SIAM J. COMPUT.
Vol. 15, No. 2, May 1986

OPTIMAL POINT LOCATION IN A MONOTONE SUBDIVISION*

HERBERT EDELSBRUNNER†, LEONIDAS J. GUIBAS‡ AND JORGE STOLFI§

Abstract. Point location, often known in graphics as "hit detection," is one of the fundamental problems of computational geometry. In a point location query we want to identify which of a given collection of geometric objects contains a particular point. Let $\mathcal{S}$ denote a subdivision of the Euclidean plane into monotone regions by a straight-line graph of m edges. In this paper we exhibit a substantial refinement of the technique of Lee and Preparata [SIAM J. Comput., 6 (1977), pp. 594–606] for locating a point in $\mathcal{S}$ based on separating chains. The new data structure, called a *layered dag*, can be built in $O(m)$ time, uses $O(m)$ storage, and makes possible point location in $O(\log m)$ time. Unlike previous structures that attain these optimal bounds, the layered dag can be implemented in a simple and practical way, and is extensible to subdivisions with edges more general than straight-line segments.

Key words. point location, monotone polygons, planar graphs, partial order, graph traversal, computational geometry

1. Introduction. *Point location* is one of the fundamental problems in computational geometry. In the two-dimensional case, we are given a subdivision $\mathcal{S}$ of the plane into two or more regions, and then asked to determine which of those regions contains a given query point p. If the same subdivision is to be used for a large number of queries, as is often the case, we can reduce the total cost of this task by preprocessing the subdivision into a data structure suitable for the search. We will measure the performance of a proposed solution to this problem by three quantities, the preprocessing cost P, the storage cost S, and the query cost Q.

Optimal solutions for this search problem have been known since Lipton and Tarjan [LT] and Kirkpatrick [Ki]. For a subdivision $\mathcal{S}$ with m edges these optimal solutions simultaneously attain $S = O(m)$, $Q = O(\log m)$ and, under certain assumptions, also $P = O(m)$. The Lipton–Tarjan method is based on their graph separator theorem, and so far has been only of theoretical interest. Kirkpatrick's method, which consists of building a hierarchy of coarser and coarser subdivisions, is implementable, but still the implied constants are so large as to make current implementations of little practical interest. In addition, neither of these techniques seems to extend in a natural way to curved-edge subdivisions.

Historically, Dobkin and Lipton [DL] were the first to achieve $O(\log m)$ query time, while using $O(m^2)$ space. Their method was subsequently refined by Preparata [P] so that $O(m \log m)$ space suffices. Later Bilardi and Preparata [BP] again gave a refinement that achieves $O(m)$ space in the expected case, while retaining $O(m \log m)$ space and $O(\log m)$ query time in the worst case. These solutions are applicable to curved-edge subdivisions and seem to admit of efficient implementations.

A substantially different approach was taken by Shamos [S] and led to the well-known point location paper of Lee and Preparata [LP], based on the construction

* Received by the editors October 13, 1983, and in final revised form January 28, 1985.

† Institutes for Information Processing, Technical University of Graz, A-8010 Graz, Austria. The work of this author was supported by the Austrian Fonds zur Förderung der wissenschaftlichen Forschung.

‡ Digital Equipment Corporation, Systems Research Center, Palo Alto, California 94301.

§ Digital Equipment Corporation Systems, Research Center, on leave from the University of São Paulo, São Paulo, Brazil. Part of this research was conducted while this author was at the Xerox Palo Alto Research Center, Palo Alto, California. The work of this author was supported in part by a grant from Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

of separating chains. This data structure attains $P = O(m \log m)$, $S = O(m)$, and $Q = O(\log^2 m)$. The constants of proportionality in these expressions are quite small, and the algorithms, particularly the query one, are simpler than those of Kirkpatrick. While Kirkpatrick's algorithm requires the regions to be triangular, that of Lee and Preparata works for regions of a more general shape (monotone polygons, which include the convex ones). Recently Chazelle [C2] described a variant of this solution as a special instance of a general method for "searching in history". His structure needs the same amount of space and time. These techniques again are applicable to curved-edge subdivisions, although this possibility was not examined.

In a separate development, Edelsbrunner and Maurer [EM] came up with a space-optimal solution that works for general subdivisions, and even for sets of arbitrary nonoverlapping regions. The query time is $Q = O(\log^3 m)$, but it can be improved to $Q = O(\log^2 m)$ for rectangular subdivisions, where the structure becomes especially simple. For this reason a generalization to rectangular point location in higher dimensions has also succeeded; see Edelsbrunner, Haring and Hilbert [EH].

The purpose of this paper is to show an elegant modification to the separating chain method of Lee and Preparata that yields a new optimal point location algorithm for monotone subdivisions of the plane. The algorithm is based on a new data structure called the *layered dag*. In this new structure the separating chains built into a binary tree by Lee and Preparata are refined so that (1) once a point has been discriminated against a chain, it can be discriminated against a child of that chain with constant extra effort, and (2) the overall storage only doubles. The layered dag simultaneously attains $S = O(m)$ and $Q = O(\log m)$.[1] An additional insight allows us to build this dag (as well as the original Lee-Preparata tree) in only $O(m)$ time. Not only is this new structure optimal with respect to all of preprocessing, space, and query costs, but in fact a simple implementation, with small constants of proportionality, is possible. Like its Lee-Preparata predecessor, it also extends to curved-edge subdivisions.

In the organization of the paper we have adopted the policy that each new data structure is first introduced by how it is to be used, and then by how it is to be constructed. We have placed emphasis throughout on implementation considerations, as well as the underlying theory. Section 2 describes the basic notions surrounding monotone polygons and subdivisions. Section 3 shows how nonmonotone subdivisions can be made monotone. Sections 4, 5 and 6 introduce a partial ordering of the regions and its use in getting a complete family of separators. Section 7 reviews the Lee-Preparata structure, while §§ 8 and 9 introduce the layered dag, and explain its use in point location and its construction. Section 10 gives an algorithm for constructing a complete family of separators based on a traversal of the subdivision. Two implementation issues are taken up in §§ 11 and 12; these may be omitted on a first reading. Section 11 describes some bit-twiddling trickery used to give us linear preprocessing time, while § 12 discusses how subdivisions can be traversed without auxiliary storage. Finally § 13 contains some further applications of the layered dag to problems in computational geometry.

2. Monotone polygons and subdivisions. An *interval* is a convex subset of a straight line, i.e., the whole line, a ray, a segment, a single point, or the empty set. An interval is *proper* if it contains more than one point, and is *open* if it does not contain its endpoints (if any). A subset of the plane is said to be *monotone* if its intersection with any line parallel to the y axis is a single interval (possibly empty).

[1] This refinement is similar to a technique proposed by Cole [C] and, in a different context, by Chazelle [C3] (the hive graph).

We define a *polygon* as an open, connected, and simply connected subset of the plane whose boundary can be partitioned into finitely many points (*vertices*) and open intervals (*edges*). Note that, according to these definitions, polygons may have infinite extent. A *subdivision* is a partition of the plane into a finite number of disjoint polygons (*regions*), edges, and vertices; these parts are collectively called the *elements* of the subdivision. It follows from the definitions that every edge is on the boundary between two regions (not necessarily distinct), that every vertex is incident to some edge, that every endpoint of an edge is a vertex, and that every region (unless there is only one) has some edge on its boundary. From these facts and from Euler's theorem, we can conclude that a subdivision with m edges has $O(m)$ vertices and regions, thus justifying our use of m as the measure of problem size.

A subdivision is said to be *monotone* if all its regions are monotone and it has no vertical edges. The last condition is a technical one: it is imposed only to simplify the proofs and algorithms and can be removed with some care. Figure 1 illustrates a monotone subdivision (arrowheads denote edges extending to infinity).

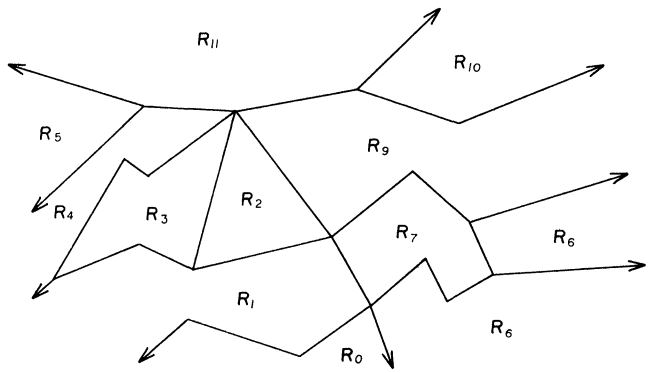

FIG. 1. *A monotone subdivision.*

With minor caveats and modifications, monotone subdivisions include many interesting subcases, such as triangulations, subdivisions of the plane into convex pieces, and so forth. The lemma below shows that monotone subdivisions are precisely the "regular planar straight-line graphs" as defined by Lee and Preparata [LP]:

LEMMA 1. *A Subdivision with no vertical edges is monotone if and only if every vertex is incident to at least two distinct edges, one at its left and one at its right.*

Proof. The "only if" part is easy, since if, for example, all edges incident to a vertex v pointed to the right, then the region to the left of v would have a disconnected intersection with a vertical line through v.

For the converse, assume R is a region that is not monotone, and let l be a vertical line whose intersection with R consists of two or more components, as in Fig. 2. Since R is connected, any two components I_1, I_2 of the intersection are connected by a simple path π in R. We can always choose I_1, I_2, and π so that I_1 is above I_2, and π does not meet l except at its endpoints $p_1 \in I_1$ and $p_2 \in I_2$. Then π and the interval $[p_1, p_2]$ delimit a closed and bounded region S of the plane.

Suppose π lies to the left of l. The boundary of R must enter S at the lower endpoint of I_1 (and also at the upper endpoint of I_2). Since the boundary cannot meet the path π, there must be some vertex of the subdivision in the interior of S. Let v be the *leftmost* such vertex; clearly all edges incident to v lie to the right of it, and we have proved the lemma. A similar argument holds if π lies to the right of l. □

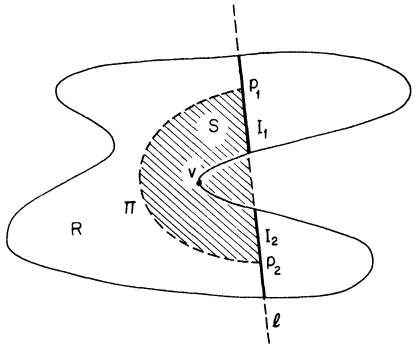

FIG. 2

Therefore, we can check whether any subdivision is monotone by enumerating all the edges incident to each vertex, and checking whether they satisfy Lemma 1. Each edge will be examined at most twice, and each vertex once, so this algorithm runs on $O(m)$ time.

3. Regularization. Lee and Preparata [LP] have shown that an arbitrary subdivision with m edges can be refined into a monotone subdivision having $\leqq 2m$ edges in $O(m \log m)$ time, a process they termed *regularization.* (They define a vertex as being *regular* if it satisfies the condition of Lemma 1.) For the sake of completeness we reproduce their algorithm here, in a slightly different setting.

The task of the regularization process is to add new edges to the subdivision so as to make each vertex regular. Each new edge must connect two existing vertices (or extend from an existing vertex to infinity), and may not intersect any other edges. In other words, we can connect two vertices only if they are *visible* from each other. See Fig. 3 for an example.

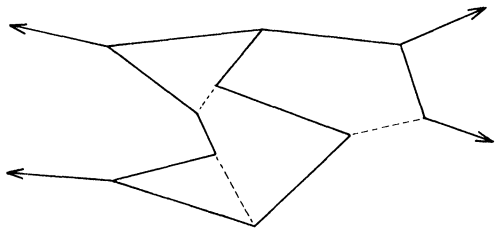

FIG. 3. *Regularizing a nonmonotone subdivision.*

The regularization algorithm is based on the *sweeping line* paradigm that has been used extensively in computational geometry. We imagine that a vertical line sweeps the plane from left to right. We call *active* those vertices, edges, and regions currently intersecting the sweeping line. The list of active elements changes only when the line passes through a vertex, at which time some edges may end and others may start. Except at those moments, the active edges have an obvious vertical ordering, and cut the sweeping line into one or more *active intervals.* To each active interval we assign a *generator,* which is the last vertex swept over that happened to lie on or between the two active edges bounding that interval. This vertex may be a left endpoint of those edges, as Fig. 4 shows, or it may be an irregular vertex with no right-going edge, as Fig. 5 shows.

LEMMA 2. *The generator of an active interval is visible from any point in that interval.*

Proof. By definition, the hatched region in Figs. 4 and 5 is free of vertices and edges. □

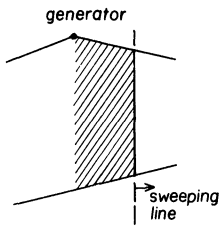

FIG. 4

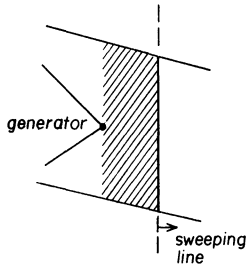

FIG. 5

The regularization algorithm simulates this line sweep. We start by sorting all vertices of the subdivision by their x-coordinates. We will assume these coordinates are all distinct; if necessary, we can enforce this condition by rotating the subdivision through a sufficiently small angle. We also augment the subdivision with two dummy vertices at $x = \pm\infty$, to which are incident all edges having infinite left and right extent, respectively. During the sweep, we maintain a balanced search tree whose leaves represent the active intervals of the current sweep line (and the associated generators), ordered by y-coordinate. Conceptually, we imagine that the sweep line jumps from one corridor between successive vertices to the next, because within a corridor neither the ordering nor the generators of the active intervals can change.

Consider what happens when we pass a vertex v with l left-going and r right-going edges. At that moment we must delete the $l+1$ intervals bounded by edges that end at v, and insert in their place the $r+1$ intervals bounded by edges that start at v. In particular, if $l=0$, we must delete the currently active interval in which v lies, which is bounded by the edges immediately above and below v. Similarly, if $r=0$, we add one new interval bounded by those two edges. Updating the generators is straightforward: the vertex v simply becomes the generator of the $r+1$ new intervals.

When an active interval is about to be deleted, we check whether a new regularizing edge should be added to the subdivision. If the generator u of that interval has no right-going edges, or the new vertex v has no left-going ones, we add a new edge between u and v. See Fig. 6. By Lemma 2, this edge does not intersect any others.

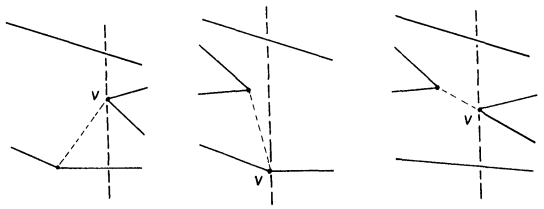

FIG. 6. *Adding new edges.*

When the sweep line reaches the vertex at $x = +\infty$, all irregular vertices will have been fixed in this way. The number of new edges is at most the original number of irregular vertices, which in turn is at most m. The running time is dominated by the cost of sorting of the vertices and manipulating the balanced tree, which is $O(m \log m)$ in the worst case. The problem of whether regularization can be done in time faster than $O(m \log m)$ remains open.

4. The vertical ordering. We denote by ΠA the orthogonal projection of a set A on the x-axis. The projection ΠR of a monotone polygon is an open interval of the

x-axis, whose endpoints, if any, are the projections of at most two *extremal vertices* of R. The *frontier* of R is the boundary of R minus its extremal vertices. This frontier consists of two disjoint pieces, the *top* and the *bottom* of the region R. Each is either empty, or a connected polygonal line, which may extend to infinity in one or both directions.

Given two subsets A and B of the plane, we say that A *is above* B (and write $A \gg B$) if for every pair of vertically aligned points (x, y_a) of A and (x, y_b) of B we have $y_a \geqq y_b$, with strict inequality holding at least once. In this case we also say that B *is below* A (and write $B \ll A$). Some of these concepts are illustrated in Fig. 7. For the rest of this section, we will restrict $\gg$ to the elements of a fixed monotone subdivision $\mathcal{S}$, with n regions and m edges. Then $\gg$ has several interesting properties listed in the lemmas below (straightforward proofs will be omitted).

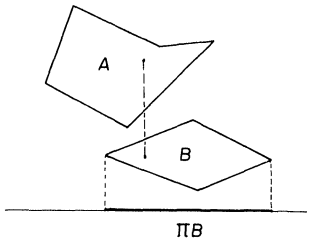

FIG. 7. $A \gg B, \Pi B$.

LEMMA 3. *For any two elements A and B of a monotone subdivision, $A \gg B$ if and only if $A \neq B$ and there is some vertical line segment with lower endpoint in B and upper endpoint in A.* □

LEMMA 4. *Any two elements A and B of a monotone subdivision satisfy exactly one of $A = B$, $A \ll B$, $A \gg B$, or $\Pi A \cap \Pi B = \varnothing$.* □

If three elements A, B and C are intercepted by a common vertical line, then from $A \ll B$ and $B \ll C$ we can conclude $A \ll C$. Therefore, the relation $\ll$ is transitive (and in fact a linear order) when restricted to all the elements intercepted by a given vertical line. Transitivity may not hold without this restriction, but the following property will be true in any case:

LEMMA 5. *The relation $\ll$ is acyclic.*

Proof. Suppose not. Let $E_0 \ll E_1 \ll E_2 \ll \cdots \ll E_k = E_0$ be a cycle of $\ll$ with minimum length, where each E_i is an element of the subdivision $\mathcal{S}$. From Lemma 4, it follows immediately that $k > 2$.

The x-projections of any two consecutive elements E_i and E_{i+1} in this cycle must have some abscissa x_i in common. If for some i we had $\Pi E_i \subset \Pi E_{i-1}$, then the vertical line through x_i would meet E_{i-1}, E_i, and E_{i+1}; transitivity would hold, and we would

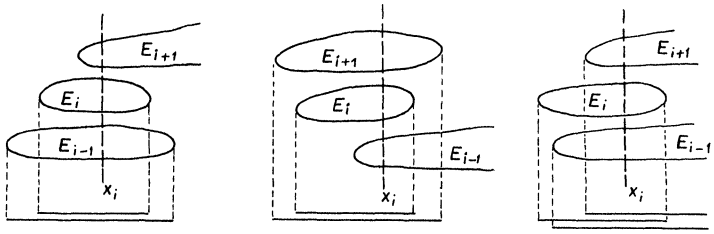

FIG. 8

have $E_{i-1} \ll E_{i+1}$. See Fig. 8. But then we could omit E_i and still get a cycle, contradicting the assumption that k is minimum. For analogous reasons, we cannot have $\Pi E_i \subset \Pi E_{i+1}$.

Let E_i be one of the "leftmost" cycle elements, i.e., such that none of the intervals ΠE_j contains a point to the left of ΠE_i. The projections ΠE_{i-1} and ΠE_{i+1} of its neighbors must both meet ΠE_i, and, by what we have just said, extend beyond its right endpoint. But then there would be again a vertical line meeting all three elements, implying $E_{i-1} \ll E_{i+1}$ and k is not minimum. We conclude that no such cycle exists. □

We say that a region A is *immediately above* a region B (and write $A > B$) if $A \gg B$ and the frontiers of the two regions have at least one common edge; see Fig. 9.

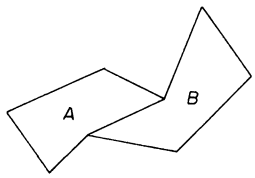

FIG. 9. $A > B$.

In general, the $>$ relation is stronger than $\gg$, but the following is easily seen to be true:

LEMMA 6. *The transitive closure of* $\gg$ (*restricted to the regions of a subdivision*) *is the same as that of* $>$. □

5. Separators. A *separator* for a subdivision $\mathcal{S}$ is a polygonal line s, consisting of vertices and edges of $\mathcal{S}$, with the property that it meets every vertical line at exactly one point. Since s extends from $x = -\infty$ to $x = +\infty$, any element of the subdivision that is not part of s is either above or below it. See Fig. 10. The elements of s have pairwise disjoint projections on the x-axis, and so can be ordered from left to right; the first and last elements are infinite edges.

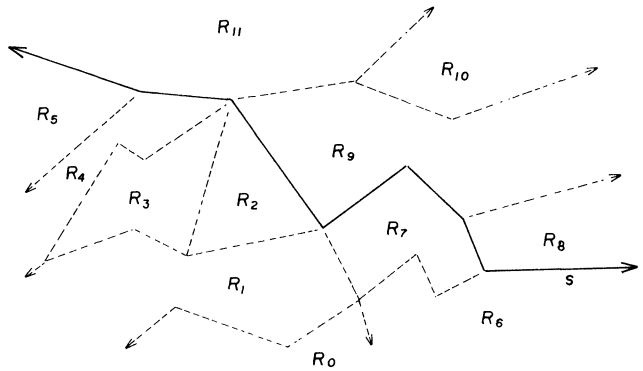

FIG. 10. *A separator.*

A *complete family of separators* for a monotone subdivision $\mathcal{S}$ with n regions is a sequence of $n-1$ distinct separators $s_1 \ll s_2 \ll \cdots \ll s_{n-1}$. There must be at least one region between any two consecutive separators, and also one below s_1 and one above s_{n-1}. If a region is below a separator s_i, it must also be below any separator s_j with $j > i$. We can conclude that, if $\mathcal{S}$ admits a complete family of separators, its regions can be enumerated as $R_0, R_1, \cdots, R_{n-1}$ in such a way that $R_i \ll s_j$ if and only if $i < j$;

in particular,

$$R_0 \ll s_1 \ll R_1 \ll s_2 \ll \cdots \ll s_{n-1} \ll R_{n-1}. \tag{1}$$

Given a complete family of separators and an enumeration of the regions as in (1), let us denote by index (R) the index of a region R in the enumeration. Then $s_{\text{index}(R)} \ll R \ll s_{\text{index}(R)+1}$ (whenever those separators are defined). Also, if we let above (e) be the region above the edge or vertex e, and below (e) the one below it, the following holds:

LEMMA 7. *If* $i = \text{index (below } (e))$ *and* $j = \text{index (above } (e))$, *then the separators containing e will be exactly* $s_{i+1}, s_{i+2}, \cdots, s_j$. □

6. Existence of a complete family of separators. It is a well-known result that any acyclic relation over a finite set can be extended to a linear (that is, total) order. Therefore, by using Lemma 5 we conclude that there is an enumeration $R_0, R_1, \cdots, R_{n-1}$ of the regions of $\mathcal{S}$ that is compatible with $\ll$, i.e., such that $R_i \ll R_j$ implies $i < j$. Furthermore, any enumeration $R_0, R_1, \cdots, R_{n-1}$ of the regions that is compatible with $\prec$ is also compatible with $\ll$, and vice-versa.

Since a region with nonempty bottom frontier is immediately above some other region, and therefore not minimal under $\prec$, the first region in any such enumeration has no bottom frontier, and extends to infinity in the $-y$ direction. Since vertical edges are not allowed, its x-projection is the whole x-axis. Similarly, the last region R_{n-1} is the only one with no top frontier, and also projects onto the whole x-axis. Therefore, we always have $R_0 \ll R_i \ll R_{n-1}$ for $0 < i < n-1$.

We are ready to prove the main result of this section:

THEOREM 8. *Every monotone subdivision* $\mathcal{S}$ *admits a complete family of separators.*

Proof. Let $R_0, R_1, \cdots, R_{n-1}$ be a linear ordering of the regions of $\mathcal{S}$ that is compatible with the $\ll$ relation, i.e. $R_i \ll R_j$ only if $i < j$. For $i = 1, 2, \cdots, n-1$, let s_i be the collection of all edges and vertices that are on the frontier between regions with indices $< i$ and regions with indices $\geqq i$. For example, Fig. 10 shows the separator s_8.

Now consider any vertical line l, and let $R_{i_1}, R_{i_2}, \cdots, R_{i_q}$ be the regions it meets, from bottom to top. Since l meets R_0 and R_{n-1}, and $R_{i_1} \ll R_{i_2} \ll \cdots \ll R_{i_q}$, we will have $0 = i_1 < i_2 < \cdots < i_q = n-1$. Therefore, there is exactly one point on l that is on the frontier between a region with index $< i$ and a region with index $\geqq i$, that is, on s_i. Furthermore, the intersection of l with s_i will be equal to or above that with s_{i-1}, and for some such lines (those that meet R_{i-1}) the two intersections will be distinct. So, we have $s_1 \ll s_2 \ll \cdots \ll s_{n-1}$.

Clearly, the elements of s_i have disjoint x-projections, and therefore can be ordered from left to right; they must be alternately edges and vertices, the first and last being infinite edges. To prove that s_i is a separator, it remains only to show that s_i is connected; if that were not the case, we would have some vertex v of s_i that is distinct from, but has the same x-coordinate as, one endpoint of an adjacent edge e of s_i; see Fig. 11.

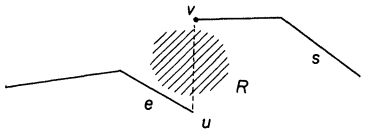

FIG. 11

But then we would have $u \ll R \ll v$ (or vice-versa), for some region R; and therefore $e \ll R \ll v$ (or vice versa), contradicting the construction of s_i. Therefore, each s_i is a separator, and $s_1, s_2, \cdots, s_{n-1}$ is a complete family of them. □

7. A point location algorithm. Later on we will tackle the problem of efficiently computing a complete family of separators for a monotone subdivision. Let us therefore assume for now that we have such a family $s_1, s_2, \cdots, s_{n-1}$, with a corresponding enumeration of the regions $R_0, R_1, \cdots, R_{n-1}$ satisfying (1); we will show next how they can be used to determine, in time $O(\log^2 m)$, the unique element of $\mathcal{S}$ that contains a given point p.

The algorithm we will describe is essentially that of Lee and Preparata [LP], and uses two levels of binary search. The inner loop takes a separator s_i (as a linear array of edges and vertices, sorted by x-coordinate), and determines by binary search an edge or vertex e of s_i whose x-projection contains the abscissa p_x of p. By testing p against e, we will know whether p is above s_i or below s_i (or, possibly, on e itself, in which case the search terminates). The outer loop performs binary search on i, so as to locate p between two consecutive separators s_i and s_{i+1}, that is to say in a region R_i.

Besides the separators, the algorithm assumes the index, index (R) can be obtained for any given region R, and similarly the adjacent regions above (e) and below (e) can be obtained from an edge e, all in constant time. We will see that the construction of these tables is part of the process of constructing the family of separators. The search algorithm uses these tables to reduce substantially the storage requirements (and also speed up the search a little bit).

Let **T** be the infinite, complete binary search tree with internal nodes $1, 2, 3, \cdots$ and leaves $0, 1, 2, 3, \cdots$, as in Fig. 12. The tree **T** is used as a flowchart for the outer loop of the search algorithm, with the convention that each internal node i represents a test of p against the separator s_i, and each leaf j represents the output "p is in R_j". While reading the algorithm it should be borne in mind that "left" in the tree corresponds to "down" in the subdivision. The left and right children of an internal node k of **T** will be denoted by $l(k)$ and $r(k)$, respectively. We let lca (i, j) be the *lowest common ancestor* in **T** of the leaves i and j, that is, the root of the smallest subtree of **T** that contains both i and j.

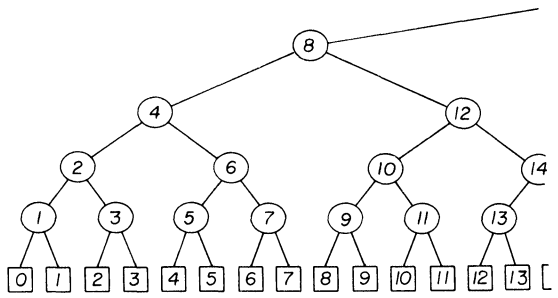

FIG. 12. *The tree* **T**.

When testing p against a separator, we adopt the convention that each edge contains its right endpoint but not its left. This is unambiguous since there are no vertical edges. If the algorithm detects that p lies on some edge e during a discrimination against a separator, it can terminate the search and, by comparing p with the right endpoint of e, determine if our point is a vertex of the subdivision.

ALGORITHM 1. *Point location in a monotone subdivision.*

{*The algorithm returns in the variable loc a reference to the vertex, edge, or region of* $\mathcal{S}$ *containing p.*}

1. Set $i \leftarrow 0$, $j \leftarrow n-1$, $k \leftarrow \text{lca}(0, n-1)$.
2. While $i < j$, do:
   {*At this point p is above the separator* s_i *and below* s_{j+1} (*whenever those separators exist*). *That is, p is in one of the regions* R_i, $R_{i+1}, \cdots, R_j$ (*or in some edge or vertex between two of these regions*). *At each iteration of this loop, either i increases or j decreases, and k* (*a common ancestor of i and j*) *moves down one level in the tree* **T**}.
   3. If $i < k \leqq j$ then
      4. Find (by binary search) an edge e in s_k such that $p_x \in \Pi e$.
         Let $a \leftarrow \text{index (above } (e))$, $b \leftarrow \text{index (below } (e))$.
         {*By Lemma 7, e belongs to the separators* $s_{b+1}, s_{b+2}, \cdots, s_a$. *Therefore, by testing p against e we conclude it is either on e, above* s_a, *or below* s_{b+1}.}
      5. If p is on e, set $loc \leftarrow e$ and terminate the search.
      6. If p is above e, set $i \leftarrow a$; else set $j \leftarrow b$.
   7. Else
      8. If $k > j$ set $k \leftarrow l(k)$; else (if $k \leqq i$) set $k \leftarrow r(k)$.
9. Set $loc \leftarrow R_i$ and terminate the search.

The binary search along each separator s_k can be performed in $O(\log m)$ time if the edges of s_k are stored in a linear array or balanced binary search tree sorted from left to right. By the first iteration, the variable $k = \text{lca}(0, n-1)$ points to an internal node of **T** at level $\lceil \log n \rceil$; at each iteration it descends one level, so we have $O(\log n)$ iterations, and a total time bound of $O(\log n \log m) = O(\log^2 m)$. This bound is tight: Fig. 13 shows a subdivision with m edges that realizes it. This example has $\sqrt{m}+1$ regions and a family of $\sqrt{m}$ disjoint separators with $\sqrt{m}$ edges each.

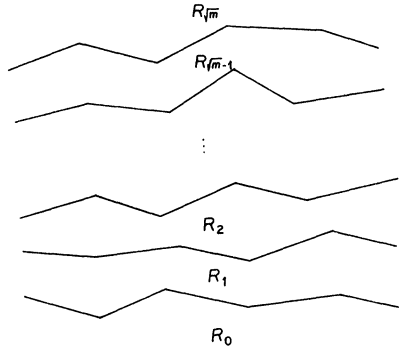

FIG. 13. *A subdivision that is bad for Algorithm* 1.

Note that by keeping track of the variables i and j we are sometimes able to skip the binary search for p_x in some separators. This optimization may improve the average running time of the algorithm in practice, but does not affect the worst-case bound. It was included primarily to make algorithm 1 more similar to the variants developed further on.

If we were to independently represent each separator as a linear array, with all its edges and vertices, we would have to store $n-1$ separators, whose average length can be as large as $\Theta(m)$ for some classes of subdivisions. So, the storage requirement

of this basic method is $\Theta(m^2)$ in the worst case. However, after p has been tested against the edge e in step 6, i and j are updated in such a way that we will never again look at the edges of any other separator that contains e. Therefore, an edge need only be stored in the *first* separator containing it that would be encountered in a search down the tree **T**. Specifically, if the edge e is in the common frontier of regions R_i (below) and R_j (above), by Lemma 7 it belongs to separators $s_{i+1} \cdots s_j$ and so it suffices to store it in s_k, where k is the least common ancestor of i and j. This is the highest node in **T** whose separator contains e.

Note that only those edges assigned to s_k according to the above rule are actually stored in such a structure. In general these will form a proper subset of all the original edges of s_k, so between successive stored edges of s_k there may be *gaps*. See Fig. 14. Actually, it may happen that all the edges of s_k are stored higher up in the tree, so that s_k is reduced to a single gap, extending from $x = -\infty$ to $x = +\infty$. The ordered list of stored edges and gaps corresponding to separator s_k will be termed the *chain* c_k. Note that to each subtree of **T** rooted at a node k there corresponds a "partial subdivision" consisting of a range of regions and the edges and vertices between them. The separator s_k splits this partial subdivision into two others, each having half the regions; the gaps of c_k correspond to edges of s_k that lie on the upper or lower boundary of the partial subdivision, as shown in Fig. 14.

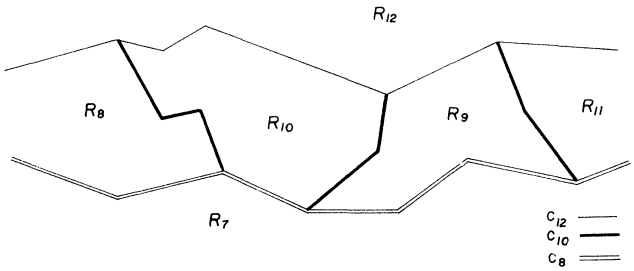

FIG. 14. *Stored edges and gaps.*

The total storage required to represent the chains is only $O(m)$; in § 10 we show how they can be constructed in $O(m)$ time. The derivation of this bound is contingent on our ability to compute the least common ancestor of any two leaves of **T** in $O(1)$ time. This is made possible by the fixed, regular structure of the search tree **T** (see § 11). The point location phase proper could easily be adapted to search a more conventional linked tree structure, but such structures seem to admit no simple $O(1)$ algorithm for lca determination.

8. A faster point location method. Our hope to obtain a faster algorithm for point location comes from the fact that there is some obvious information loss in the method of § 7. Specifically, when we discriminate a point against a chain c_k, we must localize it in the x coordinate to within an edge or a gap of c_k. Yet when we continue the search down some child of k, we start this localization process all over again. It would be nice if each edge or gap in a chain pointed to the place on each child chain where the x search on that child will finish. The trouble is that an edge or gap of the parent can cover many edges or gaps of the child.

The novel idea in the technique we present in this section is to refine the chains so that the localization of p_x in one chain allows us to do the same localization in its children with only *constant* extra effort. In its ultimate form, this idea leads to breaking up each chain at the x coordinates of all vertices of the subdivision. A bit of thought will convince the reader, however, that such an extensive refinement will require

quadratic storage for its representation. Instead, we describe below a refinement that produces for each chain c_k a list L_k of x-values, defining a partitioning of the x axis into *x-intervals.* Each such interval of L_k overlaps the x-projection of exactly one edge or gap of c_k and *at most two* x-intervals of the lists $L_{l(k)}$ and $L_{r(k)}$. As we will see in § 9, this last condition is compatible with keeping the overall storage linear.

The lists L_k and their interconnections can be conveniently represented by a linked data structure that we call the *layered dag.* This is a directed acyclic graph whose nodes correspond to tests of three kinds: *x-tests, edge tests,* and *gap tests.* A list L_k is represented in the dag by a collection of such nodes: each x-value of L_k gives rise to an x-test, and each interval between successive x-values to an edge or gap test. See Fig. 15.

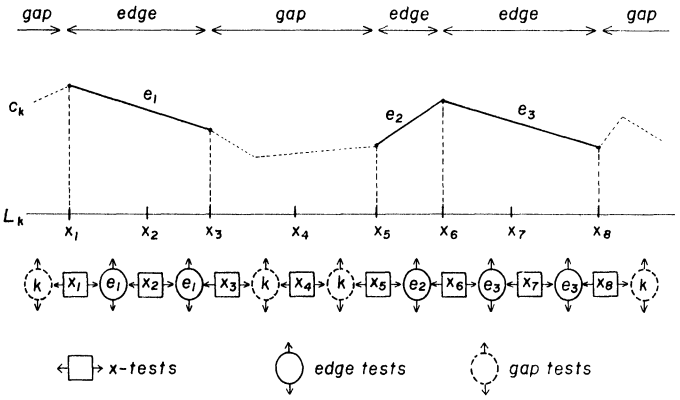

FIG. 15. *The dag nodes for list* L_k.

An x-test node t contains the corresponding x-value of L_k, denoted by xval (t), and two pointers left (t) and right (t) to the adjacent edge or gap nodes of L_k. An edge or gap test node t contains two links down (t) and up (t) to appropriate nodes of $L_{l(k)}$ and $L_{r(k)}$. In addition, an edge test contains a reference edge (t) to the edge of c_k whose projection covers the x-interval represented by t. A gap test node contains instead the chain number chain $(t) = k$. The various types of nodes present are illustrated in Fig. 16.

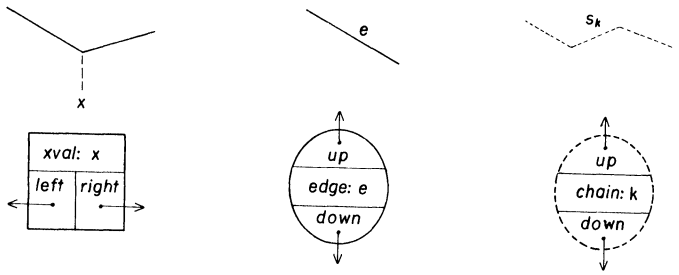

FIG. 16. *The nodes of the layered dag.*

Let us define more precisely the meaning of the links down (t) and up (t). The properties of the refined lists ensure that the x-interval of L_k corresponding to an edge or gap test t covers either one x-interval I of $L_{l(k)}$, or two such intervals I_1, I_2 separated by some element x_k of $L_{l(k)}$. In the first case, down (t) points to the edge or gap test of $L_{l(k)}$ corresponding to the interval I; in the second case, down (t) points to the x-test corresponding to the separating abscissa x_k. Similarly, the link up (t) points to

a node of $L_{r(k)}$ defined in an analogous manner. In the special case when $r(k)$ and $l(k)$ are leaves of **T**, we let down $(t) = \text{up}(t) = nil$.

The layered dag then consists of the test nodes for all lists $L_1, L_2, \cdots, L_{n-1}$, linked together in this fashion. We can use this dag to simulate algorithm 1; the main difference is that each time we move from a separator to one of its children, the down or up pointers (possibly together with an x-test) allow us to locate x_p in the new chain in $O(1)$ time. As before, the variables i and j keep track of the interval of separators in which the point p is known to lie, namely above s_i and below s_{j+1}. This interval is updated after each edge test exactly as in the previous algorithm, and is used during a gap test. When we come to a gap test, we know that the chain number of the gap is not interior to the interval in question. This gives us an unambiguous test as to whether we are above or below the chain of the gap. By the time the search algorithm gets to a null up or down link, the interval will have been narrowed down to a single region.

We have now presented enough details about the structure of the layered dag that we can give the code for the point-location algorithm. The layered dag contains a distinguished node *root* where the point location search begins. This node is the root of a balanced tree of x-tests whose leaves are the edge tests corresponding to the list for the root node of **T**.

ALGORITHM 2. *Fast point location in a monotone subdivision.*

{*This algorithm takes as input the root node of the layered dag and the number n of regions. Its output, as in algorithm* 1, *is placed in the variable loc.*}

1. Set $i \leftarrow 0$, $j \leftarrow n-1$, $t \leftarrow root$.
2. While $i < j$ do:
   {*At this point we know p is above the separator* s_i *and below* s_{j+1} (*whenever those separators exist*). *That is, p is in one of the regions* $R_i, R_{i+1}, \cdots, R_j$ (*or in some edge or vertex between two of these regions*). *The variable t points to a test node in the layered dag, which together with its descendants will allow us to locate the point p among those regions.*}
   3. If t is an edge test then let $e \leftarrow \text{edge}(t)$ and do:
      {*At this point we know* p_x *lies within* Πe.}
      4. If p is on e, set $loc \leftarrow e$ and terminate the algorithm.
      5. If p is above e, set $t \leftarrow \text{up}(t)$ and $i \leftarrow \text{index}(\text{above}(e))$.
         Else set $t \leftarrow \text{down}(t)$ and $j \leftarrow \text{index}(\text{below}(e))$.
   6. Else if t is an x-test then do:
      {*The following x-test routes us to the appropriate edge of the next chain we need to test against.*}
      7. If $p_x \leqq \text{xval}(t)$ then $t \leftarrow \text{left}(t)$ else $t \leftarrow \text{right}(t)$.
   8. Else t is a gap test; do
      {*We have already compared p against the appropriate edge of the chain of the gap test. We just need to recontruct how that comparison went.*}
      9. If $j < \text{chain}(t)$ then $t \leftarrow \text{down}(t)$ else $t \leftarrow \text{up}(t)$.
10. Set $loc \leftarrow R_i$ and terminate the search.

9. Computing the layered dag. Now that we understand how the layered dag is to be used, we will describe how it is to be constructed. Our starting point will be the tree **T** and the chains c_k defined in § 7; recall that c_k consists of those edges of s_k that do not belong to any ancestor of s_k (that is, to any separator whose index is an ancestor of k in **T**).

Our construction of the layered dag proceeds from the bottom up and happens simultaneously with the refinement of the chains c_k. We first describe how the x values

in L_k are obtained. Note that we already have at our disposal three sorted lists of x values: those corresponding to $L_{l(k)}$, to $L_{r(k)}$, and also to the endpoints of the edges stored with the chain c_k associated with node k in the chain tree. The x values in L_k are a merge of those in c_k, and *every other one* of those present in each of $L_{l(k)}$ and $L_{r(k)}$. By convention, if k is a terminal node of the chain tree (so it corresponds to a region), or $k \geqq n$, then L_k is empty. We imagine now that, in a bottom-up fashion, this is done for every node k of the chain tree. The propagation of every other value from the children to the father constitutes the chain refinement we had mentioned in § 8. This refinement has two desirable properties:

LEMMA 9. *An interval between two successive x values in L_k overlaps at most two intervals in $L_{l(k)}$, or $L_{r(k)}$, respectively.* □

LEMMA 10. *The total number of x values in the lists L_k, summed over all k, is at most 4m.*

Proof. If a_k denotes the number of edges in c_k, then

$$\sum_{k \in T} a_k = m,$$

since each edge of the subdivision occurs in exactly one chain c_k. Let b_k denote the number of x values in L_k, and A_k (resp. B_k) denote the sum of a_i (resp. b_i) over all nodes i in the subtree rooted at k. To prove the lemma it suffices to show that

$$B_r \leqq 4A_r = 4m$$

for the root node $r = \text{lca}\,(0, n-1)$.

As an inductive hypothesis now assume that

$$B_i + b_i \leqq 4A_i \tag{2}$$

for $i = l(k)$ or $i = r(k)$. This hypothesis is trivially true for the leaves of **T**. Observe now that

$$B_k = B_{l(k)} + B_{r(k)} + b_k,$$

and that

$$b_k \leqq 2a_k + (b_{l(k)} + b_{r(k)})/2,$$

since each edge in c_k contributes at most two x-values to L_k. Applying the inductive hypothesis (2) yields

$$B_k + b_k \leqq 4A_k,$$

which proves the same conclusion for k. This concludes the proof of the lemma. □

Intuitively, half of the x values of c_k propagate to the father chain, a quarter to the grandfather, an eighth to the great-grandfather, and so on. Although this argument is not strictly correct it illustrates why we can expect the combined length of the L lists to remain linear in m, and in fact just twice as great as the total number of x values in the chains c_k.

The construction of the x-test, edge test, and gap test nodes representing the list L_k in the layered dag is now straightforward, as well as the setting of the left and right fields of the x-tests. The down links of L_k can be set according to the rules stated in § 8, by traversing simultaneously the two lists L_k and $L_{l(k)}$ from left to right. The up pointers are analogous. In fact, it is possible to build L_k and link it to the rest of the dag in a single simultaneous traversal of c_k, $L_{l(k)}$, and $L_{r(k)}$. This bottom-up process terminates with the construction of the root chain L_r, where $r = \text{lca}\,(0, n-1)$. As a final

step we produce a tree of x-test nodes corresponding in a natural way to a binary search on the x-values of the list L_r. The leaves of the tree are made to point to the appropriate edge test nodes of L_r. All nodes of the layered dag can be reached from the root of that tree.

ALGORITHM 3. *Construction of the layered dag.*

1. Set $i \leftarrow 1$. While $i < n$ do:
   {*Construct one more level of the tree* **T**. *The first node in this level is* i *and the difference between successive nodes is* $2i$. *A node* k *on this level is the common ancestor of the leaves* $k-i$ *through* $k+i-1$; *its children* (*if* $i>1$) *are the internal nodes* $k-i/2$ *and* $k+i/2$.}
2. Set $k \leftarrow i$. While $k - i < n$ do:
   {*Create the list* L_k *from the chain* c_k *and the lists* $L' = L_{l(k)}$ *and* $L'' = L_{r(k)}$ (*if they exist*).}
   3. If $i = 1$, set $L' \leftarrow \varnothing$, else set $L' \leftarrow L_{k-i/2}$.
   4. If $i = 1$ or $k + i/2 \geqq n$, set $L'' \leftarrow \varnothing$, else set $L'' \leftarrow L_{k+i/2}$.
   5. If $k \geqq n$, let c_k be a single gap from $x = -\infty$ to $x = +\infty$.
      Split the edges and gaps of c_k at every other x-value of L' and L''.
   6. Set $L_k \leftarrow \varnothing$. For each edge or gap e in c_k, do:
      7. Append to L_k an edge or gap test t representing e.
         Set edge $(t) \leftarrow e$ or chain $(t) \leftarrow k$, as appropriate.
      8. If $L' = \phi$, let down $(t) \leftarrow$ *nil*.
         Else, if Πe overlaps only one x-interval of L', let down (t) point to the corresponding edge or gap test of L'.
         Else Πe overlaps two x-intervals of L'; create a new x-test node t' that chooses between the two corresponding edge or gap tests of L', and let down $(t) \leftarrow t'$.
      9. Similarly, set up (t) to *nil*, to an edge or gap test of L'', or to a new x-test that chooses between two tests of L''.
   10. Set $k \leftarrow k + 2i$.
   11. Set $i \leftarrow 2i$.
12. Let $r \leftarrow \text{lca}(0, n-1)$. Construct a binary tree of x-tests for the list L_r, and let *root* point to this tree.

Note that several different nodes of L_k may point to the same node of a child; an example is shown in Fig. 17. Thus the resulting dags are not trees. We remark that

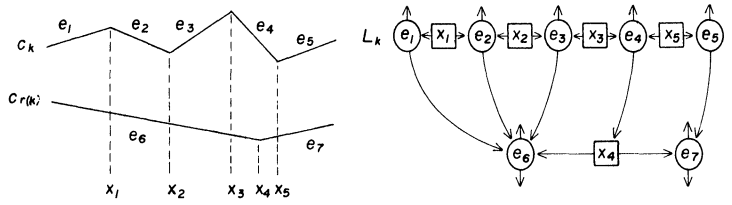

FIG. 17. *Convergence in the dag.*

the structure built by Kirkpatrick [Ki] also corresponds to a dag. This "sharing" of subtrees seems to be an essential feature of algorithms for point location that simultaneously attain optimal space and query time.

To cut down on the number of links, we may consider storing the edge and gap test nodes of each list L_k in consecutive memory locations, in their natural left-to-right

order, with the x-tests between them. The initial location of x_p in L_r could then be carried out by standard binary search. Also, in the construction of L_k we would be able to scan the lists $L_{l(k)}$ and $L_{r(k)}$ without any auxiliary pointers or tables. Finally, this sequential allocation would eliminate the need for the left and right links of x-tests, a fact that may reduce the total storage used by pointers in the dag by about one third.

In any case, the initial location of x_p in the root list L_r can be determined in $O(\log m)$ time. After that, Algorithm 2 executes exactly $\lceil \lg n \rceil$ edge or gap tests (one at each level of **T**), and at most that many x-tests. So the total query time is $O(\log m)$. A list L_k with t x-values is represented in the layered dag by at most t x-tests and $t+1$ edge/gap tests and, as we have seen, it can be constructed in $O(t)$ time. Using Lemma 10, we conclude that the layered dag contains at most $4m$ x-tests and $4m+n-1$ edge and gap tests, and can be built in total time $O(m+n)$ from the chain tree **T**. In summary, we have shown that

THEOREM 11. *Assuming that the chain tree for a subdivision with m edges is given, the layered dag data structure can be constructed in $O(m)$ time, takes $O(m)$ space, and allows point location to be done in $O(\log m)$ time.* □

10. Constructing a complete family of separators. We proceed now to the description of an efficient algorithm that constructs the chain tree **T** representing a complete family of separators for a given monotone subdivision $\mathcal{S}$. As suggested by the proof of Theorem 8, the first pass of this algorithm enumerates the regions in a linear order compatible with $\ll$. A second pass enumerates the edges and vertices of $\mathcal{S}$, in such a way that the edges and vertices of each separator are visited in order of increasing x-coordinate (even though this may not be true for elements belonging to different separators). Therefore, by just appending each visited element to its appropriate separator, we will get the sorted lists required in Algorithms 1 and 3.

As in § 3, we will add to S two dummy vertices at $x=-\infty$ and $x=+\infty$, which are endpoints of all edges with infinite left or right extent, respectively. If we orient every edge of $\mathcal{S}$ from right to left, we obtain a planar embedding of a directed, acyclic graph S with exactly one source and one sink (solid lines in Fig. 18). The enumeration we need in the second pass is basically a *compatible traversal* of this graph, in which we visit a vertex only after visiting all its outgoing edges, and we visit an edge only after visiting its destination. This is a form of topological sorting, as discussed in [Kn].

Consider now the dual graph S^* whose vertices are the regions of $\mathcal{S}$, and where for each edge e of $\mathcal{S}$ there is an edge e' from above (e) to below (e). By what we have seen in §§ 4–6, S^* too is acyclic and has exactly one source and one sink. We can draw S^* on top of S (dotted lines in Fig. 18) in such a way that each of its edges e' crosses only the corresponding edge e of S, and that exactly once (going down).

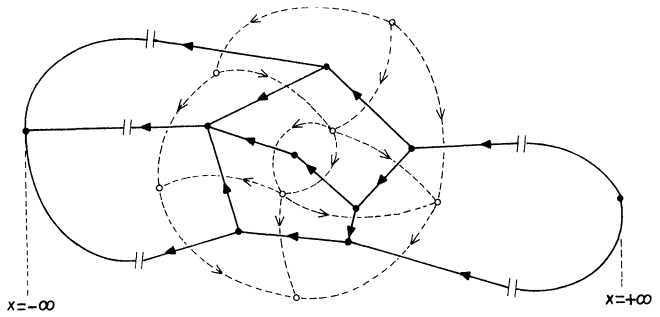

FIG. 18. *The graphs S and S**.

Therefore, S^* is planar, and corresponds to the topological dual of S. It turns out that S^* represents for the first pass what S does for the second: a compatible traversal of S^* will visit the regions in an order consistent with $\prec$ and $\ll$.

Therefore, both passes reduce to a generic graph traversal algorithm, applied first to S^* and then to S. In the first pass, as each region R is visited, we store in a table (or in a field of the region's record) its serial number index (R) in the enumeration. In the second pass, as each edge or vertex e is visited, we obtain the indices $a =$ index (above (e)) and $b =$ index (below (e)) of the regions immediately above and below it, and append e to the separating chain c_k where $k = \text{lca}(a, b)$. With an appropriate representation for subdivisions, it is possible to accomplish this graph traversal without any auxiliary storage. This topic is discussed in § 12.

11. The lowest common ancestor function. If the construction of the separating chains c_i as described above is to run in $O(m)$ time, it is essential that the total time to compute lca (index (below (e)), index (above (e))) for all edges e be $O(m)$. This rules out the straightforward algorithm that starts at the leaves i and j, and moves up one level of **T** at a time, following "parent" links, until the two paths join at a common node. This naïve algorithm has running time $\Omega(\log |i-j|)$, and it is possible to have subdivisions in which $|\text{index (below } (e)) - \text{index (above } (e))| = \Omega(\sqrt{m})$ for $\Omega(m)$ edges e, thus giving an overall running time of $\Omega(m \log m)$.

Algorithms for computing in $O(1)$ time the least common ancestor function on general binary trees have been published by Harel [H]. His algorithms are probably too complex to be of practical use here, but they can be considerably simplified thanks to the regular structure of our search tree **T**. On this tree, the value of lca (i, j) has a simple interpretation in terms of the binary representations of i and j. Let $\nu = \lceil \lg n \rceil$ be the number of bits needed to represent any number from 0 through $n-1$, and let

$$i = (a_{\nu-1}a_{\nu-2} \cdots a_{r+2}\ 0\ a'_r \cdots a'_1 a'_0)_2,$$
$$j = (a_{\nu-1}a_{\nu-2} \cdots a_{r+2}\ 1\ a''_r \cdots a''_1 a''_0)_2.$$

Then

$$\text{lca}(i, j) = (a_{\nu-1}a_{\nu-1} \cdots a_{r+2}\ 1\ 0\ \cdots 0\ 0)_2.$$

Loosely speaking, lca (i, j) is the longest common prefix of i and j, followed by 1 and padded with zeros.

An efficient formula for computing lca (i, j) is based on the function $\text{msb}(k) = \text{lca}(0, k) = 2^{\lceil \lg k \rceil - 1}$, the *most significant bit of* k. Its values for $k = 1, 2, \cdots, n-1$ are 1, 2, 2, 4, 4, 4, 4, 8, 8, $\cdots$, $2^{\nu-1}$; these numbers can be easily precomputed in $O(n)$ time and stored in a table with $n-1$ entries, so we can compute msb (k) in $O(1)$ time. Then we can express the lca function as

$$\text{lca}(i, j) = j \wedge \neg(\text{msb}(i \oplus j) - 1)$$

where $\oplus$, $\wedge$, and $\neg$ are the boolean operations of bitwise "exclusive or", "and", and complement. We assume these boolean operations can be computed in $O(1)$ time, like addition and subtraction. We feel their inclusion in the model is justified, since their theoretical complexity is no greater than that of addition, and most computers have such instructions.[2] For use in this formula, it is preferable to tabulate $\neg(\text{msb}(k) - 1)$ instead of msb (k).

[2] In fact, some machines can even compute $\lceil \lg k \rceil - 1$ from k ("find first 1 bit") and 2^q from q ("shift left q bits"), in a single instruction cycle. Under a more rigorous log-cost complexity model, all time and space bounds in this paper and in the main references should be multiplied by an extra log m factor.

Another way of computing lca is based on the *bit-reversal* function rev (k), that reverses the order of the last ν bits in the binary expansion of k. For example, for $n=16$ and $k=0, 1, \cdots, 15$, the values of rev (k) are 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15. Using this function we get the formula

$$\text{lca}\,(i,j) = \text{rev}\,(k \oplus (k-1)) \wedge j,$$

where $k = \text{rev}\,(i \oplus j)$.

12. Compatible traversal of an acyclic graph. The input to the compatible graph traversal routine we mentioned in § 10 is a planar embedding G of a directed, acyclic, and connected graph with exactly one sink and one source, both on the exterior face of G. See Fig. 19.

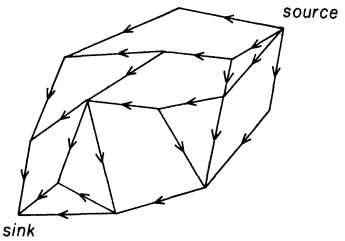

FIG. 19. *The input to Algorithm* 4.

Such an embedding defines a "counterclockwise" circular ordering of the edges incident to any given vertex u. The *post-order traversal* of G is defined as a listing of all its vertices and edges such that

(i) an edge is listed (or *visited*) only after its destination,
(ii) a vertex is visited only after all its outgoing edges, and
(iii) edges with same origin are visited in counterclockwise order.

This is clearly a compatible traversal as defined in § 10. The post-order traversal is unique, and is a particular case of the general depth-first graph traversal described by Tarjan [Ta]. This problem admits a straightforward recursive solution. Given a vertex u (initially the source of G), we enumerate the edges out of u, in counterclockwise order. For each edge e, we first recursively apply the procedure to its destination v (unless it has been previously visited). We then visit e and proceed to the next edge. After all edges out of u have been visited, we visit u and exit. This algorithm runs in $O(m)$ time and requires only $O(m)$ auxiliary storage, the latter consisting of the recursion stack and one mark bit per vertex (to distinguish the nodes that have already been visited).

In the rest of this section we will show that this post-order traversal can be performed without an auxiliary stack or any mark bits on the vertices, provided the data structure used to represent the subdivision is rich enough. This improvement is of significant practical interest, even though it does not affect the $O(m)$ space bound.

As we observed, the embedding of G in the plane defines a counterclockwise ordering of the edges incident to a given vertex u. In this ordering all outgoing edges occur in consecutive positions, and the same is true of the incoming ones. To see why, consider any two edges e_1, e_2 entering u, and any two edges g_1, g_2 leaving u. Let π_1, π_2 be two paths from the source vertex of G that end with the edges e_1 and e_2, and let σ_1, σ_2 be the two paths to the sink vertex that begin with g_1, g_2. See Fig. 20.

Since G is acyclic, both π_1 and π_2 are disjoint from σ_1 and σ_2 (except for u itself). Now, the paths π_1 and π_2 together divide the plane in two (or more) regions. If the two pairs of edges were interleaved, at least one of the paths σ_1 and σ_2 would have

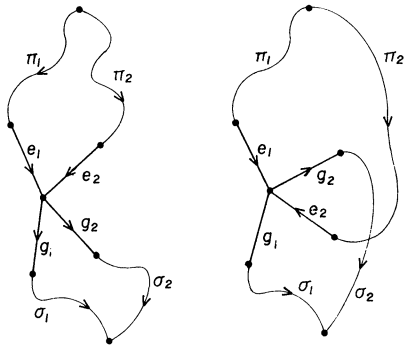

FIG. 20

to cross π_1 or π_2, since they start on different regions but have a common destination. This proves the above assertion.

If u has both incoming and outgoing edges, this result establishes a *linear* order for each class with well-defined "first" and "last" elements, which will be denoted by first in (u), last in (u), first out (u), and last out (u). See Fig. 21. To make this definition meaningful also for the source and sink of G, we will introduce a dummy edge base (G) that connects the sink back to the source across the exterior face of G. We may consider the resulting graph G' as embedded on the upper half of a sphere, with base (G) running across the bottom half, as in Fig. 22. In the graph G', the first and last outgoing edges of the source of G will be those incident to the exterior face of G. A similar statement applies to the sink of G.

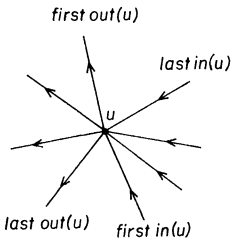

FIG. 21

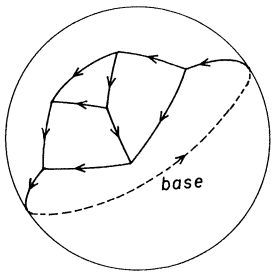

FIG. 22

Let us mechanically translate the recursive post-order algorithm into an iterative one, using an explicit stack Q to save the value of e when simulating recursive calls (the value of u need not be saved, since it is always the origin of e).

ALGORITHM 4. *Post-order traversal of an acyclic planar graph with one source and one sink.*

1. Set $Q \leftarrow \emptyset$, $u \leftarrow$ source of G, $e \leftarrow$ first out (u).
2. Repeat the following steps:
   {*At this point u is unvisited, and e is the first unvisited edge out of u.*}
   3. While $e \neq$ base (G) and the destination v of e has not been visited yet,
      4. Set $u \leftarrow v$, push e onto the stack Q, and set $e \leftarrow$ first out (u).
   5. If $e \neq$ base (G), visit e.
   6. While $e =$ last out (u) do
      7. Visit u.
      8. If Q is empty, the algorithm terminates. Else,
         9. Pop the topmost edge of Q, and assign it to e.
            Set u to the origin of e.
      10. Visit e.
   11. Set e to the next edge out of u.

The following claims about Algorithm 4 are a direct consequence of the planarity and acyclicity of G, and can easily be proved by induction on the number of executed steps:

(I) Before every step, the edges in the stack Q form a path in G from the source to u;
(II) an edge is stacked onto Q (step 4) only if its destination is still unvisited;
(III) an edge is unstacked (step 9) only after its destination has been visited;
(IV) an edge is visited only after its destination has been visited;
(V) a vertex is visited only after its last outgoing edge has been visited;
(VI) every edge is stacked at most once;
(VII) for any given vertex, at most one incoming edge ever gets stacked; and
(VIII) an edge is visited only after the previous edge with same origin (if any) is visited.

In particular, from (IV), (V), and (VIII) we conclude that all vertices and edges of G are visited, and conditions (i)-(iii) are satisfied. Algorithm 4 defines a subgraph H of G, consisting of all the edges that ever get stacked onto Q. Every vertex of G is reachable from the source via a directed path in H (the contents of the Q at any instant when the variable u is that vertex), and has at most one incoming edge in H. It follows that H is an oriented spanning tree of G, as illustrated in Fig. 23. We remark

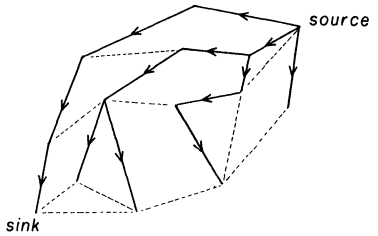

FIG. 23. *The spanning tree H.*

that the order in which the vertices of G are visited corresponds to the post-order traversal of the tree H, as defined by Knuth [Kn]. We will show now that the only edge of H (if any) entering a vertex u is last in (u). More precisely, the following lemma holds:

LEMMA 12. *Before any step of algorithm* 4, *if the stack Q is not empty, its topmost edge is last in* (u); *otherwise u is the source of G.*

Proof. The second part of the lemma is obvious, since the contents of Q is always a path in G from the source to u. Let us then assume Q is not empty. Let u_1 be the current value of u, π be the path in Q, and e_1 be the last edge of π (i.e., the top of Q). See Fig. 24. Suppose e_2 is an edge distinct from e_1 but having the same destination u_1. From assertions (II) and (III) above we conclude that e_1 and e_2 have yet to be

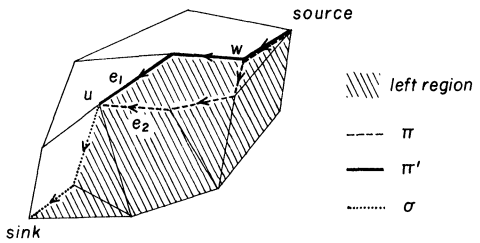

FIG. 24

visited. By the time e_2 is visited by Algorithm 4, the variable u will contain the origin of e_2; the contents of Q at that time, plus the edge e_2, will define another directed path π' from the source to u_1. Since G is acyclic, the path π' neither contains nor is contained in π; in fact, because of property (VII) the paths π and π' must diverge exactly once, at some vertex $w \neq u_1$, and converge exactly once at u_1. Therefore all edges of π between w and u_1 must be unstacked (and visited) before e_2 is visited. In particular, the edge α through which π leaves w is visited before the corresponding edge α' of π', and therefore α' must follow α in counterclockwise order around w.

Let R be the complement of the outer face of G, and let σ be any directed path from u_1 to the sink of G. The concatenation of π and σ divides R in two (not necessarily connected) regions, which we call *left* and *right* (with the direction of $\pi\sigma$ being taken as "forwards"). The path π' cannot cross σ (otherwise the two would give rise to a cycle), and its first edge α' lies in the left region; therefore, after leaving w the path π' must lie entirely in the left region, and in particular e_2 precedes e_1 in the counterclockwise order around u_1. By repeating this argument for all possible edges $e_2 \neq e_1$ into u_1, we conclude that $e_1 = \text{last in}(u)$. □

Therefore, instead of retrieving e from the stack in step 9, we can simply set $e \leftarrow \text{last in}(u)$. Note also that the test "$e \neq \text{base}(G)$ and v has not been visited yet" in step 3 is evaluated and yields true if and only if e is pushed into Q by the following step, so we can replace that test by the condition "$e \neq \text{base}(G)$ and $e = \text{last in}(v)$". These observations enable us to dispense with the recursion stack and the "visited" bits on the vertices.

A representation for the embedded graph G that allows the efficient computation of first out and its companions is the *quad-edge* data structure [GS]. This representation is similar to the well-known "winged edge" and DCEL data structures [B], [MP], but has over them the important advantage of encoding simultaneously both G and its dual embedding, in precisely the same format, at a negligible extra cost in storage. This allows the post-order traversals of both S and S^* to be performed by the same procedure, applied to the same data structure.

The quad-edge structure by itself can only represent an *undirected* embedded graph, such as the undirected subdivision $\mathcal{S}$. However, every edge $\bar{e}$ of $\mathcal{S}$ is represented by two distinct records in the structure, corresponding to the two possible orientations of $\bar{e}$. Therefore, to refer to the edge $\bar{e}$ of the structure we must actually refer to a specific *directed* version of $\bar{e}$. Given such a directed edge e, the quad-edge data structure gives immediate access to:

org (e) the origin vertex of e,
dest (e) the destination vertex of e,
onext (e) the next counterclockwise directed edge with the same origin,
dnext (e) the next counterclockwise directed edge with the same destination,
sym (e) the same edge directed the other way, and
rot (e) the dual of the edge e, directed from the right to the left faces of e.

A *directed graph* G, such as S or S^*, can be represented by the quad-edge encoding of the corresponding undirected graph, plus a predicate forward (e, G) that tells whether the directed edge e of the structure is actually an edge of G. Clearly, e is in G if and only if sym (e) is not in G, so forward (e, G) $\equiv \neg$forward (sym (e), G). In our case, forward (e, S) is simply the test of whether the x-coordinate of dest (e) is smaller than that of org (e). Similarly, in the dual graph S^* the predicate forward (e, S^*) tests whether the region dest (e) is immediately below the region org (e). This turns out to be the same as $\neg$forward (rot (e), S). The dummy edges that we must add to S and S^* are the only exception: we have rot (base (S^*)) = base (S), and yet both are

forward. For both graphs, we also have

$$e = \text{last in}(\text{dest}(e)) \Leftrightarrow \text{forward}(e, G) \wedge \neg\text{forward}(\text{dnext}(e), G),$$

$$e = \text{last out}(\text{org}(e)) \Leftrightarrow \text{forward}(e, G) \wedge \neg\text{forward}(\text{onext}(e), G),$$

$$e = \text{last in}(u) \Rightarrow \text{sym}(\text{dnext}(e)) = \text{first out}(u).$$

These identities allow us to remove also the calls to first out and its companions from Algorithm 4. Algorithm 5 below incorporates all these modifications.

ALGORITHM 5. *Post-order traversal of an acyclic planar graph with one source and one sink using $O(1)$ auxiliary storage.*

1. Set $u \leftarrow \text{dest}(\text{base}(G))$ and $e \leftarrow \text{sym}(\text{dnext}(\text{base}(G)))$.
2. Repeat the following steps:
   {*At this point u is unvisited,* forward (e, G) *is true, and e is the first unvisited edge out of u.*}
   3. While $e \neq \text{base}(G) \wedge \neg\text{forward}(\text{dnext}(e), G)$, do
      {*Here e is the last edge into its destination v, so v has not been visited yet.*}
      4. Set $u \leftarrow \text{dest}(e)$, and set $e \leftarrow \text{sym}(\text{dnext}(e))$.
      {*This sets e to* first out (u).}
   {*Now e is the dummy edge, or its destination has already been visited.*}
   5. If $e \neq \text{base}(G)$, visit e.
   6. While $\neg\text{forward}(\text{onext}(e), G)$ do
      {*Here e has been visited and is* last out (u).}
      7. Visit u.
      8. If $u = \text{dest}(\text{base}(G))$, the algorithm terminates. Else,
         {*Compute* last in (u), *and backtrack through it.*}
         9. Set $e \leftarrow \text{sym}(\text{onext}(e))$.
            While $\text{forward}(\text{dnext}(e), G)$, do $e \leftarrow \text{dnext}(e)$.
            Set $u \leftarrow \text{org}(e)$.
      10. Visit e.
   11. Set $e \leftarrow \text{onext}(e)$.

The equivalence between Algorithms 4 and 5 is straightforward. It is also easy to show that the latter runs in $O(m)$ time for a subdivision with m edges. Every edge of the graph is assigned to e at most twice: once in the enumeration of the edges out of a vertex v (step 11), and once in the determination of last in (v) (step 9).

13. Conclusions and applications. We have introduced a new data structure, the layered dag, which solves the point location problem for a monotone subdivision of the plane in optimal time and space. The main idea has been to refine the chains introduced by Lee and Preparata and connect the refined chains by links. The latter concept originates with Willard [W] and has found frequent application since then. The layered dag can be built from standard subdivision representations in linear time, as follows. We use the graph traversal algorithm of § 12 to enumerate the regions of the subdivision in a way compatible with the vertical ordering presented in § 4. Another traversal of the subdivision then allows us to build the chain tree representing a complete family of separators, as in § 10. Finally, the layered dag is built from the chain tree, as explained in § 9. The point location algorithm using this structure has

been given in § 8. Compared to previous optimal solutions, the advantage of the layered dag is that

- it admits a simple, practical implementation, and
- it can be extended to subdivisions with curved edges.

We will not discuss in detail here how to generalize our method to work for curved-edge subdivisions. Certain requirements for such a generalization to work are clear. We must be able to break up edges into monotone pieces, to introduce the additional edges required by regularization, and to test on what side of (a monotone segment of) an edge a point lies. Our time bounds will be maintained as long as we are able to in constant time:

- cut an edge into monotone pieces,
- add a monotone regularization edge between two existing monotone edges, and
- test if a point p is above or below a monotone edge e.

The layered dag also yields improved solutions for several other problems in computational geometry. All these problems are reduced to the subdivision search problem treated earlier. For example, subdivisions with circular edges occur in the *weighted Voronoi diagram* of a point set [AE]. There, each point p in a finite set U has associated a positive weight $w(p)$ and the region $R(p)=\{x \mid d(x, p)/w(p) \leqq d(x, q)/w(q)$, for all $q \in U\}$. The layered dag offers the first optimal method for locating a point in the diagram defined by these regions.

Finally, certain problems related to windowing a two-dimensional picture given as a collection of line segments have been reduced to subdivision search by Edelsbrunner, Overmars, and Seidel [EO]. The layered dag provides a way to extend their methods to more general curves without losing efficiency.

Acknowledgments. We would like to thank Andrei Broder, Dan Greene, Mary-Claire van Leunen, Greg Nelson, Lyle Ramshaw, and F. Frances Yao, whose comments and suggestions have greatly improved the readability of this paper.

REFERENCES

[AE] F. AURENHAMMER AND H. EDELSBRUNNER, *An optimal algorithm for constructing the weighted Voronoi diagram in the plane*, Pattern Recognition, 17 (1984), pp. 251-257.

[BM] J. L. BENTLEY AND H. A. MAURER, *A note on Euclidean near neighbor searching in the plane*, Inform. Proc. Letters, 8 (1979), pp. 133-136.

[BP] G. BILARDI AND F. P. PREPARATA, *Probabilistic analysis of a new geometric searching technique*, Manuscript, Dept. EECS, Univ. Illinois, Urbana, 1982.

[B] I. C. BRAID, *Notes on a geometric modeller*, C.A.D. Group Document No. 101, Computer Laboratory, Univ. Cambridge, England, July 1979.

[C] R. COLE, *Searching and storing similar lists*, Tech. Rep. 83 New York Univ., New York, 1983.

[C1] B. M. CHAZELLE, *An improved algorithm for the fixed-radius neighbor problem*, Inform. Proc. Letters, 16 (1983), pp. 193-198.

[C2] ———, *How to search in history*, Report CS-83-08, Dept. Computer Science, Brown Univ., Providence, RI, 1983; also Proc. International Symposium on Fundamental Computer Theory, Springer-Verlag, Berlin, 1983.

[C3] ———, *Filtering search: A new approach to query-answering*, Proc. 24th Symposium on the Foundations of Computer Science, 1983, pp. 122-132.

[DL] D. P. DOBKIN AND R. J. LIPTON, *Multidimensional searching problems*, this Journal, 5 (1976), pp. 181-186.

[EH] H. EDELSBRUNNER, G. HARING AND D. HILBERT, *Rectangular point location in d dimensions with applications*, Comput. J., to appear.

[EM] H. EDELSBRUNNER AND H. A. MAURER, *A space-optimal solution of general region location*, Theoret. Comput. Sci., 16 (1981), pp. 329-336.

[EO] H. EDELSBRUNNER, M. H. OVERMARS AND R. SEIDEL, *Some methods of computational geometry applied to computer graphics*, Computer Vision, Graphics and Image Processing, 28 (1984), pp. 92-108.

[GS] L. J. GUIBAS AND J. STOLFI, *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams*, Proc. 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 221-234.

[H] D. HAREL, *A linear time algorithm for the lowest common ancestor problem*, Proc. 21st IEEE Symposium on Foundations of Computer Science, 1980, pp. 308-319.

[Ki] D. G. KIRKPATRICK, *Optimal search in planar subdivisions*, this Journal, 12 (1983), pp. 28-35.

[Kn] D. E. KNUTH, *The Art of Computer Programming, Vol.* 1: *Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1975.

[LP] D. T. LEE AND F. P. PREPARATA, *Location of a point in a planar subdivision and its applications*, this Journal, 6 (1977), pp. 594-606.

[LT] R. J. LIPTON AND R. E. TARJAN, *Applications of a planar separator theorem*, Proc. 18th IEEE Symposium on Foundations of Computer Science, 1977, pp. 162-170.

[MP] D. E. MULLER AND F. P. PREPARATA, *Finding the intersection of two convex polyhedra*, Theoret. Comput. Sci., 7 (1978), pp. 217-236.

[P] F. P. PREPARATA, *A new approach to planar point location*, this Journal, 10 (1981), pp. 473-482.

[PS] F. P. PREPARATA AND K. J. SUPOWIT, *Testing a simple polygon for monotonicity*, Inform. Proc. Lett., 12 (1981), pp. 161-164.

[S] M.I. SHAMOS, *Geometric complexity*, Proc. 7th ACM Symposium on Theory of Computing, 1975, pp. 224-233.

[Ta] R. E. TARJAN, *Depth first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146-160.

[W] D. E. WILLARD. *New data structures for orthogonal queries*, this Journal, 14 (1985), pp. 242-253.