Design and Implementation of an Efficient Hierarchical Speed-up Technique for Computation of Exact Shortest Paths in Graphs

Diploma thesis
at the
Institute for Theoretical Informatics
University of Karlsruhe (TH)

by

Kirill Müller

Supervised by:
Prof. Dorothea Wagner
Dr.rer.nat. Frank Schulz
Dipl.-Math. Martin Holzer
Dipl.-Inf. Daniel Delling

Acknowledgements

I would like to thank my supervisors, Prof. Dorothea Wagner, Dr. Frank Schulz, Martin Holzer und Daniel Delling, for the extensive support with the development of this work and for many hints, motivating discussions and ideas.

Contents

1 Introduction

Computation of shortest paths in graphs is a central requirement for many applications. Route planning in traffic networks is perhaps the best-known one. DIJKSTRA's algorithm [Dij59] solves this problem efficiently, but query time may be intolerably long for large graphs, such as the road map graph of a country or even a continent. In practice, optimality of the result often is abandoned by employing a heuristics. In this work, we focus only on exact solutions to the shortest-path problem.

We cannot afford precomputing shortest paths between all pairs of vertices in terms of neither computation time nor space. However, a graph can be preprocessed in an off-line stage so that subsequent on-line queries take only a fraction of the time used by DIJKSTRA's algorithm. Recent preprocessing techniques [SS05, GH05] yield a considerable speed-up for the query time while maintaining optimality of the solution.

In this work, preprocessing is developed to the maximum. We introduce a multi-level technique based on hierarchical decomposition that outsources almost all of the effort needed to compute a shortest path to the off-line preprocessing stage. A long-lasting preprocessing and a fair amount of preprocessed data is accepted. We parallelize the preprocessing to perform it in reasonable time on conventional hardware. Our technique fits best into an environment where query time is invaluable but long preprocessing times can be afforded, such as a car navigation system or a web-based route planner.

Unlike many other preprocessing techniques, we do not precisely enrich the graph with detailed information that allows a modified version of DIJKSTRA's algorithm to selectively skip edges. Instead, we compute a large number of small graphs to be interconnected afterwards. For each possible query, we are able to combine a *search space graph* in which the distance between two dedicated vertices matches the length of the shortest path queried for. A query is answered by determining the distance in the search space graph, for which we derive a fast algorithm in this work.

The trade-off between preprocessing effort and query time is adjustable. For fixed parameters, we can provide a guarantee for the query time before even starting the preprocessing, and tightly refine this guarantee by analyzing the preprocessed data. We can also state the average number of edges considered by a query for a preprocessed graph. 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 entirely fits into main memory, the query performance of our technique is by all means competitive to other recent approaches.

The remainder of this work is structured as follows. Section 2 offers an overview of other speed-up techniques for source-target shortest-path queries. In Section 3, we show the basic idea of our preprocessing technique and prove its correctness. Section 4 describes two options for optimization that reduce both the size of the preprocessed data and the query time. In Section 5, we give a detailed description of the preprocessing algorithm, including the parallelization scheme. After that, we briefly describe our implementation that was used for the empirical study in Section 7. The final section summarizes our work and proposes further enhancements.

2 Related Work

In this section, we briefly present other preprocessing approaches and highlight common aspects and differences to our approach.

HEPV An approach very similar to ours, called HEPV (Hierarchically Encoded Path Views) [JHR98], has been presented by Ning Jing, Yun-Wu Huang and Elke Rundensteiner. Here, too, a hierarchical decomposition has been used to conquer the computational complexity of the preprocessing. Furthermore, this approach also constructs a dedicated search space graph for a given query.

In this paper, also the computation of the course of a shortest path and the update of the preprocessed data upon edge modification has been considered in detail. Both aspects remain out of scope for this work. However, due to the similarity between our work and the HEPV approach, we are confident that we would be able to apply many, if not all, of the missing concepts to our work. On the other hand, optimization of the preprocessed data and parallelization are not considered by the HEPV approach.

Multi-level approach for timetables The multi-level approach presented by Frank Schulz, Dorothea Wagner and Christos Zaroliagis in [SWZ02] performs a preprocessing for timetable graphs by means of a hierarchical decomposition. We use many concepts, including the component tree and the computation of the components that must be visited, in a similar fashion. Unlike ours, this preprocessing approach employs a modification of Dijkstra's algorithm. Thus, the preprocessed data essentially consists of an enriched input graph. Nevertheless, our speed-up technique was greatly influenced by this multi-level approach. For instance, the upward, downward and level edges from [SWZ02] correspond to the upward, downward or level graphs from our technique.

For our work, we used the implementation presented in [HSW06] to obtain a hierarchical decomposition. As we considered only road map graphs, the technique denoted by "Planar-Separator criterion" in above work seemed to be a good choice, and turned out to perform well for our test instances.

Briefly, a decomposition is obtained by repeated application of the Planar Separator theorem [LT79, LT80]. Special care must be taken for road map graphs that are not entirely planar. For details, we refer to [HPS+05].

HiTi graphs The *Hierarchical multi graph model* introduced by Sungwon Jung and Sakti Pramanik [JP96] is another multi-level approach. The most eye-catching difference is, that the decomposition is based on edges rather than on vertices. Also, the query algorithm is a modification of Dijkstra's algorithm that prunes the search space by skipping edges.

Highway Hierarchies For this recent technique, due to Peter Sanders and Dominik Schultes [SS05], the input graph is recursively searched for "important edges", called highway edges. The edges are also arranged in a hierarchy. However, no hierarchical decompo-

sition of the input graph is used. The query employs a modification of the bi-directional version of DIJKSTRA's algorithm that advances the highway hierarchy both from the start and the end vertex until the search frontiers eventually meet.

Reach for A* Recently, Andrew Goldberg and Chris Harrelson have implemented another preprocessing technique [GH05] that combines the well-known A* approach with reach-based routing proposed by [Gut04]. This technique also uses a variant of DIJKSTRA's bi-directional algorithm to perform the query.

3 Formal Description and Correctness

This section provides a basic description of our acceleration technique. We show what kind of preprocessed information we use, and how we execute a source-target query provided the data is already available. For this work, we are only interested in the length of a shortest path.

After defining some symbols, we describe our notion of a hierarchical decomposition. This description is used for the definition of our preprocessed information. Next, we show how this data can be used to carry out a query, and prove the correctness of the query algorithm.

3.1 Definitions

A *weighted graph* $G(V, c)$ consists of a vertex set V and a nonnegative cost function $c : V \times V \to \mathbb{R}_+^\infty$. For all $v \in V$, $c(v, v) = 0$. We omit the commonly used edge set. Instead, by the term "edge" we denote a vertex pair with finite cost. Missing edges feature an infinite cost for the associated pair of vertices. (The "number of edges" refers to the count of vertex pairs with finite cost. Accordingly, "removing an edge" means setting the according value for the cost function to infinity.) We write $V[G]$ and $c[G]$ to denote, respectively the vertex set and the cost function of a specific graph G.

A graph induced by a road map is simply called *road map graph*.

A *path* $p = \langle v_1, v_2, \ldots, v_z \rangle$ is a sequence of vertices in V with $c(v_x, v_{x+1}) < \infty$ for all x with $1 \leq x < z$. Paths have a *length* $c(p)$ that is obtained by summing up the costs of all adjacent vertex pairs. For a given path, the *subpath* from index x to index y with $1 \leq x \leq y \leq z$ is denoted by $p_{x \to y} := \langle v_x, v_{x+1}, \ldots, v_y \rangle$.

We call a graph *connected* iff for every pair of vertices there is a path in the graph's undirected version $G(V, \min(c, c^T))$. We do not care about strong connectivity.

Every graph G has an associated *distance function* $d[G] : V \times V \to \mathbb{R}_+^\infty$ that returns the length of a shortest path between two vertices (or ∞ where no such path exists). We may omit the index if it can be deduced from the context. Obviously, the triangle inequality always holds for a graph's distance function: A detour never decreases the cost of a path.

The *graph union* $G_1(V_1, c_1) \cup G_2(V_2, c_2) =: G(V, c)$ is obtained by setting $V := V_1 \cup V_2$ and $c := \min(c_1, c_2)$. This operation is expedient especially if $V_1 \cap V_2 \neq \emptyset$: Common vertices are merged in the union graph.

The *directed subset closure* $G(V, c)|_{V_1 \to V_2}$ of a graph G for vertex subsets $V_1, V_2 \subseteq V$ is a graph that has only vertices from $V_1 \cup V_2$ and edges from vertices in V_1 to vertices in V_2 with costs equal to the distance in the original graph: $c\big[G|_{V_1 \to V_2}\big] := d[G]|_{V_1 \times V_2}$. The *(simple) subset closure* for $V_1 \subseteq V$ is a special case of the directed subset closure: $G(V, c)|_{V_1} := G(V, c)|_{V_1 \to V_1}$. Any directed subset closure $G(V, c)|_{V_1 \to V_2}$ can be computed in $|V_1|$ iterations of DIJKSTRA's single-source algorithm.

Throughout this work, the term *input graph* denotes the graph we want to query the distances for. We use n to denote the *number of levels* for the hierarchical decomposition of the input graph.

3.2 Decomposition

The hierarchical decomposition of a given input graph $G(V, c)$ is a cornerstone of our speed-up technique. Owing to that, we define it precisely and also prove rather obvious implications. We shall focus on decomposition first and introduce the hierarchy in the next subsection. (To improve readability, the proofs for this and the next subsection are given in Appendix A).

Definition 1 (Decomposition). The decomposition induces a nonempty *index set* I and splits $G(V, c)$ into $|I|$ components $G_i(V_i, c_i)$ for $i \in I$ so that the following properties all hold:

1. Only vertices of the input graph are used:
$$V_i \subseteq V.$$
2. Each component has at least two vertices:
$$|V_i| \geq 2.$$
3. The components' cost functions is obtained by restricting the input graph's cost function to the component vertices:
$$c_i = c|_{V_i \times V_i}.$$
4. Every vertex and every edge is contained in at least one component:
$$\bigcup_{i \in I} G_i = G.$$
5. All $G_i(V_i, c_i)$ are connected (in their undirected version).
6. No edges exist between distinct components:
$$c(v_1, v_2) < \infty \Rightarrow \exists i : v_1, v_2 \in V_i.$$

We define the *separator set* that contains all vertices that exist in more than one component:

$$S := \bigcup_{i,j \in I, i \neq j} V_i \cap V_j.$$

Apparently, we desire a decomposition that yields a small separator set.

Figure 1 shows a graph and a possible decomposition. We will use this sample graph throughout this work and highlight different aspects of interest.

Although our notion of a component that also contains separators can be found in several related works (e.g., [Fre87], [JHR98]), others use a notion where components are induced by removing the separator set from the set of vertices ([SWZ02, Hol03, HPS+05]). With the alternative notion, separator vertices do not belong to any component. Both notions actually describe the same thing, but in a slightly different way. The following lemma shows the equivalence of the two notions and serves as base for several other lemmas:

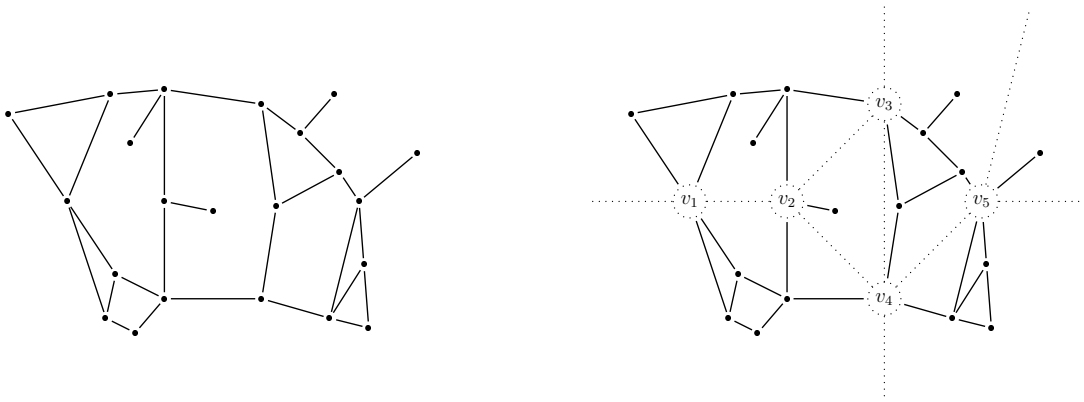

Figure 1: A graph that has been decomposed.
The drawing at the left shows a graph. For the sake of clarity, we omit costs and directions for edges throughout this work.
The right-hand side drawing shows a possible decomposition for this graph. Dotted lines show component boundaries. The vertices that lie on a component boundary (denoted by v_1 to v_5) belong to all adjacent components — thus, they are separators.

Lemma 2. *For any pair of component indices $i, j \in I$, $i \neq j$, and for any pair of vertices $s \in V_i$, $t \in V_j$, every path in G between s and t contains at least one vertex in S.*

We use B_i to denote the set of *boundary vertices* around component G_i, i.e., those vertices in that component that are separators:

$$B_i := V_i \cap S.$$

A component can be left only through one of its boundary vertices. The first separator encountered on any path that leaves the component is such a boundary vertex. This is shown in the next lemma:

Lemma 3. *For any pair of component indices $i, j \in I$, $i \neq j$, and for any pair of vertices $s \in V_i$, $t \in V_j$, every path in G between s and t contains at least one boundary vertex of each G_i and G_j, that is, at least one vertex of B_i and at least one vertex of B_j.*

A decomposition into only one component (i.e., $|I| = 1$) is called the *empty decomposition*. For a given graph, the empty decomposition is unique apart from the choice of the the index set's only element. The following corollary shows that only the empty decomposition does not feature any separators:

Corollary 4. *The separator set S is empty iff $|I| = 1$.*

The next corollary deals with paths without separators except for the end vertex which may or may not be a separator: Such a path does not leave the component where it origins.

Corollary 5. *For any component index $i \in I$ and for any two vertices $s \in V_i \setminus S, t \in V_i$, any path $p = \langle v_1, v_2, \ldots, v_z \rangle$ in G that, apart from t, does not contain vertices in S, contains only vertices in V_i.*

For the sake of completeness, we provide a mirrored version of above corollary:

Corollary 6. *For any component index $i \in I$ and for any two vertices $s \in V_i, t \in V_i \setminus S$, any path $p = \langle v_1, v_2, \ldots, v_z \rangle$ in G that, apart from s, does not contain vertices in S, contains only vertices in V_i.*

For a given separation, we choose a canonical *home component function* $h : V \to I$, that indicates the index of a component where a vertex is located:

$$\forall v \in V : v \in G_{h(v)}.$$

This is a precise definition only for the empty separation. For any separation into more than one component, we can choose, for separator vertices, between more than one component index as value for this function. We fix one such function for each separation and use it consistently.

3.3 Component Hierarchy

In this subsection, we define the component hierarchy: One component is contained by some parent (unless it is a top-level component) and contains several children itself (unless it is a bottom-level component). As usual, containment means that all edges and vertices of a child component are contained in the parent component, too. For the definition, we use multiple decompositions of the same input graph and impose restrictions on the decompositions.

Definition 7 (Component hierarchy). A sequence of decompositions

$$\left\{\{G_i^0 : i \in I^0\}, \{G_i^1 : i \in I^1\}, \ldots, \{G_i^{n-1} : i \in I^{n-1}\}, \{G_i^n : i \in I^n\}\right\}$$

forms a *component hierarchy of depth* n, denoted by $\left\{G_i^k\right\}$, iff the following properties all hold:

1. The decomposition at level n is an empty decomposition with 0 as the only index:

$$I^n := \{0\}.$$

2. For each $k \in \{1, \ldots, n\}$ and for each $i \in I^k$ there exist *relationship sets* $H_i^k \subseteq I^{k-1}$ with the following properties:

   (a) At each level, each component is assigned to at most one relationship set:

$$\forall i, j \in I^k, i \neq j : H_i^k \cap H_j^k = \emptyset.$$

   (b) At each level, each component is assigned to at least one relationship set:

$$\bigcup_{i \in I^k} H_i^k = I^{k-1}.$$

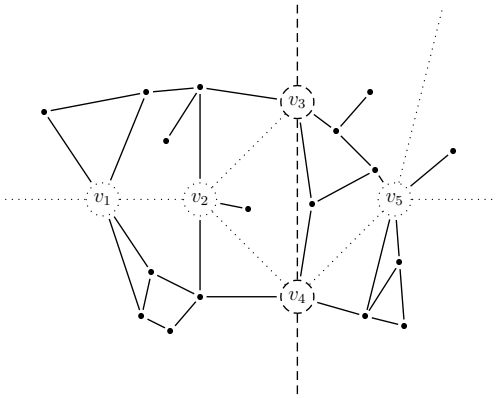

Figure 2: A graph in a possible two-level decomposition.
Dashed and dotted lines show component boundaries for level 1 and level 0, respectively. (The dashed line also separates level 0 components, cf. Lemma 8.) Vertices v_3 and v_4 are separators both at levels 0 and 1.

(c) The relationship set H_i^k defines, at level $k-1$, the indices of those components that are contained in G_i^k, the i-th component at level k:

$$\forall i \in I^k : G_i^k = \bigcup_{j \in H_i^k} G_j^{k-1}.$$

The trivial level n is included only for notational convenience, we also refer to it as *universe*. According to the definition, level 0 is always the lowest level with the finest separation. The components at level 0 do not have children. Consequently, level $n-1$ is the highest non-trivial level, all components at this level share the universe as parent. The universe itself has no parents, as the name suggests.

In such a hierarchy, we use S^k to denote the separator set at level k. The following lemma shows that the separator sets form a descending subset sequence with respect to k.

Lemma 8. *For all* $k \in \{1, \ldots, n\}$, $S^k \subseteq S^{k-1}$.

The relationship sets H_i^k indicate, for each component, which child components it contains. Conversely, at level k with $0 \leq k < n-1$, for every component G_i^k with $i \in I^k$, a *parent component index* $f^k(i)$ exists so that $i \in H_{f^k(i)}^{k+1}$. It is easy to see from Definition 7 that every component (apart from the universe) features exactly one parent component. Therefore, our definition of f^k is precise. Note that $f^{n-1}(i) = 0$ for all $i \in I^{n-1}$.

By rewriting Definition 7.2c, we instantly obtain

$$G_i^k \subseteq G_{f^k(i)}^{k+1}$$

for each suitable k and i.

Every component G_i^k has its set of boundary vertices B_i^k, defined as for the flat case. Note that B_0^n is empty — indeed, the universe is boundless.

In this work, we do not require home component functions for levels higher than level 0, so $h(v) \in I^0$ denotes the home component of a vertex v at level 0.

Figure 2 shows our graph in a possible hierarchical decomposition in two levels.

3.4 Search Space Parts

In this subsection, we formally define the so-called *search space part graphs* (in short, *search space parts*). Later, when answering a query for G, we merge several search space parts into a *search space graph* and run a shortest-path algorithm on that graph.

There are five kinds of search space parts: entry, exit, upward, downward and level graphs. Informally, the entry and exit graphs connect a vertex (separator or not) to boundary vertices of the home component at level 0, the upward and downward graphs connect boundary vertices of components at adjacent levels, and the level graphs connect boundary vertices of components that share the same parent component. To build a search space graph, we stick together one entry, several upward, one level, several downward and one exit graph.

The following definition specifies the properties of the part graphs. Figures 3, 4 and 5 show examples for the five part graph kinds.

Definition 9 (Search space parts). For a given hierarchical decomposition $\left\{G_i^k\right\}$ of an input graph G, we define the search space part graphs as follows:

Common properties All search space parts $\mathscr{P}$ share the following properties:

- Each search space part $\mathscr{P}$ is a directed bipartite graph. The vertices are divided into *source vertices* and *drain vertices*. Edges exist only from source to drain vertices, i.e., source vertices have no incoming, and drain vertices have no outgoing edges.
- A vertex of $\mathscr{P}$ is a pair with a vertex of G as the first and an integer as the second tuple component:

  $$V[\mathscr{P}] \subseteq V[G] \times \mathbb{Z}.$$

- The vertex set of $\mathscr{P}$ is essentially a disjoint union of two subsets of V, called *source base vertex set* and *drain base vertex set* (in short, *source base* and *drain base*) and denoted by Σ and Δ, respectively. (Although this is not supported by our notation, we would like to stress that the source and drain vertex sets are potentially different for each search space part.)

  Furthermore, each search space part defines two distinct integers that are used as *disambiguators* for the disjoint union:

  $$k_\sigma, k_\delta \in \mathbb{Z}, k_\sigma \neq k_\delta.$$

  (Again, k_σ and k_δ are chosen separately for each search space part.)

  As the name suggests, the source base vertex set (which is a subset of V) generates the source vertices of $\mathscr{P}$ (which are two-tuples): For a source base vertex $v_1 \in \Sigma$, we form the corresponding source vertex in $\mathscr{P}$ by using the source disambiguator k_σ as second tuple component:

  $$v_1 \in \Sigma \Longrightarrow (v_1, k_\sigma) \in V[\mathscr{P}]\,.$$

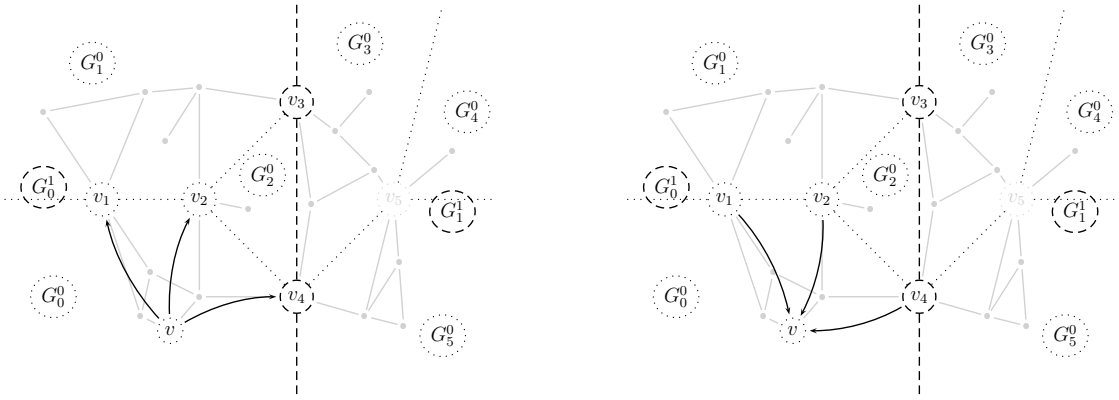

Figure 3: Entry and exit graphs.
The drawing on the left shows a designated vertex v with curved edges representing distances, in G_0^0, from v to the boundary vertices of its home component (edge targets). We do not examine paths that leave G_0^0. To compose the entry graph $\mathscr{E}_v$, we use these distances and append, to the source vertex v and to the boundary vertices, an integer tuple component for disambiguation: The vertex set of $\mathscr{E}_v$ becomes $\{(v,0),(v_1,1),(v_2,1),(v_4,1)\}$.
In the right-hand side drawing, the edges that build up the exit graph $\mathscr{X}_v$ are highlighted. Corresponding entry and exit graphs are antisymmetric except for the second tuple component of the vertices.

Conversely, a drain base vertex $v_2 \in \Delta$ induces a drain vertex in $\mathscr{P}$:

$$v_2 \in \Delta \Longrightarrow (v_2, k_\delta) \in V[\mathscr{P}]\,.$$

As expected from the disjoint union, a vertex $v \in \Sigma \cap \Delta$ has two corresponding vertices in $\mathscr{P}$, disambiguated with k_σ and k_δ. Yet, each vertex in $\mathscr{P}$ has exactly one corresponding vertex in G.

Formally, we set

$$V[\mathscr{P}] := (\Sigma \times \{k_\sigma\}) \cup (\Delta \times \{k_\delta\})\,.$$

(Recall also the commonly used definition of the disjoint union.)

- For a source vertex (v_1, k_σ), the cost to and a drain vertex (v_2, k_δ) matches the length of the shortest path from v_1 to v_2 in some component G_l^m. (Here, we fix l and m for one search space part, i.e., we use the same component to compute all costs. However, the indices are potentially different for each search space part.) Intuitively, each edge in $\mathscr{P}$ can be thought of as a "shortcut" for a shortest path in a component of G.

  Formally, for $v_1 \in \Sigma$ and $v_2 \in \Delta$, we set the cost of the corresponding edge in $\mathscr{P}$ as follows:

$$c[\mathscr{P}]\left((v_1, k_\sigma), (v_2, k_\delta)\right) := d[G_l^m]\,(v_1, v_2).$$

Entry graph For $s \in V$, an *entry graph* $\mathscr{E}_s$ represents the connection of a designated source vertex s to all the boundary vertices of the corresponding home component:

- The only source base vertex is s. We call s the *entry vertex*.

- The drain base is the set of boundary vertices of the home component, at level 0, of vertex s.
- We use $k_\sigma := 0$ and $k_\delta := 1$ as disambiguators.
- The cost from $(v, 0)$ to each appropriate $(t, 1)$ equals the distance between v and t in the home component. We do not consider paths that leave the home component:

$$c[\mathscr{E}_s]\left((s,0),(v,1)\right) := d\big[G^0_{h(s)}\big]\,(s,v).$$

Formally, an entry graph is a part graph with the following parameters:

$$\Sigma := \{s\},\ \Delta := B^0_{h(s)},\ k_\sigma := 0,\ k_\delta := 1,\ l := h(s),\ m := 0.$$

Exit graph For $t \in V$, an *exit graph* $\mathscr{X}_t$ is sort of an "inverse" entry graph that connects all boundary vertices of t's home component to t:

- The source base is the set of boundary vertices of the home component, at level 0, of vertex t.
- The only drain base vertex is t. We also refer to t as *exit vertex*.
- We use $k_\sigma := -1$ and $k_\delta := 0$ as disambiguators.
- The cost from each appropriate $(v, -1)$ to $(t, 0)$ equals the distance between v and t in the home component. Here, we also focus only on paths inside the home component:

$$c[\mathscr{X}_t]\left((v,-1),(t,0)\right) := d\big[G^0_{h(t)}\big]\,(v,t).$$

We can also specify an exit graph as a parametrized search space part graph:

$$\Sigma := B^0_{h(t)},\ \Delta := \{t\},\ k_\sigma := -1,\ k_\delta := 0,\ l := h(t),\ m := 0.$$

Upward graph For $0 \le k < n-1$ and for $i \in I^k$, an *upward graph* $\mathscr{U}_i^k$ connects the boundary vertices of component G_i^k to those of its parent component:

- The source base is B_i^k.
- The drain base is $B^{k+1}_{f^k(i)}$.
- We use $k_\sigma := k+1$ and $k_\delta := k+2$ as disambiguators.
- The cost between each pair $((s,k+1),(t,k+2))$ with $(s,t) \in B_i^k \times B^{k+1}_{f^k(i)}$ equals the distance between s and t in the parent component, not taking into account paths that leave that parent component:

$$c\big[\mathscr{U}_i^k\big]\left((s,k+1),(t,k+2)\right) := d\Big[G^{k+1}_{f^k(i)}\Big]\,(s,t).$$

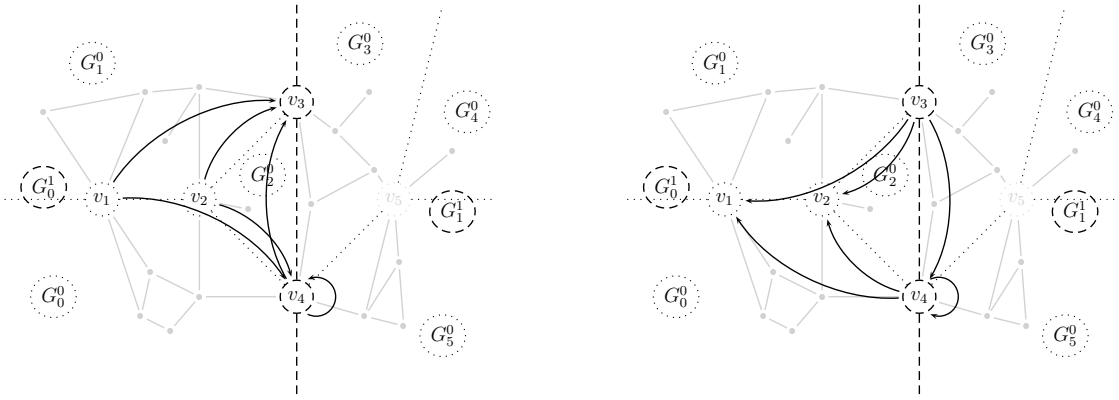

Figure 4: Upward and downward graphs.
In the left-hand side drawing, the curved edges depict the distances between boundary vertices of G_0^0 (edge sources) and those of its parent, G_0^1 (edge targets). The distances consider only paths that do not leave G_0^1, possible shorter paths via vertices in G_1^1 are not taken into account here. As v_4 is contained both in B_0^0 and B_0^1, it features a self-loop which we avoid otherwise. The upward graph $\mathscr{U}_0^0$ is composed from these distances by appending, to each boundary vertex used, an integer tuple component to assure uniqueness: The vertex set of $\mathscr{U}_0^0$ becomes $\{(v_1,1),(v_2,1),(v_4,1),(v_3,2),(v_4,2)\}$.
The drawing on the right shows the same for the downward graph $\mathscr{D}_0^0$. Nothing has changed except for the direction of the edges: The downward graphs are antisymmetric to the corresponding upward graphs.

We also provide an overview of this part graph's parameters:

$$\Sigma := B_i^k,\ \Delta := B_{f^k(i)}^{k+1},\ k_\sigma := k+1,\ k_\delta := k+2,\ l := i,\ m := k.$$

Additionally, we represent k in terms of k_σ:

$$k = k_\sigma - 1.$$

Downward graph For $0 \leq k < n-1$ and for $i \in I^k$, a *downward graph* $\mathscr{D}_i^k$ is an "inverse" upward graph, the source and drain bases are swapped. Downward graphs connect, for a component G_i^k, the boundary vertices of the parent component to those of the component itself:

- The source base is $B_{f^k(i)}^{k+1}$.
- The drain base is B_i^k.
- We use $k_\sigma := -k-2$ and $k_\delta := -k-1$ as disambiguators.
- The cost between each pair $((s,-k-2),(t,-k-1))$ with $(s,t) \in B_{f^k(i)}^{k+1} \times B_i^k$ equals the distance between s and t in the parent component, disregarding paths that leave the parent component:

$$c\left[\mathscr{D}_i^k\right]((s,-k-2),(t,-k-1)) := d\left[G_{f^k(i)}^{k+1}\right](s,t).$$

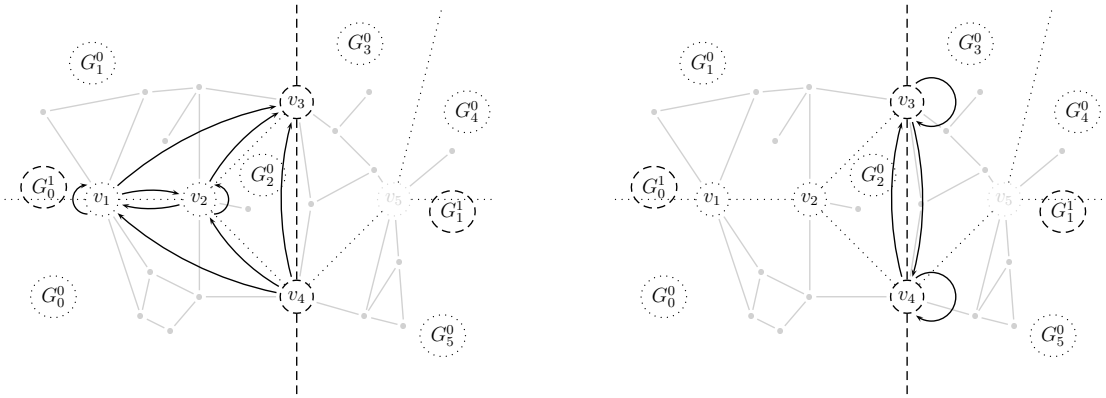

Figure 5: Level graph.
In the drawing at the left, the curved edges stand for the distances between boundary vertices of G_0^0 (edge sources) and those of G_1^0 (edge targets). The distances are in respect to the whole input graph, paths that do leave G_0^1 are also taken into account here. The vertices v_1 and v_2 are contained both in B_0^0 and B_1^0, so a self-loop has been added here for consistency. As with the other search space parts, we construct the level graph $\mathscr{L}_{0,1}^0$ by appending an integer tuple component to each of above boundary vertices for disambiguation: The vertex set of $\mathscr{L}_{0,1}^0$ becomes $\{(v_1,1),(v_2,1),(v_4,1),(v_1,-1),(v_2,-1),(v_3,-1)\}$.
The drawing at the right-hand side shows the same for $\mathscr{L}_{0,1}^1$. As v_3 and v_4 both are boundary vertices of G_0^1 and G_1^1, we obtain $\{(v_3,2),(v_4,2),(v_3,-2),(v_4,-2)\}$ as vertex set for $\mathscr{L}_{0,1}^1$.

Again, we summarize this by explicitly indicating the parameters for that part graph:

$$\Sigma := B_{f^k(i)}^{k+1},\ \Delta := B_i^k,\ k_\sigma := -k-2,\ k_\delta := -k-1,\ l := i,\ m := k.$$

We show how k can be computed from k_σ:

$$k = -k_\sigma - 2.$$

Level graph Finally, for $0 \le k \le n-1$ and for $i,j \in I^k$ with $f^k(i) = f^k(j)$, a *level graph* $\mathscr{L}_{i,j}^k$ connects boundary vertices of G_i^k to those of G_j^k:

- The source base is B_i^k.
- The drain base is B_j^k.
- We use $k_\sigma := k+1$ and $k_\delta := -k-1$ as disambiguators.
- The cost between each pair $((s,k+1),(t,-k-1))$ with $(s,t) \in B_i^k \times B_j^k$ equals the distance between s and t in the input graph. Here, we explicitly consider, for $k < n-1$, even those paths that leave the parent component. (For $k = n-1$, the universe is the parent.)

$$c\big[\mathscr{L}_{i,j}^k\big]\left((s,k+1),(t,-k-1)\right) := d[G]\,(s,t).$$

As usual, we shall formalize this. (Recall that $G_0^n = G$.)

$$\Sigma := B_i^k,\ \Delta := B_j^k,\ k_\sigma := k+1,\ k_\delta := -k-1,\ l := 0,\ m := n.$$

At last, we rewrite the equation for k and k_σ:

$$k = k_\sigma - 1.$$

Note the slight differences between up-/downward and level graphs. The latter are also defined for level $n-1$. Furthermore, in level graphs the distance between vertices matches the distance between corresponding vertices in the input graph, which may be shorter than the distance in the parent component graph used for entry, exit, upward and downward graphs.

In Section 5, we will show how to compute the search space parts efficiently. For now, we rely on their properties only and prove that we can use them to compute the distances between arbitrary vertex pairs in G.

3.5 Query

To answer a source-target shortest-path query between a source vertex s and a target vertex t, we craft a comparatively small search space graph for this vertex pair. (That is, for each distinct pair of vertices, we get a different search space graph.) For this subsection, we focus on a single distance query between some fixed vertices $s, t \in V$. Unless stated otherwise, we assume distinct home components:

$$h(s) \neq h(t).$$

The search space graph is combined, by means of graph union, from the search space parts introduced in the previous subsection: As mentioned before, we use one entry, several upward, one level, several downward and one exit graph. More precisely, we use $\mathscr{E}_s$ and $\mathscr{X}_t$ as entry and exit graph, respectively. By that, the entry and exit vertices (namely, $(s, 0)$ and $(t, 0)$) correspond to s and t in the input graph. The distance between those two vertices in the search space graph shall be proven to equal the distance between s and t in G. Thus, by running a distance query in our search space graph, we can solve the distance query in G.

Our search space graph is, in a way, a "tiny version" of the input graph G: Every path in G between s and t has a corresponding path in our search space graph that is at most as long as the path in G. The same is true in the opposite direction. By that, shortest paths must have the same length in both graphs. This is proven by our main theorem. To prepare the proof, we spot, on each path in G, those vertices that have corresponding vertices in the search space graph: Every path must contain certain boundary vertices in a well-defined order. (Recall that, apart from the entry and exit graphs, vertices in search space part graphs correspond to boundary vertices.)

First, we need to determine which components must be traversed by any path from s to t. We define the index sequences $i[k]$ and $j[k]$, starting at the corresponding home components at level 0, as follows:

$$i[0] \;:=\; h(s), \quad i[k+1] \;:=\; f^k(i[k])$$

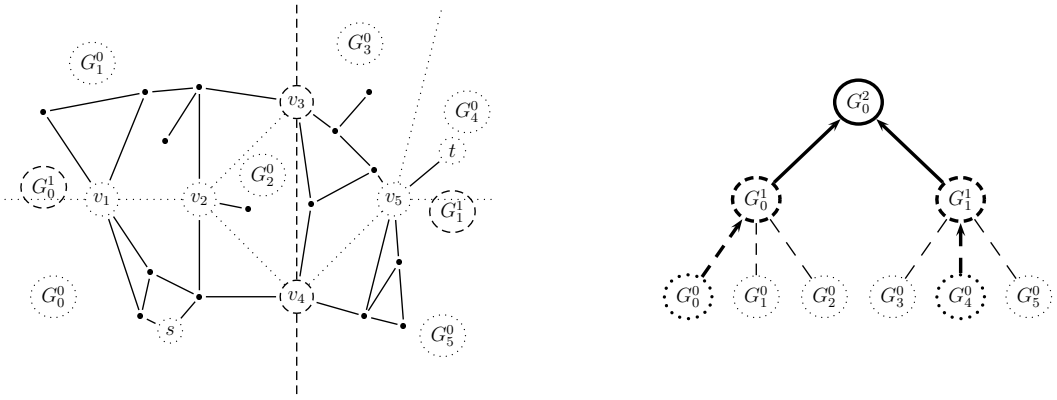

Figure 6: Component tree.
In computing the length of a shortest path, at first, we determine those components that must inevitably be hit by our path. We start in the home components of both source and target vertex, simultaneously walk up the hierarchy tree until we meet eventually, and note the components that we have visited in our walk.
If we wanted to retrieve a shortest path from s to t in the graph shown in the left-hand side drawing, we would thus start at G_0^0 and G_4^0, then move to their respective parent components G_0^1 and G_1^1, and finally meet at G_0^2, the universe.
In the drawing at the right, the bold lines show this path in the component tree. Here, $i[0\dots2] = \{0,0,0\}$, $j[0\dots2] = \{4,1,0\}$ and $m = 1$, as $i[1] \neq j[1]$ and $i[2] = j[2]$. All components drawn in bold contain the source or the target vertex, while G_0^2 is the only component that contains both.

$$j[0] \;:=\; h(t), \quad j[k+1] \;:=\; f^k(j[k])\,.$$

Let m be the smallest index so that $i[m] = j[m]$. As $f^{n-1}(l) = 0$ for all suitable l, we know that $i[n] = j[n]$, so m exists and is not greater than n. Informally, we simultaneously walk upward the component hierarchy tree from both source and target components at level 0, until we eventually meet at some level $m \leq n$. Note that also, for distinct home components of s and t, we can deduce $m > 0$. We provide a detailed example in Figure 6.

The components $G_{i[k]}^k$ and $G_{j[k]}^k$ form a subset sequence: For all $k < n$, the set inequalities

$$\begin{aligned} G_{i[k]}^k &\subseteq G_{i[k+1]}^{k+1} \\ G_{j[k]}^k &\subseteq G_{j[k+1]}^{k+1} \end{aligned}$$

both hold, which follows immediately by the definitions of component hierarchy and parent component index. In fact, any path between s and t crosses the boundary vertices of $G_{i[k]}^k$ in ascending order before crossing those of $G_{j[k]}^k$ in descending order, both with respect to k. We prove the following three lemmas to derive that in the next corollary. Figure 7 illustrates this statement.

Lemma 10. *For every path $p = \langle v_1, v_2, \dots, v_z \rangle$ between s and t with $v_1 = s$ and $v_z = t$, for each $0 \leq k < m$, the path contains at least one boundary vertex of component $G_{i[k]}^k$ and*

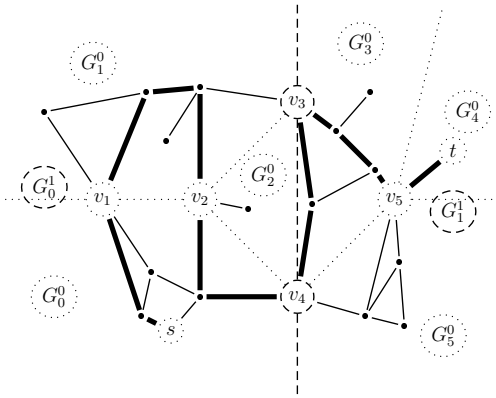

Figure 7: Boundary vertex indexes.
The drawing above shows a rather complicated path between s and t. As a result of Lemma 10, for all components G_0^0, G_0^1, G_1^1 and G_4^0, at least one boundary vertex must be encountered. We use the earliest for G_0^0 and G_0^1 and the latest for G_1^1 and G_0^4 — namely, v_1, v_4, v_3 and v_5 (in this order). Note that v_1 has been visited no later than v_4, and that v_5 has been visited no earlier than v_3, according to Lemma 11. Furthermore, as required by Lemma 12, v_4 is visited no later than v_3. Starting from one, v_1 is the third, v_4 is the eighth, v_3 is the tenth, and v_5 is the thirteenth vertex touched by the path. Thus, $x[0] = 3$, $x[1] = 8$, $y[1] = 10$ and $y[0] = 13$ — a nondecreasing sequence, as Corollary 13 states.

at least one of component $G^k_{j[k]}$. That is, there exist a (smallest) vertex index $x[k]$ with $v_{x[k]} \in B^k_{i[k]}$, and a (largest) vertex index $y[k]$ with $v_{y[k]} \in B^k_{j[k]}$ for each $0 \le k < m$.

Proof. For $k < m$, we have $i[k] \neq j[k]$, as m is the smallest index with $i[m] = j[m]$. We apply Lemma 3, which states that all paths between distinct components must contain boundary vertices, and conclude the claim straight away. □

Lemma 11. *For every path $p = \langle v_1, v_2, \ldots, v_z \rangle$ between s and t with $v_1 = s$ and $v_z = t$, $x[k]$ is ascending and $y[k]$ is descending with respect to k. That is, for each $k < m-1$, the following both inequalities hold:*

$$\begin{aligned} x[k] &\le x[k+1] \\ y[k] &\ge y[k+1]. \end{aligned}$$

Proof. We explicitly prove the first inequality only and deduce the second by antisymmetry.

Fix a path p between s and t arbitrarily, and $k < m-1$. As $x[k]$ is the index of the first separator vertex at level k, all v_x with $x < x[k]$ are not separators at level k. According to Lemma 8, they are no separators at level $k+1$, too. Hence, $x[k]$ is the smallest index in p that qualifies a vertex for possibly being a separator at level $k+1$, which is required for being a boundary vertex at level $k+1$. This satisfies our claim. □

Lemma 12. *For every path $p = \langle v_1, v_2, \ldots, v_z \rangle$ between s and t with $v_1 = s$ and $v_z = t$, the first boundary vertex at level $m-1$ for component $G^{m-1}_{i[m-1]}$ occurs before the last boundary vertex for component $G^{m-1}_{j[m-1]}$. That is,*

$$x[m-1] \le y[m-1].$$

Proof. Like in Lemma 11, all v_x with $x < x[m-1]$ are not separators at level $m-1$. As $v_{y[m-1]}$ is supposed to be a boundary vertex at level $m-1$, it must at least be a separator at this level. This makes $x[m-1]$ the smallest possible value for $y[m-1]$, as claimed. □

We provide the next corollary as a summary:

Corollary 13. *For every path $p = \langle v_1, v_2, \ldots, v_z \rangle$ between s and t with $v_1 = s$ and $v_z = t$, the vertices $v_{x[k]}$ and $v_{y[k]}$ for $0 \leq k < m$, as defined in Lemma 10, always occur in the following order:*

$$v_{x[0]}, v_{x[1]}, \ldots, v_{x[m-1]}, v_{y[m-1]}, \ldots, v_{y[1]}, v_{y[0]}$$

Proof. Immediately from Lemmas 11 and 12. □

Now, we are ready to construct the search space graph. We connect one entry, m upward, one level, m downward and one exit graphs. The following definition describes this construction in detail.

Definition 14 (Search space graph). The graph

$$\mathscr{G}_{s,t} := \mathscr{E}_s \cup \left(\bigcup_{0 \leq k < m-1} \mathscr{U}^k_{i[k]} \right) \cup \mathscr{L}^{m-1}_{i[m-1],j[m-1]} \cup \left(\bigcup_{0 \leq k < m-1} \mathscr{D}^k_{j[k]} \right) \cup \mathscr{X}_t$$

is called the *search space graph for* (s,t). The vertices $\nu_s := (s,0)$ and $\nu_t := (t,0)$ are called *source* or *drain vertex of the search space*, respectively.

Figure 8 explains the construction of the search space graph.

It can be easily verified from Definition 9, that each edge in the search space graphs originates from exactly one search space part used in the composition. That is, for a given edge in the search space graph, we can deduce exactly which search space part contains this edge. This is stated more precisely in the following lemma:

Lemma 15. *Each edge $((v_1, k_\sigma), (v_2, k_\delta))$ in $\mathscr{G}_{s,t}$, with $v_1, v_2 \in V$ and $k_\sigma, k_\delta \in \{0, \ldots, m-1\}$, can be found in one and only one of the search space parts used to compose $\mathscr{G}_{s,t}$.*

Proof. If $((v_1, k_\sigma), (v_2, k_\delta))$ is an edge of $\mathscr{G}_{s,t}$, it must be present in one of the search space parts used to create $\mathscr{G}_{s,t}$, because the graph union employs only vertices and edges of the graphs to be united and does not add new edges.

We distinguish between five cases. Each case matches a type of search space part. For each case, we quote the search space part where this edge came from.

1. $k_\sigma = 0, k_\delta = 1$. In this case, the edge originates in the entry graph $\mathscr{E}_s$, as no other search space part features edges with 0 or 1, respectively, as disambiguators.

2. $k_\sigma > 0, k_\delta > 0$. This condition is fulfilled only for edges in an upward graph. That given, we know that $k_\delta = k_\sigma + 1$. From the definition of the upward graph follows, for $k := k_\sigma - 1$, that $\mathscr{U}^k_{i[k]}$ contains this edge.

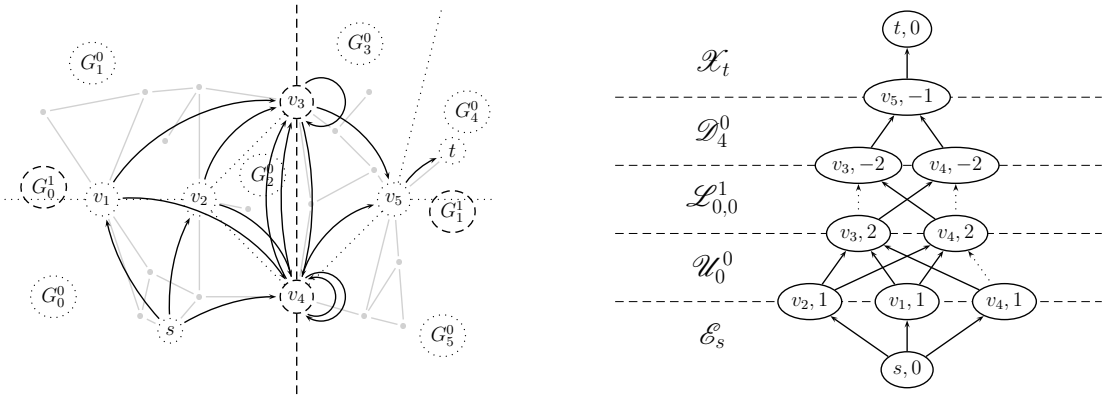

Figure 8: Search space graph.
For the graph shown in the left-hand side drawing, the corresponding search space graph for a source-target query between the vertices s and t is composed from $\mathscr{E}_s$, $\mathscr{U}_0^0$, $\mathscr{L}_{0,1}^1$, $\mathscr{D}_4^0$ and $\mathscr{X}_t$. The highlighted edges represent distances between boundary vertices that were used to generate those search space parts. (Compare that to Figures 3 to 5.)
The drawing on the right shows $\mathscr{G}_{s,t}$. The vertices of this graphs are two-tuples, each formed from a vertex of G and an integer disambiguator. Each edge corresponds uniquely to one of the highlighted edges in the left-hand side drawing; the dotted ones on the right have a cost of zero, they correspond to self-loops in the drawing at the left. The distance between $(s, 0)$ and $(t, 0)$ in $\mathscr{G}_{s,t}$ equals that between s and t in G.
As shown in the drawing, the search space graph can be arranged in stages, where each stage corresponds to a search space part the graph has been composed of. This implies also a topological sort of $\mathscr{G}_{s,t}$.

3. $k_\sigma > 0, k_\delta < 0$. Here, we have an edge from a level graph. In this case, $k_\delta = -k_\sigma$. By applying the definition of the level graph for $k := k_\sigma - 1$, we conclude that $\mathscr{L}_{i[k],j[k]}^k$ must contain our edge.

4. $k_\sigma < -1$. This case is antisymmetrical to case 2. Our edge must be part of a downward graph, thus, $k_\delta = k_\sigma + 1$. From the definition of the downward graph follows, for $k := -k_\sigma - 1$, that $\mathscr{D}_{i[k]}^k$ contains this edge.

5. $k_\sigma = -1, k_\delta = 0$. The argumentation for the final case is just like that for case 1, this edge is part of the exit graph $\mathscr{X}_t$.

The case differentiation is complete, as the conditions used are disjoint, and no other cases may exist for the disambiguators. □

We call an edge from $\mathscr{G}_{s,t}$ by the type of the search space part where this edge originates. That is, an edge from $\mathscr{E}_s$ is called *entry edge*, and so on. This will help to clarify later case differentiations.

In our main theorem, we prove that we can use the search space graph to compute the distance between s and t in G: The distance between source and drain vertex in $\mathscr{G}_{s,t}$ matches the distance between s and t in G.

Theorem 16. *The distance between s and t in G is the same as the distance between $(s,0)$ and $(t,0)$ in $\mathscr{G}_{s,t}$:*

$$d[G]\,(s,t) = d[\mathscr{G}_{s,t}]\,((s,0),(t,0)).$$

Proof. We prove that for every path in G between s and t, there exists a corresponding path in $\mathscr{G}_{s,t}$ between $(s,0)$ and $(t,0)$ that is not longer, and vice versa. Thereby, it is proven that the distance, or length of a shortest path, must be the same in both graphs.

($\Leftarrow$) Fix an arbitrary path $\pi = \langle \nu_1, \nu_2, \ldots, \nu_z \rangle$ in $\mathscr{G}_{s,t}$ with $\nu_1 = (s,0)$ and $\nu_z = (t,0)$. We show that there exists a corresponding path $p = \langle v_1, v_2, \ldots, v_w \rangle$ in G with $v_1 = s$ and $v_w = t$ that is not longer than π.

We start with an empty path p for G. For each adjacent pair of vertices $(\nu_x, \nu_{x+1}) = ((v_x, k_x), (v_{x+1}, k_{x+1}))$ in π, we append, to p, some shortest path $p_x := \langle v_x, \ldots, v_{x+1} \rangle$ between v_x and v_{x+1} in G. Thus, $c[G]\,(p_x) = d[G]\,(v_x, v_{x+1})$. As $\nu_1 = (s,0)$ and $\nu_z = (t,1)$, the path p indeed is a path between s and t. We show, for each subpath added to p, that its length is not larger than the cost of the corresponding edge in $\mathscr{G}_{s,t}$. We use Lemma 15 to classify the edges in π, and Definition 9 to bound the path cost:

Entry edge

$$d[G]\,(v_x, v_{x+1}) \leq d\big[G^0_{h(s)}\big]\,(v_x, v_{x+1}) = c[\mathscr{E}_s]\,(\nu_x, \nu_{x+1}).$$

Upward edge Set $k := k_x$:

$$d[G]\,(v_x, v_{x+1}) \leq d\Big[G^{k+1}_{i[k+1]}\Big]\,(v_x, v_{x+1}) = c\big[\mathscr{U}^k_{i[k]}\big]\,(\nu_x, \nu_{x+1}).$$

Level edge Set $k := k_x$, too:

$$d[G]\,(v_x, v_{x+1}) = c\Big[\mathscr{L}^{k-1}_{i[k-1],j[k-1]}\Big]\,(\nu_x, \nu_{x+1}).$$

Downward edge Set $k := -k_{x+1}$ (antisymmetrical to the "upward" case):

$$d[G]\,(v_x, v_{x+1}) \leq d\Big[G^{k+1}_{j[k+1]}\Big]\,(v_x, v_{x+1}) = c\big[\mathscr{D}^k_{j[k]}\big]\,(\nu_x, \nu_{x+1}).$$

Entry edge (antisymmetrical to the "entry" case):

$$d[G]\,(v_x, v_{x+1}) \leq d\big[G^0_{h(t)}\big]\,(v_x, v_{x+1}) = c[\mathscr{X}_t]\,(\nu_x, \nu_{x+1}).$$

Thus, the length of p_x is never greater than the cost of the edge (ν_x, ν_{x+1}) in $\mathscr{G}_{s,t}$. By that, we can safely claim that p is at most as long as π.

($\Rightarrow$) Fix an arbitrary path $p = \langle v_1, v_2, \ldots, v_z \rangle$ in G with $v_1 = s$ and $v_z = t$. We show that there exists, in $\mathscr{G}_{s,t}$, a corresponding path $\pi = \langle \nu_1, \nu_2, \ldots, \nu_w \rangle$ with $\nu_1 = (s,0)$ and $\nu_w = (t,0)$ not longer than p.

For the definition of the path, we use, for p, the index sequences $x[k]$ and $y[k]$ defined in Lemma 10. Set $\pi := \langle (s,0), (v_{x[0]},1), \ldots, (v_{x[m-1]},m), (v_{y[m-1]},-m), \ldots, (v_{y[0]},-1), (t,0)\rangle$. As $m > 0$, the path contains at least the vertices $(v_{x[m-1]},m)$ and $(v_{y[m-1]},-m)$, apart from start and end vertex. Yet to prove is that each adjacent vertex pair indeed is an edge in $\mathscr{G}_{s,t}$. For each vertex pair, we name a search space part from Definition 14 that contains this vertex pair. Again, we distinguish between five cases:

Entry edge The edge $((s,0),(v_{x[0]},1))$ is present in the entry graph $\mathscr{E}_s$, because $v_{x[0]}$ is a boundary vertex of $G^0_{h(s)}$.

Upward edge For $0 \le k < m-1$, the edge $((v_{x[k]},k+1),(v_{x[k+1]},k+2))$ can be found in the upward graph $\mathscr{U}^k_{i[k]}$, for both $v_{x[k]}$ and $v_{x[k+1]}$ being boundary vertices for the components $G^k_{i[k]}$ and $G^{k+1}_{i[k+1]}$, respectively.

Level edge The level graph $\mathscr{L}^{m-1}_{i[m-1],j[m-1]}$ contains the edge $(v_{x[m-1]},m),(v_{y[m-1]},-m)$, as both $v_{x[m-1]}$ and $v_{y[m-1]}$ are boundary vertices for $G^{m-1}_{i[m-1]}$ and $G^{m-1}_{j[m-1]}$, respectively.

Downward edge For $0 \le k < m-1$, the edge $((v_{y[k]},-k-2),(v_{y[k+1]},-k-1))$ is present in the downward graph $\mathscr{D}^k_{j[k]}$, in analogy to the "upward" case.

Exit edge The edge $((v_{y[0]},-1),(t,0))$ appears in the exit graph $\mathscr{X}_t$, this is antisymmetrical to the "entry" case.

To prove that π is not longer than p, we split p into subpaths and bound each edge of π by the length of a distinct subpath:

$$p = p_{1\to x[0]} + \cdots + p_{x[m-2]\to x[m-1]} + p_{x[m-1]\to y[m-1]} + p_{y[m-1]\to y[m-2]} + \cdots + p_{y[0]\to z}.$$

We know from Corollary 13 that the start index of above subpaths is never larger than the end index, so we may decompose p like that.

Throughout the following case differentiation, we use the definition of the search space parts to bound the distance of a search space edge by a distance in some component of the input graph and argue that the subpath does not leave that component:

Entry edge The cost of the entry edge $\varepsilon := ((s,0),(v_{x[0]},1))$ can be bounded by the length of $p_{1\to x[0]}$:

$$\begin{aligned} c[\mathscr{E}_s](\varepsilon) &= d\left[G^0_{h(s)}\right](s, v_{x[0]}) \\ &\le c\left[G^0_{h(s)}\right](p_{1\to x[0]}) &= c[G](p_{1\to x[0]}). \end{aligned}$$

The second equality (line 2) follows from Corollary 5, owing to the fact that all vertices with index smaller than $x[0]$ are not separators at level 0.

Upward edge For $0 \leq k < m-1$, we use the length of $p_{x[k]\to x[k+1]}$ to limit the cost of an upward edge $\upsilon := ((v_{x[k]}, k+1), (v_{x[k+1]}, k+2))$:

$$\begin{aligned} c\left[\mathscr{U}^k_{i[k]}\right](\upsilon) &= d\left[G^{k+1}_{i[k+1]}\right](v_{x[k]}, v_{x[k+1]}) \\ &\leq c\left[G^{k+1}_{i[k+1]}\right](p_{x[k]\to x[k+1]}) \quad = \quad c[G]\,(p_{x[k]\to x[k+1]}). \end{aligned}$$

As in case 1, we obtain the second equality from the definition of $x[k+1]$ and Corollary 5. Note that, for the given bounds for k, we use each of the subpaths

$$p_{x[0]\to x[1]}, \ldots, p_{x[m-2]\to x[m-1]}$$

exactly once.

Level edge The cost of the level edge $\lambda := ((v_{x[m-1]}, m), (v_{y[m-1]}, m))$ is not higher than the length of $p_{x[m-1]\to y[m-1]}$:

$$\begin{aligned} c\left[\mathscr{L}^{m-1}_{i[m-1],j[m-1]}\right](\lambda) &= d[G]\,(v_{x[m-1]}, v_{y[m-1]}) \\ &\leq c[G]\,(p_{x[m-1]\to y[m-1]}). \end{aligned}$$

Downward edge Antisymmetrically to the "upward" case, we use, for $0 \leq k < m-1$, the length of $p_{x[k]\to x[k+1]}$ to bound the cost of a downward edge $\delta := ((v_{x[k]}, -k-2), (v_{x[k+1]}, -k-1))$:

$$\begin{aligned} c\left[\mathscr{D}^k_{i[k]}\right](\delta) &= d\left[G^{k+1}_{j[k+1]}\right](v_{y[k+1]}, v_{y[k]}) \\ &\leq c\left[G^{k+1}_{j[k+1]}\right](p_{y[k+1]\to y[k]}) \quad = \quad c[G]\,(p_{y[k+1]\to y[k]}). \end{aligned}$$

Here, we employ every subpath

$$p_{y[m-1]\to y[m-2]}, \ldots, p_{y[1]\to y[0]}$$

exactly once.

Exit edge Finally, we perform the estimation for the exit edge $\xi := ((v_{y[0]}, -1), (t, 0))$ in analogy to that of the exit edge, using $p_{y[0]\to z}$:

$$\begin{aligned} c[\mathscr{X}_t](\xi) &= d\left[G^0_{h(t)}\right](v_{y[0]}, t) \\ &\leq c\left[G^0_{h(t)}\right](p_{y[0]\to z}) \quad = \quad c[G]\,(p_{y[0]\to z}). \end{aligned}$$

Thus, we can bound each edge of π by the length of a corresponding subpath of p. We use each subpath of p exactly once in this estimation. This proves that π indeed is not longer than p.

We have explicitly stated, in both directions, for any path in one graph a corresponding path in the other graph that is not longer. This shows that the distance between s and t in G can be computed from that between $(s, 0)$ and $(t, 0)$ in $\mathscr{G}_{s,t}$. □

Next, we will show how to compute the distance in a search space part graph in linear time. As we shall see, any search space part graph is a DAG, i.e., it contains no paths with the same start and end vertex. To show that, we define a relation for all vertices of the search space graph $\mathscr{G}_{s,t}$, show that it is a strict linear order and conclude that it is indeed a topological sort for $\mathscr{G}_{s,t}$ in the forthcoming two lemmas.

Definition 17 (Relation for the vertices of $\mathscr{G}_{s,t}$). We define a relation $\triangleleft$ for vertices of $\mathscr{G}_{s,t}$ as follows:

- The vertex $(s, 0)$ is the "minimum": For all $(v, k_\delta) \in V[\mathscr{G}_{s,t}] \setminus \{(s, 0)\}$,

$$(s, 0) \triangleleft (v, k_\delta).$$

- The vertex $(t, 0)$ is the "maximum": For all $(v, k_\sigma) \in V[\mathscr{G}_{s,t}] \setminus \{(t, 0)\}$,

$$(v, k_\sigma) \triangleleft (t, 0).$$

- The other vertices are arranged in ascending order with respect to the value of the disambiguator (second tuple component), except that vertices with positive disambiguator values are always arranged before vertices with negative disambiguator values in our relation. Formally, any two vertices $(v_1, k_\sigma), (v_2, k_\delta) \in V[\mathscr{G}_{s,t}]$ with $k_\sigma \neq 0$ and $k_\delta \neq 0$ are ordered as

$$(v_1, k_\sigma) \triangleleft (v_2, k_\delta)$$

  iff one of the following conditions is true:

  - $0 < k_\sigma < k_\delta$
  - $k_\sigma < k_\delta < 0$
  - $k_\delta < 0 < k_\sigma$

The relation is irreflexive and transitive. Hence, it is a strict linear order:

Lemma 18. *The relation $\triangleleft$ is a strict linear order.*

Proof. Irreflexivity is obvious. Transitivity can be proven by case differentiation. □

The linear order is similar for all possible search space graphs. Source and drain vertices are always minimum or maximum, respectively. For the other vertices, those that originate in an upward graph are smaller than those from a downward graph; vertices from an upward graph are smaller than those from any upward graph at a higher level; and vertices from a downward graph are ordered before vertices from downward graphs at lower levels. Intuitively, this order matches the sequence of visited vertices for any path from source to drain in $\mathscr{G}_{s,t}$. The following lemma formalizes that:

Lemma 19. *The linear order $\lhd$ is a topological sort for $\mathscr{G}_{s,t}$. That is, for any edge $((v_1, k_\sigma), (v_2, k_\delta))$ in $\mathscr{G}_{s,t}$ that is not a self-loop, the source vertex is smaller than the target vertex in our linear order:*

$$(v_1, k_\sigma) \lhd (v_2, k_\delta).$$

Proof. Fix an arbitrary edge $((v_1, k_\sigma), (v_2, k_\delta))$ in $\mathscr{G}_{s,t}$ with $(v_1, k_\sigma) \neq (v_2, k_\delta)$. We prove the claimed relation by means of case differentiation by edge kind, in accordance with Lemma 15:

Entry edge Here, the source node is the minimum in our partial order: $(v_1, k_\sigma) = (s, 0)$. The claim follows straight away.

Upward edge From Lemma 15 it follows, for this case, that $k_\sigma > 0$ and $k_\delta = k_\sigma + 1$. By that, $0 < k_\sigma < k_\sigma$, hence, our claim is satisfied.

Level edge Finally, for level edges, $k_\sigma > 0$ and $k_\delta = -k_\sigma$. Accordingly, it follows that $k_\delta < 0 < k_\sigma$, from which our claim follows.

Downward edge By antisymmetry to the second case, we have $k_\sigma < -1$ and $k_\delta = k_\sigma + 1$ for this case. As the inequality $k_\sigma < k_\delta < 0$ holds then, we conclude our claim for this case, too.

Exit edge Antisymmetrically to the first case, the target node is the maximum in our partial order for this case: $(v_2, k_\delta) = (t, 0)$.

We have concluded our claim for each possible case. This proves that indeed $\lhd$ is a topological sort for $\mathscr{G}_{s,t}$. □

According to [CLRS01], Sect. 22.4, p. 549, this makes the search space graph a DAG. This enables us to run a query in linear time in terms of number of vertices and edges of the search space. Likewise, in [CLRS01], Sect. 24.2, pp. 592-594, a source-target shortest-path algorithm for arbitrary DAGs is presented and proven correct.

Algorithm 1 is a variant of that algorithm that is tailored to our search space graphs. We do not need to explicitly compute the topological sort, as it is a structural property of any search space graph.

The algorithm uses two procedures. The procedure `update` updates the distance labels of all drain vertices for a given search space part. We call this procedure once for each search space part. Before each call to `update`, all distance labels of the the current search space part's source vertices have already been computed, as we obey the order imposed by our topological sort $\lhd$. For the first call, there is only one source, namely, ν_s, the source of the search space graph, for which we know the distance to be zero. The procedure `update` calls `relax` once for each edge of the search space part graph. The "relaxation of an edge" means the update of the distance label for an edge's target vertex if the distance via the edge's source vertex is shorter than the previously known distance.

Algorithm 1: Query algorithm.

foreach *vertex* ν *of* $\mathscr{G}_{s,t}$ **do**
  $d[\nu] \leftarrow \infty$
$d[\nu_s] \leftarrow 0$
initialize d to ∞ for all other vertices in $\mathscr{G}_{s,t}$
`update`$(\mathscr{E}_s)$
for $k \leftarrow 0$ **to** $m-1$ **do**
  `update`$(\mathscr{U}^k_{i[k]})$
`update`$(\mathscr{L}^{m-1}_{i[m-1],j[m-1]})$
for $k \leftarrow m-1$ **downto** 0 **do**
  `update`$(\mathscr{D}^k_{i[k]})$
`update`$(\mathscr{X}_t)$
return $d[\nu_t]$

Procedure `update`$(\mathscr{P})$

forall *source vertices* ν_1 *of* $\mathscr{P}$ **do**
  foreach *drain vertex* ν_2 *of* $\mathscr{P}$ *with* $c[\mathscr{P}](\nu_1, \nu_2) < \infty$ **do**
    `relax`(ν_1, ν_2)

Procedure `relax`(ν_1, ν_2)

$d[\nu_2] \leftarrow \min(d[\nu_2], d[\nu_1] + c(\nu_1, \nu_2))$

An execution of `relax` can be performed in constant time. Each run of `update` calls `relax` for each edge exactly once, and enumerates all source and drain vertices. As we call `update` once for each search space part, the algorithm requires linear time in the number of vertices and edges in the search space graph.

Finally, we show how to deal with those queries where source and target lie in the same home component. We offer two options:

1. For all components at level 0, precompute and store the distances between all pairs of vertices.

2. Run a source-target query algorithm.

The first option tends to be tremendously expensive in terms of memory consumption. Assuming a graph that has been decomposed in components of roughly the same size $r := |V|/|I^0|$, we need to store

$$|I^0| \cdot r^2 = \frac{|V|}{|I^0|} \cdot \left(|I^0| \cdot \frac{|V|}{|I^0|} \right) = r \cdot |V|$$

distance values. If many queries for nearby vertices can be expected, this may seem feasible. However, the vast majority of precomputed distances might never be queried for.

For the second option, we cannot assume that a shortest path will not leave the common home component. However, we can efficiently determine the length of a shortest path that contains a boundary vertex: Like in the general case, we can construct a search space graph and query the distance between source and drain vertex. By setting $m := 1$, we can use $\mathscr{G}_{s,t}$ from Definition 14 straight away. Note that no upward or downward graph will be utilized in this case. Furthermore, as $i[0] = j[0]$, we use a level graph that we would never employ in the general case. Similarly to the proof of Theorem 16, we can prove that the distance in the search space graph equals the length of the shortest among all paths between s and t that contain at least one boundary node.

Thus, we can run a source-target query algorithm on the home component to determine the distance inside the component, then compare this distance to the length of a shortest path via a boundary vertex. The shortest of both distances is the length of a shortest path in the whole input graph.

If the components at bottom level are small enough, we can employ Dijkstra's algorithm to determine the distance between two vertices in a bottom-level component. Yet this approach is efficient only if the number of vertices in a bottom-level component is low. For a large component at the bottom level, we can recursively apply the whole preprocessing with this bottom-level component as input graph. Note that this is essentially different from just adding one more level to the hierarchical decomposition. A detailed analysis is beyond the scope of this work.

4 Optimization

We have introduced the general approach for our speed-up technique in the previous section. In this section, we shall refine it by removing unneeded edges from the search space parts and transforming the search space parts into equivalent graphs with fewer edges. Both improvements reduce both size of the preprocessed data and query time, while maintaining correctness.

We assume that the search space parts have been computed and stored as complete bipartite graphs. The next section shows how the search space parts can be computed efficiently.

4.1 Removal of Superseded Edges

Recall that the edges in the search space parts introduced in the previous section represent shortest paths in some component of the input graph. For instance, an edge in an upward graph represent a shortest path between boundary vertices at adjacent levels. The search space parts are always used in connection with other search space parts to form a search space graph where a distance query from source to drain is executed. If we can prove, for an edge in a search space part, that its omission will not change the distance between source and drain in *any* possible search space graph, we can safely remove it.

In this section, we will provide a sufficient condition for removing superseded edges in upward graphs and prove that the removal can occur in arbitrary order. From that, we derive a straightforward algorithm that removes all superseded edges from an upward graph, and analyze its run time. We will also present, without proof, the corresponding algorithms for downward and level graphs. For the remainder of this section, we will focus on a fixed upward graph $\mathscr{U}_i^k$, unless stated otherwise.

Intuitively, an edge can be omitted if it represents a shortest path that immediately re-enters the child component G_i^k. Figure 9 provides a detailed example. We can even quote a slightly weaker condition that relies only on distances in child separator closures. For this, we define a relation between edges of the same search space part. We can omit those edges that are related, on the right-hand side, to any other edge:

Definition 20 (Supersedement). Let $\nu_{\looparrowright} = (v_{\looparrowright}, k+1)$ and $\nu_{\curvearrowright} = (v_{\curvearrowright}, k+1)$ be two source vertices, and $\nu_* = (v_*, k+2)$ be a drain vertex of $\mathscr{U}_i^k$. The edge $(\nu_{\curvearrowright}, \nu_*)$ *supersedes* the edge $(\nu_{\looparrowright}, \nu_*)$, iff the following two conditions are fulfilled:

$$d\big[G_i^k\big]\,(v_{\looparrowright}, v_{\curvearrowright}) > 0$$

$$d\big[G_i^k\big]\,(v_{\looparrowright}, v_{\curvearrowright}) + c\big[\mathscr{U}_i^k\big]\,(\nu_{\curvearrowright}, \nu_*) \leq c\big[\mathscr{U}_i^k\big]\,(\nu_{\looparrowright}, \nu_*).$$

In this case, we write

$$(\nu_{\curvearrowright}, \nu_*) \lessdot_L (\nu_{\looparrowright}, \nu_*).$$

We call $(\nu_{\curvearrowright}, \nu_*)$ the *superseding* and $(\nu_{\looparrowright}, \nu_*)$ the *superseded* edge. (As for the vertex indexes, intuitively, the looped arrow stands for a detour, while the curved arrow is supposed to indicate a more direct path. The star symbolizes the common target of both edges.)

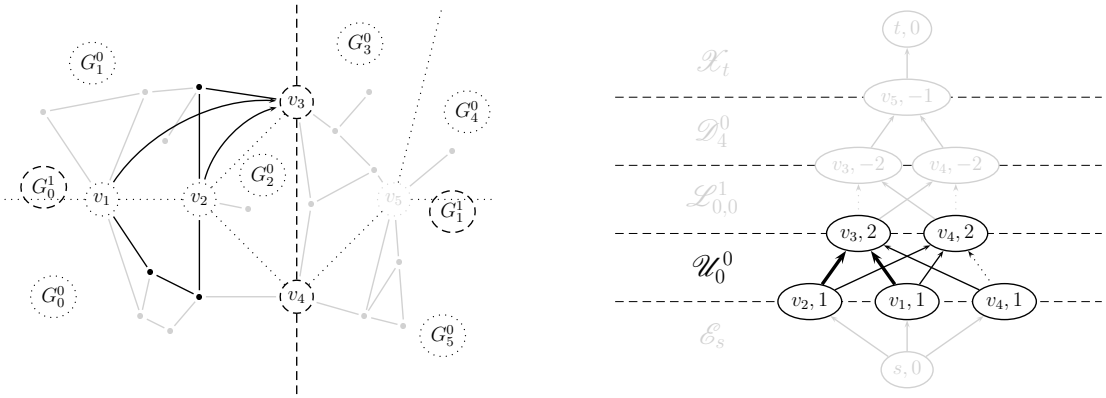

Figure 9: Superseded edges.
In the left-hand side drawing, the highlighted straight edges represent a shortest path between v_1 and v_3 in G_0^1. The curved edges represent distances that are used to create $\mathscr{U}_0^0$. The corresponding edges in the upward graph are shown in bold in the drawing at the right.
As a shortest path from v_1 to v_3 visits v_2 before leaving G_0^0, we can safely claim that any path from *any* vertex in G_0^0 to v_3 via v_1 cannot be shorter than a shortest path to v_3 via v_2: For any shortest path via v_1, there is an equally short path that uses both v_1 and v_2, but does not leave G_0^0 before reaching v_2. Thus, the distance between v_1 and v_3 serves no purpose for the upward graph $\mathscr{U}_0^0$, and the corresponding edge in the upward graph can be removed. In this case, the edge $((v_2, 1), (v_3, 2))$ supersedes the edge $((v_1, 1), (v_3, 2))$.

We use the supersedement relation as an indicator when to omit edges. We shall show, in the next lemma, that we are able to omit any superseded edge.

Lemma 21. *For an arbitrary search space graph $\mathscr{G}_{s,t}$ that contains $\mathscr{U}_i^k$ (i.e., $s \in G_i^k$ and t "far enough" from s), let $\nu_{\curvearrowright} = (v_{\curvearrowright}, k+1)$ and $\nu_{\curvearrowright} = (v_{\curvearrowright}, k+1)$ be two source vertices, and $\nu_* = (v_*, k+2)$ be a drain vertex of $\mathscr{U}_i^k$. Furthermore, let $(\nu_{\curvearrowright}, \nu_*)$ supersede $(\nu_{\curvearrowright}, \nu_*)$, i.e.,*

$$(\nu_{\curvearrowright}, \nu_*) \lessdot_L (\nu_{\curvearrowright}, \nu_*).$$

Then, for any shortest path between source and drain of the search space graph that contains the superseded edge, there exists another path that has the same length and contains the superseding edge instead.

Proof. (Sketch.) Fix arbitrarily $s, t \in V$ so that $\mathscr{G}_{s,t}$ contains $\mathscr{U}_i^k$ according to Definition 14. Fix a shortest path $\pi = \langle \nu_1, \nu_2, \ldots, \nu_z \rangle$ between source and drain of the search space graph that contains the superseded edge, i.e., with $\nu_1 = \nu_s$, $\nu_z = \nu_t$, $\nu_x = \nu_{\curvearrowright}$ and $\nu_{x+1} = \nu_*$. (Note that the edge (ν_x, ν_{x+1}) is contained in $\mathscr{U}_i^k$.) For $\pi_{1 \to (x+1)}$, the subpath of π between ν_s and ν_*, we are going to find an equally long path π'', also between ν_s and ν_*, that contains $\nu_{\curvearrowright}$ instead of $\nu_{\curvearrowright}$. The concatenation of π'' and $\pi_{(x+1) \to z}$ is a path between ν_s and ν_t that is as long as π and contains $\nu_{\curvearrowright}$ instead of $\nu_{\curvearrowright}$. Thus, by presenting an appropriate path π'', we will have satisfied our claim.

Let π' be a shortest path between ν_s and $\nu_{\curvearrowright}$. Then, by concatenating π' and the superseding edge $(\nu_{\curvearrowright}, \nu_*)$, we obtain a path π'' between ν_s and ν_* that is as long as $\pi_{1 \to (x+1)}$:

$$\pi'' := \pi' + (\nu_{\curvearrowright}, \nu_*).$$

The following backward argumentation proves that the lengths of π'' and $\pi_{1\to(x+1)}$ are equal. For a transition, the expressions that have changed are highlighted. Some of the non-obvious transitions are marked with a number on the right and explained in detail below:

$$
\begin{aligned}
& & c[\mathscr{G}_{s,t}]\,(\pi'') &= c[\mathscr{G}_{s,t}]\,(\pi_{1\to(x+1)}) & \\
\Leftrightarrow\ & c[\mathscr{G}_{s,t}]\,(\pi') + c[\mathscr{G}_{s,t}]\,(\nu_{\curvearrowright}, \nu_*) &= c[\mathscr{G}_{s,t}]\,(\pi_{1\to(x+1)}) & \\
\Leftarrow\ & c[\mathscr{G}_{s,t}]\,(\pi') + c[\mathscr{G}_{s,t}]\,(\nu_{\curvearrowright}, \nu_*) &\leq c[\mathscr{G}_{s,t}]\,(\pi_{1\to(x+1)}) & (1)\\
\Leftrightarrow\ & c[\mathscr{G}_{s,t}]\,(\pi') + c[\mathscr{G}_{s,t}]\,(\nu_{\curvearrowright}, \nu_*) &\leq c[\mathscr{G}_{s,t}]\,(\pi_{1\to x}) + c[\mathscr{G}_{s,t}]\,(\nu_{\curvearrowright\!\!\to}, \nu_*) & \\
\Leftrightarrow\ & c[\mathscr{G}_{s,t}]\,(\pi') - c[\mathscr{G}_{s,t}]\,(\pi_{1\to x}) &\leq c[\mathscr{G}_{s,t}]\,(\nu_{\curvearrowright\!\!\to}, \nu_*) - c[\mathscr{G}_{s,t}]\,(\nu_{\curvearrowright}, \nu_*) & \\
\Leftrightarrow\ & d[\mathscr{G}_{s,t}]\,(\nu_s, \nu_{\curvearrowright}) - d[\mathscr{G}_{s,t}]\,(\nu_s, \nu_{\curvearrowright\!\!\to}) &\leq c\big[\mathscr{U}_i^k\big]\,(\nu_{\curvearrowright\!\!\to}, \nu_*) - c\big[\mathscr{U}_i^k\big]\,(\nu_{\curvearrowright}, \nu_*) & (2)\\
\Leftarrow\ & d[\mathscr{G}_{s,t}]\,(\nu_s, \nu_{\curvearrowright}) - d[\mathscr{G}_{s,t}]\,(\nu_s, \nu_{\curvearrowright\!\!\to}) &\leq d\big[G_i^k\big]\,(v_{\curvearrowright\!\!\to}, v_{\curvearrowright}) & (3)\\
\Leftrightarrow\ & d\big[G_i^k\big]\,(s, v_{\curvearrowright}) - d\big[G_i^k\big]\,(s, v_{\curvearrowright\!\!\to}) &\leq d\big[G_i^k\big]\,(v_{\curvearrowright\!\!\to}, v_{\curvearrowright}) & (4)\\
\Leftrightarrow\ & d\big[G_i^k\big]\,(s, v_{\curvearrowright}) &\leq d\big[G_i^k\big]\,(s, v_{\curvearrowright\!\!\to}) + d\big[G_i^k\big]\,(v_{\curvearrowright\!\!\to}, v_{\curvearrowright}). &
\end{aligned}
$$

Because the triangle inequality always holds for distance functions in graphs, we have proven our claim by that. We explain the logical transitions that have been marked with a number on the right as follows:

1. As π is a shortest path, its subpath $\pi_{1\to(x+1)}$ is also a shortest path. No other path can be shorter.

2. (Left.) The path $\pi_{1\to x}$ is a subpath of a shortest path, which makes it a shortest path, too. The path π' is a shortest path by definition. Hence, we can replace path cost by distance.

   (Right.) The edges $(\nu_{\curvearrowright\!\!\to}, \nu_*)$ and $(\nu_{\curvearrowright}, \nu_*)$ both are contained in the upward graph $\mathscr{U}_i^k$.

3. (Right.) By rewriting the definition of supersedement:

$$d\big[G_i^k\big]\,(v_{\curvearrowright\!\!\to}, v_{\curvearrowright}) \leq c\big[\mathscr{U}_i^k\big]\,(\nu_{\curvearrowright\!\!\to}, \nu_*) - c\big[\mathscr{U}_i^k\big]\,(\nu_{\curvearrowright}, \nu_*).$$

4. (Left.) A shortest path between ν_s and ν_x in $\mathscr{G}_{s,t}$ represents a shortest path between s and $v_{\curvearrowright\!\!\to}$ in G_i^k. The next lemma states that the lengths of both paths match.

□

We provide the next lemma as final brick for the proof of Lemma 21. It is, in a way, similar to our main theorem: We prove that the distance from the source vertex ν_s to any other vertex ν of the search space graph matches the corresponding distance in some component of G. The proof can be found in the appendix.

Lemma 22. *For any pair of vertices $s, t \in V$ with $h(s) \neq h(t)$, and for any vertex $\nu = (v, k+1)$ of $\mathscr{G}_{s,t}$ with $k \geq 0$, the distance between ν_s and ν in $\mathscr{G}_{s,t}$ equals the distance between s and v in $G_{i[k]}^k$:*

$$d[\mathscr{G}_{s,t}]\,(\nu_s, \nu) = d\big[G_{i[k]}^k\big]\,(s, v).$$

We have proven that we can remove any one superseded edge without changing the distance between source and drain. In fact, we can iteratively remove *all* edges that feature a superseding edge. For that, we show that our supersedement relation indeed imposes an order:

Lemma 23. *The relation $\lessdot_L$ is a strict partial order. That is, it is irreflexive and transitive.*

Proof. (Sketch.)

Irreflexivity Both

$$((v_{\curvearrowright}, k), (v_*, k+1)) \lessdot_L ((v_{\curvearrowright\!\!\rightarrow}, k), (v_*, k+1))$$

and

$$((v_{\curvearrowright\!\!\rightarrow}, k), (v_*, k+1)) \lessdot_L ((v_{\curvearrowright}, k), (v_*, k+1))$$

at the same time would imply

$$d\big[G_i^k\big]\,(v_{\curvearrowright\!\!\rightarrow}, v_{\curvearrowright}) = -d\big[G_i^k\big]\,(v_{\curvearrowright}, v_{\curvearrowright\!\!\rightarrow}),$$

from which by non-negativity of edge costs would follow

$$d\big[G_i^k\big]\,(v_{\curvearrowright\!\!\rightarrow}, v_{\curvearrowright}) = 0,$$

a contradiction to the definition of supersedement.

Transitivity Effectively by triangle inequality of the distance in graphs.

□

As our supersedement relation $\lessdot_L$ is a strict partial ordering, we can conclude that each transitive sequence has an endpoint. If we repeatedly apply Lemma 21, such an endpoint edge can never be removed. By that, for each candidate edge that has a superseding edge, there always exists a superseding edge that will remain in the upward graph and that we can use for Lemma 21 at any time in the process. Because no edge removal forbids other edges to be removed that could have been before, we can remove superseded edges in arbitrary order. This is what Algorithm 4 does:

Algorithm 4: Removal of superseded edges.

```
1 forall drain vertices ν_* of 𝒰_i^k do
2     forall source vertices ν_↷ of 𝒰_i^k do
3         forall source vertices ν_↷→ of 𝒰_i^k do
4             if (ν_↷, ν_*) ⋖_L (ν_↷→, ν_*) then
5                 c[𝒰_i^k] (ν_↷→, ν_*) ← ∞
```

Symmetrically, we can provide a similar supersedement relation for downward graphs:

Definition 24 (Supersedement in downward graphs). In a downward graph $\mathscr{D}_i^k$, an edge $(\nu_*, \nu_{\curvearrowright})$ supersedes another edge $(\nu_*, \nu_{\curvearrowright\!\!\rightarrow})$ iff the following two conditions are fulfilled:

$$d\big[G_i^k\big]\,(v_{\curvearrowright}, v_{\curvearrowright\!\!\rightarrow}) > 0$$

$$c\big[\mathscr{D}_i^k\big]\,(\nu_*, \nu_{\curvearrowright}) + d\big[G_i^k\big]\,(v_{\curvearrowright}, v_{\curvearrowright\!\!\rightarrow}) \leq c\big[\mathscr{D}_i^k\big]\,(\nu_* k + 1, \nu_{\curvearrowright\!\!\rightarrow} k).$$

In this case, we write

$$(\nu_*, \nu_{\curvearrowright}) \lessdot_R (\nu_*, \nu_{\curvearrowright\!\!\rightarrow}).$$

The relation $\lessdot_R$ specifies removable edges in downward graphs just as $\lessdot_L$ did for upward graphs. The proof for that would be completely symmetrical to the one for the upward graphs. For the sake of completeness, we present Algorithm 5 that performs removal of superseded edges.

Algorithm 5: Removal of superseded edges for downward graphs.

```
1 forall source vertices ν_* of D_i^k do
2     forall drain vertices ν_↷ of D_i^k do
3         forall drain vertices ν_↷→ of D_i^k do
4             if (ν_*, ν_↷) ⋖_R (ν_*, ν_↷→) then
5                 c[D_i^k](ν_*, ν_↷→) ← ∞
```

Finally, for a level graph $\mathscr{L}_{i,j}^k$, we can use *both* supersedement relations to check if we can omit an edge. The intuition behind that is, that a shortest path can either immediately re-enter the source component G_i^k and leave it via another boundary vertex on its way to G_j^k, or that it can enter the drain component G_j^k via another boundary vertex and reach the target from inside G_j^k, or even both. By subsequential execution of both Algorithms 4 and 5 we can remove all superseded edges in a level graph. The proof for that is essentially a repetition of the proof for upward and downward graphs: For level graphs, both source and drain vertex are connected to an upward or a downward graph, respectively.

For a search space part $\mathscr{P}$, let $|\Sigma|$ and $|\Delta|$ denote the number of source and drain vertices. As each algorithm features a triply nested loop, we can state the following run times for removing the superseded edges:

Upward graphs $O(|\Sigma|^2 \cdot |\Delta|)$

Downward graphs $O(|\Sigma| \cdot |\Delta|^2)$

Level graphs $O(|\Sigma| \cdot |\Delta| \cdot (|\Sigma| + |\Delta|))$

In practice, the values of $|\Sigma|$ and $|\Delta|$ are usually below 100, so we can afford a cubic algorithm here. Any optimal algorithm for this problem needs to check at least once each edge of the search space part, which, in general, requires

$$O(|\Sigma| \cdot |\Delta|)$$

run time.

4.2 Construction of Equivalent Graphs

A search space part has been defined as a complete directed bipartite graph. Recall that in our query algorithm (Algorithm 1) we used the search space parts as parameters to the procedure `update` that updated the distance labels for all drain vertices. We are free to modify the implementation of `update`, provided that its functionality is preserved.

In particular, we can replace a search space part by an equivalent DAG. Here, equivalence means that the substitute has at least the same set of source and drain vertices, plus an arbitrary number of new, unique vertices, and that the distance between each pair of a source and a drain vertex matches the cost of the corresponding edge in the search space part. By reducing the number of edges used, we reduce both storage space for this search space part and query time for the search space graphs this search space part is used in. We use the term *minimization* to denote such a replacement in general.

First of all, we shall show why we expect a minimization to result in a reduction of the number of edges if road map graphs are used as input graph. The edges in a search space part represent shortest paths between boundary vertices in the input graph. These shortest paths cannot be disjoint for a planar input graph if both components have more than two boundary vertices. (If the paths were disjoint, the graph formed of these paths, a subgraph of the input graph, could be reduced to the Kuratowski graph $K_{3,3}$, in contradiction to planarity.) Moreover, many shortest paths may share a common vertex — the further the components are from each other, the more probable this becomes. Even if not all paths share a common vertex, often we can find an equivalent graph with considerably fewer edges than the complete bipartite graph. Figure 10 illustrates how the graph formed by unifying shortest paths between source and drain vertices can be shrunken to yield an equivalent graph with fewer edges. We refer to this minimization technique as *path overlay*. — As road map graphs can be considered "almost planar", above applies to them to a very high degree, too.

For the ideal case, imagine a decomposition of the road map graph of Europe where both London and Paris result in a top-level component. For a query from a point in London to a point in Paris, the search space graph would include the level graph between London and Paris. Now, any shortest path from any point in London to any point in Paris most probably crosses the English Channel via ferry between Dover and Calais. To render all distances between all pairs of boundary vertices, it is sufficient to keep the distance from all boundary vertices of London to Calais, and from Calais to all boundary vertices of Paris, instead of keeping all distances between all pairs. If many boundary vertices are affected, the saving is enormous.

In this work, a source-drain graph that has an equivalent directed star graph is called *perfectly minimizable*. For our example, the level graph between London and Paris has this property. In Figure 11, we show an example of a better optimization than in Figure 10 and identify a perfectly minimizable subgraph.

For the minimization problem, we offer a heuristic approach called *star minimization* that uses weighted bipartite graphs as input. By that, we can optimize even if we do not know the structure of the paths behind a search space part, as opposed to the path

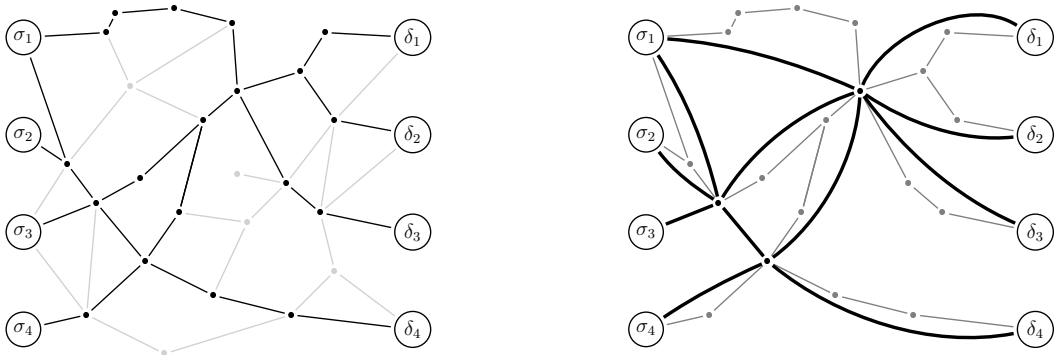

Figure 10: Minimization by path overlay (1).
The drawing at the left shows a directed weighted graph. The highlighted edges are contained in at least one shortest path between any source vertex at the left to any drain vertex at the right.
In the right-hand side drawing, the shaded edges are those that participate in a shortest path. The other edges have been omitted. The bold curved edges are the result of the contraction of all vertices with indegree or outdegree of one; the distance between any source and any drain in the graph made of the curved edges matches that in the original graph. It contains 12 edges, which is four fewer than the number of distinct pairs of source and drain vertices.

overlay technique. The algorithm is robust and comparatively simple. Again, we denote missing edges by infinite edge costs. The heuristics performs well in practice, and we propose extensions and further options. As for our speed-up technique, the minimization constitutes an optimization and not a requirement, we are content with a potentially suboptimal solution here.

In the remainder of the current subsection, we present our heuristic approach by providing a description and pseudocode. We also analyze briefly the run time requirement for our heuristics. At the end, in Figure 12, we offer a visual description for the most important stages of the heuristics.

Unlike the optimization by supersedement shown in the previous subsection, we do not need to consider the context a search space part is used in. Owing to that, in this subsection we examine directed bipartite graphs with vertex set $V := \Sigma \cup \Delta$ for disjoint Σ and Δ. We call Σ the *source vertex set* and Δ the *drain vertex set* and allow only edges from a source to a drain vertex. We use the term *bipartite source-drain graph* to denote such a graph.

Our heuristics tries to minimize the number of edges by introducing exactly *one* additional vertex, called *center vertex* or *center* and denoted by ζ. We add edges to and from the central vertex in hope that many edges between a source and a drain vertex have the same length as a path via the central vertex and thus can be removed afterwards. The newly added edges are denoted by the term *central edge*. In contrast, the edges of the complete bipartite graph are referred to as *original edges* or *source-drain edges*.

Obviously, we may not insert central edges that result in a shorter path than before between any source and drain vertex. However, we allow central edges to have negative

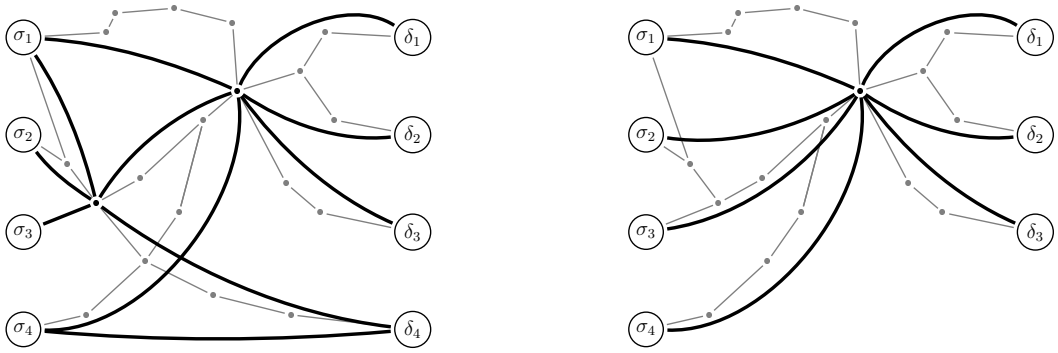

Figure 11: Minimization by path overlay (2).
The contracted graph from Figure 10 at the right is not the minimum equivalent graph in terms of number of edges: The drawing at the left shows an equivalent graph with 11 edges. In this special case, we can save one edge by contracting a vertex of both indegree and outdegree two.
Observe that if the drain vertex δ_4 is not considered, the graph is perfectly minimizable. This is shown in the drawing at the right.

costs. By the insertion of central edges, a source-drain edge becomes *obsolete* iff its cost equals the length of a shortest path via center. Obsolete edges can be removed without affecting the distance between any source vertex and any drain vertex.

Before presenting the pseudocode, we shall introduce one of the most important concepts in our heuristics. The *gain* from the central edges is defined to be the number of obsolete edges minus the number of central edges. We try to construct a cost function for the central edges that maximizes the gain. We foreclose that, in our pseudocode, for a given set of source and drain vertices and a given cost function, the function `computeGain` returns the gain defined above. Later, we present the pseudocode for this function and explain it.

Algorithm 6 presents a pseudocode for our heuristic approach. We shall describe it in detail. In Figure 12, we provide an example for the most important stages of the algorithm.

First, we add the central vertex ζ to our graph. The main loop that starts in line 2 iterates over all source vertices. We call ρ the *root vertex* of each iteration. (Each source vertex is treated exactly once as root vertex.) Then, the root vertex is connected to the central vertex with an edge of cost zero. In turn, the central vertex gets connected to all drain vertices with costs that match the cost of the original edge from root to the corresponding drain. We use the term *root edge* to denote an edge added in this initialization phase, i.e., an edge from root to center or from center to any drain. The costs of the root edges remain unchanged for a fixed root vertex. For each root vertex, we use a distinct drain vertex set Δ_σ, as we intend to exclude, for each source vertex, a potentially different set of gain-reducing drain vertices.

Due to the root edges, we can omit all edges from root to any drain. Yet no gain is achieved by that transformation — on the contrary, the graph contains one additional edge, the edge from root to center. (The function `computeGain` would return -1.)

Algorithm 6: Star minimization of a source-drain graph.

```
1  add a new vertex ζ as central vertex
2  forall source vertices ρ ∈ Σ (root vertex) do
3      c(ρ, ζ) ← 0
4      forall drain vertices δ ∈ Δ do  c(ζ, δ) ← c(ρ, δ)
5      g_old ← alterCostFunction(ρ, ζ, Σ, Δ, c)
6      Δ_ρ ← Δ
7      repeat
8          g_best ← −∞
9          forall drain vertices δ′ ∈ Δ_ρ do
10             g_cnt ← alterCostFunction(ρ, ζ, Σ, Δ_ρ \ {δ′}, c)
11             if g_best < g_cnt then
12                 g_best ← g_cnt
13                 δ_best ← δ′
14         if g_best >= g_old then
15             Δ_ρ ← Δ_ρ \ {δ_best}
16             c(ζ, δ_best) ← ∞
17             g_old ← g_best
18     until g_best < g_old
19     memorize g_best and Δ_ρ
20 return of all root vertices, the minimized graph with the best gain
```

The root vertex, the central vertex and the source and drain vertex sets are passed to `alterCostFunction` function as arguments. This function finds, for a given set of drain vertices, a cost function that leads to the maximal gain under the restriction that costs for root edges are not modified. We shall explain it in detail before returning to the description of the algorithm.

In the function `alterCostFunction`, for all source vertices σ except the root vertex, the minimum length of an edge to the central vertex is computed and assigned to the cost function c. Recall that we can add a central edge only if it does not shorten existing paths. As we do not alter costs for root edges, this imposes a minimum length for an edge from a source vertex to a central vertex, and we use this minimum for each new central edge. The minimum is stored in c_{min}, it is initialized with $-\infty$. For each drain vertex, we check if it requires the cost of the current edge from source to center to be higher, and adjust it accordingly. Note that we use a local set of drain vertices that does not necessarily contain all drain vertices from our source-drain graph: In subsequent passes, we remove drain vertices that lead to a low gain. If, after considering all drain vertices, the resulting minimum is ∞, we effectively add an edge of infinite cost, i.e., we omit this central edge. Conversely, for $-\infty$, the edge cost does not matter, and we can skip the edge and also set its cost to ∞. Finally, as a courtesy to the caller, the function returns the gain that is

Function `alterCostFunction`(ρ, ζ, Σ, Δ, c)

```
1  forall source vertices σ ∈ Σ \ {ρ} do
2      c_min ← −∞
3      forall drain vertices δ ∈ Δ do
4          if c(σ, δ) ≠ ∞ ∨ c(ρ, δ) ≠ ∞ then
5              c_cnt ← c(σ, δ) − c(ρ, δ)
6              if c_min < c_cnt then
7                  c_min ← c_cnt
8      if c_min = −∞ then
9          c_min ← ∞
10     c(σ, ζ) ← c_min
11 return computeGain(ζ, Σ, Δ, c)
```

obtained for the modified cost function.

The function `alterCostFunction` generates the cost function with the best gain possible for the provided set of drain vertices under the restriction that no root edge is modified. If any of the altered edges from source to center had a lower cost than computed by `alterCostFunction`, there would be a source-drain edge with costs higher than the corresponding path via center.

Now, we analyze how our minimization algorithm employs `alterCostFunction`. First, the cost function for the unchanged drain vertex set Δ is computed in line 5. Between lines 9 and 13, the algorithm computes, for all drain vertices, the gain that would result if this drain vertex was removed. If we can, by removing a drain vertex, achieve a better gain, in lines 14-17, the drain vertex whose removal induces the best gain is finally removed from our set of drain vertices Δ_ρ, and the next iteration starts where the algorithm looks for yet another drain vertex to be removed. If no better gain is possible by removing a single drain vertex, the algorithm memorizes the set of drain vertices that lead to the best gain for this root vertex, and continues by using the next source vertex as root, if possible.

Finally, after all source vertices have been processed as root vertex, we use the root vertex that produced the globally best gain and construct the corresponding minimized graph. Essentially, the construction is performed like in the function `alterCostFunction`, except that, in addition, edges with matching lengths are removed.

At last, we shall present the function `computeGain`. The function starts with a gain of zero, stored in the variable g. In the lines 1-5, for each pair of source and drain vertex, the cost of the source-drain edge is compared to the length of the path via the central vertex. If the edge cost matches the length of the path, we can later omit the source-drain edge, and thus increment the gain. Between line 6 and 9, we subtract, from the gain, the number of newly added edges to and from the central vertex. Ultimately, we return the gain that is stored in g.

We cannot guarantee optimality for the solution returned by the algorithm. However,

Function `computeGain`(ζ, Σ, Δ, c)

```
1  g ← 0
2  forall source vertices σ ∈ Σ do
3      forall drain vertices δ ∈ Δ do
4          if (σ, δ) ≠ ∞ then
5              if c(σ, ζ) + (ζ, δ) = (σ, δ) then g ← g + 1
6  forall source vertices σ ∈ Σ do
7      if c(σ, ζ) ≠ ∞ then g ← g − 1
8  forall drain vertices δ ∈ Δ do
9      if c(ζ, δ) ≠ ∞ then g ← g − 1
10 return g
```

if the graph is perfectly minimizable, our algorithm finds the best solution.

The algorithm in its pure form needs

$$O(|\Sigma|^2 \cdot |\Delta|^3)$$

time to complete in the worst case: For each of the $|\Sigma|$ source vertices, we need at most $|\Delta|$ iterations of the repeat loop that starts at line 7, each requiring $O(|\Delta|)$ calls to `alterCostFunction`. In turn, `alterCostFunction` needs $O(|\Sigma| \cdot |\Delta|)$ run time, as it includes a nested loop for all pairs of source and drain vertices. The function `computeGain`, called by `alterCostFunction`, requires asymptotically the same time for similar reasons.

For our implementation, we have reduced the worst-case run time to

$$O(|\Sigma|^2 \cdot |\Delta|^2)$$

with a modification to `alterCostFunction`: We simultaneously compute, for each drain vertex, the gain that results in its removal. Another optimization that we have implemented is, that the algorithm quits immediately once a perfect minimization has been found. As the source and drain vertex sets usually contain no more than 100 vertices, in total, we can afford to employ this algorithm.

Of course, we are not obliged to perform the star optimization if it results in a negative gain. This is especially true if we have only one source or only one drain vertex. In this case, we simply skip this optimization step.

As the graph produced by the star optimization is a DAG, we can afford negative edge costs without possible penalties for the query time. This is essential, as we might be able to produce a better gain with negative edges allowed. Recall from the previous section that we used the procedure `update` to update the distance labels of all drain vertices of a search space part, once the distance labels to all source vertices have been computed. We need only a slight modification to the `update` procedure: After all edges that originate in a source vertex have been visited, we visit all edges that originate in the central vertex. For the sake of completeness, we present a revision of the `update` procedure called `updateStar`.

Procedure `updateStar(`$\mathscr{P}$`, `ζ`)`

```
1 forall source vertices ν1 of 𝒫 do
2     foreach drain vertex ν2 of 𝒫 with c[𝒫](ν1, ν2) < ∞ do
3         relax(ν1, ν2)
4     relax(ν1, ζ)
5 foreach drain vertex ν2 of 𝒫 with c[𝒫](ζ, ν2) < ∞ do
6     relax(ζ, ν2)
```

Another possible argumentation is that the central vertices perfectly fit in our topological sort $\triangleleft$ for the vertices of a search space graph, and that a search space graph composed of star-minimized search space parts remains a DAG.

The star optimization can be easily combined with the removal of superseded edges: Both optimizations remove source-drain edges, and only the star optimization adds new vertices and edges. By that, we can remove a source-drain edge either if it is superseded or obsoleted by a path via center. We can fine-tune the combination by improving `computeGain` so that only those edges are counted as obsolete that have not been already superseded.

Further enhancements of our heuristics are possible, but were beyond the scope of this work. For instance, we could allow more than one central vertex. Also, minimization by path overlay as in Figures 10 and 11 could produce good results.

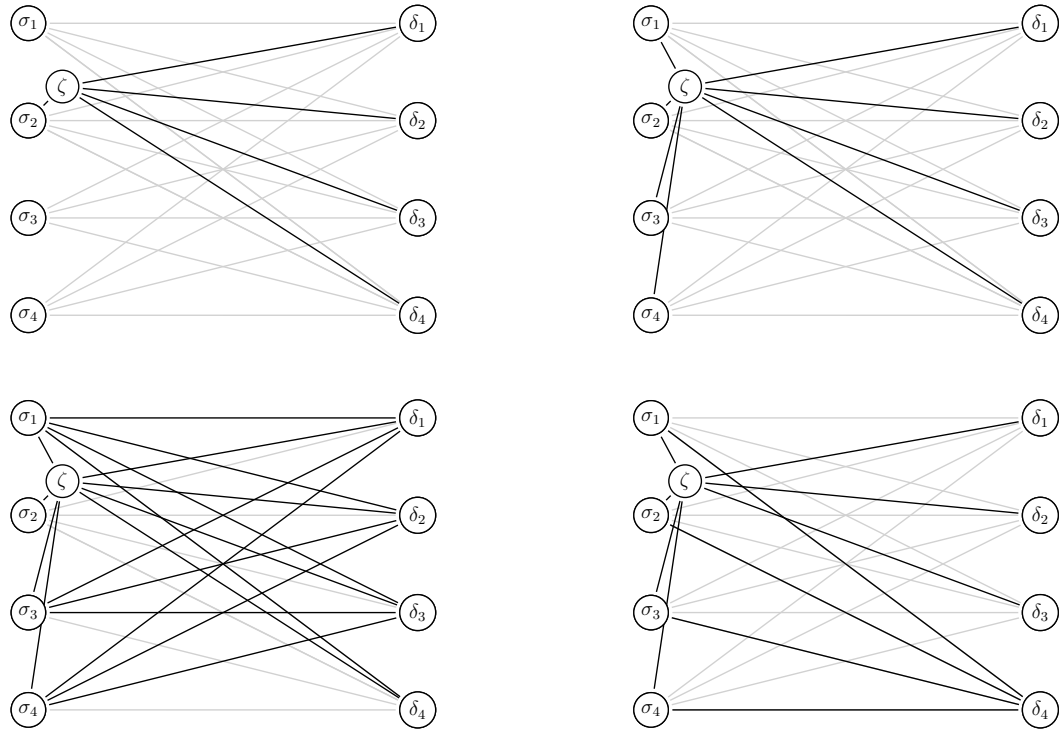

Figure 12: Star minimization of a source-drain graph.

In this figure, we use the bidirected source-drain graph induced by the distances between source and drain vertices in Figures 10 and 11.

The drawing at the top left shows the result of the initialization phase (lines 3 to 4 of Algorithm 6) for σ_2 as root vertex. The cost of the edge from σ_2 to the central vertex ζ equals zero, the cost of each edge from the central vertex to any drain vertex equals the cost of the edge from the root vertex to this drain vertex. The highlighted edges represent the newly added edges, their cost is fixed for this root vertex.

The drawing at the top right shows the edges added by the function `alterCostFunction` in lines 1 to 9. The costs of the new edges are minimal under the restriction that no path from any source to any drain vertex via the central vertex is shorter than the corresponding edge.

In the two drawings at the bottom, we present a hypothetical example. We cannot conclude which edges need to remain in the graph without specifying edge costs. Instead, we assume that edge costs have been chosen so that above situation arises. (Assume that the distance between σ_2 and δ_4 is large compared to the distance from σ_2 to all other drain vertices.)

The drawing at the bottom left shows a source-drain graph after the first execution of `alterCostFunction` in line 5. The shaded edges are obsolete. Note that the gain is negative here: We need eight new central edges, but only six source-drain edges become obsolete.

In the right-hand side drawing, the same is shown if δ_4 is deleted from the set of drain vertices Δ_ρ for this root vertex. Now, we need to keep all edges to δ_4, as this vertex cannot be reached via the center anymore. However, many other edges become obsolete now: We achieve a total gain of 5, which is as good as with the sophisticated contraction presented in the left-hand side drawing of Figure 11.

5 Preprocessing Algorithm

In the previous two sections, we have shown how our speed-up technique works in general and presented options for further optimization. The current section is devoted mainly to the preprocessing procedure. We show in detail how to produce the preprocessing information for a given input graph, and prove the correctness of our algorithm. Furthermore, we investigate the data flow of our algorithm and provide a parallelization scheme. The description of both algorithm and parallelization scheme highly conforms to the actual implementation. We use a given hierarchical decomposition of an input graph to produce unoptimized search space parts, i.e., complete bipartite graph. These can be subsequently optimized, as shown in the previous section.

Recall that we used search space parts of different kinds to compose a search space graph. To be able to answer a shortest-path query for an arbitrary pair of vertices, we need to precompute, optimize and store all search space parts that possibly might be required. In total, we need the following search space parts:

- We need to be able to get from any vertex in the input graph to its boundary vertices, and back. That is, for each vertex $v \in V$, we need the entry and exit graphs $\mathscr{E}_v$ and $\mathscr{X}_v$.
- Once we are at a boundary vertex, we may need to move to boundary vertices at higher levels, and back. In other words, for each $k \in \{0, \ldots, n-2\}$ and for each $i \in I^k$, we need the upward and downward graphs $\mathscr{U}_i^k$ and $\mathscr{D}_i^k$.
- Finally, we connect boundary vertices of components at the same level that share their parent component. To put it another way, we need, for each $k \in \{0, \ldots, n-1\}$ and for each $i, j \in I^k$ with $i \neq j$ and $f^k(i) = f^k(j)$, the level graph $\mathscr{L}_{i,j}^k$.

Note that, for $n = 1$, we do not use any upward or downward search space parts.

It is very expensive, in terms of run time, to compute all search space parts naïvely, through multiple runs of DIJKSTRA's single-source algorithm: Potentially, each source vertex of each level graph requires all vertices and edges of the input graph to be considered. Fortunately, we can use the properties of the hierarchical decomposition to design an efficient preprocessing algorithm.

Here, we assume that the graph already has been decomposed hierarchically, i.e., that a hierarchical decomposition according to Section 3.3 has been provided. Apart from that, the preprocessing consists of two stages:

Distances between separators For each component at each level, we compute and store the distance between all pairs of vertices that are separators one level beneath. This preprocessing stage serves as a base for the actual computation of the search space parts.

Search space parts Once all of the above distances have been computed, we can fairly easily extract all entry, exit, upward and downward search space parts from this information, and need only a little more work to compute all level graphs.

We present, in detail, each preprocessing stage. Finally, we explain how the algorithm can be parallelized.

5.1 Child Separator Subset Closure

The goal of this preprocessing stage is to determine, for each parent component in the hierarchy, the distances between the boundary vertices of the child components. By design, we do not take into account paths that leave the parent component here.

To formalize this, we define for $k \in \{1, \ldots, n\}$ the set of *child separators* C_i^k — vertices of the component G_i^k that are separators one level below:

$$C_i^k = V_i^k \cap S^{k-1}.$$

From Lemma 8, we immediately derive $B_i^k \subseteq C_i^k$. Furthermore, the equality $C_0^n = S^{n-1}$ holds: The child separators of the universe are exactly the top-level separators.

Now we can formulate our problem as follows: Compute the subset closure

$$G_i^k(V_i^k, c_i^k)\big|_{C_i^k}$$

for each $k \in \{1, \ldots, n\}$ and $i \in I^k$. We use the term *child separator subset closure* (in short, *child separator closure*) to denote this special subset closure.

For $k = n$, above formula converts into

$$G(V, c)\big|_{S^{n-1}}.$$

That is, we compute the distance between all top-level separator vertices for this choice of k.

In the following proofs, we often use another special subset closure, the one created by restricting a component to its boundary vertices:

$$G_i^k(V_i^k, c_i^k)\big|_{B_i^k}.$$

We call this the *boundary closure* of a component G_i^k.

To compute the child separator subset closures, we employ a bottom-up approach, bearing in mind that subpaths of shortest paths are shortest paths themselves. That is, for $k > 1$, we use the already computed closures at level $k - 1$ to compute the closures at level k. The initial source for the child separator closures at level 1 are the components at the lowest level. Algorithm 10 outlines the basic procedure.

This algorithm saves us from the necessity of keeping the potentially huge input graph in memory at the time of preprocessing. In turn, this improves the benefit from a parallelization of the preprocessing.

We prove the next lemma to specify more precisely the informal algorithm mentioned above. Figure 13 provides an intuitive description for the next lemma. From the lemma, we also conclude the algorithm's correctness in the forthcoming corollary. Finally, we specify the same algorithm more precisely by using actual operations in place of the loose textual representation.

Algorithm 10: Computation of child separator closures (brief)

```
forall i ∈ I^1 do
    compute G_i^1|_{C_i^1} from the child components of G_i^1
for k ← 2 to n do
    forall i ∈ I^k do
        join all G_{i'}^{k-1}|_{C_{i'}^{k-1}} with i' ∈ H_i^k
        compute G_i^k|_{C_i^k} from the join
```

Lemma 25. *For all $k \in \{1, \dots, n\}$ and for all $i \in I^k$, define the* boundary closure union *as the graph union of all boundary closures for all child components of G_i^k:*

$$\bar{G}_i^k := \bigcup_{i' \in H_i^k} \left(G_{i'}^{k-1}\big|_{B_{i'}^{k-1}} \right).$$

Then, for any pair of source and target vertices $s, t \in C_i^k$, the distance in the component G_i^k can be computed from that in the boundary closure union $\bar{G}_i^k$:

$$d\big[G_i^k\big](s,t) = d\big[\bar{G}_i^k\big](s,t).$$

Proof. (Sketch) Fix arbitrary $s, t \in C_i^k$. We prove this lemma by showing that there exists, for each path between s and t, a corresponding path in $\bar{G}_i^k$, also between s and t, with at most the same length; and vice versa. Then, we can conclude that the distance must be the same in both graphs. This technique is similar to that used in the proof of our main theorem.

($\Leftarrow$) The proof for this direction essentially constitutes an edge-wise expansion of an arbitrary path in the boundary closure union $\bar{G}_i^k$ to one in the component G_i^k. This expansion cannot increase path length, as we shall see.

Fix an arbitrary path $\bar{p} = \langle v_1, v_2, \dots, v_z \rangle$ in $\bar{G}_i^k$ with $v_1 = s$ and $v_z = t$. For each $x \in \{1, \dots, z-1\}$, fix a shortest path $p_x := \langle v_x, \dots, v_{x+1} \rangle$ in G_i^k. Such a path must exist, otherwise the edge (v_x, v_{x+1}) would have infinite costs, in contrary to the path property of $\bar{p}$.

We create the corresponding path p in G_i^k by adding, to $\bar{p}$, the intermediate vertices of the path p_x between each pair (v_x, v_{x+1}). The cost of an edge in $\bar{p}$ can be bounded by the length of the corresponding path in G_i^k:

$$c\big[G_i^k\big](p_x) = d\big[G_i^k\big](v_x, v_{x+1}) \leq \min_{i' \in H_i^k} d\big[G_{i'}^{k-1}\big](v_x, v_{x+1}) = c\big[\bar{G}_i^k\big](v_x, v_{x+1}).$$

Intuitively, a hop in $\bar{p}$ corresponds to multiple hops in p, the total cost of those hops in p matches the cost of the single hop in $\bar{p}$. As this inequality holds for each edge of $\bar{p}$, our claim is satisfied.

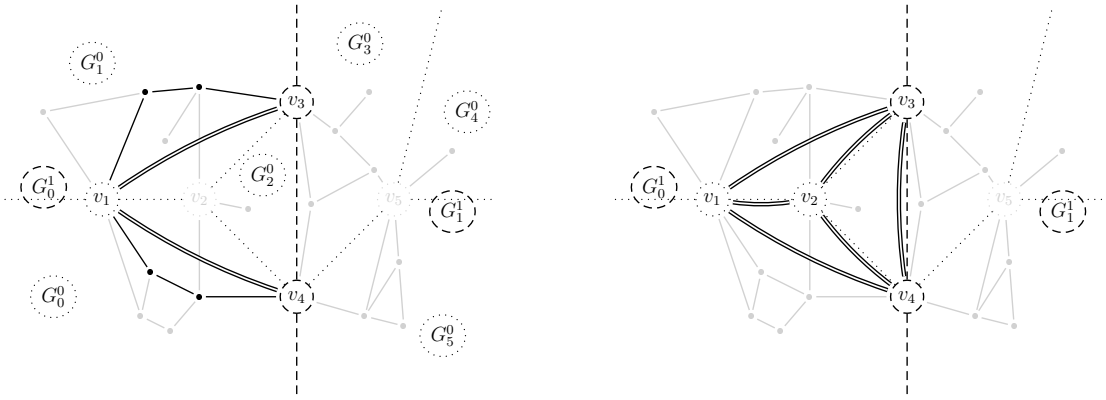

Figure 13: Child separator closure graph.
In the left-hand side drawing, a path p between v_3 and v_4 in G_0^1 is highlighted. The corresponding path $\bar{p}$ in $\bar{G}_0^1$ is denoted by double-lined edges, which themselves represent shortest paths in different components: The edge (v_3, v_1) represents a shortest path in G_1^0, whereas the edge (v_1, v_4) stands for a shortest path in G_0^0.
For the proof of Lemma 25, observe the following: If $\bar{p}$ was a shortest path in the boundary closure union $\bar{G}_0^1$, its edges would correspond to a shortest path in G_0^1. Conversely, if p was a shortest path in G_0^1, its length would be the same as the length of $\bar{p}$.
The double-lined edges in the drawing on the right represent all edges of the child separator closure $G_0^1\big|_{C_0^1}$. All pairs of vertices in $C_0^1 = \{v_1, v_2, v_3, v_4\}$ are connected in this subset closure. Edge costs denote lengths of shortest paths in G_0^1. Corollary 26 states that we can compute this closure from the corresponding boundary closure union $\bar{G}_0^1 = G_0^0\big|_{B_0^0} \cup G_1^0\big|_{B_1^0}$.

($\Rightarrow$) For this direction, we perform a path contraction for a path in G_i^k, leaving only the vertices that are contained in the boundary closure union $\bar{G}_i^k$. We show that this contraction does not increase the length of the path.

Fix an arbitrary path $p = \langle v_1, v_2, \ldots, v_z \rangle$ in G_i^k with $v_1 = s$ and $v_z = t$. The corresponding path $\bar{p}$ in $\bar{G}_i^k$ is obtained by removing all vertices from p that are not in C_i^k. (The vertices s and t remain in the path, since they are in C_i^k by definition.) This is not exactly the inverse operation to ($\Leftarrow$), since the removed vertex sequences do not necessarily represent the intermediate vertices of shortest paths in G_i^k. However, for two vertices $v_x, v_y \in C_i^k$ of p with y smallest so that $y > x$, the path $p_{x \to y}$ entirely lies in one child component of G_i^k. As edges in $\bar{G}_i^k$ represent shortest paths between boundary vertices of child components, we can bound the length of each subpath by the length of the corresponding edge in $\bar{G}_i^k$. Therefore, the length of $\bar{p}$ is not greater than the length of p, according to our claim. □

The next corollary presents a closed notation of Lemma 25: We provide an identity for computing a child separator closure from a boundary closure union.

Corollary 26. *For all $k \in \{1, \ldots, n\}$ and for all $i \in I^k$, we can compute the child separator closure of a component G_i^k by computing a subset closure of the boundary closure union $\bar{G}_i^k$:*

$$G_i^k\big|_{C_i^k} = \bar{G}_i^k\big|_{C_i^k}.$$

Proof. Apply Lemma 25 to all pairs of vertices from $C_i^k \times C_i^k$. □

All boundary vertices are also child separators: $B_i^k \subseteq C_i^k$. Thus, we can immediately conclude by the definition of the subset closure, that a boundary closure is a subgraph of the corresponding child separator closure:

$$G_i^k\big|_{B_i^k} \subseteq G_i^k\big|_{C_i^k}.$$

This allows us to compute any requested boundary closure $G_i^k\big|_{B_i^k}$ simply by using the child separator closure $G_i^k\big|_{C_i^k}$ minus vertices not in B_i^k, i.e., the non-boundary vertices.

Before presenting the precise formulation for the algorithm, we anticipate that we will also need certain directed subset closures at level 0 to be able to easily extract the entry and exit graphs. Namely, we need to compute, for all $i \in I^0$,

$$G_i^0\big|_{V_i \to B_i^0}$$

(for the entry graphs) and

$$G_i^0\big|_{B_i^0 \to V_i}$$

(for the exit graphs). For both of above special directed subset closures, we use the term *elevating closure*.

As a boundary vertex for a component is part of this component, i.e., $B_i^0 \subseteq V_i$, we also know that the boundary closure is contained in each of the elevating closures:

$$G_i^0\big|_{B_i^0} \subseteq G_i^0\big|_{V_i \to B_i^0} \text{ and } G_i^0\big|_{B_i^0} \subseteq G_i^0\big|_{B_i^0 \to V_i}.$$

In analogy to the last paragraph, to obtain the boundary closure $G_i^0|_{B_i^0}$, we use an elevating graph and skip all non-boundary vertices, i.e., all vertices that are not in B_i^0.

Algorithm 11 provides a detailed description for this preprocessing stage. By Corollary 26, we have proven its correctness. We considerably speed up the computation of the child separator closures compared to the straightforward approach, since the boundary closure unions $\bar{G}_i^k$ have much less vertices than the original component graphs G_i^k.

5.2 Search Space Parts

With the computation of the child separator subset closures from the previous subsection, we are now ready to derive the search space part graphs. As mentioned already, for each entry, exit, upward and downward graph, the edge lengths are already present in some child separator closure and only need to be extracted. This is possible, as we do not take into account paths outside the upper-level component for these kinds of part graph. We briefly state, for each of above kinds of search space part, where to obtain the edge lengths from, before explaining the more complicated computation of level graphs.

Algorithm 11: Computation of child separator closures (detailed)

```
forall i ∈ I^0 do
    compute the elevating closures G_i^0|_{V_i→B_i^0} and G_i^0|_{B_i^0→V_i} using DIJKSTRA's single-source algorithm
    extract G_i^0|_{B_i^0} from the elevating closure G_i^0|_{V_i→B_i^0}
for k ← 1 to n do
    forall i ∈ I^k do
        Ḡ_i^k ← ∅
        forall i' ∈ H_i^k do
            Ḡ_i^k ← Ḡ_i^k ∪ G_{i'}^{k-1}|_{B_{i'}^{k-1}}
        compute Ḡ_i^k|_{C_i^k} using DIJKSTRA's single-source algorithm
        G_i^k|_{C_i^k} ← Ḡ_i^k|_{C_i^k}
        if k < n then
            extract G_i^k|_{B_i^k} from G_i^k|_{C_i^k}
```

Entry and exit graphs For all $s \in V$ and for $i := h(s)$, the next equation follows instantly from the definition of the subset closure:

$$\forall v \in B_i^0 : d\big[G_i^0\big]\,(s,v) = c\Big[G_i^0\big|_{V_i \to B_i^0}\Big]\,(s,v).$$

Recall from the definition of the entry graph that these were the distances used to construct $\mathscr{E}_s$. Antisymmetrically, for all $t \in V$ and for $i := h(t)$, the following equation holds:

$$\forall v \in B_i^0 : d\big[G_i^0\big]\,(v,t) = c\Big[G_i^0\big|_{B_i^0 \to V_i}\Big]\,(v,t).$$

These are precisely the distances the exit graph $\mathscr{X}_t$ can be constructed from.

Hence, we can immediately look up the edge lengths for the entry and exit graphs from the directed subset closures of G_i^0 that also are computed by Algorithm 11.

Upward and downward graphs For $l := f^k(i)$, the following two identities result from the definition of the subset closure, too:

$$\forall (s,t) \in B_i^k \times B_l^{k+1} : d\big[G_l^{k+1}\big]\,(s,t) = c\Big[G_l^{k+1}\big|_{C_l^{k+1}}\Big]\,(s,t)$$
$$\forall (s,t) \in B_l^{k+1} \times B_i^k : d\big[G_l^{k+1}\big]\,(s,t) = c\Big[G_l^{k+1}\big|_{C_l^{k+1}}\Big]\,(s,t)$$

By definition of the upward and downward graphs, these are exactly the distances that need to be computed for these kinds of part graph.

Thus, the upward and downward graphs can also be extracted as soon as Algorithm 11 has computed the corresponding child separator closure.

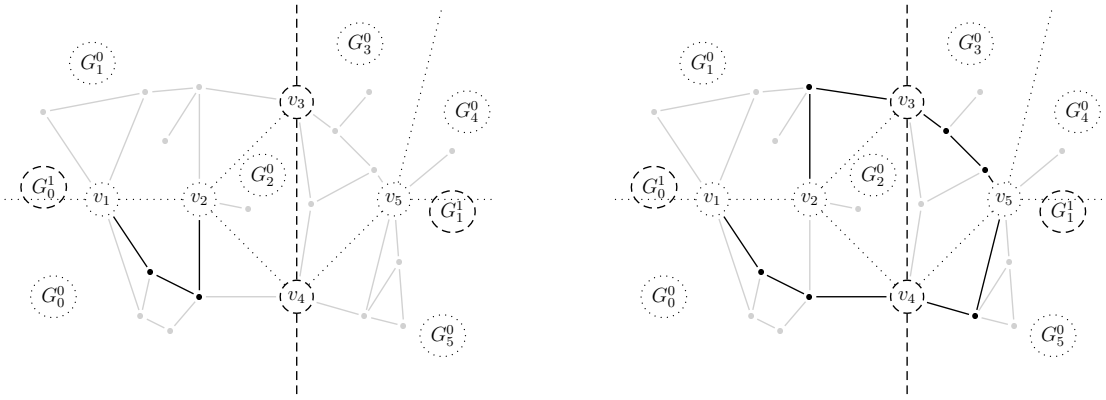

Figure 14: Two possible paths between child separators.
A path between v_1 and v_2 can either stay in the common parent component G_0^1, or leave it. The drawings above show an example for both cases. In the second case, the component G_0^1 may be left and re-entered only via its boundary vertices. Thus, any leaving and re-entering path must contain at least one boundary vertex of G_0^1, in our example, v_3 or v_4. (In fact, any path that truly leaves G_0^1 must contain at least two boundary vertices. We make do with the weaker condition that one vertex must be contained here, as it is still a necessary condition for a path to leave the common parent component.)

Level graphs Computation of the level graphs is not as easy, because paths that leave the parent component need to be incorporated, and such paths are not considered in the computation of the child separator closures. Fortunately, we can use an approach similar to the one in Section 5.1 to avoid scanning the whole input graph for shortest paths outside a component.

For this, we analyze the possible courses of a shortest path that starts in the component G_i^k and ends in the component G_j^k. Here, we are interested in the *globally shortest path*, i.e., in the shortest of all paths in the whole input graph G; the length of such a path is referred to as *global distance*. As in the definition of the level graph, both components share a common parent component G_l^{k+1} with $l = f^k(i) = f^k(j)$, but the path may exit and re-enter the parent component. We observe that either of the following may apply:

1. The shortest path entirely lies in the common parent component G_l^{k+1}. For being a shortest path, its cost matches both the global distance and the distance in G_l^{k+1}. In turn, the distance in G_l^{k+1} can be looked up in the child separator subset closure for G_l^{k+1}, which has been computed by Algorithm 11.

2. The shortest path leaves the parent component. That given, the distance in G_l^{k+1} may be longer than the distance in the input graph G. Such a path must visit at least one of the boundary vertices of the common parent component G_l^{k+1}. By that, there must be, in the sequence of the vertices in the path, a first and a last boundary vertex. We can split the path in three parts, either of which may be empty: From the source to the first boundary vertex, from the first to the last boundary vertex, and from the last boundary vertex to the target. Again, the lengths of the first and third subpath match the distance in the parent component G_l^{k+1} and can be obtained

from the child separator closure for G_l^{k+1} computed by Algorithm 11. However, in general, the cost of the second subpath is lower than the length of the corresponding edge in the child separator closure.

Figure 14 presents an example for both cases. Note that the second case is possible only for $k < n-1$, as top-level components share the universe as parent which has no boundary vertices.

For level graphs, we are interested precisely in the global distances, or lengths of a globally shortest path, that match above description. In the two cases above, almost all distances can be looked up in the child separator closure for G_l^{k+1}. Still, we need to determine the global distance between boundary vertices of the parent component G_l^{k+1}.

In the graph

$$G|_{B_l^{k+1}},$$

edge costs represent the global distances between boundary vertices of G_l^{k+1}. We use the term *surrounding graph for* G_l^{k+1} to denote this graph. If we computed all distances in the surrounding graph, we could determine the global distance between any two vertices in G_i^k and G_j^k, respectively, from the two cases above, without considering the input graph in detail. Reflecting above considerations on the second case, for components at level $k = n-1$, the surrounding graph is empty.

Luckily, we need only a little extra work to compute a surrounding graph. Recall from the definition of the level graph, that we use global distances to derive the cost function for a level graph. More precisely, for $k < n-1$, for the level graph $\mathscr{L}_{l,l}^{k+1}$, we use exactly the global distances between all boundary vertices of the component G_l^{k+1}. (In other words, the surrounding graph $G|_{B_l^{k+1}}$ is a homomorphism of the level graph $\mathscr{L}_{l,l}^{k+1}$.) That is, by computing $\mathscr{L}_{l,l}^{k+1}$, we effectively compute the surrounding graph $G|_{B_l^{k+1}}$.

In Definition 14, we did not use level graphs $\mathscr{L}_{i,j}^k$ with $i = j$ for a search space graph. However, such level graphs are well-defined, as we have never demanded $i \neq j$ for level graphs $\mathscr{L}_{i,j}^k$. For these special level graphs, we use the term *reflexive level graph*.

Now, by computing the level graphs at level $k+1$ and all child separator closures, we provide all preconditions required to compute efficiently the level graphs at level k. The proceeding for computing the level parts is shown briefly in the following Algorithm 12.

In the forthcoming lemma, we shall refine the actions performed inside the innermost loop for $k < n-1$: For a component, we construct a graph where the distance between any two child separators matches the global distance. After having proven the lemma, we return to the algorithm again and provide a detailed version that contains formal instead of textual descriptions of the actions taken. Figure 15 illustrates the lemma.

Lemma 27. *For* $0 \leq k \leq n-1$ *and for* $l \in I^{k+1}$*, define the* embraced graph

$$\widetilde{G}_l^{k+1} := G_l^{k+1}\big|_{C_l^{k+1}} \cup G|_{B_l^{k+1}}.$$

Then, for all pairs of vertices $s, t \in C_l^{k+1}$*, the global distance matches that in the embraced*

Algorithm 12: Computation of the level graphs (brief)

```
1  forall i ∈ I^{n-1} do
2      forall j ∈ I^{n-1} do
3          compute the level graph 𝒢_{i,j}^{n-1} by looking up the edge lengths in the already
           computed child separator closure G_0^n|_{C_0^n} = G|_{S^{n-1}}
4  for k ← n − 2 downto 0 do
5      forall l ∈ I^{k+1} do
6          create the surrounding graph G|_{B_l^{k+1}} from the reflexive level graph 𝒢_{l,l}^{k+1}
7          retrieve the already computed child separator closure graph G_l^{k+1}|_{C_l^{k+1}}
8          forall i ∈ H_l^{k+1} do
9              forall j ∈ H_l^{k+1} do
10                 use the child separator closure to determine the length of a shortest
                   path among all paths inside the parent component G_l^{k+1}
11                 use the child separator closure and the surrounding graph to
                   determine the length of a shortest path among all paths that contain
                   a boundary vertex of the parent component G_l^{k+1}
12                 set the cost of the corresponding edge in the level graph 𝒢_{i,j}^k to the
                   minimum of both lengths
```

graph $\widetilde{G}_l^{k+1}$*:*

$$d[G]\,(s,t) = d\Big[\widetilde{G}_l^{k+1}\Big]\,(s,t).$$

Proof. As in the proof of Lemma 25, we show that for each path in G between s and t, there exists a corresponding path $\widetilde{G}_l^{k+1}$ that is not longer, and vice versa.

($\Leftarrow$) In this direction, we perform the expansion of a path in the embraced graph by adding intermediate vertices of globally shortest paths.

Fix an arbitrary path $\widetilde{p} = \langle v_1, v_2, \ldots, v_z \rangle$ in the embraced graph $\widetilde{G}_l^{k+1}$ with $v_1 = s$ and $v_z = t$. We construct the corresponding path p in G by expanding each adjacent pair (v_x, v_{x+1}) of vertices in $\widetilde{p}$ to a corresponding shortest path in G. That is, for each (v_x, v_{x+1}) with $x \in \{1, \ldots, z-1\}$, we insert, into p, the intermediate vertices of a shortest path $p_x := \langle v_x, \ldots, v_{x+1} \rangle$. Each expansion does not increase distance, as for each pair (v_x, v_{x+1}) in $\widetilde{p}$, the length of the expanded path p_x is not longer than the length of the edge in $\widetilde{G}_l^{k+1}$. Recall that $\widetilde{G}_l^{k+1}$ is the result of a graph union, so the cost of an edge equals the minimum of the edge's cost in the unified graphs. By that, we only need to verify the following both inequalities:

$$c[G]\,(p_x) = d[G]\,(v_x, v_{x+1}) \quad \leq \quad c\Big[G_l^{k+1}\big|_{C_l^{k+1}}\Big]\,(v_x, v_{x+1})$$

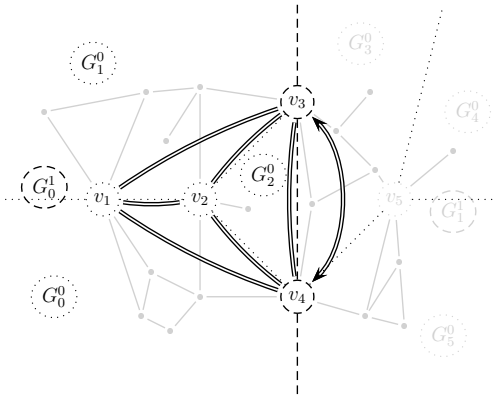

Figure 15: Illustration for Lemma 27.

Assume we want to compute the length of the shortest path between v_2 and v_1 in G. As shown in Figure 14, we cannot rule out the possibility that a shortest path leaves G_0^1 via v_4 and re-enters it via v_3. From Lemma 2, we can deduce that a path can leave or re-enter G_0^1 only via its boundary vertices (namely, v_3 or v_4). We use the surrounding graph $G|_{B_0^1}$ as a shortcut for all shortest paths between the boundary nodes of G_0^1. Then, leaving and re-entering G_0^1 means only one hop in $G|_{B_0^1}$, and we do not need to dig into other level 1 components (here, G_1^1).

In above drawing, the double line with arrows represents the only edge of $G|_{B_0^1}$. All shaded edges in component G_1^1 are already covered by $G|_{B_0^1}$ and can be ignored. We can easily extract this surrounding graph from $\mathscr{L}_{0,0}^1$.

The shaded edges in component G_0^1 do not need to be considered in detail when computing the level graph. Instead, the according child separator closure $\bar{G}_0^1$ (denoted by double-lined edges) is used to quickly retrieve the distance between level 0 separator nodes, cf. also Figure 13.

$$c[G]\,(p_x) = d[G]\,(v_x, v_{x+1}) \quad \leq \quad c\Big[G|_{B_l^{k+1}}\Big]\,(v_x, v_{x+1}).$$

It is easy to see from the definition of the subset closure, that both inequalities actually hold. This is slightly imprecise, as the second inequality is defined only if $v_x, v_{x+1} \in B_l^{k+1}$. However, if v_x or v_{x+1} is no boundary vertex, the edge (v_x, v_{x+1}) is no member of $G|_{B_l^{k+1}}$, and the first inequality suffices to bound the costs of the edge in $\widetilde{G}_l^{k+1}$.

Thus, we can bound the subpath's length by the cost of the edge in the embraced graph $\widetilde{G}_l^{k+1}$:

$$c[G]\,(p_x) = d[G]\,(v_x, v_{x+1}) \leq c\Big[\widetilde{G}_l^{k+1}\Big]\,(v_x, v_{x+1}).$$

As we can do that for each edge of $\widetilde{p}$, the path p in G is at most as long as $\widetilde{p}$ in $\widetilde{G}_l^{k+1}$.

($\Rightarrow$) For this direction, we construct, for a given path in the input graph, a path with either one or three edges in the embraced graph. We use one edge in the embraced graph if the path does not leave the common parent component, and three if it does. As such, this construction closely resembles the differentiation between the two cases at the beginning of the current subsection: The given path in the input graph either stays in the parent component, or it contains a boundary vertex and potentially leaves it.

Fix an arbitrary path $p = \langle v_1, v_2, \ldots, v_z \rangle$ in G with $v_1 = s$ and $v_z = t$. The construction of the corresponding path $\widetilde{p}$ in $\widetilde{G}_l^{k+1}$ depends on whether p contains vertices from S^{k+1}.

1. If p contains no vertices from S^{k+1}, then p does not contain any vertices not from G_l^{k+1} either, according to Corollary 5. This assures that p is not shorter than a shortest path in G_l^{k+1}:
$$d\left[G_l^{k+1}\right](s,t) \leq c[G](p).$$
Set $\widetilde{p} := \langle s,t \rangle$. As the vertices s and t also cannot be boundary vertices of G_l^{k+1}, i.e., $s,t \notin S^{k+1}$, we conclude that the edge (s,t) in the embraced graph originates from the child separator closure of G_l^{k+1}. By that,
$$c\left[\widetilde{G}_l^{k+1}\right](s,t) = d\left[G_l^{k+1}\right](s,t).$$
The path $\widetilde{p}$ consists of only one edge:
$$c\left[\widetilde{G}_l^{k+1}\right](\widetilde{p}) = c\left[\widetilde{G}_l^{k+1}\right](s,t).$$
Finally, by connecting the three relations above, we obtain
$$c\left[\widetilde{G}_l^{k+1}\right](\widetilde{p}) \leq c[G](p).$$

2. In case p does contain a vertex of S^{k+1}, let x be the smallest and y be the largest index so that $v_x, v_y \in S^{k+1}$. That given, no vertex with index smaller than x or larger than y can be a separator at level $k+1$. As a result of Corollaries 5 and 6, the subpaths $p_{1\to x}$ and $p_{y\to z}$ entirely lie in G_l^{k+1}. (Notice that we expressly allow the special cases $v_x = v_y$, $x = y$, $x = 1$ or $y = z$, this does not affect our argumentation.)
Set $\widetilde{p} := \langle s, v_x, v_y, t \rangle$. By definition, the length of this path equals the sum of the edge costs:
$$c\left[\widetilde{G}_l^{k+1}\right](\widetilde{p}) = c\left[\widetilde{G}_l^{k+1}\right](s,v_x) + c\left[\widetilde{G}_l^{k+1}\right](v_x,v_y) + c\left[\widetilde{G}_l^{k+1}\right](v_y,t).$$
Each summand can be bounded by lengths of the corresponding (possibly empty) subpaths of p, according to the definition of the embraced graph:

$$\begin{aligned}
c\left[\widetilde{G}_l^{k+1}\right](s,v_x) &= d\left[G_l^{k+1}\right](s,v_x) \\
&= d[G](s,v_x) \\
&\leq c[G](p_{1\to x}) \\
c\left[\widetilde{G}_l^{k+1}\right](v_x,v_y) &= \min\left\{c\left[G|_{B_l^{k+1}}\right](v_x,v_y), c\left[G_l^{k+1}|_{C_l^{k+1}}\right](v_x,v_y)\right\} \\
&= \min\left\{d[G](v_x,v_y), d\left[G_l^{k+1}\right](v_x,v_y), \right\} \\
&= d[G](v_x,v_y) \\
&\leq c[G](p_{x\to y}) \\
c\left[\widetilde{G}_l^{k+1}\right](v_y,t) &= d\left[G_l^{k+1}\right](v_y,t) \\
&= d[G](v_y,t) \\
&\leq c[G](p_{y\to z})
\end{aligned}$$

We know that $d\left[G_l^{k+1}\right](s, v_x) = d[G](s, v_x)$ and $d\left[G_l^{k+1}\right](v_y, t) = d[G](v_y, t)$, because $p_{1\to x}$ and $p_{y\to z}$ entirely lie in G_l^{k+1}. The first, second or third inequality is obviously obeyed for $s = v_x$, $v_x = v_y$ or $v_y = t$, respectively, as edge costs are required to be nonnegative, and the cost from a vertex to itself is defined to be zero.

By summing up above three inequalities, we obtain the desired result for this case:

$$\begin{aligned} c\left[\widetilde{G}_l^{k+1}\right](\widetilde{p}) &= c\left[\widetilde{G}_l^{k+1}\right](s, v_x) + c\left[\widetilde{G}_l^{k+1}\right](v_x, v_y) + c\left[\widetilde{G}_l^{k+1}\right](v_y, t) \\ &\leq c[G](p_{1\to x}) + c[G](p_{x\to y}) + c[G](p_{y\to z}) \\ &= c[G](p). \end{aligned}$$

The case differentiation is complete, as the second case covers all paths that are not covered by the first case. Thus, we have quoted a corresponding path in the embraced graph for each possible path in the input graph.

We have stated for each path in the embraced graph a corresponding path in the input graph, and vice versa. Our proof is complete. □

Algorithm 13 shows, in detail, how to compute all level graphs. Basically, it is identical to Algorithm 12, only the actions performed have been formulated precisely as a result of Lemma 27. We were able to merge the two outer loops of Algorithm 12 into one main loop.

This algorithm performs correctly, as we have shown in the previous lemma. Still, lines 10 and 13 of Algorithm 13 deserve further explanation. Effectively, in line 10 we compute for a given i the distance from all boundary vertices of G_i^k to all child separator vertices of the current parent component. Then, in line 13, we can look up the distance in $\widetilde{G}_l^{k+1}$ for each possible pair of vertices instead of computing it from scratch.

Note that we could also have computed

$$\widetilde{G}_l^{k+1}\Big|_{C_l^{k+1}}$$

before line 9 to achieve the same result roughly twice as fast: Our algorithm uses most vertices in B_i^k as starting vertex for DIJKSTRA's single-source algorithm more than once. However, the proposed algorithm performs better in terms of memory usage and parallelizability. We need to keep in memory only $|B_i^k| \cdot |C_l^{k+1}|$ path lengths at any time, instead of $|C_l^{k+1}|^2$ which may be substantially larger. Furthermore, for $k = n - 1$, there is only one possible value for l, namely, 0. With Algorithm 13, we can potentially assign each $i \in H_l^{k+1}$ to a different processor, while the seemingly optimal approach can be executed on one processor only, unless we employ a sophisticated communication pattern.

5.3 Parallelization

In this subsection, we provide a scheme for parallelizing the preprocessing procedure for our speed-up technique. First, we outline the basic strategy for parallelizing the preprocessing:

Algorithm 13: Computation of the level graphs (detailed)

```
1  for k ← n − 1 downto 0 do
2      forall l ∈ I^{k+1} do
3          retrieve G_l^{k+1}|_{C_l^{k+1}} from the already computed child subset closures
4          if k < n − 1 then
5              create G|_{B_l^{k+1}} from the reflexive level graph 𝒢_{l,l}^{k+1}
6          else
7              use the empty graph as G|_{B_l^{k+1}}
8          G̃_l^{k+1} ← G_l^{k+1}|_{C_l^{k+1}} ∪ G|_{B_l^{k+1}}
9          forall i ∈ H_l^{k+1} do
10             compute G̃_l^{k+1}|_{B_i^k → C_l^{k+1}} using Dijkstra's single-source algorithm
11             forall j ∈ H_l^{k+1} do
12                 foreach (s, t) ∈ B_i^k × B_j^k do
13                     set the length of the corresponding edge in 𝒢_{i,j}^k:
                       c[𝒢_{i,j}^k] ((s, k + 1), (t, −k − 1)) ← d[G̃_l^{k+1}] (s, t)
```

We split the preprocessing algorithm into small tasks that must obey a linear precedence relation imposed by the data flow between tasks. This is done by introducing a generic system of parametrized task types. For this system, we identify a bottleneck and show a possible workaround. Finally, we briefly justify our way of subdividing the preprocessing algorithm.

Assume a computational problem that has been split up into several sub-tasks with a cycle-free static dependency relation between tasks. That is, each task has a set of *precondition tasks* and potentially constitutes as precondition for other tasks itself. As a cycle-free static dependency is also deadlock-free, we can use a lazy execution scheme to solve the computational problem: We assign a task to a free processor as soon as all precondition tasks have finished. If no processor is free, the task is put into ready state and assigned to the next free processor. With a static *priority* assigned to each task, we can fine-tune the choice of the next task. We can employ this parallelization scheme to our preprocessing algorithm, provided that we can find a suitable subdivision into smaller tasks. A task can have four states:

Blocked There exist unfinished precondition tasks.

Ready All precondition tasks have finished, but this task has not been started yet.

Running The task has been started, but still is in process of computation.

Finished Computation of this task is complete. By that, other depending tasks may have become ready.

As we required the dependency relation to be static, the only transitions possible are from blocked to ready, from ready to running and from running to finished.

The *scheduling*, or assignment of tasks to processors, is performed by an omniscient *master* processor that keeps track of the processor load. Correspondingly, the processors the tasks are assigned to are called *slaves*. We make do with a greedy priority-based scheduling here, as our dependency graph is deadlock-free.

Recall from the previous two subsections, that some computations needed to be carried out in a well-defined order. To sum up, the following dependencies were introduced by our preprocessing algorithms:

- To compute a child separator closure at non-bottom level k for a component G_i^k, we need the child separator closures of all child components at level $k-1$.
- For any search space part, we need a certain subset closure to be available:

  Entry and exit graph For search space parts $\mathscr{E}_v$, $\mathscr{X}_v$ with $v \in V_i$, we need one of the elevating closures $G_i^0|_{V_i \to B_i^0}$ or $G_i^0|_{B_i^0 \to V_i}$, respectively.

  Upward and downward graph A search space part $\mathscr{U}_i^k$ or $\mathscr{D}_i^k$ at level k requires the child separator closure of the corresponding parent component $G_{f^k(i)}^{k+1}$ to be available.

  Level graph For $\mathscr{L}_{i,j}^k$, a level graph at level k, we also need the child separator closure of the common parent component at level $k+1$.

- Finally, a level graph also requires the reflexive level graph for its parent component to have been computed already.

For the parallelization, we split the computation of our preprocessing in tasks of different kinds. We need to choose this subdivision carefully. If it is too finely-grained, we may encounter an intolerable decrease of performance. On the other hand, a split too coarse decreases the benefit from parallelization by creating *bottlenecks* — tasks that are a direct or indirect precondition for many other tasks. Situations may occur where the bottleneck task is the only task running and no ready tasks are available because all other tasks depend upon the bottleneck task.

In the following, we show the seven *task types* employed by our solution. Each task type serves a predefined purpose. A *task* is specialized by providing parameters for a task type. The preconditions for a task type depend only on the task's parameters and are therefore static for a given set of tasks.

To each task, a priority that is potentially dependant on its parameters is assigned. As opposed to the commonly used interpretation, high-valued priorities indeed mean precedence over low-valued priorities to avoid confusion. The priority is a soft criterion, we

employ it only if we have more than one ready task. That is, a task with a low priority can be executed while a task with a high priority still is blocked.

We have a designated parameter-less task type called `Goal` that does not compute anything. It only serves as a collector for all task that we need for our preprocessing, as it employs these tasks as preconditions. As soon as this task has finished, our computation is complete.

The first type of task, `Decomposition`, computes a hierarchical decomposition for a given input graph G. We treat this task as "black box". This task type is the only one that contains no requirements. Obviously, we must begin by computing the only task of this type.

Task `Decomposition`
Priority : 2
compute: a hierarchical decomposition for the input graph G

The task type `ClosureDir` computes the elevated closures needed for the entry and exit graphs and also for the child separator closures.

Task `ClosureDir(`i`)`
Priority : 1
require : `Decomposition`
compute: the elevating closure $G_i^0|_{V_i^0 \to B_i^0}$
compute: the elevating closure $G_i^0|_{B_i^0 \to V_i^0}$

Tasks of the type `Closure` compute child separator closures for a component G_i^k. Each tasks that computes upward, downward and level graphs is dependent on a task of this type. As shown in this section, we use child separator closures at a lower level to compute those at a higher level. Thus, we require, for $k > 0$, all child separator closures of all child components of G_i^k. Due to our choice for the priority, the computation of all closures at low level starts before all closures at high level, but after all tasks of type `ClosureDir` have started. (We do not expressly require all task of type `ClosureDir` to have finished before a task of type `Closure` can start, as this might also create a bottleneck.)

For the task type `EntryExit`, we depend only on the directed subset closures computed by `ClosureDir`. Besides `Goal`, no other tasks depend on tasks of this type. Hence, we assign tasks of this type a low priority.

Tasks of the type `UpDown` need a specific child separator closure to be computed. For the same reasons as with `EntryExit`, we employ here a low priority, too. Note that tasks of this type also optimize the upward and downward graphs with the algorithms presented in Section 4.

Recall that level graphs used the surrounding graph that could be obtained from a reflexive level graph at the level above. By that, tasks of the type `Level` depend on a child separator closure and on another task of the same type. The choice for the priority mirrors

Task `Closure`(k, i)

Priority : $-k$
if $k = 0$ **then**
  require : `ClosureDir`(i)
  compute: the boundary closure $G_i^0|_{B_i^0}$
else
  forall $i' \in H_i^k$ **do**
    require : `Closure`($k-1$, i')
  compute: the child separator closure $G_i^k\big|_{C_i^k}$

Task `EntryExit`(i)

Priority : $-n-1$
require : `ClosureDir`(i)
forall $v \in V_i^0$ **do**
  compute: the entry graph $\mathscr{E}_v$
  compute: the exit graph $\mathscr{X}_v$

Task `UpDown`(k, i)

Priority : $-n-1$
require : `Closure`(k, i)
forall $i' \in H_i^k$ **do**
  compute: the optimized upward graph $\mathscr{U}_{i'}^{k-1}$
  compute: the optimized downward graph $\mathscr{D}_{i'}^{k-1}$

this: To avoid potential bottlenecks, we want all computations for level graphs at a higher level to have started before starting those for level graphs at a lower level. Here, we also perform the optimization of the level graph right after having computed it.

Task `Level`(k, i)

Priority : k
$l \leftarrow f^k(i)$
require : `Closure`($k+1$, l)
if $k < n-1$ **then**
  require : `Level`($k+1$, l)
forall $j \in H_l^{k+1}$ **do**
  compute: the optimized level graph $\mathscr{L}_{i,j}^k$

The final task simply requires all tasks that, in total, compute all search space parts our preprocessing consists of. Its priority does not matter, as the final task does not perform

Task `Goal`

forall $i \in I^0$ **do**
  require : `EntryExit(`i`)`
for $k \leftarrow 1$ **to** $n - 1$ **do**
  forall $i \in I^k$ **do**
    require : `UpDown(`k`,` i`)`
forall $i \in I^k$ **do**
  require : `Level(`k`,` i`)`

any computation by itself. When the final task has finished, our preprocessing is complete.

Figure 16 shows all tasks executed by the preprocessing of our sample graph as a dependency graph. Apart from the decomposition, the task that computes the child separator closure at the highest level is the main bottleneck: The tasks for level graphs all depend on it, directly or indirectly. This problem is partially compensated by our choice of the priorities. Effectively, we delay the computation of entry, exit, upward and downward graphs until the task for the top-level child separator closure starts. By that, one processor computes the bottleneck task `Closure(`n`, 0)`, while the others take care of the entry/exit and upward/downward tasks.

However, in our experiments, we have observed that the task `Closure(`n`, 0)` constitutes a real bottleneck for some input instances: All processors but one were idle for some time until this task has eventually finished. It may be worthwhile to further split this special task into one sub-task for each processor available and merge the results later.

Finally, we shall briefly motivate our subdivision of the preprocessing algorithm into tasks. Our set of task types is sort of a natural partitioning of the non-parallel algorithms: The tasks of type `ClosureDir` and `Closure` together form Algorithm 11, while a single level task is equivalent to the execution of Algorithm 13 for a fixed i. (This reflects our considerations on parallelizability from the previous subsection.) As the entry, exit, upward and downward graphs are not used as input for other computations, we are unbound in the choice of the granularity of these tasks. By that, we are able to use such a granularity that each child separator closure is utilized by exactly one task of type `UpDown`. Accordingly, each elevating closure is employed by exactly one task of type `EntryExit`.

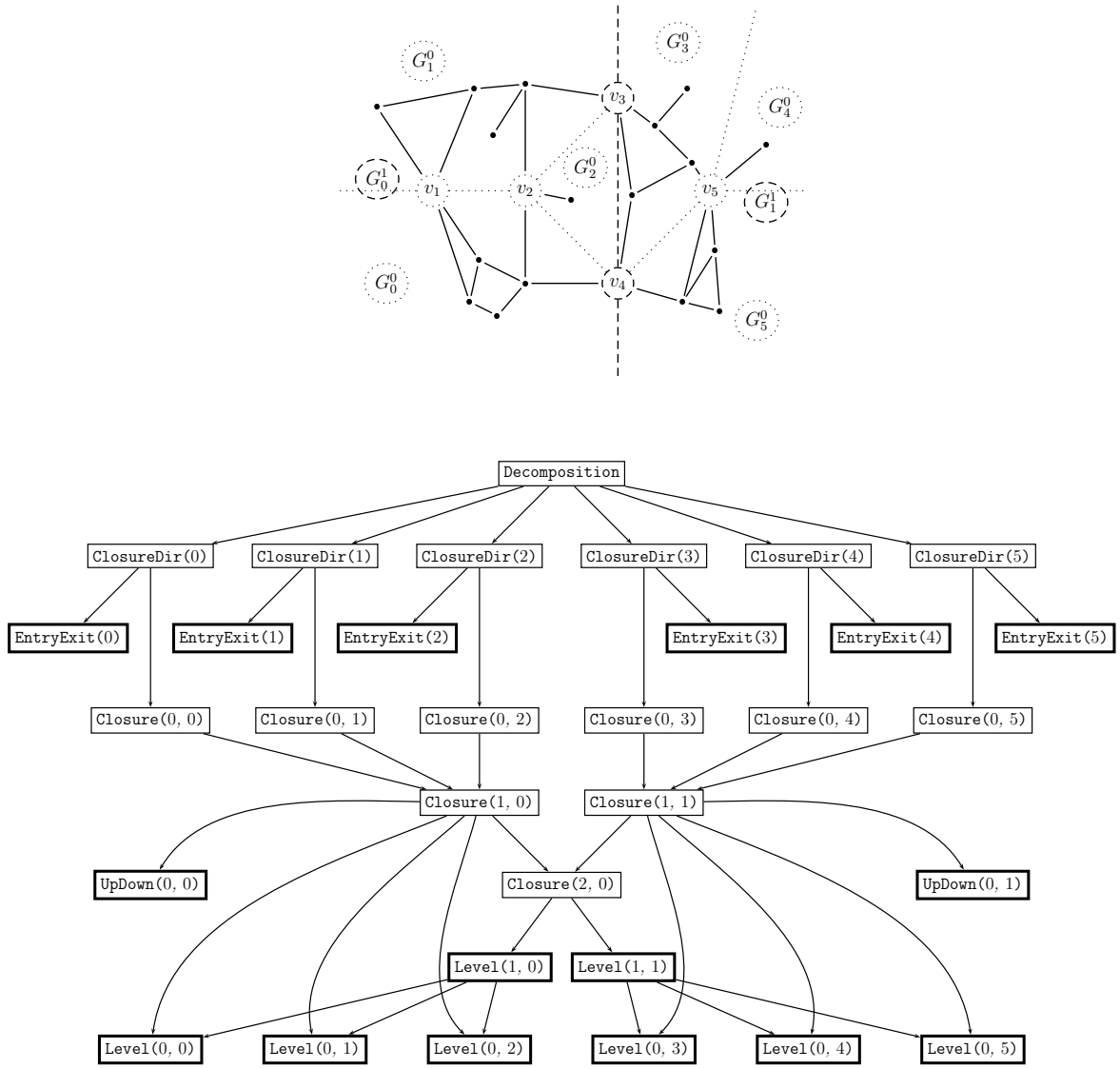

Figure 16: Dependencies between tasks.

The drawing at the top shows, once again, our sample graph in the hierarchical decomposition used throughout this work.

In the drawing at the bottom, each box represents a task for the precomputation of our sample graph. The arrows represent dependencies. Only the task `Decomposition` does not depend on any other task. The task `Goal` has been omitted for the sake of clarity; instead, all tasks that are a prerequesite for `Goal` have a bold-lined box. Note the two main bottlenecks: The task `Decomposition` at the beginning, and the task `Closure(2, 0)`.

6 Implementation

This section is devoted to a brief presentation of our reference implementation that has been used for the empirical analysis shown in the next section.

The implementation consists of several applications, all written in C++. For compilation, we used GCC in version 3.4.3 running under SuSE Linux 2.6.11 on a machine with two AMD Opteron processors. In addition, most programs employed LEDA, the Library of Efficient Data types and Algorithms [NM99], in version 5.0.1.

In the following, we highlight some interesting facts about our implementation.

Scalable and parallel From the beginning, perhaps the main goal has been to provide an implementation that would be able to handle huge input graphs. We attained this goal: Our implementation can even handle graphs as large as the road map graph of Western Europe. However, we believe that the implementation still can be sped up by magnitudes.

It is worth mentioning that many subtle technical problems that would have remained unnoticed with small inputs came up in the process of implementation, as we processed larger inputs.

We used the Message Passing Interface [GLS94], as a base for our parallel implementation of the preprocessing procedure. Of course, MPI can be used in a much more sophisticated way than for implementing a master-slave task scheduling system. Anyway, it suited our needs: By using the MPICH implementation [MCH] of MPI, we were able to configure, in short time, a computer cluster from the four machines available to us. As MPI is portable, our implementation can also be run on other parallel architectures, e.g., on a massively parallel shared-memory machine.

Also, the memory footprint for our parallelized preprocessing is considerably low: We get by with 512 MB of RAM per processor for preprocessing the road map graph of Western Europe.

All data produced by our preprocessing application, intermediate or final, is stored to and read from secondary storage. We access the data via NFS (Network File System), this saved us from implementing a communication pattern. However, this also constitutes a major performance penalty, as the creation of a file via NFS is expensive. Owing to that, to save file create operations, we merged data into single files that best would have resided in different files. We expect that, by passing data via MPI instead of via NFS, we can again improve the performance of our implementation.

Human-readable data Whenever there was a need to store intermediate or final data, XML has been employed. This is another slowdown, as binary data can be written and read significantly faster than XML. However, this simplified debugging a lot, as we were able to use standard Unix text processing tools like `grep`, `sed`, `sort`, `uniq` and the like, to quickly analyze the preprocessed data when the implementation produced wrong results.

To minimize performance impact, we used `FleXML` [Ros], a tool that transforms a given DTD into a table driven parser. (More precisely, it creates a `flex` rule set [LMB92].) A

parser that is generated by this tool accepts only documents that conform to the DTD the parser has been created for. In exchange, these documents can be parsed really fast compared to generic XML parsers that accept a variety of document types.

The DTDs used for our applications are presented in Appendix B.

As we are usually interested only in a small part of the data an XML document contains, we made use of *forward pointers* in our documents. The forward pointers were implemented as attributes named `offset`, values for this attribute must contain exactly 16 hexadecimal digits. The `offset` attribute stores the file offset for a specific record in an XML document. By reading the offset for a forward reference, we already know where the data we are looking out for resides in the document, and can quickly position to it instead of reading over the whole document. As we do not know the position of a record when creating the XML document, we insert a placeholder as value for the `offset` attribute and overwrite this placeholder with the real file offset once we know it.

Generic command interface for unattended execution Often, several commands need to be executed repeatedly, but with different parameters, e.g., for different input graphs. As the commands usually constitute a pipe in the sense of data flow, the second command needs to wait for the first command to complete, and the whole process should be aborted on error. Reflecting this, a shell script has been written that creates, based on the parameters, a custom `Makefile` that has a target for each command that needs to be executed. Afterwards, `make` is executed. This script serves as the only control for our applications, the applications are never started manually. This is convenient, as we only need to remember the `make` target names instead of the commands with all their options, and we can run a command pipe unattended without taking special care.

Restartable If the preprocessing is interrupted, it can be later restarted at the point of interruption without unnecessarily recomputing data. This is possible, as the preprocessing application checks, for each file to be created, if it already exists and if it is in a correct format. In the affirmative case, there is no need to rewrite the file. As each check can be performed very quickly, without the need to read the whole file, the overhead for stopping and restarting the preprocessing is low.

7 Empirical Analysis

In this section, we present the results of the experiments that we conducted with our implementation. As input, we used road maps graphs of Western European countries, courtesy of PTV AG, Karlsruhe.

We tested two series, one with road map graphs of different countries and the other for different decompositions of a fixed road map graph. Finally, we present our result for the preprocessing of the road map graph of Western Europe.

For our tests, we used four machines, each equipped with two AMD Opteron processors and interconnected by a separate switched Gigabit Ethernet. The characteristics of each machine are listed in Table 1.

Name	RAM	CPU model name	CPU clock rate	L1 cache
`compute3`	4 GB	2 x AMD Opteron 248	2 x 2.1 GHz	2 x 1 MB
`compute4`	8 GB	2 x AMD Opteron 248	2 x 2.1 GHz	2 x 1 MB
`compute5`	16 GB	2 x AMD Opteron 252	2 x 2.5 GHz	2 x 1 MB
`compute6`	16 GB	2 x AMD Opteron 252	2 x 2.5 GHz	2 x 1 MB

Table 1: Characteristics of the machines used in our tests.

For each test, we started one master and eight slave tasks that performed the preprocessing in parallel. The preprocessed data was written to and read from a separate hard disk with a capacity of 60 GB installed on `compute4`. Access to this hard disk was performed via NFS. The coordinating master process always was executed on this machine, too.

For this section, we use informally the terms *preprocessing size* and *preprocessing time* to denote the total size of the data needed to execute an arbitrary query, and the elapsed time to produce this data. We do not consider the time needed to obtain a hierarchical decomposition of the input graph.

As a parameter to our tests, we use the *granularity of the hierarchical decomposition* (in short, *granularity*) that declares, for each level in the hierarchical decomposition, a maximum for the number of boundary vertices a component may have. For a given granularity, we obtain a hierarchical decomposition with at most the required number of boundary vertices for each component. Note that a granularity constitutes an implicit guarantee for the maximum size a search space can possibly have.

7.1 Graph Size vs. Preprocessing Time

In the first series of experiments, we ran the preprocessing for our speed-up technique for several countries of different sizes. We used two levels and required components at level 0 to have no more than 20, and components at level 1 to have no more than 40 boundary vertices, respectively.

Table 2 presents the main results for this series. Notably, there is no obvious dependency between the number of vertices and the time or space required for the preprocessing. We

Country	Number of vertices	Number of edges	Time (s)	Size, compressed (MB)	Time per 1,000 vertices (s)	Throughput (KB/s)	Size per vertex (bytes)
Belgium	458,936	1,093,454	21:41	185	2.83	145	422
Spain	695,770	1,546,558	20:46	193	1.79	158	290
The Netherlands	850,186	2,034,591	14:58	174	1.06	198	214
Sweden	1,546,984	3,580,059	22:22	310	0.87	236	210
Italy	2,078,477	4,824,859	55:07	509	1.59	158	257

Table 2: Preprocessing of the road map graphs of different countries with a fixed granularity.

assume that the structure and density of a country's road network affects these values stronger than the number of vertices alone. The amount of produced preprocessing data per second, denoted by the term *throughput*, suggests that the preprocessing is CPU-bound rather than being I/O-bound.

As the preprocessing size tends to be quadratic in the number of vertices for a fixed granularity, we conclude that this granularity is too fine to be applicable for preprocessing the whole road map graph of Western Europe. We mention the compressed size here, as it reflects the amount of main memory needed to load the preprocessed data better than the uncompressed size.

7.2 Trade-off between Preprocessing Effort and Query Time

In the next series, we considered different granularities for a fixed input graph formed from the road map graphs of Spain and Portugal. The graph contains 855,660 vertices. We considered hierarchical decompositions of two levels only.

Table 3 shows an overview of the results. The data is grouped by the granularity at level 0, as preprocessings with different granularities at the lowest level cannot easily be compared: For coarser granularities, the size of the bottom-level components increases and fewer vertex pairs are covered by our preprocessing.

In our test, we analyze three *size indicators*: The initially guaranteed search space size, imposed by the granularity, and the maximum and average search space sizes as observed from the preprocessed data. In each group, all size indicators are nondecreasing. This

Granularity at level 0 (boundary vertices per component)	Granularity at level 1 (boundary vertices per component)	Preprocessing size (compressed MB)	Preprocessing time (m:ss)	Throughput (compressed KB/s)	Initially guaranteed search space size	Maximum search space size	Average search space size
20	25	268	36:02	127	1665	658	213
20	30	130	22:23	99	2140	916	286
20	35	88	17:18	86	2665	1133	387
20	40	104	19:16	92	3240	1440	530
20	45	100	22:13	77	3865	1440	563
20	50	98	16:16	102	4540	1608	655
25	30	117	13:38	147	2450	940	229
25	35	55	10:17	91	3025	1291	358
25	40	46	8:15	94	3650	1604	561
25	45	41	8:23	84	4325	1604	611
25	50	38	7:26	87	5050	1676	680
30	35	43	6:55	105	3385	1360	343
30	40	31	5:41	94	4060	1649	567
30	45	27	4:43	97	4785	1649	615
30	50	24	4:29	90	5560	1664	700
35	40	19	3:00	107	4470	1556	484
35	45	14	2:56	82	5245	1615	525
35	50	11	2:12	84	6070	1830	619
40	45	13	2:45	83	5705	1360	367
40	50	10	1:59	86	6580	1559	467
45	50	10	1:42	98	7090	1559	429

Table 3: Preprocessing of a fixed input graph with different granularities.

holds, by definition, for the initially guaranteed size. However, for the preprocessings in the first group, both size and time are not monotonic — a local minimum for our series is at (20;35). With higher differences between the granulations at level 1 and level 0, the number of level graphs at level 0 increases in a nonlinear fashion. So, the granularity (20;35) can be considered a good trade-off between preprocessing effort and query time.

Furthermore, the granularity at the top level, if chosen too fine, may result in an infeasible preprocessing size, as the distances between all top-level separators need to be computed. Compare, for instance, the preprocessing size for (20;25) to that for (20;30) and (20;35).

In Figure 17 we present three diagrams for this experiment. We analyze the impact of the optimization of the search space parts on their size and the relationship between the size indicators. An interpretation of the diagrams can be found underneath.

7.3 The Road Map Graph of Western Europe

The preprocessing of the road map graph of Western Europe was the heftiest stress test for our implementation. For this preprocessing, we used a three-level setup with a granularity of at most 20, 40 or 80 boundary vertices per component, respectively. By that, we guarantee that our search spaces have less than

$$80 \cdot 80 + 2 \cdot 80 \cdot 40 + 2 \cdot 40 \cdot 20 + 2 \cdot 20 = 6400 + 6400 + 1600 + 40 = 14440$$

edges each. However, the actual search spaces are by far smaller on average, mainly due to optimization of the search space parts.

General information At a glance, we present some key facts about the preprocessing for the road map graph of Western Europe in Table 4.

In Table 5, we show the impact of search space part optimization on the sizes of the level graphs at the top level. We focus on these level graphs, as they account for the vast majority of the edges in our preprocessed data. About a quarter of all level graphs could be reduced to star graphs. For each level graph, half of the edges could be reduced on average, either by supersedement, by obsoletion or by both. The next two lines show the number of edges that can be removed if just one of the optimization methods is used. The last line states that, on average, one fifth of the edges of each level graph were considered reducible by both optimization methods.

Query Unfortunately, we were unable to construct the entry and exit graphs for our queries, as this would have exceeded the amount of hard disk space available. This is merely a restriction caused by wasteful usage of memory and not a fundamental problem. Instead, we compose the entry and exit graphs on demand from the bottom level components for our query algorithm.

To test our speed-up technique, we computed the distances between more than 36,000 randomly chosen pairs of vertices, both with Dijkstra's algorithm and with our query

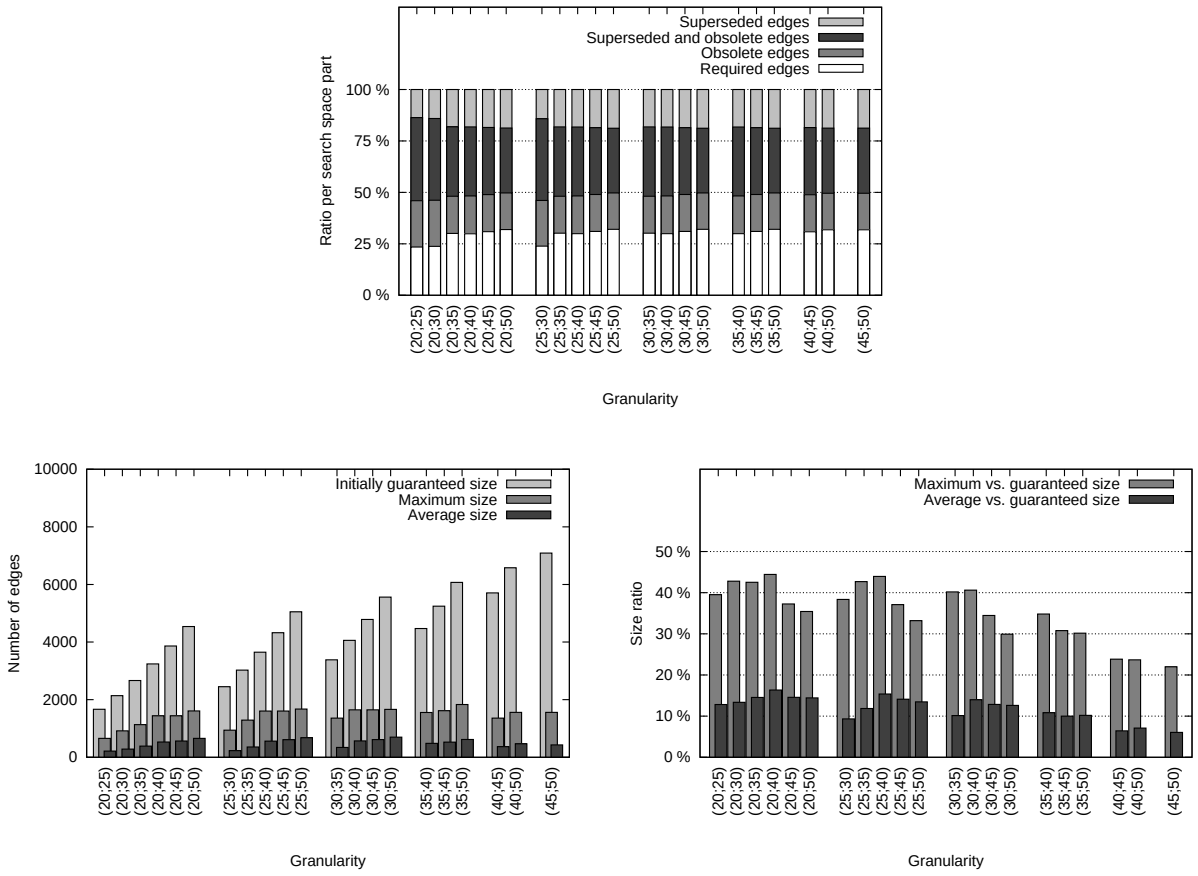

Figure 17: Spain and Portugal: Ratio of required edges. Sizes of search space graphs.
In the diagram at the top, we analyze the impact of the optimization of search space parts on their size. We count, for each level graph at the top level, the number of superseded and obsolete edges and relate it to the size of the bipartite graph. The edges that are removed by both optimization techniques are counted separately. The plotted data represents the mean over all level graphs at the top level. On average, no more than a third of the edges of the bipartite graph are needed. Note also that roughly half of the edges of a search space graph are superseded.
The two diagrams at the bottom correlate the size indicators. The maximum size is never bigger than half the guaranteed size, while the average size can be expected to be around 10 % of the guaranteed size.

Countries used	A, B, CH, D, DK, E, F, I, L, N, NL, P, S
Vertices	15.4 million
Edges	35.7 million
Average vertex out-degree	2.33
Total elapsed preprocessing time	20 hours @ 8 processors
Size of the input data (GraphML)	7.5 GB
Size of the input data (GraphML, compressed)	521 MB
Total size of preprocessed data (XML)	14.6 GB
Total size of preprocessed data (compressed)	2.19 GB
Memory overhead for preprocessing (factor)	1.95
Memory overhead for preprocessing (factor, compressed)	4.3
Additional bytes per vertex (XML)	525
Additional bytes per vertex (compressed)	154

Table 4: The preprocessing for the road map graph of Western Europe.

Perfectly minimizable	27 %
Average superseded and/or removed edges	51 %
Average superseded edges	40 %
Average obsolete edges	31 %
Average superseded and obsolete edges	20 %

Table 5: Western Europe: Optimization of the level graphs at the top level.

algorithm. We considered only pairs of vertices with distinct home components. The preprocessing resided completely on secondary storage, only the search space parts necessary for each query were loaded, and dropped again after the query has been performed.

First, we compare our speed-up technique to DIJKSTRA's algorithm. As a measure, we use the size of the *search space*. For DIJKSTRA's algorithm, the search space consists of all edges touched. The same holds for our algorithm, too. However, as our query algorithm touches each edge of a search space graph exactly once, the size of the search space equals the size of the search space graph — hence the name.

In Figure 18, three diagrams show the sizes of the search space with DIJKSTRA's algorithm and with our speed-up technique and the resulting speed-up. For our set of queries, we observed an average speed-up of 9,960.

Figure 19 presents an analysis of the distribution and the frequency density of the sizes of the search space graphs used in our queries.

Query run time is depicted in Figure 20. For this, we separately loaded, for each search space graph, the required search spaces from secondary storage. Each search space part is loaded into a separate data structure. We measured the time needed to initialize the distance labels for all vertices and to perform the query. To allow for a more precise measurement, we ran each query 1,000 times and divided the total time by the number of

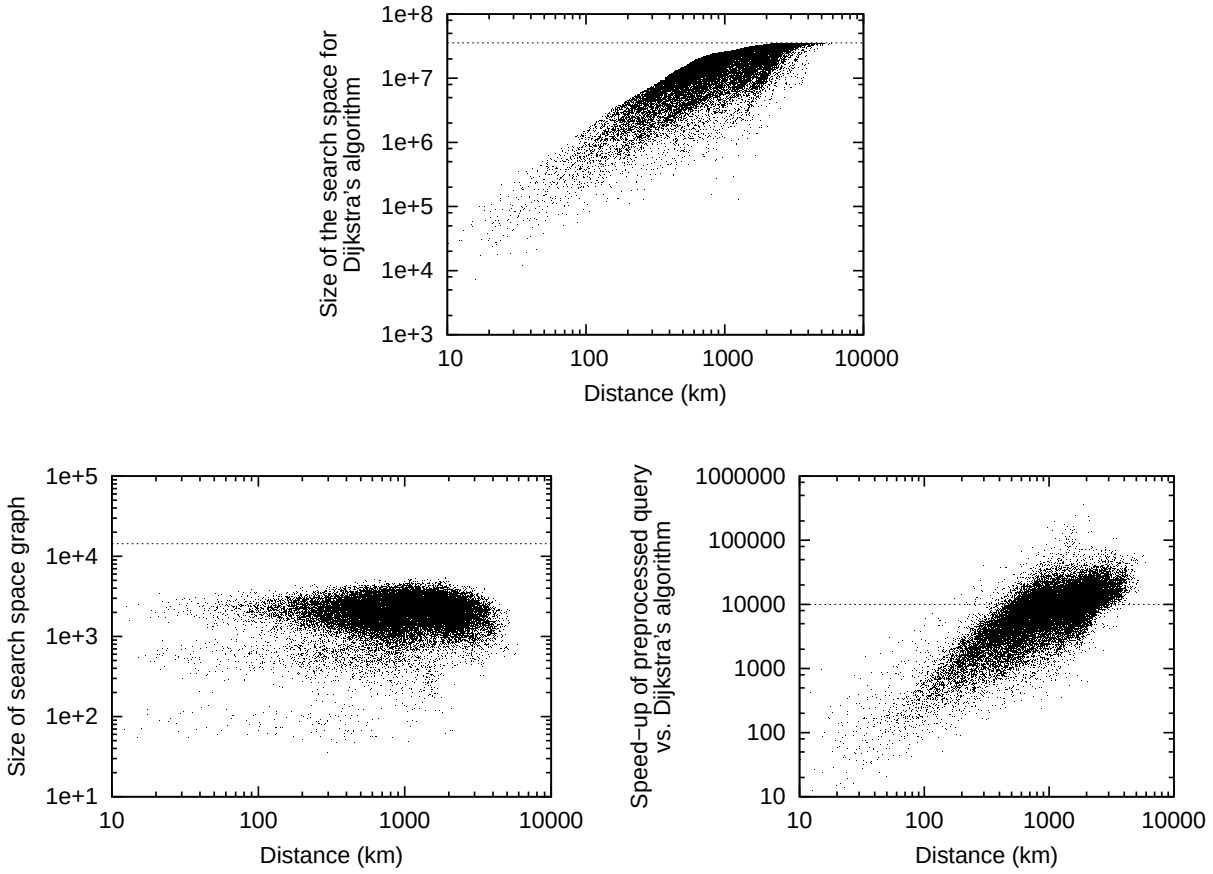

Figure 18: Western Europe: Query.

In the three diagrams above, we analyze the performance of more than 36,000 queries between randomly chosen vertex pairs for the road map graph of Western Europe. All diagrams are plotted against the length of a shortest path between the randomly chosen vertices. Each query executed matches a point in each diagram. As the vertices were chosen at random, most vertex pairs were comparatively far from each other.

The diagram at the top visualizes the size of the search space for DIJKSTRA's algorithm. The search space is naturally bounded by 35.7 million, the number of edges in the graph. The dashed line represents this bound.

The diagram at the bottom left shows the sizes of the search space graphs with our speed-up technique for the same set of queries, which is precisely the number of edges our query algorithm needs to consider. The diagram resembles three clouds stacked over each other: One fat one at the top, around the y-value of 2,000, a second, much thinner cloud around 500, and a third one, vaguely noticeable, around 80. The clouds represent queries against search space graphs consisting of seven, five or three search space parts, respectively. Note also that our initial guarantee of no more than 14,440 edges per search space graph, visualized by a dashed line, has been significantly undercut.

In the diagram at the bottom right, we compared the performance of our speed-up technique with that of DIJKSTRA's algorithm. For that, we divided the size of the search space of DIJKSTRA's algorithm by the size of our search space graph. The average speed-up of 9,960 is marked with a dashed line. The fact that our query algorithm requires only linear time in the number of edges has not been incorporated into this diagram.

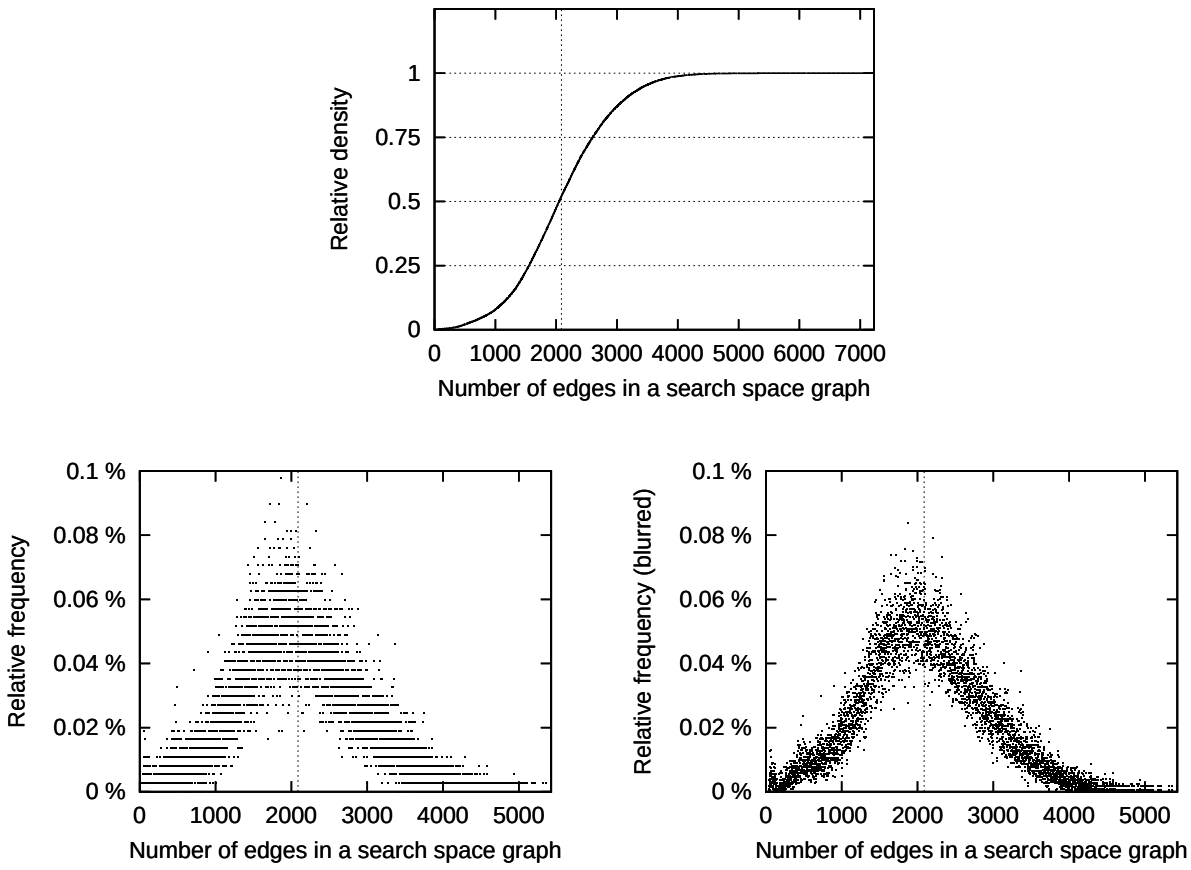

Figure 19: Western Europe: Distribution of search space graph sizes for a given set of queries.

In the plots, the density and the frequency for the number of edges in the search space graphs encountered for our set of queries are shown. The vertical line at 2,088 denotes the average size of all search space graphs, which is pretty close to the median. We counted 5,426 edges in the biggest search space graph used for our queries.

The plot at the bottom right is simply a blurred version of the plot at the bottom left. It has been obtained by convoluting the measured data with a narrow normal distribution.

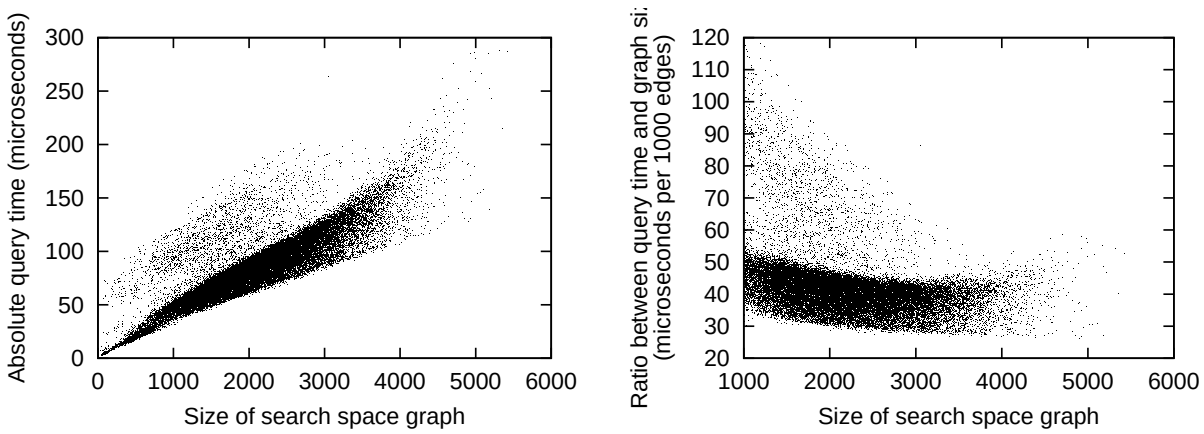

Figure 20: Western Europe: Query run time.
In the two plots above, we analyze the run time of our query algorithm. Each dot represents a query executed.
The plot at the left shows the run time with respect to the size of the corresponding search space graph. Note that the run time is mostly scattered around a line through origin, except for a comparatively small number of search space graphs with few edges.
The plot at the right helps to estimate the asymptotic behaviour. Owing to that, we have considered only search space graphs with more than 1,000 edges. If we plot the same ratio for the smaller search space graphs as well, we need a much larger scale for the ratio to fit all queries. The plot suggests asymptotically linear run time with respect to the number of edges.

runs.

In the plots, comparatively few queries took longer than average, especially for small numbers of edges in the search space graph. An investigation of the reasons for this may be worthwhile, but was beyond the scope of this work.

Furthermore, we observe a slight increase of the average query times for search spaces larger than 3,000 or 4,000. We hold cache effects responsible for this, as search space graphs with a size of 3,000 edges or more may have a footprint large enough to exceed the size of the L1 cache, while smaller ones may fit. Yet, a precise analysis also fell out of the scope of this work.

In the following, we shall discuss how our query algorithm will perform if employed in an industry-strength system that keeps all search space parts in main memory. Our implementation of the query algorithm does not copy the search space parts to construct the graph explicitly. Thus, we only need to determine and locate the search space parts required for a query before we can run our query algorithm. This lookup can be implemented by means of few array accesses if the vertex set V and the component index set I both are compact integer sets, or with few accesses to a dictionary structure otherwise. Furthermore, most our search space parts resided completely in the processor's L1 cache even before the query was executed. For an industry-strength system, we need to account for the time needed to load the search space parts from RAM into the L1 cache. We are confident that lookup, location and caching can be implemented to take at most as long as the time

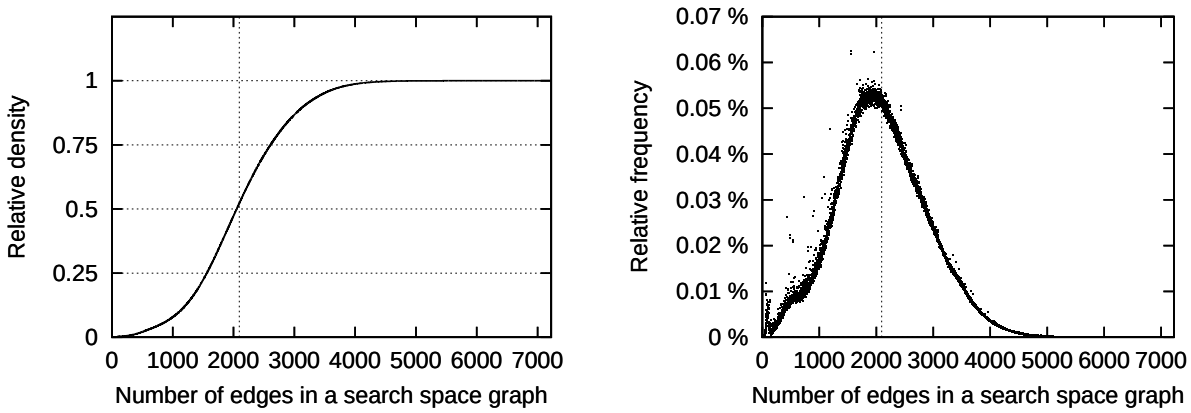

Figure 21: Western Europe: The sizes of all possible search space graphs.
The plots above show the density and the frequency for the number of edges in all possible search space graphs for our preprocessing of the road map graph of Western Europe. The mean, or expected size of a search space graph, equals 2,095. It has been marked with a vertical dashed line. Again, mean and median almost coincide. The biggest of all possible search space graphs has 7,223 edges.

needed to perform the query itself.

Off-line analysis From our preprocessed data, we are able to compute the frequency distribution for the sizes of all possible search space graphs. From that, we can derive the expected size of a search space part for a random pair of vertices and also strictly bound the size any search space can have. Of course, we do not iterate over all pairs of vertices to obtain the distribution. On the contrary, we consider the number of edges of each search space part only once and derive the desired distribution by means of addition and convolution of the size distributions for the particular search space parts.

Figure 21 shows the frequency density and the frequency distribution for the sizes of search space graphs among all possible search space graphs. The distribution strongly resembles a Gaussian distribution, apart from a tiny part of the search space graphs has more than 4,000 edges. Due to these few search space graphs, we can only guarantee that the search space will have no more than 7,223 edges with this preprocessing. Notably, the maximum of all possible sizes of search space graphs is quite precisely a half of the initial guarantee of 14,440 imposed by the granularity.

We explain briefly a feasible technique for beating down the guaranteed size of a search space graph, as this would be beyond the scope of this work. First, we specify an upper bound for the size of any search space, say, 5,000 in our case. Obviously, level graphs at the top level potentially use more edges than the other kinds of search space part. If we encounter a *bad* level graph $\mathscr{L}_{i,j}^{n-1}$, i.e., one that would generate search space parts exceeding our requested guarantee, we compute and store the graph union between this bad level graph and all upward graphs that can be connected to this bad level graph. Informally, we *bypass* the bad level graph by direct connections from the boundary verti-

ces of all child components of G_i^k to the boundary vertices of G_j^k. Note that this does not obsolete the original upward graphs, as other level graphs still may be connected to them. However, the particular search space graphs that would normally contain our bad level graph now can be built with the help of the readily available bypasses and would exhibit a smaller size. Other variants of bypasses are also possible.

Finally, in Figure 22 we compare the search space graph sizes observed with our queries and those of all possible search space graphs. For this, we plot the difference between the two frequency densities. As expected, both densities differ only slightly.

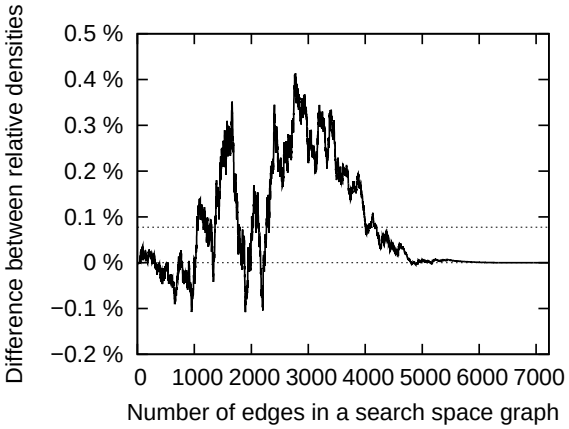

Figure 22: Western Europe: Difference between size distributions.
The above plot shows the difference between the frequency densities for sizes of search space graphs: We subtracted the average sizes from the sizes from our queries. The lower dashed line represents zero, whereas the upper one denotes the mean of the difference, which is lower than 1 ‰.

8 Conclusion and Outlook

In this work, we have presented an efficient preprocessing approach to the shortest-path problem. We have provided a strong theoretical fundament and proven the correctness of each aspect of our approach. Our theoretical considerations were backed up by empirical evidence.

The main contribution in this work is a scalable implementation of our speed-up technique that can handle huge input graphs such as the road map graphs of Western Europe. The achievable query performance turned out to be competitive to state-of-the-art techniques. In the scope of this work, many prospects could not have been investigated in depth. In the following, we depict some of topics that, as we believe, deserve further attention.

Improved implementation Our implementation could be optimized for speed to improve preprocessing time. The options include, but are not limited to the following:

- Employment of binary data formats in place of XML, while maintaining human readability through converters from the binary formats to XML.
- Tailored data structures.
- Usage of MPI instead of NFS to exchange data.
- Faster heuristics for the minimization of the search space parts.

Loading the whole preprocessed data into RAM In our reference implementation, the preprocessing resided on secondary storage. By loading the whole amount of preprocessing data into main memory, we could test our implementation for practical setups.

Efficient implementation for external memory Our preprocessing technique naturally allows an implementation based on external memory. Essentially, such an implementation has been used to carry out our experiments. We could optimize our implementation in terms of block accesses to secondary storage.

Path views Storage and retrieval of the course of a shortest path is a major requirement for practical applications. To allow for that, we could enrich our preprocessing by path information as in [JHR98].

Faster query through bi-directional search Queries against our search space graphs can be performed by simultaneously looking for a path from source and from target. By that, we can reduce the number of level edges needed to consider.

Parallelized query We could parallelize the query procedure by spreading the preprocessed data over several machines (e.g., in a computer cluster) where every machine is responsible for looking up paths in a distinct top-level component. Apart from task scheduling, communication is required only for queries with comparatively large distances between source and target.

Dynamization Due to the multi-level approach, we believe that only small parts of the preprocessed data need to be updated when an edge changes in the input graph.

Combination with other techniques We believe that we can enhance our technique by including some aspects covered by other approaches, e.g., time-dependent edge costs.

Theoretical analysis For graph minimization, we have employed a heuristics that performs well in practice. However, it remains an open question whether the construction of a minimal equivalent graph is a hard problem.

A Proofs

Lemma 2. *For any pair of component indices $i, j \in I$, $i \neq j$, and for any pair of vertices $s \in V_i$, $t \in V_j$, every path in G between s and t contains at least one vertex in S.*

Proof. If $s \in S$ or $t \in S$, the claim follows straight away. From here on, assume $s \in V_i \setminus S$ and $t \in V_j \setminus S$.

For the sake of contradiction, assume we have a path $p = \langle v_1, v_2, \ldots, v_z \rangle$ with $v_1 = s$, $v_z = t$ and $\forall x : v_x \notin S$. The path must contain at least one vertex in $V_i \setminus S$ and at least one vertex not in $V_i \setminus S$, namely, s and t. (We have $t \notin V_i \setminus S$, because $V_i \setminus S$ and $V_j \setminus S$ are disjoint by definition of S.) That given, we can choose x so that $v_x \in V_i \setminus S$ and $v_{x+1} \notin V_i \setminus S$ and get $c(v_x, v_{x+1}) < \infty$ from the path property. We conclude that even $v_{x+1} \notin V_i$, since $v_{x+1} \notin S$ according to our assumption that no separators be touched by the path.

This contradicts the requirement that edges exist only between vertices of the same component: By definition of decomposition, there must exist an $l \in I$ with $v_x, v_{x+1} \in V_l$. Since $v_x \in V_i \setminus S$, we have $v_x \notin V_l$ for all $l \neq i$; otherwise v_x would be a member of both V_i and V_l and thus contained in S. On the other hand, $v_{x+1} \notin V_i$. From this follows that no such l can exist, which proves our assumption wrong. □

Lemma 3. *For any pair of component indices $i, j \in I$, $i \neq j$, and for any pair of vertices $s \in V_i$, $t \in V_j$, every path in G between s and t contains at least one boundary vertex of each G_i and G_j, that is, at least one vertex of B_i and at least one vertex of B_j.*

Proof. We prove the containment of at least one vertex of B_i in any path between s and t. The other claim follows by symmetry.

Fix any path $p = \langle v_1, v_2, \ldots, v_z \rangle$ with $v_1 = s$ and $v_z = t$. It follows from Lemma 2 that p must contain at least one separator. Let x be the smallest index so that $v_x \in S$. We claim that also $v_x \in B_i$. To prove that, we only need to prove that $v_x \in V_i$, according to the definition of the boundary vertices.

For $x = 1$, we are through, because $v_1 = s \in V_i$. For $x > 1$, we assume, for the sake of contradiction, that $v_x \notin V_i$. Then, by definition of separation it follows that $v_{x-1} \notin V_i$, as edges may only connect vertices in the same component, and $c(v_{x-1}, v_x) < \infty$ by the path property. Consequently, there must be some $l \in I$ with $l \neq i$ and $v_{x-1} \in V_l$. We again apply Lemma 2 for s and v_{x-1} and derive the presence of a separator vertex in the subpath $p_{1 \to (x-1)}$, in contradiction to our prerequisite that x be the first separator of p. □

Corollary 4. *The separator set S is empty iff $|I| = 1$.*

Proof. ($\Rightarrow$) Assume $|I| > 1$ and $S = \emptyset$ for the sake of contradiction. Choose two component indices $i, j \in I$ so that $i \neq j$. The vertex sets V_i and V_j are not empty, so choose arbitrary $s \in V_i$ and $t \in V_j$. Because G is connected, there exists a path between s and t. From Lemma 2 follows, that this path must contain at least one vertex in S. This leads to contradiction to our assumption that S be empty.

($\Leftarrow$) For $|I| = 1$, the separator set is empty by definition: No $i, j \in I$ with $i \neq j$ exist. □

Corollary 5. *For any component index $i \in I$ and for any two vertices $s \in V_i \setminus S, t \in V_i$, any path $p = \langle v_1, v_2, \ldots, v_z \rangle$ in G that, apart from t, does not contain vertices in S, contains only vertices in V_i.*

Proof. For the sake of contradiction, assume that p contains a vertex $v_x \notin V_i$ with $x < z$. Due to our prerequisite, $v_x \notin S$. Choose j with $v_x \in V_j$. Such a j must exist by Definition 1.4. We know that $j \neq i$ due to $v_x \notin V_i$. According to Lemma 2, the path $p' := \langle v_1, \ldots, v_x \rangle$ is supposed to contain a separator, which contradicts our prerequisite. □

Corollary 6. *For any component index $i \in I$ and for any two vertices $s \in V_i, t \in V_i \setminus S$, any path $p = \langle v_1, v_2, \ldots, v_z \rangle$ in G that, apart from s, does not contain vertices in S, contains only vertices in V_i.*

Proof. Symmetric to the one for Corollary 5. □

Lemma 8. *For all $k \in \{1, \ldots, n\}$, $S^k \subseteq S^{k-1}$.*

Proof. Fix a $k \in \{1, \ldots, n\}$. Choose any vertex $v \in S^k$. We will prove that v is also contained in S^{k-1}.

By the definition of the separator set, there must exist at least two distinct component indices i, j with $v \in V_i^k$ and $v \in V_j^k$. Choose component indices $i' \in H_i^k$ and $j' \in H_j^k$ so that $v \in V_{i'}^{k-1}$ and $v \in V_{j'}^{k-1}$. Such indices must exist, since there must be at least one child component of G_i^k and G_j^k, respectively, to contain v, cf. Definitions 7.2b and 7.2c. Because of Definition 7.2a, the indices are distinct: $i' \neq j'$. So, we have two distinct components that contain v at level $k-1$: $v \in G_{i'}^{k-1}$ and $v \in G_{j'}^{k-1}$. This makes v a separator at level $k-1$, i.e., $v \in S^{k-1}$. □

Lemma 22. *For any pair of vertices $s, t \in V$ with $h(s) \neq h(t)$, and for any vertex $\nu = (v, k+1)$ of $\mathscr{G}_{s,t}$ with $k \geq 0$, the distance between ν_s and ν in $\mathscr{G}_{s,t}$ equals the distance between s and v in $G_{i[k]}^k$:*

$$d[\mathscr{G}_{s,t}](\nu_s, \nu) = d\big[G_{i[k]}^k\big](s, v).$$

Proof. (Sketch.) We only offer an intuition for the course of the proof. Figure 23 informally presents the lemma's main statement.

The definition of ν implicitly states that v is a boundary vertex for the component $G_{i[k]}^k$ at level k. Owing to that, we can prove our claim by induction over k.

The root for the induction at $k = 0$ immediately follows from the definition of the entry graph.

For $k > 0$, any path to a boundary vertex at level k must pass at least one boundary vertex at level $k-1$. We indicate, for each path between s and v in G, a corresponding path between ν_s and ν in $\mathscr{G}_{s,t}$ that is at most as long as the path in G, and vice versa. We use the induction hypothesis for level $k-1$ when we bound the path lengths. — The inductional part of the proof strongly resembles the proof of our main theorem. □

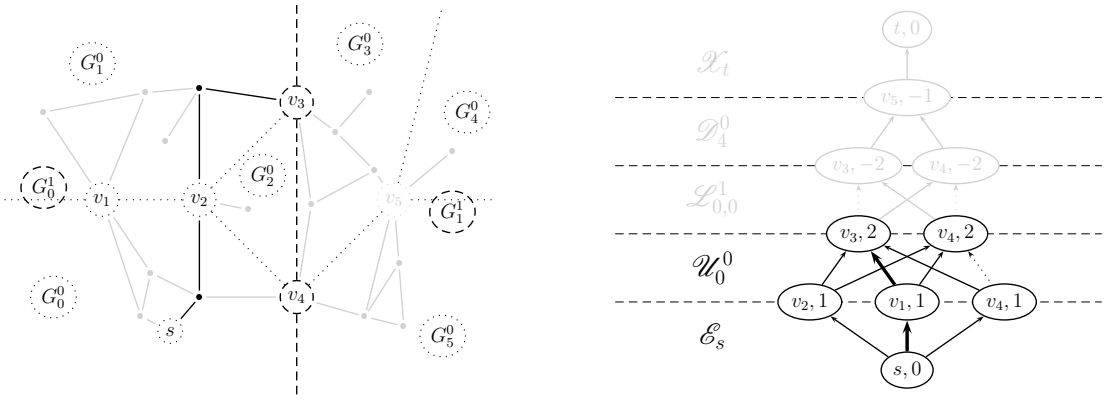

Figure 23: Illustration for Lemma 22.

The highlighted edges in the drawing at the left denote a shortest path between s and v_3 in G. The bold edges in the right-hand side drawing represent a shortest path between $(s, 0)$ and $(v_3, 2)$ in $\mathscr{G}_{s,t}$. Both paths correspond to each other and are equal in length. We can prove that such corresponding shortest paths can be found for each boundary vertex of a component that contains s.

B Document Type Definitions

```
<!ELEMENT compchildren (parent, children)>

  <!ELEMENT parent ((neighbors|universe)?)>
    <!ATTLIST parent comp-id ID #REQUIRED>
    <!ATTLIST parent level CDATA #IMPLIED>

    <!ELEMENT neighbors (separator*)>

      <!ELEMENT separator (neighbor*)>
        <!ATTLIST separator node-id CDATA #REQUIRED>

        <!ELEMENT neighbor EMPTY>
          <!ATTLIST neighbor comp-id ID #REQUIRED>

    <!ELEMENT universe EMPTY>

  <!ELEMENT children (child*)>

    <!ELEMENT child EMPTY>
    <!ATTLIST child comp-id ID #REQUIRED>
    <!ATTLIST child path CDATA #IMPLIED>
```

Figure 24: Contents of `compchildren.dtd`.
Documents that conforms to `compchildren.dtd` store information about a component's children in the hierarchical decomposition.

```
<!ELEMENT nodetocomp (n*)>
  <!ELEMENT n EMPTY>
    <!ATTLIST n x ID #REQUIRED>
    <!ATTLIST n c IDREF #REQUIRED>
```

Figure 25: Contents of `nodetocomp.dtd`.
We use a single document that conforms to `nodetocomp.dtd` to store the home component for each vertex.

```
<!ELEMENT closure (node*,end,source*)>
  <!ELEMENT node comp*>
    <!ATTLIST node node-id ID #REQUIRED>
    <!ATTLIST node type (sep|nosep) "nosep">
    <!ATTLIST node offset CDATA #REQUIRED>

    <!ELEMENT comp EMPTY>
      <!ATTLIST comp comp-id CDATA #REQUIRED>

  <!ELEMENT end EMPTY>
    <!ATTLIST end offset CDATA #REQUIRED>

  <!ELEMENT source target*>
    <!ATTLIST source node-id IDREF #REQUIRED>

    <!ELEMENT target EMPTY>
      <!ATTLIST target node-id IDREF #REQUIRED>
      <!ATTLIST target dist CDATA #REQUIRED>
      <!ATTLIST target pred-id IDREF #IMPLIED>
```

Figure 26: Contents of `closure.dtd`.
Documents conforming to `closure.dtd` store the child separator closures for a component.

```
<!ELEMENT searchspaceparts (type-desc,base-comp,
                            end-fwd,rel-comp-fwd+,rel-comp+)>
  <!ELEMENT type-desc EMPTY>
    <!ATTLIST type-desc type (up|down|level)>

  <!ELEMENT base-comp n*>
    <!ATTLIST base-comp comp-id CDATA #REQUIRED>

    <!ELEMENT n EMPTY>
      <!ATTLIST n x ID #IMPLIED>
      <!ATTLIST n k (b|r|i)>
      <!ATTLIST n p CDATA #REQUIRED>

  <!ELEMENT end-fwd EMPTY>
    <!ATTLIST end-fwd offset CDATA #REQUIRED>

  <!ELEMENT rel-comp-fwd EMPTY>
    <!ATTLIST rel-comp-fwd comp-id CDATA #REQUIRED>
    <!ATTLIST rel-comp-fwd offset CDATA #REQUIRED>

  <!ELEMENT rel-comp (n+,s+)>
    <!ATTLIST rel-comp comp-id CDATA #REQUIRED>

    <!ELEMENT s t*>
      <!ATTLIST s x IDREF #IMPLIED>
      <!ATTLIST s p CDATA #REQUIRED>

      <!ELEMENT t EMPTY>
        <!ATTLIST t x IDREF #IMPLIED>
        <!ATTLIST t p CDATA #REQUIRED>
        <!ATTLIST t d CDATA #REQUIRED>
```

Figure 27: Contents of `searchspaceparts.dtd`.
A document that conforms to `searchspaceparts.dtd` stores a set of related upward, downward or level graphs. The layout of these documents has been optimized for size.

Index

References

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition edition, 2001.

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

[Fre87] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16(6):1004–1022, 1987.

[GH05] Andrew Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005)*. SIAM, 2005.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and engineering computation. 1994.

[Gut04] Ronald J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *ALENEX/ANALC*, pages 100–111, 2004.

[Hol03] Martin Holzer. Hierarchical Speed-up Techniques for Shortest-Path Algorithms. Master's thesis, Dept. of Informatics, University of Konstanz, Germany, February 2003.

[HPS+05] Martin Holzer, Grigorios Prasinos, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Engineering Planar Separator Algorithms. In *Proceedings of the 13th European Symposium on Algorithms (ESA), 2005*. Springer-Verlag, October 2005.

[HSW06] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multi-level overlay graphs for shortest-path queries. In *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2006. To appear.

[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 Transactions on 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 Stanley Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 76–84. IEEE Computer Society, 1996.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc (2nd ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1992.

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

[LT80] Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, 1980.

[MCH] MPICH – a portable implementation of MPI. `http://www-unix.mcs.anl.gov/mpi/mpich1/`.

[NM99] Stefan Näher and Kurt Mehlhorn. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999. `http://www.algorithmic-solutions.com`.

[Ros] Kristoffer Rose. Generating fast validating XML processors. `http://temppeli.org/doc/flexml/html/paper.html`.

[SS05] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *Proceedings 17th European Symposium on Algorithms (ESA)*, volume 3669 of *Springer LNCS*, pages 568–579. Springer, 2005.

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using multi-level graphs for timetable information in railway systems. In *Proceedings 4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, volume 2409 of *LNCS*, pages 43–59. Springer, 2002.