MAINTENANCE OF CONFIGURATIONS IN THE PLANE

Mark H. Overmars and Jan van Leeuwen

RUU-CS-79-9

September 1979/June 1980

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–53 1454
The Netherlands

MAINTENANCE OF CONFIGURATIONS IN THE PLANE

Mark H. Overmars and Jan van Leeuwen

Technical Report RUU-CS-79-9

September 1979/June 1980

Department of Computer Science
University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht
the Netherlands

All correspondence too:

Dr. Jan van Leeuwen

Dept. of Computer Science

University of Utrecht

P.O. Box 80.002

3508 TA Utrecht

the Netherlands

Most results of this paper were announced in a detailed abstract, presented at the 12th Annual ACM Symposium on Theory of Computing [18].

MAINTENANCE OF CONFIGURATIONS IN THE PLANE*

Mark H. Overmars and Jan van Leeuwen

Abstract. For a number of common configurations of points (lines) in the plane, we develop datastructures in which insertions and deletions of points (or lines, respectively) can be processed rapidly without sacrificing much of the efficiency of query answering which known static structures for these configurations permit. As a main result we establish a fully dynamic maintenance algorithm for convex hulls that can process insertions and deletions of single points in only $O(\log^3 n)$ steps or less per transaction, where n is the number of points currently in the set. The algorithm has several intriguing applications, including the fact that the "trimmed" mean of a set of n points in the plane can be determined in only $O(n \log^3 n)$ steps. Likewise, efficient algorithms are obtained for dynamically maintaining the common intersection of a set of half-spaces and for dynamically maintaining the maximal elements of a set of points. The results are all derived by means of one master technique, which is applied repeatedly and which captures an appropriate notion of "decomposability" for configurations.

1. Introduction

Computational geometry (cf. Shamos [23, 25]) concerns itself with the design and analysis of algorithms for dealing with sets of points, lines, polygons and other objects in 2- and higher dimensional space. The sets considered are usually static and the datastructures used are nearly always inadequate for efficiently accommodating insertions and deletions. In this paper we shall attempt to remedy the lack of sufficiently fast dynamic maintenance algorithms for a variety of common configurations in the plane, some of immediate practical interest.

---

*Authors' addresses: Dept. of Computer Science, University of Utrecht, P.O. Box 80.002, 3508 TA Utrecht, the Netherlands.

The problem to convert the intrinsically static datastructures of searching problems into dynamic ones (henceforth referred to as "dynamization") was recently put forward in very general terms by Bentley [3]. He characterized a large class of problems (which he termed "decomposable searching problems") which are particularly amenable to dynamization. In Bentley [3] and in Saxe & Bentley [22] a number of surprisingly powerful techniques were presented, which can be called into action on any decomposable searching problem and which may drastically reduce the update times needed, without the search or query times thereby rising beyond tolerable limits.

While the theory as it stands is applicable to a wide variety of "point problems", Saxe & Bentley [22, appendix] observed already that their techniques were apparently insufficient to handle entire configurations (such as convex hulls) dynamically as well. Yet many of the geometric configurations commonly considered intuitively are "decomposable". We shall prove for a number of different types of geometric configurations that efficient dynamizations can be achieved and identify the concept of decomposability which all these configurations seem to share.

In the sections to follow we shall present efficient algorithms to dynamically maintain the convex hull of a set of points, the common intersection of a collection of halfspaces and the contour of maximal elements of a set of points. The results are often of the sort that insertions and deletions of objects can be performed in only $O(\log^2 n)$ or $O(\log^3 n)$ steps each, where n is the current number of objects in the set. In several instances no better bounds than $O(n)$ or worse were known before, in some the problem to support deletions too has never been discussed before. An extensive list of applications is discussed in various intermittent sections, some of immediate interest to such areas as computational statistics (cf. Shamos [24]). For example, we shall present a method to maintain two sets of points in the plane at a cost of only $O(\log^3 n)$ time for each insertion or deletion, such that the question of whether the two sets are separable by a straight line can be answered in only $O(\log^2 n)$ time.

An interesting feature of the algorithms we present is that they all follow (more or less) by applying one and the same technique, which can be taken as additional evidence that the configurations we consider have a common type of decomposability. Some of the searching problems we consider, such as containment in the common intersection of a set of halfspaces, even are decomposable in Bentley's sense. It will appear that the efficiency of algorithms derived by applying any of the standard dynamizations (as they are known) to the currently best static solutions of these problems does

not even come near the efficiency attained by the especially engineered maintenance algorithms we develop here. On the other hand, we have no proof that the bounds and methods we use are anywhere near optimality and further improvements remain open.

2. Dynamically maintaining a convex hull (prelude)

In the past many different algorithms have been proposed to determine the convex hull of a set of n points $p_1, \ldots, p_n$ in the plane [4, 9, 10, 12, 15, 20]. The algorithms usually operate on a static set and have a worst case running time of $O(n \log n)$ or $O(nh)$, where h is the number of points appearing on the hull.

An early algorithm of Graham [10], for example, operates by locating an interior point S of the convex hull first and ordering all points p_1 to p_n by polar angle around S. In this order the points span the contour of a simple, star-shaped polygon and it only takes a single walk around the polygon to "draw in" the convex hull (figure 1). Since it always has to sort, Graham's

---

INSERT FIGURE 1 ABOUT HERE

---

algorithm will be tied to an $\Omega(n \log n)$ worst case lowerbound. On the other hand, the very fact that we normally want to obtain the ordered contour of the convex hull implies that sorting must be implicit in any algorithm and the $\Omega(n \log n)$ worst case lowerbound applies to all of them which deliver a convex hull in such terms [25]. Even if we merely want to mark which of the p_i are hull-points (duplicates allowed) and don't care about the actual contour at all, then an $\Omega(n \log n)$ worst case lowerbound can still be shown [28], even in a quadratic decision tree model [30].

Nearly all convex hull algorithms known today (like Graham's) require that all inputs are read and stored before any processing can begin. Such algorithms are said to operate "off-line". Shamos [24] apparently first noted that in certain applications one might want to have an efficient "on-line" algorithm instead, which will have the convex hull of p_1 to p_i complete and ready before p_{i+1} is added to the set. Because of the n log n lowerbound, updates of the convex hull due to the addition of a single

point will cost at least $\Omega(\log n)$ on the average. Preparata [19] recently showed an algorithm to insert a point and update the convex hull in a way which never exceeds the $O(\log n)$ even as a worst case bound. Briefly, his technique amounts to the following. Suppose the extreme points among p_1 to p_i are kept ordered by polar angle around an interior point S of the current hull and are stored in a proper, concatenable queue (see [1]). When p_{i+1} is presented we first determine whether it lies inside or outside the current hull, by inspecting the sector it belongs to (which can be found by binary search, see figure 2a). When p_{i+1} lies in the interior no update is needed. When p_{i+1} lies in the exterior (see figure 2b), determine the tangents $\overline{xp}_{i+1}$ and $\overline{yp}_{i+1}$ to the

---

INSERT FIGURE 2 ABOUT HERE

---

current hull, omit the points on the arc between x and y "illuminated" by p_{i+1} and insert p_{i+1} for them instead. The non-easy part concerns the design of a proper queue structure (a geared-up AVL-tree will do), such that binary search on the hull can be performed in only $O(\log n)$ steps in worst case (e.g. to find the tangents needed) in addition to the ordinary $O(\log n)$ insertion, deletion and splitting behaviour.

It is clear that none of the previous algorithms are fully dynamic, since at best they support insertions only. Yet there are a number of practical problems (cf. section 5) in which it is required to have an efficient algorithm to restore the convex hull when points are deleted from the set. This creates a tremendous problem for all existing algorithms, even for Preparata's [19]. They virtually all go by the principle that points found to be in the interior of the (current) convex hull will not be needed ever and can be thrown away, and some are even especially designed to eliminate as many points from further consideration as they can to cut down on the ultimate running time. This can no longer be maintained if we allow deletions to occur. It is most easily demonstrated by the fact that, when an extreme point of the current convex hull is deleted, the hull can "snap back" (see figure 3) and tighten itself around some old points of the interior ... which suddenly find themselves to be part of the new convex hull! Observe also (figure 3) that the number

---

INSERT FIGURE 3 ABOUT HERE

---

of points added to the hull after deleting a point can be rather large. We will show that, despite these apparent complications, the set of n points can be structured and its convex hull maintained at a cost of only $O(\log^3 n)$ or less for each insertion and deletion. The time required for insertions can even be kept within an $O(\log^2 n)$ bound, by a judicious choice of datastructures.

3. Dynamically maintaining a convex hull (representation)

Given the task to maintain it dynamically, an immediate problem is how to actually represent the convex hull of a set. The usual way to keep points ordered "around a fixed interior point S" is no longer feasible, because repeated insertions and deletions can cause the set to wander off and put S in its exterior. It is avoided by adopting a new representation of the convex hull (see figure 4), consisting of its separate left and right faces. Thus, the convex hull is represented by

---

INSERT FIGURE 4 ABOUT HERE

---

means of two very special, convex arcs.

Let P be a set of points in the plane, let $\infty_L=(-\infty,0)$ and $\infty_R=(+\infty,0)$.

Definition. The lc-hull of P is the convex hull of $P \cup \{\infty_R\}$, the rc-hull of P is the convex hull of $P \cup \{\infty_L\}$.

The lc- and rc-hull of a set are illustrated in figure 5a and 5b,

---

INSERT FIGURE 5 ABOUT HERE

---

respectively. We will concentrate on the lc-hull of a set, as its rc-hull is treated in completely the same way. Note that the lc-hull is a convex arc which begins at the rightmost point of highest y-coordinate and ends at the rightmost point of lowest y-coordinate and tightly bounds the set from the left. Points along the lc-hull appear in sorted order by y-coordinate! It will be necessary for later purposes to store the points along the lc-hull in this order (i.e., by ordered y-coordinates) in a concatenable queue Q_L (figure 6). The

---

INSERT FIGURE 6 ABOUT HERE

---

contour of the rc-hull is stored likewise in a concatenable queue Q_R. We want Q_L and Q_R to be balanced search trees.

Lemma 3.1. Given the lc- and rc-hull of a set of n points, one can determine whether an arbitrary point p lies inside, outside or on the convex hull in only O(log n) steps.

Proof

We will only consider the question whether p lies inside, outside or on the lc-hull. From this and the response to the same query w.r.t. the rc-hull the required answer can be derived immediately. Let $p = (x_p, y_p)$. By means of an O(log n) search down Q_L one can determine two consecutive hull-points p_i and p_j such that $y_{p_i} \leq y_p \leq y_{p_j}$. If no two such points exists, then p lies above or below the lc-hull. Otherwise (see figure 7) it only takes a trivial

---

INSERT FIGURE 7 ABOUT HERE

---

test to determine where p is located w.r.t. the lc-hull.

□

The lc-hull (and likewise the rc-hull) of a set P is a decomposable configuration in the following sense. Split P (with its points ordered by y-coordinate) by a horizontal line into two parts A and C, as in figure 8. The lc-hull of P is composed of portions of the lc-hulls of A and C, and a bridge B connecting the two parts.

INSERT FIGURE 8 ABOUT HERE

The following result is crucial for much of the entire construction and shows that, once the representation of the lc-hulls of A and C is known, the representation of the lc-hull of $P = A \cup C$ can be determined in an efficient (but tedious) manner. In the proof we shall encounter some specific requirements on the Q-structures, very similar to Preparata's [19].

Theorem 3.2. Let $p_1, \ldots, p_n$ be n arbitrary points in the plane, ordered by y-coordinate. If the representations of the lc-hull of $p_1, \ldots, p_i$ and of $p_{i+1}, \ldots, p_n$ are known (any $1 \leq i < n$), then the lc-hull of the entire set can be built in $O(\log^2 n)$ steps.

Proof

Let $P = \{p_1, \ldots, p_n\}$. Think of $\{p_1, \ldots, p_i\}$ as A and of $\{p_{i+1}, \ldots, p_n\}$ as C (ref. figure 8). Let the lc-hulls of A and C be given in terms of concatenable queues Q_A and Q_C respectively, representing the ordered contours as suggested in figure 9. Since p_1 to p_n are sorted

---

INSERT FIGURE 9 ABOUT HERE

---

by y-coordinate, the sets A and C indeed are separated by a horizontal line and, to find the lc-hull of P, all we have to do is to determine the bridge B. For, let the bridge (which is the common tangent of A and C) "touch" A at u and C at d. Then we can build Q_L (the representation of P's lc-hull)

as follows: split Q_A at u (u included in the "first" part), split Q_C at d (d included in the "last" part) and concatenate the first part of Q_A and the last part of Q_C. (Hence u and d have now become consecutive, correctly representing the joining edge.) It is clear that this construction takes only O(log n) steps by the usual results for concatenable queues .... provided we know what u and d are.

Efficient tangent determination (cf. Preparata [19]) requires that one can perform binary search on the lc-hulls. Ordinarily concatenable queues do not permit one to do so, because they provide no random access to their leaves. It would require O(log n) search down the Q-structure for every next point to be visited and a total of $O(\log^2 n)$ steps to do binary search. If we relax our searching policy only slightly, then the undesirable overhead in "halving" search segments can be avoided. Let us augment each node of a Q-structure with pointers to its descendants with highest and smallest y-coordinate, respectively (see figure 10a). It is easily verified that the usual concatenable queue structures (AVL-trees, 2-3 trees, BB[α]-trees) have update, split and concatenation routines in which this sort of information can be maintained at no significant extra cost. A "binary search" now merely descends down a path

---

INSERT FIGURE 10 ABOUT HERE

---

of the tree and whenever a node is reached representing a search segment [p, r] (on an lc-hull, see figure 10b), then we only need to inspect the two inner leaves (q_1 and q_2 in figure 10b) pointed to by its sons to determine on what segment ([p, q_1] or [q_2, r]) the search must be continued. We shall assume from now on that Q_A and Q_C and all later Q-structures are augmented with the extra pointers at each node as indicated.

Let p be an arbitrary point of A. (Important is only the fact that p lies above C.) The one-and-only tangent to C's lc-hull can be determined in O(log n) steps by "binary search" down Q_C in the following way. Let t_p be the point to be found on C's lc-hull where the tangent through p "touches" it. All we need is a simple criterion to tell in what direction t_p is located whenever the search leads us to inspect yet another point q of the lc-hull of C. Only a limited number of cases can occur, when we consider how the line $\overline{pq}$ intersects with the contour of the lc-hull:

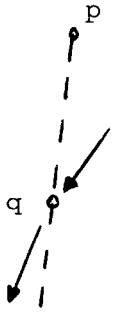
case (i)

In this situation, q is the point we searched for and we can stop

case (ii)

In this situation, q is located past t_p and the search must continue on the first part of the segment considered

case (iii)

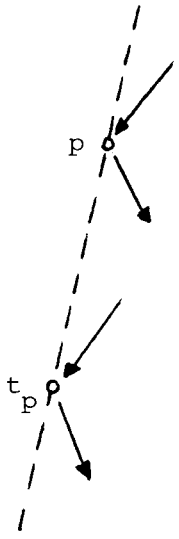


In this situation, q is located before t_p and the search must continue on the second (last) part of the segment considered

All these cases can be distinguished at only O(1) cost. Hence the search down Q_C can be performed and guided at a cost of only O(1) total for each node visited and will correctly turn up t_p in O(log n) steps at most. This routine for tangent determination will be used in the final part of our proof, now to come.

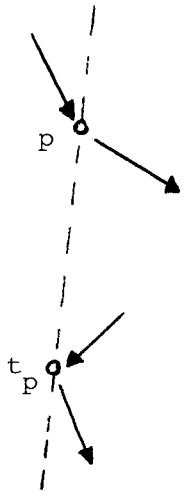
The bridge B is easily recognized as the one common tangent of the lc-hulls of A and C. Observe that d = t_u and the routine for tangent determination can be used to find d (hence B) ... provided we know the location of u. We will show how u can be located by "binary search" on A's lc-hull, in one search down the proper path in Q_A. To guide the search at each node and to decide on what subsegment the search must continue, all we need is an easy criterion to tell in what direction (backward or forward) u is located whenever we probe at a next point p on A's lc-hull. We can tell by inspecting the line $\overline{pt_p}$, the tangent through p to C's lc-hull (which can be determined by the routine sketched earlier). Only a limited number of cases can occur, when we consider how this tangent intersects with A's lc-hull:

case (i)



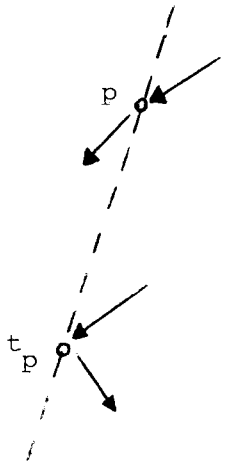
In this situation, p is the point u we searched for and we can stop

case (ii)



In this situation, p must be past u and the search must continue on the first part of the segment considered

case (iii)



In this situation, p is located before u and the search must continue on the second "half" of the segment considered

These cases can all be easily distinguished. Hence each step of the search down Q_A really takes only $O(\log n)$, the amount of work to determine the tangent needed to decide on what sub-segment to continue the search for u. In this way, u is found after at most $O(\log^2 n)$ steps total.

Once u is known we get B in only O(log n) steps and the construction of Q_L can proceed as indicated in another O(log n) steps.

□

Theorem 3.2. suggests an interesting algorithm to construct the lc- and rc-hulls, hence the entire convex hull, of a static set of n points in the plane. Let us assume for simplicity that $n = 2^k$ for some k. First sort the points by y-coordinate in O(n log n) steps. Next, for i from 1 to k, repeatedly determine the lc- and rc-hulls of horizontally separated groups of 2^i points each by "composition" (as suggested in 3.2.) from the lc- and rc-hulls of their constituent, and likewise horizontally separated halves of 2^{i-1} points (which were just constructed at the previous iteration). The number of steps needed to build the hulls amounts to about

$$n + \frac{n}{2}\log^2 2 + \frac{n}{4}\log^2 4 + \ldots = \sum_{i=1}^{k} \frac{n}{2^i}\log^2 2^i = O(n)$$

and the composition of the lc- and rc-hull to obtain the complete convex hull is a near trivial matter afterwards.

Corollary 3.3. The convex hull of a static set of n points in the plane can be found in only O(n) steps after all points have been sorted by y-coordinate.

We note that the given algorithm for convex hull determination is similar in many ways to one of Preparata & Hong [20], although the latter still requires O(n log n) steps after the initial sorting to complete.

We have no indication that the algorithm of theorem 3.2. is best possible and it is conceivable that the $O(\log^2 n)$ bound can be improved. The actual bound on "bridge" determination will be crucial in the analysis of later algorithms, a sufficient reason to symbolize the best run-time possible by a special function.

Definition. Let $J_1(n)$ be the best run-time achievable by any algorithm that finds the one common tangent of two horizontally separated lc-hulls of n points (represented as concatenable queues).

We shall assume that $J_1(n) = \Omega(\log n)$.

4. Dynamically maintaining a convex hull (structure and algorithms)

From now on we shall assume that the convex hull of a set of points in the plane is represented by the junction of its lc- and rc-hull. It will appear that the lc-hull of a set (and likewise, its rc-hull) is easier to maintain dynamically than the convex hull itself is directly. Yet the results derived for lc-hull maintenance will hold ipso facto for the convex hull as well.

As we must accommodate both insertions and deletions, it is conceivable that some information must be maintained about the arrangement of the points currently in the interior of the lc-hull of the set.

Let the points of the set be sorted by y-coordinate and let they be stored by this attribute in a binary search tree T. We usually assume that no two points have the same y-coordinate, but it is in no way essential for the constructions to follow. It is natural to augment T and to associate with each node α a concatenable queue Q_α representing the lc-hull of the set of points stored at the leaves of its subtree. By theorem 3.2. one can obtain Q_α from the structures Q_γ and Q_δ associated with the sons γ and δ of α (see figure 11) in only $O(J_1(n))$ steps, but there is a slight complication as far as the efficiency is concerned. Observe that Q_γ and Q_δ must be split to yield the pieces for Q_α and that they are "destroyed" for further use if we do so. If we want to build Q_α from Q_γ and Q_δ and retain Q_γ and Q_δ as they are, then we would have to spend much more than $J_1(n)$ time just to copy the segments of Q_γ and Q_δ which need to be joined to form Q_α. Fortunately, as can be seen from

---

INSERT FIGURE 11 ABOUT HERE

---

figure 8 and is suggested by figure 11, Q_α is built in a very regular way and is obtained by concatenating the proper head segment of Q_γ and tail segment of Q_δ with the "bridge" in between. It is clear that we might as well cut the required segments off from Q_γ and Q_δ and pass them on to α, leaving γ and δ with only a fragment of their original associated structure. If we remember at node α where the bridge connecting the two segments was put when we built Q_α, then we only have to split it at this very spot to obtain the two "pieces" again and concatenate them to the left-over pieces at γ and δ to fully reconstruct Q_γ and Q_δ.

In the structure so obtained we can go down in the tree and reassemble the Q-structures at the nodes bordering a path from the pieces reclaimed by the continued splitting of the Q_γ or Q_δ on our way down, and later climb back along the same path, meanwhile rebuilding the Q_α-structure for each node α visited and passing on the part we need as we proceed to its father. Going down can be done rather fast and only requires a few O(log n) routines for splitting and (re)concatening Q-structures per node visited, but going up normally requires $J_1(n)$ steps per node (unless old information can be used). We shall see how this intriguing structure functions below.

As it stands we have obtained an intriguing augmented search tree structure T^*, in which with each interior node α is associated the fragment Q^*_α of the lc-hull of the set of points it covers that was not used in building the lc-hull of its father. The lc-hull of the entire set will normally be available at the root, as this characterization implies.

Proposition 4.1. After sorting points by y-coordinate (i.e., after building T), the augmented tree T^* can be obtained in only O(n) additional steps.

Proof

This follows essentially from the argument given to prove corollary 3.3. The amount of work to construct the information at any of the $n/2^i$ nodes in the i^{th} level from below of T is still bounded by $O(\log^2 2^i)$, as the cost for bridge determination is dominant over the costs for splitting and concatenating the information needed from their sons.

□

We will show that T^* can be maintained efficiently at all times. Let the following information be associated with each internal node α:

(i) $f(\alpha)$ = a pointer to the father of α (if any),

(ii) $lson(\alpha)$ = a pointer to the left son of α,

(iii) $rson(\alpha)$ = a pointer to the right son of α,

(iv) $max(\alpha)$ = the largest y-value in the subtree of $lson(\alpha)$,

(v) $Q^*(\alpha)$ = the segment of Q_α (head or tail) which did not contribute to $Q_{f(\alpha)}$,

(vi) $B(\alpha)$ = the number of points on the segment of Q_α (tail or head) which does belong to $Q_{f(\alpha)}$

Clearly (i) to (iv) are needed to let T* function as a search tree, (v) is the "piece" of Q_α left after sending the other half up to $f(\alpha)$ and (vi) enables us to reconstruct the position of the bridge used in building $Q_{f(\alpha)}$ from its "left" and "right" components.

Notation. For a concatenable queue Q, let Q[k ...l] denote the concatenable queue consisting of the k^{th} up to l^{th} elements of Q. For concatenable queues Q_1 and Q_2 of horizontally separated sets of points, let $Q_1 \cup Q_2$ denote their concatenation as a single queue.

For queues Q, Q_1 and Q_2 of O(n) elements each, the queues Q[k ...l] and $Q_1 \cup Q_2$ (when defined) can be obtained in only O(log n) steps when properly implemented (cf. [1]) ... although the original queues may be destroyed when we build them.

Given the search structure T* for a set of points (with the complete lc-hull of the set at the root), we shall first devise an important routine (DOWN) to reconstruct the full Q_β at an arbitrary node β where it is needed. There will be some additional sidebenefits from DOWN as well, as will soon be apparent. The construction begins at the root and descends down the search path towards β node after node, meanwhile disassembling the full Q-structure just build (or rather, reconstruct) at a father and reassembling the complete Q-structure at its two sons before continuing in a particular direction. Later β will be the father of a (suspected) leaf and the search for it will be guided by the usual decision criterion (involving max) in binary search trees. We omit this detail from the specification of DOWN given here.

```
procedure DOWN(α, β);
   {α is the internal node which was just reached in the
    search towards β. Q*(α) contains the complete lc-hull
    of the set of points covered.}
   begin
      if α = β then goal reached
               else
```

```
        begin
          {We split Q*(α) and reconstruct the Q-
           structures at its two sons}
          {Cut Q*(α) at the bridge ... }
           Q1 := Q*(α) [1 .. B(lson(α))];
           Q2 := Q*(α) [B(lson(α)) + 1 .. *];
          {... and glue the pieces back onto the queues
           left at the two sons}
           Q*(lson(α)) := Q*(lson(α)) ∪ Q1;
           Q*(rson(α)) := Q2 ∪ Q*(rson(α));
          {Continue the search in the right direction}
           if β below lson(α)
                    then
                       DOWN(lson(α), β)
                    else
                       DOWN(rson(α), β)
        end
end of DOWN;
```

Note the precise order in which the pieces of $Q^*(\alpha)$ are glued onto the queues at the sons of α. The routine is called as DOWN(root, β). Let T^* currently have n leaves (i.e. # P = n).

Lemma 4.2. DOWN always reaches its goal after $O(\log^2 n)$ steps.

Proof

Since T is balanced, no node β can be deeper than O(log n). It follows that DOWN will visit at most O(log n) nodes α on its way, no matter what β is. The amount of work DOWN spends at each node is certainly bounded by O(log n) per node, as it only involves some standard operations for concatenable queues of size O(n) at the node.

□

In addition to Q_β, the call of DOWN(root, β) produces the full Q-structure (and thus the complete lc-hull of all points below it) at each node α whose father is on the search path towards β but which isn't on it itself (see figure 12). These full structures are kept for later use, the Q^*-fields of nodes on the search path itself (except β) have temporarily become vacuous.

INSERT FIGURE 12 ABOUT HERE

DOWN will normally be called because we want to update the set of points below β and thus ... the lc-hull Q_β at this node. After having done so we can climb back up the search tree again node after node, each time reassembling the (new) lc-hull at a next higher node by taking pieces from the Q-structure at its sons in a way which should now be familiar. The necessary Q-structures are available, at one son (the one on the search path) because we just built it and at the other son because DOWN conveniently put it there (and left it there) on its way to β. There is just one catch to this all. Because we updated the set below β, presumably by inserting or deleting a point, the tree T^* may have gotten out of balance. We shall see later that there is a way to perform local rebalancings in T^* efficiently, despite the fact that the associated structures at the nodes involved in a rebalancing may have to be redistributed completely. We delegate the task to a routine BALANCE. The procedure UP given below will be the counterpart to DOWN. It starts at β and gradually works its way up, restoring both the Q^*-structures and the balance of the tree along the search path.

```
procedure UP(α);
  {α is the node most recently reached on the way back to
   the root. Q*(lson(α)) and Q*(rson(α)) contain the complete
   lc-hulls of the sets below lson(α) and rson(α), respec-
   tively.}
begin
   determine the bridge connecting Q*(lson(α)) and
   Q*(rson(α)) and thus the numbers of points B1 and
   B2 which they must each contribute into Q*(α);
   {record these numbers}
   B(lson(α)) := B1;
   B(rson(α)) := B2;
```

```
        {Cut the necessary pieces off from the queues ...}
        Q1 := Q*(lson(α)) [1 .. B1];
        Q2 := Q*(rson(α)) [*-B2 .. *];
        {effectively leaving the remaining parts at the sons}
        {... and put them together to form the lc-hull of
        the joint set}
        Q*(α) := Q1 ∪ Q2;
        if out of balance then BALANCE(α);
        if α = root then goal reached else UP(f(α))
    end of UP;
```

Note what pieces from $Q^*(lson(\alpha))$ and $Q^*(rson(\alpha))$ together form $Q^*(\alpha)$. After the subtree below β has been updated (and balanced, if necessary), the given routine is called as UP(f(β)) ... provided β wasn't the root already.

Lemma 4.3. UP always reaches its goal after $O(\log n.J_1(n) + R)$ steps, where R is the cost of all rebalancings required along the search path during the particular action.

Proof

Starting at any β in a balanced tree, UP will need to visit no more than O(log n) nodes before it terminates at the root. At each node visited, UP spends $J_1(n)$ steps finding the bridge it needs and another O(log n) steps to perform some standard operations on concatenable queues. The costs for rebalancing T^* as we go up along the search path add up to R by definition.

□

To get an impression of R, we shall delve into the necessary actions for rebalancing a single node α. It is not obvious that one can always rebalance T and restore the associated information at the nodes, without the need for costly restructuring operations. We shall restrict ourselves to familiar types of balanced trees like AVL-trees and BB[α]-trees (see e.g. [1, 21]), which can be rebalanced by means of local rotations. Let us examine the case in which a single rotation must be carried out at a node α (see figure 13). The case in which a double rotation must be carried out is very similar and will not be discussed in detail. The necessary actions at node α are initiated by the procedure BALANCE, referred to in UP.

INSERT FIGURE 13 ABOUT HERE

BALANCE is called just after Q_α was reconstructed. It appears that we have to undo this step, using one iteration of DOWN, to obtain the complete $Q_{lson(\alpha)}$ and $Q_{rson(\alpha)}$ again and prepare for a different construction of the same Q_α. It follows that we better decide the need to rebalance at α before we construct Q_α, i.e., at the beginning of UP instead of at the end. We leave this modification for the reader to implement.

Lemma 4.4. Each call of BALANCE requires only $O(\log n + J_1(n))$ steps.

Proof

Referring to figure 13, let the sons of $lson(\alpha)$ be β and γ. Given $Q_{lson(\alpha)}$, we can reconstruct the complete Q_β and Q_γ in just $O(\log n)$ steps by performing one iteration of DOWN. Let δ be the new "right son" of α as a result of the rotation. Observing that the complete Q-structures are available at β, γ and (the old) $rson(\alpha)$, it is clear that we can restore the proper information at the nodes involved and climb back to α (where we were) by restarting UP at node δ. It follows that a single rotation can be carried out at the expense of at most $O(\log n + J_1(n))$ extra steps. The analysis for double rotations proceeds in very much the same way and yields the same estimate.

□

We now have all ingredients available to prove a first version of our result on convex hull maintenance.

Theorem 4.5. The convex hull of a set of n points in the plane can be maintained at a cost of $O(\log^2 n + \log n.J_1(n))$ per insertion and deletion.

Proof

Using T^* as the underlying datastructure, we would proceed as follows to insert or delete a point p. Remember that we have to update both the lc- and the rc-hull of the set. We shall only describe the necessary actions for the lc-hull.

First we search down T*, using p's y-coordinate, to find out in what leaf p is (or must be) stored. We do so by means of the procedure DOWN, which at the same time will restore the complete lc-hulls at all nodes directly bordering the search path towards p at a cost of $O(\log^2 n)$. After p is inserted or deleted as a leaf at the bottom of the tree, we must climb back to rebalance the tree in accordance with the normal routines for the type of balanced tree chosen and to reconfigure (update) the associated information at all nodes on the search path. This we do by means of the procedure UP, which takes care of any rebalancings required and repeats putting a new Q-structure together at a node and cutting it again to build the new Q-structure at the next higher node, until the root is reached. By lemmas 4.3. and 4.4., UP takes $O(\log n.J_1(n))$ in basic costs and an additional $O(\log n + J_1(n))$ for each rebalancing required. Since the number of rebalancings will not exceed $O(\log n)$, the total time required to execute UP is certainly bounded by $O(\log^2 n + \log n.J_1(n))$.

□

As $J_1(n) \geq \log n$, theorem 4.5. essentially tells us that updates of the convex hull take $O(\log n.J_1(n))$. Since $J_1(n) = O(\log^2 n)$, it follows that the convex hull of a set of n points in the plane can be maintained at a cost of only $O(\log^3 n)$ per insertion and deletion. We can improve the result somewhat, by more carefully examining the cost of UP's actions after an insertion. It appears that, as the result of an insertion, the location of the bridge between the two constituent halves of the lc-hull (or ... the rc-hull) at a node α on the search path cannot shift too drastically. Rather than spending a full $O(J_1(n))$ for bridge construction using theorem 3.2., we shall employ a simpler method which takes only $O(\log n)$ to limit the overall costs.

Theorem 4.5*. The convex hull of a set of n points in the plane can be maintained at a cost of $O(\log^2 n + \log n.J_1(n))$ per deletion and a cost of $O(\log^2 n + r.J_1(n))$ per insertion, where r is the number of rebalancings required in performing the insertion.

Proof

The global action of DOWN and UP is left unchanged. In particular, the time bound for processing a deletion is left what it was. Only when an insertion has taken place, UP will call on a different technique to determine bridges as it climbs up the search path. Remember that when DOWN passed a

node α, it reconstructed the complete Q-structures at lson(α) and rson(α). Let the set of points below lson(α) be C and below rson(α) be A, and let the bridge be the line-segment (tangent) connecting u $\in$ A and d $\in$ C. See figure 14. We shall assume that, when DOWN passes α and splits Q_α, it

---

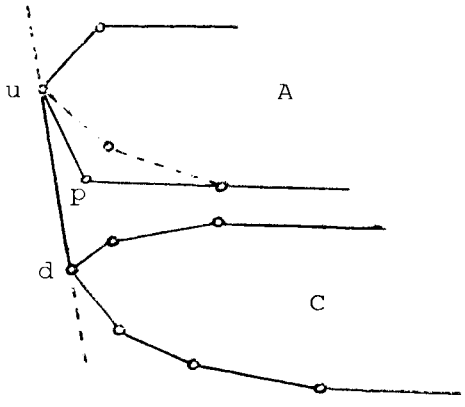
INSERT FIGURE 14 ABOUT HERE

---

stores the current location of the bridge (i.e. the nodes d and u) for later reference.

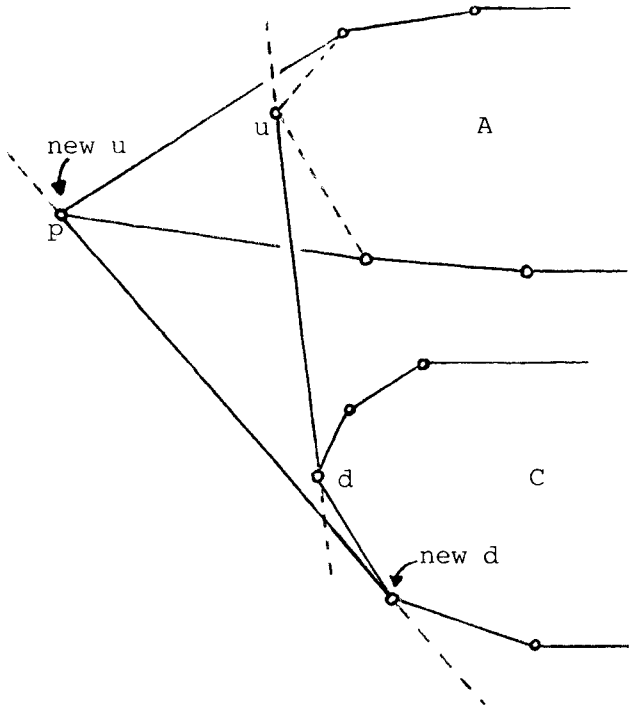
Suppose that the insertion of a new point p took place in the subtree of rson(α), i.e., in the set A. Imagine that UP is coming back and has just completed building a new $Q_{rson(\alpha)}$. Its first action at α will be to determine the (new) bridge between $Q_{lson(\alpha)}$ and the updated $Q_{rson(\alpha)}$. It is at this stage that we recall d and u. When p has not become part of $Q_{rson(\alpha)}$'s contour, then $Q_{rson(\alpha)}$ ... and hence the bridge ... has in fact remained entirely the same as it was. When p has, it must have been exterior to the old $Q_{rson(\alpha)}$ and the new contour is obtained by taking the tangents through p to the old contour and the continuation of the old contour from the two tangent points onward. To find the new bridge between $Q_{lson(\alpha)}$ and $Q_{rson(\alpha)}$, only three different situations can arise.

Case(i): p is completely to the right of $\overline{ud}$.



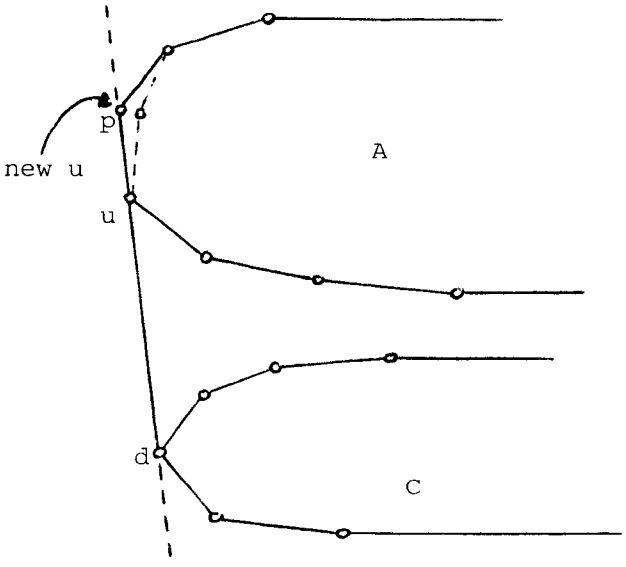
In this situation u has remained part of A's lc-hull and $\overline{ud}$ still is the common tangent of $Q_{lson(\alpha)}$ and $Q_{rson(\alpha)}$. To determine the relative position of u on the new contour of $Q_{rson(\alpha)}$ will take an easy O(log n), but nothing further needs to be done.

Case(ii): p lies completely to the left of $\overline{ud}$.



In this situation, obviously u can no no longer be the right tangent point. The other points from the old contour cannot serve as tangent point for the bridge either, otherwise they would have served for it even before p got inserted (contradicting the status of u). It follows that the new bridge must necessarily touch $Q_{rson(\alpha)}$ at p. The bridge itself is now completely determined as the tangent through p to $Q_{lson(\alpha)}$ which, by the technique explained in the proof of 3.2., requires only O(log n) steps to compute.

Case(iii): p lies on $\overline{ud}$.



In this situation the bridge will remain the same "line", but its tangent point on Q_{rson} may change. If p does not lie between u and d, then it will become the "new u" and $\overline{pd}$ is the new bridge. If p lies between u and d then, by the ordinary convention for convex hulls, $\overline{ud}$ remains the bridge between the two lc-hulls.

It follows that in all cases finding the new bridge takes at most O(log n) steps, at each node UP encounters on its way towards the root after an insertion. The remaining basic actions UP performs at each node require no more than O(log n) also, which means that the total costs of all basic actions UP performs along the search path add up to a mere $O(\log^2 n)$. When BALANCE calls for a rotation at a node, we follow its original actions to "reshuffle" a few lc-hulls immediately below it at a cost of $O(\log n + J_1(n))$. After a rotation is carried out, UP can continue the way we described. Hence we only need to add $O(r.\log n + r.J_1(n)) = O(\log^2 n + r.J_1(n))$ in extra costs, where

r is the total number of rebalancings required. Thus insertions can be processed in $O(\log^2 n + r.J_1(n))$ steps total.

□

We conclude

Theorem 4.6. The convex hull of a set of n points in the plane can be maintained at a cost of $O(\log^2 n)$ per insertion and a cost of $O(\log^3 n)$ per deletion.

Proof

Using that $J_1(n) = O(\log^2 n)$, the time bound for deletions is immediate from theorem 4.5*. Let us choose to represent T as an AVL-tree. It is well-known that processing an insertion in an AVL-tree will lead to at most one rebalancing along the search path (see e.g. [21] p.243 or [29] p.226). It follows from 4.5*. that insertions will cost no more than $O(\log^2 n)$ steps in this case.

□

Note that theorem 4.6. has become rather dependent on the type of balanced tree we use for T. When it can be shown that $J_1(n) = O(\log n)$, theorem 4.6. can be improved to read that both insertions and deletions of points can be processed in $O(\log^2 n)$ steps, regardless of the type of tree chosen (provided it can be kept in balance by means of local rotations). An improvement of this sort will change many $O(n \log^3 n)$ bounds in the next section into $O(n \log^2 n)$ bounds.

5. Applications of the dynamic convex hull algorithm.

There are numerous problems in computational geometry and more applied fields, which can be solved by using convex hull determination as a tool (cf. Shamos [24]). The algorithm we devised for dynamically maintaining a convex hull in the plane will enable us to tackle a few inherently dynamic problems, for which good bounds were lacking until now.

In statistics considerable attention has been given to finding estimators which identify the center of a population. For 1-dim data it has given rise to the concept of an "α-trimmed mean", obtained by taking the mean value of the points remaining after discarding the upper- and lower α-tiles of

the set. (See Huber [14] for a historical account of the concept.) Since the α-tiles can be determined in only $O(n)$ time no matter how the set of n points is given (Blum et.al. [6]), the trimmed mean follows in only $O(n)$ steps all together. In 2 dimensions a similar idea has given rise to the concept of "peeling" a convex hull (Tukey [27]), again to remove some fixed percentage of outlying points from the set. Each time a point is removed, the convex hull must be updated accordingly. Green [11] has indicated what statistical information can be obtained through peeling in 2- and more dimensions, but the computational complexity of it definitely is no longer linear.

Shamos [25] reported an $O(n^2)$ algorithm for peeling a set of n points in the plane, based on an iterated version of Jarvis' convex hull algorithm (Jarvis [15]). Green and Silverman [12] gave an algorithm to peel a set using Eddy's convex hull algorithm (Eddy [9]), that isn't any better in worst case but seems to perform well in practice. Shamos [25] argued that any algorithm for peeling a set must take $\Omega(n \log n)$ steps in worst case, but he gave it as an open problem to actually beat the existing $O(n^2)$ algorithms. We can apply theorem 4.6. to show

Theorem 5.1. One can peel a set of n points in the plane in only $O(n \log^3 n)$ steps.

Proof

Given a set of n points, first build the data structure T^* for the entire set as described in Section 4. By proposition 4.1. this can be done in only $O(n \log n)$ steps total. Next one can do any n deletions one likes, at a cost of $O(\log^3 n)$ steps per deletion. Hence the peeling of the set can be completed within $O(n \log^3 n)$ steps. (It is noted that this does not take any time into account that may be required to decide what point to peel off next.)

A closely related problem concerns finding the convex layers of a set of points in the plane. Starting with the convex hull as the 1^{st} layer, the i^{th} layer is defined as the convex hull of the set of points remaining after peeling all previous layers off (see figure 15). The statistical significance

---

INSERT FIGURE 15 ABOUT HERE

---

was recognized by Barnett [2], who defined the c-order of a point as being the rank-number of the convex layer to which it belongs. Intuitively, points of low rank correspond to extreme observations that should be treated separately or even be discarded (cf. Huber [14]). Points of highest rank can be viewed as medians of the set.

Shamos [25] argued once again that determining the c-order of all points (which he called their "depth") requires $\Omega(n \log n)$ steps in worst case, but only had his $O(n^2)$ algorithm for peeling to determine these values. We can show

Theorem 5.2. One can determine the joint convex layers of a set of n points in the plane (hence Barnett's c-order groups) in only $O(n \log^3 n)$ steps.

Proof

Assume that the i^{th} convex layer has c_i points, with $\sum_{i \geq 1} c_i = n$. We begin by building the structure T^* as described in Section 4 (viz. proposition 4.1.) at a total cost of $O(n \log n)$. It immediately yields the first convex layer of the set, its convex hull, at the root of the structure. In general the concatenable queue Q associated with the root will contain the representation of the i^{th} convex layer, for some $i \geq 1$. It will take only $O(c_i)$ time to traverse Q and to list which points constitute the current layer. To obtain the next layer, delete each of the c_i points of the current layer from the set. It will cost $O(c_i \log^3 n)$ steps. The total time needed to "peel" off all convex layers will thus be in the order of

$$n \log n + \sum_{i \geq 1} c_i + \sum_{i \geq 1} c_i \log^3 n$$

which is $O(n \log^3 n)$.

□

Note that the convex layers can actually be output in the form of internally linked data-structures, just like any convex hull representation. This will be handy for the next observation (see figure 16).

Given the convex layers of a set, one may traverse the points in clockwise order layer after layer, beginning with the outer layer and each time using a "forward" tangent to step over onto the next inner layer.

INSERT FIGURE 16 ABOUT HERE

The path so obtained (a "spiral") connects all points of the set, does not intersect itself and has the property that all corners in traversal order are convex. As the required tangents can be determined in only O(log n) steps each (cf. Shamos [25]), the following result is immediate.

Theorem 5.3. Given n points in the plane, one can determine a connecting spiral in only $O(n \log^3 n)$ steps.

If meaningful at all, spirals give a systematic enumeration of the points of a population by "significance". Spirals are by no means unique, but are completely determined by the starting point on the outermost convex layer (i.e., the convex hull) and their "direction".

Returning to convex hulls, we can now apply theorem 4.6. to answer a basic question posed in Saxe and Bentley [22]. It concerns a dynamic variant of the simplest type of convex hull searching ("does x belong to the interior of the convex hull of F"), which they left open.

Theorem 5.4. One can maintain a set F of n points in the plane at a cost of $O(\log^2 n)$ time per insertion and of $O(\log^3 n)$ time per deletion, such that queries of the form "does x belong to the interior of the current convex hull of F" can still be answered in O(log n) time.

Proof

It is immediate from theorem 4.6. The concatenable queue available at the root of the data-structure is a full-fledged representation of the convex hull at all times, and by lemma 3.1. queries of the form stated can be answered in O(log n) time whenever needed.

□

A last and intriguing application of dynamic hull maintenance relates to the separability of discrete pointsets in the plane (see e.g. Shamos [23]). Two sets are said to be separable if one can draw a straight line such that one set is entirely to its left, the other one entirely to its right. It is well-known that two sets are separable if and only if their convex hulls

are disjoint. See Hadwiger and Debrunner ([13], sect. 3) for some classical facts concerning separability of sets. Efficient algorithms for deciding static separability would compute the convex hulls of the two sets and see if they are disjoint. Unfortunately the best previously known algorithms for deciding whether two convex k-gons are disjoint do not sufficiently take advantage of any preprocessing and run in O(k) steps, which is too much in a dynamic environment when k is large. But we can show

Proposition 5.5. With suitable preprocessing, one can determine whether two convex k-gons in the plane are disjoint or not in only $O(\log^2 k)$ steps.

Proof

We shall assume that convex k-gons are preprocessed into a concatenable queue which allows for "binary search along the contour", such that tangents and line-intersections can be determined in O(log k) steps (as in Shamos [25], also Preparata [19]).

Consider two convex k-gons A and B. By spending at most log k steps, one can search down A's representation and determine two points a_1 and a_2 on the contour such that the arcs $\overset{\frown}{a_1a_2}$ and $\overset{\frown}{a_2a_1}$ have $\frac{k}{2}$ points each. Draw the line $\overline{a_1a_2}$ (which effectively cuts A in half) and determine the points b_1 and b_2 of intersection with B, in another O(log k) steps. If $\overline{a_1a_2}$ does not intersect B (implying that no b_1 and b_2 are found), then B must lie entirely above or entirely below this line. We need only test the location of a single point of B with respect to $\overline{a_1a_2}$ to find out which is the case. If B lies entirely above $\overline{a_1a_2}$, then we can effectively eliminate the "lower" half of A, as it can impossibly contain an intersection. Otherwise we eliminate the "upper" half of A (we will indicate how momentarily).

Assume that $\overline{a_1a_2}$ does intersect B. If the intervals $[a_1a_2]$ and $[b_1b_2]$ on the line are not disjoint, then neither are A and B and our procedure can terminate. If the intervals are disjoint, then we proceed as follows (see figure 17). Assume that b_2 and a_1 are adjacent. (A similar development applies when b_1 and a_2 are adjacent.) Draw a tangent l_B to B through b_2 and a tangent l_A to A through a_1. If l_B and l_A

INSERT FIGURE 17 ABOUT HERE

---

meet below the line, then A and B cannot intersect above it and we may as well eliminate the "upper" half of A. We do so by splitting the preprocessed form of A (while keeping a record of where we split), at a cost of only $O(\log k)$, effectively throwing half of its number of points away. If l_B and l_A meet above the line, then we proceed similarly. In each case we apply the same procedure recursively and continue unless $l_B \not\parallel l_A$ (implying that A and B are disjoint) or A has been reduced to 1 or 2 points (and separability can be decided "by hand").

Note that each step takes $O(\log k)$ time and we either reach a decision or can eliminate another half of A. Thus no more than $\log k$ steps of this sort can be taken, accounting for the $O(\log^2 k)$ total time bound. After the answer is reached, the splitting of A must be "undone". Using the record of past splits, one can put the pieces back together in the right order within the same time-bound and bring A into its original shape as if nothing happened.

□

It so happens that the structure implicit in theorem 4.6. maintains convex hulls in a form suitable for proposition 5.5. (cf. section 2). Thus we conclude

Theorem 5.6. One can maintain two sets A and B of points in the plane such that insertions take $O(\log^2 n)$ time and deletions take $O(\log^3 n)$ time each (where n is the current size of the set on which they operate) and, whenever needed, separability can be decided in only $O(\log^2 n)$ time.

Note in theorem 5.6. that we could as well precompute the answer to a separability query after every insertion or deletion, thus effectively hiding the "query time" in the given bounds for the update times and resulting in a query time of $O(1)$. Recently, Chazelle and Dobkin [8] have shown that the bound in proposition 5.5. can be improved to $O(\log n)$ by a process that eliminates parts of both A and B. It gives a corresponding improvement of theorem 5.6., when properly implemented.

6. Dynamically maintaining the common intersection of a set of halfspaces (representation, structure and on-line maintenance)

A problem remotely similar to convex hull determination concerns the computation of the common intersection of a set of n halfspaces in the plane. A halfspace is a part of the plane entirely to the left or to the right of a specified straight line. The common intersection of a set of n such half-

spaces is a convex polygon with at most n edges, where the polygon could very well be empty or have an "open" side (see figure 18). If we interprete a halfspace

---

INSERT FIGURE 18 ABOUT HERE

---

as the set of points satisfying some inequality $ax + by \leq c$, then the problem we consider is easily motivated as that of determining the region of all points which satisfy a system of such inequalities simultaneously.

Shamos and Hoey [26] have shown that the common intersection of a set of n halfspaces in the plane can be found in $O(n \log n)$ steps. There is more than one way to actually achieve this bound, but all techniques used until now do not apply to an on-line environment and work for static sets only. Even partial results apparently are lacking concerning the dynamic version of this problem, in which we would randomly insert or delete halfspaces. We will show that a suitable notion of decomposability can again be identified and exploited in this problem, to obtain a dynamic maintenance algorithm along very much the same lines of reasoning as in the case of convex hulls. In this section we shall consider some of the necessary representational details, which are somewhat more technical and tedious than for convex hulls (largely because halfspaces are harder to deal with than points, compare Brown [7]).

A halfspace is bounded by a straight line, which is determined once we know its slope and a point. The slope of the bounding line will be called the slope of the halfspace in question. If we orient lines such that they always point "upwards", then we can fully determine a halfspace by specifying a line and indicating whether to take the "left" of the "right" part of the space (see figure 19). In this way

---

INSERT FIGURE 19 ABOUT HERE

---

we can refer to the "left" and "right" halfspaces of a set, respectively.

As for convex hulls it will be advantageous to distinguish between the left and right halfspaces of a set and to maintain their common intersections separately.

Definition. The l-intersection of a given set of halfspaces is the common intersection of the "left" halfspaces of the set. The r-intersection is the common intersection of the "right" halfspaces of the set.

Let us consider what representation we must choose for the l-intersection of a set of halfspaces. An l-intersection is an open convex domain, bounded to the right by a convex arc made up of connected segments of the bounding lines of the contributing left halfspaces (see figure 20). Considering the boundary, it is important

---

INSERT FIGURE 20 ABOUT HERE

---

to observe that the halfspaces which "contribute" to it do so in increasing order by slope. It clearly suggests that the l-intersection of a set of halfspaces must be represented by the subset of contributing halfspaces sorted by slope. With the representation for lc-hulls in mind, we will assume that the contributing halfspaces are stored in sorted order at the leaves of some binary search tree Q_L (see figure 21), which supports the repertoire of a concatenable queue

---

INSERT FIGURE 21 ABOUT HERE

---

and which keeps its leaves chained in a doubly linked list as well. If it is required to determine an edge of the boundary (viz. a corner point), then it is sufficient to just intersect the bounding line of a halfspace in Q_L with the bounding lines of the neighboring leaves. Because this takes

only O(1), we can for all practical purposes identify the leaves of Q_L with the edges of the boundary of the l-intersection in traversal order (figure 21). We also assume that Q_L is "internally" linked in a way as described in section 2, to allow for binary searches over the boundary. It will enable us to detect whether a point lies to the left or to the right and to compute the point(s) of intersection with a straight line on the boundary in only O(log n) steps, using a search procedure almost identical to the one for closed convex n-gons.

The r-intersection of a set of halfspaces will be represented in a concatenable queue Q_R in completely the same fashion. Notice that Q_L and Q_R always consist of disjoint sets of halfspaces, because they are synthesized from the disjoint subsets of left and right halfspaces respectively. The idea is to dynamize the common intersection of a set by separately maintaining the l- and r-intersection of the set as represented in Q_L and Q_R. The following analog of lemma 3.1. shows why this may be promising.

Lemma 6.1. Given the l- and r-intersection of a set of n halfspaces, one can determine whether an arbitrary point p lies inside, outside or on the boundary of the common intersection of the set in only O(log n) steps.

Proof

Just observe e.g. that p lies inside the common intersection of the set if and only if it lies "left" of the boundary of the l-intersection and "right" of the boundary of the r-intersection. In this way the required answers can be obtained by knowing p's location with respect to the l- and r-intersection, respectively, which one can determine in O(log n) each from Q_L and Q_R.

□

To simplify later formulations, we introduce the following terminology.

Definition. The l-boundary is the boundary of the l-intersection of a set of halfspaces (as it is represented in Q_L), the r-boundary is the boundary of the r-intersection of the set (represented in Q_R).

Separately maintaining the l- and r-intersection of a set apparently fails to keep track of what the common intersection really is, although one can answer queries about it. To solve our problem one must be able to compute the common intersection (i.e., the convex boundary of it) with only little extra effort. It should be clear

---

INSERT FIGURE 22 ABOUT HERE

---

that, in order to determine the "contour" of the common intersection, one must compute the intersection of the l- and r-boundaries as "open" n-gons. See figure 22 for some conceivable cases.

Theorem 6.2. The point(s) where the l- and r-boundaries intersect can be found in $O(\log^2 n)$ steps.

Proof

Let the l- and r-boundary be called L and R, respectively. We assume that L and R are given by means of the concatenable queues Q_L and Q_R introduced earlier. It is important to note that we make no assumptions about any relation that might exist between L and R. All we use is that L is "open" to the left and R is "open" to the right.

Computing the intersection of L and R proceeds in two phases. First we try to locate some point p on R that lies to the left of L. (If there is no such p, then the intersection is empty.) Once such a p has been found, L and R must intersect (except in some degenerate cases) and there is at most one point of intersection "above" and "below" p respectively. The next phase will locate these points of intersection, whenever they exist.

Phase 1.

Our goal is to find a point p on R that lies to the left of L or to establish the fact no such point exists.

When L consists of one straight line only, it takes no more than $O(\log n)$ steps to determine whether it intersects R at all (using binary search over R) and, if so, to pick a point p as desired (see figure 23 a, b, c, d). When L

INSERT FIGURE 23 ABOUT HERE

---

consists of 2 halflines intersecting at some point q, the location of q with respect to R and the points of intersection of L and R can be found

in O(log n) steps by considering the "lines" of L as if they were separate. Many cases can occur (see figure 24 a-h), but all are easily detected and a point p can be chosen with no extra effort. It shows that when Q_L is "down" to 1 or 2 leaves

---

INSERT FIGURE 24 ABOUT HERE

---

the choice can be decided. For larger L (i.e., when Q_L has more than 2 leaves) we will attempt to split off a part at most half as large repeatedly to reduce the problem, while narrowing down the search for some p if one exists. It is important now that Q_L was chosen to be a splittable data-structure, so a proper invariant can be maintained. In particular we will guarantee that, after splitting, the "end-most" elements of the subset of L on which the algorithm continues are extended to infinity (even though they may contribute only a segment to the original Q_L). It should not be mysterious, as it will follow from the algorithm that in this way only useless parts of the boundaries are shielded off (figure 25).

INSERT FIGURE 25 ABOUT HERE

We shall now argue how the recursive algorithm operates on L (i.e., on Q_L) and narrows down the search by repeated halving.

Consider the "middle" segment of L. From the information at the root of Q_L we know which segment it is at no special charge (figure 26). Let the end-points of the segment be q_1 and q_2. Let us now consider the horizontal lines l_1 and l_2 through q_1 and q_2 (respectively). Because of the nature of L, the lines l_1 and l_2 do not intersect L anywhere else. And because l_1 and l_2 are horizontal, they also intersect R in at most one

---

INSERT FIGURE 26 ABOUT HERE

---

point each. It takes at most O(log n) steps to find these points of intersection r_1 and r_2, if they exist, using a binary search over R. We shall distinguish a number of cases that can now occur.

Case(i): there are no points of intersection (i.e., r_1 and r_2 do not exist).

This forces R to be in very specific parts of the plane. The different cases are shown in figure 27 a-d and are very easy

---

INSERT FIGURE 27 ABOUT HERE

---

to distinguish. In cases (a) and (b) we only need to intersect R with $\overline{q_1 q_2}$ (in another O(log n) steps) to determine the precise location of R. In case (a) a point p can be selected, in case (b) it can not be. Cases (c) and (d) are fully symmetric and we need only determine for a single point of R whether it lies above or below l_1 to distinguish the two. Let us just consider case (c). In this case it is conceivable that there is an intersection of L and R above line l_1, but there definitely is none below it. It means we may as well confine the search to the "upper" half of L (see figure 28), which is stored in the subtree of Q_L below the left-son of its root.

INSERT FIGURE 28 ABOUT HERE

After redefining the parameters, the search procedure can continue with an "L" of half the original size.

Case(ii): there is a point of intersection on either l_1 or l_2 that lies to the left of L.

It should be clear that such a point of intersection can be taken as a point "p" for our purposes.

Case(iii): there is one point of intersection and it lies to the right of L.

We omit a detailed discussion of this case, becuase the considerations needed here are very similar to the next case.

Case(iv): there are two points of intersection and both lie to the right of L (i.e., r_1 and r_2 exist and both are located to the right of q_1 and q_2).

The different situations that can arise are shown in figure 29 a-d and can be distinguished easily. Case (b) is

---

INSERT FIGURE 29 ABOUT HERE

---

detected by intersecting R with $\overline{q_1q_2}$ in O(log n) steps. If it occurs a point p can be chosen. Otherwise a decision can not immediately be made. The cases (a), (c) and (d) can be distinguished completely by comparing the slope of R's segments at r_1 and r_2 with the slope of $\overline{q_1q_2}$. Only in cases (c) and (d) there is hope that an intersection may still exist. Just considering case (c) (case (d) is similar), it is clear that an intersection can only occur above l_1. Thus the proper action is to split off the lower portion of L again and to repeat the procedure on the resulting subproblem.

It should be clear that all possible cases have been dealt with and that each case takes O(log n) steps to detect and to handle. Because in each case either a decision is reached or the size of the part of L on which the procedure is repeated is about cut in half, the entire process cannot continue for more than O(log n) times. Hence phase 1 terminates after $O(\log^2 n)$ steps. If any datastructures (Q_L or Q_R) were split for greater efficiency, then they should be put back together. But it is clear that this stays within the $O(\log^2 n)$ bound as well.

Phase 2

If no point p was found in phase 1, then L and R do not intersect. We assume that a p was found and now aim for the construction of the intersection. Observe that in some cases considered in phase 1 the actual points of intersection were found already. Also, the case in which L consists of just 1 or 2 contributing halfspaces was dealt with in the introduction of phase 1. We shall consider what to do for larger L and devise a halving procedure to search for the points of intersection.

Phase 1 gave us a horizontal line that intersects R in p and L in some point q, with p to the right of q (figure 30). It means that R and L have at

---

INSERT FIGURE 30 ABOUT HERE

---

most one point of intersection above and below l respectively. We shall search for these points separately. Because both searches proceed in very much the same way, we only discuss how to find the intersection above l (if one exists). Note that we can split off the part of L (i.e., of Q_L) below l and eliminate it from the current process.

From the root of the datastructure for the current L, we can obtain the middle segment in only O(1) steps (see figure 31). As before, we draw the horizontal lines l_1 and

---

INSERT FIGURE 31 ABOUT HERE

---

l_2 through the end-points q_1 and q_2 of the segment, and determine their intersections r_1 and r_2 with R (if these intersections exist) in the usual O(log n) steps. The six

---

INSERT FIGURE 32 ABOUT HERE

---

different situations, depending on the form of R, are displayed in figure 32 and can usually be distinguished based on whether r_1 and/or r_2 exist and, if they do, where they are located. We consider the following cases.

Case(i): there is at most one point of intersection to the left of L (figure 32 a-d).

It covers the situations in which r_1 either does not exist or lies to the right of L (i.e., right of q_1 on l_1). If r_2 exists and lies to the left of L, then the intersection of R and L can be located by just intersecting R with $\overline{q_1 q_2}$ in another O(log n) steps. In all other cases we must conclude

that the intersection of R and L lies "below" the current segment and we can can cut off and eliminate the upper half of the current L from further consideration. The same search procedure is now repeated, on a "new" L that is half the size of before.

Case(ii): both r_1 and r_2 exist and lie to the left of L (figure 32 e-f). Now we are still not sure that R and L actually intersect (above l) but, if they do, we know the intersection must lie "above" the current segment on L. Thus we can split the datastructure again and confine the search to the upper half of L.

It should be clear that all possible cases have been dealt with. Each case takes O(log n) work and if no intersection is found yet, the search continues with "L" cut in half. Obviously this cannot go on for more than O(log n) times. Hence phase 2 terminates in a total number of $O(\log^2 n)$ steps as well. Any splittings (of Q_L and Q_R) that were made must be un-done, but this will take no more than $O(\log^2 n)$ either.

Phases 1 and 2 together solve the problem of intersecting L and R within the time-bound stated.

□

Thus, glueing the "left" and "right" constituents of the common intersection is not as easy as it was for the left and right sides of a convex hull, but the upperbound is not discouraging. It is conceivable that a better algorithm for theorem 6.2. exists.

Definition. Let $J_2(n)$ be the best bound achievable for computing the point(s) of intersection of an l- and r-boundary consisting of at most n segments each, represented in a concatenable queue with "log n" characteristics.

Proposition 6.3. Given representations of the l- and r-boundaries as concatenable queues, one can compute (a datastructure containing) the boundary of the common intersection of a set of n halfspaces in $O(J_2(n))$ steps.

Proof

It takes $J_2(n)$ steps to find the intersection of the l- and r-boundaries. In an additional O(log n) steps, one can split off the parts of these boundaries which enclose the common intersection of the domains (see figure 22) and join them in a single representation of the resulting convex (open) n-gon.

□

We assume that $J_2(n) = \Omega(\log n)$ and take $J_2(n) = O(\log^2 n)$ for all concrete upperbounds. The results support our earlier decision to separately maintain the l- and r-intersection of a set of halfspaces. The common intersection can be determined when needed with relatively little computational effort. In the remainder we shall consider how the l-intersection of a set of halfspaces can be dynamically maintained.

It appears to be fairly easy to maintain the l-intersection of a set of halfspaces when only insertions occur. The following result can be obtained, of interest for an on-line construction of the l-intersection of a set (in the spirit of Preparata [19]).

Theorem 6.4. One can compute the l-intersection of a set of n halfspaces in the plane by adding its elements into the structure one after the other, such that each time it takes only O(log n) steps to fully update the current l-boundary after an insertion.

Proof

Maintaining the current l-boundary L as a concatenable queue Q, let us see what happens when an other halfspace h is inserted in the set. We assume, as we may, that h is indeed a left halfspace.

The key to updating Q is to consider in what way h intersects L. Only a limited number of cases can occur (figure 33 a-e), which can all be distinguished easily after running the O(log n) algorithm to determine the actual point(s) where h cuts through L. We shall only explain how to proceed when h intersects L at two points (figure 33 e), as the necessary actions in all other cases are very similar and can be left to the reader.

INSERT FIGURE 33 ABOUT HERE

Hence let h intersect L in the points p and q, which are located on the bounding lines of halfspaces h_1 and h_2 (respectively) currently in the set. To update Q one must delete all halfspaces currently "between" h_1 and h_2 (by slope) and insert h for them instead. It is clear that the structure chosen for Q allows one to perform this in only O(log n) steps.

The total time to update Q, hence the current l-intersection, after each insertion remains within the O(log n) bound.

□

Theorem 6.4. shows that, as for convex hulls, there is a "real-time" algorithm for building 1-intersections. In a very similar way one can, in fact, obtain a realtime algorithm to build the common intersection of a set of halfspaces itself.

When both insertions and deletions must be processed, a more involved procedure must be followed. Regardless of whether they contribute to the boundary of the common intersection or not, it is important to keep all halfspaces in a datastructure T. Because halfspaces contribute to common intersections in increasing order of slope, we choose for T a balanced binary search tree in which halfspaces are kept sorted by slope (figure 34). Ideally we now want to augment T and associate with each

---

INSERT FIGURE 34 ABOUT HERE

---

internal node α of T a concatenable queue Q_α containing (the 1-boundary of) the 1-intersection of the halfspaces in its subtree! Before we do so, we need to establish one more basic fact for 1-intersections (viz. 1-boundaries).

The 1-intersection of a set of halfspaces H is a decomposable configuration in the following sense. Sort the elements of H by slope and split H at some arbitrary point, to obtain two subsets A and C of halfspaces which have slope less than or greater than a certain halfspace h, respectively. It turns out that, as in the case of convex hulls, the 1-intersection of H can be determined with relatively little computational effort from the 1-intersections of A and of C separately.

Theorem 6.5. Let $H = \{h_1, \ldots, h_n\}$ be a set of halfspaces, sorted by slope. Given the 1-intersections of $A = \{h_1, \ldots, h_i\}$ and of $C = \{h_{i+1}, \ldots, h_n\}$ as concatenable queues (any $1 \leq i < n$), the 1-intersection of H can be computed in $O(\log n + J_2(n))$ steps (represented in a concatenable queue again).

Proof

By the earlier remarks it is sufficient to consider the 1-boundaries of the sets in question. Let the 1-boundaries of A and C be given. Using that A and C are "separated" by slope, there must exist a halfspace h (i.e., a bounding line) whose slope is just in between. Draw an arbitrary halfspace h of such a slope. The different situations that can arise are displayed in figure 35 a-b (where h can be of any slope), for the sake of clarity.

INSERT FIGURE 35 ABOUT HERE

The main observation should be that the 1-boundaries of A and C intersect in precisely one point q. Clearly the 1-boundary of H is obtained by taking C's boundary up to q and continuing on A's boundary from q onwards.

It is not very hard to compute q, because the same algorithm as explained in theorem 6.2. will apply. This is most easily seen when we tilt figure 34 a-b and put h in the position of the x-axis, by a simple change of coordinates. The halfspaces comprising the set A still face leftwards, but the halfspaces in C now face "the other way". Thus for all practical purposes the "boundary" of C has become an r-boundary and theorem 6.2. applies litterally. It should be noted that the representations of A and C are still valid as they were, as long as the change of coordinates is carried through in all manipulations After finding q in $O(J_2(n))$ steps, we split the queues representing A and C's 1-boundaries and glue them together in the right order, to obtain the 1-boundary of H in only O(log n) additional steps.

□

We shall exploit theorem 6.5. in a dynamic algorithm for maintaining the 1-intersection of a set in the next section.

We observe that, as a bonus, theorem 6.5. gives us a method to construct the common intersection of a set of halfspaces in a very special way.

Theorem 6.6. There is an algorithm to compute the common intersection of a set of n halfspaces that, after sorting the halfspaces by slope in O(n log n) steps, takes only O(n) additional steps to complete.

Proof

It is sufficient (by 6.2.) to consider the computation of the 1-intersection only. Sort the given set and proceed as follows. For simplicity we assume that $n = 2^k$, some k. For i from 1 to k repeat computing the 1-intersection of a next group of 2^i halfspaces from the 1-intersection of each of the two constituent "halves" as computed in the previous round. Using theorem 6.5. this procedure takes

$$n + \frac{n}{2}\log^2 2 + \ldots + \frac{n}{2^i}\log^2 2^i + \ldots = O(n)$$

steps after the initial sort.

□

7. Dynamically maintaining the common intersection of a set of halfspaces (algorithms and applications)

From now on we shall assume that the common intersection of a set of halfspaces is represented as the "junction" of its l- and r-intersection. We shall concentrate on the dynamic maintenance of the l-intersection of a set, because the results will carry over ipso facto to the common intersection as such. The reason for it is clear: the l-intersection is decomposable in a way similar to lc-hulls and the hope is justified that a full dynamization can be obtained along the same lines.

Assume that all (left-) halfspaces presently in the set are stored at the leaves of a balanced binary tree T, using their slope as the sorting key (see figure 34). It is tempting to associate with each internal node α of T the concatenable queue Q_α representing the l-intersection of the set of halfspaces in its subtree, but the development in Section 4 has taught us to try and be more clever. From the decomposability of l-intersections as expressed in theorem 6.5. it is clear that Q_α can be computed efficiently from the queues "stored" at the two sons γ and δ of α. From the proof of theorem 6.5. it is clear also that Q_α is obtained in a very regular way from Q_γ and Q_δ, generally by taking a front piece of the first and a tail piece of the second. Thus a situation completely similar to that for lc-hulls has been created (see e.g., figure 11).

We conclude that we must augment T to a tree T^* in which the internal nodes α have associated with it the left- or right portion of Q_α that was not used to form the l-intersection (as a concatenable queue) at its father node! The l-intersection of the complete set will be available at the root of T^*. The further details concerning T^* are completely the same as they were in Section 4. In particular one can immediately obtain the following analog to theorem 4.5.:

Theorem 7.1. The common intersection of a set of halfspaces in the plane can be maintained at a cost of only $O(\log^2 n + \log n.J_2(n))$ steps per insertion and deletion, where n is number of halfspaces currently in the set.

Proof

The procedures DOWN and UP as they were developed in Section 4 carry over

without any change (except terminology). Insertions and deletions are processed in the same way as described in the proof of theorem 4.5. for convex hulls. The time analysis carries over with J_1 replaced by J_2. Note that by proposition 6.3. it would take no more than $O(J_2(n))$ extra steps to maintain the common intersection of the set from the l- and r-intersections as they are kept up-to-date, which is well within the bound stated.

□

As we have (as yet) no better bound for J_2 than $\log^2 n$, theorem 7.1. tells us that the common intersection of a set of n halfspaces can be maintained at a cost of only $O(\log^3 n)$ steps per insertion and deletion. When J_2 is improved to e.g. log n, a log-factor can be saved in this result and its subsequent applications. A slight improvement can be obtained by more carefully examining the cost of UP's action after an insertion, completely in the spirit of theorem 4.5*. The idea is that, when a halfspace h is added, it cannot change the current l-boundary of a set too drastically (as it can only cut off an existing segment of it by a straight line) and enables one to "connect" the boundaries of neighboring sets of halfspaces separated by slope by merely looking for the effect of h, without the need for a fresh $O(\log^2 n)$ search for their one point of intersection.

Theorem 7.2. The common intersection of a set of halfspaces in the plane can be maintained at a cost of $O(\log^2 n)$ per insertion and a cost of $O(\log^3 n)$ per deletion, where n is the number of halfspaces currently in the set.

Proof

The procedures DOWN and UP are essentially left as they are, except that we will change the way UP computes the l-boundary of a set $H = A \cup C$ from the l-boundaries of "separated" sets of halfspaces A and C after an insertion. For the procedure below to work, it is necessary that at each node α of T^* it is remembered what the point of intersection q_α is of the l-boundaries of the halfspaces below α's sons (because it is the point where these boundaries were glued together).

Let h be the halfspace to be inserted. After performing DOWN and creating a leaf for h, let us consider the necessary actions when UP has reached a node α (see figure 36). Let the sons of α be γ and δ, and let the l-boundaries

---

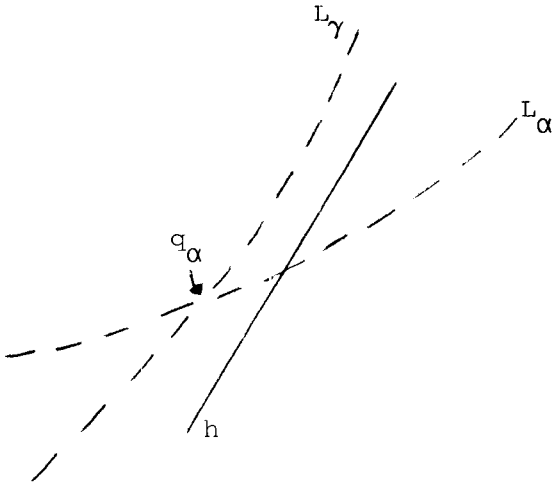
INSERT FIGURE 36 ABOUT HERE

---

of the sets of halfspaces they cover (before h is inserted) be L_γ and L_δ. When UP reaches it, the point q_α stored at node α is the (unique) point where L_γ and L_δ intersect. Let us assume that h got inserted in the subtree below γ. (If h got inserted below δ, then a very similar argument would apply.)

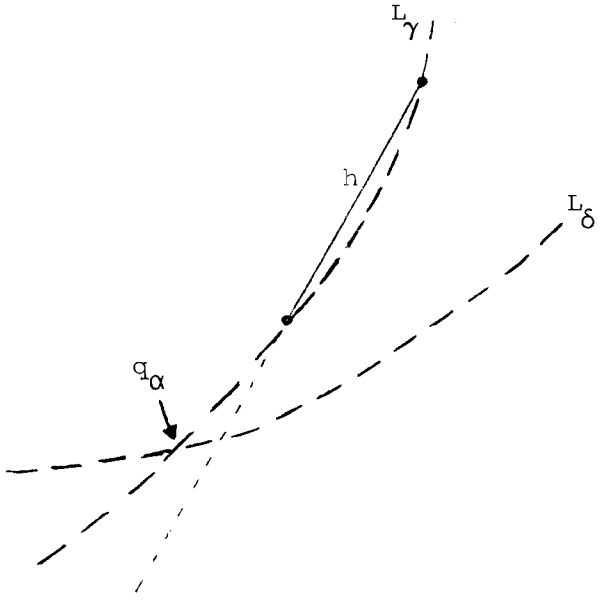
Because h got inserted, the 1-boundary L_γ may have changed. Let us consider what possible changes h can cause to the original L_γ and what it means if we want to compute the new point of intersection between (the new) L_γ and L_δ. After all, this computation would be required to determine the new 1-boundary L_α. Only three different situations can occur.

Case(i): h does not contribute to L_γ.



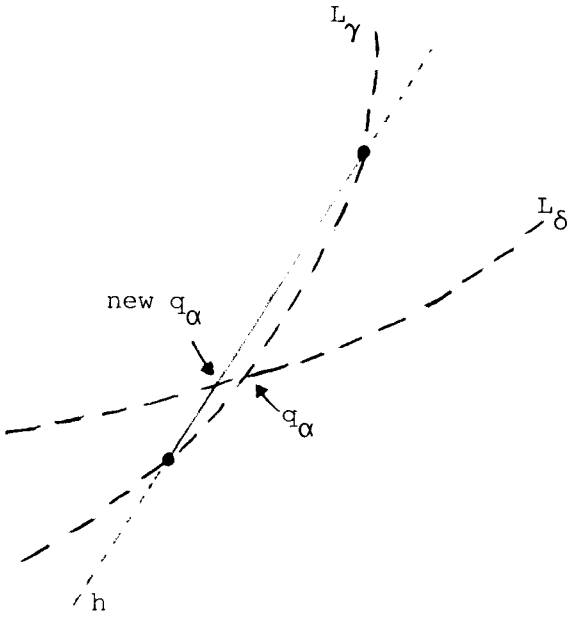
It means that h lies "to the right" of L_γ. The situation is easy enough to recognize, by testing whether h occurs in the concatenable queue currently representing L_γ. If indeed h does not contribute to L_γ, then UP does not have to recompute anything (except perhaps to restore balance) as it proceeds upwards.

Case(ii): h does contribute to L_γ but the current q_α lies above or to the left of it.



This situation again is easy to detect. The "contour" of L_γ has now changed and contains a segment of h. But when q_α lies above or to the left of h, the part of L_γ that intersects L_δ has <u>not</u> changed. In particular it means that the new L_γ and L_δ still intersect at the same point q_α.

Case(iii): h does contribute to L_γ and the current q_α lies below or to the right of it.



Again the contour of L_γ has been changed because of h and this time a new point of intersection with L_δ must be computed. But a moment's reflection shows that the new point of intersection must be the point where h and L_δ intersect! By means of the representation chosen for the l-boundaries like L_δ, it takes only O(log n) steps to compute the intersection with a straight line.

It follows that in all cases the "new" point of intersection between L_γ and L_δ can be computed in only O(log n) steps, at each node UP encounters as it works its way towards the root after an insertion. The remaining actions performed at each node are simple O(log n) operations on concatenable queues, except when a rebalancing is called for. The total cost to process an insertion in the way we described will be $O(\log^2 n + r.J_2(n))$, where r is the number of rebalancings (rotations) required because of the insertion.

Choosing T to be an AVL-tree (compare theorem 4.6.), the number r referred to need never be larger than 1. It follows that the costs for processing an insertion can be kept to within $O(\log^2 n)$. The costs for processing a deletion remains at $O(\log^3 n)$.

□

Halfspaces come up in a number of interesting problems in the plane and theorem 7.2. will help us to obtain dynamizations of an unexpected efficiency. A first application concerns the simplest type of intersection query: "does the point x belong to the common intersection of the set of halfspaces H". This is a particularly interesting type of query, because it is an example of a decomposable searching problem in the sense of Bentley [3] to which previously only very general dynamization methods were believed applicable (which yield only average or worse bounds than we can now obtain). Combining 7.2. and 6.1. we conclude

Theorem 7.3. One can dynamically maintain the common intersection of a set of halfspaces in the plane such that insertions and deletions can be processed in $O(\log^2 n)$ and $O(\log^3 n)$ steps, respectively, and queries of the form "does x belong to the current common intersection" can be answered in only $O(\log n)$ steps at any moment, where n denotes the number of halfspaces in the set.

The same result holds for queries of the form "is the common intersection currently empty".

The common intersection of a set of halfspaces plays a role, for instance, in finding the kernel of a simple polygon (i.e., a closed polygon with no intersecting edges). The kernel of a simple polygon is most easily described as the set of points in its interior from which all sides of the polygon are completely visible (i.e., from endpoint to endpoint). It is the common intersection of the halfspaces facing the interior, obtained by extending the sides of the polygon to become bounding lines. Shamos and Hoey [26] first reported an $O(n \log n)$ algorithm for determining the kernel of a simple n-gon. Later Lee and Preparata [17] showed that when the contour of the n-gon is given in traversal order an $O(n)$ algorithm suffices. We can efficiently maintain the kernel of a dynamically changing polygon, assuming that the changes merely involve the insertion and deletion of edges which keep the polygon simple.

Theorem 7.4. One can dynamically maintain the kernel of a simple n-gon at a cost of only $O(\log^3 n)$ steps per transaction, assuming that transactions merely involve the insertion and/or deletion of some edges that keep the polygon simple.

A last but perhaps most interesting application involves some elementary notions from linear programming. A linear program in n variables consists of a set of linear inequalities and a linear object function F, which must be minimized (or maximized) over the feasible region of points which satisfy all inequalities simultaneously. It is well-known that the feasible region is polyhedral and that (except in degenerate cases) F assumes its extreme values at the extreme points of the polyhedron. We observe that the feasible region is nothing but the common intersection of the set of halfspaces determined by the linear inequalities of the linear program.

Theorem 7.5. One can dynamically maintain the feasible region of a linear program in 2 variables at a cost of only $O(\log^2 n)$ steps for each inequality added and $O(\log^3 n)$ for each inequality deleted.

8. Dynamically maintaining the maximal elements of a plane set (on-line construction and representation)

Another problem commonly considered in computational geometry concerns the computation of the maximal elements of a set (in the plane). Let points be partially ordered in the usual manner by coordinates. Thus for $x = (x_1, x_2)$ and $y = (y_1, y_2)$ we write $x \leq y$ if and only if $x_1 \leq y_1$ and $x_2 \leq y_2$. A point x is called maximal in a set P when $x \in P$ and no $y \in P$ exists with $y > x$. It is customary to draw horizontal and vertical lines from each of the maximal elements of a set (see figure 37), until they

---

INSERT FIGURE 37 ABOUT HERE

---

cross. It connects the maximal elements of a set by a contour of horizontal and vertical line-segments, creating a "staircase" going up in leftward direction. Having the entire set to its left, the contour of maximal elements in not unlike an rc-hull as introduced in Section 3. We shall discover that the analogy can be carried a long way through, to obtain a dynamization of the problem once again by very much the same line of reasoning.

Definition. The contour spanned by the maximal elements of a set of points in the plane will be called its m-contour.

Computing the maximal elements of a set is equivalent to computing its m-contour. The representations normally allow us to identify the two without any considerable overhead.

For a static set of n points in the plane Kung, Luccio and Preparata [16] have shown how the maximal elements can be computed in O(n log n) steps and supplied an argument of why this bound is essentially optimal (see also van Emde Boas [28]). From a more general viewpoint, maximal element determination is but a special case of the ECDF searching problem which requests that for each $x \in P$ the number $A(x) = \#\{y \in P \mid x < y\}$ be computed. (Maximal elements are precisely those points x which have $A(x) = 0$.) Using a recursive splitting strategy, Bentley and Shamos [5] showed that ECDF searching in d-dimensional space can be solved in only $O(n \log^{d-1} n)$ time. For $d = 2$ it yields yet another O(n log n) solution which is completely unadaptive in a dynamic environment. We will show how the "m-contour" can be maintained dynamically.

Clearly the m-contour of a set is only a way to visualize the arrangement of its maximal elements more easily. Observe (when viewed from left to right) that the maximal elements occur along the contour in increasing order by x-coordinate and, at the same time, in decreasing order by y-coordinate. This property is a very useful invariant and makes it possible to store the maximal elements in an efficient concatenable queue Q (figure 38) which, when properly managed, can be used for binary searching both on x- and on y-coordinate along the contour. It enables us to make

---

INSERT FIGURE 38 ABOUT HERE

---

the following claim.

Lemma 8.1. Given the m-contour of a set of n points in the plane (as a concatenable queue), one can compute its intersection with any horizontal or vertical line in only O(log n) steps.

Proof

Let the maximal elements of the set be numbered as m_0, m_1, ... in the (sorted) order in which they appear along the m-contour. We will only show the argument for computing the intersection of the contour with a horizontal line $y = c$.

It is crucial to note that Q can be used for binary searching on the y-coordinates of the maximal elements in the set,

---

INSERT FIGURE 39 ABOUT HERE

---

merely by disregarding their x-coordinates (the elements appear sorted on either coordinate). Assuming that $c \leq y_{m_0}$ (which is required for there to be any intersection at all), it takes only O(log n) steps to find an i such that $c = y_{m_i}$ or $y_{m_i} < c \leq y_{m_{i+1}}$. The cases are illustrated in figure 39 a and b respectively (the case in which m_i is the "last" element on the contour is

easily handled). In the first case the intersection is a line-segment of known location and size on the line, in the second case it is the point (x_{m_i}, c).

□

The result of lemma 8.1. can be shown for all straight lines of slope between 0 and 90 degrees.

Before we tackle a general dynamic version of our problem, we shall prove that the contour of maximal elements of a set can be updated efficiently whenever a new point is added to the set. It yields a result very similar in spirit to Preparata's real-time algorithm [19] for convex hull construction.

Theorem 8.2. One can compute the maximal elements of a set of n points in the plane (as a queue) by adding its points one after the other and updating a current contour completely in O(log n) steps after each insertion.

Proof

Assume that a current m-contour is stored in Q as described and let a next point p of the set be coming in. By considering the horizontal line through p and intersecting it with the m-contour one can determine whether p lies to the left of (or on) the contour or not. If it does, then it can not be maximal and can be discarded forever. Otherwise any one of the cases shown in figure 40 a-c can happen (m_k denotes the "last" element

---

INSERT FIGURE 40 ABOUT HERE

---

on the contour). By inspecting the x- and y-coordinates of the end-points of the current contour and comparing with those of p, one can easily distinguish between these three cases.

We shall consider case b (figure 41) only, as the argument for the remaining cases is completely similar. From the previous stage we know at what point q_1 the horizontal line through p intersects the contour. (If p is on one line with a current maximal element, then we let it be q_1.) In the same way we now compute the point q_2 on the

---

INSERT FIGURE 41 ABOUT HERE

---

contour where the vertical line through p intersects (see figure 41). To update the m-contour correctly one must delete the "segment" from q_1 to q_2 (i.e., delete the maximal elements on this stretch) and insert p for it. As Q is a concatenable queue, this can be accomplished in O(log n) steps by ordinary datastructure manipulation.

Because the necessary intersections can be computed in O(log n) steps as well by lemma 8.1., the bound of O(log n) applies to the entire construction for each point added.

□

As in the case of convex hulls (cf. Preparata [19]), theorem 8.2. is the best uniform result one can hope for. Yet the structure that is maintained will not be adequate for supporting deletions as well, because it ignores the need to keep track of the "interior" of the hull of current maximal elements (compare Section 2). This we shall now patch.

To do so we shall follow a very similar approach as for convex hulls and halfspaces. Let us store all points of

---

INSERT FIGURE 42 ABOUT HERE

---

the set in a data-structure T that can be dynamically maintained. As the maximal elements we wish to select will eventually appear in sorted order by y-coordinate along the contour, it is reasonable to choose for T a balanced binary search tree in which points are entered with their y-coordinate as a key (figure 42). For the very same reason we could have chosen to maintain points in sorted order by x-coordinate, but we have not done so to preserve the similarity of our approach with the approach in Section 3 (for lc-hulls). Ideally we would now augment T and associate with every internal node α concatenable queue Q_α containing the maximal elements (in order) of the set of

points covered by its subtree (see figure 43). While this has always been

---

INSERT FIGURE 43 ABOUT HERE

---

the first step in previous problems, we also know that we must look for an additional property that enables us to "glue" neighboring m-contours when neighboring subsets are taken together.

The m-contour of a set of points in the plane is a decomposable configuration in the following sense. Let the points be sorted by y-coordinate (as they are) and split the set by drawing an arbitrary horizontal line in two disjoint subsets A and C (see figure 43). It turns out that the m-contours of two horizontally separated subsets can be combined with relatively little computational effort, to obtain the m-contour of the original set.

Theorem 8.3. Let $P = \{p_1, \ldots, p_n\}$ be a set of points in the plane, ordered by y-coordinate. Given the m-contours of $A = \{p_1, \ldots, p_i\}$ and of $C = \{p_{i+1}, \ldots, p_n\}$ as concatenable queues (any $1 \leq i < n$), the m-contour of P can be computed in only O(log n) steps.

Proof

Let the contours of A and C be given in concatenable queues Q_A and Q_C, respectively. Note that A and C are separated by an (imaginary) horizontal line and that A lies above C. Let p be the "last" maximal element, i.e., the rightmost (and lowest) point on A's contour.

Considering the set P as the union of A and C, it should be clear that the maximal elements of A are also maximal in P but that this is not necessarily true for the maximal elements of C. Draw the vertical line through p (the "last" edge of A's contour) and compute

---

INSERT FIGURE 44 ABOUT HERE

---

the point q (if it exists ...) where it intersects C's contour. The different cases that can arise are shown in figure 44 a-b.

When no intersection exists (figure 44 a), the m-contour of A will "pass" entirely in front of the set C and no element of C can be maximal in P. It follows that the m-contour of P is identical to the m-contour of A.

When there is an intersection q (figure 44 b), the m-contour of P is obtained by concatenating the contour of A with the contour of maximal elements of C after q. The representation as a concatenable queue is obtained by splitting the front end up to q off from Q_C and appending the remaining part to Q_A. This can be accomplished in only O(log n) steps by standard routines on the given concatenable queues.

As the computation of q costs no more than O(log n) either by lemma 8.1., the entire construction terminates within O(log n) steps.

□

Observe the similarly of theorem 8.3. with theorem 3.2. (for lc-hulls) and theorem 6.5. (for l-intersections of halfspaces). But note that the "composition" of the separated contours can now be constructed a factor of log n faster than in these previous cases. It will have a succinct effect on the later results, in which theorem 8.3. will be applied.

From theorem 8.3. one may derive yet another algorithm to compute the maximal elements of a set of n points. It will have the interesting property that, after sorting the points by y-coordinate, only O(n) steps are needed to complete the construction.

Theorem 8.4. There is an algorithm to compute the maximal elements of a set of n points in the plane that, after sorting the points by y-coordinate, takes only O(n) steps to complete.

The proof is completely analogous to that of e.g. theorem 6.6.

9. Dynamically maintaining the maximal elements of a plane set (algorithm and applications)

In the previous section we have developed a number of tools that will now be applied. We shall follow the same line of reasoning as before to obtain a fully dynamic maintenance procedure for the maximal elements of a set.

Let us assume that all points currently in the set are stored at the leaves of a balanced binary search tree T, using their y-coordinate as the sorting key. It is tempting again to associate with every internal node α a concatenable queue Q_α containing the maximal elements (in their natural ordering) of the set of points "covered" by α. In Q_α we do keep track of the x-coordinates too,

because of the simultaneous ordering by x- and y-coordinate which maximal elements exhibit. From past experiences we know that the associated information at the nodes must be altered a bit, to obtain a truly efficient dynamic data-structure.

From the decomposability of m-contours as expressed in theorem 8.3. it follows that a structure Q_α as intended can be computed efficiently from the queues associated with the sons γ and δ of α (cf. figure 43). From the proof of theorem 8.3. it is also clear that Q_α is obtained in a very regular fashion from Q_γ and Q_δ, generally by concatenating Q_γ (rather than a portion of it) with a tail part of Q_δ. This yields a situation very much like that for e.g. lc-hulls as developed in Section 3.

It follows that we must augment T to obtain a tree T^* in which with every node α is associated the portion of Q_α (kept as a queue) that was not used in building the m-contour associated with its father. Because of the very special properties of this problem, it implies that at least at one of the sons an empty structure remains (compare the proof of 8.3.). The m-contour of the complete set will be available in one piece at the root of T^*. The maintenance of T^* is programmed in very much the same way as indicated in Section 4.

Theorem 9.1. The maximal elements of a set of points in the plane can be maintained at a cost of only $O(\log^2 n)$ steps per insertion and deletion, where n denotes the current number of elements in the set.

Proof

Given the structure of T^*, procedures DOWN and UP can be defined for it as we did in Section 4. Insertions and deletions are processed using these routines in completely the same way as described in the proof of theorem 4.4. The time analysis carries over too, with J_1 replaced by the "log n" bound implied by theorem 8.3. We conclude that the necessary updates of the structure after each insertion or deletion can be made in $O(\log^2 n)$ steps total.

□

Hence the paradigm of "decomposability" has led us to an efficient dynamic structure for yet another problem. We mention a number of applications of theorem 9.1. which are usually easy to derive.

A fundamental problem in this context is that one would like to maintain a set in the plane and be able to answer queries of the sort "is x a maximal element of the current set" efficiently. It so happens that such queries are

decomposable in the sense of Bentley [3], yet none of the standard dynamizations of static solutions will result in the low bounds we obtain here.

Theorem 9.2. One can dynamically maintain a set of n points in the plane at a cost of only $O(\log^2 n)$ per insertion and deletion, such that queries of the form "is x a maximal element of the set" can be answered in only $O(\log n)$ time.

Proof

Use the structure implied by theorem 9.1. To find out whether a point x belongs to the current contour of maximal elements one merely needs to search down the concatenable queue associated with the root.

□

A number of other applications are best formulated in terms of the concept of "dominance".

Definition. Given a set of points B, a point x is dominated "in" B if and only if there is a $y \in B$ such that $x < y$. A set A is said to be dominated by B if every $x \in A$ is dominated in B.

Clearly a point x is dominated in B if and only if it is not maximal in B. Thus the (decomposable!) searching problem of whether an arbitrary point is dominated in the current set can be dynamized within the same bounds as given in theorem 9.2. A set of points A is dominated by a similar set B just when no point of A is maximal in $A \cup B$. It takes a little work, but the information can be maintained along with the two sets.

Theorem 9.3. One can maintain two sets A and B in the plane such that insertions and deletions take at most $O(\log^2 n)$ steps each (where n is the total number of points) and the information of whether A is dominated by B is maintained at no extra charge.

Proof

Do not maintain A and B separately, but maintain the maximal elements of $A \cup B$ according to the method of theorem 9.1. and keep track of the elements of A in it (if any) by a double-linked sub-list of the current contour. To manage it, one must keep track of these sublists in all queues Q_α associated with nodes α in T, i.e., in the contiguous pieces of these queues kept around.

The internal nodes of these queues must also keep a flag indicating whether they are any elements of A in the subtree below. It will enable us to modify the algorithms for splitting a queue in O(log n) steps, such that with little extra effort the embedded sub-list of elements of A can be split too. The ordinary algorithms for concatenating or updating queues can be modified also, such that the extra information is correctly maintained at the nodes.

It is easily verified that in the construction in the proof of theorem 8.3. and in the algorithms implied by DOWN and UP for processing insertions and deletions the embedded lists can be managed within the same time-bounds. To determine whether A is dominated by B it suffices to see whether the embedded list of A-elements in the m-contour of A ∪ B, as it is available at the root of T*, contains at least one element. This obviously takes only O(1) time.

□

It should be noted that the proof of theorem 9.3. shows more than is stated. It indicates that one can keep track of the "contribution" of a particular subset to the maximal elements of the entire set and even list the contributed elements, when required, in the exact order in which they occur on the contour.

10. Conclusion

We have presented efficient data structures and algorithms for processing insertions and deletions in sets in the plane, while maintaining the correct shape of some derived configuration at the same time. We have obtained fully dynamic structures and algorithms for the convex hull of a set of points, for the common intersection of a set of halfspaces and for the maximal elements of a set of points again. In all these problems we have followed a very similar line of reasoning and have obtained dynamizations based on one technique, that happens to apply in all these instances.

The main ingredient in all problems is a suitable notion of "decomposability" of the configuration that must be maintained. Having identified it and observing that "neighboring" configurations contribute localized portions to the configuration for the union, a same technique of cutting configurations and only maintaining the left-over portions at internal nodes of a covering balanced tree is applied to achieve the high efficiency for updating algorithms. The efficiency of "composing" configurations after a decomposition of the set determines much of the efficiency of the dynamizations.

We expect that the same techniques we have developed here will be of use to obtain a good number of very efficient dynamic solutions to other problems

in computational geometry, viz. in maintaining configurations. But the proper notion of decomposability may have to be invented time and again for every different problem, as it seems very difficult to capture it adequately. Yet we hope to have made a step in the right direction to let dynamizations of any sort desired be available by acts of standard "engineering".

11. References

[1] Aho, A.V., J. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms, Addison-Wesley, Reading, Mass. (1974).

[2] Barnett, V., The ordering of multivariate data (with discussion), J. Roy. Stat. Soc. (A), 139 (1976) 318-354.

[3] Bentley, J.L., Decomposable searching problems, Inform. Proc. Lett. 8 (1979) 244-251.

[4] Bentley, J.L. and M.I. Shamos, Divide and conquer for linear expected time, Inform. Proc. Lett. 7 (1978) 87-91.

[5] Bentley, J.L. and M.I. Shamos, A problem in multivariate statistics: algorithm, data structure and applications, Techn. Rep. CMU-CS-78-110, Carnegie Mellon University (1978).

[6] Blum, M., R.W. Floud, V.R. Pratt, R.L. Rivest and R.E. Tarjan, Time bounds for selection, J. Comput. Syst. Sci. 7 (1972) 448-461.

[7] Brown, K.Q., Fast intersection of halfspaces, Techn. Rep. CMU-CS-78-129, Carnegie Mellon University (1978).

[8] Chazelle, B. and D.P. Dobkin, Detection is easier than computation, Proc. 12th Annual ACM Symp. Theory of Computing, Los Angeles, 1980, 146-153.

[9] Eddy, W.F., A new convex hull algorithm for planar sets, ACM Trans. Math. Software 3 (1977) 398-403, 411-412.

[10] Graham, P.J., An efficient algorithm for determining the convex hull of a finite planar set, Inform. Proc. Lett. 1 (1972) 132-133.

[11] Green, P.J., Convex hulls in the analysis of bivariate data, University of Durham, Sheffield, UK, paper read at the Conference "Looking at Multivariate Data", 1980.

[12] Green, P.J. and B.W. Silverman, Constructing the convex hull of a set of points in the plane, Computer J. 22 (1979) 262-266.

[13] Hadwiger, H. and H. Debrunner, Combinatorial geometry in the plane, Holt, Rinehart and Winston, New York (1963).

[14] Huber, P.J., Robust statistics: a review, Annals Math. Statistics 43 (1972) 1041-1067.

[15] Jarvis, R.A., On the identification of the convex hull of a finite set of points in the plane, Inform. Proc. Lett. 2 (1973) 18-21.

[16] Kung, H.T., F. Luccio and F.P. Preparata, On finding the maxima of a set of vectors, J. ACM 22 (1975) 469-476.

[17] Lee, D.T. and F.P. Preparata, An optimal algorithm for finding the kernel of a polygon, J. ACM 26 (1979) 415-421.

[18] Overmars, M.H. and J. van Leeuwen, Dynamically maintaining configurations in the plane, Proc. 12th Annual ACM Symp. Theory of Computing, Los Angeles, 1980, 135-145.

[19] Preparata, F.P., An optimal real-time algorithm for planar convex hulls, C. ACM 22 (1979) 402-405.

[20] Preparata, F.P. and S.J. Hong, Convex hulls of finite sets of points in two and three dimensions, C. ACM 20 (1977) 87-93.

[21] Reingold, E.M., J. Nievergelt and N. Deo, Combinatorial algorithms: theory and practice, Prentice-Hall, Englewood Cliffs, NJ (1977).

[22] Saxe, J.B. and J.L. Bentley, Transforming static data structures to dynamic structures, Conf. Rec. 20th Annual IEEE Symp. on Foundations of Computer Science, San Juan, Puerto Rico, 1979, 148-168.

[23] Shamos, M.I., Geometric complexity, Proc. 7th Annual ACM Symp. on Theory of Computing, Albuquerque, May 1975, 224-233.

[24] Shamos, M.I. Geometry and statistics: problems at the interface, in: J.F. Traub(ed), Recent results and new directions in algorithms and complexity, Acad. Press, New York (1976) 251-280.

[25] Shamos, M.I., Computational geometry, Ph. D. Thesis Yale University, 1978(to be published).

[26] Shamos, M.I. and D. Hoey, Geometric intersection problems, Conf. Rec. 17th Annual IEEE Symp. on Foundations of Computer Science, Houston, Oct. 1976, 208-215.

[27] Tukey, J.W., referred to in [24] p. 267.

[28] van Emde Boas, P., On the $\Omega(n \log n)$ lowerbound for convex hull and maximal vector determination, Inform. Proc. Lett. 10 (1980) 132-136.

[29] Wirth, N., Algorithms + data structures = programs, Prentice Hall, Englewood Cliffs, NJ (1976).

[30] Yao, A.C-C., A lower bound to finding convex hulls, STAN-CS-79-33, Computer Science Dept., Stanford University (1979).

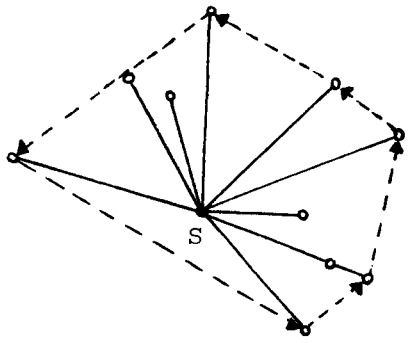


figure 1

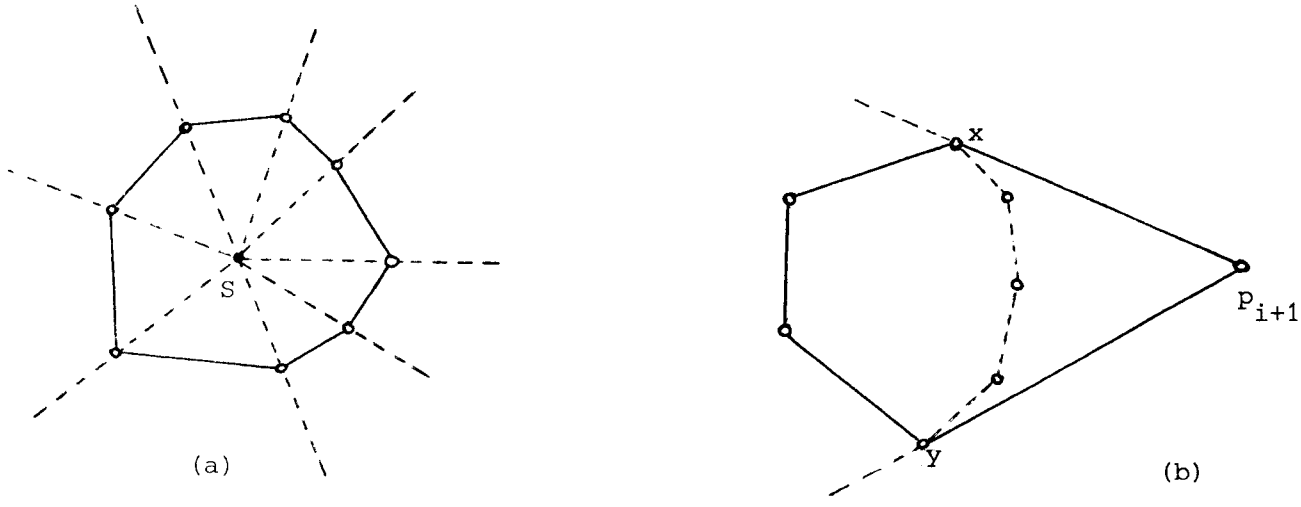


figure 2

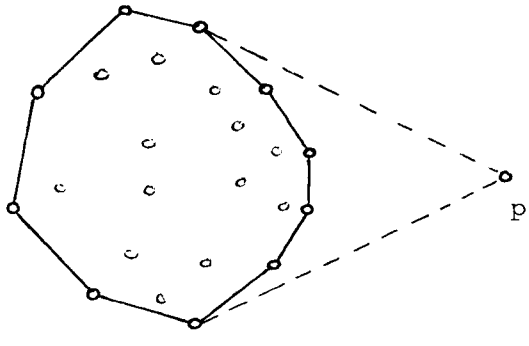


figure 3

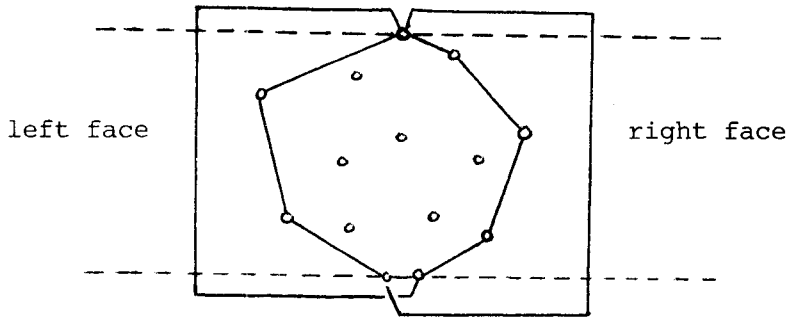
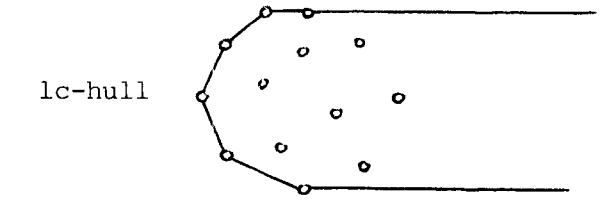


figure 4



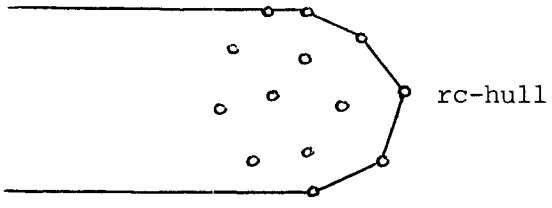


figure 5

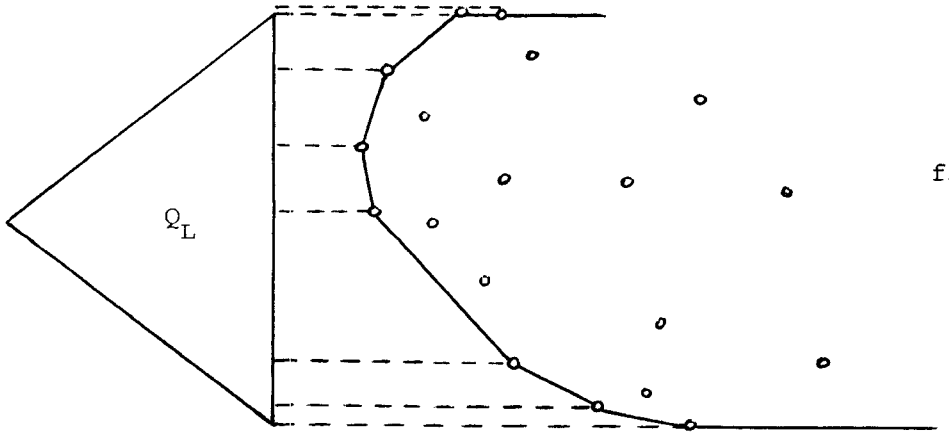


figure 6

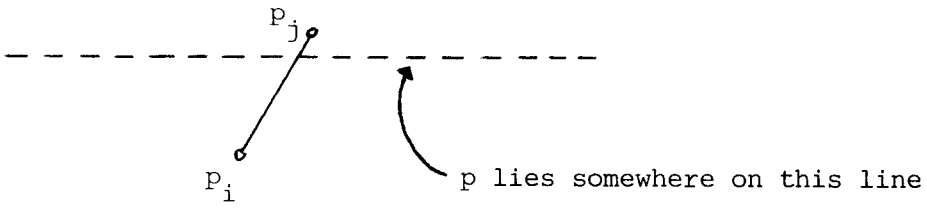


figure 7

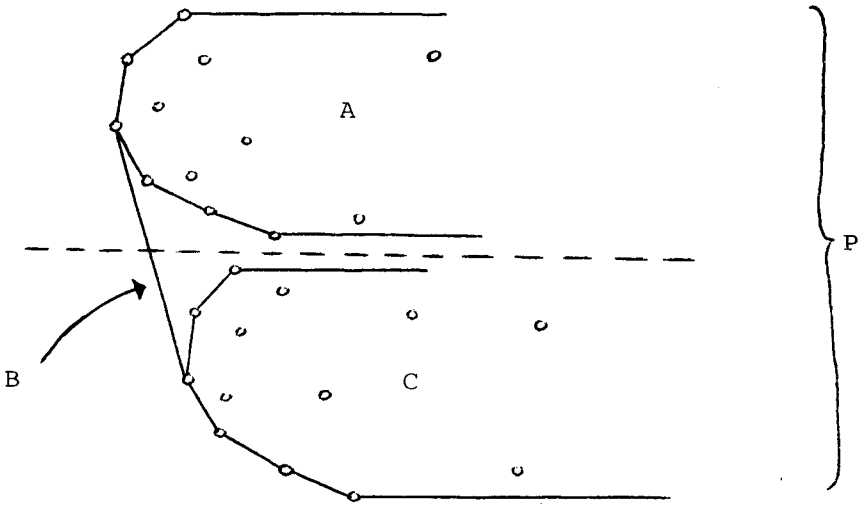


figure 8

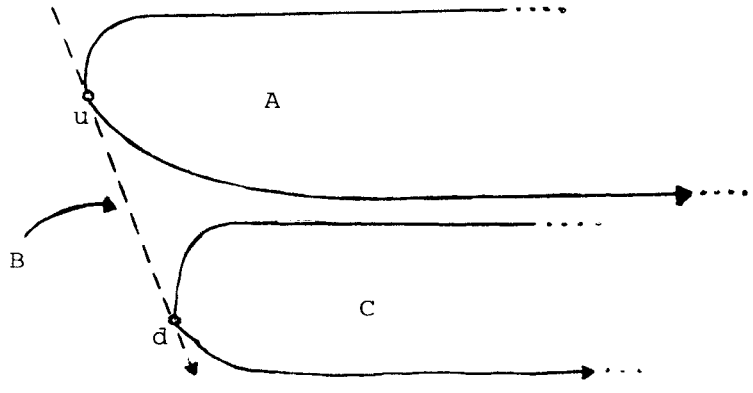


figure 9

figure 10

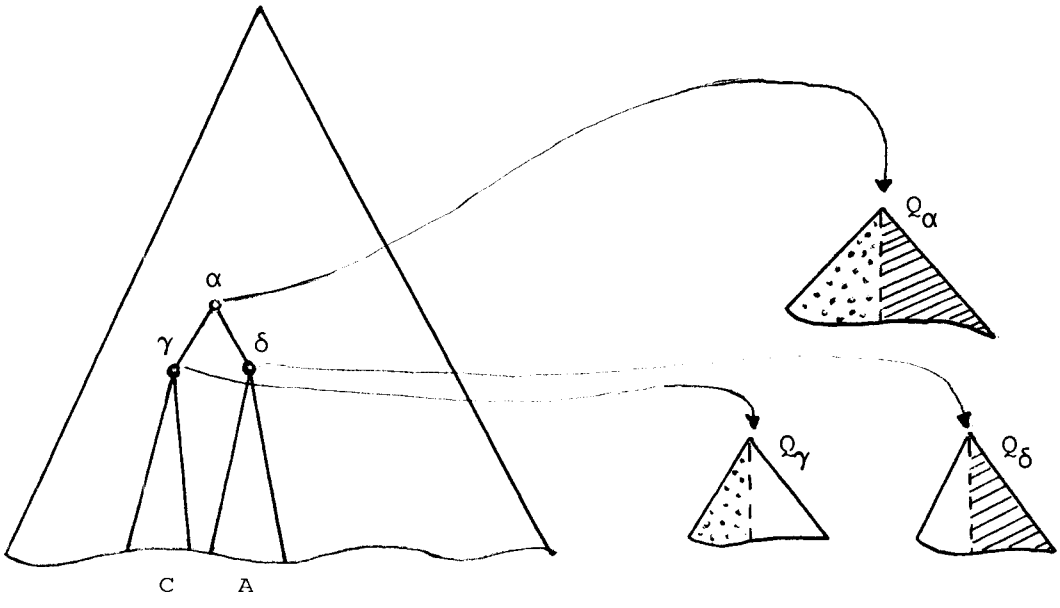


figure 11

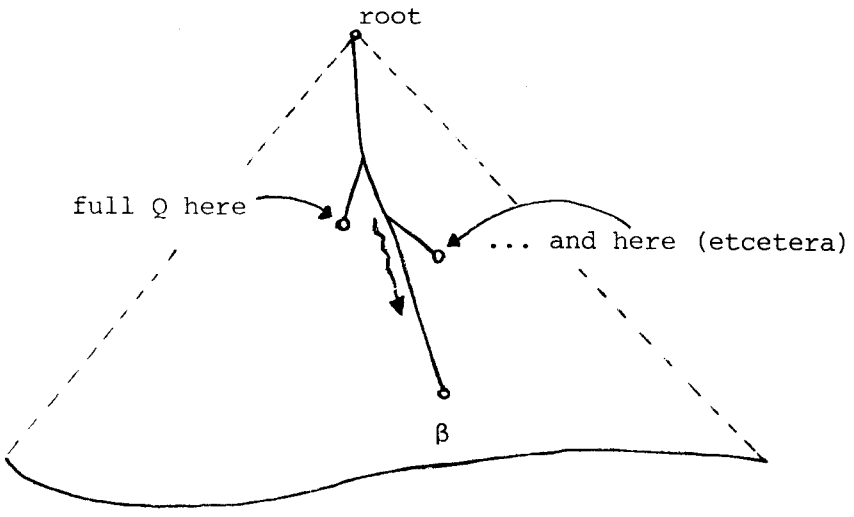


figure 12

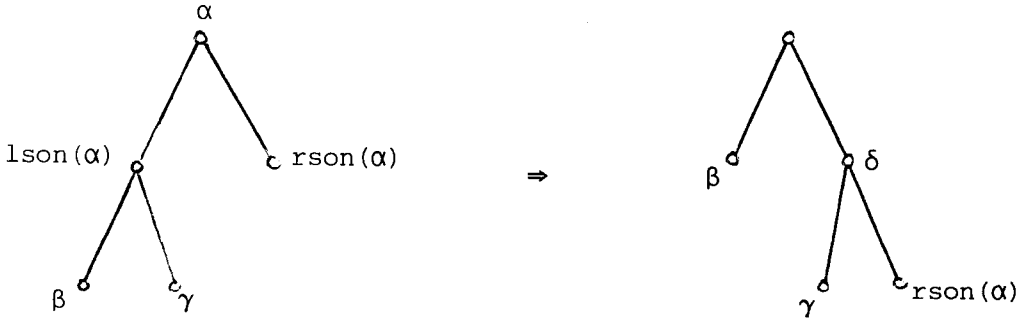
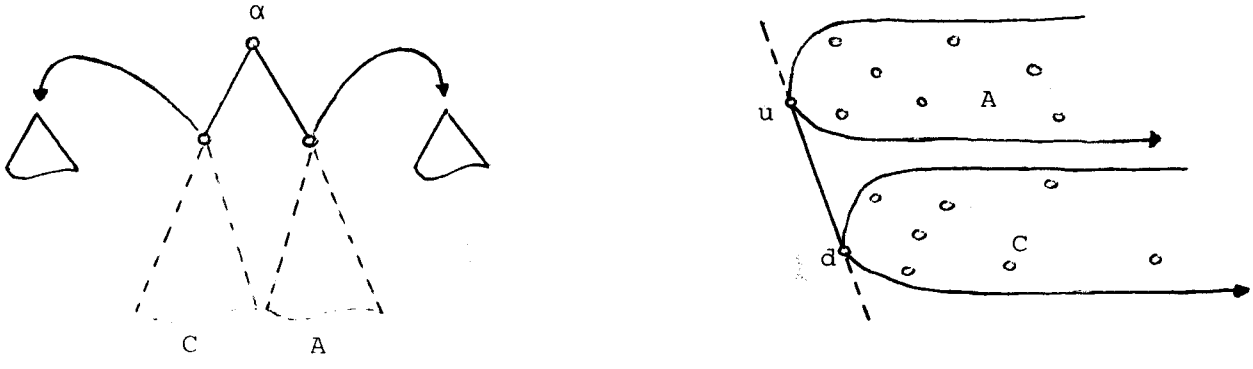


figure 13



(a) (b)

figure 14

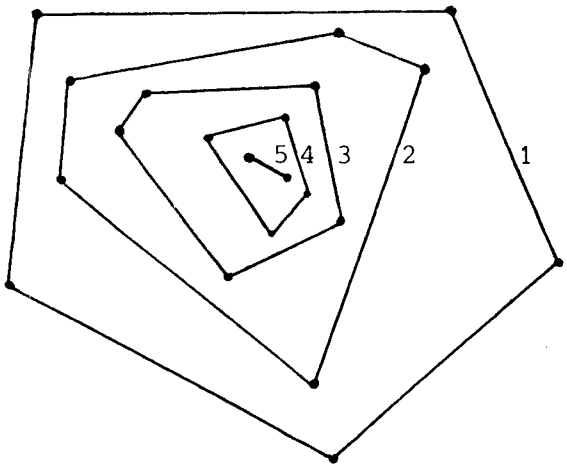


figure 15

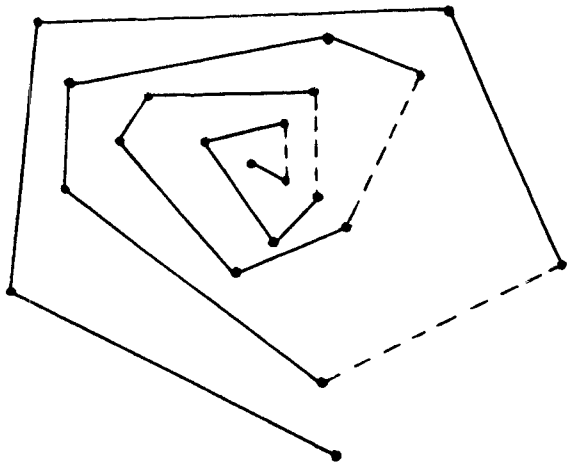


figure 16

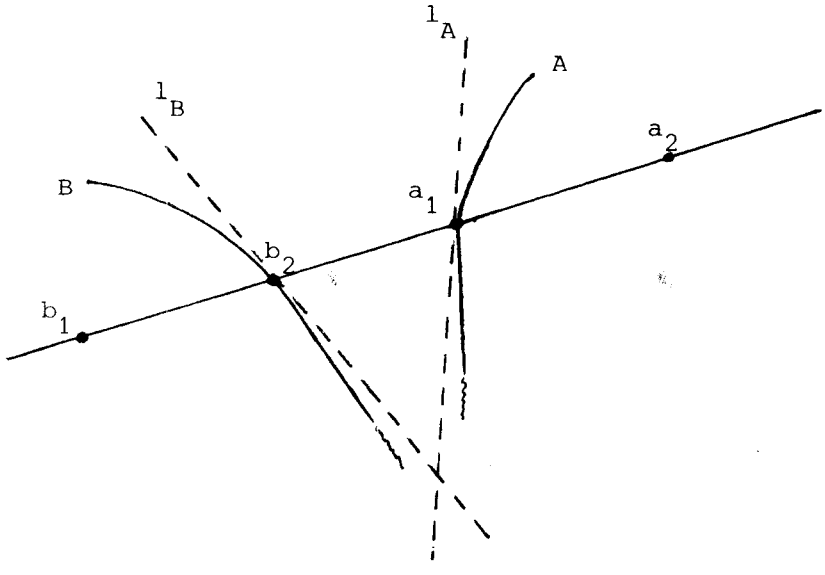


figure 17

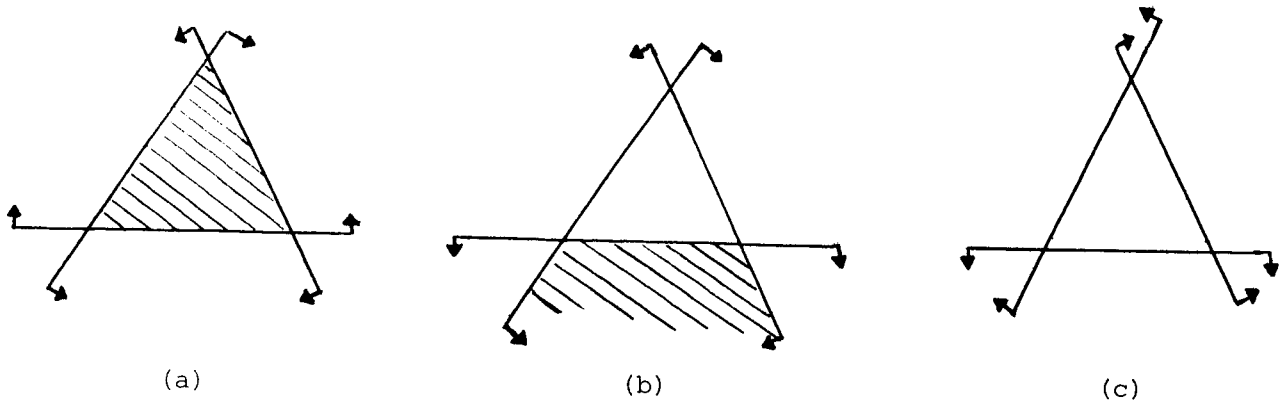


figure 18

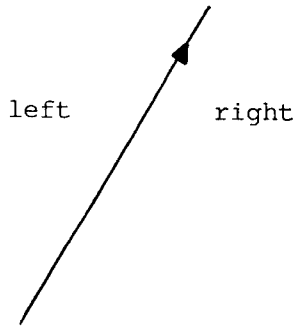


figure 19

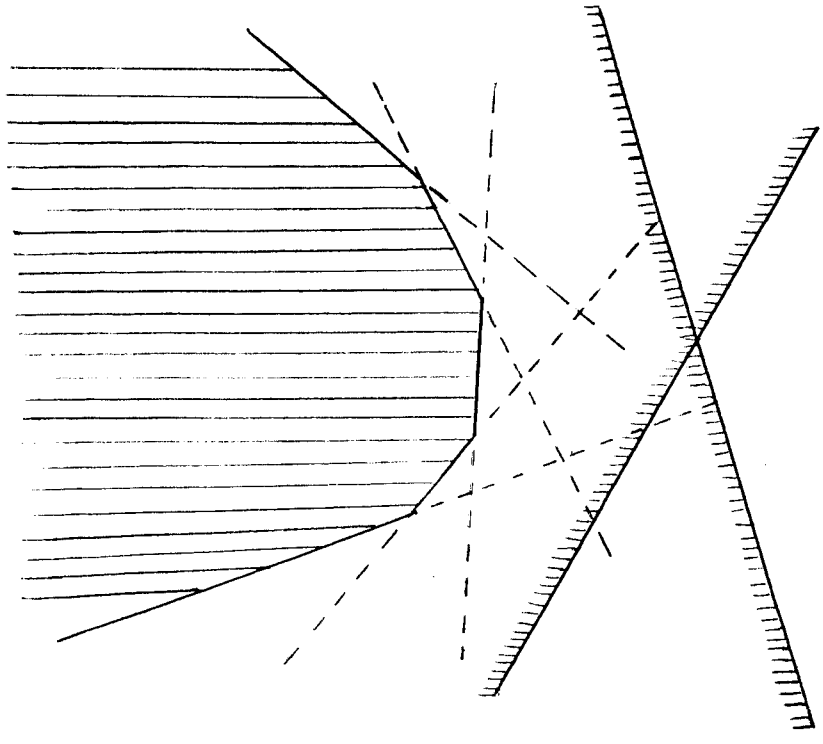


figure 20

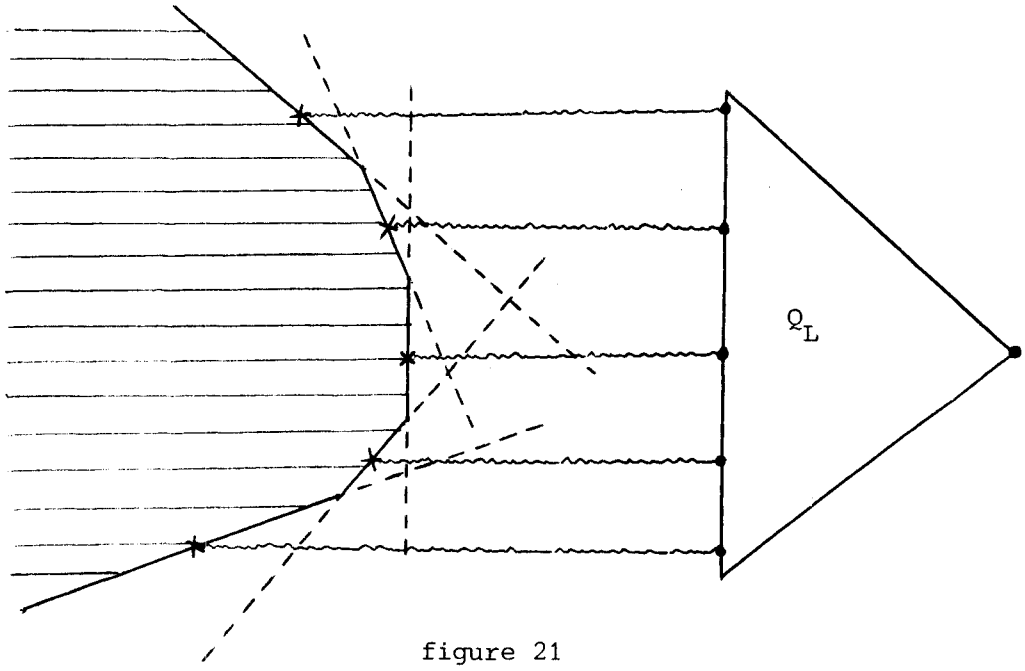


figure 21

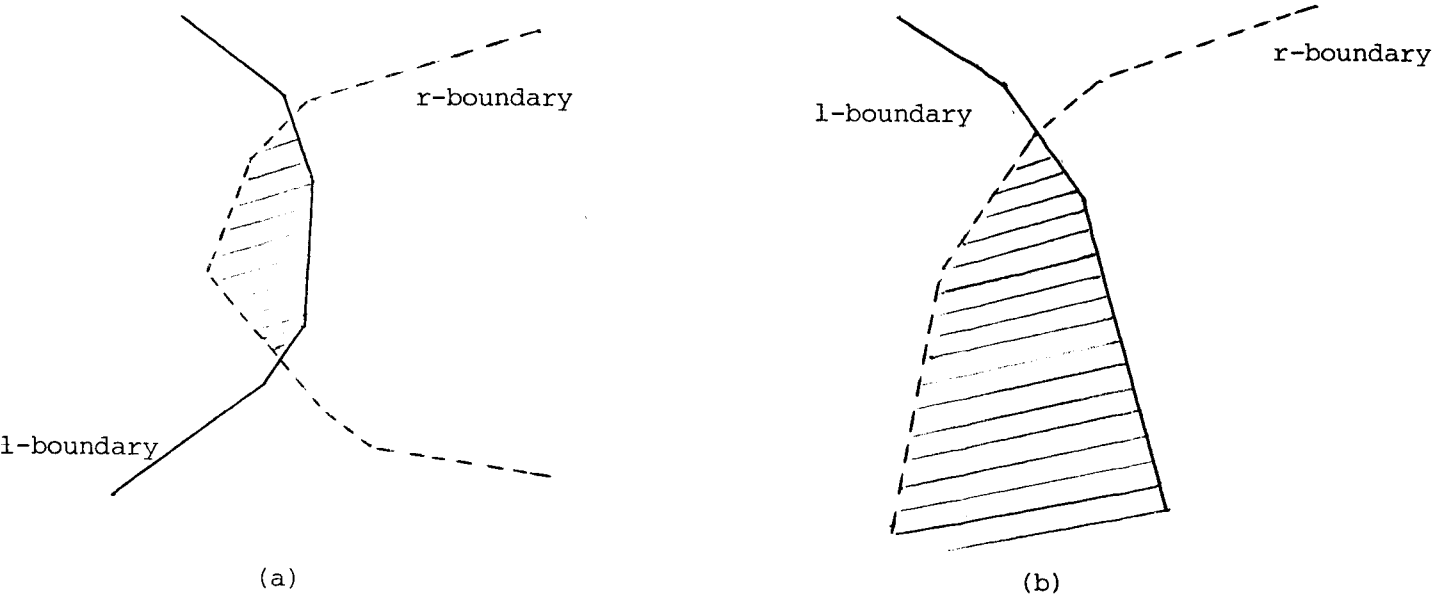


figure 22

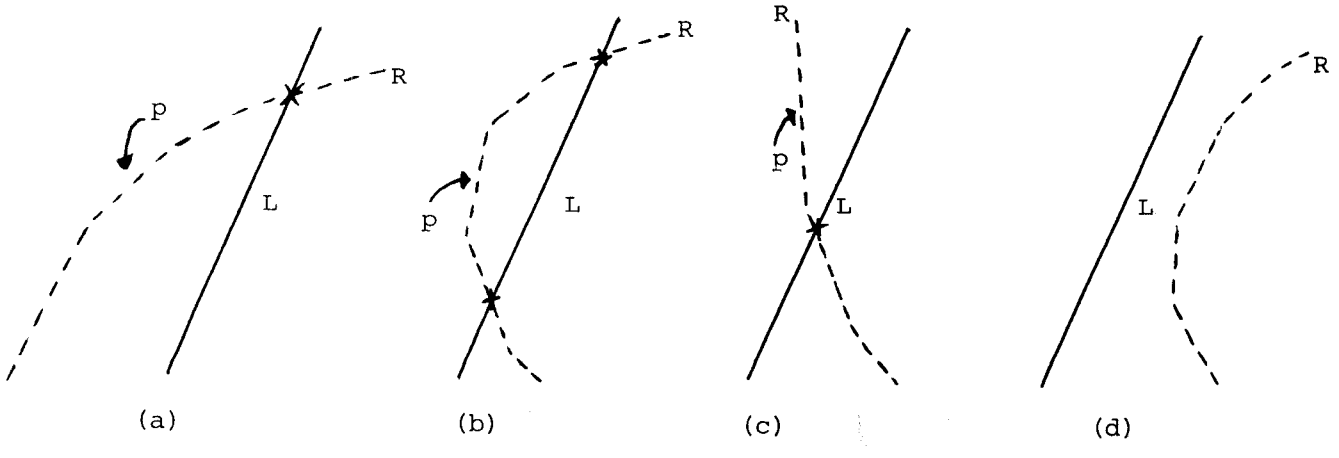


figure 23

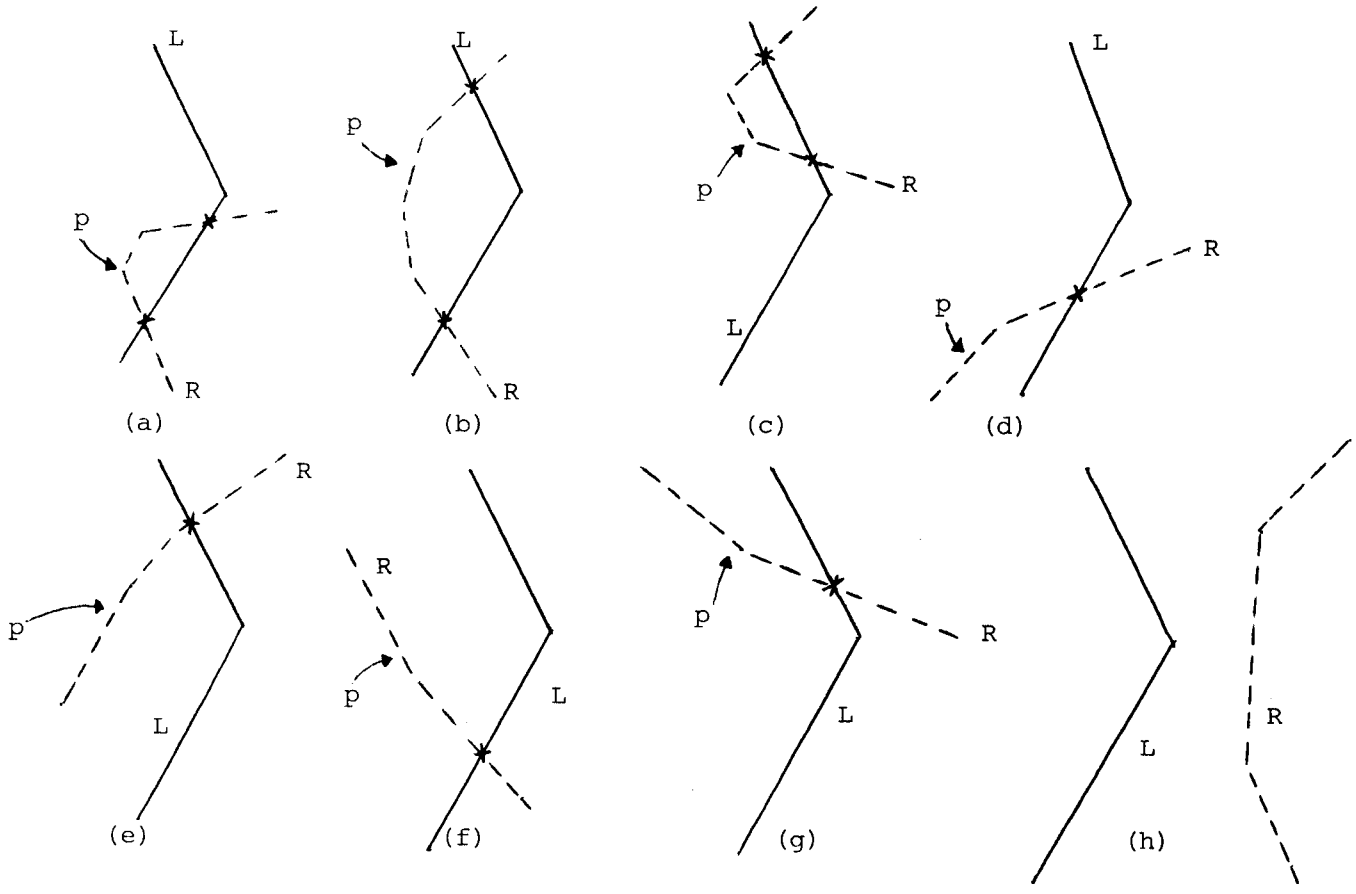


figure 24

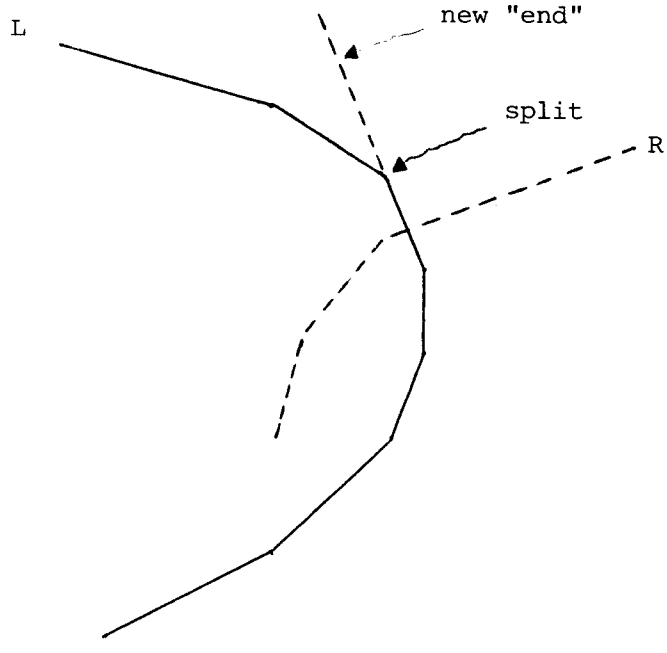


figure 25

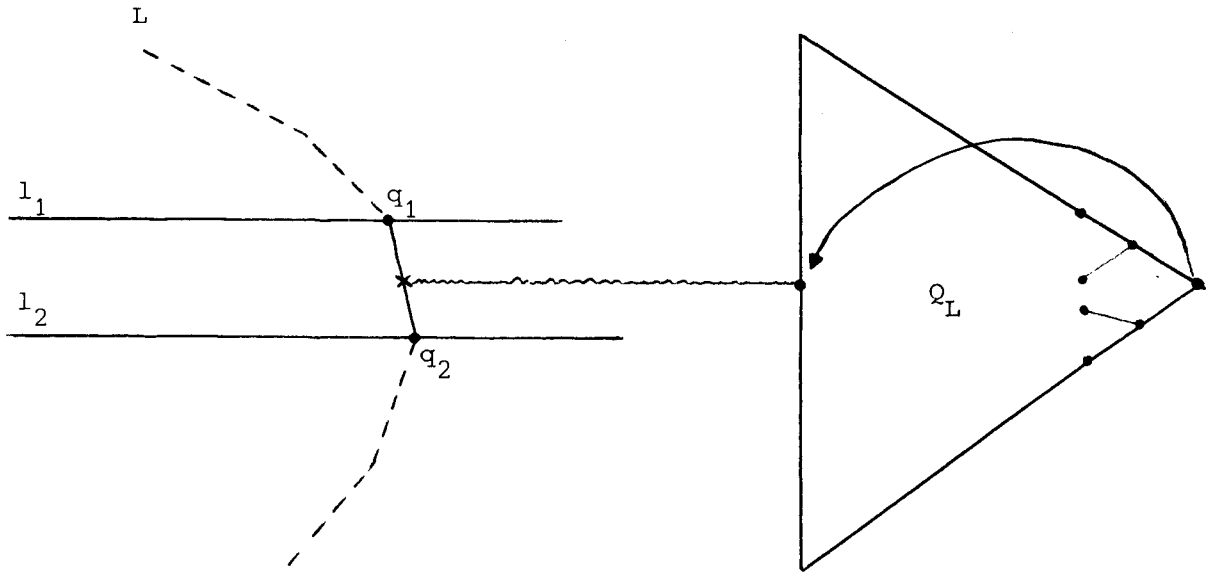


figure 26

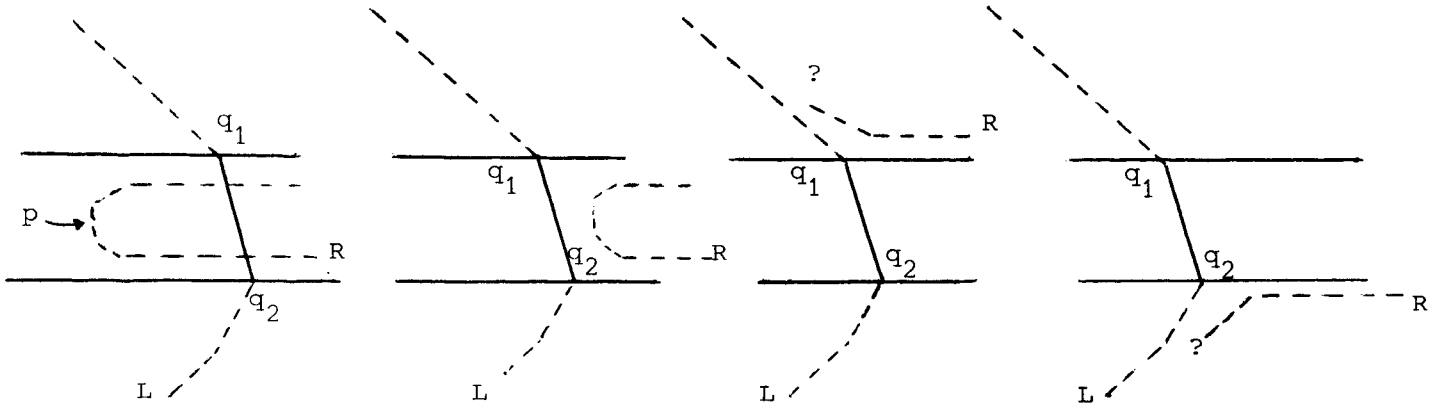


figure 27

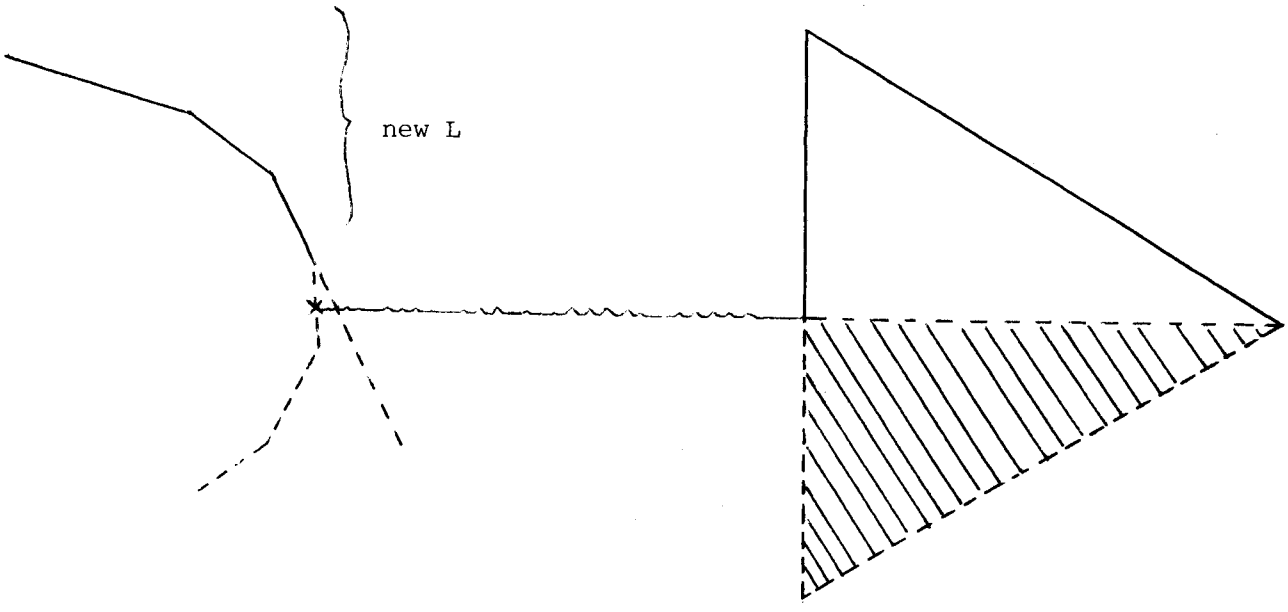


figure 28

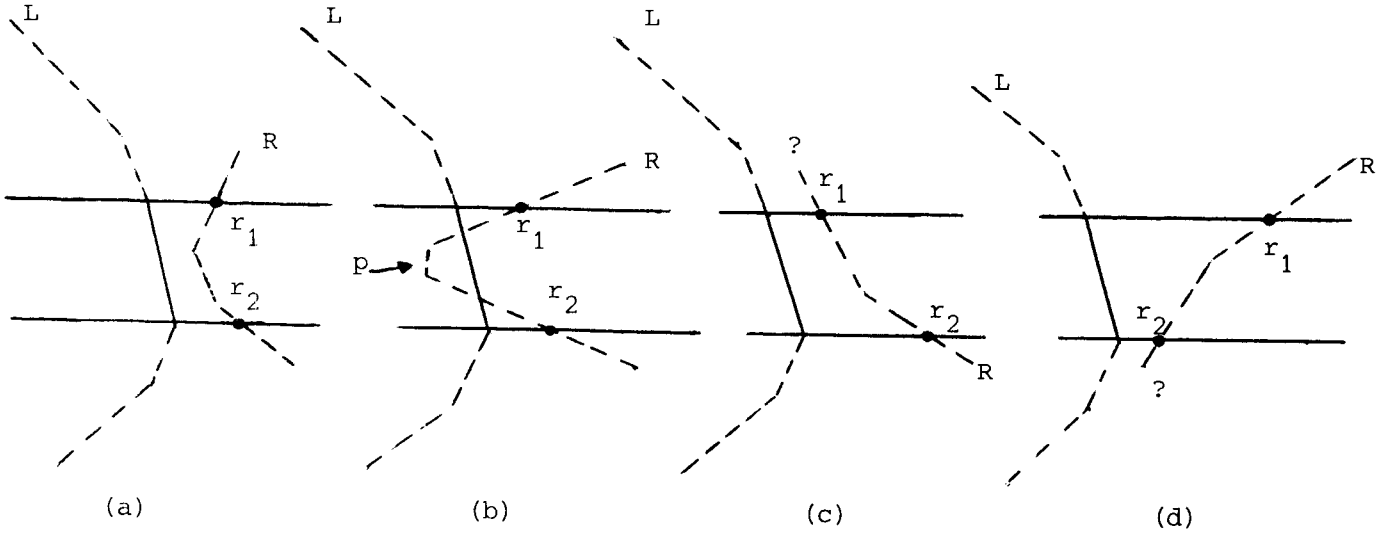


figure 29

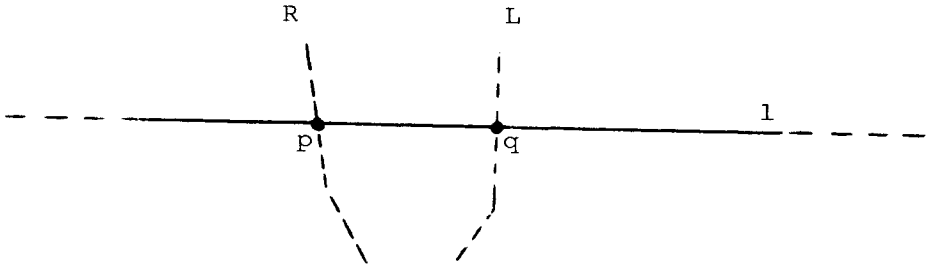


figure 30

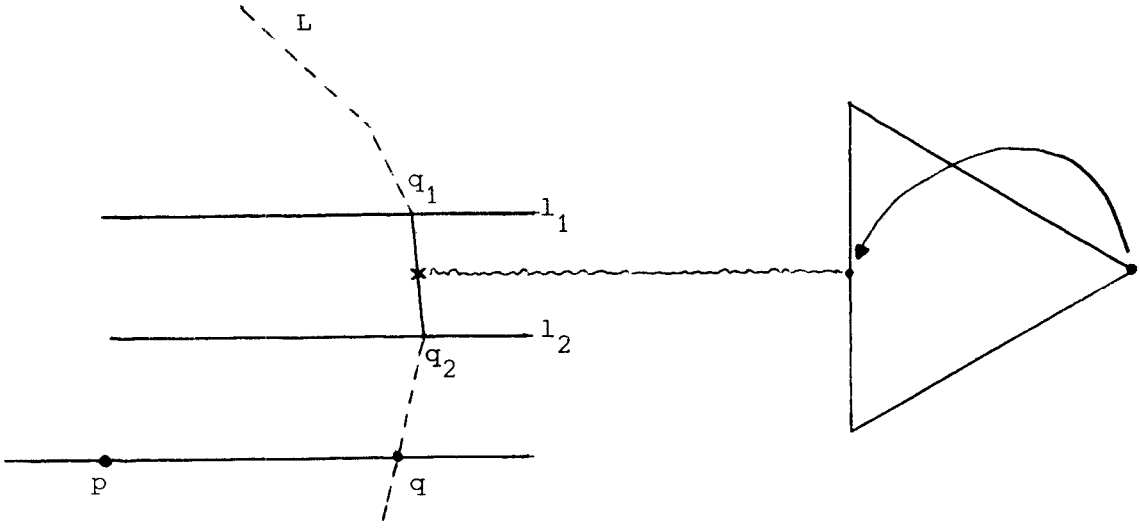


figure 31

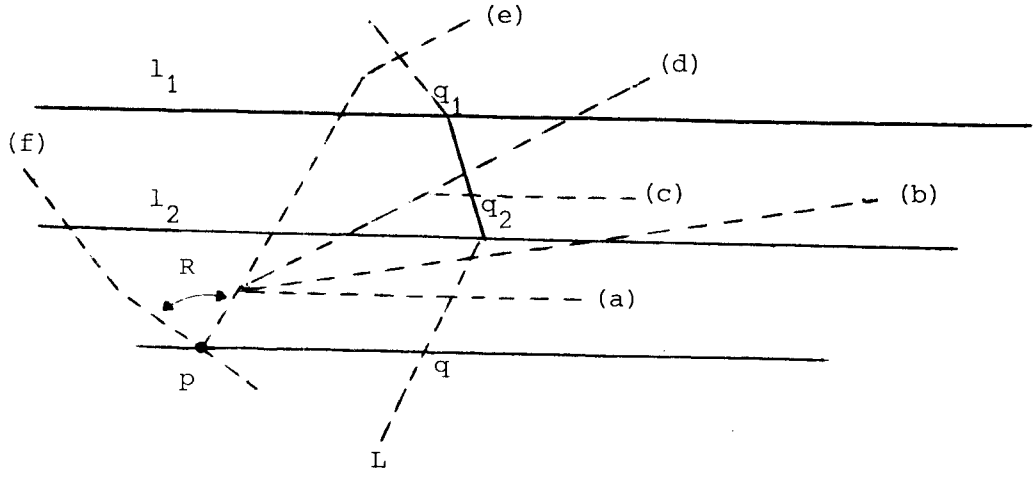


figure 32

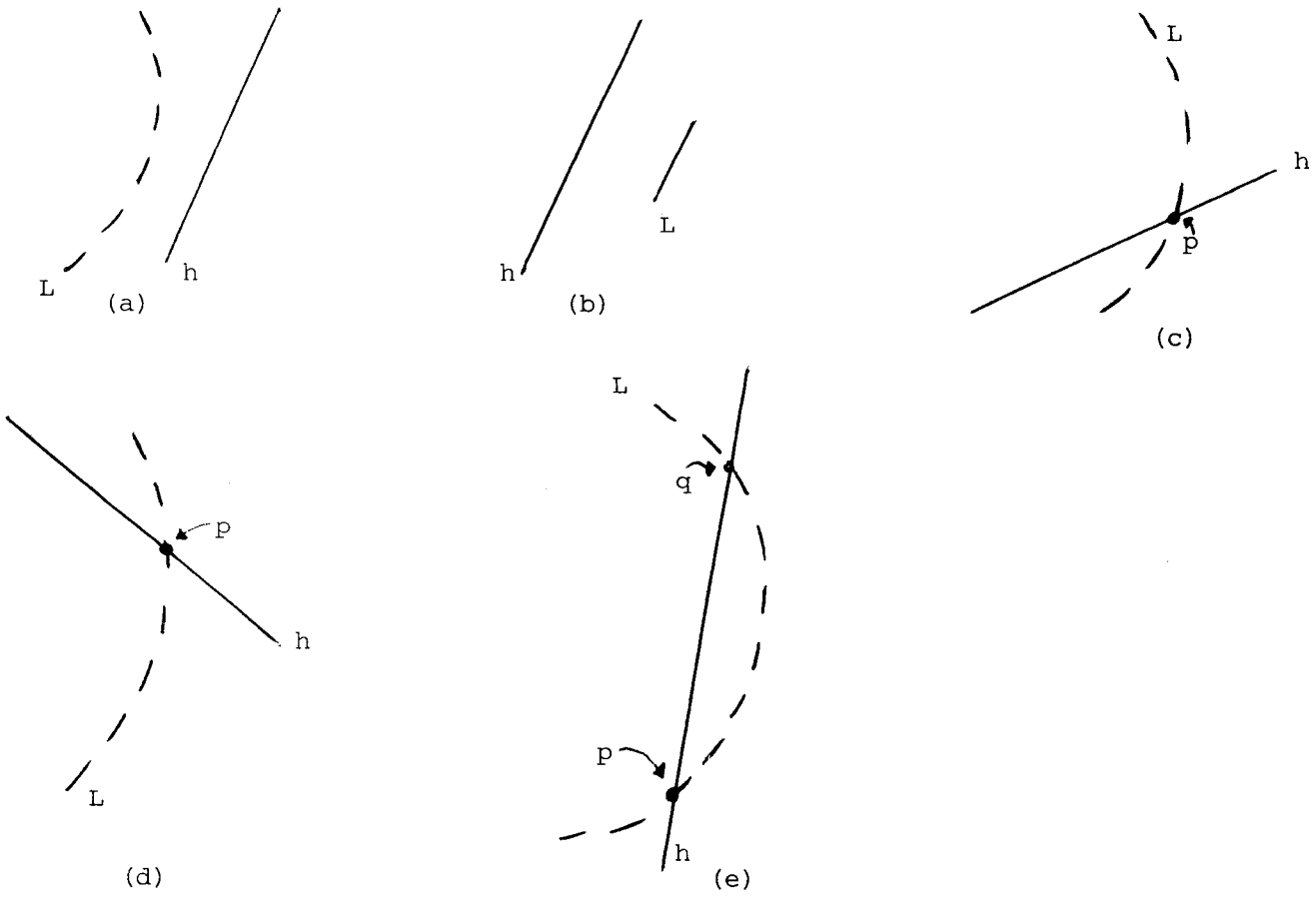


figure 33

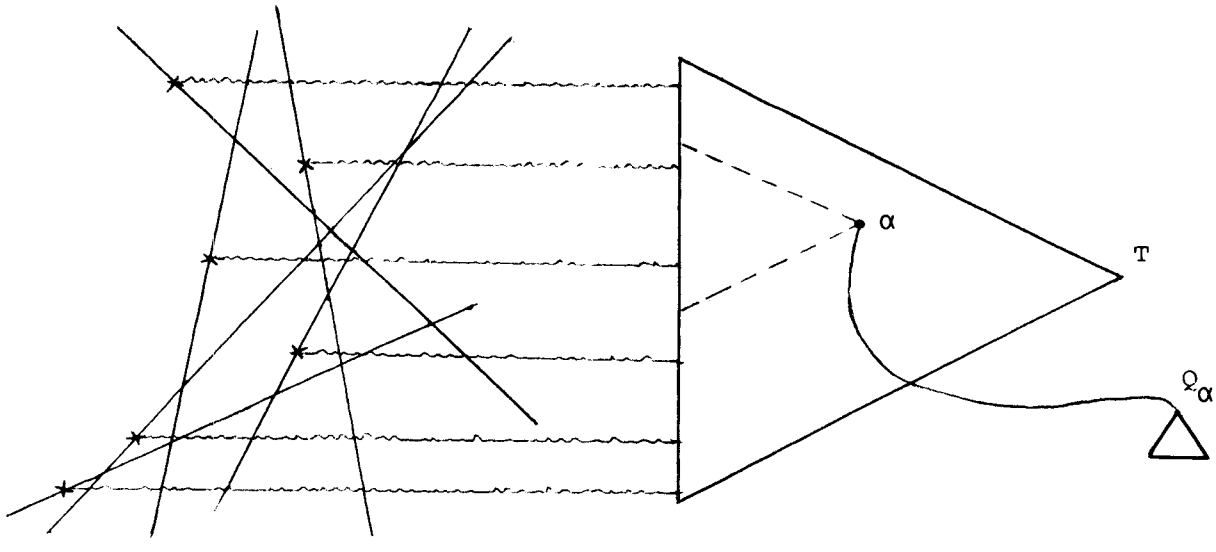


figure 34

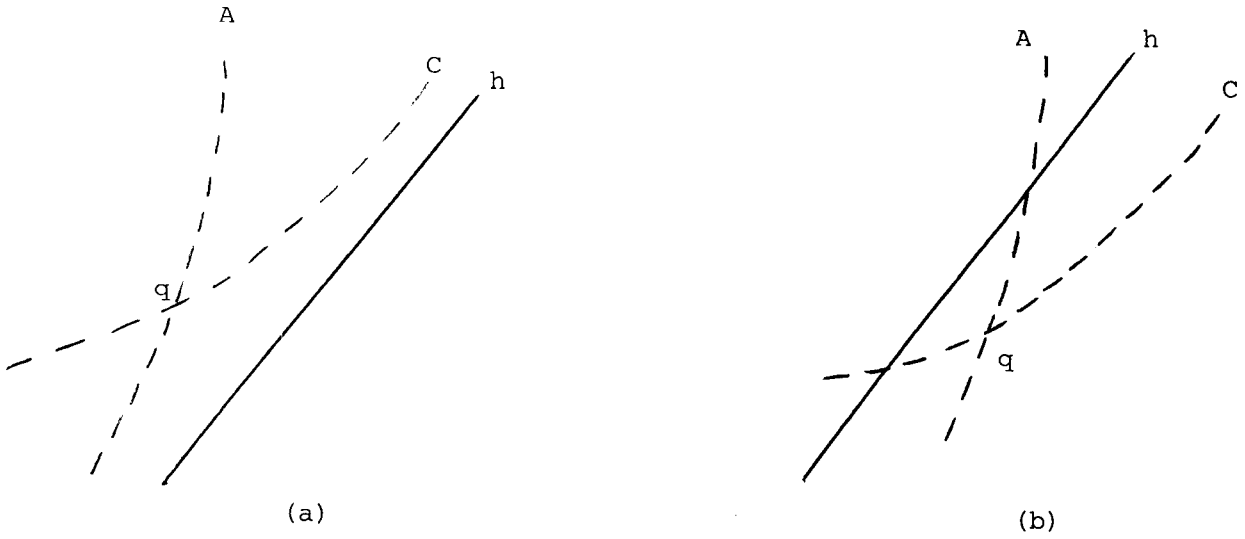


figure 35

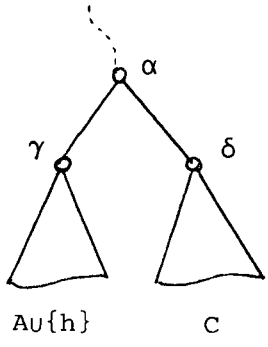


figure 36

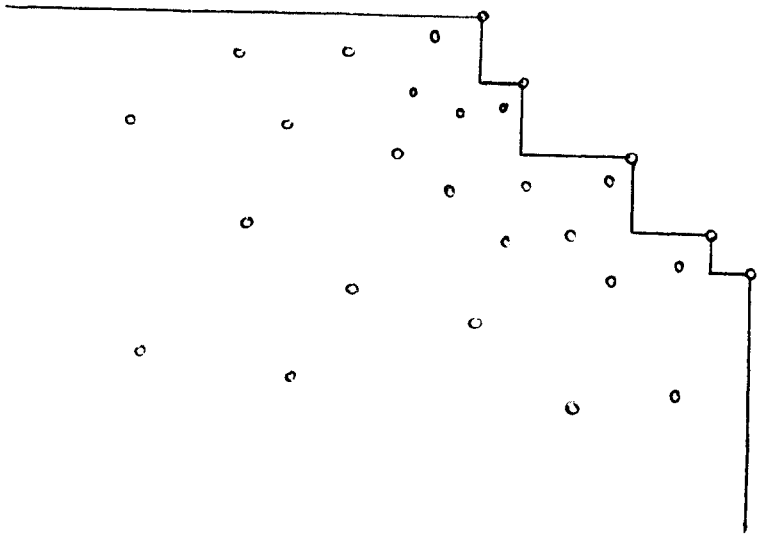


figure 37

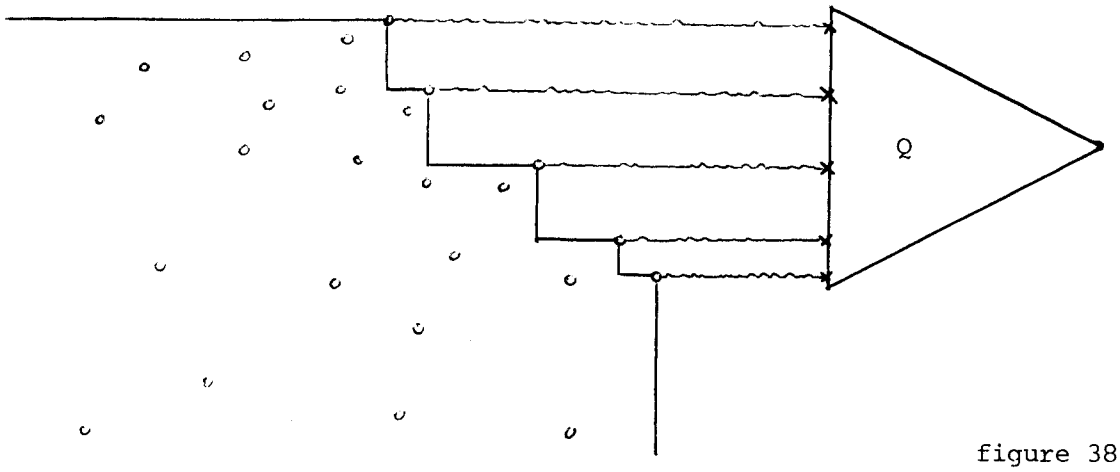


figure 38

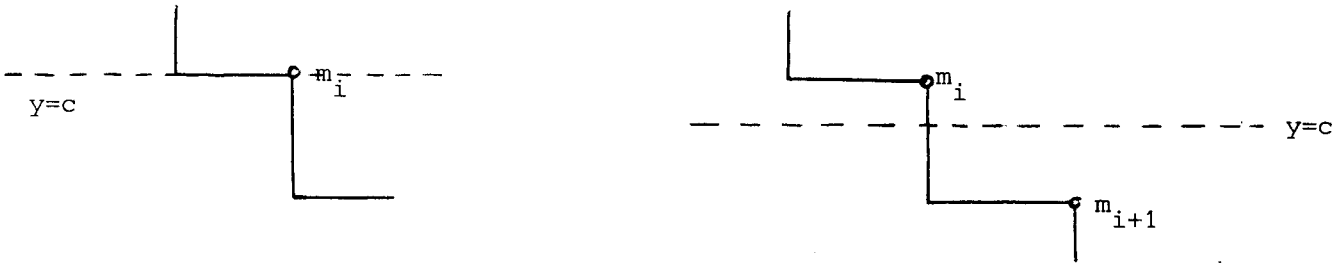


figure 39

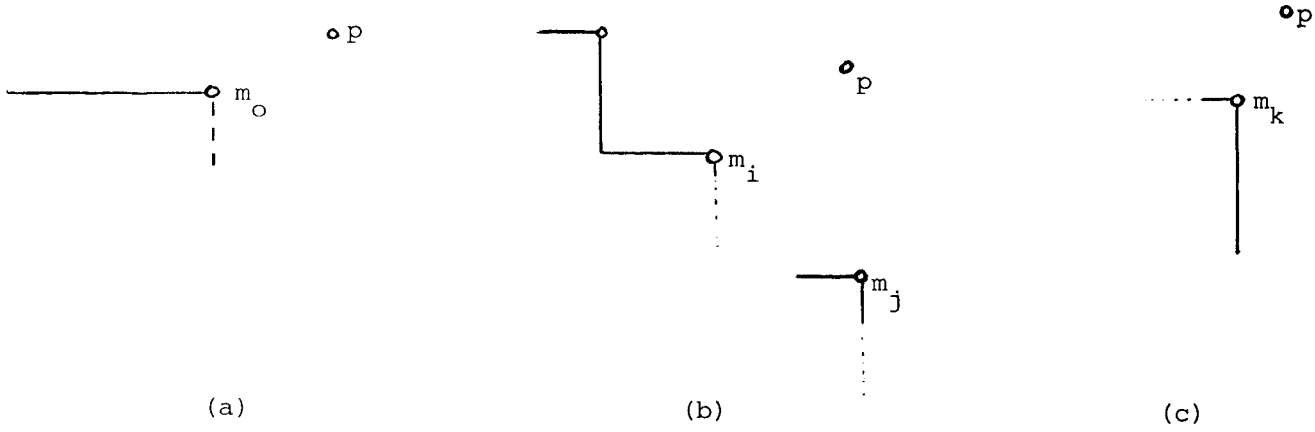


figure 40

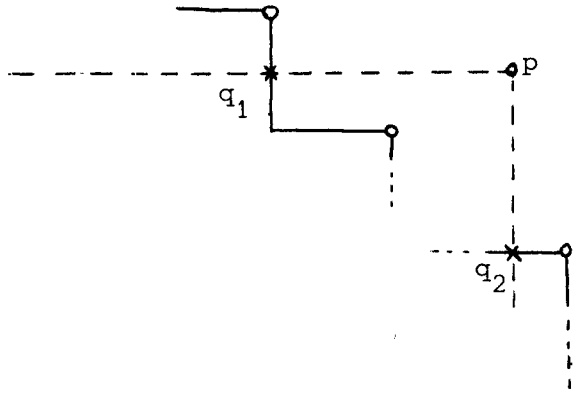


figure 41

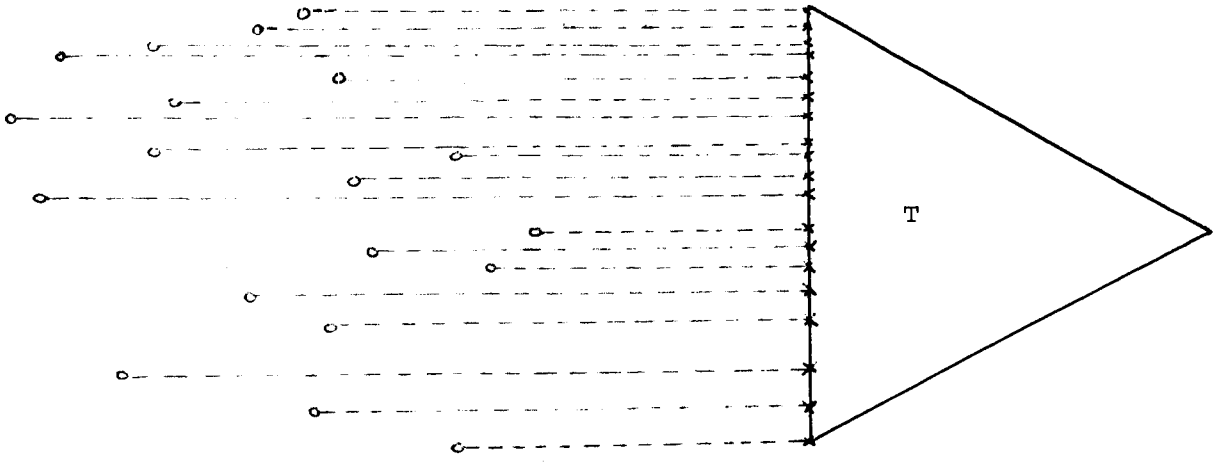


figure 42

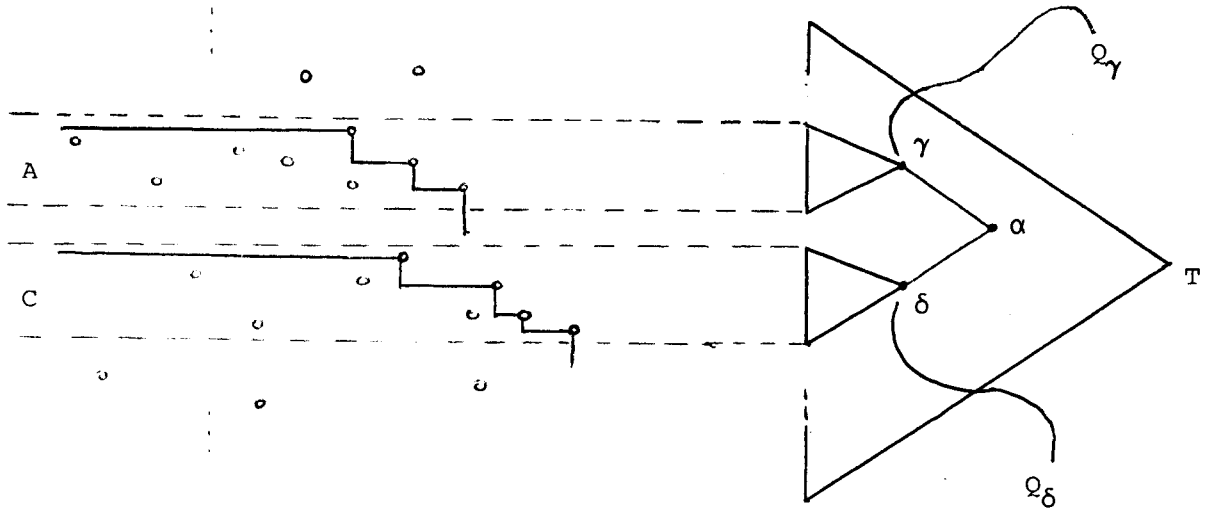


figure 43

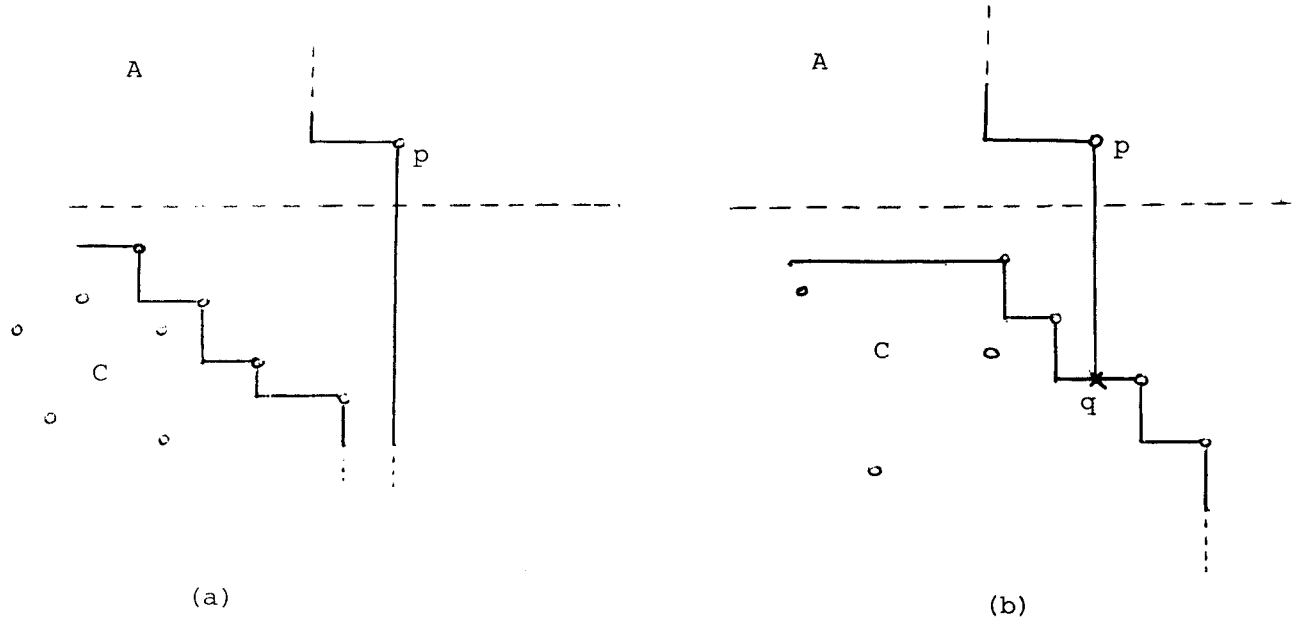


figure 44