SMALLEST PATHS IN POLYGONS

by

Kenneth M. McDonald

B.Sc. (Hons.), University of Saskatchewan, 1986

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY

May, 1989

APPROVAL

Name:	Kenneth M. McDonald
Degree:	Master of Science
Title of Thesis:	Smallest Paths in Polygons

Examining Committee:

Chairman: Dr. A. H. Dixon

Dr. J. G. Peters, Senior Supervisor

Dr. L. J. Hafer

Dr. A. L. Liestman

Dr. K. K. Gupta, External Examiner

Date Approved: July 6, 1989

Title of Thesis/Project/Extended Essay

Smallest Paths in Polygons.

Author:

(signature)

Kenneth Murray McDonald

(name)

September 24, 1989

(date)

ABSTRACT

A rectilinear path is a path composed only of horizontal and vertical line segments. Such paths may be constrained by requiring that they lie only within certain areas. One way of doing this is to require that a rectilinear path be entirely contained within a given simple polygon. We show that between any two non-vertex points in any simple polygon, there is a rectilinear path entirely contained within the polygon, which simultaneously possesses the properties that it is of no greater length and has no more bends than any other rectilinear path between the two given points and lying within the given polygon. Such a path is termed a smallest path. We further give an O(n log n) sequential algorithm to calculate the length and number of bends of a smallest path between two given points in any simple polygon, where n is the number of vertices of the polygon, and also develop parallel algorithms to do the same task, which run in O($\log^2$ n) time using n/log n processors, or in O(log n log log n) time using n processors. Finally, we consider the case where rather than being restricted to the inside of a simple polygon, we restrict our rectilinear paths to lie outside a set of rectlinear objects, and given an O(n^3) seqential algorithm to determine if a smallest path exists in such an environment, where n is the total number of corners in the set of rectilinear obstacles.

ACKNOWLEDGMENTS

My thanks to my senior supervisor, Dr. Joseph Peters, for help and advice concerning not only my thesis, but many other areas of my academic career as well.

Thanks to my parents, for encouraging me and supporting me in my academic endeavours.

Thanks, too, to the other members of my thesis committee, who provided a careful and conscientious reading of my thesis as well as other help, and to Dr. Binay Battacharya, who aided me in the (to me) new field of computational geometry.

Finally, a special note of appreciation to the Office of the Dean of Graduate Studies, who were most helpful early in my Master's program.

Table of Contents

Table of Contents (continued)

1. Introduction

Rectilinear paths are paths between points consisting only of vertical and horizontal line segments. As with most types of paths, the most common metric for measuring rectilinear paths is the length of the path, and in many environments, it may be desirable to find shortest rectilinear paths between points. Another obvious metric for use with rectilinear paths may be obtained by considering the number of bends (or distinct segments) in a rectilinear path; this metric could perhaps be termed the "straightness" of a rectilinear path, and in various situations it may be desirable to find rectilinear paths containing a minimum number of bends.

The problem of finding shortest rectilinear paths in the plane has been extensively studied. Most variants of the problem involve finding a shortest rectilinear path between two points in the plane that avoids a set of obstacles which may be rectangles [1], rectilinear polygons [2], or simple polygons [3; 4]. Variants in which the path must use line segments from a given set of lines have also been studied [5]. There are also many related papers dealing with robot motion planning. See [6] for a survey.

Problems involving minimum-bend paths have not received as much attention as shortest paths. Lipski [7; 8] has studied the variant in which the path must use line segments from a given set of lines. Asano, Sato, and Ohtsuki [9] describe an algorithm based on Lipski's for finding a minimum-bend rectilinear path between two points which avoids rectilinear obstacles. Suri [10], Ke [11] and others have studied a variety of problems involving minimum-bend non-rectilinear paths.

By combining the problems of trying to find a minimum-length rectilinear path and a minimum-bend rectilinear path, and attempting to find a rectilinear path which is simultaneously minimum-length and minimum-bend, an interesting new problem can be synthesized, which we term the *smallest path problem.* We call a path which simultaneously minimizes length and number of bends a *smallest path.*

There are various areas of application for smallest rectilinear paths. In the VLSI setting, it is sometimes desirable to find paths such that all horizontal wires lie in the first layer and all vertical wires lie in the second layer [12]. A smallest path minimizes both distance and the number of vias. Smallest paths might also be useful when planning traffic routes in cities with grid-like road systems because minimum-distance routes which require the fewest turns are in general the most efficient.

The main result of this thesis is an algorithm for finding the length and number of segments of smallest paths between pairs of points in simple polygons. It is shown that, except for paths involving some easily identifiable vertex points, smallest paths between any two points in a simple polygon always exist. Examples are given to show that the number of segments of a smallest path in a simple polygon is not necessarily bounded by any function of the number of vertices n of the polygon, and an O(n log n) time sequential algorithm is given to find the dimensions (length and number of segments) of a smallest path between any two points in a simple polygon. In the process, an interesting technique for finding the dimensions of smallest paths through a restricted class of simple polygons called funnels is developed. The sequential algorithm is then generalized to a parallel algorithm for use on a CREW PRAM, which can find dimensions of a smallest path between two points in a simple polygon in $O(\log^2 n)$ time, using O(n/log n) processors, or in O(log n log log n) time, using O(n) processors. Finally, a brief consideration of finding smallest paths in an environment containing rectangular obstacles is given. It is shown that in such an environment, a smallest path between two points does not necessarily exist. Given that there are n rectangular obstacles in the environment, an $O(n^3)$ time algorithm is given to determine, for two specified points, if a rectilinear path of length less than or equal to a specified rational number L and number of segments less than or equal to a specified integer exist. This algorithm also permits determination, in the same amount of time, of whether or not a smallest path between the points exists.

The main original contributions of the thesis may be summarized as follows: 1) Smallest-path problems are defined, one class of smallest-path problems (involving smallest paths in simple polygons) is thoroughly investigated, and another class of smallest path problems (concerning smallest paths around rectilinear barriers) is briefly discussed (see chapter 7): 2) A technique for simplifying a polygon into a structure called a *pseudogon* is demonstrated: 3) A class of polygons called *funnels* is defined, and it is shown how to find the number of segments and length of a smallest path through such polygons: 4) Efficient sequential and parallel algorithms are given to find the length and number of segments in smallest paths in simple polygons.

These points all seem to be quite new to the field. This researcher has encountered no papers which even mention the idea of simultaneously minimizing both number of segments and length of a rectilinear path. For example, Larson and Li in [4] discuss minimum-length rectilinear paths in the presence of barriers, and must deal with the possibility that some paths will have a large number of segments due to the presence of the barriers, but make no attempt to minimize the number of path segments. Addressing the second point, while many other algorithms use simplification of polygons in their execution (trapezoidal decomposition and triangulation are both commonly used techniques which may be viewed as polygonal simplification), these techniques are not common to problems dealing with rectilinear paths; as well, the construction of what is termed the *pseudogon*, and the use of the properties inherent in it, are certainly new to rectilinear path problems. *Funnels* also seem to be an entirely new concept in the field of rectilinear path problems, which is not surprising, as the work done in this thesis on finding minimum-segment rectilinear paths (or more accurately, the number of segments in minimum-segment rectilinear paths) in the presence of non-rectilinear barriers (i.e. the non-rectilinear sides of a simple polygon) seems to be entirely new. All previous known work in finding minimum-segment rectilinear paths [5; 7; 8; 9; 12] has considered only rectilinear barriers. Finally, since the smallest-path problem is a new problem, and many of the considerations it raises are also new, the algorithms to solve it are original.

The thesis is organized as follows. Chapter 2 gives basic definitions, and proves the existence of smallest paths in simple polygons. Chapter 3 gives a topological analysis of the smallest path problem. Chapter 4 uses the results of chapter 3 to develop an efficient sequential algorithm for determining the dimensions of a smallest path between two points in a simple polygon, but defers discussion of a particular subclass of simple polygons called funnels until chapter 6. Chapter 5 shows how the algorithms of chapter 4 may be adapted for use on a parallel processing machine, and also defers discussion of funnels until chapter 6. Chapter 6 addresses the problems presented in finding the dimensions of smallest paths in funnels, and gives efficient sequential and parallel algorithms for doing so. Chapter 7 discusses smallest paths in environments containing rectangular obstacles. Finally, Chapter 8 provides a summary of the thesis. Three appendices respectively discuss the use of traversal numbers (used in chapters 4 and 5), give a graphical illustration of the execution of the SIMPLIFY algorithm described in Chapters 4 and 5, and illustrate the parallel prefix operation (used throughout the parallel portions of the thesis).

2. The Smallest-Path Problem

Definitions: [simple polygon, boundary, interior, rectilinear path, in, within]: A *simple polygon* Q is a finite region on the plane whose boundary, denoted *bdy(Q)*, is given by a cycle of line segments, connected end-to-end and otherwise non-intersecting. The finite region the boundary encloses, not including the boundary itself, is called the interior of Q, or *int(Q)*, and Q refers to int(Q) $\cup$ bdy(Q). A *rectilinear path* from a point x to a point y is a sequence of horizontal and vertical line segments, connected end-to-end, such that the free end of the first segment in the sequence is at x, and the free end of the last segment in the sequence is at y. A point is *in* Q (or alternatively, *within* Q) if it is a member of the set of points bdy(Q) $\cup$ int(Q), and a path is in Q or within Q if every point on the path is within Q.

Definitions: [line segment, maximal line segment, chord]: For the purposes of this thesis, we define a *line segment* to be a rectilinear line segment within Q, unless otherwise noted. A *maximal line segment* L is a rectilinear line segment in Q such that there is no rectilinear line segment in Q longer than L and containing L. A *chord* C of a polygon Q is a line segment such that the endpoints of C intersect bdy(Q), and no other points on C intersect bdy(Q). This definition corresponds to what would normally be called a rectilinear chord, but since all chords referred to in this thesis will be rectilinear, we simply call them chords.

Definitions: [length, straightness]: If P is a rectilinear path between two points in Q, then the *length* of P, denoted *len(P)*, is the sum of the lengths of the segments which make up P, and the *straightness* of P, *seg(P)*, is the number of distinct segments forming P (1 + the number of bends in P). We say that a path P is *shorter* than a path P′ if len(P) < len(P′), and P is *straighter* than P′ if seg(P) < seg(P′). P is *shortest* out of a set of paths M if P is in M and no path in M is shorter than P, and P is *straightest* in a set of paths M if P is in M and no path in M is straighter than P.

Definitions: [shortest, straightest, smallest paths between points]: Let S and T be two points in Q. A *shortest* path P from S to T is a shortest path out of the set of all paths in Q from S to T. A *straightest* path P′ from S to T is a path which is straightest out of the set of all paths in Q from S to T. A *smallest* path from S to T is a path from S to T which is both shortest and straightest.

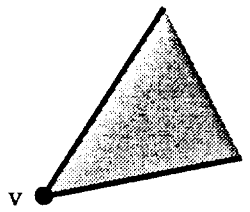
Definitions: [shortest, straightest, smallest paths from a point to a line segment]: Let S be a point in Q and let L be a rectilinear line segment in Q. A *shortest* path from S to L is a rectilinear path P1 in Q, from S to some point w1 on L, s.t. if w2 is any point on L and P2 is a shortest path from S to w2, then len(P2) ≥ len(P1). A *straightest* path from S to L is a rectilinear path P1 in Q, from S to some point w1 on L, s.t. if w2 is any point on L and P2 is a straightest path from S to w2, then seg(P2) ≥ seg(P1). A *smallest* path from S to L is a path from S to L which is both shortest and straightest.

Definitions: [shortest, straightest, smallest paths between two line segments in Q]: Let L1 and L2 be two line segments in Q. A path P from a point w1 on L1 to a point w2 on L2 is a *shortest* path from L1 to L2 if for any path P′ from a point w1′ on L1 to a point w2′ on L2, len(P′) ≥ len(P). P is a *straightest* path from L1 to L2 if for any path P′ from a point w1′ on L1 to a point w2′ on L2, seg(P′) ≥ seg(P). P is a *smallest* path from L1 to L2 if P is both a shortest and a straightest path from L1 to L2.

Definition: [dimensions of a path]: We define the *dimensions* of a rectilinear path to be the 2-tuple composed of the length of the path and the number of segments in the path.

Problem: [smallest-path problem]: Given a simple polygon Q, and points S and T in Q, the *smallest-path problem* is to find the dimensions of a smallest path from S to T in Q, if one exists.

figure 1



If v is a vertex of Q and the grey region is the interior of Q, then there is no rectilinear path of a finite number of segments in Q from v to any other point in Q.

Note: If S or T is on a vertex v of Q such that the two segments of bdy(Q) incident on v are non-rectilinear and in the same quadrant of the plane relative to v (see figure 1), then there can be no smallest path of a finite number of segments in Q from S to T, and so in the following, we assume that neither S nor T is located on such a vertex. As well, we will assume for the sake of convenience in later parts of the thesis that S≠T.

Definitions: [notational devices]: In order to avoid continuous redefinition of frequently used geometric constructs, we will let certain identifiers *always* represent a specific type of geometric construct. In particular, the following apply unless specifically noted otherwise:

Q will denote an arbitrary simple polygon in which we wish to find a smallest path.

P will denote an arbitrary rectilinear path in Q.

$\overline{xy}$ will denote the rectilinear line segment between points x and y.

C will denote an arbitrary rectilinear chord of Q.

L will denote an arbitrary rectilinear line segment in Q.

S, T will indicate the points in Q between which we wish to find a smallest path.

"Arbitrary" in the above should be understood to mean subject to any restrictions that may be given for a particular use of an identifier.

Nearest Point Lemma: Let P be a shortest path from S to a maximal rectilinear line segment L, and let x be the point on L at which P ends. The following statements are then true:

1) If y is a point on L and P_{Sy} is a shortest path from S to y, then the length of P_{Sy} is given by $\text{len}(P_{Sy}) = \text{len}(P) + \text{len}(\overline{xy})$.
2) There is a shortest path from S to L, which ends at x and which is also a straightest path from S to L, and is therefore a smallest path from S to L.

Proof: [by induction on the number of segments in a straightest path from S to L]

Note: In the following proof, be careful to distinguish between a shortest path from S to a line segment L, where the path happens to end at a point x on L, and a shortest path from S to a point x, where x happens to be a point on a line segment L. The two paths are not necessarily the same length, and the difference is important.

Basis: For a maximal line segment L such that there is a shortest path of one or two segments from S to L, the lemma is obviously true.

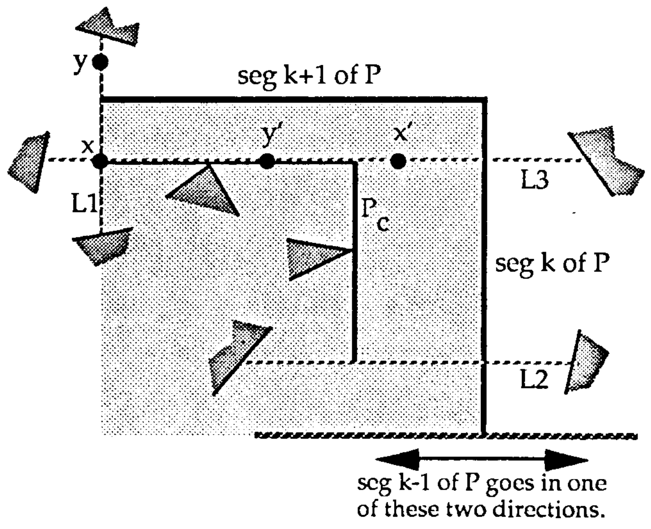
Inductive Assumption: For some integer k, k≥2, assume that the lemma holds for all maximal line segments L such that there is a straightest path P from S to L having k or fewer segments.

Inductive Step: Let L1 be a maximal line segment such that a straightest path P from S to L1 has k+1 segments. Because L1 is maximal, the k+1st segment of P (incident on L1) will be perpendicular to L1. We continue this proof with reference to figure 2, and all orientations and directions mentioned in the proof should be taken relative to this diagram.

If every maximal line segment L2 passing through a point of the kth segment of P and perpendicular to that segment intersected L1, then the light grey rectangle shown in the diagram would be entirely contained in Q, and a k-1-segment path could be constructed to a point on L1 by extending segment k-1 of P (if necessary) until it intersects L1, and taking the point of intersection as the terminus of a k-1 segment path from S to L1. We have defined L1 to be a maximal line segment s.t. the straightest path from S to L1 takes at least k+1 segments, so this is impossible—there must be at least one maximal line segment L2 through segment k of P and perpendicular to it, such that L1 and L2 do not intersect, as shown in the drawing.

Having obtained L2, we can construct a smallest path P_c (c stands for corner, because P_c forms a corner) between L2 and L1. Such a path will obviously

figure 2



Thick lines represent paths in Q, thin lines represent bdy(Q), dark grey areas indicate ext(Q), and dotted lines represent maximal line segments in Q.

exist—it is the two-segment path from a point on L2 to a point on L1 such that moving the horizontal segment of P_c any farther down (in this diagram) will cause some part of it to be outside of Q, and moving the vertical segment to the left (in this diagram) will cause some part of it to be outside of Q. Let x be the endpoint of P_c on L1. Given P_c we can form a maximal horizontal line segment L3 containing the horizontal segment of P_c and intersecting the kth segment of P. P_c will be in the grey rectangle defined by the kth and k+1st segments of P, and so L3 will intersect the kth segment of P.

Let $P_{Sx'}$ be a smallest path from S to L3, ending at some point x′, and let P_{Sx} be the path from S to x formed by adding $\overline{x'x}$ to the end of $P_{Sx'}$. By the inductive assumption, P_{Sx} is a shortest path from S to x and has length

$$\mathrm{len}(P_{Sx}) = \mathrm{len}(P_{Sx'}) + \mathrm{len}(\overline{x'x}) \qquad \textit{(1).}$$

Now let P_{Sy} be a shortest path from S to some point y on L1. Because of the way in which L3 was constructed, P_{Sy} will intersect L3 at some point y′ (the y′ shown in the diagram is purely arbitrary, and is shown simply to give the reader a concrete representation of y′). The length of P_{Sy} may be given as

$$\mathrm{len}(P_{Sy}) = \mathrm{len}(P_{Sy'}) + \mathrm{len}(P_{y'y}) \qquad \textit{(2),}$$

where $P_{Sy'}$ and $P_{y'y}$ are the subpaths of P_{Sy} from S to y′ and from y′ to y respectively. Since P_{Sy} is a shortest path from S to y, $P_{Sy'}$ will be a shortest path from S to y′, and so by the inductive assumption, we have

$$\mathrm{len}(P_{Sy'}) = \mathrm{len}(P_{Sx'}) + \mathrm{len}(\overline{x'y'}) \qquad \textit{(3).}$$

$P_{y'y}$ can be created as a one- or two-segment path (one segment from y′ to x, and if x≠y, another segment from x to y), so its length can be given as rectilinear distance between y′ and y, expressed as

$$\mathrm{len}(P_{y'y}) = \mathrm{len}(\overline{y'x}) + \mathrm{len}(\overline{xy}) \qquad \textit{(4).}$$

If $\mathrm{len}(\overline{y'x})$ were greater than $\mathrm{len}(\overline{x'x})$, then P_{Sy} would not be a shortest path from S to y (a shorter path to y could be constructed by going through x′), and so we can assume that $\text{len}(\overline{y'x}) \leq \text{len}(\overline{x'x})$—that is, y′ is on the portion of L1 between x′ and x, and so

$$\text{len}(\overline{x'y'}) + \text{len}(\overline{y'x}) = \text{len}(\overline{x'x}) \qquad (5).$$

These observations lead to the identity

$$\begin{aligned}
\text{len}(P_{Sy}) &= \text{len}(P_{Sy'}) + \text{len}(P_{y'y}), \quad \text{by } (2)\\
&= \text{len}(P_{Sx'}) + \text{len}(\overline{x'y'}) + \text{len}(\overline{y'x}) + \text{len}(\overline{xy}), \quad \text{by } (3) \text{ and } (4)\\
&= \text{len}(P_{Sx'}) + \text{len}(\overline{x'x}) + \text{len}(\overline{xy}), \quad \text{by } (5)\\
&= \text{len}(P_{Sx}) + \text{len}(\overline{xy}), \quad \text{by } (1)
\end{aligned}$$

which proves the first statement of the lemma, as P_{Sx} is a shortest path from S to x.

From the above identity, it is apparent that $\text{len}(P_{Sy})$ will be minimized (for y a point on L1) when x=y, and so P_{Sx} is a shortest path from S to L1. As well, since L3 can be reached by a k-segment path (the subpath of P from S to L3), and since $P_{Sx'}$ is a smallest and therefore straightest path from S to L3, we can see that $P_{Sx'}$ can have at most k segments, and so P_{Sx} will have at most k+1 segments—the number of segments in $P_{Sx'}$ plus the single segment $\overline{x'x}$. Since by definition of L1, any path from S to L1 has at least k+1 segments, we conclude that P_{Sx} has k+1 segments, and is thus a straightest path from S to L1. These two facts mean that P_{Sx} is a smallest path from S to L1, proving the second statement of the lemma.

End of nearest point lemma.

Definition: [nearest point]: If L is a line segment in Q and there is some point x on L such that all smallest paths from S to L terminate at x, then x is called the *nearest point* of L relative to S. Since S will usually be understood to be arbitrary but fixed, we generally refer to the nearest point of L. Note that the definition implies that if L has a nearest point, then it has a unique nearest point.

Corollary: Any maximal line segment L in Q has a nearest point relative to any point S in Q.

Proof: Let L be a maximal line segment in Q, and let S be a point in Q. Let P be a shortest path from S to L, which ends at some point x on L. If there were any other point y on L such that there was a path from S to y of the same length as P or less, it would be easy to show that statement 1 of the nearest point lemma was untrue. Hence, it must be the case that all other paths from S to a point y on L, where $y \neq x$, must have length greater than P. This is sufficient to ensure that x must be the nearest point of L relative to S.

Corollary: *Any* rectilinear line segment L′ in Q has a nearest point. That is, a smallest path from S to L′ exists, and all smallest paths from S to L′ end at some unique point x.

Proof: This can easily be seen by letting L be the maximal line segment containing L′, and then doing cases depending on whether or not L′ includes the nearest point of L. Note that if L′ does not contain the nearest point of L, a smallest path from S to L′ may require one more segment than a smallest path from S to L.

Smallest-Path Theorem: For any two points S and T in Q other than vertices such as shown in figure 1, a smallest path from S to T exists.

Proof: Let S and T be two points such that a straightest path from S to T consists of k segments, and let P be a k-segment path from S to T. Choose L to be a maximal line segment containing the kth segment of P. Using the nearest point lemma, let P′ be a smallest path from S to L, ending at some point x on L. The path formed by concatenating P′ and $\overline{xT}$ is a k-segment (and therefore straightest) path from S to T, and by the first statement of the Nearest Point Lemma it is also a shortest path from S to T. Therefore, it is a smallest path from S to T.

3. The Topology of the Smallest-Path Problem

Given a simple polygon Q and points S and T in Q, it is generally the case that there will be large areas of Q through which no smallest path from S to T will pass. This occurs because of the fact that any path from S to T passing through these areas would be longer than a shortest path from S to T. The purpose if this chapter is to show how such areas may be identified, provide geometrical constructs which will be later used for algorithmically removing such areas, and proving some needed properties about what is left after these unnecessary areas are removed.

This is a thesis written by a computing science student, and intended to be read by others in the field of computing science. The author has only a basic knowledge of topology, and assumes the same to be true for most of the readers of the thesis. Nonetheless, the thesis depends upon some nontrivial topological constructs; this creates a conflict, with the topological naivete of the author and readers on one side, and the need for reasonable mathematical rigor on the other.

Fortunately, the needed proofs concern objects in an environment (the Cartesian plane) where most people have a well developed, correct intuition concerning the behaviour of constructs such as paths, polygons, points, and so forth. The following section will thus take a highly diagrammatic, somewhat informal approach to proving the results it needs—while not completely rigorous, this approach will satisfactorily demonstrate the existence of the constructs and theorems necessary to the remainder of the thesis, and has the advantage of being accessible to the reader[1].

The reader may find this chapter somewhat strange in that, while the thesis deals with the problem of finding the dimensions of smallest paths in simple polygons, the entirety of this

1. And, for that matter, the author.

chapter is concerned with eliminating areas of simple polygons through which no *shortest* paths may pass. However, recall that we have already shown that smallest paths in simple polygons are always possible (barring the involvement of pathological vertices); thus, by removing areas of the polygon through which no shortest path can travel, we also remove areas through which no smallest path can travel. Removal of these areas will give us a geometric structure with particularly attractive properties which make finding a smallest path quite easy.

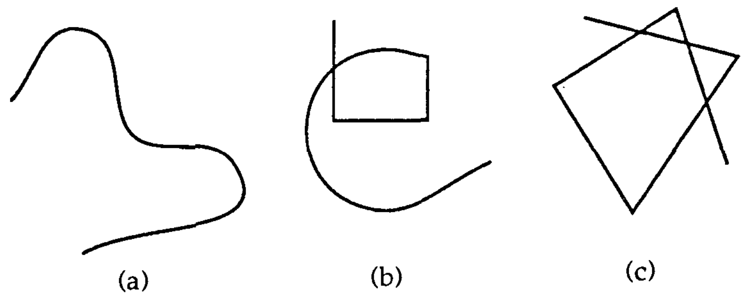
The remainder of this chapter consists of four sections. The first, very short section, contains a few points which do not belong in any other section. The second section shows how, given Q, S, and T, some parts of Q may be removed from consideration, as no shortest and hence no smallest path from S to T will pass through those parts of Q. The third section proves a particularly important lemma concerning those points in regions of Q which the second section shows may be removed. The final section proves that what remains after removing these unnecessary parts of Q is a particularly simple and well-defined structure called the *pseudogon*. This is the structure through which we will eventually find our smallest path

3.1. Miscellaneous

When considering the topology of simple polygons and paths within them, it is frequently convenient to switch between views of geometric constructs as the constructs themselves, defined in some "high-level" manner, and constructs as sets of points. For convenience, we do not go to great length to differentiate these viewpoints, so phrases such as, "a point w is in a line segment L," are common, meaning of course that w is one of the points in the set of points which constitute L.

The following definition is used in later subsections.

figure 3



Examples of arcs. (a) is simple, while (b) and (c) are self-intersecting.

Definition: [arc]: An *arc* is a continuous curve of finite length joining two points on the plane. Examples of arcs are shown in figure 3. A precise topological definition of an arc would involve defining topologically continuous mappings and several other concepts, an onerous task. We trust instead to the intuition of the reader. An arc is *simple* if it does not intersect itself so as to bound a finite region of the plane. (a) in the given examples of arcs is simple, while (b) and (c) are not. An arc which is not simple is called *self-intersecting*. Note that a path is just an arc composed of rectilinear line segments.

The following lemmas are used many times in later sections.

Lemma: Any subpath of a shortest path is shortest.

Proof: [by contradiction]: Let P be a shortest path, and let P′ be a subpath of P. If there were a shorter path P″ between the endpoints of P′, we could substitute P″ for P′ in P, and hence make P shorter. This is not possible, and so P′ must be a shortest path.

Lemma: No self-intersecting path is shortest.

Proof: Obvious.

The following assumptions are necessary in later sections. Their formal proofs would require some fairly intricate topology. Luckily, the fact that they are correct (or at least, that they are correct in the real plane) is obvious.

Definition: [closed curve, simple closed curve]: A *closed curve* is an arc whose endpoints are congruent, i.e. an arc which self-intersects at its endpoints. A *simple closed curve* is an arc which self-intersects only at its endpoints. Simple closed curves are also known as *Jordan curves*. All simple polygons are simple closed curves.

Assumption: Let C be a simple closed curve in the plane, bounding a finite area A, A inclusive of C. If C′ is any closed curve (not necessarily simple) contained in A, then all finite areas bounded by C′ are contained in A.

Assumption: Let C be a simple closed curve, let L be a maximal line segment in C partitioning C into two or more distinct finite area, and let w1 and w2 be two points inside of C but not in the same area of C as induced by L. Any path from w1 to w2 which does not cross C must intersect L at at least one point.

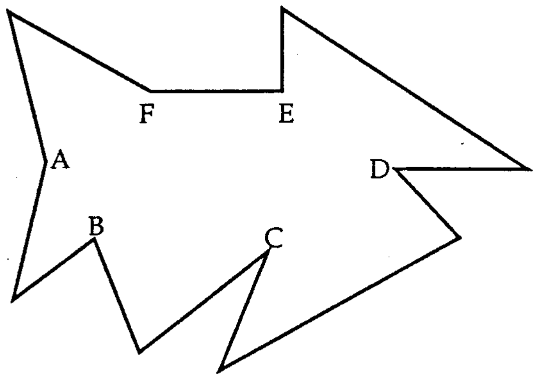
3.2. Unnecessary Regions

We now show that various sections of Q are such that no smallest path from S to T in Q will contain a point in these sections. The strategy will be to show that there are some areas of Q which no shortest path will pass through. Because a smallest path is necessarily a shortest path, it follows that no smallest path will pass through these areas either.

Definitions: [highest, lowest, leftmost, rightmost, extreme, doubly extreme, horizontally extreme, vertically extreme, total extreme points]: See figure 4 for illustrations of the following definitions. Let v be a vertex of Q, concave into Q (i.e. the angle across v taken through Q is greater than 180 degrees). We say v is a *highest* point of Q if its y-coordinate is greater than or equal to the y-coordinates of all points on the two boundary segments of Q incident on v. *Lowest*, *rightmost*, and *leftmost* points are defined in the obvious similar manner. A vertex which is highest, lowest, rightmost, or leftmost is called an *extreme point.* An extreme point which is highest or lowest is a *horizontally extreme* point, and an extreme point which is leftmost or rightmost is a *vertically extreme* point. An extreme point which is both horizontally and vertically extreme is a *doubly extreme point.* A highest point v1 is *total highest* if its y-coordinate is greater than the y-coordinate of any other point v2 on either of the two boundary segments incident on v1, and similarly for lowest, leftmost, and rightmost. A *total horizontal extreme point* is one which is total highest or total lowest, and similarly for total vertical extreme points. A point which is both total vertical extreme and total horizontal extreme is a *total doubly extreme point.*

Definitions: [induced subpolygons, interior and exterior boundaries]: If L=$\overline{ab}$ is a rectilinear line in Q, and there is a subpath B of bdy(Q) from a to b such that

figure 4



A is total rightmost, B is total highest, C is total highest and total rightmost and therefore total doubly extreme. D is highest and total leftmost, and so is doubly extreme but not total doubly extreme. E is lowest and rightmost, but not total in either respect. F is lowest. B, C, D, E, and F are horizontally extreme vertices, while A, C, D, and E are vertically extreme.

$L \cap B = \{a, b\}$ and $L \cup B$ forms the boundary of a polygon Q´, we say that L *induces* the subpolygon Q´ on Q. Q´ will be contained in Q because bdy(Q´) is contained in Q, and Q´ will be simple because its boundary is non-intersecting except at the endpoints of adjacent boundary segments. L is called the *interior boundary* of Q´, denoted ibd(Q´), and B is called the *exterior boundary* of Q´, denoted ebd(Q´).

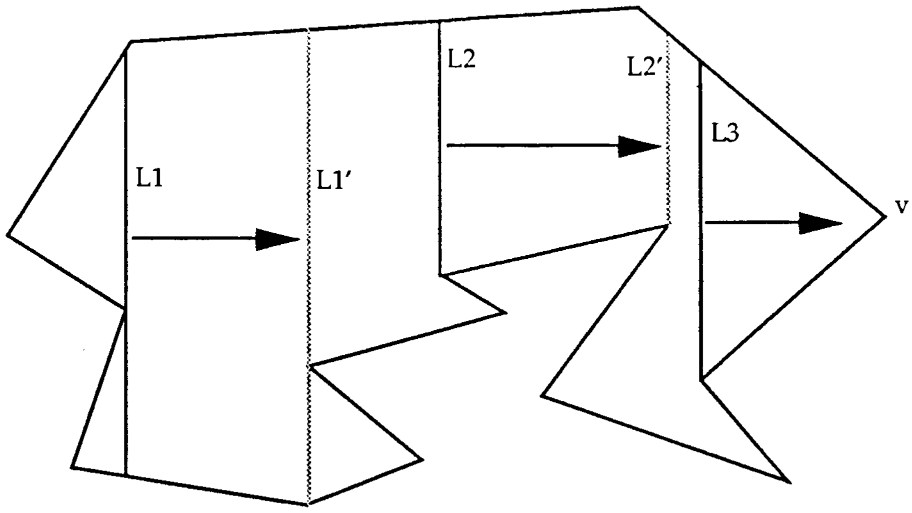
Notation: Here and in the following, we use the '\' symbol to implement relative complement. (Set subtraction.)

Definitions: [region, interior and exterior boundary of a region, unnecessary region, unnecessary point]: Let L be a rectilinear line in Q inducing the subpolygon Q´, and let R be the area of Q defined by R=Q´\L. We say R is the *region* induced by L, foregoing the more general connotations of the word region. We also say that R is the region induced by, or bounded by, Q´. The interior boundary and exterior boundary of R, denoted ibd(R) and ebd(R) respectively, are identical to the interior and exterior boundaries of Q´. Note that while Q´ contains both its interior and exterior boundaries, R does not contain ibd(R), and does not contain the endpoints of ebd(R). If neither S nor T is in R, then R is said to be an *unnecessary* region. If w is a point in an unnecessary region, then w is said to be an *unnecessary point.*

Exclusion Lemma: Let R be an unnecessary region of Q. No smallest path from S to T will contain a point in R.

Proof: Let L be ibd(R), and let P be a path from S to T which passes through a point w in R. Both endpoints of P must be outside of R, since R is an unnecessary region, and so there will be a subpath of P which starts and ends in Q\R, and passes through the point w in R; hence, if we view P as a directed path, we can see that P must either start on L or cross L before it reaches w, and must either end on L, or cross L after it reaches w. Let P´ be the subpath of P s.t. P´ contains w and the endpoints of P´ are on L, but no other point of P´ is

figure 5



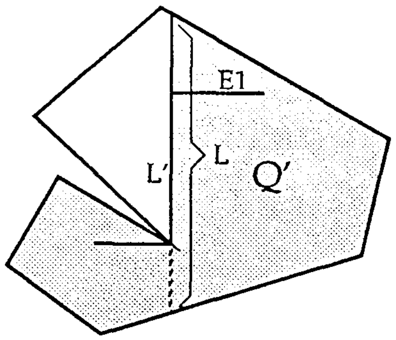
Examples of sliding. L1 may be slid to L1′, or any point in between, but not past L1′. L2 may be slid to L2′ or any point in between, but not past L2′. L3 may be slid to the right, but not to v.

on L. Because L is rectilinear and w is not on L, P′ will be longer than the subsegment of L between the endpoints of P′, so by substituting for P′ the subsegment of L which joins the two ends of P′, we could make P shorter, and so P cannot be a smallest path.

Definition: [slide]: Let L be a vertical line segment in Q with both endpoints on bdy(Q) such that neither endpoint of L is a rightmost extreme point, and no other point of L intersects a leftmost extreme point. We can *slide* L to the right by continuously increasing the x-coordinate of C, while continuously adjusting the y-coordinates of the endpoints of C so that the endpoints remain on bdy(Q). Sliding L right, past a point where one of L's endpoints is on a rightmost extreme point, or past a point where a non-endpoint of L intersects a leftmost extreme point, or to a point where the endpoints of L merge, is not defined. See figure 5 for examples of sliding.

Definition: [orientation of subpolygons or regions relative to the line segments inducing them]: Let Q′ be a subpolygon (R be a region) induced by the vertical line segment L′,

figure 6



The shaded subpolygon Q′ is induced by the solid line L′, contained in the line L. Q′ is to the right of L (and to the right of L′) as shown by the existence of E, but is not to the left of L, as the only line segment on L′ projecting to the left into Q′ is on an endpoint of L′.

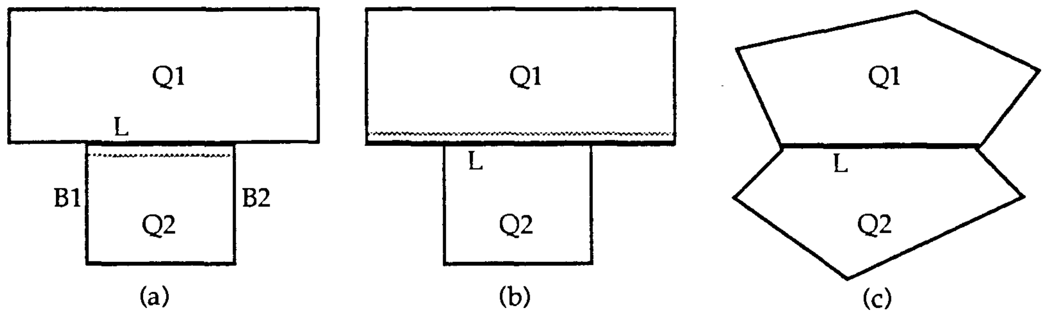
where L′ is contained in the vertical line segment L. We say Q′ (R) is to the right of L if there is some point w on L′ but not an endpoint of L′, and some line segment E perpendicular to L′ and with its left endpoint at w, such that E is in Q′ (E\{w} is in R). See figure 6. We can in an obvious similar manner define what it means for Q′ (R) to be to the left of L, or if L is horizontal, what it means for Q′ (R) to be to the above or below L.

Definition: [partial pseudochord]: Let L be a vertical line segment in Q containing a vertical extreme point or S or T, or a horizontal line segment containing a horizontal extreme point or S or T, such that L induces a region R. Then L is called a *partial pseudochord.*

Lemma: Any unnecessary region R1 induced by a rectilinear line segment L is contained in an unnecessary region R2 induced by some partial pseudochord.

Proof: [constructive]: WLOG, let R1 be an unnecessary region induced by and to the left of the vertical line segment L. If L contains a vertical extreme point or S or T, then we are

figure 7



In (a), the thick line L pseudochord-induces Q2, as can be seen by the presence of the dotted line connecting points on B1 and B2, but L does not pseudochord-induce Q1. In (b), the reverse is true. In (c), L is not a pseudochord, as it does not contain a horizontal extreme point and hence is not a partial pseudochord.

done, otherwise we can slide L to the right, until one of the endpoints of L reaches a rightmost extreme point or S or T, or until a non-endpoint of L intersects a leftmost extreme point or S or T. One of these events must occur, for if they did not, it would mean that neither S nor T were in the region to the right of L, and hence were not in Q. After we have finished sliding L, it contains a vertical extreme point or S or T, and the region R2 induced by and to the left of L is unnecessary and contains R1, so the lemma is proved.

Definitions: [half-plane subpolygon, pseudochord, pseudochord-induced subpolygons and regions]: Let L be a line segment in Q inducing a subpolygon Q′. Let B1 and B2 be the two boundary segments of ebd(Q′) which are incident on the endpoints of L. If there is a rectilinear line segment L′, parallel to but distinct from L and contained in Q′, such that one endpoint of L′ is on B1 and the other endpoint of L′ is on B2, then Q′ is a *half-plane* subpolygon. A line segment in Q is a *pseudochord* if it is a partial pseudochord and induces a half-plane subpolygon Q′; Q′ is said to be *pseudochord-induced* by L. The region R′=Q′\L is also said to be pseudochord-induced by L. See figure 7 for examples.

Inclusion Lemma: All points in an unnecessary region are in an unnecessary region pseudochord-induced by some pseudochord.

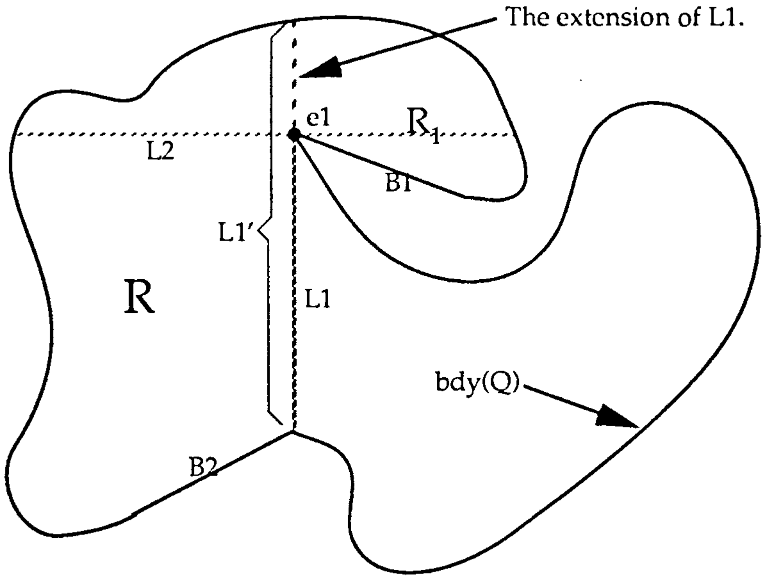
Proof: Let w be a point in an unnecessary region R1 induced by a rectilinear line segment. By a previous lemma, there is a partial pseudochord L1 inducing an unnecessary region R containing R1 and hence w.

Now assume WLOG that L1 is vertical and R is to the left of L1. Let Q1=ebd(R)$\cup$L1$\cup$R be the subpolygon bounding R, and let B1 and B2 be the two segments of ebd(R) incident on the endpoints of L1. If B1 and B2 can be joined with a vertical line in R, then L1 is a pseudochord which pseudochord-induces R, and we are done. Assume that B1 and B2 cannot be joined with a vertical line in R. The remainder of this proof will be done with the aid of a diagram.

Consider figure 8. It shows R, L1, B1, and B2, with B2 in the half-plane to the left of and including L1, and B1 in the half-plane to the right of and including L1. Note that R includes the region labeled R_1. Because B1 goes the "wrong way", L1 is not a pseudochord. However, if we form the line segment L1´ containing L1 by projecting the ends of L1 up and down in Q as far as they will go (in the case of figure 8, this results in an extension to L1 upwards), then L1´ will contain one or more pseudochords pseudochord-inducing unnecessary regions containing all of the points in R strictly to the left of L1´, and one or more pseudochords pseudochord-inducing regions containing all of the points in regions R to the right of L1´. For example, in figure 8, L1´ itself is a pseudochord pseudochord-inducing an unnecessary region containing all points in R to the left of L1´, and the extension that was added to L1 is a pseudochord which pseudochord-induces an unnecessary region containing all points in R to the right of L1´—those points in the region R_1. Thus, L1´ contains pseudochords which pseudochord-induce unnecessary regions containing all points in R except those points of R which are on L1´\L1.

Now, let e1 be the top endpoint of L1. If L1´ extends above e1 (as in figure 8) we form a line segment L2 by extending e1 as far to the left and the right as it will go in R, as shown

figure 8



in the diagram. L2 will contain a pseudochord pseudochord-inducing an unnecessary region containing all of the points in L1′\L1 above e1. (In the diagram, L2 itself is this pseudochord.) If necessary, we could construct a similar line segment with the bottom endpoint e2 of L1, and so show that some pseudochord induces an unnecessary region containing all points in L1′\L1 below e2. This construct will work even if B1 or B2 or both are parallel to L1.

The above shows that every point in R is in an unnecessary region pseudochord-induced by some pseudochord. Since w is in R1 and R1 is in R, w is in an unnecessary region induced by some pseudochord. Because w was chosen arbitrarily from those points in R1, which was itself an arbitrary unnecessary region, it follows that all points in unnecessary regions are in an unnecessary region pseudochord-induced by some pseudochord.

This ends the proof for the inclusion lemma.

Lemma: For any polygon Q with n vertices, there are at most O(n) pseudochords in Q.

Proof: WLOG, let w be a highest extreme point in Q. If there are any horizontal pseudochords with one endpoint at w and extending right from w, then there will be at most two such horizontal pseudochords, one inducing a region above itself, and the other inducing a region below itself. Likewise, there are at most two horizontal pseudochords with an endpoint at w and extending to the left from w, at most two vertical pseudochords with an endpoint at w and extending up from w, and at most two vertical pseudochords with an endpoint at w and extending down from w.

Now form a line $\overline{w1w2}$, where w1 is the point found by projecting w to the left until the projection intersects a lowest point of Q or until taking the projection any further to the left would take it outside of Q, whichever comes first, and where w2 is found in the same manner by projecting w to the right. $\overline{w1w2}$ is the only horizontal pseudochord which contains w as a non-endpoint. Likewise, there is at most one (there may not be any) vertical pseudochord containing w as a non-endpoint.

Thus, each extreme point in Q may be contained in at most 10 distinct pseudochords[2]. It is also easy to show that S and T can be contained in at most a constant number of pseudochords. Since each pseudochord must by definition contain at least one extreme point, there are O(n) pseudochords in Q, where n is the number of vertices in Q. As well, each pseudochord can pseudochord-induce at most two regions—for instance, a horizontal pseudochord HP will pseudochord-induce at most one region above HP and at most one region below HP. Therefore, the number of pseudochord-induced regions in Q to test for being unnecessary is at most O(n), and the number of unnecessary regions which must be removed from Q, in order to remove all unnecessary points from Q, is at most O(n).

2. The actual bound is much lower than this, but proving this lower bound is more difficult.

3.3. The Bad Point Lemma

This section proves an important lemma concerning the points contained in unnecessary regions.

Definition: [bridge]: If P is a path in Q and there is a rectilinear line segment L in Q such that the endpoints of L are on P but L is not contained in P, then L is a *bridge* for P.

Lemma: If P is a shortest path, then there are no bridges for P.

Proof: Let P be a shortest path and assume it has a bridge L. We can take the minimal subpath P′ of P and the minimal subsegment L′ of L such that L′ is a bridge for P′. L′ and P′ will intersect only at their endpoints, and since L′ is a rectilinear line segment and P′ is not identical to L′, it is apparent that the distance traveled by P′ is longer than the distance traveled by L′. However, this means that by substituting L′ for P′ in P, we can make P shorter, which contradicts the fact that P is a shortest path, and so our assumption that P has a bridge must be incorrect.

Definition: [3-path]: Any rectilinear path composed of a sequence of three line segments is a *3-path.*

Lemma: Let P be a rectilinear path. If all 3-paths in P are shortest, then P is shortest.

Proof: [by induction]:

Basis: For paths P of one or two segments, the lemma is vacuously true. For a path P having three segments, the lemma is trivially true.

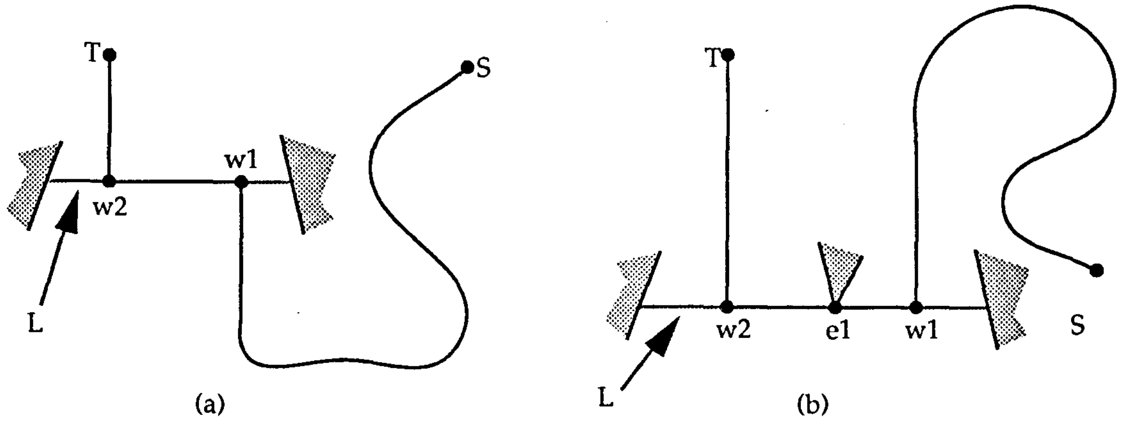
Inductive Assumption: For some integer k, k≥3, assume for all paths P having k or fewer segments that if all 3-paths contained in P are shortest, then P is shortest.

Inductive Step: Let P be a path from a point S to a point T with k+1 segments, such that all 3-paths contained in P are shortest. Let P_i denote the subpath of P consisting of the first i segments of P, counting from the segment incident on S. If $i \leq k$, we have by the inductive assumption that P_i is shortest.

WLOG orient P so that the k+1st segment of P (incident on T) is vertical and intersects the kth segment of P at its lower endpoint. There are two possible configurations for the last three segments of P, as shown in figure 9. We call the configuration in (a) the 'Z' configuration, and the configuration in (b) the 'U' configuration. In either case, let L be the maximal line segment in Q containing the kth segment of P. The rest of the proof will be done referring to the illustrations in figure 9, using their orientations.

In (a), it is apparent that any path from S to T must intersect L. In (b), because (by assumption) the last three segments of P form a shortest 3-path, there must be some

figure 9



The two configurations (barring reflections and rotations) for the last three segments of a k+1 segment path, k > 3.

lowest extreme point e1 of bdy(Q) incident on the kth segment of P, such that there are no bridges between the k-1st and k+1st segments of P, as shown in the diagram; otherwise, there would be a non-shortest 3-path in P. Some subsegment L´ of L to the right of (and possibly including) e1 will induce a region R containing S but not T. Any path from a point in R to a point outside of R, such as T, will have to intersect L´, and so must also intersect L. Therefore, in both (a) and (b), any path and hence any shortest path from S to T must intersect L.

Let w1 and w2 be the right and left endpoints, respectively, of the kth segment of P. In both (a) and (b), the nearest point of L relative to S must be on or to the right of w1, otherwise we could use the nearest point lemma to show than P_k (the subpath of P from S to w2) was not a shortest path. Let N be the nearest point of L relative to S, and let P´ be a shortest path from S to T, which as previously shown must intersect L at some point w3. By a previous lemma the subpath P'_{w3T} of P´ from w3 to T will be a shortest path, and since there is a two segment path from any point on L to T, the length of P'_{w3T} will be just the rectilinear distance between w3 and T, which can be expressed as

$$\text{len}(P'_{w3T}) = \text{len}(\overline{w3w2}) + \text{len}(\overline{w2T}) \qquad \textit{(1).}$$

If D is the length of a shortest path from S to L, then by the nearest point lemma we have that

$$\text{len}(P') = D + \text{len}(\overline{Nw3}) + \text{len}(\overline{w3w2}) + \text{len}(\overline{w2T}) \qquad \textit{(2).}$$

A path of length D + len($\overline{Nw2}$) + len($\overline{w2T}$) from S to T can be constructed by taking a shortest path from S to L, and adding to it the segments $\overline{Nw2}$ and $\overline{w2T}$—since P´ is a shortest path from S to T, we have

$$\text{len}(P') \leq D + \text{len}(\overline{Nw2}) + \text{len}(\overline{w2T}) \qquad \textit{(3).}$$

From *(2)*, however, it is easy to see that

$$\text{len}(P') \geq D + \text{len}(\overline{Nw2}) + \text{len}(\overline{w2T}) \qquad \textit{(4),}$$

and so by *(3)* and *(4)*, len(P´) = D + len($\overline{Nw2}$) + len($\overline{w2T}$). Since len(P_k) = D + len($\overline{Nw2}$), the length of a shortest path from S to w2, and since the subpath of P from S to w2 is a shortest path, we have that len(P′) is the length of P; and since P′ is a shortest path from S to T, it follows that P is a shortest path from S to T.

This completes the proof of the 3-path lemma.

Definition: [bad point]: A point w in Q is called a *bad point* if the sum of the lengths of shortest paths from S to w and from w to T is greater than the length of a shortest path from S to T.

Bad Point Lemma: All bad points are unnecessary.

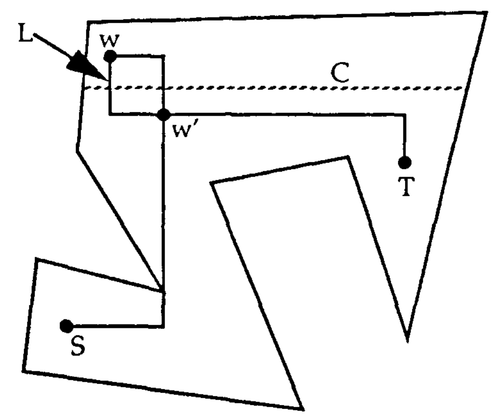
Proof: [constructive]: For any bad point w, we will show how to construct a line segment containing a subsegment which induces an unnecessary region containing w.

Let w be a bad point in Q, let P1 be a shortest path from S to w, and let P2 be a shortest path from T to w. We consider two cases; either P1 and P2 intersect at some point other than w, or they do not.

1) If P1 and P2 intersect at a point other than w, then let w′ be the point on P1 nearest S which intersects P2. Because subpaths of shortest paths are also shortest paths, the subpath $P1_{w'w}$ of P1 from w′ to w must be the same length as the subpath $P2_{w'w}$ of P2 from w′ to w. Therefore, if we modify P1 by replacing $P1_{w'w}$ with $P2_{w'w}$ then P1 will still be a shortest path from S to w. See figure 10.

   Because P1 and P2 intersect and follow one another on their approach to w, there is some rectilinear line segment L in Q with one end at w such that the segments of P1 and P2 incident on w both contain L. Take any point on L other than its endpoints, and construct a chord C perpendicular to L through this point. C induces two subpolygons Q1 and Q2 on Q, with w in Q1\C, and the segments of P1 and P2

figure 10

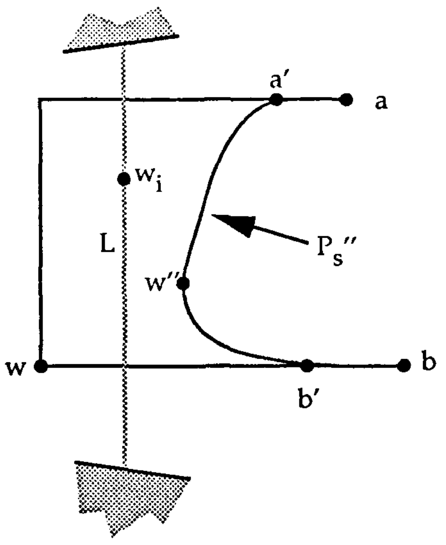


One way of obtaining L and a chord C whicn induces an unnecessary region containing w.

incident on w both crossing into Q2. P1 and P2 cannot intersect C at any other point, for if either did, C would contain a bridge for P1 and/or P2, which is impossible, as P1 and P2 are shortest paths. Since P1 and P2 cannot intersect C at any other point, their other endpoints S and T must be in Q2, and not in Q1. Therefore, C induces the unnecessary region Q1\C containing w.

2) Assume P1 and P2 do not intersect at any point other than w; then we can form a path P from S to T by appending P2 to P1. w is a bad point, and therefore P is not a shortest path from S to T, and so by the 3-path lemma we have that P contains some non-shortest 3-path P′. Because P1 and P2 are both shortest, neither will contain a non-shortest 3-path, and so P′ must have one segment in one of P1 or P2, say P1, and the other two segments in the other of P1 and P2, in this case P2. If the three segments of P′ form a ‘Z’ configuration, as for example do the last three segments of the path from S to T shown in figure 9(a), then P′ cannot be a non-shortest 3-path, so the three segments of P′ must form a ‘U’ configuration, as shown in figure 11. Because P′ is non-shortest, we can find a shortest path P′′ between the ends a and b of P′. Using the orientation and constructs of figure 11, we can see

figure 11



Grey regions indicate ext(Q). P′ is the three-segment rectilinear path connecting a and b. P″ is the path formed by moving from a to a′, from a′ to b′ along P_s'', and then moving from b′ to b. P_s' is the path from a′ to w to b′. The simple curve G mentioned in the text is the curve found by starting at b′, going along P_s'' to a′, going left and then down the rectilinear segments to w and then going right to b′.

that P′′ cannot intersect any point to the left of or on the vertical line through w, for if P′′ did, it would necessarily be at least as long as P′. (This can be seen through simple vector addition.)

Now, consider both P′ and P′′ as directed paths from a to b. Let a′ be the point farthest from a on P′′ such that the subpaths of P′ and P′′ from a to a′ are identical, and let b′ be the point farthest from b on P′′ such that the subpaths of P′ and P′′ from b′ to b are identical. a′ and b′ will both be on both P′ and P′′. Let P_s' be the subpath of P′ from a′ to b′ (s stands for subpath in this case, and has nothing to do with the point S), and let P_s'' be the subpath of P′′ from a′ to b′. P_s'' will be shortest because it is a subpath of a shortest path; it cannot intersect the lower or upper horizontal segments of P′ at any other points than a′ and b′, for if it did, then one of the horizontal segments of P′ would contain a bridge for P_s''. As was previously shown, P'' cannot intersect any point on or to the left of the vertical segment of P', and so neither can P_s''. Therefore, P_s'' and P_s' intersect at only the points a' and b', and so form a simple closed curve G, as shown in figure 11. Since G is contained in Q, all points in G will also be in Q.

Let w'' be the leftmost point of P_s'', and let L be a maximal vertical line segment passing through some point w_i, where w_i is to the right of w, to the left of w'', above b', and below a'. w_i will be an interior point of G, and by virtue of the fact that all points in G are in Q, L will certainly contain single points on the upper and lower horizontal segments of P', which is to say, on P1 and P2 (which, recall, are shortest paths from S to w and from w to T respectively). However, P1 and P2 cannot intersect L at any other point, otherwise L would contain bridges for P1 and/or P2. Therefore, both S and T will be in region(s) generated by and to the right of L, and w will be in some region R generated by and to the left of L. Because R does not contain either of S or T, it is unnecessary, and the lemma is proved for this case.

End of bad point lemma.

Corollary: Every point not in an unnecessary region of Q is contained in some shortest path from S to T.

3.4. Structure of the Pseudogon

Definitions: [pseudogon, cutQ]: Let Q be a polygon with points S and T in Q between which we wish to find a smallest path, and let cutQ be the structure resulting when all unnecessary points have been removed from Q. We call cutQ the *pseudogon* for Q. Since the simple polygon in which we are trying to find a smallest path is for convenience always named Q, we will in the following use "cutQ" and "the pseudogon" interchangeably, to avoid monotony.

The remainder of this section shows that the pseudogon will have a particularly simple structure, well-suited for efficient analysis by computer.

3.4.1. General Properties of the Pseudogon

Lemma: cutQ can be expressed as a finite union of simple polygons, line segments, and points.

Proof: Let cutQH be the structure formed from Q by removing all points of Q contained in unnecessary regions of Q pseudochord-induced by horizontal pseudochords of Q, and let cutQV be the construct formed from Q by removing all points of Q in unnecessary regions pseudochord-induced by vertical pseudochords. It is apparent that both cutQH and cutQV can be expressed as a finite union of simple polygons and line segments. cutQ is just cutQH $\cap$ cutQV. Because the intersection of a simple polygon or line segment with another simple polygon or line segment is always expressible as a finite union of simple polygons, line segments, and points, and because cutQH and cutQV are both expressible as a finite union of line segments and simple polygons, it is therefore apparent that cutQ = cutQH $\cap$ cutQV is expressible as a finite union of points, line segments, and simple polygons. In fact, as will become apparent, cutQ is always expressible as a finite union of line segments and simple polygons.

Definitions: [boundary points of a pseudogon, subpolygons and isolated line segments of a pseudogon]: Let cutQ be the pseudogon generated by removing all unnecessary points from Q. The *boundary points* of cutQ are exactly those points w satisfying at least one of the following:

1) w is on a pseudochord of Q which pseudochord-induces an unnecessary region, but

w is not itself an unnecessary point.

2) w is a point in bdy(Q) and is not an unnecessary point.

If Q′ is a simple polygon whose boundary is a subset of the set of boundary points of cutQ, then we say Q′ is a *subpolygon* (or just a polygon) of cutQ. If L is a line segment in cutQ such that no subsegment of L forms part of a boundary segment for a subpolygon of cutQ, then L is an *isolated line segment* of cutQ.

Since there are a finite number of pseudochords, and a finite number of unnecessary regions which will thus be removed from Q to produce cutQ, it is easy to see that the boundary points of cutQ will be contained in a finite number of line segments. We now turn our attention to making more powerful statements.

Lemma: cutQ is connected. That is, for any two points in cutQ, there is a rectilinear path in cutQ connecting them.

Proof: By a corollary of the bad point lemma, all points in cutQ are on some shortest path from S to T. Since all shortest paths from S to T are in cutQ, as was shown by a previous lemma, it follows that for any two points w1 and w2 in cutQ, there must be a rectilinear path from S to w1 in cutQ, and a rectilinear path from S to w2 in cutQ, and hence a rectilinear path from w1 to w2 through S in cutQ.

Corollary: cutQ may be expressed as a finite union of isolated line segments and simple polygons (because any single point of cutQ must be contained in a subpolygon or isolated line segment of cutQ, in order for cutQ to be connected.)

Definition: [arcwise connected]: We say a set of points X is *arcwise connected* if for any two points w1 and w2 in the set and a finite distance apart (using either the Euclidian or rectilinear metrics, it doesn't matter which) there is some arc in X connecting w1 and w2. A set which is not arcwise connected is *arcwise disconnected.*

Lemma: If two sets X and Y are arcwise connected, and there is a path from a point in X to a point in Y such that the path is in $X \cup Y$, then $X \cup Y$ is arcwise connected.

Proof: Obvious.

Definitions: [U, exterior of a set of points]: For convenience, we henceforth reserve the letter U to denote the real plane, the universe of discourse. If X is a set of points in U, then the *exterior* of X, denoted ext(X), is just $U \backslash X$.

Lemma: ext(cutQ) is arcwise connected.

Proof: Let w1 and w2 be any two points a finite distance apart in ext(cutQ). If they are both in ext(Q), there there is obviously a path between them in ext(Q) and hence in ext(cutQ). If w1 is in ext(Q) and w2 is in a pseudochord-induced unnecessary region R of Q, then from the fact that R is arcwise connected within itself (obvious) and from the fact that R will include some part of bdy(Q) (obvious), it is apparent that we can draw a path in R from w2 to a point w3 on bdy(Q) and in R, and then draw a path in $R \cup$ext(Q) from w3 to w1, and so construct a path in $R \cup$ext(Q) from w2 to w1. If both w1 and w2 are in pseudochord-induced unnecessary regions of Q, then we can draw a path in ext(cutQ) from w1 to a point w3 in ext(Q), and then a path in ext(cutQ) from w3 to w2, and thus construct a path in ext(cutQ) from w1 to w2.

Definitions: [ball, interior of a set of points]: Let w be a point in U, the real plane; the *ball of radius r centred at w*, denoted $B_r(w)$, is the set of all points w′ in U such that the distance from w to w′ is less than or equal to r. If we use the Euclidian metric, then a ball centred at w will actually be a disc centred at w, and if we use the rectilinear metric, then a ball centred at w will actually be a diamond centred at w. Either of these types of balls is adequate for our purposes, and because it is more familiar, we will use the disc. A point w in a set of points X is said to be an *interior point* of X if there is some finite, positive real number r such that $B_r(w)$ is a subset of X. The *interior* of X, denoted int(X), is the set of all interior points of X. If X is a simple polygon, then this definition of interior is in accord with the previous, less formal definition for the interior of a simple polygon.

Lemma: No point of bdy(cutQ) is in int(cutQ).

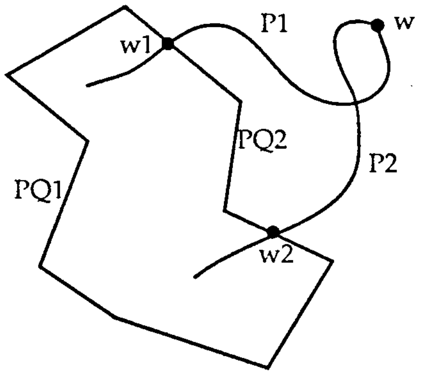
Proof: Let w be a point in bdy(cutQ). Then w is either a point on bdy(Q), or a point on a pseudochord L pseudochord-inducing an unnecessary region R in Q, or both. If w is a point on bdy(Q), then every ball of finite radius centred at w will include points in ext(Q), and hence in ext(cutQ). If w is a point on a pseudochord L which pseudochord-induces an unnecessary region R, then every ball of finite radius and centred at w will include point in R, and hence in ext(cutQ). This proves the lemma.

3.4.2. Subpolygons in the Pseudogon

Entry Lemma: Let Q′ be a subpolygon of cutQ, and let w be a point in cutQ\Q′. Then there is a unique point w1 on bdy(cutQ) such that any path from w to any point in Q′ must contain w1.

Proof: Let P1 and P2 be paths in cutQ from w to points in Q′. Because they connect a point in Q′ with a point not in Q′, both P1 and P2 will have to intersect bdy(Q′). Assume that w1 is the first point along P1 at which P1 intersects bdy(Q′), and that w2 is the first point along P2 at which P2 intersects bdy(Q′). If for all possible P1 and P2 w1=w2, then we are done, otherwise assume $w1 \neq w2$. As shown in figure 12, w1 and w2 divide bdy(Q′) into two paths PQ1 and PQ2 such that $PQ1 \cup P1 \cup P2$ forms a closed curve (not necessarily

figure 12



If P1 and P2 are both in cutQ, then because ext(cutQ) is arcwise connected we have that the area bounded by P1, P2, and PQ1 is in cutQ. This would mean that points on PQ2 (except possibly w1 and w2) would be interior points of cutQ, and hence not boundary points of cutQ.

simple) bounding a finite area G containing PQ2. Assume some point w´ in int(G) is in ext(cutQ); any path from w´ to a point in ext(Q) would have to cross one of PQ1, P1, or P2, which would contradict a previous lemma stating that ext(cutQ) is connected. Thus, G must be in cutQ. However, this means that some point w´´ on PQ2 would be in int(cutQ), which is not possible, as all points on bdy(Q´) and hence all points on PQ2 are boundary points of cutQ, and a previous lemma shows that no boundary points of cutQ are interior points of cutQ. Thus, our previous assumption that w1 and w2 are distinct was erroneous; w2 must be identical to w1, and since P2 was a path from w to an arbitrary point in Q´, it follows that any path in cutQ from w to a point in Q´ must contain w1.

Lemma: Both S and T are on bdy(cutQ).

Proof: We show that S is not in int(cutQ). If S is on bdy(Q), then it must be on bdy(cutQ), and we are done. If it is not on bdy(Q), then it is in int(Q); in this case, there will be one or two vertical pseudochords through S, pseudochord-inducing regions R1 and R2 to the right and left of S; T cannot be in both R1 and R2, so at least one of R1 and R2 will be unnecessary. Thus, S will be a point on a pseudochord which pseudochord-induces an unnecessary region of Q, and so by definition will be on bdy(cutQ.)

Definition: [required points]: A point w in cutQ is a *required point* if every shortest path from S to T in cutQ contains w.

Required Point Lemma: Every subpolygon Q′ of cutQ contains two distinct required points on its boundary.

Proof: Let Q′ be a subpolygon of cutQ. We proceed by cases:

1) Assume neither S nor T is in Q′. By a corollary to the bad point lemma, every point in Q′ is contained in some shortest path from S to T, and by the entry lemma, there is a point w1 on bdy(Q′) such that every path from S to a point in Q′ must contain w1, and a point w2 on bdy(Q′) such that any path from T to a point in Q′ must contain w2.

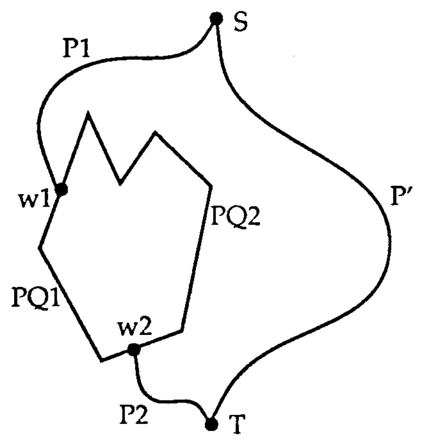
   First we show that w1 and w2 are distinct. Assume w1 and w2 are the same; then let w3 be another point in Q′; there will be some shortest path from S to T containing w3, and by the nature of w1 and w2, this path must have a subpath going from S to w3 through w1, and a subpath going from w3 to T through w2; this would mean that the path was self-intersecting and so not shortest; thus, our assumption that w1 and w2 were the same must be wrong, and hence w1 and w2 are distinct.

   From the definitions of w1 and w2, any shortest path from S to T containing a point in Q′ must contain w1 and w2. Now, if all shortest paths from S to T contain both w1 and w2, we are done; otherwise, there is a shortest path P from S to T containing w1 and w2, and a shortest path P′ from S to T which does not contain any point of Q′. Let P1 be the subpath of P from S to w1, and let P2 be the subpath of P from w2 to T.

The situation is as shown in figure 13. w1 and w2 split bdy(Q′) into two paths PQ1 and PQ2 from w1 to w2 such that P1∪PQ1∪P2∪P′ bounds a finite area containing PQ2. However, by an argument identical to one used in the entry lemma, this would then mean that all points on PQ2 except possibly its endpoints would be in int(cutQ), and so could not be boundary points for a subpolygon of cutQ. This is a contradiction, and so our assumption that there is some shortest path from S to T not containing w1 and w2 must be incorrect, and w1 and w2 are required points.

2) If exactly one of S or T, say T, is in Q′, then by the immediately preceding lemma T is on bdy(Q′) and forms one required point for Q′. Since S is outside of Q′, there is a point w1 such that any shortest path from S to a point in Q′ (such as T) contains w1. If w1 is distinct from T, then we are done. If w1 is not distinct from T, then take a point w2 in Q′, distinct from w1; there must be a shortest path P from S to T containing w2; by the nature of w1(=T), P must go from S through T to w4, which makes it apparent that P is not a shortest path from S to T. Our assumption that w1=T must therefore be incorrect, and w1 is our second required point.

figure 13



The finite area enclosed by P′, P1, P2, and PQ1 must be in cutQ, and therefore the points along PQ2 will be in int(cutQ).

3) If both S and T are in Q´, then they must both be on bdy(Q´). By assumption, S≠T, and so S and T themselves are our required points.

From the immediately previous proof, it is apparent that for any subpolygon Q´ of cutQ, we can find two distinct required points w1 and w2 on bdy(Q´) (with one or both possibly being S and/or T) such that for any shortest path P from S to T, P will have a subpath P1 from S to w1 shorter than any other subpath of P from S to a point in Q´, and P will have a subpath P2 from w2 to T shorter than any other subpath of P from a point in Q´ to T.

Definition: [points of subpolygons nearest S and T]: Let w1 and w2 be as indicated above; we call w1 the *point of Q´ nearest S*, and w2 is the *point of Q´ nearest T*. Together, w1 and w2 are called the *nearest points* of Q´.

Lemma: If Q´ is a subpolygon of cutQ, then any shortest path P from S to T contains a subpath P´ which is a shortest path in Q´ between the nearest points of Q´.

Proof: Let w1 be the point of Q´ nearest S, and let w2 be the point of Q´ nearest T. Because w1 and w2 are nearest points, and hence required and distinct, any shortest path P from S to T will have to contain a shortest subpath P´ from w1 to w2. If P´ is in Q´ then we are done, otherwise assume P´´ is a subpath of P´ such that the endpoints of P´´ are in Q´ but the other points of P´´ are not in Q´. P´´ together with some subsection of bdy(Q´) will bound a finite area containing the rest of bdy(Q´) (the situation is similar to that shown in figure 12), and as in previous such situations, we can show that this means some points of bdy(Q´) must be in int(cutQ), which is a contradiction. Thus, our assumption that some subpath of P´ is not contained in Q´ must be false, and P´ satisfies the lemma.

Lemma: Let Q´ be a subpolygon of cutQ, let P be a shortest path from S to T, and let P´ be a subpath of P between the nearest points of Q´. The points in P´ are the only points in Q´ on P.

Proof: Suppose w is a point of Q´ on P but not on P´. Assume WLOG that w is on the subpath of P from S to the point w1 of P´ nearest S. w cannot be w1, because w1 is a point of P´ but w is not. But then the subpath of P from S to w, a point in Q´, does not contain w1; this is impossible, as w1 was chosen to be the point of Q´ nearest S. Thus, our assumption that w is not on P´ must be wrong.

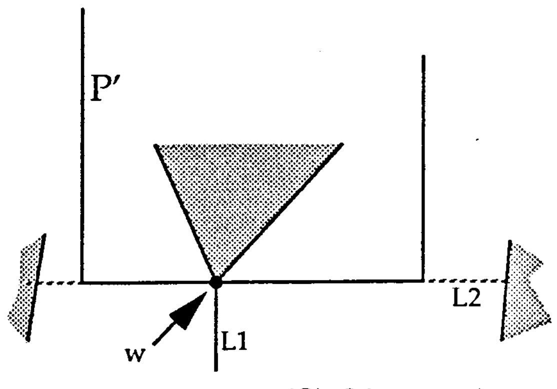
Lemma: Let Q´ be a subpolygon of cutQ, and let P be any shortest path in Q´. No 3-path in P forms a 'U' configuration.

Proof: Assume some subpath P´ of P does form a 'U', as shown in figure 14. We use the orientation and labeling of figure 14 for the remainder of this proof. Because P is a shortest path, there must be no bridge connecting the vertical segments of P´; the only thing which could prevent the existence of such a bridge is the presence of a lowest extreme point w of bdy(Q) on the horizontal segment of P´, such as shown in figure 14. Because Q and Q´ are both simple polygons, w cannot be contained in any segment of bdy(Q) or bdy(Q´) other than the two segments of bdy(Q´)$\subseteq$Q for which w is an endpoint, and so there exists a vertical line segment L1 of some finite length and contained in Q and in Q´, such that one endpoint of L1 is on w and the rest of L1 is below w and contained in int(Q´)$\subseteq$int(Q). Let L2 be the maximal horizontal line segment in Q containing w. L2 will generate a region R containing the points in L1\{w}. P in Q cannot intersect L2 at any point not on the horizontal segment of P´, for if it did, then a bridge would exist for P, and P would be not be a shortest path in either Q or cutQ. This makes its apparent that neither S nor T can be in L2$\cup$R, and so R is an unnecessary region. However, if R is an unnecessary region, then L1\{w} cannot be in cutQ, let alone in int(Q´), which is a contradiction. Therefore, our original assumption that there is some subpath of P which forms a 'U' must be incorrect.

Definitions: [doubly-monotone path, doubly-monotone polygon, ends of a doubly-monotone polygon, funnel]: Let P be a directed path composed of straight (not necessarily rectilinear) line segments; P is a *doubly-monotone path* if every segment of P, considered as a vector, points into the same quadrant of the plane as every other segment of P. A simple polygon Q is a *doubly-monotone polygon* if bdy(Q) can be considered to be made up of two doubly-monotone paths P1 and P2, joined at their endpoints. The two points formed by the intersection of the endpoints of P1 and P2 (i.e. the two points in P1 $\cap$ P2) are called the *ends* of Q. If Q is a doubly-monotone polygon and bdy(Q) is such that there is a rectilinear boundary segment of Q adjacent to each end of Q, then Q is a *funnel.* Requiring that Q have rectilinear boundary segments adjacent to each of its ends is necessary and sufficient to ensure that a rectilinear path may be constructed within Q and between its ends. See figure 15 for examples of funnels.

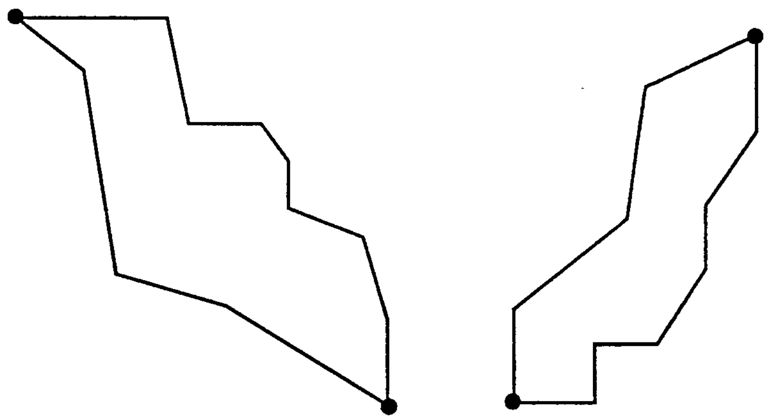
Note that if Q is a rectangle, then it is a funnel, but its ends are not uniquely determined—they may be either pair of diagonally opposite vertices of Q, depending on the context. If Q is a funnel which is not a rectangle, then this problem disappears, and the ends of Q will always be unique.

figure 14



Grey regions denote ext(Q). L1 must exist in both Q and Q′, otherwise they are not simple polygons. However, L2 generates an unnecessary region containing L1\{w}, and so L1 cannot exist in Q′.

figure 15



Two examples of funnels, with ends as indicated.

Lemma: If Q′ is a subpolygon of cutQ, w1 and w2 are the nearest points of Q′, and P is a shortest path in Q′ from w1 to w2, then P is doubly monotone.

Proof: Obvious from the fact that by the previous lemma, P may not have any 3-paths in a 'U' configuration.

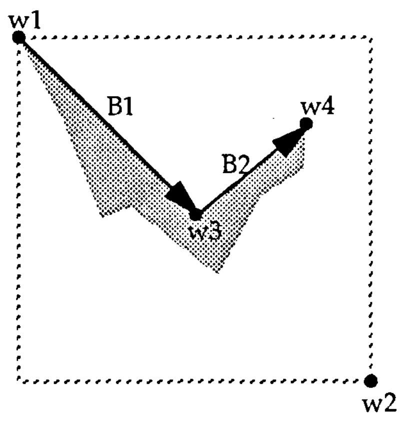
Funnel Lemma: Any subpolygon Q′ of cutQ is a funnel, and the nearest points of Q′ are the ends of the funnel.

Proof: Let w1 and w2 be the distinct nearest points of Q′. Assume they share the same y- (or x-) coordinate. Any shortest path P between w1 and w2 must be doubly monotone and hence a straight line. By previous lemmas, P is also a subpath of a shortest path P′ from S to T, and the only points of Q′ contained in P′ are those in P. However, a corollary of the bad point lemma states that every point of Q′ has some shortest path from S to T containing it. This is clearly impossible for any points in Q′ not on $\overline{w1w2}$, and so Q′ may not contain any points not on $\overline{w1w2}$. But for this to be the case, Q′ must be a single line, not a polygon, and so by contradiction, our original assumption that w1 and w2 shared the same y- or x-coordinate must be incorrect.

Now, let w1 and w2 be such that they share neither their y- nor x-coordinates. WLOG assume that w1 is above and to the left of w2. We wish to show that Q′ is a funnel; we do this by considering bdy(Q′) as being formed as the union of two directed paths, P1 and P2, which each go from w1 to w2 and do not otherwise intersect. Given that w1 and w2 are distinct, as is the case here, the boundary of any simple polygon can be represented in this manner, and so we do not miss any possible cases by this representation. To prove that Q′ is a funnel, we must first show that each of P1 and P2 is a doubly-monotone path. We do this somewhat informally, and with the aid of figure 16, as follows.

First, note that no part of Q′ can be outside of the isothetic rectangle defined by w1 and w2, for if there were a point w of Q′ outside of this rectangle, there would by previous lemmas have to be a shortest path from w1 to w2 going through this point, and thus outside the rectangle defined by w1 and w2; but because a shortest path from w1 to w2 is doubly-monotone and hence has length equal to just the rectilinear distance between w1 and w2,

figure 16



Grey region denotes the *interior* of Q′, dotted rectangle is the isothetic rectangle defined by w1 and w2. Any path in Q′ from w1 to w2 and containing w4 will be of greater length than the rectilinear distance from w1 to w2, and so will not be a shortest path from w1 to w2.

this is impossible.

Choose P1 to be the directed path from w1 to w2 contained in bdy(Q´) such that int(Q´) is to the right of P1 relative to a traversal of P1. Let B1 be the segment of P1 incident on w1. Considered as a segment of a directed path, it must point into the lower right-hand quadrant of the Cartesian plane centred at w1, because of the restriction that Q´ is confined to the rectangle defined by w1 and w2. Let w3 be the other end of B1 (the end that is not w1), let B2 be the next segment of P1 after B1 (B2 originates at w3), and assume that B2 points into the upper right-hand quadrant of the Cartesian plane centred at w3. Let the other endpoint of B2 (the endpoint that is not w3) be w4. Because Q´ is simple, any path from w1 to w4 will have to go down to or below w3 before going back up to reach w4, and so will have length greater than the rectilinear distance between w1 and w4. Clearly, then, any path from w1 to w2 containing w4 will be also have length greater than the rectilinear distance between w1 and w2. But then, the shortest path from S to T containing w4 will have a non-shortest subpath (the subpath from w1 to w2 containing w4), and so will not in fact be a shortest path from S to T. Thus, by contradiction, our assumption that B2 can point up and to the right must be incorrect.

In a similar vein, we can show that B2 may not point into the lower left-hand or upper left-hand quadrants of the Cartesian plane centred at w3, and so must point into the lower right-hand quadrant. We can then repeat this process for the next segment B3 in P1, and so by an informal induction continue until P1 terminates at w2, showing that each directed segment of P1 points into the lower right-hand quadrant of the Cartesian plane centered at its origin. P1 is therefore a doubly-monotone path.

The same procedure will work to show that P2 is a doubly monotone path; the fact that int(Q´) is on the left-hand side of P2, rather than the right-hand side, makes no difference.

We have now shown (albeit informally) that the boundary of Q´ may be represented as the union of two doubly-monotone paths, intersecting at w1 and w2 (the endpoints of the paths), and not intersecting at any other point. This means that Q′ is a doubly-monotone polygon, with ends w1 and w2. To show that Q′ is a funnel, we still need to show that there is a rectilinear boundary segment of Q′ adjacent to each of w1 and w2. However, this is both a necessary as well as sufficient condition for the existence of a rectilinear path in Q′ from w1 to w2, and since we know that such a path exists, we can see that this final condition also must be fulfilled. Thus, Q′ is a funnel.

The fact that w1 and w2, the nearest points of Q′, are also the ends of Q′ is obvious from the fact that w1 and w2 are the only points of intersection of P1 and P2, the doubly-monotone paths which make up bdy(Q′).

End of funnel lemma.

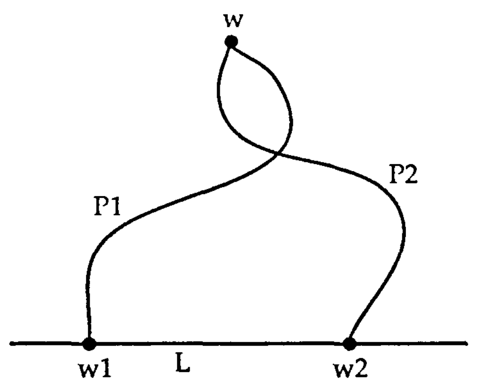
Definitions: [ends of a funnel nearest S and T]: Since the nearest points of a funnel are just its ends, we can speak of the end of a funnel nearest S, or the end of a funnel nearest T.

3.4.3. Isolated Line Segments in the Pseudogon

Various statements concerning subpolygons in cutQ have been presented and proved. Most of these statements have counterparts concerning isolated line segments in cutQ, and these counterparts have proofs very similar to those given for the statements made about subpolygons of cutQ. Going through these proofs in detail would be a tedious experience, both for the author and the reader. Instead, we will prove the isolated line segment version of the entry lemma in detail, and then make further statements about the properties of isolated line segments in cutQ with rather terse proofs.

Isolated Line Segment Entry Lemma: Let L be an isolated line segment in cutQ, and w

figure 17



The finite region bounded by L, P1, and P2 must b contained in cutQ, and so the points on L between w1 and w2 must be points on the boundary of a subpolygon of cutQ.

a point of cutQ not in L. There is a unique point w1 such that every path from w to a point in L contains w1.

Proof: Given L and w as above, let P1 be a path in cutQ from w to a point in L, and let w1 be the point of P1 nearest w which is also on L. If all paths in cutQ from w to a point on L contain w1 then we are done, otherwise let P2 be a path in cutQ from w to a point on L such that P2 does not contain w1, and let w2 be the point on P2 nearest w that is contained in L. The situation is as shown in figure 17. The closed curve (not necessarily a simple closed curve) formed by $P1\cup P2\cup\overline{w1w2}$ is completely contained in cutQ, and because ext(cutQ) is connected, the finite area G bounded by this curve must also be in cutQ. Because cutQ can be expressed as a finite union of subpolygons and line segments, G must be contained in some subpolygon(s) of cutQ; however, this would mean that those points of L between w1 and w2 are either in the interior of some subpolygon of cutQ, or are on the boundary of some subpolygon of cutQ (the case shown in figure 17), and in either case form a subsegment of L that cannot be an isolated line segment (by the definition of isolated line segment). Therefore, our original assumption that there is a path from w to a point in L not containing w1 must be false, and w1 satisfies the lemma.

As can be seen, the main difference between the above proof and that of the original entry lemma is that in the above, we show an assumption contradicting the lemma leads to the result that some subsegment of L is either in the interior of a subpolygon of cutQ or on the boundary of a subpolygon of cutQ, while in the proof of the original entry lemma, we showed that an assumption contrary to the lemma led to the conclusion that some boundary points of cutQ where in fact interior points of a subpolygon of cutQ. This difference is the major difference between proofs of properties of subpolygons of cutQ, and proofs of the similar properties of isolated line segments of cutQ.

Isolated Line Segment Required Point Lemma: If L is an isolated line segment of cutQ, then there are two distinct required points on L.

Proof: As for the Required Point Lemma, but substitute "L" for every instance of "Q′" or "bdy(Q′)", modify the diagram given with the Required Point Lemma (figure 13, page 38) so that a rectilinear line segment L is put in the place of Q′, and take as the contradiction the fact that $\overline{w1w2}$ must be inside of or part of a boundary segment for a subpolygon of cutQ.

Definitions: [points of an isolated line segment nearest S and T]: For any isolated line segment L in cutQ, let w1 and w2 be distinct required points on L (with one or both possibly being S and/or T) such that for any shortest path P from S to T, P will have a subpath P1 from S to w1 shorter than any other subpath of P from S to a point in L, and P will have a subpath P2 from w2 to T shorter than any other subpath of P from a point in L to T. If they exist, w1 and w2 are each unique in this regard; we call w1 the *point of Q′ nearest S*, and w2 is the *point of L nearest T*. Together, w1 and w2 are called the *nearest points* of L.

Lemma: Nearest points exist for any isolated line segment in cutQ.

Proof: Assume WLOG that there is no point of L nearest S. Because L possesses at least two distinct required points, there must then be shortest paths P1 and P2 from S to T such that the point w1 on P1 nearest S and contained in L is distinct from the point w2 on P2 nearest S and contained in L. By arguments identical to those used above, it is then easy to show that $\overline{w1w2}$ cannot be part of an isolated line segment because it either bounds or is contained in a subpolygon of cutQ.

Lemma: The nearest points of an isolated line segment L in cutQ are the endpoints of that segment.

Proof: Assume this is not the case, let w1 and w2 be the nearest points of L, and let e be an endpoint of L which is not a nearest point. It is easy to show that e must be a bad point, which (by contradiction) proves the lemma.

Note: Since the endpoints of an isolated line segment L are also its nearest points, we can speak of the end of L nearest S or the end of L nearest T, as was done with funnels.

Lemma: Let L be an isolated line segment in cutQ. Any shortest path from S to T in cutQ has L as a subpath.

Proof: Trivial, from the fact that the nearest points of L are its endpoints.

3.4.4. Specific Structure of the Pseudogon

Definition: [element of cutQ]: An *element* of cutQ is a subpolygon of cutQ or a maximal isolated line segment of cutQ.

Lemma: If E1 and E2 are two elements of cutQ, then E1 and E2 intersect at at most one point.

Proof: By cases.

1) If both E1 and E2 are isolated line segments, then if they are perpendicular they will intersect at at most one point, and if they are parallel, they will not intersect at all (for if they did, they would not be maximal isolated line segments, and so would not be elements of cutQ.)

2) If E1 is an isolated line segment and E2 is a subpolygon, assume E1 and E2 intersect at distinct points w1 and w2; in this case, we have the situation shown in figure 18, and since E1 and one subsection of bdy(E2) define a finite region containing the rest of bdy(E2), we have by arguments used previously that some parts of bdy(E2) must be in int(cutQ), which is a contradiction.

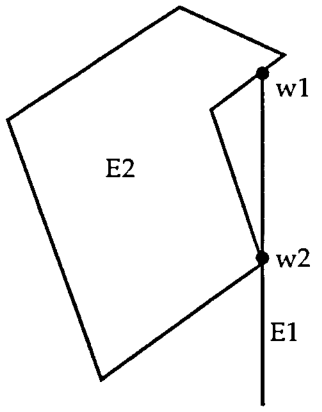
3) If E1 and E2 are both subpolygons of cutQ, a situation similar to that shown in case 2 arises, i.e. some boundary points of cutQ must be in int(cutQ), which is a contradiction.

So, by contradiction, we have shown that E1 and E2 may not intersect at more than one point.

Lemma: Let E1 and E2 be elements of cutQ which intersect at a point w. Then w contains the point nearest S in one of E1 or E2, and the point nearest T in the other.

Proof: Let E1 and E2 be elements of cutQ intersecting at the single point w. Let P be a shortest path from S to T containing w; by a corollary of the bad point lemma, P must exist. We proceed by cases:

figure 18



If E1 and E2 intersect at w1 and w2, the shorter section of bdy(E2) between w1 and w2 will be in int(cutQ).

1) w is not a nearest point of E1 or E2: By previous lemmas, P must contain a subpath P1 in E1, between the nearest points of E1 and containing w, and P must contain a subpath P2 in E2, between the nearest points of E2 and containing w. See figure 19, which gives an example where both E1 and E2 are subpolygons of cutQ. Since E1 and E2 do not intersect except at w, P1 and P2 will not intersect except at w; but since P1 and P2 travel between the nearest points of E1 and E2 respectively, and all of these nearest points are distinct from w, it is apparent that w cannot be an endpoint of either P1 or P2. Because P1 and P2 are subpaths of P, w must be a point where P self-intersects, and so P cannot be a shortest path. This contradiction leads us to conclude that our assumption that none of the nearest points of E1 and E2 are on w must be false.

2) If w is a nearest point of E1, but is not a nearest point of E2, the argument is much the same as for case 1; P, P1, and P2 are constructed in the same manner, and while w may be an endpoint of P1, it will still not be an endpoint of P2, so P still self-intersects at w, leading to a contradiction.

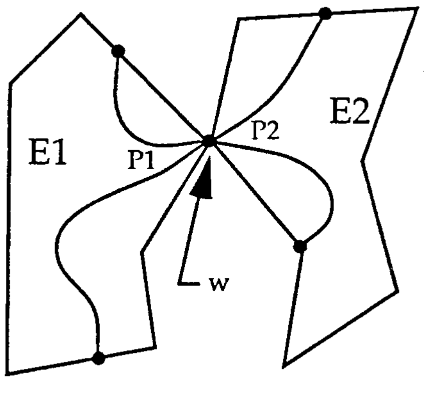
3) If the point in E1 nearest S is on w and the point in E2 nearest S is on w, then we again construct P, P1, and P2 as in case 1. As before, P1 and P2 will be disjoint, except for w. There are two possibilities:

   i) S=w, in which case P1∪P2 constitutes a subpath of P passing through S, but not actually containing S as an endpoint. This makes it apparent that P cannot be a shortest path from S to T.

   ii) S≠w, in which case there must be a subpath P3 of P from S to w; because w is the point of both E1 and E2 nearest S, w is the only point of E1 or E2 contained in P3. Because P1 is entirely contained in E1, P2 in E2, and since P3 does not contain any points of E1 or E2 outside of w, it follows that P1, P2, and P3 intersect only w, and thus P self-intersects at w in much the same manner as in case 2.

Of course, the same type of proof applies when dealing with the points of E1 and E2 nearest T.

figure 19



P1 between the nearest points of E1 and P2 between the nearest points of E2.

4) The only possibility left is that w contains the point nearest S of one of E1 and E2, and the point nearest T of the other. This is the possibility we wished to prove, and by elimination of other possibilities, we have done so. Note that w will contain only these nearest points of E1 and E2, as the other nearest points of E1 and E2 are necessarily distinct from w.

Pseudogon Lemma: Let cutQ be the pseudogon obtained from a simple polygon Q. cutQ can be expressed as a finite union of n elements $E_1, E_2, \ldots, E_n$ consisting of funnels and maximal isolated line segments, such that if $w1(E_i)$ and $w2(E_i)$ denote the ends of E_i nearest S and T respectively, then $w1(E_1)=S$, $w2(E_i)=w1(E_{i+1})$ for $1 \leq i < n$, $w2(E_n)=T$, and the various elements do not intersect otherwise.

Proof: Obvious from the preceding lemmas.

4. Finding the Dimensions of Smallest Paths Efficiently

In the previous chapter, we have shown that bad points, those through which no shortest and hence no smallest path from S to T may pass, may be removed from Q. In doing so, we have developed a powerful set of tools for simplifying Q, so as to make it easier to find a smallest path in Q from S to T. From the exclusion lemma, we know that no point in an unnecessary region of Q will be in a smallest path from S to T, and so we can remove all points in such regions from Q. From the inclusion lemma, we know that all unnecessary points in Q are included in an unnecessary region of Q pseudochord-induced by some pseudochord, and by finding and eliminating at most O(n) pseudochord-induced unnecessary regions of Q, where n is the number of vertices in Q, we will remove all unnecessary points in P. Once this simplification of Q has taken place, it will be much easier to find a smallest path in the remaining parts of Q. The major sections of this chapter address the efficient removal of unnecessary regions from Q, and finding the dimensions of a smallest path from S to T in the resulting pseudogon.

4.1. Representation and Size of Q, S, and T

A simple polygon may be given as an ordered list or vector of the coordinates of its vertices, or as a list of line segments making up the boundary of the polygon, in any case taking up O(n) space for an n vertex polygon under the usual assumption of a unit-cost RAM. Under the same assumption, S and T take up only an additional constant amount of storage. Thus, the obvious measure for an instance of the smallest-path problem is n, the number of vertices in the enclosing polygon.

4.2. Removing Unnecessary Points

Chapter 3 proved various lemmas concerning the topology of smallest paths in simple polygons, including the fact that all unnecessary points are contained in unnecessary regions pseudochord-induced by a set of O(n) pseudochords, where n is the number of vertices in Q, and the fact that removal of all unnecessary points from Q results in a simple structure known as the pseudogon. Using these results, we will now develop an algorithm SIMPLIFY which analyzes Q, S, and T, and in O(n log n) time produces the resulting pseudogon, cutQ.

4.2.1. The SIMPLIFY Algorithm—Overview

Describing SIMPLIFY in a standard algorithmic language would be tedious and unenlightening, as there are many minor details cluttering up an actual implementation. The following is a prose outline of SIMPLIFY, which glosses over many details. It is intended to outline to the reader the basic skeleton of SIMPLIFY, which will be fleshed out later, by describing the details of each step. Appendix II provides a series of illustrations, showing the important constructs generated by SIMPLIFY as it processes a polygon, on a step-by-step basis (with some steps omitted), and should be read in conjunction with the following explanation.

Begin SIMPLIFY {Basic outline of SIMPLIFY}

1) If the edges of Q are not already sorted according to their traversal order, do so.

2) Calculate traversal numbers for Q, based on S and T, and store these numbers (two for each segment) with the segments. traversal numbers are used to determine if given regions of Q are unnecessary (do not contain either of S or T). See Appendix I for details concerning the use of traversal numbers.

3) Find the horizontal and vertical extreme points in Q, and find boundary edges of Q directly above, below, to the left, and to the right of the extreme points and S and T.

Comment: Step 3 ends the preprocessing phase of SIMPLIFY, which constructs the data structures needed to provide basic information to the remaining steps of SIMPLIFY. The next phase is the "horizontal" processing phase, contained in steps 4-7.

4) Find all horizontal pseudochords in Q (including the horizontal pseudochords through S and T), and store these pseudochords in a data structure called HS.

5) For each pseudochord L in HS, find the region R which L pseudochord-induces on Q, and if R is an unnecessary region, indicate this by marking the endpoints of ebd(R), which are just the endpoints of L. If no unnecessary region is pseudochord-induced by L, then we have no further use for L, so it may be removed from HS.

Comment: Note that for any unnecessary region R, the endpoints of ebd(R) are not in R, so the marked endpoints of ebd(R) should not themselves be taken as part of R.

6) Traverse bdy(Q), propagating the information generated by step 5, in order to mark all segments or partial segments of bdy(Q) which comprise part of ebd(R) for some unnecessary region R found in step 5.

7) For each pseudochord L in HS, if L is contained in an unnecessary region pseudochord-induced by another member of HS (i.e. an unnecessary region identified by steps 5 and 6) then remove L from HS.

Comment: Let L1 be a pseudochord in HS which pseudochord-induces an unnecessary region R1, and let L2 be a pseudochord in HS which pseudochord-induces an unnecessary region R2 s.t. R2 is a proper subset of R1. The execution of step 7 will remove L2 from HS. When step 7 has been completed, there will be no unnecessary regions (as pseudochord-induced by pseudochords in HS) nested within other unnecessary regions. So, for each unnecessary region R pseudochord-induced by a remaining member L of HS, we may associate ebd(R) with L, as L is the unique pseudochord in HS which pseudochord-induces an unnecessary region R containing points of ebd(R). This association will be performed by marking each segment B or partial segment PB in bdy(Q) which forms a segment of ebd(R) as "belonging" to the pseudochord in HS which pseudochord-induces R. The marking is done by step 8.

8) Traverse the boundary of Q, marking each boundary segment or partial boundary segment of Q which forms a boundary segment of ebd(R), for some unnecessary region R pseudochord-induced by a horizontal pseudochord, as belonging to the pseudochord in HS which pseudochord-induces R

Comment: Step 8 ends the "horizontal" processing phase of SIMPLIFY. The next phase is the "vertical" phase, which consists of repeating steps 4-8, but using all vertical pseudochords through vertical extreme points, and storing these pseudochords in VS.

9-13) Repeat steps 4 through 8, with the modifications that everywhere the word "horizontal" appears, "vertical" should be substituted, and (for the sake of form) everywhere "HS" appears, "VS" should be substituted. References to other steps in the horizontal phase should be appropriately renumbered to refer to the similar steps in the vertical phase.

Comment: Step 13 ends the "vertical" processing phase of SIMPLIFY. At this point, we have identified the minimum set HS of horizontal pseudochords which will "remove" as much as possible of Q when only horizontal pseudochords are considered, and we have identified the minimum set VS of vertical pseudochords which will "remove" as much as possible of Q when only vertical pseudochords are considered. We have also marked the boundary of Q so as to identify those segments or parts of segments on bdy(Q) which form a boundary segment of ebd(R), for some unnecessary region R. By traversing bdy(Q) and removing the marked segments/parts of segments, we will be left with those sections of bdy(Q) which will actually be used in the pseudogon. What remains to be done is to identify and remove those horizontal pseudochords or parts of horizontal pseudochords which are in an unnecessary region pseudochord-induced by some vertical pseudochord, and to find and remove those vertical pseudochords or parts of vertical pseudochords contained in an unnecessary region pseudochord-induced by some horizontal pseudochord. This is done by "clipping" the pseudochords of each set against the pseudochord of the other set, as handled by steps 14-16.

14) For each pseudochord L in HS, check to see if L is entirely contained in an unnecessary region R pseudochord-induced by some vertical pseudochord. Any such R would have been identified during the vertical processing phase of simplify. If L is contained in such an unnecessary region, remove it from HS.

15) Repeat step 14, with the roles of vertical and horizontal pseudochords reversed.

16) For each pseudochord L in HS, remove those sections of L in unnecessary regions pseudochord-induced by a member of VS, and for each member L of VS, remove those sections of L in unnecessary regions pseudochord-induced by members of HS.

Comment: Having marked which sections of bdy(Q) are to be removed, and having clipped HS and VS against each other, all that remains to be done is to go through the rubble, so to speak, and out of it build the pseudogon cutQ.

17) Traverse bdy(Q), collecting those subsections of it which are *not* marked for removal. Use these subsections, together with the pseudochords remaining in HS and VS, to build cutQ as a series of funnels and isolated line segments from S to T.

End SIMPLIFY

4.2.2. The PROJECT Function

In describing SIMPLIFY in more detail in the next section, we will make use of an auxiliary function PROJECT. This section gives an operational definition of PROJECT, which should be read for a complete understanding of how SIMPLIFY works, and details on the implementation of PROJECT, which the reader may ignore if he or she so desires.

4.2.2.1. Operational Definition of PROJECT

PROJECT is a function which takes three arguments, the first being a point w, the second being one of the four rectilinear directions (up, down, left, right), and the third being a polygon (which w is assumed to be in), and projects w in the given direction, until w hits the boundary of Q and certain conditions are satisfied. PROJECT returns a two-tuple <w′, segmentOf(w′)> as result, where w′ is a point on bdy(Q), and segmentOf(w′) is the boundary segment on which w′ is located, or pair of boundary segments if w′ is a vertex. PROJECT is precisely defined as follows:

{ In the following, [d w] should be read as "to the left of w", "above w", etc., depending on the direction of d. }

PROJECT(d, w, Q) =

<w, segmentOf(w)> if there is no point w´ [d w] s.t. $\overline{ww'}$ is in Q

<w´, segmentOf(w´)> where w´ is the first point [d w] s.t. w´ is on bdy(Q) and w´ is contained in a boundary segment which does not contain w.

{ This last does not mean that w´ is not contained by any boundary segment containing w, but simply that w´ must at least be in at least one boundary segment not containing w. If w´ is a vertex of Q, it may be on both types of boundary segments. This case ensures that vertices which have a rectilinear boundary segment incident upon them do not just project into that segment, but that instead their projection reaches another boundary segment. }

See figure 20 for an example of the operation of PROJECT.

When using a tuple returned by PROJECT, we will not bother explicitly accessing the fields of the tuple, but will just refer to, "the point returned by PROJECT," or, "the segment returned by PROJECT".

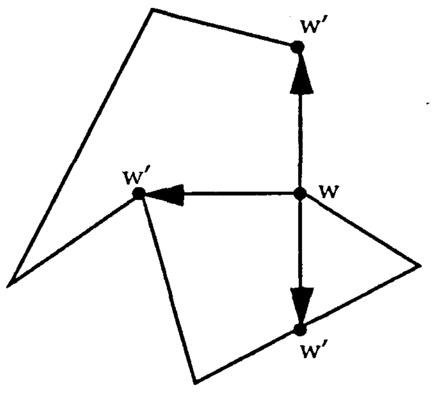
The above is an operational definition of PROJECT. As shown in the next section, this function can be constructed to run in constant time, with O(n log n) preprocessing.

4.2.2.2. Implementation of PROJECT

Given a simple polygon Q, the trapezoidal edges for a vertex v of Q are those boundary segments B directly above or below v such that there is a vertical line segment L within Q joining v and B. If we wish to project v up, and there is no vertical boundary segment incident on v and above it, then v's projection (if it exists) will clearly be on the upper trapezoidal edge of v. We may check for a vertical boundary segment incident on and above v in constant time, return the upper end of the segment as the projection of v, if such a segment exists, and otherwise find the upper trapezoidal edge of v, take the intersection of it with the vertical line through v in constant time, and return this as the projection of v upwards. Hence, the time to project v upwards is at most the time to find the upper trapezoidal edge of v. We can perform a similar operation if we wish to project v downwards, or in one of the horizontal directions.

A paper by Chazelle and Incerpi[13] shows how the trapezoidal edges for all n vertices in a simple polygon may be found by an algorithm taking O(n log n) time to execute. Since there is a bounded number of trapezoidal edges associated with each vertex, we can store these edges with their vertices so that any query on the trapezoidal edges of a vertex may thereafter be answered in constant time.

figure 20



Projecting w up, down, and to the left with PROJECT. Projecting w to the right just returns w.

We also need to be able to apply PROJECT to S and T, if they are not vertices of Q. For any point w in Q, it is an easy matter to find the edge of bdy(Q) directly above w; we sort the edges to get rid of those edges completely to the left of w, sort the remaining edges to get rid of those edges to the right of w, and sort again to get rid of those edges below w, and to find the edge directly above w. Since we need to apply this procedure only a constant number of times, and only in the preprocessing phase of the algorithm, preprocessing S and T so that PROJECT may later be applied to them in constant time may also be done in O(n log n) time.

Therefore, PROJECT may be made to run in constant time, assuming O(n log n) preprocessing.

4.2.3. <u>SIMPLIFY In More Detail</u>

Many of the steps in the outline of SIMPLIFY above are described in a purposely vague manner, so as not to obscure the structure of the algorithm. The following paragraphs fill in the details, to the degree necessary to show how each step of SIMPLIFY may be completed in O(n) or O(n log n) time.

Steps 1, 2, and 3—Sort edges into traversal order, compute traversal numbers, find extreme points and preprocess them for use with PROJECT.

The first three steps, the preprocessing phase, are the easiest. In step 1, if the edges of Q are not sorted in traversal order around the boundary of Q, this can be accomplished by considering each edge as a pair of points containing links back to their parent edges, and then sorting all of the points, primarily by x-coordinate and secondarily by y-coordinate. Identical endpoints of adjacent segments will be placed next to one another by this process, and by using the edge information supplied with the points, we can then start at some vertex v1 of Q, take an edge e1 incident to v, take the other endpoint v2 of e1, take the other edge e2 incident to v2, and continue in this manner, chaining around the boundary of Q until arriving back at v. Sorting for step 1 may be done in O(n log n) time, and constructing a traversal by chaining will take an additional O(n) time.

Once step 1 is complete, step 2 can be performed in O(n) time by traversing the boundary of Q. Finding horizontal extreme points from step 3 can also be done in O(n) time, and once these are found, the information needed to use PROJECT with these extreme points and S and T in future steps may be preprocessed in O(n log n) time, as detailed in the section on the implementation of PROJECT.

Step 4—Find all horizontal pseudochords

Step 4 is slightly more complex. The complications arise from the fact that a single horizontal line L in Q may contain many pseudochords, if L passes through many horizontal extreme points which all happen to share the same y-coordinate. The following makes use of the fact that no extreme point will be contained by more than a constant number of pseudochords, as was shown previously.

4.1) Sort the horizontal extreme points of Q, plus S and T, by y-coordinate. The rest of this algorithm is to be performed for each set HEy of these points with the same y-coordinate.

4.2) For each point w in HEy, find the points returned by PROJECT(w, left) and PROJECT(w, right), and add them to HEy.

4.3) Sort HEy by x-coordinate, and get rid of multiple copies of the same points.

4.4) Let e1 be the leftmost point in HEy, and let e2 be the rightmost point in HEy.

All horizontal extreme points in HEy will be between (and not on) e1 and e2, and so all horizontal pseudochords with the given y-coordinate will be on $\overline{e1e2}$. Sweep $\overline{e1e2}$ from e1 to e2 (left to right) by iterating down the sorted points in HEy. Let w be the point under inspection. Let an upper pseudochord be one which pseudochord-induces a region below itself, and a lower pseudochord be a pseudochord which pseudochord-induces a region above itself. We analyze by cases:

1) w is total lowest: then w signals the end (right endpoint) of a lower pseudochord, and the beginning (left endpoint) of a new lower pseudochord.
2) w is lowest, but not total lowest: then w signals the end of a lower pseudochord, if the horizontal segment of bdy(Q) incident on w is to the right of w, or the beginning of a lower pseudochord, if the horizontal segment of bdy(Q) incident on w is to the left of w.
3) w is total highest: as for (1), but for upper pseudochords.
4) w is highest, but not total highest: as for (2), but for upper pseudochords.
5) w is S or T and not on a non-horizontal segment of bdy(Q): then ignore w.
6) w is neither lowest nor highest (and may be S or T): then w is on bdy(Q), and depending on whether the points immediately to the left of w are contained in Q or not, w will signal either an end for both an upper and lower pseudochord, or a beginning for both an upper and lower pseudochord. (e1 and e2 are handled by this case, but points between them may also be handled by this case, if $\overline{e1e2}$ is not wholly contained in Q.)

During the sweep of $\overline{e1e2}$, we add to HS pseudochords found by the sweep as we encounter their right endpoints. Note that at no point do we have to keep track of more than two pseudochords, one upper and one lower.

Step 4.1 can be done in O(n log n) time. Applied over all the HEy, step 4.2 takes O(n) time, since PROJECT can be done in O(1) time and the total of all points in all HEy will be at most O(n). Applied over all the distinct HEy, step 4.3 will cumulatively take between O(n) and O(n log n) time, depending on the partitioning of the HEy. Finally, step 4.4 can be done over all the HEy in O(n) time, as HS is maintained in an unsorted order, and so we can add an element to HS in constant time. So, step 4 can be done in a total of O(n log n) time.

Step 5—Mark boundary of Q at points intersected by horizontal pseudochords

The main components of step 5 are the detection and marking of unnecessary regions. Detecting unnecessary regions can easily be done in constant time for each pseudochord to be examined, as described in appendix I, for a total time of at most O(n). When marking the boundary segments of unnecessary subpolygons (i.e. those parts of bdy(Q) which will later be removed), it is not sufficient to simply mark where bdy(Q) will be cut; we also need to mark which side of each cutpoint will be removed. We do this with arrows pointing in the direction of the boundary subpath to be removed. An example of this can be seen in figure 21. An arrow marker could simply consist of a point and a direction associated with the boundary segment on which the point is located.

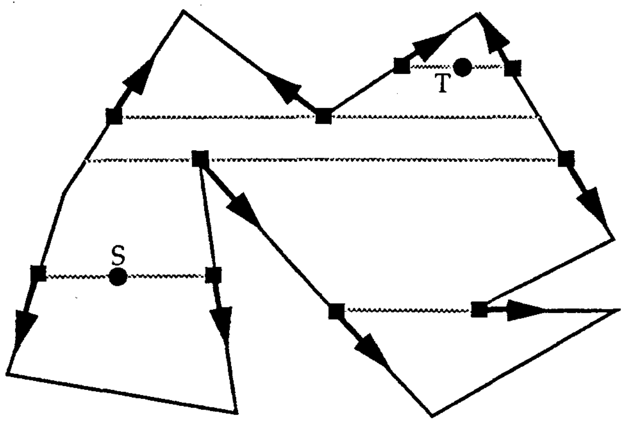
A potential problem is that in unusual cases, we could have up to O(n) arrow markers on a single boundary segment of Q. If not handled carefully, this could cause efficiency problems in later processing steps, which must use the marker information associated with each boundary segment. Luckily, this problem is easily dealt with, for if ever there are three distinct arrow markers A1, A2, and A3 on a single boundary segment B, where A1, A2, and A3 have been generated by horizontal pseudochords L1, L2, and L3 respectively, then at least one of L1, L2, or L3 is in an unnecessary region pseudochord-induced by one of the other two of L1, L2, and L3. WLOG say that L1 is in an unnecessary region pseudochord-induced by L2; then L1 and its associated arrow markers, including A1, may be removed, and in this manner we can ensure that there are never more than two arrow markers generated by horizontal pseudochords on any boundary segment. Note, however, that if L1 were vertical and L2 horizontal, then we cannot perform the same trick, as L1 and L2 may cross. Therefore, we will need up to two additional arrow markers associated with each boundary segment of Q, to mark the ends of vertical pseudochords occurring at each segment, and we will need to differentiate between markers induced by horizontal or vertical pseudochords.

Finally, we arbitrarily state that an arrow marker whose origin is at a vertex of Q should be stored with the boundary segment of Q into which it points. With this decision, we have concluded our detailed description of step 5.

Step 6—Propagate marking from step 5 to all boundary segments

While step 5 indicates subpaths of bdy(Q) to later be disposed of by marking the endpoints of sections of bdy(Q) which are to be excised, it does not mark every boundary segment in

figure 21



Using arrows to mark the endpoints of external boundaries of unnecessary regions. The marking shown in this diagram would be generated when processing HS.

those subpaths; instead, it only marks such segments as are touched by a horizontal pseudochord which pseudochord-induces an unnecessary region. In order to propagate this information to all affected segments, step 6 traverses bdy(Q), counting arrow markers as it goes, and marking boundary segments or (in the case of boundary segments divided into subsegments by an arrow marker) boundary subsegments according to its count at that point. The count begins at 0; every time an arrow marker pointing in the direction of the traversal is encountered, the count is incremented by 1, and every time an arrow marker pointing opposite to the direction of the traversal is encountered, the count is decremented by 1. At the end of the traversal, those segments or subsegments to be kept will be marked with the lowest count generated during the traversal; since step 6 cannot be sure of starting on a segment marked for keeping, it may turn out that the lowest count generated during the traversal was a negative number. In any case, once step 6 has completed the count traversal and found a lowest count, it does a second traversal, marking segments and subsegments for disposal if they were assigned a count greater than the lowest count on the first traversal. A constant time will be spent processing each boundary segment, for a total time of O(n) spent in step 6.

Note: The purpose of steps 5 and 6 is to mark those segments and subsegments of bdy(Q) which are in unnecessary regions pseudochord-induced by *horizontal* pseudochords. Likewise, the purpose of the analogues of steps 5 and 6 in the "vertical phase" of processing is to mark those segments and subsegments of bdy(Q) which are in unnecessary regions pseudochord-induced by *vertical* pseudochords. Keeping this horizontal and vertical information seperate is necessary for the correct functioning of other steps.

Step 7—Remove redundant horizontal pseudochords

Once step 6 is done, we check the endpoints e1 and e2 of each pseudochord L in HS. If e1 and e2 are on a portion of a boundary subsegment of Q marked for disposal during step 6, then L is entirely contained in an unnecessary subpolygon induced by another horizontal pseudochord, and L and its arrow markers may all be removed, as the information they provide is subsumed by other information. Note that because no two horizontal pseudochords will cross, we will never have a case where e1 indicates that L may be removed, but e2 indicates the opposite, so, for each L, we need actually check only one of its endpoints. There are at most O(n) pseudochords to check, and checking each may be accomplished in constant time, so the total time for step 7 is O(n).

Step 8—Mark boundary segments according to what pseudochords they belong to

Once step 7 is complete, we have removed all unnecessary horizontal pseudochords, relative to the set of horizontal pseudochords—there are no longer any horizontal pseudochords inside unnecessary regions pseudochord-induced by other horizontal pseudochords. We have also marked those sections of bdy(Q) contained in an unnecessary region pseudochord-induced by some horizontal pseudochord. As a result, we may indicate each marked segment/subsegment of bdy(Q) as having been marked for disposal by a *unique* pseudochord in HS—or to put this in a terser form, we further mark each marked (sub)segment as "belonging" to the element of HS which causes that (sub)segment to be marked. This may easily be done in O(n) time by numbering the remaining elements of HS, putting these numbers into the corresponding arrow markers, and then performing a traversal of bdy(Q) to propagate this information to all boundary segments.

Steps 9-13—Repeat steps 4-8 with a vertical orientation

Steps 9-13 are the analogues in the vertical processing phase of steps 4-8 in the horizontal processing phase. Except for the change in orientation, nothing is different. Just remember to keep the vertical information seperate from the horizontal information.

Steps 14-15—Remove redundant horizontal and vertical pseudochords

Up to this point, the vertical and horizontal pseudochords have been considered seperately from one another—this tradition ends with steps 14 and 15. These two steps check for horizontal pseudochords contained entirely within unnecessary subpolygons induced by vertical pseudochords, and vice-versa, and remove such wholly contained pseudochords. The check is done by considering the two ends e1 and e2 of each horizontal pseudochord L, and checking these ends against the vertical pseudochord information to see if e1 and e2 are both on boundary segments or subsegments which are marked for disposal by the same vertical pseudochord. If this is the case, then L is contained in an unnecessary region induced by a vertical pseudochord and may be removed. Vertical pseudochords are checked in the obvious similar manner. Note that both ends of a pseudochord must be checked for inclusion in the same unnecessary region, as vertical and horizontal pseudochords can cross, which case is handled in the next step.

Using the information generated in steps 8 and 13, steps 14 and 15 may be performed in O(n) time.

Step 16—Clip pseudochords against each other

We have now reduced HS and VS to the point where each member of HS and VS will play a part in the final pseudogon. However, the members of HS and VS may still cross into the unnecessary regions pseudochord-induced by each other. Step 16 remedies this. It is accomplished by checking each end of each member LV of VS. If an endpoint e1 of LV is on a part of bdy(Q) in an unnecessary region RH pseudochord-induced by some horizontal pseudochord LH, then LH and LV each cross into (but are not wholly contained in) the unnecessary region pseudochord-induced by the other. For any LV, there are at most two such LH's (one for each end of LV), and each such LH may be found in constant time by using the information generated in step 8. Once LH is known, we simply readjust the y-coordinate of e1 and the x-coordinate of e2 so that e1 and e2 are the same point, i.e. the point of intersection of LH and LV. For each vertical pseudochord LV, this process can be carried out for each end in constant time, and since there are at most O(n) vertical pseudochords, step 16 can be accomplished in O(n) time.

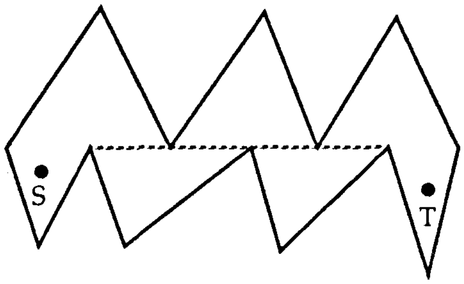
Step 17—Form cutQ from remaining pseudochords and boundary segments

Step 17 goes through the remaining boundary segments and pseudochords of HS, VS, and bdy(Q), and produces a representation of the pseudogon as a series of elements (isolated line segments and funnels) connected end-to-end, from S to T. This is done in a number of steps:

Comment: Some parallel pseudochords may overlap—this occurs, for instance, in a polygon such as shown in figure 22. Step 1 takes care of this problem.

1) First, sort the remaining elements of HS by y-coordinate. Then, for each set HSy of HS with identical y-coordinates, label the endpoints of each member of HSy as to whether they are left or right endpoints. Sort these endpoints into a vector V by x-coordinate, and iterate down V, merging overlapping members of HSy. These various sorting stages may be performed in cumulative O(n log n) time, and the

figure 22



The dotted line in the polygon above contains three overlapping horizontal pseudochords, each of which pseudochord-induces an unnecessary region.

merging of overlapping members of HSy may then be carried out in linear time on the size of HSy, for each HSy, resulting a total sequential time of O(n log n) for this step.

After processing HS in this manner, we merge overlapping elements of VS, using a similar process.

Comment: There will be no overlapping segments in the remaining sections of bdy(Q), so we do not need to merge overlapping segments of bdy(Q).

2) Identify each remaining segment in HS, VS, and bdy(Q) with a unique identification number, and label the endpoints of each segment with the same number, in O(n) time. For brevity, we will henceforth refer to the remaining segments of HS, VS, and bdy(Q) as just boundary segments, meaning segments of bdy(cutQ).

3) In merging overlapping members of HS and VS, we may have inadvertently merged some boundary segments of subpolygons of cutQ with isolated line segments of cutQ, or merged a boundary segment of one funnel with a boundary segment of another.

   To correct this, first sort all boundary endpoints by y coordinate, and then within each set Sy of segment endpoints with the same y-coordinate, sort the members of Sy by x-coordinate. By iterating down the members of Sy, we can identify each horizontal boundary segment HB of cutQ, and find the points inside HB at which other boundary segments of cutQ intersect HB; HB should be segmented at these points, and we do so. Doing this processing will take O(n log n) time, due to the sorting.

After having processed the horizontal segments of cutQ in this manner, re-sort the endpoints of the boundary segments and process in the obvious similar manner, so as to segment the vertical boundary segments of cutQ.

4) Move all endpoints of all boundary segments into a vector V. Sort these endpoints primarily by x-coordinate, then secondarily by y-coordinate. Congruent points will be placed next to one another in V. Total time for this step is O(n log n).

5) Initialize the variable CP (current point) to S.

6) Find the point or points in V congruent to CP, doable in O(n) time by a linear search. There are two possible cases:

6.1) If two points of V are congruent to CP, then two segments of bdy(cutQ) are incident on CP, and CP is one endpoint of a funnel. Segment information will permit us to find the other ends of the boundary segments incident on CP, and then by chaining from one endpoint of a boundary segment to the next (as was done in step 1 of SIMPLIFY), we can move down each doubly-monotone path making up the funnel, building the funnel in a seperate data structure as we go. The other end of the funnel will be reached when a point congruent to T is encountered, or a point with more than two boundary segments incident on it is encountered. (This last condition is testable in constant time, as if there are, say, three boundary segments incident on a vertex w, there will be three adjacent copies of w in V.) Once the other end of the funnel is reached, set CP to this other end, remove from V the information concerning this funnel, and if necessary (if T was not reached), repeat step 6 on the next element of cutQ, whose endpoint nearest S will now be CP.

6.2) If there is only one point of V congruent to CP, then CP is one end of an isolated line segment or series of line segments. We can chain down these line segments, inserting them into the copy of cutQ under construction as we go. We proceed in this manner until T is encountered or a vertex with more than one segment of bdy(cutQ) is encountered, at which point we modify CP to the new vertex, remove from V the information concerning the line segments we have chained down, and if necessary (if T has not been reached), repeat step 6.

Step 6 will spend a constant amount of time on each boundary segment of cutQ, and hence requires a total of O(log n) time.

7) After T has been reached, the elements of cutQ will have been extracted from HS, VS, and bdy(Q) in the order a smallest path from S to T would encounter them. Each of the above 6 steps took a maximum of O(n log n) time, and so step 17 of SIMPLIFY takes a total of O(n log n) time. This completes step 17 of SIMPLIFY.

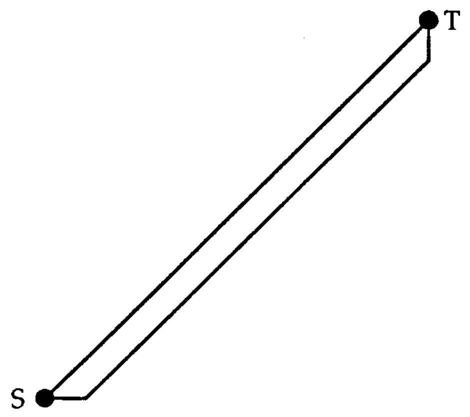
4.3. Analyzing the Pseudogon

At the end of step 17 of SIMPLIFY, we have reduced our original polygon Q to a pseudogon, cutQ. Since the pseudogon was constructed by removing regions of Q which by the exclusion lemma would not contain any part of a smallest path from S to T, it is apparent that by finding a smallest path in the pseudogon, we also find a smallest path in Q. However, there is a very significant potential efficiency problem with actually specifying a smallest path in the pseudogon, as discussed in the next subsection. Instead, we will concentrate on finding the *dimensions* of a smallest path from S to T in the pseudogon, a task which will be addressed by a later subsection.

4.3.1. Smallest Paths in the Pseudogon

Generating a smallest path through cutQ is a matter of generating smallest paths through the isolated line segments and funnels comprising cutQ, and linking these various smallest paths. Generating smallest paths through the isolated line segment portions of cutQ can easily be accomplished by a traversal of cutQ, using the information generated by step 17 of SIMPLIFY, and if we knew a smallest path through each funnel in cutQ, a further traversal of cutQ would serve to link all the smallest paths together. However, consider the polygon shown in figure 23, which is a funnel and its own pseudogon. Without specifying any particular strategy for finding a smallest path from S to T in this polygon, it is apparent that a large number of segments will be needed in *any* rectilinear path from S to T in the polygon. Even worse, polygons of the same shape, but with less distance between the diagonal boundary segments, will need many more segments in a rectilinear path from S to T. In fact, as we consider narrower and narrower polygons, the number of segments needed in a smallest path from S to T grows without being bounded in any way by the number of vertices in the enclosing polygon. Since generating a smallest path will always involve at least O(1) time for each segment in the path, it is obvious that we cannot obtain an "efficient" algorithm for generating a smallest path. Hence, in the next subsection we

figure 23



As the diagonal sides of the polygon move closer together, the number of bends in a smallest path from S to T will grow without bound.

turn our attention to finding the dimensions (length and straightness) of a smallest path from S to T.

Note, however, that although the number of segments in a smallest path through a polygon or funnel such as in figure 23 is not bounded by the number of vertices in the polygon (or funnel), this does not mean that the number of segments in a smallest path through such a polygon is unbounded on the input size of the problem. Polygons such as in figure 23 are "long" and "narrow", and so require large absolute differences in the x- and y- coordinates of their various vertices. Given this, it seems likely that in a logarithmic-cost RAM, a more realistic model than the unit-cost RAM for large problem instances, the number of segments in a smallest path from S to T would have a bound exponential on the input size of the problem. My thanks to Luis Goddyn of the Department of Mathematics for reminding me of this point.

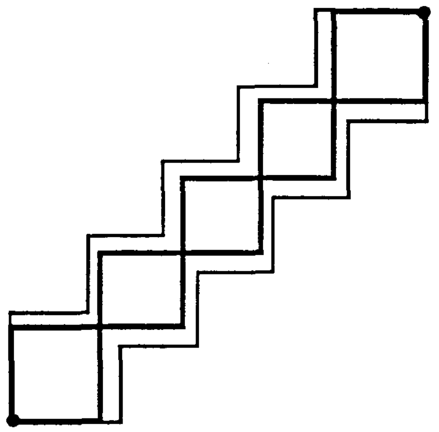
4.3.2. Dimensions of Smallest Paths in the Pseudogon

Calculating the dimensions of a smallest path from one end of a funnel to the other is a significant problem in its own right, and we defer discussion of it until chapter 5. For now we assume that the dimensions of a smallest path through a funnel (between its ends) can be calculated in O(m) time, where m is the number of vertices which define the funnel. This result will be shown in chapter 6.

Given a pseudogon represented as a chain of elements of cutQ from S to T, and with the above assumption, finding the dimensions of a smallest path from S to T in cutQ (and hence in Q) is reasonably straightforward. Basically, we calculate the dimensions of a smallest path through each element of cutQ (between the ends of each element), and decide at the intersection w of each adjacent pair of elements whether or not a smallest path from S to T will require a bend at w; if so, w is assigned a value of 1, otherwise it receives a value of 0. Then, by chaining down the elements of cutQ from S to T, we can cumulatively find the number of bends (or segments) a smallest path from S to T will require, and its length. There will be at most O(n) elements of cutQ, where n is the number of vertices in Q, so this process is executable in O(n)+f(n) time, where f(n) is the order function representing the time needed to calculate the dimensions of smallest paths through all the funnels of cutQ. As will be shown in chapter 6, dimensions of a smallest path through each funnel may be found in O(m) sequential time, where m is the number of vertices in the funnel, and since the total number of vertices in all funnels will not exceed n, f(n) will be just O(n).

The one point which makes this process slightly more involved than the above might suggest is the possibility of funnels such as shown in figure 24. Here we have two smallest paths through the funnel, indicated by the thick lines, such that each of the paths arrives at each end of the funnel going in a different direction than the other path. These paths really represent two classes of paths possible in any such funnel, each possible smallest path through such a funnel being put into one class or the other depending on whether its last segment is horizontal or vertical.

figure 24

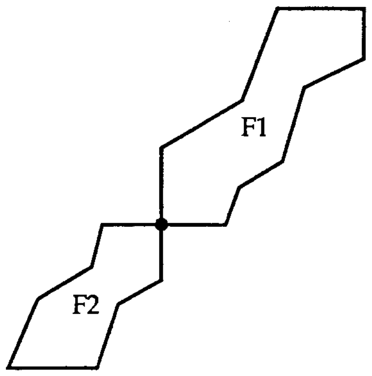


A funnel with two smallest paths between its ends, each path arriving at each endpoint in a direction different from the other path.

Funnels such as this could conceivably cause problems if they were chained together, as a "ripple effect" might be encountered, where we are unsure of exactly which class of smallest paths of such a funnel to consider in calculating the dimensions of a smallest path through cutQ. If there were O(n) of these funnels, with two potential classes of smallest paths through each, then we would have $O(2^n)$ possible combinations to consider, in order to determine the correct number of bends in a smallest path through cutQ. n could be as large as the number of vertices in Q.

Closer inspection shows that this problem does not arise. In order to have a ripple effect as described above, we need to have a series of funnels such that the intersection point of adjacent funnels has two rectilinear boundary segments of each funnel incident on it, as shown in figure 25. However, this configuration will never arise, as if it did, it would imply that either Q was not a simple polygon, or that some shortest path from S to T is not entirely contained in cutQ, which contradicts a previous lemma. The reader is invited to verify this in more detail. Any other configuration of an intersection point w between a funnel and a funnel or a funnel and a line segment forces a smallest path from S to T to

figure 25



Funnels in cutQ will never intersect in this manner.

pass through w with a specific orientation, and thus uncertainty as to which orientation should be assumed is impossible. Thus, no ripple effect may occur.

This does not mean that funnels such as shown in figure 24 will not occur in cutQ. It does mean that deciding whether we should use one class of path or the other through such a funnel F, when trying to minimize the number of bends that will occur in a smallest path from S to T at the intersection points of elements of cutQ, may be made as a local decision. Thus, handling funnels such as shown in figure 24 will cause the running time for the overall algorithm to be multiplied by at most two, because at most we need to do a local consideration in each funnel of which of two possible classes of smallest paths to use in that funnel.

5. Finding the Dimensions of Smallest Paths in Parallel

A CREW PRAM is a theoretical model of a parallel processing computer. The acronym stands for Concurrent Read Exclusive Write Parallel Random Access Machine. A CREW PRAM assumes some number of processors n, all operating simultaneously. At any time, any processor can read any memory location that is not being written to by another processor, and can write any memory location that is not being written to by another processor. The dimensions of a smallest path from S to T in a polygon Q may be found in O($\log^2$ n) time on a n/log n-processor CREW PRAM, which is an optimal speedup of the sequential algorithm, or in O(log n log log n) time on a n-processor CREW PRAM, which is a non-optimal speedup, but a better time bound. The following material describes how this may be accomplished.

To solve the overall problem in parallel, we first describe how the simplification of the polygon into a pseudogon may be solved in O(log n) parallel time, using O(n) processors. Then, assuming the problem of finding the dimensions of smallest paths through funnels has been solved (which solution will be given in chapter 6), we show how the dimensions of a smallest path through the resulting pseudogon may be obtained in O(log n) parallel time using O(n) processors . Chapter 6 will show that the dimensions of smallest paths through the various funnels may be found in O($\log^2$ n) time, using O(n/log n) processors, or in O(log n log log n) time using O(n) processors. Since we will be able to simplify the polygon and (ignoring the time to solve the funnel problem) find the dimensions of a smallest path through the pseudogon in O(log n) time using O(n) processors, it will immediately follow that the entire problem may be solved in O($\log^2$ n) time on a n/log n-processor CREW PRAM, or in O(log n log log n) time on a n-processor CREW PRAM.

5.1. A Parallel Algorithm for Generating cutQ

We assume the presence of n processors, one assigned to each vertex of Q. The following are how the steps of SIMPLIFY (previously described) may be done in O(log n) parallel time.

Step 1—Sort Polygon Edges into Traversal Order

As for the sequential version, we sort the points defining line segments primarily by x-coordinate and secondarily by y-coordinate, using Cole's sorting algorithm[14], which provides a low-overhead O(log n), n processors general-use sort. We then use recursive doubling [15] to fuse portions of the boundary into successively larger boundary portions in traversal order, starting with adjacent boundary segments. This will permit us to build a traversal order data structure of the boundary segments in O(log n) phases, each phase taking O(1) time.

Step 2—Calculate traversal Numbers

Assume we are calculating the traversal numbers relative to S. At each vertex of Q, we calculate in constant time the "traversal number increment"—the amount the traversal number would change when going through that vertex in a sequential traversal. Then, using parallel prefix, the traversal numbers over all the boundary segments may be calculated in O(log n) time using n/log n processors. See appendix III for a description of the parallel prefix operation.

Step 3—Find Horizontal and Vertical Extreme Points, Compute Trapezoidal Edges

Each processor may decide in constant time if the vertex it is assigned to is a horizontal or vertical extreme point. An O(log n log log n)time algorithm using n processors for trapezoidal edge finding is given in [16], and this timing may be improved to O(log n) time using n processors through the application of fractional cascading[17].

Step 4—Find Horizontal Pseudochords

We take the extreme points as sorted on their y-coordinate, possible in O(log n), n processors parallel time. This sorted set may be partitioned so that all vertices in the same partition have the same y-coordinate; partitioning in this manner may be accomplished in O(log n) time, through use of the parallel prefix algorithm. Next, for each vertex v, find the points returned by PROJECT(v, left) and PROJECT(v, right), and add them to v's partition. This may be accomplished by storing all vertices v into a vector which leaves an empty element to each side of each v, and then storing the results of the PROJECT functions in the empty elements. Each partition is then sorted by x-coordinate and duplicates are removed, taking a maximum of O(log n) parallel time for all partitions. Then, with a constant number of vertices assigned to each processor, each point in a partition may be tested in constant time as to whether it is just a point on a boundary, or a highest or lowest, total or non-total extreme point. Depending upon the results of this test, we can mark each point as being the start (left endpoint) or end (right endpoint), or both start and end of some upper or lower pseudochord, or in the case of S and T, as being uninvolved. See step 4.4 in the sequential description of this step of the algorithm for details on how each point should be marked. Once this marking is accomplished, it may be distributed via two applications of parallel prefix (one for upper pseudochords and one for lower pseudochords) using a maximum of O(log n) time, and the pseudochords themselves may then easily be extracted. See appendix III for an example of how parallel prefix may be used to distribute information through a vector in this manner.

Step 5—Mark points at which horizontal pseudochords intersect boundary segments

Step 5 considers each pseudochord, and marks where that pseudochord intersects the boundary of Q if the pseudochord induces an unnecessary subpolygon on Q. Testing for unnecessary subpolygons is easily done in constant parallel time by making use of traversal numbers, but care must be taken with cases where we have a large number of markers to put on one boundary segment. Assume we are considering the right ends of horizontal pseudochords, and wish to perform a mark of the boundary segments according to these right ends. We may order the horizontal pseudochords by y-coordinate, number them according to their order, and associate with each pseudochord the boundary segment the right end of the pseudochord will mark, using some arbitrary numbering of the boundary segments to identify them. By sorting this vector on the field containing the boundary segment identification numbers and partitioning it according to these numbers, we can then find out exactly what and how many horizontal pseudochords have their right end on each particular boundary segment. From this point, processing is easy—we could, for instance, associate with each boundary segment a vector large enough to contain the right ends of all pseudochords incident on that boundary segment, use the vector to contain the pseudochord marking information, and when marking is complete, perform a parallel prefix on the vector to distribute the marking information across the edge.

Using Cole's sorting algorithm and the parallel prefix algorithm, we may do all of the above in O(log n) time using n processors.

Step 6—Traverse bdy(Q), propagating information from step 5

By putting the salient information from step 5 into a traversal-ordered vector of edges and then doing a parallel prefix on that vector, this step may be accomplished in O(log n), n/log n processors time.

Step 7—Remove redundant horizontal pseudochords

O(n) horizontal pseudochords may be checked in constant time to see if they are redundant, and actual removal may be done in a number of ways, in a maximum of O(log n) parallel time, assuming we wish a compacted (no holes) data structure after the removal.

Step 8—Mark boundary segments or parts thereof according to what pseudochords the segments belong to

May be done with parallel prefix in O(log n) parallel time.

Steps 9-13—Vertical analogues of steps 4-8

As described.

Steps 14 and 15—Remove redundant horizontal and vertical pseudochords

This involves only a local check on each remaining pseudochord, so is doable in constant time, plus an additional O(log n) time for data structure compaction, if desired.

Step 16—Clip horizontal and vertical pseudochords against one another

May be done in parallel for each pseudochord, in constant time.

Step 17—From the remnants, build cutQ in traversal order

As in the sequential implementation of step 17, the parallel version of step 17 of SIMPLIFY begins by first merging overlapping vertical and horizontal boundary segments, then segmenting the merged segments to the extent necessary to ensure that merged segments do not go through vertices of bdy(cutQ). The iterations over vectors performed in the sequential version may be done in the parallel version using parallel prefix, and the other processing involved in these steps is parallelizable in an obvious manner.

As in step 1 of the parallel version of SIMPLIFY, we use recursive doubling to build up pointer links spanning subpaths of bdy(cutQ) which contain only vertices with exactly two segments of bdy(cutQ) incident on them. To do this, we must first sort the vertices of bdy(cutQ) so that congruent points end up adjacent to one another, as was done in step 17 of the sequential version of the algorithm. Using Cole's algorithm, this may be accomplished in O(log n) time using n processors. This step makes the recursive doubling possible because we now have a constant-time test to determine if more than two segments of bdy(cutQ) are incident on one vertex, and so we know when *not* to continue fusing subpaths of bdy(cutQ) during the recursive doubling. Given n vertices in cutQ, the recursive doubling will take O(log n) steps using n processors, each step executing in constant time.

Once we have determined the endpoints of subpaths of bdy(cutQ) containing only vertices with exactly two segments of bdy(cutQ) incident upon them, it is an easy matter to identify which subpaths constitute part of a funnel boundary (just those subpaths which share their endpoints with some other subpath), and which subpaths consist of chains of isolated line segments. By assigning a unique id number to each subpath identified by the recursive doubling, and then running the recursive doubling process in reverse, we can distribute each subpath id to each boundary segment in the given subpath. Sorting on the subpath id then gives a set of vertices (some of which are duplicated, because they belong to more than one funnel/isolated line segment) partitioned according to which funnel or chain of isolated line segments the vertices belong to. From this point, it is a simple matter to build each funnel or chain of isolated line segments, and then by sorting on the endpoints of the funnels/line segment chains and performing one more recursive doubling to link everything together, we can build cutQ as a series of pseudogon elements from S to T in O(log n) time, using n processors.

Since each of the above steps is doable in O(log n) time using an n-processor CREW

PRAM, we conclude that the parallel version of SIMPLIFY may be accomplished in O(log n) time using n processors.

5.2. Finding the Dimensions of Smallest Paths in Parallel

In the chapter concerning smallest paths through funnels, it will be shown that the dimensions of a smallest path through any funnel can be obtained in $O(\log^2 n)$ time using n/log n processors, or in O(log n log log n) time using n processors, where n is the number of vertices in the funnel. Given that step 17 of SIMPLIFY has produced a representation of cutQ as a series of line segments and funnels from S to T, chapter 6 shows how to find, in parallel, the dimensions of smallest paths through each of these line segments and funnels, find the appropriate bend increment to use at each intersection of two elements of cutQ, and then use parallel prefix to determine the dimensions of a smallest path from S to T. Time bounds for these operations will be a cumulative maximum of $O(\log^2 n)$ using n/log n processors, or O(log n log log n) using n processors (the limit on efficiency being imposed by the parallel processing of funnels, as described in chapter 6).

Except for the use of parallel prefix, finding the dimensions of a smallest path through cutQ in parallel is almost the same as doing the same thing sequentially, so refer to the section on finding smallest path dimensions sequentially (section 4.3.2) for more details.

6. Funnels

Smallest paths through funnels may require a number of segments not bounded by any function on the number of vertices in the funnel. This provides a strong incentive to look for procedures which permit efficient determination of the dimensions of smallest paths through funnels, without necessarily constructing such paths. Procedures of such a nature are the subject of this chapter of the thesis.

The following exposition will be considerably simplified if we make certain assumptions about the nature of the funnels we are dealing with. The following section of this chapter gives various definitions, and then describes these assumptions. Further sections in the chapter will then show how to efficiently determine the dimensions of smallest paths through such funnels sequentially and in parallel, and a closing section will discuss what need be done to generalize these results to all funnels.

6.1. Definitions, Assumptions, and Basic Results Concerning Funnels

Recall from Chapter 3 that a funnel is a doubly-monotone polygon with the added property that each end of a funnel has a rectilinear boundary segment incident on it. The following definitions apply.

Definitions: [degenerate, simple, and diagonal funnels, PDFs]: A *degenerate* funnel is one which has only three boundary segments. A *simple* funnel is one which has four boundary segments. A *diagonal* funnel is a non-degenerate funnel which has only diagonal boundary segments, excepting one rectilinear segment adjacent to each end of the funnel. A *parallel diagonal funnel*, abbreviated PDF, is a diagonal funnel F such that the two rectilinear boundary segments of F (one adjacent to each end of F) are parallel.

Assumption: From this point until the end of this chapter (where there is a subsection on dealing with funnels which are not parallel diagonal funnels), we will assume that all funnels are parallel diagonal funnels, unless explicitly noted otherwise.

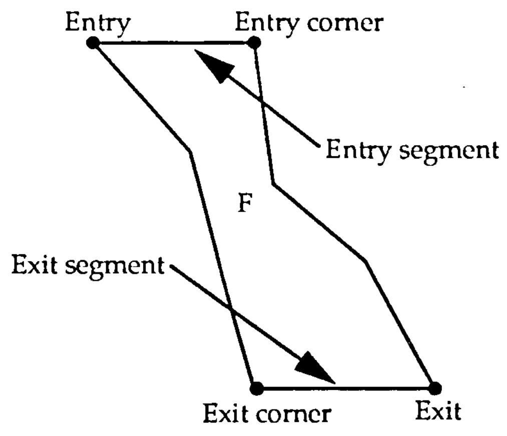
Definitions: [entry, exit, end-segment, funnel corner]: Let F be a funnel. We will be considering only paths through F which connect the two ends. For convenience, we will treat any path P within F and between its two ends as a directed path, and say that the end of F which is the origin of P is the *entry* to F, and the end of F which is the terminus of P is the *exit* of F. The two rectilinear boundary segments adjacent to the ends of F (one at each end) are called the *end-segments*; the end-segment adjacent to the funnel entrance is called the *entry segment* and the one adjacent to the funnel exit is called the *exit segment.* The endpoints of the end-segments which are not the ends of F are called the *corners* of F; the corner of F which is an endpoint of the entry segment is the *entry corner* and the corner of F which is an endpoint of the exit segment if the *exit corner.* See figure 26 for examples.

Definition: [standard position]: A funnel is in *standard position* if it is oriented so that its entry segment is horizontal, and its exit point is below and to the right of the entry point. The diagonal boundary segments of a funnel in standard form will all slope down and to the right. The funnel of figure 26 is in standard position.

Any funnel can be put into standard position by use of reflections and rotations, and doing so permits us to talk about it without the problem of handling different orientations. A funnel put into standard position can be returned to its original position by inverting the original sequence of reflections and rotations, and any paths constructed in the funnel can be transformed along with it, so considering funnels only in standard position involves no loss of generality.

Definitions: [projection of a point through a funnel, projection path, standard projections]:

figure 26



Components of a parallel diagonal funnel.

Let F be a PDF in standard position, and let w be a point on the entry segment of F, but not the entry point itself. The *projection* of w through F, denoted $p_F(w)$ is the point w′ on the exit segment of F obtained by the following process:

Project w down until it intersects a point w1 on bdy(Q).
Project w1 to the right until it intersects a point w2 on bdy(Q).
Project w2 down until it intersects a point w3 on bdy(Q).
Project w3 to the right . . .

.

.

.

Project wn-1 (for some n) to the right until it hits a point wn on bdy(Q).
Project wn down until it hits a point w′ on the exit segment.

Of course, we may not even need as many projections as explicitly given above, eg. we might find w′ immediately upon projecting w downward.

The *projection path* generated by projecting w through F, denoted $pp_F(w)$ is the path whose segments are $\overline{ww1}$, $\overline{w1w2}$, $\overline{w2w3}$, . . ., $\overline{wn\text{-}1wn}$, $\overline{wnw'}$.

By projecting points up and to the left, we can in a similar manner define the *inverse projection* of a point w′ on the end segment onto a point w on the entry segment, denoted $p'_F(w')$, and the accompanying *inverse projection path,* denoted $pp'_F(w')$. p_F and p'_F are inverse functions, and so $p'_F(p_F(w))=w$, and $p_F(p'_F(w'))=w'$. The *standard projection* and *standard projection path* are obtained by projecting the entry corner through F, and the *inverse standard projection* and *inverse standard projection path* are obtained by projecting the exit corner through F.

When there is no danger of confusion as to which funnel we wish to project a point through, we will abbreviate $p_F(w)$ to just p(w), and $pp_F(w)$ to just pp(w). As well, $p_F(w)$ and $p'_F(w')$ are distinguishable as a projection and an inverse projection simply by virtue of the fact that w is on the entry segment of F while w′ is on the exit segment of F. Thus, as long as a point on which the projection is to be taken is specified, we do not need to explicitly differentiate between projection and inverse projection functions, and so for convenience we will generally write $p'_F(w')$ as just $p_F(w')$, and similarly for $pp'_F(w')$.

Projection Path Lemma: For any parallel diagonal funnel F, the standard projection path is a straightest path from the entry segment to the exit segment.

Proof: [by induction on the number of segments in the standard projection path]:

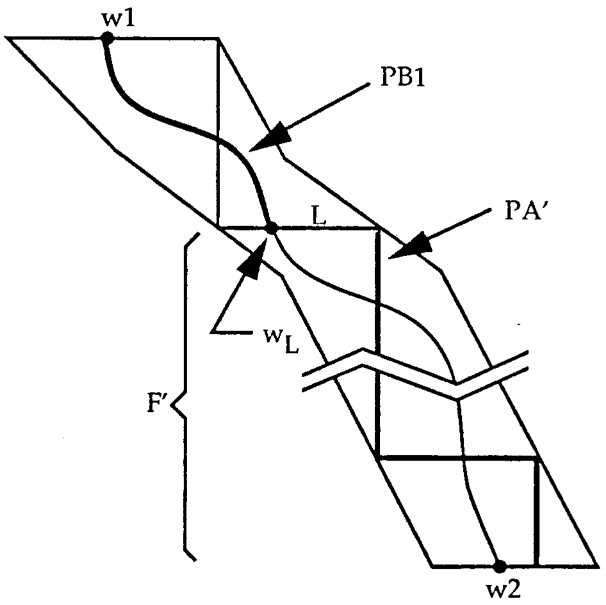
First, note that because the first and last segments of any path P between the ends of a parallel diagonal funnel will be parallel, the number of segments in P must be odd.

Basis: For any PDF F, the entry and exit segments will be non-intersecting, so if the standard projection path through F has just one segment, the lemma is obvious.

Inductive Assumption: For some integer k, for any PDF F, assume that if the standard projection path P of F has k or fewer segments, then P is a straightest path from the entry segment of F to the exit segment of F.

Inductive Step: Let F be some PDF such that its standard projection path PA has k+2 segments. We can form a parallel diagonal subfunnel F′ of F by taking that portion of F below and including the first horizontal segment L in PA. See figure 27. The subpath PA′ of PA in F′ from the entry corner of F′ to the exit segment of F′ will then form a k-segment standard projection path for F′, and so the inductive assumption applies. L forms a chord for F, inducing two regions in F with one region containing the entry segment of F and one region containing the exit segment of F, so any path from a point on the entry segment of F to a point on

figure 27



A highly schematic diagram. In particular, note that PB is shown as a smooth curve, when it is of course a rectilinear path. For clarity, many of the components of the diagram are not named.

the exit segment of F must cross L. Let PB be a path from a point w1 on the entry segment of F to a point w2 on the exit segment of F, let w_L be the first point of PB which is on L, and let PB1 be the subpath of PB from w1 to w_L. There are two cases, depending on the configuration of PB1.

1) If PB1 is anything other than the first segment in PA, then let PB2 be the subpath of PB from w_L to w2. PB1 must have at least three segments (a vertical segment leaving the entry segment of F, a horizontal segment, and a vertical segment arriving at w_L), and by the inductive assumption, PB2 must have at least k segments; even if the last segment of PB1 and the first segment of PB2 form a single segment in the path PB, this still means that PB must have at least k+2 segments.

2) If PB1 is just the first segment in PA, then because the first segment of PA can by definition go no farther down, the second segment of PB will have to be a horizontal segment contained in the second segment of PA, linking the first segment of PB with the subpath PB3 made up of the third to the last segments of PB. This means that the inductive assumption applies to PB3, and so this subpath has at least k segments. Since the horizontal second segment of PB cannot merge with the vertical first segment of PB or the vertical first segment of PB3—the third segment of PB—this means that PB is composed of at least k segments in PB3, plus its own first two segments, for a total of at least k+2 segments.

Therefore, we have by induction that the standard projection path through any diagonal funnel F is a straightest path between the end-segments of F. This completes the proof of the projection path lemma.

Corollary: The inverse standard projection path through a funnel F is also a straightest path between the end-segments of F.

Definition: [standard end-to-end path and inverse standard end-to-end path through a funnel]: Let F be a funnel with entry e1, exit e2, entry corner c1 and exit corner c2, and let c1′ and c2′ be the projection and inverse projection respectively. The *standard end-to-end path* through F is the path from e1 to e2 formed by the concatenation of $\overline{e1c1}$, the standard projection path, and $\overline{c1'e2}$. The *inverse standard end-to-end path* is the path from e1 to e2 formed by the concatenation of $\overline{e1c2'}$, the inverse standard projection path through F, and $\overline{c2e2}$.

End-to-End Lemma: Let F be a funnel in standard position, with entry e1 and exit e2, and let P be the standard end-to-end path through F. P is a smallest path from e1 to e2.

Proof: P will be a shortest path because it is monotone in both rectilinear directions. Also, if P′ is the subpath formed by removing the first and last segments of P, we know by the projection path lemma that P′ is a straightest path from the entry segment of F to the exit segment of F. Because F is a diagonal funnel, any path between its endpoints must have a horizontal first and last segment, and a subpath between the entry and exit segments, which for P is P′. Since P′ is a straightest path between the two end-segments, and since the first and last horizontal segments are unavoidable, we conclude that P is a straightest path between the two ends of F. Since P is both shortest and straightest, it is therefore smallest.

Corollary: The inverse standard end-to-end path through a funnel F is also a smallest path between the endpoints of F.

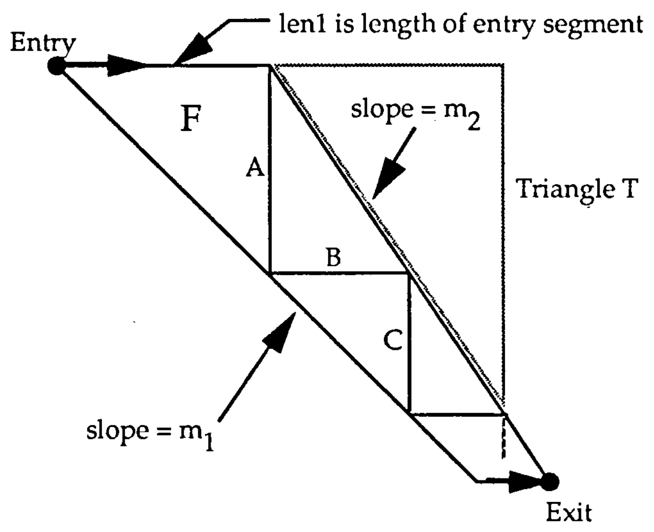
Note: The end-to-end lemma means that to find the number of segments in a smallest path through a funnel, we need simply find the number of segments in a standard projection path through that funnel, and add two. The problem of finding the length of a smallest path through a funnel has a trivial solution—this length will just be the rectilinear distance between the funnel ends. Therefore, succeeding sections concentrate on how to find the number of bends in a standard projection path through an arbitrary funnel.

6.2. Projection Paths through Simple Funnels

This section will show that the projection of any point through a simple PDF, and the number of segments in its projection path, can both be found in constant time.

Consider a simple PDF in standard form; in particular, consider the funnel F shown in figure 28. In this case, the exit segment is shorter than the entry segment, causing the funnel to narrow as it moves downwards. If the standard projection path through F has just one, two, or three segments, we can calculate it in constant time. Otherwise, let A, B, and C be the first three segments in the standard projection path P through F.

figure 28



A simple funnel—A, B, and C are the first three segments inthe standard projection path through F.

If the two diagonal boundary segments of f have slopes m_1 and m_2 as shown, and the lengths of A, B, and C are a, b, and c respectively, we can certainly say that $|m_1| = c/b$, and $|m_2| = a/b$, and hence $|m_1/m_2| = (c/b)/(a/b) = c/a$. Because of the way P is defined, this calculation will produce the same ratio when applied to any pair of successive vertical segments excepting the last vertical segment, and so we can see that any vertical segment in P, except the first and last vertical segments, will have length (c/a)*L, where L is the length of the preceding vertical segment in P. The length of the first vertical segment in P will be just a, and the length of the last vertical segment in P is not necessarily related to the lengths of the preceding segments in any way, because this last segment may (and usually will) reach the exit segment before it encounters the lower diagonal boundary of P. If we say that P has k+1 vertical segments, then it is easy to see that the lengths of the first k vertical segments may be obtained through the use of the ration c/a, while the length of the last vertical segment of P cannot necessarily be obtained in this manner. The total vertical distance d traveled by P must be

$$\textit{(1)} \quad d = \sum_{i=0}^{k-1} a(c/a)^i + u = a \cdot \frac{1-(c/a)^k}{1-(c/a)} + u; \quad c < a$$

where u is the length of the last vertical segment in P, which cannot be included in the summation because its length cannot be predicted from the length of the previous vertical segment in P. Using the closed form of the expression, we can solve for k and obtain

$$\textit{(2)} \quad k = \log_{(c/a)}[(1 - (d - u)(1 - c/a)/a]$$

where d will of course be just the vertical dimension of the funnel. Making use of the fact that $u \leq a(c/a)^k$, and hence

$$\textit{(3)} \quad \sum_{i=0}^{k-1} a(c/a)^i + u \leq \sum_{i=0}^{k} a(c/a)^i$$

we can then set u in (2) to 0, solve for k, and take $\lceil k \rceil$ as the number of vertical segments in P. Given $\lceil k \rceil$, the number of vertical segments in P, the number of horizontal segments in P will be $\lceil k \rceil - 1$, and so the total number of segments in P is $2\lceil k \rceil - 1$.

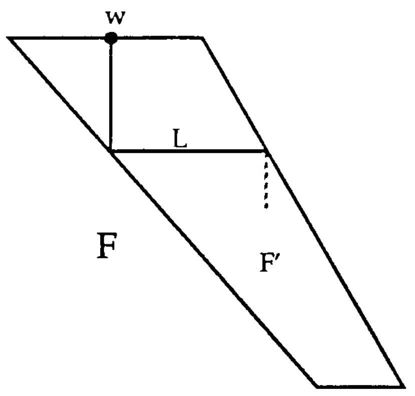
From the preceding, it is apparent that the following pieces of information may be calculated in constant time:

i) $c/a = |m_1/m_2|$.
ii) $a = len1 * |m_1|$, where len1 is the length of the entry segment.
iii) k, from equation *(2)*.
iv) number of segments in P = $2\lceil k \rceil - 1$
v) vertical dimension of P = vertical dimension of F.
vi) horizontal dimension of P = $a*[1-(c/a)^{\lceil k-1 \rceil}]/[1-(c/a)]/|m_2|$. This is derived from the fact that $a*[1-(c/a)^{\lceil k-1 \rceil}]/[1-(c/a)]$ is the vertical distance traveled by P less the length of the last (vertical) segment of P; that dividing this result by $|m_2|$ gives the horizontal distance traveled by P can be seen be considering the width of triangle T in figure 28.

The above applies to simple PDFs which narrow as they move from the entry to the exit. For parallel simple diagonal funnels which widen as they move from the entry to the exit, we will need to use a different closed form in place of equation *(2)*; alternatively, we could do a conceptual switch of the entry and exit. Parallel simple diagonal funnels whose diagonal sides are themselves parallel have a particularly simple closed form (just $d = ka + u$) which must be used in place of *(2)*.

The above result may be used to find the projection or dimensions of a projection path for any point on the entry segment of F, and not just the entry corner. Let w be some point on the entry segment of a simple PDF F, as shown in figure 29. The first and if necessary the

figure 29



The second segment of $pp_F(w)$ induces the subfunnel F′ on F, and the third to the last segments of $pp_F(w)$ form a standard projection path for F.

second segments of $pp_F(w)$ can be calculated in constant time; if it exists, the second segment L of $pp_F(w)$ induces a subfunnel F′ of F, below L. The third to the last segments of $pp_F(w)$ will form a standard projection path for F′, and so it is easy to analyze this section of $pp_F(w)$ and then use the results of such an analysis to find the $p_F(w)$ and the straightness of $pp_F(w)$.

From the above, it is obvious that the projection of any point w on the entry segment of a simple funnel F through F may be found in constant time, as may the number of segments in the projection path of w.

6.3. Sequential Calculation of Smallest Paths through Non-Simple Funnels

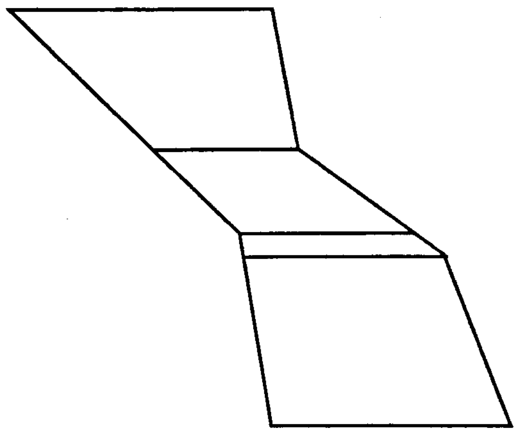
Given any non-simple funnel, we can view it as a union of simple funnels. The smallest set of simple funnels whose union forms a non-simple funnel F is known as the *decomposition* of F.

Definition: [decomposition of a funnel]: Let F be a PDF, let V be the set of vertices of F, and let V_d=V\{e1, e2, c1, c2}, where e1 and e2 are the funnel ends and c1 and c2 are the funnel corners. The set H_d of horizontal chords of F with one or both endpoints in V_d is called the *decomposition set* for F, and the partitioning H_d induces is called the *decomposition* of F. See figure 30 for an example of the decomposition of a funnel.

Given a funnel F with n vertices, we can partition it into O(n) simple funnels in O(n) time by doing a traversal and merge of the vertices of F, assuming the edges of F are stored in traversal order. Once we have obtained a decomposition, we can take the entry corner of F, project it through the topmost simple funnel of the decomposition in constant time, project the resulting point through the next simple funnel in the decomposition in constant time, and so forth, and in this way find a standard projection path through F in O(n) time.

6.4. Parallel Calculation of Smallest Paths through Non-Simple Funnels

figure 30



The decomposition of a PDF

In this section, we will show how the number of segments in a straightest path through a PDF may be obtained in $O(\log^2 n)$ time using $n/\log n$ processors, or in $O(\log n \log\log n)$ time using n processors. This assumes that the funnel is given as a vector of vertices sorted in traversal order.

6.4.1. Funnel Medians

To calculate the number of segments in a smallest path through a PDF in parallel, we make use of the fact that any end-segment of any PDF F may be considered as being made up of two line segments, such that all projection paths through F originating on one of these line segments have a certain number of segments, and all projection paths through F originating on the other of the segments have a different number of segments. This result is described more formally and proved in this section, and then used in later sections.

Notation: [intervals]: The following will make extensive use of standard interval notation, where $[x_1, x_2]$ denotes the real interval from x_1 to x_2 inclusive, (x_1, x_2) denotes the real interval from x_1 to x_2 exclusive of x_1 and x_2, and $(x_1, x_2]$ and $[x_1, x_2)$ denote the intervals from x_1 to x_2 exclusive of x_1 and x_2 respectively.

Definitions: [entry and exit medians, median pair, projection path pair, major and minor entry and exit intervals]: Let F be a PDF with entry e1, exit e2, entry corner c1, exit corner c2, entry segment s1, and exit segment s2, and having k segments in its standard projection path. We use x- and y-subscripts to denote the x- and y-coordinates respectively of the subscripted point. If there are points m1 on s1 and m2 on s2 such that the following two statements hold:

1) Every point w on s1 s.t. $w_x \in (e1_x, m1_x)$ projects to a point w' on s2 s.t. $w'_x \in (m2_x, e2_x)$, using a projection path of k+2 segments, and

2) Every point w on s1 s.t. $w_x \in [m1_x, c1_x]$ projects to a point w′ on s2 s.t. $w'_x \in [c2_x, m2_x]$, using a projection path of k segments

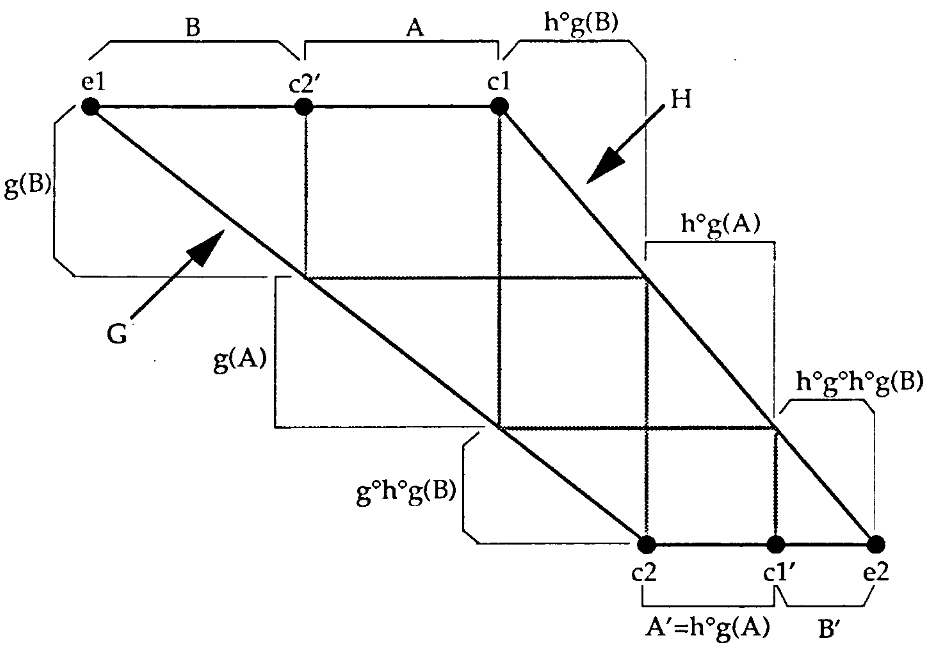
then the 2-tuple <k, k+2> is called the *projection path pair* for F, and m1 and m2 are called *median points* of F, with m1 being the *entry median* and m2 the *exit median.* Collectively, m1 and m2 are often called the *median pair* for F. The open-ended subsegment of s1 with points whose x-coordinates are in the range $(e1_x, m1_x)$ is called the *minor entry interval* for F, and the corresponding subsegment of s2 with points having x-coordinates in the range $(m2_x, e2_x)$ is called the *minor exit interval* for F. The subsegment of s1 with points having x-coordinates in the range $[m1_x, c1_x]$ is the *major entry interval* for F, and the subsegment of s2 with points whose x-coordinates are in $[c2_x, m2_x]$ is called the *major exit interval* for F.

We have the following lemmas.

Simple Interval Lemma: Let F be a simple PDF with entry e1, exit e2, entry corner c1, and exit corner c2. p(c1) is an exit median for F, and p(c2) is an entry median for F.

Proof: Let F be a simple PDF with a standard projection path of k segments, two diagonal boundary segments with the leftmost labeled G and rightmost labeled H, and all other parts of F labeled as shown in the example (with k=3) of figure 31. From G we can form the linear function g, valid on the interval $[e1_x, c2_x]$ and defined by g(x)=y s.t. (x, y) is a point on G. Similarly, from H we can define the linear function h(y)=x s.t. (x, y) is a point on H, h valid for $y \in [c1_y, e2_y]$. If w is a point on $\overline{e1c1}\backslash\{e1\}$ (a point on the entry segment but not the entry point) and w_x is the x-coordinate of w, then it is apparent that w_x will be the x-coordinate of the first segment of pp(w), and that the x-coordinate of the (2i+1)st segment of pp(w) will be $(h{\circ}g)^i(w_x)$, for i an integer s.t. $3 \leq (2i+1) \leq k$, where k is the number of segments in pp(w). As well, since h and g are both order-inverting linear

figure 31



functions, we have that $(h°g)^i$ is an order-preserving linear function for all $i > 0$, so that if w1, w2, and w3 are all points on $\overline{e1c1}$ s.t. $w1_x < w2_x < w3_x$, and $t = (h°g)^i$ for some i is s.t. $t(w1_x)$, $t(w2_x)$, and $t(w3_x)$ all exist, then $t(w1_x) < t(w2_x) < t(w3_x)$, which means the (2i+1)st segment of pp(w2) will be between the (2i+1)st segments of pp(w1) and pp(w3). If w1=c2′=p(c2), w3=c1, and w2 is any point on $\overline{c2'c1}$, then the x-coordinate of the kth segment of pp(w2) will lie between the x-coordinates of the point c2 and c1′=p(c1). Therefore, p(w2) will be on $\overline{c2c1'}$ and pp(w2) will contain k segments. In a similar manner, it can be shown that if a point w4 is on $\overline{e1c2'}\backslash\{e1, c2'\}$, then the jth vertical segment of pp(w4) is to the left of the jth vertical segment of pp(c2′), but that the j+1st vertical segment of pp(w4) is to the right of the jth vertical segment of pp(c1), and that therefore any point w4 on $\overline{e1c2'}\backslash\{e1, c2'\}$ must project to a point on $\overline{c1'e2}\backslash\{c1', e2\}$, using a projection path of k+2 segments. The same process works in reverse of course (i.e. projecting from the exit segment of F to the entry segment). This proves the simple interval lemma.

From the above, it is easy to see that the linear function t1 which maps the x-coordinates of points on $\overline{c2'c1}$ to the x-coordinates of their projections maps the interval $[c2'_x, c1_x]$ into and onto the interval $[c2_x, c1'_x]$. Thus, t1 must be defined by

$$t1(x) = (x - c2'_x)\left(\frac{c1'_x - c2_x}{c1_x - c2'_x}\right) + c2_x \quad \text{for} \quad c2'_x \le x \le c1_x$$

and similarly, the linear function t2 which maps the x-coordinates of points on $\overline{e1c2}\setminus\{e1, c2'\}$ to the x-coordinates of their projections must be defined by

$$t2(x) = (x - e1_x)\left(\frac{e2_x - c1'_x}{c2'_x - e1_x}\right) + c1'_x \quad \text{for} \quad e1_x < x < c2'_x$$

Definition: [height of a funnel]: If F is a funnel with k chords in its decomposition set, then the *height* of F is defined to be k+1. This is just the number of simple funnels in the decomposition of F.

General Interval Lemma: Any PDF F has a median pair.

Proof: [By induction on the height of F]:

Basis: If the height of F is 1, then F is a simple funnel, and by the simple interval lemma, we know that F has a median pair.

Inductive Assumption: Assume for some integer k that any funnel with height less than or equal to k has a median pair.

Inductive Step: Let F be a funnel with height k+1. Let C be a chord in the decomposition set of F, and let FA and FB be the subfunnels C induces on F. Both FA and FB will have height ≤ k, so the inductive assumption applies. Now let ma1 and ma2 be the entry and exit medians respectively for FA, and mb1 and mb2 be the entry and exit medians respectively for FB. There are three cases, one with ma2 to the left of mb1, one with ma2 to the right of mb1, and one with ma2 and mb1 being identical.

Note: In the following, for the sake of convenience we shall be rather loose with interval notation, and make statements such as, "the point r is on (e1, c1)", which should be read as, "r is a point on $\overline{e1c1}\setminus\{e1, c1\}$," or as, "r is a point on $\overline{e1c1}$ such that $r_x \in (e1_x, c1_x)$". The same shorthand will also be used with closed intervals.

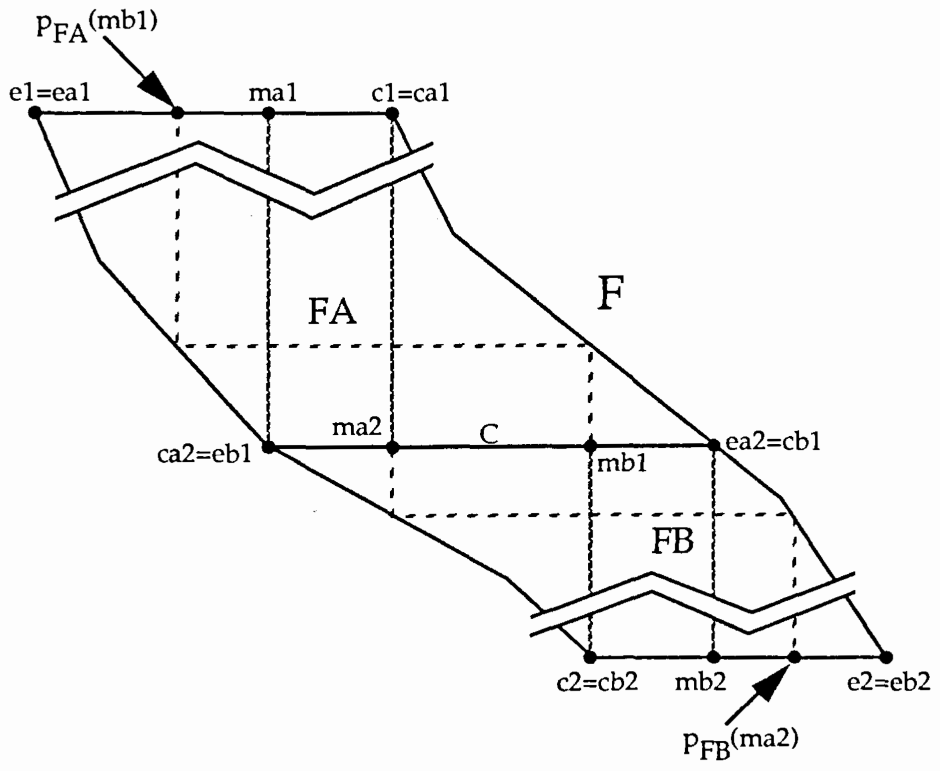
case 1: [ma2 to the left of mb1]: This situation is shown in figure 32. Assume that a standard projection path through FA has k_a segments, and a standard projection path through FB has k_b segments. Using the inductive assumption and the medians ma1, ma2, mb1, and mb2, it can be seen that points on (e1, p_{FA}(mb1)) will project through F to points on (p_{FB}(ma2), e2), using projection paths of $(k_a+2)+(k_b+2)-1$ segments, each projection path consisting of a k_a+2 segment subpath from a point w1 on (e1, p_{FA}(mb1)) to a point w2 on (ma2, mb1), a further k_b+2 segment subpath from w2 to a point w3 on (p_{FB}(ma2), e2), and a correction of -1 because the last segment of the path from w1 to w2 and the first segment of the path from w2 to w3 are a single segment in the path from w1 to w3. So, points on (e1, p_{FA}(mb1)) project through F to points on (p_{FB}(ma2), e2) with projection paths of k_a+k_b+3 segments. In a similar manner, we can show that points on [p_{FA}(mb1), ma1) project through F to points on [cb2, mb2) using paths of $(k_a+2)+k_b-1=k_a+k_b+1$ segments, and that points on [ma1, ca1] project to points on [mb2, p_{FB}(ma2)] using paths of $k_a+(k_b+2)-1=k_a+k_b+1$ segments. Note that the endpoints of these intervals may need to be handled as special cases.

Therefore, (e1, p_{FA}(mb1)) projects through F to (p_{FB}(ma2), e2) in k_a+k_b+3 segments, and [p_{FA}(mb1), ca1] projects to [cb2, p_{FB}(ma2)] in k_a+k_b+1 segments, and of course the inverse projections map (p_{FB}(ma2), e2) to (e1, p_{FA}(mb1)) and [cb2, p_{FB}(ma2)] onto [p_{FA}(mb1), ca1], using k_a+k_b+3 and k_a+k_b+1 segments respectively. This means that p_{FA}(mb1) and p_{FB}(ma2) meet the requirements for a entry/exit median pair for F, and the lemma is proved for case 1.

case 2: [ma2 to the right of mb1]: This case can be analyzed in the same manner as case 1, and the only difference is that projection paths will take k_a+k_b+1 or k_a+k_b-1 segments to go through F, rather than k_a+k_b+3 or k_a+k_b+1.

case 3: [ma2 and mb1 are the same]: This works out just like case 2, if we consider [ma2, mb1] as a single-point interval.

figure 32



Medians in F are formed by projecting the exit median of FA through FB and projecting the entry median of FB through FA respectively.

In fact, as can be seen by considering figure 32, p_{FA}(mb1) and p_{FB}(ma2) are just the projections of c2 and c1 respectively, and from the diagram and the above proof, it is clear that for any PDF F, the median pair for F can be obtained just by finding the projections of the corners of F.

6.4.2. The Parallel Funnel Algorithm

In describing a parallel algorithm for finding the number of segments in a straightest path through a PDF F, we will assume that the decomposition of F consists of 2^k simple subfunnels, for some integer k. This will make both the algorithmic description and timing analysis significantly simpler, without changing any of their salient features.

The algorithm can be outlined as follows:

1) Decompose F into 2^k simple subfunnels, and find the medians of these subfunnels.

   Next, consider F as the root node of a complete binary tree; the children of any funnel F´ in the tree are the two subfunnels F1´ and F2´ making up F´ s.t. F1´ and F2´ contain the same number of simple funnels. The leaves of F will be the simple funnels into which F can be decomposed. This leads to step 2.

2) While the tree described above has not been collapsed back into its root node F, do the following:

   For each pair of leaf subfunnels F1´ and F2´ with common parent F´, merge F1´ and F2´ by finding the medians of their union (the medians of F´) and the projection path pair for F´, store this information in F´, and remove F1´ and F2´ from the tree. On the next iteration of this step, F´ will be a leaf subfunnel.

We will process each level of the tree rooted at F in parallel, with the result that processing F will consist of k+1 distinct phases; the first phase decomposes F and calculates the median pair and projection path pair for all simple subfunnels of F, and each successive stage merges 2^j pairs of adjacent subfunnels from the preceding phase into 2^j subfunnels, for some $j<k$.

We will consider two different cases, one where we have O(n/log n) processors to process the polygon Q, and hence O(m/log m) processors for each funnel of m vertices, and another where we have O(n) processors for Q, and so O(m) processors for a funnel of m vertices. The first case will result in optimal parallel speedup for the entire algorithm, while the second case will give suboptimal speedup but a faster time bound. Neither case gives optimal speedup for the processing of funnels. (Recall that the dimensions of a smallest path through a funnel of m vertices may be calculated in O(m) time.) This is due to the fact that the parallel algorithm actually does more work—as well as calculating the dimensions of a smallest path through a funnel, it constructs a data structure which permits the projection of any point on the entry segment of the funnel through to the exit segment in O(log m) sequential time.

6.4.2.1. Decomposing a Funnel

As previously stated, we assume that we are given a PDF F represented as a vector of m vertices, sorted in traversal order. If the ends of F are not identified, this is irrelevant; we can in constant time and using m processors or in O(log m) time using m/log m processors test each funnel vertex as to whether or not it forms an acute angle. If it does, then it is a funnel end, otherwise it is not. Once the ends have been identified, it is a simple matter to split the vector of vertices into two vectors V1 and V2, each containing the vertices of one of the doubly-monotone paths which make up the funnel. This can be accomplished in constant time, with m processors, by indexing on each element of the original vector, relative to the positions of the funnel ends.

Once V1 and V2 have been obtained, we mark the vertices in each according as to whether they are on V1 or V2, and then merge V1 and V2 into a single vector V in $O(\log^2 m)$ time using m/log m processors [18], or in O(log m log log m) time using m processors [19; 20], depending on the number of processors we are using. Assuming our original funnel F was in standard position, V will be a vector containing the y-coordinates of the tops and bottoms of all the simple funnels in the decomposition of F. By storing with each element of V the two boundary edges of F incident on the vertex contained by that element of V, and then performing a parallel prefix operation on V so that boundary segment incidence information for vertices from V1 is distributed to vertices from V2 and vice-versa, we can find in further O(log m) time using m/log m processors the boundary segments of F intersected by the partition set of F. It is then a trivial matter to build a vector representing the simple funnels in F.

The "tree" of subfunnels described in the outline of the parallel funnel algorithm is a conceptual tool only—it need not be constructed.

6.4.2.2. Merging Funnels

Given two adjacent subfunnels whose medians have been computed, we merge them by projecting the appropriate median of each through the other, to find the medians for the parent funnel. So, in figure 32, to merge FA and FB into F, we project ma2 through FB, and mb1 through FA. At the same time, we calculate the projection path pair for F by considering the relation of ma2 to mb1, and applying the rules from the general interval lemma.

Definitions: [merge partitions]: Let s1 be the entry segment for some PDF F containing m vertices, and assume that s1 less the entry point for F can be given as a disjoint union of $k \le (m\text{-}2)$ subsegments $s1_1, s1_2, \ldots, s1_k$ of s1, where each endpoint of a subsegment may be either open (the subsegment does not include the endpoint), or closed (the subsegment does include the endpoint), and such that associated with each subsegment $s1_i$ is a linear function f_i which, given the x-coordinate of a point w in $s1_i$ returns the x-coordinate of $p_F(w)$. Such a partition of s1 is called an *entry partition.* We can define a similar partition on the exit segment s2 of F, containing linear functions which also map x-coordinates of points on their associated subsegments to the x-coordinates of the projections of those points through F; such a partition of s2 is called the *exit partition.* Together, the entry and exit partitions of F are known as the *merge partitions* of F. For any particular merge partition, the points which define that partition (the endpoints of the subsegments of the partition, along with information describing whether each endpoint is open or closed) are know as the *partition points* of that merge partition. Note that for any merge partition, the number of partition points in that partition is one greater than the number of subsegments defined by the partition.

Lemma: Every PDF has merge partitions.

Proof: [by constructive induction]: Every simple PDF has merge partitions defined just by the medians of the PDF and the four linear functions which can be used to project points back and forth in the PDF. Consider a non-simple PDF F made up of subfunnels FA and FB, containing j and k vertices respectively, as shown in figure 32. If we assume that FA and FB both have merge partitions, then we can form an entry partition for F by the following process:

1) Let points1 be the set of partition points for the entry partition part1 of FA.
2) Let points2 be the set of points obtained by projecting the partition points for the entry partition part2 of FB through FA.
3) Let points3 be the union of points1 and points2. points3 will induce a partition

part3 on the entry segment s1 of F. Let $s3_i$ be any subsegment (interval) of part3 defined by adjacent points of part3. If w is a point on $s3_i$, then $pp_F(w)$ will start in a subsegment $s1_j$ of part1, and will pass through a subsegment $s2_k$ of part2. If the projection functions associated with these subsegments are $f1_j$ and $f2_k$ respectively, then we can form the function $f3_i = f1_j \circ f2_k$, which will be a projection function for $s3_i$. Deciding if a subsegment of part3 contains its endpoints is easily done by inspecting the subsegment on part1 containing part3, and the subsegment of part2 whose projection through FA contains part1, to see if these subsegments contain their endpoints. points1 and points2 contain j-1 or fewer points and k-1 or fewer points respectively (by our assumption that FA and FB have merge partitions), so because the endpoints of points1 and points2 are identical and hence each be included only once, we have that the number of partition points in part3 will be at most (j+k-2)-2. Therefore, part3 will have at most j+k-5 subsegments in it, and so there will be at most j+k-5 linear functions associated with part3. Since F will contain at least j+k-3 vertices (up to three vertex counts may "disappear" in the merge of FA and FB), this means that part3 satisfies the requirements for an entry partition of F.

In a similar manner, we can find an exit partition for F. Of course, some attention to detail will be needed to ensure that open and closed endpoints of subsegments are processed properly. Keeping track of whether endpoints are open or closed is a minor technical matter, and will not be discussed further in this section.

Since simple PDFs have merge partitions, and since the merge of two funnels with merge partitions also has merge partitions, it is easy to see by an informal induction that all PDFs have merge partitions.

End of proof.

We return to the merge of FA and FB, and discuss how to obtain the merge partitions of F efficiently. The exit partition of FA can be expressed as a vector VA of O(m) rational numbers, with associated information to indicate whether or not a segment contains one or both of its endpoints, and the entry partition of FB can be expressed as a similarly sized vector of rationals and associated information VB. The elements of VA and VB can be considered as points whose y-coordinates are all identical and implicit, where each point may have further information associated with it. Assume that the projection function for a segment in a VA or VB is stored with one of the endpoints of the segment. Mark each element of VA and VB according to whether it is in VA or VB, and then take the merge of VA and VB by x-coordinate into a vector V in O(log m) time using m/log m processors [18], or in O(log log m) time using m processors with the algorithm of [20], which is implementable on a CREW PRAM[19]. Every element tB of V originally from VB will be in a subsegment of the exit segment of FA defined by elements tA1 and tA2 in V and originally from VA. Parallel prefix may be used in O(log m) time to distribute to all such tB from VB the information as to what elements tA1 and tA2 from VA define a subsegment containing tB. tA1 and tA2 are adjacent points in the exit partition of FA which define a subsegment of the exit segment of FA containing tB, and since the projection function for projecting a point in this subsegment through FA is associated with one of tA1 or tA2, we can use one processor to find this projection function and project tB onto the entry segment of F in constant time. The projection of all points defined by VB (the partition points of the entry partition of FB) onto the entry segment of FA may be then accomplished in constant time using m processors, and because the projected points will be in the same relative order, they may easily be cast into the form of a sorted vector.

The partition points of the entry segment of FA, together with the projections onto the entry segment of FA of the partition points of the entry segment of FB, are then marked as to which of these two classes they fall into, and merged into a vector PS of partition points for the entry segment of F, in O(log m) time. Each point pB in PS which was obtained by projecting a point from VB through FA will be on a subsegment of the entry segment of

FA defined by two adjacent points pA1 and pA2 from the entry partition of FA and contained in PS. Conversely each point in PS which was originally a partition point of the entry segment of FA will be bracketed by two points obtained from the projection through FA of two points from VB. This information may be distributed to each point in PS through two applications of parallel prefix to PS. For the purposes of the next paragraph, this information will be known as *bracketing information.*

Now, let p1 and p2 be two adjacent partition points in PS. The subsegment $\overline{p1p2}$ of the entry segment of F defined by p1 and p2 will be the intersection of a subsegment ssA of the entry segment of FA defined by adjacent points of the entry partition of FA, and a subsegment of the entry segment of FA, defined by the projection through FA of adjacent partition points of the entry segment of FB, which also define a subsegment ssB of the entry segment of FB. The projection of any point w in $\overline{p1p2}$ through F can be obtained by projecting it to a point of ssB, using the linear projection function fA_i associated with ssA, and then projecting the resulting point through FB, using the linear projection function fB_j associated with ssB. So, any point w on $\overline{p1p2}$ may be projected through F by use of the linear function $f_k = fB_j \circ fA_i$. f_k forms a projection function for $\overline{p1p2}$. fA_i and fB_j will both be obtainable as functions associated either with p1 or p2, or with points obtainable (in constant time, assuming the appropriate links have been kept) from the bracketing information associated with p1 and p2. Thus, we can form f_k in constant time, and since the entry partition for F will contain O(m) points, we can calculate the projection functions associated with the entry partition in constant time using m processors, or in O(log m) time using m/log m processors.. This completes the computation of the entry partition of F.

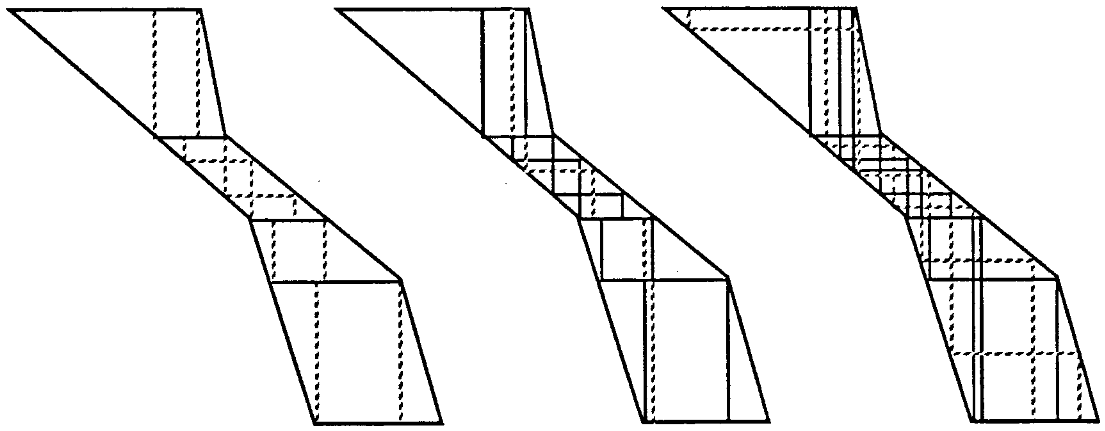
Of course, the exit partition of F may be calculated in a similar manner.

A diagram, showing the projections after each major step in the processing of a funnel F which decomposes into four simple funnels, is shown in figure 33.

6.4.2.3. Overall Timing for the Parallel Funnel Algorithm

From the above, it is apparent that the merge of two funnels, each containing O(m) vertices, may be done in O(log m) time using m/log m processors, or in O(log log m) time using m processors. Once a funnel F containing $2^k = O(m)$ simple funnels has been decomposed, it may then be processed using k merge phases. (If necessary, a funnel may be "padded out" with null simple funnels so that it contains a total of 2^k simple funnels—this will never more than double the number of simple funnels to be considered.) If we index the merging of the simple funnels of F as the first merge phase, and assume that processors are evenly distributed over subfunnels during any particular merge phase (which will be true to within as small an error as desired in the asymptotic limit), then

figure 33



Projections in a funnel being merged in three steps. Dotted lines indicate projections produced by the step just completed, solid lines indicate projections produced in previous steps. The projection paths are shown for ease of understanding, but are not actually generated by the process. After the first step, entry and exit partitions have been found for the simple funnels. After the second step (the first merge), entry and exit partitions have been found for the funnel containing the top two simple funnels, and for the funnel containing the bottom two simple funnels. After the third step (the second merge phase), entry and exit partitions have been found for the entire funnel. Because each projection is accomplished using linear functions produced in the previous step, no projection takes more than constant time. However, each merge phase will take more than constant time, due to the need to merge vectors of projected points.

merge phase i will perform 2^{k-i} seperate merges in parallel, each of these merges involving two subfunnels containing a total of not more than $4*2^i$ vertices. Thus, it will take O(log i) time to merge subfunnels at merge phase i if we are assuming m processors, or O(i) time to merge the same subfunnels if we are assuming m/log m processors. Therefore, the total timing for the merge portion of the parallel funnel algorithm on a funnel F containing m vertices is just

$$T(m) = \sum_{i=1}^{\log m} i = O(\log^2 m)$$

using m/log m processors, or

$$T(m) = \sum_{i=1}^{\log m} \log i = O(\log m \log \log m)$$

using m processors. The decomposition of F may be accomplished with either timing, and so the dimensions of a smallest path through any PDF F with m vertices can be found in $O(\log^2 m)$ time using m/log m processors, or in O(log m log log m) time using m processors.

6.5. Smallest Paths through non-PDFs

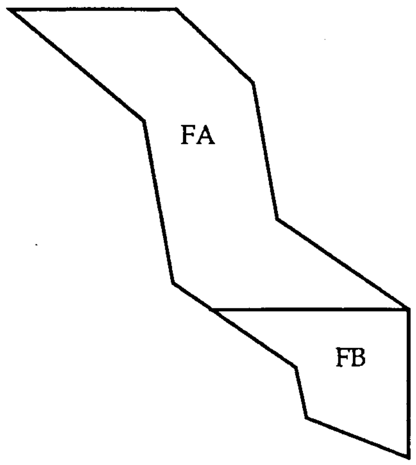
There are a number of extensions and modifications which must be made to the preceding sections, if we wish to handle anything other than parallel diagonal funnels.

Firstly, degenerate (i.e. triangular) and rectangular funnels should be handled as special cases. Two-segment smallest paths through such funnels are trivially findable, and so we need not apply any of the definitions or methods outlined above for general PDFs.

We may encounter non-parallel funnels, those in which the end-segments are perpendicular rather than parallel. In such a situation, a carefully drawn line will divide the funnel into a PDF and another funnel through which every straightest path has two segments, and so this case may easily be solved. See figure 34 for an example of this.

Next, non-diagonal funnels (those in which more than two boundary segments may be rectilinear) require some care. First, since each funnel end may have two rectilinear boundary segments incident on it, there may not be a unique pair of end-segments. A simple workaround to this problem is to process the funnel for every possible pair of end-segments, a maximum of four different combinations, and keep all the results for possible future use. Of course, more elegant solutions to this difficulty are easily imaginable. A second difficulty posed by the inclusion of rectilinear boundary segments which are not end-segments is the possibility that a projection path segment may run along a boundary segment. The current definition for the projection function does not take this into account, and must be appropriately modified, as must various other related elements of the preceding sections. These modifications are tedious but otherwise trivial. Thirdly, the inclusion of rectilinear boundary segments other than the end-segments raises the

figure 34



The above non-parallel funnel may be considered as a composition of a parallel funnel FA, and a funnel FB whose particularly simple structure makes analysis of straightest paths through FB quite easy.

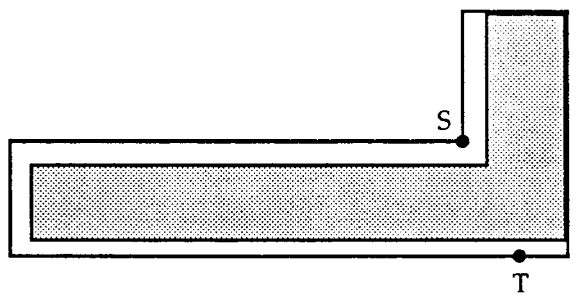
possibility that some of the simple funnels in the decomposition of a PDF may be rectangular, which in turn raises the question of how to define the end-segments for a rectangular funnel in such a situation. The simplest solution is to simply remove all rectangular subfunnels from the decomposition of a PDF, as these subfunnels can have no effect on the straightness of paths through the PDF—they affect the lengths of paths only.

Finally, various combinations of these cases must be considered. An enumeration of such cases would be boring to the point of tears, and for the most part unenlightening, and I leave it to the intrepid reader to verify that the techniques outlined above are applicable (in modified form) to all such combinations, if he or she so wishes to check.

6.6. Summary of Results Concerning Funnels

It has been shown that given any PDF F containing m vertices, the dimensions of a smallest path through F (from one end of F to the other) may be calculated in O(m) sequential time, or in O($\log^2$ m) time using m/log m processors, or in O(log m log log m) time using m processors. Brief descriptions have also been given as to how these results may be generalized to any funnel, with no loss of asymptotic efficiency.

figure 35



Shortest and straightest paths exist between S and T, but no path between them is smallest.

7. Smallest Paths through Rectilinear Obstacles

Definition: [RO Smallest Path Problem]: Consider the following problem: We are given a set of simple, nonintersecting rectilinear polygons, specified as lists of coordinates indicating the positions of their corners; a source point S and a sink point T, neither in any of the obstacles, although they may occur on the obstacle boundaries; a rational number r; and an integer k. In such a situation, a smallest path from S to T may not exist— see figure 35. One problem is to determine if there exists a rectilinear path from S to T, not intersecting any of the polygons except possibly at the polygon boundaries, such that the length of the path is $\leq$ r and the number of bends in the path is $\leq$ k. We call this problem the *RO smallest path problem.*

In this chapter we will show how this problem may be solved in $O(n^3)$ time, where n is the number of corners in the set of obstacles.

With each corner in the obstacle set, we associate two arrows, originating at the corner, and going out along the segments of the obstacle which form the corner. We also associate four arrows with each of the source and sink, in the four rectilinear directions. (See figure 36.) There are no other arrows to be talked about in this problem, so, "arrows," always refers the arrows associated with obstacle corners, or the source or sink.

Arrow Theorem: Let I be an instance of the RO smallest path problem, with n corners in the obstacle set. Assume there exists a rectilinear path P of k bends and length r from S to T, not passing through any of the obstacle polygons. Then there exists a directed rectilinear path P′ from S to T of k or fewer bends and length less than or equal to r which does not pass through any of the obstacle polygons, such that each segment of the path passes through the origin of at least one arrow in the direction pointed by the arrow.

Proof: [constructive]: We show how P may be modified to produce a path P′ which satisfies the theorem.

It is apparent that segments of P incident on the source or sink will satisfy the theorem. (If it is a sink arrow, "passes through," may simply mean, "intersects.") Now consider any segment of P other than the initial or final segment. Such a segment may have its adjacent segments on one side, or on opposite sides, as shown in figure 37.

figure 36

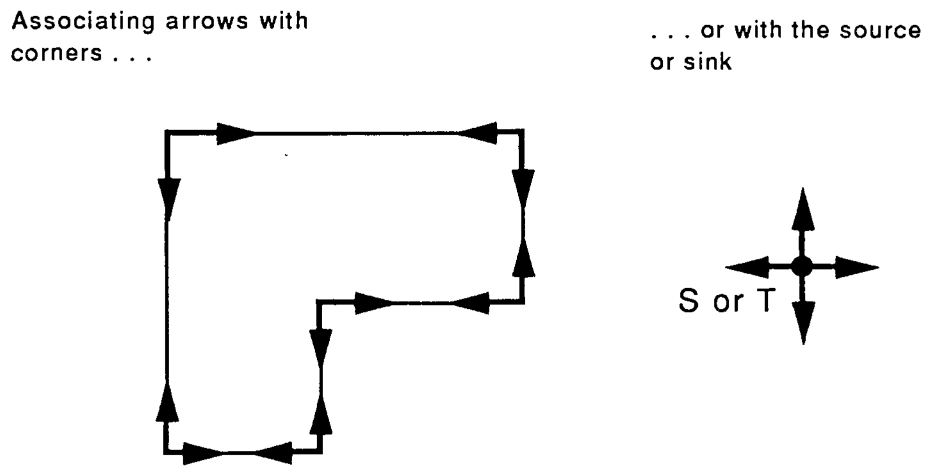
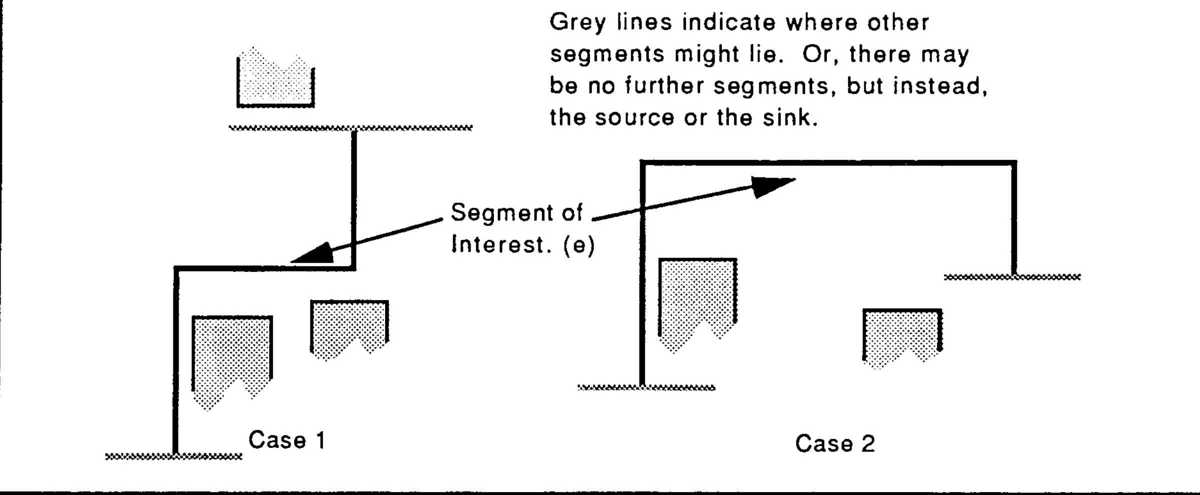

Associating arrows with corners . . .
. . . or with the source or sink
S or T

figure 37

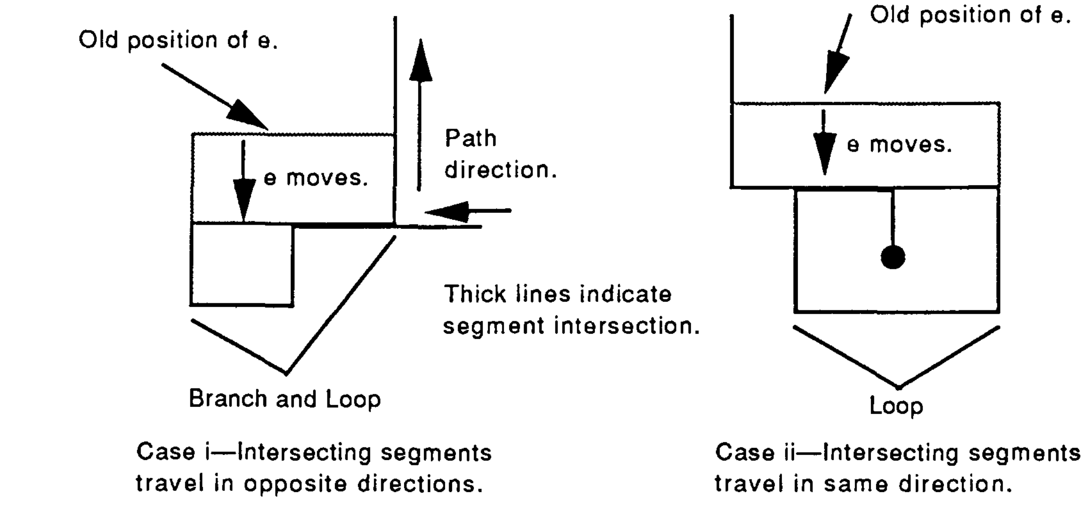

Grey lines indicate where other segments might lie. Or, there may be no further segments, but instead, the source or the sink.
Segment of Interest. (e)
Case 1
Case 2

Consider case 1. If the segment of interest, call it e, passes through the origin of an arrow pointing in the direction of travel of the segment, then that segment is in accordance with theorem 1, and we need not consider it further. Otherwise, our strategy is to raise or lower e (or, if e is vertical, to move it to one side or the other), without causing it to go through an obstacle. The choice of raising or lowering is decided by which direction the path is traveling—we shift e in the direction opposite to the direction of travel of its two adjacent segments. Say we lower e. This is accomplished by decreasing the y coordinates of the endpoints of e, and shortening or lengthening the adjacent segments to keep the path connected. In doing so, one of three things can happen:

a) The segment we are shortening disappears, and e comes into contact with the source or the sink. If this happens, e now passes through the origin of one of the source or sink arrows, in the direction of the arrow.

b) e hits the boundary of an obstacle. If this happens, the geometry of the situation ensures that e will pass through the origin of at least one corner arrow in the direction of the arrow.

figure 38


Old position of e.
e moves.
Path direction.
Thick lines indicate segment intersection.
Branch and Loop
Case i—Intersecting segments travel in opposite directions.
Old position of e.
e moves.
Loop
Case ii—Intersecting segments travel in same direction.

c) e intersects another path segment with the same orientation. (i.e. horizontal or vertical.) See figure 38. In this case, a loop, or a branch, or a loop at the end of a branch, is formed. If the intersecting segments travel in opposite directions (case i of figure 38), we can simply chop off the loop and/or branch. If the intersecting segments travel in the same direction (case ii), we have a spiral. One of the source or sink will be inside the spiral, and one will be outside the spiral. In this case, there will be a loop, but no extraneous branch. We simply remove the loop, except for that part of it where the two segments intersect.

If we consider case 2 of figure 37, essentially the same transformations can be applied. The only difference is that e must be moved so that both of its adjacent segments become shorter.

Considering the three possible results of transforming the path in the manner outlined above, we see that they share the following two properties:

1) They result in a path with a length less than or equal to the original path, and number of bends less than or equal to the original path.

2) They result in a reduction by at least one of the number of path segments which do not pass through the origin of an arrow pointing in the same direction as the path segment.

And, since at least one of the above three transformations can be applied whenever there is at least one path segment not going though the origin of an arrow in the direction of the arrow, we conclude that repeated applications of the transformations will eventually produce a path with length less than or equal to the original, and number of bends less than or equal to the original, such that each path segment intersects the origin of an arrow which points in the same direction as the path segment.

This proves the arrow theorem.

With this theorem proved, we can now construct an $O(n^3)$ algorithm to determine if, given a set of rectilinear obstacles, points S and T, a rational r, and an integer k, there exists a rectilinear path from S to T, of length less than or equal to r and number of bends less than or equal to k, which does not pass through any of the given obstacles.

The algorithm makes use of the fact that, given a set of m rectilinear line segments H we can, in $O(m \log^2 m)$ time, construct a *segment tree*, which then permits one to take another rectilinear segment $\bar{e}$ and find in $O(\log^2 m)$ time whether $\bar{e}$ intersects a segment in H [7][3].

The first step in the algorithm is to construct a segment tree consisting of the sides of obstacles. This takes $O(n \log^2 n)$ time.

3. These time bounds can actually be improved slightly beyond this, at a considerable increase in complexity, as explained by Lipski's paper. These improvements have no effect on the net running time of the algorithm presented here.

Next is a special case test, which consists of determining if S and T share the same x or y coordinate, and if they do, test using the segment tree to see if the straight line joining them intersects any obstacles. If not, then we are done, as we have a shortest possible path with the smallest possible number of bends. This test handles the possibility that there might be a zero bend path joining S and T, and takes $O(\log^2 n)$ time to perform.

Then, we construct an $O(n^2)$ adjacency matrix, M, formed by taking the cross-product of the set of arrows Arr with itself. The matrix is initialized to infinity. For every vertical arrow v and horizontal arrow h, the matrix element M[v,h] is then assigned the value x iff there is a directed rectilinear path having one bend, of length x, from the origin of v to the origin of h such that the segment intersecting the origin of v has the same direction as v, and the segment intersecting the origin of h has the same direction as h. Likewise, the element M[h,v] is given value x iff there is a path of one bend from the origin of h to the origin of v, etc. Determining if there is a path from v to h or h to v is simple. We first ask if v and h are in the proper configuration, i.e. is it possible to draw a one-bend path from the first to the second, maintaining correct directionality? This test can be performed by cases, in O(1) time. If such a path exists, there will be only one—split it into its two segments, and test each to see if it passes through the edge of an obstacle, in $O(\log^2 n)$ time. If neither segment does, then M[v,h] or M[h,v] is set to the rectilinear distance between their origins. There are $O(n^2)$ <v,h> pairs to consider, so setting up M takes $O(n^2 \log^2 n)$ time.

There are two things to realize when considering the rest of the algorithm. First, when trying to find a path with length no greater than r and bends no more than k, we need not look for paths with loops, as any such path contains a path without loops which is closer to optimal. Second, any path without loops which meets the conditions given in the arrow theorem can have at most n+1 bends, as shown below.

Lemma: Consider a RO smallest path problem instance I and path, as described in the arrow theorem. Then there exists a (loopless) path from S to T of the same or lesser length and the same or lesser bends, and with no more than n+1 bends, where n is the total number of corners of polygons in the problem.

Proof: We see this by counting arrows. The first and last segment in the path each touch the origins of at least four arrows. The other segments each touch the origins of at least two arrows. There are only (2n+8) arrows in total—two on each obstacle corner, and four each on S and T—so a path which does not loop (intersect itself) can have at most n+2 segments—n+1 bends.

So, we need only look for (loopless) paths with n+1 or fewer bends.

The main part of the algorithm will fill in an $O(n^2)$ array, A. The array is indexed first on the set of arrows Arr of the problem instance I, and second on an integer index ranging from 1 to n+1. A is originally initialized to infinity. At the end of the algorithm, the element A[v, i] of A will have a finite value x iff there is a path of length x and i bends, starting at S and ending at the origin of arrow v, such that the last segment of the path travels in the same direction as v. Similarly for A[h, i].

We set the A[v, 1]'s and A[h, 1]'s by the following:

```
FOR each v, element of the vertical arrows of Arr
    FOR h, the horizontal arrows originating at S
        A[v,1] := M[h,v]
FOR each h, element of the horizontal arrows of Arr
    FOR v, the vertical arrows originating at S
        A[h,1] := M[v,h]
```

This leaves each element of A with a second coordinate of 1 set to the distance of a one-bend path from S to the arrow corresponding to the first coordinate of the element, assuming such a path exists. If no such path exists, the element is set to infinity. This entire initialization process takes O(n) time.

The main part of the algorithm is as follows:

```
FOR N := 2 to n+1
    FOR each v, element of the vertical arrows of I
        FOR each h, element of the horizontal arrows of I
            IF A[h, N-1] + M[h, v] < A[v, N]
            THEN A[v, N] := A[h, N-1] + M[h, v]

            IF A[v, N-1] + M[v, h] < A[h, N]
            THEN A[h, N] := A[v, N-1] + M[v, h]
```

Recall that A[h, i] is the i-bend distance from S to h, and M[h, v] is the 1-bend distance from h to v. Each iteration of the outer loop extends rectilinear paths rooted at S by one bend. The "<" tests in the IF statements ensure that of multiple i-bend paths to an arrow, the shortest i-bend path is chosen. Order of calculation in the two inner loops is not of concern, because only information calculated before the present iteration of the outer loop is used. Time for this part of the algorithm is, obviously, $O(n^3)$.

Finally, given a maximum length r, and a maximum number of bends k, we inspect the elements A[a,i], where a is one of the four arrows associated with T, and i is in the range 1..min{n+1, k}. If any of these elements of A have a value less than or equal to r, then there is a path of length no greater than r, and bends no greater than k, from S to T. Performing this check takes O(n) time.

In summary, the algorithm is comprised of the following parts, taking the given times:

1) Build the segment tree—O(n $\log^2$ n).
2) Check for straight line from S to T—O($\log^2$ n).
3) Build M—O($n^2 \log^2$ n).
4) Build A—O(n^3).
5) Check for an answer—O(n).

The total time for the algorithm is O(n^3), where n is the number of vertices in the obstacle set.

A few notes about this algorithm are in order.

It is inherently inefficient in making use of r and k, using them only when the main part of the algorithm is done. Speeding up the algorithm will likely involve using these values to limit the amount of work done by the algorithm, *before* it checks for an answer.

Once the algorithm has run, we can get the answer for any r and k in O(n) time.

The algorithm answers if there is a path fitting r and k, but does not actually give one. Modifying it to return a path is quite easy, as we can simply keep a matrix of links corresponding to the elements of A, and whenever an element of A is modified, indicating that it has been "reached" by some path, we update the corresponding element of the link matrix to point to the element of A from which the path "reaching" e was extended. Backtracking will then enable us to extract a path.

8. Concluding Remarks

In this thesis, we have defined a new problem, that of finding smallest paths between points, which combines aspects of two previously well-known and useful problems, namely finding minimum-length rectilinear paths and finding minimum-segment rectilinear paths. It has been shown that smallest paths between two points in a simple polygon always exist (except for easy to detect cases involving pathological vertices). The problem of actually finding smallest paths in a simple polygon has been studied and examples have been given to indicate that generating a smallest path in a simple polygon is a problem with an inherently large lower performance bound, as a function of the number or vertices in the polygon. By proving and exploiting non-obvious properties of smallest paths in simple polygons, an efficient $O(n \log n)$ time algorithm has been given to find dimensions of smallest paths between two points in a simple polygon, where n is the number of vertices in the polygon. This algorithm has been generalized to an $O(\log^2 n)$ time parallel algorithm with optimal parallel speedup, and to an $O(\log n \log \log n)$ time parallel algorithm with slightly suboptimal parallel speedup. In the course of designing both the parallel and sequential smallest path algorithms, an interesting subproblem, that of finding smallest paths through funnels, has been studied and solved. Finally, some research concerning smallest paths in an environment containing rectilinear obstacles has been done. In this environment, it has been shown that smallest paths do not necessarily exist between points, and the smallest path problem has been cast in a more general form, as a decision problem. An algorithm has been developed to decide this problem in time polynomial in the number of obstacle vertices.

Research to date has left many questions unanswered, and several interesting problems suggest themselves. The current algorithm for smallest paths in simple polygons requires that the source and destination of the path be specified before any processing begins. It would be worthwhile to generalize this to a query problem, where only the source point S is given, and the polygon is preprocessed so that given any other point T in the polygon, the dimensions of a smallest path between S and T may be quickly ascertained. Preliminary research into this problem indicates that it is perhaps solvable in $O(n \log^2 n)$ or even $O(n \log n)$ sequential preprocessing time, with $O(\log n)$ query time. An even more interesting prospect is that of preprocessing the polygon in such a manner that a query as to the dimensions of a smallest path between any two points in the polygon may be quickly answered. Although formidable, some very preliminary research indicates that this form of the query problem may also be solvable with a very reasonable worst-case time bound, by isolating vertices of the polygon through which all smallest paths between certain sections of the polygon will travel.

Given that the above query problems may be solved in an efficient sequential manner, it seems likely that queries could then be easily conducted in parallel. In order to make maximal use of this ability, however, the preprocessing for the problem solutions should also be accomplished in parallel, and parallelizing this preprocessing would be another worthwhile goal. The research done on funnels in this paper should be a significant help in achieving this aim.

Problems involving smallest paths not restricted to the interior of simple polygons have been left virtually untouched by this thesis, and a large number of interesting topics suggest themselves in this area. One task of immediate interest would be that of obtaining a more efficient polynomial time algorithm for determining minimal paths in environments containing rectilinear obstacles. Even after obtaining efficient basic smallest path algorithms in this environment (to the degree possible, as smallest paths in such an environment will not always exist), practical applications in many areas may well benefit from query and/or parallel versions. Finally, considering non-rectilinear (polygonal) obstacles should provide some very interesting, and in all likelihood some very difficult, problems. Finding efficient smallest path algorithms in such environments will necessitate the use of techniques for finding smallest paths in environments with rectilinear obstacles, along with techniques for handling constructs such as funnels. Even after both of these problems are well understood, combining their solutions to provide an algorithm for finding smallest paths around general polygonal barriers will probably require a significant amount of effort.

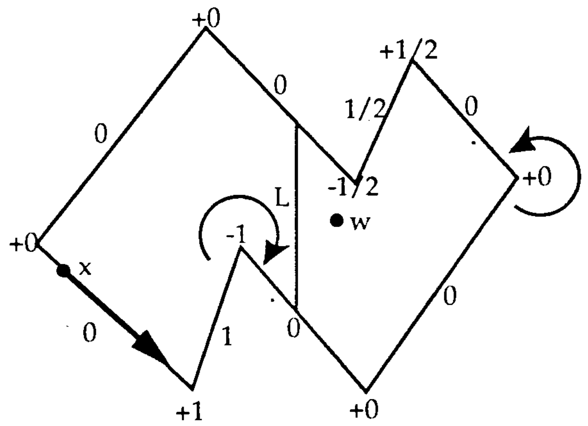
Appendix I: traversal numbers.

traversal numbers provide a convenient algorithmic method of determining if a point w is inside or outside of a simple polygon Q. We illustrate their use with an example.

Consider the polygon shown in figure 39. We wish to determine if the point w is inside or outside of Q, by doing a traversal of bdy(Q). Assume for now that w is not on bdy(Q). The algorithm we will use is best understood by visualizing oneself at the point x on bdy(Q), facing in the direction indicated by the arrow. To do a traversal of bdy(Q), walk along the boundary until a vertex is reached, then turn in the obvious manner to line up with the next boundary segment (technically, turn continuously so that the semi-infinite ray indicating current direction does not sweep over the boundary segment being exited from), and continue the traversal, going on in this manner until the starting point is reached.

The traversal is started with a counter variable initialized to some number, generally 0, and is performed looking only straight ahead, even when doing a turn at a polygon vertex.

figure 39



Traversal numbers (vertex increments and cumulative sums) for a traversal of Q starting at x, relative to w. Traversal numbers for the two subpolygons induced by L have not been shown, to maintain clarity, but note that the accumulated sum across L for the rightmost of these subpolygons is consistent, and so w is inside this subpolygon, where the opposite is true for the leftmost subpolygon. Circular arrows indicate direction of turns at sample vertices.

Thus, the line of sight during the traversal may be considered as a semi-infinite ray. We view ourselves (the person doing the traversal) as stationary, while the plane moves around us; every time w comes into the line of sight from the left, or leaves the line of sight to the right, the counter variable is incremented by 1/2; every time w comes into the line of sight from the right, or leaves the line of sight to the left, the counter variable is decremented by 1/2. Thus, every time w appears to cross the line of sight from left going to the right, the counter variable will be incremented by 1, and every time w appears to cross the line of sight going to the left, the counter variable is decremented by 1. The turn performed at each vertex of the polygon will have a certain increment or decrement associated with it, and the segments of bdy(Q) will have some cumulative sum associated with them, as the line of sight will never change when traversing along a boundary segment. These increments and sums are shown in figure 39.

As it turns out, if upon returning to the starting point, the current value of the counter variable is the same as its initial value, we will have that w is inside of Q. If the counter has any other value, then w is outside of Q. This is provable through either the use of partial derivatives (which will show this property for general simple curves), or (for simple polygons only), through induction. A sketch of an inductive proof for this statement as applied to simple polygons is given at the end of this appendix.

The advantage of using traversal numbers in this manner is that if we now take a line segment L in Q inducing a subpolygon Q′, we can find the increments across the vertices L forms in Q′ in constant time, and in further constant time check if the accumulated sum across L is consistent—if it is, then w will be in Q′, otherwise w will not be in Q′. The reader is encouraged to verify this fact with L as shown in figure 39.

There remains the problem of what to do if w is on L or on bdy(Q). Handling the case that w is on L is easy—we can simply check in constant time if this is the case. To handle the case that w might be on bdy(Q), we can preprocess Q by numbering its boundary segments, and determining the number N of the boundary segment of Q on which w resides. Any subpolygon of Q induced by a horizontal or vertical line segment will have an external boundary defined by some numbered sequence (taken modulo the number of boundary segments of Q) of boundary segments of Q—determining, in constant time, if N is in this range will answer if N is in the induced subpolygon. (There are a few other details necessary to use this technique—for instance, the first and last segments of the exterior boundary of the induced subpolygon of Q may not be complete boundary segments of Q, and this must be tested for—but the details are minor and easily handled.)

The remainder of this appendix sketches a proof that traversal numbers in simple polygons will operate as described above.

Definition: [outcropping of a simple polygon, shear]: Let Q be a simple polygon, let $\overline{w1w2}$ and $\overline{w2w3}$ be two adjacent boundary segments of Q with endpoints w1, w2, and w3, and assume that the line $\overline{w1w3}$ is entirely contained in Q and intersects bdy(Q) at only w1 and w3. The triangle T1 whose boundary is defined by $\overline{w1w2}$, $\overline{w2w3}$, and $\overline{w1w3}$ is called an *outcropping* of Q, and the *shear* of Q relative to T1, denoted shear(Q, T1) is given by shear(Q, T1) = $(Q\backslash T1)\cup\overline{w1w3}$.

Lemma: Let Q be a simple polygon with four or more boundary segments, and let B be a boundary segment of Q. Q contains an outcropping which does not intersect B except at possibly a single point which is one of the endpoints of B.

Proof: [by induction]:

Basis: If Q is just a quadrilateral, then it will have at least one diagonal which will induce two outcroppings of Q. Any boundary segment B of Q can be a boundary segment for at most one of these outcroppings, and so the other of the two outcroppings will satisfy the lemma.

Inductive Assumption: For some k, k≥4, assume that any simple polygon with k or fewer sides (and at least 4 sides) satisfies the lemma.

Inductive Step: Let Q be a polygon with k+1 sides, and let B be one of the sides of Q. We take it as obvious that there are two points w1 and w2 in Q such that the line $\overline{w1w2}$ is contained in Q and does not intersect bdy(Q) except at the points w1 and w2. $\overline{w1w2}$ induces two subpolygons Q1 and Q2 on Q, each with k or fewer sides. Assume WLOG that Q1 is the subpolygon which has B as a boundary segment. If Q2 is an outcropping of Q, then we are done, otherwise Q2 has at least 4 vertices and by the inductive hypothesis we can find an outcropping of Q2 which does not contain $\overline{w1w2}$; this will also be an outcropping of Q, and obviously it will satisfy the lemma, so we are again done.

Lemma: Let Q be a simple polygon with five or more vertices (i.e. five or more boundary segments). Q has at least two outcroppings T1 and T2 such that T1 and T2 do not intersect except possibly at a single point which is a vertex of Q.

Proof: We take it as obvious that there are two points w1 and w2 in Q such that the line $\overline{w1w2}$ is contained in Q and does not intersect bdy(Q) except at the points w1 and w2. $\overline{w1w2}$ induces two subpolygons Q1 and Q2 on Q. If one of Q1 or Q2 (say Q1) is an outcropping, then it is T1, and in this case, Q2 must have four or more sides, so by a previous lemma we can find an outcropping of Q2 which does not contain $\overline{w1w2}$, and this will be T2. If neither Q1 nor Q2 is a triangle, then be a previous lemma they must respectively contain outcroppings T1 and T2 which do not contain $\overline{w1w2}$. In either case, T1 and T2 will satisfy the lemma.

Lemma: Let Q be a simple polygon, and let w be a point in Q and not in bdy(Q). The method or traversal numbers described previously provides a valid way of determining whether or not w is in Q.

Proof: [by induction on the boundary segments of Q]:

Basis: If Q is a triangle, then during a traversal of Q, w will never appear to cross the line-of-sight of the person doing the traversal if w is inside of Q, and will cross the line-of-sight only once if it is outside of Q. In either case, the result is as desired. If Q is a quadrilateral, then analysis is slightly more complex (but only slightly), as a quadrilateral may have a single vertex which is concave into the quadrilateral. It is left as an exercise to the reader to verify that traversal numbers function correctly in quadrilaterals.

Inductive Assumption: Assume for some integer k, k≥4, that traversal numbers provide a valid method of determining if a point w is inside or outside a simple polygon Q, if Q has k or fewer boundary segments.

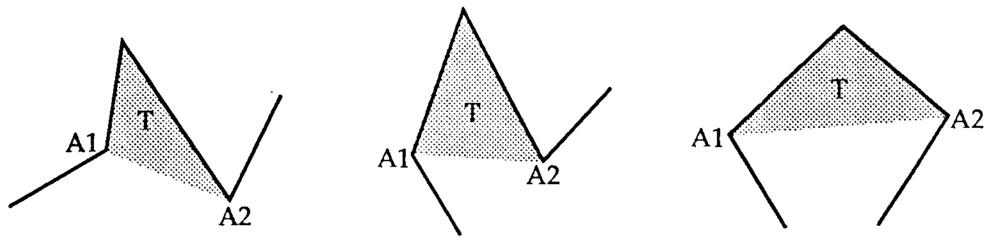
Inductive Step: Let Q be a simple polygon with k+1 boundary segments, and let w be a point inside or outside of Q, but not on bdy(Q). By a previous lemma, Q will contain two outcropping T1 and T2 which do not intersect except except possibly at a vertex of Q. Since w is not on bdy(Q), it will be contained in at most one of T1 or T2, and one of T1 or T2 will not contain w. Call the outcropping which does not contain w T.

We consider the two segments of bdy(Q) which are also boundary segments of T, along with the two segments of bdy(Q) adjacent to these segments. These are the *involved segments*. There are three cases to consider, as shown in figure 40, depending on whether or not the angles A1 and A2 between T and the segments adjacent to it are convex or concave relative to Q. We will prove the first of these cases, and leave the others to the reader.

Consider figure 41, which shows us the outcropping T of Q, and the boundary segments of Q immediately adjacent to T. From the way we chose T, we know that w is not in T. Therefore, the traversal numbers for Q should indicate that w is inside or outside of Q if the traversal numbers for shear(Q, T) respectively indicate that w is inside or outside of shear(Q, T). By the inductive assumption we know that traversal numbers will give a correct result on shear(Q, T), so if we can show that traversal numbers give the identical answer on Q, we will have completed our inductive step.

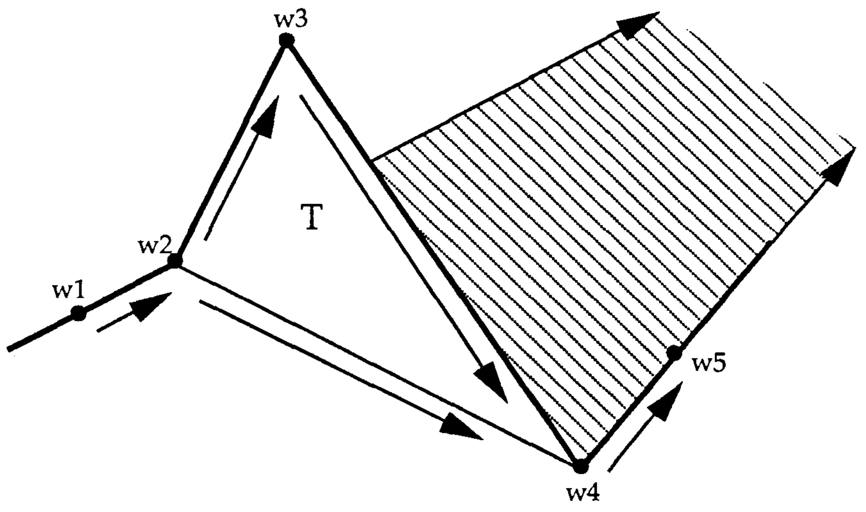
That traversal numbers answer identically as to whether or not w is in Q or in shear(Q, T) may be seen by considering two points w1 and w5, one on each of the boundary segments of Q adjacent to T. We will show that the traversal number changes by the same amount when going from w1 to w5 in Q is it does when going from w1 to w5 in shear(Q, T). Since Q and shear(Q, T) are otherwise identical, it will follow that the traversal sequences are otherwise identical, and so traversal numbers will obviously report w inside of Q iff they report w inside of shear(Q, T).

figure 40



The three possible cases for an outcropping T in Q, depending on whter the angles the outcropping forms with the boudary segments on either side of it are concave or convex relative to Q.

figure 41



Arrows along boundary segments indicate direction of traversal. If w is in the striped region, then traversing the path w1w2w3w4w5 or the path w1w2w4w5 will result in a net decrement of 1 to the traversal number counter, and if w is not in this region, then either traveral will leave the traversal number from w1 to w5 unchanged.

That the traversal number changes by the same amount in going from w1 to w5 in either Q or shear(Q, T) can be shown by performing "symbolic traversals" of the paths w1w2w3w4w5 and w1w2w4w5, and asking if there are any places where w could be located so as to cause the traversal increment over one path to be different from the traversal increment over the other. By doing such "symbolic traversals", it is easy to see that the only place w might be which would cause this to be true is inside T, which by our choice of T is not possible. The other possibilities are as desired; in figure 41, if w is in the striped region, then either traversal from w1 to w5 will result in a decrement of 1 to the traversal number, while if w is in any other permissible region, then neither traversal from w1 to w5 will have a net effect on the traversal number. (Note: Actually, we must take into account the special cases that w may occur on one or both of the rays which define the boundary of the striped region. This is quite simple, it simply involves considering traversal increments of $\pm 1/2$, and we leave it to the reader to make this extension.)

Depending on the configuration of the boundary segments of Q adjacent to T, the striped region may be finite or infinite, but because it does not depend on the boundary segments of Q which define T, it will be the same in both Q and shear(Q, T).

Thus, the answer returned by a traversal of Q as to whether w is inside Q will be the same as the answer returned by a traversal of shear(Q, T) as to whether w is inside shear(Q, T). Since w is inside shear(Q, T) iff w is inside Q, and since by the inductive assumption traversal numbers answer correctly as to whether or not w is inside shear(Q, T), we have that traversal numbers also answer correctly as to whether or not w is inside Q.

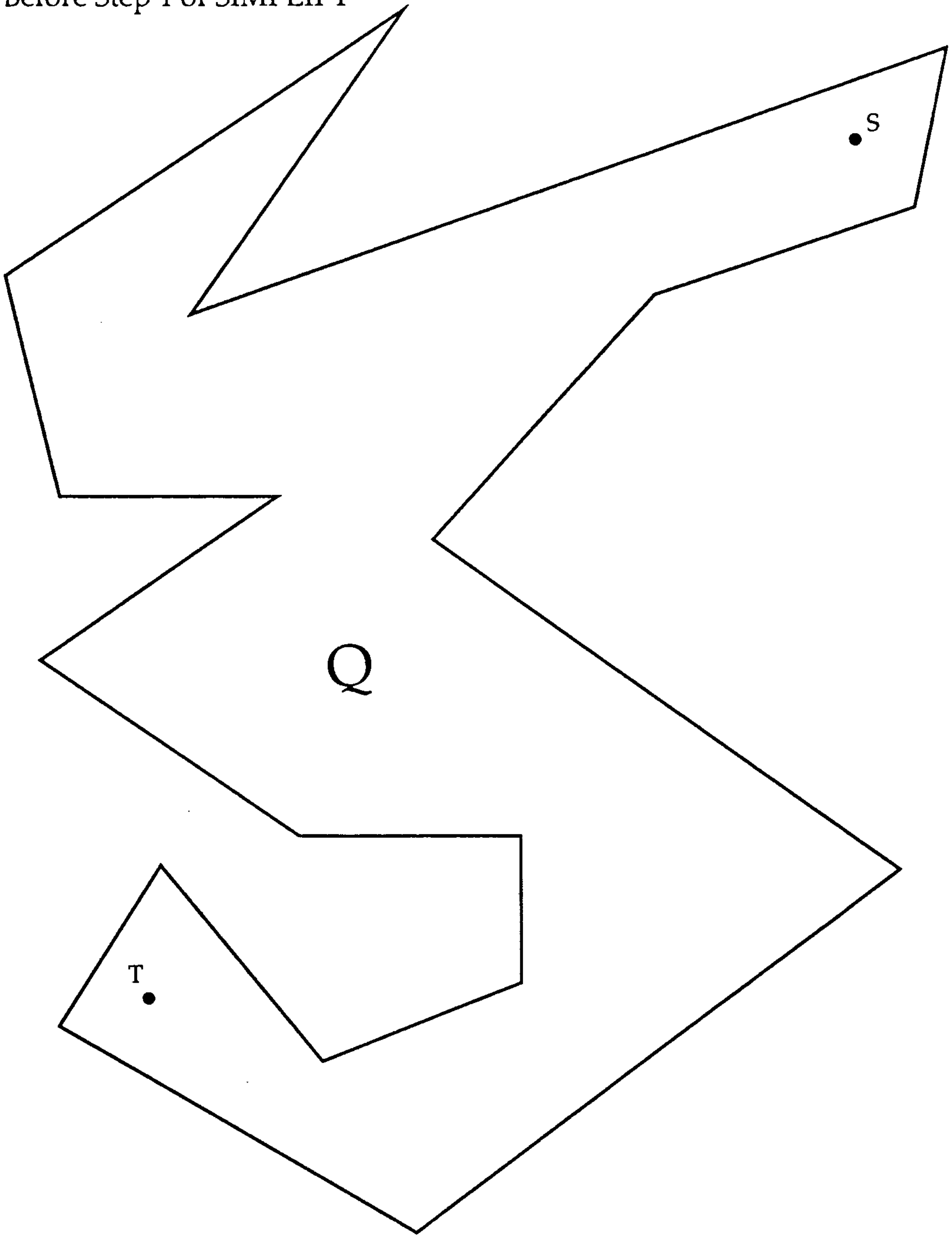
Conclusion: By induction, we have shown that traversal numbers provide a valid method of determining if a point is inside or outside of a simple polygon Q, where Q has any number of sides.

Appendix II: An Example

This appendix contains a series of illustrations showing the major constructs and information found at various stages in the processing of a polygon by SIMPLIFY. The data structures and data links used in the processing are for the most part not shown.

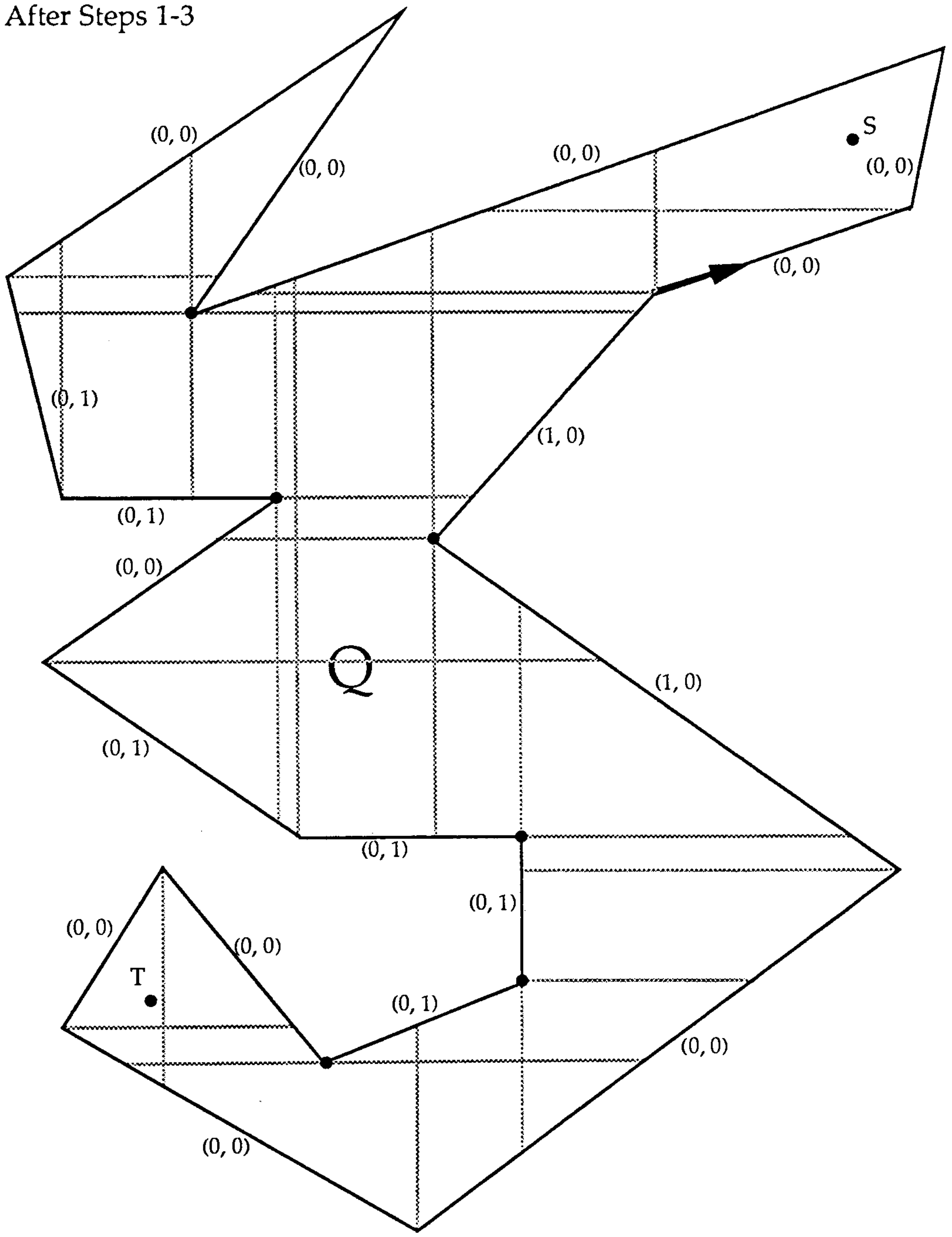
The illustrations begin on the following page.

Before Step 1 of SIMPLIFY



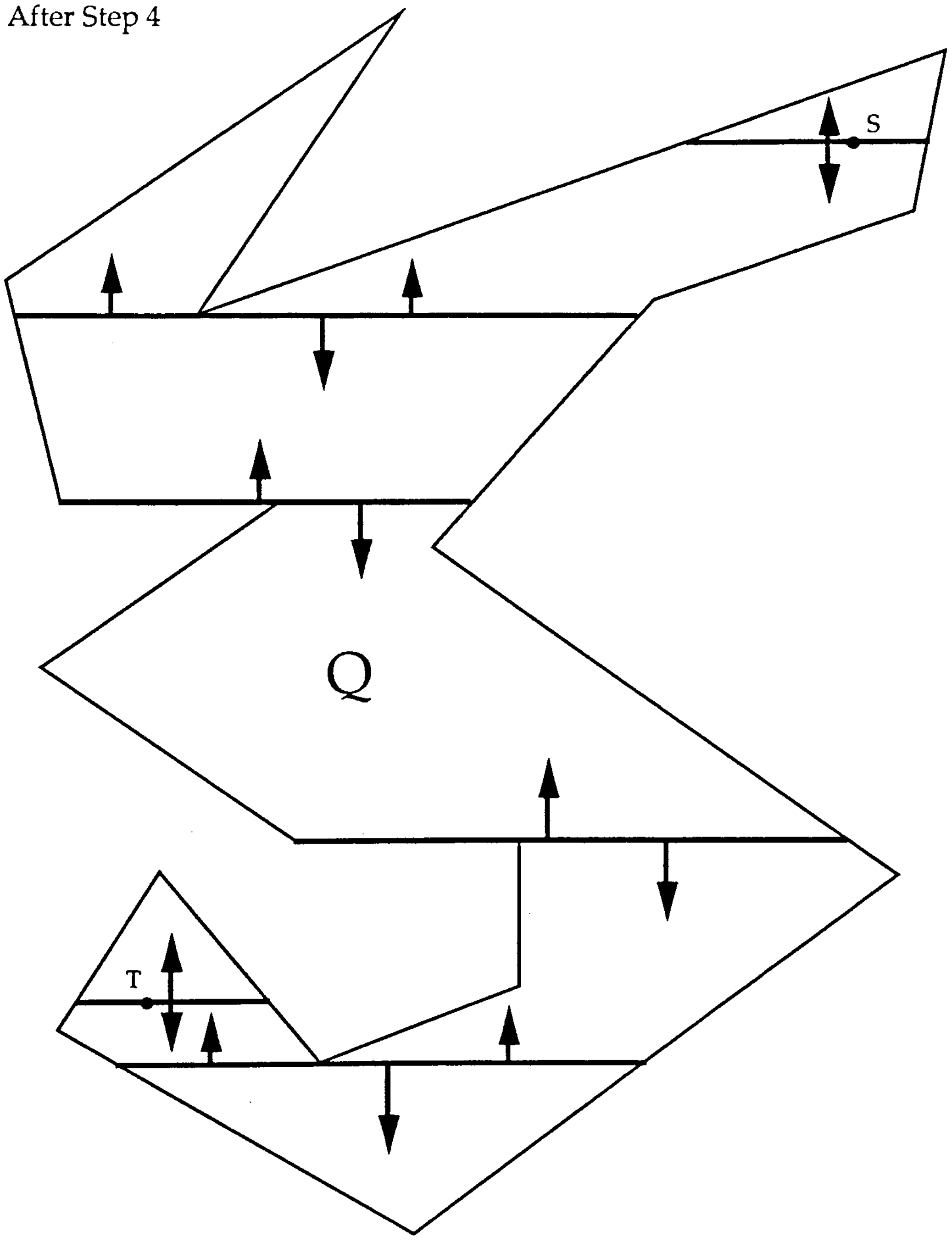
We wish to find a smallest path from S to T in Q.

After Steps 1-3



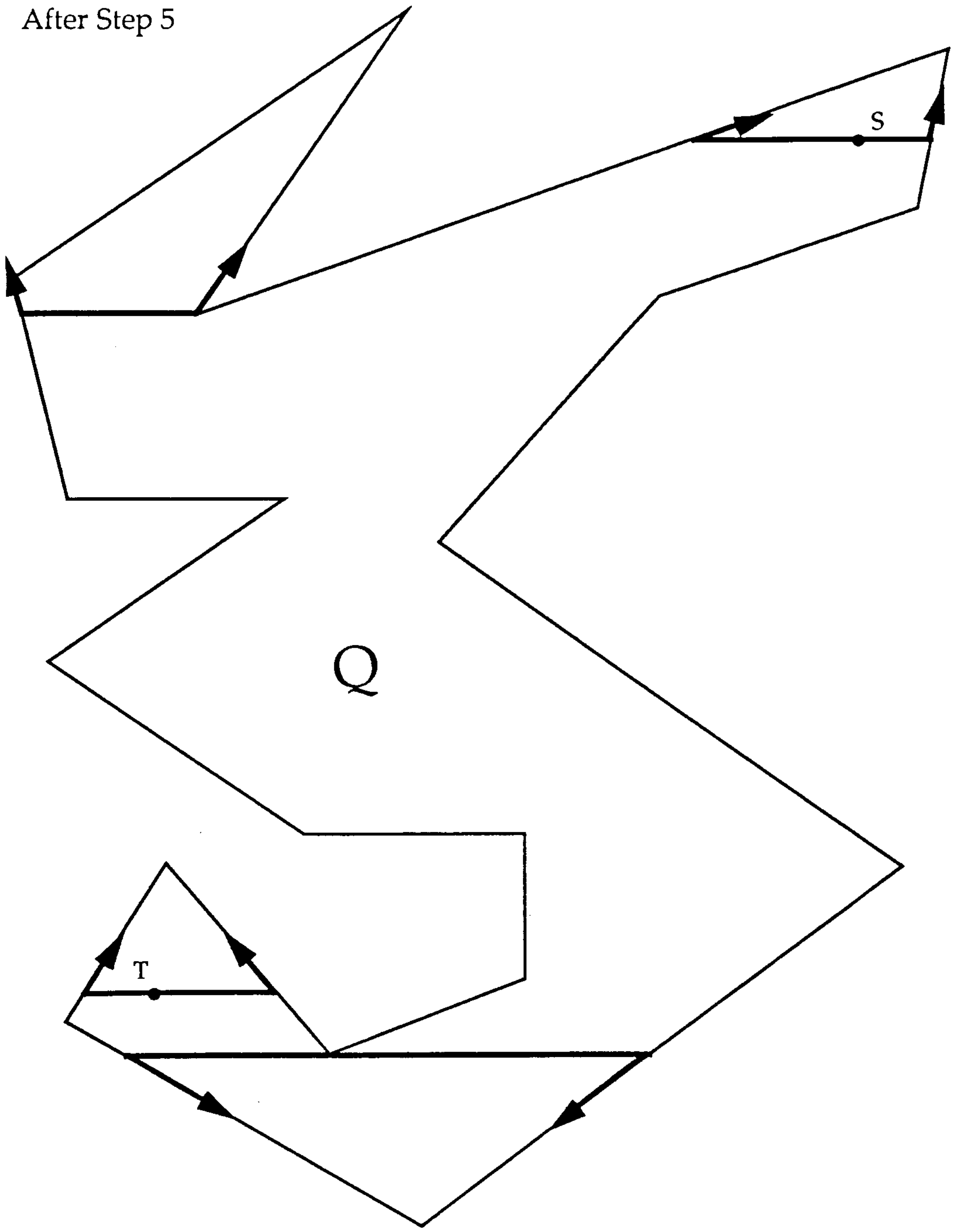
Traversal numbers (for (S, T), given by a traversal from the arrow in the direction shown), extreme points, and trapezoidal edges, as found by the first three steps of SIMPLIFY. The traversal numbers are low because the spirality of Q is purposely low. Most of the trapezoidal edges will not be used by SIMPLIFY. Most of the information given in this diagram will be deleted in later diagrams, for the sake of clarity.

After Step 4



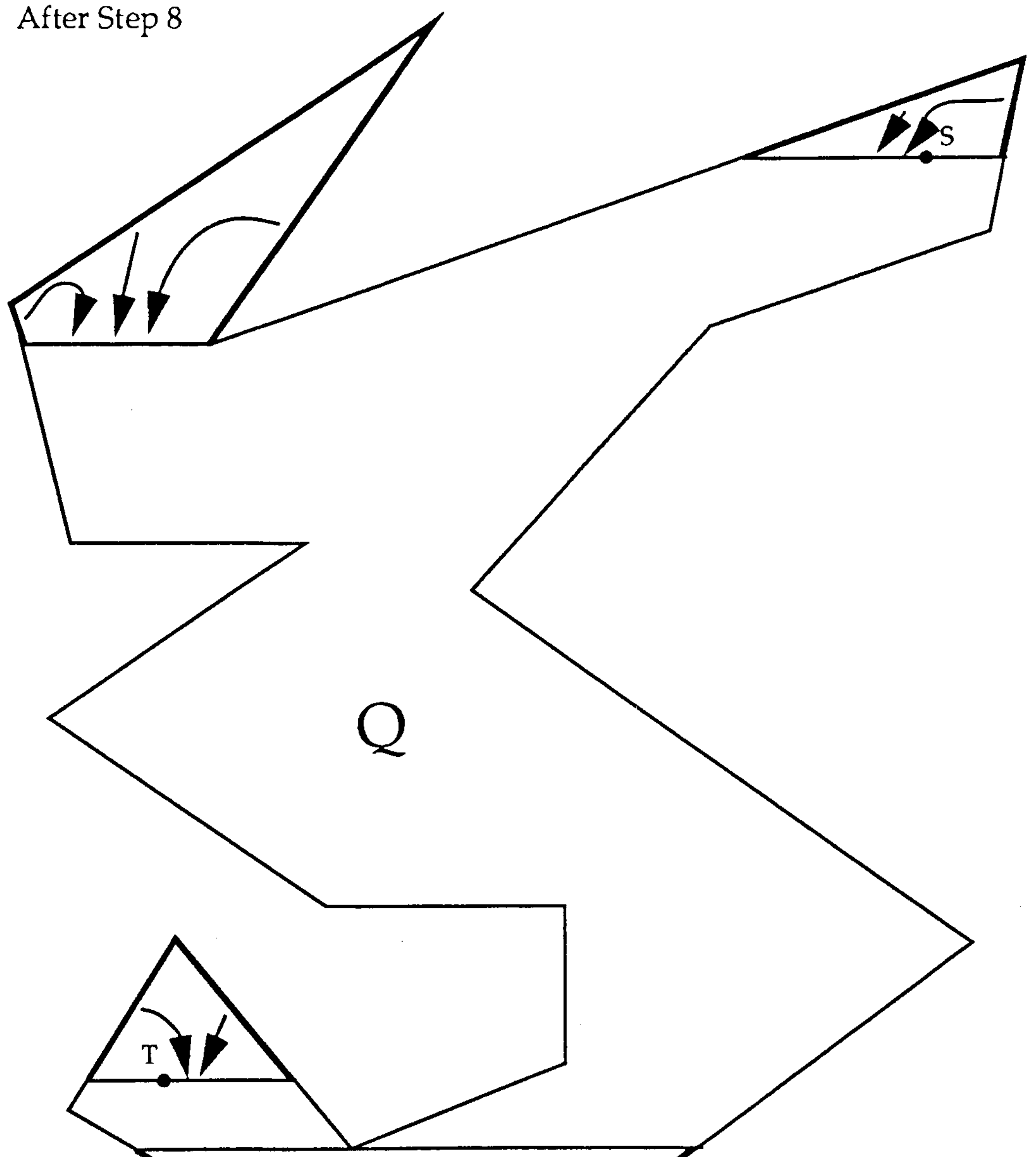
The thick lines are horizontal pseudochords, as found by step 4 of SIMPLIFY. Many of them overlap or are contained in others, so some endpoints are not discernible. Arrows from the centre of each pseudochord point into the region(s) induced by that pseudochord.

After Step 5



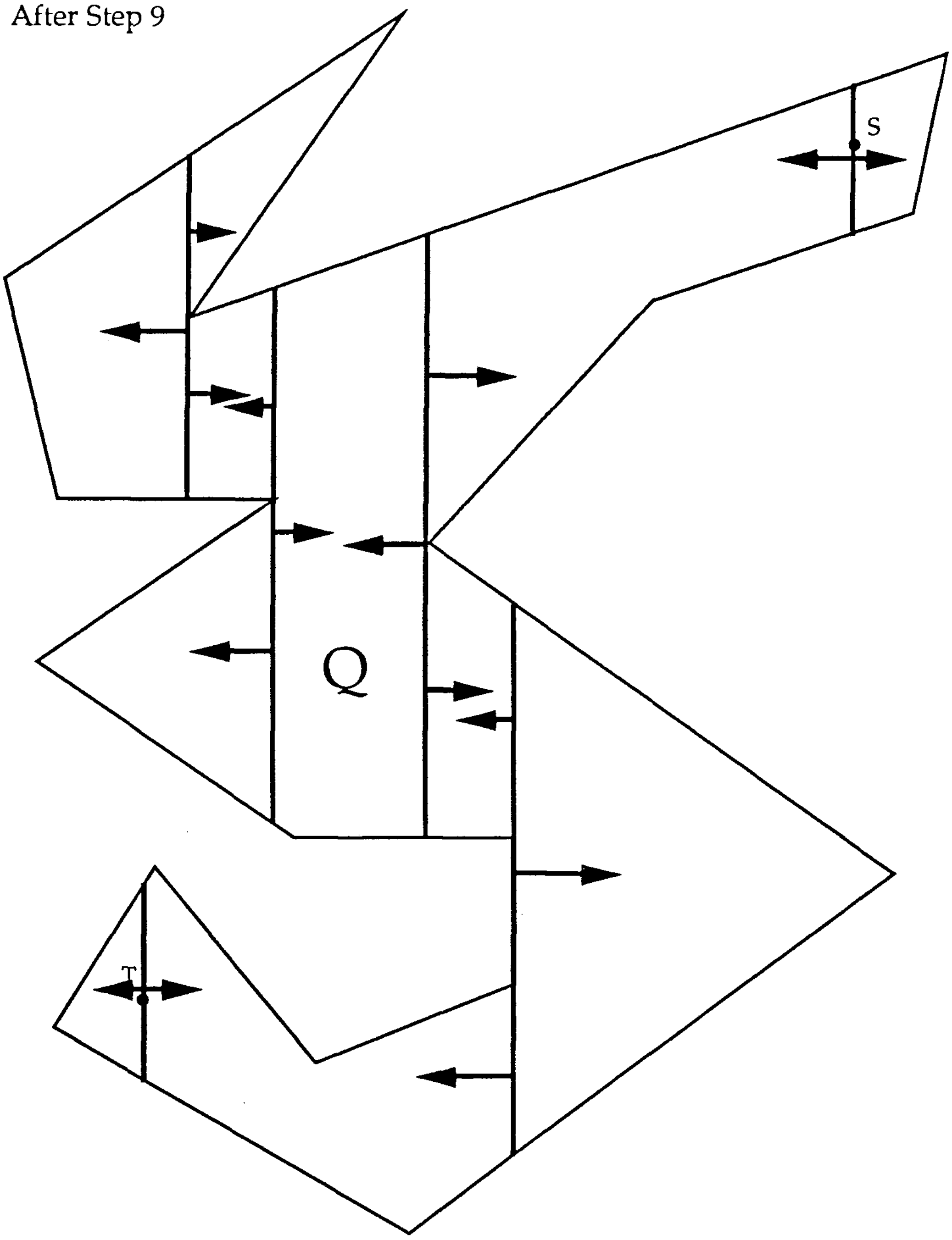
Step 5 of SIMPLIFY marks the endpoints of ebd(R), for unnecessary regions R induced by horizontal pseudochords.

After Step 8



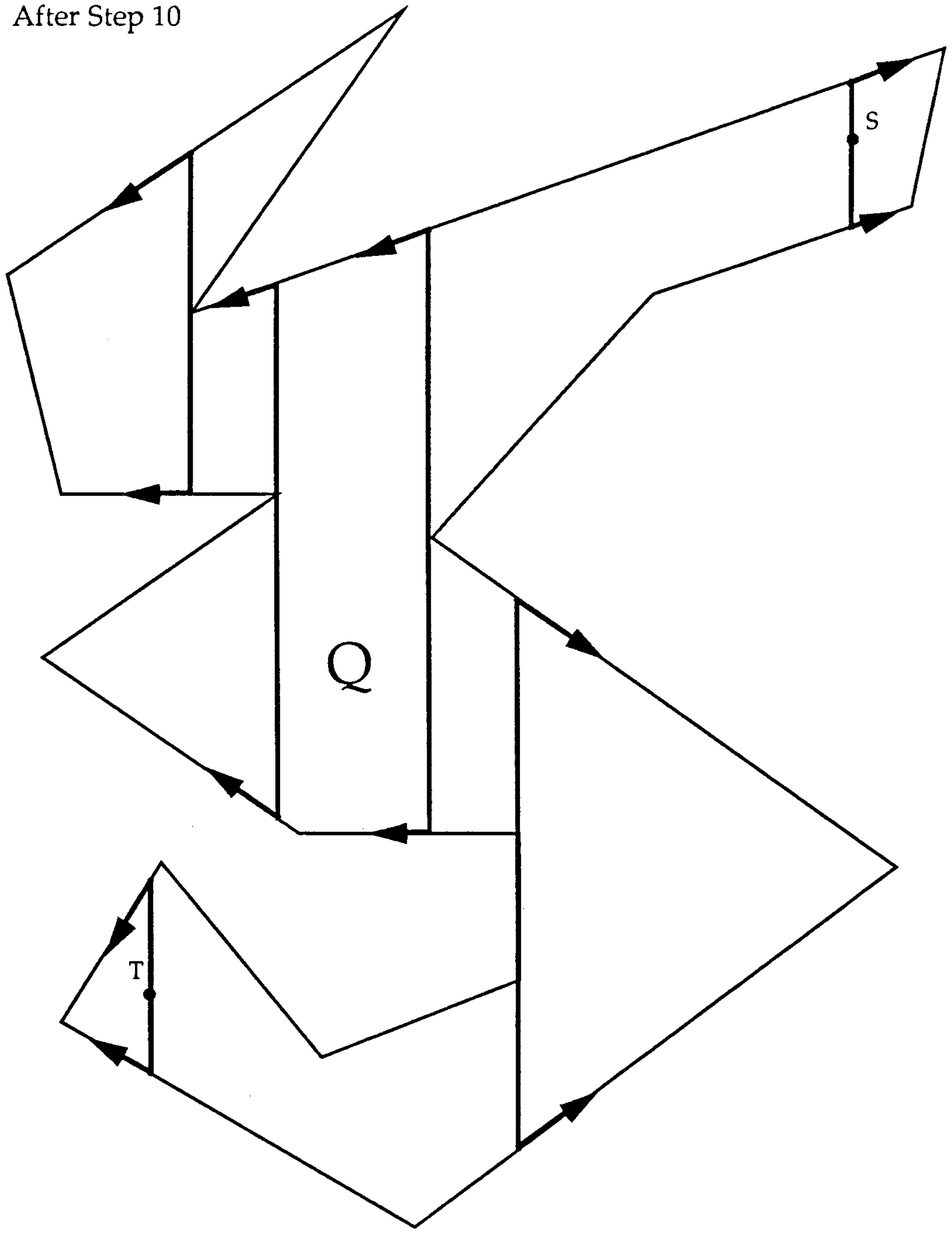
Steps 6 marks those sections of bdy(Q) which are in an unnecessary region induced by a horizontal pseudochord, as shown by the thick lines. Step 7 remove redundant horizontal pseudochords. For each subsegment B of bdy(Q) which forms part of the boundary of an unnecessary region R, step 8 marks the horizontal pseudochord inducing R, as shown by the arrows. Because there are no redundant horizontal pseudochords in this example, these three steps are collapsed into one diagram—see their counterparts in the vertical processing phase for more detail.

After Step 9



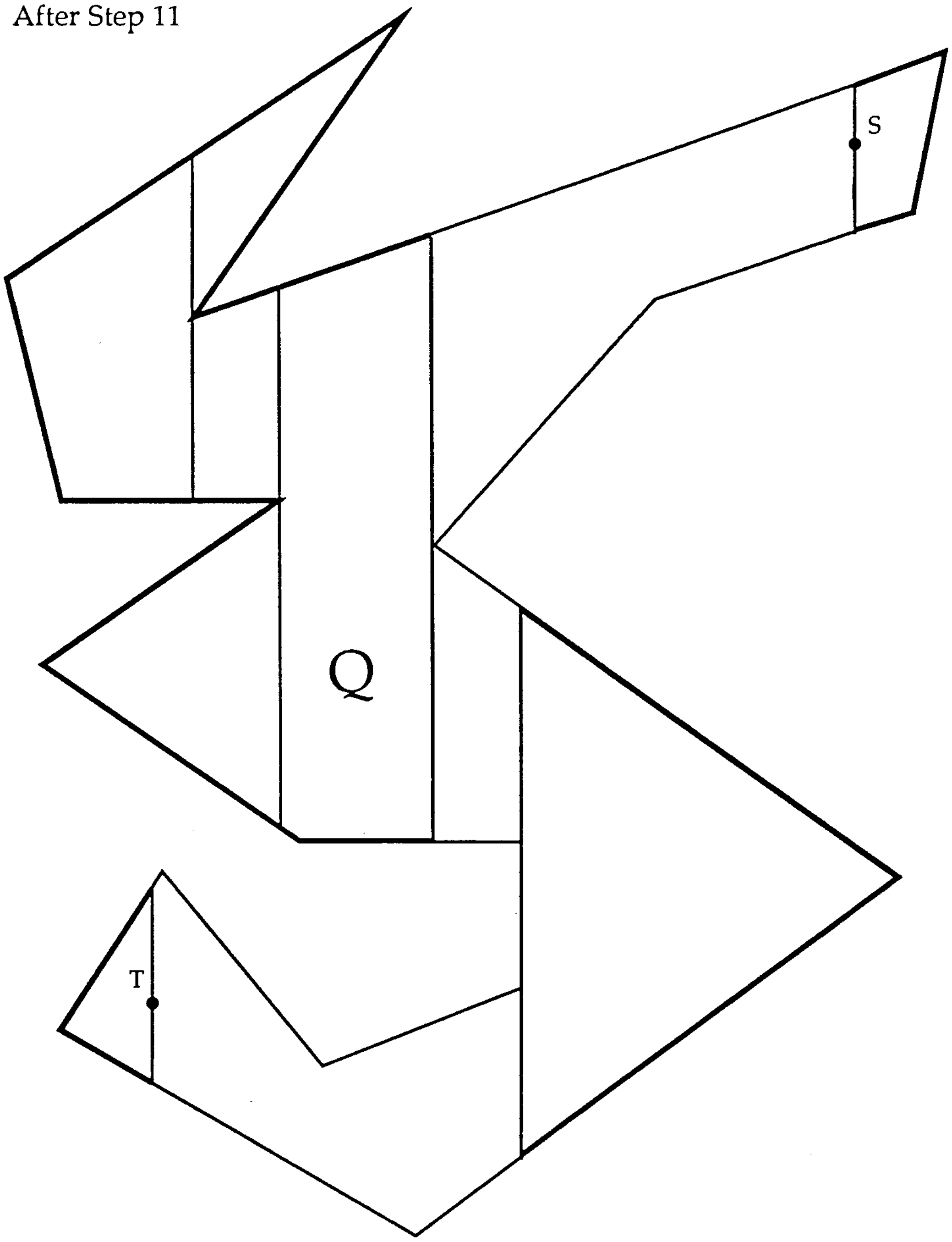
The thick lines are vertical pseudochords, as found by step 9 of SIMPLIFY. Arrows from the centre of each pseudochord point into the region(s) induced by that pseudochord. Because the pseudochords overlap or contain other pseudochords, not all endpoints are visible.

After Step 10



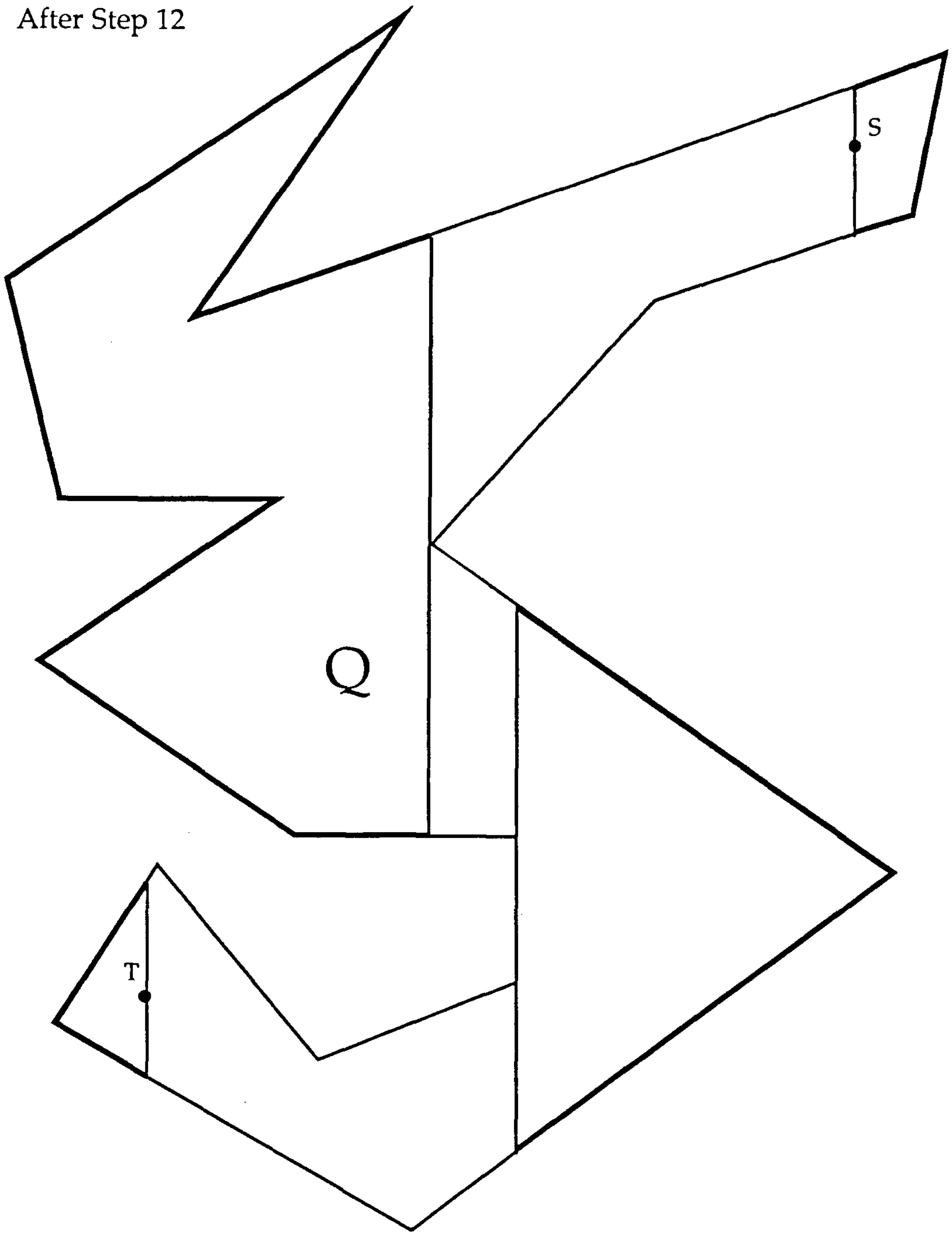
Step 10 marks the ends of exterior boundaries for unnecessary regions R induced by vertical pseudochords.

After Step 11



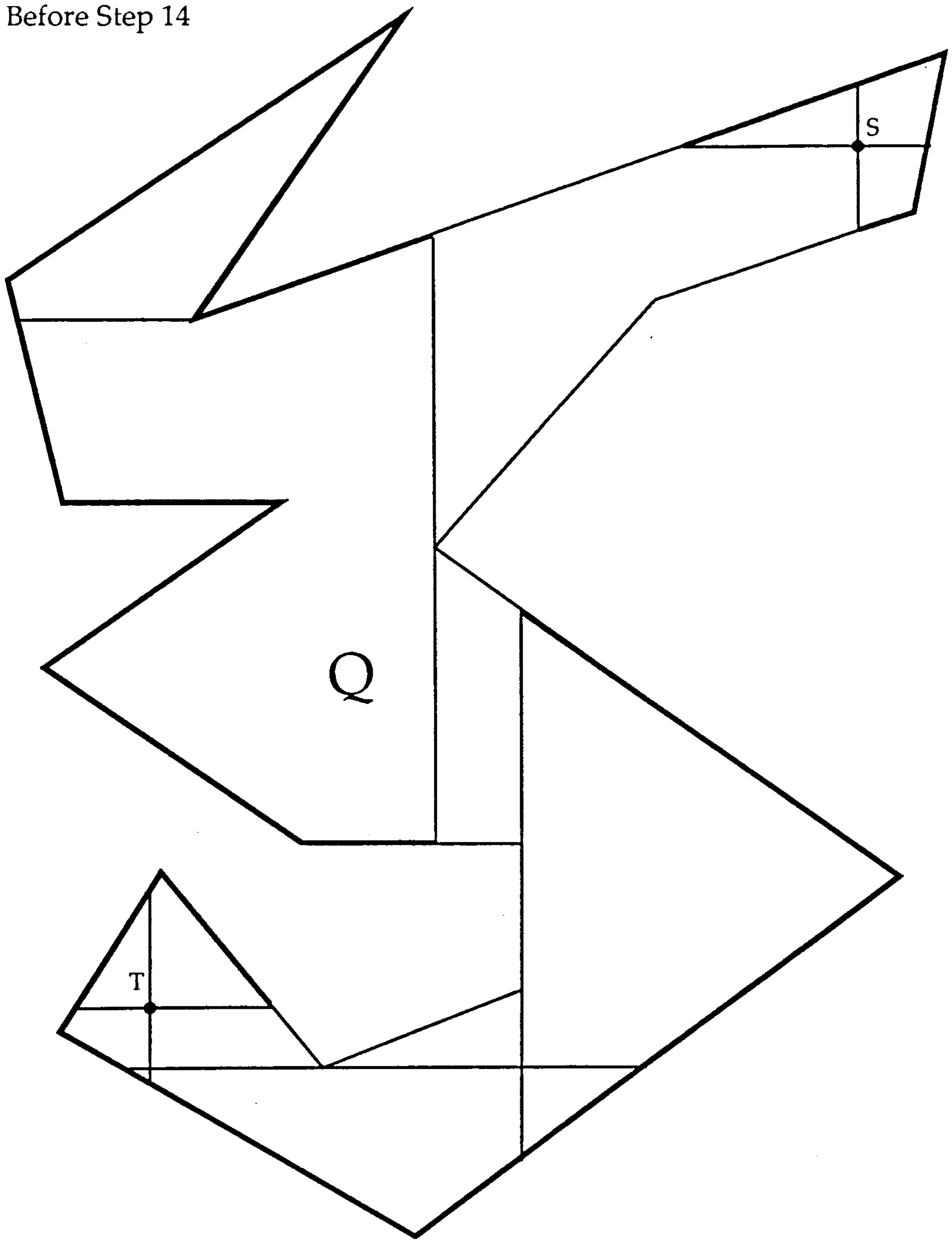
Step 11 marks all portions of bdy(Q) in unnecessary regions induced by vertical pseudochords—marked sections of bdy(Q) are shown by thick lines.

After Step 12



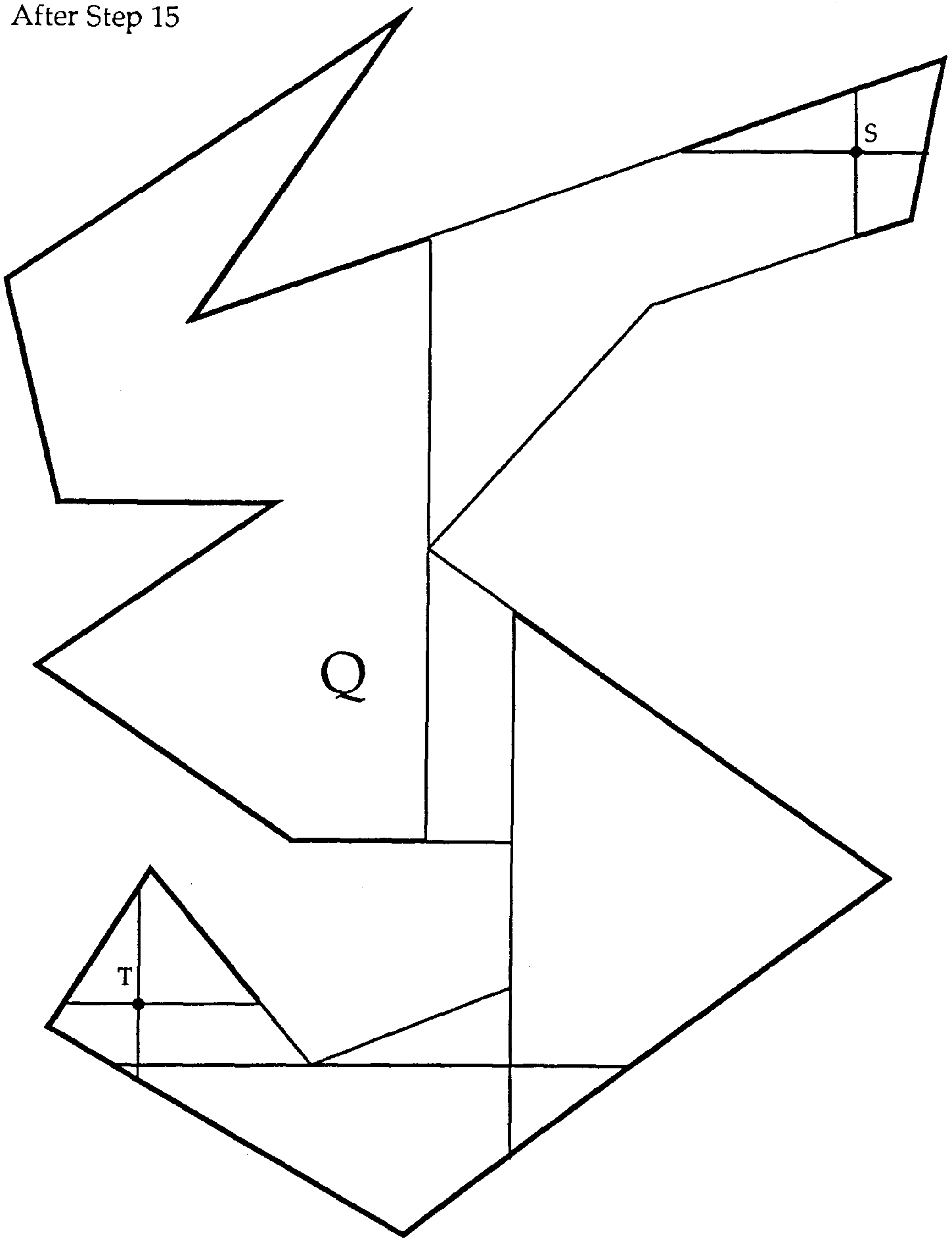
Step 12 removes vertical pseudochords entirely contained in an unnecessary region induced by other vertical pseudochords. Step 13, marking which vertical pseudochords own which sections of unnecessary parts of bdy(Q), will not be shown, as it is clear from the above.

Before Step 14



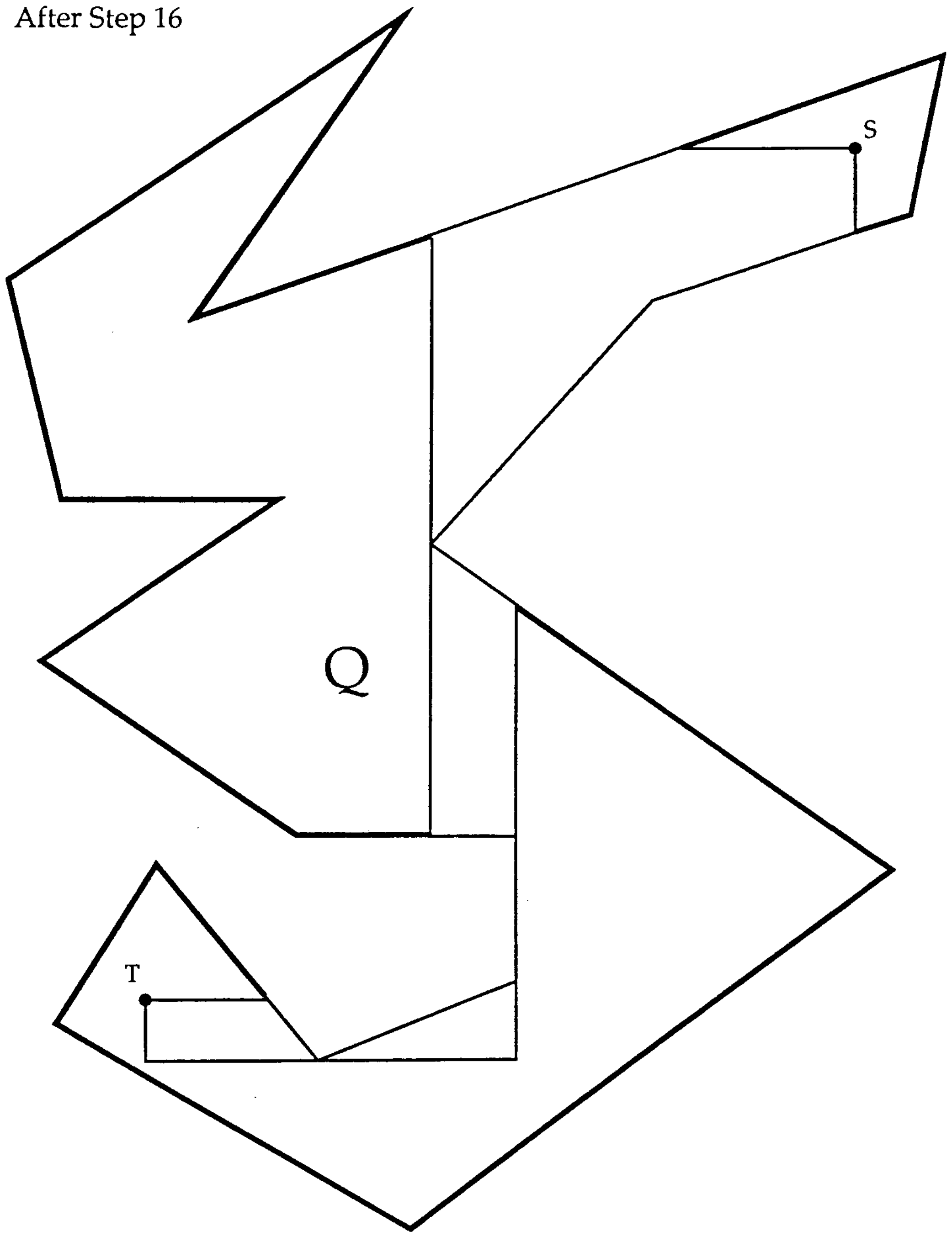
This is the state of Q after step 13 and before execution of steps 14 and 15. All remaining pseudochords are shown, as are those parts of bdy(Q) marked in previous steps as being part of a pseudochord-induced unnecessary region.

After Step 15



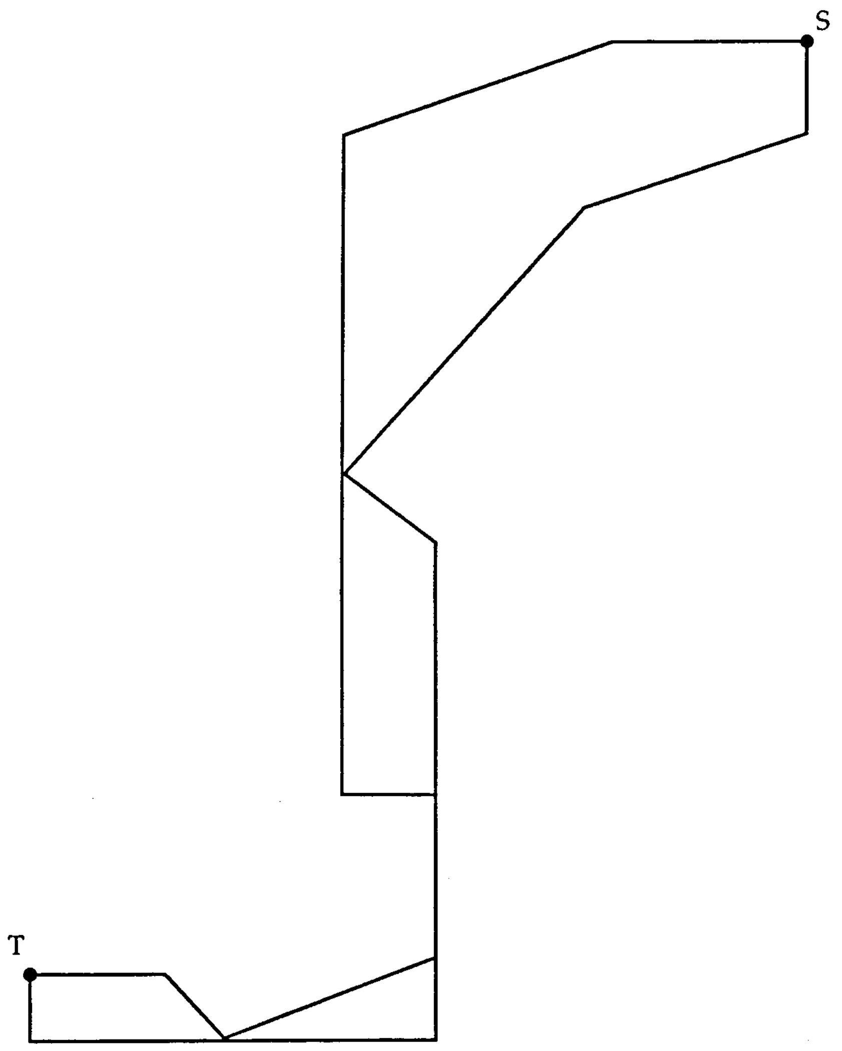
Q immediately after execution of steps 14 and 15. Horizontal pseudochords contained in unnecessary regions pseudochord-induced by vertical pseudochord have been removed, and vertical pseudochords in pseudochord-induced unnecessary regions induced by horizontal pseudochords have been removed. In this example, only a single pseudochord has actually been removed as a result of this process. Segments of bdy(Q) in unnecessary regions are shown by thick lines.

After Step 16



After execution of step 16, the ends of the remaining pseudochords have been adjusted so they no longer cross into unnecessary regions pseudochord-induced by pseudochords with a different orientation.

After Step 17



At the end of step 17, those portions of bdy (Q) in an unnecessary region of Q (shown by the thick lines in the previous illustration) have been removed, and the remaining sections of bdy(Q) and the remaining sections of pseudochords have been organized into the pseudogon.

Appendix III: The Parallel Prefix Operation

Parallel prefix is simply an operation to find, in parallel O(log n) time using n/log n processors, the n initial partial sums of an n-element vector. However, its great utility in distributing information through the elements of a vector may not at first be apparent to readers unfamiliar with the operation. This appendix gives an example of parallel prefix used in this manner.

We represent vectors as series of integers bounded by square brackets: [1 2 3 4 5]. We name the parallel prefix function *pre*. As a simple example of what we mean by the initial partial sums, pre[3 5 3 7 4] = [3 8 11 18 22].

pre can distribute information across the elements of a vector. For instance, suppose, 1 represents a left parentheses and -1 represents a right parentheses. A series of -1's, 1's, and 0's can be used to represent a series of (non-nested) parentheses and place-holders (representing concrete data in another vector, perhaps.) For example, "(a b c)(d e)(f g h i)" might be represented as the two vectors

V1= [1 0 0 0 -1 1 0 0 -1 1 0 0 0 0 -1] and

V2=[0 a b c 0 0 d e 0 0 f g h i 0].

To find out from the vector representation if f is enclosed in the same set of parentheses as h, we first modify V1 to V1′ by replacing every -1 with a 0 (which takes constant time with n processors):

V1′= [1 0 0 0 0 1 0 0 0 1 0 0 0 0 0],

take

V1″= pre(V1′) = [1 1 1 1 1 2 2 2 2 3 3 3 3 3 3]

and check to see if the element of V1″ corresponding to f in V2 is the same as the element of V1″ corresponding to h in V2; if these elements are the same, then f and h are in the same set of parentheses.

References

[1] P.J. de Rezende, D.T. Lee, and Y.F. Yu, "Rectilinear Shortest Paths with Rectangular Barriers.", *Proc. 1st ACM Symp. on Computational Geometry*, 1985, pp. 204-213.

[2] Y.F. Wu, P. Widmayer, D.F. Schlag, and C.K. Wong, "Rectilinear Shortest Paths and Minimum Spanning Trees in the Presence of Rectilinear Obstacles." *IIEEE Trans. Comput.* C-36 (1987), pp. 321-331.

[3] K. Clarkson, S. Kapoor, and P. Vaidya, "Rectilinear Shortest Paths through Polygonal Obstacles in O($\log^2$ n) Time," *Proc. 3rd ACM Symp. on Computational Geometry*,1987, pp. 251-257.

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

[5] T. Asano, "Generalized Manhattan Path Algorithm with Applications," *IEEE Trans. Computer-Aided Design* 7 (1988), pp. 797-804.

[6] S.H. Whitesides, "Computational Geometry and Motion Planning." in *Computational Geometry*, (G.T. Toussaint, Ed.), Elsevier Science Publishers B.V. (North-Holland), 1985, pp. 377-427.

[7] W. Lipski, ""Finding a Manhattan Path and Related Problems," *Networks* 13 (1983), pp. 399-409.

[8] W. Lipski, "An O(n log n) Manhattan Path Algorithm," *Inform. Process. Lett.* 19 (1984), pp. 99-102.

[9] T. Asano, M. Sato, and T. Ohtsuki, "Computational Geometry Algorithms," in *Layout Design and Verification*, (T. Ohtsuki, Ed.), Advances in CAD for VLSI 4, North-Holland, 1986, pp. 295-347.

[10] S. Suri, "A Linear Time Algorithm for Minimum Link Paths Inside a Simple Polygon," *Comp. Vision, Graphics, and Image Proc.* 35(1986), pp. 99-110.

[11] Y. Ke, "An Efficient Algorithm for Link-Distance Problems," Proc. 5th ACM Symp. on Computational Geometry, 1989, pp. 69-78.

[12] T. Ohtsuki, and M. Sato, "Gridless Routers for Two-Layers Interconnection," *Proc. IEEE Int. Conf. on Computer Aided Design*, 1984, pp. 76-79.

[13] B. Chazelle and J. Incerpi, "Triangulating a Polygon by Divide-and-Conquer," *Proc. of 21st Allerton Conf. on Comm. Control, and Comp.*, 1983, pp. 447-456.

[14] R. Cole, "Parallel Merge Sort," *Siam J. Comput.*, Vol. 17, No. 4, Aug. 1988, pp. 770-785.

[15] Richard M. Karp and Vijaya Ramachandran, "A Survey of Parallel Algorithms for Shared-Memory Machines," Univ. of Calif. at Berkeley Comp. Sci. Dept. Report #UCB/CSD 88/408, March 1988.

[16] M. J. Atallah, and M. T. Goodrich, "Efficient Plane Sweeping in Parallel, Preliminary Version," *Proc. 2nd ACM Symp. on Computational Geometry*, 1986, pp. 216-225.

[17] M.J. Atallah, R. Cole, M.T. Goodrich, "Cascading Divide and Conquer: A Technique for Designing Parallel Algorithms," *Proc. 28th FOCS*, 1987, pp. 151-160.

References (continued)

[18] Y. Shiloach and U. Vishkin, "Finding the Maximum, Merging, and Sorting in a Parallel Computation Model," *Journal of Algorithms* 2(1981), pp. 88-102.

[19] A. Borodin and J.E. Hopcroft, "Routing, Merging, and Sorting on Parallel Models of Computation," *Jour. of Comp. and Sys. Sci.*, Vol. 30, No. 1, Feb. 1985, pp. 130-145

[20] L. Valiant, "Parallelism in Comparison Problems," *SIAM Jour. on Comp.*, Vol. 4, No. 3, Sept. 1975, 348-355.