Electronic Notes in Theoretical Computer Science 92 No. 1 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume92.html 19 pages

Dynamic Shortest Paths Containers

Dorothea Wagner [a,1], Thomas Willhalm [a,1], and Christos Zaroliagis [b,1]

[a] *Institut für Logik, Komplexität und Deduktionssysteme*
Universität Karlsruhe
76128 Karlsruhe, Germany

[b] *Computer Technology Institute, and*
Department of Computer Engineering & Informatics
University of Patras, 26500 Patras, Greece

Abstract

Using a set of geometric containers to speed up shortest path queries in a weighted graph has been proven a useful tool for dealing with large sparse graphs. Given a layout of a graph $G = (V, E)$, we store, for each edge $(u, v) \in E$, the bounding box of all nodes $t \in V$ for which a shortest u-t-path starts with (u, v). Shortest path queries can then be answered by Dijkstra's algorithm restricted to edges where the corresponding bounding box contains the target.

In this paper, we present new algorithms as well as an empirical study for the dynamic case of this problem, where edge weights are subject to change and the bounding boxes have to be updated. We evaluate the quality and the time for different update strategies that guarantee correct shortest paths in an interesting application to railway information systems, using real-world data from six European countries.

Key words: geometric container, dynamic shortest path, graph layout.

1 Introduction

In this paper we consider a typical application in traffic, and in particular in railway, systems where a central server has to answer a huge number of customer queries asking for their best itineraries. The most frequently encountered applications of the above scenario involve route planning systems

[1] This work was partially supported by the Human Potential Programme of the European Union under contract no. HPRN-CT-1999-00104 (AMORE). Part of this work was done while the second author was visiting the Computer Technology Institute in Patras.

©2003 Published by Elsevier Science

PSfrag replacements

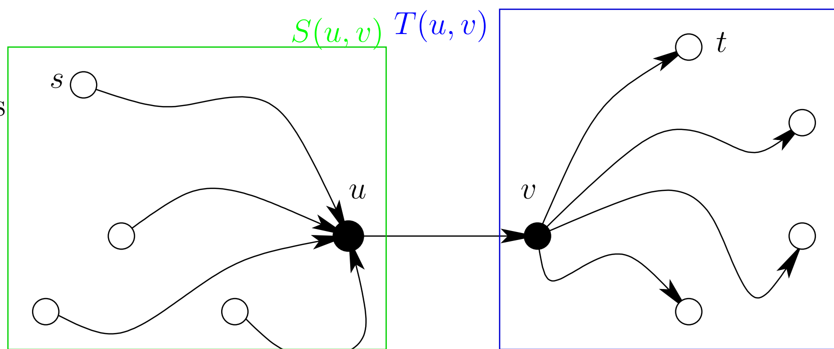


Fig. 1. A target container $T(u, v)$ contains (at least) all nodes t for which a shortest path from u to t starts with the edge (u, v). Analogously, a source container $S(u, v)$ contains all nodes s for which a shortest path from s to v ends with the edge (u, v).

for cars, bikes and hikers [21,2], or scheduled vehicles like trains and busses [19,11,13,17,10]. Further applications include spatial databases [18] and web searching [3]. Users of such a system continuously enter their requests for finding their "best" connections.

The algorithmic core problem that underlies the above problem is a special case of the single source shortest path problem on a given directed graph with nonnegative edge lengths related to a layout of the graph which is also provided. The particular graph is quite large (though sparse), and hence space requirements are only acceptable to be linear in the number of nodes.

In [17], angular sectors were introduced to speed up the processing of such shortest path queries. In a preprocessing step, the angular sector of each edge is determined that contains all nodes to which a shortest path using this edge exists. The shortest path queries are then answered by Dijkstra's algorithm [5] restricted to edges where the target node is inside the angular sector. Note that this method is guaranteed to find a shortest path, but it requires a time consuming all-pairs shortest paths computation as a preprocessing.

A recent experimental study in [20] replaces the angular sectors by other convex geometric containers and compares their impact on the number of visited nodes and the running time. Surprisingly, simple bounding boxes turn out to produce the fastest algorithm and are also competitive in the number of visited nodes.

This idea of geometric pruning can be extended to bi-directional search [12]. A second set of bounding boxes is determined by reversing all edges and running the preprocessing a second time on this modified graph. As illustrated in Figure 1, we will refer to the bounding boxes of this graph with reversed edges as "source containers" (containing the sources of shortest paths that end with this edge) in contrast to "target containers" (containing the target of shortest paths that start with this edge). A forward step in the bi-directional search checks the target containers whereas a backward step uses source containers.

All previous approaches, however, deal with the static version of the investigated problem. In this paper, we are concerned with the dynamic version of the above mentioned scenario; namely, with the case where the graph may dynamically change over time as streets may be blocked, built, or destroyed, and trains may be added or canceled. In this work, we present new algorithms that dynamically maintain geometric containers when the weight of an edge is increased or decreased (note that these cases cover also edge deletions and insertions). We also report on an experimental study with real-world railway data. Our experiments show that the new algorithms are 2-3 times faster than the naive approach of recomputing the geometric containers from scratch.

Our dynamic algorithms are perhaps the first results towards an efficient algorithm for the dynamic single source shortest path problem without using the output complexity model – introduced in [14,15] and extended in [7,8] – under which algorithms for the dynamic single source shortest path problem are usually analyzed. We would also like to mention that existing approaches for the dynamic all-pairs shortest paths problem (see e.g., [6,16,4,1,9], and [22] for a recent overview) are not applicable to maintain geometric containers, because of their inherent quadratic space requirements.

In the next section, a formal description of the dynamic shortest path containers and necessary definitions are given. The subsequent section 3 recapitulates the static case of the preprocessing. Section 4 contains algorithms to update geometric containers for weight decreases and weight increases. Experiments and results are described in section 5 before the conclusion in the last section.

2 Definitions

A directed simple *graph* G is a pair (V, E), where V is a finite set and $E \subseteq V \times V$. The elements of V are the *nodes* and the elements of E are the *edges* of the graph G. Throughout this paper, the number of nodes $|V|$ is denoted by n and the number of edges $|E|$ is denoted by m. A *path* in G is a sequence of nodes $u_1, \ldots, u_k$ such that $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$. The edges of a graph are *weighted* by a function $w : E \to \mathbb{R}_0^+$. We interpret the weights as edge lengths in the sense that the *length of a path* is the sum of the weights of its edges. Throughout the paper we assume that for all pairs $(s, t) \in V \times V$, the shortest path from s to t is unique. [2]

If n denotes the number of nodes, a graph (without multiple edges) can have up to n^2 edges. We call a graph *sparse*, if the number of edges m is in $O(n)$, and we call a graph *large*, if one can only afford a memory consumption in $O(n)$. In particular for large sparse graphs, n^2 space is not affordable.

We assume that we are given a *layout* $L : V \to \mathbb{R}^2$ in the Euclidean plane. For ease of notation we will identify a node $v \in V$ with its location $L(v) \in \mathbb{R}^2$

[2] This can be achieved by adding a small fraction to the edge weights, if necessary.

in the plane. Throughout the paper, we will assume that the layout is fixed.

We call a region $R \subset \mathbb{R}^2$ with

$$\left\{(x, y) \in \mathbb{R}^2 : x_{\min} \leq x \leq x_{\max} \wedge y_{\min} \leq y \leq y_{\max}\right\}$$

an (axes-parallel) *rectangle*. Given a set of points $P \subset \mathbb{R}^2$, the smallest rectangle R with $P \subset R$ is called the *bounding box* of P. Note that the bounding box of a finite set P always exists and is unique.

Definition 2.1 Let $G = (V, E)$, $w : E \to \mathbb{R}_0^+$ be a weighted graph with layout $L : V \to \mathbb{R}^2$. A rectangle $T(u, v)$ for an edge $(u, v) \in E$ is called a *(consistent) target container* of (u, v), if $T(u, v)$ contains (at least) all nodes t for which there is a shortest u-t-path starting with the edge (u, v). Similarly, a rectangle $S(u, v)$ for an edge $(u, v) \in E$ is called a *(consistent) source container* $S(u, v)$ of (u, v), if $S(u, v)$ contains (at least) all nodes s for which there is a shortest s-v-path ending with the edge (u, v).

Note that further nodes may be part of a target container. However, at least the nodes that can be reached by a shortest path starting with e must be in $T(e)$. We will refer to the additional nodes as *wrong nodes*, since they lead us the wrong way.

3 Creating Consistent Containers

We now describe in detail how to compute $T(s, x)$ for all edges $(s, x) \in E$. The complete algorithm is shown as Algorithm 1 (light gray lines indicate Dijkstra's original pseudocode, while the rest indicate our modifications). To determine $T(s, x)$ for every edge $(s, x) \in E$, Dijkstra's algorithm is run for each node $s \in V$. We keep a node array A where the entry $A[v]$, $v \in V$, stores the first edge (s, x) in a shortest s-v-path in G. This can be constructed in a way similar to that of a shortest path tree: Every time the distance label of a node v is adjusted via (u, v), we set $A[v]$ to (u, v), if $u = s$, and to $A[u]$, otherwise (lines 14–17). When a node u is removed from the priority queue, $A[u]$ holds the outgoing edge of s with which a shortest path from s to u starts. In line 18 and 19, for each edge (s, x) the bounding box of all nodes $y \in V$ with $A[u] = (s, x)$ is computed. It is a consistent target container for (s, x). The overall running time is $O(n^2 \log n)$, because Dijkstra's algorithm runs in $O(n \log n)$ time for sparse graphs. The storage requirement is $O(n)$.

Consistent source containers can be created by reversing the edges and running **Create-Containers** on this modified graph.

4 Updating Containers

If a weight of an edge is changed, some source and target containers must be enlarged to stay consistent. More precisely, for every new shortest path $u_0, u_1, \ldots, u_{k-1}, u_k$ in the graph, $T(u_0, u_1)$ and $S(u_{k-1}, u_k)$ have to be updated

Create-Containers
Input: graph $G = (V, E)$, weights $w : E \to \mathbb{R}_0^+$
Output: consistent target containers

```
 0  for all s ∈ V do
 1     set d(u) := ∞ for all nodes u ∈ V
 2     create empty priority queue Q
 3     insert source s in Q and set d(s) := 0
 4     while priority queue is not empty
 5        get node u with smallest tentative distance in Q
 6        for all neighbor nodes v of u
 7           set new-dist := dist(u) + w(u, v)
 8              if new-dist < dist(v)
 9                 if d(v) = ∞
10                    insert neighbor node v in Q with priority new-dist
11                 else
12                    set priority of neighbor node v in Q to new-dist
13                 set dist(v) := new-dist
14                 if u = s
15                    set A[v] := (s, v)
16                 else
17                    set A[v] := A[u]
18     for all nodes y ∈ V\{s}
19        enlarge the bounding box of A[y] to contain y
```

Algorithm 1. Create-Containers. Running a modification of Dijkstra's algorithm for all nodes $s \in V$ to create consistent target containers.

to include u_k and u_0, respectively. Throughout this section, we will mark variables before the update with the subscript "old" and updated values with the subscript "new".

4.1 Increasing an edge weight

Let us first consider the case of increasing the weight of an edge $(x, y) \in E$. We will show that it suffices to consider paths that start at vertices in $S_{\text{old}}(x, y)$ to correct all source containers (see Figure 2). The update itself can be achieved by running a truncated Dijkstra's algorithm for all these nodes.

Lemma 4.1 *Let $G = (V, E)$, $w : E \to \mathbb{R}_0^+$ be a weighted graph. Assume that the increase of the weight of an edge $(x, y) \in E$ creates a new shortest s-t-path P_{new} in G. Then, before the weight change, (x, y) is the last edge of a shortest s-y-path and the first edge of a shortest x-t-path.*

Proof. Let P_{old} denote the old shortest path from s to t. Since the weight $w(x, y)$ is increased, $(x, y) \in P_{\text{old}}$. Let P_{sy} denote the first part of this path P_{old} from s to y. Since a sub-path of a shortest path is again a shortest path, P_{sy} was the shortest path from s to y. For symmetric reasons, the first edge

PSfrag replacements

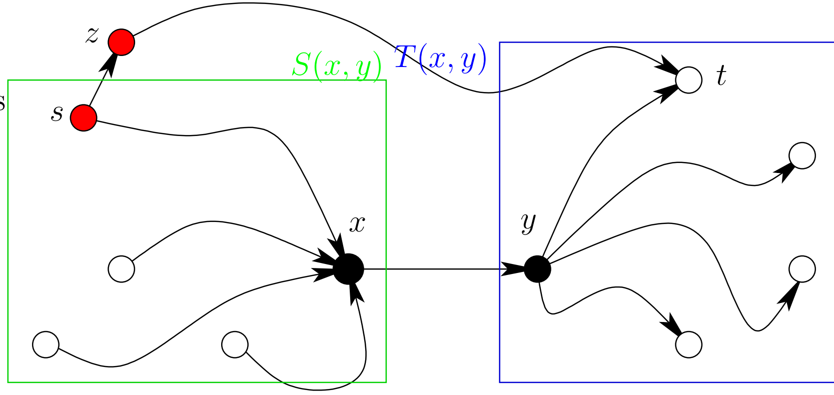


Fig. 2. When the weight of the edge (x, y) is increased, the source s of a new shortest path from s to t must be inside $S_{\text{old}}(x, y)$.

of a shortest x-t-path is (x, y). □

Corollary 4.2 *Let $G = (V, E)$, $w : E \to \mathbb{R}_0^+$ be a weighted graph. Assume that the increase of the weight of an edge $(x, y) \in E$ creates a new shortest s-t-path P_{new} in G. Then*

$$s \in S_{\text{old}}(x, y) \qquad \text{and} \qquad t \in T_{\text{old}}(x, y).$$

So, to enlarge the containers, it suffices to search for shortest paths that start from a node in $S_{\text{old}}(x, y)$ and end with a node in $T_{\text{old}}(x, y)$. Running **Create-Containers** restricted to nodes in $S_{\text{old}}(x, y)$ therefore fixes the source containers. However, it is possible to further truncate Dijkstra's algorithm using the following Lemma.

Lemma 4.3 *Let $G = (V, E)$, $w : E \to \mathbb{R}_0^+$ be a weighted graph and let P_{new} be a path from a node s to a node t that has become a shortest path because of an increase of the weight of an edge (x, y). Then, for all nodes $u \in P_{\text{new}}$:*

$$d_{\text{new}}(s, u) < d_{\text{new}}(s, x) + w_{\text{new}}(x, y) + d_{\text{new}}(y, u)$$

Proof. The new shortest path P_{new} does not contain the edge (x, y), and the sub-path of P_{new} from s to u is also a shortest path that does not contain the edge (x, y). The right hand side of the inequality is the length of some path from s to u containing (x, y). Since shortest paths are assumed to be unique, the lemma follows immediately. □

Algorithm 2 (**Increase-Edge-Weight**(x, y)) combines the result of Corollary 4.2 and Lemma 4.3. Dijkstra's algorithm is only run for all nodes in $S_{\text{old}}(x, y)$ and nodes are only inserted into the queue Q, if the condition of Lemma 4.3 is fulfilled. The rest of the nodes that do not satisfy Lemma 4.3 are never inserted in the queue Q.

Theorem 4.4 ***Increase-Edge-Weight**(x, y) (Algorithm 2) enlarges all target containers after an increase of the weight $w(x, y)$ as necessary.*

Increase-Edge-Weight(x, y)
Input: graph $G = (V, E)$, weights $w : E \to \mathbb{R}_0^+$,
consistent target containers for old edge weight $w_{\text{old}}(x, y)$,
increased edge weight $w_{\text{new}}(x, y)$
Output: consistent target containers for increased edge weight $w_{\text{new}}(x, y)$

```
0    for all s ∈ S_old(x,y) do
1      set d(u) := ∞ for all nodes u ∈ V
2      create empty priority queue Q
3      insert source s in Q and set d(s) := 0
4      while priority queue is not empty
5        get node u with smallest tentative distance in Q
5a       if u ≠ s enlarge T(A[u]) to contain u
6        for all neighbor nodes v of u
7          set new-dist := dist(u) + w(u,v)
7a         if new-dist < d(s,x) + w_new(x,y) + d(y,v)
8            if new-dist < dist(v)
9              if d(v) = ∞
10               insert neighbor node v in Q with priority new-dist
11             else
12               set priority of neighbor node v in Q to new-dist
13             set dist(v) := new-dist
14             if u = s
15               set A[v] := (s,v)
16             else
17               set A[v] := A[u]
```

Algorithm 2. Increase-Edge-Weight(x, y). Dijkstra's algorithm truncated to enlarge target containers after an increase of the weight $w(x, y)$. A neighbor v is only visited, if the path from s to v does not contain the edge (x, y).

Proof. Consider the shortest path P from s to t that is found by an unmodified Dijkstra's algorithm. If for all nodes $v \in P$ the condition in line `7a` is fulfilled, the path P is found by **Increase-Edge-Weight**(x, y), because the pruning does not change the order in which the edges are processed. □

Let

$$\texttt{Pot-Aff}(s) := \{v \in V : d_{\text{new}}(s, v) < d_{\text{new}}(s, x) + w_{\text{new}}(x, y) + d_{\text{new}}(y, v)\}$$

denote the set of *potentially affected nodes* for $s \in V$ after an increase of the edge weight $w(x, y)$. For each node $s \in S_{\text{old}}(x, y)$, **Increase-Edge-Weight**(x, y) runs Dijkstra's algorithm restricted to the graph induced by $\texttt{Pot-Aff}(s)$. The running time of **Increase-Edge-Weight**(x, y) is therefore linear in

$$\sum_{s \in S_{\text{old}}(x,y)} |\texttt{Pot-Aff}(s)|$$

If $|\texttt{Pot-Aff}(s)|$ is bounded for all $s \in S_{\text{old}}(x, y)$ by some p, then the running

time is $O(k \cdot p \log p)$ where $k := |\{s \in S_{\text{old}}(x, y)\}|$.

The source containers are the counterpart to target containers, if all edges in the graph are reversed. So, **Increase-Edge-Weight**(x, y) applied to a graph with reversed edges enlarges the source containers as necessary.

The algorithm **Increase-Edge-Weight**(x, y) uses distance values $d(u, x)$ and $d(y, u)$ for different $u \in E$, which have to be computed beforehand. So, the overall method for an increase of $w(x, y)$ is as follows:

(i) Run Dijkstra's algorithm for the source y to compute $d(y, u)$ for all $u \in V$.

(ii) Run Dijkstra's algorithm with reversed edges for the source x to compute $d(u, x)$ for all $u \in V$.

(iii) Run **Increase-Edge-Weight**(x, y) to enlarge target containers.

(iv) Run **Increase-Edge-Weight**(x, y) with reversed edges to enlarge source containers.

4.2 Decreasing an edge weight

Similar to the case of an increase, we can prove a lemma about start and end nodes of new shortest paths for the case of a decrease. This time however, the updated source and target containers must be used.

Lemma 4.5 *Let $G = (V, E)$, $w : E \to \mathbb{R}_0^+$ be a weighted graph. Assume that the decrease of the weight of an edge $(x, y) \in E$ creates a new shortest s-t-path P_{new} in G. Then, after the weight change, (x, y) is the last edge of a shortest s-y-path and the first edge of a shortest x-t-path.*

Proof. Obviously, the edge (x, y) must be part of this path P_{new}. Let P_{sy} denote the sub-path of P_{new} from s to v. As a sub-path of a shortest path P_{sy} is also a shortest path. In particular, P_{sy} is a shortest path that ends with the edge (x, y). For symmetric reasons, the first edge of a shortest x-t-path is (x, y). □

Corollary 4.6 *Let $G = (V, E)$, $w : E \to \mathbb{R}_0^+$ be a weighted graph. Assume that the decrease of the weight of an edge $(x, y) \in E$ creates a new shortest s-t-path P_{new} in G. Then*

$$s \in S_{\text{new}}(x, y) \qquad \text{and} \qquad t \in T_{\text{new}}(x, y).$$

In order to run a (truncated) Dijkstra for all nodes in $T_{\text{new}}(x, y)$, it is necessary to compute $T_{\text{new}}(x, y)$, i.e. to enlarge it if necessary. This can be achieved by Algorithm 3 **Compute-T_{new}**(x, y). The container is enlarged similar to its creation in **Create-Containers**. If a new shortest path is found to a node t, the container $T(x, y)$ is enlarged in line 5a. This variant of Dijkstra's algorithm is truncated to the part of the graph, where distance labels change.

Theorem 4.7 *Let $G = (V, E)$, $w : E \to \mathbb{R}_0^+$ be a weighted graph and $S_{\text{old}}(x, y)$*

Compute-$T_{\text{new}}(x, y)$
Input: graph $G = (V, E)$, weights $w : E \to \mathbb{R}_0^+$,
consistent target containers for old edge weight $w_{\text{old}}(x, y)$,
decreased edge weight $w_{\text{new}}(x, y)$
Output: consistent target container $T_{\text{new}}(x, y)$

```
0     set s := x
1     set d(u) := ∞ for all nodes u ∈ V
2     create empty priority queue Q
3     insert source s in Q and set d(s) := 0
4     while priority queue is not empty
5       get node u with smallest tentative distance in Q
5a      if u ≠ s and T(A[u]) = (x, y) enlarge T((x, y)) to contain u
6       for all neighbor nodes v of u
7         set new-dist := dist(u) + w(u, v)
7a        if new-dist < w_old(x, y) + d_old(y, u)
8           if dist(v) > new-dist
9             if d(v) = ∞
10              insert neighbor node v in Q with priority new-dist
11            else
12              set priority of neighbor node v in Q to new-dist
13            set dist(v) := new-dist
14            if u = s
15              set A[v] := (s, v)
16            else
17              set A[v] := A[u]
```

Algorithm 3. **Compute-$T_{\text{new}}(x, y)$**. Dijkstra's algorithm truncated to enlarge target container $T(x, y)$ after an decrease of the weight $w(x, y)$. A neighbor v is only visited, if the distance from x to v is shorter than the old distance $d_{\text{old}}(x, v)$.

a consistent source container. ***Compute-$T_{\text{new}}(x, y)$*** *(Algorithm 3) enlarges the source container for a decrease of the weight $w(x, y)$ as necessary.*

Proof. It is obvious that without line 7a the algorithm **Compute-$T_{\text{new}}(x, y)$** works as expected. If a node v is excluded in line 7a, we distinguish between two cases. If `new-dist` $= w_{\text{new}}(x, y) + d_{\text{old}}(y, v)$, the distance of v has not changed. Furthermore, the distance has not changed for all nodes a where the shortest x-a-path contains v. Ignoring nodes $v \in V$ with `new-dist` $= w_{\text{new}}(x, y) + d_{\text{old}}(y, v)$ therefore does not change the result of the algorithm. If `new-dist` $> w_{\text{new}}(x, y) + d_{\text{old}}(y, v)$, there exists a shorter path from x to v that does not contain (u, v). The node v can therefore be ignored in this case, too. □

Our final goal is to run a truncated version of Dijkstra's algorithm for all nodes in $S_{\text{new}}(x, y)$ to adjust all start containers after an edge weight decrease. The truncation is realized similarly to Lemma 4.3, but this time the old weight

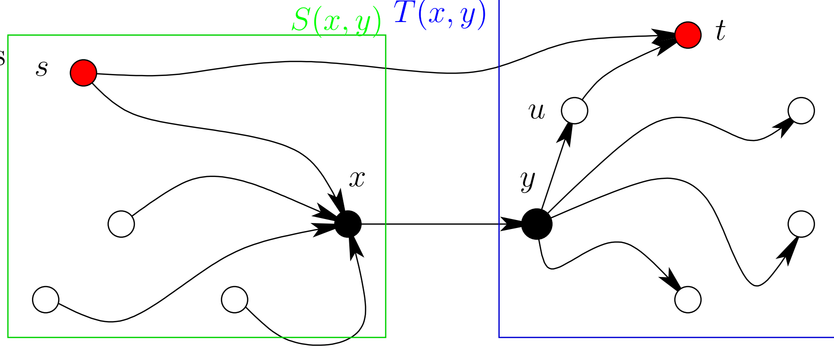


Fig. 3. When the weight of the edge (x, y) is decreased, for all nodes u on a new shortest path from s to t, $d_{\text{new}}(s, u) < d_{\text{new}}(s, x) + w_{\text{old}}(x, y) + d_{\text{new}}(y, u)$.

of the edge (x, y) is used in the comparison.

Lemma 4.8 *Let $G = (V, E)$, $w : E \to \mathbb{R}_0^+$ be a weighted graph and let P_{new} be a path from a node s to a node t that has become a shortest path because of a decrease of the weight of an edge (x, y). Then, for all nodes $u \in P_{\text{new}}$:*

$$d_{\text{new}}(s, u) < d_{\text{new}}(s, x) + w_{\text{old}}(x, y) + d_{\text{new}}(y, u)$$

Proof. Since $w_{\text{new}}(x, y) < w_{\text{old}}(x, y)$, the new distance $d_{\text{new}}(s, t)$ must be shorter than the old distance $d_{\text{old}}(s, t)$. The new shortest path P_{new} does contain the edge (x, y) in contrast to the old shortest path from s to t. Therefore

$$\begin{aligned} d_{\text{new}}(s, t) &= d_{\text{new}}(s, x) + w_{\text{new}}(x, y) + d_{\text{new}}(y, t) \\ &< d_{\text{new}}(s, x) + w_{\text{old}}(x, y) + d_{\text{new}}(y, t) \end{aligned}$$

Consider now some node $u \in P_{\text{new}}$ (illustrated in Figure 3). Let $P_{s,u}$ denote the sub-path of P_{new} from s to u. If $P_{s,u}$ does not contain (x, y), i.e. if the edge (x, y) appears in P_{new} after u,

$$d_{\text{new}}(s, u) < d_{\text{new}}(s, x) < d_{\text{new}}(s, x) + w_{\text{old}}(x, y) + d_{\text{new}}(y, t)$$

since $w_{\text{old}}(x, y) > 0$. If $P_{s,u}$ contains (x, y), then

$$d_{\text{new}}(s, u) = d_{\text{new}}(s, x) + w_{\text{new}}(x, y) + d_{\text{new}}(y, u)$$

otherwise a shorter path from s to u would exist which contradicts the fact that P_{new} is a shortest path. Since $w_{\text{old}}(x, y) > w_{\text{new}}(x, y)$ the lemma follows. □

Using Lemma 4.6 and 4.8, the correctness of **Decrease-Edge-Weight**(x, y) (Algorithm 4) follows immediately as stated in the following Theorem.

Theorem 4.9 ***Decrease-Edge-Weight**(x, y) (Algorithm 4) enlarges all target containers after a decrease of the weight $w(x, y)$.* □

Let

$$\texttt{Pot-Aff}'(s) := \{v \in V : d_{\text{new}}(s, v) < d_{\text{new}}(s, x) + w_{\text{old}}(x, y) + d_{\text{new}}(y, v)\}$$

denote the set of *potentially affected nodes* for $s \in V$ after a decrease of the edge weight $w(x, y)$. For each node $s \in S_{\text{new}}(x, y)$, **Decrease-Edge-**

Decrease-Edge-Weight(x, y)
Input: graph $G = (V, E)$, weights $w : E \to \mathbb{R}_0^+$,
  consistent target containers for old edge weight $w_{\text{old}}(x, y)$,
  decreased edge weight $w_{\text{new}}(x, y)$,
consistent source container $S_{\text{new}}(x, y)$
Output: consistent target containers for decreased edge weight $w_{\text{new}}(x, y)$
0 for all $s \in S_{\text{new}}(x, y)$ do
1   set $d(u) := \infty$ for all nodes $u \in V$
2   create empty priority queue Q
3   insert source s in Q and set $d(s) := 0$
4   while priority queue is not empty
5   get node u with smallest tentative distance in Q
5a   if $u \neq s$ enlarge $T(A[u])$ to contain u
6   for all neighbor nodes v of u
7   set `new-dist` := `dist`$(u) + w(u, v)$
7a   if `new-dist` $< d(s, x) + w_{\text{old}}(x, y) + d(y, v)$
8   if `dist`$(v) >$ `new-dist`
9   if $d(v) = \infty$
10   insert neighbor node v in Q with priority `new-dist`
11   else
12   set priority of neighbor node v in Q to `new-dist`
13   set `dist`(v) := `new-dist`
14   if $u = s$
15   set $A[v] := (s, v)$
16   else
17   set $A[v] := A[u]$

Algorithm 4. Decrease-Edge-Weight(x, y). Dijkstra's algorithm truncated to enlarge target containers after an decrease of the weight $w(x, y)$. A neighbor v is only visited, if the path from s to v is shorter than the shortest path from s to v that uses edge (x, y) with the old weight $w_{\text{old}}(x, y)$.

Weight(x, y) runs Dijkstra's algorithm restricted to the graph induced by `Pot-Aff`$'(s)$. The running time of **Decrease-Edge-Weight**(x, y) is therefore linear in

$$\sum_{s \in S_{\text{new}}(x,y)} |\texttt{Pot-Aff}'(s)|$$

If $|\texttt{Pot-Aff}'(s)|$ is bounded for all $s \in S_{\text{new}}(x, y)$ by some p, then the running time is $O(k \cdot p \log p)$ where $k := |\{s \in S_{\text{new}}(x, y)\}|$.

In summary, the combination of **Compute-**$T_{\text{new}}(x, y)$ for reversed edges and **Decrease-Edge-Weight**(x, y) enlarges the target containers after an decrease of an edge weight $w(x, y)$. As in the previous case, source containers can be treated similarly with all edges reversed. The complete algorithm to update source and target containers after an edge decrease is therefore:

(i) Run Dijkstra's algorithm for the source y to compute $d(y, u)$ for all $u \in V$.

(ii) Run Dijkstra's algorithm with reversed edges for the source x to compute $d(u, x)$ for all $u \in V$.

(iii) Run **Compute-**$T_{\text{new}}(x, y)$ to enlarge the target containers $S(x, y)$.

(iv) Run **Compute-**$T_{\text{new}}(x, y)$ for y with reversed edges to enlarge the source container $S(x, y)$.

(v) Run **Decrease-Edge-Weight**(x, y) to enlarge start containers.

(vi) Run **Decrease-Edge-Weight**(x, y) with reversed edges to enlarge source containers.

5 Experiments

5.1 Experimental Setup

We performed an experimental study to evaluate the performance and quality of our algorithms. More precisely the following two questions were examined:

- How much time is needed to update the containers (on average)?
- How much do the containers differ from containers computed from scratch? (Remember that we do not shrink containers in our updates.)

The quality of a set of containers was evaluated according to the following criterion.

Definition 5.1 Let C denote the set of containers we want to examine and C_{ref} denote the reference set of containers that has been computed from scratch. For both sets, we count the number of nodes inside all containers $\sum_{e \in E} |\{t \in C(e)\}|$. Both sums are bounded by $n \cdot m$. We therefore define the *quality* of C as:

$$\frac{nm - \sum_{e \in E} |\{t \in C(e)\}|}{nm - \sum_{e \in E} |\{t \in C_{\text{ref}}(e)\}|}$$

This fraction equals 1, if the number of wrong nodes inside containers is the same for C and C_{ref}, but is biased by the number of correct nodes. If all containers in C contain the entire graph, the quality of C is 0.

The test graphs in our computational study are railway networks of different European countries. The nodes of such a graph are the stations and an edge between two stations exists iff there is an non-stop connection. The edges are weighted by the average travel time. The sizes of these networks are given in Table 1.

For each graph, we increase the weight of 100 random edges to a large value (i.e. the sum of all weights in the graph). This is similar to removing the edge from the graph. After every weight change, the containers are updated according to section 4.1. A second set of containers is determined from scratch to compute the quality and compare the computation time.

For the evaluation of decreasing edge weights, we start with the graph where 100 random edges have been set to a large weight. The weights are then decreased to their original values. Again, the updated containers are compared to newly computed containers.

Apart from the algorithms in section 4, we evaluated updates, if source containers are not maintained. In this case, **Increase-Edge-Weight**(x, y) and **Decrease-Edge-Weight**(x, y) cannot iterate over all nodes in $S_{\text{old}}(x, y)$ and $S_{\text{new}}(x, y)$, respectively. The nodes, where containers have to be updated, must be determined beforehand. According to Lemma 4.1 and 4.5, for such a node $s \in V$, the last edge on a shortest s-y-path is (x, y). These nodes can be determined by a run of a modified Dijkstra starting at y with reversed edges. Note that in this case only half of the containers need maintenance.

Both variants, with source containers and without source containers, find those nodes for which the containers of incident edges must be updated. For both variants, we studied three methods to update the container of an edge:

- Enlarge the container as described in Section 4. A variant of Dijkstra's algorithm pruned according to Lemma 4.3 and 4.8 enlarges the containers.
- Compute the container from scratch. The result is slightly different from recomputing all containers from scratch, because some may shrink but are not updated. However, the distances of all nodes to x and from y to all nodes are not needed in this case and their computation can be omitted.
- Enlarge the container to infinity (without any further computation). If the entire graph is inside the container, it is certainly consistent. However, the quality of the containers is going down rapidly. Again, the distances of all nodes to x and from y to all nodes are not needed in this case and their computation can be omitted.

All six variants have been implemented in C++ based on the graph structure provided by LEDA 4.4. The programs were compiled with GCC 3.2 and run on a single Intel Xeon with 2.4 GHz performing Linux 2.4.

5.2 Computational Results

Figures 4 and 5 depict the changes of the quality for updated containers. In both cases – with and without source containers – the outcome is very similar. Furthermore, the case of an edge weight increase resembles the case of an edge weight decrease.

- If containers are only enlarged, their quality decreases most of the time as expected. It is interesting to note that the larger the graph, the larger its quality remains. Single "bad" containers are clearly less important, if the graph contains more edges. For large graphs, the quality stays close to 1 even after 100 updates.
- If the containers are simply set to infinity, the situation is dramatically different though. After a few updates, the containers settle in a state where

	source container		used			not used		
	container update		set to infinity	enlarge	from scratch	set to infinity	enlarge	from scratch
	n	m						
nl	409	1215	1331	2.17	2.45	269	2.52	5.02
au	1660	4327	8093	1.99	2.54	1551	2.07	2.13
ch	2279	6015	10211	2.18	2.56	2235	2.77	2.48
i	2399	8008	9552	2.85	2.77	2095	2.80	2.64
fr	4598	14937	17691	2.09	2.87	4382	2.55	4.25
de	6884	18601	33160	1.72	2.04	6568	2.53	2.56

Table 1
Average speed-up for updating the containers after increasing an edge weight

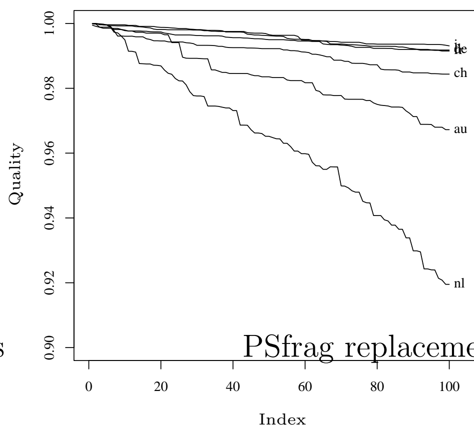
almost all nodes are inside all containers. Such a state is clearly not desirable, because no nodes are pruned by Dijkstra's algorithm for queries.

The situation is slightly better in Figure 5, where source containers are not used, because they may already be very bad from previous updates. Still the resulting containers are not acceptable after a few updates.

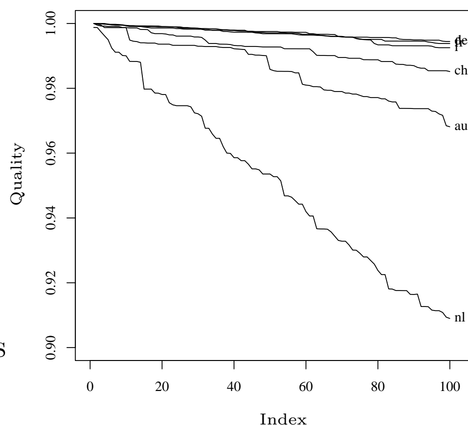
- The case where containers of adjacent edges are recomputed from scratch is missing in the figures, since the resulting containers coincide most of the time with the newly computed containers. In other words, the quality equals 1 after almost every update. In practice, such updates can therefore be considered as good as using freshly determined containers.

The analysis of the time measurements are shown in Table 1 for weight increases and Table 2 for weight decreases. The first two columns list the number of nodes and the number of edges in the respective graphs. The other six columns refer to the six cases that have been examined in our study. The three types of updates (enlarge the containers to infinity, enlarge the containers according to Lemma 4.3 and 4.8, and recompute the containers from scratch) were tested with maintenance of source containers and without it (using a backward Dijkstra instead). Although the time improvements are huge, if the containers are enlarged to infinity, these values are more or less meaningless, because of the inacceptable quality. Discussing them does not really make sense and they are mainly shown for sake of completeness.

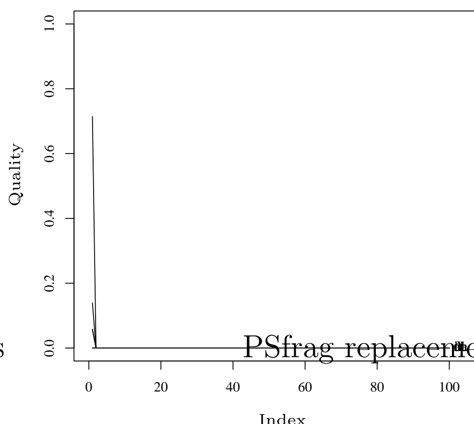
An interesting observation is the fact that the speed-up factor does not seem to be correlated with the size of the graph. Furthermore the similarity



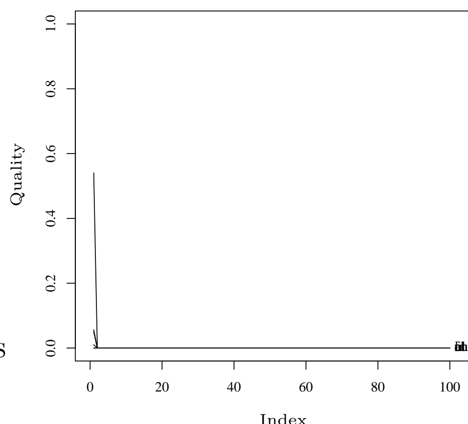
(a) enlarged containers after increasing weights



(b) enlarged containers after decreasing weights



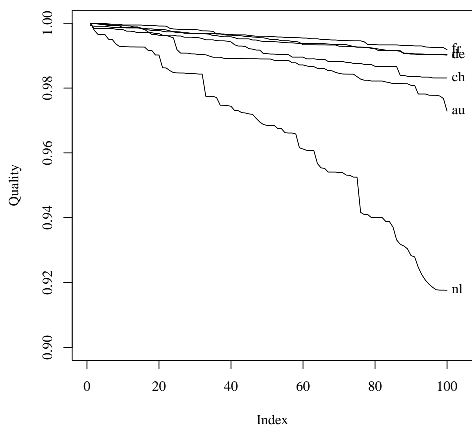
(c) infinite containers after increasing weights



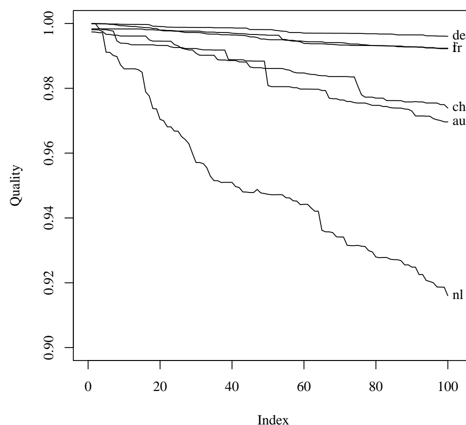
(d) infinite containers after decreasing weights

Fig. 4. The quality of containers after 100 changed edge weights, if source containers are available and maintained. In the upper diagrams, the containers are updated by a pruned Dijkstra for all nodes in the source container of the edge with changed weight (see Algorithms 2 and 4). In the lower diagrams, containers are enlarged to infinity for all edges incident to nodes inside $S_{\text{old}}(x, y)$.

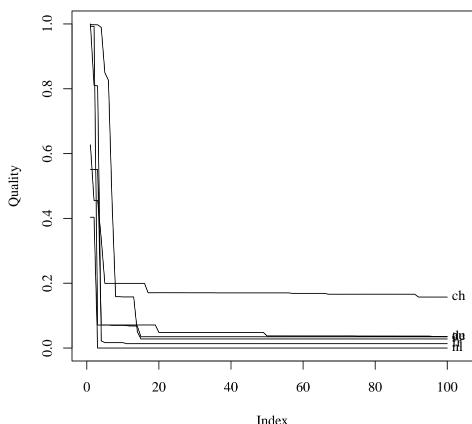
of the algorithms for increasing and decreasing edge weights probably explains the similar behavior in terms of timings. The speed-up values with and without source containers are quite similar, but note that the absolute time values with source containers are about twice as large. Maintaining source



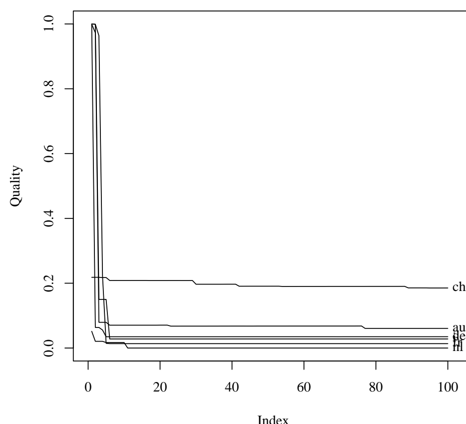
(a) enlarged containers after increasing weights



(b) enlarged containers after decreasing weights



(c) infinite containers after increasing weights



(d) infinite containers after decreasing weights

Fig. 5. The quality of containers after 100 changed edge weights. In the upper diagrams, the containers are enlarged by a pruned Dijkstra for all nodes s with (x, y) as the last edge of a shortest s-y-path. In the lower diagrams, the containers are enlarged to infinity for all edges incident to nodes s with (x, y) as the last edge of a shortest s-y-path.

containers can therefore only justified, if they are used otherwise (e.g. for a bi-directional search). The most interesting observation however is the fact that using a pruned Dijkstra (column "enlarge") is often slower than Dijkstra without pruning (column "from scratch"). Obviously the additional check and

	source container		used			not used		
	container update		set to infinity	enlarge	from scratch	set to infinity	enlarge	from scratch
	n	m						
nl	409	1215	173	2.00	2.70	228	3.43	3.02
au	1660	4327	700	1.90	2.38	1429	2.10	2.10
ch	2279	6015	953	2.24	2.58	2300	2.85	3.09
i	2399	8008	965	2.75	2.68	2097	2.65	2.94
fr	4598	14937	1932	2.21	2.66	4291	3.13	3.36
de	6884	18601	2974	1.71	2.21	6583	2.31	2.76

Table 2
Average speed-up for updating the containers after decreasing an edge weight

computing the distances to x and from y for all nodes outweigh the gain of the pruning.

6 Conclusion and Outlook

We have seen that it is possible to speed up the maintenance of geometric containers by a factor of about 2-3 while preserving optimality in almost all cases. Enlarging containers to infinity leads to a cascading effect that destroys the benefit of geometric containers. If containers are only enlarged, the presented pruning of Dijkstra's algorithm does not justify the loss of quality.

It would be interesting to find other simplifications that guarantee consistent containers, but realize a good compromise between optimality and running time. Furthermore, our results suggest that it should be possible to get a speed-up factor of about 2 with an (provable) optimal update strategy. Finally, it might be possible to combine edge weight increases and edge weight decreases in a single algorithm.

References

[1] Ausiello, G., G. F. Italiano, A. Marchetti-Spaccamela and U. Nanni, *Incremental algorithms for minimal length paths*, Journal of Algorithms **12** (1991), pp. 615–638.

[2] Barrett, C., K. Bisset, R. Jacob, G. Konjevod and M. Marathe, *Classical and*

contemporary shortest path problems in road networks: Implementation and experimental analysis of the transims router, in: R. Möhring and R. Raman, editors, *ESA 2002*, LNCS **2461** (2002), pp. 126–138.

[3] Barrett, C., R. Jacob and M. Marathe, *Formal-language-constrained path problems*, SIAM Journal on Computing **30** (2000), pp. 809–837.
URL http://epubs.siam.org/sam-bin/dbq/article/33771

[4] Demetrescu, C. and G. F. Italiano, *A new approach to dynamic all pairs shortest paths*, in: *Proceedings of the thirty-fifth ACM Symposium on Theory of Computing (STOC 2003)* (2003), pp. 159 – 166.
URL http://doi.acm.org/10.1145/780542.780567

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

[6] Even, S. and H. Gazit, *Updating distances in dynamic graphs*, Methods of Operations Research **49** (1985), pp. 371–387.

[7] Frigioni, D., *Semidynamic algorithms for maintaining single-source shortest path trees*, Algorithmica **22** (1998), pp. 250–274.
URL http://www.springerlink.com/link.asp?id=23udd5uwx8lp

[8] Frigioni, D., A. Marchetti-Spaccamela and U. Nanni, *Fully dynamic output bounded single source shortest path problem*, in: *SODA*, 1996, pp. 212–221.

[9] King, V., *Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs*, in: *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS'99)*, 1999, pp. 81–91.

[10] Müller-Hannemann, M. and K. Weihe, *Pareto shortest paths is often feasible in practice*, in: G. Brodal, D. Frigioni and A. Marchetti-Spaccamela, editors, *WAE 2001*, LNCS **2461** (2001), pp. 185–197.

[11] Nachtigall, K., *Time depending shortest-path problems with applications to railway networks*, European Journal of Operational Research **83** (1995), pp. 154–166.

[12] Pohl, I., *Bi-directional search*, in: B. Meltzer and D. Michie, editors, *Sixth Annual Machine Intelligence Workshop*, Machine Intelligence **6** (1971), pp. 137–140.

[13] Preuss, T. and J.-H. Syrbe, *An integrated traffic information system*, in: *Proc. 6th Int. Conf. Appl. Computer Networking in Architecture, Construction, Design, Civil Eng., and Urban Planning (europIA '97)*, 1997.

[14] Ramalingam, G. and T. W. Reps, *An incremental algorithm for a generalization of the shortest-path problem*, Journal of Algorithms **21** (1996), pp. 267–305.

[15] Ramalingam, G. and T. W. Reps, *On the computational complexity of dynamic graph problems*, Theoretical Computer Science **158** (1996), pp. 233–277.

[16] Rohnert, H., *A dynamization of the all pairs least cost path problem*, in: *Proc. Symp. Theoretical Aspects of Computer Science (STACS'85)*, LNCS **182** (1985), pp. 279–286.

[17] Schulz, F., D. Wagner and K. Weihe, *Dijkstra's algorithm on-line: An empirical case study from public railroad transport*, Journal of Experimental Algorithmics **5** (2000).
URL http://www.jea.acm.org/2000/SchulzDijkstra/

[18] Shekhar, S., A. Fetterer and B. Goyal, *Materialization trade-offs in hierarchical shortest path algorithms*, in: *Symposium on Large Spatial Databases*, 1997, pp. 94–111.

[19] Siklóssy, L. and E. Tulp, *Trains, an active time-table searcher*, in: *Proc. 8th European Conf. Artificial Intelligence*, 1988, pp. 170–175.

[20] Wagner, D. and T. Willhalm, *Geometric speed-up techniques for finding shortest paths in large sparse graphs*, in: *Proc. 11th European Symposium on Algorithms (ESA 2003)*, LNCS (2003), to appear.

[21] Zahn, F. B. and C. E. Noon, *A comparison between label-setting and label-correcting algorithms for computing one-to-one shortest paths*, Journal of Geographic Information and Decision Analysis **4** (2000).
URL http://www.geodec.org/

[22] Zaroliagis, C., *Implementations and experimental studies of dynamic graph algorithms*, in: R. Fleischer, B. Moret and E. M. Schmidt, editors, *Experimental Algorithmics*, LNCS **2547** (2002), pp. 229–278.