Engineering Highway Hierarchies

Peter Sanders and Dominik Schultes

Universität Karlsruhe (TH), 76128 Karlsruhe, Germany, {sanders,schultes}@ira.uka.de

Abstract. Highway hierarchies exploit hierarchical properties inherent in real-world road networks to allow fast and exact point-to-point shortest-path queries. A fast preprocessing routine iteratively performs two steps: first, it removes edges that only appear on shortest paths close to source or target; second, it identifies low-degree nodes and bypasses them by introducing shortcut edges. The resulting hierarchy of highway networks is then used in a Dijkstra-like bidirectional query algorithm to considerably reduce the search space size without losing exactness. The crucial fact is that 'far away' from source and target it is sufficient to consider only high-level edges.

Various experiments with real-world road networks confirm the performance of our approach. On a 2.0 GHz machine, preprocessing the network of Western Europe, which consists of about 18 million nodes, takes 13 minutes and yields 48 bytes of additional data per node. Then, random queries take 0.61 ms on average. If we are willing to accept slower query times (1.10 ms), the memory usage can be decreased to 17 bytes per node. We can guarantee that at most 0.014% of all nodes are visited during any query. Results for US road networks are similar.

Highway hierarchies can be combined with goal-directed search, they can be extended to answer many-to-many queries, and they are a crucial ingredient for other speedup techniques, namely for transit-node routing and highway-node routing.

1 Introduction

Computing fastest routes in road networks from a given source to a given target location is one of the showpieces of real-world applications of algorithmics. Many people frequently use this functionality when planning trips with their cars. There are also many applications like logistic planning or traffic simulation that need to solve a huge number of shortest-path queries. In principle we could use Dijkstra's algorithm [1]. But for large road networks this would be far too slow. Therefore, there is considerable interest in speedup techniques for route planning. Most approaches, including ours, assume that the road network is *static*, i.e., it does not change very often. Then, we can allow some preprocessing that generates auxiliary data that can be used to accelerate all subsequent queries. The preprocessing should be sufficiently fast to deal even with very large road networks, the auxiliary data should occupy only a moderate amount of space, and the queries should be as fast as possible.

1.1 Related Work

A detailed overview on shortest-path speedup techniques can be found in [2].

Bidirectional Search. A classical technique is *bidirectional search* which simultaneously searches forward from the source and backwards from the target until the search frontiers meet. Many more advanced speedup techniques use bidirectional search as an ingredient.

Goal Direction. Road networks allow effective goal-directed search using $\mathbf{A}^*$ *search* [3]: lower bounds define a vertex potential that directs search towards the target. This approach was recently shown to be very effective if lower bounds are computed using precomputed shortest-path distances to a carefully selected set of about 20 **L***andmark* nodes [4, 5] using the **T**riangle inequality (*ALT*).

The Precomputed Cluster Distances (PCD) technique [6] also uses precomputed distances for goal-directed search, yielding speedups comparable to ALT, but using less space.

The network is partitioned into clusters and the shortest connection between any pair of clusters is precomputed. Then, during a query, upper and lower bounds can be derived that can be used to prune the search.

Another goal-directed approach is to precompute for each edge 'signposts' that support the decision whether the target can possibly be reached on a shortest path via this edge. During a query only promising edges have to be considered. Various instantiations of this general idea have been presented [7–13]. While these methods exhibit good query performance, preprocessing times are quite large and so far no experimental results for the largest publicly available road networks have been published.

Separators. Perhaps the most well known property of road networks is that they are almost planar, i.e, techniques developed for planar graphs will often also work for road networks. Queries accurate within a factor $(1 + \epsilon)$ can be answered in near constant time using $O((n \log n)/\epsilon)$ space and preprocessing time [14]. Recently, this approach has been efficiently implemented and experimentally evaluated on a road network with one million nodes [15]. While the query times are very good (less than 20 μs for $\epsilon = 0.01$), the preprocessing time and space consumption are quite high (2.5 hours and 2 GB, respectively). Using $O(n \log^3 n)$ space and preprocessing time, query time $O(\sqrt{n} \log n)$ can be achieved [16] for directed planar graphs without negative cycles.

Another previous approach is the *separator-based multi-level method* [7, 17]. The idea is to use a set of nodes V_1 whose removal partitions the graph $G = G_0$ into small components. Now consider the *overlay graph* $G_1 = (V_1, E_1)$ where edges in E_1 are *shortcuts* corresponding to shortest paths in G that do not have inner nodes that belong to V_1. Routing can now be restricted to G_1 and the components containing s and t respectively. This process can be iterated yielding a multi-level method. A limitation of this approach is that the graphs at higher levels become much more dense than the input graphs, thus limiting the benefits gained from the hierarchy. Also, computing small separators can become quite costly for large graphs.

Reach-Based Routing / REAL. Let $R(v) := \max_{s,t \in V} R_{st}(v)$ denote the *reach* of node v where $R_{st}(v) := \min(d(s, v), d(v, t))$. Gutman [18] observed that a shortest-path search can be stopped at nodes with a reach too small to get to source or target from there. Goldberg et al. [19, 20] have considerably strengthened this approach by introducing various improvements, in particular a combination with ALT, yielding the REAL algorithm. Its query performance is similar to our highway hierarchies, while the preprocessing times are usually worse; a comparison can be found in Section 6.7.

Heuristics. In the last decades, commercial navigation systems were developed which had to handle ever more detailed descriptions of road networks on rather low-powered processors. Vendors resolved to heuristics still used today that do not give any performance guarantees: A^* search with estimates on the distance to the target rather than lower bounds or heuristic hierarchical approaches [21, 22].

1.2 Our Contributions

Our *exact* highway hierarchies (first published in [23, 24]) are inspired by *heuristic* hierarchical approaches. It is a bidirectional technique. While the search is inside some local area around source or target, all roads of the network are considered. Outside these areas, however, the search is restricted to 'important' roads. This general idea can be iterated and

applied to a hierarchy consisting of several levels. The crucial point is the definition of 'important streets'. In previous heuristic variants, this definition is based on a classification of the streets according to their type (motorway, national road, regional road, ...). Such a classification requires manual tuning of the data and a delicate trade-off between speed and suboptimality of the computed routes. In our exact variant, however, nodes and edges are classified fully automatically in a preprocessing step in such a way that all shortest paths are preserved. By this means, we win not only exactness, but also greater speed since we can build high-performance hierarchies consisting of many levels without worrying about the quality of the results.

In the preprocessing phase, we alternate between two procedures: edge reduction and node reduction. *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 neighbourhood radius $r(v)$ we are free to choose. An edge (u, v) is a highway edge if it belongs to some shortest path from a node s to a node t such that (u, v) is neither fully contained in the neighbourhood of s nor in the neighbourhood of t, i.e., $d(s, v) > r(s)$ and $d(u, t) > r(t)$. In all our experiments, neighbourhood radii are chosen such that each neighbourhood contains a certain number H of nodes. H is a tuning parameter that can be used to control the rate at which the network shrinks.

Node reduction (also called *contraction*) 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.

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. Highway hierarchies are the first speedup technique that was able handle the largest available road networks giving query times measured in milliseconds. There are two main reasons for this success: Under the above reduction routines, the road network shrinks in a geometric fashion from level to level and remains sparse and near planar, i.e., levels of the highway hierarchy are in some sense *self similar*. The other key property is that preprocessing can be done using limited local searches starting from each node. Preprocessing is also the most nontrivial aspect of highway hierarchies. In particular, long edges (e.g. long-distance ferry connections) make simple minded approaches far too slow. Instead we use fast heuristics that compute a superset of the set of highway edges.

Some further optimisations allow to drop the average query times below one millisecond on a 2.0 GHz machine—even for a road network with more than 30 million nodes. One of these optimisations is an all-pairs distance table that we precompute for the topmost level L so that forward and backward search can be stopped as soon as all entrance points to level L have been found. Then, the remaining gap can be bridged by performing a moderate number of simple table lookups.

We cannot give a general worst-case bound better than Dijkstra's. So far, this drawback applies to all other exact speedup techniques, where an implementation is available, as well. However, in contrast to most of them, we can provide *per-instance worst-case guarantees*, i.e., for a given graph, we can determine an upper bound for the search space size of *any* possible point-to-point query performing only a linear number of unidirectional highway queries.

1.3 Subsequent Work

Various other speedup techniques were inspired by our highway hierarchies or even use them as their starting point. Goldberg et al. adopted the introduction of shortcuts in order to im-

prove both preprocessing and query times of the REAL algorithm. There is a many-to-many variant [25] and a combination with ALT [26]. Furthermore, the fastest implementation of transit-node routing [27, 28] allowing query times of a few microseconds is also based on our highway hierarchies. The same applies to highway-node routing [29], a very recent approach that can be used to handle dynamic scenarios, traffic jams for example. An alternative, heuristic approach to dealing with dynamic scenarios, which is based on highway hierarchies as well, has been developed by Nannicini et al. [30].

1.4 Outline

After beginning with some preliminaries in Section 2, we formally define the *highway hierarchy* of a given graph in Section 3. Then, Section 4 deals with both procedures of the preprocessing phase, the edge reduction (i.e., the *construction* of a highway network) and the node reduction (i.e., the *contraction* of a highway network). The basic query algorithm is introduced in Section 5. Furthermore, several optimisations are presented and some advanced topics, like outputting complete path descriptions and dealing with turning restrictions, are discussed. In Section 6, we present a wide range of experimental results, dealing with various real-world road networks, parameter settings, and scenarios of application. We do not only give average query times, but also a detailed analysis of queries with different degrees of difficulty, per-instance worst-case upper bounds, and comparisons to other speedup techniques.

2 Preliminaries

Graphs and Paths. We expect a *directed* graph $G = (V, E)$ with n nodes and m edges (u, v) with *nonnegative* weights $w(u, v)$ as input. The *length* $w(P)$ of a path P is the sum of the weights of the edges that belong to P. $P^* = \langle s, \ldots, t\rangle$ is a *shortest path* if there is no path P' from s to t such that $w(P') < w(P^*)$. The *distance* $d(s, t)$ between s and t is the length of a shortest path from s to t or ∞ if there is no path from s to t. If $P = \langle s, \ldots, s', u_1, u_2, \ldots, u_k, t', \ldots, t\rangle$ is a path from s to t, then $P|_{s'\to t'} = \langle s', u_1, u_2, \ldots, u_k, t'\rangle$ denotes the *subpath* of P from s' to t'. We use $u \prec_P v$ to denote that a node u precedes[1] a node v on a path $P = \langle \ldots, u, \ldots, v, \ldots\rangle$; we just write $u \prec v$ if the path P that is referred to is clear from the context.

Dijkstra's Algorithm. Dijkstra's algorithm [1] can be used to solve the *single-source shortest-path (SSSP) problem*, i.e., to compute the shortest paths from a single source node s to all other nodes in a given graph. It is covered by virtually any textbook on algorithms, e.g. [31, 32], so that we confine ourselves to introducing our terminology: Starting with the source node s as root, Dijkstra's algorithm grows a *shortest-path tree* that contains shortest paths from s to all other nodes. During this process, each node of the graph is *unreached*, *reached*, or *settled*. A node that already belongs to the tree is *settled*. If a node u is settled, a shortest path P^* from s to u has been found and the distance $d(s, u) = w(P^*)$ is known. A node that is adjacent to a settled node is *reached*. Note that a settled node is also reached. If a node u is reached, a path P from s to u, which might not be the shortest one, has been found and a *tentative distance* $\delta(u) = w(P)$ is known. A node u that is not reached is *unreached*; for such a node, we have $\delta(u) = \infty$.

In case that the shortest paths in a graph are not unique, Dijkstra's algorithm can be easily modified to determine *all* shortest paths between s and any node $u \in V$. This means that not a shortest-path tree is grown, but a shortest-path *directed acyclic graph* (DAG).

[1] This does *not* necessarily mean that u is the *direct* predecessor of v.

A *bidirectional* version of Dijkstra's algorithm can be used to find a shortest path from a given node s to a given node t. Two Dijkstra searches are executed in parallel: one searches from the source node s in the original graph $G = (V, E)$, also called *forward graph* and denoted as $\overrightarrow{G} = (V, \overrightarrow{E})$; another searches from the target node t backwards, i.e., it searches in the *reverse graph* $\overleftarrow{G} = (V, \overleftarrow{E})$, $\overleftarrow{E} := \{(v, u) \mid (u, v) \in E\}$. The reverse graph $\overleftarrow{G}$ is also called *backward graph*. When both search scopes meet, a shortest path from s to t has been found.

3 Highway Hierarchy

A *highway hierarchy* of a graph G consists of several levels $G_0, G_1, G_2, \ldots, G_L$, where the number of levels $L + 1$ is given. We will provide an inductive definition of the levels:

- Base case (G'_0, G_0): level 0 ($G_0 = (V_0, E_0)$) corresponds to the original graph G; furthermore, we define $G'_0 := G_0$.
- First step ($G'_\ell \to G_{\ell+1}, 0 \le \ell < L$): for given *neighbourhood radii*, we will define the *highway network* $G_{\ell+1}$ of a graph G'_ℓ.
- Second step ($G_\ell \to G'_\ell, 1 \le \ell \le L$): for a given set $B_\ell \subseteq V_\ell$ of *bypassable* nodes, we will define the *core* G'_ℓ of level ℓ.

*First step (*highway network*).* For each node u, we choose nonnegative *neighbourhood radii* $r^{\rightarrow}_\ell(u)$ and $r^{\leftarrow}_\ell(u)$ for the forward and backward graph, respectively. To avoid some case distinctions, we set $r^{\rightarrow}_\ell(u)$ and $r^{\leftarrow}_\ell(u)$ to infinity for $u \notin V'_\ell$ (Radius Property R1) and for $\ell = L$ (R2). In all other cases, neighbourhood radii have to be $\neq \infty$ (R3).

The level-ℓ *neighbourhood* of a node $u \in V'_\ell$ is $\mathcal{N}^{\rightarrow}_\ell(u) := \{v \in V'_\ell \mid d_\ell(u, v) \le r^{\rightarrow}_\ell(u)\}$ with respect to the forward graph and, analogously, $\mathcal{N}^{\leftarrow}_\ell(u) := \{v \in V'_\ell \mid d^{\leftarrow}_\ell(u, v) \le r^{\leftarrow}_\ell(u)\}$ with respect to the backward graph, where $d_\ell(u, v)$ denotes the distance from u to v in the forward graph G_ℓ and $d^{\leftarrow}_\ell(u, v) := d_\ell(v, u)$ in the backward graph $\overleftarrow{G_\ell}$.

The *highway network* $G_{\ell+1} = (V_{\ell+1}, E_{\ell+1})$ of a graph G'_ℓ is defined by the set $E_{\ell+1}$ of *highway edges*: an edge $(u, v) \in E'_\ell$ belongs to $E_{\ell+1}$ iff there are nodes $s, t \in V'_\ell$ such that the edge (u, v) appears in some shortest path $\langle s, \ldots, u, v, \ldots, t\rangle$ from s to t in G'_ℓ with the property that $v \notin \mathcal{N}^{\rightarrow}_\ell(s)$ and $u \notin \mathcal{N}^{\leftarrow}_\ell(t)$. Figure 1 gives an example. The set $V_{\ell+1}$ is the maximal subset of V'_ℓ such that $G_{\ell+1}$ contains no isolated nodes.

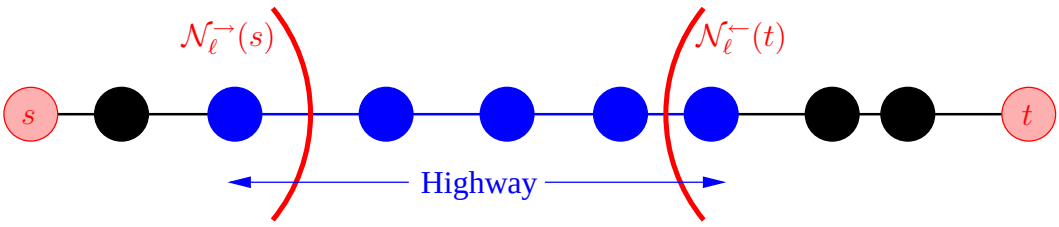

Fig. 1. A shortest path from a node s to a node t. Edges that leave the neighbourhood of s or t and edges that are completely outside the neighbourhoods of s and t are highway edges.

*Second step (*core*).* For a given set $B_\ell \subseteq V_\ell$ of *bypassable* nodes, we define the set S_ℓ of *shortcut edges* that bypass the nodes in B_ℓ: for each path $P = \langle u, b_1, b_2, \ldots, b_k, v\rangle$ with $u, v \in V_\ell \setminus B_\ell$ and $b_i \in B_\ell, 1 \le i \le k$, the set S_ℓ contains an edge (u, v) with $w(u, v) = w(P)$. The *core* $G'_\ell = (V'_\ell, E'_\ell)$ of level ℓ is defined in the following way:

$V'_\ell := V_\ell \setminus B_\ell$ and $E'_\ell := (E_\ell \cap (V'_\ell \times V'_\ell)) \cup S_\ell$. This definition is illustrated in Figure 2. Removing all core nodes from G_ℓ yields connected *components of bypassed nodes*.

The *level* $\ell(e)$ *of an edge* e is $\max\{\ell \mid e \in E_\ell \cup S_\ell\}$. For an edge (u, v), we usually write just $\ell(u, v)$ instead of $\ell((u, v))$. The highway hierarchy can be interpreted as a single graph $\mathcal{G} := (V, E \cup \bigcup_{i=1}^{L} S_i)$ where each node and each edge has additional information on its membership in the various sets $V_\ell, V'_\ell, B_\ell, E_\ell, E'_\ell, S_\ell$.

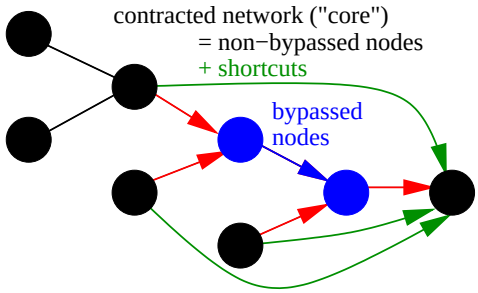

Fig. 2. The core of a highway network consists of the subgraph induced by the set of non-bypassed nodes and additional shortcut edges.

4 Construction

4.1 Computing the Highway Network

Neighbourhood Radii. Let us fix any rule that decides which element Dijkstra's algorithm removes from the priority queue in the case that there is more than one queued element with the smallest key. Then, during a Dijkstra search from a given node u, all nodes are settled in a fixed order. The *Dijkstra rank* $\text{rk}_u(v)$ of a node v is the rank of v w.r.t. this order. u has Dijkstra rank $\text{rk}_u(u) = 0$, the closest neighbour v_1 of u has Dijkstra rank $\text{rk}_u(v_1) = 1$, and so on.

We suggest the following strategy to set the neighbourhood radii. For this paragraph, we interpret the graph G'_ℓ as an undirected graph, i.e., a directed edge (u, v) is interpreted as an undirected edge $\{u, v\}$ even if the edge (v, u) does not exist in the directed graph. Let $d^{\leftrightarrow}_\ell(u, v)$ denote the distance between two nodes u and v in the undirected graph. For a given parameter H_ℓ, for any node $u \in V'_\ell$, we set $r^{\rightarrow}_\ell(u) := r^{\leftarrow}_\ell(u) := d^{\leftrightarrow}_\ell(u, v)$, where v is the node whose Dijkstra rank $\text{rk}_u(v)$ (w.r.t. the undirected graph) is H_ℓ. For any node $u \notin V'_\ell$, we set $r^{\rightarrow}_\ell(u) := r^{\leftarrow}_\ell(u) := \infty$ (to fulfil R1).

Originally, we wanted to apply the above strategy to the forward and backward graph one after the other in order to define the forward and backward radius, respectively. However, it turned out that using the same value for both forward and backward radius yields a similar good performance, but needs only half the memory.

Fast Construction: Outline. Given a graph G'_ℓ, we want to construct a highway network $G_{\ell+1}$. We start with an empty set of highway edges $E_{\ell+1}$. For each node $s_0 \in V'_\ell$, two phases are performed: the forward construction of a partial shortest-path DAG B (containing *all* shortest paths from s_0 to any node $u \in B$) and the backward evaluation of B. The construction is done by an SSSP search from s_0; during the evaluation phase, paths from the leaves of B to the root s_0 are traversed and for each edge on these paths, it is decided whether to add it to $E_{\ell+1}$ or not. The crucial part is the specification of an abort criterion for the SSSP search in order to restrict it to a 'local search'.

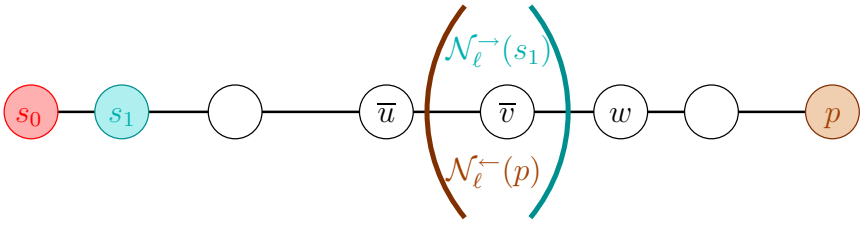

Fig. 3. Abort criterion.

Phase 1: Construction of a Partial Shortest-Path DAG. A Dijkstra search from s_0 is executed. In order to keep track of all shortest paths, for each node in the partial shortest-path DAG B, we manage a list of (tentative) parents: when an edge (u, v) is relaxed such that $d_\ell(s_0, u) + w(u, v) = \delta(v)$, then u is added to the list of tentative parents of v. During the search, a reached node is either in the state *active* or *passive*. The source node s_0 is active; each node that is reached for the first time (*insert*) and each reached node that is updated (*decreaseKey*) is set to active iff any of its tentative parents is active. When a node p is settled, we consider all shortest paths P' from s_0 to p as depicted in Figure 3. The state of p is set to passive if

$$\forall \text{ shortest paths } P' = \langle s_0, \ldots, p \rangle :$$

$$s_1 \prec p \ \wedge \ p \notin \mathcal{N}_\ell^{\rightarrow}(s_1) \ \wedge \ s_0 \notin \mathcal{N}_\ell^{\leftarrow}(p) \ \wedge \ |P' \cap \mathcal{N}_\ell^{\rightarrow}(s_1) \cap \mathcal{N}_\ell^{\leftarrow}(p)| \leq 1 \tag{1}$$

When no active unsettled node is left, the search is *aborted* and the growth of B stops.

An example for Phase 1 of the construction is given in Figure 4. The intuitive reason for s_1 (which is the first successor of s_0 on the path P') to appear in the abort criterion is

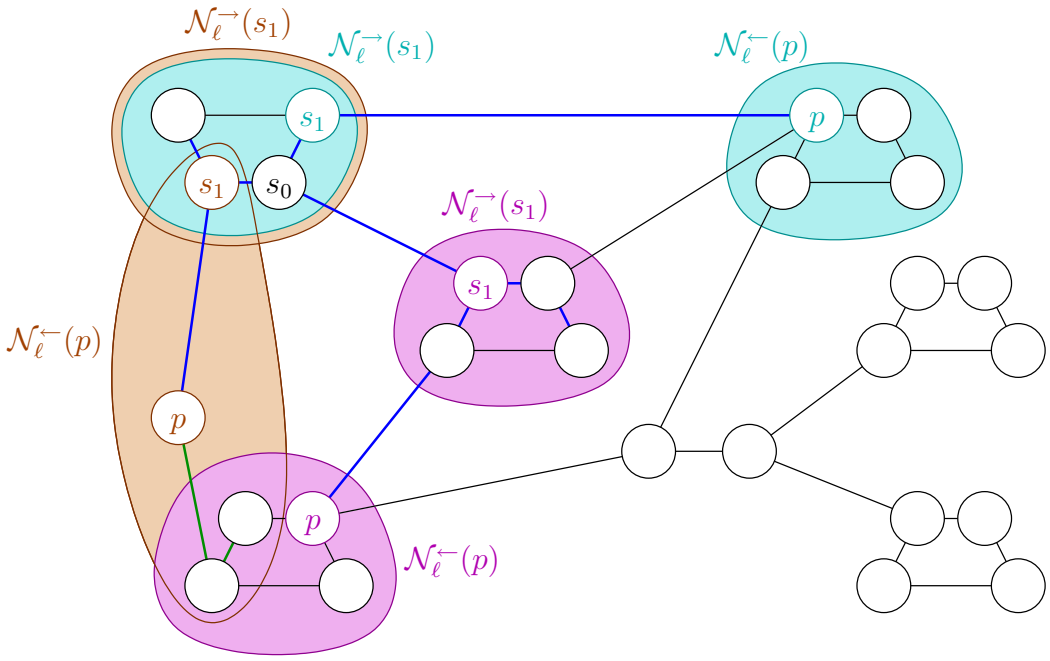

Fig. 4. An example of Phase 1 of the construction. The weight of an edge is the length of the line segment that represents the edge in this figure. The neighbourhood size H_ℓ is 3. An SSSP search is performed from s_0. The abort criterion applies three times: the involved nodes s_1 and p and the corresponding neighbourhoods are marked in cyan, magenta, and brown, respectively. In the brown case, the intersection of the concerned neighbourhoods contains exactly one element; in the other two cases, the intersections are empty. All edges that belong to s_0's partial shortest-path tree are coloured: edges that leave active nodes are blue, edges that leave passive nodes are green.

the following: When we deactivate a node p during the search from s_0, we decide to ignore everything that lies behind p. We are free to do this because the abort criterion ensures that s_1 can take 'responsibility' for the things that lie behind p, i.e., further important edges will be added during the search from s_1. (Of course, s_1 will refer a part of its 'responsibility' to its successor, and so on.)

Phase 2: Selection of the Highway Edges. During Phase 2, exactly all edges (u, v) are added to $E_{\ell+1}$ that lie on paths $\langle s_0, \ldots, u, v, \ldots, p\rangle$ in the partial shortest-path DAG B with the property that $v \notin \mathcal{N}_\ell^{\rightarrow}(s_0)$ and $u \notin \mathcal{N}_\ell^{\leftarrow}(p)$. The example from Figure 4 is continued in Figure 5.

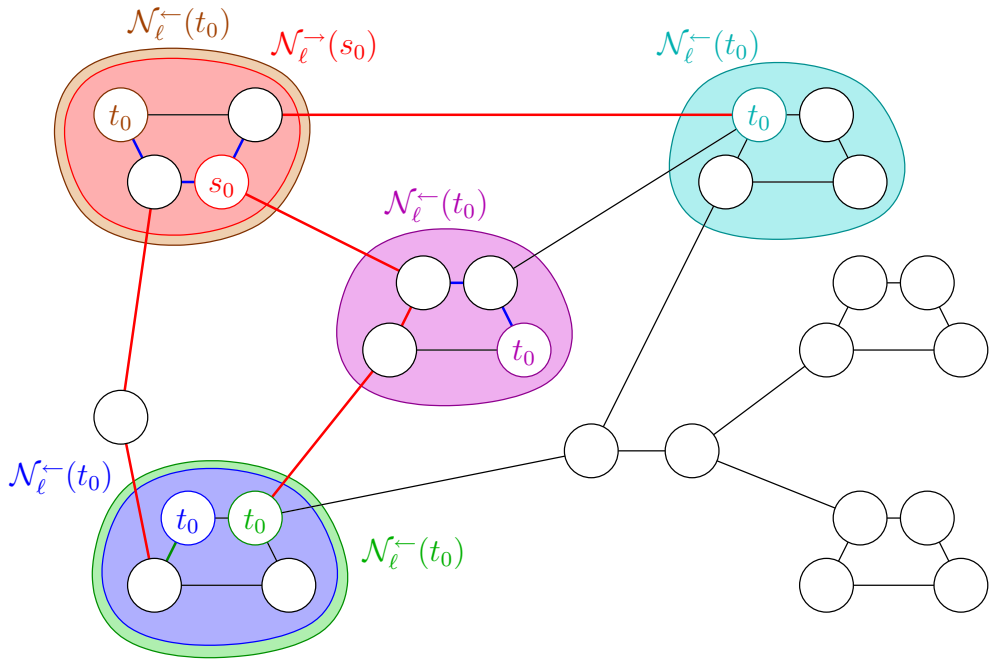

Fig. 5. An example of Phase 2 of the construction. s_0's partial shortest path tree has five leaves t_0, which are marked in different colours. The edges that are added to $E_{\ell+1}$ are highlighted.

Theorem 1. *An edge $(u, v) \in E'_\ell$ is added to $E_{\ell+1}$ by the construction algorithm iff it belongs to some shortest path $P = \langle s, \ldots, u, v, \ldots, t\rangle$ and $v \notin \mathcal{N}_\ell^{\rightarrow}(s)$ and $u \notin \mathcal{N}_\ell^{\leftarrow}(t)$.*

Proof. In this proof, we will refer to the following Neighbourhood Property N1 that follows directly from the neighbourhood definition: Consider a shortest path $\langle s, \ldots, u, \ldots, t\rangle$ in G'_ℓ. Then, $t \in \mathcal{N}_\ell^{\rightarrow}(s)$ implies $u \in \mathcal{N}_\ell^{\rightarrow}(s)$ and $s \in \mathcal{N}_\ell^{\leftarrow}(t)$ implies $u \in \mathcal{N}_\ell^{\leftarrow}(t)$.

$\Leftarrow$) Consider the node s_0 on $P|_{s\to u}$ such that $v \notin \mathcal{N}_\ell^{\rightarrow}(s_0)$ and $d_\ell(s_0, v)$ is minimal. Such a node s_0 exists because the condition $v \notin \mathcal{N}_\ell^{\rightarrow}(s_0)$ is always fulfilled for $s_0 = s$. The direct successor of s_0 on P is denoted by s_1. Note that $v \in \mathcal{N}_\ell^{\rightarrow}(s_1)$ [*]. We show that the edge (u, v) is added to $E_{\ell+1}$ when Phase 1 and 2 are executed from s_0. Due to the specification of Phase 2, it is sufficient to prove that after Phase 1 has been completed, the partial shortest-path DAG B contains a node $p \in P|_{s_0\to t}$ such that $v \preceq p$ and $u \notin \mathcal{N}_\ell^{\leftarrow}(p)$.

If $t \in B$, this statement is obviously fulfilled for $p := t$ since $v \preceq t$ and $u \notin \mathcal{N}_\ell^{\leftarrow}(t)$. Otherwise ($t \notin B$), the search is not continued from some node $t_0 \prec t$ on $P|_{s_0\to t}$. We can conclude that t_0 is passive because, otherwise, its successor on $P|_{s_0\to t}$ would adopt its active state and the search would not be aborted at that time. Since s_0 is active and t_0 is passive, either t_0 or one of its ancestors must have been switched from active to passive. Let p denote

the first passive node on $P|_{s_0 \to t} = \langle s_0, s_1, \ldots, p, \ldots, t_0, \ldots, t \rangle$. Due to the definition of the abort condition, we have $s_1 \prec p \wedge p \notin \mathcal{N}_\ell^{\rightarrow}(s_1) \wedge s_0 \notin \mathcal{N}_\ell^{\leftarrow}(p) \wedge |P' \cap \mathcal{N}_\ell^{\rightarrow}(s_1) \cap \mathcal{N}_\ell^{\leftarrow}(p)| \leq 1$ [**], where $P' = P|_{s_0 \to p}$. The facts that $v \in \mathcal{N}_\ell^{\rightarrow}(s_1)$ [see *] and $p \notin \mathcal{N}_\ell^{\rightarrow}(s_1)$ [see **] imply $v \prec p$ due to N1. In order to obtain a contradiction, we assume $u \in \mathcal{N}_\ell^{\leftarrow}(p)$. Since $s_0 \notin \mathcal{N}_\ell^{\leftarrow}(p)$ [see **], this implies $s_0 \prec u$ by N1. Hence, $s_1 \preceq u$. Because $v \in \mathcal{N}_\ell^{\rightarrow}(s_1)$ [see *], we obtain $u \in \mathcal{N}_\ell^{\rightarrow}(s_1)$ due to N1. Similarly, we get $v \in \mathcal{N}_\ell^{\leftarrow}(p)$ since $v \prec p$ and $u \in \mathcal{N}_\ell^{\leftarrow}(p)$. Thus, $\{u, v\} \subseteq P' \cap \mathcal{N}_\ell^{\rightarrow}(s_1) \cap \mathcal{N}_\ell^{\leftarrow}(p)$. Therefore, $|P' \cap \mathcal{N}_\ell^{\rightarrow}(s_1) \cap \mathcal{N}_\ell^{\leftarrow}(p)| \geq 2$, which is a contradiction to [**]. We can conclude that $u \notin \mathcal{N}_\ell^{\leftarrow}(p)$.

$\Rightarrow$) Since each path $\langle s_0, \ldots, u, v, \ldots, p \rangle$ in B is a shortest path, the claim follows directly from the specification of Phase 2. □

Algorithmic Details: Phase 1. For an efficient implementation, we keep track of a *border distance* $b(x)$ and a *reference distance* $a(x)$ for each node x in B. Along a path P' as depicted in Figure 3, we assign $b(x)$ the distance from the root to the border of the neighbourhood of s_1 as soon as s_1 is settled. This value is passed to all successors on the path, which allows to determine the first node w outside $\mathcal{N}_\ell^{\rightarrow}(s_1)$, i.e., its direct predecessor $\overline{v}$ is the last node inside $\mathcal{N}_\ell^{\rightarrow}(s_1)$. In order to fulfil the abort condition, we have to make sure that $\overline{v}$ is the only node on P' within $\mathcal{N}_\ell^{\rightarrow}(s_1) \cap \mathcal{N}_\ell^{\leftarrow}(p)$. Therefore, we want to check whether $\overline{v}$'s direct predecessor $\overline{u}$ belongs to $\mathcal{N}_\ell^{\leftarrow}(p)$. To allow an easy check, we determine, store, and propagate the reference distance from s_0 to $\overline{u}$ as soon as w is settled. Knowing the reference distance $d_\ell(s_0, \overline{u})$, the current distance $d_\ell(s_0, p)$ and p's neighbourhood radius $r_\ell^{\leftarrow}(p)$, checking $\overline{u} \notin \mathcal{N}_\ell^{\leftarrow}(p)$ is then straightforward. If there are several shortest paths from s_0 to some node x, we determine appropriate maxima of the involved border and reference distances.

More formally, for any node x in B, $\pi(x)$ denotes the set of parent nodes in B. To avoid some case distinctions, we set $\pi(s_0) := \{s_0\}$, i.e., the root is its own parent. For the root s_0, we set $b(s_0) := 0$ and $a(s_0) := \infty$. For any other node $x \neq s_0$, we define $b'(x) := d_\ell(s_0, x) + r_\ell^{\rightarrow}(x)$ if $s_0 \in \pi(x)$, and 0, otherwise; $b(x) := \max(\{b'(x)\} \cup \{b(y) \mid y \in \pi(x)\})$; $a'(x) := \max\{a(y) \mid y \in \pi(x)\}$; and $a(x) := \max\{d_\ell(s_0, u) \mid y \in \pi(x) \wedge u \in \pi(y)\}$ if $a'(x) = \infty \wedge d_\ell(s_0, x) > b(x)$, and $a'(x)$, otherwise.

Then, we can easily check the following abort criterion at a settled node p:

$$a(p) + r_\ell^{\leftarrow}(p) < d_\ell(s_0, p) \tag{2}$$

Lemma 1. *(2) implies (1).*

Proof. We prove the contraposition "¬ (1) implies ¬ (2)", i.e., we assume that there is some shortest path P' from s_0 to p such that $p \preceq s_1 \vee p \in \mathcal{N}_\ell^{\rightarrow}(s_1) \vee s_0 \in \mathcal{N}_\ell^{\leftarrow}(p) \vee |P' \cap \mathcal{N}_\ell^{\rightarrow}(s_1) \cap \mathcal{N}_\ell^{\leftarrow}(p)| \geq 2$ and show that $a(p) + r_\ell^{\leftarrow}(p) \geq d_\ell(s_0, p)$.
Case 1: $p \preceq s_1$. If $p = s_0$, then $a(p) = \infty$, which yields ¬ (2). Otherwise ($p = s_1$), $b(p) \geq d_\ell(s_0, p) + r_\ell^{\rightarrow}(p)$, $a'(p) = \infty$, and $a(p) = a'(p)$ since $d_\ell(s_0, p) \leq b(p)$, which implies ¬ (2).
Case 2: $s_1 \prec p \wedge p \in \mathcal{N}_\ell^{\rightarrow}(s_1)$. Due to N1 (see proof of Theorem 1), we have $\forall x, s_1 \preceq x \preceq p : x \in \mathcal{N}_\ell^{\rightarrow}(s_1)$. Hence, $\forall x : d_\ell(s_0, x) \leq d_\ell(s_0, s_1) + r_\ell^{\rightarrow}(s_1) \leq b(x)$. By an inductive proof, we can show that $a(p) = \infty$, which yields ¬ (2).
Case 3: $s_1 \prec p \wedge p \notin \mathcal{N}_\ell^{\rightarrow}(s_1) \wedge s_0 \in \mathcal{N}_\ell^{\leftarrow}(p)$. We have $d_\ell(s_0, p) \leq r_\ell^{\leftarrow}(p)$, which directly implies ¬ (2).
Case 4: $s_1 \prec p \wedge p \notin \mathcal{N}_\ell^{\rightarrow}(s_1) \wedge s_0 \notin \mathcal{N}_\ell^{\leftarrow}(p) \wedge |P' \cap \mathcal{N}_\ell^{\rightarrow}(s_1) \cap \mathcal{N}_\ell^{\leftarrow}(p)| \geq 2$. The assumption of Case 4 implies that there are two nodes $\overline{u}$ and $\overline{v}$, $s_1 \preceq \overline{u} \prec \overline{v} \preceq p$, that belong to $P' \cap \mathcal{N}_\ell^{\rightarrow}(s_1) \cap \mathcal{N}_\ell^{\leftarrow}(p)$. If $a(p) = \infty$, we directly have ¬ (2). Otherwise, there has to be some node w on P' such that $a'(w) = \infty \wedge d_\ell(s_0, w) > b(w)$. Obviously, $w \neq s_0$. Consider

such a node w that maximises $d_\ell(s_0, w)$, i.e., for all nodes $x \succ w$ the above stated condition does not hold, which implies $a(x) = a'(x) \geq a(w)$. In particular, $a(p) \geq a(w)$. We have $b(w) \geq d_\ell(s_0, s_1) + r_\ell^{\rightarrow}(s_1)$. We can conclude that $d_\ell(s_0, w) > d_\ell(s_0, s_1) + r_\ell^{\rightarrow}(s_1)$ and, thus, $w \notin \mathcal{N}_\ell^{\rightarrow}(s_1)$. We obtain, by N1, $\overline{u} \prec \overline{v} \prec w$. Hence, $a(w) \geq d_\ell(s_0, \overline{u})$, which implies $a(p) \geq d_\ell(s_0, \overline{u})$. Furthermore, since $\overline{u} \in \mathcal{N}_\ell^{\leftarrow}(p)$, we have $r_\ell^{\leftarrow}(p) \geq d_\ell(\overline{u}, p)$. Adding up the last two inequalities yields $a(p) + r_\ell^{\leftarrow}(p) \geq d_\ell(s_0, p)$, which corresponds to $\neg$ (2). □

Algorithmic Details: Phase 2. For a node $u \in B$, we define $\mathcal{B}(u) := \{u\} \cup \{v \mid v \text{ is a descendant of } u \text{ in } B\}$ and the *slack* $\Delta(u) := \min_{w \in \mathcal{B}(u)} (r_\ell^{\leftarrow}(w) - d_\ell(u, w))$. For a leaf b, we have $\mathcal{B}(b) = \{b\}$ and $\Delta(b) = r_\ell^{\leftarrow}(b)$. The slack of an inner node u can be computed using only the slacks of its children $C(u)$: $\Delta(u) = \min\left(r_\ell^{\leftarrow}(u), \min_{c \in C(u)} \Delta_c(u)\right)$, where $\Delta_c(u) := \Delta(c) - d_\ell(u, c)$. This leads to an equivalent, recursive definition.

The tentative slacks $\widehat{\Delta}(u)$ of all nodes u in B are set to $r_\ell^{\leftarrow}(u)$. We process all nodes in the reverse order as they were settled. This guarantees that all descendants of some node u have been processed before u is processed. We can stop as soon as a node $u \in \mathcal{N}_\ell^{\rightarrow}(s_0)$ is encountered. We maintain the invariant that the tentative slack $\widehat{\Delta}(u)$ of an element u that is processed is equal to the actual slack $\Delta(u)$. When a node u is processed, for each parent p of u in B, we perform the following steps: compute $\Delta_u(p) = \Delta(u) - d_\ell(p, u)$; if $\Delta_u(p) < 0$, the edge (p, u) is added to $E_{\ell+1}$; if $\Delta_u(p) < \widehat{\Delta}(p)$, the tentative slack $\widehat{\Delta}(p)$ is set to $\Delta_u(p)$. Figure 6 gives an example.

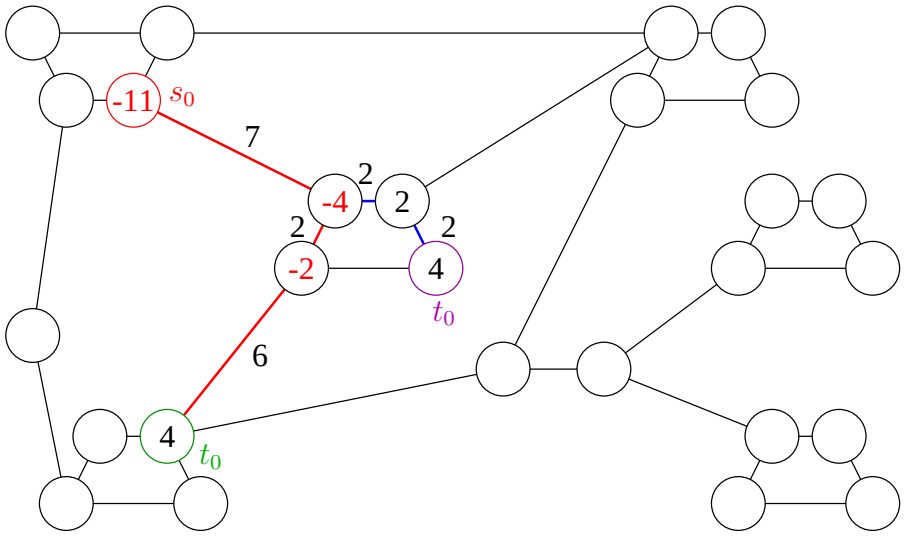

Fig. 6. An example of the *slack-based method* that realises Phase 2 of the construction. The process is shown only for a part of the tree. As before, the weight of an edge is the length of the line that represents the edge in this figure. For the sake of transparency, the (rounded) weights are given explicitly for the relevant edges. Furthermore, the slacks of the involved nodes are given. Edges that are added to $E_{\ell+1}$ are red, edges that are not added blue.

Theorem 2. *An edge (u, v) is added to $E_{\ell+1}$ by the* slack-based method *introduced above iff it lies on a path $\langle s_0, \ldots, u, v, \ldots, p\rangle$ in the partial shortest-path DAG B with the property that $v \notin \mathcal{N}_\ell^{\rightarrow}(s_0)$ and $u \notin \mathcal{N}_\ell^{\leftarrow}(p)$.*

Proof. $\Leftarrow$) From the definition of the slack of a node, it follows that

$$\Delta_v(u) = \Delta(v) - d_\ell(u, v) \leq r_\ell^{\leftarrow}(p) - d_\ell(v, p) - d_\ell(u, v) = r_\ell^{\leftarrow}(p) - d_\ell(u, p) < 0$$

because $u \notin \mathcal{N}_\ell^{\leftarrow}(p)$. Since $v \notin \mathcal{N}_\ell^{\rightarrow}(s_0)$, v is processed at some point. Then, $\Delta_v(u)$ is computed and, since it is negative, the edge (u, v) is added to $E_{\ell+1}$.

$\Rightarrow$) Only edges that belong to a path in B from s_0 to a node p are considered. The condition $v \notin \mathcal{N}_\ell^{\rightarrow}(s_0)$ is never violated because the traversal from the leaves to the root, and consequently, the addition of edges to $E_{\ell+1}$, is not continued when a node $v \in \mathcal{N}_\ell^{\rightarrow}(s_0)$ is encountered. If an edge (u, v) is added, the condition $\Delta_v(u) < 0$ is fulfilled. Hence, $\Delta(u) = \min_{w \in \mathcal{B}(u)} (r_\ell^{\leftarrow}(w) - d_\ell(u, w)) \leq \Delta_v(u) < 0$. Therefore, there is a node p such that $d_\ell(u, p) > r_\ell^{\leftarrow}(p)$, i.e., $u \notin \mathcal{N}_\ell^{\leftarrow}(p)$. □

Theorem 3. *Let V_B denote the set of nodes of s_0's partial shortest-path DAG B. Let $G_B = (V_B, E_B)$ denote the subgraph of G'_ℓ that is vertex induced by V_B. The complexity of Phase 1 and 2 started from s_0 is $T_{\textit{Dijkstra}}(|G_B|)$.*

Proof. The number of nodes of G_B is denoted by n', the number of edges by m'. The complexity of Phase 1 corresponds to the complexity of a SSSP search in G_B started from s_0, i.e., $O(n' + m')$ outside the priority queue plus n' *insert* and n' *deleteMin* operations plus at most m' *decreaseKey* operations. During Phase 2, each node and each edge is processed at most once, i.e., Phase 2 runs in $O(n' + m')$. □

Speeding Up the Highway Network Construction. Even a single active endpoint of a long edge (e.g., a long-distance ferry connection) can cause a large search space during construction, although most nodes of the search space might already be passive. To face this undesirable effect, we declare an active node v to be a *maverick* if $d_\ell(s_0, v) > f \cdot r_\ell^{\rightarrow}(s_0)$, where f is a parameter. When all active nodes are mavericks, the search from passive nodes is no longer continued. This way, the construction process is accelerated and $E_{\ell+1}$ becomes a superset of the highway network. Hence, queries will be slower, but still compute exact shortest paths. The *maverick factor* f enables us to adjust the trade-off between construction and query time.

4.2 Computing the Core

In order to obtain the core of a highway network, we contract it, which yields several advantages. The search space during the queries gets smaller since bypassed nodes are not touched and the construction process gets faster since the next iteration only deals with the nodes that have not been bypassed. Furthermore, a more effective contraction allows us to use smaller neighbourhood sizes without compromising the shrinking of the highway networks. This improves both construction and query times. However, bypassing nodes involves the creation of shortcuts, i.e., edges that represent the bypasses. Due to these shortcuts, the average degree of the graph is increased and the memory consumption grows. In particular, more edges have to be relaxed during the queries. Therefore, we have to carefully select nodes so that the benefits of bypassing them outweigh the drawbacks.

We give an iterative algorithm that combines the selection of the bypassable nodes B_ℓ with the creation of the corresponding shortcuts. We manage a stack that contains all nodes that have to be considered, initially all nodes from V_ℓ. As long as the stack is not empty, we deal with the topmost node u. We check the *bypassability criterion* $\#\text{shortcuts} \leq c \cdot (\deg_{\text{in}}(u) + \deg_{\text{out}}(u))$, which compares the number of shortcuts that would be created when u was bypassed with the sum of the in- and outdegree of u. The magnitude of the contraction is determined by the parameter c. If the criterion is fulfilled, the node is bypassed, i.e., it is added to B_ℓ and the appropriate shortcuts are created. Note that the creation of the shortcuts alters the degree of the corresponding endpoints so that bypassing one node can influence the bypassability criterion of another node. Therefore, all adjacent nodes that have been removed from the stack earlier, have not been bypassed, yet, and are bypassable now are pushed on the stack once again.

Theorem 4. *If $c < 2$, $|E'_\ell|$ is in $O(|V_\ell| + |E_\ell|)$.*

Proof. If a node u is bypassed, the number of edges in the (tentative) core is increased by $\mathcal{D}_u := \#\text{shortcuts} - \deg_{\text{in}}(u) - \deg_{\text{out}}(u)$. (We have to subtract $\deg_{\text{in}}(u)$ and $\deg_{\text{out}}(u)$ since the edges incident to u no longer belong to the core.) Note that $\#\text{shortcuts} = \deg_{\text{in}}(u) \cdot \deg_{\text{out}}(u) - \deg_{\leftrightarrow}(u)$, where $\deg_{\leftrightarrow}(u)$ denotes the number of adjacent nodes v that are connected to u by both an edge (u, v) and an edge (v, u). (We have to subtract $\deg_{\leftrightarrow}(u)$ to account for the fact that a 'shortcut' that would be a self-loop is not created.) We can conclude that $\mathcal{D}_u \leq \deg_{\text{in}}(u) \cdot \deg_{\text{out}}(u) - \deg_{\text{in}}(u) - \deg_{\text{out}}(u)$. If $\deg_{\text{in}}(u) \leq 1$ or $\deg_{\text{out}}(u) \leq 1$, we obtain $\mathcal{D}_u \leq 0$. Now, we deal with the case that $\deg_{\text{in}}(u) \geq 2$ and $\deg_{\text{out}}(u) \geq 2$. Since $\deg_{\leftrightarrow}(u) \leq \min(\deg_{\text{in}}(u), \deg_{\text{out}}(u))$, a node that fulfils the bypassability criterion also fulfils $\deg_{\text{in}}(u) \cdot \deg_{\text{out}}(u) \leq c \cdot (\deg_{\text{in}}(u) + \deg_{\text{out}}(u)) + \min(\deg_{\text{in}}(u), \deg_{\text{out}}(u))$. The inequality $x \cdot y \leq c(x + y) + \min(x, y)$ has only finitely many solutions (x, y) for $x, y \in \mathbb{N}, x, y \geq 2$ if $c \in \mathbb{R}$ is a constant less than 2. Consider the solution (x, y) that maximises $k := x \cdot y$. If there is no solution, take $k := 0$. Note that k is a constant that only depends on the constant c. We can conclude that $\mathcal{D}_u \leq k$.

Each node from V_ℓ is bypassed at most once. For each bypassed node, the number of edges in the (tentative) core is increased by at most k. Therefore, $|E'_\ell| \leq k \cdot |V_\ell| + |E_\ell|$. □

If we used $\#\text{shortcuts} \leq \max(\deg_{\text{in}}(u), \deg_{\text{out}}(u))$ as bypassability criterion, we would get a contraction that would be very similar to our earlier trees-and-lines method [23]. Note that the general version presented above allows a more effective contraction by setting c appropriately.

Limiting Component Sizes. To reduce the observed maximum query time, we implement a limit on the number of hops a shortcut may represent. By this means, the sizes of the components of bypassed nodes are reduced—in particular, the first contraction step tended to create quite large components of bypassed nodes so that it took a long time to leave such a component when the search was started from within it.

5 Query

Our *highway query algorithm* is a modification of the bidirectional version of Dijkstra's algorithm. Note that in contrast to the construction, during the query we need *not* to keep track of ambiguous shortest paths. For now, we assume that the search is *not* aborted when both search scopes meet. This matter is dealt with in Section 5.3. We only describe the modifications of the forward search since forward and backward search are symmetric. In addition to the *distance* from the source, each node is associated with the search *level* and the *gap* to the 'next applicable neighbourhood border'. The search starts at the source node s in level 0. First, a local search in the neighbourhood of s is performed, i.e., the gap to the next border is set to the neighbourhood radius of s in level 0. When a node v is settled, it adopts the gap of its parent u minus the length of the edge (u, v). As long as we stay inside the current neighbourhood, everything works as usual. However, if an edge (u, v) crosses the neighbourhood border (i.e., the length of the edge is greater than the gap), we switch to a higher search level ℓ. The node u becomes an *entrance point* to the higher level. If the level of the edge (u, v) is less than the new search level ℓ, the edge is *not* relaxed—this is one of the two restrictions that cause the speedup in comparison to Dijkstra's algorithm (Restriction 1). Otherwise, the edge is relaxed: v adopts the new search level ℓ and the gap to the border of the neighbourhood of u in level ℓ since u is the corresponding entrance point to level ℓ.

We have to deal with the special case that an entrance point to level ℓ does not belong to the core of level ℓ. In this case, the search is continued inside a component of bypassed nodes till the level-ℓ core is entered, i.e., a node $u \in V'_\ell$ is settled. At this point, u is assigned the gap to the border of the level-ℓ neighbourhood of u. Note that before the core is entered (i.e., inside a component of bypassed nodes), the gap has been infinity (according to R1). To increase the speedup, we introduce another restriction (Restriction 2): when a node $u \in V'_\ell$ is settled, all edges (u, v) that lead to a bypassed node $v \in B_\ell$ in search level ℓ are *not* relaxed, i.e., once entered the core, we will never leave it again.

Figure 7 gives a detailed example of the forward search of a highway query. The search starts at node s. The gap of s is initialised to the distance from s to the border of the neighbourhood of s in level 0. Within the neighbourhood of s, the search process corresponds to a standard Dijkstra search. The edge that leads to u leaves the neighbourhood. It is not relaxed due to Restriction 1 since the edge belongs only to level 0. In contrast, the edge that leaves s_1 is relaxed since its level allows to switch to level 1 in the search process. s_1 and its direct successor are bypassed nodes in level 1. Their neighbourhoods are unbounded, i.e., their neighbourhood radii are infinity so that the gap is set to infinity as well. At s'_1, we leave the component of bypassed nodes and enter the core of level 1. Now, the search is continued in the core of level 1 within the neighbourhood of s'_1. The gap is set appropriately. Note that the edge to v is not relaxed due to Restriction 2 since v is a bypassed node. Instead, the direct shortcut to s_2 is used. Here, we switch to level 2. In this case, we do not enter the next level through a component of bypassed nodes, but we get directly into the core. The search is continued in the core of level 2 within the neighbourhood of s'_2. And so on.

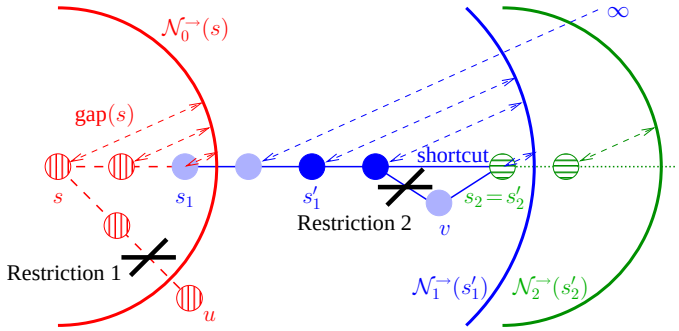

Fig. 7. A detailed example of a highway query. Only the forward search is depicted. Nodes in level 0, 1, and 2 are vertically striped, solid, and horizontally striped, respectively. In level 1, dark shades represent core nodes, light shades bypassed nodes. Edges in level 0, 1, and 2 are dashed, solid, and dotted, respectively.

Despite of Restriction 1, we always find the optimal path since the construction of the highway hierarchy guarantees that the levels of the edges that belong to the optimal path are sufficiently high so that these edges are not skipped. Restriction 2 does not invalidate the correctness of the algorithm since we have introduced shortcuts that bypass the nodes that do not belong to the core. Hence, we can use these shortcuts instead of the original paths.

5.1 The Basic Algorithm

We use two priority queues $\overrightarrow{Q}$ and $\overleftarrow{Q}$, one for the forward search and one for the backward search. For each search direction, a node u is associated with a triple $(\delta(u), \ell(u), \text{gap}(u))$, which we often call *key*. It consists of the (tentative) distance $\delta(u)$ from s (or t) to u, the search level $\ell(u)$, and the gap gap(u) to the next applicable neighbourhood border. Only the

first component $\delta(u)$ is used to decide the priority within the queue.[2] We use the remaining two components for a tie breaking rule in the case that the same node is reached with two different keys $k := (\delta, \ell, \text{gap})$ and $k' := (\delta', \ell', \text{gap}')$ such that $\delta = \delta'$. Then, we prefer k to k' iff $\ell > \ell'$ or $\ell = \ell' \wedge \text{gap} < \text{gap}'$. Note that *any* other tie breaking rule (or even omitting an explicit rule) will yield a correct algorithm. However, the chosen rule is most aggressive and has a positive effect on the performance. Figure 8 contains the pseudo-code of the highway query algorithm.

input: source node s and target node t
output: distance $d(s,t)$

```
 1  d' := ∞;
 2  insert(Q→, s, (0, 0, r0→(s))); insert(Q←, t, (0, 0, r0←(t)));
 3  while (Q→ ∪ Q← ≠ ∅) do {
 4      select direction ⇋ ∈ {→, ←} such that Q⇋ ≠ ∅;
 5      u := deleteMin(Q⇋);
 6      if u has been settled from both directions then d' := min(d', δ→(u) + δ←(u));
 7      if gap(u) ≠ ∞ then gap' := gap(u) else gap' := r⇋ℓ(u)(u);
 8      foreach e = (u, v) ∈ E⇋ do {
 9          for (ℓ := ℓ(u), gap := gap'; w(e) > gap;
                ℓ++, gap := r⇋ℓ(u)); // go "upwards"
10          if ℓ(e) < ℓ then continue; // Restriction 1
11          if u ∈ V'ℓ ∧ v ∈ Bℓ then continue; // Restriction 2
12          k := (δ(u) + w(e), ℓ, gap − w(e));
13          if v has been reached then decreaseKey(Q⇋, v, k); else insert(Q⇋, v, k);
14      }
15  }
16  return d';
```

Fig. 8. The highway query algorithm. Differences to the bidirectional version of Dijkstra's algorithm are marked: additional and modified lines have a framed line number; in modified lines, the modifications are underlined.

Remarks:

- Line 4: The correctness of the algorithm does not depend on the strategy that determines the order in which the forward and the backward searches are processed. However, the choice of the strategy can affect the running time in the case that an abort-on-success criterion is applied (see Section 5.3).
- Line 7: This line deals with the special case that the entrance point did not belong to the core when the current search level ℓ was entered, i.e., the gap was set to infinity. In this case, the gap is set to $r^{\rightleftharpoons}_{\ell(u)}(u)$. This is correct even if u does not belong to the core, either, because in this case the gap stays at infinity.
- Line 9: It might be necessary to go upwards more than one level in a single step.
- Line 13: In the decreaseKey operation, the old key of v is only replaced by k if the above mentioned condition is fulfilled, i.e., if (a) the tentative distance is improved or (b) stays unchanged while the tie breaking rule succeeds. In the latter case (b), no priority queue operation is invoked since the priority (the tentative distance) has not changed.[3]

[2] If the search direction is not clear from the context, we will explicitly write $\overrightarrow{\delta}(u)$ and $\overleftarrow{\delta}(u)$ to distinguish between u's priority in $\overrightarrow{Q}$ and $\overleftarrow{Q}$.

[3] That way, we also avoid problems that otherwise could arise when an already settled node is reached once again via a zero weight edge.

Algorithmic Details. If we group the outgoing edges (u, v) of each node u by level, we can avoid looking at edges (u, v) in levels $\ell(u, v) < \ell(u)$ since Restriction 1 would always apply to them. We can do without explicitly testing Restriction 2 if all edges (u, v) with $k := \ell(u, v), u \in V'_k$, and $v \in B_k$ have been downgraded to level $k - 1$. Then, the test of Restriction 1 also covers Restriction 2.

5.2 Proof of Correctness

Difficulties. Although the basic concepts (e.g. the definition of the highway network) and the algorithm are quite simple, the proof of correctness gets surprisingly complicated. The main reason for that is the fact that we cannot prove that *the* shortest path is found since there might be several shortest paths of the same length. We could assume that the shortest paths in the input are unique or that the uniqueness can be guaranteed by adding small fractions to the edge weights as it is done by other authors who face similar problems. However, the former would be too restrictive since usually, in real-world road networks, there are at least a few ambiguous instances, and a reliable realisation of the latter would be rather difficult. Furthermore, the introduction of shortcuts adds a lot of ambiguity even if it was not present in the input.

Therefore, if we pick any shortest path P to show that it is found by the query algorithm, it can happen that a node u on P is settled from another node than its predecessor on P. Of course, in this case, u will still be assigned the optimal distance from the source, but the search level and the distance to the next neighbourhood border may be different than expected so that we have to adapt to the new situation.

Outline. We face the above mentioned difficulties in the following way: First, we show that the algorithm terminates and deal with the special case that no path from the source to the target exists (Section A.1). Then, we introduce some definitions and concepts that will be useful in the main part of the correctness proof. In Section A.2, we define for a given path, a corresponding *contracted* path and an *expanded* path, where subpaths in the original graph are replaced by shortcuts or vice versa, respectively. In Section A.3, we first define the concepts of *last neighbour* and *first core node*, which, iteratively applied, lead to an *unidirectional labelling* of a given path. Figure 9 gives an example. Applying a forward and a backward labelling to the same path then allows the definition of a *meeting level* and a *meeting point* (Figure 10). The latter requires a case distinction since the forward and backward labelling may either meet in some core or in some component of bypassed nodes. Finally, we introduce the term *highway path*, a path whose properties exactly comply with the two restrictions of the query algorithm. Figure 11 gives an example.

In Section A.4, we deal with the reachability along a highway path. Basically, we show that if the query has settled some node u on a highway path with the appropriate key, then

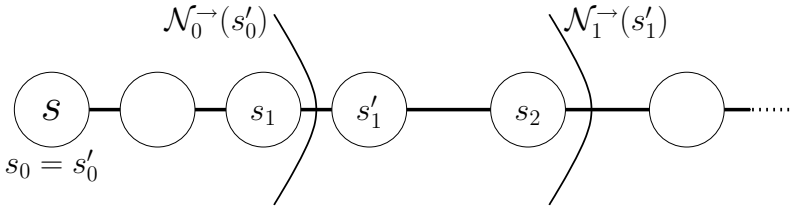

Fig. 9. Example for a forward labelling of a path P. The labels s_0 and s'_0 are set to s (base case). The node s_1 is the last neighbour of s'_0 (denoted by $\overrightarrow{\omega}_0^P(s'_0)$), the node s'_1 is the first level-1 core node (denoted by $\overrightarrow{\alpha}_1^P(s_1)$), s_2 is the last neighbour of s'_1, and so on.

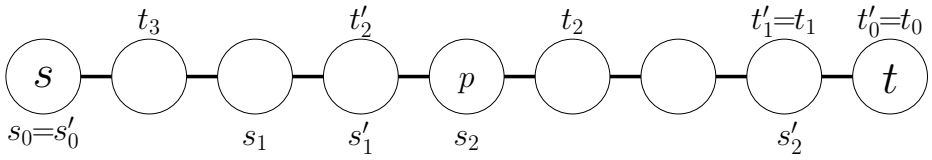

Fig. 10. Example for a forward and backward labelling (depicted below and above the nodes, respectively). The meeting level is 2, the meeting point is p.

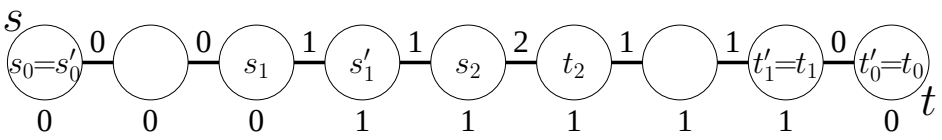

Fig. 11. Example for a highway path. Each edge belongs at least to the given level, each node at least to the given core level.

u's successor on that path can be reached from u with the appropriate key as well (Lemmas 6 and 7, which are proved using the auxiliary Lemma 5). In other words, if there is a highway path, the query can follow the path (at least if there was no ambiguity).

In Section A.5, we use all concepts and lemmas introduced in the preceding sections to conduct the actual correctness proof, where we also deal with ambiguous paths. The general idea is to say that at any point the query algorithm has some valid *state* consisting of a shortest s-t-path P and two nodes $u \preceq \overline{u}$ that split P into three parts such that the first and the third part are paths in the forward and backward search tree, respectively, and the second part is a contracted path. For such a valid state, we can prove that any node on the first and third part has been settled with the appropriate key (Lemma 8). Furthermore, we can show that P is a highway path (Lemma 9).

When the algorithm is started, the nodes s and t are settled and some shortest s-t-path P in the original graph exists. (The special case that no s-t-path exists has already been dealt with.) Consequently, our *initial* state is composed of the contracted version of P and the nodes s and t, which makes it a valid state. A *final* state is a valid state where forward and backward search have met, i.e., they have settled a common node $u = \overline{u}$. Originally, we wanted to show that a shortest path is found. Now, we see (in Lemma 10) that it is sufficient to prove that a final state is reached.

We have already defined the meeting point p on a path. We fall back on this definition and intend to prove that forward and backward search meet at p. When we look at any valid non-final state, it is obvious that at least one search direction can proceed to get closer to p, i.e., we have $u \prec p$ or $p \prec \overline{u}$ (Lemma 11). We pick such a *non-blocked* search direction. Let us assume w.l.o.g. that we picked the forward direction. We know that u has been settled with the appropriate key and that P is an optimal highway path (Lemmas 8 and 9). Due to the 'reachability along a highway path' (Lemmas 6 and 7), we can conclude that u's successor v can be reached with the appropriate key as well, in particular with the optimal distance from s. A node that can be reached with the optimal distance will also be settled at some point with the optimal distance. However, we cannot be sure that v is settled with u as its parent since the shortest path from s to v might be ambiguous. At this point the state concept gets handy: we just replace the subpath of P from s to v with the path in the search tree that actually has been taken yielding a path P^+; we obtain a new state that consists of P^+ and the nodes v and $\overline{u}$. We prove that the new state is valid (Lemma 12).

Thus, we can show that from any valid non-final state another valid state is reached at some point. We also show in Lemma 12 that we cannot get into some cycle of states since

in each step the length of the middle part of the path is decreased. Hence, starting from the initial state, eventually a final state is reached so that a shortest path is found (Theorem 5).

The actual proof can be found in Appendix A.

5.3 Optimisations

Rearranging Nodes. Similar to [20], after the construction has been completed, we rearrange the nodes by core level, which improves locality for the search in higher levels and, thus, reduces the number of cache misses.

Speeding Up the Search in the Topmost Level. Let us assume that we have a distance table that contains for any node pair $s, t \in V'_L$ the optimal distance $d_L(s,t)$. Such a table can be precomputed during the preprocessing phase by $|V'_L|$ SSSP searches in G'_L. Using the distance table, we do not have to search in level L. Instead, when we arrive at a node $u \in V'_L$ that leads to level L, we add u to the initially empty set $\overrightarrow{I}$ or $\overleftarrow{I}$ depending on the search direction; we do not relax the edge that leads to level L. After all entrance points have been encountered, we consider all pairs $(u,v) \in \overrightarrow{I} \times \overleftarrow{I}$ and compute the minimum path length $D := \overrightarrow{\delta}(u) + d_L(u,v) + \overleftarrow{\delta}(v)$. Then, the length of the shortest s-t-path is the minimum of D and the length d' of the tentative shortest path found so far (in case that the search scopes have already met in a level $< L$).

For the sake of a simple incorporation of this optimisation into the highway query algorithm, we slightly revise the properties R1 and R2: we use two distinguishable values ∞_1 and ∞_2 that are larger than any real number and set $r_\ell^{\leftrightharpoons}(u) := \infty_1$ for any ℓ and any node $u \notin V'_\ell$ (R1) and $r_L^{\leftrightharpoons}(u) := \infty_2$ for any node $u \in V'_L$ (R2). Then, we just add two lines to Figure 8 and modify Line 16:

between Lines 7 and 8:
7a **if** gap$' \neq \infty_1 \wedge \ell(u) = L$ **then** $\{\overleftrightharpoons{I} := \overleftrightharpoons{I} \cup \{u\}$; **continue**;$\}$

between Lines 11 and 12:
11a **if** gap $\neq \infty_1 \wedge \ell = L \wedge \ell > \ell(u)$ **then** $\{\overleftrightharpoons{I} := \overleftrightharpoons{I} \cup \{u\}$; **continue**;$\}$

16 **return** $\min(\{d'\} \cup \{\,\overrightarrow{\delta}(u) + d_L(u,v) + \overleftarrow{\delta}(v) \mid u \in \overrightarrow{I}, v \in \overleftarrow{I}\,\})$;

In Section A.6, we show that our proof of correctness still holds when the distance table optimisation is applied.

Abort on Success. In the bidirectional version of Dijkstra's algorithm, we can abort the search as soon as both search scopes meet. Unfortunately, this would be incorrect for our highway query algorithm. Therefore, we use a more conservative criterion: after a tentative shortest path P' has been encountered (i.e., after both search scopes have met), the forward (backward) search is not continued if the minimum element u of the forward (backward) queue has a key $\delta(u) \geq w(P')$. Obviously, the correctness of the algorithm is not invalidated by this abort criterion. In [23] we tried using more sophisticated criteria in order to reduce the search space. However, it turned out that this simple criterion, since it can be evaluated so efficiently, yields better query times in spite of a somewhat larger search space. Note that when the distance table optimisation is used and random queries are performed, our simple abort criterion is very close to an optimal criterion even with respect to the search space size: our experiments indicate that less than 1% of the search space is visited after the first meeting of forward and backward search.

5.4 Outputting Complete Path Descriptions

The highway query algorithm in Figure 8 only computes the distance from s to t. In order to determine the actual shortest path, we need to store pointers from each node to its parent in the search tree. Note that the algorithm could be easily modified to compute *all* shortest paths between s and t by just storing more than one parent pointer in case of ambiguities. However, subsequently, we only deal with a single shortest path.

We face two problems in order to determine a complete description of the shortest path: (a) we have to bridge the gap between the forward and backward topmost core entrance points (in case that the distance table optimisation is used) and (b) we have to expand the used shortcuts to obtain the corresponding subpaths in the original graph.

Problem (a) can be solved using a simple algorithm: We start with the forward core entrance point u. As long as the backward entrance point v has not been reached, we consider all outgoing edges (u, w) in the topmost core and check whether $d_L(u, w) + d_L(w, v) = d_L(u, v)$; we pick an edge (u, w) that fulfils the equation, and we set $u := w$. The check can be performed using the distance table. It allows us to greedily determine the next hop that leads to the backward entrance point.

Problem (b) can be solved without using any extra data (Variant 1). For each shortcut $(u, v) \in S_\ell$ on the shortest path, we perform a search from u to v in order to determine the represented path in G_ℓ. This search can be accelerated by using the knowledge that the first edge of the path enters a component C of bypassed nodes, the last edge leads to v, and all other edges are situated within the component C. The represented path in G_ℓ may contain shortcuts from sets $S_k, k < \ell$, which are expanded recursively. In the end, we obtain the represented path from u to v in the original graph.

However, if a fast output routine is required, it is necessary to spend some additional space to accelerate the unpacking process. We use a rather sophisticated data structure to represent unpacking information for the shortcuts in a space-efficient way (Variant 2). In particular, we do not store a sequence of node IDs that describe a path that corresponds to a shortcut, but we store only *hop indices*: for each edge (u, v) on the path that should be represented, we store its rank within the ordered group of edges that leave u. Since in most cases the degree of a node is very small, these hop indices can be stored using only a few bits. The unpacked shortcuts are stored in a recursive way, e.g., the description of a level-2 shortcut may contain several level-1 shortcuts. Accordingly, the unpacking procedure works recursively.

To obtain a further speed-up, we have a variant of the unpacking data structures (Variant 3) that caches the complete descriptions—without recursions—of all shortcuts that belong to the topmost level, i.e., for these important shortcuts that are frequently used, we do not have to use a recursive unpacking procedure, but we can just append the corresponding subpath to the resulting path.

5.5 Turning Restrictions

A turning restriction (in its simplest and most common form) is expressed as an edge pair $((u, v), (v, w))$: the edge (v, w) must not be traversed if the node v has been reached via the edge (u, v). Dealing with turning restrictions is a well-studied problem [33, 34]. In principle, there are two basic approaches: modifying the query algorithm or modelling the restrictions into the graph, which introduces additional artificial nodes and edges at affected road junctions. The latter technique can be applied irrespective of the used query algorithm.

In case of highway hierarchies, we expect that modelling turning restrictions into the graph only slightly deteriorates the performance since the artificial nodes usually have a

very small degree so that most of them get bypassed in the very first contraction step. Furthermore, turning restrictions are often encountered at local streets that are not promoted to high levels of the hierarchy so that the negative impact is bounded to the lower levels. With respect to memory consumption, it is important to note that after the preprocessing has been completed, artificial nodes and edges at road junctions that only belong to level 0 can be abandoned provided that the query algorithm (which in level 0 just corresponds to Dijkstra's algorithm) is modified appropriately to handle turning restrictions.

6 Experiments

Apart from Section 6.8, all experimental results refer to the scenario where we only want to compute the shortest-path length between two nodes without outputting the actual route. Turning restrictions are exclusively handled in Section 6.9.

6.1 Implementation

We implemented highway hierarchies in C++, using the C++ Standard Template Library and making extensive use of *generic programming* techniques using C++'s template class mechanism. As graph data structure, we use our own implementation of an *adjacency array* extended by an additional layer that contains level-specific data for each node and level that the node belongs to. We use 32 bits to store edge weights and path lengths. *Binary heaps* are used as priority queues. Note that in case of road networks only a comparatively small number of *decreaseKey*-operations is performed. Furthermore, the number of nodes that are in the priority queue at the same time is very small in case of highway hierarchies (usually less than 100 nodes). Therefore, using a more sophisticated priority queue implementation is not likely to increase the performance significantly. For some more details on the implementation, we refer to Appendix B.

6.2 Environment and Instances

The experiments were 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 10.0 (kernel 2.6.13). The program was compiled by the GNU C++ compiler 4.0.2 using optimisation level 3.

We deal with the road networks of Western Europe[4] and of the USA (without Hawaii) and Canada. Both networks have been made available for scientific use by the company PTV AG. The original graphs contain for each edge a length and a road category, e.g., motorway, national road, regional road, urban street. We assign average speeds to the road categories[5], compute for each edge the average travel time, and use it as weight. In addition, we perform experiments on a publicly available version of the US road network (without Alaska and Hawaii) that was obtained from the TIGER/Line Files [35]. However, in contrast to the PTV data, the TIGER graph is undirected, planarised and distinguishes only between four road categories (40, 60, 80, 100 km/h), in fact 91% of all roads belong to the slowest category so that you cannot discriminate them.

Table 1 summarises important properties of the used road networks and the key results of the experiments.

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

[5] For Europe: 10, 20, . . ., 130 km/h; for USA/CAN: 16, 24, 32, 40, 56, 64, 72, 80, 88, 96, 96, 104, 112 km/h.

Table 1. Overview of the used road networks and key results. '∅overhead/node' accounts for the *additional* memory that is needed by our highway hierarchy approach (divided by the number of nodes) compared to a space-efficient bidirectional implementation of Dijkstra's algorithm. Query times are average values based on 10 000 random s-t-queries. 'Speedup' refers to a comparison with Dijkstra's algorithm (unidirectional). Worst case is an upper bound for *any* possible query in the respective graph.

		Europe	USA/CAN	USA (Tiger)
INPUT	#nodes	18 029 721	18 741 705	24 278 285
	#directed edges	42 199 587	47 244 849	58 213 192
	#road categories	13	13	4
PARAM.	average speeds [km/h]	10–130	16–112	40–100
	H	30	40	40
PREPROC.	CPU time [min]	13	17	15
	∅overhead/node [byte]	48	46	34
QUERY	CPU time [ms]	0.61	0.83	0.67
	#settled nodes	709	871	925
	#relaxed edges	2 531	3 376	3 823
	speedup (CPU time)	9 935	7 259	9 303
	speedup (#settled nodes)	12 715	10 750	12 889
	worst case (#settled nodes)	2 388	2 428	2 505

6.3 Parameters

Default Settings. Unless otherwise stated, the following default settings apply. We use the *maverick factor* $f = 2(i-1)$ for the i-th iteration of the construction procedure, the contraction rate $c = 2$, the shortcut hops limit 10, and the neighbourhood sizes H as stated in Table 1—the same neighbourhood size is used for all levels of a hierarchy. First, we contract the original graph.[6] Then, we perform five iterations of our construction procedure, which determines a highway network and its core. Finally, we compute the distance table between all level-5 core nodes.

Self-Similarity. For two levels ℓ and $\ell+1$ of a highway hierarchy, the *shrinking factor* is the ratio between $|E'_\ell|$ and $|E'_{\ell+1}|$. In our experiments, we observed that the highway hierarchies of Europe and the USA were almost *self-similar* in the sense that the shrinking factor remained nearly unchanged from level to level when we used the same neighbourhood size H for all levels—provided that H was not too small.

Figure 12 demonstrates the shrinking process for Europe. Note that the first contraction step is not shown. In contrast to our default settings, we do not stop after five iterations. For most levels and $H \geq 70$, we observe an almost constant shrinking factor (which appears as a straight line due to the logarithmic scale of the y-axis). The greater the neighbourhood size, the greater the shrinking factor. The last iteration is an exception: the highway network collapses, i.e., it shrinks very fast because nodes that are close to the border of the network usually do not belong to the next level of the highway hierarchy, and when the network gets small, almost all nodes are close to the border. In case of the smallest neighbourhood size ($H = 30$), the shrinking factor gets so small that the network does not collapse even after 14 levels have been constructed.

Varying the Neighbourhood Size. Note that in order to simplify the experimental setup all results in the remainder of Section 6.3 have been obtained without rearranging nodes by level. However, since we want to demonstrate the effects of choosing different parameter settings, the relative performance is already very meaningful.

[6] In Section 3, we gave the definition of the highway hierarchies where we first construct a highway network and then contract it. We decided to change this order in the experiments, i.e., to start with an initial contraction phase, since we observed a better performance in this case.

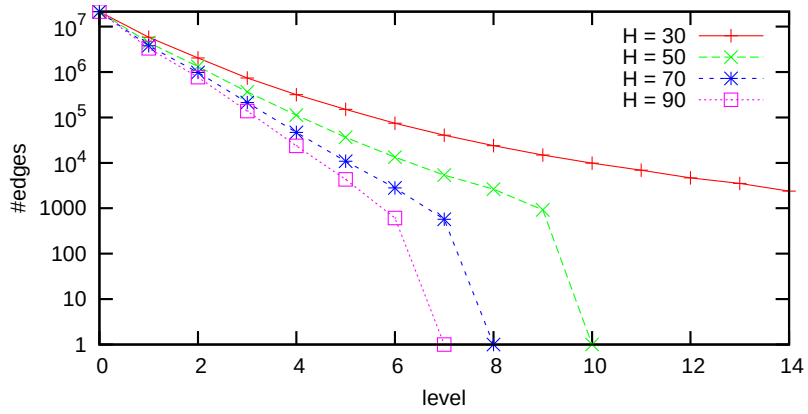

Fig. 12. Shrinking of the highway networks of Europe. For different neighbourhood sizes H and for each level ℓ, we plot $|E'_\ell|$, i.e., the number of edges that belong to the core of level ℓ.

In one test series (Figure 13), we used all the default settings except for the neighbourhood size H, which we varied in steps of 5. On the one hand, if H is too small, the shrinking of the highway networks is less effective so that the level-5 core is still quite big. Hence, we need much time and space to precompute and store the distance table. On the other hand, if H gets bigger, the time needed to preprocess the lower levels increases because the area covered by the local searches depends on the neighbourhood size. Furthermore, during a query, it takes longer to leave the lower levels in order to get to the topmost level where the distance table can be used. Thus, the query time increases as well. We observe that the preprocessing time is minimised for neighbourhood sizes around 40. In particular, the optimal neighbourhood size does not vary very much from graph to graph. In other words, if we used the same parameter H, say 40, for all road networks, the resulting performance would be very close to the optimum. Obviously, choosing different neighbourhood sizes leads to different space-time trade-offs.

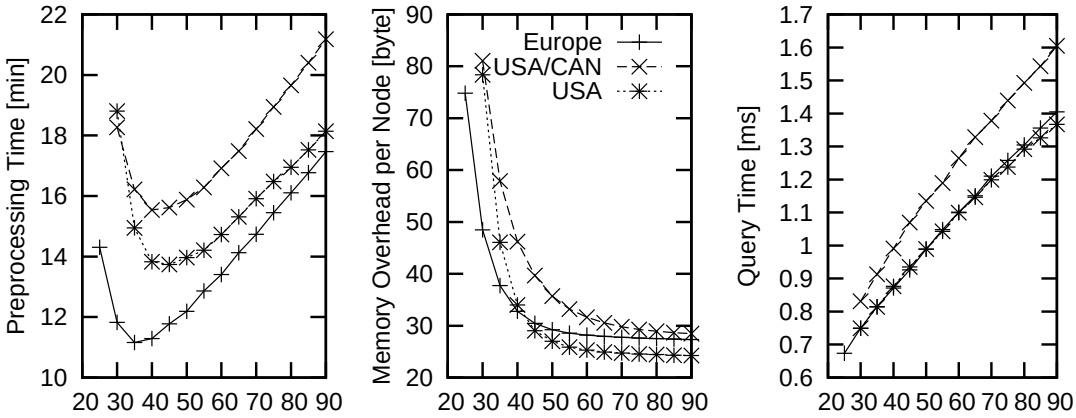

Fig. 13. Preprocessing and query performance depending on the neighbourhood size H.

Varying the Contraction Rate. In another test series (Table 2a), we did not use a distance table, but repeated the construction process until the topmost level was empty or the hierarchy consisted of 15 levels. We varied the contraction rate c from 0.5 to 2.5. In case of $c = 0.5$ (and $H = 30$), the shrinking of the highway networks does not work properly so that the topmost level is still very big. This yields huge query times. Choosing larger contraction rates

reduces the preprocessing and query times since the cores and search spaces get smaller. However, the memory usage and the average degree are slightly increased since more shortcuts are introduced. Adding too many shortcuts ($c = 2.5$) further reduces the search space, but the number of relaxed edges increases so that the query times get worse.

Varying the Number of Levels. In a third test series (Table 2b), we used the default settings except for the number of levels, which we varied from 6 to 11. Note that the original graph and its core (i.e., the result of the first contraction step) counts as one level so that for example '6 levels' means that only five levels are constructed. In each test case, a distance table was used in the topmost level. The construction of the higher levels of the hierarchy is very fast and has no significant effect on the preprocessing times. In contrast, using only six levels yields a rather large distance table, which somewhat slows down the preprocessing and increases the memory usage. However, in terms of query times, '6 levels' is the optimal choice since using the distance table is faster than continuing the search in higher levels. We omitted experiments with less levels since this would yield very large distance tables consuming very much memory.

Results for further combinations of neighbourhood size, contraction rate, and number of levels can be found in Table 5 and 6 in Appendix C.

Table 2. Preprocessing and query performance for the European road network depending on the contraction rate c (a) and the number of levels (b). 'overhead' denotes the average memory overhead per node in bytes.

contr. rate c	PREPROCESSING time [min]	over-head	∅deg.	QUERY time [ms]	#settled nodes	#relaxed edges
0.5	83	30	3.2	391.73	472 326	1 023 944
1.0	15	28	3.7	5.48	6 396	23 612
1.5	11	28	3.8	1.93	1 830	9 281
2.0	11	29	4.0	1.85	1 542	8 913
2.5	11	30	4.1	1.96	1 489	9 175

(a)

# levels	PREPROC. time [min]	over-head	QUERY time [ms]	#settled nodes
6	12	48	0.75	709
7	10	34	0.93	852
8	10	30	1.14	991
9	10	30	1.35	1 123
10	10	29	1.54	1 241
11	10	29	1.67	1 326

(b)

6.4 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. On the other hand, it would be hardly more realistic to arbitrarily cut the graph into smaller pieces. Therefore, we decided to measure local queries within the big graphs: For each power of two $r = 2^k$, we choose random sample points s and then use Dijkstra's algorithm to find the node t with Dijkstra rank $\text{rk}_s(t) = r$. 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. Figure 14 shows the query times. Note that for ranks up to 2^{18} the median query times are scaling quite smoothly and the growth is much slower than the exponential increase we would expect in a plot with logarithmic x axis, linear y axis, and any growth rate of the form r^ρ for Dijkstra rank r and some constant power ρ; the curve is also not the straight line one would expect from a query time logarithmic in r. For ranks $r \geq 2^{19}$, the query times hardly rise due to the fact that the all-pairs distance table can bridge the gap between the forward and backward search of these queries irrespective of dealing with a small or a large gap. In case of Europe and USA/CAN, the query times drop for $r = 2^{24}$ since r is only slightly smaller than the number of nodes so that the target lies close to the border of the respective road network which implies some kind of trivial sense of goal direction for the backward search (because, in the beginning, we practically cannot go into the wrong direction).

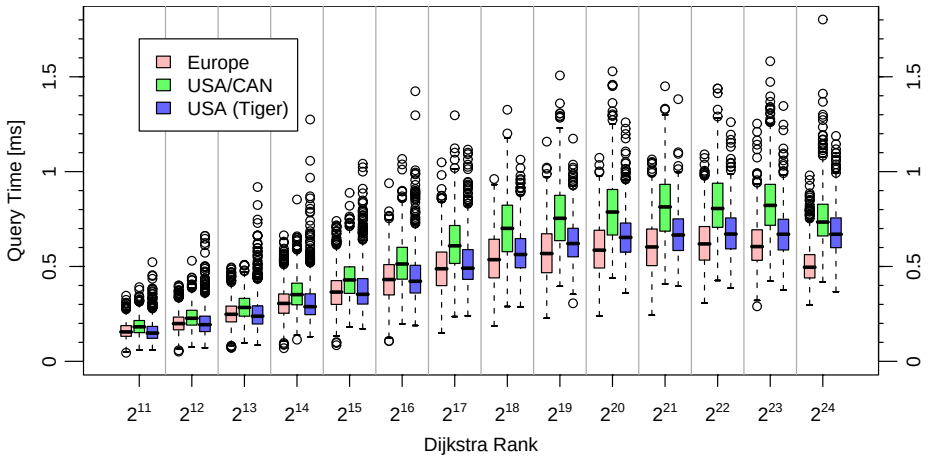

Fig. 14. Local queries. The distributions are represented as box-and-whisker plots [36]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

6.5 Space Saving

If we omit the first contraction step and use a smaller contraction rate ($\Rightarrow$ less shortcuts), use a bigger neighbourhood size ($\Rightarrow$ higher levels get smaller), and construct more levels before the distance table is used ($\Rightarrow$ smaller distance table), the memory usage can be reduced considerably. In case of Europe, using seven levels, $H = 100$, and $c = 1$ yields an average overhead per node of 17 bytes. The construction and query times increase to 55 min and 1.10 ms, respectively.

6.6 Worst Case Upper Bounds

By executing a query from each node of a given graph to an added isolated dummy node and a query from the dummy node to each actual node in the backward graph, we obtain a distribution of the search space sizes of the forward and backward search, respectively. We can combine both distributions to get an upper bound for the distribution of the search space sizes of bidirectional queries: when $\mathcal{F}_{\rightarrow}(x)$ ($\mathcal{F}_{\leftarrow}(x)$) denotes the number of source (target) nodes whose search space consists of x nodes in a forward (backward) search, we define $\mathcal{F}_{\leftrightarrow}(z) := \sum_{x+y=z} \mathcal{F}_{\rightarrow}(x) \cdot \mathcal{F}_{\leftarrow}(y)$, i.e., $\mathcal{F}_{\leftrightarrow}(z)$ 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 $\max\{z \mid \mathcal{F}_{\leftrightarrow}(z) > 0\}$ for the worst case without performing all n^2 possible queries.

Figure 15 visualises the distribution $\mathcal{F}_{\leftrightarrow}(z)$ as a histogram. In a similar way, we obtained a distribution of the number of entries in the distance table that have to be accessed during an s-t-query. While the average values are reasonably small (4 066 in case of Europe), the worst case can get quite large (62 379). For example, accessing 62 379 entries in a table of size 9 351 $\times$ 9 351 takes about 1.1 ms, where 9 351 is the number of nodes of the level-5 core of the European highway hierarchy. Hence, in some cases the time needed to determine the optimal entry in the distance table might dominate the query time. We could try to improve the worst case by introducing a case distinction that checks whether the number of entries that have to be considered exceeds a certain threshold. If so, we would not use the distance table, but continue with the normal search process. However, this measures would have only little effect on the *average* performance.

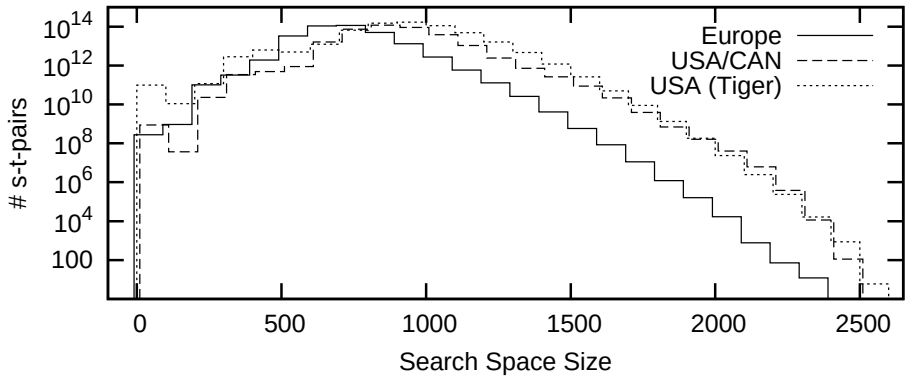

Fig. 15. Histogram of upper bounds for the search space sizes of s-t-queries. To increase readability, only the outline of the histogram is plotted instead of the complete boxes.

6.7 Comparisons

In Table 3, we compare our highway hierarchies with some of the most competitive methods where experimental results are available for the Western European and the US road network, namely with the REAL algorithm [20], transit-node routing [27], and highway-node routing [29]. For REAL and transit-node routing slightly smaller graphs were used, namely the largest connected component of each road network consisting of about 99% of all nodes. Experiments with the REAL algorithm have been performed on a slightly different machine (dual-processor, 2.4 GHz AMD Opteron).

Although a comparison is difficult since all approaches allow different choices of parameter settings yielding different space-time trade-offs, we can make some general statements: The strength of transit-node routing is clearly the extremely good query performance. Highway-node routing has an outstandingly low memory consumption, while the query times are competitive to highway hierarchies and REAL or even slightly superior in case of the US road network. Highway hierarchies can achieve very low preprocessing times or a quite low memory consumption, while query times are reasonably good in all cases. REAL's performance is similar to highway hierarchies except for the preprocessing times, which tend to be considerably higher.

It is very important to note that both highway-node routing and the implementation of transit-node routing considered in this section are based on our highway hierarchies. Thus, at the moment neither of these methods can supersede the highway hierarchies approach. Actually, the results for highway-node routing are better than the ones published in [29]

Table 3. Comparison between highway hierarchies (HH), the REAL algorithm, transit-node routing (TNR) and highway-node routing (HNR). For the former three approaches, different parameter settings are examined. 'disk space' denotes the total amount of memory needed to store the preprocessed data *including* the original graph on disk.

	Europe				USA (Tiger)			
	PREPROCESSING		QUERY		PREPROCESSING		QUERY	
method	time [min]	disk space [MB]	time [ms]	#settled nodes	time [min]	disk space [MB]	time [ms]	#settled nodes
HH	13	1 241	0.61	709	15	1 324	0.67	925
HH (mem)	55	697	1.10	1 863	70	942	1.21	2 143
REAL (16,1)	97	1 849	1.22	814	64	3 028	1.14	675
REAL (64,16)	141	1 015	1.11	679	121	1 575	1.05	540
TNR (eco)	46	2 304	0.0134	N/A	59	3 073	0.0115	N/A
TNR (gen)	164	4 714	0.0056	N/A	205	6 108	0.0049	N/A
HNR	15	503	0.88	1 017	16	640	0.50	760

since in the meantime the highway hierarchies have improved and highway-node routing directly benefits from that.

6.8 Outputting Complete Path Descriptions

So far, we have reported only the times needed to compute the shortest-path length between two nodes. Now, we determine a complete description of the shortest path. In Table 4 we give the additional preprocessing time and the additional disk space for the unpacking data structures. Furthermore, we report the additional time that is needed to determine a complete description of the shortest path and to traverse[7] it summing up the weights of all edges as a sanity check—assuming that the query to determine the shortest-path length has already been performed. That means that the total average time to determine a shortest path is the time given in Table 4 plus the query time given in previous tables[8]. Note that Variant 1 is no longer supported by the current version of our implementation so that the numbers in the first data row of Table 4 have been obtained with an older version and different settings.

We can conclude that even Variant 3 requires little additional preprocessing time and only a moderate amount of space. With Variant 3, the time for outputting the path remains considerably smaller than the time to determine the path length and a factor 3–5 smaller than using Variant 2. The US graph profits more than the European graph since it has paths with considerably larger hop counts, perhaps due to a larger number of degree two nodes in the input. Note that due to cache effects, the time for outputting the path using preprocessed shortcuts is likely to be considerably smaller than the time for traversing the shortest path in the original graph.

Table 4. Additional preprocessing time, additional disk space and query time that is needed to determine a complete description of the shortest path and to traverse it summing up the weights of all edges—assuming that the query to determine its lengths has already been performed. Moreover, the average number of hops—i.e., the average path length in terms of number of nodes—is given. The three algorithmic variants are described in Section 5.4.

	Europe				USA (Tiger)			
	preproc. [s]	space [MB]	query [ms]	# hops (avg.)	preproc. [s]	space [MB]	query [ms]	# hops (avg.)
Variant 1	0	0	17.22	1 366	0	0	39.69	4 410
Variant 2	69	126	0.49	1 366	68	127	1.16	4 410
Variant 3	74	225	0.19	1 366	70	190	0.25	4 410

6.9 Turning Restrictions

We did an experiment with the German road network (a subgraph of our European network) and real-world turning restrictions (also provided by PTV) to verify our expectation that incorporating the restrictions into the graph has only a little effect on the performance. The results are positive: the preprocessing time does not change, the total number of nodes and edges in the highway hierarchy only increases by 4%, and the query times rise by 3%.

6.10 Distance Metric

When we apply a distance metric instead of the usual (and for most practical applications more relevant) travel time metric, the hierarchy that is inherent in the road network is less

[7] Note that we do *not* traverse the path in the original graph, but we directly scan the assembled description of the path.

[8] Note that in the current implementation outputting path descriptions and the feature to rearrange nodes by level are mutually exclusive. However, this is not a limitation in principle.

distinct since the difference between fast and slow roads fades. We no longer observe the self-similarity in the sense that a fixed neighbourhood size yields an almost constant shrinking factor. Instead, we have to use an increasing sequence of neighbourhood sizes to ensure a proper shrinking. For Europe, we use H =100, 200, 300, 400, 500 to construct five levels before an all-pairs distance table is built. Constructing the hierarchy takes 34 minutes and entails a memory overhead of 36 bytes per node. On average, a random query then takes 4.88 ms, settling 4 810 nodes and relaxing 33 481 edges. Further experiments on different metrics can be found in [26].

6.11 An Even Larger Road Network

Very recently, we obtained a new version of the European road network that is larger than the old one and covers more countries[9]. It has been provided for scientific use by the company ORTEC and consists of 33 726 989 nodes and 75 108 089 directed edges. We use the same parameters as for the old version (in particular, $H = 30$) and observe a very good shrinking behaviour: we have 1.87 times as many nodes in the beginning, but after the construction of the same number of levels only 1.04 times as many nodes remain. Thus, the same number of levels is sufficient, only the distance table gets slightly bigger. We arrive at a preprocessing time of 18 minutes, a memory overhead of 37 bytes per node, and query times of 0.60 ms for random queries; on average, 685 nodes are settled and 2 457 edges are relaxed.

7 Discussion

Highway hierarchies are a simple, robust and space-efficient concept that allows very efficient exact fastest-path queries even in huge real-world road networks. These attributes have been confirmed in an extensive experimental study. Although highway hierarchies are already very useful when applied directly, their usefulness extends to a much wider range: some concepts like the contraction of a network have turned out to be advantageous for other speedup techniques as well; they can be extended to deal with many-to-many queries; the currently fastest shortest-path algorithm for static road networks is based on them; an efficient approach to dealing with dynamic scenarios like traffic jams uses highway hierarchies in its preprocessing phase; . . .

Nevertheless, a lot of interesting questions remain. How to handle mobile devices with limited fast memory? How to deal with multiple objective functions or with time-dependent edge weights? What about public transportation networks?

We are optimistic that highway hierarchies and related methods are a promising starting point to tackle several of these problems.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
2. Sanders, P., Schultes, D.: Engineering fast route planning algorithms. In: 6th Workshop on Experimental Algorithms. Volume 4525 of LNCS., Springer (2007) 23–36
3. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on System Science and Cybernetics **4** (1968) 100–107
4. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms. (2005) 156–165
5. Goldberg, A.V., Werneck, R.F.: An efficient external memory shortest path algorithm. In: Workshop on Algorithm Engineering and Experimentation. (2005) 26–40

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

6. Maue, J., Sanders, P., Matijevic, D.: Goal directed shortest path queries using Precomputed Cluster Distances. In: 5th Workshop on Experimental Algorithms (WEA). Volume 4007 of LNCS., Springer (2006) 316–328
7. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: An empirical case study from public railroad transport. In: 3rd Workshop on Algorithm Engineering. Volume 1668 of LNCS., Springer (1999) 110–123
8. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In: 11th European Symposium on Algorithms. Volume 2832 of LNCS., Springer (2003) 776–787
9. Lauther, U.: 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., IfGI prints, Institut für Geoinformatik, Münster (2004) 219–230
10. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. In: 4th International Workshop on Efficient and Experimental Algorithms. (2005) 126–138
11. Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speed up Dijkstra's algorithm. In: 4th International Workshop on Efficient and Experimental Algorithms. (2005) 189–202
12. Köhler, E., Möhring, R.H., Schilling, H.: Fast point-to-point shortest path computations with arc-flags. In: 9th DIMACS Implementation Challenge [37]. (2006)
13. Lauther, U.: An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags. In: 9th DIMACS Implementation Challenge [37]. (2006)
14. Thorup, M.: Compact oracles for reachability and approximate distances in planar digraphs. In: 42nd IEEE Symposium on Foundations of Computer Science. (2001) 242–251
15. Muller, L.F., Zachariasen, M.: Fast and compact oracles for approximate distances in planar graphs. In: 15th European Symposium on Algorithms. Volume 4698 of LNCS., Springer (2007) 657–668
16. Fakcharoenphol, J., Rao, S.: Planar graphs, negative weight edges, shortest paths, and near linear time. In: 42nd IEEE Symposium on Foundations of Computer Science. (2001) 232–241
17. Schulz, F., Wagner, D., Zaroliagis, C.D.: Using multi-level graphs for timetable information. In: 4th Workshop on Algorithm Engineering and Experiments. Volume 2409 of LNCS., Springer (2002) 43–59
18. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: 6th Workshop on Algorithm Engineering and Experiments. (2004) 100–111
19. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A^*: Efficient point-to-point shortest path algorithms. In: Workshop on Algorithm Engineering & Experiments, Miami (2006) 129–143
20. Goldberg, A., Kaplan, H., Werneck, R.: Better landmarks within reach. In: 6th Workshop on Experimental Algorithms. Volume 4525 of LNCS., Springer (2007) 38–51
21. Ishikawa, K., Ogawa, M., Azume, S., Ito, T.: Map Navigation Software of the Electro Multivision of the '91 Toyota Soarer. In: IEEE Int. Conf. Vehicle Navig. Inform. Syst. (1991) 463–473
22. Jagadeesh, G., Srikanthan, T., Quek, K.: Heuristic techniques for accelerating hierarchical routing on road networks. IEEE Transactions on Intelligent Transportation Systems **3** (2002) 301–309
23. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: 13th European Symposium on Algorithms. Volume 3669 of LNCS., Springer (2005) 568–579
24. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: 14th European Symposium on Algorithms. Volume 4168 of LNCS., Springer (2006) 804–816
25. Knopp, S., Sanders, P., Schultes, D., Schulz, F., Wagner, D.: Computing many-to-many shortest paths using highway hierarchies. In: Workshop on Algorithm Engineering and Experiments. (2007)
26. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway hierarchies star. In: 9th DIMACS Implementation Challenge [37]. (2006)
27. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In transit to constant time shortest-path queries in road networks. In: Workshop on Algorithm Engineering and Experiments. (2007) 46–59
28. Bast, H., Funke, S., Sanders, P., Schultes, D.: Fast routing in road networks with transit nodes. Science **316** (2007) 566
29. Schultes, D., Sanders, P.: Dynamic highway-node routing. In: 6th Workshop on Experimental Algorithms. Volume 4525 of LNCS., Springer (2007) 66–79
30. Nannicini, G., Baptiste, P., Barbier, G., Krob, D., Liberti, L.: Fast paths in large-scale dynamic road networks. arXiv:0704.1068v2 [cs.NI] (2007)
31. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 2nd edn. MIT Press (2001)
32. Skiena, S.S.: The Algorithm Design Manual. Springer (1998)
33. Schmid, W.: Berechnung kürzester Wege in Straßennetzen mit Wegeverboten. PhD thesis, Universität Stuttgart (2000)
34. Müller, K.: Berechnung kürzester Pfade unter Beachtung von Abbiegeverboten. Student Research Project, Universität Karlsruhe, supervised by F. Schulz and D. Wagner (2005)
35. U.S. Census Bureau, Washington, DC: UA Census 2000 TIGER/Line Files. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html` (2002)
36. R Development Core Team: R: A Language and Environment for Statistical Computing. `http://www.r-project.org` (2004)
37. 9th DIMACS Implementation Challenge: Shortest Paths. `http://www.dis.uniroma1.it/~challenge9/` (2006)

A Query—Proof of Correctness

Additional Notations. '$\circ$' denotes *path concatenation*. $\mathrm{succ}(u, P)$ and $\mathrm{pred}(u, P)$ denote the direct successor and predecessor of u on P, respectively. We just write $\mathrm{succ}(u)$ and $\mathrm{pred}(u)$ if the path is clear from the context. For two nodes u and v on some path, $\min(u, v)$ denotes u if $u \preceq v$ and v otherwise. $\max(u, v)$ is defined analogously. $d_P(u, v) := w(P|_{u \to v})$ denotes the distance from u to v along the path P. Note that for any edge (u, v) on P, we have $w(u, v) = d_P(u, v)$.

A.1 Termination and Special Cases

Since we have set the neighbourhood radius in the topmost level to infinity (R2), we are never tempted to go upwards beyond the topmost level. This observation is formalised in the following lemma.

Lemma 2. *The for-loop in Line 9 of the highway query algorithm always terminates with $\ell \leq L$ and $(\ell = L \to \mathit{gap} = \infty)$.*

Proof. We only consider iterations where the forward search direction is selected; analogous arguments apply to the backward direction. By an inductive proof, we show that at the beginning of any iteration of the main while-loop, we have $\ell(u) \leq L$ and $(\ell(u) = L \to \mathrm{gap}(u) = \infty)$ for any node u in $\overrightarrow{Q}$.
Base Case: True for the first iteration, where only s belongs to $\overrightarrow{Q}$: we have $\ell(s) = 0 \leq L$ and $\mathrm{gap}(s) = r_0^{\rightarrow}(s)$ (Line 2), which is equal to infinity if $L = 0$ (due to R2).
Induction Step: We assume that our claim is true for iteration i and show that it also holds for iteration $i + 1$. Due to the induction hypothesis, we have $\ell(u) \leq L$ and $(\ell(u) = L \to \mathrm{gap}(u) = \infty)$ in Line 5. If $\ell(u) = L$, we have $\mathrm{gap} = \mathrm{gap}' = r_{\ell(u)}^{\rightarrow}(u) = \infty$ (Line 7 and 9, R2); thus the for-loop in Line 9 terminates immediately with $\ell = \ell(u) = L$. Otherwise ($\ell(u) < L$), the for-loop either terminates with $\ell < L$ or reaches $\ell = L$; in the latter case, we have $\mathrm{gap} = r_{\ell}^{\rightarrow}(u) = \infty$ (Line 9, R2); hence, the loop terminates.

Thus, in any case, the loop terminates with $\ell \leq L$ and $(\ell = L \to \mathrm{gap} = \infty)$. Therefore, if the node v adopts the key k in Line 13 (either by a decreaseKey or an insert operation), the new key fulfils the required condition.

This concludes our inductive proof, which also shows that the claim of this lemma holds during any iteration of the main while-loop. □

It is easy to the see that the following property of Dijkstra's algorithm also holds for the highway query algorithm.

Proposition 1. *For each search direction, the sequence of distances $\delta(u)$ of settled nodes u is monotonically increasing.*

Now, we can prove that

Lemma 3. *The highway query algorithm terminates.*

Proof. The for-loop in Line 9 always terminates due to Lemma 2. The for-loop in Line 8 terminates since the edge set is finite. The main while-loop in Line 3 terminates since each node v is inserted into each priority queue at most once, namely if it is unreached (Line 13); if it is reached, it either already belongs to the priority queue or it has already been settled; in the latter case, we know that $\delta(v) \leq \delta(u) \leq \delta(u) + w(e)$ (Proposition 1; edge weights are nonnegative) so that no priority queue operation is performed due to the specification of the decreaseKey operation. □

The *special case* that there is no path from s to t is trivial. The algorithm terminates due to Lemma 3 and returns ∞ since no node can be settled from both search directions (otherwise, there would be some path from s to t). For the remaining proof, we assume that a shortest path from s to t exists in the original graph G.

A.2 Contracted and Expanded Paths

Lemma 4. *Shortcuts do not overlap, i.e., if there are four nodes $u \prec u' \prec v \prec v'$ on a path P in G, then there cannot exist both a shortcut (u, v) and a shortcut (u', v') at the same time.*

Proof. Let us assume that there is a shortcut $(u, v) \in S_\ell$ for some level ℓ. All inner nodes, in particular u', belong to B_ℓ. Since u' does not belong to V'_ℓ, a shortcut that starts from u' can belong only to some level $k < \ell$. If there was a shortcut $(u', v') \in S_k$, the inner node v would have to belong to B_k, which is a contradiction since $v \in V'_\ell$. □

Definition 1. *For a given path P in a given highway hierarchy $\mathcal{G}$, the* contracted *path* $\mathrm{ctr}(P)$ *is defined in the following way: while there is a subpath $\langle u, b_1, b_2, \ldots, b_k, v\rangle$ with $u, v \in V'_\ell$ and $b_i \in B_\ell, 1 \leq i \leq k, k \geq 1$, for some level ℓ, replace it by the shortcut edge $(u, v) \in S_\ell$.*

Note that this definition terminates since the number of nodes in the path is reduced by at least one in each step and the definition is unambiguous due to Lemma 4.

Definition 2. *For a given path P in a given highway hierarchy $\mathcal{G}$, the level-ℓ* expanded *path* $\exp(P, \ell)$ *is defined in the following way: while the path contains a shortcut edge $(u, v) \in S_k$ for some $k > \ell$, replace it by the represented path in G_k.*

Note that this definition terminates since an expanded subpath can only contain shortcuts of a smaller level.

A.3 Highway Path

Consider a given highway hierarchy $\mathcal{G}$ and an arbitrary path $P = \langle s, \ldots, t\rangle$. In the following, we will bring out the structure of P w.r.t. $\mathcal{G}$.

Last Neighbour and First Core Node. For any level ℓ and any node u on P, we define the *last succeeding level-ℓ neighbour* $\overrightarrow{\omega}^P_\ell(u)$ and the *first succeeding level-ℓ core node* $\overrightarrow{\alpha}^P_\ell(u)$: $\overrightarrow{\omega}^P_\ell(u)$ is the node $v \in \{x \in P \mid u \preceq x \wedge d_P(u, x) \leq r^{\rightarrow}_\ell(u)\}$ that maximises $d_P(u, v)$, and $\overrightarrow{\alpha}^P_\ell(u)$ is the node $v \in \{t\} \cup \{x \in P \cap V'_\ell \mid u \preceq x\}$ that minimises $d_P(u, v)$. The *last preceding neighbour* $\overleftarrow{\omega}^P_\ell(u)$ and the *first preceding core node* $\overleftarrow{\alpha}^P_\ell(u)$ are defined analogously.

Unidirectional Labelling. Now, we inductively define a forward *labelling* of the path P. The labels s_0 and s'_0 are set to s and for $\ell, 0 \leq \ell < L$, we set $s_{\ell+1} := \overrightarrow{\omega}^P_\ell(s'_\ell)$ and $s'_{\ell+1} := \overrightarrow{\alpha}^P_{\ell+1}(s_{\ell+1})$. Furthermore, in order to avoid some case distinctions, $s_{L+1} := t$. For an example, we refer to Figure 9.

Proposition 2. *The following properties apply to the (Unidirectional) forward labelling of P:*

- U1: $s = s_0 = s'_0 \preceq s_1 \preceq s'_1 \preceq \ldots \preceq s_L \preceq s'_L \preceq s_{L+1} = t$
- U2a: $\forall \ell, 0 \leq \ell \leq L : \forall u, s'_\ell \preceq u \preceq s_{\ell+1} : d_P(s'_\ell, u) \leq r^{\rightarrow}_\ell(s'_\ell)$
- U2b: $\forall \ell, 0 \leq \ell \leq L : \forall u \succ s_{\ell+1} : d_P(s'_\ell, u) > r^{\rightarrow}_\ell(s'_\ell)$
- U3: $\forall \ell, 0 \leq \ell \leq L : \forall u, s_\ell \preceq u \prec s'_\ell : u \notin V'_\ell$
- U4: $\forall \ell, 0 \leq \ell \leq L : s'_\ell = t \vee s'_\ell \in V'_\ell$

A backward labelling (specifying nodes t_ℓ and t'_ℓ) is defined analogously.

Meeting Level and Point. The *meeting level* λ of P is 0 if $s = t$ and $\max\{\ell \mid s_\ell \preceq t_\ell\}$ if $s \neq t$. Note that $\lambda \leq L$ (in any case) and $t_{\lambda+1} \prec s_{\lambda+1}$ (in case that $s \neq t$). The *meeting point* p of P is either t_λ (if $t_\lambda \preceq s'_\lambda$) or $\min(s_{\lambda+1}, t'_\lambda)$ (otherwise). Figure 10 gives an example.

Proposition 3. *The following properties apply to the Meeting point p of P:*

- M1: $s_\lambda \preceq p \preceq t_\lambda$
- M2: $t_{\lambda+1} \preceq p \preceq s_{\lambda+1}$
- M3: $\forall \ell, 0 \leq \ell \leq L : (s'_\ell \prec p \rightarrow p \preceq t'_\ell) \wedge (p \prec t'_\ell \rightarrow s'_\ell \preceq p)$

Proof. The case $s = t$ is trivial. Subsequently, we assume $s \neq t$. In order to prove M1, M2, and (M3 for $\ell = \lambda$), we distinguish between two cases.
Case 1: $t_\lambda \preceq s'_\lambda$. Then, $p = t_\lambda$. M1 is fulfilled due to the definition of the meeting level, which implies $s_\lambda \preceq t_\lambda$. Furthermore, due to U1, we have $t_{\lambda+1} \preceq t'_\lambda \preceq t_\lambda = p \preceq s'_\lambda \preceq s_{\lambda+1}$ so that M2 and (M3 for $\ell = \lambda$) are fulfilled.
Case 2: $s'_\lambda \prec t_\lambda$. Then, $p = \min(s_{\lambda+1}, t'_\lambda)$.
Subcase 2.1: $s_{\lambda+1} \preceq t'_\lambda$. Then, $p = s_{\lambda+1}$. We have $s_\lambda \preceq s'_\lambda \preceq s_{\lambda+1} = p \preceq t'_\lambda \preceq t_\lambda$ so that M1 and (M3 for $\ell = \lambda$) are fulfilled. Furthermore, M2 holds due to $t_{\lambda+1} \prec s_{\lambda+1}$.
Subcase 2.2: $t'_\lambda \prec s_{\lambda+1}$. Then, $p = t'_\lambda$. Since $s'_\lambda \prec t_\lambda \preceq t$, we know that $s'_\lambda \in V'_\lambda$ (due to U4). Thus, we have $s'_\lambda \preceq t'_\lambda \preceq t_\lambda$ (otherwise ($t'_\lambda \prec s'_\lambda \preceq t_\lambda$), we would have a contradiction with U3). Hence, $s_\lambda \preceq s'_\lambda \preceq t'_\lambda = p \preceq t_\lambda$ so that M1 and (M3 for $\ell = \lambda$) are fulfilled. M2 holds as well since $t_{\lambda+1} \preceq t'_\lambda = p \prec s_{\lambda+1}$.

It remains to show M3 for $\ell < \lambda$ and for $\ell > \lambda$. In the former case, M3 holds due to M1, which implies $s'_\ell \preceq s_\lambda \preceq p \preceq t_\lambda \preceq t'_\ell$ (U1). In the latter case, M3 holds due to M2, which implies $t'_\ell \preceq t_{\lambda+1} \preceq p \preceq s_{\lambda+1} \preceq s'_\ell$ (U1). □

Highway Path. $P = \langle s, \ldots, t \rangle$ is a *highway path* (Figure 11) iff the following two Highway properties are fulfilled:

- H1: $\forall \ell, 0 \leq \ell \leq L$: H1(ℓ)
- H2: $\forall \ell, 0 \leq \ell \leq L$: H2(ℓ)

where

- H1(ℓ): $\forall (u, v), s'_\ell \preceq u \prec v \preceq t'_\ell : u, v \in V'_\ell$
- H2(ℓ): $\forall (u, v), s_\ell \preceq u \prec v \preceq t_\ell : \ell(u, v) \geq \ell$

A.4 Reachability Along a Highway Path

We consider a path $P = \langle s, \ldots, t \rangle$. For a node u on P, we define the *reference level* $\overline{\ell}(u) := \max(\{0\} \cup \{i \mid s_i \prec u\})$.

Proposition 4. *For any two nodes u and v with $u \preceq v$, the following reference Level properties apply:*

- L1: $0 \leq \overline{\ell}(u) \leq L$
- L2: $s_{\overline{\ell}(u)} \preceq u$
- L3: $u \preceq s_{\overline{\ell}(u)+1}$
- L4: $\overline{\ell}(u) \leq \overline{\ell}(v)$

Definition 3. *A node u is said to be* Appropriately *reached/settled with the key $k = (\delta(u), \ell(u), gap(u))$ on the path P iff all of the following conditions are fulfilled:*

- $A_1(k, u)$: $\delta(u) = d_0(s, u)$

– $A_2(k,u)$: $\ell(u) = \overline{\ell}(u)$
– $A_3(k,u)$: $\text{gap}(u) = \begin{cases} \infty & \text{if } u \preceq s'_{\ell(u)} \\ r^{\rightarrow}_{\ell(u)}(s'_{\ell(u)}) - d_P(s'_{\ell(u)}, u) & \text{otherwise} \end{cases}$
– $A_4(u)$: $\forall i : t \neq s'_i \preceq u \rightarrow u \in V'_i$

The following (somewhat technical) lemma will be used to prove Lemmas 6 and 7. Basically, it states that in the highway query algorithm the search level and the gap to the next applicable neighbourhood border are set correctly.

Lemma 5. *Consider a path $P = \langle s, \ldots, t\rangle$ and an edge (u,v) on P. Assume that u is* settled *by the highway query algorithm appropriately with some key k. We consider the attempt to relax the edge (u,v). After Line 9 has been executed, the following Invariants apply w.r.t. the variables ℓ and gap:*

– I1: (a) $s_\ell \preceq u \wedge$ (b) $v \preceq s_{\ell+1}$
– I2: $\ell = \overline{\ell}(v)$
– I3: $\text{gap} = \begin{cases} \infty & \text{if } v \preceq s'_\ell, \\ r^{\rightarrow}_{\ell}(s'_\ell) - d_P(s'_\ell, u) & \text{otherwise.} \end{cases}$

Proof. We distinguish between two cases in order to prove I1 and I3.
Case 1: zero iterations of the for-loop in Line 9 take place ($\ell = \ell(u)$).
In this case, we have $\ell = \ell(u)$ and $w(u,v) \leq \text{gap}'$. Hence, $s_\ell \preceq u$ due to $A_2(k,u)$ and L2 ($\Rightarrow$ I1a). In order to show I1b and I3, we distinguish between three subcases:

– *Subcase 1.1:* $u \prec s'_\ell \Rightarrow v \preceq s'_\ell \preceq s_{\ell+1}$ (U1) ($\Rightarrow$ I1b). Furthermore, because of $\text{gap}(u) = \infty$ ($A_3(u,k)$), we have gap $=$ gap$'$ $= r^{\rightarrow}_{\ell(u)}(u) = \infty$ due to U3 and R1 ($\Rightarrow$ I3 since $v \preceq s'_\ell$).
– *Subcase 1.2:* $u = s'_\ell \Rightarrow \text{gap}(u) = \infty$ ($A_3(u,k)$) $\Rightarrow w(u,v) \leq \text{gap}' = r^{\rightarrow}_{\ell}(u)$ (Line 7) $\Rightarrow d_P(s'_\ell, v) \leq r^{\rightarrow}_{\ell}(s'_\ell)$ (since $u = s'_\ell$) $\Rightarrow v \preceq s_{\ell+1}$ (U2b) ($\Rightarrow$ I1b). Furthermore, gap $=$ gap$' = r^{\rightarrow}_{\ell}(u) = r^{\rightarrow}_{\ell}(s'_\ell) - d_P(s'_\ell, u)$ (since $u = s'_\ell$) implies I3 since $s'_\ell \prec v$.
– *Subcase 1.3:* $u \succ s'_\ell \Rightarrow \text{gap}(u) = r^{\rightarrow}_{\ell}(s'_\ell) - d_P(s'_\ell, u)$ ($A_3(u,k)$). By Lemma 2, $\ell \leq L$ and $(\ell = L \rightarrow \text{gap} = \infty)$. If $\ell = L$, we have $v \preceq t = s_{L+1} = s_{\ell+1}$ ($\Rightarrow$ I1b) and $\text{gap} = \infty = r^{\rightarrow}_{\ell}(s'_\ell) - d_P(s'_\ell, u)$ (R2) ($\Rightarrow$ I3 since $s'_\ell \prec v$). Subsequently, we deal with the remaining case $\ell < L$. The facts that $u \preceq t$ and $s'_\ell \prec u$ imply $s'_\ell \neq t$, which yields $s'_\ell \in V'_\ell$ due to U4. Hence, due to R3, $\text{gap}(u) \neq \infty \Rightarrow w(u,v) \leq \text{gap}' = \text{gap}(u)$ (Line 7) $\Rightarrow d_P(u,v) \leq r^{\rightarrow}_{\ell}(s'_\ell) - d_P(s'_\ell, u) \Rightarrow d_P(s'_\ell, v) \leq r^{\rightarrow}_{\ell}(s'_\ell) \Rightarrow v \preceq s_{\ell+1}$ (U2b) ($\Rightarrow$ I1b). Furthermore, $\text{gap} = \text{gap}' = \text{gap}(u) = r^{\rightarrow}_{\ell}(s'_\ell) - d_P(s'_\ell, u)$ implies I3 since $s'_\ell \prec v$.

Case 2: at least one iteration of the for-loop takes place ($\ell > \ell(u)$).
We claim that after any iteration of the for-loop, we have $u = s_\ell$. Proof by induction:
Base Case: We consider the first iteration of the for-loop. Line 9 and the fact that an iteration takes place imply $w(u,v) > \text{gap}'$, which means that $\text{gap}' \neq \infty$. We distinguish between two subcases to show that $d_P(s'_{\ell(u)}, v) > r^{\rightarrow}_{\ell(u)}(s'_{\ell(u)})$.

– *Subcase 2.1:* $u \preceq s'_{\ell(u)} \Rightarrow \text{gap}(u) = \infty$ ($A_3(u,k)$) $\Rightarrow w(u,v) > \text{gap}' = r^{\rightarrow}_{\ell(u)}(u)$ (Line 7) $\Rightarrow r^{\rightarrow}_{\ell(u)}(u) \neq \infty$. We have $s_{\ell(u)} \preceq u \preceq s'_{\ell(u)}$ due to L2, $A_2(u,k)$, and the assumption of Subcase 2.1. However, we can exclude that $s_{\ell(u)} \preceq u \prec s'_{\ell(u)}$: this would imply $u \notin V'_{\ell(u)}$ (U3) and, thus, $r^{\rightarrow}_{\ell(u)}(u) = \infty$ (R1). Therefore, $u = s'_{\ell(u)} \Rightarrow d_P(s'_{\ell(u)}, v) > r^{\rightarrow}_{\ell(u)}(s'_{\ell(u)})$
– *Subcase 2.2:* $u \succ s'_{\ell(u)} \Rightarrow s'_{\ell(u)} \neq t \Rightarrow s'_{\ell(u)} \in V'_{\ell(u)}$ (U4). Furthermore, $\text{gap}(u) = r^{\rightarrow}_{\ell(u)}(s'_{\ell(u)}) - d_P(s'_{\ell(u)}, u)$ ($A_3(u,k)$) $\Rightarrow \text{gap}(u) \neq \infty$ (due to R3 since $\ell(u) < L$ (Lemma 2) and $s'_{\ell(u)} \in V'_{\ell(u)}$) $\Rightarrow d_P(u,v) = w(u,v) > \text{gap}' = \text{gap}(u) = r^{\rightarrow}_{\ell(u)}(s'_{\ell(u)}) - d_P(s'_{\ell(u)}, u)$ (Line 7) $\Rightarrow d_P(s'_{\ell(u)}, v) > r^{\rightarrow}_{\ell(u)}(s'_{\ell(u)})$

From $d_P(s'_{\ell(u)}, v) > r^{\rightarrow}_{\ell(u)}(s'_{\ell(u)})$, it follows that $s_{\ell(u)+1} \prec v$ (U2a), which implies $s_{\ell(u)+1} \preceq u$. Hence, $u = s_{\ell(u)+1}$ (since $u \preceq s_{\ell(u)+1}$ due to L3 and $A_2(k, u)$).
Induction Step: Let us now deal with the iteration from level i to level $i+1$ for $i \geq \ell(u)+1$. We have $w(u, v) > \text{gap} = r^{\rightarrow}_i(u)$, which implies $r^{\rightarrow}_i(u) \neq \infty$. Starting with $u = s_i \preceq s'_i \preceq s_{i+1}$ (induction hypothesis, U1), we can conclude that $u = s'_i$ (U3, R1) $\Rightarrow d_P(s'_i, v) > r^{\rightarrow}_i(s'_i) \Rightarrow s_{i+1} \prec v$ (U2a) $\Rightarrow s_{i+1} \preceq u \Rightarrow u = s_{i+1}$ (since $u \preceq s_{i+1}$). This completes our inductive proof.

After the last iteration, we have $u = s_\ell \preceq s'_\ell$ ($\Rightarrow$ I1a). Furthermore, $w(u, v) \leq r^{\rightarrow}_\ell(u)$. If $u \prec s'_\ell$, we obtain $v \preceq s'_\ell \preceq s_{\ell+1}$ ($\Rightarrow$ I1b) and gap $= r^{\rightarrow}_\ell(u) = \infty$ due to U3 and R1 ($\Rightarrow$ I3 since $v \preceq s'_\ell$). Otherwise ($u = s'_\ell$), we get $d_P(s'_\ell, v) \leq r^{\rightarrow}_\ell(s'_\ell)$, which implies $v \preceq s_{\ell+1}$ as well (U2b) ($\Rightarrow$ I1b); furthermore, gap $= r^{\rightarrow}_\ell(u) = r^{\rightarrow}_\ell(s'_\ell) - d_P(s'_\ell, u)$ (since $u = s'_\ell$) implies I3 since $s'_\ell \prec v$. This completes the proof of I1 and I3.

I2 ($\overline{\ell}(v) = \ell$) directly follows from $s_\ell \prec v \preceq s_{\ell+1}$ (I1). □

Lemma 6. *Consider a highway path $P = \langle s, \ldots, t\rangle$ and an edge (u, v) on P such that u precedes the meeting point p. Assume that u has been appropriately* settled*. Then, the edge (u, v) is not skipped, but relaxed.*

Proof. We consider the relaxation of the edge (u, v). Due to Lemma 5, the Invariants I1–I3 apply after Line 9 has been executed. Now, we consider Line 10 of the highway query algorithm.

I1 and M2 imply $s_\ell \preceq u \prec p \preceq s_{\lambda+1}$. Hence, $\ell \leq \lambda$. Thus, $u \prec p \preceq t_\lambda \preceq t_\ell$ (M1). By H2, we obtain $\ell(u, v) \geq \ell$. Therefore, the edge (u, v) is not skipped at this point.

Moreover, we prove that the condition in Line 11 is not fulfilled since (u, v) belongs to a highway path. This means that the edge (u, v) is not skipped at this point, either. We have to show that $u \notin V'_\ell \vee v \notin B_\ell$. We have $s_\ell \preceq u$ (I1). If $u \prec s'_\ell$, we get $u \notin V'_\ell$ (U3). Otherwise, we have $s'_\ell \preceq u \prec v \preceq p \preceq t'_\ell$ (M3), which yields $v \notin B_\ell$ (H1).

Therefore, (u, v) is not skipped, but relaxed. □

Lemma 7. *Consider a shortest path $P = \langle s, \ldots, t\rangle$ and an edge (u, v) on P. Assume that u has been appropriately* settled *with some key k. Furthermore, assume that the edge (u, v) is not skipped, but relaxed. Then, v can be appropriately* reached *from u with key k'.*

Proof. We consider the relaxation of the edge (u, v). Due to Lemma 5, the Invariants I1–I3 apply after Line 9 has been executed. Therefore—since (u, v) is not skipped, but relaxed—, the node v can be reached with the key

$$k' = (\delta'(v), \ell'(v), \text{gap}'(v)) := (\delta(u) + w(u, v), \ell, \text{gap} - w(u, v)).$$

Thus, $A_1(k', v)$, $A_2(k', v)$, and $A_3(k', v)$ hold since P is a shortest path and due to $A_1(k, u)$, I2, and I3.

Consider an arbitrary i such that $t \neq s'_i \preceq v$. To prove $A_4(v)$, we have to show that $v \in V'_i$. Due to U4, this is true for $s'_i = v$. Now, we deal with the remaining case $s'_i \preceq u \prec v$. Since $v \preceq s_{\ell+1} \preceq s'_{\ell+1}$ (I1, U1), we have $i \leq \ell$. The case $\ell = 0$ is trivial; hence, we assume $\ell > 0$. Since the edge (u, v) is not skipped, we know that Restriction 1 does not apply. Therefore, we have $\ell(u, v) \geq \ell$, which implies $v \in V_\ell \subseteq V'_{\ell-1}$. For $i < \ell$, we have $V'_{\ell-1} \subseteq V'_i$ and are done. For $i = \ell$, we have $u \in V'_\ell$ due to $A_4(u)$. This implies $v \notin B_\ell$ since Restriction 2 does not apply as well. $v \in V_\ell$ and $v \notin B_\ell$ yield $v \in V'_\ell$. □

Analogous considerations hold for the backward search.

A.5 Finding an Optimal Path

Source and target nodes s and t are given such that a shortest path from s to t exists.[10]

Definition 4. *A* state z *is a triple* $(P, u, \overline{u})$*, where* P *is a* s*-*t*-path,* $u, \overline{u} \in V \cap P$*, and* $u \preceq \overline{u}$*.*

Definition 5. *A state* $z = (P, u, \overline{u})$ *is* valid *iff all of the following valid State properties are fulfilled:*

- S1: $w(P) = d_0(s, t)$
- S2: $P|_{u \to \overline{u}}$ is contracted, i.e., $P|_{u \to \overline{u}} = \mathrm{ctr}(P|_{u \to \overline{u}})$
- S3: $P|_{s \to u}$ and $P|_{\overline{u} \to t}$ are paths in the forward and backward search tree, respectively.

Lemma 8. *Consider a valid state* $z = (P, u, \overline{u})$ *and an arbitrary node* $x, s \preceq x \preceq u$*, on* P*. Then,* x *has been appropriately settled. Analogously for backward search.*

Proof. Base Case: True for s. *Induction Step:* We assume that $y, s \preceq y \prec u$, has been appropriately settled and show that $x = \mathrm{succ}(y)$ is appropriately settled as well. Since (y, x) belongs to the forward search tree (S3), we know that (y, x) is not skipped, but relaxed. The other prerequisites of Lemma 7 are fulfilled as well (due to the induction hypothesis and S1). Thus, we can conclude that x can be appropriately *reached* from y. Since (y, x) belongs to the forward search tree, we know that x is also *settled* from y. □

Lemma 9. *If* $z = (P, u, \overline{u})$ *is a valid state, then* P *is a highway path.*

Proof. All labels (e.g., s'_ℓ) in this proof refer to P. We show that the highway properties H1 and H2 are fulfilled by induction over the level ℓ.
Base Case: H2(0) trivially holds since $\ell(u, v) \geq 0$ for *any* edge (u, v).
Induction Step (a): H2(ℓ) → H1(ℓ). We assume $s'_\ell \prec t'_\ell$. (Otherwise, H1(ℓ) is trivially fulfilled.) This implies $s'_\ell \neq t$. Consider an arbitrary node x on $P|_{s'_\ell \to t'_\ell}$. We distinguish between three cases.
Case 1: $x \preceq u$. According to Lemma 8, $A_4(x)$ holds. Hence, $x \in V'_\ell$ since $s'_\ell \preceq x$.
Case 2: $u \preceq x \preceq \overline{u}$. We have $y := \max(u, s'_\ell) \in V'_\ell$ (either by Lemma 8: $A_4(u)$ or by U4). Analogously, $\overline{y} := \min(\overline{u}, t'_\ell) \in V'_\ell$. Since $u \preceq y \preceq x \preceq \overline{y} \preceq \overline{u}$ and $P|_{u \to \overline{u}} = \mathrm{ctr}(P|_{u \to \overline{u}})$ (S2), we can conclude that $x \notin B_\ell$. Furthermore, we have $x \in V_\ell$ (due to H2(ℓ)). Thus, $x \in V'_\ell$.
Case 3: $\overline{u} \preceq x$. Analogous to Case 1.
Induction Step (b): H1(ℓ) ∧ H2(ℓ) → H2(ℓ+1). Let $\overline{P}$ denote $\exp(P|_{s'_\ell \to t'_\ell}, \ell)$ and consider an arbitrary edge (x, y) on $\overline{P}$. If (x, y) is part of an expanded shortcut, we have $\ell(x, y) \geq \ell + 1$ and $x, y \in V_{\ell+1} \subseteq V'_\ell$. Otherwise, (x, y) belongs to $P|_{s'_\ell \to t'_\ell}$, which is a subpath of $P|_{s_\ell \to t_\ell}$, which implies $x, y \in V'_\ell$ and $\ell(x, y) \geq \ell$ by H1(ℓ) and H2(ℓ). Thus, in any case, $\ell(x, y) \geq \ell$, $x, y \in V'_\ell$, and (x, y) is not a shortcut of some level $> \ell$. Hence, $\overline{P}$ is a path in G'_ℓ. Now, consider an arbitrary edge (u, v), $s_{\ell+1} \preceq u \prec v \preceq t_{\ell+1}$, on P. If (u, v) is a shortcut of some level $> \ell$, we directly have $\ell(u, v) \geq \ell+1$. Otherwise, (u, v) is on $\overline{P}$ as well. Since $s_{\ell+1} \prec v$, we have $d_P(s'_\ell, v) > r^{\rightarrow}_\ell(s'_\ell)$ (U2b). Moreover, S1 implies that $\overline{P}$ is a shortest path in G'_ℓ and, in particular, $d_{\overline{P}}(s'_\ell, v) = w(\overline{P}|_{s'_\ell \to v}) = d_\ell(s'_\ell, v)$. Using the fact that $d_{\overline{P}}(s'_\ell, v) = d_P(s'_\ell, v)$, we obtain $d_\ell(s'_\ell, v) > r^{\rightarrow}_\ell(s'_\ell)$ and, thus, $v \notin \mathcal{N}^{\rightarrow}_\ell(s'_\ell)$.

Analogously, we have $u \notin \mathcal{N}^{\leftarrow}_\ell(t'_\ell)$. Hence, the definition of the highway network $G_{\ell+1}$ implies $(u, v) \in E_{\ell+1}$. Thus, $\ell(u, v) \geq \ell + 1$. □

Definition 6. *A valid state is either a* final *state (if* $u = \overline{u}$*) or a* non-final *state (otherwise).*

[10] The special case that there is no path from s to t is treated in Section A.1.

We pick any shortest s-t-path P. The state $(\mathrm{ctr}(P), s, t)$ is the *initial* state. Since forward and backward search run completely independently of each other, any serialisation of both search processes will yield exactly the same result. Therefore, in our proof, we are free to pick—w.l.o.g.—any order of forward and backward steps. We assume that at first one forward and one backward iteration is performed, which implies that s and t are settled. At this point, the highway query algorithm is in the initial state. It is easy to see that the initial state is a valid state. Due to the following lemma, it is sufficient to prove that a final state is eventually reached.

Lemma 10. *Getting to a final state is equivalent to finding a shortest s-t-path.*

Proof. $u = \overline{u}$ means that forward and backward search meet. Due to Lemma 8, we can conclude that both u and $\overline{u}$ are settled with the optimal distance (A_1), i.e., $\overrightarrow{\delta}(u) = d_0(s, u)$ and $\overleftarrow{\delta}(\overline{u}) = d_0(\overline{u}, t)$. Since $u = \overline{u}$ lies on a shortest path (due to S1), we have $d(s, t) = d_0(s, u) + d_0(\overline{u}, t)$. Line 6 implies $d' \leq \overrightarrow{\delta}(u) + \overleftarrow{\delta}(\overline{u}) = d(s, t)$. In fact, this means that the algorithm returns $d' = d(s, t)$ since this is already optimal. □

Definition 7. *For a valid state $z = (P, u, \overline{u})$, the forward direction is said to be* blocked *if $p \preceq u$. Analogously, the backward direction is blocked if $\overline{u} \preceq p$.*

Lemma 11. *For a non-final state $z = (P, u, \overline{u})$, at most one direction is blocked.*

Proof. Since z is a non-final state, we have $u \prec \overline{u}$, which implies $u \prec p$ or $p \prec \overline{u}$. □

Definition 8. *The* rank $\rho(z)$ *of a state $z = (P, u, \overline{u})$ is $|\{x \in P \mid u \preceq x \preceq \overline{u}\}|$.*

Lemma 12. *From any non-final state $z = (P, u, \overline{u})$, another valid state z^+ is reached at some point such that $\rho(z^+) < \rho(z)$.*

Proof. We pick any non-blocked direction—due to Lemma 11, we know that there is at least one such direction. Subsequently, we assume that the forward direction was picked; the backward direction can be dealt with analogously.

We have $u \prec p$ and observe that all prerequisites of Lemma 6 are fulfilled due to Lemmas 9 and 8. Hence, we can conclude that the edge $(u, v := \mathrm{succ}(u))$ is not skipped, but relaxed. Thus, since P is a shortest path (S1), v can be reached with the optimal distance due to Lemma 7 (A_1). The fact that the algorithm terminates (Lemma 3) implies that the queue $\overrightarrow{Q}$ gets empty at some point, i.e., every element has been deleted from $\overrightarrow{Q}$. In particular, we can conclude that v is deleted at some point. Since v has been reached with the optimal distance, it will also be settled with the optimal distance (due to the specification of the decreaseKey operation, which guarantees that tentative distances are never increased). Let P' denote the path from s to v in the forward search tree. We set $z^+ := (P^+ := P' \circ P|_{v \to t}, v, \overline{u})$. We have $w(P^+) = w(P') + w(P|_{v \to t}) = d_0(s, v) + d_0(v, t) = d_0(s, t)$ ($\Rightarrow$ S1). S2 is fulfilled since $P^+|_{v \to \overline{u}}$ is a subpath of $P|_{u \to \overline{u}}$. S3 holds due to the construction of P^+. Hence, z^+ is valid. Furthermore, $\rho(z^+) = \rho(z) - 1$. □

Theorem 5. *The highway query algorithm finds a shortest s-t-path.*

Proof. From Lemma 12 and the fact that the codomain of the rank function is finite, it follows that eventually a final state is reached, which is equivalent to finding a shortest s-t-path due to Lemma 10. □

A.6 Distance Table Optimisation

To prove the correctness of the distance table optimisation, we introduce the following new lemma and adapt a few definitions and proofs from Section A.5 to the new situation.

Lemma 13. *Consider a valid state $z = (P, u, \overline{u})$ with $u \prec s'_L$. When u's edges are relaxed, neither the condition in Line 7a nor the condition in Line 11a is fulfilled.*

Proof. Due to Lemma 8, u has been appropriately settled with some key k. We distinguish between two cases.
Case 1: $u \prec s_L$. From $s_{\ell(u)} = s_{\overline{\ell}(u)} \preceq u \prec s_L$ ($A_2(k, u)$, L2), it follows that $\ell(u) < L$ (U1). Hence, the condition in Line 7a is not fulfilled. Furthermore, we have $s_\ell \preceq u \prec s_L$ after Line 9 has been executed (Lemma 5: I1). Thus, $\ell < L$, which implies that the condition in Line 11a is not fulfilled as well.
Case 2: $s_L \preceq u \prec s'_L$. First, we show that the condition in Line 7a is not fulfilled. We assume $\ell(u) = L$. (Otherwise, the condition cannot be fulfilled.) Due to $A_3(k, u)$, we have $\text{gap}(u) = \infty$. Hence, $\text{gap}' = r^{\rightarrow}_{\ell(u)}(u) = r^{\rightarrow}_L(u) = \infty_1$ by R1 since $u \notin V'_L$ (U3). Now, we prove that the condition in Line 11a is not fulfilled. We assume $\ell = L \wedge \ell > \ell(u)$. (Otherwise, the condition cannot be fulfilled.) Due to Line 9, we get $\text{gap} = r^{\rightarrow}_\ell(u) = r^{\rightarrow}_L(u) = \infty_1$ (as above). □

Definition 6'. *A valid state is either a* final *state (if $u = \overline{u}$ or $s'_L \preceq u \wedge \overline{u} \preceq t'_L$) or a* non-final *state (otherwise).*

Lemma 10. *Getting to a final state is equivalent to finding a shortest s-t-path.*

Proof. In the proof of this lemma in Section A.5, we have already dealt with the case $u = \overline{u}$. Now, consider the new case $u \prec \overline{u} \wedge s'_L \preceq u \wedge \overline{u} \preceq t'_L$. We show that s'_L is added to the set $\overrightarrow{I}$. Since $s'_L \preceq u$, s'_L has been appropriately settled with some key k (due to Lemma 8). We consider the attempt to relax the edge $(s'_L, v := \text{succ}(s'_L))$ and distinguish between two cases.
Case 1: $s_L = s'_L$. $\ell = \overline{\ell}(v)$ (I2), $s_L = s'_L \prec v$, and $\overline{\ell}(v) \leq L$ (L1) imply $\ell = \overline{\ell}(v) = L$. Furthermore, $A_2(k, s'_L)$ and the assumption of Case 1 yield $\ell(s'_L) = \overline{\ell}(s'_L) < L = \ell$. In addition, $\text{gap} = \infty_2 \neq \infty_1$ by I3 (since $s'_\ell \prec v$), the fact that $s'_L \in V'_L$ (U4), and R2. Hence, the condition in Line 11a is fulfilled so that s'_L is added to $\overrightarrow{I}$.
Case 2: $s_L \prec s'_L$. By $A_2(k, s'_L)$, $A_3(k, s'_L)$, the assumption of Case 2, and $\overline{\ell}(s'_L) \leq L$ (L1), we get $\ell(s'_L) = \overline{\ell}(s'_L) = L$ and $\text{gap}(s'_L) = \infty$. Thus, $\text{gap}' = r^{\rightarrow}_L(s'_L) = \infty_2 \neq \infty_1$ (R2). Hence, the condition in Line 7a is fulfilled so that s'_L is added to $\overrightarrow{I}$.

Analogously, we can prove that t'_L is added to the set $\overleftarrow{I}$. Since P is a highway path (due to Lemma 9), the subpath $P|_{s'_L \to t'_L}$ is a path in G'_L and, thus, $d_0(s'_L, t'_L) = d_L(s'_L, t'_L)$. Hence, $w(P) = d_0(s, s'_L) + d_L(s'_L, t'_L) + d_0(t'_L, t)$ is the length of a shortest s-t-path and, since the algorithm finds a path with a length $\leq \overrightarrow{\delta}(s'_L) + d_L(s'_L, t'_L) + \overleftarrow{\delta}(t'_L)$ and since $\overrightarrow{\delta}(s'_L) = d_0(s, s'_L)$ and $\overleftarrow{\delta}(t'_L) = d_0(t'_L, t)$ (due to Lemma 8: A_1), we can conclude that a shortest s-t-path is found. □

Definition 7'. *For a valid state $z = (P, u, \overline{u})$, the forward direction is said to be* blocked *if $p \preceq u$ or $s'_L \preceq u$. Analogously, the backward direction is blocked if $\overline{u} \preceq p$ or $\overline{u} \preceq t'_L$.*

Lemma 11. *For a non-final state $z = (P, u, \overline{u})$, at most one direction is blocked.*

Proof. Since z is a non-final state, we have $u \prec \overline{u}$ and $(u \prec s'_L \vee t'_L \prec \overline{u})$. To obtain a contradiction, let us assume that both directions are blocked, i.e., ($p \preceq u$ or $s'_L \preceq u$) and

($\overline{u} \preceq p$ or $\overline{u} \preceq t'_L$). Consider the case $p \preceq u$ and $\overline{u} \preceq t'_L$. Hence, $p \preceq u \prec \overline{u} \preceq t'_L$. Due to M3, we can conclude that $s'_L \preceq p \preceq u$. Since $s'_L \preceq u$ and $\overline{u} \preceq t'_L$, we have a contradiction. The remaining three cases are analogous or straightforward. □

Lemma 12. *From any non-final state $z = (P, u, \overline{u})$, another valid state z^+ is reached at some point such that $\rho(z^+) < \rho(z)$.*

Proof. The proof of this lemma in Section A.5 still works since the added two lines (7a and 11a) have no effect due to Definition 7' and Lemma 13. □

B Implementation

The graph is 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 the level data that is specific to highway hierarchies. In addition to the index of the first outgoing edge, each node u stores its level-0 neighbourhood radius $r_0(u)$. Moreover, for each node u, all outgoing edges (u, v) are grouped by their level $\ell(u, v)$. Between the node and the edge array, we insert another layer: for each node u and each level $\ell > 0$ that u belongs to, there is a *level node* u_ℓ that stores the radius $r_\ell(u)$ and the index of the first outgoing edge (u, v) in level ℓ. All level nodes are stored in a single array. Each node u keeps the index of the level node u_1. Figure 16 illustrates the graph representation.

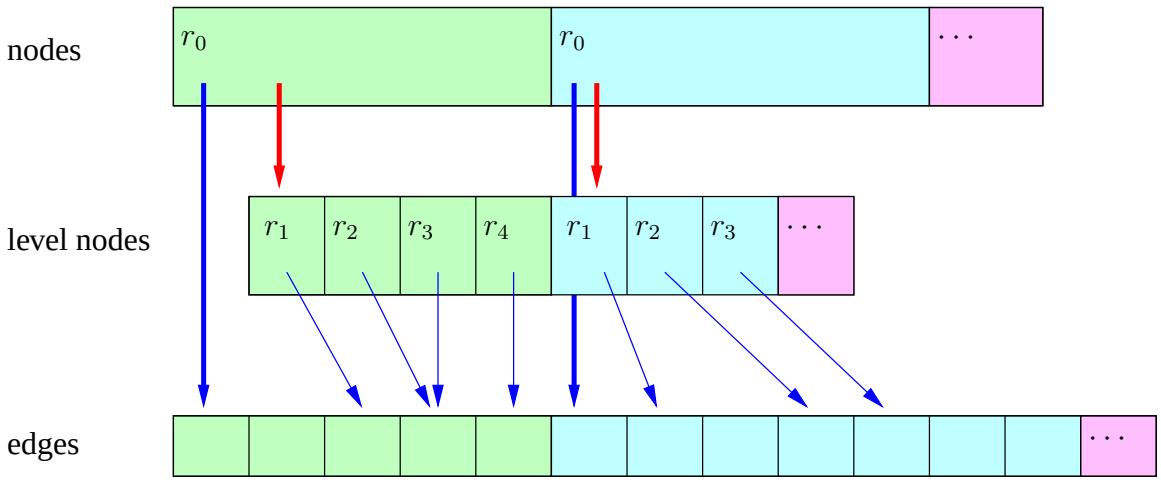

Fig. 16. An adjacency array, extended by a level-node layer.

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 and the fast preprocessing algorithm is checked against a naive implementation.

C Experiments

In addition to the experiments presented in Section 6.3, we have considered many more combinations of neighbourhood size, contraction rate, and number of levels. The results are given in Table 5 and 6.

Table 5. Preprocessing and query performance for the European road network depending on the contraction rate c and the neighbourhood size H. We do not use a distance table, but repeat the construction process until the topmost level is empty or the hierarchy consists of 15 levels.

contr. rate c	nbh. size H	PREPROCESSING time [min]	∅overhead/ node [byte]	∅deg.	QUERY time [ms]	#settled nodes	#relaxed edges
0.5	30	83	30	3.2	391.73	472 326	1 023 944
	40	83	28	3.2	267.57	334 287	711 082
	50	87	27	3.2	188.55	242 787	506 543
	60	86	27	3.2	135.27	177 558	362 748
	70	87	26	3.2	101.36	135 560	271 324
	80	89	26	3.1	73.40	99 857	196 150
	90	87	25	3.1	55.02	75 969	146 247
1.0	30	15	28	3.7	5.48	6 396	23 612
	40	15	28	3.7	2.62	3 033	11 315
	50	17	27	3.6	2.13	2 406	8 902
	60	18	27	3.6	1.93	2 201	8 001
	70	19	26	3.6	1.80	2 151	7 474
	80	20	26	3.6	1.79	2 193	7 392
	90	22	26	3.6	1.78	2 221	7 268
1.5	30	11	28	3.8	1.93	1 830	9 281
	40	12	28	3.8	1.72	1 628	7 672
	50	13	27	3.7	1.56	1 593	6 975
	60	14	27	3.7	1.53	1 645	6 697
	70	15	27	3.7	1.51	1 673	6 590
	80	17	27	3.7	1.51	1 726	6 719
	90	18	27	3.7	1.54	1 782	6 655
2.0	30	11	29	4.0	1.85	1 542	8 913
	40	11	29	3.9	1.64	1 475	7 646
	50	12	28	3.9	1.48	1 470	6 785
	60	14	28	3.8	1.46	1 506	6 650
	70	15	28	3.8	1.45	1 547	6 649
	80	16	27	3.8	1.49	1 611	6 935
	90	17	27	3.8	1.53	1 675	6 988
2.5	30	11	30	4.1	1.96	1 489	9 175
	40	11	29	4.0	1.70	1 453	7 822
	50	12	29	4.0	1.58	1 467	7 119
	60	14	29	3.9	1.57	1 493	7 035
	70	15	28	3.9	1.54	1 536	6 905
	80	16	28	3.9	1.55	1 583	7 094
	90	18	28	3.9	1.58	1 645	7 204

Table 6. Preprocessing and query performance for the European road network depending on the number of levels and the neighbourhood size H. In the topmost level, a distance table is used.

#levels	nbh. size H	PREPROCESSING time [min]	∅overhead/ node [byte]	∅deg.	QUERY time [ms]	#settled nodes	#relaxed edges
5	40	14	60	3.9	0.67	691	2 398
	50	13	40	3.9	0.77	818	2 892
	60	14	32	3.8	0.87	938	3 361
	70	15	30	3.8	0.96	1 058	3 837
	80	16	28	3.8	1.05	1 165	4 278
	90	17	28	3.8	1.13	1 269	4 697
6	30	12	48	4.0	0.75	709	2 531
	40	11	33	3.9	0.87	867	3 171
	50	12	29	3.9	0.99	1 015	3 759
	60	13	28	3.8	1.10	1 157	4 299
	70	15	28	3.8	1.21	1 292	4 837
	80	16	28	3.8	1.30	1 414	5 311
	90	17	27	3.8	1.40	1 521	5 817
7	30	10	34	4.0	0.93	852	3 195
	40	11	29	3.9	1.07	1 025	3 894
	50	12	28	3.9	1.20	1 187	4 538
	60	13	28	3.8	1.32	1 344	5 166
	70	15	28	3.8	1.39	1 462	5 689
	80	16	27	3.8	1.47	1 578	6 179
	90	18	27	3.8	1.53	1 668	6 661
8	30	10	30	4.0	1.14	991	3 853
	40	11	29	3.9	1.27	1 171	4 624
	50	12	28	3.9	1.36	1 321	5 283
	60	14	28	3.8	1.43	1 455	5 887
	70	15	28	3.8	1.46	1 546	6 338
	80	16	27	3.8	1.48	1 611	6 935
	90	18	27	3.8	1.53	1 675	6 988
9	30	10	30	4.0	1.35	1 123	4 532
	40	11	29	3.9	1.45	1 289	5 338
	50	12	28	3.9	1.48	1 417	5 931
	60	14	28	3.8	1.47	1 506	6 429
	70	15	28	3.8	1.46	1 547	6 649
10	30	10	29	4.0	1.54	1 241	5 214
	40	11	29	3.9	1.57	1 380	6 012
	50	12	28	3.9	1.51	1 468	6 470
	60	14	28	3.8	1.46	1 506	6 650
11	30	10	29	4.0	1.67	1 326	5 847
	40	11	29	3.9	1.65	1 445	6 627
	50	13	28	3.9	1.49	1 470	6 785