Algorithmica (1992) 8:55–88

Algorithmica
© 1992 Springer-Verlag New York Inc.

L_1 Shortest Paths Among Polygonal Obstacles in the Plane[1]

Joseph S. B. Mitchell[2]

Abstract. We present an algorithm for computing L_1 shortest paths among polygonal obstacles in the plane. Our algorithm employs the "continuous Dijkstra" technique of propagating a "wavefront" and runs in time $O(E \log n)$ and space $O(E)$, where n is the number of vertices of the obstacles and E is the number of "events." By using bounds on the density of certain sparse binary matrices, we show that $E = O(n \log n)$, implying that our algorithm is nearly optimal. We conjecture that $E = O(n)$, which would imply our algorithm to be optimal. Previous bounds for our problem were quadratic in time and space.

Our algorithm generalizes to the case of fixed orientation metrics, yielding an $O(n\varepsilon^{-1/2} \log^2 n)$ time and $O(n\varepsilon^{-1/2})$ space approximation algorithm for finding Euclidean shortest paths among obstacles. The algorithm further generalizes to the case of many sources, allowing us to compute an L_1 Voronoi diagram for source points that lie among a collection of polygonal obstacles.

Key Words. Shortest paths, Voronoi diagrams, Rectilinear paths, Wire routing, Fixed orientation metrics, Continuous Dijkstra algorithm, Computational geometry, Extremal graph theory.

1. Introduction. Recently, there has been much interest in the problem of finding shortest *Euclidean* (L_2) paths for a point moving in a plane cluttered with polygonal obstacles. This problem can be solved by constructing the *visibility graph* of the set of obstacles and then searching this graph using Dijkstra's algorithm [Di] or an A^* algorithm [Ni]. The bottleneck in the computation is in the construction of the visibility graph. It has been known for some time that the visibility graph can be constructed in time $O(n^2 \log n)$ [Le], [LW], [Mi1], [SS], where n is the number of vertices describing the set of obstacles. More recently, it has been shown that the visibility graph can be constructed in worst-case optimal time $O(n^2)$ [AAGHI], [We], or in output-sensitive time $O(e + n \log n)$ [GM], [KM], where $e = O(n^2)$ is the number of edges in the visibility graph. In some special cases in which the shortest path is known to possess certain monotonicity properties, algorithms that require $O(n \log n)$ time to compute shortest paths are known [LP], [Mi1], [SS]. While other algorithms for the general case are known whose running time can be written in other, possibly more favorable, terms [Mi3], [RS], it remains

[1] Partially supported by a grant from Hughes Research Laboratories, Malibu, California and by NSF Grant ECSE-8857642. Much of this work was done while the author was a Ph.D. student at Stanford University, under the support of a Howard Hughes Doctoral Fellowship, and an employee of Hughes Research Laboratories.

[2] Department of Applied Mathematics and Statistics, State University of New York at Stony Brook, Stony Brook, NY 11794-3600, USA. Email: jsbm@ams.sunysb. edu.

Received May 28, 1987; revised December 15, 1990. Communicated by Bernard Chazelle.

an open problem to devise a worst-case subquadratic time algorithm for finding shortest Euclidean paths.

Our Problem. We consider here the L_1 (rectilinear) version of the two-dimensional shortest path problem: Determine a path of minimum L_1 length from s to t that avoids the interiors of a set of disjoint simple polygonal obstacles described by a set of n vertices. We allow an arbitrary polygonal path from s to t, but we measure the lengths of each of its segments according to the L_1 metric. A closely related problem that is also solved by our method is that of finding a shortest *rectilinear* path from s to t. A rectilinear path is one that is always parallel to either the x- or the y-axis, and its length can be measured either by Euclidean distance or L_1 distance, since these lengths will be the same. The rectilinear path problem naturally arises in certain wire-routing applications in which one is interested in finding the shortest path for a wire that is constrained to avoid a given set of components and must always run up, down, left, or right.

Previous Work. Larson and Li [LL] studied the problem of finding all minimal rectilinear distance paths among a set of m origin-destination pairs in a plane with polygonal obstacles. Their algorithm runs in time $O(m(m^2 + n^2))$, which specializes to $O(n^2)$ for the case of only one origin and one destination. The special case in which all obstacles are rectangles has been solved in optimal time $\Theta(n \log n)$ [DLW] by exploiting the monotonicity of shortest paths. Furthermore, [DLW] show that $\Omega(n \log n)$ is a lower bound on the time complexity in the case of rectangular obstacles, implying the same lower bound for our more general problem.

Our Results. In this paper we provide the first subquadratic time algorithm for the general problem (with polygonal obstacles). Our algorithm runs in time $O(E \log n)$ and space $O(E)$, where E is the number of "events" in our algorithm and is related to a function $g(n)$ that bounds the maximum number of ones in a binary n-by-n matrix that obeys certain sparsity conditions. Appealing to a recent result of Bienstock and Györi [BG], which shows that $g(n) = O(n \log n)$, we conclude that the running time of our algorithm is no worse than $O(n \log^2 n)$, nearly achieving the known lower bound and considerably improving the previous best-known quadratic bound. We suspect that $E = O(n)$. If our conjecture is true, then the algorithm we present here is actually *optimal*, running in time $O(n \log n)$.

We should note that an earlier draft of this paper [Mi2] claimed a bound of $O(n \log n/\log \log n)$ on E, which relied on bounding the number of ones in a matrix without an "L-quadrilateral" pattern. The bound used a complex combinatorial argument in which there was found an error. While we still believe that L-quadrilateral-free matrices obey the stated upper bound, the proof eludes us at this time.

As in [DLW], our algorithm can solve the following query form of the problem: Given a source point s, we build a structure, a *Shortest Path Map* (as originally defined in [LP]), such that, for any query point t, we can find the length of a shortest path from s to t in $O(\log n)$ time (by point location within the map), and we can output a shortest path in additional time $O(k)$, where k is the number of "turns" in the path.

Our results generalize to the case of "fixed orientation metrices" [WWW], of which the L_1 and L_∞ metrics are special cases, and to multiple source points. Thus, our algorithm yields an almost-optimal algorithm for computing the Voronoi diagram according to any fixed orientation metric of a set of sites in a two-dimensional polygonal space. An immediate consequence of this result is an efficient algorithm ($O(n\varepsilon^{-1/2} \log^2 n)$ time and $O(n\varepsilon^{-1/2})$ space) for finding ε-optimal Euclidean shortest paths or Voronoi diagrams among obstacles in the plane. These bounds compare favorably to the recent results of [Cl], who finds an ε-optimal path in time $O(n/\varepsilon + n \log n)$, after spending $O((n/\varepsilon) \log n)$ time to build a data structure of size $O(n/\varepsilon)$.

Our Method. We use a technique called a "continuous Dijkstra algorithm" [Mi1], [MMP], [MP], which considers the effects of sweeping an advancing "wavefront" from a source point s to all points of free space $\mathscr{F}$. (The *wavefront* at distance D is the set of points p of $\mathscr{F}$ for which the shortest path length from s to p is D.) In order to simulate the advancement of the wavefront, we must update our data structure at each of E *events*. The special property of the L_1 metric that makes it easy to keep track of the propagating wavefront is the fact that it is always composed of straight-line segments that are diagonal (oriented at $\pm 45°$). This means that determining events in the continuous Dijkstra method involves answering *segment dragging queries* of the forms to be discussed in Section 3. Thanks to the ingenious data structures of Chazelle [Ch1], [Ch2], we are able to perform the necessary queries and updates in $O(\log n)$ time for each event, while using only linear storage.

The proof of the upper bound on the number of events uses a charging scheme that leads to the study of the maximum number of ones that can appear in a binary matrix that is not allowed to contain certain "violated patterns." Our research points to a very interesting class of problems, intimately related to extremal graph theory, involving the study of matrices that are sparse due to a set of violated patterns. We provide the first application of these combinatorial bounds to geometric problems. One might think of our method as a two-dimensional generalization of the method of Davenport–Schinzel sequences (e.g., see [Sh], [SCKLPS]), which involves bounding the length of a (one-dimensional) sequence of symbols in which certain patterns (alternations) are known not to occur. Recently, in fact, this method has been applied to other problems in geometry. [PaS] have used the sparse matrix method to bound the size of the queue in the Bentley–Ottman line sweeping algorithm, and [CEGS] have used the method to bound the number of cycles that appear in certain arrangements of lines in space. We expect to see continued application of this technique to other problems in combinatorial and computational geometry.

Remark. Since the original writing of this paper, [CKV] have independently obtained an algorithm with time and space bounds similar to ours, although by very different methods. Also, [Wi] has recently obtained bounds (by a similar method to that of [CKV]) that are nearly optimal, especially in some special cases. Both [CKV] and [Wi] use a technique that involves computing a sparse "shortest paths preserving" graph, which is guaranteed to include shortest paths between pairs of vertices.

2. Preliminaries. We assume that "free" space is a (multiply) connected, closed and bounded set $\mathscr{F}$ given by a simple polygon with a set of disjoint (open) simple polygonal holes ("obstacles"). Assuming that there is an outer bounding polygon is not essential, but it simplifies the presentation slightly and can be made without loss of generality. We denote the set of all obstacle vertices by $\mathscr{V}$ and let $n = |\mathscr{V}|$ be the combinatorial size of the problem instance. For a vertex $p \in \mathscr{V}$, we denote by *p.pred* and *p.succ* the predecessor and successor vertices in the list of vertices defining the obstacle containing p, where the list is given in counterclockwise order about the boundary of the obstacle (i.e., in such a way that free space is on the right when the list is traversed).

We let $s \in \mathscr{F}$ be a "source" (or starting) point, and we let $t \in \mathscr{F}$ denote any "destination" (or goal) point. Without loss of generality, we can assume that $s \in \mathscr{V}$, since we can think of there being a "point obstacle" at s.

For simplicity of discussion, we will make the following *General Positioning Assumption* (GPA) about the data of the problem: *No three vertices of* $\mathscr{V}$ *are collinear, and no two vertices lie along a horizontal, vertical, or diagonal* ($\pm 45°$) *line.* We make this assumption without loss of generality, since we can always perturb the data slightly to achieve the GPA.

Given two points $q_1 = (x_1, y_1)$ and $q_2 = (x_2, y_2)$ in the plane, the L_1 (*rectilinear*) distance between them is defined to be

$$d(q_1, q_2) = d_1(q_1, q_2) = |x_1 - x_2| + |y_1 - y_2|.$$

We measure paths according to their rectilinear length. Throughout the paper, unless otherwise specified, we will use the terms distance and length to refer to rectilinear distance and rectilinear length.

A path Π from s to t will be called an (s, t)-*path*. A path is said to be *feasible* if it lies within $\mathscr{F}$. A feasible (s, t)-path that has minimum rectilinear length among all feasible (s, t)-paths is called a *shortest path* from s to t. We denote the length of a shortest (s, t)-path by $l(s, t)$. Note that, while the length of a shortest (s, t)-path is unique, there will in general be (uncountably) many shortest (s, t)-paths.

Our problem is to find a shortest path from s to t. We will in fact be finding a *shortest path tree rooted at s* (denoted SPT(s)) and we will show how it can be extended to a *shortest path map rooted at s* (denoted SPM(s)). An SPT(s) is a tree, with nodes corresponding to vertices $\mathscr{V}$ and edges corresponding to line segments between vertices, such that the polygonal path from s to $v \in \mathscr{V}$, obtained by joining nodes (vertices) along the (unique) path of SPT(s) from node s to node v, is a shortest path. Shortest path trees rooted at s are not unique. The SPT(s) that we construct is a specific one that we can define precisely once we have more terminology.

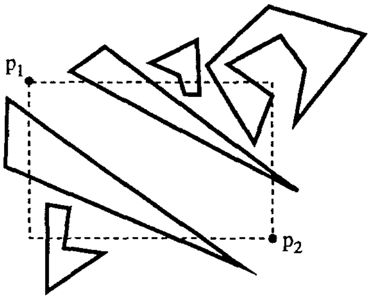

Fig. 1. Defining immediately accessible.

Given two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ in the plane, we say that p_1 is *northwest of* p_2 if $x_1 \leq x_2$ and $y_1 \geq y_2$. Similar definitions apply to the terms *southeast*, *southwest*, and *northeast*. We abbreviate these four directions as SE, NE, NW, SW, and we will use the symbol δ to indicate a direction $\delta \in \mathscr{D} = \{\text{SE, NE, NW, SW}\}$.

We define the (closed) rectangle *cornered* at p_1 and p_2 to be the set $R(p_1, p_2) = \{(x, y)\colon \min\{x_1, x_2\} \leq x \leq \max\{x_1, x_2\}$ and $\min\{y_1, y_2\} \leq y \leq \max\{y_1, y_2\}\}$. We will say that point $p_1 \in \mathscr{F}$ is *immediately accessible* from $p_2 \in \mathscr{F}$ if p_1 and p_2 are contained in the same connected component of $R(p_1, p_2) \cap \mathscr{F}$, and there are no vertices (other than possibly p_1 and p_2) on the boundary of this component. (Thus, the portion of the wavefront propagating out from p_2 that hits point p_1 encounters no vertices before hitting p_1.) Note that if p_1 is immediately accessible from p_2, then p_1 and p_2 must be mutually visible (meaning that $\overline{p_1 p_2} \subset \mathscr{F}$). See Figure 1. For $p \in \mathscr{F}$, we define $acc^{\delta}(p)$, $\delta \in \mathscr{D}$, to be the set of all points q that lie in direction δ from p and that are immediately accessible from p. See Figure 2 for an example.

We can now define the particular shortest path tree that we construct more precisely. For each vertex $v \in \mathscr{V}$, there are possibly many vertices $u \in \mathscr{V}$ with the property that v and u are mutually visible and $l(s, v) = l(s, u) + d(u, v)$. Each such u is a possible parent of v in some shortest path tree rooted at s. To make specific the unique shortest path tree, SPT(s), that is constructed by our algorithm, we define the *parent* of v to be the leftmost vertex $u \in \mathscr{V}$ with the properties that

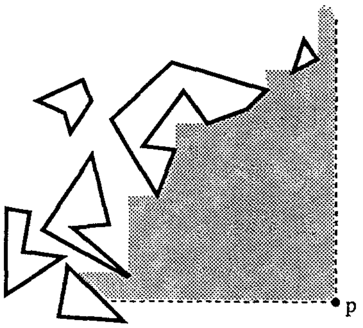

Fig. 2. The set of points, $acc^{NW}(p)$, is shown shaded.

$l(s, v) = l(s, u) + d(u, v)$ and v is immediately accessible from u. (Note that, by the definition of immediate accessibility, or by the GPA, there can be only one leftmost point u with the two specified properties.) The parents of the vertices $\mathscr{V}$ define the tree SPT(s). We now make the following simple observation:

LEMMA 1. *If there exists a feasible (s, t)-path, then there exists a shortest (s, t)-path, $s = v_0, v_1, \ldots, v_k = t$, such that v_{i+1} is immediately accessible from v_i for each $i = 0, 1, \ldots, k-1$, and $v_i \in \mathscr{V}$.*

PROOF. If there exists a feasible (s, t)-path, then there exists a shortest (s, t)-path Π that passes through a maximal number of obstacle vertices. Path Π visits a sequence of obstacle vertices, $s = v_0, v_1, \ldots, v_k = t$. We claim that v_{i+1} is immediately accessible from v_i. If this were not the case, then the connected component of $R(v_i, v_{i+1}) \cap \mathscr{F}$ that contains v_i and v_{i+1} must contain some other obstacle vertex. Let $u \in \mathscr{V}$ be such a vertex that is closest to the line segment $\overline{v_i v_{i+1}}$. Then, Π could be modified to include the vertex u, without increasing its L_1 length, contradicting the maximality of the number of vertices along Π. □

A shortest path tree is a structure giving shortest paths from the source s to all other vertices. A *shortest path map* with respect to s, SPM(s), is a generalization of this structure to allow one to answer shortest path queries from s to any point $t \in \mathscr{F}$ by performing a point location of t within the map. Shortest path maps for Euclidean shortest path problems have been studied in [LP], [Mi1], [Mi3].

Before giving a brief description of the structure of SPM(s), we need some definitions. The bisector $b(p_1, p_2)$ between two points p_1 and p_2, having "weights" w_1 and w_2, is defined to be the locus of all points q such that $d(q, p_1) + w_1 = d(q, p_2) + w_2$. In our problem, the weight associated with a point will be the length of a shortest path from the source s to the point. Various cases are shown in Figure 3, where it is observed that the bisector may consist of an entire quadrant of the plane (as in cases (c) and (d)). This introduces some ambiguity if we wish to confine attention to one-dimensional bisectors, as there are an infinite number that would suffice in cases (c) or (d). We choose (arbitrarily) to resolve this ambiguity in favor of *vertical* bisectors (shown as thick solid lines in the figure).

We define a shortest path map SPM(s) to be a partitioning of the plane into *cells* $C(r)$ $(r \in \mathscr{V})$ with the following property: if $t \in C(r)$, then t is immediately accessible from r and $l(s, t) = l(s, r) + d(r, t)$. Vertex r (the *root* of cell $C(r)$) is on the boundary of $C(r)$, and all points of $C(r)$ are immediately accessible from r. If cell $C(r)$ is adjacent to cell $C(r')$, then the boundary shared by $C(r)$ and $C(r')$ is a subset of the bisector $b(r, r')$ between r and r'. See Figure 4 for an example of SPM(s). (The GPA is violated in this picture.)

We assume that SPM(s) is endowed with pointers from each vertex to its parent in SPT(s). The subdivision is defined such that if $t \in C(r)$, then a shortest path from s to t is obtained by following a shortest path from s to r (e.g., following the path given by SPT(s)), and then proceeding from r to t along any path of length $d(r, t)$ in $C(r)$ (e.g., the straight line segment from r to t).

Thus, if we are given SPM(s), the length of a shortest path from s to t can be

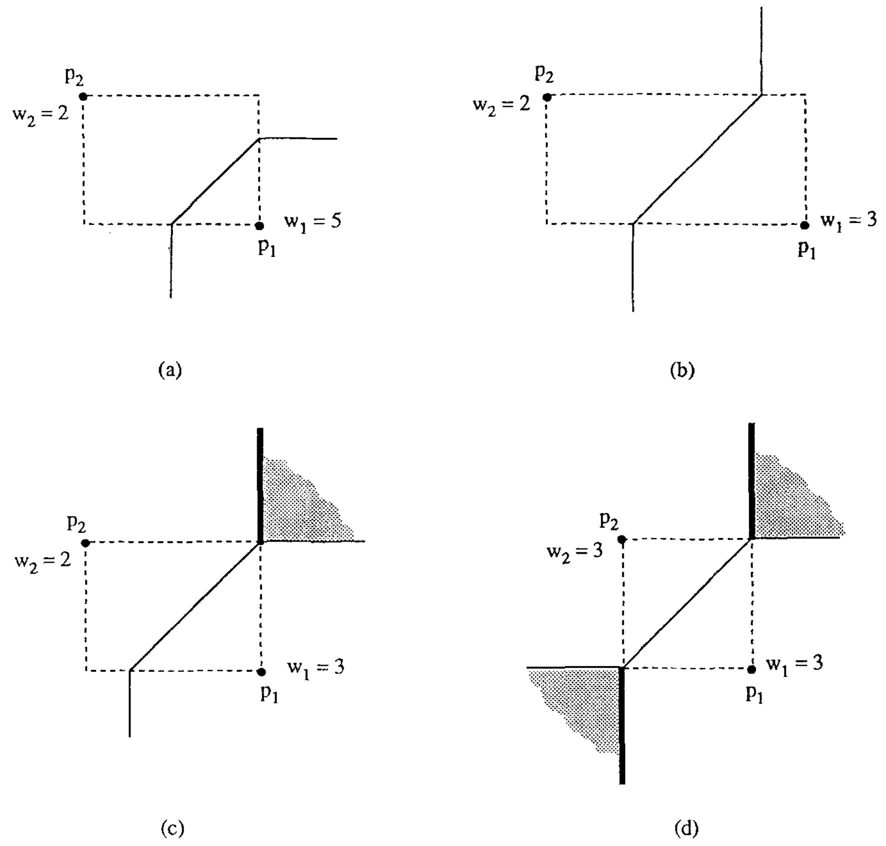

Fig. 3. The bisector between the weighted points p_1 and p_2.

found in $O(\log n)$ time by solving a point location problem [EGS], [Ki], [Pr] to determine the cell $C(r)$ containing t. The length will be $l(s, t) = l(s, r) + d(r, t)$. Then, a shortest path from s to t can be backtraced in time $O(k)$, where k is the number of vertices in the shortest path that we trace.

3. Segment Dragging Queries.

Our main algorithm's efficiency comes from being able to preprocess $\mathscr{F}$ so that "segment dragging" queries of the following form can be answered efficiently: Determine the next point or segment "hit" by a query segment $\overline{qq'}$ when it is "dragged" in a specified manner. We assume that segments are dragged in such a way that the segment being dragged remains parallel to a fixed direction, while the endpoints of the segment slide along two straight paths, which we call *tracks*.

Parallel Tracks. The familiar "range search for a min (max)" problem is a special case of a segment dragging problem in which we preprocess a set of points $P = \{p_1, p_2, \ldots, p_n\}$ so that we can report "hits" by a horizontal query segment

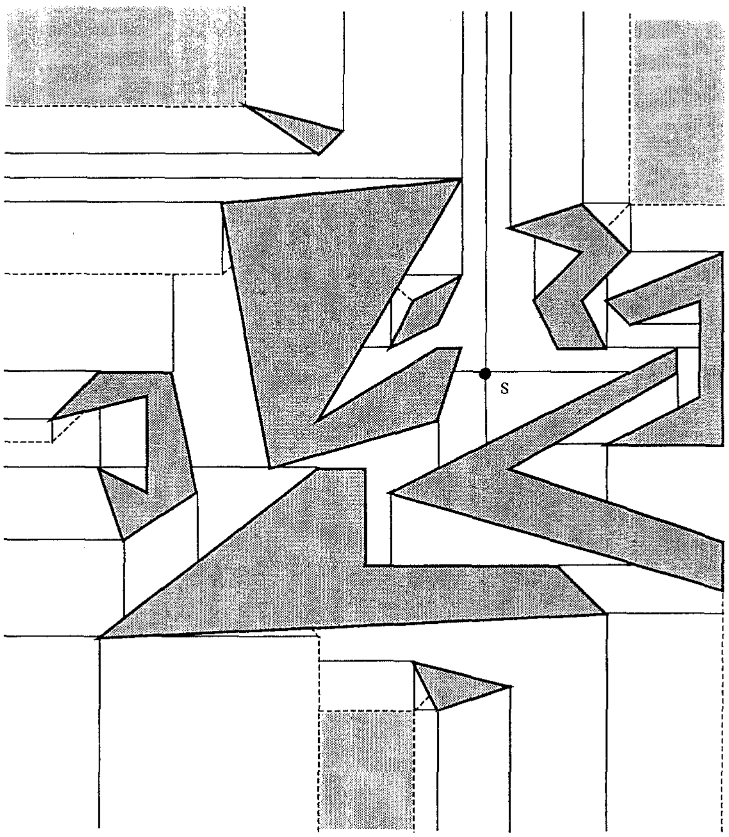

Fig. 4. A shortest path map SPM(s).

$\overline{qq'}$ being dragged in the direction of positive (negative) y, with its endpoints sliding on two vertical lines. See Figure 5(a). In the usual rectangular range query problem, one wishes to say something about the set of points inside a given query rectangle (such as "report them," "count them," or "find the one with minimum (maximum) y-coordinate"). In our special case of the segment dragging problem, the rectangular query region is a semi-infinite vertical strip whose base is the query segment $\overline{qq'}$. The best known solution to the range query for a max problem is that of Chazelle [Ch1], where this problem is solved in preprocessing time $O(n \log n)$ and query time $O(\log n)$, with a space complexity of $O(n \log^{\varepsilon} n)$, where ε is an arbitrarily small positive real number. (Alternately, the (time, space) complexities can be $(n \log^{1+\varepsilon} n, n)$ or $(n \log n \log \log n, n \log \log n)$.) For the special case needed to answer our segment dragging problem, however, Chazelle is able to achieve query times of $O(\log n)$ with a space complexity of $O(n)$ [Ch2].

We also need to answer segment dragging queries for inclined segments (always at some fixed angle θ) that we drag with endpoints sliding along vertical tracks. This problem is easily seen to be equivalent to the horizontal segment dragging

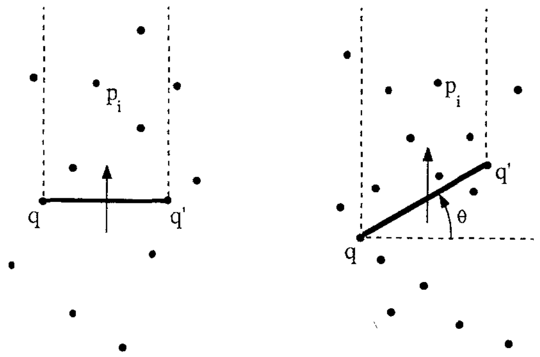

Fig. 5. (a) Dragging a horizontal segment. (b) Dragging an inclined segment.

problem, as all that is needed is to transform the coordinates of the collection of points to the coordinate system defined by the y-axis and the (oriented) line at angle θ (see Figure 5(b)). Thus, after $O(n \log n)$ preprocessing, inclined segment dragging queries can be answered in time $O(\log n)$.

Nonparallel Tracks: Dragging Out of a Corner. We will also have need of segment dragging queries in the case that the tracks are not parallel, but at fixed orientations, and the segment is dragged "out of the corner" defined by the intersecting tracks: From a query point r, find the first point hit by a segment $\overline{qq'}$ at inclination θ that is being dragged parallel to itself so that its endpoints q and q' slide along the rays l_1 and l_2 (which are rooted at point r and have inclinations φ_1 and φ_2, respectively). We assume that the segment $\overline{qq'}$ starts being dragged from a position such that triangle $\triangle rqq'$ contains no point $p_i \in P$; that is, we can think of q and q' as being very close to r on the rays l_1 and l_2. Thus, the query is two-dimensional since it is fully specified by giving the point r. (The angles θ, φ_1, and φ_2 are given and fixed.) See Figure 6. This segment dragging problem is solved by converting it to a point location problem in an appropriate planar

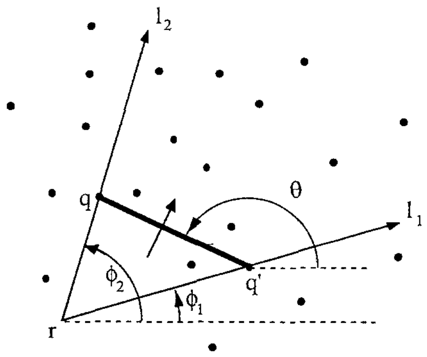

Fig. 6. Dragging a segment out of a corner along two rays.

subdivision, $S(\theta, \varphi_1, \varphi_2)$, such that an answer to the query is given by doing an $O(\log n)$ time point location query of r in $S(\theta, \varphi_1, \varphi_2)$.

Very briefly, this subdivision is built as follows: We use a sweep line method in which the sweeping line is at inclination θ. Assume, without loss of generality, that the angles φ_1 and φ_2 both lie in the first quadrant, so that our sweep line l moves to the northeast. As l encounters points of P, we update the subdivision. There will be a northeast boundary of the subdivision at any given instant. When a new point p_i is encountered, we extend a ray from p_i in the direction of $\varphi_1 + \pi$ and another ray in the direction of $\varphi_2 + \pi$. Where these rays intersect the northeast boundary of the current subdivision, we mark points $q_{1,i}$ and $q_{2,i}$. The segments $\overline{p_i q_{1,i}}$ and $\overline{p_i q_{2,i}}$ are added to the subdivision, the northeast boundary is updated accordingly, and we continue sweeping the line l. This algorithm builds the subdivision $S(\theta, \varphi_1, \varphi_2)$ in time $O(n \log n)$, as each update of the northeast boundary requires two $O(\log n)$ binary searches to locate and insert the points $q_{1,i}$ and $q_{2,i}$. An example is shown in Figure 7. Once we have this subdivision, if we locate r in, say, the shaded region of Figure 7, then the next point hit by the segment $\overline{qq'}$ will be p, the upper right vertex of the region. If r lies northeast of the final northeast boundary, then the segment $\overline{qq'}$ can be dragged off to infinity without hitting a point. (The reader is referred to Figure 4.2 of [EOS], where a similar *ru-point subdivision* is illustrated in the solution of "next-point search problems"; the ru-point subdivision corresponds to $S(\pi/2, 0, \pi/4)$ in our notation.)

The algorithm just described for building the subdivision $S(\theta, \varphi_1, \varphi_2)$ is easily extended to include the case of queries in the presence of both points and polygonal obstacles (instead of just points $\{p_1, \ldots, p_n\}$). We assume now that once an endpoint (q or q') of the dragged segment hits the interior of an obstacle edge, that endpoint starts to slide along the edge, still maintaining the segment $\overline{qq'}$ at inclination θ. To handle these queries in the presence of obstacles, we simply modify the algorithm above so that during the line sweep the northeast boundary will contain

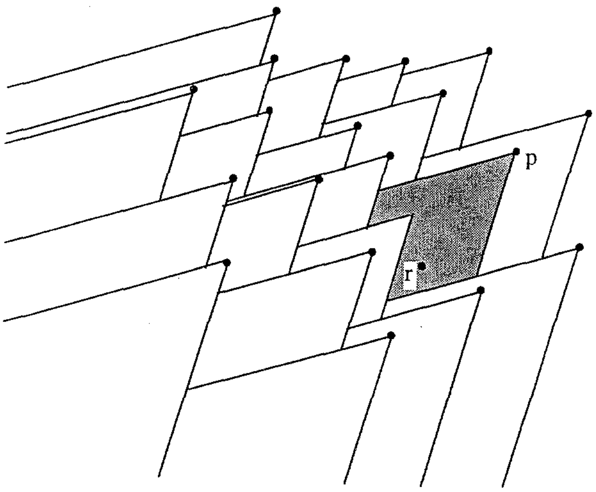

Fig. 7. Subdivision $S(\theta, \varphi_1, \varphi_2)$.

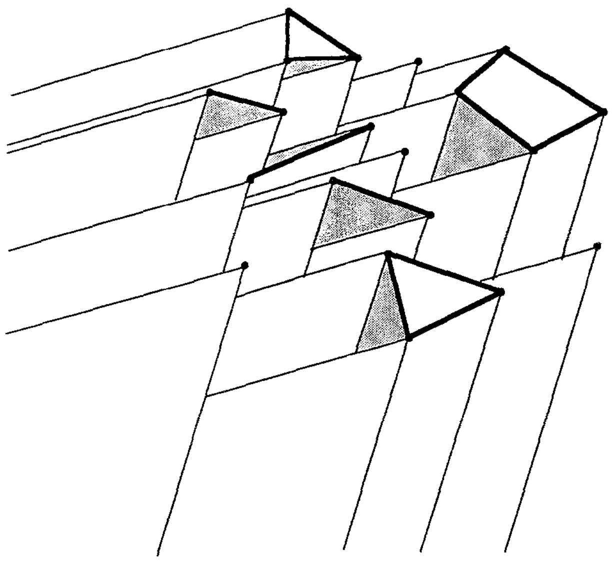

Fig. 8. Subdivision $S(3\pi/4, 0, \pi/2)$ in the presence of obstacles.

segments of the subdivision as well as obstacle segments. Figure 8 shows an example of such a subdivision. Note that if r is located in one of the shaded regions of Figure 8, then both endpoints of $\overline{qq'}$ will hit an obstacle edge (the one forming the northeast boundary of the region), and the segment $\overline{qq'}$ will never hit an obstacle vertex; thus, the obstacles have a "shadowing" effect.

We allow the special cases in which $\theta = \varphi_1$ or $\theta = \varphi_2$. The corresponding subdivisions ($S(\theta, \theta, \varphi_2)$ or $S(\theta, \varphi_1, \theta)$, resp.) are built exactly as above and solve the problem of dragging a ray so that it remains parallel while its endpoint slides along another ray (either l_2 or l_1, resp.). In the presence of obstacles, the ray is allowed to extend only until it first hits an obstacle boundary.

Remark (Application to Voronoi Diagrams). As an aside, note that by building the subdivisions $S(3\pi/4, 0, \pi/2)$, $S(5\pi/4, \pi/2, \pi)$, $S(7\pi/4, \pi, 3\pi/2)$, and $S(\pi/4, 3\pi/2, 2\pi)$ for a given set P of n points, we have, in fact, solved the closest point problem in $O(n \log n)$ time and $O(n)$ space (optimal time and space). To find the closest point to any query point q, we simply have to locate q in each of the four subdivisions and pick the closest of the four resulting choices. (If desired, the four subdivisions can be merged into one subdivision (the Voronoi diagram) within the given time bound, thereby eliminating the need to do four point location queries per q.) This is an alternative algorithm to that given in [LWo] for the construction of the Voronoi diagram in the L_1 (L_∞) metric. While [LWo] give a divide-and-conquer algorithm, we see that the problem can also be solved by our plane-sweep algorithm within the same time and space complexity. Our algorithm also generalizes immediately to the case of fixed orientation metrics, giving a plane-sweep alternative to the divide-and-conquer algorithm of [WWW].

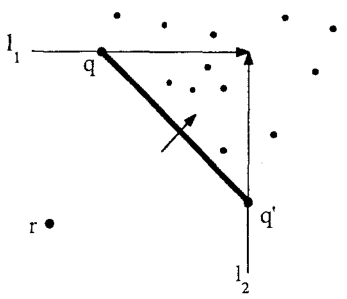

Fig. 9. Dragging a segment into a corner.

Nonparallel Tracks: Dragging into a Corner. The subdivision $S(\theta, \varphi_1, \varphi_2)$ discussed above solves the problem of dragging a segment *out* of a corner; our application needs also to consider the problem of dragging a segment *into* a corner. Consider the situation depicted in Figure 9. The segment $\overline{qq'}$ is dragged so that q and q' slide along the track rays l_1 and l_2 (respectively), and we ask for the first point hit by $\overline{qq'}$ before it "dies" into the corner w. (In our application, we will be working in the presence of obstacles. We will be given that segment $\overline{qq'}$ intersects no obstacle interiors.) This query is a special type of range query for a max in which the query region is a triangle (of known, fixed angles) instead of a rectangle. Unfortunately, we know of no method to answer these queries in $O(\log n)$ time and linear space with $O(n \log n)$ preprocessing. We leave it as an interesting open problem whether queries of this form can be answered efficiently.

In our algorithm, we circumvent this issue by using a combination of queries of the types we know how to solve efficiently (as described above). If an efficient solution can be found for the case of dragging a segment into a corner, the presentation of our algorithm (and its proof) can be simplified.

Summary (Our Application). Our interest in these segment dragging problems will be to examine the effect of a "wavefront" propagating through a collection of polygonal obstacles. Determining which point is hit next by a dragged segment will be critical to selecting the next "event" that occurs as the wavefront moves through the free space. The segment dragging queries of use in our case will be those of dragging segments inclined at angles $\pi/4$ (or $3\pi/4$) along tracks parallel to the coordinate axes (north, south, east, and west). We will also be building the subdivisions $S(3\pi/4, 0, \pi/2)$, $S(5\pi/4, \pi/2, \pi)$, $S(7\pi/4, \pi, 3\pi/2)$, and $S(\pi/4, 3\pi/2, 2\pi)$, so that queries of the form "What is the closest point to r to the northeast?" can be answered in $O(\log n)$ time. Additionally, we will need the subdivisions $S(\pi/4, \pi/4, \pi/2)$, $S(3\pi/4, 3\pi/4, \pi)$, $S(5\pi/4, 5\pi/4, 3\pi/2)$, and $S(7\pi/4, 7\pi/4, 2\pi)$. All of these subdivisions will be understood to include the effect of the obstacles.

4. The Algorithm

The Continuous Dijkstra Paradigm. Our algorithm operates in much the same spirit as the well-known Dijkstra algorithm [Di]. A "signal" is propagated from the source s to all other points of free space $\mathscr{F}$. Once an obstacle vertex p receives the signal for the first time, it propagates it further. Point p is considered to be *permanently labeled* with the time, $d(p)$, at which it first received the signal. The label $d(p)$ is the length, $l(s, p)$, of shortest paths from s to p, and we refer to it as the *depth* of p. We continue to propagate signals until every obstacle vertex has received the signal. We need only keep track of discrete *events* that take place as the signal propagates through the free space $\mathscr{F}$.

In effect, then, our method is to sweep the plane with an advancing "wavefront." The *wavefront at distance D, rooted at s*, is the set of points p of $\mathscr{F}$ for which $l(s, p) = D$. The special property of the L_1 metric that makes it relatively easy to keep track of the propagating wavefront is the fact that it is always composed of straight line segments that are diagonal (oriented at $\pm 45°$). Even with this nice structure, we find it difficult to keep track of the exact wavefront. Instead, we consider "wavefronts" propagating in each of the four directions $\delta \in \mathscr{D} = \{\text{SE, NE, NW, SW}\}$, and we allow these wavefronts to "run over" each other. In particular, we do not actually keep track of when one portion of the wavefront (say, propagating northwest) first runs into another portion of the wavefront (say, propagating southeast), as these events may be difficult to detect. Instead, we only detect collisions of wavefronts with *obstacles*, doing the necessary "clipping" of wavefronts only after we discover that two of them have hit the same obstacle vertex.

We allow a vertex to be hit from each of the four directions, possibly several times. By allowing wavefronts to sweep over the same portion of the plane more than once, we are being conservative (with respect to "clipping") but potentially wasteful (with respect to inflating the number of events). The goal of our complexity analysis (Section 6) is to show that our wasteful strategy does not yield a huge number of events. In particular, we will show that the number, E, of events has an upper bound of $O(n \log n)$ and that the update time per event is only $O(\log n)$.

By considering wavefronts separately in each of the four directions, we have decomposed the problem in much the same way as was done in [MMP] for the problem of computing discrete geodesics, where points on an edge were "hit" independently by waves coming from each side of the edge. In the discrete geodesic problem, two subdivisions of each edge were built, according to the structure of the shortest paths to points on the edge, one subdivision corresponding to each of the two directions from which the paths may be incident to the edge. (Paths may hit an edge by passing through either of the two faces adjacent to the edge.) In the problem considered here, the shortest path to a point may come from one of four directions $\delta \in \mathscr{D} = \{\text{SE, NE, NW, SW}\}$.

Dragged Segments. First, we define more formally what is meant by a *dragged segment*. A dragged segment is a line segment that makes up a portion of a wavefront. There are nine basic types of dragged segments of each of the four inclinations ($\theta \in \{\pi/4, 3\pi/4, 5\pi/4, 7\pi/4\}$, corresponding to propagation SE, NE, NW, and SW, respectively), depending on the nature of the left and right rays. Each track ray may be a horizontal ray, a vertical ray, or an obstacle edge. These cases are illustrated in Figure 10 for a dragged segment of inclination $5\pi/4$

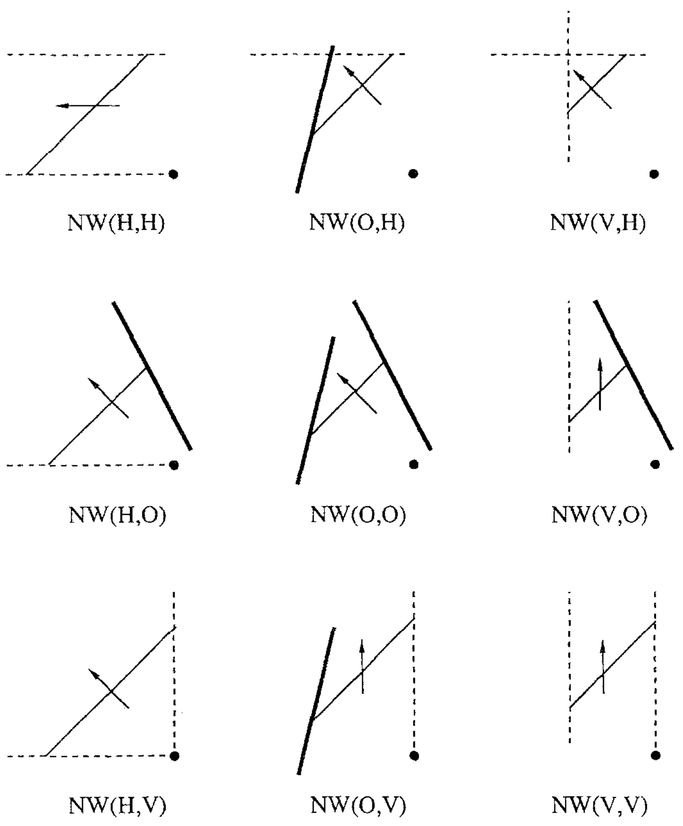

Fig. 10. Nine cases of dragged segments propagating NW.

(propagating NW), where names are assigned to each of the cases. For example, NW(left-ray, right-ray) indicates a NW dragged segment whose left track ray is of type left-ray $\in \{H, V, O\}$ and whose right track ray is of type right-ray $\in \{H, V, O\}$, with "H," "V," and "O" indicating a track ray that is horizontal, vertical, or obstacle, respectively. All points on a dragged segment are at the same L_1 distance from the root of the segment. This is why the dragged segments are thought of as "wavefronts," and, in this sense, we can think of the roots of dragged segments as "virtual" sources that act to propagate the wavefront from the original source s.

The data structure for a dragged segment has the following information associated with segment $\overline{qq'}$:

- Its inclination, θ, which is the angle from the positive x-axis to the oriented segment $\overline{q'q}$, and will always be either $\pi/4$, $3\pi/4$, $5\pi/4$, or $7\pi/4$ in the case of the L_1 metric.
- Its endpoints q and q', which are the positions of the segment's endpoints at the moment the segment is first instantiated, before it starts being "dragged."
- Its left and right *track rays*—these are the rays along which q and q' must be

dragged, and they may be horizontal or vertical rays through free space or rays containing obstacle edges.

- The *stop points* L and R of the left and right track rays—these are the first obstacle points "hit" by the left and right track rays. (If the track rays intersect each other at point u before they hit obstacles, then $L = R = u$, where u is the inside corner of the corresponding segment dragging query.)
- Its *root* r, which is an obstacle vertex that is responsible for propagating the portion of the wavefront to which the segment belongs.
- Its *contact list*, which is the set of (at most four) obstacle edges that the dragged segment touches (including the obstacle edges on which its endpoints may be sliding).
- Its *event position* $\overline{q_e q_e'}$, which is the next position of the segment at which the contact list changes.
- Its *event point* p, which is the point that is responsible for the change in the contact list when the segment reaches its event position. The event point p must lie on the boundary of an obstacle, and it will either be a stop point or a vertex. (The GPA allows us to assume that the event point is *unique*.)
- The *event distance* $d(r) + d(r, p)$, which is simply the distance from s at which the event point is encountered by the segment.

Types of Events. We can now describe the three basic types of events that can occur:

I. When the event point p is one of the stop points (L or R), we say that the segment has *reached its stop point.*
II. When p is interior to the dragged segment in its event position, we say that there has been an *interior collision.*
III. When p is a vertex encountered by an endpoint of the dragged segment, we say there has been an *endpoint collision.*

We say that a root r *hits* or *collides with* an obstacle vertex p if some dragged segment rooted at r has p as an event point.

We define the *region swept out* by a dragged segment as the set of points in the plane that are intersected by the segment at some position of the segment between the time it is instantiated (at $\overline{qq'}$) and the time it hits its event point p.

Other Data Structures. Our algorithm maintains a list of "active" dragged segments in a priority queue (called the *event queue*), with the segments ordered according to their event distances. The *next event* is the dragged segment whose event distance is minimum and is obtained by popping the queue. In the case of ties, we can order the event distances by the lexicographic ordering of the x- and y-coordinates of their roots.

Each obstacle vertex $v \in \mathscr{V}$ has associated with it a sorted list, the *SE-hit list*, $\mathscr{R}^{SE}(v) = \{r_1, \ldots, r_N\}$ of roots r_i of dragged segments that are southeast of v and are such that the dragged segment has "hit" point v (i.e., v has been an event point for a dragged segment rooted at r_i, and this event has already occurred). The points of $\mathscr{R}^{SE}(v)$ are kept in order of increasing y-coordinates (which is also the order of increasing x-coordinates since the points of $\mathscr{R}^{SE}(v)$ form a staircase path going northeast). Similar definitions apply to the hit lists $\mathscr{R}^{NE}(v)$, $\mathscr{R}^{NW}(v)$, and $\mathscr{R}^{SW}(v)$. Any particular list $\mathscr{R}^{\delta}(v)$, $\delta \in \mathscr{D} = \{SE, NE, SW, NW\}$, could have $O(n)$ entries, so it appears that the space requirement could become quadratic (and that the time to build the lists could become $O(n^2 \log n)$); however, the total size of all lists will be bounded above by E, the number of events, which we show (Section 6) to be almost linear.

Also associated with each obstacle vertex $v \in \mathscr{V}$ is a *permanent label*, $d(v)$, which, at the conclusion of the algorithm, gives the length $l(s, v)$ of shortest paths from s to v. Initially, $d(v) = +\infty$ for all $v \in \mathscr{V}$. We say that v has been *permanently labeled* if $d(v) < +\infty$. We say that a *non*-vertex point x has been permanently labeled if it lies in the region swept out by some dragged segment. Each vertex v also has a pointer, parent(v), which, at the conclusion of the algorithm, points to the parent of v in SPT(s). Initially, parent(v) = NIL.

The algorithm proceeds as follows:

ALGORITHM

(0) (**Initialize**) Permanently label s with 0. Initialize SEGLIST to be the set of four dragged segments rooted at s of types NE(V, H), NW(H, V), SW(V, H), and SE(H, V). Determine the next events for each of these, and initialize the event queue to consist of these four events (along with their distance labels).

(1) (**Main Loop**) While there is an entry in the event queue that has a finite label, remove one, $\overline{qq'}$, which corresponds to the smallest label and call the procedure *Propagate* ($\overline{qq'}$).

The details of the algorithm are contained in the procedure *Propagate*. Intuitively, to *propagate* a dragged segment $\overline{qq'}$ means to allow a "wavefront" of signals to advance past the event point p in the direction that $\overline{qq'}$ is being dragged. This usually involves creating new dragged segments corresponding to the advancing wavefront, or in "clipping" the segment $\overline{qq'}$ so that the continuation of the sweep of the wavefront does not sweep over regions of the plance that we know to be better reached from some other root (from the same direction).

The procedure *Propagate* does different things depending on whether the next event is of Type I, II, or III and on whether or not the event point p has already been permanently labeled. The cases are illustrated in Figures 11, 12, and 13. We now give more details.

Procedure *Propagate* ($\overline{qq'}$)

(0) Let $p = (x_p, y_p)$ be the event point, let $r = (x_r, y_r)$ be the root, and let L and R be the left and right stop points of $\overline{qq'}$. $\overline{q_e q_e'}$ is the event position of the dragged segment (i.e., its position when it is in contact with p). Assume (without loss of generality) that the segment $\overline{qq'}$ is inclined at angle $5\pi/4$ and propagating northwest, so that r is southeast of p. According to the nature of the event point p, go to one of the cases (1)–(3) below.

(1) (**p is a stop point**) (Type I event) If $p = L$, then insert a dragged segment (of

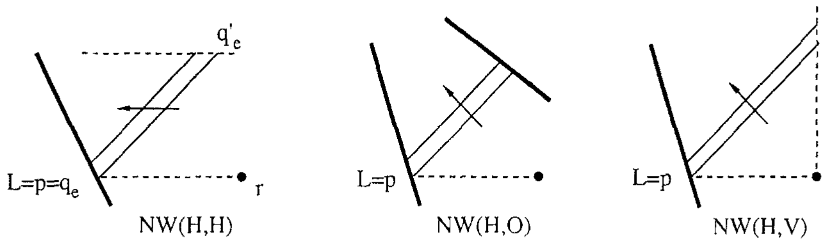

Fig. 11. Type I event: Reaching a stop point.

type NW(O, H), NW(O, O), or NW(O, V)) rooted at r whose right track ray is that of $\overline{qq'}$ and whose left track ray is the obstacle segment containing L. Otherwise ($p = R$), insert a dragged segment (of type NW(H, O), NW(O, O), or NW(V, O)) rooted at r whose left track ray is that of $\overline{qq'}$ and whose right track ray is the obstacle segment containing R. Figure 11 shows the cases in which a NW(H, ·) segment hits a left stop point $p = L$. (Charge point p.)

(2) (**p is not yet permanently labeled**) Permanently label p with the distance $d(p) = d(r) + d(r, p)$ and set parent$(p) = r$. Set $\mathscr{R}^{SE}(p) = \{r\}$. Insert new segments according to the cases below. (Charge point p.)

  (a) (**p is interior to $q_e q'_e$**) (Type II event) In this case, the wavefront "splits" at p, and we insert two or more new dragged segments, according to the orientation of the segments (p, *p.succ*) and (p, *p.pred*). Below, we enumerate the possible cases for the orientation, θ, of the (directed) segment (p, *p.succ*) and the corresponding new dragged segments that continue the propagation to the *right* of the splitting point p; the case analysis based on the orientation of the segment (p, *p.pred*) for how to propagate dragged segments to the *left* of p is similar.

    (i) $\theta \in (\pi/4, \pi/2)$. Insert a NW dragged segment rooted at r whose right track ray is that of $\overline{qq'}$ and whose left track ray is the obstacle segment (p, *p.succ*).

    (ii) $\theta \in (\pi/2, \pi)$. Insert a NW dragged segment rooted at r whose right track ray is that of $\overline{qq'}$ and whose left track ray is the northward ray through p. Also insert a dragged segment of type NW(O, V), of initial length zero, rooted at p whose right track ray is the northward ray through p and whose left track ray is the obstacle segment (p, *p.succ*).

    (iii) $\theta \in (\pi, 5\pi/4)$. Insert a NW dragged segment rooted at r whose right track ray is that of $\overline{qq'}$ and whose left track ray is the northward ray through p. Also insert dragged segments of types NW(H, V) and SW(O, H), rooted at p, whose initial lengths are zero.

  Figure 12 shows various cases.

  (b) ($p = q_e$) (Type III event) There are four cases, depending on the orientation, θ, of the (directed) segment (p, *p.succ*), which are almost identical to those considered in (a) above.

    (i) $\theta \in (\pi/4, \pi/2)$. Insert a NW dragged segment rooted at r whose right track ray is that of $\overline{qq'}$ and whose left track ray is the obstacle segment (p, *p.succ*).

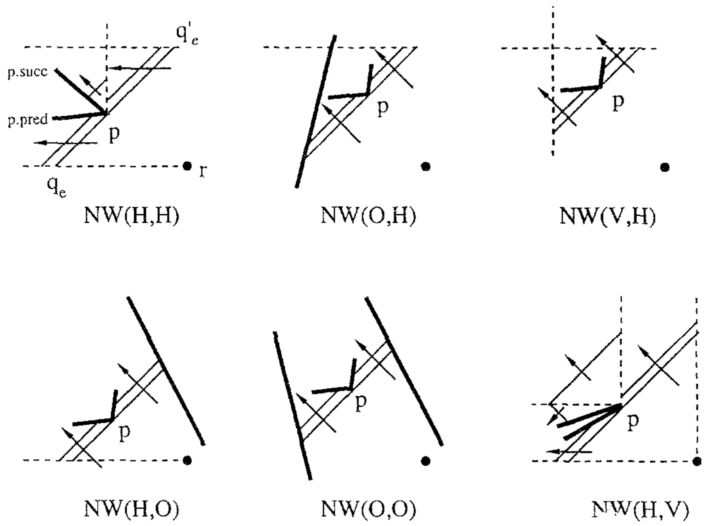

Fig. 12. Type II event: Interior collision.

(ii) $\theta \in (\pi/2, \pi)$. Insert a NW dragged segment rooted at r whose right track ray is that of $\overline{qq'}$ and whose left track ray is the northward ray through p. Also insert a dragged segment of type NW(O, V), of initial length zero, rooted at p whose right track ray is the northward ray through p and whose left track ray is the obstacle segment $(p, p.succ)$.

(iii) $\theta \in (\pi, 3\pi/2)$. Insert a NW dragged segment rooted at r whose right track ray is that of $\overline{qq'}$ and whose left track ray is the northward ray through p. Also insert dragged segments of types NW(H, V) and SW(O, H), rooted at p, whose initial lengths are zero.

(iv) $\theta \in (3\pi/2, 2\pi)$. Insert a NW dragged segment rooted at r whose right track ray is that of $\overline{qq'}$ and whose left track ray is the northward ray through p. Also insert dragged segments of types NW(H, V), SW(V, H), and SE(O, V), rooted at p, whose initial lengths are zero.

Figure 13 shows various cases.

(c) $(p = q'_e)$ (Type III event) Similar to case 2(b).

(3) (**p is already permanently labeled, and** $\mathscr{R}^{SE}(p) = \varnothing$) This is the case in which p has already been hit from some direction, but not yet hit from the southeast. Set $\mathscr{R}^{SE}(p) = \{r\}$. Insert new segments according to the cases below. (Charge point p.)

(a) (**p is interior to** $q_e q'_e$) (Type II event) As in step 2(a) above, we split the wavefront at p, and consider cases for propagation on each of the two sides of p. Again, we present only the details of the cases for the propagation to the *right* of p; the left side is handled similarly. The only difference between what we do now and what we did before is that we omit inserting the dragged segments that are rooted at p, since further propagation from p has been considered already when p was first hit. We have two cases, depending on the orientation, θ, of the (directed) segment $(p, p.succ)$:

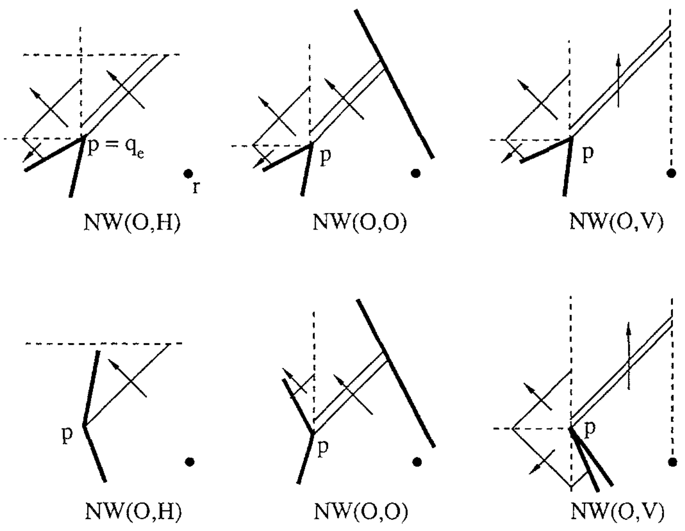

Fig. 13. Type III event: Endpoint collision.

(i) $\theta \in (\pi/4, \pi/2)$. Insert a NW dragged segment rooted at r whose right track ray is that of $\overline{qq'}$ and whose left track ray is the obstacle segment $(p, p.succ)$.

(ii) $\theta \in (\pi/2, 5\pi/4)$. Insert a NW dragged segment rooted at r whose right track ray is that of $\overline{qq'}$ and whose left track ray is the northward ray through p.

(b) $(p = q_e)$ (Type III event) There are two cases, depending on the orientation, θ, of the (directed) segment $(p, p.succ)$. These are almost exactly the same as the dragged segments that are inserted to the right of p in case 3(a) above, except that the second case now applies to the range $\theta \in (\pi/2, 2\pi)$.

(c) $(p = q'_e)$ (Type III event) Similar to case 3(b).

(4) (**p is already permanently labeled, and $\mathcal{R}^{SE}(p) \neq \emptyset$**) This is the case in which p has already been hit from the southeast. Locate and insert r in the hit list $\mathcal{R}^{SE}(p) = \{r_1, \ldots, r_N\}$ by finding the points r_i and r_{i+1} $(0 \leq i \leq N)$ such that the y-coordinate of r lies between that of r_i and that of r_{i+1} (where we define the y-coordinate of r_0 to be $-\infty$ and that of r_{N+1} to be $+\infty$). Refer to Figure 14. Insert new segments according to the cases below.

(a) (**p is interior to $\overline{qq'}$**) (Type II event) The propagation of the wavefront portion represented by $\overline{qq'}$ will possibly continue, but only after "clipping" is done on the west or north to account for the fact that other dragged segments that have previously hit p are sweeping "ahead" of $\overline{qq'}$.

(i) $(i = N)$ (Clip on the west) If $x_{q'_e} > x_{r_N}$, then insert a NW dragged segment rooted at r whose right track ray is that of $\overline{qq'}$ and whose left track ray is the northward ray through r_N. (Charge the lower left corner of $R(r, p)$ (namely, point (x_p, y_r)) with a "West Clip (WC).") If $x_{q'_e} \leq x_{r_N}$, then no new dragged segment needs to be inserted, since, by the time

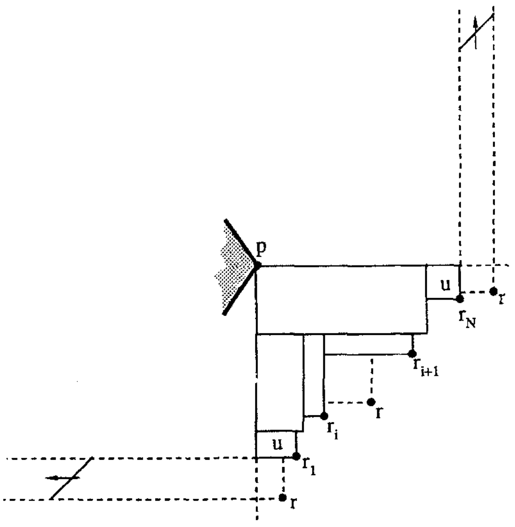

Fig. 14. Clipping at p using $\mathscr{R}^{SE}(p)$.

it hits p, $\overline{qq'}$ has already swept over the necessary portion of free space to the right of r_N.

(ii) ($0 < i < N$) No new dragged segment needs to be inserted, since, by the time it hits p, $\overline{qq'}$ has already swept over the necessary portion of free space to the right of r_i and below r_{i+1}.

(iii) ($i = 0$) (Clip on the north) If $y_{q_e} < y_{q_1}$, then insert a NW dragged segment rooted at r whose left track ray is that of $\overline{qq'}$ and whose right track ray is the westward ray through r_1. (Charge the upper right corner of $R(r, p)$ (namely, point (x_r, y_p)) with a "North Clip (NC)." If $y_{q_e} \geq y_{r_1}$, then no new dragged segment needs to be inserted, since, by the time it hits p, $\overline{qq'}$ has already swept over the necessary portion of free space below r_1.

(b) ($p = q_e$) (Type III event) This case is similar to case 4(a), except that there can be clipping on the west but not on the north (since, necessarily, $y_{q_e} = y_p \geq y_{r_1}$).

(c) ($p = q'_e$) (Type III event) This case is similar to case 4(a), except that there can be clipping on the north but not on the west (since, necessarily, $x_{q'_e} = x_p \geq x_{r_N}$).

Remark (Charging Events). The specification of the procedure includes instructions to "charge" some point at each event. This accounting scheme will allow us to give an upper bound on the total number of events that can occur. The analysis of our charging scheme will be discussed in more detail in Section 6.

Inserting a Dragged Segment. By "inserting" a dragged segment $\overline{qq'}$, we mean to compute its stop points, find its event point, event position, and event distance, and update the event queue accordingly. When we insert a dragged segment into the priority queue, we must determine its event point and corresponding event distance, by the methods of Section 3. For types NW(H, O), NW(H, V), NW(O, O), and NW(O, V), this is done simply by locating an appropriate point, $\rho = (x_{q'} - \varepsilon, y_q + \varepsilon)$ (for very small $\varepsilon > 0$), in the subdivision $S(5\pi/4, \pi, \pi/2)$. This works, since, in our algorithm, we can easily check that it will always be the case that the triangle $\triangle\rho qq'$ is obstacle-free, so we can consider the dragged segment to have started at point ρ, with zero length. For types NW(H, H) and NW(V, V), the segment insertion is done by checking each of the two track rays for stop points and doing an appropriate range query by the methods of [Ch2].

The remaining three types of segment dragging queries (NW(O, H), NW(V, H), and NW(V, O)) are potentially problematic, since, as mentioned in Section 3, we do not know how to preprocess in time $O(n \log n)$ to do them in $O(\log n)$ time. We determine the next event for a type NW(V, H) dragged segment by calling the procedure *Find-Next-Event-NW(V, H)* $(\overline{qq'})$, which we now define. Similar procedures can be defined for the types NW(O, H) and NW(V, O), and for the other three directions $\delta \in \{SE, NE, SW\}$. If a method is found for doing the segment dragging queries "into a corner" directly, then these procedures will be unnecessary for our algorithm.

Procedure *Find-Next-Event-NW(V, H)* $(\overline{qq'})$

(0) Let c be the corner into which $(\overline{qq'})$ is being dragged. (For a NW(V, H) segment, point c is the intersection of the vertical line through q with the horizontal line through q'.)

(1) Drag $(\overline{qq'})$ northward until (a) q reaches c, or (b) q' hits a stop point, or (c) a vertex w is hit by the dragged segment at position $\overline{q_e q'_e}$. In case (a), we can stop, since no obstacle vertex is hit by a NW(V, H) dragged segment; the procedure returns with a message that no event point exists. In case (b), we continue dragging the segment northward, now with q' sliding along the appropriate obstacle segment; then, case (a) or (c) must occur. In case (c), we check the point w: If $w \in R(r, c)$, the procedure stops and returns point w; otherwise, we let $q = q_e$, $q' = w$, and we return to (1) to continue dragging northward. ("Charge" point w.)

The basic idea of the above procedure is to drag the segment $\overline{qq'}$ *northward* instead of dragging it into the corner. If we are lucky enough to hit first a point that lies inside the corner, then we are done. Otherwise, we clip the northward dragging segment on the right at the obstacle that stopped us, and we continue dragging the clipped segment northward. The fact that the above procedure will eventually find the next point hit by a dragged segment of type NW(V, H) is clear, since the region swept out by the segments we drag northward contains the triangle $\triangle qcq'$. But the fact that, before arriving at the desired event point p, we may hit many points w that are outside (above) the corner into which we should be dragging is a potential source of problems. However, each such point that we hit is "charged," and we are able to show (in Lemma 5 of Section 6) that no point is charged more than once by the calling of this procedure.

Note that, since the left and right track rays for a NW(V, H) segment dragging query cross each other, it is possible that there is no event point detected (case (a) of step 1), in which case there is no change to the event queue when we "insert" the segment. In essence, the dragged segment simply "dies" into the corner, encountering no vertices along the way.

5. Correctness of the Algorithm.

The correctness of our algorithm follows directly from Lemma 1 and two simple observations:

(1) Let v be a vertex with parent r in SPT(s), and let $\delta \in \mathscr{D}$ be the direction from r to v. Then, once vertex r is encountered by the advancing wavefront, dragged segments rooted at r are created in such a way as to sweep out all of the region $acc^{\delta}(r)$, *except* those points of $\mathscr{F}$ that are missed due to clipping (in step 4 of *Propagate*).

(2) The clipping that is done in step 4 of *Propagate* is done conservatively: When a dragged segment is clipped and replaced by a new dragged segment that sweeps out a smaller region, this smaller region includes all points that are best reached by the root r.

The first observation follows from checking each case of the procedure *Propagate*, making certain that the wavefront is advanced in all possible directions.

The proof of the second observation is also straightforward. Assume that a NW dragged segment rooted at r is clipped on the west along the vertical line through point r_N, as a result of a collision with event point p. We claim that no point v in the region $acc^{NW}(r_N) \cap acc^{NW}(r)$ has parent r in SPT(s). Otherwise, point v is better reached from r than from r_N, which would imply that point p is better reached from r than from r_N, contradicting the fact that r_N hit p before r did.

LEMMA 2. Propagate *correctly assigns distance labels, $d(v)$, and parent pointers, parent(v), to vertices $v \in \mathscr{V}$. Thus, our algorithm correctly constructs the shortest path tree, SPT(s).*

PROOF. The proof proceeds by induction on the iteration count of the main loop of the algorithm and mimics the usual proof of correctness of Dijkstra's algorithm [Di]. We claim that each time we permanently label an obstacle vertex $p = v$ (done in step 2 of *Propagate*), we label it with the correct shortest path length from s. Assume that v is labeled with the distance $d(v) = d(r) + d(r, v)$ in step 2 of *Propagate*, due to a collision of a dragged segment rooted at r with the vertex $p = v$. Let D be the event distance at which the collision takes place. Assume inductively that all distance labels of vertices at distance less than D from s have been assigned correctly. If we were incorrect in labeling v with $d(r) + d(r, v)$, then there must exist a different root, $r' \neq r$, with the property that v is immediately accessible from r' and $d(r') + d(r', v) < D = d(r) + d(r, v)$. Thus, r' is at a distance less than D from s, implying (by the induction hypothesis) that it has already been (correctly) permanently labeled and (by observations (1) and (2) above) that there must be a dragged segment in the event queue that is rooted at r' and that has an event distance less than D, a contradiction. □

6. Complexity of the Algorithm.

6. Complexity of the Algorithm. In Section 3 we showed that each of the segment dragging queries that we require can be performed in time $O(\log n)$ after an $O(n \log n)$ preprocessing, using $O(n)$ storage. Furthermore, stop points can be computed in overall time $O(n \log n)$. Updates to the hit lists can also be done in logarithmic time. Thus, the overall running time of our algorithm is bounded by $O(E \log n)$, where E is the number of events.

Our goal in this section is to bound E, the total number of events. We do this by bounding the total number of dragged segments that are ever created, since each dragged segment corresponds to exactly one event. In each call to the procedure *Propagate*, we create a small number (at most 4) of new dragged segments according to the several cases outlined. In each case involving a segment creation, we "charge" some point for the creation.

Let us review the charging scheme specified in *Propagate*. We consider, without loss of generality, the case of propagating a dragged segment to the northwest:

(1) If event point p is a stop point, then we charge the event to p. There are at most $4n$ stop points, and each is charged at most twice—once because of a left track ray, and once because of a right track ray.
(2) If vertex p is permanently labeled in *Propagate*, then we charge p, and this is done only once for each vertex.
(3) If vertex p has been permanently labeled, but has not yet been hit from the southeast ($\mathscr{R}^{SE}(p) = \varnothing$), then we charge p. This results in at most three additional charges to vertex p.
(4) If vertex p has previously been hit from the southeast, then, if a new dragged segment is created, we either charge the southwest corner of the rectangle $R(r, p)$ with a West Clip (WC), or we charge the northeast corner of the rectangle with a North Clip (NC). (Note that these points that we charge may, in some cases, not lie in free space. This will not matter in our arguments that bound the number of charged points.)

It is easy to bound the number of charges in cases (1)–(3) by $O(n)$: there are only a linear number of points that can be charged and each is charged only a constant number of times. In case (4), however, we charge "grid points": a *grid point* is a point at the intersection of a horizontal line through a vertex and a vertical line through a vertex. There are a quadratic number of grid points, so we must be careful in counting the number of grid charges. Our objective is to show that only a very small number (at most $O(n \log n)$) of the grid points can be charged in the worst case. The proof of this claim relies crucially on the fact that certain patterns of "charge" are not possible on the grid. We restrict our attention to bounding the number of West Clips (WC's) that are charged to grid points when propagating dragged segments to the northwest; other cases are handled similarly.

An example of the charging of WC's to grid points is shown in Figure 15. Here, it is assumed that the length of the shortest paths from s to the roots r_i ($i = 1, 2,$

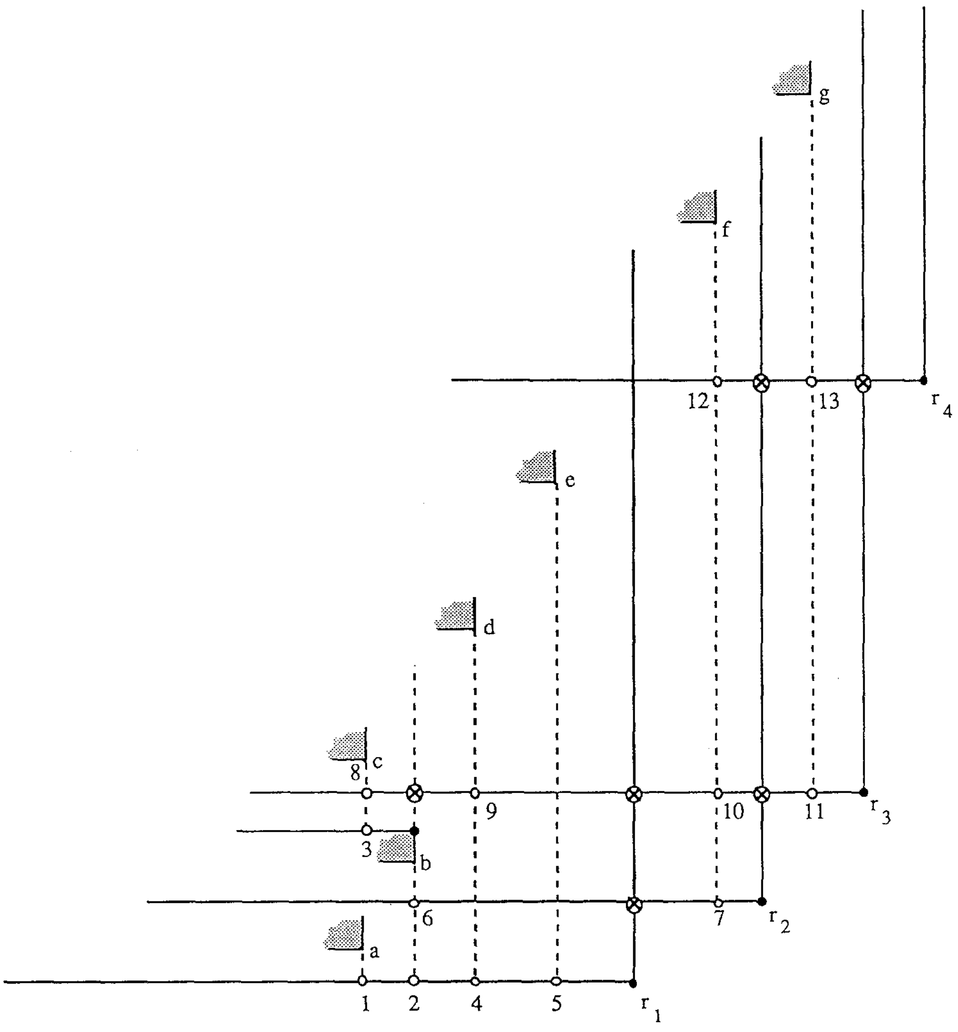

Fig. 15. Example of charging of WC's to grid points.

3, 4) *are ordered as follows*: $d(r_1) \ll d(r_2) \ll d(r_3) \ll d(r_4)$. With this assumption, we get the West Clip charges as shown by small hollow circles. The numbers by the charges indicate the order in which the collisions take place. The large circles with the " × " indicate the positions of the actual clip points. (The vertical lines through the clip points are the new left track rays that are implied by the West Clips.)

We bound the number of WC's by relating it to the number of ones that can occur in a binary matrix in which certain "violated patterns" cannot exist.

Our first claim is that rectangles are violated patterns:

LEMMA 3. *In propagating dragged segments to the northwest, procedure* Propagate *will not charge the four corners of a rectangle with a West Clip.*

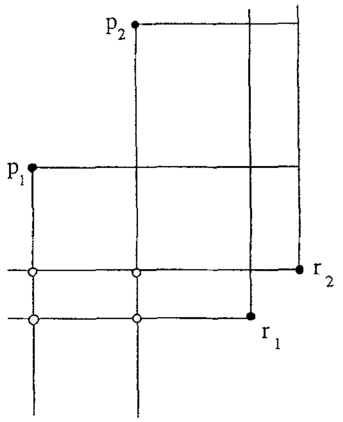

Fig. 16. Proof of Lemma 3.

Proof. To see that the four corners of a rectangle cannot be charged WC, assume the contrary. (Refer to Figure 16.) If r_1 hit p_1 first, then when r_2 hits p_1, it will be clipped to the right of r_1, and would then never hit p_2. Similarly, if r_2 hits p_1 first, then r_1 would be clipped above when it hits p_1, so it would never hit p_2. But a rectangle of WC charges implies that all four collisions must occur, a contradiction. □

Now, think of the grid points as defining elements of an n-by-n matrix, whose rows correspond to horizontal grid lines through vertices (which are distinct by the GPA) and whose columns correspond to vertical grid lines through vertices. Let the entry of the matrix be one if the corresponding grid point is charged WC and let it be zero otherwise. Note that each grid point can be charged WC at most once.

We say that a binary matrix contains no rectangles (or is "rectangle-free") if there do not exist distinct rows i and i' and distinct columns j and j' such that all four matrix elements at (i,j), (i,j'), (i',j), and (i',j') are ones. It is not difficult to show that an m-by-n binary matrix with no rectangle of ones can have at most $O((m+n)^{1.5})$ ones. (See [Bo], or see [Lo], Problem 10.36(a).) In fact, the bound of $O((m+n)^{1.5})$ is best possible, since there are rectangle-free matrices achieving this density of ones (see [Lo]). These facts can also be interpreted as results from *extremal graph theory* (as in [Bo], [Lo]).

Thus, an immediate corollary of Lemma 3 is that the number of WC's (and, hence, the number of events E) is bounded by $O(n^{1.5})$. To get a better bound on the number of WC's, we examine a larger class of violated patterns of WC charges. Let $x_1 < x_2 \leq x_3$ and $y_1 < y_2$. Then we will show that it is not possible for the four points (x_1, y_1), (x_3, y_1), (x_1, y_2), and (x_2, y_2) all to be charged with a West Clip. (We call such a set of four points a *trapezoid*.)

Lemma 4. *In propagating dragged segments to the northwest, procedure* Propagate *will not charge the four corners of a trapezoid with a West Clip.*

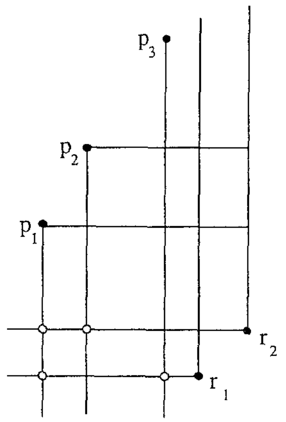

Fig. 17. A trapezoid of West Clip charges.

PROOF. Refer to Figure 17. Assume that a trapezoid exists, as shown in the figure. Note that the order in which points p_1, p_2, and p_3 are hit by a diagonal line dragged to the northwest must be p_1, p_2, p_3. (This follows since, if for example, p_3 were to be the first one hit, then when r_1 hits p_3 and gets west clipped, it is cut off from being able to hit p_1 or p_2.) Then, r_1 must have hit p_1 *before* r_2 did (otherwise, r_1 would have been north clipped when it hit p_1, rather than being west clipped). But then, when r_2 does hit p_1, it is west clipped at a point at or east of point r_1, meaning that it could not have gone on to collide with p_2. This is a contradiction. □

If we again interpret the grid points as elements of a binary matrix in which there is a one corresponding to grid points that are charged with a West Clip and a zero otherwise, then we can get a bound on the number of charges by studying the density of trapezoid-free binary matrices. Bienstock and Györi [BG] have studied such matrices (motivated, in fact, by our application) and have shown the following result:

PROPOSITION 1 (Bienstock and Györi). *The number of ones in a trapezoid-free binary n-by-n matrix is* $O(n \log n)$.

In fact, the stated upper bound is tight: examples exist of trapezoid-free binary matrices with $\Omega(n \log n)$ ones [Sch]. An immediate corollary is then:

COROLLARY 1. *The number of West Clips (and, hence, the number of events E) is* $O(n \log n)$.

We can continue to find other violated patterns of charges in hopes of obtaining tighter bounds on the number of WC's. In an earlier draft of this paper [Mi2], we showed, for example, that there can be no "*L*-quadrilateral" of charges on the grid points. (An *L-quadrilateral* is given by the four points (i_1, j_1), (i_2, j_1), (i_1, j_2),

(i', j') for indices $i_1 \le i' < i_2$ and $j_1 < j' \le j_2$.) In [Mi2], we claimed an upper bound of $O(n(\log n/\log\log n))$ on the number of ones in an L-quadrilateral-free binary matrix, but there was an error in the proof, so this question remains open. [BG] have shown a lower bound of $\Omega(n(\log n/\log\log n))$, which we conjecture to be tight. The example of a matrix that yields the lower bound in [BG], however, cannot occur as a set of WC charges in our problem, so it does not imply a corresponding lower bound on the number of events for our algorithm. It remains to be seen whether other violated patterns of charges can be found that will lead to a linear bound on the number of WC's and hence a proof of optimality of our algorithm.

Bounding Collisions of Find-Next-Event-NW(V, H). In order to complete the complexity analysis of our algorithm, we must show that the procedure we use to perform queries of dragging segments into corners does not consider too many collisions. Recall that *Find-Next-Event-NW(V, H)* determines the next collision caused by dragging a segment into a corner c to the northwest. This is a type of segment dragging query that we do not know how to answer in optimal time, so it was simulated by actually dragging the segment northward and checking the collision point for being inside the corner (that is, inside the rectangle $R(r, c)$). If it is not, we clip the segment on the right at the collision point, and we drag it northward again, continuing until we either hit a point that is inside the corner or we discover that no such point exists. Each time we hit a point w that lies outside the corner, we charge it. How many such charges are there? Our next lemma shows that no vertex w will be charged more than once, inplying that the total amount of work expended in calls to *Find-Next-Event-NW(V, H)* is $O(n \log n)$.

LEMMA 5. *Each obstacle vertex w is hit by procedure* Find-Next-Event-NW(V, H) *at most once. A similar chaim holds for procedures* Find-Next-Event-NW(O, H) *and* Find-Next-Event-NW(V, O).

PROOF. We prove only the claim for NW(V, H) dragged segments; similar proofs can be written for the other two cases. The proof is by contradiction. Assume that w is hit twice by one or more calls to procedure *Find-Next-Event-NW(V, H)*: once when $\overline{qq'}$, rooted at r, is dragged into corner c, and once when $\overline{pp'}$, rooted at r', is dragged into corner c'. Let $\overline{q_e q'_e}$ and $\overline{p_e p'_e}$ be the corresponding event positions when the dragged segments are in contact with w.

If $c \neq c'$, then we assume, without loss of generality, that c' is south of c, implying that c' is southwest of c, since our propagation rules imply that c is immediately accessible from r. Also, then, if $r \neq r'$, it follows that r' must be southwest of r.

First, note that in order for w to have been hit, we must have $x_q < x_w < x_{q'}$ and $x_p < x_w < x_{p'}$. This means that w is to the right of c and c' and to the left of r and r'.

We consider two cases, depending on whether or not $r = r'$:

(a) If $r = r'$, then we get a contradiction as follows. We can immediately rule out the case $c = c'$, since this would imply that the point w is hit twice by the same call to *Find-Next-Event-NW(V, H)*. Thus, we can assume we have a situation as depicted in Figure 18.

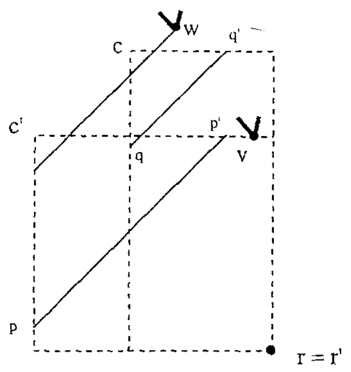

Fig. 18. Proof of Lemma 5: Case $r = r'$.

The right track ray of $\overline{pp'}$ (which is the horizontal line through c') can only come about due to a vertex v on this ray that is left of $r = r'$ and right of p', and that is immediately accessible from r. But this implies that c cannot be immediately accessible from r, a contradiction.

(b) If $r \neq r'$, then we get a contradiction as follows. We consider two cases, according to whether c' is north or south of r:

(i) If c' is north of r, we get the case shown in Figure 19. The right track for $\overline{pp'}$ must pass through a vertex v that is immediately accessible from r' and that lies left of r' and right of p' $(x_{p'} < x_v < x_{r'})$. Then, the only way that c can be immediately accessible from r is for v not to lie in the connected component of $R(r, c) \cap \mathscr{F}$ that contains r and c. This implies that there is an obstacle boundary segment cutting through $R(r, c)$, such that w lies above the segment and r' lies below it. Such an obstacle prevents the procedure *Find-Next-Event-NW*(V, H) $(\overline{pp'})$ from being able to hit vertex w. (Note that this argument remains valid when $c = c'$.)

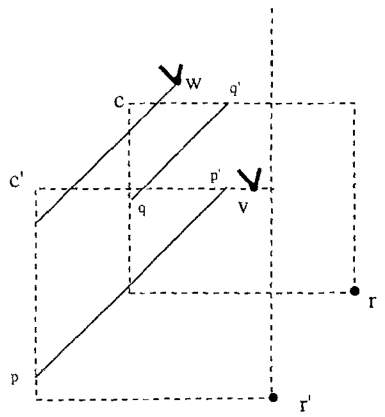

Fig. 19. Proof of Lemma 5: Case $r \neq r'$ and c' lies above r.

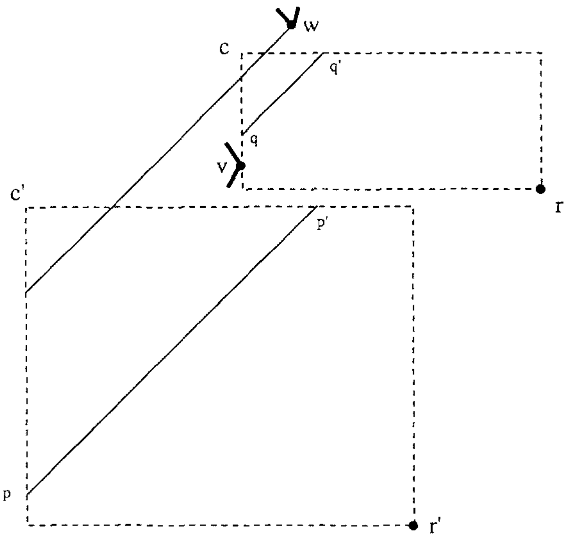

Fig. 20. Proof of Lemma 5: Case $r \neq r'$ and c' lies below r.

(ii) If c' is south of r, we get the picture in Figure 20. The left track ray of $\overline{qq'}$ must pass through a vertex v that is immediately accessible from r and that lies above r and below q $(y_r < y_v < y_a)$. But this implies that a call to *Find-Next-Event-NW*(V, H) $(\overline{pp'})$ cannot cause a collision with w, since the obstacle containing v will "shield" w: when we drag northward an inclined segment whose left endpoint slides along $x = x_{c'}$, the obstacle containing v must be hit before w is hit, and then, as the procedure trims the dragged segment on the right, the dragged segment can progress further northward only once it makes it around the westernmost corner of the obstacle containing v. This contradiction concludes the proof. □

We can finally state our main result as a theorem:

THEOREM 1. *Given a source point s and a set of polygonal obstacles with a total of n vertices, one can build an L_1 shortest path tree SPT(s) rooted at s in time $O(E \log n)$ and space $O(E)$, where $E = O(n \log n)$ is the number of events in the main algorithm. The shortest path tree SPT(s) can be extended to a shortest path map SPM(s) in time $O(n \log n)$, so that queries for the shortest path length to any destination t can be answered in time $O(\log n)$ and a shortest path from s to t can be reported in time $O(k + \log n)$, where k is the number of turns in the path.*

PROOF. The correctness of the algorithm for generating the SPT(s) was established in the previous section. Thus, we must now argue only the time and space complexity of the algorithm.

Each call to *Propagate* will require at most a constant number of segment dragging queries, each of which costs us $O(\log n)$ time (not counting dragging into corners). Each iteration of the loop in *Find-Next-Event-NW*(V, H) requires time $O(\log n)$ to drag the segment northward, and, by Lemma 5, there will be at most $O(n)$ iterations in total. Each insertion or lookup into a list $\mathscr{R}^{\delta}(v)$ costs $O(\log n)$, and there will be at most $O(E)$ insertions and lookups. Thus, the total time complexity of our algorithm is $O(E \log n)$. Also, clearly, the data structures will require $O(E)$ space. Corollary 1 shows that $E = O(n \log n)$. This proves the claimed bounds for the SPT problem.

The extension to the SPM can be done by at least two methods. One method is to modify our existing algorithm to do more bookkeeping, so that we compute bisectors as we go, and can output the SPM directly. A second method is to use our existing algorithm to compute the SPT(s), which yields the lengths of shortest paths from s to every other vertex v. We can then construct the SPM(s) by considering each vertex v to be a weighted source, with (additive) weight $d(v)$, and applying a weighted constrained Voronoi diagram algorithm according to the L_1 metric, constrained by the set of obstacles. This can be done, for instance, by the sweep method of Fortune [Fo], enhanced to handle obstacles, as in Seidel [Se]. (Seidel considers only the unweighted case, but, as pointed out in Fortune [Fo], the (additive) weighted case can be handled by imagining that the "cones" erected on each source point are raised by an amount equal to the weight.) □

Backtracing a Shortest Path. There are two possibilities for the path we output: we can require that it be rectilinear, or we can allow it to be any polygonal path, while we measure its length according to the L_1 metric. In either case, we begin by locating the query point t in the SPM(s). We then follow back pointers from the root r of the cell containing t to the root of the cell containing r, etc., until we reach point s. The path obtained in this way is the polygonal path from s to r in the shortest path tree SPT(s), together with the segment from r to t. If we require a rectilinear path from s to t, we now simply replace each line segment in the polygonal path by a staircase path in free space. These staircase paths are easy to identify in time proportional to their size (see [LL]).

Note that if we require that the path from s to t be rectilinear, then we must be prepared to have the number of its links very large. In fact, it may have a countably infinite number of links in the case that s (or t) lies at a vertex of a polygon whose interior angle is greater than $3\pi/2$. If the cone of free space at s does not include any of the four coordinate directions, then a rectilinear path from s to any other feasible point will require a countably infinite number of links (see [LL]).

Multiple Sources. An immediate generalization of our algorithm is to the case of multiple sources. We simply modify the initialization of the main algorithm (Step 0) to insert the four dragged segments that surround each source point at the beginning. This allows us to build a Voronoi diagram for multiple source points that lie among a collection of polygonal obstacles in the L_1 plane. It is important that each source point be considered to be a vertex of the free space so that the source points are included as if they were point obstacles, thereby allowing collisions to be detected that result in clipping propagation from various different source points.

THEOREM 2. *We can construct the L_1 Voronoi diagram of a set of K sites among a set of simple polygonal obstacles in time $O(E \log N) = O(N \log^2 N)$, where $N = K + n$ and $E = O(N \log N)$ is the number of events.*

Note that this then solves the problem of [LL] with m origin-destination pairs in time $O((m + n) \log^2(m + n))$, improving the previous bound of $O(m(m^2 + n^2))$.

Fixed Orientations. Another generalization of our algorithm allows us to solve shortest path problems involving distances with fixed orientations (see [WWW]). Basically, a fixed orientation metric defines the distance between two points to be the length of the shortest path that travels only along a given set A of fixed orientations. The L_1 metric is the special case in which the fixed orientations are 0 and $\pi/2$: it measures lengths of rectilinear paths. A "circle" with respect to a distance function with $k = |A|$ fixed orientations is given by a $2k$-gon. Thus, wavefronts among obstacles in this distance function are piecewise-linear with (oriented) segments of $2k$ different inclinations. There has been nothing special about the orientations 0 and $\pi/2$ in our discussions; in particular, the segment dragging query problems of Section 3 were discussed for arbitrarily inclined segments sliding along arbitrarily inclined track rays. The complexity of our algorithm gets multiplied by a factor of k when we pass from L_1 metrics to distances with k fixed orientations.

THEOREM 3. *Consider distances between points of the plane to be measured according to a fixed orientation distance function defined by a set of k directions. Then, within a multiply-connected polygonal space $\mathcal{P}$, one can build a shortest path map or a Voronoi diagram in time $O(kE \log n)$ and space $O(kE)$, where $E = O(n \log n)$ is the number of events in our main algorithm.*

If the fixed orientations A that define a distance function are evenly spaced in the range $[0, \pi)$, then as k grows large, the fixed orientation distance becomes a close approximation to the Euclidean distance (the percentage error decreases like $1/k^2$). This implies the following corollary, which compares favorably to the recent results of [Cl], who finds an ε-optimal path in time $O(n/\varepsilon + n \log n)$, after spending $O((n/\varepsilon) \log n)$ time to build a data structure of size $O(n/\varepsilon)$.

COROLLARY 2. *One can compute ε-optimal shortest paths and geodesic Voronoi diagrams in the Euclidean plane cluttered with polygonal obstacles in time $O(n\varepsilon^{-1/2} \log^2 n)$ and space $O(n\varepsilon^{-1/2} \log n)$.*

7. Conclusion. We have presented an algorithm for computing shortest paths and Voronoi diagrams according to the L_1 metric (and more generally, fixed orientation metrics) in a planar environment with polygonal obstacles. The algorithm runs in time $O(E \log n)$ and space $O(E)$, where E has been shown to be $O(n \log n)$ and is conjectured to be $O(n)$.

Our complexity analysis applied combinatorial bounds on the densities of certain sparse binary matrices. This two-dimensional "Davenport–Schinzel" reasoning is likely to continue to be a tool in other geometric algorithms. It is an interesting area of research to explore other combinatorial bounds on sparse matrices; several specific open questions were suggested in Section 6. Very recently, Füredi and Hajnal [FH] have addressed several related questions concerning densities of sparse matrices, adding to the results of Bienstock and Györi [BG], which were motivated by our geometric application.

It should be possible to extend our method to give an efficient algorithm for the construction of the L_1 (or fixed orientation metric) Voronoi diagram of a set of pairwise-disjoint polygonal sources.

It would be interesting to apply our continuous Dijkstra method to the three-dimensional problem of finding L_1 shortest paths among orthohedral obstacles. The goal is to get a subquadratic time algorithm, thereby improving the naive grid-based algorithm discussed in [Mi2].

REMARK. There has been some recent work on the problems addressed here that has taken place since the writing of this paper: [CKV] have independently obtained an algorithm with time and space bounds similar to ours, although by very different methods. Also, [Wi] has recently obtained bounds (by a similar method to that of [CKV]) that are nearly optimal, especially in some special cases. Both [CKV] and [Wi] use a technique that involves computing a sparse "shortest paths preserving" graph, which is guaranteed to include shortest paths between pairs of vertices.

Acknowledgments. This research was partially supported by NSF Grants IRI-8710858 and ECSE-8857642 and by a grant from Hughes Research Laboratories. Much of the work was done while the author was affiliated with the Hughes Research Laboratories, Malibu, California. The author would like to thank his advisor, Christos Papadimitriou, and D. T. Lee for helpful discussions and suggestions on these problems, and to thank Bernard Chazelle for pointing out the use of his data structures [Ch2] for solving segment dragging queries in linear space. Many fruitful discussions on the density of trapezoid-free matrices were with Esther Arkin, Dan Bienstock, Alan Hoffman, Offer Kella, Ed Schmeichel, and Karel Zikan.

References

[As] T. Asano, Rectilinear shortest paths in a rectilinear simple polygon, *Trans. IECE of Japan*, **E69** (1985), 750–758.

[AAGHI] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai, Visibility of disjoint polygons, *Algorithmica*, **1** (1986), 49–63.

[BG] D. Bienstock and E. Györi, An extremal problem on sparse 0–1 matrices, *SIAM Journal on Discrete Mathematics*, **4** (1991), 17–27.

[Bo] B. Bollobás, *Extremal Graph Theory*, Academic Press, New York, 1978.

[Ch1] B. Chazelle, A functional approach to data structures and its use in multidimensional searching, *SIAM Journal on Computing*, **17** (1988). 427–462.

[Ch2] B. Chazelle, An algorithm for segment dragging and its implementation, *Algorithmica*, **3** (1988), 205–221.

[CEGS] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir, Lines in space—Combinatorics, algorithms and applications, *Proc. 21st Annual ACM Symposium on Theory of Computing*, 1989, pp. 382–393.

[Cl] K. Clarkson, Approximation algorithms for shortest path motion planning, *Proc. 19th Annual ACM Symposium on Theory of Computing*, New York City, May 25–27, 1987, pp. 56–65.

[CKV] K. Clarkson, S. Kapoor, and P. Vaidya, Rectilinear shortest paths through polygonal obstacles in $O(n \log^2 n)$ time, *Proc. Third Annual ACM Symposium on Computational Geometry*, Waterloo, Ontario, 1987, pp. 251–257.

[DLW] P. J. de Rezende, D. T. Lee, and Y. F. Wu, Rectilinear shortest paths with rectangular barriers, *Proc. First Annual ACM Symposium on Computational Geometry*, 1985, pp. 204–213.

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

[EGS] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, Optimal point location in a monotone subdivision, *SIAM Journal on Computing*, **16** (1986), 317–340.

[EOS] H. Edelsbrunner, M. H. Overmars, and R. Seidel, Some methods of computational geometry applied to computer graphics, *Computer Vision, Graphics, and Image Processing*, **28** (1984), 92–108.

[Fo] S. J. Fortune, A sweepline algorithm for Voronoi diagrams, *Algorithmica*, **2** (1987), 153–174.

[FH] Z. Füredi and P. Hajnal, Davenport–Schinzel theory of matrices, Manuscript, Department of Mathematics, MIT, October, 1989.

[KM] S. Kapoor and S. N. Maheshwari, Efficient algorithms for Euclidean shortest path and visibility problems with polygonal obstacles, *Proc. Fourth Annual ACM Symposium on Computational Geometry*, Urbana-Champaign, IL, June 6–8, 1988, pp. 172–182.

[Ki] D. G. Kirkpatrick, Optimal search in planar subdivisions, *SIAM Journal of Computing*, **12** (1983), 28–35.

[LL] R. C. Larson and V. O. Li, Finding minimum rectilinear distance paths in the presence of barriers, *Networks*, **11** (1981), 285–304.

[Le] D. T. Lee, Proximity and reachability in the plane, Ph.D. Thesis, Technical Report R-831, Dept. of Electrical Engineering, University of Illinois, November 1978.

[LCY] D. T. Lee, T. H. Chen, and C. D. Yang, Shortest rectilinear paths among weighted obstacles, *Proc. Sixth Annual ACM Symposium on Computational Geometry*, 1990, pp. 301–310.

[LP] D. T. Lee and F. P. Preparata, Euclidean shortest paths in the presence of rectilinear boundaries, *Networks*, **14** (1984), 393–410.

[LWo] D. T. Lee and C. K. Wong, Voronoi diagrams in L_1- (L_∞-) metrics with 2-dimensional storage applications, *SIAM Journal of Computing*, **9**(1) (1980), 200–211.

[Lo] L. Lovász, *Combinatorial Problems and Exercises*, North-Holland, Amsterdam, 1979.

[LW] T. Lozano-Perez and M. A. Wesley, An algorithm for planning collision-free paths among polyhedral obstacles, *Communications of the ACM*, **22** (1979), 560–570.

[Mi1] J. S. B. Mitchell, Planning shortest paths, Ph.D. Thesis, Department of Operations Research, Stanford University, August, 1986.

[Mi2] J. S. B. Mitchell, Shortest rectilinear paths among obstacles, Technical Report No. 739, School of Operations Research and Industrial Engineering, Cornell University, April, 1987.

[Mi3] J. S. B. Mitchell, A new algorithm for shortest paths among obstacles in the plane, *Annals of Mathematics and Artificial Intelligence*, **3** (1991), 83–106.

[MMP] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou, The discrete geodesic problem, *SIAM Journal on Computing*, **16**(4) (1987), 647–668.

[MP] J. S. B. Mitchell and C. H. Papadimitriou, The weighted region problem: Finding shortest paths through a weighted planar subdivision, *Journal of the ACM*, **38**(1) (1991), 18–73.

[Ni] N. J. Nilsson, *Principles of Artifical Intelligence*, Tioga Publishing Co., Palo Alto, CA, 1980.

[PaS] J. Pach and M. Sharir, On vertical visibility in arrangements of segments and the queue size in the Bentley–Ottman line sweeping algorithm, *First Canadian Conference on Computational Geometry*, Montreal, Quebec, August 21–25, 1989. To appear: *SIAM Journal on Computing.*

[Pa] C. H. Papadimitriou, An algorithm for shortest-path motion in three dimensions, *Information Processing Letters*, **20** (1985), 259–263.

[Pr] F. P. Preparata, A new approach to planar point location, *SIAM Journal on Computing*, **10** (1981), 473–482.

[PrS] F. P. Preparata and M. I. Shamos, *Computational Geometry*, Springer-Verlag, New York, 1985.

[RS] J. H. Reif and J. A. Storer, Shortest paths in Euclidean space with polyhedral obstacles, Technical Report CS-85-121, Computer Science Department, Brandeis University, April, 1985.

[Sch] E. Schmeichel, Private Communication, Department of Mathematics and Computer Science, San Jose State University, 1986.

[Se] R. Seidel, Constrained Delaunay triangulations and Voronoi diagrams with obstacles, Technical Report Rep. 260, Institute für Information Processing, Graz, pp. 178–191, June, 1988.

[Sh] M. Sharir, *Davenport–Schinzel Sequences and their Geometric Applications*, NATO ASI Series, Vol. F40, Theoretical Foundations of Computer Graphics and CAD (R. A. Earnshaw, ed.), Springer-Verlag, Berlin, 1988, pp. 253–278.

[SCKLPS] M. Sharir, R. Cole, K. Kedem, D. Leven, R. Pollack, and S. Sifrony, Geometric applications of Davenport–Schinzel sequences, *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science*, 1986, pp. 77–86.

[SS] M. Sharir and A. Schorr, On shortest paths in polyhedral spaces, *SIAM Journal on Computing*, **15** (1) (1986), 193–215.

[We] E. Welzl, Constructing the visibility graph for n line segments in $O(n^2)$ time, *Information Processing Letters*, **20** (1985), 167–171.

[Wi] P. Widmayer, Network design issues in VLSI, Manuscript, Institut für Informatik, University Freiburg, Rheinstraße 10–12, 7800 Freiburg, West Germany, 1989.

[WWW] P. Widmayer, Y. F. Wu, and C. K. Wong, On some distance problems in fixed orientations, *SIAM Journal on Computing*, **16** (4), (1987), 728–746.