Geometric Searching and Link Distance

(Extended Abstract)

Gautam Das Giri Narasimhan

Mathematical Sciences Department
Memphis State University
Memphis, TN 38152

ABSTRACT

Given n orthogonal line segments on the plane, their *intersection graph* is defined such that each vertex corresponds to a segment, and each edge corresponds to a pair of intersecting segments. Although this graph can have $\Omega(n^2)$ edges, we show that breadth first search can be accomplished in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. As an application, we show that the *minimum link rectilinear path* between two points s and t amidst rectilinear polygonal obstacles can be computed in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space, which is optimal. We mention other related results in the paper.

1 Introduction

Given a set S of n orthogonal line segments on the plane, the *intersection graph* of S is defined as follows: each segment corresponds to a vertex in the graph, and an edge connects a pair of vertices if the two corresponding segments intersect. Clearly this graph is bipartite with the horizontal and vertical segments forming two independent sets.

This graph can potentially have $\Omega(n^2)$ edges. Given an initial segment h, the main result of this paper is an efficient algorithm to label every segment by its shortest distance from h in the graph. This is equivalent to breadth first search. Our algorithm does not explicitly generate all the edges, and in fact runs in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. It uses elementary data structures such as binary trees and priority queues. Using this algorithm we solve several other problems efficiently.

Asano and Imai have shown how to perform breadth first and even depth first search on such graphs in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n \log n)$ space ([IA86], [IA87]). They use fairly complex data structures. However their techniques have wider applications.

The primary application of our result is motivated by a motion planning problem. Suppose we are given a collection of disjoint rectilinear polygonal obstacles inside a

rectilinear polygonal room on the plane, and a pair of points s and t in the free space. Let the input size be n. Consider the problem of computing the *minimum link rectilinear path* between s and t that avoids all the obstacles. Such a path is composed of a minimum number of line segments, each of which is parallel to either the x or the y axis. The number of line segments on the path is known as the *link distance* between the pair of points. We first reduce the problem to performing *two* breadth first searches on an intersection graph of $\mathcal{O}(n)$ orthogonal line segments. The technique in [IA86] and [IA87] can accomplish these searches in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n \log n)$ space. Instead, we show how to use our algorithm to solve the problem in $\mathcal{O}(n \log n)$ time and linear space. We also solve several other related problems.

Before we summarize our results, we introduce the following definitions. Given some rectilinear polygonal obstacles inside a polygonal room, let H (resp. V) be the set of horizontal (vertical) line segments formed by extending each horizontal (vertical) polygonal edge within the free space (possibly in both directions) until it hits an obstacle or the room. Our results are listed below, and assume rectilinear geometry.

Results

1. Given a collection of polygonal obstacles inside a room, and their H and V sets, we show that breadth first search on the intersection graph of $H \cup V$ can be accomplished in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space, using elementary data structures.
2. Given *any* collection S of line segments, we can perform breadth first search on their intersection graph in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. This is a generalization of the previous result.
3. Given a collection of polygonal obstacles inside a room, and points s and t, the minimum link path can be computed in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space, which is optimal.
4. Given a collection of polygonal obstacles in a room, and a point s, we can preprocess the input in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space such that given a query point t, its minimum link path from s can be reported in $\mathcal{O}(\log n + k)$ time, where k is the link distance. Its link distance from s can be reported in $\mathcal{O}(\log n)$ time.

Remarks

Our results are interesting for several reasons. Firstly, as was noted by Imai and Asano, extending their scheme in [IA86] to perform breadth first search in $\mathcal{O}(n)$ space seems difficult, although we note that their technique is quite general and easily allows other searching schemes such as depth first search. Secondly, to achieve our bounds, we employ several interesting ideas. To start with, we show that if the line segments were the same as those generated by extending the edges of a group of polygons, then the claimed bounds can be achieved. The algorithm uses a key idea of searching the

horizontal and vertical lines *separately*, which allows us to exploit various geometric properties. We then generalize this result for *any* set S of line segments by showing that there exists a group of polygons such that $H \cup V = S$. Given S, these polygons can be computed in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. Interestingly, this is related to the problem of computing *many faces* in an *arrangement* of line segments [EGS88].

Next, consider the applications to link distance problems, which is motivated by motion planning. Here a robot has to navigate amidst obstacles inside a cluttered workspace, where translational motion is considered cheap while directional changes are considered expensive. Thus we seek to minimize the links along the path. Several results have appeared for the nonrectilinear versions of the problem. Suri has studied the problem of minimum link distances inside a simple polygon without holes, and has obtained linear time optimal algorithms [S86]. Several people have also studied related concepts such as the *link diameter* and the *link center* of the region ([S90], [LPS87], [K89], [DLS89]). The rectilinear link distance problem has been studied for the case of a simple polygon without holes ([B91]). The problem becomes more complex when we allow multiple obstacles. Recently Mitchell et al. [MRW90] have designed an algorithm for computing nonrectilinear link distances amidst multiple polygons, and their algorithm runs in $\mathcal{O}(n^2 \alpha(n) \log^2 n)$ time and $\mathcal{O}(n^2)$ space, where $\alpha(n)$ is the inverse Ackermann's function.

The above result is suboptimal, and in fact this is generally true for most existing algorithms for shortest path problems amidst polygonal obstacles under various distance metrics. For example, consider the shortest path problem under the *Euclidean metric*. The best known algorithm runs in $\mathcal{O}(n^2)$ time and $\mathcal{O}(n^2)$ space [GM87]. Under rectilinear geometry, this problem can be solved in $\mathcal{O}(n \log^2 n)$ time and $\mathcal{O}(n \log n)$ space [CKV87]. Optimal algorithms have only been obtained for very restricted inputs. For example, if the input is a collection of rectangles, then the rectilinear Euclidean shortest path can be computed in $\mathcal{O}(n \log n)$ time [DLW89]. This is possible by a plane sweep because it can be shown that such paths have to be monotone in some direction.

In our case, we allow rectilinear polygons of arbitrary shapes, and the resulting minimum link paths need not be monotone in any direction. Yet our algorithm runs in optimal time and space.

The rest of this paper is organized as follows. In Section 2, we discuss the breadth first search algorithm (Result 1). Section 3 discusses how to generalize this algorithm where the input is an arbitrary set of orthogonal line segments (Result 2). Several problems related to minimum link distances are discussed in Section 4 (Results 3 and 4). We conclude with some open problems.

2 Labelling Algorithm

For this algorithm we are given a collection of disjoint rectilinear polygonal obstacles inside a polygonal room in the plane along with their H and V sets (as described in the Introduction). We are also given a distinguished line segment h in $H \cup V$. Without loss of generality we assume that h is a horizontal line segment. Consider the intersection graph I obtained by all these horizontal and vertical line segments. In this section we show how to perform an efficient breadth-first search in this graph starting from h so that every line segment is labelled with its distance (in I) from h.

For the sake of simplicity, we will assume throughout this section that all the x-coordinates (resp. y-coordinates) of the vertical (resp. horizontal) sides of the polygons are distinct, although our algorithm can be modified to tackle the general case.

Outline

An efficient search in I is achieved by searching in two different planar partitions – the horizontal planar partition, and the vertical planar partition. The horizontal (vertical) planar partition, which is denoted by HPP (VPP), includes $H(V)$ and all the edges of the obstacles. The algorithm works in two parts, each of which is divided into many phases. In the first part, the algorithm searches in the HPP to compute the shortest path from h to all the other horizontal segments. In the second part, the algorithm performs a search in the VPP to compute the shortest path from h to all the vertical segments.

We first describe the search performed in the HPP. In each phase the algorithm determines the set of horizontal line segments that are at distance 2 away from the horizontal line segments discovered in the previous phase. We denote the set of new horizontal segments chosen in phase k by H_k with $H_0 = \{h\}$. One way to imagine the algorithm is by using ideas from Suri et al. [S90]. One could imagine that in phase k the algorithm determines all new horizontal segments that get illuminated by placing light sources on the horizontal segments in H_{k-1} and shining them in the upward or downward direction. Hence in each phase the algorithm effectively jumps two steps by going from a set of horizontal segments H_{k-1} (at distance $2(k-1)$ from h) to a set of horizontal segments H_k (at distance $2k$ from h).

In the second part, the algorithm does a search in the VPP to label all the vertical line segments. In phase k of the second part, the algorithm labels all vertical segments at distance $2k-1$ from h in I.

Intuitively, the search is made efficient by maintaining two "complementary" data structures (HPP and VPP) instead of one. Maintaining these two data structures separately reduces the information stored about all the intersections between the segments. However, there is enough information stored in each of the structures to perform the requisite search.

It is possible to alternate the phases of the search in the HPP with that of the search in the VPP. This will ensure that all segments labelled k will be processed before any of the segments labelled $k+1$.

We now get into more details of the labelling algorithm.

Searching the HPP

The set H_0 consists only of the line segment h. In phase k the algorithm computes the set of new horizontal segments H_k that get illuminated by placing a light source on some segment from H_{k-1}.

We first describe an outline of phase k, which consists of *Initialization*, *UpSweep*, and *DownSweep* procedures. In the initialization procedure, the algorithm places light sources on the set of segments in H_{k-1}. It is clear that not all line segments in H_{k-1} need to be attached with a light source. We only need to attach light sources to portions of an "outermost" set of segments (as shown in Figure 2.1). The upsweep procedure directs all the light sources in the upward direction and labels (with label $2k$) all the new horizontal segments that get illuminated in the process. The horizontal segments are labelled in sorted order, sorted according to the y-coordinates of these segments. This is followed by downsweep, which directs all the light sources in the downward direction, and labels (with label $2k$) as many new segments as possible (in the order of decreasing y-coordinates). Figures 2.1, 2.2, and 2.3 illustrate the three procedures.

We now describe the upsweep procedure in more detail. At the start of an upsweep, the outermost line segments in H_{k-1} will be called *Fronts*, and beams of light are directed upwards from portions of each Front. Each such beam is called a *Window*. The upsweep consists of a sequence of upward advancements of Fronts. The next Front that is selected is the one that advances to the horizontal segment above with the least y-coordinate. This is facilitated by using a priority queue. These new Fronts get labelled $2k$. Effectively, the light sources are moved from the previous segment to the illuminated portions of these new segments. However, the original beams (i.e., windows) are likely to get modified (*narrowed, split* or *terminated*) if they encounter obstacle edges. Each Front is thus a dynamic collection of disjoint windows (see Figure 2.4 for examples). For a particular Front, when all its windows have terminated (which happens when all the light beams have been terminated by obstacles), the Front is also terminated. The upsweep terminates when all the Fronts terminate.

There are two main data structures used by the upsweep procedure. Since the set of windows in a Front are a set of disjoint intervals, the windows associated with a Front will be maintained as a balanced tree structure that maintains disjoint intervals. The balanced tree structure used here must allow the following operations: *Search, Insert, Merge, Split, Delete* (see Figure 2.4 for examples). Each of these operations can be performed in $\mathcal{O}(\log n)$ time, assuming any of the basic balanced tree structures like red-black trees. The second data structure is the priority queue, which is maintained as

a heap. This facilitates the following operations: *DeleteMin, Insert*, both of which can be performed in $\mathcal{O}(\log n)$ time.

The following are some of the subtle problems that the upsweep correctly handles. Firstly, propagating a set of windows to the new Front can be achieved by a constant number of elementary operations on the windows data structure of the previous Fronts. Secondly, using a heap structure effectively eliminates the possibility of a new segment being processed by different advancing Fronts at different times. Thirdly, a Front could encounter a segment of a previous phase. If this happens, the Front is immediately terminated. The reason is that either everything above it has already been processed, or there will be another Front that will carry on the advancement in that direction. This is illustrated in Figure 2.2, where Front c is terminated when it advances and hits Front b, since Front a would illuminate any new segments that c would have.

The downsweep procedure is similar to the upsweep procedure. There is one situation that needs further explanation. It is possible that a Front encounters a segment that was processed by the upsweep. However, this is ignored, and the sweep continues. The reason is that the downward sweep may have more segments that need to be labelled that could not possibly have been processed by an upward sweep. Thus some set of line segments could get processed twice within a phase (but not more number of times). No line segment is processed once in two different phases. This is crucial for our analysis.

Searching the VPP

This part of the algorithm is also divided into phases. The first phase involves finding out all the vertical segments that intersect h. These set of segments can also be set up as a collection of Fronts that can illuminate other vertical segments by shining light in the right or the left direction. From this point onwards, the algorithm proceeds along the lines of the earlier part of searching the HPP, with the difference that the search is done in the VPP. It suffices to say that in phase k, it will determine the vertical segments at distance $2k-1$ from h in I.

Analysis of Algorithm

Each line segment is processed in exactly one phase. Processing a line segment involves: 1) removing it from the heap structure, which can be performed in $\mathcal{O}(\log n)$ time; 2) modifying the Windows data structure, which can be performed in $\mathcal{O}(\log n)$ time; 3) finding the next segment to be inserted into the heap (which takes $\mathcal{O}(1)$ time), and inserting that segment into the heap (which again takes $\mathcal{O}(\log n)$ time). Within each phase, each line segment may be processed twice, once along an upward sweep, and once along a downward sweep. Consequently, it is clear that the algorithm runs in time $\mathcal{O}(n \log n)$.

The heap data structure uses $\mathcal{O}(n)$ space. At any instant in the algorithm there are at most n Windows active in all the Fronts put together. Hence the space complexity of the algorithm is $\mathcal{O}(n)$.

3 Arbitrary Sets of Orthogonal Line Segments

In this section we will show that given *any* set S of orthogonal line segments, breadth first search on their intersection graph can be accomplished in $\mathcal{O}(n \log n)$ time and linear space. To do this, we shall compute a collection of polygons inside a polygonal room such that $S = H \cup V$, where H and V are formed by extending each horizontal (vertical) edge of the polygons in the free space (possibly in both directions) until it hits the room or another polygon. After this is done, the algorithm in Result 1 can be used.

Let $S = H1 \cup V1$, where $H1$ ($V1$) is the set of horizontal (vertical) line segments in S. Consider the *arrangement* [EGS88] of the segments in S on the plane, as Figure 3.1 shows. Let us compute those faces of the arrangement (including the external face) that contain an endpoint of some segment in S. These *faces* will be our polygonal obstacles, with the room being the external face. We will shortly describe how to efficiently compute these faces. For now, let us imagine that these faces have been computed, and we are ready to run the algorithm in Result 1 on the intersection graph of their H and V segments. The following facts hold (some of them are trivial).

1. $H = H1$ and $V = V1$. Thus the intersection graph of $H \cup V$ is the same as the intersection graph of S.
2. If the intersection graph of S is disconnected then some of the obstacles may themselves have holes within which other obstacles may reside. However, this is not a problem for our algorithm in Result 1, because it will only restrict the search to one component.
3. The collection of polygons may have several horizontal (vertical) edges sharing the same y (x) co-ordinate. Again, this is not a problem because the algorithm in Result 1 can be modified to handle such degeneracies.

Thus, all we need to do first is to compute these faces efficiently. Let P be the set of $2n$ endpoints of all segments in S. Our problem is a particular instance of a more general problem, which is as follows. Given a set of arbitrarily oriented segments S and a point set P, compute all faces of the arrangement of S that contain some point in P. This problem has been solved in [EGS88], unfortunately it has a running time which is unacceptable for our purposes. However, we can exploit the orthogonality of our S and modify the algorithm to run in $\mathcal{O}(n \log n)$ time and linear space. For this it will help if we briefly review the algorithm in [EGS88].

The algorithm works roughly as follows. S is divided into two sets S_1 and S_2 of approximately equal size. Then the faces in the arrangement of S_1 (S_2) which contain points of P are recursively computed. Each face resembles a polygon with holes. The faces from S_1 (S_2) are known as the *red* (*blue*) polygons. The algorithm then performs what is known as a *red-blue merge*, which outputs every connected component of the intersections between the red and blue polygons that contain points of P. This merge is performed by a plane sweep, and if all the red (blue) polygons had r (b) edges the running time of the merge is $\mathcal{O}((r+b+|P|)\log(r+b+|P|))$ [EGS88]. The algorithm uses a priority queue as a data structure and requires $\mathcal{O}(r+b+|P|)$ space.

In our case we can considerably simplify the above algorithm. We can eliminate recursion and perform the red-blue merge only once. Partition S into $H1$ and $V1$. We know that horizontal (vertical) segments do not intersect among themselves. Thus the arrangement of $H1$ ($V1$) has *only one* multiply connected face which is the entire plane, with the horizontal (vertical) segments representing "holes". Let us call the face of $H1$ ($V1$) the red (blue) face. We run the red-blue merge only once with these two polygons and the point set P as input. The output is clearly the set of obstacles we are seeking. Since r, b, and $|P|$ are each $\mathcal{O}(n)$, the algorithm runs in $\mathcal{O}(n\log n)$ time and $\mathcal{O}(n)$ space.

In the next section we shall discuss some applications of our breadth first search algorithms.

4 Minimum Link Path Problems

In this section we consider the following problem. Suppose we are given a collection of rectilinear polygons inside a polygonal room, and points s and t within the free space. Let the total input size be n. We are required to compute the minimum link path from s to t which avoids all the obstacles. We have designed an algorithm for solving the problem in $\mathcal{O}(n\log n)$ time and $\mathcal{O}(n)$ space. In this version of the paper, we describe the algorithm for finding the link distance. We will only briefly outline how to modify the algorithm for computing the actual path.

Consider the VPP and HPP of the set of polygons, with s and t being treated as point obstacles. Both partitions can be computed in $\mathcal{O}(n\log n)$ time and $\mathcal{O}(n)$ space by a plane sweep algorithm described in [FM84]. It is easy to see that there exists a minimum link path which is confined to the *grid* formed by overlaying VPP on HPP. Of course we do not want to compute the grid as it will be too time consuming. Let S be the set of the horizontal segments of HPP and the vertical segments of VPP. Clearly s is associated with a horizontal (vertical) segment h_s (v_s), and similarly t is associated with a horizontal (vertical) segment h_t (v_t). The minimum link path has to start along either h_s or v_s, and end along either h_t or v_t.

We now make two copies of each partition, called HPP_h, HPP_v, VPP_h and VPP_v. The labelling algorithm is first run with h_s as the initial segment. In this run the partitions HPP_h and VPP_h are labelled. The algorithm is then run with v_s as the initial segment. In this run the partitions HPP_v and VPP_v are labelled. At this stage we have four partitions with each segment labelled. Consider for example VPP_h. The label associated with v_t, say k, tells us that the minimum link path from s to t which *starts* along h_s and *ends* along v_t has $k+1$ links. By examining all four partitions, we can find out the minimum link distance from s to t if the path originated along either h_s or v_s and terminated along either h_t or v_t. The minimum of all four values gives us the link distance between s and t. To compute the actual path, we have to modify the labelling algorithm to keep back pointers so that the actual path can be retrieved by following pointers. We omit the details in this version of the paper.

Clearly the algorithm runs in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. Notice that if we had performed the breadth first searches as in [IA86], our space complexity would have been nonlinear.

Optimality Proof

By reducing integer sorting to the the minimum link path problem, we show that our algorithm is optimal.

Assume that you are given n integers $a_1, \ldots, a_n$. Construct a polygon, P_i for each a_i. P_i is a strip of width $\epsilon = 0.1$ connecting the following points: $(a_i, a_i - \epsilon), (a_i, -a_i), (-a_i, -a_i), (-a_i, a_i), (a_i, a_i)$. It leaves a gap of width ϵ in the top right corner of the square region that the strip encloses. Hence, if $a_i > a_j$ then P_i completely encloses P_j. Now consider the problem of determining the minimum link path from the origin to $(B, 0)$, where B is 1 more than the largest integer in the input. The minimum link path will have to extricate itself from each of the polygonal regions. In the process it must sort the numbers.

Link Distance Query Problem

We next consider the *query* version of the above problem. Suppose we are given a collection of polygons in a room, and a point s. We have to preprocess the input into a data structure such that given any query point t, its minimum link path from s can be reported efficiently.

We describe an algorithm which takes $\mathcal{O}(n \log n)$ preprocessing time and $\mathcal{O}(n)$ space, and answers each query in $\mathcal{O}(\log n + k)$ time where k is the link distance between s and t. In fact we describe in detail an $\mathcal{O}(\log n)$ algorithm for reporting the link distance, and briefly outline how that may be modified to extract the path.

We first compute the HPP and VPP of the free space, treating s as a point obstacle. The point s corresponds to two line segments h_s and v_s. We then make two copies of each partition, called HPP_h, HPP_v, VPP_h and VPP_v. The labelling algorithm is first

run with h_s as the initial segment. In this run the partitions HPP_h and VPP_h are labelled. The algorithm is then run with v_s as the initial segment. In this run the partitions HPP_v and VPP_v are labelled. At this stage we have four partitions with each segment labelled. Finally we organize each partition into a data structure for *planar point location* queries [K83]. All of the above can be done in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.

We are now ready for query processing. Given a t, for each of the four partitions we find out the respective rectangle that contains it. Consider for example VPP_h. Suppose t is contained in a rectangle, both of whose vertical sides were labelled k. This means that the minimum link path from s to t which starts along h_s and ends along a vertical segment through t has $k+1$ links. But suppose the two vertical sides had different labels. Clearly they cannot differ by more than one, so let one be k and the other be $k+1$. This means that the minimum link path from s to t which starts along h_s and ends along a vertical segment through t has $k+2$ links.

Thus we can find out the minimum link distance from s to t if the path originated along either h_s or v_s and terminated along either a horizontal or vertical segment through t. The minimum of all four values gives us the link distance between s and t. Clearly all four point locations can be performed in $\mathcal{O}(\log n)$ time.

To compute the actual path, we have to modify the labelling algorithm to keep back pointers so that the actual path can be retrieved by following pointers. We omit the details in this version of the paper.

5 Conclusions

In this paper we show that breadth first search can be accomplished in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space in an intersection graph of n orthogonal line segments. The main idea behind the algorithm is that it searches the horizontal and vertical lines separately. We apply it to several link distance problems to obtain optimal algorithms. We conclude with some open problems.

1. Can depth first search be done on a set of orthogonal line segments in $\mathcal{O}(n \log n)$ time and linear space?
2. Are there other applications for the techniques used in this paper, namely that of maintaining separate data structures for the horizontal and vertical line segments?
3. Can the *Link Diameter*, and the *Link Center* problems [S90] be solved more efficiently in the rectilinear case?

6 References

[B91] de Berg, On Rectilinear Link Distance, *Computational Geometry: Theory and Application*, to appear.

[CKV87] Clarkson, Kapoor, and Vaidya, Rectilinear Shortest Paths through Polygonal Obstacles in $\mathcal{O}(n \log^2 n)$ time, *ACM Symposium on Comp. Geometry*, 1987.

[DLS89] Djidjev, Lingas, and Sack, An $\mathcal{O}(n \log n)$ Algorithm for Finding a Link Center in a Simple Polygon, *Proceedings of Sixth STACS, Lecture Notes in Computer Science, Springer Verlag Series*, 1989.

[EGC88] Edelsbrunner, Guibas, and Sharir, The Complexity of Many Faces in Arrangements of Lines and Segments, *ACM Symposium on Comp. Geometry*, 1988.

[FM84] Fournier, and Montuno, Triangulating a Simple Polygon and Equivalent Problems, *ACM Trans. on Graphics*, 1984.

[GM87] Ghosh, and Mount, An Output Sensitive Algorithm for Computing Visibility Graphs, *IEEE FOCS*, 1987.

[IA86] Imai, and Asano, Efficient Algorithm for Geometric Graph Search Problems, *SIAM J. of Comp.*, 1986.

[IA87] Imai, and Asano, Dynamic Orthogonal Segment Intersection Search, *J. of Algorithms,* **8** (1987), pp. 1-18.

[K89] Ke, An Efficient Algorithm for Link Distance Problems, *ACM Symposium on Comp. Geometry*, 1989.

[K83] Kirkpatrick, Optimal Search in Planar Subdivision, *SIAM J. of Computing*, 1983.

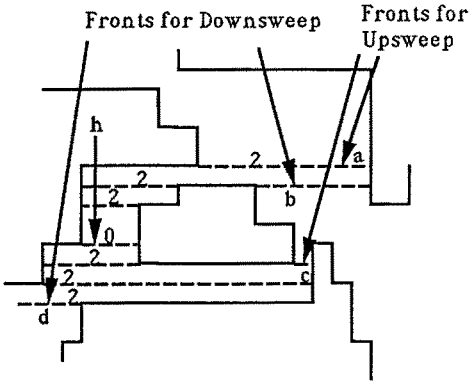
[LPS87] Lenhart, Pollack, Sack, Seidel, Sharir, Suri, Toussaint, Whitesides, and Yap, Computing the Link Center of a Simple Polygon, *ACM Symposium on Comp. Geometry*, 1987.

[MRW90] Mitchell, Rote, and Woeginger, Minimum Link Paths among Obstacles in the Plane, *ACM Symposium on Comp. Geometry*, 1990.

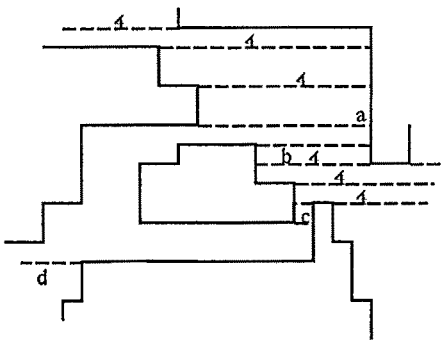
[RLW87] de Rezende, Lee, and Wu, Rectilinear Shortest Path with Rectangular Barriers, *Discrete and Comp. Geometry*, 1987.

[S86] Suri, A Linear Time Algorithm for Minimum Link Paths inside a Simple Polygon, *Computer Vision, Graphics, Image Processing*, 1986.

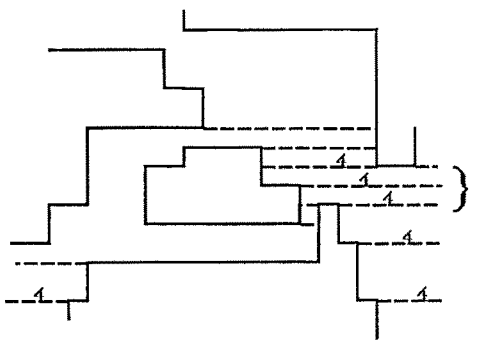
[S90] Suri, On some Link Distance Problems in a Simple Polygon, *IEEE Trans. on Robotics and Automation*, 1990.



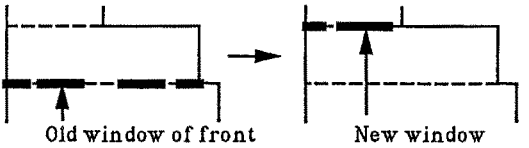
Initialization of Phase 2
Figure 2.1



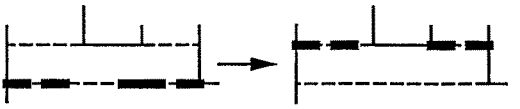
Upsweep
Figure 2.2



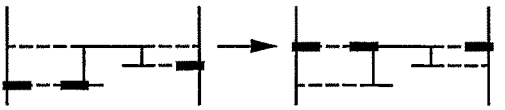
Downsweep
Figure 2.3



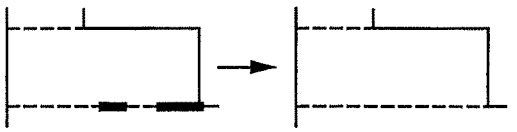
Window set splits at new front.



Window set splits. Some windows narrow.

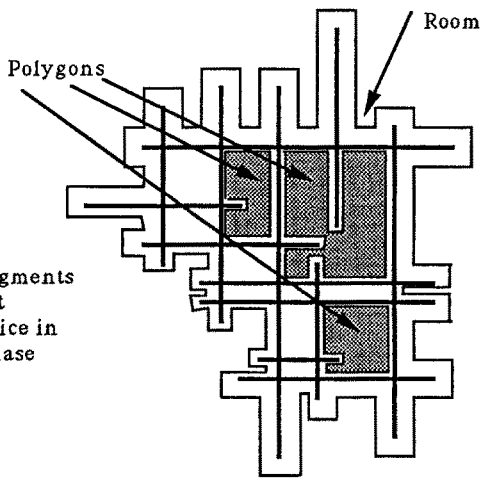


Window sets merge.



Windows terminate. Front terminates.

Fronts and Window Sets
Figure 2.4



Polygons from Segments
Figure 3.1