

CSL *COORDINATED SCIENCE LABORATORY*

APPLIED COMPUTATION THEORY GROUP

**PLANE-SWEEP ALGORITHMS FOR
INTERSECTING GEOMETRIC FIGURES**

J. NIEVERGELT
F.P. PREPARATA

REPORT R-863

UILU-ENG 78-2256

UNIVERSITY OF ILLINOIS – URBANA, ILLINOIS

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PLANE-SWEEP ALGORITHMS FOR INTERSECTING GEOMETRIC FIGURES		5. TYPE OF REPORT & PERIOD COVERED
7. AUTHOR(s) J. Nievergelt and F. P. Preparata		6. PERFORMING ORG. REPORT NUMBER R-863; ACT-18 UILU-ENG 78-2256
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		8. CONTRACT OR GRANT NUMBER(s) NSF MCS 78-13642; N00014-79-C-0424
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program; National Science Foundation		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE October 1979
		13. NUMBER OF PAGES 22
		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Computational geometry, concrete complexity, combinatorial algorithms, intersection problems, maps, polygons		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) We consider various types of geometric figures in the plane defined by points connected by straight line segments, such as polygons (not necessarily simple) and maps (embedded planar graphs); and the problem of computing the intersection of such figures by means of a "greedy" type of algorithm that sweeps the plane unidirectionally. Let n be the total number of points of all the figures involved, and s the total number of intersections of line segments. We show that in the general case (no convexity) a previously known algorithm can be extended to compute in time		

PLANE-SWEEP ALGORITHMS FOR
INTERSECTING GEOMETRIC FIGURES

by

J. Nievergelt and F. P. Preparata

This work was supported in part by the National Science Foundation under Grant MCS 78-13642 and Joint Services Electronics Program under Contract N00014-79-C-0424.

Reproduction in whole or in part is permitted for any purpose of the United States Government.

This report is issued simultaneously by the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, and by the Institut für Informatik, Eidgenössische Technische Hochschule, Zürich.

Approved for public release. Distribution unlimited.

PLANE-SWEEP ALGORITHMS FOR INTERSECTING GEOMETRIC FIGURES

J. Nievergelt and F. P. Preparata

Abstract

We consider various types of geometric figures in the plane defined by points connected by straight line segments, such as polygons (not necessarily simple) and maps (embedded planar graphs); and the problem of computing the intersection of such figures by means of a "greedy" type of algorithm that sweeps the plane unidirectionally. Let n be the total number of points of all the figures involved, and s the total number of intersections of line segments. We show that in the general case (no convexity) a previously known algorithm can be extended to compute in time $O((n+s)\log n)$ and space $O(n+s)$ all the connected regions into which the plane is divided by intersecting the figures. When the regions of each figure are convex, the same can be achieved in time $O(n\log n+s)$ and space $O(n)$.

Keywords and phrases:

Computational geometry, concrete complexity, combinatorial algorithms, intersection problems, maps, polygons.

CR categories:

5.3, 5.32.

Addresses of authors:

J. Nievergelt, Informatik, ETH, CH-8092 Zurich, Switzerland.

F. P. Preparata, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois 61801, USA.

This work was supported in part by the National Science Foundation under Grant MCS 78-13642 and Joint Services Electronics Program under Contract N00014-79-C-0424.

1. INTRODUCTION

One type of algorithm that emerged from recent research in computational plane geometry promises to be efficient for all problems to which it applies. It sweeps the plane "from left to right", advancing a "front" or "cross-section" from point to next point. All processing is done at this moving front, without any backtracking, with a horizon or look-ahead of only one point. Algorithms of this type are often called "greedy," in contrast to exhaustive search algorithms that create tentative partial results perhaps to be discarded later. An important issue in computational complexity is to understand which problems can be solved by greedy algorithms. We contribute to this issue, and present two efficient algorithms for problems that have applications in geographic data processing.

Shamos and Hoey [6,7] presented an algorithm that, by sweeping the plane unidirectionally, determines in time $O(n \log n)$ whether or not n line segments are free of intersections. Bentley and Ottmann [2] have extended this algorithm to report all s intersections of n line segments within time $O((s+n) \log n)$. We elaborate on this type of algorithm in two directions. First, we show that, within the same asymptotic effort, it can compute all connected regions into which the plane is divided by the line segments, where each region is presented by a cyclic list of all its boundary vertices and/or segments. Second, we show that assumptions of convexity allow one to improve these asymptotic bounds. Specifically, we present an algorithm that computes all s intersections of two convex maps (embedded planar graphs with convex regions) with a total of n points in space $O(n)$ and time $O(n \log n + s)$, which is optimal in s .

2. REGIONS FORMED BY A POLYGON

Later sections of the paper will refer to several versions of the problem of computing regions of the plane formed by embedded graphs. Here we present in detail the version which is simplest for expository purposes: let the given graph be a polygon, i.e., a closed chain of n line segments (or equivalently, a cyclic list of n points in the plane). The reader who has understood how the algorithm works in this case will have no difficulty in following the brief description of how it applies to the intersection of more general plane-embedded graphs.

2.1 Statement of the Problem and Terminology

Given a sequence of n points $V_i = (x_i, y_i)$, $i = 1, 2, \dots, n$, in the plane, a polygon with vertices V_i is the sequence of line segments $\overline{V_1 V_2}$, $\overline{V_2 V_3}$, \dots , $\overline{V_n V_1}$. These n line segments in general define s intersection points $W_j = (x_j, y_j)$, $j = 1, 2, \dots, s$. When $s = 0$ the polygon is called simple and divides the plane into two regions, an internal bounded region R_1 and an external unbounded region R_0 . In general $s = O(n^2)$, and the polygon divides the plane into $r+1 \geq 2$ disjoint regions, namely the external unbounded region R_0 and r simply connected internal regions R_1, \dots, R_r (when the polygon is non-degenerate, $r = s+1$). Each region is itself a polygon that has as its vertices some subset of $\{V_1, \dots, V_n, W_1, \dots, W_s\}$. The desired result is a list of all regions, where each region is given by a cyclic list of its vertices, starting with the right-most vertex; the external region in clockwise order, the internal regions in counterclockwise order. Figure 1 illustrated these concepts.

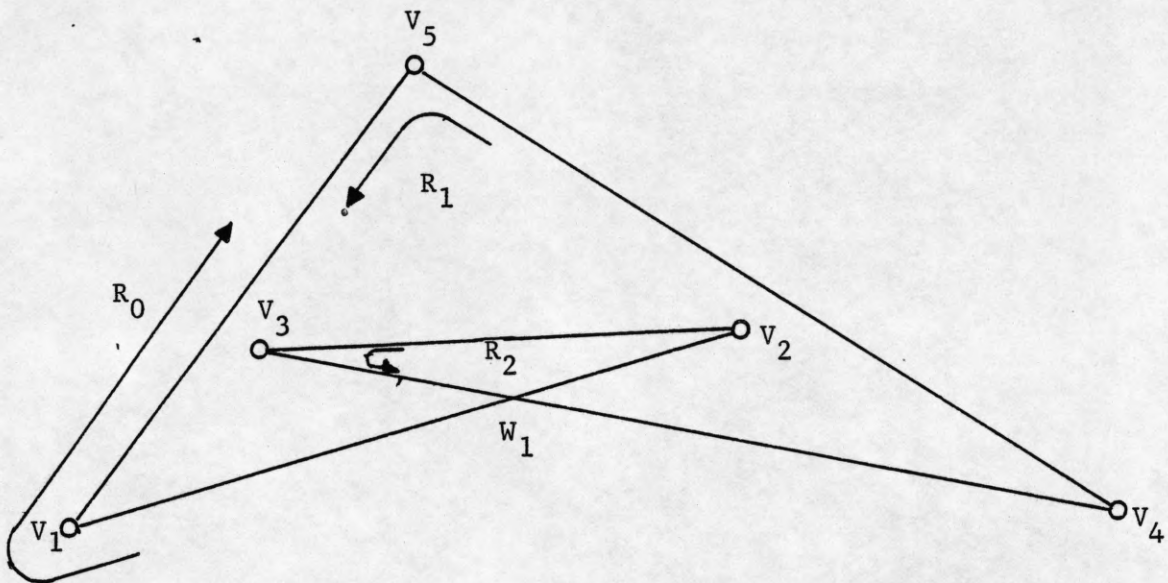


Fig. 1. A polygon of 5 line segments and the regions it defines:

$$R_0: V_4 W_1 V_1 V_5$$





$$R_1: V_4 V_5 V_1 W_1 V_3 V_2 W_1$$

$$R_2: V_2 V_3 W_1$$

As Fig. 1 suggests, we make certain assumptions of non-degeneracy, namely: all points (originally given or intersection) are distinct; a point lies on only those line segments on which it must lie, and no others (e.g., V_2 does not lie on $\overline{V_4 V_5}$). Section 4 discusses the modifications required to handle degenerate pictures.

The notions above are sufficient to describe the problem and its solution. In order to describe the algorithm which computes the solution, we introduce auxiliary concepts that reflect the dynamic aspects - a unidirectional sweeping of the plane. Call the originally given points V_i and the intersection points W_j simply points, P_1, P_2, \dots, P_{n+s} , sorted in order of increasing x-coordinate. (We assume for ease of exposition that no two points have equal x-coordinates; if P_i and P_{i+1} have equal x-coordinate,

a lexicographic ordering on the pair (x,y) suffices for the following discussion to apply.) Under our assumptions of non-degeneracy and distinct x -coordinates, each point can be classified uniquely into one of the following 4 categories:

- a start point 
- an end point 
- a bend 
- an intersection point 

points originally given as vertices of the polygon.

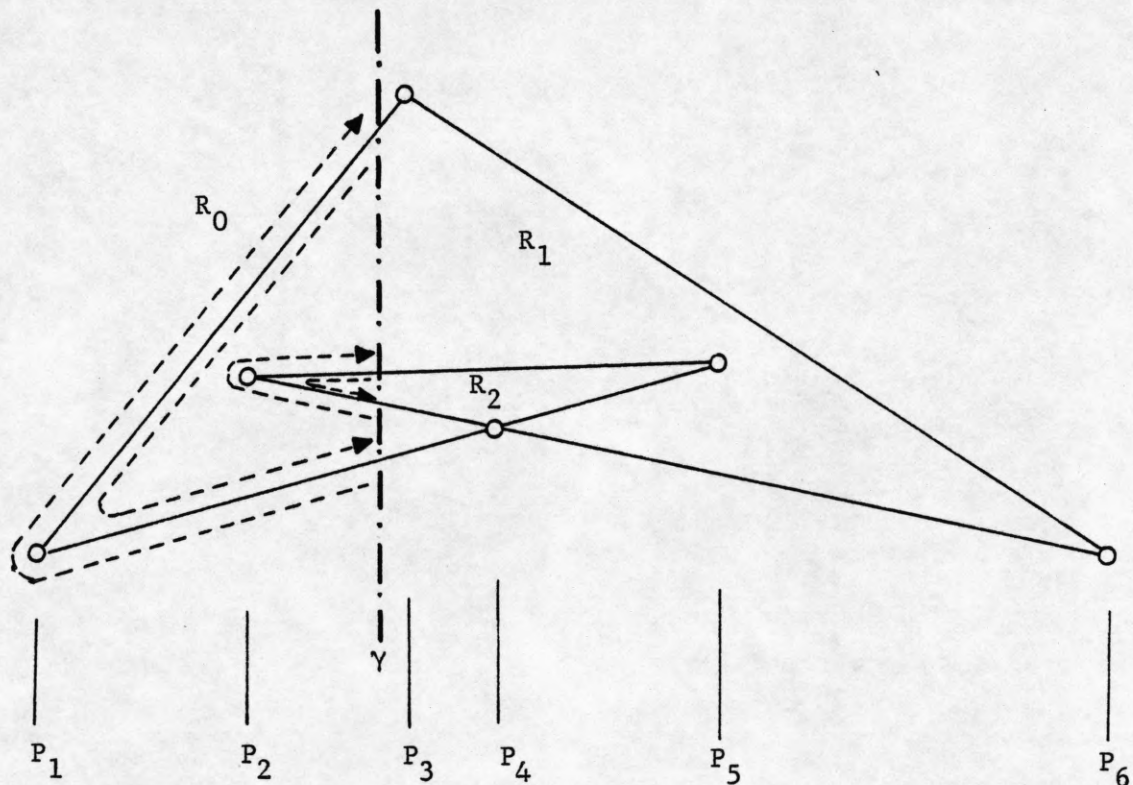


Fig. 2. Figure 1 revisited. Points have been ordered according to increasing x -coordinate. Cross section γ contains one interval for each of the regions R_0 and R_2 , and two intervals for two branches of R_1 that have split. Oriented "rubber bands" trailing the cross section γ are shown as broken lines.

A cross section is a vertical line in the plane along with all the information about which line segments, regions and components it cuts, and in what order. The line segments cut by a cross section partition it into intervals. We are mainly concerned with cross sections that do not pass through any points; the set of all (topologically equivalent) cross sections that lie between two adjacent points is called a slice. A cross section that passes through (exactly) one point is called a transition.

As the current cross section sweeps the figure, it drags along "rubber bands" that hug the periphery of regions, as shown in Fig. 2. Much of the specification of the region-finding algorithm is concerned with properly maintaining these rubber bands across transitions.

2.2 Data Structures Maintained by the Algorithm

The region-finding algorithm to be presented in section 2.3 operates upon three data structures. These are crucial for understanding the algorithm as well as for its efficiency. We present these data structures at a fairly abstract level: by postulating what operations must be performed and how much time is available for them (asymptotically in terms of the problem parameters n and s). We refer to standard textbooks [1,4] for concrete implementations that realize the postulated time bounds.

The x-structure X

At any moment X contains those points that have been discovered so far and are yet to be processed, sorted according to increasing x -coordinate. Points are assigned a type according to the classification of section 2.1. The x -structure is a priority queue that must support the following

operations within time bound $O(\log k)$ when it contains k entries:

- MIN: find and remove the entry with minimal x-coordinate
- INSERT: insert a new entry with given x-coordinate.

Heaps or balanced trees are suitable for implementing priority queues.

Initial content: the n originally given points, sorted, with their classification.

Final content: empty.

At each transition, the point which defines this transition is removed, and at most two intersection points are inserted into X . During execution a total of $n+s$ points move through the x -structure, hence the maximal number of entries at any given time is $< n+s$. Since $s = O(n^2)$, any operation on the x -structure can be done in time $O(\log(n+s)) = O(\log n)$.

The y-structure Y

Y contains all the information about a cross section which is representative of its entire slice. It has an entry for each segment intersected by the slice, including two sentinels corresponding to $y = +\infty$ and $y = -\infty$, and thus it never has more than $n+2$ entries. Y is a dictionary (see [1]) that must support the following operation within time bound $O(\log k)$ when it contains k entries:

- FIND: given a point (x,y) , find the line segment in the cross section at x whose y -value does not exceed y .

In addition, in constant time, the y -structure supports the following operations:

- DELETE(s): given a pointer to an entry s, delete that entry.
- INSERT(s;t): given a pointer to an entry s, insert a new entry t following that entry.
- PREDECESSOR(s): given a pointer to an entry s, obtain the pointer to the immediately preceding entry.
- SUCCESSOR(s): given a pointer to an entry s, obtain the pointer to the immediately following entry.

Various types of balanced trees, with additional "predecessor" and "successor" pointers can be used to implement such dictionary. Initial and final content of the y-structure: the pair $(-\infty, +\infty)$. Figure 3 shows the y-structure of the slice between P_4 and P_5 in Fig. 2.

Pointer into periphery of region	Boundary of interval	Name of region of which this interval is a part
	$+\infty$	
	$\overline{P_3 P_6}$	R_0
	$\overline{P_2 P_5}$	R_1
	$\overline{P_1 P_5}$	R_2
	$\overline{P_2 P_6}$	$R_4 (!)$
	$-\infty$	R_0

Fig. 3. The y-structure of the slice between points P_4 and P_5 in Fig. 2. The field " $\overline{P_3 P_6}$ " contains the definition of the line segment connecting the two points P_3 and P_6 . Given an x-value, this entry allows evaluation of the corresponding y-value in time $O(1)$. Notice the entry $R_4 (!)$ under "Name of region." It refers to a tentative region which was created at transition P_4 . In a left-to-right scan of the plane it will only become known at transition P_5 that region R_4 is to be identified with region R_1 . The field "Pointer into periphery" will be explained when we discuss the r-structure.

The r-structure R

The r-structure is the key to our algorithm. It integrates information about the regions and their peripheries as it is accumulated during the unidirectional sweep. It is initialized to be empty and terminates empty. For any given cross section it contains information about exactly those regions that are cut by this cross section. Specifically, with each segment in the cross section it associates pointers to the two cyclic lists of vertices on the boundaries of the regions which are respectively above and below that segment; equivalently, with each interval determined by two adjacent segments it associates the cyclic list of the boundary vertices of that part of the region which lies to the left of the cross section. Figure 4 shows the r-structure relative to a cross section in the slice between P_4 and P_5 of Fig. 2.

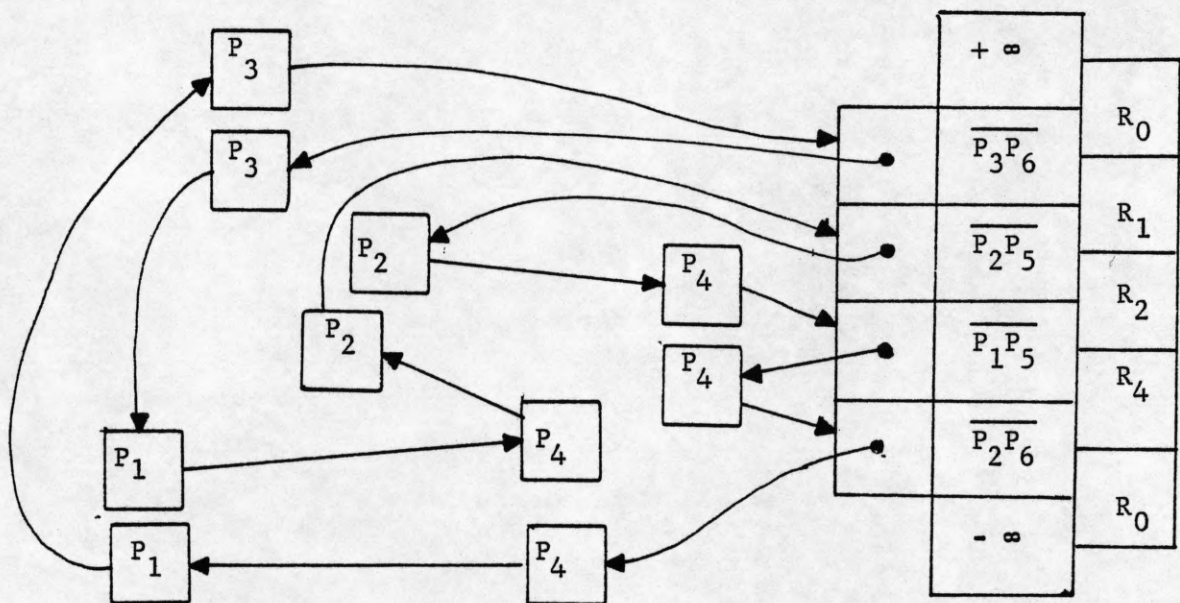


Fig. 4. The r-structure of a cross-section in slice P_4P_5 of Fig. 2, shown attached to the y-structure.

The description above is a static picture of R. Section 2.3 presents the dynamic picture, how R is updated at each transition, and how the region boundaries can be printed to produce the output.

2.3 The Region-Finding Algorithm

The algorithm that sweeps the plane and outputs the region boundaries has the following simple overall structure (here X, Y, and R are the three data structures previously described):

Procedure SWEEP

begin X \leftarrow n given points (sorted by x-coordinate)

Y \leftarrow $(-\infty, +\infty)$, name of region \leftarrow R₀

R \leftarrow \emptyset

while X \neq \emptyset do

begin P \leftarrow MIN(X)

 TRANSITION(P)

end

end

Procedure TRANSITION is the advancing mechanism of SWEEP, and denotes all the work involved in processing one point (P) and moving the "front" from the slice to the left of this point to the slice immediately to the right of it; in this process it updates the corresponding data structures and builds up the result in an output structure. TRANSITION will be invoked exactly (n+s) times and, since it will be shown that one invocation uses O(logn) time, an O((n+s)logn) time bound on the performance of SWEEP will result. Figure 5 illustrates the situation when transition is invoked: for any given segment s, A(s) and B(s) are respectively the cyclic lists of vertices of the regions bordering s above and below, with the shown orientations.

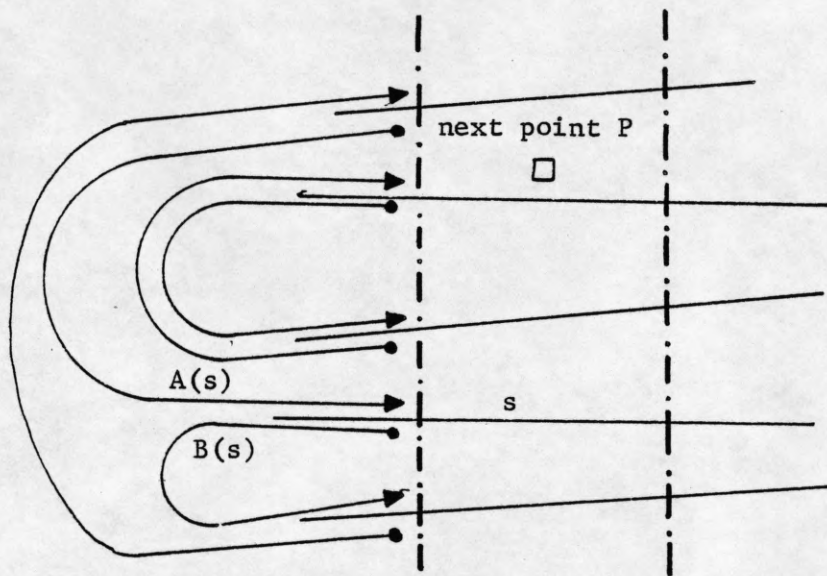






Fig. 5. The situation faced by TRANSITION. The "next point" P is of one of 4 types:

bend , start , end , intersection  .
 A(s) and B(s) are the cyclic lists of vertices of the regions bordering s above and below, respectively. Associated with s there are pointers to the head of A(s) and the tail of B(s).

Lists are thought of as being ordered from left to right; thus, for a point P and a list L, "P*L" denotes that P has been added to L as its new head, whereas "L*P" denotes that P is the new tail.

Similarly, for two lists L_1 and L_2 , $L_1 * L_2$ denotes their concatenation. We shall now give a detailed description of TRANSITION. The function $\text{INTERSECT}(s_1, s_2)$ checks in time $O(1)$ whether two segments s_1 and s_2 intersect, and if so, inserts the intersection point into X in time $O(\log |X|)$.

```

procedure TRANSITION (P)
1. begin s ← FIND(y[P])
2.   case P is "bend": (*see figure 6a*)
3.     begin t ← segment beginning at P
4.           h ← SUCC(s)
5.           l ← PRED(s)
6.           if (l ≠ -∞) then INTERSECT(l,t)
7.           if (h ≠ +∞) then INTERSECT(t,h)
8.           A(s) ← A(s)*P
9.           B(s) ← P*B(s)
10.          Replace s with t
11.        end
12.     case P is "end": (*see figure 6b*)
13.       begin h ← SUCC(s)
14.             t ← PRED(s)
15.             l ← PRED(t)
16.             if (l ≠ -∞) and (h ≠ +∞) then INTERSECT(l,h)
17.             Link A(t)*P*B(s) and output it
18.             Link A(s)*P*B(t)
19.             DELETE(s)
20.             DELETE(t)
21.           end
22.       case P is "start": (*see figure 6c*)
23.         begin t ← SUCC(s)
24.               (l,h) ← segments starting at p
25.               if (s ≠ -∞) then INTERSECT(s,l)
26.               if (t ≠ +∞) then INTERSECT(h,t)
27.               INSERT(s;l)
28.               INSERT(l;h)
29.               Link B(l)*P*A(h)
30.               Link B(h)*P*A(l) (*B(l),A(l),B(h),A(h) consist of P alone*)
31.             end
32.         case P is "intersection": (*see figure 6d*)
33.           begin h ← SUCC(s)
34.                 t ← PRED(s)
35.                 l ← PRED(t)
36.                 if (l ≠ -∞) then INTERSECT(l,s)
37.                 if (h ≠ +∞) then INTERSECT(t,h)
38.                 Link A(t)*P*B(s) and output it
39.                 A(s) ← A(s)*B
40.                 B(t) ← P*B(t)
41.                 interchange s and t
42.                 Link B(t)*P*A(s) (*A(s) and B(t) consist of P alone*)
43.               end
44.         end
45.       end
46.     end
47.   end

```

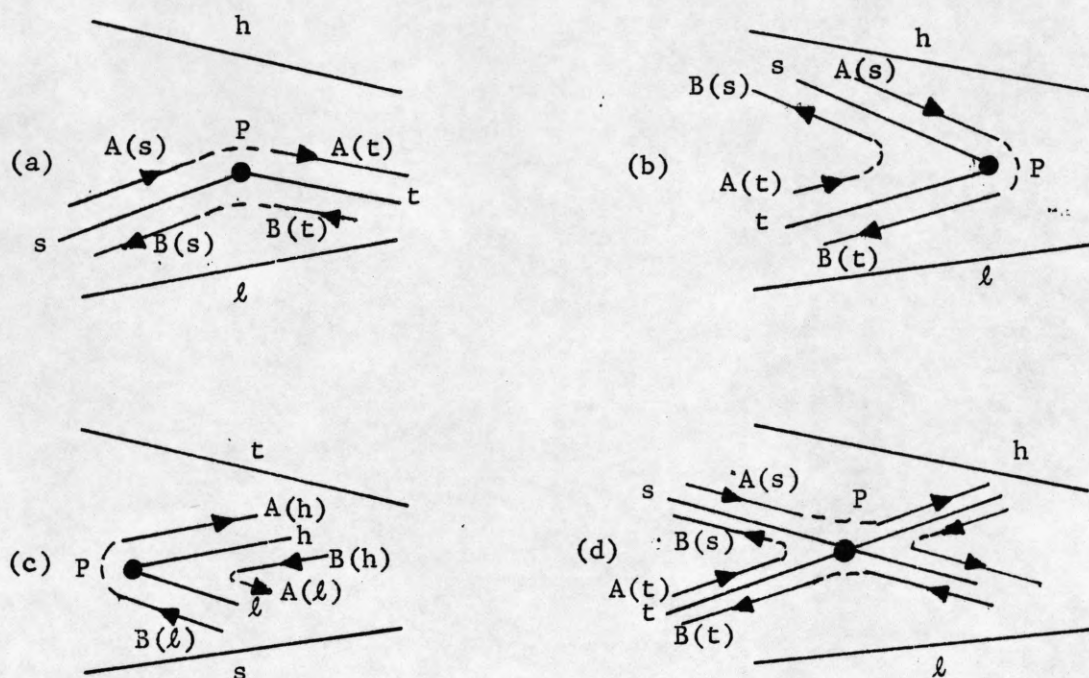


Fig. 6. Illustration of the four cases treated by procedure TRANSITION.

As a general comment to the outlined algorithm, all four cases have an analogous format. After segment s has been located (Step 1) in Y in time $O(\log|Y|)$, the other segments which are candidates for intersections are found in time $O(1)$ (e.g., Steps 3, 4, 5); next, intersections are sought and, if necessary, X is updated in time $O(\log|X|)$ (e.g., Steps 6, 7); finally, R (e.g., Steps 8, 9) and Y (e.g., Step 10) are updated, both in time $O(1)$.

We have seen in section 2.2 that $|X| = O(n)$ and $|Y| = O(s) = O(n^2)$, and thus all the operations in TRANSITION require time $O(\log n)$. Since the algorithm that sweeps the plane makes $n+s$ transitions, it requires time $O((n+s)\log n)$, thus confirming a prior claim.

3. INTERSECTION OF CONVEX MAPS

A map is a planar graph G embedded in the plane: each vertex of G is represented as a point and each edge as a straight line segment in such a way that edges intersect only at common vertices. A map subdivides the plane into r simply connected internal regions R_1, \dots, R_r and one external unbounded region R_0 . A map is convex if each internal region is convex and the complement of the external region is convex.

Given two convex maps G_1 and G_2 with n_1 and n_2 vertices, respectively, and s intersections of edges of G_1 with edges of G_2 ; we will show that the set of these s intersections can be computed in time $O(n \log n + s)$ and space $O(n)$, where $n = n_1 + n_2$. A straightforward application of the plane-sweep algorithm described in [2], which does not take advantage of convexity, would yield this result in time $O((n+s) \log n)$ and space $O(n+s)$. The problem of computing intersections of rectangles with parallel edges has been studied by Bentley and Wood [3] and McCreight [5]. Their results are not directly comparable to those of this section, since rectangles may intersect even if their edges don't.

Let G_1 and G_2 be two convex maps. For $i = 1, 2$, let e_i be an edge of G_i , and assume that e_1 and e_2 intersect in a single point u (degenerate cases can be handled with no difficulty). Define r_i as the region of G_i such that $r_1 \cap r_2$ lies entirely to the left of u (Fig. 7). We now define a plane domain U as follows. Let $e'_1 \neq e_1$ be an edge - if it exists - on the boundary of r_1 which intersects e_2 , and let e'_2 be analogously defined. If e'_i exists, define U_i as the convex hull of the extremes of e_i and e'_i , otherwise U_i is the unbounded half plane-strip orthogonal to e_i on the side of r_i ; then let $U \triangleq U_1 \cap U_2$ (Fig. 7).

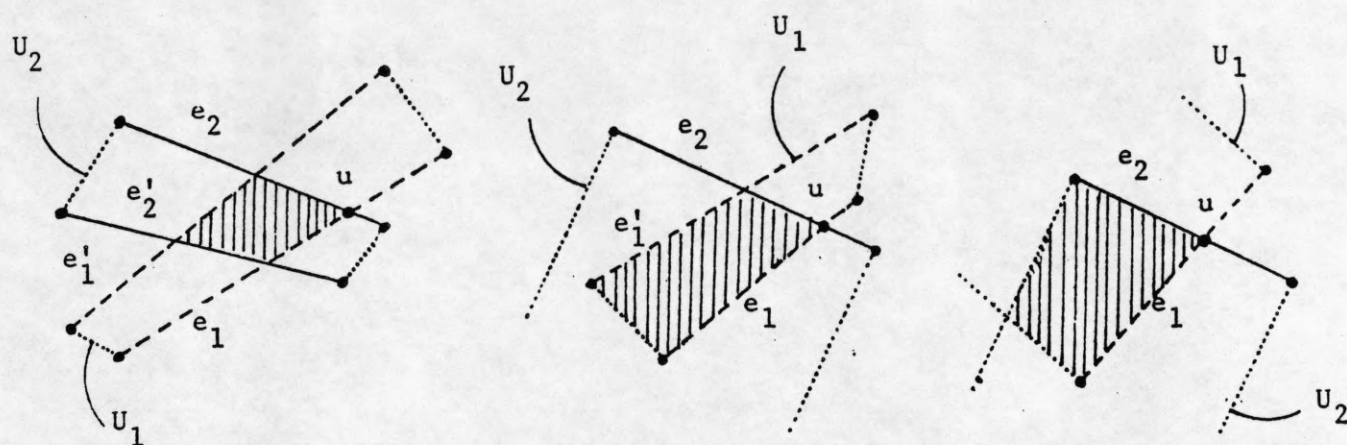


Fig. 7. The domain U is shown cross-hatched.

We claim that U contains in its interior no edge, nor portion of edge, either of G_1 or of G_2 . It suffices to show that U_2 contains in its interior no edge, nor portion of edge, of G_1 . This is obvious when U_1 is unbounded, because in this case U_1 is contained in the unbounded exterior region of G_1 ; when U_1 is bounded, then, by convexity, $U_1 \subseteq r_1$, and obviously the claim holds.

Consider now a cross-section γ at the abscissa of a vertex v of, say, G_1 (Fig. 8a) and suppose that edges e_1 and e_2 , of G_1 and G_2 respectively, are adjacent in γ and intersect at point u . By the preceding argument, the wedge comprised between the vertical through v , e_1 , and e_2 does not contain edges, nor portion of edges, of G_1 and G_2 ; hence we advance the cross section to include point u to its left (see Fig. 8b). This amounts to exchanging the order of e_1 and e_2 in γ , so that we may proceed with the

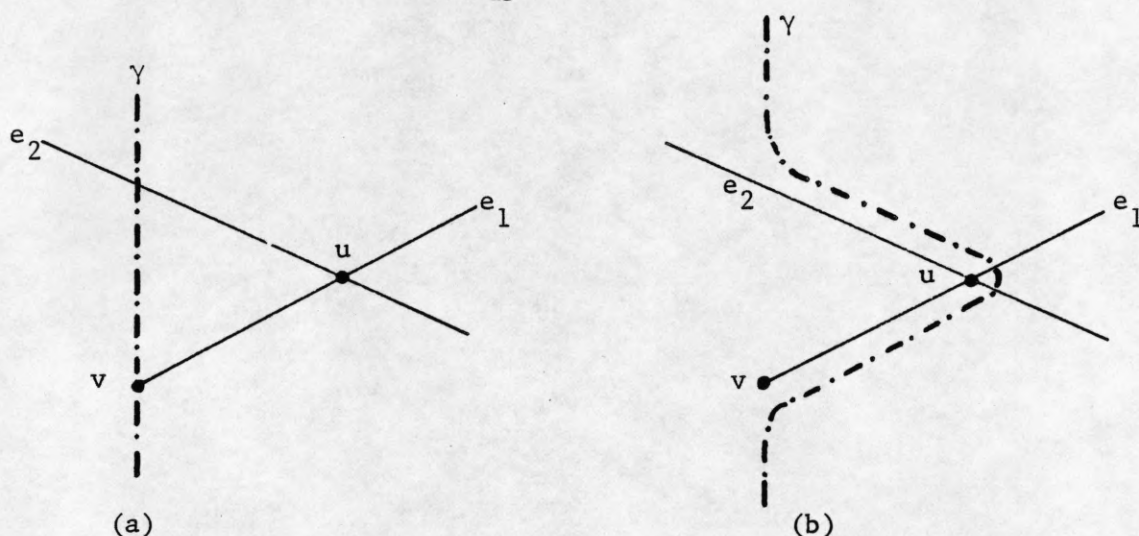


Fig. 8. The current cross section advances by one point.

verification of whether e_1 and e_2 have further intersections with edges of G_2 and G_1 , respectively. Specifically, for each intersection found two new adjacencies arise which are potentially intersection-producing; each such adjacency is then placed into a queue for later processing. The plane-sweeping algorithm will scan from left to right the vertices of both maps and perform the corresponding deletions and intersection of edges (Steps 5-7 and 9-13, respectively, in the following procedure SWEEP); for any new vertex, at most two adjacencies are created which may potentially yield intersections: specifically, if the edges issuing to the right of a vertex are ordered counterclockwise, these adjacencies may involve the first and the last elements of this sequence of edges. After this initialization of the queue (Steps 12 and 15 of SWEEP), for each intersection found the cross section is dynamically updated, i.e., an exchange of order of two segments takes places (Step 23) and two new adjacencies are examined for possible addition to the queue (Steps 20-22). At any point in the execution of the algorithm the cross section corresponds to a curve

in the plane (not at straight line, necessarily!) which bounds the discovered intersections to their right (Fig. 9).

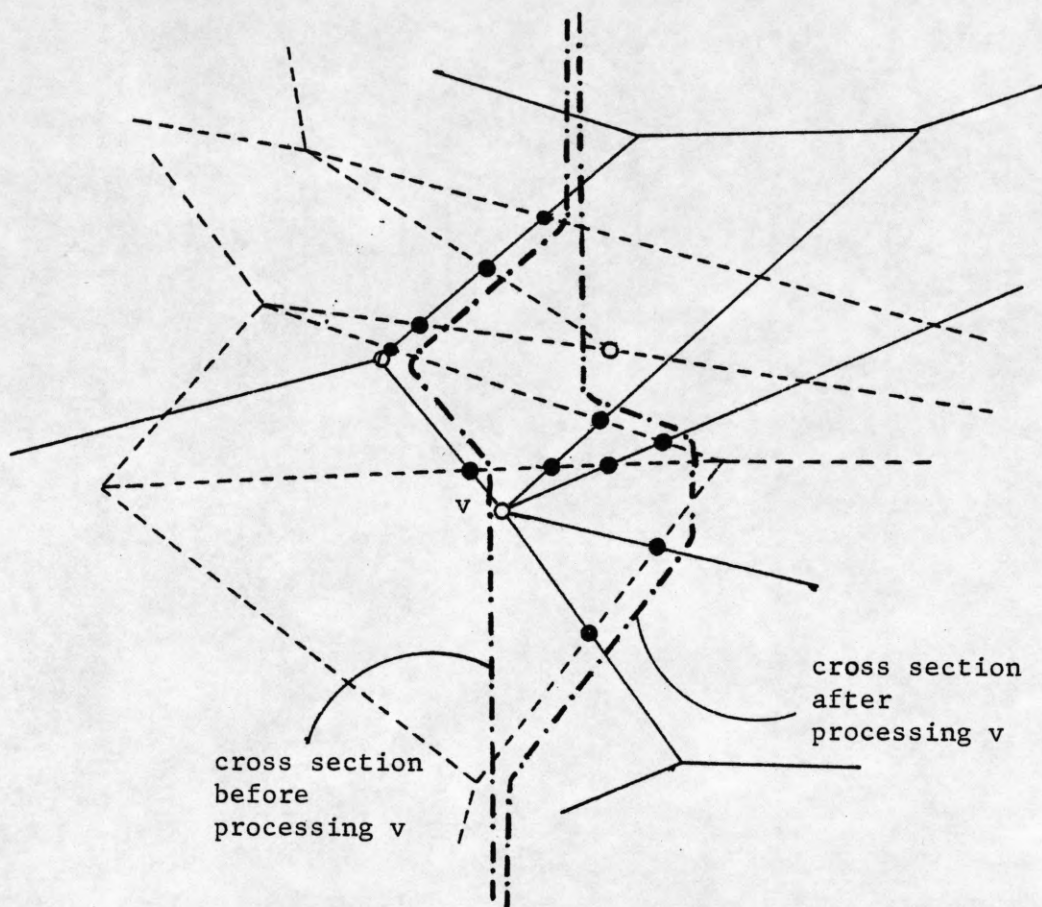


Fig. 9. Structure of cross sections in the proposed technique. Edges of G_1 are shown in solid lines, those of G_2 in broken lines.

A description of the procedure is given below. For each vertex v (either of G_1 or G_2) we denote as $L(v)$ and $R(v)$ the sets of edges incident to v and lying, respectively, to its left and to its right. The y -structure Y is a dictionary, while the x -structure X is simply an array; I denotes the set of the intersections found; Q is a queue (implemented as a linear list), for which " $\leftarrow Q$ " and " $Q \leftarrow$ " denote the operations "remove" and "add." For simplicity, the algorithm omits consideration of the r -structure.

procedure SWEEP

```

1. begin   for each vertex  $v$  of  $G_1 \cup G_2$  do sort counterclockwise the
           terms of  $L(v)$  and  $R(\bar{v})$ 
2.   sort the vertices of  $G_1 \cup G_2$  by increasing abscissa and place
           them in  $X[1:n]$ 
3.    $Y \leftarrow \phi$ ,  $I \leftarrow \phi$ ,  $Q \leftarrow \phi$ 
4.   for  $i \leftarrow 1$  until  $n-1$  do
5.     begin while  $(L(X[i]) \neq \phi)$  do
6.       begin  $e \leftarrow$  next edge in  $L(X[i])$ 
7.         DELETE( $e$ )
           end (*all edges incident to  $X[i]$  from the left
           are deleted*)
8.          $s \leftarrow$  FIND( $y(X[i])$ )
9.         while  $(R(X[i]) \neq \phi)$  do
10.        begin  $e \leftarrow$  next edge in  $R(X[i])$ 
11.          INSERT( $s;e$ )
12.          if ( $s$  and  $e$  belong to different maps) then
13.             $Q \leftarrow (s,e)$ 
14.             $s \leftarrow e$ 
           end (*all edges issuing from  $X[i]$  to the right
           are inserted*)
15.         $s' \leftarrow$  SUCC( $s$ )
           if ( $s$  and  $s'$  belong to different maps) then  $Q \leftarrow (s,s')$ 
           (*at most two pairs of edges have joined  $Q$ *)
16.        while  $Q \neq \emptyset$  do
17.          begin  $(e_1, e_2) \leftarrow Q$ 
18.            if ( $e_1$  and  $e_2$  intersect) then
19.              begin  $I \leftarrow$  IU( $e_1, e_2$ )
20.                 $e' \leftarrow$  PRED( $e_1$ ),  $e'' \leftarrow$  SUCC( $e_2$ )
21.                if ( $e'$  and  $e_2$  belong to different
22.                  maps) then  $Q \leftarrow (e', e_2)$ 
23.                if ( $e_1$  and  $e''$  belong to different
24.                  maps) then  $Q \leftarrow (e_1, e'')$ 
25.                exchange ( $e_1, e_2$ ) in  $Y$ 
           end
           end
           end

```

Notice that intersections are discovered only when marching on an edge from left-to-right; where an edge is deleted (Step 7) all intersections of this edge have already been found. Moreover, when edges are deleted no intersection-producing adjacencies are generated; indeed if e (being deleted) is incident to v from the left, either there is some e' incident to v from the right which maintains separation, or v is the rightmost vertex of its map: but in this case, no adjacencies between edges of G_1 and G_2 are possible to the right of v . The running time of the algorithm is easily shown to be $O(n \log n)$. Indeed such is the running time of Steps 1 and 2; loop 5-7 uses $O(\log n)$ time for each vertex, and is executed n times; similarly does loop 9-13. Notice that Steps 12, 15, 21, and 22 each use $O(1)$ time, and they are collectively executed $O(s)$ times; finally, loop 16-23 involves $O(s)$ manipulations of the y -structure (Steps 20 and 23) each of which uses $O(1)$ time, since pointers to e_1 and e_2 are available. Thus we conclude that our map intersection algorithm runs in time $O(n \log n + s)$. The $O(n)$ space bound is obvious.

4. VARIANTS OF THE PLANE-SWEEP ALGORITHM

In order to assess the generality of the plane-sweep algorithm, we cast the two instances used in sections 2 and 3 into a common frame.

Both algorithms have the following structure:

Algorithm SWEEP:

1. Initialize x-structure
 y-structure
 (perhaps other data)
2. while x-structure not empty do TRANSITION

where TRANSITION is of the form

1. $P = (x,y) \leftarrow$ remove next point from x-structure
2. with y locate an interval in the y-structure
3. compute i_p new intersections and process these.

The y-structure is identical for both algorithms. It stores the current cross-section consisting of $O(n)$ entries in a data structure that supports the operation FIND in logarithmic time, and the operations PRED and SUCC in constant time.

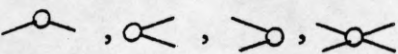
The x-structure is rather different for the two problems we have discussed. The simple case is illustrated by the convex map problem: all relevant transitions are known a priori: the $n = n_1 + n_2$ vertices of the two given graphs. After they have been sorted they can be stored in any static data structure suitable for sequential processing (i.e., the operation NEXT takes constant time) - for example an array. The reason is that the intersections being computed can be processed entirely (an $O(1)$ operation) when they are encountered; hence they

don't need to be considered to be transitions, i.e., they don't need to be stored for deferred processing and retrieved from the x-structure (an $O(\log n)$ operation). By contrast, in the regions-of-a-polygon problem, a computed intersection must be treated as a transition, to be stored and retrieved from the x-structure. This requires a dynamic data structure, which supports the operations MIN and INSERT, and cannot be as efficient as a static data structure. The mere fact that operations on the x-structure now require logarithmic as opposed to constant time, however, would not affect the asymptotic time requirement of the algorithm, since this access time gets absorbed in the $n \log n$ term. The difference between $O(n \log n + s)$ and $O((n+s) \log n)$ is merely due to the fact that $n+s$ transitions move through the x-structure as opposed to n .

This comparison is summarized in the following table:

	regions-in-a-loop	convex maps
SWEEP:		
1. Initialize	sort: $O(n \log n)$	sort: $O(n \log n)$
2. while-do TRANSITION	$(n+s)$ times	n times
TRANSITION:		
1. $P \leftarrow$ remove from x-structure	MIN: $O(\log n)$	NEXT: $O(1)$
2. locate interval in cross section	FIND: $O(\log n)$	FIND: $O(\log n)$
3. process i_p new intersections	$i_p = 0, 1$ or 2 INSERT: $O(\log n)$	i_p times $O(1)$, where $\sum i_p = s$.
TOTAL:	$O(n \log n)$ + $(n+s)O(\log n)$	$O(n \log n)$ + $s \cdot O(1)$

The two algorithms presented can easily be combined to compute the regions of the intersection of two arbitrary maps (non-convex) in time $O((n+s) \log n)$.

In order to do this, however, the classification of points into the 4 categories  (section 2.3) must be changed to deal with one general type of point where an arbitrary number of edges meet. This modification resolves the problem of degeneracy mentioned in section 2.1. An intersection between more than 2 edges in the same point is simply treated as a vertex of high degree. By means of the same generalization, the regions of the intersection of two convex maps can be computed in time $O(n \log n + s)$.

ACKNOWLEDGEMENT

We are grateful to the following people for helpful comments during the development of this paper: H. R. Gnägi, P. Läuchli, T. M. Liebling, E. M. McCreight, K. Lieberherr, J. Waldvogel.

REFERENCES

1. A. Aho, J. E. Hopcroft, and J. D. Ullman, "Analysis and Design of Computer Algorithms," Addison-Wesley, Reading, Mass. 1974.
2. J. L. Bentley and T. A. Ottman, "Algorithms for reporting and counting geometric intersections," IEEE Transactions on Computers, Vol. C-28, No. 9, pp. 643-647, September 1979.
3. J. L. Bentley and D. Wood, "An optimal worst case algorithm for reporting intersections of rectangles," Technical Report, Dept. of Computer Science, Carnegie-Mellon University, July 1979.
4. D. E. Knuth, "The art of computer programming," Vol. 3, "Sorting and Searching," Addison-Wesley, 1973.
5. E. M. McCreight, personal communication.
6. M. I. Shamos and D. Hoey, "Closest-point problems," 16th Annual Symposium on Foundations of Computer Science, 151-162, 1975.
7. M. I. Shamos and D. Hoey, "Geometric intersection problems," 17th Annual Symposium on Foundations of Computer Science, 208-215, 1976.