Geometric Shortest Path Containers[*]

Dorothea Wagner[†] Thomas Willhalm[†] Christos Zaroliagis[‡]

February 26, 2004

Abstract

In this paper, we consider Dijkstra's algorithm for the single source single target shortest path problem in large sparse graphs. The goal is to reduce the response time for on-line queries by using precomputed information. Due to the size of the graph, preprocessing space requirements can be only linear in the number of nodes. We assume that a layout of the graph is given. In the preprocessing, we determine from this layout a geometric object for each edge containing all nodes that can be reached by a shortest path starting with that edge. Based on these geometric objects, the search space for on-line computation can be reduced significantly. Shortest path queries can then be answered by Dijkstra's algorithm restricted to edges where the corresponding geometric object contains the target.

We present an extensive experimental study comparing the impact of different types of objects. The test data we use are real-world traffic networks, the typical field of application for this scenario. Furthermore, 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 geometric containers have to be updated. We evaluate the quality and the time for different update strategies that guarantee correct shortest paths. Finally, we present a software framework in `C++` to realize the implementations of all of our variants of Dijkstra's algorithm. A basic implementation of the algorithm is refined for each modification and – even more importantly – these modifications can be combined in any possible way without loss of efficiency.

Keywords: Graph algorithms

1 Introduction

We consider a typical application in traffic 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 for cars, bikes and hikers [47, 3] or scheduled vehicles like trains and buses [25, 28]. Similar, query intensive applications include spatial databases [36] and web searching [4]. Users of such systems continuously enter their requests for finding their "best connections" and the main goal is to reduce the (average) response time for answering a query.

The algorithmic core problem that underlies the above applications 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 only linear in the number of nodes space requirements are acceptable. The application of shortest path computations in travel networks is widely covered in the literature; see e.g., [3, 25, 28, 47]. One of the features of travel planning is the fact that the network does not change

[*]This work was partially supported by the Human Potential Programme of the European Union under contract no. HPRN-CT-1999-00104 (AMORE), by the DFG under grant WA 654/12-1, by the European Commission - Fet Open project COSIN - COevolution and Self-organisation In dynamical Networks - IST-2001-33555, and by the EU within the 6th Framework Programme under contract 001907 (integated project DELIS). Part of the work was done while the second author was visiting the Computer Technology Institute in Patras, and while the third author was visiting the University of Karlsruhe.

[†]Institut für Logik, Komplexität und Deduktionssysteme Universität Karlsruhe, Postfach 6980, 76128 Karlsruhe, Germany. Emails: {`dwagner,willhalm`}`@ira.uka.de`.

[‡]IEEE member. Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece; and Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece. Email: `zaro@ceid.upatras.gr`.

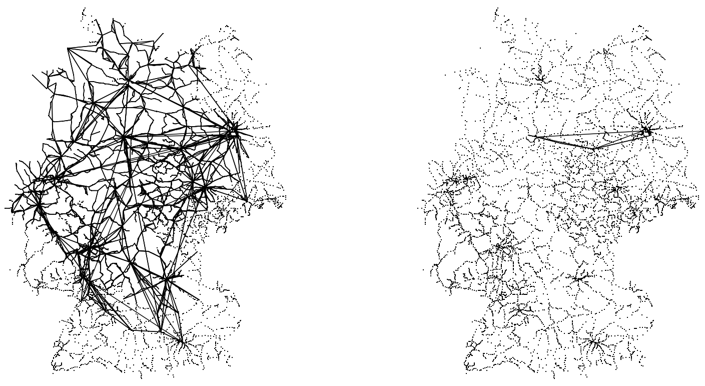


Figure 1: The search space for a query from Hannover to Berlin for Dijkstra's algorithm (left) and Dijkstra's algorithm with pruning using bounding boxes (right).

for a certain period of time while there are many queries for shortest paths. This justifies a heavy preprocessing of the network to speed up the queries (see e.g., [32, 33]). Although pre-computing and storing the shortest paths for all pairs of nodes would give us "constant-time" shortest-path queries, the quadratic space requirement for traffic networks with more than 10^5 nodes makes it prohibitive. The most commonly used approach for answering shortest path queries concerns variants of Dijkstra's algorithm [47], targeting at reducing its *search-space* (number of nodes visited by the algorithm).

In this paper, we explore the possibility to reduce the search space of Dijkstra's algorithm by using precomputed information that can be stored in $O(n + m)$ space. Our main contribution is that we use the given layout of the graph to extract geometric information to answer the on-line queries fast. In fact, this paper shows that storing partial results reduces the number of nodes visited by Dijkstra's algorithm to only 10%. (Figure 1 gives an illustrative example for the German railway network.) We use a very fundamental observation on shortest paths. In general, an edge that is not the first edge on a shortest path to the target can be safely ignored in any shortest path computation to this target. More precisely, we apply the following concept. In the preprocessing, for each edge e a set of nodes $S(e)$ is computed which are the nodes that can be reached by a shortest path starting with e. While running Dijkstra's algorithm, those edges e for which the target is not in $S(e)$ are ignored.

As storing all sets $S(e)$ would need $O(mn)$ space, we relax the condition by storing a geometric object for each edge that contains *at least* the nodes in $S(e)$. The shortest path queries are then answered by Dijkstra's algorithm restricted to those edges for which the target node is inside their associated geometric object. Note that this method does in fact still lead to a correct result, but may increase the number of visited nodes to more than the strict minimum (i.e., the number of nodes in the shortest path). In order to generate the geometric objects, a layout $L : V \to \mathbb{R}^2$ is used. For the application of travel information systems, such a layout is given by the geographic locations of the nodes. It is however not required that the edge lengths are derived from the layout. In fact, for some of our experimental data this is not even the case.

We would like to mention that a particular type of geometric objects, the angular sectors, has been introduced in [32] for the special case of a time table information system. Our results, however, are more general in two respects: (a) we examine the impact of various different geometric objects; and (b) we consider Dijkstra's algorithm for general embedded graphs. We present an extensive experimental study comparing the impact of these objects using real-world test data from traffic networks, a typical field of application for the considered scenario. It turns out that a significant improvement can be achieved by using other geometric objects than angular sectors. Actually, in some cases the speed-up is even a factor of about two.

The second contribution of this paper concerns the dynamic version of the above mentioned scenario; namely, 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 [29, 30] and extended in [14, 15] – 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., [11, 31, 7, 2, 22], and [48] for a recent overview) are not applicable to maintain geometric containers, because of their inherent quadratic space requirements.

The last contribution of this paper concerns methodological issues regarding our implementations, which have been carried out in `C++`. Implementing and supporting that many variations of Dijkstra's algorithm in `C++` is a tedious task if it is not planned carefully. We employ several techniques to maintain a common code base that is at the same time small, flexible and efficient. We use a blend of the design pattern *template method* [16], parameterized inheritance [5, 18], and template meta-programming [1, 9].

Adding functionality to graph algorithms can be achieved by the design pattern *template method* [16] or an extension of the design pattern *visitor*, the approach of the BOOST graph library [38]. Our work deviates from the latter and is closer to the former, which it actually enhances to grasp parts of *aspect-oriented programming*. Aspect-oriented programming tries to provide a modular way to overcome the single dimension of functional decomposition by the design pattern *template method* (see e.g., [40]). More precisely, it is necessary to change the inheritance hierarchy to create arbitrary combinations of aspects. To support aspect oriented programming in `C++`, an extension of the `C++` language is proposed in [40]. However, such an extension can be avoided through the use of parameterized inheritance [18] (also known as mix-in classes [5]) and template meta-programming [1, 9], which provides the base to a solution with standard `C++` compilers.

The rest of the paper is organized as follows. The next section contains – after some definitions – a formal description of our shortest path problem. Section 3 presents more precisely how and why the pruning of edges and its preprocessing works, before we describe the geometric objects and other aspects of our experiments and present the statistics and computational results. Section 4 contains algorithms to update geometric containers after weight changes. After proving some useful properties, different update strategies are presented with their corresponding experiments and results. The software engineering framework for our algorithms is presented in Section 5. We discuss parameterized inheritance as a tool to interweave new aspects into an existing implementation of an algorithm including passing lists of arguments to adapted algorithms in a type-safe manner and automatically resolving dependencies between different variations of our algorithm. We conclude in Section 6. Preliminary portions of this work appeared in [43, 44].

2 Definitions and Problem Description

2.1 Graphs

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 $1 \leq i < k$. A path with $u_1 = u_k$ is called a *cycle*. A graph (without multiple edges) can have up to n^2 edges. We call a graph *sparse*, if $m = O(n)$, and we call a graph *large*, if one can only afford a memory consumption in $O(n)$. In particular, for large sparse graphs $O(n^2)$ space is not affordable.

We assume that we are given a *layout* $L : V \to \mathbb{R}^2$ of the graph in the Euclidean plane. For ease of notation we will identify a node $v \in V$ with its location $L(v) \in \mathbb{R}^2$ in the plane, and thus we shall use the terms "node" and "point" interchangeably. Throughout the paper, we assume that the layout is fixed.

2.2 Shortest Path Problem

Let $G = (V, E)$ be a directed graph whose edges are *weighted* by a function $w : E \to \mathbb{R}$. 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. The *(single source single target) shortest path problem* consists in finding a path of minimum length from a given source $s \in V$ to a given target $t \in V$. Note that the problem is only well defined for all pairs, if G does not contain negative cycles (cycles with negative length). In the presence of negative weights but not negative cycles, it is possible, using Johnson's algorithm [20], to convert in $O(nm + n^2 \log n)$ time the original edge weights $w : E \to \mathbb{R}$ to non-negative edge weights $w' : E \to \mathbb{R}_0^+$ that result in the same shortest paths. Hence, we can safely assume in the rest of this paper that edge weights are non-negative. We also assume throughout the paper that for all pairs $(s, t) \in V \times V$, the shortest path from s to t is unique.[1]

The classical algorithm for computing shortest paths in a directed graph with non-negative edge weights is that of Dijkstra [8] (Algorithm 1 without lines 3a and 5a). In the comparison model, Dijkstra's algorithm implemented with Fibonacci heaps [13] is still the fastest known algorithm for the general case of arbitrary non-negative edge lengths, taking $O(m + n \log n)$ worst-case time. For special cases (e.g., undirected graphs, integral or uniformly distributed edge weights) better algorithms are known [17, 24, 26, 41].

3 Geometric Pruning

3.1 Shortest Paths Containers

In this section, we introduce the concept of containers which helps to reduce the search space of Dijkstra's algorithm. Containers are used to keep the nodes which are potentially useful for shortest path computations. This idea gives rise to DIJKSTRA'S ALGORITHM WITH PRUNING (Algorithm 1), which reduces the search space by examining at each iteration only a subset of the neighbors of a node (line 5a); the differences to DIJKSTRA'S ALGORITHM are shown in boldface.[2] The idea is illustrated in Fig. 2. The condition in line 5a is formalized by the notion of a consistent container.

Definition 1 Let $G = (V, E)$, $w : E \to \mathbb{R}$ be a weighted graph. We call a set of nodes $C \subseteq V$ a *container*. A container C associated with an edge (u, v) is called *consistent*, if for all shortest paths from u to t that start with the edge (u, v), the target t is in C.

In other words, $C(u, v)$ is consistent, if $S(u, v) \subseteq C(u, v)$, where $S(u, v)$ represents the set of nodes x for which the shortest u-x path starts with the edge (u, v). Note that further nodes may be part of a consistent container. However, at least the nodes that can be reached by a shortest path starting with (u, v) must be in $C(u, v)$. We will refer to the additional nodes as *wrong nodes*, since they lead us the wrong way.

Theorem 2 *Let $G = (V, E)$, $w : E \to \mathbb{R}$ be a weighted graph and for each edge e let $C(e)$ be a consistent container. Then* DIJKSTRA'S ALGORITHM WITH PRUNING *finds a shortest path from s to t.*

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

[2]The initialization of `dist` in line 1 for each run of DIJKSTRA'S ALGORITHM can be omitted by introducing a global integer variable "time" and replacing the test $\texttt{dist}(v) = \infty$ by checking a time stamp for every node. See e.g., [32] for a detailed description.

```
1   for all nodes $u \in V$ set dist($u$) := $\infty$
2   initialize priority queue $Q$ with source $s$ and set dist($s$) := 0
3   while priority queue $Q$ is not empty
3a      **if $u = t$ return**
4       get node $u$ with smallest tentative distance dist($u$) in $Q$
5       for all neighbor nodes $v$ of $u$
5a          **if $t \in C(u, v)$**
7               set new-dist := dist($u$) + $w(u, v)$
8               if new-dist < dist($v$)
9                   if dist($v$) = $\infty$
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
```

Algorithm 1: Dijkstra's algorithm with pruning. Neighbors are only visited, if the edge (u, v) is in the consistent container $C(u, v)$. (Differences to Dijkstra's algorithm are printed in bold face.)

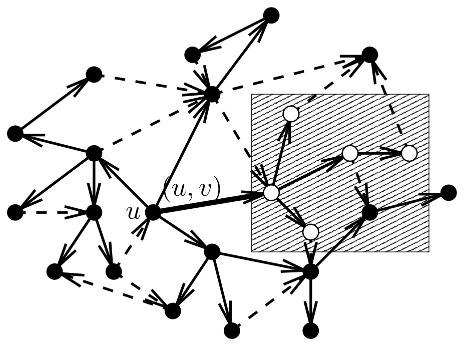


Figure 2: Dijkstra's algorithm is run for a node $u \in V$. Let the white nodes be those nodes that can be reached on a shortest path using the edge (u, v). A geometric object is constructed that contains these nodes. It may contain other nodes, but this does only affect the running time and not the correctness of Dijkstra's algorithm with pruning.

Proof: Consider the shortest path P from s to t that is found by Dijkstra's algorithm. If for all edges $e \in P$ the target node t is in $C(e)$, the path P is found by Dijkstra's algorithm with pruning, because the pruning does not change the order in which the edges are processed. A sub-path of a shortest path is again a shortest path, so for all $(u, v) \in P$, the sub-path of P from u to t is a shortest u-t-path. Then by definition of consistent container, $t \in C(u, v)$. □

This idea of pruning can be extended to bi-directional search [27]. A second set of containers is determined by reversing all edges and running the preprocessing a second time on this modified graph. We will refer to the geometric objects of this graph with reversed edges as *reverse containers.* A forward step in the bi-directional search checks the normal containers whereas a backward step uses reverse containers.

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

Definition 3 Let C denote a set of containers and for each edge $e \in E$ let $S(e) \subseteq V$ denote the set of nodes that can be reached by a shortest path starting with e. For both sets, we count the number of nodes inside all containers: $\sum_{e \in E} |\{t \in C(e)\}|$ and $\sum_{e \in E} |\{t \in S(e)\}|$. Both sums are bounded by $n \cdot m$. We therefore define the *quality* of C as:

$$\frac{n \cdot m - \sum_{e \in E} |\{t \in C(e)\}|}{n \cdot m - \sum_{e \in E} |\{t \in S(e)\}|}$$

This fraction is biased by the number of correct nodes. It equals 1, if the number of wrong nodes inside containers is zero, while it becomes 0, if all containers in C contain the entire graph.

3.2 Creating Consistent Containers

We now describe in detail how to compute $C(s, x)$ for all edges $(s, x) \in E$. The complete algorithm is shown as Algorithm 2. (The differences to Dijkstra's algorithm are printed in bold face.)

```
0   for all $s \in V$ do
1       for all nodes $u \in V$ set $\mathtt{dist}(u) := \infty$
2       initialize priority queue $Q$ with source $s$ and set $\mathtt{dist}(s) := 0$
3       while priority queue $Q$ is not empty
4           get node $u$ with smallest tentative distance $\mathtt{dist}(u)$ in $Q$
4a          if $u \neq s$ enlarge $C(A[u])$ to contain $u$
5           for all neighbor nodes $v$ of $u$
7                   set $\mathtt{new\text{-}dist} := \mathtt{dist}(u) + w(u, v)$
8                   if $\mathtt{new\text{-}dist} < \mathtt{dist}(v)$
9                       if $\mathtt{dist}(v) = \infty$
10                          insert neighbor node $v$ in $Q$ with priority $\mathtt{new\text{-}dist}$
11                      else
12                          set priority of neighbor node $v$ in $Q$ to $\mathtt{new\text{-}dist}$
13                      set $\mathtt{dist}(v) := \mathtt{new\text{-}dist}$
14                      if $u = s$
15                          set $A[v] := (s, v)$
16                      else
17                          set $A[v] := A[u]$
```

Algorithm 2: Create-Containers. Running a modification of Dijkstra's algorithm for all nodes $s \in V$ to create consistent containers. (Differences to Dijkstra's algorithm are printed in bold face.)

Recall that $S(s, x)$ is the set of all nodes t with the property that there is the (unique) shortest s-t-path that starts with the edge (s, x). To determine $S(s, x)$ 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. Enlarging $C(A[u])$ to contain u in line 4a therefore constructs consistent containers $C(s, x)$ for all neighbors x of s.

Since Dijkstra's algorithm runs in $O(n \log n)$ time for sparse graphs, the overall running time is $O(n^2 \log n)$ plus the time to construct the containers. The storage requirement is $O(n)$.

For some types of containers, it is not possible to be constructed *on-line* by enlarging them when a new node is inserted. In other words, there exists no efficient method to update a container $C(s, x)$ with a new node u that has turned out to be in $S(s, x)$, and it is necessary to actually create the sets $S(s, x)$ in memory in line 4a. The sets $S(s, x)$ can then be used to construct the containers $C(s, x)$ *off-line*, after Dijkstra's algorithm has finished for s. Remark that the storage requirement is still $O(n)$, because $\sum_{(s,x) \in E} |S(s, x)| \leq n$.

3.3 Geometric Containers

The containers that we are using are geometric objects. Recall from Section 2 that a *layout* $L : V \to \mathbb{R}^2$ of the graph in the Euclidean plane is given.

In the previous section, we explained how consistent containers are used to prune the search space of DIJKSTRA'S ALGORITHM. In particular, the correctness of the result does not depend on the layout of the graph that is used to construct the containers. However, the impact of the container for speeding up Dijkstra's Algorithm does depend on the relation of the layout and the edge weights. This section describes the geometric containers that we used in our tests. To use the containers for speeding up Dijkstra's algorithm and thus be able to answer on-line queries fast, we require that a geometric container has a description of constant size and that its containment test takes constant time.

Disk Centered at Tail. For each edge (u, v), the disk with center at u and minimum radius that covers $S(u, v)$ is computed. This is the same as finding the maximal distance of all nodes in $S(u, v)$ from u. The size of such an object is constant, because the only value that needs to be stored is the radius[3] which leads to a space consumption that is linear in the number of edges. The radius can be determined on-line by increasing it if necessary.

Ellipse. An extension of the disk is the ellipse with foci u and v and minimum radius needed to cover $S(u, v)$. It suffices to remember the radius, which can be found on-line similarly as in the disk case.

Angular Sector. Angular sectors are the objects that were used in [32]. For each edge (u, v) a node p left of (u, v) and a node q right of (u, v) are determined such that all nodes in $S(u, v)$ lie within the angular sector $\angle(p, u, q)$. The nodes p and q are chosen in a way that minimizes the angle $\angle(p, u, q)$. They can be determined in an on-line fashion: If a new node w is outside the angular sector $\angle(p, u, q)$, we set $p := w$ if w is to the left of (u, v) and $q := w$ if w is to the right of it. (Note that this is not necessarily the minimum angle at u that contains all points in $S(u, v)$.)

Circular Sector. By intersecting an angular sector with a disk at the tail of the edge, we get a circular sector. Obviously the minimal circular sector can be found on-line and needs only constant space (two points and the radius).

Smallest Enclosing Disk. The smallest enclosing disk is the unique disk with smallest area that includes all points. We use the implementation in CGAL [12] of Welzl's algorithm [45] with expected linear running time. The algorithm works off-line and storage requirement is at most three points.

Smallest Enclosing Ellipse. The smallest enclosing ellipse is a generalization of the smallest enclosing disk. Therefore, the search space using this container will be at most as large as for smallest enclosing disks (although the actual running time might be larger since the inclusion test is more expensive). Again, Welzl's algorithm is used. The space requirement is constant.

Bounding Box (Axis-Parallel Rectangle). This is the simplest object in our collection. It suffices to store four numbers for each object, which are the lower, upper, left and right boundary of the box. The bounding boxes can easily be computed on-line while the shortest paths are computed in the preprocessing.

Edge-Parallel Rectangle. Such a rectangle is not parallel to an axis, but to the edge to which it belongs. Thus, for each edge, the coordinate system is rotated and then the bounding box is determined in this rotated coordinate system. Our motivation to implement this container was the insight that the target nodes for an edge are usually situated in the direction of the

[3]In practice, the squared radius is stored to avoid the computationally expensive square root function.

edge. A rectangle that targets in this direction might therefore be a better model for the geometric region than one that is parallel to the axes. Note that storage requirements are actually the same as for a bounding box, but additional computations are required to rotate the coordinate system.

Intersection of Rectangles. The rectangle parallel to the axes and the rectangle parallel to the edge are intersected, which should lead to a smaller object. The space consumption of the two objects sums up, but is still constant, and as both objects can be computed on-line the intersection can be as well.

Smallest Enclosing Rectangle. We allow the rectangle to be oriented in any direction and search for one with smallest area containing all points. The algorithm from [42] finds such a rectangle in linear time. However, due to numerical inconsistencies we had to incorporate additional tests to assure that all points are in fact inside the rectangle. As for the minimal enclosing disk, this container has to be calculated off-line, but needs only constant space for its orientation and dimensions.

Smallest Enclosing Parallelogram. A parallelogram is a generalization of a rectangle, and surprisingly Toussaint's idea to use rotating calipers can been extended to find the smallest enclosing parallelogram [34]. Space consumption is constant and the algorithm is off-line.

Convex Hull. As the convex hull is the smallest enclosing convex polygon of the points, it does not fulfill our requirement that containers must be of constant size. It is included here, because it provides a lower bound for all convex objects. If there is a best convex container, it cannot exclude more points than the convex hull.

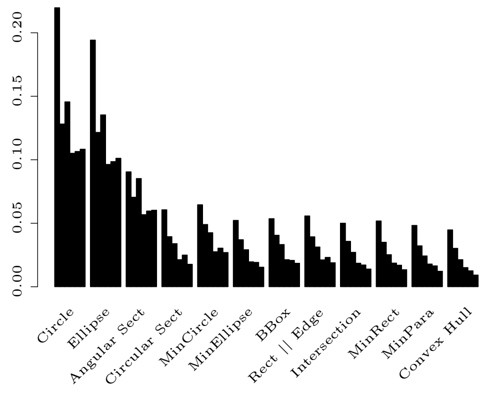
For some types of containers, it is obvious that they are at least as good as others. In particular, if a container is a subset of another container, using the first container in Dijkstra's algorithm with pruning excludes at least as many nodes as the second container. If we assume that the distribution of the nodes is uniformly at random, the expected number of nodes inside a geometric container is proportional to its area. The larger the container, the larger the average number of wrong nodes should be.

3.4 Experimental Setup

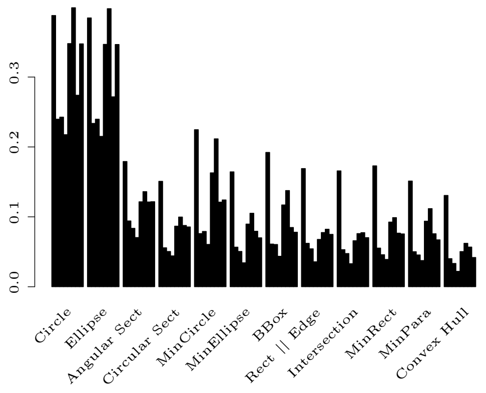
We implemented the algorithm in `C++` using `g++ 2.95.3`. We used the graph data structure from LEDA 4.3 (see [23]) as well as the Fibonacci heaps and the convex hull algorithm provided. I/O was done by the LEDA extension package for GraphML with Xerces 2.1. For the minimal disks, ellipses and parallelograms, we used CGAL 2.4 [12]. In order to perform efficient containment tests for minimal disks, we converted the result from arbitrary precision to built-in doubles. To overcome numerical inaccuracies, the radius was increased if necessary to guarantee that all points are in fact inside the container. For minimal ellipses, we used arbitrary precision which affects the running time but not the search space. Instead of calculating the minimal disk (or ellipse) of a point set, we determine the minimal disk (or ellipse) of the convex hull. This speeds up the preprocessing for these containers considerably. Although CGAL also provides an algorithm for minimal rectangles, we decided to implement one ourselves, because one cannot simply increase a radius in this case. Due to numeric instabilities, our implementation does not guarantee to find the minimal container, but asserts that all points are inside the container. The convex hulls was computed with LEDA [23]. The experiments were performed on an Intel Xeon with 2.4 GHz on the Linux 2.4 platform.

It is crucial to this problem to do the statistics with data that stem from real applications. We are using two types of data:

Street Networks. We have gathered street maps from various public Internet servers. They cover some American cities and their surroundings. Unfortunately the maps did not contain



(a) Railway: Netherlands 409, Austria 1660, Switzerland 2279, Italy 2399, France 4598, Germany 6884



(b) Streets: Moab 1444, Sacramento 3045, New Hampshire 16471, Lodi 25982, Healdsburg 38823, Point Arena 45073, Road77 45852, Polk 51510

Figure 3: Average number of visited nodes relative to Dijkstra's algorithm for all graphs and geometric objects. The graphs are ordered according to the number of nodes.

more information than the mere location of the streets. In particular, streets are not distinguished from freeways, and one-way streets are not marked as such, which makes these graphs bidirected with the Euclidean edge length. The street networks are typically very sparse with an average degree hardly above 2. The size of these networks varies from 1444 to 20466 nodes.

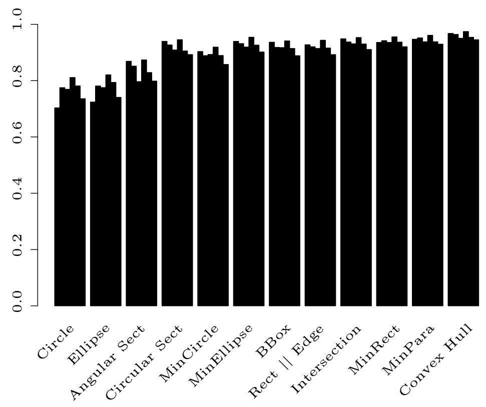
Railway Networks. The railway networks of different European countries were derived from the winter 1996/1997 time table. The nodes of such a graph are the stations and an edge between two stations exists iff there is an non-stop connection. The edges are weighted by the average travel time. In particular, here the weights do *not* directly correspond to the layout. They have between 409 nodes (Netherlands) and 6884 nodes (Germany) but are not as sparse as the street graphs.

All test sets were converted to the XML-based GraphML file format [6] to allow a unified processing.

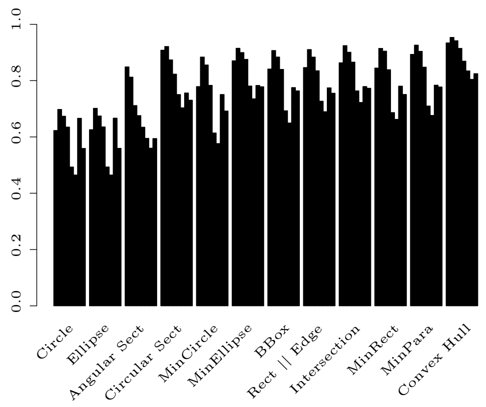
We sampled random single source single target queries to determine the average number of nodes that are visited by the algorithm. Sampling is based on the *length of the* $(1-\frac{1}{\alpha})$*-confidence interval on this average* which is $2t_{n-1,1-\frac{\alpha}{2}}sn^{-\frac{1}{2}}$ [46], where $t_{n-1,1-\frac{\alpha}{2}}$ denotes the $1-\frac{\alpha}{2}$-quantile of Student's t-distribution with $n-1$ degrees of freedom, where n is the number of samples, s the standard error, and α our chosen error probability. The sampling was done until the length of the 95%-confidence interval was smaller than 5% of the average search space. For $n > 100$ we approximated the t-distribution by the normal distribution. Note that the sample mean $\overline{x}$ and the standard error s can be calculated recursively with $\overline{x}_{(n)} = \frac{1}{n}\left(\overline{x}_{(n-1)}(n-1)+x_n\right)$ and $s^2_{(n)} = \frac{1}{n-1}\left[(n-2)s^2_{(n-1)} + (n-1)\overline{x}^2_{(n-1)} + x_n^2 - n\overline{x}^2_{(n)}\right]$, where the subscripts in brackets mark the sample size. Using these formulas, it is possible to run random single source single target shortest path queries until the length of the confidence interval is small enough.

3.5 Computational Results

Figure 3(a) depicts the results for railway networks. The average number of nodes that the algorithm visited are shown. To enable the comparison of the result for different graphs, the numbers are relative to the average search space of Dijkstra's algorithm (without pruning).

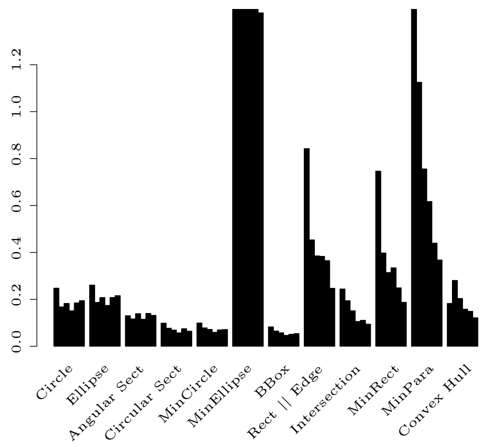


(a) Railway: Netherlands 409, Austria 1660, Switzerland 2279, Italy 2399, France 4598, Germany 6884

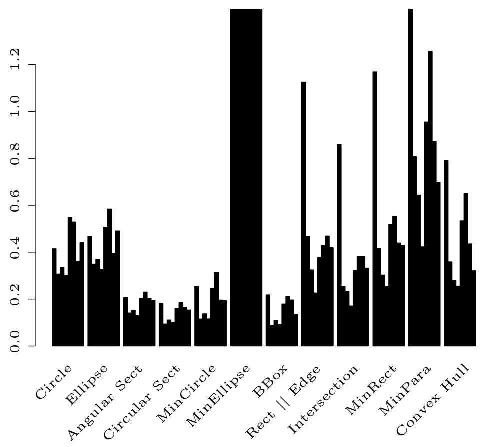


(b) Streets: Moab 1444, Sacramento 3045, New Hampshire 16471, Lodi 25982, Healdsburg 38823, Point Arena 45073, Road77 45852, Polk 51510

Figure 4: Quality according to Definition 3 for all graphs and geometric objects. The graphs are ordered according to the number of nodes.



(a) Railway: Netherlands 409, Austria 1660, Switzerland 2279, Italy 2399, France 4598, Germany 6884



(b) Streets: Moab 1444, Sacramento 3045, New Hampshire 16471, Lodi 25982, Healdsburg 38823, Point Arena 45073, Road77 45852, Polk 51510

Figure 5: Average query running time relative to DIJKSTRA'S ALGORITHM for all data sets and geometric objects. The values for minimal ellipse and minimal parallelogram are clipped. They use arbitrary precision and are therefore much slower than the other containment tests.

As expected, the pruning for disks around the tail is by far not as good as the other methods. Note however, that the average search space is still reduced to about 10%. The only type of objects studied previously [32], the angular sectors, result in a reduction to about 6%, but, if both are intersected, we get only 3.5%. Surprisingly the result for bounding boxes is about the same as for the better tailored circular sectors, directed and even minimal rectangles or parallelograms.

Of course the results of more general containers like minimal rectangle vs. bounding box are better, but the differences are not very big. Furthermore, the difference to our lower bound for convex objects is comparatively small.

The data sets are ordered according to their size. In most cases the speed-up is better the larger the graph. This can be explained by the observation that the search space of a lot of queries is already limited by the size of the graph.

In Figure 4, the quality according to Definition 3 is shown. Comparing Figures 3 and 4 confirms that the quality reflects the search-space. Containers that result in few visited nodes have a higher quality. Furthermore the quality is not so dependent on the size of the graph, because the quality measure is normalized.

Finally, we examined the average running time. We depict them in Fig. 5, again relative to the running time of the unmodified Dijkstra. It is obvious that the slightly smaller search space for the more complicated containers does not pay off. In fact the simplest container, the axis-parallel bounding box, results in the fastest algorithm.

4 Updating Containers

If a weight of an edge is changed, some containers must be updated to stay consistent. Generally speaking, for every new shortest path $u_0, u_1, \ldots, u_{k-1}, u_k$ in the graph, $C(u_0, u_1)$ has to be updated to include u_k. If we maintain containers C^{rev} for reversed edges (e.g., to perform a bi-directional search), u_0 must be added to $C^{\text{rev}}(u_{k-1}, u_k)$. (We will refer to the containers for this graph with reversed edges as *reverse containers* and mark them with the superscript "rev".)

In this section, we will present necessary conditions for new shortest paths when edge weights increase or decrease. They enable us to maintain consistent containers without running completely from scratch CREATE-CONTAINERS (Algorithm 2). Throughout the 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 $(x, y) \in E$. The following lemma is suited to restrict the set of containers that we have to update.

Lemma 4 *Let $G = (V, E)$, $w : E \to \mathbb{R}$ 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 of a shortest x-t-path is (x, y). □

By the definition of (reverse) containers, the first and last nodes of new shortest paths can be described as follows.

Corollary 5 *Let $G = (V, E)$, $w : E \to \mathbb{R}$ 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*

$$s \in C^{\text{rev}}_{\text{old}}(x, y) \qquad \textit{and} \qquad t \in C_{\text{old}}(x, y).$$

Lemma 6 *Let $G = (V, E)$, $w : E \to \mathbb{R}$ 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. □

4.2 Decreasing an edge weight

Similar to the case of a weight increase, we can prove a lemma about start and end nodes of new shortest paths for the case of a weight decrease.

Lemma 7 *Let $G = (V, E)$, $w : E \to \mathbb{R}$ 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 is also a shortest path, P_{sy} is a shortest that ends with the edge (x, y). For symmetric reasons, the first edge of a shortest x-t-path is (x, y). □

Again the condition provides a simple test using the containers $C(x, y)$ and $C^{\text{rev}}(x, y)$. This time however, both containers must already be updated.

Corollary 8 *Let $G = (V, E)$, $w : E \to \mathbb{R}$ 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*

$$s \in C^{\text{rev}}_{\text{new}}(x, y) \qquad \text{and} \qquad t \in C_{\text{new}}(x, y).$$

In order to run a (modified) Dijkstra for all nodes in $C_{\text{new}}(x, y)$, it is necessary to compute $C_{\text{new}}(x, y)$, i.e., to enlarge it if necessary. This can be done similarly to its creation in Create-Containers (Algorithm 2). In contrast to Create-Containers, the loop in line 0 is replaced by a single run for $s := x$. Furthermore in line 4a, only the container $C(x, y)$ must be enlarged (i.e., $A[u] = (x, y)$). Finally, Dijkstra's algorithm can be truncated to the part of the graph, where distance labels change. This can be achieved by executing lines 8–17 only if `new-dist` $< w_{\text{old}}(x, y) + d_{\text{old}}(y, u)$. If a node v is excluded because `new-dist` $\geq w_{\text{old}}(x, y) + d_{\text{old}}(y, u)$, 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.

Similarly to Lemma 6 a condition for edge weight decreases can be realized, but this time the old weight of the edge (x, y) is used in the comparison.

Lemma 9 *Let $G = (V, E)$, $w : E \to \mathbb{R}$ 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 $d_{\text{new}}(s, t) = d_{\text{new}}(s, x) + w_{\text{new}}(x, y) + d_{\text{new}}(y, t) < d_{\text{new}}(s, x) +$

$w_{\text{old}}(x, y) + d_{\text{new}}(y, t)$. Consider now some node $u \in P_{\text{new}}$. 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, then $d_{\text{new}}(s, u) < d_{\text{new}}(s, x) < d_{\text{new}}(s, x) + w_{\text{old}}(x, y) + d_{\text{new}}(y, t)$, 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. □

4.3 Update strategies

As running Create-Containers after each update is not desirable, we look for faster methods to maintain consistent containers (but possibly with worse quality). If containers are too large, then their quality is decreased but their consistency is preserved. The first helpful observation is the fact that only a part of the containers may be too small. According to Lemma 4 and 7, only those containers have to be updated that belong to an outgoing edge of a node $s \in V$, where the last edge on a shortest s-y-path is (x, y). We will call such nodes s *potentially affected* and denote their number by p. The potentially affected nodes can be determined by a run of a modified Dijkstra starting at y with reversed edges.

If we maintain reverse containers, Corollary 5 and 8 provide an even simpler method to find containers that may need maintenance. When a weight of an edge (x, y) has changed, only those containers must be updated that belong to an outgoing edge of a node in $C^{\text{rev}}(x, y)$. Symmetrically the reverse containers that belong to an incoming edge of a node in $C(x, y)$ should be checked. If the weight has been decreased, the containers $C(x, y)$ and $C^{\text{rev}}(x, y)$ must be updated as described in the previous section *before* we determine the nodes inside them.

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

Compute the container from scratch. The result is slightly different from recomputing all containers from scratch, because not all containers are recomputed. A container that can shrink is not necessarily updated. As Dijkstra's algorithm is run for every potentially affected node, the overall running time is bounded by $O(p \cdot n \log n)$.

Set 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 drops dramatically with this method although the running time – being linear in p – is appealing.

Enlarge the container as much as necessary. A variant of Dijkstra's algorithm truncated according to Lemma 6 and 9 can be used to enlarge on-line containers. (This method is not applicable for off-line containers.) More precisely, the lines 8–17 of Create-Containers are only executed for a node v that satisfies `new-dist` $< d(s, x) + w_{\text{new}}(x, y) + d(y, v)$ or `new-dist` $< d(s, x) + w_{\text{old}}(x, y) + d(y, v)$, respectively. In particular, the rest of the nodes are never inserted in the queue Q. The complexity of this update strategy is therefore $O(p \cdot k \log k)$, if for all potentially affected nodes s an upper bound k exists for the size of the set of nodes fulfilling the condition in Lemma 6 and 9 for edge weight increases and decreases, respectively.

The conditions in Lemma 6 and 9 use distance values $d(u, x)$ and $d(y, u)$ for different $u \in E$, which have to be computed beforehand by running two instances of Dijkstra's algorithm.

The reverse containers are the counterpart to normal containers, if all edges in the graph are reversed. So, the same algorithm can be applied to a graph with reversed edges to enlarge the reverse containers as necessary.

4.4 Experimental Setup

We performed an experimental study to evaluate the performance and quality of our algorithms. More precisely we examined how much time is needed to update the containers (on average) and

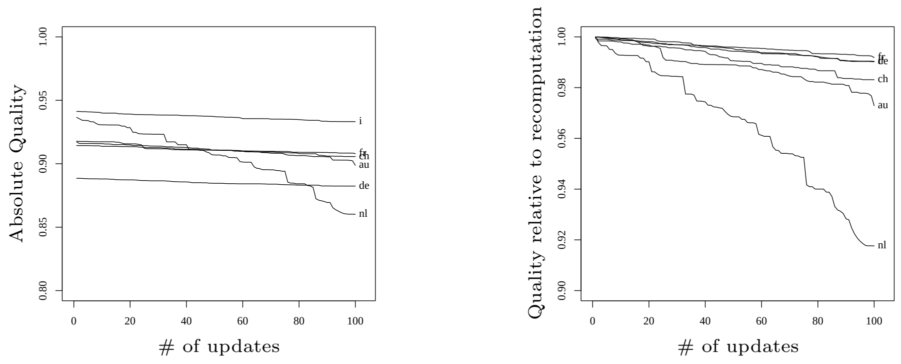


Figure 6: The quality of updated containers for 100 increased edge weights. The containers are enlarged by a truncated Dijkstra for all nodes s with (x, y) as the last edge of a shortest s-y-path. In the right diagram the quality is divided by the quality of containers computed from scratch.

how much the containers differ from containers computed from scratch. (Remember that we do not shrink all containers in our updates.)

As the experiments in the static case showed that bounding boxes are the fastest container type in practice, the experiments for the dynamic version are performed with this geometric container. Since the construction of all containers from scratch (to compare them to updated containers) is very time consuming, we evaluate the different update strategies only for the railway networks.

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 4.3. 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.

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.

4.5 Computational Results

In order to reflect the fact that also the quality of containers that are computed from scratch varies from graph to graph and after each update, we examined the the quality of updated containers relative to the quality of containers that are computed from scratch. It turns out that in both cases – with and without reverse 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 (Figure 6). 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 with a quality below 0.2 where 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.

In the case where containers of incident edges are recomputed from scratch the resulting containers coincide most of the time with the newly computed containers. In other words, the quality

	reverse containers		used			not used		
	container update		set to infinity	enlarge	from scratch	set to infinity	enlarge	from scratch
	n	m						
nl	409	1215	1331; 173	2.17;2.00	2.45;2.70	269; 228	2.52;3.43	5.02;3.02
au	1660	4327	8093; 700	1.99;1.90	2.54;2.38	1551;1429	2.07;2.10	2.13;2.10
ch	2279	6015	10211; 953	2.18;2.24	2.56;2.58	2235;2300	2.77;2.85	2.48;3.09
i	2399	8008	9552; 965	2.85;2.75	2.77;2.68	2095;2097	2.80;2.65	2.64;2.94
fr	4598	14937	17691;1932	2.09;2.21	2.87;2.66	4382;4291	2.55;3.13	4.25;3.36
de	6884	18601	33160;2974	1.72;1.71	2.04;2.21	6568;6583	2.53;2.31	2.56;2.76

Table 1: Average speed-up for updating the containers with increasing weights (first number) and decreasing weights (second number).

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. 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. They report the speed-up factor as the ratio between the time required for computing containers from scratch and the time for updating the containers. The three types of updates (enlarge the containers to infinity, enlarge the containers according to Lemma 6 and 9, and recompute the containers from scratch) were tested with maintenance of reverse 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 unacceptable quality. We report them only for the 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 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 reverse containers are quite similar, but note that the absolute time values with reverse containers are about twice as large. Maintaining reverse containers can therefore be justified only if they are used for other purposes as well (e.g., for bidirectional 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 computing the distances to x and from y for all nodes outweigh the gain of the pruning.

5 Implementation

In the implementation of experimental algorithms, switching flexibly between different variants is a major issue apart from the efficiency of the algorithm. In our case, a basic algorithm (Dijkstra's algorithm) is refined in different ways in order to compare their performance and output. As a concrete example, the shortest path computation with Dijkstra's algorithm can be improved by goal-directed search or geometric pruning. Apart from such refinements, other so-called *aspects* [40] can be added to the algorithm: operation counting, time measurement or debugging output.

5.1 Adding Aspects by Parameterized Inheritance

For complex algorithms, it is favorable to keep the code of every aspect separate from the basic algorithm while preserving the efficiency. The basic algorithm is defined as a class with virtual functions at major key points as sketched below:

```
struct Algo {
  virtual void init() {...}
  virtual bool finished() {...}
  virtual void do_something() {...}
```

```
    void run() {
      init();
      while (!finished())
       { do_something(); }
    }
  };
```

In our case, the algorithm is of course DIJKSTRA'S ALGORITHM. The constructor function of `Algo` initializes the priority queue, `init()` inserts the source in the priority queue, `finished()` tests whether the priority queue is empty, and so on. The function `do_something()` is just a mere representative for the number of functions inside the loop.

Suppose we are interested in the number of calls to the function `do_something()`. The aspect in question is then added by deriving a subclass that modifies virtual functions of the base class. This realization has the drawback that different aspects cannot be combined freely, because the base class is fixed. We get a much more flexible system, if we turn the aspect into a mix-in class, i.e., we make the base class a template argument.

```
template<typename Base> struct AspectCount::public Base {
  int operations;
  virtual void init() { Base::init(); operations=0; }
  virtual void do_something() { Base::do_something(); ++operations; }
};
```

Note that calling the base class function is mandatory in order to preserve the correctness of the algorithm and other aspects.

We are now ready to explain how different variants of Dijkstra's algorithm can be realized using aspects. Suppose that another aspect is implemented similarly to the above, e.g., `AspectTarget` for terminating DIJKSTRA'S ALGORITHM when the target is finished. The four variants of Dijkstra's algorithm with operation counting, termination at target, and termination at target with operation counting, can be easily instantiated:

```
Dijkstra AlgoA;                                AlgoA.run(source,target);
AspectCount<Dijkstra> AlgoB;                   AlgoB.run(source,target);
AspectTarget<Dijkstra> AlgoC;                  AlgoC.run(source,target);
AspectTarget<AspectCount<Dijkstra> > AlgoD; AlgoD.run(source,target);
```

Thus, it is possible with parameterized inheritance to instantiate all combinations of aspects, while it is not necessary to actually implement the (exponentially many) combinations. Since all functions are inlined, the function calls can be optimized away by the compiler which leads to a flexible and efficient set of algorithms.

Other aspects that we implemented include storing a layout of the graph, goal-directed search, geometric pruning, performing operation counts including mean value and variance, constructing on-line or off-line containers in CREATE-CONTAINERS, or truncating DIJKSTRA'S ALGORITHM according to Lemma 6 or 9 for updates after changing edge weight.

5.2 Constructor Parameter List

The initialization of the algorithm is done by the constructor of the class and necessary parameters are given as arguments to the constructor. We will now discuss the question how additional parameters that are needed by aspects can be provided. We use a technique inspired by [10], but omit the creation of a repository.

Consider again Dijkstra's algorithm. The parameters that are given to the constructor are the graph and the edge lengths in this case. A speed-up technique that uses a layout of the graph would expect the layout in addition to the graph and the edge lengths.

The constructor of the aspect has to pass along the parameters to the base class.

```
Algo::Algo(Type1 Parameter1){...}
```

```
template<typename Base>
AspectA<Base>::AspectA(Type1 Parameter1, Type2 Parameter2):
  public Base(Parameter1)
{ do something with Parameter2 }
```

If there is another AspectB that expects another Parameter and we want to freely combine them, we have the problem that the constructor does not know which parameters the base class (including other aspects) needs. Either we have to write constructors for all combinations of parameters, or we have to sacrifice type safety by passing a list of pointers to `void` that are then casted to the respective parameters. A third method, the one that we follow, is to pass a meta-list of types. We will now describe, how such a meta-list can be realized in a way similar to that in [9].

```
struct END {};
static END End;

template<typename T, typename NEXT=END> struct LISTITEM {
  T &value;
  NEXT next;
};
```

The template class `LISTITEM` is the list item of our parameter list. It holds a value and a pointer to the next list item. The class `END` is used as an anchor to create an empty list. Usage of the list is as follows.

```
struct Algo {
  typedef LISTITEM<Type1> ParamType;
  Algo(ParamType Parameter);
};

template<typename Base> struct AspectA::public Base {
  typedef LISTITEM<Type2,Base::ParamType> ParamType;
  AspectA(ParamType Parameter):Base(Parameter.next)
  { do something with Parameter.value }
};
```

An aspect that needs to add a parameter can do this by attaching it to the parameter type `ParamType`. It can access the value of the additional parameter as `Parameter.value` and pass the rest of the list `Parameter.next` to its base class. The list that is given to the base class could be the parameter list of `Algo`, but it can also contain further arguments that are needed by other aspects. Since the parameter list `Base::ParamList` of the template argument `Base` is used, both cases are handled by this construction.

In order to nicely construct the parameter list, we need the class `LISTITEM` to provide generic constructors for lists with different lengths:

```
template<typename T, typename NEXT=END>
struct LISTITEM {
  T &value;
  NEXT next;

  LISTITEM(T &t):value(t),next(Nil){}
  template<typename T2> LISTITEM(T2 &t2, T &t):value(t), next(t2) {}
  template<typename T2, typename T3> LISTITEM(T3 &t3, T2 &t2, T &t):
      value(t), next(t3,t2) {}
  // ... and so on
};
```

The best way to show how simple it is to use a parameter list is to provide an example. Assume our basic algorithm Dijkstra needs the parameters G and w. Furthermore, we would like to use an aspect PruningAspect, which needs the parameter L, and an aspect CountingAspect without additional parameters. To realize such an algorithm, it is simply needed to provide the three parameters as the meta-list:

```
PruningAspect<CountingAspect<Dijkstra> >::ParamType P(G, w, L);
PruningAspect<CountingAspect<Dijkstra> > Algo(P);
```

For obvious reasons, it is required to know which aspects need what additional parameters. Furthermore, the respective parameters must be given in the same order as the aspects are added.

5.3 Dependencies

Another issue of the code organization is that some aspects depend on others. To give a concrete example, goal-directed search as well as geometric pruning depend both on a layout of the graph. Template meta-programming can also be used to check dependencies of such concepts [39]. In our case, concepts coincide most of the time with the use of a mix-in class (a derived class where the base class is a template parameter). This enables us to actually *add* the mix-in class in case it is needed but has not been included yet. (The aspect “layout” must only be added once if both goal-directed search and geometric pruning are used.)

In order to check a condition at compile-time, we make use of partial specialization as presented in [9]:

```
template<bool Cond, typename A, typename B>
struct IF { typedef A RET; };

template<typename A, typename B>
struct IF<false,A,B> { typedef B RET; };
```

The template class IF can be used to decide at compile time, which type A or B is used depending on the condition Cond. The “return value” of the meta function IF is the type RET. Consider the definition

```
IF<sizeof(int)<4, int, long>::RET a
```

If the size of int is 4, the general definition of IF is used. Therefore a is of type int. If the size of int is smaller than 4, the specialization of IF is used and a is of type long.

Our goal is to test at compile time whether the given base class T provides a certain aspect Aspect (e.g., whether the algorithm class stores a layout of the graph). Hence, what we need is to test whether the base class T is of the form Aspect<B> for some class B. Furthermore the base class T also provides the aspect Aspect, if it is *derived* from Aspect<B> (e.g., because some other aspect has been added after Aspect). Testing whether a class T is derived from a class B is a little bit tricky (see Chapter 2.7 in [1] for more details).

```
template<typename T, template<typename L> class Aspect>
struct Provides {
private:
  class Yes { char a[1]; };
  class No { char a[10]; };

  static No ProvidesTest( ... );
  template<typename S> static Yes ProvidesTest( Aspect<S> const* );

public:
  static bool const RET = (sizeof(ProvidesTest(static_cast<T*>(0))) ==
                           sizeof(Yes));
};
```

The aspect to test is provided as a template-template argument (a template class as template argument) of the class `Provides`. The template function `ProvidesTest(Aspect<S> const*)` accepts a pointer to `Aspect<S>` for some class `S` or a pointer to a class that is derived from `Aspect<S>`. The function `ProvidesTest(...)` can take any pointer as argument. Both functions will actually never be called, so there is no need to define them. What's important about these functions is that they differ in their return type. If we call `ProvidesTest()` with a pointer to a class derived from `Aspect<S>`, the return type would be `Yes`. Otherwise it would be `No`. Even more important is that the sizes of their return types differ. Hence, we can distinguish by `sizeof(ProvidesTest(static_cast<T*>(0))` whether `T` is derived from `Aspect<S>`. The return value of the meta function `Provides` is stored in `RET`.

Using the template classes `IF` and `Provides`, we are now able to add to a base class an aspect if and only if it is not already included:

```
template<typename Base, template<typename L> class Aspect>
struct EnsureAspect {
  typedef typename IF<Provides<Base,Aspect>::RET, Base,
                           Aspect<Base> >::RET  RET;
};
```

Usage is as follows:

```
template<typename Base>
struct A:public EnsureAspect<Base,B>::RET {
  typedef typename EnsureAspect<Base,B>::RET MyBase;
  // ... further class members
};
```

The aspect `A` (e.g., geometric pruning) needs the base class to include aspect `B` (e.g., layout). If `Base` does not provide it, `A` is not derived from `Base` but from `B<Base>`. For our convenience, the type of the actual base class is remembered as `MyBase`.

For a concrete application we return to our main example. Geometric pruning `PruningAspect` and goal-directed search `GoalDirectedAspect` both need a layout `LayoutAspect`. If they ensure the usage of `LayoutAspect` as shown above, it is not necessary to include this aspect by hand:

```
PruningAspect<LayoutAspect<Dijkstra> >::ParamType P1(G, Lengths, Layout);
PruningAspect<LayoutAspect<Dijkstra> > Algo1(P1);

PruningAspect<Dijkstra>::ParamType P2(G, Lengths, Layout);
PruningAspect<Dijkstra> Algo2(P2);
```

Both instantiations `Algo1` and `Algo2` result in the same algorithm. Furthermore, the aspects `PruningAspect` and `GoalDirectedAspect` can be combined as

```
PruningAspect<GoalDirectedAspect<Dijkstra> >::ParamType P3(G,Lengths,Layout);
PruningAspect<GoalDirectedAspect<Dijkstra> > Algo3(P3);
```

Again, it is necessary to know which parameters are needed and in which order according to the added aspects *including* their dependencies. However instantiating an algorithm gets much shorter and clearer, since only the main aspects need to be mentioned.

6 Conclusion

We have seen that using a layout may lead to a considerable speed-up in Dijkstra's algorithm, if one allows a suitable preprocessing. Actually, we are able to reduce the search space to only 5 – 10%, while even "bad" containers result in a reduction to less than 50%. The somewhat surprising result is that the simple bounding box outperforms other geometric objects in terms of CPU cycles in a lot of cases. The presented technique can easily be combined with other methods:

- The geometric pruning is independent of the priority queue. Algorithms using a special priority queue such as [24, 17] can easily be combined with it. The decrease of the search space is in fact the same (but the actual running time would be different).
- Goal-directed search [35] or A^* has been shown in [37, 19] to be very useful for transportation networks. As it simply modifies the edge weights, a combination of geometric pruning and A^* can be realized in a straightforward manner.
- Bidirectional search [27] can be integrated by reverting all edges and running the preprocessing a second time. Space and time consumption for the preprocessing simply doubles.
- In combination with a multi-level approach [21, 33], one constructs a graph containing all levels and inter-level edges first. The geometric pruning is then performed on this graph.

In the dynamic case, 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] A. Alexandrescu; *Modern C++ design: generic programming and design patterns applied*; Addison-Wesley; 2001

[2] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, U. Nanni; Incremental algorithms for minimal length paths; *Journal of Algorithms*; 12:615; 1991

[3] C. Barrett, K. Bisset, R. Jacob, G. Konjevod, M. Marathe; Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router; *Proc. 10th European Symposium on Algorithms (ESA 2002)*, eds. R. Möhring, R. Raman; LNCS 2461; 126–138; Springer; 2002

[4] C. Barrett, R. Jacob, M. Marathe; Formal-language-constrained path problems; *SIAM Journal on Computing*; 30(3):809; 2000

[5] G. Bracha, W. Cook; Mixin-based inheritance; *Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proc. European Conference on Object-Oriented Programming*, ed. N. Meyrowitz; 303–311; ACM Press; 1990

[6] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, M. Scott; GraphML progress report; *Proc. Graph Drawing (GD 2001)*, eds. P. Mutzel, M. Jünger, S. Leipert; LNCS 2265; 501–512; Springer; 2001

[7] C. Demetrescu, G. F. Italiano; A new approach to dynamic all pairs shortest paths; *Proc. 35th ACM Symposium on Theory of Computing (STOC 2003)*; 159 – 166; ACM Press; 2003

[8] E. W. Dijkstra; A note on two problems in connexion with graphs; *Numerische Mathematik*; 1:269; 1959

[9] U. Eisenecker, K. Czarnecki; *Generative Programming in C++*; chap. 10, 397–501; Addison-Wesley; 2000

[10] U. W. Eisenecker, F. Blinn, K. Czarnecki; A solution to the constructor-problem of mixin-based programming in C++; *Proc. 1st Workshop on C++ Template Programming*; Erfurt, Germany; 2000

[11] S. Even, H. Gazit; Updating distances in dynamic graphs; *Methods of Operations Research*; 49:371; 1985

[12] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, S. Schönherr; On the design of CGAL a computational geometry algorithms library; *Softw. – Pract. Exp.*; 30(11):1167; 2000

[13] M. L. Fredman, R. E. Tarjan; Fibonacci heaps and their uses in improved network optimization algorithms; *Journal of the ACM (JACM)*; 34(3):596; 1987

[14] D. Frigioni; Semidynamic algorithms for maintaining single-source shortest path trees; *Algorithmica*; 22(3):250; 1998

[15] D. Frigioni, A. Marchetti-Spaccamela, U. Nanni; Fully dynamic output bounded single source shortest path problem; *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'96)*; 212–221; 1996

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides; *Design Patterns: Elements od Reusable Object-Oriented Software*; Addison-Wesley Professional Computing Series; Addison-Wesley; 1995

[17] A. V. Goldberg; A simple shortest path algorithm with linear average time; *Proc. 9th European Symposium on Algorithms (ESA 2001)*, ed. F. Meyer auf der Heide; LNCS 2161; 230–241; Springer; 2001

[18] F. J. Hauck; Inheritance modeled with explicit bindings: an approach to typed inheritance; *SIGPLAN Notices*; 28(10); 1993

[19] R. Jacob, M. Marathe, K. Nagel; A computational study of routing algorithms for realistic transportation networks; *Proc. 2nd Workshop on Algorithm Engineering (WAE'98)*, ed. K. Mehlhorn; 167–178; 1998

[20] D. B. Johnson; Efficient algorithms for shortest paths in sparse networks; *Journal of the ACM (JACM)*; 24(1):1; 1977

[21] S. Jung, S. Pramanik; HiTi graph model of topographical road maps in navigation systems; *Proc. 12th IEEE Int. Conf. Data Eng.*; 76–84; 1996

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

[23] K. Mehlhorn, S. Näher; *LEDA, A platform for Combinatorial and Geometric Computing*; Cambridge University Press; 1999

[24] U. Meyer; Single-source shortest-paths on arbitrary directed graphs in linear average-case time; *Proc. Symposium on Discrete Algorithms*; 797–806; 2001

[25] K. Nachtigall; Time depending shortest-path problems with applications to railway networks; *European Journal of Operational Research*; 83(1):154; 1995

[26] S. Pettie, V. Ramachandran, S. Sridhar; Experimental evaluation of a new shortest path algorithm; *Proc. Algorithm Engineering and Experiments (ALENEX'02)*; LNCS 2409; 126–142; Springer; 2002

[27] I. Pohl; Bi-directional search; *Proc. 6th Annual Machine Intelligence Workshop*, eds. B. Meltzer, D. Michie; Machine Intelligence 6; 137–140; Edinburgh University Press; 1971

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

[29] G. Ramalingam, T. W. Reps; An incremental algorithm for a generalization of the shortest-path problem; *Journal of Algorithms*; 21(2):267; 1996

[30] G. Ramalingam, T. W. Reps; On the computational complexity of dynamic graph problems; *Theoretical Computer Science*; 158:233; 1996

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

[32] F. Schulz, D. Wagner, K. Weihe; Dijkstra's algorithm on-line: An empirical case study from public railroad transport; *ACM Journal of Experimental Algorithmics*; 5(12); 2000

[33] F. Schulz, D. Wagner, C. Zaroliagis; Using multi-level graphs for timetable information; *Proc. Algorithm Engineering and Experiments (ALENEX'02)*; LNCS 2409; 43–59; Springer; 2002

[34] C. Schwarz, J. Teich, A. Vainshtein, E. Welzl, B. L. Evans; Minimal enclosing parallelogram with application; *Proc. 11th Annual Symposium on Computational Geometry*; 434–435; ACM Press; 1995

[35] R. Sedgewick, J. S. Vitter; Shortest paths in euclidean space; *Algorithmica*; 1(1):31; 1986

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

[37] S. Shekhar, A. Kohli, M. Coyle; Path computation algorithms for advanced traveler information system (ATIS); *Proc. 9th IEEE Intl. Conf. Data Eng.*; 31–39; 1993

[38] J. Siek, L.-Q. Lee, A. Lumsdaine; *The Boost Graph Library: User Guide and Reference Manual*; Addison-Wesley; 2002

[39] J. Siek, A. Lumsdaine; Concept checking: Binding parametric polymorphism in C++; *Proc. 1st Workshop on C++ Template Programming*; Erfurt, Germany; 2000

[40] O. Spinczyk, A. Gal, W. Schroder-Preikschat; AspectC++: An aspect-oriented extension to the C++ programming language; *Proc. 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, eds. J. Noble, J. Potter; Conferences in Research and Practice in Information Technology 10; 53–60; ACS, Sydney, Australia; 2002

[41] M. Thorup; Undirected single source shortest path in linear time; *Proc. IEEE Symposium on Foundations of Computer Science (FOCS'97)*; 12–21; 1997

[42] G. Toussaint; Solving geometric problems with the rotating calipers; *Proc. IEEE Mediteranian Electrotechnical Conference (MELECON 1983)*, ed. E. N. Protonotarios; A10.02/1–4; IEEE, NY; 1983

[43] D. Wagner, T. Willhalm; Geometric speed-up techniques for finding shortest paths in large sparse graphs; *Proc. 11th European Symposium on Algorithms (ESA 2003)*, eds. G. D. Battista, U. Zwick; LNCS 2832; 776–787; Springer; 2003

[44] D. Wagner, T. Willhalm, C. Zaroliagis; Dynamic shortest path containers; *Proc. Algorithmic MeThods and Models for Optimization of RailwayS (ATMOS 2003)*, ed. A. Marchetti-Spaccamela; Electronic Notes in Theoretical Computer Science 92; 2003; to appear

[45] E. Welzl; Smallest enclosing disks (balls and ellipsoids); *New Results and New Trends in Computer Science*, ed. H. Maurer; LNCS 555; Springer; 1991

[46] R. R. Wilcox; *Fundamentals of modern statistical methods: substantially improving power and accuracy*; Springer; 2001

[47] F. B. Zahn, 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(2); 2000

[48] C. Zaroliagis; Implementations and experimental studies of dynamic graph algorithms; *Experimental Algorithmics*, eds. R. Fleischer, B. Moret, E. M. Schmidt; LNCS 2547; 229–278; Springer; 2002