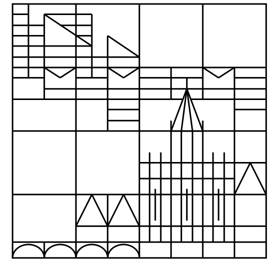


Universität Konstanz

Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs

Dorothea Wagner
Thomas Willhalm

Konstanzer Schriften in Mathematik und Informatik
Nr. 183, Januar 2003

ISSN 1430–3558

© Fachbereich Mathematik und Statistik
© Fachbereich Informatik und Informationswissenschaft
Universität Konstanz
Fach D 188, 78457 Konstanz, Germany
Email: preprints@informatik.uni–konstanz.de
WWW: http://www.informatik.uni–konstanz.de/Schriften/

Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs

D. Wagner and T. Willhalm*
Universität Konstanz, Fach D188, 78457 Konstanz

January 23, 2003

Abstract

In this paper, we consider Dijkstra's algorithm for the single source single target shortest paths problem in large sparse graphs. The goal is to reduce the response time for online queries by using precomputed information. For the result of the preprocessing, we admit at most linear space. We assume that a layout of the graph is given. From this layout, in the preprocessing, we determine for each edge a geometric object containing all nodes that can be reached on a shortest path starting with that edge. Based on these geometric objects, the search space for online computation can be reduced significantly. We present an extensive experimental study comparing the impact of different types of objects. The test data we use are traffic networks, the typical field of application for this scenario.

1 Introduction

In this paper, we consider the problem to answer a lot of single-source single-target shortest-path queries in a large graph. We admit an expensive preprocessing in order to speed up the query time. From a given layout of the graph we extract geometric information that can be used in the online computation to answer the queries. (See also [17, 18].) A typical application of this problem is a route planning system for cars, bikes and hikers or scheduled vehicles like trains and busses. Usually, variants of Dijkstra's algorithm are used to realize such systems.

In the comparison model, Dijkstra's algorithm [3] with Fibonacci heaps [5] is still the fastest known algorithm for the general case of arbitrary non-negative edge lengths. Algorithms with worst case linear time are known for undirected graphs and integer weights [23]. Algorithms with average case linear time are known for edge weights uniformly distributed in the interval $[0, 1]$ [11] and for edge weights uniformly distributed in $\{1, \ldots, M\}$ [6]. A recent study [14] shows that the sorting bottleneck can be also avoided in practice for undirected graphs.

The application of shortest path computations in travel networks is also widely covered by the literature: [26] compares different algorithms to compute shortest paths trees in real road networks. In [27] the two best label setting and label correcting algorithms were compared in respect to single source single target shortest path computations. They conclude that for shorter paths a label correcting algorithm should be preferred. In [1] Barrett et al. present a system that covers formal language constraints and changes over time. Modeling an (interactive) travel information system for scheduled transport is covered by [22] and [13], and a multi-criteria implementation is presented in [12]. A distributed system which integrates multiple transport providers has been realized in [16].

One of the features of travel planning (independant of the vehicle type) is the fact that the network does not change 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. Although

*The authors acknowledge financial support from EU for the RTN AMORE, contract no HPRN-CT-1999-00104 and from DFG under grant WA 654/12-1.

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 10^5 and more nodes prevents us from doing so.

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. In fact, this papers shows that storing partial results reduces the number of nodes visited by Dijkstra's algorithm to 10%. 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 ignored safely in any shortest path computation to this target. More precisely, we apply the following concept:

- In the preprocessing, for each edge e, the set of nodes $S(e)$ is stored that can be reached by a shortest path starting with e.
- While running Dijkstra's algorithm, 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* $S(e)$. Remark that this 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 for example 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 even not the case.

In [17] angular sectors were introduced for the special case of a time table information system. Our results are more general in two respects: We examine the impact of various different geometric objects and consider Dijkstra for general embedded graphs. 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 next section contains – after some definitions – a formal description of our shortest path problem. Section 3 gives precise arguments how and why the pruning of edges works. In Section 4, we describe the geometric objects and the test graphs that we used in our experiments. The statistics and results are presented and discussed in the last section before the summary.

2 Definitions and Problem Description

2.1 Graphs

A directed *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*. The edges of a graph 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.

If n denotes the number of nodes, a graph can have up to n^2 edges. We call a graph *sparse*, if the number of edges m is in $o(n^2)$, and we call a graph *large*, if one can only afford a memory consumption that is linear in the size of the graph, i.e. in $O(n+m)$. In particular for large sparse graphs, n^2 space is not affordable.

2.2 Shortest Path Problem

Given a weighted graph $G = (V, E)$, $w : E \to \mathbb{R}$, the *single source single target shortest paths problem* consists in finding a shortest path 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. Using Johnson's algorithm [8] it is possible to convert in $O(nm \log n)$ time the edge weights $w : E \to \mathbb{R}$

```
1   insert source s in priority queue Q and set dist(s) := 0
2   while target t is not marked as finished and priority queue is not empty
3       get node u with highest priority in Q and mark it as finished
4       for all neighbor nodes v of u
5               set new-dist := dist(u) + w((u, v))
6               if neighbor node v is unvisited
7                   set dist(v) := new-dist
8                   insert neighbor node v in priority queue with priority −dist(v)
9                   mark neighbor node v as visited
10              else
11                  if dist(v) > new-dist
12                      set dist(v) := new-dist
13                      increase priority of neighbor node to −dist(v)
```

Algorithm 1: DIJKSTRA'S ALGORITHM

to non-negative edge weights $w' : E \to \mathbb{R}$ that result in the same shortest paths. We will therefore assume in the rest of this paper, that edge weights are non-negative.

A closer look at DIJKSTRA'S ALGORITHM, see Algorithm 1, reveals that one needs to visit all nodes of the graph to initialize the marker "visited", which obviously prevents a sub-linear query time. This shortcoming can be eliminated by introducing a global integer variable "time" and replacing the marker "visited" by a time stamp for every node. These time stamps are set to zero in the preprocessing. The time variable is incremented in each run of the algorithm and, instead of marking a node as visited, the time stamp of the node is set to the current time. The test whether a node has been visited in the current run of the algorithm can then be replaced by a comparison of the time stamp with the current time. (For sake of clarity however, we didn't include this technique in the pseudo-code.)

3 Geometric Pruning

3.1 Pruning the Search Space

The goal of DIJKSTRA'S ALGORITHM WITH PRUNING, see Algorithm 2, is to decrease the number of visited nodes, the "search space", by visiting only a subset of the neighbors (line 4a). The idea is illustrated in Figure 1. The condition is formalized by the notion of a consistent container:

Definition 1 *Given a weighted graph $G = (V, E)$, $w : E \to \mathbb{R}_0^+$, 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 $S(u, v) \subseteq C(u, v)$.

Theorem 2 *Given a weighted graph $G = (V, E), w : E \to \mathbb{R}_0^+$ and for each edge e a consistent container $C(e)$, 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 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, $t \in C(u, v)$. □

1	insert source s in priority queue Q and set $\texttt{dist}(s) := 0$
2	while target t is not marked as *finished* and priority queue is not empty
3	get node u with highest priority in Q and mark it as *finished*
4	for all neighbor nodes v of u
4a	if $t \in C(u, v)$
5	set $\texttt{new-dist} := \texttt{dist}(u) + w((u, v))$
6	if neighbor node v is *unvisited*
7	set $\texttt{dist}(v) := \texttt{new-dist}$
8	insert neighbor node v in priority queue with priority $-\texttt{dist}(v)$
9	mark neighbor node v as *visited*
10	else
11	if $\texttt{dist}(v) > \texttt{new-dist}$
12	set $\texttt{dist}(v) := \texttt{new-dist}$
13	increase priority of neighbor node to $-\texttt{dist}(v)$

Algorithm 2: DIJKSTRA'S ALGORITHM WITH PRUNING. Neighbors are only visited, if the edge (u, v) is in the consistent container $C(u, v)$.

PSfrag replacements

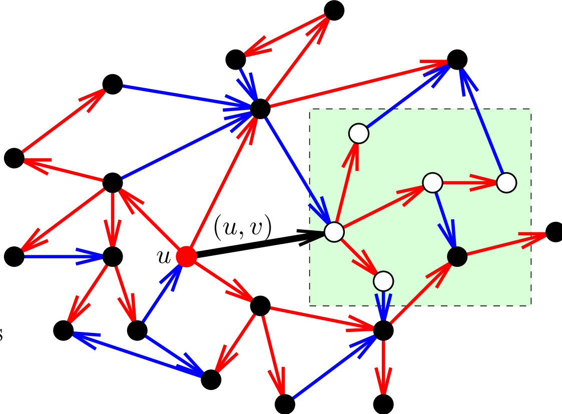


Figure 1: DIJKSTRA'S ALGORITHM is run for a node $u \in V$. Let the white nodes be those nodes that can be reached on a shortest path using the edge (u, v). A geometric object is constructed that contains these nodes. It may contain other points, but this does only affect the running time and not the correctness of DIJKSTRA'S ALGORITHM WITH PRUNING.

3.2 Geometric Containers

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

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. We require that a container has a description of constant size and that its containment test takes constant time.

Recall that $S(u, v)$ is the set of all nodes t with the property that there is a shortest u-t-path starts with the edge (u, v). For some types of containers we need to know $S(u, v)$ explicitly in order

to determine the container $C(u, v)$ associated with the edge (u, v). Therefore we now detail how to compute $S(u, v)$. To determine the sets $S(u, v)$ for every edge $(u, v) \in E$, Dijkstra's algorithm is run for each node u. For each node $t \in V$, we store the edge (u, v) with $t \in S(u, v)$, i.e. the first edge of the u-t-path in the shortest paths tree. This can be done in a similar way one constructs the shortest path tree: In the beginning, all nodes v adjacent to u are associated with the edge (u, v) in an array $A[v]$. Every time the distance label of a node t is adjusted via (p, t), we update $A[t]$ with $A[p]$, i.e. we associate with t the edge of its predecessor p. When a node is marked as *finished*, $A[t]$ holds the outgoing edge of u with which a shortest path from u to t starts. We can then construct the sets $S(u, v)$ for all edges incident to u. The storage requirement is linear in the number of nodes and the geometric containers can then be easily constructed and attached to the edges.

On the other hand, it is possible for some geometric containers to be constructed without actually creating the sets $S(u, v)$ in memory. These containers can be constructed *online* by adding one point after the other without storing all points explicitly. In other words, there exists an efficient method to update a container $C(u, v)$ with a new point t that has turned out to lie in $S(u, v)$.

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[1] which leads to a space consumption linear in the number of edges. The radius can be determined online by simply 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 online similarly as in the disk case by simply increasing it if necessary.

Angular Sector. Angular sectors are the objects that were used in [17]. 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 online fashion: If a new point w is outside the angular sector $\angle(p, u, q)$, we set $p := w$ if w is left of (u, v) and $q := w$ if w is right of it. (Note that this is not necessarily the minimum angle at u that contains all points in $S(u, v)$.)

Circular Sector. Angular sectors have the big disadvantage that they are not bounded. 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 online 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 [4] of Welzl's algorithm [25] with expected linear running time. The algorithm works offline 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 online while the shortest paths are computed in the preprocessing.

[1] Of course in practice, the squared radius is stored to avoid the computationally expensive square root function.

Edge-Parallel Rectangle. Such a rectangle is not parallel to an axis, but to the edge to which it belongs. So, 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. 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 sum up, but are still constant, and as both objects can be computed online the intersection can be as well.

Smallest Enclosing Rectangle. If we allow a rectangle to be oriented in any direction and search for one with smallest area, things are not as simple anymore. The algorithm from [24] finds it 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 offline, but needs only constant space for its orientation and dimensions.

Smallest Enclosing Parallelogram. Toussaint's idea to use rotating calipers has been extended in [19] to find the smallest enclosing parallelogram. Space consumption is constant and the algorithm is offline.

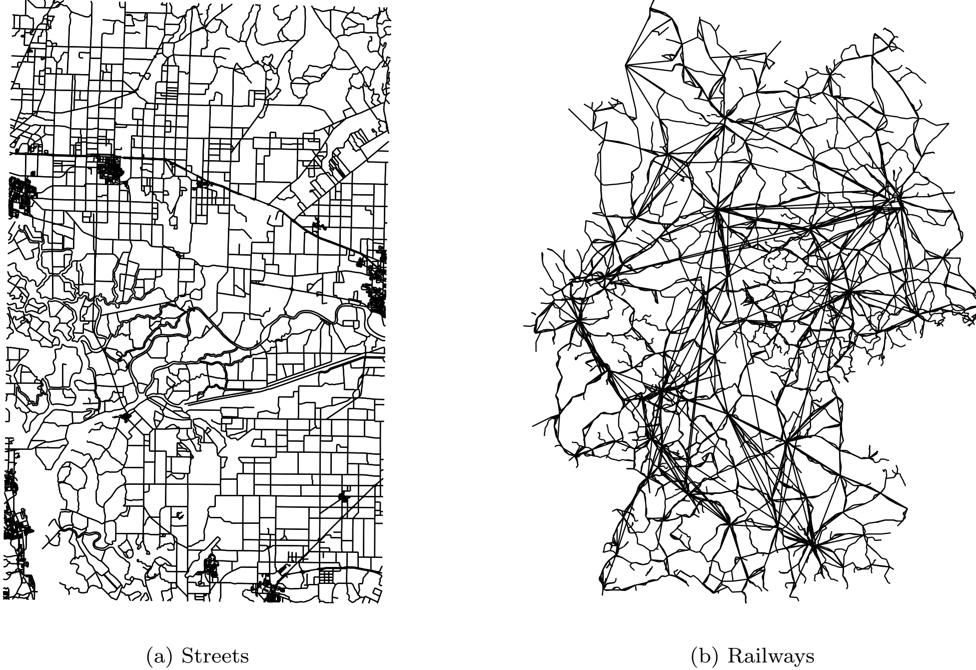
Convex Hull. The convex hull does not fulfill our requirement that containers must be of constant size. It is included here, because it provides a lower bound for all convex objects. If there is a best convex container, it cannot exclude more points than the convex hull.

4 Implementation and Experiments

We implemented the algorithm in C++ using the GNU compiler g++ 2.95.3. We used the graph data structure from LEDA 4.3 (see [10]) 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. 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. We computed the convex hulls with LEDA [10]. 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. A typical example is depicted in Figure 2(a). 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.



(a) Streets

(b) Railways

Figure 2: Samples from our test data

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 (see Appendix A).

Railway Networks. The railway networks of different European countries (see Figure 2(b)) 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.

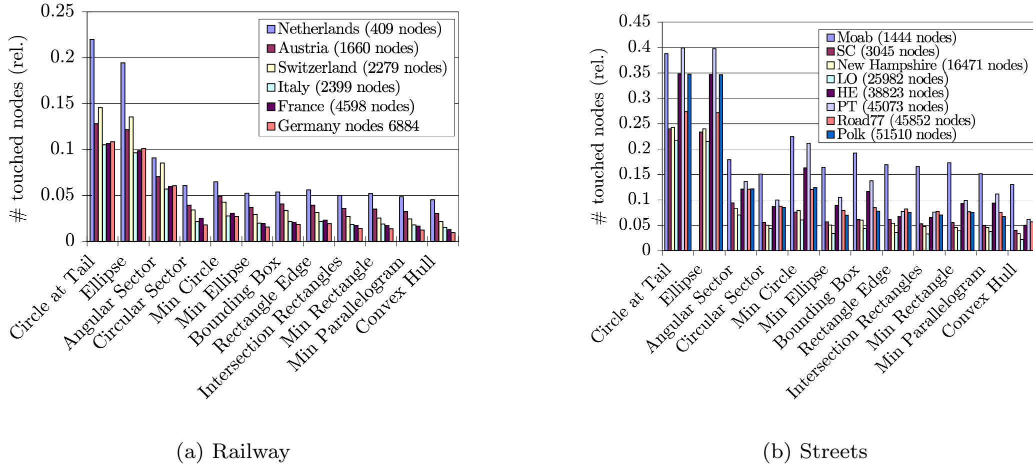
All test sets were converted to the XML-based GraphML file format [2] to allow us a unified processing.

We sampled random single source single target queries to determine the average number of nodes that are visited by the algorithm. The sampling was done until the length of the 95% confidence interval was smaller than 5% of the average search space. The length of the confidence interval is $2t_{n-1,1-\frac{\alpha}{2}}sn^{-\frac{1}{2}}$, where $t_{n-1,1-\frac{\alpha}{2}}$ denotes the upper critical error for the t-distribution with $n-1$ degrees of freedom. 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:

$$\overline{x}_{(k)} = \frac{1}{k}\left(\overline{x}_{(k-1)}(k-1) + x_k\right)$$

$$s^2_{(k)} = \frac{1}{k-1}\left[(k-2)s^2_{(k-1)} + (k-1)\overline{x}^2_{(k-1)} + x_k^2 - k\overline{x}^2_{(k)}\right]$$

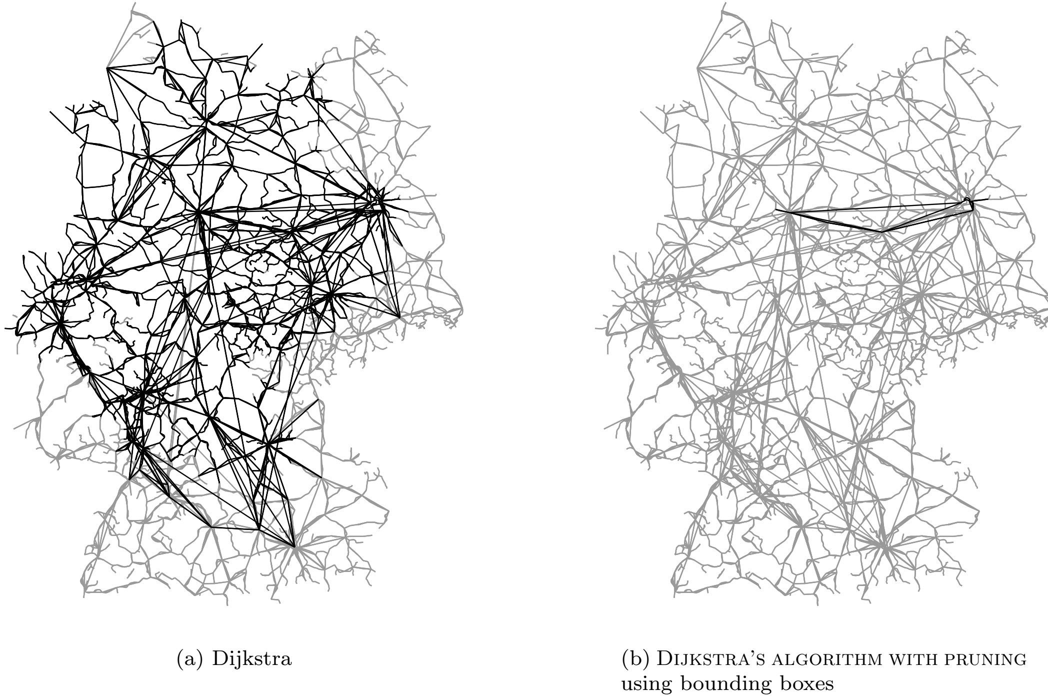
where the subscript (k) marks the mean value and standard error for k samples, respectively. Using these formulas, it is possible to run random single-source single-target shortest-path queries until the confidence interval is small enough.



(a) Railway

(b) Streets

Figure 3: Relative average search space.



(a) Dijkstra

(b) Dijkstra's algorithm with pruning using bounding boxes

Figure 4: The search space for a query from Hannover to Berlin

5 Results and Discussion

Figure 3(a) depicts the results for railway networks. (For numbers, we refer the reader to Appendix A.) The average number of nodes are shown that are visited by the algorithm. 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 pruning for disks around

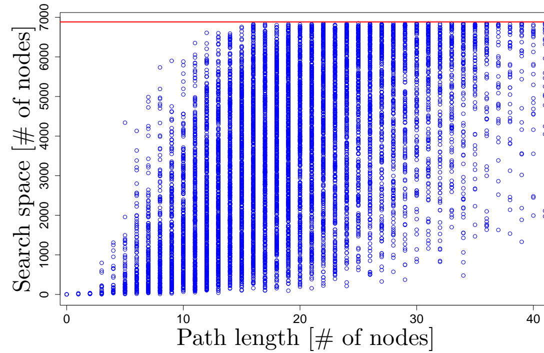


Figure 5: The size of the search space for the German railway network. For each of 10000 queries, a circle is drawn positioned according to the path length and the size of the search space. The horizontal dotted line marks the total number of nodes in the graph (6884).

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 [17], 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. Figure 4 gives an illustrative example.

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 the lower bound for convex objects, the convex hull, 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. Figure 5 compares the number of visited nodes to the length of the shortest paths (in number of nodes). For each of 10000 queries, a circle is drawn with respective coordinates.

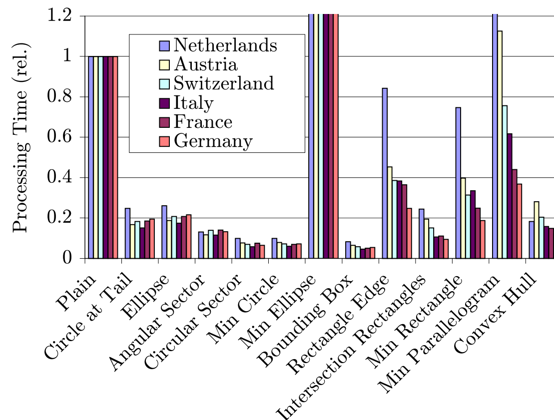


Figure 6: Average query running time. The values for minimal ellipse and minimal parallelogram are clipped. They use arbitrary precision and are therefore much slower than the other containment tests.

Apart from the number of visited nodes, we examined the average running time. We depict them in Figure 6, 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.

6 Conclusion

We have seen that using a layout may lead to a considerably speed-up, if one allows a preprocessing. Actually, we are able to reduce the search space to $5 - 10\%$ 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 in fact independent of the priority queue. Algorithms using a special priority queue such as [11, 6] 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 of course).
- Goal-directed search [20] or A^* has been shown in [21, 7] to be very useful for transportation networks. As it simply modifies the edge weights, a combination of geometric pruning and A^* can be realized straight forward.
- Bidirectional search [15] 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 [9, 18], one constructs a graph containing all levels and inter-level edges first. The geometric pruning is then performed on this graph.

Acknowledgment

We thank Jasper Möller for his help in implementing and Alexander Wolff for many helpful comments regarding the presentation of this paper.

References

[1] C. Barrett, K. Bisset, R. Jacob, G. Konjevod, and M. Marathe. Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the transims router. In *ESA 2002*, (eds.) R. Möhring and R. Raman, vol. 2461 of *LNCS*, pp. 126–138 (Springer), 2002.

[2] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Scott. GraphML progress report. In *GD 2001*, (eds.) P. Mutzel, M. Jünger, and S. Leipert, vol. 2265 of *LNCS*, pp. 501–512 (Springer), 2001.

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

[4] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. – Pract. Exp.*, 30(11): pp. 1167–1202, 2000.

[5] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3): pp. 596–615, 1987.

[6] A. V. Goldberg. A simple shortest path algorithm with linear average time. In *ESA 2001*, (ed.) F. M. auf der Heide, vol. 2161 of *LNCS*, pp. 230–241 (Springer), 2001.

[7] R. Jacob, M. Marathe, and K. Nagel. A computational study of routing algorithms for realistic transportation networks. In *WAE'98*, (ed.) K. Mehlhorn. 1998.

[8] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1): pp. 1–13, 1977.

[9] S. Jung and S. Pramanik. Hiti graph model of topographical road maps in navigation systems. In *Proc. 12th IEEE Int. Conf. Data Eng.*, pp. 76–84. 1996.

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

[11] U. Meyer. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Symposium on Discrete Algorithms*, pp. 797–806. 2001.

[12] M. Müller-Hannemann and K. Weihe. Pareto shortest paths is often feasible in practice. In *WAE 2001*, (ed.) A. M.-S. G.S. Brodal, D. Frigioni, vol. 2461 of *LNCS*, pp. 185–197 (Springer), 2001.

[13] K. Nachtigall. Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research*, 83(1): pp. 154–166, 1995.

[14] S. Pettie, V. Ramachandran, and S. Sridhar. Experimental evaluation of a new shortest path algorithm. In *ALENEX'02*, pp. 126–142. 2002.

[15] I. Pohl. Bi-directional search. In *Sixth Annual Machine Intelligence Workshop*, (eds.) B. Meltzer and D. Michie, vol. 6 of *Machine Intelligence*, pp. 137–140 (Edinburgh University Press), 1971.

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

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

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

[19] C. Schwarz, J. Teich, A. Vainshtein, E. Welzl, and B. L. Evans. Minimal enclosing parallelogram with application. In *Proceedings of the eleventh annual symposium on Computational geometry*, pp. 434–435 (ACM Press), 1995.

[20] R. Sedgewick and J. S. Vitter. Shortest paths in euclidean space. *Algorithmica*, 1(1): pp. 31–48, 1986.

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

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

[23] M. Thorup. Undirected single source shortest path in linear time. In *IEEE Symposium on Foundations of Computer Science*, pp. 12–21. 1997.

[24] G. Toussaint. Solving geometric problems with the rotating calipers. In *MELECON 1983*, (ed.) E. N. Protonotarios, pp. A10.02/1–4 (IEEE, NY), 1983.

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

[26] F. B. Zahn and C. E. Noon. Shortest path algorithms: An evaluation using real road networks. *Transportation Science*, 32(1): pp. 65–73, 1998.

[27] F. B. Zahn and C. E. Noon. A comparison between label-setting and label-correcting algorithms for computing one-to-one shortest paths. *Journal of Geographic Information and Decision Analysis*, 4(2), 2000.

A Average Search Space and Running Time for all Instances

		Plain	Circle at Tail	Ellipse	Angular Sector	Circular Sector	Min Circle	Min Ellipse	Bounding Box	Rectangle Edge	Intersection Rectangles	Min Rectangle	Min Parallelogram	Convex Hull
Netherlands nodes	409	227.5	50.0	44.2	20.6	13.8	14.7	11.9	12.2	12.7	11.4	11.8	11.0	10.2
edges	1215	646.5	198.9	179.5	91.1	68.1	78.9	67.6	69.2	71.0	65.9	67.3	64.5	60.6
time [μs]		289.1	71.6	75.4	37.7	28.5	28.7	3248.2	23.9	243.6	70.6	215.8	846.8	52.8
Austria nodes	1660	814.9	104.5	99.1	57.5	32.2	40.0	30.2	33.1	32.1	29.2	28.6	26.3	24.7
edges	4327	2108.0	394.0	373.6	217.8	136.6	170.4	132.1	143.4	139.7	127.6	125.3	118.2	109.3
time [μs]		1082.4	181.9	202.5	126.3	83.4	85.0	5273.0	70.4	491.0	210.4	430.5	1218.2	303.8
Switzerland nodes	2279	1023.6	149.1	138.6	87.3	34.8	43.6	29.9	34.1	32.1	27.8	25.9	24.9	21.9
edges	6015	2647.3	557.9	522.1	335.2	169.1	204.4	148.6	164.9	157.8	140.4	130.9	129.5	115.1
time [μs]		1485.5	271.4	308.6	206.6	102.8	107.0	5381.4	85.6	572.5	224.8	467.0	1123.0	302.7
Italy nodes	2399	1220.0	128.3	117.6	69.3	26.1	33.7	24.0	26.1	25.9	22.7	22.8	21.9	18.5
edges	8008	4018.6	678.1	623.4	428.0	201.5	251.1	195.8	207.7	208.6	190.1	190.0	185.5	164.0
time [μs]		1990.9	301.7	346.9	229.5	114.4	118.7	4735.6	92.7	763.4	210.3	666.1	1228.5	315.0
France nodes	4598	2395.2	255.3	236.2	143.0	59.9	73.0	46.1	49.9	55.5	41.2	40.8	39.5	30.2
edges	14937	7677.3	1356.0	1245.2	809.9	408.5	487.9	334.3	362.0	388.1	308.0	299.9	297.3	236.5
time [μs]		4581.5	847.8	952.9	642.1	343.8	319.5	8200.4	234.2	1672.7	507.4	1141.2	2014.5	683.6
Germany nodes	6884	3476.2	376.7	351.9	209.7	61.7	93.9	53.8	64.5	65.9	48.7	46.9	42.6	32.2
edges	18601	9276.2	1653.9	1548.9	880.1	359.8	514.3	326.0	377.5	377.1	300.1	290.8	268.7	214.4
time [μs]		6418.6	1249.1	1385.0	847.5	413.4	459.0	9122.6	347.6	1586.7	605.6	1202.8	2360.8	780.2
Moab nodes	1444	723.0	280.6	278.1	129.6	109.1	162.5	118.9	139.0	122.3	119.9	125.1	109.4	94.5
edges	3060	1548.5	637.9	632.7	298.2	249.2	373.6	271.5	318.9	280.1	274.9	287.1	250.6	215.4
time [μs]		871.3	361.5	407.9	179.9	158.8	221.6	20395.7	190.3	980.6	749.3	1018.4	2073.2	689.9
SC nodes	3045	1494.2	358.3	349.3	140.8	83.4	113.8	84.8	91.4	93.0	79.4	82.8	75.0	60.2
edges	7310	3654.7	993.2	969.1	389.2	238.8	325.8	244.4	264.9	269.8	231.1	240.0	219.4	175.3
time [μs]		2257.7	693.3	789.1	318.8	212.8	260.9	34736.5	196.1	1055.9	576.5	941.6	1822.7	810.6
New nodes	16471	7984.0	1938.0	1914.5	668.0	403.6	632.4	404.2	483.3	434.0	380.3	366.0	364.0	267.0
Hampshire edges	34530	16875.7	4309.6	4256.9	1489.5	912.8	1421.0	909.6	1086.4	978.7	857.4	825.7	823.5	603.4
time [μs]		13841.2	4650.5	5114.3	2094.9	1541.2	1905.2	596121.0	1516.3	4493.5	3210.8	4191.3	8907.0	3860.6
LO nodes	25982	12054.6	2623.4	2594.4	849.8	533.6	731.3	414.1	526.6	431.5	398.6	472.2	453.3	267.0
edges	57620	26835.9	6124.8	6058.1	1968.0	1254.9	1725.3	978.6	1241.4	1032.0	953.4	1120.7	1077.9	632.7
time [μs]		22241.0	6683.1	7291.7	2900.8	2246.3	2581.1	56817.2	2039.0	5033.4	3802.2	5626.5	9413.0	5685.7
HE nodes	38823	19414.5	6755.2	6733.7	2361.1	1685.1	3165.4	1742.8	2273.9	1316.7	1279.7	1798.5	1823.5	979.2
edges	79988	39950.5	14206.4	14159.8	4986.8	3553.4	6675.9	3675.6	4799.9	2794.9	2715.0	3810.3	3858.0	2069.3
time [μs]		32971.0	18115.7	16692.2	6730.5	5324.8	8142.0	198930.0	5914.9	12437.5	10641.8	17134.0	31499.0	17604.3
PT nodes	45073	22435.1	8956.7	8926.6	3052.9	2239.2	4749.2	2360.7	3089.0	1742.6	1708.2	2220.9	2509.2	1396.0
edges	91314	45385.2	18390.2	18328.4	6268.8	4608.7	9762.8	4863.6	6357.7	3601.6	3530.1	4585.0	5176.4	2886.4
times[μs]		36900.7	19518.9	21555.2	8499.3	6892.1	11593.2	2645990.0	7809.4	15824.3	14135.9	20438.8	46369.6	23983.6
Road77 nodes	45852	17108.5	4688.3	4650.2	2076.2	1501.7	2073.2	1359.9	1451.9	1409.1	1325.6	1313.6	1298.5	973.0
edges	98098	37965.2	11194.0	11102.8	5063.7	3646.4	4956.0	3260.0	3470.2	3397.5	3189.2	3142.4	3119.3	2321.9
time [μs]		32443.0	11678.3	12823.2	6559.7	5381.9	6366.0	809337.0	6381.2	15223.0	12414.6	14254.8	28339.6	14138.1
Polk nodes	51510	21902.8	7612.3	7591.9	2666.5	1876.2	2722.2	1535.6	1711.0	1643.3	1540.9	1657.8	1471.9	919.2
edges	110676	48124.6	17947.9	17900.0	6372.0	4500.1	6484.2	3661.8	4081.4	3932.3	3685.3	3956.7	3522.8	2183.5
time [μs]		42589.5	18765.2	20907.4	8281.2	6553.8	8255.2	214553.0	5731.1	17856.9	14158.4	18271.5	29745.4	13677.3