Robust, Almost Constant Time Shortest-Path Queries in Road Networks[⋆]

Peter Sanders and Dominik Schultes

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

Abstract. When you drive to somewhere 'far away', you will leave your current location via one of only a few 'important' traffic junctions. Starting from this informal observation, we develop a generic algorithmic approach—*transit node routing*—that allows us to reduce quickest-path queries in road networks to a small number of table look-ups. We implement this basic approach using *highway hierarchies*. For the road maps of Western Europe and the United States, our best query times improve over the best previously published figures by two orders of magnitude. This is more than one million times faster than the best known algorithm for general networks. We also explain how to compute complete descriptions of shortest paths (and not only their lengths) very efficiently.

1 Introduction

Computing an optimal route in a road network between specified source and target nodes (i.e., places/intersections) is one of the showpieces of real-world applications of algorithmics. Besides the omnipresent application of car navigation systems and internet route planners, even faster route planning is needed for massive traffic simulation and optimisation in logistics systems. Beyond mere computational efficiency, the methods presented here also give quantitative insight into the structure of road networks and justify the way humans do route planning.

The classical algorithm for route planning—Dijkstra's algorithm [1]—iteratively visits all nodes that are closer to the source node than the target node before reaching the target. On road networks for a subcontinent like Western Europe or the USA, this takes about ten seconds on a state-of-the-art workstation. Since this is too slow for many applications, commercial systems use heuristics that do not guarantee optimal routes. Therefore, there has been considerable interest in speedup techniques for computing *optimal* routes.

Recently, Bast, Funke and Matijevic [2] have introduced a notion we call *transit node routing* which is based on the following two key observations: First, there is a relatively small set of *transit nodes*, about 10 000 for the US road network, with the property that for every pair of nodes that are 'not too close' to each other, the shortest path between them passes through *at least one* of these transit nodes. Second, for every node, the set of transit nodes encountered first when going far—we call these *access nodes*—is small (about 10). They have implemented this idea using a uniform grid to define 'sufficiently far away'. This way about 98% of all queries can be answered using a few table look-ups. However, since the remaining 2% of the local queries are orders of magnitude slower, they are not able to report query times that outperform the fastest existing implementation, which uses highway hierarchies [3]. Furthermore, preprocessing time, though subquadratic, is very high.

Independently, Müller et al. [4] have developed a similar approach based on vertex separators. This approach uses several CPU-days of preprocessing and more space than fits on a

[⋆] Partially supported by DFG grant SA 933/1-3.

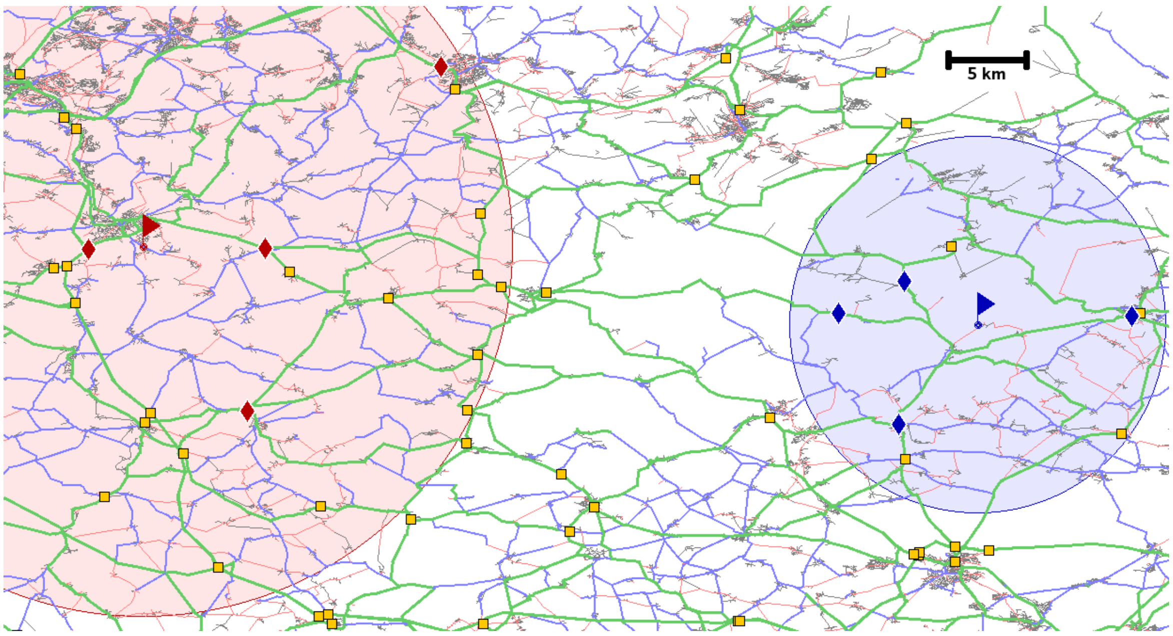

Fig. 1. Finding the optimal travel time between two points somewhere between Saarbrücken and Karlsruhe amounts to retrieving the 2×4 *access nodes* (diamonds), performing 16 table look-ups between all pairs of access nodes, and checking that the two disks defining the *locality filter* do not overlap. The figure draws the levels of the highway hierarchy using colours grey, red, blue, and green for levels 0–1, 2, 3, and 4, respectively. *Transit nodes* are drawn as small orange squares.

single hard disk. After all data needed for a query is present in the processor cache, a query still takes about 50 μs.

We present the first complete implementation of transit node routing. We first develop transit node routing into a generic technique in Section 2 that can be instantiated in many ways. In particular, we add further *layers* to transit node routing that allow to handle local queries as well. We instantiate this approach for highway hierarchies [5, 3] in Section 3. Figure 1 gives an example. Experiments reported in Section 4 give average query times of about 5 μs and query times around 20 μs for slowest category of queries. Our preprocessing times are slower than for highway hierarchies alone but faster than in [2]. Our main focus is on computing *quickest path* even if we use the term *short*. However, we also give some results on computing travel distances.

Related Work

Bidirectional Search. A classical technique is *bidirectional search* which simultaneously searches forward from s and backwards from t until the search frontiers meet. Many more advanced speedup techniques (including ours) use bidirectional search as an ingredient.

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 [6]. Using $O(n \log^3 n)$ space and preprocessing time, query time $O(\sqrt{n} \log n)$ can be achieved [7] for directed planar graphs without negative cycles. A previous practical approach based on separators is the *separator based multi-level method* [8]. The idea is to partition the graph into small components 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.

Using more space and preprocessing time, separators can be used for transit node routing. The separator nodes become transit nodes and the access nodes are the border nodes of the component of v. Local queries are those within a single component. Another layer of transit nodes can be added by recursively finding separators of each component. Müller et al. [4] have essentially developed this approach (using different terminology). An interesting difference to generic transit node routing is that the required information for routing between any pair of components is arranged together. This takes additional space but has the advantage that the information can be accessed more cache efficiently (it also allows subsequent space optimisations). Although separators of road networks have much better properties than the worst case bounds for planar graphs would suggest, separator based transit node routing needs many more access nodes than our schemes (≈ 80 rather than ≈ 10 per node for Western Europe). This leads to higher space consumption, preprocessing time, and query time. The main reason for the difference in number of access nodes is that the separator approach does not take the 'sufficiently far away' criterion into account that is so important for reducing the number of access nodes in our scheme.

Highway Hierarchies. 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 previous papers [5, 3] we introduced the idea to *automatically* compute *highway hierarchies* that yield *optimal* routes *uncompromisingly quickly*. The basic idea is to define a neighbourhood for each node to consist of its H closest neighbours. Now an edge (u, v) is a highway edge if there is some shortest path $\langle s, \dots, u, v, \dots t\rangle$ such that neither u is in the neighbourhood of t nor v is in the neighbourhood of s. This defines the first level of the highway hierarchy. After contracting the network to remove low degree nodes, the same procedure (identifying the highway network at the next level followed by contraction) is applied recursively. We obtain a hierarchy. The query algorithm is bidirectional Dijkstra with restrictions on relaxing certain edges. Roughly, far away from source or target, only high level edges need to be considered. Highway hierarchies are successful (several thousand times faster than Dijkstra) because of the property of real world road networks that for *constant neighbourhood size* H, the levels of the hierarchy *shrink geometrically*. One can view this as a *self-similarity*—each level of the hierarchy looks similar to the original network, just a constant factor smaller. Under certain (somewhat optimistic) assumptions, this self-similarity yields *logarithmic* query time in contrast to the superlinear query time of Dijkstra's algorithm.

Reach Based Routing. Comparable effects can be achieved with the closely related technique of *reach based routing* [9, 10].

Distance Tables. In [3] transit node routing is *almost* anticipated. Precomputed all-to-all distances on some sufficiently high level—say K— of the highway hierarchy are used to terminate the local searches when they ascended far enough in the hierarchy. The main differences to transit node routing is that access points are computed online and that only distances within level K of the highway hierarchy (rather than distances in the underlying graph) are precomputed. This leads to much larger sets of access points (≈ 55) that made precomputing them appear much less attractive as it actually is. It was also not addressed, how to decide *when* the distance given by the distance table is the actual shortest path distance.

Goal Direction. Another interesting property of road networks is that they allow effective goal directed search using **A*** *search* [11]: 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 [12, 13] using the **T**riangle inequality (*ALT*). In combination with reach based routing, this is one of the fastest known speedup techniques [10]. An interesting observation is that in transit node routing, the access nodes could be used as landmarks (with aid of the distance tables). The resulting lower bound could be used for distinguishing local and global queries or for guiding local search.

Geometry. Finally, a tempting property of road networks is that nodes have a geographic position. Even if this information is not available, equally useful coordinates can be synthesised [14]. Interestingly, so far, successful geometric speedup techniques have always been beaten by related non-geometric techniques (e.g. [11] by [12, 13] or [15] by [16, 17]). We initially thought that the highway hierarchy approach outperforming the grid based approach to transit node routing would turn out to be another instance of this phenomenon. However, currently it looks like the highway hierarchy approach needs a geometric locality filter for good performance.

2 Transit Node Routing

To simplify notation we will present the approach for undirected graphs. However, the method is easily generalised to directed graphs and our highway hierarchy implementation already handles directed graphs. Consider any set $\mathcal{T} \subseteq V$ of *transit nodes*, an *access mapping* $A : V \rightarrow 2^{\mathcal{T}}$, and a *locality filter* $L : V \times V \rightarrow \{\mathsf{true}, \mathsf{false}\}$. We require that $\neg L(s,t)$ implies that the shortest path distance is

$$d(s,t) = \min\left\{d(s,u) + d(u,v) + d(v,t) : u \in A(s), v \in A(t)\right\} \quad . \tag{1}$$

In principle, we can pick any set of transit nodes, any access mapping, and any locality filter fulfilling Equation (1) to obtain a transit node query algorithm:
Assume we have precomputed all distances between nodes in $\mathcal{T}$.
If $\neg L(s,t)$ then compute $d(s,t)$ using Equation (1)
Else, use any other routing algorithm.

Of course, we want a good choice of $(\mathcal{T}, A, L)$. $\mathcal{T}$ should be small but allow many global queries, L should efficiently identify as many of these global query pairs as possible, and we should be able to store and evaluate A efficiently.

We can apply a *second layer* of *generalised transit node routing* to the remaining local queries (that may dominate some real world applications). We have a node set $\mathcal{T}_2 \supset \mathcal{T}$, an access mapping $A_2 : V \rightarrow 2^{\mathcal{T}_2}$, and a locality filter L_2 such that $\neg L_2(s,t)$ implies that the shortest path distance is defined by Equation 1 or by

$$d(s,t) = \min\left\{d(s,u) + d(u,v) + d(v,t) : u \in A_2(s), v \in A_2(t)\right\} \quad . \tag{2}$$

In order to be able to evaluate Equation 2 efficiently we need to precompute the local connections from $\{d(u,v) : u,v \in \mathcal{T}_2 \wedge L(u,v)\}$ which cannot be obtained using Equation 1.

In an analogous way we can add further layers.

General Techniques

We now describe techniques that can be used together with any set of transit nodes. The more specific techniques presented in Section 3 will refine and in some cases replace these general techniques.

Computing Access Nodes: Backward Approach. Start a Dijkstra search from each transit node $v \in \mathcal{T}$. Run it until all paths leading to nodes in the priority queue pass over another node $w \in \mathcal{T}$. Record v as an access node for any node u on a shortest path from v that does not lead over another node in $\mathcal{T}$. Record an edge (v,w) with weight $d(v,w)$ for a *transit graph* $G[\mathcal{T}] = (\mathcal{T}, E_{\mathcal{T}})$. When this local search has been performed from all transit nodes, we have found all access nodes and the distance table can be computed using an all-pairs shortest path computation in $G[\mathcal{T}]$.

Layer 2 Information is computed similarly to the top level information except that a search on the transit graph $G[\mathcal{T}_2]$ can be stopped when all paths in the priority queue pass over a top level transit node $w \in \mathcal{T}$. Level 2 distances from each node $v \in \mathcal{T}_2$ can be stored space efficiently in a static hash table. We only need to store distances that actually improve on the distances obtained going via the top level $\mathcal{T}$.

Computing Access Nodes: Forward Approach. Start a Dijkstra search from each node u. Stop when all paths in the shortest path tree are ‘covered’ by transit nodes. Take these transit nodes as access points of u. Applied naively, this approach is rather inefficient. However, we can use two tricks to make it efficient. First, during the search we do not relax the edges leaving transit nodes. This leads to the computation of a superset of the access points. Fortunately, this set can be easily reduced if the distances between all transit nodes are already known: if an access point v' can be reached from u via another access point v on a shortest path, we can discard v'. Second, we can only determine the access point sets $A(v)$ for all nodes $v \in \mathcal{T}_2$ and the sets $A_2(u)$ for all nodes $u \in V$. Then, for any node u, $A(u)$ can be computed as $\bigcup_{v \in A_2(u)} A(v)$. Again, we can use the reduction technique to remove unnecessary elements from the set union.

Locality Filters. There seem to be two basic approaches to transit node routing. One that starts with a locality filter L and then has to find a good set of transit nodes $\mathcal{T}$ for which L works (e.g., [2]). The other approach starts with $\mathcal{T}$ and then has to find a locality filter that can be efficiently evaluated and detects as accurately as possible whether local search is needed (e.g., Section 3). One approach that we found very effective is to use the information gained when computing the distance table for layer $i+1$ to define a locality filter for layer i. For example, we can compute the radius $r^i(u)$ of a circle around every node $u \in \mathcal{T}_{i+1}$ that contains for each entry $d(u,v)$ in the layer-$(i+1)$ table the meeting point of a bidirectional search between u and v. We can use this information in several ways. We can (pre)compute conservative circle radii for arbitrary nodes v as $r^i(v) := \max\{||v-u||_2 + r^i(u) : u \in A_{i+1}(v)\}$. Note that even if we are not able to store the information gathered during a precomputation at layer $i+1$, it might still make sense to run it in order to gather the more compact locality information.

Space Efficient Storage of Access Nodes. If all shortest paths from a node v to its access nodes $A(v)$ have to go over nodes from a set M, we can exploit that $A(v) \subseteq A(M) := \bigcup_{u \in M} A(u)$. Moreover, if the nodes in M are 'close' to v, we can expect that $A(M)$ is not too much bigger than $A(v)$. Therefore, as long as we can efficiently find M, it suffices to store access node information with a subset of the nodes. This subset might be $\mathcal{T}_2$ or a separator partitioning the graph into small pieces.

Outputting Shortest Paths (rather than only distances). First note that in a graph with bounded degree (e.g. a road network) and with a (near) constant time distance oracle, we can output a shortest path from s to t in (near) constant time per edge: Look for an edge (s, u) such that $w(s, u) + d(u, t) = d(s, t)$, output (s, u). Continue by looking for a shortest path from u to t. Repeat until t is reached. We can speed up this process by two measures. Suppose the shortest path uses the access nodes $x \in A(s)$ and $y \in A(t)$. First, while reconstructing the path from s to x (and from y to t) we can use this access node information to eliminate all search for the right access nodes and perform only a single distance table look-up. Second, reconstructing the path from x to y can work on the transit graph $G[\mathcal{T}]$ rather than on the original graph. We can precompute information that allows us to output the paths associated with each edge in $G[\mathcal{T}]$ in time linear in the number of edges of G it contains. Note that long distance paths will mostly consist of these precomputed paths so that the time per edge can be made very small. This technique can be generalised to multiple layers.

3 Instantiation Using Highway Hierarchies

Preliminaries. For each node v, we define some neighbourhood node set $\mathcal{N}(v)$. Then, the *highway network* of a graph $G = (V, E)$ is defined by its edge set: an edge $(u, v) \in E$ belongs to the highway network iff there are nodes $s, t \in V$ such that the edge (u, v) appears in the shortest path $\langle s, \ldots, u, v, \ldots, t \rangle$ with the property that $v \notin \mathcal{N}(s)$ and $u \notin \mathcal{N}(t)$. The size of a highway network (in terms of the number of nodes) can be considerably reduced by a contraction procedure: for each node v, we check a *bypassability criterion* that decides whether v should be *bypassed*—an operation that creates shortcut edges (u, w) representing paths of the form $\langle u, v, w \rangle$. The graph that is induced by the remaining nodes and enriched by the shortcut edges forms the *core* of the highway network.

A *highway hierarchy* of a graph G consists of several levels $G_0, G_1, G_2, \ldots, G_L$. Level 0 corresponds to the original graph G. Level 1 is obtained by computing the *highway network* of level 0, level 2 by computing the highway network of the core G_1' of level 1 and so on.

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* $\mathrm{rk}_s(v)$ of a node v is the rank of v w.r.t. this order.

Transit Nodes. Nodes on high levels of a highway hierarchy have the property that they are used on shortest paths far away from starting and target nodes. 'Far away' is defined with respect to the Dijkstra rank. Hence, it is natural to use (the core of) some level K of the highway hierarchy for the transit node set $\mathcal{T}$. Note that we have quite good (though indirect) control over the resulting size of $\mathcal{T}$ by choosing the appropriate neighbourhood sizes and the appropriate value for $K =: K_1$. In our current implementation this is level 4, 5, or 6.

In addition, the highway hierarchy helps us to efficiently compute the required information. Note that there is a difference between the *level* of the highway hierarchy and the *layer* of transit node search.

Access Nodes and Distance Tables. We use our highway hierarchy based code for many-to-many routing to compute the top level distance table [18]. Roughly, this algorithm first performs independent backward searches from all transit nodes and stores the gathered distance information in *buckets* associated with each node. Then, a forward search from each transit node scans all buckets it encounters and uses the resulting path length information to update a table of tentative distances. This approach can be generalised for computing distances at layer $i > 1$.

We use the forward approach from Section 2 to compute the access point sets. (In our case, we do not perform Dijkstra searches, but highway searches [3].)

Figure 2 summarises the representation used for running our algorithm. We have two variants. Variant *economical* aims at a good compromise between space consumption, preprocessing time and query time. Economical uses $K = 5$ and reconstructs the information needed for the layer-1 query using information only stored with nodes in $\mathcal{T}_2$. Variant *generous* accepts larger distance tables by choosing $K = 4$ (however using somewhat larger neighbourhoods for constructing the hierarchy). Generous stores all information required for a query with every node. To obtain a high quality layer-2 filter L_2, the generous variant performs a complete layer-3 preprocessing based on the core of level 1 and also stores a distance table for layer 3.

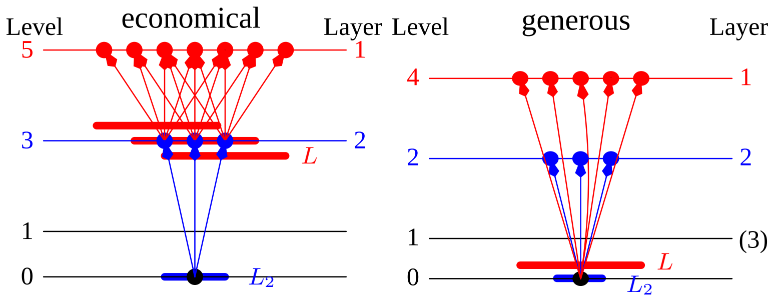

Fig. 2. Representations of information relevant to highway hierarchy transit node routing.

Queries are performed in a top down fashion. For a given query pair (s, t), first $A(s)$ and $A(t)$ are computed. Then table look-ups in the top level distance table yield a first guess for $d(s, t)$. Now, if $\neg L(s, t)$ we are done. Otherwise, the same procedure is repeated for layer two. If even $L_2(s, t)$ is true, we perform a bidirectional highway hierarchy search that can stop if both the forward and backward search radius exceed the upper bounds computed at layers 1 and 2. Furthermore, the search need not expand nodes at the core of level K_2 since paths going over these nodes are covered by the search in layers 1 and 2. In the generous variant, the search is already stopped at the level-1 core nodes, which form the access point set for layer 3. Additional look-ups in the layer-3 table ensure the correctness of this variant.

4 Experiments

4.1 Environment, Instances, and Parameters

The experiments were done on one core of an 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 two road networks. The network of Western Europe[1] has been made available for scientific use by the company PTV AG. Only the largest strongest connected component is considered. The original graph contains 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 addition to this *travel time metric*, we perform experiments on a variant of the European graph with a *distance metric*. The network of the USA (without Alaska and Hawaii) has been obtained from the TIGER/Line Files [19]. Again, we consider only the largest strongest connected component, and we deal with both a travel time and a distance metric. In contrast to the PTV data, the TIGER graph is undirected, planarised and distinguishes only between four road categories. All graphs[2] have been taken from the DIMACS Challenge website [20]. Table 1 summarises the properties of the used networks.

Table 1. Properties of the used road networks.

	Europe	USA
#nodes	18 010 173	23 947 347
#directed edges	42 560 279	58 333 344
#road categories	13	4
average speeds [km/h]	10–130	40–100

In Section 4.2 we report only the times needed to compute the shortest path distance between two nodes without outputting the actual route, while in Section 4.3, we also give the times needed to get a complete description of the shortest paths.

Since it has turned out that a better performance is obtained when the preprocessing starts with a contraction phase, we practically skip the first construction step (by choosing neighbourhood sets that contain only the node itself) so that the first highway network virtually corresponds to the original graph. Then, the first real step is the contraction of level 1 to get its core. Note that compared to [3, 21], we use a slightly improved contraction heuristic, which sorts the nodes according to degree and then tries to bypass the node with the smallest degree first.

The shortcut hops limit (introduced in [21]) is set to 10. The settings of the other parameters (some of them have been introduced in [5, 3]) can be found in Tab. 2. Note that when using the travel time metric (time), for all levels of the hierarchy, we use a constant contraction rate c and a constant neighbourhood size H—a different one for the economical (eco)

[1] 14 countries: Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the UK

[2] Note that the experiments on the TIGER graphs had been performed before the final versions, which use a finer edge costs resolution, were available. We did not repeat the experiments since we expect hardly any change in our measurement results.

and the generous (gen) variant. For the distance metric (dist), we use linearly increasing sequences for c and H.

Table 2. Parameters.

metric	time		dist
variant	eco	gen	eco
levels of layers 1–2(–3)	5–3	4–2–1	6–4
neighbourhood size H	60	110	90, 180, 270, . . .
contraction rate c	1.5	1.5	1.5, 1.6, 1.7, . . .

4.2 Main Results

Table 3 gives the preprocessing times for both road networks and both the travel time and the distance metric; in case of the travel time metric, we distinguish between the economical and the generous variant. In addition, some key facts on the results of the preprocessing, e.g., the sizes of the transit node sets, are presented. It is interesting to observe that for the travel time metric in layer 2 the actual distance table size is only about 0.1% of the size a naive $|\mathcal{T}_2| \times |\mathcal{T}_2|$ table would have. As expected, the distance metric yields more access points than the travel time metric (a factor 2–3) since not only junctions on very fast roads (which are rare) qualify as access point. The fact that we have to increase the neighbourhood size from level to level in order to achieve an effective shrinking of the highway networks leads to comparatively high preprocessing times for the distance metric.

Table 3. Statistics on preprocessing for the highway hierarchy approach. For each layer, we give the size (in terms of number of transit nodes), the number of entries in the distance table, and the average number of access points to the layer. 'Space' is the total *overhead* of our approach.

			layer 1			layer 2			layer 3			
	metric	variant	$\|\mathcal{T}\|$	|table| [$\times 10^6$]	$\|A\|$	$\|\mathcal{T}_2\|$	|table$_2$| [$\times 10^6$]	$\|A_2\|$	$\|\mathcal{T}_3\|$	|table$_3$| [$\times 10^6$]	space [B/node]	time [h]
USA	time	eco	12 111	147	6.1	184 379	30	4.9	–	–	111	0:59
		gen	10 674	114	5.7	485 410	204	4.2	3 855 407	173	244	3:25
	dist	eco	15 399	237	17.0	102 352	41	10.9	–	–	171	8:58
EUR	time	eco	8 964	80	10.1	118 356	20	5.5	–	–	110	0:46
		gen	11 293	128	9.9	323 356	130	4.1	2 954 721	119	251	2:44
	dist	eco	11 610	135	20.3	69 775	31	13.1	–	–	193	7:05

Table 4 summarises the average case performance of transit node routing. For the travel time metric, the generous variant achieves average query times more than two orders of magnitude lower than highway hierarchies alone [3]. At the cost of a factor 2.4 in query time, the economical variant saves around a factor of two in space and a factor of 3.5 in preprocessing time.

Finding a good locality filter is one of the biggest challenges of a highway hierarchy based implementation of transit node routing. The values in Tab. 4 indicate that our filter is suboptimal: for instance, only 0.0064% of the queries performed by the economical variant in the US network with the travel time metric would require a local search to answer them correctly. However, the locality filter L_2 forces us to perform local searches in 0.278% of all cases. The high-quality layer-2 filter employed by the generous variant is considerably more effective, still the percentage of false positives is about 90%.

For the distance metric, the situation is worse. Only 92% and 82% of the queries are stopped after the top layer has been searched (for the US and the European network, respectively). This is due to the fact that we had to choose the cores of levels 6 and 4 as layers 1 and 2 since the shrinking of the highway networks is less effective so that lower levels would be too big. It is important to note that we concentrated on the travel time metric—since we consider the travel time metric more important for practical applications—, and we spent comparatively little time to tune our approach for the distance metric. For example, a variant using a third layer (namely levels 6, 4, and 2 as layers 1, 2, and 3), which is not yet supported by our implementation, seems to be promising. Nevertheless, the current version shows feasibility and still achieves an improvement of a factor of 71 and 56 (for the US and the European network, respectively) over highway hierarchies alone [21, Tab. 5, with distance table optimisation].

Table 4. Performance of transit node routing with respect to 10 000 000 randomly chosen (s,t)-pairs. Each query is performed in a top-down fashion. For each layer i, we report the percentage of the queries that is answered correctly in some layer $\leq i$ and the percentage of the queries that is stopped after layer i (i.e., $\neg L_i(s,t)$).

			layer 1 [%]		layer 2 [%]		layer 3 [%]		
	metric	variant	correct	stopped	correct	stopped	correct	stopped	query time
USA	time	eco	99.86	98.87	99.9936	99.7220	–	–	11.5 μs
		gen	99.89	99.20	99.9986	99.9862	99.99986	99.99984	4.9 μs
	dist	eco	98.43	91.90	99.9511	97.7648	–	–	87.5 μs
EUR	time	eco	99.46	97.13	99.9908	99.4157	–	–	13.4 μs
		gen	99.74	98.65	99.9985	99.9810	99.99981	99.99972	5.6 μs
	dist	eco	95.32	81.68	99.8239	95.7236	–	–	107.4 μs

The remainder of this section refers to the travel time metric. Since the overwhelming majority of all cases are handled in the top layer (about 99% in case of the US network), the average case performance says little about the performance for more local queries which might be very important in applications. Therefore we use the method developed in [5] to get more detailed information about the query time distributions for queries ranging from very local to global. Figure 3 gives for each variant (economical/generous) and for each value r on the x-axis a distribution for 1 000 queries with random starting point s and the target node t with Dijkstra rank $\mathrm{rk}_s(t) = r$. The distributions are represented as box-and-whisker plots [22]: 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. (Appendix A contains analogous figures for the European network with the travel time metric and for both networks with the distance metric.)

For the generous approach, we can easily recognise the three layers of transit node routing with small transition zones in between: For ranks 2^{18}–2^{24} we usually have $\neg L(s,t)$ and thus only require cheap distance table accesses in layer 1. For ranks 2^{12}–2^{16}, we need additional look-ups in the table of layer 2 so that the queries get somewhat more expensive. In this range, outliers can be considerably more costly, indicating that occasional local searches are needed. For small ranks we usually need local searches and additional look-ups in the table of layer 3. Still, the combination of a local search in a very small area and table look-ups in all three layers usually results in query times of only about 20 μs.

In the economical approach, we observe a high variance in query times for ranks 2^{15}–2^{16}. In this range, all types of queries occur and the difference between the layer-1 queries and the local queries is rather big since the economical variant does not make use of a third layer. For smaller ranks, we see a picture very similar to basic highway hierarchies with query time growing logarithmically with Dijkstra rank.

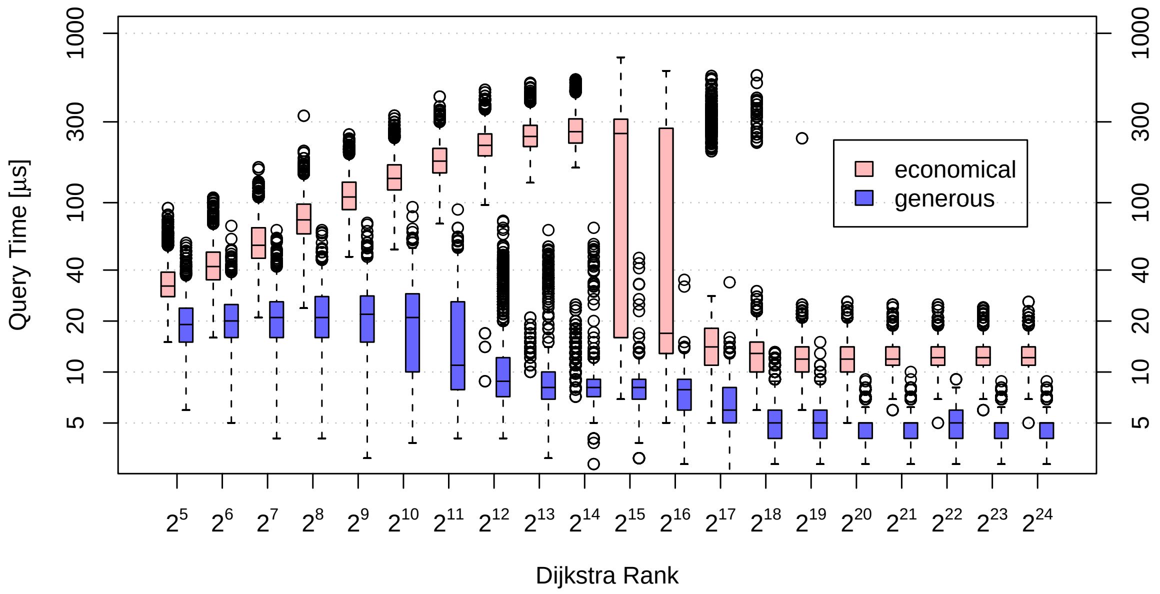

Fig. 3. Query times for the USA with the travel time metric as a function of Dijkstra rank.

4.3 Complete Description of the Shortest Path

For a given node pair (s, t), in order to get a complete description of the shortest s-t-path, we first perform a transit node query and determine the layer i that is used to obtain the shortest path distance. Then, we have to determine the path from s to the forward access point u to layer i, the path from the backward access point v to t, and the path from u to v. In case of a local query, we can fall back on [21].

Currently, we provide an efficient implementation only for the case that the path goes through the top layer. In all other cases, we just perform a normal highway search and invoke the methods from [21]. The effect on the average times is very small since more than 99% of the queries are correctly answered using only the top search (in case of the travel time metric; cp. Tab. 4).

When a node s and one of its access points u are given, we can determine the next node on the shortest path from s to u by considering all adjacent nodes s' of s and checking whether $d(s, s') + d(s', u) = d(s, u)$. In most cases, the distance $d(s', u)$ is directly available since u is also an access point of s'. In a few cases—when u is not an access point of s'—, we have to consider all access points u' of s' and check whether $d(s, s') + d(s', u') + d(u', u) = d(s, u)$. Note that $d(u', u)$ can be looked up in the top distance table. Using this subroutine, we can determine the path from s to the forward access point u and from the backward access point v to t.

A similar procedure can be used to find the path from u to v (cp. [21]). However, in this case, we consider only adjacent nodes u' of u that belong to the top layer as well because only for these nodes we can look up $d(u', v)$. Since there are shortest paths between top layer nodes that leave the top layer—we call such paths *hidden paths*—, we execute an additional preprocessing step that determines all hidden paths and stores them in a special data structure (after the used shortcuts have been expanded). Whenever we cannot find the next node on the path to v considering only adjacent nodes in the top layer, we look for the right hidden path that leads to the next node in the top layer.

In order to unpack the used shortcuts (i.e., determine the subpaths in the original graph that correspond to the shortcuts), we use the method from [21, Variant 3]. In Tab. 5 we give the additional preprocessing time and the additional disk space for the hidden paths and 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[3] it summing up the weights of all edges as a sanity check—assuming that the distance query has already been performed. That means that the total average time to determine a shortest path is the time given in Tab. 5 plus the query time given in Tab. 4.

Table 5. 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. These figures refer to experiments on the graphs with the travel time metric using the generous variant.

	preproc. [min]	space [MB]	query [μs]	# hops (avg.)
USA	4:04	193	258	4 537
EUR	7:43	188	155	1 373

5 Conclusions and Future Work

We have demonstrated that query times for quickest paths in road networks can be reduced by another two orders of magnitude compared to the best previous techniques—highway hierarchies and reach based routing. Building on highway hierarchies, this can be achieved using a moderate amount of additional storage and precomputation. Paradoxically, the biggest problem for the application of transit node routing may be that it is far too fast for classical route planning. Already the previous best techniques had query time comparable to the time needed for just traversing the quickest path, let alone communicating or drawing it. Still, in applications like traffic simulation or optimisation problems in logistics, we may need a huge number of shortest path distances and only few actual shortest paths. We also consider the proof that few access nodes suffice for all long distance quickest paths to be an interesting insight into the structure of road networks.

Although conceptually simple, an efficient implementation of transit node routing has so many ingredients that there are many further optimisations opportunities and a large spectrum of trade-offs between query time, preprocessing time, and space usage. For reducing the average query time, we could try to precompute information analogous to edge flags or geometric containers [16, 17, 15] that tells us which access nodes lead to which regions of the graph.

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

There are many interesting ways to choose transit nodes. For example nodes with high node reach [9, 10] could be a good starting point. Here, we can directly influence $|\mathcal{T}|$, and the resulting reach bound might help defining a simple locality filter. However, it seems that geometric reach or travel time reach do not reflect the inhomogeneous density of real world road networks. Hence, it would be interesting if we could efficiently approximate reach based on the Dijkstra rank.

Another interesting approach might be to start with some locality filter that guarantees uniformly small local searches and then to view it as an optimisation problem to choose a small set of transit nodes that cover all the local search spaces.

Parallel processing can easily be used to accelerate preprocessing, or to execute many queries in parallel. With very fine grained multi-core parallelism it might even be possible to accelerate an individual query. Forward local search, backward local search, and each table look-up are largely independent of each other.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
2. Bast, H., Funke, S., Matijevic, D.: Ultrafast shortest-path queries with linear-time preprocessing. 9th DIMACS Implementation Challenge (2006)
3. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: 14th European Symposium on Algorithms (ESA). Volume 4168 of LNCS., Springer (2006) 804–816
4. Müller, K.: Design and implementation of an efficient hierarchical speed-up technique for computation of exact shortest paths in graphs. Master's thesis, Universtät Karlsruhe (2006) supervised by D. Delling, M. Holzer, F. Schulz, and D. Wagner.
5. 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
6. Thorup, M.: Compact oracles for reachability and approximate distances in planar digraphs. In: 42nd IEEE Symposium on Foundations of Computer Science. (2001) 242–251
7. 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
8. 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
9. 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
10. 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
11. 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**(2) (1968) 100–107
12. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms. (2005) 156–165
13. Goldberg, A.V., Werneck, R.F.: An efficient external memory shortest path algorithm. In: Workshop on Algorithm Engineering and Experimentation. (2005) 26–40
14. Wagner, D., Willhalm, T.: Drawing graphs to speed up shortest-path computations. In: 7th Workshop on Algorithm Engineering and Experiments. (2005)
15. 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
16. 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
17. 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
18. 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) to appear, `http://algo2.iti.uka.de/schultes/hwy/`.
19. U.S. Census Bureau, Washington, DC: UA Census 2000 TIGER/Line Files. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html` (2002)

20. 9th DIMACS Implementation Challenge: Shortest Paths. `http://www.dis.uniroma1.it/~challenge9/` (2006)
21. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway hierarchies star. 9th DIMACS Implementation Challenge (2006)
22. R Development Core Team: R: A Language and Environment for Statistical Computing. `http://www.r-project.org` (2004)

A Further Experiments

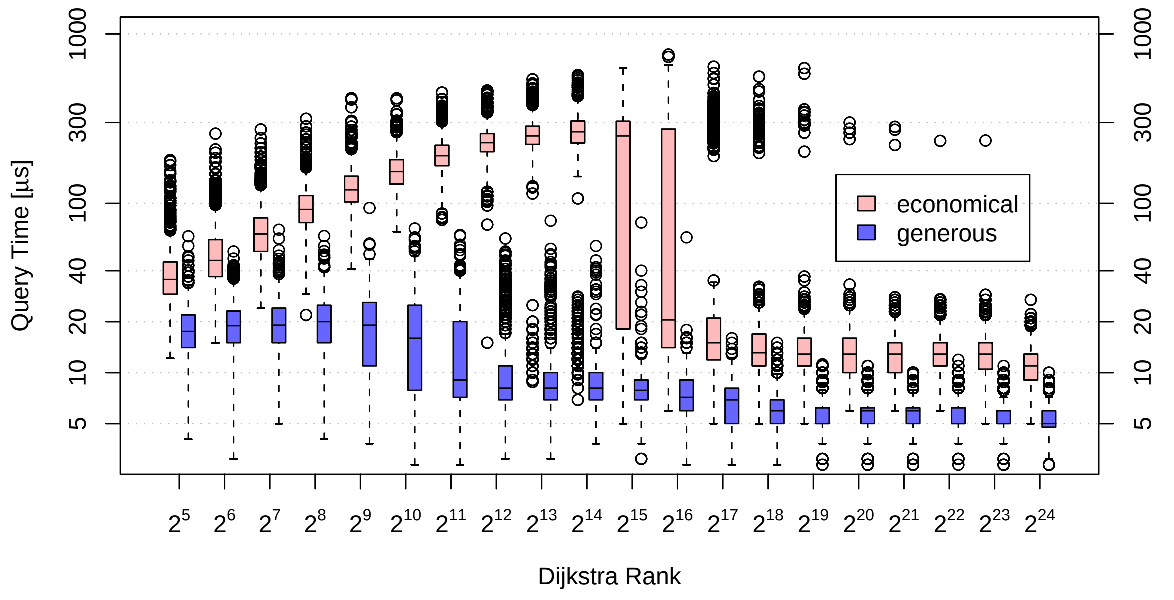

Fig. 4. Query times for Europe with the travel time metric as a function of Dijkstra rank.

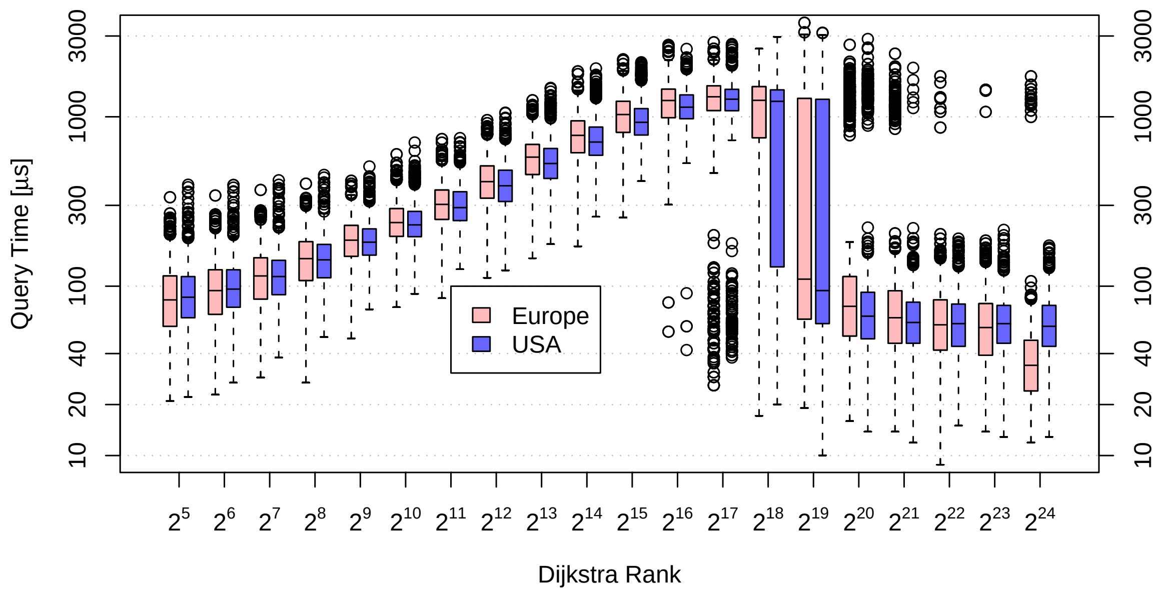

Fig. 5. Query times for the distance metric as a function of Dijkstra rank.