High-Performance Multi-Level Graphs[*]

Daniel Delling[†] Martin Holzer[†] Kirill Müller[†] Frank Schulz[‡]
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 [1, 2], which makes use of a hierarchical decomposition of the input graph and pre-computation of supplementary information. We develop this preprocessing to the maximum and introduce several ideas to enhance this approach considerably, by reorganizing the pre-computed 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 a road graph containing over 15 million vertices. Our query algorithm computes the distance for any pair of vertices in no more than 70 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 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 [3] 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. Recent preprocessing techniques [4, 5] yield for graphs of sizes similar to ours considerable average query times of around 2 ms and just under 1 ms, respectively, while maintaining optimality of the solution.

In this work, we present a further enhancement of the multi-level technique given in [1], which is based on a hierarchical decomposition of the input graph and computation of an auxiliary

[*]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.

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. We develop the preprocessing to the maximum: Our 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 [1, 2] 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 two given vertices. This search graph is acyclic, and we outline 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. With our 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 70 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 [1], 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 most of which rely on Dijkstra's shortest-path algorithm [3]. 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 [6, 7, 8, 9, 10, 11, 12, 4]; 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 [1, 2, 13, 14, 15, 5].

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 [1, 2] (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 *search 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 [13] 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 [14], 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 [15, 5], 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. In a latest further development [16], this approach has been extended by a concept very similar to ours to take advantage of precomputed distances between all vertices selected at the topmost level(s). Moreover, this approach—*transit node routing*—has been generalized in such a way that the basic concept behind our high-performance multi-level technique is covered as well. The major differences include the fact that we use graphs rather than matrices to represent the all-pairs distances, as the former naturally induce means for optimization. Also, transit note routing is designed to deal with travel time as edge weights more efficiently than with route lengths, which is not true for our technique.

2 Multi-Level Graphs

The formal description of our high-performance multi-level graphs is divided into two parts. The first reviews the classical multi-level technique as well as points out several enhancements to it, where the structure in general follows that of [1, Sect. 2]; modifications and additional concepts applied are highlighted explicitly. In Section 2.2, we then 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. In addition to the classical variant, the search graph can now be transformed into an acyclic graph, which permits computing distances 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: $V \supset S_1 \supset S_2 \supset \cdots \supset S_l$. 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 $|V|$. Such separator sets can be obtained, for instance, through repeated application of the Planar-Separator Theorem [17, 18]. 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 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 $S_0 = V$. (This is a slight difference to [1].)

- 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 $Adj(C)$. For $v \in S_i$ (i.e., $C_i^v = \{v\}$), we define $Adj(C_i^v) := \{v\}$.
- We now introduce a new term: A component C_i^v together with its adjacent vertices is called the *wrapped component* $G_i^v = G \cap (V(C_i^v) \cup Adj(C_i^v))$.

Multi-Level Graph Each level of the multi-level graph $\mathcal{M}$ is determined by a set of edges. For each $i \in \{1, \ldots, l\}$, 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 than in [1], where they are 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.

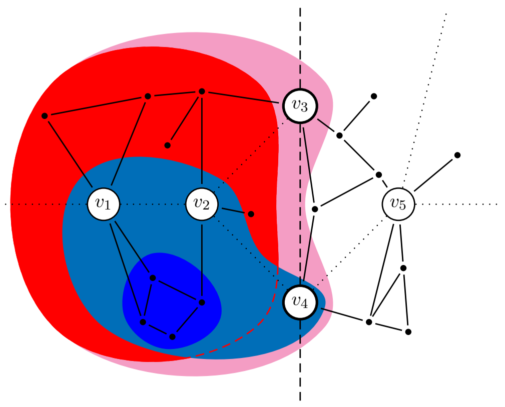


Figure 1: Hierarchy.
Components (dark colors) and belonging wrapped components (light colors) at level 1 (blue) and 2 (red), respectively. Note that vertex v_4 is a separator vertex of both level 1 and 2.

Construction For a level $i \in \{1, \ldots, l\}$, 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 set to 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 $i \in \{1, \ldots, l-1\}$ 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 [1], 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 performed recursively 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 in order 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 work at hand explores the fact that 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 [19, Sect. 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 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 $(C_0^s, C_1^s, \dots, C_L^s = C_L^t, \dots, C_1^t, C_0^t)$. 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 $i \in \{1, \dots, L-1\}$.

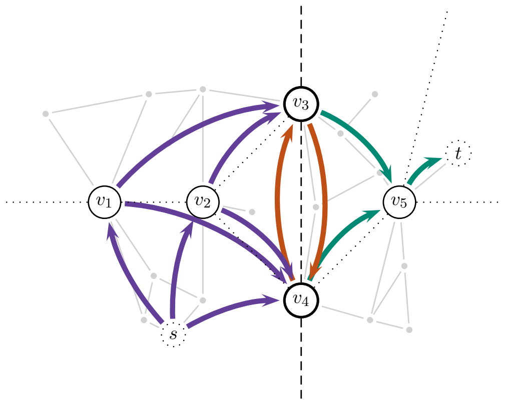


Figure 2: The search graph $\mathcal{M}_{st}$.

Figure 2 shows an example, where 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 [1], 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 very modifications require a new proof of correctness.

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 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 $i \in \{0, \dots, L-1\}$, 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 $Adj(C^s_{i-1})$, let u be the first such vertex. We can split a shortest s-v path into two (possibly empty) subpaths at u. The s-u subpath contains only vertices from G^s_{i-1} 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. $\square$

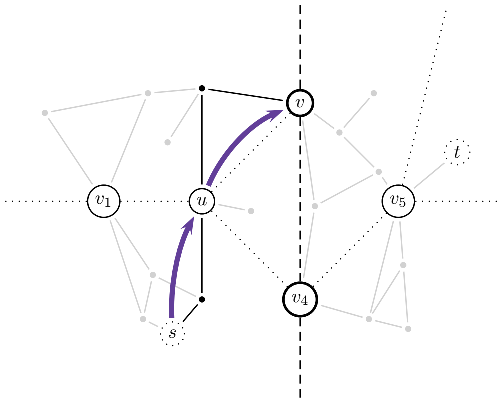


Figure 3: Illustration for Lemma 1.

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^s_0 and C^t_0 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 $w \in Adj(C^s_{L-1})$ and $z \in Adj(C^t_{L-1})$ 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^s_{L-1} and G^t_{L-1}, 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. $\square$

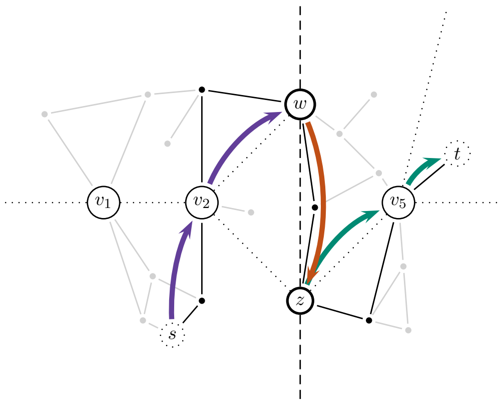


Figure 4: Illustration for Theorem 2.

Query The graph $\mathcal{M}_{st}$ can be transformed into an equivalent DAG by creating at most L copies of each vertex; refer to [19] 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 $C = C^s_1 = C^t_1$, 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 Optimizing 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. Thus, the partial graphs in total require about twice as much storage as $\mathcal{M}$, 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 $z \in \mathit{Adj}(C_{i-1})$, then any s-v path via w in any search graph that uses $\mathscr{U}_i$ has a path via z that is no longer than the path via w. 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.

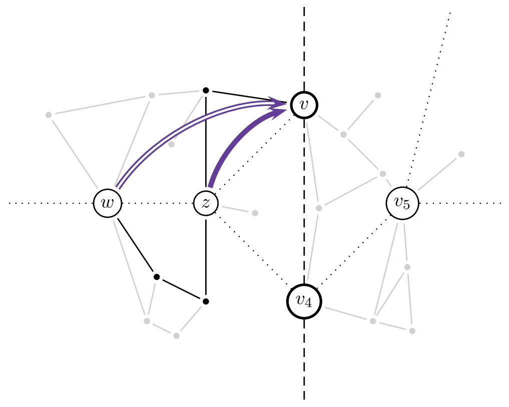


Figure 5: Superseded edges.

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 $\mathscr{U}_i$: 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. All partial graphs are optimized seperately: an edge that is superseded in one partial graph may or may not be superseded in another partial graph. For details, cf. [19].

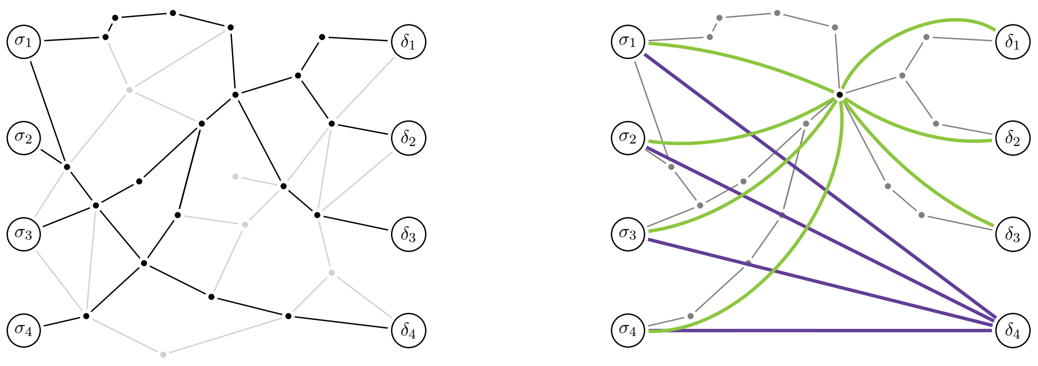


Figure 6: Constructing equivalent graphs.

Constructing equivalent graphs Another 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. An example is given in Figure 6. 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 the optimized graph is 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. This results in a partial graph with eleven instead of 16 edges.

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 balances 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.

3 Experiments

In this section, we present the results of our experimental evaluation. As input we used the road map of Western Europe[1,2], provided by the PTV AG, Karlsruhe. This graph has approximately 15.4 million vertices and 35.7 million edges, where edge lengths correspond to route lengths. We used four different machines for our experiments, all of them running SUSE Linux 9.3: two dual AMD Opterons 252, clocked at 2.6 GHz with 16 GB of RAM and 2 x 1 MB of L2 cache, and two dual Opterons 248, clocked at 2.2 GHz with 4 and 8 GB of RAM, respectively, and 2 x 1 MB of L2 cache. The program was compiled with GCC 3.4, using optimization level 3 and the LEDA library (version 5.01). The queries were executed on an Opteron 248.

Preprocessing We separated our input graph as follows. To obtain an initial separation, we selected those vertices from our dataset that are marked as border vertices. Each country was then separated by recursively applying the Planar-Separator Theorem [17, 18]. In total, the computation of this separation took 24 hours using all 8 CPUs. Note that different methods for separation may be applied as well, for a systematic analysis see [2]. There it was shown that Planar-Separator works well for multi-level techniques.

We used a three-level setup with granularities of 20, 40, and 80 for the maximum number of adjacent vertices per component, respectively. These granularities yield a reasonable tradeoff between preprocessing and query times. For the given separation, preprocessing took another 24 hours employing all 8 CPUs, generating partial graphs with a total amount of about 246.1 million edges for upward and downward graphs of level 1 and another 285.1 million edges for the remaining upward, level, and downward graphs. For better debugging, we chose XML as intermediate and 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 a maximum search space of only about 6 000, 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 fully reduced to

[1]consisting of 13 countries: Austria, Belgium, Switzerland, Germany, Denmark, Spain, France, Italy, Luxemburg, Norway, the Netherlands, Portugal, and Sweden.

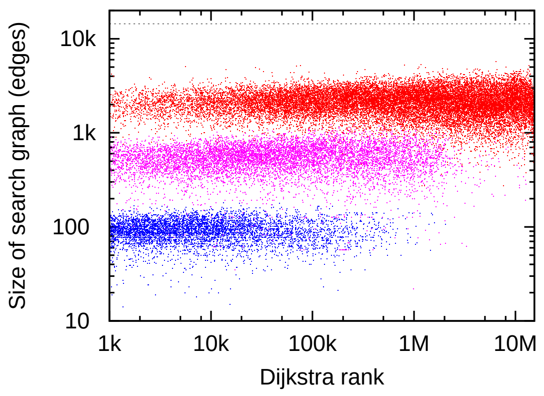
[2]The main focus of this investigation was realizability of employing our multi-level technique to answer shortest-path queries with graphs of several million vertices. Therefore and due to the large experimental effort—especially regarding the preprocessing—we settled for this graph.

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%.

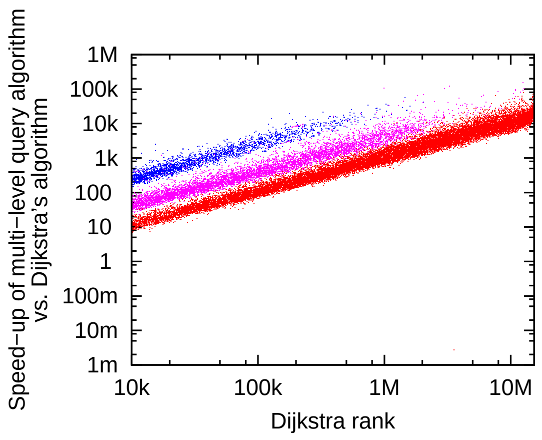
Queries We ran 40 000 random queries exponentially distributed by *Dijkstra rank*[3], and compared the search spaces achieved with both Dijkstra's algorithm and our technique (we chose an exponential distribution over the uniform distribution as the former captures the intuition that in real scenarios, shorter distances are queried more often than longer ones).

In our setup, 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. Beforehand, the query algorithm loads the partial graphs from external memory and flushes the L2 cache by performing copy operations on two buffers 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 our set of queries in the same order and used the median of the three measurements.

Search Space Figures 7(a) and 7(b) show search space size and speed-up compared to Dijkstra's algorithm, plotted against the Dijkstra rank. 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, experiments show that the dependency between rank and touched edges is linear for Dijkstra's algorithm. We have three horizontal clouds at approximately 100



(a) search space



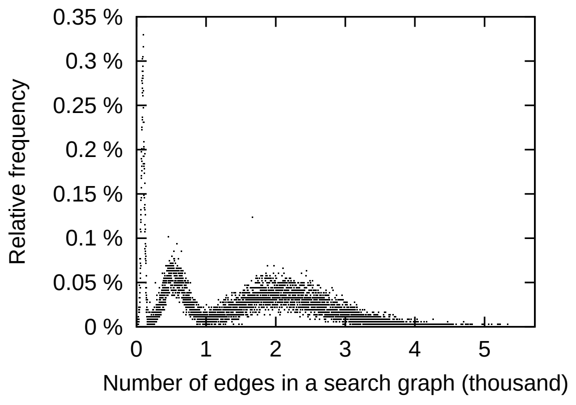
(b) speed-up

Figure 7: Search space for random queries.

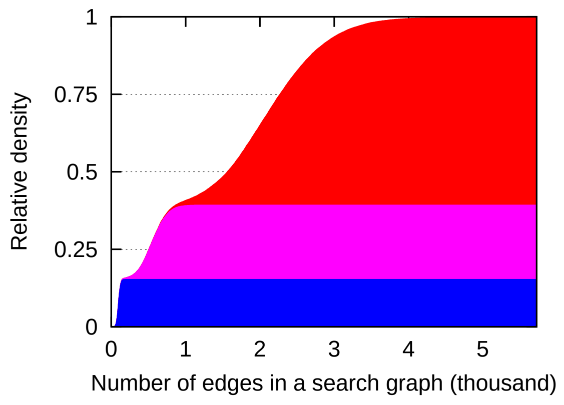
(blue), 500 (purple), and 2 000 (red) touched edges for the search space. They derive from the fact that the number of upward and downward graphs used for the search graphs varies between 1 and 3. Therefore, we have a search graph constructed from three, five, and seven partial graphs. The number of 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 6 000 edges.[4] Concerning the search space, we have speed-ups of up to 20 000 for very high Dijkstra ranks.

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

[4]A rigorous analysis of the preprocessed data proves a maximum search graph size of 7 223, cf. [19].



(a) relative frequency

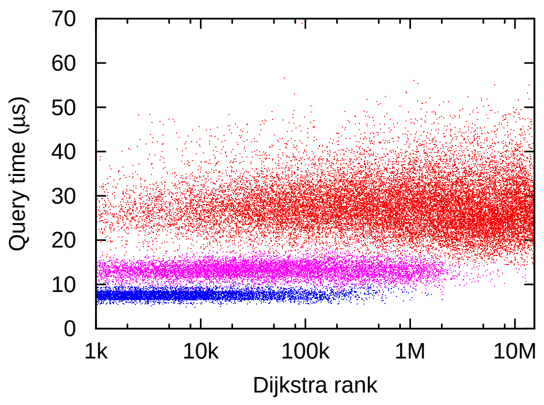


(b) relative density

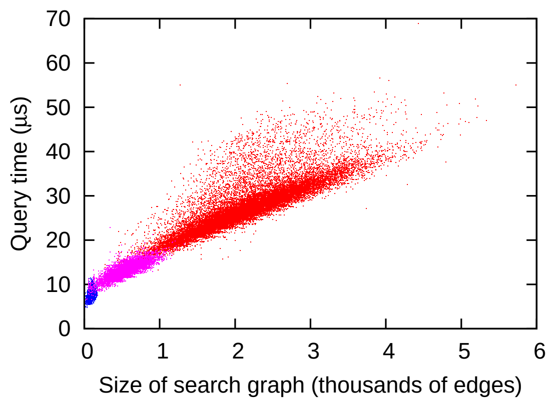
Figure 8: Relative frequency and density for random queries.

Figure 8 shows the relative density and frequency of the number of edges in the search graph. For the relative frequency three peaks arise: the first at 100, the second at 500, and the third at 2 000 edges. These three peaks 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 Figures 9(a) and 9(b) confirms that the main impact on the query time is the number of partial graphs. For three parts (blue), we have query times below 11 μs, for five parts (purple) between 7 and 23 μs, and for seven parts (red) the query time varies between 12 and 70 μs.[5] Note, too, that query time never falls below 5 μs even for the smallest search graphs. For our set of queries, we observe an average query time of 22 μs.



(a) Dijkstra rank



(b) search space

Figure 9: Query times for random queries.

Figure 10 depicts the relation between the sizes of the search graphs and the query times obtained using our technique. Again each blue, purple, and red dot represents one single query using three, five, and seven partial graphs, respectively. The figure confirms a nearly linear

[5]There is only one query that took more than 60 μs in all three runs. However, there are queries that require larger search graphs to be analyzed. The reason might be that the memory allocator has to perform some extra work for this query: we measure the allocation time for the array of distance labels, too, and this seems to be the only component in our query algorithm that is able to cause this behavior.

relation. This is supported 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 observe some artifacts with high per-edge execution times that have an adverse impact on the range of the y-axis.

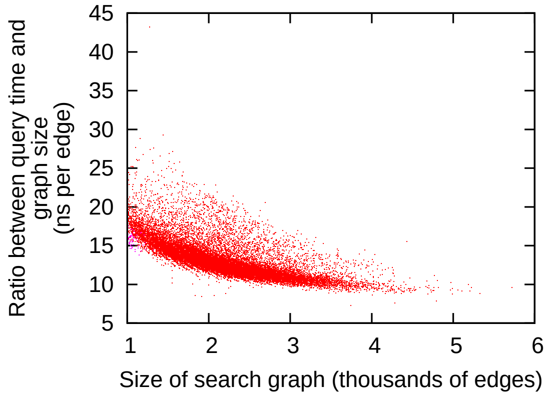


Figure 10: Relation between size of the search graph and query times.

Dijkstra rank For a more detailed analysis of the size of the search graph, Figure 11 gives an overview of the dependency of number of partial graphs and search graph size compared to the Dijkstra rank. For the search graph size we use a box plot showing the five quartiles. 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. For each rank given, we ran 1 000 queries.

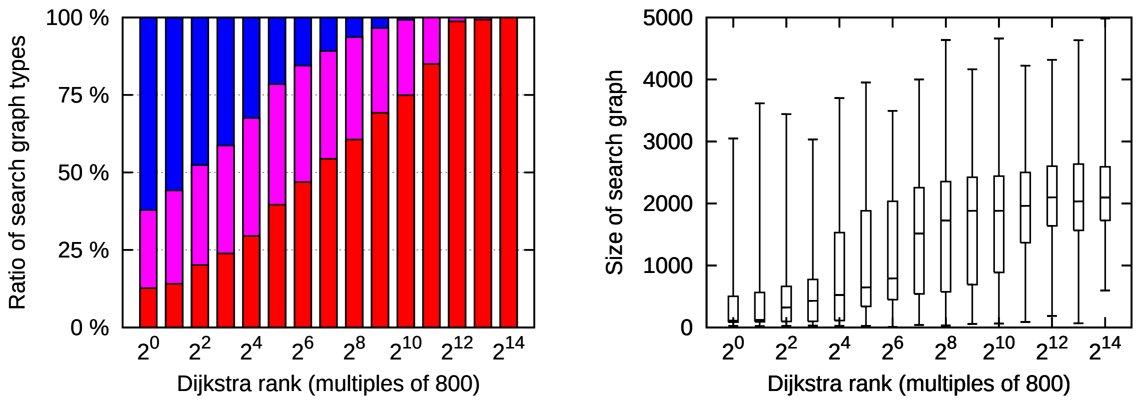


Figure 11: Search graph sizes and distribution of number of partial graphs used for different Dijkstra ranks.

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

We observe two significant jumps for the median. It doubles from $800 \cdot 2$ to $800 \cdot 2^2$ and from $800 \cdot 2^6$ to $800 \cdot 2^7$. Doublechecking with the left plot, we observe that at the first jump the percentage of three partial graphs drops below 50% while at the second jump the percentage of seven partial graphs exceeds 50%. Between these jumps the median increases only slightly. So, it is confirmed again that the search space is 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 [1] 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. Note that distance matrices instead of partial graphs could be used for maintaining the distances between separators. The advantage of using partial graphs is that optimization directly leads to a sparsification of the partial graphs and thus to a reduction of data.

In an experimental study with a road graph, our 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 70 μs and 22 μs on average. For experiments on different granularities we would like to refer the reader to [19]. We want to point out that since the publication of the latter work we have been able to improve query runtime considerably by optimizing the representation of the partial graphs and the query algorithm.

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 [20, 4], it was shown that combinations of certain speed-up techniques perform better than the individual techniques. In particular, a bidirectional variant of our query algorithm might be beneficial in that the number of touched level edges 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 [14], enriched with path information.

References

[1] Schulz, F., Wagner, D., Zaroliagis, C.: Using multi-level graphs for timetable information in railway systems. In: Proc. Algorithm Engineering and Experiments. Volume 2409 of LNCS., Springer (2002) 43–59

[2] Holzer, M., Schulz, F., Wagner, D.: Engineering multi-level overlay graphs for shortest-path queries. In: Proc. Algorithm Engineering and Experiments, SIAM (2006) 156–170

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

[4] Goldberg, A., Kaplan, H., Werneck, R.: Reach for A*: Efficient point-to-point shortest path algorithms. In: Proc. Algorithm Engineering and Experiments, SIAM (2006) 129–143

[5] Sanders, P., Schultes, D.: Engineering highway hierarchies. In: Proc. 14th European Symposium on Algorithms. Volume 4168 of LNCS., Springer (2006) 804–816

[6] Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: An empirical case study from public railroad transport. In: Proc. Workshop on Algorithm Engineering, Springer (1999) 110–123 Also published in: ACM Journal of Experimental Algorithms **5** (2000).

[7] Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In: Proc. European Symposium on Algorithms. Volume 2832 of LNCS., Springer (2003) 776–787

[8] Wagner, D., Willhalm, T., Zaroliagis, C.: Geometric containers for efficient shortest-path computation. ACM Journal of Experimental Algorithmics **10** (2005) 1–30

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

[10] Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In: Geoinformation und Mobilität — von der Forschung zur praktischen Anwendung. Volume 22., IfGI prints, Institut für Geoinformatik, Münster (2004) 219–230

[11] Möhring, R.H., Schilling, H., Schütz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speed up Dijkstra's algorithm. In: Proc. Workshop on Experimental and Efficient Algorithms. Volume 3503 of LNCS., Springer (2005) 189–202

[12] Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. In: Proc. Workshop on Experimental and Efficient Algorithms, Springer (2005) 126–138

[13] Jung, S., Pramanik, S.: HiTi graph model of topographical roadmaps in navigation systems. In: Proc. Data Engineering, IEEE Computer Society (1996) 76–84

[14] Jing, N., Huang, Y.W., Rundensteiner, E.A.: Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. IEEE Trans. Knowledge and Data Engineering **10** (1998) 409–432

[15] Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: Proc. European Symposium on Algorithms. Volume 3669 of LNCS., Springer (2005) 568–579

[16] Bast, H., Matijevic, D., Funke, S., Sanders, P., Schultes, D.: In transit to constant time shortest-path queries in road networks. Submitted to ALENEX07 (2006)

[17] Lipton, R.J., Tarjan, R.E.: A separator theorem for planar graphs. SIAM Journal on Applied Mathematics **36** (1979) 177–189

[18] Holzer, M., Prasinos, G., Schulz, F., Wagner, D., Zaroliagis, C.: Engineering planar separator algorithms. In: Proc. European Symposium on Algorithms. Volume 3669 of LNCS., Springer (2005) 628–639

[19] Müller, K.: 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 (2006) `http://i11www.iti.uka.de/teaching/theses/files/da-kmueller-06.pdf`.

[20] Holzer, M., Schulz, F., Willhalm, T.: Combining speed-up techniques for shortest-path computations. In: Proc. Workshop on Experimental and Efficient Algorithms. Volume 3059 of LNCS., Springer (2004) 269–284