JOURNAL OF COMPUTER AND SYSTEM SCIENCES **23**, 166–204 (1981)

Maintenance of Configurations in the Plane

MARK H. OVERMARS* AND JAN VAN LEEUWEN

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

Received September 8, 1980; revised March 2, 1981

For a number of common configurations of points (lines) in the plane, we develop data structures 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 attain. 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 $O(\log^2 n)$ steps 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 $O(n \log^2 n)$ 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 closely related to the existence of divide-and-conquer solutions.

1. INTRODUCTION

Computational geometry (cf. Shamos [24, 26]) 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 data structures 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.

The problem of converting the intrinsically static data structures 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, termed "decomposable searching problems," which are particularly amenable to dynamization. In Bentley [3] and in Saxe and Bentley [23] 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 becoming intolerably high.

* The work of this author is supported by the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

0022-0000/81/050166–39$02.00/0

While the theory as it stands is applicable to a wide variety of "point problems," Saxe and Bentley [23, 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 show that insertions and deletions of objects can be performed in only $O(\log^2 n)$ 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 also has never been discussed before. An extensive list of applications is discussed in various intermediate sections, some of immediate interest to such areas as computational statistics (cf. Shamos [25]). For example, we shall present a method to maintain two sets of points in the plane at a cost of only $O(\log^2 n)$ 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 $O(\log n)$ 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 takes advantage of the existence of "similar" divide-and-conquer solutions for the construction of the configurations at hand. Some of the searching problems we consider, such as containment in the common intersection of a set of halfspaces, are even 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 $p_1,..., p_n$ in the plane [4, 9, 10, 12, 15, 21]. 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 (Fig. 1). Since it always has to sort, Graham's algorithm will be tied to an $\Omega(n \log n)$ worst case lower bound. On the other hand, the very fact that we normally want to obtain

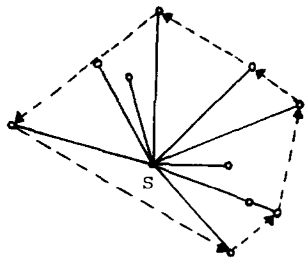


FIGURE 1

the ordered contour of the convex hull implies that sorting must be implicit in any algorithm and the $\Omega(n \log n)$ worst case lower bound applies to all of them which deliver a convex hull in such terms [26]. Even if we merely want to mark which of the p_i are hull-points (duplicates allowed) and do not care about the actual contour at all, then an $\Omega(n \log n)$ worst case lower bound can still be shown [29], even in a quadratic decision tree model [31].

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 [25] 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 $n \log n$ lower bound, updates of the convex hull due to the addition of a single point will cost at least $\Omega(\log n)$ on the average. Preparata [20] 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 to which it belongs (which can be found by binary search around S, see Fig. 2a). When p_{i+1} lies in the interior no update is needed. When p_{i+1} lies in the exterior (see Fig. 2b), determine the tangents $\overline{xp_{i+1}}$ and $\overline{yp_{i+1}}$ to the 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 [20]. 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

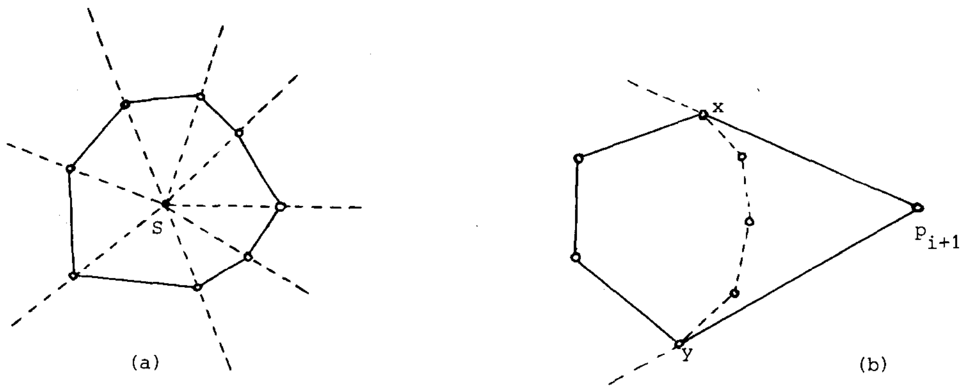


FIGURE 2

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 Fig. 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 (Fig. 3) that the number 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 $O(\log^2 n)$ for each insertion and deletion.

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 method, keeping points ordered around a fixed interior point S, is no longer feasible, because repeated insertions and

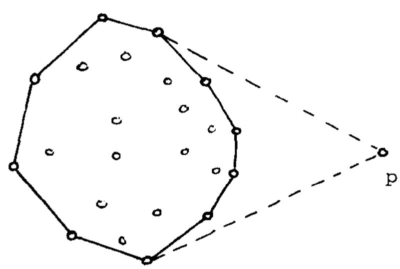


FIGURE 3

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, consisting of its separate left and right faces. Thus, the convex hull is represented by 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 Figs. 4a and b, 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. The 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. We will occasionally refer to these representations as "Q-structures."

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 $y_{p_i} \leqslant y_p \leqslant y_{p_j}$. If no two such points exists, then p lies above or below the *lc*-hull. Otherwise it only takes one comparison with the boundary edge $\overline{p_i p_j}$ of the *lc*-hull to determine where p is located w.r.t. the *lc*-hull. ∎

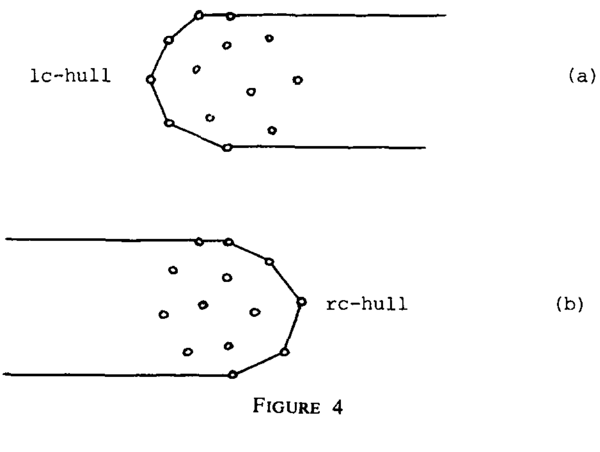


FIGURE 4

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 Fig. 5. 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. 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 [20].

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

Proof. Let $P = \{p_1,..., p_n\}$. Think of $\{p_1,..., p_i\}$ as A and of $\{p_{i+1},..., p_n\}$ as C. 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. Since p_1 to p_n are sorted by y-coordinate, the sets A and C indeed are separated by a horizontal line.

To find the *lc*-hull of P, all we have to do is to determined 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 [20]) requires that one can perform binary search on the *lc*-hulls. To this end we augment each node of a Q-structure with pointers to its leaves with highest and smallest y-coordinate, respectively. It is easily

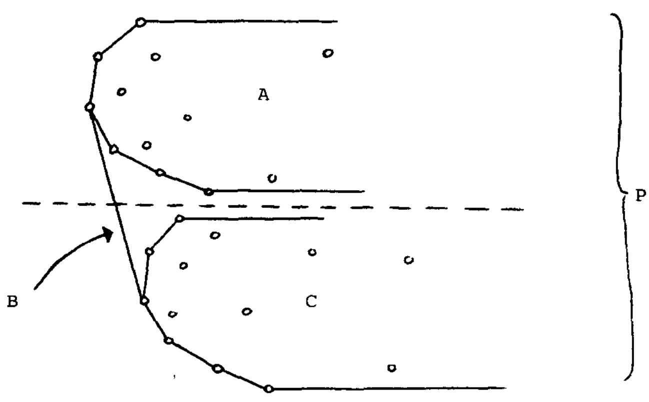
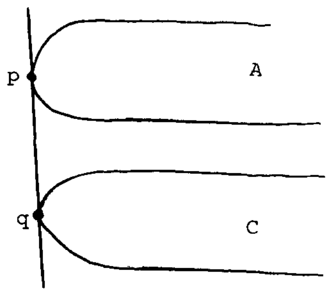


FIGURE 5

verified that the usual concatenable queue structures (AVL-treees, 2-3 trees, $BB[\alpha]$-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 of the tree and, whenever a node is reached representing a search segment $[p, r]$ on an lc-hull, then we only need to inspect the two inner leaves q_1 and q_2 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.

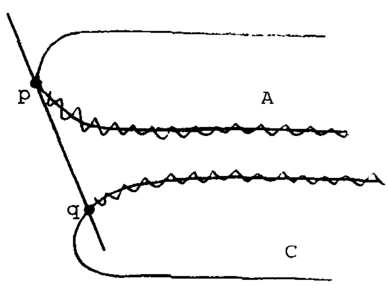
Given a point p on A and a point q on C we shall develop a criterion that enables us to eliminate the parts before or after p and q. Once we have such a criterion, repeatedly choosing p and q in the middle of the remaining parts of A and C enables us to find u and d, hence B, in $O(\log n)$. Let us look at the way $\overline{pq}$ intersects A and C. The following cases can occur.

Case a.



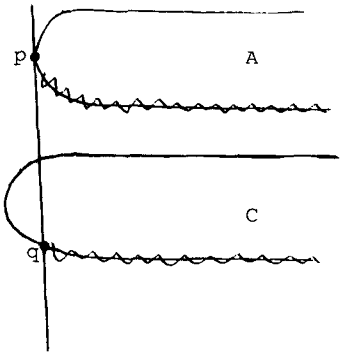
In this case $p = u$ and $q = d$ and we are done.

Case b.



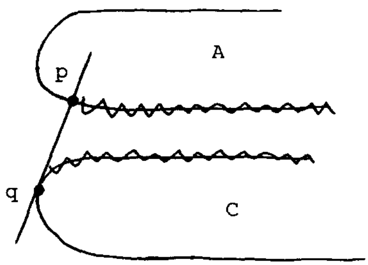
In this case the part of C before q can be deleted, and also the part of A after p.

Case c.



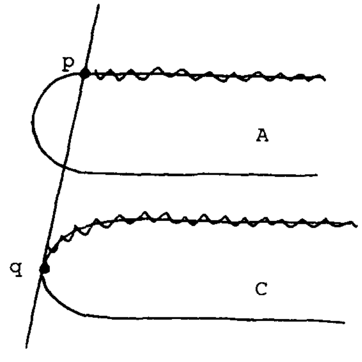
In this case the part of C following q can be deleted, and also the part of A following p.

Case d.



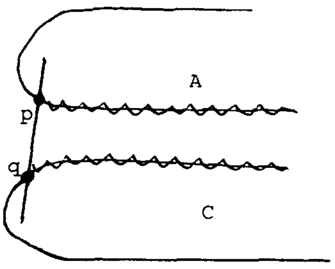
Similar to Case b.

Case e.



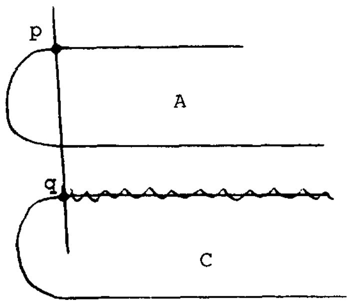
Similar to Case c.

Case f.



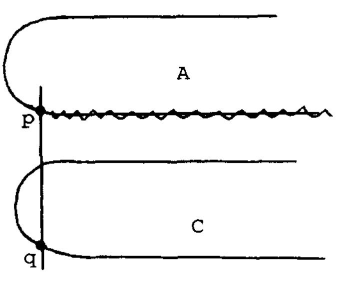
Again a part of both A and C can be deleted.

Case g.



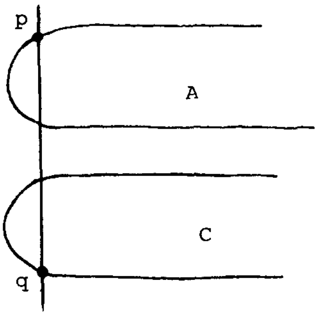
In this case only a part of C can be deleted.

Case h.



Similar to Case g.

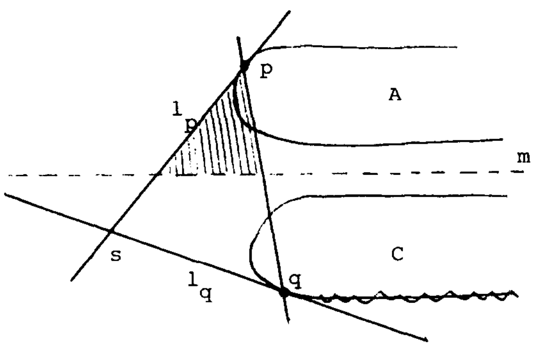
Case i.



This time we cannot say immediately what part of A or C can be deleted. This follows from the fact that when, e.g., u lies in the bottom part of A, d may lie on either side of q.

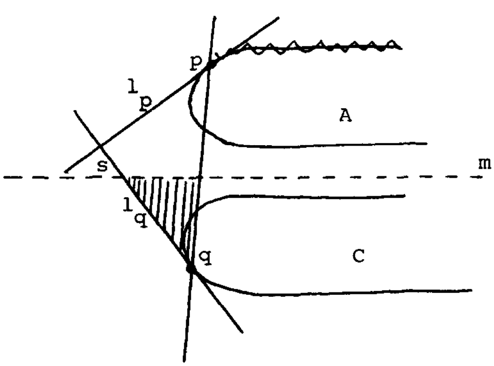
Let us consider Case i more carefully. Let m be the dividing line of the pointset. Let l_p be a tangent through p, let l_q be a tangent through q and let l be the line through p and q. l_p and l_q intersect at some point s left of l. The following two cases can occur.

Case i1. s lies below (or on) m.



Clearly, u can only lie in the shaded area or above p. It follows that d does not lie past q. Hence we can delete the part of C following q.

Case i2. s lies above m.



In this case the argument is completely similar and we can delete the part of A before p.

It follows that in all Cases a–i we either reach a decision or can delete half of A and/or C. Because all cases can be distinguished in constant time, the total time needed to find B is bounded by $O(\log n)$. ∎

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

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

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 and Hong [21], although the latter still requires $O(n \log n)$ steps after the initial sorting to complete.

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 them be stored by this attribute in a binary search tree T. We usually assume that no two points have the same y-coordinate, although this 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 Fig. 6) in only $O(\log n)$ steps, but there is a slight

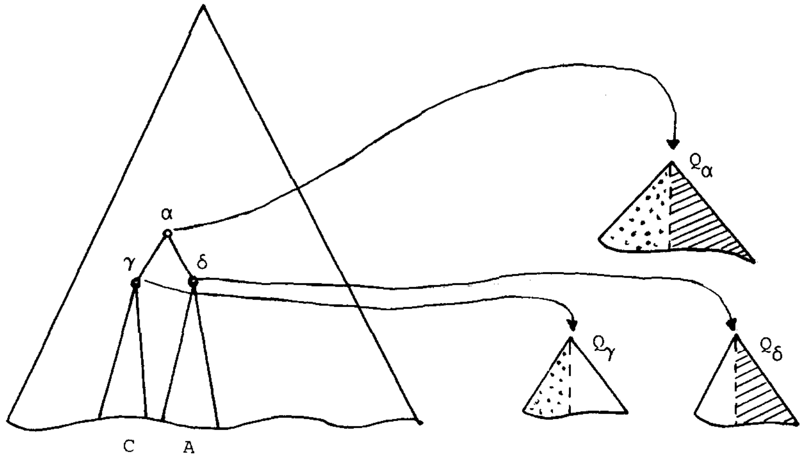


FIGURE 6

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 $O(\log n)$ time just to copy the segments of Q_γ and Q_δ which need to be joined to form Q_α. Fortunately 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)concatenating Q-structures per node visited. Going up also requires $O(\log n)$ steps per node, although it involves a few more complicated routines. We shall see how this intriguing structure functions below.

As it stands we have obtained an 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 ith level from below of T is still bounded by $O(\log 2^i)$, 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) $\text{lson}(\alpha) =$ a pointer to the left son of α,

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

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

(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) that 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 kth up to lth 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 the important routine DOWN to reconstruct the full Q_β at an arbitrary node β.

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, reconstructed) 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 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 by node α.}
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 $O(\log n)$, as it only involves some standard operations for concatenable queues of size $O(n)$ at a 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. These full structures are kept for later use; the Q^*-fields of nodes on the search path itself (except β) have temporarily become vacuous.

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 reconstructed it 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. $Q^*(\text{lson}(\alpha))$ and $Q^*(\text{rson}(\alpha))$ contain the complete *lc*-hulls of the sets below lson(α) and rson(α), respectively.}

begin

determine the bridge connecting $Q^*(\text{lson}(\alpha))$ and $Q^*(\text{rson}(\alpha))$ and thus the numbers of points B_1 and B_2 which they must each contribute into $Q^*(\alpha)$;

{record these numbers}

$B(\text{lson}(\alpha)) := B_1$;

$B(\text{rson}(\alpha)) := B_2$;

{Cut the necessary pieces off from the queues ...}

$Q_1 := Q^*(\text{lson}(\alpha))[1 \,..\, B_1]$;

$Q_2 := Q^*(\text{rson}(\alpha))[*{-}B_2 \,..\, *]$;

{effectively leaving the remaining parts at the sons}

{... and put them together to form the *lc*-hull of the joint set}

$Q^*(\alpha) := Q_1 \cup Q_2$;

if out of balance *then* BALANCE(α);

if α = root *then* goal reached *else* UP($f(\alpha)$)

end of UP;

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

LEMMA 4.3. UP *always reaches its goal after* $O(\log^2 n + R)$ *steps, where* R *is the cost of the rebalancings required along the search path.*

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 $O(\log n)$ 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 or 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 $BB[\alpha]$-trees (see, e.g., [1, 22]), 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. 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. 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 $Q_{\mathrm{lson}(\alpha)}$ and $Q_{\mathrm{rson}(\alpha)}$ 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* $O(\log n)$ *steps.*

Proof. Let the sons of $\mathrm{lson}(\alpha)$ be β and γ. Given $Q_{\mathrm{lson}(\alpha)}$, 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) $\mathrm{rson}(\alpha)$, 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 $O(\log n)$ 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 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* $O(\log^2 n)$ *per insertion and deletion.*

Proof. Using T^* as the underlying data structure, 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 $O(\log^2 n)$ in basic costs and an additional $O(\log n)$ 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 $O(\log^2 n)$. ▮

5. Applications of the Dynamic Convex Hull Algorithm

There are numerous problems which can be solved by using convex hull determination as a tool (cf. Shamos [25]). 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-dimensional 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 two dimensions a similar idea has given rise to the concept of "peeling" a convex hull (Tukey [28]), 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 two and more dimensions, but the computational complexity of it definitely is no longer linear.

Shamos [26] 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 [26] 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 4.5 to show

THEOREM 5.1. *One can peel a set of n points in the plane in only* $O(n \log^2 n)$ *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 $O(n \log n)$ steps. Next one can do any n deletions one likes, at at cost of $O(\log^2 n)$ steps per deletion. Hence the peeling of the set can be completed within $O(n \log^2 n)$ steps. (It is noted that this does not take the 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 1st layer, the ith layer is defined as the convex hull of the set of points remaining after peeling all previous layers off. The statistical significance was recognized by Barnett [2], who defined the c-order of a point at 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 [26] 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 $O(n \log^2 n)$ steps.*

Proof. Assume that the ith convex layer has c_i points, with $\sum_{i \geqslant 1} c_i = n$. 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 ith convex layer, for some $i \geqslant 1$. 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 $O(c_i \log^2 n)$ steps. The total time needed to "peel" off all convex layers will thus be in the order or

$$n \log n + \sum_{i \geqslant 1} c_i + \sum_{i \geqslant 1} c_i \log^2 n$$

which is $O(n \log^2 n)$. ∎

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.

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. 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 [26]), the following result is immediate.

THEOREM 5.3. *Given n points in the plane, one can determine a connecting spiral in only $O(n \log^2 n)$ 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 apply Theorem 4.5 to answer a basic question posed in Saxe and Bentley [23]. 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 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 4.5. The concatenable queue available at the root of the data structure is a full-fledged representation of the convex hull at all time, 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 [24]). 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. Recently, Chazelle and Dobkin [8] have given an algorithm for deciding whether two convex k-gons are disjoint in $O(\log k)$ steps. Using their result and Theorem 4.5 we obtain

THEOREM 5.5. *One can maintain two sets A and B of points in the plane such that insertions and deletions take $O(\log^2 n)$ time each (where n is the current size of the set on which they operate) and, whenever needed, separability can be decided in only $O(\log n)$ time.*

Note in Theorem 5.5 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 $O(1)$.

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 halfspaces is a convex polygon with at most n edges, where the polygon could very well be empty or have an "open" side. If we interpret a halfspace as the set of points satisfying some inequality $ax + by \leqslant c$, 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 [27] 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" or the "right" part of the space. In this way 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 intersection 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 he "right" halfspaces of the set.

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 Fig. 7). Considering the boundary, it is important 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

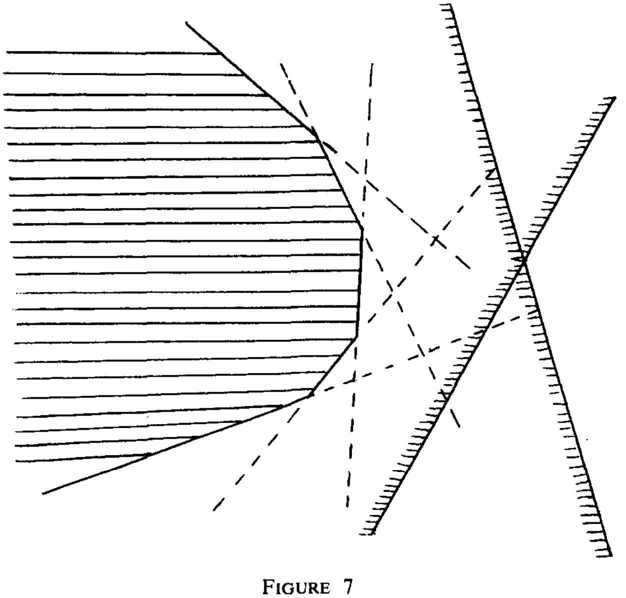


FIGURE 7

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 Fig. 8), which supports the repertoire of a concatenable queue and which keeps its leaves chained in a doubly linked list as well. If it is required to determine an edge of the boundary of the *l*-intersection (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 taken 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 (Fig. 8). 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 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. Note 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 is 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

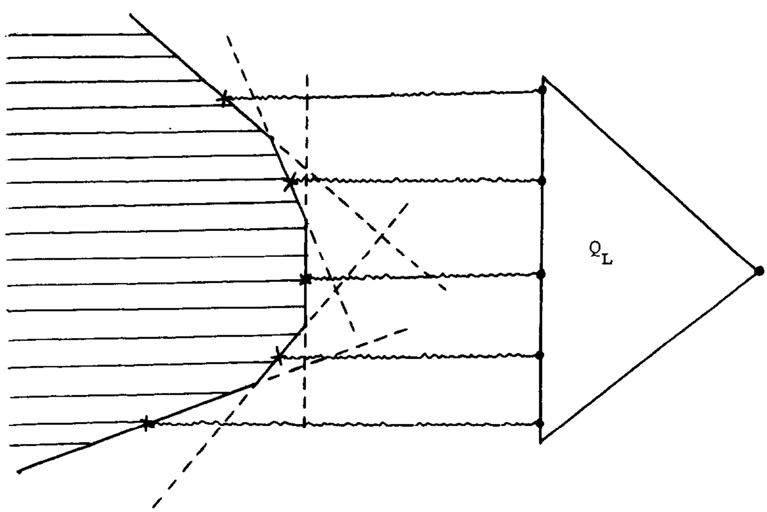


FIGURE 8

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.

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 Fig. 9 for some possible cases.

THEOREM 6.2. *The point(s) where the l- and r-boundaries intersect can be found in $O(\log n)$ 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 horizontal line m such that the point of intersection of R and m lies left of the point of intersection of L and m (see Fig. 10a). If no such line m exists then the intersection of L and R is empty. Otherwise, we know that there is at most one point of intersection of L and R above m and one point below m. The second phase will locate these points of intersection, if they exist.

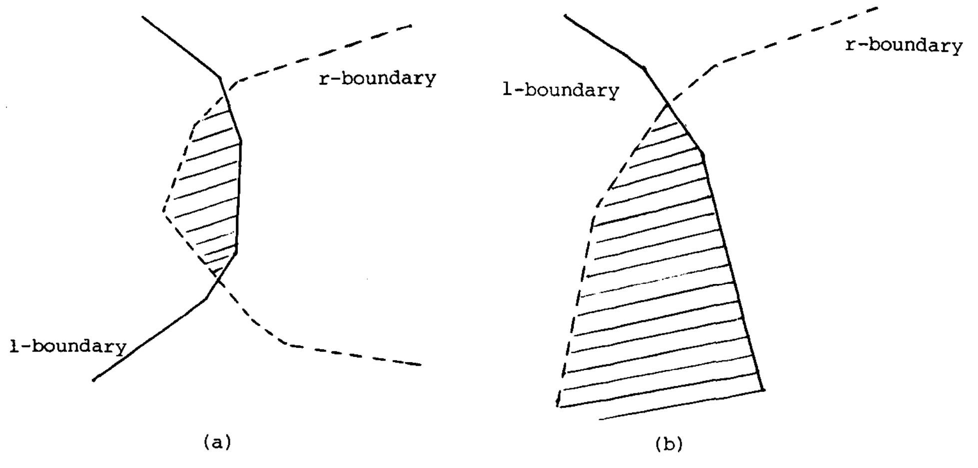


FIGURE 9

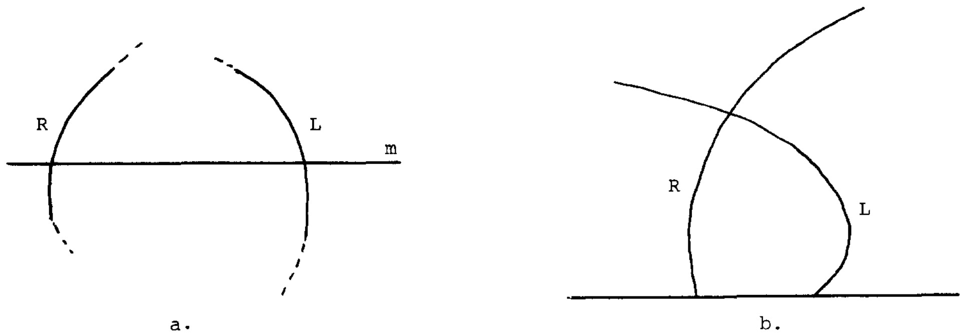


FIGURE 10

The first phase is most easily described using a result of Chazelle and Dobkin [8]. They show that, given two convex polygons, one can determine if they intersect and, when they do, find a witness of the intersection (i.e., a point lying in both polygons) in $O(\log n)$ steps. Because L and R are convex polygons, we can use this result to find such a witness (or know that L and R do not intersect). The horizontal line m through this witness will have the desired property. Its intersection with L and R can easily be computed in another $O(\log n)$ steps.

We will proceed by only considering the parts of L and R above m, as the intersection of the parts below m can be treated likewise. We split Q_L and Q_R at the points of intersection with m (see Fig. 10b). We know that L and R have at most one point of intersection above m. As in the proof of Theorem 3.2 (for finding a bridge) we would like to have a criterion to tell, given a point p on L and a point q on R, to what side of p and/or q on the respective arcs the (possible) point of intersection lies, in order that we can discard the parts on the other side from further consideration. Once we have such a criterion, we can find the point of intersection by iteration, every time choosing new p and q in the middle of the convex arcs left. In this way, either L or R is halved at every step and the process must converge in $O(\log n)$ steps.

Let s be the intersection of L and m, m' a horizontal line through p and l the line through s and p. l and m' divide the halfplane above m into four regions I to IV (see Fig. 11). L runs only through regions II and IV and R starts somewhere (on m) in region I. Given a point q on R, the following cases can occur.

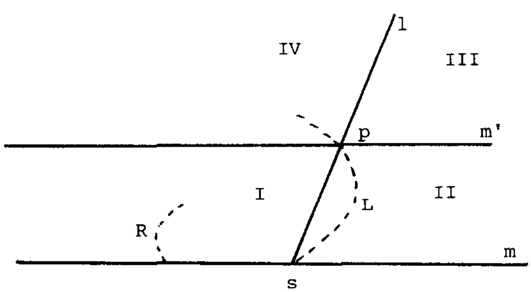
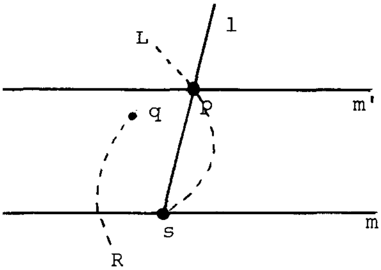


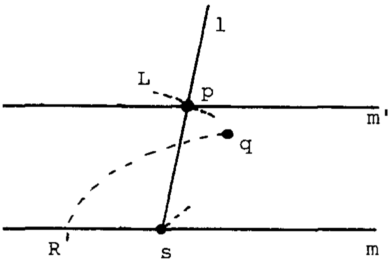
FIGURE 11

Case a. q lies in region I.



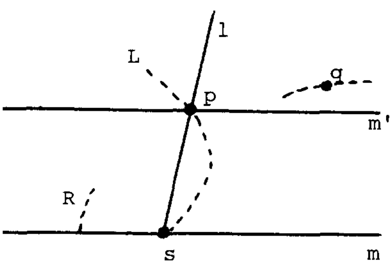
Because the point of intersection of R and m lies left of s the point of intersection of L and R lies above q. Hence the part of R below q can be deleted.

Case b. q lies in region II.



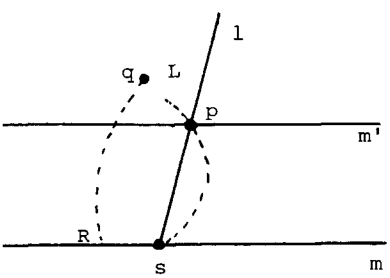
R must have crossed l below p and hence the point of intersection of L and R lies below p. Hence the part of L above p can be deleted.

Case c. q lies in region III.



Because L does not run through region III the point of intersection must lie below q and the part of R above q can be deleted.

Case d. q lies in region IV.



In this case R can never run through region II and the intersection must lie above p. Hence the part of L below p can be deleted.

It follows that when p and q were chosen as midpoints of their arcs, in each of the four cases we can delete half of L or half of R. Note that in each of the four cases, after the right part of an arc has been eliminated, an equal situation is created (perhaps using m' instead of m) on which the process can be iterated. When L and R have been reduced to line segments we can find the point of intersection by a simple direct test.

In a similar way we can find the point of intersection below m in $O(\log n)$ steps. After we have found the two points of intersection we have to undo all operations to restore Q_L and Q_R. ∎

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 can be done efficiently.

PROPOSITION 6.3. *Given representations of the l- and r-boundaries as concatenable queues, one can compute the boundary of the common intersection of a set of n halfspaces on $O(\log n)$ steps.*

Proof. It takes $O(\log n)$ 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 and join them in a single representation of the resulting convex (open) n-gon. ∎

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 [20]).

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 it takes only $O(\log n)$ steps to fully update the current l-boundary after each insertion.*

Proof. Maintaining the current l-boundary L as a concatenable queue Q, let us see what happens when another halfspace h is inserted in the set. We assume, as we may, that h is indeed a left halfspace. Viewing h as r-boundary R, we can compute the two points of intersection (if any) with L in $O(\log n)$ steps using Theorem 6.2. After these points are found, we must split Q at these points and concatenate some parts to add h to Q, in a total of $O(\log n)$ steps also. ∎

Theorem 6.4 shows that, as for convex hulls, there is a "real-time" algorithm for building l-intersections. In a very similar way one can, in fact, obtain a real-time 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 data structure 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 (Fig. 12). Ideally we now want to augment T and associate with each internal node α of T a concatenable queue Q_α containing (the l-boundary of) the l-intersection of the halfspaces in its subtree! Before we do so, we need to establish one more basic fact for l-intersections (viz. l-boundaries).

The l-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 l-intersection of H can be determined with relatively little computational effort from the l-intersections of A and of C separately.

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

Proof. By the earlier remarks it is sufficient to consider the l-boundaries of the sets in question. Let the l-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 Figs. 13a–b (where h can be of any slope). The main observation should be that the l-boundaries of A and C intersect in precisely one point q. Clearly the l-boundary of H is obtained by taking C's boundary up to q and continuing on A's boundary from q onwards.

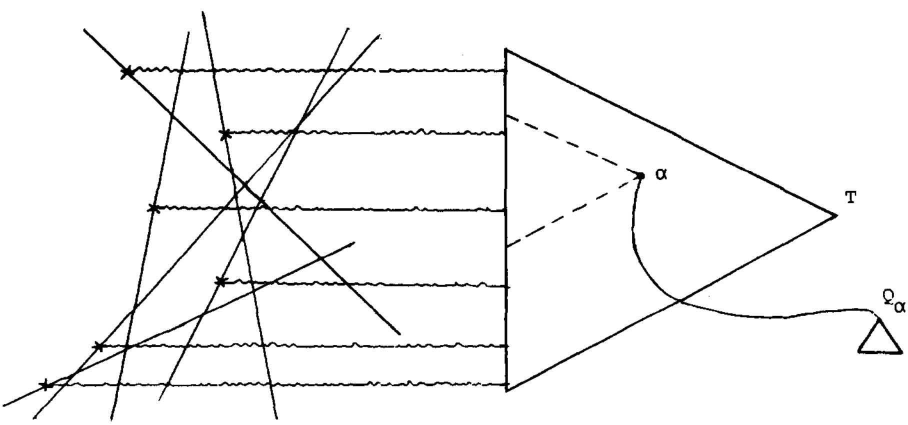


FIGURE 12

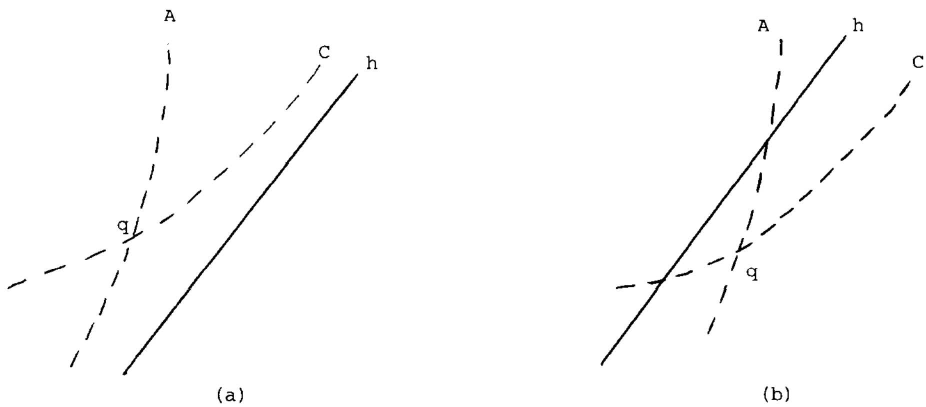


FIGURE 13

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 Figs. 13a–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. 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 $O(\log n)$ steps, we split the queues representing A and C's l-boundaries and glue them together in the right order, to obtain the l-boundary of H in only $O(\log n)$ additional steps. ∎

We shall exploit Theorem 6.5 in a dynamic algorithm for maintaining the l-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 Theorem 6.2) to consider the computation of the l-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 l-intersection of a next group of 2^i halfspaces from the l-intersection of each of the two constituent "halves" as computed in the previous round. Using Theorem 6.5 this procedure takes

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

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 by 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 (ref. Fig. 12). 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 to 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.

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 of 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 $O(\log^2 n)$ 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 fulls. The time analysis carries over also. Note that by Proposition 6.3 it would take no more than $O(\log n)$ 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. ∎

Halfspaces come up in a number of interesting problems in the plane and Theorem 7.1 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 Theorem 7.1 and Lemma 6.1 we conclude

THEOREM 7.2. *One can dynamically maintain the common intersection of a set of halfspaces in the plane such that insertions and deletions can be processed in* $O(\log^2 n)$ *steps, and queries of the form "does* x *belong to the current common intersection" can be answered in only* $O(\log n)$ *steps, 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 crossing 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 [27] 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 7.3. *One can dynamically maintain the kernel of a simple* n*-gon at a cost of only* $O(\log^2 n)$ *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 7.4. *One can dynamically maintain the feasible region of a linear program in two variables at a cost of only* $O(\log^2 n)$ *steps for each inequality added or 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 $x \leqslant y$ if and only if $x_1 \leqslant y_1$ and $x_2 \leqslant y_2$. A point x is called maximal in a set P when $x \in P$ and no $y \in P$ exists with $x < y$ (i.e., $x \leqslant y$ and $x \neq y$). It is customary to draw horizontal and vertical lines from each of the maximal elements of a set (see Fig. 14), until they cross. This 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 is not unlike an *rc*-hull as introduced in Section 3.

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 *et al.* [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 [29]). 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.

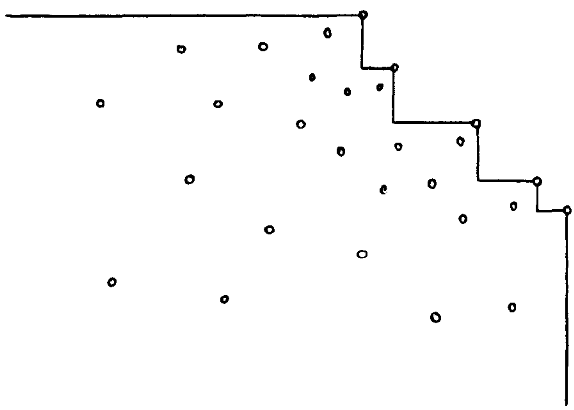


Figure 14

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 (Fig. 15) 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 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 $m_0, m_1, \ldots$ 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, merely by disregarding their x-coordinates (the elements appear sorted on either coordinate). Assuming that $c \leqslant y_{m_0}$ (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 $y_{m_i} < c < y_{m_{i+1}}$. The cases are illustrated in Fig. 16 *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

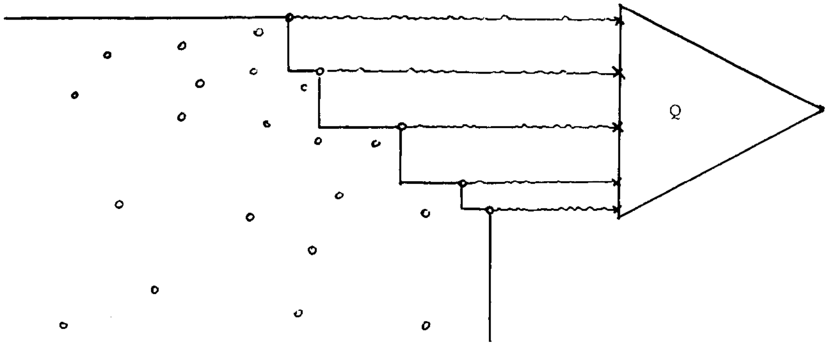


FIGURE 15

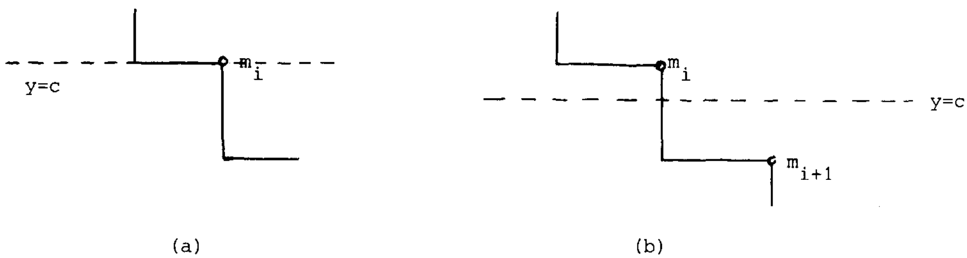


FIGURE 16

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 [20] 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 a queue 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. Otherwise any one of the cases shown in Figs. 17a–c can happen (m_k denotes the "last" element 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 (Fig. 18) 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 contour where the vertical line through p intersects (see Fig. 18). To update the m-

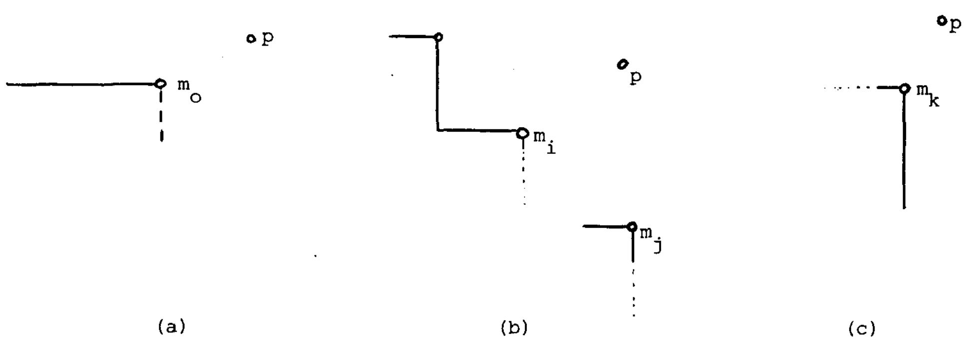


FIGURE 17

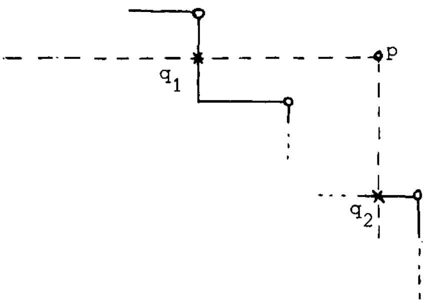


FIGURE 18

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.

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 [20]), 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, because it ignores the need to keep track of the "interior" of the hull of current maximal elements (compare Section 2).

To accomodate deletions as well we shall follow a very similar approach as for convex hulls and halfspaces. Let us store all points of 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. 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 a concatenable queue Q_α containing the maximal elements (in order) of the set of points covered by its subtree (see Fig. 19). While this has always been 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 (which they are) and split the set, by drawing an arbitrary horizontal line, in two disjoint subsets A and C (see Fig. 19). 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 $P = \{p_1, \dots, p_n\}$ be a set of points in the plane, ordered by y-coordinate. Given the m-contours of $A = \{p_1, \dots, p_i\}$ and of $C = \{p_{i+1}, \dots, p_n\}$ as*

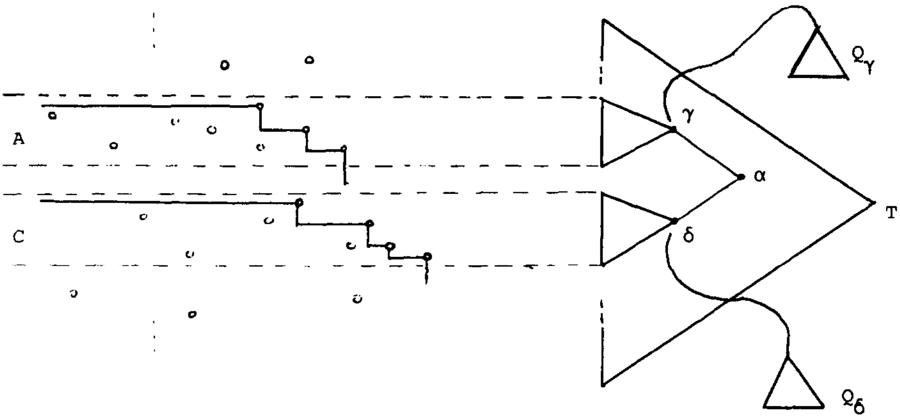


FIGURE 19

concatenable queues (any $1 \leqslant i < n$), *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 the point q(if it exists ...) where it intersects C's contour. The different cases that can arise are shown in Figs. 20a–b.

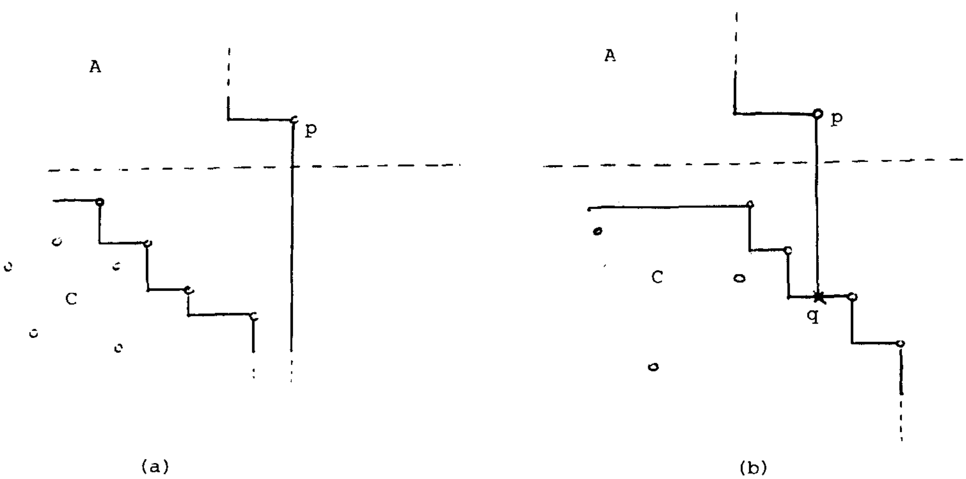


FIGURE 20

When no intersection exists (Fig. 20a), 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 (Fig. 20b), 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 similarity of Theorem 8.3 with Theorem 3.2 (for *lc*-hulls) and Theorem 6.5 (for *l*-intersections of halfspaces).

9. Dynamically Maintaining the Maximal Elements of a Plane Set (Algorithms 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. Fig. 19). 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 (and not just 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, this implies that at least at one of the sons an empty structure remains (compare the proof of Theorem 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 main-*

tained 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. We conclude that the necessary updates of the structures 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 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.

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. ∎

It so happens that such queries are decomposable in the sense of Bentley [3]. A direct solution to the problem that achieves also an update time of $O(\log n)$, was recently obtained by Lueker [18].

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. 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) as a doubly-linked sub-list of the current contour. To manage it, one must keep track of these sub-lists in all queues Q_α associated with nodes α in T, i.e., in the pieces of these queues that are kept at the internal nodes. The internal nodes of these queues themselves must also keep a flag indicating whether there 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 $A \cup B$, as it is available at the root of T^*, is empty. 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. In all these problems we have followed a very similar line of reasoning and have obtained dynamizations based on one technique, which 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 problems that lend themselves to a divide-and-conquer approach. 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.

11. Final Remarks

After a preliminary version of this paper appeared in the STOC 80 proceedings, the authors obtained various improvements in the efficiency of the methods used. Several people, including H. Edelsbrunner (Graz), I. Gowda (Vancouver), J. B. Saxe (Pittsburgh) and G. F. Swart (Seattle), have suggested or independently proved the current Theorems 3.2 and 6.2 which originally stated $O(\log^2 n)$ bounds. Recently, M. H. Overmars clarified the connection between the techniques used here and the existence of divide-and-conquer algorithms for problems in general.

12. References

1. A. V. Aho, J. Hopcroft, and J. D. Ullman, "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, Mass., 1974.
2. V. Barnett, The ordering of multivariate data (with discussion), *J. Roy. Statist. Soc. Ser. A* **139** (1976), 318–354.
3. J. L. Bentley, Decomposable searching problems, *Inform. Process. Lett.* **8** (1979), 244–251.
4. J. L. Bentley and M. I. Shamos, Divide and conquer for linear expected time, *Inform. Process. Lett.* **7** (1978), 87–91.
5. J. L. Bentley 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. M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, Time bounds for selection, *J. Comput. System Sci.* **7** (1972), 448–461.
7. K. Q. Brown, "Fast Intersection of Halfspaces," Techn. Rep. CMU-CS-78-129, Carnegie-Mellon University, 1978.
8. B. Chazelle and D. P. Dobkin, Detection is easier than computation, *in* "Proceedings, 12th Annual ACM Symp. Theory of Computing, Los Angeles, 1980," pp. 146–153.
9. W. F. Eddy, A new convex hull algorithm for planar sets, *ACM Trans. Math. Software* **3** (1977), 398–403, 411–412.
10. P. J. Graham, An efficient algorithm for determining the convex hull of a finite planar set, *Inform. Process. Lett.* **1** (1972), 132–133.
11. P. J. Green, Convex hulls in the analysis of bivariate data, University of Durham, Sheffield, UK, paper read at the Conference "Looking at Multivariate Data," 1980.
12. P. J. Green and B. W. Silverman, Constructing the convex hull of a set of points in the plane, *Comput. J.* **22** (1979), 262–266.
13. H. Hadwiger and H. Debrunner, "Combinatorial Geometry in the Plane," Holt, Rinehart & Winston, New York, 1963.
14. P. J. Huber, Robust statistics: a review, *Ann. Math. Statist.* **43** (1972), 1041–1067.
15. R. A. Jarvis, On the identification of the convex hull of a finite set of points in the plane, *Inform. Process. Lett.* **2** (1973), 18–21.
16. H. T. Kung, F. Luccio, and F. P. Preparata, On finding the maxima of a set of vectors, *J. Assoc. Comput. Mach.* **22** (1975), 469–476.

17. D. T. LEE AND F. P. PREPARATA, An optimal algorithm for finding the kernel of a polygon, *J. Assoc. Comput. Mach.* **26** (1979), 415–421.
18. G. S. LUEKER, "A Transformation for Adding Range Restriction Capability to Dynamic Data Structures for Decomposable Searching Problems," Techn. Rep. #129, ICS Department, UC Irvine, 1979.
19. M. H. OVERMARS AND J. VAN LEEUWEN, Dynamically maintaining configurations in the plane, *in* "Proceedings, 12th Annual ACM Symp. Theory of Computing, Los Angeles, 1980," pp. 135–145.
20. F. P. PREPARATA, An optimal real-time algorithm for planar convex hulls, *Comm. ACM* **22** (1979), 402–405.
21. F. P. PREPARATA AND S. J. HONG, Convex hulls of finite sets of points in two and three dimensions, *Commun. ACM* **20** (1977), 87–93.
22. E. M. REINGOLD, J. NIEVERGELT, AND N. DEO, "Combinatorial Algorithms: Theory and Practice," Prentice–Hall, Englewood Cliffs, N. J., 1977.
23. J. B. SAXE AND J. L. BENTLEY, Transforming static data structures to dynamic structures, *in* "Proceedings, Conf. Rec. 20th Annual IEEE Symp. on Foundations of Computer Science, San Juan, Puerto Rico, 1979," pp. 148–168.
24. M. I. SHAMOS, Geometric complexity, *in* "Proceedings, 7th Annual ACM Symp. on Theory of Computing, Albuquerque, May 1975," pp. 224–233.
25. M. I. SHAMOS, Geometry and Statistics: problems at the interface, *in* "Recent Results and New Directions in Algorithms and Complexity (J. F. Traub, Ed.)," pp. 251–280, Academic Press, New York, 1976.
26. M. I. SHAMOS, "Computational Geometry," Ph. D. thesis, Yale University, 1978.
27. M. I. SHAMOS AND D. HOEY, Geometric intersection problems, *in* "Proceedings, Conf. Rec. 17th Annual IEEE Symp. on Foundations of Computer Science, Houston, Oct. 1976," pp. 208–215.
28. J. W. TUKEY, referred to in [25, p. 267].
29. P. VAN EMDE BOAS, On the $\Omega(n \log n)$ lowerbound for convex hull and maximal vector determination, *Inform. Process. Lett.* **10** (1980), 132–136.
30. N. WIRTH, "Algorithms + Data Structures = Programs," Prentice–Hall, Englewood Cliffs, N. J., 1976.
31. A. C-C. YAO, "A Lower Bound to Finding Convex Hulls," STAN-CS-79-33, Computer Science Dept., Stanford University, 1979.