Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks

Diploma Thesis

at

Institut für Theoretische Informatik
Universität Karlsruhe (TH)

of

Robert Geisberger

handed in:

01. July 2008

supervised by:

Prof. Dr. Peter Sanders
Dr. Dominik Schultes

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, 01.07.2008

Robert Geisberger

Abstract

We present a route planning technique solely based on the concept of node *contraction*. We contract or remove one node at a time out of the graph and add *shortcut edges* to the remaining graph to preserve shortest paths distances. The resulting *contraction hierarchy (CH)*, the original graph plus shortcuts, also defines an order of "importance" among all nodes through the node selection. We apply a modified bidirectional Dijkstra algorithm that takes advantage of this node order to obtain shortest paths. The search space is reduced by relaxing only edges leading to more important nodes in the forward search and edges coming from more important nodes in the backward search. Both search scopes eventually meet at the most important node on a shortest path. We use a *simple* but *extensible* heuristic to obtain the node order: a priority queue whose priority function for each node is a linear combination of several terms, e.g. one term weights nodes depending on the sparsity of the remaining graph after the contraction. Another term regards the already contracted nodes to allow a more uniform contraction. Depending on the application we can select the combination of the priority terms to obtain the required hierarchy. We have five times lower query times than the best previous hierarchical Dijkstra-based speedup techniques and a *negative* space overhead, i.e., the data structure for distance computation needs *less* space than the input graph. CHs can serve as foundation for many other route planning techniques, leading to improved performance and reduced memory consumption for many-to-many routing, transit-node routing, goal-directed routing or mobile and dynamic scenarios.

Contents

1 Introduction

1.1 Motivation

Solving shortest-path problems and related problems is a field of research for a long time. Dijkstra's algorithm [7] basically solves the shortest-path problem but it is not efficient for large graphs. Lately large real-world graphs like road networks of whole continents become available and need more sophisticated algorithms. Their applications range from planning a motorcycle tour with a mobile device to facility location problems of large industrial companies. The question is always, how much space overhead and preprocessing time we are willing to spend and how fast our shortest path queries are going to be solved. We aim to provide a new technique that can offer a new foundation for all sorts of Dijkstra based algorithms. A selection of those algorithms are many-to-many shortest paths [21], transit-node routing [2] or upcoming time-dependent routing algorithms. The goal was to keep things simple but extendable, with reasonable hardware requirements and still calculating optimal routes.

1.2 Related work

This section is a shortened version of the related work section in [32, 34]. There has recently been extensive research on point-to-point shortest path speedup techniques to calculate the shortest path distance $d(s,t)$ between two nodes s and t. Beneath classical results like Dijkstra's algorithm [7], they can be categorized into hierarchical and goal-directed techniques, and combinations of both. We will focus on the most important and recent speedup techniques, for a more detailed overview, we refer to [32, 34].

1.2.1 Classical Results

Dijkstra's Algorithm [7] maintains an array of *tentative distances* for each node. The algorithm *settles* (or *visits*) the nodes of the road network in the order of their distance to the source node s and maintains the invariant that the tentative distance is equal to the correct distance for settled nodes. When a node u is visited, its outgoing edges (u,v) are relaxed: the tentative distance of v is set to the length of the path from s via u to v provided that this leads to an improvement. Dijkstra's algorithm can be stopped when the target node is visited. The size of the search space is $O(n)$ and $n/2$ nodes on the average.

Usually, Dijkstra's algorithm is implemented using a priority queue. Since $O(n)$ operations are required, this leads to $O(n \log n)$ execution time in the comparison based model.

A simple improvement is to execute Dijkstra's algorithm simultaneously forwards from the source node s and backwards from the target node t. Such a *bidirectional search* can derive the shortest path from the gathered information once some node has been settled from both directions [6]. In a road network, where search spaces will take a roughly circular shape, we can expect a speedup of around two – two disks with half the radius of one disk have half the area. Many more advanced speedup techniques, including contraction hierarchies, use bidirectional search as an mandatory ingredient.

1.2.2 Hierarchical Approaches

Reach-Based Routing. Let $R(v) := \max\{R_{st}(v) \mid s,t \in V\}$ denote the *reach* of node v, where $R_{st}(v) := \min(d(s,v), d(v,t))$. Gutman [17] observed that a shortest-path search can

be pruned at nodes with a reach too small to get to source or target from there. The basic approach was considerably strengthened by Goldberg et al. [11, 14, 15], in particular by a clever integration of *shortcuts*.

Highway Hierarchies (HHs) [30, 31, 34] group nodes and edges in a hierarchy of levels by alternating between two subroutines: *Node reduction* removes low degree nodes by bypassing them with newly introduced shortcut edges. In particular, all nodes of degree one and two are removed by this process. Edge reduction removes *non-highway edges*, i.e., edges that only appear on shortest paths close to source or target. More specifically, every node v has a neighborhood radius $r(v)$ we are free to choose. An edge (u, v) is a highway edge if it belongs to some shortest path P from a node s to a node t such that (u, v) is neither fully contained in the neighborhood of s nor in the neighborhood of t, i.e., $d(s, v) > r(s)$ and $d(u, t) > r(t)$. The query algorithm is very similar to bidirectional Dijkstra search with the difference that certain edges need not be expanded when the search is sufficiently far from source or target. HHs are similar to contraction hierarchies. However, we avoid the costly edge reduction step and use a more sophisticated node reduction routine that was previously a mere helper for the main work-horse edge reduction. Interestingly, this is sufficient without an eventual explosion of the average degree in the remaining graph.

Highway-Node Routing (HNR) [35, 34] is a generalization of the multi-level routing scheme with *overlay graphs* that preserve shortest paths distances on subsets of important nodes, so called *highway nodes*. First multi-level routing schemes used the fact that road networks are almost planar [8, 20] but it turned out that using HHs to define the highway node sets is superior. This is achieved using a new query algorithm that stalls suboptimal branches of search on lower levels of the hierarchy. Given a hierarchy of highway-node sets, we recursively compute the shortcuts bottom up. Shortcuts from level ℓ are found by *local*, i.e limited, searches in level $\ell - 1$ starting from nodes in level ℓ. This is very fast and easy to update when edge weights change. Contraction Hierarchies are an extreme case of the hierarchies in HNR – every node defines its own level of the hierarchy.

Transit-Node Routing (TNR) precomputes not only a distance table for important (*transit*) nodes but also all relevant connections between the remaining nodes and the transit nodes [3, 34]. Since it turns out that only about ten such access connections are needed per node, one can "almost" reduce routing in large road networks to about 100 table lookups. Interestingly, the difficult queries are now the local ones where the shortest path does not touch any transit node. This problem is solved by introducing several *layers* of transit nodes. Between lower layer transit nodes, only those routes need to be stored that do not touch the higher layers. Transit node routing (e.g., using appropriate levels of a HH for transit node sets) reduces routing times to a few microseconds at the price of preprocessing times an order of magnitude larger than HHs alone. Also considerably more space is required and it is less amenable to dynamization. Since it relies on another hierarchical speedup technique for its preprocessing, contraction hierarchies may be able to improve TNR.

1.2.3 Advanced Goal-Directed Search and Combinations

Landmark-Based A^* Search. In [12, 13, 16] the ALT algorithm is presented that is based on $\underline{A}^*$ search [18], $\underline{L}$*andmarks*, and the $\underline{T}$riangle inequality. After selecting a small number of

landmarks, for all nodes v, the distances $d(v, \mathcal{L})$ and $d(\mathcal{L}, v)$ to and from each landmark $\mathcal{L}$ are precomputed. For nodes v and t, the triangle inequality yields for each landmark $\mathcal{L}$ two lower bounds $d(\mathcal{L}, t) - d(\mathcal{L}, v) \leq d(v, t)$ and $d(v, \mathcal{L}) - d(t, \mathcal{L}) \leq d(v, t)$. The maximum of these lower bounds is used during an A^* search to add a sense of direction to the search process.

REAL. Goldberg et al. [11, 14, 15] have successfully combined their advanced version of REach-based routing with the landmark-based A^* search (the ALt algorithm), obtaining the REAL algorithm. In the most recent version [14, 15], they introduce a variant where landmark distances are stored only with the more important nodes, i.e., nodes with high reach value. By this means, the memory consumption can be reduced significantly.

Edge Flags (also called *arc flags*) [24, 22, 27, 28, 23, 25, 19] precompute for each edge "signposts" that support the decision whether the target can possibly be reached on a shortest path via this edge. The graph is partitioned into k regions. For each edge e and each region r, one flag is computed that indicates whether e lies an a shortest path to some node in region r. Dijkstra's algorithm can take advantage of the edge flags: edges have to be relaxed only if the flag of the region tat the target node belongs to is set. Obviously, inside the target region, the "signposts" provided by the edge flags get less useful. This problem can be avoided by performing a bidirectional query so that forward and backward search can meet somewhere in the middle.

SHARC [4] extends and combines ideas from highway hierarchies (namely, the contraction phase, which produces SHortcuts) with the edge flag (also called ARC flag) approach. It also incorporates a priority queue to contract nodes. The result is a fast *unidirectional* query algorithm, which is advantageous in scenarios where bidirectional search is prohibitive. In particular, using an approximative variant allows dealing with time-dependent networks efficiently.

CHASE [5] is a combination of contraction hierarchies and edge (arc) flags (Contraction Hierarchy + Arc flagS + highway-nodE routing). After the hierarchy construction using HNR, the subgraph H of the h highest level nodes is partitioned into k regions and arc flags are computed for them. The bidirectional query algorithm from HNR is used until the search reaches H, then it activates the edge flags, yielding query times below 20μs for the Western European road network.

Transit-Node Routing and Edge Flags [5] is currently the fastest speedup technique. It requires even more preprocessing time and space than native TNR, but can decrease the average query time to 1.9 μs for the Western European road network.

1.3 Our contribution

We present a very simple approach to hierarchical routing. Assume the nodes of a weighted directed graph $G = (V, E)$ are ordered by "importance" given a total node order $<$. Let $u < v$, then the node v is more important than u. We now construct a hierarchy by *contracting* the nodes in this order. A node u is contracted by removing it from the network in such a way that shortest paths in the remaining *overlay graph* are preserved. This property is achieved by replacing paths of the form $\langle v, u, w \rangle$ by a *shortcut* edge $\langle v, w \rangle$. Note that the shortcut $\langle v, w \rangle$ is

only required if $\langle v, u, w \rangle$ is the only shortest path from v to w. We shall view the contraction process as a way to add all discovered shortcuts to the edge set E. We obtain a *contraction hierarchy (CH)*.

Although "optimal" node ordering seems a quite difficult problem, already very simple local heuristics turn out to work quite well. The basic idea is to keep the nodes in a priority queue sorted by some estimate of how attractive it is to contract a node. The main ingredient of this heuristic estimate is the *edge difference*: The number of shortcuts introduced when contracting v minus the number of edges incident to v. The intuition behind this is that the contracted graph should have as few edges as possible. Even using only edge difference, quite good CHs are computed. However, further refinements are useful. In particular, it is important to contract nodes "uniformly".

Our new hierarchies are to some extend a specialization of HNR but they are nevertheless a new approach in the sense that the node ordering and hierarchy construction algorithms used in [35, 34] are only efficient for a small number of geometrically shrinking levels. In contrast, we use a separate level for each node. We also give a faster and more space efficient query algorithm exploiting the total order among all nodes.

For routing, we split the CH $(G = (V, E), <)$ into an *upward graph*

$$G_\uparrow := (V, E_\uparrow) \text{ with } E_\uparrow := \{(u, v) \in E : u < v\}$$

and a *downward graph*

$$G_\downarrow := (V, E_\downarrow) \text{ with } E_\downarrow := \{(u, v) \in E : u > v\} \quad .$$

For a shortest path query from s to t, we perform a modified bidirectional Dijkstra shortest path search, consisting of a forward search in $G_\uparrow$ and a backward search in $G_\downarrow$. If, and only if, there exists a shortest s-t-path in the original graph, then both search scopes eventually meet at a node v that has the highest order of all nodes in a shortest s-t-path.

1.4 Outline

We first repeat some basics from graph theory in Section 2. In Section 3 our new CHs are explained in full detail including node order selection (Section 3.2), node contraction (Section 3.3) and the query algorithm (Section 3.5). Applications of CHs are presented in Section 4. Finally, we present our experimental study in Section 5 followed by a discussion in Section 6.

2 Preliminaries

This section was taken from [34], we use exactly the same formulation and notation.

2.1 Definitions

We will use some common definitions from graph theory. Let $G = (V, E)$ be an *directed* graph with *weight* function $w : E \to \mathbb{R}^+$ that assigns a *positive* weight to each edge in G. For an edge $(v, w) \in E$ we usually write $w(v, w)$ instead of $w((v, w))$. Usually $n = |V|$ denotes the number of nodes and $m = |E|$ denotes the number of edges. A *path* P in G is a sequence of nodes

$$\langle v_1, \ldots, v_k \rangle$$

with $(v_i, v_{i+1}) \in E$ for all $i = 1, \ldots, k-1$. For $1 \leq i < j \leq k$ the subpath of P between v_i and v_j is denoted by $P|_{v_i \to v_j} = \langle v_i, \ldots, v_j \rangle$.

The weight function can be extended to this path as the sum of all edge weights, see Figure 1:

$$w(P) := \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

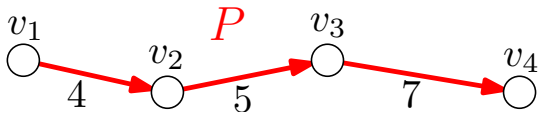


Figure 1: A path $P = \langle v_1, v_2, v_3, v_4 \rangle$. The numbers on the edges denote the edge weight, so $w(P) = 16$.

For each pair $(v, w) \in V \times V$ there exists a unique shortest path *distance*, if there is at least one path between them.

$$d_G(v, w) := \min \{w(P) \mid P = \langle v, \ldots, w \rangle \text{ path in } G\} \text{ with } \min \emptyset = \infty$$

The graph G as parameter is omitted if the graph is clear from the context. For a given graph $G = (V, E)$ all shortest path distances define a function

$$d_G : V \times V \to \mathbb{R}_0^+ \cup \{\infty\}.$$

Let $V' \subseteq V$, then $G[V'] := (V', E')$ with $E' := \{(x, y) \in E \mid x, y \in V'\}$ is the subgraph of G induced by the node set V'.

3 Contraction Hierarchies

In this section we will define CHs and explain how to construct them. The main parts of the construction are the node order selection (Section 3.2) and the node contraction (Section 3.3). Then we put them in the context of highway-node routing [34] (Section 3.4) and present our query algorithm (Section 3.5).

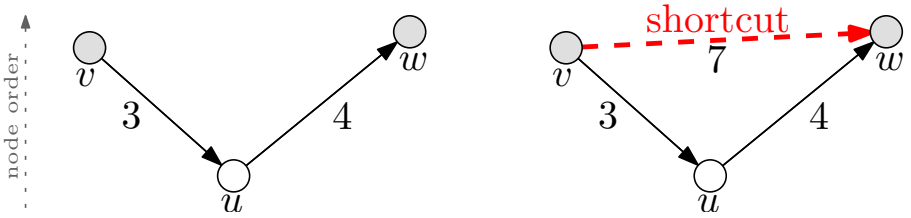
3.1 Definition

The basic idea of contraction hierarchies is to find a total order $<$ among all nodes and execute the following procedure:

Algorithm 1: SimplifiedConstructionProcedure($G = (V, E)$,$<$)

```
1 foreach u ∈ V ordered by < ascending do
2     foreach (v, u) ∈ E with v > u do
3         foreach (u, w) ∈ E with w > u do
4             if ⟨v, u, w⟩ "may be" the only shortest path from v to w then
5                 E:= E ∪ {(v, w)} (use weight w(v, w):= w(v, u) + w(u, w))
```

Such an edge added in Line 5 is called a *shortcut edge* or just *shortcut*, see Figure 2. They only represent existing paths in the current graph and are used to preserve shortest paths if the node u and all its incident edges is removed from the graph. If an edge (v, w) already exists in G but has larger weight than a new shortcut (v, w), then the Line 5 only reduce the weight of the already existing edge.



(a) before (b) after

Figure 2: Adding a shortcut edge. Node order: $u < v < w$.

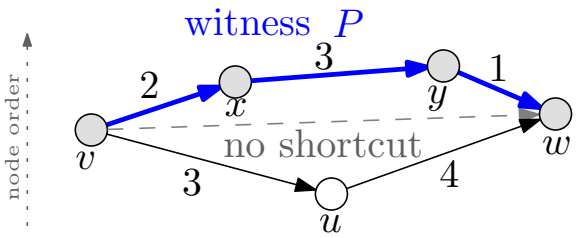


Figure 3: No shortcut because a witness $P = \langle v, x, y, w\rangle$ exists with $w(\langle v, u, w\rangle) \geq w(P)$.

A path $P = \langle v, \ldots, w\rangle \neq \langle v, u, w\rangle$ between v and w with $w(P) \leq w(\langle v, u, w\rangle)$ is called a *witness path* or just *witness* for the triple v, u, w, see Figure 3. The name is derived from the fact that such a path witnesses that $\langle v, u, w\rangle$ is not a shortest path or not the only shortest path and allows to omit a shortcut.

The whole step of finding witnesses and adding shortcuts for node u, Lines 2–5, is called the *contraction* of u. Because after that step, for all shortest s-t-paths P in G, with $s, t > u$, that have u in their interior, exists a shortest s-t-path P' in G without u in its interior. Applied recursively, it is easy to see that there even exists an shortest s-t-path P' with only nodes $> u$ in its interior. In the context of the contraction of node u, an edge $(v, u) \in E$ is called an *incoming edge* of u, (u, w) an *outgoing edge* of u. The nodes $> u$ are called *remaining nodes*, the incident edges are called the *remaining edges* and the graph induced by the remaining nodes is called the *remaining graph*.

The tuple $(G = (V, E), <)$ consisting of the resulting graph G of Algorithm 1 and the node order $<$ is called a *contraction hierarchy (CH)*. The node order partitions the nodes into n distinct *levels*.

3.2 Node Order Selection

After presenting the basic concept and the construction of CHs, we now show more details of the node order selection. The chosen node order has a huge impact on the usability of a CH. We will first present a *simple* and *extensible* heuristic using a *priority queue* and then present several possible *priority terms* partitioned into five categories.

We can chose any node order to get a correct procedure. However, this choice has a huge influence on preprocessing and query performance. Our first observation is, that for the contraction of a node u, we only need to know the nodes v with $v > u$. So an iterative development of the node order $<$ is possible starting with the lowest node. We can manage a sequence $\langle v_1, \ldots, v_k \rangle$ of already contracted nodes ordered by $<$ and iteratively add $v_{k+1} \in V \setminus \{v_1, \ldots, v_k\}$ just prior to the contraction of v_{k+1}. The selection of the next node v_{k+1} is done using a priority queue. The priority of a node u is the linear combination of several priority terms and estimates the attractiveness to contract this node. A priority term is a particular property of the node u and can be related to the already contracted nodes and the remaining nodes. Thus the priority terms can change after the contraction of a node and need to be *updated*.

3.2.1 Lazy Updates

The priority queue has the node with the *smallest* priority on top, so *increasing* priorities can be updated in time by updating the priority of the node on top of the priority queue before it is removed. If this node is still on top after the update it will be removed, otherwise the new topmost node will be processed in the same way. This procedure is called *lazy updates* and yields improvements in practice and in some cases even in theory.

To further improve lazy updates, we also trigger an update of the priority of all remaining nodes, if there were recently too many lazy updates. A check interval t is given and if more than a certain fraction $a \cdot t$ of successful lazy updates occurs during a check interval, the update is triggered. We currently only use $a := 1$. A lazy update is successful if it changes the priority of the topmost node.

3.2.2 Edge Difference

Arguably the most important priority term is the edge difference. Intuitively, the number of edges in the remaining graph should decrease with the number of nodes. We considered two properties of u: the edge difference and the number of new edges after the contraction of u.

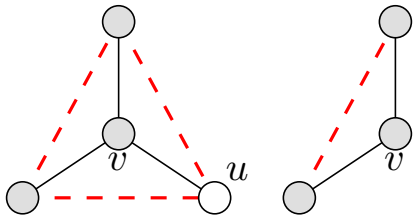
Let $G' = (V', E')$ be the remaining graph after the contraction of the direct predecessor of u and $G'' = (V'', E'')$ the remaining graph after the contraction of u. Note that we only *simulate* the contraction of u to calculate the edge difference and other properties.

Edge difference. The edge difference is calculated between the two graphs G' and G''. This is the most important parameter to get a good contraction, otherwise the remaining graph will most likely converge to the complete graph.

New edges. An edge is a new edge if and only if it needs to be added to G''. Note that not all shortcut edges are new edges, because sometimes just the edge weight function changes. This parameter did not prove to be useful and we omitted it in the presented experiments for the sake of simplicity.

For the implementation of the edge difference, we used the difference in the space requirements to store E' and E'', because it is quite similar to the theoretical edge difference $|E''| - |E'|$ but takes the space consumption into account. For example, *two* edges (v, w) and (w, v) with the same weight $w(v, w) = w(w, v)$ can be stored as only *one* edge with two additional forward and backward flags. This also applies to the number of new edges.

After the contraction of a node u, the edge difference of other nodes can change. In most cases only neighbors of u are affected since the edges incident to u do no longer belong to the remaining graph, see Figure 4. Our experience shows that updating all neighbors of u after its contraction is a good compromise between accuracy and performance.



(a) before (b) after

Figure 4: After the contraction of node u, the edge difference and the number of new edges of a neighbor v changes from $3 - 3 = 0$ to $1 - 2 = -1$. Note that an undirected edge is counted twice.

But an affected node v can be arbitrarily far away, see Figure 5 for an example. During the contraction of the node u a shortcut (x_r, y_r) is not necessary because of the witness path $\langle x_r, x'_2, \ldots, x'_r, v, y'_r, \ldots, y'_2, y_r \rangle$. Prior to the contraction of u, during a simulated contraction to calculate the edge difference of node v a shortcut (x_1, y_1) is not necessary because of the witness path $\langle x_1, x_2, \ldots, x_r, u, y_r, \ldots, y_2, y_1 \rangle$. After the contraction of u the only x_1-y_1-path

$$\langle x_1, x_2, \ldots, x_r, x'_2, \ldots, x'_r, v, y'_r, \ldots, y'_2, y_r, \ldots, y_2, y_1 \rangle$$

is not a valid witness path for the contraction of v because it has v in its interior. Thus a shortcut is necessary leading to an increased edge difference.

If after the contraction of a node u, the priority of a node v changes but is not updated in the priority queue, nodes may be extracted from the priority in a different order than with an update. Under certain conditions, lazy updates can reestablish the correct order.

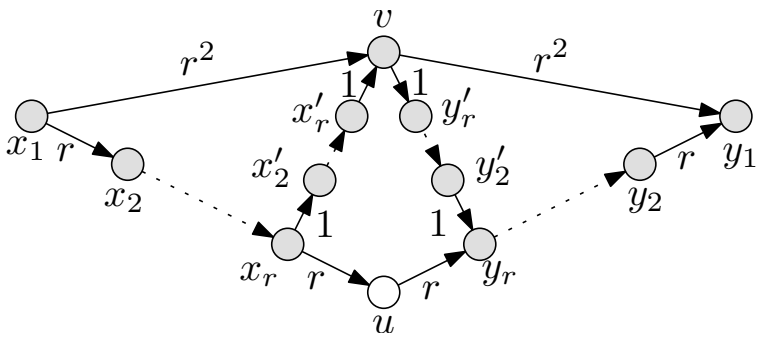


Figure 5: After the contraction of node v the edge difference and the number of the node u changes. Directed, weighted graph.

Lemma 1. *Nodes of a graph $G = (V, E)$ are contracted in the correct order if (a) after the contraction of a node all of its neighbors are updated, (b) the local searches for witnesses are unlimited, (c) the edge difference is the only priority term and has a positive coefficient, (d) lazy updates are used and (e) the minimum element is extracted from the priority queue.*

Proof. The edge difference of a node v depends only on the incoming edges $(x, v) \in E'$ and outgoing edges $(v, y) \in E'$ and the existing witnesses $P = \langle x, \ldots, y \rangle$. After the contraction of a node u, the incoming and outgoing edges change only for all neighbors of u. These changes are covered by (a). So only existing witness paths can affect the edge difference of a node v that is not a neighbor of u. Because of (b), we will not find a witness after the contraction of u if we found no one before. Thus witness paths can only vanish, leading to an increasing priority (c). Lazy updates (d) will adjust those priorities in time (e). □

The most limiting condition of the previous lemma is the unlimited local search. But in practice we need those limits to reduce the preprocessing time. However, if the search space limits are sufficiently large, we can expect only few errors in the node order.

3.2.3 Cost of Contraction

Using the edge difference as sole priority is a simple approach showing query times even faster than previous highway-node routing implementations [34]. But the preprocessing times can be quite large. Introducing a cost of contraction to the priority was the next logical decision. For the contraction of node u, we perform a Dijkstra search from each node v incident to an incoming edge (v, u) of u to find witness paths to each node w incident to an outgoing edge (u, w), see Section 3.3 for more details. Because we can stop the search once all such nodes w are settled and we can also limit the search (Section 3.3.1), those searches are called *local* searches. Those local searches to find witness paths have the biggest share of the preprocessing time. That is the reason why we chose the sum of the search space sizes of the local Dijkstra searches for the cost of contraction for a node u, more precisely the number of settled nodes.

We will introduce some optimizations later in Section 3.3 that do not use the Dijkstra algorithm for the local searches, in those cases we chose another appropriate quantity as the cost of contraction.

Keeping the cost of contraction up-to-date in the priority queue is quite costly. The contraction of a node in a search tree of the local search can affect the cost of contraction. So after the contraction of a node w, it would be necessary to recalculate the priority of each node that has w in their local search spaces. Most local search spaces are small but there are exceptions.

If there is an edge with a large edge weight, e.g. a long-distance ferry connection, the search space can grow arbitrarily and with it the number of nodes that trigger a change in the cost of contraction. A simplified example can be found in Figure 6.

In our implementation we will omit exakt updates for the sake of performance, update only the neighbors of the contracted node and eventually use lazy updates to cope with drastic increases of search space sizes.

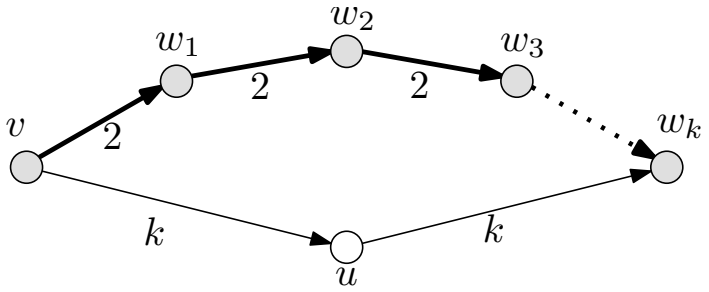


Figure 6: After the elimination of a node w_i, $1 \leq i \leq k$, the search space for the local search starting at node v changes and with it the cost of contraction of node u. Thick edges belong to the search tree, edge weights are explicitly given.

3.2.4 Uniformity

A central idea of highway-node routing is to speed up queries by searching in overlay graphs with fewer nodes and edges. If an node u is settled, only edges (u, v) with $v > u$ are relaxed, see Section 3.5. Doing so can drastically reduce the search space. But there are circumstances than can slow down a query. Think, for example, of a dead-end valley like in Figure 7. If the nodes are contracted in the order $u_1, u_2, \ldots, u_k$, a query starting at u_1 will relax the edge (u_1, u_2), then the node u_2 is settled and will relax the edge (u_2, u_3) and so on. The query settles all k nodes $u_1, \ldots, u_k$ until the exit x of the valley is reached. If you want to reach the exit x from u_1, the best thing would be a direct shortcut (u_1, x). But we do not want to store direct shortcuts for all pairs of nodes. Instead we will present a parameter that can reduce the number of settled nodes to $O(\log k)$.

Contracted neighbors. Count for each node the number of previous neighbors that have been contracted.

Given the dead-end valley in Figure 7 with arbitrary $k \in \mathbb{N}$ and a node ordering performed with the coefficients given in Table 1, between $u/3$ and $u/2$ nodes will be contracted having their contracted neighbor counter at 0. In the case of $u/3$ nodes, every third node is contracted, e.g. starting with u_1 (Table 1(b)) then u_4 (Table 1(c)), u_7 (Table 1(d)) and so on, and after that all the remaining nodes have their contracted neighbor counter at 1. Recursively applied, this yields a maximum contracted neighbor counter of $\lceil \log_{3/2} k \rceil$ for the nodes $u_1, \ldots, u_k$ in the dead end valley. This also limits the number of nodes that are settled, until a modified forward Dijkstra search, described in the query algorithm above, leaves the valley. The reason is the nature of the contracted neighbor counter. Whenever a node in this example is contracted, it has the minimum value of the contracted neighbor counters of all remaining nodes and it will increase the counter of its remaining neighbors by 1. The edges to those remaining neighbors are the ones that are going to be relaxed during a query.

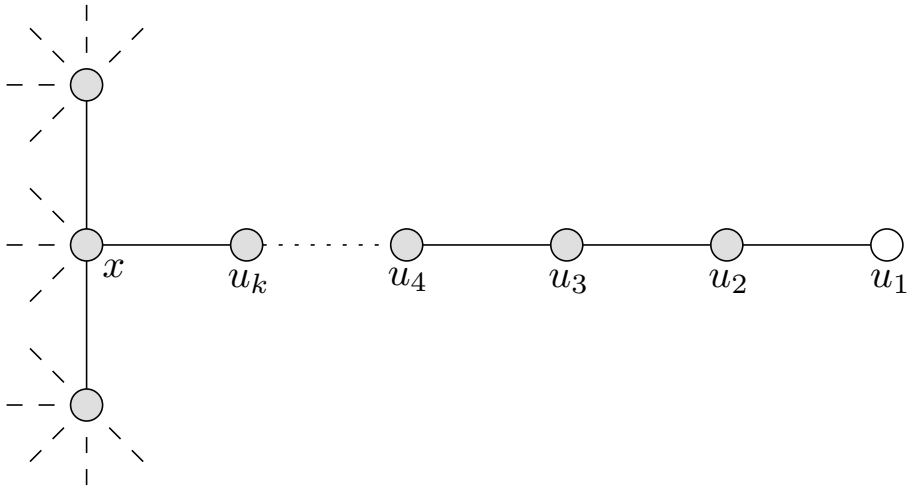


Figure 7: Example of a dead-end valley, all edges have unit distance. Among the nodes $u_1, \ldots, u_k$, always the rightmost node will be contracted, if the priority consists only of edge difference (rightmost node needs no shortcut and has one incident edge $\Rightarrow$ difference -1, all other need one shortcut and have two incident edges, the edge difference is always -1) and cost of contraction (rightmost node has one settled node, all other have two).

The implementation of the contracted neighbors counter is straightforward, we need one counter per node, independent of the number of edges. The update of this counter in the priority queue is quite fast as well, only the neighbors need to be updated.

Related to the contracted neighbors counter is another priority term, it also counts the number of already contracted nodes, but now on edges. To get a better intuition for this priority term, we can also say that we count the number of edges in the original graph a shortcut represents.

Sum of original edges of the new shortcuts. Count the number of original edges of the newly added shortcuts during the contraction of a node, see Figure 8. For convenience, we will refer to this term as the *original edges term*.

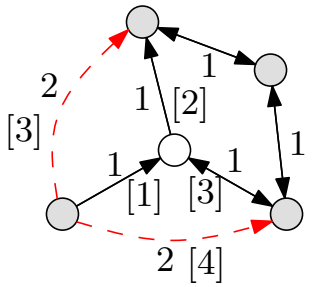


Figure 8: A directed, weighted graph. The numbers in brackets denote the number of original edges, an edge represents. Note that edges with [1] are original edges. The white node in the middle will be contracted. Two shortcuts, represented by dashed arrows, are necessary. The sum of original edges of the new shortcuts is $3 + 4 = 7$.

This property should avoid shortcuts that represent too many original edges. The motivation is to use it for transit-node routing where the current implementation [34] only supports few node levels. We create those few, e.g. 5, levels by defining ranges on the levels of a CH

(a) initialization

		u_k	$\cdots$	u_2	u_1
edge difference	1×	-1	-1	-1	-1
search space	1×	2	2	2	1
contr.neighbors	3×	0	0	0	0
priority		1	1	1	0

(b) after contraction of node u_1

	u_k	$\cdots$	u_4	u_3	u_2
1×	-1	-1	-1	-1	-1
1×	2	2	2	2	1
3×	0	0	0	0	1
	1	1	1	1	3

(c) after contraction of node u_4

	u_k	$\cdots$	u_6	u_5	u_3	u_2
1×	-1	-1	-1	-1	-1	-1
1×	2	2	2	2	2	1
3×	0	0	0	1	1	1
	1	1	1	4	4	3

(d) after contraction of node u_7

	u_k	$\cdots$	u_8	u_6	u_5	u_3	u_2
1×	-1	-1	-1	-1	-1	-1	-1
1×	2	2	2	2	2	2	1
3×	0	0	0	1	1	1	1
	1	1	1	4	4	4	3

Table 1: Node order selection including contracted neighbors. Edge difference and search space account for the priority with coefficient 1, contracted neighbors with coefficient 3. Tables (a)–(d) show the evaluation of the priority terms and the final priority for each node during first three possible steps of contraction. Note that other node orders are possible since not every node has a distinct priority.

and assign each range to one new level. It is important, that from each node in a level, the next level can be reached via a path with only a few hops. So having shortcuts representing only a few original edges will hopefully lead to a hierarchy with this property. And fast path unpacking routines that store complete representations of shortcuts may also benefit from this new priority term. Their memory consumption is lowered and that is especially important on mobile devices.

For our implementation, an additional original edge counter is necessary for each shortcut leading to space requirements linear in the number of shortcuts[1]. So for graphs with many edges and shortcuts it may be necessary to omit this priority term and just use Voronoi regions or contracted neighbors because their space requirements are only linear in the number of nodes. Another possible implementation could count the number of original edges by unpacking the shortcut, however, this inflicts a lot of overhead leading to a time consuming node ordering process. Since this priority term depends on the number of necessary shortcuts, it has at least the same constraints regarding updates as the edge difference. So the contraction of nodes, that are arbitrarily far away, can affect this property.

Counting contracted neighbors is simple to implement but it seems a little bit ad-hoc to ensure uniform contraction. The next approach uses something more specific to this task.

Definition 1. *Let u be currently the highest contracted node, so all $v \in G$ with $v \leq u$ are contracted and all $v \in G$ with $v > u$ are not contracted. The Voronoi region of $v \in G$, $v > u$ is defined as follows:*

$$R(v) := \{x \in G \mid x \leq u \text{ and } d(v,x) < \infty \text{ and } \forall y \in G, y > u \Rightarrow d(v,x) \leq d(y,x)\} \cup \{v\}$$

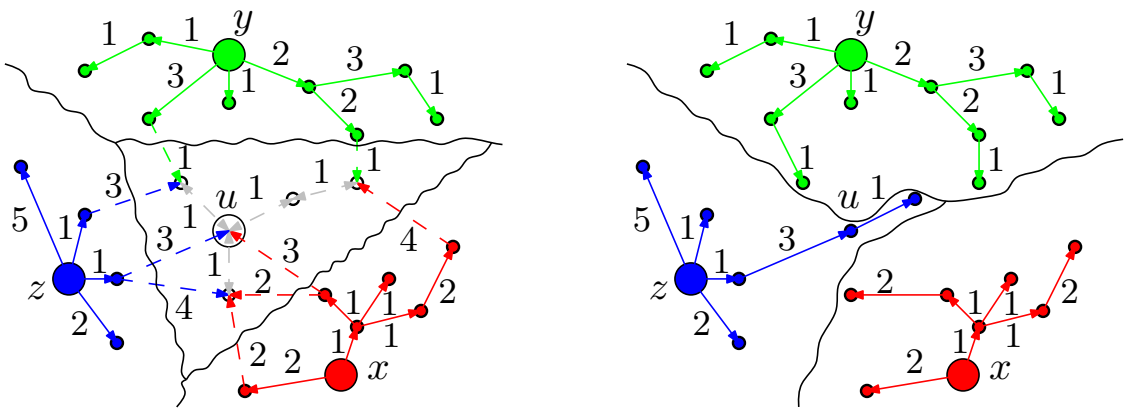
The Voronoi regions based on shortest paths should include a certain amount of uniformity into our contraction hierarchies. The goal is to contract nodes with a bigger Voronoi region later.

[1]For a simpler implementation, even original edges have this additional counter.

Voronoi region. Count the number of already contracted nodes in each Voronoi region $R(u)$ of a remaining node u. Because the number of nodes in the Voronoi regions of the remaining nodes grows quite fast, we also extract the square root before we add this property to the linear combination of the priority. If we think of the Voronoi region as a disc and of the number of nodes as its area, extracting the square root calculates something like the diameter of the disc.

Note that usually Voronoi regions are defined on metric spaces, but our graph G with shortest path distance d is not necessarily one because we allow directed graphs and thus d may not be symmetric. The triangle inequality however is satisfied. With the definition of $R(u)$ above, it is possible that two regions may overlap. In practice, a node that is on the border of two or more regions is assigned to only one region to allow an efficient management of all regions.

After the contraction of a node u, it is necessary to distribute the nodes in $R(u)$ among all neighboring Voronoi regions. Basically this is done with a modified Dijkstra algorithm, the same algorithm as described in [26] for partitioning. The priority queue is initialized with each neighbor of u and distance 0. Each settled node that previously belonged to u recursively belongs to the region that its parent belongs to, see Figure 9 for an example.



(a) before (b) after

Figure 9: The white node u in the middle is contracted. The waved lines denote possible borders of the Voronoi regions, solid arrows affiliation to a Voronoi region and dashed arrows possible new affiliations. Because our graph is not necessarily a metric space, there are no unique border lines. All the nodes in $R(u)$ including u are assigned either to $R(x)$, $R(y)$ or $R(z)$ using a modified Dijkstra algorithm.

For an efficient implementation, it is necessary to know all neighbors in terms of shared borders of Voronoi regions of the contracted node u. There must not necessarily exist edges to all of these neighbors in the remaining graph. But if we know all nodes in $R(u)$ then we can use the edges incident to those nodes to find the neighboring Voronoi regions. Our implementation saves the nodes in $R(u)$ as a single linked list and their distance to u as an array. This allows the application of the following algorithm that only needs to regard all nodes in $R(u)$ and its incident edges. It has the same result as the basic algorithm above but it is more suitable for the specific task of updating the Voronoi regions. After the contraction of u, we scan through the linked list and check for each node v all incident edges (w, v). If such a w is found with $w \in R(x)$, with $x \in G$, x not contracted, add or update the node v in the priority queue with distance $d(x, v) = d(x, w) + w(w, v)$. After that initialization of the priority queue, always settle

the node with the lowest distance and add it to the Voronoi region of its parent. Only edges leading to $R(u) \cup \{u\}$ need to be relaxed, this does not affect the correctness of the algorithm since the Voronoi regions are based on shortest paths. It is possible, that there are contracted nodes that do not belong to any Voronoi region as Figure 10 illustrates.

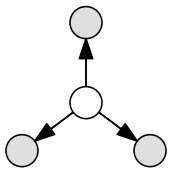


Figure 10: A directed, unweighted graph. There exists no path from one of the grey nodes to the white node in the middle. So after the contraction of the white node, it will not be in any Voronoi region of the remaining nodes.

Compared to the counting of contracted neighbors, Voronoi regions have more processing overhead because of the additional step of distributing the nodes in the Voronoi region of the currently contracted to the neighboring Voronoi regions.

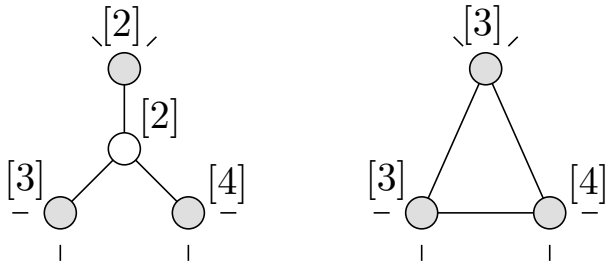
3.2.5 Cost of Queries

The uniformity properties of the previous section are useful to speed up the queries in the resulting contraction hierarchy. Adding a property that tries to limit the size of the query search space results in even faster query times. More precisely we want to limit the depth of the shortest paths tree for any source node $s \in V$ and target node $t \in V$. Remember, that a query using a modified Dijkstra algorithm relaxes only edges to nodes with higher order than the current node. Let $\text{depth}(s, v)$ be the depth of the node v in the shortest paths tree grown by our query algorithm with source node s, if v is not in the tree, the value is -1. Let $\text{depth}(s) := \max\{\text{depth}(s, v) \mid v \in V\}$. For simplification, we ignore the direction of the edges and regard every edge as undirected edge.

Search space depth. We want to reduce the depth of the shortest paths trees our modified Dijkstra algorithm grows during a query. The later a node is contracted, the higher is its order, and it is more likely that it increases the depth of an shortest paths tree. So if there is a node u that is still not contracted, has a contracted neighbor v and there is a node $s \in V$ with a large value for $\text{depth}(s, v)$, node u should be contracted later since there is a chance that $\text{depth}(s, u) = \text{depth}(s, v) + 1$. If node u is contracted later, it is more unlikely that there will be another node u' that now has u as contracted neighbor with $\text{depth}(s, u') = \text{depth}(s, u) + 1$. We use an upper bound on the search space depth as priority term.

Now we will explain how we implemented the above priority term. Our upper bound is a rough estimation but simple to implement and effective. We only need an integer array A with an entry for each node in G initialized with $\langle 0, \ldots, 0 \rangle$. After the contraction of node $u \in G$ we update each adjacent node $v \in G$ that is not already contracted with

$$A[v] := \max\{ \overbrace{A[v]}^{\text{current upper bound}} , \underbrace{A[u] + 1}_{\text{upper bound including } u} \}.$$



(a) before (b) after

Figure 11: Detail of a graph before and after the contraction of the white node. The number in brackets denotes the current upper bound $A[\cdot]$.

Figure 11 shows an example.

Since the depth of a directed tree is equal to the maximum number of nodes (minus 1) on a path starting at the root node, it is easy to use a simple induction over the number of nodes in a such a path to prove the correctness of the upper bound. For a path $\langle x_1, \ldots, x_{r-1}, x_r \rangle$ apply the induction hypothesis to the subpath $\langle x_1, \ldots, x_{r-1} \rangle$.

3.2.6 Global measures

We can prefer contracting globally unimportant nodes based on some path based centrality measure such as betweenness [9, 1] or reach [17]. But we cannot straightforward use the centrality value of a node as a priority term since the range of values is too large. For example betweenness centrality values range from 0 to about n^2, for reach, the values range from 0 to half of the diameter of the graph. Instead of applying functions like the square root or the logarithm, we propose a technique independent of the range of the values. Our idea is to adjust the contribution to the priority of a node relative to the centrality values of its adjacent nodes in the remaining graph.

For a remaining node $u \in G$, let $C(u) \in \mathbb{R}$ be the centrality value and $N(u) \subseteq V$ the set of remaining adjacent nodes. Then for node u, we add the fraction of neighbors with smaller centrality value to the priority, see Figure 12, with an appropriate coefficient:

$$\frac{|\{v \in N(u) \mid C(v) < C(u)\}|}{|N(u)|} \text{ or 0 if and only if } N(u) = \emptyset.$$

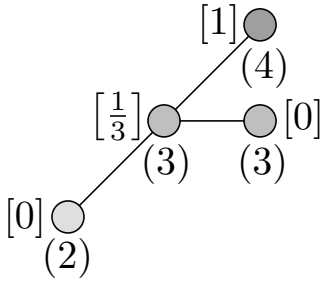


Figure 12: Undirected graph, the number in parentheses denotes the centrality value and the number in brackets the fraction of neighbors with smaller centrality value.

The set $N(u)$ of remaining adjacent nodes can change after the contraction of a adjacent node, directly implying the update instruction of the above property.

Dealing with large graphs, it is difficult to get exact centrality values. Therefore we used approximations. In detail, we used canonical betweenness [10] because we had very good approximation algorithms and precomputed approximation data available.

3.3 Node Contraction

This section fills in the details that are omitted in Algorithm 1, especially the part how witness paths are located. We will first describe a general approach using Dijkstra's algorithm and then present several optimizations to achieve faster preprocessing times.

Let $G' = (V', E')$ be the remaining graph after the contraction of the direct predecessor of u. For the contraction of a node u, we face a many-to-many shortest path problem from source nodes $v \in S := \{v \mid (v, u) \in E'\}$ incident to incoming edges of u to all target nodes $w \in T := \{w \mid (u, w) \in E'\}$ incident to outgoing edges of u. For such a pair $v \neq w$, we want to decide whether $\langle v, u, w \rangle$, if it is a shortest v-w-path, is the only shortest v-w-path in G'. A simple way to implement this is to perform for each source node v a forward shortest-path search starting at v in the current remaining graph G' excluding u until all target nodes $T \setminus \{v\}$ are settled, see Figure 13. Such a limited search is called a *local search*.

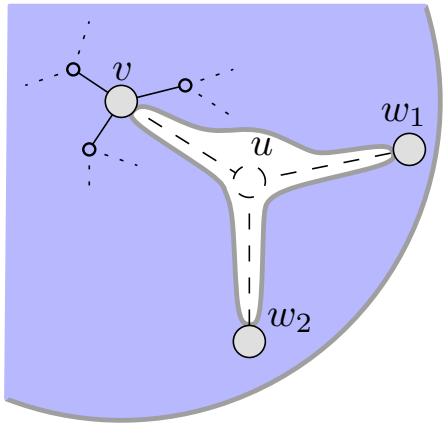


Figure 13: Forward search starting at node v until the target nodes w_1 and w_2 are settled. The node u is ignored during the search. The search space is framed by thick gray border lines.

Let $d_v(w)$ be the shortest path distance found by this search. We add a shortcut edge if and only if $d_v(w) > w(v, u) + w(u, w)$, i.e. if the shortest v-w-path excluding u will be longer. We can additionally stop the search from a node x when it has reached distance $w(v, u) + \max\{w(u, w) \mid (u, w) \in E' \setminus \{(u, v)\}\}$.

3.3.1 Limit Local Searches

Local searches to locate witness paths can be quite time consuming, especially if long-distance edges like ferry connections are involved. To create a node order and a CH based on this node order, we currently perform those local searches at least twice for each edge. The first search happens during a simulated node contraction to calculate the edge difference and other priority terms, and the last search is performed to actually contract the node. Between those two searches there may be additional local searches to update several priority terms. We do not store the first search because it possibly needs to be updated after shortcuts are added during the contraction of nodes

We propose two approaches to limit the local searches. Note that the limitation of local searches does not invalidate our algorithm since only a subset of witness paths is found resulting in a superset of shortcut edges. However, due to additional shortcuts, the remaining graph will be more dense and subsequent contractions are more time consuming. Therefore the local search limit needs to be carefully selected and should possibly change after the contraction of a certain number of nodes.

Limit the number of settled nodes. A local search, implemented as a modified Dijkstra algorithm, can be stopped after a certain number of nodes is settled.

Limit the number of hops / edges from the start node. Only find shortest-paths with a limited number of edges. We will call this *hop limit*.

Limiting the number of settled nodes is straightforward, but, in our experience, leads to more dense remaining graphs and does not speed up the contraction a lot. However, if we only use it to estimate the edge difference and the number of new edges and perform the actual contraction without limit, it speedups the node ordering without severe disadvantages.

Hop limits are the stall-in-advance technique introduced by highway-node routing [34]. Stall-in-advance limits the number of hops of the local search from the first covered node. In CHs, if we want to contract a node u in the remaining graph, all nodes except for u will have higher order than u. So all those nodes are covering nodes and stall-in-advance equals a simple hop limit. To achieve further improvements, we propose *staged* hop limits. We start the node ordering and contraction with a small hop limit, e.g. 1, this is enough to contract nodes that do not require shortcut edges and all witness paths consist of one edge. Then we switch to higher hop limits because otherwise the remaining graph will get to dense. We measure the density of the remaining graph with the average node degree of the remaining nodes. Therefore it seems natural to use the average node degree to trigger the hop limit switches. After a hop limit switch, it is necessary to recalculate the priority of all remaining nodes. If the edge difference has sufficient weight in the priority term, we observe that this leads to a temporary decrease of the average node degree. So it makes sense to increase hop limits, but with the same average node degree to trigger the switch.

Fast Local 1-Hop Search. For a 1-hop search, we do not require a modified Dijkstra algorithm. We just scan through all incident edges of the start node v to find the target node w, see Figure 14. This is repeated for each pair of edges $(v,u),(u,w) \in E'$ with $v \neq w$. Since an edge array is used, all edges likely remain in cache making this operation very fast. The 1-hop search make sense if lots of edges of the graph are shortest paths, like in a road network. In this case, a 1-hop search should allow to contract a significant amount of nodes without too many additional shortcuts being added. We use as cost of contraction (priority term) the number of pairs $(v,u),(u,w) \in E'$ with $v \neq w$.

Fast Local 2-Hop Search. In case of a 2-hop limit, we implemented a simple variant of the many-to-many shortest path algorithm described in [21] instead of performing a local forward search from each $v \in S$. A bucket $b(x)$ is associated with each node $x \in V'$. Bucket $b(x)$ stores a set of pairs (w,d) representing paths from x to $w \in T$ with length d encountered during a 1-hop backward search from all target nodes, see Figure 15. During the following 1-hop forward search from a node $v \in S$, we maintain an array D of *tentative distances* to each target node. These distances are initialized to ∞ (or some sufficiently large value). The forward search,

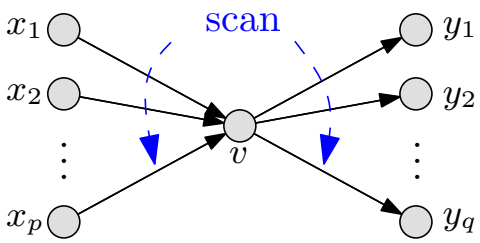


Figure 14: The local 1-hop search starting at node v scans through all remaining incident edges.

starting at v, scans $b(x)$ for each edge $(v, x) \in E' \setminus \{(v, u)\}$. For each pair (w, d) stored in $b(x)$, it sets $D[w]$ to $\min\{D[w], w(v, x) + d\}$. After that, if $D[w]$ is larger than $w(v, u) + w(u, w)$, a shortcut edge (v, w) is added. To find 1-hop witnesses of the form (v, w), we can just scan the bucket $b(v)$ after all backward searches. We use as cost of contraction (priority term) the number of bucket entries plus the number of scanned edges during the 1-hop forward search.

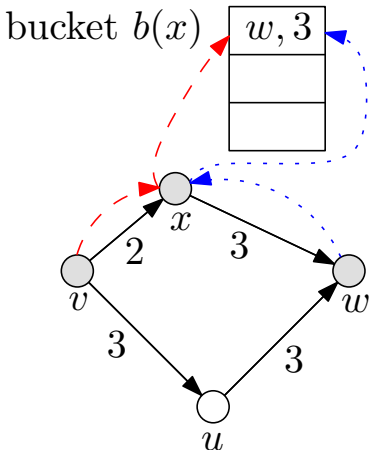


Figure 15: The 1-hop backward search from w stores distance $w(x, w)$ into bucket $b(x)$, represented by dotted arrows. The following 1-hop forward search from v scans bucket $b(x)$ to get distance to w over x, represented by dashed arrows.

1-Hop Backward Search. To speed up a local search from node $v \in S$ with hop limit $a \geq 3$, we can first perform a $(a-1)$-hop forward search using a modified Dijkstra's algorithm. The distance to a node $w \in T$, whose shortest path from v has no more than a edges, is either already known because w is settled ($\leq a-1$ edges) or can be found by a 1-hop backward search using

$$d_v(w) = \min\{d_v(x) + w(x, w) \mid (x, w) \in E' \text{ and } x \text{ settled}\} \quad .$$

The distance limit to the forward search changes, we now stop the search if the last settled node exceeds the distance

$$w(v, u) + \max\{w(u, w) - \min\{w(x, w) \mid (x, w) \in E'\} \mid (u, w) \in E' \setminus \{(u, v)\}\} \quad .$$

An example with $a = 5$ is illustrated in Figure 16.

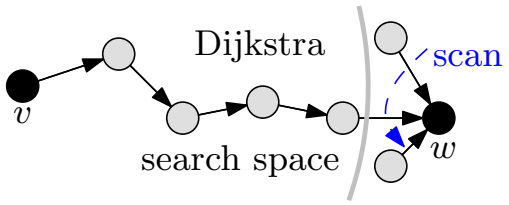


Figure 16: A local search from v to w. The Dijkstra search space is limited by 4 hops. The 5th hop is added performing a 1-hop backward search from w by scanning through all incident backward edges.

3.3.2 On-the-fly Edge Reduction

CHs are based on the concept of node contraction. There is no subsequent edge reduction phase with considerable time needs like for HHs [30, 31, 34]. Nevertheless edge reduction is meaningful if it removes only edges that are not on any shortest path. Such an edge reduction can be partially incorporated in the node contraction phase without serious costs. If the search for witnesses is performed by a local Dijkstra search from each node $v \in S$, it contains additional information for a simple edge reduction. If there is an edge $(v, x) \in E'$ with $d_v(x) < w(v, x)$, then this edge is dispensable for the shortest paths calculation and can be removed, see Figure 17. The main cost of the additional edge reduction is to scan through all remaining incident edges (v, x) of node v and to read $d_v(x)$. Because v was the start node of a just finished local Dijkstra search, all this information is likely to be in cache and will cause almost no overhead. Moreover, since the remaining graph will be more sparse, it can speedup the contraction of remaining nodes. And since only edges that are not on any shortest path are removed, shortest path queries will be faster.

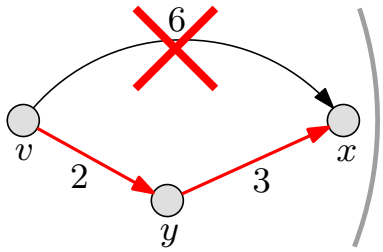


Figure 17: A local search from v in a directed weighted graph. Since $d_v(x) = 5 < 6 = w(v, x)$, the edge (v, x) can be removed because it is not necessary for shortest paths calculation.

3.4 Relation to Highway-Node Routing

To put our work in the context of highway-node routing (HNR) [34], we need to adopt some concepts, more precisely the concept of *canonical shortest paths*, *covering paths* and *overlay graphs*. HNR is a general approach for hierarchical routing. It does relay on another technique to define a hierarchy on the node set. CHs are an extreme case where every node has its own level. We use the query algorithm of HNR, so we can adopt the correctness proof for the query algorithm if we can prove that CHs are an application of HNR.

For readers who are not familiar with the concepts of HNR, we now repeat some of the definitions from [34].

Canonical Shortest Paths. For a given Graph $G = (V, E)$, $\mathcal{U}(G)$ is a set of *canonical shortest paths* if it contains for each connected pair $(s, t) \in V \times V$ exactly one unique shortest path from s to t such that $P = \langle s, \ldots, s', \ldots, t', \ldots, t \rangle \in \mathcal{U}(G)$ implies that $P|_{s' \to t'} \in \mathcal{U}(G)$.

It is easy to see that Dijkstra's algorithm always finds canonical shortest paths if we have a total order on the nodes and, in case of ambiguities, prefer the parent node with the *higher* order. For a CH $(G, <)$ we use the set $\mathcal{U}(G, <)$ of canonical shortest paths found by Dijkstra's algorithm and the node order $<$.

Covering-Paths Set. We consider a graph $G = (V, E)$, a node subset $V' \subseteq V$, a node $s \in V$, and a set $C \subseteq \{\langle s, \ldots, u \rangle \mid u \in V'\}$ of paths in G.

The set C is a *covering-paths set* of s w.r.t. V' if for any node $t \in V'$ that can be reached from s, there is a node $u \in V'$ on some shortest s-t-path P such that $P|_{s \to u} \in C$, i.e.,

$$P = \langle \underbrace{s, \ldots, \overbrace{u}^{\in V'}}_{\in C}, \ldots, \overbrace{t}^{\in V'} \rangle$$

A covering-paths set C is a *canonical covering-paths set* if for any node $t \in V'$ that can be reached from s, there is a node $u \in V'$ on the *canonical* shortest s-t-path P such that $P|_{s \to u} \in C$.

Overlay Graph. For a given graph G and a node set V', the graph $G' = (V', E')$ is an *overlay graph* of G if for all $(u, v) \in V' \times V'$, we have $d_{G'}(u, v) = d_G(u, v)$. Figure 18 gives an example. An overlay graph is a subgraph that preserves the shortest path distances of G.

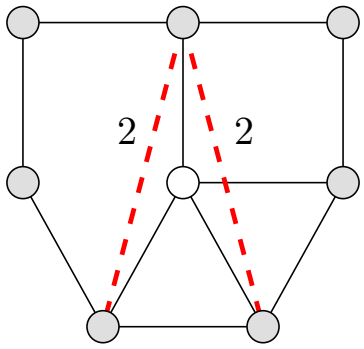


Figure 18: A minimal overlay graph (shaded nodes, all incident edges, dashed shortcuts with given edge weights) of the depicted graph (solid edges, unit edge weights).

Multi-Level Overlay Graph. The overlay graph definition can be applied iteratively to define a multi-level hierarchy. For given *highway-node sets* $V =: V_1 \supseteq V_2 \supseteq \cdots \supseteq V_L$, we define the *multi-level overlay graph* $\mathcal{G} = (G_1, G_2, \ldots, G_L)$ in the following way: $G_1 := G$ and for each $\ell \geq 1$, $G_{\ell+1}$ is an overlay graph of G_ℓ.

Now we will apply those concepts to contraction hierarchies to prove the following theorem.

Theorem 1. *Let $(G = (V, E), <)$ be a contraction hierarchy, $V = \langle v_1, v_2, \ldots, v_n \rangle$ with $v_1 < v_2 < \cdots < v_n$, $V_i := \{v_i, v_{i+1}, \ldots, v_n\}$ for $i \in \{1, 2, \ldots, n\}$. Then for the highway-node sets $V = V_1 \supseteq V_2 \supseteq \cdots \supseteq V_n$, $\mathcal{G} := (G[V_1], G[V_2], \ldots, G[V_n])$ is a multi-level overlay graph.*

Before we can prove the theorem, we need some additional lemmas.

Lemma 2. *Let $u \in V$, $G_u := (V_u, E_u) := G[\{v \in V \mid v \geq u\}]$. Then all edges incident to u that are in E_u are already in the remaining graph after the contraction of the node $y := \max\{x \in V \mid x < u\}$ that is contracted directly before u.*

Proof. During the contraction of node u, no shortcut incident to u is added, only shortcuts incident to adjacent nodes of u are added. And after the contraction of node u, only edges (v, w) with incident nodes $v, w > u$ are considered, so no new edge incident to u is added after the contraction of the node y. □

Lemma 3. *Let $u \in V$, $G_u := (V_u, E_u) := G[\{v \in V \mid v \geq u\}]$. The search for witnesses in Algorithm 1 during the contraction of node u equals the computation of the canonical covering-paths sets for all nodes $v \in V'_u := V_u \setminus \{u\}$ w.r.t. to V'_u and the set $\mathcal{U}(G, <)$ of canonical shortest paths.*

Proof. We modify Dijkstra's algorithm so that it prefers shortest paths that do not have u in their interior but still computes a canonical shortest paths set. This "preference" is expressed in Line 4 with the term "may be". We need canonical covering-paths sets $C(v)$ for all nodes $v \in V'_u$ w.r.t. to V'_u. Two cases are distinguished:

- *Case 1:* There is an incoming edge $(v, u) \in E_u$. Let

  $$C(v) := \{\langle v, w\rangle \mid (v, w) \in E_u \text{ and } w \neq u\} \cup$$
  $$\{\langle v, u, w\rangle \mid (u, w) \in E_u \text{ and } \langle v, u, w\rangle \text{ "may be" the only shortest path from } v \text{ to } w\},$$

  then $C(v)$ can be calculated using the witnesses found during the contraction of node u, since (v, u) and every edge $(u, w) \in E_u$ has been considered during the contraction of node u as proven in Lemma 2. $C(v)$ is by definition a covering-paths set of v w.r.t. to V'_u. Now we need to prove that it is a *canonical* covering-paths set. If not, for the sake of contradiction, there must be a $w \in V'_u$ with $(u, w) \in E_u$ and $\langle v, u, w\rangle \in \mathcal{U}(G, <)$ but another shortest v-w-path P is found with only nodes $> u$ in its interior, preventing the addition of a shortcut (v, w). Let $P = \langle v, \ldots, x, w\rangle$, then $x > u$ and by our construction of $\mathcal{U}(G, <)$ with a modified Dijkstra algorithm, P would be favored over $\langle v, u, w\rangle$ as the canonical shortest v-w-path, so $\langle v, u, w\rangle$ is not in $\mathcal{U}(G, <)$, contradiction.

- *Case 2:* There is no incoming edge $(v, u) \in E_u$. Then $C(v) := \{\langle v, w\rangle \mid (v, w) \in E_u\}$ is a trivial canonical covering-paths set of v w.r.t. to V'_u since every adjacent node of v is in V'_u.

□

Proof of Theorem 1. Let $u \in V \setminus \{v_n\}$, for the contraction of node u, we perform a witness search, see Algorithm 1. According to Lemma 3, this equals the computation of canonical covering-paths sets for all nodes $v \in V'_u := V_u \setminus \{u\}$ w.r.t. to V'_u. As proven in [34], we can use those canonical shortest-paths sets to compute an overlay graph G'_u for the graph G_u and the node set V'_u. This is done by appending edges (v, w) for each $v \in V'_u$ and $\langle v, \ldots, w\rangle \in C(v)$ with weight $w(u, v) := w(\langle v, \ldots, w\rangle)$. According to the definition of $C(v)$ in Lemma 3, only edges (v, w) with $\langle v, u, w\rangle \in C(v)$ need to be added, all other edges already exist. Those edges (v, w) are exactly the shortcut edges added in Line 5 of Algorithm 1. For $u' := \min V'_u$, those shortcuts are already in $G_{u'}$ along with all edges in E_u that are not incident to u. Note that the shortcuts added during the contraction of node u do not need to be shortest paths and can be replaced by even shorter shortcuts, but this does not invalidate that the graph $G_{u'}$ is an overlay graph

of G_u because the graph G'_u is one. Since $u \in V \setminus \{v_n\}$, there exists a $k \in \{1, 2, \ldots, n-1\}$, $u = v_k$ and $u' = v_{k+1}$. By the definition of G_u and $G_{u'}$, $G_u = G[V_k]$ and $G_{u'} = G[V_{k+1}]$. So $G[V_1] = G$ and since $u \in V \setminus v_n$ arbitrarily chosen, for each $\ell \in 1, 2, \ldots, n-1$, $G[V_{\ell+1}]$ is an overlay graph of $G[V_\ell]$ finally proving that $\mathcal{G}$ is an multi-level overlay graph. □

In a nutshell, Algorithm 1 iteratively constructs overlay graphs for a given node order resulting in an *multi-level overlay graph* with n distinct levels. So the query algorithms for HNR can be applied, the topic of the next section.

3.5 Query

Our query algorithm is a symmetric Dijkstra-like bidirectional procedure. We use the asynchronous, aggressive variant from highway-node routing, see [34]. We will shortly repeat the most important definitions and algorithms and emphasize their characteristics applied to contraction hierarchies followed by a correctness proof. After that, we describe how to unpack a shortest path, i.e. how to get a shortest path in the original graph without any shortcut edges. Finally, the query speedup technique *stall-on-demand* is explained.

The query algorithm does not relax edges leading to nodes lower than the current node. This property is reflected in two *search graphs*. The *upward graph*

$$G_\uparrow := (V, E_\uparrow) \text{ with } E_\uparrow := \{(u, v) \in E \mid u < v\})$$

and, analogously, the *downward graph*

$$G_\downarrow := (V, E_\downarrow) \text{ with } E_\downarrow := \{(u, v) \in E \mid u > v\}).$$

We perform forward search in $G_\uparrow$ and a backward search in $G_\downarrow$, see Figure 19 for two examples.

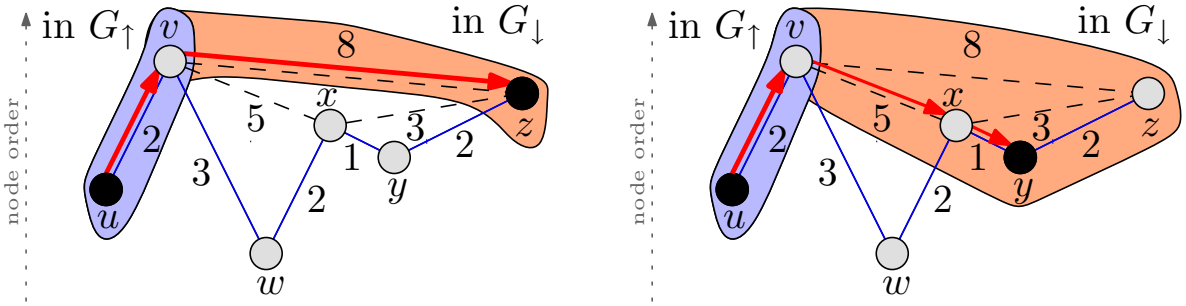


Figure 19: Two examples of queries. The graph is directed, the numbers on the edges denote the edge weights. Normal arrows denote edges in the original graph, dashed arrows denote shortcuts and thick arrows the path found by the query algorithm. The forward and backward search space is framed.

Forward and backward search are interleaved, we keep track of a tentative shortest-path length and abort the forward/backward search process when all keys in the respective priority queue are greater than the tentative shortest-path length (abort-on-success criterion). Note that we are *not* allowed to abort the entire query as soon as both search scopes meet for the first time. The reason for this is illustrated in Figure 20. A interleaved, bidirectional Dijkstra search is started from source node s and target node t. At first, the forward search settles s

and the backward search settles t. In the second step, both search scopes settle x and they meet. But the path $\langle s, x, t\rangle$ is not a shortest s-t-path. Only in the third step, the backward search settles y and relaxes the edge (s, y), and the shortest s-t-path $\langle s, y, t\rangle$ is found for the first time.

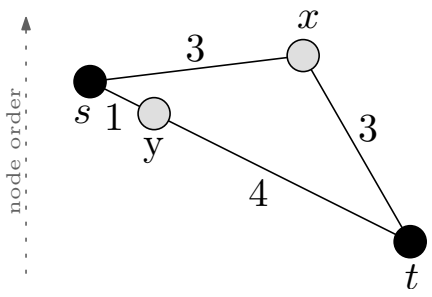


Figure 20: Undirected graph, the numbers on the edges denote the edge weights. Example explaining the necessity of the abort-on-success criterion of the query.

Theorem 2. *The query algorithm is correct.*

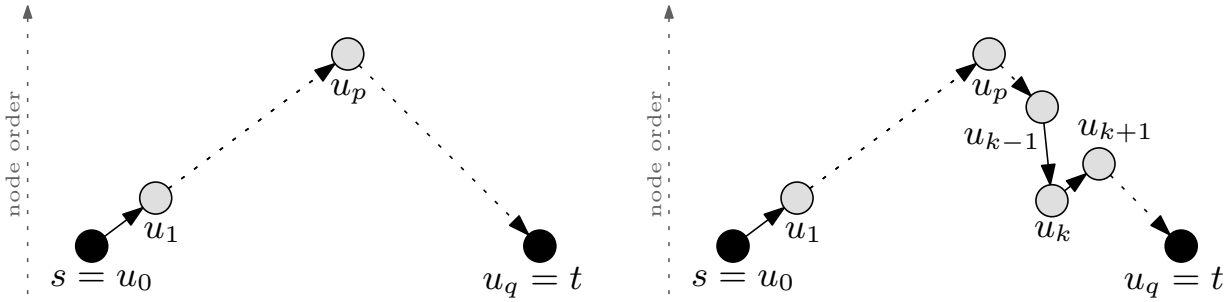
Proof. In Section 3.4, Theorem 1 we show that a contraction hierarchy is a multi-level overlay graph, so the correctness proof in [34] for the asynchronous, aggressive query algorithm is applicable. For better understanding, we present a self-contained proof specific for contraction hierarchies.

Let $(G = (V, E), <)$ be a contraction hierarchy. Let $s, t \in V$ be a *source* and a *target* node. By the definition of a shortcut, it only represents an already existing path in the graph, the shortest path distance between s and t in the CH is the same as in the original graph, especially the existence of a shortest s-t-path is the same. Every shortest s-t-path in the original graph still exists in the contraction hierarchy but there may be additional shortest s-t-paths. But since we use a modified Dijkstra algorithm that does not relax all incident edges of a settled node, our query algorithm does only find particular ones. In detail, exactly the shortest paths of the form

$$\langle s = u_0, u_1, \ldots, u_p, \ldots, u_q = t\rangle \text{ with } p, q \in \mathbb{N}, \tag{1}$$

$$u_i < u_{i+1} \text{ for } i \in \mathbb{N}, i < p \text{ and } u_j > u_{j+1} \text{ for } j \in \mathbb{N}, p \leq j < q.$$

are found by our query algorithm, see Figure 21(a). We will prove that if there exists a shortest s-t-path then there exists a shortest s-t-path of the form (1), too.



(a) found by query algorithm

(b) not found by query algorithm

Figure 21: Examples of shortest s-t-paths with respect to their necessary form for the query algorithm.

Let $P = \langle s = u_0, u_1, \ldots, u_p, \ldots, u_q = t \rangle$ with $p, q \in \mathbb{N}$, $u_p = \max P$, be a shortest s-t-path that is not of the form (1), like e.g. in Figure 21(b), then there exists a $k \in \mathbb{N}, k < q$ with $u_{k-1} > u_k < u_{k+1}$. Our query algorithm will not find P but another shortest s-t-path that can be recursively constructed out of P. Let $M_P := \{u_k \mid u_{k-1} > u_k < u_{k+1}\}$ the set of local minima excluding nodes s, t. We show that there exists a shortest s-t-path P' with $M_{P'} = \emptyset$ or $\min M_{P'} > \min M_P$. Let $u_k := \min M_P$ and consider the two edges $(u_{k-1}, u_k), (u_k, u_{k+1}) \in E$. Both edges already exist at the beginning of the contraction of node u_k according to Lemma 2. So there is either a witness path $Q = \langle u_{k-1}, \ldots, u_{k+1} \rangle$ consisting of nodes higher than u_k with $w(Q) \leq w(u_{k-1}, u_k) + w(u_k, u_{k+1})$ (even $=$ since P is a shortest path) or a shortcut (u_{k-1}, u_{k+1}) of the same weight is added. So the subpath $P|_{u_{k-1} \to u_{k+1}}$ can either be replaced by Q or by the shortcut (u_{k-1}, u_{k+1}). The resulting path P' is still a shortest s-t-path consisting of nodes higher than u_k. Since $n < \infty$, there must be a shortest s-t-path P'' with $M_{P''} = \emptyset$ equal to P'' is of the form described in (1). This path P'' can be found by our query algorithm proving its correctness. However, note that there need not be a unique shortest s-t-path of the form (1), it is sufficient that there is at least one. □

Both search graphs $G_\uparrow$ and $G_\downarrow$ can be represented in a single, space-efficient data structure: an adjacency array. Each node has its own edge group of incident edges. Since we perform forward search in $G_\uparrow$ and backward search in $G_\downarrow$, we only need to store an edge in the edge group of the less important incident node. More about our implementation of graph data structures can be found in Appendix A.

3.5.1 Outputting Complete Path Descriptions

The query algorithm described above can return a shortest path in the contraction hierarchy. In order to output a complete description of the computed shortest path, we have to unpack the shortcut edges to obtain the represented subpaths in the original graph. We propose a recursive unpacking routine based on the fact that each shortcut edge represents a subpath, not necessarily in the original graph, consisting of exactly two edges, see Algorithm 1. As long as there is a shortcut edge in the path, we replace it by the two edges that originated the creation of this shortcut. Finally, we will have a path without any shortcut edges, that is equivalent to a path in the original graph, see Figure 22.

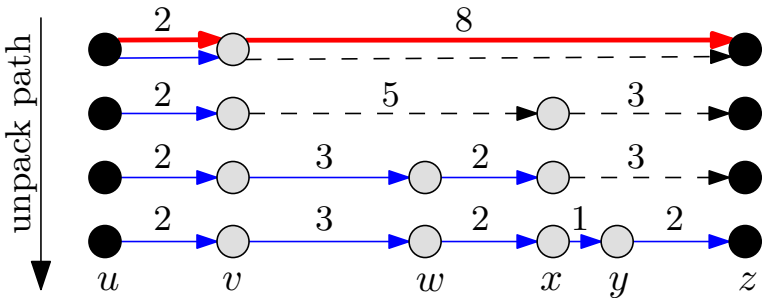


Figure 22: Unpacking of the shortest u-z-path of Figure 19. Directed graph, the numbers on the edges denote the edge weights. Normal arrows denote the shortest u-z-path in the original graph, dashed lines denote shortcuts and thick arrows the path found by the query algorithm.

Consider a shortcut (v, w) that represents a path $\langle v, u, w \rangle$. One possible implementation stores the middle node u in the data structure that represents the shortcut. If we do not store this additional information and scan through the incident edges of v and w to find (v, u)

and (u, w) because we want to save space, we face a dilemma. Remember the data structure representing the search graphs $G_\uparrow$ and $G_\downarrow$. Since $v, w > u$, the edges (v, u) and (u, w) are both stored in the edge group of u and not in the edge groups of v and w, as illustrated in Figure 23. So we either need to store the middle node u with every shortcut (v, w) or we need to store each edge, that is part of a shortcut, in the edge group of both incident nodes. The first variant is usually more space-efficient.

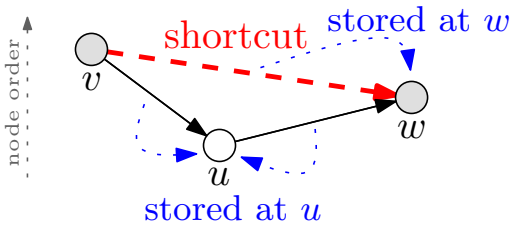


Figure 23: Efficient search graph data structure. Directed graph, normal arrows denote edges, the fat dashed arrow denotes a shortcut edge, the dotted arrows denote the affiliation to an edge array. Edges are only stored at the incident node with smaller order.

It is possible to implement the unpacking algorithm with time requirements linear in the number of edges of the unpacked path since each step increases the number of edges in the current path by exactly one edge. The location of the two originating edges of a shortcut can be implemented in constant time by storing them in the shortcut edge data structure.

3.5.2 Optimizations

Because our query algorithm does not relax all edges incident to a settled node, it is possible to settle nodes that are reached over a suboptimal path. Since we only want to find shortest paths, we can prune the search at such nodes. This technique is called *stall-on-demand*, see [34]. We will explain it only for forward search, but it can be applied to the backward search in the same way. We will start with an exhaustive example, see Figure 24. The edge (x, u) is not relaxed because $u < x$. So the node u will be settled with tentative distance 8 and path $\langle s, u\rangle$ that is suboptimal since path $\langle s, x, u\rangle$ has weight 4. It does not make any sense to relax the edges incident to node u because they will also result in suboptimal paths, we say we *stall* the node u with *stalling distance* 4. To stall the node v, that is settled before u and has at this time no neighbor that witnesses a suboptimal path, we need a *stalling process* to identify nodes that have been reached on a suboptimal path. Because we use search graphs, we cannot stall the node u while we settle node x because the edge (x, u) is only stored in the edge array of u. Therefore, our stalling process starts at node u, and stalls all nodes with suboptimal paths using a breadth first search. The BFS stops at nodes that are already stalled or that are not going to be stalled. We will only stall nodes with higher order than u since we use search graphs. In our example, the nodes v, w and z will be stalled. Node y will not be stalled because its tentative distance is currently 9 with path $\langle s, x, y\rangle$ and the stalling process compares it to the path $\langle s, x, u, v, w, y\rangle$ with weight 12. To ensure a correct query algorithm, we *unstall* nodes if they are reached via a shorter path than the previous one found by the modified Dijkstra algorithm. The distance responsible for stalling the node is not taken into account. In our example, after node u is settled with distance 8, its stalling process stalls node z because $w(\langle s, z\rangle) = 15 > 13 = w(\langle s, x, u, v, w, z\rangle)$. But by the time node y is settled with distance 9, the edge (y, z) is relaxed and node z is unstalled with tentative distance 11 and path $\langle s, x, y, z\rangle$.

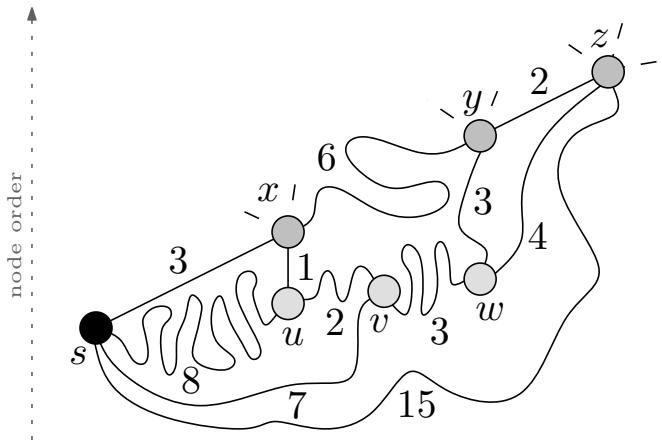


Figure 24: Motivation for stall-on-demand technique. Detail of an undirected graph, numbers denote edge weights.

Obviously, pruning the search at stalled nodes that have been demonstrably settled via a suboptimal path cannot invalidate the correctness proven in Theorem 2 since a continuation of the search from these nodes can never contribute to a shortest path.

The observing reader may note that our stall-on-demand technique can be further improved to stall even more nodes. But our tests show negative performance improvements and only a small increase of the number of stalled nodes. So we will share our experience with the purpose to make the mistake only once. To stall even nodes with lower order than u, we could use the parent pointers, managed by the modified Dijkstra query algorithm. We call this *deep stall-on-demand*. The stall-on-demand technique can speed up the query by factor 2 or more. But our experiments show that deep stall-on-demand consumes a little bit of this speedup. Nevertheless, for other algorithms like many-to-many [21], the number of settled and not stalled nodes is important and deep stall-on-demand pays off. Small further improvements of the number of stalled nodes are possible. We describe these improvements now but we do not use them in any of our experiments. So it is possible to unstall a node only if its new tentative distance is less than the stalling distance. But that leads to an increased query time because of the additional overhead. Also continuing the stalling process at already stalled nodes, if the new stalling distance is smaller that the old one, will likely increase the number of stalled nodes. To switch from BFS to a local Dijkstra search is another choice. This may result in lower stalling distances and more stalled nodes. A not so obvious improvement is to stall a node u even if the stalling distance over a node x, $d(x) + w(x, u) = d(u)$. However, to preserve a correct query algorithm, we only are allowed to stall nodes in the subsequent stalling process that are of lower order than x. We prove the correctness only for the forward search.

Lemma 4. *Let $(G = (V, E), <)$ be a contraction hierarchy. During the forward search, nodes $u \in V$ with tentative distance $d(u)$ get stalled, if there is a node $x \in V, x > v$ with $d(x) + w(x, u) \leq d(u)$. The subsequent stalling process stalls only nodes $w < x$. Then the query algorithm is still correct.*

Proof. Let r be the highest node in a shortest s-t-path. The query algorithm will fail if it does not reach such a node r because a node on a path to r is stalled. If $r \geq x$, then no node on a path from x to r would be stalled. And because of the properties of a CH, if u or another stalled node is on a shortest path to r, it can be replaced by nodes higher than x since there is a shortest path to those nodes via x. If $r < x$ and r is not reached because a node y on a path

to r got stalled because of x, there is a shortest s-y-path containing node x and subsequently a shortest s-r-path containing x, so there exists a shortest s-t-path containing $x > r$. This shortest path can still be found by the query algorithm, even if node r is not reached. □

Note that the above proof even holds if we perform a Dijkstra search instead of a BFS in the stalling process. The problem of the criteria of Lemma 4 is, that our data structure for the search graph does not directly stores the level or order of the nodes. So the decision, whether a node is lower than node x, cannot always be decided. Additional space is required to store the level of each node leading to decreased performance. Another possibility would be to use the level as node ID, but this leads to decreased performance because of increased cache misses.

3.6 Storing Witness Paths

To construct a CH out of a graph $G = (V, E)$, we need a node order $<$ and need to add necessary shortcuts to G. In the dynamic or time-dependent scenario, we perform the preprocessing to a given time (static graph), know all shortcuts and the node ordering but edge weights can change and we want to *revalidate shortcuts* and add additional necessary shortcuts. One method is to perform the contraction from scratch using the previous node order. Here, we present a method that can speedup the contraction: we store all shortcuts and for all pairs of edges $(v, u), (u, w) \in E$ with no added shortcut (v, w), see Algorithm 1, we store the witness path. Consider a pair of edges $(v, u), (u, w) \in E$ with $v, w > u$ and node u is currently being contracted, see Figure 25. In the best case, we have stored the information to determine the

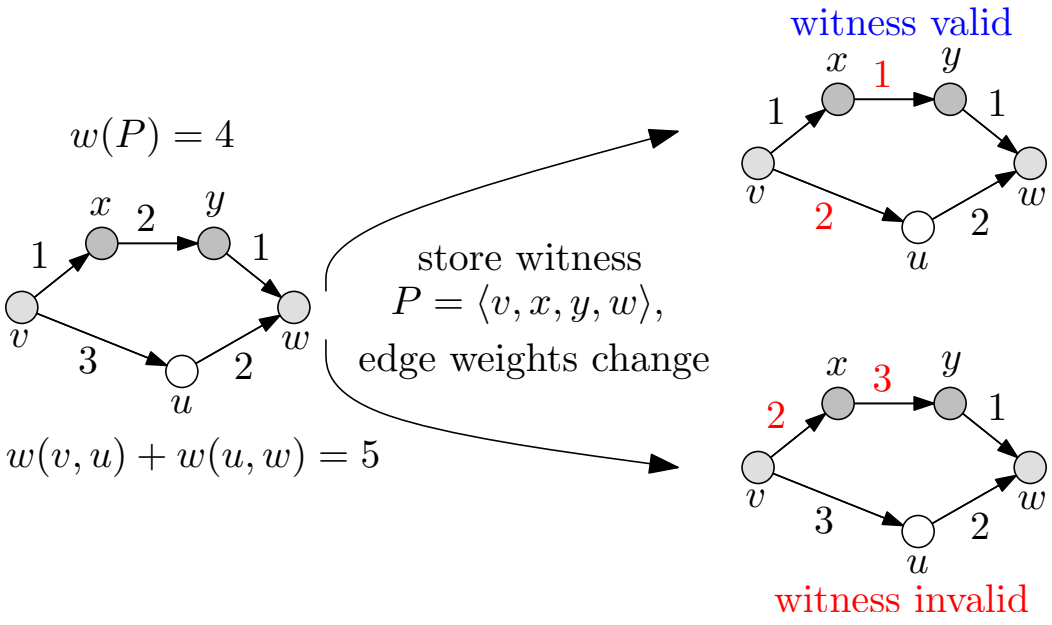


Figure 25: Using a stored witness path to determine necessary shortcuts during the contraction of node u. In this example, a witness P for the path $\langle v, u, w \rangle$ is found. Depending on the changed edge weights, the witness can stay valid or will be invalid.

necessity of a shortcut. If there previously was a shortcut (v, w), we can add it with updated weight, or we already know a possible witness path and can check if it is still not longer that $w(v, u) + w(u, w)$. If the shortcuts and witness paths are efficiently stored, this is considerably faster than a local search. Especially in the time-dependent scenario this may be useful since time-dependent Dijkstra algorithms are time consuming and witness paths of the static graph

may still be valid in the time-depended graph. However, not all witness paths will still be valid and additional shortcuts (v, w) are added without available shortcut and witness information for the contraction of the nodes v or w. For these nodes, a local search for witnesses is still necessary.

Some remarks:

- We hoped to limit the additional local searches by using only 1-hop and 2-hop searches, see Section 3.3.1. However, this proves not useable because the remaining graph will be to dense since the local searches are too limited. To limit the number of shortcuts, we need to remove the 2-hop limit but then the local searches are Dijkstra-based. So in the time-dependent scenario we still need an adapted Dijkstra algorithm and we do not further analyze the usage of stored witnesses and just shortly illuminate our experiences with it.

- On-the-fly edge reduction should not be used with witnesses. Because of changed edge weights, an edge that will be later used in an witness path, can be removed, rendering the witness unusable. The witness with the removed edge could still be valid since it does not need to be a shortest path but only be at most as long as the path containing the currently contracted node.

- Since preprocessing is already really fast with limited local searches and lazy updates, we observe that for an efficient implementation, the handling and storage of witness paths is crucial. Otherwise the additional overhead erodes the saved preprocessing time.

4 Applications

Since CHs are a hierarchical technique based on a bidirectional Dijkstra algorithm, there are many existing applications for them. In this section we will elaborate some of them in more detail and present results for them in Section 5.

4.1 Many-to-Many Routing

Instead of a point-to-point query, the goal of many-to-many routing is to find *all* distances between a given set S of source nodes and set T of target nodes. Two obvious solutions would be to perform $|S|$ Dijkstra queries without a target, thus solving the single source shortest path problem from each node in S, or perform $|S| \cdot |T|$ point-to-point queries. But for large $|S|, |T|$ this is impractical. A better approach [21] is to perform only $|S|$ forward searches and $|T|$ backward searches omitting the abort-on-success criterion and storing the results in a proper data structure. In more detail, we first complete backward searches from each node $t \in T$, storing t and the distance to t in a *bucket* at each reached node. Note, that it is only necessary to do this for non-stalled nodes since we are only interested in shortest paths. After that, we execute a forward search from each node $s \in S$, scan the buckets at each reached node. Let $d_s^{\uparrow}(v)$ and $d_t^{\downarrow}(v)$ denote the distance to v found by forward and backward search, respectively, if v is not reached, then the value is ∞. Since we omitted the abort-on-success criterion, $d(s,t) = \min\left\{d_s^{\uparrow}(v) + d_t^{\downarrow}(v) \mid v \in V\right\}$ because our query algorithm is correct (Theorem 2).

The authors of this many-to-many technique observed that the bucket scans during the forward searches are the most time-consuming operation. So CHs are a good improvement because of their small search space leading to fewer bucket entries. Another solution, presented by the authors, is an *asymmetric* modification to the original algorithm: They *prune* the backward search at the entrance to the highest level. And because the graph induced by the nodes in the highest level is an overlay graph, the algorithm is still correct. Because of the pruning, fewer nodes are reached in the backward search, causing fewer bucket entries and finally decreasing the cost for bucket scans. The analogon to the asymmetric approach for CHs is to contract not all nodes but leave some nodes in the topmost level, the *core*. The core size determines the degree of asymmetry and depends on the asymmetry of $|S|$ and $|T|$ for an efficient implementation. Intuitively, $|T|$ should be at least as large as $|S|$ and the core size should increase with increasing $|T| \,/\, |S|$.

4.2 Transit-Node Routing

Transit-node routing [3, 34] is currently the fastest static routing technique available. It is based on a simple observation intuitively used by humans: When you start from a source node s and drive to somewhere "far away", you will leave your current location via one of only a few "important" traffic junctions, called (forward) *access nodes* and arrive at your target via only a few backward access nodes. All access nodes together are called the *transit node set*. The main disadvantage is that it needs considerably more preprocessing time than other speedup techniques. The preprocessing for transit-node routing is essentially a generalization of many-to-many routing between the access node sets. But it would go beyond the scope of this diploma thesis if we would implement a complete preprocessing with CHs. Instead, we are using the node order of a CH to define a few sets of transit nodes and apply the algorithm provided by [34]. Using priority terms like the edge difference and the original edges term may

result in a hierarchy with fast preprocessing time and small sets of access nodes for each node in the graph.

4.3 Changing all Edge Weights

In CHs, we can distinguish between two main phases of preprocessing, node ordering and hierarchy construction. Similar to highway-node routing, we do not have to redo node ordering when the edge weights change – for example when we switch from driving times for a fast car to a slow truck. Hierarchy construction ensures correctness for *all* node orderings. Since the node ordering needs considerably more time than the hierarchy construction, it would be economical to perform the node ordering only once and the hierarchy construction for each speed profile separately. The intuition behind this is that most important nodes remain important even if the actual edge weights change – both sports cars and trucks are fastest on the motorway. In [34], HNR based on HHs demonstrated itself insensitive against changing all edge weights. Even switching from travel time metric to distance metric is feasible. Hopefully, CHs show the same behavior, because in combination with their small search spaces and preprocessing time they would give a good foundation for time-dependent routing.

4.4 Implementation on Mobile Devices

Due to its small memory overhead and search space, CHs are a good starting point for route planning on mobile devices. Currently there already exists an implementation [33] of a CH query algorithm that incorporates a compressed external-memory graph for further reduced space requirements. The saved space can be used for other useful applications on the mobile device to improve the added value.

4.5 Reuse of Shortcuts in other Applications

CHs provide not only a hierarchy of nodes like for many-to-many routing and TNR, but they also provide shortcut edges. These edges maintain the shortest paths distances and may enrich the original graph, even if we do not adopt the node ordering. Although the resulting graph has more edges than the original graph, it may be easier to process since e.g. the depth of a shortest paths search is reduced. Examples of possible applications are reach-based routing [11] or SHARC [4].

5 Experiments

This section reuses some the structure, methodology and formulations of [34]. Also the environment and the used test instances are the same.

5.1 Implementation

An exhaustive description of the implementation would go beyond the scope of this thesis so that we restrict ourselves to some important aspects. The program was written in C++. We started with the code provided by [34], especially the part for HNR, that already provides useful data structures for graphs and priority queues and a fast implementation of Dijkstra's algorithm with many modifications. We adapted these to the new needs of CHs, especially the support of a large number of levels. No libraries, except for the C++ Standard Template Library were used to implement the algorithms described in Sections 3.2, 3.3 and 3.5. Overall, our program consists of about 7 700 lines of code.

To obtain a robust implementation, we include extensive consistency checks in assertions and perform experiments that are checked against reference implementations, i.e., queries are checked against Dijkstra's algorithm.

For more details on the implementation, in particular on the employed data structures, we refer to Appendix A. A short documentation of the code is provided by Appendix B.

5.2 Experimental Setting

5.2.1 Environment

Experiments have been done on one core of a single AMD Opteron Processor 270 clocked at 2.0 GHz with 8 GB main memory and 2 × 1 MB L2 cache, running SuSE Linux 10.3 (kernel 2.6.22). The program was compiled by the GNU C++ compiler 4.2.1 using optimization level 3.

5.2.2 Instances

Main Instance. Most of our experiments have been done on a road network of Western Europe[2] (which we often just call "Europe") which has been made available for scientific use by the company PTV AG. For each edge, its length and its road category are provided. There are four major road categories (motorway, national road, regional road, urban street), which are divided into three subcategories each. In addition, there is one category for forest and gravel roads.

Additional Instances. In addition, we also performed some experiments on two other road networks. A publicly available version of the US road network (without Alaska and Hawaii) that was obtained from the TIGER/Line Files [36] (*USA (Tiger)*). However, in contrast to Europe, the TIGER graph is undirected, planarised and distinguishes only between four road categories, in fact 91% of all roads belong to the slowest category so that you cannot discriminate them. And very recently, a new version of the European road network (*New Europe*) that is lager than

[2]Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the UK

the old one and covers more countries[3] became available. It has been provided for scientific use by the company ORTEC.

Different Metrics. For most practical applications, a *travel time* metric is most useful, i.e., the edge weights correspond to an estimate of the travel time that is needed to traverse an edge. In order to compute the edge weights, we assign an average speed to each road category (see Table 2). In some cases, we also deal with a *distance* metric (where we directly use the provided lengths). For Europe, we used the distance metric on a subgraph, the largest strongly connected component (scc). Table 3 gives the sizes of the used road networks.

	motorway			national			regional			urban			
	fast		slow	fast		slow	fast		slow	fast		slow	
Europe	130	120	110	100	90	80	70	60	50	40	30	20	10
USA (Tiger)		100			80			60			40		

Table 2: Average Speeds [km/h]. The last column contains the average speed for "forest roads, pedestrian zones, private roads, gravel roads or other roads not suitable for general traffic".

road network	#nodes	#directed edges
Europe	18 029 721	42 199 587
→ Europe (scc)	18 010 173	42 188 664
→ Belgium	463 514	1 104 943
USA (Tiger)	23 947 347	57 708 624
→ South Carolina	463 652	1 091 530
New Europe	33 726 989	75 108 089

Table 3: Test Instances. In case of Europe, we give the size of both variants: the original one and the largest strongly connected component (scc). The symbol → denotes a subgraph relation.

5.2.3 Preliminary Remarks

Unless otherwise stated, the experimental results refer to the scenario where the road network of *Europe* with *travel time* metric is used, and only the shortest-path *length* is computed without outputting the actual route.

When we specify the memory consumption, we usually give the *overhead*, which accounts for the *additional* memory that is needed by our approach compared to a space-efficient *unidirectional* implementation of Dijkstra's algorithm. This overhead is always expressed in "bytes per node". Note that this differs from [34] where it is compared to an bidirectional implementation. For Europe, the bidirectional implementation requires about 1.0 B/node more than the unidirectional implementation.

[3] In addition to the old version, the Czech Republic, Finland, Hungary, Ireland, Poland, and Slovakia.

5.3 Methodology

5.3.1 Random Queries

As a simple, widely used and accepted performance measure, we run queries using source-target pairs that are picked *uniformly at random*. The advantage of this measure is that it can be expressed by a single figure (the average query time) and that it is independent of a particular application. In addition to the average query time, we also often give the average search space size and the average number of relaxed edges. Unless otherwise stated, in our experiments, we pick 100 000 random source-target pairs.

5.3.2 Local Queries

For use in applications it is unrealistic to assume a uniform distribution of queries in large graphs such as Europe or the USA. Therefore, we also measure *local queries* within the big graphs. We choose random sample points s and for each power of two $r = 2^k$, we use Dijkstra's algorithm to find the node t with Dijkstra rank $\text{rk}_s(t) = r$. The Dijkstra rank rk_s is the order in which the nodes got settled during the search starting at node s. We then use our algorithm to make an s-t-query. By plotting the resulting statistics for each value $r = 2^k$, we can see how the performance scales with a natural measure of difficulty of the query. We represent the distributions as a box-and-whiskers plot [29]: 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. Such plots are based on 1 000 random sample points s. For an example, see Figure 29.

5.3.3 Worst Case Upper Bounds

For any bidirectional approach where forward and backward search can be executed independently of each other, we can use the following technique to obtain a *per-instance worst-case guarantee*, i.e., an upper bound on the search space size for any possible point-to-point query for a given fixed graph G: By executing forward search from each node of G until the priority queue is empty, and no abort criterion is applied, and identical for the backward search, we obtain a distribution of the search space sizes of the forward and backward search, respectively. We can combine both distributions to generate an upper bound for the distribution of the search space sizes of bidirectional queries: when $\mathcal{F}_\uparrow(x)$ ($\mathcal{F}_\downarrow(x)$) denotes the number of source (target) nodes whose search space consists of x nodes in a forward (backward) search, we define

$$\mathcal{F}_\updownarrow(z) := \sum_{x+y=z} \mathcal{F}_\uparrow(x) \cdot \mathcal{F}_\downarrow(y), \tag{2}$$

i.e. $\mathcal{F}_\updownarrow(z)$ is the number of s-t-pairs such that the upper bound of the search space size of a query from s to t is z. In particular, we obtain the upper bound $\max\left\{z \mid \mathcal{F}_\updownarrow(z) > 0\right\}$ for the worst case without performing all n^2 possible queries. An examples can be found in Figure 30.

5.4 Contraction Hierarchies

5.4.1 Parameters

Despite the simplicity of the description of CHs, there are many parameters, i.e., the coefficients of the priority terms in the priority function for the node ordering (see Section 3.2) and the limits to the local searches for the contraction (see Section 3.3.1). We face an optimization problem given an objective function.

Objective Function. The most important objective is usually the average query time. Secondary, preprocessing time and space consumption are relevant. Since contraction hierarchies already have very low space consumption, we focused on two different variants:

- **aggressive variant:** only the average query time
- **economical variant:** the product of the average query time and the construction time

Both objective functions need to be *minimized*.

Coefficients of the Priority Function. There are currently eight priority terms in five categories. A special case is the coefficient 0 meaning that the corresponding priority term is ignored. So we need to choose the subset of priority terms we want to use and then need to optimize all of its coefficients. We use a simple but successful approach: *coordinate search* starts with a given set of coefficients and changes one or two of them, the other coefficients remain unchanged. We performed the coordinate search manually and will now describe our systematical proceeding in more detail. We start with a range, e.g. 1 – 1000 and step size 200 and perform the node ordering, contraction and queries for all of the values in the range. If the objective function has its minimum on the borders, we increase the range. If the minimum is within the range, we narrow the range to this minimum and decrease the step size, this is repeated until the improvements in the objective function are below some threshold. Then we advance to the next coefficient and repeat the same procedure. In our experience, each coefficient needs to be processed once or twice. Since node ordering and contraction in large graphs is time consuming and we only have limited resources, we performed the first coordinate search on a smaller subgraph, e.g. on the road network of Belgium instead of whole Europe. After that, we have a quite good set of coefficients for Europe and check whether we can improve it with small changes. In our experience the edge difference needs larger coefficients if the graph gets larger, all other priority terms are quite stable.

Local Search Limits. The aggressive variant has no search space limit during the node contraction. We only apply a limit on the settled nodes during the weight calculation, that corresponds to a simulated node contraction. For our main test instance, we always used a limit of 1 000 nodes, a smaller limit will result in a dense remaining graph and a larger limit only increases the preprocessing time without significant advantages for the average query time. But for other input graphs it may be necessary to change this limit.

For the economical variant, we chose a staged hop limit that changes if the average degree changes, see Section 3.3.1. We need to optimize the hop limits and the average degree limits triggering the changes. We started with the development of the average degree during node contraction with fixed hop limits, see Figure 26. We see that for hop limits below four, the

average degree eventually explodes. So we started with hop limits 1 – 5 and average degree limits from the graph if the curve of the next hop limit goes to far away. Then we tried to optimize the objective function by changing the average degree limits or by removing entire hop limits, i.e. switch directly from hop limit 3 to hop limit 5.

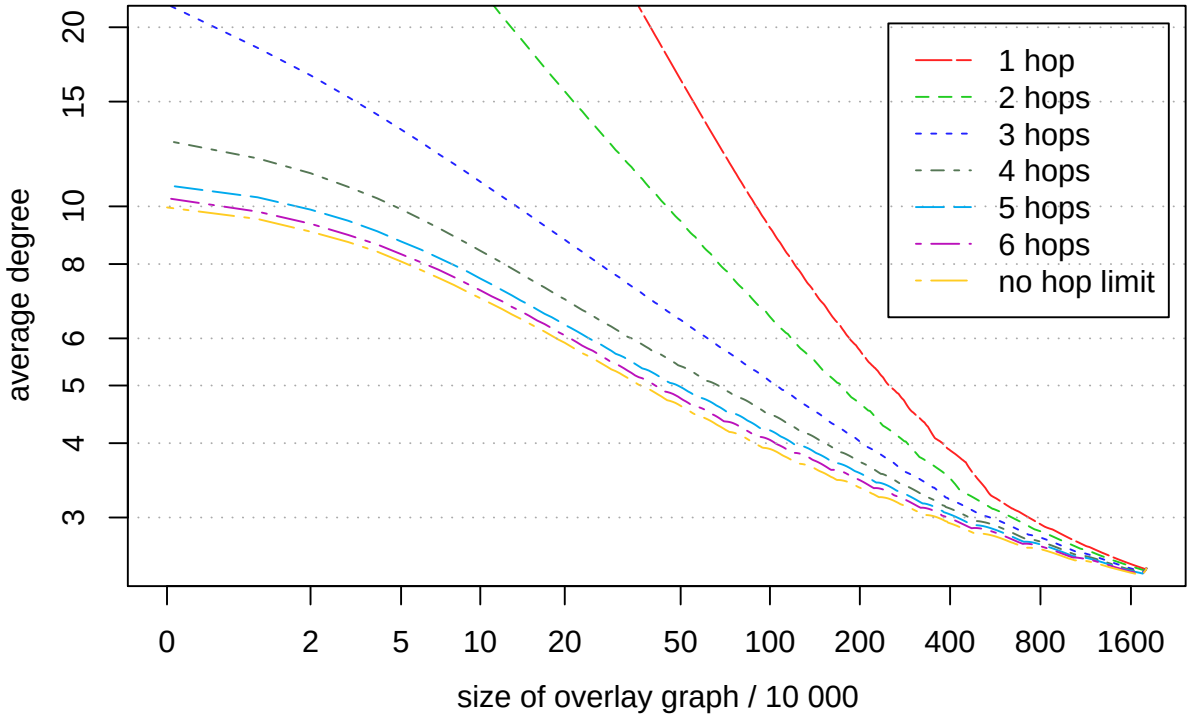


Figure 26: Average degree development for different hop limits.

Sensitivity of Tuning Parameters. We apply the parameters and search space limits found for Europe with travel time metric to the other graphs without changes. For two graphs we evaluated the sensitivity of the performance w.r.t. the most important parameters, see Figures 27, 28. The first subgraph is the road network of Belgium that is part of Europe and the second subgraph is the road network of South Carolina that is part of USA (Tiger). Both road networks are roughly of the same size and show comparable behavior, although they are from different continents and the road network of South Carolina is undirected and planarised where the one of Belgium is not. The main difference can be found looking at the vertical axis, the construction and query times for South Carolina are smaller. We will now have a closer look to each parameter shown in the analysis. The edge difference is the most important parameter. The hierarchy construction time increases if it has too small weight since the remaining graph will be to dense. But the difference is not that large because another priority term, the original edges term, partially takes the edge difference into account. If we would ignore the other priority term, the impact of the edge difference increases significantly. The search space coefficient shows noisy behavior but has only little impact for Belgium. The impact is larger for South Carolina, there is one outlier with query time at 49 μs, that is a decrease by 23%. Counting contracted neighbors is important to decrease the query time but it should not be weighted too large or otherwise the construction time will increase disproportionately high. Altogether contracted neighbors is a well behaving priority term. The original edges term is clearly weighted too little in both road networks, it could decrease construction and query time up to 12%. Switching from hop limit 1 to hop limit 2 at average degree 3.3 is a good choice, a larger limit does not speedup the construction significantly, since the parameter in general has only little effect. The average degree limit between hop limits 2, 3 and 5 needs to balance between construction and average query time, our current choice is in favor of the query time but values between 10 and 20 would work well, too. The query time increases with increasing coefficient since both limits between hop limit $2 \rightarrow 3$ and $3 \rightarrow 5$ are changed together having a larger part of the contraction being performed with hop limit 2. The parameters not shown have even less effect for wide range of values. Summarized, Belgium is less sensitive to parameter changes than South Carolina, this is not surprising since the parameters have been tuned for Europe. The query time for South Carolina can be improved by at least 23%, so a new parameter search is advisable if the current graph significantly differs from the graph used to obtain the parameter set, and peak performance is required. Otherwise our proposed parameter set performs quite well, see also Section 5.4.8 for additional tests on larger instances.

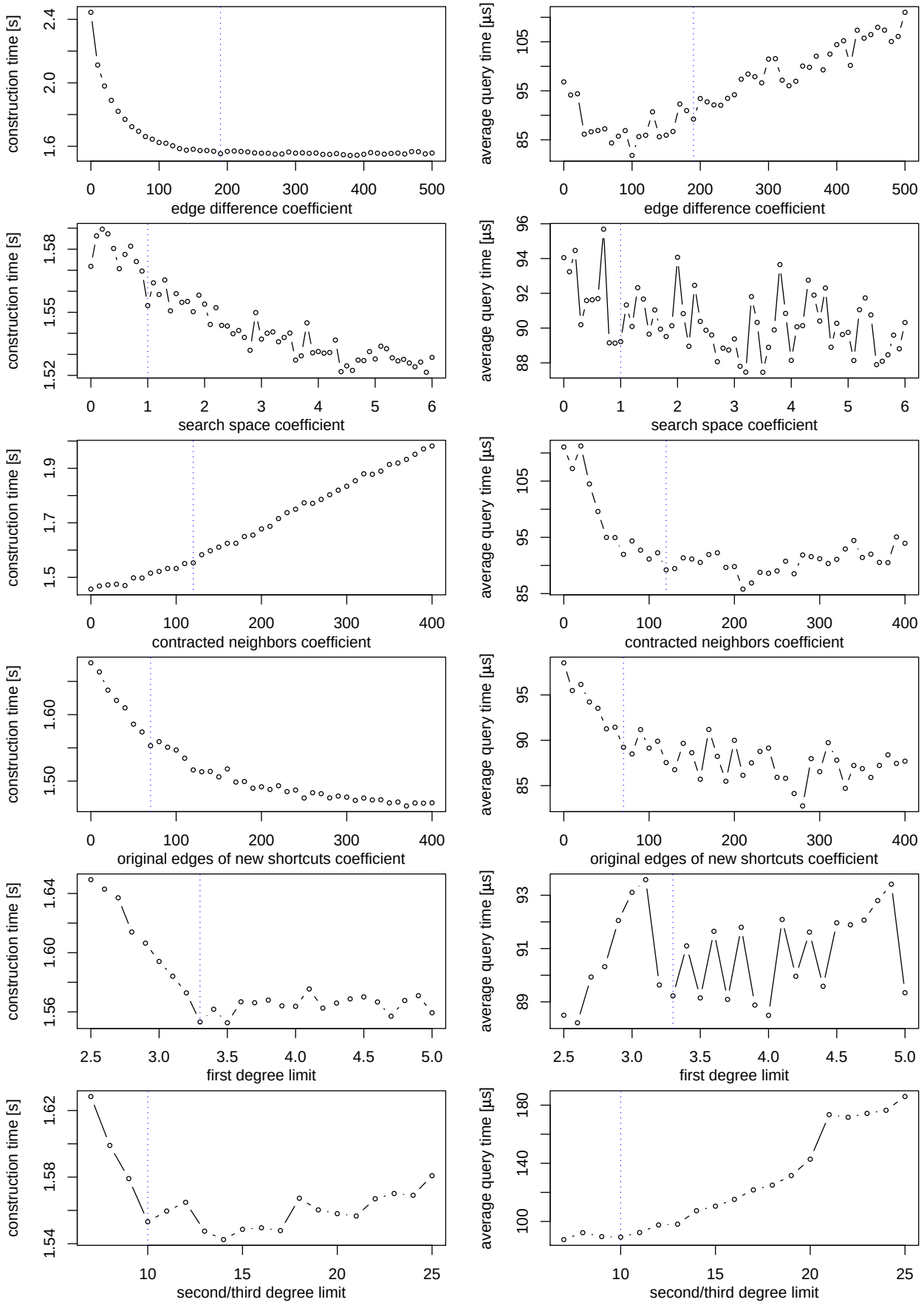


Figure 27: Sensitivity analysis with road network of Belgium. Parameters used: edge difference multiplier 190, search space multiplier 1, contracted neighbors multiplier 120, original edges term multiplier 70, hop@degree limits: 1@3.3, 2@10, 3@10, 5

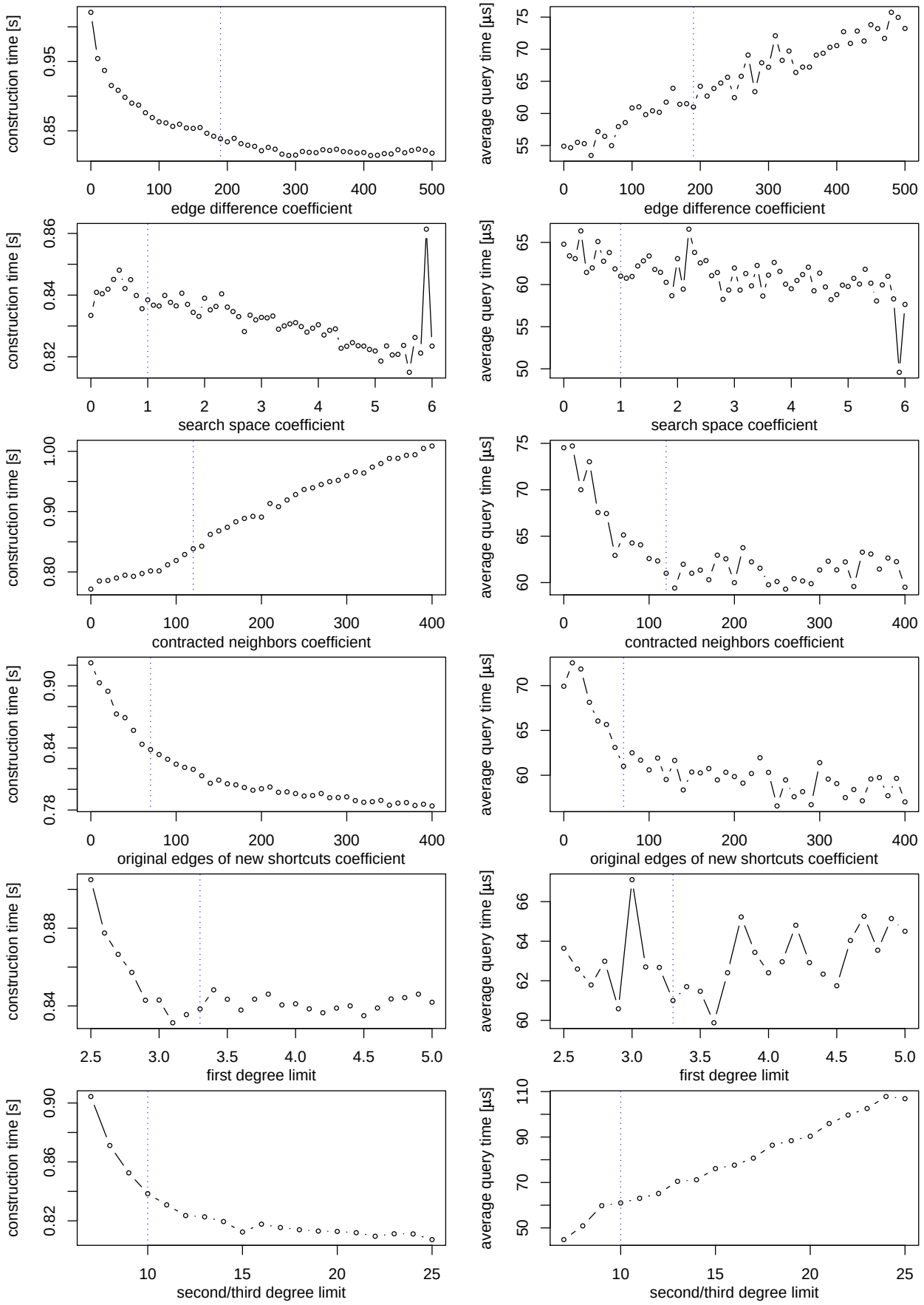


Figure 28: Sensitivity analysis with road network of South Carolina (USA). Parameters used: edge difference multiplier 190, search space multiplier 1, contracted neighbors multiplier 120, original edges term multiplier 70, hop@degree limits: 1@3.3, 2@10, 3@10, 5

5.4.2 Main Results

We start with evolving sets of priority terms and search space limits, together called a *method*[4], to get a deeper insight into them, see Table 4. We compare our results to the fastest variant of highway-node routing (HNR) from [35] using the same system environment. Note that this version of HNR outperforms all previous speedup techniques with comparable preprocessing time so that focusing on HNR is meaningful. As for CHs, we used search graphs for HNR, too.

Using the edge difference (letter E) as sole priority term yields a CH that already outperforms HNR in terms of query time. However, the preprocessing time is more than seven times larger. In the next method we also regard the cost of contraction as a priority term (letter S) resulting in more than two times better node ordering time and 14 times better hierarchy construction time. The imbalance between the improvement of those two parts is due to the additional local searches during the node ordering, especially the initialization of the priority queue takes more than 30 minutes. So we limit the local searches (letter L), improving the node ordering time by an additional factor four.

Adding the contracted neighbors counter (letter D) accelerates the query, the average is below 200 μs, more than four times faster than HNR. The algorithm in Line EDSL is a simple combination of contracted neighbors and improved preprocessing time, fast query times and *negative* space consumption for shortest path distance calculation. Using the original edges term (letter O) as uniformity term, we get better preprocessing times and space overhead but increased query time compared to the contracted neighbors term. We now split our search for the perfect method, the economical objective is to minimize the product of query time and hierarchy construction time, the aggressive objective is to minimize the product of query time and preprocessing times. To further increase the preprocessing time, we exchange the limit on the settled nodes (letter L) through an hop limit (digit 5) leading to a two times better node ordering time. Applying staged hop limits (digits 1235) shrinks the preprocessing time below 10 minutes and below HNR. The original edges term (letter O) can further improve preprocessing, query and space overhead. After having removing the contracted neighbors counter (Line EOS1235), we get our best algorithm regarding the economical objective and we will call it our *economical* variant. It is better than HNR with respect to all listed measures. We will now investigate the second, even more important objective: minimizing the query time. We first exchange the current uniformity term for Voronoi regions (letter V), and add a priority term to estimate the cost of queries (letter Q) to decrease the query time. This leads to an algorithm being five times faster than HNR, however we need to invest more time into preprocessing. The original edges term (letter O) as second uniformity term decreases the query time by an additionally 4% being our new best algorithm (Line EVOSQL) regarding the second objective. We will call it the *aggressive* variant and it will use it together with the economical variant for further experiments.

Using betweenness[5] approximations (letter W) can improve the query time by additional 3%.[6] But since betweenness approximation is time consuming, additional experiments would be necessary to investigate the usage of fast approximation results with CHs. And because the improvement is quite small, we merely present this under "possibilities" and do not further investigate global measures as priority terms.

[4] A method is in practice a set of command-line arguments for our program, see Appendix C.

[5] The execution times for betweenness approximation [10] are not included in Tab. 4.

[6] Preliminary experiments with reach-approximations were not successful.

Table 4: Performance of various node ordering heuristics. Terms of the priority function: E=edge difference, D=deleted/contracted neighbors, S=search space size, W=relative betweenness, V=$\sqrt{\text{Voronoi region size}}$, L=limit search space on weight calculation, Q=upper bound on edges in search paths, O=original edges term. Digits denote hop limits for testing shortcuts. The bottom line shows the performance for highway-node routing using the code from [35].

	method	node ordering [s]	hierarchy construction [s]	query [μs]	nodes settled	non-stalled nodes	edges relaxed	space overhead [B/node]
	E	13 010	1 739	670	1 791	1 127	4 999	-0.6
	ES	5 355	123	245	614	366	1 803	-2.5
	ESL	1 158	123	292	758	465	2 169	-2.5
	ED	7 746	1 062	183	403	236	1 454	-1.3
	EDL	2 071	576	187	418	243	1 483	-1.3
	EDSL	1 414	165	175	399	228	1 335	-1.6
	EO	5 979	758	250	617	395	2 119	**-3.1**
	EOL	1 274	319	245	604	383	2 119	**-3.1**
	EOSL	1 110	145	222	531	313	1 802	-3.0
ECONOMICAL	ED5	634	98	224	470	250	1 674	-0.6
ECONOMICAL	EDS5	652	99	213	462	256	1 651	-1.1
ECONOMICAL	EDS1235	545	57	223	459	234	1 638	1.6
ECONOMICAL	EDSQ1235	591	64	211	440	236	1 621	2.0
ECONOMICAL	EDOSQ1235	555	59	198	435	241	1 540	1.5
ECONOMICAL	EDOS1235	498	53	200	438	239	1 514	1.1
ECONOMICAL	EOS1235	**451**	**48**	214	487	275	1 684	0.6
AGGRESSIVE	EDSQL	1 648	199	173	385	220	1 378	-1.1
AGGRESSIVE	EVSQL	1 627	170	159	368	209	1 181	-1.7
AGGRESSIVE	EVOSQL	1 666	165	152	356	207	1 163	-2.1
AGGRESSIVE	EVOSQ5	821	96	174	395	217	1 297	-1.5
AGGRESSIVE	EDSQWL	1 629	199	163	372	218	1 293	-1.5
AGGRESSIVE	EVSQWL	1 734	180	154	359	208	1 159	-2.0
AGGRESSIVE	EVOSQWL	1 757	173	**147**	**350**	**205**	**1 127**	-2.3
	HNR	594	203	802[a]	981	630[b]	7 737	3.4

[a] Query time for the complete graph, for the search graph it would be most likely 5% better, but we do not have the value because of a compiler version change.

[b] for the complete graph

5.4.3 Local Queries

Figure 29 shows the query times according to the methodology introduced in Section 5.3.2. Local queries are more in step with actual practice and both variants of CHs are superior to HNR for *all* Dijkstra ranks. The lower median query time, the minima-maxima-distance and the quartile-distance are about three to four times smaller. HNR has a superlinear increase of the median on this plot with a logarithmic x axis. CHs show a near linear increase for Dijkstra ranks $r \leq 2^{23}$ and for $r = 2^{24}$ it even *decreases*. This is interesting as we do not use an all-pairs distance table for higher levels where we can observe a similar decrease [34].

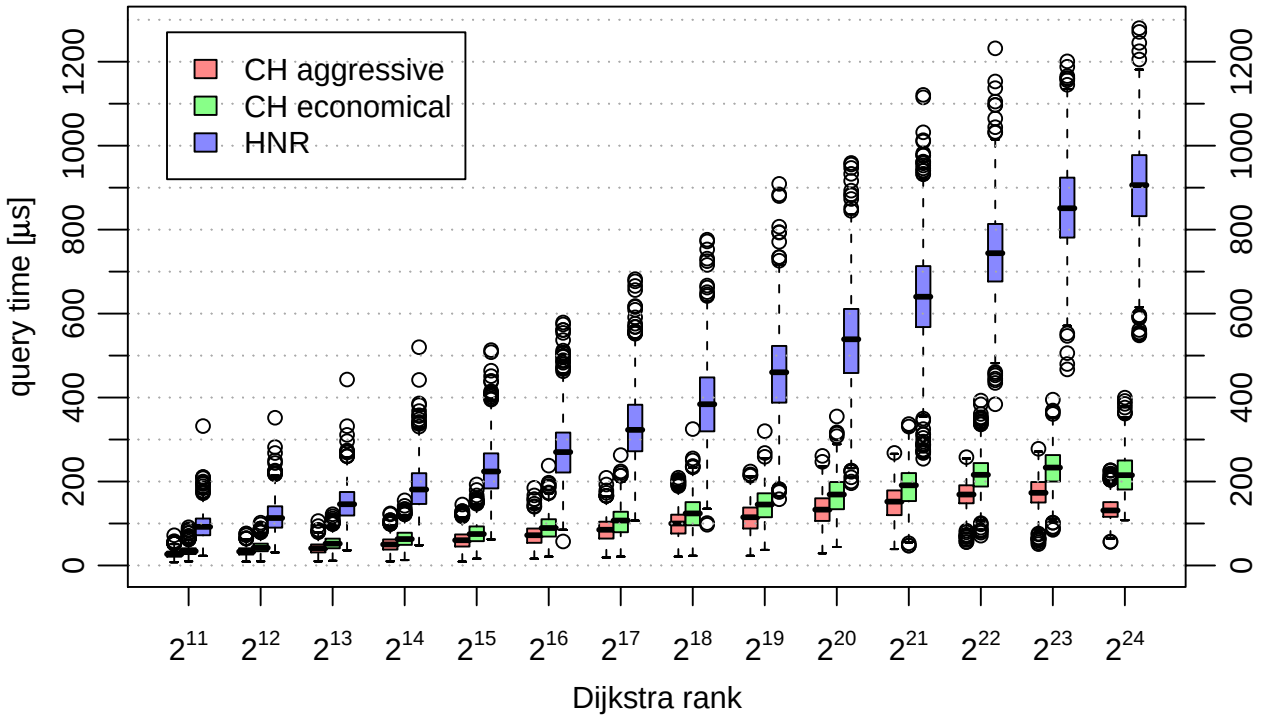


Figure 29: Local queries.

5.4.4 Worst Case Upper Bounds

To give a worst case guarantee for queries in the Europe road network, we give an upper bound on the settled nodes for *any* point-to-point search, see Section 5.3.3. For a point (x, y) on a line in Figure 30, the definition is that all but $y\%$ queries have at most x settled nodes in both forward and backward search space together. Again, we get clear improvements over HNR up to 60%. For the aggressive, variant we have an upper bound on the settled nodes that is only a factor 2.4 larger than the average number of settled nodes and 99.9% of all queries are below 575 settled nodes. Using linear extrapolation on the average query time and the average number of settled nodes, we obtain an upper bound on the query time of 362 μs. The number of relaxed edges has the upper bound of 3 169 edges leading to an extrapolated maximum query time of 414 μs.

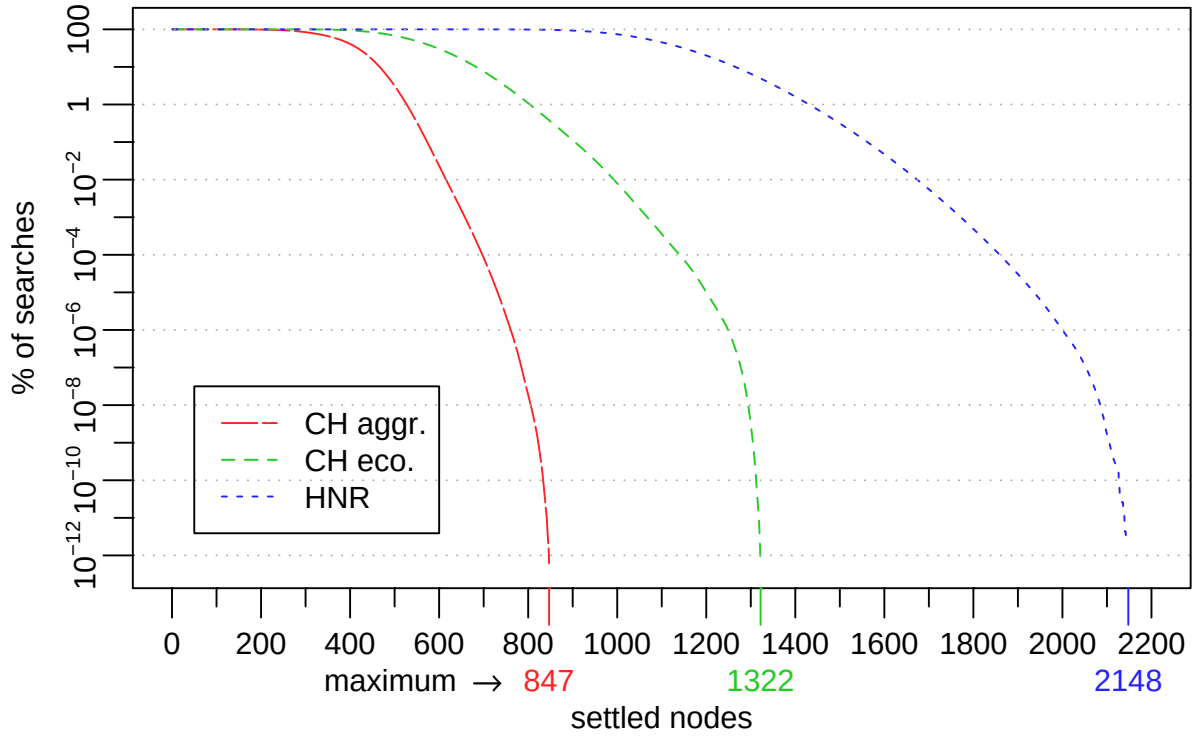


Figure 30: Upper bound on search space in settled nodes for the worst percentages of queries.

5.4.5 Outputting Complete Path Descriptions

So far, we have reported only the times and memory needed to compute the shortest-path length between two nodes. To determine a complete path description we now apply the algorithm described in Section 3.5.1. Since we need to store the middle node for each shortcut, the memory requirements for each edge increase, also leading to increased preprocessing and query times (Table 5). In more detail, for shortest path distance we have an edge data structure with 8 B/edge, for a complete path description our current implementation needs 12 B/edge. The comparatively large space overhead is owing to the fact that we even use 12 B/node for non-shortcut edges. If we would implement a more sophisticated version with only 8 B/edge for non-shortcut edges, we would achieve a space overhead of only 1.2 B/node for the aggressive variant. The hierarchy construction increases up to 13% and the query time by 12%. The path expansion of both variants is about the same as it mainly depends on the number of edges in the expanded path.

Among the path unpacking times we have seen, the path expansion times are only outperformed by the fastest variant for highway hierarchies in [31] that explicitly stores completely unpacked representations of the most important shortcuts. Note that this optimization works for any shortcut-based speedup technique including CHs.

	CH aggressive	CH economical
hierarchy construction [s]	176	54
query [μs]	170	238
expand path [μs]	323	321
space overhead [B/node]	6.2	10.3
edges in contracted path	20.9	23.4
edges in expanded path (avg.)	1 369.7	1 369.4

Table 5: Unpacking paths.

5.4.6 Changing all Edge Weights

In [34], HNR based on HHs demonstrated itself insensitive against changing all edge weights and even switching from travel time metric to distance metric. Here, we use the economical variant of CHs. The only modification is that for the contraction with the different speed profile, we changed the hop@degree-limits to 1@3.3, 2@10, 5@10.1, 7 otherwise the average node degree would eventually explode. Our CHs are somehow less adaptable to such changes, see Table 6. Our default speed profile is similar to the fast car profile and thus the hierarchy construction time increases only by 15%. The more the speed profile changes from the default profile, the more the hierarchy construction time increases. This fact strongly expresses itself in the construction time with the distance metric, it is increased by factor 35 and even 66% higher than the node ordering time. In every case, the query times are still faster than for HNR. Our explanation for the weak performance of CHs is that HNR with HHs partition the nodes only into a few, e.g. 12, levels and those partitions are less vulnerable to changes than the fine-grained CHs with a single node in each level. To get a node ordering that is suitable for both, travel time and distance metric, we could use a metric representing the mean value between the other two metrics. This yields an increase of the hierarchy construction time by factor 1.7 and a factor 2.1 by the query time for the travel time metric and factors 6 and 3.8 for the distance metric, respectively.

Table 6: Different speed profiles. The times in brackets refer to the case when node ordering was done with the same speed profile and the main times are for the case that node ordering was done for our default speed profile.

	default	fast car	slow car	slow truck	distance
hier. construction [s]	48	60 (52)	63 (53)	71 (55)	4 762 (137)
query [μs]	214	209 (215)	244 (250)	328 (306)	20 561 (2 382)

5.4.7 Stall-on-Demand

To show the performance of the stall-on-demand technique (Section 3.5.2), we present tests without stall-on-demand. In Table 7, the query performance of the aggressive and the economical variant are shown. The stall-on-demand technique yields improvements of the query time up to almost a factor 3 and thus is an important technique for fast queries in a CH. In comparison to HNR, CHs do not rely that much on stall-on-demand because the improvement factor is rather small and we even have better query times than HNR if stall-on-demand is not used. The author of [34] supplied preliminary experimental results for HNR, stating that stall-on-demand decreases the number of settled nodes by a factor 11, indicating that HNR should not be used without stall-on-demand. This is also interesting for a possible time-dependent algorithm based on CHs since time-dependent stall-on-demand is likely to become less efficient.

	CH aggressive	CH economical
query [μs]	313 (2.1)	612 (2.9)
settled nodes	904 (2.5)	1 632 (3.4)
nonstalled nodes	904 (4.4)	1 632 (5.9)
relaxed edges	4 909 (4.2)	9 885 (5.9)

Table 7: Query performance without stall-on-demand technique. The numbers in brackets denote the improvement factor of the stall-on-demand technique.

5.4.8 Other Inputs

We tested our aggressive and economical variant also on different road networks and metrics to examine the robustness of CHs. Table 8 shows the results. We did not repeat the parameter search (Section 5.4.1) for those graphs but used the values used for the Europe road network in Table 4. So we could expect additional improvements if we would repeat the parameter search. There are no real surprises, the economical variant has fast preprocessing times with reasonable query times whereas the aggressive shows fast query times. The USA (Tiger) graph shows slightly larger preprocessing times than the Europe graph but faster query times, those effects are already known from HNR. The New Europe graph is a larger network thus requiring more preprocessing time. The hierarchy construction time for the aggressive variant is more than a factor 2 larger than expected. Thats because the limit on the settled nodes for the local searches during the priority calculation is too restrictive and does not penalty searches enough which have e.g. more than 2 million settled nodes in an search that has no limit on the number of settled nodes. The additional time for contracting such nodes is not significant for node ordering, but significant for hierarchy construction. The construction time can be reduced if we increase the limit on the settled nodes, e.g. a factor 3 increase yields a construction time of 259 s, and even reduces the node ordering to 2 047 s. For the travel time metrics, we still have negative space overhead for shortest path distance calculation. We get a different picture for the distance metric where each edge represents the driving distance, but nothing unexpected. Because there are no real fast routes that could be preferred over other slower routes, it is less clear to identify important nodes and more shortcuts are necessary. Compared to HNR, our query time and space consumption is still very low. For Europe, [34] gives 2:04 minutes construction time and average query times of 9 230 μs, thus CHs preform queries more than four times faster with only 12 s more preprocessing time.

Table 8: Different graphs.

		TRAVEL TIME				DISTANCE			
		USA Tiger		New Europe		Europe		USA Tiger	
		aggr.	eco.	aggr.	eco.	aggr.	eco.	aggr.	eco.
	node ordering [s]	1 684	626	2 420	657	5 459	2 853	3 586	1 775
DISTANCE	hierarchy construction [s]	181	61	646	72	264	137	255	113
	query [μs]	96	180	213	303	1 940	2 276	645	1 857
	nodes settled	283	526	439	629	1 582	2 216	1 081	3 461
	non-stalled nodes	157	309	247	351	658	962	485	2 100
	edges relaxed	885	1 845	1 732	2 600	15 472	19 227	7 905	27 755
	space overhead [B/node]	-2.6	-1.3	-2.0	-0.3	0.6	1.5	-1.5	-0.9
PATH	hierarchy construction [s]	191	68	673	82	287	152	269	122
	query [μs]	107	198	243	345	2 206	2 615	721	2 121
	expand path [μs]	1 105	1 107	972	953	798	792	1 268	1 336
	space overhead [B/node]	5.8	7.8	5.6	8.5	10.2	11.7	7.4	8.3
	edges	21	26	21	24	21	29	22	40
	edges expanded	4 548	4 548	4 139	4 136	3 291	3 291	5 128	5 128

5.5 Many-to-Many Shortest Paths

For many-to-many routing, we do not use the aggressive variant but the method EVSQL from Table 4 because it shows slightly better performance, e.g. a 10 000×10 000 table is computed in 10.2 s instead of 11.0 s. We first compare CHs to HNR for symmetric instances in Table 9. CHs are more than two times faster than HNR, only for small instances the factor is not that large. Using the asymmetric technique described in Section 4.1 is successful for asymmetric instances with $|S| < |T|$, see Figure 31. We compared it to a naive implementation using an unidirectional Dijkstra algorithm on the original graph performing $|S|$ queries and CHs performing $|S| \cdot |T|$ queries. With increasing $|T| \,/\, |S|$, the core size should increase for best performance. For $|S| \geq 4$, the asymmetric many-to-many technique is superior. No asymmetry (core size 1) is required, if $|S| = |T|$, this slightly differs from the results for HNR since there, even for symmetric instances, asymmetry accelerates the calculation. We startet the core size at 2^{11} and multiplied it in every step by 4 until core size 2^{21}. Using a smaller core could not beat the symmetric variant on symmetric instances and using larger cores could not beat the Dijkstra algorithm for $|S| \leq 2$. Also, for $|T| \geq 2^{21}$ and core size below 2^{15}, the program provided by [34] could not store all bucket entries in the main memory, so we could not plot those values, but they would likely behave as expected. This is not an error in the program but merely a limitation of the main memory, in the original paper [21], they had $|S| \cdot |T|$ about five times smaller than ours.

Table 9: Computing $|S| \times |S|$ distance tables using CHs and HNR. The times for HNR are from [34] using an older compiler version that generates slightly slower code. All times are given in seconds.

$\lvert S\rvert$	100	500	1 000	5 000	10 000	20 000
CH	0.4	0.5	0.6	3.3	10.2	36.6
HNR	0.4	0.8	1.4	8.5	23.2	75.1

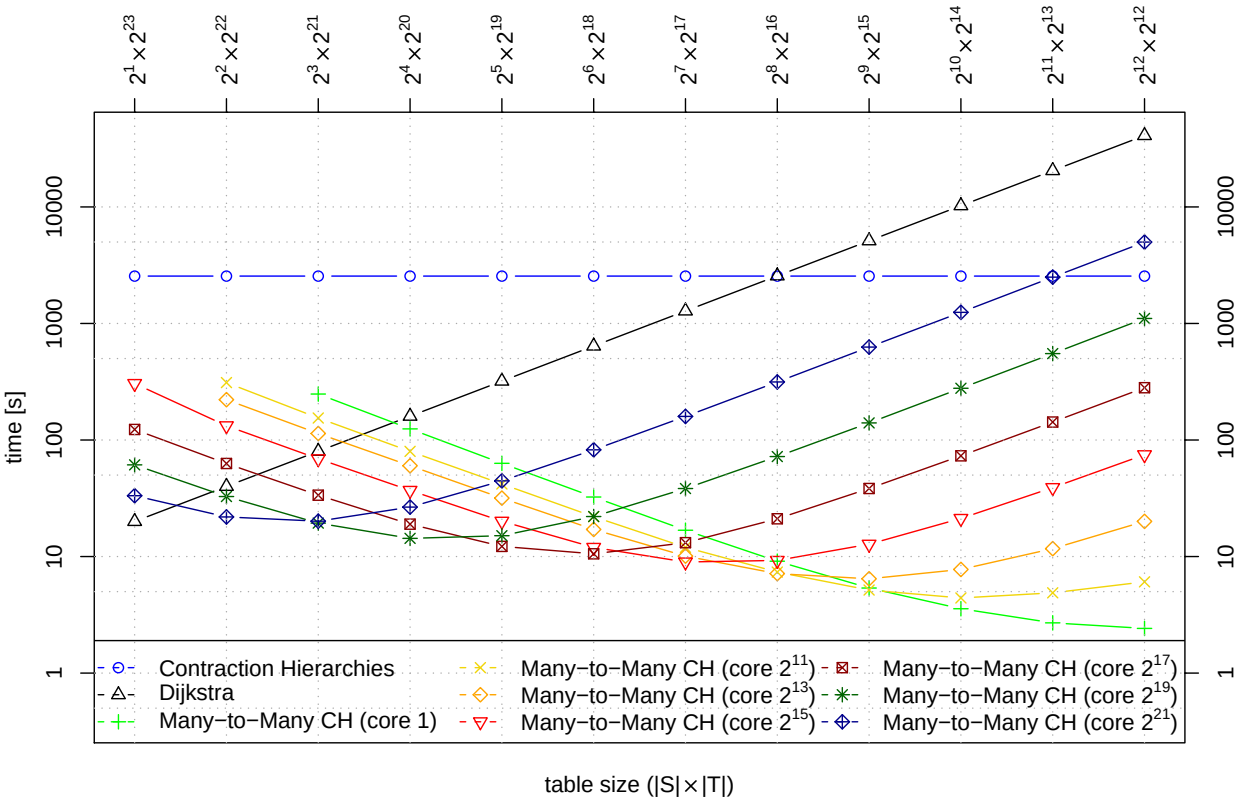


Figure 31: Comparison of algorithms and algorithm variants for asymmetric instances. We do not perform the CH and Dijkstra searches because they are too time consuming. Instead, the plottet times are the product of the average query time and the number of necessary queries.

5.6 Transit-Node Routing

To obtain a good hierarchy for TNR, we could not rely on the aggressive or economical variant because other properties of the hierarchy are important for TNR. We do not use a preprocessing solely based on CHs. Instead, the node ordering is used to determine the transit-node sets for the implementation from [34]. For this purpose we partitioned the nodes into five levels[7]. We compare the results with TNR based on HHs, the generous variant of [34] is used. The generous variant has the fastest query times.

During our experiments, we learned, that not the methods (Table 4) with the fastest query times are necessarily the best methods to use with TNR. The query times are very good for most methods, but the construction time of TNR varies significantly. In our experience, the upper bound on the length of search paths (letter Q) and the original edges term (letter O) are important. Also, a small hop limit can have a positive effect on the construction time of TNR. So we use the economical variant with additional letter Q for very good preprocessing time (Line EDOSQ1235) and the aggressive variant with a hop limit for best query time (Line EVOSQ5). Both variants are listed in Table 10 as separate columns. Even though we have no natural partition of the nodes like HHs, CHs with five levels are superior to HHs. We could improve the preprocessing time, including node ordering, by over 60%. The number of access nodes is reduced, which in turn results in 39% better query time and lower space consumption. Also, the locality filter works better with fewer not correctly answered queries in each level.

[7]Partition of the generous variant using HHs from [34]: level 4: 9 458 nodes, level 3: 48 008 nodes, level 2: 235 743, level 1: 2 446 093, level 0: 15 290 419 nodes.

Table 10: Comparison of HH and CH w.r.t. to TNR. Both hierarchies generate the same cardinalities of transit-node sets. The size of the level 2 distance table is given relative to the size of a complete table. The query statistics is w.r.t. 10 000 000 randomly chosen (s, t)-pairs. Each query is performed in a top-down fashion. For each level ℓ, we report the percentage of the queries that are not answered correctly in some level $\geq \ell$ and the percentage of the queries that are not stopped after level ℓ.

	CH economical	CH aggressive	HH
preprocessing time [h]	**0:46**	0:56	1:15
query time [μs]	3.3	**3.1**	4.3
overhead [B/node]	193	**187**	247
level 3 transit nodes	9 458	9 458	9 458
level 3 access nodes	9.9	9.8	11.3
level 2 transit nodes	293 209	293 209	293 209
level 2 distance table [%]	0.10	0.11	0.14
level 2 access nodes	4.0	3.6	4.4
level 3 wrong [%]	0.19	0.17	0.25
level 3 cont'd [%]	1.40	1.42	1.55
level 2 wrong [%]	0.0012	0.0012	0.0016
level 2 cont'd [%]	0.0141	0.0284	0.0180
level 1 wrong [%]	0.00017	0.00014	0.00019
level 1 cont'd [%]	0.0141	0.0284	0.0180

6 Discussion

6.1 Conclusion

CHs provide a new foundation for hierarchical routing methods in road networks. Its main advantages are its simplicity and its superior performance compared to HNR with HHs [35]. The combination of a priority queue heuristic for node sorting with extensible priority terms, and a specialized node contraction technique that contracts only one node at a time, convinces as sole routing algorithm with some improvements in preprocessing time and a factor five improvement in query time compared to HNR. The improvements to the space requirements are also noteworthy, we have negative space overhead for shortest path distance calculation. And CHs are currently the best basis for more sophisticated routing algorithms like TNR [3] and many-to-many routing [21]. Those algorithms are faster than CHs but crucially depend on another technique to provide a hierarchy. CHs fill this gap and outperform all previous existing techniques.

6.2 Future Work

For one part, CHs could not improve the node ordering time significantly compared to HHs. A lot of time of the node ordering is spent in local searches to identify witness paths. Many of those local searches are quite similar, think e.g. about the update of the neighbors of a contracted node. If we could store the search spaces and reuse them later, it has the potential to speedup the node ordering significantly. However, additional techniques, like partitioning of large graphs, seem to be necessary to cope with the large memory requirements.

In this thesis, we introduced some useful priority terms. But those are surely not the last word spoken about it. Depending on the application, other terms can be required to form a hierarchy with desired properties. Currently, a routing technique for the time-dependent scenario is developed, based on CHs, that already profits from an additional priority term. And it is not impossible that other priority terms, especially for uniformity, improve the performance of CHs themselves.

TNR currently uses a geometric locality filter. It might be a good idea to add a uniformity term to the priority function based on geometry to achieve further improvements. Additionally, the preprocessing for TNR can be completely based on CH to speedup the preprocessing. We will possibly see such developments in the near future.

Currently, CHs only support static road networks. Another improvement would be to implement dynamization techniques [35]. A first step toward this was changing the entire cost function, but updates for only a few changed edges should be processed faster than a complete hierarchy construction step. Especially for the mobile scenario edge weight changes need to be considered due to traffic jams.

References

[1] J. M. Anthonisse. The rush in a directed graph. Technical Report BN 9/71, Stichting Mathematisch Centrum, Amsterdam, 1971.

[2] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant shortest-path queries in road networks. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 46–59. SIAM, 2007.

[3] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[4] R. Bauer and D. Delling. SHARC: Fast and robust unidirectional routing. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2008.

[5] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. In *7th Workshop on Experimental Algorithms (WEA)*, 2008.

[6] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1962.

[7] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[8] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci*, 72(5):868–889, 2006.

[9] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40:35–41, 1977.

[10] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2008.

[11] A. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A^*: Efficient point-to-point shortest path algorithms. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 129–143, Miami, 2006.

[12] A. V. Goldberg and C. Harrelson. Computing the shortest path: A^* meets graph theory. Technical Report MSR-TR-2004-24, Microsoft Research, 2004.

[13] A. V. Goldberg and C. Harrelson. Computing the shortest path: A^* meets graph theory. In *16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.

[14] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Better landmarks within reach. In *9th DIMACS Implementation Challenge [?]*, 2006.

[15] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Better landmarks within reach. In *6th Workshop on Experimental Algorithms (WEA)*, volume 4525 of *LNCS*, pages 38–51. Springer, 2007.

[16] A. V. Goldberg and R. F. Werneck. Computing point-to-point shortest paths from external memory. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 26–40, 2005.

[17] R. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 100–111, 2004.

[18] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics*, 4(2):100–107, 1968.

[19] M. Hilger. Accelerating point-to-point shortest path computations in large scale networks. Diploma Thesis, Technische Universität Berlin, 2007.

[20] P. N. Klein. Multiple-source shortest paths in planar graphs. In *16th ACM-SIAM Symposium on Discrete Algorithms*, pages 146–155. SIAM, 2005.

[21] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner. Computing many-to-many shortest paths using highway hierarchies. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[22] E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of shortest path and constrained shortest path computation. In *4th International Workshop on Efficient and Experimental Algorithms (WEA)*, 2005.

[23] E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. In *9th DIMACS Implementation Challenge [?]*, 2006.

[24] U. Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, Institut für Geoinformatik, Münster, 2004.

[25] U. Lauther. An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags. In *9th DIMACS Implementation Challenge [?]*, 2006.

[26] J. Maue, P. Sanders, and D. Matijevic. Goal directed shortest path queries using Precomputed Cluster Distances. *ACM Journal of Experimental Algorithmics*, 2007. invited submission for special issue on WEA 2006.

[27] R. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up Dijkstra's algorithm. In *4th International Workshop on Efficient and Experimental Algorithms (WEA)*, pages 189–202, 2005.

[28] R. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up Dijkstra's algorithm. *ACM Journal of Experimental Algorithmics*, 11(Article 2.8):1–29, 2006.

[29] R Development Core Team. R: A Language and Environment for Statistical Computing. http://www.r-project.org, 2004.

[30] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *13th European Symposium on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 568–579. Springer, 2005.

[31] P. Sanders and D. Schultes. Engineering highway hierarchies. In *14th European Symposium on Algorithms (ESA)*, volume 4168 of *LNCS*, pages 804–816. Springer, 2006.

[32] P. Sanders and D. Schultes. Engineering fast route planning algorithms. In *6th Workshop on Experimental Algorithms (WEA)*, volume 4525 of *LNCS*, pages 23–36. Springer, 2007.

[33] P. Sanders, D. Schultes, and C. Vetter. Mobile Route Planning, 2008. in preparation, http://algo2.iti.uka.de/schultes/hwy/.

[34] D. Schultes. *Route Planning in Road Networks*. PhD thesis, Universität Karlsruhe (TH), 2008.

[35] D. Schultes and P. Sanders. Dynamic highway-node routing. In *6th Workshop on Experimental Algorithms (WEA)*, volume 4525 of *LNCS*, pages 66–79. Springer, 2007.

[36] U.S. Census Bureau, Washington, DC. UA Census 2000 TIGER/Line Files. http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html, 2002.

A Implementation

An exhaustive description of every single aspect of the implementation would go beyond the scope of this thesis. Thus, after some rather general statements in Section 5.1, here, we mainly focus on the graph data structures. The description is partly adopted from [34].

A.1 Graph Data Structures

We use two graph data structures, an updateable graph for the node ordering and hierarchy construction, and a search graph for queries. A previous implementation of [34] was available and modified for the needs of CHs, especially a large number of levels. But we will provide a self-contained description. Both implementations of these graphs share a common interface so that queries can also be performed on the updateable graph. We first describe the common details of both graphs and then elaborate the details.

The graph is always represented as *adjacency array*, which is a very space-efficient data structure that allows fast traversal of the graph. There are two arrays, one for the nodes and one for the edges. The edges (u, v) are grouped by the source node u and store only the ID of the target node v and the weight $w(u, v)$. Each node u stores the index of its first outgoing edge in the edge array. In order to allow a search in the backward graph, we have to store an edge (u, v) also as backward edge (v, u) in the edge group of node v. In order to distinguish between forward and backward edges, each edge has a forward and a backward flag. By this means, we can also store two-way edges $\{u, v\}$ (which make up the large majority of all edges in a real-world road network) in a space-efficient way: we keep only one copy of (u, v) and one copy of (v, u), in each case setting both direction flags.

The basic adjacency array has to be extended in order to incorporate necessary level information. For node ordering, hierarchy construction and query, we need to distinguish between edges (u, v) with $u > v$ and $u < v$. This defines a partition on the edge group of node u. We first store the edges (u, v) with $u > v$ and then the edges (u, v) with $u < v$. For fast access, a second index into the edge array to the first edge (u, v) with $u < v$ is stored, see Figure 32.

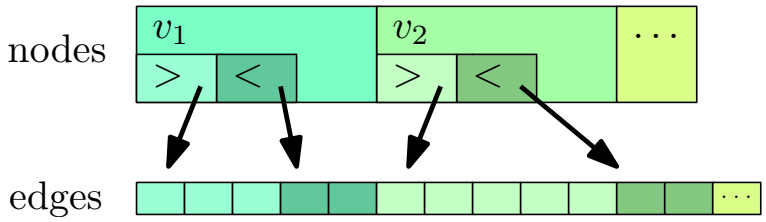


Figure 32: An adjacency array, each edge group is partitioned by level relative to the owner.

A.1.1 Updateable Graph

For node contraction (Section 3.3) we need to add shortcuts to the graph. And for the on-the-fly edge reduction (Section 3.3.2) we need to delete edges. So we need to allow the addition and deletion of edges at any time. Deletion is comparatively simple: we allow holes in the edge array. This is implemented using a third index into the edge array pointing to the end of the edge group. A deleted edge causes a hole that is filled by the last edge in the partition. If it was the first partition, we fill the hole with the last edge of the second partition and decrement the second edge index. In every case the third edge index is decreased. In order to allow efficient

additions as well, we ensure that if a node u has x edges, its edge group has a capacity of at least[8] $\min\{2^y \mid 2^y \geq x\}$, i.e., we reserve some space for additional edges. Now, adding an edge is straightforward provided that the capacity is not exceeded – we just have to move edges between the two partitions to make room at the right spot for a new edge. If, however, the capacity is exceeded, we copy the whole edge group to the end of the edge array (which is, in fact, a resizable STL vector) and double its capacity. Of course, the first edge index of u has to be updated accordingly. The second and third edge index is stored relative to the first edge index. Note that these memory management strategies employed by our flexible graph data structure are similar to those used by an STL vector.

A.1.2 Search Graph

The search graph does not store an edge $e = (u, v)$ in the edge group of both nodes. According to Section 3.5, we only need to provide the upward graph $G_\uparrow$ for forward search and $G_\downarrow$ for backward search. If $u < v$, then $e \in G_\uparrow$ and we store the edge in the edge group of u as forward edge. If $u > v$ then $e \in G_\downarrow$ and we store the backward edge (v, u) in the edge group of v. Note that for this backward edge holds $v < u$. So we only store edges (u, v) with $u < v$ and only one index in the edge array for each node is necessary because the other partition of the edge group is always empty.

A.2 Priority Queue

Specification. Manages a set of elements with associated totally ordered priorities and supports the following operations:

- *insert* – insert an element,
- *deleteMin* – retrieve the element with the smallest priority and remove it,
- *decreaseKey* – set the priority of an element that already belongs to the set to a new value that is less than the old value,
- *increaseKey* – set the priority of an element that already belongs to the set to a new value that is larger than the old value,

Used by all variants of Dijkstra's algorithm and the node ordering algorithm.

Implementation. We cannot use the priority queue implementation that the Standard Template Library provides since the *decreaseKey* operation is not supported. Therefore, we use the *binary heap* implementation provided by [34] extended by the *increaseKey* operation.

[8] The capacity can be even higher if edge deletions have taken place. This is due to the fact that the capacity is never reduced.

B Code Documentation

This is not nearly a complete and exhaustive code documentation. It should merely provide an entrance point to somebody that needs to use or extend our code. For a more detailed documentation we refer to the inline documentation of our code.

B.1 UML Class Diagrams

For the three main use cases: node order selection, hierarchy construction and query, we will present simplified UML class diagrams. Only the most important classes, methods, attributes and associations are included. They are only an abridged overview of the structure of our code but nevertheless useful to somebody who is not familiar with our code.

Node order selection. Figure 33. The class command::NodeOrder parses the command-line arguments. It imports the original graph as datastr::graph::UpdateableGraph object and prepares the parameters for the node ordering in a processing::ConstructCH::WeightCalculation object. Both objects are supplied to an instance of processing::ConstructCH that performs the node ordering and hierarchy construction calling createHierarchy(). The node contraction happens in processNode(). After that, the graph object is a CH and the node order can be exported using writeLevels().

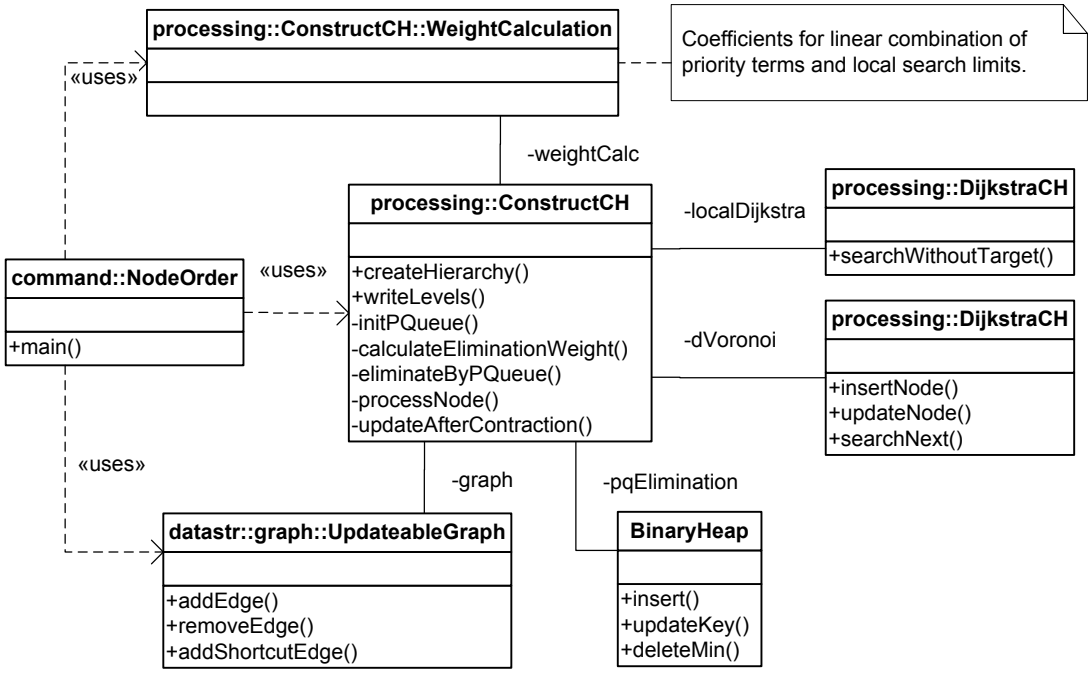


Figure 33: An UML class diagram for the use case: node order selection.

Hierarchy construction. Figure 34. Now the class command::Construct parses the command-line arguments. Instead of creating a node order from scratch, the parameters for node contraction are prepared as a processing::ConstructCH::ContractParameters object. It is supplied to an instance of processing::ConstructCH, along with the original graph. The node order is imported using readLevels(). A call to constructHierarchy() forms the supplied datastr::graph::UpdateableGraph object into a CH.

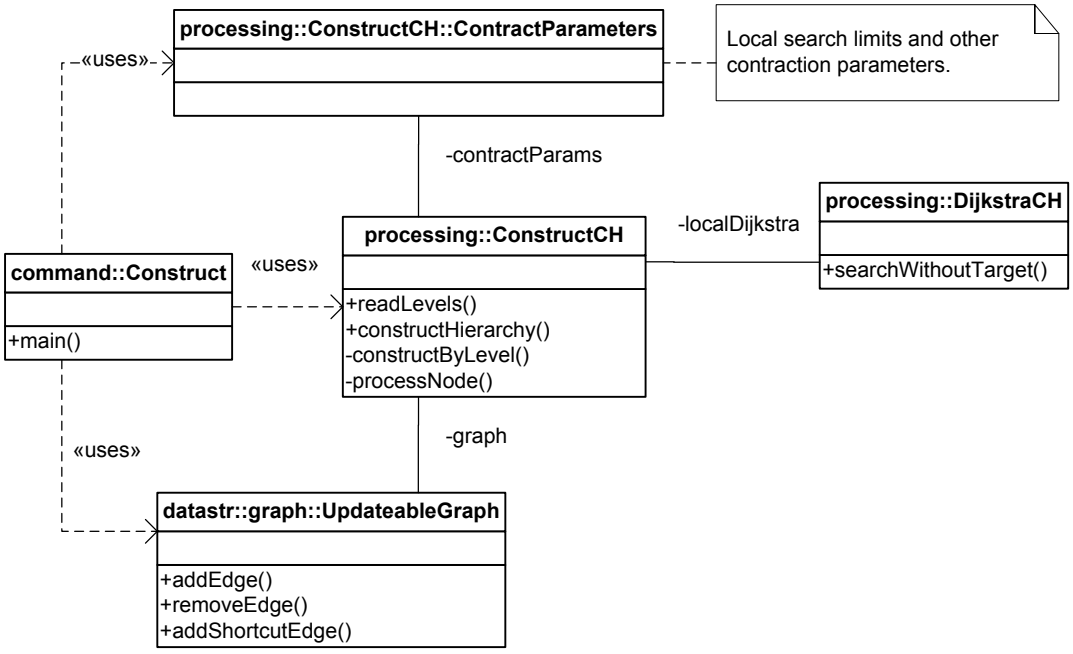


Figure 34: An UML class diagram for the use case: hierarchy construction.

Query. Figure 35. Before we can run a query, we need a CH. Currently two approaches are supported: construct a CH using a given node order or read the search graph from file. The command-line arguments are parsed in the class command::Construct. Then either a hierarchy construction is performed (see above) and the resulting graph is transformed to an datastr::graph::SearchGraph, or we deserialize() a previously serialized search graph. The search graph is supplied to a processing::DijkstraCH instance and followup point-to-point queries are performed using bidirSearch(). To expand a path to obtain a path in the original graph, call the method pathTo() after the successful query. We added more details regarding the class processing::DijkstraCH that are omitted in the other diagrams, since here, they do not reduce the clarity.

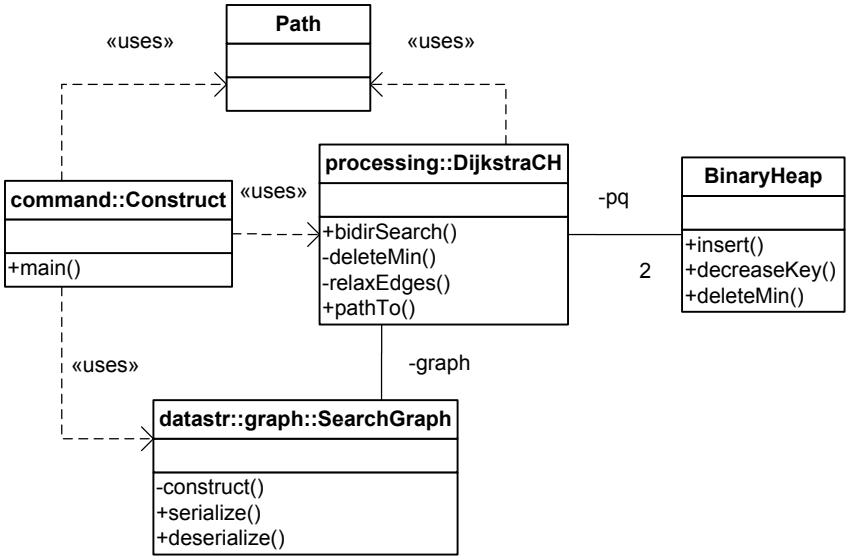


Figure 35: An UML class diagram for the use case: query.

C Command-Line Arguments

The command-line arguments for the `main`-program compiled out of the source code assigned to this thesis are listed in Table 11. Note however, that this is only the subset of command-line arguments required to specify the contraction. Additional parameters are necessary to specify the action and the input/output files that should be used. A short documentation of the command-line arguments can be found in the file `docu/index.html` of the source code.

Table 11: Command line arguments for the methods in Table 4

method	E	D	V	O	S	Q	W	L	12345	lazy upd.
	`-x`	`-w`	`-V`	`-e`	`-y`	`-S`	`-z`	`-n`	`-k`	`-p`
E	190									1000
ES	190				1					1000
ESL	190				1			1000		1000
ED	190	120								1000
EDL	190	120						1000		1000
EDSL	190	120			1			1000		1000
EO	190			600						1000
EOL	190			600				1000		1000
EOSL	190			600	1			1000		1000
ED5	190	120							5	1000
EDS5	190	120			1				5	1000
EDS1235	190	120			1				1,3.3,2,10,3,10,5	1000
EDSQ1235	190	120			1	145			1,3.3,2,10,3,10,5	1000
EDOS1235	190	120		70	1				1,3.3,2,10,3,10,5	1000
EDOSQ1235	190	120		70	1	145			1,3.3,2,10,3,10,5	1000
EOS1235	190			600	1				1,3.3,2,10,3,10,5	1000
EDSQL	190	120			1	145		1000		1000
EVSQL	190		60		1	145		1000		1000
EVOSQL	190		60	70	1	145		1000		1000
EVOSQ5	190		60	70	1	145			5	1000
EDSQWL	190	120	55		1	145	400	1000		1000
EVSQWL	190		55		1	145	400	1000		1000
EVOSQWL	190		55	70	1	145	400	1000		1000

Zusammenfassung

In dieser im Englischen verfassten Arbeit wird eine Routenplanungstechnik beschrieben, die allein auf dem Konzept der *Knotenkontraktion* aufbaut. Es wird jeder Knoten u einzeln nacheinander kontrahiert, d.h. aus dem Graph entfernt und *Abkürzungskanten* eingefügt um die Längen aller kürzesten Wege zu erhalten. Dazu werden alle eingehenden Kanten (v, u) und ausgehenden Kanten (u, w) betrachtet und falls der Pfad $\langle v, u, w\rangle$ der einzige kürzeste Weg zwischen v und w ist, wird eine Abkürzungskante (v, w) eingefügt mit Gewicht $w(v, w) := w(v, u) + w(u, w)$. Das Ergebnis ist eine Kontraktionshierarchie, im Englischen „*contraction hierarchy (CH)*“ die aus dem Originalgraph und allen Abkürzungskanten besteht. Durch die Reihenfolge der Knotenkontraktion wird auch eine Ordnung der Knoten nach „Wichtigkeit“ fixiert. Diese Ordnung wird in einem modifizierten, bidirektionalen Dijkstra-Algorithmus verwendet um schnell kürzeste Wege zu finden. Der Suchraum wird verkleinert, indem während der Vorwärtssuche nur Kanten relaxiert werden, die zu wichtigeren Knoten führen und während der Rückwärtssuche nur solche, die von wichtigeren Knoten kommen. Beide Suchräume treffen sich schließlich bei dem wichtigsten Knoten eines kürzesten Weges. Durch die Abkürzungskanten in der CH wird die Korrektheit des Suchalgorithmus gewährleistet. Da während der Suche nicht alle Kanten eines Knoten relaxiert werden, können Knoten auf suboptimalen Pfaden erreicht werden. Durch das Erkennen solcher Knoten und dem Abbruch der Suche an selbigen kann der Suchraum nochmals deutlich eingeschränkt werden. Der gefundene kürzeste Weg P kann Abkürzungskanten benutzen, so dass er nicht direkt ein Weg im Originalgraph sein muss. Um den Weg P auszupacken, also einen Weg P' im Originalgraph zu erhalten, wird eine rekursive Funktion verwendet. Jede Abkürzungskante (v, w) wurde während der Kontraktion eines Knotens u für einen Pfad $\langle v, u, w\rangle$ eingefügt, deswegen wird sie durch die beiden Kanten (v, u),(u, w) ersetzt. Wird der Knoten u mit der Abkürzungskante gespeichert, ist der Gesamtaufwand zum Auspacken von P linear in der Anzahl Kanten in P'.

Um die Knotenordnung zu erstellen, wird eine *einfache* und *erweiterbare* Heuristik verwendet. Sie benutzt im Kern eine Prioritätswarteschlange, deren Prioritätsfunktion jedem Knoten eine Linearkombination verschiedener Terme zuordnet. Einer der wichtigsten Terme ist die Kantendifferenz. Das ist die Differenz aus der Anzahl inzidenter Kanten des Knotens u und der Anzahl benötigter Abkürzungskanten für die Kontraktion von u. Dieser Term sorgt dafür, dass der Graph durch das Kontrahieren nicht zu dicht wird, also zu viele Kanten enthält, was die Suche verlangsamen würde. Ein weiterer Term fördert die uniforme Auswahl des nächsten zu kontrahierenden Knotens. Knoten in höheren Hierarchieebenen sollen möglichst gleichmäßig über den Graph verteilt sein um eine hohe Beschleunigung der Suche bei beliebig gewählter Punkt-zu-Punkt Anfrage zu erreichen. Weitere Terme sind in die Klassen Kontraktionskosten, Suchkosten und globale Maße eingeteilt. Abhängig von der Anwendung können verschiedene Prioritätsterme kombiniert werden um die benötigte Hierarchie zu erhalten. Die komplette Knotenordnung wird erstellt, indem jeweils der unwichtigste Knoten u aus der Prioritätswarteschlange entfernt und dann kontrahiert wird. Anschließend werden die Prioritätswerte der verbleibenden Knoten aktualisiert. Diese drei Schritte werden wiederholt, bis alle Knoten kontrahiert wurden. Eine Aktualisierung des Prioritätswertes ist notwendig, da sich beispielsweise die Kantendifferenz bei einigen Knoten durch das Einfügen von Abkürzungskanten oder das Entfernen der inzidenten Kanten von Knoten u ändern kann.

Im Vergleich zum bisher besten hierarchischen, Dijkstra-basierten Verfahren kann die Suchzeit nochmals um einen Faktor fünf verkleinert werden. Für ein großes Straßennetzwerk von Westeuropa ist die Obergrenze der betrachteten Knoten einer beliebigen Suche *kleiner* als die durchschnittliche Anzahl Knoten in einem kürzesten Weg. Und das geht bei einem *negativen*

Speicherüberhang, d.h. die Datenstruktur für die Distanzberechnung benötigt *weniger* Speicher als der Eingabegraph. Möglich wird das durch die Verwendung einer Suchgraph-Datenstruktur, in der jede Kante, auch wenn sie ungerichtet ist, nur einmal gespeichert werden muss.

CHen sind als Basis für viele andere Routingtechniken geeignet, um die Leistung und den Speicherplatzverbrauch zu verbessern. Beispiele hierfür sind die Distanztabellenberechnung, Transit-Node Routing, zielgerichtetes Routing oder mobile und dynamische Szenarien.