Engineering Highway Hierarchies*

Peter Sanders and Dominik Schultes

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

Abstract. In [1], we presented a shortest path algorithm that allows fast point-to-point queries in graphs using preprocessed data. Here, we give an extensive revision of our method. It allows faster query and preprocessing times, it reduces the size of the data obtained during the preprocessing and it deals with directed graphs. Some important concepts like the *neighbourhood radii* and the *contraction of a network* have been generalised and are now more flexible. The query algorithm has been simplified: it differs only by a few lines from the bidirectional version of DIJKSTRA's algorithm. We can prove that our algorithm is correct even if the graph contains several paths of the same length.

Experiments with real-world road networks confirm the effectiveness of our approach. Preprocessing the network of Western Europe, which consists of about 18 million nodes, takes 15 minutes and yields 68 bytes of additional data per node. Then, random queries take 0.76 ms on average. If we are willing to accept slower query times (1.38 ms), the memory usage can be decreased to 17 bytes per node. For the European and the US road networks, we can guarantee that at most 0.05% of all nodes are visited during any query.

1 Introduction

Computing fastest routes in road networks is one of the showpieces of real-world applications of algorithmics. In principle we could use DIJKSTRA's algorithm. But for large road networks this would be far too slow. Therefore, there is considerable interest in speedup techniques for route planning. Commercial systems use information on road categories to speed up search. "Sufficiently far away" from source and target, only "important" roads are used. This requires manual tuning of the data and a delicate tradeoff between computation speed and suboptimality of the computed routes. In a previous paper [1] we introduced the idea to *automatically* compute *highway hierarchies* that yield *optimal* routes *uncompromisingly quickly*. This was the first speedup technique that was able to preprocess the road network of a continent in realistic time and obtain large speedups (several thousands) over DIJKSTRA's algorithm. Since this was a prototype, we made several simplifying assumptions. Our system was limited to undirected graphs, we only had a proof for a simplified version of the query algorithm, practitioners criticised the considerable constant factor in space consumption, and the query algorithm was fairly complicated.

* Partially supported by DFG grant SA 933/1-3.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 804–816, 2006.
© Springer-Verlag Berlin Heidelberg 2006

In this paper we tackle all these issues. We originally thought that this would be a more or less routine case study in algorithm engineering. However, we arrived at some algorithmically interesting new and more general concepts and we obtained results we did not expect. In particular, our system became at the same time considerably simpler, more space efficient, and faster with respect to both preprocessing and query time.

Our Contributions. Perhaps the most crucial definition for highway hierarchies is a specification of the concept of *local search*. Section 3 allows directed graphs and an individual neighbourhood radius for each node. The highway network—a set of edges that suffice for all shortest paths outside of local neighbourhoods—can then still be computed using methods analogous to [1]. Since the highway network is very sparse, it is then important to *contract* it by removing nodes of small degree. In [1] this was done using specialised routines for *attached trees* and *lines* of nodes with degree two. First experiments indicated that a straight forward adaptation of these concepts to directed graphs leads to deteriorating performance. In Section 4 we describe a simpler, more general method that leads to *better* performance even for undirected graphs: A node v can be *bypassed* by replacing all edge pairs of the form $(u, v), (v, w)$ with $u \neq w$ by a *shortcut* (u, w). For a tuning parameter c, if the number of introduced shortcuts is smaller than c times the number of removed edges adjacent to v, the node is actually bypassed.

Section 5 describes a simple query algorithm for directed highway hierarchies. Its pseudocode is only four lines longer than code for ordinary bidirectional Dijkstra. Since highway hierarchies are additionally similar to the heuristic hierarchies used in industry, we are very optimistic that they are easy to use in products. Moreover, the simplified algorithm also allows us to give a complete correctness proof.

Section 6 deals with the abort criterion that can be applied when forward and backward search have met. In contrast to the undirected prototype from [1], we *drop* optimisations intended for pruning the search space. While this inflates the search space by about 50%, our measurements indicate that the net effect on the running time is an improvement by more than 50%. Furthermore, we describe how an additional acceleration can be obtained by computing a complete distance table for the topmost level of the highway hierarchy.

The experiments in Section 7 give a strong indication that directed highway hierarchies are currently the most efficient technique for route planning. The tuning parameters turn out to work uniformly well for all the inputs or at least suboptimal values only lead to small performance degradations. Highway hierarchies also allow per-instance worst case performance guarantees, i.e., we can give a good approximation of the complete query time distribution including the worst case for all n^2 possible query pairs without actually executing this astronomic number of queries.

Related Work. For a detailed review of practical and theoretical speedup techniques we refer to [2, 3, 4]. Here we restrict ourselves to the latest news and the concepts needed to understand the problems at hand. A classical technique is *bidirectional search*, which simultaneously searches forward from s and back-

wards from t until the search frontiers meet. For the remaining techniques, we distinguish between two basic speedup effects. Some techniques direct the search towards the target node (and backward search towards the source node), other approaches exploit the *hierarchy* inherent in road networks, and some incorporate both effects by storing information about nodes reached by shortest paths via some edge. Besides highway hierarchies, the most effective hierarchy based technique is *reach based routing* [5] which was considerably strengthened in [6]. Interestingly, the methods used to efficiently compute highway hierarchies in [1] also turned out to be crucial for computing reaches. Reach based routing combined with the strong sense of goal direction from the *landmark method* (the REAL algorithm) beats [1] with respect to query time whereas it needs significantly more preprocessing time. Our new results achieve better query times, a factor ≥ 26 smaller preprocessing times, and need less space.

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. We assume w.l.o.g. that there are no self-loops, parallel edges, and zero weight edges in the input—they could be dealt with easily in a preprocessing step. 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. 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'.

Dijkstra's Algorithm. Dijkstra's algorithm 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, 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 either *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. Nodes that are not reached are *unreached*.

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 provide an inductive definition:

- 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 a nonnegative *neighbourhood radius* $r^{\rightarrow}_\ell(u)$ for the forward graph and a radius $r^{\leftarrow}_\ell(u) \ge 0$ for the backward graph. To avoid some case distinctions, for any direction $\leftrightharpoons \in \{\rightarrow, \leftarrow\}$, we set the neighbourhood radius $r^{\leftrightharpoons}_\ell(u)$ to infinity for $u \notin V'_\ell$ and for $\ell = L$.

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 the subgraph of G'_ℓ induced by the edge set $E_{\ell+1}$: 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 the canonical shortest path[1] $\langle s, \ldots, u, v, \ldots, t \rangle$ from s to t in G'_ℓ with the property that $v \notin \mathcal{N}^{\rightarrow}_\ell(s)$ and $u \notin \mathcal{N}^{\leftarrow}_\ell(t)$.

*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 $E'_\ell := (E_\ell \cap (V'_\ell \times V'_\ell)) \cup S_\ell$.

4 Construction

Neighbourhood Radii. 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 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 in the undirected graph, all nodes are settled in a fixed order. The *Dijkstra rank* $\mathrm{rk}_u(v)$ of a node v is the rank of v w.r.t. this order. u has DIJKSTRA rank $\mathrm{rk}_u(u) = 0$, the closest neighbour v_1 of u

[1] For each connected node pair (s,t), we select a unique *canonical shortest path* in such a way that each subpath of a canonical shortest path is canonical as well. For details, we refer to [1].

has DIJKSTRA rank $\text{rk}_u(v_1) = 1$, and so on. 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 $\text{rk}_u(v)$ is H_ℓ; $d^{\leftrightarrow}_\ell(u, v)$ denotes the distance between u and v in the undirected graph. Applying this strategy to the forward and backward graph one after the other in order to define individual forward and backward radii yields a similar good performance, but needs twice the memory.

Fast Construction of a Highway Network. The fast construction method introduced in [1] has been modified in order to deal with directed graphs and the new, more general neighbourhood definition. For details, we refer to the full paper.

Contraction of a Graph. 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. It happens that shortcuts that were created once are discarded later when one of its endpoints is bypassed. Note that we will get a contraction that is similar to our trees-and-lines method [1] if we set $c = 0.5$.

5 Query

Our *highway query algorithm* is a modification of the bidirectional version of DIJKSTRA's algorithm. For now, we assume that the search is *not* aborted when both search scopes meet. This matter is dealt with in Section 6. We only describe the modifications of the forward search since forward and backward search are

symmetric. In addition to the *distance* from the source, the key of each node includes 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[2]. If the relaxation is successful, 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, as soon as the level-ℓ core is entered, i.e., a node $u \in V'_\ell$ is settled, u is assigned the gap to the border of the level-ℓ neighbourhood of u. Note that before the core is entered, the gap has been infinity. 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.

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.

The Algorithm. We use two priority queues $\overrightarrow{Q}$ and $\overleftarrow{Q}$, one for the forward search and one for the backward search. The key of a node u is a triple $(\delta(u), \ell(u), \text{gap}(u))$, the (tentative) distance $\delta(u)$ from s (or t) to u, the search level $\ell(u)$, and the gap $\text{gap}(u)$ to the next applicable neighbourhood border. A key $(\delta, \ell, \text{gap})$ is less than another key $(\delta', \ell', \text{gap}')$ iff $\delta < \delta'$ or $\delta = \delta' \wedge \ell > \ell'$ or $\delta = \delta' \wedge \ell = \ell' \wedge \text{gap} < \text{gap}'$. Figure 1 contains the pseudo-code of the highway query algorithm. The proof of correctness is included in the full paper.

6 Optimisations

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. We therefore 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

[2] To *relax* an edge means to execute Line 11 in Fig. 1.

```
input: source node s and target node t

[1]  →Q.insert(s, (0, 0, r_0^→(s)));  ←Q.insert(t, (0, 0, r_0^←(t)));
 2   while (→Q ∪ ←Q ≠ ∅) do {
 3       ⇆ ∈ {→, ←};                                          // select direction
 4       u := ⇆Q.deleteMin();
[5]      if gap(u) ≠ ∞ then gap′ := gap(u) else gap′ := r^⇆_ℓ(u)(u);
 6       foreach e = (u, v) ∈ ⇆E do {
[7]          for (ℓ := ℓ(u), gap := gap′;   w(e) > gap;   ℓ++)
                 gap := r^⇆_ℓ+1(u);                           // go "upwards"
[8]          if ℓ(e) < ℓ then continue;                       // Restriction 1
[9]          if u ∈ V′_ℓ ∧ v ∈ B_ℓ then continue;             // Restriction 2
[10]         k := (δ(u) + w(e), ℓ, gap − w(e));
11           if v ∈ ⇆Q then ⇆Q.decreaseKey(v, k); else ⇆Q.insert(v, k);
12       }
13   }
```

Fig. 1. 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.

minimum element u of the forward (backward) queue has a key $\delta(u) \geq w(P')$. More sophisticated rules used in [1] turned out to be too expensive in terms of query time.

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 V'_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 a set $\overrightarrow{I}$ or $\overleftarrow{I}$ depending on the search direction; we do not relax the edge that leads to level L. After the sets $\overrightarrow{I}$ and $\overleftarrow{I}$ have been determined, we consider all pairs $(u,v), u \in \overrightarrow{I}, v \in \overleftarrow{I}$, and compute the minimum path length $D := d_0(s,u) + d_L(u,v) + d_0(v,t)$. Then, the length of the shortest s-t-path is the minimum of D and the length of the tentative shortest path found so far (in case that the search scopes have already met in a level $< L$). This optimisation can be included in the highway query algorithm (Fig. 1) by adding two lines:

between Lines 5 and 6:
5a **if** $\text{gap}' \neq \infty \wedge \ell(u) = L$ **then** $\{\overleftrightarrows{I} := \overleftrightarrows{I} \cup \{u\}$; **continue**;$\}$

between Lines 9 and 10:
9a **if** $\text{gap} \neq \infty \wedge \ell = L \wedge \ell > \ell(u)$ **then** $\{\overleftrightarrows{I} := \overleftrightarrows{I} \cup \{u\}$; **continue**;$\}$

7 Experiments

Environment and Instances. The experiments were done on one core of an AMD Opteron Processor 270 clocked at 2.0 GHz with 4 GB main memory and 2 $\times$ 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 use 32 bits to store edge weights and path lengths.

We deal with the road networks of Western Europe[3] 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, compute for each edge the average travel time, and use it as weight. In order to compare ourselves with [1, 6], we also perform experiments on another version of the US road network (without Alaska and Hawaii) that was obtained from the TIGER/Line Files [7]. However, in contrast to the PTV data, the TIGER graph is undirected, planarised and distinguishes only between four road categories.

As in [1, 6], we report only the times needed to compute the shortest path distance between two nodes without outputting the actual route. In order to obtain the corresponding subpaths in the original graph, we are able to extract the used shortcuts without using any extra data. However, if a fast output routine is required, we might want to spend some additional space to accelerate the unpacking process. For details, we refer to the full paper. Table 1 summarises the properties of the used road networks and key results of the experiments.

Parameters. Unless otherwise stated, the following default settings apply. We use the contraction rate $c = 1.5$ and the neighbourhood sizes H as stated in Tab. 1—the same neighbourhood size is used for all levels of a hierarchy. First, we contract the original graph. Then, we perform four iterations of our construction procedure, which determines a highway network and its core. Finally, we compute the distance table between all level-4 core nodes.

In one test series (Fig. 2), we used all the default settings except for the neighbourhood size H, which we varied from 40 to 90. On the one hand, if H is too small, the shrinking of the highway networks is less effective so that the level-4 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 we get good space-time tradeoffs for neighbourhood sizes around 60. In particular, we find that a good choice of the parameter H does not vary very much from graph to graph.

In another test series (Tab. 2a), we did not use a distance table, but repeated the construction process until the topmost level was empty or the hierarchy

[3] 14 countries: at, be, ch, de, dk, es, fr, it, lu, nl, no, pt, se, uk.

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). The amount of memory needed to store the original graph is not included. 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	50	60	60
PREPROC.	CPU time [min]	15	20	18
	∅overhead/node [byte]	68	69	50
QUERY	CPU time [ms]	0.76	0.90	0.88
	#settled nodes	884	951	1 076
	#relaxed edges	3 182	3 630	4 638
	speedup (CPU time)	8 320	7 232	7 642
	speedup (#settled nodes)	10 196	9 840	11 080
	worst case (#settled nodes)	8 543	3 561	5 141

consisted of 15 levels. We varied the contraction rate c from 0.5 to 2. In case of $c = 0.5$ (and $H = 50$), the shrinking of the highway networks does not work properly so that the topmost level is still very big. This yields huge query times. Note that in [1] we used a larger neighbourhood size to cope with this problem. 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 increased since more shortcuts are introduced. Adding too many shortcuts ($c = 2$) further reduces the search space, but the number of relaxed edges increases so that the query times get worse.

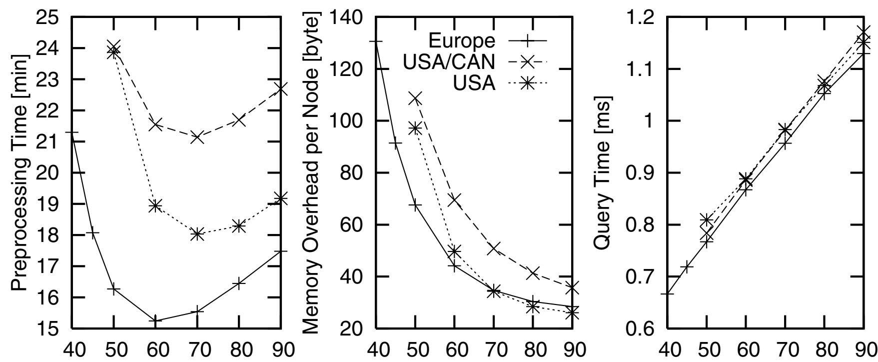

Fig. 2. Preprocessing and query performance depending on the neighbourhood size H

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	89	27	3.2	176.05	242 156	505 086
1	16	27	3.7	1.97	2 321	8 931
1.5	13	27	3.8	1.58	1 704	7 935
2	13	28	3.9	1.70	1 681	8 607

(a)

# levels	PREPROC. time [min]	over-head	QUERY time [ms]	#settled nodes
5	16	68	0.77	884
7	13	28	1.19	1 290
9	13	27	1.51	1 574
11	13	27	1.62	1 694

(b)

In a third test series (Tab. 2b), we used the default settings except for the number of levels, which we varied from 5 to 11. 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 five levels yields a rather large distance table, which somewhat slows down the preprocessing and increases the memory usage. However, in terms of query times, '5 levels' is the optimal choice since using the distance table is faster than continuing the search in higher levels.

Space Saving. If we omit the first contraction step and use a smaller contraction rate ($\rightsquigarrow$ less shortcuts), use a bigger neighbourhood size ($\rightsquigarrow$ higher levels get smaller), and construct more levels before the distance table is used ($\rightsquigarrow$ 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 1.38 ms, respectively.

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 spaces of the forward and backward search, respectively. We can combine both distributions to get an upper bound for the distribution of the search spaces 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 3 visualises the distribution $\mathcal{F}_{\leftrightarrow}(z)$ as a histogram.

For the European road network, we observe several outliers between 7 800 and 8 600. The investigation of some samples indicates that these outliers are situated on some islands next to the Norwegian coast. Since Norway is sparsely populated and the road network is very sparse as well, we know that the neighbourhoods in

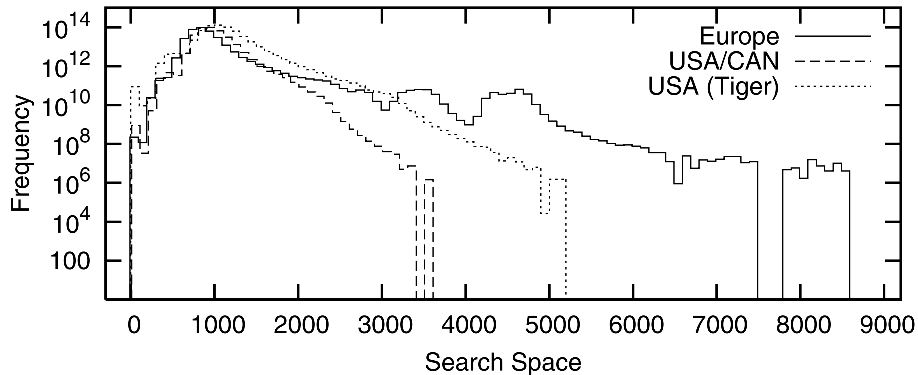

Fig. 3. Histogram of upper bounds for the search spaces of s-t-queries. To increase readability, only the outline of the histogram is plotted instead of the complete boxes.

low levels, which are defined by a fixed number of road junctions, cover a large geographic area. Hence, the search spreads very far before entering a reasonably high search level. When several densely populated areas are encountered while the search level is still quite low, the total search space size gets comparatively large. To improve the worst case, it might be a good idea to introduce adaptive neighbourhood sizes instead of fixed ones so that the above mentioned effect can be avoided.

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 (2 874 in case of Europe), the worst case can get quite large (62 250). For example, accessing 62 250 entries in a table of size $13\ 465 \times 13\ 465$ takes about 1 ms, where 13 465 is the size of the level-4 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 measure would have only little effect on the average performance.

Comparisons. In Tab. 3, we compare several variants of our HH algorithm with the REAL algorithm, which is the method from [6] that features the best query times. Experimental results for the USA (Tiger) graph are taken from [6]. Results for the European graph have been provided by Andrew Goldberg.[4]

[4] These latter results have to viewed as tentative since the networks of Europe and North America are different (long distance ferries,. . .) and this was the very first attempt to run REAL on the European network. It is likely that future versions of REAL will yield better results.

Table 3. Comparison between the REAL algorithm [6] and our highway hierarchies. In addition to the current version of the highway hierarchies with the default settings (HH), we provide results that have been obtained using settings that reduce the memory usage (HH mem). Furthermore, we give old values (HH old) from [1]. Note that the *disk space* includes the memory that is needed to store the original graph.

	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 old	161	892	7.4	4 065	255	1 171	7.04	3 912
REAL	1 625	1 746	2.8	1 867	459	2 392	1.84	891
HH	15	1 570	0.8	884	18	1 686	0.88	1 076
HH mem	55	692	1.4	1 976	65	919	1.60	2 217

Note that the CPU times cannot be compared directly since the implementation of the REAL algorithm was executed on an AMD Opteron running at 2.4 GHz, while our machine only runs at 2.0 GHz. We also have to be careful when we compare the memory usage. In [6] a translation table is created that can be used to unpack shortcuts. We have subtracted the size of the translation tables from the disk spaces used by the REAL algorithm in order to account for the fact that our numbers do not include space for such a table.

Compared to the old highway hierarchies we see big improvements. An order of magnitude reduction in both preprocessing and query time. The main difference between HH and REAL is the dramatically smaller preprocessing time of HH. We see a factor 26 for the USA and a factor 108 for Europe. HH queries are also significantly faster than REAL. Only for the Tiger graph, REAL has a smaller search space. All variants of HH need less space than REAL. The main reason is the overhead for storing distances to landmarks.

8 Discussion

Highway hierarchies are a simple, robust and space efficient concept that allows very efficient fastest path queries even in huge realistic road networks. No other technique has reported such short query times although highway hierarchies have not yet been combined with goal directed search and although none of the previous techniques is competitive w.r.t. preprocessing time. Real-world applications suggest a number of additional challenging problems: How to handle mobile devices with limited fast memory? How to update or patch the hierarchy when edge weights change, e.g. due to traffic jams? What about multiple objective functions or time dependent edge weights? We are optimistic that several of these problems can be solved, in particular because plain highway hierarchies are so fast that even a 100 fold slow-down w.r.t. query time or preprocessing time would be acceptable in some situations.

References

1. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: 13th European Symposium on Algorithms (ESA). Volume 3669 of LNCS., Springer (2005) 568–579
2. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms. (2005) 156–165
3. Willhalm, T.: Engineering Shortest Path and Layout Algorithms for Large Graphs. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik (2005)
4. Schultes, D.: Fast and exact shortest path queries using highway hierarchies. Master's thesis, Universität des Saarlandes (2005)
5. 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)
6. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A^*: Efficient point-to-point shortest path algorithms. In: Workshop on Algorithm Engineering & Experiments. (2006)
7. U.S. Census Bureau, Washington, DC: UA Census 2000 TIGER/Line Files. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html` (2002)