LINEAR TIME ALGORITHMS FOR VISIBILITY AND SHORTEST PATH PROBLEMS INSIDE SIMPLE POLYGONS

Leo Guibas
John Hershberger
Daniel Leven
Micha Sharir
Robert E. Tarjan

CS-TR-039-86

May 1986

LINEAR TIME ALGORITHMS FOR VISIBILITY AND SHORTEST PATH PROBLEMS INSIDE SIMPLE POLYGONS

Leo Guibas[1,2], *John Hershberger*[1], *Daniel Leven*[3], *Micha Sharir*[3],[4], *Robert E. Tarjan*[5],[6]

[1] Computer Science Department

Stanford University

Stanford, CA 94305

[2] DEC/SRC

130 Lytton Ave.

Palo Alto, CA 94301

[3] School of Mathematical Sciences,

Tel-Aviv University

Tel Aviv 69978, ISRAEL

[4] Courant Institute of Mathematical Sciences

New York University

NY, NY 10012

[5] AT&T Bell Laboratories

Murray Hill, NJ 07974

(6) Department of Computer Science

Princeton University

Princeton, NJ 08544

ABSTRACT

We present linear time algorithms for solving the following problems involving a simple planar polygon P: (i) Computing the collection of all shortest paths inside P from a given source vertex s to all the other vertices of P; (ii) Computing the subpolygon of P consisting of points that are visible from a segment within P; (iii) Preprocessing P so that for any query ray r emerging from some fixed edge e of P, we can find in logarithmic time the first intersection of r with the boundary of P; (iv) Preprocessing P so that for any query point x in P, we can find in logarithmic time the portion of the edge e that is visible from x; (v) Preprocessing P so that for any query point x inside P and direction u, we can find in logarithmic time the first point on the boundary of P hit by the ray at direction u from x; (vi) Calculating a hierarchical decomposition of P into smaller polygons by recursive polygon cutting, as in [Ch]. (vii) Calculating the (clockwise and counterclockwise) "convex ropes" (in the terminology of [PS]) from a fixed vertex s of P lying on its convex hull, to all other vertices of P. All these algorithms are based on a recent linear time algorithm of Tarjan and Van Wyk for triangulating a simple polygon, but use additional techniques to make all subsequent phases of these algorithms also linear.

1. Introduction

Recently Tarjan and Van Wyk [TV] have developed a linear-time algorithm for triangulating simple polygons, thereby improving the previous $O(n \log n)$ algorithm of [GJPT], and solving a major open problem in computational geometry. This result has extended in a significant way the

Work on this paper by the fourth author has been supported by Office of Naval Research Grant N00014-82-K-0381, National Science Foundation Grant No. NSF-DCR-83-20085, by grants from the Digital Equipment Corporation, and the IBM Corporation, and by a Grant from the U.S-Israeli Binational Science Foundation.

list of problems already known to be solvable in linear time on simple polygons, which has included e.g. calculation of the convex hull of such a polygon [GY], [MA], calculation of the subpolygon of P visible from a given point [Le], [EA], and more. In addition, there are many problems known to be linear-time equivalent to the triangulation problem for simple polygons (for a list of these see e.g. [FM]) that are now therefore also solvable in linear time. Also, several other problems on simple polygons were given linear time solutions, provided that a triangulation of the given polygon is already available, and are thus now also solvable in linear time. These problems include calculation of the shortest path inside a simple polygon between two specified points [LP], preprocessing a simple polygon to support logarithmic-time point location queries [Ki], [EGS], stationing guards in simple art galleries [Fi], etc.

In this paper we continue the exploration for linear-time algorithms for simple polygons. We present several new such algorithms, which are all based on the availability of a triangulation of the given polygon, but exploit additional new techniques to achieve the linear-time goal.

Our new linear time algorithms solve the following problems for a given simple polygon P with n sides.

(1) Given a fixed source point X inside P, calculate the shortest paths inside P from X to all vertices of P (in fact our algorithm even provides a (linear time) preprocessing of P into a data-structure from which the length of the shortest path inside P from X to any desired target point Y can be found in time $O(\log n)$; the path itself can be found in time

$O(\log n + k)$, where k is the number of segments along this path).

(2) Given a fixed edge e of P, calculate the subpolygon $Vis(P,e)$ consisting of all points in P visible from (some point on) e.

(3) Given e as above, preprocess P so that, given any query ray r emanating from e into P, the first point on the boundary of P hit by r can be found in $O(\log n)$ time.

(4) Given e as above, preprocess P so that, given any point X inside P, the subsegment of e visible from X can be computed in $O(\log n)$ time.

(5) Preprocess P so that, given any point X inside P and direction u, the first point $hit(X,u)$ on the boundary of P hit by the ray at direction u from X can be computed in $O(\log n)$ time.

(6) Calculate a hierarchical balanced decomposition tree of P by recursively cutting P along diagonals, as in [Ch].

(7) Given a vertex X of P lying on its convex hull, calculate for all other vertices Y of P the clockwise and counterclockwise *convex ropes* around P from X to Y, when such paths exist (these are polygonal paths in the exterior of P from X to Y that wrap around P, always turning in a clockwise (resp. counterclockwise) direction; cf. Section 5 for more detail).

Our results improve previous algorithms given for some of these problems (cf. [Ch], [CG], [PS]). Most of our algorithms are based on the solution to Problem (1), and exploit interesting relationships between visibility and shortest-path problems for a simple polygon. Our technique for

solving Problem (1) extends the technique of Lee and Preparata [LP] for calculating the shortest path inside P between a single pair of points, and uses *finger trees,* a data-structure for efficient access to an ordered list when there is locality of reference (see Guibas, McCreight, Plass and Roberts [GMPR]) and Huddleston and Mehlhorn [HM]), to obtain an overall linear-time performance.

The paper is organized as follows. In Section 2 we present the linear-time solution to Problem (1). The visibility problems (2) - (4) are then solved in Section 3. The polygon cutting procedure and the shooting problem (Problems (5)-(6)) are solved in Section 4, and the convex rope algorithm for Problem (7) is presented in Section 5.

2. Calculating the Shortest Path Tree of a Simple Polygon

Let P be a simple polygon having n vertices, and let s be a given *source vertex* of P. (Actually, our algorithm will also apply, with some minor modifications, in case s is an arbitrary point interior to, or on the boundary of P. For the sake of exposition, the algorithm below is described for s a vertex of P, and we later comment on the modifications required to handle the case of an arbitrary source s.) Denote, for each vertex v of P, the Euclidean shortest path from s to v inside P by $\pi(s, v)$. It is well known (see e.g. [LP]) that $\pi(s, v)$ is a polygonal path whose corners are vertices of P, and that $\bigcup_v \pi(s, v)$, taken over all vertices v of P, is a planar tree $Q_s(P)$ (rooted at s), which we call the *shortest path tree* of P (with respect to s); this tree has altogether n nodes, namely the vertices of P, and its edges are

straight segments connecting these nodes. Our goal is to calculate this tree in linear time.

Let G be a triangulation of the interior of P (which can be computed in $O(n)$ time, using Tarjan and Van Wyk's algorithm [TV]). The planar dual T of G (whose vertices are the triangles in G and whose edges join two such triangles if they share an edge) is a tree, each of whose vertices has degree at most 3. Thus, for each vertex t of P, there is a unique minimal path π in T from some triangle containing s to another triangle containing t, which induces an ordered sequence of diagonals $d_1, d_2, \ldots, d_l$ of P (to be more precise, d_1 should be chosen as the first diagonal between two adjacent triangles in π that does not terminate at s, and d_l should be chosen as the last such diagonal not not terminating at t; this takes care of situations in which s or t is a vertex of more than one triangle in G). Each diagonal d_i thus divides P into two parts containing s and t respectively and therefore $\pi(s,t)$ must intersect only diagonals d_i, and each of them exactly once.

Let $d = uw$ be a diagonal or an edge of P and let a be the least common ancestor of u and w in the shortest path tree $Q_s(P)$. It is shown in [LP] that $\pi(a,u)$, $\pi(a,w)$ are both *outward-convex;* i.e. the convex hull of each of these subpaths lies outside the region bounded by $\pi(a,u)$, $\pi(a,w)$, and by the segment uw. Following [LP], we call the union $F = F_{uw} = \pi(a,u) \cup \pi(a,w)$ the *funnel* associated with $d=uw$, and a the *cusp* of that funnel. Suppose next that d is a diagonal of P used by G, and let Δuwx be the unique triangle in G having d as an edge that does not intersect the area bounded between F and d. Then the shortest path from s to x must start with $\pi(s,a)$ and then either

continue along the straight segment ax if this segment does not intersect F, or else proceed along either $\pi(a,u)$ or $\pi(a,w)$ to a vertex v such that vx is a tangent to F at v, and then continue along the straight segment vx (see Fig 2.1). These observations form the basis of the algorithm of Lee and Preparata [LP], and of ours.

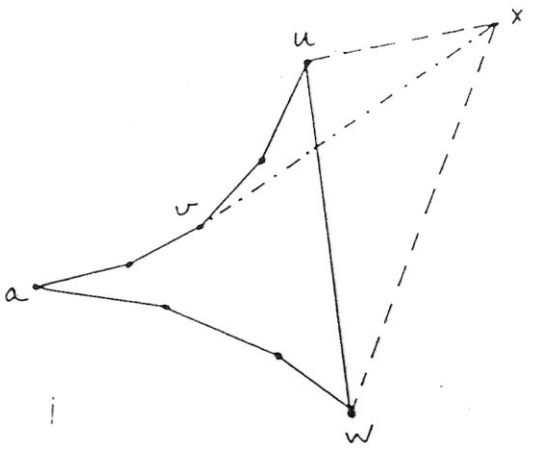

Fig. 2.1

We are now ready to describe our algorithm. We first triangulate P in $O(n)$ time (as in [TV]). As the algorithm comes to process a diagonal $d = uw$ of P, it maintains the current funnel $F = F_{uw}$ as a sorted list $[u_l, u_{l-1}, \ldots, a, w_1, \ldots, w_k]$, where $a = u_0 = w_0$ is the cusp of F, $\pi(a,u) = [u_0, \ldots, u_l]$, $\pi(a,w) = [w_0, \ldots, w_k]$ (either of these sublists can be empty), and $u_l = u$, $w_k = w$. This list is stored in a *finger tree* (cf. [GMPR], [HM]). This structure is essentially a search tree equipped with *fingers* (which, in our application, are always placed at the first element and at the last element of the tree). This structure supports searching for an element x in time $O(\log \delta)$, where δ is the distance from x to the nearest

finger, and also supports operations that split the tree into two subtrees at an element x in amortized time $O(\log \delta)$, with δ as above. The algorithm also maintains a pointer $CUSP(F)$ to the cusp a of the present funnel.

The algorithm begins by placing s and an adjacent vertex v_1 in F, with $CUSP(F) = s$. It then proceeds recursively as follows.

ALGORITHM PATH(F)

Let u and w be the first and the last elements of F, and let $a = CUSP(F)$ (thus $F = \pi(a,u) \cup \pi(a,w)$). Let Δuwx be the unique triangle in the triangulation G of P that has uw as an edge and that has not yet been processed (cf. Fig. 2.1).

(a) Search F for an element v, for which vx is a tangent to F at v (if the straight line segment ax does not intersect F then $v = a$). It is easy to check that each (unsuccessful) "comparison" performed at some node v^* during this search determines a unique side of v^* in which the desired v lies, so that the binary search paradigm is applicable in this case. We then split $F \cup \{x\}$ into two new funnels $F_1 = [u, \ldots, v, x]$ and $F_2 = [x, v, \ldots, w]$. If v belongs to $\pi(a,u)$ then we set $CUSP(F_1) := v$, $CUSP(F_2) := a$. If, on the other hand, v belongs to $\pi(a,w)$ then $CUSP(F_1) := a$, $CUSP(F_2) := v$.

(b) Set $\pi(s,x) := \pi(s,v) \cup vx$ (Actually we just store a back pointer from x to v. The collection of all these pointers will constitute the required shortest path tree $Q_s(P)$).

(c) If the line segment ux is a diagonal of P then we call recursively

PATH(F_1).

(d) If the line segment wx is a diagonal of P then we call recursively PATH(F_2).

The main difference between our algorithm and the algorithm of Lee and Preparata [LP] is in the techniques for representation, searching and splitting of funnels. In [LP] the search for the vertex v is a linear search starting at one designated endpoint of F. This is sufficient to guarantee the linearity of their procedure since in their case the vertices of D that are scanned during the search for v are no longer required, as the algorithm always continues with only one of the funnels F_1 or F_2 (depending on whether the next diagonal to be crossed is xw or xu). However in our case the algorithm may have to continue recursively with both funnels and thus requires a funnel searching and splitting strategy that uses finger trees (and is thus subtler than the simple linear list representation used in [LP]), to obtain the desired linear time complexity.

The correctness of our algorithm is a direct consequence of the correctness of the algorithm of Lee and Preparata. To bound the time required by the algorithm we argue as follows. Let T be the dual tree of the triangulation of P. Using Euler's formula for planar maps it is easily checked that T has $n-2$ nodes. Without loss of generality, suppose s lies in just one triangle τ_0 of T, which we take to be the root of T. (If s is a vertex of P which lies in several triangles of G, then at least one of them will be bounded by an edge of P incident to s, and we can start the algorithm from that triangle. It is easily checked that the algorithm will then propagate correctly

the funnel structure to all the other triangles containing s; note that the funnels for (the edges of) these triangles are all trivial.) Thus each node of T (including the root) has 0, 1, or 2 children. Clearly, our algorithm is essentially a depth-first traversal of T. With each node ζ of T we associate two parameters:

m_ζ - the size (i.e. number of edges) of the funnel F at the time ζ is being processed.

n_ζ - the number of edges of P that bound triangles in the subtree of T rooted at ζ.

When our algorithm processes the node ζ of T, it splits its funnel into two parts and then appends a new edge to both parts to form the funnels of the children ζ_1, ζ_2 of ζ in T. If the split parts of the funnel of ζ contain m_1 and $m_2 = m_\zeta - m_1$ edges respectively, then $m_{\zeta_1} = m_1 + 1$, $m_{\zeta_2} = m_2 + 1$, and the (amortized) cost of processing ζ using finger trees is $K(\zeta) = O(\min(\log m_{\zeta_1}, \log m_{\zeta_2}))$. Note also that if ζ has two children ζ_1, ζ_2 then $n_{\zeta_1} + n_{\zeta_2} = n_\zeta$ and $n_{\zeta_1}, n_{\zeta_2} \geq 1$. If ζ has just one child ζ' then $n_{\zeta'} = n_\zeta - 1$, and if ζ is a leaf then $n_\zeta = 2$. The complexity of our algorithm essentially depends only on the growth of the function m_ζ over the nodes $\zeta \in T$. This function grows by at most 1 when decending from a node ζ, having just one child ζ', to ζ'; if ζ has two children ζ_1, ζ_2 then m_ζ is split into two parts, and each child inherits one part plus 1.

We begin our analysis with the following observation: The "direct costs" $K(\zeta)$ at nodes $\zeta \in T$ which have just one child or are leaves, sum up to at

most $O(n)$. Indeed, suppose $\zeta \in T$ has just one child ζ', and that the funnel at ζ had been split up to two parts having m' and $m_\zeta - m'$ edges respectively. Then the vertices of P lying in one of those parts will never be encountered again by the algorithm (by the same reasoning used in [LP] to justify the linearity of their procedure). The number of such vertices is at least $\min(m', m_\zeta - m') - 1 \geq K(\zeta) - 2$. Thus the sum of all these $K(\zeta)$ is proportional to at most $n + 2|T| \leq 3n$, as claimed.

Next consider the total direct costs at nodes having two children. We claim that it is sufficient to consider only cases in which m_ζ grows exactly by 1 at each node of T having a single child, because these cases provide maximal growth of the function m down the tree T. Under this additional assumption, we have:

Lemma 2.1: Let $\zeta \in T$, and let the leaves of the subtree T_ζ of T rooted at ζ be $\eta_1, \ldots, \eta_k$. We then have

$$\sum_{j=1}^{k} m_{\eta_j} = m_\zeta + |T_\zeta|$$

where $|T_\zeta|$ is the number of edges in T_ζ.

Proof: An immediate consequence of the transitive closure of the relations

$$\sum_{j=1}^{t} m_{\zeta_j} = m_\zeta + t$$

where $\{\zeta_j\}_{j=1}^{t}$ are the children of ζ ($t = 0$, 1, or 2). □

Corollary: In the same notations we have

$$m_\zeta \leq \sum_{j=1}^{k} m_{\eta_j}$$

Using these notations, we define $M_\zeta = \sum_{j=1}^{k} m_{\eta_j} \geq m_\zeta$. For each $\zeta \in T$ let $C(\zeta)$ denote the total cost of processing nodes with two children in the subtree of T rooted at ζ. Then plainly

$$C(\zeta) = \begin{cases} 0 & \text{if } \zeta \text{ is a leaf} \\ C(\zeta') & \text{if } \zeta \text{ has just one child } \zeta' \\ C(\zeta_1) + C(\zeta_2) + O(\min(\log m_{\zeta_1}, \log m_{\zeta_2})) & \text{if } \zeta \text{ has two children } \zeta_1, \zeta_2 \end{cases}$$

In solving these recurrence formulas, we can clearly assume without loss of generality that each node in T is either a leaf or has two children. Moreover, replacing m_ζ by M_ζ in these formulas, we obtain the recurrence formula

$$C(\zeta) = \begin{cases} 0 & \text{if } \zeta \text{ is a leaf} \\ C(\zeta_1) + C(\zeta_2) + O(\min(\log M_{\zeta_1}, \log M_{\zeta_2})) & \text{if } \zeta \text{ has two children } \zeta_1, \zeta_2 \end{cases}$$

But $M_\zeta = M_{\zeta_1} + M_{\zeta_2}$, if ζ has children ζ_1, ζ_2 and $M_\zeta \geq 1$ for all nodes ζ. Hence if $C^*(k)$ is the maximal cost $C(\zeta)$ for any node ζ with $M_\zeta = k$, then we obtain the formula

$$C^*(m) = \max_{1 \leq k \leq m-1} \{C^*(k) + C^*(m-k) + O(\min(\log k, \log (m-k)))\}$$

whose solution is $C^*(m) = O(m)$ (cf. [Me, p. 185]). Finally, by Lemma 2.1 we have for the root τ_0 of T

$$M_{\tau_0} = m_{\tau_0} + |T| = n - 1$$

Thus the total complexity of the algorithm is

$$O(n) + O(M_{\tau_0}) = O(n).$$

Summing up our analysis, we obtain

Theorem 2.1: The shortest paths inside a simple polygon P from a fixed source vertex to all the other vertices of P can all be calculated in linear time.

Remark: Although finger trees are not too complicated to implement, we could have obtained a simpler, and only slightly less efficient version of the above procedure by maintaining funnels simply as doubly-linked linear lists (the same data structure as that used in [LP]), and by performing each search through a funnel in a linear manner, starting simultaneously from both endpoints of the funnel. The complexity analysis of this modified procedure is almost identical to that given above, except that the direct costs at each triangle processed are now linear, rather than logarithmic, in the subfunnel sizes. This leads to a recurrence formula for C^* of the form

$$C^*(m) = \max_{1 \le k \le m-1} \{C^*(k) + C^*(m - k) + O(\min(k,m-k))\}$$

whose solution is (cf. [GK, pp. 25 - 27]) $C^*(n) = O(n \log n)$.

Remark: If the source s is not a vertex of P, we can modify the algorithm as follows. Suppose s is internal to a single triangle $\Delta = \Delta uvw$ of G. Then we can split Δ into three subtriangles Δsuv, Δsvw, Δswu, all having s as a vertex, and repeat the algorithm three times, each time starting at one of these triangles, and propagating the funnel structure only through the edge of that triangle which is also an edge of Δ. Similar problem splitting can be employed when s lies on an edge of some of the triangles in G. It is easily checked that the modified algorithm also produces the desired shortest path tree in overall linear time.

An extended algorithm

The algorithm described above can be extended to produce additional information regarding shortest paths from the source point s to arbitrary

points inside P. We describe such an extension that produces in linear time a partitioning of P into $O(n)$ disjoint triangular regions, such that each region consists of all points X, the shortest paths to which all pass through the same sequence of vertices of P. To do this, we first need the following lemma.

Lemma 2.2: For each edge e of P, let $\Phi(e)$ denote the region bounded by e and by the funnel F_e. Then

(a) Let x be a point inside such a region $\Phi(e)$. Let v be a vertex in the corresponding funnel such that vx is tangent to the funnel (in the terminology of the algorithm described above). Then the shortest path from s to x is the concatenation of the shortest path from s to v with the segment vx.

(b) The interiors of the regions $\Phi(e)$ are all disjoint, and the total number of edges along their boundaries is $O(n)$.

Proof: The first part of the lemma follows by the same argument (taken from [LP]) used to justify the correctness of our algorithm. As to the second claim, note first that if $\Phi(e_1)$ and $\Phi(e_2)$ had a point x in common, then x would necessarily have two distinct shortest paths reaching it from s (one for each region Φ containing x), which is impossible for a simple polygon. Furthermore, since the funnels constituting the boundaries of the regions $\Phi(e)$ are outward convex, it follows that each pair of such regions can have at most one edge in common. Since the number of these regions is n, it follows by Euler's formula that the total number of their edges is also $O(n)$.

□

Let e be an edge of P, and let $\Phi(e)$ be the corresponding region of P.

Assume that the funnel F_e has the form $[u_l,u_{l-1}, \ldots ,u_1,a,w_1, \ldots ,w_k]$ with a as its cusp (thus $e = u_l w_k$). Denote a also as u_0 and as w_0. Then, for each $i=0, \ldots ,l-1$ (resp. for each $i=0, \ldots ,k-1$) the ray emanating from u_i (resp. from w_i) and passing through u_{i+1} (resp. through w_{i+1}) hits e, and its portion between e and u_i (resp. w_i) is fully contained in $\Phi(e)$. These rays partition $\Phi(e)$ into $k+l-1$ disjoint triangles, such that each triangle has two vertices lying on e and its third vertex (called its *apex*) belongs to the funnel F_e. Moreover, it follows immediately from Lemma 2.2(a) that if $x \in \Phi(e)$ belongs to the triangle with apex q then qx is tangent to the funnel at q, and thus the shortest path from s to x is the concatenation of the shortest path from s to q and the segment qx.

Hence the collection of all triangles obtained this way for all regions $\Phi(e)$ yields a partitioning of P into disjoint triangles, whose total number, by Lemma 2.2(b), is $O(n)$. We can then use any one of the linear-time algorithms of [Ki] or of [EGS] to preprocess this partitioning into a data structure that supports $O(\log n)$-time point location queries. The preceding argument implies that for each target point x in P we can find in $O(\log n)$ time the last vertex q of P on the shortest path from s to x. Thus, if we store at each such vertex q the length of the shortest path from s to q, we can then calculate the length of the shortest path from s to x in additional $O(1)$ time; the path itself can be calculated in additional $O(k)$ time, by simply traversing the path in the shortest-path tree from q to s. To sum up, we have

Theorem 2.2: Given a simple polygon P with n sides, and some source point s within P, one can preprocess P in linear time, such that, for each query

target point x in P, the length of the shortest path from s to x can be calculated in $O(\log n)$ time, and the path itself can be calculated in time $O(\log n + k)$, where k is the number of segments from which this path is composed.

Remark: An alternative technique for calculating the shortest path tree of a simple polygon from a given source point in linear time has been independently obtained by El Gindy [EG2].

3. Visibility Within a Simple Polygon

In this section we study a collection of problems involving visibility within a simple polygon. These problems have been studied in various recent papers [EA], [Le], [AT], [CG], [EG], [LL], [As], and a variety of algorithms have been developed to solve them. Some of the simpler problems already have linear time solutions, whereas others have been given $O(n \log n)$ solutions. Here we present linear-time solutions for all these problems, again using the linear-time triangulation algorithm of [TV], and an interesting relationship between visibility and shortest path problems.

Let P be a simple polygon with n sides. The problems studied in this section, and solved in linear time, are

I. Given a point x inside P, calculate the *visibility polygon* $Vis(P,x)$ consisting of all points $y \in P$ that are *visible* from x (i.e. such that the segment xy is fully contained within P). (This problem is simpler than the subsequent ones, and in fact there exist known linear-time algorithms for it [EA], [Le].)

II. Given a segment e inside P, calculate the (weak) visibility polygon $Vis(P,e)$ consisting of all $y \in P$ that are visible from some point on e. (An $O(n \log n)$ solution is given in [CG]; Avis and Toussaint [AT] present a linear time algorithm for determining whether $Vis(P,e) = P$.)

III. Given such a segment e, preprocess P so that for each query ray r emanating from some point on e into P, the first intersection of r with the boundary of P can be calculated in $O(\log n)$ time. (Again, an $O(n \log n)$ solution is given in [CG].)

IV. As in III, preprocess P so that for each query point $x \in P$, the subsegment of e visible from x can be calculated in $O(\log n)$ time. (An $O(n \log n)$ solution is given in [CG].)

Consider first Problem I; although this problem already has linear time solutions, it is still worthwhile to sketch our own linear time algorithm for this problem as a preparatory step toward the solution of the more complicated problems II-IV. It is well known that $Vis(P,x)$ is a simple polygon; its vertices are either vertices of P that are visible from x or are "shadows" cast on the boundary of P by such visible vertices (these are points y visible from x such that the segment xy passes through a vertex of P; cf. Fig. 3.1).

Suppose without loss of generality that x is a vertex of P (if not, it is easy to construct in linear time a triangulation of the interior of P in which x is also a vertex of some triangles). Calculate the shortest path tree T from x using the (extended) algorithm given in Section 2. Then clearly the vertices of

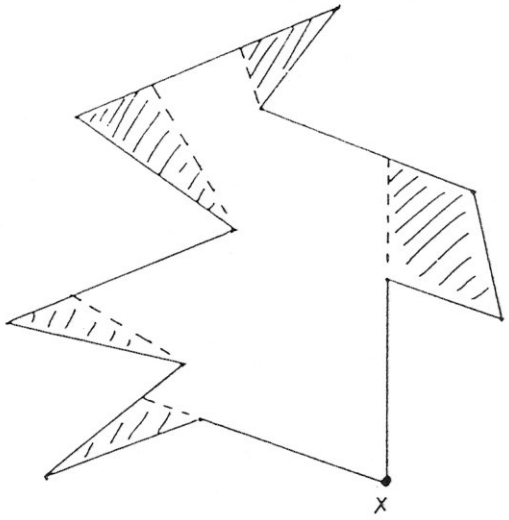

Fig. 3.1. Visibility of a polygon from a point.

P visible from x are the direct children of x in T. Moreover, the extended shortest-path algorithm partitions the boundary of P into $O(n)$ disjoint segments, and the shadows cast on the boundary of P by the visible vertices are simply the endpoints of those segments that are visible from x (i.e. segments e for which the shortest paths from x to points on e are straight segments). These observations enable us to calculate $Vis(P,x)$ by simply traversing the boundary of P in, say clockwise order, collecting all visible subsegments along this boundary, and replacing contiguous non-visible portions of the boundary by straight segments connecting visible vertices with their shadows. Thus we have

Theorem 3.1: The visibility polygon $Vis(P,x)$ can be calculated in linear time.

Next consider Problem II. Let A, B be the endpoints of e. It is easily checked that in this case $Vis(P,e)$ is a simple polygon whose vertices are either

(i) vertices of P visible from e; or

(ii) shadows cast on the boundary of P by rays that emanate from A or from B and pass through a vertex of P visible from that endpoint; or

(iii) shadows cast by rays r that emanate from some interior point on e and pass through two vertices x, y of P, such that the complement of P lies on one side of r in the vicinity of x, and on the other side of r in the vicinity of y.

See Fig. 3.2 for an illustration of $Vis(P,e)$.

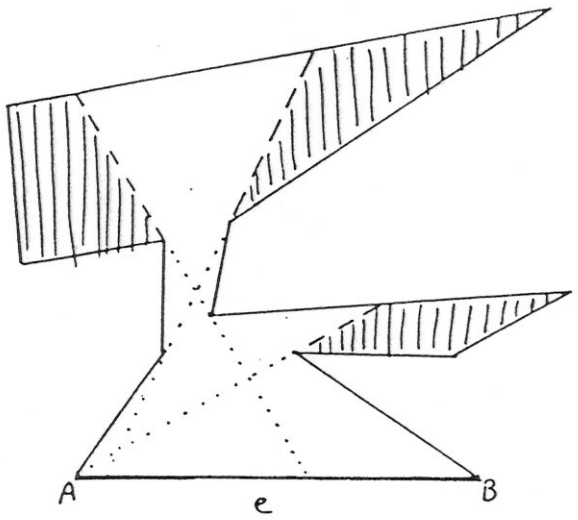

Fig. 3.2. Visibility of a polygon from an edge.

Let $e' = CD$ be another edge of P. Let $\pi(X,Y)$ denote the shortest path inside P between the two points X and Y. We say that $\pi(A,C)$ is *outward convex* if the convex angles formed by successive segments of this path with the directed line AB keep increasing. A symmetric definition of outward convexity applies to shortest paths from B.

Lemma 3.2: If e' contains a point in its relative interior that is visible from e then (up to exchanging C and D) the two paths $\pi(A,C)$ and $\pi(B,D)$ are both outward convex.

Proof: Let $x \in e'$ be visible from some point z on e. Suppose without loss of generality that C is that endpoint of e' for which A and C lie on the same side of the line xz. Then the shortest path $\pi(A,C)$ must lie entirely on one side of xz, and as a matter of fact it does not cross the polygonal path $AzxC$. Since the area R between $AzxC$ and $\pi(A,C)$ is fully contained in P, it follows that $\pi(A,C)$ must be outward convex, or else we could shortcut it by a segment contained in R, thus in P too. The claim concerning $\pi(B,D)$ now follows by a completely symmetric argument. □

In the case described by the preceding lemma, we call the union of $\pi(A,C)$ and $\pi(B,D)$ the *hourglass* for the pair (e,e').

Apply the shortest-path algorithm of Section 2 to the two source vertices A and B, and also compute on the fly, for each vertex C of P, whether the paths $\pi(A,C)$, $\pi(B,C)$ are outward convex, where these calculations take $O(1)$ time per vertex. Let $e' = CD$ be another edge of P. If the two paths $\pi(A,C)$ and $\pi(B,D)$ are not both outward convex, and also the two paths

$\pi(A,D)$ and $\pi(B,C)$ are not both outward convex, then by Lemma 3.2 e' is not visible from e. Thus suppose without loss of generality that the two paths $\pi(A,C)$ and $\pi(B,D)$ are outward convex. It is easy to see that the shortest path $\pi(A,D)$ must then be the concatenation of three subpaths: a subpath $\pi(A,X)$ of $\pi(A,C)$ (where X is some point lying on $\pi(A,C)$); a straight segment XY, where Y lies on $\pi(B,D)$; and the subpath $\pi(Y,D)$ of $\pi(B,D)$. Moreover XY must be a common tangent to both paths $\pi(A,C)$ and $\pi(B,D)$ (see Fig. 3.3). The path $\pi(B,C)$ has a symmetric structure of the form $\pi(B,W) \mid\mid WZ \mid\mid \pi(Z,C)$, for appropriate points $W \in \pi(B,D)$, $Z \in \pi(A,C)$. It now follows that the subsegment of e' visible from e is that which is delimited by the intersections of e' with the two lines XY and WZ. Note also that, in the terminology of Section 2, when the shortest-path algorithm is run with A as the source, the funnel $F_{e'}$ associated with the segment e' has X as its cusp, and Y as an adjacent vertex. Similarly, when the algorithm runs with B as the source, W is the cusp of the funnel $F_{e'}$ and Z is an adjacent vertex in that funnel.

These observations suggest a straightforward method for calculating the points X, Y, W and Z. That is, as we execute the shortest-paths algorithm with A as a source, and reach an edge $e' = CD$ of P for which the path $\pi(A,C)$ is outward convex, we simply take X to be the cusp of the current funnel, and Y to be the next vertex on that funnel on its portion between X and D. The points W and Z are then found in a completely symmetric manner when running the shortest path algorithm with B as a source. Hence, to calculate $Vis(P,e)$ we simply have to traverse the boundary of P in, say,

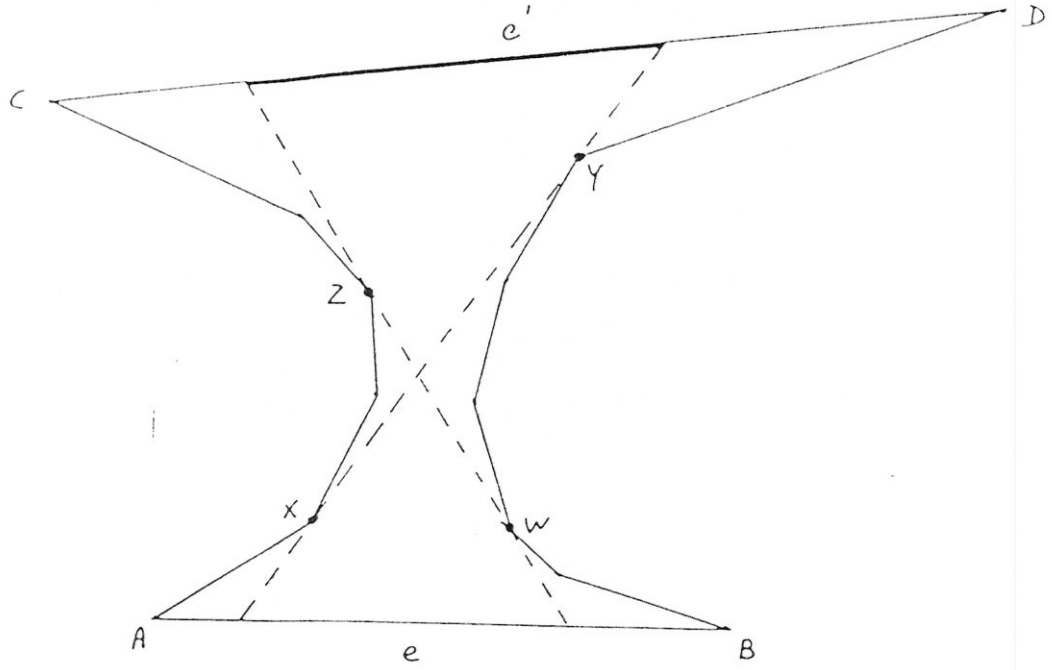

Fig. 3.3. Visibility of one edge of a polygon from another and the corresponding hourglass.

clockwise order, collecting all visible subsegments along this boundary in the manner explained above, and replacing contiguous non-visible portions of the boundary by straight segments connecting visible vertices with their appropriate shadows. Thus we have

Theorem 3.3: The visibility polygon $Vis(P,e)$ of P with respect to an edge e can be calculated in linear time.

Remark: A similar connection between shortest paths and visibility inside a simple polygon, and also some of the technical tools developed above, have also been obtained independently by Toussaint [To].

Next Consider Problem IV. Let $X \in P$ be an arbitrary point that is

visible from some point $z \in e$. An immediate generalization of Lemma 3.1 implies that both paths $\pi(A,X)$, $\pi(B,X)$ must be outward convex. The converse statement is also true, as is easily checked; that is, if both $\pi(A,X)$ and $\pi(B,X)$ are outward convex, then X is visible from e. Moreover, let the last straight segment in $\pi(A,X)$ (resp. in $\pi(B,X)$) be CX (resp. DX) for some vertex C (resp. D) of P. Then the portion of e visible from X is delimited by the intersections of the lines CX, DX with e.

Thus to preprocess P as required in Problem IV, we simply execute the (extended) shortest-path algorithm of Section 2 twice, once with A as a source, and once with B as a source. This yields two partitionings Π_1, Π_2 of P into zones of influence by the vertices of P, which we then further preprocess into two corresponding data structures that support $O(\log n)$ point location queries. This can be done in linear time, using the techniques in [Ki] or [EGS]. In addition, we store at each vertex C of P indications whether the paths $\pi(A,C)$, $\pi(B,C)$ are outward convex, and (as usual) also pointers to the two fathers of C in the two shortest-path trees produced by the two runs of the algorithm.

Now let $X \in P$ be a given query point. First locate X in the two partitions Π_1 and Π_2, and obtain the two corresponding influencing vertices C, D of P. Then, in constant time, we can test whether the paths $\pi(A,X) = \pi(A,C) \,||\, CX$, $\pi(B,X) = \pi(B,D) \,||\, DX$ are both outward convex, and if so, find the intersections of the lines CX, DX with e, to obtain the desired subsegment of e that is visible from X.

Remark: Problem IV is stronger than the corresponding problem P3 studied in [CG], in that here the query point X can be any point inside P, whereas in [CG] it is required to lie on the boundary of P.

Next consider Problem III. Here we make use of the duality between rays emanating from e and points in the *two-sided plane* (2SP for short), as described in [GRS], [CG]. Following [CG], we will produce a partitioning Π of the 2SP into convex regions, each region containing the duals of all rays emanating from e and hitting the same edge of P; this same partitioning is obtained in [CG] in $O(n \log n)$ time, and we show here how to obtain it in linear time.

To this end we make use of the analysis of Problem II given above. Let $e' = CD$ be another edge of P that contains points visible from e. As above, we can then assume that the two paths $\pi(A,C)$, $\pi(B,D)$ are both outward convex. Let XY, WZ be the two common tangents to these paths, where $X,Z \in \pi(A,C)$ and $W,Y \in \pi(B,D)$. Let $R(e')$ be the region in Π corresponding to e'. Then clearly the boundary of $R(e')$ consists of points that are duals of rays r emanating from e and hitting points on e', such that r passes through a vertex of P (which can be either one of the endpoints A,B,C,D, or another vertex of P that r "grazes" on its way from e to e'. It is clear from the preceding analysis that such a vertex must lie on one of the paths $\pi(A,C)$, $\pi(B,D)$, or, more precisely, on one of their subpaths $\pi(X,Z)$, $\pi(W,Y)$.

Moreover, it is also easily checked that the vertices of $R(e')$ correspond either to rays that pass through two vertices of P that are adjacent in one of

the subpaths $\pi(X,Z)$, $\pi(W,Y)$, or to the two extreme rays XY, WZ. It is also easy to establish adjacency of the vertices of $R(e')$ along its boundary. Specifically, two such vertices must correspond to two rays passing respectively through FG and GH, where F, G, H are three vertices of P that are either adjacent along one of the paths $\pi(X,Z)$, $\pi(W,Y)$, or are such that G is one of W,X,Y, or Z, F is adjacent to G along XY or WZ, and H is adjacent to G along $\pi(X,Z)$ or $\pi(W,Y)$.

The preceding arguments also imply that the total number of vertices in Π is at most proportional to the sum of the sizes of the funnels for the edges of P, which are obtained during execution of the shortest path algorithm of Section 2. It is easy to check that this sum is linear in n, so that Π has only $O(n)$ vertices (cf. also [CG]).

It is also easy to calculate adjacency of regions in Π. Specifically, it suffices to consider adjacency of regions near a vertex τ of Π. By the preceding analysis, τ is the dual of a ray r emanating from e and passing through two vertices G, H of P. Then we can apply a simple local, though somewhat lengthy, case analysis (which requires only $O(1)$ time), to enumerate all possible pairs of edges e', e'' whose regions $R(e')$, $R(e'')$ in Π are adjacent near τ; generally, each of these edges will either lie adjacent to G or H, or contain one of the endpoints of r, if this endpoint is different from G and H. We leave details of this case analysis to the reader.

All these observations imply that Π can be calculated in linear time from the output of the executions of the shortest path algorithm of Section 2 with A and B as sources, which themselves take only linear time. Having

calculated Π, we next apply to it one of the linear time preprocessing algorithms of [Ki] or of [EGS] for point location, thus obtaining a data structure from which the region of Π containing (the dual of) any query ray r, and thus also the edge of P first hit by r, can be found in $O(\log n)$ time.

4. Linear preprocessing for the shooting problem in a simple polygon

In this section we show how, given a simple polygon P, we can build in linear time and space a data structure that solves the *shooting problem* for P, as defined by Chazelle and Guibas [CG, Section 3]. This problem asks for preprocessing P so that, given any point X inside P and any direction u, we can quickly compute the point $hit(X,u)$ where the ray emanating from X in direction u hits the boundary of P for the first time. If the polygon P has n sides, the method presented in [CG] solves the shooting problem in $O(\log n)$ time per query, after builing a structure in $O(n \log n)$ time that requires $O(n)$ space. The contribution of this section is to show how to build exactly the same shooting structure used by [CG] in linear time.

We start with a balanced decomposition S of our simple polygon P, obtained by recursively subdividing the polygon according to Chazelle's [Ch] polygon cutting theorem. The method presented in [Ch] runs in $O(n \log n)$ time, but we are able to compute the same decomposition tree in linear time. To do this, we note that Chazelle's algorithm actually calculates a balanced decomposition of the dual tree T of a triangulation of P (see the following subsection for a precise definition of this notion). Since Tarjan and Van Wyk's algorithm produces such a triangulation in linear time, it remains to

show that the following tree decomposition step can also be done in linear time.

4.1. Balanced decomposition of a binary tree in linear time

Let T be a given binary tree with n nodes. If an edge e of T is removed, then T is partitioned into two subtrees. If we now similarly partition each of these subtrees and continue doing this recursively, until the fragments left are single nodes, we obtain another tree structure, which is known as a *decomposition* of T. Such a decomposition is called *balanced* if there is a positive constant α such that each time a subtree T' is partitioned by the removal of an edge, each of the two fragments obtained has size at least $\alpha|T'|$, where $|T'|$ denotes the size of T'. We now make two remarks on such decompositions. First of all, it is clear that in a balanced decomposition the fragment containing a particular node v can be split only $O(\log n)$ times. Secondly, it is known that in any binary tree there is an edge whose removal leaves two components, each with at least $\lfloor (n+1)/3 \rfloor$ nodes. Such an edge is always adjacent to the *centroid* of the tree and can be found in linear time [Ch]. As a result, a balanced hierarchical decomposition of T with $\alpha = 1/3$ can be found in $O(n \log n)$ time total, by applying the centroid partition to each fragment recursively.

In this subsection we will show how the centroid edge for partitioning each fragment can be computed at a cost of only $O(\log n)$ operations, after some appropriate data structures have been set up. The overall cost for obtaining the balanced hierarchical decomposition will then be reduced to

$O(n)$. Our method makes use of an auxiliary *ternary* tree A to facilitate the splitting. The tree A has the same nodes as T, but while T can be arbitrary, A is balanced in a strong sense: it has at most $n/2^h$ nodes of height h. In fact A itself represents a decomposition of T. If all ancestors in A of a node v (but not v itself) are deleted from T together with their incident edges, one of the resulting fragments of T will contain exactly the same nodes as the subtree of A rooted at v.

Once we have the auxiliary tree A, we can find the desired centroid edge splitting T in logarithmic time. Moreover, in the same time bound, we can break A into two auxiliary trees, one for each of the two resulting fragments of T. By continuing this process all the way down to the leaves we obtain the desired balanced hierarchical decomposition of T.

We define the auxiliary tree A by first labelling the nodes of T by certain integer labels. We associate with each node v of T (and hence of A) an *index* i_v and a *label* b_v. The index i_v is both the height of v in A and the number of trailing zeroes in b_v when written in binary. Intuitively, these quantities can be defined as follows. If v is a leaf of T, then $b_v = 1$. Otherwise, if w and z denote the children of v in T, the label b_v is defined as follows: Let i be the position of the leftmost carry in the computation of $b_w + b_z + 1$. The bits of b_v are equal to those of $b_w + b_z + 1$ at and to the left of position i. To the right of position i all bits of b_v are 0. (Note that since no carry arises in the bits of b_v to the left of position i, these bits can be obtained by adding the corresponding bits of b_w and b_z; note also that the i-th bit of b_v must be 1.)

We now present the formal definitions of i_v and b_v. We adopt the following conventions: The notation T_v refers to the subtree of T whose root is the node v. The descendants of v include v itself. Also, as in the remainder of this paper, all logarithms are to the base 2.

To help us construct A, we want the indices i_v to represent a decomposition of T in the following sense: If all nodes v with indices i_v greater than k are deleted from T (together with their incident edges), each remaining subtree includes at most one node with index k. We express this requirement as a static property using *path indices*. The index of a path in T is the maximum index of all interior nodes on the path (i.e. excluding the two end nodes of the path), or -1 if the path contains no interior nodes (i.e. consists of a single edge). The index i_v of node v is defined in terms of the indices of v's descendants (including v): it is the smallest non-negative integer j such that for each $k \geq j$, at most one descendant of v with index k is reachable from v by a path of index less than k (in particular, no proper descendant of v is reachable from v along a path of index less than j). The index of a leaf is taken to be 0. For an example of indices and labels, see Fig. 4.1.

We now give a formal interpretation of the labels b_v to complement the intuitive definition above. We treat b_v as a bit vector: $b_v = \sum_{j \geq 0} 2^j b_v[j]$. The entry $b_v[j]$ is 1 if and only if some (exactly one) descendant of v with index $j \geq i_v$ is reachable from v by a path of index less than j. This means that $b_v[i_v]$ is always 1 (take v itself as the corresponding descendant) and $b_v[j]$ is

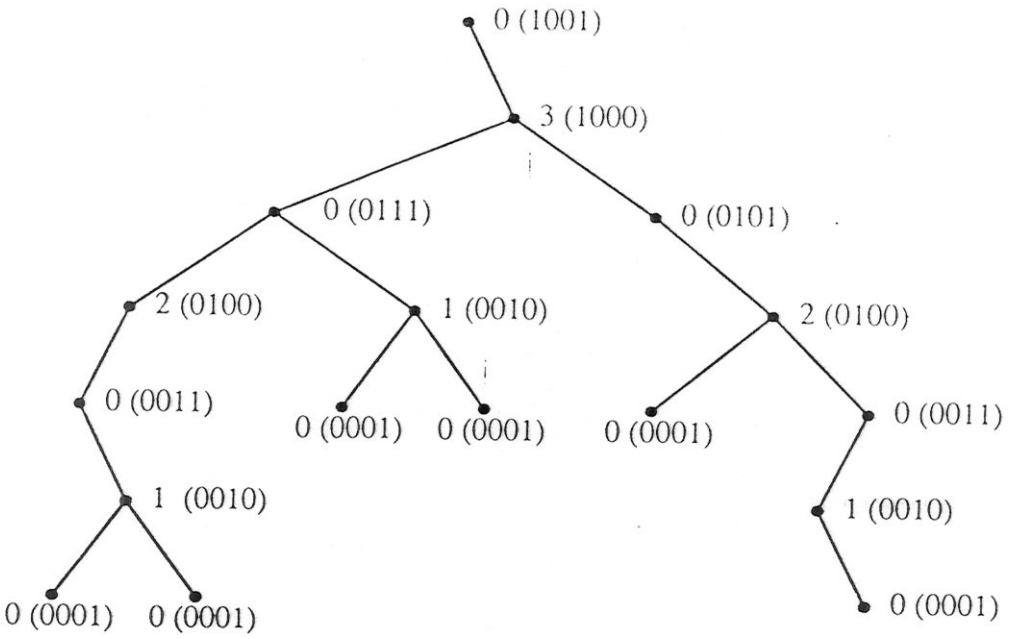

Fig. 4.1. Indices and labels of nodes of T.
Each node is labelled with i_v (b_v).

always 0 for $j < i_v$. If v is a leaf, then $i_v = 0$ and $b_v = 1$. If v has children w and z, then b_w and b_z determine i_v as follows:

$$i_v = \min \{j \geq 0 \mid b_w[j] = b_z[j] = 0 \text{ and } b_w[k] \cdot b_z[k] = 0 \text{ for all } k > j\}.$$

(If v has only one child w, then $b_z[j] = b_z[k] = 0$ in this expression.) Indeed, let j be the index given by the above formula. Note first that for each $k > j$ at most one of $b_w[k]$, $b_z[k]$ is 1. Suppose $b_w[k] = 1$. Then $i_w \leq k$, and there

exists a unique u in T_w of index k reachable from w (and thus also from v) along a path of index less than k. On the other hand, $b_z[k] = 0$, so no node with analogous properties exists in T_z. Hence for each $k > j$ there exists at most one node in T_v of index k which is reachable from v along a path of index less than k. Similarly no node in T_v other than v has index j and is reachable from v along a path of index less than j. This shows $j \geq i_v$. But the arguments just used and the definition of i_v imply that i_v itself is one of the indices satisfying the condition in the above formula. Hence $j = i_v$.

Once i_v is known, b_v has a simple definition:

$$b_v[j] = \begin{cases} b_w[j] + b_z[j] & \text{if } j > i_v \\ 1 & \text{if } j = i_v \\ 0 & \text{if } j < i_v. \end{cases}$$

Note that these definitions accord with the intuitive ones given above. Given b_w and b_z, bitwise logical operations and table lookup can determine i_v and b_v in constant time (even without these bitwise operations, the algorithm will still run in linear time; see below).

The node indices guide the construction of A. The node a with highest index is the root of A. Removal of this node and its incident edges from T generates at most three subtrees in T, which recursively define the (at most) three subtrees in A of the root node. The lemma that follows shows that the construction is well-defined, in the sense that the indices do represent a decomposition of T.

Lemma 4.1: If all nodes with indices greater than some positive j are removed from T, along with their incident edges, each remaining subtree of T

has exactly one node with maximal index.

Proof: By the definition of the index function, no two nodes with index j, one a descendant of the other in T, are joined by a path of index less than j. Similarly, no node with index $k < j$ is joined to two descendant nodes with index j by paths of index less than j. Hence, after nodes with index greater than j are deleted, each node with index j is alone in its subtree. Furthermore, if node deletion leaves a subtree with $k < j$ as its maximum node index, the subtree remains unchanged if all nodes with index greater than k are deleted. It follows that there is only one node with maximum index in each subtree. □

Lemma 4.2: (a) For each node v, $|T_v| \geq b_v$.

(b) for each index j, there are at most $\left\lfloor |T|/2^j \right\rfloor$ nodes with that index in T.

Proof: The proofs of both statements are inductive. The first follows easily from the intuitive definition of b_v. For a leaf v, $|T_v| = b_v = 1$. Since $b_v \leq b_w + b_z + 1$, induction implies that $|T_v| = |T_w| + |T_z| + 1 \geq b_v$.

As to the proof of (b), let us define $D(v)$ to be the set of v's descendants whose indices are at most i_v and which are joined to v by paths of index less than i_v. Plainly, no node other than v with index i_v can be an ancestor of one of these nodes and reach it by a path of index less than i_v.

We next show by an inductive argument that $|D(v)| \geq 2^{i_v}$. Indeed, this is clearly true for v a leaf. If v has children w and z, then a node q is in $D(v)$ if either $q = v$ or q is a descendant of w or of z, say for definiteness a descendant of w, such that $i_q < i_v$, $i_w < i_v$, and the index of the path from w

to q is also less than i_v. In this latter case, let $j < i_v$ be the maximum node index along the path from w to q (including these two nodes), and let u be the unique node along this path with index j. Then clearly $q \in D(u)$ and $b_w[j] = 1$. It is also easy to check the converse statement, namely that $D(v)$ contains each set $D(u)$ for a descendant u of w or of z that causes $b_w[j]$ or $b_z[j]$ to be 1 for any $j < i_v$. But the sets $D(u)$ are all disjoint. Indeed, if $q \in D(u) \cap D(u')$, then without loss of generality we can assume that u is a descendant of u' which is a descendant of w. But if $i_u \leq i_{u'}$ then u cannot have caused $b_w[i_u]$ to be 1, and if $i_u > i_{u'}$ then q cannot belong to $D(u')$. Thus, by induction hypothesis,

$$|D(v)| \geq 1 + \sum_{j<i_v} 2^j (b_w[j] + b_z[j]) \geq 2^{i_v}$$

(because $1 + b_w + b_z$ has a carry at the i_v-th bit), and, since all the sets $D(v)$ for nodes with the same index i_v are disjoint, (b) follows. □

Since i_a is the height of A, part (b) of the preceding lemma shows that A has height at most $\lfloor \log n \rfloor$.

Constructing A:

It is possible to determine the indices of the nodes and build A during a single postorder (depth-first) traversal of T. The two trees T and A have the same node set. To distinguish the edge sets, we will speak of the *edges* of T and of the *links* of A. For each label b_v, the construction requires a vector p_v of pointers to nodes. If $b_v[j]$ is 1, then $p_v[j]$ points to the (unique) descendant of v in T that has index j and is reachable by a path of index less than j. The depth-first search defines a path π from the root of T to the

current node. The construction maintains b_v and p_v for each node v that is a child of a node on π but is not on π itself. Each such v is the root of a subtree T_v;, and these subtrees are all disjoint. The vectors b_v and p_v take $O(\log |T_v|)$ space, which is linear when summed over all such nodes v.

When the postorder traversal visits a node v, the algorithm constructs b_v and p_v from the vectors stored at the children w and z of v. At the same time it builds auxiliary tree links for nodes that appear in p_w and p_z but not in p_v. Such a node (suppose it is $u \in T_w$) has an index less than i_v, so its parent in A is either v or a node of T_w. In fact, its parent in A is the node with minimal index $j > i_u$ reachable from u by a path in T of index less than i_u. These observations imply that if j is the largest integer less than i_v such that $b_w[j] = 1$, then v is the parent of $p_w[j]$ in A. Similarly, if $j > k$ are integers less than i_v such that $b_w[j] = b_w[k] = 1$ and $b_w[l] = 0$ for $j > l > k$, then $p_w[j]$ is the parent of $p_w[k]$ in A. Linking these nodes to their parents in A takes time proportional to i_v.

When the algorithm reaches the root t of T, it links the nodes pointed to by p_t as if t were the child of a node with index $\lfloor \log n \rfloor + 1$. The root a of A is the highest-indexed node appearing in p_t.

Even without logical operations on the b_v vectors, the construction of A requires only linear time. When the traversal visits v (with children w and z), the algorithm takes time proportional to the number of links made plus the logarithm of the smaller of $|T_w|$ and $|T_z|$, which gives a linear overall bound. In the same time bound a traversal of A can be used to compute $|A_v|$ for each node v; this information is needed in order to split the auxiliary tree.

Remark: A two-pass construction algorithm may be simpler than the one given, especially if bitwise operations are available. The first pass computes node indices in one traversal of T. The second pass builds A from the bottom up, using an auxiliary graph T' which is maintained dynamically during the construction, and which is initialized to T. As long as T' is not a single node, the algorithm picks a node v in T' with minimum index, selects v's lowest-indexed neighbor in T' to be its parent in A, creates pairwise edges between all of v's neighbors in T', and then deletes v. Including the created edges, a node v can have at most $3i_v + 3$ incident edges in T', which gives a logarithmic bound on the degree of any node and a linear overall bound on the number of edges. Each edge is examined once, so the algorithm constructs A in linear time. We leave it to the reader to check the correctness of this alternative procedure.

Decomposing T:

Given A, only a linear amount of additional work is needed to find a balanced decomposition of T. The algorithm first uses A to split T into balanced subtrees, then builds an auxiliary tree for each fragment, and finally decomposes each fragment recursively. Using the subtree sizes $|A_v|$, a simple top-down search of A finds a centroid edge e_c in time proportional to the height of A. Removing e_c from T results in two subtrees R and B. For ease of exposition, let us assume that the nodes of these two subtrees are painted red and black, respectively, and that R contains a, the root of A.

To allow recursive splitting, R and B must have auxiliary trees A^R and A^B

built on top of them. Each of R and B has a unique node with maximum index. These nodes, one of which is a, are the roots of A^R and A^B. Splitting A to form the two new auxiliary trees is relatively straightforward. Let v be the endpoint of e_c with smaller index. The path P in A from a to v includes the other endpoint of e_c. If the nodes are augmented with their preorder and postorder numbers in T, then a constant-time test can determine the color of a node. The first black node on P is the root of A^B. The new auxiliary trees are constructed by dividing P into two monochromatic paths. In A^R and A^B, every node w on P has as its successor the next node on P with the same color as w. The counts $|A_w|$ are still valid in A^R and A^B except at nodes w on P, where they must be recomputed. These changes take time proportional to the length of P. To see that the modifications of A are correct, note that at most one tree A_v is dichromatic for nodes v of a given index, and that after P is modified, each node has auxiliary tree links only to nodes of its own color.

The fragments R and B can be recursively decomposed with the aid of A^R and A^B. To analyze the cost of this construction, we note that auxiliary tree nodes cannot increase in height as the decomposition proceeds, and that the total number of nodes in A of height k is at most $n/2^k$. The cost of splitting an auxiliary tree whose root has height k is $O(k)$, and furthermore no node can appear as the root of such an auxiliary tree more than $O(k)$ times total. Therefore the whole decomposition takes time

$$O(\sum_{k=0}^{\lfloor \log n \rfloor} k^2 \frac{n}{2^k}) = O(n).$$

The preceding discussion constitutes a proof of the following theorem:

Theorem 4.1: It is possible to find a balanced hierarchical decomposition of an arbitrary binary tree in linear time.

Remark: An alternative technique for obtaining such a balanced tree decomposition is obtained by using a simplified version of the dynamic tree data structure (as described in [Ta, Ch. 5]), which can also support logarithmic-cost tree-splitting operations. However the method sketched above is somewhat simpler and more direct.

4.2. The shooting problem

The balanced decomposition S of T as obtained above is most usefully thought of as a balanced tree. The leaves of that tree correspond to the triangles of an underlying triangulation T of P, while the internal nodes correspond to the diagonals in T. The root of S represents a diagonal d that partitions P into two subpolygons P_1 and P_2. Each of these subpolygons is in turn partitioned by a diagonal, and so on till we just have the triangles in T. Because S is a balanced decomposition, the maximum depth of any leaf is $O(\log n)$. We give each diagonal d of the triangulation an integer *label* $\lambda(d)$, which represents its level or depth in the tree structure. By convention the diagonal corresponding to the root s of the tree S has label 1; the children of the root have label 2, and so on.

The key idea for solving the shooting problem is to store, for certain pairs of diagonals (d_1,d_2), a representation of all lines that cut d_1 and d_2 but do not intersect the portion of P between d_1 and d_2. This set of lines is

compactly represented by the hourglass for the pair (d_1,d_2), as discussed in Section 3. We will denote by Λ the list of all such pairs. It is easiest to construct these auxiliary structures from the bottom up.

Consider the following process, which we term the *merging process:* Start with the underlying triangulation T; its triangles can be considered as the leaves of the balanced tree S referred to above. For each triangle, at least two of whose sides are diagonals, add all pairs of bounding diagonals to the list Λ of pairs to be considered. Now remove all diagonals of the highest label. This creates new regions by merging pairs of old regions (triangles). It cannot happen that three or more regions get merged into one, since diagonals with the same label are never adjacent in T. If R_1, R_2 are two regions being merged, then add to the list Λ all pairs (d_1,d_2), where d_1 (resp. d_2) is a diagonal bounding R_1 (resp. R_2) and remaining after the merge. We continue this process, at each stage removing all diagonals of the next label and adding to the list Λ all new pairs of diagonals bounding one of the newly formed regions. Because of the structure of the balanced decomposition of P, it will never be the case that two diagonals with the same label become adjacent. Thus at each stage only pairs of regions get merged. Furthermore, any region that ever arises in this process will have at most a logarithmic number of diagonals on its boundary, as no two of them can have the same label.

In S, the balanced decomposition tree of P, each pair of diagonals (d_1,d_2) produced by the above process corresponds to an (ancestor, descendant) pair of nodes. This can be most easily seen by imagining the time-reversal of the

merging process. A leaf of the current decomposition is expanded by introducing a new highest numbered diagonal g. Inductively we assume that the region corresponding to this leaf is a descendant of all diagonals bounding it. The introduction of the new diagonal obviously preserves this invariant. Furthermore, the new pairs of the form (e,g) that must now be added to Λ are clearly (ancestor, descendant) pairs. This proves our claim. From now on, whenever we write a pair (d_1,d_2) of diagonals in Λ, we will follow the convention that the first element is the ancestor and the second the descendant.

Let S^* denote the decomposition tree after the addition of all edges corresponding to the diagonal pairs in Λ (all edges of S naturally are represented in Λ). Let e be a particular diagonal of the decomposition that occurs with label $\lambda(e) = \text{depth}(e)$. How many pairs of the form (e,g) can be in Λ (recall that g must be a descendant of e)? Note that g is uniquely determined by its level and the side of e it lies on - it is the "nearest" diagonal to e of the right level and on the appropriate side. Thus no more than $2(\tau(e) - \lambda(e))$ such pairs can exist in Λ, where $\tau(e)$ is the maximum depth in S of any node in the subtree rooted at e. Let $\mu(e)$ denote the total number of pairs in Λ that arise out of the subtree of S rooted at e. Then the above remarks prove that $\mu(e) \leq 2 \sum (\tau(g) - \lambda(g))$, where the sum is taken over all descendants g of e, including e itself. If s, l and r denote respectively the root of S and the root's left and right children, then we can write

$$\mu(s) = \mu(l) + \mu(r) + O(\log n) ,$$

where the last term is the contribution of s to Λ. Since S is balanced, there is

some constant α, $0<\alpha<1/2$, such that the subtrees of each node x of S are in size at least the fraction α of the whole tree rooted at x. By standard techniques it then follows that the above recurrence has a solution of the form $\mu(s) = O(|S|) = O(n)$. This proves that Λ, and therefore S^*, has linear size.

As we remarked at the beginning, our aim is to associate with each pair (e,g) in Λ a visibility structure representing all lines cutting diagonals e and g, but not the portion of the polygon P between these diagonals. Such lines are constrained to avoid the two inwards concave chains defined by the hourglass illustrated in Fig. 4.2. These chains are the convex hulls of the two polygonal paths joining e and g along P.

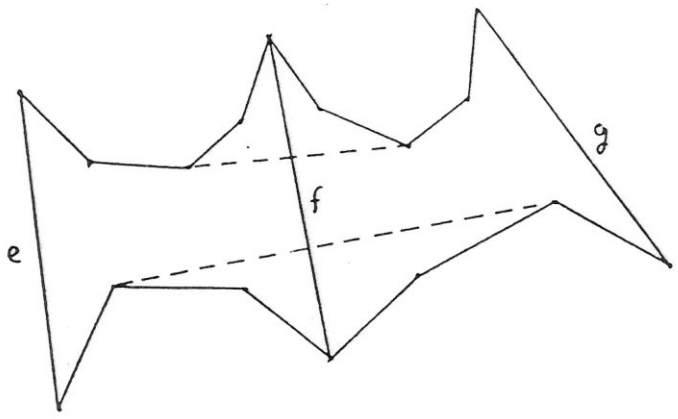

Fig. 4.2.

Suppose that we are in the midst of the diagonal-removing process and are currently working on diagonal f, whose removal merges two regions, one containing the diagonal e and the other the diagonal g. Suppose also that we have already computed the hourglasses for the pairs (e,f) and (f,g). Then in order to compute the hourglass for (e,g) it suffices to compute the outer common tangents of the corresponding pairs of concave chains in the hourglasses for (e,f) and (f,g). This is illustrated in Fig. 4.2. Note that since only two new edges are needed to form the new hourglass from the old ones, the total number of edges in all the hourglasses will be proportional to the number of diagonal pairs in Λ, i.e. it will be $O(n)$.

Recall our convention to write each diagonal pair (e,g) in Λ so that e is the ancestor and g the descendant. The size of the hourglass of (e,g) is bounded by the size $w(e,g)$ of the portion of P between e and g. Now we claim that $\log w(e,g) = O(\tau(e) - \lambda(e))$, for in a balanced decomposition the height of each subtree is logarithmic in its size. So the cost of computing the common tangents needed in the construction of the hourglass of e and g is $O(\tau(e) - \lambda(e))$. Therefore, if $\kappa(e)$ denotes the total cost of these common tangent computations for all pairs of diagonals in the subtree rooted at e, we have $\kappa(e) = O(\sum (\tau(g) - \lambda(g))^2)$, where the sum ranges over all descendants g of e (including e). By arguments entirely analogous to those used above we can write a recurrence of the form

$$\kappa(s) = \kappa(l) + \kappa(r) + O(\log^2 n)$$

for the total cost of computing the common tangents and conclude that it too is $O(n)$.

We will be traversing the search structure S^* from the top down in order to solve the shooting problem. The details are exactly as in [CG]. Because of this, we allocate each hourglass edge to the *highest* hourglass in S^* that has it as an edge - in other words to the last one formed in the above merging process. This means that at each stage, as we compute the hourglass of (e,g) from the hourglasses of (e,f) and (f,g), we need first to find the common tangents discussed above, and then split the old hourglasses where the common tangents touch them. The extremal pieces get joined by the tangent to form the new hourglass, while the inner pieces are left with the old hourglasses. See Fig. 4.2 for an illustration. This splitting and joining can be implemented in the same order of magnitude time as the common tangent computation, if a balanced tree structure is used to represent the hourglasses. The details are straighforward and therefore omitted. This implies that the entire search structure S^* can be computed in linear time and that it occupies linear space. We have therefore shown that:

Theorem 4.2: Given a simple polygon P of n sides, it is possible in linear time and space to construct an auxiliary structure that allows us to solve the shooting problem for P in time $O(\log n)$ per query. These bounds are optimal.

Note: We have been able to avoid the use of finger trees in the above construction because we started out with a balanced decomposition. We could in fact have obtained a linear shooting structure in linear time from *any* decomposition S of P, whether balanced or not. This requires the use of

finger trees in a manner analogous to that of Section 2. However the resulting shooting structure S^* can no longer guarantee logarithmic search time, as there is no logarithmic bound on its depth.

5. The Convex Rope Algorithm

As an additional application of the shortest path algorithm of Section 2, we consider the following *Convex Rope problem,* as posed by Peshkin and Sanderson [PS]: Let P be a simple polygon, and let s be a vertex of P lying on its convex hull. Let v be another vertex of P. The *clockwise convex rope* from s to v is the shortest polygonal path $(s = p_0, \ldots, p_m = v)$ starting at s and ending at v that does not enter the interior of P and that is clockwise convex, in the sense that the directed segment p_ip_{i+1} lies to the right of the directed segment $p_{i-1}p_i$, for $i = 1, \ldots, m-1$. The counterclockwise convex rope from s to v is defined in a symmetric manner (see Fig. 5.1). Not all the vertices of P necessarily admit convex ropes from s. Those vertices v that do admit both clockwise and counterclockwise ropes from any such s are precisely those that are "visible from infinity" (cf. [PS]); calculation of these convex ropes is required in [PS] to plan reachable grasps of P by a simple robot arm. In [PS], an $O(n^2)$ algorithm is presented. Using the shortest path algorithm of Section 2, we obtain an improved algorithm running in linear time.

The convex rope problem can be solved as follows. Compute first the convex hull of P in linear time (cf. [PS], [GY]). The clockwise (resp. counterclockwise) convex rope from s to any vertex v on the convex hull can

simply be calculated by moving along the convex hull in a clockwise (resp. counterclockwise) direction from s to v. For each vertex v not on the convex hull, v lies inside a simple polygon Q (a "bay" of P) bounded by some subsequence of the sides of P and by an edge of the convex hull that is not a side of P (see Fig 5.1; note also that the collection of all these bays can be found in linear time).

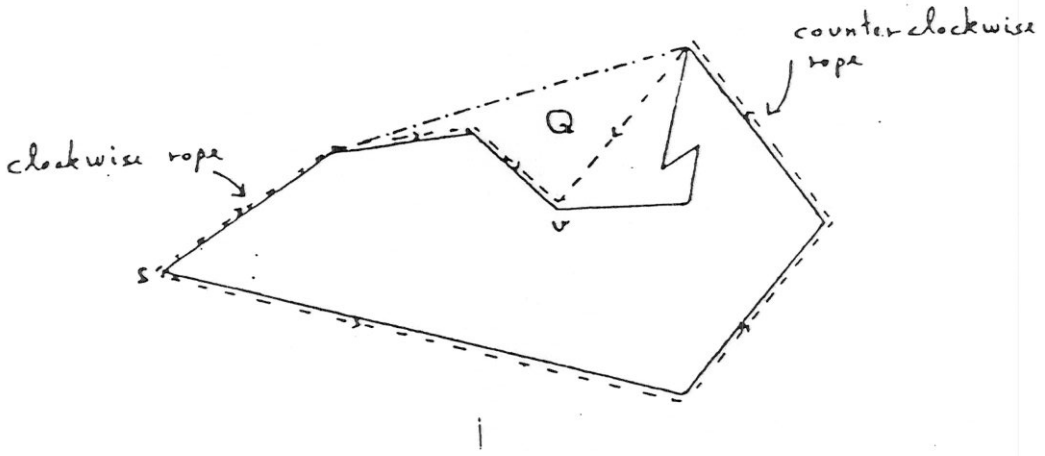

Fig. 5.1. The convex rope problem.

Let v_1, v_2 be the endpoints of this edge such that v_1 is reached first from s when moving along the convex hull in a clockwise direction. The clockwise (resp. counterclockwise) convex rope from s to v is then the clockwise convex rope from s to v_1 (resp. the counterclockwise convex rope from s to v_2) followed by the shortest path from v_1 to v (resp. from v_2 to v) within Q,

provided that this shortest path is clockwise (resp. counterclockwise) convex (otherwise the required convex rope does not exist). Hence we can use the shortest path tree algorithm of Section 2 to calculate the shortest paths from v_1 and from v_2 to all the vertices of Q (and also to check whether these paths are convex in the required directions) in time $O(|Q|)$. Since the sum of the sizes of all the "bays" Q of P is $O(n)$, it follows that we can solve the convex rope problem in $O(n)$ time.

6. Conclusion

We have presented a collection of linear time algorithms for solving a variety of shortest paths and visibility problems inside a simple polygon, exploiting interesting relationships between these two types of problems. Our work has enriched the collection of problems solvable in linear time for simple polygons, but there are quite likely many additional problems for which linear time solutions can be developed. For example, Suri [Su] has recently extended our technique to solve in linear time the k-visibility problem, in which, given a simple polygon P and an edge e of P, we wish to partition P into disjoint subparts $P_1, P_2, \cdots$ such that P_1 contains all points in P directly visible from some point on e, P_2 contains all points in P visible from some point in P_1 but not from e, and so on.

Acknowledgement

The authors wish to thank Chris Van Wyk for useful discussions concerning the problems studied in this paper.

References:

[As] T. Asano, Efficient algorithms for finding the visibility polygon for a polygonal region with holes, manuscript, University of California at Berkeley.

[AT] D. Avis and G.T. Toussaint, An optimal algorithm for determining the visibility of a polygon from an edge, *IEEE Trans. on Computers,* C-30 (1981) pp. 910-914.

[Ch] B. Chazelle, A theorem on polygon cutting with applications, *Proc. 23th IEEE Symp. on Foundations of Computer Science,* 1982, pp. 339-349.

[CG] B. Chazelle and L. Guibas, Visibility and intersection problems in plane geometry, *Proc. ACM Symposium on Computational Geometry,* 1985, pp. 135-146.

[EGS] H. Edelsbrunner, L. Guibas and J. Stolfi, Optimal point location in monotone subdivisions, DEC/SRC Tech. Rept. 2, 1984.

[EG] H.A. El Gindy, An efficient algorithm for computing the weak visibility polygon from an edge in simple polygons, manuscript, McGill University, 1984.

[EG2] H.A. El Gindy, Hierarchical decomposition of polygons with applications, Ph.D. Dissertation, School of Computer Science, McGill University, 1985.

[EA] H.A. El Gindy and D. Avis, A linear algorithm for computing the visibility polygon from a point, *J. Algorithms* 2 (1981) pp. 186-197.

[Fi] S. Fisk, A short proof of Chvatal's watchman theorem, *J. Comb. Theory,* Ser. B, 24 (1978), p. 374.

[FM] A. Fournier and D.Y. Montuno, Triangulating simple polygons and equivalent problems, *ACM Trans. on Graphics* 3 (1984) (2) pp. 153-174.

[GJPT] M.R. Garey, D.S. Johnson, F.P. Preparata and R.E. Tarjan, Triangulating a simple polygon, *Information Processing Letters,* Vol. 7, 4, 1978, pp. 175-179.

[GMPR] L. Guibas, E. McCreight, M. Plass and J. Roberts, A new representation for linear lists, *Proc. 9th ACM Symp. on Theory of Computing,* 1977, pp. 49-60.

[GRS] L. Guibas, L. Ramshaw and J. Stolfi, A kinetic framework for computational geometry, *Proc. 24th IEEE Symp. on Foundations of Computer Science,* 1983, pp. 100-111.

[GY] R.L. Graham and F.F. Yao, Finding the convex hull of a simple polygon, *J. of Algorithms.*

[GK] D.H. Greene and D.E. Knuth, *Mathematics for the analysis of algorithms,* Birkhauser, Boston 1982.

[Hi] P.T. Highman, The ears of a polygon, *Inf. Proc. Letters* 15 (1982) pp. 196-198.

[HM] S. Huddleston and K. Mehlhorn, A new data structure for representing sorted lists, *Acta Informatica* 17 (1982) pp. 157-184.

[Ki] D. Kirkpatrick, Optimal search in planar subdivisions, *SIAM J. Computing,* 12 (1983) pp. 28-35.

[Le] D.T. Lee, Visibility of a simple polygon, *Comp. Vision, Graphics and Image Processing* 22 (1983) pp. 207-221.

[LL] D.T. Lee and A. Lin, Computing the visibility polygon from an edge, manuscript, Northwestern University, 1984.

[LP] D.T. Lee and F.P. Preparata, Euclidean shortest paths in the presence of rectilinear barriers, *Networks* Vol 14, 3, 1984, pp. 393-410.

[MA] D. McCallum and D. Avis, A linear algorithm for finding the convex hull of a simple polygon, *Inf. Proc. Letters* 9 (1979) pp. 201-206.

[Me] K. Mehlhorn, *Data Structures and Efficient Algorithms, Vol. I: Sorting and Searching,* Springer Verlag, Berlin (1984).

[PS] M.A. Peshkin and A.C. Sanderson, Reachable grasps on a polygon: The convex rope algorithm, *Technical Report CMU-RI-TR-85-6,* Carnegie-Mellon University, 1985 (to appear in *IEEE J. Robotics and Automation* 2 (1) (1986)).

[PS2] F.P. Preparata and K.J. Supowit, Testing a simple polygon for monotonicity, *Inf. Proc. Letters* 12 (1981) pp. 161-164.

[Su] S. Suri, Finding minimum link paths inside a simple polygon, to appear in *Comp. Vision, Graphics and Image Processing,* 1986.

[Ta] R.E. Tarjan, *Data Structures and Network Algorithms,* CBMS-NSF Regional Conference Series in Applied Mathematics, SIAM, Philadelphia 1983.

[TV] R.E. Tarjan and C. Van Wyk, A linear time algorithm for triangulating simple polygons, manuscript, October 1985.

[To] G. Toussaint, Shortest path solves edge-to-edge visibility in a polygon, Tech. Rept. SOCS-85.19, McGill University, 1985.