DIMACS Series in Discrete Mathematics
and Theoretical Computer Science

High-Performance Multi-Level Routing

Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz,
and Dorothea Wagner

Abstract. Shortest-path computation is a frequent task in practice. Owing to ever-growing real-world graphs, there is a constant need for faster algorithms. In the course of time, a large number of techniques to heuristically speed up Dijkstra's shortest-path algorithm have been devised. This work reviews the multi-level technique to answer shortest-path queries exactly [**24, 9**], which makes use of a hierarchical decomposition of the input graph and precomputation of supplementary information. We develop this preprocessing to the maximum and introduce several ideas to enhance this approach considerably, by reorganizing the precomputed data in partial graphs and optimizing them individually.

To answer a given query, certain partial graphs are combined to a search graph, which can be explored by a simple and fast procedure. The concept behind the construction of the search graph is such that query times depend mainly on the number of partial graphs included. This is confirmed by experiments with different road graphs, each containing several million vertices, and time, distance, and unit metrics. Our query algorithm computes the distance between any pair of vertices in no more than 40 μs, however, a lengthy preprocessing is required to achieve this query performance.

1. Introduction

Computation of shortest paths is a central requirement for many applications, such as route planning or network search. Facing real-world data, the need for speed remains unabated: collection of geographic information is enhanced constantly, resulting in increasingly comprehensive road graphs; public-transportation networks often comprise datasets from different means of transportation, such as train, tram, ferry, and even airplane schedules; and the graph representing the WWW is growing faster than ever. There are two basic approaches to tackle this task: relying on approximate algorithms, or devising faster exact ones. We opt for the latter.

2000 *Mathematics Subject Classification.* Primary 68R10; Secondary 90B20.

Key words and phrases. Shortest paths, graph, speed-up technique, decomposition, hierarchy, levels, preprocessing, supersedement, graph compression.

This work was partially supported by the Future and Emerging Technologies Unit of EC (IST priority – 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

Since its publication in 1959, Dijkstra's famous algorithm for calculation of shortest paths in a directed graph with nonnegative edge weights [**4**] has been subject to many improvements. Due to enormous space requirement (quadratic in the number of vertices), we cannot afford precomputing shortest paths between all pairs of vertices. However, graphs can be preprocessed at an off-line step so that subsequent on-line queries take only a fraction of the time used by Dijkstra's algorithm. One recent speed-up technique [**1, 22**], which also relies on a preprocessing, yields for the European road network that we also use for our experiments a considerable average query time of 4 μs when travel times as edges weights are used, but of comparatively high 38 μs for travel distances.

In this work, we present a further enhancement of the multi-level technique given in [**24**], which is based on a hierarchical decomposition of the input graph and computation of an auxiliary graph containing additional information. The use of this precomputed data allows, at the on-line stage, for reduction in search space and, consequently, query time. We develop the preprocessing to the maximum: our new variant outsources almost all of the effort needed to compute a shortest path to the preprocessing stage. It therefore fits best into an environment where query time is invaluable but long preprocessing times (and a fair amount of precomputed data) can be afforded, such as car navigation systems or web-based route planners. While in [**24, 9**] the multi-level approach was shown to be effective for graphs of up to 100 000 vertices, we are now able to handle much bigger graphs still within reasonable time.

The main differences to the former multi-level technique concern the following issues. During the preprocessing stage, instead of one single multi-level graph we compute a large number of small *partial* graphs. We show that for each possible query, there is a search graph combined of several partial graphs which preserves the distance between the dedicated vertices. This graph is acyclic, and we give a simple linear-time procedure to search it. The advantage of dealing with multiple graphs is that each of them can be optimized individually, which is achieved by two measures: first, omission of edges whose relaxation will never create a shorter path; and second, transformation of the partial graphs into equivalent graphs that preserve all shortest paths but have fewer edges. What is more, we make use of the fact that the preprocessing is parallelizable.

The trade-off between preprocessing effort and query time is adjustable. For fixed parameters, we can provide a guarantee for both the number of edges considered by the search algorithm and the query time. With our implementation, keeping the preprocessed data in secondary storage, we can answer a query through few random accesses to that storage. If the preprocessed data fits entirely into main memory, our query performance is competitive to that of other recent approaches: we obtain query times of less than 40 μs (except for very few outliers) for graphs with up to 24 million vertices, representing the Western European and US road networks. Moreover, our approach yields *equal* performance for all metrics investigated (travel times, travel distances, and unit edge lengths).

In the remainder of this section, we classify our approach in the context of other shortest-path speed-up techniques. The next section briefly reviews the multi-level technique as presented in [**24**], and shows the various refinements made. An experimental study is presented in Section 3, and we conclude in Section 4 addressing some aspects to be explored in the future.

1.1. Related Work. This paper is strongly based on [**18**], which is the master's thesis by one of the authors. It can be seen as a further development in that, e.g., preprocessing time could be improved through refinement of our code. Other pieces of information, however, such as details on some proofs or algorithmic aspects, can, in this work, not be displayed in full length, so we may refer the interested reader to [**18**].

There are a large number of other techniques to speed up single-pair shortest-path algorithms, most of which rely on Dijkstra's algorithm [**4**]. In the following survey, we focus on methods that in a preprocessing step compute some additional information, which is used at the on-line step for answering a shortest-path query. We differentiate between techniques that attach the precomputed data to the graph's vertices or edges, permitting the on-line algorithm to quickly decide which parts of the graph can be pruned [**23, 26, 27, 7, 15, 17, 14, 5**], and such that precompute a hierarchical auxiliary graph, a slender part of which suffices to answer a given shortest-path query [**24, 9, 12, 11, 20, 21**]. We want to briefly review the latter works and point out their relationship to ours.

As mentioned above, the method presented in this work uses the same basic concepts as the one described in [**24, 9**] (which will occasionally be referred to as the *classic* multi-level technique), where the following enhancements are made. The auxiliary data is distributed to many partial graphs, which can afterwards be thinned out and optimized individually. Given start and destination vertices, we combine several partial graphs to obtain an acyclic search graph, which can be explored by the on-line stage in linear time.

The *HiTi model* by Jung and Pramanik [**12**] is similar to the classic multi-level technique, except that it uses edge separators rather than vertex separators. Also with *hierarchical encoded path views*, presented by Jing, Huang, and Rundensteiner [**11**], various partial graphs are computed, which are combined appropriately to form a search graph for a given query. No graph optimization is used, but a compression technique to also keep track of the course of shortest paths is given.

Finally, the *highway hierarchies* technique, introduced by Sanders and Schultes [**20, 21**], computes a hierarchy of coarsenings of the input graph, where the search algorithm proceeds in a bidirectional fashion and needs to consider vertices of only one level of hierarchy at a time. In a further development [**2**], highway hierarchies have been extended to *transit node routing*, which also takes advantage of precomputed all-pair distances of a selection of vertices; the difference to our approach is that these distances are not represented by graphs but matrices, which do not seem to induce as simple means for optimization. The given description of highway node routing is generic enough so that it also covers the basic concept behind our approach. One further difference concerns the way of determining selected vertices; second, transit node routing is designed to yield better speed-ups with the travel time instead of distance metric, which is not true for our technique.

2. Multi-Level Graphs

The formal description of our high-performance multi-level technique (HPML) is divided into two parts. The first one reviews the classic multi-level technique [**24**] and points out the major modifications and enhancements made to it. In the second part, we present the core ingredient to achieve massive reduction in search space and query time, optimization of the partial search graph.

2.1. Enhancing Multi-Level Graphs. A multi-level graph $\mathcal{M}$ extends a weighted digraph $G = (V, E)$ through multiple *levels* of edges, depending on a sequence of vertex subsets. For a pair of vertices $s, t \in V$, a subgraph $\mathcal{M}_{st}$ of $\mathcal{M}$, called *search graph*, with the same s-t distance as in G can be determined efficiently. As the search graph is substantially smaller than G, it allows for answering the given query much faster.

The description of our model is organized as follows. We first fix some notation needed to define multi-level graphs, show how to construct these, and briefly address the issue of parallelizing the preprocessing step. To extract from the whole search graph a search graph $\mathcal{M}_{st}$ sufficent to answer a given query $(s,\ t)$, we have to define an auxiliary datastructure called *component tree*, and give a formal proof that shortest s-t paths in G and $\mathcal{M}_{st}$ are of equal length. The search graph $\mathcal{M}_{st}$ can be transformed into an acyclic graph, which property yields a simple and fast search algorithm compared to Dijkstra's algorithm. Finally, due to construction of our model, short-distance queries have to be treated differently.

2.1.1. *Notation.* To create a multi-level graph, we use a sequence of vertex subsets, denoted by $\mathscr{S} = \langle S_i \rangle$ with $1 \leq i \leq l$. Each S_i is called a *separator set.* The separator sets are decreasing with respect to set inclusion: $V \supset S_1 \supset S_2 \supset \ldots \supset S_l$. Best performance can be achieved when the graph $G - S_i$ falls apart into 'many' components of similar size, while $|S_i|$ is 'small' compared to $|V|$. For the decomposed graph $G - S_i$, we shall use the following definitions.

- By $\mathscr{C}_i$, we denote the set of maximal connected components at level i. A connected component $C \in \mathscr{C}_i$ itself is a weighted graph, whose vertices are referred to by $V(C)$.
- For a vertex $v \in V \setminus S_i$, let $C_i^v \in \mathscr{C}_i$ be the component with $v \in C_i^v$. We call C_i^v the *home component* of v at level i. To simplify notation, we define $C_i^v := \{v\}$ for $i \in \{0, \ldots, l\}$ and $v \in S_i$, and let $S_0 = V$.
- We call a vertex $v \in S_i$ *adjacent* to a component $C \in \mathscr{C}_i$ if there is an edge between v and a vertex in C in either direction. The set of all vertices adjacent to C is denoted by $\mathit{Adj}(C)$. For $v \in S_i$ (i.e., $C_i^v = \{v\}$), we define $\mathit{Adj}(C_i^v) := \{v\}$.
- A component C_i^v together with its adjacent vertices is called the *wrapped component* $G_i^v = G \cap (V(C_i^v) \cup \mathit{Adj}(C_i^v))$.

Figure 1 shows two components (darker shades) and their belonging wrapped components (lighter shades) at levels 1 (smaller components) and 2 (larger components), respectively, as an example for the above definitions. The adjacent vertex sets are $\{v_1, v_2, v_4\}$ and $\{v_3, v_4\}$. Note that vertex v_4 is adjacent to both components, as it is a separator vertex at both levels 1 and 2.

The above definition requires the components C_i^v to be connected; however, we do not rely on this property. If two components C_i^v and C_i^w share the same parent component, we can merge these two components into one new component $C_i^{vw} = C_i^v \cup C_i^w$. The adjacent vertices of the merged component are the vertices that are adjacent to at least one original component. If $\mathit{Adj}(C_i^w) \subseteq \mathit{Adj}(C_i^v)$, merging reduces the total number of components without increasing the number of adjacent vertices for any component. It is advisable to do so, as reducing the total number of components also reduces preprocessing time. For our test instance, merging components leads to a reduction of the total number of components by up to two orders of magnitude.

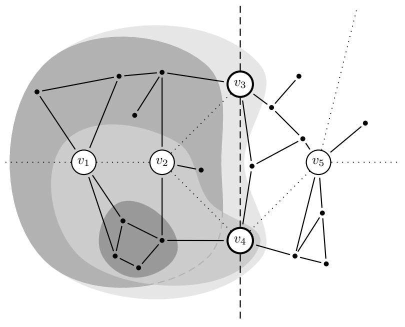


FIGURE 1. Hierarchy due to graph decomposition: components (darker shades) with belonging wrapped components (lighter shades) at levels 1 (smaller components) and 2 (larger components).

2.1.2. *Multi-Level Graph.* Each level of the multi-level graph $\mathcal{M}$ is determined by a set of edges. For each $i \in \{1, \dots, l\}$, we construct three sets of edges from the following *candidate sets*:

Level edges: $E_i \subseteq S_i \times S_i$.
Upward edges: $U_i \subseteq S_{i-1} \times S_i$.
Downward edges: $D_i \subseteq S_i \times S_{i-1}$.

For a level $i \in \{1, \dots, l\}$, a candidate edge $(v, w) \in U_i, D_i$ is elected to be an upward or downward edge only if there is a v-w path in G that does not contain any vertices in S_i besides v or w (in other words, both endpoints must be contained in the same wrapped component G_i). The weight of an upward or downward edge is set to the length of a shortest such paths. Note that this equals the v-w distance in the wrapped component G_i; there may exist shorter paths in G that leave G_i.

A level edge at level $i \in \{1, \dots, l-1\}$ exists if both of its endpoints are contained in the same wrapped component G_{i+1} at level $i+1$. For level l, we simply use all candidate edges: $E_l := S_l \times S_l$. The weight of a level edge matches the distance in G. This constitutes an essential difference to [**24**], where level edges were defined similarly to upward and downward edges. The purpose of this modification is query runtime, allowing to look up distances between vertices in S_i instantly instead of plowing through all level edges, however, at the expense of an increased number of level edges.

Constructing the level edge set naïvely would be too expensive in terms of preprocessing time because determining distances in G may in general require consideration of the whole input graph. We suggest an efficient two-pass construction method. In the first, bottom-up, pass, the upward and downward edge sets are computed: computation of U_i and D_i is performed iteratively using the corresponding edge sets at level $i-1$. The second pass is carried out top-down: to construct E_i, the set of level edges at level i help restrict this computation to a bounded local search; the set E_l is computed directly using U_l.

2.1.3. *Parallelization.* Due to the different levels of hierarchy induced by the vertex subsets $\mathscr{S}$, this construction process does not need to consider the whole input graph G at once. On the contrary, the preprocessing can be split up into tasks so that each one operates on exactly one wrapped component (potentially,

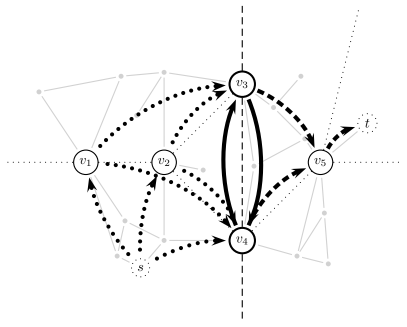


FIGURE 2. The search graph $\mathcal{M}_{st}$: edges from $\mathscr{L}$, $\mathscr{U}_i$, and $\mathscr{D}_i$ are shown as thick lines with solid, dotted, and dashed styles, respectively.

each task could be assigned to a distinct processor, provided that the data flow dependencies between the tasks are obeyed, which would yield speed-up almost linear in the number of processors).

2.1.4. *Component Tree.* The nesting of the separator sets is reflected by the component sets $\mathscr{C}_i$: each component $C_i \in \mathscr{C}_i$ is fully contained in exactly one *parent component* of $\mathscr{C}_{i+1}$, i.e., $C_i \subseteq C'_{i+1}$ for some $C'_{i+1} \in \mathscr{C}_{i+1}$. In addition, we define the *root* or *universe* component $C_{l+1} := G$ that serves as parent for all components in $\mathscr{C}_l$, and a *leaf component* $C_0^v := \{v\}$ for every vertex $v \in V$. For the leaf components, we use C_1^v as parent. The parent relationship naturally induces a tree of components.

2.1.5. *Search Graph.* When for a given pair of vertices $s, t \in V$ simultaneously walking up the component tree from C_0^s and C_0^t towards the root, the paths eventually meet at some component $C_L^s = C_L^t$, the lowest common ancestor of C_0^s and C_0^t. With our notation, the path between C_0^s and C_0^t in the component tree is $(C_0^s, C_1^s, \ldots, C_L^s = C_L^t, \ldots, C_1^t, C_0^t)$. In fact, any s-t path must visit these components in this order.

Now we construct the *search graph* $\mathcal{M}_{st}$, a subgraph of $\mathcal{M}$ with the same s-t distance as in G. The edge set of $\mathcal{M}_{st}$ is the union of the following sets:

$$\begin{aligned}
\mathscr{L} &:= E_{L-1} \cap \left(Adj(C_{L-1}^s) \times Adj(C_{L-1}^t)\right), \\
\mathscr{U}_i &:= U_i \cap \left(Adj(C_{i-1}^s) \times Adj(C_i^s)\right), \text{ and} \\
\mathscr{D}_i &:= D_i \cap \left(Adj(C_i^t) \times Adj(C_{i-1}^t)\right),
\end{aligned}$$

where $i \in \{1, \ldots, L-1\}$.

Figure 2 shows an example, where edges from the sets $\mathscr{L}$, $\mathscr{U}_i$, and $\mathscr{D}_i$ are shown as thick lines with solid, dotted, and dashed styles, respectively. Owing to the altered definition of the level edge set compared to [**24**], we can afford including only a subset of E_{L-1} in the edge set of $\mathcal{M}_{st}$. Note that $\mathscr{L}$ (and therefore also $\mathcal{M}_{st}$) is defined only for $L > 1$. These modifications require a new proof of correctness.

2.1.6. *Correctness.* In the following, we shall prove that $\mathcal{M}_{st}$ can be used for answering the s-t shortest-path query in G. First notice that by definition every edge in $\mathcal{M}_{st}$ has a weight at least as large as the distance between the corresponding vertices in G. Hence, any distance in $\mathcal{M}_{st}$ cannot be smaller than the corresponding

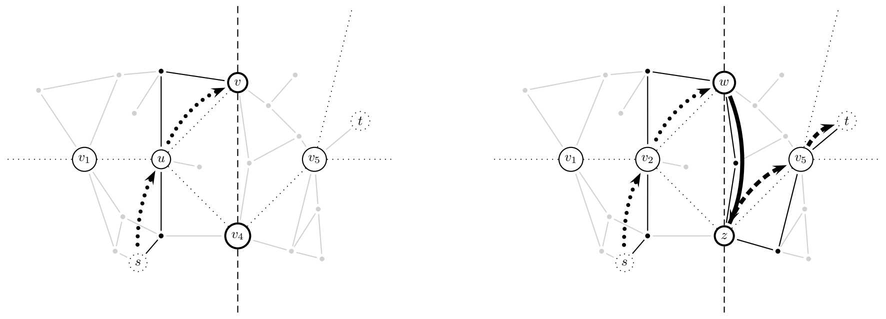


FIGURE 3. Illustration for Lemma 1 (left) and Theorem 2 (right): A shortest path in G (highlighted thin lines) has a corresponding path, of the same length, in $\mathcal{M}_{st}$ (thick lines).

distance in G. It remains to prove that for a shortest s-t path in G there is a path in $\mathcal{M}_{st}$ of equal length.

LEMMA 1. *For $i \in \{0, \ldots, L-1\}$, the distance from s to any vertex $v \in Adj(C_i^s)$ in $\mathcal{M}_{st}$ matches that in G_i^s, the wrapped component around s at level i. Conversely, the distance from any vertex $v \in Adj(C_i^t)$ to t in $\mathcal{M}_{st}$ matches that in G_i^t.*

PROOF. (By induction.) We shall prove only the first part, as the second follows immediately by symmetry. For $i = 0$, the claim is obvious. For $i > 0$, any s-v path in G must contain a vertex in $Adj(C_{i-1}^s)$, let u be the first such vertex. We can split a shortest s-v path at u into two (possibly empty) subpaths. The s-u subpath contains only vertices from G_{i-1}^s and therefore has an equivalent path in $\mathcal{M}_{st}$ by the induction hypothesis. On the other hand, the edge (u, v) is contained in $\mathscr{U}_i$ and its weight corresponds to the length of the u-v subpath. □

THEOREM 2. *If $L > 1$, the s-t distance is equal in the graphs G and $\mathcal{M}_{st}$.*

PROOF. The value L is the level of the lowest common ancestor of C_0^s and C_0^t in the component tree. Therefore, the vertices s and t reside in different home components at level $L-1$, and any s-t path must contain a vertex in S_{L-1}. Let vertex $w \in Adj(C_{L-1}^s)$ and $z \in Adj(C_{L-1}^t)$ be the first or last such vertex, respectively. Again, we split a shortest s-t path at w and z. The s-w and z-t subpaths contain only vertices from G_{L-1}^s and G_{L-1}^t, respectively. According to Lemma 1, these subpaths have equivalent paths in $\mathcal{M}_{st}$. The edge (w, z) is part of $\mathscr{L}$, its weight equals the w-z distance in G. □

2.1.7. *Query.* The search graph $\mathcal{M}_{st}$ can be transformed into an equivalent DAG by creating at most L copies of each vertex. Recall that the edges of $\mathcal{M}_{st}$ are the union of the edge sets $\mathscr{L}$, $\mathscr{U}_i$ and $\mathscr{D}_i$. We call the graph induced by such an edge set a *partial graph*, and distinguish between *level*, *upward*, and *downward parts*, accordingly. A partial graph is a directed bipartite graph, apart from some *twofold* vertices that have both incoming and outgoing edges. We can *unfold* any partial graph into an equivalent directed bipartite graph by creating a copy of each twofold vertex, directing the edges with a twofold vertex as target to the corresponding copies, and adding a zero-cost edge from each original twofold vertex to its copy. In the example in Figure 2, vertex v_4 of the upward part $\mathscr{U}_2$ is twofold, because it has an incoming edge (v_1, v_4) and an outgoing edge (v_4, v_3). Unfolding this partial graph generates a copy $v_4{}'$ of v_4 with new edges $(v_1, v_4{}')$, $(v_3, v_4{}')$ and $(v_4, v_4{}')$, the latter having a length of zero.

After that, each vertex of the unfolded partial graph has either only outgoing or only incoming edges; we distinguish between *source* and *drain* vertices. An unfolded version of the search graph can be created by joining the unfolded partial graphs. The drain vertices of one partial graph match the source vertices of another partial graph. As such, the join can be interpreted as a stacking of partial graphs. All paths traverse this stack in the same direction, thus no cycles exist. Refer to [**18**] for a formal proof.

For a DAG, an *s*-*t* shortest-path query can be performed in $O(V + E)$ time. Since the topological structure of our DAG is known in advance, the query algorithm can be reduced to initialization of the vertex distance labels and update of the distance label of each edge's target vertex in the order imposed by the topological structure.

2.1.8. *Nearby Vertices.* Path lookup in $\mathcal{M}_{st}$ works only for $L > 1$, i.e., for source and target vertices from different home components at level 1. For vertices from the same home component $C = C_1^s = C_1^t$, we fall back to Dijkstra's algorithm. However, we avoid leaving the home component in our search. Instead, we use the appropriate edges in $E_1 \cap (Adj(C) \times Adj(C))$ as shortcut for paths that leave C. By keeping the components small, we can state a runtime guarantee for the case of nearby vertices, too.

2.2. Optimizing Partial Graphs. In contrast to the classic variant, where a multi-level graph is stored as a whole, we spread it over a large number of partial graphs (as seen before, any search graph can be constructed through the union of a number of appropriate partial graphs). The foremost advantage is that each of them can be optimized individually using two different techniques: pruning of edges that cannot contribute to a shortest path and conversion of a partial graph into an equivalent one with more vertices but fewer edges.

2.2.1. *Pruning Superseded Edges.* Consider an upward part $\mathscr{U}_i$ at level $i > 1$. This partial graph connects the adjacent vertices of two related components C_{i-1} and C_i at neighboring levels; let G_{i-1} and G_i be the corresponding wrapped components, and consider a fixed edge $(w, v) \in \mathscr{U}_i$. Now, if a shortest *w*-*v* path in G_i passes another vertex $z \in Adj(C_{i-1})$ and the *w*-*z* subpath does not leave G_{i-1}, then for any *s*-*v* path via w there is a path via z that is no longer. This also holds for any search graph that uses $\mathscr{U}_i$: for any *s*-*v* path via edge (w, v) there is a path via edge (z, v) that is no longer. That is, we can safely remove the edge (w, v) from the upward part $\mathscr{U}_i$; this edge is called *superseded* by the edge (z, v). An example is shown in Figure 4.

To determine if an edge $(w, v) \in \mathscr{U}_i$ is superseded by another edge, we only need to check local distances between adjacent vertices of C_{i-1}, together with edge weights of the upward part $\mathscr{U}_i$. Edge (w, v) is superseded by edge $(z, v) \in \mathscr{U}_i$, if $c(w, v) = d(w, z) + c(z, v)$ and $d(w, z) > 0$.[1]

The pruning algorithm simply checks each pair of edges in $\mathscr{U}_i$ sharing the same target vertex for supersedement, and removes the superseded ones on the fly. Because supersedement is a strict partial order, the algorithm finds and removes all

[1]Here, c denotes the edge weight function in $\mathscr{U}_i$, and d refers to the local distance in G_{i-1}.

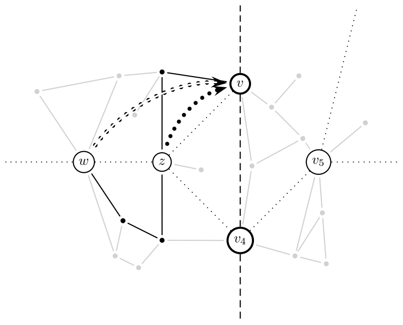


FIGURE 4. Superseded edges. Edge (w, v) (double-dotted line) is superseded by (z, v) (dotted line), because there is a shortest w-v path via z in the input graph (highlighted thin lines).

desired edges (cf. [**18**] for a formal proof). Analogously, we can eliminate superseded edges in downward and level parts. All partial graphs are optimized separately: an edge superseded in one may or may not be superseded in another partial graph.

The pruning algorithm can be further refined by determining *superseded vertices* in a first pass. For upward parts, a vertex w is superseded by another vertex z iff all edges starting at w are superseded by the corresponding edge starting at z. Superseded vertices can be omitted completely in the further processing, because all edges connected to this vertex are superseded. To remove superseded vertices, we iterate over the pairs of source vertices (those with outgoing edges) and check for supersedement by iterating over all outgoing edges. Downward and level parts are handled likewise. For the partial graphs obtained in our experimental evaluation, this first pass takes only a fraction of the time needed for removing all superseded edges, and usually removes many candidate edges, significantly reducing the execution time for the main pruning algorithm.

2.2.2. *Constructing Equivalent Graphs.* A further optimization technique is based on the following idea. The number of edges of a partial graph can be reduced by introducing auxiliary vertices and replacing many original edges with few edges through the new vertices such that distances are preserved. An example is given in Figure 5. Edges highlighted in the left graph are contained in at least one shortest σ-δ path. The edges in the corresponding partial graph are made up from the lengths of these shortest paths. Without optimization this would lead to a complete bipartite graph with 16 edges, while an optimized equivalent graph needs only eleven edges, as shown at the right: the twelve edges from $\{\sigma_1, \ldots, \sigma_4\}$ to $\{\delta_1, \ldots, \delta_3\}$ may be replaced with a star-like graph of seven edges, since the corresponding shortest paths between the vertices in question contain a shared central vertex. In the best case, all underlying shortest paths contain at least one common vertex, and the optimization results in a star-like graph.

We have implemented a simple heuristic that adds a single, so-called *central*, vertex, decides in a greedy manner which of the original vertices to connect to the central vertex, and balances the weight function for the new edges. The goal of the balancing is to remove a maximal number of original edges: if a path via the central vertex is of equal length as the corresponding original edge, the latter can

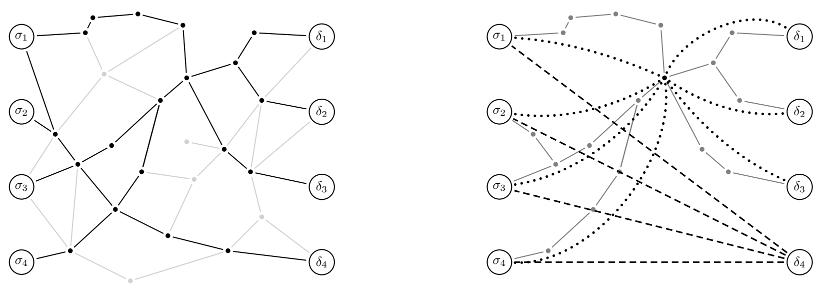


FIGURE 5. Constructing equivalent graphs. Left: sample graph; highlighted edges are contained in a shortest σ-δ path. Right: belonging search graph with edge compression applied; dotted and dashed edges are contained in the graph.

be removed. The balancing must be distance preserving; in particular, no newly introduced path may be shorter than the corresponding original edge.

3. Experiments

As input data to our experimental evaluation, we use road networks of Western Europe, provided by PTV AG for scientific use, and the USA, taken from the TIGER/Line Files [**25**], with around 18 million vertices and 42.6 million edges and 23.9 million vertices and 58.3 million edges, respectively. For both graphs, both distances and travel times are available for each edge; in order to compare our approach to similar ones, we test our graphs also with the unit edge metric.

For the test runs, we used a machine with two AMD Opteron 2218 processors, 32 GB of shared RAM, and 2×1 MB of L2 cache, where each processor features two cores clocked at 2.6 GHz; the time measurements given refer to execution on a single core. The program was compiled with the GCC 3.4, using optimization

TABLE 1. DIMACS Challenge benchmarks: query times in ms for subgraphs of the US network and different metrics.

	metric	
graph	time	dist
NY	25.4	23.0
BAY	29.9	29.0
COL	43.2	38.8
FLA	119.3	112.0
NW	143.6	143.0
NE	193.2	195.4
CAL	254.5	248.8
LKS	396.6	377.5
E	607.1	558.5
W	1 343.9	1 115.4
CTR	5 892.9	4 778.0
USA	7 741.4	5 908.5

level 3 and the LEDA library (version 5.01). Results on the DIMACS Challenge benchmarks can be found in Table 1.

The subsequent presentation of our results is structured along the two ways of obtaining a *hierarchical decomposition* of the input graphs, planar separators and METIS, the latter being a freely available tool for graph partitioning.

Decomposition. To determine for a given graph sets of selected vertices, the graph is first decomposed in a hierarchical fashion. This process is governed by two parameters, the number of levels and *granularity*, the latter being fixed either through the maximum component size allowed or a maximum number of adjacent vertices per component for each level. Both options have their advantages: limiting the component size generates a balanced decomposition in the sense that each higher-level component contains roughly the same number of lower-level components, while limiting the number of adjacent vertices yields a smaller variance in the search graph sizes and thus allows for predicting the maximum search graph size even before starting the preprocessing. Given an input graph, a hierarchical decomposition is obtained by declaring, iteratively for each level to be generated, vertices selected so long until the granularity criterion for that level is reached. For the next-lower level, repeat this process applied to the decomposition found so far with the specific granularity.

3.1. Planar-Separator Theorem.

Decomposing our graphs by means of the planar-separator theorem (PST) [**16, 8**] requires some preparatory step: due to their nature, road graphs are but almost planar since they account for bridges, highway ramps, etc., incurring crossings in those very places. We therefore planarize the input graph first by adding vertices at edge crossings, and eventually have to appropriately retranslate the separation found for the planarized graph into one satisfying the original graph. The planarized input graph is recursively split into two parts so long until the granularity condition holds. Our experiments involve a three-level decomposition with a granularity of 80-40-20 adjacent vertices per component. Note that such a granularity naturally induces an upper bound of $80^2 + 2 \cdot 40 \cdot 80 + 2 \cdot 20 \cdot 40 + 2 \cdot 20 = 14\,440$ edges in the search graph, which can be stated even *before* running our preprocessing.

3.1.1. *Preprocessing.* Decomposition with PST takes about a week for the Europe graph (distance metric),[2] but our implementation caches the course of the computation, and this cache can be reused to create decompositions with any given granularity with just little computational effort. Preprocessing requires about 24 hours on one core and about 8 hours on four cores, resulting in 543 million edges, 288 million of which belong to upward and downward graphs at level 1.

Each edge in a partial graph can be encoded using six bytes (four for the length assigned to it and one each for source and target vertices). Hence, the space needed to store the whole preprocessed data amounts to an overhead of 181 additional bytes per vertex (543 million edges · 6 bytes per edge / 18 million vertices) of the input graph. If we skipped the optimization of the partial graphs, the preprocessing would contain 3 210 million edges; in other words, the optimization step reduces the preprocessing size to 17 %.

For each kind of partial graphs, optimization has a different impact. For all higher-level partial graphs, roughly half of the edges are removed by supersedement.

[2]We used METIS to create an initial decomposition into components of about 500 000 vertices each, as our PST implementation is unable to process the input graph as a whole.

However, superseded vertices only exist in level graphs. For most level graphs much smaller equivalent graphs can be found, reducing the total size of the level graphs to 7.5 % in combination with supersedement when the distance metric is used. Note that with time or unit metric, this figure drops to 4.9 % and 5 %, respectively. Our heuristics for computing equivalent graphs has almost no effect for upward and downward graphs. Summarizing, our edge reduction heuristics work very well for level graphs, but yield only moderate results for upward and downward graphs.

Recall that the granularity of the preprocessing naturally induces an upper bound of 14 440 edges in the search graph. However, an analysis of the preprocessed data reveals that due to optimization, the largest search graph contains only 5 262 edges.

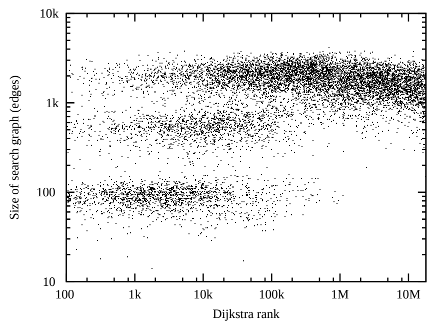
3.1.2. *Query Performance.* Unless otherwise stated, we use the road network of Europe, applying the distance metric. We evaluate 10 000 queries picked at random according to an exponential distribution, with a *Dijkstra rank*[3] of between 100 and $|V|$ (we chose an exponential over a uniform distribution as the former captures the intuition that with real-world information systems, short-distance occur more frequently than long-distance queries). Note that only those queries are reported that can be handled by our approach, i.e., s-t queries with s and t in different home components. However, the entry and exit graphs of our technique propose distances from and to all boundary vertices within one component, which could be used as landmark data for low-range queries, i.e., queries within one component. As observed in [**6, 3**], landmark-based routing performs very well for those types of queries (below 1 ms, except for outliers).

We measured the time needed to initialize an array of distance labels and to relax all edges of all partial graphs the search graph consists of. Prior to this, the query algorithm loads the partial graphs from external memory and flushes the L2 cache by performing copy operations on two memory buffers, each as large as the cache. This setup accurately simulates a client-server system that answers random source-target queries and holds all partial graphs in RAM: in general, a partial graph needed for a query is not present in the L2 cache and must be fetched from RAM. (Note that to reduce the impact of outliers, we repeated three times the execution of each query and used the median of the three measurements.)

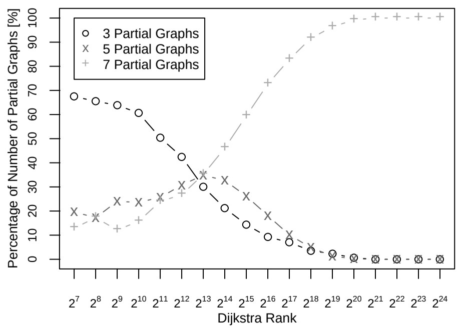
Figure 6(a) shows the search space size, plotted against the Dijkstra rank, where each dot represents a query. Because our technique is dominated by relaxed edges, search space is measured in these terms rather than by settled vertices (experiments show that the dependency between rank and relaxed edges is indeed linear for Dijkstra's algorithm).

There are three horizontal clouds discernible, at approximately 100, 500, and 2 000 of relaxed edges. They correlate with the number of upward and downward graphs in the search graphs of between 1 and 3, resulting in search graphs constructed from three, five, or seven partial graphs. This observation is supported by Figure 6(b), which depicts the number of partial graphs depending on Dijkstra rank. As expected, with increasing rank the search graphs tend to comprise more partial graphs. Altogether, is seems as if the number of relaxed edges depends more on the number of partial graphs from which the search graph is constructed than on the pure rank of a query.

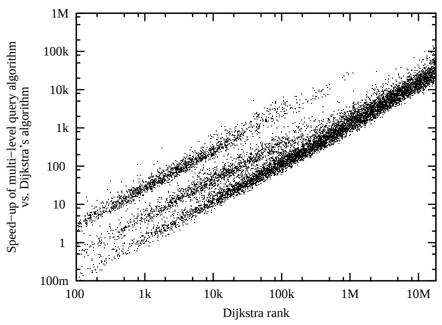
[3]For an s-t query, the Dijkstra rank of vertex v is the number of vertices settled before v is settled.



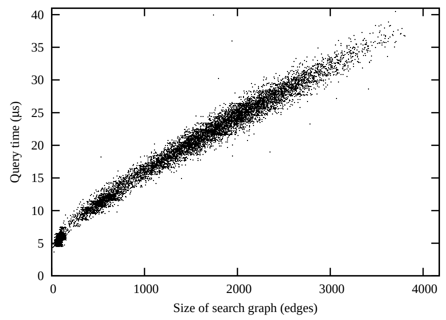
(a) Search space in terms of relaxed edges. Each dot depicts for one query the number of relaxed edges in relation to its Dijkstra rank.



(b) Distribution of number of partial graphs (the search graph is constructed from) in relation to Dijkstra rank.



(c) Speed-up of our approach over Dijktra's algorithm in terms of relaxed edges.



(d) Query time in relation to the search graph size.

FIGURE 6. Results for our approach using the road network of Europe as input. As metric, distances are applied.

In terms of search space, we achieve speed-ups of up to 100 000 for higher Dijkstra ranks (cf. Figure 6(c)). However, a couple of small-rank queries lead to factors of less than 1: such a slow-down occurs when source and target vertices are close to each other in the input graph, but belong to different home components at level 2 or even 3 so that relatively big five- or seven-level search graphs have to explored.

Figure 6(d) depicts search graph size depending on query time: we observe an almost linear relationship. In general, all queries are executed in less than 40 μs. Moreover, for search graphs with 2 000 edges or more, we can state a query runtime of roughly 12 ns per edge.

3.1.3. *Robustness.* Up to now, we have shown that our approach performs very well with the distance metric. In order to prove robustness to the metric applied, we ran a larger series of experiments with both the Europe and the US networks and travel times, distances, and unit lengths. To facilitate comparison of our approach to similar ones, we now employ queries distributed uniformly at random. Table 2 reports average running times as well as the percentages of queries executed.

TABLE 2. Preprocessing and (uniform) random queries performance for different metrics on the European and US network. The size of the preprocessing is given in number of edges in all generated partial graphs. The search space is given in number of relaxed edges within search graph. Note that only those queries are reported which can be performed by our approach. The percentage of executed queries is given in column 6.

		PREPRO	QUERY		
		size	search space	time	executed
graph	metric	[# edges]	[# relaxed edges]	[μs]	[%]
Europe	time	469 M	1 494	18.8	99.90
	dist	543 M	1 617	20.3	99.96
	unit	470 M	1 485	19.3	99.93
USA	time	782 M	1 462	19.3	99.94
	dist	848 M	1 547	20.0	99.94
	unit	774 M	1 441	19.2	99.97

We observe that the metric chosen has almost no impact on query times or preprocessing. Regarding partial-graph optimization, there are no significant differences in the number of additional edges between the various metrics, either. Of all queries, more than 99.9 % were executed; assuming an average time of about

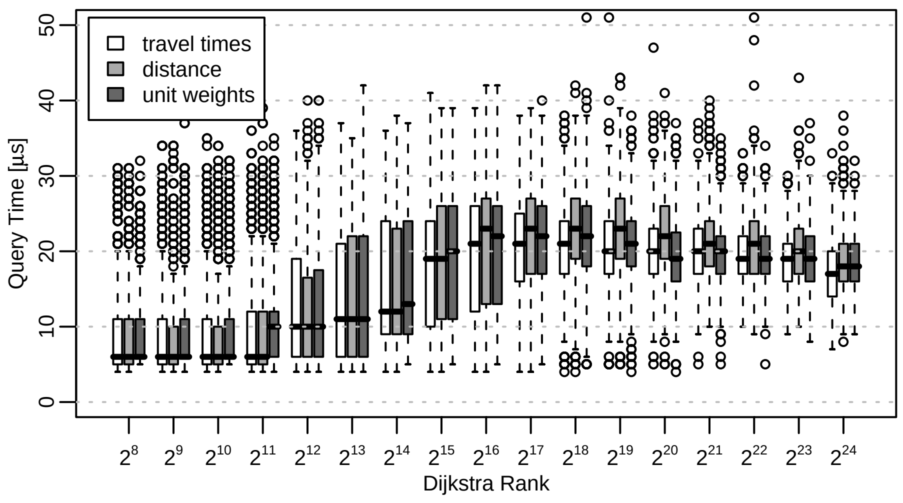


FIGURE 7. Comparison of the query times for different metrics (travel times, distances and unit lengths) using the Dijkstra rank methodology [**20**] on the road network of Europe. The results are represented as box-and-whisker plot [**19**]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually. Note that only those queries are reported that can be performed by our approach.

TABLE 3. Preprocessing and (uniform) random queries performance for subgraphs of the US networks, taken from the DIMACS homepage. As metric, we apply travel times. Our current implementation cannot handle small graphs. Thus, no results for NY, BAY, and COL are given. The search space is given in number of edges within the partial graphs. In addition to the columns given in Table 2 we also report the ratio of additional edges per node in the original graph (column 4). Note that only those queries are reported which can be performed by HPML. The percentage of executed queries is given in column 7.

	PREPROCESSING			QUERY		
	time	size	#edges	search	time	executed
graph	[min]	[# edges]	$/\|V\|$	space	[μs]	[%]
FLA	8	19 M	17.8	331	8.6	90.00
NW	7	22 M	18.2	243	8.9	71.40
NE	444	141 M	92.8	291	8.4	98.60
CAL	324	136 M	72.0	596	11.2	98.40
LKS	320	130 M	47.1	1 071	14.8	99.30
E	39	71 M	19.7	1 824	21.0	99.60
W	89	133 M	21.3	1 440	18.6	99.80
CTR	260	354 M	25.1	1 847	21.6	99.97

1 *ms* for the remaining 0.1 % of (low-range) queries, the average query times as of Table 2 would increase by 1 μs. Hence, we wind up with an overall average time of less than 22 μs for all inputs.

In order to check whether this robustness with respect to metrics holds for all types of queries (low-, mid-, and long-range), Figure 7 shows the performance of our approach using the Dijkstra rank methodology [**20**]. As input we use the European instance. Strikingly, the performance of our approach is (almost) independent of the applied metric for all types of queries.

All of the above-said is confirmed by experiments with different subgraphs of the US network and distance metric, the results being summarized in Table 3. However, on smaller graphs the number of executed queries drops to values of 71.4%. Preprocessing times differ greatly from that for the whole graph as now only two instead of three levels are used.

3.1.4. *Comparison.* Table 4 contrast the results of our approach to the most prominent speed-up techniques presented at the DIMACS workshop applied to all graphs and metrics, with queries distributed uniformly at random.

The results show clearly that in terms of preprocessing time, HPML cannot compete with any other technique. Comparing query times with the time metric, our approach as well as the TNR-variants all yield values of less than 20 μs, where the latter techniques still outperform ours. The very strength of our approach unfolds when the distance metric is used: HPML query times do not change significantly, while with transit node routing they increase by factors of up to 26. Using the unit metric, all of these approaches yield similar performance. To sum up, our findings corroborate the robustness of our approach regarding edge metric, which does not hold for transit node routing.

TABLE 4. Performance of the most prominent speed-up techniques in comparison to our high-performance multi-level (HPML) approach. More precisely, we report preprocessing and query times for highway hierarchies star (HH*) [**3**], REAL [**6**], grid-based transit node routing (grid-TNR) [**1**], and transit node routing based on highway hierarchies (HH-TNR) [**22**].

		Europe		USA	
		PREPRO	QUERY	PREPRO	QUERY
metric	technique	[h:mm]	[μs]	[h:mm]	[μs]
time	HH*	0:22	550	0:28	600
	REAL	2:20	1110	2:01	1050
	HH-TNR	1:15	4.3	1:25	3.3
	Grid-TNR	58:00	13	7:00	17.8
	HPML	$\approx$ 24:00	18.8	$\approx$ 36:00	19.3
dist	HH*	0:49	1950	0:59	1740
	REAL	1:30	1160	2:18	1800
	HH-TNR	2:42	37.6	3:37	86.1
	Grid-TNR	29:00	56	9:00	69.4
	HPML	$\approx$ 24:00	20.3	$\approx$ 36:00	20.0
unit	HH*	0:27	990	0:32	890
	REAL	3:49	1140	2:27	1160
	HH-TNR	0:53	13.1	3:59	19.8
	Grid-TNR	17:00	12	9:00	30.3
	HPML	$\approx$ 24:00	19.3	$\approx$ 36:00	19.2

3.2. METIS. As an alternative to compute graph decompositions we also use the METIS collection [**13**]. These tools allow to divide graphs into a given number of *partitions* of roughly equal size, where the edge cut, i.e., the number of edges with source and target vertex located in different partitions, is minimized. Since our preprocessing technique requires a selection of vertices instead of edges, we subsequently compute a greedy vertex cover on the edge cut obtained from METIS. This procedure can be carried out recursively to obtain a hierarchical decomposition.

METIS runs amazingly fast for our test instances: decomposition into any number of components requires less than one minute compared to one week for PST; thus, a hierarchical decomposition can be obtained in about ten minutes. As a further advantage, the input graph does not have to be planar. On the other hand, METIS produces separators that are about 20 % larger than those generated by PST; similar results were reported in [**8**]. While a decomposition with limited component sizes can be obtained quite naturally using METIS, we cannot easily create a decomposition with limited maximum number of adjacent vertices per component; to achieve the latter, we have to perform recursive two-way partitioning, as described for PST.

For our evaluation, we settled for a decomposition into three levels, as preliminary experiments showed that two levels would consume too much space, while employing a fourth level would not pay off. Further tests suggested granularities of 120 000-4 000-360 of adjacent vertices for the Europe and 120 000-3 300-300 for the US network. To compare METIS with PST, we used the very same granularity also for PST: compared to the granularity used in Section 3.1, the preprocessing time for the Europe graph and distance metric is slightly smaller, whereas search graph size doubles on average.

Unfortunately, the overall results were not as promising as with adjacent-vertex granularities. When METIS is used, the size of preprocessed data grows by 50 % and the average search graph size doubles. For all preprocessings, the maximal search graph was more than ten times larger than the average; this is unfavorable if a tight guarantee for query times is required. Summing up, the quality of the decomposition is of utmost importance for the efficiency of our speed-up technique.

4. Conclusion

We have shown how to enhance the classic multi-level approach [**24**] in such a way that an even greater deal of the effort to compute a shortest path can be shifted to the preprocessing stage. The main developments concern distribution of the multi-level graph to many partial graphs. These permit to be sparsified fairly easily, which leads to massive reduction of the total amount of precomputed data and hence of query times.

In an experimental study with road graphs, where an extensive preprocessing as well as larger amounts of additional data could be afforded, our approach proved highly effective: speed-ups achieved over Dijkstra's algorithm reach factors of up to around 3 000. Furthermore, our approach has shown to be robust to different edge metrics with respect to both preprocessing and query performance: except for very few outliers, query time does not exceed 40 μs.

For future work, we see the following points of improvement: concerning our implementation, use of custom-tailored data structures as well as of locality, as exploited in [**6, 3**]; at an algorithmic level, development of alternative heuristics to construct equivalent graphs, and combination with other speed-up techniques (in [**10, 5**], certain combinations were shown to perform better than the individual techniques). Another interesting question would be that of dynamization: due to the hierarchical nature of our approach, we believe that only small parts of the preprocessed data need to be updated upon an edge change in the input graph. Finally, storage and retrieval of the course of a shortest path is a major requirement for practical applications, which could be solved through similar concepts as in [**11**].

References

1. Holger Bast, Stefan Funke, and Domagoj Matijevic, *TRANSIT: Ultrafast Shortest-Path Queries with Linear-Time Preprocessing*, 9th DIMACS Challenge on Shortest Paths, November 2006, An updated version of the paper appears in this book.
2. Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes, *In transit to constant time shortest-path queries in road networks*, 9th Workshop on Algorithm Engineering and Experiments (ALENEX), 2007, pp. 46–59.
3. Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner, *Highway Hierarchies Star*, 9th DIMACS Challenge on Shortest Paths, November 2006, An updated version of the paper appears in this book.
4. Edsger W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik **1** (1959), 269–271.
5. Andrew Goldberg, Haim Kaplan, and Renato Werneck, *Reach for A*: Efficient point-to-point shortest path algorithms*, Proc. Algorithm Engineering and Experiments, SIAM, 2006, pp. 129–143.

6. Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck, *Better Landmarks within Reach*, 9th DIMACS Challenge on Shortest Paths, November 2006, An updated version of the paper appears in this book.
7. Ronald J. Gutman, *Reach-based routing: A new approach to shortest path algorithms optimized for road networks.*, Proc. Algorithm Engineering and Experiments, SIAM, 2004, pp. 100–111.
8. Martin Holzer, Grigorios Prasinos, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis, *Engineering planar separator algorithms*, Proc. European Symposium on Algorithms, LNCS, vol. 3669, Springer, 2005, pp. 628–639.
9. Martin Holzer, Frank Schulz, and Dorothea Wagner, *Engineering multi-level overlay graphs for shortest-path queries*, Proc. Algorithm Engineering and Experiments, SIAM, 2006, pp. 156–170.
10. Martin Holzer, Frank Schulz, and Thomas Willhalm, *Combining speed-up techniques for shortest-path computations*, Proc. Workshop on Experimental and Efficient Algorithms, LNCS, vol. 3059, Springer, 2004, pp. 269–284.
11. Ning Jing, Yun-Wu Huang, and Elke A. Rundensteiner, *Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation*, IEEE Trans. Knowledge and Data Engineering **10** (1998), no. 3, 409–432.
12. Sungwon Jung and Sakti Pramanik, *HiTi graph model of topographical roadmaps in navigation systems*, Proc. Data Engineering, IEEE Computer Society, 1996, pp. 76–84.
13. George Karypis, *METIS*, 1998, `http://www-users.cs.umn.edu/~karypis/metis`.
14. Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling, *Acceleration of shortest path and constrained shortest path computation.*, Proc. Workshop on Experimental and Efficient Algorithms, Springer, 2005, pp. 126–138.
15. Ulrich Lauther, *An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background*, Geoinformation und Mobilität — von der Forschung zur praktischen Anwendung, vol. 22, IfGI prints, Institut für Geoinformatik, Münster, 2004, pp. 219–230.
16. Richard J. Lipton and Robert Endre Tarjan, *A separator theorem for planar graphs*, SIAM Journal on Applied Mathematics **36** (1979), no. 2, 177–189.
17. Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm, *Partitioning graphs to speed up Dijkstra's algorithm.*, Proc. Workshop on Experimental and Efficient Algorithms, LNCS, vol. 3503, Springer, 2005, pp. 189–202.
18. Kirill Müller, *Design and implementation of an efficient hierarchical speed-up technique for computation of exact shortest paths in graphs*, Master's thesis, Department of Informatics, University of Karlsruhe, Germany, June 2006, online available at `http://i11www.iti.uka.de/teaching/theses/files/da-kmueller-06.pdf`.
19. R Development Core Team, *R: A Language and Environment for Statistical Computing*, `http://www.r-project.org`, 2004.
20. Peter Sanders and Dominik Schultes, *Highway hierarchies hasten exact shortest path queries*, Proc. European Symposium on Algorithms, LNCS, vol. 3669, Springer, 2005, pp. 568–579.
21. ———, *Engineering highway hierarchies*, Proc. 14th European Symposium on Algorithms, LNCS, vol. 4168, Springer, 2006, pp. 804–816.
22. ———, *Robust, Almost Constant Time Shortest-Path Queries on Road Networks*, 9th DIMACS Challenge on Shortest Paths, November 2006, An updated version of the paper appears in this book.
23. Frank Schulz, Dorothea Wagner, and Karsten Weihe, *Dijkstra's algorithm on-line: An empirical case study from public railroad transport*, Proc. Workshop on Algorithm Engineering, Springer, 1999, Also published in: ACM Journal of Experimental Algorithms **5** (2000), pp. 110–123.
24. Frank Schulz, Dorothea Wagner, and Christos Zaroliagis, *Using multi-level graphs for timetable information in railway systems*, Proc. Algorithm Engineering and Experiments, LNCS, vol. 2409, Springer, 2002, pp. 43–59.
25. U.S. Census Bureau, Washington, DC, *UA Census 2000 TIGER/Line Files*, `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html`, 2002.
26. Dorothea Wagner and Thomas Willhalm, *Geometric speed-up techniques for finding shortest paths in large sparse graphs*, Proc. European Symposium on Algorithms, LNCS, vol. 2832, Springer, 2003, pp. 776–787.

27. Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis, *Geometric containers for efficient shortest-path computation*, ACM Journal of Experimental Algorithmics **10** (2005), 1–30.

Daniel Delling, Universität Karlsruhe (TH), Fakultät für Informatik, Postfach 69 80, 76128 Karlsruhe, Germany
E-mail address: `delling@ira.uka.de`

Martin Holzer, Universität Karlsruhe (TH), Fakultät für Informatik, Postfach 69 80, 76128 Karlsruhe, Germany
E-mail address: `mholzer@ira.uka.de`

Kirill Müller, Universität Karlsruhe (TH), Fakultät für Informatik, Postfach 69 80, 76128 Karlsruhe, Germany
E-mail address: `mail@kirill-mueller.de`

Frank Schulz, PTV Planung Transport Verkehr AG, Stumpfstrasse 1, 76131 Karlsruhe, Germany
E-mail address: `frank.schulz@ptv.de`

Dorothea Wagner, Universität Karlsruhe (TH), Fakultät für Informatik, Postfach 69 80, 76128 Karlsruhe, Germany
E-mail address: `wagner@ira.uka.de`