DYNAMICALLY MAINTAINING CONFIGURATIONS IN THE PLANE
(detailed abstract)

Mark H. Overmars and Jan van Leeuwen

Department of Computer Science, University of Utrecht
P.O.Box 80.012, 3508 TA Utrecht, the Netherlands

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 of known static structures for these configurations. 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 that one can "peel" a set of n points in only $O(n\log^3 n)$ steps and that one can maintain two sets at a costs of only $O(\log^3 n)$ or less per insertion and deletion such that it never takes more than $O(\log^2 n)$ steps to determine whether the two sets are separable by a straight line. Also 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 plane points. The results are all derived by means of one master technique, which is applied repeatedly and which seems to capture an appropriate notion of "decomposability" for configurations.

1. Introduction.

Computational geometry (cf. Shamos [20,22]) 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 accomodating both insertions and deletions of objects. 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 to convert static datastructures 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 amenable to dynamization. Bentley [3] and Saxe and Bentley [19] presented several powerful techniques, which can be called into action on any decomposable searching problem to obtain reasonable update times, without the search or query times becoming very large. The techniques primarily support insertions, but later studies have addressed the problem of supporting deletions fast too [15,25,26].

While the theory as it stands is applicable to a wide variety of point problems, Saxe and Bentley [19, appendix] observed already that their techniques were insufficient to dynamize entire configurations (such as convex hulls) as well. Yet many of the geometric configurations commonly considered intuitively have a "decomposable flavor."

We shall prove for a number of different types of geometric configurations that efficient dynamizations can be achieved and identify the sort of decomposability which all these configurations seem to share (without attempting, at this time, to develop a general theory of it). We shall present efficient algorithms to dynamically maintain the convex hull of a set of points, the common intersection of a collection of halfspaces, the contour of maximal elements of a set of points and several other configurations in the plane. 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 was never even discussed before. Many applications will be mentioned, of which some are of immediate interest to such areas as computational statistics (cf. Shamos [21]) 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 master technique, which can be taken as additional evidence that the configurations we consider share 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 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 a possibility.

2. Dynamically maintaining a convex hull (prelude).

In the past few years 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, e.g. [7,8,11,17]. 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. Nearly all convex hull algorithms known today require that all inputs are read and stored before any processing can begin. Shamos [22] 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 to convex hull construction [20,24,28] updates due to the addition of a single point will cost at least $\Omega(\log n)$ on the average. Preparata [16] recently showed an algorithm to insert a point and update the convex hull within $O(\log n)$ as a worst case bound.

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 Preparata's [16]. They virtually all go by the principle that points found to be in the interior of the convex hull can be thrown away, and some are especially designed to eliminate as many points from further consideration as they can to cut down on the ultimate expected 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" and some old points of the interior suddenly become part of the new convex hull (figure 1). We will show that 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.

3. Dynamically maintaining a convex hull (representation).

Given the task to maintain it dynamically, an immediate problem is how to 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 (figure 2), consisting of its separate left and right faces. 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 figures 3 and 4, respectively. We will concentrate on the lc- hull of a set, as its rc- hull is treated in 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 which 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 by ordered y- coordinates in a concatenable queue Q_L (figure 5). The contour of the rc- hull is stored likewise in a concatenable queue Q_R. We clearly 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 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 immediately derived. 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 $y_{p_i} \le y_p \le y_{p_j}$ (if $y_{p_i} = y_{p_j}$ then also $x_{p_i} \le x_p < x_{p_j}$). If no two such points exist, then p lies above or below the lc- hull. Otherwise we only need to test whether p lies to the left or to the right of $\overline{p_i p_j}$ to determine its location 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 by a horizontal line into two parts A and C, as in figure 6. 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.

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 $1 \le i < n$), then the lc- hull of the entire set can be built in $O(\log^2 n)$ steps.

The proof is based on finding B as the common tangent of A and C's lc- hull, splitting Q_A and Q_C and glueing the pieces above and below the bridge together (figure 6). 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. 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 from the lc- hulls of their likewise horizontally separated halves of 2^{i-1} points (which were constructed at the previous iteration). The number of steps needed to build the hulls amounts to about

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

The composition of the lc- and rc- hull to obtain the complete convex hull is trivial.

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 [17], although the latter still requires O(n log n) steps after the initial sorting to complete.

We have no indication that theorem 3.2. is best possible and it is conceivable that the $O(\log^2 n)$ bound can be improved.

Definition. Let J(n) be the time to find the one common tangent of two horizontally separated lc- hulls of n points total (represented as concatenable queues).

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

4. Dynamically maintainining 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. As we must accommodate both insertions and deletions, some information must be maintained about the arrangement of the points currently in the interior of the lc- hull. It seems reasonable to maintain information about the inner layers of the current set, i.e., about the lc- hulls of horizontally separated subsets of the points present.

Let the points of the set be sorted by y- coordinate and let they be stored 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 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_σ associated with the sons γ and σ of α. If we want to build Q_α from Q_γ and Q_σ and retain Q_γ and Q_σ as they are, then we would have to spend much more than J(n) time just to copy the segments of Q_γ and Q_σ which need to be joined to form Q_α. Note that Q_α is obtained by concatenating the proper head segment of Q_γ and tail segment of Q_σ, with the bridge in between. It is clear that we might as well cut the required segments off from Q_γ and Q_σ and pass them on to α to assemble Q_α by concatenation, leaving γ and σ with only a fragment of their original associated structure (figure 7). _If we remember at node α where the bridge was when we built Q_α_, then we only have to split it at this very spot to obtain the two "halves" again which must be concatenated to the left- over pieces at γ and σ to fully reconstruct Q_γ and Q_σ.

This leads to 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. We will show that T^* can be maintained efficiently. 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) that 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, 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 T^*, we shall first devise a routine (DOWN) to reconstruct the full Q_β at an arbitrary node β. There will be some additional side benefits 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 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 will be guided by the usual decision criterion in binary search trees.

```
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...}
          Q_1 := Q*(α) [1..B(lson(α))];
          Q_2 := Q*(α) [B(lson(α))+1..* ];
          {... and glue the pieces back into the
           queues left at the two sons}
        $Q^*(\mathrm{lson}(\alpha))$ := $Q^*(\mathrm{lson}(\alpha)) \cup Q_1$ ;
        $Q^*(\mathrm{rson}(\alpha))$ := $Q_2 \cup Q^*(\mathrm{rson}(\alpha))$;
        {Continue the search in the right direction}
        if $\beta$ below lson($\alpha$)
          then DOWN (lson($\alpha$), $\beta$)
          else DOWN (rson($\alpha$), $\beta$)
      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). One easily shows

Lemma 4.1. DOWN always reaches its goal after $O(\log^2 n)$ steps.

In addition to Q_β , the call of DOWN (root,β) produces the full Q- structure (thus the complete lc- hull of all points below it) at each node α whose father is on the search path but which isn't on it itself. 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. 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. 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($\alpha$);
{$\alpha$ is the node most recently reached on the way back to the root. $Q^*(\mathrm{lson}(\alpha))$ and $Q^*(\mathrm{rson}(\alpha))$ contain the complete lc- hulls of the sets below lson($\alpha$) and rson($\alpha$), respectively.}
  begin
    determine the bridge connecting $Q^*(\mathrm{lson}(\alpha))$ and
    $Q^*(\mathrm{rson}(\alpha))$ and thus the numbers of points $B_1$
    and $B_2$ which they must each contribute into
    $Q^*(\alpha)$;
    {record these numbers}
    B(lson($\alpha$)) := $B_1$;
    B(rson($\alpha$)) := $B_2$;
    {Cut the necessary pieces off from the queues...}
    $Q_1$:= $Q^*(\mathrm{lson}(\alpha))[1..B_1]$;
    $Q_2$:= $Q^*(\mathrm{rson}(\alpha))[*-B_2..*]$;
    {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 ($\alpha$);
    if $\alpha$= root then goal reached else UP(f($\alpha$))
  end of UP;
```

Note what pieces from $Q^*(\mathrm{lson}(\alpha))$ and $Q^*(\mathrm{rson}(\alpha))$ 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. One can show

Lemma 4.2. UP always reaches its goal after $O(\log n.J(n)+R)$ steps, where R is the cost of all rebalancings required along the search path during the particular action.

To get an impression of R, we shall delve into the necessary actions for rebalancing a single node α. We shall restrict ourselves to familiar types of balanced trees like AVL- trees and BB[α]- trees (e.g.[1,18]), which can be rebalanced by means of local rotations. Let us examine the case in which a single rotation must be carried out at node α (see figure 8). The case in which a double rotation must be carried out is very similar and will not be discussed in detail. When BALANCE is called in the procedure UP, we have just completed building Q_α. We shall 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.

Lemma 4.3. A rotation can be carried out in $O(\log n + J(n))$ steps.

Proof. Referring to figure 8, let the sons of lson(α) 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 σ be the new "right son" of α as a result of the rotation.

Observing that the complete Q- structures are present at β,γ and (the old) rson(α), we can restore the proper information at the nodes involved and climb back to α(where we were) by starting UP again at node σ. Thus a rotation give rise to at most O(log n + J(n)) extra steps. The analysis for double rotations proceeds in the same way and yields the same estimate. □

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

Theorem 4.4.a.The convex hull of a set of n points in the plane can be maintained at a cost of $O(\log^2 n + \log n . J(n))$ per insertion and deletion.

Proof.We would proceed as follows to insert or delete a point p. We shall describe the necessary actions only for the lc- hull. First we search down T^* using p's y- coordinate fo find out in which leaf p is (or must be) stored. We do so by using the procedure DOWN, which will restore the complete lc- hulls at all nodes immediately bordering the search path at a total cost of only $O(\log^2 n)$.The next step will depend on whether p must be inserted or deleted. If we think of T as an AVL- tree or a BB[α]- tree, then it amounts to the creation or elimination of a single leaf and takes just O(1) time. Next we must process the change at the bottom of the tree using the standard routines for the type of balanced tree chosen and we must reconfigure the associated information at all nodes on the search path. We do so by means of the procedure UP. By lemma 4.2, UP takes O(log n. J(n)) in basic costs and, using lemma 4.3, another O(log n + J(n)) for each rebalancing. As the number of rebalancings required will never be larger than O(log n), the total time UP takes will be bounded by $O(\log^2 n + \log n .\ J(n))$.□

Using that $J(n)= O(\log^2 n)$, theorem 4.4.a leads to the conclusion 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 spent on UP when processing an insertion. One can show that in this case the new bridge required at a node can be computed in only O(log n) steps, using knowledge about were the old bridge was.

Theorem 4.4.b. The convex hull of a set of n points in the plane can be maintained at a cost of $O(\log^2 n + \log n .\ J(n))$ per deletion and a cost of $O(\log^2 n + r.\ J(n))$ per insertion, where r is the number of rebalancings required in performing the insertion.

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 a cost of $O(\log^3 n)$ per deletion.

Proof. Using that $J(n)= O(\log^2 n)$, the time bound for deletions follows. Let us choose to represent T as an AVL- tree. It is wellknown that in processing an insertion in an AVL- tree at most one rebalancing will be needed [18,27]. It follows from theorem 4.4.b that in this circumstance insertions will never take more than $O(\log^2 n)$ steps.□

When it can be shown that J(n)=O(log n), then theorem 4.5 can be improved to read that both insertions and deletions can be processed in $O(\log^2 n)$.

5. Applications of the dynamic convex hull algorithm.

There are numerous problems in computational geometry which can be solved by using convex hull determination as a tool (cf. Shamos [20]). The algorithm we devised for dynamically maintaining a convex hull will allow us to tackle a number of 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. In 2 dimensions it leads to the concept of "peeling" a convex hull, usually to remove a fixed percentage of outlying points of the set (Huber [10]). Each time a point is removed, the convex hull must be updated accordingly. Shamos [22] reported an $O(n^2)$ algorithm for peeling a set of n points in the plane, based on an iteration of Jarvis' convex hull algorithm ([11]). Green and Silverman [9] gave an algorithm based on Eddy's algorithm ([7]), which isn't better in worst case. Shames [22] gave it as an open problem to do better than $O(n^2)$. We can apply theorem 4.5 to show

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

A closely related problem concerns finding the convex layers of a plane set of points. The statistical significance 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. From theorem 4.5 we may derive

Theorem 5.2 One can determine the joint convex layers of a set of n points in the plane (and hence

Barnett's c- order groups) in only $O(n \log^3 n)$ steps.

Given the convex layers of a set, one may traverse points in clockwise order layer after layer, each time using a "forward" tangent to step over to the next inner layer. The resulting path (a "spiral") connects all points in the set, does not intersect itself and has the property that all corners in traversal order are convex. See figure 9.

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

Returning to convex hulls, we can now also answer a question posed in Saxe and Bentley [19]. It concerns the dynamization of the simplest type of convex hull searching ("is x within the convex hull of set F"), to which their methods did not apply.

Theorem 5.4 One can maintain a set F of n points in the plane at a cost of $O(\log^2 n)$ per insertion and $O(\log^3 n)$ per deletion, such that queries can still be answered in $O(\log n)$ steps.

A last and intriguing application concerns the concept of separability (Shamos [20]). Two sets in the plane are said to be separable if one can draw a line such that one set is entirely to its left, the other one entirely to its right. Two sets are separable if and only if their convex hulls are disjoint (Shamos [20]). Unfortunately the best previously known algorithms for deciding whether two convex n-gons are disjoint do not sufficiently take advantage of any preprocessing and run in $O(n)$ steps.

Theorem 5.5. One can determine whether two (preprocessed) convex n-gons in the plane are disjoint or not in only $O(\log^2 n)$ steps.

Proof (sketch). Let the convex n-gons be A and B. Take two points a_1 and a_2 on A such that the arcs $\widehat{a_1a_2}$ and $\widehat{a_2a_1}$ have $\frac{n}{2}$ points each. Draw the line $\overline{a_1a_2}$ and determine its intersection with B. If the n-gons have been properly preprocessed, the points a_1 and a_2 and the points of intersection b_1 and b_2 can be found in log n steps. If the intervals $[a_1a_2]$ and $[b_1b_2]$ on the line are not disjoint, then neither are A and B. If the intervals are disjoint (see figure 10), then we proceed as follows. Let b_2 and a_1 be adjacent. Draw a tangent l_A of A through a_1 and a tangent l_B of B through b_2. If l_A and l_B meet above the line, then A and B cannot intersect below it, we can throw away the lower half of A (splitting its preprocessed form in log n time) and repeat the procedure. Other cases are treated likewise. If $l_A // l_B$ (or A has been reduced to 1 or 2 points), then A and B are disjoint. At each step (costing a total of log n) we either reach a decision or can eliminate another half of A. Hence the $O(\log^2 n)$ bound. Afterwards the splitting of A must be undone again, by reverting the process.□

Fortunately the dynamic convex hull algorithm keeps convex hulls in a form suitable for theorem 5.5. Thus we have

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

6. Dynamically maintaining the common intersection of a set of halfspaces.

A halfspace is a part of the plane entirely to the left or to the right of a line. The common intersection of a set of n halfspaces is a convex polygon (possibly open or empty) with at most n edges. Shamos and Hoey [23] have shown that such a common intersection can be found in $O(n \log n)$, but their technique is off- line and works for static sets only. The dynamization is harder to obtain than for convex hulls, but we will show that a same approach will work.

Again we separately maintain two sets, the l- intersection of halfspaces which are "open to the left" (see figure 11) and the r-intersection of halfspaces which are turned the other way. Each such intersection is a "convex arc" with its two arms extending to infinity. Each arc will have its edges stored at the leaves of some concatenable queue again, ordered by angle. By a tedious argument one can show (thus improving on Brown [6])

Theorem 6.1. One can determine the common intersection of l-intersection and r-intersection (hence the common intersection of the set) in only $O(\log^2 n)$ steps.

Glueing the parts together is not as easy as it was for the left and right sides of a convex hull, but the upperbound isn't bad. Let us concentrate on maintaining the l-intersection. Note that halfspaces contribute to the l-intersection (if they do...) in the order of their direction! Thus we shall keep halfspaces sorted by angle and observe the following decomposition property (compare 3.2)

Theorem 6.2 Let $h_1,\dots,h_n$ be n arbitrary halfspaces in the plane, sorted by angle. Given the representation of the 1-intersections of $h_1,\dots,h_i$ and of $h_{i+1},\dots,h_n$ (any $1 \leq i < n$), the representation of the 1-intersection of $h_1,\dots,h_n$ can be constructed in only $O(\log^2 n)$ steps.

An augmented balanced tree can now be built in very much the same way as we did for convex hulls. By very similar maintenance procedures we can show

Theorem 6.3. One can dynamically maintain the common intersection of a set of n halfspaces in the plane (as a convex polygon) at a cost of only $O(\log^3 n)$ per insertion and deletion.

As a bonus we obtain an algorithm to construct the common intersection of a set which, after the initial sorting of all halfspaces by direction in $O(n \log n)$ steps, takes only $O(n)$ additional steps.

A first application concerns the simplest type of intersection searching "does x belong to the common intersection of a set of n halfspaces F". It is a particularly interesting problem, because it is an example of a decomposable searching problem in the sense of Bentley [3] (see also Saxe and Bentley [19]), to which previously only general dynamization methods were believed to be applicable.

Theorem 6.4. One can dynamically maintain the common intersection of a set of n halfspaces in the plane, such that insertions and deletions can be processed in $O(\log^3 n)$ steps each and queries of the form "does x belong to the current common intersection" can still be answered in $O(\log n)$.

Overmars and van Leeuwen [15] (see also van Leeuwen and Maurer [25]) have recently developed some new dynamization techniques, which specifically apply to structures from which objects can be deleted cheaply and which can be rebuilt fast by exploiting that objects left have remained in sorted order. It leads to a result better than 6.4 on the average for updates, but slightly worse for querying.

Theorem 6.5. One can dynamically maintain the common intersection of a set of halfspaces in the plane, such that over any sequence of n transactions on an initially empty set insertions cost an average of $O(\log n)$ and deletions an average of $O(\log^3 n)$ steps, while queries of the form "does x belong to the current common intersection" can be answered in $O(\log^2 n)$.

The common intersection of a set of halfspaces also plays a role in finding the kernel of a simple polygon. Briefly, the kernel of a polygon is the set of points in its interior from which all sides of the polygon are fully visible. Shamos and Hoey [23] first reported an $O(n \log n)$ algorithm for kernel-determination. Lee and Preparata [13] later showed that knowledge of the ordered contour of the n-gon can be exploited to obtain an $O(n)$ algorithm. We can efficiently maintain the kernel of a dynamically changing polygon, assuming changes merely involve the insertion or deletion of edges which keep the polygon simple.

Theorem 6.6. One can dynamically maintain the kernel of a (simple) n-gon at a cost of only $O(\log^3 n)$ per insertion and deletion of an edge.

A last, but certainly important application involves maintaining the feasible region of a linear program (in the sense of linear programming).

Theorem 6.7. One can dynamically maintain the feasible region of a linear program in two variables at a cost of only $O(\log^3 n)$ for each insertion or deletion of an inequality.

7. Dynamically maintaining the maximal elements of a set.

Let points in the plane be ordered in the usual, coordinate-wise manner. A point x is called maximal in a set S when $x \in S$ and no $y \in S$ exists with $y > x$. The maximal elements of a set form a one-sided contour not unlike a side of a "convex hull" (see figure 12). Kung, Luccio and Preparata [12] have shown that the maximal elements of a static set of n elements in the plane can be determined in $O(n \log n)$ steps, Bentley and Shamos [4] proved it again as an application of their algorithm for ECDF searching. The algorithms are off-line and are, essentially, based on a divide- and- conquer strategy. We will show that the maximal elements of a set can be maintained efficiently as points are added to it one at a time, a result very similar in spirit to Preparata's "real-time" algorithm [16] for convex hull construction.

Theorem 7.1. One can dynamically maintain the maximal elements of a set "real-time", i.e., in only $O(\log n)$ steps for each new point from a collection of n that is added.

To obtain a fully dynamic algorithm, we have to use our earlier technique again. Let us keep points sorted by x- coordinate and let the contour of current maximal elements be stored in a concatenable queue. The following decomposition property can be shown (compare 3.2. and 6.2.)

Theorem 7.2. Let $p_1,\dots,p_n$ be n arbitrary points in the plane, sorted by x- coordinate. Given the representation of the contours of maximal elements of $p_1,\dots,p_i$ and of $p_{i+1},\dots,p_n$ (any $1 \leq i < n$), the representation of the contour of maximal elements of $p_1,\dots,p_n$ can be constructed in O(log n) steps.

Thus, the "composition" of separated contours can now be constructed a factor log n faster than in previous cases. The composite contour again takes very regular pieces off the contours of both halves and a dynamic structure can be devised which operates in very much the same way as it did for convex hull maintenance.

Theorem 7.3. One can dynamically maintain the maximal elements of a set of n points in the plane, at a cost of only $O(\log^2 n)$ steps per insertion and deletion.

As a bonus we obtain a new algorithm for constructing the maximal elements of a static set which, after the initial sorting of all points by x- coordinate, takes only O(n) steps to complete. From theorem 7.3. one may derive that the (decomp.) searching problem "is x a maximal element of the set" can be processed in $O(\log^2 n)$ steps. Applying the general dynamization method of Overmars and van Leeuwen [15] to it, enables one to reduce the maintenance costs to O(log n) for insertions and $O(\log^2 n)$ for deletions on the average at the expense of a query time of $O(\log^2 n)$. The (decomposable) searching problem "is x dominated by an element of the set" can be solved and maintained dynamically at a cost of only $O(\log^2 n)$ per transaction too. Finally, let us say that a set of points A is dominated by another set B when for each $x \in B$ there is a $y \in B$ such that $x \leq y$.

Theorem 7.4. One can maintain two sets A and B in the plane such that insertions and deletions take at most $O(\log^2 n)$ each (where n is the total number of current elements) and the information of whether one dominates the other is kept up-to-date at no extra charge.

8. 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, Inf. Proc. Lett. 8 (1979) 244-251.

[4] Bentley, J.L. and M.I. Shamos, Divide and conquer for linear expected time, Inf. 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] Brown, K.Q., Fast intersection of half spaces, Techn. Rep. CMU-CS-78-129, Carnegie Mellon University (1978).

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

[8] Graham, R.L., An efficient algorithm for determining the convex hull of a finite planar set, Inf. Proc. Lett. 1 (1972) 132-133.

[9] 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.

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

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

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

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

[14] Overmars, M.H. and J. van Leeuwen, Maintenance of configurations in the plane, Tech. Rep. RUU- CS- 79-9 (in press), Dept. of Computer Science, University of Utrecht, 1979.

[15] Overmars, M.H. and J. van Leeuwen, Two general methods for dynamizing decomposable searching problems, Techn. Rep. RUU-CS-79-10, Dept. of Computer Science, University of Utrecht, 1979.

[16] Preparata, F.P., An optimal real-time algorithm

for planar convex hulls, C. ACM. 22 (1979) 402-405.

[17] 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.

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

[19] 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, Oct. 1979, pp 148-168.

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

[21] 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), pp 251-280.

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

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

[24] van Emde Boas, P., On the n log n lowerbound for convex hull and maximal vector determination, Rep. 79-13, Dept. of Math., University of Amsterdam (1979).

[25] van Leeuwen, J. and H.A. Maurer, Dynamic systems of static datastructures, Bericht 42, Institut f. Informationsverarbeitung, TU Graz, 1980.

[26] van Leeuwen, J. and D. Wood, Dynamization of decomposable searching problems, Techn. Rep. RUU-CS-79-5, Dept. of Computer Sci., University of Utrecht, 1979 (to appear in Inf. Proc. Lett).

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

[28] Yao, A.C-C., A lower bound to finding convex hulls, STAN-CS-79-733, Computer Science Dept, Stanford University, 1979.

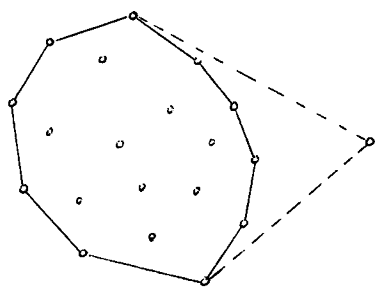
figure 1

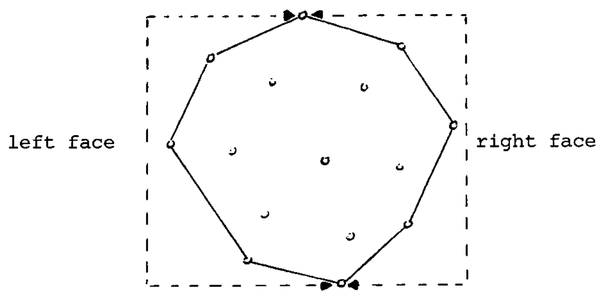

figure 2

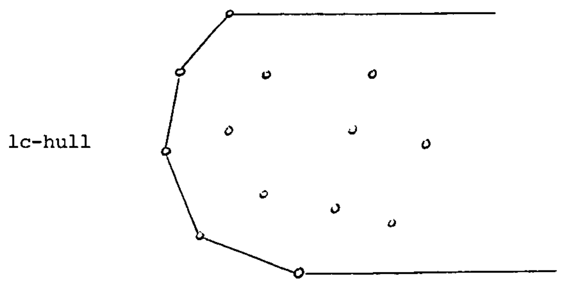

figure 3

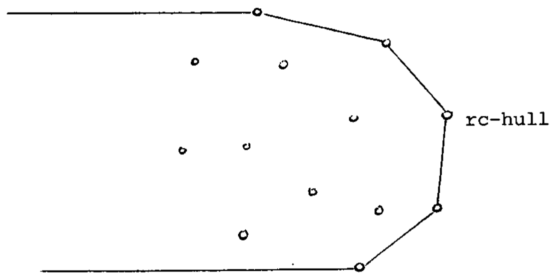

figure 4

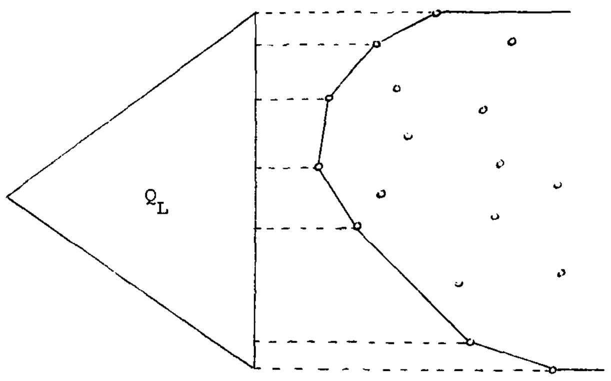

figure 5

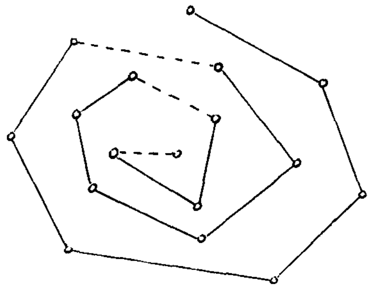

figure 9

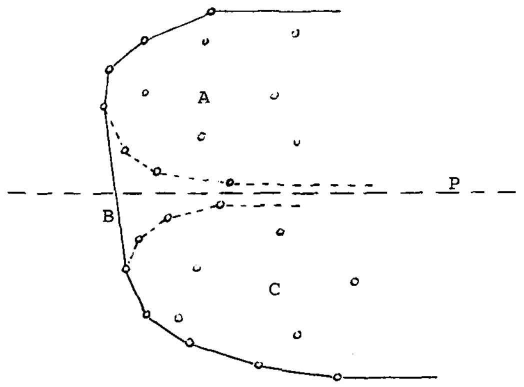

figure 6

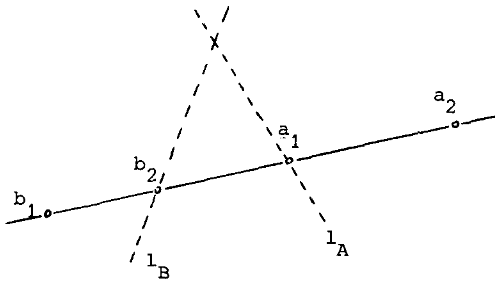

figure 10

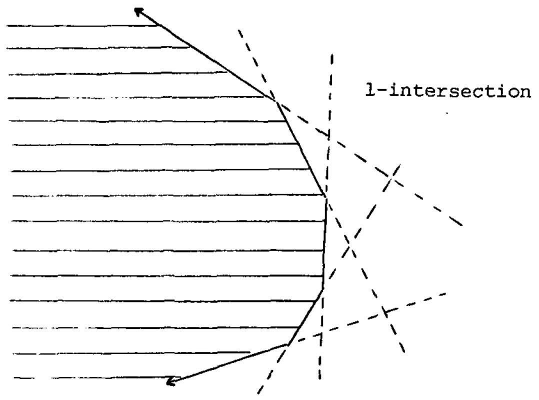

figure 11

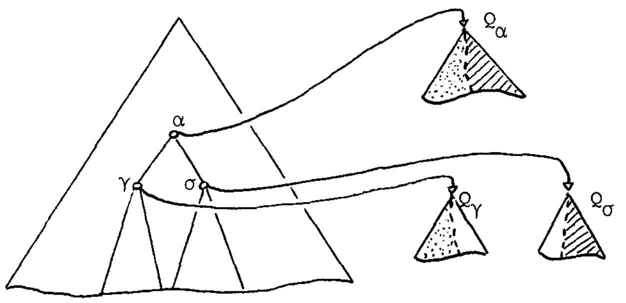

figure 7

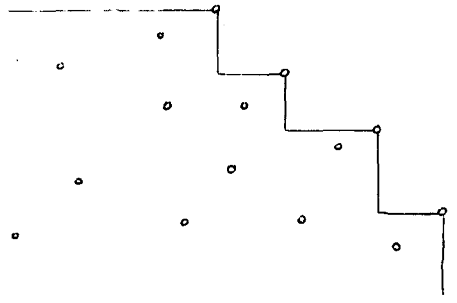

figure 12

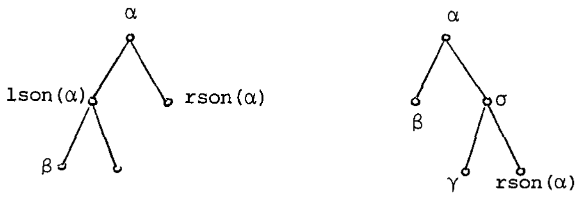

figure 8