TRANSIT
Ultrafast Shortest-Path Queries with Linear-Time Preprocessing

Holger Bast Stefan Funke Domagoj Matijevic

Max-Planck-Institut für Informatik, Saarbrücken, Germany

{bast,funke,dmatijev} at mpi-inf dot mpg dot de

Abstract

We introduce the concept of *transit nodes*, as a means for preprocessing a road network, with given coordinates for each node and a travel time for each edge, such that point-to-point shortest-path queries can be answered extremely fast.

The transit nodes are a set of nodes as small as possible with the property that every shortest path that is *non-local* in the sense that it covers a certain not too small euclidean distance passes through at least on of these nodes. With such a set and precomputed distances from each node in the graph to its few, closest transit nodes, every non-local shortest path query becomes a simple matter of combining information from a few table lookups.

For the US road network, which has about 24 million nodes and 58 million edges, we achieve a worst-case query processing time of about 10 microseconds (not milliseconds) for 99% of all queries. This improves over the best previously reported times by two orders of magnitude.

1 Introduction

The classical way to compute the shortest path between two given nodes in a graph with given edge lengths is Dijkstra's algorithm [3]. The asymptotic running time of Dijkstra's algorithm 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 in this paper, this is $O(n \log n)$. There is strong theoretical evidence that, without preprocessing and without any assumptions on the graph except that the edge lengths be non-negative, Dijkstra's algorithm is essentially optimal [13]. In the following we will take an understanding of the basic workings of Dijkstra's algorithm as well as the associated standard terminology (relaxing an edge, settling a node) for granted. For details, see for example [2].

Our benchmark throughout this paper will be the US road network, which has about 24 million nodes and 58 million edges. Edge lengths are travel times, so that shortest paths are actually paths with minimum travel time. We will continue to speak of *shortest*, however, because that is more familiar and to stress the wider applicability of our transit node idea. On the US road network, a good implementation of Dijkstra's algorithm on a single state-of-the-art PC takes a few seconds, on average, for a random query. Note that for a random query, source and target are likely to be far away from each other, in which case Dijkstra's algorithm will settle a large portion of all nodes in the network before eventually reaching the target.

The focus of this paper is on how to preprocess a road network in order to speed up subsequent shortest-path queries. This follows a recent surge of interest in exactly that problem. Note that

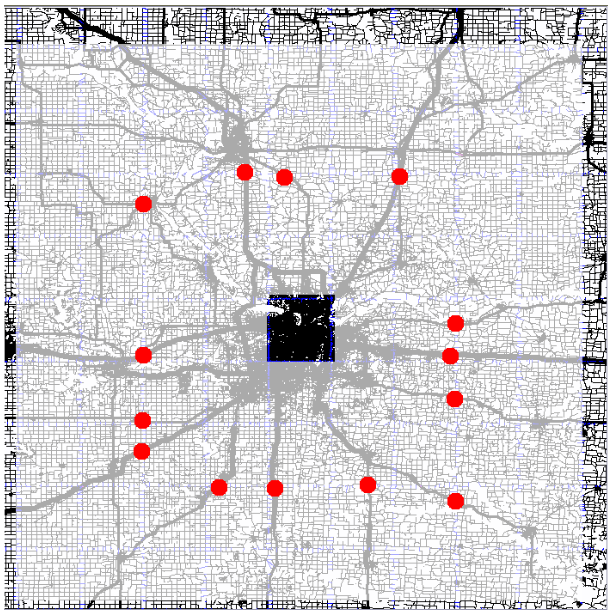

Figure 1: Transit nodes (red/bold dots) for a part of a city (center, dark) when travelling far (outside the light-gray area).

road networks are particular in at least two ways, as compared to arbitrary graphs: (i) the nodes have small degree, 2.4 on average for the US road network, with little variation; and (ii) 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 technique, like those in related previous works, makes extensive use of these two properties.

2 Our results

We present a new algorithm, coined TRANSIT, which can answer non-local shortest path queries extremely fast, by combining information from a small number of lookups in a table. We implemented TRANSIT and tested it on the US road network (US), which has about 24 millions nodes and 58 million edges. We achieve an average query processing time of around 10 microseconds (not milliseconds) for 99 % of all queries, when only the length (travel time) of the shortest path is required. The remaining 1 % of the queries are local in the sense that source and target are geometrically very close to each other. We also provide a simple algorithm for dealing with those few local queries efficiently. However, we prefer to view our transit node approach as a *filter*: the vast majority of all queries can be processed extremely fast, leaving a small fraction of local queries, which can be processed by any other method. Note that already Dijktra's algorithm would process local queries by orders of magnitudes faster than arbitrary random queries.

Our processing times for the non-local queries beat the best previously reported figure of about 1 millisecond, due to Sanders and Schultes [11], *by two orders of magnitude*. When the full path, with all its edges, is to be output, we achieve an average query processing time of about 5 milliseconds on the US road network. This latter result stands without comparison, because all of the more recent, sophisticated algorithms use some form of edge compression, which does not easily allow them to output the edges along the shortest path.

The basic idea of TRANSIT is as follows. For a given road network, compute a small set of *transit nodes* with the property that every shortest path between that covers a certain not too small euclidean distance passes through at least one of these transit nodes. For every node in the given graph, then compute a set of *closest transit nodes*, with the property that every shortest path starting from that node and passing through a transit node at all (which it will if it goes sufficiently far), will pass through one of these closest transit nodes. These sets of closest transit nodes turn out to be very small: about 10 on average for our choice of transit nodes on the US road network. This allows us to precompute, for each node, the distances to each of its closest transit nodes. Also, the overall number of transit nodes turns out to be small enough so that we can easily precompute and store the distances between all pairs fo transit nodes.

A non-local shortest path query can then easily be answered as follows. For a given source node *src* and target node *trg*, fetch the precomputed sets of closest transit nodes T_{src} and T_{trg}, respectively. For each pair of nodes $t_{src} \in T_{src}$ and $t_{trg} \in T_{trg}$ compute the length of the shortest path passing through these nodes, which is $d(src, t_{src}) + d(t_{src}, t_{trg}) + d(t_{trg}, t)$. Note that all three distances in this sum have been precomputed. The minimum of these $|T_{src}| \cdot |T_{trg}|$ lengths is the length of the shortest path.

Given an algorithm for length-only shortest path queries, one can easily compute the edges along the shortest path using a few length-only shortest path queries per edge on the shortest path. To see this, assume we have already found a portion of the shortest path from the source to a node u. To find the next edge on the path, we simply launch a length-only shortest path query for each of the adjancent nodes of u. Given the length of the portion of the shortest path we already know, its total length, and the length of the edges adjacent to u, it is then easy to tell which of these edges is next on the shortest path.

We want to stress that there are natural applications, where length-only shortest path queries are good enough, and not all the edges along the path are required. Most car navigation systems, for example, have at most a local view of the road network (if any). In that case it suffices to know the next few edges on the shortest path, and these can be computed by just a few length-only shortest-path queries, as described above.

We decribe TRANSIT in more detail in Section 4.

3 Related Work

We give a quick survey of work directly relevant to the problem of preprocessing road networks for subsequent fast shortest-path querying.

Gutman in [6] proposes a general concept of *edge levels.* Consider an edge e that appears "in the middle" of a shortest paths, shortest with respect to travel time, between two nodes that are a certain distance d apart, distance with respect to some arbitrary other metric, e.g. euclidean distance. Then the level of e is the higher, the larger d is. Gutman defines levels with respect to euclidean distance, but he notes that *any* metric can be used for the discrimination of the "in the middle" property. He presents simple algorithms which compute upper bounds for the edge levels and instruments those to obtain more efficient exact shortest path queries on moderate-size road networks. Due to the use of the euclidean metric as classifying metric, his approach allows for several variants of Dijkstra, in particular a natural goal-directed (unidirectional) version as well as efficient one-to-many shortest path queries. The – compared to later work – less competitive running times, both for the preprocessing phase as well as the queries are mostly due to the lack of an efficient compression scheme, which – in particular for the networks induced by higher level edges – improves the processing time considerably.

Later Sanders and Schultes have adopted a different classifying metric for their so-called *Highway-Hierarchies*. In an ordinary Dijkstra computation from a source *src*, say that the rth node settled has Dijkstra rank r with respect to *src*. Sanders and Schultes say that the level of an edge (u, v) is high if it is on a shortest path between some *src* and *trg* such that v has high Dijkstra rank with respect to *src* and u has high Dijkstra rank with respect to *trg*. They achieve an improvement of an order of magnitude both in preprocessing time as well as query times, mainly because of a highly efficient compression and pruning scheme in the higher levels of the network. The output of the algorithm is a path containing compressed edges, though, and it is not clear how much effort in both time and space is necessary to uncompress those edges. Their variant is also inherently bidirectional, so both goal-direction as well as one-to-many queries are not easily added.

Goldberg et al. in [5] combine edge levels with a compression scheme and they use lower bounds, based on precomputed distances to a few landmarks, to allow for a more goal-directed search. They report running times comparable to those of [10]. Their space consumption is somewhat higher though, because every node in the network has to store distances to all landmarks.

Most recently, Sanders and Schultes [11] have presented the so far best combination of preprocessing and query time. They show how to preprocess the US road network in 15 minutes, for subsequent query times of, on the average, 1 millisecond. As mentioned before, however, their scheme outputs only the shortest path distance, and it is not all clear how to get the actual shortest path edges efficiently. While we could not yet come close to their extremely fast preprocessing time, our cost-only scheme beats their query time by two orders of magnitude.

Möhring et al. [9, 7], and, in independent work, Lauther [8] explored *edge signs* as a means to achieve very fast query times. Intuitively, an edge sign says whether that edge is on a shortest path to a particular region of the graph. In an extreme case, an edge could have a sign to every node on the shortest path to which it lies. A shortest path query could then be answered by simply following the signs to the target without any detour. However, to precompute these perfect signs requires an all-pairs shortest-path computation, which takes quadratic time and would be infeasible already for a small portion of the whole road network, say the network of California. It is shown in [9, 7] and [8] how to cut down on this preprocessing somewhat, by putting up signs to sufficiently large regions of the graph, but precomputation of those signs is still prohibitive for large networks.

Most recently, Sanders and Schultes have combined our transit node idea with their highway hierarchies [12]. This combination achieves extremely fast processing and fast preprocessing at the same time, at the price of a somewhat more complex algorithm and a higher space consumption. A joint paper presenting and comparing both approaches, our simple geometric one and the one based on highway hierarchies, is to appear at ALENEX'07 [1]. In our conclusions in Section 6, we give some details on why the tranist node idea goes together with highway hierarchies particulary well.

4 The TRANSIT algorithm

4.1 Intuition

The basic intuition behind our approach is very simple: imagine you live in a big city and intend to travel long-distance by car. What you will observe is that irrespectively of where your final destination is (as long it is reasonably far away) and where exactly you live in the city, there will

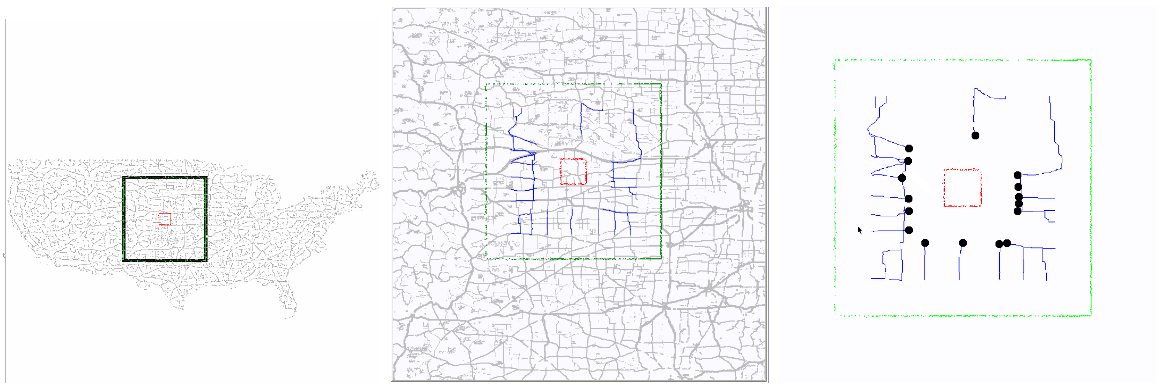
Figure 2: Transit neighborhood of a cell in a 64 × 64 subdivision of the US.

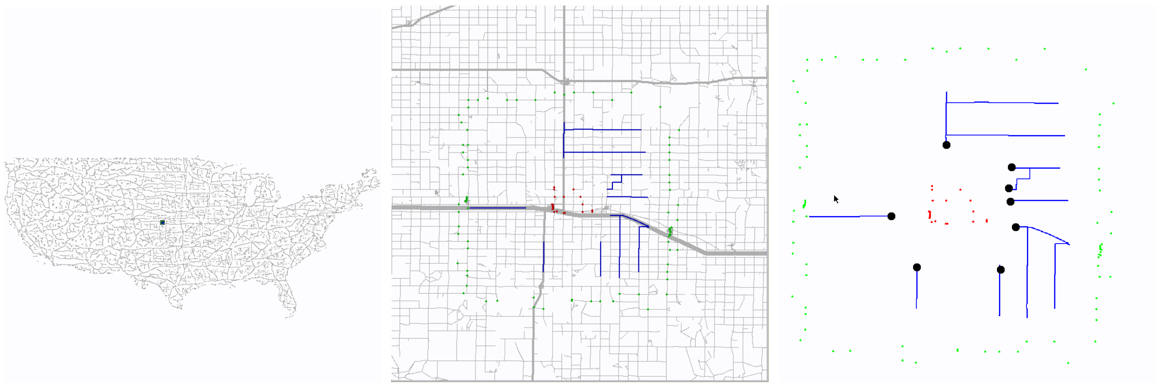
Figure 3: Transit neighborhood of a cell in a 1024 × 1024 subdivision of the US.

be few roads via which you will actually leave the urban area when travelling on a shortest path to your destination. In Figure 1 we have depicted these roads for the center part of a city. No matter where you start your journey inside the central region (in dark) – if your final destination lies outside the light-grey area and you travel on a shortest path, you will pass through one of the 14 marked roads (red/bold dots). This property that long-distance trips (where the length is to be seen relative to the "starting region") pass through few *transit nodes* is in fact to some degree invariable to scale. The example in Figure 1 shows the transit nodes for a cell in a 256 × 256 subdivision of the road network of the US; there are 14 of them. Figures 2 and 3 show transit nodes (or more precisely transit neighborhoods by which we compute transit nodes) for cells of a 64 × 64 and 1024 × 1024 subdivision of the US respectively. They exhibit 17 and 8 transit nodes respectively.

In essence our approach is then to compute an (in our case geometric) subdivision of the network into cells and determine their transit nodes, such that the total number of transit nodes is $o(\sqrt{n})$. This allows us to precompute all pairwise distances between transit nodes and store them in sublinear space. Furthermore each node stores distances to the transit nodes of its resident cell. At query time a simple lookup yields the exact distance between any source-target pair provided they are not too close to each other.

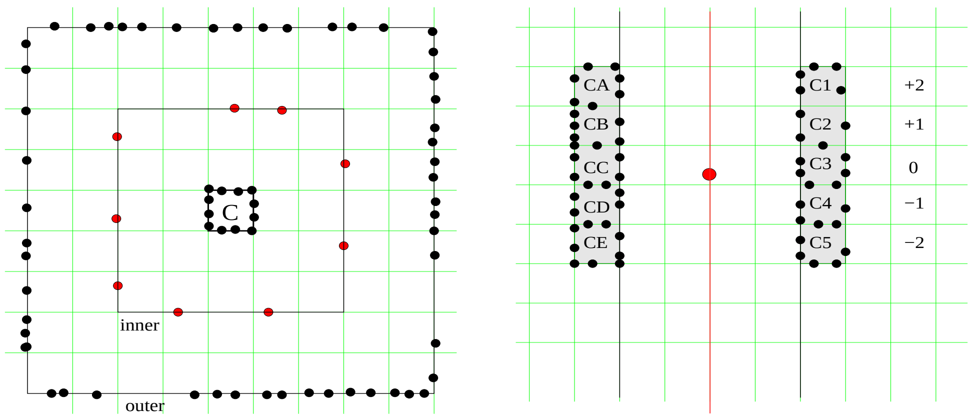

Figure 4: Definition and computation of transit nodes in the grid-based construction.

4.2 Computing the Set of Transit Nodes

Consider the smallest enclosing *square* of the set of nodes (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 center. 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. Define V_{outer} and V_{inner} accordingly. See the left side of Figure 4 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, each of them 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. Figure 4 will again help to understand this. But such a computation would take several days even for a relatively coarse grid like the 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 one of the endpoints 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. Define C_{right} accordingly. See Figure 4, right; there we have $C_{\text{left}} = \{CA, CB, CC, CD\}$ and $C_{\text{right}} = \{C1, C2, C3, C4\}$. 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 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 exactly the set of transit nodes defined above (the union of all sets of access nodes). 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 generalizes to a sequence of refinements. In our experiments we used grids with $2^i \times 2^i$ cells.

4.3 Computing the Distance Tables

For each node v, the distances to the access transit nodes of its cell can be easily memorized from the Disjktra 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. Since the number of transit nodes is small (less than 8 000 for the US road network, using a 128×128 grid), this takes negligible time. The space consumption of these distance tables is discussed in Sections 4.7 and 4.8 below.

4.4 Shortest-path queries (length only)

We next describe how to compute the *length* of the shortest path between a given source node *src* and a given target node *trg*, based on the preprocessing described in the previous two subsections. We here give a description for the scenario where we have precomputed only a single level of transit nodes. The extension to a hierachy of grids is straightforward, and will be explained in Section 4.7.

0. If *src* and *trg* are less than four grid cells (with respect to the grid used in the precomputation) apart, compute the distance from *src* to *trg* via an algorithm suitable for local shortest-path queries; a number of possibilities are described in Section 4.6. Otherwise, perform the following steps:

1. Fetch the lists T_{src} and T_{trg} of the closest transit nodes for the grid cells containing *src* and *trg*, respectively. Also fetch the lists of precomputed distances $d(src, t_{src}), t_{src} \in T_{src}$ and $d(s, t_{trg}), t_{trg} \in T_{trg}$.

2. For each pair of $t_{src} \in T_{src}$ and $t_{trg} \in T_{trg}$ compute the sum of the lengths of the shortest path from *src* to t_{src}, from t_{src} to t_{trg}, and from t_{trg} to *trg*, which is $d(src, t_{src}) + d(t_{src}, t_{trg}) + d(t_{trg}, trg)$. Note that we may have $t_{src} = t_{trg}$, in which case $d(t_{src}, t_{trg}) = 0$.

3. Compute the length of the shortest path from *src* to *trg* as the minimum of the $|T_{src}| \cdot |T_{trg}|$ distances computed in step 2.

The algorithm is easily seen to be correct. Steps 1-3 will only be executed if source and target are more than four grid cells apart. Then, by the definition of the transit nodes in Section 4.2, the shortest path between source and target must pass through at least one transit node. But then, by the definition of closest transit nodes, the shortest path from *src* to *trg* will pass through one of the closest transit node of *src* as well as through one of the closest transit nodes of *trg*. The shortest path will therefore be among those tried in step 2, and none of the other paths tried can be shorter.

Since we have precomputed the distances from each node to its closest transit nodes and the distances between each pair of transit nodes, steps 1-3 take time $O(|T_{src}| \cdot |T_{trg}|)$. The average number of closest transit nodes of a node is a small constant — about 10 for the US road network.

4.5 Shortest-path queries (with edges)

In this subsection, we describe how we can enhance the procedure given in the previous subsection to also output the edges along the shortest path from a given source node *src* to a given target node *trg*.

Assume that we have executed the procedure from the previous subsection, that is, we already know the length of the shortest path from *src* to *trg*. Assume that we have already found the part of the shortest path from *src* to some u (initially, $u = src$). Let $d(u, trg)$, which we can compute as $d(src, trg) - d(src, u)$, be the length of the part of the path which we have not found yet. Then the next node on the shortest path is that node v adjacent to u with the property that $d(u, trg) = c(u, v) + d(v, trg)$. This node can therefore be easily identified from the nodes adjacent to u, if only we can compute the distances $d(v, trg)$. But these are just instances of the problem we solved in the previous subsection: given two nodes, compute the length of the shortest path between them.

As described so far, the computation of $d(v, trg)$ would resort to the special algorithm for local shortest-path queries when v and *trg* are less than four grid cells apart. We can avoid this, if we compute the shortest path from *src* only until four grid cells away from *trg*, and, symmetrically, compute the shortest path from *trg* until four grid cells away from *src*. This will give us the full path if *src* and *trg* are at least eight grid cells apart. Otherwise, there is indeed no way around runnning the local algorithm.

4.6 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 for length-only shortest-path queries; less than eight grid cells apart in that way when computing the edges along the path), we cannot compute the shortest path via the transit nodes. This makes sense intuitively: there is hardly any hierarchy of roads in an area like, for example, downtown Manhattan, and a shortest path between two locations within the same such area will mostly consist of (small) roads of the same kind. In such a situation, no small set of transit nodes exist.

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 128×128 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 drawback is that we would need the full implementation of another method, and that this method requires additional space and precomputation time.

For our experiments, we used a simple extension of Dijkstra's algorithm using geometric edge levels and shortcuts, as outlined in Section 3. 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 unique 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.

4.7 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.

	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{T}\rvert \times \lvert\mathcal{T}\rvert$/node	avg. $\lvert A\rvert$	% 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 $\lvert\mathcal{T}\rvert$ of transit nodes, space consumption of the distance table, average number $\lvert A\rvert$ 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.

To achieve a small fraction of local queries and 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 generalization to an arbitrary number of levels would be straightforward.

The first level is an 128×128 grid, which we precompute just as described so far. The second level is an 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. A more sophisticated approach to multi-level transit node routing is described in [1].

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 horizonatal or vertical direction. Then compute the shortest path using the transit nodes and distances computed for that grid, just as described in Sections 4.4 and 4.5. 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.

4.8 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.

5 Implementation and Experiments

5.1 Experimental results

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 our experimental results.

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 our new algorithm TRANSIT, for the US road network.

TRANSIT achieves an average query time of 12 microseconds for 99% of all queries. Together with our simple algorithm for the local queries, described in Section 4.6, 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. the one from [11], for the local queries, however at the price of a larger space requirement and a much 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 (Section 4.7) plus 5 bytes per node for the edge levels and shortcuts for the local queries (Section 4.6).

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). This is still competitive with the processing times reported in [11] and its closest competitors [10] [4] [5]. All of these schemes, however, *do not* output edges along the shortest path, nor can they be easily modified to do so without a severe slowdown in query time. This is because in all of these works, some kind of *compression* of subpaths is used. This compression is essential for the fast query processing times. While it is easy for these algorithms to output a sequence of somehow *compressed* edges, it is unclear how to obtain the sequence of the actual edges along the path.

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.

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 the 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 Graphical User Interface

We have gone to quite some pain to implement a relatively comfortable graphical user interface (GUI) for displaying our road networks plus a number of additional elements. The GUI is implemented in C++ using the Gtkmm library, which gives instant response times for dragging and zooming also for large road networks like that of the US. The GUI runs in its own thread, so that user and redraw events can be interleaved with computation and other code.

The GUI supports seamless dragging and zooming with the mouse (wheel), as one knows it, for example, from tools like Google Earth. This is very convenient for navigating in a large network quickly, but that was also the part that cost us the most work. The graph has to be divided into

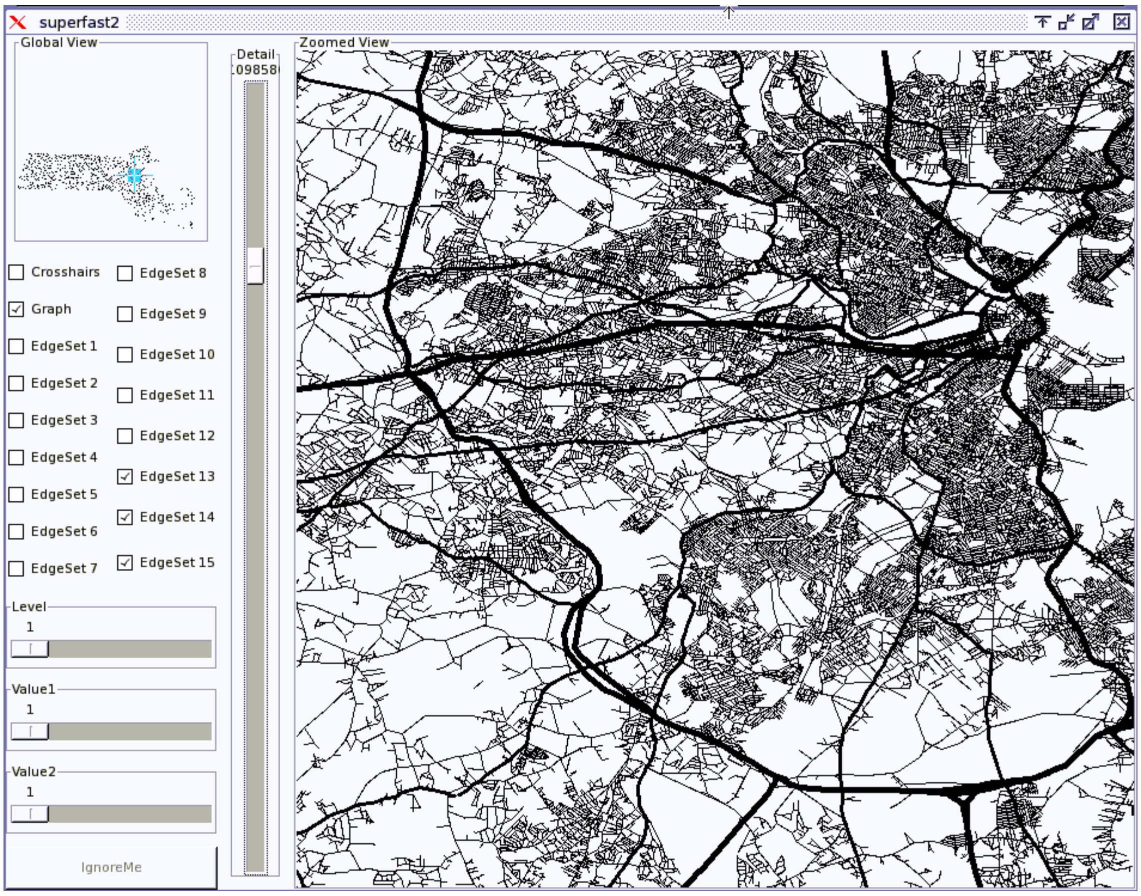

Figure 5: Screenshot of our interactive graphical user interface.

relatively small chunks, and only those chunks must be drawn which are actually visible from the current perspective and position. Also, there have to be priorities between edges, because always drawing all edges tends to clutter up the display and is an efficiency problem, too. The GUI also supports the drawing of custom objects, like cross hairs (to visualize important locations), arrows along roads (to visualize something like edge signs), etc.

6 Conclusions

Transit nodes are a simple, yet powerful idea: they reduce the shortest-path computation for all but a small fraction of local queries to a few table lookups. In this paper we have focused on presenting this idea and giving a simple geometric algorithm realizing that idea.

We have seen that transit nodes can be implemented with 21 bytes per node, on top of the space required for storing the graph itself (which in our representation requires about 32 bytes per node). This is small compared to the space requirements of the best previously reported methods.

We require about 20 hours of preprocessing time for the US road network, which is reasonable but not fast. Following our work, Sanders and Schultes have combined the transit node idea with their highway hierarchies, simultaneously achieving very fast query processing times for all queries and fast preprocessing times with a moderate space consumption. Highway hierarchies go together particulary well with the transit node idea for three main reasons: first, for the set of transit nodes one can simply take a level from the highway hierarchy; second, highway hierachies give very good processing times for the local queries, which cannot be processed via transit nodes; third, highway hierarchies can be precomputed in less than an hour for the US road network. A joint work presenting and comparing both approaches, the simple geometric one and the one based on highway hierarchies, is to appear at ALENEX'07 [1].

The algorithms in this paper work for undirected graphs. Generalization to directed graphs is not trivial but looks feasible. Sanders and Schultes managed to do this rather easily in their improvement from [10] to [11].

A more difficult open problem is how to make our data structure dynamic, that is, how to update our data structures in response to only small changes in the graph, like an increase of a few edge lengths due to a traffic jam. None of the mentioned previous works on the problem has tackled this problem so far, except for a trivial recomputation from scratch.

References

[1] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, 2007.

[2] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

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

[4] A. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *16th Symposium on Discrete Algorithms (SODA'05)*, pages 156–165, 2005.

[5] A. Goldberg, H. Kaplan, and R. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*, 2006.

[6] R. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, 2004.

[7] E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of shortest path and constrained shortest path computation. In *4th Workshop on Experimental and Efficient Algorithm (WEA'05)*, pages 126–138, 2005.

[8] U. Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Münster GI-Tage*, 2004.

[9] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up dijkstra's algorithm. In *4th Workshop on Experimental and Efficient Algorithm (WEA'05)*, pages 189–202, 2005.

[10] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *13th European Symposium on Algorithms (ESA'05)*, pages 568–579, 2005.

[11] P. Sanders and D. Schultes. Engineering highway hierarchies. In *14th European Symposium on Algorithms (ESA'06)*, pages 804–816, 2006.

[12] P. Sanders and D. Schultes. Robust, almost constant time shortest-path queries on road networks. In *DIMACS Implementation Challenge Shortest Paths*, 2006.

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