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. Recently, other research groups and we have largely independently developed this informal observation into *transit node routing*, a technique for reducing quickest-path queries in road networks to a small number of table lookups. The contribution of our paper is twofold. First, we present a generic framework for transit node routing that allows almost constant time routing for both global and local queries. Second, we develop a highly tuned implementation 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 five 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.

In Section 2, we develop a generic framework for *transit node routing*, which is based on two key observations: First, there is a relatively small set of *transit nodes*—about 10 000 for the Western European or 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. When distances from all nodes to their respective access nodes and between all transit nodes have been precomputed, a 'non-local' shortest-path query can be reduced to a few table lookups. An important ingredient is a *locality filter* that decides whether source and target are too close so that we need a special treatment to guarantee the correct result. In order to handle such local queries more efficiently, we add further *layers* to the basic approach.

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

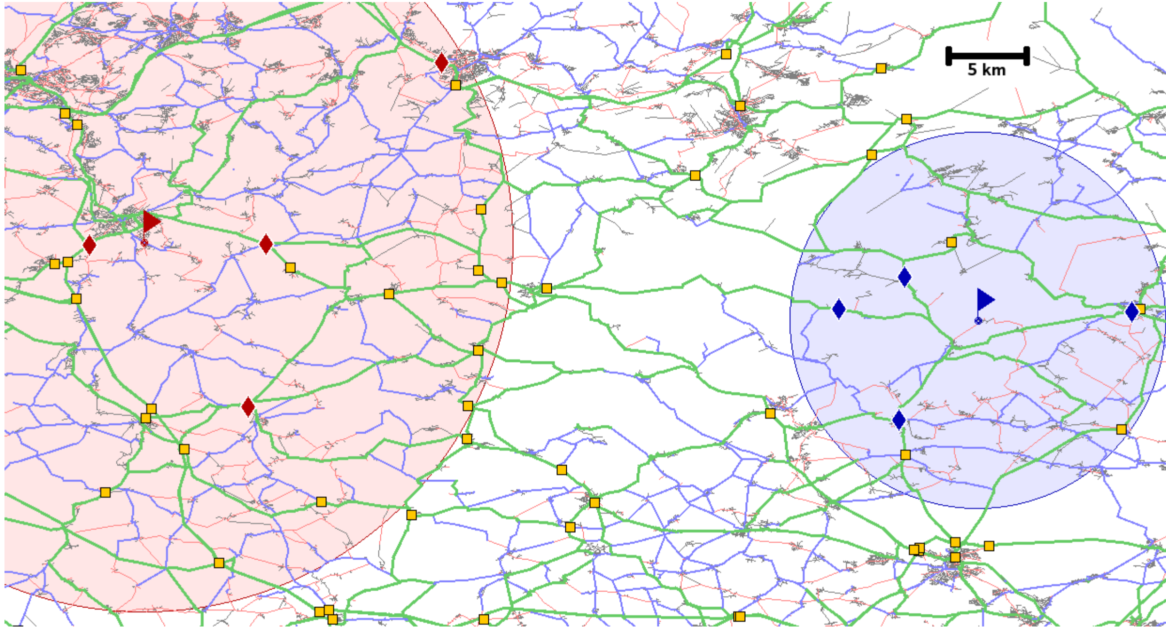

Fig. 1. Finding the optimal travel time between two points somewhere between Saarbrücken and Karlsruhe amounts to retrieving the 2×4 *access nodes* (diamonds), performing 16 table lookups between all pairs of access nodes, and checking that the two disks defining the *locality filter* do not overlap. 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.

Transit node routing can be instantiated in many ways. In Section 3, we present one particular instantiation, which is based on *highway hierarchies* [2, 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 main focus is on computing *quickest*-path[1] lengths. However, we also give some results on outputting a complete description of the quickest path and 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.

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 [2, 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 $\langle s, \ldots, u, v, \ldots t\rangle$ 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

[1] Note that we often use the term '*shortest* path' as a synonym for 'quickest path'.

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* [4, 5].

Using 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 nodes are computed online and that only distances within level K of the highway hierarchy (rather than distances in the underlying graph) are precomputed. The latter leads to considerably larger sets of access nodes (≈ 55 instead of 10) 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.

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 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. Independently from our work, Müller et al. have essentially developed this approach, using different terminology[2]. Note that their first results [9] were published before any other implementation of transit node routing. However, it took some time till reliable measurement data were available[3] [10]. 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 about

[2] We chose to interpret their work using the transit node terminology in order to point out similarities to our work.

[3] In their implementation, the preprocessed data is stored on a hard disk. Using a more compact representation, the data would fit into main memory. Therefore, when measuring query times, it is justifiable to assume that the required data was in main memory. This situation makes performing experiments more difficult.

4–8 times as many access nodes as our scheme (depending on the used metric) leading to much higher preprocessing times. 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 approach, in particular in case of the travel time metric.

Grid-Based Transit Node Routing. Bast, Funke and Matijevic proposed the transit node routing approach based on a geometric grid [11]: The network is subdivided into uniform cells. Border nodes of these cells that are needed for 'long-distance' travel are used as access nodes. The union of all access nodes forms the transit node set. As a locality filter it is sufficient to check whether source and target lie a certain number of cells apart.

They were the first to explicitly formulate the central observations and concepts of transit node routing[4]. Our work was completed a few weeks later and has been accomplished largely independently from theirs except for the fact that their observation that about ten access nodes per node were sufficient motivated us to rethink our access node definition leading to a considerable reduction from around 55 to about ten, which made an implementation for large graphs much more practicable, accelerated our development process significantly and yielded very good query times. While most algorithms described in [11] cater to the specific grid-based approach, we prefer a more generic notion of transit node routing and regard our highway-hierarchy-based implementation only as one possible (and very successful) instantiation of transit node routing.

In a joint paper [12], both implementations are contrasted. One noticeable difference is that we deal with all types of queries in a highly efficient way, while the grid-based variant only answers non-local queries very quickly (which, admittedly, constitute a very large fraction of all queries if source and target are picked uniformly at random). The grid-based variant is designed for comparatively modest memory requirements, while our highway-hierarchy-based implementation has significantly smaller preprocessing and average query times. Note that our implementation would need considerably less memory if we concentrated only on undirected graphs and non-local queries as it is done in the grid-based implementation.

Computing Distance Tables. For given source and target node sets, a table containing the distances between all source-target node pairs can be computed very efficiently using a many-to-many shortest path algorithm [13] based on highway hierarchies. The development of this algorithm was another step on the way from the highway hierarchies enhanced by a distance table to transit node routing since it allowed to compute distances in the original graph between all level-K nodes of the highway hierarchy.

Geometry. 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. [15] by [16, 17] or [18] by [19, 20]). 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

[4] In particular, they introduced the term 'transit node'. In a joint paper [12], we adopted some formulations and terms from [11] to describe the generic approach. For the sake of simplicity, we decided to keep these phrases in this paper.

performance. Arriving at this observation was our final step to a fully functional version of transit node routing.

Goal Direction. Another interesting property of road networks is that they allow effective goal directed search using **A*** *search* [15]: 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 [16, 17] using the **T**riangle inequality (*ALT*). In combination with reach based routing, this is one of the fastest known speedup techniques [5]. 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.

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* $A : V \to 2^{\mathcal{T}}$ that maps a vertex to its access node set, and a *locality filter* $L : V \times V \to \{\mathsf{true}, \mathsf{false}\}$ that decides whether an s-t-query is a 'local query' or not. We require that $\neg L(s,t)$ implies that the shortest path distance is

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

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.

Figure 2 gives a schematic representation of transit node routing. 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.

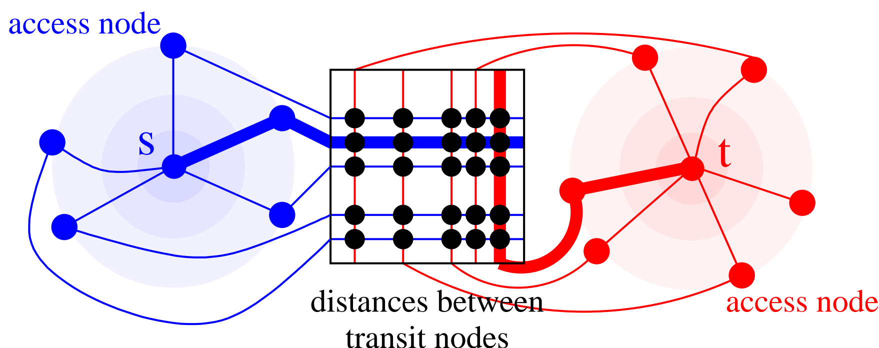

Fig. 2. Schematic representation of transit node routing.

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 $A_2 : V \to 2^{\mathcal{T}_2}$, 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

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

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.

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.

2.1 Preliminaries

During a Dijkstra search from some node s, we say that a settled node u is *covered* by a node set V' if there is at least one node $v \in V'$ on the path from the root s to u. A reached but not settled node is *covered* if its tentative parent is covered. The current partial shortest-path tree B is *covered* if all currently reached but not settled nodes (i.e., all nodes in the priority queue) are covered. All nodes $v \in V' \cap B \setminus \{s\}$ whose parent in B is not covered are *covering nodes*. In addition, the root s is a covering node if $s \in V'$.

2.2 Computing Access Nodes: Backward Approach

From each transit node $v \in \mathcal{T}$, run a Dijkstra search[5], until the partial shortest-path tree B is covered by $\mathcal{T} \setminus \{v\}$. For any non-covered node u in B, record v as an access node for u, and for any covering node w, record an edge (v,w) with weight $d(v,w)$ for a *transit graph* $G[\mathcal{T}] = (\mathcal{T}, E_{\mathcal{T}})$. Figure 3 gives an example. 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 $G[\mathcal{T}]$.

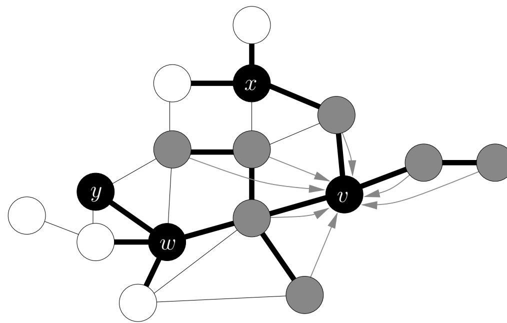

Fig. 3. Example for the backward approach to the computation of access nodes. Edge weights correspond to the lengths of the drawn line segments. The black nodes belong to $\mathcal{T}$. The search is started from v. All thick edges belong to the partial shortest-path tree. The non-covered nodes are highlighted in grey: for these nodes, v is an access node (suggested by the arrows pointing to v). Note that in this example x and w (but not y) are covering nodes.

[5] Note that in a *directed* graph, we would perform a *backward* search, i.e., a search in the reverse graph, which explains the name of this approach.

Layer-2 Information is computed similarly to the top-layer information. However, in this case, we do not have to compute a complete distance table, but it is sufficient to store only distances that actually improve on the distances obtained going via the top layer $\mathcal{T}$. This can be done space efficiently in a static hash table. In order to compute the required distances, for each node $v \in \mathcal{T}_2$, the single-source shortest-path search from v in $G[\mathcal{T}_2]$ can be stopped as soon as the partial shortest-path tree is covered by $\mathcal{T} \setminus \{v\}$.

2.3 Computing Access Nodes: Forward Approach

Start a Dijkstra search from each node u. Stop when the partial shortest-path tree is covered by the transit node set $\mathcal{T}$. Take the covering transit nodes as access nodes 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 a computation of a superset of the access nodes. Fortunately, this set can be easily reduced if the distances between all transit nodes are already known: if an access node y can be reached from u via another access node w on a shortest path, we can discard y. Figure 4 gives an example. Second, we can only determine the access node 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.

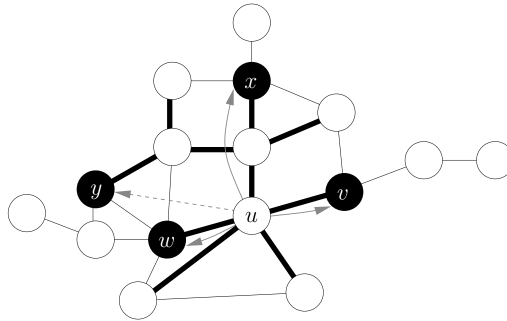

Fig. 4. Example for the forward approach to the computation of access nodes including the first, but not the second 'trick'. Edge weights correspond to the lengths of the drawn line segments. The black nodes belong to $\mathcal{T}$. The search is started from u. All thick edges belong to the search tree. The nodes v, w, x, and y are covering nodes. However, y can be removed from this set since the path from u via w to y turns out to be shorter than the path that has been found. Thus, u has only three access nodes.

2.4 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., [11]). 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).

In the latter case, 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 specify a *geometric* locality filter in the following way. For each node

$u \in \mathcal{T}_{i+1}$, we compute the radius $r^i(u)$ of a circle around u 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. Then, for nodes $u, v \in \mathcal{T}_{i+1}$, the locality filter is defined such that $L_i(u,v)$ is true iff the cirles around u and v touch or intersect. It is easy to see that this definition complies with the requirements formulated at the beginning of this section: if $d(u,v)$ cannot be computed using layers $\leq i$, then there will be a corresponding entry in the layer-$(i+1)$ distance table, which implies that both the circle around u and the circle around v contain the meeting point of a bidirectional search between u and v; thus, both circles touch or intersect so that $L_i(u,v)$ is true.

This locality filter can be extended to work for all nodes by (pre)computing conservative circle radii for arbitrary nodes v as $r^i(v) := \max\{||v-u||_2 + r^i(u) : u \in A_{i+1}(v)\}$, where $||v-u||_2$ denotes the Euclidean distance between u and v (Fig. 5). 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 effective locality information.

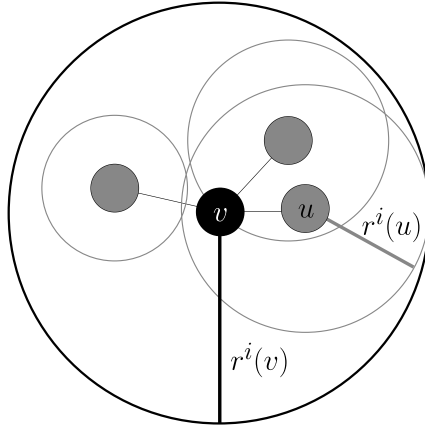

Fig. 5. Example for the extension of the geometric locality filter. The grey nodes constitute the set $A_{i+1}(v)$.

2.5 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.

2.6 Outputting Complete Descriptions of the Shortest Paths

Generally, in a graph with bounded degree (e.g., a road network) using 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,s') such that $d(s,s') + d(s',t) = d(s,t)$, output (s,s'). Continue by looking for a shortest path from s' to t. Repeat until t is reached.

In the special case of transit node routing, we can speed up this process by two measures. Suppose the shortest path uses the access nodes $u \in A(s)$ and $v \in A(t)$. First, while

reconstructing the path from s to u, we can determine the next hop by considering all adjacent nodes s' of s and checking whether $d(s,s') + d(s',u) = d(s,u)$. Usually[6], the distance $d(s',u)$ is directly available since u is also an access node of s'. Analogously, the path from v to t can be determined.

Second, reconstructing the path from u to v 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 that 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

3.1 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 $\langle s, \dots, u, v, \dots, t\rangle$ 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 $G_0, G_1, G_2, \dots, G_L$. 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.

3.2 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. For further layers, we use (the cores of) lower levels of the highway hierarchy. Note that there is a difference between the term 'level' (of the highway hierarchy) and the term 'layer' (of transit node routing).

[6] In a few cases—when u is not an access node of s' (which can only happen if the shortest paths in the graph are not unique)—, we have to consider all access nodes 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.

3.3 Access Nodes and Distance Tables

We use our highway hierarchy based code for many-to-many routing to compute the top level distance table [13]. 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 $i > 1$. As a byproduct of the distance table computations, we obtain geometric locality filters as described in Section 2.4.

We use the forward approach from Section 2.3 to compute the access node sets. (In our case, we do not perform Dijkstra searches, but highway searches [3].)

Figure 6 summarises the setup 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 two layers and reconstructs the access node set and the locality filter needed for the layer-1 query using information only stored with nodes in $\mathcal{T}_2$, i.e., for a layer-1 query with source node s, we build the union $\bigcup_{u \in A_2(s)} A(u)$ of all layer-1 access nodes of all layer-2 access nodes of s to determine on-the-fly a layer-1 access node set for s. Similarly, a layer-1 locality filter for s is built using the locality filters of the layer-2 access nodes (cp. Section 2.4). 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.

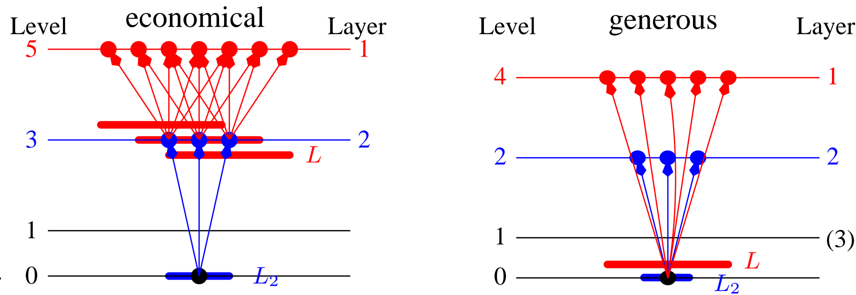

Fig. 6. Representations of information relevant to highway hierarchy transit node routing. The chosen settings refer to the travel time metric. Note that we use the *cores* of the given levels as transit node sets.

3.4 Queries

Queries are performed in a top-down fashion. For a given query pair (s, t), first $A(s)$ and $A(t)$ are either looked up or computed (cp. Section 3.3) depending on the used variant. Then table lookups 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 bound computed at layers 1 and 2. Furthermore,

the search need not expand from any node $u \in \mathcal{T}_2$ 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 node set for layer 3. Additional lookups in the layer-3 table ensure the correctness of this variant.

3.5 Outputting Complete Descriptions of the Shortest Paths

The general methods from Section 2.6 can be applied rather directly to the highway-hierarchy-based implementation in order to determine a complete description of the shortest path. In case of a local query, we can fall back on the routines used in the highway hierarchies approach [21].

In order to unpack the used shortcuts[7] (i.e., determine the subpaths in the original graph that correspond to the shortcuts), we use a rather sophisticated data structure to represent unpacking information for the shortcuts in a space-efficient way. In particular, we do not store a sequence of node IDs that describe a path that corresponds to a shortcut, but we store only *hop indices*: for each edge (u, v) on the path that should be represented, we store its index minus the index of the first edge of u. Since in most cases the degree of a node is very small, these hop indices can be stored using only a few bits. The unpacked shortcuts are stored in a recursive way, e.g., the description of a level-2 shortcut may contain several level-1 shortcuts. Accordingly, the unpacking procedure works recursively.

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

4 Experiments

4.1 Environment, Instances, and Parameters

The experiments were done on one core of a single AMD Opteron Processor 270 clocked at 2.0 GHz with 8 GB main memory and 2×1 MB L2 cache, running SuSE Linux 10.0 (kernel 2.6.13). The program was compiled by the GNU C++ compiler 4.0.2 using optimisation level 3. Benchmark results can be found in Tab. 6 in Appendix A.

We deal with two road networks. The network of Western Europe[8] 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 variants of the European graph with a *distance metric* and the *unit metric*. The network of the USA (without Alaska and Hawaii) has been obtained from the TIGER/Line Files [22]. Again, we consider only the largest strongest connected component. In contrast to the PTV data, the TIGER graph is

[7] Here, we do not only mean the shortcut edges between source/target and the respective access node, but also the edges of the transit graph that lie on the shortest path from the forward to the backward access node.

[8] 14 countries: Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the UK

undirected, planarised and distinguishes only between four road categories. All graphs[9] have been taken from the DIMACS Challenge website [23]. 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 [2, 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) and the generous (gen) variant. For the distance (dist) and unit metrics, we use linearly increasing sequences for c and H.

Table 2. Parameters. Note that we use the *cores* of the given levels as transit node sets.

metric	time		dist	unit
variant	eco	gen	eco	eco
levels of layers 1–2(–3)	5–3	4–2–1	6–4	5–3
neighbourhood size H	60	110	90, 180, 270, ...	80, 100, 120, ...
contraction rate c	1.5	1.5	1.5, 1.6, 1.7, ...	1.5, 1.6, 1.7, ...

4.2 Main Results

Preprocessing. Table 3 gives the preprocessing times for both road networks and all three metrics; 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

[9] Note that the experiments on the full 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 except for a slight increase of the memory consumption since more bits are needed to store certain path lengths.

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 nodes than the travel time metric (a factor 2–3) since not only junctions on very fast roads (which are rare) qualify as access nodes. 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.

The unit metric ranks somewhere in between. Although computing shortest paths in road networks based on the unit metric seems kind of artificial, we observe a hierarchy in this scenario as well: when we drive on urban streets, we encounter much more junctions than driving on a national road or even a motorway; thus, the number of road segments on a path is somewhat correlated to the road type. This explains why the performance of the unit metric does not strongly deviate from the travel time 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 nodes to the layer. 'Space' is the total *overhead* of our approach.

	metric	variant	layer 1 $\|\mathcal{T}\|$	layer 1 $\|$table$\|$ [$\times 10^6$]	layer 1 $\|A\|$	layer 2 $\|\mathcal{T}_2\|$	layer 2 $\|$table$_2\|$ [$\times 10^6$]	layer 2 $\|A_2\|$	layer 3 $\|\mathcal{T}_3\|$	layer 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
	unit	eco	13 329	178	8.7	136 546	39	6.0	–	–	121	1:32
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
	unit	eco	2 488	6	13.0	86 928	77	7.7	–	–	123	1:25

Random Queries Using the Travel Time Metric. 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. Further experiments on various subgraphs of the US road network (see Tab. 7 in Appendix A) support our claim that we achieve almost constant query times irrespective of the size of the road network: while the sizes range between 264 346 and 23 947 347 nodes, the query times vary only from 3.7 to 5.0 μs for the generous variant.

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%.

Random Queries Using the Distance Metric. 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 [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 are not answered correctly in some layer $\leq i$ and the percentage of the queries that are not stopped after layer i (i.e., $L_i(s,t)$).

			layer 1 [%]		layer 2 [%]		layer 3 [%]		
	metric	variant	wrong	cont'd	wrong	cont'd	wrong	cont'd	query time
USA	time	eco	0.14	1.13	0.0064	0.2780	–	–	11.5 μs
		gen	0.11	0.80	0.0014	0.0138	0.00014	0.00016	4.9 μs
	dist	eco	1.57	8.10	0.0489	2.2352	–	–	87.5 μs
	unit	eco	0.42	2.15	0.0115	0.4800	–	–	18.7 μs
EUR	time	eco	0.54	2.87	0.0092	0.5843	–	–	13.4 μs
		gen	0.26	1.35	0.0016	0.0190	0.00019	0.00028	5.6 μs
	dist	eco	4.68	18.32	0.1761	4.2764	–	–	107.4 μs
	unit	eco	1.87	11.52	0.0204	2.2199	–	–	23.1 μs

Local Queries Using 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 some applications. Therefore we use the method developed in [2] to get more detailed information about the query time distributions for queries ranging from very local to global. Figure 7 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 $\text{rk}_s(t) = r$. The distributions are represented as box-and-whisker plots [24]: 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 lookups 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 lookups in the table of layer 3. Still, the combination of a local search in a very small area and table lookups 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.

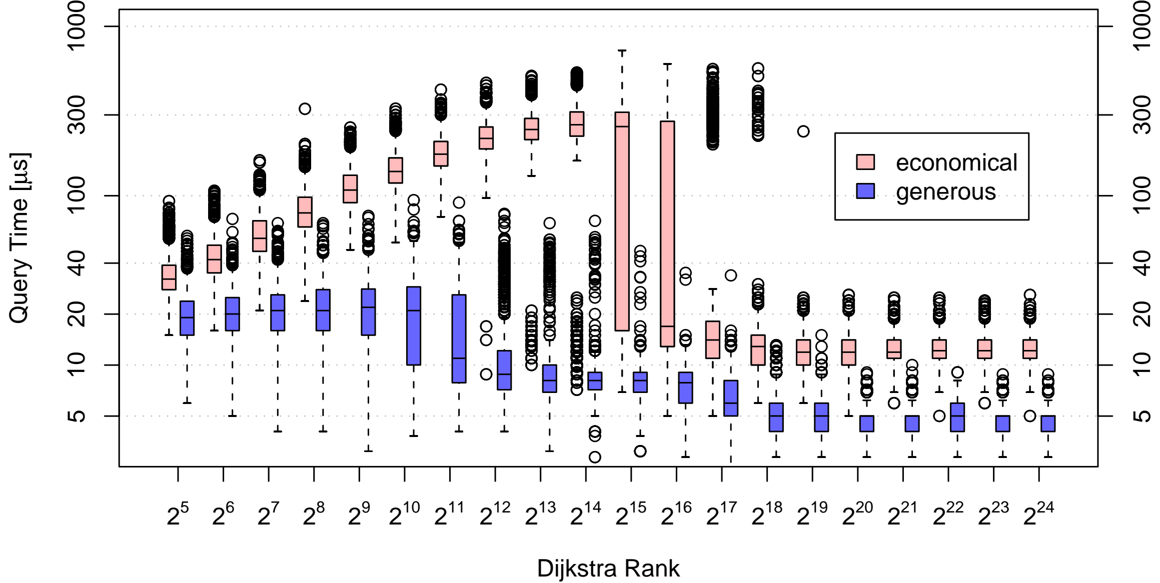

Fig. 7. Query times for the USA with the travel time metric as a function of Dijkstra rank.

4.3 Outputting Complete Descriptions of the Shortest Paths

Table 5 deals with the traversal of a complete description of the shortest path based on the method described in Section 3.5. 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).

We give the additional preprocessing time and the additional disk space for the unpacking data structures. Furthermore, we report the additional time that is needed to determine a complete description of the shortest path and to traverse[10] 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

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

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 a few actual shortest paths.

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 example, in order to reduce the latter, we could apply the generic method from Section 2.5 to our highway-hierarchy-based implementation: when access node information is stored only at the core of level 1, the size of the access node data will be reduced by a factor of six. For reducing the average query time, we could try to precompute information analogous to edge flags or geometric containers [19, 20, 18] that tells us which access nodes lead to which regions of the graph.

There are many interesting ways to choose transit nodes. For example nodes with high node reach [4, 5] 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 lookup are largely independent of each other.

Acknowledgements

We would like to thank Holger Bast, Stefan Funke, Kirill Müller, and Dorothea Wagner for interesting discussions on transit node routing and Timo Bingmann for work on visualisation tools.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
2. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: 13th European Symposium on Algorithms. Volume 3669 of LNCS., Springer (2005) 568–579
3. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: 14th European Symposium on Algorithms. Volume 4168 of LNCS., Springer (2006) 804–816

4. 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
5. 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
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. 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.
10. Delling, D., Holzer, M., Müller, K., Schulz, F., Wagner, D.: High-performance multi-level graphs. In: 9th DIMACS Implementation Challenge [23]. (2006)
11. Bast, H., Funke, S., Matijevic, D.: TRANSIT—ultrafast shortest-path queries with linear-time preprocessing. In: 9th DIMACS Implementation Challenge [23]. (2006)
12. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In transit to constant time shortest-path queries in road networks. In: Workshop on Algorithm Engineering and Experiments. (2007)
13. 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)
14. Wagner, D., Willhalm, T.: Drawing graphs to speed up shortest-path computations. In: 7th Workshop on Algorithm Engineering and Experiments. (2005)
15. 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
16. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms. (2005) 156–165
17. Goldberg, A.V., Werneck, R.F.: An efficient external memory shortest path algorithm. In: Workshop on Algorithm Engineering and Experimentation. (2005) 26–40
18. 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
19. 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
20. 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
21. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway hierarchies star. In: 9th DIMACS Implementation Challenge, `http://www.dis.uniroma1.it/~challenge9/`. (2006)
22. U.S. Census Bureau, Washington, DC: UA Census 2000 TIGER/Line Files. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html` (2002)
23. 9th DIMACS Implementation Challenge: Shortest Paths. `http://www.dis.uniroma1.it/~challenge9/` (2006)
24. R Development Core Team: R: A Language and Environment for Statistical Computing. `http://www.r-project.org` (2004)

A Further Experiments

Table 6. DIMACS Challenge [23] benchmarks for US (sub)graphs (query time [ms]).

graph	metric time	metric dist
NY	29.6	28.5
BAY	34.7	33.3
COL	51.5	49.0
FLA	134.8	120.5
NW	161.1	146.1
NE	225.4	197.2
CAL	291.1	235.4
LKS	461.3	366.1
E	681.8	536.4
W	1 211.2	988.2
CTR	4 485.7	3 708.1
USA	5 355.6	4 509.1

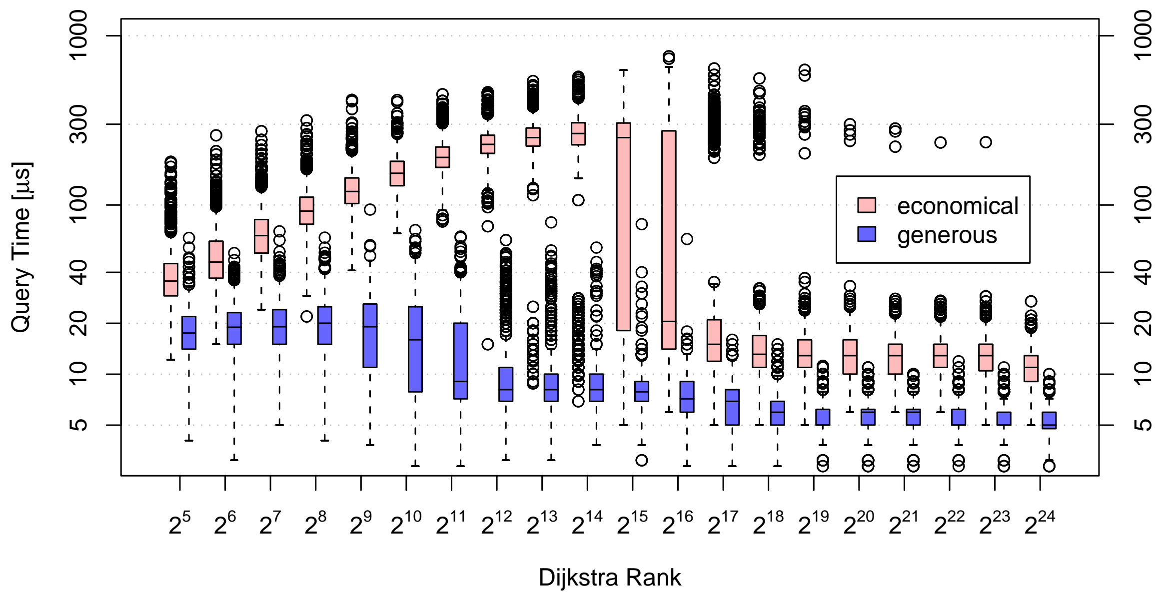

Fig. 8. Query times for Europe with the travel time metric as a function of Dijkstra rank.

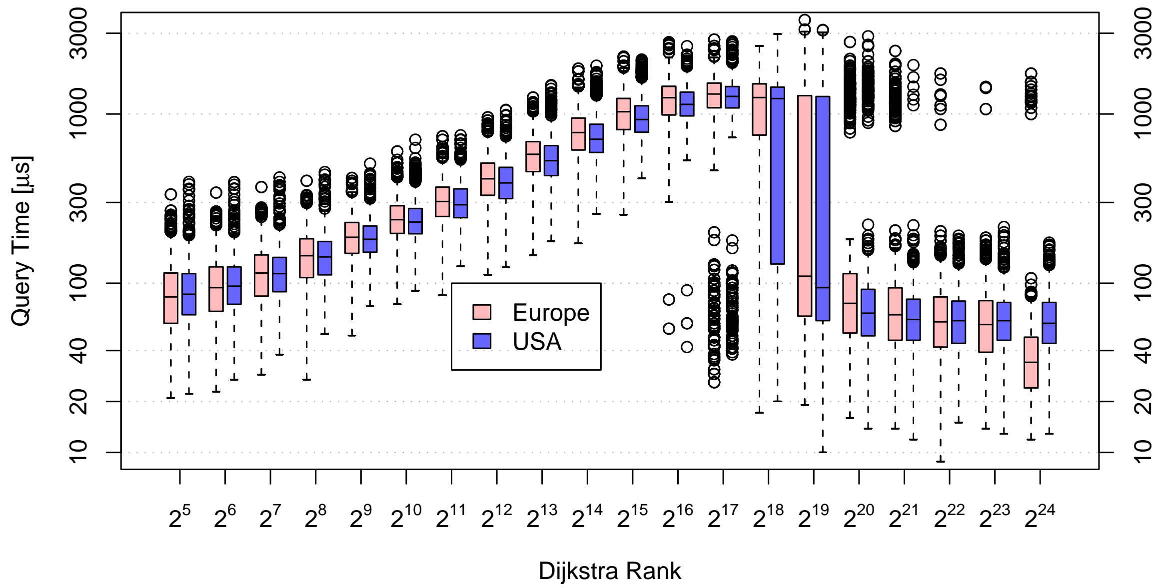

Fig. 9. Query times for the distance metric as a function of Dijkstra rank.

Table 7. Results for US subgraphs with travel time metric using the generous variant.

graph	#nodes	preproc. time [min]	total disk space [MB]	query time [μs]
NY	264 346	4	147	4.6
BAY	321 270	2	105	4.2
COL	435 666	3	156	4.7
FLA	1 070 376	7	418	3.8
NW	1 207 945	7	325	3.7
NE	1 524 453	16	578	4.1
CAL	1 890 815	15	554	3.8
LKS	2 758 119	26	890	4.2
E	3 598 623	30	1 159	4.4
W	6 262 104	47	1 801	4.2
CTR	14 081 816	148	4 169	5.0
USA	23 947 347	205	6 108	4.9