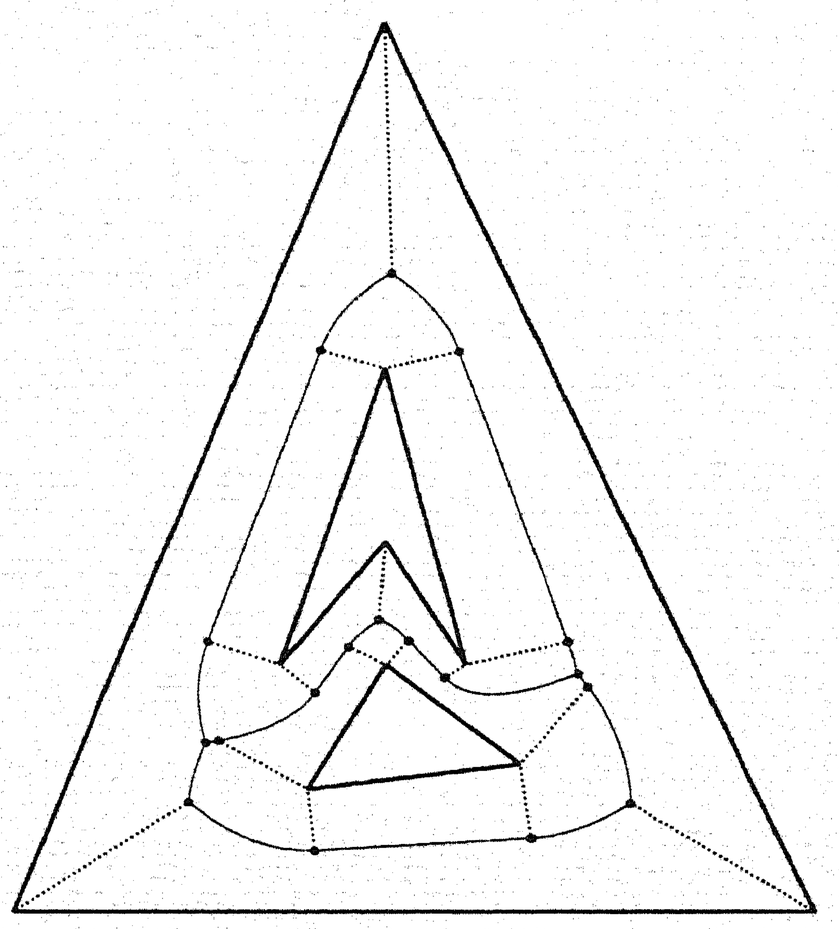
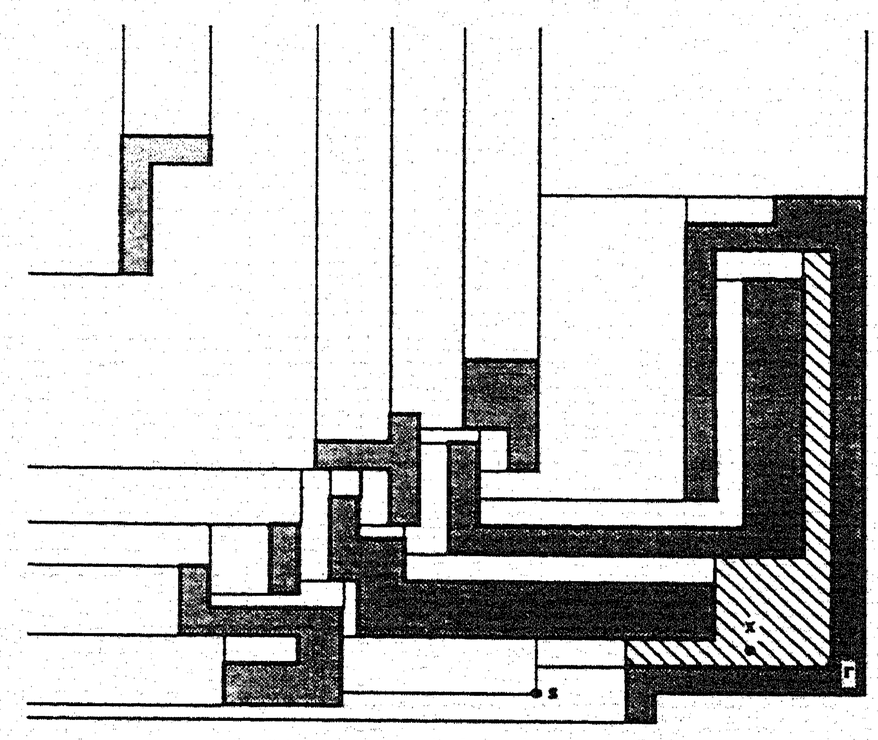
Four Papers on Computational Geometry





Contributors

• Varol Akman (Netherlands) • Chanderjit Bajaj (USA) • Randolph Franklin (USA) • Joe Mitchell (USA) • Hans Rohnert (West Germany) • Gordon Wilfong (USA)

Prepared for

Computational and Geometric Aspects of Robotics
1st Int. Conf. on Industrial and Applied Math. (ICIAM'87)
Cité des Sciences et de l'Industrie
Paris (June 29 - July 3, 1987)

Published by
Centrum voor Wiskunde en Informatica (CWI)
Kruislaan 413, 1098 SJ Amsterdam
The Netherlands

Four Papers on Computational Geometry

Varol Akman

Centrum for Wiskunde en Informatica
Amsterdam, the Netherlands

Chanderjit Bajaj

Purdue University
West Lafayette, Indiana, USA

Randolph Franklin

Rensselaer Polytechnic Institute
Troy, New York, USA

Joe Mitchell

Cornell University
Ithaca, New York, USA

Hans Rohnert

Universität Saarbrücken
Saarbrücken, West Germany

Gordon Wilfong

AT & T Bell Laboratories
Murray Hill, New Jersey, USA

Abstract Robotics is essential for the reindustrialization of the world and assorted computer science ideas did and will contribute to the flourishing of robotics as a computational discipline. This collection of papers treat several motion planning problems central to robotics from the viewpoint of computational geometry and complexity. Most of the papers will be presented in "Computational and Geometric Aspects of Robotics," a minisymposium to be held during the *First International Conference on Industrial and Applied Mathematics (ICIAM'87)*, Paris, France (June 29 - July 3, 1987).

Keywords Computational geometry, robotics, computational complexity, algorithmic motion planning, Piano Mover's problem, Findpath problem, Voronoi diagram, Configuration space, wavefront propagation, Dijkstra's algorithm, acceleration bounded motion, velocity constrained motion, paths with bounded curvature, algebraic object models, safe paths, maximum spanning tree, planar point location, geometer's workbench, shortest rectilinear paths, moving discs.

CR Categories (1987) D.2.6, E.1, F.2.2, G.2.2, I.2.9, I.3.5.

Note Papers in this collection may be published by their authors elsewhere.

Foreword

This report includes four papers on computational geometry — specifically its use in robotics problems. Excluding the first paper[†], the papers will be presented at the ICIAM'87 Minisymposium MS/58, *Computational and Geometric Aspects of Robotics*, organized by myself. ICIAM'87 (First International Conference on Industrial and Applied Mathematics) is being organized by INRIA and is sponsored by GAMM, IMA, SIAM, and SMAI (see disclaimer below). It will take place at the Cité des Sciences et de l'Industrie Paris, France, June 29 - July 3, 1987.

I gratefully thank the authors of the papers, especially Joe Mitchell, Hans Rohnert, and Gordon Wilfong for their enthusiastic cooperation and excellence. I, for one, am looking forward to hearing their exciting talks based on these articles. I state with regret that, due to potential copyright problems, one participant of MS/58, Chanderjit Bajaj, could not contribute his full paper to this collection. His abstract appears as the last page of this report.

I thank the director of my institute, P.C. Baayen, and my supervisor, Paul ten Hagen, for their kind permission to make this report possible. Help of the CWI Publications Department is also sincerely acknowledged.

Varol Akman
CWI
Amsterdam, May 1987

Disclaimer

[†] A talk based on the first paper was presented at the *SIAM Conference on Solid Modeling and Robotics*, July 1985, Albany, New York.

Contents

A Workbench to Compute Unobstructed Shortest Paths in Three-Space

Randolph Franklin
Rensselaer Polytechnic Institute
Troy, New York, USA

Varol Akman
Centrum for Wiskunde en Informatica
Amsterdam, the Netherlands

Abstract

Recently, the following problem has gained considerable importance in computational geometry:

> FINDPATH: Given a set of obstacles and two points (source and goal), calculate the shortest path between these points under the Euclidean metric, constrained to avoid intersections with the given obstacles.

We present three implementations to solve specific instances of FINDPATH in three-space where the obstacles become polyhedra. In the first two cases there exists a single convex polyhedron and the source and the goal points are on its boundary or exterior. These solutions make use of planar developments of polyhedra and polyhedral visibility. The last case is based on a locus method. It partitions the boundary of a convex polyhedron given only the source on it so that for a later goal on the boundary, the shortest path is found efficiently. It makes use of standard point location algorithms for a straight-edge planar subdivision once the partitioning is done.

0. Introduction and Recent Algorithms in Three-Space

Let $P=\{P_1, \cdots, P_n\}$ be a prescribed set of disjoint polyhedra and $s, g \in R^3$ be distinct points which are *not* internal to any P_i. The class of rectifiable curves which have endpoints s and g and which do not intersect any intP_i will be denoted by $C(s, g : P)$. For C in this class, $l(C)$ will denote the

length of C under the Euclidean (L_2) metric. An interesting problem in computational geometry asks for the shortest one among these curves:

FINDPATH
INSTANCE: Polyhedra $P = \{P_1, \cdots, P_n\}$ such that $P_i \bigcap P_j = \Phi$, $i \neq j$, and $s, g \epsilon R^3$ such that $s \neq g$ and s, g do not belong to $\text{int}P_i$, $1 \leqslant i \leqslant n$.
QUESTION: Which $C \epsilon C(s, g : P)$ has the shortest length?

The following theorem must be intuitively clear:

THEOREM 0.1 There exists a $C^* \epsilon C(s, g : P)$ such that for all $C \epsilon C(s, g : P)$, $l(C^*) \leqslant l(C)$. Moreover, every such C^* is a polygonal path with its possible bend points belonging to some edges of some members of P.
Proof Omitted. □

It is noted that the polygonal path C^* found in theorem 0.1 is not necessarily unique. (In fact, there may be exponentially many shortest paths in terms of n.) We shall call any such path an s-to-g *shortest path*. Thus, FINDPATH asks for the characterization (i.e., the determination of the bend points) of an s-to-g shortest path.

There is a wealth of material in the general area of *motion planning* which comprises other familiar robotics problems such as FINDSPACE, MAKESPACE, etc. in addition to FINDPATH. For brevity, we shall refer the reader to a recent work by Akman[1] which deals with FINDPATH in greater detail and lists numerous references. Franklin and Akman also inspect the same problem from various perspectives in [8, 9, 10, 11].

Sharir and Schorr's work[32] is another detailed study on shortest paths. They mainly consider the case of BOUNDARY FINDPATH (locus) (cf. section 3) and present an algorithm which works in time $O(n)$ per query where n is the measure of complexity of the polyhedron, say the number of vertices. Their algorithm is based on the idea of "ridge" points on the object. A ridge point on the polyhedron has the property that there exists at least two shortest paths to it from the source. It turns out that the set of ridge points is a union of line segments and that the union of the vertices and the ridge points is a closed connected set. Defining the union of the latter with the shortest paths from the source to every vertex, one partitions the polyhedron boundary into disjoint connected regions (called "peels") whose interiors are free of vertices or ridge points. The peels can be constructed in $O(n^3 \log n)$ (preprocessing) time using complicated techniques whose implementation seems extremely difficult. The size of the data structure that is created by the preprocessing step is $O(n^2)$.

O'Rourke et al[24] use Sharir and Schorr's ideas to extend the problem to a polyhedral surface which is no more supposed to be convex. (However, it should be orientable.) The shortest path that they calculate is only a "geodesic", i.e., it is confined to the surface and thus may not be the true shortest path. Their algorithm runs in $O(n^5)$ time. Furthermore, it does not follow a locus approach; using their algorithm on each new query (goal point) would take $O(n^4)$ time.

Mitchell and Papadimitriou[23] give an algorithm to solve this last problem in a locus setting. (It is noted that they are still computing geodesics, not true shortest paths.) Theirs is an $O(n^2 \log n)$ time algorithm for subdividing the surface of an arbitrary polyhedron (possibly of positive genus) so that the length of the shortest path from a given source to any goal on the surface is obtainable by simple point location. It has striking similarities to Dijkstra's method for shortest paths in graphs. As in our algorithm to be presented in section 3, point location is achieved in time $O(\log n)$, after which the actual shortest path is backtracked in time proportional to the number of faces that it traverses on the boundary.

1. Shortest Paths on a Convex Polyhedron

Consider the following specific instance of FINDPATH:

> BOUNDARY FINDPATH
> INSTANCE: Convex polyhedron P, specified points $s, g \in \mathrm{bd}P$ where $s \neq g$.
> QUESTION: Which $C \in \mathrm{bd}P$ has the shortest length?

Before we solve this problem we give an argument as to its importance. Let $P = \{P_1, \cdots, P_n\}$ be a set of *convex* polyhedra. If we are allowed to compute a reasonably short (but *not* the shortest) s-to-g path then we can pursue the following strategy. Let P' be the subset of P such that every member of P' is intersected by sg. (If a polyhedron is intersected by sg only once then it is not included in P'; thus P' consists of polyhedra intersected by sg at precisely two points.) Applying BOUNDARY FINDPATH to each member of P' we obtain an s-to-g polygonal path which comprises two types of curves: line segments through free space, and polygonal paths along the boundaries of the polyhedra between where the path "lands" from free space and "takes off" again (figure 1). Repeated optimization of this path is possible and will frequently yield a better (with fewer bend points) and shorter path (figure 2) although it is not difficult to come up with cases where repeated optimization might cause clashes.

We shall assume that the boundary representation is used to define a

polyhedron P. In this representation scheme each vertex is defined by its x, y, z coordinates and each face is given as a list of pointers to the vertices, ordered in counterclockwise around the boundary of the face with respect to a point above it. It is convenient to think of vertex labels or face labels as positive integers.

DEFINITION The *face graph (Fgraph)* of a convex polyhedron P is an undirected graph $Fgraph = (FV, FE)$ with unit arc weight, $FV = \{i : F_i$ is a face of $P\}$ and $FE = \{(i, j) : F_i$ and F_j are adjacent$\}$. (Two faces are *adjacent* if they share an edge.) □

EXAMPLE Figure 3(a) shows the face graph of a cube. In figure 3(b), the face graph of a parallelepiped is given to note that *Fgraph* only preserves the adjacency information. Figure 3(c) shows that two faces with a common point only are *not* considered adjacent by this definition. □

DEFINITION Let $C \in \mathrm{bd}P$ be an s-to-g polygonal path. The sequence of faces that C enters defines the *face visit sequence* of C which will be denoted by $\mathrm{fvs}C$. □

Thus $\mathrm{fvs}C$ is a walk in *Fgraph* between the nodes corresponding to faces F_s and F_g, the faces of P containing s and g, respectively. An immediate consequence of the above definition is:

LEMMA 1.1 Let $C^* \in \mathrm{bd}P$ be an s-to-g shortest path. Then $\mathrm{fvs}C^*$ is a simple walk in *Fgraph*.
Proof. If this is not true then C^* enters a face at least twice. Recalling the fact that the faces of P are all convex, we can then further shorten C^*, a contradiction. □

From now on, all face visit sequences will therefore be assumed to be simple.

In addition to *Fgraph*, a useful construct that will be used by BOUNDARY FINDPATH is the planar development of a given face visit sequence. It is well-known that the boundary of a convex polyhedron has the structure of a planar graph. Therefore, the totality of the faces of P, situated in three-space in certain mutual relationship, can be represented in two-space (specifically the xy-plane) by a system of polygons identified with the faces of P. The relationship between a planar development and the planar polygonal polygonal schema will be apparent after the following description of how to obtain the latter.

In the xy-plane associate with each face of P a polygon having the same metrical form. (Two polygons have the same *metrical form* if they can be

made to coincide by translations and rotations.) Define the *glue* relationship between the pairs of edges of these polygons such that two glued edges come from the *same* edge of P. Figure 4 illustrates this for a cube. Each edge in the planar polygonal schema is glued to exactly one edge.

DEFINITION A *planar development* corresponding to a face visit sequence $1, \cdots, k$ is a union of polygons $F_1, \cdots, F_k$ of the planar polygonal schema of P. In the planar development two polygons F_i and F_{i+1}, $1 \leqslant i < k$ are united along the edge that they are glued to each other, and do not overlap. □

DEFINITION The *image* of a point on a polyhedron under a planar development is the point in the plane that it ends up under the development. □

DEFINITION A planar development is *legal* if the line segment connecting the images of the source and the goal is internal to the development. □

EXAMPLE Figure 5 shows several planar developments computed and drawn by SP, our shortest path workbench (cf. appendix). The objects are as follows. Figure 5(a): cube, figure 5(b): icosahedron, figure 5(c) and (d): dodecahedron. It is noted that the last development is not legal. □

It should be apparent that once a planar development is built, it can be moved to any position and orientation in the plane without changing the intrinsic geometry of the paths. We shall now give a procedural definition of a planar development:

DEFINITION To compute the *planar development* of a face visit sequence $1, \cdots, k$, start with F_1. If affF_1 is parallel to the xy-plane then translate P by a suitable amount so that F_1 is now in the xy-plane; otherwise, let the dihedral angle between affF_1 and the xy-plane be D and and rotate P about the line aff$F_1 \bigcap xy$-plane by D to map F_1 to the xy-plane. The remaining faces F_i are inductively handled as follows. Let e be the common edge of F_{i-1} and F_i whose dihedral angle is D. Rotate P by D about aff$e$ to place F_i to the xy-plane while avoiding overlaps with F_{i-1}'s polygon which is already there. □

Now we are ready for:

Algorithm BOUNDARY FINDPATH
1. Let F_s and F_g be the faces of P including s and g, respectively. Assume that $F_s \neq F_g$; otherwise the shortest path is $C^* = sg$.
2. Let FVS be the set of all simple walks in $Fgraph$ of P, between the nodes corresponding to F_s and F_g. Initialize $vs^* = \Phi$ and $l^* = +\infty$.

3. For each member of FVS do the following steps:

3.1 Compute the planar development corresponding to this face visit sequence. Let s' and g' be the images of s and g in the xy-plane under the same development. (They can be computed along with the planar development.)

3.2 If the development in step 3.1 is not legal then continue with step 3. Otherwise, if $d(s',g') < l^*$ then replace vs^* with the current face visit sequence, let $l^* = d(s',g')$, and continue with step 3.

4. At this point l^* is the length of the shortest path and vs^* is the face visit sequence that should be used to to compute the shortest path itself. To do this, first compute the planar development of vs^* (and s' and g') and intersect $s'g'$ with all pairwise common edges of the polygons in the development. The intersection points in the plane are then easily used to compute the bend points of the shortest path C^*. We know from the planar development of vs^* the distance of an intersection point from a vertex in the plane. All we need is to identify the vertex of P in three dimensions that led to this vertex. Then marking the point which is away from this vertex the same distance over the edge touched by the shortest path we locate the bend point for one intersection. The others are found completely analogously.

End

An efficient way of checking whether a planar development is legal follows. Let $e_1, \cdots, e_t$ be the sequence of edges that are glued in the development. Then the development is legal if $s'g'$ intersects every e_i. Note that this is always easier than testing if $s'g'$ is internal to the planar development's boundary.

To list the simple walks between two nodes of a graph we can use the algorithm of Yen[36] which works in $O(kn^3)$ if there are n nodes in the graph and we require the first k simple walks in increasing walk length. Katoh et al[15] give an improved algorithm for the same task with a time complexity $O(kf(n,m))$ under the assumption that shortest walks form one node to all others can be found in $f(n,m)$ time where m is the number of arcs in the graph. Since $f(n,m)$ is either $O(n^2)$ or $O(m \log n)$ in the worst case, this algorithm is more efficient than Yen's.

Determining the value of cardFVS is difficult. Garey and Johnson[12] state that the following problem is NP-hard:

K-th SHORTEST PATH
INSTANCE: Graph $G=(V,E)$, positive integer lengths l_e for each $e \in E$, specified nodes $s,t \in V$, positive integers b and k.
QUESTION: Are there k or more distinct simple walks from s to

t in G, each having total length b or less?

They also mention that K-th SHORTEST PATH remains NP-hard even if $l_e = 1$ for all $e \epsilon E$, and is solvable in pseudo-polynomial time (polynomial in card V, k, and $\log b$) and accordingly, in polynomial time for any fixed k (e.g., Yen's algorithm). The difficulty of K-th SHORTEST PATH basically resides in the following counting problem which was proven to be #P-complete by Valiant[34]:

> S-T PATHS (SELF-AVOIDING WALKS)
> INSTANCE: Graph $G = (V, E)$, specified nodes $s, t \epsilon V$.
> QUESTION: What is the number of walks from s to t that visit every node *at most* once?

The problem of counting (s, t)-walks in (s, t)-planar graphs is also #P complete[28]. (A graph is called *source-sink planar*, or (s, t)-*planar* in short, if it has a planar representation with nodes s and t on the boundary.) Provan[29] states that the approximation problem for (s, t)-walks is unsolved, in the sense that the following problem is open:

> "For *any* fixed $\epsilon < 1$, does there exist a polynomial algorithm which for a given (s, t)-planar graph G, will give an approximation N_0 for the number N of (s, t)-walks in G which satisfies $|N - N_0| < \epsilon N$?"

On the other hand, for *any* fixed $\epsilon > 0$, can it be proven that the above problem is NP-hard? Currently, the only known approximations are to count the minimum length walks or to enumerate as many walks as possible (using a large value of k in Yen's algorithm, for example).

It is not hard to find a convex polyhedron which has an exponential number of simple walks in its *Fgraph* (figure 6). If there are l lateral faces of this pyramid-like object (not counting the small triangular faces) then the number of simple walks between the nodes F_s and F_g in the figure is $\Omega(2^{l/2})$. This result also shows that our BOUNDARY FINDPATH algorithm is of exponential time complexity in the number of faces of P. As mentioned before, there exist polynomial algorithms by other researchers for this problem. Unfortunately, theirs do not seem to admit practical implementations. In the light of this, the algorithm presented in this section is applicable for objects of moderate complexity. We can also try to test only a certain section (e.g., first few in increasing walk length) of the face visit sequences between the source and the goal faces for an object with many faces with the hope that the shortest path is generated by a short face development sequence. The shortest path rendered by the legal developments found among the developments that

these sequences give may be taken as the true shortest path although this is certainly vulnerable to an adversary.

2. Shortest Paths around a Convex Polyhedron

Now we inspect the following variant of BOUNDARY FINDPATH:

> EXTERIOR FINDPATH
> INSTANCE: Convex polyhedron P and points s ,g where at least one of them is external to P , $s \neq g$.
> QUESTION: Which $C \epsilon C(s, g; P)$ has the shortest length?

Without loss of generality, we shall treat the case where s and g are both outside P . In this case, the following fact is useful:

LEMMA 2.1 Let $H = \mathrm{conv}(\{s, g\} \bigcup \mathrm{vert} P)$. Then an s-to-g shortest path for EXTERIOR FINDPATH is entirely on $\mathrm{bd}H$.
Proof Omitted. □

Thus, once H is computed using standard three-space convex hull algorithms, we can apply BOUNDARY FINDPATH to the instance made of H , s , and g . (Preparata and Hong[26] give an algorithm to find the three-dimensional convex hull in $O(n \log n)$ time for n points.) However, there is a slight difficulty with this approach. Assuming that we want to know which edges of the original polyhedron the shortest path touches, we must keep extra information with H , i.e., which vertex of H comes from which vertex of P . Below, we shall give a more direct method to obtain H while keeping this information implicitly using a visibility-based approach. (Sutherland et al[33] give an overview of polyhedral visibility.)

DEFINITION For a convex polyhedron P and a point x external to it, the *silhouette edges* of P are members of

$$\{e : e \epsilon F_1 \text{ and } e \epsilon F_2 \text{ where } F_1 \epsilon F_{vis} \text{ and } F_2 \epsilon F_{invis}\}$$

Here, F_{vis} (resp. F_{invis}) is the set of visible (resp. invisible) faces of P from viewpoint x . □

Clearly, $F_{vis} \bigcap F_{invis} = \Phi$ since a face of a convex polyhedron is either completely visible or completely hidden to an observer. Let $E_{sil,s}$ (resp. $E_{sil,g}$) denote the silhouette edges of P from s (resp. g). It is clear that the faces of H will be the union of three *disjoint* sets:

$$\mathrm{bd}H = F_{tri\,,s} \bigcup F_{tri\,,g} \bigcup (F_{invis\,,s} \bigcap F_{invis\,,g})$$

Here $F_{tri\,,s}$ (resp. $F_{tri\,,g}$) is a set of triangular faces each characterized by an edge of $E_{sil\,,s}$ (resp. $E_{sil\,,g}$) and s (resp. g). In essence, these are the lateral faces of a pyramid-like object with (generally nonplanar) basis $E_{sil\,,s}$ (resp. $E_{sil\,,g}$) and apex s (resp. g). It is noted that the geometric complexity of object H is the same as with P.

We conclude this section with an algorithm to compute the silhouette edges of P from a point x external to it:

Algorithm SILHOUETTE
1. Compute $F_{vis\,,x}$, the visible faces of P from viewpoint x, by checking line segments xc_i where c_i is the center of mass of face F_i against all F_j, $j \neq i$ for intersection. $F_{vis\,,x}$ consists of all F_i which do not cause any intersection.
2. Let the totality of the visible edges of P from x be $E_{vis\,,x} = \{e : e \in F$ where $F \in F_{vis\,,x}\}$. Sort $E_{vis\,,x}$ and eliminate *both* of duplicate members. The remaining edges are precisely the edges of $E_{sil\,,x}$.
End

Figure 7 demonstrates the working of EXTERIOR FINDPATH on a simple object.

3. Partitioning the Boundary of a Convex Polyhedron

Finally, consider the following variant of BOUNDARY FINDPATH:

> BOUNDARY FINDPATH (locus)
> INSTANCE: Same as in BOUNDARY FINDPATH except that g is not given. However, it is guaranteed that, when specified, g will be on $\mathrm{bd}P$.
> QUESTION: Preprocess P so that for any specified $g \in \mathrm{bd}P$ the s-to-g shortest path is computed efficiently.

A practical case which would benefit from BOUNDARY FINDPATH (locus) is as follows. Consider a truck on the surface of a mountain modeled as a convex polyhedron, say, a pyramid. Suppose that the truck is required to carry material from a fixed location on the surface to several points, say, construction sites. Then, it is reasonable to compute the shortest routes for the truck (approximated as a point) more efficiently than can be achieved by repeated applications of BOUNDARY FINDPATH for each specified destination.

This is a powerful paradigm of computational geometry known as the *locus*

approach and is studied in Overmars[25]. In solving a problem using this approach, we are allowed to spend some initial effort (i.e., preprocessing) to construct a data structure which will let us answer future requests (i.e., queries) quickly. To be effective, this assumes two things. First, the number of the query points must be large to validate such an initial effort. Second, the data structure must embody succinctly the locus of the required solution and must enjoy the existence of a fast search procedure to retrieve it.

We shall now summarize one such data structure suitable for solving BOUNDARY FINDPATH (locus). The data structure is known as the *Voronoi diagram* and was first introduced to computational geometry by Shamos[31]. Let $S=\{x_1, \cdots, x_n\}$ be a subset of R^2. For $1 \leqslant i \leqslant n$ let

$$\mathrm{regn} x_i = \{y : \mathrm{d}(x_i, y) \leqslant \mathrm{d}(x_j, y) \text{ for all } j\}$$

be the *Voronoi region* of point x_i. The Voronoi diagram of S, denoted by $\mathrm{vor}S$, partitions the plane into $\mathrm{card}S = n$ regions, one for each member of S. The (open) Voronoi region of point x_i consists of all points of R^2 closer to x_i than any other point of S. For $1 \leqslant i, j \leqslant n$, letting $H_{ij} = \{y : \mathrm{d}(x_i, y) \leqslant \mathrm{d}(x_j, y)\}$ (the halfspace defined by the perpendicular bisector of $x_i x_j$), it is seen that for $i \neq j$, $\mathrm{regn} x_i = \bigcap H_{ij}$. Thus, $\mathrm{regn} x_i$ is a convex polygonal region and $\mathrm{vor}S$ is equal to the union of the boundaries of all $\mathrm{regn} x_i$. For every vertex x of $\mathrm{vor}S$ there are at least three points x_i, x_j, x_k in S such that $\mathrm{d}(x, x_i) = \mathrm{d}(x, x_j) = \mathrm{d}(x, x_k)$. The Voronoi diagram for a set of n points has at most $2(n-2)$ vertices and $3(n-2)$ edges[22].

The Voronoi diagram of S can be computed in $O(n \log n)$ time [30] and this is optimal with respect to a wide range of computational models[17]. Unfortunately, practical Voronoi programs which are both fast and reliable are difficult to write mainly due to the special cases in the diagram that are to be handled precisely. Among the published algorithms, those by Avis and Bhattacharya[2], Lee[19], and Guibas and Stolfi[14] seem to be more promising for practical use. On the other hand, it is possible to construct slower implementations which handle special cases without much effort.

We now summarize how to search Voronoi diagrams in logarithmic time in the number of the edges, n, of the diagram, i.e., we cite methods which let one find the point $x \in S$ such that for a query point $y \in R^2$, $\mathrm{d}(x, y)$ is minimum. The search methods we shall review are more general than searching Voronoi diagrams in that they are based on searching a *planar subdivision*, i.e., a straight-edge embedding of a planar graph. The underlying problem is generally known as *planar point location* in computational geometry. Let us call a subset of the plane *monotone* if its intersection with any line

parallel to the y-axis is a single interval (possibly empty). A subdivision is monotone if all its regions are monotone. Mehlorn[22] shows that a *simple* planar subdivision (a subdivision with only triangular faces) can be searched in time $O(\log n)$ after spending preprocessing time $O(n)$ and storage space $O(n)$. He also gives the following property to show that the last two bounds apply to *general* subdivisions also, with a small penalty in preprocessing time:

LEMMA 3.1 If the searching problem for simple planar subdivisions with n edges can be solved with search time $O(\log n)$, preprocessing $O(n)$, and space $O(n)$, then the searching problem for general subdivisions with n edges can be solved with the same search time and space but preprocessing $O(n \log n)$. If all faces of the generalized subdivision are convex then $O(n)$ preprocessing is sufficient.
Proof Mehlorn[22]. (Lee and Preparata[18] also show that an arbitrary subdivision with n edges can be refined to a monotone subdivision having at most $2n$ edges in $O(n \log n)$ time.) □

Now we shall review some planar point location algorithms in the literature which either achieve these bounds or come close. Dobkin and Lipton[3] were the pioneers to obtain an $O(\log n)$ query time but they use $O(n^2)$ space. Preparata[27] modifies their method to prove that $O(n \log n)$ space is sufficient. His solution is implementable. In an important paper Lee and Preparata[18] give an algorithm which is based on the construction of separating chains. Their algorithm achieves $O(n \log n)$ preprocessing and $O(n)$ space yet has a query time of $O(\log^2 n)$. The constants hidden in these expressions are small and make their algorithm practically useful. (Their algorithm works for monotone subdivisions only.) Edelsbrunner and Maurer[4] give a space-optimal solution which works for general subdivisions (and even families of nonoverlapping subdivisions). Their query time is $O(\log^3 n)$. Lipton and Tarjan[20, 21] give a method with $O(\log n)$ query time and $O(n)$ preprocessing and space (and thus optimal in all respects). Although based on a graph separator theorem which has many far-reaching consequences, they admit that their method is of only theoretical interest because of the implementation difficulties. Kirkpatrick[16] gives another method with the same bounds. His method builds a hierarchy of subdivisions and seems to be implementable. Finally, Edelsbrunner et al[5] give a substantial improvement of the technique of Lee and Preparata and again attain the optimal bounds in all three respects. The importance of their method is that it seems to admit an efficient implementation along with extensibility to subdivisions with curved edges.

Now we are ready to present our algorithm for BOUNDARY FINDPATH (locus). In the following we show the partitioning for only a face of P (other than F_s) which we shall denote by F_g. To partition bd$P$, we apply the

following algorithm for each face. (Note that no partitioning is necessary for F_s due to convexity.)

Algorithm BOUNDARY FINDPATH (locus): Preprocessing
1. Find all simple face visit sequences between F_s and F_g using *Fgraph* of P.
2. For each face visit sequence found in step 1, compute the image of s with respect to the planar development which starts with face F_g and ends with F_s. (Note that we are *not* required to compute the whole development, but just the image point.)
3. Compute the Voronoi diagram of the image points calculated in step 2 using standard Voronoi programs mentioned above. It is required that with each image point (Voronoi center) we store the face visit sequence which is used to arrive it.
4. Clip the diagram obtained in step 3 with respect to "window" F_g so as to preserve only those parts of it within the polygon F_g. (Any of the standard graphics algorithms such as the one due to Sutherland and Hodge[7] can be used for clipping.)
End

Assume that for a given P, the above preprocessing is carried out for all faces (except F_s). Thus, we have a family of planar subdivisions for each face such that for each region of a given subdivision we know the ordered sequence of faces to be developed to the plane if g specified within that region. More specifically, we can apply the following algorithm when we are queried with a new g:

Algorithm BOUNDARY FINDPATH (locus): Querying
1. Compute the face F_g holding a given g. If $F_g = F_s$ then the shortest path is trivially sg.
2. Using standard planar point location algorithms mentioned above locate g inside the planar subdivision belonging to F_g.
3. Since we stored the face development sequence used to arrive this region we can now use it to compute the s-to-g shortest path. Note that there may be cases where g will be shared by at least two regions and thus there will be at least two shortest paths each of which is obtained via a different development sequence.
End

Figure 8 shows the Voronoi partitioning on a face of a cube using the above algorithm. This figure was drawn by SP. It is noted that in figure 8(a) the given face is partitioned into 4 regions whereas in figure 8(b) this number becomes 6. This effect was obtained by simply moving the source to another location on the source face.

In general, the number of regions a given face F_g is partitioned by this algorithm is expected to be small. An informal argument for this is as follows. Consider the images of s in the plane of F_g. If the face visit sequence for a particular image is long then the image will usually end up in a point farther away from F_g compared to another image obtained by a shorter visit sequence. Thus, only a small portion of the possible face visit sequences between F_s and F_g can render images in the plane close to F_g and contribute to the Voronoi diagram on it.

Acknowledgment

The material reported herein has been supported by the National Science Foundation under grants ECS 80-21504 and ECS 83-51942.

-- o --

Appendix: SP - A Workbench to Compute Shortest Paths

SP is a family of programs written in Franz Lisp[6] and Macsyma command language[13] to experiment with shortest paths. A detailed description of SP is given in [1].

SP was designed with the following philosophy. Let W be a workspace (e.g., a bounding box) which includes a set of polyhedral obstacles. SP is given a geometric description of the members of W and from that point on should be able to compute shortest paths inside W between given pairs of points using the algorithms some of which presented in the preceding sections. It is imperative that SP has some graphics facilities and can supply the user with views of W so that she can have an intuitive feeling about the correctness of a particular computation. In that sense, SP resembles to Verrilli's system[35]; it provides the user with facilities to carry out needed computations, but at the same time needs her intervention here and there. Following a rapid prototyping approach, we either simply excluded those computations which we do not currently know how to perform effectively, or reformulated them to be controlled by user advice at certain points.

Currently, one can only work with a single convex polyhedron using the Franz part of SP. There are facilities to implement BOUNDARY FINDPATH, EXTERIOR FINDPATH, and BOUNDARY FINDPATH (locus). It is also possible to implement an approximate FINDPATH algorithm for a workspace with several convex polyhedra as outlined in section 1 and depicted in figures 1 and 2. Using Macsyma parts of SP it is possible to compute shortest paths in a general workspace with many objects (which may be nonconvex) although this is not fully automated in the light of the combinatorial

explosion that known FINDPATH algorithms have. (Nevertheless, if the user specifies the list of edges that the shortest path must visit, then the problem is solved without much effort.) It is also possible (using Macsyma) to work on FINDPATH (locus) (which is the general version of BOUNDARY FINDPATH (locus)) although this is not automated yet. (In [11] all computations were done in this way.)

Finally, we give some additional examples computed and drawn by SP. Figure 9(a) and (b) show two shortest paths on the boundary of a dodecahedron. Similarly, figure 10 shows a shortest path on the boundary of an icosahedron. Figure 11 demonstrates a shortest path around a cube. This was computed after constructing a new object via EXTERIOR FINDPATH and then applying BOUNDARY FINDPATH on it. Figure 12 shows the partitioning of the boundary of a cube in the presence of a source point on face 1. Figure 12(a), (b), (c), (d), and (e) respectively depict the regions induced on goal faces 2, 3, 4, 5, and 6.

References

1. V. Akman, "Shortest Paths Avoiding Polyhedral Obstacles in 3-Dimensional Euclidean Space," Ph.D. dissertation, Dept. of Electrical, Computer, and Systems Eng., Rensselaer Polytechnic Institute, Troy, N.Y., Jun. 1985.
2. D. Avis and B. K. Bhattacharya, "Algorithms for Computing the d-Dimensional Voronoi Diagrams and Their Duals," in *Advances in Computing Research*, ed. F. P. Preparata, pp. 159-180, JAI Press, 1983.
3. D. P. Dobkin and R. J. Lipton, "Multidimensional Searching Problems," *SIAM Journal on Computing*, vol. 5, no. 2, pp. 181-186, 1976.
4. H. Edelsbrunner and H. A. Maurer, "A Space-optimal Solution of General Region Location," *Theoretical Computer Science*, vol. 16, pp. 329-336, 1981.
5. H. Edelsbrunner, L. J. Guibas, and J. Stolfi, "Optimal Point Location in a Monotone Subdivision," Tech. Rep. 2, DEC Systems Research Center, Palo Alto, CA, Oct. 1984.
6. J. K. Foderaro, K. L. Sklower, and K. Layer, *The FRANZ LISP Manual,* Univ. of California, Berkeley, CA, Jun. 1983.
7. J. D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, Reading, MA, 1982.
8. W. R. Franklin and V. Akman, "Shortest Paths between Source and Goal Points Located on/around a Convex Polyhedron," *Proc. of the 22nd Allerton Conf. on Communication, Control, and Computing*, Monticello, IL, Sep. 1984.
9. W. R. Franklin, V. Akman, and C. Verrilli, "Voronoi Diagrams with Barriers and on Polyhedra for Minimal Path Planning," *The Visual Computer - An International Journal on Computer Graphics*, Springer-Verlag, 1985 (to appear).
10. W. R. Franklin and V. Akman, "Partitioning the Space to Calculate Shortest Paths to any Goal around Polyhedral Obstacles," *Proc. of the 1st Annual Workshop on Robotics and Expert Systems (Houston, TX, Jul. 1985)*, Instrumentation Society of America, 1985 (to appear).

11. W. R. Franklin and V. Akman, "Euclidean Shortest Paths in 3-Space, Voronoi Diagrams with Barriers, and Related Complexity and Algebraic Issues," *Proc. of the NATO Advanced Study Institute on Fundamental Algorithms for Computer Graphics (Ilkley, Yorkshire, Apr. 1985)*, Springer-Verlag, 1985 (to appear).

12. M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-completeness,* W. H. Freeman, San Francisco, CA, 1979.

13. Mathlab Group, *MACSYMA Reference Manual,* 2 vols., Lab. for Computer Science, Massachusetts Inst. of Technology, Cambridge, MA, 1983.

14. L. J. Guibas and J. Stolfi, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams," *Proc. 15th Annual ACM Symp. on Theory of Computing*, pp. 221-234, 1983.

15. N. Katoh, T. Ibaraki, and H. Mine, "An Efficient Algorithm for k-Shortest Simple Paths," *Networks*, vol. 12, pp. 411-427, 1982.

16. D. G. Kirkpatrick, "Optimal Search in Planar Subdivisions," *SIAM Journal on Computing*, vol. 12, no. 1, pp. 28-35, Feb. 1983.

17. V. Klee, "On the Complexity of d-Dimensional Voronoi Diagrams," *Archiv der Mathematik*, vol. 34, pp. 75-80, 1980.

18. D. T. Lee and F. P. Preparata, "Location of a Point in a Planar Subdivision and its Applications," *SIAM Journal on Computing*, vol. 6, no. 3, pp. 594-606, Sep. 1977.

19. D. T. Lee, "On k-Nearest Neighbor Voronoi Diagrams in the Plane," *IEEE Trans. on Computers*, vol. 31, no. 6, pp. 478-487, Jun. 1982.

20. R. J. Lipton and R. E. Tarjan, "A Separator Theorem for Planar Graphs," *SIAM Journal on Applied Mathematics*, vol. 36, no. 2, pp. 177-189, Apr. 1979.

21. R. J. Lipton and R. E. Tarjan, "Applications of a Planar Separator Theorem," *SIAM Journal on Computing*, vol. 9, no. 3, pp. 615-627, Aug. 1980.

22. K. Mehlorn, *Data Structures and Algorithms (vol. 3: Multi-dimensional Searching and Computational Geometry),* 3 vols., Springer-Verlag, Heidelberg, Berlin, 1984.

23. J. S. B. Mitchell and C. H. Papadimitriou, "The Discrete Geodesic Problem," Manuscript, Dept. of Computer Science, Stanford Univ., Stanford, CA, 1985.

24. J. O'Rourke, S. Suri, and H. Booth, "Shortest Paths on Polyhedral Surfaces," *Proc. of the 2nd Annual Symp. on Theoretical Aspects of Computer Science (Lecture Notes on Computer Science 182)*, pp. 243-254, Springer-Verlag, New York, Jan. 1985.

25. M. H. Overmars, "The Locus Approach," Tech. Rep. RUU-CS-83-12, Computer Science Dept., Univ. of Utrecht, Utrecht, the Netherlands, Jul. 1983.

26. F. P. Preparata and S. J. Hong, "Convex Hulls of Finite Sets of Points in Two and Three Dimensions," *Communications of the ACM*, vol. 20, no. 2, pp. 87-93, Feb. 1977.

27. F. P. Preparata, "A New Approach to Planar Point Location," *SIAM Journal on Computing*, vol. 10, no. 3, pp. 473-482, Aug. 1981.

28. J. S. Provan, "The Complexity of Reliability Computations in Planar and Acyclic Graphs," Tech. Rep. 83/12, Operations Research and Systems Analysis Curriculum, Univ. of North Carolina, Chapel Hill, NC, Dec. 1984.

29. J. S. Provan, private communication, 1985.

30. M. I. Shamos and D. Hoey, "Closest-point Problems," *Proc. of the 16th IEEE Annual Symp. on Foundations of Computer Science*, pp. 151-162, Oct. 1975.

31. M. I. Shamos, "Computational Geometry," Ph.D. dissertation, Dept. of Computer Science, Yale Univ., New Haven, CT, 1978.

32. M. Sharir and A. Schorr, "On Shortest Paths in Polyhedral Spaces," *Proc. of the 16th Annual ACM Symp. on Theory of Computing*, pp. 144-153, 1984.

33. I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A Characterization of Ten Hidden-surface Algorithms," *ACM Computing Surveys*, vol. 6, no. 1, pp. 1-55, Mar. 1974.

34. L. G. Valiant, "The Complexity of Enumeration and Reliability Problems," *SIAM Journal on Computing*, vol. 8, no. 3, pp. 410-421, Aug. 1979.

35. C. Verrilli, "One Source Voronoi Diagrams with Barriers, a Computer Implementation," Tech. Rep. IPL-TR-060, Image Processing Lab., Rensselaer Polytechnic Inst., Troy, N.Y., Feb. 1984.

36. J. Y. Yen, "Finding the k-Shortest Loopless Paths in a Network," *Management Science*, vol. 17, no. 11, pp. 712-716, Jul. 1971.

Figures

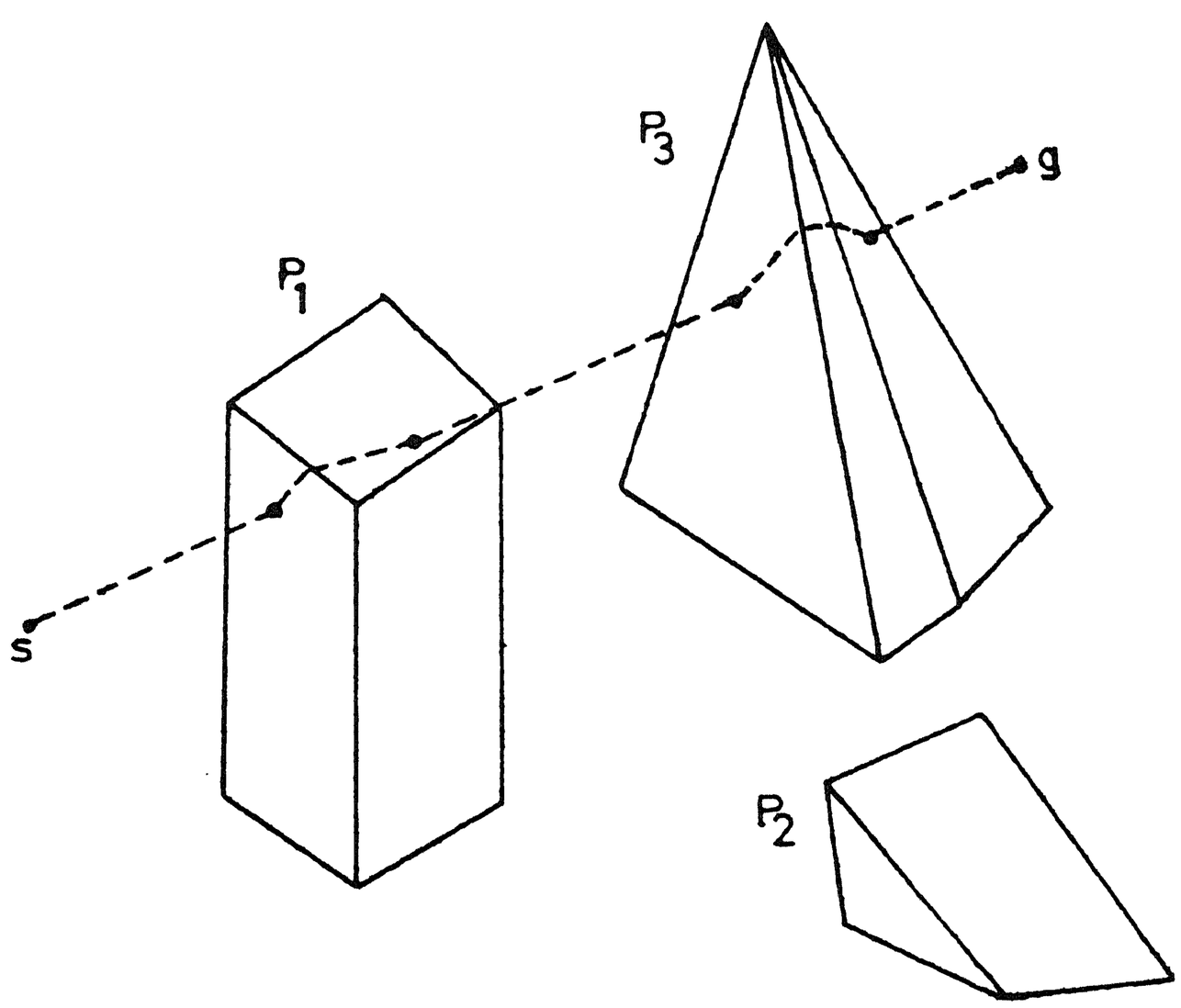


Figure 1 A reasonably short path between s and g in the presence of convex polyhedral obstacles.

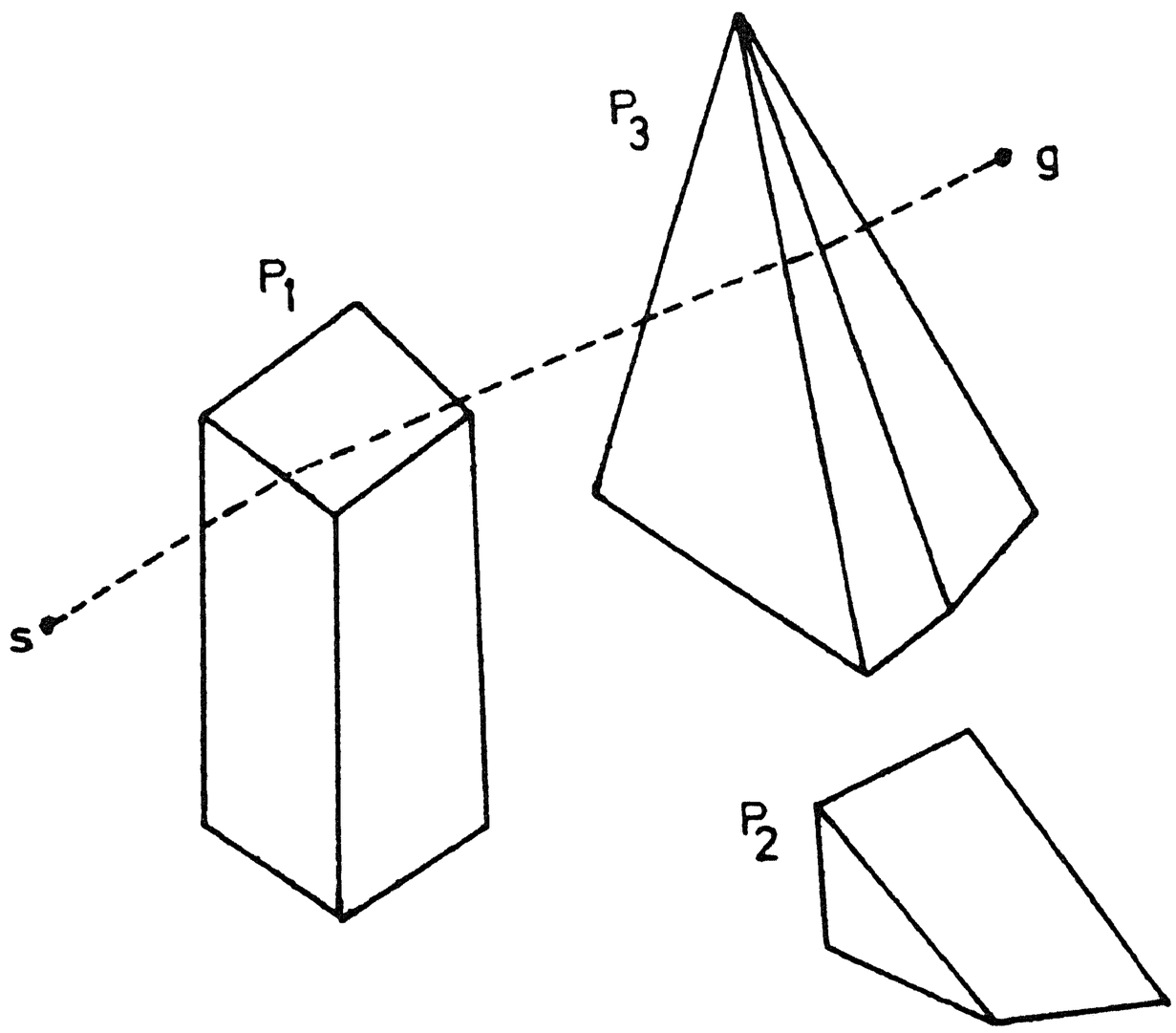


Figure 2 Further optimization of the path shown in figure 1 yields a shorter path with fewer bend points.

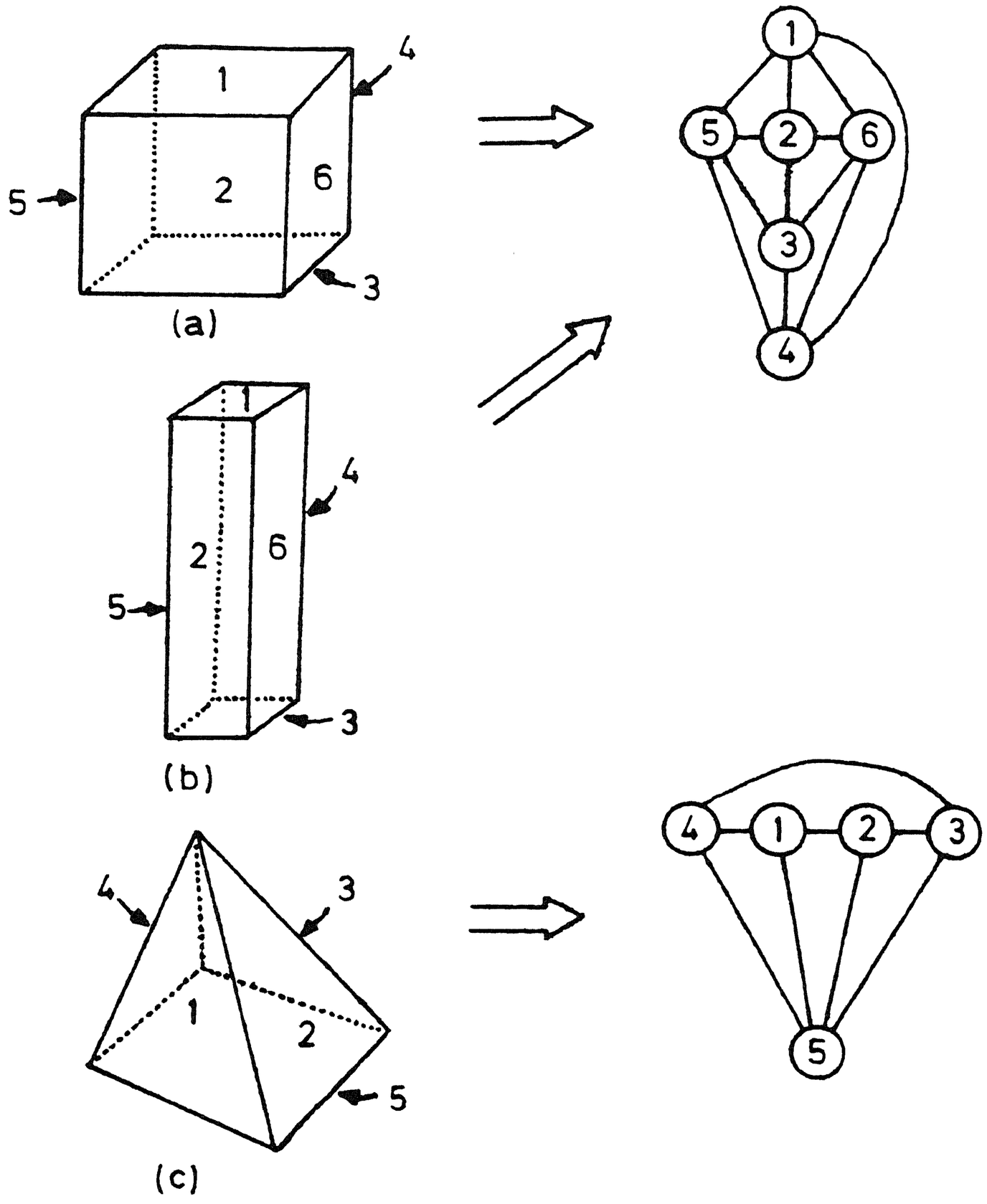
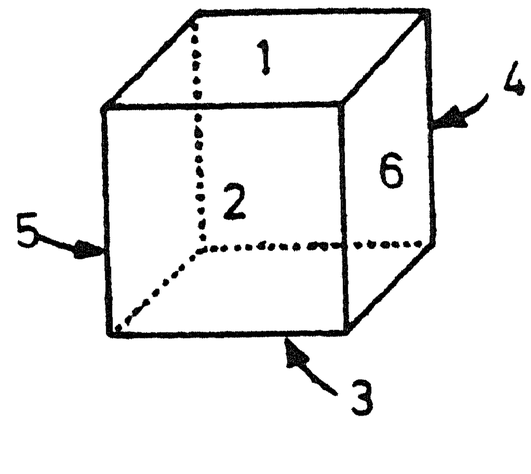


Figure 3 Face graphs of convex polyhedra:
(a) cube, (b) prism, (c) pyramid.



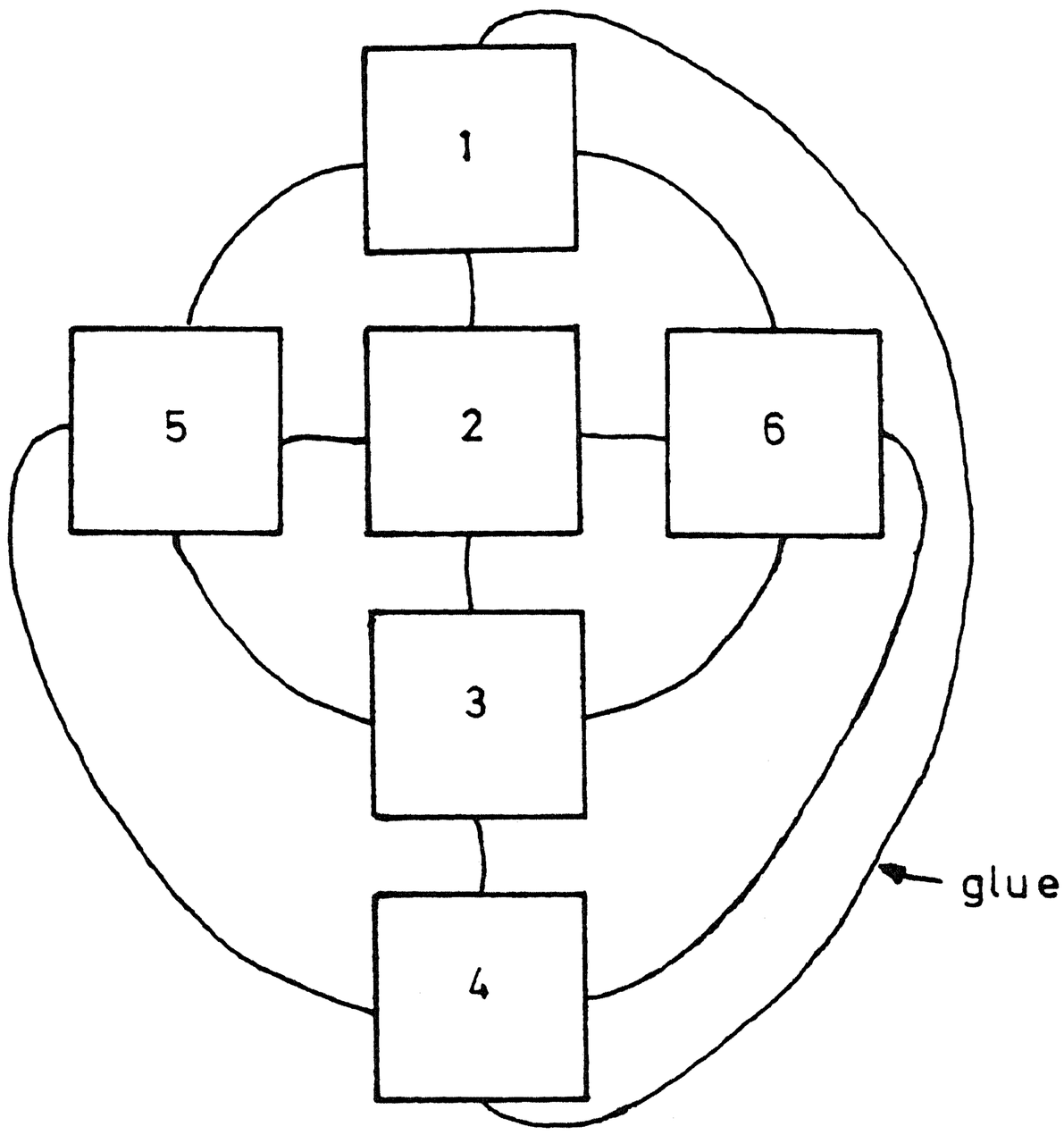


Figure 4 The planar polygonal schema of a cube.

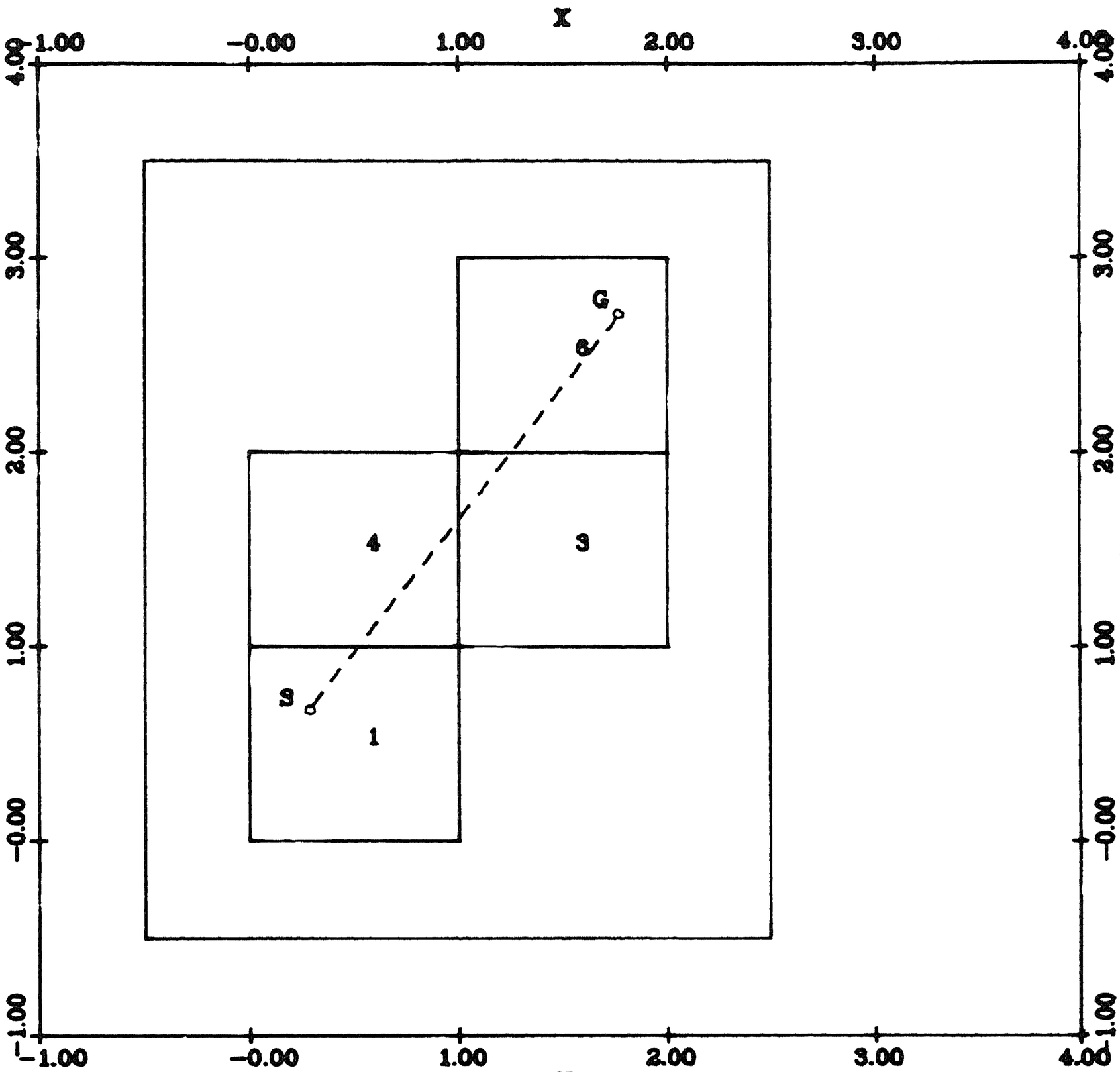
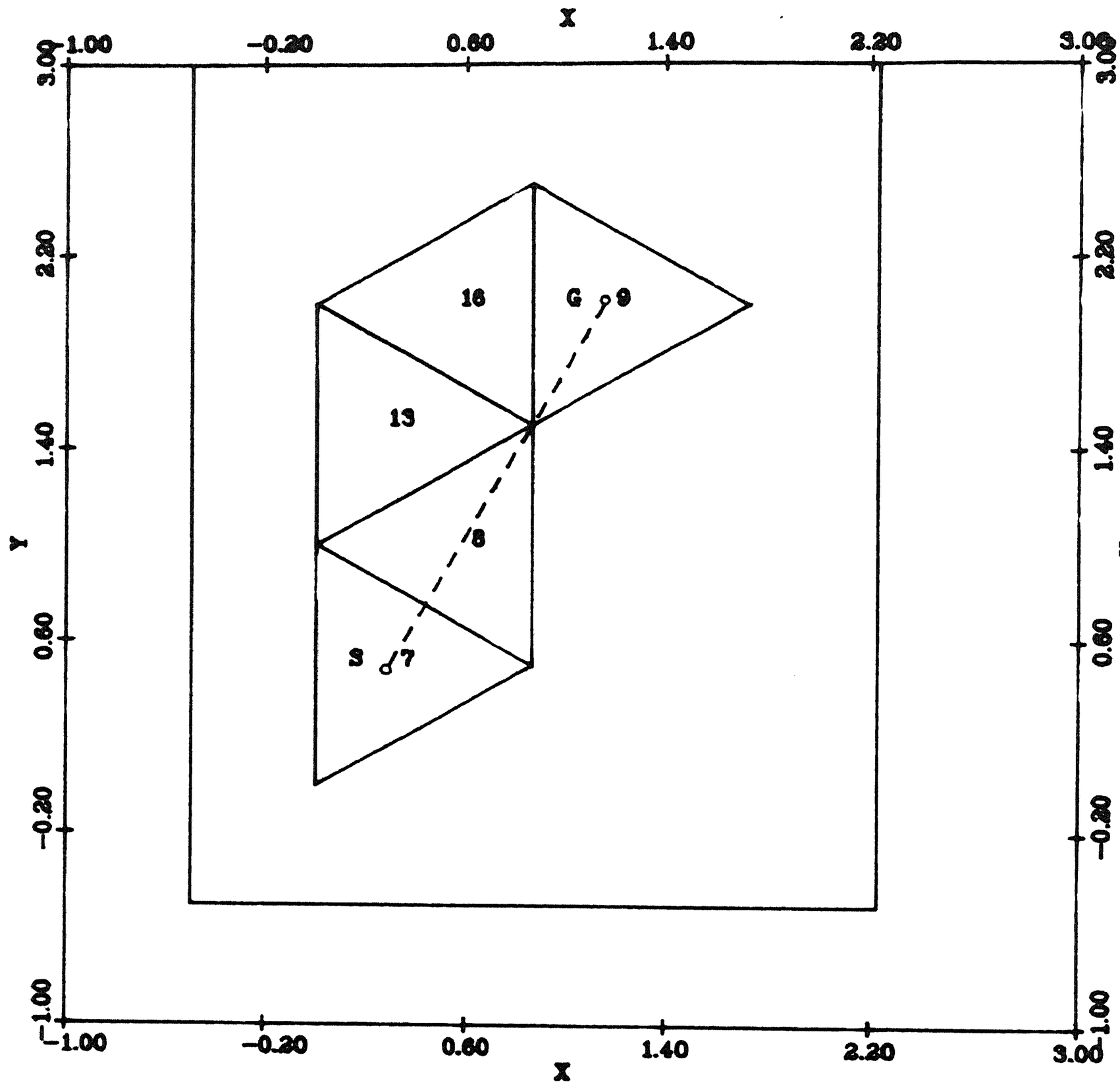
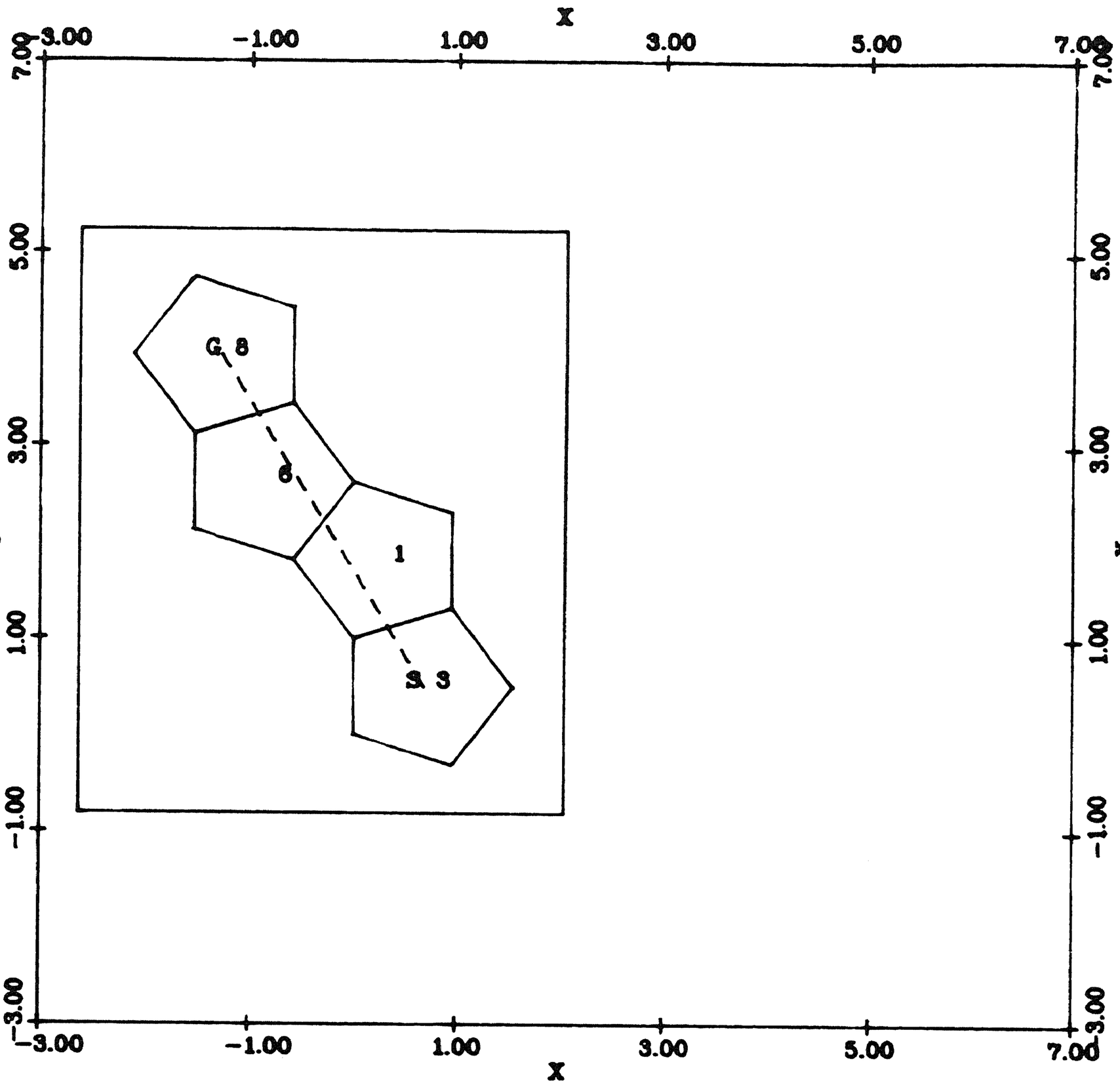


Figure 5 Some planar developments computed and drawn by SP. The objects that are developed are as follows: (a) cube, (b) icosahedron, (c) and (d) dodecahedron.



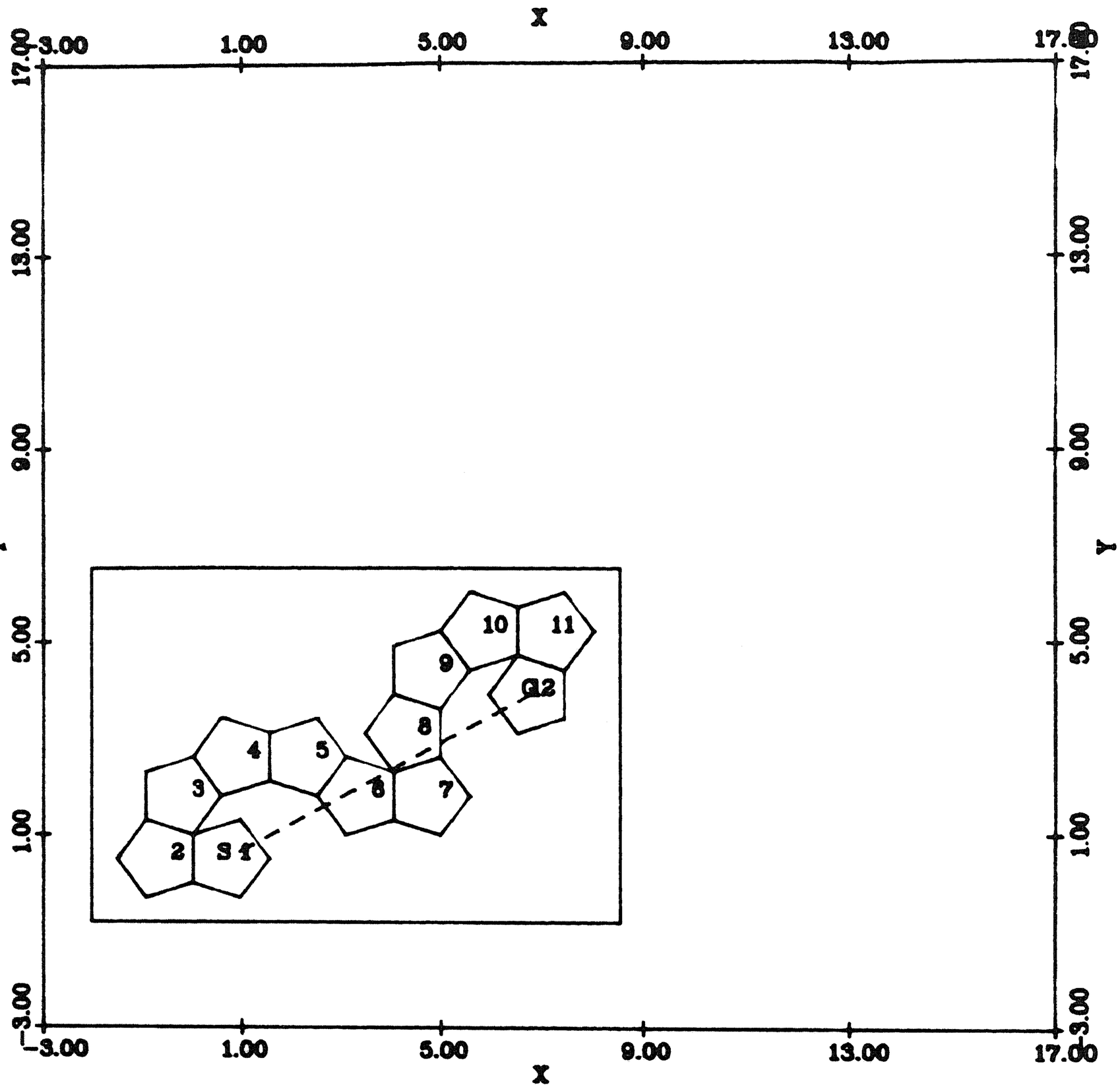
Face development sequence: 7 8 13 16 9
Source face no.: 7
Goal face no.: 9
Source point coords.: 0.948 0.230 0.504
Goal point coords.: 0.192 1.208 0.504
Source point coords. (plane): 0.289 0.500
Goal point coords. (plane): 1.154 2.000
Source-to-goal distance: 1.732

(b)



Face development sequence:	3	1	6	8
Source face no.:	3			
Goal face no.:	8			
Source point coords.:	0.380	1.447	0.616	
Goal point coords.:	0.996	−0.447	1.612	
Source point coords. (plane):	0.688	0.500		
Goal point coords. (plane):	−1.276	3.927		
Source-to-goal distance:	3.950			

(c)



Face development sequence:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Source face no.:		**1**										
Goal face no.:		**12**										
Source point coords.:			**0.688**		**0.500**			**0.000**				
Goal point coords.:			**0.688**		**0.500**			**2.227**				
Source point coords. (plane):			**0.688**		**0.500**							
Goal point coords. (plane):			**6.782**		**3.927**							
Source-to-goal distance:			**6.991**									

(d)

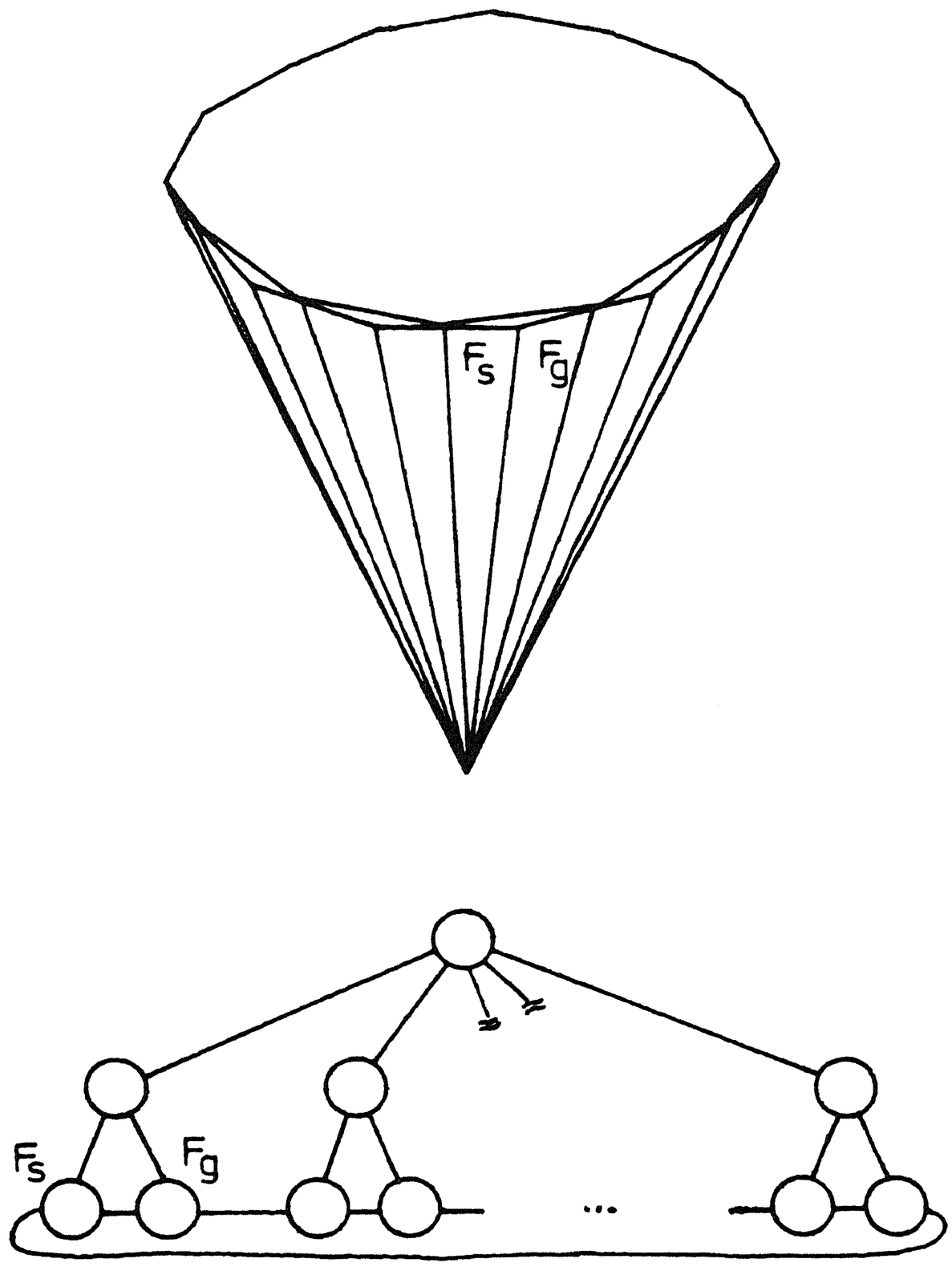


Figure 6 There exist an exponential number of simple walks between nodes F_s and F_g in *Fgraph* of this object.

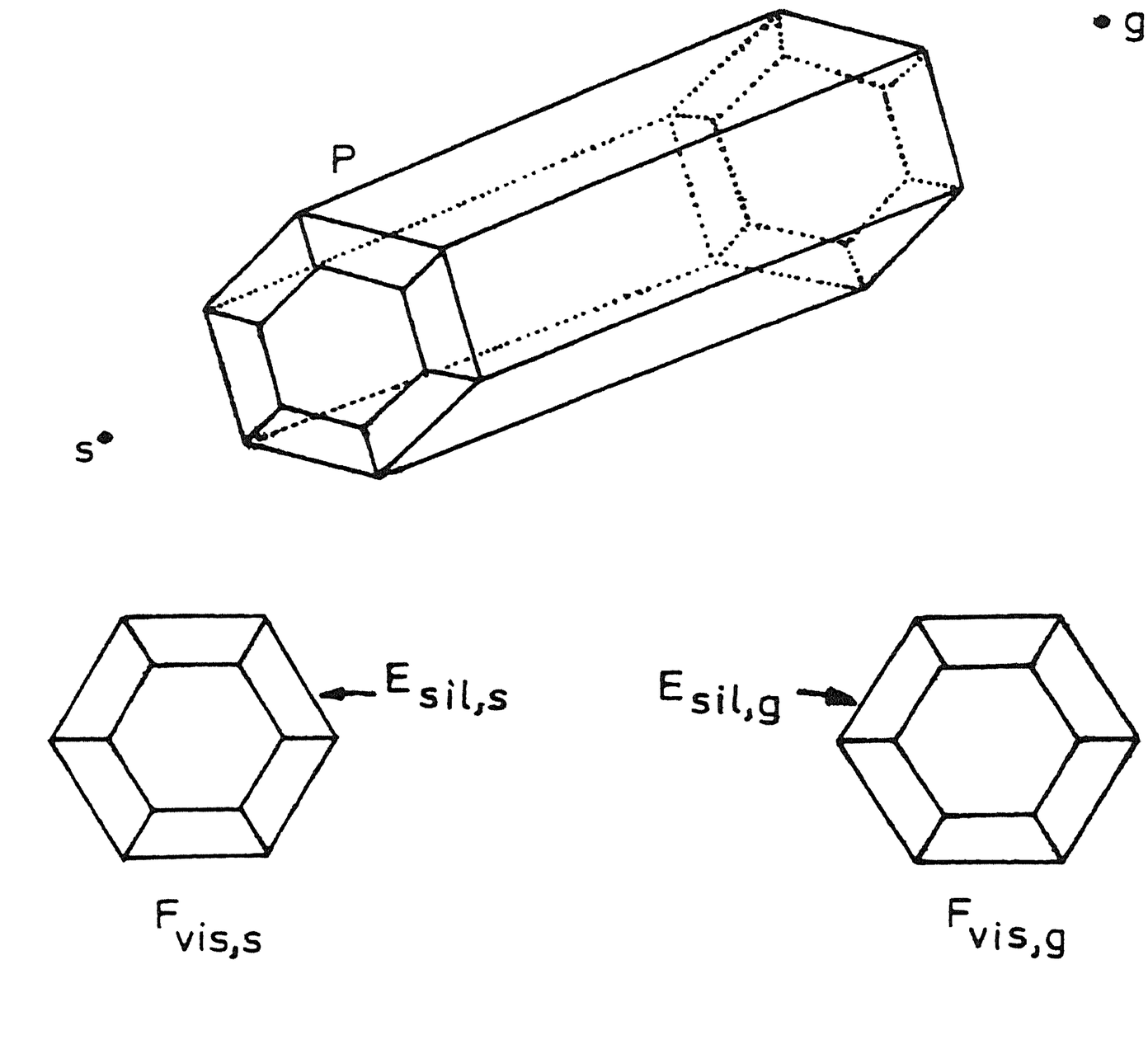
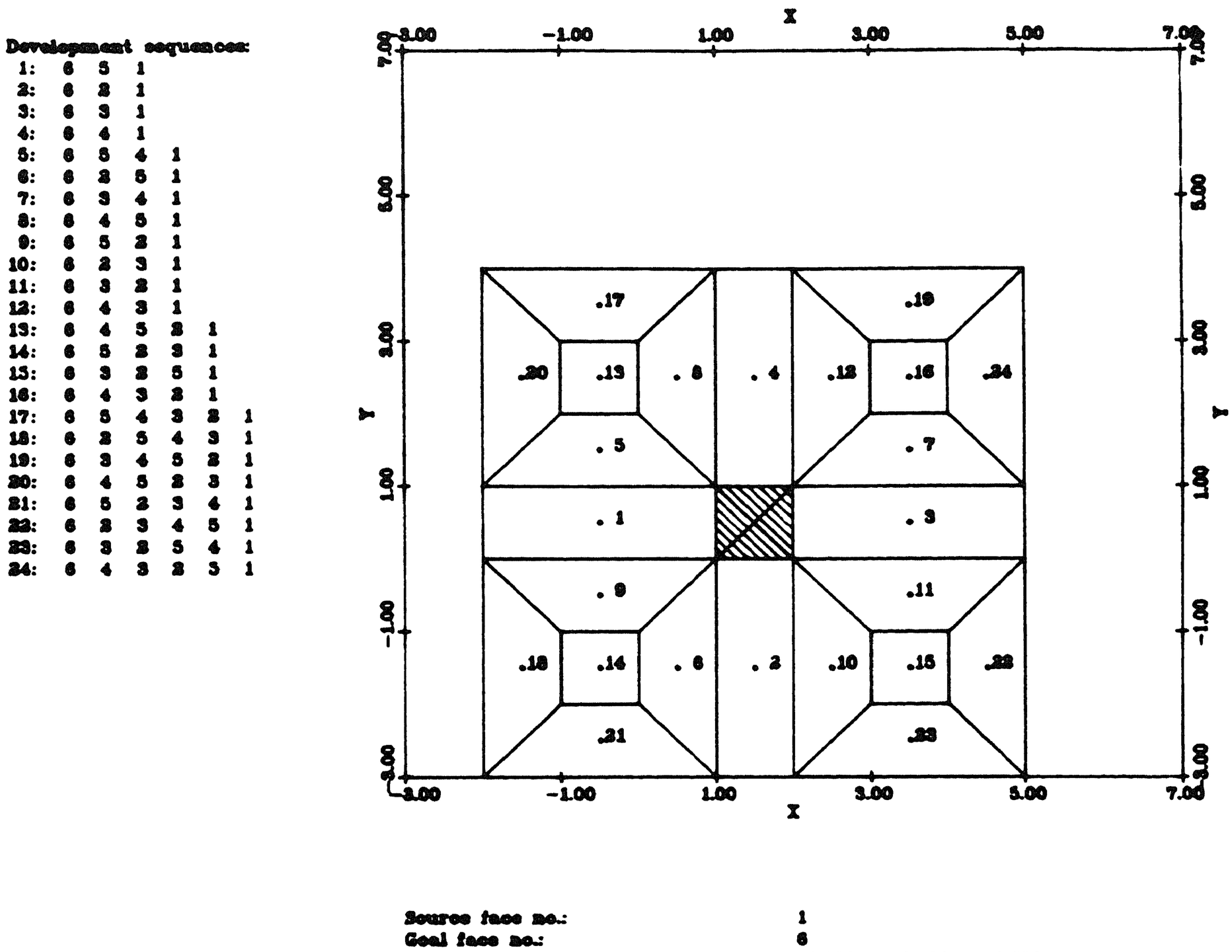


Figure 7 Demonstration of how EXTERIOR FINDPATH works via silhouettes.

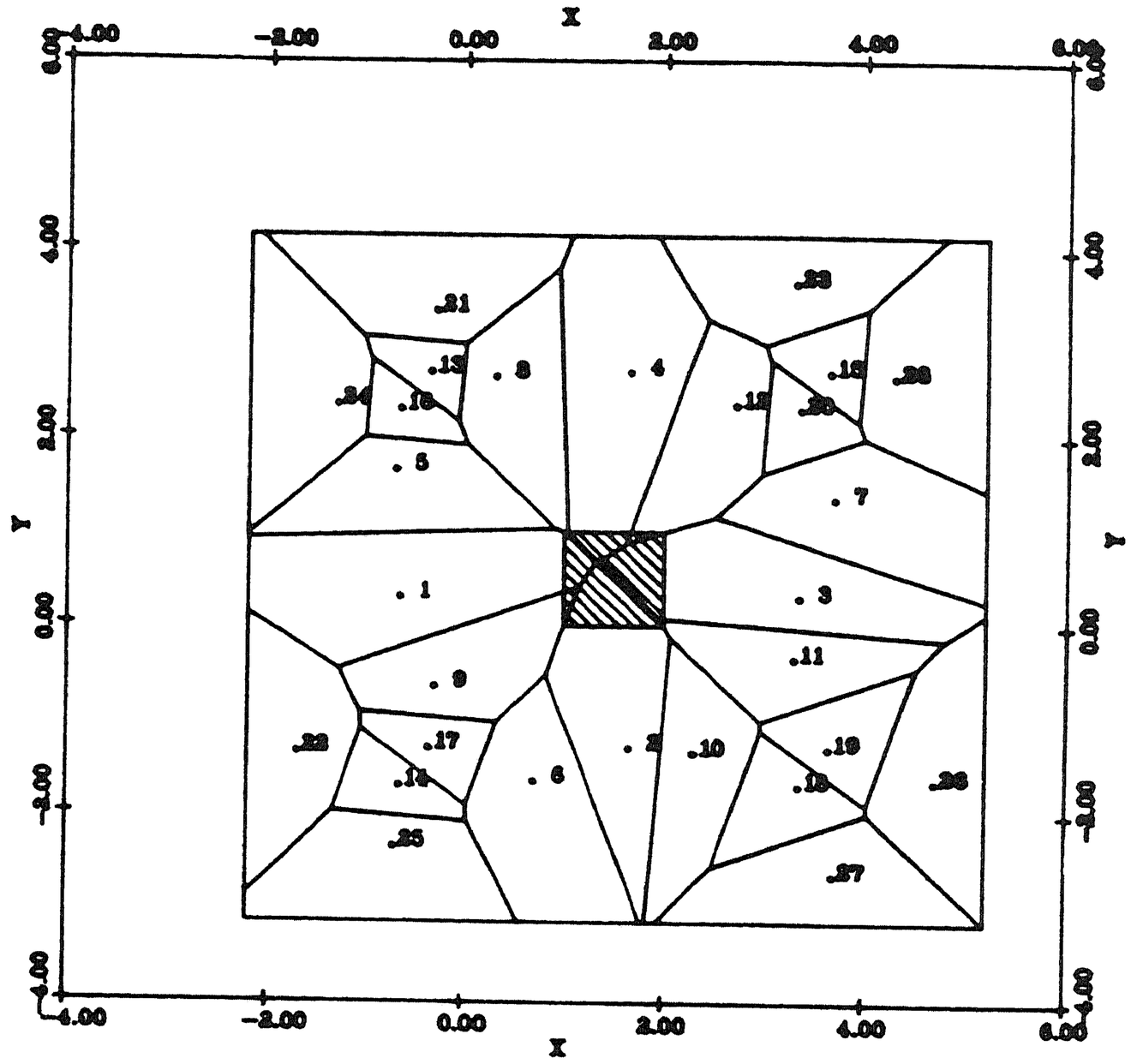


(a)

Figure 8 Demonstration of BOUNDARY FINDPATH (locus) on a face of a cube. In figure 8(a) there are 4 regions on the goal face (the shaded polygon) as a result of Voronoi partitioning. Figure 8(b) shows the effect of moving the source on the source face to another location.

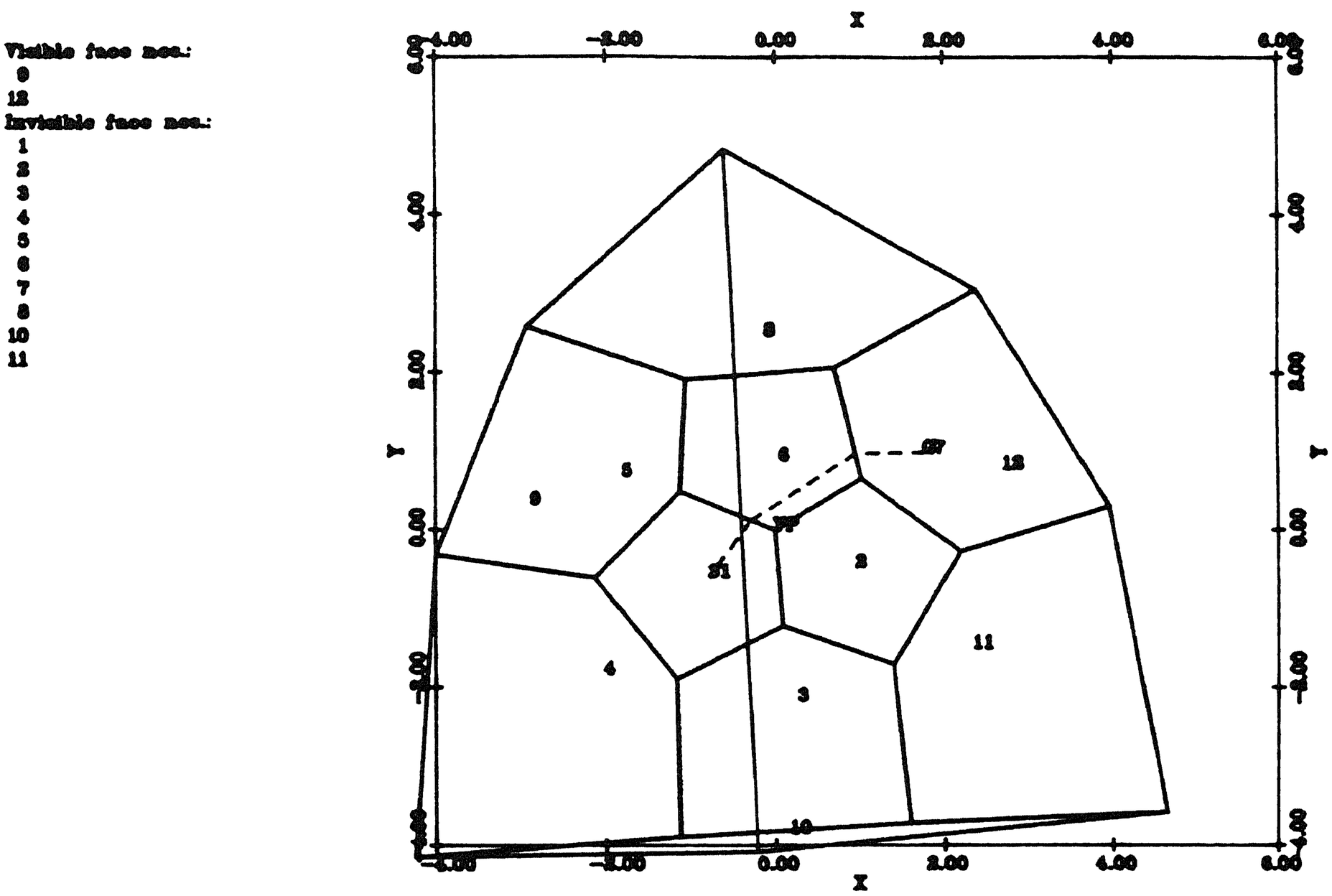
Development sequences:

1:	6	5	1			
2:	6	2	1			
3:	6	3	1			
4:	6	4	1			
5:	6	5	4	1		
6:	6	2	5	1		
7:	6	3	4	1		
8:	6	4	5	1		
9:	6	5	2	1		
10:	6	2	3	1		
11:	6	3	2	1		
12:	6	4	3	1		
13:	6	5	4	3	1	
14:	6	2	5	4	1	
15:	6	3	4	5	1	
16:	6	4	5	2	1	
17:	6	5	2	3	1	
18:	6	2	3	4	1	
19:	6	3	2	5	1	
20:	6	4	3	2	1	
21:	6	5	4	3	2	1
22:	6	2	5	4	3	1
23:	6	3	4	5	2	1
24:	6	4	5	2	3	1
25:	6	5	2	3	4	1
26:	6	2	3	4	5	1
27:	6	3	2	5	4	1
28:	6	4	3	2	5	1



Source face no.: 1
Goal face no.: 6
Source point coords.: 0.000 0.300 0.850
No. of developments: 28

(b)

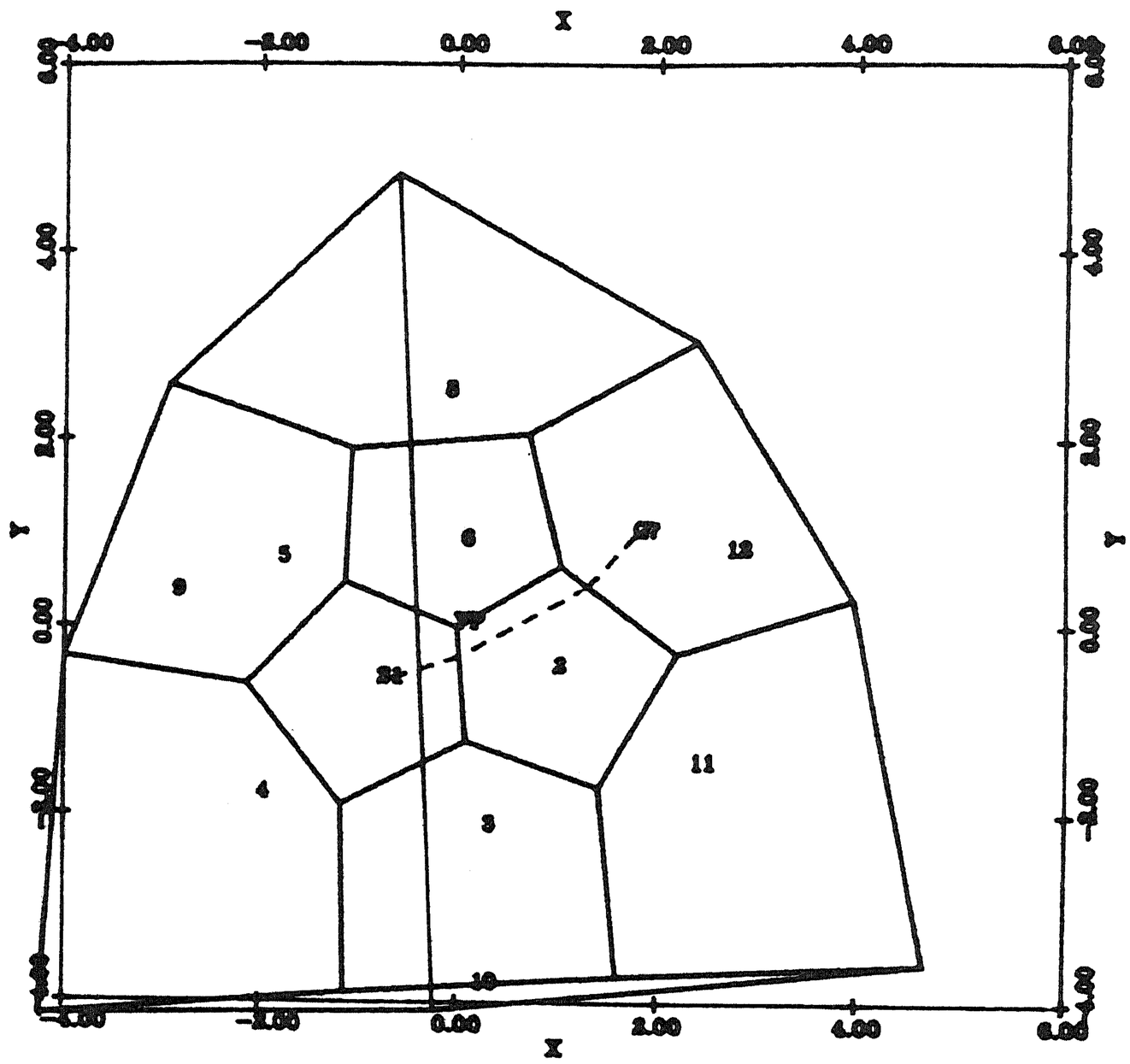


Viewpoint: 1.500 0.000 2.500
Source face no.: 1
Goal face no.: 7
Source point: 0.000 0.500 0.000
Goal point: -0.118 -0.086 1.611
Face development sequence: 1 6 7
Shortest path length: 2.618
Shortest path bend points: 0.000 0.500 0.000
0.263 -0.086 0.000
-0.236 -0.447 0.000
-0.118 -0.086 1.611

(a)

Figure 9 Shortest paths on the boundary of a dodecahedron.
Both (a) and (b) are shortest paths. This is a perspective view of the object as computed by SP.

```
Visible face nos.:
 9
 12
Invisible face nos.:
 1
 2
 3
 4
 5
 6
 7
 8
 10
 11
```




```
Viewpoint:                          1.500     0.800     2.500
Source face no.:                    1
Goal face no.:                      7
Source point:                       0.688     0.500     0.000
Goal point:                        -0.118    -0.086     1.611
Face development sequence:      1   2   7
Shortest path length:               2.618
Shortest path bend points:          0.688     0.500     0.000
                                    0.000     0.276     0.000
                                   -0.498    -0.086     0.966
                                   -0.118    -0.086     1.611
```

(6)

Figure 10 A shortest path on the boundary of an icosahedron.
This was computed by SP.

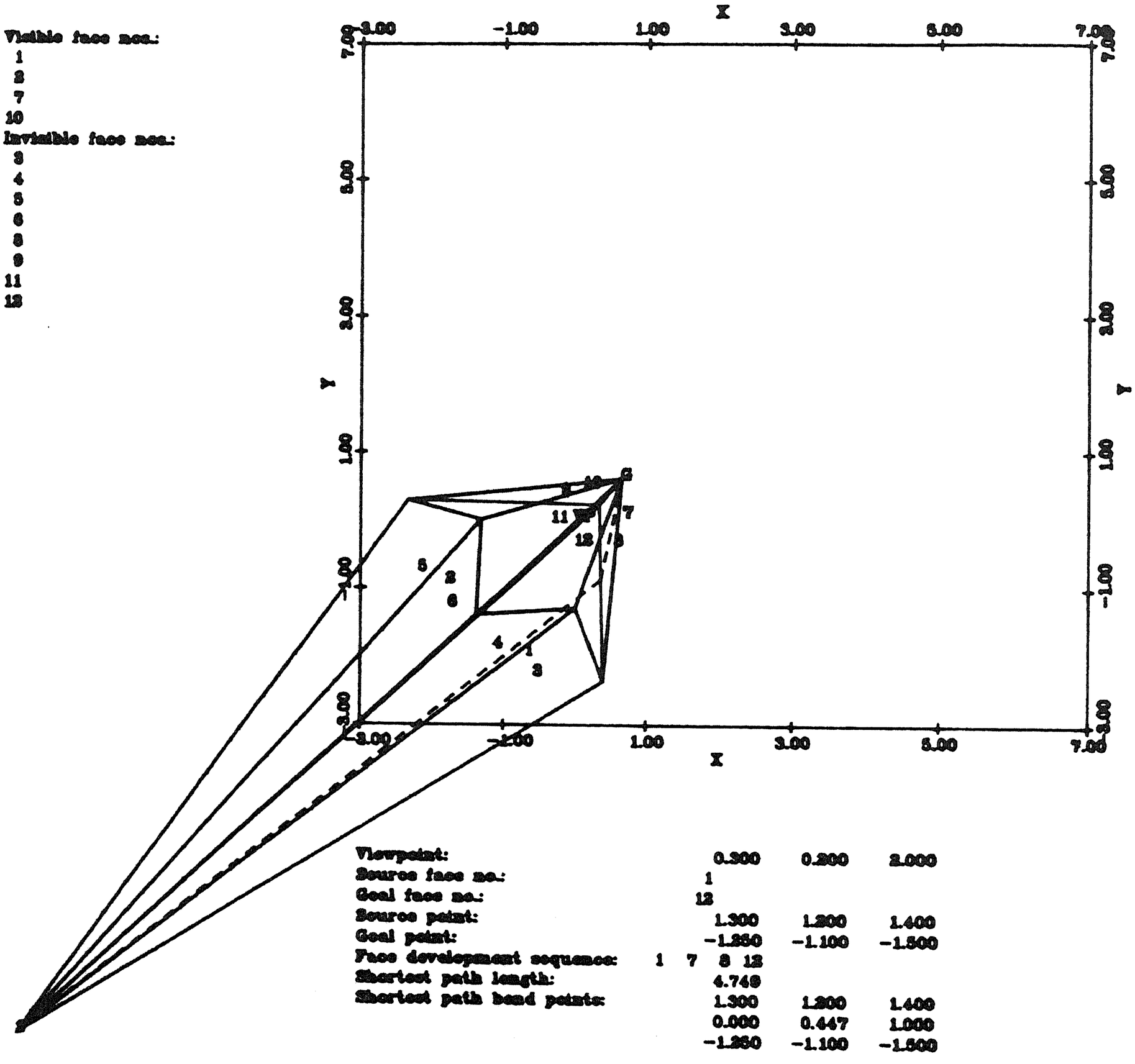


Figure 11 A shortest path around a cube. This was computed by SP after computing the new object and then applying BOUNDARY FINDPATH on it.

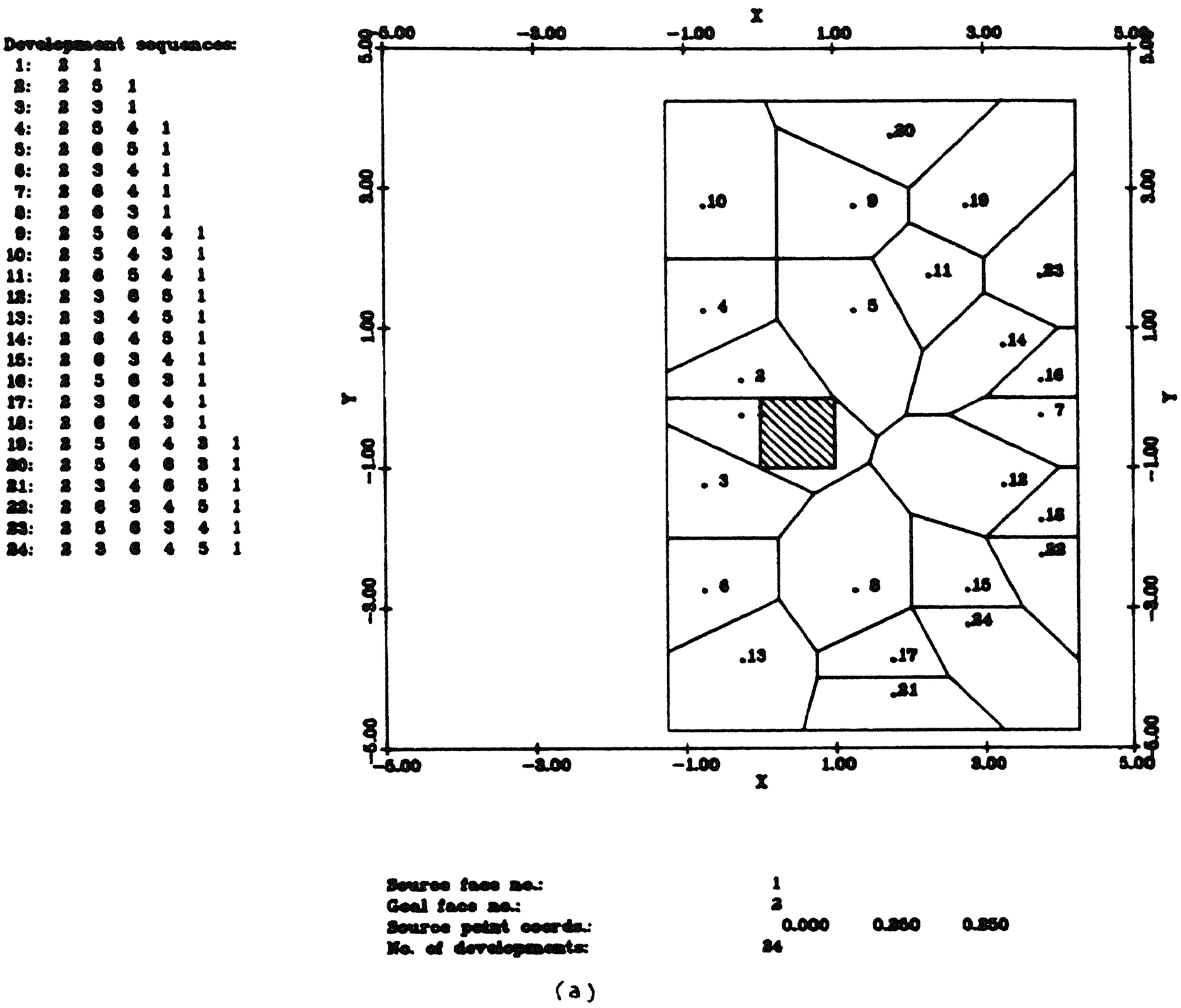
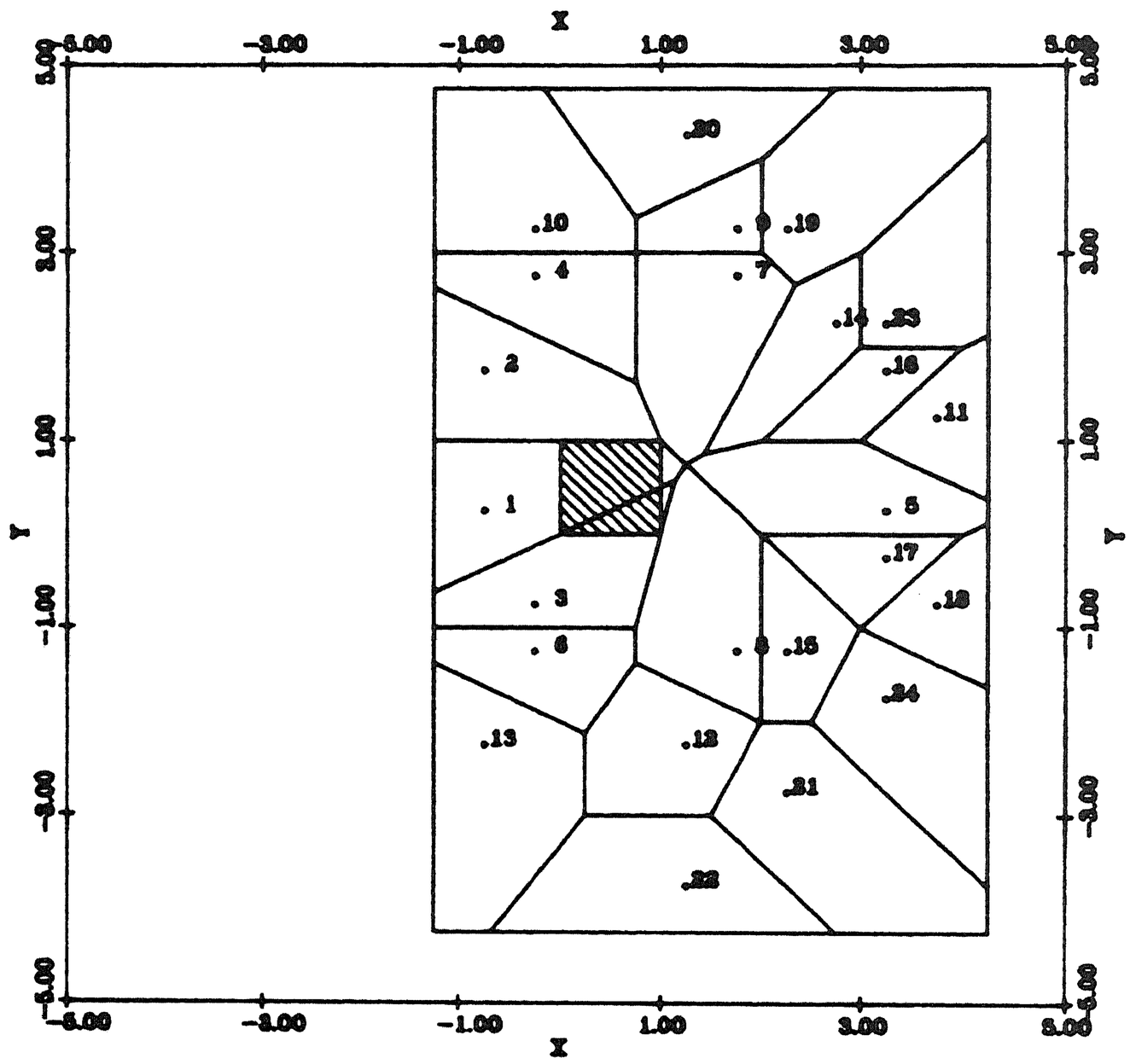


Figure 12 This shows the partitioning of the boundary of a cube in the presence of a source on face 1. Parts (a), (b), (c), (d), and (e) respectively show the regions induced on goal faces 2, 3, 4, 5, and 6. These figures were computed by SP.

Development sequences:

1:	3	1				
2:	3	4	1			
3:	3	2	1			
4:	3	4	5	1		
5:	3	6	5	1		
6:	3	2	5	1		
7:	3	6	4	1		
8:	3	6	2	1		
9:	3	4	6	5	1	
10:	3	4	5	2	1	
11:	3	6	5	4	1	
12:	3	2	6	5	1	
13:	3	2	5	4	1	
14:	3	6	4	5	1	
15:	3	6	2	5	1	
16:	3	4	6	2	1	
17:	3	6	5	2	1	
18:	3	2	6	4	1	
19:	3	4	6	5	2	1
20:	3	4	5	6	2	1
21:	3	2	6	5	4	1
22:	3	2	5	6	4	1
23:	3	4	6	2	5	1
24:	3	2	6	4	5	1

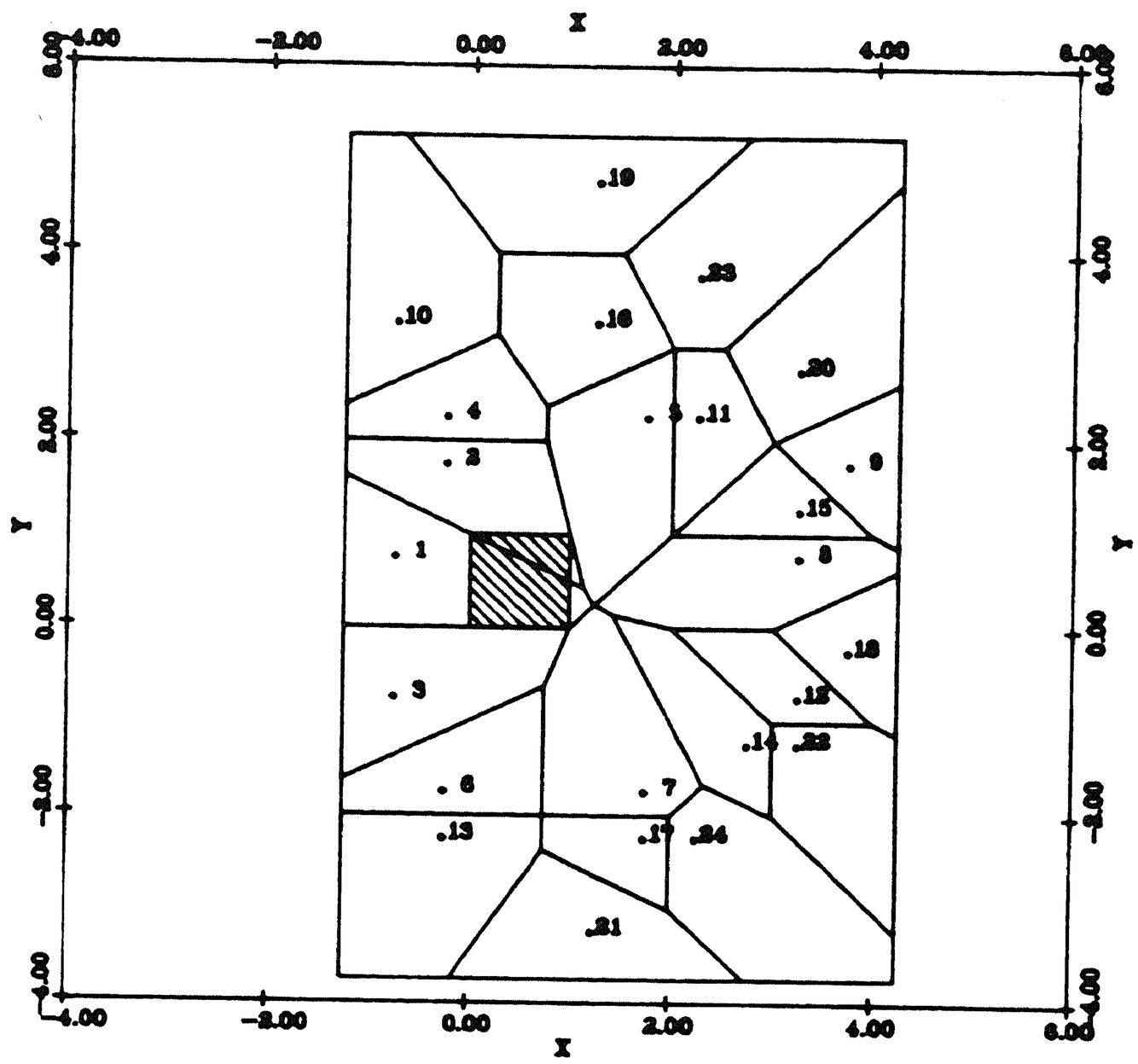


Source face no.: 1
Coal face no.: 3
Source point coords.: 0.000 0.250 0.250
No. of developments: 24

(b)

Development sequences:

1:	4	1				
2:	4	5	1			
3:	4	3	1			
4:	4	5	2	1		
5:	4	6	5	1		
6:	4	3	2	1		
7:	4	6	3	1		
8:	4	6	2	1		
9:	4	5	6	3	1	
10:	4	5	2	3	1	
11:	4	6	5	2	1	
12:	4	3	6	5	1	
13:	4	3	2	5	1	
14:	4	6	3	2	1	
15:	4	6	2	5	1	
16:	4	5	6	2	1	
17:	4	3	6	2	1	
18:	4	6	2	3	1	
19:	4	5	2	6	3	1
20:	4	6	5	2	3	1
21:	4	3	2	6	5	1
22:	4	6	3	2	5	1
23:	4	5	6	2	3	1
24:	4	3	6	2	5	1

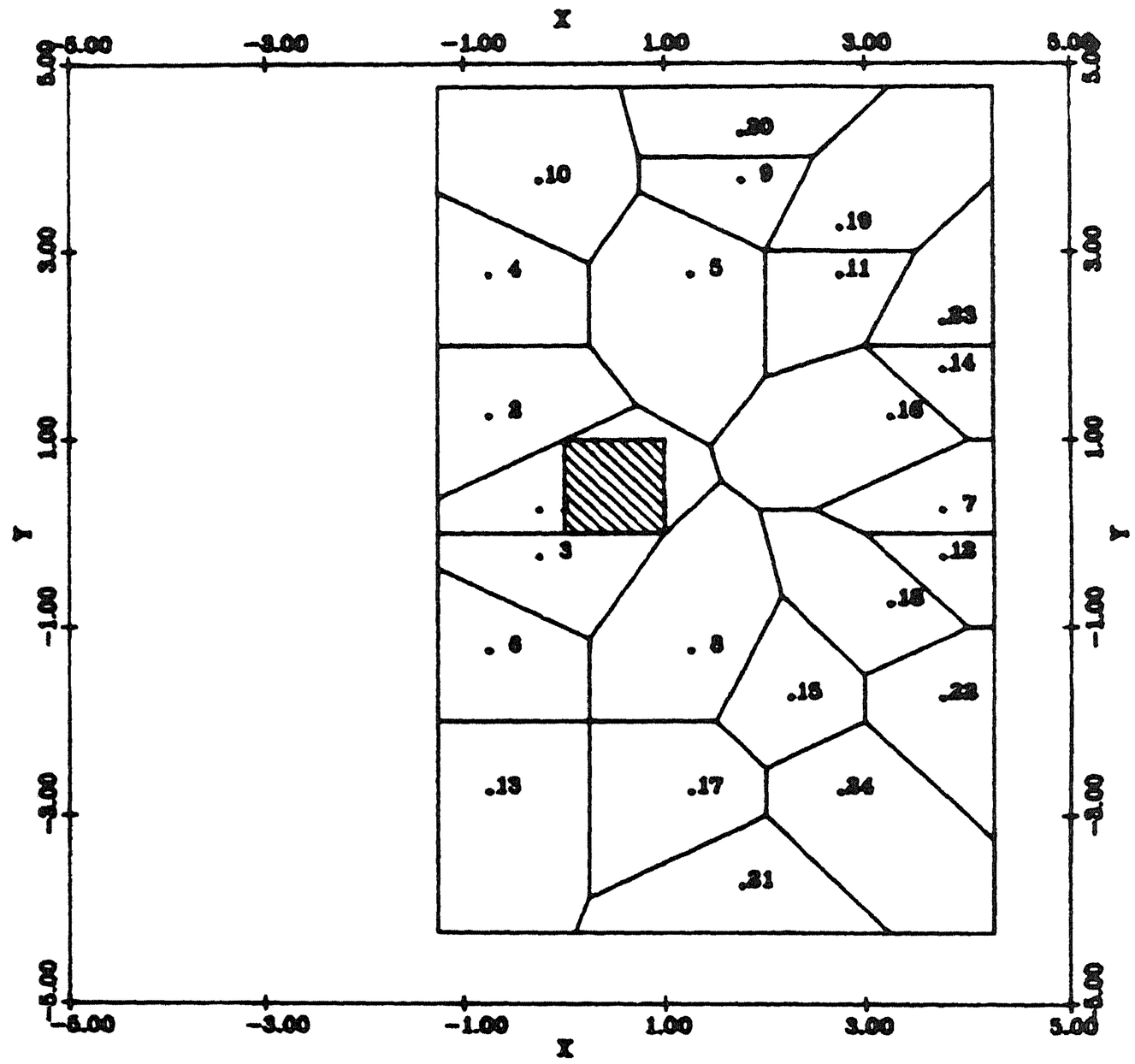


Source face no.: 1
Goal face no.: 4
Source point coords.: 0.000 0.250 0.250
No. of developments: 24

(c)

Development sequences:

1:	5	1				
2:	5	4	1			
3:	5	2	1			
4:	5	4	3	1		
5:	5	6	4	1		
6:	5	2	3	1		
7:	5	6	3	1		
8:	5	6	2	1		
9:	5	4	6	3	1	
10:	5	4	3	2	1	
11:	5	6	4	3	1	
12:	5	2	6	4	1	
13:	5	2	3	4	1	
14:	5	6	3	4	1	
15:	5	6	2	3	1	
16:	5	4	6	2	1	
17:	5	2	6	3	1	
18:	5	6	3	2	1	
19:	5	4	6	3	2	1
20:	5	4	3	6	2	1
21:	5	2	3	6	4	1
22:	5	6	2	3	4	1
23:	5	4	6	2	3	1
24:	5	2	6	3	4	1

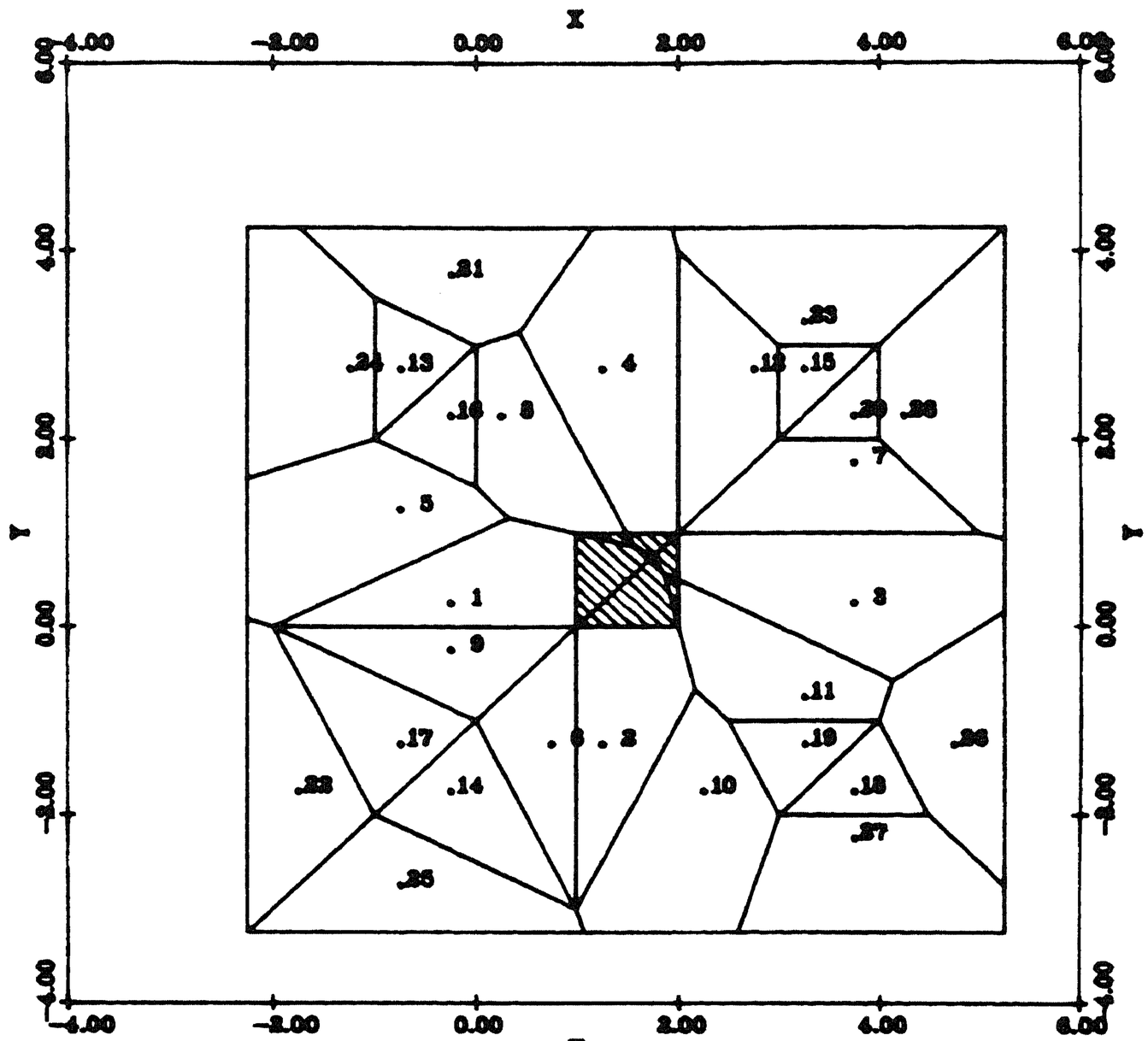


Source face no.: 1
Goal face no.: 5
Source point coords.: 0.000 0.250 0.250
No. of developments: 24

(d)

Development sequences:

1:	6	5	1			
2:	6	2	1			
3:	6	3	1			
4:	6	4	1			
5:	6	5	4	1		
6:	6	2	5	1		
7:	6	3	4	1		
8:	6	4	5	1		
9:	6	5	2	1		
10:	6	2	3	1		
11:	6	3	2	1		
12:	6	4	3	1		
13:	6	5	4	3	1	
14:	6	2	5	4	1	
15:	6	3	4	5	1	
16:	6	4	5	2	1	
17:	6	5	2	3	1	
18:	6	2	3	4	1	
19:	6	3	2	5	1	
20:	6	4	3	2	1	
21:	6	5	4	3	2	1
22:	6	2	5	4	3	1
23:	6	3	4	5	2	1
24:	6	4	5	2	3	1
25:	6	5	2	3	4	1
26:	6	2	3	4	5	1
27:	6	3	2	5	4	1
28:	6	4	3	2	5	1



Source face no.: 1
Goal face no.: 6
Source point coords.: 0.000 0.250 0.250
No. of developments: 28

(e)

Wm. Randolph Franklin

Wm. Randolph Franklin is an associate professor with the Electrical, Computer, and Systems Engineering Department at Rensselaer Polytechnic Institute. During 1985-86 he was on sabbatical at the University of California at Berkeley. His research interests include graphics, geometry algorithms, and artificial intelligence.

Franklin received the BSc in computer science from the University of Toronto in 1973, and the AM and PhD in applied mathematics (computer science) from Harvard in 1975 and 1978. He is a member of ACM, IEEE, and SIAM.

Franklin's electronic address is franklin@csv.rpi.edu.

Varol Akman

Varol Akman is a researcher in the Interactive Systems Department of Center for Mathematics and Computer Science (CWI), Amsterdam. His current research is concentrated on naive physics, and design as an intellectual activity. His other research interests include geometry and graphics programming. For the 1985-86 academic year he was a guest researcher with the Department of Computer Science at the University of Utrecht.

Akman received the BSc (1979, honors) in electrical engineering and the MSc (1980, honors) in computer engineering from the Middle East Technical University, Ankara. From 1980 to '85 he was a Fulbright scholar and a research assistant at Rensselaer Polytechnic Institute, New York — learning computer graphics at the Center for Interactive Computer Graphics, and later at the Image Processing Laboratory — and received the PhD in computer and systems engineering. His thesis has recently been published by Springer-Verlag in *Lecture Notes in Computer Science* series. Akman is a member of ACM, IEEE, and SIAM.

Akman's electronic address is varol@cwi.nl (on USENET).

SHORTEST RECTILINEAR PATHS AMONG OBSTACLES

Joseph S. B. Mitchell*

School of Operations Research and Industrial Engineering

Cornell University

Ithaca, NY 14853

Abstract

We present an algorithm for determining the shortest path according to the L_1 (L_∞) metric between a source and a destination in the presence of disjoint rectilinear obstacles in the plane. Our algorithm runs in time $O(g(n)\log n)$ and requires $O(g(n))$ space, where n is the number of vertices of the obstacles and $g(n)$ is the maximum number of ones allowable in a certain sparse binary matrix. We show that $g(n) = O(n\frac{\log n}{\log\log n})$ and we conjecture that $g(n)$ is actually linear, which would imply our algorithm to be optimal. Our bounds are the first subquadratic bounds for finding shortest obstacle-free paths which are not necessarily monotone. We build a planar subdivision (a Shortest Path Map) such that, after the initial running of our algorithm, the length of the shortest path from the source to a query point avoiding all the obstacles can be reported in time $O(\log n)$ by locating the query point in the subdivision. The actual shortest path from the source to a query point can be reported in time $O(k + \log n)$, where k is the number of "turns" in the path. The algorithm uses a "wavefront" propagation technique which runs much in the spirit of Dijkstra's algorithm. The algorithm can be generalized to the case of arbitrary simple polygons and to the case of fixed orientation metrics. 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 rectilinear obstacles in time $O(g(N)\log N)$, where N is the maximum of the number of sources and the number of obstacle vertices. We also describe a simple polynomial-time algorithm which, for fixed number of dimensions, finds shortest rectilinear paths among a set of orthohedral obstacles.

Mitchell is also with the Hughes Artificial Intelligence Center, Hughes Aircraft Company.

1. Introduction

Recently, much work has been done on the two-dimensional *Euclidean* (L_2) shortest path problem with polygonal obstacles. This problem can be solved easily in $O(n^2 \log n)$ time by constructing the *visibility graph* [Le, LW, Mi, SS], and $O(n \log n)$ is attainable in special cases when shortest paths possess certain monotonicity properties [LP, Mi, SS] (here, n is the total number of vertices in the polygonal obstacles). The visibility graph construction problem has been solved in optimal time $O(n^2)$ by [We] and [AAGHI]. Also, an algorithm for the two-dimensional Euclidean shortest path problem which runs in time $O(nm + n \log n)$ (where m is the number of disjoint simple polygonal obstacles) has been announced [RS]. This new algorithm solves the *single-source* shortest path problem by constructing a triangulation (analogous to the *Shortest Path Map* that we build) in which queries asking for the shortest distance from the source to any goal point can be answered in time $O(\log n)$ (and paths can be backtraced in $O(k)$ time, where k is the number of vertices along the shortest path). It remains an open problem to devise a subquadratic time algorithm for finding shortest Euclidean paths in nontrivial cases in which the optimal paths are not known to be monotone.

One version of the two-dimensional shortest path problem which is of considerable interest in the design and layout of circuits is the following: Determine the shortest route from source point s to destination point t in the plane for a wire which must be routed so it is always parallel to the x- or y-axis and which is not allowed to intersect the interiors of any of a given set of aligned rectilinear obstacles. An aligned rectilinear obstacle is a simple polygon such that every edge is parallel to either the x- or y-axis. We let $\mathcal{O}$ be the obstacle space consisting of the interiors of all of the given rectilinear polygons, $\mathcal{V}$ be the set of vertices of all obstacles, and we measure space and time complexity in terms of $n = |\mathcal{V}|$. The constraint that the wire always be parallel to a coordinate axis implies that the metric to be used in this application is the L_1 metric.

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 this paper we provide a solution to the general problem (with polygonal obstacles) which runs in time $O(g(n) \log n)$ and space $O(g(n))$, where $g(n)$ is the number of "events" in our algorithm and is related to the maximum number of ones in a binary matrix which obeys certain sparsity conditions. Using a recent result of Bienstock and Győri [BG], we show that $g(n) = O(n \frac{\log n}{\log \log n})$, so that the running time of our algorithm is $O(n \frac{\log^2 n}{\log \log n})$, nearly achieving the known lower bound and considerably improving the previous quadratic bound. We strongly suspect that $g(n) = O(n)$. If our conjecture is true, then the algorithm we present here is actually *optimal*, running in time $\theta(n \log n)$.

As in [DLW], our algorithm actually solves the query form of the problem: Given a source point s and a set of obstacles with n vertices, we build a structure (a *Shortest Path Map*, as originally defined in [LP]) such that queries which ask for the shortest path (resp., length of the shortest path) from s to a query point t may be answered quickly in time $O(k + \log n)$ (resp., $O(\log n)$), where k is the number of "turns" of the shortest path.

Our algorithm uses a technique which has been called a "continuous Dijkstra algorithm" [Mi, MMP, MP]. It basically considers the effects of propagating a "wavefront" from a source point s to all points in the plane. (The *wavefront rooted at s at distance D* can be defined formally as the boundary of the set of nonobstacle points which can be reached from s along a path of length D or less.) In doing so, we must update a structure at each of $g(n)$ *events*. Thanks to the ingenious data structures of Chazelle [Ch1,Ch2], we are able to update in $O(\log n)$ time after each event, while using only linear storage. The approach is very similar to the familiar line sweep technique of computational geometry (see [PS]); it could in fact be called a *wavefront sweep* technique.

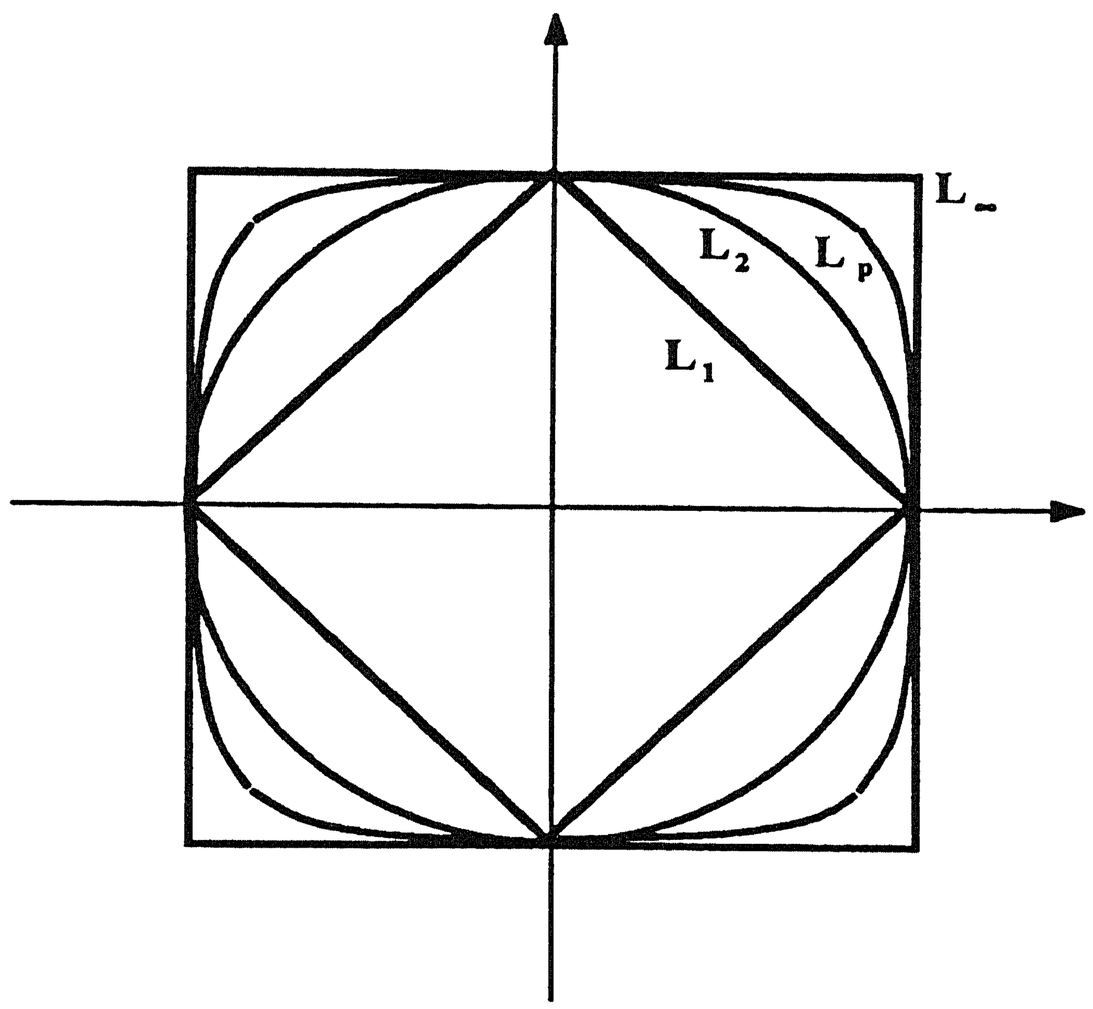


Figure 1. Circles in different metrics.

This continuous Dijkstra approach is analogous to tracking the wavefront of the disturbance after a pebble has been dropped in a pond with "island" obstacles. Such was the approach used in [MMP] to solve the discrete geodesic problem on the surface of an arbitrary three-dimensional polyhedron. In Euclidean metric problems, the initial wavefront has the shape of a circle. In the case of the L_1 metric, "circles" are of the shape of a diamond. (For L_∞, they are the shape of squares; and for L_p with $1 < p < \infty$, they are "in

between".) See Figure 1. The special property of the L_1 norm which makes it easy to track the propagating wavefront is the fact that the wavefront is always composed of straight line segments that are oriented at 45 degrees. Determining events in the continuous Dijkstra framework involves answering *segment dragging queries* of the forms to be discussed in Section 3.

The proof of the upper bound on the number ($g(n)$) of events uses a charging scheme which leads to the study of the maximum number of ones that can appear in a binary matrix which is not allowed to contain certain "violated patterns". We get a crude bound of $g(n) = O(n^{1.5})$ by looking at "rectangle-free" matrices. Our research points to a very interesting class of problems involving the study of matrices which are sparse due to a variety of violated patterns. Since the suggestion of this class of problems, there has already been progress, including a recent proof by [BG] that the maximum number of ones that can appear in a binary matrix which has no "L-quadrilaterals" (see Section 7) is $G(n) = \Theta(n\frac{\log n}{\log\log n})$. We use this result to show that $g(n) = O(G(n))$, showing that our algorithm is almost optimal. Hopefully, further investigations into the density of the relevant "sparse matrix" that arises from our algorithm will lead to a proof that the algorithm is indeed optimal.

Some applications have need of shortest paths according to the L_∞ metric. We simultaneously solve this problem when we solve the L_1 case, for we need only appeal to the isometry that exists between the L_1 and the L_∞ metrics (see [LWo]). In particular, there is a linear mapping f from the plane with L_1 metric to the plane with L_∞ metric defined as follows: $f(x,y) = ((x+y)/2, (y-x)/2)$. Both f and f^{-1} are isometries; thus, problems in the L_∞ metric can be transformed via f^{-1} to problems in the L_1 metric. We therefore consider only the case of the L_1 metric.

Section 9 will discuss further generalizations of our algorithm to the case of fixed orientation metrics [WWW] and to multiple source points. The bottom line is that our algorithm yields an almost optimal algorithm for computing the Voronoi diagram according to any fixed orientation metric (L_1 and L_∞ are special cases of fixed orientation metrics) of a set of sites in a space which is cluttered by polygonal obstacles (the obstacles need not be rectilinear).

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", which will be discussed in Section 9. 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) aligned rectilinear polygons and that $\mathcal{V}$ ($n = |\mathcal{V}|$) is the set of obstacle vertices. While our algorithm will work for the more general case of simple polygonal obstacles, we defer discussion of the generalization to Section 9, and concentrate our discussion

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

Our problem can now formally be stated as follows.

(RSP) RECTILINEAR SHORTEST PATH PROBLEM

Instance: *Two points s and t in the plane and a collection of disjoint rectilinear obstacles with sides parallel to the coodinate axes.*

Question: *Find a shortest path in the L_1 metric from s to t not intersecting any of the obstacles.*

The problem we actually solve is the query form of the RSP, and from this solution we can solve the RSP within the same time bound.

(SSRSP) SINGLE-SOURCE RECTILINEAR SHORTEST PATH PROBLEM

Instance: *One source point s in the plane and a collection of disjoint rectilinear obstacles with sides parallel to the coodinate axes.*

Question: *Build a structure which allows one to compute a shortest path (in the L_1 metric) from s to any query point t which does not intersect any of the obstacles.*

The structure that is built to answer the question in the SSRSP problem is called a *Shortest Path Map* [LP, Mi]. It 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]). More will be said about this in Section 4.

The advantage of solving SSRSP instead of just RSP is that if we wish to change destination points (while keeping s fixed), we can find the new shortest path very efficiently. Also, the SSRSP is related to the *multiple-source* problem, in which we are asked to compute a Voronoi diagram for a set of many sources in the presence of a set of obstacles. It turns out that our algorithm will solve the multiple-source problem efficiently as well. We defer the discussion to Section 9, and concentrate now on the single-source problem.

The fact that we can solve SSRSP in the same time as we can solve RSP (and we know of no algorithm that solves RSP faster than SSRSP) is reminiscent of the fact that nobody has discovered a shortest path algorithm for graphs which takes any less time (worst-case, asymptotic) to compute the shortest path length from s to t than it does to compute the shortest path length from s to every other node.

We define the *rectilinear grid* on a set of points $\{q_1, \ldots, q_n\}$ in the plane to be the graph obtained by drawing the set of all lines through the q_i's which are parallel to the x- or y-axes and placing nodes at every crossing point of two lines (and defining edges to be the line segments between consecutive nodes on a line).

See Figure 2. Our first observation is the rather obvious fact that there will be shortest paths in the L_1 norm around a given set of rectilinear obstacles which lie on a grid defined by the set of all vertices of the obstacles.

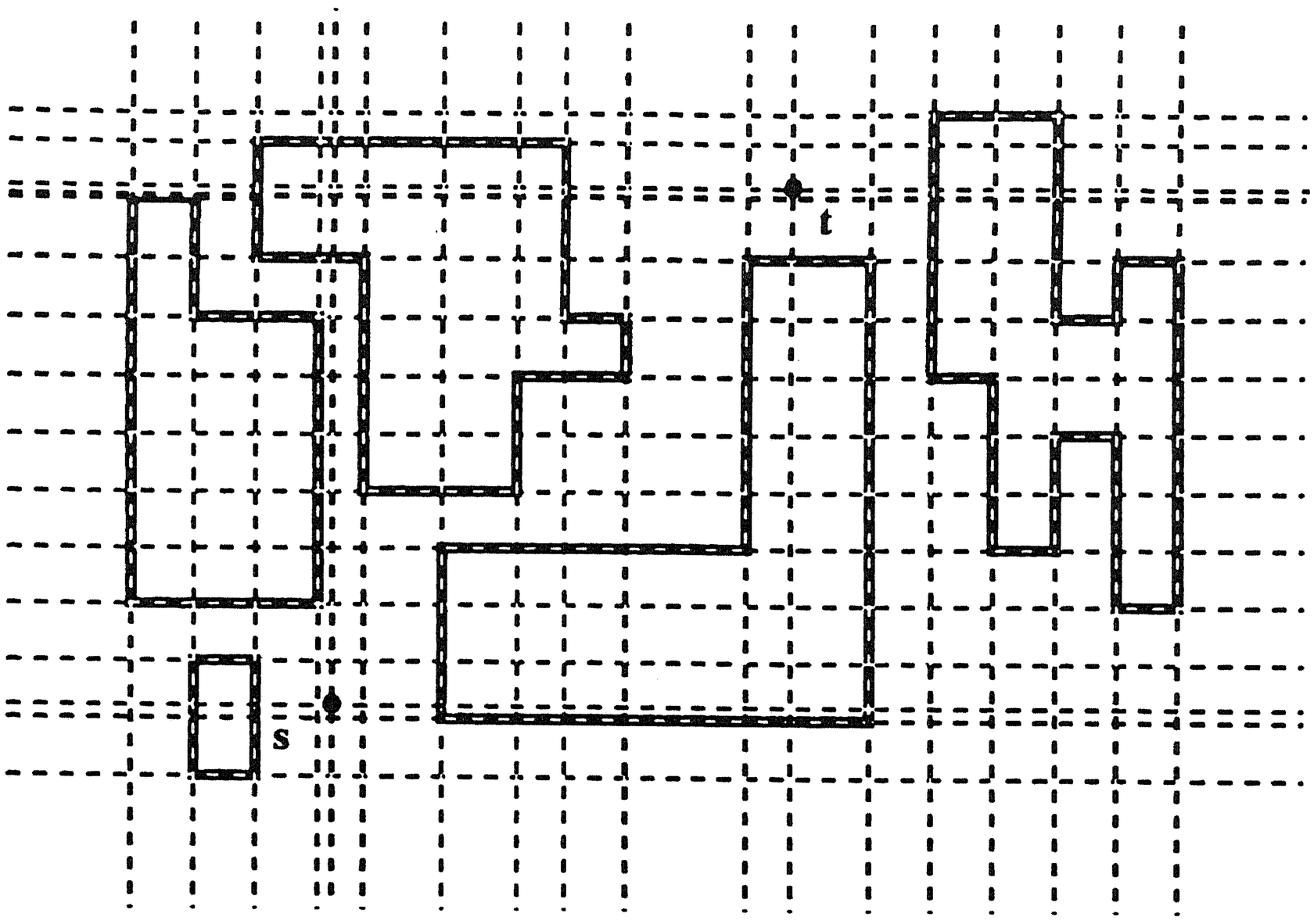


Figure 2. The grid graph defined by a set of obstacles.

Lemma 2.1 *If there exists a path from s to t which avoids a given set of rectilinear obstacles, then there exists an obstacle-avoiding shortest path which lies entirely on the grid defined by the vertices of the obstacles and s and t.*

Using the observation of Lemma 2.1, it is easy to arrive at an algorithm to find shortest paths. One first constructs the relevant part of the grid graph (that which is not interior to any obstacle), and then one runs Dijkstra's algorithm on the resulting graph. The grid will have $O(n^2)$ nodes and edges, and it can be constructed in time $O(n^2)$. Dijkstra's algorithm can then be run in time $O(n^2 \log n)$, yielding an algorithm for shortest paths with total running time $O(n^2 \log n)$ and space $O(n^2)$.

There is actually a sparser (although still $O(n^2)$ in size) grid graph which works as the basis for shortest paths. Imagine firing a bullet from every vertex (and s and t) in each of the four directions (north, south, east, and west). Bullets get stopped by obstacles. The graph whose nodes are the crossing points of bullet paths and whose edges connect consecutive nodes along the paths is a subgraph of the grid used in Lemma 2.1; however, it is still sufficiently complete to include a shortest path from s to t, should one exist. See Figure 3.

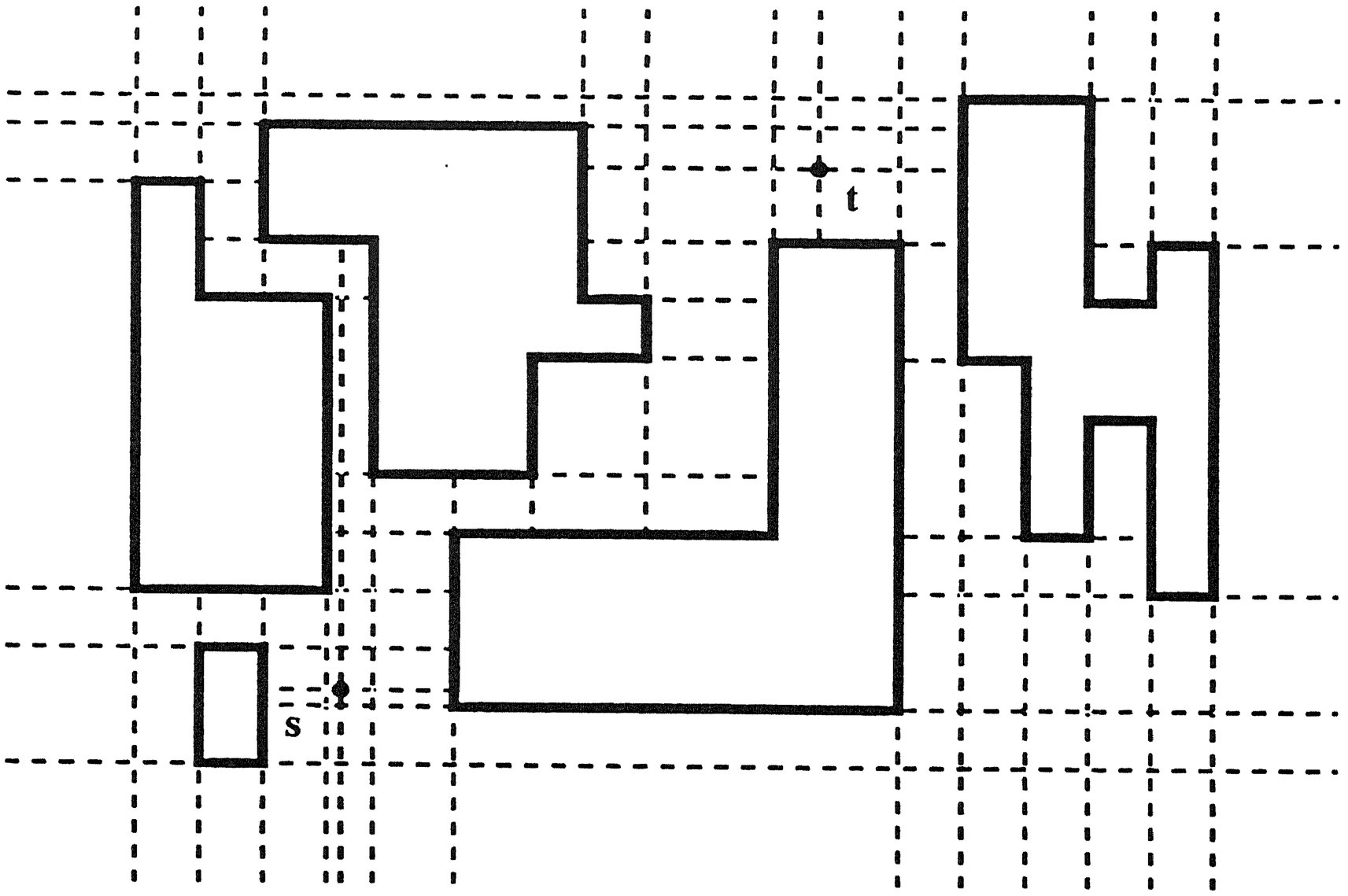


Figure 3. A simple grid graph defined by a set of obstacles.

We shall call the resulting graph a *simple grid graph*, and we note that it can be constructed in time $O(n^2)$: The bullet paths can be determined in time $O(n \log n)$ (this can be done in $O(\log n)$ time per vertex by using the *next-element* subdivision of [EOS] or the Horizontal or Vertical Adjacency Map of [PS]), while the determination of all crossings of bullet paths can be done in $O(n^2)$.

Lemma 2.2 *If there exists a path from s to t which avoids a given set of rectilinear obstacles, then there exists an obstacle-avoiding shortest path which lies entirely on the* simple *grid defined by the vertices of the obstacles and s and t.*

We can use the simple grid graph to yield a slightly better algorithm. Note that the simple grid graph may still have a quadratic number of nodes (at the crossing points of bullet paths), which will again lead to an $O(n^2 \log n)$ algorithm if we apply Dijkstra to it directly. But we can use the simple grid graph to define a new graph, a type of "visibility graph", which will have only a linear number of nodes (just those at the vertices of the original obstacle space). First, we say that vertex v is *visible from* vertex v' if a bullet path out of v either hits v' or intersects a bullet path out of v'. (This is similar to the notion of *simply communicating vertices* in [LL]). If two vertices v and v' are visible from one another, then there exists an obstacle-free path between them of length $d(v, v')$. We now build a graph whose nodes are the obstacle vertices (and s and t), and we connect two nodes by an edge if the corresponding vertices are visible from one another. The length of an edge is just the (L_1) distance between the vertices. The resulting graph has $O(n^2)$ edges and $O(n)$

nodes, so Dijkstra can be run to find the shortest path from s to t in time $O(n^2)$ (and space $O(n^2)$). This is similar to the technique of [LL], where these same complexity bounds are achieved for our problem.

One approach to improving the running time of this shortest path algorithm is to look for a sparser subgraph of the grid graph which is guaranteed to contain shortest paths. If, for example, we could show that a shortest path will always exist in a very sparse subgraph of the grid graph, then we could potentially improve these running times. If the sparse subgraph corresponds to only a linear number of crossing points, then this approach would in fact yield an optimal algorithm. (More generally, if one could show that adding $f(n)$ crossings (other than the ones that occur at obstacle vertices) suffices, then one would have an $O((n+f(n))\log(n+f(n)))$ algorithm for finding shortest paths.) We do not pursue this approach here, but rather look at the application of the continuous Dijkstra paradigm to this problem. Our approach yields a computationally simple and elegant algorithm, and improves the known bounds to $O(g(n)\log n)$ time and $O(g(n))$ space (where $g(n)$ is shown to be $O(n\frac{\log n}{\log\log n})$ and is conjectured to be $O(n)$). We therefore achieve an almost-optimal algorithm.

> *Remark*: Note that the grid graph approach as we have described it solves only the RSP problem, and does not explicitly solve the SSRSP problem. However, it is possible to use the grid graph approach to build a shortest path *tree* from a source point s so that the shortest path from s to any obstacle vertex can be found efficiently (by simply backtracing a path through the shortest path tree). It is further possible to build a shortest path map from the shortest path tree very efficiently (in $O(n\log n)$ time).

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 segments 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 4(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

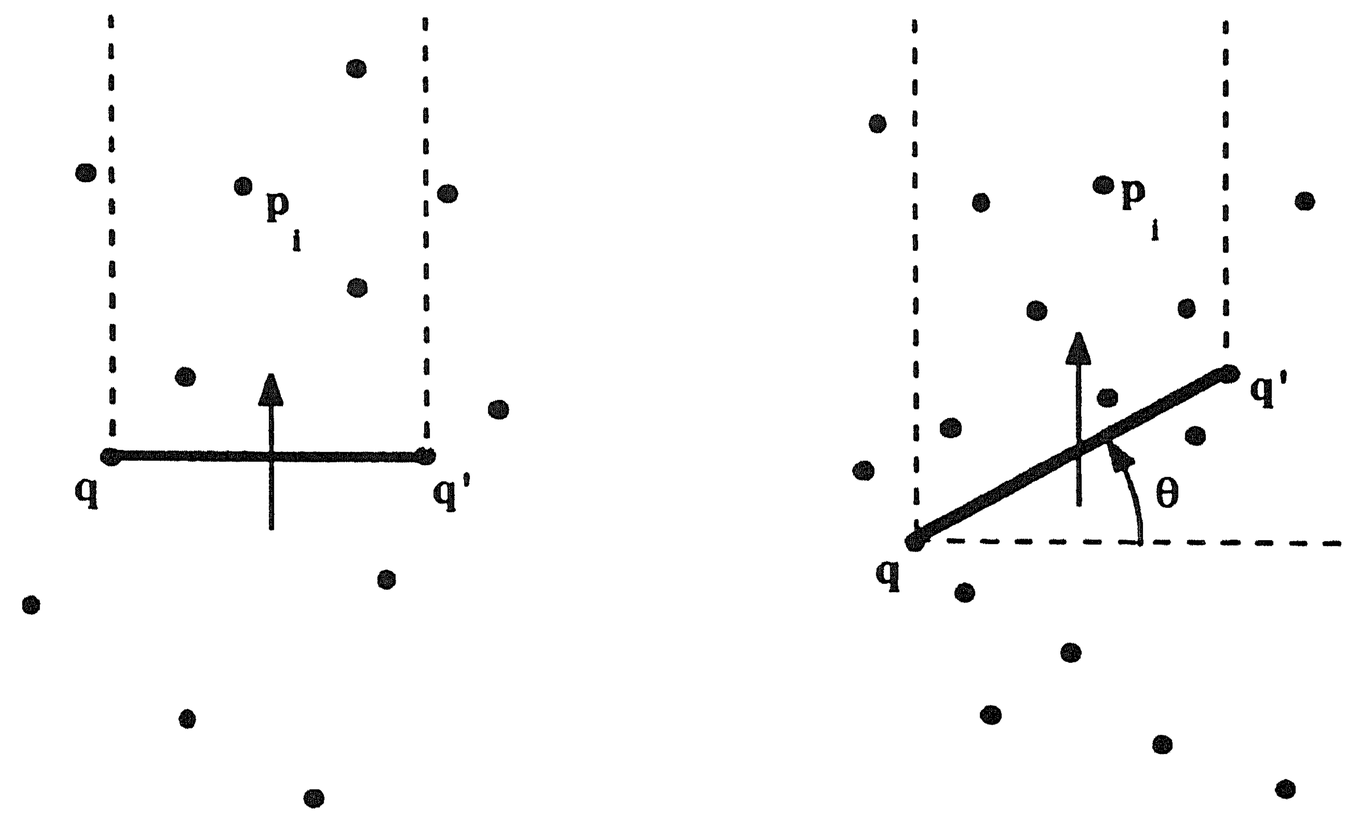


Figure 4. (a) Dragging a horizontal segment. (b) Dragging an inclined segment

needed to answer our segment dragging problem, however, Chazelle is able to achieve query times of $O(\log n)$ with a space complexity of $O(n)$ [Ch3]. (The technique uses the linear-size data structure Chazelle presented in [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 4(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 5. 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

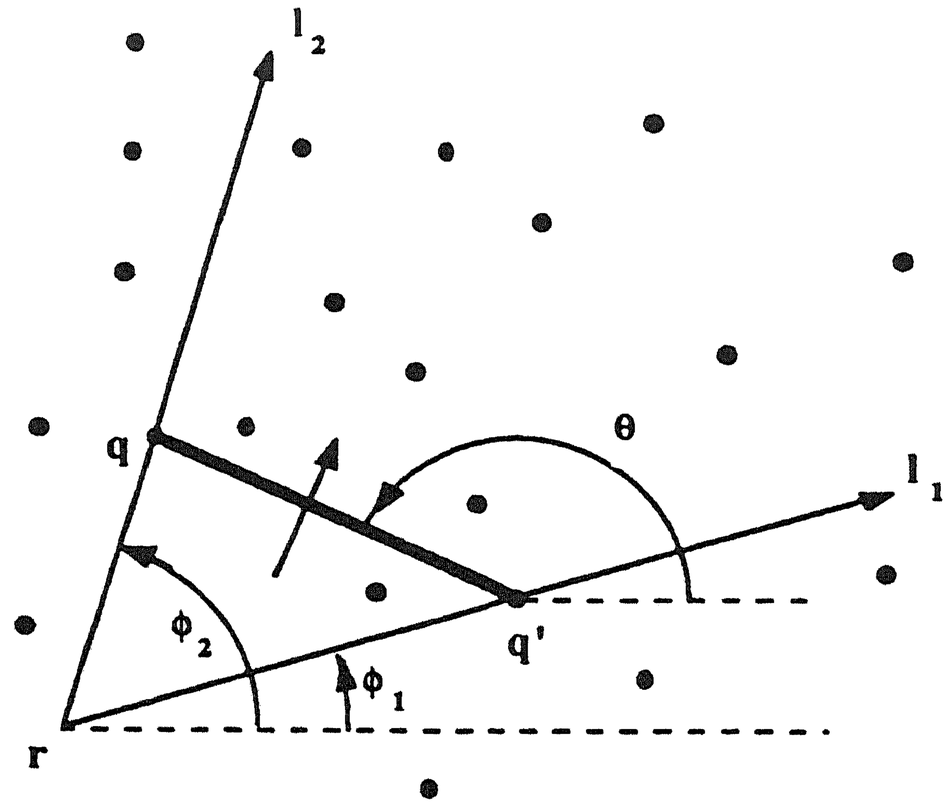


Figure 5. Dragging a segment along two rays.

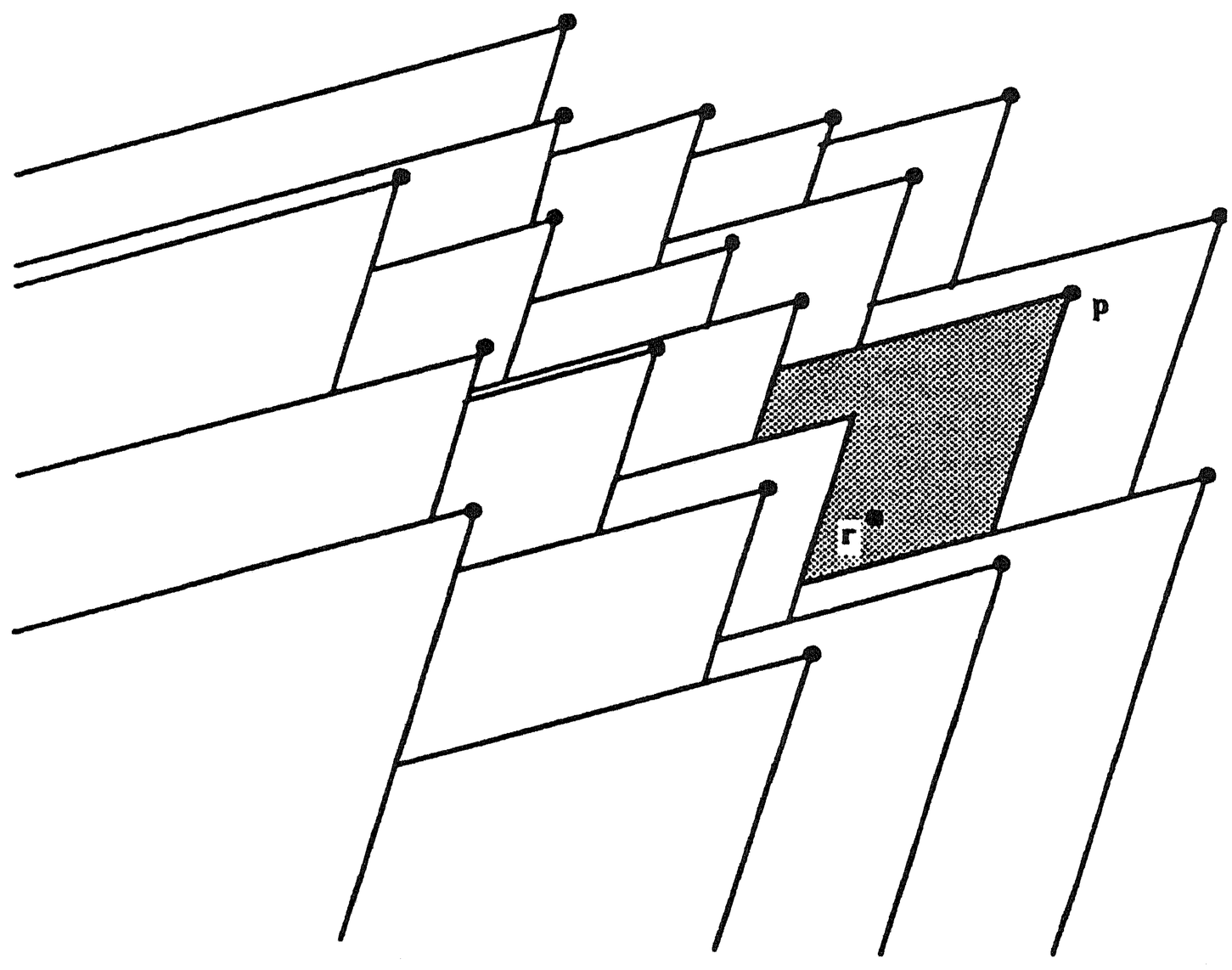


Figure 6. Subdivision $S(\theta, \phi_1, \phi_2)$.

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 6. Once we have this subdivision, if we locate r (in time $O(\log n)$, [Ki, Pr]) in, say, the shaded region of Figure 6, 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 $S(\pi/2, 0, \pi/4)$).

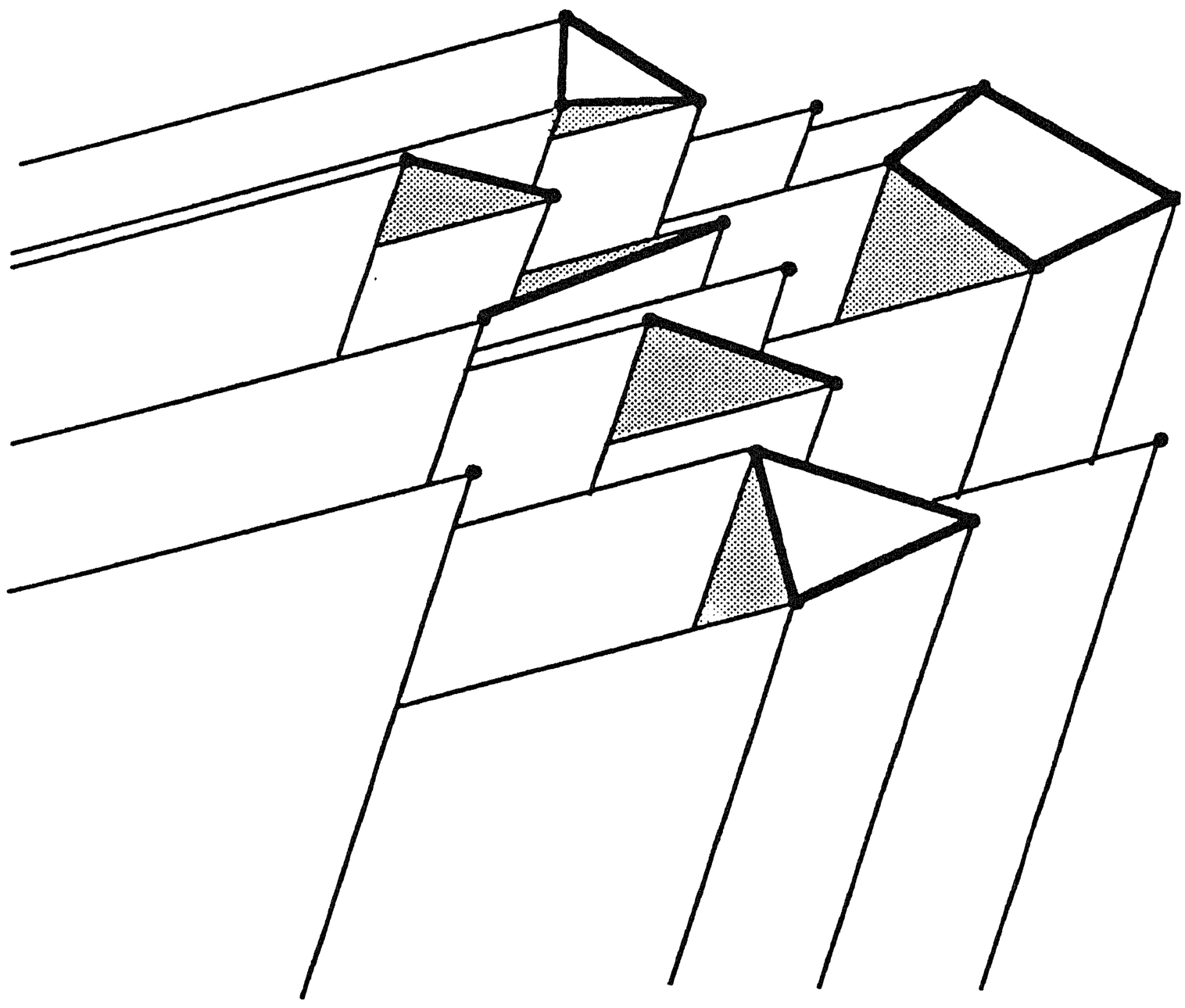


Figure 7. Subdivision $S(\theta, \phi_1, \phi_2)$ in the presence of segments and points.

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 and line segments (instead of just points $\{p_1, \ldots, p_n\}$). The segments may be edges of simple polygons (polygonal obstacles in our applications). 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 segments of the original set. See Figure 7. Note that if r is located in one of the shaded regions of Figure 7, 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 point. (The obstacles have a "shadowing" effect.)

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].

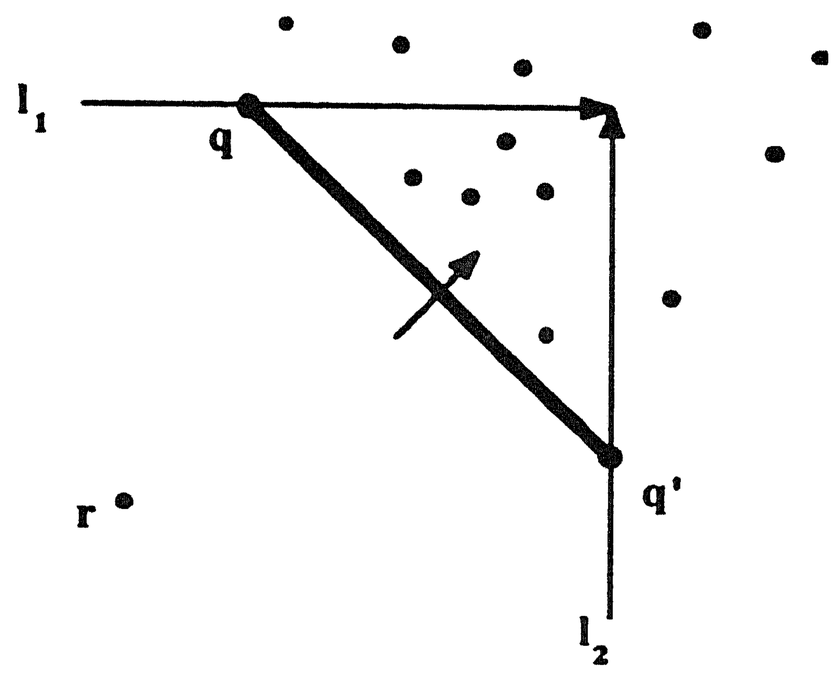


Figure 8. Dragging a segment into a corner.

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).

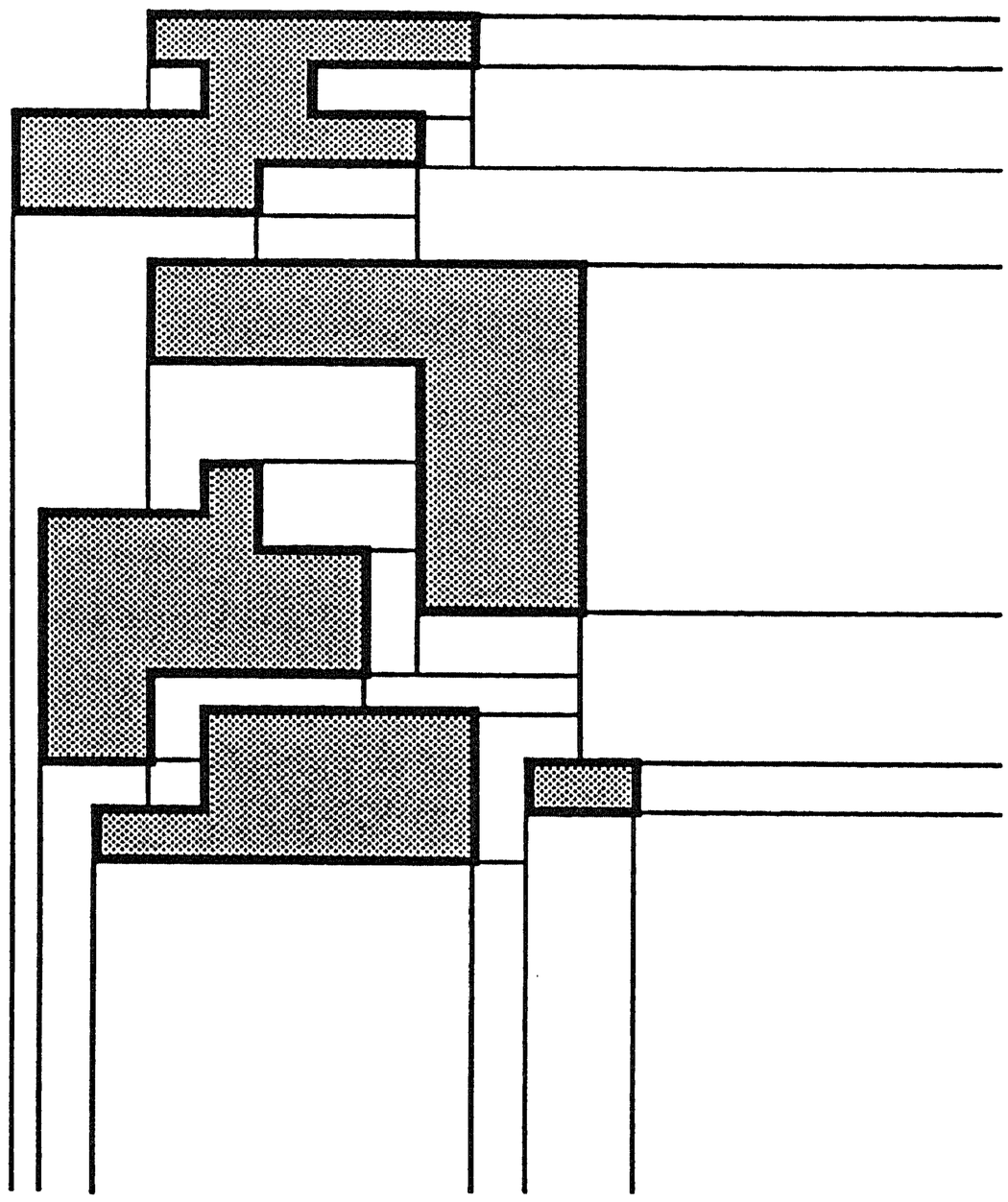


Figure 9. Subdivision $S(5\pi/4, \pi/2, \pi)$ in the presence of rectilinear obstacles.

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 rectilinear obstacles. See Figure 9 for an example showing $S(5\pi/4, \pi/2, \pi)$.

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.

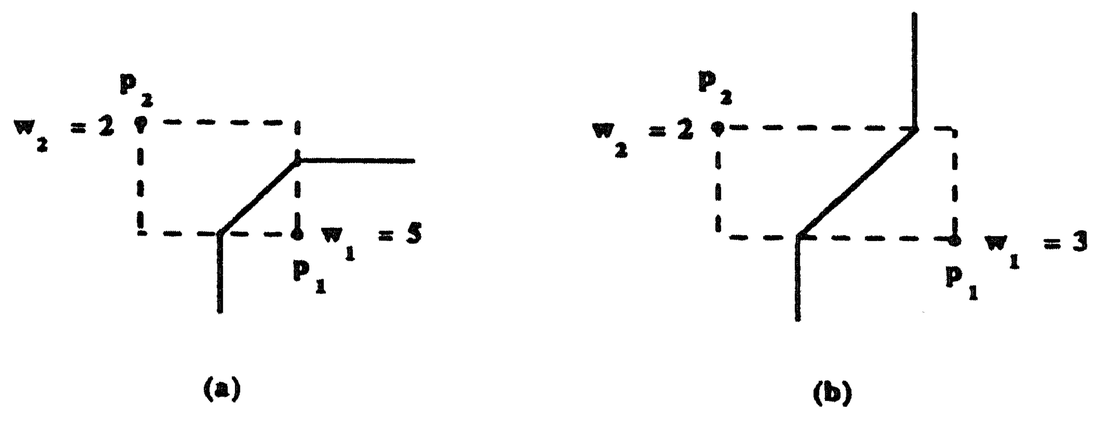


Figure 10. Bisectors.

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 $R(p, q) \cap \mathcal{O} = \emptyset$ (ie., the rectangle does not intersect any obstacle). Let r be any vertex of any obstacle. Then the set $H^{NW}(r)$ of all points p which are northwest of r and immediately accessible from r will be called the *northwest hull rooted*

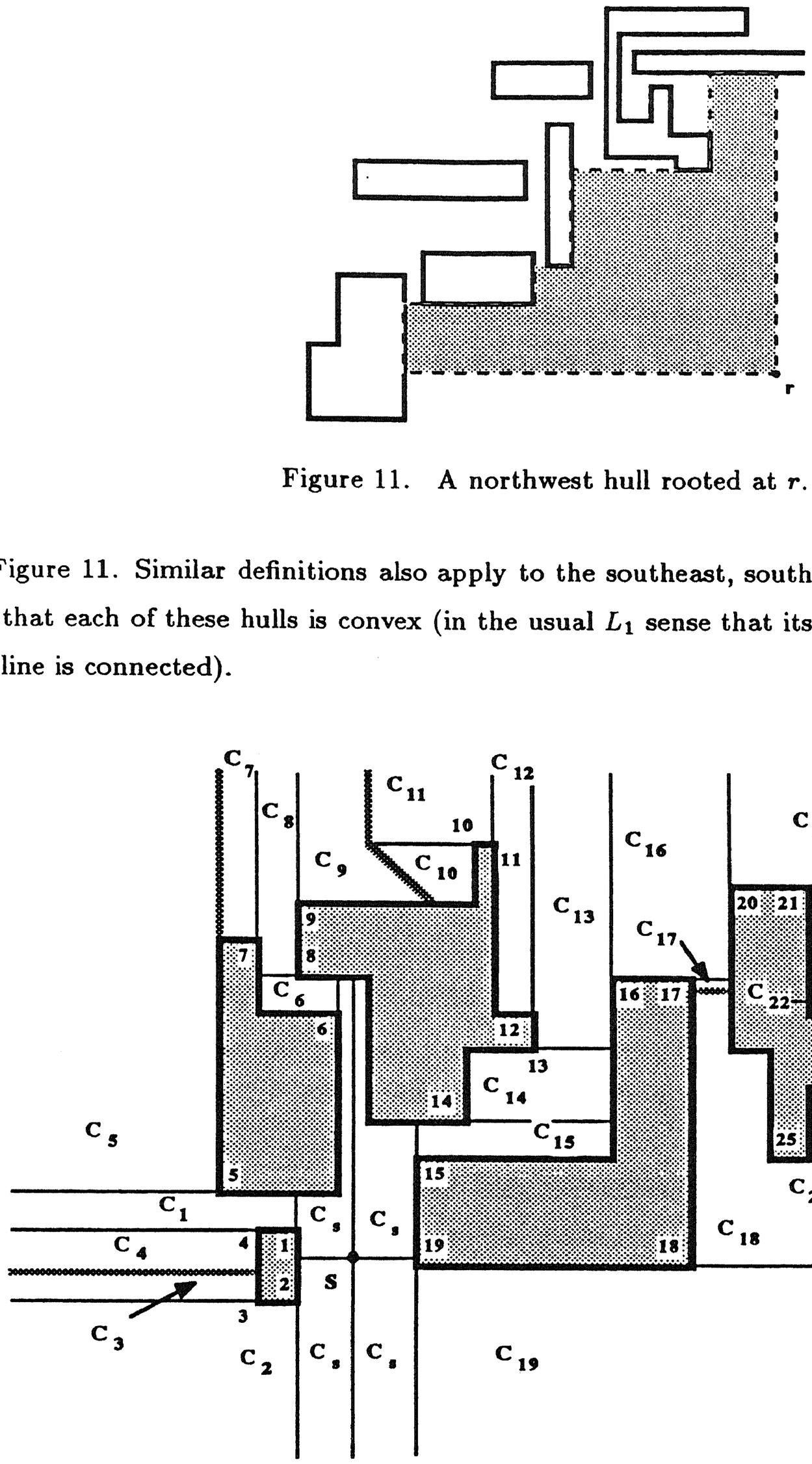


Figure 11. A northwest hull rooted at r.

at r. See Figure 11. Similar definitions also apply to the southeast, southwest, and northeast hulls rooted at r. Note that each of these hulls is convex (in the usual L_1 sense that its intersection with any horizontal or vertical line is connected).

Figure 12. A Shortest Path Map.

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

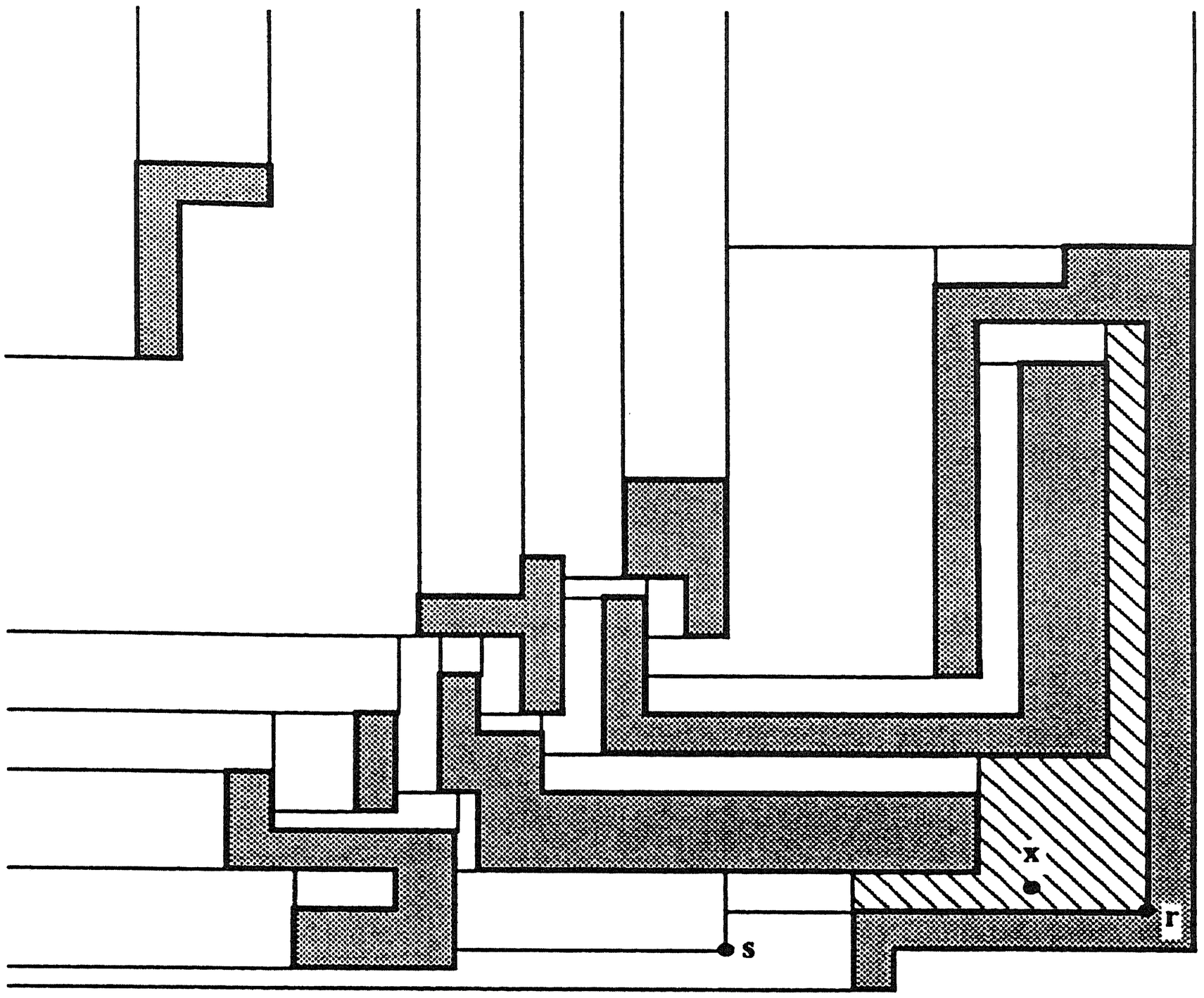


Figure 13. $\mathcal{S}^{SE}$.

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 the boundary of $R(r, t)$). 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 12 for an example of an SPM. Note that the cell $C(r)$ will be a subset of one of the four hulls rooted at r.

Our algorithm actually constructs the SPM $\mathcal{S}$ by constructing four related subdivisions along the way. We let $\mathcal{S}^{SE}$ be the subdivision of the plane into cells $C^{SE}(r)$ such that $x \in C^{SE}(r)$ if and only if r is southeast of x and $d(x, r) + d(r) = \min\{d(x, r') + d(r') : r'$ is southeast of x and immediately accessible

from x}. (Ties can be broken according to some lexicographic ordering of the roots that are the argmin's of the above expression.) See Figure 13 for an example, where the cell $C^{SE}(r)$ which contains point x is shown shaded. Note that each cell $C^{SE}(r)$ is a rectilinear polygon whose edges lie on the grid implied by the obstacle vertices and that each cell $C^{SE}(r)$ will be a subset of the northwest hull $H^{NW}(r)$. Note also that the shortest path to x may not be to go through root r at all. (Indeed, in the figure it is clearly optimal to get to x along a direct rectilinear route from s.) Similar definitions apply to $\mathcal{S}^{NE}$, $\mathcal{S}^{NW}$, and $\mathcal{S}^{SW}$. Using these four subdivisions, a shortest path to any point t may be determined by locating t in each subdivision, since a shortest path to t will go through a root which must be in one of the four directions from t. If we wish, these subdivisions can be merged into a single SPM $\mathcal{S}$ (and this merge step can easily be done within the time bounds of our algorithm), or we can just keep all four subdivisions and do four point location queries for each destination point t.

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 (establishing a wavefront from p in each of the four directions: SE, SW, NE, NW). 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 obstacle vertex has received the signal from each of the four directions. 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 $O(n\frac{\log n}{\log\log n})$ and that the update time per event is only $O(\log n)$.

By considering wavefronts separately in each of the four directions, we have decomposed the problem in much the same way as was done in [MMP] for the problem of computing discrete geodesics, where points on an edge were "hit" independently by waves coming from each side of the edge (the resulting "f-free paths" allowed the use of *structured monotonicity* [Mo] to solve the problem). In the discrete geodesic problem, two subdivisions of each edge were built, according to the structure of the shortest paths to points on the edge, one subdivision corresponding to each of the two directions from which the paths may be incident to the edge. (Paths may hit an edge by passing through either of the two faces adjacent to the edge.) In the problem considered here, the shortest path to a point may come from one of four directions, so we build four subdivisions ($\mathcal{S}^\delta$). The advantage of decomposing our problem into the four subproblems is that it makes the "clipping" of wavefronts easier (refer to Figure 18 in Section 6). 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.

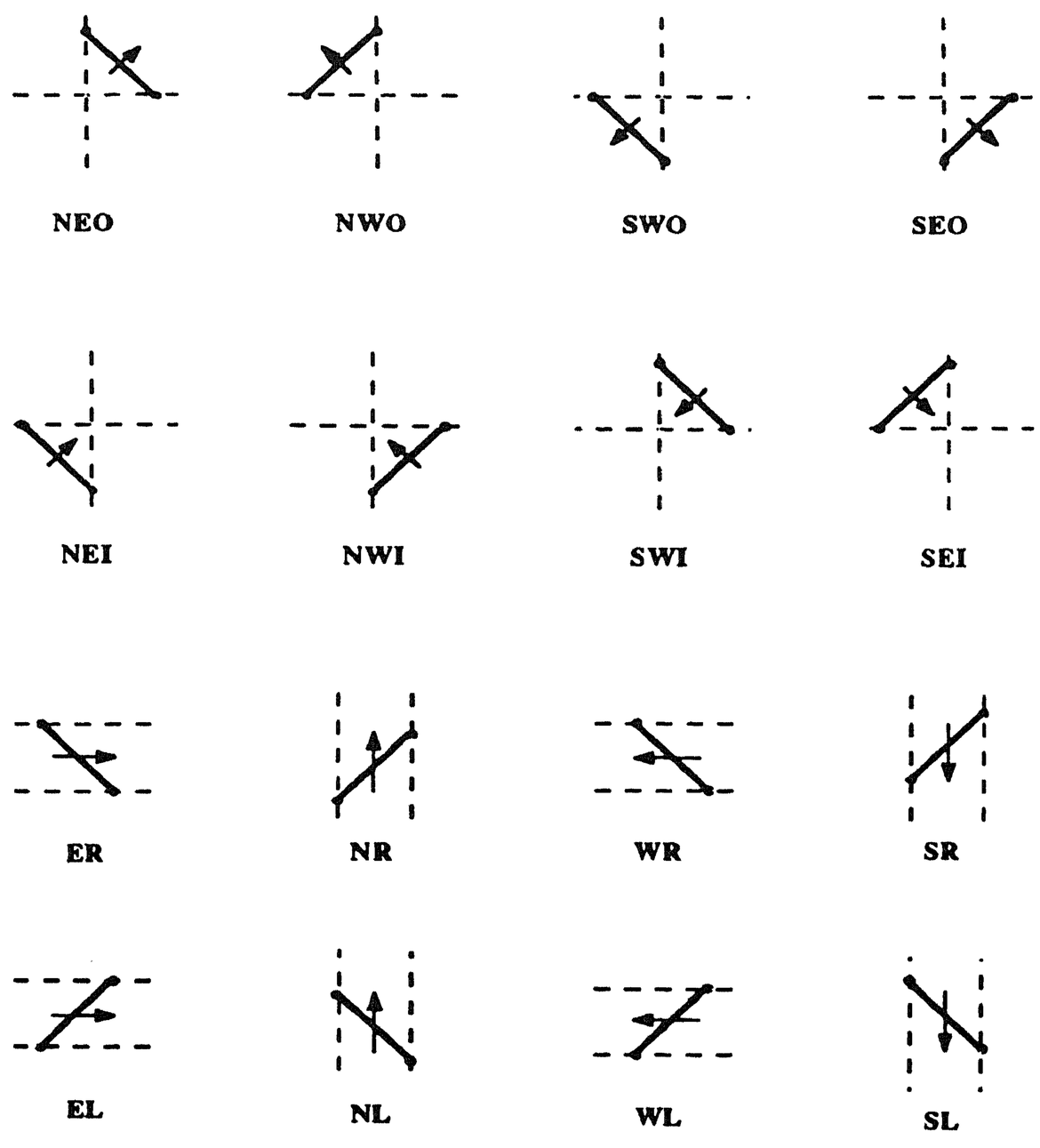


Figure 14. Sixteen cases of dragged segments.

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 NEO, NWO, SWO, and SEO. 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 which has a finite label, remove the one, $\overline{qq'}$, with the smallest label and do the procedure Propagate $(\overline{qq'})$.*

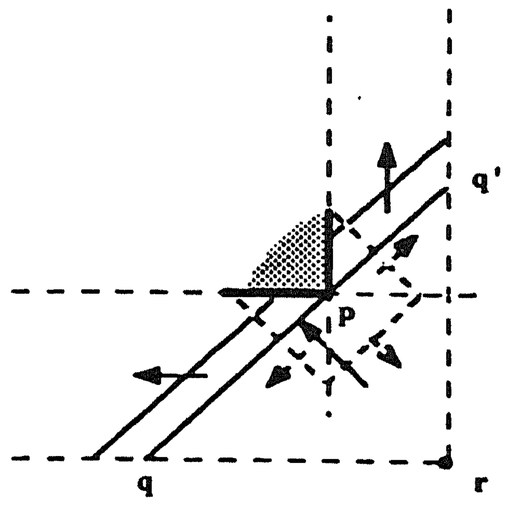
The details of the algorithm are contained in the procedure *Propagate*. Intuitively, to *propagate* a dragged segment $\overline{qq'}$ means to allow a "wavefront" of signals to advance past the event point p in the direction that $\overline{qq'}$ is being dragged. This usually involves creating new 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 15, 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.

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

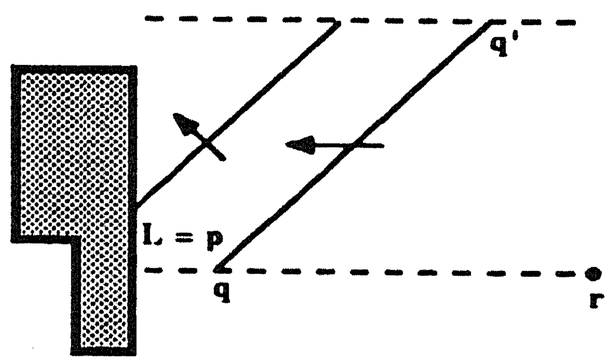
0). *Let $p = (x_p, y_p)$ be the event point, let $r = (x_r, y_r)$ be the root, and let L and R be the left and right stop points of $\overline{qq'}$. Assume (without loss of generality) that the segment $\overline{qq'}$ is inclined at angle $5\pi/4$ and propagating northwest (type NWO, NWI, NR, or WL), so that r is southeast of p.*

1). **(if p is a stop point)** *(Type II event) If $p = L$, then insert a dragged segment (of type NWI or NR) whose right track ray is that of $\overline{qq'}$ and whose left track ray is the northward ray through p. Otherwise ($p = R$), insert a dragged segment (of type NWI or WL) whose left track ray is that of $\overline{qq'}$ and whose right track ray is the westward ray through p. (Charge point p.)*



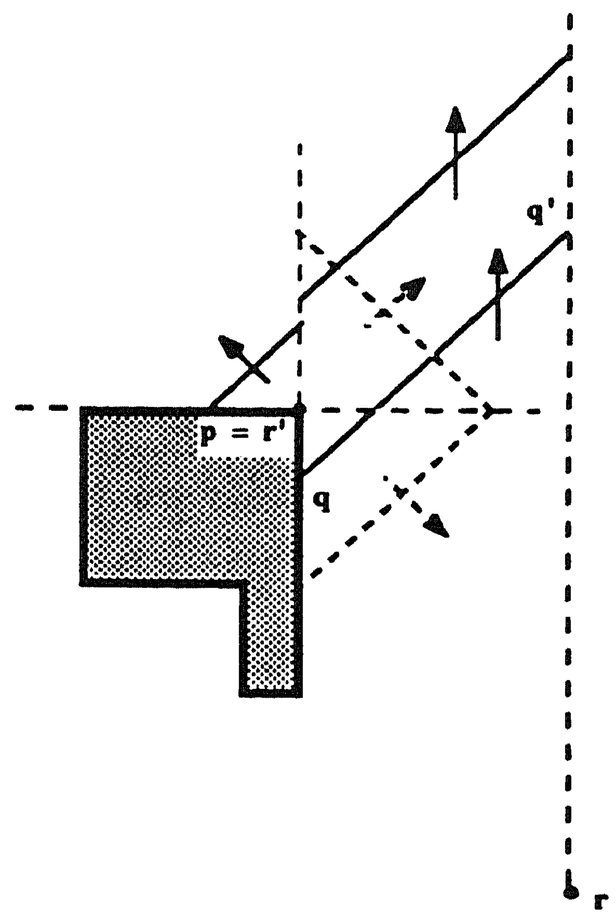
Type I : Interior Collision

NWO → WL, NR rooted at r
(SWO, SEO, NEO rooted at p)



Type II : Reaching the stop point

WL → NWI



Type III : Endpoint collision

NR → NR rooted at r
NWO rooted at r'
(SEO, NEO rooted at r')

Figure 15. Propagating a dragged segment.

2). (else if p already permanently labeled) *If $\mathcal{R}^{SE}(p) = \emptyset$, then go to (a) below. Otherwise, locate and insert r in the list $\mathcal{R}^{SE}(p) = \{r_1,\ldots,r_N\}$ by finding the points r_i and r_{i+1} $(0 \leq i \leq N)$ such that the y-coordinate of r lies between that of r_i and that of r_{i+1} (where we define the y-coordinate of r_0 to be $-\infty$ and that of r_{N+1} to be $+\infty$), and go to (b) below. Refer to Figure 16.*

(a). *(if $\mathcal{R}^{SE}(p) = \emptyset$) Set $\mathcal{R}^{SE}(p) = \{r\}$. Insert new segments according to (i) or (ii) below. (Charge point p.)*

(i). *(if $p = q_e$ or $p = q'_e$) (Type III event) Insert a dragged segment rooted at r of the same type as $\overline{qq'}$ but with intitial position $\overline{q_e q'_e}$. If $\overline{qq'}$ is of type NWO, then the next hit is determined by locating the point $r + \epsilon\hat{j} - \epsilon\hat{i}$ in the subdivision $S(5\pi/4, \pi/2, \pi)$.*

(ii). *(else p is interior to $\overline{q_e q'_e}$) (Type I event) Insert a dragged segment rooted at r (of type NWI or WL) whose left track ray is that of $\overline{qq'}$ and whose right track ray is the westward ray through p. Insert a dragged segment rooted at r (of type NWI or NR) whose*

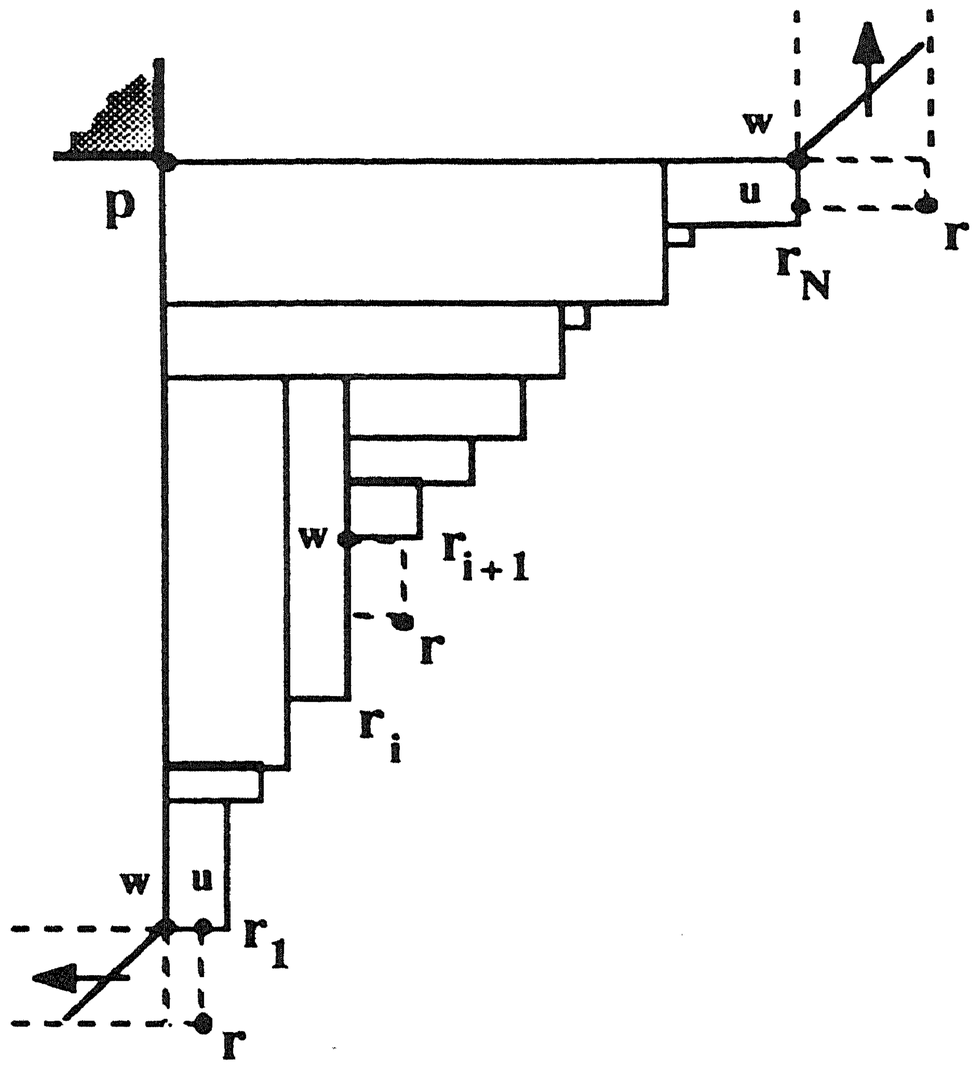


Figure 16. Clipping at p using $\mathcal{R}^{SE}(p)$.

right track ray is that of $\overline{qq'}$ *and whose left track ray is the northward ray through* p.

(b). (else if $i = 0$) *(Clip on the north) Insert a dragged segment rooted at* r *(of type NWI or WL) whose left track ray is that of* $\overline{qq'}$ *and whose right track ray is the westward ray through* r_1. *(Charge the upper right corner point of* $R(r,p)$ *(*(x_r, y_p)*) with a "North Clip (NC)".)*

(c). (else if $0 < i < N$) *Stop propagation from* r. *(Charge point* r *with a "Stop Propagation".)*

(d). (else $i = N$) *(Clip on the west) Insert a dragged segment rooted at* r *(of type NWI or NR) whose right track ray is that of* $\overline{qq'}$ *and whose left track ray is the northward ray through* r_N. *(Charge the lower left corner point of* $R(r,p)$ *(*(x_p, y_r)*) with a "West Clip (WC)".)*

3). (else p **not yet permanently labeled**) *Permanently label* p *with the distance* $d(p) = d(r) + d(r,p)$ *and set* $\text{pred}(p) = r$. *Set* $\mathcal{R}^{SE}(p) = \{r\}$. *Instantiate* p *as a new root by inserting dragged segments rooted at* p *of types NEO, SWO, and SEO (except those directions occupied by the obstacle containing* p*). Insert new segments according to (a) or (b) below. (Charge point* p.*)*

(a). (if $p = q_e$ or $p = q'_e$) *(Type III event) Insert a dragged segment rooted at* r *of the same type as* $\overline{qq'}$ *but with intitial position* $\overline{q_e q'_e}$. *If* $\overline{qq'}$ *is of type NWO, then the next hit is determined by locating the point* $r + \epsilon\hat{j} - \epsilon\hat{i}$ *in the subdivision* $S(5\pi/4, \pi/2, \pi)$.

(b). (else p **is interior to** $\overline{q_e q'_e}$) *(Type I event) Insert a dragged segment (of type NWI or WL)*

rooted at r whose left track ray is that of $\overline{qq'}$ and whose right track ray is the westward ray through p. Insert a dragged segment (of type NWI or NR) rooted at r whose right track ray is that of $\overline{qq'}$ and whose left track ray is the northward ray through p.

We have used the notation that $\hat{i} = (1,0)$ (resp., $\hat{j} = (0,1)$) is the unit vector in the x-direction (resp., y-direction). The interpretation of $r + \epsilon\hat{j} - \epsilon\hat{i}$ 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 $r + \epsilon\hat{j} - \epsilon\hat{i}$ 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 $+\infty$.

The specification of the procedure includes instructions to "charge" some point. Basically, we are specifying an "accounting scheme" which will keep track of the total number of events that can occur. In each case, *Propagate* charges some point of the grid graph for the collision which just took place. The analysis of our charging scheme will be discussed in more detail in Section 7.

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 subdivisions $\mathcal{S}^\delta$ 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 algorithm we have specified has made no assumptions about the rectilinear obstacles having to have special structure (such as being convex or rectangular). In fact, we will mention in Section 9 the extension of this algorithm to the case of arbitrary simple polygons. Rather than using special structure of shortest paths (such as monotonicity, in the rectangular obstacle case [DLW]), our wavefront propagation technique

relies on sweeping out all of the obstacle-free space, clipping wavefronts accordingly when a vertex is rehit.

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-NWI* ($\overline{qq'}$)

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. (If q' lies on an obstacle, w is found by locating q in subdivision $S(\pi/4, \pi/4, \pi/2)$.) 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 NWI event point. Otherwise, if q_e is north of c, then stop (no obstacle vertex is hit by a NWI 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 $\Delta qcq'$. 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

When our algorithm does propagation, it relies on doing segment dragging queries to determine the closest obstacle point in a particular direction under certain constraints. Segment queries which drag a segment out of a corner are based on 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 $S(\pi/4, 3\pi/2, 2\pi)$, and these subdivisions include the effect of obstacles. However, when we perform segment dragging queries along parallel rays (such as types NR and WL), we use range query techniques which do not explicitly take into account the presence of obstacles. That is, the query is answered according to what the next *vertex* hit will be. The effect of obstacles is to "shadow" vertices, potentially blocking them from

being hit first. Just because a certain vertex was the first to be hit does not imply that it was the first obstacle point to be hit. The purpose of the next lemma is to show that, provided the stop point has not been hit, we can guarantee that if p is the answer to an NR segment dragging query, then the region swept out by the dragged segment is obstacle-free.

Lemma 6.1 *If $\overline{qq'}$ is a dragged segment of type NR which has event point p, and p is not a stop point, then no obstacle point was encountered by the dragged segment before it reached p.*

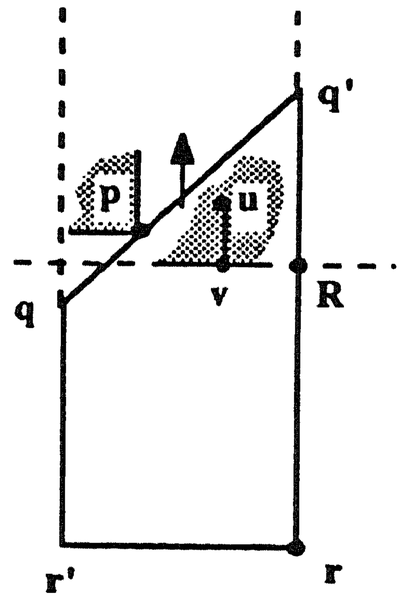


Figure 17. Proof of Lemma 6.1.

Proof: We need to show that the trapezoid $rr'qq'$ is obstacle-free for the diagram shown in Figure 17. Assume to the contrary that am obstacle point u existed interior to the trapezoid. Follow a ray to the south from u until the obstacle containing u is exitted. Call the exit point v. Now, unless v is a vertex (which would already be a contradiction), v must lie on a horizontal boundary segment of the obstacle. Following this segment to the right, we must exit the trapezoid before encountering a vertex. But this implies the existence of a stop point, R, above r, where propagation of the segment would have been stopped before the event point p was hit. ∎

> *Remark*: The above proof relies on the rectilinearity of obstacles to infer that the boundary containing v is horizontal. We shall return to this issue in Section 9, where we shall see that in case the obstacles are not rectilinear, a minor modification is needed.

We must also show that the "clipping" of dragged segments that is done in procedure *Propagate* is valid. What we actually show is that the region clipped by following the rules of step 2) of the procedure will not cause some region of the cell $C^{SE}(r)$ to be inaccessible to the resulting dragged segments that continue after propagation. This means that we do not clip "too much"; it may be the case, though, that we clip "too little", in that we may clip along a ray no portion of which lies on the boundary of the cell $C^{SE}(r)$. Hence we will not be able to charge the clip to a boundary segment of the planar subdivision S^{SE}, as we would

hope in order to get a simple proof of the linearity of the number of events. This subject will be treated in more detail in the next section.

Lemma 6.2 *Assume that p is an obstacle vertex which has already been reached from the southeast from root r_i (so that $r_i \in \mathcal{R}^{SE}(p)$). If the next event is that a segment $\overline{qq'}$ rooted at r hits p from the southeast, then any point $x \in H^{NW}(r_i) \cap H^{NW}(r)$ is better reached from the southeast through r_i than through r. This says that for any $r_i \in \mathcal{R}^{SE}(p)$, $H^{NW}(r_i) \cap C^{SE}(r) = \emptyset$.*

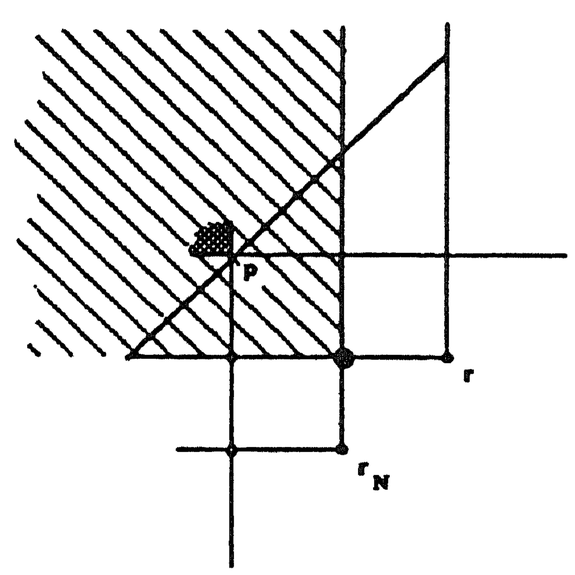


Figure 18. Proof of Lemma 6.2.

Proof: The proof follows immediately from the fact that if some point $x \in H^{NW}(r_i) \cap H^{NW}(r)$ were to be better reached from r than from r_i (from the southeast), then the segment $\overline{qq'}$ would hit p before a segment rooted at r_i would hit p, contradicting the fact that r_i is already an element of the list $\mathcal{R}^{SE}(p)$ at the time of the collision of $\overline{qq'}$ with p. Referring to Figure 18, then we see that no point west of the vertical line through r_N can be hit by r before it is hit by r_N. ∎

We say that a segment $(\overline{qq'})$ *separates* a rectangle $R(x, y)$ if $R(x, y) \setminus (\overline{qq'})$ is disconnected (that is, $\overline{qq'} \cap R(x, y) \neq \emptyset$ and $q, q' \notin R(x, y)$).

Lemma 6.3 *At any given stage of the algorithm, if $x \in C^{\delta}(r)$, where r is a vertex which has been permanently labeled, and x has not yet been permanently labeled, then there is a dragged segment, $\overline{qq'}$, in the event queue which is rooted at r and which separates the rectangle $R(r, x)$.*

Proof: The proof is by induction on the iteration count of the main loop. At the first iteration, we know that if x is directly accessible to s, then one of the four dragged segments surrounding s separates the rectangle $R(s, x)$. Now assume that the assertion holds after the first k iterations. Assume that at iteration $k + 1$ we are about to propagate segment $\overline{qq'}$ (rooted at r) to the northwest, and that the event position is $\overline{q_e q'_e}$. Assume that $x \in C^{\delta}(r)$ is not permanently labeled. Then, by the induction hypothesis, there exists a dragged segment in the event queue which separates $R(r, x)$. If this separating segment is not $\overline{qq'}$, then it will still separate $R(r, x)$ after propagation. So assume that $\overline{qq'}$ separates $R(r, x)$. If x lies in the region

swept out by $\overline{qq'}$, then it becomes permanently labeled after propagation (so we are done). Otherwise, $\overline{q_e q'_e}$ separates $R(r, x)$. When we propagate $\overline{qq'}$, we delete it from the event queue and establish either one or two new dragged segments (in the northwest direction) which are subsets of $\overline{q_e q'_e}$. We must check that when we propagate a dragged segment $\overline{qq'}$ rooted at r, then all points of $C^{SE}(r)$ are still potentially hit by the resulting dragged segments that we insert at propagation. This follows by a case-by-case analysis of the propagation rules, using Lemma 6.2 to check that any clipping is valid. ∎

The region swept out by all the dragged segments rooted at r will be a subset of one of the hulls $H^{\delta}(r)$ ($\delta \in \mathcal{D}$) of r (one can check in each case that the propagation of segments is limited to the hull). We now show that the region swept out by all dragged segments introduced by the algorithm is the entire obstacle-free plane, thus showing that the wavefront propagation is complete:

Lemma 6.4 *The dragged segments introduced during the algorithm sweep out the entire obstacle-free part of the plane.*

Proof: Let x be any point in the obstacle-free part of the plane. Then, $x \in C^{\delta}(r_k)$ for some obstacle vertex r_k and some direction $\delta \in \mathcal{D}$. Let $\{s, r_1, \ldots, r_k, r_{k+1} = x\}$ be the points along the shortest path from s to x that we get from backtracing through the SPM. Assume that at the conclusion of the algorithm x is not in the region swept out by the dragged segments. Then, there is a first root, r_j (possibly equal to r_k), along the path to x such that r_j is permanently labeled, but r_{j+1} is not. By construction, $r_{j+1} \in C^{\delta_j}(r_j)$ (where $\delta_j \in \mathcal{D}$ is the direction from r_j to r_{j+1}). Then, by Lemma 6.3 we know that there must be a dragged segment in the current event queue which separates the rectangle $R(r_j, r_{j+1})$. If this dragged segment has an event point of NIL, then r_{j+1} would be in the region swept out by it (and would be permanently labeled), contradicting the assumption that r_{j+1} is not yet permanently labeled. Otherwise, if the segment has a non-NIL event point, then we get a contradiction to the fact that the event queue has no entries with finite labels at the conclusion of the algorithm. ∎

Lemma 6.5 *In Step 2(a) of procedure* Propagate *(the case that p has not yet been labeled from the southeast, but it has been permanently labeled), p is a vertex on the boundary of the cell $C^{SE}(r)$. In Step 3 (the case that p has not yet been permanently labeled), the procedure properly labels p (with $d(p) = d(r) + d(r, p)$), and p must also be a vertex on the boundary of the cell $C^{SE}(r)$.*

Proof: The proof proceeds by induction on the iteration count of the main loop and mimics the usual proof of correctness of Dijkstra's algorithm. We show that Step 3 works properly. (The proof that Step 2(a) works properly is similar.) We claim that each time we permanently label an obstacle vertex, we label it with the correct shortest path length from s. This will follow from the fact that at the instant an event of label D (event point p, with root r) is popped from the queue, every vertex of distance $D' < D$ has already been permanently labeled (correctly). If event point p should have been permanently labeled from the root $r' \neq r$, then p is immediately accessible from r' and $d(r') + d(r', p) < D = d(r) + d(r, p)$ (so that r' is at a

distance less than D from s, implying that it has already been permanently labeled). Then by Lemma 6.3 there must be a dragged segment $\overline{q_1 q_1'}$ in the event queue which is rooted at r' and which separates the rectangle $R(r', p)$. But then $\overline{q_1 q_1'}$ has event distance *less* than D, a contradiction. ∎

Theorem 6.1 *The algorithm correctly computes the subdivisions $\mathcal{S}^\delta$, $\delta \in \mathcal{D}$.*

7. Complexity of the Algorithm

How many events can there be in running our algorithm? Each time an event occurs, we charge the event to some point on the grid defined by the vertices of the obstacle. The charged point may be an obstacle vertex, a crossing point of two "bullet paths", or a point where a bullet path hits an obstacle; thus, we are charging some subset of the nodes of the simple grid graph as defined in Section 2. Since there are a quadratic number of such grid points, we must be careful not to charge too many of them.

Let us review the charging scheme specified in *Propagate*. If event point p is a stop point, then we "charge" the event to p. (There are at most $4n$ stop points.) Otherwise, if p has not been labeled before, then we charge the event to obstacle vertex p. (There are precisely n obstacle vertices, and each can be charged at most 4 times.) Otherwise, when we propagate the event segment past p to the northwest, we make an insertion into the list $\mathcal{R}^{SE}(p)$. Each time such an insertion is made, we either charge the southwest corner of the rectangle $R(r, p)$ with a West Clip (WC), charge the northeast corner of the rectangle with a North Clip (NC), or charge the root r with a Stop Propagation (which can occur at most 4 times at each of the n vertices). (Similar statements hold for propagation in the other directions.)

Thus, defining $g(n)$ to be the total number of events, we get that $g(n) \leq 12n + g^w_{nw}(n) + g^n_{nw}(n) + g^w_{sw}(n) + g^s_{sw}(n) + g^e_{ne}(n) + g^n_{ne}(n) + g^e_{se}(n) + g^s_{se}(n)$, where $g^w_{nw}(n)$ is the number of West Clips (WC) that occur when doing propagation to the northwest (and the other terms are defined similarly). We will concentrate on bounding the term $G(n) = g^w_{nw}(n)$, as the others are exactly the same.

An example of the charging of WC's to grid graph points is shown in Figure 19. Here, it is assumed that the length of the shortest paths from s to the roots r_i $(i = 1, 2, 3, 4)$ are ordered as follows: $d(r_1) << d(r_2) << d(r_3) << d(r_4)$. With this assumption, we get the West Clip charges as shown by small hollow circles. The numbers by the charges indicate the order in which the collisions take place. The large circles with the "x" indicate the positions of the actual clip points. (The vertical lines through the clip points are the new left track rays that are implied by the West Clips.)

Remark 1: It is not the case that when a wavefront is clipped (either on the West or the North) that it is clipped "correctly", in that the clipping may not be at the boundary of the cell of the final Northwest subdivision. For example, in Figure 19 there are three clip points in the row with root r_3 (those that occur when r_3 collides with c, d, and f), but only the rightmost clip corresponds to the actual left boundary of the cell $C^{SE}(r_3)$. Lemma 6.2 assures that we do not clip off too much, but it does not guarantee that the

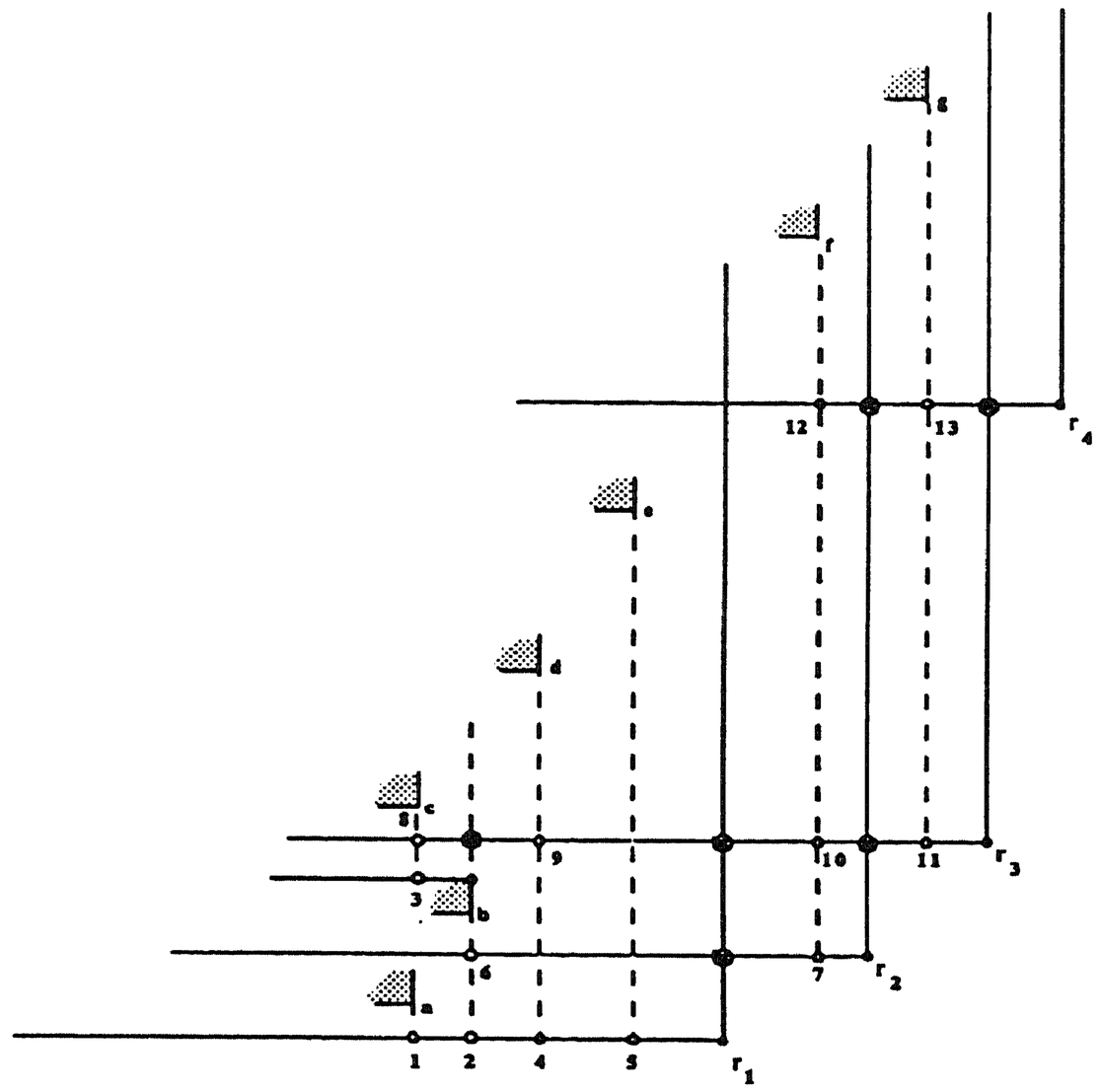


Figure 19. Example of charging scheme for northwest propagation.

clip is the most effective clip possible. (Indeed, if clipping were to occur only along the boundaries of cells $C^{SE}(r)$, then we could charge each clip to a boundary segment of the planar subdivision $\mathcal{S}^{SE}$, yielding a linear bound on the number of events. Ideally, for instance, we would clip propagation from r_3 at the vertical line through r_2 immediately upon discovering the collision with point c.) Our charging scheme, though, will show that this propagation scheme will not result in too many hits.

We analyze the complexity of $G(n)$ by relating it to the number of ones that can occur in a binary matrix in which certain "violated patterns" cannot exist. Recall that procedure *Propagate* charges each West Clip to a point of the simple grid graph, namely, that point in the lower left corner of the rectangle $R(r,p)$. Our first claim is that this charging scheme cannot result in all four corners of a rectangle being charged with a West Clip (that is, there do not exist four distinct nodes of the simple grid graph defined on the obstacle vertices such that all four nodes are charged WC and the four nodes are on the corners of a rectangle).

Lemma 7.1 *Procedure* Propagate *will not charge the four corners of a rectangle with a West Clip.*

Proof: To see that the four corners of a rectangle cannot be charged WC, assume the contrary. (Refer to Figure 20.) If r_1 hit p_1 first, then when r_2 hits p_1, it will be clipped to the right of r_1, and would then never hit p_2. Similarly, if r_2 hits p_1 first, then r_1 would be clipped above when it hits p_1, so it would never hit p_2. But a rectangle of WC charges implies that all four collisions must occur, a contradiction. ∎

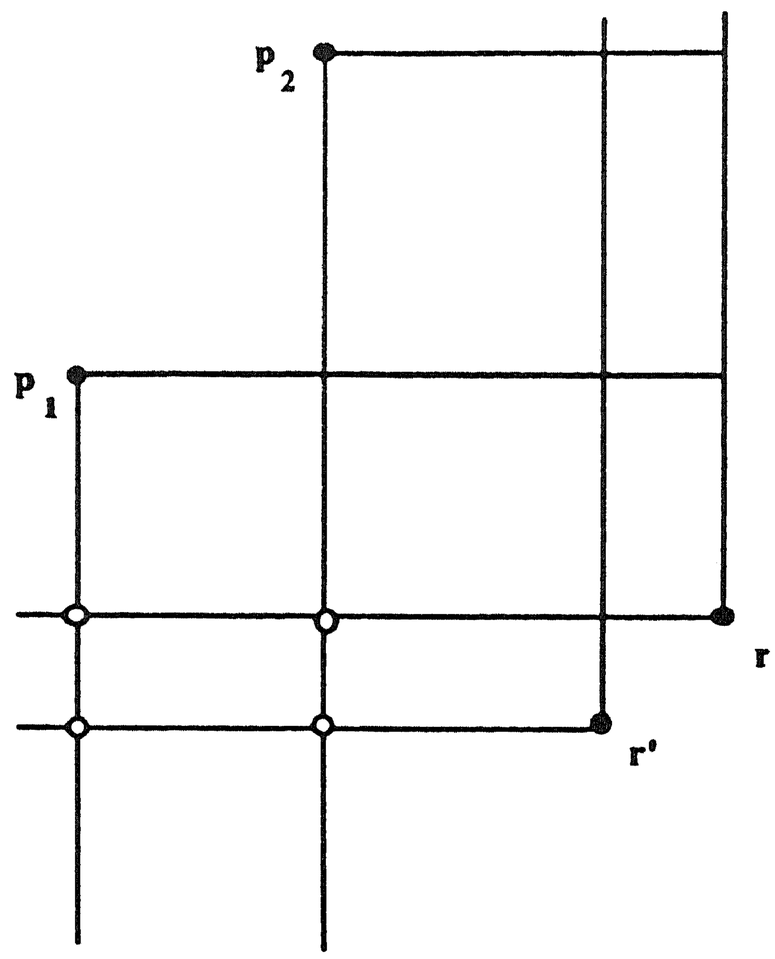


Figure 20. Proof of Lemma 7.1.

Now, think of the nodes of the grid graph as defining elements of a matrix, whose rows correspond to distinct horizontal obstacle edges and whose columns correspond to distinct vertical obstacle edges. Let the entry of the matrix be one if the corresponding grid point is charged WC and let it be zero otherwise. (Note that each grid point can be charged WC at most once.) The matrix will have $M = \frac{n}{2}$ rows and $N = \frac{n}{2}$ columns (so that $M + N = n$). We say that a binary matrix contains no rectangles (or is "rectangle-free") if there do not exist distinct rows i and i' and distinct columns j and j' such that all four matrix elements at (i, j), (i, j'), (i', j), and (i', j') are ones. We now claim:

Lemma 7.2 *In an M-by-N binary matrix which has no rectangle of ones will have at most $O((M+N)^{1.5})$ ones.*

Proof: (Due to Schmeichel [Sch]. See also [Lo], problem 10.36(a).) Let n_i be the number of 1's in row i. Then, we claim that

$$\sum_{i=1}^{M} \binom{n_i}{2} \leq \binom{N}{2}.$$

To see this, we argue as follows: Each pair of ones in row i "eliminates" a pair of columns. Once a pair of columns is eliminated, no other pair of ones in any row can appear in that pair of columns (otherwise, there would be a rectangle of ones). Thus, there cannot be more pairs of ones than there are pairs of columns. Now, together with the inequality

$$\left(\frac{\sum n_i}{M}\right)^2 \leq \frac{\sum n_i^2}{M},$$

(which follows from Jensen's inequality), we get that

$$\frac{1}{M}\left(\sum n_i\right)^2 - \sum n_i \leq N(N-1).$$

Letting $S = \sum n_i$ be the total number of ones in the matrix, then we have that $S^2 - MS - NM(N-1) \leq 0$, which implies that

$$S \leq \frac{M}{2}\left(1 + \sqrt{1 + 4\frac{N}{M}(N-1)}\right) = O((M+N)^{1.5}),$$

which finishes the proof. ∎

Corollary 7.3 *There will be at most $G(n) = O(n^{1.5})$ grid graph points charged with WC, and hence $g(n) = O(n^{1.5})$.*

Remark 2: The $O((M+N)^{1.5})$ bound on the number of ones in a rectangle-free matrix can be shown to be tight for $M = N = p^{2k} + p^k + 1$, p a prime and k an integer, by appealing to the theory of projective planes [Lo, Sch]. That is, there are examples of rectangle-free binary matrices which achieve the upper bound on the number of ones. Thus, using only the violated pattern of rectangles, we cannot do any better than a bound of $G(n) = O(n^{1.5})$.

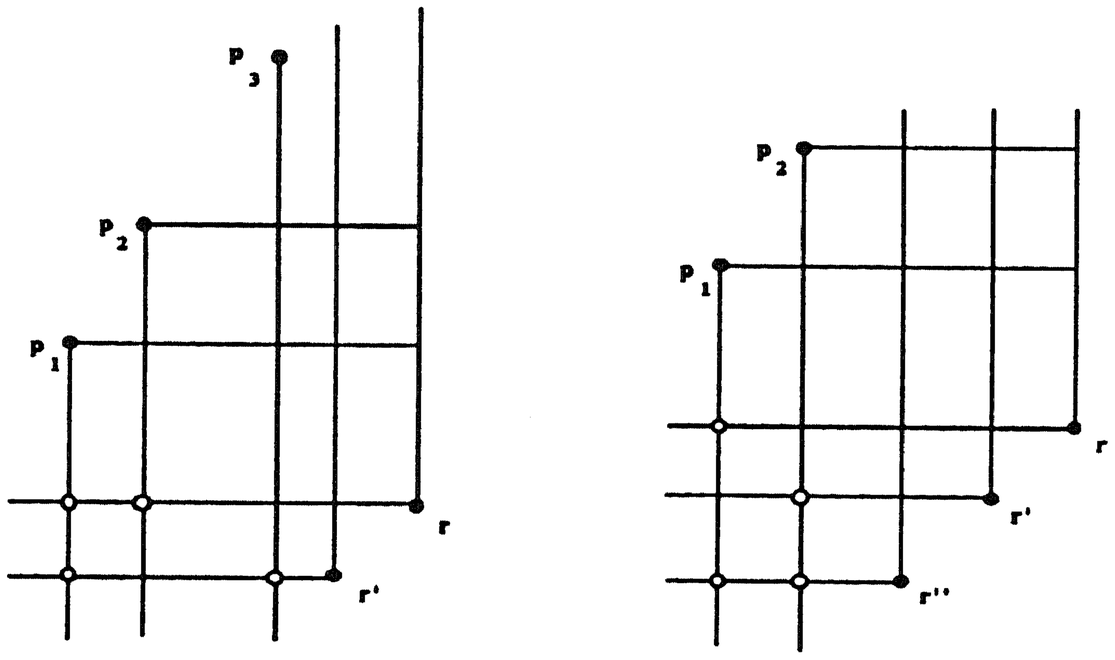


Figure 21. Other violated patterns.

To get a better bound on the number of West Clips, we look for other violated patterns of WC charges. An example of another violated structure is that of a specific type of trapezoid. Namely, we can show that charges will not form a trapezoid of the following form (we call this a *Type I trapezoid*): Let i and i' be two rows, with $i' > i$. Let j, j', and j'' be three columns, with $j < j' \leq j''$. Then we claim that there can be no set of WC charges on the points (i,j), (i',j), (i,j'), and (i',j''). (Refer to the left drawing in Figure 21.) Furthermore, another violated trapezoid is of the following form (we call this a *Type II trapezoid*): Let i,

i', and i'' be three rows, with $i \leq i' < i''$. Let j and j' be two columns, with $j < j'$. Then we claim that there can be no set of charges on the points (i,j), (i',j'), (i'',j), and (i'',j'). (Refer to the right drawing in Figure 21.) Note that, as defined, Type I (or Type II) trapezoids include rectangles.

Lemma 7.4 *There can be no trapezoids of Types I or II among the grid points charged with a WC.*

Proof: Refer to Figure 21. Assume that a trapezoid of Type I exists, as shown on the left in the figure. Note that the order in which points p_1, p_2, and p_3 are hit by a diagonal line dragged to the northwest must be p_1, p_2, p_3. (This follows since, if for example, p_3 were to be the first one hit, then when r' hits p_3 and gets west clipped, it is cut off from being able to hit p_1 or p_2.) Then, r' must have hit p_1 *before* r did (otherwise, r' would have been north clipped when it hit p_1, rather than being west clipped). But then, when r does hit p_1, it is west clipped at a point at or east of point r', meaning that it could not have gone on to collide with p_2. This is a contradiction. The proof for Type II trapezoids is similar. ∎

We conjecture that the maximum number of ones in an M-by-N matrix without trapezoids of Type I (or of Type II) is $o((M+N)^{1.5})$; however, we have not yet been able to show this. We should mention that Schmeichel [Sch] has found an example of an N-by-N matrix without trapezoids of Type I which contains $\Omega(N \log N)$ ones. A stronger statement of sparsity is to disallow both trapezoids of Type I and those of Type II (as is indeed the case for our WC charges, as claimed in Lemma 7.4 above). The lower bound of $\Omega(N \log N)$ on the maximal number of ones in a Type I trapezoid-free N-by-N matrix is an example matrix which has many Type II trapezoids, so it does not apply to the case of considering both violated patterns simultaneously. The only upper bound we have for the case of no Type I or Type II trapezoids is the trivial one of $O((M+N)^{1.5})$ given by the rectangle-free property. It is an interesting open problem to improve these upper and lower bounds. Is it true that Type I trapezoid-free N-by-N matrices have a maximum number of ones given by $\Theta(N \log N)$? Can one show that matrices without either type of trapezoid can have at most $o(N \log N)$ ones?

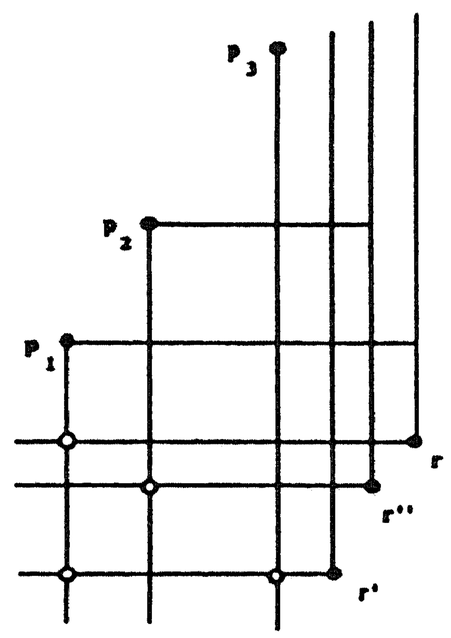


Figure 22. A violated L-quadrilateral.

Continuing our quest to reduce the upper bound on $G(n)$, we now investigate a considerably stronger example of a violated pattern. Refer to Figure 22. We claim that if three charges exist which form an "L" pattern (in the figure, those defined by (r', p_1), (r, p_1), and (r, p_3)), then *all* of the points of the grid graph corresponding to points *interior* to the rectangle defined by these three charges are ruled out as having been charged. Note that the rectangle-free property says that the upper right corner of the rectangle cannot be charged, Type I trapezoids rule out charging the top edge of the rectangle (except the left-most point), and Type II trapezoids rule out charging the right edge of the rectangle (except the bottom-most point). Thus, a strong statement of the sparsity condition is as follows. In a binary matrix there can be no pattern of the following form: for $i \leq i'' < i'$ and $j < j'' \leq j'$, if there are ones at the locations (i, j), (i', j), and (i', j'), then there cannot be a one at location (i'', j''). (See Figure 23.) We say that such matrices are free of "L-quadrilaterals". Note that a matrix that is free of L-quadrilaterals is by definition automatically free of rectangles and trapezoids of Types I and II.

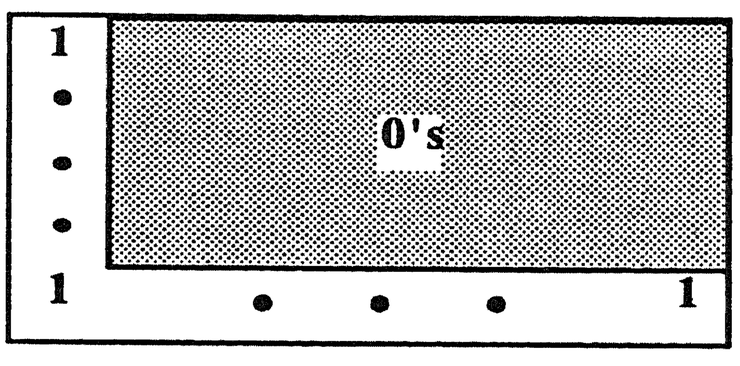


Figure 23. Sparsity condition.

Lemma 7.5 *There can be no L-quadrilateral of WC charges to grid points.*

Proof: Refer to Figure 24. First, note that the order in which points p_1, p_2, and p_3 are hit by a diagonal line dragged to the northwest must be p_1, p_2, p_3. (This follows since, if for example, p_3 were to be the first one hit, then when r' hits p_3 and gets west clipped, it is cut off from being able to hit p_1 or p_2.) Also, r' must hit p_1 before r does (otherwise, r' would be north clipped upon colliding with p_1). Where can r'' be? It must be east of r' (so that $R(r', p_1)$ is obstacle-free) and it must be west of r (so that $R(r'', p_2)$ is obstacle-free). Now the wavefront propagating from r'' would normally hit p_1 before it got to p_2 (and if this were to happen, then r'' would never get a chance to hit p_2, as it would be west clipped at r'). What stopped r'' from hitting p_1? Propagation from r'' must have been west clipped at some point east of p_1. This could happen either because r'' hit a left stop point some place east of p_1 (which would violate the fact that $R(r', p_1)$ is obstacle-free), or because r'' collided with some point (q_1) west of p_1 which caused a west clip to occur east of p_1 (at the vertical line through r_1, the root causing the west clip). Now, r_1 must be east of p_1 and west of p_2, and of course it must be south of r' in order for $R(r', p_1)$ to be obstacle-free. Also, q_1 is hit from r_1 before p_1 is hit. But this means that we must protect against r' hitting q_1 before it hits p_1 (otherwise, r'' will be clipped on the west by r' due to the fact that they both hit q_1). This implies

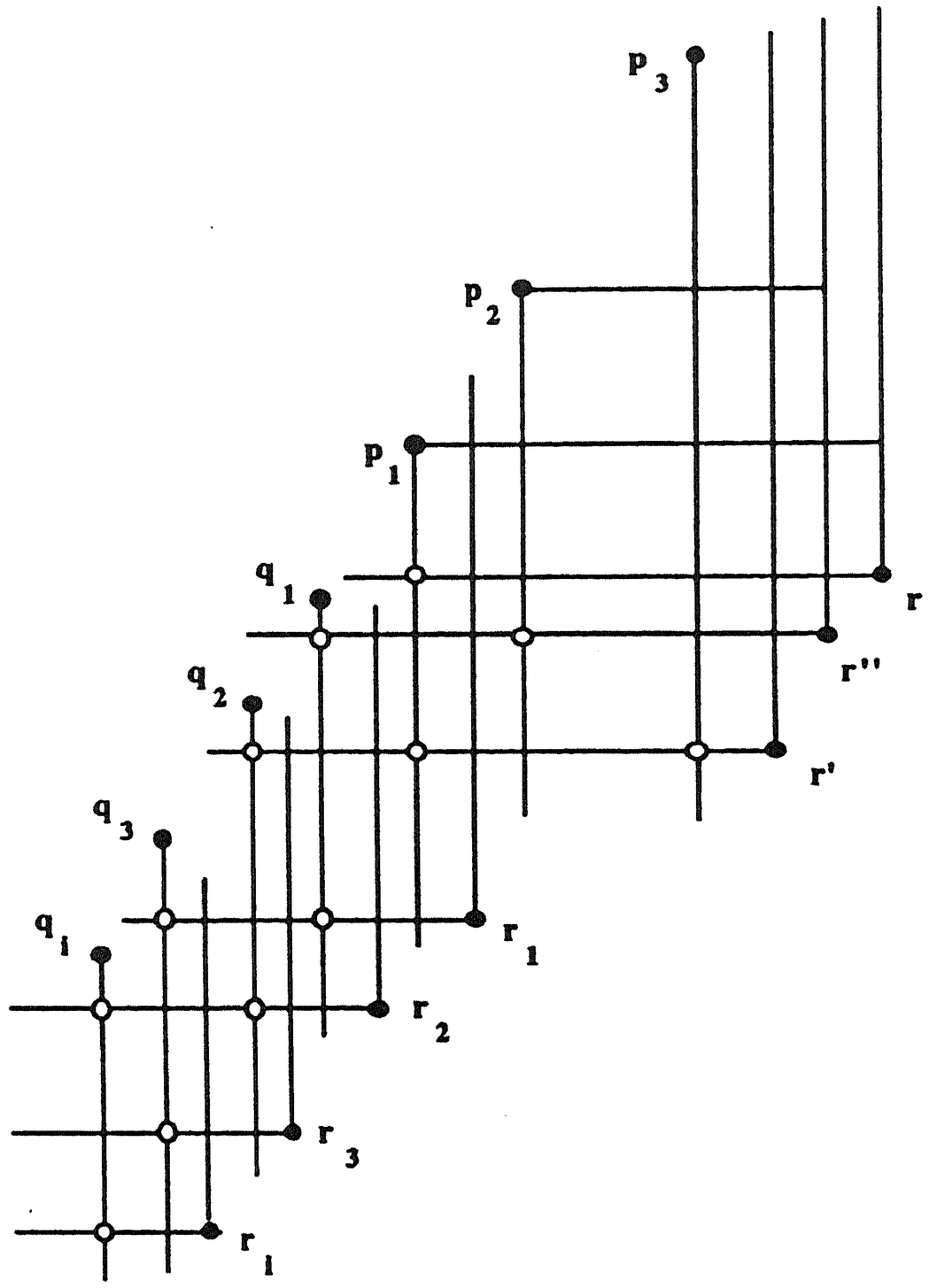


Figure 24. Proof of Lemma 7.5.

the existence of a point q_2 which caused r' to be west clipped along the vertical line through a point r_2. Continuing in this way we see that we end up with the implication that an infinite list of such roots r_i and vertices q_i must exist. This is clearly a contradiction. ∎

Thus, the maximum number of events that can occur in our algorithm is bounded above by the maximum number of ones that can exist in an L-quadrilateral-free binary matrix. We originally conjectured that this maximum density of ones would be linear in the size of the matrix. However, Bienstock and Györi [BG] have recently resolved the question of the maximum density of L-quadrilateral-free matrices and found that the number is slightly superlinear.

Lemma 7.6 *The maximum number of ones in an L-quadrilateral-free binary N-by-N matrix is $\Theta(kN)$, where k is defined by $k^k = N$; thus, the maximum number of ones is $\Theta(N\frac{\log N}{\log\log N})$.*

Corollary 7.7 *There will be at most $G(n) = O(n\frac{\log n}{\log\log n})$ grid graph points charged with WC, and hence $g(n) = O(n\frac{\log n}{\log\log n})$.*

Thus, the sparsity condition of being L-quadrilateral free is not enough to prove a linear bound on $g(n)$. However, other violated patterns can be shown, and these may lead to a proof of linearity. Thus, more work is needed in order to resolve whether or not our algorithm is actually optimal. In the end, there may turn out to exist a very simple charging scheme to show that the number of events is linear.

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

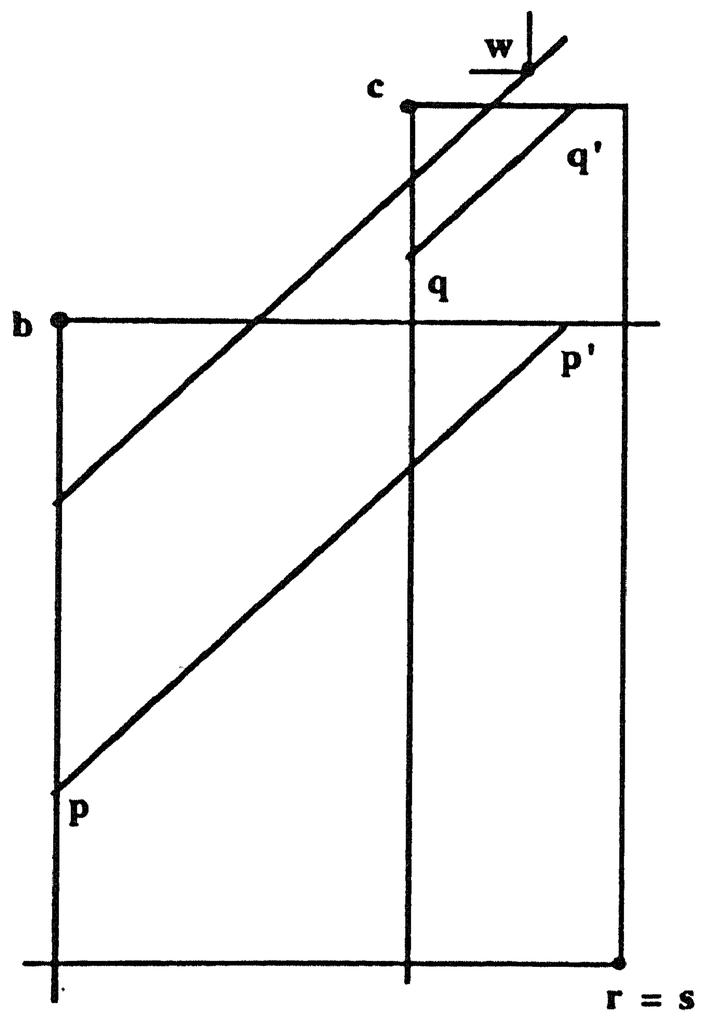


Figure 25. Proof of Lemma 7.8: Case $r = s$.

Proof: Assume that w is hit by a dragged segment in procedure *Find-Next-Event-NWI*. Let $\overline{qq'}$ be the original position of the segment being dragged into corner c, let its root be r, and let the segment in contact with w be $\overline{q_e q'_e}$. Assume that w is also hit by a call to *Find-Next-Event-NWI* in which the original segment is

$\overline{pp'}$, it is being dragged into corner b, its root is s, and the segment in contact with w is $\overline{p_e p'_e}$. Without loss of generality, assume that b is south of c (implying that b is southwest of c, since both b and c are immediately accessible from r).

(a). If $r = s$, then we get a contradiction as follows. (Refer to Figure 25.) First, we note that the point p' must either be on the boundary of an obstacle, or it must lie on a horizontal line along which propagation from r was clipped on the north, or it lies on the horizontal line through an obstacle edge which is due north of r (meaning that there is a stop point due north of r on the horizontal line through b). (This follows by the specification of the procedure *Propagate*.) We check now that each case results in a contradiction.

(i). If p' lies on an obstacle boundary, then, by the general positioning assumption, we know that p' is not an obstacle vertex, so some part of the obstacle containing p' must lie west of r. But then *all* of the obstacle edge containing p' must lie west of c, since c is immediately accessible from r. In particular, p' must lie west of c. But if p' is west of c, there is no way for $\overline{p_e p'_e}$ to hit w, as p'_e must be west of p'.

(ii). If p' lies on a horizontal boundary along which clipping on the north was done, then the dragged segment $\overline{qq'}$ could not exist, since q' is north of p'. (Immediately after a dragged segment rooted at r is clipped on the north, it will be the only segment rooted at r propagating to the northwest.)

(iii). If there were a horizontal obstacle boundary segment due north of r along the horizontal line through b, then we would have a contradiction to the fact that c is immediately accessible from r.

(b). If $r \neq s$, then we get a contradiction as follows. We consider two cases: b is north of r, and b is south of r.

(i). If b is north of r, we get the case shown in Figure 26. Now, there must exist a root r_1 due south of c which is responsible for the left vertical track ray through c. (There could not have been an obstacle boundary causing a left stop point at the lower left corner of rectangle $R(r, c)$, as this would violate the fact that b is immediately accessible from s.) Since r_1 is responsible for west clipping r, there must be a point q_1 which was hit by both r and r_1. (In the figure, charges are shown by small hollow circles.) Similarly, there must exist a root r_2 due south of b which is responsible for the left vertical track ray through b. (There could not have been an obstacle boundary causing a left stop point at the lower left corner of rectangle $R(s, b)$, as this would violate the fact thatq_1 is immediately accessible from r_1.) Let q_2 be the point which was hit by both s and r_2, causing the west clip of s along the vertical line through r_2.

Now, r_1 is not allowed to hit q_2 (otherwise, s would have been west clipped by r_1 instead of r_2), so there must exist a point q_3 which was hit by r_1 and by some root r_3 (which lies west of q_1 but east of q_2. Continuing this line of reasoning leads to the existence of an infinite sequence of such points r_i and q_i. (The argument is similar to that of the proof of Lemma 7.5.)

(ii). If b is south of r, we get the picture in Figure 27. Now something caused s to be west clipped along the vertical line through c (and the clipping event point had to occur along a diagonal line which is southeast of the diagonal line through w). If the clipping were caused by an obstacle boundary that lies along the vertical line through c, then the segment $\overline{pp'}$ could not have been dragged in such a way to hit w,

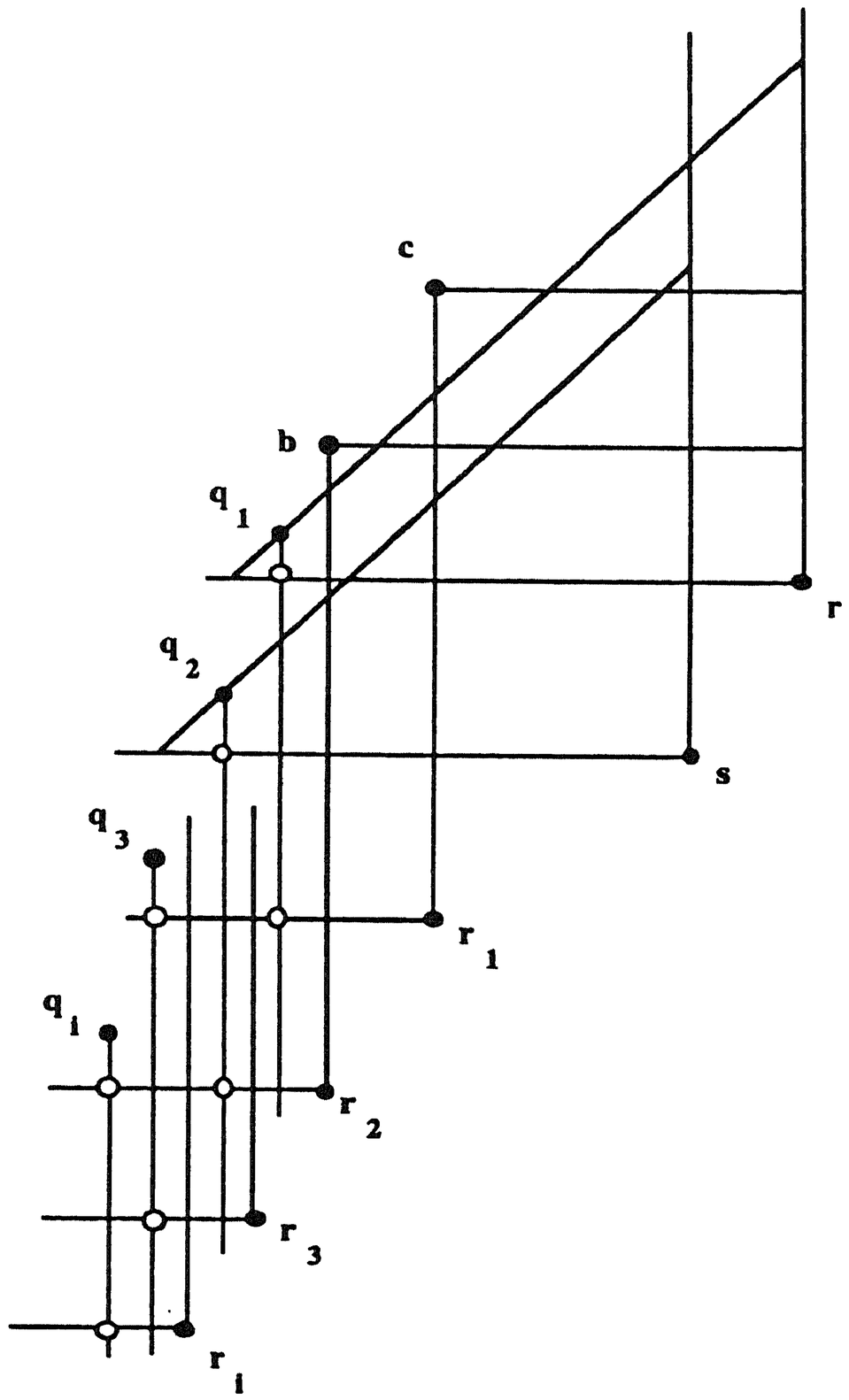


Figure 26. Proof of Lemma 7.8: Case $r \neq s$ and b above r.

as it would hit the obstacle first. Thus, the clipping must have been caused by some root r' due south of c. Note that r' must also be south of r in order for b to be immediately accessible from r. Now roots r' and s both hit some point u. Also, u must lie on a diagonal line which is southeast of the diagonal line through w. But this implies that $\overline{pp'}$ would hit u before hitting w, a contradiction. ∎

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

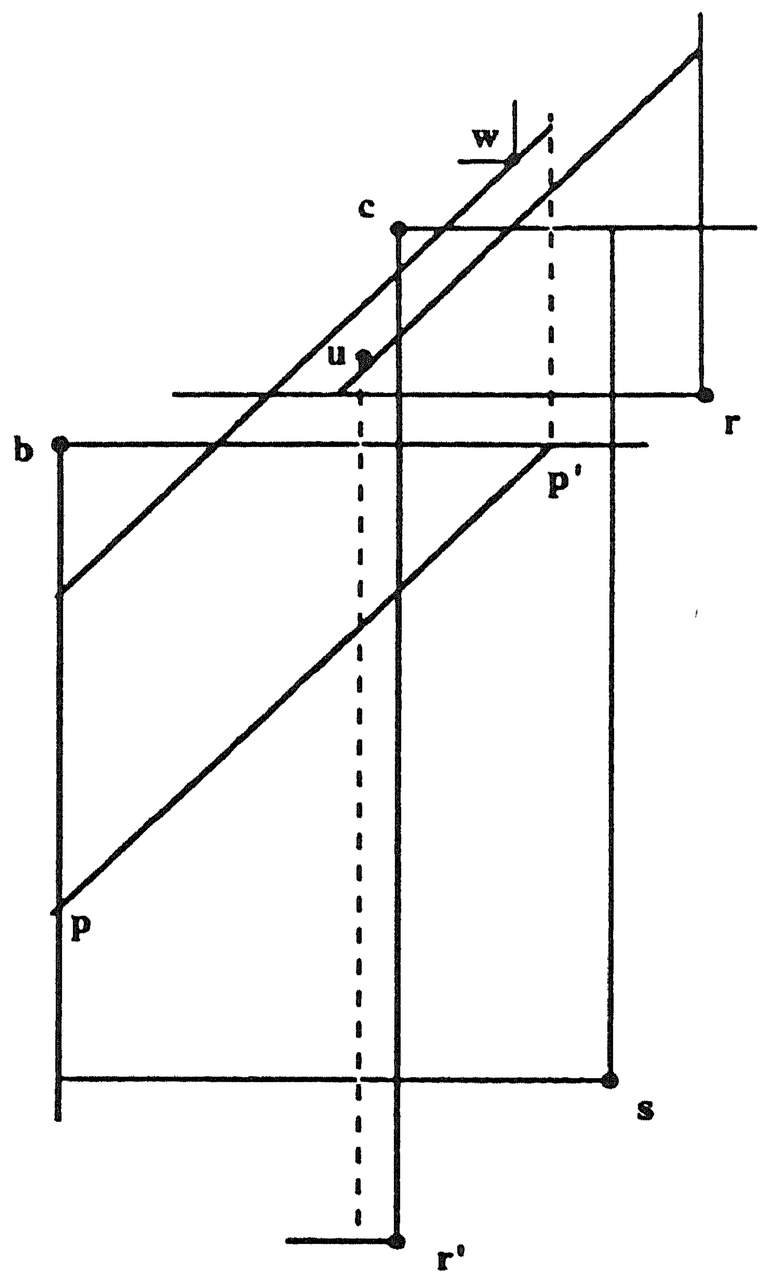


Figure 27. Proof of Lemma 7.8: Case $r \neq s$ and b below r.

By keeping track of the paths of the endpoints of all dragged segments, we can actually build the subdivisions $\mathcal{S}^\delta$ ($\delta \in \mathcal{D}$) as the algorithm proceeds. At the conclusion of the algorithm, the subdivisions can be put into a structure which is appropriate for efficient point location queries [K,Pr]. Also, the four subdivisions $\mathcal{S}^\delta$ can be merged into the SPM in time $O(n \log n)$.

Theorem 7.1 *The algorithm above solves the Single-Source Rectilinear Shortest Path Problem for a rectilinear obstacle space with n vertices in time $O(g(n) \log n)$ using $O(g(n))$ space, where $g(n)$ is the number of events and is bounded by $O(n \frac{\log n}{\log \log n})$. 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.*

8. Higher Dimensions

We turn now to the problem of finding shortest paths in higher dimensions. Finding shortest *Euclidean* paths in three or more dimensions is a very hard problem. In three dimensions, for example, the best known algorithms are an exponential-time algorithm [RS] and a fully polynomial approximation scheme

[Pa]. One approach to understanding better the multiple-dimension shortest path problem (and to arriving at approximate solutions to the Euclidean problem) is to examine the shortest path problem in other simple metrics. We look here at the case of the L_1 metric (or equivalently, the L_∞ metric).

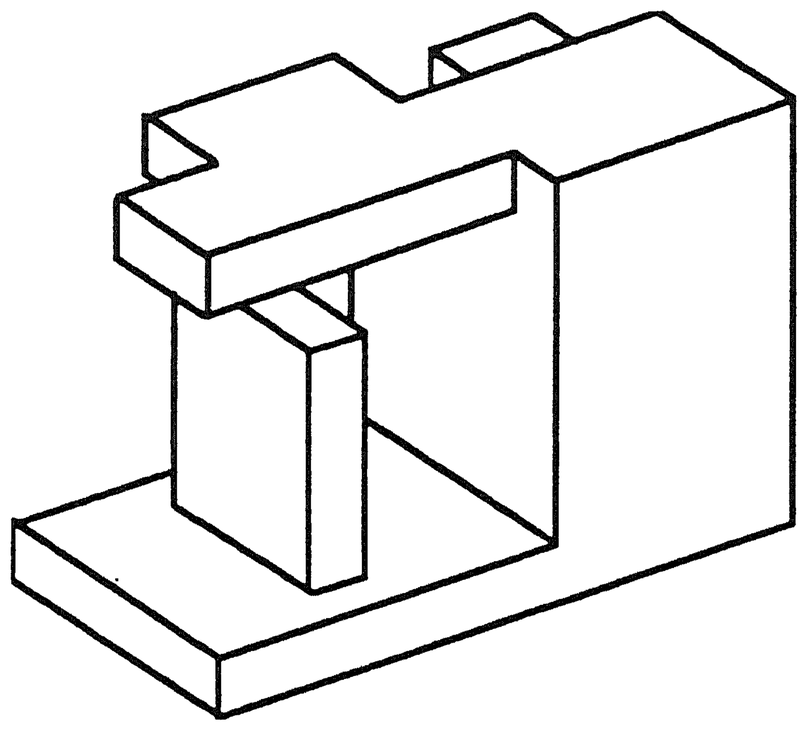


Figure 28. A set of orthohedral 3-dimensional obstacles.

Our approach to this problem is quite simple, but surprisingly we have found no mention of these results in the literature. We simply generalize to higher dimensions the observation that shortest rectilinear paths in the plane can be found that lie on the grid graph defined by the obstacles.

Consider d-dimensional space in which the coordinate system is specified in terms of the unit vectors $e_1, e_2, \ldots, e_d$, where e_i is the unit row vector with a 1 in the ith location. The L_1 distance between point $p = (p_1, p_2, \ldots, p_d)$ and point $p' = (p'_1, p'_2, \ldots, p'_d)$ is defined as $\Sigma_j |p_j - p'_j|$.

The obstacles are now assumed to be *orthohedral*. A d-dimensional polyhedron is orthohedral if every one of its facets (which have dimension $d-1$) is perpendicular to one of the coordinate axes (ie, lies in a hyperplane defined by one of the e_i's) and each facet is itself an orthohedral polyhedron in $d-1$ dimensions. We define an orthohedral obstacle for $d = 2$ to be any rectilinear simple polygon. An example for $d = 3$ is shown in Figure 28. A d-dimensional orthohedral obstacle can alternatively be defined as a union of d-dimensional hyperrectangles whose faces are perpendicular to the coordinate axes.

The basic idea behind finding a polynomial-time algorithm to find shortest paths in d dimensions is that we can limit our search to a d-dimensional grid graph defined by the vertices of the scene and the points s and t.

The orthohedral *grid* defined by a set of points $\{q_1, \ldots, q_n\}$ in d dimensions is the graph whose node set is given by $\mathcal{N} = \{(q_1, \ldots, q_d) : q_j \in \{q_{1,j}, \ldots, q_{n,j}\}\}$ and whose edge set is given by $\mathcal{E} = \{(q, q') : q, q' \in \mathcal{N}, q$ differs from q' in at most one coordinate, and no other node lies on the segment between q and $q'\}$. This construction is the natural generalization of the two-dimensional rectilinear grid graph.

Lemma 2.1 now generalizes to d dimensions.

Lemma 8.1 *If there exists a shortest path (in L_1) from s to t which avoids a given set of orthohedral obstacles in d dimensions, then there exists an obstacle-avoiding shortest path which lies entirely on the orthohedral grid defined by the vertices of the obstacles and s and t.*

Proof: The proof is by induction on the dimension d. The details are omitted here. ∎

As in the two-dimensional case, there is a slightly sparser graph which would work as the basis for shortest paths.

Using the observation of Lemma 8.1, it is easy to arrive at an algorithm to find shortest paths. One first constructs the relevant part of the grid graph (that which is not interior to any obstacle), and then one runs Dijkstra's algorithm on the resulting graph. The grid will have $O(n^d)$ nodes and $O(dn^d)$ edges, and it can be constructed in time $O(dn^d)$. Dijkstra's algorithm can then be run in time $O(dn^d \log n)$, yielding an algorithm with total running time $O(dn^d \log n)$ and space $O(dn^d)$.

Theorem 8.1 *The d-dimensional rectilinear shortest path problem can be solved in time $O(dn^d \log n)$ and space $O(dn^d)$.*

9. Extensions

One 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 rectilinear obstacles in the L_1 plane. The running time is then $O(g(N) \log N)$ with a space complexity of $O(g(N))$, where N is the sum of the number of sources and the number of obstacle vertices and again $g(N)$ is the number of events (which was shown to be bounded by $O(n^{1.5})$). A similar approach allowed Mount [Mo] to generalize the solution of the discrete geodesic problem [MMP] to the case of multiple sources. Note that this then solves the problem of [LL] with m origin-destination pairs in time $O(g(m+n)\log(m+n))$, improving the previous bound of $O(m(m^2+n^2))$.

For simplicity, we have described only the case of rectilinear obstacles in this paper; however, the algorithm generalizes immediately to the case of disjoint simple polygons. The algorithm proceeds as before, with modifications in procedure *Propagate* to allow for arbitrary obstacle edges. (See Figure 29 for an example of how propagation behaves at a vertex of a simple polygon.) We note here a few of the modifications needed in the case of simple polygonal obstacles. First, note that the subdivisions $\mathcal{S}^\delta$ will no longer have rectilinear cells, due to the nonrectilinear obstacles. Also, note that the Type II events will no longer be possible, assuming that we make a general positioning assumption about there being no horizontal or vertical obstacle edges. In other words, Type II events were actually a product of the degeneracy of the rectilinear case.

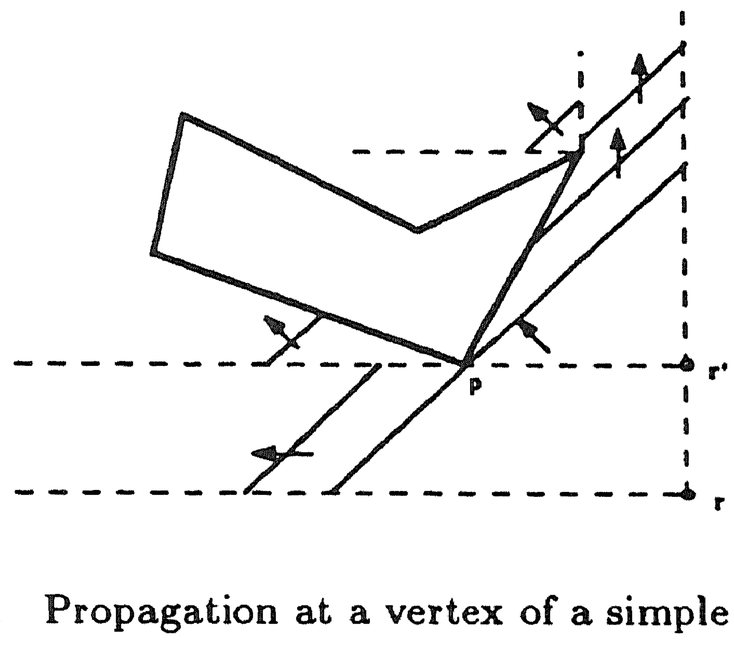


Figure 29. Propagation at a vertex of a simple polygon.

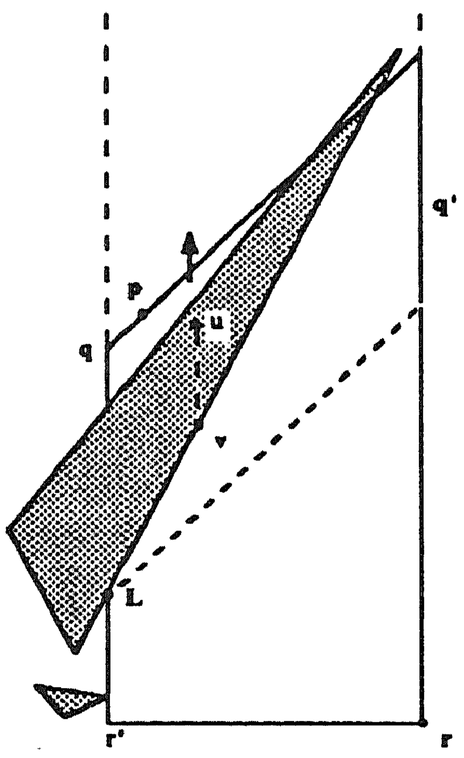


Figure 30. The need for detecting both left and right stop points.

Another slight modification that must be made is that for dragged segments whose left and right track rays are parallel, we must examine *both* tracks for a stop point. For example, to do a segment dragging query of type NR in the presence of rectilinear obstacles, we showed (in Lemma 6.1) that it was sufficient to check for stop points on the right track ray. Consider, however, the situation shown in Figure 30. By allowing the dragged segment to hit p, we have missed detecting the collision with the obstacle containing point u. The inherent problem is that the boundary segment through v need not be horizontal, and may therefore intersect $\overline{qq'}$ and the left track ray, avoiding crossing the right track ray. By similar reasoning to that of the proof of Lemma 6.1, though, we can show that if we check for both a left and a right stop point, then we can always make sure that the event point is accessible from the root r. Once a NR segment hits a stop point L on its left track, we must continue the propagation in a slightly different way: We insert a "virtual root" on the right track at the same y-coordinate as the left stop point. We then propagate a

dragged segment of type NWO out from the virtual root. This allows the propagation to "feel the effect" of the obstacle hit at the left stop point. (This is no different from the similar situation when we hit a right stop point and must begin dragging a segment into an inside corner.) For example, we see that in Figure 29, after collision with point p, the dragged segment of type NR which continues northward to the right of p actually immediately hits its left stop point, p. (This will occur whenever there is an interior collision at a point p such that the vertical ray through p enters the obstacle containing p at p.) We therefore insert the virtual root r' and perform an NWO query from r' to determine the next event.

In summary, for the case of simple polygonal obstacles, our algorithm generalizes to build the SPM in time $O(g(n)\log n) = O(n\frac{\log^2 n}{\log\log n})$, and from it we can answer queries in $O(\log n)$ time about the length of the shortest path from s to a query point. We can give the sequence of k obstacle vertices along a shortest path to any t in $O(k + \log n)$ time.

> *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]).

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 and $\pi/2$. The case in which there are k fixed orientations along which distances are measured gives rise to piecewise linear wavefronts with k segments from each virtual source. The segment dragging queries as discussed in Section 3 were sufficiently general to handle these cases. Note that if the fixed orientations are evenly spaced (in $[0, \pi)$), then as k grows large, the fixed orientation distance becomes a close approximation to the Euclidean distance (the percentage error decreases like $1/k^2$). Distances with fixed orientations can be defined in three dimensions as well. This approach may lead to a new approximation scheme for the Euclidean problem in higher dimensions.

We feel that our approach will also yield an efficient algorithm for the construction of the L_1 Voronoi diagram of a collection of disjoint line segments (or polygons). This should also generalize to the case of fixed orientation metrics.

10. Conclusion

We have presented an algorithm for computing Voronoi diagrams according to the L_1 metric (and more generally, fixed orientation metrics) in the presence of polygonal obstacles. The algorithm runs in time $O(g(n)\log n)$ and space $O(g(n))$, where $g(n)$ is a function which has been shown to be almost linear ($O(n\frac{\log n}{\log\log n})$) and is conjectured to be linear. Thus, our algorithm is almost optimal, and we conjecture that it can be shown that the number of events $g(n)$ is linearly bounded, implying optimality of our method.

We leave it as an open problem to find other, potentially simpler, charging schemes to prove upper bounds on the number of events.

Our investigations into the densities of certain sparse matrices suggests an interesting area of research. There are a variety of violated patterns of ones that can be studied. We ask what the maximum density of ones is in a binary matrix with no violated patterns. (See [MS].) Several specific open questions were suggested in Section 7. These questions are related to extremal problems for graphs [Lo].

Our algorithm builds four separate subdivisions by considering the propagation of the wavefront in the four directions individually. The algorithm could have been written with construction of only one subdivision. Then, when a dragged segment hits a vertex which has been already labeled, we do clipping according to the bisector between the current root and the root that labeled the hit vertex. This prevents having to keep track of four separate roots which cause a vertex to be labeled. The disadvantage of this direct technique is that it seems to complicate the charging schemewhich bounds the number of events. It should be possible, though, to devise a proof based on this straight-forward method.

One obvious open problem in two dimensions is to find more efficient algorithms for determining shortest Euclidean paths in two dimensions in the presence of a general collection of polygonal obstacles.

We feel that substantial improvements of the algorithm for higher dimensions are possible. In particular, we feel that a wavefront approach similar to that used here in the two-dimensional case will yield an algorithm in d dimensions with a time complexity of $O(n \log^{d-1} n)$.

Also of extreme interest is the problem of shortest Euclidean paths among polyhedral obstacles in three or more dimensions.

There has been some recent progress on the shortest path problem in the L_1 metric which has taken place since the writing of this paper: Clarkson, Kapoor, and Vaidya [CKV] have obtained an $O(n \log^2 n)$ algorithm for computing the shortest path according to the L_1 metric among a collection of polygonal obstacles. Their method is that which was suggested in Section 2, namely that of finding a very sparse subgraph of the grid graph which works for finding shortest paths. They are able to show that it suffices to consider $O(n \log n)$ crossing points of the bullet paths.

Acknowledgment

This research was partially supported by a grant from the Hughes Aircraft Company. The author is affiliated with the Hughes Artificial Intelligence Center in Calabasas, California. The author would like to thank his advisor Christos Papadimitriou and D. T. Lee for helpful discussions and suggestions on these problems, and to thank Bernard Chazelle for pointing out the use of his data structures [Ch2] for solving segment dragging queries in linear space. Many fruitful discussions on the density of trapezoid-free matrices were with Esther Arkin, Dan Bienstock, Alan Hoffman, Offer Kella, Ed Schmeichel, and Karel Zikan.

References

[AAGHI] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai, "Visibility-Polygon Search and Euclidean

Shortest Paths", *Proc. 26th Annual Symposium on Foundations of Computer Science*, 1985.

[BG] D. Bienstock and E. Györi, Bellcore, Private Communication, 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, "Slimming Down Search Structures: A Functional Approach to Algorithm Design", *Proc. 26th Annual Symposium on Foundations of Computer Science*, 1985.

[Ch3] B. Chazelle, Private communication, February, 1986.

[CKV] K. Clarkson, S. Kapoor, and P.M. Vaidya, "Rectilinear Shortest Paths Through Polygonal Obstacles in $O(n \log^2 n)$ Time", To Appear, *Third ACM Conference on Computational Geometry*, June 1987.

[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] Dijkstra "A Note On Two Problems in Connection With Graphs", *Numerische Mathematik*, 1959.

[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 J. Comput.*, 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.

[Le] D.T. Lee, "Proximity and Reachability in the Plane", Ph.D. Thesis, Technical Report ACT-12, Coordinated Science Laboratory, University of Illinois, Nov. 1978.

[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.

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

[LW] T. Lozano-Perez and M.A. Wesley, "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles", *Communications of the ACM*, 22 (1979), pp. 560-570.

[Mi] 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.)

[MMP] J.S.B. Mitchell, D.M. Mount, and C.H. Papadimitriou, "The Discrete Geodesic Problem", Technical Report, Department of Operations Research, Stanford University, 1985. (To Appear: *SIAM Journal of Computing*, 1987).)

[MP] J.S.B. Mitchell and C.H. Papadimitriou, "The Weighted Region Problem", Technical Report, Department of Operations Research, Stanford University, 1985.

[MS] J.S.B. Mitchell and E. Schmeichel, "On the Density of Sparse Matrices", Manuscript in preparation, School of Operations Research and Industrial Engineering, Cornell University, 1987.

[Mo] D.M. Mount, "Voronoi Diagrams on the Surface of a Polyhedron", Technical Report 1496, Department of Computer Science, University of Maryland, 1985.

[Pa] C.H. Papadimitriou "An Algorithm for Shortest-Path Motion in Three Dimensions", *Information Processing Letters*, 20 (1985), pp. 259-263.

[Pr] F.P. Preparata, "A New Approach to Planar Point Location", *SIAM J. Comput.*, 10 (1981), pp. 473-482.

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

[RS] J.H. Reif and J.A. Storer, "Shortest Paths in Euclidean Space with Polyhedral Obstacles", Technical Report CS-85-121, Computer Science Department, Brandeis University, April, 1985.

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

[SS] M. Sharir and A. Schorr, "On Shortest Paths in Polyhedral Spaces", *SIAM Journal of Computing* Vol. 15, No. 1, pp. 193-215, February 1986.

[We] E. Welzl, "Constructing the Visibility Graph for n Line Segments in $O(n^2)$ Time", *Information Processing Letters*, Vol. 20 (1985), pp. 167-171.

[WWW] P. Widmayer, Y. F. Wu, and C. K. Wong, "Distance Problems in Computational Geometry with Fixed Orientations", *Proc. First Annual ACM Conference on Computational Geometry*, 1985.

Joseph S. B. Mitchell

Assistant Professor, Operations Research and Industrial Engineering **Cornell**

356 Upson Hall, 607/255-9148

B.S., M.S. 1981 (Carnegie-Mellon); M.S., Ph.D. 1986 (Stanford)

While a student at Carnegie-Mellon, Mitchell won the Andrew Carnegie Society award for outstanding scholarship. At Stanford he received the Howard Hughes Doctoral Fellowship. Mitchell worked for the Hughes Artificial Intelligence Research Center for five years, and joined the Cornell faculty in 1986.

Selected Publications

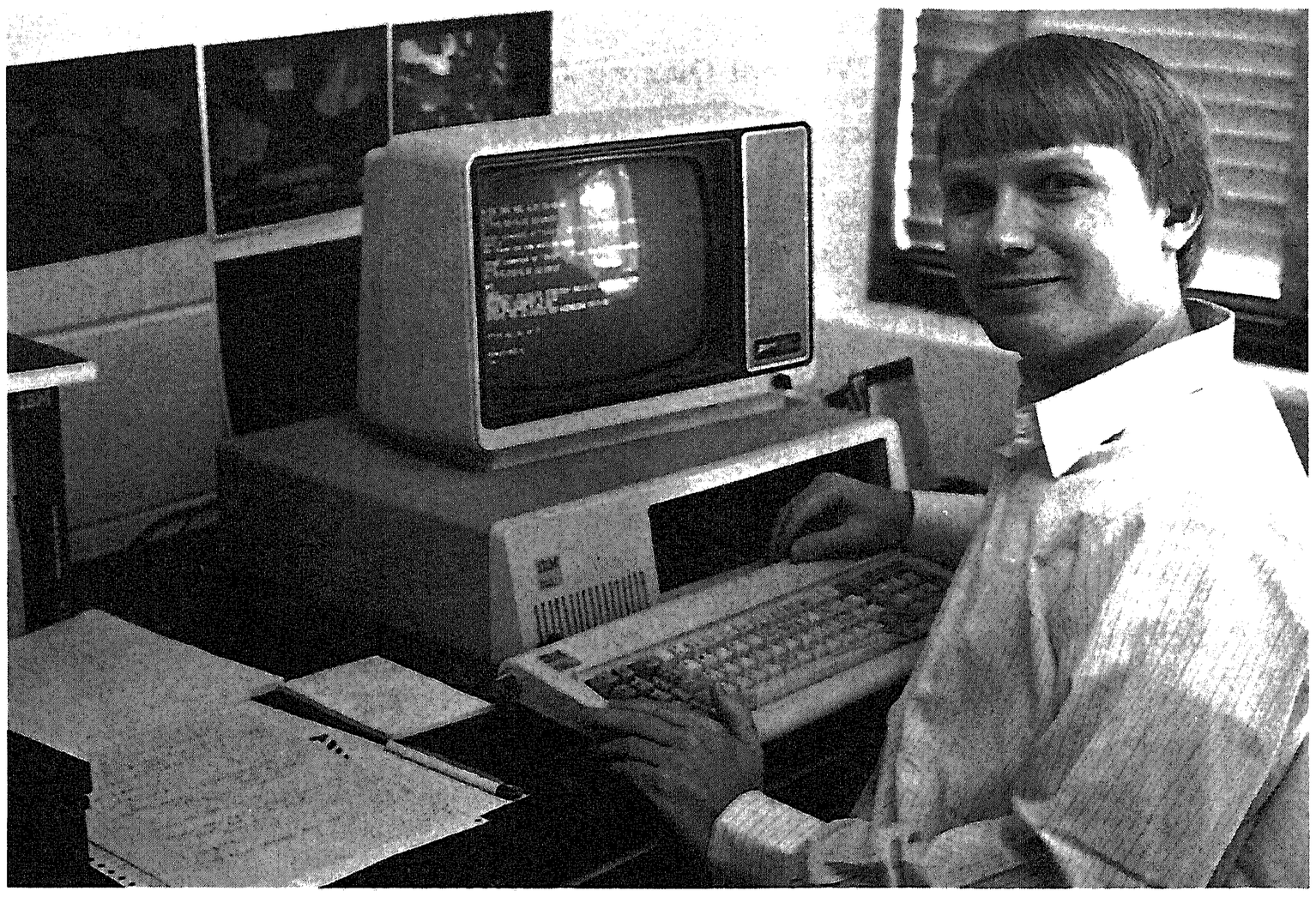
Mitchell, J. 1984. An autonomous vehicle navigation algorithm. In *Applications of artificial intelligence,* pp. 153-58. Washington, D.C.: Society of Photo-Optical Instrumentation Engineers.

Mitchell, J. 1985. Planning shortest paths. Paper read at Society for Industrial and Applied Mathematics Conference on Geometric Modeling and Robotics, 15-18 July 1985, in Albany, NY.

Mitchell, J. 1986. 98%-effective lot-sizing for one-warehouse multi-retailer inventory systems with backlogging. *Operations Research* in press.

Mitchell, J., D. Mount, and C. Papadimitriou. 1986. The discrete geodesic problem. *SIAM Journal of Computing* in press.

Mitchell, J., and C. Papadimitriou. 1986. *The weighted region problem.* Department of Operations Research technical report. Stanford, CA: Stanford University.



My primary research interests are in the fields of computational geometry, motion planning, autonomous vehicle control, and geometric reasoning. I am also interested in combinatorial optimization, queueing theory, and inventory control.

Our research in motion planning centers on finding a path for a robot through an environment cluttered with obstacles. Our approach is to impose optimality criteria on robot motion and try to find the shortest path from start to goal. Even a two-dimensional representation is challenging, and when the robot is capable of rotation the difficulty is greater. We are examining ways of defining the criteria for optimality in order to make the problem tractable and provide a useful solution.

An important application of motion planning is autonomous vehicle control; for example, moving a robot in an unknown environment, relying on sensory data. We are studying the related area of terrain navigation, particularly map representation. In order to successfully navigate unknown or cluttered terrain, we need to store and use sensed data to make a meaningful "map" that can be understood using geometric reasoning.

Moving a Disc between Polygons

Hans Rohnert

FB 10, University of Saarbrücken, West Germany

Abstract: An algorithm is given for finding a collision free path for a disc between a collection of polygons having n corners in total. The polygons are fixed and can be preprocessed. One query specifies the radius r of the disc to be moved and start and destination point of the center of the disc. The answer whether a feasible path exists is given in time $O(\log n)$. Returning a feasible path is done in additional time proportional to the length of the description of the path. Preprocessing time is $O(n \log n)$ and space complexity is $O(n)$.

1. Introduction

In this paper we consider a 2-dimensional variant of the piano mover's problem: Given is a collection of simple, disjoint polygons with a total of n corners. They are assumed to be lying in the Euclidean plane and to be figuring as obstacles for a moving circular disc. Answer on-line questions of the form: Can a disc of radius r be moved from start point a to destination point b without colliding with the obstacles? An efficient solution for this relatively simple problem is desired since algorithms for complicated shapes of robots are slow and since on the other hand such a robot can be approximated by an enclosing disc. Hence our algorithm can be used as an initial step. If it can not output a feasible path, i.e., a path being continuous and collision free, the user should struggle with more details of the robot and change to more expensive algorithms. Yet, since our algorithm outputs also the maximum width of the bottlenecks of all conceivable paths the user can refrain directly from more expensive algorithms if the robot can not pass such a bottleneck whatever are the shapes of the bottleneck and of the robot.

To answer queries quickly we preprocess the obstacle space and answer thereafter one query for specified radius, start and destination point in time $O(\log n)$ if a yes/no-answer is required and in time $O(k+\log n)$ if the explicit output of a feasible path is required. Here k is the length of the description of the returned path. Often $k \ll n$. In any case, $k = O(n)$. We do not find a shortest path, rather just a path avoiding the obstacles. But due to the construction the returned path has another useful property which may be important for practical applications: The minimum distance between the disc and the obstacles during the motion is maximal among all feasible paths. Space complexity is $O(n)$. The preprocessing is done in time $O(n \log n)$.

As pointed out by Ó'Dúnlaing and Yap [7] there exists a collision free path for a disc between points a and b iff a movement restricted to the Voronoi diagram of the polygons (except the initial and final phase) is possible. They give an algorithm for moving a disc by first constructing the Voronoi diagram (*VD* for short). One query takes $O(n)$ time for moving from a to a point a' on *VD*, searching for a suitable path on *VD* from a' to b', and then moving from b' to b.

Our algorithm is also based on *VD* but does not search it completely for a path. Rather we extract from it a maximum spanning tree. Therefore we define a weight function on the edges of *VD*. The weight of an edge of *VD* is the minimum distance over all points on the Voronoi edge to the neighboring polygons. A maximum spanning tree is built by starting with the vertices of *VD* as leaves. Components of disjoint sets of Voronoi vertices are connected in decreasing order of the weight of the Voronoi edges running between these components. The interior nodes of the maximum spanning tree are labeled with the selected edges. A Union-Find procedure helps us to construct efficiently the maximum spanning tree in bottom-up fashion. After this preprocessing one query consists of locating a and b in the obstacle space by means of point location in time $O(\log n)$ and then computing the nearest common ancestor (or *nca* for short) of two Voronoi vertices easily reachable from a resp. b. If its label is greater than the radius r of the disc and if the initial

and final phase consisting of moving towards VD and away from VD are manageable then a collision free movement of a disc with radius r from a to b is possible. To describe a feasible path we determine recursively the nca's of the two subpaths from a to the "left" endpoint of $nca(a,b)$ and from the "right" endpoint of $nca(a,b)$ to b and combine the returned edges into a directed path.

The paper is organized as follows. In Section 2 we give some preliminaries. In Section 3 we present the algorithm of Ó'Dúnlaing and Yap [7] and our idea in contrast. The details of our algorithm follow in Section 4. Section 5 is devoted to further questions such as generalizing to arbitrary, intersecting polygons.

2. Preliminaries

The *piano mover's problem* is to move continuously a body within a subspace R of $\mathbb{R}^n$ without colliding with the bordering obstacles. In our specific application we restrict to the 2-dimensional Euclidean plane. The moving body is a rigid disc D with radius r. The 2-dimensional open region R in which D can move is bounded by a set F of disjoint, simple polygons. To simplify the exposition we call the open line segments of the polygons *walls*. A meeting point of two walls is a *corner*. The elements of the Voronoi diagram are called *vertices* and *edges*. The elements of the auxiliary trees constructed in the preprocessing are called *leaves*, *nodes* and *links*.

F has n corners and n walls connecting the corners. We assume that the walls are directed line segments so that the interior of the considered polygon is to the right of its walls. For algorithmic treatment we assume that the whole scenery is included by a large triangle Δ. Later on we will discuss the question how large Δ should be. Then we can view R as a polygonal region with holes. Our task is to preprocess R so that a continuous movement of a disc D without collisions can be found. Inputs to one query are the radius r of D and the initial placement a and the final placement b of the center c of D. We answer two types of queries where the latter is an extension of the former:

(i) Is there a path complying with the specified restrictions (and return the bottleneck, if desired)?

(ii) If "Yes", return such a path.

Therefore we use the generalized Voronoi diagram $VD(F,R)$. We define $VD(F,R)$ as the set of points p such that at least two points of the bordering polygons are nearest to p. Distance is measured in Euclidean metric, i.e., the distance of point x from F is $d(x,F) = \min\{dist_2(x,y);\ y \in F\}$, where $dist_2(x,y)$ is the Euclidean distance of points x and y. Note that $d(x,F)$ is always well-defined in our application. As we argue in Section 4, $VD(F,R)$ can be constructed in time $O(n \log n)$ and linear space.

For a formal definition of $VD(F,R)$ we need some more notation. For a point $x \in R$ we define $Near(x)$ as the set of points in F nearest to x: $Near(x) = \{y \in F;\ \forall y' \in F : dist_2(x,y') \geq dist_2(x,y)\}$. Then $VD(F,R) = \{x \in R;\ |Near(x)| \geq 2\}$.

The points where at least 3 bisectors of walls and corners meet form the set of Voronoi vertices V, i.e, $V = \{x \in R;\ |Near(x)| \geq 3\}$. If E denotes the set of Voronoi edges, i.e., $E = \{x \in R;\ |Near(x)| = 2\}$, we can write $VD(F, R) = (V, E)$. This notation is chosen to suggest that $VD(F, R)$ is a planar graph (disregarding the dangling ends of Voronoi edges leading into corners, see Section 4 for an illustration). The weight of a Voronoi edge e is defined as $weight(e) = \min\{d(x, F);\ x \in e\}$ where $x \in e$ means that x is a point lying on e including its endpoints.

Let $Near(x) = \{p\}$. Then $Projection(x)$ is the point where the semi-infinite ray from p through x meets $VD(F, R)$, i.e., this ray shows how to come away from the nearest polygon towards $VD(F, R)$. If $|Near(x)| \geq 2$ then $Projection(x) = x$. Now we can state one property needed in the proof of correctness for the query algorithm.

Lemma 1. *Let $x \in R$. If $Projection(x) \neq x$ then $d(y, F)$ increases strictly when y moves on the line segment from x towards $Projection(x)$.*

Proof: For any point y on the directed line segment from $Near(x)$ through x to $Projection(x)$ it is easy to see that $Near(x) = Near(y)$ if $y \neq Projection(x)$ and $Near(x) \subseteq Near(y)$ otherwise. Hence $d(y, F) > d(y', F)$ if y' lies on this line segment between $Near(x)$ and y. ∎

3. Basic idea

O'Dúnlaing and Yap [7] propose roughly the following query algorithm:

(1) **if** $d(a, F) \leq r$ **or** $d(b, F) \leq r$
  then Halt: "Unfeasible start or destination point";
(2) move from a to $Projection(a)$;
(3) search $VD(F, R)$ for a feasible path from $Projection(a)$ to $Projection(b)$;
  if no such path exists
  then Halt: "There is no way for a disc of radius r from a to b";
(4) move from $Projection(b)$ to b.

The correctness of this algorithm relies on

Lemma 2. *Let a resp. b be a feasible start resp. destination point for disc D with radius r. There is a feasible path from a to b in R iff there is a feasible path from $Projection(a)$ to $Projection(b)$ restricted to $VD(F, R)$.*

Proof: "⇒": Let p be a feasible path from a to b in R. If we replace every point x on p by $Projection(x)$ we get a continuous path p' restricted to $VD(F, R)$. Since $d(Projection(x), F) \geq d(x, F)$ for every x this path is feasible if p is.
"⇐": Let q be a feasible path from $Projection(a)$ to $Projection(b)$. We form path q' by adding the line segment from a to $Projection(a)$ to the front and the line segment

from b to *Projection*(b) to the end of q. Path q' is feasible since a and b are feasible locations for the center of D and since the distance to F increases along the added line segments by Lemma 1. ∎

The complexity of this algorithm is determined as follows. For computing $d(a, F)$ and $d(b, F)$ the distances of a and b to all walls and corners are determined and the minima selected. *Projection*(a) is computed by intersecting the semi-infinite ray from *Near*(a) through a with all curve segments of $VD(F, R)$ if *Near*(a) is a singleton set. Otherwise *Projection*(a) = a. In the same way *Projection*(b) is computed. Step (3) is essentially a simple graph exploration and runs in time proportional to the size of the underlying graph which is $VD(F, R)$. Since $VD(F, R)$ has size $O(n)$ as we prove in Section 4 the query algorithm of Ó'Dúnlaing and Yap [7]runs in time $O(n)$.

If we want to reach sublinear query time we can not afford to compute *Near* and *Projection* by considering all walls and corners and the whole $VD(F, R)$. Instead we must take care to determine quickly the neighborhood of a and b for computing *Near*(a) and *Near*(b) and restrict the search for a path onto a suitable subgraph of $VD(F, R)$ containing "enough" feasible paths. This subgraph is the maximum spanning tree of $VD(F, R)$. Correctness of this approach is proven in the next section.

4. The details of the algorithm

This section is devoted to the details of our algorithm and to the proof of correctness. First we present the algorithm that does the preprocessing and builds the data structures used later on to speed up the queries.

(1) Construct $VD(F, R)$ and represent it in an appropriate form;
(2) Build a generalized planar subdivision for the planar graph consisting of F and $VD(F, R)$;
(3) Extract from $VD(F, R)$ a maximum spanning tree $MST(VD(F, R))$;
(4) Build from $MST(VD(F, R))$ auxiliary trees according to Harel and Tarjan [3] for computing nearest common ancestors.

Now we explain the preprocessing algorithm. There are different approaches to construct $VD(F, R)$ in time $O(n \log n)$, see the papers of Fortune [2], Kirkpatrick [4] and Yap [11] for details. The primitives for which $VD(F, R)$ is constructed are open line segments (the walls) and points between them (the corners). The output of the algorithms computing generalized Voronoi diagrams consists of two parts: the internal and the external Voronoi diagram. The internal Voronoi diagram describing maximal distances inside the polygons is not used by our algorithm, and hence it can be dropped. The external Voronoi diagram ($VD(F, R)$ in our application) describes the partition of the immediate interior of Δ into Voronoi cells. In contrast to the

Voronoi diagram for point sites $VD(F, R)$ does not consist of straight line segments only, but includes also parabolic curve segments. Yet all edges of $VD(F, R)$ are computationally simple in the sense that constant time suffices to decide whether a point is right or left a directed edge. In Figure 1 the polygons bounding F are shown by heavy line segments. The Voronoi edges are shown by the slim and by the dotted curve segments. The slim curve segments are the Voronoi edges due to two primitives belonging to two different polygons. The dotted curve segments are the Voronoi edges due to two primitives belonging to the same polygon. The dots represent the Voronoi vertices.

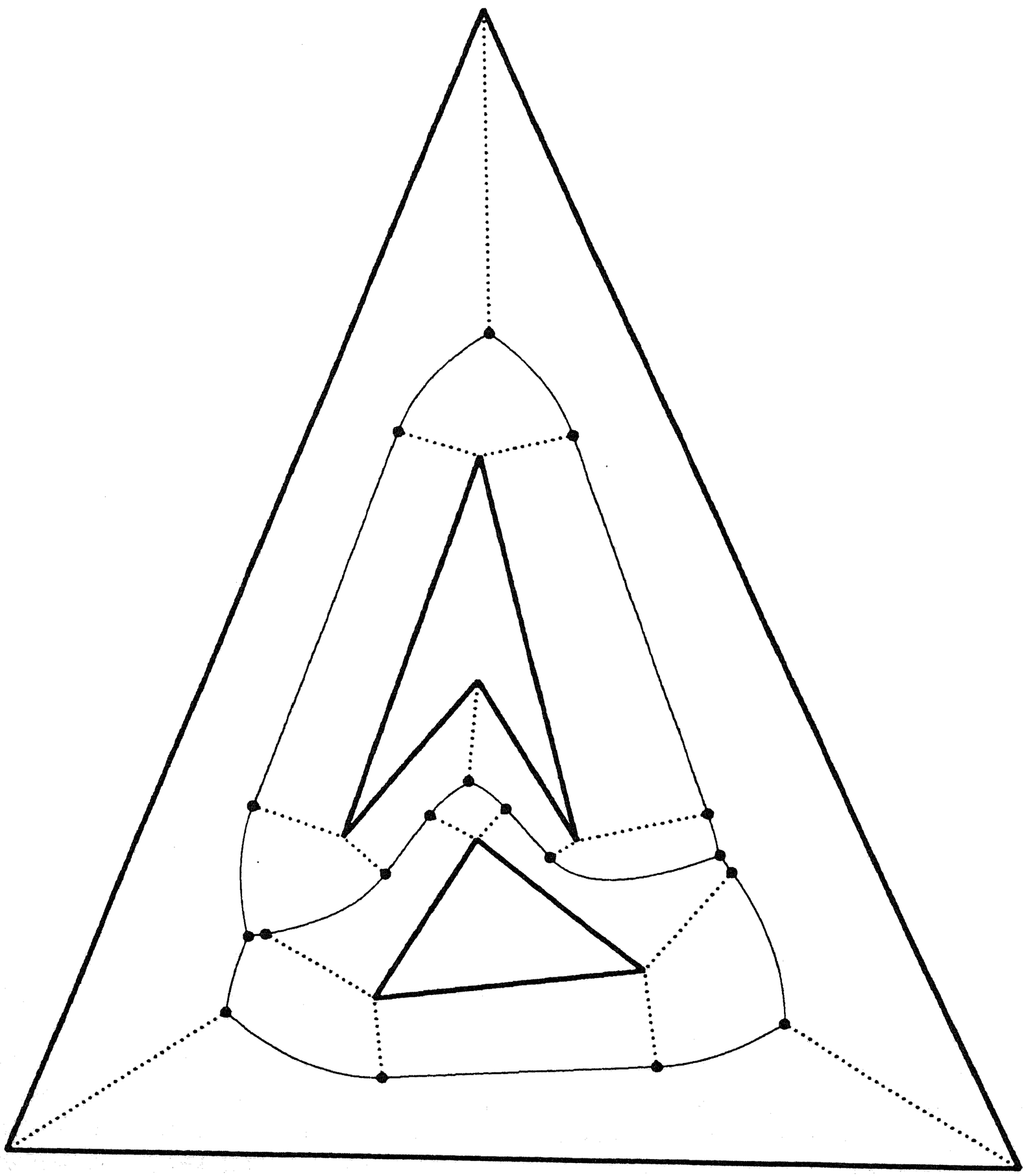


Fig. 1. Polygonal region with external Voronoi diagram

Lemma 3. *For $n \geq 6$ $VD(F,R)$ has at most $2n-3$ faces, at most $6n-15$ edges and at most $4n-10$ vertices.*

Proof: We start with $n = 6$ since in this case the scenery consists of $\triangle$ and one triangle in its interior. The cases of $n < 6$ give no sense because we do not consider isolated points and line segments as polygons. Working with the described primitives and using the Euclidean metric every corner and every wall gives rise to at most one Voronoi cell (or face in the language of topology). More precisely, every convex corner and every wall gives rise to exactly one face, in contrast to the concave corners having no related faces. Furthermore $\triangle$ adds 3 instead of 6 faces because its corners are concave with respect to the scenery. This gives the bound on the number of faces. Now we consider the geometric dual $D(VD(F,R))$ of $VD(F,R)$. It can be constructed as follows. Every face of $VD(F,R)$ is represented by one point in its interior. Two points are connected if their faces share an edge. Note that it is impossible that two faces share more than one edge. Since the dual of a planar graph is planar $D(VD(F,R))$ has at most $3(2n-3)-6 = 6n-15$ edges. This bound holds also for the number of edges of $VD(F,R)$ since they are in one to one correspondence to the edges of $D(VD(F,R))$. The Voronoi vertices are points where at least 3 Voronoi edges meet. If we count in each vertex all outgoing edges then we count each edge twice. Hence the number of Voronoi vertices is at most $\frac{2}{3}(6n-15) = 4n-10$. ∎

The representation of $VD(F,R)$ to be output by the algorithm constructing it is as follows. Since the interior of a Voronoi cell is the locus of all points nearest to the uniquely defined primitive giving rise to this cell it is natural to label each cell with the name of the related primitive. A cell is described by the sequence of bounding edges in, say, clockwise order. Crosspointers are erected between the two occurrences of each edge. An edge is described by its weight, its two endpoints, i.e., two Voronoi vertices with its coordinates, an algebraic equation describing it as a curve in the plane and the two primitives causing it.

We are facing one more difficulty: one primitive can share many Voronoi edges with other primitives and hence we might get trouble to determine quickly *Projection*(x) for some x, also if the Voronoi cell in which x lies and hence *Near*(x) are known. To overcome this we divide (conceptually) each cell into an ordered sequence of sectors on which binary search is possible. For a cell c and one of its bounding edges e the sector $s(c,e)$ is the locus of all points $x \in c$ for which *Projection*$(x) \in e$. Since *Projection* is uniquely defined these sectors form a disjoint partition of c. We have to distinguish two cases.

Case 1: Cell c belongs to a convex corner v.
Then the sectors have the form of triangles and are separated by rays starting at v and ending in Voronoi vertices. The third side of such a sector is the edge giving rise to that sector. Each sector is connected since the edges bounding c are continuous and non-overlapping from the viewpoint of v. We number these sectors in that order as a ray scanning clockwise between the two bisectors that start in v and are perpendicular on its two adjacent walls meets the edges.

Case 2: Cell c belongs to a wall w.
Then the sectors have a quadrangular form and are separated by rays starting in perpendicular direction from w. The third side of such a sector is the edge giving rise to that sector and the fourth side is the part of w between the two rays bounding that sector. These sectors are also connected and are ordered just as their fourth parts partition the directed line segment w. ∎

Since each edge gives rise to two sectors the number of sectors is bounded by $12n - 30$. It is not necessary to store explicitly the sequences of sectors; rather it suffices to attach to each primitive its ordered list of edges that form the third sides of its sectors. If these lists permit random access it is possible to determine by binary search in time logarithmic in the length of the considered list which edge is hit when drawing a ray from $Near(x)$ through x for some query point x. The attachment of these lists to the primitives is easily done in time $O(n)$ when $VD(F, R)$ is given.

For a query we first want to compute $Near(a)$ and $Near(b)$. This is done by locating a and b in $VD(F, R)$. The returned Voronoi cells give the names of the two primitives nearest to a resp. b. If a or b lies on $VD(F, R)$ then only the edge or the vertex of $VD(F, R)$ on which that point lies should be returned. Fast location of a and b in $VD(F, R)$ is possible by using the generalized planar subdivision proposed by Edelsbrunner [1]. The edges of the subdivision are formed by the walls and the Voronoi edges. To make the subdivision y-regular in the sense of [1] auxiliary vertices and edges are added to eliminate y-extreme vertices and edges not being y-monoton. This transformation takes time $O(n \log n)$ and is described in the papers of Edelsbrunner [1] and of Lee and Preparata [5]. The space complexity is not increased by addition of these "few" vertices and edges and remains $O(n)$. Also the used search structure, the layered chain tree, takes $O(n)$ space and $O(n)$ time for construction. Then location of a point x in the generalized planar subdivision by using the layered chain tree takes $O(\log n)$ time. As described in the last paragraph we can compute by binary search $Projection(a)$ and $Projection(b)$ after locating a and b in the generalized subdivision and computing $Near(a)$ and $Near(b)$. This takes also time $O(\log n)$ since the total length of the lists attached to the primitives is $O(n)$. Note that it is possible but not necessary to refine the subdivision by decomposing the cells into sectors.

The third preprocessing step consists of the computation of a maximum spanning tree $MST(VD(F, R))$ for $VD(F, R)$. The definition of a *maximum spanning tree* is as follows. Its vertex set is identical with the set of Voronoi vertices. Its edge set is a subset of the Voronoi edges with two properties:

a) The vertices and the subset of the edges together form a tree. Such a tree is called a *spanning tree*.

b) If W is the weight of a *maximum* spanning tree where the weight of a tree is the sum of the weights of its edges then $W \geq W'$ for the weight W' of every other spanning tree for $VD(F, R)$.

If the edge weights are not pairwise disjoint then maximum spanning trees are not necessarily unique. Yet this does not cause trouble since we are content with any

maximum spanning tree. We want only to emphasize that $MST(VD(F,R))$ is that maximum spanning tree we find in step (3) of the preprocessing algorithm.

We represent $MST(VD(F,R))$ internally by a so called *edge tree* and build this tree in bottom-up fashion. The leaves of the edge tree are all lined up on the same level and in one to one correspondence to the vertices of $VD(F,R)$. Each leaf is labeled with the name of one vertex. We combine leaves into larger and larger components until we have one single tree. The order of combinations is determined by the weight of the Voronoi edges connecting components. We consider the Voronoi edges in decreasing order of their weight and add a new node to the edge tree if the considered edge connects two different components of vertices. The new node is labeled with the two endpoints and the weight of the considered edge. Figure 2 shows a maximum spanning tree in simplified form (edges are drawn as straight line segments and vertices as labeled circles) and its associated edge tree.

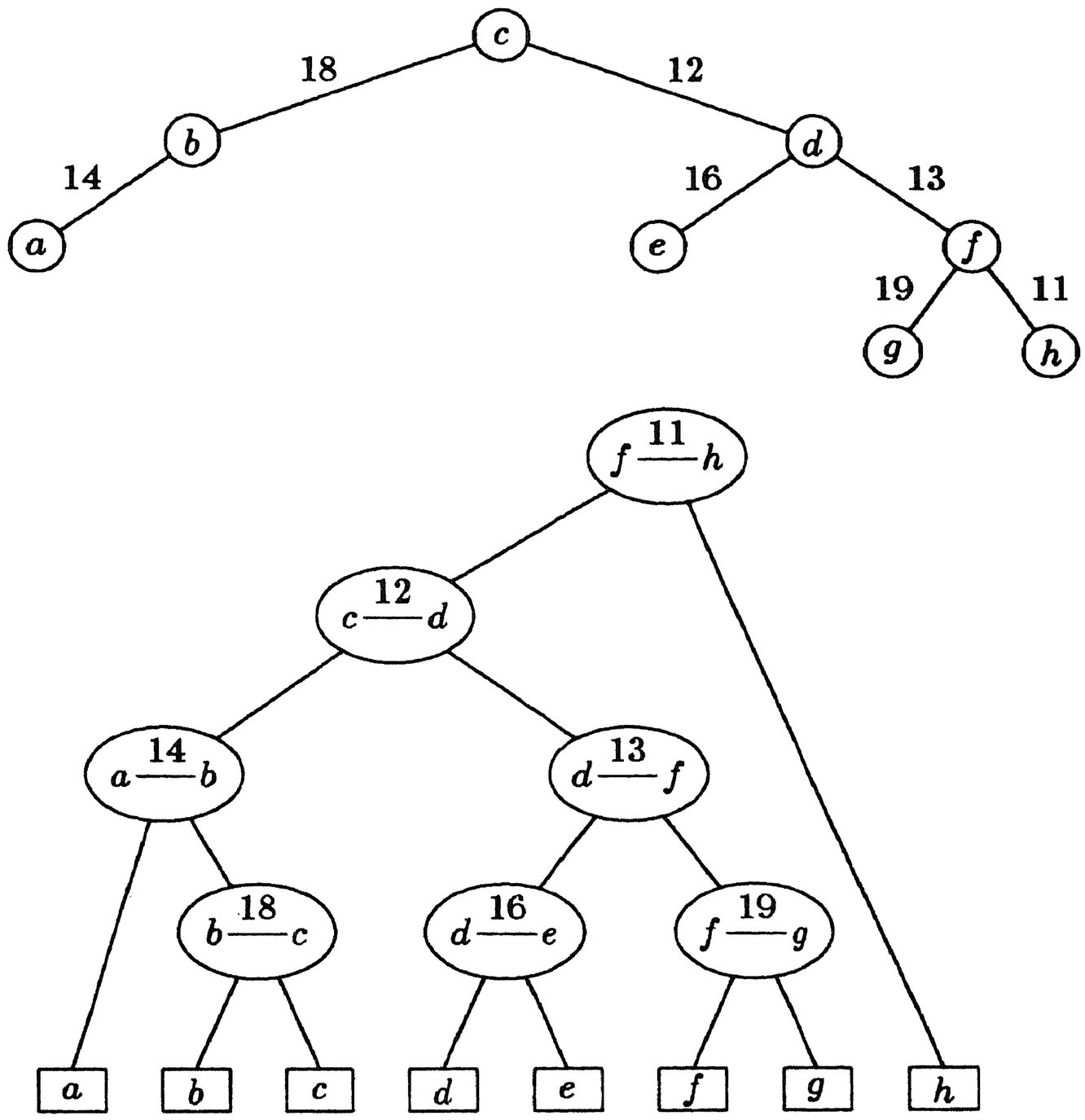


Fig. 2. *MST* and edge tree

To build the edge tree in an efficient manner, we sort firstly the Voronoi edges according to their weight. Then we use the well-known Union-Find data structure of Tarjan [10] to decide if the considered edge combines two components of Voronoi

vertices or if it runs between two Voronoi vertices that are already in the same component. For simplicity this data structure is maintained separately only for the sake of returning the identification of the roots of the one or two subtrees representing the two components, i.e., the edge tree is not concerned by path compression and not by making the root of the component with fewer nodes to a son of the root of the other component. If the two Finds for the two endpoints of the considered edge return the same identification then the edge is not added to the edge tree. Otherwise we add a new node to the edge tree as described above. Now we prove that the result of the described procedure is indeed a maximum spanning tree.

Lemma 4. *The constructed edge tree describes a maximum spanning tree for* $VD(F,R) = (V,E)$.

Proof: We show by induction on the sequence of the combination steps that there is a maximum spanning tree T containing all edges added so far to the edge tree. This is clearly true before any edge is added. Assume now that we have a set of trees, each one describing one component of Voronoi vertices and edges belonging to T. If the next considered edge (v_i, v_j) runs inside one component it is not added to the edge tree and there is nothing to show. If (v_i, v_j) is added to the edge tree and if $(v_i, v_j) \in T$ the induction step is finished. If (v_i, v_j) is added and $(v_i, v_j) \notin T$ then there is a cycle in $T \cup \{(v_i, v_j)\}$ and an edge (v'_i, v'_j) on this cycle leaving one of the two components combined by (v_i, v_j). $T \cup \{(v_i, v_j)\} - \{(v'_i, v'_j)\}$ is also a spanning tree. Since $\mathit{weight}((v_i, v_j)) \geq \mathit{weight}((v'_i, v'_j))$ by the order in which we process the Voronoi edges $T \cup \{(v_i, v_j)\} - \{(v'_i, v'_j)\}$ is also a *maximum* spanning tree. Since the resulting edge tree is binary by construction and has $|V|$ leaves it has $|V|-1$ interior nodes. Hence the labels of its interor nodes are the $|V|-1$ edges of a maximum spanning tree. ∎

The time complexity of preprocessing step (3) is dominated by the time bound of $O(n \log n)$ for sorting the edges according to their weight. The operations on the maximum spanning tree under construction take linear time. The complexity to perform $c \cdot n$ Unions and $d \cdot n$ Finds for constants c and d is $O(n \cdot \alpha(n,n))$ where α is a functional reverse of Ackermann's function. Here $c \leq 4$ and $d \leq 6$ since the number of Unions is equal to $|V|-1 \leq 4n-11$ and the number of Finds is bounded by $|E| \leq 6n-15$. The most remarkable property of α is that it increases extremely slowly. A simpler Union-Find method is also available and takes time $O(n \log n)$, see Mehlhorn [6] and Tarjan [10]. Space complexity of all data structures is $O(n)$.

To answer a query if a feasible path between two vertices of $VD(F,R)$, say a' and b', exists we compute the bottleneck of that path from a' to b' whose minimal distance to the obstacles is maximal. Since $VD(F,R)$ consists of the paths of maximal distance to the obstacles and $MST(VD(F,R))$ contains for every pair of vertices exactly the path of maximal distance we have only to determine the bottleneck of the path from a' to b' in $MST(VD(F,R))$. We prove formally the last property in

Lemma 5. *Let* $G = (V,E)$ *be a graph,* s *and* t *two arbitrary vertices of* G *and*

$T = (V, E')$ a maximum spanning tree for G. Then the minimal edge weight on the unique path in T from s to t is maximal over all paths from s to t in G.

Proof: We denote the path in T from s to t with p and the minimal edge weight along this path with $minw(p)$. Assume that there is a path p' from s to t in G but not entirely in T with $minw(p') > minw(p)$. Let $e' = (x', y')$ be the edge on p' with $weight(e') = minw(p')$. Accordingly, we define $e = (x, y)$. We can divide the vertices of p' into two subsets A and B such that all vertices of A are connected with x via tree edges without using e and all vertices of B are connected with y via tree edges without using e, and there is an edge $e'' \in p'$ connecting A and B that is not in T since there would be a circle in T otherwise. Replacing e by e'' gives another spanning tree T' with $weight(T') > weight(T)$ since $weight(e'') \geq weight(e') = minw(p') > minw(p) = weight(e)$. This is a contradiction to the maximality of T. The lemma follows. ∎

Since the edge tree describes the links between components of $MST(VD(F, R))$ in from bottom to top decreasing order of edge weights the labeling of the nearest common ancestor of a' and b' gives the minimal distance to the obstacles during the journey from a' to b'. If the labeling of $nca(a', b')$ is not greater than the radius of the disc to be moved then there is no feasible path from a' to b'.

The computation of $nca(a', b')$ in the edge tree is done by applying the algorithm of Harel and Tarjan [3]. They consider the problem of computing nearest common ancestors in an arbitrary tree T. By constructing in linear time and space a so called compressed tree and a balanced binary tree they convert the original problem into an *nca*-problem on a complete binary tree. Since the nearest common ancestor of two nodes in a complete binary tree can be determined by direct calculation this problem takes constant time on a RAM.

Now we have all data structures at hand to answer queries of type (i): is there a feasible path for disc D with radius r from point a to point b? First we locate in time $O(\log n)$ the two Voronoi cells containing a and b. Then we compute in constant time $Near(a)$, $Near(b)$, $d(a, F)$ and $d(b, F)$. If $d(a, F) \leq r$ or $d(b, F) \leq r$ or a or b lies inside a polygon the algorithm says "No" and halts. Otherwise we proceed and determine $Projection(a)$ and $Projection(b)$ by binary search on the sectors described above in time $O(\log n)$. If $Projection(a)$ lies on an edge e excluding its endpoints we choose one of its endpoints, say a', to move to as follows: First we determine in constant time one point $\min(e)$ on this edge with minimal distance to the obstacles, i.e., $\min(e) = \{x;\ \forall y \in e : d(x, F) \leq d(y, F)\}$. (This point is not unique if e runs between two parallel walls. In this case we choose any one of the endpoints of e as $\min(e)$. If e ends in a corner v of a polygon we set $\min(e) = v$.) If $\min(e)$ is unique it is one of the endpoints of e or the apex of a parabolic curve segment. We choose a' so that $Projection(a)$ lies between $\min(e)$ and a' on e. Since distance d does not decrease from $\min(e)$ towards a' it is safe to move the disc from $Projection(a)$ along e towards a'. Analogously we determine b'.

Now we can look for a feasible path between vertices a' and b'. Such a path exists iff the label of $nca(a', b')$ is greater than r. This condition can be checked in

constant time with help of the data structures described above.

To answer queries of type (ii), i.e., to output a feasible path from a to b if it exists, we check its existence and compute two Voronoi vertices a' and b' as described above. The path p from a' to b' scattered over the edge tree is composed by the recursive procedure *findpath*. We impose a successor relation on p by directing it from a' to b'.

procedure *findpath*(x, y);
if leaf x is right of leaf y **then** exchange x and y;
$(r, q) \leftarrow nca(x, y)$;
successor$(r) \leftarrow q$;
predecessor$(q) \leftarrow r$;
if $r \neq x$ **then** *findpath*(x, r);
if $q \neq y$ **then** *findpath*(q, y);
end.

The call *findpath*(a, b) returns p twice, once starting in a' and going via the successor relation to b' and once starting in b' and going via the predecessor relation to a'. The running time of *findpath*(a', b') is proportional to the number of edges of p which in turn is bounded by $|V| - 1 \leq 4n - 11$.

5. Concluding Remarks

Our algorithm solves the problem of finding paths of maximal distance to the obstacles. These paths are relatively safe for the moving robots but tend to be long. If for example the two query points lie close to another in a large room then the algorithm may move the disc first to the middle of the room and then to the destination point. Here local optimization strategies can be helpful.

How large should Δ be? Its minimal distance to the original polygons must surpass twice the maximal value of d along the Voronoi edges between the original polygons. If we denote these values with Δ_{min} and d_{max} we can formulate this condition as $\Delta_{min} > 2 \cdot d_{max}$. This is necessary because a disc must be free to move around the polygons if it can not move between them. The 2 is due to the fact that d_{max} refers to the radius and Δ_{min} to the diameter of the moving disc. If the query algorithm outputs as bottleneck a value greater than d_{max} then the returned path goes around the polygons and is nearly feasible whatever is the (artificial) value of Δ_{min}. One has only to push the disc away from the Voronoi diagram if its diameter surpasses Δ_{min}.

One possibility to generalize the setting is to allow intersecting polygons. Then we determine before the described preprocessing the contour by using an algorithm of Ottmann, Widmeyer and Wood [8] and replace the original polygons by the contour. This takes time $O((n + s) \log n)$ where s is the number of intersection points. Note that the number of corners may become quadratic. Also the scenery breaks up into 2-dimensional connected components. For each one of them the

data structures described in Section 4 must be constructed. A query has first to determine if a and b lie in the same component and then to restrict to the data structures of this component.

6. Acknowledgements

I thank Kurt Mehlhorn for directing my interest onto this problem and Emo Welzl for helpful criticism. Stefan Schirra gains merits for valuable proofreading. Finally I beg our VAX's pardon for stressing it several hours in fabricating Figure 1.

7. References

[1] H. EDELSBRUNNER : "An Optimal Solution for Searching in General Planar Subdivisions", Technical Report 1983, Institutes for Information Processing, Technical University of Graz, Austria

[2] S. FORTUNE : "A Sweepline Algorithm for Voronoi Diagrams", Proc. of the Second Symp. on Computational Geometry 1986, pp. 313–322

[3] D. HAREL, R. E. TARJAN : "Fast Algorithms for Finding Nearest Common Ancestors", SIAM Journal on Computing, Vol. 13, No. 2, 1984, pp. 338–355

[4] D. G. KIRKPATRICK : "Efficient Computation of Continuous Skeletons", IEEE FOCS 1979, pp. 18–27

[5] D. T. LEE, F. P. PREPARATA : "Location of a point in a planar subdivision and its applications", SIAM Journal on Computing, Vol. 6, 1977, pp. 594–606

[6] K. MEHLHORN : "Data Structures and Algorithms", Vol. 1, pp. 296–304, published 1984 by Springer-Verlag, Berlin Heidelberg New York Tokyo

[7] C. Ó'DÚNLAING, C. K. YAP : "A 'Retraction' Method for Planning the Motion of a Disc", Journal of Algorithms, Vol. 6, pp. 104–111 (1985)

[8] T. OTTMANN, P. WIDMEYER, D. WOOD : "A Fast Algorithm for Boolean Mask Operations", Inst. für Angew. Mathematik und Formale Beschreibungsverfahren, D-7500 Karlsruhe, Rept. No. 112 (1982)

[9] M. I. SHAMOS : "Geometric Complexity", Proc. of the Seventh Annual ACM Symposium on Theory of Computing, pp. 224–233 (1975)

[10] R. E. TARJAN : "Efficiency of a good but not linear union algorithm", Journal of the ACM 22, pp. 215–225 (1975)

[11] C. K. YAP : "An $O(n \log n)$ Algorithm for the Voronoi Diagram of a Set of Simple Curve Segments", Technical Report No. 161, New York University, Dept. of Computer Science, May 1985

Personal Record

Hans Rohnert was born in Saarbrücken, West Germany, on May 18, 1959. He received the M.S. degree in Computer Science from the University of Saarbrücken in 1984. Currently, he is research assistant and Ph.D. candidate of Kurt Mehlhorn, also Saarbrücken. His research interests include data structures and efficient algorithms, especially computational geometry and motion planning.

He is a member of EATCS and GI, the union of the european theoretical computer scientists and of the german computer scientists, respectively.

Publications:

- "A dynamization of the all pairs shortest path problem", Proceedings of the 2nd Annual Symposium on Theoretical Aspects of Computer Science, LNCS 182, 1985, pp. 279–286.
- "Shortest Paths in the Plane with Convex Polygonal Obstacles", Information Processing Letters 23 (1986), pp. 71–76.
- "New Algorithms for Shortest Paths avoiding Convex Polygonal Obstacles", Technical Report A 86/02, FB 10, University of Saarbrücken.
- "Moving a Disc between Polygons", to appear.

Recent Advances in Algorithmic Motion Planning

Gordon Wilfong
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

Algorithmic motion planning has become an important problem since the rise in interest in automated manufacturing. Researchers have studied problems concerning planning the motion of an object in a known static environment, in a known changing environment, in an unknown environment and in a reconfigurable environment. This paper will survey some of the more recent results in the field. The reader should also see [Y2] and [SS4].

2. Motion Planning in a Static Environment

As in [LW] the configuration of a system of objects with k degrees of freedom can be represented by a k-tuple. The set of all such tuples is called *configuration space*. In any motion planning problem there are a set of constraints on the allowable configurations (e.g. no object may intersect any other object or obstacle). The configurations that satisfy these constraints will be called *free configurations*. The general problem is: Given a starting and desired final free configuration of a system of objects and a description of the static environment in which the system is to move (i.e. given the constraints), find a continuous motion of the objects (keeping the intermediate configurations free) from the starting to final configuration (if one exists, otherwise report that no such motion exists).

2.1. Upper bounds

The most general result for motion planning in the presence of a static environment is that due to Schwartz and Sharir [SS2] where they show that for any system of objects whose motion is to be planned in a given static environment where the set of free configurations of the objects forms a semi-algebraic set there is an algorithm to solve the motion planning problem in worst-case time which is doubly exponential in the number of degrees of freedom of the objects but polynomial in the number of algebraic constraints defining the free configurations and their maximum degree. This result is obtained by partitioning configuration space using a cell decomposition method due to Collins[Co] and computing adjacencies of the various cells. Recently Canny [Ca] has improved the result by developing a method that partitions configuration space more coarsely. Canny's method has a time bound which is singly exponential in the number of degrees of freedom.

There have been a large number of papers improving on the upper bounds for

special cases of the motion planning problem. Efficient algorithms have been developed for discs (e.g. [SS3],[OY],[Y1]), polygons (e.g. [SS1],[HW]), linkages (e.g. [HJW1], [HJW3]), ladders (e.g. [SiS], [KO], [LS]), rotating arms (e.g. [FWY]) and other objects.

2.2. Lower bounds

The first complexity result concerning motion planning appears in [R]. It is shown that deciding if there is a motion in a 3-dimensional environment between configurations of an object that consists of multiple polyhedra freely linked together is PSPACE-complete. The proof that the problem is PSPACE-hard is by reduction from the acceptance problem for polynomial space bounded Turing machines. The construction used for the proof produces an object which consists of a bar of length equal to the polynomial space bound with "hooked" arms freely linked to the bar at unit intervals. The environment is a complex arrangement of channels corresponding to the state transitions of the Turing machine.

There are several lower bound results for planar linkages [HJW2] [HJW3], [JP] where a planar linkage is a collection of hinged rods that is restricted to lie in the plane. A linkage that is a sequence of segments $A_1, A_2, \cdots, A_n$ such that the only hinge joints are joints that hinge consecutive segments together is called an *arm*. The non-hinged endpoint of the first segment is fixed at a point in the plane (although it is allowed to rotate about that point). The non-hinged endpoint of the last segment is called the *tip* of the arm.

In [HJW3] they show that even if the environment is very simple deciding whether the tip can reach a point from some initial configuration of the arm is NP-hard. Towards this end it is shown that deciding whether a arm (in the absence of any obstacles) can be folded (i.e. the angle at every hinge is 0 or π) so that its folded length is at most k is NP-complete. This is accomplished by reducing the NP-complete problem Partition (see [GJ]) to the folding arm problem. Next a construction is used to reduce the folding problem to the problem of deciding whether the tip of the arm can reach a given point in the plane. The construction uses a narrow tunnel with a wall in the middle such that the arm can reach around the wall in the tunnel if and only if the arm can be folded to a length no more than k (the width of the tunnel). This result is improved in [JP] where it is shown that the problem is actually PSPACE-hard.

In [HJW2] they study reachability problems for general planar linkages. Here the linkage is allowed to have certain endpoints fixed to the plane. In particular it is shown that deciding if an endpoint of a given segment of the linkage can reach a specified point in the plane when the linkage starts in a certain configuration is PSPACE-hard. This is accomplished by constructing a linkage to simulate a linear bounded automaton (LBA) computation. The size of the linkage is linear in the size of the description of the LBA and the size of the input. Since the acceptance problem for LBA's is PSPACE-complete (see [HU]) the reachability problem for general 2-dimensional linkages is PSPACE-hard. Notice that this result does not depend on a complex environment and in fact there are no obstacles in the construction at all. In [JP] they show that fixed endpoints can be simulated by restricting the linkage to lie in a square and adding additional links.

Hopcroft, Schwartz and Sharir [HSS] study the problem of coordinated motion of many 2-dimensional objects in a confined area. They show that deciding whether

rectangles within a rectangular boundary can be moved between two configurations is PSPACE-hard. The result is obtained by a sequence of reductions from a known PSPACE-complete problem. In particular they show that a certain rewriting system for formal strings can simulate LBA computations and then reduce that problem to a symbol transposition problem. In a general rewriting system characters can appear and disappear in the string being manipulated. In a symbol transposition problem the collection of characters in the string remains the same (they just move around within the string). Therefore a symbol transposition problem (where certain characters can be thought of as representing physical objects) more closely corresponds to moving a physical system where objects do not disappear and reappear. The symbol transposition problem is then reduced to the problem of deciding if a motion of "nearly rectangular" objects (rectangles with certain slots and tabs) exists. Lastly this problem is reduced to the problem of deciding if there is a motion between configurations of rectangles within a rectangular boundary. Their construction for this proof works whether or not rotation of the rectangles is allowed. Hopcroft and Wilfong [HW] have shown that if rotations are not allowed then the motion planning problem for rectangles in a rectangular boundary is in PSPACE and hence is PSPACE-complete. Other lower bound results for multiple objects can be found in [SY1] and [SY2].

In [N] Natarajan studies various motion planning problems for a robot with damping and position uncertainty. He provides a construction showing that in 3-space the problem of planning the motion of a point with damping and position uncertainty is PSPACE-hard.

Several papers have provided lower bounds for motion planning for simple single objects. For example, O'Rourke [O] has shown a lower bound for computing the motion of a ladder (a line segment) in a 2-dimensional polygonal environment. He uses a clever construction to show that moving a ladder between two positions could require as many as $O(n^2)$ simple motions where n is the complexity of the environment. In [KO] a complicated construction shows that moving a ladder in 3-dimensional space could require as many as $O(n^4)$ simple motions.

O'Dunlaing [OD2] shows that planning the motion of a single disc in a polygonal environment with $O(n)$ complexity requires $\Omega(n\log n)$ time. This is a tight lower bound as O'Dunlaing and Yap [ODY] show that this problem can be solved with $O(n\log n)$ preprocessing in $O(n)$ query time. The lower bound result is achieved by reducing the problem of sorting to the motion planning problem. For a given sequence $x_1, x_2, \cdots x_n$, to be sorted, an environment is constructed that is a hallway bounded below by the x-axis and above by the line $y = 3n + 6$. In the hallway there are vertical barriers at each x_i such that the height of the barrier at x_i encodes the subscript i. The object is to move a disc from left to right. This requires moving the disc around every barrier. The order in which the barriers appear from left to right is recorded by recording the corresponding subscript the first time it is detected that the disc is passing around a given barrier. Clearly this provides a sorting of the x_i's.

3. Optimal Motions

Most of the results in optimal motion planning concern finding shortest paths for a point object in polygonal (or polyhedral) space. The reader is advised to see [A] for a good overview of this area.

There are $O(n^2)$ time algorithms for finding the shortest path in polygonal space where n is the number of vertices in the space (see [AAGHI], [We] and [RSt]). Clarkson, Kapoor and Vaidya [CKV] have a method of finding a shortest rectilinear path in polygonal space in $O(n\log n)$ time. Chew [Ch] develops an $O(n^2\log n)$ time algorithm for finding the shortest path of a disc in a polygonal environment. The problem in 3-dimensional space seems to be much more difficult. A recent result of Canny and Reif [CR] shows that the problem of finding a shortest path in 3-dimensional polyhedral space is NP-hard.

Other optimization algorithms are described in the section on constrained motion. These algorithms deal with the problem of minimizing the time to get to a desired location when the moving object has an upper bound its allowable speed.

4. Planning in an Unknown Environment

A different approach to motion planning is the case where the environment is unchanging but unknown. This is a useful model for guiding an autonomous vehicle in a new environment. For this approach a global description of the obstacles to avoid is unavailable and so the moving object has to deal with the obstacles dynamically as they are encountered. The vehicle is assumed to have some sort of sensor capability in order to detect when it is touching (or near) an obstacle and some sort of position and orientation reckoning system. Various algorithms for vehicles and arms moving in unknown environments can be found in [LS1],[LS2] and [L].

5. Constrained Motion

Recently there has been work on problems of motion planning where the object to be moved has various constraints on its ability to move other than just physical obstacles to avoid. The velocity constrained problem and the acceleration constrained problems are concerned with moving a constrained object to a desired location while avoiding moving obstacles. In the curvature constrained problem the obstacles are static but the object has a bound on how tightly it can turn a corner.

5.1. Velocity constrainted motion

Reif and Sharir [RS] consider the problem of moving an object in an environment where there are obstacles moving about according to known simple trajectories. They consider various versions of such problems in 1, 2 and 3 dimensions.

They use an interesting construction that they call a "time-machine" to show that the problem of planning the motion of an object with bounded velocity modulus in 3-dimensional space amongst rotating obstacles is PSPACE-hard even if the object is a disc. In previous lower bound results for motion planning amongst static obstacles the construction depended on the object having $\Omega(n)$ degrees of freedom whereas in this case the object has only $O(1)$ degrees of freedom. This indicates that to plan the motion of even very simple systems in the presence of moving obstacles is likely to be

difficult as opposed to the case of static obstacles where there is a polynomial time procedure when there are only $O(1)$ degrees of freedom for the object [SS2].

In light of the PSPACE-hardness result they then consider what they call "asteroid avoidance problems". These problems consider the case where the object to be moved is a convex polyhedron whose only allowed motion is translation and it has a bound on its velocity modulus. Also each obstacle is a convex polyhedron that translates at a constant velocity and obstacles never collide with one another. They construct time-configuration space which is configuration space as in [LW] with an extra dimension for time. It is then shown that it is sufficient to consider motions that are a finite alternation of motions along the obstacles in time-configuration space and motions with maximum velocity modulus. They present an algorithm for the case where the object is restricted to move along a straight line and the obstacles that cross the line are 2-dimensional. The algorithm has a worst-case running time of $O(n\log n)$ where n is the number of obstacle edges. The technique used is basically a line-sweep of time-configuration space recording the accessible positions at each moment in time where a corner of a configuration obstacle lies.

The next case considered is where the object is allowed to translate anywhere in the plane (i.e. it is not confined to move along a line). An $O(n^{2(k+2)}k)$ time algorithm is developed where again n is the number of obstacle corners and k is the number of obstacles. The 3-dimensional problem is solved by formulating the problem as a problem in the theory of real closed fields and solving it using the method of Collins [Co] and so produce a decision procedure that runs in time $2^{n^{O(1)}}$. space.

Various problems dealing with moving obstacles when the object to be moved has no bound on velocity modulus or where all the obstacles move more slowly than the allowable motion of the object are also studied in [RS].

The solutions presented in [RS] for the asteroid avoidance problem can be refined to give motions to a designated position at the earliest possible time. Solutions for the problem of finding a motion to minimize the time of arrival, to minimize the distance traveled by the object and variations of these can be found in [W1] in the 1-dimensional case. Aronov, Fortune and Wilfong [AFW] consider the problem of computing the minimum velocity modulus bound that would allow a feasible path to exist in the 1-dimensional case. Using $O(n\log n)$ preprocessing time and $O(m + \log n)$ query time (where m is the number of segments that make up the resulting path) the algorithm in [AFW] produces the minimum allowable velocity bound as well as a feasible path given that bound.

5.2. Acceleration bounded motion

O'Dunlaing [OD1] considers the problem of planning the motion of a point along a line where "barriers" on both sides of the moving point move in a known manner. The problem is to get the point from a starting position at a given time moving with a certain velocity to a final position at a given time moving with a stated velocity while avoiding the barriers. The constraint on the motion of the point is that it can not exceed a given acceleration modulus.

The motion of the barriers is given by two functions of time $g(t)$ and $f(t)$. The functions f and g are restricted to parabolic splines. Let n be the total number of times

that f and g change acceleration. O'Dunlaing presents two algorithms for the problem, one with worst-case running time of $O(n^2)$ and another with worst-case running time of $O(n\log n)$.

The method used for the first algorithm defines time-configuration space as in [RS] except in this case the only obstacles are the paths the two barriers follow. These two paths define the upper and lower bounds of the position of the moveable point at any instance of time. Suppose f defines the lower bound and g defines the upper bound. Let M be the bound on the acceleration modulus of the point. Notice that if the point travels with constant acceleration M (or $-M$) the path in time-configuration space is parabolic. The first step in the algorithm is to refine f and g so that the resulting f and $\hat{g}$ have the property that for any time t the point can reach $\hat{f}(t)$ at time t and be moving with velocity given by the derivative of $\hat{f}$ at t and similarly for $\hat{g}$. One operation needed to refine f and g is called *filling* and this removes the pairs (x,t) from consideration such that if the point was at position x at time t then any allowable trajectory from there would result in the point crashing through a barrier. Another operation, *shading* removes pairs (x,t) such that there is no way for the point to get to x at time t because of its acceleration constraint. The functions $\hat{f}$ and $\hat{g}$ are found by shading f and g and then filling the resulting functions and then repeating the process. To see if a particular (x,t) in time-configuration space is reachable by the point is equivalent to checking if $\hat{f}(t) \leq x \leq \hat{g}(t)$. O'Dunlaing shows how $\hat{f}$ and $\hat{g}$ can be constructed in $O(n^2)$ time.

It is also necessary to see if the desired velocity is attainable at the destination position and time. It is shown that the maximum attainable velocity at (x,t) is given by the supremum of the velocities at (x,t) on a maximum acceleration trajectory that does not cross $\hat{f}$. The minimum attainable velocity is similarly defined. These velocity bounds can be calculated in $O(n)$ time. Therefore the entire algorithm is $O(n^2)$.

The second method iteratively finds the accessible position and velocities at the points where f and g change acceleration. It is a more complicated method that considers the problem in terms of *phase space* which is the space of triples of time, position and velocity. It results in an $O(n\log n)$ time algorithm.

5.3. Paths with bounded curvature

Fortune and Wilfong [FW] have studied a motion planning problem where there is a constraint placed on the curvature of the allowable paths. This problem can be thought of as planning the motion of a point that has an associated direction so that the point can only move forward as defined by its direction and its direction can change subject to a rate constraint. A *placement* is a pair (position, direction). Thus the goal is to find a path p from an initial placement to a final placement within a given polygonal environment so that the curvature of the path at any point is no more than 1. (See Guggenheimer [G] for a formal definition of curvature.) Such a path is called an *feasible path.*

The motion planning problems without kinematic constraints discussed earlier relied on the topological path connectedness of points in configuration space while the other motion planning problems with constraints studied the "forward-connectedness" of points in time-configuration space. In this case the notion of some sort of configuration space is not so clear. Certainly one could choose the set of placements to be the configuration space but then the definition of path connectedness is very complicated

because of the curvature constraint.

The general method used to solve the curvature constrained problem is outlined as follows. First it is shown that if any feasible path exists then there is a feasible path that consists of a finite number of pieces each of which is a straight line segment or an arc of radius 1. Define a *jump* to be a path that consists of a unit radius arc followed by a line segment tangent to the arc followed by another unit radius arc. Notice that any one or two of the pieces of a jump might have zero length. It is then shown that if there is a feasible path then there must be a feasible path that consists of a jump from the initial placement to a wall or corner of the environment followed by a sequence of jumps such that the endpoint of each jump is a wall or corner, followed by a jump from a wall or corner to the final placement. Call such a path of jumps a *canonical path.* Therefore it is sufficient to compute for each wall and corner the set of placements at that wall or corner reachable from the initial placement by a canonical path and then test if there is a jump from any reachable placement at a corner or wall to the final placement.

The algorithm given in [FW] relies on a decision procedure for the reals ([Co]) and has a worst-case running time that is doubly-exponential in the size of the problem.

It would be nice to extend these results from planning the motion of a point to planning the motion of a line segment (a stick) or a polygonal object. To capture some notion of the curvature constraint in the case of a stick consider the following model. Designate one endpoint of the stick the head and the other the tail. Let a be an interior point on the stick and define the allowed motion of the stick to be forward in the direction of the head of the stick and the stick can rotate about the point a. Suppose the stick is lying almost parallel to a wall and we wish to move the stick so that it becomes perpendicular to the wall with the tail touching the wall. The wall restricts the stick from just rotating to the desired position and so the tail of the stick slides along the wall as we try to rotate the stick towards a perpendicular position. A problem arises because the head of the stick traces out a fairly complex curve involving logarithms as the tail of the stick moves along a wall. Unlike the point problem where the limiting curves consisted of arcs and straight lines it is now necessary to consider these more complex curves. It is not known how to solve equations of arbitrarily nested logarithms. This indicated that the problem of a stick would be very difficult.

Notice also that the curvature constraint problem for a point is a special case of the more general problem of planning the motion of a point between two positions in the plane where a velocity is given for the point at each position and there is a bound on the acceleration of the point. The problem considered in [FW] is the case where the only allowed acceleration is normal to the path. Therefore the point moves at a constant speed but it can change directions at a limited rate.

6. Changeable Environment

Another useful model to consider is where the environment not only contains static obstacles but there are also obstacles that can be moved by the robot whose motion is to be planned. This captures the case where a robot is picking up various things in its workspace and moving them about. Since the robot is able to pick up and move obstacles an important question to ask is which obstacles should be picked up and moved (and to where) in order for the robot to perform a given task. There are only a

few known results for such motion planning problems. An initial study in this area [W2] develops an algorithm that runs in time $O(n^3)$ in the case where there is just one movable obstacle in a polygonal environment. If there are many movable obstacles the problem becomes much more difficult. In fact the problem becomes NP-hard even if there are only three different types of obstacles allowed (two sizes of rectangles and an 'L'-shaped piece) and the robot is a square [W2]. Following a suggestion by Aronov it can be shown that if many different types of obstacles are allowed then the construction in [HSS] can be used (along with a very small square moving object) to show that the problem is PSPACE-hard.

7. Conclusions

While much progress has been made in the area of algorithmic motion planning there still remain many areas of open problems. For example the study of motion planning where the moving object has constraints on the permissible types of motion still has many open problems. Those results referred to in this paper are but a first step in an important practical area of concern.

References

[AHU] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[A] V. Akman, *Unobstructed Shortest Paths in Polyhedral Environments*, Springer-Verlag Lecture Notes in Computer Science, No. 251, 1987.

[AFW] B. Aronov, S. Fortune and G. Wilfong, "Minimum Speed Motions", *SIAM Conf. on Applied Geometry*, 1987.

[AAGHI] Takao Asano, Tetsuo Asano, L. Guibas, J. Hershberger and H. Imai, "Visibility-Polygon Search and Euclidean Shortest Paths", *Proc. of the 26th Symposium on Foundations of Computer Science* (1985), pp. 155-164.

[Ca] J. Canny, MIT PhD Thesis (1987).

[CR] J. Canny and J. Reif, private communication.

[Ch] L.P. Chew, "Planning the Shortest Path for a Disc in $O(n^2 \log n)$ time", *Proc. 1st ACM Symp. on Computational Geometry* (1985), pp. 214-220.

[CKV] K. Clarkson, S. Kapoor and P. Vaidya, "Rectilinear Paths through Polygonal Obstacles in $O(n \log n)$ Time", *Proc. 3rd ACM Symp. on Computational Geometry* (1987).

[Co] G.E. Collins, "Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition", *Proc. 2nd GI Conf. on Automata Theory and Formal Languages*. Springer-Verlag Lecture Notes in Computer Science, No. 33 (1975), pp. 134-183.

[FWY] S. Fortune, G. Wilfong and C. Yap, "Coordinated Motion of Two Robot Arms", *Proc. 1986 IEEE Int. Symp. on Robotics and Automation*, pp. 1216-1223.

[FW] S. Fortune and G. Wilfong, "Planning Constrained Motion", unpublished manuscript.

[GJ] M.R. Garey and D.S. Johnson, *Computers and Intractability, A Guide to the*

Theory of NP-Completeness, W.H. Freeman and Co., San Francisco, CA., 1979.

[G] H.W. Guggenheimer, *Differential Geometry*, McGraw-Hill Book Co., Inc., New York NY, 1963.

[HJW1] J.E. Hopcroft, D.A. Joseph and S.H. Whitesides, "Determining Points of a Circular Region Reachable by Joints of a Robot Arm", Computer Science Tech. Report 82-516, Cornell University, 1982.

[HJW2] J.E. Hopcroft, D.A. Joseph and S.H. Whitesides, "Movement Problems for 2-Dimensional Linkages", *SIAM J. Computing*, 13 (1984), pp. 610-629.

[HJW3] J.E. Hopcroft, D.A. Joseph and S.H. Whitesides, "On the Movement of Robot Arms in 2-Dimensional Bonded Regions", *SIAM J. Computing*, 14 (1985), pp. 315-333.

[HSS] J.E. Hopcroft, J.T. Schwartz and M. Sharir, "On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE Hardness of the 'Warehouseman's Problem' ", *Int. J. on Robotics Research* 3(4), pp. 76-88, 1984.

[HU] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Pub. Co., Reading, Mass., 1979.

[HW] J.E. Hopcroft and G.T. Wilfong, "Reducing Multiple Object Motion Planning to Graph Searching", *SIAM J. on Comput.*, Vol. 15, No. 3, pp. 768-785, Aug. 1986.

[JP] D.A. Joseph and W.H. Plantinga, "On the Complexity of Reachability and Motion Planning Questions", *Proc. 1st ACM Symp. on Computational Geometry* (1985), pp. 62-66.

[KO] Y. Ke and J. O'Rourke, "Moving a Ladder in Three Dimensions: Upper and Lower Bounds", *Proc. 3rd ACM Symp. on Computational Geometry* (1987).

[LS] D. Leven and M. Sharir, "An Efficient and Simple Motion Planning Algorithm for a Ladder Moving in Two-Dimensional Space Amidst Polygonal Barriers", *Proc. 1st ACM Symp. on Computational Geometry* (1985), pp. 221-227.

[LW] T. Lozano-Perez and M. Wesley, "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles", *Comm. ACM*, Vol. 22, No. 10 (October 1979), pp. 560-570.

[L] V.J. Lumelsky, "Dynamic Path Planning for a Planar Articulated Robot Arm Moving Amidst Unknown Obstacles", *Automatica, Journal of Intern. Fed. of Automatic Control*, June 1987.

[LS1] V.J. Lumelsky and A. Stepanov, "Dynamic Path Planning for a Mobile Automaton with Limited Information on the Environment", *IEEE Trans. on Automatic Control*, Vol. AC-31, No. 11, Nov. 1986.

[LS2] V.J. Lumelsky and A. Stepanov, "Path Planning Strategies for a Point Mobile Automaton Moving Amidst Unknown Obstacles of Arbitrary Shape", *Algorithmica*, 1987.

[N] B.K. Natarajan, "On Moving and Orienting Objects", Ph.D. Thesis, Dept. of Computer Science Tech. Rept. 86-775, Cornell Univ., Aug. 1986.

[OD1] C. O'Dunlaing, "Motion Planning with Inertial Constraints", NYU Courant

Institute of Math. Sciences Robotics Technical Report No. 73, July 1986.

[OD2] C. O'Dunlaing, "A Tight Lower Bound for the Complexity of Path-Planning for a Disc", NYU Courant Institute of Math. Sciences Robotics Technical Report No. 75, July 1986.

[ODY] C. O'Dunlaing and C.K. Yap, "A 'Retraction' Method for Planning the Motion of a Disc", *J. Algorithms* 6 pp.104-111.

[O] J. O'Rourke, "A Lower Bound on Moving a Ladder", Tech. Rept. 85/20, Dept. of EECS, The Johns Hopkins Univ., Nov. 1984.

[OY] J. O'Rourke and C.K. Yap, "The Voronoi Method for Motion Planning: I. The Case of a Disc", *J. of Algorithms*, Vol. 6, pp. 104-111, 1985.

[R] J. Reif, "Complexity of the Mover's Problem and Generalizations", *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pp.421-427, 1979.

[RS] J. Reif and M. Sharir, "Motion Planning in the Presence of Moving Obstacles", *Proc. of the 26th Symposium on Foundations of Computer Science* (1985), pp. 144-154.

[RSt] J. Reif and J.A. Storer, "Shortest Paths in Euclidean Space with Polyhedral Obstacles", Tech. Rept. CS-85-121, Computer Science Dept., Brandeis Univ., April 1985.

[SS1] J.T. Schwartz and M. Sharir, "On the 'Piano Movers' Problem: I. The Case of a Rigid Polygonal Body Moving Amidst Polygonal Barriers", *Comm. Pure Appl. Math.*, Vol. 36, pp. 345-398 (1983).

[SS2] J.T. Schwartz and M. Sharir, "On the 'Piano Movers' Problem: II. General Techniques for Computing Topological Properties of Real Algebraic Manifolds", *Advances in Applied Mathematics* No. 4, pp. 298-351 (1983).

[SS3] J.T. Schwartz and M. Sharir, "On the 'Piano Movers' Problem: III. Coordinating the Motion of Several Independent Bodies: The Special Case of Circular Bodies Amidst Polygonal Barriers", *Int. J. of Robotics Research*, Vol. 2, No.3, Fall 1983, pp. 46-75.

[SS4] J.T. Schwartz and M. Sharir, "Motion Planning and Related Geometric Algorithms in Robotics", NYU Courant Institute of Math. Sciences Robotics Technical Report No. 79, Aug. 1986.

[ShS] M. Sharir and A. Schorr, "On the Shortest Paths in Polyhedral Spaces", *Proc. of the Sixteenth Annual Symposium on Theory of Computing* (1984), pp. 144-154.

[SiS] S. Sifrony and M. Sharir, "An Efficient Motion Planning Algorithm for a Rod Moving in Two-Dimensional Polygonal Space", Technical Rept. 85-40, The Eskenasy Institute of Computer Science, Tel Aviv Univ., Aug. 1985.

[SY1] P.Spirakis and C.K. Yap, "On the Combinatorial Complexity of Motion Coordination", NYU Courant Institute of Math. Sciences Robotics Technical Report No. 12, April 1983.

[SY2] P.Spirakis and C.K. Yap, "Strong NP-hardness of Moving Many Discs", *Inf. Proc. Letters*, Vol. 19, No. 1, 1984, pp. 55-59.

[We] E. Welzl, "Constructing the Visibility Graph for n Line Segments in $O(n^2)$

Time", *Inf. Proc. Letters* 20 (1985), pp. 167-172.

[W1] G.T. Wilfong, "Planning Motions in Time", unpublished manuscript, 1985.

[W2] G.T. Wilfong, "Complexity of Pick and Place", unpublished manuscript, 1986.

[Y1] C.K. Yap, "Coordinating the Motion of Several Discs", NYU Courant Institute of Math. Sciences Robotics Technical Report No. 16, Feb. 1984.

[Y2] C.K. Yap, "Algorithmic Motion Planning", *Advances in Robotics, Volume 1: algorithmic and geometric aspects*, ed. J.T. Schwartz and C.K. Yap, Lawrence Erlbaum Assoc., Hillsdale NJ, 1986.