Highway Hierarchies Hasten Exact Shortest Path Queries

Peter Sanders[1,⋆] and Dominik Schultes[1,2]

[1] Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
sanders@ira.uka.de
[2] Universität des Saarlandes
mail@dominik-schultes.de

Abstract. We present a new speedup technique for route planning that exploits the hierarchy inherent in real world road networks. Our algorithm preprocesses the eight digit number of nodes needed for maps of the USA or Western Europe in a few hours using linear space. Shortest (i.e. fastest) path queries then take around eight milliseconds to produce exact shortest paths. This is about 2 000 times faster than using Dijkstra's algorithm.

1 Introduction

Computing shortest paths in graphs (networks) with nonnegative edge weights is a classical problem of computer science. From a worst case perspective, the problem has largely been solved by Dijkstra in 1959 [1] who gave an algorithm that finds all shortest paths from a starting node s using at most $m+n$ priority queue operations for a graph $G=(V,E)$ with n nodes and m edges.

However, motivated by important applications (e.g., in transportation networks), there has recently been considerable interest in the problem of accelerating *shortest path queries*, i.e., the problem to find a shortest path between a source node s and a target node t. In this case, Dijkstra's algorithm can stop as soon as the shortest path to t is found.

A classical technique that gives a constant factor speedup is *bidirectional search* which simultaneously searches forward from s and backwards from t until the search frontiers meet. All further speedup techniques either need additional information (e.g., geometry information for *goal directed search*) or *precomputation*. There is a trade-off between the time needed for *precomputation*, the *space* needed for storing the precomputed information, and the resulting *query time*. In Section 1 we review existing precomputation approaches, which have made significant progress, but still fall short of allowing fast *exact* shortest path queries in very large graphs.

In particular, from now on we focus on shortest paths in large *road networks* where we use 'shortest' as a synomym for 'fastest'. The graphs used for North

⋆ Partially supported by DFG grant SA 933/1-2.

G.S. Brodal and S. Leonardi (Eds.): ESA 2005, LNCS 3669, pp. 568–579, 2005.
© Springer-Verlag Berlin Heidelberg 2005

America or Western Europe already have around 20 000 000 nodes so that significantly superlinear preprocessing time or even slightly superlinear space is prohibitive. To our best knowledge, all commercial applications currently only compute paths heuristically that are not always shortest possible. The basic idea of these heuristics is the observation that shortest paths "usually" use small roads only locally, i.e., at the beginning and at the end of a path. Hence the heuristic algorithm only performs some kind of *local search* from s and t and then switches to search in a *highway network* that is much smaller than the complete graph. Typically, an edge is put into the highway network if the information supplied on its road type indicates that it represents an important road.

Our approach is based on the idea to compute *exact* shortest paths by defining the notion of *local search* and *highway network* appropriately. This is very simple. We define local search to be a search that visits the H closest nodes from s (or t) where H is a tuning parameter. This definition already fixes the highway network. An edge $(u, v) \in E$ should be a highway edge if there are nodes s and t such that (u, v) is on the shortest path from s to t, v is not within the H closest nodes from s, and u is not within the H closest nodes from t. Section 2 gives a more formal definition of the basic concepts used in this paper.

One might think that an expensive all-pairs shortest path computation is needed to find the highway network. However, in Section 3 we show that each highway edge is also within some local shortest path tree B rooted at some $s \in V$ such that all leaves of B are "sufficiently far away" from s.

So far, the highway network still contains all the nodes of the original network. However, we can prune it significantly: Isolated nodes are not needed. Trees attached to a biconnected component can only be traversed at the beginning and end of a path. Similarly, paths consisting of nodes with degree two can be replaced by a single edge. The result is a *contracted highway network* that only contains nodes of degree at least three. We can iterate the above approach, define local search on the highway network, find a "superhighway network", contract it,... We arrive at a multi-level highway network — a *highway hierarchy.*

Section 4 develops a query algorithm that uses highway hierarchies. After several correctness preserving transformations we get a bidirectional, Dijkstra-like search in a single graph that contains all levels. The only modifications affect the selection of edges to be relaxed and how to finish the search when the search frontiers from s and t meet.

In Section 5 we summarise experiments using detailed road networks for Western Europe and the USA. Using a uniform neighbourhood size of 125 and 225 respectively, the graphs shrink geometrically from level to level. This leads to preprocessing time around four hours and average query times below 8 ms. Possible future improvements are discussed in Section 6. Proofs and additional experimental data can be found in the full version at `http://www.dominik-schultes.de/hwy/`.

Related Work

There is so much literature on shortest paths and preprocessing that we can only highlight selected results that help to put our results into perspective. In the following, speedup refers to the acceleration compared to the unidirectional

variant of Dijkstra's algorithm that stops when the target is found. For recent, more detailed overviews we refer to [2,3].

Perhaps the most interesting theoretical results on route planning are algorithms for planar graphs that might be adaptable to route networks since those are "almost planar. Using $O(n \log^3 n)$ space and preprocessing time, query time $O(\sqrt{n} \log n)$ can be achieved [4] for directed planar graphs without negative cycles. Queries accurate within a factor $(1+\epsilon)$ can be answered in near constant time using $O((n \log n)/\epsilon)$ space and preprocessing time [5]. However, for very large graphs we need linear space consumption so that these approaches seem not directly applicable to the problem at hand.

The previous practical approach closest to ours is the *separator based multi-level method* [6]. The idea is to partition the graph into small subgraphs by removing a (hopefully small) set of separator nodes. These separator nodes together with edges representing precomputed paths between them constitute the next level of the graph. Queries then only need to search in the partitions of s and t and in the higher level graph. This process can be iterated. At least for road networks speedups so far seem to be limited to a factor around ten whereas better speedup can be observed for railway transportation problems [6]. Disadvantages compared to our method are that performance depends on very small (and thus hard to find) separators and that the higher level graphs get quite dense so that going to many levels quickly reaches a point of diminishing return. In contrast, our method has a very simple definition of what constitutes the higher level graphs and our higher level graphs remain sparse.

Reach based routing [7] excludes nodes from consideration if they do not contribute to any path long enough to be of use for the current query. Speedups up to 20 are reported for graphs with about 400 000 nodes using about 2 hours preprocessing time. Our method is an order of magnitude faster both in terms of query time and in terms of preprocessing time.

Most other preprocessing techniques are different from our approach in that they focus the search towards the target. Very high speedups (hundreds or even around 2000) are reported for *geometric containers* [8,3] and *bit vectors* [9,10]. Both methods store information with each edge. Queries use this information to decide whether this edge can possibly lead to the target. Our method achieves similar speedups but needs much less time for preprocessing: To compute k-bit vectors, $O(\sqrt{kn})$ global shortest path searches are needed (assuming planar graphs) and geometric containers even need an all-pairs computation.

An interesting alternative are *landmark* based lower bounds for strengthening goal directed search [2]. For global queries, about 16 global shortest path computations during preprocessing suffice to achieve speedup around 20. However, the landmark method needs a lot of space — one distance value for each node-landmark pair. It is also likely that for real applications each node will need to store distances to different sets of landmarks for global and local queries. Hence, landmarks have very fast preprocessing and reasonable speedups but consume too much space for very large networks.

2 Preliminaries

We expect an *undirected* graph $G = (V, E)$ with n nodes and m edges with nonnegative weights as input.[1] We assume w.l.o.g. that there are no self-loops, parallel edges, or 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. $P^* = \langle s, \dots, t\rangle$ 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 $P = \langle s, \dots, s', u_1, u_2, \dots, u_k, t', \dots, t\rangle$ is a path from s to t, then $P|_{s' \to t'} = \langle s', u_1, u_2, \dots, u_k, t'\rangle$ denotes the *subpath* of P from s' to t'.

Dijkstra's Algorithm. In the context of Dijkstra's algorithm, we use the following terminology: each node is either *unreached*, *reached*, or *settled*. If a node u is reached, at least one path (not necessarily the shortest one) from the source node s to u has been found and u has been inserted into the priority queue. If a node v is *settled*, it is reached and has been removed from the priority queue by a *deleteMin* operation; a shortest path from s to v has been found.

Canonical Shortest Paths. A *selection of shortest paths* $\mathcal{SP}$ contains for each connected pair $(s,t) \in V \times V$ exactly one shortest path from s to t. Such a selection is called *canonical* if $P = \langle s, \dots, s', \dots, t', \dots, t\rangle \in \mathcal{SP}$ implies that $P|_{s' \to t'} \in \mathcal{SP}$. The elements of a canonical selection are called *canonical shortest paths*. If Dijkstra's algorithm is started from each node $s \in V$, for each connected pair (s,t) exactly one shortest path is determined. In the full paper some modifications of Dijkstra's algorithm are described which ensure that the obtained selection of shortest paths is canonical.

Locality. Let us fix any rule that decides which element Dijkstra's algorithm removes from the priority queue when there is more than one queued element with the smallest key. Then, during a Dijkstra search from a given node s, all nodes are settled in a fixed order. The *Dijkstra rank* $r_s(v)$ of a node v is the rank of v w.r.t. this order. s has Dijkstra rank $r_s(s) = 0$, the closest neighbour v_1 of s has Dijkstra rank $r_s(v_1) = 1$, and so on. For a given node s, the distance of the H-closest node from s is denoted by $d_H(s)$, i.e., $d_H(s) = d(s,v)$, where $r_s(v) = H$. The *H-neighbourhood* $\mathcal{N}_H(s)$ (or just *neighbourhood* $\mathcal{N}(s)$) of s is $\mathcal{N}(s) := \{v \in V \mid d(s,v) \leq d_H(s)\}$. [2]

Highway Hierarchy. For a given parameter H, the *highway network* $G_1 = (V_1, E_1)$ of a graph G is defined as the set of edges $(u,v) \in E$ that appear in a canonical shortest path $\langle s, \dots, u, v, \dots, t\rangle$ from a node $s \in V$ to a node

[1] Unless otherwise stated, we always deal with *undirected* edges. The restriction to undirected graphs simplifies the presentation of our approach and the implementation. However, our method can be generalised to *directed* graphs.

[2] For directed graphs we also need an analogous value $\bar{d}_H(\cdot)$ that refers to the reverse graph $\bar{G} := (V, \{(v,u) \mid (u,v) \in E\})$. $\bar{\mathcal{N}}(\cdot)$ is defined correspondingly. From now on, whenever the target node t or the backward search from t is concerned, we have to keep in mind that $\bar{G}$, $\bar{d}_H(\cdot)$, and $\bar{\mathcal{N}}(\cdot)$ apply.

$t \in V$ with the property that $v \notin \mathcal{N}_H(s)$ and $u \notin \mathcal{N}_H(t)$. The set V_1 is the maximal subset of V such that G_1 contains no isolated nodes.

The *2-core* of a graph is the maximal vertex induced subgraph with minimum degree two. A graph consists of its 2-core and *attached trees*, i.e., trees whose roots belong to the 2-core, but all other nodes do not belong to it. A *line* in a graph is a path $\langle u_0, u_1, \ldots, u_k \rangle$ where the inner nodes $u_1, \ldots, u_{k-1}$ have degree two. From the highway network G_1 of a graph G, the *contracted highway network* G'_1 of the graph G is obtained by taking the 2-core of G_1 and, then, removing the inner nodes of all lines $\langle u_0, u_1, \ldots, u_k \rangle$ and replacing each line by an edge (u_0, u_k). Thus, the highway network G_1 consists of the contracted highway network (or short, just *core*) G'_1 and some *components*, where 'component' is used as a generic term for 'attached tree' and 'line'. In this paper, 'components' is used always in this specific sense and *not* to denote 'connected components' in general.

The *highway hierarchy* is obtained by applying the process that leads from G to G'_1 iteratively. The original graph $G_0 := G'_0 := G$ constitutes Level 0 of the highway hierarchy, G_1 corresponds to Level 1, the highway network G_2 of the graph G'_1 is called Level 2, and so on.

3 Construction

For each node $s_0 \in V$, we compute and store the value $d_H(s_0)$. This can be easily done by a Dijkstra search from each node s_0 that is aborted as soon as H nodes have been settled. Then, we start with an empty set of highway edges E_1. For each node s_0, two phases are performed: the forward construction of a partial shortest path tree B and the backward evaluation of B. The construction is done by a single source shortest path (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_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'.

Phase 1: Construction of a Partial Shortest Path Tree. A Dijkstra search from s_0 is executed. 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*) adopts the activation state from its (tentative) parent in the shortest path tree B. When a node p is settled using the path $\langle s_0, s_1, \ldots, p \rangle$, then p's state is set to passive if $|\mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$. When no active unsettled node is left, the search is aborted and the growth of B stops.

Phase 2: Selection of the Highway Edges. During Phase 2, all edges (u, v) are added to E_1 that lie on paths $\langle s_0, \ldots, u, v, \ldots, t_0 \rangle$ in B with the property that $v \notin \mathcal{N}(s_0)$ and $u \notin \mathcal{N}(t_0)$, where t_0 is a leaf of B. This can be done in time $O(|B|)$.

Theorem 1. *An edge $(u, v) \in E$ is added to E_1 by the construction algorithm iff it belongs to some canonical shortest path $P = \langle s, \ldots, u, v, \ldots, t \rangle$ and $v \notin \mathcal{N}(s)$ and $u \notin \mathcal{N}(t)$.*

Speeding Up Construction. An active node v is declared to be a *maverick* if $d(s_0, v) > f \cdot d_H(s_0)$, 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_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.

Theorem 2. *The highway network can be contracted in time $O(m + n)$.*

Highway Hierarchy. The result of the contraction is the contracted highway network G'_1, which can be used as input for the next iteration of the construction procedure in order to obtain the next level of the highway hierarchy.

4 Query

The *highway hierarchy* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of the graphs $G_0, G_1, G_2, \ldots, G_L$, which are arranged in $L+1$ levels. For each node $v \in V$ and each $i \in \{j \mid v \in V_j\}$, there is one copy of v, namely v_i, that belongs to level i of $\mathcal{G}$. Accordingly, there are several copies of an edge (u, v) when u and v belong to more than one common level. These edges, which connect two nodes in the same level, are called *horizontal* edges. Additionally, $\mathcal{G}$ contains a directed edge $(v_\ell, v_{\ell+1})$ for each pair $v_\ell \in V_\ell, v_{\ell+1} \in V_{\ell+1}$, where v_ℓ and $v_{\ell+1}$ are copies of the same node v. These additional edges are called *vertical* and have weight 0. For each node v, not only one value $d_H(v)$ is known, but for each level $\ell < L$, there is a distance $d^\ell_H(v)$ from v to the H-closest node in the graph G'_ℓ; if a node v does not belong to G'_ℓ, $d^\ell_H(v)$ is defined to be $+\infty$; furthermore, $d^L_H(v) := +\infty$. Correspondingly, we use the notation $\mathcal{N}^\ell(v)$ to refer to the set $\{v' \in V'_\ell \mid d(v, v') \leq d^\ell_H(v)\}$, which is the *neighbourhood* of v in the graph G'_ℓ. Note that the neighourhood of a node that belongs to a component is unbounded, i.e., it contains all nodes of the core of the corresponding level. The same applies to $\mathcal{N}^L(v)$, for any v.

The *multilevel query algorithm* that works on $\mathcal{G}$ is a modification of the bidirectional version of Dijkstra's algorithm. The source and target nodes of an s-t query are the corresponding copies of s and t in level 0. For the time being, we omit the abort-on-success criterion, i.e., we do not abort when both search scopes meet, but continue until both searches terminate; then, we consider all nodes that have been settled from both sides as meeting points and take the shortest path that has been found by this means. The modifications consist of two restrictions:

1. *In each level ℓ, no horizontal edge is relaxed that would leave the neighbourhood $\mathcal{N}^\ell(v^*)$ of the corresponding entrance point v^*.* Each node that belongs to the core and has been settled via a horizontal edge that leaves a component and each node that has been settled via a vertical edge is an *entrance point.* In addition, the source and the target nodes of the query are entrance points. The *corresponding entrance point* of a settled node v is the last entrance point on the path to v.

2. *Components are never entered using a horizontal edge.* An edge (u, v) *enters* a component if either u belongs to the core and v to a component or u belongs to a line and v to an attached tree. However, an edge from an attached tree to a line *leaves* the attached tree and does not rank among the edges that enter a component. Note that the endpoint(s) of a component do not belong to the component but to the core (or to the line in case of the root of a tree that is attached to a line).

Theorem 3. *For any given $s, t \in V$, the multilevel query algorithm finds the shortest path from s to t in G.*

Proof Idea. It is known that the bidirectional version of Dijkstra's algorithm works correctly. We have to show that the imposed restrictions do not affect the correctness. When Restriction 1 applies, it is always possible to switch to the next level using a vertical edge. Due to the definition of the highway network, it is guaranteed that the corresponding part of the shortest path which we are looking for can be found in the next level. A path from s that *enters* a component is not traversed due to Restriction 2. However, from the point of view of t, this path *leaves* the component so that the edge that has been skipped during the search from s can be relaxed in the reverse direction during the search from t. Hence, the path can be found in spite of Restriction 2. These arguments can be used in an inductive proof over the number of levels. □

Collapse of the Vertical Dimension. So far, we allow that several copies of the same node are reached. However, it can be shown that it is sufficient if at most one copy of a node is reached via a horizontal edge, namely the copy with the smallest tentative distance or, if there are several copies with the same smallest tentative distance, the copy in the lowest level.

Due to this observation, we can let the vertical dimension collapse. We can interpret the highway hierarchy $\mathcal{G}$ as one plain graph, i.e., there are no copies of the nodes distributed over several levels. Basically, this graph corresponds to the original graph G enhanced by some additional data: each edge (u, v) is assigned a maximum level $\ell(u, v)$, i.e., it belongs to the levels $0, 1, \ldots, \ell(u, v)$; each node v is assigned to at most one component $c(v)$; a component $c(v)$ belongs to a certain level $\ell(c(v))$, which is equal to the level its inner edges belong to. Furthermore, the value $d_H^\ell(v)$ is stored only if $v \in G'_\ell$. Our implementation is based on this interpretation of $\mathcal{G}$.

Abort-on-Success. In the bidirectional version of Dijkstra's algorithm, we can abort as soon as both search scopes meet, i.e., there is one node v that is settled in both search scopes. Then, the shortest path P from s to t does not necessarily consist of the shortest paths from s to v and from v to t, but it is well known that it is always ensured that the right meeting point v' has already been reached from both sides. The crucial precondition for this fact is that all nodes whose distance from s is less than $d(s, v)$ have been settled in the search scope of s, and all nodes whose distance from t is less than $d(t, v)$ have been settled in the search scope of t.

Unfortunately, we cannot adopt the abort-on-success criterion as it stands because, in general, the multilevel query algorithm does not fulfil this precondition as several edges are not relaxed due to Restriction 1 and 2 so that we cannot guarantee that all nodes up to a certain distance have been settled. We have already shown that the algorithm is correct all the same because if an 'important' edge is not relaxed (e.g. a component is not entered), then it is relaxed from the other side (e.g. the component is left). However, we have to *wait* until the reverse search has relaxed this edge, i.e., we must not abort too early. Nevertheless, even if we have skipped an edge $e = (u, v)$ at node u, we *can* abort after both search scopes have met as soon as it is certain that e will not be relaxed from v during the final steps of the search. We can use the following approach. Let $\mathcal{E}_s$ denote the set of all horizontal edges that have been skipped during the search from s. $\mathcal{E}_t$ is defined accordingly. After both search scopes have met, we can abort as soon as the search from t has finished search level $\hat{\ell}_s := \max_{e \in \mathcal{E}_s} \ell(e)$ and the search from s has finished level $\hat{\ell}_t := \max_{e \in \mathcal{E}_t} \ell(e)$. A search level ℓ is *finished* when there are no reached but unsettled nodes in level ℓ or below. If the search level ℓ is finished, edges e in levels $\ell(e) \leq \ell$ cannot be relaxed any longer. Hence, when the search from t has finished search level $\hat{\ell}_s$, it is certain that no edge e that belongs to a level $\ell(e) \leq \hat{\ell}_s$ will be relaxed during the final steps of the search, in particular, no edge that has been skipped during the search from s will be traversed by the backward search. There are several possibilities to further improve this abort criterion, which are described in the full paper.

5 Experiments

Implementation. We use a binary heap priority queue. Our current implementation leaves room for reducing both running time and memory usage.

Environment. The experiments were done on a 64-bit machine with 8 GB main memory and 1 MB L2 cache, using one out of four AMD Opteron processors clocked at 2.2 GHz, running SuSE Linux (kernel 2.6.5). The program was compiled by the GNU C++ compiler 3.3.3 using optimisation level 3.

Instances. Basically, we deal with two test instances, namely, the road networks of the United States of America (minus Alaska and Hawaii) and of Western Europe. The former was obtained from the TIGER/Line Files [11] by merging the relevant data of all counties. The latter contains the 14 European countries Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the UK. The data has been made available for scientific use by the company PTV AG. In some cases, we restrict our experiments to the German road network. In all cases, as we deal with undirected graphs, we ignored the restrictions caused by one-way streets.

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

Table 1. Overview of the used road networks and key results. The parameter H is used iteratively until the construction leads to an empty highway network. We provide average values for 10 000 queries, where the source and target nodes are chosen randomly. 'Speedup' refers to a comparison with Dijkstra's algorithm[3]. 'Efficiency' [2] denotes the number of nodes that belong to the computed shortest paths divided by the number of nodes that are settled by the multilevel query algorithm. For Germany, we give the memory usage on a 32-bit machine in parentheses.

		USA	Europe	Germany
input	#nodes	24 278 285	18 029 721	4 345 567
	#edges	29 106 596	22 217 686	5 446 916
	#degree 2 nodes	7 316 573	2 375 778	604 540
	#road categories	4	13	13
parameters	average speeds [km/h]	40–100	10–130	10–130
	H	225	125	100
construction	CPU time [h:min]	4:15	2:41	0:30
	#levels	7	11	11
query	CPU time [ms]	7.04	7.38	5.30
	#settled nodes	3 912	4 065	3 286
	speedup (CPU time)	2 654	2 645	680
	speedup (#settled nodes)	3 033	2 187	658
	efficiency	113%	34%	13%
	main memory usage [MB]	2 443	1 850	466 (346)

Fast vs. Precise Construction. During various experiments, we came to the conclusion that it is a good idea *not* to take a fixed maverick factor f for all levels of the construction process, but to start with a low value (i.e. fast construction) and increase it level by level (i.e. more precise construction). For the following experiments, we used the sequence $0, 2, 4, 6, \ldots$.

Best Neighbourhood Sizes. 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 the USA and Europe 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. We kept this approach and applied the same H iteratively until the construction led to an empty highway network. Figure 1 demonstrates the shrinking process for Europe. For most levels, 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 first iteration (Level 0→1) and the last few iterations are exceptions: at the first iteration, the construction works very well due to the characteristics of the real world road network (there are many trees and lines that can be contracted); at the last iterations, 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

[3] The averages for Dijkstra's algorithm are based on only 1 000 queries.

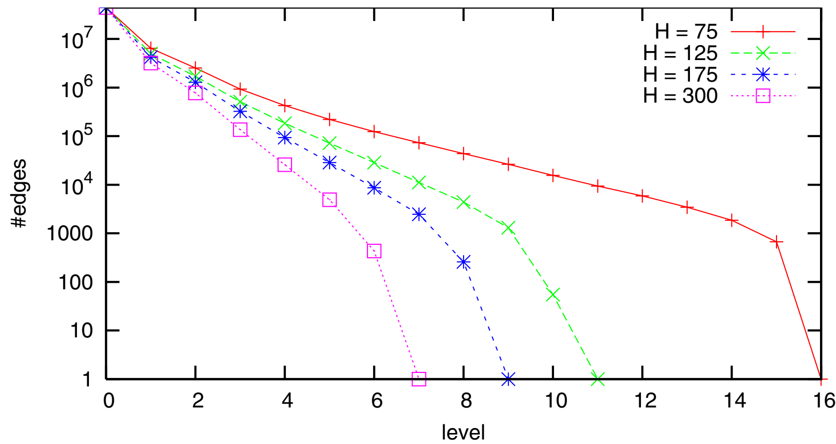

Fig. 1. Shrinking of the highway networks of Europe. For different neighbourhood sizes H and for each level ℓ, we plot $|E'_\ell|$, i.e., the number of edges that belong to the core of level ℓ.

to the next level of the highway hierarchy, and when the network gets small, almost all nodes are close to the border.

Multilevel Queries. Table 1 contains average values for queries, where the source and target nodes are chosen randomly. For the two large graphs we get a speedup of more than 2 000 compared to Dijkstra's algorithm both with respect to (query) time[4] and with respect to the number of settled nodes.

For our largest road network (USA), the number of nodes that are settled during the search is *less* than the number of nodes that belong to the shortest paths that are found. Thus, we get an efficiency that is greater than 100%. The reason is that edges at high levels will often represent long paths containing many nodes.[5]

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 $r_s(t) = r$. 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 2 shows the query times. Note that the median

[4] It is likely that Dijkstra would profit more from a faster priority queue than our algorithm. Therefore, the time-speedup could decrease by a small constant factor.

[5] The reported query times do not include the time for expanding these paths. We have made measurements with a naive recursive expansion routine which never take more than 50% of the query time. Also note that this process could be radically sped up by precomputing unpacked representations of edges.

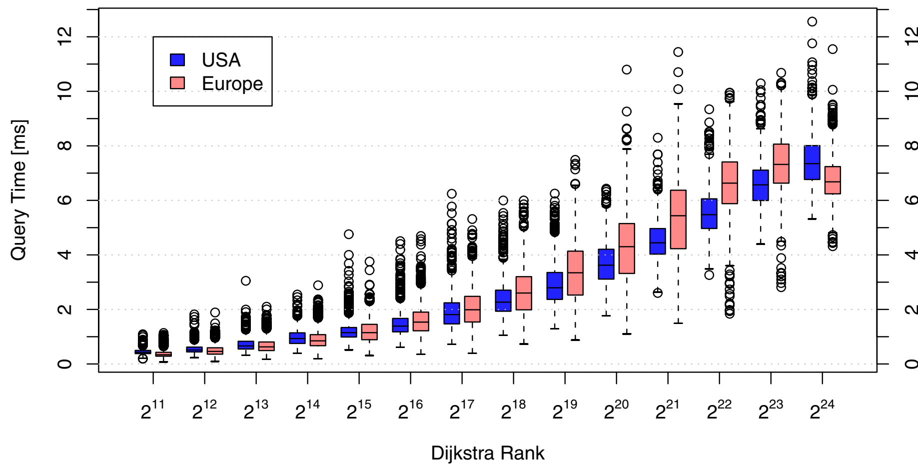

Fig. 2. Multilevel Queries. For each road network and each Dijkstra rank on the x-axis, 1 000 queries from random source nodes were performed. The results are represented as box-and-whisker plot [12]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

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.

6 Discussion

Starting from a simple definition of local search, we have developed nontrivial algorithms for constructing and querying highway hierarchies. We have demonstrated that highway hierarchies of the largest road networks currently used can be constructed in a few hours, i.e., fast enough to allow daily updates. The space consumption is only a small constant factor of the input size. The query times around 10 ms are more than fast enough for interactive use. The only previous speedup techniques that would achieve comparable speedup (bit vectors, geometric containers) have prohibitive preprocessing times for very large graphs.

Even faster preprocessing is a major issue for future work. We see many small (and not so small) opportunities for improvement. The local nature of preprocessing makes it likely that highway hierarchies can be quickly updated dynamically when only a few edges (e.g., for taking traffic jams into account) or a region of the network changes. We can also easily parallelise preprocessing.

Even faster queries are also interesting. For example, for some traffic simulations, millions of shortest paths queries are needed and there is no overhead for a user interface. Besides many small improvements (faster priority queues. . .)

a combination with other speedup techniques seems interesting. In particular, bit vectors, geometric containers, or landmarks give the search a strong sense of direction that highway hierarchies lack, i.e., these two basic approaches may complement one another. Moreover, the higher levels of the hierarchy are so small that superlinear time or space may be tolerable as long as the contributions of the lower levels can be incorporated efficiently.

Acknowledgements

We would like to thank Andrew Goldberg, Rolf Möhring, Matthias Müller-Hannemann, Heiko Schilling, Frank Schulz, Mikkel Thorup, and Dorothea Wagner for interesting discussions on various speedup techniques. Martin Holzer, Domagoj Matijevic, Frank Schulz, and Thomas Willhalm have also helped with data and tools for processing graphs.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
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, Technische Universität Karlsruhe (2005)
4. Fakcharoenphol, J., Rao, S.: Negative weight edges, shortest paths, near linear time. In: 42nd Symposium on Foundations of Computer Science. (2001) 232–241
5. Thorup, M.: Compact oracles for reachability and approximate distances in planar digraphs. In: 42nd Symposium on Foundations of Computer Science. (2001) 242–251
6. 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
7. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: 6th Workshop on Algorithm Engineering and Experiments (ALENEX). (2004)
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: Proc. Münster GI-Days. (2004)
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)
11. U.S. Census Bureau, Washington, DC: UA Census 2000 TIGER/Line Files. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html` (2002)
12. The R Development Core Team: R: A Language and Environment for Statistical Computing, Reference Index. `http://www.r-project.org/` (2004)