Algorithmica (1987) 2: 209-233

Algorithmica

Linear-Time Algorithms for Visibility and Shortest Path Problems Inside Triangulated Simple Polygons

Leonidas Guibas,[1,2] John Hershberger,[1] Daniel Leven,[3] Micha Sharir,[3,4] and Robert E. Tarjan[5]

Abstract. Given a triangulation of a simple polygon P, we present linear-time algorithms for solving a collection of problems concerning shortest paths and visibility within P. These problems include calculation of the collection of all shortest paths inside P from a given source vertex s to all the other vertices of P, calculation of the subpolygon of P consisting of points that are visible from a given segment within P, preprocessing P for fast "ray shooting" queries, and several related problems.

Key Words. Triangulation, Simple polygon, Visibility, Shortest paths, Ray shooting, Computational geometry.

1. Introduction. Recently, Tarjan and Van Wyk [30] developed an algorithm for triangulating simple polygons that runs in time $O(n \log\log n)$, thereby improving the previous $O(n \log n)$ algorithm of [11] and making significant progress on a major open problem in computational geometry. Even though this result falls short of the goal of achieving a linear time bound, it shows that triangulating simple polygons is a simpler problem than sorting, and raises the hope that linear time triangulation might be possible. This result thus renews interest in linear-time algorithms on already-triangulated polygons, a considerable number of which have been recently developed (for a list of these see, e.g., [10] and [30]). (Such algorithms should, of course, be contrasted with linear-time algorithms on "raw" simple polygons, such as calculation of the convex hull of such a polygon P [12], [22], calculation of the subpolygon of P visible from a given point [8], [19], and others.) Problems known to be solvable in linear time, given a triangulation of the polygon P, include calculation of the shortest path inside P between two specified points [21], preprocessing P to support logarithmic-time point location queries [18], [5], and stationing guards in simple art galleries [9].

[1] Computer Science Department, Stanford University, Stanford, CA 94305, USA.
[2] DEC/SRC, 130 Lytton Avenue, Palo Alto, CA 94301, U.S.A.
[3] School of Mathematical Sciences, Tel-Aviv University, Tel Aviv 69978, Israel.
[4] Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA.
Work on this paper by this author has been supported by Office of Naval Research Grant N00014-82-K-0381, National Science Foundation Grant No. NSF-DCR-83-20085, and by grants from the Digital Equipment Corporation, the IBM Corporation, and from the U.S.-Israel Binational Science Foundation.
[5] Department of Computer Science, Princeton University, Princeton, NJ 08544, USA, and AT&T Bell Laboratories, Murray Hill, NJ 07974, USA. Work on this paper by this author has been supported by National Science Foundation Grant DCR-86-05962.

Received May 14, 1986; revised December 9, 1986. Communicated by Bernard Chazelle.

In this paper we continue the search for linear-time postprocessing algorithms on triangulated simple polygons. We present several such algorithms, which solve solve the following problems, given a triangulated 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. (Our algorithm even provides a linear-time processing 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 the path.)
2. Given a fixed edge e of P, calculate the subpolygon $\mathrm{Vis}(P, e)$ consisting of all points in P visible from (some point on) e.
3. Given a fixed edge e of P, 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 a fixed edge e of P, 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 any direction u, the first point $\mathrm{hit}(x, u)$ on the boundary of P hit by the ray in direction u from x can be computed in $O(\log n)$ time. (To solve this problem, we use the techniques described in [4], but show how to construct the data structure they need in linear time.)
6. Calculate a balanced decomposition tree of P by recursively cutting P along diagonals, as in [3].
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; see Section 2.1.)

Our results improve previous algorithms for some of these problems (see [3], [4], and [24]). Most of our algorithms are based on the solution to Problem 1 and exploit interesting relationships between visibility and shortest path problems on a simple polygon. Our technique for solving Problem 1 extends the technique of Lee and Preparata [21] for calculating the shortest path inside P between a single pair of points. To obtain an overall linear-time performance, it uses *finger search trees*, a data structure for efficient access into an ordered list when there is locality of reference (see [14], [17], and [30]).

The paper contains four sections. In Section 2 we present our linear-time solution to the shortest path problem (Problem 1), and also obtain a solution to Problem 7 as an easy application. In Section 3 we solve the visibility problems (Problems 2–4). In Section 4 we address Problem 5, and in the Appendix we present our balanced tree decomposition algorithm for Problem 6.

2. Calculating a Shortest Path Tree for a Simple Polygon. Let P be a (triangulated) simple polygon having n vertices, and let s be a given *source vertex* of P. (Our algorithm will also apply, with some minor modifications, to the case in

which s is an arbitrary point in the interior or on the boundary of P. For the sake of exposition, the algorithm below is described for the case in which s is a vertex of P, and we later comment on the modifications required to handle the case of an arbitrary source s.) For each vertex v of P, we denote the Euclidean shortest path from s to v inside P by $\pi(s, v)$. It is well known (see [21]) 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 plane tree $Q_s(P)$ rooted at s, which we call the *shortest path tree* of P with respect to s. This tree has 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, once a triangulation of P is given.

Let G be a triangulation of the interior of P. 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 three. 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 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 the diagonals d_i, $1 \leq i \leq l$, and each of them exactly once.

Let $d = uw$ be a diagonal or an edge of P and let a be the deepest common ancestor of u and w in the shortest path tree $Q_s(P)$. It is shown in [21] that $\pi(a, u)$, $\pi(a, w)$ are both *outward convex*; i.e., the convex hull of each of these subpaths lies outside the open region bounded by $\pi(a, u)$, $\pi(a, w)$, and by the segment uw. This implies that along each of $\pi(a, u)$ and $\pi(a, w)$ the slopes of the sides change in a monotonic fashion. Following [21], 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 the funnel. Suppose next that d is a diagonal of P used by G, and let $\triangle uwx$ 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 Figure 2.1). These observations form the basis for the algorithm of Lee and Preparata [21], and for ours.

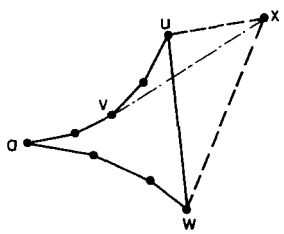

Fig. 2.1. Splitting a funnel.

We are now ready to describe our algorithm. As the algorithm comes to process a diagonal $d = uw$ of P, it maintains the current funnel $F = F_{uw}$ as a sorted list $[u_l, u_{l-1}, \ldots, u_1, a, w_1, \ldots, w_k]$, 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 except for a), and $u_l = u$, $w_k = w$. We denote in the algorithm the cusp a of the present funnel F by CUSP(F).

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 $a = \text{CUSP}(F)$ (thus $F = \pi(a, u) \cup \pi(a, w)$). Let $\triangle uwx$ be the unique triangle in the triangulation G of P that has uw as an edge and that has not yet been processed (see Figure 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 $v = a$). The search is very similar to that of finding a tangent to a convex polygon from an exterior point (as described, e.g., on p. 115 of [25]); thus 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 binary search is applicable. 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 set CUSP(F_1) := v, CUSP(F_2) := a. If, on the other hand, v belongs to $\pi(a, w)$ then set 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 call PATH(F_1) recursively.
(d) If the line segment wx is a diagonal of P then call PATH(F_2) recursively.

The list representing the present funnel F is stored in a *finger search tree* (see [14], [17], and [30]). This structure is essentially a search tree equipped with *fingers* (which, in our application, are always placed at the first node and at the last node of the tree in symmetric order). To facilitate the search for the vertex v at which the tangent from x touches F, we store with each item v of F appearing in the search tree two points, to its predecessor and successor in F. This enables us to calculate the slopes of the two edges of F incident to v in constant time, and by comparing these slopes with that of vx we can tell in constant time on which side of v the binary search should continue. Thus, this structure supports searching for a tangent from a point x in time $O(\log \delta)$, where δ is the distance from the point of tangency to the nearest finger, and also supports operations that split the tree into two subtrees at an item v in amortized time $O(\log \delta)$, with δ as above.

The main difference between our algorithm and the algorithm of Lee and Preparata [21] is in the techniques for representation, searching, and splitting of funnels. In [21] 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 F 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 search trees (and is thus subtler than the simple linear list representation used in [21]), in order 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.

The following lemma proves some additional properties of funnels, which we will use in an enhancement of the algorithm, to be described later in this section.

LEMMA 2.1. *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 union of these regions is the entire polygon P. 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 [21]) used to justify the correctness of our algorithm. As to the second part, 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. As to the second claim, let x be any point inside P, and let Δ be the triangle in the triangulation of P which contains x. It is clear that after the above algorithm processes Δ, it will have produced some funnel F_e, where e is one of the sides of Δ, such that x lies between F_e and e. If e is an edge of P the claim is immediate; otherwise, follow the recursion of our algorithm from e further on. Each of these recursive steps takes a region bounded between some funnel F_t and its associated diagonal t, splits it into two subregions, and adds a portion of the current triangle to each of these subregions. Thus if the input region for such a step contains x, one of the output regions must also contain x. Thus eventually our algorithm will have reached an edge e for which $\Phi(e)$ contains x. Finally, 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 $O(n)$. □

To bound the time required by the algorithm we argue as follows. Let T be the dual tree of the triangulation of P. It is easily seen 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 that lies in several triangles of the given triangulation, 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. The algorithm will then correctly propagate the funnel structure to all the other triangles containing s; the funnels for (the edges of) the triangles containing s 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 the parameter m_ζ denoting the size (i.e., number of edges) of the funnel F just before ζ is processed.

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 search trees is $K(\zeta) = O(\min(\log m_{\zeta_1}, \log m_{\zeta_2}))$. 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 one when descending 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 one.

We begin our analysis with the following observation: the "direct costs" $K(\zeta)$ at nodes $\zeta \in T$ that have just one child or are leaves, sum to at most $O(n)$. Indeed, suppose $\zeta \in T$ has just one child ζ', and that the funnel at ζ has been split into two parts having m' and $m_\zeta - m'$ edges, respectively. Then the vertices of P lying in one of these parts will never be encountered again by the algorithm (by the same reasoning used in [21] 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 $K(\zeta)$'s 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 by exactly one at each node ζ of T having a single child, because these cases provide maximum growth of the function m down the tree T. Under this additional assumption we have:

LEMMA 2.2. *For a node $\zeta \in T$, let the leaves of the subtree T_ζ of T rooted at ζ be $\eta_1, \ldots, \eta_k$, and set $M_\zeta = \sum_{j=1}^{k} m_{\eta_j}$. We then have*

$$M_\zeta = m_\zeta + |T_\zeta|,$$

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

PROOF. The lemma follows from summing the equations

$$\sum_{j=1}^{t} m_{\xi_j} = m_\xi + t$$

for each internal node ξ of T_ζ, where $\{\xi_j\}_{j=1}^{t}$ are the children of ξ ($t = 1$ or 2). □

COROLLARY. *In particular, we have* $m_\zeta \leq M_\zeta$.

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

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

In solving these recurrence formulas, we can 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 (using the preceding Corollary) 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 $C^*(m) = O(m)$ (see p. 185 of [23]). Finally, by Lemma 2.2 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

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

Summarizing our analysis, we obtain:

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

REMARK. As finger trees are complicated to implement, we could have obtained a simpler but less efficient version of the algorithm by maintaining funnels simply as doubly linked linear lists (the same data structure as that used in [21]), and by performing each search through a funnel in a linear manner, starting simultaneously from both ends 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 smaller of the subfunnel sizes. This leads to a recurrence formula for C^* of the form

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

whose solution is $C^*(n) = O(n \log n)$ (see pp. 25–27 of [13]).

REMARK. Another possibility is to represent the funnels by self-adjusting search trees [27]. The dynamic optimality conjecture of Sleator and Tarjan for such trees (see [27]) suggests that the resulting shortest path algorithm runs in linear time, although we do not know how to prove this.

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 $\triangle = \triangle uvw$ of G. Then we split $\triangle$ into three subtriangles $\triangle suv$, $\triangle svw$, $\triangle swu$, each 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 $\triangle$. Similar problem splitting can be employed when s lies on an edge of one or two of the triangles in G. It is easily checked that the modified algorithm produces the desired shortest path tree in linear time.

An Extended Algorithm. The algorithm described above can be extended to produce information regarding shortest paths from the source point s to arbitrary points inside P. We shall describe such an extension, which 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. The extended algorithm is based on Lemma 2.1 given above and runs as follows.

Let e be an edge of P, and let $\Phi(e)$ be the corresponding region of P as defined in Lemma 2.1. Assume that the funnel F_e has the form $[u_l, u_{l-1}, \ldots, u_1, a, w_1, \ldots, w_k]$ with $a = u_0 = w_0$ as its cusp (thus $e = u_l w_k$). Then, for each $i = 0, \ldots, l-1$ (respectively for each $i = 0, \ldots, k-1$) the ray emanating from u_i (respectively from w_i) and passing through u_{i+1} (respectively through w_{i+1}) hits e, and its portion between e and u_i (respectively 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 2.1(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 2.1(b), is $O(n)$. We can then use either one of the linear-time algorithms of [18] and [5] 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 $O(1)$ additional time; the path itself can be calculated in $O(1)$ additional time per segment on the path, by simply traversing the path in the shortest path tree from q to s. Thus we have the following result:

THEOREM 2.2. *Given a triangulated simple polygon P with n sides and some source point s within P, one can preprocess P in linear time so that, for any 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 extracted in time $O(\log n + k)$, where k is the number of segments from which this path is composed.

REMARK. Related work on shortest paths inside a simple polygon can be found in [7].

2.1. The Convex Rope Algorithm. As an immediate and relatively simple application of the shortest path algorithm just presented, we consider the *convex rope problem*, posed by Peshkin and Sanderson [24]: let P be a simple polygon, let s be a vertex of P lying on the convex hull of P, and 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 clockwise angle from the directed segment $p_{i-1}p_i$ to the directed segment p_ip_{i+1} is less than π, for $i = 1, \ldots, m-1$. The counterclockwise convex rope from s to v is defined in a symmetric manner (see Figure 2.2). Not all the vertices of P need have convex ropes from s. The vertices v that have both clockwise and counterclockwise ropes from any hull vertex s are precisely those that are "visible from infinity" (see [24]); calculation of these convex ropes is required in [24] to plan reachable grasps of P by a simple robot arm. An $O(n^2)$ algorithm is presented in [24]. Using the shortest path algorithm given above, we obtain an improved algorithm running in linear time, plus the time for triangulating P.

The convex rope problem can be solved as follows. First compute the convex hull of P in linear time (see [22] and [12]). The clockwise (respectively counterclockwise) convex rope from s to any vertex v on the convex hull can now be calculated by moving along the convex hull in a clockwise (respectively counterclockwise) direction from s to v. For each vertex n 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 Figure 2.2; note also that the collection of all these bays can be found in linear time).

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 (respectively counterclockwise) convex rope from s to v is then the clockwise convex rope from s to v_1 (respectively the counterclockwise convex rope from s

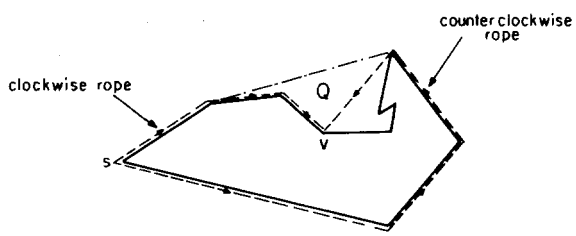

Fig. 2.2. The convex rope problem.

to v_2) followed by the shortest path from v_1 to v (respectively from v_2 to v) within Q, provided that this shortest path is clockwise (respectively counterclockwise) convex. (Otherwise the required convex rope does not exist.) Hence we can use the shortest path tree algorithm presented above 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.

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 [9], [19], [2], [4], [6], [20], [1], and a variety of algorithms have been developed to solve them. Some of the simpler problems already have linear-time solutions, whereas others only have $O(n \log n)$ solutions. Here we present linear-time solutions for all these problems, again based on the availability of a triangulation of P. Our approach relies on an interesting relationship between visibility and shortest path problems.

Let P be a triangulated 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* $\mathrm{Vis}(P, x)$ 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 there exist known linear-time algorithms for it [8], [19].)

II. Given a segment e inside P, calculate the (weak) visibility polygon $\mathrm{Vis}(P, e)$ consisting of all $y \in P$ that are visible from some point on e. (An $O(n \log n)$ solution is given in [4]; Avis and Toussaint [2] present a linear-time algorithm for determining whether $\mathrm{Vis}(P, e) = P$.)

III. Given a segment e inside P, 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. (An $O(n \log n)$ solution is given in [4].)

IV. Given a segment e inside P, 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 [4].)

We shall first consider Problem I; although this problem already has linear-time solutions, it is worthwhile to sketch our solution as a preparatory step toward the solution of the more complicated problems (Problems II–IV). It is well known that $\mathrm{Vis}(P, x)$ is a simple polygon; each of its vertices is either a vertex of P that is visible from x or a "shadow" cast on the boundary of P by such a visible vertex (i.e., a point y visible from x such that the segment xy passes through a vertex of P; see Figure 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 triangle). Calculate the shortest path tree T from x using the

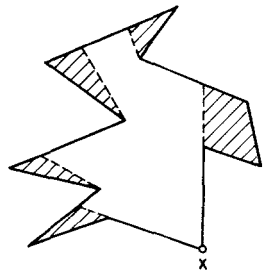

Fig. 3.1. Visibility of a polygon from a point.

(extended) algorithm given in Section 2. Then clearly the vertices of P visible from x are the 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 $\text{Vis}(P, x)$ by simply traversing the boundary of P, say in clockwise order, collecting all visible subsegments along this boundary, and replacing contiguous nonvisible portions of the boundary by straight segments connecting visible vertices with their shadows. Thus we have:

THEOREM 3.1. *The visibility polygon* $\text{Vis}(P, x)$ *can be calculated in linear time, given a triangulation of* P.

Next we consider Problem II. Let a, b be the endpoints of e. It is easily checked that in this case $\text{Vis}(P, e)$ is a simple polygon, each of whose vertices is either:

(i) a vertex of P visible from e; or
(ii) a shadow cast on the boundary of P by a ray that emanates from a or from b and passes through a vertex of P visible from that endpoint; or
(iii) a shadow cast by a ray r that emanates from some interior point on e and passes through two vertices x, y of P, such that the exterior 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 Figure 3.2 for an illustration of $\text{Vis}(P, e)$.

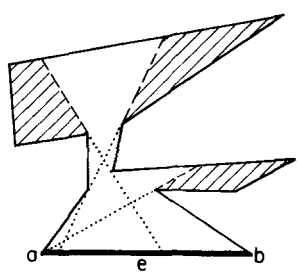

Fig. 3.2. Visibility of a polygon from an edge.

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, as in the case of funnels, 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 $\pi(b, d)$.

LEMMA 3.1. *If e' contains a point in its interior that is visible from e then (up to exchanging c and d) the two paths $\pi(a, c)$ and $\pi(b, d)$ are 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 indeed 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 also in P. The claim concerning $\pi(b, d)$ follows by a symmetric argument. □

In the situation described by the preceding lemma, we call the union of $\pi(a, c)$ and $\pi(b, d)$ the *hourglass* for the pair (e, e').

Suppose we apply the shortest path algorithm of Section 2 to the two source vertices a and b, and also compute, for each vertex c of P, whether the path $\pi(a, c)$ (respectively $\pi(b, c)$) is outward convex. (The latter 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 the two paths $\pi(a, d)$ and $\pi(b, c)$ are also 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 be the concatenation of three subpaths: a subpath $\pi(a, x)$ of $\pi(a, c)$, where x is a point lying on $\pi(a, c)$, a line segment xy, where y is a point lying on $\pi(b, d)$, and the subpath $\pi(y, d)$ of $\pi(b, d)$. Moreover xy must be a common tangent to both of the paths $\pi(a, c)$ and $\pi(b, d)$ (see Figure 3.3). The path $\pi(b, c)$ has a symmetric structure of the form $\pi(b, w) \| wz \| \pi(z, c)$ (where $\|$ denotes path concatenation),

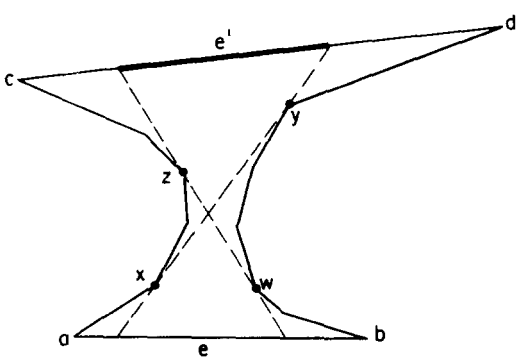

Fig. 3.3. Visibility of one edge of a polygon from another and the corresponding hourglass.

for appropriate points $w \in \pi(b, d)$, $z \in \pi(a, c)$. It follows that the subsegment of e' visible from e is that 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. Namely, as we execute the shortest path 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 the part of the funnel between x and d. The points w and z are found in a completely symmetric manner when running the shortest path algorithm with b as a source. To calculate $\mathrm{Vis}(P, e)$ we simply traverse the boundary of P, say in clockwise order, collecting all visible subsegments along this boundary in the manner explained above, and replacing contiguous nonvisible portions of the boundary by straight segments connecting visible vertices with their appropriate shadows. Thus we have:

THEOREM 3.2. (*The visibility polygon* $\mathrm{Vis}(P, e)$ *of* P *with respect to an edge* e *can be calculated in linear time* (*assuming a triangulation of* P *is given*).

REMARK. A similar connection between shortest paths and visibility inside a simple polygon as well as some of the technical tools developed above have been obtained independently by Toussaint [31].

Next let us 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 of the paths $\pi(a, x)$ and $\pi(b, x)$ 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 $\pi(a, x)$ (respectively in $\pi(b, x)$) be cx (respectively dx) for some vertex c (respectively 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 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 associated with 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 [18] or [5]. In addition, we store at each vertex c of P, flags indicating whether the paths $\pi(a, c)$ and $\pi(b, c)$ are outward convex, and (as usual) also pointers to the two parents 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 we locate x in the two partions Π_1 and Π_2 and obtain the two corresponding influencing vertices c, d of P. Then, in additional 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 more general than the corresponding Problem P3 studied in [4], in that here the query point x can be any point inside P, whereas in [4] it is required to lie on the boundary of P.

Finally, we 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 [15] and [4]. Following [4], we will produce a partioning 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. The same partitioning is obtained in [4] in $O(n \log n)$ time; 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 know that the two paths $\pi(a, c)$, $\pi(b, d)$ are 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 follows 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, each vertex of $R(e')$ corresponds either to a ray that passes through two vertices of P that are adjacent in one of the subpaths $\pi(x, z)$, $\pi(w, y)$, or to one of 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 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 that are obtained during execution of the shortest path algorithm of Section 2. This sum is linear in n, which implies that Π has only $O(n)$ vertices (see also [4]).

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. 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. Having calculated Π, we next apply to it one of the linear-time preprocessing algorithms of [18] and [5] for point location, 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 General Shooting Problem in a Simple Polygon. In this section we show how, given a triangulated 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 [4, Section 3]. This problem calls 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 [4] solves the shooting problem in $O(\log n)$ time per query, after building a structure in $O(n \log n)$ time that requires $O(n)$ space. The contribution of this section is to show how to build the shooting structure used by [4] in linear time. It is assumed here that the reader is familiar with the construction described in that paper.

To start with, we need a balanced decomposition S of our simple polygon P; such a decomposition can be obtained by recursively subdividing the polygon according to Chazelle's [3] polygon-cutting theorem. His algorithm actually calculates a balanced decomposition of the dual tree T of a triangulation of P (a precise definition of "balanced" is given below), and runs in time $O(n \log n)$. For our purposes this is not sufficiently fast, so in the Appendix we present a linear-time procedure for decomposing an arbitrary binary tree in a balanced fashion. Such a decomposition is best described by a recursive process. At the top level we remove one edge and split our tree into two subtrees. Each of the subtrees is then split in the same way, and so on recursively until single-node trees remain. A decomposition of this type is called balanced if at each state the two fragments that arise after the edge removal have size which is at least some fixed fraction α of the size of the tree currently being split.

Let G denote the underlying triangulation of our polygon P. The balanced decomposition S of the dual tree of G obtained as described in the Appendix is most usefully thought of as a balanced tree. The leaves of the tree S correspond to the triangles of G; the internal nodes of S correspond to the diagonals in G. 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 until P is decomposed into triangles. 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 depth in the tree S. By convention the diagonal corresponding to the root s of the tree S has depth 1; the children of the root have depth 2, and so on.

As described in [4], 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 represented by the hourglass for the pair (d_1, d_2), as discussed in Section 3.

In [4] it is shown how to build a data structure for solving shooting queries in logarithmic time. This construction takes linear time and space once all edges belonging to the hourglasses are known.

We now describe a method for computing the edges of all the hourglasses in linear time, given the triangulation G and its associated balanced tree decomposition S. We will denote by Λ the set of all diagonal pairs delimiting hourglasses. It is easiest to construct these hourglass structures from the bottom up. Specifically, consider the following process, which we term the *merging process*: start with the underlying triangulation G, whose triangles are considered as the leaves of a balanced tree S as discussed above. For each triangle, at least two of whose sides are diagonals, add all pairs of its bounding diagonals to the set Λ of pairs to be considered (either one or three pairs get added per triangle). Now remove all diagonals of the largest depth in S from the triangulation G. This creates new regions by merging pairs of old regions (initially triangles). It cannot happen that three or more regions get merged into one, since diagonals with the same depth in S are never adjacent in G. If R_1, R_2 are two regions being merged, then add to the set Λ all pairs (d_1, d_2), where d_1 (respectively d_2) is a diagonal bounding R_1 (respectively R_2) and remaining after the merge. Continue this process, at each stage removing all diagonals of the currently largest depth and adding to the set 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, during the merging process, two diagonals of the same depth become bounding edges of the same region. 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 depth.

In S, the balanced decomposition of the dual tree of G, 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-depth 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 in S. 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 in S.

Let S^* denote the graph formed from the decomposition tree by adding all edges corresponding to the diagonal pairs in Λ (all edges of S joining internal nodes 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 depth and the side of e it lies on—it is the diagonal of T "nearest" to e of the right depth and on the appropriate side of e. 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, both elements of which correspond to nodes in the subtree of S rooted at e. Then the above remarks imply 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 the root of S and its left and right children, respectively, 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, $0 < \alpha < \frac{1}{2}$, such that the subtrees of each node x of S are of size at least the fraction α of the size of the whole tree rooted at x. By standard techniques one can prove by induction that $\mu(s) = O(|S|) = O(n)$. This proves that Λ, and therefore S^*, has linear size.

As we remarked at the beginning of the section, 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 outward convex chains defined by the hourglass illustrated in Figure 4.1. These chains are the convex hulls of the two polygonal paths joining e and g along P.

Suppose that we are in the midst of the merging 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 outward convex chains in the hourglasses for (e, f) and (f, g). This is illustrated in Figure 4.1. 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, not counting repetitions, is proportional to the number of diagonal pairs in Λ, i.e., it is $O(n)$.

Recall our convention to write each diagonal pair (e, g) in Λ so that e is the ancestor and g the descendant in S. The size of the hourglass of (e, g) is bounded by the number $w(e, g)$ of triangles of T contained in the part of P between e and g. We claim that $\log w(e, g) = O(\tau(e) - \lambda(e))$, because in a balanced decomposition the height of each subtree is logarithmic in its size. Thus 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 $\kappa(e) = O(\Sigma(\tau(g) - \lambda(g))^2)$, where the sum ranges over all descendants

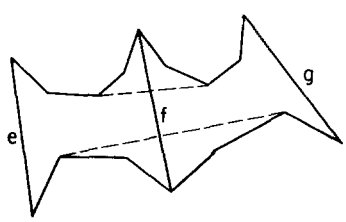

Fig. 4.1. Merging two hourglasses into a new hourglass.

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 prove (again using induction) that this cost is also $O(n)$.

In order to attain a linear size shooting structure, we cannot afford to store explicitly all the hourglasses (because of potential edge duplication). As explained in [4], each hourglass edge is allocated to, and stored only at, that hourglass in S^* that has it as an edge and is 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 to find the common tangents discussed above, and then split the old hourglasses where the common tangents touch them. The extremal hourglass pieces get jointed by the tangent to form the new hourglass, while the inner pieces are left associated with the old hourglasses. See Figure 4.1 for an illustration. This splitting and joining can be implemented in time of the same order of magnitude as that of the common tangent computation, if a balanced tree structure is used to represent the hourglasses. In this way, continuing exactly as in [4] the entire shooting structure of [4] can be computed in linear time and space.We have therefore shown that:

THEOREM 4.1. *Given a triangulated simple polygon P of n sides, it is possible to construct, in linear time and space, an auxiliary structure with which the shooting problem for P* can be solved in $O(\log n)$ *time per query. These bounds are optimal.*

Note: We have been able to avoid the use of finger search trees in the above construction because we started out with a balanced decomposition. We could have obtained a shooting structure in linear time and space from *any* decomposition S of P, whether balanced or not. This requires the use of finger search trees in a manner analogous to that of Section 2. However, the resulting shooting structure S^* no longer guarantees logarithmic search time, as there is no logarithmic bound on its depth.

5. Conclusion. We have presented a collection of linear-time algorithms for solving a variety of shortest path and visibility problems inside a triangulated simple polygon, exploiting interesting relationships between these two types of problems. Our work has enlarged the collection of problems solvable in linear time for triangulated simple polygons; we expect many additional problems for such polygons to have linear-time solutions. For example, Suri [28] 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 $P_1, P_2, \ldots$ 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.

Our results indicate once again the centrality to planar computational geometry of the question of whether there exists a linear-time triangulation algorithm for simple polygons.

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

Appendix. 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 F is partitioned by the removal of an edge, each of the two fragments obtained has size at least $\alpha|F|$, where $|F|$ denotes the size (number of nodes) of F. We make two remarks on such decompositions. First, it is clear that in a balanced decomposition the fragment containing a particular node v can be split only $O(\log n)$ times. Second, 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 [3]. As a result, a balanced decomposition of T with $\alpha = \frac{1}{4}$ can be found in $O(n \log n)$ time total, by recursively applying the centroid partition to each fragment.

In this appendix we will show how a centroid edge for partitioning each fragment can be computed in only $O(\log n)$ time, after some appropriate data structures have been set up. The overall time for obtaining a balanced decomposition will then be reduced to $O(n)$. Our method makes use of an auxiliary ternary tree A^T, for brevity written A, to facilitate the splitting. The tree A has the same nodes as T, but whereas 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 in logarithmic time an appropriate edge at which T should be split. Moreover, in the same time bound, we can break A into two auxiliary trees, one for each of the two fragments of T resulting from the split. By applying this process recursively we obtain the desired balanced decomposition of T.

In this appendix 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, all logarithms are base 2.

To define the auxiliary tree A, we need to attach to the nodes of T certain integer labels. We associate with each node v of T (and of A) an integer *index* i_v and an integer *label* b_v. The index i_v will represent the height of v in A and will also equal the number of trailing zeros in b_v written in binary. Formally, b_v

is 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 constructed by the following recursive procedure: let i be the position of the leftmost carry in the computation of $b_w + b_z + 1$ (one can easily check that the addition order does not matter). 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 (or taking the "exclusive or" of) the corresponding bits of b_w and b_z; note also that the ith bit of b_v must be 1.) Finally, i_v is defined to be the number of trailing zeros of b_v.

We now present an equivalent definition of these labels and indices which provides some intuition as to the properties of these values related to the tree T. 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 whose index is 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 defined to be 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 or even of a single vertex). The index i_v of node v is given in terms of the indices of the descendants of v in T (including v): it is the smallest nonnegative 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 with index j is reachable from v along a path of index less than j). The index of a leaf is taken to be 0. The intuition behind this somewhat obtuse definition can be seen by considering the case where T is one long path, as in Figure A.1(a). In this case the requirement stated produces the familiar "ruler function." For a more general example of indices and labels, see Figure A.1(b).

The labels b_v can now be given the following interpretation. We treat b_v as a bit vector: $b_v = \sum_{j \geq 0} 2^j b_v[j]$, where $b_v[j] = 0$ or 1. The entry $b_v[j]$ is 1 if and only if some (and therefore 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 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

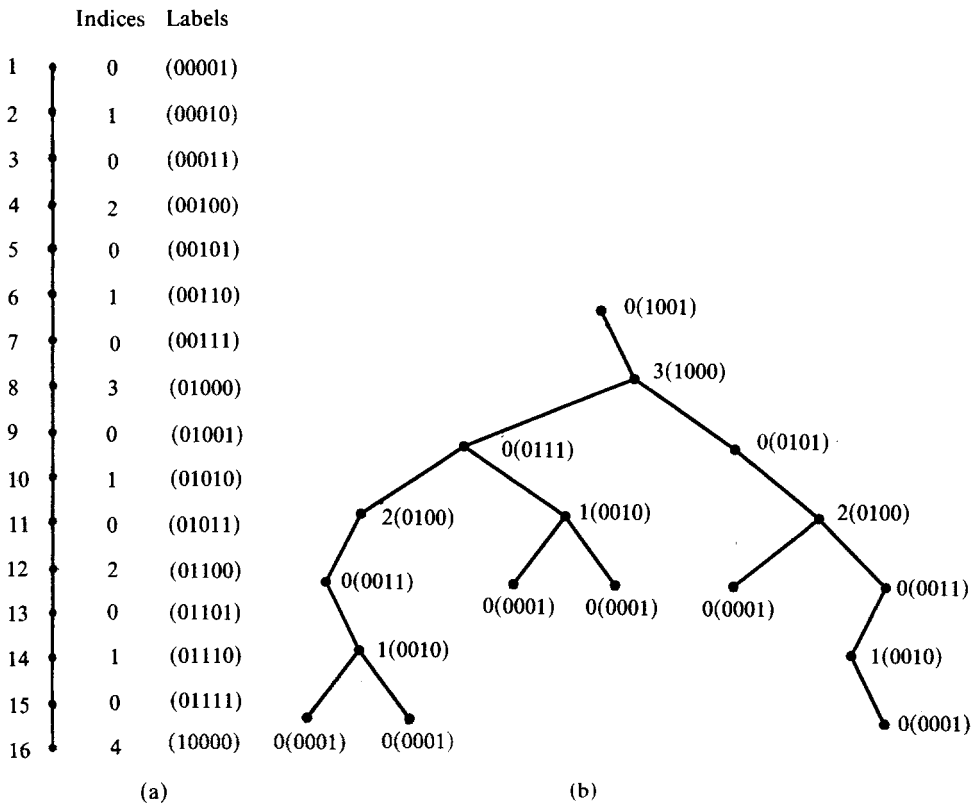

Fig. A.1. (a) Indices giving the ruler function and the labels that produce them. (b) Indices and labels of nodes of a more general tree T. Each node is labeled with $i_v(b_v)$

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:

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

In the linear chain example of Figure A.1(a), the b_v's are just consecutive *integers* written in binary. Note that these definitions accord with the ones given at the beginning of the appendix. 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 single 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 A.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 maximum 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 a 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 A.2

(a) *For each node v, $|T_v| \geq b_v$.*
(b) *For each index j, there are at most $\lfloor |T|/2^j \rfloor$ 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 the descendants of v whose indices are at most i_v and which are joined to v by paths of index less than i_v. Plainly, besides v, no node with index i_v can be an ancestor of one of these nodes w and reach w 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 the 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_vth 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 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 linear additional time, a traversal of A can be used to compute $|A_v|$ and store it at each node v; this information is needed in order to split the auxiliary tree later on.

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, in time proportional to the height of A, an appropriate e_c to remove. Deleting 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 for 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 these two new auxiliary trees is relatively straightforward. Let v be the endpoint of e_c with smaller index. The path π 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 π is the root of A^B. The new auxiliary trees are constructed by dividing π into two monochromatic paths. In A^R and A^B every node w on π has as its successor the next node on π with the same color as w. The counts $|A_w|$ are still valid in A^R and A^B except at nodes w on π, where they must be recomputed. These changes take time proportional to the length of the path π. To see that these modifications of A are sufficient, observe that at most one tree A_v is dichromatic for nodes v of a given index, and that after π is modified, each node has auxiliary tree links only to nodes of its own color. Now 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 durling the recursion. Therefore the whole decomposition takes time

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

The preceding discussion constitutes a proof of the following theorem:

THEOREM A.1. *It is possible to find a balanced decomposition of an arbitrary binary tree in linear time.*

REMARK. An alternative $O(n)$-time technique for finding a balanced tree decomposition is obtained by using a simplified version of the dynamic tree data structure (as described in Chapter 5 of [29]), which supports logarithmic-cost tree-splitting operations.

References

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

[2] D. Avis and G. T. Toussaint, An optimal algorithm for determining the visibility of a polygon from an edge, *IEEE Trans. Comput.*, **30** (1981), 910–914.

[3] B. Chazelle, A theorem on polygon cutting with applications, *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, 1982, pp. 339–349.

[4] B. Chazelle and L. Guibas, Visibility and intersection problems in plane geometry, *Proceedings of the ACM Symposium on Computational Geometry*, 1985, pp. 135–146 (also submitted to *Discrete Comput. Geom.*).

[5] H. Edelsbrunner, L. Guibas, and J. Stolfi, Optimal point location in monotone subdivisions, Technical Report 2, DEC/SRC, Palo Alto, CA, 1984.

[6] H. A. El Gindy, An efficient algorithm for computing the weak visibility polygon from an edge in simple polygons, Manuscript, McGill University, 1984.
[7] H. A. El Gindy, Hierarchical decomposition of polygons with applications, Ph.D. Dissertation, School of Computer Science, McGill University, 1985.
[8] H. A. El Gindy and D. Avis, A linear algorithm for computing the visibiility polygon from a point, *J. Algorithms*, **2** (1981), 186–197.
[9] S. Fisk, A short proof of Chvatal's watchman theorem, *J. Combin. Theory Ser. B*, **24** (1978), 374.
[10] A. Fournier and D. Y. Montuno, Triangulating simple polygons and equivalent problems, *ACM Trans. Graphics*, **3** (1984), 153–174.
[11] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, Triangulating a simple polygon, *Inform. Process. Lett.*, **7** (1978), 175–179.
[12] R. L. Graham and F. F. Yao, Finding the convex hull of a simple polygon, *J. Algorithms*, **4** (1983), 324–331.
[13] D. H. Greene and D. E. Knuth, *Mathematics for the Analysis of Algorithms*, Birkhäuser, Boston, 1982.
[14] L. Guibas, E. McCreight, M. Plass, and J. Roberts, A new representation for linear lists, *Proceedings of the Ninth ACM Symposium on Theory of Computing*, 1977, pp. 49–60.
[15] L. Guibas, L. Ramshaw, and J. Stolfi, A kinetic framework for computational geometry, *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, 1983, pp. 100–111.
[16] P. T. Highman, The ears of a polygon, *Inform. Process. Lett.*, **15** (1982), 196–198.
[17] S. Huddleston and K. Mehlhorn, A new data structure for representing sorted lists, *Acta Inform.*, **17** (1982), 157–184.
[18] D. Kirkpatrick, Optimal search in planar subdivisions, *SIAM J. Comput.*, **12** (1983), 28–35.
[19] D. T. Lee, Visibility of a simple polygon, *Comput. Vision, Graphics Image Process.*, **22** (1983), 207–221.
[20] D. T. Lee and A. Lin, Computing the visibility polygon from an edge, Manuscript, Northwestern University, 1984.
[21] D. T. Lee and F. P. Preparata, Euclidean shortest paths in the presence of rectilinear battiers, *Networks*, **14** (1984), 393–410.
[22] D. McCallum and D. Avis, A linear algorithm for finding the convex hull of a simple polygon, *Inform. Process. Lett.*, **9** (1979), 201–206.
[23] K. Mehlhorn, *Data Structues and Efficient Algorithms*, Vol. I, Springer-Verlag: Berlin, 1984.
[24] 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. Also *IEEE J. Robotics Automat.* (1986) (to appear).
[25] F. P. Preparata and M. I. Shamos, *Computational Geometry*, Springer-Verlag, New York, 1985.
[26] F. P. Preparata and K. J. Supowit, Testing a simple polygon for monotonicity, *Inform. Process. Lett.*, **12** (1981), 161–164.
[27] D. D. Sleator and R. E. Tarjan, Self-adjusting binary search trees, *J. Assoc. Comput. Mach.*, **32** (1985), 652–686.
[28] S. Suri, Finding minimum link paths inside a simple polygon, *Comput. Vision, Graphics Image Process.* (1986) (to appear).
[29] R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS–NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathemataics, Philadelphia, PA, 1983.
[30] R. E. Tarjan and C. Van Wyk, An $O(n \log \log n)$ time algorithm for triangulating simple polygons, *SIAM J. Comput.* (submitted).
[31] G. Toussaint, Shortest path solves edge-to-edge visibility in a polygon, Technical Report SOCS-85.19, McGill University, 1985.