Information Processing Letters 35 (1990) 85–92
North-Holland

29 June 1990

SIMPLIFIED LINEAR-TIME JORDAN SORTING AND POLYGON CLIPPING

Khun Yee FUNG and Tina M. NICHOLL *
Department of Computer Science, The University of Western Ontario, London, Ont., Canada N6A 5B7

Robert E. TARJAN **
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA; and Department of Computer Science, Princeton University, Princeton, NJ 08544, USA

Christopher J. VAN WYK
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA

Communicated by D. Gries
Received 18 August 1989
Revised 16 January 1990

Given the intersection points of a Jordan curve with the x-axis in the order in which they occur along the curve, the *Jordan sorting problem* is to sort them into the order in which they occur along the x-axis. This problem arises in clipping a simple polygon against a rectangle (a "window") and in efficient algorithms for triangulating a simple polygon. Hoffman, Mehlhorn, Rosenstiehl, and Tarjan proposed an algorithm that solves the Jordan sorting problem in time that is linear in the number of intersection points, but their algorithm requires the use of a sophisticated data structure, the level-linked search tree. We propose a variant of the algorithm of Hoffman et al. that retains the linear-time bound but simplifies both the primary data structure and the operations it must perform.

Keywords: Analysis of algorithms, computational geometry, data structures

1. Polygon clipping via Jordan sorting

Let P be a simple n-vertex planar polygon and let W be a rectangular window in the plane. The problem of *clipping P against W* is to compute the polygon or set of polygons S that bound the intersection of the interior of P with the interior of W. The polygon clipping problem is fundamental in computational geometry and in computer graphics; it arises, for example, in window management and in hidden surface removal [13]. In the latter application, the problem is actually more general, in that both P and W can be polygons with holes, but our results on the basic problem extend to the more general problem. In our discussion, we assume that every intersection point of P and W is a crossing point, i.e., a point where P crosses from the inside to the outside of W. Our approach extends easily to handle portions of P that are tangent to W at points or in line segments.

The process of polygon clipping divides naturally into two steps:

(1) Find the points of intersection of P and W;

(2) find the polygonal curves into which the points of intersection divide P and W, and group these curves to form the set S of output polygons.

Step (1) involves primarily geometric computation. It can be carried out in $O(n)$ time by pro-

* Research partially supported by NSERC Canada.
** Research partially supported by the National Science Foundation, Grant DCR-8605962, and the Office of Naval Research, Contract N00014-87-K-0467.

ceeding from vertex to vertex along P and determining, for each line segment of P, whether and where it crosses W. An asymptotic running-time bound of $O(n)$ follows immediately from the fact that there are $O(n)$ intersection points. Minimizing the constant factor in the running time involves interesting issues; see, e.g., [8]. This method of performing step (1) produces the intersection points in the order in which they occur along P, and as a side effect determines the polygonal curves into which the intersection points divide P.

Step (2), on the other hand, involves primarily topological computation. The hard part of the computation is to sort the intersection points into the order in which they occur along W. Once this is done, producing the set of output polygons is straightforward [13].

Much work on polygon clipping has concentrated on step (1) at the expense of step (2). Sutherland and Hodgman [10] mentioned the sorting problem in an appendix; the same appendix proposes a grouping method that is based on the computation of shortest paths and that can produce degenerate nonsimple polygons as output. Liang and Barsky [7] ignored the need for step (2) entirely. Weiler and Atherton [13] described a correct grouping method but did not explicitly discuss the sorting problem.

If a general sorting algorithm is used to sort the intersection points, the time for step (2), and hence for the entire polygon clipping problem, is $O(n \log n)$. From an algorithmic point of view, the interesting aspect of polygon clipping is that a general sorting algorithm is not required. Indeed, step (2) can be performed in $O(n)$ time, and hence so can polygon clipping.

Let us abstract the sorting problem slightly to make it easier to study. Consider a Jordan curve C in the plane that crosses the x-axis everywhere it touches it. (A *closed* Jordan curve is a homeomorphic image of a circle. An *open* Jordan curve is a homeomorphic image of a line segment. Most topologists use the term "Jordan curve" to mean a closed Jordan curve. We extend the term, however, to mean either an open or a closed Jordan curve; the key property of Jordan curves is that they do not have self-crossings. To be concrete, one can think of C as a simple polygonal curve.) Let $z_1, z_2, \ldots, z_n$ be the sequence of intersection points of C with the x-axis in the order in which they occur along C. The *Jordan sorting problem* is to sort the given sequence $z_1, z_2, \ldots, z_n$ by x-coordinate.

An algorithm for Jordan sorting can be applied directly to the sorting part of step (2) of polygon clipping. Jordan sorting is also needed in efficient algorithms for triangulating a simple polygon [4,12]. Hoffman, Mehlhorn, Rosenstiehl and Tarjan [6] devised an $O(n)$ time Jordan sorting algorithm. The heart of their method is a sophisticated data structure, the *level-linked search tree*.

Our purpose in this paper is to simplify the algorithm of Hoffman et al. while preserving the $O(n)$ time bound. First, we simplify the kinds of operations needed on the key data structure. We show that Jordan sorting can be reduced to performing an intermixed sequence of two kinds of operations on a collection of sorted lists:

(i) *insert* an item at the front or the back of a given list;

(ii) *find* an item in a given list and *split* the list just before or just after this item.

The algorithm of Hoffman et al. requires a third list operation in which a middle section of a list is split out and the two end sections are concatenated. Since our algorithm does not require this third list operation, we obtain a second simplification: the use of *heterogeneous finger trees* to represent the list, in place of level-linked trees (which implement *homogeneous finger trees*).

The remainder of the paper consists of three sections. In Section 2, we reduce the Jordan sorting problem to the list manipulation problem discussed above. In Section 3 we show that the list manipulation problem can be solved in $O(n)$ time using heterogeneous finger trees or any equivalently efficient data structure. Section 4 contains some concluding remarks.

2. Jordan sorting via list manipulation

Let us review the approach of Hoffman et al. [6] to the Jordan sorting problem (see also [12]).

For notational concision, we interpret the expression "$z_i < z_j$" to mean that the x-coordinate of z_i is less than the x-coordinate of z_j. We write $\langle z_{i-1}, z_i\rangle$ to denote the part of C that goes from z_{i-1} to z_i without crossing the x-axis; thus, if $i \equiv j \pmod 2$, $\langle z_{i-1}, z_i\rangle$ and $\langle z_{j-1}, z_j\rangle$ lie on the same side of the x-axis. We also write $\langle z_i, \ldots, z_j\rangle$ to denote $\bigcup_{i \leqslant k < j} \langle z_k, z_{k+1}\rangle$.

The sequence $z_1, z_2, \ldots, z_n$ gives rise to two forests, as follows. Assume that $\langle z_1, z_2\rangle$ lies above the x-axis. (If not, reflect C about the x-axis.) For $1 < i \leqslant n$, let $l_i = \min\{z_{i-1}, z_i\}$ and $r_i = \max\{z_{i-1}, z_i\}$. We say that a pair $\{z_{i-1}, z_i\}$ *encloses a point* z if $l_i < z < r_i$. We say that a pair $\{z_{i-1}, z_i\}$ *encloses a pair* $\{z_{j-1}, z_j\}$ if $i \equiv j \pmod 2$ and $\{z_{i-1}, z_i\}$ encloses both z_{j-1} and z_j; thus, $\langle z_{j-1}, z_j\rangle$ lies between $\langle z_{i-1}, z_i\rangle$ and the x-axis. For any two pairs $\{z_{i-1}, z_i\}$ and $\{z_{j-1}, z_j\}$ such that $i \equiv j \pmod 2$, the simplicity of C implies that each pair encloses an even number of points (zero or two) of the other. The Hasse diagram of the "encloses" relation on the set of pairs $\{\{z_{2i-1}, z_{2i}\} \mid 1 \leqslant i \leqslant \lfloor \frac{1}{2}\rfloor\}$ is a forest, called the *upper forest*. Similarly, the Hasse diagram of the "encloses" relation on the set of pairs $\{\{z_{2i}, z_{2i+1}\} \mid 1 \leqslant i \leqslant \lfloor \frac{1}{2}(n-1)\rfloor\}$ is called the *lower forest*. In both of these forests, we order each set of siblings by placing $\{z_{i-1}, z_i\}$ before $\{z_{j-1}, z_j\}$ if $r_i < l_j$; this makes each forest into an ordered forest. By adding a dummy pair $\{-\infty, \infty\}$ to each forest, we create two ordered trees, called the *upper tree* and the *lower tree* (see Fig. 1).

A *family* is a set of pairs consisting of a pair and all of its children (in the appropriate tree). Suppose that the set of finite pairs in the family is $\{z_{i_1-1}, z_{i_1}\}, \{z_{i_2-1}, z_{i_2}\}, \ldots, \{z_{i_k-1}, z_{i_k}\}$. If the family does not include the pair $\{-\infty, \infty\}$, then it corresponds to a simple closed curve formed from $\langle z_{i_1-1}, z_{i_1}\rangle, \langle z_{i_2-1}, z_{i_2}\rangle, \ldots, \langle z_{i_k-1}, z_{i_k}\rangle$ along with appropriate parts of the x-axis interconnecting them; the region inside this curve is the *family region*. If the family includes the pair $\{-\infty, \infty\}$, then it corresponds to a simple open curve formed from $\langle z_{i_1-1}, z_{i_1}\rangle, \langle z_{i_2-1}, z_{i_2}\rangle, \ldots, \langle z_{i_k-1}, z_{i_k}\rangle$ along with parts of the x-axis interconnecting them and two rays that go from the leftmost curve to $-\infty$ and from the rightmost curve to ∞; the connected region bounded by this curve that contains no other parts of the x-axis is the family region. The family region for a family in the upper tree lies above the x-axis, while the family region for a family in the lower tree lies below the x-axis (see Fig. 2).

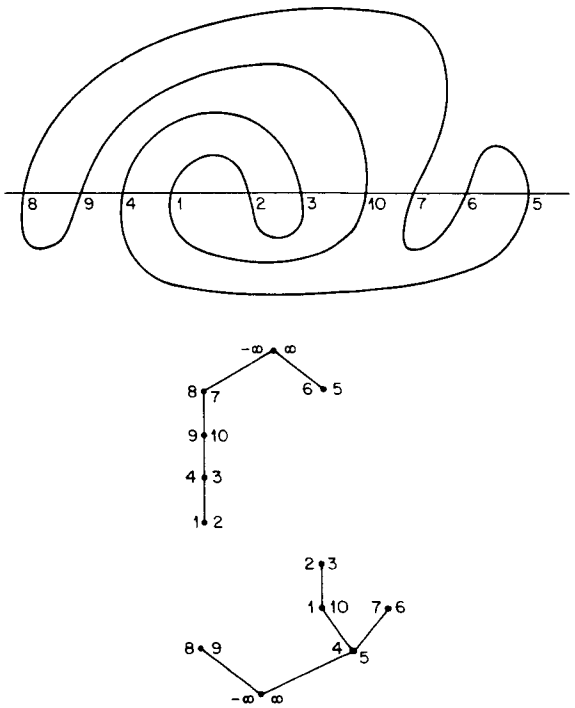

Fig. 1. A Jordan curve and its upper and lower family trees. In this and the following three figures, the intersection z_i of the curve with the x-axis is labelled i.

The Jordan sorting algorithm proceeds incrementally, processing the points $z_1, z_2, \ldots, z_n$ one at a time and building the upper tree, the lower tree, and a list of the points in sorted order. Processing the point z_i involves inserting z_i into the appropriate position in the sorted list and, if $i > 1$, adding $\{z_{i-1}, z_i\}$ to the appropriate tree (the upper tree if i is even, the lower tree if i is odd).

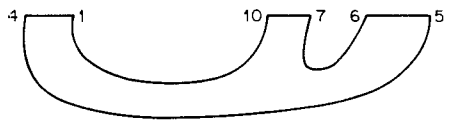

Fig. 2. The family region corresponding to $\{z_4, z_5\}$ and its children in the lower family tree of the curve in Fig. 1.

Before examining the details of this algorithm, we make a new observation about the family regions that simplifies the required processing.

Locality lemma (see Fig. 3). *Suppose that* $z_1, z_2, \ldots, z_i$ *have been processed so far, i.e., the upper and lower trees have been constructed representing the part of curve C from* z_1 *to* z_i. *Let F be the family so far constructed containing the pair* $\{z_{i-1}, z_i\}$ *as a child, and let R be the corresponding family region; observe that no part of* $\langle z_1, \ldots, z_i \rangle$ *can lie in the interior of R. Let* $z_j \neq z_{i-1}$ *be the point among the points in pairs of F such that* z_i *and* z_j *are adjacent, i.e., no point in a pair of F lies between* z_i *and* z_j. *Consider the point* z_k *where C first enters R after passing through* z_i *(if C ever enters R). Then either*

(a) z_k *lies between* z_i *and* z_j; *or*

(b) *the parent pair in F is* $\{-\infty, \infty\}$ *and* z_i *and* z_k *bracket all finite points in pairs of F other than* z_i.

Proof. The proof is by contradiction. Suppose the lemma were not true. Then there would be two finite points in pairs of F, say z_p and z_q, such that z_p but not z_q lay between z_i and z_k (see Fig. 4). Since R is a path-connected region, there would be a simple curve S lying entirely inside R connecting z_i and z_k. Curve S, together with $\langle z_i, \ldots, z_k \rangle$, would form a simple closed curve O with z_p on its inside and z_q on its outside. The part of $\langle z_1, \ldots, z_k \rangle$ that connects z_p and z_q would have to intersect O. Since R is a family region, this intersection could not occur on S, so it would have to occur on $\langle z_i, \ldots, z_k \rangle$; but this would violate the simplicity of C. This contradiction establishes the lemma. □

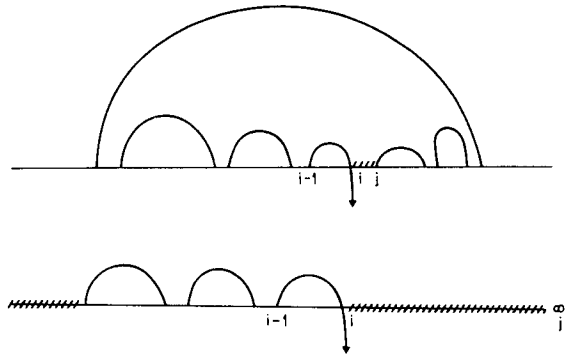

Fig. 3. The locality lemma determines where C can reenter region R after it leaves at z_i. At the top, region R is finite, and C must reenter it between z_i and z_j. At the bottom, region R is infinite, and C may reenter it to the right of z_i or to the left of all finite points.

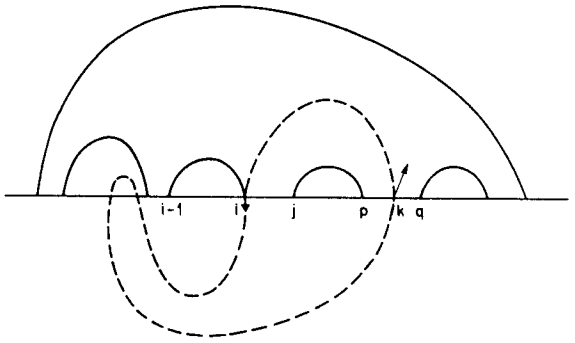

Fig. 4. In this illustration of the proof of the locality lemma, the boundary of region O is dashed. The portion of the boundary outside R is $\langle z_i, z_{i+1} \rangle, \ldots, \langle z_{k-1}, z_k \rangle$.

The locality lemma allows us to represent each family by one or two lists, called *sibling lists*. Consider a family F whose most recently added pair is $\{z_{i-1}, z_i\}$. If $\{z_{i-1}, z_i\}$ is the parent pair of F, then F is represented by one sibling list containing all the pairs in F except $\{z_{i-1}, z_i\}$. If $\{z_{i-1}, z_i\}$ is not the parent pair of F, then F is represented by two sibling lists, one containing all pairs in F both of whose points are less than or equal to z_i and the other containing all pairs in F both of whose points are greater than or equal to z_i (one of these lists may be empty). Sibling lists never include parent pairs; each finite pair occurs in two families (once as a parent and once as a child) but in only one sibling list.

Now we are ready to describe the details of the Jordan sorting algorithm.

Jordan sorting algorithm. Initialization consists of creating a sorted list containing $-\infty$, z_1, z_2, z_3, and ∞ in sorted order, and creating two singleton sibling lists containing $\{z_1, z_2\}$ and $\{z_2, z_3\}$.

Repeat the following steps for i from 4 to n:

Step 1: Let v be the point in the sorted list preceding z_{i-1}. If i is odd and $v = z_1$, replace v by its predecessor in the sorted list. (Point z_1 is in no pair in the lower family.) Let $\{z_{j-1}, z_j\}$ be the pair containing v such that $i \equiv j \pmod 2$. [Thus, $\langle z_{j-1}, z_j\rangle$ and $\langle z_{i-1}, z_i\rangle$ lie on the same side of the x-axis.] (If $v = -\infty$, let $\{z_{j-1}, z_j\}$ be $\{-\infty, \infty\}$.)

Step 2: Let w be the point in the sorted list following z_{i-1}. If i is odd and $w = z_1$, replace w by its successor in the sorted list. Let $\{z_{k-1}, z_k\}$ be the pair containing w such that $i \equiv k \pmod 2$. [Thus, $\langle z_{k-1}, z_k\rangle$ and $\langle z_{i-1}, z_i\rangle$ lie on the same side of the x-axis.] (If $w = \infty$, let $\{z_{k-1}, z_k\}$ be $\{-\infty, \infty\}$.)

Step 3: Assume that $z_{i-1} < z_i$; the other case is symmetric.

(a) [Insert $\{z_{i-1}, z_i\}$ into its proper place with respect to its siblings to the left.] If $\{z_{j-1}, z_j\}$ encloses z_{i-1}, create a new singleton sibling list containing $\{z_{i-1}, z_i\}$; otherwise, insert $\{z_{i-1}, z_i\}$ after $\{z_{j-1}, z_j\}$ in the sibling list containing $\{z_{j-1}, z_j\}$. (The locality lemma implies that $\{z_{j-1}, z_j\}$ is last on this list.)

(b) [Split off any children of $\{z_{i-1}, z_i\}$.] If $\{z_{k-1}, z_k\}$ does not enclose z_{i-1}, split the sibling list containing $\{z_{k-1}, z_k\}$ into two lists, one containing all pairs whose points are less than z_i, the other containing all pairs whose points are greater than z_i. (The locality lemma implies that $\{z_{k-1}, z_k\}$ is first on its sibling list before the list is split.) One or the other of the split lists may be empty.

(c) [Insert z_i into its proper place in the sorted list.] If $\{z_{i-1}, z_i\}$ has no children, insert z_i immediately after z_{i-1} in the sorted list. Otherwise, let $\{z_{m-1}, z_m\}$ be the rightmost child of $\{z_{i-1}, z_i\}$; insert z_i immediately after z_m in the sorted list.

(see Fig. 5).

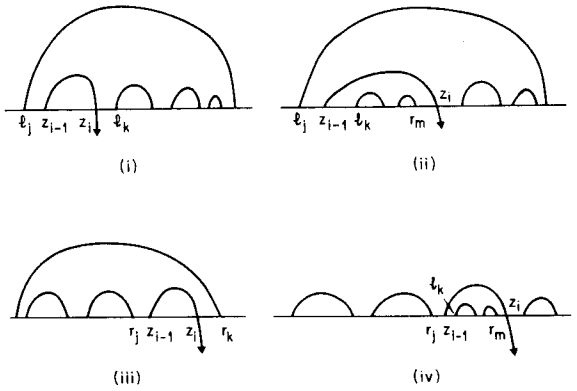

Fig. 5. Illustration of Step 3 of the algorithm. In (i) and (ii), substep (a) creates the singleton sublist $\{z_{i-1}, z_i\}$; in (iii) and (iv), substep (a) adds $\{z_{i-1}, z_i\}$ after $\{z_{j-1}, z_j\}$. In (i), (iii), and (iv), substep (b) splits off zero or more children of $\{z_{i-1}, z_i\}$.

The correctness of this algorithm follows from the locality lemma and other properties of Jordan curves. Let us make a few comments about the data structures needed to implement the algorithm. We assume the input is given in the form of a doubly linked list, with *next*(z) and *prev*(z) being the successor and predecessor of intersection point z along C, respectively. The output is also represented by a doubly linked list, with *after*(z) and *before*(z) being the successor and predecessor of intersection point z along the x-axis. In addition, a bit *even*(z_i) indicates for each intersection point z_i whether i is even or not. With this representation, it is not necessary to construct the pairs explicitly. A pair $\{z_{i-1}, z_i\}$ can be represented just by z_{i-1}, since $z_i = next(z_{i-1})$ is computable in constant time. Each of the sibling lists is then just a list of intersection points, rather than a list of pairs of points.

The total space requirements of the algorithm are four pointers and one bit per point, plus whatever space is necessary to represent the sibling lists. The running time of the algorithm is $O(n)$ plus the time needed to perform $O(n)$ insertion and split operations on sibling lists. In the next section we discuss a data structure for representing the sibling lists that requires $O(n)$ time and space for the list operations.

3. Insertion and splitting operations on sorted lists

We have reduced the Jordan sorting problem to a pure problem in data structures, which we can formulate as follows. Maintain a collection of sorted lists whose items are selected from a totally ordered universe U, subject to the following three kinds of operations:

(i) *make-list*(x): Create a new singleton list containing x.

(ii) *insert*(x, y): If $x < y$, insert x at the front of the list containing y; in this case, y must initially be at the front of its list. If $x > y$, insert x at the back of the list containing y; in this case, y must initially be at the back of its list.

(iii) *split*(x, L): Split list L into two lists, one containing all items less than or equal to x and the other containing all items greater than x.

A sequence of m intermixed operations is to be performed on an initially empty collection of lists. We seek a list representation such that the total time for m operations is $O(m)$.

A suitable list representation is the *heterogeneous finger tree*. This is a balanced search tree in which the pointers along the two ribs of the tree (the paths from the root to the first and last nodes in symmetric order) go up instead of down. We omit a detailed description of the data structure, since it can be found elsewhere [12]. Heterogeneous finger trees support the desired list operations in the following amortized time bounds: $O(1)$ for *make-list* and *insert*, and $O(\log(\min\{l-k, k\}+2))$ for *split*, where l is the size of the input list and k and $l-k$ are the sizes of the two output lists. By "amortized" time bounds we mean that for any sequence of operations, the sum of the amortized time bounds is an upper bound on the sum of the actual times. See Tarjan's survey paper [11] for a thorough discussion of amortization.

The amortized time bound for *split* is too large to conclude immediately that the total time for a sequence of m operations starting with no lists is $O(m)$. By using an extra amortization argument, however, we can reduce the amortized time of *split* to $O(1)$. The reasoning is the same as that used by Goldberg and Tarjan [5] to charge splitting time to concatenations. (In our case this time is charged to insertions.)

We use the idea of a *potential function* [11]. We define the *potential* of a list of size l to be $c(l - \log l)$, where c is the constant in the amortized time bound for splitting, and the base of the logarithm is two. We define the *total potential* of a collection of lists to be the sum of their potentials, and the *nominal time* of a list operation to be its amortized time bound plus the net increase in potential it causes. For any sequence of list operations, the sum of the nominal times equals the sum of the amortized time bounds plus the final total potential minus the initial total potential. The initial total potential is zero (there are no lists initially) and the final total potential is nonnegative. Thus the total time required to perform any sequence of m list operations is at most the sum of the nominal time bounds.

The nominal time to initialize a list is $O(1)$. The nominal time for an insertion is also $O(1)$, because its amortized time is $O(1)$ and the increase in potential when the list grows from length l to $l+1$ is

$$\begin{aligned} & c(l+1-\log(l+1)-l+\log l) \\ &= c\left(1+\log\frac{l}{l+1}\right) \\ &\leqslant c = O(1). \end{aligned}$$

Finally, the nominal time for a split is $O(1)$; when sublists of sizes $k \geqslant 1$ and $l-k \geqslant 1$ are formed by a split, the amortized time is $c(\log(\min\{k, l-k\}+2))$, and the potential increase is

$$\begin{aligned} & c(k-\log k+(l-k)-\log(l-k)-l+\log l) \\ &= c(-\log k-\log(l-k)+\log l) \\ &= c\left(-\log\min\{k, l-k\}+\log\frac{l}{\max\{k, l-k\}}\right) \\ &\leqslant c(-\log\min\{k, l-k\}+1), \end{aligned}$$

since $l \leqslant 2\max\{k, l-k\}$. Therefore the sum of the amortized time and the potential increase is at most $3c = O(1)$. Since the nominal time for each

kind of list operation is O(1), the total time for m list operations is $O(m)$, as desired.

4. Remarks

The Jordan sorting algorithm can be extended easily to two related problems: the recognition problem, i.e., testing whether a given sequence can arise as the sequence of intersection points of a Jordan curve with the x-axis (see [6]), and the problem of Jordan sorting with error-correction required by the efficient triangulation algorithm of Tarjan and Van Wyk [12]. Neither extension affects the $O(n)$ time bound.

There are two obvious ways to apply the Jordan sorting algorithm to clip a polygon against a convex window. Sutherland and Hodgman [10] suggest that the polygon be clipped sequentially against each line that contains a side of the window; the set of polygons resulting from clipping against each side becomes the input for clipping against the next side. It may, however, be more efficient to clip the polygon against the entire window at once. To do this, we need only be able to compare two intersection points with respect to their order around the window boundary. If we choose a point on the window boundary as the origin, a point on the interior as the center, and a direction in which the order increases (say clockwise), then we can perform each comparison in O(1) time by computing the angles of the rays joining the center to the points to be compared.

Throughout this paper, and particularly in the preceding paragraph, we have ignored the time required to find the points of intersection between the n-sided polygon P and the window W. If W has a fixed number of sides, then all of the intersection points can be found in $O(n)$ time, by a generalization of the method described in Section 1. If W is convex and has k sides, then we can traverse P to find the intersection points, using a binary search algorithm on the sides of W to discover whether each vertex lies inside or outside the polygon and where each edge of p intersects W; the total time for this method is $O(n \log k)$. If W is not convex, then the number of intersections I between P and W is $O(nk)$; the algorithm of Chazelle and Edelsbrunner [3] can be used to find all I intersections in $O(I + (n + k) \log(n + k))$ time, and in fact produces enough information to perform step (2) directly.

To summarize, when the clipping window W has a fixed number of sides, step (1) can be performed in $O(n)$ time, so the complexity of step (2) determines the complexity of any polygon clipping algorithm. When W has a variable number of sides or is not convex, however, the dominance of step (2) in the complexity of a polygon clipping algorithm is no longer so obvious.

As noted in the paper by Hoffman et al. [6], the use of the splay tree [9], a self-adjusting form of search tree, gives a very simple algorithm for Jordan sorting that may run in linear time. This method may well be preferable in practice to the algorithm we have presented, even though the linear-time bound for the method based on splay trees remains only a conjecture.

For completeness, we remark that another proposed simplified Jordan sorting algorithm [1] is incorrect [2].

References

[1] F. Aurenhammer, Jordan sorting via convex hulls of certain non-simple polygons, in: *Proc. 3rd Annual Symposium on Computer Geometry* (1987) 21–29.

[2] F. Aurenhammer, On-line sorting of twisted sequences in linear time, *BIT* **28** (1988) 194–204.

[3] B. Chazelle and H. Edelsbrunner, An optimal algorithm for intersecting line segments in the plane, in: *Proc. 29th Annual Symposium on Foundations of Computer Science* (1988) 590–600.

[4] K.L. Clarkson, R.E. Tarjan and C.J. Van Wyk, A fast Las Vegas algorithm for triangulating a simple polygon, *Discrete and Comput. Geom.*, to appear.

[5] A.V. Goldberg and R.E. Tarjan, Finding minimum-cost circulations by successive approximation, *Math. Oper. Res.*, to appear.

[6] K. Hoffman, K. Mehlhorn, P. Rosenstiehl and R.E. Tarjan, Sorting Jordan sequences using level-linked search trees, *Inform. and Control* **68** (1986) 170–184.

[7] Y.D. Liang and B.A. Barsky, An analysis and algorithm for polygon clipping, *Comm. ACM* **26** (1983) 868–877.

[8] T.M. Nicholl, D.T. Lee and R.A. Nicholl, An efficient new algorithm for 2-D line clipping: its development and analysis, in: *Proc. SIGGRAPH '87*, Comput. Graphics **21** (4) (1987) 253–262.

[9] D.D. Sleator and R.E. Tarjan, Self-adjusting binary search trees, *J. ACM* **32** (1985) 652–686.
[10] I.E. Sutherland and G.W. Hodgman, Reentrant polygon clipping, *Comm. ACM* **17** (1974) 32–42.
[11] R.E. Tarjan, Amortized computational complexity, *SIAM J. Algebraic Discrete Methods* **6** (1985) 306–318.
[12] R.E. Tarjan and C.J. Van Wyk, An $O(n \log \log n)$-time algorithm for triangulating a simple polygon, *SIAM J. Comput.* **17** (1988) 143–178; Erratum: *SIAM J. Comput.* **17** (1988) 1061.
[13] K. Weiler and P. Atherton, Hidden surface removal using polygon area sorting, in: *Proc. SIGGRAPH '77*, Comput. Graphics **11** (2) (1977) 214–222.