Minimum-Link c-Oriented Path Queries

John Adegeest Mark Overmars Jack Snoeyink

RUU-CS-91-27
July 1991

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Minimum-Link c-Oriented Path Queries

John Adegeest Mark Overmars Jack Snoeyink

Technical Report RUU-CS-91-27
July 1991

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Minimum-Link c-Oriented Path Queries*

John Adegeest Mark Overmars Jack Snoeyink†

Department of Computer Science
Utrecht University

Abstract

We consider paths, composed of a minimum number of line segments parallel to a fixed set of c orientations, that avoid a set of obstacles in the plane. We preprocess n line segment obstacles with disjoint interiors and a starting point A into a data structure using $O(cn)$ space. With this data structure, we can determine the minimum number of line segments l of a path from A to a query point B in $O(c \log n)$ time and construct a path in additional $O(l)$ time. Preprocessing takes $O(c^2 n \log n)$ time and space or $O(c^2 n \log^2 n)$ time and $O(c^2 n)$ space. Usually c is a constant—as an example, $c = 2$ for rectilinear paths.

1 Introduction

Although the shortest distance between two points in the plane is a straight line, the shortest possible path usually is not. The presence of obstacles makes finding shortest paths an interesting computational problem. The number and kind of obstacles are not the only factors that determine the difficulty of the problem; also important is how 'shortest' is measured, what other restrictions are placed on the path, and whether precomputation is allowed.

In this paper, we study the "link metric," in which the length of a path is the number of straight line segments that it consists of. This is a natural metric for robots or machines for which turning is more expensive than moving along a line. Shortest paths under the link metric have been studied by several researchers. Linear time algorithms [10, 14, 20] are known for paths inside simple polygons. Mitchell, Rote, and Woeginger [15] have given an algorithm for the general problem that runs in $O(E\alpha(n)\log^2 n)$ time and $O(E)$ space, where n is the number of obstacle segments, $\alpha(n)$ is the slowly growing inverse of Ackermann's function, and E is the size of the "visibility graph," which can be $\Theta(n^2)$.

We restrict our paths (and obstacles) to be *c-oriented*—that is, to consist of segments with a fixed set of c orientations, as defined in the next section. R. Güting [12], driven by applications and hopes for more efficient algorithms, defined c-oriented polygons as a generalization of orthogonal polygons; he and others [13, 14, 18] have looked at various geometry problems

*This research was supported by the ESPRIT Basic Research Action No. 3075 (project ALCOM). The second author was also partially supported by the Dutch Organization for Scientific Research (N. W. O.).

†On leave from the Department of Computer Science of the University of British Columbia.

restricted to c fixed orientations. Usually, c is a small constant: The most common example, rectilinear or Manhattan paths, are a special case with $c = 2$ and VLSI design sometimes adds the 45° angles, setting $c = 4$.

Under the Euclidean metric, researchers have shown that rectilinear ($c = 2$) shortest paths among rectilinear barriers can be found much faster than unrestricted paths: the current fastest algorithm runs in $O(n \log^2 n)$ time [3] for rectilinear paths, as compared to building and searching the visibility graph in $O(n^2)$ time and space [11] for unrestricted paths.

Under the link metric, linear algorithms for rectilinear paths in a simple polygon are known [5, 14] and a very recent algorithm of Das and Narasimhan [4] finds the minimum link rectilinear path among rectilinear obstacles in $O(n \log n)$ time and linear space. Unfortunately, only the algorithm of Hershberger and Snoeyink [14] extends to minimum link c-oriented paths for greater c and that only in simple polygons.

In this paper, we preprocess a set of n line segment obstacles with disjoint interiors and a starting point A into a data structure of $O(cn)$ size that can 1) determine the number of links l of a minimum path to a query point B in $O(c \log n)$ time and 2) construct a minimum link path in additional $O(l)$ time. Preprocessing takes $O(c^2 n \log n)$ time and space or $O(c^2 n \log^2 n)$ time and $O(c^2 n)$ space. Our approach is a common one: for a given starting point A we iteratively determine the set of all points that can be reached by paths of one link, two links, etc. This must be done carefully to achieve the stated bounds because the description any one of these sets may have quadratic complexity.

In the next section, we define c-oriented paths and reachable regions. In section 3 we investigate *segment deletion queries*, which are a useful subproblem for building reachable regions. Then, in section 4, we show how to use segment deletion queries to build a query structure for minimum link c-oriented paths.

2 Preliminaries

In this section we define some geometric objects such as c-oriented paths and reachable regions and investigate some of their properties.

Given a set of n obstacle segments with disjoint interiors, a *path* from A to B is any curve joining A and B that does not cross any obstacle segment.[1] We assume that a fixed set C of c orientations is given and call a segment *c-oriented* if it is parallel to one of these c orientations. A path is *c-oriented* if it consists of c-oriented line segments. Rectilinear paths, for example, consist of segments parallel to the x and y axes, so they are 2-oriented. Our problem is to find the path with the fewest c-oriented segments, or links, that goes from A to B among n obstacle segments.

[1] The precise definition is this: a path does not cross the segment s if, for every subsegment $s' \subset s$ that does not include an endpoint, there is an $\epsilon > 0$ such that the path does not intersect the ϵ neighborhood of s' on one side of the line containing s.

If the paths are restricted to be c-oriented, then it is wise to assume that the n obstacle segments are also c-oriented—not only because in applications such as VLSI the obstacles are often previous paths, but also because any angle that doesn't contain one of the c orientations requires infinitely many links to reach its vertex. Figure 1 shows a rectilinear path between two obstacles with vertices on the origin and angles of α and $\pi/4$. The path starting at $(1,1)$ can go down to $(1,\tan\alpha)$ and over to $(\tan\alpha,\tan\alpha)$. Using $2i$ links, the smallest x coordinate that can be reached is $(\tan\alpha)^i > 0$; though the origin is approached, it is never attained.

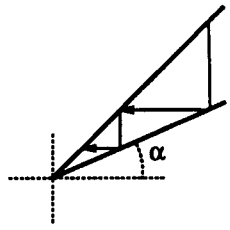

Figure 1: A path toward the origin

We have been using the word "reach" informally; let us now define it formally. We say that a set X can *reach a point p in orientation τ* if there is a line segment s, parallel to τ, from p to a point in X such that s does not cross any obstacle segment. The set X can *reach* the point p if X can reach p in some orientation. The points that a closed set can reach in a given orientation are a union of trapezoids.

Lemma 2.1 *The points that a closed set X can reach in orientation τ are those inside trapezoids (and on trapezoid edges) that intersect X in a trapezoidation, parallel to τ, of the obstacles and the extreme points of X.*

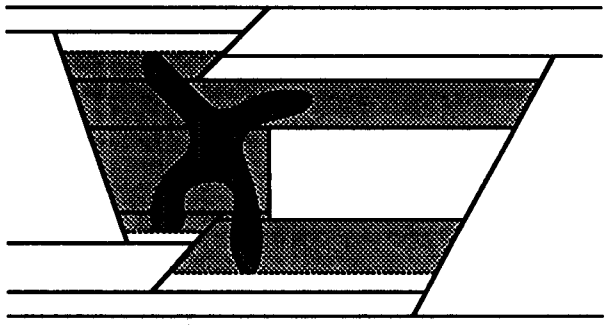

Figure 2: The points that reach X in the horizontal orientation

Proof: The points that X can reach in orientation τ lie on segments parallel to τ that intersect X and do not cross an obstacle. In the trapezoidation described in the lemma and depicted in figure 2, the set X either cuts completely through a trapezoid or does not intersect the interior at all. Therefore, the union of segments equals the trapezoids intersecting X. ∎

We define the *ith reachable region $\mathcal{R}_i$* as the set of all points that A can reach by a c-oriented path with i or fewer links. Thus, the first reachable region consists of the longest line segments through A in the c permitted orientations that do not cross obstacles. The second region adds to this all points that can reach one of these line segments, as shown by the shading in figure 3. Clearly, an equivalent definition of $\mathcal{R}_i$ is the set of points that $\mathcal{R}_{i-1}$ can reach.

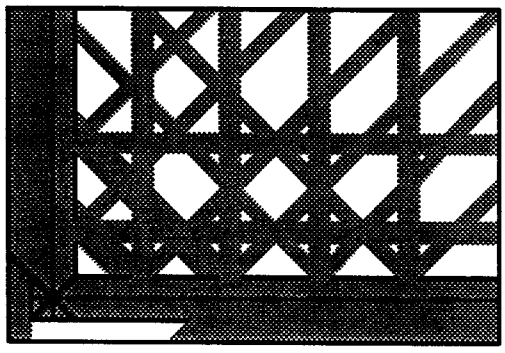

Figure 3: Reachable regions $\mathcal{R}_1$ (dotted lines) and $\mathcal{R}_2$ (shaded) for four orientations

Lemma 2.2 *The reachable region $\mathcal{R}_i$ is a polygonal region bounded by obstacles and line segments parallel to the c permitted orientations.*

Proof: Lemma 2.1 implies that $\mathcal{R}_i$ is the union of trapezoids with the c orientations that intersect $\mathcal{R}_{i-1}$. ∎

We say that the non-obstacle boundary segments of reachable region $\mathcal{R}_i$ are *windows*. A trivial lemma proves that any point p that is added to $\mathcal{R}_{i-1}$ to form $\mathcal{R}_i$, in other words, any point $p \in \mathcal{R}_i \setminus \mathcal{R}_{i-1}$, can reach a window of $\mathcal{R}_{i-1}$.

Lemma 2.3 *The points added to $\mathcal{R}_{i-1}$ to form $\mathcal{R}_i$ can be reached by a window of $\mathcal{R}_{i-1}$.*

Proof: A point outside $\mathcal{R}_{i-1}$ that is reached by $\mathcal{R}_{i-1}$ must be reached from a non-obstacle boundary segment, which is a window of $\mathcal{R}_{i-1}$. ∎

This leads to the basic idea of the algorithm: form region $\mathcal{R}_i$ by finding the trapezoids that lemma 2.1 says can reach each window of $\mathcal{R}_{i-1}$. Unfortunately, figure 3 illustrates that a reachable region may have quadratically many windows. Thus, we cannot afford to compute or store an explicit representation of region $\mathcal{R}_i$ or of its windows. Instead, we compute c structures of linear complexity, one for each permitted orientation, whose union covers $\mathcal{R}_i$. In these structures, we determine the trapezoids containing points that are reached by a set of line segments that contain all windows but are contained in the appropriate reachable region. To avoid looking at a trapezoid too many times, we study the problem of segment deletion queries in the next section. Segment deletion queries will be an important subroutine in the c-oriented path algorithm.

The fact that we work with a cover leads to its own complications—one must avoid doing too much work in any area of one structure that has already been covered in another structure. For example, infinitely many horizontal and vertical windows could be generated in the horizontal and vertical structures for the angle shown in figure 1 if it is not known that the region is covered in the permitted α or $\pi/4$ orientations. We address these complications in section 4.

3 Segment deletion queries

In this section we give two solutions to a simple form of dynamic point location, the problem of *segment deletion queries.*

Problem 3.1 *Given a set of n horizontal line segments $S = \{s_1, s_2, \ldots, s_n\}$, form a data structure that, given a vertical query segment v, can report all segments that intersect v and delete them from the structure.*

The first solution uses an interval tree with priority search trees as secondary structures to answer m queries in $O(m \log^2 n + n \log n)$ total time using $O(n)$ space. The second uses persistent search trees to achieve $O((m+n)\log n)$ total time using $O(n \log n)$ space. Preprocessing for both solutions is $O(n \log n)$.

For the first approach, we define an interval tree as follows: The distinct x coordinates of the endpoints of segments in S divide the x axis into $n' < 2n$ bounded adjacent intervals; let $M = \{m_1, m_2, \ldots, m_{n'}\}$ be the midpoints of these intervals. We label nodes recursively, beginning at the root, with the midpoints of M and store the segments of S.

Given a set of adjacent midpoints M, a set of segments S, and a subtree, label the root with the median midpoint $\hat{m} \in M$. We use $\hat{m}$ to partition M into two sets: $M_l = \{m < \hat{m} | m \in M\}$, the midpoints less than $\hat{m}$, and $M_r = \{m > \hat{m} | m \in M\}$, those greater than $\hat{m}$. We partition S into three sets: $S_{\hat{m}}$, the segments intersecting the vertical line $x = \hat{m}$, and S_l and S_r, the segments left and right of the line $x = \hat{m}$. We store the segments $S_{\hat{m}}$ at the node labelled $\hat{m}$ and pass the sets M_l and S_l to the left child, if they are not empty, and pass M_r and S_r to the right child, if they are not empty. Then we recursively form the subtrees for the children. Since each segment s_i crosses some midpoint line and we halve the number of midpoints in going from parent to child, the height of the interval tree is $O(\log n)$.

The segments stored at node $\hat{m}$ each have one endpoint with x coordinate greater than $\hat{m}$ and one smaller. We store these segments in two priority search trees, with priority increasing with x on the right and decreasing with x on the left. We describe only the right structure—the one on the left is done with mirrors.

We build a complete binary search tree on the y coordinates of the stored segments. At each internal node t, we also store the segment with the greatest x coordinate that is in the subtree of t. We can delete a segment by removing its leaf and repairing the priority on the path to the root in at most $O(\log n)$ time. (Since only deletions are performed in this tree, no rebalancing is needed to preserve the $O(\log n)$ access time.)

Theorem 3.1 *Using an interval tree with priority search trees, one can store n horizontal segments and answer m segment deletion queries in $O(m \log^2 n + n \log n)$ time using $O(n)$ space.*

Proof: To answer a query for a vertical segment $v = \langle (x_v, y_{\min}), (x_v, y_{\max}) \rangle$, we visit the interval tree nodes on a root to leaf path. At an interval tree node labelled $\hat{m}$, we compare x_v to $\hat{m}$ to decide which of the two associated priority search trees to use and whether to visit the left or right child of $\hat{m}$.

If $x_v \leq \hat{m}$, then we search for $y_{\min}$ and $y_{\max}$ in the left priority search tree. For each of the $O(\log n)$ subtrees between the two search paths, we look at the segment s stored at the subtree's root. While v intersects s, we report and delete s, bringing a new segment to the root. When v does not intersect s, then all segments in the subtree end before x_v. If $x_v \geq \hat{m}$, instead, we handle the right priority search tree similarly.

For each of the $O(\log n)$ interval tree nodes visited by a query, we spend $O(\log n)$ time searching plus $O(\log n)$ time for each segment that we report and delete. Thus, the query time is as claimed. Notice that the priority search trees have size proportional to the number of segments they store and that each segment $s_i \in S$ is associated with the unique highest level node whose midpoint line s_i crosses. Therefore, the storage is linear in n. ∎

For the second approach, we use the node-copying method of persistent data structures [7]. Let $x_1, x_2, \ldots, x_{n'}$ be the sorted list of distinct x coordinates of segments of S. We initialize a balanced binary search tree on segments that end on the line $x = x_1$ and set $root(1)$ to be the root. Since these segments are assumed disjoint, they can be ordered by aboveness.

Now, assume we have a binary search structure $root(i-1)$ for the segments whose left endpoints are left of $x = x_i$ and whose right endpoints are right of $x = x_{i-1}$. To form the structure $root(i)$ we wish to delete all segments with right endpoint on $x = x_i$ and add all segments with left endpoint on $x = x_i$. Rather than modifying the previous structure $root(i-1)$, however, we make copies of the nodes that would be modified and modify the copies; we set $root(i)$ to be the copy of the root. Using standard balanced binary search tree algorithms, we know that the deletion or insertion of a segment will change at most $\log n$ nodes on the root-to-leaf path of the tree. Thus, both the time and space for the construction of all search trees is $O(n \log n)$.

When we begin to query and delete segments, we assume that every segment and every node of the tree is unmarked. Given the query segment $v = \langle (x_v, y_{\min}), (x_v, y_{\max}) \rangle$, we locate x_v in the interval $[x_i, x_{i+1})$ by binary search. We then search for y_{min} and for $y_{\max}$ in the tree $root(i)$, stopping either search when a marked node is encountered. We mark any unmarked nodes of subtrees between the search paths to $y_{\min}$ and $y_{\max}$; if the segment stored at a newly marked node is unmarked, we report and mark it.

Theorem 3.2 *Using persistent search tree methods, one can store n horizontal segments and answer m segment deletion queries in $O((m+n)\log n)$ time using $O(n \log n)$ space.*

Proof: First, we prove that the algorithm correctly reports all segments remaining in the data structure that are intersected by the query segment. Though a tree node may have several parents in the data structure, it has a unique subtree. By induction on the number of nodes, one can prove that when the root of a subtree is marked, the subtree contains only marked nodes. Since marked nodes store marked segments, the query algorithm does not miss any segments by stopping the search at marked nodes.

For each query, one performs binary searches using the x and both y coordinates of the query's endpoints. This takes $O(\log n)$ time. We can assume that the result of this searching is a list of at most $\log n$ subtrees with unmarked roots that lie between the y

search paths and, thus, need to be marked. (Discarding previously marked subtree roots merely increases the constant in the big-O.) Thus, finding the nodes to mark takes a total of $O(m \log n)$ time for m queries.

We argued above that the total size of the structure is $O(n \log n)$. Since each node is marked once, the cost of marking nodes is also bounded by $O(n \log n)$. ∎

We remark that both solutions extend to sets of disjoint segments that are not necessarily horizontal. The second method works as stated above; for the first, the priority search can be replaced by a clever procedure of Cheng and Janardan [2].

4 Computing Reachable Regions

We first give an algorithm to compute the reachable regions using segment deletion queries, then prove it correct and analyze its running time.

4.1 The Algorithm

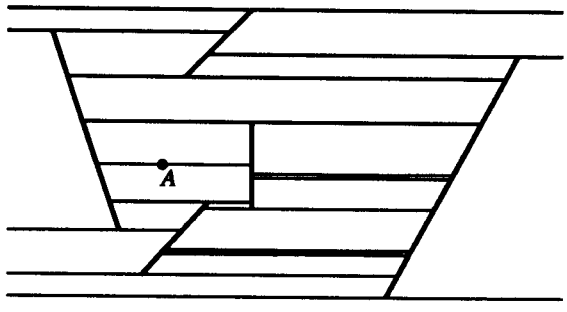

Figure 4: A horizontal trapezoidation with two horizontal obstacles

Our primary data structure consists of c trapezoidations: $Trap(\tau)$ is obtained by rotating τ, one of the c permitted orientations, to horizontal position, and then cutting horizontally from each vertex until encountering an obstacle. We call these horizontal cuts the *trapezoid edges* of $Trap(\tau)$. We consider the starting point A as one of the vertices for the purpose of computing trapezoids, so the first reachable region is the set of trapezoid edges through A.

To avoid degeneracies caused by points lying on the same horizontal line, we rotate the τ-oriented obstacle segments infinitesimally from horizontal, as shown in figure 4. Thus, there are four trapezoids incident on a τ-oriented obstacle segment, two of which are infinitesimally skinny.

We will label all trapezoids and edges of $Trap(\tau)$ by the number of links in the minimum path to A that starts in the trapezoid or edge with a segment parallel to τ—some trapezoids will need to be subdivided to perform this labelling. Thus, we provide every trapezoid edge with a label and a link field and every trapezoid with a label and link field and records for additional top and bottom subdividing edges with their own label and link fields. Initially, we set all labels to ∞.

For each trapezoidation $Trap(\tau)$, we compute $c-1$ segment deletion query structures. Structure $Sdq(\tau,\sigma)$ stores the trapezoid edges of $Trap(\tau)$for segment deletion queries parallel to σ, where $\sigma \neq \tau$ is one of the c permitted orientations. We also add a *representative segment* in the middle of each trapezoid and include these segments in the segment deletion query structures. Each representative segment stands for and has a pointer to the trapezoid that contains it.

To begin the algorithm, label the trapezoid edges through A with 1, set their link fields to A, and place them in a bag $\mathcal{B}$. This is the end of stage 1, which computes the first reachable region $\mathcal{R}_1$. Notice that the bag $\mathcal{B}$ contains all windows of $\mathcal{R}_1$.

To compute the region $\mathcal{R}_i$ at stage i, we assume that we have computed $\mathcal{R}_{i-1}$ and have a bag $\mathcal{B}$ of line segments contained in $\mathcal{R}_{i-1}$, such that every window of $\mathcal{R}_{i-1}$ is contained in some segment in $\mathcal{B}$. We initialize a new bag $\mathcal{B}'$ to empty.

For each trapezoidation $Trap(\tau)$, we perform the following procedure. Discard all segments with orientation τ from $\mathcal{B}$. For every remaining segment $s \in \mathcal{B}$, we wish to label all the edges and trapezoids of $Trap(\tau)$ that are not yet included in the reachable region. We begin with the segments and trapezoids that a segment $s \in \mathcal{B}$ cuts completely through; to this end, we perform point location with the ends of s in $Trap(\tau)$ and form a segment s' by trimming off the ends of s that do not cut a trapezoid completely.

If s has orientation $\sigma \neq \tau$, perform a segment deletion query for s' in the structure $Sdq(\tau,\sigma)$ and report the trapezoid representatives and trapezoid edges that s' intersects. If the representative of trapezoid Δ is reported, then s cuts completely through Δ. If Δ is labelled ∞, then we must make Δ part of the reachable region $\mathcal{R}_i$: set Δ's label to i and link field to s. Similarly, if the label of any reported trapezoid edge e is ∞, then we set edge e's label to i and link field to s and also put e in the new bag $\mathcal{B}'$.

After all segments of $\mathcal{B}$ have been considered for the trapezoidation $Trap(\tau)$, then we have labelled all edges and trapezoids of $Trap(\tau)$ that have been cut completely by some segment. We must still handle trapezoids that are partially cut.

We begin by collecting, for each trapezoid Δ that is still labelled ∞, the segments in $\mathcal{B}$ whose endpoints lie in Δ. (See figure 5.) Lemma 4.2, in the next subsection, will show that no segment is completely contained in a trapezoid labelled ∞. Thus, the segments ending in Δ fall into two groups: the segments that cut the top boundary edge of Δ and those that cut the bottom edge. Let s_t be the segment cutting the top boundary whose endpoint p_t is lowest and s_b be the segment cutting the bottom that whose endpoint p_b is highest. (One of s_t or s_b may be missing.)

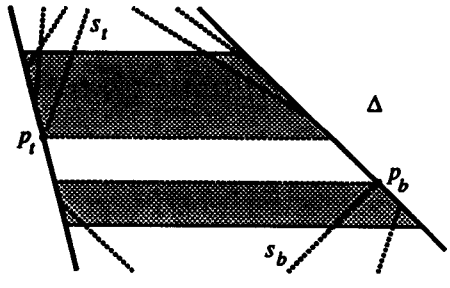

Figure 5: A partially cut trapezoid Δ

We must now subdivide the trapezoid into parts that are in reachable region $\mathcal{R}_i$ because segments s_t or s_b can reach them in direction τ (horizontal) and, perhaps, into parts that are not in the region $\mathcal{R}_i$. Recall that the trapezoid has records for top and bottom subdividing edges with link and label fields. If segment s_t exists, we set the top edge to the segment

through p_t parallel to τ, set its label to i, and its link field to s_t. Similarly, if s_b exists, we set the bottom edge to the segment through p_b, its label to i, and its link to s_b. We throw each top or bottom edge formed into the new bag $\mathcal{B}'$ because they may be windows of $\mathcal{R}_i$. We leave any remaining part of the trapezoid Δ to be labelled by the next stage of the algorithm.

Strictly speaking, what we have computed is a covering of Δ and not a subdivision. The shaded trapezoids in figure 5 overlap Δ and will even overlap each other if p_b is higher than p_t. We must remember this when we describe the query algorithm, but first we discuss how to terminate the preprocessing algorithm.

The claim, which we prove in the next subsection, is that after executing this procedure in every trapezoidation, the trapezoids and edges with finite labels exactly cover the reachable region $\mathcal{R}_i$. The new bag $\mathcal{B}'$ contains all boundary edges with link distance i, thus $\mathcal{B}'$ contains the windows of $\mathcal{R}_i$. If $\mathcal{B}'$ is non-empty, then we are ready to go to stage $i+1$. Otherwise, we have computed the reachability of the entire plane and are ready to answer queries.

To answer a query for the number of links of the minimum path to B, we locate B in each of the c trapezoidations $\mathit{Trap}(\tau)$. If B is contained in a trapezoid edge in $\mathit{Trap}(\tau)$, then the label of the edge is the distance from A when the first step is in direction τ. Otherwise B is contained in a trapezoid $\Delta \in \mathit{Trap}(\tau)$. If the subdividing edges of Δ have finite labels, then we test first whether B is above of the top edge or below of the bottom edge and, if so, take the corresponding edge label as the distance. In all other cases, the label of Δ is the distance from A when the first step is in direction τ. The minimum of the c distances found is the length of the minimum link path. A path that achieves this distance can be formed by following the associated link fields to A.

4.2 The Analysis

We can make one easy observation that helps both to prove correctness and to analyze the running time.

Observation 4.1 *All segments put in the bag at stage i are existing or newly formed trapezoid edges, which extend from obstacle to obstacle.*

There are two claims about segments partially cutting a trapezoid, one implicit and one explicit, that we prove before arguing that the entire algorithm is correct. The implicit claim is in the assignment to the subdividing edges. Since we do not check if an edge had already been assigned a value, we need to prove that such an assignment can be made only once.

Lemma 4.1 *If a subdividing edge of a trapezoid Δ is labelled i, then Δ is cut completely by an edge of label at most i. Thus, Δ is subdivided at most once.*

Proof: Let e be a subdividing edge in trapezoid Δ. If e is labelled i, then e was created by a segment s, labelled $i-1$, that ends on one of the obstacles of Δ, as illustrated in figure 6. Denote the orientation of this obstacle by τ and note that the orientation of s is not τ. Therefore, s crosses a trapezoid edge t with orientation τ formed by one of the obstacle endpoints; the label of t is thus at most i. Edge t cuts completely through Δ proving the lemma. ∎

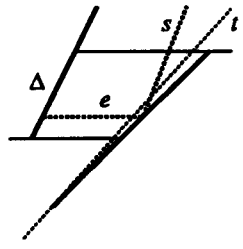

Figure 6:
s created e

The proof of the explicit claim, that no segment has both endpoints in a partially cut trapezoid, is similar.

Lemma 4.2 *A trapezoid labelled ∞ cannot contain a segment in the bag $\mathcal{B}$.*

Proof: As we observed, only existing trapezoid edges and newly-formed subdividing edges are put into the bag $\mathcal{B}$. Since the initial edges of a trapezoidation always touch a vertex, a segment contained in a trapezoid Δ is a newly created edge, call it e, that ends on the two obstacles of Δ.

As before, if e is labelled i, then e was created by a segment s labelled $i-1$ and there is a trapezoid edge t parallel to one of the obstacles that intersects s and e and is labelled i. Edge t cuts completely through trapezoid Δ, thus Δ receives a finite label (at most $i+1$) before e is considered. ∎

With these two subtleties out of the way, the rest of the correctness proof is straightforward.

Theorem 4.1 *The algorithm correctly computes a covering of the ith reachable region $\mathcal{R}_i$.*

Proof: We can prove inductively that every segment put in the bag $\mathcal{B}'$ in stage i is in $\mathcal{R}_i$ and that all windows of $\mathcal{R}_i$ are contained in segments put into the bag. For the base case, computing $\mathcal{R}_1$ in stage 1, this is true.

At the beginning of stage i, the induction hypothesis says that the segments of $\mathcal{B}'$ are in $\mathcal{R}_{i-1}$ and contain all windows of $\mathcal{R}_{i-1}$. The first condition and the definition of $\mathcal{R}_i$ imply that the points that these segments reach are contained in $\mathcal{R}_i$; the second and lemma 2.3 imply that these points and $\mathcal{R}_{i-1}$ contain $\mathcal{R}_i$.

By lemma 2.2, we know that the windows of $\mathcal{R}_i$ are trapezoid edges that are reached by the windows of $\mathcal{R}_{i-1}$. Furthermore, as long as $c > 1$, windows of $\mathcal{R}_{j-1}$ for $j \leq i$ will not be on the boundary of $\mathcal{R}_j$ because they reach points on both sides of themselves. Therefore we can use only previously unreached trapezoid edges as windows of $\mathcal{R}_i$. Our algorithm separately finds the two types of trapezoid edges that are contained in $\mathcal{R}_i$ but not in $\mathcal{R}_{i-1}$—first the existing trapezoidation edges by performing segment deletion queries, and second the subdivision edges. The correctness of the first computation is argued in section 3; the correctness of the second is argued above. Thus, the algorithm in stage i correctly finds segments of $\mathcal{R}_i$ that contain all windows of $\mathcal{R}_i$. ∎