Shortest Paths in the Plane with Polygonal Obstacles

JAMES A. STORER

Brandeis University, Waltham, Massachusetts

AND

JOHN H. REIF

Duke University, Durham, North Carolina

Abstract We present a practical algorithm for finding minimum-length paths between points in the Euclidean plane with (not necessarily convex) polygonal obstacles. Prior to this work, the best known algorithm for finding the shortest path between two points in the plane required $\Omega(n^2 \log n)$ time and $O(n^2)$ space, where n denotes the number of obstacle edges. Assuming that a triangulation or a Voronoi diagram for the obstacle space is provided with the input (if is not, either one can be precomputed in $O(n \log n)$ time), we present an $O(kn)$ time algorithm, where k denotes the number of "islands" (connected components) in the obstacle space. The algorithm uses only $O(n)$ space and, given a source point s, produces an $O(n)$ size data structure such that the distance between s and any other point x in the plane (x is not necessarily an obstacle vertex or a point on an obstacle edge) can be computed in $O(1)$ time. The algorithm can also be used to compute shortest paths for the movement of a disk (so that optimal movement for arbitrary objects can be computed to the accuracy of enclosing them with the smallest possible disk).

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Euclidean plane, minimal movement problem, motion planning, mover's problem, polygonal obstacles, robotics, shortest path

1. *Introduction*

The classical *mover's problem* is: Given a *source point* and a *destination point* along with a set of polyhedral *obstacles* in two or three dimensional Euclidean

This paper was completed in April 1985 [Reif and Storer, 1985] and was submitted to this journal in June 1985; at that time and during the preceding year when the research was undertaken, J. Storer was partially supported by NSF grant number DCR 84-03244 and J. Reif was partially supported by the Office of Naval Research grant number N000-14-80-C-0647, while visiting the Laboratory for Computer Science at MIT

Authors' addresses. J. A. Storer, Computer Science Department, Center for Complex Systems, Brandeis University, Waltham, MA 02254, J. H Reif, Computer Science Department, Duke University, Durham, NC 27706

Preprint of paper appearing in Journal of the ACM(JACM) 41:5, September, 1994, pp. 982-1012.

space, can a given polyhedron (often referred to as a *sofa* or *piano*) be moved from the source point to the destination point without coming in contact with any of the obstacles. The *generalized mover's problem* allows the object to be moved to consist of a collection of polyhedra freely linked together at various vertices. Both the classical and generalized mover's problems have obvious applications to robotics motion planning problems and have been of interest to researchers in this field for some time (e.g., Lozano-Perez [1980], Lozano-Perez and Wesley [1979], Wangdahl et al. [1974], Vaccaro [1974]; see Schwartz et al. [1987] for further references). Although the generalized mover's problem is PSPACE-hard [Reif 1979] even for planar reachability of simple linkages [Hopcroft et al. 1982; Joseph and Plantinga 1985], the classical mover's problem can be solved in polynomial time [Reif 1979; Schwartz and Sharir 1981, 1982]. In this paper, we consider the *two-dimensional minimal movement problem*; that is, the problem in two dimensions of determining the shortest possible movement, if one exists.

We limit our attention to the movement of a single point in the plane. Besides its relevance to computational geometry, this apparently simple problem is fundamental to more general versions of the minimal movement problem. For example, we show how our algorithm can be used to compute minimal movement for a disk; this gives rise to minimal movement computations for arbitrary objects and obstacles, to within the accuracy obtainable by enclosing the object with the smallest possible disk and enclosing obstacles with polygons (in fact, our construction allows polygons to have rounded corners of arbitrary radius). One application of the efficient computation of minimal movement is to robotics. For example, consider a warehouse with a robot server that must repeatedly proceed from a source point (the service window) to various points in the warehouse (to retrieve objects).

Throughout this paper, we let n denote the size of the obstacle space (the number of obstacle edges) and k denote the number of *islands* in the obstacle space (number of connected components). For many practical problems, it may be that $k \ll n$. For example, the layout of a particular floor in an office building may be such that the hallways divide the layout into only a few distinct connected components even though it requires hundreds of thousands of edges to accurately describe the layout.

A straightforward algorithm for finding shortest paths in the plane is to construct a graph containing one vertex for each obstacle vertex and one edge for each pair of obstacle vertices that are mutually visible; (this graph has $O(n^2)$ edges and can be constructed in $O(n^2 \log n)$ time and $O(n^2)$ space) and then apply Dijkstra's single-source shortest path algorithm for graphs;[1] this approach is used by Sharir and Schorr [1984] to derive an $O(n^2 \log n)$ algorithm for finding the Euclidean shortest path between two points that avoids a set of polygons (Larson and Li [1981] use this type of approach to derive a quadratic algorithm for finding the rectilinear shortest path between two points that avoids a set of polygons and present algorithms of greater than quadratic

[1]For a graph of $|V|$ vertices and $|E|$ edges, standard implementations of Dijkstra's algorithm require $O(|V|^2)$ or $O(|E| \log |E|)$ time (e.g., see Aho et al. [1983]). Using Fibonacci heaps, a slightly better asymptotic worst-case bound of $O(|E| + |V| \log |V|)$ can be obtained [Fredman and Tarjan 1984]. In addition, specialized algorithms may be appropriate for restricted classes of graphs (e.g., Frederickson [1984] and Sedgewick and Vitter [1984]).

complexity for multiple origin-destination pairs). More recent approaches that have quadratic complexity include Welzl [1985], Asano et al. [1986], and Hershberger and Guibas [1988], Mitchell and Papadimitriou [1985] survey techniques for computing shortest paths, and the paper of Mitchell and Papadimitriou [1991] addresses shortest paths through weighted regions and contains additional references to recent work. Clarkson [1987] considers approximation algorithms.

It should be noted that more efficient algorithms are known for some special cases. Chazelle [1982] presents an $O(n \log n)$ algorithm for finding the Euclidean shortest path between two points inside a simple polygon; this algorithm is linear when combined with the linear time triangulation algorithm for a polygon of Chazelle [1990]. Guibas et al. [1986] present an algorithm for the Euclidean single-source multiple destination problem inside a simply polygon; this algorithm is also linear when combined with Chazelle [1990]. Lee and Preparata [1984] present an $O(n \log n)$ algorithm for finding the Euclidean shortest path between two points that avoids n disjoint parallel line segments. de Rezende et al. [1985] and Wu et al. [1987] present an $O(n \log n)$ algorithm for finding the rectilinear shortest path between two points that avoids a set of rectangles (with sides parallel to the coordinate axes). Clarkson et al. [1987] present an $O(n \log n^2)$ algorithm and Mitchell [1987] presents an $O(n \log n^2/\log\log n)$ algorithm for finding the rectilinear shortest path between two points that avoids a set of polygons.

Prior to this work, it has been open as to whether more efficient algorithms exist for the general Euclidean single-source multiple destination problem with polygonal obstacles. In this paper, we present an efficient algorithm for the case when the number of islands is small. Our algorithm assumes that along with the input is provided either a triangulation or a Voronoi diagram for the obstacle space.[2] Both of these data structures can be precomputed in $O(n \log n)$ time. Since they are both common general purpose data structures, we have chosen to leave the time for their computation out of the statement of our running times so that our results are independent of improved algorithms for either one of these data structures that might be available for specific classes of data.[3]

Given that a triangulation or a Voronoi diagram for the obstacle space is provided with the input, we present an $O(kn)$ time algorithm for the Euclidean plane (with arbitrary polygonal obstacles) that uses only $O(n)$ space and, given a *source* point, produces a data structure that is a triangulation of the plane with the following properties:

(1) The data structure uses $O(n)$ space.
(2) Point location queries can be answered in $O(\log n)$ time.
(3) Given the location of a point x (x is not necessarily an obstacle vertex or a point on an obstacle edge), the (Euclidean) distance from x to the source can be computed in $O(1)$ time.

[2]We mean the Voronoi diagram of the edges of the obstacle space, often called the *generalized* Voronoi diagram in the literature [Drysdale 1979; Kirkpatrick 1979; Yap 1984]; this is different from one for the points of the obstacle space [Shamos 1975; 1978].
[3]For example, when there is only a single island, Chazelle [1990] presents a linear time triangulation algorithm.

(4) The minimal length path between the source and any point x in the plane can be output in time proportional to the number of edges it contains (it must be that any minimal length path consists of a sequence of at most $O(n)$ straight line segments).

The algorithm efficiently computes the shortest path between two points for practical problems where $k \ll n$. In addition (and perhaps more importantly), once the data structure has been constructed, a sequence of queries about the distance of points from the source can be answered efficiently.

The data structure produced by this algorithm works as follows: For a set of polygonal obstacles S and a source point s, we augment S with a set of $O(n)$ line segments to form S^+, a (planar) triangulation of the finite region of the plane that contains S. Associated with each vertex v of this triangulation are the values:

$d(v)$ = length of a minimal length path between s and v,

$b(v)$ = a vertex adjacent to v in S^+ that is along a minimal path between s and v.

Hence, after S^+ has been constructed, for a vertex v it is possible in $O(1)$ time to determine the length of a minimal path between s and v (and by using the $b(\ )$ pointers, a minimal length path between s and v can be output in time proportional to the number of edges it contains). Furthermore, for each point x of the plane that is not a vertex of S^+, a minimal length path between s and x can be obtained by placing a straight line segment between x and the vertex v of the triangle T containing x such that $d(x, v) + d(v)$ is minimum, and then following $b(\ )$ pointers back to s (actually, for some of the algorithms we present, it may also be necessary to compare $d(x, v) + d(v)$ with $d(x, w) + d(w)$ where w is one of the vertices of the three triangles that shares an edge with T).

Hence, if it is known in which triangle a point p lies, then the length of a minimal length path between s and p can be determined in $O(1)$ time. If it is not known in which triangle p lies, then this can first be determined in $O(\log n)$ time. That is, it is possible to augment our data structure in $O(n \log n)$ time (the augmented structure is linear in the size of the original one) so that point location queries ("Which triangle contains p?") can be answered in $O(\log n)$ time.[4]

We assume the standard RAM model[5] of computation augmented with real variables, where the following operations on real variables can be performed in constant time:

(1) Tests of the form $x \geq 0$.
(2) Arithmetic operations of the form $x + y$, $x - y$, $x * y$, and x/y.
(3) The square root operation.

This assumption provides a simple machine independent environment in which to study practical constructive computational geometry problems, similar assumptions are commonly used by other authors in this area. In practice, the

[4]Such an algorithm for point location in a triangulated plane was first given by Lipton and Tarjan [1977] and a more practical algorithm to do this is given by Kirkpatrick [1983].
[5]See, for example, Aho et al. [1983].

time required for the operations specified above, as well as the precision that can be expected, depends on the hardware being used.

Section 2 contains basic definitions, including the definition of an obstacle space. An obstacle space can be thought of as a number of islands contained in a room with arbitrary walls, as depicted in Figure 1; the dashed line shows a shortest path between points A and B. The enclosing wall allows the notion of an obstacle space to model practical problems where movement is restricted to some finite area. Our algorithm can also handle the case where we are interested in shortest paths from the source to all points of the (infinite) plane. Section 3 examines the special case where there are no islands; the approach used for this algorithm is central for our algorithm for the case when islands are present. Section 4 presents an "island merging" algorithm that solves the single-source shortest path problem (with islands) for all vertices of the obstacle space. Section 5 augments the data structure developed in Section 4 to handle the single-source shortest path problem for all points of the plane. Section 6 generalizes the results of Section 5 to the minimal movement of a disc.

2. *Basic Definitions*

Figure 1 depicts the general flavor of an obstacle space: a straight-line planar map that consists of a polygonally bounded area, containing polygonal obstacles. We assume the reader to be familiar with formal definitions of standard notions such as a planar map, faces of a planar map, how planar maps are represented, etc. For additional background on computational geometry, the reader may refer to the textbooks of Preparata and Shamos [1988] and Edelsbrunner [1987]. Before precisely defining the single-source shortest path problem for an obstacle space, we must define the notion of an obstacle. Not only may polygons (a straight-line planar map such that each vertex has degree exactly 2) be nonconvex, but we allow objects that are not really polygons in the strict sense, because degenerate 1-dimensional objects are allowed to "hang" off polygons.

Definition 2.1. A *generalized polygon* is a straight-line planar map Q such that there exists an internal face F of Q, called the *primary face* of Q, such that all edges of Q lie along the perimeter of F or along the perimeter of the external face.

Figure 3 shows two generalized polygons: One is the line segment XY and the other is everything else in the figure (which contains the line segment XY in its primary face).

Definition 2.2. An *obstacle space* O is a finite set of generalized polygons satisfying the following properties:

(1) No two members of O intersect.
(2) Exactly one member of O, called the *enclosing wall* of O, contains all of the other members of O in its primary face.
(3) All members of O that are not the enclosing wall, called *islands*, are generalized polygons and no island contains any other islands of O in any of its internal faces.

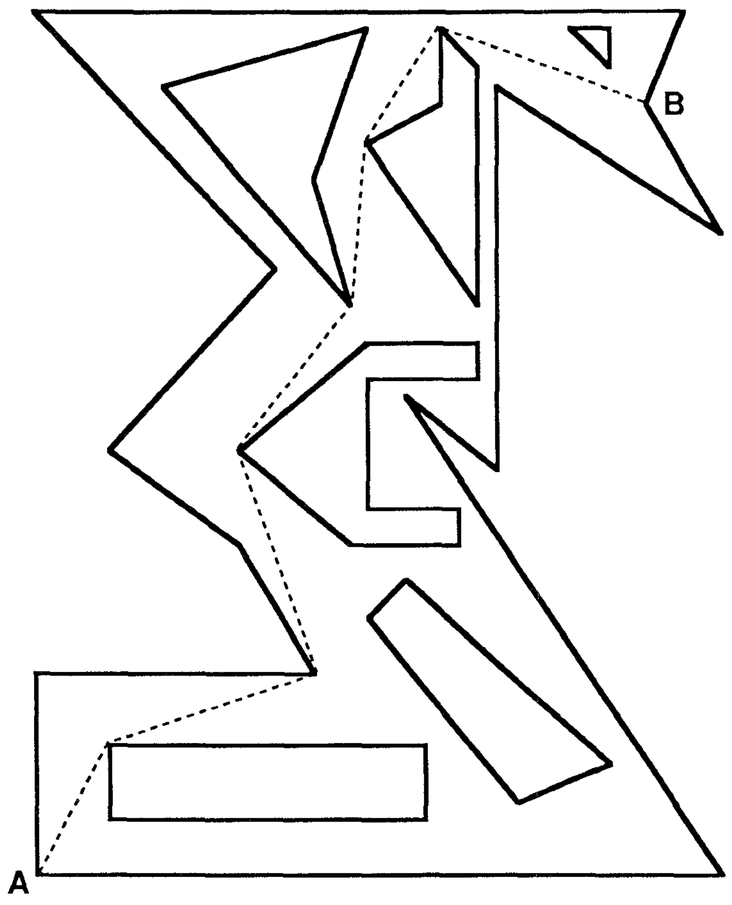

FIG. 1. An obstacle space.

Any point x of the plane that is not in the exterior face of the enclosing wall of O and not in the internal face of an island of O is referred to as a *point of* O. Note that a vertex of O is a point of O, but a point of O may not be a vertex. A *shortest path between* two points s and t of O is a minimal length 1-dimensional curve that does not cross an obstacle edge.

Figure 1 is an example of an obstacle space; the dashed line shows a shortest path between points A and B. Figure 2 is an example of an obstacle space that contains exactly one island (the interior of the island has been shaded to make it easier to distinguish). Figure 3 is also an example of an obstacle space that contains exactly one island (the line segment XY).

Several aspects of our definition of an obstacle space should be noted:

—Movement that follows obstacle edges is allowed.

—Both the enclosing wall and islands may be arbitrarily nonconvex; Figure 2 depicts an enclosing wall that "spirals" with a single island.

—Obstacles with degenerate features (features with zero width), as depicted by Figure 3, are allowed; in fact, a single point is a legal island.

—for the parameter k, the enclosing wall counts as an island, as does the source point if it is not part of the enclosing wall or an island.

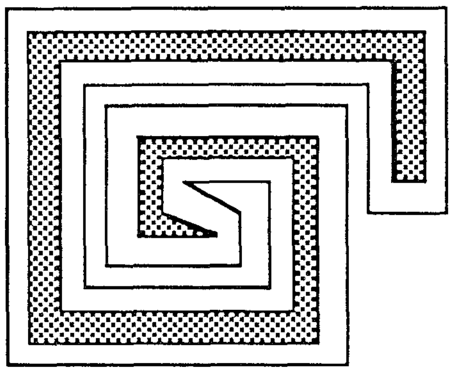
FIG. 2. Nonconvex obstacles.

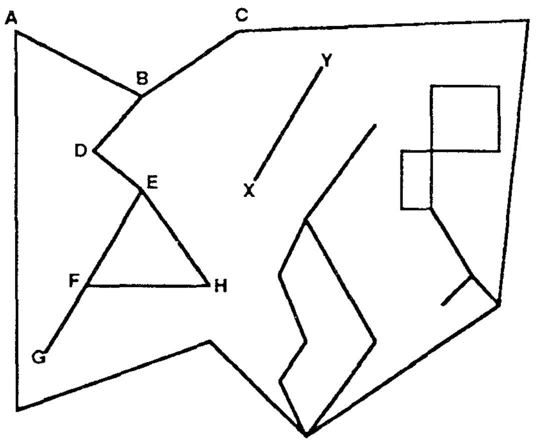

FIG. 3 Obstacles with degenerate features.

We are now ready to formally define the single-source shortest path problem.

Definition 2.3. The *2-dimensional single-source, multiple destination shortest path problem*, which we henceforth refer to as simply the *single-source shortest path problem*, is:

Input: An obstacle space O and a source point s (s is a point of O).

Output: A straight-line planar map O^+, called a *single source data structure* with the following properties:

(1) The size of O^+ is linear in the size of O, and O^+ contains O as a subset.

(2) All internal faces of O^+ are triangles (i.e., a face bounded by exactly three edges).

(3) Associated with each vertex v of O^+ are the two values:
$d(v)$: The length of the shortest path from v to s.
$b(v)$: A vertex v^+ that is adjacent to v and along a shortest path from v to s.

(4) Associated with each triangle T of O^+ are two vertices p and q, called the *exit vertices* of T,[6] such that for every point x of the plane that is contained in (or lies on an edge of) T, the shortest path from x to the source is obtained by determining which of $d(x, p) + d(p)$ and $d(x, q) + d(q)$ is smaller, travelling in a straight line from x to that vertex, and then following $b(\,)$ pointers back to s. That is, the length of a shortest path between s and x can be computed in $O(1)$ time and the path itself can be constructed in time proportional to the number of edges it contains.

[6]One of the exit vertices is of the three vertices of T. The other exit vertex (which is not needed for much of what we shall do) is a vertex of one of the three triangles that shares an edge with T.

The *unbounded* single-source shortest path problem is like the regular single-source shortest path problem except that the obstacle space does not have an enclosing wall and the data structure for the output is augmented with a set of simple nonintersecting quadratic curves that partition the infinite region of the plane; point location for these infinite regions (to find between which pair of curves a point lies) can be done in $O(\log n)$ time, the length of a shortest path between s and any point in the infinite region can be computed in $O(1)$ time, and the path itself can be constructed in time proportional to the number of edges it contains.

The following well-known fact guarantees that the above definition is well-founded:

If s and t are two points in an obstacle space O, then all minimal length 1-dimensional curves between s and t consist of a sequence of straight-line segments whose endpoints (with the possible exception of s and t) are vertices of O. Furthermore, if O contains no islands, then the shortest path from s to t is unique.

We close this section with a technical note concerning "triangles" like the face F shown in Figure 4 (face F is bounded by five vertices; however, the 4 vertices a, x, y, and d are colinear). One way to deal with such faces is to store with each edge (u, v) on the adjacency list of u in an obstacle space O the value $reach(u, v)$, which is the farthest vertex that can be reached by traveling from u in a straight line in the direction of v along the edges of O (e.g., in Figure 4 $reach(a, x) = reach(x, y) = reach(y, d) = d$). Another way is to fully triangulate by adding additional edges (e.g., in Figure 4, the additional edges (c, x) and (c, y) would be added). In this alternate representation, given a point z inside a face F, the shortest path from z to the source point is obtained by going in a straight line from x to one of the three vertices associated with the face in the original structure that contains F (this line may cross some of the new edges, but no "real" edges) and then following $b(\,)$ pointers as usual. The precise way in which faces like face F in Figure 4 are handled is not important, and we shall not address this issue further.

3. *Single-Source Problem without Islands*

In this section, we present an $O(n \log n)$ algorithm for the single-source shortest path problem with no islands. We approach this problem by first presenting an $O(n \log n)$ algorithm for the single-source, single-destination shortest path problem without islands, where a destination vertex t is specified in addition to the source vertex s and the problem is to find a shortest path from s to t that avoids obstacles. Although, as mentioned in the introduction, this simpler problem is already known to be solvable in $O(n \log n)$ time, the algorithm we present now uses techniques that motivate the more complicated algorithm for the single-source problem without islands that is presented later in the next section.

We start by describing the *shortcut* operation. The idea is as follows: Given a vertex y along the enclosing wall of the obstacle space O that forms an acute angle with its neighboring vertices x and z, if neither the source s or the destination t is in the triangle defined by x, y, and z, then we might as well

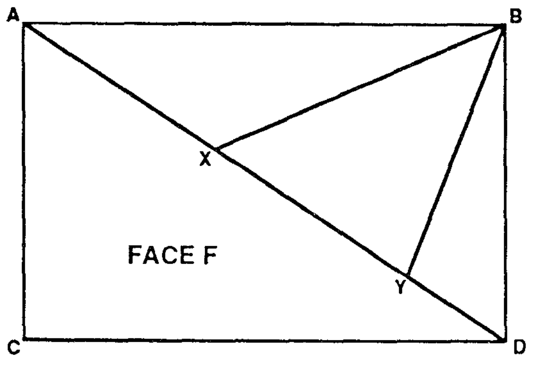

FIG 4. A pseudo-triangle.

replace the portion (x, y, z) of the enclosing wall by (x, z) and then proceed to solve the smaller problem. The problem with doing this is that the straight line from x to z may be obstructed by other portions of the enclosing wall of O, as depicted in Figure 5. In this case, the best we can do is to bend around the intruding walls as closely as possible; this is shown as a dashed line in Figure 5. We now formally define *shortcut* as the operation of replacing the path (x, y, z) by the dashed line shown in Figure 5. In fact, the following definition allows the slightly more general case where x and z are not necessarily the endpoints of the edges adjacent to y.

Definition 3.1. Let S be the perimeter of an internal face F of a planar map, let y be an acute vertex of S (with respect F), and let x and z be points that lie along the each of the two edges adjacent to x (i.e., x and z are on different edges adjacent to y, $x \neq y$, and $z \neq y$). Let C be the convex hull of the vertices x, z, and all those vertices of S that are in the interior of the triangle defined by x, y, and z. Then C is either a path from x to z consisting of colinear line segments, in which case *shortcut*(x, y, z) is defined to be C, or C is a cycle formed from two disjoint paths from x to z, in which case *shortcut*(x, y, z) is defined to be the closest of these two paths to x (i.e., if C were added to S, *shortcut*(x, y, z) would be the path that formed a face with x and was homotopic to the path (x, y, z)). Let *short_first*(x, y, z) be the vertex of *shortcut*(x, y, z) that is closest to x, and *short_last*(x, y, z) be the vertex of *shortcut*(x, y, z) that is closest to z. If *short_first*$(x, y, z) = z$ (and hence *short_last*$(x, y, z) = x$) or *short_first*(x, y, z) and *short_last*(x, y, z) are the same, then the *basis* of *shortcut*(x, y, z) is defined to be empty; otherwise, it is defined to be the path in S between *short_first*(x, y, z) and *short_last*(x, y, z) (actually, there are two such paths; we mean the one that does not contain $(x$, *short_first*$(x, y, z))$ and $($*short_last*$(x, y, z), z))$. The set *dead*(x, y, z) is portion of the plane defined by the union of the following two sets:

(1) The line segment (x, y) less the point x and the line segment (y, z) less the point z, together with the region defined by the cycle formed by (x, y), (y, z), and *shortcut*(x, y, z).
(2) The edges of the basis of *shortcut*(x, y, z) not in *shortcut*(x, y, z) together with the regions defined by these edges and the *shortcut* path.

To reduce notation, we shall simply write *short_cut*(y) and *dead*(y) when it is the case that x and z are the vertices of S that are adjacent to y (the full

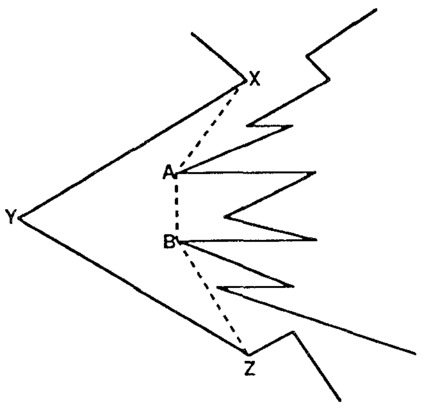

FIG. 5. The shortcut operation.

generality of *shortcut*(x, y, z) where x and z are arbitrary points on the edges adjacent to y will not be used until later in this section). The following lemma gives a formal characterization of how portions of the obstacle space that intersect *dead*(y) may be "discarded" when computing a shortest path.

LEMMA 3.1. *If y is an acute vertex in an obstacle space without islands such that dead(y) does not contain the source s or the sink t, then dead(y) is disjoint from any minimal path.*

PROOF. Assume the contrary, and suppose that a shortest path P from s to t intersected *dead*(y) where y was in an acute vertex such that *dead*(y) does not contain s or t. The only way that P can enter *dead*(y) is to cross *shortcut*(y) at some point a and the only way that P can leave *dead*(y) is to cross *shortcut*(y) at some other point b (it could be that $a = b$). Replacing the portion of P in *dead*(y) by the path from a to b along *shortcut*(y) would result in a shorter path from s to t than P, contradicting the fact that P is a shortest path from s to t.

Given the above lemma, Algorithm 3.1 (Figure 6) is a simple strategy for computing the shortest path between a source vertex s and the destination vertex t of an obstacle space O (without islands); it works by repeatedly replacing an acute vertex y (where *dead*(y) does not contain s or t) and its two incident edges by *shortcut*(y) (i.e., the portion of O that intersects *dead*(x) is "thrown out"). Before Algorithm 3.1 begins, there is a single generalized polygon (the enclosuring wall, with no islands). After the algorithm finishes, there is just a single path (a minimal length path between s and t). During the execution of the algorithm there is, in general, a sequence of generalized polygons connected by portions of a minimal length path. That is, replacing an acute vertex y (and the two edges incident to y) by *shortcut*(x) may split a generalized polygon into two generalized polygons connected by *shortcut*(y); whenever this happens, one or both of the vertices adjacent to y are enqueued if they are not already present in the queue and they have become acute as a result of the *shortcut* operation (and their dead regions do not contain s or t).

THEOREM 3.1. *Algorithm 3.1 halts after at most n iterations of the* **while** *loop and produces a minimal length path between s and t.*

PROOF. First, to verify that Algorithm 3.1 always halts after at most n iterations of the **while** loop, observe that exactly one vertex is dequeued on

```
1. Enqueue all acute vertices $y$ of $O$ such that $dead(y)$ does not contain $s$ or $t$.

2. while (queue is not empty) do begin
        a. Dequeue a vertex $y$ and let $x$ and $z$ be its adjacent vertices
        b. $P := shortcut(y)$
        c. Replace $(x,y)$, $(y,z)$, and the basis edges by $P$
        d. if (basis of $P$ is not empty) then begin
               if ($x$ is acute, not in queue, and $s,t$ not in $dead(x)$) then enqueue $x$
               if ($z$ is acute, not in queue, and $s,t$ not in $dead(z)$) then enqueue $z$
   end
```

FIG. 6. Algorithm 3.1—Single-source, single-destination, without islands.

each iteration and this vertex can never be added again (since it is deleted from O in Step (2c) of the **while** loop). Furthermore, the only vertices that can go from being obtuse to acute as a result of Step (2c) are x and z. Since Step (2c) never disconnects O, by Lemma 3.1, it must be that the straight-line planar map produced by Algorithm 3.1 together with its internal faces must contain all minimal length paths between s and t. Thus, we are left with showing that Algorithm 3.1 always transforms O to a single path; that is, the resulting planar map has no internal faces. Assume that this is not the case. Since the *shortcut* operation has the effect of transforming a polygon to two polygons connected by a simple path, it must be that the straight-line planar map M that remains after the execution of Algorithm 3.1 consists of a sequence of polygons connected by simple paths (vertices along a polygon or along a path have degree 2, vertices where paths connect to polygons have three, and a polygon has at most two vertices of degree 3). Furthermore, M must consist of at most two polygons connected by a single path because if a polygon did not contain either s or t, it would have at least on acute vertex that could be added to the queue. Since either s or t or both must be a vertex of a polygon in M (otherwise, M would be a simple path), without loss of generality assume that s is a vertex of a polygon P in M. If M is not a single polygon, then for the remainder of this proof, let t denote the vertex of degree 3 of P. Let P_1 and P_2 be the two distinct paths from s to t in P and, if they exist, let p_1 be the first acute vertex on the path P_1 from s to t and p_2 be the first acute vertex on the path P_2 from s to t (p_1 and p_2 cannot be s or t). It can't be that both P_1 and P_2 contain acute vertices because it is not possible for t to be in both $dead(p_1)$ and $dead(p_2)$. Hence, without loss of generality, assume that P_1 has no acute vertices (except possibly s or t) and $dead(p_2)$ contains t. But then the path from p_2 to t in P (that does not pass through s) must contain at least one additional acute vertex that could have been added to the queue; a contradiction.

Algorithm 3.1 is not efficient (a straightforward implementation requires linear space and quadratic time). In addition, even if Step (c) is modified to be

c. Add the line segments of P to O.

(so that the algorithm never deletes any edges of O), it may not be possible to augment the resulting planar map to form a legal single-source data structure as defined by Definition 2.3. Nevertheless, the use of the *shortcut* operation in Algorithm 3.1 motivates the approach that we shall now take for the single-source shortest path problem (without islands).

We first define two "conservative" versions of the *shortcut* operation, called *crosscut* and *extend*; essentially, these two operations take just the first edge of *shortcut* and extend it until it hits an obstacle. Given the *crosscut* and *extend* operations, the algorithm proceeds by starting at s and expanding outward to triangulate the obstacle space; that is, initially the planar map is the original obstacle space without islands, at the completion of the algorithm we are left with a planar map such that the primary face has been triangulated, and at any point during the execution of the algorithm there is a connected region of the primary face that is not triangulated.

Definition 3.2. Let S be the perimeter of an internal face F of a planar map. Let x be a vertex of S such that a vertex y adjacent to x is acute (with respect to F). Let z be the other vertex adjacent to y (i.e., $z \neq x$). Define:

$$\textit{cross\_first}(x, y, z) = \textit{short\_first}(x, y, z)$$

If *shortcut*(x, y, z) has an empty basis, then define *crosscut*(x, y, z) to be (x, z). Otherwise, *crosscut*(x, y, z) is the line segment obtained by extending the segment $(x, \textit{cross\_first}(x, y, z))$ from *cross_first*(x, y, z) until it intersects the line segment (y, z); this point of intersection is *cross_last*(x, y, z). Since the points x and y together with the face F uniquely determine the point z, we use the notation *crosscut*(x, y, F), *cross_first*(x, y, F), and *cross_last*(x, y, F) interchangeably with the above notation. The *crosscut* operation divides the original face F into *primary face* (the face containing the vertex y on its perimeter), the *secondary face* (the face containing the vertex x on its perimeter), and possibly a number of additional faces called *induced faces*. If *cross_first*(x, y, z) and *cross_last*(x, y, z) are the same, then there are no induced faces. If there are vertices of the obstacle space in addition to *cross_first*(x, y, z) that lie on the line segment from x to *cross_last*(x, y, z), then there will be more than one induced face.

Figure 7 illustrates the *crosscut* operation where $a = \textit{cross\_first}(x, y, z)$, $c = \textit{cross\_last}(x, y, z)$, and point b just happens to be colinear with a and c, causing there to be more than one induced face; note that this is a case where the technical note at the end of Section 2 applies. Figure 8 is similar to Figure 7 for the *extend* operation that we shall define next. The *extend* operation is virtually identical to the *crosscut* operation except that the line segment introduced is restricted to be at least colinear with the edge incident to x that is not (x, y).

Definition 3.3. Let S be the perimeter of an internal face F of a planar map. Let x be a vertex of S such that a vertex y adjacent to x is acute (with respect to F). LEt z be the other vertex adjacent to y (i.e., $z \neq x$). Let w be the other vertex adjacent to x (i.e., $w \neq y$). If the (infinite) line defined by the two points w and x does not intersect the line segment (y, z), then define *extend*(x, y, z) = *crosscut*(x, y, z) (and define *ex_first*(x, y, z) and *ex_last*(x, y, z) to be *cross_first*(x, y, z) and *cross_last*(x, y, z), respectively). Otherwise, let $\hat{z}$ denote this intersection point and define *extend*(x, y, z) = *crosscut*$(x, y, \hat{z})$ (and define *ex_first*(x, y, z) and *ex_last*(x, y, z) to be *cross_first*$(x, y, \hat{z})$ and

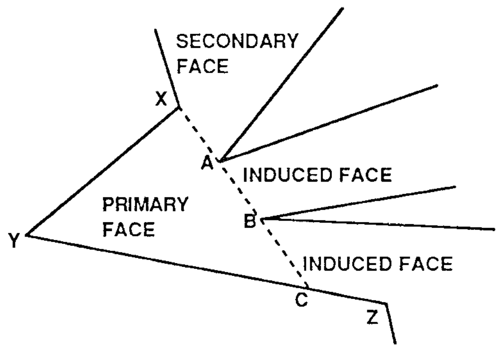

FIG. 7. The crosscut operation.

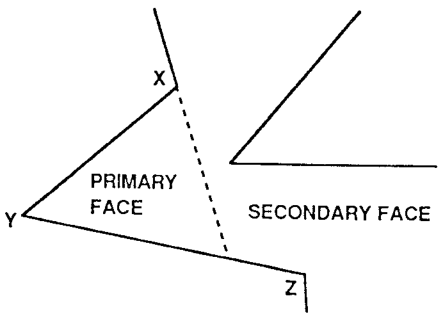

FIG. 8. The extend operation.

cross_last(x, y, $\hat{z}$), respectively). Since the points x and y together with the face F uniquely determine the point z, we use the notation *extend*(x, y, F), *ex_first*(x, y, F), and *ex_last*(x, y, F) interchangeably with the above notation. As with the *crosscut* operation, the *extend* operation divides the original face F into the *primary face* (the face containing the vertex y on its perimeter), of this *extend* operation, the *secondary face* (the face containing the vertex x on its perimeter), and a number of *induced faces*.

LEMMA 3.2. *Let x be a vertex on the perimeter of a face F such that a vertex y adjacent to x is acute. Then the shortest path between x and any point p in F may have points in common with at most one of the primary, secondary, and induced faces of crosscut(x, y, F) and at most one of the primary, secondary, and induced faces of extend(x, y, F).*

PROOF. This should be clear from inspection of Figures 6 and 7.

Algorithm 3.2 is the procedure *SPACE*[7] which takes two arguments, a vertex x and a face F; it will always be the case that the shortest path from any vertex on F to the source point must pass through x. Given an obstacle space O (without islands) and source vertex s, the initial call is *SPACE*(s, F), where F is the primary face of O. In order to simplify the presentation, Algorithm 3.2 makes use of the following procedure:

```
procedure UPDATE(x,y) :
  $d(y) := d(x) + \|(x, y)\|$
  $b(y) := x$
```

[7]After the writing of this paper, a similar approach was considered in Guibas et al. [1986].

When a minimal path between s and a vertex x is known and it has also been determined that a minimal path between s and a vertex y passes through x, $UPDATE(x, y)$ is called. Algorithm 3.2 works as follows: Case 1 is the degenerate condition where F is a simple triangle; and so it suffices to simply update points y and z and return. In Case 2, there is an acute vertex adjacent to x to which we can apply the *crosscut* operation; so the *crosscut* edge is added in Step (2b) and then the algorithm is called recursively on the secondary, induced, and primary faces in Steps (2c), (2d), and (2e). In Case 3 (where there is not an acute vertex adjacent to x), Step (3a) searches along the perimeter of F from x to find an acute vertex, Step (3b) works its way back to x by applying the extend operation to each vertex, and Step (3c) calls the algorithm recursively on the portion of F that remains.

Each triangle of the data structure produced by Algorithm 3.2 can have exactly one of its three vertices designated as the exit vertex (there is no need here for more than one exit vertex per triangle). We have not bothered to include this labeling with the presentation of Algorithm 3.2. In fact, in practice, it may be simpler not to bother with the labeling and when given a point x in the interior of a triangle defined by the three vertices (v_1, v_2, v_3), compute the minimum of $d(x, v_1) + d(v_1)$, $d(x, v_2) + d(v_2)$, and $d(x, v_3) + d(v_3)$ to determine through which vertex of the triangle to exit.

THEOREM 3.2. *Algorithm* 3.2 (*when called with x as the source and F as the primary face*) *always halts and transforms the obstacle space into a single-source data structure.*[8] *In addition, assuming that a triangulation or a Voronoi diagram for the obstacle space is provided with the input, it can be implemented to run in* $O(n)$ *time and space.* (*See Figure* 9.)

PROOF. To verify that the algorithm always halts, observe that all recursive calls are made on a face with a number of edges on its perimeter that is at least one less than the number of edges on the perimeter of the face of the current call. Algorithm 3.2 is initially called with s as the first argument and the primary face of the obstacle space as the second argument.

To verify that the algorithm always transforms the obstacle space into a single-source data structure, it suffices to show that for each recursive call $SPACE(x, F)$ it is always true that the shortest path from any point in F to s must pass through x. For the recursive calls of Step (2), this follows directly from Lemma 3.2. For the recursive calls of Step (3), we observe that the shortest path between x and $\hat{x}$ must follow the edges of the perimeter connecting x and $\hat{x}$.

We now consider the running time of the algorithm. If we exclude the time spent on *crosscut* and *extend* operations, the running time of the algorithm is proportional to the number of new edges introduced (by the *crosscut* and *extend* operations); since either a new *crosscut* or *extend* edge forms a triangle for the primary face (and the new edge can be "charged" to the vertex of this triangle that is not common to this new edge) or the new edge passes through two vertices of F (and since edges can never cross, only a linear number of pairs can be connected).

Let us now consider the implementation of the *crosscut* and *extend* operations. The obvious implementation is to simply follow the faces of the triangu-

[8] In fact, every triangle will have exactly one exit point, which is one of its three vertices.

```
case 1, $F$ is a triangle with vertices $x$, $y$, and $z$: UPDATE(x,y); UPDATE(x,z):

case 2, A vertex $y$ adjacent to $x$ (on $perimeter(F)$) is acute w.r.t.$F$:
    a) Compute $crosscut(x, y, F)$ and let:
        $u := cross\_first(x, y, F)$
        $v := cross\_last(x, y, F)$
        $F_p :=$ primary face of $crosscut(x, y, F)$
        $F_s :=$ secondary face of $crosscut(x, y, F)$
    b) Add $crosscut(x, y, F_p)$ to $G$
    c) SPACE($x$, $F_s$)
    d) for each induced face $F_i$ of $crosscut(x, y, F)$ do begin
        UPDATE(x,u);
        SPACE($u$,$F_i$);
        end
    e) SPACE($x$,$F_p$)

case 3, Both vertices adjacent to $x$ (on $perimeter(F)$) are obtuse w r.t. $F$:
    a) $\hat{x}$:=$x$; $y$:=$clock(\hat{x}, F)$
    while ($y$ is obtuse w.r t. $F$) do begin
        UPDATE($\hat{x}$,y)
        $\hat{x}$:=$y$; $y$:=$clock(\hat{x}, F)$
        end
    b) while ($\hat{x} \neq x$) do begin
        while ($\hat{x}$ is not 180 degrees w.r.t. $F$) do begin
            Proceed exactly as with case 2 except use.
                $\hat{x}$ for $x$
                $extend$ for $crosscut$
                $ex\_first$ for $cross\_first$
                $ex\_last$ for $cross\_last$
            end
        $\hat{x} := c\_clock(\hat{x}, F)$
        end
    c) SPACE($x$,$F$)
```

FIG. 9. Algorithm 3.2—SPACE(x, F) single-source, shortest path algorithm, without islands.

lation or Voronoi diagram. That is, by following the line segment from x to z through the faces, we will discover all obstructing vertices and edges, and hence be able to compute the *crosscut* or *extend* path to be tangent to the vertex or edge that protrudes most towards y. The problem with this is that the number of vertices and edges that we encounter in this process may be large (possibly $O(n)$). The key observation is that these are vertices and edges that we will, at some point, have to incorporate anyway, so that we can simply process them as they are encountered. That is, we can modify Algorithm 3.2 as it has been presented thus far, so that each computation of a *crosscut* or *extend* operation is implemented via a sequence of recursive calls. For example, Figure 10 depicts six vertices. A through F that might be encountered in *crosscut*(X, F), where F denotes the single face defined by Figure 10 less than the dashed lines; here, Case 2 of Algorithm 3.2 would expand to:

UPDATE(X, A); SPACE(A, F_B)
UPDATE(A, B); SPACE(B, F_C)
SPACE(X, F_D)

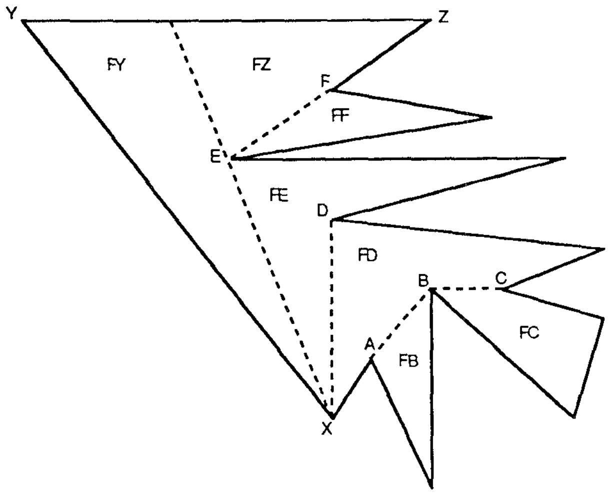

FIG. 10. Vertices encountered in a crosscut operation.

SPACE(X, F_E)
UPDATE(X, E); SPACE(E, F_F)
SPACE(E, F_Z)
SPACE(X, FY)

A "visibility stack" must be maintained in order to compute in $O(1)$ time the first component in a call to space; for example, when vertex C is encountered in Figure 10, we must be able in $O(1)$ time to find vertex B, the first vertex on a minimal path back to X, before making the call SPACE(B, F_C). The visibility stack for the operation *crosscut*(X, F), in Figure 10 would be manipulated as follows:

PUSH(X)
PUSH(A)
PUSH's and corresponding POP's when computing SPACE(A, F_B)
PUSH(B)
PUSH's and corresponding POP's when computing SPACE(B, F_C)
PUSH(C)
POP(C)
POP(B)
POP(A)
PUSH's and corresponding POP's when computing SPACE(X, F_D)
PUSH(D)
POP(D)
PUSH's and corresponding POP's when computing SPACE(X, F_E)
PUSH(E)
PUSH's and corresponding POP's when computing SPACE(E, F_F)
PUSH(F)
POP(F)
PUSH's and corresponding POP's when computing SPACE(E, F_Z)

Since each vertex is placed on the visibility stack at most once, the total time for maintaining the visibility stack is $O(n)$.

4. *Island Merging*

The approach of the last section for when no islands are present was to start at the source vertex and "grow" outward to triangulate the obstacle space. As this process progresses, the source point effectively moves, so that at any point in time, the minimal length path from the source to any point in an as yet unexplored region of the obstacle space goes through a unique "virtual" source vertex. However, when islands are present, it is not clear which way to go around them. Recall that we let k denote the number of islands in the obstacle space (and that the enclosing wall counts as one of the islands).

In this section we present an algorithm that connects the islands to produce an obstacle space with no islands but with as many as $O(k)$ "virtual" source points. The island merging algorithm computes the shortest path from the source to all vertices of the obstacle space; the next section will discuss how the data structure produced by the island merging algorithm can be triangulated so that shortest paths to points in the obstacle space that are not vertices can also be calculated.

Definition 4.1. A path P between two points x and y in an obstacle space O with source point s and virtual source points $s_1 \cdots s_k$ is *safe* if for any point p of O, there is a minimal length path between p and s that is allowed to pass through virtual source points but does not cross P (however, p may intersect P).

LEMMA 4.1. *Let P be a shortest path between the source s and some point p in an obstacle space O. Then for any two points x and y on P, the path (x, y) is safe.*

PROOF. Assume the contrary; that is, suppose that there is a point p in O for which there is no shortest path to s that does not cross (x, y). Let Q be a shortest path between q and s; going from q to s along Q, let a be the first point (excluding q) that is in (x, y) and going from a to s on Q let b be the first point (excluding a) that is in P (it may be that $b = s$). Then, the path constructed by going from p to a along P, from a to b along Q, and then from b to s along P is shorter than going from a to s along P, which contradicts P being a minimal length path between p and s.

The idea behind the island merging algorithm to be presented shortly is to successively link the islands of the obstacle space together with safe paths. The following lemma provides the mechanism for doing this.

LEMMA 4.2. *Given an obstacle space O consisting of an enclosing wall W together with a single island I such that the source vertex s is along the perimeter of I, assuming that a triangulation or a Voronoi diagram for O is provided with the input, then it is possible to compute in $O(n)$ time a safe path P between s and some point w of W.*

PROOF. A simple approach for finding such a path P is to first find a vertex v of I and a vertex w of W such that v and w are mutually visible, form the planar map $\overline{O}$ by adding the line segment (v, w) to O (by construction this line segment cannot cross any edges of O), and then run Algorithm 3.2 to compute a minimal length path P in $\overline{O}$ from $\underline{s}$ to w. Because w is an endpoint of the edge (v, w), P cannot cross (v, w) in $\overline{O}$. Hence, P is also a minimal length path between s and w in O, and by Lemma 4.1, it follows that P is safe. Furthermore, the time to compute P is $O(n)$, provided that the vertices v and

w can be found in $O(n)$ time. This can be done by letting v be a leftmost vertex of I (taking v to be any point on the convex hull of I will do) and then using the triangulation or Voronoi diagram to identify a vertex w of W that is visible from v. Finding a leftmost vertex of I can be done in $O(n)$ time by simply examining all vertices of I and selecting the one with smallest horizontal coordinate.

Algorithm 4.1 (Figure 11) is the island merging algorithm. As depicted in Figure 12, it takes as input an obstacle space and a source point and produces as output the obstacle space together with a set of k safe line segments that cause the obstacle space to be a single straight-line planar map.

LEMMA 4.3. *Assuming that provided with an obstacle space O is a triangulation or a Voronoi diagram, Algorithm 4.1 uses time $O(kn)$ and space $O(n)$ to augment O with a set of at most k safe paths that cause O to become a single planar map.*

PROOF. By Lemma 4.1, the path P added in Step (1) must be safe. In addition, each of the iterations of the **while** loop of Step (4) adds a safe path and leaves O_{cut} as a connected planar map. Analysis of the running time goes as follows: Using Lemma 4.2, Step (1) can be done in $O(n)$ time. Step (2) can be done in $O(1)$ time. Step (3) can be done in $O(n)$ time. Hence, it suffices to verify that each of the at most $k - 1$ iterations of the **while** loop of Step (4) can be done in $O(n)$ time. Step (A) can be done in $O(1)$ time. By applying Lemma 4.2 and then Algorithm 3.2, Step (B) can be done in $O(n)$ time; although the data structure is not a single generalized polygon, in this step, Algorithm 3.2 is only being applied to a portion of the data structure that is (the territory between p and the virtual source points). Note also that the construction of Algorithm 3.2 that is outlined in the proof of Theorem 3.3 is easily modified to work when a triangulation or Voronoi diagram for O is used in place of the one for O_{cur}. Step (C) can be done in $O(n)$ time by traversing the triangulation for Voronoi diagram. The data structure updates of Step (D) can easily be done in $O(n)$ time.

THEOREM 4.1. *Given an obstacle space together with its Voronoi diagram and a source point s, assuming that a triangulation or a Voronoi diagram for the obstacle space is provided with the input, a shortest path data structure for s to all vertices of the obstacle space*[9] *can be computed in $O(kn)$ time and $O(n)$ space.*

PROOF. First, Algorithm 4.1 (island merging algorithm) can be run. Second, we can label each vertex of the obstacle space with weight infinity. Third, for each virtual source point, we can run Algorithm 3.2 (single-source shortest paths without islands) and update all vertices reachable from that source point.[10]

COROLLARY 4.1a. *Assuming that a triangulation of a Voronoi diagram for the obstacle space is provided with the input, the single-source, single-destination problem can be solved in $O(kn)$ time and $O(n)$ space.*

[9]That is, each vertex of the obstacle space will be correctly labeled with its $d(\,)$ and $h(\,)$ values but there is no guarantee that these values can be used to compute shortest paths for arbitrary points of the plane in the obstacle space.

[10]The obvious implementation of this third step takes time $O(kn)$ (which suffices for this proof); however, this can be done more efficiently by maintaining a priority queue and running Algorithm 3.2 in parallel in a breadth-first fashion from all of the virtual source points.

```
(1) Initialize $O_{cur}$ to be the subset of the obstacle space consisting of the enclosing wall $W$ together with
    the island $I$ containing the source point $s$. If $O$ is two components (i.e., $I \neq W$), then let $P$ be a shortest
    path between $s$ and some vertex $v$ of $W$ and add $\hat{P}$ to $O$, where $\hat{P}$ is the portion of $P$ that does not
    intersect $I$ or $W$.
(2) Initialize the set of virtual source points $S$ to be the source point and assign the source weight 0
(3) Initialize $O_{islands}$ to be all islands that are not in $O_{cur}$.
(4) while ($O_{islands}$ is not empty) do begin
    (A) Let $I$ be any island of $O_{islands}$ and let $v$ be one of its vertices
    (B) Compute the shortest path $P$ from $v$ to the source in $O_{cur}$ (which may pass through virtual source
        points); do this by computing the shortest paths from $v$ to all virtual source points (and then add
        in the weights of the virtual source points to determine which path is shortest).
    (C) Going from $s$ to $v$ along $P$, let $x$ be the first point that $P$ intersects a vertex or edge of an island $\hat{I}$
        of $O_{islands}$ (it could be that $x$ is $v$ and $\hat{I} = I$). Going from $x$ to $s$ along $P$, let $y$ be the first point
        that is a vertex of $O_{cur}$ and let $\hat{P}$ be the sub-path of $P$ consisting of the line segment $(x, y)$.
    (D) Modify the data structure as follows:
        • Remove $\hat{I}$ from $O_{islands}$.
        • Add $\hat{P}$ and $\hat{I}$ to $O_{cur}$.
        • if $y$ is not in $S$ then add $y$ to $S$ ($weight(y)$ = distance to $s$ along $P$)
end
```

FIG. 11. Algorithm 4.1—Island-merging algorithm.

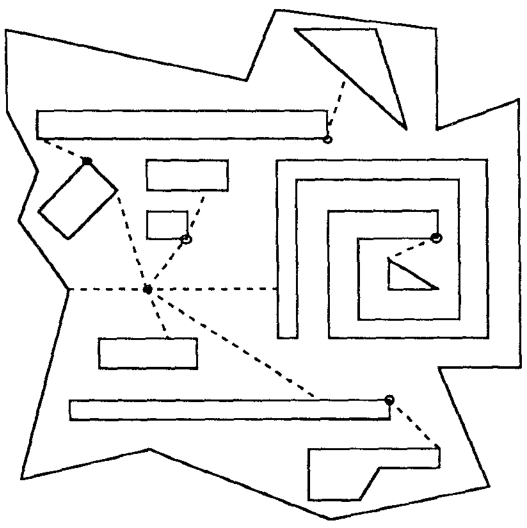
FIG. 12. Island merging.

PROOF. This follows directly from the above theorem. In addition, an alternate approach that is a bit simpler than the proof of the above theorem is to proceed as follows. Given a source point s, we can first run Algorithm 4.1. Then, given any destination point t, we can run Algorithm 3.2 (the single-source shortest path algorithm without islands) with t as the source point to compute the shortest path from t to all virtual source points that are reachable from t. Finally, the best of these at most k possible paths can be chosen (i.e., add the length of these paths to the weights of their respective virtual source points to determine which one is best).

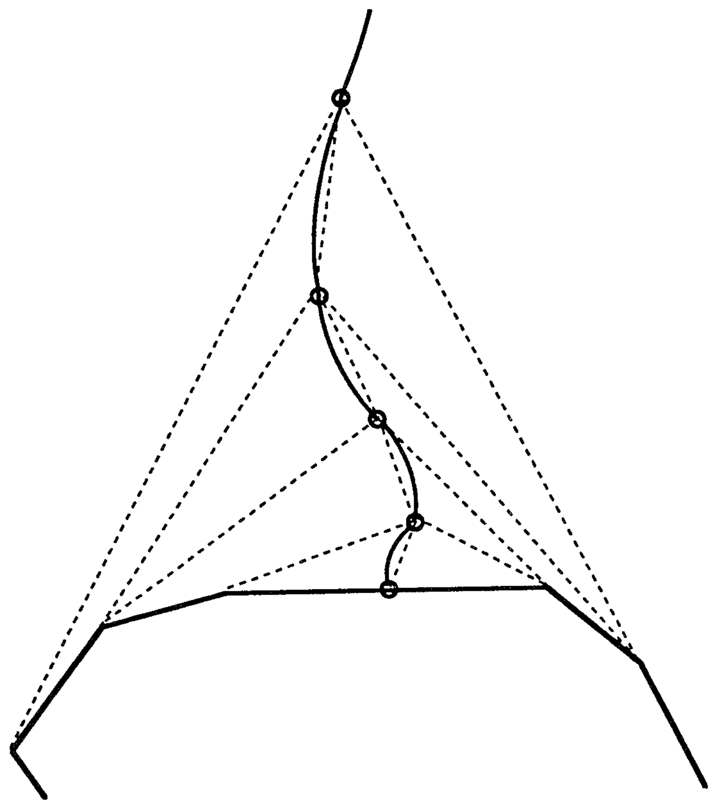

FIG. 13. A style.

5. *The Single Source Problem*

The construction of the last section does not, in general, fully triangulate the plane to yield a solution to the single-source shortest path problem. Instead, it may leave a number of untriangulated regions with virtual source points along their border. The following definition introduces the notion of a "flower" data structure that contains quadratic curves that are ridge points between portions of these regions with different shortest paths back to the source. This definition and the lemmas following it provide the machinery to triangulate such regions. Before proceeding, it may help the reader to look ahead to Figures 13, 14, and 15 to get an idea of where we are heading.

Definition 5.1. A *stamen* is a connected straight-line planar map whose perimeter consists of the union of a finite set of polygons such that any two of these polygons intersect at most a single vertex. A vertex on the perimeter of this stamen that is acute with respect to the external face is a *critical* vertex. Critical vertices have *weights* associated with them (which in practice will be the distance between this critical vertex and the source). A stamen must have the property that cyclic traversal of its critical points (as defined by the ordering induced by the perimeter of the stamen) produces a (not necessarily convex) polygon, called the *perianth*. An *anther* of a stamen is a critical vertex together with the two consecutive edges of the perimeter of the stamen that

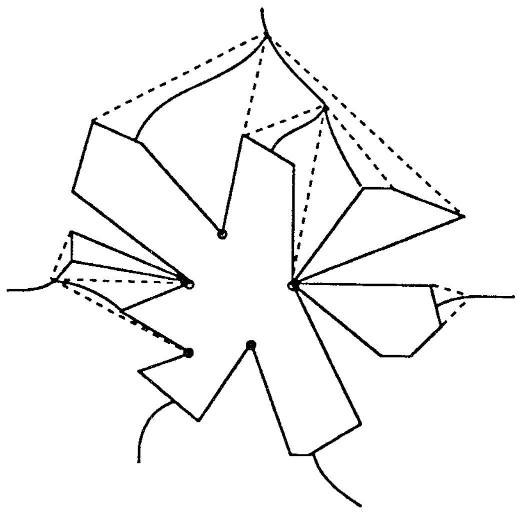
FIG. 14. An outward-growing flower.

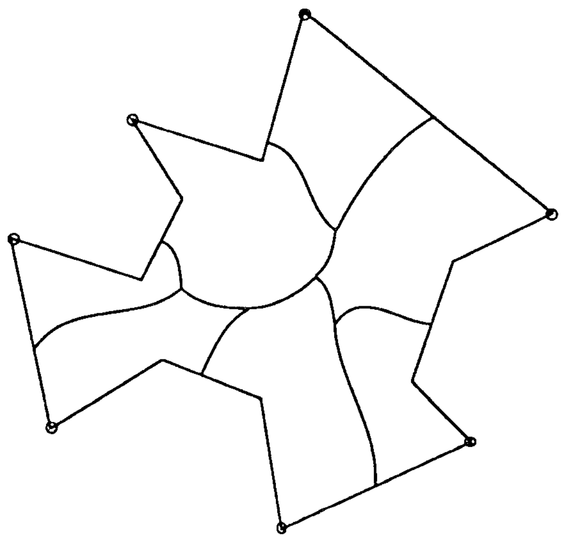
FIG. 15. An inward-growing flower.

are incident to this critical point; note that the two anthers can share a common critical point and at most one common edge. Two anthers are said to be *consecutive* if no critical point of any other anther lies between the critical points of these anthers when traversing the perimeter of the stamen. The anthers partition the perimeter into convex paths called the *petals* of the stamen, that connect two consecutive anthers. A *outward growing flower* is a planar map that consists of a stamen together with a lattice of one dimensional curves, called *styles*.

As shown in Figure 13, each style is a sequence of quadratic curves that is constructed as follows: Initially, for each petal, a style originates from the point on the petal, called the *base* of the style, that is equal distance from either of the petal's associated anthers (i.e., the geometric distance to each anther added to the weight of the anther is the same). The points on the style are those for which the minimum distance to the portion of the petal on one side of the base is the same as the minimum distance to the other side of the petal. Each style is terminated at the first point that it intersects another style or the stamen. For each such intersection point, a new style is originated based on the two anthers to either side of the anthers delimiting the styles forming the intersection point. This process is repeated until no new intersection points are introduced.

We leave it to the reader to verify that styles are well defined and that the number of curves that compose a given style is no more than the number of vertices on the corresponding petal. Furthermore, a "joint" point that connects curve c_i to curve $c_i + 1$ of a given style can be located by projecting one of the edges of the petal until it intersects c_i.

The *head* of a style is one of the following points:

—If the style is finite, then the head is the other end from the base.
—If the style is infinite and has no joints (it consists of a single quadratic curve), then the head of the style is the base.
—If the style is infinite and has at least one joint, then the head is the last joint (traveling away from the base).

Given a style s with head h and base b that separates two critical vertices v_1 and v_2, the *bounding pseudo-triangle* for s is the polygon formed from the following edges:

—The line segment (h, x), where x is the first line segment on the shortest path from h to v_1, and the line segment (h, y), where x is the first line segment on the shortest path from h to v_2; these two line segments form the *peak* of the bounding pseudo-triangle.
—The portion of the petal defined by v_1 and v_2 between x and y; this is called the *base* of the bounding pseudo-triangle.

Even though the base of a bounding pseudo-triangle is not in general a line segment, it is convenient to think of the base as a line-segment and we shall henceforth refer to a bounding pseudo-triangle as simply a *bounding triangle*.

Figure 14 illustrates an outward growing flower with eight petals and eight anthers (critical points are circled); the dashed lines are the bounding triangles.

Definition 5.2. An *inward growing flower* is defined in a similar fashion to an outward growing flower except that the stamen surrounds the styles.

Figure 15 illustrates an inward growing flower.

Fact 5.1 (*Facts about Bounding Triangles*). The reader may verify the following facts about bounding triangles:

(1) The bounding triangles of two styles that share a common endpoint share a common edge.
(2) A given style cannot intersect the bounding triangle of another style (except at its endpoint).
(3) The base of a bounding triangle can contain at most one virtual source point; that is, the base of a bounding triangle consists of a sequence of at most 2 convex paths.

LEMMA 5.1. *Given a stamen of size m with h anthers, the styles for the corresponding flower can be constructed in $O(hm)$ time.*

PROOF. Our basic approach is to make $h - 1$ passes; on each pass, in $O(m)$ time another style is added. The constructions for inward and outward growing flowers are virtually identical. The main difference is that with inward growing flowers all styles must terminate by intersecting another style or by intersecting the stamen, whereas with outward growing flowers, a style may end as a semi-infinite simple quadratic curve; in either case, we say that the style *terminates*. Going clockwise, let the h anthers be labeled $a_1 \cdots a_h$. We start by constructing the style that is determined by a_1 and a_2, call it s_1, until it terminates. Next, we extend the style determined by a_2 and a_3, call it s_2, until it terminates. If s_2 terminates by intersecting s_1 at a point x_1, then delete the portion of s_1 extending outward from x_1 and grow s_2 outward from x_1 (based on the anthers a_1 and a_3) until it terminates. At stage i, the style s_i that is determined by a_i and $a_i + 1$ is grown outward until either it terminates or intersects some style s_j, $1 \leq j < i$, at a point x_i; the portion of s_j extending outward from x_i is removed and then s_i is extended outward from x_i until it terminates. The key observation to verify that each phase can be computed in $O(m)$ time is when a style is being "grown," it is only necessary to check for intersection with the stamen and with at most two bounding triangle edges; given Fact 4.1, intersection with a bounding triangle edge implies intersection with the corresponding style (and the section of this style that is intersected can be determined by traversing the style).

Given the notion of an inward growing flower, it is possible to resolve the "holes" left by the construction of the last section.

THEOREM 5.1. *Assuming that a triangulation or a Voronoi diagram for the obstacle space is provided with the input, the single-source shortest path problem can be solved in $O(kn)$ time and $O(n)$ space.*

PROOF. After applying the construction of the last section, the obstacle space is fully triangulated except possible for at most $k/2$ regions whose borders form stamens;[11] the anthers of these stamens are virtual source points. We can now apply the construction of Lemma 5.1 to build flowers in these regions. Next, the peaks of all bounding triangles can be added and then the faces can be fully triangulated, as illustrated in Figure 13. Note that here we may need to have two exit vertices associated with a triangle T; one that is a vertex of T and one that is a vertex of one of the three triangles that share an edge with T.

THEOREM 5.2. *Assuming that a triangulation or a Voronoi diagram for the obstacle space is provided with the input, the unbounded single-source shortest path problem can be solved in $O(kn)$ time and $O(n)$ space.*

PROOF. We can first surround the obstacle space with a triangle and apply Theorem 5.1. Second, we can remove the triangle and all edges incident to its vertices to form a stamen. Third, we can apply the construction of Lemma 5.1 to build an outward growing flower. Given an outward growing flower, such as

[11]Actually, such a region may not quite satisfy the definition of stamen in that a petal may have at most one acute vertex (i.e., the petal consists of sequence two convex paths). However, this is easily accommodated in the construction of Lemma 5.1.

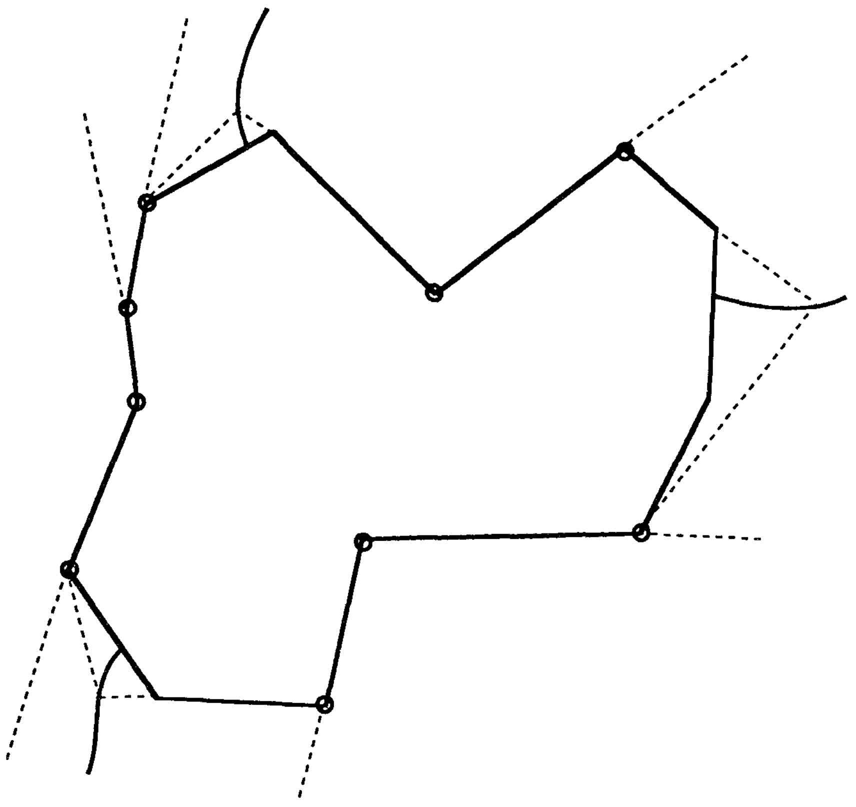

FIG. 16. Partitioning the exterior region.

the one shown in Figure 16, we can add the bounding triangle peaks and rays emanating from each petal vertex, shown by dashed lines in Figure 16 (this introduces new virtual source points). The areas inside the bounding triangles can be handled as in Theorem 5.1. In addition, we can assume that the perianth is convex, since points inside its convex hull can be handled using Theorem 5.1. With the addition of the dashed lines, the exterior region is now divided into sectors and point location can be done as follows: Each sector is either bounded by two rays (which meet at a virtual source point) or it is bounded by two rays and a bounding triangle peak and contains a simple quadratic curve (with no joints) that emanates from the head of the bounding triangle. Given a point p in the infinite region, we can determine in which sector it lies with a simple binary search procedure that works as follows:

Let $r_1 \cdots r_m$ be the rays listed in clockwise order. Construct the path P consisting of r_1, $r_m/2$, and a line segment connecting the source points that these two rays emanate from (this is a chord across the perianth). Now by checking three inequalities, we can determine on which side of P the point p lies. Next, we choose either the ray $r_m/4$ or $r_3m/4$ (depending on which side of P that p lies), without loss of generality suppose it is $r_m/4$, and then check which side of $r_m/4$ p is on. This process continues for at most $\log n$ steps.

Once the sector containing p has been determined, we are done if it is a sector not containing a curve (we can travel via a straight line to the associated virtual source point) or we must first check one quadratic inequality to determine on which side of the quadratic curve that p lies.

6. *Minimal Movement of a Disc*

It is known how to compute efficiently whether the movement of a disk between two points in the presence of polygonal obstacles is possible [O'Dunlaing et al. 1983], and Chew [1985] presents a $O(n^2 \log n)$ algorithm for shortest path movement of a disc. In this section, we show how to generalize the construction of the last section (with only a $O(1)$ increase in time and space) to movement of a disc. Our approach is to first show how an obstacle space O can be "padded" to form a new space $D_r(O)$ in which movement of a point is equivalent to movement of a disc of radius r in O. After this has been done, we show how to modify the construction of the last section to handle more general classes of obstacle spaces that arise in the transformation of O to $D_r(O)$.

Definition 6.1. For a real $r \geq 0$, D_r denotes the planar closed disc of radius r; D_r is said to be located at a point p if its center is p. Let p be a point in an obstacle space O with source point s. We say p is *r-acceptable* with respect to O if when D_r is located at p its boundary does not include any point in a face of an island of O. We say p is r-reachable with respect to O and s if D_r can be continuously translated from s to p without being located at a point that is not r-acceptable with respect to O. A path between s and p is *r-legal* with respect to O and s if it passes only through points that are reachable with respect to O and s. The set of *r-critical* points with respect to O, $D_r(O)$, is the set of points which are positions of D_r when the boundary of D_r contains at least one point that lies on an edge of O, but no point that lies on an edge of O is contained in the interior of D_r. In addition, we let $D_r(O, s)$ denote the subset of $D_r(O)$ consisting of those points that are r-reachable with respect to O and s. When r, O, and s are understood, we simply talk about a point being acceptable, reachable, or critical and a path being legal.

LEMMA 6.1. *The edges of $D_r(O)$ are either straight line segments or arc segments of radius r. $D_r(O, s)$ forms a planar map such that $D_r(O, s)$ together with the points of its internal faces is the set of reachable points.*

PROOF. Left to the reader.

LEMMA 6.2. *Given an obstacle space O of size n together with its Voronoi diagram V and a real $r \geq 0$, $D_r(O, s)$ can be constructed in $O(n)$ time.*

PROOF. It suffices to show how to construct $D_r(O)$, since any "closed off" areas of $D_r(O)$ (all regions except the one containing the source) can be discarded to get $D_r(O, s)$. In $O(n)$ time, we can decompose O into a set of S of convex polygons that intersect only on their boundaries; call each element of S a *fundamental polygon*. For each H in S, let

$$C_H = \{p\colon \textit{p is equal distance between a point of H and a point of } O - H\}$$

be the *fundamental cycle* for H. Then C_H forms a cycle of edges of V (and it must be that the interior of C_H contains the interior of H). Let X_H be the set of points in $D_r(O)$ that are in C_H or its interior. Then:

$$D_r(O) = \bigcup_{H \in S} X_H.$$

Hence, it suffices, for each H in S, to compute X_H. Let $e_0, e_1, \ldots, e_{m_H} = e_0$ be the edges of C_H listed in clockwise order where $e_i = (u_i, v_i)$. Let a_i and b_i be the points of H that are the closest to u_i and v_i respectively. Since H is convex, then either $a_i = b_i$ or a_i and b_i lie on the same edge of H. Let R_i be the region enclosed by the simple cycle formed by e_i and the (possibly trivial) line segments (a, b), (a, u), and (b, v). Let:

$$X_{H,i} = R_i \cap X_H.$$

Then since

$$X_H = \bigcup_{i=1}^{m_H} X_{H,i},$$

it suffices to compute $X_{H,i}$ for each i. Since e_i is a quadratic curve segment and hence convex, $X_{H,i}$ can consist of at most two curves, both of which are either straight-line segments (parallel and at distance r from (a,b)) or arc segments (of radius r centered at $a = b$).

Figure 17 depicts an Island consisting of three fundamental polygons and it shows the curves C_H and X_H for the middle fundamental polygon H; in addition, for a "typical" region R_i are labeled the points u_i, v_i, a_i, b_i, x_i, and y_i (with the subscript i dropped to make the figure more readable).

Thus, X_H can be computed in time $O(m_H)$ by simply considering each R_i and computing $X_{H,i}$ in $O(1)$ time. Since each edge in V appears in at most two different fundamental cycles, it must be that

$$\sum_{H \in S} m_H$$

is $O(n)$ and hence the total time to compute $D_r(O)$ is $O(n)$.

Definition 6.2. A *rounded triangle* is a cycle consisting of three curves, where each curve consists of an arc segment connected to and tangent to a line-segment connected to and tangent to an arc segment. Furthermore, each arc segment that forms one of the segments of an edge bends towards the interior of the cycle; that is, if the arc segment is completed to a circle, then the interior of this circle does not intersect the interior of the cycle.

Figure 18 is an example of a rounded triangle. Note that given any point in the interior of a rounded triangle, it is possible in $O(1)$ time to compute the minimal length path between this point and any of the three vertices of the triangle.

We now generalize the definition of the single-source shortest path problem to the problem of partitioning the obstacle space into rounded triangles that allow us to compute in $O(1)$ time for any point p inside such a triangle the length of a minimal length pair for moving a disc from p back to the source.

Definition 6.3. The *single-source shortest path problem for a disc* is:

Input: A radius $r \geq 0$ and an obstacle space O with source point s.
Output: A planar map O^+ with the following properties:

(1) The size of O^+ is linear in the size of O and O^+ contains O as a subset.
(2) All internal faces of O^+ are rounded triangles.
(3) Associated with each vertex of O^+ are two values:

d(x): The length of the shortest r-legal path from x to s (this length is ∞ if no such r-legal path exists).

b(x): A vertex x^+ on the perimeter of the face containing x that is along a shortest r-legal path from x to s.

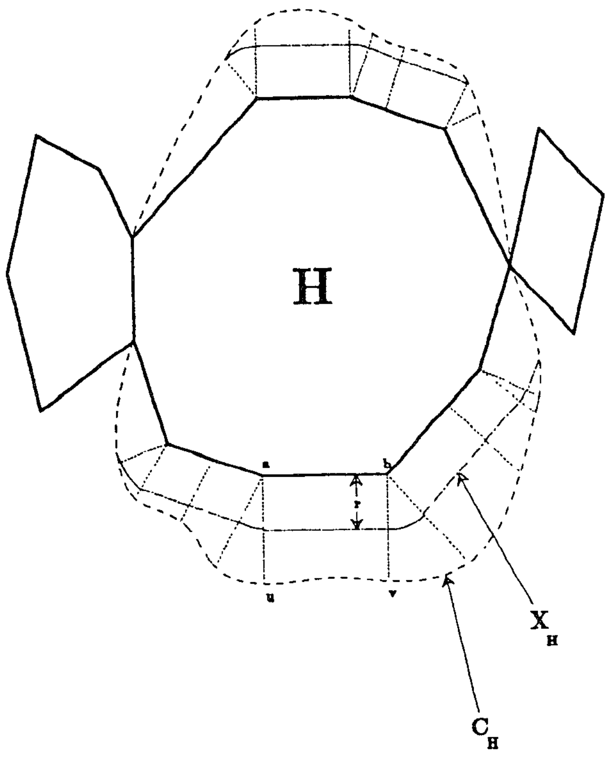

FIG. 17. Padding a fundamental polygon.

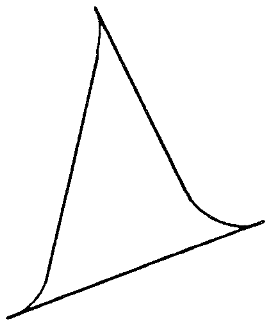

FIG. 18. A rounded triangle.

(4) Associated with each rounded triangle T of O^+ are two *exit vertices* such that for every point x that is contained in (or lies on the border of) T, the shortest path from x to the source s can be obtained by computing in $O(1)$ time a shortest path from x to the closest exit vertex, and then following $b(\,)$ pointers back to s. That is, the length of a shortest path between s and x can be computed in $O(1)$ time and the path itself can be constructed in time proportional to the number of edges it contains.

The *unbounded* single-source shortest path problem for a disc is like the regular single-source shortest path problem for a disc except that the obstacle space taken does not have an enclosing wall and the data structure is augmented with a set of simple non-intersecting quadratic curves that partition the

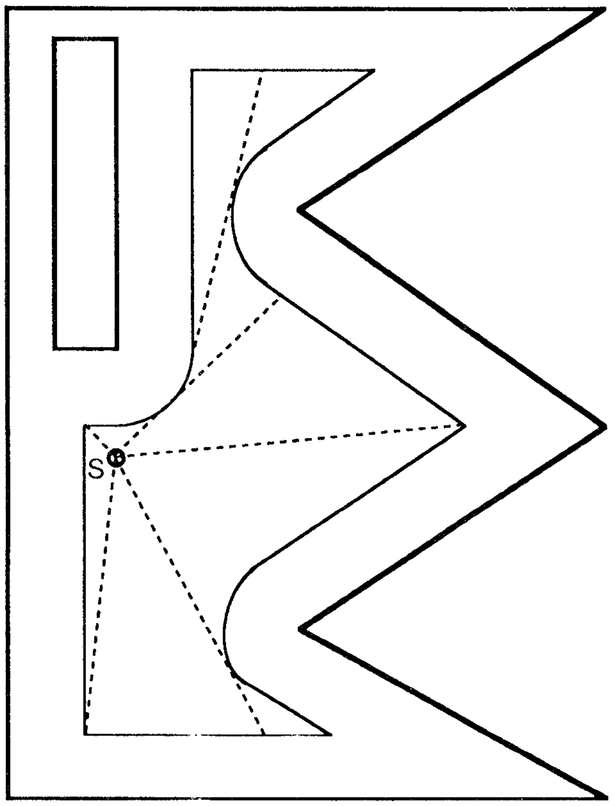

FIG. 19. A rounded triangulation for a disc.

infinite region of the plane; point location for these infinite regions (to find between which pair of curves a point lies) can be done in $O(\log n)$ time and the minimal legal distance from any point in the infinite region can be computed in $O(1)$ time.

THEOREM 6.1. *Assuming that the Voronoi diagram is provided with the input, the single-source shortest path problem for a disc can be solved in $O(kn)$ time and $O(n)$ space.*

PROOF. From Lemma 6.2, it follows that movement of disc in an obstacle space can be transformed to the problem of moving a point in a rounded obstacle space; hence, it suffices to consider the single-source shortest path problem for a rounded obstacle space. The only difference between this problem and the standard one is that two edges that used to come together at an obtuse vertex now come together at an arc segment. It is straightforward to generalize the crosscut and extend operations to place edges tangent to these arcs. As an example, Figure 19 shows an obstacle space (thick lines), the padding performed by Lemma 6.2 (thin lines), and the partitioning into rounded triangles from a source s (dashed lines).

One remaining detail is point location; since we have not produced a true triangulation of the space, we can't directly invoke a standard point location algorithm. However, we can add additional edges to the obstacle space so that all arcs are enclosed by a triangle. Given this, the space will be completely triangulated, but each triangle may contain one arc segment and one $O(1)$-degree curve segment (from the flower construction). Since both the arc and the curve segment are convex, this partitions the triangle into at most five regions and hence once it is determined in which triangle a point lies (in $O(\log n)$ time), in $O(1)$ additional time the proper region within the triangle can be determined.

The proof of the above theorem can easily be applied to slightly more general obstacle spaces that are described by the following definition:

Definition 6.4. A *rounded obstacle space* is like a regular obstacle space except that obstacle edges are allowed to be arc segments or line-segments; however, if an edge if an arc segment, it must bend towards the interior of the obstacle space.[12]

COROLLARY 6.1a. *The above theorem applies even for rounded obstacle spaces.*

PROOF. Movement of a point follows directly from the proof of the above theorem. For movement of a disc, observe that even starting with a rounded obstacle space, the padding performed by (the straightforward generalization of) Lemma 6.2 is still a rounded obstacle space.

REFERENCES

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1983. *Data Structures and Algorithms.* Addison-Wesley, Reading, Mass.

ASANO, T., GUIBAS, L., HERSHBERGER, J., AND IMAI, H. 1986. Visibility of disjoint polygons. *Algorithmica 1*, 49–63.

CHAZELLE, B. 1982. A theorem on polygon cutting with applications. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science* (Chicago, Ill.). IEEE, New York, pp. 339–349.

CHAZELLE, B. 1990. Triangulation of a simple polygon in linear time. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science.* IEEE, New York, pp. 220–230.

CHEW, L. P. 1985. Planning the shortest path for a disc in $O(n^2 \log n)$ time. In *Proceedings of the 1st Annual ACM Symposium on Computational Geometry* (Baltimore, Md., June 5–7). ACM, New York, pp. 214–220.

CLARKSON, K. 1987. Approximation algorithms for shortest path motion planning. In *Proceedings of the 19th Annual ACM Symposium on the Theory of Computing* (New York, N.Y., May 25–27). ACM, New York, pp. 56–65.

CLARKSON, K., KAPOOR, S., AND VAIDYA, P. M. 1987. Rectilinear shortest paths through polygonal obstacles in $O(n \log^2 n)$ time. In *Proceedings of the 3rd Annual ACM Symposium on Computational Geometry* (Waterloo, Ontario, Canada, June 8–10). ACM, New York, pp. 251–257.

DE REZENDE, P. J., LEE, D. T., AND WU, Y. F. 1985. Rectilinear shortest paths rectangular barriers. In *Proceedings of the 1st ACM Symposium on Computational Geometry* (Baltimore, Md.). ACM, New York, pp 204–212.

DRYSDALE, R. L. 1979. Generalized Voronoi diagrams and geometric searching. Tech. Rep. STAN-CS-79-705. Computer Sci. Dept, Stanford Univ., Stanford, Calif.

EDELSBRUNNER, H. 1987. *Algorithms in Combinatorial Geometry.* Springer-Verlag, New York.

FREDERICKSON, G. N 1984. Fast algorithms for shortest paths in planar graphs, with applications. Tech. Rep. CSD TR 486. Dept Comput. Sci. Purdue Univ., Lafayette, Ind.

FREDMAN, M. L., AND TARJAN, R. E. 1984. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proceedings of the 25th Annual IEEE Symposium on the Foundations of Computer Science* (Singer Island, Fla.). IEEE, New York, pp. 338–346.

[12] That is, a rounded obstacle space can be thought of as a regular obstacle space where some of the obtuse vertices have been replaced by arc segments.

GUIBAS, L., HERSHBERGER, J., LEVEN, D., SHARIR, M., AND TARJAN, R. 1986. Linear time algorithms for visibility and shortest path problems inside simple polygons. In *Proceedings of the 2nd Annual ACM Conference on Computational Geometry* (Yorktown Heights, N.Y., June 2–4). ACM, New York, pp. 1–13.

HERSHBERGER, J., AND GUIBAS, L. J. 1988. An $O(n^2)$ shortest path algorithm for a non-rotating convex body. *J. Algorithms 9,* 18–46.

HOPCROFT, J. E., JOSEPH, D. A., AND WHITESIDES, S. H. 1982. Movement problems for 2-dimensional linkages. Tech. Rep. TR82-515. Computer Sci. Dept., Cornell Univ., Ithaca, N.Y.

JOSEPH, D. A., AND PLANTINGA, W. H. 1985. On the complexity of reachability and motion planning questions. In *Proceedings of the 1st Annual ACM Conference on Computational Geometry* (Baltimore, Md., June 5–7). ACM, New York, pp. 62–66.

KIRKPATRICK, D. G. 1979. Efficient computation of continuous skeletons. In *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science.* IEEE, New York, pp. 18–27.

KIRKPATRICK, D. G. 1983. Optimal search in planar subdivisions. *SIAM J. Comput. 12,* 1, 28–35.

LARSON, R. C., AND LI, V. O. K. 1981. Finding minimum rectilinear distance paths in the presence of barriers. *Networks 11,* 285–304.

LEE, D. T., AND PREPARATA, F. P. 1984. Euclidean shortest paths in the presence of rectilinear barriers. *Networks 14,* 393–410.

LIPTON, R. J., AND TARJAN, R. E. 1977. Applications of a planar separator theorem. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science.* IEEE, New York, pp. 162–170.

LOZANO-PEREZ, T. 1980. Automatic planning of manipulation transfer movements. Memo 606. Artificial Intelligence Laboratory, MIT, Cambridge, Mass.

LOZANO-PEREZ, T., AND WESLEY, M. A. 1979. An algorithm for planning oscillation-free paths among polyhedral obstacles. *Commun. ACM 22,* 10 (Oct.), 560–570.

MITCHELL, J. S. B. 1987. Shortest rectilinear paths among obstacles. Tech. Rep. 739. School of Operations Research and Industrial Engineering. Cornell Univ., Ithaca, N.Y.

MITCHELL, J. S. B., AND PAPADIMITRIOU, C. H. 1985. Planning shortest paths. *SIAM Conference on Geometric Modeling and Robotics* (Albany, N.Y.). SIAM, New York, pp. 1–21.

MITCHELL, J. S. B., AND PAPADIMITRIOU, C. H. 1991. The weighted region problem: Finding shortest paths through a weighted planar subdivision. *J. ACM 18,* 1 (Jan.), 18–73.

O'DÚNLAING, C., SHARIR, M., AND YAP, C. K. 1983. Retraction: A new approach to motion-planning. In *Proceedings of the 15th ACM Symposium on Theory of Computing* (Boston, Mass., Apr. 25–27). ACM, New York, pp. 207–220.

PREPARATA, F. P., AND SHAMOS, M. I. 1988. *Computational Geometry: An Introduction.* Springer-Verlag, New York.

REIF, J. H. 1979. Complexity of the mover's problem. In *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science* (San Juan, P.R.). IEEE, New York, pp. 421–427.

REIF, J. H., AND STORER, J. A. 1985. Shortest paths in Euclidean space with polyhedral obstacles. Tech. Rep. CS-85-121. Computer Sci. Dept., Brandeis Univ., Waltham, Mass.

SCHWARTZ, J. T., HOPCROFT, J. E., AND SHARIR, M. 1981. On the piano movers problem. 1: The case of a 2-dimensional rigid polygonal body moving amidst barriers. Tech. Rep. TR39. Computer. Sci. Dept., New York Univ., New York, N.Y.

SCHWARTZ, J. T., HOPCROFT, J. E., AND SHARIR, M. (EDS.). 1987. *Planning Geometry, and Complexity of Robot Motion.* Ablex Publishers, Norwood, N.J.

SCHWARTZ, J. T., AND SHARIR, M. 1981. On the piano movers problem. 1: The case of 2-dimensional rigid polygonal body moving amidst barriers. Tech. Rep. TR 39. Comput. Sci. Dept., New York Univ., New York.

SCHWARTZ, J. T., AND SHARIR, M. 1982. On the piano movers problem. 2: General techniques for computing topological properties of real algebraic manifolds. Tech. Rep. TR41. Comput. Sci. Dept., New York Univ., New York, N.Y.

SEDGEWICK, R., AND VITTER, J. S. 1984. Shortest paths in Euclidean graphs. In *Proceedings of the 25th Annual IEEE Symposium on the Theory of Computing* (Singer Island, Fla.). IEEE, New York, pp. 417–424.

SHAMOS, M. I. 1975. Geometric complexity. In *Proceedings of the 7th ACM Symposium on the Theory of Computing* (Albuquerque, N.M., May 5–7). ACM, New York, pp. 224–233.

SHAMOS, M. I. 1978. Computational geometry. Ph.D. dissertation. Computer Sci. Dept., Yale Univ., New Haven, Conn.

SHARIR, M., AND SCHORR, A. 1984. On shortest paths in polyhedral spaces. In *Proceedings of the 16th ACM Symposium on the Theory of Computing* (Washington, D.C., Apr. 30–May 2). ACM, New York, pp. 144–153.

VACCARO, H. 1974. Alternative techniques for modeling distance. Masters Thesis in Civil Engineering. MIT, Cambridge, Mass.

WANGDAHL, G. E., POLLACK, S. M, AND WOODWARD, J B. 1974. Minimum-trajectory pipe routing. *J. Ship Res. 18,* 46–49.

WELZL, E. 1985. Constructing the visibility graph for n-line segments in $O(n^2)$ time. *Inf Proc. Lett. 20,* 167–171.

WU, Y.-F., WIDMAYAER, P., SCHLAG, M. D. F., AND WONG, C. K. 1987. Rectilinear shortest paths and minimum spanning trees in the presence of rectilinear obstacles *IEEE Trans Comput. C-36,* 3 (Mar.), 321–331.

YAP, C K. 1984. An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments. Tech. Rep. Courant Inst. of Math. Sci., New York, New York, N Y.