ELSEVIER

Computational Geometry 7 (1997) 37–57

Computational Geometry Theory and Applications

LR-visibility in polygons

Gautam Das [1], Paul J. Heffernan, Giri Narasimhan

Mathematical Sciences Department, The University of Memphis, Memphis, TN 38152, USA

Communicated by Anna Lubiw and Jorge Urrutia; submitted 29 September 1993; accepted 13 June 1995

Abstract

We give a linear-time algorithm which, for a simple polygon P, computes all pairs of points s and t on P that admit LR-visibility. The points s and t partition P into two subchains. We say that P is LR-visible with respect to s and t if each point of P on the chain from s to t is visible from some point of the chain from t to s and vice-versa.

Keywords: Polygonal visibility; Weak visibility

1. Introduction

We consider here the *LR-visibility* problem for simple polygons. Let P be a simple polygon represented by a simple, closed, polygonal chain. Any two points s and t on P partition P into two subchains, which we call L and R, for left and right chains. The LR-visibility question asks whether each point of L can see a point of R, and whether each point of R can see a point of L. If the answer is yes, we say that P is *LR-visible with respect to s and t*. Fig. 1 shows a polygon that is LR-visible with respect to some pair of points (or simply LR-visible), and Fig. 2 one that is not LR-visible with respect to any pair of points (or not LR-visible).

We state four versions of the LR-visibility problem for a polygon P:

1. determine whether a given pair s and t admits LR-visibility;
2. determine whether there exists a pair s and t which admits LR-visibility;
3. return a pair s and t which admits LR-visibility, if indeed such a pair exists;
4. return *all* pairs s and t which admit LR-visibility.

Version (4) is the strongest, and an algorithm for it also solves the first three versions. In this paper we solve to optimality the strongest version: we give a $\Theta(n)$-time algorithm that computes all pairs of points s and t that admit LR-visibility for a simple polygon P with n vertices. The output is in the form of $O(n)$ pairs of subchains S_i and T_i, such that any pair of points $s \in S_i$ and $t \in T_i$ is a

[1] Supported in part by NSF Grant CCR-930-6822.

SSDI 0925-7721(95)00042-9

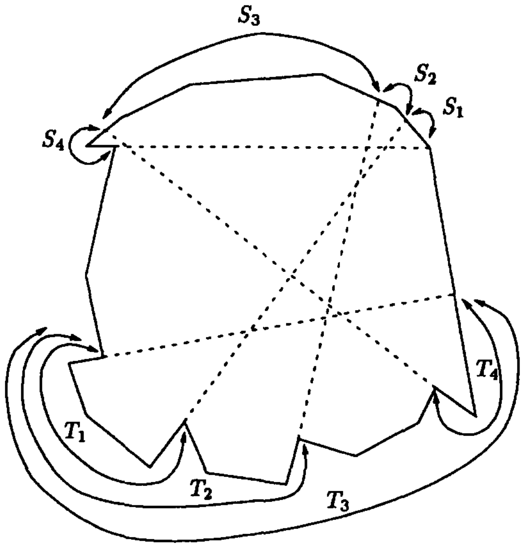

Fig. 1. An LR-visible polygon.

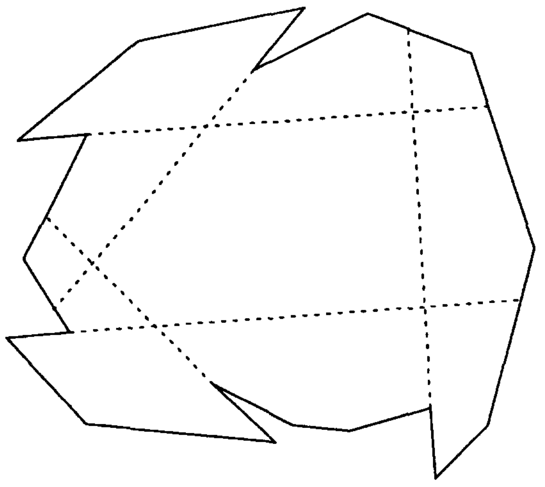

Fig. 2. A polygon which is not LR-visible.

valid pair s and t. Even though there may be an infinite number of LR-visible pairs, the output can be represented in $O(n)$ space. In Fig. 1, the output subchains $(S_1, T_1), \dots, (S_4, T_4)$ are shown.

The question of LR-visibility falls in the larger area of weak visibility in polygons. To say that two sets are weakly-visible means that *every* point in either set is visible from *some* point in the other set. A simple polygon P is weakly-visible from an edge e if e and $P \backslash e$ are weakly-visible. A weakly-visible chord c of P is one such that c and P are weakly-visible. A polygon is LR-visible with respect to s and t if its corresponding left chain L and right chain R are weakly-visible.

One interesting class of weak visibility problems is obtained by investigating paths that one or more guards can traverse along so that every point of a simple polygon is visible from some point on the paths. In this sense, for a polygon P every LR-visible pair of points (s,t) defines such a path (either the left (or right) subchain from s to t) for one guard.

Weak-visibility has received much attention from researchers. The term was first introduced by Avis and Toussaint [1], who gave a linear-time algorithm which determines whether a polygon is weakly-visible from a given edge. Sack and Suri [14] gave a linear-time algorithm which for a polygon computes all weakly-visible edges, and Chen [3] gave an optimal parallel algorithm for this problem. Chen has also solved the problem of computing the shortest subedge (connected subset of an edge) from which a polygon is weakly-visible [4]. Ke [11] and Doh and Chwa [7] have given $O(n \log n)$-time algorithms which determine whether a polygon has a weakly-visible chord, and if so construct one; Ke's algorithm is able to return the shortest such chord. Recently, Das et al. [6] have produced an $O(n)$-time algorithm which constructs *all* weakly-visible chords.

Heffernan [9] has given a linear-time algorithm which determines whether P is LR-visible with respect to a pair of points s and t. An $O(n \log n)$-time method that computes all LR-visible pairs s and t is given by Tseng and Lee [15]. In both papers the authors are actually addressing a more complex problem is polygonal visibility called the two-guard problem, of which LR-visibility is a subproblem. Their results imply that before this paper, the weakest form of the LR-visibility problem, version (1) above, has been solved to optimality, while the strongest form, version (4), had been solved with an $O(n \log n)$ time-bound. This paper solves version (4) to optimality by presenting a $\Theta(n)$-time algorithm.

This paper is of interest not only because we present an optimal result for an intriguing problem in polygonal visibility, but also on account of the techniques we employ, and because of the relationship between LR-visibility and other problems in polygonal visibility, such as the two-guard problem and the weakly-visible chord problem. LR-visibility is a subproblem of the weakly-visible chord problem, in the sense that a chord c partitions P into two subchains which must be weakly-visible in order for c to be a weakly-visible chord. In a recent paper [6], the authors exploit this relationship by using the result of the present paper as a subprocedure.

A similar relationship exists for the two-guard problem. While it has many formulations, we will state just one for the sake of illustration: a polygon P is walkable from point s to point t if one "guard" can traverse the left chain L and the other the right chain R from s to t while always remaining covisible. Other formulations require that the guards to move monotonically or that one guard traverses from t to s. As for LR-visibility, different versions of each formulation exist, such as testing a fixed pair s and t, determining if a pair s and t exists, and finding one or all pairs s and t. LR-visibility is a subproblem of two-guard, in the sense that a polygon must be LR-visible with respect to s and t in order to be walkable for that pair. Thus, algorithms which solve a version of the two-guard problem must also solve a version of LR-visibility, and some examples were listed above. Since the present paper solves the strong version of LR-visibility to optimality, it may prove to be an important step towards producing stronger results for the two-guard problem, just as it has already for the weakly-visible chord problem.

The paper is organized as follows. Section 2 describes the notation for the paper, as well as some of the preprocessing that is needed by the algorithm. Section 3 introduces some properties of LR-visible pairs of points. Section 4 contains the heart of the paper and describes how to compute all nonredundant components, while Section 5 uses that to compute all LR-visible pairs of points. Our conclusions are summarized in Section 6.

2. Preliminaries

We define notation for this paper. A *polygonal chain* in the plane is a concatenation of line segments. The endpoints of the segments are called *vertices*, and the segments themselves are *edges*. If the segments intersect only at the endpoints of adjacent segments, then the chain is *simple*, and if a polygonal chain is closed we call it a *polygon*. In this paper, we deal with a simple polygon P of n vertices, and its interior, $\mathrm{int}(P)$. The segment between two points x and y is denoted $\overline{xy}$, and $\mathrm{int}(\overline{xy}) = \overline{xy}\backslash\{x, y\}$. Two points $x, y \in P$ are *visible* (or *covisible*) if $\overline{xy} \subset P \cup \mathrm{int}(P)$. The (Euclidean) distance between two points x and y is denoted $\mathrm{dist}(x, y)$. We assume that the input is in general position, which means that no three vertices are collinear, and no three lines defined by edges intersect in a common point.

If x and y are points of P, then $P_{\mathrm{CW}}(x, y)$ ($P_{\mathrm{CCW}}(x, y)$) is the subchain obtained by traversing P clockwise (counterclockwise) from x to y. The subchains $P_{\mathrm{CW}}(x, y)$ and $P_{\mathrm{CCW}}(x, y)$ includes their endpoints x and y. Subchains may also be denoted by S_i or T_i (which is the notation used for representing the output of the algorithm of the paper). These subchains are assumed to be counterclockwise subchains, i.e., as we traverse from their starting points to their ending points, we would be traversing along P in the counterclockwise direction. Their starting and ending points will also be called their *left* and *right* endpoints, respectively.

We let $d(x, y)$ denote the direction of a ray or line from x through y, and $\vec{r}(x, \alpha)$ represent the ray rooted at x in direction α. Two rays with common endpoint x partition the plane into two regions, each of which is the union of a set of rays with endpoint x. A *cone* is defined as the region containing all rays encountered as we sweep counterclockwise from $\vec{r}(x, y_1)$ to $\vec{r}(x, y_2)$, and is denoted as $\mathrm{cone}(d(x, y_1), d(x, y_2))$ (or $\mathrm{cone}(y_1, x, y_2)$) (see Fig. 3). We can also think of a cone as an interval of directions. We write $\mathrm{int}(\mathrm{cone}(y_1, x, y_2))$ to represent $\mathrm{cone}(y_1, x, y_2)\backslash\{\vec{r}(x, y_1), \vec{r}(x, y_2)\}$, a cone minus its boundary directions. For a vertex x of P, let x^+ be the vertex adjacent to x in the clockwise direction, and x^- the vertex adjacent in the counterclockwise direction. For a point $w \in P$ which is not a vertex, let w^- and w^+ be the endpoints of the edge containing w, in the clockwise and counterclockwise directions, respectively.

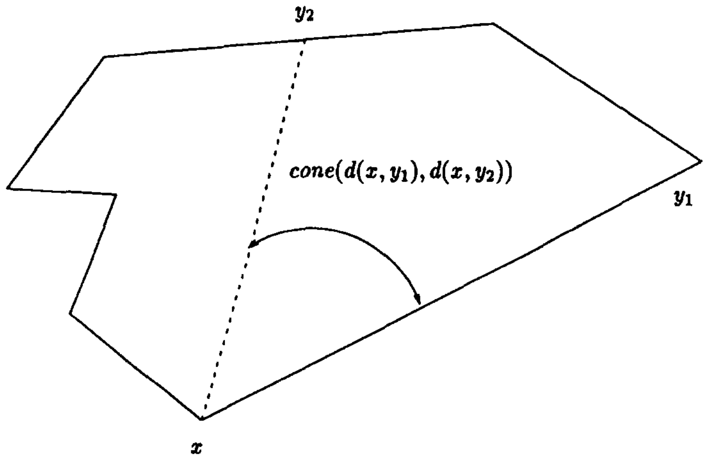

Fig. 3. Definition of a cone.

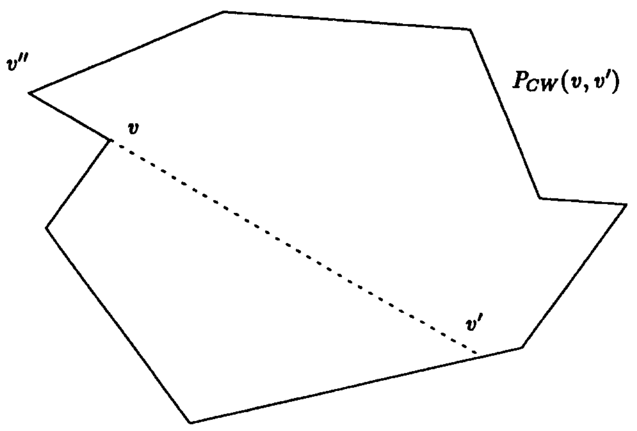

Fig. 4. A clockwise component.

The *ray shot* from a point $x \in P$ in direction α consists of "shooting" a "bullet" from x in direction α which travels until it hits a point of P. Formally, for a ray $\vec{r}(x, \alpha)$ rooted at x, where $\alpha \in \operatorname{int}(\operatorname{cone}(x^+, x, x^-))$, the *hit point* of this ray shot is the point of $(P \backslash \{x\}) \cap \vec{r}(x, \alpha)$ closest to x. We will sometimes denote a ray shot by writing its corresponding ray. Note that the ray shot $\vec{r}(x, \alpha)$ is defined only if $\alpha \in \operatorname{int}(\operatorname{cone}(x^+, x, x^-))$.

Each reflex vertex defines two special ray shots as follows. We let $\vec{r}_{\mathrm{CW}}(v) = \vec{r}(v, d(v^-, v))$ represent the *clockwise ray shot* from v. If v' is the hit point of the clockwise ray shot, then the subchain $P_{\mathrm{CW}}(v, v')$ is the *clockwise component* of v (see Fig. 4). Counterclockwise ray shots and components are defined in the same way. A component is *redundant* if it is a superset of another component.

We say that a set of points $x_1, \ldots, x_k$ on P appear in *counterclockwise order* if, starting at x_1, a counterclockwise traversal of P encounters the points in the order given. A counterclockwise order is not unique because of the starting point; e.g., the following orders are equivalent: (1) x, y, z; (2) y, z, x; (3) z, x, y.

The *shortest path* between two vertices w and v of P, denoted $\mathrm{SP}(w, v)$, is the (Euclidean) minimum-distance curve with endpoints w and v lying entirely in $P \cup \operatorname{int}(P)$. Shortest paths are unique. This means that two shortest paths cannot cross twice, since this would imply distinct shortest paths between a pair of points. The path $\mathrm{SP}(w, v)$ is always a polygonal chain, whose interior vertices are also vertices of P. This can be seen by a local analysis: if one of the above two conditions is violated, some small amount of local improvement is possible. By a similar argument, we have the following.

Lemma 1. *If w and v are vertices of P, and $\mathrm{SP}(w, v)$ is the shortest path directed from w to v, then any vertex of $\mathrm{SP}(w, v) \backslash \{w, v\}$ that lies on $P_{\mathrm{CW}}(w, v)$ is a left turn, while a vertex of $\mathrm{SP}(w, v)$ on $P_{\mathrm{CCW}}(w, v)$ is a right turn.*

We write $\mathrm{FE}(w, v)$ to denote the *first edge* of $\mathrm{SP}(w, v)$; that is, the edge of $\mathrm{SP}(w, v)$ incident to w. The direction of this edge away from w is denoted $\mathrm{dFE}(w, v)$. The *parent* of w is the vertex of $\mathrm{SP}(w, v)$ adjacent to w; in other words, it is the other endpoint of $\mathrm{FE}(w, v)$. The following is a simplification of a lemma established in [9].

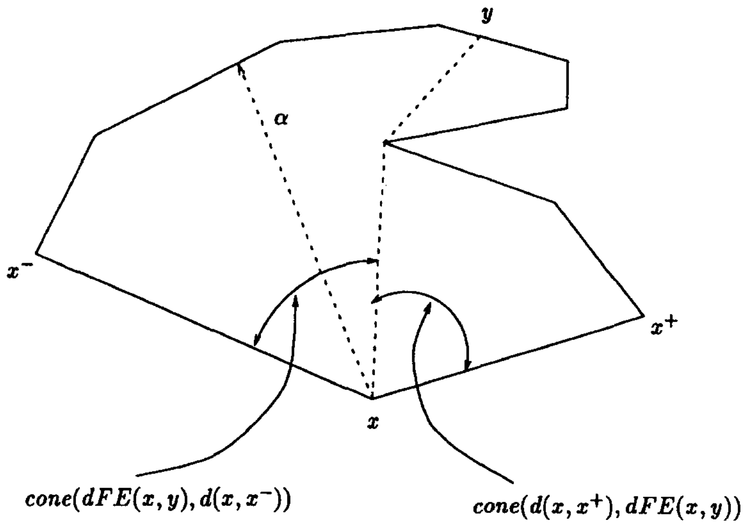

Fig. 5. Proof of Lemma 2.

Lemma 2. *Given points x and y of P, and a direction $\alpha \in \text{int}(\text{cone}(x^+, x, x^-))$, the ray shot $\vec{r}(x,\alpha)$ hits $P_{\text{CCW}}(x,y)$ if $\alpha \in \text{int}(\text{cone}(d(x,x^+), \text{dFE}(x,y)))$. Also, if $\alpha \in \text{int}(\text{cone}(\text{dFE}(x,y), d(x,x^-)))$ then it hits $P_{\text{CW}}(x,y)$.*

Proof. By a slight modification of the proof of [9, Lemma 3] (see Fig. 5). □

We define an *order query* as follows. Given distinct vertices x and y of P and given a direction $\alpha \in \text{int}(\text{cone}(x^+, x, x^-))$. Let x' denote the hit point of the ray shot $\vec{r}(x,\alpha)$. An order query answers whether x' lies on $P_{\text{CCW}}(x,y)$ or $P_{\text{CW}}(x,y)$. In other words, it tells whether the three points obey the counterclockwise order x, x', y or x, y, x'. If we perform an order query with a reflex vertex x and with either the clockwise or counterclockwise ray shot from x, then x' is not a vertex by the general position assumption. Thus, if y is also a vertex, the order query has only one correct response.

To answer order queries efficiently, our algorithm uses shortest path information. The *shortest path tree* from a vertex v of P, denoted SPT(v), is the union of all shortest paths SP(v,w), for w a vertex of P. For a simple polygon P, the shortest path tree from a vertex v can be constructed in $O(n)$ time [8]. In [9], a method is described which for a vertex v allows one to return FE(w,v) and dFE(w,v) for any point $w \in P$ in $O(1)$ time, after $O(n)$ preprocessing time (note that $w \in P$ is not necessarily a vertex). As a subprocedure this method constructs SPT(v) by using the algorithm of [8].

Being able to return dFE(w,v) for any point $w \in P$ in constant time after linear-time preprocessing means (by Lemma 2) that one can perform order queries from a vertex v in constant time. When this happens, we say that "a polygon P is preprocessed for order queries from vertex v".

A similar result holds even for subpolygons of P. Let x and y be points of P such that $\overline{xy}$ is a chord, and let $P' = P_{\text{CCW}}(x,y) \cup \overline{xy}$. For two points v and w in P', the shortest path between them in P' is the same as their shortest path in P. Consequently, if for a given vertex $v \in P'$ we perform the preprocessing step of [9], then we can perform constant-time order queries within P (not merely P') for v and ray shots from a point $w \in P_{\text{CCW}}(x,y)$. The preprocessing time is proportional to the number of vertices of P', which is the number of vertices on $P_{\mathrm{CCW}}(x,y)$. When this happens, we say that "the subpolygon P' is preprocessed for order queries from v." We summarize below.

Lemma 3. *Given a subpolygon P' of P (which may or may not equal P), and a vertex v of P'. If P' is preprocessed for order queries from v, then for any point w of P' and direction α such that the ray shot $\vec{r}(w,\alpha)$ is defined in P', one can determine in $\mathrm{O}(1)$ time whether the ray shot hits $P_{\mathrm{CW}}(w,v)$ or $P_{\mathrm{CCW}}(w,v)$. The preprocessing of P' can be achieved in time linear in the number of vertices of P'.*

In the remainder of the paper, when we say that P' has been preprocessed for order queries from v, we mean that the preprocessing necessary to apply Lemma 3 has been performed.

3. Properties

In this section we describe some of the properties of LR-visible pairs of points.

As noted in [10], the family of components completely determines LR-visibility of P, since a pair of points s and t admits LR-visibility if and only if each component of P contains either s or t. The definition of redundant gives the following.

Lemma 4. *A polygon P is LR-visible with respect to s and t if and only if each nonredundant component of P contains either s or t.*

We now state a simple consequence of the above lemma.

Lemma 5. *A polygon P with three disjoint components is not LR-visible.*

Proof. Given a pair of points s and t, if a polygon has three disjoint components, then one of the components necessarily contains neither s nor t. By Lemma 4, the polygon cannot be LR-visible. □

Lemmas 4 and 5 explain why the polygon in Fig. 1 is LR-visible, while the polygon in Fig. 2 is not LR-visible. In the latter, any point s on a subchain S_i is LR-visible to any point t on the corresponding subchain T_i. We now describe S_i and T_i more rigorously. The endpoints of nonredundant components partition P into a collection of intervals that we call *basis intervals*, and denote $S_1,\ldots,S_k$, ordered counterclockwise. Note that components include their endpoints. Thus a basic interval may or may not contain either of its endpoints. A nondegenerate basic interval would contain its left (right) endpoint if and only if it is the left (right) endpoint of a component. A reflex vertex defines a degenerate basic interval consisting of a single point. By Lemma 4, all points of a basic interval form LR-visible pairs with the same collection of partners. Thus, we denote as T_i the set of points such that (x,y) is an LR-visible pair for all $x \in S_i$ and $y \in T_i$. In Fig. 1, the LR-visible partners of $S_1,\ldots,S_4$ have been shown, because the remaining basic intervals do not provide any new LR-visibility information.

Lemma 6. *T_i is a connected set; that is, it is either P, the empty set, or a non-empty subinterval of P composed of the union of adjacent basic intervals.*

Proof. By Lemma 4, T_i is either the entire polygon P, the empty set, or the union of basic intervals. In the last case, suppose the basic intervals are not adjacent. Them there exist points w, x, y and z in counterclockwise order, where $w, y \in T_i$ and $x, z \notin T_i$. Let $\mathcal{D}$ be the set of nonredundant components that do not intersect S_i. Since w and y are in T_i they intersect every component of $\mathcal{D}$; thus each component of $\mathcal{D}$ intersects at least one of x and z. If $x(z)$ were to intersect all components of $\mathcal{D}$ then $x(z)$ would be in T_i, so there must be at least one component of $\mathcal{D}$ which does not contain x (and thus contains z) and another which does not contain z (and thus contains x). These two components cover P (their union is P), so at least one of them intersects S_i, a contradiction. □

Lemma 7. *If, for a basic interval S_i, we have $S_i \cap T_i \neq \emptyset$, then $T_i = P$.*

Proof. Take a point $x \in S_i \cap T_i$. The pair (x, x) is LR-visible, which by Lemma 4 implies that x intersects all components, which implies that for any $y \in P$ we have that x and y together intersect all components and thus form an LR-visible pair. □

As the basic intervals $S_1, \ldots, S_k$ are ordered counterclockwise on P, it can be shown also that the sets of starting and ending points of $T_1, \ldots, T_k$ are also respectively ordered counterclockwise. In fact, as one moves counterclockwise from S_i to S_{i+1}, one either leaves or enters a nonredundant component, which may result in either the starting or ending endpoint of T_i moving counterclockwise in order to form T_{i+1}.

4. Nonredundant components

We discuss here our method for constructing all nonredundant components of a polygon P. The main tool is a procedure which produces a superset of all nonredundant clockwise components. A symmetric procedure does the same for counterclockwise components. From these two sets the nonredundant components in sorted order can be extracted, as we will now show.

Suppose we have a set of clockwise components which contains all the nonredundant ones. As we traverse P in clockwise order, we encounter a beginning point and an ending point of each component. Since the beginning points are vertices of P, they can be sorted in linear time. Suppose we traverse P twice counterclockwise. Each time we encounter a beginning point, we compare the ending point of the component to the ending point of the previous component; if the current component contains the previous component, then the current component is redundant and therefore is deleted from the list of components. We must traverse P twice since one of the first components considered may be redundant with respect to one of the last ones. After an analogous procedure is performed for counterclockwise components, we have two lists of components, each in sorted order, which can be merged and pruned of redundant components in linear time to obtain a sorted list of all nonredundant components.

Constructing the components with standard ray shooting techniques would yield a time bound of $O(n \log n)$, since each shot requires $O(\log n)$ time [5]. By strategically choosing not to construct certain redundant components, our algorithm is able to perform faster.

Throughout this section we employ the following notation: if v is a reflex vertex of P then v' is the other endpoint of the clockwise component rooted at v. Thus $P_{\mathrm{CW}}(v, v')$ is a clockwise component. We now give the procedure which constructs a superset of the nonredundant clockwise components.

4.1. Constructing all nonredundant clockwise components

The outline of the procedure is as follows. We fix a vertex x_0 of P, and initialize a set $S_{x_0} \leftarrow \emptyset$ which will contain clockwise components. We traverse P once counterclockwise from x_0 with a pointer a. Whenever a encounters a reflex vertex, we possibly compute a' and add the clockwise component $P_{\mathrm{CW}}(a,a')$ to S_{x_0}. The component $P_{\mathrm{CW}}(a,a')$ is not added to S_{x_0} only if $P_{\mathrm{CW}}(a,a')$ is redundant with respect to the component most recently added to S_{x_0}. Thus, at termination S_{x_0} is a superset of all nonredundant clockwise components.

At any moment of execution the procedure has a fixed vertex x, initially set to x_0, where P has been preprocessed for order queries from x. In addition to the pointer a used to find reflex vertices, the procedure maintains a pointer b which is used to find the clockwise hit points of the reflex vertices.

There is one caveat to the above description: the procedure may terminate early. However, we will show that early termination occurs only if P has three pairwise disjoint clockwise components, and we know from Lemma 5 that a polygon with three pairwise disjoint components has no LR-visible pairs. The possibility of early termination arises out of the need to maintain an invariant, namely that the points x, a, b occur in counterclockwise order. If a encounters a reflex vertex with $a' \in P_{\mathrm{CCW}}(x,a)$, then attempting to set b equal to a' will violate the invariant. This requires that we "restart" the procedure by setting x and b equal to a, and continuing the traversal with a using this new value of x. (The original value x_0 is stored, for if a reaches x_0 all clockwise ray shots have been examined and the procedure is complete.) Later we will show that if we are asked to "restart" a third time, then P has no LR-visible pairs (Corollary 1), which permits us to halt. The variable COUNT in the procedure keeps track of the number of restarts.

We now give the pseudocode.

```
1. Choose a vertex $x_0$; set COUNT $\leftarrow 0$ and $S_{x_0} \leftarrow \emptyset$.
2. Set $x \leftarrow x_0$.
3. Compute SPT($x$); set $a, b \leftarrow x$.
4. ($b = a$)
   Traverse $P$ counterclockwise with $a = b$ until $a$ encounters a reflex vertex or $a$ encounters $x_0$; if
   $a$ encounters $x_0$ then RETURN $S_{x_0}$.
5. ($a$ is a reflex vertex)
   Test whether $a' \in P_{\mathrm{CCW}}(x,a)$; if so, set $x \leftarrow a$ and COUNT $\leftarrow$ COUNT $+1$, and if COUNT $= 3$
   then RETURN "early termination: there exist three pairwise disjoint components" else go to (3).
6. ($a' \in P_{\mathrm{CCW}}(a,x)$, so we look for $a'$)
   Set closest($a$) $\leftarrow \infty$.
7. Traverse $P$ counterclockwise with $b$ until $b$ lies on $\vec{r}_{\mathrm{CW}}(a)$; test whether $b = a'$; if not then set
   closest($a$) $\leftarrow$ min{closest($a$), dist($a, b$)} and goto (7).
8. ($b = a'$)
   Add $P_{\mathrm{CW}}(a,b)$ to $S_{x_0}$; set $y, z \leftarrow a$ and $y', z' \leftarrow b$.
9. Construct SPT$_{P_y}(y')$.
10. ($b \neq a$ and $b = z'$)
    Traverse counterclockwise with $a$ until $a$ encounters a reflex vertex or $a$ encounters $y'$.
11. If $a = z' = y'$ then goto (4).
12. If $a = z' \neq y'$ then set $y \leftarrow z$ and $y' \leftarrow z'$ and goto (9).
```

```
13. ($a$ is a reflex vertex)
    Test whether $a' \in P_{\mathrm{CCW}}(x,a)$; if so, set $x \leftarrow a$ and COUNT $\leftarrow$ COUNT $+1$, and if COUNT $= 3$
    then RETURN "early termination: there exist three pairwise disjoint components" else goto (3).
14. ($a' \in P_{\mathrm{CCW}}(a,x)$)
    Test whether $a' \in P_{\mathrm{CCW}}(a,y')$; if so then goto (10).
15. ($a' \in P_{\mathrm{CCW}}(y',x)$)
    If $y = z$ (i.e., $y' = z'$) then goto (18).
16. (test whether $a' \in P_{\mathrm{CCW}}(y',z')$)
    If $\tau(a) \in \mathrm{SP}(z',\tau)\backslash\{\tau\}$ then goto (10).
17. Traverse $\mathrm{SP}(z',y')$ forwards with $\tau$ until reaching $\tau(a)$; if the direction of $\vec{r}_{\mathrm{CW}}(a)$ is to the right
    of $d(a,\tau(a))$ then goto (10).
18. ($a' \in P_{\mathrm{CCW}}(z',x)$, so we look for $a'$)
    Set closest$(a) \leftarrow \infty$.
19. Traverse counterclockwise with $b$ until $b$ lies on $\vec{r}_{\mathrm{CW}}(a)$; test whether $b = a'$; if not then set
    closest$(a) \leftarrow \min\{\mathrm{closest}(a), \mathrm{dist}(a,b)\}$ and goto (19).
20. ($b = a'$)
    Add $P_{\mathrm{CW}}(a,b)$ to $S_{x_0}$; construct $\mathrm{SP}(b,y')$ (if $z' = y'$ this is done directly and if $z' \neq y'$ this is
    done by first constructing $\mathrm{SP}(b,z')$ and merging it with $\mathrm{SP}(z',y')$); set $z \leftarrow a$ and $z',\tau \leftarrow b$;
    goto (10).
21. Output $S_{x_0}$.
```

4.2. Description

In this subsection we give an intuitive description of the above pseudocode procedure.

The procedure maintains a point x which we can think of as an anchor. Under a value of x, the pointers a and b are initialized to x, and they traverse counterclockwise about P. The invariant that x, a and b are in counterclockwise order is maintained. In order to verify the invariant as the values of a and b change, it is necessary that P be preprocessed for order queries from x.

Occasionally it will be no longer possible to maintain the invariant under the current value of x, because the pointer b is about to traverse past x. In this event the procedure "restarts" itself by updating the value of x to a. In the pseudocode, this occurs when step 5 or step 13 goes to step 3. The variable COUNT, initialized to zero, counts the number of "restarts." Since Corollary 1 (given below) shows that the third restart indicates that P has no LR-visible pairs, the procedure terminates with this conclusion if COUNT reaches the value three (steps 5 and 13).

We now describe the behavior of the procedure under a fixed value of x. We will refer to several tests and procedures which are described in the following subsection.

Initially a and b are equal to x. The base operation is a counterclockwise traversal of P with a. At any moment the points x, a, b are in counterclockwise order, but the point b may or may not equal a. At times b will not be equal to a because it has traversed counterclockwise on its own in order to find a hit point a' (steps 7 and 19). If $b = a$ then while a traverses we maintain $b = a$ (step 4). When a is a reflex vertex, the counterclockwise order a, b, a' is obeyed, where b may or may not equal a'.

If a encounters x_0 then all reflex vertices have been considered, thus the procedure returns S_{x_0} and is complete (step 4).

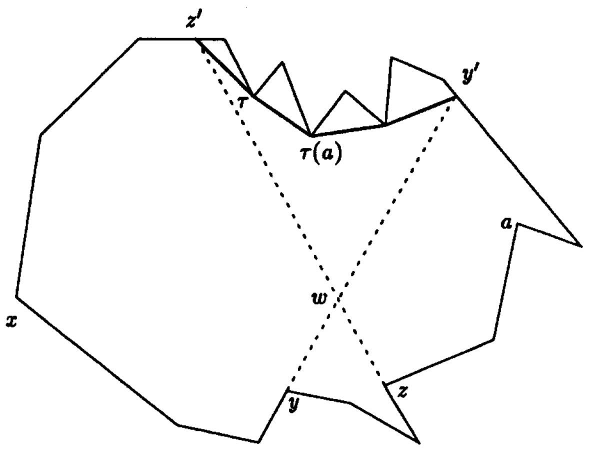

Fig. 6. SP(z', y').

Suppose a reaches a reflex vertex and $b = a$ (step 5). The fact that $b = a$ implies that a has not yet reached a reflex vertex under this value of x, or a has traversed past the hit point z' of the most recently constructed component. First we test in O(1) time by an order query whether $a' \in P_{\mathrm{CCW}}(x, a)$, and if so we must "restart" by setting $x \leftarrow a$; if this is the third restart we halt and declare that there are no LR-visible pairs, otherwise we proceed with this new value of x (we go to step 3). If $a' \in P_{\mathrm{CCW}}(a, x)$ we continue as follows. We find a' (in steps 6 and 7) by traversing counterclockwise with b until b encounters a', and we add the component $P_{\mathrm{CW}}(a, a')$ to S_{x_0}. The value closest(a), initialized to ∞ in step 6, is the distance from a of the closest point $b \in \vec{r}_{\mathrm{CW}}(a)$ considered so far as a candidate for a' in step 7; we have dist(b, a) = closest(a) if $b = a'$, and the method which tests whether $b = a'$ (Lemma 9 below) requires closest(a). We set $y, z \leftarrow a$ and $y', z' \leftarrow a'$ (step 8), and preprocess the subpolygon

$$P_y = P_{\mathrm{CCW}}(y, y') \cup yy'$$

for order queries from y' (step 9).

Suppose a reaches a reflex vertex and $b \neq a$ (step 13). The fact that $b \neq a$ implies that a has not yet reached the hit point z' of the most recently constructed component, and that $b = z'$. As shown in Fig. 6, the following invariant holds: a lies on $P_{\mathrm{CCW}}(y, y')$ where P_y has been preprocessed for order queries from y', the component $P_{\mathrm{CW}}(z, z')$ is the one most recently added to S_{x_0}, the counterclockwise order x, y, z, a, y', z' holds, and SP(z', y') has been constructed (we may or may not have $y = z$). We first test whether $a' \in P_{\mathrm{CCW}}(x, a)$, and if so we "restart" as described above. If $a' \in P_{\mathrm{CCW}}(a, x)$ we test whether $a' \in P_{\mathrm{CCW}}(a, y')$ (step 14; we use an order query within P_y), and if the answer is no we test whether $a' \in P_{\mathrm{CCW}}(y', z')$ (steps 15–17). Efficiently testing whether $a' \in P_{\mathrm{CCW}}(y', z')$ requires use of SP(z', y') and several tangent points τ and $\tau(a)$; the use of these points will be explained in the next subsection. If the answers to any of the tests in steps 14–17 turn out to be yes, i.e., $a' \in P_{\mathrm{CCW}}(a, y')$ or $a' \in P_{\mathrm{CCW}}(y', z')$ then $P_{\mathrm{CW}}(a, a')$ is redundant with respect to $P_{\mathrm{CW}}(z, z')$, so we do not compute a' (we go to step 10). If we determine that $a' \in P_{\mathrm{CCW}}(z', x)$, then we compute a' by traversing counterclockwise with b from z' until reaching a', and we add $P_{\mathrm{CW}}(a, a')$ to S_{x_0} (steps 18–20; again, closest(a) assists in the test of determining whether $b = a'$). We set $z \leftarrow a$ and

$z' \leftarrow a'$ and update $\mathrm{SP}(z', y')$ (step 20); the manner in which $\mathrm{SP}(z', y')$ is updated is described in the next subsection.

Suppose a reaches y'. If $y = z$ (step 11) then $y' = z'$, so a is about to traverse past z'; thus a continues its traversal with $b = a$ (i.e., go to step (4)). If $y \neq z$ (step 12) then we set $y \leftarrow z$ and $y' \leftarrow z'$ and preprocess P_y for order queries from y' (the shortest path $\mathrm{SP}(z', y')$ becomes the single point $z' = y'$).

4.3. The tests

Several subprocedures are referred to in the pseudocode. These are testing whether $a' \in P_{\mathrm{CCW}}(y', z')$ (step 16), and being able to determine whether a point $b = a'$. We will soon describe how to perform these tests efficiently. We will also show that early termination occurs only if there are no LR-visible pairs. First we show that being able to perform the above tests yields a correct algorithm.

If the procedure does not terminate early, then the pointer a makes one complete traversal of P, so every reflex vertex is considered. If $a' \in P_{\mathrm{CCW}}(x, a)$ then the component $P_{\mathrm{CW}}(a, a')$ is added to S_{x_0} after x is reset to a (steps 5 and 13), and in several cases b is used to find a'. The component $P_{\mathrm{CW}}(a, a')$ is not added to S_{x_0} only if $a' \in P_{\mathrm{CCW}}(a, z')$, but in this event $P_{\mathrm{CW}}(a, a')$ is redundant with respect to $P_{\mathrm{CW}}(z, z')$. Thus, at termination S_{x_0} contains all nonredundant clockwise components. If the procedure terminates early then, by Corollary 1 below, P has no LR-visible pairs.

We now discuss performance of the tests. Steps 15–17 test whether $a' \in P_{\mathrm{CCW}}(y', z')$. If $y' = z'$ then we know that $a' \in P_{\mathrm{CCW}}(z', x)$ and thus go to step 18. If $y' \neq z'$ then we must test whether $a' \in P_{\mathrm{CCW}}(y', z')$. To do this we use $\mathrm{SP}(z', y')$.

In step 16, the tangent from a to $\mathrm{SP}(z', y')$ is constructed in order to determine whether $a' \in P_{\mathrm{CCW}}(y', z')$. (In this discussion, a tangent segment from a to $\mathrm{SP}(z', y')$ is not necessarily a chord in P.) The point of tangency is denoted τ. In general, we let $\tau(x)$ represent the tangent point on $\mathrm{SP}(z', y')$ from x for a point $x \in P_{\mathrm{CCW}}(z, y')$ (thus $\tau(z) = z'$).

First let us discuss the structure of the path $\mathrm{SP}(z', y')$. The segments $\overline{yy'}$ and $\overline{zz'}$ are chords of P, and they intersect in a point w in $\mathrm{int}(P)$ (see Fig. 6). The closed chain $P' = P_{\mathrm{CCW}}(y', z') \cup \overline{z'w} \cup \overline{wy'}$ is a simple polygon and a subpolygon of P. Thus the shortest path from z' to y' within this subpolygon is the same as within P, and it consists only of left turns (by Lemma 1).

To determine whether $a' \in P_{\mathrm{CCW}}(y', z')$, we compare $d(a^-, a)$, the direction of $\vec{r}_{\mathrm{CW}}(a)$, with $d(a, \tau(a))$, the direction of the tangent segment from a to $\mathrm{SP}(z', y')$ (refer to Fig. 7). We know that $d(a^-, a) \in \mathrm{int}(\mathrm{cone}(d(a, a^+), d(a, a^-)))$. We also know that the chain $\mathrm{SP}(z', y')$ lies entirely in $\mathrm{cone}(d(a, a^+), d(a, \tau(a)))$. Furthermore, in order for a', the hit point of $\vec{r}_{\mathrm{CW}}(a)$, to lie outside of $P_{\mathrm{CCW}}(z, z')$, there must be a point of $\vec{r}_{\mathrm{CW}}(a) \cap \overline{zz'}$ which is closer to a than is a' (since the ray shot must cross the chord $\overline{zz'}$). If $d(a^-, a) \in \mathrm{int}(\mathrm{cone}(d(a, \tau(a)), d(a, a^-)))$ then $\vec{r}_{\mathrm{CW}}(a)$ does not intersect $\mathrm{SP}(z', y')$ and therefore cannot hit $P_{\mathrm{CCW}}(y', z')$. If $d(a^-, a) \in \mathrm{cone}(d(a^+, a), d(a, \tau(a)))$ then the ray shot $\vec{r}_{\mathrm{CW}}(a)$ will hit a point of $\mathrm{SP}(z', y')$ (and consequently $P_{\mathrm{CCW}}(y', z')$) before it can reach the chord $\overline{zz'}$. Therefore knowledge of $\tau(a)$ is sufficient to determine whether $a' \in P_{\mathrm{CCW}}(y', z')$.

To assist in finding $\tau(a)$, we use a pointer τ which traverses $\mathrm{SP}(z', y')$ forwards (i.e., from z' to y'), and is initialized to z'. Recall that $\tau(x)$ represents the tangent point on $\mathrm{SP}(z', y')$ from x. Given a vertex a and an initial vertex τ on $\mathrm{SP}(z', y')$, we can determine whether $\tau(a)$ equals τ, lies on $\mathrm{SP}(z', \tau) \backslash \{\tau\}$, or lies on $\mathrm{SP}(\tau, y') \backslash \{\tau\}$, by comparing the direction of $\overline{a\tau}$ with those of the edges of $\mathrm{SP}(z', y')$ adjacent to τ. If we know that $\tau(a) \in \mathrm{SP}(\tau, y')$, we can find $\tau(a)$ by traversing

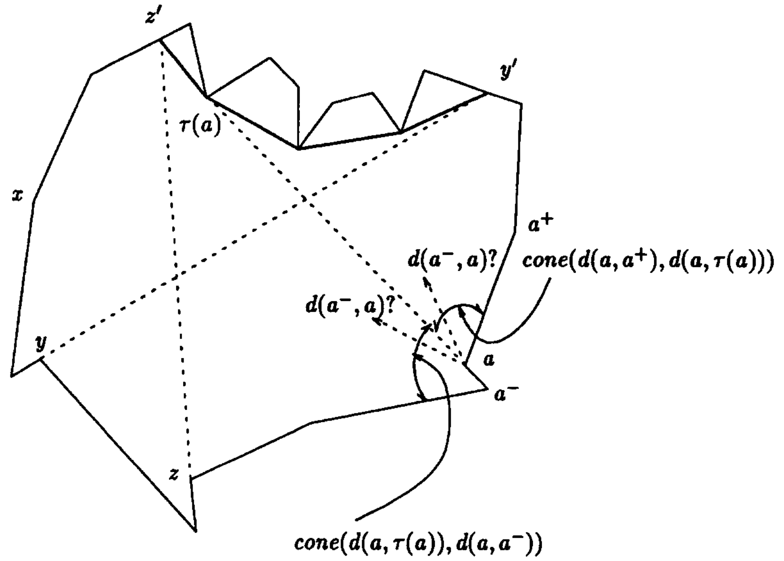

Fig. 7. Determining whether $a' \in P_{\mathrm{CCW}}(y', z')$.

$\mathrm{SP}(z', y')$ forwards with τ; the required time will be linear in the size of $\mathrm{SP}(\tau, \tau(a))$. We show below (Lemma 8) that if we enter step 16 and $\tau(a) \in \mathrm{SP}(z', \tau)\backslash\{\tau\}$ then $a' \in P_{\mathrm{CCW}}(y', z')$, implying that $P_{\mathrm{CW}}(a, a')$ is a redundant component and hence can be ignored. Thus, in the procedure, τ only traverses forwards on $\mathrm{SP}(z', y')$, never backwards. For a new value of a, we first determine whether $\tau(a)$ lies on $\mathrm{SP}(z', \tau)\backslash\{\tau\}$ (step 16); if not we search for $\tau(a)$ by continuing the forward traversal of $\mathrm{SP}(z', y')$ with τ until $\tau = \tau(a)$.

Lemma 8. *If, upon entering step* 16, *we have* $\tau(a) \in \mathrm{SP}(z', \tau)\backslash\{\tau\}$, *then* $a' \in P_{\mathrm{CCW}}(y', z')$.

Proof. Refer to Fig. 8. Upon entering step 16, the current value of τ is the point of tangency $\tau(c)$ obtained for a previous reflex vertex c; it is possible that $c = z$, and c must follow z, i.e., we have the counterclockwise order x, z, c. Earlier in the procedure, the pointer a was set to the vertex c, and the values of x and y have not changed since that time. This is true because the value of τ is reset whenever y is updated. We claim that $c' \in P_{\mathrm{CCW}}(y', z')$. If $c = z$ then this is true because $c' = z'$. If $c \neq z$ then $\tau(c)$ was computed in step 17 only because it was previously determined (in steps 13 and 14) that $c' \notin P_{\mathrm{CCW}}(x, c)$ and $c' \notin P_{\mathrm{CCW}}(c, y')$, and if c' had been on $P_{\mathrm{CCW}}(z', x)$ then z would have been updated to c. Thus $c' \in P_{\mathrm{CCW}}(y', z')$. Actually, we can see that $c' \in P_{\mathrm{CCW}}(y', \tau(c))$.

Let e be the hit point of the ray shot from $\tau(c)$ in direction $d(\tau(c), c)$ (e may or may not equal c). Because $\overline{zz'}$ is a chord, we have $e \in P_{\mathrm{CCW}}(z, z')$. We know that $e \notin P_{\mathrm{CCW}}(y', z')$ by the way e is defined. We have that $e \notin P_{\mathrm{CCW}}(c, y')\backslash\{c\}$ because $c' \in P_{\mathrm{CCW}}(y, \tau(c))$ and $\overline{cc'}$ is a chord, so no point of $P_{\mathrm{CCW}}(c, c')$ can intersect the segment $\overline{c\tau(c)}$. Thus $e \in P_{\mathrm{CCW}}(z, c)$.

The chord $\overline{e\tau(c)}$ partitions P into two subpolygons $P_1 = P_{\mathrm{CCW}}(e, \tau(c)) \cup \overline{e\tau(c)}$ and $P_2 = P_{\mathrm{CW}}(e, \tau(c)) \cup \overline{e\tau(c)}$, where $a \in P_1$ and $P_{\mathrm{CCW}}(z', x) \subset P_2$. A ray shot in P from a point w of P_1 cannot hit a point of P_2 unless w lies to the right of the directed line $\vec{l}(e, \tau(c))$. But a lies to the left of $\vec{l}(e, \tau(c))$, since $\tau(a) \in \mathrm{SP}(z', \tau)\backslash\{\tau\}$. Therefore, $a' \in P_1$.

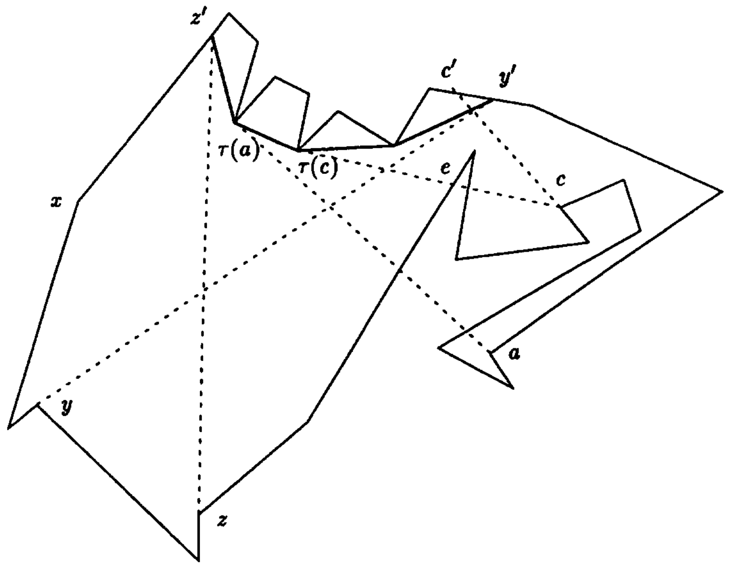

Fig. 8. Proof of Lemma 8.

Since we know that $a' \in P_{\mathrm{CCW}}(y', x)$, and $P_{\mathrm{CCW}}(y', z') \subset P_1$ while $P_{\mathrm{CCW}}(z', x) \subset P_2$, we have $a' \in P_{\mathrm{CCW}}(y', z')$. □

The construction of the shortest path $\mathrm{SP}(z', y')$ is performed in step 20. The path $\mathrm{SP}(b, y')$ is constructed, and then z' is updated to equal b. There are two cases, $y' = z'$ and $y' \neq z'$. If $y' = z'$, then $\mathrm{SP}(b, y')$ is constructed directly as follows. The segments $\overline{yy'}$ and $\overline{ab}$ are chords, which intersect at a point w. Then $P' = P_{\mathrm{CCW}}(y', b) \cup \overline{bw} \cup \overline{wy'}$ is a subpolygon of P, and a shortest path within P' is also a shortest path in P. Thus we construct $\mathrm{SP}(b, y')$ within P' in time proportional to the size of $P_{\mathrm{CCW}}(y', b)$, by [8].

If $y' \neq z'$ then it is unacceptable to construct $\mathrm{SP}(b, y')$ as above because the vertices of the chain $P_{\mathrm{CCW}}(y', z')$ were previously charged with the cost of constructing $\mathrm{SP}(z', y')$. Instead we do the following. Construct $\mathrm{SP}(b, z')$ directly within the subpolygon

$$P_1 = P_{\mathrm{CCW}}(z', b) \cup \overline{bw_1} \cup \overline{w_1 z'},$$

where $w_1 = \overline{zz'} \cap \overline{ab}$. We note that

$$P_2 = P_{\mathrm{CCW}}(y', b) \cup \overline{bw_2} \cup \overline{w_2 z'}$$

is a subpolygon of P, where $w_2 = \overline{yy'} \cap \overline{ab}$. Thus $\mathrm{SP}(b, y')$ consists of the union of two subportions of $\mathrm{SP}(b, z')$ and $\mathrm{SP}(z', y')$ with a single bridge (see Fig. 9). Merging two convex chains in this manner can be performed in time proportional to the number of vertices on either chain below the bridge [13, pp. 89–90]. The total time spent constructing shortest paths is $O(n)$, since each vertex of P is charged at most once with constructing a shortest path directly, and each vertex is below a bridge at most once during the process of merging.

Total traversal time with τ is also $O(n)$. The pointer τ always traverses forward on $\mathrm{SP}(z', y')$. When $\mathrm{SP}(z', y')$ is merged with $\mathrm{SP}(b, z')$ to form a new shortest path, the portion $\mathrm{SP}(z', \tau)$ which has been

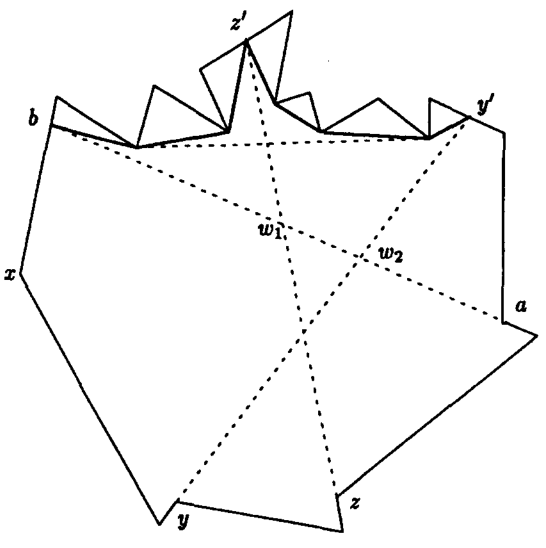

Fig. 9. Merging SP(b, z') with SP(z', y') to get SP(b, y').

already traversed will be under the bridge; this can be seen by noting that ab is a chord and $\tau = \tau(a)$ is the tangent point from a to SP(z', y') (see Fig. 9).

In steps 7 and 19 we test whether the current value of b equals a'. We use the following lemma to show that these tests can be performed.

Lemma 9. *Given points x, a_{init}, a, b_{init} and b, where (1) the points are in counterclockwise order, (2) a_{init} and b_{init} are covisible, (3) the ray shot from a in direction $d(a,b)$ is defined and does not hit $P_{\text{CCW}}(a,b)\backslash\{b\}$, (4) b is the point of $\vec{r}(a, d(a,b)) \cap P_{\text{CCW}}(b_{\text{init}}, b)$ which is closest to a, (5) P is preprocessed for order queries from x. Then we can determine in* O(1) *time whether a and b are visible.*

Proof. Let a' and b'' denote the hit points of the shots $\vec{r}(a, d(a,b))$ and $\vec{r}(b, d(b,a))$. We are trying to determine whether $b = a'$, or, equivalently, whether $a = b''$. We can determine whether $a' \in P_{\text{CCW}}(a,x)$ and whether $b'' \in P_{\text{CCW}}(x,b)$ by order queries. Unless the answer to both queries is yes, a and b cannot be visible (recall that the points obey the counterclockwise order x, a, b).

Suppose $a' \in P_{\text{CCW}}(a,x)$ and $b'' \in P_{\text{CCW}}(x,b)$. By condition (3) of the given, we have $a' \in P_{\text{CCW}}(b,x)$. Assume $a' \neq b$. Let a'' be the point of $\vec{r}(b, d(b,a)) \cap P_{\text{CCW}}(b,x)\backslash\{b\}$ closest to b. Since $b'' \in P_{\text{CCW}}(x,b)$ and b faces a, there is a point of $P_{\text{CCW}}(x,b)$ which intersects int$(\overline{ba''})$. No point of $P_{\text{CCW}}(x,a)$ can intersect int$(\overline{ba''})$ because of the chord $\overline{aa'}$, and no point of $P_{\text{CCW}}(a_{\text{init}}, b_{\text{init}})$ can intersect int$(\overline{ba''})$ because of the chord $\overline{a_{\text{init}} b_{\text{init}}}$. Also, no point of $P_{\text{CCW}}(b_{\text{init}} b)$ can intersect int$(\overline{ba''})$ because of condition (4) of the given. This is a contradiction, which implies that $a' = b$. Thus a and b are visible if and only if $a' \in P_{\text{CCW}}(a,x)$ and $b'' \in P_{\text{CCW}}(x,b)$, a condition which can be checked in O(1) time by order queries. □

Our method of finding a' requires that whenever a is a reflex vertex and $a' \in P_{\text{CCW}}(a,x)$, the points a, b, a', x are in counterclockwise order; this ensures that by traversing counterclockwise, the pointer b will encounter a' before x. This invariant holds because initially $b = a$, and every time b

encounters a point on $\vec{r}_{CW}(a)$ we test whether $b = a'$. We need only to show that the conditions of the above lemma are satisfied whenever we test $b = a'$.

We can check if b is the point of $\vec{r}_{CW}(a) \cap P_{CCW}(a,b)$ closest to a by keeping the value closest(a); if b is not closest then $b \neq a'$. To analyze the other conditions, we realize that there are two cases, corresponding to steps 7 and 19 of the pseudocode. In step 7, a is initially equal to b, so $a_{\text{init}} = a = b_{\text{init}}$, implying conditions (1) and (2). In step 19, b is initially equal to z', where $a' \in P_{CCW}(z', x)$. Thus conditions (1) and (2) are satisfied as we set $a_{\text{init}} = z$ and $b_{\text{init}} = z'$. In either step, we know inductively that $a' \notin P_{CCW}(b_{\text{init}}, b) \backslash \{b\}$. Consequently the conditions of Lemma 9 are satisfied, and in steps 7 and 19 we can test in O(1) time whether $b = a'$.

We stated earlier that if the procedure terminates because the value of x is changed for a third time then P has no LR-visible pairs. We show this now.

Lemma 10. *If the value of x is reset three times, then P has three pairwise disjoint clockwise components.*

Proof. Let x_i be the value of x after x has been updated i times. When x is updated, it is set equal to the current value of a, which is necessarily a reflex vertex. For $i \geqslant 1$, we let $a_i = x_i$. Since x is updated whenever $a' \in P_{CCW}(x, a)$, we have the following counterclockwise order of points: x_0, a_1', $a_1 = x_1$, a_2', $a_2 = x_2$, ... (see Fig. 10). Since a does not traverse past x_0, the clockwise components $P_{CW}(a_i, a_i')$ are pairwise disjoint. The lemma follows from the fact that a new component $P_{CW}(a_i, a_i')$ exists for each instance of the value of x being reset. □

Corollary 1. *If the value of x is reset three times, then P has no LR-visible pairs.*

Proof. By Lemmas 10 and 5. □

We have shown that the procedure correctly generates all nonredundant clockwise components. We now discuss the time-complexity. Since x can be reset at most twice, it suffices to show the

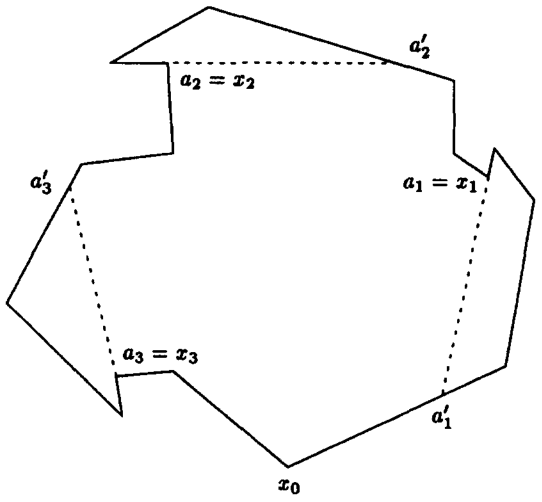

Fig. 10. Proof of Lemma 10.

work performed under a fixed value of x is $O(n)$. The pointers a and b each make a single traversal around P, and all operations are constant time. Preprocessing P for order queries from x is $O(n)$.

The total time required to preprocess P_y for order queries for y', over all P_y, is $O(n)$. For a given value of y, P_y can be preprocessed in time proportional to the number of vertices on $P_{\mathrm{CCW}}(y, y')$. A vertex v can lie on $P_{\mathrm{CCW}}(y, y')$ for two different values of y only if y is about to be updated to equal z in step 12 where $v \in P_{\mathrm{CCW}}(z, y')$. However, in this scenario there are no reflex vertices on $P_{\mathrm{CCW}}(z, y')$ which generate nonredundant clockwise components, so v lies on $P_{\mathrm{CCW}}(y, y')$ for at most two values of y. Thus the total time required to preprocess all subpolygons P_y for order queries is $O(n)$.

We have established the following.

Theorem 1. *The procedure in this section constructs a superset of the set of nonredundant clockwise components of a simple polygon P in $O(n)$ time.*

5. Computing all LR-visible pairs

In the previous section we showed an algorithm that outputs a sorted list of all nonredundant components of a simple polygon P in time linear in the number of vertices of P. In this section we show how to compute all LR-visible pairs of points in linear time given a sorted list of the nonredundant components of P.

5.1. The 2-cut problem

Given a list of nonredundant components, Tseng and Lee [15] showed how to compute all the LR-visible pairs of vertices by reducing the problem (in linear time) to the *2-cut circular arcs* problem. The 2-cut circular arcs problem is defined as follows: given a unit circle $\mathcal{K}$, and a sorted list S of circular arcs (or circular intervals), determine a *2-cuttable pair* of points (s, t), i.e., determine a pair of points (s, t) such that every circular arc of S contains either s or t. Tseng and Lee presented an $O(n \log n)$-time algorithm to solve the 2-cut circular arcs problem as well as the problem of computing all 2-cuttable pairs of points [15]. We show that given a sorted list of nonredundant components, simpler data structures can be used to implement Tseng and Lee's algorithm to run in $O(n)$ time.

Tseng and Lee's reduction to the 2-cut circular arcs problem is as follows. Given a polygon P, and the set of its nonredundant components S, perform the following mapping: map the P to a unit circle $\mathcal{K}$; map every component into an arc (circular interval) on the circle such that the relative ordering of the endpoints of the nonredundant components on P remains the same as that of the endpoints of the circular intervals on $\mathcal{K}$. As a result, the problem of finding a pair of points (s, t) such that every nonredundant component of P intersects either s or t now reduces to the problem of finding a 2-cuttable pair of points (s, t) on the unit circle $\mathcal{K}$. Similarly, the problem of finding *all* pairs of points (s, t) such that every nonredundant component of P intersects either s or t now reduces to the problem of finding *all* 2-cuttable pairs of points (s, t) on the unit circle $\mathcal{K}$.

Since every vertex of the polygon can generate at most 2 components—one clockwise and one counterclockwise—a polygon P with n vertices can only have at most $2n$ components, and, consequently, at most $4n$ (actually, $3n$) endpoints of nonredundant components. Thus the mapping mentioned above can be performed in $O(n)$ time.

We assume that each interval is described by its endpoints. Its clockwise endpoint will be called its *left* endpoint, while its counterclockwise endpoint will be referred to as its *right* endpoint. Therefore when one traverses from the left endpoint to the right endpoint in the counterclockwise direction, one is traversing along the circular interval. Lowercase letters near the beginning of the alphabet such as a_1, a'_1, b_1, b'_1, c, d, e, f, etc., will be used to denote circular intervals. Lowercase letters near the end of the alphabet such as s, t, u, v, w, x, y, z, etc., will be used to denote points on $\mathcal{K}$ or on P. Uppercase letters as in P_i, P'_i, will be used to denote sets of intervals. For a circular interval a, we use left[a] and right[a] to denote its left and right endpoints, respectively. We use the notation $\mathcal{K}(u)$ to denote the circular interval consisting of the entire circle $\mathcal{K}$ starting from u and ending at u.

5.2. Computing all 2-cuttable pairs

Tseng and Lee [15] proposed an algorithm to compute *all* 2-cuttable pairs of points and showed how to implement it in $O(n \log n)$ time. Even though there are an infinite number of these pairs of points, they show how this can be implemented by outputting a linear number of pairs of circular intervals. We present here a linear-time implementation of their algorithm. For the sake of completeness and to describe our implementation more clearly we reproduce the algorithm for computing all 2-cuttable pairs from [15]. While the algorithm reproduced here is the same as in [15], there are minor changes in the notation to maintain consistency with the rest of this paper.

Their algorithm consists of two steps. The goal of step (1) is determine if there exists a 2-cuttable pair by computing two minimal intervals b_1 and b_2 on $\mathcal{K}$ such that for any 2-cuttable pair, one cut must be in b_1 while the other must be in b_2. The goal of step (2) is to compute all 2-cuttable pairs on the pair of intervals (b_1, b_2) by performing a simultaneous sweep of the two intervals.

Tseng and Lee describe an $O(n \log m)$-time implementation of step (1) of the algorithm and an $O(n)$-time implementation of step (2) of the algorithm. Following the algorithm we show how step (1) can be implemented in linear time.

```
**Algorithm**
PROCEDURE ALL_2-CUTTABLE_PAIRS ($(S, U)$).
**Input**: Sorted list of intervals $S$.
**Output**: $U = \{(e_i, f_i),\ 1 \leqslant i \leqslant k \mid (e_i, f_i)$ is a 2-cuttable pair of intervals, and every 2-cuttable pair
        of points is contained in at least 1 such pair, and $k = O(n)\}$.
  (1) [Find $b_1$, $b_2$ and $P_3$]
      (a) Find two disjoint intervals $a_1$ and $a_2$;
        **if** no such two intervals **then**
            Let $d$ be any interval and let $x$ = left[$d$] and $y$ = right[$d$];
            Let $z$ be the left endpoint of the first interval not containing $x$ clockwise from $y$,
                or let $z$ be $x$ if that left endpoint does not exist;
            $b_1 \leftarrow \mathcal{K}(x)$; $b_2 \leftarrow \mathcal{K}(y)$; $P_3 \leftarrow S - \{d\}$; Go to (2);
        **else**
            $b_1 \leftarrow a_1$; $b_2 \leftarrow a_2$; $P_1 \leftarrow \{a_1\}$; $P_2 \leftarrow \{a_2\}$;
            Divide the intervals in $S - \{a_1, a_2\}$ into four subsets:
```

```
            $P_0'$ : intervals that intersect neither $b_1$ nor $b_2$,
            $P_1'$ : intervals that intersect $b_1$ but not $b_2$,
            $P_2'$ : intervals that intersect $b_2$ but not $b_1$,
            $P_3'$ : intervals that intersect both $b_1$ and $b_2$;
    (b) while $P_0' \cup P_1' \cup P_2' \neq \emptyset$ do
          if $P_0' \neq \emptyset$ then Report there is no 2-cuttable pair for $S$ and Stop;
          else
            $b_1' \leftarrow b_1 \cap (\bigcap_{a \in P_1'} a)$; $b_2' \leftarrow b_2 \cap (\bigcap_{a \in P_2'} a)$;
            if either $b_1'$ or $b_2'$ is empty then Report there is no 2-cuttable pair for $S$ and Stop;
            else
              $b_1 \leftarrow b_1'$; $b_2 \leftarrow b_2'$; $P_1 \leftarrow P_1 \cup P_1'$; $P_2 \leftarrow P_2 \cup P_2'$;
              Divide the intervals in $P_3'$ into four subsets $P_0''$, $P_1''$, $P_2''$, $P_3''$ similar to above;
              $P_0' \leftarrow P_0''$; $P_1' \leftarrow P_1''$; $P_2' \leftarrow P_2''$; $P_3' \leftarrow P_3''$;
    (c) $P_3 \leftarrow P_3'$; $x \leftarrow$ left[$b_1$]; $y \leftarrow$ left[$b_2$]; $z \leftarrow y$;
(2) [Find all 2-cuttable pairs of intervals $(e_i, f_i)$ in $(b_1, b_2)$]
    $i \leftarrow 1$; $u \leftarrow x$;
    Scan $b_1$ counterclockwise from $x$ to right[$b_1$]:
    while not reaching right[$b_1$] do
    (a) if right[$a$] is reached for some interval a then
          Output $(e_i, f_i) = ((u, \text{right}[a]), (z, z'))$, where $z'$ is the right endpoint of the first interval
                not containing $u$ scanned counterclockwise from $z$, or $z'$ is $z$ if that right endpoint
                does not exist;
          if $a$ does not contain $z$ then
            $z \leftarrow$ left[$a$];
            if $z \notin f_i$ then $(u, z) \leftarrow$ NEXT$(u, z)$; else $u \leftarrow$ right[$a$];
          else $u \leftarrow$ right[$a$];
          $i \leftarrow i + 1$;
    (b) else if left[$a$] is reached for some interval $a$ then
            Output $(e_i, f_i) = ((u, \text{left}[a]), (z, \text{right}[a]))$;
            $u \leftarrow$ left[$a$]; $i \leftarrow i + 1$;
    Output $(e_i, f_i) = ((u, \text{right}[b_1]), (g, \text{right}[b_2]))$;
```

Implementation

Tseng and Lee [15] mention how step (1a) can be implemented in linear time. However, to help implement step (1b) in linear time we exploit the crucial point that the input to this algorithm is a sorted list of proper circular intervals arising out of a list of sorted nonredundant components output by the algorithm of Section 4. Consequently, in step (1a), when we divide the intervals in $S - \{a_1, a_2\}$ into the four sets of intervals P_0', P_1', P_2' and P_3', the first three sets, namely, P_0', P_1' and P_2' (the last set P_3' is implemented differently as described below) are represented as a doubly-linked list of intervals sorted according to their (right and left) endpoints with pointers maintained to the first and last item in each list. Note that since the intervals in P_3' intersect both b_1 and b_2, they must be intervals that have one endpoint in b_1 and the other endpoint in b_2. The set P_3' is represented as two doubly-linked lists of sorted intervals (call them P_{31}' and P_{32}'). The list P_{31}' (P_{32}') consists of intervals in P_3' with their left endpoints in b_1 (b_2) with the first item being the interval with its left endpoint immediately following (in the clockwise sense) the left[b_1] (right[b_2]).

Now consider one iteration from step (1b). At any instant, let the leftmost and rightmost intervals in P_1' be a_1' and a_1''. Since these are the first and last items in the linked list P_1', the step $b_1' \leftarrow b_1 \cap (\bigcap_{a \in P_1'} a)$ can be performed in O(1) time by simply taking the intersection of the three intervals a_1', a_1'' and b_1. Similarly the step $b_2' \leftarrow b_2 \cap (\bigcap_{a \in P_2'} a)$ can also be performed in O(1) time.

To divide the intervals in P_3' into the four subsets in step (1b), we scan both the lists P_{31}' and P_{32}' from the start as well as from the tail end. The candidates for P_1'' can be found at the front (back) end of P_{31}' (P_{32}'), while the candidates for P_2'' can only be found at the tail (front) end of the list P_{32}' (P_{31}'). So the idea is to scan P_{31}' from the front (back) end and to move an interval to the list P_{31}' (P_{32}') if the interval satisfies the condition that it intersects b_1 (b_2) and not b_2 (b_1). If an interval that intersects both b_1 and b_2 is encountered the scan is stopped from that end. If an interval that intersects neither is encountered the entire algorithm is terminated. The remaining intervals are made part of P_3'' (which is also implemented as two doubly-linked lists) and are not scanned during that iteration. Summing over all the iterations, step (1b) takes $O(n)$ time.

We also assume that P_3 is implemented as a sorted list of endpoints of intervals. In step (1c) to implement the statement $P_3 \leftarrow P_3'$, we assume that the lists P_{31}' and P_{32}' are merged and a sorted list of endpoints of the intervals in the two lists are obtained. Implementing P_3 as a sorted list of endpoints will enable an efficient implementation of the scanning of the endpoints in b_1 in step (2). The rest of step (2) can be implemented in $O(n)$ time as described by Tseng and Lee [15]. It should also be pointed out that NEXT(x, y) (used in step (2)) is defined in [15] as the first 2-cuttable pair of points (x', y') reached by moving x and/or y counterclockwise on $\mathcal{K}$. However, the definition is used in a more limited situation. It is assumed that (w, y) is a 2-cuttable pair of points, and after a counterclockwise motion of w, the pair (x, y) is reached, which is not 2-cuttable. Now NEXT(x, y) is computed by moving y counterclockwise until a 2-cuttable pair (x, y') is reached.

5.3. Outputting all LR-visible pairs

The procedure ALL_2-CUTTABLE_PAIRS produces a set of pairs of circular intervals such that any pair of points taken one each from a pair of intervals forms a 2-cuttable pair of points. Because of the way step (2) of the procedure works, it is clear that each e_i, such that $1 \leqslant i \leqslant k$ and $(e_i, f_i) \in U$, is disjoint. Every pair of 2-cuttable points is contained in at least one pair of intervals (e_i, f_i). What remains is to map the circular intervals back to polygon chains of the polygon P. Since the algorithm produces $O(n)$ pairs of intervals, this reverse mapping can also be performed in $O(n)$ time. What results is a set of pairs of polygonal chains $U' = (S_i, T_i)$, $1 \leqslant i \leqslant k$, such that the polygon P is LR-visible with respect to any pair of points taken one each from a pair of polygonal chains from U'. Based on our earlier discussions, the resulting polygonal pairs of polygonal chains contain all pairs of points with respect to which the polygon P is LR-visible.

Thus, putting all the pieces together gives us the following theorem.

Theorem 2. *There exists a linear-time algorithm to output a set of pairs of polygonal chains containing all pairs of LR-visible points for a given polygon P.*

6. Conclusions

In this paper, we have presented an optimal linear-time algorithm for solving the most general version of the LR-visibility problem for simple polygons, namely, the problem of finding all pairs of points with respect to which a simple polygon is LR-visible. The basic technique used is that of sweeping the polygon a constant number of times to find nonredundant components, after which the LR-visible pairs are produced by sweeping the nonredundant components.

What is interesting is that this technique was also used by the authors to solve the most general version of the weakly visible chords problem, namely, the problem of finding all weakly visible chords of a simple polygon [6].

The LR-visibility problem is also closely related to the 2-guards problem and its variants. It would be interesting to see whether these techniques are useful in solving any of those problems to optimality.

References

[1] 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.
[2] B. Chazelle, Triangulating a simple polygon in linear time, Discrete Comput. Geom. 6 (1991) 485–524.
[3] D.Z. Chen, An optimal parallel algorithm for detecting weak visibility of a simple polygon, in: Proc. 8th ACM Symp. Comput. Geom. (1992) 63–72; to appear in Internat. J. Comput. Geom. Appl.
[4] D.Z. Chen, Optimally computing the shortest weakly visible subedge of a simple polygon, Tech. Report No. 92–028, Dept. of Computer Sciences, Purdue University, May 1992.
[5] B. Chazelle and L. Guibas, Visibility and intersection problems in plane geometry, Discrete Comput. Geom. 4 (1989) 551–581.
[6] G. Das, P.J. Heffernan and G. Narasimhan, Finding all weakly-visible chords of a polygon in linear time, Manuscript (1993).
[7] J. Doh and K. Chwa, An algorithm for determining internal line visibility of a simple polygon, J. Algorithms 14 (1993) 139–168.
[8] L. Guibas, J. Hershberger, D. Leven, M. Sharir and R. Tarjan, Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons, Algorithmica 2 (1987) 209–233.
[9] P.J. Heffernan, An optimal algorithm for the two-guard problem, in: Proc. 9th ACM Symp. Comput. Geom. (1993) 348–358; to appear in Internat. J. Comput. Geom. Appl.
[10] C. Icking and R. Klein, The two guards problem, Internat. J. Comput. Geom. Appl. 2 (1992) 257–285.
[11] Y. Ke, Detecting the weak visibility of a simple polygon and related problems, Tech. Report, The Johns Hopkins University (1987).
[12] D.T. Lee and F.P. Preparata, An optimal algorithm for finding the kernel of a polygon, J. Assoc. Comput. Mach. 26 (1979) 415–421.
[13] F.P. Preparata and S.J. Hong, Convex hulls of finite sets of points in two and three dimensions, CACM 20 (1977) 87–93.
[14] J.-R. Sack and S. Suri, An optimal algorithm for detecting weak visibility, IEEE Trans. Comput. 39 (1990) 1213–1219.
[15] L.H. Tseng and D.T. Lee, Two-guard walkability of simple polygons, Manuscript (1993).