An Optimal Algorithm for L_1 Shortest Paths Among Obstacles in the Plane **(Draft)**

Joseph S. B. Mitchell [*]

Department of Applied Mathematics and Statistics
State University of New York, Stony Brook, NY 11794-3600
jsbm@ams.sunysb.edu

Abstract

We present an optimal $\Theta(n \log n)$ algorithm for determining shortest paths according to the L_1 (L_∞) metric in the presence of disjoint polygonal obstacles in the plane. Our algorithm requires only linear $O(n)$ space to build a planar subdivision (a Shortest Path Map) with respect to a fixed source point such that the length of a shortest path from the source to any query point can be reported in time $O(\log n)$ by locating the query point in the subdivision. An actual shortest path from the source to the query point can be reported in additional time $O(k)$, where k is the number of "turns" in the path. The algorithm uses the continuous Dijkstra methodology of propagating a "wavefront" through the plane. The algorithm can be generalized to find shortest paths according to any "fixed orientation" metric, yielding an $O(\frac{n \log n}{\sqrt{\epsilon}})$ approximation algorithm for finding Euclidean shortest paths among obstacles. The algorithm can further be generalized to the case of multiple sources to build a Voronoi diagram for multiple source points which lie among a collection of obstacles in time $\theta(N \log N)$, where N is the maximum of the number of sources and the number of obstacle vertices.

Key Words: shortest paths, continuous Dijkstra algorithms, shortest path maps, Voronoi diagrams, computational geometry

1 Introduction

There has been much work recently on various shortest path problems in computational geometry. The general problem is to find a "shortest" path from one point to another, given a set of path constraints and a means of assigning costs to paths. The simple case is that of requiring the path to remain within a multiply connected region, and measuring the cost as the length of the path (e.g., according to the Euclidean metric); the resulting problem is that of finding the shortest path for a point moving among a set of "obstacles". Emphasis has been on the planar case since it is known that the three-dimensional problem is NP-hard [CR]. A variety of problems arise, depending on the choice of distance criterion. Usually, we are interested in Euclidean or L_1 shortest paths. The worst-case time bound for the Euclidean case is presently stuck at $O(n^2)$, while the problem for L_1 distance was first solved in subquadratic ($O(n^{1.5} \log n)$) time by Mitchell [Mi2]. There have been improvements to the original work of [Mi2] (yielding bounds of $O(n \log^2 n)$ [Mi3] and $O(n \log^{1.5} n)$ [CKV, Wi]), but before this work the question of obtaining an optimal algorithm remained open.

[*]Partially supported by a grant from Hughes Research Laboratories, Malibu, CA, and by NSF Grant ECSE-8857642.

Previously, Larson and Li [LL] studied the problem of finding all minimal rectilinear distance paths among a set of m origin-destination pairs in the 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 $O(n \log n)$ [DLW] by exploiting the monotonicity of shortest paths. Furthermore, [DLW] show that $\Omega(n \log n)$ is a lower bound in the case of rectangular obstacles (and hence also in the more general case of polygonal obstacles).

In more recent work, [Mi2, Mi3] have used a continuous Dijkstra algorithm to arrive at an algorithm whose time bound is $O(g(n) \log n)$, where $g(n)$ is the density of a certain "sparse matrix" of zeros and ones (which, among other patterns, is not allowed to have rectangles or certain types of trapezoids of ones). The best bound known at this time on $g(n)$ is $O(n \log n)$. (The original report [Mi3] erroneously claimed that [BG] had proved that $g(n)$ is bounded by $O(\frac{n \log n}{\log \log n})$; an error was detected in the proof, so the conjecture remains open.) A very different approach is used by [CKV, Wi], who devise clever ways of obtaining a sparse subgraph of the grid graph defined by firing rays horizontally and vertically from each vertex. In particular, these methods are very useful in obtaining a sparse *shortest-path-preserving graph*, which guarantees that shortest paths between any pair of points (from among the original vertices) can be found among paths in the reduced grid graph. Very recently, Yang, Chen, and Lee [YCL] have developed an algorithm for the *weighted* version of our problem (that in which there is a cost per unit distance for traveling in each of several rectilinear regions).

In this paper, we present an optimal time and optimal space algorithm for computing the shortest path map among obstacles according to the L_1 (or L_∞) metric. Figure 1 illustrates a shortest path map with respect to source point s. Figure 2 illustrates a shortest path map in the case that all obstacles are rectilinear. The algorithm generalizes to allow multiple source points (thereby computing a Voronoi diagram in optimal time), and to allow multiple fixed orientation metrics [WWW] (thereby providing the best time bound for computing approximately optimal Euclidean shortest paths).

The method of our algorithm is that of the "continuous Dijkstra" paradigm, as was applied in [Mi2] and [Mi3]. In fact, our algorithm is essentially identical to that of [Mi2, Mi3], with one major simplifying step which makes the analysis easier and improves the time bound.

The basic idea behind our algorithm is the following: We propagate a "wavefront" out from the source, keeping track of "events" that occur when the wavefront makes critical changes. A crucial property that we are employing is the fact that the wavefront will be piecewise-linear, with segments of fixed orientations. This allows us to compute events by performing "segment dragging queries", which have been solved in optimal ($O(\log n)$) time and linear space by Chazelle [Ch1, Ch2].

We do not keep track of the exact structure of the wavefront at any instant (since we do not know how to solve for the next event in such a case), but rather we allow the wavefront to "run over" itself in a controlled way. (Basically, we acknowledge only those events that occur when a segment of the wavefront collides with an obstacle, rather than checking for collisions between one wavefront segment and another.) A very similar algorithm was used to obtain a nearly-optimal bound, where the proof of the time complexity relied on using various "sparse matrix" combinatorial arguments [Mi2, Mi3]. Here, we improve upon the technique by getting an even sparser "matrix" (one that is trivially linear in density), allowing us to show that the number of events remains linear ($O(n)$) despite the fact that we do not monitor the wavefront precisely.

2 Preliminaries

Given two points $q_1 = (x_1, y_1)$ and $q_2 = (x_2, y_2)$ in the plane, we define the L_p $(1 \leq p \leq \infty)$ distance between them as $d_p(q_1, q_2) = (|x_1 - x_2|^p + |y_1 - y_2|^p)^{1/p}$ for $p < \infty$ and as $d_\infty(q_1, q_2) = \max\{|x_1 - x_2|, |y_1 - y_2|\}$ for $p = \infty$. When $p = 1$, this definition gives us the *rectilinear distance* (or the L_1 distance) from q_1 to q_2 as $d(q_1, q_2) = d_1(q_1, q_2) = |x_1 - x_2| + |y_1 - y_2|$. Both L_1 and L_∞ are special cases of the more general "fixed orientation metrics" [WWW]. For purposes of the description of our algorithm, we shall restrict our attention to the case of the L_1 metric.

We assume that the obstacle space $\mathcal{O}$ is a collection of (interiors of) simple polygons and that $\mathcal{V}$ $(n = |\mathcal{V}|)$ is the set of obstacle vertices. We assume that the obstacles and the source point all lie within a very large rectangle, which we add to the obstacle list. Thus, we can think of the workspace as a rectangle with a set of "holes". The purpose of the bounding rectangle is to "catch" the wavefronts that we propagate and to assure that all of free space is properly subdivided.

For simplicity of discussion, we will make the following *General Positioning Assumption* (GPA) about the data of the problem: *No two vertices of the obstacle space lie on a horizontal, vertical, or diagonal ($\pm 45°$) line.* We make this assumption without loss of generality since one can always perturb the data slightly to achieve the GPA.

We are given a fixed source point s and we are asked to construct the *Shortest Path Map* (SPM$(s, \mathcal{O})$) with respect to s and $\mathcal{O}$. The SPM is a subdivision which allows one to look up the shortest path length to a destination point t simply by locating t in the subdivision (which can be done in optimal time $O(\log n)$ [Ki, Pr]). See [LP, Mi1, Mi4] for discussions of shortest path maps with respect to Euclidean distances. We will see that our algorithm immediately generalizes to the case of multiple source points, in which case the output structure is a Voronoi diagram with respect to L_1 obstacle-avoiding distance.

3 Segment Dragging Problems

We will need to have solutions to several "segment dragging" problems at our disposal in the main algorithm. By a segment dragging problem we mean a problem in which we are to preprocess a set of points and polygons such that 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. These problems are analogous to the *next-point search problems* of [EOS].

The simple 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 $\overline{qq'}$ being dragged in the direction of positive (negative) y is the familiar "range search for a min (max)" problem. See Figure 3(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] in which he solves this problem in preprocessing time $O(n \log n)$ and query time $O(\log n)$, with a space complexity of $O(n \log^{\epsilon} n)$, where ϵ is an arbitrarily small positive real number. (Alternately, the (time, space) complexities can be $(n \log^{1+\epsilon} n,\ n)$ 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) which we drag upwards (in the direction of the positive y-axis). This problem is easily seen to be equivalent to the horizontal 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 3(b)). Thus, after preprocessing, inclined segment dragging queries can be answered in time $O(\log n)$.

Another type of segment dragging query is the following: From a query point r, find the first point hit by a segment $\overline{qq'}$ at inclination θ which 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; 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 4. This segment dragging problem is solved by converting it to a point location problem (thereby using the so-called *locus approach* to solving problems in computational geometry). We build a subdivision, which we will call $S(\theta, \phi_1, \phi_2)$, such that an answer to the query is given by a point location of r in the subdivision.

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, 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 $\phi_1 + \pi$ and another ray in the direction of $\phi_2 + \pi$. 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 $S(\theta, \phi_1, \phi_2)$ 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 $q_{2,i}$. The result is as shown in Figure 5. Once we have this subdivision, if we locate r (in time $O(\log n)$, [Ki, Pr]) in, say, the shaded region of Figure 5, 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 (the ru-point subdivision is what we refer to as $S(\pi/2, 0, \pi/4)$).

The algorithm just described for building the subdivision $S(\theta, \phi_1, \phi_2)$ is easily extended to include the case of queries in the presence of both points, line segments, and polygons (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 segments of the subdivision as well as obstacles from the original set. (The sweep line intersects some set of the obstacles. In the free space between two consecutive intersections with obstacles, the frontier that lies below the sweep line consists of a set of segments at orientations ϕ_1 and ϕ_2 and segments bounding obstacles, and these segments form a "staircase" frontier that has the following property: Any line at an orientation between ϕ_1 and ϕ_2 intersects the frontier at a single point. This monotonicity property allows us to do $O(\log n)$ time updates when a new vertex is encountered. A case analysis shows that this property is maintained at each of the events (collisions of the sweep line with an obstacle vertex).) See Figure 6 and Figure 7. Note that if r is located in one of the shaded regions of Figure 6, 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. (The obstacles have a "shadowing" effect.) We can associate with the shaded region the label of the segment into which $\overline{qq'}$ "dies".

We allow the special cases in which $\theta = \phi_1$ or $\theta = \phi_2$. The corresponding subdivisions ($S(\theta, \theta, \phi_2)$ or $S(\theta, \phi_1, \theta)$, 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.

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 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. This is an alternative algorithm to that given in [LWo] for the construction of the Voronoi diagram in the L_1 (L_∞) metric, although we have no computational experience to suggest that this approach is any better than the standard divide-and-conquer algorithm. (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.) By using the subdivisions for the relevant segment-dragging queries, note that this technique also applies to give an alternative solution to the closest point query problem for fixed orientation metrics, requiring the same running time as the divide-and-conquer algorithm of [WWW].

One other type of segment dragging query will arise in our application. Consider the situation depicted in Figure 8. The segment $\overline{qq'}$ is dragged so that q and q' slide along the rays l_1 and l_2 (respectively). Point w is called the *inside corner* of the rays l_1 and l_2. The subdivision $S(\theta, \phi_1, \phi_2)$ solved the problem of dragging segment *out* of a corner; we now look at the problem of dragging a segment *into* a corner. (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. Our algorithm uses a technique which avoids these queries by using a combination of queries of the types described above (see Section 5). However we leave it as an interesting open problem whether queries of this form can be answered efficiently (and thereby whether the statement of our algorithm can be simplified).

Our interest in these segment dragging problems will primarily be to examine the effect of a "wavefront" propagating through a collection of rectilinear 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 segments in the set of polygonal obstacles. See Figure 9 for an example showing $S(5\pi/4, \pi/2, \pi)$ in the presence of rectilinear obstacles.

4 Subdivisions Induced by the Obstacles

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 $d(q, p_1) + w_1 = d(q, p_2) + w_2$. (The weight associated with a point will be the length of the shortest path from the source s to the point.) Various cases are shown in Figure 10, where it is also 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. Then, in cases (c) and (d), we define the bisector to be the thick solid line on the vertical boundary of the bisector regions.

Given two points $q_1 = (x_1, y_1)$ and $q_2 = (x_2, y_2)$ in the plane, we say that q_1 is *northwest of* q_2 if $x_1 \leq x_2$ and $y_1 \geq y_2$. Similar definitions apply to the terms *southeast, southwest*, and *northeast.* Define $R(q_1,q_2) = \{(x, y) : \min\{x_1,x_2\} \leq x \leq \max\{x_1,x_2\}$ and $\min\{y_1,y_2\} \leq y \leq \max\{y_1,y_2\}\}$ as the (closed) rectangle *cornered* at q_1 and q_2. We will say that point p is *immediately accessible* from q if $R(p, q) \cap \mathcal{O} = \emptyset$ (i.e., the rectangle does not intersect any obstacle).

Given a source point s, our algorithm will produce a subdivision $\mathcal{S}$, called a *Shortest Path Map* (SPM), of the plane into *cells* $C(r)$ ($r \in \mathcal{V}$). 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'. The subdivision is such that if $t \in C(r)$, then a shortest path from s to t is obtained by following any shortest path from s to r, and then proceeding directly from r to t (i.e., along a rectilinear path). Thus, if we are given an SPM, the length of shortest paths from s to t can be found in $O(\log n)$ time by solving a point location problem [Ki, Pr] to determine the cell $C(r)$ containing t. The length will be $d(t) = d(r, t) + d(r)$, where $d(r)$ is the length of shortest paths from s to r (the *depth* of r). Then the actual 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 (and, certainly, $k < n$). See Figure 1 and Figure 2 for examples of an SPM.

5 The Algorithm

Our algorithm operates in much the same spirit as the famous Dijkstra algorithm [Di]. A "signal" is propagated from the source s to all other points in the plane (not interior to an obstacle). 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 (from any of the four directions). The label $d(p)$ is the length of shortest paths from s to p (the *depth* of p). We continue to propagate signals until every point has received the signal. We need only keep track of discrete *events* which take place as the wavefronts expand. We will show that the number of events has an upper bound of $O(n)$ and that the update time per event is only $O(\log n)$.

Our algorithm maintains a priority queue consisting of a set of "dragged segments" which form portions of the current wavefront. Each dragged segment has associated with it a *root* (which is labeled by its distance from the closest source point); an *event point* which is the next collision (with an obstacle or another source point) for the dragged segment; and an *event distance*, which is the distance from the source to the root plus the distance from the root to the event point. The basic structure of the algorithm is to process events according to the ordering of the priority queue until the queue is empty.

When we process a collision, we must examine two basic cases: either the event point p has been hit before (by some other dragged segment), or the event point has never before been hit (see Figure 2). In the latter case, we simply label the event point with its distance from the source, and we can show that this labeling gives the true minimum distance from a source point to the event point. We also must do a simple case analysis to determine how the wavefront changes as a result of the collision, and we instantiate the appropriate new dragged segments. In the first case, we must adjust one or more dragged segments according to the *bisectors* that exists between the root of the segment that just hit the event point and the roots of segments that previously hit the event point. The crucial property of our algorithm that guarantees its efficiency is that no obstacle vertex will be hit more than eight times. (Previous bounds showed that on average each vertex could not be hit more than $O(\sqrt{n})$ [Mi2] or $O(\log n)$ [Mi3] times.) The major difference between our current algorithm and that of [Mi2, Mi3] is that we now propagate segments in all four directions (northwest, northeast, southeast, and southwest) simultaneously, rather than splitting the problem into the construction of four different subdivisions and then merging them. The more global analysis results in both a simpler algorithm and a better time bound.

We do not actually keep track of when one wavefront (say, propagating northwest) first runs into another 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.

Our algorithm uses a few simple data structures. First, we define what is meant by a *dragged segment*. A dragged segment is a portion of a wavefront boundary. Associated with each such segment $\overline{qq'}$ is the following information: 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 segments endpoints at the instant 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); 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); its *root* (which is the obstacle vertex which is responsible for propagating the portion of the wavefront to which the segment belongs); its *event position* $\overline{q_e q'_e}$ (the next position of its endpoints at which the segment changes its *contact list*, the list of obstacle points and obstacle edges which it is touching); its *event point* p; and the *event distance*.

The event point is the point on the boundary of an obstacle which is in contact with the segment in its event position but which is not in contact with the segment before it reaches its event position. (The GPA allows us to assume that the event point is *unique*.) The event distance is simply the distance from s at which the event point is encountered by the segment. It is given by $d(r) + d(r, p)$. For the case in which a dragged segment can be moved off to infinity without changing its contact list, we define a special event called the *event at infinity*, define the event point to be NIL, and set the event distance to $+\infty$. Note that the stop points are easily computed in $O(\log n)$ time by using the *next-element* subdivision of [EOS] or the Horizontal or Vertical Adjacency Map of [PS]; if the track rays intersect each other before they hit obstacles, then $L = R =$ their point of intersection, which is the inside corner of the corresponding segment dragging query.

When the event point p is interior to the dragged segment in its event position, we say that there has been an *interior collision* (a type I event); when p is one of the stop points (L or R), we say that the segment has *reached its stop point* (this is a type II event); and when p is an endpoint of the event position (but not a stop point), we say there has been an *endpoint collision* (a type III event). 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 and the time it hits its event point. If a segment's event point is NIL, then the region swept out by it is the semi-infinite part of the plane bounded by the segment $\overline{qq'}$ and the track rays.

There are sixteen basic types of dragged segments, depending on the orientation of the segment and of the left and right rays. These cases are illustrated in Figure 14, where names are assigned to the cases (for example, "NEO" stands for "NorthEast Outside", "NEI" stands for "NorthEast Inside", "ER" stands for "East Right", and "EL" stands for "East Left"). All points on a dragged segment are at the same L_1 distance from the root of the segment. In this sense, we can think of the roots of dragged segments as "virtual" sources which act to propagate the wavefront from the original source s.

The 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.

Each obstacle vertex $v \in \mathcal{V}$ has associated with it a sorted list $\mathcal{R}^{SE}(v) = \{r_1, \ldots, r_N\}$ of roots r_i of dragged segments which 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 $\mathcal{R}^{SE}(v)$ are kept in order of increasing y-coordinates (which will also be the order of increasing x-coordinates since the points of $\mathcal{R}^{SE}(v)$ will form a staircase path going northeast). To provide properly for degeneracies, we will allow r_1 to be due south of v and allow r_N to be due east of v. Similar definitions apply to $\mathcal{R}^{NW}(v)$, $\mathcal{R}^{NE}(v)$, and $\mathcal{R}^{SW}(v)$. Any particular list $\mathcal{R}^{\delta}(v)$, $\delta \in \mathcal{D} = \{SE, NE, SW, NW\}$, 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 $g(n)$, the number of events, and we will show that $g(n)$ is almost linear (and we conjecture that it actually is linear).

Also associated with each obstacle vertex $v \in \mathcal{V}$ is a *permanent label* $d(v)$, which gives the length of the shortest path from s to v. Initially, $d(v) = +\infty$ for all $v \in \mathcal{V}$. 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 there exist obstacle vertices r and v and a direction $\delta \in \mathcal{D}$ such that $r \in \mathcal{R}^{\delta}(v)$ (which implies that both r and v have been permanently labeled) and $x \in R(r, v)$ (that is, x lies in the region swept out by some dragged segment). Each vertex v also has a pointer, *pred*(v), back to the root r $(\neq v)$ of the cell $C(r)$ of the SPM which contains v (we can break ties according to lexicographic order).

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 the one, $\overline{qq'}$, with the smallest label and do 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 plane which we know to be better reached from some other root (from the same direction). Note that we are actually keeping four "types" of wavefronts (corresponding to the four directions $\delta \in \mathcal{D}$), and we allow two different types of wavefronts to "run over" each other.

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. Various cases are illustrated in Figure ??, where each of the three types of events are illustrated. If the event point p has not been labeled before, then the dashed segments are instantiated.

We have used the notation that $\hat{\imath} = (1, 0)$ (resp., $\hat{\jmath} = (0, 1)$) is the unit vector in the x-direction (resp., y-direction). The interpretation of $r + \epsilon\hat{\jmath} - \epsilon\hat{\imath}$ is that it is the point *just* northwest of point r ($\epsilon > 0$ is arbitrarily small). Since r is a vertex of the subdivision $S(5\pi/4, \pi/2, \pi)$, we can locate the point $r + \epsilon\hat{\jmath} - \epsilon\hat{\imath}$ in $S(5\pi/4, \pi/2, \pi)$ by finding the region containing r whose interior includes the interior of the quadrant northwest of r. In any reasonable representation of the subdivision, this will be doable in constant time.

By "inserting" a dragged segment, we mean to compute its stop points, find its event point, event position, and event distance, and update the event queue accordingly. If the dragged segment has its next event at infinity, then the segment is added to the end of the event queue, with a special marker indicating its event distance is $+\infty$.

Since clipping is done only when more than one dragged segment encounters the same vertex, we will not be determining the proper subdivision ($\mathcal{S}^{SE}$) in that region of the plane for which no obstacles lie to the northwest (that is, the set $\{t = (x, y) : \mathcal{O} \cap (-\infty, x] \times [y, +\infty) = \emptyset\}$). A simple remedy to this problem is to include in the set $\mathcal{O}$ four *point obstacles at infinity*, namely the points $(-\infty, +\infty)$, $(+\infty, +\infty)$, $(+\infty, -\infty)$, $(-\infty, -\infty)$. (Of course, in a real implementation one would use a sufficiently large integer M instead of ∞.) This has the effect of giving us the subdivision of the entire plane, since every root will eventually encounter some event point. (The four special points serve to "catch" the dragged segments that would otherwise have continued to slide out to the event at infinity.) Another option would be to enclose the obstacle space in a large bounding rectangle.

We should mention that it is straightforward to construct the actual subdivision $\mathcal{S}$ from the information obtained while running the algorithm. The construction can be done in time proportional to the size of the output, using the knowledge of the collisions and clippings that took place when propagating from each root. Another alternative is to note that the algorithm directly constructs the shortest path *tree*, giving the information necessary to backtrace the shortest path from s to each obstacle vertex. It is then possible to construct the shortest path map from the tree in $O(n \log n)$ time. The details are omitted here.

The specification of *Propagate* needs one further elaboration. We must describe what is meant by "inserting" a dragged segment of type NWI (or NEI, SWI, SEI), since we do not know how to solve this type of segment dragging query in optimal time $O(\log n)$ (see Section 3). We determine the next event for a type NWI dragged segment by calling the procedure *Find-Next-Event-NWI* ($\overline{qq'}$), which we now define. (Similar procedures can be defined for the other three directions (NEI, SWI, and SEI).)

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

(0). *Let c be the corner into which ($\overline{qq'}$) is being dragged. (Point c is the intersection of the vertical line through q with the horizontal line through q'.)*

(1). *Drag ($\overline{qq'}$) north until it hits the next point, w. ("Charge" point w.)*
Let $\overline{q_e q'_e}$ be the event position when the segment is in contact with w. ("Charge" point w.)

(2). *If $w \in R(r, c)$, then stop and output the point $p = w$ as the next NW(V,H) event point. Otherwise, if q_e is north of c, then stop (no obstacle vertex is hit by a NW(V,H) query segment). Otherwise, let $q = q_e$, $q' = w$, and go to (1).*

The above procedure works simply by dragging the segment $\overline{qq'}$ *northward* instead of dragging it into the corner. If we are lucky enough to hit first a point which 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 NWI is fairly 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 which 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 Section 7) that no point is charged more than once by the calling of this procedure.

6 Correctness of the Algorithm

Theorem 1 *The algorithm correctly computes the subdivision $\mathcal{S}$.*

Proof. Omitted in this abstract. Proof of correctness closely follows the proof given in [Mi3]. □

7 Complexity of the Algorithm

How many events can there be in running our algorithm? In [Mi2, Mi3] we used a charging scheme that charged events to points on the grid defined by the vertices of the obstacle. Then, by arguing that certain patterns of charged points could not occur, we arrived at bounds of $O(n^{1.5})$ and $O(n \log n)$ on the number of events. We conjecture that a closer analysis of the forbidden patterns of charge may yield a linear bound on the number of events in the algorithm of [Mi2, Mi3]. Our approach here is to get a linear bound on the number of events in our algorithm by showing the following lemma:

Lemma 2 *If r is the root of a dragged segment going northwest that hits p, then if r' also hits p from the southeast, then r' cannot lie within the circle of radius $d_1(r,p)$ centered at r. This implies that there can be only two collisions of a vertex p from the southeast.*

Proof. The key point is that if r' lies within the circle, then it will have interacted with r (or with some other root) and would have had its propagation clipped already, making it impossible for it to have encountered p. Here, we assume that r hits p before r' does. □

Next, we must show that the procedure we use to perform queries of dragging segments into corners does not hit too many vertices. Recall that *Find-Next-Event-NWI* 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? If we can show that no vertex w will be charged more than once, then the total amount of work expended on these types of queries is $O(n \log n)$. Our next lemma shows that indeed this is true.

Lemma 3 *Each obstacle vertex w is hit by procedure* Find-Next-Event-NWI *at most once.*

Proof. Follows the proof of [Mi3]. □

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-NWI* requires time $O(\log n)$, and there will be at most $O(n)$ iterations in total. Each insertion or lookup into a list $\mathcal{R}^{\delta}(v)$ costs $O(\log n)$, and there will be at most $O(n)$ insertions and lookups. Thus, the total time complexity of our algorithm is $O(n \log n)$. Also, clearly, the data structures will require $O(n)$ space.

By keeping track of the paths of the endpoints of all dragged segments, we can actually build the subdivision $\mathcal{S}$ as the algorithm proceeds. At the conclusion of the algorithm, the subdivision can be put into a structure which is appropriate for efficient point location queries [Ki, Pr].

Theorem 4 *The algorithm above constructs a shortest path map in obstacle space with n vertices in time $O(n \log n)$ using $O(n)$ space. After preprocessing, queries for the shortest path length to any destination can be answered in time $O(\log n)$ and the shortest path can be reported in time $O(k + \log n)$, where $k < n$ is the number of turns in the shortest path.*

Remark. The actual enumeration of an obstacle-free rectilinear path achieving the shortest path length could require a countably infinite number of points in a very special case. Consider, for example, the rectilinear path from a vertex of a polygon whose interior angle is greater than $3\pi/2$. If the exterior cone at the vertex does not include any of the four coordinate directions, then the shortest path from the vertex to any other feasible point will require a countably infinite number of "kinks" (see [LL]).

8 Extensions

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 which lie among a collection of polygonal obstacles in the L_1 plane. The running time is then $O(N \log N)$ with a space complexity of $O(N)$, where N is the sum of the number of sources and the number of obstacle vertices. Note that this then solves the problem of [LL] with m origin-destination pairs in time $O((m + n)\log(m + n))$, improving the previous bound of $O(m(m^2 + n^2))$.

Another generalization of our algorithm allows us to solve shortest path problems involving distances with fixed orientations (see [WWW]). The L_1 metric is the special case in which the fixed orientations are 0, $\pi/2$, π, and $3\pi/2$. The case in which there are K fixed orientations along which distances are measured gives rise to piecewise linear wavefronts, with K different types of wavefront segments. The segment dragging queries discussed in Section 3 are sufficiently general to handle these cases (since the inclinations of the segments are fixed). The result is that our algorithm goes through as before with no significant changes. The complexity becomes $O(nK \log n)$, which is optimal for fixed K. Note that if the fixed orientations are evenly spaced (in $[0, 2\pi)$), 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 immediately implies an approximation algorithm for ϵ-optimal Euclidean paths that runs in time $O(\frac{n \log n}{\epsilon^2})$. (In fact, within this time bound we build an ϵ-shortest path map or Voronoi diagram.)

9 Conclusion

We have presented an algorithm for constructing Voronoi diagrams (and, hence, shortest path maps) according to the L_1 metric (and more generally, fixed orientation metrics) in the presence of polygonal obstacles. The algorithm runs in time $\Theta(n \log n)$ and space $O(n)$. After this preprocessing, one can locate a query point in time $O(\log n)$, after which the distance to the nearest source is reported in constant time and a shortest path to the nearest source is reported in time proportional to the number of turns in the path.

Several open problems are suggested by our work. First, can our algorithm be simplified or understood in a more compact way? The underlying concept of our algorithm is quite simple, but it involves many different cases. It is likely that a cleaner approach is possible. Second, can our methods be extended to higher dimensions to yield improved bounds over those of [CKV, Mi2, Mi3]? (Note that distances with fixed orientations can be defined in three dimensions as well, which may lead to a new approximation scheme for the Euclidean shortest path problem in higher dimensions.) Third, can our method be applied to the case of weighted regions (possibly with the restriction that they be rectilinear) to give a competitive algorithm to that of [YCL]? Finally, our approach may also generalize to the case in which the sources are line segments (or polygons) instead of points.

Acknowledgements

The author is partially supported by NSF Grants IRI-8710858 and ECSE-8857642 and by a grant from Hughes Research Laboratories, Malibu, CA.

References

[BG] D. Bienstock and E. Györi, "An Extremal Problem on Sparse 0-1 Matrices", Manuscript, Bellcore, 1987. To appear: *SIAM Journal on Discrete Mathematics.*

[CR] J. Canny and J. Reif, "New Lower Bound Techniques for Robot Motion Planning Problems", *Proc. 28th FOCS*, pp. 49-60, Oct. 1987.

[Ch1] B. Chazelle, "A Functional Approach to Data Structures And Its Use in Multidimensional Searching", Technical Report No. CS-85-16, Dept. of Computer Science, Brown University, September, 1985.

[Ch2] B. Chazelle, "An Algorithm for Segment Dragging and Its Implementation", *Algorithmica* (1988) **3**, 205-221.

[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 Conference 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", *First ACM Conference on Computational Geometry*, June 1985, pp. 204–213.

[Di] E.W. Dijkstra "A Note On Two Problems in Connexion With Graphs", *Numerische Mathematik*, **1** (1959), pp. 269-271.

[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), pp. 92-108.

[Ki] D.G. Kirkpatrick, "Optimal Search in Planar Subdivisions", *SIAM Journal on Computing*, **12** (1983), pp. 28-35.

[LL] R.C. Larson and V.O. Li, "Finding Minimum Rectilinear Distance Paths in the Presence of Barriers", *Networks*, 11 (1981), pp. 285-304.

[LP] D.T. Lee and F.P. Preparata, "Euclidean Shortest Paths in the Presence of Rectilinear Boundaries", *Networks*, **14** (1984), pp. 393-410.

[LWo] D. T. Lee and C. K. Wong, "Voronoi Diagrams in L_1- $(L_\infty-)$ Metrics With 2-Dimensional Storage Applications", *SIAM Journal of Computing*, **9**(1), pp. 200-211, 1980.

[Mi1] J.S.B. Mitchell, "Planning Shortest Paths", PhD Thesis, Department of Operations Research, Stanford University, August, 1986. (Available as Research Report 561, Artificial Intelligence Series, No. 1, Hughes Research Laboratories, Malibu, CA.)

[Mi2] J.S.B. Mitchell, "Shortest Rectilinear Paths Among Obstacles", Technical Report, Department of Operations Research, Stanford University, 1986.

[Mi3] J.S.B. Mitchell, "L_1 Shortest Paths Among Polygonal Obstacles in the Plane", appears in *Algorithmica* **8** (1992), pp. 55–88.

[Mi4] J.S.B. Mitchell, "A New Algorithm for Shortest Paths Among Obstacles in the Plane", *Annals of Mathematics and Artificial Intelligence* **3** (1991), pp. 83–106.

[Pr] F.P. Preparata, “A New Approach to Planar Point Location”, *SIAM Journal on Computing*, 10 (1981), pp. 473-482.

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

[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**, No. 4, August 1987, pp. 728-746.

[YCL] C.D. Yang, T.H. Chen, and D.T. Lee, “Shortest Rectilinear Paths Among Weighted Obstacles”, *Proc. 6th Annual ACM Symposium on Computational Geometry*, 1990, pp. 301–310.