In Transit to Constant Time Shortest-Path Queries in Road Networks*

Holger Bast† Stefan Funke† Domagoj Matijevic† Peter Sanders‡
Dominik Schultes‡

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 an algorithmic approach—*transit node routing*—that allows us to reduce quickest-path queries in road networks to a small number of table lookups. We present two implementations of this idea, one based on a simple grid data structure and one based on *highway hierarchies*. For the road map of the United States, our best query times improve over the best previously published figures by two orders of magnitude. Our results exhibit various trade-offs between average query time ($6\,\mu s$ to $63\,\mu s$), preprocessing time (62 min to 1200 min), and storage overhead (27 bytes/node to 247 bytes/node).

1 Introduction

Computing optimal routes in a road network $G = (V, E)$ is one of the showpieces of real-world applications of algorithmics. The classical way to compute the shortest path between two given nodes in a graph with given edge lengths is Dijkstra's algorithm [4]. Its asymptotic running time is $O(m + n \log m)$, where n is the number of nodes, and m is the number of edges. For graphs with constant degree, like the road networks we consider, this is $O(n \log n)$. Our benchmark throughout this paper is the US road network [22], which has about 24 million nodes and 58 million edges. On this network, Dijkstra's algorithm takes more than a second on a state-of-the-art workstation to compute the shortest path between two random nodes. This is too slow for many applications.

While it is still an open question, whether Dijktra's algorithm is optimal for single-source single-target queries in general graphs, there is an obvious $\Omega(n + m)$ lower bound, because every node and every edge has to be looked at in the worst case. Sublinear query time hence requires some form of preprocessing of the graph. For general graphs, constant query time can only be achieved with superlinear space requirement; this is due to a recent result by Thorup and Zwick [21]. Like previous works, we therefore exploit special properties of road networks, in particular, that the nodes have low degree and that there is a certain hierarchy of more and more important roads, such that further away from source and target the more important roads tend to be used on shortest paths.

Our approach, coined *transit node routing*, is based on the following two key observations on large road networks. First, there is a relatively small set of what we call *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 for the US road network. The corresponding intuition is illustrated in Figure 1.

The idea behind transit node routing is now to precompute distances between transit nodes and distances from all nodes to their access nodes. Together with an effective notion of 'sufficiently far away' this allows most queries to be answered using only a few table lookups. Our fastest query times of $6\,\mu s$ improve over the best previously published numbers by *two orders of magnitude.*

The structure of the paper is as follows: After reviewing related work in Section 2, we introduce transit node routing more formally in Section 3. In particular, we introduce several *layers* of more and more local transit nodes that are required to achieve good performance for all possible queries.

In Section 4 we describe a simple grid-based implementation of these techniques tuned for space efficiency and provide an experimental evaluation thereof. In Section 5 we give a more sophisticated implementation based on highway hierarchies that achieves the fastest

*Partially supported by DFG grant SA 933/1-3, by the EU within the 6th Framework Programme under contract 001907 "Dynamically Evolving Large Scale Information Systems" (DELIS), and by the Max Planck Center for Visual Computing and Communication (MPC-VCC) funded by the German Federal Ministry of Education and Research (FKZ 01IMC01).

†Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany, {bast,funke,dmatijev}@mpi-inf.mpg.de

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

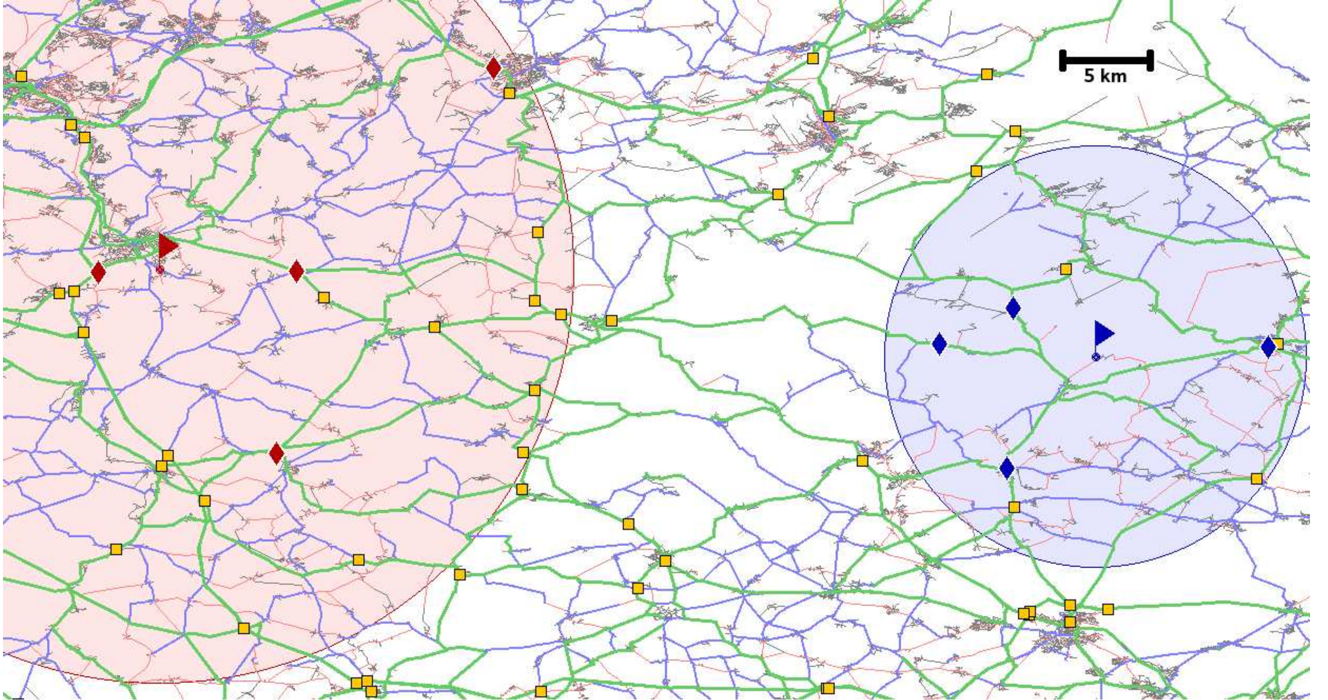

Figure 1: Finding the optimal travel time between two points somewhere between Saarbrücken and Karlsruhe amounts to retrieving the 2×4 *access nodes* (diamonds), performing 16 table lookups between all pairs of access nodes, and checking that the two disks defining the *locality filter* do not overlap.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

possible query and preprocessing times and also provide a detailed experimental analysis.[1] Section 6 summarises the results and outlines avenues for further research including additional possible approaches to transit node routing.

2 Related Work

2.1 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.

2.2 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 [20]. Using $O(n \log^3 n)$ space and preprocessing time, query time $O(\sqrt{n} \log n)$ can be achieved [5] for directed planar graphs without negative cycles. A previous practical approach based on separators is the *separator based multi-level method* [19]. 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 for each component. Müller et al. [14] 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

[1] Transit node routing was first proposed by the first three authors in [2], with a simple geometric implementation. Shortly afterwards, the last two authors combined the transit node idea with highway hierarchies [18]. For these historical reasons, the experimental parts of Sections 4 and 5 are not unified and kept separately.

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[2], preprocessing time, and query time. In our grid based scheme, the *candidates* for transit nodes form a (comparably bad) separator but only a small subset of this separator needs to be selected. The main reason for the difference in the 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 schemes.

2.3 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 [16, 17] 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 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.

2.4 Reach Based Routing. Comparable effects can be achieved with the closely related technique of *reach based routing* [9, 6].

2.5 Distance Tables. In [17] 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. This leads to much larger sets of access nodes (≈ 55) that made precomputing them appear much less attractive than 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.

2.6 Goal Direction. Another interesting property of road networks is that they allow effective goal directed search using $\mathbf{A}^*$ *search* [10]: 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 [7, 8] using the **T**riangle inequality (*ALT*). In combination with reach based routing, this is one of the fastest known speedup techniques [6]. An interesting observation is that in transit node routing, the access nodes could be used as landmarks (with the help of the distance tables). The resulting lower bound could be used for distinguishing local and global queries or for guiding local search.

2.7 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 [24]. Interestingly, so far, successful geometric speedup techniques have always been beaten by related non-geometric techniques (e.g. [10] by [7, 8] or [23] by [12, 13]). 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 test for good performance.

3 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* $A : V \rightarrow 2^{\mathcal{T}}$ that maps a vertex to its access node

[2] The current implementation uses so much space that some data does not even fit on the hard disk of the machine used and thus must be recomputed every time.

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 $d(s,t) =$

$$\min\{d(s,u)+d(u,v)+d(v,t) : u\in A(s), v\in A(t)\} \quad (3.1)$$

In principle, we can pick any set of transit nodes, any access mapping, and any locality filter fulfilling Equation (3.1) 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 (3.1)
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 $A_2 : V \to 2^{\mathcal{T}_2}$, and a locality filter L_2 such that $\neg L_2(s,t)$ implies that the shortest path distance is defined by Equation 3.1 or by $d(s,t) =$

$$\min\{d(s,u) + d(u,v) + d(v,t) : u \in A_2(s), v \in A_2(t)\} \quad (3.2)$$

In order to be able to evaluate Equation 3.2 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 3.1. 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 4 and Section 5 will refine and in some cases replace these general techniques.

3.1 Computing Access Nodes: Backward Approach. From each transit node $v \in \mathcal{T}$, start a backward Dijkstra search, i.e., a search in the reverse graph. 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 $G[\mathcal{T}]$.

3.2 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 $\mathcal{T}$.

3.3 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 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 the 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 v' can be reached from u via another access node v on a shortest path, we can discard v'. 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.

3.4 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., Section 4). 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 5). 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 $r^i(v) := \max\left\{||v-u||_2 + r^i(u) : u \in A_{i+1}(v)\right\}$, where $||v-u||_2$ denotes the Euclidean distance between u and 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.

3.5 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.

3.6 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 $d(s, u) + d(u, t) = d(s, t)$, 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 lookup. 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.

If we use a separator based approach for storing access nodes, we can bridge the gap from s to a separator node as follows: Let M denote the border nodes of the partition R containing s. Associate a bit vector of *edge flags* of size $|A(M)|$ with each edge in R. Knowing the entrance node we are heading for, we can decide which edge to take by just inspecting the edge flags. Note that this approach somewhat resembles [12, 13] with the difference that this is information leading *out* of a local region rather than information leading *to* a *global* region. When the border of one region is reached, we can then switch to the next region on the shortest path to the access node w.

4 A Geometric (Grid-Based) Implementation

In this section we will present a natural and straightforward implementation of the *transit* idea. The resulting data structure uses very little space compared to the graph itself, and achieves query times around $10\mu s$ for about 99% of the queries.

4.1 Transit Nodes.

Consider the smallest enclosing *square* of the set of nodes V (coming with x and y coordinate each), and the natural subdivision of this square into a *grid* of $g \times g$ equal-sized square cells, for some integer g. We define a set of transit nodes for each cell C as follows. Let S_{inner} and S_{outer} be the squares consisting of 5×5 cells and 9×9, respectively, each with C at their centre. Let E_C be the set of edges which have one endpoint inside C, and one outside, and define the set V_C of what we call *crossing nodes* by picking for each edge from E_C the node with smaller ID (we want to avoid considering both endpoints of an edge). Define V_{outer} and V_{inner} accordingly, see Figure 2, left, for an illustration. The set of *access nodes* for the cell C is now the set of nodes v from V_{inner} with the property that there exists a shortest path from some node in V_C to some node in V_{outer} which passes through v. The overall set of transit nodes is just the union of these sets over all cells. It is easy to see that if two nodes are at least four grid cells apart in either horizontal or vertical direction, then the shortest path between the two nodes must pass through one of these transit nodes. Also note that if a node is a transit node for some cell, it is likely to be a transit node for many other cells (all two cells away) too.

A naive way to compute these sets of transit nodes would be as follows. For each cell, compute all shortest paths between nodes in V_C and V_{outer}, and mark all nodes in V_{inner} that appear on at least one these shortest paths (again, Figure 2 will help to understand this). But such a computation would take several days even for a (for our purposes relatively coarse) 128×128 grid.

As a first improvement, consider the following simple *sweep-line algorithm*, which runs Dijkstra computations within a radius of only *three* grid cells (instead of five, as in the naive approach). Consider one vertical line of the grid after the other, and for each such line do the following. Let v be the endpoint with smaller ID of an edge intersecting the line. We run a local Dijkstra computation for each such v as follows: let C_{left} be the set of cells two grid units left of v and which have vertical distance of at most 2 grid units to the cell containing v. Accordingly define C_{right}. See Figure 2, right; there we have $C_{\text{left}} = \{CA, CB, CC, CD, CE\}$ and $C_{\text{right}} = \{C1, C2, C3, C4, C5\}$. We start the local Dijkstra at v until all nodes on the boundary of the cells in C_{left} and C_{right} respectively are settled; we remember for all settled nodes the distance to v. This Dijkstra run settles nodes at a distance of roughly 3 grid cells. After having performed such a Dijkstra computation for all nodes v on the sweep line, we consider all pairs of boundary nodes (v_L, v_R), where v_L is on the boundary of a cell on the left and v_R is on the boundary of a cell on

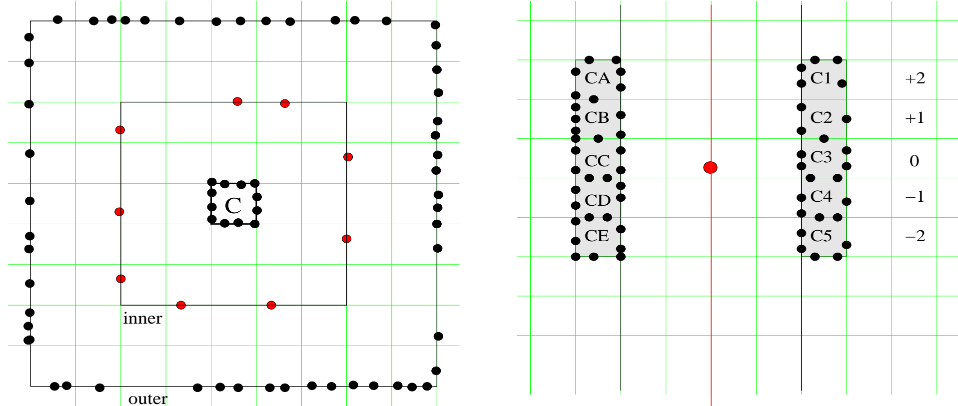

Figure 2: Definition and computation of transit nodes in the grid-based construction.

the right and the vertical distance between those cells is at most 4. We iterate over all potential transit nodes v on the sweep line and determine the set of transit nodes for which $d(v_L, v) + d(v, v_R)$ is minimal. With this set of transit nodes we associate the cells corresponding to v_L and v_R, respectively.

It is not hard to see that two such sweeps, one vertical and one horizontal, will compute the set of transit nodes defined in the previous subsection. The computation is space-efficient, because at any point in the sweep, we only need to keep track of distances within a small strip of the network. The consideration of all pairs (v_L, v_R) is negligible in terms of running time. As a further improvement, we first do the above computation for some *refinement* of the grid for which we actually want to compute transit nodes. For the finer grid, we consider only those sweep lines, which also lie on the coarser grid. When computing the transit nodes for the coarser grid, we can then restrict ourselves to nodes from the sets of transit nodes computed for the finer grid. This easily generalises to a sequence of refinements. In our experiments we use grids with $2^i \times 2^i$ cells.

4.2 Access Nodes and Distance Tables. The access nodes for a node v are just the transit nodes of the cell containing v. The distances from v to these transit nodes can be easily memorised from the Dijkstra computations which had these transit nodes as source. A standard all-pairs shortest-path computation gives us the distances between each pair of transit nodes. Note that we do not need to consider the whole original graph for this computation but can operate on a small graph only consisting of the transit nodes and (weighted) edges as memorised from the above Dijkstra computations. Since the number of transit nodes is typically small (e.g. less than 8 000 for the US road network, using a 128×128 grid), this takes negligible time.

4.3 Queries. The query algorithm is extremely simple for the grid-based approach. Given a pair of source and target node, we determine whether the two nodes are more than four grid cells in apart in either horizontal or vertical direction. If so, then by construction the shortest path must pass through at least one transit node, and we can do table lookup as described in Section 3. Otherwise, we resort to another shortest-path algorithm as described in the following.

4.4 Dealing with the Local Queries. If source and target are very close to each other (less than four grid cells apart in both horizontal and vertical direction), we cannot compute the shortest path via the transit nodes. The good news is that most shortest-path algorithms are much faster, when source and target are close to each other. In particular, Dijkstra's algorithm ist about a thousand times faster for local queries, where source and target are at most four grid cells apart, for an 256×256 grid laid over the US road network, than for arbitrary random queries (most of which are long-distance). However, the non-local queries are roughly a million times faster and the fraction of local queries is about 1 %, so the average running time over all queries would be spoiled by the local Dijkstra queries.

Instead, we can use any of the recent sophisticated algorithms to process the local queries. Highway Hierarchies, for example, achieve running times of a fraction of a millisecond for local queries, which would then only slightly affect the average processing time over all queries. The main drawback here is the additional space requirement.

As we were aiming for a very space efficient solution

we used a simple extension of Dijkstra's algorithm using geometric edge levels (as in [9]) and shortcuts. This extension uses only six additional bytes per node. An edge has level l if it is on the middle of a shortest path, where the sum of the euclidean lengths of the edges along that path are above a certain monotonic function $f(l)$. For each node u, we insert at most two shortcuts as follows: consider the first level, if any, where u lies on a chain of degree-2 nodes (degree with respect to edges of that level); on that level insert a shortcut from u to the two endpoints of this chain. In each step of the Dijkstra computation for a local query, then consider only edges above a particular level (depending on the current euclidean distance from source and target), and make use of any available shortcuts suitable for that level. This algorithm requires an additional 5 bytes per node. Note that 'uncompressing' edges in a compressed shortest path is completely straightforward with this scheme and does not require any additional memory.

4.5 Saving Space via a Multi-Level Grid. In our implementation as described so far, there is an obvious tradeoff between the size of the grid and the percentage of local queries which cannot be processed via transit node routing. For a very coarse grid, say 64×64, the number of transit nodes, and hence the table storing the distances between all pairs of transit nodes, would be very small, but the percentage of local queries would be as large as 10 %. For a very fine grid, say 1024×1024, the percentage of local queries is only 0.1 %, but now the number of transit nodes is so large, that we can no longer store, let alone compute, the distances between all pairs of transit nodes. Table 1 gives the exact tradeoffs, also with regard to preprocessing time. Note that the average query processing time for the non-local queries is around 10 microseconds, independent of the grid size.

To achieve a small fraction of local queries and a small number of transit nodes at the same time, we employ a *hierarchy of grids*. We briefly describe the two-level grid, which we used for our implementation. The generalisation to an arbitrary number of levels would be straightforward.

The first level is a 128×128 grid, which we precompute just as described so far. The second level is a 256×256 grid. For this finer grid, we compute the set of all transit nodes as described, but we compute and store distances only between those pairs of these transit nodes, which are local with respect to the 128×128 grid. This is a fraction of about 1/200th of all the distances, and can be computed and stored in negligible time and space. Note that in this simple approach, the space requirement for the individual levels simply add up.

Query processing with such a hierarchy of grids is straightforward. In a first step, determine the coarsest grid with respect to which source and target are at least four grid cells apart in either horizontal or vertical direction. Then compute the shortest path using the transit nodes and distances computed for that grid as described before. If source and target are at most four grid cells apart with respect to even the finest grid, we have to resort to the special algorithm for local queries.

4.6 Reducing the Space Further. As described so far, for each level in our grid hierarchy, we have to store the distances from each node in the graph to each of its closest transit nodes. For the US road network, the average number of closest transit nodes per node is about 11, independent of the grid size, and most distances can be stored in two bytes. For a two-level grid, this gives about 44 bytes per node.

To reduce this, we implemented the following additional heuristic. We observed that it is not necessary to store the distances to the access nodes for *every* node in the network. Consider a simplification of the road network where chains of degree 2 nodes are contracted to a single edge. In the remaining graph we greedily compute a *vertex cover*, that is, we select a set of nodes such that for every edge at least one of its endpoints is a selected node. Using this strategy we determine about a third of all nodes in the network to store distances to their respective access nodes. Then, for the source/target node v of a given query we first check whether the node is contained in the vertex cover, if so we can proceed as before. If the node is not contained in the vertex cover, a simple local search along chains of degree 2 nodes yields the desired distances to the access nodes. The average number of distances stored at a node reduces from 11.4 to 3.2 for the 128×128 grid of the US, without sigificantly affecting the query times. The total space consumption of our grid data structure then decreases to 16 bytes per node.

4.7 Implementation and Experiments. We tested all our schemes on the US road network, publically available via `http://www.census/gov/geo/www/tiger`. This is a graph with 24,266,702 nodes and 58,098,086 edges, and an average degree of 2.4. Edge lengths are travel times. We implemented our algorithms in C++ and ran all our experiments on a Dual Opteron Machine with 8 GB of main memory, running Linux. Table 2 give a summary of experimental results for our actual two-level grid approach with a reach based Dijkstra implementation for the local queries.

The grid based approach achieves an average query

	$\vert\mathcal{T}\vert$	$\vert\mathcal{T}\vert \times \vert\mathcal{T}\vert$/node	avg. $\vert A\vert$	% global queries	preprocessing
64×64	2 042	0.1	11.4	91.7%	498 min
128×128	7 426	1.1	11.4	97.4%	525 min
256×256	24 899	12.8	10.6	99.2%	638 min
512×512	89 382	164.6	9.7	99.8%	859 min
$1\,024 \times 1\,024$	351 484	2 545.5	9.1	99.9%	964 min

Table 1: Number $|\mathcal{T}|$ of transit nodes, space consumption of the distance table, average number $|A|$ of access nodes per cell, percentage of non-local queries (averaged over 100 000 random queries), and preprocessing time to determine the set of transit nodes for the US road network.

non-local (99%)	local (1%)	all queries	preprocessing	space per node
12 μs	5112 μs	63 μs	20 h	**21 bytes**

Table 2: Average query time (in microseconds), preprocessing time (in hours), and space consumption (in bytes per node) for the grid based approach, for the US road network.

time of 12 microseconds for 99% of all queries. Together with our simple algorithm for the local queries, described in Section 4.4, we get an average of 63 microseconds over all queries. This overall average time could be easily improved by employing a more sophisticated algorithm, e.g. highway hierarchies from [17], for the local queries, however at the price of a larger space requirement and a considerably more complex implementation. The space consumption of our algorithm is 21 bytes per node, which comes from 16 bytes per node for the distance tables of the two grids (Sections 4.5, 4.6) plus 5 bytes per node for the edge levels and shortcuts for the local queries (Section 4.4).

If we also output the edges along the shortest path, our average query processing becomes just about 5 milliseconds (which happens to be the average processing time for the local queries, too).

Many previous works provided a figure that showed the dependency of the processing time of a query on the *Dijkstra rank* of that query, which is the number of nodes Dijkstra's algorithm would have to settle for that query. The Dijkstra rank is a fairly natural measure of the difficulty of a query. In transit node routing, query processing times are essentially constant for the non-local queries, because the number of table lookups required varies little and is completely independent from the distance between source and target. Table 3 instead gives details on which percentage of the queries with a given Dijkstra rank are local. Note that for both the 128×128 grid and the 256×256 grid, all queries with Dijkstra rank of $2^9 = 512$ or less are local, while all queries with Dijkstra rank above $2^{21} \approx 2\,000\,000$ are non-local.

5 An Approach Based on Highway Hierarchies

5.1 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 $\langle s, \ldots, u, v, \ldots, t\rangle$ 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.

grid size	$\leq 2^9$	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	$\geq 2^{21}$
128×128	100%	100%	100%	99%	99%	99%	98%	94%	85%	64%	29%	5%	0%
256×256	100%	99%	99%	99%	97%	94%	84%	65%	36%	12%	1%	0%	0%

Table 3: Estimated fraction of queries which are local with respect to a given grid, for various ranges of Dijkstra ranks. The estimate for the column labeled 2^r is the average over 1000 random queries with Dijkstra rank in the interval $[2^r, 2^{r+1})$.

5.2 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 $K =: K_1$. In our current implementation this is level 4 or 5 in the biggest graph we have. In addition, the highway hierarchy helps us to efficiently compute the required information. Our *layer 2* is *level* $K_2 := \lceil K/2 \rceil$ of the highway hierarchy. If present, *layer 3* is *level* $K_3 := \lceil K/4 \rceil$. Note that there is a difference between the *level* of the highway hierarchy and the *layer* of transit node search.

5.3 Access Nodes and Distance Tables. We use our highway hierarchy based code for many-to-many routing to compute the top level distance table [11]. 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 3.4.

We use the forward approach from Section 3.3 to compute the access point sets. (In our case, we do not perform Dijkstra searches, but highway searches [17].)

Figure 3 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 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 3.4). 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.

5.4 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 5.3) 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 $\texttt{true}$, 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 node set for layer 3. Additional lookups in the layer-3 table ensure the correctness of this variant.

5.5 Outputting Shortest Paths. 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 node u to layer i, the path from the backward access node v to t, and the path from u to v. In case of a local query, we can fall back on the routines used in the highway hierarchies approach [3].

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

PSfrag replacements

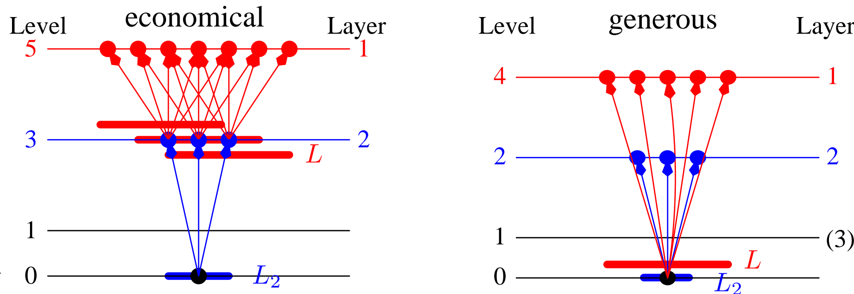

Figure 3: Representations of information relevant to highway hierarchy transit node routing.

highway search and invoke the methods from [3]. 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. 7).

When a node s and one of its access nodes 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 node of s'. In a few cases—when u is not an access node of s'—, 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. Using this subroutine, we can determine the path from s to the forward access point u and from the backward access node v to t.

A similar procedure can be used to find the path from u to v (cp. [3]). 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 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.

5.6 Experiments. We deal with two road networks. The network of Western Europe[3] 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 [22]. 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[4] have been taken from

[3] 14 countries: Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the UK

[4] 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

the DIMACS Challenge website [1]. Table 4 summarises the properties of the used 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

Table 4: Properties of the used road networks.

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.

At first, we report only the times needed to compute the shortest path distance between two nodes without outputting the actual route, while at the end of this section, 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 [17, 3], 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 [3]) is set to 10. The settings of the other parameters (some of them have been introduced in [16, 17]) can be found in Tab. 5. 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 metric (dist), we use linearly increasing sequences for c and H.

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, ...
contraction rate c	1.5	1.5	1.5, 1.6, ...

Table 5: Parameters.

Table 6 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 nodes 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 7 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 [17]. 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. 7 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 [3,

since we expect hardly any change in our measurement results.

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

Table 6: Statistics on preprocessing for the highway hierarchy approach. For layers 1 and 2, 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 [%] correct	layer 1 [%] stopped	layer 2 [%] correct	layer 2 [%] stopped	layer 3 [%] correct	layer 3 [%] stopped	query time
		eco	99.86	98.87	99.9936	99.7220	–	–	11.5 μs
USA	time	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
		eco	99.46	97.13	99.9908	99.4157	–	–	13.4 μs
EUR	time	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

Table 7: 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., $\neg L_i(s,t)$).

Tab. 5, with distance table optimisation].

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 [16] to get more detailed information about the query time distributions for queries ranging from very local to global. Figure 4 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 [15]: 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.

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.

Table 8 deals with the traversal of a complete description of the shortest path based on the implementation described in Section 5.5. 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[5] 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 av-

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

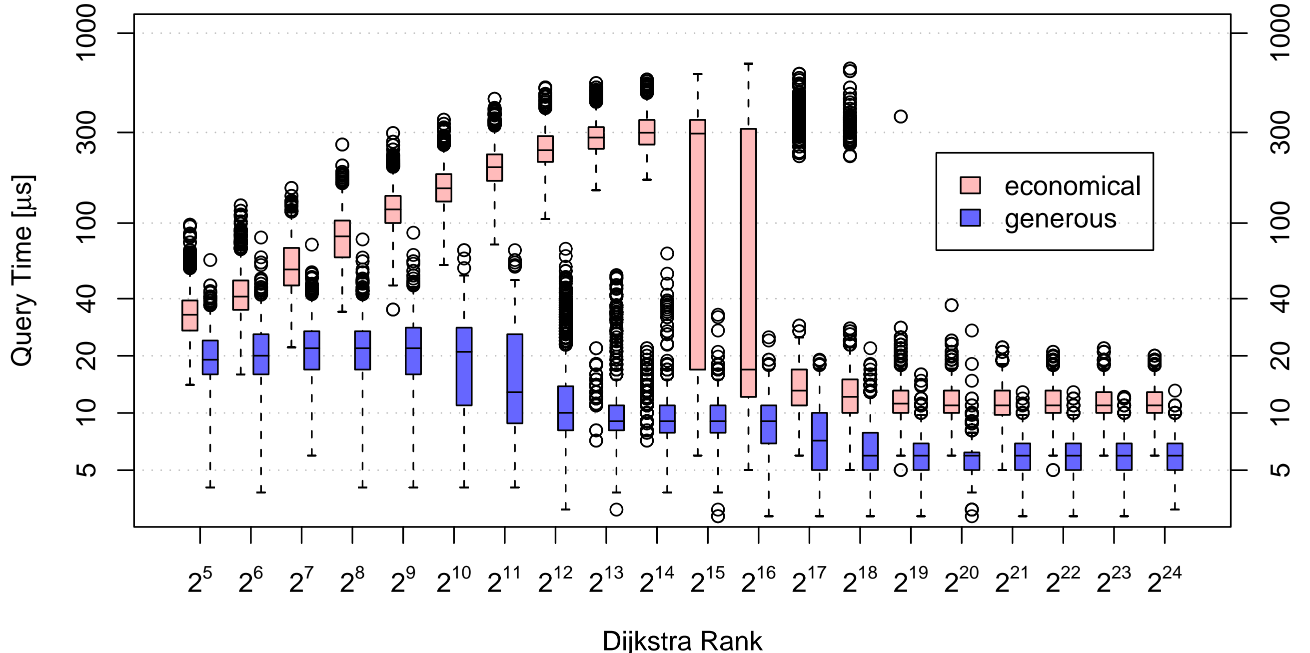

Figure 4: Query times for the USA with the travel time metric as a function of Dijkstra rank.

erage time to determine a shortest path is the time given in Tab. 8 plus the query time given in Tab. 7.

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

Table 8: 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.

6 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 but with an extremely low query time. The geometric grid approach on the other hand allows for very low space consumption at the cost of slightly higher preprocessing and query times. 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 [12, 13, 23] 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 [9, 6] 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 to then 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 Timo Bingmann for work on visualisation tools and an anonymous referee for numerous constructive comments and suggestions.

References

[1] 9th DIMACS Implementation Challenge. Shortest Paths. `http://www.dis.uniroma1.it/~challenge9/`, 2006.

[2] H. Bast, S. Funke, and D. Matijevic. TRANSIT—ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge [1]*, 2006.

[3] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway hierarchies star. In *9th DIMACS Implementation Challenge [1]*, 2006.

[4] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[5] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 232–241, 2001.

[6] A. Goldberg, H. Kaplan, and R. Werneck. Reach for A^*: Efficient point-to-point shortest path algorithms. In *Workshop on Algorithm Engineering & Experiments*, pages 129–143, Miami, 2006.

[7] A. V. Goldberg and C. Harrelson. Computing the shortest path: A^* meets graph theory. In *16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.

[8] A. V. Goldberg and R. F. Werneck. An efficient external memory shortest path algorithm. In *Workshop on Algorithm Engineering and Experimentation*, pages 26–40, 2005.

[9] R. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *6th Workshop on Algorithm Engineering and Experiments*, pages 100–111, 2004.

[10] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics*, 4(2):100–107, 1968.

[11] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner. Computing many-to-many shortest paths using highway hierarchies. In *Workshop on Algorithm Engineering and Experiments*, 2007.

[12] U. Lauther. 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, pages 219–230. IfGI prints, Institut für Geoinformatik, Münster, 2004.

[13] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up Dijkstra's algorithm. In *4th International Workshop on Efficient and Experimental Algorithms*, pages 189–202, 2005.

[14] K. Müller. 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.

[15] R Development Core Team. R: A Language and Environment for Statistical Computing. `http://www.r-project.org`, 2004.

[16] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *13th European Symposium on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 568–579. Springer, 2005.

[17] P. Sanders and D. Schultes. Engineering highway hierarchies. In *14th European Symposium on Algorithms (ESA)*, volume 4168 of *LNCS*, pages 804–816. Springer, 2006.

[18] P. Sanders and D. Schultes. Robust, almost constant time shortest-path queries in road networks. In *9th DIMACS Implementation Challenge [1]*, 2006.

[19] F. Schulz, D. Wagner, and C. D. Zaroliagis. Using multi-level graphs for timetable information. In *4th Workshop on Algorithm Engineering and Experiments*, volume 2409 of *LNCS*, pages 43–59. Springer, 2002.

[20] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 242–251, 2001.

[21] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 51(1):1–24, January 2005.

[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] D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *11th European Symposium on Algorithms*, volume 2832 of *LNCS*, pages 776–787. Springer, 2003.

[24] D. Wagner and T. Willhalm. Drawing graphs to speed up shortest-path computations. In *7th Workshop on Algorithm Engineering and Experiments*, 2005.