performance in combination with another potent goal-directed technique, ArcFlags. The two developed techniques Reach-aware ArcFlags and Hierarchy-aware ArcFlags are presented in Chap. 5.
Note that both, Highway Hierarchies and Highway Node Routing, originally used the same basic construction algorithm to obtain hierarchical levels of a graph. Therefore, similar performances were expected when combining them with ArcFlags. And thus, eventually, Highway Node Routing was chosen instead of Highway Hierarchies since it is much easier to implement and handle.

After the different combinations, highlighted in Fig. 2.11, are described in the following chapters, an extensive evaluation of their performance on different types of graphs is given in Chap. 6.

Basic and Existing Combinations

This chapter introduces a first basic combination of speed-up techniques, the AALT algorithm. It adds ALT to the ArcFlags query in order to obtain better results for local queries. Note that both applied techniques are goal-directed unlike the other combinations presented in this thesis. Afterwards, two recent combinatorial techniques are presented to provide an insight on their functionality that is reflected by the other techniques introduced later in this thesis. Both of them also use the goal-directed ALT algorithm and add it to hierarchical techniques. REAL [GKW06a] applies the Reach algorithm [GKW06a] as basis for this purpose, and the Highway Hierarchies* algorithm [DSSW06] uses the normal Highway Hierarchies [SS05]. They also enhance the single techniques by exploiting synergy effects between them.

3.1 AALT

The *AALT algorithm (ArcFlags & ALT)* is an unusual combination of speed-up techniques, in the context of this thesis, since two similar techniques are applied. Both, ALT and ArcFlags, are goal-directed methods that try to push the search in the direction of the target. As explained in Chapter 2, the ALT algorithm superimposes a potential function on the graph and uses it to determine the order in which nodes are processed. The potential itself yields lower bounds on the distance to the target node and is computed using distances from and to landmark nodes. The ArcFlags method divides the graph into several regions and adds a flag for each region to all edges, stating whether the edge is on a shortest path into the respective region. The flags are then used to prevent the query from pursuing wrong directions.
One of the disadvantages of the ArcFlags algorithm is its weak performance for local queries. Several different approaches exist to remedy this shortcoming. Multi-level partitions [MSS$^+$06] and SHARC Routing [BD08] both apply smaller partitions to decrease the impact of these queries. Here, the ALT algorithm is used instead, since it does not exhibit similar problems with local queries. Another goal of this combination is to improve upon the additional costs of both techniques. The ArcFlags algorithm has long preprocessing times and the ALT algorithm a large memory overhead. By using smaller numbers of regions or landmarks, these costs can be reduced, but the performance of the single techniques also degrades. It is expected from the combination of ArcFlags and ALT that the drop in performance is less pronounced due to the utilization of synergy effects between the two speed-up techniques.

Preprocessing

The preprocessing step of the AALT algorithm consists of the individual preprocessing routines required by the ALT algorithm (selection of landmarks, computation of landmark distances) and the ArcFlags algorithm (partitioning of the graph, computation of the ArcFlags). These routines do not have to be adapted for AALT. They can be used directly 'out-of-the-box'.

Query

The query algorithm of the bidirectional ALT algorithm, as described in Sect. 2.3.2, is taken as basis for the AALT query. It has only to be modified at two positions: Before starting each search, the regions, in which the source and target node are located, need to be determined. During the query, after touching an edge e and before inserting its target node u into the priority queue, the flags of e are checked. In the forward direction, if the flag for the target region is not set, the search is pruned at u, since continuing in the direction of e would not lead to the desired destination. The backward direction is handled respectively. The other aspects of the bidirectional ALT query, such as the termination condition or the computation of the priority keys, do not have to be changed.
Fig. 3.1 shows the search spaces of a query on the road network of the Netherlands using different algorithms. From left to right, an ALT query with 16 landmarks, an ArcFlags query with 16 regions and an AALT query with 2 regions and 4 landmarks are shown. Even though the AALT algorithm uses much less additional information, the search space of the query is smaller than for the other two algorithms. In addition, the positive impact of the AALT algorithm on local queries can be observed in Fig. 3.2. The query using only ArcFlags information is reduced to a bidirectional Dijkstra query, since the algorithm cannot benefit from this data for queries within one region. The AALT algorithm, on the other hand, can still profit from the bidirectional ALT and also yields good performances on these local queries.

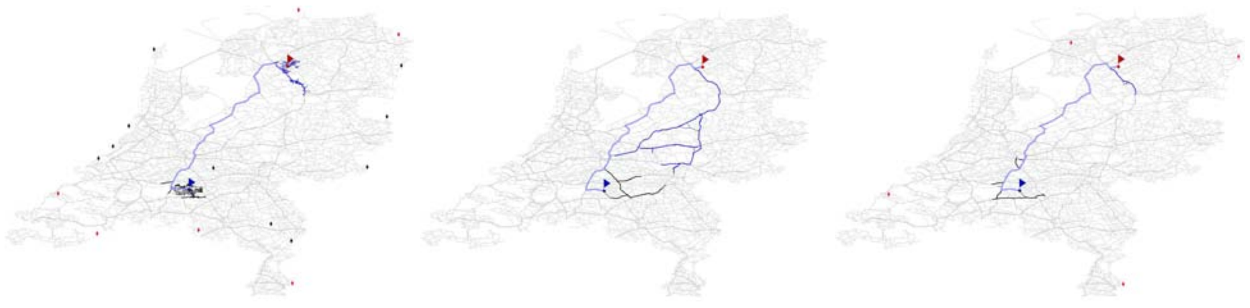

(a) ALT (16 landmarks) **(b)** ArcFlags (16 regions) **(c)** AALT (2 regions, 4 landmarks)

Figure 3.1: *Comparison of the search spaces of ALT, ArcFlags and ALT for a query on the road network of the Netherlands. The source is marked by a blue flag, the target by red one. Black edges represent the forward direction and blue edges the backward direction. The shortest path is drawn in bold. Diamonds indicate landmarks with active ones in red. Here, the ALT query with 16 landmarks has to settle 4 442 nodes, ArcFlags with 16 regions needs 1 049 nodes and the AALT query with 2 regions and 4 landmarks only requires 785 nodes.*

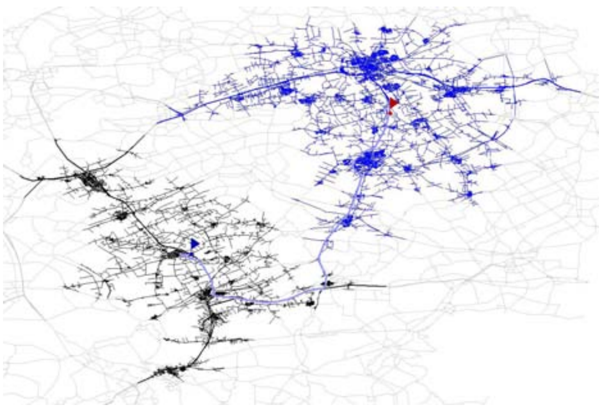
Further Optimizations

Since the preprocessing routines of the ALT algorithm and for computing ArcFlags do not have to be modified in order to be used with the AALT algorithm, all optimization methods that are applicable for each individual preprocessing routine can still be used. In particular, the methods proposed in Chap. 4 for the ALT preprocessing and in Chap. 5 for the computation of the ArcFlags can be also applied here.

Space Consumption. Since the AALT algorithm profits of two speed-up techniques, the parameters of each preprocessing routine can be tuned down to save time and space without losing much of its performance, if any at all. For example, reducing the number of regions for the ArcFlags algorithm to 16, whereas 128 would be normally used, is a reasonable measure. A more drastic alternative would be to only compute ArcFlags for one direction or to only compute landmark distances either from all nodes or to all nodes, but not for both directions. The former suggestion would lead to an additional speed-up in only one direction; the latter would decrease the quality of the lower bounds on the distances to the target.

Other Applications

A useful application for combining ArcFlags and ALT has been observed by Pajor in [Paj08]. Here, the problem of finding shortest connections in timetable information systems is analyzed. It is stated that the search space can be divided into two domains, a temporal one and a geographical one. Searches within each domain are independent of each other. ArcFlags are only used for the geographical part of the timetable queries and the ALT algorithm is only applied to the temporal part. Both of them are the best choice of speed-up techniques for their respective task. Thus, they support each other nicely, and together, the performance on timetable networks achieved by them is quite noteworthy.


(a) local ArcFlags query

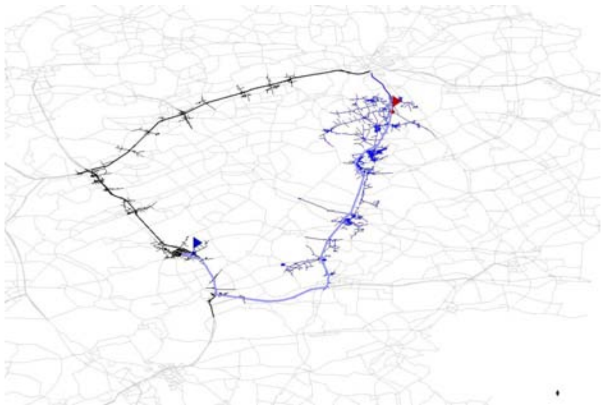

(b) local AALT query

Figure 3.2: *Comparison of the search spaces of the ArcFlags algorithm and the AALT algorithm regarding local queries within one region. The ArcFlags query acts like a bidirectional Dijkstra and has to settle 38 076 nodes. The AALT query performs like the bidirectional ALT algorithm and settles only 5 736 nodes.*

3.2 REAL

When introducing the Reach algorithm in [Gut04], Gutman already stated that the A* search could be naturally combined with it. When improving this speed-up technique, Goldberg et al. also incorporated their enhanced version of the A* search, the ALT algorithm, and called the new method the *REAL (Reach and ALT) algorithm* in [GKW06a, GKW06b, GKW07].

Preprocessing

The REAL speed-up technique requires two preprocessing algorithms. The preprocessing routine for the Reach algorithm computes shortcuts and reach values, the ALT preprocessing selects landmarks and computes distances between them and all the other nodes. The two preprocessing routines can be performed independently of each other. Although the preprocessing of the Reach algorithm changes the graph structure by adding shortcuts, distances are maintained. Therefore, the landmarks and landmark distances computed for the original graph can still be applied. However, it might be useful to use the additional information, obtained by the Reach preprocessing, to improve the ALT preprocessing, as described later.

Query

As already mentioned, the Reach algorithm can be easily combined with the A* search. The same is also true for the ALT algorithm, since it is just a variant of the A* search with a special way of computing the potential function $\pi(\cdot)$. The general approach to combine the Reach and the ALT algorithms is similar to the combination of ArcFlags and ALT, presented in the last section. A normal ALT query is performed and nodes (or edges, if using edge-reaches instead of node-reaches) are pruned according to the reach conditions, introduced in Sect. 2.3.3: When a node u is touched, the search is pruned if its reach value r(u) is smaller than both, its distance from the source d(u) and the lower bound on the distance to the target π(u). This approach works out, since the ALT algorithm retains shortest-path distances, although edge lengths change according to the potential function.
Usually, a bidirectional query is applied. In this case, appropriate potential functions are used for the search in both directions. The algorithm alternates between the forward and the backward search after each step to balance the load on the respective directions. Note that instead of using implicit lower bounds as for the bidirectional Reach algorithm, lower bounds provided by the potential function $\pi(\cdot)$ are used, since the priority keys no longer provide distances from the source or target, respectively. The reach-based pruning does not affect the stopping condition of the ALT algorithm. The query is terminated as soon as the two search spaces meet or if one of the priority queues is empty.
Fig. 3.3 gives an impression of the search spaces of the ALT and Reach algorithms compared to a REAL query, using the same preprocessing information as used for the ALT and Reach queries.

Further Optimizations

Locality Effects. The utilization of *locality effects* is a common optimization strategy. Since reach values provide a very fine hierarchy, simply sorting the nodes according to them might not always prove to be advantageous. Furthermore, an already existent locality order, e. g. given for many road networks, would also be not respected, and thus, be destroyed by this resorting. Goldberg et al. proposed to sort the $1/\alpha$ fraction of nodes with the highest reach values to the front, retaining their original order within this fraction and continuing the sorting recursively on the remaining nodes. In this way, both sources of locality effects are taken into account, the initial node-order and the reach values.

Reach-Aware Landmarks. Goldberg et al. introduced the notion of *reach-aware landmarks* for their REAL algorithm. By only storing landmark distances for nodes with a high reach value above a certain threshold R and by only choosing landmarks from this set of nodes, the preprocessing times and especially the memory overhead can be improved considerably. As experimentally shown, the impact on the query performance is minimal, since the search has to process low-reach nodes only for a short time at the beginning of the query.
The REAL query has to be modified to accommodate for the reach-aware landmarks. This results in the *partial landmark algorithm*. It starts as a bidirectional Dijkstra search with reach-pruning until it either terminates or the priority keys in both queues become larger than R. From here on, only nodes with a reach value larger than R have to be settled. Since landmark distances are available for all of these nodes, the Dijkstra search can be switched to an ALT search by flushing the priority queues and reinserting all elements with modified key values. Unfortunately, the ALT algorithm also requires landmark distances for the source and target, which are not always available. High-reach proxy nodes are introduced to solve this problem. More information on the use of proxy nodes can be found in Sect. 4.3. Here, the CALT algorithm has to deal with the same problem. The remainder of the query is performed like the normal REAL algorithm until a shortest-path is found or the search terminates for another reason.

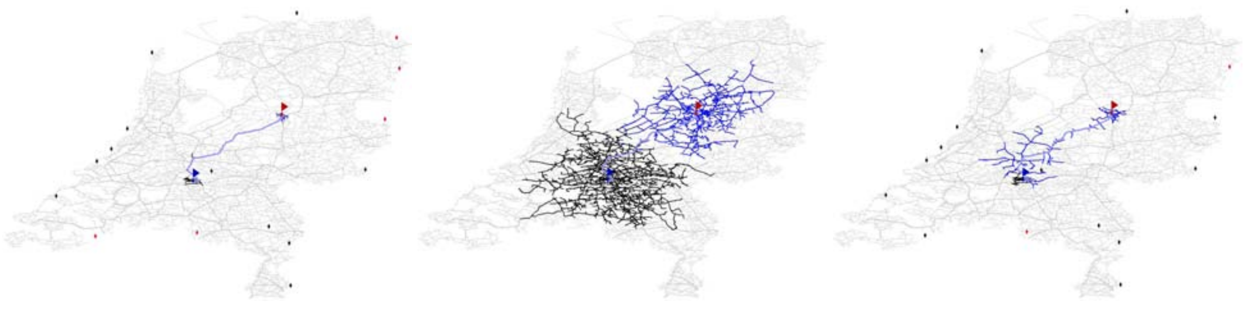

(a) ALT (16 landmarks) **(b)** Reach **(c)** REAL-(16, 1)

Figure 3.3: *Comparison of the ALT, Reach and REAL algorithms on the road network of the Netherlands. The REAL query applies the same preprocessing information as the two other queries. Here, the ALT query has to settle 2 004 nodes, the Reach query needs to settle 2 833 nodes and REAL-(16, 1) only settles 151 nodes.*

3.3 Highway Hierarchies*

The Highway Hierarchies* speed-up technique was introduced by Delling et al. in [DSSW06]. It is an extension of the Highway Hierarchies approach by Sanders and Schultes [SS05] similar to how Goldberg et al. improved upon the Reach algorithm by Gutman [Gut04], which resulted in the REAL speed-up technique [GKW06a]. In both instances, goal-direction in the form of the ALT algorithm is added to a hierarchical speed-up technique in order to improve its performance. Additionally, the hierarchical information provided by the base algorithm can be used to enhance the ALT part of the combined algorithm.

Preprocessing

As with most combinations of speed-up techniques, the preprocessing required by the Highway Hierarchies* algorithm also consists of two separate preprocessing steps. At first, the preprocessing for Highway Hierarchies is performed to compute the highway structure of the graph and to add shortcuts to the graph. Then, the preprocessing for the ALT algorithm is done by selecting landmarks and computing landmark distances. If the optimization strategies described below are not used, the preprocessing routines of Highway Hierarchies and ALT are independent of each other, since only the former one alters the structure of the graph, but it also retains all nodes and shortest-path distances between them. Thus, the actual data used by the ALT preprocessing routine stays the same.

Query

The Highway Hierarchies* query is a modified version of the normal Highway Hierarchies query. The normal query is an altered bidirectional Dijkstra search that ascends a hierarchy of highway-levels without going back to lower levels. A full distance table is used for the top-most level, containing all distances between nodes of this level. When using Highway Hierarchies*, the normal priority keys are replaced by priority keys computed with the ALT algorithm. Thus, the order in which nodes are settled is modified to usually prefer nodes closer to the target. As experimentally shown, some path from the source to the target is found pretty quickly, but the search cannot be stopped when both search spaces meet. The same problem exists for Highway Node Routing and is discussed more closely in Sect. 2.3.5. This is an evident drawback of Highway Hierarchies* compared to REAL, which can stop once a path is found. But because of the altered stopping condition, non-consistent potential functions can be used that are often easier to compute and that also often produce better lower bounds.
In addition, by having found one path between the source and the target, an upper bound on the shortest-path distance is available. Thus, the search can be pruned at nodes and edges that would yield a path with a longer distance than of the current tentative shortest-path. Note, if a node is pruned that is about to become settled, the whole priority queue can be flushed, since the pruning condition will also be true for the following nodes with even higher key values.

Note, it has been experimentally shown that selecting only one landmark in each direction is sufficient for the query, since the search is concentrated on only a small area around the source and target before the distance table can be applied. In particular, no additional landmarks have to be activated during the query, which would produce a large computational overhead due to both priority queues having to be rebuilt.

As experimentally shown, the actual speed-up gained by applying the ALT algorithm is quite small if the underlying Highway Hierarchies already apply a distance table. If not, the added technique has a much larger impact on the gain in performance of Highway Hierarchies*.

Further Optimizations

Space Consumption. Similar to the *reach-aware landmarks* approach previously introduced by Goldberg et al. in [GKW07], the memory consumption and preprocessing times of Highway Hierarchies* can also be reduced by storing landmark distances only for some subset of nodes. Here, the highway levels are suitable attribute to decide for which subset of nodes landmark distances should only be stored.
Naturally, the query has to be adapted if landmark information is only available for some higher level L of the hierarchy. The modified search starts as a normal Highway Hierarchies query until either a shortest path has been found, or all entry points into level L have been encountered. Now, the priority queues are flushed and refilled again using the modified key values of the ALT algorithm. Afterwards, the search is resumed on the highway level L, applying the goal-direction of ALT. It is continued until the stopping criterion of Highway Hierarchies* is met.
Note that this approach also requires landmark distances for the source and target as does the partial landmark algorithm for REAL. But here, the distances can be computed directly during the query instead of using preselected proxy nodes.

Core Landmarks. Independently of computing landmark distances for only a certain level of the hierarchy, the landmarks themselves can also be selected with regards to just some higher level. Delling et al. call this approach '*using core landmarks*'. The corresponding core of such a hierarchy level usually consists of a lot less nodes than the full graph, thus, landmarks normally can be chosen much faster. The maxCover algorithm is usually applied for this purpose, since the core is small enough for it to be a viable choice.

Locality Effects. Taking advantage of *locality effects*, as described more closely in Sect. 4.2 for the CALT algorithm, is an obvious choice for optimizing the performance of the Highway Hierarchies* query. Here, it is suitable to sort all nodes according to their highway level while keeping their original order within the scope of one level. This retains the inherent locality of the original graph, if existent, and improves upon it by incorporating the additional information, provided by the highway levels, into the sorting.

Contraction Approaches

This chapter deals with the contraction of graphs and how to utilize this technique for speeding-up shortest-path queries. At first, the basic concepts of graph contraction will be laid out. Following this introduction, it will be explained how a simple shortest-path query can be performed on a contracted graph, leading to the *Contracted Dijkstra (CD)* speed-up technique. Finally, the last section shows how this new technique can be combined with other speed-up techniques. The ALT algorithm was chosen for this purpose, which results in the *Contracted ALT (CALT)* speed-up technique.

4.1 Graph Contraction

The basic idea of contracting graphs was introduced by Sanders and Schultes for Highway Hierarchies in [SS05] and later significantly enhanced in [SS06]. The Reach algorithm by Goldberg et al. [GKW06a, GKW07] also applies the contraction methods introduced by Sanders. Another recent technique relying on contracting graph is Highway Nodes Routing [SS07].
A graph $G = (V, E)$, as generally used in this field of research, contains a lot of nodes that have very few connections to the other nodes. The goal of the contraction is to identify these nodes and to remove them, but retaining shortest-path distances between the remaining nodes by inserting additional shortcuts. The resulting *contracted graph* is also labeled *core* $G_C = (V_C, E_C)$ of the graph G, since it represents the basic structure of the graph.
The contracted graph G_C cannot be directly used for shortest-path queries. At first, it has to be merged with the original graph. This results in the full graph with contraction shortcuts $G_F = (V_F, E_F)$, which can subsequently be used to run queries on.

Contraction Process

The contraction of the graph is performed iteratively. Nodes are *bypassed* according to a certain *order* until no further nodes are *bypassable*. A node n is bypassed by removing it from the graph along with all of its ingoing edges $E_I(n)$ and outgoing edges $E_O(n)$. For each pair of removed edges $(u, n) \in E_I(n)$ and $(n, v) \in E_O(n)$ with $u \neq v$, a shortcut (u, v) is inserted into the graph. Its weight is set to the sum of the weights of the removed edges: $w(u, v) = w(u, n) + w(n, v)$. If there already was an edge e in the graph, connecting u and v, the shortcut is not inserted. But if the weight of the shortcut would have been smaller than $w(e)$, it is used instead.

Bypassability Criterion

In principal, every node could be bypassed by executing the procedure outlined above. But doing so is not always favourable. Generally speaking, the contraction should reduce a graph to its more 'important' nodes and decrease its overall size. But by removing a node with a lot of neighbours the graph size might even increase, since too many shortcuts would have to be inserted. In addition, using shortcuts that are too long has also been shown to be inefficient for shortest-path queries [GKW06a]: Since these shortcuts bypass large areas of the graph, a query starting in the middle of such an area would need a long time to leave it and be able to use of the shortcuts. The *hop count* of a shortcut is defined by the number of edges in the original graph that are covered by it.

These considerations lead to several conditions that have to be met in order for a node u to be deemed bypassable. In particular, the following four attributes of u are regarded and have to be bound by certain threshold values:

- **expansion factor e**
  The expansion factor is defined to be the quotient of the number of new shortcuts, that would be added to the graph if u is bypassed and the sum of the current in- and out-degree of u. A low expansion factor denotes that by removing u the graph will not be changed much.
- **maximum hop count H**
  The maximum hop count is defined to be the maximum number of hops over all shortcuts, that would be inserted into the graph by bypassing u. This parameter simply controls the length of the shortcuts. As already explained above, very long shortcuts are not beneficial for shortest-path queries.
- **in-degree i, out-degree o**
  The in-degree and the out-degree are defined to be the number of ingoing and outgoing edges of u. This parameter is used to control the growth of the graph in order to prevent it from getting to dense. A dense graph is inconvenient for a lot of preprocessing and query algorithms, since they are usually designed and optimized for sparse graphs. Also, the contraction itself would suffer, since a node with a high degree is less likely to be bypassed.

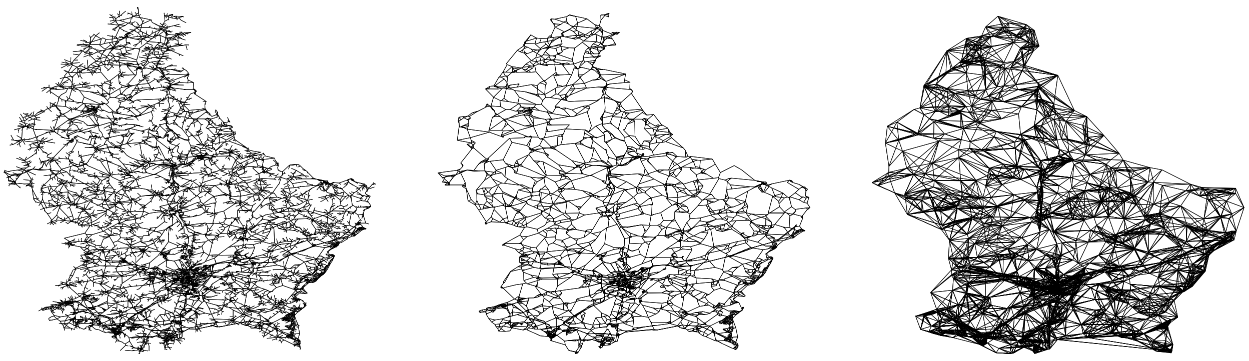
The contraction quality can be adjusted by tuning the threshold values of these attributes with the following parameters of the contraction process: the *contraction parameter c*, the *hopBound h* and the *maxDegree d*. Larger values usually lead to a greater contraction with fewer nodes and more shortcuts. Note that choice of the values for c and h has an effect on each other. For example, let h have a small value. Then, shortcuts are limited to a certain small maximum size. Thus, no further nodes can be bypassed for sufficiently large values of c since by doing so at least one shortcut would become too long. Therefore, further increasing c would not have any effect on the contraction. Also, take note that the value of d is set to infinity for all contractions performed in this thesis.

Bypassing Order

The order in which the nodes of a graph are bypassed is important: When a node is removed, the in- and out-degree of its neighbours change and so does their bypassability. Therefore, the contracted graph G_C, resulting after all bypassable nodes have been processed, depends on the order in which this is done. To determine a practical order, all nodes that are bypassable can be managed by a heap structure, according to Goldberg et al. [GKW07]. The key of a node u in the heap is defined as the product of the *expansion factor e* and the *maximum hop count H* of u. Smaller key values denote a higher priority to bypass the node. The key is chosen in such a way, that unimportant nodes that will not change the bypassability of a lot of other nodes are more likely to be removed first and that the creation of long shortcuts is discouraged.
In order to improve the quality of the contraction, nodes are reevaluated during the contraction process: After a node is bypassed, all of its current neighbours are determined and reinserted into the heap with a newly computed key. If they are already in the heap, only their key is adjusted according to the new conditions.

Contracted Graphs

The road network of Luxembourg and two contractions of this graph using different parameter sets are shown in Fig. 4.1. Figure (b) shows conservative contraction. Only degree-2 nodes or less are bypassed and shortcuts are restricted to just 10 hops. A much more aggressive contraction is depicted in Figure (c). Here, a lot of nodes have been removed but at the cost of a large increase in added shortcuts.
In order to use these contracted graphs G_C for shortest-path queries, they have to be incorporated into the original graph G: All shortcuts are added to G and changed weights of already existing edges are also carried over. In addition, every node and edge belonging to the core are marked with a flag *coreNode* or *coreEdge*, respectively. This results in the full graph with contraction shortcuts $G_F = (V_F = V, E_F = E \cup E_C)$.



(a) full graph (no contraction) **(b)** contracted graph (c=0.5, h=10) **(c)** contracted graph (c=2.0, h=30)

Figure 4.1: *Road network of Luxembourg with different contraction parameters. To the left, the original graph is shown. In the middle, a conservative contraction can be seen. On the right, an aggressive variant is depicted.*

Optimizations

Removing Shortcuts. The contraction quality can be improved by removing unnecessary shortcuts. Sometimes, shortcuts are added to the graph provide a longer connection between two nodes than an already existent path between them. Removing these shortcuts does not alter shortest paths but reduces the graph size and thus preprocessing and query times, too. The removal is performed after the graph has been fully contracted: A Dijkstra search in the contracted graph $G_C = (V_C, E_C)$ is started from each node $u \in V_C$ and continued until all neighbours v of u with $(u, v) \in E_C$ are settled. If one of its neighbours v was settled via more than one edge, the direct edge between u and v can be removed.
It is also beneficial to detect such superfluous shortcuts directly during the contraction process and remove them [Gei08]. This decreases the in- and out-degrees of the involved nodes, and thus, increases their bypassability. Being able to bypass more nodes or to bypass them in a different order might improve the quality of the resulting contraction. But this approach also increases the execution time of the contraction process because of the additional Dijkstra searches that have to be performed to identify the shortcuts. Thus, the shortcut-removal step is only performed some limited times during the contraction process and once afterwards.

Iterative Contraction. The management of candidate nodes is a large problem for the contraction algorithm. In the worst case, the heap has to hold ever node of the graph. This not only requires a lot of memory, it also considerably slows down operations on the heap. In order to decrease the memory consumption of the algorithm and the load on the heap structure, not all bypassable nodes have to be inserted into the heap at once. The contraction process can be performed iteratively in batches: Only a certain amount of nodes is inserted and processed, at once, before the next batch of nodes is inserted, continuing until all have been processed.
Using an iterative contraction creates new problems. First, the contraction quality decreases since the algorithm can no longer choose from all possible nodes when selecting the next node to be bypassed. This can be alleviated by sorting all nodes according to their initial key values and using this order to insert them into the heap, starting with the nodes with the smallest key values. Note that this corresponds to the initial node order not using the iterative approach.
A second problem of the iterative contraction is the handling of neighbours of bypassed nodes that have not been inserted into the heap, yet. There two extreme approaches: Inserting them, but this might increase the heap size greatly, especially for aggressive contraction parameters. Not inserting them, but maybe their key values are much smaller now, and thus, they could easily be bypassed, benefiting the whole contraction process. A good compromise between the two approaches is to only insert them into the heap if their new key values are smaller than the largest key value of a node already inserted into the heap.
Preliminary tests have shown that it is beneficial for the performance of the contraction process and the quality of the resulting contraction to not process all nodes of the current batch before inserting the next one but to refill the heap if its size is reduced to about 10% of its initial size.

4.2 Contracted Dijkstra

Speed-up techniques that also apply a graph contraction during their preprocessing step usually only use it as a small part of their more complex approach, e. g. Reach (Sect. 2.3.3), Highway Node Routing (Sect. 2.3.5).
In this section, a speed-up technique is introduced where preprocessing only consists of the graph contraction stated above and whose query is a slightly modified Dijkstra's algorithm. This approach is called the *Contracted Dijkstra (CD) algorithm*. A brief experimental evaluation of the algorithm that shows its good performance is given in Sect. 6.3.2.

Preprocessing

As mentioned above, the preprocessing steps needed by the Contracted Dijkstra algorithm only consist of the graph contraction outlined in the previous section:
At first, a core $G_C = (V_C, E_C)$ is computed for a given graph $G = (V, E)$. Then, both graphs, G and G_C, are merged into the full graph with contraction shortcuts $G_F = (V_F, E_F)$. Subsequently, the *coreNode* flag is set appropriately for every node in G_F indicating whether it was also part of the contracted graph G_C. The same is performed for the edges of G_F with the *coreEdge* flag denoting whether these edges also belonged to the core of the graph.

Query

The query of the Contracted Dijkstra algorithm is performed in a manner, similar to the Highway Nodes Routing query in Sect. 2.3.5. The following two variants are possible, both of them applying the full graph with contraction shortcuts G_F:

Core-Synchronized Variant. This variant is a modification of the bidirectional Dijkstra algorithm and consists of two phases. The first phase tries to reach the core on all paths that possibly contribute to a shortest path. The second phase then continues the search on the core. The precise sequence of operation of the query is described below:
The first phase is initiated by starting a normal bidirectional Dijkstra search from the source and the target. The search in both directions is continued until all entry points into the core have been settled. The *core entry points* of a node u are defined as the set of core-nodes, that can be reached from u without encountering any other core-nodes in between. A distinction is drawn between the *forward entry points* of the source and the *backward entry points* of the target. The search is pruned at core entry nodes, keeping track of the shortest distance to the entry points in each direction. If the source or the target belongs to the core, the search is pruned immediately and they become an entry point themselves. If both are core-nodes the first phase can be skipped entirely. The first phase ends, if either the priority queues of both directions are empty or if the sum of the minimum of the minimal forward priority key and the minimal distance from the source to a forward entry point and the minimum of the respective values in

the backward direction is larger than the current tentative shortest-path distance. In the latter case, a shortest path has been found. Otherwise, phase 2 has to be started.
The second phase is initialized by refilling the priority queues with the forward and backward entry nodes encountered during the first phase, using their distances from the source or target as priority keys. Then, the bidirectional Dijkstra search is resumed with these already filled queues. The search is now restricted to the core of the graph. Thus, only edges belonging to the core will be relaxed when processing the neighbours of a settled node. The search is stopped, if either one of the queries is empty, or if the sum of the minimal priority keys in both queues is larger than the shortest path encountered so far.

Asynchronous Variant. This variant of the Contracted Dijkstra query does not require two distinct phases. It is also initiated by starting a normal bidirectional Dijkstra search. But after encountering a core node, the query just moves on, without pruning this core entry point, much like the aggressive, asynchronous variant of the Highway Node Routing algorithm. After the core has been entered in one direction, the search is continued only within the core itself. It will never leave the core again. Therefore, only those neighbours of a settled core-node will be inserted into the priority queue, that also belong the core. The query can be stopped, when the search spaces of both directions meet.
This approach is much faster than the *core-synchronized variant*, since it does not have to wait until all core entry points have been reached, before continuing the search within the core. This usually proves to be very effective, since the entry points are normally reached in a very asynchronous order by the forward search and the backward search. Unfortunately, when combining the Contracted Dijkstra algorithm with other speed-up techniques, a lot of them require a synchronized start of the search process in the core to function properly (e. g. ALT, ArcFlags). Therefore, this variant only has an academic value.

Discussion

Sketch of the proof of correctness for the claim that the Contracted Dijkstra algorithm is correct:

Consider an arbitrary s-t query using the core-synchronized variant of Contracted Dijkstra. The search ends either after the first phase, or after the second phase. If an s-t path was found, it is supposed to be a shortest path.
Case 1: The query is terminated after the first phase. Thus, at least one path from s to t has been found, the smallest of them being P with a length w(P). This path is a shortest path between s and t, since the bidirectional Dijkstra that was performed is correct [Dan62]. By construction of the query, only edges belonging to the original graph have been used. The stopping condition by Goldberg et al. [GH04] guarantees that no shorter path than P could have been found afterwards. It has been modified to include core entry points by counting them as belonging to their respective priority queue, since the search could still be continued from them.

Case 2: If all shortest paths contain more than one core-node, the query has to enter the second phase, in order to determine one. In this case, a shortest path has the following principle structure w.l.o.g.: $\langle s, \ldots, u, \ldots, v, \ldots, t\rangle$, with u being a forward entry point and v being a backward entry point. The subpath $\langle u, v\rangle$ contains only core-nodes, whereas the other nodes do not belong to the core. Shortest paths found within the core are also correct in the original graph by construction of the core, since shortest-path distances are maintained. Thus, the bidirectional Dijkstra search in phase 2 finds a shortest connection between one of the forward entry points and one of the target entry points, with their initial keys as additional starting weights. This connection is also a subpath $\langle u', v'\rangle$ of a shortest s-t path. If none of the shortest paths between the source and the entry point u' contains another entry point u, u' has already been settled during phase 1 and equals u. Otherwise, the search was pruned at u and a shortest path between u and u' must be found during the second phase. If $\langle u', v'\rangle$ is a shortest path from the forward entry points to the backward entry points in the core with regards to their initial keys, one path $\langle u, u'\rangle$ must have also been settled during this query. The same holds true for v' and the target. Thus, a shortest s-t path has been found and Contracted Dijkstra is correct indeed.

The search space of a Contracted Dijkstra query on the roadmap of the Netherlands is shown in Fig. 4.2, to the right. It can be compared to the search space of the same query performed by the bidirectional Dijkstra algorithm, shown on the left side. At first glance, both seem to be very similar, but upon taking a closer look, one can see that the structure of the Contracted Dijkstra's search space is much courser than the one of the bidirectional Dijkstra. Whereas the latter algorithm has to settle over 10 000 nodes, the former only needs to check about 4 000. This disparity is mainly due to the fact that, instead of having to settle every single node on a path from u to v as the bidirectional Dijkstra has to, only one shortcut (u, v) has to be settled by the Contracted Dijkstra instead.

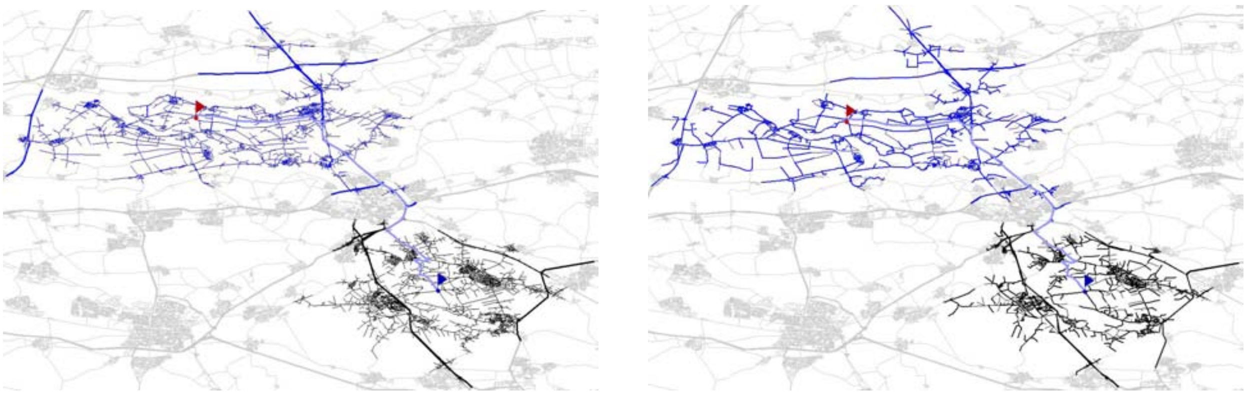

(a) bidirectional Dijkstra **(b)** Contracted Dijkstra

Figure 4.2: *Comparison of the search spaces of the bidirectional Dijkstra and the Contracted Dijkstra. A local query on the road map of the Netherlands is depicted. Contraction parameters c=0.5 and h=10 have been used for building the core. This reduced the original graph size from 893 042 nodes and 2 279 606 edges to 358 498 nodes and 1 143 006 edges in the core, adding 250 509 new shortcuts.*

The Contracted Dijkstra algorithm can be easily adapted for dynamic graphs. This fact renders the algorithm very interesting for further analysis. Especially combinations with other speed-up techniques that can also be easily adjusted for the use with dynamic graphs might be promising. The CALT algorithm presented in the next section is such a combination.

Optimizations

Early Pruning. This strategy was originally introduced by Goldberg et al. in [GKW07]. In order to reduce the query times of Contracted Dijkstra, the edges at each node are sorted according to their *coreEdge* flag, starting with set flags. Thus, when relaxing the edges of a node, the ones belonging to the core are processed first. If the search has already entered the core, relaxing the outgoing edges of a settled node can be stopped as soon as the first edge is encountered, that does not belong to the core. All edges that would be touched afterwards do not belong to the core and thus can be safely omitted. The same strategy can also be applied before entering the core by relaxing the edges in the reverse order, starting with edges not belonging to the core. Note that the target ID is usually used as secondary sorting criterion.

Locality Effects. The caching mechanism of the CPU can also be exploited to speed-up queries. In order to utilize this so-called *locality effect*, nodes that will probably be accessed in short succession have to be stored close to each other in memory. Then, if one node is accessed, the other ones near it will also be loaded into the cache, improving their subsequent access times. Regarding the *Contracted Dijkstra* algorithm, sorting and storing the nodes of the graph according to their *coreNode* flag as primary sorting criterion and their index as secondary sorting criterion seems to be most favourable. Thus, all nodes belonging to the core will be stored together in one large area and the nodes not belonging to the core will also be stored together in another area. Since nodes with a similar index are usually also close to each other in the graph, the secondary sorting criterion guarantees that the nodes within on of these two areas will also be stored in an appropriate order. Exploiting this locality effect leads to speed-ups of about $10-20\%$ for the inputs used.

Search Graph. This approach is taken from Highway Node Routing [Sch07]: All edges that are not part of any shortest path tree built by a Contracted Dijkstra query can be removed from the graph with no negative impact on the query. In particular, all edges leaving the core can be deleted. This simplifies the query, since checks for edges leaving the core no longer have to be performed. Also, fewer edges are touched in general, accelerating the query further. As an additional side-effect, the graph gets smaller and thus requires less memory. Note however that this reduction might be disadvantageous, if the Contracted Dijkstra algorithm is combined with other shortest-path techniques. Mostly, at least their preprocessing routines would have to be adapted to be aware of the missing edges. Alternatively, the edges could be added to the graph again for the preprocessing step.

4.3 CALT

The *Contracted Dijkstra* speed-up technique can be easily combined with almost any other technique. An obvious approach would be to use the small hierarchy provided by the contraction and to apply a second, preferably goal-directed, method only on the core of the graph. Thus, this method is only used during the second phase of the query. Since the core is usually much smaller than the original graph, the additional preprocessing time and space overhead, needed by the new method, will be a lot smaller than if it would be performed on the full graph. Note that the added speed-up technique will not lose much of its performance compared to an execution on the original graph since the entry-points into the core are usually reached very fast (after about 13 hops for normal contraction parameters, see Tab. 6.16). The performance of the added technique might even profit of the additional shortcuts inserted by the contraction step. Thus, the combination becomes more than just the sum of its parts.
In principal, any of the established speed-up techniques (ALT, ArcFlags, Reach, . . .) can be combined with the Contracted Dijkstra. The ALT algorithm has been chosen, since it profits the most from a reduced memory overhead. It is also a very robust technique, yielding good speed-ups on most types of graphs. Furthermore, both the Contracted Dijkstra and the ALT algorithm can be easily modified for dynamic graphs and probably also time-dependant graphs. An adaptation of the ALT algorithm for dynamic graphs has been done by Bauer in [Bau06]. The combination of the hierarchical Contracted Dijkstra algorithm and the goal-directed ALT algorithm is called *Contracted ALT (CALT)* algorithm and is explained in this section.

Preprocessing

The CALT algorithm requires two separate preprocessing steps, one for each speed-up technique involved. At first, the preprocessing for the Contracted Dijkstra has to be performed, as shown in the last section, building the core G_C and the full graph with contraction shortcuts G_F of the original graph G. Then, the contracted graph G_C is used as input for the preprocessing routines of the ALT algorithm (see Sect. 2.3.2). Landmarks are chosen from within the core and distances from and to all core-nodes and every landmark are computed.

Performing the ALT preprocessing only on the core G_C instead of on the full graph has several advantages. For example, computing landmark distances gets much faster since the applied graph is a lot smaller, even considering the added shortcuts. The space consumption of the query also decreases since landmark distances only have to be stored for core nodes. And, as shown for REAL and Highway Hierarchies* (see Chap. 3), selecting landmarks also gets faster since fewer nodes have to be considered. Furthermore, the impact on quality of the preprocessing is apparently minimal. In particular, since now, more sophisticated landmark selection strategies can be applied and more landmarks can be stored to improve the performance of the query without increasing the preprocessing times and the memory overhead too much.

Proxy Nodes

The ALT algorithm has to compute lower bounds on distances $d(s,v)$ from the source to nodes in the core of the graph and lower bounds on distances $d(u,t)$ from core-nodes to the target. For this, distances between the source and target and the landmarks are required. But since these distances are only computed for core-nodes, they are usually not available for the source and target. Therefore, *proxy-nodes* are introduced, similar to the approach Goldberg et al. used for reach-aware landmarks in [GKW07]:
A proxy-node $s' \in V_C$ of the source $s \in V_F \setminus V_C$ is defined to be the node in the core with a minimal distance from the source. In principal there are two choices for the proxy-node, one with $d(s,s')$ minimal and one with $d(s',s)$ minimal. As shown below, knowing only one of them is sufficient to compute the lower bounds. Landmark distances from and to s' can be used together with the distance $d(s',s)$ between the source and its proxy to compute lower bounds on the distance $d(s,v)$ from the source s to an arbitrary node $v \in V_C$ in the core (see Fig. 4.3).

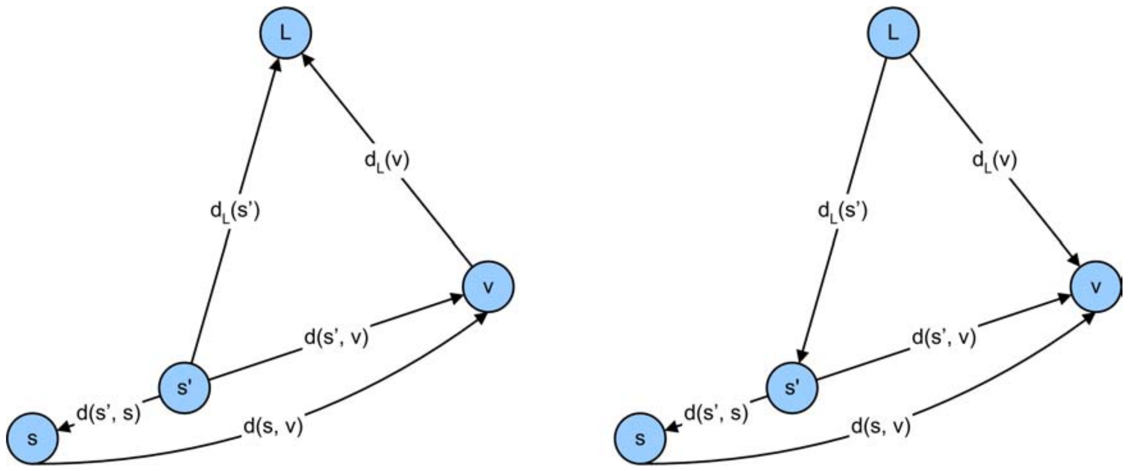


Figure 4.3: *Illustration of the relations used to obtain lower bounds on $d(s,v)$, if a source-proxy s' is present, using distances to (left figure) and from (right figure) a landmark L in the core of the graph (confer to Fig. 4.4).*

The resulting inequations to compute the lower bound on $d(s,v)$ are:

$$\begin{array}{ll} d(s',v) + d_L(v) \geq d_L(s') & \text{, triangle inequation in the core} \\ d(s',s) + d(s,v) \geq d(s',v) & \text{, new triangle inequation to obtain } d(s,v) \\ \Rightarrow d(s,v) \geq d_L(s') - d_L(v) - d(s',s) & \text{, resulting lower bound} \end{array}$$

with $d_L(\cdot)$ denoting distances to the landmark L; and

$$\begin{array}{ll} d_L(s') + d(s',v) \geq d_L(v) & \text{, triangle inequation in the core} \\ d(s',s) + d(s,v) \geq d(s',v) & \text{, new triangle inequation to obtain } d(s,v) \\ \Rightarrow d(s,v) \geq d_L(v) - d_L(s') - d(s',s) & \text{, resulting lower bound} \end{array}$$

with $d_L(\cdot)$ denoting distances from the landmark L.

The lower bounds yield a feasible potential, as proven in App. B. The maximum of both bounds can be used to obtain a better lower bound. Note, that in both cases only the distance $d(s', s)$ from the proxy to the source node is required to compute the lower bounds on $d(s, v)$.

Lower bounds on distances from a core-node $u \in V_C$ to the target $t \in V_F \setminus V_C$ are computed accordingly. Here, distances from the target to the proxy $d(t, t')$ are required, as seen in Fig. 4.4:

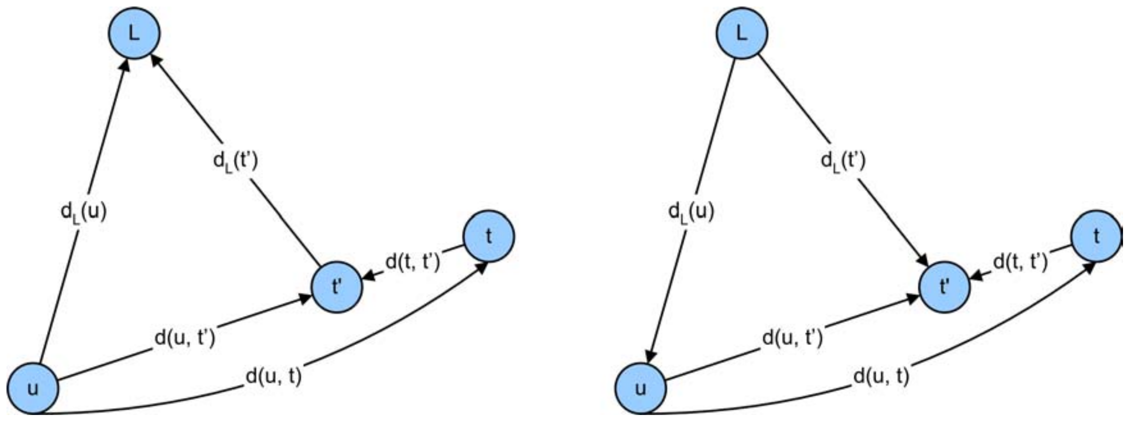


Figure 4.4: *Illustration of the relations used to obtain lower bounds on $d(u, t)$, if a target-proxy t' is present, similar to Fig. 4.3, using distances to (left figure) and from (right figure) a landmark L in the core of the graph.*

The resulting inequations to compute the lower bound on $d(u, t)$ are:

$$d(u, t) \geq d_L(u) - d_L(t') - d(t, t') \text{ , using distances to L}$$
$$d(u, t) \geq d_L(t') - d_L(u) - d(t, t') \text{ , using distances from L}$$

The source-proxy s' can be computed by a normal Dijkstra search in the reverse graph $\overline{G_F}$, starting from the source s. The query is terminated, as soon as the first node u belonging to the core is settled. The node u becomes the source-proxy s' and the distance $d(s', s)$ gets saved. The same is performed accordingly on G_F to obtain the target-proxy t' and its proxy-distance $d(t, t')$.
Proxies have only to be computed for nodes not belonging to the core. This can be performed for all of them as an additional preprocessing step. Alternatively, the proxies of the source and target can be computed during the query. Since the core is reached with only few hops for reasonable contraction parameters, the two additional Dijkstra searches require little extra time, compared to the total running time of the query. Furthermore, by computing the proxy-nodes on-demand, they and their distances do not have to be stored for all non-core nodes, which would produce a large memory-overhead.
There are graphs that do not contain a path from the core to the source node or from the target to the core. In this case, a dummy node with distances to and from all landmarks set to zero is used as proxy, with the proxy-distance also set to zero. This yields acceptable lower bounds, if only one dummy node has to be used. But if both proxies cannot be determined, the search will be reduced to a normal Dijkstra search.

Query

The query of the CALT algorithm is based on the core-synchronized variant of the Contracted Dijkstra speed-up technique with some additions and alterations concerning the ALT algorithm in the second phase of the query:

The first phase of the CALT query is performed in exactly the same way as for the Contracted Dijkstra, described above. It is terminated if both queues are empty, which denotes that the graph has been searched exhaustively up to all entry points and that the query has to continue in the core. It is also stopped, if the sum of the minimum between the minimal forward priority key and the minimal distance from the source to a forward entry point and the minimum of the respective values in the backward direction is larger than the current tentative shortest-path distance. This implies, that a shortest path has been found and that the query does not need to be resumed.
If a shortest path has not been found in the first phase, the search is continued on core of the graph in the second phase. It has to be initialized in a more complex manner than phase 2 of the Contracted Dijkstra algorithm: At first, the proxy nodes of the source and target have to be computed. Then, the queues are rebuilt, using the ALT potential functions as key values. Depending on the particular landmark management, further tasks have to be performed. The subsequent query only considers nodes and edges belonging to the core of the graph. Otherwise, it is performed just like a normal bidirectional ALT query, albeit using proxies for the source and target, when needed. The search terminates if the priority queue in one direction is empty or if the sum of the minimal priority keys in both queues is larger than shortest path found so far, modified by the potentials. This is the same stopping condition as of the ALT algorithm.

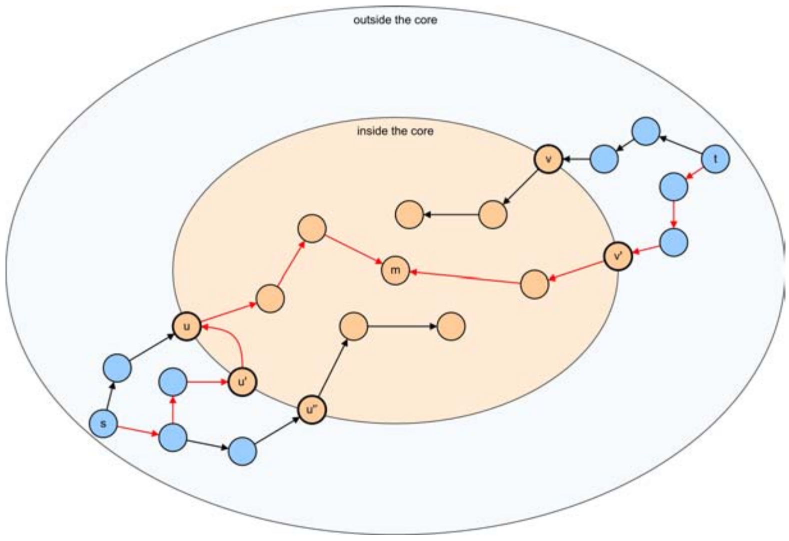


Figure 4.5: *Basic structure of a search space on contracted graphs, as generated by the Contracted Dijkstra or the CALT algorithm. Areas of the graph within and out of the core are differentiated by their respective colours, core entry points are drawn in bold. The meeting node of the search spaces in both directions is labeled with m.*

Discussion

The CALT algorithm finds correct shortest path just as the plain Contracted Dijkstra does. The basic reasonings stay the same as in Sect. 4.2. Only, instead of applying the correctness of the bidirectional Dijkstra for the search within the core, the correctness of the bidirectional ALT algorithm is used.

In Fig. 4.6 the search spaces generated by the bidirectional ALT and the CALT algorithm are shown, using the same query as in Fig. 4.2. Comparing both search spaces, the same principal characteristics as before stand out: The search space of the contract query looks similar to the one of the normal query, only much courser. This is again due to the fact, that the CALT algorithm can use shortcuts to hop between nodes and does not have to check every outgoing edge in-between.

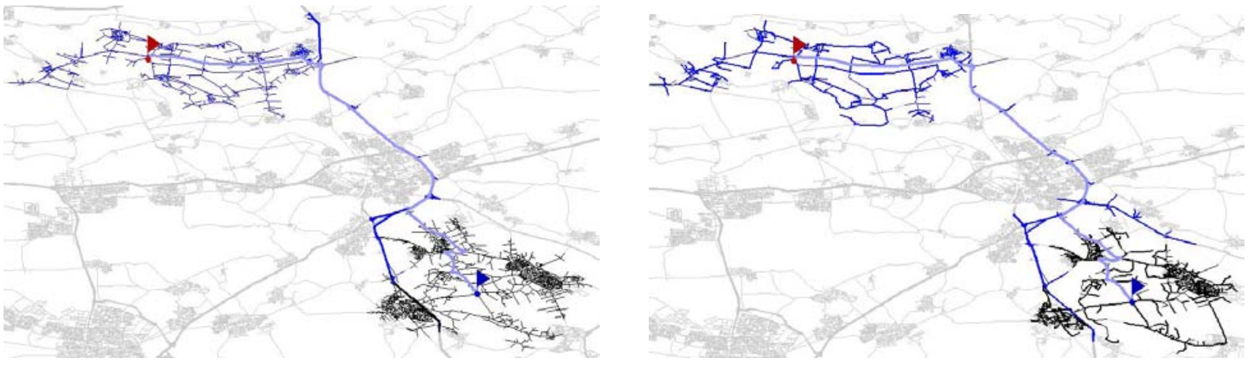

(a) bidirectional ALT **(b)** CALT

Figure 4.6: *Comparison of the search spaces of the bidirectional ALT and the CALT algorithm, using the same query as in Fig. 4.2. The bidirectional ALT has to settle 4673 nodes, the CALT only needs 1818. Both queries have been performed using 16 maxCover landmarks and contraction parameters c=0.5 and h=10 have been used.*

The Contracted ALT is similar to Highway Hierarchies* (see Sect. 3.3) in many respects. Both of them are based on a hierarchical query, Contracted Dijkstra (Sect. 4.2) or, respectively, Highway Hierarchies (Sect. 1.1) and both apply the goal-directed ALT to improve the query. Whereas Contracted Dijkstra only offers a very basic hierarchical structure with just two levels, Highway Hierarchies applies a much more complex structure. But because of its simplicity Contracted Dijkstra profits more from the addition of ALT than does Highway Hierarchies. In particular, the CALT query can apply a more simple stopping condition than Highway Hierarchies*, but nevertheless CALT stay slower than even normal Highway Hierarchies. Then again, CALT requires much less additional memory than Highway Hierarchies* and an adaptation to dynamic graphs should also be feasible more easily.
Note that the memory optimization strategy of Highway Hierarchies* is the basic case for CALT. Here, landmark distances are only stored for some higher level of the hierarchy. In this case, the query algorithms of both techniques also become quite similar with two phases, the first applying the basic search algorithm to find entry points into the core and the second also applying ALT.

Optimizations

The same basic optimization strategies that have been used for the graph contraction and the *Contracted Dijkstra* algorithm can also be applied to the *CALT* algorithm. In addition, all improvements of the ALT algorithm can be used to enhance the second phase of the CALT query. There are also optimizations unique to the CALT algorithm, coming from synergy effects between the individual techniques.

Landmarks. As already mentioned before, the number of landmarks can be increased from typically 16 to 64, 128 or even more, without generating too much overhead, since only distances to and from nodes belonging to the core have to be stored. Furthermore, the superior maxCover landmark selection strategy can also be used on larger graphs instead of the avoid algorithm, since only the core has to be processed and thus execution times will stay small.

Active Landmarks. Landmarks can be managed by an *active landmarks* approach as shown in [GW05]: At the beginning of the query, only two landmarks are used, one that provides the best lower bounds at that time, using distances from landmarks and one that uses distances to landmarks. During the query the number of active landmarks is increased up to a certain limit. Adding new landmarks is expensive, since the potentials of the nodes change and both priority queues have to be rebuilt. Therefore, new landmarks are only added after the query has proceeded at least certain distance since the last activation of a landmark. Additionally, a certain minimum number of nodes have to be settled before trying to add an additional landmark. These conditions are checked separately for both search directions.

Proxy Computation. The computation of the source and target proxy nodes can also be moved in front of the first phase. If source or target is encountered during one of the Dijkstra searches needed to determine the proxies, the search can already be aborted before it actually began. If not, the search probably has to enter the core and the proxies would have been computed anyways. It has to be decided from case to case, which approach will be more useful.

Memory Usage. Usually, additional mapping information has to be stored to convert node IDs in the full graph to node IDs in the contracted graph for accessing the correct landmark distance information. If the distances are stored at each node, this overhead can be saved on. Alternatively, if the locality optimization is used, the core nodes are all stored together. Thus, by sorting the landmark information in the same order, the same index can be used to access the nodes and its landmark information, also rendering the mapping information superfluous.

ArcFlags & Hierarchies

The combination of the powerful goal-directed ArcFlags algorithm with hierarchical speed-up techniques promises to yield interesting results. In particular, a fair improvement in query times is expected compared to the single algorithms. By restricting the use of ArcFlags to only some subgraph of the hierarchy, its preprocessing times and space consumption should also decrease. At first, this so-called *Partial ArcFlags* approach is explained. Then, two combinations with hierarchical techniques are discussed, Reach-aware and Hierarchy-aware ArcFlags that are based on Reach and Highway Nodes Routing, respectively.

5.1 Partial ArcFlags

Both, REAL and Highway Hierarchies* (see Chap. 3), add the goal-directed ALT to hierarchical techniques with great success. With ArcFlags being another potent goal-directed technique, applying it instead of ALT is an obvious choice for further studies. The benefits of ArcFlags are an excellent sense of direction and the ease of use. These should be exploited as much as possible to enhance the hierarchical techniques. In turn, the drawbacks of ArcFlags, long preprocessing times and a large memory overhead, should be compensated as much as possible by the other techniques. There are two approaches to deal with the shortcomings of ArcFlags that both have already been used, in principal, for the combinations of ALT with hierarchical techniques: Either the parameter values, i. e. the number of regions, can be decreased, or the preprocessing is only done for a certain subgraph. Usually, both approaches encompass a decline in performance, regarding ArcFlags alone. The former approach has already been applied for AALT (see Sect. 3.1). The latter approach is denoted *Partial ArcFlags technique*. It is suited especially well for a combination with hierarchical speed-up techniques since they already provide subgraphs that can be used. The requirements on the subgraph are that it has to be connected, and that no shortest path has to enter the subgraph more than once. Such subgraphs are called *cores*, subsequently. The Reach-aware and Hierarchy-aware ArcFlags algorithms, described later in this chapter, apply the Partial ArcFlags technique to improve the performance of the Reach algorithm and Highway Nodes Routing, respectively.
Note that the Partial ArcFlags technique tries to retain the preprocessing routines of the normal ArcFlags algorithm (partitioning, computation of ArcFlags) and the functionality of the query, even if ArcFlags are only used on a subgraph. Important particularities are described below.

Partitioning

As described in Sect. 2.3.4, there are many different techniques to partition a graph. For the studied graphs, the best results are usually obtained with the multi-way arc separator algorithm. Currently, there are three free implementations of this algorithm: METIS [Lab07], PARTY [MS04] and SCOTCH [Pel07]. The METIS implementation has been used for a long time, but it has two distinct disadvantages: Regions are not always connected and the number of boundary nodes is quite high. The PARTY implementation guarantees the connectivity of the regions, but the number of boundary nodes even increases. SCOTCH does not always produce connected regions, but it has the lowest number of boundary nodes. Since this value is directly related to the expected preprocessing times of the ArcFlags algorithm, the SCOTCH partitioning algorithm is subsequently applied.

To improve the quality of the partition, a *local optimization run* is performed once the initial partition has been determined. In a first step, each node u of the graph is analyzed. If there are more of its neighbours in a single different region than there are neighbours belonging to the same region as u, it is shifted to that region. All of the neighbours of u that have already been checked are requeued to be analyzed again. This usually decreases the number of boundary nodes, overall. After this step, regions that are not connected are determined. It is assumed that such regions only consist of two separate components. The smaller one is then added to an adjacent region. The region featuring the longest common border with this component is chosen, since thus, the number of boundary nodes is reduced the most. Note that on small graphs with a large number of regions a local optimization run could eliminate single regions.

Computation of ArcFlags

After a partition has been found, ArcFlags can be computed using one of the techniques described in Sect. 2.3.4. In particular, there are currently two viable variants for this task: the *bounding nodes preprocessing* [HKMS06] and the *Centralized ArcFlags preprocessing* [Gei08]. The latter one is usually much faster but also requires a lot more memory, especially for regions with a lot of boundary nodes. This is of little concern for the Partial ArcFlags technique since the cores, for which ArcFlags are computed, are usually much smaller than the full graph. Therefore, the Centralized ArcFlags preprocessing is applied, subsequently.

If a graph contains a pair of nodes, for which there exists more than one shortest path, special precautions have to be taken. It has to be guaranteed that both individual speed-up techniques do not interfere with each other by trying to find a different shortest path. In the worst case, there is found none at all if the necessary edges are pruned by the respective other technique. An easy solution is to flag all shortest paths from a node into a region. This usually decreases the performance only slightly, since for sparse graphs there are not a lot of different shortest paths. Note that the Centralized ArcFlags preprocessing is already implemented in this way.

To further improve the space consumption of the Partial ArcFlags approach, an *ArcFlags compression* can be performed [Hil07]. Here, each unique ArcFlag label is saved once in a look-up table. Then, the edges only have to hold an index to this table. This usually reduces the memory overhead considerably at the cost of only a slight decrease in query performance.

Query

Since Partial ArcFlags are only available for a certain subgraph, they cannot simply be added to an existing speed-up technique as an additional pruning step. At first, the technique has to be modified: Its query is initially performed as usual. As soon as no further nodes not belonging to the core of the graph will be touched, the ArcFlags-pruning can be enabled. This is e. g. done similarly to Contracted Dijkstra in Sect. 4.2 by stopping the search at nodes belonging to the core, so-called *entry nodes*, until all of them have been found in both directions, before continuing the search from them. Thus, the search in the core is essentially a multi-source, multi-target query, seeking a shortest path from one of the forward entry points to one of the backward entry points with regards to their distance from the source or, respectively, the target. Since the source and the target usually do not belong to the core, they cannot provide region information for the pruning step. Thus, the regions of all entry points in the forward direction are taken as source regions, respectively for the target regions. If not all of these regions are used or not all entry points are found, the shortest paths might be missed, as illustrated in Fig. 5.1.
Note that the Partial ArcFlags approach requires an underlying bidirectional query. Only in this way, the regions needed for the ArcFlags-pruning can be determined without further difficulties.

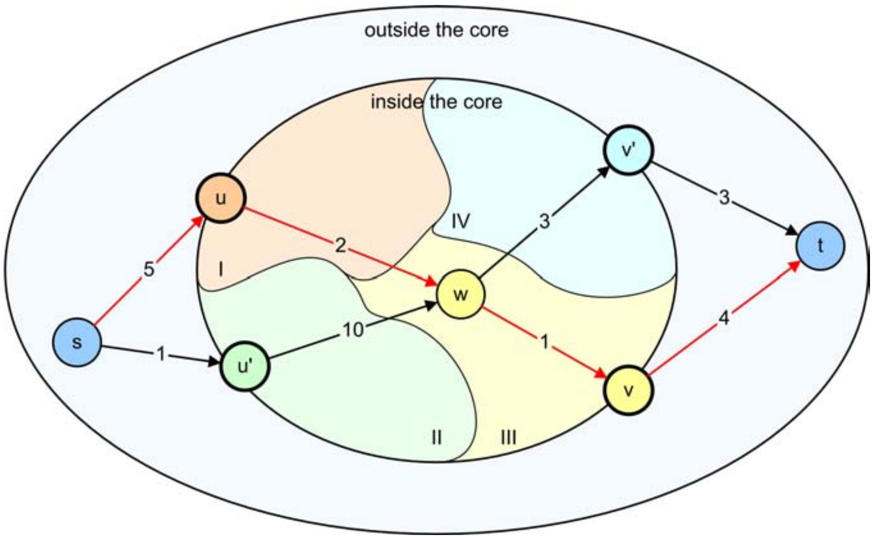


Figure 5.1: *Graph with a highlighted core. The core itself is partitioned into four regions, denoted by different colours. The shortest path from* s *to* t *is drawn in red. Entry points are drawn in bold. If e. g. the region of* u *would not be used, the backward search could not be directed into region I coming from node* w *and thus, it would not find the shortest path. Here, the forward search still yields the correct shortest path, but relying on only one search direction is not always an option and also defeats the purpose of using a bidirectional query.*

5.2 Reach-Aware ArcFlags (ReachFlags)

The *Reach-aware ArcFlags (ReachFlags) shortest-path technique* combines the hierarchical information of the graph provided by the reach values with the goal-direction of the ArcFlags. The latter technique is incorporated into the Reach algorithm according to the Partial ArcFlags approach introduced in the last chapter, applying ArcFlags only to the higher levels of the graph-hierarchy, i. e. to the subgraph with high reach values.

Preprocessing

The additional information required by the Reach-aware ArcFlags algorithm is provided by two separate preprocessing algorithms. At first, node-reach values are computed for the whole graph. Here, shortcuts are also added to the graph. Then, ArcFlags are computed for a certain subgraph, defined by the reach values.
The Reach preprocessing is performed with the approximate bounding algorithm introduced by Gutman in [Gut04] and improved later by Goldberg et al. in [GKW06a]. This technique computes reach values for edges by iteratively bounding them up to a threshold ε that gets increased in each iteration. The initial value of ε is determined as outlined by Goldberg et al. Edges with bound reaches are removed at the end of each iteration step. At the beginning of the next step, additional shortcuts are incorporated into the graph. After the preprocessing ends, the edge-reaches are converted into node-reaches. The original preprocessing algorithm is modified in the following way: Instead of always computing an initial threshold with a random component, an average threshold value is determined beforehand for each graph and used subsequently to gain results that are easier to compare. Also, the preprocessing is only performed for a certain number of iterations before it is stopped. The shortcuts that would be computed at the beginning of the next step are also included into the graph. After the edge-reach values are converted to node-reach values, all of them that have not been bound yet, i. e. that are equal to or above the current threshold value ε, are set to infinity, effectively removing them. The refinement step introduced by Goldberg et al. is omitted if the preprocessing is terminated prematurely.
The obtained reach values define a hierarchy on the graph with larger reach values denoting higher levels of the hierarchy. The core of the graph is induced by the nodes, featuring a reach value equal to or above the threshold value ε of the final iteration of the Reach preprocessing, and their adjacent original edges and shortcuts.
The core defined by the reach values is subsequently used for the ArcFlags preprocessing. It is partitioned according to one of the approaches outlined above. Then, the Centralized ArcFlags preprocessing algorithm by Hilger [Hil07] is used to compute. It computes ArcFlags by growing shortest paths from all boundary nodes of a region at once, taking advantage of identical subgraphs. More information about this technique can be found e. g. in Sect. 2.3.4.

Note that the whole graph with all of the additional shortcuts is applied for the query algorithm.

Query

As outlined in the last section, a query algorithm applying Partial ArcFlags consists of two phases. Here, the query of the Reach-aware ArcFlags algorithm is based on a bidirectional Dijkstra with reach-pruning. During the first phase, only this base algorithm is performed. When the second phase starts, pruning according to the available ArcFlags is applied, too. The actual implementation of this two-tiered query is a bit simpler than for the Contracted Dijkstra algorithm in the last section. In detail, an s-t shortest-path query is performed as follows:

In an initialization step, the edges of each node are rearranged according to the node-reach value of their respective targets, sorting larger values to the front. This is done to apply *early pruning* during the query. Afterwards, the search is commenced. It is performed as a normal bidirectional Dijkstra query with reach-pruning and also applies early pruning as explained in Sect. 2.3.3. If a node u is settled and its reach value is smaller than its key value key(u), the search pruned. Otherwise, its edges are processed. If the reach value of the target v of one of these edges is smaller than $\text{key}(u) + w(u, v)$ the search is pruned at that edge. If it is even smaller than key(u), the remaining edges are also skipped. In each step, the search direction with the minimal priority key value is chosen. The first phase is terminated if one of three conditions is true: Either one the priority queue has to be empty, or the sum of the minimal elements in both queues has to be equal to or larger than the current tentative shortest path, or the smallest priority key over both queues has to be equal to or larger than the final value of the threshold ε during the preprocessing. In the first two cases, the whole search is stopped since they represent the stopping condition of the normal Reach query. Otherwise, the search is continued with the second phase, since now, all remaining nodes belong to the core. Note that because of the way the search directions are alternated, no core-node is settled beforehand.
The second phase is initialized simply by noting the regions of all nodes remaining in the forward queue, which become the set of source regions and respectively for the target regions. The search is then continued as before but with ArcFlags- pruning in addition to the reach-pruning. When considering the ArcFlags of an edge, the search is only pruned at that edge if no flag is set for any of the possible target regions in the forward direction, respectively for the other search direction. The search terminates either if one of the priority queues is empty, or if the sum of the minimal keys in both queues is equal or larger than the current tentative shortest path. Then, the search yields a shortest path between s and t if any such path exists.

Note that the reach-pruning during the second phase is only required in order to restrain the search to core-nodes. Here, comparing reach values to infinity is sufficient since all core-nodes have a reach value of infinity according to the preprocessing. Thus, if a reach value is less than infinity, the search can be pruned at that edge or that node. Also, the condition for switching to the second phase could be simplified accordingly to a comparison to infinity instead of to the final value of threshold ε during the preprocessing.

Further Optimizations

In principal, most of the optimization techniques used for the Reach algorithm and for ArcFlags alone can also be used for Reach-aware ArcFlags either directly, or at least after adapting them slightly for this speed-up technique. There are also some further optimization ideas that require both individual algorithms. The following techniques seem to be particularly noteworthy.

Locality Effects. The nodes of the graph can be rearranged so that nodes that probably are accessed consecutively are stored near each other in memory. In this way, *cache locality effects* can be exploited to speed-up the access times, and thus, the whole query. For Reach-aware ArcFlags, it is suitable to group nodes that belong to the core and nodes not belonging to it, separating low-reach nodes from high-reach ones. The initial order of the nodes is retained within both areas in case that they have already been ordered to profit from locality effects.

Non-Core ArcFlags. If requests for ArcFlags of edges not belonging to the core are adapted to always return an empty ArcFlag label, i. e. all regions are not set, reach-pruning can be omitted entirely during the second phase of the query. This is due its primary function, keeping the search within the core, being taken over and done implicitly by the ArcFlags-pruning. This simplifies and possibly speeds-up the query. But note that further speed-up techniques like the early pruning have to be adapted accordingly.

Full Reach Processing. Instead of stopping the Reach preprocessing after several iterations, it can also be done in full. This requires more time and memory but also yields additional shortcuts and reach values, of which the query might profit. The core is still defined by a certain threshold on the node-reach values and ArcFlags are only computed for this subgraph. But now, actual reach values are available during the second phase of the query and can be used to prune search.

Space Consumption. In order to reduce the space consumption of Reach-aware ArcFlags, several optimizations can be applied. Obviously, an ArcFlags compression or even a reach compression can be used to reduce the memory overhead. In addition, if reach values are only used in the first phase, their reserved space can be used to store the node-regions in the core. Only an additional flag is required to differ between core-nodes and non-core nodes. For the typically small core-sizes used, the additional memory should decrease by about 1 to 2 bytes/node.

Edge-Reaches. Instead of using node-reach values, edge-reach values can be applied during the query. They are readily available since the preprocessing algorithm actually computes edge-reach values. They give better bounds, and thus, improve the query performance. But they also require more memory since they have to be stored for all edges and in both search directions. Note that the early pruning strategy can only be easily applied for one search direction. The required edge-sorting either can be done for the forward reach values, or for the backward reach values. Thus, the effective speed-up of using edge-reach values gets less pronounced.

5.3 Hierarchy-Aware ArcFlags (HiFlags)

The *Hierarchy-aware ArcFlags (HiFlags) algorithm* is a combination of Highway Node Routing and ArcFlags. The former technique is used as the basis of the algorithm with ArcFlags being added to some higher level L of the hierarchy according to the Partial ArcFlags approach. The hierarchical information required by Highway Node Routing as well as additional shortcuts are provided by the recent *Contraction Hierarchies algorithm* by Geisberger [Gei08].

Preprocessing

The HiFlags algorithm requires two interdependent preprocessing steps. At first, the hierarchy of the graph has to be determined, followed by the computation of the ArcFlags for some level of that hierarchy:
In the first step, hierarchical levels are computed and shortcuts are added to the graph G. This is currently done by the *Contraction Hierarchies algorithm* [Gei08]. Basically, a unique level is determined for each node. Then, the nodes are sorted according to this *hierarchy level* and subsequently removed from the graph, starting at lower levels. When a node is removed, shortcuts are added to ensure that distances between the remaining nodes are maintained. Note that the initially determined node-levels may change during the preprocessing. After all nodes have been removed, the complete hierarchical structure of the graph is given by the node-levels and the shortcuts that have been computed during this process.
For the second step of the preprocessing, the full graph with shortcuts G_F is reduced by removing all nodes and their adjacent edges with a hierarchy level below a certain threshold value L. The resulting subgraph is called *level-L core* of the graph. This core is partitioned into regions and ArcFlags are computed for it. Here, the *Centralized ArcFlags algorithm* by Hilger [Hil07] is applied. Recall that the shortest path trees of all boundary nodes are computed at once for each region and synergy effects due to identical subgraphs are exploited to speed-up the preprocessing. If there is more than one shortest path from a node into a certain region, all paths into that region are flagged. This is important with regards to the canonical shortest paths of Highway Node Routing.

With Contraction Hierarchies having as many hierarchy levels as nodes, it is inconvenient to denote the core by the threshold level. Instead, the percentage of nodes remaining in the core is also used to label the core-size. For example, a core-size of 5.0% implies that the core encompasses the 5.0% of nodes with the highest level.

Regarding the actual implementation for the query, G_F is reduced to a search graph by removing all edges leading to a lower level of the hierarchy (see Sect. 2.3.5). This improves the query performance since fewer edges have to be touched and also decreases the space consumption. In addition, ArcFlags also do not have to be saved for these removed edges further decreasing the memory usage. Note that the full graph is still needed to compute the ArcFlags.

Query

The query of Hierarchy-aware ArcFlags is based on the query of Highway Node Routing, which is described more closely in Sect. 2.3.5. Its query algorithm is modified to incorporate the two-tiered approach of Partial ArcFlags: During the first phase of the query, only Highway Node Routing is applied. Then, the second phase adds ArcFlags information. A detailed sequence of operation for this single-source, single-target query is given below:

When the query is started, a normal bidirectional Highway Node Routing search is performed. After each step, the search is alternated between the forward and the backward direction. The search is pruned at nodes with a hierarchy level of L or above. These nodes are called *entry nodes* and their distances from the source or the target are called *entry distances*. They are stored to be used later. The search in one direction is stopped either if the respective priority queue is empty, or if the minimum of the key values in that queue and the entry distances in that direction is equal to or larger than the shortest path found so far. The first phase ends when the search in both directions has been stopped. The whole search can be stopped after the first phase either if no entry points have been found in one direction, or if the tentative shortest-path distance is smaller than all entry distances and all key values remaining in the queues. In the latter case, a shortest path has been found since the remaining nodes in the priority queues and entry points, from which the search could be continued, have a distance from the source or the target, longer than the currently shortest path. Thus, they cannot be part of a shorter path. In the former case, no shortest path exists since if one still exists, it would have to be routed over the core of the graph, which is not entered in at least one direction.
If the search is not terminated after the first phase, the second phase is initialized. The remaining elements in both priority queues are flushed and the entry points that have been stored previously are reinserted into their respective queues. The regions of these entry points are noted separately for each direction to be used as a set of source or target regions, later. After the initialization, the search is continued on the core of the graph as before, but with ArcFlags-pruning enabled: When touching an edge, its ArcFlags are checked. If no flag is set for at least one of the source or target regions in the respective direction, the search is pruned. Note that the search never leaves the core again. Thus, ArcFlags are always available during the second phase of the query. The search terminates if one of the two following conditions is met for each search direction: Either its priority queue is empty, or the distance of the shortest path found so far is smaller than the smallest key in that queue. If any path exists between the source and the target, a shortest one has been found.

Note that the source and the target of a query might already have a hierarchy level of L or above. In this case, they already belong to the core of the graph. If this is true for both nodes, the query directly starts with the second phase, only using their regions for the ArcFlags-pruning in the respective direction. If it is true for only one of them, the search in that direction is directly pruned at the start and only the search in the other direction is pursued further.

Discussion

A comparison of the search spaces of Hierarchy-aware ArcFlags and of its two constituting speed-up techniques, ArcFlags and Highway Node Routing, is shown in Fig. 5.2. For each algorithm, the same query is performed on the road network of the Netherlands and the resulting search spaces are visualized:

Note that although the visualization of the search space of the ArcFlags query appears to be almost optimal, about twice as much nodes have to be settled compared to Highway Node Routing. This is due to the shortcuts introduced by Highway Node Routing. They bypass larger parts of the graph and are drawn as long paths. Thus, the visualization of the search space of Highway Node Routing appears to be much larger than it actually is.
Near the source and target, the search space of the Hierarchy-aware ArcFlags query is identical to the one of Highway Node Routing. Further away, where the core has been entered, the ArcFlags dominate the query and the search space looks more like the one of the ArcFlags query. Since the HiFlags technique also applies shortcuts, the ArcFlags component of the algorithm can also profit from them and the number of settled nodes decreases even further.

Further Optimizations

The principal optimization strategies used for ArcFlags can also be applied to Hierarchy-aware ArcFlags. The same is true for Highway Node Routing. Further optimizations, especially regarding synergy effects between the two speed-up techniques, are explained below.

Contraction Hierarchies. Optimizing Contraction Hierarchies for different types of graphs or even just for different metrics is a complex task since there are a lot of parameters that can be adjusted to improve the performance of the algorithm with regards to execution times or contraction quality. An overview of the different parameters is provided by Geisberger in [Gei08].

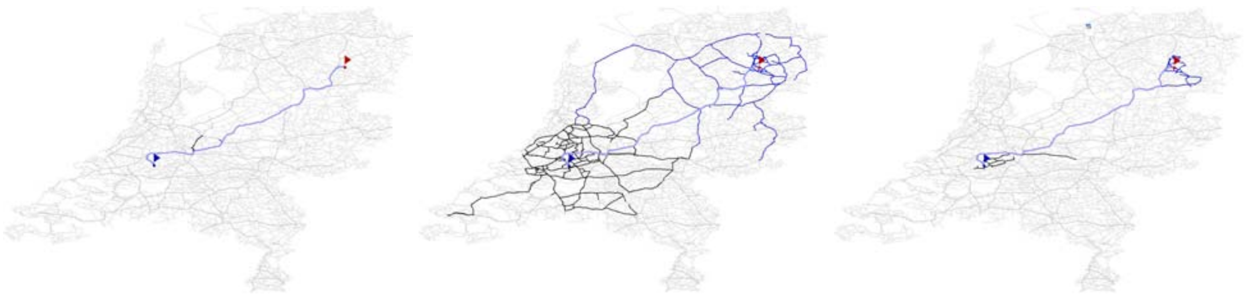

(a) bidirectional ArcFlags (16 regions) **(b)** Highway Node Routing (Contraction Hierarchy) **(c)** Hierarchy-aware ArcFlags (16 regions, core-size 0.5%)

Figure 5.2: *Comparison of the search spaces of ArcFlags, Highway Node Routing and Hierarchy-aware ArcFlags. The latter two use the same hierarchy information and the same shortcuts. Both ArcFlags queries use 16 regions for their graph partition. HiFlags applies this to the subgraph of the 0.5% nodes with highest-level, ArcFlags to the whole graph. The ArcFlags query settles 454 nodes, Highway Node Routing 278 and HiFlags only 81.*

Dense Graphs. As has been observed during the experimental studies, the preprocessing of Contraction Hierarchies on dense graphs slows down considerably towards the end when only few nodes with a high degree remain to be removed. Thus, it might prove to be advantageous to stop the preprocessing at a certain point well before all nodes have been removed and consider these nodes as all having the same highest hierarchical level. The preprocessing becomes faster but the performance of the Highway Node Routing query probably decreases. However, this effect is alleviated for the HiFlags query since ArcFlags are available on this hierarchy level to support the query.

Partitioning. As expected, using a partition based directly on the core of the graph usually yields the best results with regards to the typical requirements on graph partitions, like the distribution of the regions or the number of boundary nodes. This normally also translates to a shorter execution times and less memory consumption of the Centralized ArcFlags preprocessing routine. But apparently, it is sometimes more favourable to use a partition based either on the original graph, on the graph with shortcuts, or just on the core without shortcuts. Here, shorter preprocessing times are obtained even though the number of boundary nodes is increased. These fluctuations depend on how much the preprocessing routine is able to profit from identical subgraphs and are difficult to be quantified in advance.

Locality Effects. The performance of the Hierarchy-aware ArcFlags query can be improved further by utilizing *cache locality effects*. Nodes that are near each other in the graph and that have similar hierarchy levels are probably accessed in short succession and thus should be stored near each other in memory to profit from caching effects. The original node-order often already places nodes close together that are near each other in the graph. This order can be improved further by also incorporating the level of the nodes into the sorting. Here, the following order is suggested: The 50% of nodes with the lowest hierarchy levels are sorted to the front, followed by the next higher 30%, 10%, 5%, 3%, 1%, 0.5% and the final 0.5% of nodes with the highest level. Looking at these numbers from the other direction, starting from the highest level, they specify sets of nodes containing the highest 0.5%, 1%, 2%, 5%, 10%, 20%, 50% and 100% of the hierarchy levels. Incidentally, these are also the core-sizes used later for the experiments.

Stall-On-Demand Technique. This technique has been introduced for Highway Node Routing by Schultes [Sch07]. Thus, it can also be applied to HiFlags. Basically, the technique works as follows: If a shortest-path search reaches a node that has already been settled, this node is 'woken up' and a local search is started from it in order to find and stall other nodes that will not be part of a shortest path, according to the currently available information. The additional overhead of the stalling process makes this technique only useful if a certain minimum number of nodes can be removed from the query in this way, compared to the total number of nodes that are settled. Further details on the stall-on-demand technique can be found in Sect. 2.3.5.

Experiments

Section 2.4 introduced a variety of possible combinations of speed-up techniques. These have been discussed in the previous section with regards to their functionality. Now, their actual performance is evaluated in this chapter. At first, the experimental setup that is used to compile and test the algorithms is described. Then, the different types of graphs that are studied and the actual graph instances that are applied are presented. Afterwards, the experimental results for each technique are shown. They are evaluated extensively and the effects of choosing different preprocessing parameters are discussed. In conclusion, a summery of the obtained results is given and put into the context of other recent techniques.

6.1 Experimental Setup

The algorithms presented in this thesis are implemented in C++, solely using the Standard Template Library (STL) as additional provided code. A binary heap is used as data structure for the priority queue (see App. A for more details). The experiments are performed on one core of an AMD Opteron 2218 running SUSE Linux 10.1, unless otherwise noted. The machine is clocked at 2.6 GHz, has 16 GB of RAM and 2x 1 MB of L2 Cache. The code is compiled with GCC 4.1, using optimization level 4 and unrolling of loops as compiler flags.

Two setups are used to analyze the performance of the algorithms:

Random Queries. 10 000 random pairs of source and target nodes are selected for each graph and used for the queries. All 10 000 queries are performed to measure the performance of a speed-up technique. The relevant values such as the execution time and the number of settled nodes are logged for each individual query and the arithmetic mean over all queries is used as result. This setup is applied to all graphs and all speed-up techniques.

Local Queries. Here, 1 000 (s, t) pairs are chosen for each Dijkstra rank [SS05]. 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, ..., $2^{\log|V|}$. 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].

The queries that are performed in this chapter only provide a distance between the two nodes. A contracted shortest path, still containing shortcuts if existing, can be rebuilt according to the outline given in Sect. 2.2. Already available unpack techniques can be directly applied to the shortcuts in order to obtain the original edges [DSSW06, BFSS07, GKW07].

6.2 Graphs

In order to evaluate the individual strengths and weaknesses of the introduced combinations of speed-up techniques, a wide variety of graph types is applied to measure their performance. There are four principal categories of graphs that are studied: road networks, sensor networks, timetable information systems and grid graphs. The basic attributes of the analyzed instances of these graph types, the number of nodes and the number of edges, are given in Table 6.1. Note that the listed number of edges refers to the number of edge objects that have to stored for a bidirectional query. See App. A for further details.

Road Networks. The European road network, provided by the PTV AG [PTV79] for scientific use, and the road network of the USA, as taken from the DIMACS website [tDIC06], are chosen as representatives of this graph category. The largest strongly connected component of both of them is used as an input for the query algorithms. The provided graphs all use a distance metric. Two variants, one with travel times and another one with a unit metric, i. e. each weight is set to 1, are also generated and applied. To compute travel times, the also provided road categories of each edge are used and assigned reasonable fixed travel speeds.

	nodes	edges
Road Networks		
Europe – all metrics	18 010 173	44 436 348
USA – all metrics	23 947 347	57 708 624
Sensor Networks		
Unitdisk Graph, Deg. 5	994 980	5 101 842
Unitdisk Graph, Deg. 7	996 394	6 986 092
Unitdisk Graph, Deg. 10	999 887	9 987 286
Timetable Information Systems		
Timetable – Railway EU	1 192 736	3 578 168
Timetable – VBB	2 599 953	7 799 430
Timetable – RMV	2 277 812	6 833 104
Grid Graphs		
2-dim. Grid Graph	250 000	998 000
3-dim. Grid Graph	250 047	1 476 468
4-dim. Grid Graph	244 904	1 871 144

Table 6.1: *Listing of the basic attributes of all graphs, used in the experimental studies of the thesis at hand.*

Sensor Networks. The research activity in the field of sensor networks has increased tremendously over the past years. Routing in these distributed networks shows similar properties to routing in normal road networks. Therefore, speed-up techniques for shortest-path queries have become a point of interest in this field of research and are tested on sensor networks. To evaluate algorithms on this kind of networks, *unitdisk graphs* are often applied [KWZ03]. These synthetic graphs provide a good model of the typical structure of sensor networks. They are constructed by distributing nodes randomly on a plane and connecting nodes with a distance below a certain threshold. These distances are also used for the edge-weights. By applying different threshold values, the average degree of the nodes can be varied, leading to graphs that are rather dense or sparse. Here, graphs with an average node-degree of 5, 7 and 10 are used.

Timetable Information Systems. Timetables can be represented as static graphs. Each station at each relevant time step becomes one node of the graph and connections between them are described by the graph-edges (see [MHSWZ07] for details). Normal shortest-path queries can be used to obtain shortest connections in these *time-expanded* networks. The long-distance connections of the European railway network and the local traffics of Berlin/Brandenburg (VBB) and the Rhein-Main-Verkehrsverbund (RMV) are used as examples of this type of graphs. Note that only random queries are performed to gain preliminary results on the performance of the studied algorithms. In particular, the earliest arrival problem is not solved by the applied queries. For more information on this specific problem see e. g. [Paj08].

Grid Graphs. These synthetic graphs are often used to evaluate the influence of different graph diameters on the performance of shortest-path algorithms [BDW07]. The nodes are ordered in the form of a grid of the proper dimension and connected appropriately by bidirectional edges. The edge weights are distributed uniformly at random from 1 to 1 000. Smaller diameters are obtained by increasing the number of dimensions of the grid and, at the same time, keeping the number nodes constant. Note that this also increases the number of edges and therefore the degree of each node, leading to a denser graph. In this thesis, grids with two, three, and four dimensions are discussed.

6.3 Results

The four combinations of speed-up techniques for exact shortest-path queries that have been introduced in this thesis (AALT, CALT, Reach-aware ArcFlags, and Hierarchy-aware ArcFlags) are evaluated below as outlined in Sect. 6.1 on the graphs described in the previous section. The experimental results are presented and subsequently discussed. At the end of the chapter in Sect. 6.4, a comparison of these techniques among each other and compared to previous ones is given to determine their individual strengths and weaknesses on different graphs.

6.3.1 AALT

The AALT algorithm, an ALT algorithm enhanced by ArcFlags-pruning, is a somewhat exotic combination since both of its constituents are goal-directed algorithms. Thus, the possible gain in performance is probably less than for a combination of a goal-directed and a hierarchical technique, since both of them presumably try to exploit similar aspects of the graph. Therefore, only a brief analysis is performed for AALT, but with some interesting results.

Experimental Setup. A slightly different experimental setup than before is applied for computing the ArcFlags of the European road network. Here, a system with two AMD Opteron 2218 and 32 GB RAM is used. The other specifications stay the same. This setup usually performs about 10 to 20% slower than the system with only one CPU, since both CPUs share the same memory controller and thus wait states can ensue. The partitioning is performed with the SCOTCH algorithm [Pel07], using one local optimization run, as explained in Sect. 5.1. ArcFlags are computed with the boundary algorithm by Köhler [HKMS06]. The preprocessing data of the other graphs is taken from [BDW07]. They used the same experimental setup as initially described in Sect. 6.1 and the same preprocessing algorithm but only applied the METIS partitioning [Lab07] with no local optimization runs. ALT applies active landmarks (see Sect. 2.3.2) during the query. No further optimizations are applied.
The listed preprocessing times consist of the preprocessing required by ArcFlags (partitioning the graph, computing ArcFlags) and ALT (selecting landmarks, computing landmark distances). The memory overhead is composed of the costs of the ArcFlags (16 byte/edge), the node-regions (2 byte/node), the landmarks (4 byte each), and the landmark distances (4 byte each).

	AALT (128 regions, 16 landmarks)				AALT (128 regions, 64 landmarks)				ArcFlags (128 regions)			
	— Prepro —		— Query —		— Prepro —		— Query —		— Prepro —		— Query —	
	[s]	[B/n]	settled	[ms]	[s]	[B/n]	settled	[ms]	[s]	[B/n]	settled	[ms]
Europe												
travel times	712 427	209	1 757	0.95	713 196	593	1 613	0.85	707 338	81	2 764	0.80
distances	472 169	209	9 383	4.74	472 313	593	5 966	3.13	467 869	81	31 234	9.14
unit metric	746 663	209	1 454	0.68	746 917	593	1 244	0.68	741 849	81	2 385	0.70
Unitdisk Graphs												
degree 5	6 795	294	1 147	0.90	6 664	678	975	0.77	6 300	166	2 091	1.01
degree 7	36 404	354	1 549	1.47	36 281	738	920	0.78	35 880	226	4 761	2.56
degree 10	96 694	450	2 254	2.05	96 575	834	1 034	0.98	96 120	322	7 019	4.13
Timetables												
Railway EU	121 255	226	4 564	1.78	121 275	610	3 263	1.56	120 960	98	10 560	2.35
VBB	269 357	226	9 005	4.05	269 367	610	6 783	3.26	268 740	98	24 004	5.85
RMV	279 324	226	10 139	4.22	279 337	610	7 091	3.59	278 760	98	28 448	6.92
Grid Graphs												
2-dimensional	5 406	258	602	0.26	5 396	642	495	0.25	5 340	130	1 340	0.35
3-dimensional	49 902	319	461	0.31	49 890	703	297	0.24	49 800	191	1 685	0.62
4-dimensional	187 155	374	575	0.53	187 146	758	372	0.41	187 020	246	2 799	1.42

Table 6.2: *Results of the AALT algorithm with 128 regions, using 16 maxCover and 64 avoid landmarks, respectively. Results of the bidirectional ArcFlags algorithm with 128 regions are also shown for reference.*

	— Parameters — regions	landmarks	— Prepro — [s]	[B/n]	— Query — settled	[ms]
EU – distance metric						
AALT	(128 regions,	64 landmarks)	472 313	593	5 966	3.13
AALT	(128 regions,	16 landmarks)	472 169	209	9 383	4.74
AALT	(16 regions,	64 landmarks)	110 025	524	13 101	8.06
AALT	(16 regions,	16 landmarks)	109 859	140	29 931	16.82
ArcFlags	(128 regions,	)	467 869	81	31 234	9.14
ArcFlags	(16 regions,	)	105 573	12	251 007	81.01
ALT	(	64 landmarks)	4 203	512	67 150	42.03
ALT	(	16 landmarks)	4 226	128	218 420	127.70
EU – travel time metric						
AALT	(128 regions,	64 landmarks)	713 196	593	1 613	0.85
AALT	(128 regions,	16 landmarks)	712 417	209	1 757	0.95
AALT	(16 regions,	64 landmarks)	153 817	524	2 999	1.84
AALT	(16 regions,	16 landmarks)	153 062	140	4 932	2.82
ArcFlags	(128 regions,	)	707 338	81	2 764	0.80
ArcFlags	(16 regions,	)	147 972	12	37 404	14.42
ALT	(	64 landmarks)	5 521	512	26 630	18.37
ALT	(	16 landmarks)	4 986	128	76 621	50.76

Table 6.3: *Comparison of the results of the ArcFlags, ALT and AALT algorithms using several different parameter sets on the European road network with a distance metric and a travel time metric, respectively.*

General Results. Table 6.2 lists the results of the AALT algorithm with 128 regions, either using 16 maxCover, or, respectively, 64 avoid landmarks (refer to [GW05] for an explanation of the different landmark selection strategies). The experimental results of the bidirectional ArcFlags algorithm using the same 128 regions are also shown for reference.
As can be seen, the additional preprocessing time required to compute landmarks and landmark distances is insignificant compared to the time required for the ArcFlags preprocessing, which can take up to almost one week. The memory consumption, on the other hand, rises considerably if landmark distances have to be stored in addition to the ArcFlags, by a factor of about 1.5 to 2.5 for 16 landmarks and about two to three times more than that for 64 landmarks. The query times on all graphs decrease compared to the ArcFlags algorithm using either 16 or 64 landmarks, with the exception of the European road network with travel times or a unit metric. Since the number of settled nodes decreases considerably for all instances, the increased computational overhead of the ALT query, on which AALT is based, probably outweighs the speed-up for these two graphs. The speed-up on the other graphs is generally the more pronounced the denser the graphs become. This is due to the ALT algorithm rearranging the order in which nodes are visited, providing the ArcFlags algorithm with a more direct way to the target if there are a lot of possible paths that can be taken, which is usually true for dense graphs. Switching from 16 to 64 landmarks, considerable improvements are only observed for sensor networks with a high node-degree. Here, the query times decrease by a factor of about two. This is probably due to the query still having to settle a lot of nodes compared to the other graphs, rendering the rearrangement of the nodes to be more beneficial.

Further Parameter Values. A more extensive analysis of the different contributing speed-up techniques is shown in Table 6.3, also comparing the results on the European road network with travel times to the results with a distance metric. The AALT algorithm, the ArcFlags algorithm and the ALT algorithm are listed, using 16 or 128 regions and 16 maxCover or 64 avoid landmarks, respectively. The Arc-Flags preprocessing takes about 1.5 times longer on the graph with travel times than for the same one with the distance metric. This is due to the plain Dijkstra applied for the preprocessing being much faster for the distance metric. Regarding query times, they are up to 10 times faster for travel times graph than for the distance graph, applying the same algorithm and parameter values. Furthermore, only the distance graph profits by adding landmarks to the ArcFlags query, whereas queries on the other graph even get slower. Observing the number of settled nodes and the average number of hops over all shortest paths, which is 889.2, it seems that only the query on the distance graph visits enough nodes to profit from their rearrangement by the ALT algorithm. Probably, there are more potential shortest paths with the same distance than there are with the same travel time (e. g. a motorway and a rural road running next to each other have the same distances but different travel times). Thus, a query on the graph with a distance metric has to settle more nodes.

Summary. It was originally expected that AALT receives its primary benefit compared to ArcFlags alone from local queries within one region. For these queries, bidirectional ArcFlags performs like a bidirectional Dijkstra whereas AALT performs like bidirectional ALT. Surprisingly, the AALT algorithm shows the highest increase in performance for medium-long queries as can be observed in Fig. 6.1. For very short queries it is even slower than ArcFlags. The additional computational overhead probably outweighs the gains of the goal-direction. Thus, it seems as if ALT and ArcFlags exploit different aspects of the graph structure to improve their queries. In particular, the speed-up for medium-long queries can be explained by two effects: The search space of the bidirectional ArcFlags algorithm tends to split up, forming a cone towards the middle where both search directions meet, and thus, decreasing the performance of the query since more paths have to be tracked. On the other hand, the bounds of the ALT algorithm are particularly good far away from the source and target, as experiments have shown. Both effects seem to add up constructively, increasing the query performance.

6.3.2 CALT

The evaluation of the Contracted ALT algorithm is divided into two parts. At first, the graph contraction and the impact of using several different sets of contraction parameters are analyzed. Then, the performance of CALT queries on these graphs is studied. The queries are performed with several different landmark-sets. Additional attention is given to the transition between the first and the second phase of the query and the related measurement values.

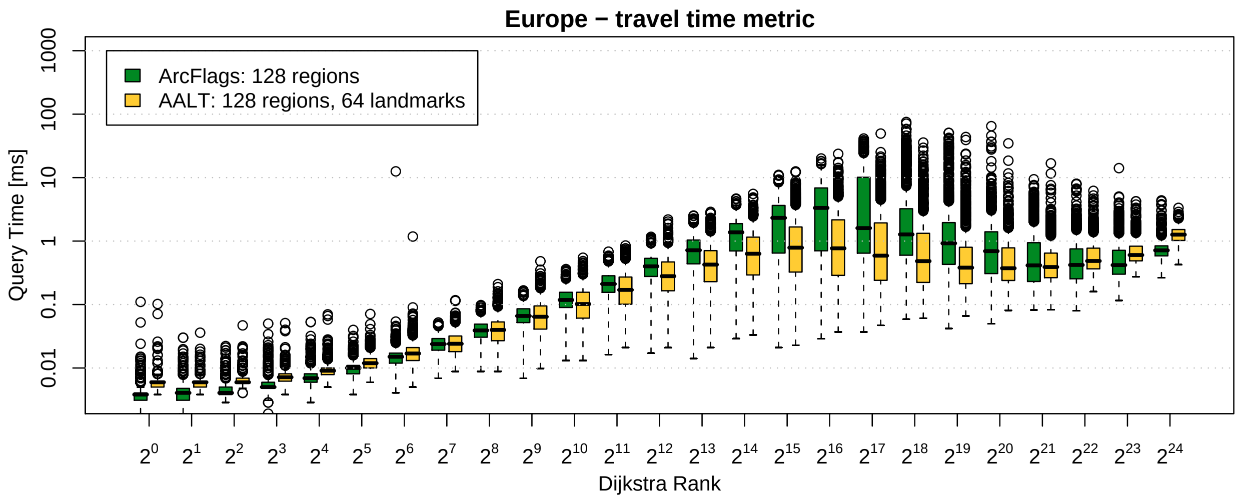


Figure 6.1: *Comparison of the performance of ArcFlags and AALT. Both algorithms use the same flags, computed for 128 regions. AALT also applies 64 avoid landmarks. The query times have been measured on the European road network with travel times. The results are presented using the Dijkstra rank methodology [SS05]. They are shown as box-and-whisker plot [Tea04]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.*

Contraction

The contraction step is the most important part of the whole CALT algorithm. If the computed cores are not viable, the performance and the quality of the subsequent preprocessing steps (landmark selection, computation of landmark distances) and of the query suffer. If the graph is not contracted enough, the core remains quite large and the preprocessing cannot profit from a small input. Also, the query cannot profit of a search restricted to a small core and its performance will be similar to a bidirectional ALT. On the other hand, if the graph is contracted too much, the query stays in the first phase for the better part of the whole search process, only performing like a bidirectional Dijkstra search and the precomputed landmark information is virtually wasted. Furthermore, the contracted graph should not become too dense; otherwise the performance of the query suffers by having to touch too many unnecessary edges. As shown later, the average number of hops from the source and the target to their respective entry points into the core provides a first estimation of the contraction quality.

Setup. Tables 6.4 - 6.8 show the results of the contraction step for each of the types of graphs that are studied. The contractions have been performed using three different sets of parameter values: a conservative one ($c = 1.5$, $h = 10$), a normal one ($c = 3.0$, $h = 30$) and an aggressive one ($c = 5.0$, $h = 100$). The contraction parameters are explained in more detail in Sect. 4.1. The tables list the number of nodes and edges of the contracted graphs, the added shortcuts and the time needed for the contraction. Additionally, the row *full graph* lists the size of the full graph for reference. Note that the number of shortcuts does not always match the listed memory requirements due to edge compression (see App. A).

	nodes	edges	shortcuts	[s]
EU – distance metric				
full graph	18 010 173	44 436 348		
c =1.5, h = 10.0	1 562 171	12 670 804	10 155 111	223
c =3.0, h = 30.0	509 824	10 127 508	7 245 085	690
c =5.0, h = 100.0	146 717	7 933 412	5 040 746	1 787
EU – travel time metric				
full graph	18 010 173	44 436 348		
c =1.5, h = 10.0	1 579 519	12 762 794	10 207 571	222
c =3.0, h = 30.0	530 891	10 361 592	7 343 081	658
c =5.0, h = 100.0	156 466	8 437 280	5 198 747	1 798
EU – unit metric				
full graph	18 010 173	44 436 348		
c =1.5, h = 10.0	1 502 070	12 129 516	9 916 081	225
c =3.0, h = 30.0	457 283	9 314 878	6 912 870	761
c =5.0, h = 100.0	118 852	6 780 910	4 589 877	1 954

Table 6.4: *Experimental results of the contraction step of the CALT algorithm on the European road network. Three different sets of contraction parameters are used for each of the three different metrics, that are analyzed.*

Road Networks. As can be observed in Table 6.4 and 6.5, the contraction performs fine on the road networks of Western Europe and the USA for all metrics and all contraction parameters, that are considered. The size of the core decreases substantially with each more aggressive parameter set, although the ratio of edges to nodes increases. Note, that the number of core-edges and the number of added shortcuts is almost identical for each contraction, meaning that the core primarily consists of new edges. As expected, the time required for performing the contraction increases for more aggressive contraction parameters. The preprocessing times and core-sizes are virtually independent of the underlying metric of the graph. Thus, similar values are obtained for all metrics.

Sensor Networks. The algorithm also yields good results on sensor networks as shown in Table 6.6, even though they get worse on the graphs with a higher average node-degree. For example, the aggressive contraction of the unitdisk graph with degree 10 still retains about one tenth of the nodes of the original graph and about half of its edges, whereas the graph with a degree of 5 can be reduced by 10 to 20 times more, using the same contraction parameters. The preprocessing times also grow several times longer, the worse the graph can be contracted. This indicates that the algorithm has problems with this type of graphs where a lot of nodes have a large degree and thus are not very suited for being bypassed.

Timetable Systems. The same behaviour as for sensor networks can also be seen in Table 6.7 for the timetable information systems, but even more pronounced. The conservative variant yields good results but the number of core-edges already increases by large amount using the normal contraction variant and even surpasses the number of edges in the original graph in two of three instances for the aggressive contraction. This is probably caused by the fairly dense

	nodes	edges	shortcuts	[s]
USA – distance metric				
full graph	23 947 347	57 708 624		
c =1.5, h = 10.0	2 137 049	15 678 376	15 228 860	361
c =3.0, h = 30.0	719 906	11 119 092	11 049 760	658
c =5.0, h = 100.0	212 693	6 847 852	6 834 922	1 728
USA – travel time metric				
full graph	23 947 347	57 708 624		
c =1.5, h = 10.0	2 139 942	15 683 760	15 234 892	360
c =3.0, h = 30.0	721 444	11 080 528	11 010 090	984
c =5.0, h = 100.0	217 095	6 922 236	6 909 684	1 745
USA – unit metric				
full graph	23 947 347	57 708 624		
c =1.5, h = 10.0	2 091 465	15 493 224	15 057 194	367
c =3.0, h = 30.0	672 297	10 693 874	10 628 886	659
c =5.0, h = 100.0	190 266	6 466 056	6 454 152	1 197

Table 6.5: *Experimental results of the contraction step of the CALT algorithm on the road network of the USA. Three different sets of contraction parameters are used for each of the three different metrics, that are analyzed.*

structure of these graphs as described above. One can also observe that the execution time of the aggressive variant is much longer than of the other two variants. This also indicates that the contracted graph is getting too dense and that the parameter values are probably chosen too large for the size of this graph.

Grid Graphs. The experimental results for final type of analyzed graph categories, grid graphs, are presented in Table 6.8. These graphs tend to cause the most problems for all of the studied

	nodes	edges	shortcuts	[s]
Unitdisk Graph, Deg. 5				
full graph	994 980	5 101 842		
c =1.5, h = 10.0	49 368	317 286	295 936	27
c =3.0, h = 30.0	11 794	95 586	94 328	28
c =5.0, h = 100.0	2 622	35 096	35 010	30
Unitdisk Graph, Deg. 7				
full graph	996 394	6 986 092		
c =1.5, h = 10.0	150 768	1 883 622	1 612 900	61
c =3.0, h = 30.0	42 257	959 514	934 780	116
c =5.0, h = 100.0	19 669	752 674	745 012	171
Unitdisk Graph, Deg. 10				
full graph	999 887	9 987 286		
c =1.5, h = 10.0	411 824	6 572 440	4 087 128	103
c =3.0, h = 30.0	195 002	5 141 664	4 612 652	401
c =5.0, h = 100.0	106 114	4 268 468	4 070 606	758

Table 6.6: *Experimental results of the contraction step of the CALT algorithm on sensor networks with varying average node-degrees. Each of the three networks is analyzed using three different sets of contraction parameters.*

	nodes	edges	shortcuts	[s]
Timetable – Railway EU				
full graph	1 192 736	3 578 168		
c =1.5, h = 10.0	212 906	1 852 640	920 209	17
c =3.0, h = 30.0	105 672	2 602 604	1 300 138	32
c =5.0, h = 100.0	62 703	4 971 266	2 485 049	200
Timetable – VBB				
full graph	2 599 953	7 799 430		
c =1.5, h = 10.0	403 022	2 535 500	1 247 979	27
c =3.0, h = 30.0	179 115	3 093 936	1 545 064	43
c =5.0, h = 100.0	95 744	6 414 774	3 206 761	228
Timetable – RMV				
full graph	2 277 812	6 833 104		
c =1.5, h = 10.0	413 536	3 795 694	1 878 006	32
c =3.0, h = 30.0	214 478	5 734 114	2 863 597	75
c =5.0, h = 100.0	133 543	11 699 762	5 848 157	794

Table 6.7: *Experimental results of the contraction step of the CALT algorithm on timetable information systems. Three different sets of contraction parameters are used for each of the three different timetables, that are analyzed.*

algorithms, as the experiments have shown. While the contraction works perfectly fine for the two-dimensional grid, the algorithm breaks down for the higher-dimensional graphs. Either the graph is not contracted at all, using the conservative parameters, or it gets considerably expanded by a large amount of additional shortcuts, using one of the two other variants. The time required for the contraction also increases tremendously for more aggressive parameter values. As with the timetable information systems, the contraction of the grid graphs suffers from their dense structure and of contraction parameter values that are too large for the size of the graphs.

	nodes	edges	shortcuts	[s]
2-dim. Grid Graph				
full graph	250 000	998 000		
c =1.5, h = 10.0	134 475	1 209 516	1 079 922	5
c =3.0, h = 30.0	73 559	1 918 488	1 890 350	34
c =5.0, h = 100.0	53 674	2 204 120	2 186 034	95
3-dim. Grid Graph				
full graph	250 047	1 476 468		
c =1.5, h = 10.0	249 749	1 477 628	3512	1
c =3.0, h = 30.0	143 937	3 130 248	2 847 770	24
c =5.0, h = 100.0	118 410	5 808 322	5 676 232	143
4-dim. Grid Graph				
full graph	244 904	1 871 144		
c =1.5, h = 10.0	244 888	1 871 208	192	1
c =3.0, h = 30.0	219 730	2 375 042	845 246	5
c =5.0, h = 100.0	146 991	5 750 394	5 313 934	82

Table 6.8: *Experimental results of the contraction step of the CALT algorithm on grid graphs of varying dimensions. Each of the three different graphs is analyzed using three different sets of contraction parameters.*

Summary. The contraction works well for sparse graphs. Dense graphs tend to produce too many shortcuts for more aggressive parameter values and an insufficient reduction otherwise. Also, the contraction parameters have to be adjusted with respect to the graph size, e. g. smaller graphs require smaller values and vice versa. In general, if the preprocessing time gets too long, the resulting contraction is usually not viable. The parameter set ($c = 3.0$, $h = 30$), labeled as *normal variant*, seems to yield promising contraction results for all types of graphs.

Query

The performance of the CALT query is analyzed for each graph with each set of contraction parameters mentioned above, using two different sets of landmarks. One uses 16 landmarks generated by the maxCover algorithm, the other uses 64 landmarks chosen by the avoid algorithm. Both sets are selected considering only the core of the graphs. Tables 6.9 - 6.13 list the obtained results for each type of graphs. The average execution time and number of settled nodes of one query is listed for both sets of landmarks. The time required for all of the preprocessing steps (contracting the graph, computing the landmarks, sorting the edges) and the space consumption of the additional information (8 byte/shortcut, 4 byte/landmark, 4 byte/landmark distance) is shown as well. The respective information for a bidirectional ALT query using the same selection method and number of landmarks is also listed for each graph as reference.
Proxy nodes of the source and the target are computed on-the-fly at the beginning of each query. Active landmarks are used to reduce the number of landmarks that have to be considered during the query. Each time after having settled ten percent of the initially projected distance from the source to the target, a checkpoint is reached and additional landmarks can be added to the active set up to a maximum of six active landmarks. This is only done if more nodes have been settled since reaching the last checkpoint than four times the number of available landmarks. Early pruning is also applied during the second phase of the query. No further optimizations are applied.

Road Networks. The algorithm works fine on both road networks with all contraction variants and metrics as shown in Tables 6.9 and 6.10. The preprocessing times always stay far below the time needed to compute the same number of landmarks for the bidirectional ALT. The memory requirements are also lowered by a considerable amount, especially for the variant using 64 landmarks, where the landmark distances constitute most of the overhead. Speed-ups of up to a factor of 10 to 20 depending on the graph and metric are gained using the aggressive contraction. The distance metric delivers a bit higher speed-ups as do the road maps of the USA overall, probably due the problematic long-distance ferry connections in Europe. The number of settled nodes grows for some graphs going from the normal contraction variant to the aggressive one. This signifies that the core is getting too small and that the search is restricted to the first phase for too long. Since the query times mostly still decrease this is alleviated by the fast ALT search in the core, but using even larger contraction parameters should be avoided.

	16 maxCover landmarks				64 avoid landmarks			
	— Prepro —		— Query —		— Prepro —		— Query —	
	[s]	[B/n]	settled	[ms]	[s]	[B/n]	settled	[ms]
EU – distance metric								
bidirected ALT	4 226	128	218 420	127.7	4 203	512	67 150	42.0
c =1.5, h = 10.0	841	17	20 804	20.7	630	50	7 540	8.3
c =3.0, h = 30.0	1 030	8	6 453	9.0	861	19	2 958	4.6
c =5.0, h = 100.0	2 044	5	4 930	5.9	1 867	8	3 850	3.7
EU – travel time metric								
bidirected ALT	4 986	128	76 621	50.8	5 521	512	26 630	18.4
c =1.5, h = 10.0	785	17	7 559	8.6	704	50	2 999	3.7
c =3.0, h = 30.0	960	8	2 878	4.6	855	20	1 394	2.4
c =5.0, h = 100.0	1 977	5	3 473	3.5	1 886	8	3 079	2.6
EU – unit metric								
bidirected ALT	4 750	128	121 480	74.3	4 844	512	33 420	21.5
c =1.5, h = 10.0	1 042	16	10 758	12.5	757	48	3 369	4.2
c =3.0, h = 30.0	1 092	7	4 211	7.2	966	17	1 629	2.8
c =5.0, h = 100.0	2 186	4	6 010	5.1	2 029	6	5 304	3.2

Table 6.9: *Experimental results of the CALT algorithm with 16 maxCover and 64 avoid landmarks on the European road network. Three different sets of contraction parameters have been analyzed for each metric.*

Sensor Networks. The performance on sensor networks also seems to be quite good as can be seen in Table 6.11. The decreasing quality of the contraction on the unitdisk graphs with a higher node-degree directly translates to a decrease in the query performance and memory overhead of the algorithm. Overall, it stays a lot faster than the bidirectional ALT and uses much less additional memory, even down to only one additional byte per node. But these benefits dwindle, the denser the graphs become. On the degree-10 graph, the CALT preprocessing even takes

	16 maxCover landmarks				64 avoid landmarks			
	— Prepro —		— Query —		— Prepro —		— Query —	
	[s]	[B/n]	settled	[ms]	[s]	[B/n]	settled	[ms]
USA – distance metric								
bidirected ALT	6 698	128	278 055	166.9	6 128	512	93 154	60.2
c =1.5, h = 10.0	1 233	17	26 267	25.1	926	51	9 739	10.7
c =3.0, h = 30.0	1 227	8	9 034	11.9	888	19	4 015	6.2
c =5.0, h = 100.0	1 982	3	5 130	6.4	1 823	7	3 735	4.0
USA – travel time metric								
bidirected ALT	7 141	128	179 178	113.3	6 983	512	71 880	48.9
c =1.5, h = 10.0	1 331	17	17 932	18.5	976	51	7 381	9.3
c =3.0, h = 30.0	1 551	8	7 093	10.3	1 232	19	3 240	5.8
c =5.0, h = 100.0	2 084	3	4 447	5.6	1 845	7	3 429	4.0
USA – unit metric								
bidirected ALT	6 374	128	245 125	148.8	6 506	512	75 836	50.1
c =1.5, h = 10.0	1 211	16	21 279	22.6	945	50	7 057	8.8
c =3.0, h = 30.0	1 216	7	8 585	13.1	886	18	3 024	5.3
c =5.0, h = 100.0	1 523	3	5 927	7.4	1 286	6	4 401	4.4

Table 6.10: *Experimental results of the CALT algorithm with 16 maxCover and 64 avoid landmarks on the road network of the USA. Three different sets of contraction parameters have been analyzed for each metric.*

	16 maxCover landmarks				64 avoid landmarks			
	— Prepro —		— Query —		— Prepro —		— Query —	
	[s]	[B/n]	settled	[ms]	[s]	[B/n]	settled	[ms]
Unitdisk Graph, Deg. 5								
bidirected ALT	490	128	10 051	9.3	344	512	3 946	4.5
c =1.5, h = 10.0	50	9	795	0.8	42	28	570	0.6
c =3.0, h = 30.0	34	2	726	0.5	32	7	689	0.5
c =5.0, h = 100.0	32	1	3 363	1.9	31	2	3 348	1.9
Unitdisk Graph, Deg. 7								
bidirected ALT	514	128	10 327	11.8	378	512	3 194	4.3
c =1.5, h = 10.0	161	32	1 853	2.7	121	90	799	1.4
c =3.0, h = 30.0	166	13	927	1.4	135	29	670	1.0
c =5.0, h = 100.0	206	9	1 792	1.8	182	16	1 673	1.5
Unitdisk Graph, Deg. 10								
bidirected ALT	566	128	11 704	15.5	428	512	3 213	5.2
c =1.5, h = 10.0	479	85	5 136	9.0	301	243	1 699	3.6
c =3.0, h = 30.0	658	62	2 523	5.5	511	137	992	2.6
c =5.0, h = 100.0	980	46	1 650	4.2	827	87	849	2.3

Table 6.11: *Experimental results of the CALT algorithm with 16 maxCover and 64 avoid landmarks on sensor networks. The algorithm has been analyzed with three different sets of contraction parameters for each network.*

longer than for the ALT algorithm. This graph is also the only one actually benefiting from using 64 landmarks, probably due to the large cores on which landmarks can be distributed more evenly. The performance on the other two graphs decreases by switching from the normal to the aggressive contraction. Here, more nodes have to be settled since the core does not cover enough of the whole graph. The query times also become longer because of the large number of edges still in the core.

	16 maxCover landmarks				64 avoid landmarks			
	— Prepro —		— Query —		— Prepro —		— Query —	
	[s]	[B/n]	settled	[ms]	[s]	[B/n]	settled	[ms]
Timetable – Railway EU								
bidirected ALT	291	128	30 021	14.4	301	512	16 245	9.8
c =1.5, h = 10.0	123	35	5 385	6.3	93	104	3 573	5.0
c =3.0, h = 30.0	158	29	3 335	7.1	87	63	2 088	5.3
c =5.0, h = 100.0	396	40	2 394	10.1	260	60	1 819	8.5
Timetable – VBB								
bidirected ALT	604	128	5 6404	27.3	600	512	31 866	19.2
c =1.5, h = 10.0	173	28	8 396	9.4	154	87	5 309	7.0
c =3.0, h = 30.0	174	18	4 622	8.8	123	45	2 830	6.3
c =5.0, h = 100.0	440	24	2 890	11.0	308	39	2 264	9.6
Timetable – RMV								
bidirected ALT	556	128	60 004	30.9	552	512	34 551	22.2
c =1.5, h = 10.0	252	36	11 708	15.1	181	106	6 974	10.7
c =3.0, h = 30.0	377	32	7 107	15.8	191	68	4 247	11.3
c =5.0, h = 100.0	1 270	49	5 064	22.2	929	71	3 630	17.9

Table 6.12: *Experimental results of the CALT algorithm with 16 maxCover and 64 avoid landmarks on timetable information systems. Three different sets of contraction parameters have been analyzed for each timetable.*

	16 maxCover landmarks				64 avoid landmarks			
	— Prepro —		— Query —		— Prepro —		— Query —	
	[s]	[B/n]	settled	[ms]	[s]	[B/n]	settled	[ms]
2-dim. Grid Graph								
bidirected ALT	65	128	2 362	1.5	54	512	1 093	0.9
c =1.5, h = 10.0	61	103	1 298	1.4	40	310	650	0.9
c =3.0, h = 30.0	113	98	798	1.5	60	211	458	1.1
c =5.0, h = 100.0	192	97	646	1.4	119	180	436	1.1
3-dim. Grid Graph								
bidirected ALT	100	128	1 759	2.1	82	512	710	1.2
c =1.5, h = 10.0	102	128	1 835	2.5	83	512	818	1.4
c =3.0, h = 30.0	202	165	1 057	3.0	101	386	557	1.9
c =5.0, h = 100.0	410	242	963	4.2	244	424	511	2.6
4-dim. Grid Graph								
bidirected ALT	133	128	1 335	2.6	118	512	662	1.8
c =1.5, h = 10.0	139	128	1 228	2.7	119	512	799	2.1
c =3.0, h = 30.0	171	142	1 275	3.1	129	487	774	2.2
c =5.0, h = 100.0	419	250	843	5.1	251	481	570	3.6

Table 6.13: *Experimental results of the CALT algorithm with 16 maxCover and 64 avoid landmarks on grid graphs. The algorithm has been analyzed with three different sets of contraction parameters for each gird.*

Timetable Systems. Table 6.12 shows the experimental results for timetable information systems. The conservative contraction seems to yield the best results, running about two to three times faster than the bidirectional ALT and requiring only about one fifth of its space. The normal contraction reduces the memory overhead further but the query times increase slightly even though the number of settled nodes decreases. This is caused by the large number of additional shortcuts in the core that are touched by the query. The aggressive variant increases the query times further even though they are still well below the query times of the bidirectional ALT. The memory overhead also starts to increase again. This is caused by the large number of shortcuts that are stored and that cannot be compensated by the reduced number of landmark distances. Note that although the number of settled nodes is about 10 times less for CALT compared to ALT, the query times only decrease by two times. This is due to the extremely large number of edges the query has to touch.

Grid Graphs. As can be seen in Table 6.13, the CALT algorithm is not very suited for grid graphs. Viable results can be obtained for two dimensional grids. Even though the query times do not improve, the memory overhead is lowered, especially when using 64 landmarks. The reason for the constant query times even with the number of settled nodes decreasing considerably is, that a lot more edges have to be touched. This is caused by the normal and the aggressive contraction both yielding about twice as much edges as the original graph. The three and four dimensional grids do not produce any useful results. The preprocessing and query times are longer than for the bidirectional ALT and the space saved by having to store fewer landmarks is compensated by the memory required for saving the additional landmarks.

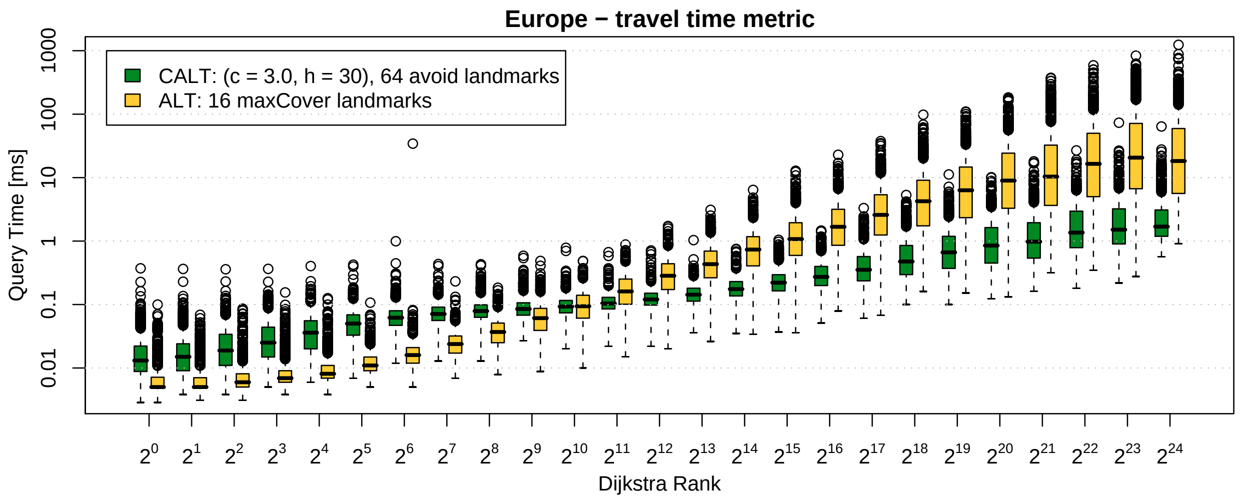


Figure 6.2: *Comparison of the bidirectional ALT algorithm with 16 maxCover landmarks and the CALT algorithm with 64 avoid landmarks on the European road network. The results are shown as a Dijkstra rank plot [SS05].*

Local Queries. A direct comparison of the ALT algorithm and the CALT algorithm on queries with different lengths is shown in Fig. 6.2. The European road network with travel times and contraction parameters of (c = 3.0, h = 30) are used for this analysis. The ALT query with 16 maxCover landmarks yields faster results for shorter queries, since it does not have to deal with two distinct phases and thus has less computational overhead. But, starting at a Dijkstra rank of about 2^{10}, the CALT algorithm with 64 avoid landmarks prevails by up to and more than one order of magnitude, regarding the longest queries. Figure 6.3 also indicates, that the CALT algorithm does not best the ALT algorithm solely because of the larger number of landmarks. For this comparison both algorithms use 64 landmarks and the results stay principally the same, only the differences between them get marginally less pronounced.

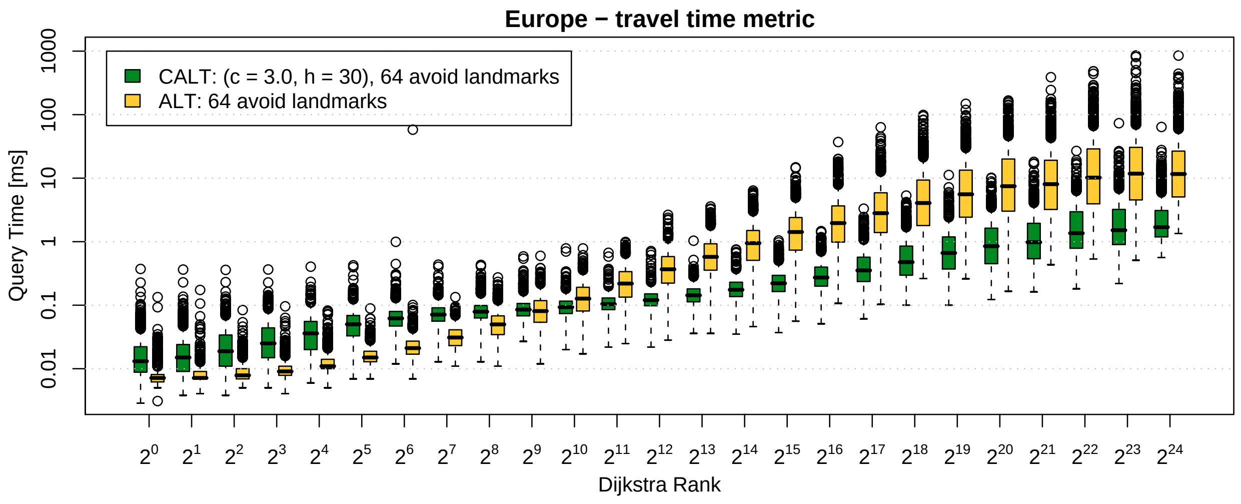


Figure 6.3: *Comparison of the bidirectional ALT algorithm and the CALT algorithm on the European road network, both using 64 avoid landmarks. The results are presented in the form of a Dijkstra rank plot [SS05].*

Number of Landmarks. Additional analyses have been performed on the European road network with a travel time metric. Table 6.14 lists the results of varying the set of landmarks. As can be seen, using the maxCover algorithm for choosing landmarks instead of the avoid algorithm does not improve the query, only the preprocessing times get longer. Increasing the number of landmarks generally improves the performance of the query at the cost of a larger memory overhead and longer preprocessing times. The duration of the landmark computation even starts to dominate over the time needed for contraction step, which happens sooner for smaller contraction parameter values. The results scale best using the normal contraction parameters. Here, even using 512 landmarks, the preprocessing time and the additional space consumption stay below the requirements of the normal bidirectional ALT algorithm with only 16 landmarks, but the resulting CALT query times are more than 30 times faster than for bidirectional ALT. Regarding the number of settled nodes, the conservative approach also seems to profit greatly from an increased number of landmarks, but the query times do not scale as well. This effect is probably due to a larger computational overhead when selecting new landmarks and rebuilding the priority queues on a larger core and for more queued nodes. The aggressive variant has exactly the opposite problem: The core obtained by the contraction is too small to really benefit from additional landmarks.
Thus, it can be concluded that the choice of the contraction parameter values has a large impact on how much a CALT query can possibly profit from increasing the number of landmarks. A normal contraction with (c = 3.0, h = 30) seems to yield the most promising and flexible results.

	— Prepro —		— Query —	
	[s]	[B/n]	settled	[ms]
EU – travel time metric				
c = 1.5, h = 10				
64 maxCover landmarks	7 361	50	3 068	4.0
64 avoid landmarks	704	50	2 999	3.7
128 avoid landmarks	1 415	95	2 105	2.6
256 avoid landmarks	3 539	185	1 636	2.0
512 avoid landmarks	10 648	365	1 338	1.9
c = 3.0, h = 30				
64 maxCover landmarks	5 406	20	1 378	2.4
64 avoid landmarks	855	20	1 394	2.4
128 avoid landmarks	1 120	35	1 163	2.0
256 avoid landmarks	1 941	65	1 051	1.7
512 avoid landmarks	4 390	125	903	1.4
c = 5.0, h = 100				
64 maxCover landmarks	4 912	8	3 049	2.6
64 avoid landmarks	1 886	8	3 079	2.6
128 avoid landmarks	1 986	13	2 992	2.3
256 avoid landmarks	2 257	22	2 953	2.2
512 avoid landmarks	3 070	39	2 909	2.1

Table 6.14: *Results, varying the set of landmarks used.*

	16 maxCover landmarks			
	— Prepro —		— Query —	
	[s]	[B/n]	settled	[ms]
EU – travel time metric				
c = 5.0, h = 100	1 977	5	3 473	3.5
c = 5.0, h = 200	2 742	4	9 919	5.0
c = 5.0, h = 500	2 948	3	29 084	11.6
c = 5.0, h = 1000	2 948	3	34 700	14.0
c = 10.0, h = 100	2 279	5	3 755	4.0
c = 20.0, h = 100	2 362	5	3 679	3.8
c = 50.0, h = 100	2 307	5	3 670	3.7

Table 6.15: *Results, varying the contraction parameters.*

Further Contraction Rates. Table 6.15 shows the results of using even larger contraction parameters than for the aggressive approach with (c = 5.0, h = 100). Using only 16 landmarks, the preprocessing time is dominated by the contraction step for all analyzed parameters sets. Increasing the contraction factor c does not alter the results by much, since the graph cannot be contracted substantially further without having the shortcuts to cover more than 100 hops. However, by increasing the maximum hop count h, the graph can be contracted even further. But this significantly decreases the performance of the query, since it is stuck in the first phase for a long time, only performing a normal bidirectional Dijkstra search and not profiting from the shortcuts of the core and the ALT search used in the second phase. Thus it is safe to say, that on a road map the size of Western Europe the contraction is saturated using the aggressive parameter set. Graphs of a different size or structure naturally require other parameters for the contraction to become saturated.

Proxy Nodes and Entry Points. The impact of computing proxy nodes for the source and target on-the-fly has also been studied on the European road network with a travel time metric. As shown in Table 6.16, the time that is required and the number of nodes that have to be settled in order to find the proxies is several orders of magnitude less than needed for the whole query. Therefore it is not an issue to compute them on-the-fly instead of beforehand. The number of hops denotes the average distance of the source and target to their respective proxy nodes,

	— Proxy Nodes —			— Entry Points —		
	[μs]	settled	hops	count	settled	hops
EU – travel time metric						
c = 1.5, h = 10	9.2	19.4	3.2	6.7	54.5	4.7
c = 3.0, h = 30	20.7	64.8	6.6	14.0	295.8	13.2
c = 5.0, h = 100	109.8	359.1	15.5	34.0	2 617.4	40.5

Table 6.16: *Additional measurement results of the CALT algorithm on the European road network with a travel time metric concerning the computation of the proxy nodes and the query up to the entry points into the core.*

measured with unit edge weights. Since these values are very low, the distance approximation does not suffer much from the use of proxy nodes.
Table 6.16 also provides additional information concerning the application of a two-tiered query. The average number of nodes that are settled in the first phase is shown, as are the number of entry points into the core that are found and their average hop-distance from the source or, respectively, the target. These numbers increase as the cores get smaller due to more aggressive contraction parameters. Note that for the aggressive parameter set, the query spends most of its time in the first phase with regards to the number of settled nodes. But it is still about as fast as or even a bit faster than the query using normal contraction parameters. This indicates that the ALT query on the smaller core can just about compensate for the long search in the first phase. Note that the hop-distance can be used as an initial estimate of the contraction quality.

	— Prepro —		— Query —	
	[s]	[B/n]	settled	[ms]
EU – travel time metric				
bidirectional Dijkstra	0	0.0	5 003 642	2 605.0
c = 1.5, h = 10	232	5.5	448 701	395.8
c = 3.0, h = 30	668	4.6	151 698	183.9
c = 5.0, h = 100	1 809	3.7	48 052	92.6

Table 6.17: *Experimental results of the bidirectional Dijkstra and the Contracted Dijkstra algorithms on the European road network with travel times. Three different sets of contraction parameters have been analyzed.*

Contracted Dijkstra. To conclude the analysis of the CALT algorithm, Table 6.17 lists the results of the Contracted Dijkstra algorithm alone without using the ALT speed-up technique in the second phase. The performance of the bidirectional Dijkstra is also shown for reference. Again, the European road network with travel times is used. As can be observed the Contracted Dijkstra is about one order of magnitude faster than the bidirectional Dijkstra at the expense of additional preprocessing time and a little memory overhead. This is about the same amount by which the CALT algorithm surpasses the bidirectional ALT algorithm. Thus, the speed-ups of both techniques act multiplicative when the algorithms are combined, meaning that they exploit disjunctive properties of the graphs to obtain their goal.

Summary. As already been reckoned in Sect. 4.3, this combination of a simple hierarchical technique and a goal-directed one turns out to be a significant step up from its constituting speed-up techniques in all relevant respects: Preprocessing times, memory overhead and query times. As already observed for the graph contraction, the parameter values (c = 3.0, h = 30) seem to be most promising. They yield the most reliable results but can still be tweaked further for individual graphs. Using 64 avoid landmarks is a good compromise between the gained speed-up and the additional memory overhead. If space is an issue, using only 16 landmarks still delivers very good results. If not, using even more landmarks can improve the query performance further depending on the underling contraction.

6.3.3 ReachFlags

Considering the performance of the REAL algorithm, that combines the Reach algorithm with the goal-direction of ALT, it is expected, that the Reach-aware ArcFlags algorithm will perform similarly, yielding equally good or even better results. It applies the Partial ArcFlags technique instead of the ALT algorithm, which usually performs a lot better, when applied to the full graph. It is attempted to reduce the higher preprocessing costs for the ArcFlags by running the preprocessing of the approximate reach values only for some few iterations, thus bounding and having to store only low reach values and by using just partial ArcFlags.

Setup. The approximate Reach preprocessing by Goldberg et al. [GKW07] is used to compute node-reach values and shortcuts. The graph is partitioned with the SCOTCH algorithm [Pel07], using one local optimization run (see Sect. 5.1). Here, the full graph with shortcuts is partitioned and the resulting node-regions are used for the core, since this yields shorter preprocessing times for some unknown reason. ArcFlags are computed with the Centralized ArcFlags algorithm. The query only applies early pruning to speed-up the search.
Tables 6.18 - 6.22 list the results of the Reach-aware ArcFlags algorithm on different types of graphs. For each graph two different core-sizes are analyzed, one obtained by running two iterations of the approximate Reach preprocessing algorithm, the other by running three iterations. The results of the normal Reach algorithm are also shown for reference. The number shortcuts added by the Reach preprocessing is listed as well as the respective core-sizes. The maximum reach value bound, when stopping the preprocessing after several iterations, is also given. In case of the Reach algorithm, the initial bound value is listed. The preprocessing times sum up the times needed to compute the reach values and shortcuts, to partition the graph and to compute the ArcFlags on the core. The overhead consists of the additional space required by the shortcuts (8 byte each), the reach values (4 byte/node), the node-regions (2 byte/core-node) and the ArcFlags (16 byte/core-edge in each search direction).

Road Networks. The road networks of Europe and of the USA have been analyzed for the three available metrics: distance, travel times and unit. The experimental results are shown in Table 6.18 and 6.19. As can be observed, the preprocessing times increase using the Partial ArcFlags approach. Apparently, stopping the Reach preprocessing after the second or the third iteration cannot compensate the additional time required to partition the graph and to compute ArcFlags. With an even smaller core, the preprocessing times decrease again. The memory overhead acts in a similar fashion. Both values are directly related to the core-sizes. Apparently, the unit metric yields the smallest core-sizes and the travel time metric the largest ones after a fixed number of iterations of the Reach preprocessing. This discrepancy is more pronounced for the European road network. Here, the core-sizes are even larger than for the USA regarding the ratio of core-nodes to the total number of nodes. The query times of ReachFlags are usually a lot faster than for Reach alone. The speed-up is more pronounced the larger the core is,

	reach		— Core —		— Prepro —		— Query —	
	bound	shortcuts	nodes	edges	[s]	[B/n]	settled	[ms]
EU – distance metric								
Reach algorithm	3 000	45 428 342			4 656	24	18 644	22.08
2 iterations	9 000	39 557 344	627 069	7 119 838	73 554	34	5 280	3.43
3 iterations	27 000	42 675 512	288 537	4 646 592	31 687	31	5 224	4.05
EU – travel time metric								
Reach algorithm	2 500	38 045 564			4 200	21	7387	6.24
1 iterations	2 500	27 058 950	2 678 156	11 065 316	24 020	36	1149	0.62
2 iterations	7 500	34 366 006	958 618	5 955 840	13 756	30	1168	0.76
3 iterations	22 500	37 263 178	261 801	2 385 494	6 425	25	2797	2.24
4 iterations	67 500	37 897 178	35 674	395 604	4 152	22	5718	5.34
EU – unit metric								
Reach algorithm	20	30 373 276			3 918	17	6 637	5.97
2 iterations	60	29 698 866	272 501	2 856 382	6 606	22	1 605	1.16
3 iterations	180	30 216 408	33 472	415 612	3 954	18	4 699	4.41

Table 6.18: *Results of the Reach-aware ArcFlags algorithm with 128 regions on the European road network with travel times, a distance metric and a unit metric. Several different core sizes have been analyzed.*

ranging anywhere between a factor of 3.7 to 8.2 for iteration-2 cores. Switching from two to three iterations only really affects the query times of the graphs with a unit metric and of the European road network with travel times. Overall, the query times are a lot slower for the graphs with a distance metric than for the other ones. For the European road network with travel times, two additional core-sizes are also analyzed, one obtained after the first iteration of the Reach preprocessing, the other after four iterations. As can be seen, increasing the core-size does not improve the query times by much but requires a lot more preprocessing time and additional memory. Further decreasing the core-size, on the other hand, reduces both values to be similar to the Reach algorithm, but the query time increases considerably.

	reach		— Core —		— Prepro —		— Query —	
	bound	shortcuts	nodes	edges	[s]	[B/n]	settled	[ms]
USA – distance metric								
Reach algorithm	43 000	51 546 204			5 462	21	18 233	20.31
2 iterations	129 000	45 565 848	768 219	7 305 646	214 769	29	7 417	4.45
3 iterations	387 000	48 643 866	367 416	4 787 388	113 815	27	6 849	4.69
USA – travel time metric								
Reach algorithm	80 000	42 905 650			3 729	18	4 261	3.90
2 iterations	240 000	39 554 808	794 747	5 743 420	19 065	25	1 636	1.02
3 iterations	720 000	42 070 634	261 070	2 398 608	7 351	21	1 844	1.33
USA – unit metric								
Reach algorithm	27	37 881 910			3 635	17	6 533	6.60
2 iterations	81	36 702 490	260 876	2 538 138	10 345	20	2 590	1.78
3 iterations	243	37 580 146	64 626	807 114	4 690	18	3 366	2.85

Table 6.19: *Experimental results of the Reach-aware ArcFlags algorithm with 128 regions on the road network of the USA with three different metrics. Using a larger and a smaller core size has been analyzed for these graphs.*

	reach bound	shortcuts	— Core — nodes	edges	— Prepro — [s]	[B/n]	— Query — settled	[ms]
Unitdisk Graph, Deg. 5								
Reach algorithm	1 700	1 946 524			175	20	1 185	0.88
2 iterations	5 100	1 897 758	9 438	77 320	202	22	616	0.44
3 iterations	15 300	1 928 974	3 006	33 180	196	21	926	0.76
Unitdisk Graph, Deg. 7								
Reach algorithm	1 145	4 767 202			2 541	42	13 790	15.88
2 iterations	3 435	4 271 928	72 609	924 938	2 849	68	4 301	3.68
3 iterations	10 305	4 479 856	38 336	481 428	2 156	56	9 190	9.22
Unitdisk Graph, Deg. 10								
Reach algorithm	890	8 775 562			112 806	74	56 910	83.69
2 iterations	2 670	6 598 278	265 246	4 074 898	55 823	188	15 605	14.45
3 iterations	8 010	7 440 600	159 660	2 346 450	31 430	139	36 224	41.64

Table 6.20: *Experimental results of the Reach-aware ArcFlags algorithm with 128 regions on a sensor network with average node-degrees of 5, 7 and 10. Two core sizes have been analyzed for a graph with each degree.*

Sensor Networks. The experimental results for sensor networks are shown in Table 6.20. All relevant measurement values increase with a growing density of the graph, up to several orders of magnitude. This increase is also reflected in the core-sizes. For the degree-5 senor network, the cores contain just about 0.9% or, respectively, 0.3% of all nodes, whereas the degree-10 graph has core-sizes of about 25.0% or, respectively, 15.0%. Stopping the Reach preprocessing after the second or after the third iteration and computing ArcFlags for the core increases the memory overhead but does not change the preprocessing times by much for the two sparser graphs. For the degree-10 graph, the preprocessing even becomes faster since the largest part of the Reach preprocessing is skipped, but it also has the highest memory overhead due to many shortcuts. Using Partial ArcFlags, the query times get considerably faster on all graphs. This is more pronounced for applying only two iterations of the preprocessing than three.

	reach bound	shortcuts	— Core — nodes	edges	— Prepro — [s]	[B/n]	— Query — settled	[ms]
Timetable – Railway EU								
Reach algorithm	180	5 784 656			3 516	43	9 061	9.67
2 iterations	540	2 796 940	563 846	2 405 312	81 392	88	7 617	3.57
3 iterations	1 620	4 292 272	286 805	1 783 622	74 608	81	4 856	3.04
Timetable – VBB								
Reach algorithm	60	11 324 398			10 755	39	26 951	29.03
2 iterations	180	6 952 968	864 733	3 602 444	-	70	-	-
3 iterations	540	8 720 498	508 571	2 595 400	-	63	-	-
Timetable – RMV								
Reach algorithm	50	11 756 266			16 765	45	38 631	44.96
2 iterations	150	5 752 322	1 123 066	4 980 590	-	95	-	-
3 iterations	450	8 368 332	653 895	3 887 410	-	89	-	-

Table 6.21: *Experimental results of the Reach-aware ArcFlags algorithm with 128 regions on three different timetable information systems. Two core sizes have been analyzed for each of these timetable networks.*

	reach bound	shortcuts	— Core — nodes	edges	— Prepro — [s]	[B/n]	— Query — settled	[ms]
2-dim. Grid Graph								
Reach algorithm	1 750	1 190 342			159	42	3 691	2.98
2 iterations	5 250	1 052 908	22 567	247 656	234	70	1 977	1.27
3 iterations	15 750	1 144 232	7 210	99 354	156	53	3 365	2.85
3-dim. Grid Graph								
Reach algorithm	500	1 605 636			13 041	55	21 832	22.45
2 iterations	1 500	777 746	148 002	1095086	14 403	170	13 718	8.80
3 iterations	4 500	1 367 374	35 089	387430	13 444	98	21 834	25.17
4-dim. Grid Graph								
Reach algorithm	270	1 398 310			124 307	50	20 097	28.09
2 iterations	810	83 312	206 586	1 412 586	73 093	193	16 623	14.35
3 iterations	2 430	656 838	111 512	877 412	124 006	141	20 097	26.54

Table 6.22: *Experimental results of the Reach-aware ArcFlags algorithm with 128 regions on two-, three- and four-dimensional grid graphs. The impact of two different core sizes have been analyzed for each of these graphs.*

Timetable Systems. Regarding the timetable information systems in Table 6.21, only the long-distance connections of the European railway network are analyzed completely. The ArcFlags computation on the two other graphs terminated prematurely due to memory constraints. This is a known problem of the current implementation of the Centralized ArcFlags algorithm used. As can be observed from the remaining data, the Reach preprocessing creates more shortcuts than for any of the previous types of graphs, with regards to number of edges in the original graph. After two iterations there are almost as much shortcuts as original edges and for the full Reach preprocessing, there are two times more shortcuts than original edges. This is reflected in the large memory overhead of the Reach algorithm with about 40 bytes/node. Adding ArcFlags doubles this amount, approximately. The actual core-sizes after the second iteration are about one third to one half of the original graph, regarding the number of nodes. The third iterations shrinks them by an additional factor of 1.5 to 2.0. The query times on the European railway network decrease by about a factor of three, using the Partial ArcFlags approach, with three iterations being the overall better choice for the Reach-aware ArcFlags algorithm.

Grid Graphs. Table 6.22 lists the experimental results of the Reach-aware ArcFlags algorithm on grid graphs. As can be seen, using an iteration-2 core speeds-up the query times by about a factor of two on all grids graphs, but at the cost of a large increase in memory consumption. Switching to an iteration-3 core, this overhead is reduced, but it is still much larger than for the Reach algorithm. Furthermore, the queries are no longer faster than simply using the Reach algorithm. At least, their preprocessing times also stay similar to the ones of the Reach algorithm. Interestingly, using the iteration-2 cores, the preprocessing times increase for the two- and three-dimensional grids, albeit only slightly for the latter one, whereas they decrease substantially for the four-dimensional grid graph. This is due to the two initial iterations taking most of the Reach preprocessing time for the 2D- and 3D-grids.

Summary. Overall, Reach-aware ArcFlags produces substantial speed-ups compared to Reach alone, up to a factor of about 10. But this comes at the cost of usually much longer preprocessing times and an increased memory overhead. The assumption that the shortened Reach preprocessing would compensate for the additional ArcFlags preprocessing did only prove true for dense graphs and using two iterations. However, the main disadvantage of this technique is the difficulty to estimate its performance in advance given the parameter values. The performance is directly dependent on the size of the core, which in turn depends on the number of iterations of the Reach preprocessing and the initial reach bound. But the core-size cannot be easily derived from these two values. In particular, applying the same values to different graphs yields widely varying core-sizes.
To deal with this problem, an alternative approach to the one studied here could be taken: The Reach preprocessing is performed until after an iteration step, the reach values for a fixed number of nodes, or more, are bound. ArcFlags are only computed for the intended core-size; additional bound reach values are either discarded or can still be applied to improve the query.

6.3.4 HiFlags

The Hierarchy-aware ArcFlags algorithm appears to be a very potent speed-up technique for finding exact shortest paths. Its performance depends on two factors, the quality of the search graph built with the Contraction Hierarchy algorithm and the size of the subgraph for which ArcFlags are computed. The former algorithm and a set of aggressive parameters are provided by Geisberger and applied 'out of the box'. Further explanations of this algorithm and its parameters can be found in [Gei08]. All search graphs but the one for the four-dimensional grid have been built using the same set of parameters. This is not optimal, since the aggressive parameters are optimized for road networks with a travel time metric, but very short query times are obtained, nonetheless. More conservative parameters had to be used for the four-dimensional grid graph, since otherwise, the preprocessing has taken too long (aborted after two weeks). The impact of varying the core-size is analyzed in greater detail for the European road network with travel times, after presenting an overview of all graph types. For all graphs, ArcFlags are computed using a graph partition of 128 regions, generated on the graph-cores with the SCOTCH partitioning algorithm [Pel07] and applying one local optimization run.

Setup. Tables 6.23 - 6.27 show the results of the Hierarchy-aware ArcFlags algorithm on several different types of graphs. The results of Highway Node Routing, using the same search graph are also given for reference. Two different core-sizes are analyzed, one using 0.5% of the nodes of the full search graph (economic variant) and one using 5.0% of the nodes (generous variant). The sizes of the different cores are listed for each graph. In case of Highway Node Routing, the size of the whole search graph, on which the queries are performed, is given instead. Note, that this size is usually smaller than the size of the original graph, due to the removal of unnecessary edges. The preprocessing time listed for each variant consists of the time to build the search graph with the Contraction Hierarchy algorithm, the time to divide the core into partitions and the time to compute the ArcFlags. For Highway Node Routing, only the construction time of the search graph is relevant. The specified overhead values consist of the savings due to a decrease in the number of graph edges (8 byte/edge) and the costs for storing the regions (2 byte/node) and the ArcFlags (8 byte/edge in each search direction) for the core. The overhead can reach negative values since only a search graph is saved with all unnecessary edges removed (see Sect. 2.3.5). Note that the actual number of edge objects that have to be saved varies slightly due to edge compression (see App. A). The average number of settled nodes and the average search times that are listed have been obtained with a query using the stall-on-demand technique and no further optimizations.

Road Networks. The algorithm performs extremely well on road networks as can be seen in Table 6.23 for Europe and in Table 6.24 for the USA. As expected, the Contraction Hierarchy algorithm produces the best results for graphs with the travel time metric. The worst ones are obtained for graphs with the distance metric. This is reflected in all relevant values, like the preprocessing time, the size of the core and the performance of the Highway Node Routing query. Interestingly, this discrepancy is much more pronounced for Europe than for the USA. By using Hierarchy-aware ArcFlags the performance of the query can be increased even further. The query times for all metrics stay well below 100 μs, using a core-size of 5.0%. The large discrepancy between travel times and the distance metric also becomes smaller. But this comes at the cost that the graphs with a distance metric require the most additional preprocessing time and space. This is due to their cores being a lot larger compared to the others with regards to the number of edges. Here, the economic variant with a core-size of 0.5% could be a good compromise, yielding query times of only about a factor of three worse, still below 220 μs, but requiring about five times less additional costs.

	— Core —		— Prepro —		— Query —	
	nodes	edges	[s]	[B/n]	settled	[μs]
EU – distance metric						
Highway Node Routing	18 010 173	44 285 842	5 335	-0.07	1 650	4 188.9
core-size 0.5%	90 051	3 935 856	13 486	6.96	175	218.2
core-size 5.0%	900 509	15 187 782	61 342	27.11	67	86.4
EU – travel time metric						
Highway Node Routing	18 010 173	38 317 936	1 515	-2.72	355	249.3
core-size 0.5%	90 051	1 501 868	1 888	-0.03	86	64.0
core-size 5.0%	900 509	8 239 136	7 449	12.07	43	28.2
EU – unit metric						
Highway Node Routing	18 010 173	40 601 813	1 339	-1.70	402	304.3
core-size 0.5%	90 051	1 643 670	1 968	1.32	119	94.3
core-size 5.0%	900 509	9 951 976	11 003	16.54	50	34.4

Table 6.23: *Experimental results of the Hierarchy-aware ArcFlags algorithm on the European road network with three different metrics. An economic variant (0.5% core) and a generous variant (5.0% core) are presented.*

	— Core — nodes	edges	— Prepro — [s]	[B/n]	— Query — settled	[μs]
USA – distance metric						
Highway Node Routing	23 947 347	54 055 295	3 421	-1.22	953	1 496.5
core-size 0.5%	119 737	2 768 598	11 135	2.49	148	153.3
core-size 5.0%	1 197 368	14 572 886	67 965	18.35	63	63.1
USA – travel time metric						
Highway Node Routing	23 947 347	50 910 897	1 633	-2.27	278	185.3
core-size 0.5%	119 737	1 122 592	2 158	-0.76	93	67.7
core-size 5.0%	1 197 368	9 893 012	13 709	11.05	46	29.8
USA – unit metric						
Highway Node Routing	23 947 347	49 155 721	1 718	-2.86	480	434.0
core-size 0.5%	119 737	1 488 734	3 092	-0.86	98	78.6
core-size 5.0%	1 197 368	9 616 862	16 094	10.09	50	37.1

Table 6.24: *Experimental results of the Hierarchy-aware ArcFlags algorithm on the road network of the USA with three different metrics. An economic variant (0.5% core) and a generous variant (5.0% core) are presented.*

Sensor Networks. The results obtained for sensor networks vary greatly with the average node-degree, as can be observed in Table 6.25. In general, the performance gets worse the denser the graphs become. The construction of the search graph takes substantially longer for higher average node-degrees and the memory savings become less, since more shortcuts are added. The preprocessing times probably 'explode' since after each node removal all neighbours of this node have to be considered and updated. And, because of the high average node-degree this results in many additional operations. The query times of Highway Nodes Routing are only good for the degree-5 network with 136.5 μs. For the two other networks only 1.8 ms or, respectively, 11.5 ms are achieved, which is in the same general area as ALT or even slower (see Table 6.11). By using economic HiFlags, the query times are reduced to about one third with virtually no additional preprocessing time, only the memory overhead increases slightly. This increase is more

	— Core — nodes	edges	— Prepro — [s]	[B/n]	— Query — settled	[μs]
Unitdisk Graph, Deg. 5						
Highway Node Routing	994 980	3 446 761	94	-13.31	236	136.5
core-size 0.5%	4 975	58 862	103	-11.40	66	43.2
core-size 5.0%	49 749	362 322	183	-1.55	43	26.2
Unitdisk Graph, Deg. 7						
Highway Node Routing	996 394	5 629 942	1 249	-10.89	1 089	1 800.2
core-size 0.5%	4 982	244 048	1 368	-3.04	424	568.1
core-size 5.0%	49 820	1 187 948	3 150	27.36	112	117.9
Unitdisk Graph, Deg. 10						
Highway Node Routing	999 887	9 512 599	34 274	-3.80	2 475	11 515.0
core-size 0.5%	5 000	599 222	34 847	15.39	1 457	4 702.9
core-size 5.0%	49 995	2 765 204	55 429	84.80	293	525.6

Table 6.25: *Experimental results of the Hierarchy-aware ArcFlags algorithm on three different sensor networks with varying densities. An economic variant (core-size 0.5%) and a generous variant (core-size 5.0%) are shown.*

pronounced for denser networks since their cores contain more edges, for which ArcFlags have to be stored. For example, the degree-10 graph has about ten times as much edges as the degree-5 graph. The huge differences in query times that are observed for Highway Node Routing, about two orders of magnitude, can be reduced to a factor of about 20 by using the generous HiFlags. Even on the on the degree-10 network, query times well below 1 ms are obtained. But this boost in performance comes at the cost of an additional five hours of preprocessing time and a memory overhead of about 85 bytes/node. The gain in speed-up by switching from the economic variant to the generous variant is less pronounced for the other two graphs but the additional costs also stay a lot smaller. The search graph for the degree-5 sensor network even still retains an effective negative memory overhead compared to the original graph size.

Timetable Systems. As shown in Table 6.26, the performance of Highway Node Routing is already quite good on timetable information systems with query times of 0.3 ms to 0.6 ms, depending on the actual graph. Preprocessing times also stay short, between 10 to 40 min. There is a small effective memory overhead since the number of edges cannot be reduced below their number in the original graph. There are only relatively few shortcuts added to the search graph, but there are also only few unnecessary edges that can be removed. The performance is further increased by using Hierarchy-aware ArcFlags, but the impact of adding ArcFlags to the query is smaller than for the previous types of graphs. This is probably due to a large search space in the first phase of the query. This becomes even more pronounced for grid graphs as shown in the next paragraph. The economic variant of HiFlags decreases the query times by a factor of about two to three without increasing the preprocessing time and memory requirements by much. By using the generous variant, query times can be improved by about one order of magnitude but the preprocessing times also increase substantially by about 10 to 20 times, as does the space consumption by about seven to ten additional bytes per node.

	— Core —		— Prepro —		— Query —	
	nodes	edges	[s]	[B/n]	settled	[μs]
Timetable – Railway EU						
Highway Node Routing	1 192 736	3 954 967	486	2.53	376	310.4
core-size 0.5%	5 964	151 552	536	6.60	229	196.1
core-size 5.0%	59 637	791 750	1 743	23.87	71	45.6
Timetable – VBB						
Highway Node Routing	2 599 953	7 737 971	1 636	-0.19	416	402.3
core-size 0.5%	13 000	391 660	2 008	4.64	125	120.6
core-size 5.0%	129 998	1 554 216	10 456	19.04	49	37.7
Timetable – RMV						
Highway Node Routing	2 277 812	7 894 162	2 584	3.73	546	583.2
core-size 0.5%	11 390	402 816	2 863	9.40	244	246.8
core-size 5.0%	113 891	1 834 072	11 946	29.59	77	62.7

Table 6.26: *Experimental results of the Hierarchy-aware ArcFlags algorithm on three different time-expanded timetable information systems. An economic variant (0.5% core) and a generous variant (5.0% core) are shown.*

	— Core —		— Prepro —		— Query —	
	nodes	edges	[s]	[B/n]	settled	[μs]
2-dim. Grid Graph						
Highway Node Routing	250 000	995 373	70	-0.08	418	293
core-size 0.5%	1 250	28 226	73	3.54	274	213
core-size 5.0%	12 500	184 876	124	23.68	101	69
3-dim. Grid Graph						
Highway Node Routing	250 047	1 939 138	13 567	14.80	2 177	6 603
core-size 0.5%	1 251	114 610	13 585	29.48	2 836	10 165
core-size 5.0%	12 503	764 222	14 632	112.70	772	1 292
4-dim. Grid Graph						
Highway Node Routing	244 904	2 762 507	133 734	29.12	14 501	60 049
core-size 0.5%	1 225	42 002	133 741	34.62	30 848	130 996
core-size 5.0%	12 246	585 688	135 630	105.75	29 811	124 801

Table 6.27: *Experimental results of Hierarchy-aware ArcFlags on two-, three- and four-dimensional grid graphs. Two variants, an economic one (core-size 0.5%) and a generous one (core-size 5.0%) have been analyzed.*

Grid Graphs. As can be seen in Table 6.27, the performance of HiFlags on grid graphs is not very good. Useful results are only obtained for the two-dimensional grid. The main problem of the two other graphs seems to be a deficient search graph. The Contraction Hierarchy algorithm apparently is not able to produce viable results for higher-dimensional grids, at least for the applied parameter set. Furthermore, the preprocessing slows down considerably towards the end of the contraction. The resulting search graphs are even larger than the original graphs with regards to the number of edges. This is reflected in the extremely long query times of Highway Nodes Routing. Using the HiFlags algorithm instead does not increase the preprocessing times by much, but the query times do not always improve. In particular, the results are very bad for the four-dimensional grid, since the query takes about twice as long with Hierarchy-aware ArcFlags than by using Highway Node Routing. This is due to the first phase of the HiFlags query having to exhaustively settle all entry points into the core before continuing with the second phase. On dense graph, this can easily amount to a lot of nodes that have to be settled. For the four-dimensional graph, 27 195 nodes are settled and 9 383 entry points are found during the first phase of the query.

Local Queries. A comparison of Highway Node Routing and the generous variant of Hierarchy-aware ArcFlags on the European road network with travel times is shown in Fig. 6.4 and 6.5, using the Dijkstra Rank representation. A locality optimization is used for the queries of both algorithms. Up to a Dijkstra Rank of about 2^{15} both speed-up techniques perform similarly with HiFlags being a bit slower due to its additional overhead. For even longer queries the impact of using ArcFlags on the core begins to show. The longer they get, the larger the fraction of the query becomes on which ArcFlags are used. Thus, the effect of the ArcFlags gradually increases and the query times become shorter and shorter. The Fig. 6.5 further emphasizes the growing speed-up of the HiFlags algorithm compared to Highway Node Routing. Here, it can

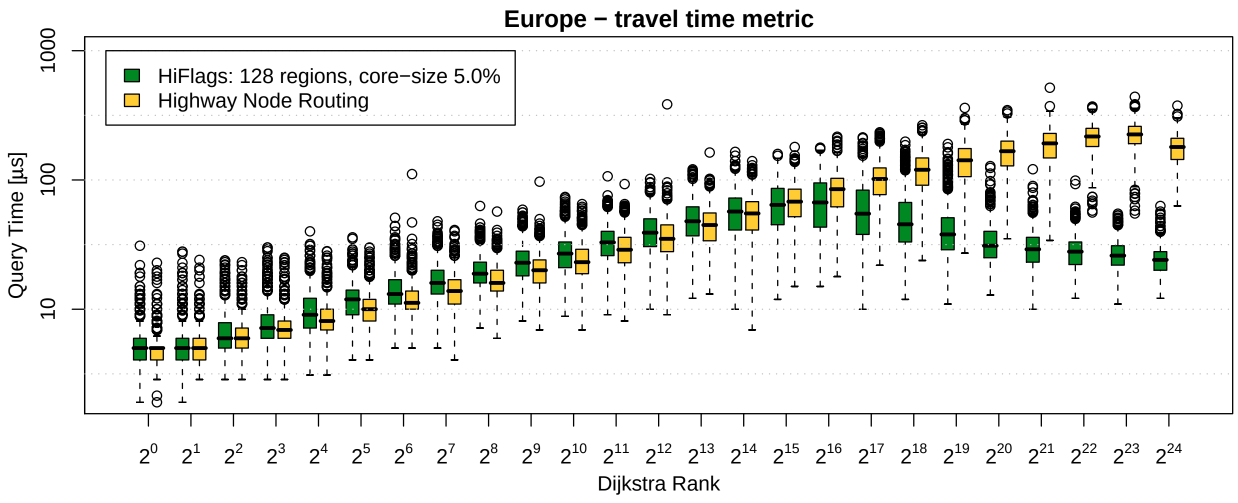


Figure 6.4: *Comparison of the performance of Highway Node Routing and Hierarchy-aware ArcFlags (5.0% core) on the road network of Europe with a travel time metric, using the Dijkstra rank methodology [SS05].*

also be observed, that the variance in the query times becomes considerably smaller and the outliers get less pronounced, the more the ArcFlags take effect, resulting in a more reliable query performance.

In addition, the performance of the economic and the generous variant of the Hierarchy-aware ArcFlags algorithm are compared in Fig. 6.6 and 6.7. For small query distances both variants need about the same average amount of time. More time is needed with growing distances until the ArcFlags start to contribute. This happens at smaller Dijkstra Ranks for the generous variant, since its core is larger and ArcFlags can be used earlier. The generous variant stays faster than the economic one, but the difference in query times becomes almost constant for the longest query distances. This is due to both variants performing equally well in each phase of the query, but the economic one staying longer in the first phase, which takes more time.

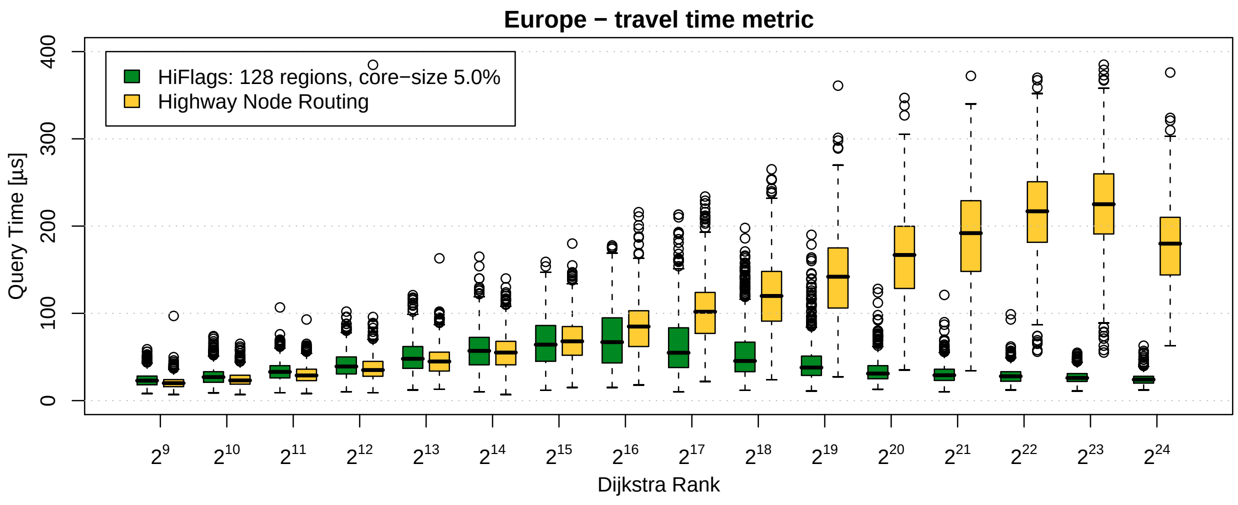


Figure 6.5: *Same data set as in Figure 6.4 but using a linear scale for the query time axis.*

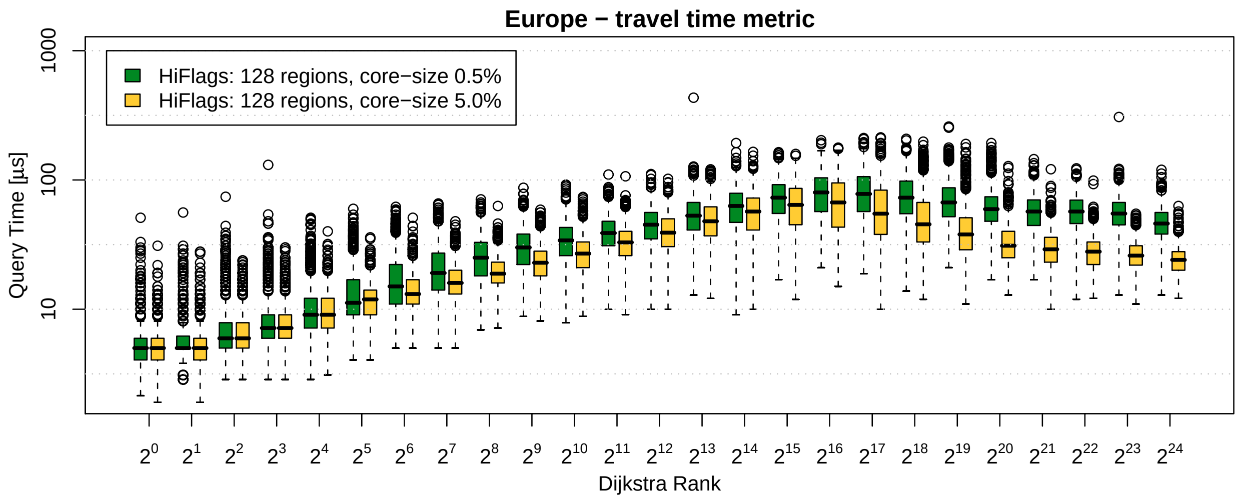


Figure 6.6: *Comparison of two variants of the HiFlags algorithm. One uses a core-size of 0.5% (economy variant) and the other one a core-size of 5.0% (generous variant). The results are presented as a Dijkstra rank plot [SS05].*

Core Sizes. Table 6.28 lists additional results of the HiFlags algorithm, concentrating on the impact of using different core-sizes on the query. The European road network with a travel time metric has been chosen for these analyses since the parameters provided for the Contraction Hierarchies algorithm are optimized for this type of graph. The queries have been performed using a locality optimization, switching off the *unroll-loops* compiler flag and applying further optimizations to the query code (smaller data types, extensive use of call-by-reference, ...). Two sets of data have been taken, one using stall-on-demand (see Sect. 2.3.5) and one with this technique switched off. As shown, the query performance can be adjusted precisely to one's needs by varying the core-size. Additional speed-up comes at the cost of longer preprocessing times and a larger memory overhead. Only up to a core-size of about 0.5% the required space stays less than needed for the full graph without preprocessed information and with just two

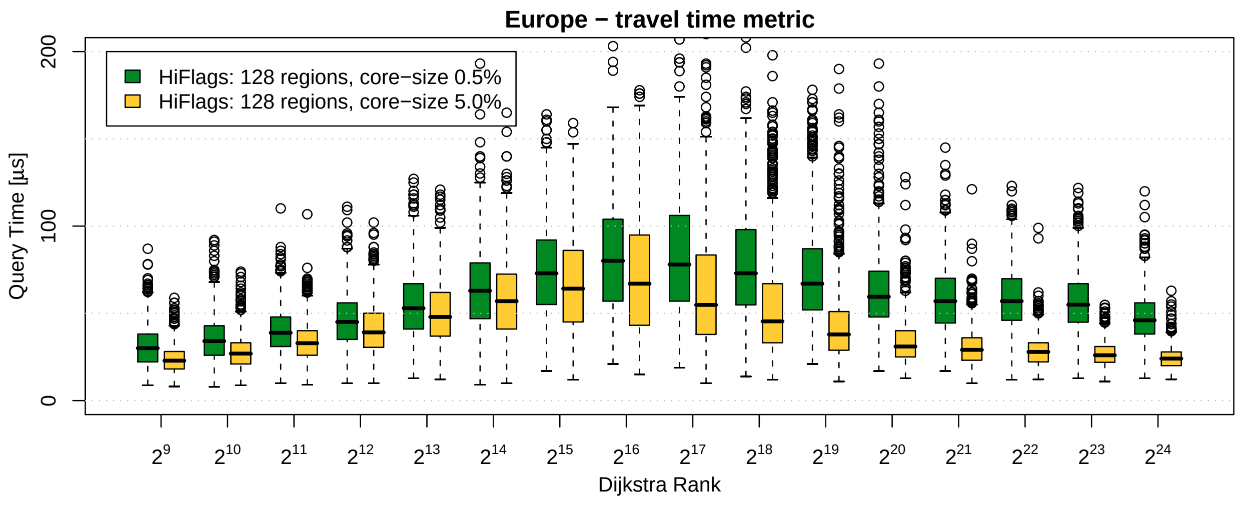


Figure 6.7: *Same data set as in Figure 6.6 but using a linear scale for the query time axis.*

	— Core —		— Prepro —		— Query — (with s.o.d.)		— Query — (without s.o.d.)	
	nodes	edges	[s]	[B/n]	settled	[μs]	settled	[μs]
EU – travel time metric								
Highway Node Routing	18 010 173	38 317 936	1 515	-2.72	355	180.0	931	286.5
core-size 0.5%	90 051	1 501 868	1 888	-0.03	86	48.8	111	44.2
core-size 1.0%	180 102	2 579 634	2 442	1.90	67	36.4	78	30.6
core-size 2.0%	360 204	4 271 378	3 704	4.94	54	28.9	59	22.9
core-size 5.0%	900 509	8 239 136	5 934	12.07	43	22.8	45	17.4
core-size 10.0%	1 801 018	13 859 566	14 659	22.16	37	19.6	39	14.8
core-size 20.0%	3 602 035	23 509 130	32 148	39.52	34	17.4	35	12.9

Table 6.28: *Further results of the Hierarchy-aware ArcFlags algorithm on the European road network with travel times. The effects of additional core sizes and the impact of the stall-on-demand (s.o.d.) technique are analyzed. Locality optimization and further code tuning have also been applied to increase the query performance.*

hours the preprocessing time stays reasonable up to a core-size of about 5.0%. If necessary, the query performance can be improved down to just 12.9 μs on average using a core-size of 20%, making the Hierarchy-aware ArcFlags algorithm the fastest traditional technique for finding shortest-paths in road networks. It can also be observed, that the stall-on-demand technique is actually impeding the performance of the query due to the additional overhead. Thus, on queries with very few settled nodes, it is better to switch off stall-on-demand even though the overall number of settled nodes increases slightly. Apparently, it does not pay off to apply the stall-on-demand technique to Hierarchy-aware ArcFlags. The number of nodes of the contracted shortest paths that are computed averages to about 21.8. This implies, that the most aggressive variant of the HiFlags algorithm has to visit only 12 nodes or about one third more nodes than absolutely necessary.

Entry Points. Further information about the first phase of the query for all analyzed core-sizes can be found in Table 6.29. The number of entry points into the core and number of nodes that are settled in this phase are listed. As expected, both values decrease with larger core-sizes.

	— Query — (with stall-on-demand)			— Query — (without stall-on-demand)		
	settled overall	settled phase 1	entry points	settled overall	settled phase 1	entry points
EU – travel time metric						
core-size 0.5%	86	60	21	111	76	30
core-size 1.0%	67	41	14	78	47	18
core-size 2.0%	54	29	10	59	31	12
core-size 5.0%	43	18	7	45	19	8
core-size 10.0%	37	13	5	39	13	6
core-size 20.0%	34	8	4	35	8	4

Table 6.29: *Additional information about the first phase of the HiFlags query, regarding the number of settled nodes and the number of entry points into the core. Results are shown with and without using stall-on-demand.*

Surprisingly, the number of nodes that are settled in the second phase almost stays constant, at least when using stall-on-demand. This implies that a reasonable lower bound for the number of settled nodes for even larger core-sizes than 20% is given by their difference in the first phase and for the whole query. Here, this amounts to 26 nodes (only four nodes short of the optimum), meaning that the performance of the query can be increased by about an additional 25% at most.

Number of Regions. By using less than 128 regions to compute the ArcFlags, the query performance should not deteriorate by much, since the graph-cores are small, in general, but the memory usage improves considerably. Preliminary experiments indicate query times of 47.9 μs and 31.4 μs on the European road network with travel times, using the generous contraction and 16 or 64 regions, respectively. For the VBB timetable network 82.1 μs and 45.6 μs are obtained. The results for 64 regions are only slightly worse than with 128 and using only 16 regions still performs better than the economic variant with 128 regions. The main benefit, however, is the small memory overhead of -1.7 and 1.1 bytes/node for Europe and 4.7 and 1.1 bytes/node for the VBB timetable network. Unfortunately, the preprocessing times increase in all cases. This is due to the concrete implementation of the centralized ArcFlags preprocessing routine used, which has problems with regions that feature a large number of boundary nodes.

ArcFlags Compression. Another way to improve the memory usage of the algorithm is to apply an ArcFlags compression strategy, as described in Sect. 5.3 and applied by [Hil07, BD08]. Preliminary analyses for HiFlags indicate an average increase in query times by about 10%. Note that the query times even improve on small graphs like on the road network of the Netherlands. This is probably due to the small number of different ArcFlag labels. Thus, the whole ArcFlags lookup-table can be stored in the cache, reducing access times by a large amount.

Summary. Hierarchy-aware ArcFlags turns out to be an extremely potent speed-up technique for sparse graphs. Not only are the query times among the fastest currently achieved, the preprocessing times and the memory overhead are also extremely low. Comparing the performance of Highway Node Routing to HiFlags, while both of them use the same hierarchy, HiFlags turns out to be more robust. Whereas the query times of Highway Node Routing vary greatly by just applying different metrics, the results of Hierarchy-aware ArcFlags stay closer together (see Table 6.23). One major drawback of Hierarchy-aware ArcFlags on dense graphs is the expensive computation of the hierarchy. To improve upon this bottleneck, the applied Contraction Hierarchies could be terminated prematurely if the contraction steps get to slow. This usually happens for the last 10 - 20% of nodes on dense graphs. ArcFlags are then applied at least these remaining nodes to speed-up the search on the top-most level.
A particularly noteworthy advantage of Hierarchy-aware ArcFlags is the potential to precisely adjust the memory overhead and the preprocessing times of the algorithm through the core-size, leading either to a faster query or to a more economic one. Core-sizes of 0.5% and 5.0% have turned out to be good values for these particular goals.

6.4 Comparison

An overview of the performances of the speed-up techniques introduced in this thesis is given in Table 6.30. Here, they are compared to several existing techniques, grouped by hierarchical methods, goal-directed methods and previous combinations. Where applicable, the performance of the actual implementation used in this thesis is given as well as the results of the original publication. As previous comparisons have shown, the different experimental setups perform about equally so that the obtained times can be roughly compared to each other. Note that the presented results for this thesis have been obtained without applying further optimizations like exploiting locality effects. This would increase the listed performances by an additional 10 - 20% as shown for Hierarchy-aware ArcFlags in Sect. 6.3.4.

The table only lists the results for the road networks of Europe and the USA using travel times, or more generally speaking, for sparse graphs with an underlying hierarchy. The performance of the algorithms varies considerably for other types of graphs as can be seen in the previous tables in this chapter. As a rule of thumb, one can say that Transit Node Routing performs best on road networks, Hierarchy-aware ArcFlags on other sparse graphs and CALT on dense graphs up to a certain point, where simpler algorithms are to be preferred.

General Findings. A more general but still noteworthy observation concerns the query times on the European road network, on the one hand, and the road network of the USA, on the other. Goal-directed techniques seem to produce better results on the European road network, whereas the hierarchical techniques favour the road network of the USA. One particular difference between these two graphs are long-distance connections that only exist for Europe, in the form of ferry crossings. Apparently, hierarchies have problems incorporating them into their level structure, while goal-direction can directly exploit such edges that cover large parts of the graph. Regarding combinations of the two approaches, no clear statement can be made. Some favour the one graph, some the other.

CALT. Both variants of the CALT algorithm take a remarkable position in this field of speed-up techniques. Although they only surpass some of the older techniques and are surpassed by all of the newer ones and all combinations, their performance is still noteworthy. In particular, if the required preprocessing time and the memory overhead of CALT is considered. For example, REAL is about two times faster than CALT-a64 but its preprocessing takes about nine times longer and its memory usage is about four times higher. Regarding memory consumption in general, only speed-up techniques based on Highway Node Routing require less additional memory. This is due to them not storing the whole graph with all edges, but only a smaller search graph. Note that this technique could also be applied to CALT, reducing its space consumption further. Finally, consider that CALT will probably be easily adaptable for dynamic graph. In total, this makes CALT a quite remarkable technique.

Partial ArcFlags. Comparing the two introduced implementations of the Partial ArcFlags approach, Reach-aware and Hierarchy-aware ArcFlags, it can be observed, that the possible speed-ups gained by adding ArcFlags to another technique are smaller for the Reach algorithm than they are for Highway Node Routing. Regarding the economic variant, the respective speed-ups are 2.79 and 3.91. Regarding the generous variant, they amount to 8.21 and 8.93, respectively. These observations are inverted when analyzing the impact of adding ALT to these techniques instead of ArcFlags. The REAL algorithm yields a speed-up of 3.13 compared to only applying the Reach algorithm, whereas Highway Hierarchies* only performs about 1.24 times better than Highway Hierarchies alone. The given ratios are based on the query times on the European road network with travel times.

The combination of Highway Hierarchies and ALT yields only small speed-ups compared to Highway Hierarchies alone for several reasons. The most significant one is its complex stopping condition. In principal, both Highway Hierarchies and Highway Node Routing require the search to continue well after the search spaces in both directions have met. But apparently, this is more detrimental for ALT than ArcFlags, since the latter benefits from pruning edges to considerably decrease the search space.

Regarding the speed-ups of Reach-aware and Hierarchy-aware ArcFlags compared to their respective hierarchical base algorithms, the differences between both approaches become less the more ArcFlags can contribute, i. e. the larger the core gets. This effect has also been noted for a single combination with regards to different graphs in Sect. 6.3.4. By applying ArcFlags to a (hierarchical) query the existing differences in query times become less pronounced.

HiFlags vs. SHARC. Note that even the economic variant of HiFlags clearly outperforms the bidirectional implementation of the recent SHARC Routing technique in every single respect: Preprocessing and queries are faster and the memory overhead is smaller. But take note that the special strength of SHARC Routing is its unidirectional variant that is only about two to three times slower than the bidirectional variant.

HiFlags vs. Transit Node Routing. Hierarchy-aware ArcFlags also closes the gap to the different variants of Transit Node Routing. Economic HiFlags is about as fast as the grid-based implementation of Transit Node Routing but has considerably lower preprocessing efforts. The generous variant even is about twice as fast, still retaining lower preprocessing efforts. Note that the best implementation of generous HiFlags is about another 50% faster than the numbers shown here (see Sect. 6.3.4 for reference). Thus, Hierarchy-aware ArcFlags is the fastest speed-up technique for road networks with the exception of the two hierarchy-based implementations of Transit Node Routing (HH-TNR, CH-TNR). But their advance in performance comes at high preprocessing costs whereas Hierarchy-aware ArcFlags only has very moderate preprocessing requirements in comparison.

Implementation Differences. Comparing the presented measurement results, the following differences can be noted between the original implementations and the applied implementations of several speed-up techniques:

- The Reach implementation by Goldberg et al. performs about 1.5 to 2.0 times faster than the implementation used here. This also affects the performance of the Reach-aware ArcFlags algorithm, which builds upon this inferior implementation of the Reach algorithm.
- Two effects add up and produce a large discrepancy in query times between CH-HNR by Geisberger et al. and CH-HNR of this thesis. In particular, their experimental setup performs about 10% faster than the one used here and they applied further optimization techniques which adds another performance gain by 10 - 20%.
- The difference in query times between ALT-m16 by Delling and ALT-m16 analyzed here primarily results from an upgrade of the experimental setup. The actual implementation is exactly the same for both experiments. The small deviation in the number of settled nodes is due to different sets of landmarks that have been produced by the maxCover algorithm.
- The preprocessing times of the ArcFlags algorithm by Hilger differ greatly from the ones obtained in this thesis. This is due to him applying the more potent Centralized ArcFlags algorithm, while here; the older boundary algorithm had to be used for the preprocessing. The available implementation of the Centralized ArcFlags algorithm still has some problems with memory consumption on large graphs. Hilger's space overhead is also smaller, even though he is using 1 000 regions instead of only 128 since he already applies an ArcFlags compression.

		Europe – travel times				USA – travel times			
		— Prepro —		— Query —		— Prepro —		— Query —	
		[min]	[B/n]	settled	[ms]	[min]	[B/n]	settled	[ms]
Hierarchical Techniques									
RE	[GKW07]	83	17	4 643	3.47	44	20	2 317	1.81
RE	Sect. 6.3.3	70	21	7 387	6.24	62	18	4 261	3.90
HH	[Sch07]	13	48	709	0.61	15	34	925	0.67
HH-HNR	[Sch07]	15	2.4	981	0.85	16	1.6	784	0.45
CH-HNR	[GSSD08]	30	-2.7	368	0.16	32	-2.3	303	0.11
CH-HNR	Sect. 6.3.4	25	-2.7	355	0.25	27	-2.3	278	0.19
grid-TNR	[BFM$^+$07]	-	-	-	-	1200	21	n/a	0.063
HH-TNR	[BFM$^+$07]	164	251	n/a	0.0056	205	244	n/a	0.0049
CH-TNR	[GSSD08]	112	204	n/a	0.0034	90	220	n/a	0.0030
Goal-Directed Techniques									
ALT-a16	[GKW07]	13	70	82 348	160.34	19	89	187 968	400.51
ALT-m16	[DW07]	85	128	74 669	53.6	103	128	180 804	129.3
ALT-m16	Sect. 6.3.2	83	128	76 621	50.8	119	128	179 178	113.3
ALT-a64	Sect. 6.3.2	92	512	26 630	18.4	116	512	71 880	48.9
AF-1000	[Hil07]	2 156	25	1 593	1.1	1 419	21	5 522	3.3
AF-128	Sect. 6.3.1	11 789	81	2 764	0.80	-	-	-	-
Previous Combinations									
REAL	[GKW07]	141	36	679	1.11	121	45	540	1.05
HH*	[Sch07]	14	72	511	0.49	18	56	627	0.55
SHARC	[BD08]	192	20	145	0.091	158	21	350	0.18
TNR + AF	[BDS$^+$08]	229	321	n/a	0.0019	157	263	n/a	0.0017
New Combinations									
AALT eco.	Sect. 6.3.1	2 551	140	4 932	2.82	-	-	-	-
AALT gen.	Sect. 6.3.1	11 887	593	1 613	0.85	-	-	-	-
CALT-m16	Sect. 6.3.2	16	8	2 878	4.6	26	8	7 093	10.3
CALT-a64	Sect. 6.3.2	14	20	1 394	2.4	21	19	3 240	5.8
ReachFlags eco.	Sect. 6.3.3	107	25	2 797	2.24	123	21	1 844	1.33
ReachFlags gen.	Sect. 6.3.3	229	30	1 168	0.76	318	25	1 636	1.02
HiFlags eco.	Sect. 6.3.4	32	0.0	86	0.064	36	-0.8	93	0.067
HiFlags gen.	Sect. 6.3.4	99	12.0	43	0.028	228	11.1	46	0.030

Table 6.30: *Overview of the performance of several recent speed-up techniques on the road network of Western Europe and the USA, both using a travel time metric. They are grouped by hierarchical methods [Reach (RE), Highway Hierarchies (HH), Highway Node Routing applying Highway Hierarchies (HH-HNR) or Contraction Hierarchies (CH-HNR), grid-based Transit Node Routing (grid-TNR), Transit Node Routing applying Highway Hierarchies (HH-TNR) or Contraction Hierarchies (CH-TNR)], goal-directed methods [A*-search with 16 or 64 avoid landmarks (ALT-a16, ALT-a64) and 16 maxCover landmarks (ALT-m16), ArcFlags with 128 or 1 000 regions (AF-128, AF-1000)], previously established combinations [REAL, Highway Hierarchies* (HH*), SHARC Routing (SHARC), Transit Node Routing with ArcFlags (TNR + AF)] and the new combinations introduced in the thesis at hand.*
The economic AALT variant uses 16 regions and 16 landmarks (AALT eco.), whereas the generous variant uses 128 regions and 64 landmarks (AALT gen.). Both CALT queries use ($c = 3.0$, $h = 30$) as contraction parameters and 16 maxCover (CALT-m16) or, respectively, 64 avoid landmarks (CALT-a64). For Reach-aware ArcFlags, using two iterations of the Reach preprocessing is denoted as economic variant (ReachFlags eco.) and three iterations as generous variant (ReachFlags gen.). Both variants of the Hierarchy-aware Arcflags algorithm use 128 regions and a core-size of 0.5% (HiFlags eco.) or 5.0% (HiFlags gen.), respectively.

Conclusion and Outlook

This final chapter gives a short conclusion on the most important results obtained by the experimental studies that have been performed within the scope of this thesis. Also, a perspective for further work in this field of research is presented, trying to suggest possible directions that are worth to be taken in the future.

Conclusion

As already stated by Holzer et al. in [HSW04], a combination of goal-directed techniques and hierarchical methods leads to a considerable speed-up of the resulting query compared to the individual algorithms, whereas combining similar techniques does not turn out to be very viable. Using goal-direction is usually better for dense graphs, while hierarchies typically perform better on sparse graphs. By combining both approaches, the different capabilities of both of them can be exploited, sometimes leaning more towards denser graphs and sometimes more towards the other direction. Furthermore, the experimental results show that applying the goal-direction just to some higher level of the hierarchy is not only feasible but also proves to be very beneficial. Preprocessing times and memory overhead of the goal-directed technique are cut down by a considerable amount while the query times of the combined algorithm stay short.

It turned out that all of the presented speed-up techniques that combine hierarchical and goal-directed methods apply the same general approach to their query algorithms. Each of them uses a two-tiered setup of some kind. In the first phase of the query, they try to find paths into a core of some sort, applying only the hierarchical approach. Goal-direction is then added only to the second phase, which is performed on said core.

Regarding REAL and Reach-aware ArcFlags on the one hand, and Highway Hierarchies* and Hierarchy-aware ArcFlags on the other, it has been experimentally shown that the ALT algorithm performs better in combination with the Reach algorithm than together with the other hierarchical techniques. This observation is reversed for the ArcFlags algorithm, which yields much better results together with hierarchical techniques based on highway structures (or more recently contraction-based hierarchies) than based on reach values.

Overall, two of the combinations analyzed in this thesis stand out in particular. First, there is Hierarchy-aware ArcFlags, which is one of the fastest known speed-up techniques. Especially on sparse graphs with an underlying hierarchy, i. e. road networks, its combined performance in both fields, preprocessing costs and query times is superior to any other technique currently available. Secondly, there is the CALT algorithm, which is a quite simple technique but turned out to be surprisingly fast for its complexity and required preprocessing costs. Furthermore, CALT is also very robust, and thus, more suited for dense graphs than Hierarchy-aware ArcFlags.

Outlook

All of the introduced speed-up techniques could probably be tuned a little further, but the following directions seem to be much more interesting and promising for future research projects:

Although the results obtained by the AALT algorithm did not turn out to be very good overall, they have shown some promise, especially with regards to denser graphs. The fact that ALT supports the ArcFlags query in the middle, where the search spaces in both directions meet and vice-versa could be exploited further, e. g. by introducing ArcFlags to the CALT algorithm. Here, its two main drawbacks, long preprocessing times and a large memory overhead, would be alleviated due to the smaller core, on which the data is processed. Furthermore, initial studies in [Paj08] have shown very promising results for applying the basic ideas of the AALT algorithm to queries on timetable networks in a more sophisticated manner. Thus, additional work in this area might prove to be worthwhile, yielding further useful insights.

The CALT algorithm might not be the fastest combination introduced but, as has already been stated, it is a very robust and very simple technique. These two attributes render CALT to be very promising for several other applications that require shortest paths. In particular, queries on dynamic or time-dependent graphs could probably profit from this new technique. Goldberg already showed in [GH04] that the ALT algorithm can be easily adapted for the use with dynamic graphs and the outlook for also adapting the graph contraction is optimistic. Further enhancements to the already good performance on static graphs could possibly be gained by revisiting the preprocessing, i. e. the contraction step, and trying to improve it.

The development of Hierarchy-aware ArcFlags is probably near the end of the line. But as the transition from using Highway Hierarchies to Contraction Hierarchies for building hierarchy levels of a graph has shown, there is still room for further improvement in this area. In particular, trying to devise hierarchies that perform more evenly for different types of graphs, without having to adjust many parameters, would provide a nice boost. In a first step, it could be attempted to carry over some of the performance on sparse graphs to denser graphs by stopping the contraction before it gets too slow and only using ArcFlags to speed-up the search on the remaining graph.

Furthermore, using hierarchies that are aware of the graph partition used by ArcFlags might also prove to be beneficial for the preprocessing and the query. Finally, adapting SHARC Routing for this approach instead of just ArcFlags could also be a worthwhile study.

To summarize, adapting the discussed speed-up techniques for new tasks and revisiting the contraction algorithms seem to be the two most promising directions to take. While examining further combinations might also yield some interesting results, this should not be given the same priority as the two directions stated before.

Data Structures

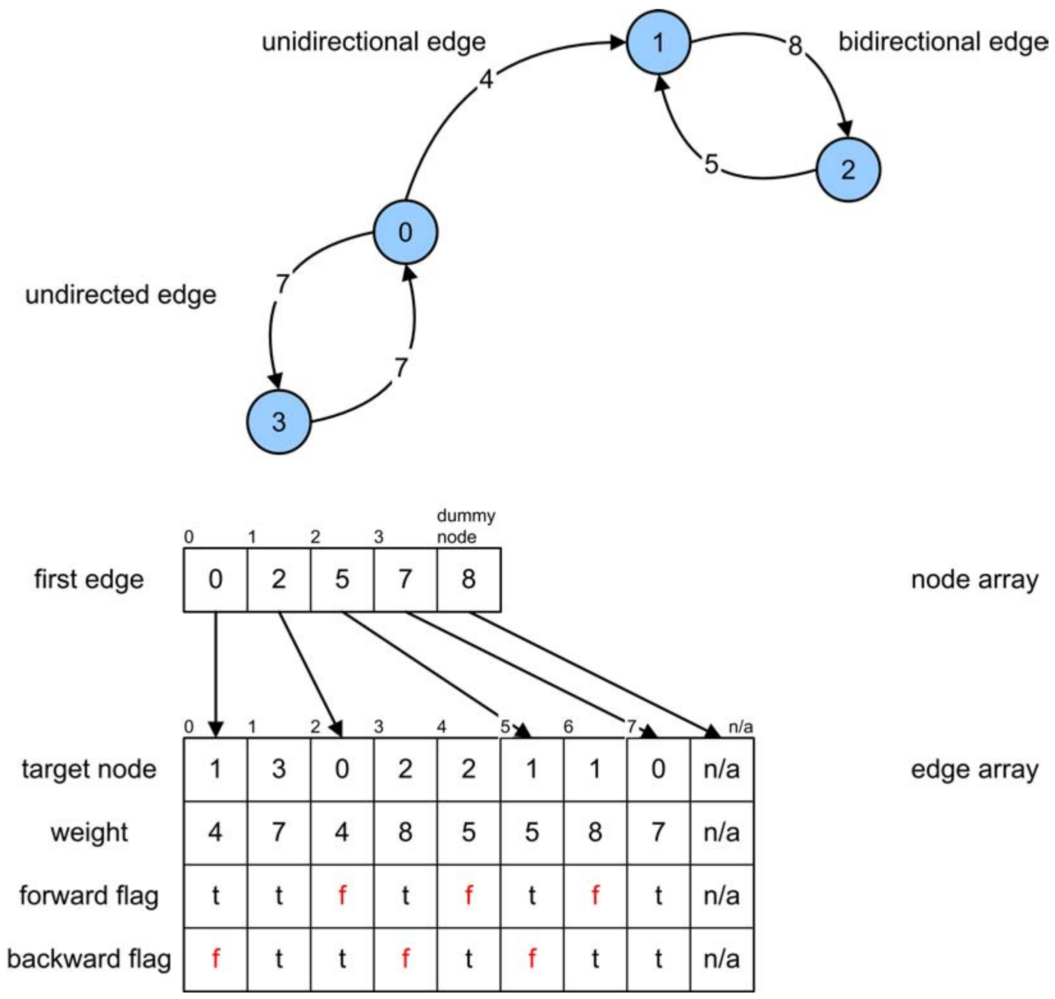
This appendix gives an overview of the data structures used by the algorithms. Static graphs are used throughout the query phase and for some of the preprocessing routines, since here, the graph stays constant. During the preprocessing, dynamic graphs are also used whenever the graph structure needs to be changed by the algorithm. At the end, the core of each shortest path algorithm based on Dijkstra's approach, the priority queue, is presented.

The contents of this appendix is primarily based on previous working notes by Delling [Del07].

A.1 Static Graph

The graph format is based on the *forward star representation* of a graph [BBK03]. Basically, there are 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*.
This 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 Dijkstra search, this shortcoming has to be remedied. Therefore, each edge is stored twice: Once at its source node and once at its target node. 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. A small sample graph and its associated data structure are shown in Fig. A.1.

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 be directly added by extending the respective template. Be careful though, adding too much data to the structs does have a negative impact on the performance. Smaller entries can be stored and retrieved faster. They also profit more from caching effects since further entries fit into the CPU cache.



	0	1	2	3	dummy node
first edge	0	2	5	7	8

	0	1	2	3	4	5	6	7	n/a
target node	1	3	0	2	2	1	1	0	n/a
weight	4	7	4	8	5	5	8	7	n/a
forward flag	t	t	f	t	f	t	f	t	n/a
backward flag	f	t	t	f	t	f	t	t	n/a

Figure A.1: *A sample graph, covering all possible sorts of edges, is depicted above. The corresponding data structure, in the form of a node array and an edge array, is shown below. The indices of both arrays are written above each element. The entries of each element are arranged vertically in columns. The entries t indicate a true value, f indicates a false value, respectively. The abbreviation n/a denotes that this element is not valid.*

The *staticGraph* template class provides some means to manage this data structure. Mainly, access methods for and iterator methods on the nodes and edges are made available. The user stays responsible for providing and maintaining a consistent representation of the graph. To ease this task, an external function is available in */shortestPathBase/algorithms* to check the consistency of a graph.

File Format. Parsing the standard GraphML format takes much too long on large graphs. Therefore, a better file format, *binary graph* (bgr), has been devised. Using this format, it only takes several seconds to read and write even a huge graph with several millions of nodes and edges. Note that edges are also stored twice on disk as they are in memory. Several converters are provided in */ssspTools/converters* to allow for transformations between various graph formats (bgr, graphml, . . .).

A file, written in the *binary graph* format, is structured as follows:

- **31 bit:** 0
- **1 bit:** 1
- **32 bit:** number of nodes
- **32 bit:** number of edges

- **n · 64 bit:** n nodes:
  - **32 bit:** x coordinate
  - **32 bit:** y coordinate

- **m · 96 bit:** m edges:
  - **31 bit:** ID of source node
  - **1 bit:** forward flag
  - **31 bit:** ID of target node
  - **1 bit:** backward flag
  - **32 bit:** edge weight

The leading 32 bits represent the header of the file format. It can be altered to indicate future revisions. IDs of nodes and edges correspond to their indices in the array, and thus, are given implicitly. The forward and backward flag indicate whether the edge runs from the source node to the target node, or the other way round. The x and y coordinates represent the graph layout. If no graph layout is given, they can be filled with zeros.
Note that the bgr-format currently does not use any form of data-compressing techniques.

A.2 Dynamic Graph

The data structure for static graphs, described above, is extended by dynamic aspects with the goal to eventually switch to this data structure for all graphs. In principal, the edge array is made larger than necessary for holding all edges and filled with empty slots. By default, the array is made twice as large and empty slots are inserted after the filled slots of each node (the same number of empty slots is inserted as there are filled slots at each node). The actual overhead can be controlled by the variable *slotFactor* and can be changed whenever necessary. Furthermore, the exact number of empty slots of each node can also be adjusted directly. If there are no empty slots left at a node, the complete array is automatically reallocated, which may take a long time. Thus, one should know in advance how much overhead is needed at every node and allocate an appropriate amount.
In addition to the methods already provided by the *staticGraph* class, further ones to aid in altering the graph structure (removing nodes and edges, inserting edges, bypassing nodes) and to compress the data structure are provided by the *dynamicGraph* class.

A.3 Priority Queue

A binary heap is used as data structure for the priority queue. The current implementation was originally provided by Dominik Schultes. The heap is realized as an array. It functions as static storage space for all elements ever added to the data structure. No element is ever removed, only marked as deleted. Heap elements are implemented as structs that can hold additional data. Usually, a heap element contains a *nodeID*, the index of stored node, and a *preNodeID*, the index of the predecessor of the node in the current search graph. The latter can be used to reconstruct one shortest path after a search has finished since all nodes added to the priority queue during the query are still stored in the heap array.
The keys of the elements are stored separately. Only they are reordered according to the heap condition; the ordering of the elements never changes. This improves the execution time of operations on the heap since only a small key instead of the whole element has to be moved in memory. But because of this separation, the index of an element within the heap array has to be known prior to being able to access it. This index is called the *count* of a node and is stored within the node-array of the graph data structure. It has to be updated every time the priority key of the node changes.

Recently, the binary heap has been extended by Delling to a fourary heap. This implementation yields improvements for queries with larger heap sizes. But for smaller ones, the gain is outweighed by the additional overhead. A bucket heap, as suggested by Goldmann in [GS95] and used ever since, is currently also being implemented by Karch.

Proofs

This appendix provides further proofs regarding the feasibility of the potentials, defined by the lower bounds that have been introduced for the ALT algorithm and the CALT algorithm.

B.1 ALT

Assumption:
Lower bounds, obtained by using one landmark and the triangle inequality for graphs as described in Sect. 2.3.2, yield a potential function that is feasible.

Proof:
Let $G = (V, E)$ be a graph with a node-set V and an edge-set E. A weight function $w(\cdot)$ assigns values to each edge. Let (u, v) be an arbitrary edge $e \in E$ with $u, v \in V$. Furthermore, let t be an arbitrary node $\in V$.

A potential function $\pi(\cdot)$ is called feasible if the reduced edge weight $w_\pi(\cdot)$ is equal to or larger than zero. It is defined as follows, with $w(u, v)$ being the normal edge weight of (u, v):

$$w_\pi(u, v) = w(u, v) - \pi(u) + \pi(v) \geq 0$$

The potential function is defined by the lower bounds, obtained with an arbitrary landmark $L \in V$, as defined in Sect. 2.3.2. Here, landmark distances to L are used w. l. o. g.:

$$\begin{aligned} d(u, t) &\geq d(u, L) - d(t, L) = \pi(u) \\ d(v, t) &\geq d(v, L) - d(t, L) = \pi(v) \end{aligned}$$

Inserting them into the equation above yields:

$$\begin{array}{lll} & w(u, v) - \pi(u) + \pi(v) & \geq 0 \\ \Leftrightarrow & w(u, v) - d(u, L) + d(t, L) + d(v, L) - d(t, L) & \geq 0 \\ \Leftrightarrow & w(u, v) - d(u, L) + d(v, L) & \geq 0 \\ \Leftrightarrow & w(u, v) + d(v, L) & \geq d(u, L) \end{array}$$

With $w(u, v) \geq d(u, v)$, this is the triangle inequality for graphs, which is proven to be true.

q.e.d.

B.2 CALT

Assumption:
Lower bounds, obtained by using a proxy-node in addition to one landmark and the triangle inequality for graphs as described in Sect. 4.3, yield a potential function that is feasible.

Proof:
Note, that the proof is performed in a similar way to the one for the ALT algorithm in App. C.

Let $G = (V, E)$ be a graph with a node-set V and an edge-set E. A weight function $w(\cdot)$ assigns values to each edge. Let (u, v) be an arbitrary edge $e \in E$ with $u, v \in V$. Furthermore, let t be an arbitrary node $\in V$ an let $t' \in V$ be its proxy-node.

A potential function $\pi(\cdot)$ is called feasible if the reduced edge weight $w_\pi(\cdot)$ is equal to or larger than zero. It is defined as follows, with $w(u, v)$ being the normal edge weight of (u, v):

$$w_\pi(u, v) = w(u, v) - \pi(u) + \pi(v) \geq 0$$

The potential function is defined by the lower bounds, obtained with an arbitrary landmark $L \in V$, as defined in Sect. 4.3. Here, landmark distances to L are used w. l. o. g.:

$$\begin{aligned} d(u, t) &\geq d(u, L) - d(t', L) - d(t, t') = \pi(u) \\ d(v, t) &\geq d(v, L) - d(t', L) - d(t, t') = \pi(v) \end{aligned}$$

Inserting them into the equation above yields:

$$\begin{aligned} & w(u, v) - \pi(u) + \pi(v) && \geq 0 \\ \Leftrightarrow\ & w(u, v) - d(u, L) + d(t', L) + d(t, t') + d(v, L) - d(t', L) - d(t, t') && \geq 0 \\ \Leftrightarrow\ & w(u, v) - d(u, L) + d(v, L) && \geq 0 \\ \Leftrightarrow\ & w(u, v) + d(v, L) && \geq d(u, L) \end{aligned}$$

With $w(u, v) \geq d(u, v)$, this is the triangle inequality for graphs, which is proven to be true.

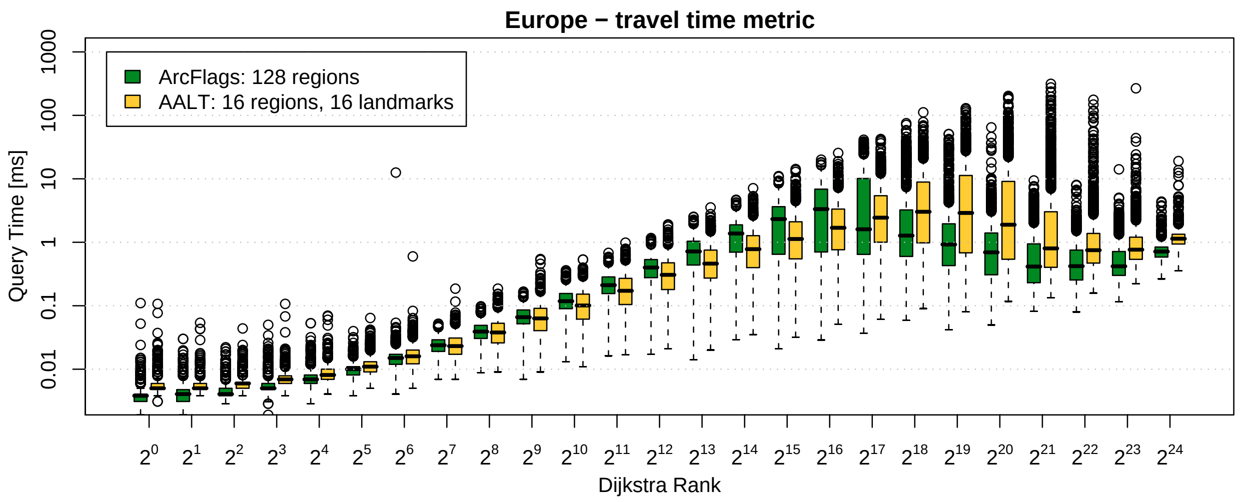
q.e.d.

Additional Results

Further results of the previously analyzed combinations of speed-up techniques are shown in this appendix. Dijkstra rank plots, comparing the performance of different algorithms, are presented.

C.1 AALT

The following Dijkstra rank plots are derived from the experiments presented in Tab. 6.3.



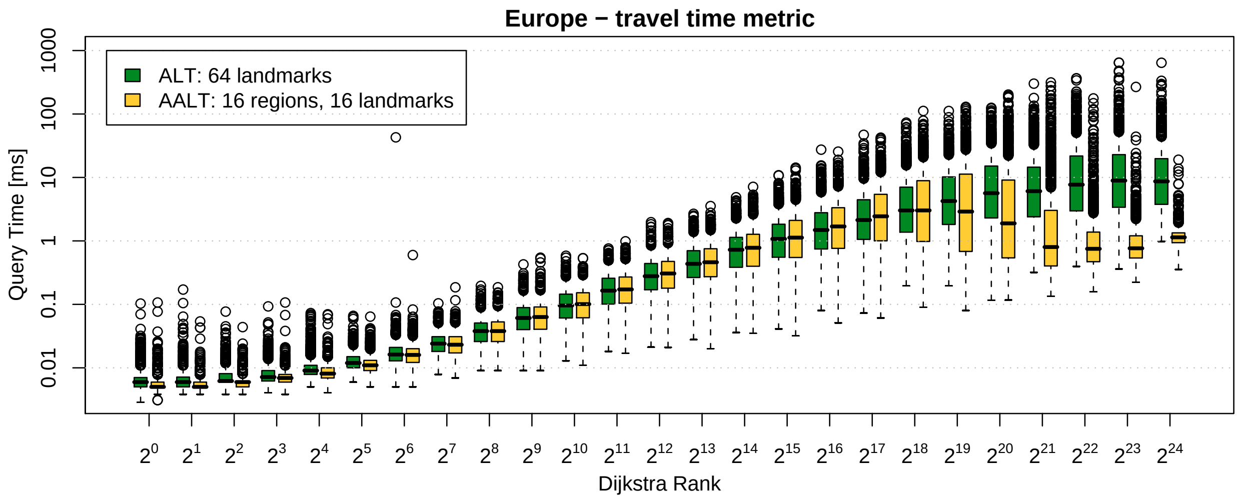
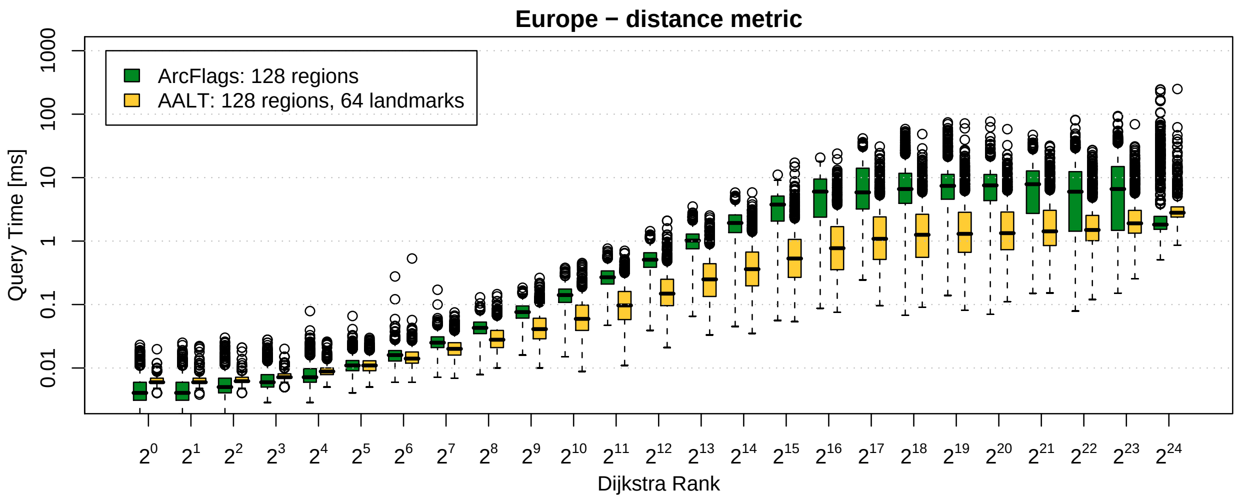
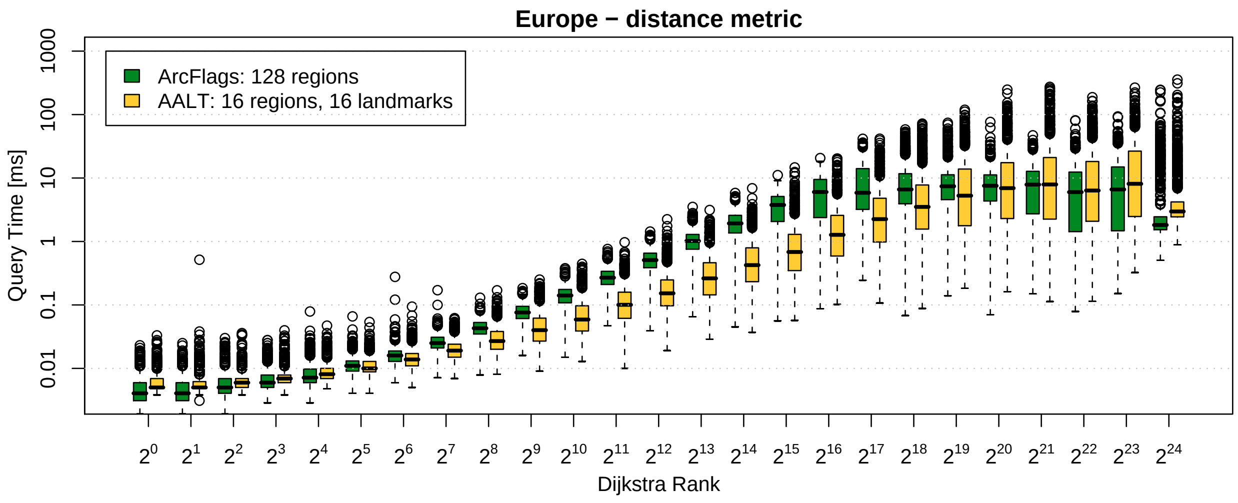


Figure C.1: *Comparison of ArcFlags, ALT and AALT on the European road network with a travel time metric.*





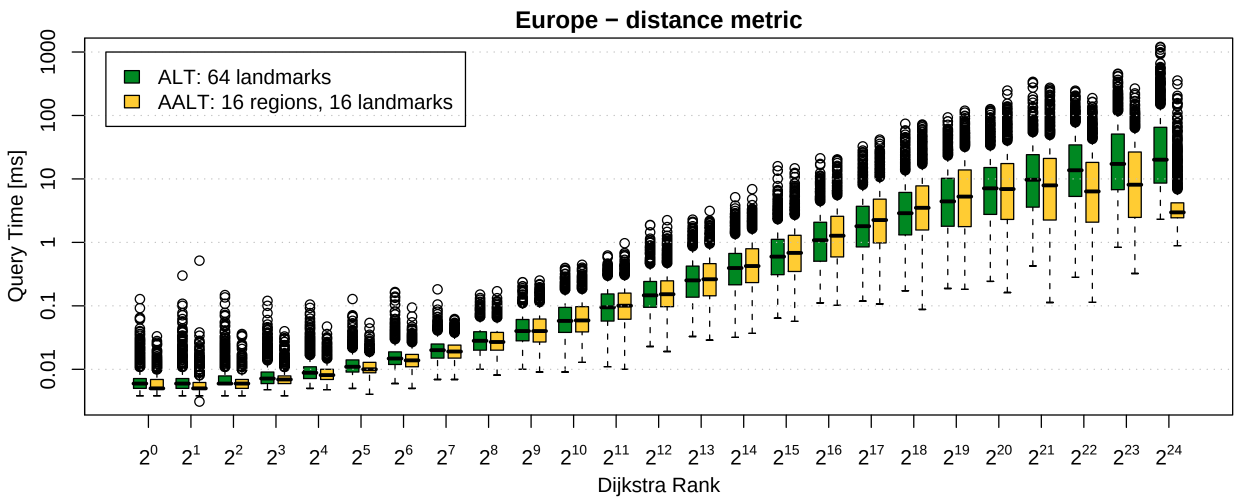
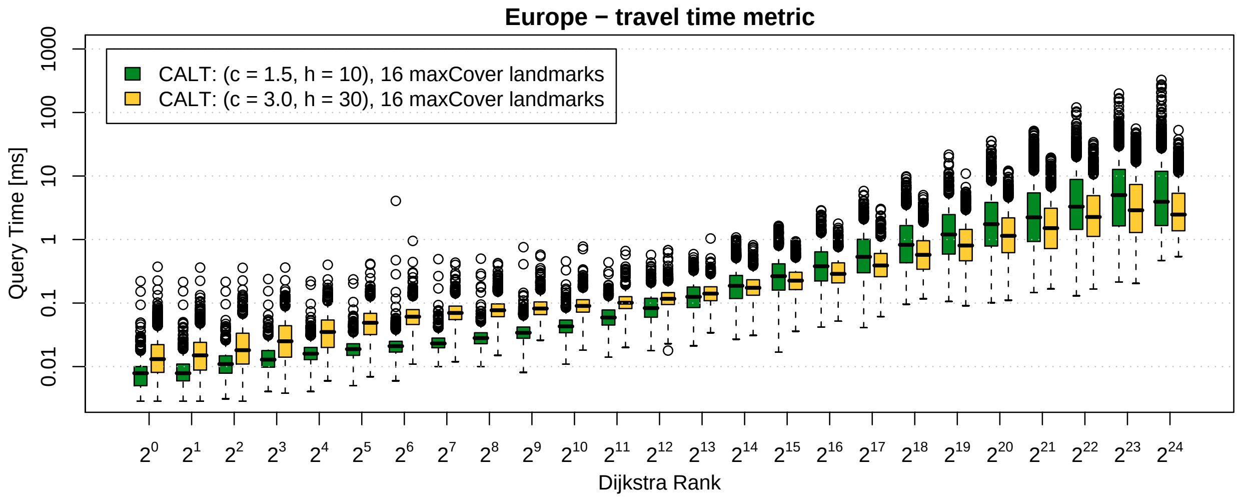
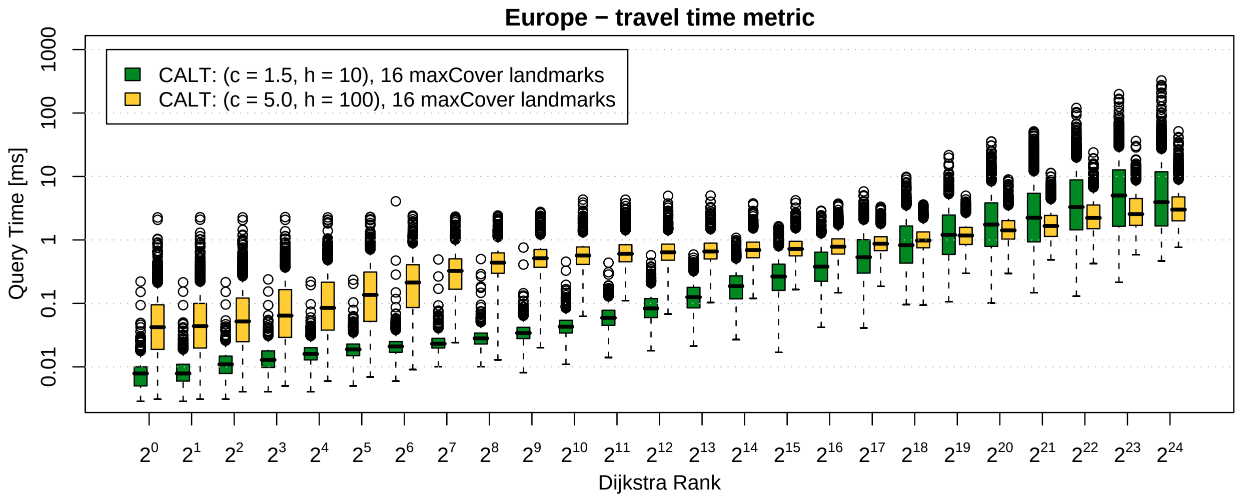


Figure C.2: *Comparison of ArcFlags, ALT and AALT on the European road network with a distance metric.*

C.2 CALT

Additional Dijkstra rank plots for the experimental results shown in Tab. 6.15 and 6.14.





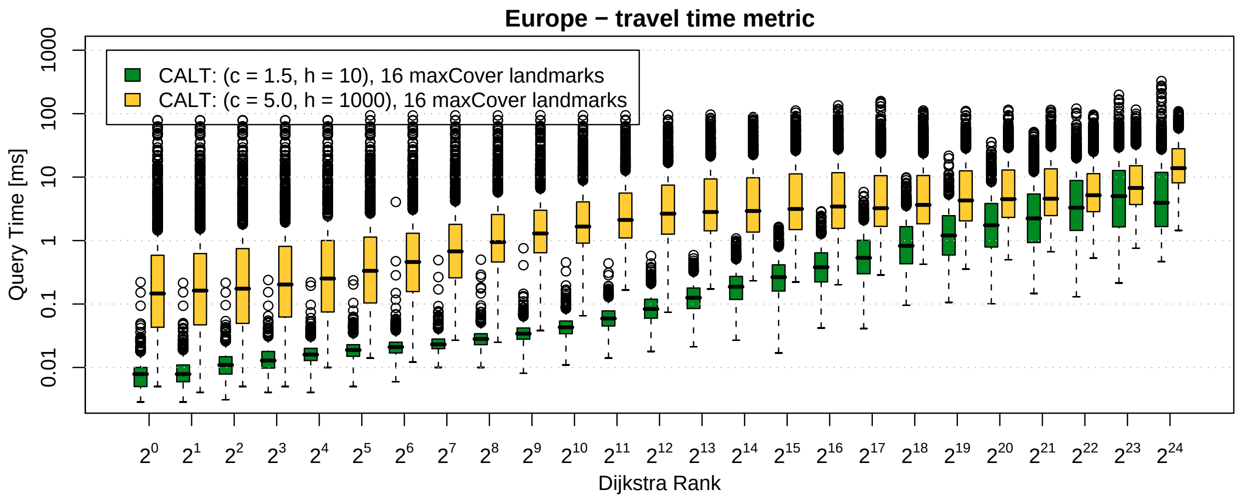
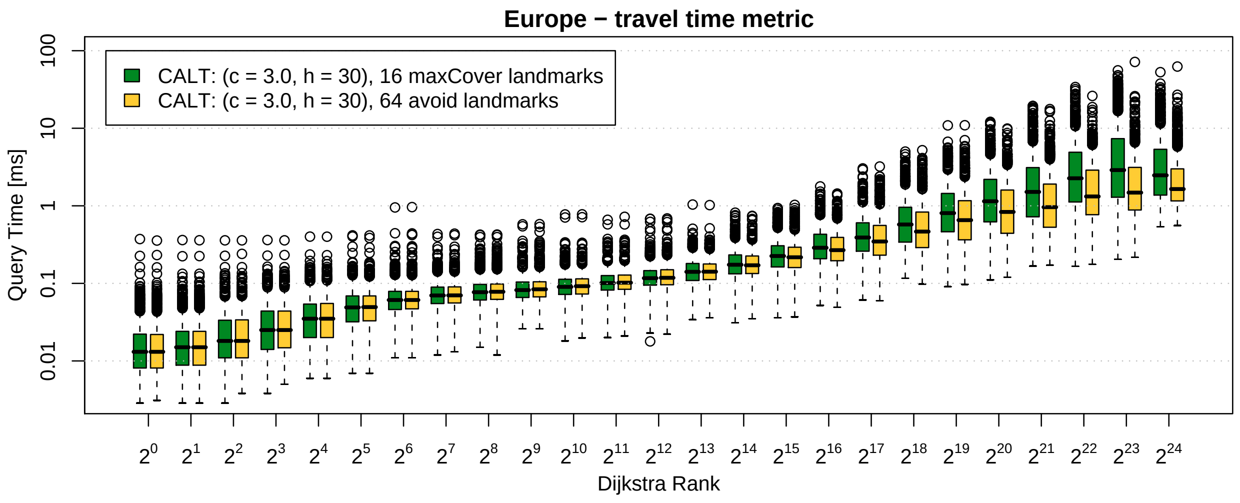


Figure C.3: *Comparison of the ALT algorithm and the CALT algorithm, varying the contraction parameters.*



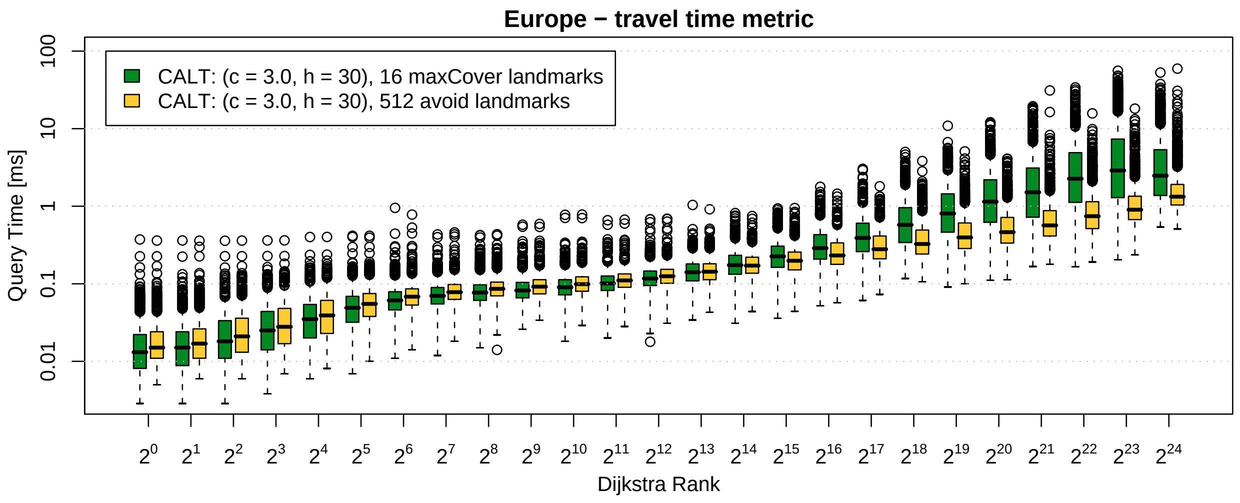
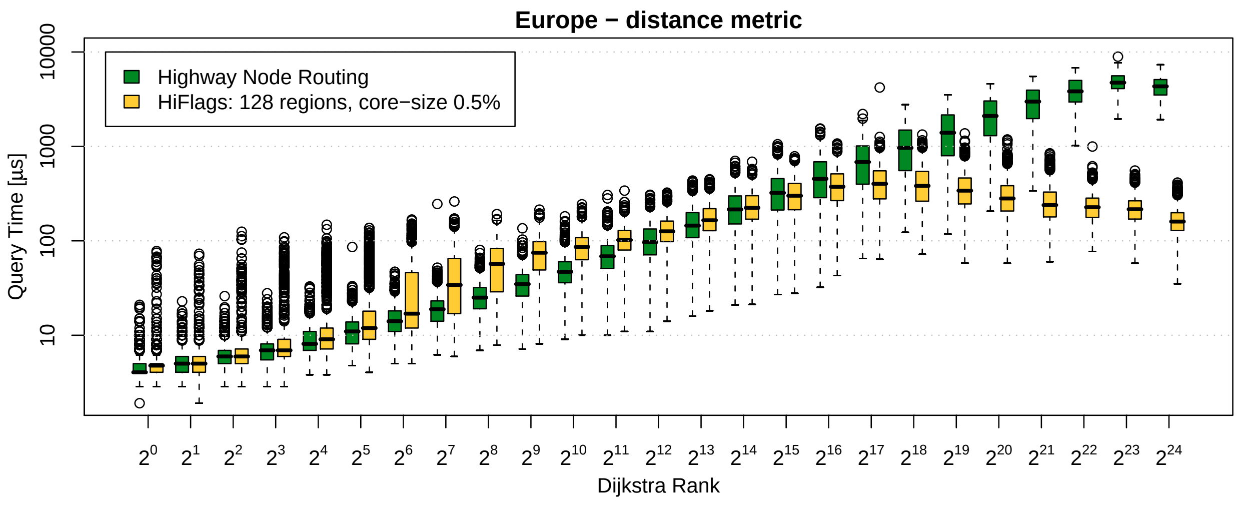
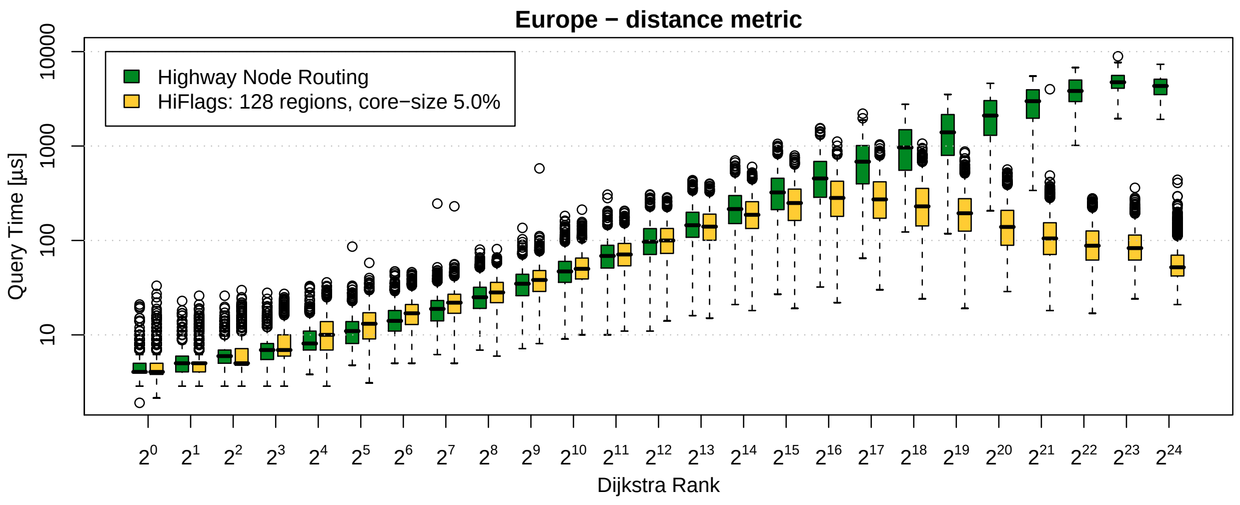


Figure C.4: *Comparison of the performance of the CALT algorithm using different numbers of landmarks.*

C.3 HiFlags

Further Dijkstra rank plots for the Hierarchy-aware ArcFlags experiments presented in Chap. 6.3.





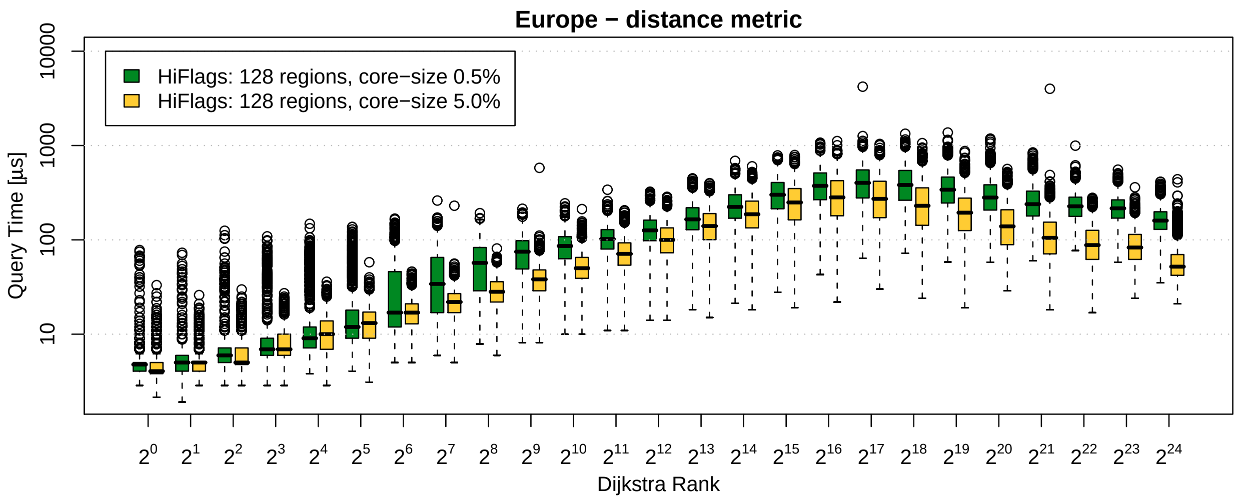
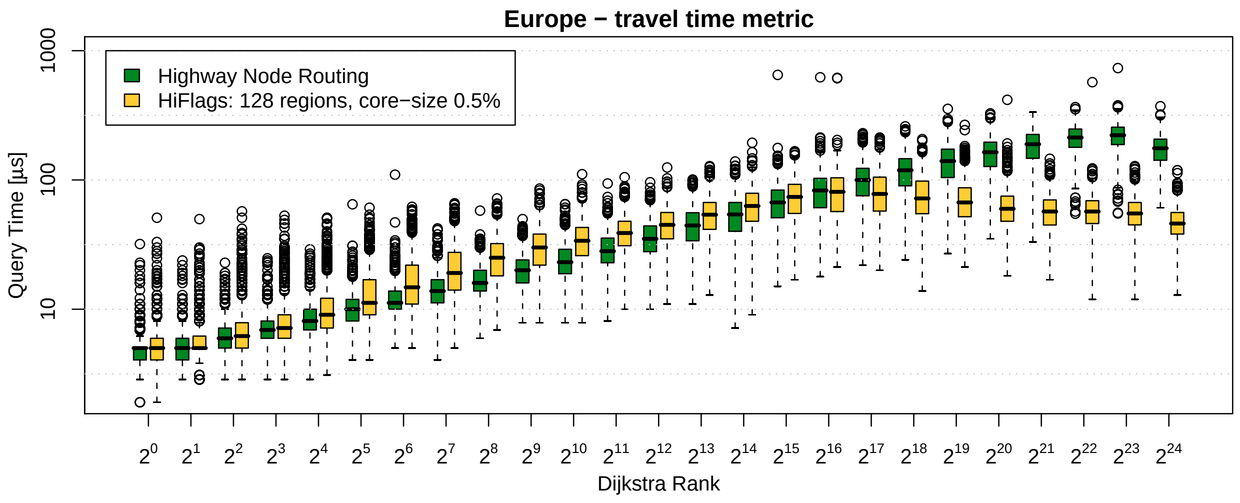
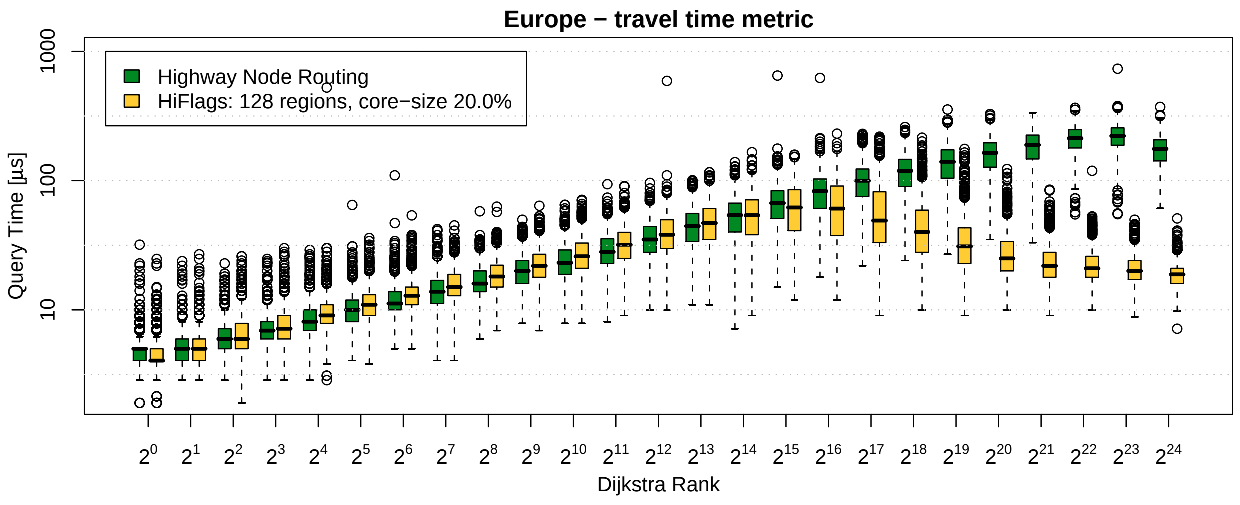


Figure C.5: *Comparison of HiFlags and Highway Node Routing on the European road map with a distance metric.*





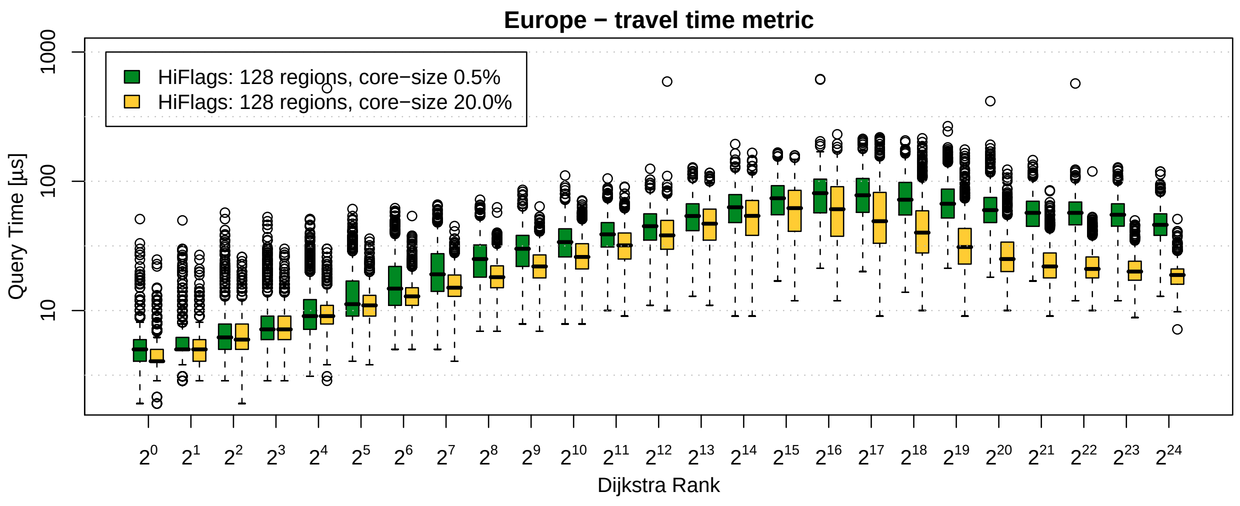
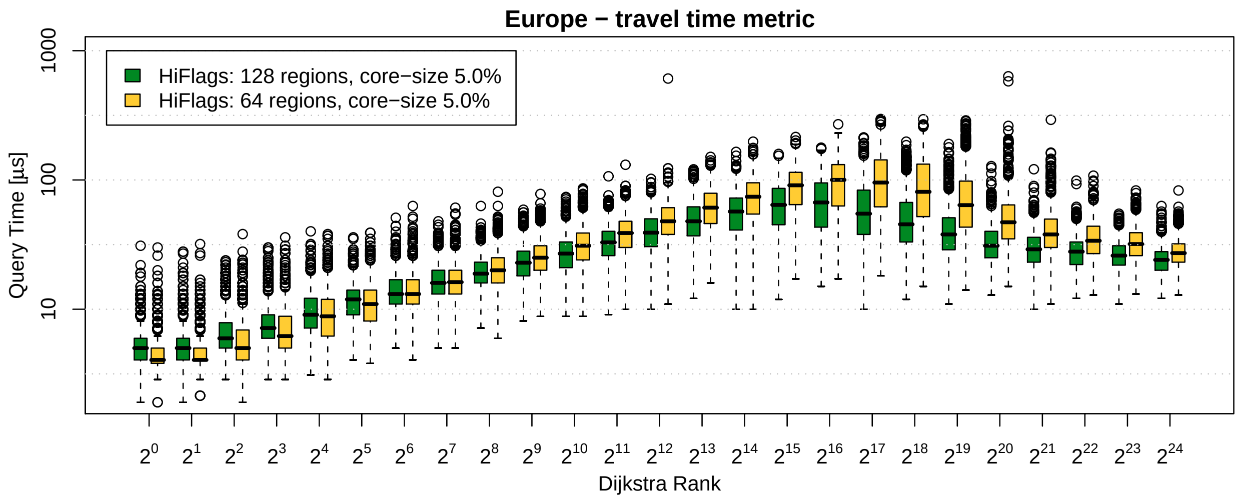


Figure C.6: *Comparison of HiFlags and Highway Node Routing on the European road network with travel times.*



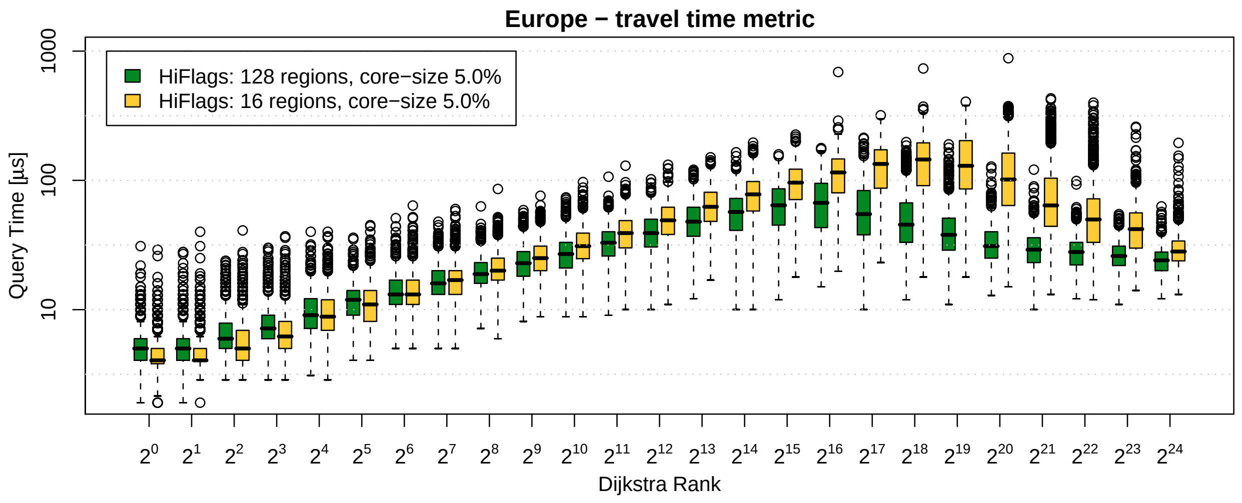


Figure C.7: *Comparison of Hierarchy-aware ArcFlags, using different numbers of regions for the ArcFlags.*

List of Figures

List of Tables

Bibliography

[Bau06] Reinhard Bauer. Dynamic Speed-Up Techniques for Dijkstra's Algorithm. Master's thesis, Institut für Theoretische Informatik - Universität Karlsruhe (TH), 2006.
(Cited on page 41.)

[BBK03] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact Representation of Separable Graphs. In *Proceedings of the 14th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'03)*, pages 679–688, 2003.
(Cited on page 97.)

[BD08] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, 2008.
(Cited on pages 2, 4, 19, 25, 87, and 91.)

[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. submitted to WEA'08, 2008.
(Cited on page 91.)

[BDW07] 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)*. Schloss Dagstuhl, Germany, 2007.
(Cited on pages 59 and 60.)

[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 page 91.)

[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 pages 3, 16, and 58.)

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.
(Cited on pages 7 and 9.)

[Dan62] George B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1962.
(Cited on pages 4, 10, and 38.)

[Del07] Daniel Delling. A Fast Framework for Engineering Shortest Path Algorithms. *Unpublished*, 2007.
(Cited on page 97.)

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
(Cited on pages 1 and 8.)

[DSSW06] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Highway Hierarchies Star. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.
(Cited on pages 1, 4, 25, 30, and 58.)

[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 91.)

[Gei08] Robert Geisberger. Contraction Hierarchies. Master's thesis, Institut für Theoretische Informatik - Universität Karlsruhe (TH), 2008.
(Cited on pages 3, 21, 36, 48, 53, 55, and 79.)

[GH04] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. Technical Report MSR-TR-200, Microsoft Research, 2004.
(Cited on pages 2, 13, 38, and 94.)

[GKW06a] 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 1, 2, 4, 15, 16, 25, 28, 30, 33, 34, and 50.)

[GKW06b] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *9th DIMACS Implementation Challenge -*

Shortest Paths, November 2006.
(Cited on pages 4, 15, 16, and 28.)

[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 4, 16, 28, 31, 33, 35, 40, 42, 58, 75, and 91.)

[GS95] Andrew V. Goldberg and Craig Silverstein. Implementations do Dijkstra's Algorithm based on Multi-Level Buckets. Technical Report 187, NEC Research Institute, 1995.
(Cited on page 100.)

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. submitted to WEA'08, 2008.
(Cited on page 91.)

[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 pages 2, 14, 16, 28, 30, and 50.)

[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 13, 46, and 61.)

[Hil07] Moritz Hilger. Accelerating Point-to-Point Shortest Path Computations in Large Scale Networks. Master's thesis, Technische Universität Berlin, 2007.
(Cited on pages 2, 18, 19, 49, 50, 53, 87, and 91.)

[HKMS06] Moritz Hilger, Ekkehard Köhler, Rolf 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, *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.
(Cited on pages 2, 18, 19, 48, and 60.)

[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 3 and 11.)

[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 pages 1, 3, 23, and 93.)

[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 pages 3 and 20.)

[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 pages 1, 3, and 23.)

[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*, pages 291–296. ACM Press, 1994.
(Cited on pages 11 and 12.)

[Joh77] Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM*, 24(1):1–13, January 1977.
(Cited on page 9.)

[KK98] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
(Cited on page 19.)

[KMS05] Ekkehard Köhler, Rolf 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 2 and 18.)

[Kwa89] James B. Kwa. BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm. *Artificial Intelligence*, 38(1):95–109, 1989.
(Cited on page 12.)

[KWZ03] Fabian Kuhn, Roger Watternhofer, and Aaron Zollinger. Worst-Case Optimal and Average-Case Efficient Geometric Ad-Hoc Routing. In *Proceedings of the 4th*

ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC'03), 2003.
(Cited on page 59.)

[Lab07] Karypis Lab. METIS - Family of Multilevel Partitioning Algorithms, 2007.
(Cited on pages 19, 48, and 60.)

[Lau97] Ulrich Lauther. Slow Preprocessing of Graphs for Extremely Fast Shortest Path Calculations, 1997. Lecture at the Workshop on Computational Integer Programming at ZIB.
(Cited on pages 2 and 17.)

[Lau04] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. volume 22, pages 219–230. IfGI prints, 2004.
(Cited on pages 2 and 17.)

[MHSWZ07] Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable Information: Models and Algorithms. In *Proceedings of the 4th Workshop on Algorithmic Methods for Railway Optimization (ATMOS'04)*, volume 4359 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 2007.
(Cited on page 59.)

[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 pages 19 and 48.)

[MSS$^+$05] Rolf 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 2 and 18.)

[MSS$^+$06] Rolf 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 2, 4, 16, 18, 19, and 25.)

[Mül06] Kirill Müller. Design and Implementation of an Efficient Hierarchical Speed-up Technique for Computation of Exact Shortest Paths in Graphs. Master's thesis, Universtät Karlsruhe, June 2006.
(Cited on page 3.)

[Paj08] Thomas Pajor. Speed-Up Techniques for Shortest Path Queries in Timetable Networks, January 2008. Student Research Project.
(Cited on pages 27, 59, and 94.)

[Pel07] Francois Pellegrini. SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package, 2007.
(Cited on pages 19, 48, 60, 75, and 79.)

[Poh71] I. Pohl. Bi-directional Search. In Bernard Meltzer and Donald Michie, editors, *Proceedings of the Sixth Annual Machine Intelligence Workshop*, volume 6, pages 124–140. Edinburgh University Press, 1971.
(Cited on page 11.)

[PTV79] PTV AG. Planung Transport Verkehr. `http://www.ptv.de`, 1979.
(Cited on page 58.)

[Sch07] Dominik Schultes. *Route Planning in Road Networks*. PhD thesis, 2007. submitted.
(Cited on pages 21, 40, 56, and 91.)

[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 3, 16, 20, 21, 25, 30, 33, 57, 63, 71, 84, and 85.)

[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 page 33.)

[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 3, 16, 20, 22, and 33.)

[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 page 1.)

[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 1, 3, and 20.)

[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 3 and 20.)

[tDIC06] 9th DIMACS Implementation Challenge. Shortest Paths. `http://www.dis.uniroma1.it/~challenge9/`, 2006.
(Cited on page 58.)

[Tea04] R Development Team. R: A Language and Environment for Statistical Computing, 2004.
(Cited on pages 57 and 63.)

[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 3.)

Deutsche Zusammenfassung

Seitdem Dijkstra 1959 den nach ihm benannten Algorithmus zum Finden kürzester Wege in einem statischen Graph vorgestellt hat, sind zahlreiche neue Techniken eingeführt worden, um diese Suche beschleunigen. Die meisten beruhen auf einem von zwei grundlegenden Ansätzen, um den Suchraum zu verkleinern und damit die Suche zu beschleunigen: Entweder wird versucht, eine Hierarchie im Graphen zu erkennen (z. B. Landstraßen und Autobahnen) und diese auszunutzen, oder es wird versucht möglichst zielgerichtet die Suche voranzutreiben. Vertreter der ersten Art sind z. B. der *Reach*–Algorithmus, *Highway Hierarchies* und *Highway Node Routing*. Zur zweiten Gruppe zählen u. a. der *ALT*–Algorithmus sowie *ArcFlags*. Da hier jeweils vollkommen unterschiedliche Eigenschaften des Graphen ausgenutzt werden, ist zu vermuten, dass eine Kombination beider Arten von Techniken besonders gut funktioniert. Dies wurde bereits für die Kombination des *ALT*–Algorithmus mit dem *Reach*–Algorithmus sowie *Highway Hierarchies* durchgeführt. Die sich ergebenden neuen Techniken wurden *REAL* bzw. *Highway Hierarchies** genannt. Zudem können unter Umständen Synergien zwischen den Techniken genutzt werden, um die nötigen Vorberechnungen zu verbessern und zu beschleunigen.
Die vorliegende Arbeit stellt vier weitere Kombinationen bestehender Beschleunigungstechniken zum Finden kürzester Wege vor. Außerdem werden ausführliche Experimente auf mehreren unterschiedlichen Graphtypen durchgeführt und die Ergebnisse präsentiert. Im Einzelnen handelt es sich hierbei um die folgenden Algorithmen:

Zunächst wird der *ALT*–Algorithmus mit *ArcFlags* kombiniert, da dies sehr einfach möglich ist. Der sich ergebende *AALT*–Algorithmus ist insofern besonders, da er aus zwei zielgerichteten Techniken zusammengesetzt ist. Entsprechend sind auch die Erwartungen an die zu erzielende Beschleunigung gegenüber den Einzelverfahren gering. Diese Vermutung wurde auch experimentell bestätigt. Zudem addiert sich der Speicheraufwand sowie die Vorberechungszeit beider Techniken direkt, was dieses Verfahren eher uninteressant macht.

Der *CALT*–Algorithmus verwendet eine sogenannte Kontraktionstechnik, um den Graph auf einen Kern wichtiger Knoten zu reduzieren. Kürzeste Wege zwischen diesen werden durch das Einbringen neuer Kanten, die die entfernten Knoten umgehen, erhalten. Die Suche versucht nun möglichst schnell diesen Kern zu erreichen und wendet auf ihm den *ALT*–Algorithmus an. Die neu eingeführten Kanten sorgen für eine zügige Suche, und die Beschränkung des *ALT*–Algorithmus auf einen kleineren Teilgraph sorgt für eine schnellere Vorberechnung und weniger Speicherverbrauch. Es hat sich gezeigt, dass dieser Algorithmus sehr gute Ergebnisse auf jeglichem Graphtyp erzielt bei gleichzeitig geringem zusätzlichen Rechenaufwand und Speicherbedarf. Obwohl er nicht mit den schnellsten Beschleunigungstechniken konkurrieren kann, erlaubt seine Einfachheit ihn auch für andere Anwendungen, wie z. B. dynamische Graphen, einzusetzen.

Die folgenden zwei Techniken nutzen beide das *Partial ArcFlags* Verfahren und kombinieren es mit dem *Reach*–Algorithmus bzw. *Highway Node Routing*. Dieses Verfahren ist eine Variante des normalen *ArcFlags*-Algorithmus, das diese Beschleunigungstechnik nur auf einen Teilgraphen anwendet. Hierbei wird versucht, einen Teilgraphen auszuwählen, über den möglichst viele kürzeste Wege laufen. Diese Wahl wird durch die Bildung von Hierarchien durch die anderen beiden Techniken unterstützt und erleichtert. Die Kombination mit dem *Reach*–Algorithmus wird als *Reach-aware ArcFlags* bezeichnet. Die mit diesem Verfahren erzielten Ergebnisse waren eher gemischt. Es hat sich v. a. als nachteilig erwiesen, dass sich die erzielbaren Ergebnisse nur bedingt aus der Wahl der Parameterwerte ableiten lassen. Die Kombination mit *Highway Node Routing* heißt *Hierarchy-aware ArcFlags* und verwendet, um die benötigte Einteilung des Graphen in Hierarchiestufen zu erhalten, zudem das *Contraction Hierarchies* Verfahren. Die erhaltenen Experimente haben das erstaunliche Potential offenbart, dass diese Kombination besitzt. Sie ist nicht nur schneller als fast jede andere Beschleunigungstechnik, die benötigte Vorberechnugszeit und der Speicheraufwand sind für die gebotene Leistung auch äußerst gering.

Die vorgestellten Techniken haben gezeigt, dass eine Kombination von hierarchischen Methoden mit zielgerichteter Suche außerordentlich gute Ergebnisse erzielt, sogar dann, wenn letztere nur auf den höheren Ebenen der Hierarchie angewandt wird. Dies verringert außerdem die anfallenden Kosten der Beschleunigungstechnik, teilweise sogar beträchtlich. Alle Techniken weisen Gemeinsamkeiten in Bezug auf den verwendeten zweistufigen Algorithmus auf. Dies verbindet sie auch mit früheren Methoden wie *REAL* oder *Highway Hierarchies**.

Abschließend lässt sich sagen, dass diese Arbeit zwei sehr vielversprechende Techniken hervorgebracht hat. Zum einen den *CALT*–Algorithmus, der ein relativ einfaches, aber dennoch schnelles und v. a. robustes Verfahren darstellt, dass sich voraussichtlich auch sehr gut für den Einsatz mit dynamischen oder sogar zeitabhängigen Graphen eignen wird. Zum anderen die *HiFlags*–Beschleunigungstechnik, die gerade auf dünnen Graphen, insbesondere wenn sie eine hierarchische Struktur besitzen, das derzeit schnellste Verfahren darstellt, mit Ausnahme von *Transit Node Routing*. Zudem zeichnet sie sich durch einen wesentlich geringeren Speicherbedarf sowie eine kürzere Vorberechnungszeit gegenüber dem anderen Verfahren aus.

Danksagung

Zuerst möchte ich mich bei Daniel Delling für die Ausschreibung und die großartige Betreuung dieser Diplomarbeit bedanken. Daniel hatte stets ein offenes Ohr für meine Fragen und stand mir mit Rat und neuen Ideen zur Seite. Außerdem hatte er sich gegen Ende die Diplomarbeit dazu bereit erklärt, meine Ausarbeitung korrekturzulesen. Ebenso danke ich Frau Prof. Dr. Dorothea Wagner für die Annahme und Bewertung dieser Arbeit.

Des Weiteren möchte ich Reinhard Bauer, Robert Geisberger und Dominik Schultes danken. Bei Reinhard bedanke ich mich für die Hilfe bei der schriftlichen Ausarbeitung der Diplomarbeit sowie das Korrekturlesen und bei Dominik für die Zurverfügungstellung seiner Erfahrung und Unterlagen über das *Highway Node Routing* Verfahren. Robert danke ich, dass er mir seine Routinen zur Vorberechnung für diese Beschleunigungstechnik bereit gestellt hat.

Außerdem möchte ich den Systemadministratoren und den weiteren Personen danken, die die reibungsfreie Arbeit am Institut für Theoretische Informatik ermöglicht haben.

Zum Schluss möchte ich mich insbesondere noch bei meinen Eltern bedanken, die mich während aller Studienabschnitte in Physik sowie Informatik in jeder nur erdenklichen Hinsicht unterstützt haben und immer für mich da waren. Sie waren eine unersetzliche Hilfe auf meinem Weg.

Hiermit versichere ich, die vorliegende Arbeit selbständig verfasst
und nur die angegebenen Hilfsmittel verwendet zu haben.

Dennis Schieferdecker

Karlsruhe, den 31. Januar 2008