Dynamic Speed-Up Techniques for Dijkstra's Algorithm

Diploma Thesis

by

Reinhard Bauer

July 31, 2006

Supervisor:	Prof. Dr. Dorothea Wagner	Faculty of Informatics
Co-Referee:	Prof. Dr. Rudolf Scherer	Faculty of Mathematics
Advisor:	Dipl.-Inform. Daniel Delling	Faculty of Informatics
Advisor:	Dipl.-Math. Martin Holzer	Faculty of Informatics
Advisor:	Dr. rer. nat. Frank Schulz	Faculty of Informatics

Universität Karlsruhe
Faculty of Mathematics

Acknowledgements

I would like to thank Prof. Dr. Dorothea Wagner for the opportunity to work on an interesting and recent topic; Dr. Frank Schulz, Martin Holzer and Daniel Delling for their steady helpfulness and the plenty of time they spent supporting me; Daniel Gentner for proof-reading this thesis; finally I want to thank Prof. Dr. Rudolf Scherer, who told me that he did not want to be thanked for doing his duty, for always doing his duty in his friendly and helpful way.

Declaration

Contents

1 Introduction

Have you ever questioned the quality of a route-planning system's output? You might be surprised that popular route-planners do not always compute optimal itinery. Figure 1 shows an example of a non-optimal route computed by an automated route-planning system. When talking of such a system we are thinking of the following, concrete application: an online-working route-planner which has to answer a huge number of queries each concerning the fastest connection between two places and each to be answered by an exact solution.

In most cases, the reason for the non-optimality of the given routes is the usage of heuristic algorithms to speed up the query. These heuristics do not guarantee the optimality of the computed path. During the last years strong efforts have been made to develop fast algorithms that also compute exact results. The achievements on this area, combined with the advance in actual hardware, now make it imaginable to use output-optimal methods in commercial applications.

Unfortunately, most of the established output-optimal algorithms are *static* in the following sense: in order to operate they need extra data computed by a time-consuming preprocessing. This preprocessing can last days for huge input data and often only fast computers are able to perform it. Therefore these techniques hardly arrange with changes in reality such as traffic jams, road works or canceled trains even if these changes are 'small' compared with the whole underlying graph. At worst this would effect in a complete recomputation from scratch after the change of only one edge on the graph.

Therefore solutions are needed that compute optimal routes but are flexible enough to deal with changes in the algorithm's input. This work shows how to efficiently recompute the preprocessed data of some of the recent output-optimal algorithms without starting from scratch. As in real-world data those changes normally accumulate (think of a traffic jam that slows down surrounding roads) we demand of the update routine to process several updates in batch and to take advantage of the closeness of the updates if possible.

Overview

In Chapter 2 the topic of this work, the *(dynamic) single-source single-target shortest path problem*, is formally defined and *Dijkstra's algorithm*, the basic algorithm solving that problem, is introduced. As Dijkstra's algorithm is fundamental for all later presented

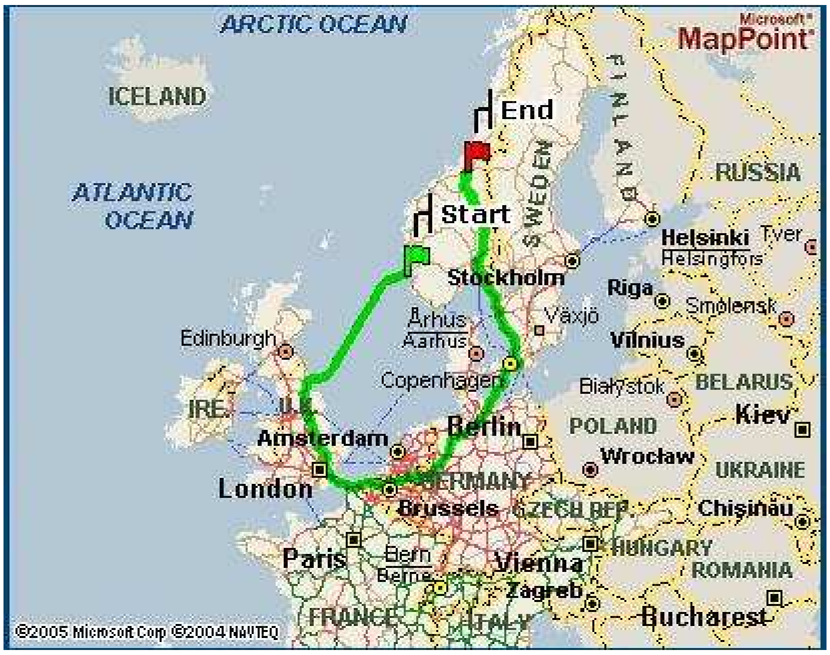


Fig. 1.1: Shortest path from Hausgesund to Trondheim (both in Norway) computed by Microsoft MapPoint in 2005

techniques we analyze important properties of the algorithm. The chapter closes with a dynamic variant of Dijkstra's algorithm.

The next chapter gives an overview of the techniques used to speed up Dijkstra's algorithm while granting the optimality of the result. Furthermore, we sketch a dynamic update method for most techniques.

Chapter 4 precisely describes the preprocessing of a speed-up technique using *reach-bounds*. This is a relatively new, promising method by Gutman (2004) and improved by Goldberg (2005).

In Chapter 5 we present our main contribution to the problem: an efficient method to update the preprocessing used by the reach-bound speed-up technique. The method only slightly increases the space-consumption of the preprocessed data, benefits from batch-updating several changes and recomputes the same data as a full recomputation from scratch would do. Chapter 6 shows the results of some own experiments concerning reach-bounds.

The last chapter summarizes the results and points out several fields of research, the author considers promising for a further development on the field of this topic.

2 Fundamentals

In this chapter we formally define the fundamentals of the topic of this thesis. The most important notions are: single-source single-target problem, single-source all-targets problem, speed-up technique and update of a speed-up technique. References to related problems and related work are being given. Then Dijkstra's algorithm, the most important algorithm for the solution of the single-source single/all-target(s) problem is described and some important properties of that algorithm are analyzed. Finally, a solution for the dynamic single-source all-targets problem is given.

2.1 Presentation of the problem

Let $G = (V, E)$ be a weighted, directed graph with n vertices, m edges and lengths $len : E \rightarrow \mathbb{R}_0^+$. A *path* with *source* s and *target* t (or shorter an *s-t-path*) in G is a k-tupel of vertices $P = (s = u_0, u_1, \ldots, u_{k-1} = t)$ where for every i between 1 and $k-1$ the edge (u_{i-1}, u_i) exists in E. The *length* of P is defined as $len(P) := \sum_{i=1}^{k-1} len(u_{i-1}, u_i)$. An s-t path is called a *shortest path* if its length is minimal among the lengths of all s-t-paths. Given two vertices s and t the *distance* from s and t is the length of a shortest s-t-path.

The most fundamental problems when dealing with shortest paths are:

single-source single-target. Given two vertices s and t, find a shortest s-t-path.

single-source all-targets. Given a vertex s. For each other vertex t in the graph, find a shortest s-t-path.

all-pairs shortest-paths. For each $(s, t) \in V^2$, find a shortest s-t-path.

A graph is called *connected* if for each $(s, t) \in V^2$ an s-t-path exists. A graph is called *dense* if the number of its edges is close to the maximal number of edges. A graph is called *sparse* if it has only few edges. We call a class of graphs *large* if one can only afford the consumption of $O(n)$ memory.

In this work, we concentrate on the single-source single-target problem on connected, large and sparse graphs. We can solve this problem efficently in $O(m \log n)$ time using Dijkstra's algorithm which we present in the next section. As even this asymptotically good runtime needs too much time for very large graphs, various variants of Dijkstra's algorithm have been developed that improve its runtime, often using additional, preprocessed data. We call such algorithms *speed-up techniques*. Most of these speed-up techniques work as follows: first, a preprocessing step is performed. The input of that step consists of the graph, the graph's edge lengths and sometimes additional data, attached to the graph. Then, using the preprocessed data, concrete single-source single-target queries are answered, most times significantly faster than through Dijkstra's algorithm.

We want to emphasize that speed-up techniques work exactly concerning the problem's solution but are heuristic in the runtime. Therefore a query performed by a speed-up technique may even take more time than a query performed by Dijkstra's algorithm.

An *update* on the graph is a change in the graph's length function. If for each edge of the graph the new length of the edge is greater (lower) or equal to the old length the update is called *incremental* (*decremental*). If both, at least one edge with increased and one with decreased length exists, the update is called *fully dynamic*. Further, we will abbreviate 'update of an edge's length' with 'edge update'.

We regard edge deletions and edge insertions as special case of updated edges: when we want to delete an edge we simply set its length to infinity. As we want to keep our proofs simple we assume that the graph remains connected after an edge has been deleted. To insert an edge we consider it as already existent with length infinity in the unaltered graph and set the edge's length in the altered graph to the value given by the update. This proceeding has to be justified separately for each speed-up technique but mostly works well: unless stated otherwise the preprocessing of the speed-up techniques does not change because of the insertion of an edge with length infinity.

The problem this work is about is that of efficiently updating the preprocessed data of a speed-up technique after the underlying graph has been updated: let G be a graph with non-negative edge lengths len_{old} and an altered (non-negative) length function len_{new}. Further let D_{old} (D_{new}) be the data computed in the preprocessing step of a speed-up technique using len_{old} (len_{new}). We say an algorithm $alg(G, len_{old}, len_{new}, D_{old})$ is an *exact recomputation* of $(G, len_{old}, len_{new}, D_{old})$ if its output is D_{new}. We say (very fuzzy) an algorithm $alg(G, len_{old}, len_{new}, D_{old})$ is a *quality preserving recomputation* of $(G, len_{old}, len_{new}, D_{old})$ if its output is as good as D_{new} with respect to the runtime of the queries speed-up technique.

Related work. Further reading on the static case of each speed-up technique can be found at the beginning of the according section in the next chapter. Most of the speed-up techniques described in this thesis are fairly new. Therefore only little research has been made on the dynamic case of these techniques. Fundamental thoughts about benchmarking dynamic shortest paths algorithms were published by Ramalingam in [RR96]. The dynamic update of shortest paths trees has been studied by Frigioni, Marchetti-Spaccamela and Nanni in [FMSN96], [FMSN98] and [FMSN00]. A solution for the dynamic update of geometric containers was given by Wagner, Willhalm and Zaroliagis in [WWZ04]. To our best knowledge no papers are available for the dynamic update of multi-level graphs, highway hierarchies and reach-values so far.

Related problems. We consider only graphs with non-negative edge lengths. This seems to be a small restriction, but enormously scales down the complexity of the single-source (single/all)-target(s) problem. In fact, the general problem allowing negative edge lengths is NP-hard, therefore efficent solutions are not likely to exist. The main problem when dealing with negative edge lengths is the existance of negative cycles. If no negative cycles exist, a problem with negative edge lengths can be transformed to one with non-negative edge lengths in polynomial time. [AMO93] gives a good overview of fundamental shortest path problems and algorithms.

An area in which proceedings that help find solutions for the recomputation of speed-up techniques may be found is the dynamic all-pairs shortest-paths problem. We refer to Demetrescu's and Italiano's paper [DI05] for a list of related work on the topic and an interesting, new algorithm solving this problem.

2.2 Canonical Shortest Paths

At this point, we want to remind the reader that a shortest path is not necessarily unique. While some applications benefit from obtaining a list of all possible shortest paths for a given problem many others are sufficiently solved by computing just one of such paths. The requirement for knowing all shortest paths occurs particularly when an algorithm first pre-selects some interesting paths and later determines the one to use.

In order to simplify the mathematical treatment of a shortest-path algorithm it is convenient to ensure the uniqueness of the shortest path for a given problem. We could do so by adding fractions to each edge, all so small that they do not have any influence further than determining which of all shortest paths for a given problem to choose. This approach seems to be very uncomfortable because of the occuring numerical problems. Another possibility is to use a deterministic rule that decides which of a set of paths to take.

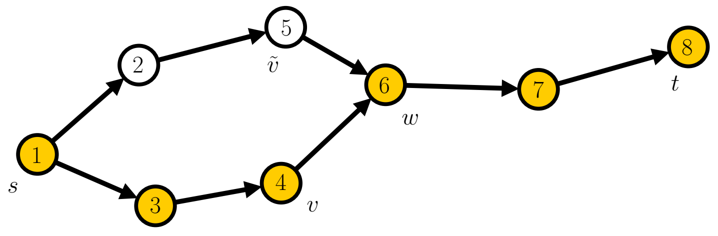


Fig. 2.1: Two shortest paths from s to t. The numbers within the vertices represent the canonical ordering, the orange vertices induce the canonical shortest path

We use an injective mapping from every vertex to $\mathbb{N}$ to determine one path from the set of all shortest s-t-paths. We call such a path a *canonical shortest path*:

Definition 1 (Canonical Shortest Paths) An injective mapping $o : V \to \mathbb{N}$ is called a *canonical ordering* of V.

Given a canonical ordering o. A shortest path P with start vertex s and end vertex t is said to be a *canonical (shortest) path* if for any shortest path $\tilde{P}$ between s and t follows:

Let $(w, \dots, t)$ be the maximal subpath ending at t that P and $\tilde{P}$ have in common. Further, let v ($\tilde{v}$) be the predecessor of w on P ($\tilde{P}$). Then

$$o(v) < o(\tilde{v}).$$

Figure 2.1 gives an example for choosing a canonical path in a graph with two shortest paths between source and target. Note that a subpath consisting of only one vertex may be possible and that a canonical ordering is implicitly given by the order in which the vertices are put down in the computer's memory.

When we describe the highway hierarchies technique (section 3.6) and reach-based pruning (section 3.7, chapter 4, chapter 5) **our aim is to compute only canonical shortest paths** and therefore **we refer to canonical shortest paths as shortest paths**. We want to stress that all speed-up techniques described in this work can be modified for handling all shortest paths. We do not do that in order to increase the readability of the text and to emphasize the real idea behind the algorithms.

Further we will use the following properties of canonical paths without explicitly mentioning them:

Uniqueness The most important property of canonical paths is that they are unique.

Existence Furthermore we will use that on a connected graph for all pairs of vertices s, t the existence of a canonical path is guaranteed.

Inheritance The subpaths of canonical paths are also canonical paths.

Computability In order to make Dijkstra's algorithm choose the canonical out of all shortest paths the algorithm has only to be slightly adapted. The changes will be shown in the next section.

Proof 1 (Properties of Canonical Shortest Paths)
The uniqueness follows directly from the definition of the canonical shortest path.

To proof the inheritance-property of shortest paths we consider a canonical shortest path $P = (s_1, s_2, \dots, s_n, u_1, \dots, u_m, t_1, \dots t_k)$ and assume that a canonical shortest path $Q = (s_i, \dots, s_n, v_1, \dots, v_l, t_1, \dots, t_k)$ with $u_1 \neq v_1$ and $u_m \neq v_l$ exists. Then must $o(v_l) < o(u_m)$ and since $\tilde{Q} = (s_1, s_2, \dots, s_n, v_1, \dots, v_l, t_1, \dots t_k)$ is also a shortest path can P not be a canonical shortest path. Therfore all subpaths of a canonical shortest path that end with the same vertex are also canonical shortest paths. The same argumentation holds for all subpaths that start with the same vertex. Applying both directions we know that all subpaths of canonical shortest paths are also canonical.

To show the existence of a canonical shortest path we describe a construction of it. Given two vertices s and t and the set S_0 of all shortest s-t-paths (S_0 can be constructed by a variant of Dijkstra's algorithm). Starting with $i = 0$ we iteratively construct a sequence of sets of paths S_i: Let $\tilde{P}_i = (p_0, p_1, \ldots, p_k = t)$ be the maximal subpath that ends with t and that all paths in S_i have in common. We want S_{i+1} to consist of exactly all paths in S_i for which the canonical ordering of the predecessor vertex of p_0 is minimal among the canonical orderings of predecessor vertices of p_0 of paths in S_i. The construction stops at iteration step j if $\#S_j = 1$. It is easy to see that the path included in S_j is a canonical s-t-path.

■

2.3 Dijkstra's Algorithm

Dijkstra's algorithm is one of the most fundamental algorithms for the single-source single/all-target(s) shortest path problem. The output of the algorithm is a list, providing for every vertex v the predecessor of v on the shortest path from the source to v and the length of that path.

The algorithm has to store the output and some extra information on its progress traversing the graph: it maintains for each vertex v a distance label $d(v)$, a parent vertex $p(v)$ and a status marker representing one of the states *unvisited*, *visited* and *finished*. All status markers are initialized to be unvisited, the distance labels to be infinity and the parent to be *nil*. After that, the source vertex is set to be *visited* and its distance is set to zero.

We provide a priority queue that contains all visited vertices keyed by the distance label, the lower the better. While there are visited labels the algorithm removes the one with the smallest distance label from the queue, marks it as finished and *relaxes* all its outgoing edges.

The relaxation of an edge (v, w) goes as follows: first it is tested if $d(w) > d(v) + \text{len}(v, w)$ or $(d(w) = d(v) + \text{len}(v, w)$ and $o(v) < o(p(w)))$. If that is true, the vertex w is marked as visited, the parent of w is set to v, and the value of the distance label is changed to $d(v) + len(v, w)$. Finally, if the vertex w was unvisited before, it will be inserted into the priority queue.

Note that the condition's second possibility $d(w) = d(v) + \text{len}(v, w)$ and $o(v) < o(p(w))$ is not a classical part of Dijkstra's algorithm. We added it here to ensure that only the canonical out of all shortest paths is chosen.

When a single-target problem is queried, the algorithm can break after the target-vertex has been marked as finished.

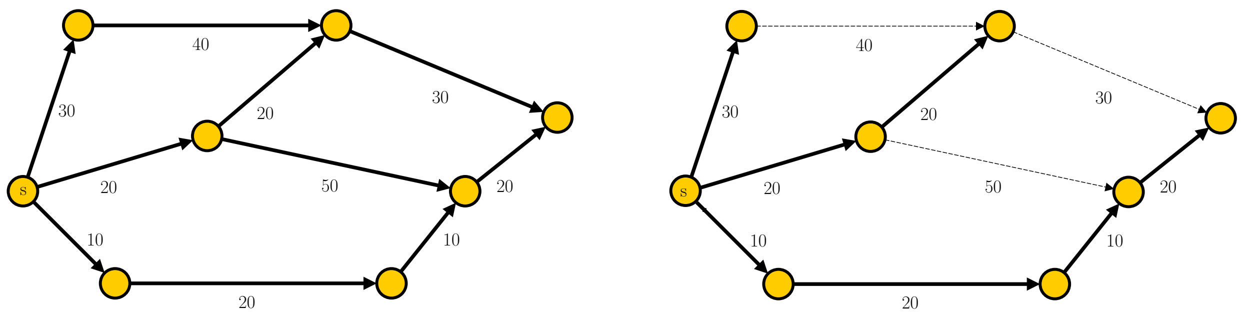


Fig. 2.2: Weighted Digraph (left side) and its shortest paths tree rooted at s (right side)

Algorithm 1: DIJKSTRA

```
1  for $v \in V \setminus \{s\}$ do $d(v) := \infty$
2  $d(s) := 0$
3  insert $s$ in $Q$
4  while $Q \neq \emptyset$ do
5      remove minimal Element $v$ from Q
6      mark $v$ as finished
7      for $e := (v, w) \in E$ do
8          if $w$ is not marked as finished then
9              if $d(v) + len(e) < d(w)$ or $(d(v) + len(e) = d(w)$ and $o(v) < o(p(w)))$
               then
10                 $d(w) := d(v) + len(e)$
11                 $p(w) = v$
12                 if $e \notin Q$ then insert $w$ in $Q$
```

Proof 2 (Correctness)
The correctness of Dijkstra's algorithm relies on the fact, that at each step, the tentative path from the source to the vertex minimal in the priority queue already is a shortest path. A complete proof can be found in [AMO93].

Given a canonical ordering o and an s-t-query. In order to prove that only canonical shortest paths are computed we assume that at least one path $P = (u_1, \ldots, u_n)$ is contained in the shortest-paths tree that is not canonical. Let Q be the canonical s-t-path and $\tilde{P} = (u_k, \ldots, u_n)$ be the maximal subpath ending at t that both paths have in common. Let q be the predecessor of u_k on Q. Then is $o(q) < o(u_{k-1})$ and the adaption of Dijkstra's algorithm would not have settled u_k via u_{k-1}.

■

Later, we will use the following notation: given the tentative shortest paths tree at an arbitrary step of Dijkstra's algorithm. Then, the *finished part* of that tree is the tree's subgraph induced by all vertices marked as finished.

Runtime. Even if the worst-case runtime of the algorithm is $O(n^2)$ on dense graphs, one can do much better on sparse graphs. The choice of the priority queue is a crucial point for the performance. If the edge lengths are natural numbers bounded by a constant C, Dials Implementation needs $O(m + nC)$, Johnson's Implementation $O(m \log\log C)$ runtime. Binary Heaps (runtime of $O(m \log n)$) and Fibonacci-Heaps (runtime of $O(m + n \log n)$) are the best performing priority queues that are known for general sparse graphs. [CLL90] contains a precise description of all these algorithms.

SetDijkstra. Dijkstra's algorithm can also be used to find for each vertex on the graph the shortest path from the nearest of a set of given vertices. We will face that problem in Chapter 5 as a subproblem of a speed-up technique's recomputation. A practical application is to check the coverage of infrastructural facilities.

SetDijkstra works as follows: given a set of 'source vertices' we run Dijkstra's algorithm, but initialize it using all source vertices instead of using only one source vertex. The resulting output contains for each vertex the predecessor on the way from the nearest source vertex. We refer to that variant as *SetDijkstra*.

2.4 Updating Shortest Path Trees

Based on the algorithms described by Frigioni, Marchetti-Spaccamela and Nanni in [FMSN00], we present an algorithm that updates an existing shortest paths tree after a set of edges has been updated. We assume that the shortest paths tree is identified by a label containing the tree-predecessor of each vertex. Furthermore the distance of each vertex to the source shall be given. The update algorithm proceeds much like Dijkstra's algorithm.

Notation. Given a graph $G = (V, E)$ with length function $len_{old} : E \rightarrow \mathbb{R}^+$ and a subset of edges U with updated edge lengths. The new length function is denoted by $len_{new} : E \rightarrow \mathbb{R}^+$. Let T_{old} be the shortest-paths tree on G with respect to len_{old} rooted at a vertex s. Let T_{new} be the tentative shortest paths tree computed by our algorithm. Initially T_{new} equals T_{old}. Let $P_{old}(v)/P_{new}(v)$ be the predecessor of the vertex v on T_{old}/T_{new}. For each $v \in V$ let $dist_{old}(v)/dist_{new}(v)$ be the distance from s to v with respect to the old/new length function. With $D(v)$ we denote the tentative distance from s to v currently computed by our algorithm. Initally $D(v)$ equals $dist_{old}(v)$.

Initialization. At the initialization step we update the distances of the target vertices of edges in U. In order to do that we provide a priority queue H containing all edges (u, v) of U keyed by the distance label $D(u)$. We iteratively remove the minimal edge (u, v) from H and set $D(v) := D(u) + len_{new}(u, v)$ if $D(v) > D(u) + len_{new}(u, v)$ or $P_{new}(v) = u$. We update the priority of an edge in the queue if the distance label of the according source vertex has been updated.

We maintain a second priority queue Q that contains each vertex v with altered and tentative distance label. The priority of v is $D(v)$. When we change the distance label of a vertex v which is not contained in Q we insert v into Q (with the new distance

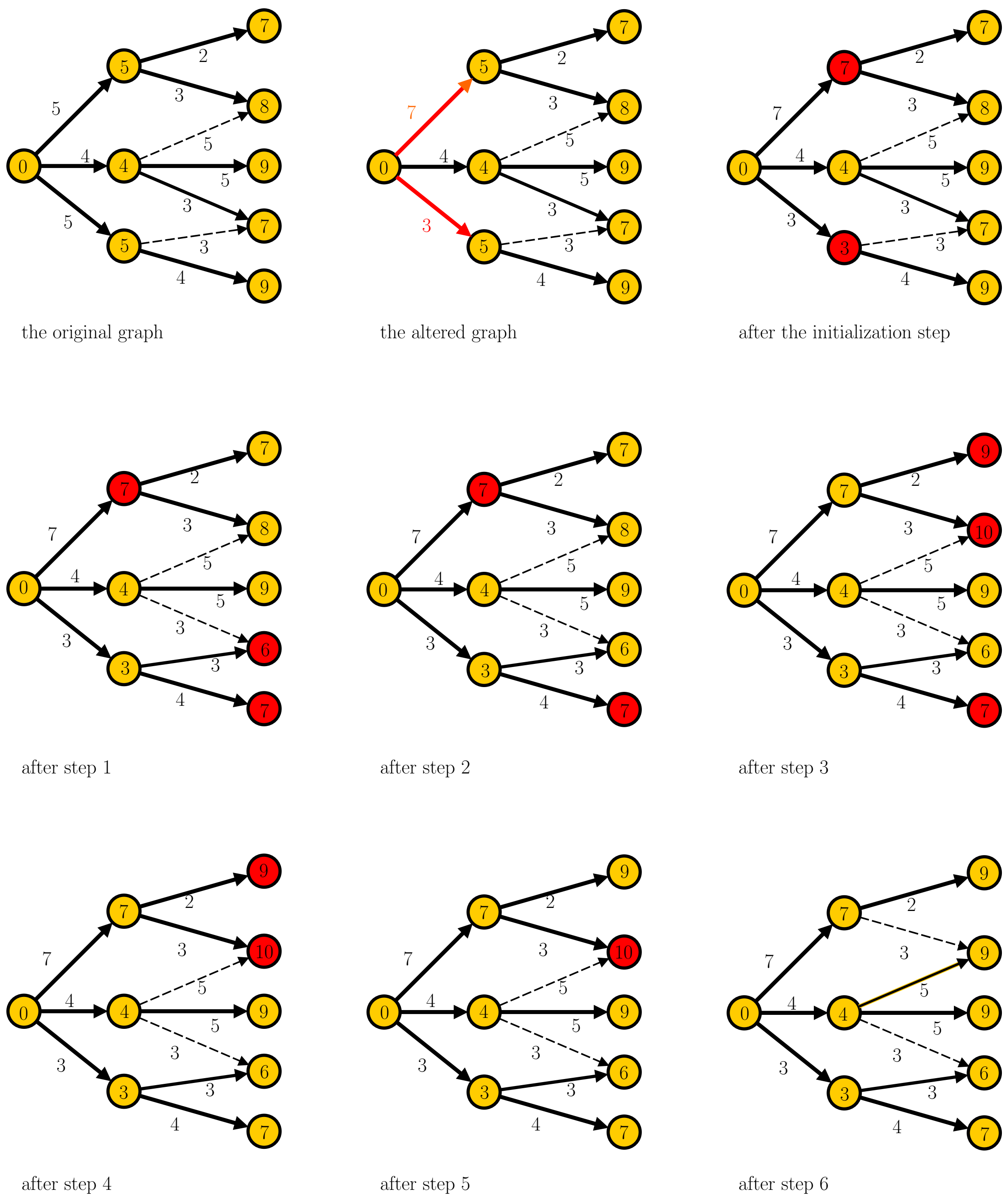


Fig. 2.3: Example for the update of a shortest paths tree. Continous lines represent edges on the (tentative) shortest paths tree, dashed lines the other edges. The vertices currently in the queue are drawn red.

as priority) and the original distance of the vertex (the actual value of $D(v)$ before the change) is saved. We call that saved value $D_{insQueue}(v)$. Each time the distance label of this vertex is changed, the priority of the vertex has to be changed to the new distance.

Main Algorithm. At the main algorithm we remove the minimal vertex m from the queue and 'process' it. We iteratively repeat that until the queue is empty. The processing of a vertex depends on the relation of its original and its actual distance:

- $\mathbf{D_{insQueue}(v) = D(v)}$. If the original distance equals the actual one, nothing is to be done.
- $\mathbf{D_{insQueue}(v) > D(v)}$. If the distance label of m has decreased we check for every outgoing edge (m, t) if the path to t containing the edge (m, t) is shorter than the shortest path to t found so far. In that case we update distance label $D(t)$ and predecessor $P(t)$ of t accordingly.
- $\mathbf{D_{insQueue}(v) < D(v)}$. If the distance label of m has increased, we check every incoming edge (s, m) if a path using that edge is shorter than the shortest path to m found so far. In that case we update distance label $D(t)$ and predecessor $P(m)$ of m accordingly. Now the distance label of m is correct and we search every outgoing edge (m, t) that is part of the tentative shortest paths tree T_{new}. For each found edge (m, t), we set the distance label of t to $dist(m) + len(m, t)$.

after processing a vertex it is removed from the queue Q. The algorithm terminates when Q is empty. Figure 2.3, page 15 shows the algorithm at work on an example graph.

Comparison with the algorithm in [FMSN00] Though the main ideas of the former described algorithm are the same as that in [FMSN00] we want to list the differences:

- The recomputation of our algorithm handles not only one edge update per time but is a fully dynamic algorithm.
- In [FMSN00] the existance of a special assignment of each edge to one of its end-vertices is used in combination with a datastructure that sorts a subset of all edges incident to a vertex in order to guarantee better worst case runtime.
- The algorithm in [FMSN00] operates on undirected graphs.

Coarse sketch of the proof of correctness

The correctness for incremental or decremental updates is proven completely analogous to the proofs in [FMSN96, RR96].

The proof of correctness for the fully dynamic case consists of two steps: first it is shown that the algorithm would work correct if the priority of each vertex v in the queue Q was the correct new distance of v. That sub-proof works much like the proof of correctness for Dijkstra's algorithm.

The second step is to show that the order in which the vertices are removed from the queue does not have any influence on the final values of D_v and (if shortest paths are are supposed to be unique) has no influence on the final values of the shortest paths tree predecessors $P(v)$. The author wants to point out that he has not finished that sub-proof in detail.

Algorithm 2: UPDATE DIJKSTRA

```
   input: Graph G, len(·), Distance[], Predecessor[]

   /* init                                                          */
1  forall edges e in update set U do
2      insert e in queue H with priority distance[e.source]
3  while queue H is not empty do
4      edge e := get minimal element from H
5      remove minimal element from H
6      if predecessor[e.target]=e.source or distance[e.target]>distance[e.source]+len[e]
       then
7          UPDATE DISTANCE(e.source, e.target)
8          forall h in H with h.source=e.target do
9              update H-priority of h

   /* step down the tree                                             */
10 while queue Q is not empty do
11     node n:=get minimal element from Q
12     remove minimal element from Q

13     if oldDistance[n]<distance[n] then
14         forall edges e with e.target=n do
15             if distance[n]>distance[e.source]+len[e] then
16                 UPDATE DISTANCE(e.source, e.target)
17         forall edges e with (e.source=n and predecessor[e.target]=n) do
18             UPDATE DISTANCE(e.source, e.target)

19     if oldDistance[n]>distance[n] then
20         forall edges e with e.source=n do
21             if distance[e.target]>distance[e.source]+len[e] then
22                 UPDATE DISTANCE(e.source, e.target)
```

Algorithm 3: UPDATE DISTANCE(fromNode, toNode)

```
1 if not Q contains toNode then
2     oldDistance[toNode]:=distance[toNode]
3 distance[toNode]:=distance[fromNode]+len[(fromNode,toNode)]
4 predecessor[toNode]:=fromNode
5 if not Q contains toNode then
6     insert toNode into Q with priority distance[toNode])
7 else
8     change priority of toNode in Q to distance[toNode])
```

Heuristic variant to improve a batch update.

At this point we want to stress that the order how vertices are removed from the queue does not influence the correctness of the algorithm. It is a heuristic strategy to improve the runtime. Our strategy works well on decremental updates: the algorithm proceeds like Dijkstra's algorithm would do but shrinks the processed part of the graph if possible. However, on incremental or fully dynamic updates there are many cases where the algorithm does not perform better than iteratively recomputing the shortest paths tree edge-update by edge-update and performs worse than a full recomputation from scratch. Figure 2.4 shows such an example.

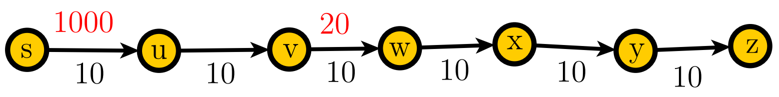


Fig. 2.4: Graph and shortest-paths tree rooted at s. The red numbers represent updated edge lengths. The optimal order to process the vertices is: u,v,w,x,y,z. Our update-algorithm would process the vertices in the following order: w,x,y,z,u,v,w,x,y,z.

To improve the performance on incremental or fully dynamic updates we propose a slight change in the algorithm: the priority of an edge (u, v) in the initialization queue H is the distance $D(v)$ of (u, v)'s target vertex instead of its source vertex. At the initialization only the first edge in H is processed, then the main algorithm is performed. The main algorithm almost works as described. The only difference is that, before a minimal vertex m from the queue is processed, the initialization step is performed for all elements of the queue H with H-priority lower than the Q-priority of m.

3 Speed-Up Techniques

In this chapter we describe several techniques that are used to speed up Dijkstra's search when solving a single-source single-target problem. All described techniques except bidirectional search and a goal-directed search variant require a preprocessing step to compute data which is later used to speed up single-source single-target queries.

Goal-directed search and the landmark technique, which is a special case of goal-directed search, alter the lengths of the original graph's edges in a way that preserves the property that shortest paths from source to target remain shortests paths but 'direct' Dijkstra's algorithm to arrive at the target while visiting fewer useless vertices.

The main idea of multi-level graphs and highway hierarchies is to build a new graph whose shortest paths correspond to shortest paths on the original graph. The new graph is built in a manner that aims to minimize the number of vertices visited by Dijkstra's algorithm. This is supported by a set of special rules which edges (not) to relax.

The edge-label technique and reach-based pruning attach additional data to each edge or vertex. This data can be used to identify branches of a shortest-paths tree that are not relevant for the solution of a given single-source single-target problem. Therefore these branches can be omitted, resulting in a faster search.

Changing the underlying graph can result in a change of the preprocessed data. The preprocessing is usually very time-consuming and a complete recomputation is often not possible. Therefore it is important to find procedures which efficiently update the preprocessing without recomputing from scratch when dealing with this situation. We will show such *dynamic update strategies* for most of the described speed-up techniques.

3.1 Bidirectional Search

3.1.1 Query

A very common speed-up technique for the single-source single-target shortest paths problem is the *bidirectional search.* This technique simultaniously performs two searches. The first, a normal Dijkstra's algorithm starts at the source and is called the *forward search.* The second is rooted at the target and is also a Dijkstra's algorithm, but applied to the reverse graph, which is the graph with the same vertex set and the reverse edge set $\overline{E} = \{(u, v) \mid (v, u) \in E\}$. We call it the *backward search.* The algorithm terminates

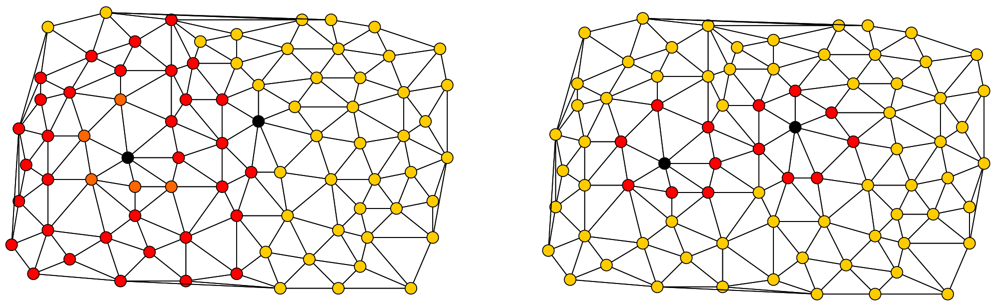


Fig. 3.1: Vertices visited (shown in red) by Dijkstra's algorithm (left) and bidirectional search (right)

when one vertex v is marked as finished by both directions. The shortest path between source and target is composed by the shortest path from source to v found by the forward search and the shortest path from v to target found by the backward search.

In [GKW05] Goldberg proposes a better stopping criterion: stop the algorithm when the sum of the minimum labels of visited vertices for the forward and reverse searches is at least the length of the shortest path seen so far.

Although performing the two searches completely simultanious would be possible on multi core/processor machines, alternating strategies must be used when using a single processor machine. A simple approach is to swap to the contrary direction every time after visiting a vertex. Another possibility is to keep the minimum distance label from visited vertices of the forward search approximately equal to the minimum distance label from visited vertices of the backward search. In order to do that the bidirectional algorithm swaps to the contrary search when the distance label of the minimal queue vertex is greater than the distance label of the minimal vertex of the contrary queue. We call this alternating strategy *distance balanced*.

The reason why this technique achieves an improvement of the runtime is a very intuitive one. The set of vertices which are visited by Dijkstra's algorithm can be imagined as a ball surrounding the source of the search. The unidirectional search needs one ball with the distance from source to target as radius. The bidirectional search on the other hand needs two balls with only half the radius, each. This diminishes the visited area, which is nothing else than the number of visited nodes.

Note that this technique can be combined with several other speed-up techniques and is an integral part of the later shown speed-up technique using reach values.

3.1.2 Dynamic Update

This speed-up technique requires no preprocessing and therefore no data has to be re-computed after altering a graph.

3.2 Goal-Directed Search

The goal-directed search, which is also called A* was introduced in [HNBR68], the description here is based on [WW06]. Its main idea is to stretch the ball of vertices visited by Dijkstra's algorithm in the direction of the target. This way, many vertices useless for the solution of a given problem will not be visited and an improvement in the algorithm's runtime is achieved.

3.2.1 Query

The search is a normal Dijkstra's algorithm but performed on an altered graph. The vertex and the edge set of the original graph stay the same but the lengths of the edges are altered in the following way:

The goal-directed search uses additional data in form of a function from the graph's vertices to reals. This function can differ for different targets. In this context, we call such a function a *potential function* and denote it by p. The new length of an edge (u, v) is assigned to $len_{new}(u, v) = len_{old}(u, v) + p(v) - p(u)$.

Note, that a concrete implementation of the goal-directed search does not need to alter the underlying graph at initialization. It only has to add the difference of the potential function $p(v) - p(u)$ to the length of (u, v) when this edge is relaxed. This way, only the edges which get relaxed during the search have to be considered to the change, which improves the runtime of the algorithm. Furthermore the search is easier to combine with other algorithms if the underlying graph stays the same.

Remember that Dijkstra's algorithm can only be applied if the underlying graph is free from negative cycles. We ensure that by claiming all new edge lengths to be non-negative. A potential function granting that property is called *feasible*:

Definition: given a weighted graph $G = (V, E)$ and a length function $len : V \to \mathbb{R}_0^+$, a potential function $p : V \to \mathbb{R}$ is called feasible if $len(u, v) - p(u) + p(v) \geq 0$ for all edges $(u, v) \in E$.

To find feasible potential functions it is useful to search tight lower bounds for the distance to the target vertex t: if $p(t) \leq 0$ then $p(v)$ is a lower bound for the distance from v to t. Hence, we can shift every feasible potential p to gain a new one $p_{new}(v) = p(v) - p(t)$ which is a valid lower bound and will result in the same search (will visit the same vertices in the same order). As tighter lower bounds will push the search more into the direction of the target, the main aim is to search those good potentials. A simple trick to extract a better potential function from a set of others is to combine them by taking the maximum:

If $p_1, p_2, \dots, p_n$ are feasible potential functions, then $p(v) = \max\{p_1(v), p_2(v), \dots, p_n(v)\}$ is a feasible potential function.

3.2.2 Correctness

Now, we are going to check the correctness of the algorithm: for each path $P = (s = v_1, v_2, \dots, v_n = t)$ on the graph the length of the path applying the old edge lengths differs from the length of the path applying the new edge length by the same amount $p(t) - p(s)$:

$$\begin{aligned} len_{new}(P) &= \sum_{i=1}^{n} len_{new}(v_i, v_{i+1}) = \sum_{i=1}^{n} len_{old}(v_i, v_{i+1}) - p(v_i) + p(v_{i+1}) \\ &= -p(s) + p(t) + \sum_{i=1}^{n} len_{old}(v_i, v_{i+1}) \\ &= -p(s) + p(t) + len_{old}(P) \end{aligned}$$

Therefore a shortest path in the altered graph is also a shortest path in the original graph.

■

3.2.3 Example Potential Functions

For road maps or other graphs with a geographic origin good lower bounds can often be found by exploiting the real-world coordinates of each vertex. These coordinates determine a layout $L : V \to \mathbb{R}^2$ of the graph. We now assume that the length of an edge (u, v) is the Euclidean distance $\|L(u) - L(v)\|$ of the edge's source and target vertex. Then, the Euclidean distance to the target t, $p(v) = \|L(v) - L(t)\|$ represents a feasible potential.

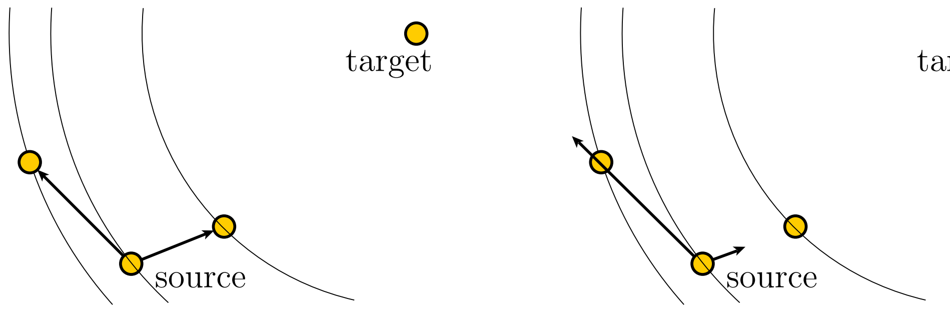


Fig. 3.2: Part of a graph whose edge lengths are induced by the Euclidean distances of the end vertices (left) and the same graph with altered edge lengths (by goal directed search). The circles centered at the target vertex represent the potential of each vertex.

Often however, problems are given where the edge lengths are not exactly proportional to the Euclidean distances but correlate. A common example for that situation is the travel time on a road map. There a corrective factor $v_{max} = \max_{(u,v)\in E} \{len(u, v)/\|L(u) - L(v)\|\}$ has to be multiplied to p: $p_{corr}(v) = v_{max} \cdot \|L(v) - L(t)\|$.

Note that this proceeding will work in any normed vector space if appropriate edge lengths are given.

3.2.4 Dynamic Update of the Example Potential Functions

The first example where edge lengths are proportional to the Euclidean distances requires no preprocessing and therefore is fully dynamic.

The only preprocessing used in the second example is figuring out v_{max}. Pure incremental edge updates can recompute v_{max} by calculating the maximum $v_{max/update} = \max_{(u,v)\in U}(len(u,v)/\|L(u) - L(v)\|$ over all edges in the set of updated edges U. The old value v_{max} has to be substituted by the maximum of v_{max} and $v_{max/update}$. This needs linear time in the number of updated edges.

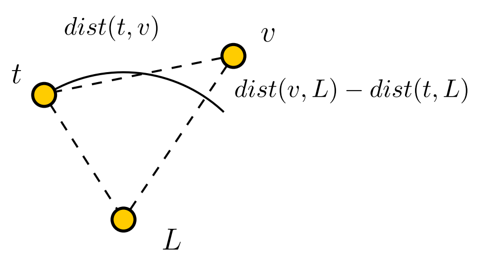
For dealing with the fully dynamic case we propose a slight change in the data structure. The edges shall be sorted by $len(u,v)/\|L(u) - L(v)\|$. This slows the preprocessing from $O(m)$ to $O(m \log m)$ where m is the number of edges in the graph. Updated edges now only have to be re-sorted. This is done in $O(k \log m)$ time, where k denotes the number of updated edges and m the number of the edges in the graph. The new value for v_{max} is the value of the last edge in the list.

3.3 Landmarks

The landmark technique has been introduced in [GH05] as main part of the ALT-algorithms (ALT is an abbreviation for A-star, landmarks, triangular inequality). It is a method to get potential functions for the goal-directed search only using the graph and its length function as input. Therefore it can be applied in case no domain specific information is given. Its idea is to grow full shortest-paths trees on a very small number of vertices (which we call *landmarks* in that context) and exploit lower bounds for the distance to the target out of these trees using the triangular inequality for graphs. We want to refer to [GH05] for an experimental study on this technique and some optimizations including the combination with bidirectional search.

3.3.1 Preprocessing

The preprocessing starts by choosing a small number of vertices of the graph, which we call *landmarks*. Then, for each landmark L we grow a full shortest paths tree rooted at L.



Given a landmark L, the triangle inequality on graphs, $dist(v, L) - dist(t, L) \leq dist(v, t)$ holds. The figure to the left is a schematic example of that inequality. Therefore $p(v) = dist(v, L) - dist(t, L)$ provides a lower bound for the distance $dist(v, t)$ from a vertex v to the target t which we use as a feasible potential function.

As described in the last section, the potential functions $p_i(v)$ derived from different landmarks can be combined to one, better potential function by taking the maximum $p(v) = \max_i\{p_i(v)\}$. For speeding-up the query it may be useful to identify a subset of landmarks which provide strong lower bounds for the distance from source to target of the search. Then the query is run only using these landmarks. Even though this may effect in visiting slightly more nodes, the savings in the calculation of the potential function p often lead to a faster query.

Picking the right landmarks is crucial for that technique. In [GH05] good results are reported for 1 to 16 landmarks at a graph size of 600.000 to 15 Mio edges. Some basic selection strategies are:

By Random. Choosing by random is a simple way of selecting the landmarks. However, the resulting potential function may be far away from being optimal.

Geometric. This approach can be used for graphs with two dimensional layouts like the ones described in the last section. It derives from the observation that having landmarks geometrically lying behind the destination tend to give good potential functions. The algorithm first picks the vertex c that is most close to the center of the graph (here, all geometric statements are meant with respect to the given 2-dimensional layout). Then, the graph is divided into pie-slice sectors centered at c, all of the sectors should contain approximately the same number of vertices. Now, for each sector, the vertex farthest away from c is chosen as landmark.

Farthest Landmark. Starting at an arbitrary vertex as first landmark, this proceeding iteratively adds new landmarks. The following condition has to be satisfied: each new landmark is chosen such that the distance of the new landmark to the nearest of all current landmarks is maximal.

3.3.2 Dynamic Update

The preprocessing of the landmark technique consists of two steps: first choosing the landmarks and then performing Dijkstra's algorithm for each landmark.

If the landmarks stay the same the preprocessing can be efficently updated by the algorithm presented in section 2.4, page 14. Note that for most landmark selection strategies only recomputing the shortest paths trees will *not* give the same result as a full recomputation from scatch would do, because the landmarks stay the same.

However, the landmarks that result from a complete re-choosing are near to the old landmarks as long as the changes in the graph stay 'little enough'. In this case, the recomputation is quality preserving. When updating the preprocessing without changing the landmarks it is important to know wether the selection remains 'good'. An indication for that is the new distance between the landmarks which is explicitely known by the shortest paths trees. Landmarks near to each other are inefficient and should by replaced by new ones.

We do not have to apply that proceeding on the random and the geometric landmark selection strategy: the edges have no influence on these strategies and therefore the landmarks remain the same after an edge update.

To deal with the farthest landmark strategy we recompute the shortest paths trees with the algorithm presented in section 2.4. Then we check on landmarks near to another. If we find a pair of such landmarks, one of both is removed and replaced by a new one selected by the farthest landmark criterion.

3.4 Edge Labels

This technique needs a geometric layout of the underlying graph and tagges to each edge (u, v) some preprocessed geometric information (the edge label) about the area of all vertices that lie on a shortest path that begins with (u, v). When an edge (u, v) is to be relaxed it is checked whether the target vertex t of the search is within the according area of (u, v). The edge (u, v) can be ignored if t is not within that area.

One can distuingish two types of edge labels: bit-vectors and geometric containers. When using bit-vectors the whole graph is separated into several areas. Given an edge (u, v), the bit-vector of (u, v) codes the information which areas contain at least one vertex on a shortest path starting at (u, v). Further reading on bit-vectors can be found in [KMS04, Lau04, MSS+05].

Geometric containers are due to Schulz, Wagner and Weihe [SWW99] and have been improved and experimental studied by Wagner, Willhalm and Zaroliagis in [Wil05, WWZ04, WW06]. The geometric container of an edge (u, v) is a geometric object that contains at least all vertices to which a shortest path starts at (u, v).

3.4.1 Basics

If for each edge (u, v) the exact set $H(u, v)$ of all vertices t for which (u, v) is on the shortest path from u to t is known, good pruning can be achieved: Dijkstra's algorithm can leave out the relaxation of each edge for which the attached target set does not contain the target of the search. It is easy to see that this pruning keeps the optimality of the computed path from source to target.

As storing all these sets is prohibitive because of the memory consumption, appropriate supersets have to be found. Such a superset $H(u, v)$ has to satisfy three requirements: first, it must be possible to determine very fast if a vertex is contained in $H(u, v)$. Second, $H(u, v)$ should not contain too much vertices that are on no shortest path starting at the edge (u, v). Finally, $H(u, v)$ has to be storaged with constant or at least very few memory. For real-world data with a given two-dimensional layout $L : V \to \mathbb{R}^2$ like road-networks geometric objects seem to be a good choice.

Note that this proceeding also works without a given real-world justified layout. One can also try to compute layouts that promise to effect in a good speed-up.

3.4.2 Geometric Containers

The geometric container of an edge (u, v) is a 'simple' geometric object that contains at least all vertices of $H(u, v)$. Out of convex objects like angular sectors or circles, rectangles have been reported to produce the best results in the algorithms runtime. The minimal, rectangular shaped, parallel to the axes geometric container of an edge is called its *bounding box*.

A bidirectional variant of pruning using bounding boxes is due to [WWZ04]. It uses two different edge-labels:

The *(consistent) target container* $T(v, w)$ of an edge (v, w) is an rectangle (an area

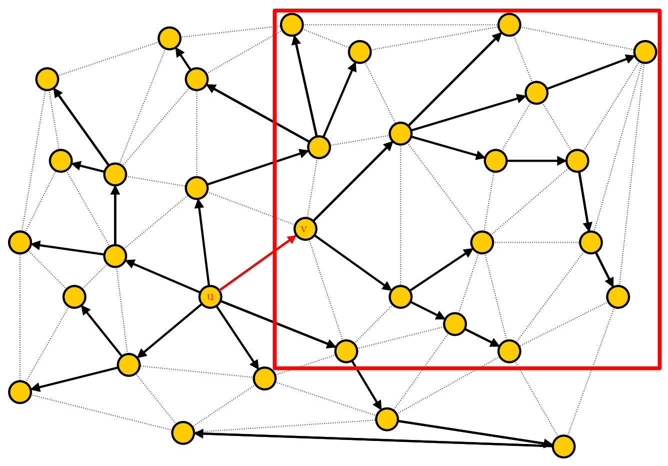


Fig. 3.3: Bounding Box of the edge (u, v). The graph's shortest paths tree rooted at u is drawn with solid lines.

$R \subset \mathbb{R}^2$ of the form $\{(x, y) \in \mathbb{R}^2 \mid x_{min} \leq x \leq x_{max}, y_{min} \leq y \leq y_{max}\})$ that contains at least all vertices t for which there is a shortest path from v to t using the edge (v, w). The *(consistent) source container* $S(v, w)$ of an edge (v, w) is a rectangle that contains at least all vertices s for which there is a shortest path from s to w using the edge (v, w).

We want to point out that such a container does not have to be minimal. It only has to contain the according bounding box.

Preprocessing. To get the target containers we run for each vertex r on the graph a slightly enhanced Dijkstra's algorithm rooted at r. During the algorithm we keep, for each labeled vertex v, the edge (r, u) on the tentative shortest path to v. When v is finished we enhance the bounding box of (r, u) to contain v, if necessary. We use this method on the reverse graph to get the source containers.

Query. To answer a single-source single-target problem with source s and target t a bidirectional Dijkstra's search is used. The forward search is altered such that each edge (u, v) is not relaxed if t is not in $T(u, v)$, the backward search does not relax every edge (u, v) with s not in $S(u, v)$.

3.4.3 Bit-Vectors

This proceeding works as follows: first, we partition the graph into k areas. Then, we assign to each edge a bit-vector with k bits. Each bit represents one of the precomputed areas. We fix an arbitrary area A and an arbitrary edge (u, v). The bit of (u, v) that represents A is set to false if (u, v) lies on no shortest path with at least one vertex in A. Otherwise the bit is set to true.

According to [Wil05], useful partitions can be found through kd-trees when dealing with road maps and through the method described in [HK00] in the general case.

Preprocessing. The preprocessing directly transfers from the preprocessing of geometric containers. [WW06] mentions a great speed-up for the preprocessing of bit-vectors: every shortest path incident to at least two different areas has to enter an area at one vertex. Therefore it is sufficent to consider only vertices on the border of an area instead of solving the complete all-pairs shortest-paths problem. We do that by solving, for each vertex that is on the border of at least one area, the single-source all-targets problem of the reverse graph.

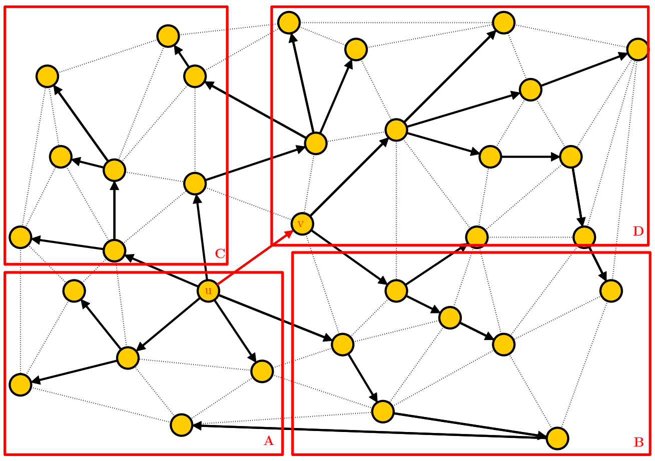


Fig. 3.4: Sample partition in 4 areas. The graph's shortest paths tree rooted at u is drawn with solid lines. The bitvector of the edge (u, v) is $(A = 0, B = 1, C = 0, D = 1)$.

3.4.4 Dynamic Update of Geometric Containers

A routine to update the preprocessing of the bidirectional variant of geometric containers has been published in [WWZ04]. It handles one edge update per time and is split into a decremental and an incremental proceeding. Both proceedings do not consider shortest paths that have been destroyed by the update but only recompute all shortest paths that have been created due to the edge update. Therefore existing edge containers cannot shrink and the update routine is not an exact recomputation. The update method has been reported to be four times faster than a recomputation from scratch would be.

Incremental Update. Given an edge (u, v) with increased length we want to limit the area of all shortest paths that have been created by the update: if an s-t path has been created by the increment of the length of the edge (u, v) then must (u, v) be on the shortest s-t-path on the old graph. It follows that (u, v) is the last edge of a shortest s-v-path and the first edge of a shortest u-t-path in the original graph. Therefore is s contained in $S_{old}(u, v)$ and t in $T_{old}(u, v)$.

To update the geometric containers we grow a full shortest-paths tree on each vertex in $S_{old}(u, v)$ when computing the target containers and on each vertex in $T_{old}(u, v)$ when computing the source containers. The existing geometric containers of vertices outgoing from these vertices are augmented like in the static preprocessing routine.

A second improvement is as follows. It can be shown that for each node x on a shortest s-t-path created by the update of the edge (u, v):

$$dist_{new}(s, x) < dist_{new}(s, u) + len_{new}(u, v) + dist_{new}(v, x)$$

Note that the inequality only holds in case shortest paths are unique. In the general case the $<$ has to be replaced by a $\leq$. To exploit the inequality we first run a Dijkstra's algorithm on the backward graph rooted u and a Dijkstra's algorithm rooted at v. Then $dist_{new}(s, u)$ and $dist_{new}(v, x)$ are known for every s and x. When we perform the update algorithm this inequality can be checked everytime an edge is relaxed by Dijkstra's algorithm and we can omit those edges whose target vertices do not fulfill it.

Decremental Update. Now we deal with the situation that the length of an edge (u, v) has been decreased. Here, the former statement changes to:

If an s-t path has been created by the decrement of the length of the edge (u, v) then must (u, v) be on the shortest s-t-path on the *new* graph. It follows that (u, v) is the last edge of a shortest s-v-path and the first edge of a shortest u-t-path on the *new* graph. Therefore is s contained in $S_{new}(u, v)$ and t in $T_{new}(u, v)$.

Since $S_{new}(u, v)$ and $T_{new}(u, v)$ are unknown at the beginning of the update the first step of the algorithm is to recompute them like in the static case. Then we proceed like in the incremental case only replacing S_{old} and T_{old} by S_{new} and T_{new}.

The improvement changes to

$$dist_{new}(s, x) < dist_{new}(s, u) + len_{\mathbf{old}}(u, v) + dist_{new}(v, x)$$

and can also be applied like in the incremental case.

3.5 Multi-Level Graphs

Multi-level graphs have been intensively experimentally studied since they were introduced by Schulz, Wagner and Weihe in [SWW99, SWW00, SWZ02]. This description is a summary of [HSW06], the most recent paper on the topic. We enhanced it by a sketch of a new update algorithm for the method.

The speed-up of this technique results from a preprocessing step at which the input graph is decomposed into $l+1$ levels. This decomposition is used to limit the search space of Dijkstra's algorithm. Furthermore additional edges are inserted that represent shortest paths connecting important vertices on the graph. These edges can be used as shortcuts for Dijkstra's Algorithm.

3.5.1 Data Structure

Given a graph $G=(V,E)$ with non-negative edge lengths $len : E \to \mathbb{R}^+$ and a subset of the graph's vertices $S \subset V$, we want to construct the *shortest path overlay graph* $G' = (S, E')$ that is defined as follows: for each $(u,v) \in S \times S$ there is an edge (u,v) in E' if and only if for any shortest u-v-path in G no internal vertex belongs to S (internal vertices are all vertices on the path except u and v). The length of each edge in E' is determined by the following condition: for each pair of vertices $(u,v) \in S \times S$ the distance from u to v on G' equals the distance of the two vertices on G.

The construction of that graph can be done by the min-overlay algorithm:

```
Algorithm 4: min-overlay($G$, $len()$, $S$)
forall vertices $u \in S$ do
    • run Dijkstra's algorithm on the graph $G$ with root $u$
    • the edge weights are pairs $(len(e), s_e)$, addition is done pairwise, the
      order is lexicographic, $s_e = \begin{cases} -1 & ; source(e) \in S \setminus \{u\} \\ 0 & ; \text{otherwise} \end{cases}$,
    • break if all vertices in the queue have distance of at most $(\cdot, -1)$
    forall vertices $v \in S \setminus \{u\}$ do
        if $dist(u,v) = (\cdot, 0)$ then
            introduce an edge $(u,v)$ in $E'$ with weight $dist(u,v)$
```

Given a sequence of l subsets of vertices S_i $(1 \leq i \leq l)$ with $V = S_0 \supset S_1 \supset S_2 \supset \ldots \supset S_l$ the *basic multi-level graph* is the result of iteratively applying the min-overlay algorithm: starting with G and S_1, the min-overlay algorithm inserts a set E_1 of edges. Each following step i, min-overlay is applied to (S_i, E_i) and S_{i+1} and inserts the set E_{i+1} of edges. We call the subgraph (S_i, E_i) the *level i*. We say a vertex v is of level i (or a level-i vertex) if i is the maximal level that contains v.

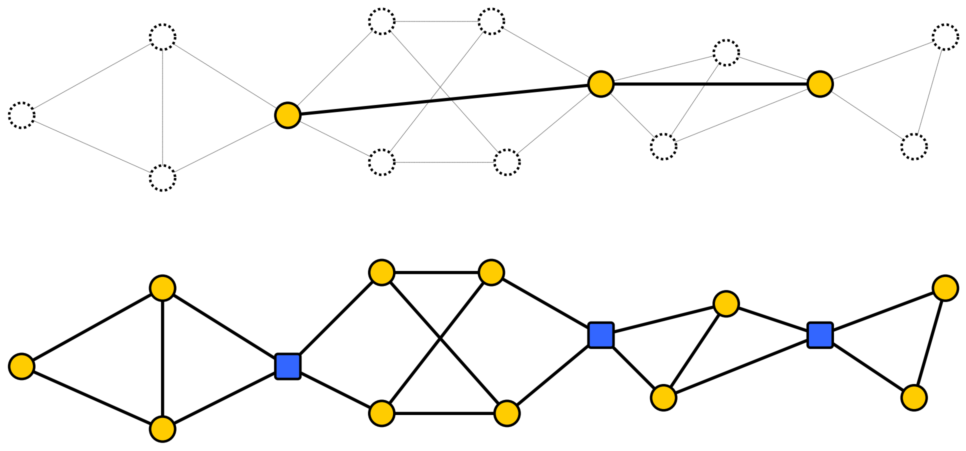


Fig. 3.5: Bidirected 2-level multilevel-graph. The quadratic vertices represent vertices in S. The lower figure shows the original graph, the upper figure shows the edges inserted by the min-overlay algorithm.

If the usage of more preprocessing time and memory consumption is acceptable, the shortest-paths queries can further be sped up by inserting even more edges to the basic multi-level graph. The resulting *extended multi-level graph* contains two new types of edges: *upward edges* from vertices in $S_{i-1} \setminus S_i$ to vertices in S_i and *downward edges* from vertices in S_i to vertices in $S_{i-1} \setminus S_i$. As in the basic multi-level graph, the length of an edge equals the length of a shortest path on the underlying graph. An upward edge (u, v) with $u \in S_{i-1} \setminus S_i$ and $v \in S_i$ (or an downward edge with $v \in S_{i-1} \setminus S_i$ and $u \in S_i$) is only inserted if and only if no other vertex $w \in S_i$ is on a shortest path from u to v.

The min-overlay algorithm can be altered to construct the extended multi-level graph. By changing the last step where the new edges are introduced we can construct downward edges: we consider also vertices $v \in V \setminus S$ and an edge is introduced if $dist(u, v) = (\cdot, 0)$. Upward edges are constructed by running Dijkstra's algorithm for all vertices of the underlying graph instead of running it only for vertices in S_i, and by introducing an edge (u, v) if and only if $dist(u, v) = (\cdot, 0)$.

3.5.2 Query

To answer a given s-t-query, only a subgraph of the basic/extended multi-level graph has to be searched. This subgraph is theoretically determined by a construct called *tree of connected components*, whose description we want to omit because it is not of importance for the update of the multi-level graph. Therefore, we only sketch the search algorithm. The extra information of the tree of connected components that is used for the query is a partition of each level $i - 1$ that is induced by the vertices in S_i. We use the following notation: given an integer $1 \leq i \leq l$ and a vertex v on the subgraph of the multi-level graph that is induced by the vertices $S_{i-1} \setminus S_i$, then C_i^v denotes the maximal connected component on that subgraph containing v. From now on, we assume that the structure of the components C_i^v is known.

Assume that s is a level-k, t is a level-h vertex. Starting a Dijkstra's search at s, the shortest s-t-path query on the basic multi-level graph has to consider only edges contained in E_k while searching in C_k^s. The component C_k^s may only be left to a vertex v of higher level $k+n$, $n \in \mathbb{N}_{>0}$. When relaxing the outgoing edges e of v, only those contained in E_{k+n} are considered. The same holds for each following level: connected components may only be left to a vertex of higher level.

When the search ascends to the highest level or a level i on which s and t are in the same connected component on the subgraph of G induced by the vertices $V \setminus S_{i+1}$, no higher levels have to be considered. All edges of that level i may be used and the search may descend in direction of t. Here, the search space is pruned analogously. When we relax outgoing edges from a vertex of level $k+n$ with $k+n > i$ we consider all edges of level i instead of all edges of level $k+n$.

The query can be further improved by a similar search using the edges of the extended multi-level graph.

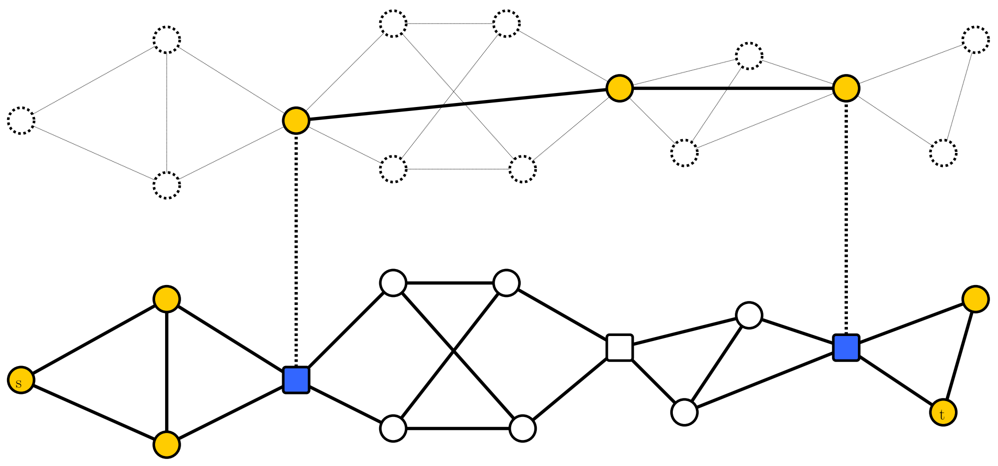


Fig. 3.6: search-space for an s-t-query on the example 2-level multi-level graph

The important tuning parameter of this technique are number of levels and number and selection of the vertices in S_i. We refer to [HSW06] for an experimental study of different criteria for selecting these vertices. An exact description of the query algorithm and proofs of the correctness of the method can be found in [SWZ02].

3.5.3 Dynamic Update

Motivation

We consider a graph $G = (V, E)$ and its min-overlay graph MO induced by a set of vertices S which we call separator vertices here. Intuitively speaking, the vertices in S separate the graph and the subgraph G^- induced by the vertices $V \setminus S$ consists of many little connected components if the vertices in S are 'well' chosen. It is obvious that an edge on the min-overlay graph either represents only one edge with end-vertices in S or connects vertices adjacent to the same component of G^-: a path containing more than one edge between two separator-vertices that are not adjacent to the same component has to traverse at least two components. Therefore it has to pass at least one other separator-vertex and is not represented by an edge in the min-overlay graph.

Let us assume an edge u has been updated. To recompute the min-overlay graph of G, we only have to consider separator vertices adjacent to the connected component containing u. The overlay edges of all other vertices stay the same. We will later treat some special cases where an edge u is not contained in any connected component.

As we allow edge insertions and deletions the structure of the connected components may alter due to an edge udate: components can grow or shrink, be unioned, parted, created or destroyed. In this case we have to identify these structure-altered components and must consider all separator vertices adjacent to either the original or the altered/new components.

We now present an algorithm that recomputes an existing multi-level graph level-by-level. The min-overlay graph of each level is recomputed only considering separator vertices adjacent to components with updated edges. We also give some strategies to further diminish the set of separator vertices considered for recomputation.

Outline

Given a graph $G = (V, E)$ with length function $len_{old} : E \to \mathbb{R}^+$, a sequence of l subsets of vertices S_i $(1 \leq i \leq l)$ with $V = S_0 \supset S_1 \supset S_2 \supset \ldots \supset S_l$ and the multi-level graph ML of G with respect to that sequence. The update is given by a new length function $len_{new} : E \to \mathbb{R}^+$. The set of all edges with altered length is denoted by U. We call G the *original graph* if we apply the edge lengths len_{old} and call it the *altered graph* if we apply the edge lengths len_{new}.

To avoid that edges exist that are contained in no connected component we alter our notion of connected component C_i^v (page 31): we want C_i^v to include also all adjacent separator vertices of that level and the edges between these vertices. If an edge connects two separator vertices not adjacent to the same component we regard those two vertices and the edge connecting them as a separate, degenerated component. We further assume that for each vertex and each level, the vertex is contained in, a label is given that identifies the according connected component(s) C_i^v.

To recompute the multi-level graph of G the graph has to be updated level-by-level. Starting with $i = 0$, we know the set of all updated edges U_i of level i and recompute that level as follows: first, we update the vertices' connected component labels. This is necessary because edge insertions or edge deletions may affect the structure of the connected components.

Then for each separator vertex s contained in at least one component C that either contains at least one element of U_i or that has changed its form, we recompute the min-overlay edges outgoing from s. We remove the min-overlay edges of deleted components. Finally, the changes U_{i+1} between the old and the new overlay graphs have to be identified.

The correctness of this algorithm follows directly from the observation that the edges inserted by the min-overlay algorithm represent only paths within a connected component.

Recomputation of the min-overlay edges outgoing from a given separator vertex

We use the min-overlay algorithm to recompute the min-overlay edges outgoing from a separator vertex s. The only change in the algorithm is to grow a shortest paths tree only from s instead of growing a shortest-paths tree on each separator vertex. Therefore the first line of Algorithm 4, page 30 changes to `for vertex` s `do.`

Full Recomputation of a Connected Component

When dealing with updates that 'seem to have a great impact on the shortest path structure of a connected component C', it is reasonable to recompute the overlay edges for each separator vertex contained in that component. Especially in the case that many edges of a component have changed their lengths, this approach is likely to be runtime-optimal among all possibilities that use no extra information gathered for handling dynamic updates.

Sophisticated Recomputation of a Connected Component

Basics. However, we believe that another strategy does better if the number of updated edges within a connected component is small in relation to the component. For simplicity, we restrict this description to components that have not changed their form. Given a component C on a graph G, the set of separator vertices S contained in C, old and new length functions len_{old}, len_{new} and the set U of edges with updated length. We call G the original graph if we apply len_{old} and call it the altered graph if we apply len_{new}.

Recomputation. We know that the update of an edge (u, v) can only influence an edge between two separator vertices s and t on the min-overlay graph if at least one shortest path between s and t has no other separator vertex on the subpath from s to u and no other separator vertex on the subpath from v to t.

We use a modification of the min-overlay algorithm to identify all vertices $S^- \subseteq S$ that are source vertices of overlay edges that have to be considered for the recompution: to find S^- we run, for each edge (u, v) in U, Dijkstra's algorithm rooted at v on the backward component (the component with the backward edge set) of the original graph. The edge weights and addition are defined analogously to the edge weights and addition in the min-overlay algorithm: the length of an edge (x, s) with $s \in S$ (that are all edges that go out from a separator vertex on the backward edge set) is $(len(x, s), -1)$. The length of each edge (w, v) with $w \neq u$ is $(len(w, v), -1)$. The length of all other edges (x, y) is $(len(x, y), 0)$. We stop the search when all vertices in the queue have distance of at most $(\cdot, -1)$. We repeat the searches on the altered graph.

Let S^- denote the set of all vertices $w \in S$ with distance $(\cdot, 0)$ visited in at least one of the searches. By the construction of S^- we know that S^- contains all separator vertices from which, either on the original or the altered graph, a shortest path starts that contains an updated edge (u, v) and has no other separator vertex on the path from w to v.

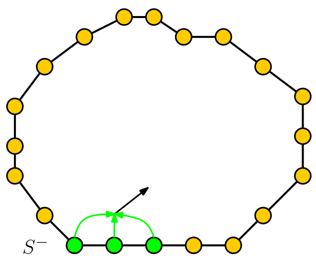


Fig. 3.7: Schematic representation of S^- in a connected component containing an updated edge.

Therefore, the outgoing overlay edges of all vertices in $S \setminus S^-$ remain the same. We only have to recompute the overlay edges outgoing from a vertex in S^-. Figure 3.7 shows a schematic example for S^- within a connected component.

If the extended multi-level graph is to be updated, we define S^- to consist of all vertices $v \in S$ with distance $(\cdot, 0)$.

Bidirectional Variant

Analogously, we can run the searches used to find S^- also on the original edge set (instead of on the reverse edge set) of the original and the altered graph with the source vertex of each updated edge as roots. We denote the resulting set by S^+ and know that S^+ contains all separator vertices *at* which, either on the original or the altered graph, a shortest path *ends* that contains an updated edge and has no other separator vertex on the path from the updated edge to the separator vertex. To recompute the according overlay edges, we can proceed as in the first case but have to run the algorithm on the reverse edge set.

A promising heuristic to reduce the cost of the min-overlay recomputation is to compute both sets, S^+ and S^-. Then, the recomputation should be performed using the set containing fewer vertices. Figure 3.8 shows an schematic example for S^- and S^+ within a connected component.

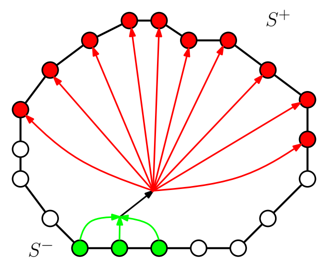


Fig. 3.8: Schematic representation of S^- and S^+ in a connected component containing an updated edge

Improvement for the Bidirectional Variant. Assume that only one edge on the graph has changed its length. Before performing the update of the basic multi-level graph, S^- and S^+ can be further diminished. Once again we want to stress that the update of an edge changes the min-overlay graph only if it lies on a shortest path, either on the original or on the altered component.

After performing the algorithm to compute S^- and S^+, we know for each updated edge (u, v) and each vertex $s^- \in S^-$ the distance from s^- to u on the original graph. For each updated edge e and each vertex $s^+ \in S^+$ we know the distance from the v to s^+.

We observe that the increment of an edge (u, v) can affect the min-overlay graph only if (u, v) is on a shortest path between two separator vertices on the original graph. We can exclude that (u, v) is on a shortest path (represented on the min-overlay graph) between $s^- \in S^-$ and $s^+ \in S^+$ if $dist(s^-, u)_{old} + len(u, v)_{old} + dist(v, s^+)_{old}$ is greater than the length of the min-overlay edge between s^- and s^+. Figure 3.9 visualizes that condition.

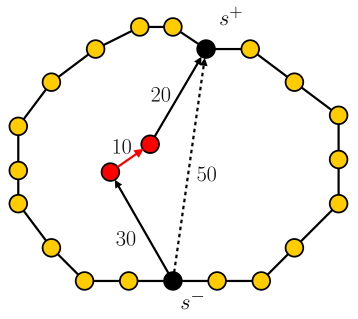


Fig. 3.9: Schematic representation of a connected component. The red edge cannot be on a path responsible for the dashed overlay edge between s^- and s^+.

A similar argument works for decremented edges. The decrement of an edge (u, v) can only affect the min-overlay graph if a path between two separator vertices s^- and s^+ is smaller than the actual shortest path.

Concluding, to recompute the overlay graph, we only have to consider separator vertices

$s^- \in S^-$ for which at least one edge with incremented length (u, v) and one vertex $s^+ \in S^+$ exists such that an overlay edge between s^- and s^+ exists and

$$dist_{old}(s^-, u) + length_{old}(u, v) + dist_{old}(v, s^+) = len_{old}(\text{overlay edge between } s^- \text{ and } s^+)$$

or at least one edge with decremented length (u, v) and one vertex $s^+ \in S^+$ exists such that either no overlay edge between s^- and s^+ exists or

$$dist_{new}(s^-, u) + length_{new}(u, v) + dist_{new}(v, s^+) < \\ len_{old}(\text{overlay edge between } s^- \text{ and } s^+)$$

Note that this strategy can be enhanced to handle updates containing multiple edges. The proceeding for pure incremental or pure decremental updates is obvious. The possibility to enhance it to handle the fully dynamic case results from the observation that a shortest path that changes because of the update of a set of edges U must contain at least one end vertex of an edge in U.

With Use of Additional Data. If we store, for each edge e of the overlay graph, all shortest paths that are responsible for the existence of e, we can further speed-up the update: we only have to consider all vertices $v \in S_i$ from which such a shortest path contains either an incremented edge or an end-vertex of a decremented edge. A problem of this strategy is that it will effect in the consumation of a huge amount of memory.

Comments. Finally, we want to stress that the choice of the sets S_i usually is dependent on the underlying graph. Our update strategy does not update the sets S_i and therefore all these proceedings are no exact recomputations of the preprocessing. However, the proceedings are useful because they are quality preserving as long as the changes between original and altered graph stay 'little enough'.

3.6 Highway Hierarchies

This fairly new technique is due to Sanders and Schultes [SS05] and works on undirected graphs. A paper [SS06] presenting an improved version that also works on directed graphs is to appear. Its main idea is to transform the original graph into a hierarchical graph containing the original graph as first level. Each level i is like the former level $i-i$ but only edges and vertices that are in the middle of shortest paths that contain many vertices on the level $i-1$ are kept on level i. Additionally, the remaining subgraph gets contracted in some way. The query uses a modified version of Dijkstra's algorithm that is run on the preprocessed, hierarchical graph and strongly prunes the search space.

3.6.1 Data Structure

The usage of highway hierarchies requires the notion of canonical shortest paths. Although in [SS05] the canonical shortest path is defined more general than in this work we restrict here to our definition and always think of the version that computes canonical shortest paths when we talk of Dijkstra's algorithm.

To distuingish which edge is far enough to the ends of a shortest path to keep it on the next level, we first have to define the notion of H-neighbourhood: given the case that the priority queue used by Djikstra's Algorithm contains more than one minimal element, we fix an arbitrary but deterministic rule which vertex to take. Then, the Dijkstra rank $r_s(v)$ is the number of vertices already finished by a Dijkstra's algorithm starting at s at the time the vertex v gets marked as finished. For a given vertex s and an integer H, we denote by $d_H(s)$ the distance of the H-closest vertex from s. The H-neighbourhood $N_H(s)$ is defined as $N_H(s) := \{v \in V \mid D(s,v) \leq d_h(S)\}$. From now on we fix an arbitrary H and only write $N(s)$.

Now we are able to describe an iterative proceeding that constructs a sequence of graphs $(G_i)_{i=0\ldots n}$ called *highway hierarchy.* Each graph in that sequence represents one level of our hierarchical graph used for the search. As mentioned, the first graph in the sequence is the original one. Each following graph is computed by building the *highway network* G_{i+1} of a contracted version G'_i of its predecessor. This is done in two steps:

The first step removes all edges (u,v) from G'_i that do not belong to a (canonical) shortest path $P = (s,\ldots,u,v,\ldots,t)$ with $v \notin N(s)$ and $u \notin N(t)$. Furthermore all vertices that became isolated are also removed. The resulting graph is the highway network G_{i+1}.

In the contraction step G'_{i+1} is built: the graph is split into its maximal vertex induced subgraph with minimum degree two (we call that the 2-core of the graph) and all attached trees (that are trees whose roots belong to the 2-core, but all other vertices do not belong to it). Then all attached trees are removed. The remaining graph may contain paths $(u_0,u_1,\ldots,u_k)$ where each inner vertex u_1,u_2,u_{k-1} has degree 2. We call that paths lines and replace every line by a new edge (u_0,u_k). The resulting graph is the contracted highway network G'_{i+1}.

3.6.2 Query

The query used by this technique is a modified bidirectional Dijkstra's algorithm that is run on a graph $\tilde{G} = (\tilde{V}, \tilde{E})$. This graph consists of all graphs $G_0, G_1, \ldots, G_L$. Note that for every vertex v and every level l with $v \in G_l$ the graph $\tilde{G}$ contains a copy v_l of v. For all vertices $v \in G$ and all pairs $v_l, v_{l+1} \in \tilde{G}$ additional edges (v_l, v_{l+1}) with length 0 are inserted. These edges are called *vertical edges* and connect the instances of the same vertex in consecutive levels. We call all other edges (those are all contained in G_i for an $i \leq L$) *horizontal edges.*

The graph is enriched with the following information: for each vertex $v \in G$ and each level $l \leq L$ the distance to the H-closest node in level l, $d^l_H(v)$ is given. By definition we set d^l_H to be infinity if $l = L$ or $v \notin G'_l$. We call the H-neighbourhood of a vertex $v \in G'_l$ $N^l(v) = \{v' \in V'_l \mid d(v, v') \leq d^l_H(v)\}$.

To answer an s-t-query, both directions of the bidirectional Dijkstra's algorithm are expanded by the following rules:

- A vertex v is an *entrance point* if it either has been settled via an vertical edge or if $v \in G'_i$ and has been finished from an horizontal edge starting at a vertex in G_i. The corresponding entrance point of a finished vertex v is the last entrance point on the path to v. In each level l, no horizontal edge is relaxed that would leave the neighbourhood $N^l(v*)$ of the corresponding entrance point $v*$.
- Never visit a vertex $v \in G_i \setminus G'_i$ by a horizontal edge starting at a vertex $v' \in G'_i$.

Furthermore a different abort criterion is used: proceed the searches until both search scopes have met. Proceed further and abort as soon as for each direction starting from $d \in \{s, t\}$ the search from d has no reached but unsettled vertex on levels i where i is lower or equal to the level of an horizonal edges that has been skipped by the opposite search.

3.6.3 Comments

A proof of the correctness, further improvements on that proceeding and a description of the highway network's construction and contraction can be found in the original works. An interesting formulation of the highway-hierarchy technique that shows the connection between this technique and reach-based pruning is stated in [GKW05].

3.7 Reach-Based Pruning

Reach is an edge measure value introduced by Gutman in [Gut04]. The reach of an edge is high, if it lies in the middle of long shortest paths. This can be used for pruning edges when performing Dijkstra's search: if the search is far enough from target and source only edges with high reach have to be considered. This way, the search space is sparsificated using the reach value. The definition of reach, described in the original work is a very general one. The definition, construction- and pruning-strategies we use in this paper hold mainly to a modification of the original reach described in [GKW05]. Section 4.8, page 57 shortly reports the differences between this description and the descriptions in [Gut04] and [GKW05]. Chapter 4, page 44 explains how to construct reach values and upper bounds for reach values while Chapter 5, page 59 describes a dynamic algorithm that efficently updates preprocessed upper bounds for reach values.

3.7.1 Definition

Definition 2 (Reach) Let P be a path from s to t and (v, w) be an edge on P. We denote by $P_{(s,w)}$ the subpath of P from s to w and by $P_{(v,t)}$ the subpath of P from v to t. Then the *reach of* (v, w) *with respect to* P

$$\mathrm{reach}_P(v,w) = \min\{\mathrm{len}(P_{(s,w)}), \mathrm{len}(P_{(v,t)})\}$$

is the minimum of the length of the prefix of P and the length of the suffix of P. The *reach of an edge* (v, w) *(within a graph* G*)* is defined as

$$\mathrm{reach}(v,w) = \max_{\substack{P \texttt{ is canonical path on } G \\ P \texttt{ contains}(v,w)}} \{\mathrm{reach}_P(v,w)\}$$

the maximum over all shortest paths P through (v, w), of the reach of (v, w) with respect to P. We call a path P *responsible for the reach of an edge* (v, w) if the reach of (v, w) with respect to P is the reach of (v, w).

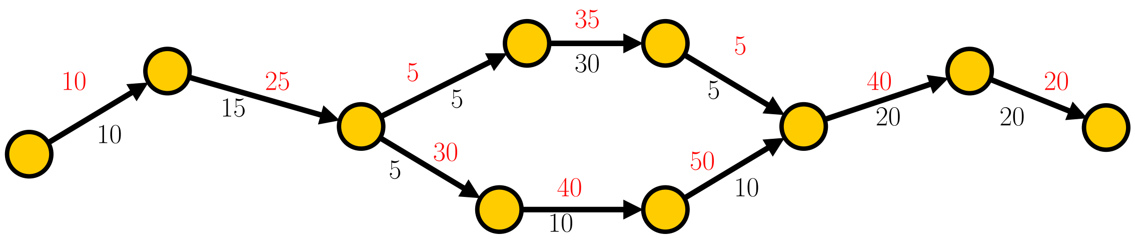


Fig. 3.10: Reach values of a sample graph. Black numbers represent edge lengths, red numbers reach values.

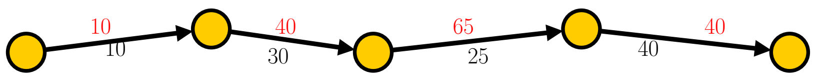


Fig. 3.11: Reach values of a sample path. Black numbers represent edge lengths, red numbers reach values.

An algorithm that computes reach values can be found in the next section. As computing reach values is very time-expensive while upper reach values can be computed in much shorter time, these are used for reach-based pruning. Their construction can also be found in the next section.

3.7.2 Query

We modify a distance balanced bidirectional Dijkstra's search to sparsificate the search space using upper reach-bounds. It is obvious that an edge (u, v) can only be on a shortest path from s to t if $dist(s, u) + len(u, v)$ or $dist(v, t) + len(u, v)$ is lower or equal to the reach of (u, v). If lower bounds $\underline{dist}(s, u)$ for the distances from s to (u, v), $\underline{dist}(v, t)$ for the distances from (u, v) to t and an upper bound $\overline{reach}(u, v)$ for the reach of (u, v) are known, we can exclude all edges (u, v) with

$$\underline{dist}(s, u) + len(u, v) > \overline{reach}(u, v) \text{ and } \underline{dist}(v, t) + len(u, v) > \overline{reach}(u, v)$$

from the search. When the edge (u, v) is relaxed by the direction that starts at s, the exact distance from s to u is known by the distance label $dist(s, u)$ of u. If v has not been finished by the opposite direction the minimal distance of all vertices in the queue of the opposite direction is a lower bound for the distance from v to t. We call a distance balanced bidirectional search using that arguments a *bidirectional bound algorithm*:

Definition 3 (Bidirectional Bound Algorithm) Given a single-source single-target problem with source s and target t on a graph $G = (V, E)$ and upper reach-bounds $\overline{reach}(u, v)$ for each $(u, v) \in E$.

By the bidirectional bound algorithm we denote the distance balanced bidirectional Dijkstra's algorithm whose forward search does not relax every edge (u, v) with

$$dist(s, u) + len(u, v) > \overline{reach}(u, v) \text{ and } \gamma + len(u, v) > \overline{reach}(u, v)$$

and whose backward search does not relax every edge (u, v) with

$$dist(v, t) + len(u, v) > \overline{reach}(u, v) \text{ and } \gamma + len(u, v) > \overline{reach}(u, v)$$

where γ denotes the smallest distance label of all vertices in the priority queue of the opposite search.

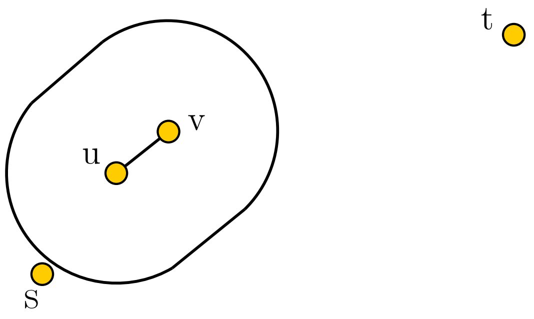


Fig. 3.12: Schematic view of a Dijkstra's search from s to t. The area around the edge (u, v) represents all vertices with distance to (u, v) lower than the reach of the edge. The edge can be pruned by both, the bidirectional bound algorithm and the self-bounding algorithm

A variant of that proceeding where both searches are more independent from each other is the *self-bounding algorithm*. Here, an edge (u, v) is not relaxed if $dist(s, u)+len(u, v) > \overline{reach}(u, v)$. The reverse search starting at t proceeds accordingly. Note, that an edge on a shortest s-t-path that is not relaxed by one search may be relaxed by the opposite. To assure the correctness of the algorithm the stopping criterion must be modified.

Definition 4 (Self-Bounding Algorithm) Given a single-source single-target problem with source s and target t on a graph $G = (V, E)$ and upper reach-bounds $\overline{reach}(e)$ for every $e \in E$.

By the self-bounding algorithm we denote the distance-balanced bidirectional Dijkstra's algorithm whose forward search does not relax every edge (u, v) with

$$dist(s, u) + len(u, v) > \overline{reach}(u, v)$$

and whose backward search does not relax every edge (u, v) with

$$dist(v, t) + len(u, v) > \overline{reach}(u, v)$$

and that uses the following stopping criterion: stop the search in a given direction if there are either no visited vertices or the minimal distance label of all visited vertices is at least half the length of the shortest path seen so far.

The implementation of both algorithms can be improved by sorting the outgoing edges (u, v) of each vertex u descending by the value $\overline{reach}(u, v)-len(u, v)$. For each edge (u, v) which has been pruned from the search, all edges (u, w) with minor value $\overline{reach}(u, w) - len(u, w)$ also have to be pruned. Therefore, the sorting enables the implementation to skip these edges without checking the pruning condition.

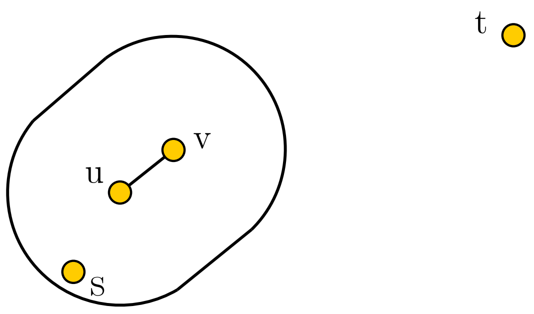


Fig. 3.13: Schematic view of a Dijkstra's search from s to t. The area around the edge (u, v) represents all vertices with distance to (u, v) lower than the reach of the edge. The edge cannot be pruned by the bidirectional bound algorithm but by the search starting at t when performing the self-bounding algorithm

3.7.3 Correctness

The correctness of the algorithms mainly transfers from the correctness of the bidirectional search. To prove the correctness of the self-bounding algorithm, the existence of two cases must be excluded additionally:

- There exists an edge e on a shortest s-t-path that is pruned by both searches.
- There exists an edge e on a shortest s-t-path that is pruned by one search and the stopping criterion of the opposite search holds before e can be relaxed.

The main argument to exclude both cases is that an edge e on a shortest s-t-path can only be pruned by the search starting at the vertex $p \in \{s, t\}$ that is further away from e.

4 Static Reach Preprocessing

We precisely describe a simplified version of the preprocessing used in [GKW05] to get upper reach-bounds for the speed-up technique described in section 3.7.

4.1 Exact Reach

Computing reach values is very time-expensive. Just applying the plain definitions, the shortest paths between any two vertices u, v must be considered. That leads to solve nearly n^2 single-source single-target shortest path problems. The following algorithm merges these problems to n single-source all-target problems and therefore grows a full shortest paths tree T_x on each vertex x. After a shortest paths tree T_x has been grown we compute for each edge e on the tree the reach with respect to the longest path through e that is contained in T_x. After building all shortest paths trees we have considered all shortest paths that are responsible for the reach of at least one edge. Therefore, given an edge e we gain the exact reach value of e by taking the maximum of its formerly computed reach values.

Algorithm 5: Exact Reach

```
1 forall edges $e \in E$ do
2     initialize reach(e)=0
3 forall vertices $x$ in $V$ do
4     grow full shortest path tree $T_x$ with root $x$
5     forall edges $(u, v)$ in $T_x$ do
6         $d =$ farthest descendant of $v$ in $T_x$
7         $height(u, v) = dist(x, v)$
8         $width(u, v) = dist(u, d)$
9         $reach(u, v) = max(reach(u, v), min(height, width))$
```

Unfortunately, this strategy is unsuitable for large graphs. As mentioned, we solve that problem by computing upper reach-bounds instead of the exact ones. From now on, we will refer to reach values as exact reaches and, as no lower bounds are used, to upper reach-bounds as reach-bounds.

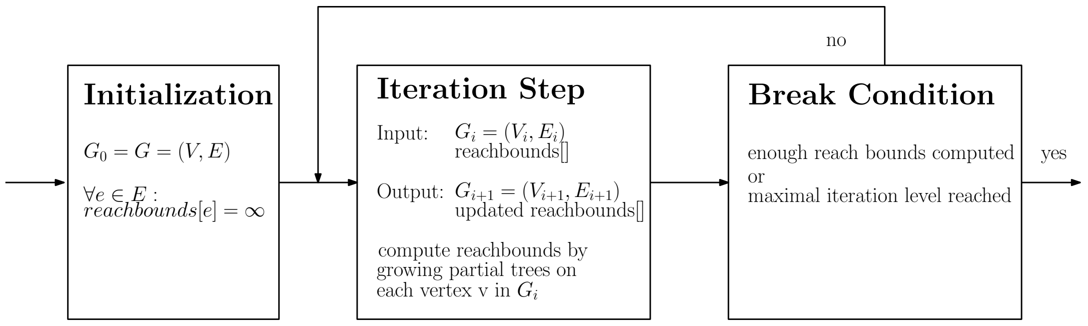
4.2 Motivation

The main idea of the reach-bound computation algorithm is similar to the one for exact reaches. Shortest paths trees are grown from every vertex v. But other than in the exact case the trees are not grown over the whole graph but will be 'cut' at a certain length. We call the resulting tree a *partial tree*. But we are only able to compute reach-bounds for every edge with exact reach lower than a certain treshold ϵ when we use partial trees instead of full shortest paths trees.

These edges will now be removed from the graph and new, bigger partial trees are grown on the resulting, sparsificated graph. Then we will use a *penalty-function* to take the deleted edges into account and are able to compute reach-bounds for edges with reach lower than a new, bigger threshold ϵ_2. This process will be iteratively repeated until enough reach-bounds are calculated.

4.3 Outline

Given a graph $G = (V, E)$ with length function $len : E \to \mathbb{R}^+$, two ascending sequences of numbers ϵ_i and δ_i which are tuning parameters that restrict the size of the partial trees, the static reach-bound computation algorithm works as follows:


no
Initialization
$G_0 = G = (V, E)$
$\forall e \in E$:
$reachbounds[e] = \infty$
Iteration Step
Input: $G_i = (V_i, E_i)$ reachbounds[]
Output: $G_{i+1} = (V_{i+1}, E_{i+1})$ updated reachbounds[]
compute reachbounds by growing partial trees on each vertex v in G_i
Break Condition
enough reach bounds computed or maximal iteration level reached
yes

At initialization, we set $G_0 := (E_0, V_0) := G$ and denote for every edge $e \in E$ with $reachBound_i(e)$ the computed reach-bound of e at iteration step i. $reachBound(e)_0$ is set to infinity for each e in E. Then the algorithm iteratively performs *(reach-bound computation-)steps* until a break condition is fulfilled. We count these steps starting with zero. The break condition needs two more tuning parameteres *maxIt* and *desiredBounds* and splits into two parts: stop the algorithm if either a certain number of iterations *maxIt* is reached or reach-bounds have been found for at least *desiredBounds* edges.

We now describe the proceeding within a single step: the input of the i-th reach-bound computation-step consists of the original graph and the 4-tupel $(G_i, reachBound_i(\cdot), \epsilon_i, \delta_i)$. The output of the i-th reach-bound computation-step is a graph $G_{i+1} \subseteq G_i$ and a valid reach-bound $reachBound_{i+1}(e)$ of each edge $e \in E \setminus E_{i+1}$. Formerly computed reach-bounds lower than infinity stay the same while reach-bounds remain infinity for edges still in the new graph G_{i+1}. Because of that we will often write $reachBound(u, v)$ instead of $reachBound_i(u, v)$.

The i-th step computes the reach-bounds implicitely by computing valid upper bounds for a variant of the original reach on the graph G_i which we call *penalty reach*. In this variant penalty functions called *in-penalty* and *out-penalty* are added to the original reach on G_i to compensate the removed edges.

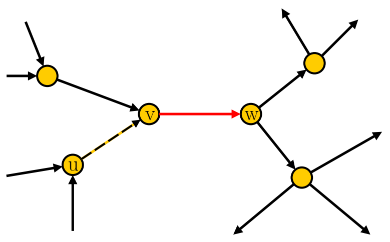
The step does so by growing shortest paths trees whose size ('size' in the sense of length of the contained shortest paths) is controled by the tuning parameter ϵ_i. To prevent these *partial trees* from growing too big ('big' in the sense of contained vertices) the threshold δ_i blocks all edges that have lengths greater than δ_i from being processed.

After all partial trees are grown we identify all edges e for which valid penalty-reaches have been determined. We remove these edges from the graph to get the input graph G_{i+1} of the next computation-step and save their values as according reach-bounds $reachBound_i(e)$. We furthermore remove all isolated nodes from G_{i+1}.

Canonical Shortest Paths. We want to stress that we restrict ourselves to compute canonical shortest paths (description on page 10). In this and the following chapter we are thinking of a canonical shortest path when we speak of a shortest path. Consequently, when we speak of Dijkstra's algorithm we are thinking of the variant that computes canonical shortest paths.

4.4 Penalty Reach

As mentioned before, we have to transform the deleted edges into some form of penalty function.



Consider the situation in the left figure. We are at the beginning of an arbitrary iteration step i, the dotted edge (u, v) has been removed from the graph G because a valid reach-bound $reach(u, v)$ has been found in a former step. When we try to determine the reach of the edge (v, w) in the original graph the problem occurs that the path P responsible for the original reach of (v, w) may contain the deleted edge (u, v). We can compute an upper reach-bound for (v, w) the following way: either P lies fully on G_i and can be computed only considering edges on G_i or P contains the edge (u, v). Then we can estimate (what we justify in section 4.7, page 54) the length of the prefix of P (the subpath from the start vertex to w) by $reachBound(u, v) + len(v, w)$ and compute the length of the suffix of P (the subpath from v to the endpoint) only considering edges on G_i. The reach-bound is computed by taking the minimum of suffix and prefix of P. Sometimes we have to deal with more than one removed, incoming edge. This is done by taking the greatest reach-bound among all incoming edges. The same proceeding symmetrically works for outgoing edges.

As described in the last section we do not process edges higher than a threshold δ_i. When we reinterpret the edge (u, v) of the last example to be such an edge our proceeding stays nearly the same: instead of estimating the prefix by $reach(u, v)$ we set that bound to infinity.

Now, we summarize these ideas in the following definitions of *in-penalty*, *out-penalty* and *penalty-reach*. The in/out penalties assign to each vertex v on the graph a penalty-value representing the former possible shortest paths that contain the removed or forbidden edges.

Definition 5 (Penalty)

Given valid upper bounds $reachBound(u, v)$ for the reach of each edge $(u, v) \in G \backslash G_i$ we define the *in-penalty* at iteration step i of a vertex $v \in V_i$ as

$$\mathrm{iP}_i(v) = \begin{cases} \infty & , \exists (u,v) \in E_i : \mathrm{len}(u,v) > \delta_i \\ \max_{(u,v) \in E \backslash E_i} \{\mathrm{reachBound}(u,v)\}. & , \textit{otherwise} \end{cases}$$

Analogously, the *out-penalty* of v is

$$\mathrm{oP}_i(v) = \begin{cases} \infty & , \exists (v,u) \in E_i : \mathrm{len}(u,v) > \delta_i \\ \max_{(v,u) \in E \backslash E_i} \{\mathrm{reachBound}(v,u)\}. & , \textit{otherwise} \end{cases}$$

We define $\max\{\emptyset\}$ to be 0.

Given a shortest path $P = (s, \ldots, v, w, \ldots, t)$, the penalty-reach of (v, w) with respect to P adds the in-penalty of s to the length of the prefix and the out-penalty of t to the length of the suffix of P. The penalty-reach of (v, w) on G_i is the maximum of the penalty-reach of (v, w) with respect to P over all shortest paths P that contain (v, w). It is a valid reach-bound for the exact reach of e. A proof of that fact is given in section 4.7.

Definition 6 (Penalty Reach)

Let P be a shortest path on G_i starting from vertex s and ending in vertex t. Given an edge (u, v) on P we define the *penalty reach* of (u, v) with respect to P as

$$\mathrm{penReach}_{P/G_i}(u,v) = \min\{iP_i(s) + \mathrm{len}(s,v), \mathrm{len}(u,t) + \mathrm{oP}_i(t)\}$$

Similar to the exact case, the *penalty reach* of an edge $(u, v) \in E_i$ is defined as

$$\mathrm{penReach}_{G_i}(u,v) = \max_{\substack{P \text{ is canonical path on } G_i \\ P \text{ contains } (u,v)}} \{\mathrm{penReach}_P / G_i(u,v)\}$$

Note that the penalty-reach of an edge $(u, v) \in E_{i+1}$ can be different from the penalty reach of $(u, v) \in E_i$.

4.5 Partial Trees

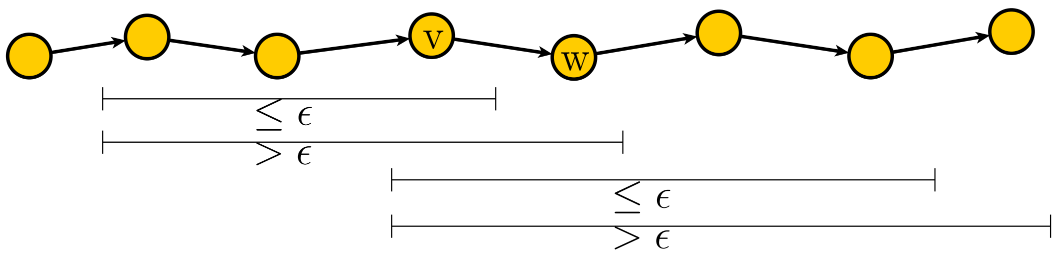
We now describe how we can find penalty reach-bounds of iteration step i for (nearly) all edges e with penalty reach of iteration step i, $penReach_{G_i}(e)$ lower than a threshold ϵ_i. The idea is to grow a shortest paths tree on each vertex $v \in G_i$ with a special break condition.

The break condition of these *partial shortest paths tree* has to ensure the following two claims, all values base on the graph G_i:

- **Claim 1:** For each edge (u, v) with $penReach_{G_i}(u, v) \leq \epsilon_i$ a shortest path P responsible for the penalty reach of (u, v) shall be included in at least one partial tree.

- **Claim 2:** For each edge (u, v) with $penReach_{G_i}(u, v) > \epsilon_i$, at least one shortest path P with $penReach_P(u, v) > \epsilon$ shall be included in at least one partial tree.

A partial tree is built as follows: start Dijkstra's algorithm and keep on processing vertices from the queue until all vertices v that are at most ϵ_i away from the successor of the root on the path to v are finished. We call all vertices v such that the distance from the successor of the root to v is lower or equal to ϵ_i the *inner circle*. Keep on processing vertices from the queue until the successors of all vertices that are at most ϵ_i away from their nearest inner circle predecessor are finished. See the next figure for an example path on such a partial tree.



We want to show the reason why the first edge outgoing from the root of the partial tree may not be counted by the next figure 4.1:

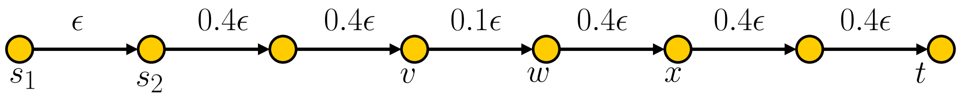


Fig. 4.1: Minimal path responsible for the reach of (v, w). the partial tree rooted at s_1 without counting the first edge contains the full path. The partial tree rooted at s_1 with counting the first edge contains only the subpath from s_1 to x. The partial tree rooted at s_2 contains only the subpath from s_2 to t which is not responsible for the reach of (v, w).

Now we can explain the reason why we do not want very long edges to be processed. Assume that an edge outgoing from the root is a hundred times longer than every other edge. Then the partial tree has to finish many other edges until this edge can be relaxed. It therefore takes a long time to build such a partial tree. Figure 4.2 visualizes that situation by an example. Real-world data justify that proceeding: The distribution of the edge lengths of the graph of the road-map of Germany (where the edge lengths represent the Euclidean distance between two points, shown on Figure 6.1, page 83) approximately follows a function of the form $a \cdot e^{-(length+t)}$ where a and t are real numbers.

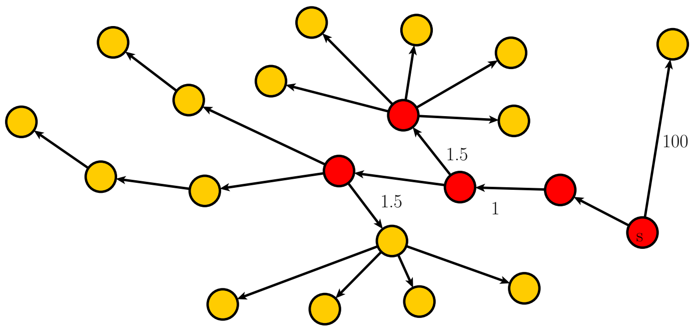


Fig. 4.2: Partial tree rooted at vertex s that was built without using the delta rule. The red vertices represent the partial tree that had been built with use of the delta rule. The according value of epsilon is 1. All edge lengths not on the figure are 30 at most.

We formalize the whole proceeding of building partial trees by the following definitions of *inner circle* and *partial tree*:

Definition 7 (Inner Circle)

Given a path P with source x. Let v be a vertex on P and x' the successor of x on the path to v if exists. v is an element of the inner circle of P with respect to ϵ if it is either the source x or $\text{dist}(x', v) \leq \epsilon$.

Given a tree T_x rooted at x. Let v be a vertex on T_x. v is an element of the inner circle of T_x with respect to ϵ if it is in the inner circle of the path from x to v.

Definition 8 (Partial Tree)

Given two numbers ϵ and δ. Let T_x be the shortest paths tree generated by Dijkstra's algorithm rooted at x for which the following two extra rules are applied:

Stopping rule Stop growing the tree when the inner circle is finished and for every vertex v which is less or exact ϵ away from the nearest inner circle predecessor one of the following condition holds: either it is a leaf and finished or all direct successors of it are finished.

Delta rule Do not relax edges e with $\text{len}(e) > \delta$.

The partial tree rooted at x with size ϵ and delta δ is the subgraph of T_x induced by the finished vertices.

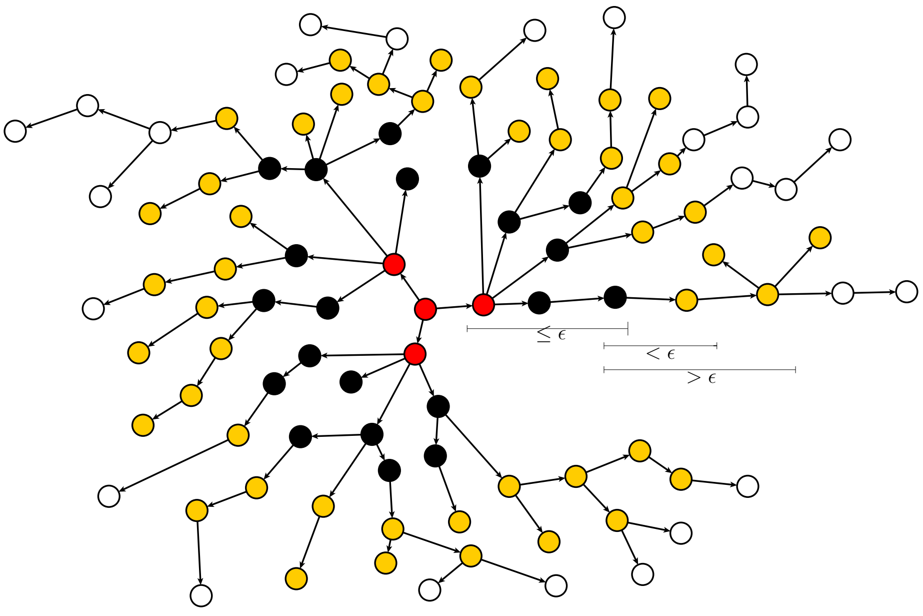


Fig. 4.3: Example of a partial tree. Red and black vertices represent the inner circle, white vertices represent 'useless' vertices that are in the partial tree because other longer paths had to be finished

After a partial tree T_x is built we can compute $reach_{T_x}(u,v)$ of each edge (u,v) on the tree. This is done by taking the minimum of the *exact depth* of (u,v) in T_x (the length of the path from the root to v) and the *exact height* of (u,v) in T_x (the length of the longest path starting at u).

To compute the according penalty reach of edges contained in T_x we have to consider each path $P = (s,\dots,v,w,\dots,t)$ on T_x and must add the in-penalties and out-penalties when computing the reach of (v,w) with respect to P. As we are only interested in the maximal penalty reach over all partial trees, we can find a slightly faster way: we consider only paths $(x,\dots,v,w,\dots,t)$ where x is the root of the partial tree. The correctness of this proceeding is easy to see: given a path $P' = (x',\dots,v,w,\dots,t)$ on T_x with $x \neq x'$ that has a higher penalty-reach than $P = (x,\dots,v,w,\dots,t)$ then this path (or a subpath of it resulting in the compuation of the same reach bound) is also contained in the partial tree rooted at x'. Therefore we have to add the in-penalty of the root vertex to the depth and the out-penalty of the last vertex of a considered path to the height. We call the new values *depth* and *height* of (v,w).

Once again, we formalize that proceeding

Definition 9 (Depth and Height)

Given a partial tree T_x. The *depth* of an edge (u,v) on T_x is $dist(x,v) + \mathrm{iP}(x)$. To every vertex l in T_x a new vertex, the so called pseudo leaf, is appended. The edge-length to the pseudo-leaf shall be $\mathrm{oP}(l)$. The *height* of an edge (u,v) on T_x is the distance between u and its farthest pseudoleaf.

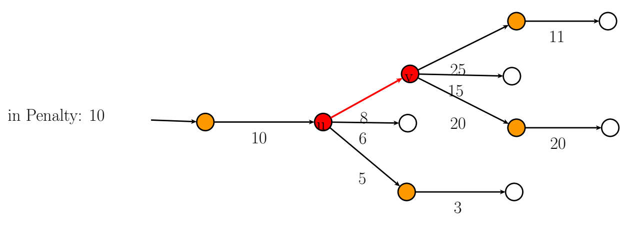


Fig. 4.4: Example for depth and height of an edge (u,v) in a partial tree. The exact depth of (u,v) is 18, the depth 28. The exact height of (u,v) is 33, the height 48.

Summary. The whole computation step i works as follows: grow a partial tree on each vertex of V_i. For each edge (u,v) in E_i save the maximal penalty reach-bound $possibleReachBound(u,v)$ of all penalty reach-bounds computed by the partial trees. Then, for each edge (u,v) with $possibleReachBound(u,v) \leq \epsilon_i$ is $possibleReachBound(u,v)$

a valid upper bound for both, the exact reach of (u, v) and the penalty-reach of the actual iteration step of (u, v).

Theorem 1 (Iteration Step Correctness) Given a graph $G_i = (V_i, E_i) \subseteq G = (V_i, E)$, a length function $len : E \rightarrow \mathbb{R}^+$, valid upper reach-bounds $reachbound(u, v)$ for every edge (u, v) in $G \setminus G_i$ and two positive numbers ϵ_i and δ_i.

Let $possibleReachBound(u, v)$ be the maximum of

$$\min\{depth_{T_x}(u, v), height_{T_x}(u, v)\}$$

over all partial trees T_x rooted at x with size ϵ_i and delta δ_i.

If $possibleReachBound(u, v) \leq \epsilon$ then $possibleReachBound(u, v)$ is a valid upper bound for the reach of (u, v) in G.

4.6 Pseudocode of the Static Algorithm

Algorithm 6: StaticReachBoundComputation(G,epsilon[],delta[])

```
input : Graph $G = (V, E)$, $len : E \rightarrow \mathbb{R}_{>0}$
        Array epsilon[], Array delta[] both of same dimension
ouput: Reach[]
        // stores the ReachBounds
        ReachIterationStep[]
        // stores the iteration step in which the reach-bound was computed
        PartialTreeRoot[]
        // stores the root of the partial tree responsible for the reach-bound

1  $G' := G$
2  foreach edge $e \in E$ do
3      $Reach[e] := 0$
4      $ReachIterationStep[e] := NULL$
5      $PartialTreeRoot[e] := NULL$
6  foreach index i, of epsilon, in ascending order do
7      foreach vertex $x$ in $V'$ do
8          $T_x$:=PartialTree($G'$,$x$,$epsilon[i]$,$delta[i]$)
9          foreach edge $e \in T_x$ do
10             if min(depth(e),height(e))>Reach[e] then
11                 Reach[e]=min(depth(e),height(e))
12                 PartialTreeRoot[e]=x
13     foreach edge $e \in E'$ do
14         if $Reach[e] > epsilon[i]$ then
15             $Reach[e] := 0$
16         else
17             ReachIterationStep[e]:=i
18     $E' := \{e \mid e \in E, bounds[e] = 0\}$
19     $V' := \{v \in V \mid \exists(u, v) \in E' \text{ or } \exists(v, u) \in E'\}$
20     $G' := (V', E')$
```

4.7 Proof of Correctness

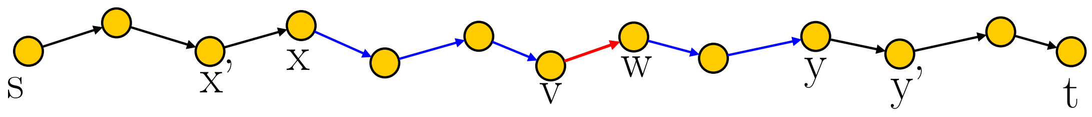
To guarantee that valid upper reach-bounds are computed, we have to proof the following two claims:

1. Penalty-Reach is greater than reach: For each G_t and each $e \in E_t$: $\text{penReach}_{G_t}(e) \geq \text{reach}_G(e)$
2. The algorithm computes upper bounds for penalty reaches correctly

4.7.1 Penalty-Reach is greater or equal to Reach

We have to show that for each G_t and each $e \in E_t$: $\text{penReach}_{G_t}(e) \geq \text{reach}_G(e)$. We do that by induction over the iteration step i. In the following we assume that partial trees are grown without the delta rule, the in- and out-penalties are never set to infinity because of an edge with length greater than δ. Because of $G = G_0$ $\text{penReach}_{G_0}(e)$ equals the reach of e and the initial step is proven.

To prove the induction step we show that $\text{penReach}_{G_{t+1}}(v, w)$ is greater or equal to $\text{penReach}_{G_t}(v, w)$ for each edge (v, w) in G_{t+1}.



We fix an arbitrary (v, w) in G_{t+1} and consider a canonical path P in G_t such that in G_t: $\text{penReach}_P(v, w) = \text{penReach}(v, w)$. Let s be the first, t be the last vertex on P, respectively. Let $(x, \ldots, v, w, \ldots, y)$ be the maximal subpath of P in G_{t+1} that contains (v, w). Let x' be the predecessor of x on P, y' be the successor of x on P.

All following values in this proof belong to step t if no other index is given. Our aim is to show that the penalty reach of (v, w) with respect to the path $(x, x_2, \ldots, x_k, v, w, y_k, \ldots, y_2, y)$ in G_{t+1} is at least as high as $\text{penReach}_P(v, w)$.

Then, because of the later following Lemma 1:

$$\text{penReach}_{G_t}(v, w) \leq \min\{\text{penReach}(x', x) + \text{len}_P(x, w), \text{penReach}(y, y') + \text{len}_P(v, y)\}$$

This transforms to

$$\begin{aligned} \text{penReach}_{G_t}(v, w) &\leq \min\{\text{iP}_{G_{t+1}}(x) + \text{len}_P(x, w), \text{oP}_{G_{t+1}}(y) + \text{len}(v, y)\} \\ &\leq \text{penReach}_{G_{t+1}}(v, w) \end{aligned}$$

The induction step is proven. Note, that the second inequality uses both, the induction hypothesis and the correct computation of upper bounds for penalty reaches in G_t. The claim is also correct when the delta rule is to be applied because at any step the penalty reach-bounds computed using the delta rule are at least as big as the penalty reach-bounds computed without.

■

Lemma 1 With the requisites of this section the follwing two inequalities hold:

$$\begin{aligned} \text{penReach}(x', x) + \text{len}_P(x, w) &\geq \text{penReach}_P(v, w) \\ \text{penReach}(y, y') + \text{len}_P(v, y) &\geq \text{penReach}_P(v, w) \end{aligned}$$

Proof 3

$$\begin{aligned} \text{penReach}(x', x) &\geq \text{penReach}_P(x', x) \\ &= \min\{\text{len}_P(s, x) + \text{iP}(s), \text{len}_P(x', t) + \text{oP}(t)\} \\ \text{penReach}(x', x) + \text{len}_P(x, w) &\geq \min\{\text{len}_P(s, x) + \text{len}_P(x, w) + \text{iP}(s), \text{len}_P(x', t) \\ &\quad -\text{len}_P(x', v) + \text{oP}(t)\} \\ &= \min\{\text{iP}(s) + \text{len}_P(s, w), \text{len}_P(v, t) + \text{oP}(t)\} \\ &= \text{penReach}_P(v, w) \end{aligned}$$

The second inequality is proven analogously.

■

4.7.2 The algorithm computes upper bounds for penalty reaches correctly

At the beginning of an iteration step i we fix an arbitrary edge (v, w). Let *possibleReachBound*(v, w) be the maximum of

$$\min\{depth_{T_x}(v, w), height_{T_x}(v, w)\}$$

over all partial trees T_x rooted at x with size ϵ and delta δ.

We want to show that if *possibleReachBound*(v, w) is lower or equal to epsilon then *possibleReachBound*(v, w) is greater or equal to the penalty reach of (v, w) at iteration step i.

It is straightforward to prove that the reach of (v, w) on G_i is lower than ϵ if *possibleReachBound*$(v, w) \leq \epsilon$. Therefore for all shortest paths $P = (s, \ldots, v, w, \ldots, t)$ is either $dist(s, w)$ or $dist(v, t)$ lower or equal to ϵ.

Let *possibleReachBound*$(v, w) \leq \epsilon$ and $\tilde{P} = (s, \ldots, v, w, \ldots, t)$ be a path responsible for the penalty reach of (v, w). Let $prefix$ denote the subpath from s to w and $suffix$ denote the subpath from v to t.

First case: $len(prefix) < len(suffix)$. Then is $(s, \ldots, v, w)$ fully contained in T_s and $\tilde{P}$ is fully contained in T_s or the height of (v, w) in T_s is greater than ϵ.

Second case: $len(prefix) < len(suffix)$. If $len(prefix) \leq \epsilon$ is $\tilde{P}$ fully contained in P_s.

Let $len(prefix)$ be greater than ϵ. Consider the minimal subpath $minprefix = (\tilde{s}, \ldots, v, w)$ of the $prefix$ such that $len(minprefix) \geq epsilon$. Then is $suffix$ fully contained in $T_{\tilde{s}}$ and the depth of (u, v) in $T_{\tilde{s}}$ is greater or equal to ϵ.

■

4.8 Alternative Reach Pruning Strategies

4.8.1 Gutman's Reach

The original concept of reach introduced in [Gut04] differs from the one we use. Minor important is that Gutman's reach values are assigned to vertices instead of edges and that the query is only unidirectional. The main difference is that Gutman's reach value is induced by a separate, alternative metric while the shortest paths responsible for these reach values remain shortest paths with respect to the original length function:

Given a graph $G = (V, E)$ with length function $len : E \to \mathbb{R}^+$ Gutman assumes that a two-dimensional layout of G and a metric $m : E \to \mathbb{R}^+$ is given such that for each edge (u, v) in E the value $m(u, v)$ is greater or equal to the Euclidean distance of both end-vertices. For road-maps the Euclidean distance of the end-vertices is recommended as such a metric.

The reach of a vertex v with respect to a path $(x_1, x_2, \ldots, x_l = v = y_1, \ldots, y_k)$ is defined as $\min\{\sum_1^{l-1} m(x_i, x_{i+1}), \sum_1^{k-1} m(y_i, y_{i+1})\}$. The reach of a vertex v is defined to be the maximum over all shortest paths (with respect to the *len*-function) P containing v of reaches with respect to P. Gutman also uses upper reach-bounds for the query, the reach-bounds are computed much like in our description. When performing the query a vertex v can be omitted if the reach of v is lower than the Euclidean distance of the source vertex and v and lower than the Euclidean distance of the target vertex and v.

4.8.2 Goldberg's Algorithm

Our version of reach-based pruning is a simplified and slightly altered version of Goldberg's description in [GKW05]. The differences are as follows:

The most significant difference is that in [GKW05] an additional technique is used to sparsificate the graph and to speed-up the preprocessing and the query: between the iteration steps a *shortcut step* is implemented that replaces the in- and outgoing edges of vertices that have only two neighbour-vertices by a shortcut edge. This proceeding does not influence the correctness of computed reach values and Goldberg reports that the preprocessing speeds up by factor 15 when using shortcuts. Further, a speed-up of factor 5 is reported when using the shortcut edges for the query.

Second, an additional version of reach-based pruning is sketched: reach-based pruning can be combined with the ALT-algorithm introduced in [GH05]. The landmarks used in that technique can be exploited as explicit lower bounds for reach-pruning.

Minor important is that in [GKW05] a heuristic earlier stopping condition for partial trees is applied that effects in shorter partial trees but accepts the computation of weaker upper bounds for the reach values.

Further, Goldberg computes reach values for edges but later transforms these edge reach values into vertex reach values. This is done to minimize the memory consumption of the preprocessed data while benefiting from the stronger reach-bounds that result from computing edge reach values. This change also requires a slightly different handling of the query.

To stronger the reach-bounds of edges with high reach a *refinement phase* is appended to the normal preprocessing. This is done by performing an exact penalty-reach computation on a graph induced by vertices with high reach.

Our version of reach uses canonical shortest paths to ensure the uniqueness and inheritence property of shortest paths while in [GKW05] small fractions are added to the edge lengths.

Finally, Goldberg, Kaplan and Warwick do not explicitely specify how to treat very long edges when growing partial trees. We introduced the delta rule and expanded the in- and out-penalties to handle that problem.

5 Dynamic Update of the Reach Preprocessing

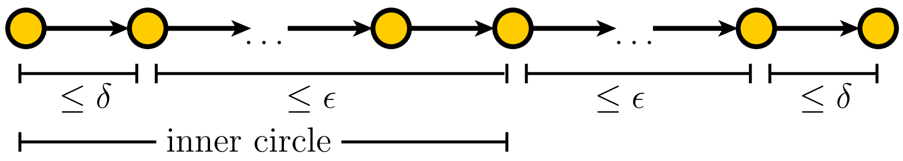
In this chapter we present an algorithm that efficently updates precomputed reach-bounds for an altered graph while guaranteeing to get the same bounds that a full recomputation from scratch by the static algorithm would provide. The algorithm handles the update of multiple edges at the same time and takes advantage of updates 'near to each other'.

5.1 A first Approach

In this section we give a small example to show the main ideas of the update algorithm. To receive a first impression of the 'locality' of the partial tree computation we take a look at the maximal height of a partial tree with size ϵ and delta δ.

Consider a partial tree T and the last vertex v on T that gets finished during the construction of the tree. According to the Definitions 7 and 8, page 50 can the path from the root of the tree to v be separated into three subpaths, listed in ascending distance from the root:

- the edge outgoing from the root
- all edges with inner circle targets without the edge outgoing from the root
- the remaining edges



Because of the delta rule the following boundaries hold: the length of the first subpath is at most δ, the length of the second at most ϵ and the length of the third at most $\epsilon + \delta$. This leads to the following lemma:

Lemma 1 (Partial Tree Bounding Lemma) *Let T be a partial tree of size ϵ and delta δ. Then the length of every path P on T is at most $2\epsilon + 2\delta$.*

Each lemma in this chapter is proven at the end of the chapter in Section 5.7, page 79.

Example. Now, we consider the following situation: reach-bounds have been computed for a graph $G_{old} = (V, E)$ which afterwards has been changed to a graph G_{new} by altering the length of one edge (u, v). Our aim is to recompute the reach preprocessing's first iteration step without starting from scratch. We can do so after identifying two sets:

- A set containing at least all edges whose reach-bounds computed in the first iteration step has changed due to the altered edge length. We call such a set a *reach update area*.
- A set containing at least all vertices on which partial trees have to be built to recompute proper reach-bounds for all edges of an associated reach update area. We call such a set a *reach recomputation area* (of the associated reach update area).

Identification of a Reach Update Area. To identify a reach update area we exploit the former estimation of a partial tree's height. We do so by growing four special shortest-paths trees. Two partial trees are rooted at v and are grown on the reverse edge set (one on G_{new} the other on G_{old}). The other two partial trees are rooted at u and are grown on the normal edge set (one on G_{new}, the other on G_{old}). Each search will be stopped when the queue is empty or every visited vertex is at least $2\epsilon + 2\delta$ away from the root.

Each shortest path P that has the following two properties

- the altered edge (u, v) is contained in P
- P is contained in a partial tree of size ϵ and delta δ on G_{new} or G_{old}

is contained in the finished part of one of the four shortest paths trees. It is easy to see that only edges on such paths can change their reach-bounds when performing the first iteration step of a complete new preprocessing. Therefore the set of all edges that is contained in at least one of the four shortest paths trees is a reach update area.

Identification of a Reach Recomputation Area. To recompute the reach values of each edge (u, v) contained in the reach update area we have to consider each partial tree whose root r has a distance to the edge's target v of at most $2\epsilon + 2\delta$. For each edge (u, v) on the reach update area we grow two shortest paths trees rooted at v: one on the reverse edge set of G_{new} the other one on the reverse edge set of G_{old}. We stop growing these trees when each visited vertex is more than $2\epsilon + 2\delta$ away from v. The set of all finished vertices that are contained in at least one of the grown trees fulfills all requirements to be a reach recomputation area of the former stated reach update area.

Recomputation Process. The recomputation itself is done by building partial trees on each vertex of the reach recomputation area and, for each edge contained in the reach update area, estimating new reach-bounds as done in the static algorithm. Figure 6.8, page 90 shows a schematic representation of the reach update area and the reach recomputation area.

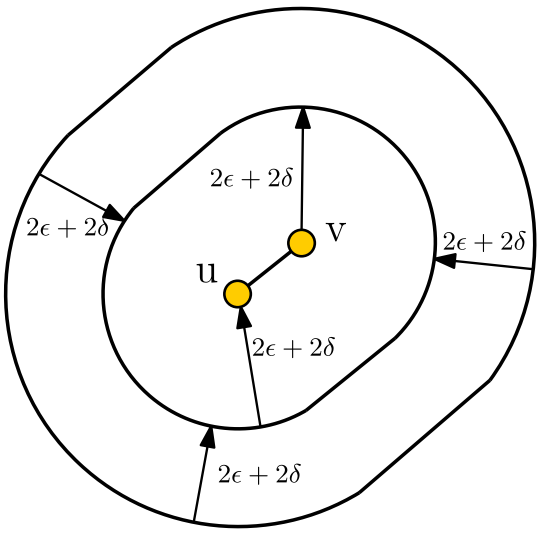


Fig. 5.1: Schematic representation of the reach update area (inner area) and the reach recomputation area (inner and outer area)

Improvement. The reach update area can be imagined as the set of all edges that lie within a ball of radius $2\delta + 2\epsilon$ (here, the distance between two vertices shall be the minimal distance of the vertices on G_{old} and G_{new}) around (u, v). Accordingly, the reach update area consists of all vertices that lie within a ball of radius $4\delta + 4\epsilon$ around (u, v). We observe (with Lemma 1, page 59) that only partial trees whose roots lie within a ball of radius $2\delta + 2\epsilon$ around the updated edge are influenced by the update of (u, v).

We exploit that the following way: we split the reach recomputation area into two areas: the *first reach recomputation area* consists of all vertices w whose distance from w to u is $2\delta + 2\epsilon$ at most (these are the roots of partial trees that may change because of the update of (u, v)). The *second reach recomputation area* consists of some vertices w with distance from w to u between $2\delta+2\epsilon$ and $4\delta+4\epsilon$ (all partial trees rooted at these vertices stay the same as in the original computation).

We now recompute the reach values of all edges in the reach update area by growing partial trees *only* on vertices of the first reach update area. After this computation we know that all edges with tentative computed reach bound greater than ϵ will be dumped after the iteration step. Therefore they do not have to be further considered. Assume that we know for some edges that the reach we have computed until now is already correct. Then we do not have to consider them for a further recomputation.

Let S be the set of all edges with still possibly uncorrect reach bound lower than ϵ. The second reach recomputation area consists of all vertices that are at most $2\epsilon + 2\delta$ away from the target of an edge in S and that are not in the first reach recomputation area. We get correct reach bounds for each edge in S by growing partial trees on each vertex of the second reach recomputation area and correcting the reach bounds of edges in S.

This proceeding is motivated by the fact that the reach recomputation area consists of at least as much vertices as the union set of first and second reach recomputation area. Good speed-up can be achieved if the reach recomputation area contains many vertices that are not in the first or the second reach recomputation area because we avoid building many 'useless' partial trees in this case. Figure 5.2, page 62 visualizes that by a picture.

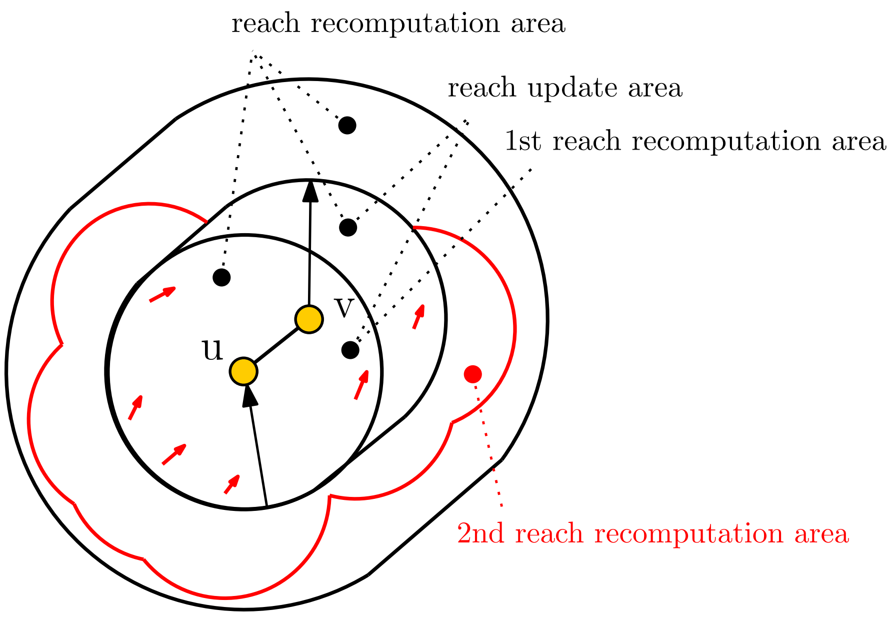


Fig. 5.2: Schematic representation of the reach update area, the first reach recomputation area, the second reach recomputation area and the reach recomputation area. Red arrows represent all edges of the reach update area that have a possible uncorrect reach bounds lower than ϵ after growing partial trees on the first reach recomputation area.

5.2 Outline

Notation. Given a graph $G = (V, E)$ with length function $len_{old} : E \to \mathbb{R}^+$ and two finite sequences of ascending, positive real numbers ϵ_i, δ_i, $1 \leq i \leq k$. We assume that reach-bounds have been computed for edges in E using the static algorithm described in the last chapter to which we refer here as *original computation*. The epsilon and delta values applied to the original recomputation are ϵ_i and δ_i. We denote by $G_{i/old} = (V, E_{i/old})$ the subgraph processed at iteration step i of the original computation. The reach-bound of each edge (u, v) computed by the original computation is denoted by $reachBound_{old}(u, v)$. We set $reachBound_{old}(u, v)$ to infinity for each edge (u, v) without valid original-computation reach-bound.

We further assume that for each edge (u, v) with reach-bound lower than infinity the root $partialTreeRoot_{old}(u, v)$ of the partial tree responsible for the reach-bound of (u, v) computed by the original computation is known. Note that this is additional data not necessary in the static case but can be computed very easily. To do that, we already added the lines 5 and 14 to the pseudo code of Algorithm 6, page 53.

Given an updated length function $len_{new} : E \to \mathbb{R}^+$ our aim is to update the reach-bound preprocessing. We denote by U the set of all edges with altered length. To avoid any ambiguity we refer to the computation from scratch with respect to $length_{new}$, ϵ_i and δ_i using the static algorithm as *full recomputation*.

The variable names when doing the full recomputation remain the same but the subscripts change from *old* to *new*. Note that $E_{0/old} = E_{0/new} = E$.

We will notate a partial tree with root x and grown on the graph $G_{i/new}$ as $T_{x/new}$, grown on the graph $G_{i/old}$ as $T_{x/old}$. We denote by $penReach_{T_{x/new}}(u, v)$ the maximum of penalty depth and penalty height of an edge (u, v) on $T_{x/new}$ with respect to the graph $G_{i/new}$. By convention, $penReach_{T_{x/new}}(u, v)$ is set to infinity if (u, v) is not included in $T_{x/new}$.

Edge deletions and insertions. As decribed in the presentation of the problem, page 9, we regard edge deletions and edge insertions as special case of updated edges by setting the according lengths to infinity. We do not consider such edges when we compute penalties. This proceeding is justified by the observation that both, exact reach values and the computed reach-bounds do not change because of edges with length infinity.

Algorithm. Our algorithm updates the static computation step-by-step using the same ϵ and δ values as used in the original preprocessing. The algorithm works as follows:

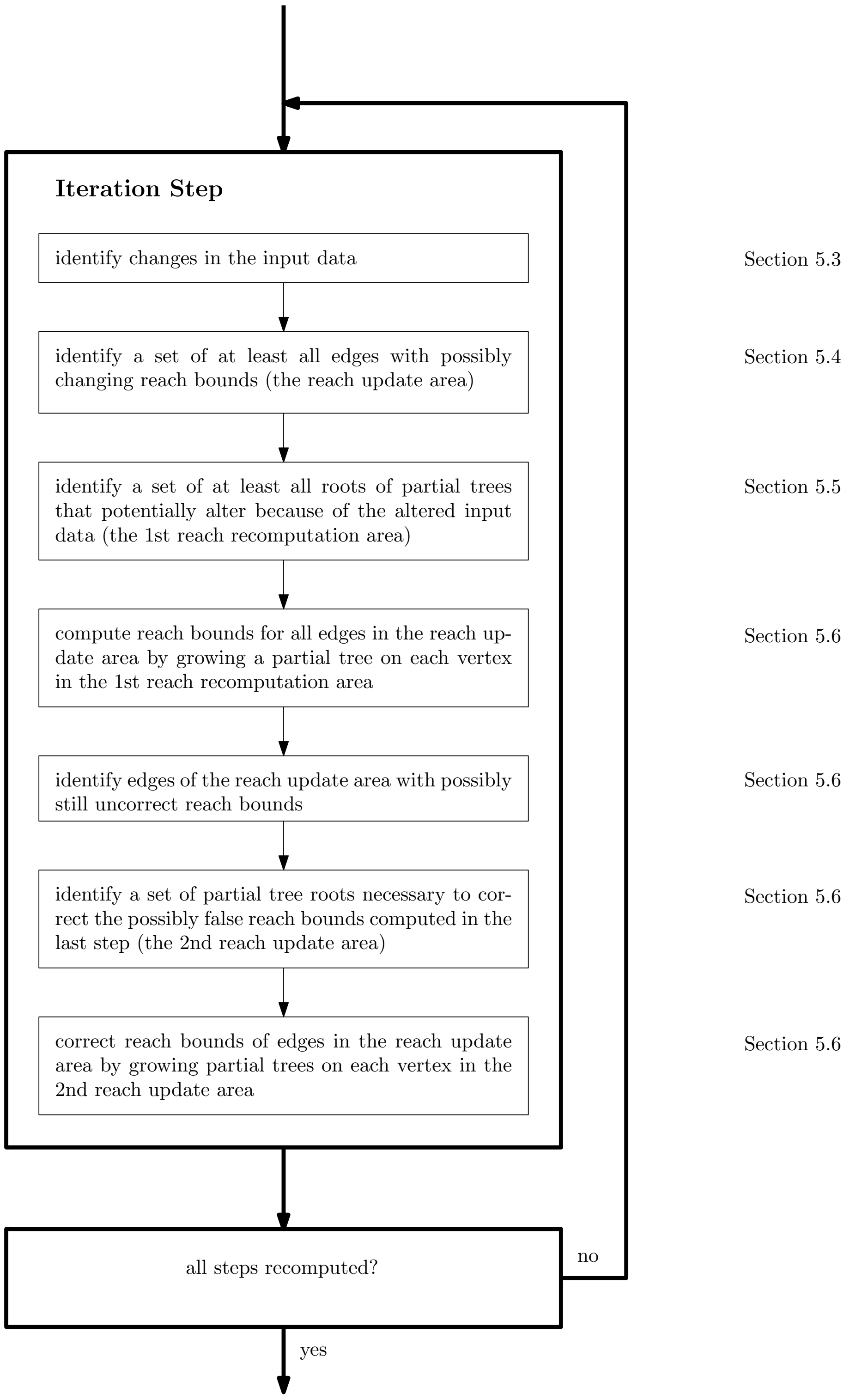

Iteration Step
identify changes in the input data
Section 5.3
identify a set of at least all edges with possibly changing reach bounds (the reach update area)
Section 5.4
identify a set of at least all roots of partial trees that potentially alter because of the altered input data (the 1st reach recomputation area)
Section 5.5
compute reach bounds for all edges in the reach update area by growing a partial tree on each vertex in the 1st reach recomputation area
Section 5.6
identify edges of the reach update area with possibly still uncorrect reach bounds
Section 5.6
identify a set of partial tree roots necessary to correct the possibly false reach bounds computed in the last step (the 2nd reach update area)
Section 5.6
correct reach bounds of edges in the reach update area by growing partial trees on each vertex in the 2nd reach update area
Section 5.6
all steps recomputed?
no
yes

Fig. 5.3: Workflow of the dynamic recomputation algorithm

5.3 Update Type

At the beginning of the iteration step the changes between the original and the new input of the iteration step have to be found. We assign to each edge e on the graph one of the following six update types:

Definition 10 (Update Type$_i$) An edge e is said to be of

UpdateType$_i$nc (no change) if the attached edge data has not changed at all. Formal definition: ($e \in E_{i/new}$, $e \in E_{i/old}$ and $len_{new}(e) = len_{old}(e)$) or ($e \notin E_{i/new}$, $e \notin E_{i/old}$ and $reachBound_{old}(e) = reachBound_{new}(e)$).

UpdateType$_i$ld$\leq \delta$ (lengths differ) if the edge is contained in the new and the old input graph of the iteration step but the edge length has changed and both edge lengths are lower or equal to δ_i. Formal definition: $e \in E_{i/new}$, $e \in E_{i/old}$, $len_{new}(e) <> len_{old}(e)$, $len_{new}(e) \leq \delta_i$ and $len_{old}(e) \leq \delta_i$.

UpdateType$_i$ld$> \delta$ (lengths differ) if the edge is contained in the new and the old input graph of the iteration step but the edge length has changed and both edge lengths are greater than δ_i. Formal definition: $e \in E_{i/new}$, $e \in E_{i/old}$, $len_{new}(e) <> len_{old}(e)$, $len_{new}(e) > \delta_i$ and $len_{old}(e) > \delta_i$.

UpdateType$_i$ld$>< \delta$ (lengths differ) if the edge is contained in the new and the old input graph of the iteration step but the edge length has changed, one edge length is greater than δ_i and one lower or equal to δ_i. Formal definition: $e \in E_{i/new}$, $e \in E_{i/old}$, $len_{new}(e) <> len_{old}(e)$ and (($len_{new}(e) > \delta_i$ and $len_{old}(e) \leq \delta_i$) or ($len_{new}(e) \leq \delta_i$ and $len_{old}(e) > \delta_i$)).

UpdateType$_i$bd (reach-bounds differ) if reach-bounds have already been computed in the old and the new input data but differ. Formal definition: $e \notin E_{i/new}$, $e \notin E_{i/old}$ and $reachBound_{old}(e) <> reachBound_{i/new}(e)$.

UpdateType$_i$cs (computation status differs) if a reach-bound has already been computed in the old input but not in the new one or the other way around. Formal definition: $e \notin E_{i/new}$, $e \in E_{i/old}$ or $e \in E_{i/new}$, $e \notin E_{i/old}$

We denote by $\mathbf{U_i}$ the subset of E that contains all edges that do not have UpdateType$_i$ nc.

We say an edge is of **UpdateType$_i$ld** if it is of UpdateType$_i$ld$> \delta$, UpdateType$_i$ld$\leq \delta$ or UpdateType$_i$ld$>< \delta$.

Obviously every edge in $E_{i/old}$ or $E_{i/new}$ is of exact one UpdateType$_i$. We also observe that no reach-bound has been computed for an edge of UpdateType$_i$ld until step i.

5.4 Reach Update Area

Now we are at the beginning of iteration step i. The input of the original computation is the graph $G_{i/old}$ and its output is $G_{i+1/old}$. The input of the full recomputation is the graph $G_{i/new}$ and its output is $G_{i+1/new}$. We know the differences U_i between $G_{i/old}$ and $G_{i/new}$. If we know a set containing all edges that differ in $G_{i+1/old}$ and $G_{i+1/new}$ (that are all edges in U_{i+1}) we only have to recompute the reach bounds of these edges and can copy the reach bounds computed at iteration step i of all other edges from $G_{i+1/old}$ to $G_{i+1/new}$.

Therefore the first task is to find a set containing at least all edges for which the input data of iteration step $i+1$ in the original computation differs from the input data of the full recomputation. We call such an area a *reach update area*.

Definition 11 (Reach Update Area)

A set containing at least all edges of U_{i+1} (all edges which are not of $\text{UpdateType}_{i+1}\text{nc}$) is called a *reach update area* (of iteration step i).

5.4.1 Reverse Partial Trees and Max Partial Trees

Reverse Partial Trees. To find a reach update area we will often have to find paths that end at a given vertex u and are on partial trees performed in the original computation or the full recomputation. To do that we grow shortest-paths trees rooted at u on the reverse edge set. As described in the first section of this chapter the length of a path on a partial tree with size ϵ and delta δ is at most $2\delta + 2\epsilon$. Therefore we can stop growing the shortest paths tree when the shortest paths to all vertices with distance of at most $2\delta + 2\epsilon$ are known.

We call such a shortest-paths tree a *reverse partial tree*. The next definition uses sets of vertices as roots. For the time being we will only use one vertex as root when growing reverse partial trees. The general definition will be helpful later in this section.

Definition 12 (Reverse Partial Tree)

Let N be a set of vertices. The reverse partial tree rooted at N with size ϵ and delta δ (notation: RT_N) is the finished part (the finished part consists of the finished vertices and the shortest paths edges connecting them) of the tentative shortest paths tree generated by Dijkstra's algorithm for which the following extra rules are applied:

Set Initialization. For all vertices v in N: set distance of v to zero and insert v into the priority queue.

Edge Set Rule. Use the reverse edge set.

Stopping Rule. Stop growing the tree when all vertices with distance of at most $2\epsilon + 2\delta$ from the nearest vertex in N are finished.

Delta Rule Do not relax edges with length greater than δ.

Canonical Rule Choose shortest paths that are canonical with respect to the normal edge set.

We will often omit size and delta of a (reverse) partial tree T_x if we grow T_x at iteration step i, the size of T_x is ϵ_i and delta of T_x is δ_i. The most important property of reverse partial trees is stated in the next lemma.

Lemma 2 Let P be a path on a partial tree rooted at a vertex x with size ϵ and delta δ. Let u be a vertex on P. Then the subpath from x to u on P is contained in the reverse partial tree rooted at $\{u\}$ with same size and delta.

Max Partial Trees. The pendant of reverse partial trees on the normal edge set are *max partial trees.* We will grow them when we want to find paths that start at a given vertex v and are contained in a partial tree.

Definition 13 (Max Partial Tree)

Let N be a set of vertices. The *max partial tree* rooted at N with size ϵ and delta δ (notation: MT_N) is the finished part of the tentative shortest paths tree generated by Dijkstra's algorithm for which the following extra rules are applied:

Set Initialization. For all vertices v in N: set distance of v to zero and insert v into the priority queue.

Stopping Rule. Stop growing the tree when all vertices with distance of at most $2\epsilon + 2\delta$ from the nearest vertex in N are finished.

Delta Rule Do not relax edges with length greater than δ.

Here the lemma changes to:

Lemma 3 Let P be a path on a partial tree rooted at x. Then each subpath $(u_1, u_2, \dots, u_n)$ of P is included in the max partial tree rooted at $\{u_1\}$ with same size and delta.

Comments. Note that, in the graph theoretical sense, max partial trees and reverse partial trees are not trees but forests. We want to stress that Lemma 2 and 3 only hold because we compute canonical shortest paths. When computing arbitrary shortest paths Lemma 2 and 3 only hold on graphs where all shortest paths are unique.

When growing partial trees without the delta-rule or if δ_i is greater than the length len_{max} of the longest edge on the graph then the length of every shortest path on a partial tree with size ϵ_i is $2\epsilon_i + 2len_{max}$ at most. In this case, the stopping rule of reverse/max partial trees can be altered to: stop growing the tree when all vertices with distance smaller than $2\epsilon_i + 2len_{max}$ are finished. This change does not affect the correctness of Lemma 2 and Lemma 3.

5.4.2 Eager Construction of a Reach Update Area

Unless stated otherwisely all claims in this subsection that concern (penalty) reach bounds, partial trees or reverse partial trees refer to reach bounds, partial trees or reverse partial trees computed at the actual iteration step i.

Aim. Our aim is to find a superset of U_{i+1}. We know that an edge with UpdateType$_{i+1}$ld must also be of UpdateType$_i$ld and it is therefore easy to find all such edges. Hence we concentrate on finding edges of the other two types (that are edges (u, v) whose reach bounds $reachBound_{i+1}(u, v)$ differ in the original computation and the full recomputation).

We observe two reasons why an edge (w, z) may be in U_{i+1}. First, it can already be in U_i. Second, an edge $(u, v) \in U_i$ is adjacent to (or contained in) either a partial tree responsible for the penalty reach of (w, z) on $G_{i/old}$ or to a partial tree responsible for the penalty reach of (w, z) on $G_{i/new}$.

Plan. We consider UpdateTypes$_i$ ld, bd and cs separately and identify for each edge (u, v) in U_i a set of edges $U_{i+1}(u, v)$ whose reach bounds computed until the beginning of step $i + 1$ differ (between the original computation and the full recomputation) because of the change of (u, v):

$(\mathbf{u}, \mathbf{v})$ is of UpdateType$_i$ld$>$ δ. In that case the difference of the length of (u, v) between G_{old} and G_{new} does not influence the reach computation at iteration step i. Therefore $U_{i+1}(u, v)$ is the empty set.

Reason: until the beginning of iteration step $i + 1$, an edge (u, v) of UpdateType$_i$ld$>$ δ is neither processed in the original computation nor in the full recomputation. The in-penalty of u and the out-penalty of v is set to infinity in both computations. Therefore no reach value computed until the beginning of step $i + 1$ changes because of an edge of that update type.

$(\mathbf{u},\mathbf{v})$ is of UpdateType$_i$ld $\leq \delta$. Here, we grow four shortest paths trees: two max partial trees (one on $G_{i/old}$ and one on $G_{i/new}$) rooted at u and two reverse partial trees (one on $G_{i/old}$ and one on $G_{i/new}$) rooted at v. All edges that possibly change their reach bounds computed before iteration step $i+1$ lie on the branch of one of the four shortest paths trees that starts with the edge (u,v).

Reason: an edge (u,v) of UpdateType$_i$ld$\leq \delta$ must be contained in a shortest path P responsible for the penalty reach of an edge (w,z) (either on the old or on the new graph) to influence the penalty reach of (w,z). To find all edges (w,z) possibly affected by (u,v) we have to remember that (u,v) can be in front of or behind (w,z) on P. We do not have to consider shortest paths that are so long that they are not included in a partial tree built on this iteration step.

We get all such edges behind (w,z) by growing two max partial trees (one on $G_{i/old}$ and one on $G_{i/new}$) rooted at u. Then all possibly affected edges are on the branch of the resulting shortest-paths trees that start with the edge (u,v). To get the edges in front of (u,v) we grow two reverse partial trees (one on $G_{i/old}$ and one on $G_{i/new}$) rooted at v. All possibly affected edges (w,z) in front of (u,v) are contained in the branch that begins with the edge (u,v) of one of both reverse partial trees.

$(\mathbf{u},\mathbf{v})$ is of UpdateType$_i$bd. In that case all edges of $U_{i+1}(u,v)$ lie on either a max partial tree rooted at v or a reverse partial tree rooted at u that is grown on either $G_{i/old}$ or on $G_{i/new}$. We do not have to consider the max partial trees if in $G \setminus G_{i/old}$ an edge $(\overline{u},v)$ exists with $\mathit{reachbound}_{old}(\overline{u},v) > \mathit{reachbound}_{old}(u,v)$ and in $G \setminus G_{i/new}$ an edge $(\tilde{u},v)$ exists with $\mathit{reachbound}_{new}(\tilde{u},v) > \mathit{reachbound}_{new}(u,v)$. The same holds symmetrically for the reverse partial trees.

Reason: given an edge (u,v) of UpdateType$_i$bd. Let (u,v) influence the penalty-reach of another edge (w,z) with path P responsible for the penalty-reach of (w,z). Then either v must be in front of w on P or u must be behind z on P. We find such edges (w,z) by growing max partial trees rooted at v and reverse partial trees rooted at u on $G_{i/old}$ and on $G_{i/new}$.

Given an edge (u,v) of UpdateType$_i$bd. This edge has no influence on the in-penalty of v if in $G \setminus G_{i/old}$ an edge $(\overline{u},v)$ exists with $\mathit{reachbound}_{old}(\overline{u},v) > \mathit{reachbound}_{old}(u,v)$ and in $G \setminus G_{i/new}$ an edge $(\tilde{u},v)$ exists with $\mathit{reachbound}_{new}(\tilde{u},v) > \mathit{reachbound}_{new}(u,v)$. The same holds symmetrically for out-penalty of u.

$(\mathbf{u},\mathbf{v})$ is of UpdateType cs or ld$><\delta$. In that case all edges of $U_{i+1}(u,v)$ lie on the union of a max partial tree rooted at v and a reverse partial tree rooted at u that is grown on either $G_{i/old}$ or on $G_{i/new}$.

Reason: given an edge (u,v) of UpdateType$_i$cs or ld$><\ \delta$. Let (u,v) influence the penalty-reach of another edge (w,z) with path P responsible for the penalty-reach of (w,z). Then either v must be on P in front of w or u must be on P behind z. We find such edges (w,z) by growing a max partial tree rooted at v and a reverse partial tree rooted at u on $G_{i/old}$ and on $G_{i/new}$.

Algorithm to compute a reach update area. For each edge (v,w) of U_i we know how to get a set of edges $U_{i+1}(v,w)$ whose reach bounds computed at iteration step i differ between original recomputation and full recomputation because of the change of (v,w). The union set of U_i and all $U_{i+1}(v,w)$ where (v,w) is in U_i is a reach update area. We summarize that in the following theorem:

Theorem 2 (Eager Construction of Reach Update Area)
Let

$$\begin{aligned}
N_{ld} &= \{ \; v \in V \mid \exists(u,v) \text{ of UpdateType}_i\text{ld} \leq \delta\} \\
N_{bd,cs} &= \{ \; u \in V \mid \exists(u,v) \text{ of UpdateType}_i\text{cs or ld} >< \delta\} \cup \\
& \{ \; u \in V \mid \exists(u,v) \text{ of UpdateType}_i bd, \\
&\qquad \nexists(u,\tilde{v}) \in E \setminus E_{i/new} \text{ with } reach_{new}(u,\tilde{v}) \geq reach_{new}(u,v), \\
&\qquad \nexists(u,\overline{v}) \in E \setminus E_{i/old} \text{ with } reach_{old}(u,\overline{v}) \geq reach_{old}(u,v)\} \\
\tilde{N}_{ld} &= \{ \; u \in V \mid \exists(u,v) \text{ is of UpdateType}_i\text{ld} \leq \delta\} \\
\tilde{N}_{bd,cs} &= \{ \; v \in V \mid \exists(u,v) \text{ is of UpdateType}_i\text{cs or ld} >< \delta\} \cup \\
& \{ \; v \in V \mid \exists(u,v) \text{ is of UpdateType}_i\text{bd}, \\
&\qquad \nexists(\tilde{u},v) \in E \setminus E_{i/new} \text{ with } reach_{new}(\tilde{u},v) \geq reach_{new}(u,v), \\
&\qquad \nexists(\overline{u},v) \in E \setminus E_{i/old} \text{ with } reach_{old}(\overline{u},v) \geq reach_{old}(u,v)\}.
\end{aligned}$$

Then the set of all edges contained in

$$\begin{aligned}
\{e \;&|\; e \in U_i\} \cup \\
\{T \;&|\; T \text{ is rev. par. tree grown on } G_{i/old} \text{ or on } G_{i/new}, \text{ rooted at } v \in N_{bd,cs}\} \cup \\
\{B \;&|\; B \text{ is a branch of the rev. par. tree grown on } G_{i/new} \text{ or on } G_{i/old}, \\
&\quad \text{rooted at } v \in N_{ld} \text{ beginning with an edge } (w,v) \text{ of UpdateType}_i \text{ ld} \leq \delta\} \cup \\
\{T \;&|\; T \text{ is max par. tree grown on } G_{i/old} \text{ or on } G_{i/new}, \text{ rooted at } v \in \tilde{N}_{bd,cs}\} \cup \\
\{B \;&|\; B \text{ is a branch of the max par. tree grown on } G_{i/new} \text{ or on } G_{i/old}, \\
&\quad \text{rooted at } v \in \tilde{N}_{ld} \text{ beginning with an edge } (v,w) \text{ of UpdateType}_i \text{ ld} \leq \delta\}
\end{aligned}$$

is a reach update area of iteration step i.

To union an edge (u,v) with a subgraph G we identify (u,v) with the subgraph $(\{u,v\},\{(u,v)\})$.

5.4.3 Lazy Construction of a Reach Update Area

Identifying a reach update area using the eager construction is very time consuming because many max partial trees and reverse partial trees have to be built. If many edges in U_i are 'near to each other' the reverse/max partial trees rooted at the end vertices of these edges often visit almost the same vertices. We want to exploit this observation to speed-up the construction of the reach update area and accept to get a reach update area that may contain more edges than the one computed by the eager construction.

Given a set of vertices N. To find the set of all vertices marked as finished by at least one max/reverse partial tree rooted at an element of N we can grow a max/reverse partial tree rooted at N. The next Lemma 4 gives the main argument for the correctness of that proceeding.

Lemma 4 (Monotony of Max Partial Trees and Reverse Partial Trees)

Let v be a vertex on a max partial tree / reverse partial tree rooted at a set N_1. Then, for every set N_2, v is on the max partial tree / reverse partial tree rooted at $N_1 \cup N_2$ that uses the same ϵ and δ values.

As special case this implies that all vertices on a partial tree with root x are contained in every max partial tree whose root set contains x.

We want to stress that though the union of all vertices contained in at least one max partial tree with root in N is included in the set of vertices contained in a max partial tree with root N, the edges contained in the set partial tree may be different from the set of edges contained in the according partial trees. The same holds for reverse partial trees.

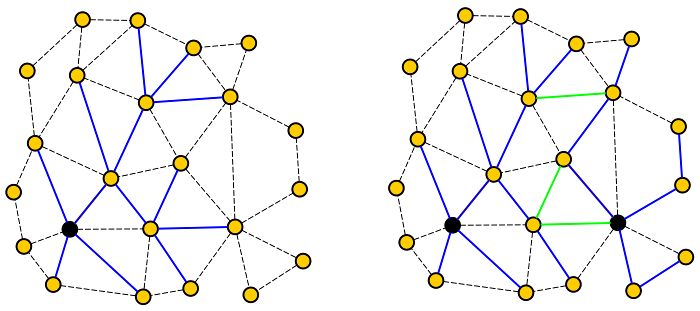


Fig. 5.4: Sample Graph. The left picture shows the max partial tree rooted at the black vertex. The right picture shows a max partial tree on the same graph with an additional root. Edges that are on the left max partial tree but not on the right are drawn green.

Lazy Construction. To get a reach update area we remember the sets occuring in Theorem 2 and grow a max partial tree rooted at $\tilde{N}_{ld} \cup \tilde{N}_{bd,cs}$ and a reverse partial tree rooted at $N_{ld} \cup N_{bd,cs}$ on both, $G_{i/old}$ and $G_{i/new}$. We know that we finish the same vertices as in Theorem 2. Therefore the set of all edges computed by Theorem 2 is included in the set of all edges (u, v) in $G_{i/new}$ where u and v are both contained in the same of one of the four max/reverse partial trees grown. We call the resulting proceeding the *lazy construction of a reach update area.*

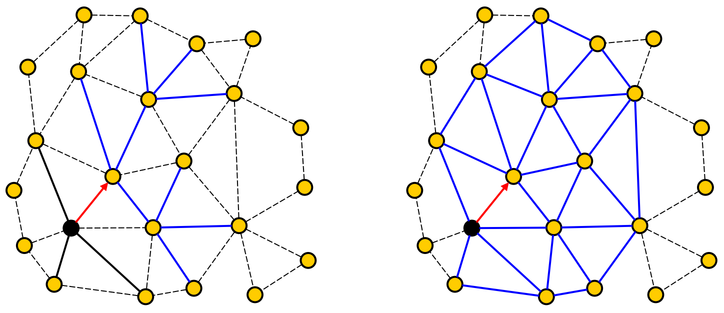


Fig. 5.5: Sample Graph. The red edge is in U_i. The blue edges represent the reach update area edges. The left picture shows the eager construction, the right picture shows the lazy construction.

Theorem 3 (Lazy Construction of Reach Update Area)
Let N_{ld}, $N_{bd,cs}$, $\tilde{N}_{ld}$ and $\tilde{N}_{bc,cs}$ be as in Theorem 2. Further, let

- RT_{old} be the reverse partial tree rooted at $N_{ld} \cup N_{bd,cs}$ built on $G_{i/old}$
- RT_{new} be the reverse partial tree rooted at $N_{ld} \cup N_{bd,cs}$ built on $G_{i/new}$
- T_{old} be the max partial tree rooted at $\tilde{N}_{ld} \cup \tilde{N}_{bd,cs}$ built on $G_{i/old}$
- T_{new} be the max partial tree rooted at $\tilde{N}_{ld} \cup \tilde{N}_{bd,cs}$ built on $G_{i/new}$

Then the union set of U_i and all edges for which at least one of these subgraphs contains the source and the target

$$\begin{array}{rcl} U_i & \cup & \\ \{(u,v) \mid u,v \in RT_{old}, (u,v) \in E_{i/old}\} & \cup & \{(u,v) \mid u,v \in RT_{new}, (u,v) \in E_{i/new}\} \cup \\ \{(u,v) \mid u,v \in T_{old}, (u,v) \in E_{i/old}\} & \cup & \{(u,v) \mid u,v \in T_{new}, (u,v) \in E_{i/new}\} \end{array}$$

is a reach update area (of iteration step i).

Decremental or incremental improvement for the lazy construction. The lazy construction can be sped-up if the update on the original graph (or, more precise, the difference between $G_{i/old}$ and $G_{i/new}$) was pure incrementel or pure decremental.

If for each edge (u, v) on $G_{i/new}$ follows that (u, v) is in $G_{i/old}$ and $len_{new}(u, v) \geq len_{old}(u, v)$ then the vertex set of a reverse/max partial tree grown on the new graph is contained in the vertex set of the according reverse/max partial tree grown on the old graph. Therefore it suffices to build all occuring trees on $G_{i/old}$.

The same argument justifies to build all occuring trees only on $G_{i/new}$ when dealing with pure decremental updates.

Corollary 1 (Decr. and Incr. Lazy Reach Update Area Construction)
With the requisites of Theorem 3 follows: let $E_{i/new} \subseteq E_{i/old}$ and for all edges (u, v) in $G_{i/new}$ be $len_{old}(u, v) \leq len_{new}(u, v)$.
Then is

$$U_i \cup \{(u, v) \mid u, v \in RT_{old}, (u, v) \in E_{i/old}\} \cup \{(u, v) \mid u, v \in T_{old}, (u, v) \in E_{i/old}\}$$

a reach update area. Let $E_{i/new} \supseteq E_{i/old}$ and for all edges (u, v) in $G_{i/old}$ be $len_{old}(u, v) \geq len_{new}(u, v)$. Then is

$$U_i \cup \{(u, v) \mid u, v \in RT_{new}, (u, v) \in E_{i/new}\} \cup \{(u, v) \mid u, v \in T_{new}, (u, v) \in E_{i/new}\}$$

a reach update area (of iteration step i).

5.5 First Reach Recomputation Area

We now search a set that contains at least all vertices from which partial trees can be grown that are 'influenced' by the change of edges in U_i. We call such a set a *first reach recomputation area*. We will later use that knowledge to shrink the area which is necessary to compute the reach bounds of edges in the reach update area.

Definition 14 (First Reach Recomputation Area)
A set that contains at least all vertices x for which

- either the partial tree $T_{x/old}$ rooted at x and grown on $G_{i/old}$ differs from (does not consist of the same edge set as) the partial tree $T_{x/new}$ rooted at x and grown on $G_{i/new}$
- or $T_{x/old}$ and $T_{x/new}$ are equal with respect to the edge set but at least one edge (u, v) exists such that $penReach_{T_{x/new}}(u, v) \neq penReach_{T_{x/old}}(u, v)$

is called a *first reach recomputation area* (of iteration step i).

The methods used to find a reach update area can also be used to find a first reach recomputation area. Given an edge (u, v) in U_i the difference between finding a reach update area and finding a reach recomputation area is that we only have to find paths contained in partial trees that *end with* u or v instead of finding paths on partial trees that *contain* u or v. Therefore we only have to consider the reverse partial trees grown to identify a reach update area. The correctness of that proceeding follows from the same arguments as the eager construction of the reach update area in the last section.

The eager construction of a first reach recomputation area uses the same reverse partial trees as the eager construction of a reach update area but does not consider the max partial trees:

Theorem 4 (Eager Construction of a First Reach Recomputation Area)
Let N_{ld}, $N_{bd,cs}$, $\tilde{N}_{ld}$ and $\tilde{N}_{bd,cs}$ be like in Theorem 2. The set of all vertices contained in

$$\begin{array}{lll}
\{u & \mid & u \in U_i \text{ and } u \in G_{i/new}\} \cup \\
\{T & \mid & T \text{ is rev. par. tree grown on } G_{i/old} \text{ or on } G_{i/new}, \text{ rooted at } v \in N_{bd,cs}\} \cup \\
\{B & \mid & B \text{ is a branch of the rev. par. tree grown on } G_{i/new} \text{ or on } G_{i/old}, \\
 & & \text{rooted at } v \in N_{ld} \text{ beginning with an edge } (w, v) \text{ of UpdateType}_i \text{ ld} \leq \epsilon\}
\end{array}$$

is a first reach recomputation area (of iteration step i).

The same holds for the lazy construction:

Theorem 5 (Lazy Construction of a First Reach Recomputation Area)
Let RT_{old} and RT_{new} be like in Theorem 3.

The set of all vertices contained in RT_{old} or RT_{new} is a first reach recomputation area (of iteration step i).

5.6 Second Reach Recomputation Area

By now, we have identified a reach update area R and a first reach recomputation area N_1. We can recompute reach-bounds of each edge $(u, v) \in R$ by growing a partial tree rooted at w for each vertex $w \in N_1$. The recomputation is done like in the static algorithm but we only recompute the penalty reach-bounds of edges contained in R.

Our remaining problem is that the reach-bounds we compute that way are not necessarily valid. In this section we describe how we can find a set of vertices, the *second reach recomputation area* on which partial trees have to be additionally grown to guarantee the correctness of the computed penalty reach-bounds.

Definition 15 (2nd Reach Recomputation Area)

Given a first reach recomputation area N_1 (of iteration step i) and a reach update area R (of iteration step i). A set of vertices N_2 such that for each edge (u, v) in R

$$\max_{x \in N_1 \cup N_2} \left\{ penReach_{T_{x/new}}(u, v) \right\} = \max_{x \in G_{i/new}} \left\{ penReach_{T_{x/new}}(u, v) \right\}$$

or

$$\max_{x \in N_1 \cup N_2} \left\{ penReach_{T_{x/new}}(u, v) \right\} > \epsilon_i$$

holds is called a *second reach recomputation area* (of iteration step i with respect to N_1 and R).

Basic Method. Given an edge (u, v) of the reach update area we already know a proceeding to find a set N of vertices on which we have to grow partial trees to recompute the $penReach_{G_{i/new}}(u, v)$: we have to consider all partial trees built on $G_{i/new}$ that contain (u, v). We get the roots of these by growing a reverse partial tree rooted at v on $G_{i/new}$. The set of all vertices included in the branch of the resulting 'reverse' shortest paths tree that starts with the edge (u, v) is such a set N and therefore $N \setminus N_1$ is a second reach recomputation area.

We also know that we can find the according vertices of all edges by growing one partial tree initialized with the target vertices of all edges. The handicap of this 'lazy' method is that we have to consider all vertices of the resulting reverse shortest paths tree and cannot restrict ourselves to the vertices included in special branches.

Sophisticated Method. There is a simple way to identify edges of the reach update area S whose penalty reach-bounds are already correct after the recomputation by growing partial trees rooted only at vertices of the first reach recomputation area N_1. We denote the penalty reach of an edge (u, v) computed only by considering partial trees grown on $G_{i/new}$ with roots in N_1 as $penReach_{N_1}(u, v)$ and the penalty reach of the actual iteration step on the original computation as $penReach_{old}(u, v)$. We remember that we modified the static reach computation to store for each edge (u, v) the root $root_{(u,v)}$ of a partial tree responsible for the penalty reach bound of (u, v).

First of all we do not have to consider each edge (u, v) with $penReach_{N_1}(u, v)$ greater than ϵ_i for further recomputation because the resulting reach-bound of (u, v) will be dumped anyway.

We can assign to each edge $(u, v) \in R$ with $penReach_{N_1}(u, v) \leq \epsilon$ and a valid reach-bound computed in the original computation until (and including) the actual step (that is an edge not included in $G_{i+1/old}$) one of the following four cases:

1. [$root_{(u,v)} \in N_1$ and $penReach_{old}(u, v) > penReach_{N_1}(u, v)$ **]**
In this case there may be a partial tree T_x with root x outside of N_1 such that (u, v) is on T_x and $penReach_{T_x}(u, v) > penReach_{N_1}(u, v)$. Therefore the tentative computed penalty reach-bound $penReach_{N_1}(u, v)$ does not have to be correct and has to be considered for a second recomputation.

2. [$root_{(u,v)} \in N_1$ and $penReach_{old}(u, v) \leq penReach_{N_1}(u, v)$ **]**
In this case the tentative computed penalty reach-bound $penReach_{N_1}(u, v)$ is already correct: because of the construction of N_1 all partial trees that have changed are rooted at an element of N_1. We know by the result of the original computation that no partial tree rooted outside N_1 is responsible for a reach of (u, v) greater than the actual known.

3. [$root_{(u,v)} \notin N_1$ and $penReach_{old}(u, v) > penReach_{N_1}(u, v)$ **]**
In this case the tentative computed penalty reach-bound $penReach_{N_1}(u, v)$ is not correct but we know that the old value $penReach_{old}(u, v)$ stays a correct penalty reach-bound: because of the construction of N_1 all partial trees that have changed are rooted at an element of N_1. We know that no partial tree rooted at an element in N_1 is responsible for a reach-bound greater than $penReach_{old}(u, v)$ and since all other partial trees do not have changed $penReach_{old}(u, v)$ stays a correct bound.

4. [$root_{(u,v)} \notin N_1$ and $penReach_{old}(u, v) \leq penReach_{N_1}(u, v)$ **]**
In this case the tentative computed penalty reach-bound $penReach_{N_1}(u, v)$ is already correct, the argumentation is the same as in case 2.

Conclusion. We summarize the results: assume penalty reach-bounds $penReach_{N_1}(u, v)$ have been computed for each (u, v) on the reach update area R by growing partial trees on N_1. Then the reach-bounds of all edges (u, v) are correct for which a valid reach-bound has been computed until (and including) the actual step of the original computation and which either apply to case 2 or case 4. Edges which apply to case 3 keep their original reach-bounds. For edges with $penReach_{N_1}(u, v)$ greater than ϵ no reach-bound will be computed in the actual step. The reach-bounds computed for the remaining edges of the given reach update area are possibly still uncorrect.

Hence we can compute a second reach recomputation area using either the eager or the lazy basic method but grow the reverse partial tree(s) considering only edges with possibly still incorrect and unknown penalty reach-bounds.

Theorem 6 (Construction of 2nd Reach Recomputation Area)

Let R be a reach update area (of iteration step i). Let N_1 be a first reach recomputation area (of iteration step i). Let $\tilde{R}$ be the set of all edges (u,v) in R with $penReach_{N_1}(u,v) \le \epsilon_i$. Let

$$\begin{aligned} R_1 &= \Big\{(u,v) \mid (u,v) \in \tilde{R}, (u,v) \in G_{i+1/old}\Big\} \\ R_2 &= \Big\{(u,v) \in \tilde{R}, (u,v) \notin G_{i+1/old}, root_{(u,v)} \in N_1\Big\} \\ H &= \Big\{r \mid r \in V, r \notin N_1, \exists (u,v) \in \tilde{R} : root_{(u,v)} = r\Big\} \end{aligned}$$

Let S be

$$\Big\{(u,v) \in R_2 \mid \max_{x \in N_1} \Big\{penReach_{T_{x/new}}(u,v)\Big\} < \max_{x \in G_{i/old}} \Big\{penReach_{T_{x/old}}(u,v)\Big\}\Big\}$$

Then the set of vertices that lie on the branch of a reverse partial tree grown on $G_{i/new}$ and rooted at v that starts with the edge (u,v) where $(u,v) \in R_1 \cup S$ unioned with H is a second reach recomputation area (eager construction).

Then the set of all vertices that lie on the reverse partial tree grown on $G_{i/new}$ and rooted at the end vertices of edges in $R_1 \cup S$ unioned with H is a second reach recomputation area (lazy construction).

Improvement using more memory

The sophisticated construction of the second reach recomputation area can only sort out edges which either already have a tentative reach-bound greater than ϵ after building partial trees on the first reach recomputation area or for which reach-bounds have been computed in the original computation until (and inclusive) the actual iteration step.

We can sort out even more edges if we store for each edge (u,v) *and* each iteration step i the root of a shortest path responsible for the penalty reach-bound of (u,v) at step i. Note that we store the root even if the reach-bound is greater than ϵ and will be dumped.

Then we know that the reach-bound of each edge (u,v) of the reach update area for which no reach-bound has been computed in the actual step of the original computation and for which the stored partial tree root is outside the first reach recomputation area will not be computed in the recomputation at the actual step. Therefore we do not have to consider (u,v) when we construct a second reach recomputation area.

5.7 Proof of Correctness

We have to show that all reach bounds recomputed by an iteration step are valid.

The correctness of the reach-bounds of edges not in the reach update area follows directly from the definition of reach update area. The correctness of the eager and the lazy construction of a reach update area follows from Theorem 2, page 70 and Theorem 3, page 72. The correctness of Theorem 2 follows from the case analysis in the same section. The correctness of Theorem 3 follows directly from Theorem 2 and Lemma 4, page 71.

The correctness of the reach-bounds of edges in the reach recomputation area follows from the definition of the second reach recomputation area. The correctness of the construction of the second reach recomputation area is due to Theorem 6, page 78. This theorem is proven in the text of the same section. To be able to apply Theorem 6, a first reach recomputation area must be correctly identified. The correctness of the lazy and eager construction of the first reach recomputation area is due to Theorem 4, page 74, Theorem 4, page 74.

We have used the following lemmata without proving them:

Proof 4 (Proof of the Partial Tree Bounding Lemma 1, page 59)
Denote by an unfinished path P on a partial tree (with size ϵ and delta δ) a path that either contains unfinished inner circle vertices or vertices with distance from their nearest inner circle predecessor of at most ϵ which are unfinished or for which unfinished successors exist. Obviously, a partial tree is grown until no unfinished paths exist.

An unfinished path has length of at most $2\epsilon + 2\delta$: the length of the edge outgoing from the root has length of at most δ, the sum of the lenghts of all other inner circle edges has length of at most ϵ. The length of the remaining edges is $\epsilon + \delta$ at most.

Let $\tilde{P}$ be a path on a partial tree. Then must either $\tilde{P}$ be unfinished or an unfinished path with length greater than the length of $\tilde{P}$ must be contained in the partial tree. Let the length of $\tilde{P}$ be greater than $2\epsilon + 2\delta$. This is a contradiction to the fact that the length of each unfinished path is $2\epsilon + 2\delta$ at most.

■

Proof 5 (Proof of Lemma 2, page 67)
Let $P = (x = x_0, \dots, x_n = u)$ be a x-u-path on a partial tree with size ϵ and delta δ rooted at x and grown on $G = (V, E_i)$. Then P is a shortest path on the graph $\tilde{G} = (V, \tilde{E}_i)$ where $\tilde{E}_i = E \setminus \{(u,v) \mid len(u,v) > \delta\}$. Therefore $(u = x_n, \dots, x_0)$ is a shortest path on the reverse graph of $\tilde{G}$. By Lemma 1 we know that the length of P is $2\epsilon + 2\delta$ at most.

Therefore $(u = x_n, \dots, x_0)$ is included in the shortest paths tree rooted at u on the reverse graph of $\tilde{G}$ which contains all vertices with distance from u of at most $2\epsilon + 2\delta$.
■

Proof 6 (Proof of Lemma 3, page 68)
As subpath of a shortest path, $(u_1, \dots u_n)$ is also a shortest path. Because of Lemma 1

is $len(u_1, \ldots, u_n)$ lower or equal to $2\epsilon + 2\delta$. Therefore it is included in the max partial tree rooted at $\{u\}$.

■

Proof 7 (Proof of Lemma 4, page 71)
Given a set of vertices N_1. The max partial tree rooted at N_1 with size ϵ and delta δ contains all vertices that have a distance of at most $2\epsilon + 2\delta$ to at least one vertex n in N_1. Since n is also contained in $N_1 \cup N_2$ for an arbitrary set of vertices N_2, v is also contained in the max partial tree rooted at $N_1 \cup N_2$. The same argumentation holds for reverse partial trees.

■

5.8 Implementation of the Dynamic Algorithm

Here we give the pseudo-code for the dynamic recomputation algorithm described in the chapter. The code refers to the lazy construction of a reach update area (Theorem 3, page 72), the lazy construction of a first reach recomputation area (Theorem 5, page 75) and the sophisticated lazy construction of a second reach recomputation area (Theorem 6, page 78). The improvement on the second reach recomputation area sketched on page 78 is not included.

The following algorithm is a sub-routine used by Algorithm 8.

Algorithm 7: ComputeReach

```
input: reach update area $RUA$
       reach recomputation area $RRA$

foreach vertex $v$ in $RRA$ do
    $T$:= partial tree on $G_{i/new}$ rooted at $v$
    foreach edge $e$ in $T \cap RUA$ do
        if $Reach_T(e) > Reach_{new}[e]$ then
            $Reach_{new}[e] := Reach_T(e)$
            $PartialTreeRoot_{new}[e] := v$
            $ReachIterationStep_{new}[e] := i$
```

Algorithm 8: Lazy Dynamic Reach Bound Recomputation

```
input : Graph $G = (V, E)$, $Len_{old}[]$, $Len_{new}[]$ $epsilon[], delta[]$
        $Reach_{old}[]$, $ReachIterationStep_{old}[]$, $PartialTreeRoot_{old}[]$
ouput: $Reach_{new}[]$, $ReachIterationStep_{new}[]$, $PartialTreeRoot_{new}[]$

/* all max/reverse partial trees grown on iteration step $i$ have size $epsilon[i]$ and
delta $delta[i]$                                                                    */

1  Initialize $Reach_{new}$ with $Reach_{old}$, $ReachIterationStep_{new}$ with $ReachIterationStep_{old}$,
   $PartialTreeRoot_{new}$ with $PartialTreeRoot_{old}$

2  for $i := 1$ to maxiterationstep do
3      RUA=∅;RUA2=∅; RRA1=∅; RRA2=∅;
4      $G_{new} = (V, E_{new}), E_{new} = \{e \in E \mid ReachIterationStep_{new}[e] \geq i\}$
5      $G_{old} = (V, E_{old}), E_{old} = \{e \in E \mid ReachIterationStep_{old}[e] \geq i\}$
6      UPDATE $U_i$, $N_{ld}$, $N_{bc,cs}$, $\tilde{N_{ld}}$, $\tilde{N}_{bc,cs}$

       /* ReachUpdateArea, 1stReachRecomputationArea                                */
7      $T_{old}$ = max partial tree grown on $G_{old}$, rooted at $\tilde{N}_{ld} \cup \tilde{N}_{bd,cs}$
8      $T_{new}$ =max partial tree grown on $G_{new}$, rooted at $\tilde{N}_{ld} \cup \tilde{N}_{bd,cs}$
9      $RT_{old}$ = reverse partial tree grown on $G_{old}$, rooted at $N_{ld} \cup N_{bd,cs}$
10     $RT_{new}$ = reverse partial tree grown on $G_{new}$, rooted at $N_{ld} \cup N_{bd,cs}$
11     $RUA$ = set of edges $(u, v)$ where $u, v \in T_{old}$ or $u, v \in T_{new}$ or $u, v \in RT_{old}$ or $u, v \in RT_{new}$
12     $RRA1$ = set of all vertices contained in $RT_{old}$, $RT_{new}$ or $U_i$
13     foreach $e$ in $RUA$ do $Reach_{new}(e) := 0$
14     ComputeReach(RUA, RRA)

       /* 2ndReachRecomputationArea                                                   */
15     foreach edge $e$ in $RUA$ do
16         if ($Reach_{new}[e] \leq epsilon[i]$, $ReachIterationStep_{old}[e] \leq i$, $Reach_{new}[e] < Reach_{old}[e]$ and
           $PartialTreeRoot_{old}[e] \in RRA1$) or ($ReachIterationStep_{old}[e] > i$ and
           $Reach_{new}[e] \leq epsilon[i]$) then
17             insert $e$ into $RUA2$
18         if $Reach_{new}[e] \leq epsilon[i]$, $ReachIterationStep_{old}[e] \leq i$, $Reach_{new}[e] < Reach_{old}[e]$ and
           $PartialTreeRoot_{old}[e] \notin RRA1$ then
19             $Reach_{new}[e] := Reach_{old}[e]$
20             $PartialTreeRoot_{new}[e] = PartialTreeRoot_{old}[e]$
21             $ReachIterationStep_{new} = i$
22     RRA2=set of all vertices contained in a reverse partial tree rooted at $RUA2$ grown on $G_{i/new}$
23     ComputeReach(RUA2, RRA2)

       /* delete reach bounds greater than $epsilon[i]$                                */
24     foreach edge $e$ in $RUA$ with $Reach_{new}[e] > epsilon[i]$ and $ReachIterationStep_{old}[e] \leq i$ do
25         $Reach_{new}[e] = 0$
26         $ReachIterationStep_{new}[e] = \infty$
27         $PartialTreeRoot_{new}[e] = null$
28     foreach edge $e$ in $RUA$ with $Reach_{new}[e] > epsilon[i]$ and $ReachIterationStep_{old}[e] > i$ do
29         $Reach_{new}[e] = Reach_{old}[e]$
30         $ReachIterationStep_{new}[e] = ReachIterationStep_{old}[e]$
31         $PartialTreeRoot_{new}[e] = PartialTreeRoot_{old}[e]$
```

6 Experiments

Road networks are extremly sparse and contain a certain hierarchy (of importance with respect to long shortest-paths) which explains why reach-based pruning performs well on road networks. Goldberg, Kaplan and Werneck experimentally have shown the extremly good performance of reach-based pruning on road networks in [GKW05].

In this chapter we report the results of some own experiments computing reach bounds on road networks.

6.1 Choice of tuning parameters

In [GKW05] the following strategy for selecting ϵ_i is proposed: given a parameter k, we choose k vertices at random and grow, for each vertex, a shortest-paths tree with exactly $\lfloor n/k \rfloor$ vertices. ϵ_0 is assigned to be twice the minimum of the distance labels of the last scanned vertex over all shortest paths trees. Furthermore $\min\{500, \lceil \sqrt{n} \rceil / 3\}$ is proposed as good value for the parameter k. Given a second parameter α, ϵ_i is computed by $\epsilon_i = \alpha^i \epsilon_0$. Here $\alpha = 3.0$ is reported to be a good value.

Since the delta-rule is not stated in [GKW05], no good values for delta are given. To get a first impression what good values for delta could be, we have a look at the distribution of the edge lengths on the underlying road networks. Figure 6.1 shows the distribution of the edge lengths on a graph representing Germany. Values between *epsilon* fourth and *epsilon* half turned out to give a good tradeoff between the speed of the preprocessing and the quality of the computed reach bounds.

6.2 Description of the tested graphs

We have tested the reach-bound preprocessing and query on graphs mapping parts of the road network of Europe. The graphs were provided by the PTV AG and the length of an edge on a graph refers to the Euclidean distance between the source and the target vertex of the edge. The following table gives an overview and a short description of all graphs used.

name	**#vertices**	**#edges**	**description**
ger	4.377.787	10.997.366	road network of germany
dnk	473.537	1.075.012	road network of denmark
fin	460.693	1.020.008	road network of finland
ka50.000	49.625	125.018	road network of a bounding box around karlsruhe
ka100.000	99.529	252.530	road network of a bounding box around karlsruhe
ka200.000	199.739	501.948	road network of a bounding box around karlsruhe
dkb100.000	99.878	250.490	road network of a rural area in franconia
st100.000	99.928	258.072	road network of an urban area in stuttgart

On page 49 we justified the delta rule by mentioning that most road networks contain many short and only few long edges. The following figure 6.1 shows the distribution of the edge lengths on the graph ger.

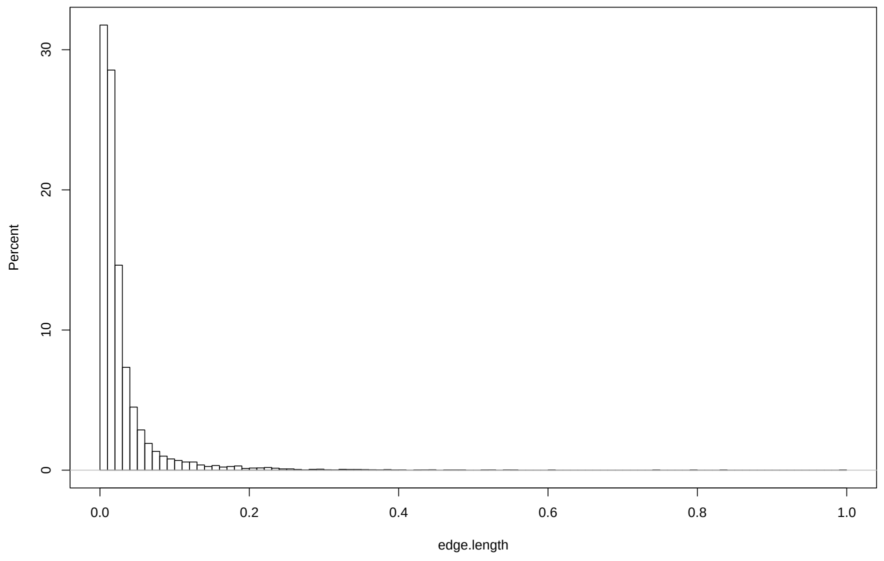


Fig. 6.1: Distribution of the lengths of 10000 randomly chosen edges of the germany graph where edge lengths correspond to travel times. The edge lengths are given relative to the length of the longest of the 10000 edges.

We want to remind the reader that at iteration step i of the static reach-bound preprocessing only valid reach bounds for edges with exact reach lower than ϵ_i can be computed. Only edges with an already computed valid upper reach bound are removed from the input-graph of the next iteration step. The strategy used for the static preprocessing mainly depends on the assumption that the graph strongly sparsificates after each iteration step. The strategy used for the reach-query also mainly depends on the assumption that most edges have low reach while only few edges with high reach exist. The next figure 6.2 shows the distribution of the exact reach values on the graph KA50.000.

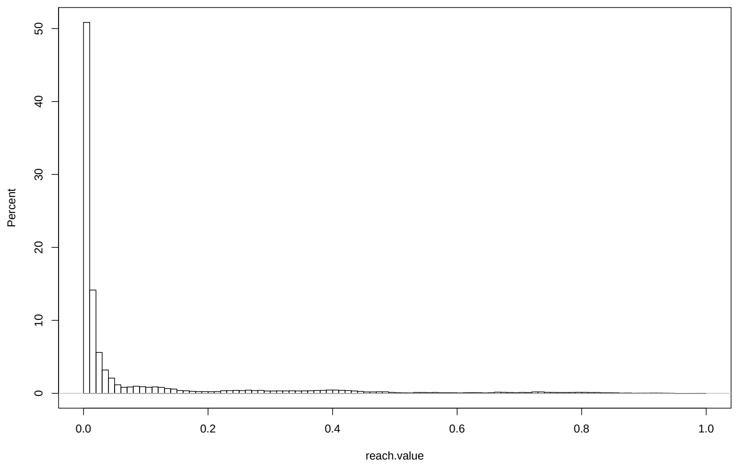


Fig. 6.2: Distribution of exact reach values on KA50.000

Figure 6.1 and 6.2 show that the distribution of the edge lengths and the distribution of the exact reach values are very similiar. We checked on a correlation between edge lengths and exact reach values on KA50.000. The following scatterplot and the little correlation coefficient of 0.165 suggest that such a correlation does not exist.

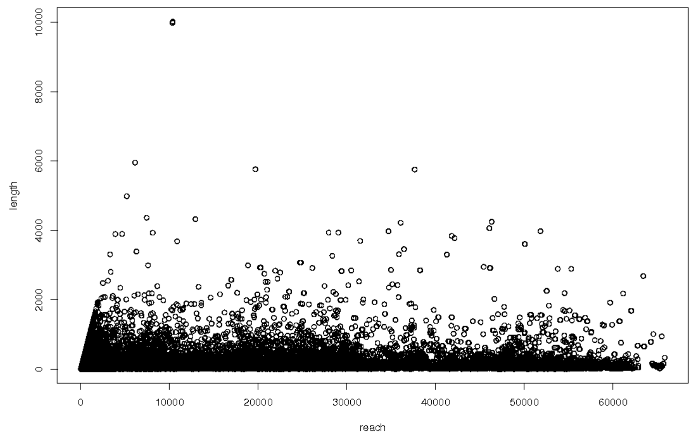


Fig. 6.3: Correlation between edge length and exact edge reach on KA50.000

6.3 Preprocessing Effort and Speed-Up of the Static Algorithm

We recall that the reach-bound computation described in [GKW05] differs from the one described in this work. Therefore we experimentally tested preprocessing and query of our algorithm on the graphs described in the last section.

The choice of the tuning parameters described in [GKW05] turned out to be also a good compromise between preprocessing time and quality of the computed reach bounds for our variant of the reach-bound computation. Furthermore we set δ_i to be $0.3\epsilon_i$.

In order to be independent from concrete implementations and hardware we measured the average speed-up by the average quotient of the number of vertices visited by Dijkstra's algorithm and the number of vertices visited by the bidirectional bound algorithm after performing 1000 randomly chosen s-t-queries. The preprocessing effort is measured by the number of partial trees built and the average number of vertices visited by these partial trees.

name	**#partial trees built**	**avg #vertices**	**avg speed-up**
dnk	778.606	11.513	10.1
fin	641.637	14.359	11.4
ka50.000	77.093	3571	7.0
exact ka50.000	49.625	49.625	7.5
ka100.000	135.939	11.216	7.0
ka200.000	360.881	11.629	5.8
dkb100.000	201.892	7.775	5.0
st100.000	184.288	6.780	5.6

6.4 Example for the sparsification during the reach-bound computation

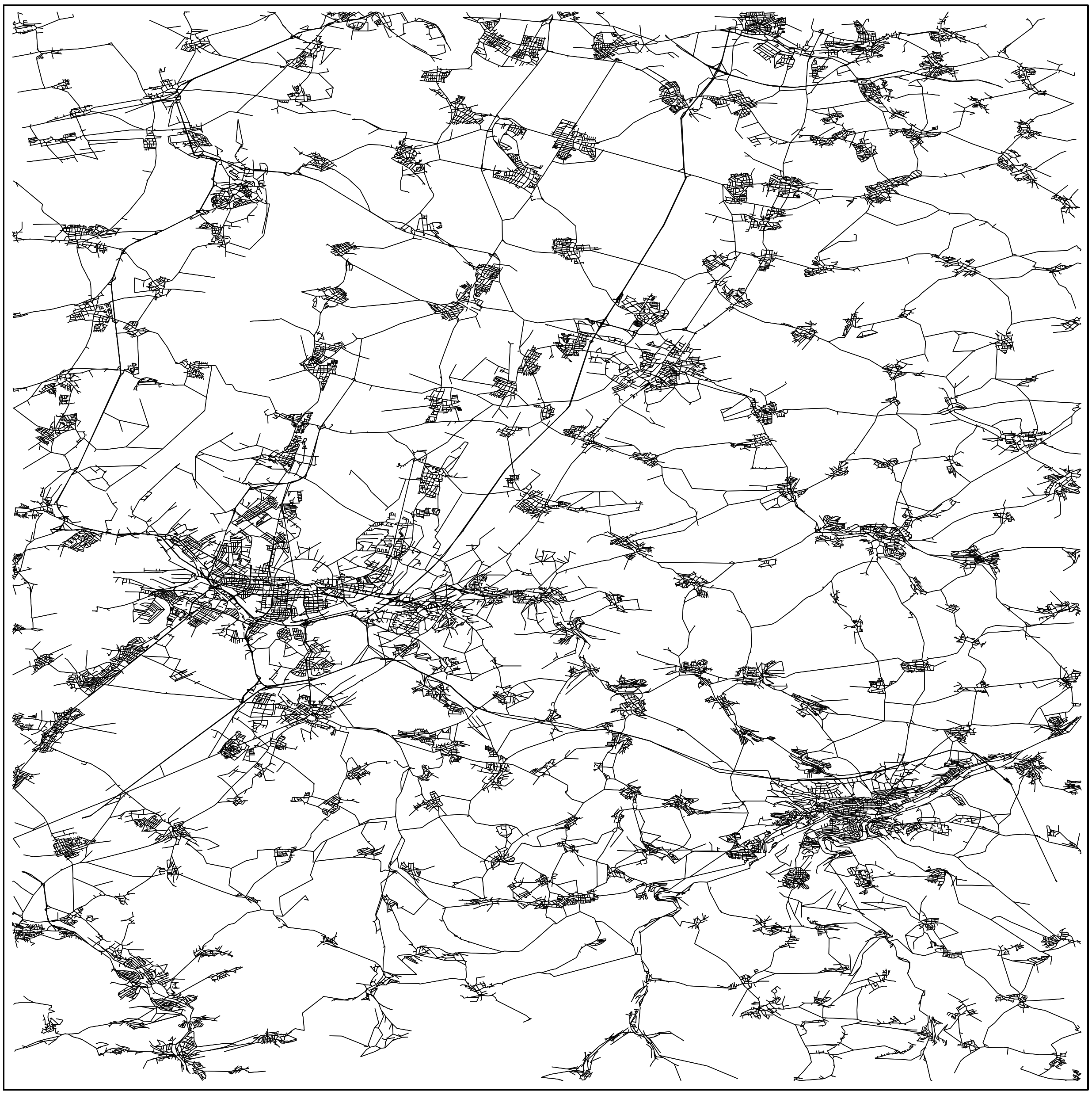


Fig. 6.4: The graph KA50.000. The number of edges is 125.018.

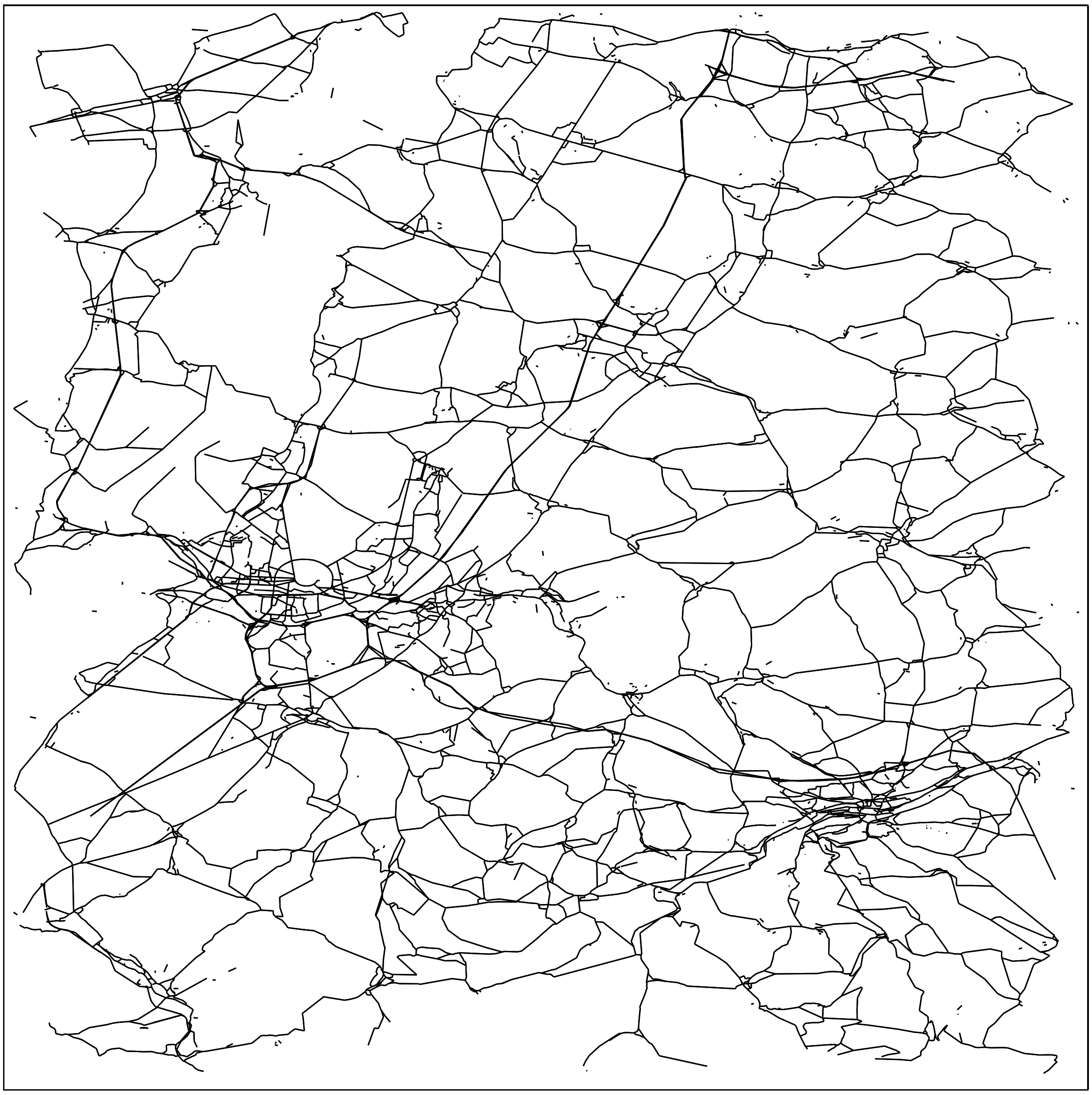


Fig. 6.5: The sparsificated KA50.000 after the first iteration step. The number of edges is 31.177.

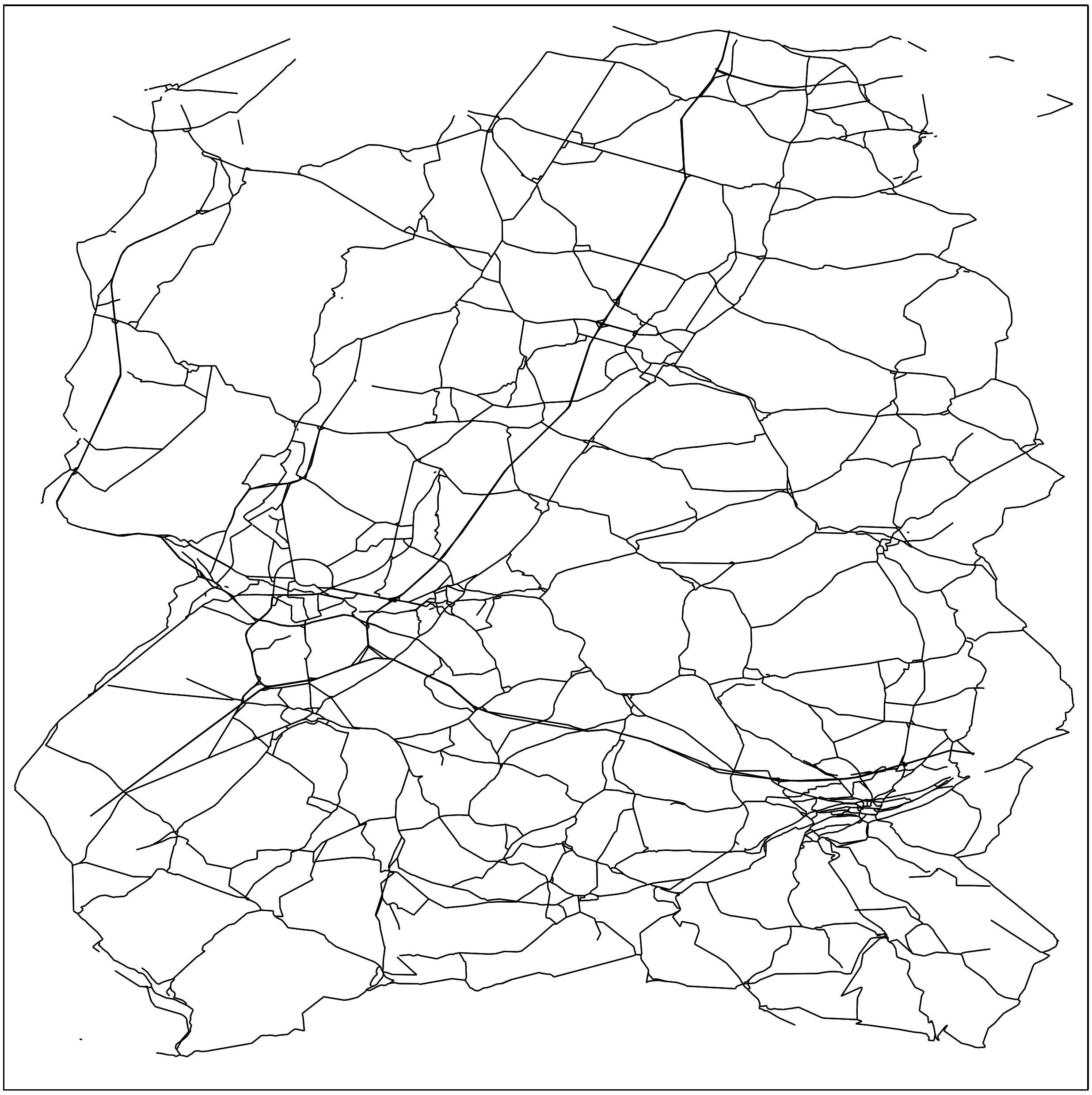


Fig. 6.6: The sparsificated KA50.000 after the second iteration step. The number of edges is 19.632.

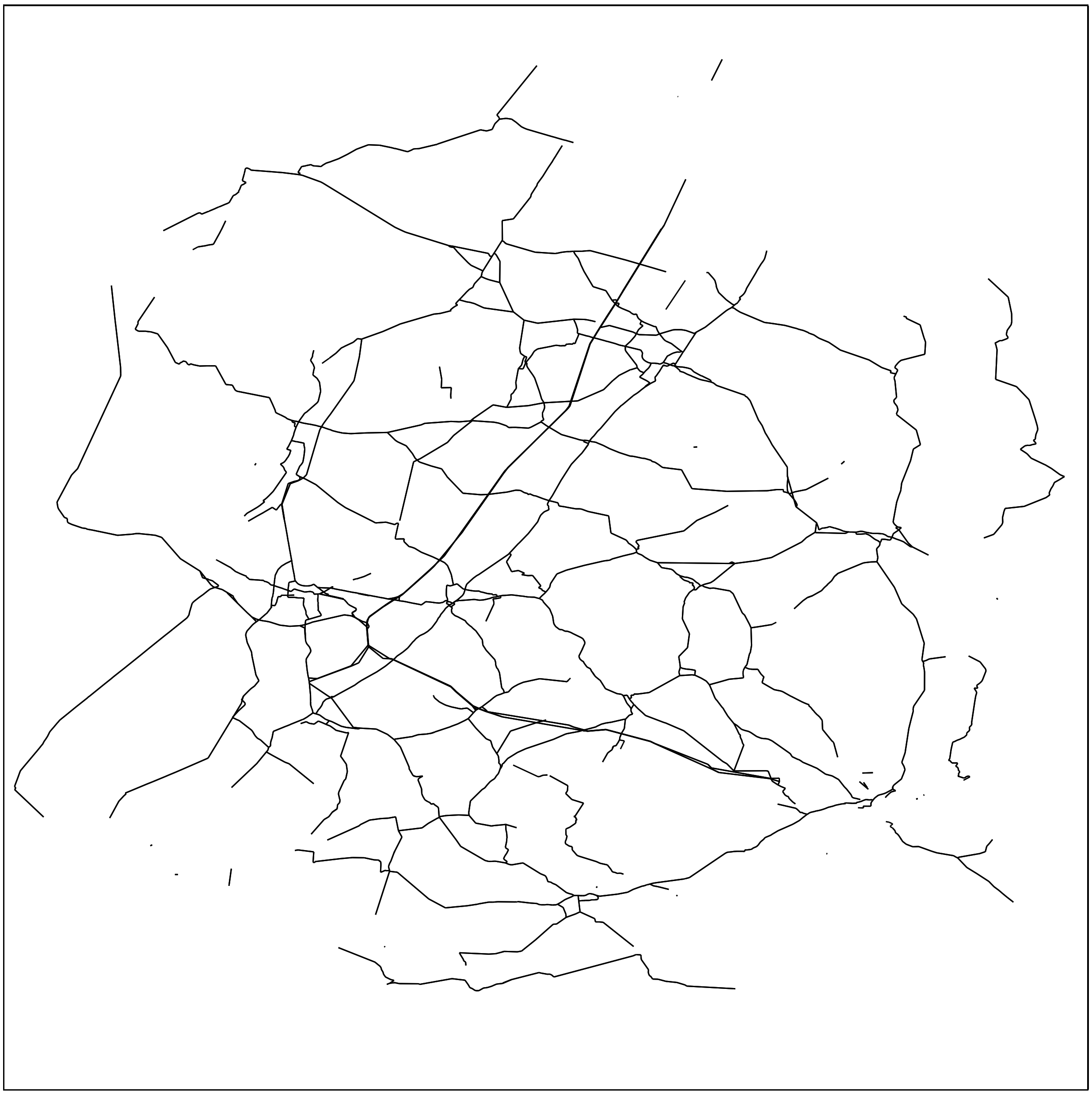


Fig. 6.7: The sparsificated KA50.000 after the third iteration step. The number of edges is 7.020.

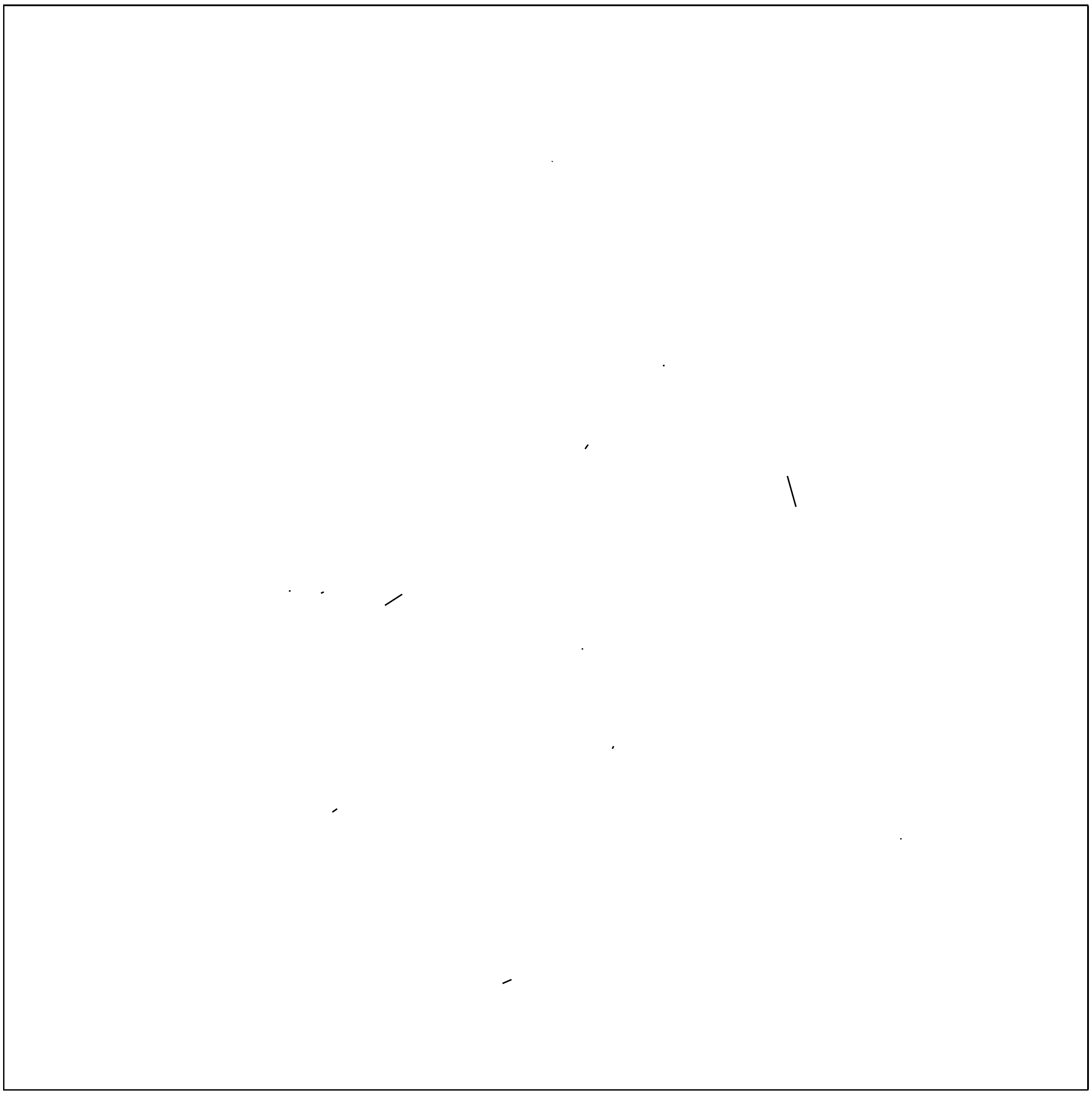


Fig. 6.8: The sparsificated KA50.000 after the fourth iteration step. The number of edges is 16.

7 Final Remarks

Conclusion. In this work, we gave an overview of some of the recent techniques used to speed-up Dijkstra's algorithm exploiting additional, preprocessed data. Considering most described speed-up techniques, we gave proposals how to dynamically update the preprocessing after a set of edges on the underlying graph have changed their lengths. Here, we focused on *landmarks*, *multi-level graphs* and precomputed *reach-bounds*.

The preprocessing algorithms of landmarks and multi-level graphs need a pre-selection of some vertices on the graph. The update strategies we proposed for both data structures compute the same preprocessed data as a recomputation from scratch by the static algorithm would provide *if* the choice of these vertices stayed the same. The static algorithm for computing upper-bounds for reach values requires two tuning parameters that usually are chosen using information on the underlying graph. The update strategy we proposed for these reach-bounds computes the same bounds as a full recomputation from scratch by the static algorithm would provide *if* these tuning parameters stayed the same.

Outlook. The runtime of all update algorithms is heuristic. In the worst case the usage of the update strategy may take more time than a full recomputation from scratch would need. Therefore it is important to experimentally study the performance of the update algorithms when applying them to real-world data (that consists of using real-world graphs and applying real-world edge updates).

Another task is to find criteria that decide whether the selection of the separator vertices used to build multi-level graphs remains good after an update on the graph and that are fast to determine. Strategies for re-choosing bad separator vertices have to be found and the update algorithm must be altered to be able to cope with re-chosen separator vertices.

The usage of shortcuts is reported to speed-up the static preprocessing of reach-bounds by factor 15 and the query by factor 5. Therefore, the most promising improvement on the reach-bounds update algorithm seems to be an enhancement that enables it to deal with shortcuts.

Since highway hierarchies are one of the fastest available speed-up techniques (in both query and preprocessing) a dynamic variant of that technique is desirable.

Finally, we want to mention that we concentrated on solutions using only few additional memory. The development of methods using more memory may further speed-up the update of the preprocessed data.

Bibliography

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows.* Prentice Hall, 1993.

[BE05] Ulrik Brandes and Thomas Erlebach, editors. *Network Analysis: Methodological Foundations [outcome of a Dagstuhl seminar, 13-16 April 2004]*, volume 3418 of *Lecture Notes in Computer Science.* Springer, 2005.

[CLL90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest R. L. *Introduction to Algorithms.* MIT Press, 1990.

[DI05] Camil Demetrescu and Giuseppe F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. *To appear in Journal of Computer and System Sciences*, 2005. Special issue devoted to the best papers selected from the 42th Annual IEEE Symposium on Foundations of Computer Science (FOCS'01).

[FMSN96] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic output bounded single source shortest path problem. In *SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 212–221, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[FMSN98] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights. In *European Symposium on Algorithms*, pages 320–331, 1998.

[FMSN00] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *J. Algorithms*, 34(2):251–281, 2000.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[GKW05] A. V. Goldberg, Haim Kaplan, and Renato Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. Technical Report MSR-TR-2005-132, Microsoft Research (MSR), October 2005.

[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.

[Hig98] Nicholas J. Higham. *Handbook of Writing for the Mathematical Sciences.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 1998.

[HK00] David Harel and Yehuda Koren. A fast multi-scale method for drawing large graphs. In *AVI '00: Proceedings of the working conference on Advanced visual interfaces*, pages 282–285, New York, NY, USA, 2000. ACM Press.

[HNBR68] P. Hart, N. Nilsson, B., and Raphael. A formal basis for the heuristic determination of minimum-cost paths. *IEEE Trans. on Systems Science and Cybernetics*, SSC-4(2):100–107, July 1968.

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

[HSW06] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. January 2006. To appear at ALENEX 2006.

[KMS04] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of shortest path computation. Article 42, Technische Universität Berlin, Fakultät II Mathematik und Naturwissenschaften, 2004.

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

[MSS$^+$05] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speed up Dijkstra's algorithm. In *WEA*, pages 189–202, 2005.

[RR96] G. Ramalingam and Thomas Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158(1–2):233–277, 1996.

[Sch05a] Dominik Schultes. Highway hierarchies hasten exact shortest path queries. Master's thesis, Universität Karlsruhe (TH), Fakultät Informatik, 2005.

[Sch05b] Frank Schulz. *Timetable Information and Shortest Paths.* PhD thesis, Universität Karlsruhe (TH), Fakultät Informatik, 2005.

[SS05] Sanders and Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA: Annual European Symposium on Algorithms*, 2005.

[SS06] Sanders and Schultes. Engineering highway hierarchies. In *Draft, submitted to ESA 2006*, 2006.

[Str00] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 2000.

[SWW99] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm online: An empirical case study from public railroad transport. In *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE 1999)*, volume 1668 of *LNCS*, pages 110–123. Springer, 1999.

[SWW00] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm online: An empirical case study from public railroad transport. *J. Experimental Algorithmics*, 5(12), 2000.

[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.

[Wil05] Thomas Willhalm. *Engineering Shortest Paths and Layout Algorithms for Large Graphs*. PhD thesis, Universität Karlsruhe (TH), Fakultät Informatik, 2005.

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

[WWZ04] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Dynamic shortest path containers. In *Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'03)*, volume 92 of *Electronic Notes in Theoretical Computer Science*, pages 65–84. Elsevier, 2004.