Geometric Containers for Efficient Shortest-Path Computation

DOROTHEA WAGNER and THOMAS WILLHALM
Universität Karlsruhe (TH)
and
CHRISTOS ZAROLIAGIS
University of Patras

A fundamental approach in finding efficiently best routes or optimal itineraries in traffic information systems is to reduce the search space (part of graph visited) of the most commonly used shortest path routine (Dijkstra's algorithm) on a suitably defined graph. We investigate reduction of the search space while simultaneously retaining data structures, created during a preprocessing phase, of size linear (i.e., optimal) to the size of the graph. We show that the search space of Dijkstra's algorithm can be significantly reduced by extracting geometric information from a given layout of the graph and by encapsulating precomputed shortest-path information in resulted geometric objects (containers). We present an extensive experimental study comparing the impact of different types of geometric containers using test data from real-world traffic networks. We also 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 and show that our new methods are two to three times faster than recomputing everything from scratch. Finally, in an appendix, we discuss the software framework that we developed to realize the implementations of all of our variants of Dijkstra's algorithm. Such a framework is not trivial to achieve as our goal was to maintain a common code base that is, at the same time, small, efficient, and flexible, as we wanted to enhance and combine several variants in any possible way.

Categories and Subject Descriptors: G.2.2 [**Graph Theory**]: Graph algorithms, Network problems; G.2.3 [**Applications**]: Traffic information systems; F.2.2 [**Nonnumerical Algorithms and Problems**]: Geometrical problems and computations, Routing and layout

General Terms: Algorithm, Design, Experimentation

This work was partially supported by the Human Potential Programme of EC under contract no. HPRN-CT-1999-00104 (AMORE), by the IST Programme of EC under contract no. IST-2002-001907 (DELIS), and by DFG under grant WA 654/12-1. 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.

Authors' addresses: Dorothea Wagner and Thomas Willhalm, Institut für Theoretische Informatik, Universität Karlsruhe (TH), Postfach 6980, 76128 Karlsruhe, Germany; email: {dwagner, willhalm}@ira.uni-karlsruhe.de and Christos Zaroliagis, 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.

Additional Key Words and Phrases: Data structures and algorithms, graph algorithms, shortest path, Dijkstra's algorithm, geometric container, traffic network

1. INTRODUCTION

A primary task in traffic information systems is to answer queries for finding best routes or optimal itineraries as efficiently as possible. In this paper, we are concerned with a typical scenario in such systems, where a central server has to answer a huge number of on-line customer queries asking for their best routes or itineraries. The most frequently encountered applications of the above scenario involve route planning systems for cars, bikes, and hikers [Zhan and Noon 2000; Barrett et al. 2002], or public transport systems for itinerary information of scheduled vehicles (like trains or buses) [Nachtigall 1995; Preuss and Syrbe 1997]. The above, query-intensive scenario, is also encountered in other domains, including spatial databases [Shekhar et al. 1997] and web searching [Barrett et al. 2000]. Since traffic information systems are real-time systems in which users continuously enter their requests for finding their best connections or routes, one of the main goals is to reduce the (average) response time for answering a query.

Answering a best-route (or optimal itinerary) query translates in computing a minimum-cost path on a suitably defined directed graph with nonnegative edge costs (or weights). This, in turn, implies that the core algorithmic problem underlying the efficient answering of queries is the single-source single-target shortest-path problem. The application of shortest-path computations in traffic networks along with the particular graph modeling is widely covered in the literature [see, for example, Barrett et al. 2002; Jung and Pramanik 2002; Nachtigall 1995; Preuss and Syrbe 1997; Schulz et al. 2000, 2002; Zhan and Noon 2000]. Although the straightforward approach of precomputing and storing shortest paths for all pairs of nodes would allow us to answer shortest-path queries optimally, the quadratic space requirements for graphs with more than 10^5 nodes makes such an approach prohibitive. Because of this and the fact that the graph associated with a traffic network is usually very large (though sparse), the main goal of almost all known approaches is to keep the space requirements as small as possible. Since traffic networks do not change for a certain period of time while there are many queries for shortest paths, a heavy preprocessing of the network (that does not blow up the space requirements) is justified in order to speed up the query time [see, for example, Jung and Pramanik 2002; Schulz et al. 2000, 2002]. The most commonly used approach for answering shortest-path queries concerns variants of Dijkstra's algorithm, targeting at reducing its *search space* (number of nodes visited by the algorithm).

Our main concern in this paper is to investigate the possibility of reducing the search space of Dijkstra's algorithm by using precomputed information that can be stored in $O(n+m)$ space, where n (resp. m) is the number of nodes (resp. edges) of the graph. Note that $O(n+m)$ is an optimal space requirement. Our main contribution is that we can significantly reduce the search space of

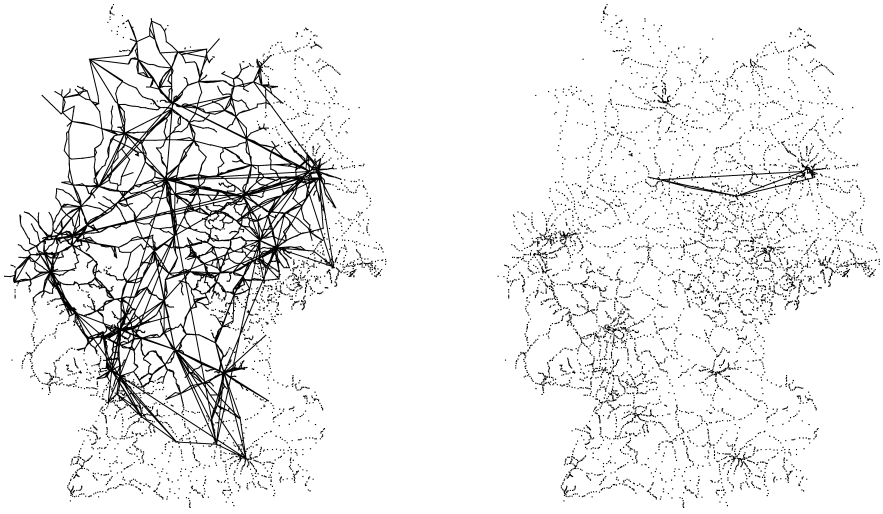


Fig. 1. The search space for a query from Hannover to Berlin for DIJKSTRA'S ALGORITHM (left) and DIJKSTRA'S ALGORITHM WITH PRUNING using bounding boxes (right).

Dijkstra's algorithm—thus answering on-line shortest-path queries fast—by extracting geometric information from a given layout of the graph. For traffic information systems such a layout is provided by the geographic locations of the nodes. In particular, our study shows that storing partial results based on geometric information reduces the number of nodes visited by Dijkstra's algorithm to only 5–10%. (Figure 1 gives an illustrative example for the German railway network.)

We use a very fundamental observation on shortest paths: 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. The main idea of our approach is as follows. Assume that during preprocessing a set of nodes $S(e)$ is computed, for each edge e, containing all nodes that can be reached by a shortest-path starting with e. Subsequently, when Dijkstra's algorithm is executed, those edges e for which the target is not in $S(e)$ are ignored. As storing all sets $S(e)$ would require $O(nm)$ space, we relax this prohibitive space requirement by storing instead a geometric object, called *container*, 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 container. Note that this method still leads to a correct result (optimal path), although it 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 containers, we use the given layout of the graph. It is, however, not required that the edge weights 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 Schulz et al. [2000] for the special case of a timetable 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. It turns out that a significant improvement can be achieved by using geometric objects other than angular sectors. Actually, in some cases, the speed-up is even a factor of about two. The somewhat surprising result is that the simple bounding box outperforms other geometric objects in terms of CPU cycles in many cases. Let us mention that two succeeding papers already took up our result and considered bounding boxes as geometric containers for efficient shortest-paths computation. In Holzer et al. [2004], a detailed experimental study for combinations of bounding boxes with other speed-up techniques is presented. The effectiveness of graph layout methods to speed-up shortest-path computations is studied in Wagner and Willhalm [2005]. The paper examines the question of how geometric speed-up techniques can be used in case there is no layout given. Both studies indicate that geometric containers also work very well for other graphs than the real-world traffic networks considered in our experiments.

The second contribution of this paper concerns the dynamic version of the considered application scenario; namely, the case where the graph may dynamically change over time as certain "disruptions" to the traffic network occur (streets may be blocked, built, or destroyed, trains/buses may be added or canceled, etc). We present new algorithms that dynamically maintain geometric containers when the weight of an edge is increased or decreased (note that these cases also cover edge deletions and insertions). We also report on an experimental study with real-world railway data. Our experiments show that the new algorithms are two to three times faster than the naive approach of recomputing the geometric containers from scratch. To the best of our knowledge, our dynamic algorithms are the first results toward an efficient algorithm for the dynamic single source shortest-path problem that does *not* use the output complexity model of computation [introduced in Ramalingam and Reps 1996a, 1996b and extended in Frigioni 1998; Frigioni et al. 1996] 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., Even and Gazit 1985; Rohnert 1985; Demetrescu and Italiano 2003; Ausiello et al. 1991; King 1999; and Zaroliagis 2002 for a recent overview] are not applicable to maintain geometric containers, because of their inherent quadratic (and in some cases even cubic) space requirements.

The remainder 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 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 changes in edge weights along with their corresponding experimental results. We conclude in Section 5. Preliminary, portions of this work appeared in Wagner and Willhalm [2003] and Wagner et al. [2004]. Finally, in the appendix of the paper, 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.

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 of $O(n)$. 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 [Johnson 1977], 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 nonnegative edge weights is that of Dijkstra [Dijkstra 1959] (Algorithm 1 without lines 3a and 5a). In the comparison model, Dijkstra's algorithm implemented with Fibonacci heaps [Fredman and Tarjan 1987] is still the fastest known algorithm for the general case of arbitrary nonnegative 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 [Goldberg 2001; Meyer 2001; Pettie et al. 2002; Thorup 1997].

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

```
 1   for all nodes u ∈ V set dist(u) := ∞
 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 ∈ C(u, v)
 7               set new-dist := dist(u) + w(u, v)
 8               if new-dist < dist(v)
 9                   if dist(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
```

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.)

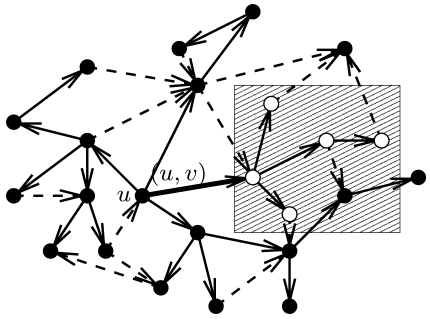


Fig. 2. Dijkstra's algorithm is run for a node $u \in V$. Let the white nodes be those nodes that can be reached on a shortest path using the edge (u, v). A geometric object is constructed that contains these nodes. It may contain other nodes, but this only affects the running time and not the correctness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

3. GEOMETRIC PRUNING

3.1 Shortest-Path 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 Figure 2. The condition in line 5a is formalized by the notion of a consistent container.

[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, for example, Schulz et al. 2000 for a detailed description.]

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.*

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 subpath of a shortest path is again a shortest path, so for all $(u, v) \in P$, the subpath of P from u to t is a shortest u-t-path. Then, by the definition of consistent container, $t \in C(u, v)$. □

This idea of pruning can be extended to bidirectional search [Pohl 1971]. 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 bidirectional search checks the normal containers whereas a backward step uses reverse containers.

The quality of a set of containers is 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 ∈ V do
 1     for all nodes u ∈ V set  dist(u) := ∞
 2     initialize priority queue Q with source s and set  dist(s) := 0
 3     while priority queue Q is not empty
 4       get node u with smallest tentative distance  dist(u) in Q
4a       if u ≠ s enlarge C(A[u]) to contain u
 5       for all neighbor nodes v of u
 7         set  new-dist :=  dist(u) + w(u, v)
 8         if  new-dist <  dist(v)
 9           if  dist(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. 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.

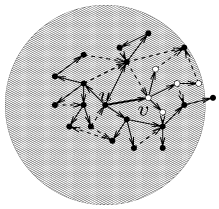
Since Dijkstra's algorithm runs in $O(n \log n)$ time for sparse graphs, the overall running time is $O(n^2 \log n)$ plus the time to construct the containers. The storage requirement is $O(n)$ plus the space required for storing the containers.

Some types of containers are 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. Note that the storage requirement is still $O(n)$, because $\sum_{(s,x)\in E} |S(s, x)| \leq n$.

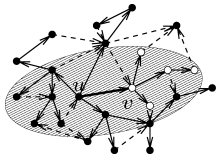
3.3 Geometric Containers

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.

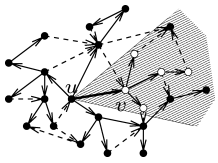
In the previous section, we explained how consistent containers are used to prune the search space of Dijkstra's algorithm. In particular, the correctness of the result does not depend on the layout of the graph that is used to construct the containers. However, the impact of the container for speeding 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 rapidly answer on-line queries, we require that a geometric container has a description of constant size and that its containment test takes constant time. We have considered the following types of containers for an edge (u, v):



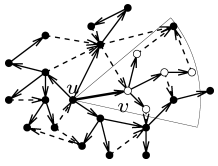
Disk Centered at Tail (Disk). For each edge (u, v), the disk with center at u and minimum radius that covers $S(u, v)$ is computed. This is the same as finding the maximal distance of all nodes in $S(u, v)$ from u. The size of such an object is constant, because the only value that needs to be stored 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 to the disk case.

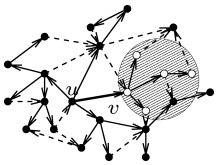


Angular Sector (Angular Sect). Angular sectors are the objects that were used in Schulz et al. [2000]. For each edge (u, v) a node p left of (u, v) and a node q right of (u, v) are determined such that all nodes in $S(u, v)$ lie within the angular sector $\angle(p, u, q)$. The nodes p and q are chosen 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 $S(u, v)$.]

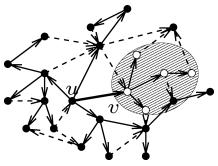


Circular Sector (Circular Sect). By intersecting an angular sector with a disk at the tail of the edge, we get a circular sector. Obviously the minimal circular sector can be found on-line and needs only constant space (two points and the radius).

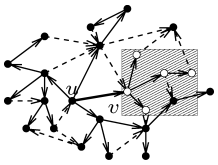
[3]In practice, the squared radius is stored to avoid the computationally expensive square root function.



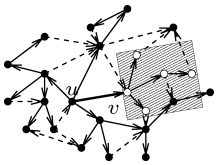
Smallest Enclosing Disk (MinDisk). The smallest enclosing disk is the unique one with smallest area that includes all points. We use the implementation in CGAL [Fabri et al. 2000] of Welzl's algorithm [Welzl 1991] with expected linear running time. The algorithm works off-line and storage requirement is, at most, three points.



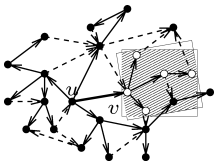
Smallest Enclosing Ellipse (MinEllipse). The smallest enclosing ellipse is a generalization of the smallest enclosing disk. Therefore, the search space using this container will be, at most, as large as for smallest enclosing disks (although the actual running time might be larger since the inclusion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



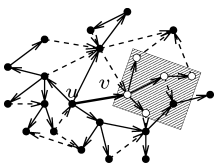
Bounding Box (BBox). This is the simplest object in our collection. It suffices to store four numbers for each object, which are the lower, upper, left, and right boundary of the box. The bounding boxes can easily be computed on-line, while the shortest paths are computed in the preprocessing.



Edge-Parallel Rectangle (Rect || Edge). 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 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.

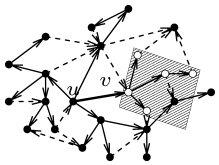


Intersection of Rectangles (Intersection). The rectangle parallel to the axes and the rectangle parallel to the edge are intersected, which should lead to a smaller object. The space consumption of the two objects sums up, but is still constant, and as both objects can be computed on-line the intersection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

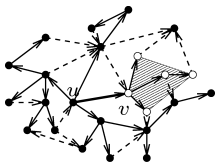


Smallest Enclosing Rectangle (MinRect). We allow the rectangle to be oriented in any direction and search for one with smallest area containing all points. The algorithm from Toussaint [1983] finds such a rectangle in linear time. However, due to numerical inconsistencies,

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 (MinPara). 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 [Schwarz et al. 1995]. 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 Mehlhorn and Näher 1999] as well as the priority queue 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 [Fabri et al. 2000]. 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 using built-in doubles. 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, in this case, one cannot simply increase a radius. Because of numeric instabilities, our implementation does not guarantee finding the minimal container, but asserts that all points are inside the container. The convex hulls was computed with LEDA [Mehlhorn and Näher 1999]. The experiments were performed on an Intel Xeon with 2.4 GHz on the Linux 2.4 platform.

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

3.4.1 *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 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 51510 nodes.

3.4.2 *Railway Networks.* The railway networks of different European countries were derived from the winter 1996/1997 timetable. The nodes of such a graph are the stations and an edge between two stations exists iff there is a nonstop 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 [Brandes et al. 2001] 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 $2t_{n-1,1-\frac{\alpha}{2}} s n^{-\frac{1}{2}}$ [Wilcox 2001], 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

$$\overline{x}_{(n)} = \frac{1}{n}(\overline{x}_{(n-1)}(n-1) + x_n)$$

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 sufficiently small.

3.5 Computational Results

Figure 3 depicts the results for railway and street 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). As expected, the use of disks centered at the edge's tail and ellipses are, by far, the worse compared to the

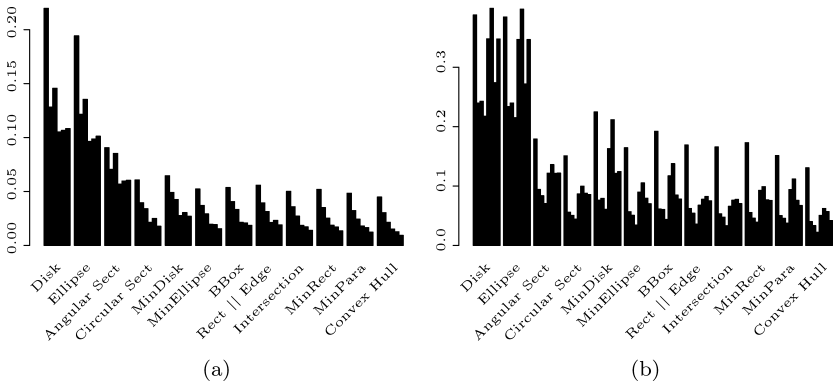


Fig. 3. Average number of visited nodes relative to DIJKSTRA'S ALGORITHM for all graphs and geometric objects. The graphs are ordered according to the number of nodes. (a) Railway: Netherlands 409, Austria 1660, Switzerland 2279, Italy 2399, France 4598, Germany 6884, (b) Streets: Moab 1444, Sacramento 3045, New Hampshire 16471, Lodi 25982, Healdsburg 38823, Point Arena 45073, Road77 45852, Polk 51510.

other methods. Note, however, that the average search space is still reduced to about 11% for railway and 25% for street networks. The type of objects studied previously [Schulz et al. 2000], the angular sectors, result in a reduction to about 6 and 10%, respectively, but the intersection of angular sectors with disks (circular sectors) result in a search space of only 3.5% for railway and 7% for street networks. Surprisingly, the result for the simplest container (bounding box) is about the same as for the better tailored containers (circular sectors, edge-parallel, and smallest enclosing rectangles, or parallelograms). Of course, the results for better tailored containers are better (e.g., smallest enclosing rectangle versus bounding box), but the differences are very small. Furthermore, the difference to our lower bound for convex objects (convex hull) is also comparatively small. The data sets are ordered according to their size. In most cases, the larger the graph, the better the speed-up, which demonstrates the scalability of our methods. This can be explained by the fact 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 as dependent on the size of the graph, because the quality measure is normalized.

Finally, we examined the average running time. We depict them in Figure 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 for answering a query.

4. UPDATING CONTAINERS

A change in the weight of an edge may require some containers to be updated in order to stay consistent. Generally speaking, for every new shortest path

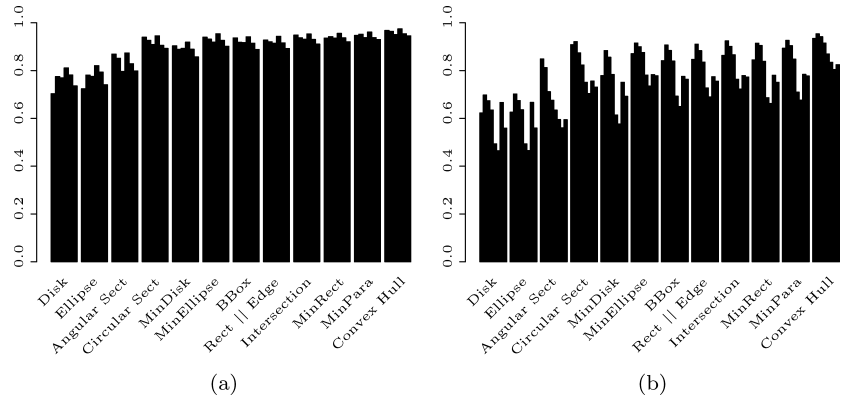


Fig. 4. Quality according to definition 3 for all graphs and geometric objects. The graphs are ordered according to the number of nodes. (a) Railway: Netherlands 409, Austria 1660, Switzerland 2279, Italy 2399, France 4598, Germany 6884. (b) Streets: Moab 1444, Sacramento 3045, New Hampshire 16471, Lodi 25982, Healdsburg 38823, Point Arena 45073, Road77 45852, Polk51510.

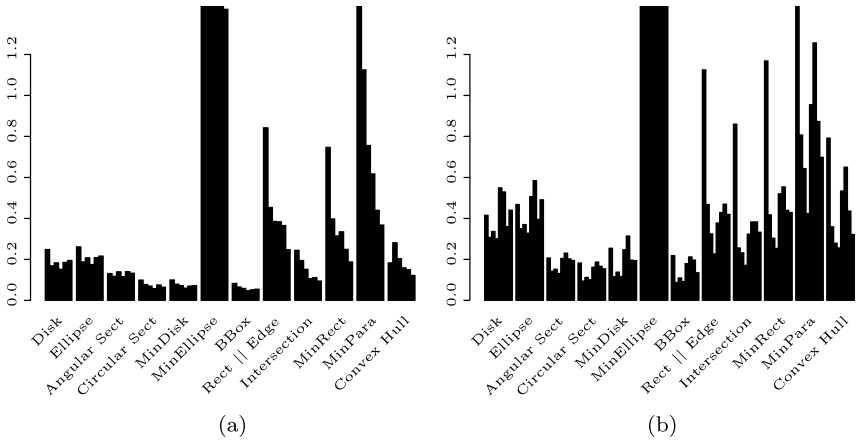


Fig. 5. Average query running time relative to Dijkstra's algorithm for all data sets and geometric objects. The values for minimal ellipse and minimal parallelogram are clipped. They use arbitrary precision and are therefore much slower than the other containment tests. (a) Railway: Netherlands 409, 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.

$u_0, u_1, \ldots, u_k$ created in the graph due to the change, $C(u_0, u_1)$ has to be updated to include u_k. If we maintain containers C^{rev} for reversed edges (e.g., to perform a bidirectional 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 this section, we will mark variables before the update with the subscript "old" and updated values with the subscript "new."

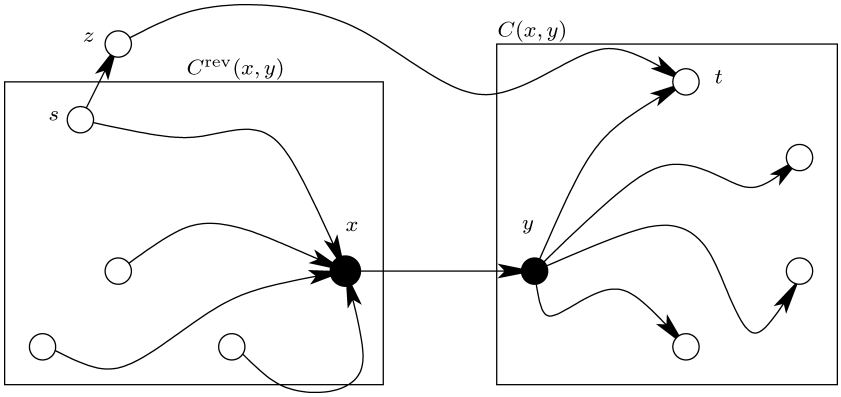


Fig. 6. When the weight of the edge (x, y) is increased, the source s of a new shortest path from s to t must be inside $S_{\text{old}}(x, y)$.

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 allows 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 as illustrated in Figure 6. Since the weight $w(x, y)$ is increased, $(x, y) \in P_{\text{old}}$. Let P_{sy} denote the part of P_{old} from s to y. Since a subpath of a shortest path is again a shortest path, P_{sy} was the shortest path from s to y with (x, y) being its last edge. The proof that (x, y) is the first edge of a shortest x-t-path is symmetric. □

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*

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

Hence, to update the containers, it suffices to search for shortest paths that start from a node in $C^{\text{rev}}_{\text{old}}(x, y)$ and end in a node in $C_{\text{old}}(x, y)$. The next lemma reduces the search space of CREATE-CONTAINERS in order to accomplish this task.

LEMMA 6. *Let $G = (V, E)$, $w : E \to \mathbb{R}$ be a weighted graph and let P_{new} be a path from node s to node t that has become a shortest path because of an increase in the weight of 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 subpath 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, the following lemma helps restricting the set of containers that need updating.

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, (x, y) belongs to P_{new}. Let P_{sy} denote the subpath of P_{new} from s to y. Since a subpath of a shortest path is also a shortest path, P_{sy} is a shortest s-y-path that ends with the edge (x, y). The proof that (x, y) is the first edge of a shortest x-t-path is symmetric. □

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*

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

Hence, contrary to the incremental case, we have to search for new shortest paths that start and end in the updated containers $C^{\text{rev}}_{\text{new}}(x, y)$ and $C_{\text{new}}(x, y)$, respectively, in order to update the rest of the containers.

The updating (i.e., enlargement) of $C_{\text{new}}(x, y)$ 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 $\texttt{new-dist} < w_{\text{old}}(x, y) + d_{\text{old}}(y, u)$. If a node v is excluded, because $\texttt{new-dist} \geq w_{\text{old}}(x, y) + d_{\text{old}}(y, u)$, we distinguish between two cases. If $\texttt{new-dist} = w_{\text{new}}(x, y) + d_{\text{old}}(y, v)$, 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 $\texttt{new-dist} = w_{\text{new}}(x, y) + d_{\text{old}}(y, v)$, therefore, does not change the result of the algorithm. If $\texttt{new-dist} > w_{\text{new}}(x, y) + d_{\text{old}}(y, v)$, there exists a shorter path from x to v that does not contain (u, v). The node v can, therefore, be also ignored in this case.

Similarly to Lemma 6, the next lemma reduces the search space of CREATE-CONTAINERS for the updating of the rest of the containers (see also Figure 7).

LEMMA 9. *Let $G = (V, E)$, $w : E \to \mathbb{R}$ be a weighted graph and let P_{new} be a path from node s to node t that has become a shortest path because of a decrease in the weight of 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

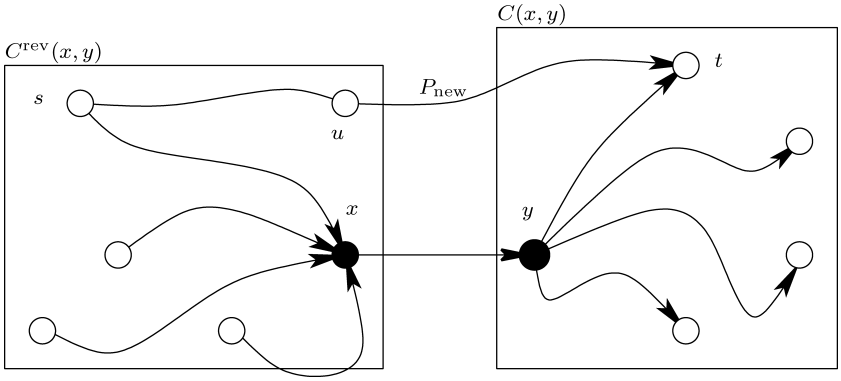


Fig. 7. When the weight of the edge (x, y) is decreased, for all nodes u on a new shortest path from s to t, $d_{\text{new}}(s,u) < d_{\text{new}}(s,x) + w_{\text{old}}(x, y) + d_{\text{new}}(y,u)$.

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 $u \in P_{\text{new}}$ (see Figure 7). Let $P_{s,u}$ denote the subpath 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 Lemmas 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, Corollaries 5 and 8 provide an even simpler method to find containers that may need maintenance. When the weight of an edge (x, y) has been increased, 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 of an edge (x, y) 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. In both cases, we

studied three different methods to update the container of an edge incident to a potentially affected node:

4.3.1 *Recompute the Container.* The natural approach is to recompute the container of an edge incident to a potentially affected node. As Dijkstra's algorithm is run for every potentially affected node, the overall running time is bounded by $O(p \cdot n \log n)$. Note that the result is substantially different from recomputing all containers from scratch, because only the containers of potentially affected nodes are recomputed. Also, a container that can shrink is not necessarily updated.

4.3.2 *Enlarge the Container as Much as Necessary.* A less expensive approach only enlarges a container of an edge incident to a potentially affected node. To enlarge on-line containers, a variant of Dijkstra's algorithm truncated according to Lemmas 6 and 9 can be used. (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 $\texttt{new-dist} < d(s,x) + w_{\text{new}}(x, y) + d(y, v)$ or $\texttt{new-dist} < d(s,x) + w_{\text{old}}(x, y) + d(y, v)$, 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 Lemmas 6 and 9 for edge weight increases and decreases, respectively.

4.3.3 *Set the Container to Infinity (without any further computation).* A "lazy" update approach just sets the container of an edge incident to a potentially affected node to infinity. 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.

The conditions in Lemmas 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. Reverse containers are normal containers when all edges in the graph are reversed. Thus, 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 (on average) is needed to update the containers and how much the containers differ from containers computed from scratch. (Recall that we do not shrink all containers in our updates.)

Since the experiments of the static case showed that bounding boxes are, in practice, the fastest container type, the experiments for the dynamic version are performed with this geometric container. We evaluate the different update strategies only for the railway networks, since the update strategy of computing containers from scratch was very time consuming.

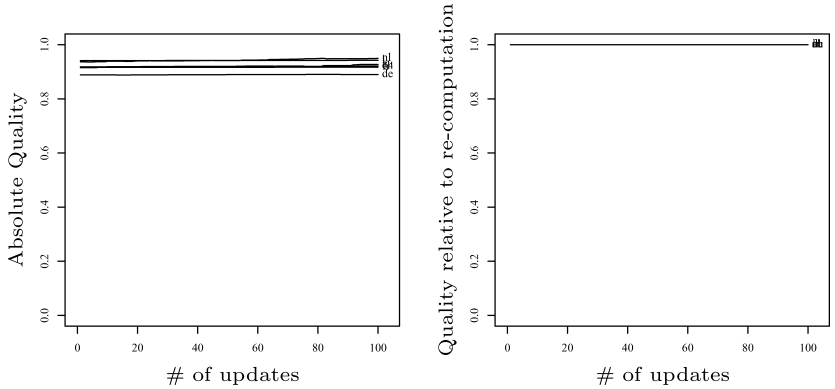


Fig. 8. The quality of updated containers for 100 increased edge weights. The containers are updated for all nodes s with (x, y) as the last edge of a shortest s-y-path. In the right diagram, the quality is divided by the quality of containers computed from scratch.

For each graph, we consider a set of random edges for which the weight is increased 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 the algorithms in 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 the decremental case, we start with the graph where a set of 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. (Details of the implementation framework can be found in the appendix.)

4.5 Computational Results

In order to reflect the fact that the quality of containers computed from scratch varies from graph to graph and after each update, we examined 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. The results for the three update methods applied to edge weight increase for 100 random edges already indicate their general behavior.

We also observed that in the case where only containers of edges incident to potentially affected nodes are recomputed, the resulting containers coincide, most of the time, with the containers that are recomputed for all nodes ("re-computation from scratch"). In other words, the quality equals 1 after almost every update (Figure 8). In practice, such updates can, therefore, be considered as good as using freshly determined containers.

If containers are only enlarged, their quality decreases most of the time, as expected (Figure 9). Empirically, the clear trend appears to follow a linear

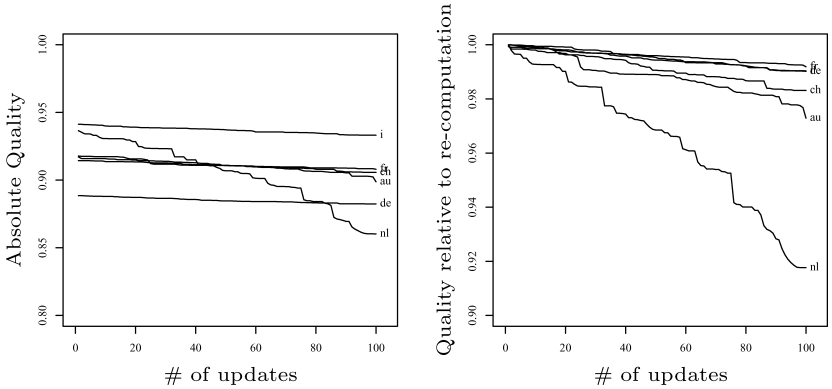


Fig. 9. The quality of updated containers for 100 increased edge weights. The containers are enlarged by a truncated Dijkstra for all nodes s with (x, y) as the last edge of a shortest s-y-path. In the right diagram, the quality is divided by the quality of containers computed from scratch.

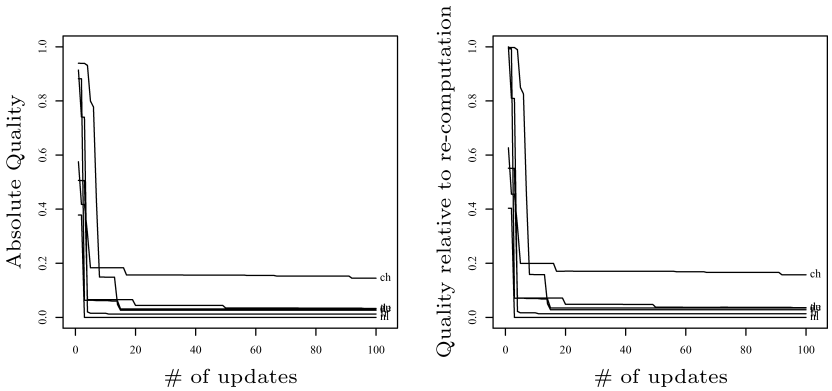


Fig. 10. The quality of updated containers for 100 increased edge weights. The containers are set to infinity for all nodes s with (x, y) as the last edge of a shortest s-y-path. In the right diagram, the quality is divided by the quality of containers computed from scratch.

function in the beginning, which will lead to fairly bad containers in a long run. It is, however, interesting to note that the larger the graph, the slower the quality decreases. Actually, for large graphs the quality is still close to 1 after 100 updates. The reason is that single "bad" containers are clearly less important, if the graph contains more edges.

If the containers are simply set to infinity, the situation is dramatically different (Figure 10). We have already observed that 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.

The analysis of the time measurements are shown in Table I. 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

Table I. Average Speed-up for Updating the Containers with Increasing Weights (First Number) and Decreasing Weights (Second Number)

	Reverse cont.		Used			Not used		
	Cont. Update		Set Affected to Infinity	Enlarge Affected	Recomp. Affected	Set Affected to Infinity	Enlarge Affected	Recomp. Affected
	n	m						
nl	409	1215	1331; 173	2.2; 2.0	2.5; 2.7	269; 228	2.5; 3.4	5.0; 3.0
au	1660	4327	8093; 700	2.0; 1.9	2.5; 2.4	1551; 1429	2.1; 2.1	2.1; 2.1
ch	2279	6015	10211; 953	2.2; 2.2	2.6; 2.6	2235; 2300	2.8; 2.9	2.5; 3.1
i	2399	8008	9552; 965	2.9; 2.8	2.8; 2.7	2095; 2097	2.8; 2.7	2.6; 2.9
fr	4598	14937	17691; 1932	2.1; 2.2	2.9; 2.7	4382; 4291	2.6; 3.1	4.3; 3.4
de	6884	18601	33160; 2974	1.7; 1.7	2.0; 2.2	6568; 6583	2.5; 2.3	2.6; 2.8

[a] Data for railway networks of Netherlands (nl), Austria (au), Switzerland (ch), Italy (i), France (fr), and Germany (de).

required for computing all 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 Lemmas 6 and 9, and recompute the containers of every potentially affected node) were tested with and without maintenance of reverse containers (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, which demonstrates the scalability of our methods. 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 affected") is often slower than Dijkstra without pruning (column "recomputed affected"). Obviously, the additional check and computation of distances to x and from y for all nodes outweigh the gain of the pruning. As almost perfect containers ("recompute") can be computed in the same amount of time, we conclude that, for practical purposes, only enlarging the containers does not pay off.

5. CONCLUSIONS

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 30%. The somewhat surprising result is that the simple bounding box outperforms other geometric objects in terms of CPU cycles in many 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 Meyer [2001] and Goldberg [2001] 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 [Sedgewick and Vitter 1986] or A^* has been shown in Shekhar et al. [1993] and Jacob et al. [1998] 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 [Pohl 1971] can be integrated by reversing all edges and running the preprocessing for a second time. Space and time consumption for the preprocessing simply doubles.
- In combination with a multilevel approach [Jung and Pramanik 2002; Schulz et al. 2002], one first constructs a graph containing all levels and interlevel edges. The geometric pruning is then performed on this graph.

A detailed study for combinations with the latter three speed-up techniques can be found in Holzer et al. [2004].

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 to 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 decreases in a single algorithm.

APPENDIX:

IMPLEMENTATION

When one implements several variants of an algorithm and wishes to experimentally compare their performance, a major goal is to maintain a small and efficient common code base that allows to switch flexibly between the different variants. In our case, a basic algorithm (Dijkstra's algorithm) is refined in different ways to improve its performance; for example, the shortest-path computation with Dijkstra's algorithm can be improved by goal-directed search or geometric pruning. Moreover, we would like to introduce new aspects to the implementations for experimental analysis or other purposes, for instance, operation counting, time measurement, and debugging output. Adding functionality to graph algorithms can be achieved by the design pattern *template method* [Gamma et al. 1995] or by an extension of the design pattern *visitor*, which is the approach followed by the BOOST graph library [Siek et al. 2002]. Our framework 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., Spinczyk et al. 2002]. In particular, it is necessary to change the inheritance hierarchy to create arbitrary combinations of aspects. One way to support aspect-oriented programming in C++ is by extending the language as proposed in Spinczyk et al. [2002]. However, such an extension can be avoided through the use of parameterized inheritance (also known as mix-in classes [Bracha and Cook 1990]) and template meta-programming [Alexandrescu 2001] that provides the base to a solution with standard C++ compilers.

A.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.

```
struct AspectA::public Algo {
  int operations;

  virtual void init() {
    Algo::init();
    operations=0;
  }

  virtual void do_something() {
    Algo::do_something();
    ++operations;
  }
};
```

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

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 CREATECONTAINERS, or truncating Dijkstra's algorithm according to Lemmas 6 or 9 for updates after changing an edge weight.

A.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 of how additional parameters that are needed by aspects can be provided. We use a technique inspired by Eisenecker et al. [2000], 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 [Alexandrescu 2001].

```
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.

A.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 both depend on a layout of the graph. Template meta-programming can also be used to check dependencies of such concepts [Siek et al. 2002]. 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 yet been included. (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 Alexandrescu [2001]:

```
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 Alexandrescu [2001] for additional 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 is 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.

REFERENCES

ALEXANDRESCU, A. 2001. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, Reading, MA.

AUSIELLO, G., ITALIANO, G. F., MARCHETTI-SPACCAMELA, A., AND NANNI, U. 1991. Incremental algorithms for minimal length paths. *Journal of Algorithms 12*, 615–638.

BARRETT, C., JACOB, R., AND MARATHE, M. 2000. Formal-language-constrained path problems. *SIAM Journal on Computing 30*, 3 (June), 809–837.

BARRETT, C., BISSET, K., JACOB, R., KONJEVOD, G., AND MARATHE, M. 2002. Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router. In *Proc. 10th European Symposium on Algorithms (ESA 2002)*, R. Möhring and R. Raman, Eds. LNCS, vol. 2461. Springer, New York. 126–138.

BRACHA, G. AND COOK, W. 1990. Mixin-based inheritance. In *Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications/Proc. European Conference on Object-Oriented Programming*, N. Meyrowitz, Ed. ACM Press, New York. 303–311.

BRANDES, U., EIGLSPERGER, M., HERMAN, I., HIMSOLT, M., AND SCOTT, M. 2001. GraphML progress report. P. Mutzel, M. Jünger, and S. Leipert, Eds. LNCS, vol. 2265. Springer, New York. 501–512.

DEMETRESCU, C. AND ITALIANO, G. F. 2003. A new approach to dynamic all pairs shortest paths. In *Proc. 35th ACM Symposium on Theory of Computing (STOC 2003)*. ACM Press, New York. 159–166.

DIJKSTRA, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 269–271.

EISENECKER, U. W., BLINN, F., AND CZARNECKI, K. 2000. A solution to the constructor-problem of mixin-based programming in C++. In *Proc. 1st Workshop on C++ Template Programming*. Erfurt, Germany.

EVEN, S. AND GAZIT, H. 1985. Updating distances in dynamic graphs. *Methods of Operations Research 49*, 371–387.

FABRI, A., GIEZEMAN, G.-J., KETTNER, L., SCHIRRA, S., AND SCHÖNHERR, S. 2000. On the design of CGAL a computational geometry algorithms library. *Softw.—Pract. Exp. 30*, 11, 1167–1202.

FREDMAN, M. L. AND TARJAN, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM) 34*, 3, 596–615.

FRIGIONI, D. 1998. Semidynamic algorithms for maintaining single-source shortest path trees. *Algorithmica 22*, 3, 250–274.

FRIGIONI, D., MARCHETTI-SPACCAMELA, A., AND NANNI, U. 1996. Fully dynamic output bounded single source shortest path problem. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'96)*. 212–221.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA.

GOLDBERG, A. V. 2001. Shortest path algorithms: Engineering aspects. In *Proc. International Symposium on Algorithms and Computation (ISAAC 2001)*, P. Eades and T. Takaoka, Eds. LNCS, vol. 2223. Springer, New York. 502–513.

HOLZER, M., SCHULZ, F., AND WILLHALM, T. 2004. Combining speed-up techniques for shortest-path computations. In *Experimental and Efficient Algorithms: Third International Workshop, (WEA 2004)*, C. C. Ribeiro and S. L. Martins, Eds. LNCS, vol. 3059. Springer, New York. 269–284.

JACOB, R., MARATHE, M., AND NAGEL, K. 1998. A computational study of routing algorithms for realistic transportation networks. In *Proc. 2nd Workshop on Algorithm Engineering (WAE'98)*, K. Mehlhorn, Ed. 167–178. Available at: http://www.mpi-sb.mpg.de/~wae98/PROCEEDINGS/.

JOHNSON, D. B. 1977. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM) 24*, 1, 1–13.

JUNG, S. AND PRAMANIK, S. 2002. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering 14*, 5, 1029–1046.

KING, V. 1999. 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)*. 81–91.

MEHLHORN, K. AND NÄHER, S. 1999. *LEDA, A platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge.

MEYER, U. 2001. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Proc. Symposium on Discrete Algorithms*. 797–806.

NACHTIGALL, K. 1995. Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research 83*, 1, 154–166.

PETTIE, S., RAMACHANDRAN, V., AND SRIDHAR, S. 2002. Experimental evaluation of a new shortest path algorithm. In *Proc. Algorithm Engineering and Experiments (ALENEX'02)*. LNCS, vol. 2409. Springer, New York. 126–142.

POHL, I. 1971. Bi-directional search. In *Proc. 6th Annual Machine Intelligence Workshop*, B. Meltzer and D. Michie, Eds. Machine Intelligence, vol. 6. Edinburgh University Press, London, 137–140.

PREUSS, T. AND SYRBE, J.-H. 1997. An integrated traffic information system. In *Proc. 6th Int. Conf. Appl. Computer Networking in Architecture, Construction, Design, Civil Eng., and Urban Planning (europIA '97)*.

RAMALINGAM, G. AND REPS, T. W. 1996a. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms 21*, 2, 267–305.

RAMALINGAM, G. AND REPS, T. W. 1996b. On the computational complexity of dynamic graph problems. *Theoretical Computer Science 158*, 233–277.

ROHNERT, H. 1985. A dynamization of the all pairs least cost path problem. In *Proc. Symp. Theoretical Aspects of Computer Science (STACS'85)*. LNCS, vol. 182. Springer, New York. 279–286.

SCHULZ, F., WAGNER, D., AND WEIHE, K. 2000. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics 5*, 12.

SCHULZ, F., WAGNER, D., AND ZAROLIAGIS, C. 2002. Using multi-level graphs for timetable information. In *Proc. Algorithm Engineering and Experiments (ALENEX'02)*. LNCS, vol. 2409. Springer, New York. 43–59.

SCHWARZ, C., TEICH, J., VAINSHTEIN, A., WELZL, E., AND EVANS, B. L. 1995. Minimal enclosing parallelogram with application. In *Proc. 11th Annual Symposium on Computational Geometry*. ACM Press, New York. 434–435.

SEDGEWICK, R. AND VITTER, J. S. 1986. Shortest paths in Euclidean space. *Algorithmica 1*, 1, 31–48.

SHEKHAR, S., FETTERER, A., AND GOYAL, B. 1997. Materialization trade-offs in hierarchical shortest path algorithms. In *Proc. Symposium on Large Spatial Databases*. 94–111.

SHEKHAR, S., KOHLI, A., AND COYLE, M. 1993. Path computation algorithms for advanced traveler information system (ATIS). In *Proc. 9th IEEE Int. Conf. Data Eng.* 31–39.

SIEK, J., LEE, L.-Q., AND LUMSDAINE, A. 2002. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, Reading, MA.

SPINCZYK, O., GAL, A., AND SCHRODER-PREIKSCHAT, W. 2002. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proc. 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, J. Noble and J. Potter, Eds. Conferences in Research and Practice in Information Technology, vol. 10. ACS, Sydney, Australia. 53–60.

THORUP, M. 1997. Undirected single source shortest path in linear time. In *Proc. IEEE Symposium on Foundations of Computer Science (FOCS'97)*. 12–21.

TOUSSAINT, G. 1983. Solving geometric problems with the rotating calipers. In *Proc. IEEE Mediteranian Electrotechnical Conference (MELECON 1983)*, E. N. Protonotarios, Ed. IEEE, New York. A10.02/1-4.

WAGNER, D. AND WILLHALM, T. 2003. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *Proc. 11th European Symposium on Algorithms (ESA 2003)*, G. D. Battista and U. Zwick, Eds. LNCS, vol. 2832. Springer, New York. 776–787.

WAGNER, D. AND WILLHALM, T. 2005. Drawing graphs to speed up shortest-path computations. In *Proc. 7th Workshop Algorithm Engineering and Experiments (ALENEX'05)*. SIAM. To appear.

WAGNER, D., WILLHALM, T., AND ZAROLIAGIS, C. 2004. Dynamic shortest path containers. In *Proc. Algorithmic MeThods and Models for Optimization of RailwayS (ATMOS 2003)*, A. Marchetti-Spaccamela, Ed. Electronic Notes in Theoretical Computer Science, vol. 92. 65–84.

WELZL, E. 1991. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, H. Maurer, Ed. LNCS, vol. 555. Springer, New York.

WILCOX, R. R. 2001. *Fundamentals of modern statistical methods: substantially improving power and accuracy*. Springer, New York.

ZAROLIAGIS, C. 2002. Implementations and experimental studies of dynamic graph algorithms. In *Experimental Algorithmics*, R. Fleischer, B. Moret, and E. M. Schmidt, Eds. LNCS, vol. 2547. Springer, New York. 229–278.

ZHAN, F. B. AND NOON, C. E. 2000. 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.

Submitted August 17, 2004; revised August 4, 2005; accepted October 13, 2005.