High-Performance Multi-Level Graphs*

Daniel Delling† Martin Holzer† Kirill Müller† Frank Schulz‡
Dorothea Wagner†

August 28, 2006

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 [SWZ02, HSW06], 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. Experiments confirm query times of less than 200 μs for a road graph with over 15 million vertices.

1 Introduction

Computation of shortest paths is a central requirement for many applications, such as route planning or search in huge networks. Facing real-world data of ever-growing size, 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. In this work, we opt for the second way.

Since its publication in 1959, Dijkstra's famous algorithm for calculation of shortest paths in a directed graph with nonnegative edge weights [Dij59] 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

*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).

†Universität Karlsruhe, Institut für Theoretische Informatik, Lehrstuhl für Algorithmik, Postfach 69 80, 76128 Karlsruhe, Germany. E-mail: {delling,mholzer,wagner}@ira.uka.de, mail@kirill-mueller.de.

‡PTV Planung Transport Verkehr AG, Stumpfstraße 1, 76131 Karlsruhe, Germany. E-mail: frank.schulz@ptv.de.

algorithm. Recent preprocessing techniques [GKW06, SS06] yield for graphs of similar sizes considerable average query times of around 2 ms and just under 1 ms, respectively, while maintaining optimality of the solution.

In this work, preprocessing is developed to the maximum. We present a further development of the multi-level technique given in [SWZ02], which is based on a hierarchical decomposition of the input graph and computation of an auxiliary graph with additional information. The use of this precomputed data at the on-line stage allows for reduction in search space and, consequently, query time. The new variant at hand 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 [SWZ02, HSW06] the multi-level approach was shown to be effective for graphs of up to 100 000 vertices, we are now able to handle graphs that are comparatively huge within reasonable time.

During the preprocessing stage, instead of one single multi-level graph we compute a large number of small (partial) graphs. The advantage of having multiple graphs is that each of them can be optimized individually. This is achieved by two measures: first, omission of so-called *superseded* edges, i.e. edges for which there exists a path in the partial graph with the same length; and second, transformation of the partial graphs into *equivalent* graphs, i.e. graphs that preserve all shortest paths but have fewer edges. What is more, we make use of the fact that the preprocessing is parallelizable. We show that for each possible query, there is a search graph combined of several partial graphs that preserves the distance between the dedicated vertices. This search graph is acyclic, and we give a simple, linear-time procedure to find a shortest path in it.

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. For an implementation that keeps 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, the query performance of our technique clearly outperforms other recent approaches: with our implementation, we obtain query times of less than 200 μs for a graph with roughly 15 million vertices, representing parts of the Western European road network.

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 [SWZ02], and shows the various refinements made. An experimental study is presented in Section 3, and we conclude with some final remarks on future aspects to be addressed.

1.1 Related Work

There are a bulk of techniques to speed up single-pair shortest-path algorithms (see [WW06] for a survey), most of which rely on Dijkstra's shortest-path algorithm [Dij59]. In this section, we focus on methods that require a preprocessing step, in which some additional information is determined beforehand; for answering a shortest-path query, the on-line search can then fall back on this data. We differentiate between the following two types: first, speed-up techniques that precompute additional data attached to the graph's vertices or edges, permitting the on-line algorithm to quickly decide which parts of the graph can be pruned [WW03, Gut04, Lau04, MSS$^+$05, GKW06]; and second, techniques that in a hierarchical fashion determine an auxiliary graph, a slender part of which typically suffices at the on-line stage to answer a shortest-path query [SWZ02, HSW06, JP96, JHR98, SS05, SS06].

We want to briefly review the latter papers and point out their relationship to ours. As mentioned above, the method presented in this work uses the same basic concepts as the technique in [SWZ02, HSW06] (which we will occasionally refer to as *classical* multi-level technique), with the following enhancements. 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 graph; the on-line search can then be reduced to a simple linear-time procedure on that acyclic graph.

The *HiTi model* by Jung and Pramanik [JP96] is very similar to the classical 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 [JHR98], various partial graphs are computed, which are later combined appropriately to form a search graph for a given query. No optimization for the single parts is applied, but a compression technique is used to also keep track of the course of shortest paths. Finally, a recent technique named *highway hierarchies*, introduced by Sanders and Schultes [SS05], computes a hierarchy of coarsenings of the input graph. The search algorithm proceeds in a bidirectional fashion and during its course needs to consider vertices of only one layer at a time.

2 Multi-Level Graphs

This section, subdivided into two parts, is devoted to the formal description of our High-Performance Multi-Level Graphs. The first part reviews the classical multi-level technique and highlights some enhancements to it; this description follows [SWZ02], Section 2. In the second, we present the main contribution of this work, optimization of the multi-level graph to allow for even smaller search spaces and query times.

2.1 Enhancing Multi-Level Graphs

A multi-level graph $\mathcal{M}$ extends a weighted digraph $G = (V, E)$ by adding multiple *levels* of edges. For a pair of vertices $s, t \in V$, a subgraph 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 than G does. In addition, the search graph can be transformed into an acyclic graph; hence, distances can be computed in linear time.

Separator Sets 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: $V \supset S_1 \supset S_2 \supset \cdots \supset S_l$. For best performance, the graph $G - S_i$ falls apart into many components of similar size, while $|S_i|$ is small compared to $|V|$. Such separator sets can be obtained, for instance, through repeated application of the Planar-Separator Theorem [LT79, HPS$^+$05].

For the decomposed graph $G - S_i$, we shall use the following definitions, for which Figure 1 gives an example.

- By $\mathscr{C}_i$, we denote the set of 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 $i \in \{0, \dots, l\}$ and $v \in S_i$, and assume $S_0 = V$ for that. (This is a slight difference to [SWZ02].)
- 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 any direction. The set of all vertices adjacent to C is denoted by $Adj(C)$. For $v \in S_i$ (i.e., $C_i^v = \{v\}$), we define $Adj(C_i^v) := \{v\}$.
- Component C_i^v together with its adjacent vertices is called *wrapped component* and denoted by $G_i^v = G \cap (V(C_i^v) \cup Adj(C_i^v))$.

Multi-Level Graph In addition to G, the sequence of separator sets $\mathscr{S}$ is used as input to construct the multi-level graph $\mathcal{M}$. A set of edges determines each level of $\mathcal{M}$. 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 $D_i \subseteq S_i \times S_{i-1}$.

The candidate sets for upward and downward edges are somewhat larger in this work compared to [SWZ02] in that they used to be restricted to $S_{i-1} \times (S_{i-1} \setminus S_i)$ and $(S_{i-1} \setminus S_i) \times S_{i-1}$, respectively. This augmentation is necessary for the subsequent optimization.

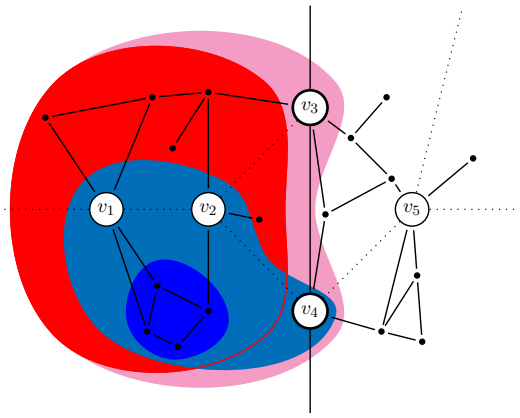

Figure 1: Hierarchy.
Components (dark colors) and belonging wrapped components (light colors) at level 1 (blue) and 2 (red), respectively. Note that vertex v_4 is a separator vertex of both level 1 and 2.

Construction For a level $i \in \{1, \dots, l\}$, a candidate edge (v, w) 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 endpoint vertices must be contained in the same wrapped component G_i. The weight of an upward or downward edge is determined by the length of a shortest of these 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 [SWZ02], where level edges were defined similarly to upward and downward edges. The purpose of this modification is query runtime: This allows 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 the distance 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, just as the classical multi-level technique proposes. Computation of U_i and D_i, the upward and downward edges at level i, is then performed 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, the level edges at level $i+1$ are used to restrict this computation to a bounded local search; the set E_l is computed directly using U_l.

Parallelization Due to the hierarchical decomposition, the construction process does not need to consider the whole input graph G at once. On the contrary, the preprocessing can be split up in tasks so that each task operates on exactly one wrapped component. Potentially, each task can be assigned to a distinct processor, provided that the data flow dependencies between the tasks are obeyed. The preprocessing speed-up achievable by that is almost linear in the number of processors used. For details on both the construction process and its parallelization, refer to [Mül06], Section 5.

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.

Search graph For the remainder of this section, we focus on a given pair of vertices $s, t \in V$. When simultaneously walking the componenent tree from C_0^s and C_0^t towards 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:

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

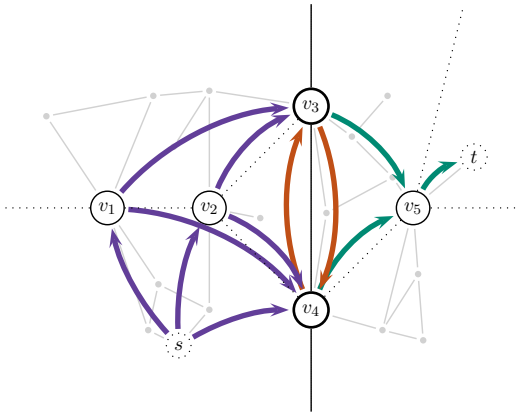

Figure 2: The search graph $\mathcal{M}_{st}$.

with $i \in \{1, \ldots, L-1\}$. Figure 2 shows an example, edges from the sets $\mathscr{L}$, $\mathscr{U}_i$, and $\mathscr{D}_i$ are colored brown, blue, and green, respectively.

Owing to the altered definition of the level edge set compared to [SWZ02], 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 $L > 1$.

Correctness In the following, we shall prove that $\mathcal{M}_{st}$ can be used for answering the 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 distance in G. It remains to prove that for a shortest s-t path in G there is an equally long path in $\mathcal{M}_{st}$.

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.[1] 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.[2] We can split a shortest s-v path into two (possibly empty) subpaths at u. 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. □

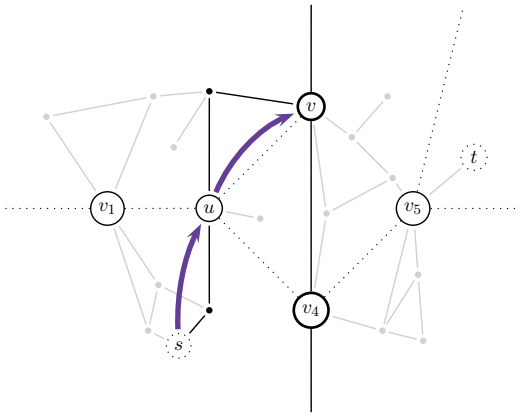

Figure 3: Illustration for Lemma 1.

Theorem 2. *If $L > 1$, the s-t distance is the same 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 or 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. □

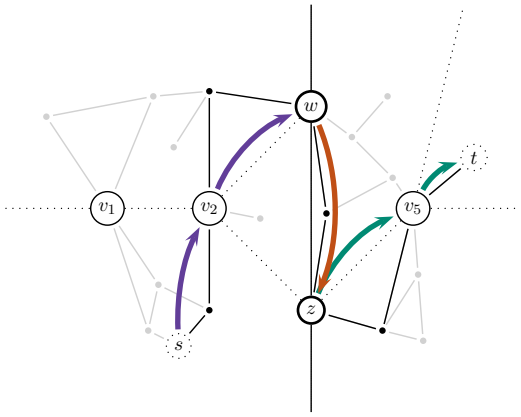

Figure 4: Illustration for Theorem 2.

[1] $Adj(C_0^s) = \{s\}$

[2] This is clearly true for $s \in Adj(C_{i-1}^s)$. For the opposite, the identity $V(C_{i-1}^s) \cap Adj(C_{i-1}^s) = \emptyset$ holds, as $s \notin S_1$ in this case. Hence, there must be a vertex in each s-v path that is in $V(C_{i-1}^s)$ but whose successor is not. This successor must be in $Adj(C_{i-1}^s)$ by definition.

Query The graph $\mathcal{M}_{st}$ can be transformed into an equivalent DAG by creating multiple copies of vertices; refer to [Mül06] for details. For a DAG, an s-t shortest-path query can be performed in $O(V + E)$ time. In addition, the topological structure of our DAG is known in advance and does not need to be computed separately for each query. 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.

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 this case, too.

2.2 Optimization of Partial Graphs

So far, we have described some adaptations to the classical multi-level technique; in what follows we introduce the central modification. Instead of storing the multi-level graph as a whole, we spread it over a great number of *partial graphs*: For each pair of vertices s and t, consider the sets of upward, level, and downward edges determined during construction of $\mathcal{M}_{st}$ and for each such edge set, store the graph induced by it separately. That given, any search graph requested for can be constructed through union of a number of appropriate partial graphs.

After that, each partial graph can be optimized individually using two different techniques: by removing so-called *superseded* edges and by constructing an equivalent graph with more vertices but fewer edges.

Note that, in general, edges from $\mathcal{M}$ occur in more than one partial graph. This increases storage requirements by a factor of about two, which is more than absorbed by the reduction in size after the optimization phase.

Removing superseded edges For an edge set $\mathscr{U}_i$, the *upward part* at level i, let G_i be the wrapped component that contains all the vertices of $\mathscr{U}_i$, and C_{i-1} be the component at level $i-1$ whose adjacent vertices make up the source vertices of each edge in $\mathscr{U}_i$. Consider an edge $(w, v) \in \mathscr{U}_i$. If a shortest w-v path in G passes another vertex $z \in Adj(C_{i-1})$, then any s-v path via w in any search graph that uses $\mathscr{U}_i$ has a path no longer via z. 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 5.

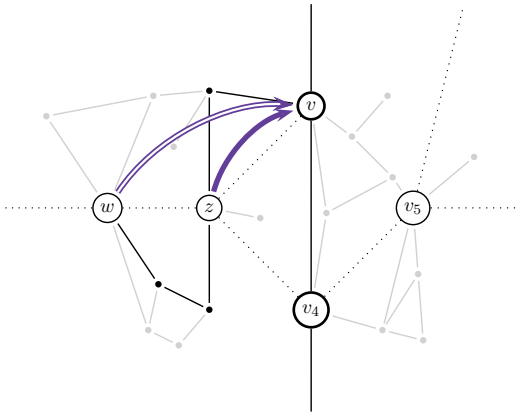

Figure 5: Superseded edges.

To determine if an edge (w, v) is superseded by another edge, we only need to consider distances between adjacent vertices of C_{i-1}, together with edge weights of the upward part $\mathscr{U}_i$:

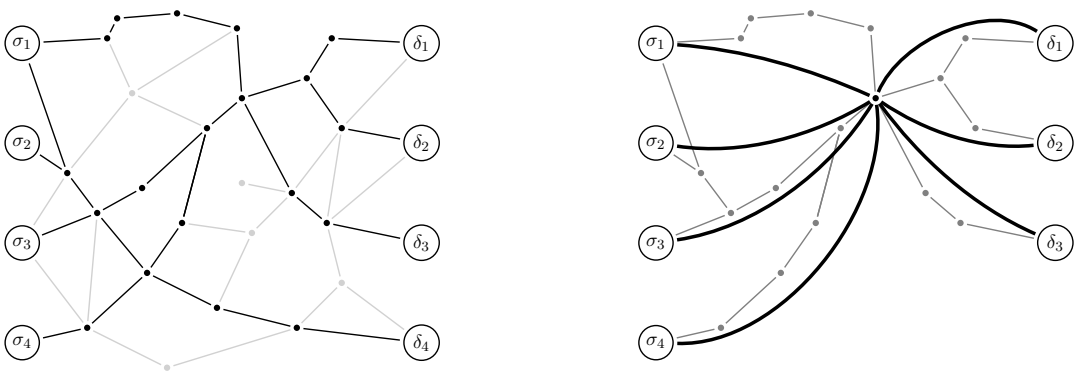

Figure 6: Constructing equivalent graphs.
The drawing at the left shows a directed weighted graph. Highlighted edges are contained in at least one shortest path from any vertex at the left to any vertex at the right. The edges in the corresponding partial graph are made up from the lengths of these shortest paths. In the drawing at the right, we show that a small number of edges (seven instead of twelve) suffices to reconstruct all distances between a subset of σ-δ pairs. This is possible, because the shortest paths between the vertices in question contain a common vertex. Experiments confirm that this is a frequent case for road graphs.

we can do without (w, v) if there is an edge $(z, v) \in \mathscr{U}_i$ with $c(w, v) = d(w, z) + c(z, v)$ and $d(w, z) > 0$. (Here, c denotes the edge weight function in $\mathscr{U}_i$, and d refers to the distance in G.) After removing a superseded edge from the upward part, there may be other superseded edges. The order in which superseded edges are removed is arbitrary, as supersedement is a strict partial order. We can therefore achieve an optimal result by applying a simple greedy scheme.

Analogously, we can eliminate superseded edges in downward and level parts. The optimization is local to the partial graphs: An edge that is superseded in one partial graph may or may not be superseded in another partial graph.

Constructing equivalent graphs A considerable amount of edges of the partial graphs can be saved by introducing auxiliary vertices and replacing many individual edges with few edges through the new vertices. This works well, because edge weights in partial graphs are in fact lengths of shortest paths in G that tend to have common vertices (cf. Figure 6). Of course, the distances may not change by this.

We have implemented a simple heuristic that adds a single, so-called *central* vertex. Then it decides in a greedy manner which of the original vertices to connect to the new central vertex and to balance the weight function for the new edges so that a maximal number of original edges can be removed. In the best case, all underlying shortest paths contain at least one common vertex, and the optimization results in a star-like graph.

In contrast to supersedement, the equivalence optimization of a partial graph does not depend on the original graph G. Therefore, other graph minimization techniques may be applied as well.

3 Experiments

In this section, we present the results of our experimental evaluation. As input we used the road map of Western Europe[3], provided by the PTV AG, Karlsruhe. This graph has approximately 15.4 million vertices and 35.7 million edges. We used four different types of Opteron computers for our experiments, all of them running SUSE Linux 9.3: two dual AMD Opterons 252 clocked at 2.6 GHz with 16 GB and RAM 2 x 1 MB of L1 cache and two dual Opterons 248 clocked at 2.2 GHz with 4 and 8 GB RAM, respectively, and 2 x 1 MB L1 cache. The program was compiled with GCC 3.4, using optimisation level 3 and the LEDA library (version 5.01). The queries were executed on an Opteron 248.

Preprocessing We separated our input graph within 24 hours on all 8 CPUs as follows. Our data contains border vertices between countries. These vertices are selected as separators for the topmost level. Each country was separated by recursively applying the Planar-Separator Theorem [LT79, HPS$^+$05]. Note that with this separation several granularities are possible.

We used a three-level setup with granularities of at most 20, 40, and 80 adjacent vertices per component, respectively. These granularites yield a good tradeoff between preprocessing and query times. For the given separation preprocessing took another 24 hours on all 8 CPUs, generating partial graphs with a total amount of about 246.1 million edges for upward and downward graphs of level 0 and another 285.1 million edges for the remaining upward, level, and downward graphs. For better debugging, we chose XML as output format, leading to a high overhead. A binary format would consume only about 6 GB in total.

Using a 20–40–80 separation, we can guarantee search spaces of less than $80 \cdot 80 + 2 \cdot 80 \cdot 40 + 2 \cdot 40 \cdot 20 + 2 \cdot 20 = 14\,440$ edges. However, we observed an average search space of only about 7 000, far smaller than the given bound, which is mainly due to the optimization of the partial graphs. An evaluation of the sparsification shows that about one quarter of the level graphs can be reduced to a star-like graph. On average, 40% of the edges are removed by supersedement and 31% by equivalence. However, 20% of the edges are removed by both routines, leading to a total removal of about 51%.

Queries We ran 10 000 exponentially distributed random queries, and compared the search spaces achieved with both DIJKSTRA's algorithm and our technique (we chose an exponential distribution over the uniform distribution as the former captures the intuition that in real scenarios, shorter distances are queried more often than longer ones). As for query times, we neglect the I/O process needed to load the XML data for a certain *s*-*t* query into main memory because an implementation keeping all partial graphs in memory seems possible. Furthermore, for exacter time measurements we repeated each query 1 000 times calculating the average.

Search Space Figure 7 shows search space size and speed-up compared to DIJKSTRA's algorithm, plotted against *Dijkstra rank*[4]. Here, the search space is measured by the number of touched edges instead of settled vertices, because our technique is dominated by touched edges. Nevertheless,

[3] consisting of 15 countries: Austria, Belgium, Switzerland, Germany, Denmark, Spain, France, Italy, Luxemburg, Norway, the Netherlands, Portugal, and Sweden.

[4] For an *s*-*t* query, the Dijkstra rank of vertex v is the number of vertices inserted in the priority queue before v.

experiments show that the dependency between rank and touched edges is linear for DIJKSTRA's algorithm.

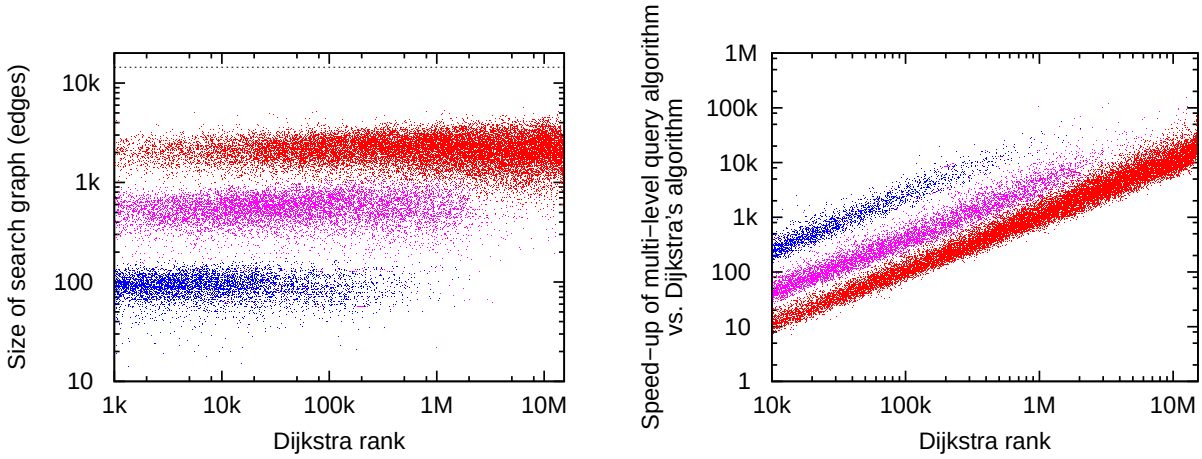

Figure 7: Search space for random queries.

We have three horizontal clouds at approximately 80 (blue), 500 (purple), and 2000 (red) touched edges for the search space. They derive from the fact that the number of upward and downward graphs used for the search graphs varies between one and three. Therefore, we have a search graph constructed from three, five and seven partial graphs. Of course, the number of touched edges highly depends on the number of partial graphs and is nearly independent of the rank. Note that we can only guarantee a search graph with less than 14 440 edges in this setup but observe search graphs of sizes below 5 000 edges. Concerning the search space, we have speedups of up to 20 000 for very high Dijkstra ranks.

Figure 8 shows the relative density and frequency of the number of edges in the search graph. For the density blue represents three, purple five and red seven partial graphs. For the relative

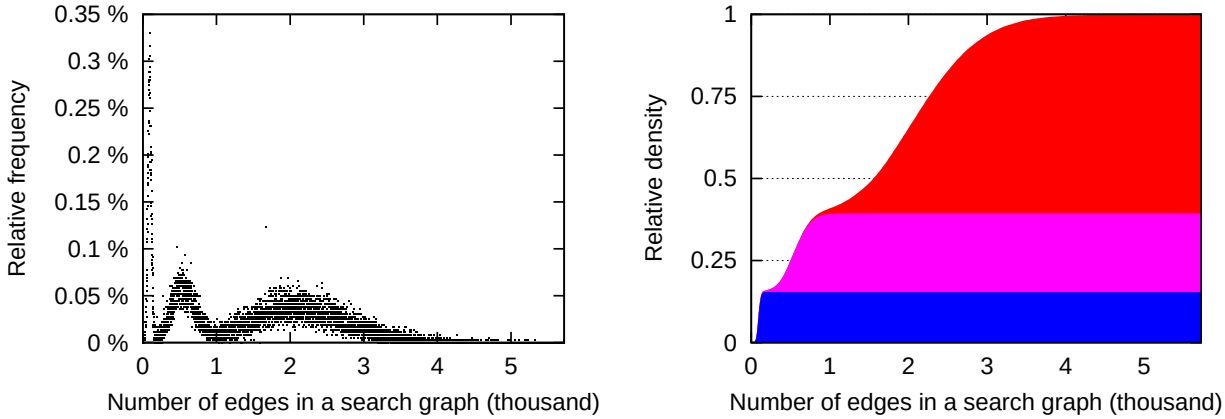

Figure 8: Distribution of search graph sizes for a given set of queries.

frequency we have three peaks: the first at 100, the second at 500 and the third at 2 000 edges.

These three peaks can be found in density as well derive from the number of partial graphs the search graph is constructed of. Summing up, for 50% of our queries we have less than 1 600 edges in the constructed search graph and with a probability of 85% we have less than 3 000 edges.

Query Times Figure 9 shows that the main impact on the query time is the number of partial graphs. For three parts (blue), we have query times below 5 μs, for five parts (purple) around 20 μs

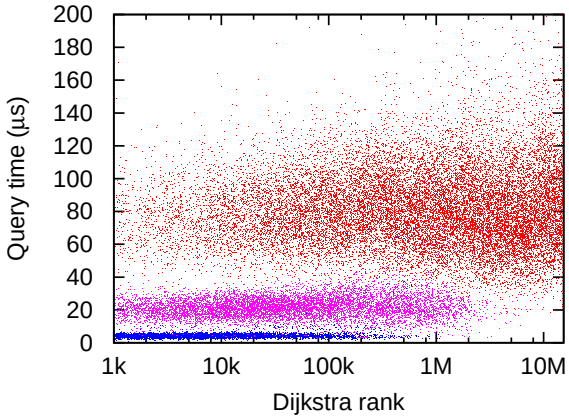

Figure 9: Query run time.

and for seven parts (red) the query time varies between 40 and 200 μs. We re-ran the queries with execution times exceeding 150 μs. We oberved query times below 150μs on the re-run, so these rare high execution times seem to be artefacts.

Figure 10 depicts the relation between size of the search graph and query times obtained using our technique. Again the blue, purple, and red dots represent single queries using three, five, and seven partial graphs, respectively. The figure confirms a nearly linear relation. This is supported

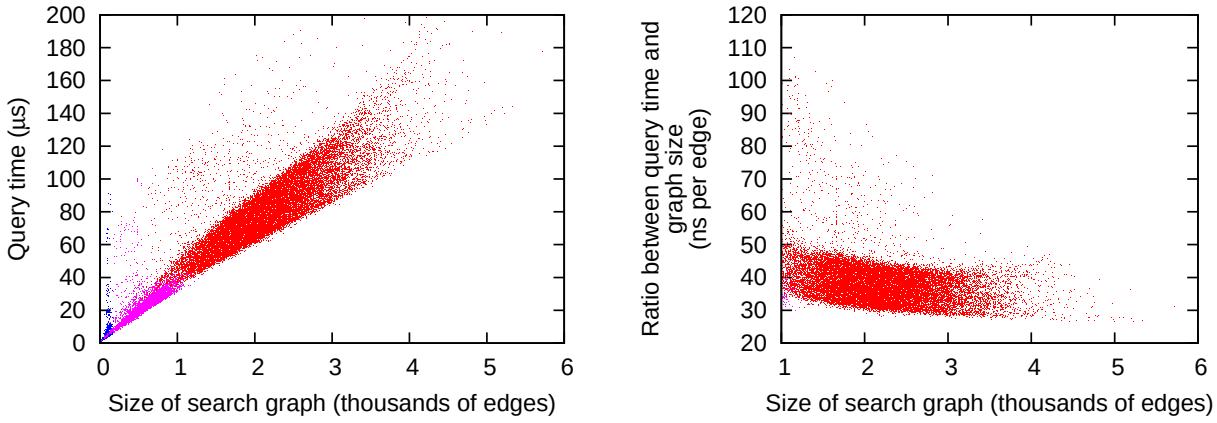

Figure 10: Relation between size of the search graph and query times.

by the almost constant behavior of time per edge with increasing size of the search graph. We plot only search graphs with more than 1 000 edges because for sizes below 1 000 we have some artefacts with high execution times leading that have a negative impact on the y-axis.

Dijkstra rank For a more detailed analysis of the size of the search graph Figure 11 gives an overview of the dependency of search graph and number of partial graphs compared to the Dijkstra rank. For the search graph size we use a box plot. Since we want to plot high ranks we start at a rank of 800 and double it until reaching a rank of 13.1 million.

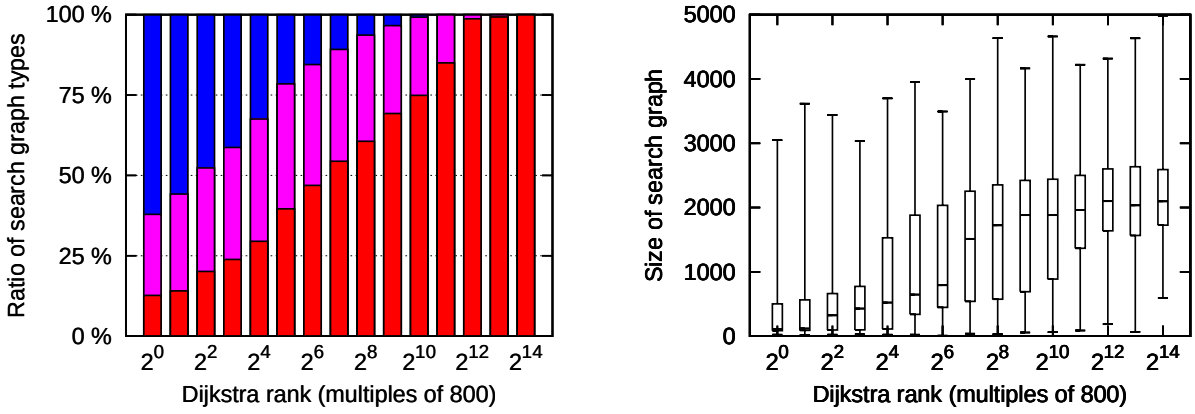

Figure 11: Search graph sizes and distribution of used partial graphs for different Dijkstra ranks.

As expected, the percentage of three partial graphs decreases with increasing rank, while the percentage of seven partial graphs increases. The percentage of five partial graphs stays the same from rank 800 to $800 \cdot 2^{10}$.

In the box plot we observe two significant gaps for the median. It doubles from 1600 to 3200 and from $800 \cdot 2^6$ to $800 \cdot 2^7$ it doubles again. Doublechecking with the left plot, we observe that at the first gap the percentage of three partial graphs drops below 50% while at the second gap the percentage of seven partial graphs exceeds 50%. Between these gaps the median only increases slightly. So the search space seems to more dependent on the number of partial graphs than on the Dijkstra rank.

4 Conclusion

We have shown how to enhance the classical multi-level approach for shortest-path computation [SWZ02] 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 small partial graphs as well as heavy optimization of these graphs. In an experimental study with a road graph, this approach proved extremely useful in a setting where an extensive preprocessing and storage of large amounts of additional data can be afforded: query times are always less than 200 μs and 50 μs on average.

There are various issues left that we consider worth being investigated: For graph minimization, we employed a heuristic that performs well in practice. As the sizes of the partial graphs have an immediate impact on the search space, however, strategies that compress the graphs even better would be helpful. In [HSW04, GKW06], it was shown that combinations of certain speed-up techniques perform better than the individual techniques. In particular, a bidirectional variant of

our algorithm might be beneficial in that the number of level edges needed to be considered can be reduced.

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. To allow for that, our preprocessing could be, similarly to [JHR98], enriched with path information.

References

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[GKW06] Andrew Goldberg, Haim Kaplan, and Renato Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *Proc. Algorithm Engineering and Experiments*, pages 129–143. SIAM, 2006.

[Gut04] Ronald J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proc. Algorithm Engineering and Experiments*, pages 100–111. SIAM, 2004.

[HPS+05] Martin Holzer, Grigorios Prasinos, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Engineering planar separator algorithms. In *Proc. European Symposium on Algorithms*. Springer, 2005.

[HSW04] Martin Holzer, Frank Schulz, and Thomas Willhalm. Combining speed-up techniques for shortest-path computations. In *Proc. Workshop on Experimental and Efficient Algorithms*, volume 3059 of *LNCS*, pages 269–284. Springer, 2004.

[HSW06] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multi-level overlay graphs for shortest-path queries. In *Proc. Algorithm Engineering and Experiments*, pages 156–170. SIAM, 2006.

[JHR98] 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(3):409–432, 1998.

[JP96] Sungwon Jung and Sakti Pramanik. HiTi graph model of topographical roadmaps in navigation systems. In *Proc. Data Engineering*, pages 76–84. IEEE Computer Society, 1996.

[Lau04] Ulrich Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Geoinformation und Mobilität — von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, Institut für Geoinformatik, Münster, 2004.

[LT79] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[MSS+05] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speed up Dijkstra's algorithm. In *Proc. Workshop on Experimental and Efficient Algorithms*, LNCS, pages 189–202. Springer, 2005.

[Mül06] Kirill Müller. Design and implementation of an efficient hierarchical speed-up technique for computation of exact shortest paths in graphs. Master's thesis, Dept. of Informatics, University of Karlsruhe, Germany, June 2006.

[SS05] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *Proc. European Symposium on Algorithms*, volume 3669 of *LNCS*, pages 568–579. Springer, 2005.

[SS06] Peter Sanders and Dominik Schultes. Engineering highway hierarchies. In *Proc. European Symposium on Algorithms*, LNCS. Springer, 2006. To appear.

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using multi-level graphs for timetable information in railway systems. In *Proc. Algorithm Engineering and Experiments*, volume 2409 of *LNCS*, pages 43–59. Springer, 2002.

[WW03] Dorothea Wagner and Thomas Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *Proc. European Symposium on Algorithms*, volume 2832 of *LNCS*, pages 776–787. Springer, 2003.

[WW06] Thomas Willhalm and Dorothea Wagner. Shortest path speedup techniques. In *Algorithmic Methods for Railway Optimization*, LNCS, 2006. To appear.