SIAM J. COMPUT.
Vol. 17, No. 1, February 1988

010

AN $O(n \log\log n)$-TIME ALGORITHM FOR TRIANGULATING A SIMPLE POLYGON*

ROBERT E. TARJAN†‡ AND CHRISTOPHER J. VAN WYK†

Abstract. Given a simple n-vertex polygon, the *triangulation problem* is to partition the interior of the polygon into $n-2$ triangles by adding $n-3$ nonintersecting diagonals. We propose an $O(n \log\log n)$-time algorithm for this problem, improving on the previously best bound of $O(n \log n)$ and showing that triangulation is not as hard as sorting. Improved algorithms for several other computational geometry problems, including testing whether a polygon is simple, follow from our result.

Key words. amortized time, balanced divide and conquer, heterogeneous finger search tree, homogeneous finger search tree, horizontal visibility information, Jordan sorting with error-correction, simplicity testing

AMS(MOS) subject classifications. 51M15, 68P05, 68Q25

1. Introduction.

Let P be an n-vertex simple polygon, defined by a list $v_0, v_1, \ldots, v_{n-1}$ of its vertices in clockwise order around the boundary. (The interior of the polygon is to the right as one walks clockwise around the boundary.) We denote the boundary of P by ∂P. We assume throughout this paper (without loss of generality) that the vertices of P have distinct y-coordinates. For convenience we define $v_n = v_0$. The *edges* of P are the open line segments whose endpoints are v_i, v_{i+1} for $0 \leq i < n$. The *diagonals* of P are the open line segments whose endpoints are vertices and that lie entirely in the interior of P. The *triangulation problem* is to find $n-3$ nonintersecting diagonals of P, which partition the interior of P into $n-2$ triangles.

If P is convex, any pair of vertices defines a diagonal, and it is easy to triangulate P in $O(n)$ time. If P is not convex, not all pairs of vertices define diagonals, and even finding one diagonal, let alone triangulating P, is not a trivial problem. In 1978, Garey, Johnson, Preparata and Tarjan [10] presented an $O(n \log n)$-time triangulation algorithm. Since then, work on the problem has proceeded in two directions. Some authors have developed linear-time algorithms for triangulating special classes of polygons, such as monotone polygons [10] and star-shaped polygons [31]. Others have devised triangulation algorithms whose running time is $O(n \log k)$ for a parameter k that somehow quantifies the complexity of the polygon, such as the number of reflex angles [13] or the "sinuosity" [5]. Since these measures all admit classes of polygons with $k = \Omega(n)$, the worst case running time of these algorithms is only known to be $O(n \log n)$. Determining whether triangulation can be done in $o(n \log n)$ time, i.e. asymptotically faster than sorting, has been one of the foremost open problems in computational geometry.

In this paper we propose an $O(n \log\log n)$-time triangulation algorithm, thereby showing that triangulation is indeed easier than sorting. The paper is a revised and corrected version of a conference paper [27] which erroneously claimed an $O(n)$-time algorithm. The goal of obtaining a linear-time algorithm remains elusive, but our

*Received by the editors September 8, 1986; accepted for publication (in revised form) April 29, 1987. Typeset on July 28, 1987 at AT&T Bell Laboratories, Murray Hill, New Jersey.

†AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

‡Department of Computer Science, Princeton University, Princeton, New Jersey 08544. The work of this author was partially supported by National Science Foundation grant DCR-8605962.

approach suggests some directions in which to look and clarifies the difficulties that must be overcome.

The starting point for our algorithm is a reduction of the triangulation problem to the problem of computing visibility information along a single direction, which we take to be horizontal. A *vertex-edge visible pair* is a vertex and an edge that can be connected by an open horizontal line segment that lies entirely inside P. Similarly, an *edge-edge visible pair* is a pair of edges that can be connected by an open horizontal line segment that lies entirely inside P. Fournier and Montuno [9] showed that triangulating P is linear-time equivalent to computing all vertex-edge visible pairs. The reduction of triangulation to computing visible pairs was independently obtained by Chazelle and Incerpi [5]. What we shall actually produce is an $O(n \log\log n)$-time algorithm for computing visible pairs, which by this reduction leads to an $O(n \log\log n)$-time triangulation algorithm.

Our visibility algorithm computes not only vertex-edge visible pairs but also possibly some edge-edge visible pairs. It is reassuring that the total number of visible pairs of either kind is linear.

LEMMA 1. *There are at most* $2n$ *vertex-edge visible pairs and at most* $2n$ *edge-edge visible pairs.*

Proof. Each vertex can be in at most two vertex-edge visible pairs, for a total over all vertices of at most $2n$. Partition P into trapezoids and triangles by drawing a horizontal line segment between each visible pair (of either kind) through the interior of P. (See Figure 1.) Each edge-edge visible pair corresponds to the bottom boundary of exactly one such trapezoid or triangle, the top boundary of which is either one or two line segments corresponding to vertex-edge visible pairs (in the case of a trapezoid) or a vertex that is in no visible pairs (in the case of a triangle). A vertex

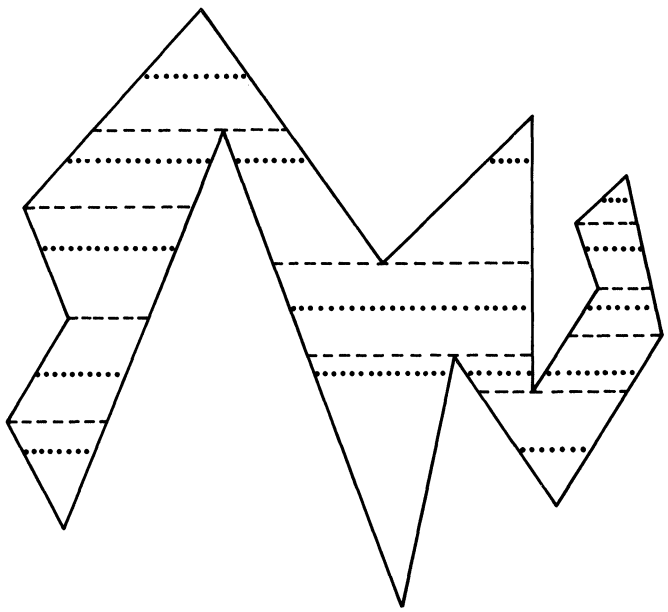

FIG. 1. *A simple polygon P, showing visibility information: dashed horizontal lines correspond to vertex-edge visible pairs; dotted horizontal lines correspond to edge-edge visible pairs.*

gives rise to at most two top boundary segments of trapezoids or to at most one top boundary of a triangle. Thus there are at most $2n$ trapezoids and triangles whose bottom boundaries correspond to edge-edge visible pairs, and hence at most $2n$ such pairs. □

The second cornerstone of our method is the intimate connection between visibility computation and the Jordan sorting problem. For a simple polygon P and a horizontal line L, the *Jordan sorting problem* is to sort the intersection points of ∂P and L by x-coordinate, given as input only a list of the intersections in the order in which they occur clockwise around ∂P. (The list of vertices of P is *not* part of the input.) Hoffman, Mehlhorn, Rosenstiehl and Tarjan [14] have presented a linear-time Jordan sorting algorithm, which actually works for any simple curve, open or closed. This algorithm, which we call "the Jordan sorting algorithm," requires that ∂P actually cross L wherever it touches it, but the algorithm is easily modified to handle tangent points, provided that each intersection point is labeled in the input as being either crossing or tangent.

Computing visible pairs is at least as hard as Jordan sorting, in the sense made precise in the following lemma:

LEMMA 2. *Using an algorithm to compute vertex-edge visible pairs, one can solve the Jordan sorting problem for an n-vertex polygon P in $O(n)$ additional time, given as input the polygon and the line L (and not the intersections).*

Proof. Compute all vertex-edge visible pairs for P. Next, turn P "inside out" by breaking P at its lowest vertex and drawing a box around it as shown in Figure 2, forming a polygon Q with $n+5$ vertices. Compute the vertex-edge visible pairs for Q. These pairs specify vertex-edge visibilities on the outside of P, and indicate which vertices of P can see arbitrarily far left or right on the outside. Partition the inside and outside of P into trapezoids, triangles, and unbounded trapezoidal regions by drawing horizontal line segments corresponding to each visible pair. Given a line L, the intersection points of ∂P and L can be read off in increasing x-order by moving left to right through the regions that intersect L. The total time for this algorithm, not including the two visibility computations, is $O(n)$. □

Since any visibility computation does Jordan sorting implicitly, it is natural to try using Jordan sorting explicitly to compute visible pairs. This leads to the following divide-and-conquer visibility algorithm (see Figure 3):

Step 1. Given P, choose a vertex v of P that does not have maximum or minimum y-coordinate. If no such v exists, stop: there are no visible pairs to compute. Otherwise, let L be the horizontal line through v.

Step 2. Determine the intersection points of ∂P and L in the order in which they occur along the boundary of P.

Step 3. Jordan sort the intersection points and report the visible pairs that correspond to consecutive intersection points along L.

Step 4. Slice P along L, dividing P into a collection of subpolygons.

Step 5. Apply the algorithm recursively to each subpolygon computed in Step 4.

The Jordan sorting algorithm has the fortuitous side effect of computing enough extra information so that Step 4 is easy. The hard part of the computation is Step 2. There are two major bottlenecks in the algorithm, either of which will make a naive implementation run in quadratic time. First, it is possible for the algorithm to report redundant visibility pairs; indeed, the example in Figure 4 shows that it can report $\Omega(n^2)$ nondistinct pairs.

We eliminate this bottleneck by modifying Step 2 to compute only some of the intersections of ∂P with L. This can cause the Jordan sorting algorithm used in Step

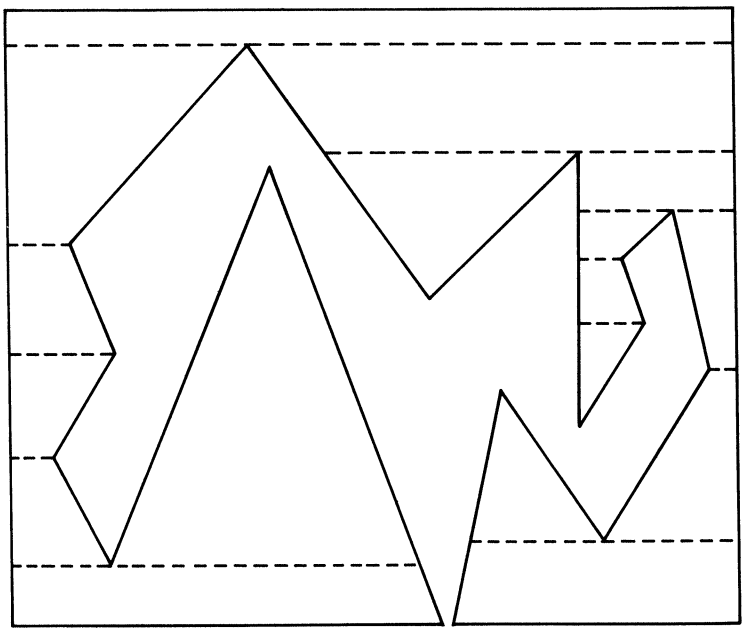

FIG. 2. *Polygon P of Figure* 1 *turned "inside out," and showing exterior vertex-edge visible pairs.*

3 to detect an error, since the sequence to be sorted need no longer consist of all intersections of a simple polygon with a line. Fortunately the sorting algorithm is incremental, and when it detects an error, we can restart it in a correct state by computing a few additional intersections and making local changes in its data structure. We call this augmented sorting method *Jordan sorting with error-correction.*

By computing only some of the intersections of ∂P and L and using Jordan sorting with error-correction, we obtain a visibility algorithm that reports only $O(n)$ visible pairs and runs in $O(n)$ time not counting the time needed to find intersections. This approach requires the use of a two-level data structure to represent polygon boundaries, but imposes no further constraints on the details of the data structure.

The second, far more serious bottleneck is the problem of actually finding the intersections. The line L divides ∂P into pieces. If each of these pieces ended up in a different subpolygon boundary, then we could obtain (with a little work) an overall $O(n)$ time bound for the visibility algorithm by using finger search trees in the boundary data structure and appealing to the linearity of the following recurrence [20, p. 185]:

$$(1) \qquad T(n) = \begin{cases} O(1) & \text{if } n = 1; \\ \max\limits_{1 \leq k < n} \{T(k) + T(n-k) + O(1 + \log \min\{k, n-k\})\} & \text{if } n > 1 . \end{cases}$$

Unfortunately the pieces of the original boundary do not stay apart but are regrouped to form the subpolygon boundaries. To beat the $O(n \log n)$ time bound of previous triangulation algorithms, we need another idea, that of *balanced divide and conquer.* We refine the visibility algorithm to choose L judiciously, so that each of the subpolygon boundaries contains a relatively small number of pieces of the original

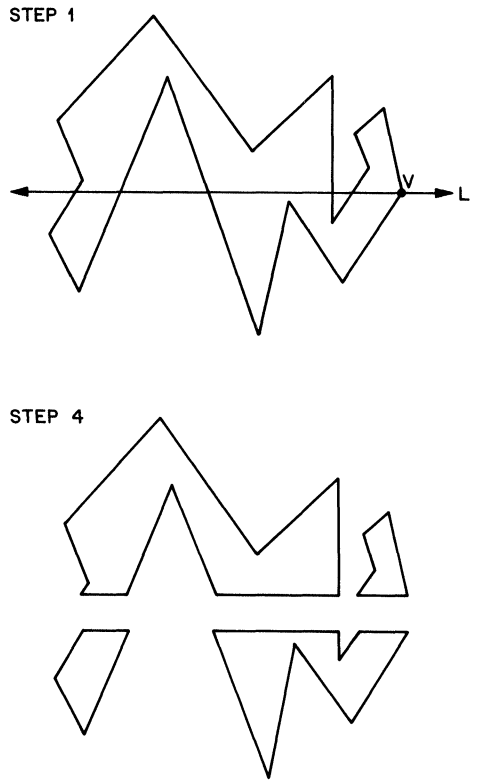

FIG. 3. *Illustrating Steps* 1 *and* 4 *of the visibility algorithm.*

boundary. Balanced divide and conquer combined with the use of finger search trees in the boundary data structure produces a visibility algorithm that runs in $O(n \log\log n)$ time.

The remainder of this paper consists of five sections and an appendix. In Section 2 we review the Jordan sorting algorithm and modify it to do error-correction. In Section 3 we present a generic visibility algorithm, based on Jordan sorting with error-correction, that reports $O(n)$ visible pairs. In Section 4 we refine the algorithm so that it uses balanced divide and conquer. In Section 5 we propose a data structure for representing the polygon boundary that consists of two levels of finger search trees. We show that with this data structure the visibility algorithm of Section 4 runs in $O(n \log\log n)$ time. We close in Section 6 with some remarks, applications, and open problems. The appendix contains a discussion of finger search trees, which are needed not only in the visibility algorithm itself but also in the Jordan sorting algorithm.

2. Jordan sorting with error-correction. Let P be a simple polygon and let v be a vertex of P. Let L be the horizontal line through v, and let $x_0 = v, x_1, \ldots, x_{m-1}$ be the intersection points of ∂P and L in clockwise order around ∂P. (Since throughout this paper we are assuming that the vertices of P have distinct y-coordinates, ∂P and

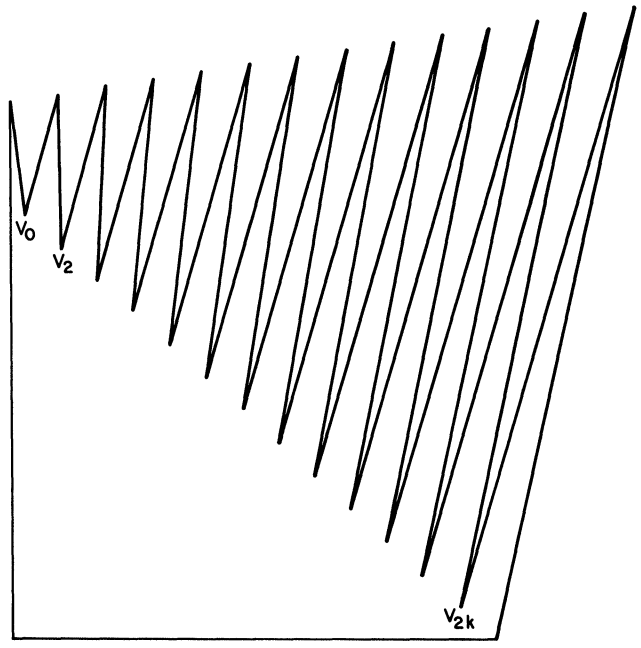

FIG. 4. *This class of polygons can cause the naive algorithm to produce a quadratic amount of output. A first slice through v_0 cuts off $k+1$ triangles. Successive slices at v_{2i}, $i=1, 2, \ldots, k$, report $k-i+1$ visible pairs, but only two are new each time.*

L intersect in a finite set of points.) Points $x_1, x_2, \ldots, x_{m-1}$ are crossing points of ∂P and L; point x_0 is either a crossing point or a tangent point. We impose a total order on the points x_i given by the order of their x-coordinates. We wish to sort $x_0, x_1, \ldots, x_{m-1}$ according to this total order.

The sequence $x_0, x_1, \ldots, x_{m-1}$ gives rise to two forests, in the following way. For convenience let $x_m = x_0$. Without loss of generality assume that the part of ∂P from x_0 to x_1 lies above L. For $0 < i \leqslant m$, let $\ell_i = \min\{x_{i-1}, x_i\}$ and $r_i = \max\{x_{i-1}, x_i\}$. We say a pair $\{x_{i-1}, x_i\}$ *encloses* a point x if $\ell_i \leqslant x < r_i$. We say two pairs $\{x_{i-1}, x_i\}$ and $\{x_{j-1}, x_j\}$ *cross* if $\{x_{i-1}, x_i\}$ encloses exactly one of x_{j-1} and x_j; $\{x_{i-1}, x_i\}$ *encloses* $\{x_{j-1}, x_j\}$ if it encloses both of x_{j-1} and x_j. The simplicity of P implies that if $i \equiv j \bmod 2$, then the two pairs $\{x_{i-1}, x_i\}$ and $\{x_{j-1}, x_j\}$ do not cross. We call this the *noncrossing property*. The Hasse diagram of the "encloses" relation on the set of pairs $\{\{x_{2i}, x_{2i+1}\} \mid 0 \leqslant i < m/2\}$ is a forest, called the *upper forest*. The Hasse diagram of "encloses" on the set of pairs $\{\{x_{2i-1}, x_{2i}\} \mid 0 < i \leqslant m/2\}$ is also a forest, called the *lower forest*. We order each set of siblings in either forest by placing $\{x_{i-1}, x_i\}$ before $\{x_{j-1}, x_j\}$ if $r_i \leqslant \ell_j$. This makes each forest into an ordered forest. We make the two forests into trees by adding the dummy pair $\{-\infty, \infty\}$ to each. Thus we obtain two trees, called the *upper tree* and the *lower tree*. (See Figure 5.) We call the set consisting of a parent in either tree and its children a *family*.

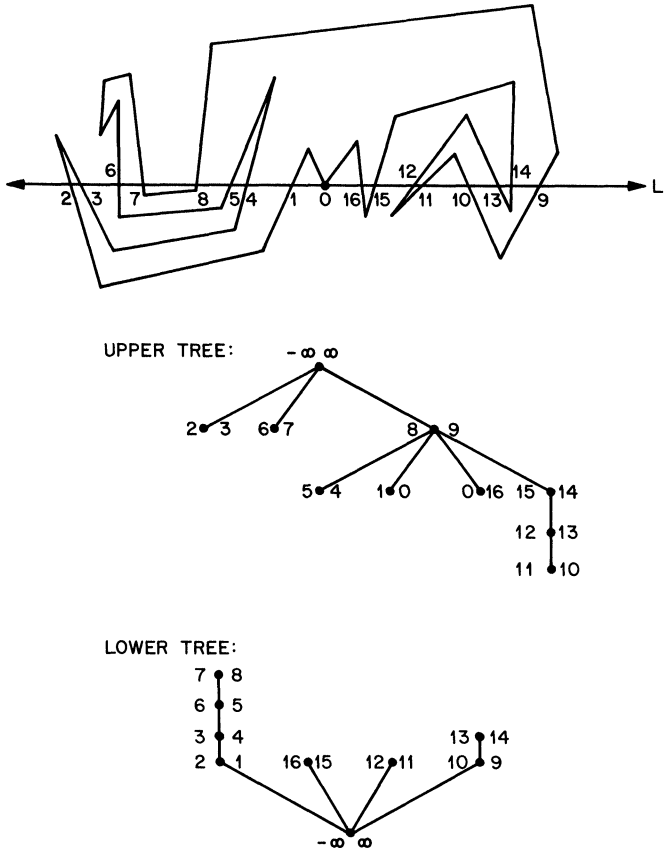

FIG. 5. *Hasse diagram of the "encloses" relation with respect to line L. (Point x_i is labelled i.)*

We shall restate the Jordan sorting algorithm [14] in a form suitable for extension to the visibility computation. The algorithm proceeds incrementally, processing the points $x_1, x_2, \ldots, x_m$ one at a time and building the upper tree, the lower tree, and a list of the points in sorted order. Initialization consists of making $\{-\infty, \infty\}$ the only pair in both trees and defining the sorted list to be $(-\infty, x_0, \infty)$. The general step consists of processing point x_i by performing the following steps. Suppose i is odd, i.e. $\{x_{i-1}, x_i\}$ is to be added to the upper tree. Assume $x_{i-1} < x_i$. (The case $x_{i-1} > x_i$ is symmetric.)

Step 1. Find the point x that follows x_{i-1} in the sorted list.

Step 2. Find the pair $\{x_{j-1}, x_j\}$ in the upper tree such that $x \in \{x_{j-1}, x_j\}$.

Step 3. Apply the appropriate one of the following four cases (see Figure 6):

Case A $(\ell_j < x_{i-1} < r_j < x_i)$. Halt: $\{x_{i-1}, x_i\}$ and $\{x_{j-1}, x_j\}$ cross.

Case B $(\ell_j < x_{i-1} < x_i \leqslant r_j)$. Make $\{x_{i-1}, x_i\}$ the new last child of $\{x_{j-1}, x_j\}$. If $i < m$, insert x_i after x_{i-1} in the sorted list.

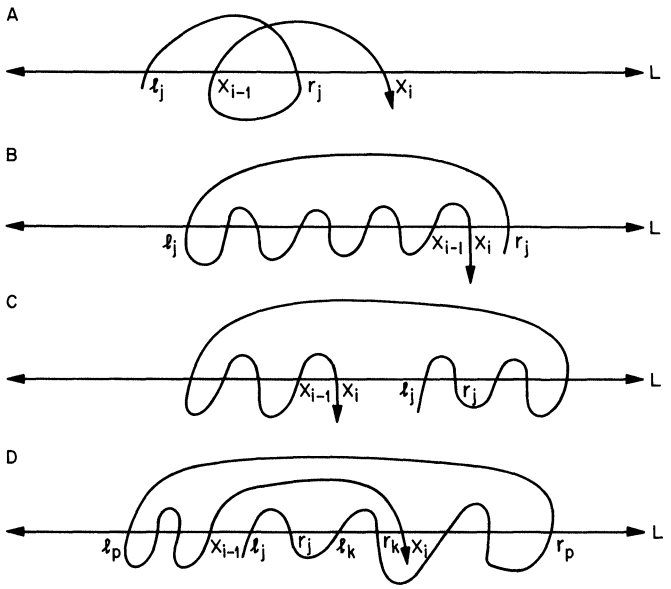

FIG. 6. *The four cases for Jordan sorting.*

Case C ($x_i \leqslant \ell_j$). Insert $\{x_{i-1},x_i\}$ into the list of siblings of $\{x_{j-1},x_j\}$ just before $\{x_{j-1},x_j\}$. If $i < m$, insert x_i after x_{i-1} in the sorted list.

Case D ($x_{i-1} < \ell_j < x_i$). In the list of siblings of $\{x_{j-1},x_j\}$, find the last one, say $\{x_{k-1},x_k\}$, such that $\ell_k < x_i$. If $r_k > x_i$, halt: $\{x_{i-1},x_i\}$ and $\{x_{k-1},x_k\}$ cross. Otherwise, if $\{x_{k-1},x_k\}$ is the last child of its parent pair $\{x_{p-1},x_p\}$ and $r_p < x_i$, halt: $\{x_{i-1},x_i\}$ and $\{x_{p-1},x_p\}$ cross. If neither of these crossings is found, remove from the list of siblings of $\{x_{j-1},x_j\}$ the sublist from $\{x_{j-1},x_j\}$ to $\{x_{k-1},x_k\}$ (inclusive) and replace it by $\{x_{i-1},x_i\}$. Make the removed sublist the list of children of $\{x_{i-1},x_i\}$. If $i < m$, insert x_i after r_k in the sorted list.

Observe that if $\{x_{i-1},x_i\}$ and $\{x_{j-1},x_j\}$ are two pairs with $i > j$ and $i \equiv j \bmod 2$, all four points x_{i-1},x_i, x_{j-1},x_j are distinct unless $i = m$ and m is odd, in which case possibly $x_i \in \{x_{j-1},x_j\}$. This means that Cases A-D exhaust the possible ordering relationships among the four points.

If i is even, i.e. $\{x_{i-1},x_i\}$ is to be added to the lower tree, the processing is analogous to the above, with a few changes needed to accommodate the fact that x_0 is in no pair in the lower tree until $x_m = x_0$ is processed (if then). The changes are as follows:

(i) Just before Step 2, if $x = x_0$, replace x by the point that follows x_0 in the sorted list.

(ii) In Step 2, find the pair $\{x_{j-1},x_j\}$ that contains x in the lower tree (instead of in the upper tree).

(iii) In Step 3, Cases B and C, if $x_{i-1} < x_0 < x_i$, insert x_i in the sorted list after x_0 (instead of after x_{i-1}).

(iv) In Step 3, Case D, if $r_k < x_0 < x_i$, insert x_i in the sorted list after x_0 (instead of after r_k).

The sorting algorithm as stated tests for crossing pairs. If the input is guaranteed to be correct (i.e. to have the noncrossing property), we can simplify the algorithm by eliminating Case A and the two tests for crossing pairs in Case D.

Making the algorithm run in linear time requires the use of appropriate data structures. Each list of siblings in the upper and lower trees is represented by a homogeneous finger search tree (see the appendix) in which each leaf is a pair in the list. In addition, there are bidirectional pointers between each pair and its first and last children (in whichever tree contains the pair). Thus each family forms a doubly-linked circular list, with the additional property that any pair in the list can be accessed from any other pair d away in either direction in $O(1+\log d)$ time. Furthermore the amortized time[1] to insert a pair next to a given one in a family list is $O(1)$, and the amortized time to remove a sublist of d pairs from a list of s pairs, given the end pairs of the sublist, is $O(1+\log(\min\{d, s-d\}+1))$.

The running time of the Jordan sorting algorithm is dominated by the time spent doing "remove a sublist/insert a pair" operations on family lists. Let $T(p,s)$ be the maximum amortized time needed to do a total of p such operations on an initial list of size s and on the removed sublists. Then $T(p,s)$ obeys the following recurrence:

$$T(p,s) = \begin{cases} 0 \quad \text{if } p=0; \\ \max\limits_{\substack{0\leqslant i<p \\ 0\leqslant d\leqslant s}} \{T(i,d+1) + T(p-i-1, s-d+1) \\ \qquad\qquad + O(1+\log(\min\{d, s-d\}+1))\} \quad \text{if } p>0. \end{cases} \tag{2}$$

A proof by induction shows that $T(p,s) = O(p+s)$. The list manipulation time of the Jordan sorting algorithm is at most $T(\lceil m/2\rceil, 1) + T(\lfloor m/2\rfloor, 1)$, from which it follows that the algorithm runs in $O(m)$ time. For further details of the algorithm and the analysis see the original paper [14]. (In our restatement of the algorithm, we have modified the data structure slightly, the main change being to eliminate circular level links in the finger search trees. These changes do not affect the $O(m)$ time bound.)

We now want to augment the Jordan sorting algorithm so that when it detects two crossing pairs, it can in certain cases restart itself in a corrected state that represents the partial sorting of a sequence modified to eliminate the crossing. In the application of Jordan sorting to the visibility computation, the sorting algorithm receives as input only a possibly noncontiguous subsequence of the sequence of intersections. In this subsequence, certain pairs are designated as *special*. (All other pairs are *normal*.)

To accommodate the operation of the triangulation algorithm, we impose on each special pair $\{x_{i-1},x_i\}$ the additional requirement that it enclose no given intersections other than x_{i-1} and x_i. We call this the *nonenclosure property*. On the other hand, special pairs are potentially modifiable: if $\{x_{i-1},x_i\}$ is a special pair, there may be additional intersections between x_{i-1} and x_i along ∂P. The Jordan sorting algorithm is allowed to request such additional intersections if it detects a crossing or a violation of the nonenclosure property.

The mechanism for providing additional intersections is a procedure named *refine*, whose input parameters consist of a special pair $\{x_{i-1},x_i\}$ and a point x enclosed by

[1] *Amortized time* is the time per operation averaged over a worst-case sequence of operations that begins with an empty data structure. For a discussion of this concept see the first author's survey paper [26].

$\{x_{i-1},x_i\}$. The procedure returns a *bracketing pair* x',x'' such that x' and x'' are intersections of ∂P and L, the four intersections occur in the order x_{i-1},x',x'',x_i along ∂P, and the five points occur in the order x_{i-1},x',x,x'',x_i (or its reverse) along L. If there is no such pair, *refine* returns nothing. If a pair is returned, the new sequence to be sorted is the old sequence with x' followed by x'' inserted between x_{i-1} and x_i. Of the three pairs that replace $\{x_{i-1},x_i\}$, the pairs $\{x_{i-1},x'\}$ and $\{x'',x_i\}$ are special and the pair $\{x',x''\}$ is normal. (This means that $\{x',x''\}$ is the *closest* pair of intersections to x.)

We shall modify the Jordan sorting algorithm so that it can handle special pairs, using *refine* when possible to eliminate violations of the noncrossing and nonenclosure properties. The modification consists of the following three additions to the algorithm (see Figure 7):

(i) In Step 1, if $\{x_{i-1},x_i\}$ is special and $x < x_i$, call *refine* $(\{x_{i-1},x_i\},x)$. If *refine* returns no pair, halt: $\{x_{i-1},x_i\}$ violates the nonenclosure property. If *refine* returns a pair (x',x''), insert x' and x'' in the sequence to be sorted between x_{i-1} and x_i and restart the processing with x'.

(ii) In Step 3, Case B, if $\{x_{j-1},x_j\}$ is special, halt: the nonenclosure property has been violated. Even if it can be restored by refining $\{x_{j-1},x_j\}$, this will produce a violation of the noncrossing property. (This also happens in Step 3, Case A: even if

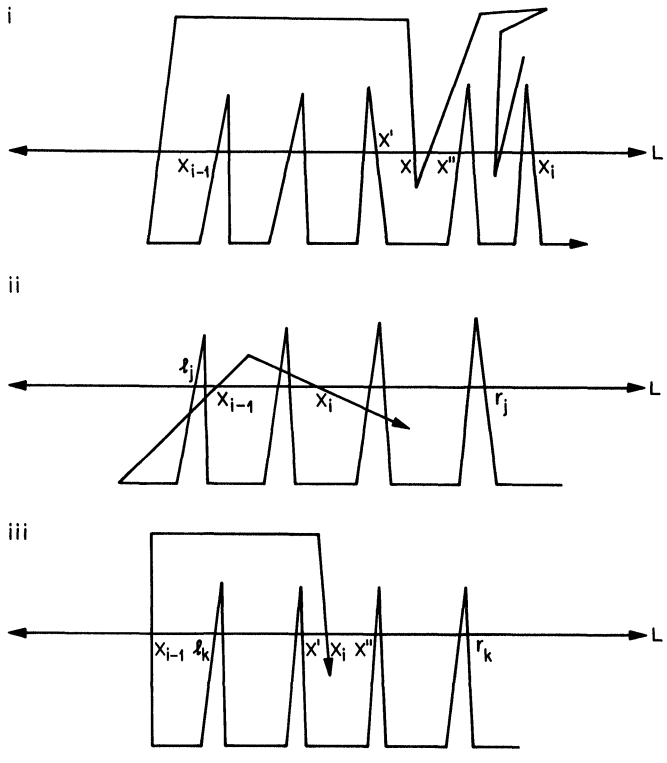

FIG. 7. *Modifications to make Jordan sorting error-correcting.*

$\{x_{j-1},x_j\}$ is special and the crossing of $\{x_{i-1},x_i\}$ and $\{x_{j-1},x_j\}$ can be eliminated by refining $\{x_{j-1},x_j\}$, this will produce a new crossing pair.)

(iii) In Step 3, Case D, if $\{x_{k-1},x_k\}$ is special and $r_k > x_i$, do not halt, but instead call *refine* $(\{x_{k-1},x_k\}, x_i)$. If *refine* returns no pair, halt: $\{x_{k-1},x_k\}$ violates the nonenclosure property. If *refine* returns a pair (x',x''), replace $\{x_{k-1},x_k\}$ in the upper forest by $\{x_{k-1},x'\}$ followed by $\{x'',x_k\}$. Insert $\{x',x''\}$ in the appropriate place in the lower forest (as a sibling or child of the pair containing x_k, whichever is appropriate). Proceed as in the remainder of Step 3, Case D, using $\{x_{k-1},x'\}$ in place of $\{x_{k-1},x_k\}$ and also in place of $\{x_{j-1},x_j\}$ if $\{x_{j-1},x_j\} = \{x_{k-1},x'\}$. That is, if $\{x_{j-1},x_j\} = \{x_{k-1},x_k\}$, replace $\{x_{j-1},x'\}$ in its list of siblings by $\{x_{i-1},x_i\}$ and make $\{x_{j-1},x'\}$ a child of $\{x_{i-1},x_i\}$. If $\{x_{j-1},x_j\} \neq \{x_{k-1},x_k\}$, replace the sublist from $\{x_{j-1},x_j\}$ to $\{x_{k-1},x'\}$ (inclusive) by $\{x_{i-1},x_i\}$, and make the sublist the list of children of $\{x_{i-1},x_i\}$. In either case insert x_i after x' in the sorted list (or after x_0 if $x' < x_0 < x_i$).

The correctness of the error-correcting Jordan sorting algorithm follows from the observation that, while the algorithm is running, a special pair $\{x_{j-1},x_j\}$ can enclose at most one intersection point $x_i \notin \{x_{j-1},x_j\}$. To see this, suppose without loss of generality that $\{x_{j-1},x_j\}$ is a pair in the upper tree. An intersection $x_i \notin \{x_{j-1},x_j\}$ can be inserted between x_{j-1} and x_j in the sorted list because of the addition of a pair $\{x_{i-1},x_i\}$ to the lower tree, but the violation of the enclosure property will be detected when the point x_{i+1} is processed, as illustrated in Figure 7(ii).

The additions necessary to make the algorithm error-correcting cost only $O(1)$ time per point processed and per refinement, not including the time spent inside calls of *refine*. (There are at most two insertions in sibling lists per refinement.) Thus the error-correcting algorithm runs in $O(m)$ time, where m is the number of intersection points in the final refined sequence. In the next section, we shall see how error-correcting Jordan sorting can be used to compute visible pairs.

3. An efficient visibility algorithm. Our algorithm for computing visible pairs follows the outline laid out in Section 1. It is a divide-and-conquer method that cuts up the original polygon into subpolygons, cuts these into smaller subpolygons, and so on, until none of the subpolygons can be further divided. In order to present the details of the method, we must first discuss the structure of the subpolygons, which we call *visibility regions*. The interior of a visibility region is a simply connected subset of the original polygon interior contained between two horizontal lines, denoted by $y = y_{\min}$ and $y = y_{\max}$ (with $y_{\min} < y_{\max}$). We require that the region boundary actually intersect both of these lines. (See Figure 8.)

The boundary of a visibility region consists of connected pieces of the boundary of the original polygon, called *boundary segments*, alternating with segments of the lines $y = y_{\min}$ and $y = y_{\max}$. Each such horizontal segment that is not a single point corresponds to a visible pair. At most one vertex of the original polygon lies on each of the lines $y = y_{\min}$ and $y = y_{\max}$. Although a visibility region is itself a simple polygon, when we speak of its vertices we mean *only* those that are vertices of the original polygon P. A boundary segment begins with a vertex or part of an edge, called a *partial edge*, and ends with a vertex or partial edge. We call the edge of the original polygon that contains such a partial edge an *end edge* of the segment.

We divide the boundary segments into three types:

top: no end edge or vertex intersects the line $y = y_{\min}$;

bottom: no end edge or vertex intersects the line $y = y_{\max}$;

side: one end edge or vertex intersects the line $y = y_{\max}$ and one intersects the line $y = y_{\min}$.

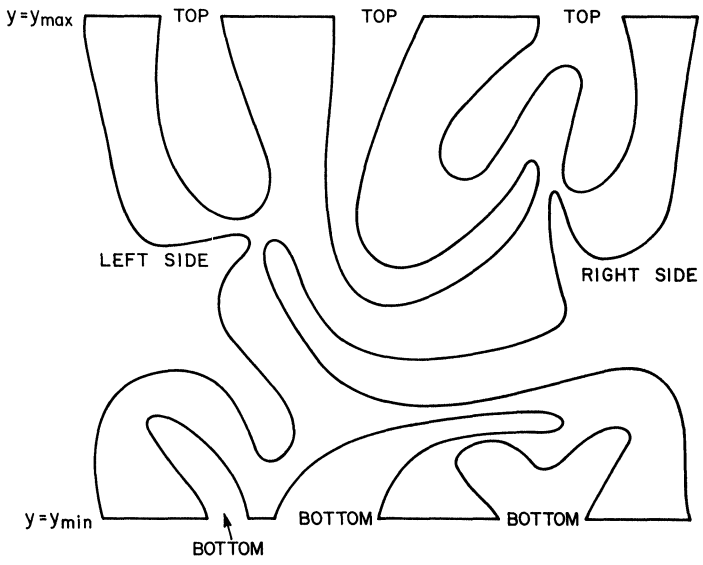

FIG. 8. *Schematic illustration of a visibility region. Each curve represents a segment of the polygon boundary.*

The degenerate case of a top or bottom boundary segment is a single vertex and no partial edges; the degenerate case of a side boundary segment is a single partial edge and no vertices. Clockwise around the boundary of a region, the boundary segments consist of four contiguous parts: a set of top segments, which together with the adjacent pieces of the line $y = y_{\max}$ forms the *top* of the boundary; a side segment, which forms the *right side* of the boundary; a set of bottom segments, which together with the adjacent pieces of the line $y = y_{\min}$ forms the *bottom* of the boundary; and another *side segment*, which forms the *left side* of the boundary. Both side segments must be present; either the top or the bottom or both can be empty.

We shall represent a visibility region by specifying $y_{\min}$ and $y_{\max}$ and the four parts of the boundary (left, right, top, and bottom). We represent the top and the bottom by lists of the boundary segments they contain, in clockwise order around the boundary. We represent the left and right sides by their single boundary segments. Finally, we represent each boundary segment by a list of the vertices in it, in clockwise order around the boundary, together with its end edges (if any). We leave unspecified the implementation of the lists that represent the boundary segments and the top and bottom boundaries of the region; this is the topic of Section 5.

Having discussed the structure of visibility regions, we now need to introduce some terminology concerning the intersections of the boundary of a region with a horizontal line. (See Figure 9.) Let V be a visibility region and let L be a horizontal line that intersects its interior. We partition the boundary segments of V into three types, depending on their relationship to L:

shallow: a segment that does not intersect L;

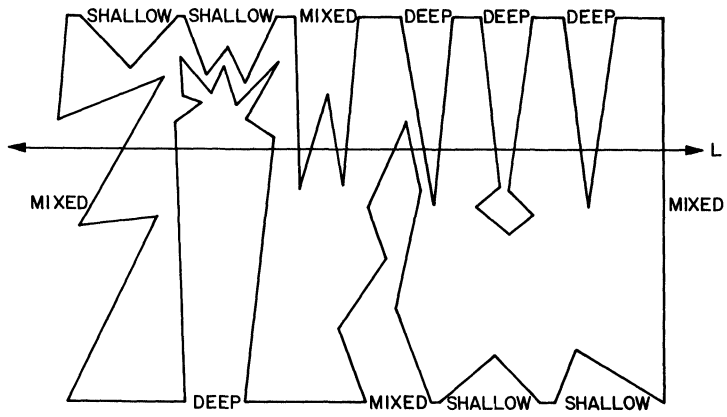

FIG. 9. *Illustrating three kinds of boundary segment. The top group includes a deep section of three segments. Both top and bottom groups include shallow sections of two segments.*

deep: a top segment whose vertices are all strictly below L or a bottom segment whose vertices are all strictly above L;

mixed: any other segment.

A side segment is definitely mixed; a top or bottom segment can be of any type. We define a *shallow section* to be a (contiguous) sublist of shallow boundary segments in the list of boundary segments clockwise around ∂V; we define a *deep section* similarly. A deep or shallow section consists entirely of top segments or entirely of bottom segments. Each of the partial edges of a deep section intersects L and these are the only intersections of the section with L. We classify the intersections of ∂V with L into two types:

nonessential: an intersection within a maximal deep section that is not the first or the last within the section;

essential: any other intersection.

The last issues we must discuss before presenting the visibility algorithm are the notion of a *special pair* and the effect of the *refine* procedure, both of which affect the running of the error-correcting Jordan sorting algorithm. A pair of intersections of ∂V and L is *special* if the intersections are the first and last in some deep section (along ∂V) and *normal* otherwise. (Observe that the intersections of a deep section with L occur in the same order along L as they do along ∂V, or in reverse order.) A call *refine* $(\{x_{i-1},x_i\}, x)$ has the following effect. Points x_{i-1} and x_i are the first and last intersections in some deep section, say S. If S can be split into two deep sections S_1 and S_2 with first and last intersections x_{i-1},x' and x'',x_i, respectively, such that $\{x',x''\}$ encloses x, then *refine* returns (x',x''). If S cannot be so split, *refine* returns no pair. Observe that if a pair (x',x'') *is* returned, $\{x_{i-1},x'\}$ and $\{x'',x_i\}$ are special pairs and (x',x'') is a normal pair, as required by the Jordan sorting algorithm.

There is one more crucial observation about special pairs. Consider a special pair $\{x_{i-1},x_i\}$ that comprises the first and last intersections of a single deep boundary segment. If the segment is a top segment T, then T together with the appropriate seg-

ment of the line $y = y_{max}$ forms a simple closed curve whose interior contains the line segment joining x_{i-1} and x_i and whose exterior contains the boundary of V other than T. By the Jordan curve theorem, $\{x_{i-1}, x_i\}$ can enclose no intersections other than x_{i-1} and x_i. Thus every special pair either can be refined or has the nonenclosure property, as required by the Jordan sorting algorithm.

We are at last ready to discuss the visibility algorithm itself. The input to the algorithm is a single visibility region V. To apply the algorithm to the original polygon, we convert the polygon into a visibility region by dividing its boundary into two side boundary segments whose end vertices are the vertices of minimum and maximum y-coordinate. The y-coordinates of these two vertices become y_{min} and y_{max} for the region. The algorithm is the same as that in Section 1 except that it computes only the essential intersections of ∂V and L in Step 2 and uses Jordan sorting with error-correction in Step 3. That is, it consists of the following five steps:

Step 1. Given V with bounding lines $y = y_{min}$ and $y = y_{max}$, choose a vertex of V having y-coordinate y_{cut} such that $y_{min} < y_{cut} < y_{max}$. If there is no such v, stop: there are no visible pairs to compute. Otherwise, let L be the line $y = y_{cut}$.

Step 2. Find the essential intersections of ∂V and L in the order in which they occur along ∂V.

Step 3. Use Jordan sorting with error-correction to sort by x-coordinate the essential intersections and any others introduced by refinement. Report the visible pairs corresponding to consecutive sorted intersections along L.

Step 4. Slice V along L, dividing V into a collection of subregions.

Step 5. Apply the algorithm recursively to each subregion formed in Step 4.

The observations made above concerning special pairs and refinement imply that the Jordan sorting step works correctly; any nonessential intersection occurs along ∂V between the members of a special pair and is available by refinement if needed.

The last detail we must fill in before undertaking an analysis of the algorithm is the effect of Step 4. The boundaries of the subregions are formed as follows. (See Figure 10.) Split ∂V at each of the intersections sorted in Step 3. Each of the pieces so formed corresponds to a pair in the upper or lower tree constructed by the Jordan sorting algorithm. Each family in each of the trees whose parent has odd depth (counting the dummy roots as being of depth zero) corresponds to a subregion. The boundary of the subregion consists of the pieces of ∂V that correspond to the pairs of the family, in the order in which the members of the family occur in the family list in the lower tree or the reverse of this order in the upper tree, interspersed with appropriate segments of the line $y = y_{cut}$, *with one crucial exception*: if D is a piece of ∂V that corresponds to a deep boundary section, before using D as part of a subregion boundary, each of its segments of the line $y = y_{min}$ or $y = y_{max}$ must be replaced by a corresponding segment of the line $y = y_{cut}$. Observe that this has *no* effect on the representation of the deep boundary section, which means that no change in the data structure representing the section is necessary. Furthermore the visibility regions cut off by this replacement and henceforth ignored are trapezoids, on which the visibility algorithm would terminate in Step 1 were it invoked on them. The visibility algorithm obtains its efficiency by avoiding any computation associated with these trivial trapezoids.

We define y_{min} and y_{max} for the subregions as follows. Consider a subregion above L, which corresponds to a family in the upper tree. The value of y_{min} for this subregion is y_{cut}. The value of y_{max} for the subregion is y_{max} for the region if the parent of the family is at depth one in the upper tree, or otherwise the maximum y-coordinate of a vertex in the piece of ∂V corresponding to the parent of the family. In

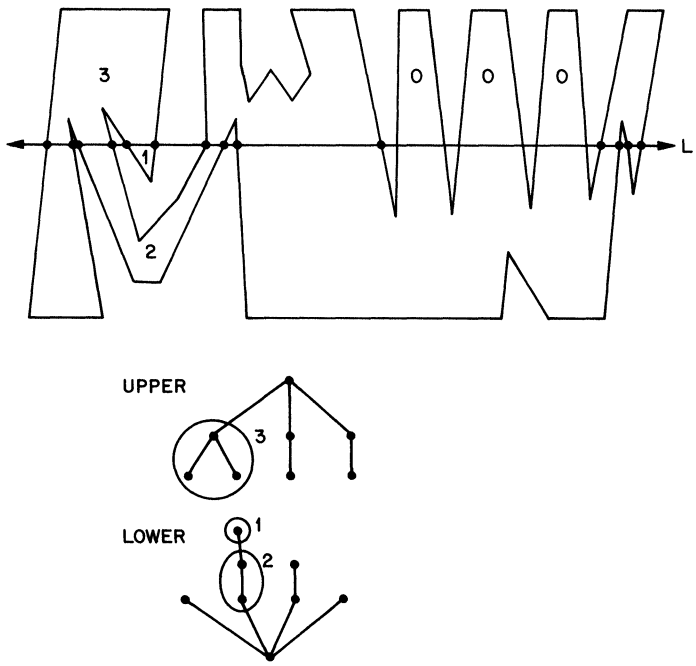

FIG. 10. *Assembling subregions. The family tree nodes associated with several regions are shown. Trapezoids labeled "0" are ignored completely.*

the latter case, the piece of ∂V corresponding to the parent consists of a single piece of a boundary segment of V. We split this piece at the vertex with maximum y-coordinate to form two pieces, which become the side boundary segments of the subregion; the top of the subregion is empty. The definitions are symmetric for subregions below L.

Let us restate the difference between the algorithm above and the one outlined in Section 1. In the former, a maximal deep section is treated as if it had only two intersections with L (the essential ones), until it is discovered that some intersection not in the section is enclosed by these two. This approximation to the truth works because the intersections in the section occur in the same order along the section as they do along L (or in reverse order). If no "foreign" intersections intervened, the algorithm of Section 1 would merely chop the section between each pair of contiguous boundary segments in Step 2 and put them back together in exactly the same order in Step 4. The new algorithm avoids this unnecessary work.

We now want to quantify the work saved by the new algorithm. Our main result is that the total number of visible pairs reported during the processing of an n-vertex polygon is $O(n)$. This implies that the time spent doing Jordan sorting, not including calls of *refine*, is $O(n)$.

LEMMA 3. *The processing of an n-vertex polygon requires at most* $n-2$ *invocations of Steps* 2–4.

Proof. Each vertex except the ones of maximum and minimum y-coordinate can be selected as v in Step 1 at most once. □

LEMMA 4. *Consider a single invocation of Step* 3. *Let k be the number of visible pairs reported during this invocation that were not reported during previous invocations. The total number of intersections sorted during this invocation, including those introduced by refinement, is $O(k+1)$.*

Proof. First we count the essential intersections. We call an essential intersection x *good* if x is a vertex (i.e. $x=v$) or if the two vertices preceding and following x along ∂V, say v' and v'', are in the same part of ∂V (top, bottom, left, or right) and not strictly on the same side of L. Otherwise x is *bad*. We claim that if x is a good intersection other than v, then the edge that contains x belongs to a newly reported visible pair. This is true if the part of ∂V from v' to v'' consists of the line segment joining v' and v'', since the visible pair containing the edge from v' to v'' that will be reported is the first one reported containing that edge. It is also true if the part of ∂V from v' to v'' consists of a partial edge from v' to the line $y = y_{\max}$ (or $y = y_{\min}$), a segment of the line $y = y_{\max}$ (or $y = y_{\min}$, respectively), and a partial edge from the line $y = y_{\max}$ (or $y = y_{\min}$, respectively) to v''. To see this, suppose without loss of generality that v' is strictly below L and ∂V contains a partial edge from v' to the line $y = y_{\max}$. (See Figure 11.) Intersection x is on this partial edge. Along the line L, the edge from v' sees something other than the edge into v'', and thus will be contained in a newly reported visible pair, since if two edges see each other horizontally, the part of each edge that sees the other is connected. Thus in either case the claim is true. The claim implies that the number of good intersections is $O(k+1)$.

Consider the bad intersections. Any mixed boundary segment contains at most two bad intersections (the first and last in the segment) and, if it is a top or bottom segment, at least one good intersection. Any maximal deep section contains at most two bad intersections (the first and last in the section). Such a section either (i) is followed by a shallow segment, (ii) is followed by a mixed top or bottom boundary segment, or (iii) contains the last segment on its side. In case (i) the last intersection is good. The number of bad intersections in case (ii) is $O(k+1)$, since the mixed segment contains at least one good intersection. At most four bad intersections (the last two within each of the top and bottom boundaries) can fall into case (iii). There are also at most two bad intersections within each of the left and right sides. Thus the number of bad intersections is $O(k+1)$.

It remains for us to count the nonessential intersections introduced by refinement. Suppose that a call *refine* $(\{x_{i-1},x_i\}, x)$ returns a pair x',x''. Both of the edges that contain x' and x'' will be contained in newly reported visible pairs, since along L they

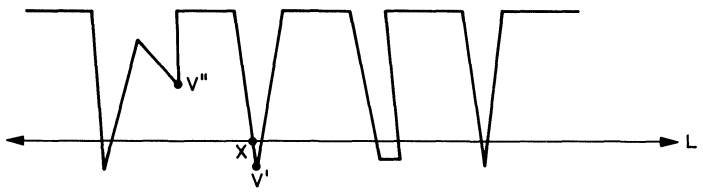

FIG. 11. *The edge that contains x belongs to a newly reported visible pair.*

see something other than each other. Thus the number of intersections introduced by refinement is $O(k)$. □

The following theorem summarizes our analysis of the algorithm so far.

THEOREM 1. *In processing an n-vertex polygon, the visibility algorithm reports $O(n)$ visible pairs and spends $O(n)$ time in Jordan sorting, not including its calls to refine. This includes all processing of subregions.*

Proof. Immediate from Lemmas 1, 3 and 4. □

4. Use of balanced divide and conquer. The visibility algorithm of Section 3 can in the worst case generate regions containing a total of $\Omega(n^2)$ boundary segments (counting a boundary segment each time it occurs in a region). As we shall see in Section 5, obtaining an $O(n \log\log n)$-time implementation of the algorithm requires reducing the total number of boundary segments to $O(n \log n)$. We do this by refining the algorithm so that it uses a balanced divide-and-conquer strategy. The refinement consists of choosing the vertex v carefully in Step 1. Roughly speaking, we want to slice a region with many boundary segments so that at most a proper fraction of the segments end up in any one of the subregions. A choice that works and that can be made quickly is the following:

Choose splitting vertex. Let region V have t top boundary segments and b bottom boundary segments. Suppose $t \geqslant b$. (Otherwise, proceed symmetrically.) If $t \leqslant 2$, choose any vertex v whose y-coordinate is not in $\{y_{\min}, y_{\max}\}$. Otherwise, divide the list of top boundary segments into three sublists, with the first and last containing $\lfloor t/3 \rfloor$ segments and the middle one the remainder. Among the segments in the middle sublist, choose as v the vertex whose y-coordinate is minimum.

We call the visibility algorithm refined to use this selection strategy the *balanced division algorithm*; we call the algorithm with an arbitrary selection strategy the *generic* algorithm. In the analysis to follow, we regard two boundary segments as distinct only if their vertex sets or their end edges (if there are any end edges) are different.

LEMMA 5. *In processing an n-vertex polygon, the generic visibility algorithm creates $O(n)$ distinct boundary segments altogether.*

Proof. The only way the algorithm can create new boundary segments is by splitting an old boundary segment in two, at an intersection point that is input to the Jordan sorting algorithm or at the vertex of maximum or minimum y-coordinate in a subregion. By Theorem 1 there are $O(n)$ such splitting points. Hence there are $O(n)$ distinct boundary segments. □

LEMMA 6. *Let V be a region that has s boundary segments. If V is sliced using balanced division, then no subregion contains more than $7s/8$ of the original boundary segments of V.*

Proof. Let V contain t top boundary segments and b bottom boundary segments. We have $s = b + t + 2$. Suppose without loss of generality that $t \geqslant b$. If $t \leqslant 2$ the lemma is immediate, since some boundary segment is cut into new boundary segments distinct from the original; thus the original appears in no subregion. Suppose $t \geqslant 3$. Consider where the top boundary segments of V end up after slicing. A segment that is mixed with respect to the slicing line L is cut into new boundary segments distinct from the original; thus the original appears in no subregion. There are at most $2\lfloor t/3 \rfloor$ deep top boundary segments, which implies that each subregion below L contains at most $b + 2t/3 + 2 \leqslant 7s/8$ of the boundary segments of V. A deep top segment cannot end up in a subregion above L; only a part of it, constituting a distinct boundary segment, can. Thus among the top segments, only the shallow ones can end up in regions above L. A set of shallow segments can end up in the same subregion only if it forms a shallow section. By the choice of L, any such shallow section can contain at

most $t - \lfloor t/3 \rfloor - 1 \leqslant 2t/3$ top boundary segments of V. Thus any subregion above L contains at most $b + 2t/3 + 2 \leqslant 7s/8$ of the boundary segments of V. □

THEOREM 2. *When the balanced division algorithm processes an n-vertex polygon, the sum over all regions of the number of boundary segments per region is* $O(n \log n)$.

Proof. One way to prove this theorem is to write down a recurrence based on Lemma 6 and solve it. Instead, we shall use an amortization argument based on a credit analysis (see [26]). When the balanced division algorithm creates a distinct new boundary segment, we assign $15 \log_{16/15} n$ credits to the segment. Each subsequent time that the segment appears in a subregion, we remove a credit. We shall show that the number of credits always remains nonnegative, which implies by Lemma 5 that the sum over all regions of the number of boundary segments is $O(n \log n)$.

To show that the number of credits remains nonnegative, we actually prove the following stronger *credit invariant*: a region that has s boundary segments has at least $s \log_{16/15} s$ credits. The invariant is certainly true initially. Suppose it is true before some invocation of Steps 1-4. Let V be the region to be subdivided, and let s be its number of boundary segments. Before the subdivision, V has at least $s \log_{16/15} s$ credits, of which we allocate $\log_{16/15} s$ to each boundary segment. One of these pays for the appearance of the segment in V, leaving $(\log_{16/15} s) - 1 = \log_{16/15}(15s/16)$ as its contribution to the credits of the subregion in which it appears. Consider a subregion V' formed by splitting V. Suppose its boundary contains p of the boundary segments of V and q newly created boundary segments; let $s' = p + q$. To verify that V' has $s' \log_{16/15} s'$ credits, there are two cases to consider. If $q > s'/15$, then the total number of credits is at least the number of credits assigned to new segments:

$$15q \log_{16/15} n \geqslant s' \log_{16/15} n \geqslant s' \log_{16/15} s'.$$

If $q \leqslant s'/15$, then since $p \leqslant 7s/8$ by Lemma 6, we have $s' \leqslant 15s/16$. The total number of credits is

$$\begin{aligned} p \log_{16/15}(15s/16) + 15q \log_{16/15} n &\geqslant p \log_{16/15}(15s/16) + 15q \log_{16/15}(15s/16) \\ &\geqslant s' \log_{16/15}(15s/16) \geqslant s' \log_{16/15} s'. \end{aligned}$$

By induction on the number of steps, the credit invariant is always true, from which the theorem follows. □

5. Representation of the boundary using finger search trees. We have now almost completed our presentation of the visibility algorithm. The task that remains is to choose a data structure for the lists that represent the boundary segments and boundary groups, and to analyze the effects of this choice. To represent both kinds of lists we use heterogeneous finger search trees (see the appendix). As we shall see, this gives an overall $O(n \log \log n)$ running time for the balanced division visibility algorithm.

We represent each boundary segment by a heterogeneous finger search tree in which each leaf contains a vertex in the segment. Left-to-right order in the tree corresponds to the order of the vertices along the segment. In addition, if the segment has one or two end edges, we store these edges with the tree. Within the tree, we maintain two heap orders, both with respect to the y-coordinates of the vertices. One is by increasing y-coordinate, the other is by decreasing y-coordinate. That is, each node in the tree contains the maximum and minimum y-coordinates of all leaves reachable from it in the tree. (See Figure 12.) This allows us, for any given value of y, to find the leftmost (or rightmost) vertex with y-coordinate less than (or greater

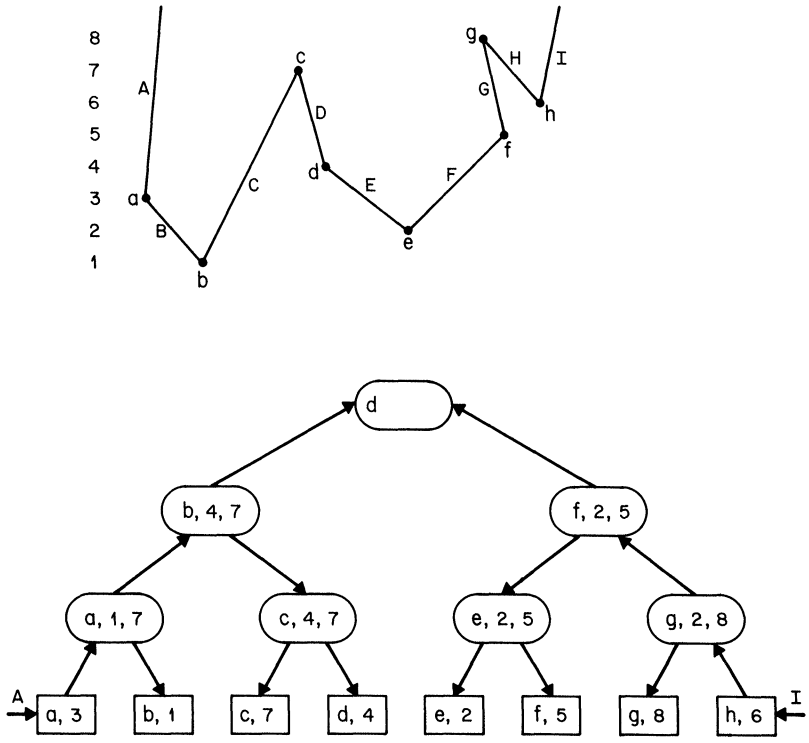

FIG. 12. *A boundary segment and its representation as a heterogeneous finger search tree. (The drawing conventions are explained in Figure 22.)*

than) the given value, in $O(1+\log(\min\{d, s-d\}+1))$ time, where s is the total number of vertices stored in the tree and the one found is the dth. We can also find the vertex of maximum (or minimum) y-coordinate in the same time. The amortized time to split the tree at the dth out of s vertices is also $O(1+\log(\min\{d, s-d\}+1))$.

We represent each list of top boundary segments or bottom boundary segments constituting the top or bottom boundary of a region by a heterogeneous finger search tree in which each leaf represents a boundary segment. Left-to-right order in the tree corresponds to the order of the boundary segments clockwise around the boundary. Each leaf contains a pointer to the tree representing the corresponding boundary segment, as well as the end vertices or edges of the segment, and the maximum and minimum y-coordinates of the vertices within the segment. We think of each node in the tree as representing the section (sublist of boundary segments) corresponding to the set of leaves reachable from the node. We store in each node the first and last end vertices or edges of the corresponding section, the number of segments in the section, and the maximum and minimum y-coordinates of vertices in the section. (See Figure 13.) All these values can be updated bottom-up in the interior of the tree and top-down along the left and right paths.

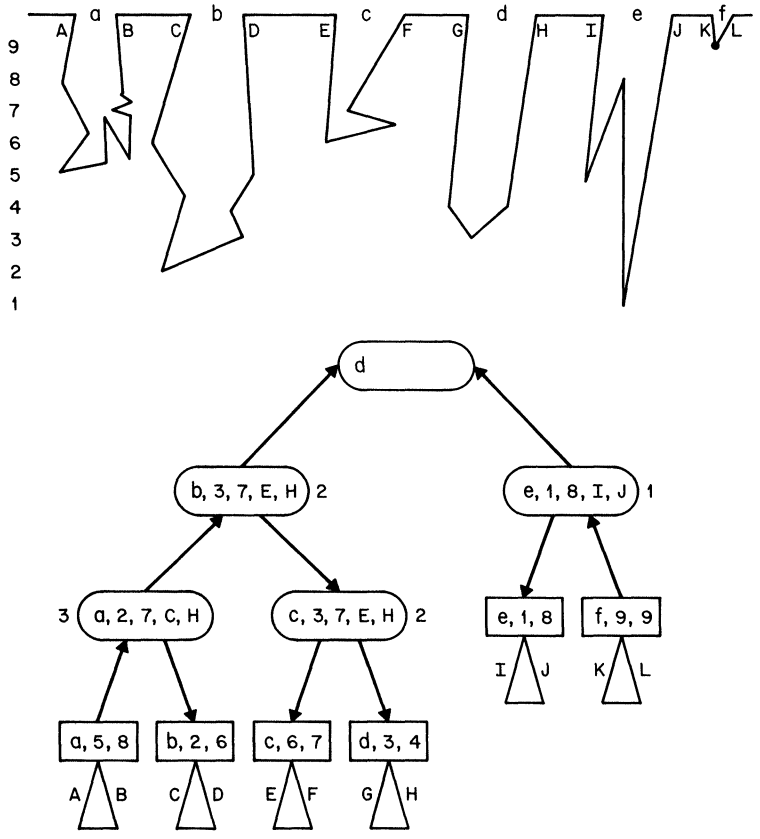

FIG. 13. *A group of boundary segments and its representation as a heterogeneous finger search tree. (The drawing conventions are explained in Figure 22; capital letters represent end edges.)*

All of the following operations can be performed in $O(1+\log(\min\{d, s-d\}+1))$ time, where the segment found is the dth out of s:

(i) Find the leftmost (or rightmost) segment in the tree that contains a vertex with y-coordinate less than (or greater than) a given value;

(ii) Find the dth segment;

(iii) Suppose all the segment vertices lie strictly above (or strictly below) a given horizontal line L, and that all end edges of the segments cross L. Given a value x, find the leftmost (or rightmost) segment in the tree having an edge whose intersection with L has x coordinate less than (or greater than) x. (All four possibilities, leftmost less than, leftmost greater than, etc., are allowed.)

In addition, inserting a new segment next to the dth out of s or splitting at the dth segment out of s takes $O(1+\log(\min\{d, s-d\}+1))$ amortized time. Concatenating two trees takes $O(1)$ amortized time.

Let us examine the manipulations of the boundary data structures required to carry out the steps of the balanced division visibility algorithm. We describe these step by step, including a timing estimate for some of the computations.

Step 1. Given V, determine the number of top and bottom boundary segments in its boundary, say t and b, respectively ($O(1)$ time). Assume $t \geqslant b$. (The other case is symmetric.) If $t \leqslant 2$, choose a segment of the boundary that contains some vertex with y-coordinate strictly between $y_{\min}$ and $y_{\max}$, and select as v the first such vertex in the segment ($O(1)$ time). If $t \geqslant 3$, find the minimum y-coordinate of the vertices in the middle third of the top boundary segments (time to be analyzed below). This defines the slicing line L.

Step 2. Split the tree representing the top boundary between each pair of segments that differ in type among the types shallow, deep, and mixed (time to be analyzed below). This splits the top boundary list into mixed segments and maximal deep and shallow sections. Repeat this for the tree representing the bottom boundary (time to be analyzed below). For each mixed segment, split its tree between each pair of consecutive vertices, one on each side of L (time to be analyzed below). Split the tree that contains the vertex lying on L at that vertex, putting the vertex in both of the resulting trees (time to be analyzed below). Each of the splits performed corresponds to an essential intersection of the boundary with L. Form a list of these intersections, in the order in which they occur along the boundary, by examining the list of trees representing the deep sections and the new boundary segments that have been formed by splitting mixed segments ($O(1)$ time per essential intersection).

Step 3. To execute a call *refine* $(\{x_{i-1}, x_i\}, x)$, examine the tree representing the deep section whose first and last intersections with L are x_{i-1} and x_i. Assume $x_{i-1} < x_i$. (The other case is symmetric.) Find in this tree the rightmost segment whose first intersection with L is less than x (time to be analyzed below). If the last intersection of this segment with L is greater than x, return no pair ($O(1)$ time). Otherwise, split the section between this segment and the next one, and return as x' and x'' the last intersection of this segment and the first intersection of the next one (time to be analyzed below).

Step 4. For each pair in the upper tree whose depth is odd and greater than one, split the tree representing the corresponding boundary segment at its vertex of maximum y-coordinate (time to be analyzed below). Put the splitting vertex in both of the resulting trees. Proceed symmetrically for the boundary segments corresponding to pairs in the lower tree. For each family in both trees, concatenate the boundary segments and deep sections corresponding to the pairs in the family to form the boundary of the subregion corresponding to the family ($O(1)$ time per pair). Because the input polygon is simple, the order in which polygon vertices appear on the boundary of the subregion is consistent with their order in the original polygon. (See Figure 14.) This means that the concatenation will never need to reverse the order of boundary segments.

Now let us analyze the running time of this implementation of the balanced division algorithm. Observe that for each segment found within a top or bottom boundary, and for each vertex found within a segment, a split occurs at that segment or vertex. Since the timing estimates for finding and splitting are the same, the time spent splitting dominates the time spent finding such segments and vertices, including the time to find the y-coordinates of the splitting vertices in Step 1. Thus, by the timing estimates above and Theorem 1, the total running time of the algorithm is $O(n)$ plus at most a constant times the time for tree insertions, splits, and concatenations.

It remains to estimate the time for tree update operations. Every insertion is at one end of a finger search tree and thus takes $O(1)$ amortized time. The updates on

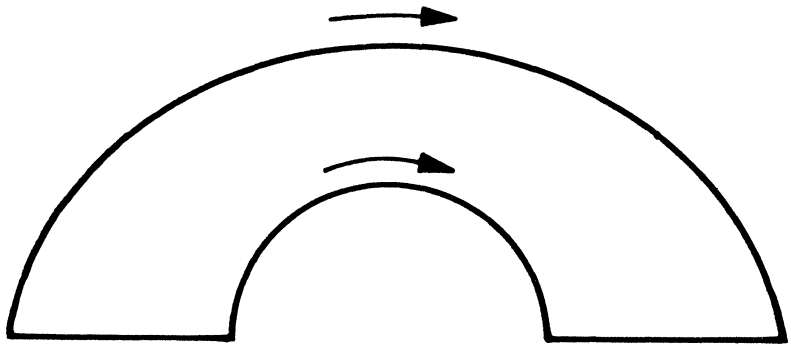

FIG. 14. *A subregion that would cause trouble for Step* 4. *The arrows indicate the direction of the two boundary segments in the original polygon. Such subregions cannot occur for simple polygons.*

the trees representing segments are only splits and insertions, of which there are $O(n)$. The total running time of these updates obeys a recurrence that is essentially the same as (1) (Section 1) and is thus $O(n)$.

The updates on the trees representing top and bottom boundaries include concatenations, and recurrence (1) does not apply. To analyze these operations, we first observe that the amortized time per concatenation or insertion is $O(1)$. Since the total number of such operations is $O(n)$, their total time is $O(n)$.

To analyze the $O(n)$ splits, consider a particular visibility region V_i that has a total of s_i boundary segments. The total time to split the two trees representing the top and bottom boundaries is $O(k_i + \Sigma_{j=0}^{k_i+1} \log s_{i,j})$, where k_i is the total number of splits and $s_{i,0}, s_{i,1}, \ldots, s_{i,k_i+1}$ are the numbers of segments in each of the trees that are created by the splits. (Starting from two trees, k_i splits produce k_i+2 trees; if there is only one tree initially, we take $s_{i,k_i+1} = 1$.) We have $s_{i,j} \geqslant 1$ for all j, and $\Sigma_{j=0}^{k_i+1} s_{i,j} \leqslant s_i+1$. Since the logarithm is a concave function, the estimate of splitting time is maximized when all the pieces are of equal size, giving a bound of $O(k_i + (k_i+2) \log ((s_i+1)/(k_i+2)))$.

We evaluate the total time to perform all $O(n)$ splits, $O(\Sigma_i (k_i + (k_i+2) \log ((s_i+1)/(k_i+2))))$, in two parts. Call V_i a *good region* if $k_i \leqslant s_i/(\log n)^2$, and *bad* otherwise. By Theorem 2, $\Sigma_i s_i = O(n \log n)$, so the total number of splits that occur while processing good regions, $\Sigma_{k_i \leqslant s_i/(\log n)^2} k_i$, is $O(n/\log n)$; using $O(\log n)$ as a generous time bound for these splits, the total time to split good regions is $O(n)$. It remains to bound the time to split bad regions:

$$O\left(\sum_{k_i \geqslant s_i/(\log n)^2} (k_i + (k_i+2) \log ((s_i+1)/(k_i+2)))\right)$$

$$= O\left(\sum_{k_i \geqslant s_i/(\log n)^2} (k_i + (k_i+2) \log \log n)\right) = O(n \log \log n),$$

since $\Sigma_i k_i = O(n)$.

We conclude that the total running time of the visibility algorithm with balanced division is $O(n \log \log n)$.

6. Remarks, applications and open problems. We have presented an $O(n \log \log n)$-time algorithm for computing horizontally visible edge-vertex pairs

inside a simple polygon. By the linear-time reduction of triangulation to the visibility problem [5],[9] we obtain an $O(n \log\log n)$-time triangulation algorithm. The main ingredients of our algorithm are Jordan sorting, balanced divide and conquer, and finger search trees. It is intriguing to note that both the Jordan sorting algorithm and the visibility algorithm use finger search trees, but of two different kinds. The Jordan sorting algorithm requires fast access in the vicinity of any position in the tree but does not need to search on a secondary heap order. Homogeneous finger search trees satisfy these requirements. The visibility algorithm itself does need to search on secondary heap orders, but requires fast access only in the vicinity of the first and last positions. Heterogeneous finger search trees satisfy these requirements. Our algorithm exploits to the fullest the properties of these structures.

Finger search trees are sufficiently complicated that one would probably not want to use them in an actual implementation. The dynamic optimality conjecture of Sleator and Tarjan [23] suggests that the $O(n \log\log n)$ time bound is still valid if splay trees (a form of self-adjusting search tree) are used in place of finger search trees. The use of splay trees might lead to a practical implementation of our algorithm, although this must be verified by experiment. Other minor changes in the algorithm might be useful in practice. We leave this as a topic for future research.

Our visibility algorithm can be modified to accommodate vertices having the same y-coordinate. To handle the resulting tangent intersection points in the Jordan sorting step (Section 2), we represent such a point x_i by a dummy pair (x_i,x_i) which we add to the lower tree if the tangency is on the top side of the splitting line L, or to the upper tree otherwise. The remaining changes to the algorithm are straightforward. (See e.g. [30].)

An efficient triangulation algorithm has a number of applications in computational geometry. These applications typically involve one (or possibly a few) triangulations and some linear-time pre- and postprocessing. Our triangulation algorithm gives $O(n \log\log n)$-time algorithms for these applications. Any improvement in the time to triangulate would give corresponding improvements in the applications. Such applications include:

(i) several polygon decomposition problems [9] (where minimality, as in [17], is not required);

(ii) regularizing (or triangulating) a planar subdivision that is given as a connected planar graph [8];

(iii) computing the internal distance between two points in a polygon, and finding the point visibility polygon for a point inside the polygon [3];

(iv) solving the single source shortest path problem inside a polygon, and computing internal visibility information from an edge inside a polygon [11];

(v) testing two polygons for intersection, and decomposing simple splinegons [24] into a union of differences of unions of convex sets [7];

(vi) determining translation separability of two simple polygons [1];

(vii) finding a shortest watchman route in a simple rectilinear polygon ([6, Thm. 3]).

An important application of our visibility algorithm is to test whether an n-vertex polygon P is simple, and to exhibit a self-intersection of ∂P if it is not simple. We shall show how to modify our algorithm to perform these tasks in $O(n \log\log n)$ time.

Even though the error-correcting Jordan sorting algorithm detects some instances of nonsimplicity as uncorrectable crossings or violations of the nonenclosure property, the successful completion of the visibility algorithm is not proof against nonsimplicity of the input polygon. Among the problems with which a guaranteed simplicity test

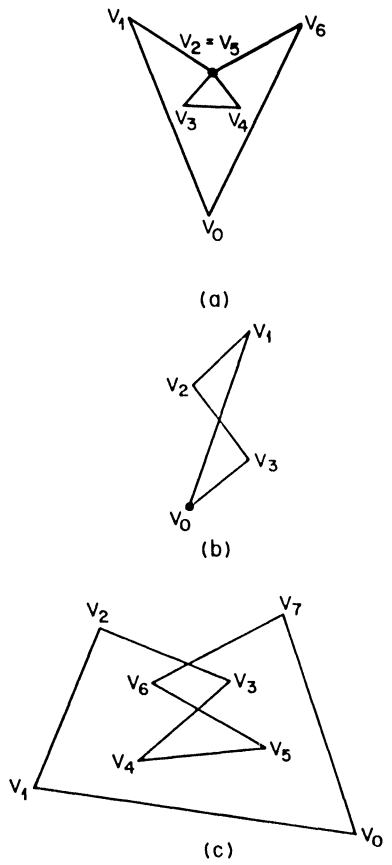

FIG. 15. *Nonsimple polygons that cause no trouble in the algorithm for computing visibility information. In* (c), *a slice through* v_5 *separates the polygon into three simple pieces.*

must cope are polygons that are self-tangent at a vertex (Figure 15a), polygons for which an interior cannot be defined by any consistent labeling of the edges (Figure 15b), and nonsimple polygons that can be sliced into simple pieces (Figure 15c).

Our algorithm for testing simplicity is as follows. First we check that no two consecutive edges of P intersect in more than their common endpoint. Next, we run the visibility algorithm on the polygon P and on its "inside-out" partner Q, defined as in the proof of Lemma 2. We abort the algorithm and declare that P is nonsimple if one of the following cases occurs:

(i) the Jordan sorting step finds an intersection point common to two parts of P other than an endpoint of two consecutive edges (see Figure 15a);

(ii) the Jordan sorting step detects an uncorrectable crossing or violation of the nonenclosure property;

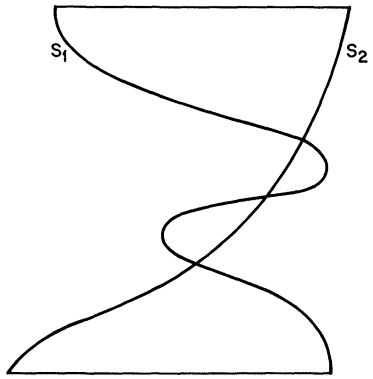

FIG. 16. *The ordering of the four corners of this visibility region implies that it is not simple.*

(iii) a subregion is constructed in Step 4 whose side boundary segments, say S_1 and S_2, are known to cross, because the intersections of S_1 and S_2 with the top bounding line are in the opposite order from the order of their intersections with the bottom bounding line (see Figure 16).

If the visibility algorithm runs to completion on both P and Q, we declare that P is simple.

In our discussion of Step 4 in Section 4, we noted that if the polygon is simple, then the order of vertices along the boundary segments is consistent with the order of the boundary of the subregion being reassembled. It is conceivable if the polygon is not simple that Step 4 could be presented with a subregion whose boundary segments appear in an anomalous order, as in Figure 14. Fortunately, this situation cannot in fact occur: the existence of such a nested pair implies by the Jordan curve theorem that the current region being processed has a nonsimple boundary, and indeed that the segments of the boundary defined by the slicing line do not have the noncrossing property. Therefore, the nonsimplicity will be detected in Step 3.

THEOREM 3. *The simplicity-testing algorithm is correct.*

Proof. Certainly if the simplicity-testing algorithm declares that P is not simple then ∂P has a self-intersection. Suppose the algorithm reports that P is simple. The visibility computations in the algorithm produce two sets of regions. Let $\mathbf{P}$ and $\mathbf{Q}$ be the sets of regions produced when the visibility algorithm runs on P and Q, respectively; each region in $\mathbf{P}$ and $\mathbf{Q}$ is either a trapezoid or a triangle. Some of the regions in $\mathbf{Q}$ are bounded by one or more edges of $Q-P$, i.e., by edges that were added to invert polygon P. Let $\mathbf{Q}'$ be the set of regions formed by taking each region in $\mathbf{Q}$ and extending it to infinity in the direction of any edge in $Q-P$. Regions in $\mathbf{Q}'$ can be trapezoids, triangles, halfplanes, or infinite regions bounded by two horizontal lines and part of a side of P.

It is tempting to say that the regions in $\mathbf{P}$ partition the "interior" of P, but we cannot say this, because we do not yet know that P *has* an interior. However, because of the way in which the regions in $\mathbf{P}$ were produced, we know that they can be glued

together along shared horizontal visibility edges to form a region that is topologically equivalent to a disk. This is necessary but not sufficient for P to be simple (consider the polygon in Figure 15c). The visibility algorithm could be modified easily to produce this "gluing," or, more properly, its dual graph, in which regions are vertices, and regions that share a horizontal visibility edge are joined by an edge in the dual graph.

Since the visibility algorithm succeeded, we also know that the regions in $\mathbf{Q}$ can be glued together along horizontal visibility edges to form a region that is topologically equivalent to the disk. This gluing can be extended naturally to $\mathbf{Q}'$, which is topologically equivalent to the punctured plane. In what follows, we use the regions in $\mathbf{P}$ and $\mathbf{Q}'$ to construct a mapping from the plane onto itself.

Let C be a circle in the plane, and choose n distinct points on C corresponding to the vertices of P; this induces a natural correspondence between points of ∂P and points of C. For each vertex-edge or edge-edge visible pair in $\mathbf{P}$ reported by the visibility algorithm, connect corresponding points on C by a path through the interior of C; make all these paths disjoint except for corresponding endpoints. (The dual graph of the regions in $\mathbf{P}$ provides a natural way to do this constructively. Processing a vertex of degree one in the dual requires that we draw a path between two points on C; that path divides the disk into two parts, one of which can be discarded and never enters into further computation of the mapping. Thus we can perform the complete construction by processing and deleting vertices of degree one from the dual until the dual is empty.) These paths divide the interior of C into regions corresponding to the regions of $\mathbf{P}$. Similarly, for each piece of visibility information in $\mathbf{Q}'$, construct a corresponding path joining two points of C and passing through the exterior of C. If the piece of information represents a vertex or edge that sees arbitrarily far to the left or right, the corresponding path leads from C to infinity. Make all the paths on the exterior of C noncrossing. This partitions the exterior of C into regions corresponding to the regions of $\mathbf{Q}'$. (See Figure 17.)

We can construct a continuous mapping h of the plane onto itself that takes each region of the interior of C onto the corresponding region of $\mathbf{P}$ and each region of the exterior of C onto the corresponding region of $\mathbf{Q}'$. The mapping is onto because every point in the plane is in some region of $\mathbf{P}$ or $\mathbf{Q}'$: For any point x, move horizontally right from x until hitting ∂P. If the right side of ∂P is hit (with respect to clockwise order around ∂P), x is in some region of $\mathbf{P}$. Otherwise x is in some region of $\mathbf{Q}'$. If ∂P is not hit, x is in some region of $\mathbf{Q}'$.

Because $\mathbf{P}$ and $\mathbf{Q}'$ are finite and the preimages of distinct regions have disjoint interiors, the mapping h is a covering map from the plane onto itself: any point x in the plane has an open neighborhood N such that $h^{-1}(N)$ is the disjoint union of a finite number of open sets [21]. Moreover, since the domain of h is connected, every point in its range is the image under h of the same number of points of its domain [21, ex. 8-3.5, p. 336]. Since any point in the open half plane that lies above the horizontal line through the highest vertex of P is the image under h of only one point, h is $1-1$. Thus h is a homeomorphism of the plane onto itself. This proves that ∂P is homeomorphic to C, so P is simple. □

If the simplicity-testing algorithm reports that P is not simple, we can produce a witness to its nonsimplicity in $O(n)$ additional time. If the nonsimplicity is detected in Case (i), the self-intersection is available immediately. Otherwise (if Case (ii) or (iii) occurs) we can find one or two bounding lines and two boundary segments S_1 and S_2 with ends on the bounding lines such that the two boundary segments are guaranteed to cross. There are seven possible configurations of the two boundary seg-

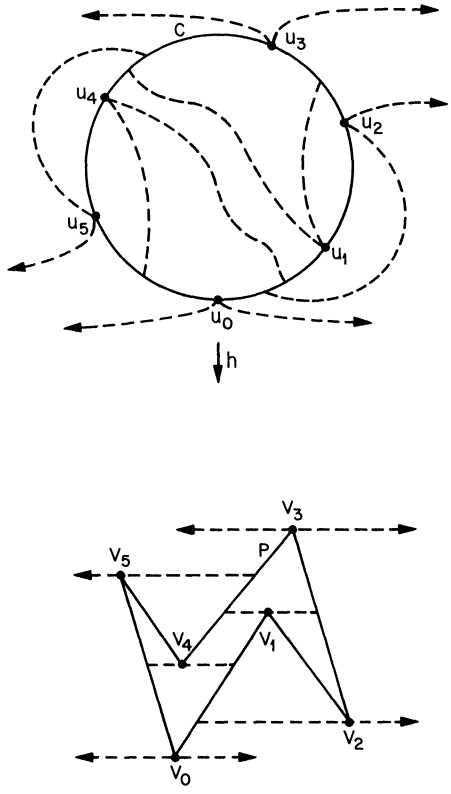

FIG. 17. *Illustrating the mapping h constructed in the proof of Theorem* 3. *For* $0 \leqslant i \leqslant 5$, $h(u_i) = v_i$. *Dashed paths in the upper figure correspond to vertex-edge visibility segments in the lower figure. Arrowheads indicate paths to infinity.*

ments, illustrated in Figures 18 and 15b. We apply the following three steps repeatedly until an explicit crossing is found:

Step 1. Choose a vertex on $S_1 \cup S_2$ not having maximum or minimum y-coordinate. Let L be the horizontal line through this vertex.

Step 2. Find all intersections of S_1 and S_2 with L, in the order in which they occur along S_1 followed by S_2.

Step 3. Jordan sort the intersections. The Jordan sorting step must either find an explicit self-intersection or a violation of the noncrossing property. If such a violation is found, let S_1' and S_2' be the crossing subsegments. Replace S_1 and S_2 by S_1' and S_2'. If S_1' and S_2' are both single partial edges, report their intersection.

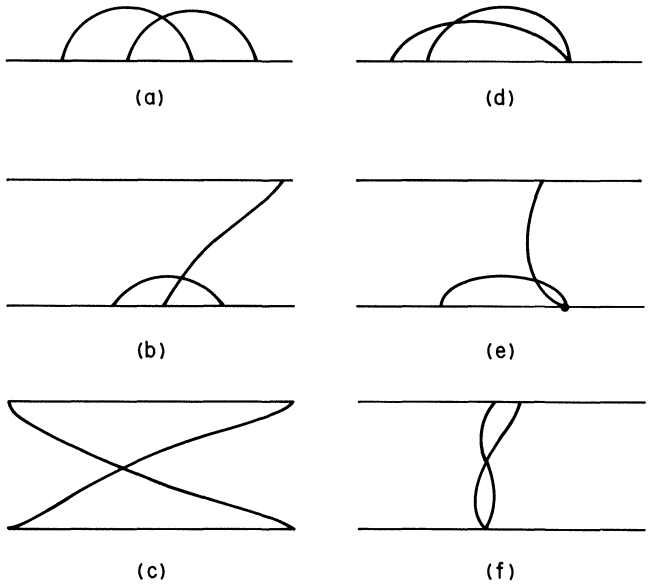

FIG. 18. *Possible configurations of crossing boundary segments:* (a) *four points on one bounding line;* (b) *three points on one bounding line and one on the other;* (c) *two points on each bounding line;* (d), (e), *and* (f) *are degenerate cases of* (a), (b), *and* (c), *respectively. Figure* 15b *depicts a degenerate case of* (f). *Degenerate cases can be detected by examining the interior angle at the point or points of degeneracy.*

An analysis like that in Sections 3–5 shows that this postprocessing takes $O(n)$ time if the segments are represented by finger search trees. (Concatenation of finger search trees and balanced division are not needed.)

The algorithms for testing simplicity and producing a witness to nonsimplicity are easily extended to work on connected polygonal paths that are not closed. The first step of the extension is to draw a line through the two endpoints of the path, thus chopping the path into boundary segments that lie entirely on one side of the line, and to Jordan sort the points of intersection between the path and the line. If the Jordan sorting detects a violation of the noncrossing property, the algorithm for finding a crossing can be applied directly to the two boundary segments involved. Otherwise we use processing akin to that in Step 4, augmented to cope with boundary reversals as in Figure 14, to construct polygons that lie entirely on one side of the line; the path is simple if and only if each of those polygons is simple.

The algorithms for computing horizontal visibility information, for testing simplicity, and for producing a witness to nonsimplicity, can be extended to work on curves that obey certain mild restrictions. (See e.g. [22],[24].) To prepare the curve for processing add vertices to each edge to form a curve each of whose edges is monotone in the y-direction. The algorithms can now be run directly (given suitable procedures for

computing the intersection of a curved edge and a line), because the edges of the object have the property that if a collection of edges crosses two horizontal lines then the edges cross both lines in the same order. The other extension occurs in simplicity testing: we must check that the left and right sides of each trivial visibility region do not cross. If all of the output regions have this property, then the curve is simple; otherwise we have an immediate witness to its nonsimplicity. Both the preparation of the curves and the postprocessing of the output regions can be performed in $O(n)$ time.

Several open problems remain. First and foremost, of course, is determining whether there is a linear-time triangulation algorithm, or even a $o(n \log\log n)$-time algorithm. Resolving this question seems to require a new idea. One possible approach is to invent a data structure for representing a polygonal curve that will allow fast computation of its intersections with an arbitrary horizontal line segment, or even with an arbitrary horizontal half line. Perhaps such a data structure can be built using information computed by the visibility algorithm called recursively on small pieces of the boundary, say of size $O(\log n)$. The result might be a visibility algorithm running in $O(n \log^* n)$ time. Another open problem is to determine how fast all the self-intersections of a polygonal curve can be computed: can bounds better than those for an arbitrary collection of line segments [4] be obtained?

Appendix. Finger search trees. A *finger search tree* is a type of balanced search tree in which access in the vicinity of certain preferred positions, indicated by *fingers*, is especially efficient. Finger search trees were introduced by Guibas, McCreight, Plass and Roberts [12] and further developed by many other researchers [2],[15],[18],[28],[29]. We shall discuss two kinds of finger search trees with slightly different properties, *heterogeneous trees* and *homogeneous trees*. We base our development on a particular kind of balanced tree, the *red-black tree* [20],[25], although other kinds of balanced trees, such as a,b-trees [16],[19], form a suitable basis as well. We are mainly interested in amortized, not worst-case, complexity bounds.

For our purposes a *binary search tree* is a full binary tree in which each external node contains a distinct item selected from a totally ordered universe, with the left-to-right order of external nodes consistent with the total order on the items. Each internal node contains a *key*, which is an item greater than or equal to all items in its left subtree and less than all items in its right subtree. We can use the keys to search for the largest item in the tree less than or equal to a given one, by starting from the root and going to the left child if the item in the current node is greater than or equal to the given one, going to the right child otherwise, and repeating until an external node is reached. The desired item is either the one in the external node reached or the one in the preceding external node, which can be found by backing up the search path to a right child, starting from its left sibling, and going through right children to an external node. The time to search for an item is proportional to the tree depth.

A *red-black tree* is a binary search tree in which each node has one of two colors, *red* or *black*. The node colors obey the following constraints (see Figure 19):

(i) all external nodes are black;

(ii) all paths from the root to an external node contain the same number of black nodes;

(iii) any red node, if it has a parent, has a black parent.

The depth of a red-black tree containing n items is $O(\log n)$. To insert a new item into a red-black tree, we search for the greatest item less than it in the tree. When the search reaches an external node, we replace this node by an internal node having two children, the old external node and a new external node containing the new

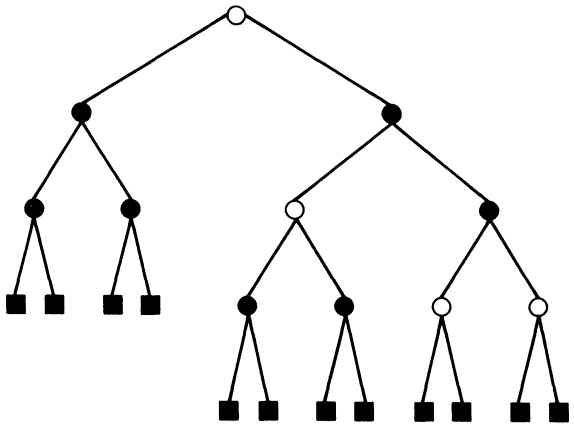

FIG. 19. *A red-black tree; solid nodes are black; hollow nodes are red. (Only the colors of the nodes are shown.)*

item. The new internal node contains as its key the smaller of the two items in its children. The new internal node is colored red. This may violate the red constraint (iii). To restore the red constraint, we proceed bottom-up along the search path, applying the recoloring transformation in Figure 20a until it no longer applies, followed by one application of Figure 20b, c, or d if necessary.

A deletion is similar. To delete an item, we find the external node containing it. We replace the parent of this node by the sibling of the node to be deleted. This may violate the black constraint (ii), producing a node that is *short*: all paths down from it to external nodes contain one fewer black node than paths down from its sibling. To restore the black constraint, we proceed bottom-up, applying the recoloring transformation of Figure 21a until it no longer applies, followed by one application of Figure 21b if necessary, and then possibly one application of Figure 21a, c, d, or e.

The worst-case insertion or deletion time in an n-item red-black tree is $O(\log n)$, but the amortized insertion/deletion time is only $O(1)$, not counting the time to search for the node at which the insertion or deletion takes place. (This is a restatement of a result of Huddleston and Mehlhorn [16] and Maier and Salveter [19] concerning a,b-trees.) To prove this, we define the *potential* of a red-black tree to be the number of black nodes with two black children plus twice the number of black nodes with two red children. We define the *actual time* of an insertion or deletion to be one plus the number of local transformations applied and the *amortized time* to be the actual time plus the net increase in potential caused by the operation. With these definitions, if we start with an empty tree, the total actual time for a sequence of insertions and deletions is at most the sum of the amortized times, since the initial potential is zero and the potential is always nonnegative. Furthermore the amortized time of an insertion or deletion is $O(1)$, since any of the transformations in Figures 20 and 21 increases the potential by $O(1)$ and the nonterminal transformations 20a and 21a both decrease the potential by at least one.

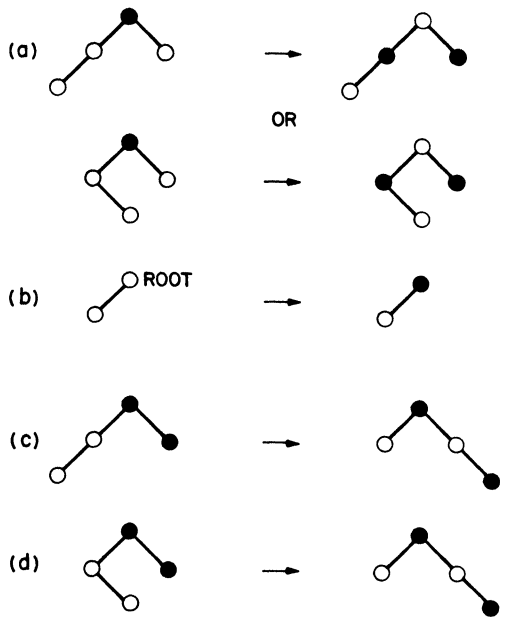

FIG. 20. *The rebalancing transformations in red-black tree insertion. Symmetric cases are omitted. All unknown children of red nodes are black. In cases* (c) *and* (d) *the bottommost black node shown can be external.*

In an ordinary binary search tree, each node points to its two children. We convert such a tree into a *heterogeneous finger search tree* by making each node along the left path[2] point to its parent instead of its left child, and each node along the right path point to its parent instead of its right child. Access to the tree is by two fingers pointing to the leftmost and rightmost external nodes. (See Figure 22.)

In an n-item heterogeneous search tree, we can search for an item d positions away from either end in $O(1+\log(\min\{d,n-d\}+1))$ time, by searching up along the left and right paths concurrently until we find a subtree or two subtrees guaranteed to contain the desired item, and then searching down in this subtree or subtrees. Furthermore we can insert or delete an item d positions from either end in $O(1+\log(\min\{d,n-d\}+1))$ amortized time. We can also search for an item based on its position or based on a secondary heap order. To accommodate search by position, we store in each internal node the number of external nodes reachable from it (by paths of pointers). To accommodate search based on a secondary heap order, we assume each item has an associated secondary value. We store in each internal node the minimum and maximum values of items reachable by paths of pointers. (See Figure 22.) By searching up along the left and right paths concurrently and then down into an appropriate subtree or subtrees, we can perform the following kinds of searches in $O(1+\log(\min\{d,n-d\}+1))$ time:

(i) Find the dth item in the tree;

[2]The *left path* in a binary tree is the path from the root through left children to an external node. The right path is defined symmetrically.

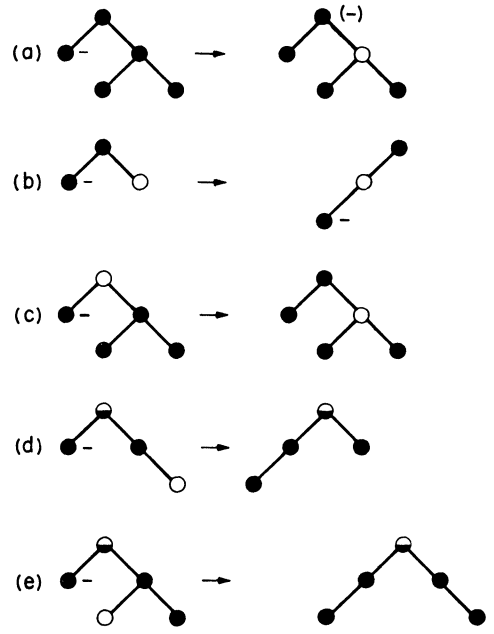

FIG. 21. *The rebalancing transformations in red-black tree deletion. The two ambiguous (half-solid) nodes in* (d) *have the same color, as do the two in* (e). *Minus signs denote short nodes. In* (a), *the top node after the transformation is short unless it is the root.*

(ii) Find the leftmost (or rightmost) item whose secondary value is at least (or at most) a given value, if the item found is the dth.

The auxiliary position and secondary value information must be updated when insertions and deletions are performed. This updating can be done bottom-up along the search path, i.e. bottom-up within the tree and top-down along the left or right path. The amortized time to insert or delete the dth item, including the search time, is $O(1+\log(\min\{d,n-d\}+1))$.

We now wish to extend our repertoire of update operations to include concatenation and splitting of trees. We shall discuss only the effect of these operations on the tree structure; it is easy to verify that the pointers, keys, and auxiliary position and secondary value information can be updated in the claimed time bounds. We define the *rank* of a node in a red-black tree to be the number of black internal nodes on any path from the node down to an external node; the rank of an external node is zero. We can compute the rank of a node in time proportional to the rank by walking down through the tree.

Concatenation is the simpler operation to describe. Suppose we wish to combine two trees T_1 and T_2 into a single tree; we assume that all items in T_1 are less than all items in T_2. Let x_1 with rank r_1 and x_2 with rank r_2 be the roots of T_1 and T_2, respectively. Assume $r_1 \leqslant r_2$. (The other case is symmetric.) To concatenate T_1

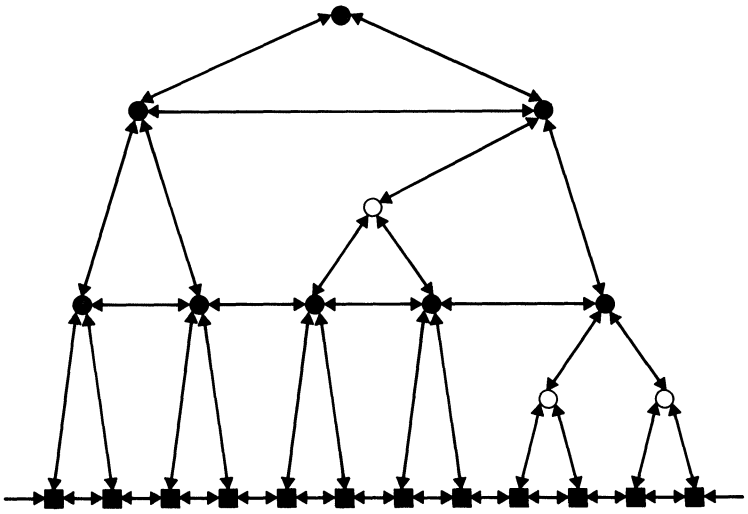

FIG. 22. *A heterogeneous red-black finger search tree. The colors of nodes are not shown. The items are the letters a through f. The numbers in external nodes are secondary values. The numbers in internal nodes are the minimum and maximum secondary values reachable from the nodes. The numbers outside the nodes are the number of external nodes reachable from them.*

and T_2, we walk up the left path of T_2 until we reach a node, say y, with rank equal to r_1. We replace y in T_2 by a new red node whose left child is x_1 and whose right child is y, correcting any violation of the red constraint as in the case of an insertion. The amortized time for the concatenation is $O(1+\min\{r_1,r_2\})$. If we change the definition of potential so that the potential of a tree is the rank of its root plus the number of black nodes with no black children, then the amortized time of a concatenation is $O(1)$; the amortized time for inserting or deleting the dth item out of n remains $O(1+\log(\min\{d,n-d\}+1))$.

Suppose we wish to split a tree T containing n items at the dth item, dividing it into a tree T_1 containing the first d items and a tree T_2 containing the last $n-d$ items. First we locate the dth item. Then we walk up along the search path to the left or right path, deleting every node along the search path except the external node containing the dth item. Assume we reach a node x on the left path. We concatenate the trees to the left of the search path (including the one consisting of the single node containing the dth item) in right-to-left order to form T_1. We concatenate the trees to the right of the search path whose roots are descendants of x to form a tree T_2'. If node x has no parent, then tree T_2' is the desired T_2. Otherwise, there remains another tree T_2'' containing the parent of x, say y. Tree T_2'' is missing a node, since node x was deleted. We replace node y by its right child, and repair the possible resulting shortness as in a deletion. Then we concatenate T_2' and T_2'' to form T_2. A careful analysis (see e.g. [20, pp. 214-216]) shows that the amortized time for splitting is $O(1+\log(\min\{d,n-d\}+1))$ for either the new or the old definition of potential.

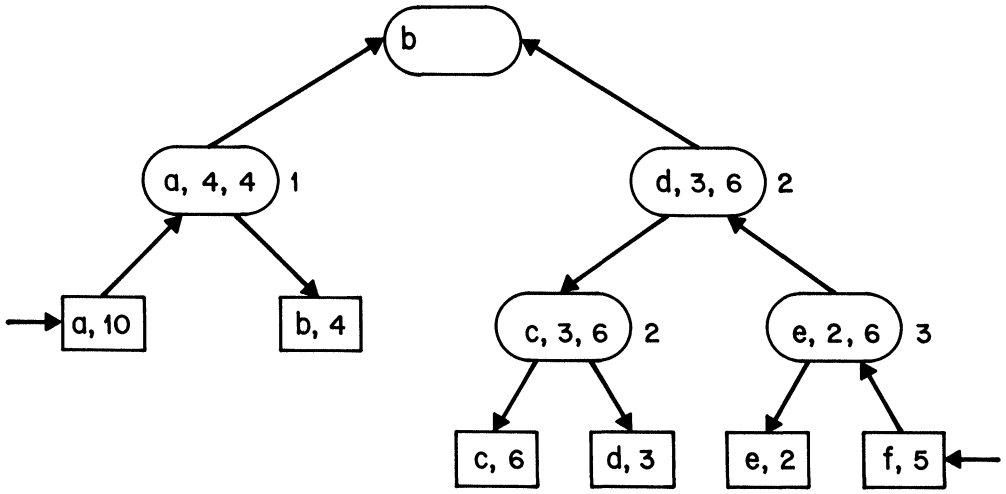

FIG. 23. *The pointers in a homogeneous red-black finger search tree.*

Heterogeneous finger search trees are used in the visibility algorithm to represent parts of region boundaries. The term "heterogeneous" refers to the fact that the pointer structure favors certain specific access positions, namely the two ends. In contrast, *homogeneous* finger search trees support fast access in the vicinity of *any* item. We make a red-black tree into a homogeneous finger search tree by adding additional pointers to it; namely, we make each node point to its two children and to its parent. Each black node also points to its *left* and *right neighbors*, the black nodes of the same rank that precede and follow the given node in symmetric order. (See Figure 23.) These extra pointers are called *level links*.

The level links support searching for a given item starting from an arbitrary item in the tree in $O(1+\log(\min\{d,n-d\}+1))$ time, where d is the number of items between the two given items. The search proceeds up from the starting item following parent pointers and level links until a subtree or two in which the desired item must be located are found; then the search proceeds downward in the standard way. Simultaneously with the search from the given starting item, searches are performed starting from the first and last items in the list; all three searches terminate when the desired item is first located (or discovered not to be in the tree).

Homogeneous finger search trees support fast searches only with respect to the total order on the items, not searches by position or searches based on secondary heap order. On the other hand, the extra time needed to update level links is only $O(1)$ per local transformation (of the kinds in Figures 19 and 20), and thus the amortized time bounds of insertion, deletion, concatenation, and splitting are the same in homogeneous trees as they are in heterogeneous trees. Furthermore, homogeneous trees support a more drastic splitting operation, called *three-way splitting*: given two items x and y in a tree T, remove from T the sublist of items from x to y (inclusive) to form two trees, one representing the removed sublist and the other representing the remaining

items. The amortized time for a three-way splitting operation is $O(1+\log(\min\{d,n-d\}+1))$, where d is the number of items in the removed sublist. See [14] for details on how a three-way split can be performed within this time bound.

Acknowledgments. We thank Brenda Baker for her many helpful comments on an earlier draft of this paper, and Bernard Chazelle for his thorough reading of this version of the paper.

REFERENCES

[1] B. K. BHATTACHARYA AND G. T. TOUSSAINT, *A linear algorithm for determining the translation separability of two simple polygons*, Technical report SOCS 861, School of Computer Science, McGill University, Montreal, Canada, 1986.

[2] M. R. BROWN AND R. E. TARJAN, *Design and analysis of a data structure for representing sorted lists*, this Journal, 9 (1980), pp. 594-614.

[3] B. CHAZELLE, *A theorem on polygon cutting with applications*, Proc. 23rd Annual Symp. on Found. of Comput. Sci., 1982, pp. 339-349.

[4] ———, *Intersecting is easier than sorting*, Proc. Sixteenth Annual ACM Symp. on Theory of Comput., 1984, pp. 125-134.

[5] B. CHAZELLE AND J. INCERPI, *Triangulation and shape complexity*, ACM Trans. on Graphics, 3 (1984), pp. 135-152.

[6] W.-P. CHIN AND S. NTAFOS, *Optimum watchman routes*, Proc. Second Annual Symp. on Computational Geometry, 1986, pp. 24-33.

[7] D. P. DOBKIN, D. L. SOUVAINE AND C. J. VAN WYK, *Decomposition and intersection of simple splinegons*, Algorithmica, to appear.

[8] H. EDELSBRUNNER, L. J. GUIBAS AND J. STOLFI, *Optimal point location in a monotone subdivision*, this Journal, 15 (1986), pp. 317-340.

[9] A. FOURNIER AND D. Y. MONTUNO, *Triangulating simple polygons and equivalent problems*, ACM Trans. on Graphics, 3 (1984), pp. 153-174.

[10] M. R. GAREY, D. S. JOHNSON, F. P. PREPARATA AND R. E. TARJAN, *Triangulating a simple polygon*, Inform. Process. Lett., 7 (1978), pp. 175-180.

[11] L. GUIBAS, J. HERSHBERGER, D. LEVEN, M. SHARIR AND R. E. TARJAN, *Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons*, Algorithmica, to appear.

[12] L. J. GUIBAS, E. M. MCCREIGHT, M. F. PLASS AND J. R. ROBERTS, *A new representation for linear lists*, Proc. Ninth Annual ACM Symp. on Theory of Comput., 1977, pp. 49-60.

[13] S. HERTEL AND K. MEHLHORN, *Fast triangulation of a simple polygon*, Proc. Conf. Found. of Comput. Theory, New York, Springer-Verlag, Berlin, 1983, pp. 207-218.

[14] K. HOFFMAN, K. MEHLHORN, P. ROSENSTIEHL AND R. TARJAN, *Sorting Jordan sequences in linear time using level-linked search trees*, Inform. and Control, 68 (1986), pp. 170-184.

[15] S. HUDDLESTON, *An efficient scheme for fast local updates in linear lists*, Dept. of Information and Computer Science, University of California, Irvine, CA, 1981.

[16] S. HUDDLESTON AND K. MEHLHORN, *A new data structure for representing sorted lists*, Acta Inform., 17 (1982), pp. 157-184.

[17] J. M. KEIL, *Decomposing a polygon into simpler components*, this Journal, 14 (1985), pp. 799-817.

[18] S. R. KOSARAJU, *Localized search in sorted lists*, Proc. Thirteenth Annual ACM Symp. on Theory of Comput., 1981, pp. 62-69.

[19] D. MAIER AND C. SALVETER, *Hysterical B-trees*, Inform. Process. Lett., 12 (1981), pp. 199-202.

[20] K. MEHLHORN, *Data Structures and Efficient Algorithms, Volume 1: Sorting and Searching*, Springer-Verlag, Berlin, 1984.

[21] J. R. MUNKRES, *Topology: A First Course*, Prentice-Hall, Englewood Cliffs, NJ, 1975.

[22] A. A. SCHÄFFER AND C. J. VAN WYK, *Convex hulls of piecewise smooth Jordan curves*, J. Algorithms, 8 (1987), pp. 66-94.

[23] D. D. SLEATOR AND R. E. TARJAN, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., 32 (1985), pp. 652-686.

[24] D. L. SOUVAINE, *Computational Geometry in a Curved World*, Ph.D. dissertation, Department of Computer Science, Princeton University, 1986.

[25] R. E. TARJAN, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[26] ———, *Amortized computational complexity*, SIAM J. Algebraic and Discrete Methods, 6 (1985), pp. 306-318.

[27] R. E. TARJAN AND C. J. VAN WYK, *A linear-time algorithm for triangulating simple polygons*, Proc. Eighteenth Annual ACM Symp. on Theory of Comput., 1986, pp. 380-388.

[28] A. K. TSAKALIDIS, *AVL-trees for localized search*, Automata, Languages, and Programming, 11th Colloquium, J. Paredaens, ed., Lecture Notes in Computer Science 172, Springer-Verlag, Berlin, 1984, pp. 473-485.

[29] ———, *An optimal implementation for localized search*, A84/06 Fachbereich Angewandte Mathematik und Informatik, Universität des Saarlandes, Saarbrücken, West Germany, 1984.

[30] C. J. VAN WYK, *Clipping to the boundary of a circular-arc polygon*, Computer Vision, Graphics, and Image Processing, 25 (1984), pp. 383-392.

[31] T. C. WOO AND S. Y. SHIN, *A linear time algorithm for triangulating a point-visible polygon*, ACM Trans. on Graphics, 4 (1985), pp. 60-69.