On the Convex Layers of a Planar Set

BERNARD CHAZELLE

Abstract—Let S be a set of n points in the Euclidean plane. The convex layers of S are the convex polygons obtained by iterating on the following procedure: compute the convex hull of S and remove its vertices from S. This process of peeling a planar point set is central in the study of robust estimators in statistics. It also provides valuable information on the morphology of a set of sites and has proven to be an efficient preconditioning for range search problems. An optimal algorithm is described for computing the convex layers of S. The algorithm runs in $O(n \log n)$ time and requires $O(n)$ space. Also addressed is the problem of determining the depth of a query point within the convex layers of S, i.e., the number of layers that enclose the query point. This is essentially a planar point location problem, for which optimal solutions are therefore known. Taking advantage of structural properties of the problem, however, a much simpler optimal solution is derived.

I. Introduction

Let $S = \{p_0, \cdots, p_{n-1}\}$ be a set of n points in the Euclidean plane. The set of convex layers of S, denoted $C(S)$ in the following, is the set of convex polygons defined iteratively as follows: compute the convex hull of S and remove its vertices from S (Fig. 1). The convex layers of a point set can be seen as a natural extension of its convex hull. In [17] Shamos mentions applications of this concept to pattern recognition and statistics. A central problem in robust estimation is that of evaluating an unbiased estimator that is not too sensitive to outliers, i.e., observations lying abnormally far from the others. To tackle the two-dimensional version of this problem, Tukey has suggested removing the outliers of a point set by peeling or shelling the set in the manner described above, iterating on this process until only a prescribed fraction of the original points remain [9].

Another illustration of the importance of convex layers in computational geometry has come up recently in the context of a well-known retrieval problem. The halfplane range search problem involves preprocessing n points in the Euclidean plane so that for any query line L, the subset of points lying on a given side of L can be reported effectively. The use of convex layers allowed Chazelle, Guibas, and Lee [6] to derive an optimal solution to this problem.

Manuscript received October 28, 1983; revised January 10, 1985. This work was supported in part by the National Science Foundation under Grant MCS 83-03925 and the Office of Naval Research and the Defense Advanced Research Projects Agency under Contract N00014-83-K-0146 and DARPA Order 4786. The material in this paper was partially presented at the 21st Annual Allerton Conference on Communication, Control, and Computing, Monticello, IL, October 1983.

The author is with the Department of Computer Science, Brown University, Providence, RI 02912, USA.

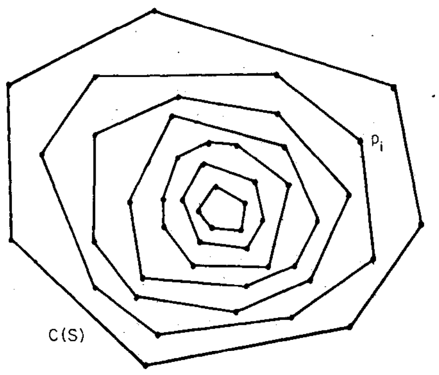

Fig. 1. Convex layers of point set.

Besides its practical relevance, the problem of computing the convex layers of a point set is also interesting in its own right, for it intuitively represents a geometric "equivalent" to sorting. Indeed, considering the various algorithms known for computing the convex hull of a set of points, one is tempted to draw a parallel with sorting algorithms. The Jarvis march [10] resembles selection sort, Bentley and Shamos's method [3] smacks of merge sort, and Eddy's algorithm [7] is strongly reminiscent of quicksort. There is, however, a fundamental difference that often makes computing convex hulls easier than sorting; this is the fact that the output is a convex polygon that may contain only a small fraction of the original points. This is what allows the existence of linear-expected-time algorithms for computing convex hulls under certain distributions of the points [2], [3], [18]. Knowing that similar results are provably impossible to obtain in the case of sorting [1], one can appreciate the intrinsic difference between the two problems. One way of bridging this complexity gap is precisely to require the explicit computation of all the convex layers of the set of points, for it then becomes impossible to take advantage of the possible scarcity of the output in order to bound the time complexity of the problem.

This paper describes an $O(n)$ space, $O(n \log n)$ time algorithm for computing the convex layers of S. Because the convex hull of S is one of the convex layers, computing $C(S)$ requires $\Omega(n \log n)$ time [17], [20]. Our algorithm is therefore optimal. A number of $O(n^2)$ time algorithms for computing convex layers have been found [8], [17], but the most efficient method previously known for this problem requires $O(n \log^2 n)$ time [13]. It is based on a general technique for maintaining the convex hull of a point set in a dynamic environment. Any point can be inserted or deleted in $O(\log^2 n)$ time, while efficient retrieval of the convex hull is possible at any given time. The main contribution of this paper is to show that the deletions involved in the computation of $C(S)$ can be "batched" together (as well as simplified) to give an $O(n \log n)$ overall running time.

We also address the problem of determining the depth of a query point within the convex layers of S, i.e., the number of layers that enclose the query point. This can be reduced to a planar point location problem and can therefore be solved optimally [11], [12]. The specific structure of the problem at hand, however, allows us to use a new point location technique that is both optimal and practical, and avoids many of the complications of previous algorithms.

This paper is organized as follows. In Section II we introduce the basic ingredients of our method and prove a number of preliminary results. In Section III we give a detailed description of the algorithm for computing convex layers, and in Section IV we attack the problem of determining the depth of a query point.

II. The Ingredients

We endow the Euclidean plane with a Cartesian system of reference (Ox, Oy). Each convex boundary in $C(S)$ can be represented as the concatenation of two convex polygonal lines, called upper and lower chains (Fig. 2). Let a (resp. b) denote the point of the convex layer C with minimum (resp. maximum) x-coordinate. The upper (resp. lower) chain of C runs clockwise (resp. counterclockwise) from a to b. Note that the lower and upper chains are the same if C consists of one or two points only. From now on, we will concentrate on the computation of, say, the upper chains of $C(S)$; the other case is strictly similar. Assume without loss of generality, that $p_0, \cdots, p_{n-1}$ appear in this order by nondecreasing x coordinates. Consider the complete binary tree, denoted T, whose leaves are $p_0, \cdots, p_{n-1}$, from left to right. Let $S(v)$ be the set of points stored at the leaves of the subtree of T rooted at node v, and let $U(v)$ be the upper chain of the convex hull of $S(v)$, also referred to as the upper hull of $S(v)$. The union of all the edges in $U(v)$, for all nodes v in T, forms a planar graph G that is easily shown to be connected and acyclic, i.e., forming a free tree. G is called the hull graph of S. Fig. 3 depicts a hull graph and its correspondence with the binary tree T. Each edge of the hull graph is a so-called tangent to two convex chains, and is in one-to-one correspondence with a node of T.

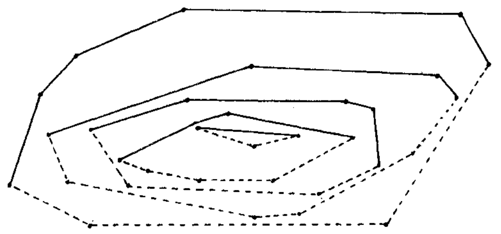
Fig. 2. Upper and lower chains.

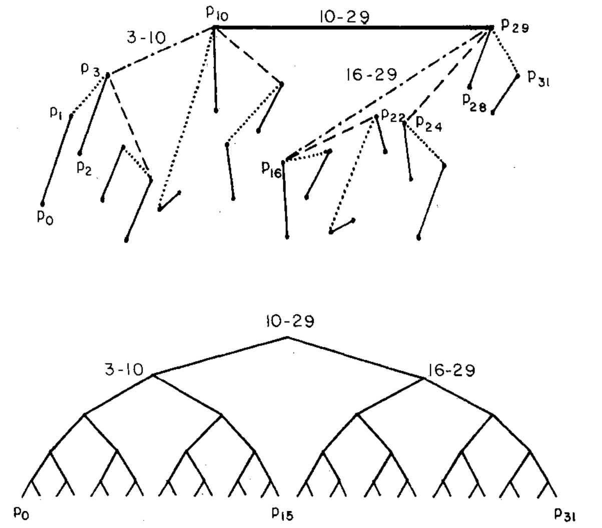

Fig. 3. Hull graph of S.

Note that this data structure is essentially a simplified variant of the structure used by Overmars and van Leeuwen for computing convex hulls dynamically [13]. Similarly to G, we define G' as the hull graph of S with respect to lower chains. Computing G is a straightforward operation, as long as we can efficiently compute the tangent to two upper chains. An algorithm for a very similar task has been described by Preparata and Hong [15], so we only sketch the procedure.

Let U (resp. V) be an upper chain with vertices $u_1, \cdots, u_m$ (resp. $v_1, \cdots, v_p$), in clockwise order. Assume that all the vertices of U have smaller x coordinates than the vertices of V. The tangent to U and V is defined as the unique edge joining U and V in the upper chain of the convex hull of $U_1 \cup U_2$. It is easy to compute this segment s as follows. Set s to $u_1 v_i$, for $i = 1, 2, \cdots$, until both v_{i-1} and v_{i+1} lie below s. The segment s is now tangent to V but, in general, not to U. Next, we take the other endpoint of s to $u_2, u_3, \cdots$ successively, until we reach a vertex u_j for which both u_{j-1} and u_{j+1} lie below s. During the course of this operation, we must be careful to roll the line passing through s around V. Keeping track of the two angles formed by s and U, V allows us to determine the next point to go to in constant time. Since in this process both endpoints move around their respective upper chains in clockwise order, the total running time of the algorithm is $O(m + p)$.

We are now in a position to compute the graph G. Before proceeding, however, let us specify the data structure used for its representation. We use a traditional adjacency-list structure, whereby each vertex p has associated with it a vertex-to-edge list $V(p)$ with the names of its adjacent vertices. Each vertex-to-edge list $V(p)$ consists of two sublists, $V_1(p)$ and $V_2(p)$, defined as follows. $V_1(p)$ (resp. $V_2(p)$) is an angularly sorted doubly linked list containing the vertices adjacent to p with smaller (resp. greater) x coordinates than p (Fig. 4). We define the top

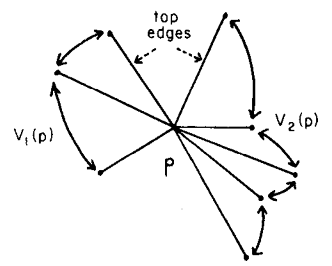

Fig. 4. Vertex-to-edge lists.

edge of $V_1(p)$ (resp. $V_2(p)$) to be the edge with minimum (resp. maximum) slope in $V_1(p)$ (resp. $V_2(p)$). Each of these lists is assumed to be supplied with a pointer to its top edge.

We compute G by divide-and-conquer. Assume that we have already computed the hull graphs of both $\{p_0, \cdots, p_{\lfloor n/2 \rfloor}\}$ and $\{p_{\lfloor n/2 \rfloor+1}, \cdots, p_{n-1}\}$. All we have to do is determine the tangent of the upper hulls of these two sets, using the algorithm described earlier. It is easy to determine the two upper chains on which the tangent computation is based, using the extra pointer supplied with each vertex-to-edge sublist.

III. The Convex Layer Algorithm

The basic idea is to iterate on the following process: make a copy of the names of the vertices on the upper chain, beforehand; then proceed to delete each of them from G, in turn, reconfiguring the hull graph after each deletion; the order in which the deletions are carried out is unimportant. Consider now the graph G' defined with respect to lower hulls. We proceed similarly, i.e., delete the vertices of the lower hull of G'. Next, we must reflect the cross deletions in G and G', that is, delete the lower (resp. upper) hull vertices from G (resp. G'). To distinguish them from cross deletions, we call the first type of deletions direct deletions. One difficulty in these operations is that the deletion of a single vertex can cause a major upset in the hull graph, and its reconstruction can take on the order of n operations. The key feature of the algorithm, however, is to ensure that these upsets always average out to yield an $O(n \log n)$ worst-case running time. Let us first describe direct deletions, and then examine the correctness and complexity of the algorithm. Next, we will show that cross deletions can be viewed as simple instances of direct deletions, so that no drastically different treatment is really needed. We summarize the overall algorithm in high-level form; initially $i = 0$.

Algorithm

```
Convex Layers $(S, G, G')$
  Begin
  If $S = \varnothing$ then stop.
  $U \leftarrow$ upper hull of $S$.
  $L \leftarrow$ lower hull of $S$.
  $i \leftarrow i + 1$.
  $C_i \leftarrow U \cup L$.
  Direct-delete each vertex of $U$ from $G$.
  Direct-delete each vertex of $L$ from $G'$.
  Cross-delete each vertex of $L$ from $G$.
  Cross-delete each vertex of $U$ from $G'$.
  Let $W, H, H'$ be the new settings of $S, G, G'$.
  Convex Layers $(W, H, H')$
  End.
```

A. Direct Deletions

Before describing the algorithm in detail, we will visualize the basic process on a running example. Fig. 5 illustrates the various stages of the deletion of point p_{29} of Fig. 3. The idea is to consider all tangents adjacent to p_{29} one by one and pull them down towards $y = -\infty$. The order in which the tangents are considered is crucial: $p_{29}p_{28}$, $p_{29}p_{31}$, $p_{29}p_{24}$, $p_{29}p_{16}$, $p_{29}p_{10}$. Let the depth of a node of T be its number of ancestors. The tangents to be considered form a subsequence of a leaf-to-root path of T. These nodes are to be considered by decreasing depths. Note that when pulling down $p_{29}p_{16}$, the tangent folds

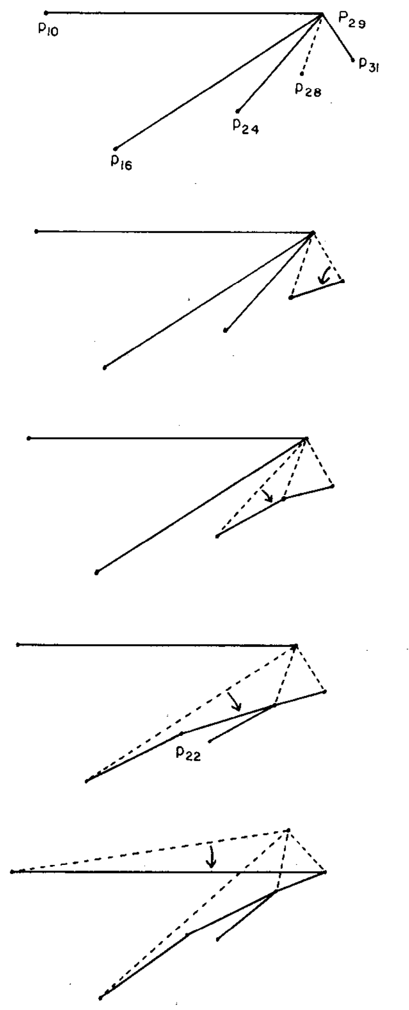

Fig. 5. Deletion in action.

over another vertex, p_{22}. In general, a pulling operation will be accomplished by conceptually replaying the previous two pulling operations. This replay might be necessary in order to guide the current tangent on its way down, and in particular find at little cost the vertices over which it must fold. We are now ready to give the details of the algorithm.

Suppose that we are at an arbitrary stage in the process, and that we wish to delete a vertex p of the upper hull being currently processed. Let $v_1, \cdots, v_l$ be the nodes encountered in T when traversing the path from p to the root. Every v_i corresponds to a tangent edge t_i joining two upper hulls (Fig. 6). Note that p lies on exactly one of

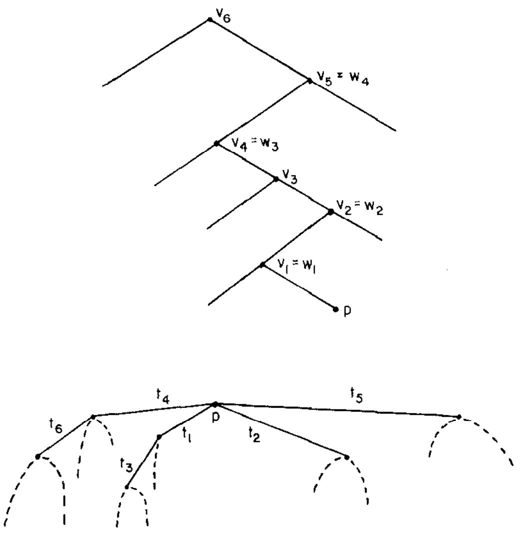

Fig. 6. Computation tree.

these two hulls. It may be the case that because of previous deletions one of these hulls is empty. We can make this case consistent with the others, however, by assuming a dummy vertical tangent t_i. Let $w_1, \cdots, w_k$ be the subsequence of $\{v_1, \cdots, v_l\}$ that corresponds to tangent edges with p as an endpoint. This subsequence can be itself partitioned into the two subsequences formed respectively by the tangent edges "of the form qp" and "of the form pq," from left to right (qp: t_4, t_1 – pq: t_5, t_2 in Fig. 6; these two expressions "of the form qp" and "of the form pq" will often be referred to in the following). It is easy to see that the former (resp. latter) subsequence corresponds exactly to $V_1(p)$ (resp. $V_2(p)$). In the running example of Fig. 5, we have $V_1(p_{29}) = \{p_{10}, p_{16}, p_{24}, p_{28}\}$ and $V_2(p_{29}) = \{p_{31}\}$. An important point to realize is that although G and G' will keep on losing vertices and being reshaped during the computation, the tree T will always remain the same throughout the algorithm. This tree is only a conceptual tool and need not even be implemented explicitly.

The first step consists of retrieving from $V(p)$ the sequence of tangent edges $t_{i_1}, \cdots, t_{i_k}$ corresponding to the nodes $w_1, \cdots, w_k$. This involves merging the lists $V_1(p)$ and $V_2(p)$ according to the following criterion: q_1 precedes q_2 iff the node in T corresponding to the tangent pq_1 is a descendant of the node corresponding to pq_2. This merge is done very easily in $O(l)$ operations by traversing the path from v_1 to v_l in T. In practice, of course, we can emulate the tree T by observing that if $p = p_i$, the path to the root corresponds to the left-to-right sequence of bits in the binary representation of the integer i. The bits of i are to be interpreted with 0 and 1 corresponding, respectively, to left and right turns down the tree. Therefore, the merge follows this prescription: q_1 precedes q_2 iff its index has a longer common prefix with i than the index of q_2 has. For the sake of consistency, we define t_0 as a dummy tangent extending vertically from p towards $y = -\infty$. t_0 will be considered indifferently of type pq or qp, depending on the situation at hand. This addition is meant to ensure that p always has at least one tangent of both types.

For any node v of T define $G(v)$ as the subgraph of G induced by the leaves of the subtree of T rooted at v. Deleting p from G involves updating the sequence of graphs $G(w_1), \cdots, G(w_k)$, in this order. Suppose that we are about to update $G(w_i)$. Without loss of generality we can assume that the corresponding tangent $t = pc$ is of the form pq. Let ap (resp. pb) be the last tangent of type qp (resp. pq) that we processed. Assume for the time being that both a and b are well-defined vertices. Since $G(w_{i-1})$ has been already computed, its upper hull is now available and, as may be noticed, has its portion between a and b lying below the wedge (pa, pb). Let a', b', and c' be the vertices of the hulls currently examined, following a in clockwise order, b in counterclockwise order, and c in counterclockwise order, respectively (Fig. 7). The special cases where $a' = b'$ or $a' = b$ and $b' = a$ can be handled in a uniform manner, so we need not consider these situations separately. We next define the angles $\alpha = \angle(aa', ap)$, $\beta = \angle(bp, bb')$, $\gamma = \angle(cp, cc')$, $\delta = \angle(pc, ap)$, and $\epsilon = \angle(pb, pc)$. Updating $G(w_i)$ involves "pulling down" the vertex p until it disappears from the upper hull of the points in $G(w_i)$. To do so we must compute which of the five angles α, β, γ, δ, ϵ is the first to become null. It is easy to visualize the current process by imaging that pa, pb, and pc are tight rubber bands and that p is being pulled down. The deletion of p from $G(w_i)$ is essentially a simulation of this physical process. To do so we must compute the various placements of p at which one of the rubber bands experiences some break of continuity, i.e., some of the angles above becomes null.

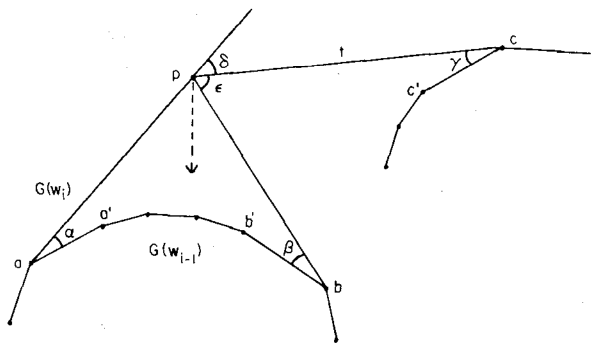

Fig. 7. Close-up of direct deletions.

We can avoid angle calculations altogether if we simply compute the intersection of the vertical line passing through p with the lines passing through aa', bb', and cc' (Fig. 8). The intersection with the largest y coordinate indicates the next placement of p (c^* in Fig. 8), barring the terminal cases $\delta = 0$ or $\epsilon = 0$, which can be checked by testing segment collinearity.

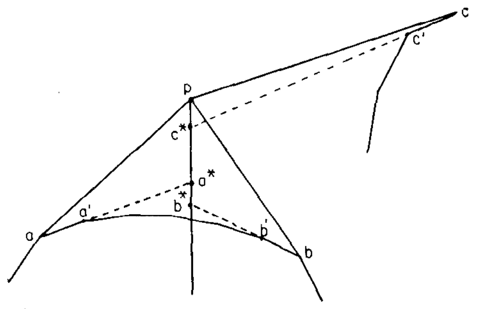

Fig. 8. Avoiding angle calculations.

The vertex a', b', or c' that determines the next location of p is said to be wrapped. We iterate on this process until δ or ϵ becomes null, whichever happens first. Figs. 9(a) and (b) illustrate the two important cases, $\delta = 0$ and $\epsilon = 0$. Note that the addition of the dummy tangent t_0 allows us to treat the case where any of a, b, or c is not properly defined, in similar (yet simpler) ways. We may omit the details of the special cases, which are quite straightforward to handle. As p is being pulled down, the segments pa, pb, and pc must be updated at every step of the computation. Let a^+ (resp. b^+, c^+) denote the running endpoint ($\neq p$) of pa (resp. pb, pc). In general, we can distinguish among three kinds of wrapping; a point wrapped by pa^+, pb^+, or pc^+ is said to be lower wrapped, inner wrapped, or upper wrapped, respectively. A point that is wrapped by both pb^+ and pc^+ is understood as being wrapped solely by pc^+, i.e., to be upper wrapped (Fig. 9(b)).

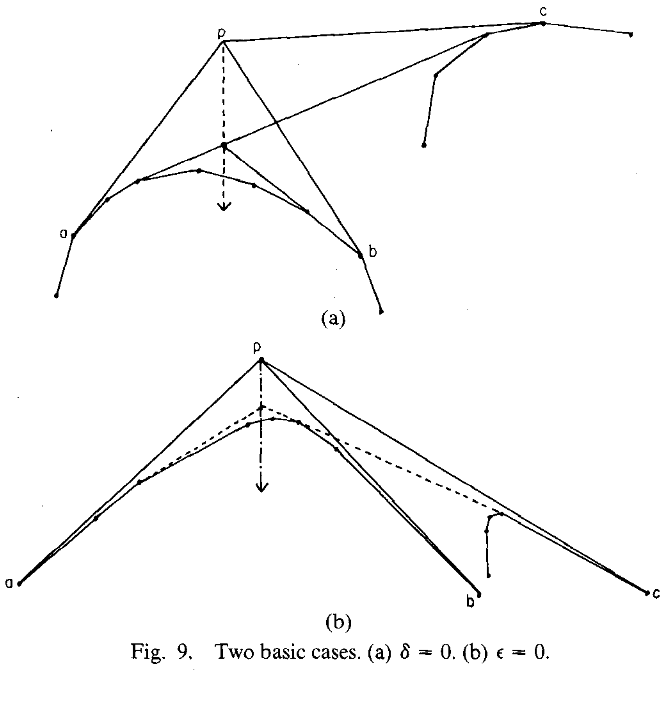

Fig. 9. Two basic cases. (a) $\delta = 0$. (b) $\epsilon = 0$.

It is important to observe that since p is on the upper hull of G, all the edges of the form pa, pb, and pc are top edges in $V_2(a)$, $V_1(b)$, and $V_1(c)$, respectively. This allows us to gain access to the vertices a', b', c' as well as update the lists $V_2(a)$, $V_1(b)$, and $V_1(c)$ in constant time. This remark also applies to pa^+, pb^+, and pc^+ at the generic step of the computation. We conclude by presenting Lemma 1.

Lemma 1: The number of operations involved in updating $G(w_i)$ is proportional to the number of vertices wrapped during the updating.

We will successively prove the correctness of the algorithm and study its complexity.

Lemma 2: The algorithm outlined above correctly computes the new shape of G after the direct deletions of all the vertices on its upper hull.

Proof: Let $C(v)$ be the upper hull of the vertices in $G(v)$. The first observation to make is that removing p from G causes the disappearance of all the edges in $V(p)$, but only these. Furthermore, it causes the introduction of a number of new edges that will all be in $C(v_1), \cdots, C(v_l)$ after the deletion of p. We will show by induction that the algorithm properly updates each $C(v_i)$. Let v_{i-1} and v be the two sons of v_i. We must distinguish between two cases.

1) p is not an endpoint of the tangent associated with v_i (Fig. 10). $C(v_i)$ is still formed by joining $C(v)$ and $C(v_{i-1})$ with the same tangent associated with v_i as before. By induction, $C(v_{i-1})$ will have then be properly updated, so no additional work is required.

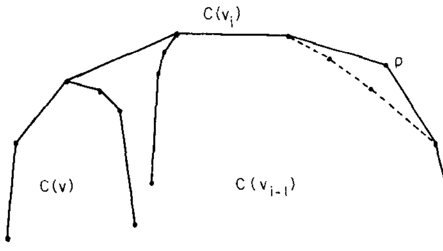

Fig. 10. Point p is not on tangent.

2) The tangent associated with v_i is of the form pq (Fig. 7) (the case qp is strictly similar). The procedure given above correctly computes the new upper hull of the vertices of $G(v) \cup G(v_{i-1})$, provided that the angles α, β, γ, δ, ϵ in Fig. 7 are well defined. γ, δ, and ϵ are already defined in the original setting of G; as for α and β, they originate from the fact that the new setting of $C(v_{i-1})$ differs from the previous one only between a and b, and that the part of $C(v_{i-1})$ between these two vertices lies strictly below the wedge (pa, pb).

This completes the proof.

B. Cross Deletions

We can now look at the cross-deletion process, where the vertices to be deleted are on the lower chain of the convex hull of S. We remove them one at a time, say, in clockwise order, applying the method described previously. A few

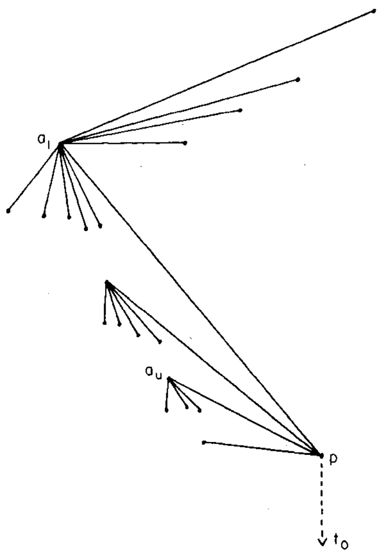

Fig. 11. One of the lists $V_1(p)$ or $V_2(p)$ is empty.

remarks are in order, however. Let p be the vertex to be removed. By construction, it is obvious that the two top edges in $V(p)$, i.e., pa in $V_1(p)$ and pb in $V_2(p)$ always form a convex angle, i.e., $(pa, pb) \leq \pi$. Since on the other hand, p is on the convex hull of S, it follows that one of the lists $V_1(p)$ or $V_2(p)$ has to be empty (i.e., reduced to the dummy edge t_0) (Fig. 11). Note that this remains true for all the lower hull vertices yet to be removed, even after any number of them have been already cross-deleted. This shows that cross deletions correspond to one of the special (easy) cases mentioned earlier.

The second observation has to do with the access to the adjacent vertices of p. Suppose without loss of generality that $V(p) = V_1(p) = \{pa_1, \cdots, pa_u\}$, with pa_1 being the top edge. It is important to be able to gain constant-time access to the location where p is stored in each of the lists $V_2(a_1), \cdots, V_2(a_u)$. As we have seen it earlier, this is indeed a requirement if we wish to preserve the result of Lemma 1, i.e., if we want to ensure that the time to cross-delete p is proportional to the number of wrappings. Unfortunately, it is no longer true that every edge pa_i will be a top edge in $V_2(a_i)$, for p is not on the upper hull any more. The next result suggests a way to circumvent this difficulty.

Lemma 3: At most one edge pa_i $(i = 1, \cdots, u)$ is not a top edge in $V_2(a_i)$.

Proof: We say that a segment t dominates a segment s if they share an endpoint and for every point (x, y) of s there exists a point (x, z) of t such that $z \geq y$. We wish to show that for each i $(1 < i \leq u)$, $a_i p$ is the top edge of $V_2(a_i)$. Suppose that it is not; then there exists a segment $a_i q$ that dominates $a_i p$. But by definition of G, the segment $a_1 p$ dominates $a_i p$; it immediately follows that $a_1 p$ and $a_i q$ intersect, which is impossible.

Since pa_1 is the only edge that does not lend itself to direct access, we can afford to scan the list $V_2(a_1)$ linearly, for it obviously contains no more than $\lceil \log_2 n \rceil$ entries. This will add a term $O(\log n)$ in the time to cross-delete p. We are now ready to examine the complexity of the overall algorithm.

Theorem 1: It is possible to compute the convex layers of a set of n points in $O(n \log n)$ time, using $O(n)$ space.

Proof: Let us first look at the cost of direct deletions. In the course of updating $G(v)$, for $i = 1, \cdots, l$, any vertex either lower- or upper-wrapped is said to be promoted, while any vertex inner-wrapped is said to be confirmed. The crucial observation is that in the course of computing $C(S)$, 1) the number of operations is $O(n \log n +$ total number of promotions and confirmations), and 2) vertices are never confirmed twice in a row, i.e., if they are confirmed more than once, they must be promoted at least once in between each confirmation.

The first point follows directly from previous remarks. To understand the second feature of promotion, let's take a closer look at the wrapping of a point q. In the following we refer to the height of a node of T as the number of edges of its longest path to a leaf of T. Let w be the highest node of T (immediately before the wrapping) such that q is a vertex of $C(w)$. If q is lower-wrapped or upper-wrapped, it immediately becomes a vertex of an upper hull $C(v)$ such that v is an ancestor of w. Node v thus replaces node w as the highest node z of T such that q is a vertex of $C(z)$. Since T has height $\lceil \log_2 n \rceil$, each point cannot be promoted more than $\lceil \log_2 n \rceil$ times.

Let us turn next to the confirmation process. As we mentioned earlier, pulling down ap and bp is a replay of previous action that is only meant to guide the pulling of p (Fig. 7). More precisely, consider any vertex q that is inner-wrapped by pb. At the previous stage, q was lower- or upper-wrapped by pb, therefore that wrapping was a promotion. Consequently any confirmation must be immediately preceded by a promotion. This shows that a point cannot be wrapped more than $2\lceil \log_2 n \rceil$ times, which implies that the overall time required by direct-deletions is $O(n \log n)$.

We can treat cross deletions in an identical way, charging each vertex wrapped in the manner just described. Each deleted vertex may be also charged an extra $O(\log n)$ to account for the linear scan of one of the lists $V_1(*)$ or $V_2(*)$. This completes the proof.

IV. Computing the Depth of a Point

The depth of a query point q within $C(S)$ is defined as the number of convex layers that enclose q (see [13], [17] for practical applications of this notion). Let $S_1, \cdots, S_k$ be the convex layers of S, labeled in increasing order inwards. Rather than computing only the depth of q, let us consider the more general problem of determining the pair of consecutive layers (S_m, S_{m+1}) between which q lies. This pair may be reduced to a single layer if q lies either outside S_1 or inside S_k. The index m will be denoted $m(q)$ from now on, with the convention that $m(q) = 0$ (resp. $m(q) = k$) if q lies outside of S_1 (resp. inside of S_k).

This problem is essentially a planar point location problem, i.e., a problem of determining which region of a straight-line subdivision of the plane contains q. Two optimal algorithms are known for solving this problem. Chronologically, the first is due to Lipton and Tarjan [12]. It relies on a far-reaching theorem on planar graphs, and unfortunately does not lend itself to any kind of reasonably simple implementation. Using a more straightforward approach, Kirkpatrick derived another optimal algorithm [11], which although conceptually quite simple, is yet to be shown fully practical. In [4], [14], Preparata and Bilardi have devised, analyzed, and implemented a near-optimal planar point location algorithm that avoids most of the overhead incurred by the other algorithms. Whereas the first two methods use a linear space data structure to answer any query in $O(\log n)$ time, the algorithms last mentioned, however, require $O(n \log n)$ space for still an optimal $O(\log n)$ response time.

We wish to show here how planar point location in the graph $C(S)$ can be done optimally, combining the simplicity of [4], [14] with the performance of [11], [12]. More precisely, we will present a practical method for computing $m(q)$ in $O(\log n)$ time by using $O(n)$ storage. The algorithm rests on two basic ideas: to begin with, we describe an $O(n \log n)$ space, $O(\log n)$ time method, to which we then apply the concept of filtering search [5] to reduce the space requirement to $O(n)$.

A. The Preliminary Algorithm

Let us organize the set of layers in $C(S)$ into a complete binary tree T, such that each node is associated with a layer of $C(S)$, and an inorder traversal of T corresponds to the order $S_1, \cdots, S_k$. Each node v in T contains a pointer to a data structure, $DS(v)$, built from the layer associated with v, which we denote $S(v)$. In a first stage, we may suppose that $DS(v)$ is simply an array giving a clockwise description of $S(v)$. Let C be a point inside S_k, and let $R(q)$ be the ray emanating from C that passes through the point q. Since C also lies inside the convex polygon $S(v)$, we can use the array $DS(v)$ to compute by binary search the intersection of $S(v)$ and $R(q)$ [17]. This allows us to determine in $O(\log n)$ time whether q lies inside or outside of $S(v)$.

Setting the tree T in this fashion leads directly to an $O(n)$ space, $O(\log^2 n)$ time algorithm, which we can somewhat speed up, at the expense of a factor of log n space. To do so we apply a standard technique [19], which involves adding extra pointers between adjacent nodes of the search tree. Let $R(v)$ be the (angularly) sorted list of rays $R(p)$, for all vertices p of $S(v)$. We next define $L(v)$ as the sorted list of rays in $R(v) \cup D(v)$, with $D(v) = [\cup R(z) |$ all descendents z of $v]$. We can compute all the lists $L(v)$ recursively, in $O(n \log k) = O(n \log n)$ time, simply merging $R(v)$, $L(v_1)$, and $L(v_2)$, where v_1 and v_2 are the children of v.

The next step is to refine the boundary of $S(v)$ with the added information of $L(v)$. More precisely, we augment the set of vertices of $S(v)$ by adding to it all the intersections between $S(v)$ and the rays of $L(v)$. This creates a new boundary, $S^*(v)$, which is simply a "copy" of $S(v)$ supplied with additional vertices. Note that any edge in $S^*(v)$ forms a wedge with respect to C that is covered by exactly one wedge from $S^*(v_1)$ and one wedge from $S^*(v_2)$ (Fig. 12). This allows us to define two pointers from each edge of $S^*(v)$, one to each of the edges in $S^*(v_1)$ and $S^*(v_2)$ that correspond to a covering wedge. The data structure $DS(v)$ can now be defined as a clockwise description of $S^*(v)$ along with this set of extra pointers. It is easy to see that the overall structure can be computed in $O(n \log n)$ time.

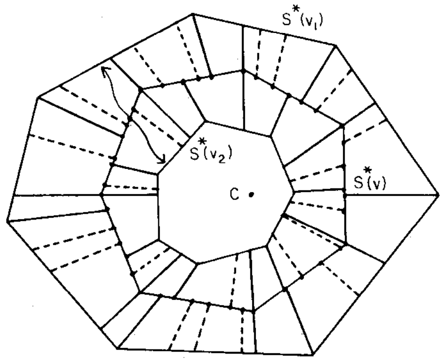

Fig. 12. Simple planar point location.

The point location algorithm will now proceed as follows. Let r be the root of T. Use a binary search to determine, in $O(\log n)$ operations, the edge e of $S^*(r)$ intersected by the ray $R(q)$. Testing the relative position of e and q will indicate whether we should iterate the search in the left or right subtree. Once this has been determined, we can follow one of the extra pointers associated with e so as to avoid any further binary search. This will, indeed, lead us directly to the new edge to be compared against q. This new algorithm will thus yield an $O(\log n)$ search time, but at the price of added storage, i.e., $O(n \log k) = O(n \log n)$.

B. The Optimal Algorithm

The notion of filtering search was developed in [5] as a provision to balance out the various cost factors of enumerating algorithms. These are algorithms that involve the explicit enumeration of the objects of a database that are found to satisfy some property with respect to a query object. For example, one might ask to report which of n given points lie within a query rectangle. The time complexity of problems of this category is often made of two distinguishable cost factors. These are the search time, usually $O(\log n)$ or a polynomial thereof, and the report time, i.e., the number of objects to be reported. Filtering search involves balancing these two quantities, which in essence comes down to making the search technique all the more naive as the number of objects to be reported is large. Another prescription of filtering search is to use the leeway provided by the search time to simplify the supporting data structure. In particular, this encompasses the well-known idea of switching algorithms in midstream, depending on the current size of the input yet to be processed. A good illustration of this idea can be found in Sedgewick's analysis and fine tuning of quicksort [16]. This general technique can be applied to the problem at hand. We begin with a technical result.

Lemma 4: It is possible to find, in $O(n \log n)$ time, a sequence V of convex layers among $\{S_1, \cdots, S_k\}$, such that the total number of vertices in V is $O(n/\log n)$ and at most $4 \log n$ layers can be found between any pair of consecutive layers in V.

Proof: If $k \le \log n$ simply let V be the empty set and the lemma is satisfied. From now on assume that $k > \log n$. Let $H = \{S_{i_1}, \cdots, S_{i_k}\}$ be the convex layers of S, sorted by increasing cardinalities, i.e., the number of vertices in S_{i_j}, denoted α_j, satisfies $\alpha_j \le \alpha_{j'}$, for any $j < j'$. Here is a tentative method for assigning layers to V. Go through each layer of H in turn and assign them to V, with the provision that for each layer assigned we remove from H roughly $\log n$ "neighboring" layers. More precisely, let W denote the complement of V in $C(S)$.

Initially, both sets V and W are empty. Apply the following procedure for $j = 1, \cdots, k$. If S_{i_j} has already been assigned to either V or W, iterate; else, assign S_{i_j} to V, and each layer in P to W, where the set P is defined as follows ($S_x = \varnothing$, if $x \notin \{1, \cdots, k\}$):

$$P = \left\{S_{i_j - \lfloor \log n \rfloor}, S_{i_j - \lfloor \log n \rfloor + 1}, \cdots, S_{i_j - 1}\right\} \cup \left\{S_{i_j + 1}, S_{i_j + 2}, \cdots, S_{i_j + \lfloor \log n \rfloor}\right\}.$$

The procedure has been actually simplified to the extreme so as to emphasize the main idea, and we must now proceed to fill in the details. First of all, the set P must be defined a little more carefully. As mentioned earlier, what we really want to do is "clear out" a certain number of convex layers right next to the current layer picked for membership in V. We set this number tentatively to the value $2\lfloor \log n \rfloor$, but as we want to avoid reassigning previously assigned layers, we redefine P as $P_1 \cup P_2$, with $P_1 = \{S_{i_j - c_1}, S_{i_j - c_1 + 1}, \cdots, S_{i_j - 1}\}$ and $P_2 = \{S_{i_j + 1}, S_{i_j + 2}, \cdots, S_{i_j + c_2}\}$; the quantities c_1 and c_2 are defined as the largest integers not exceeding $\lfloor \log n \rfloor$, such that none of the layers in P_1 and P_2 has been previously assigned. Assigning layers to V and W will result in marking off disjoint intervals in $\{1, \cdots, k\}$. Since, ideally (for reasons to become apparent later on), we would like to see all the values of c_1 and c_2 very near $\lfloor \log n \rfloor$, we want to avoid the situation where a new layer is assigned to V but only a very small interval can be marked off in $\{1, \cdots, k\}$. To circumvent this difficulty, we just have to extend P a little, whenever necessary. More precisely, before assigning the layers in P, we should check whether the longest sequence of yet unassigned layers of the form $\{S_{i_j - c_1 - 1}, S_{i_j - c_1 - 2}, \cdots\}$ (resp. $\{S_{i_j + c_2 + 1}, S_{i_j + c_2 + 2}, \cdots\}$) has a length exceeding $\lfloor \log n \rfloor$; if not, we then append this sequence of layers to P_1 (resp. P_2).

This will ensure that every subsequent interval marked off is of length at least $\lfloor \log n \rfloor$. As a result, the m layers assigned to V partition the sequence $S_1, \cdots, S_k$ into a number of consecutive subsequences of layers in W, each of which has length between $\lfloor \log n \rfloor$ and $4\lfloor \log n \rfloor$. (The 4 comes from the fact that P might be originally of size $2\lfloor \log n \rfloor$ and then be extended with a sequence of $\lfloor \log n \rfloor$ elements on each side.) Assigning layers by increasing cardinalities ensures that for any h-vertex layer assigned to V at any stage, there are at least $h\lfloor \log n \rfloor$ vertices in the layers assigned to W at that stage. It follows that the total number of vertices in the layers assigned to V is $O(n/\log n)$. Ensuring an overall $O(n \log n)$ running time is trivial.

We are now in a position to apply the previous algorithm for computing $m(q)$ in order to report the pair of consecutive layers in V that enclose a query point. The reduced size of V will ensure that only $O((n/\log n)\log(n/\log n)) = O(n)$ space is now needed, instead of $O(n \log n)$. After this preliminary point location, the problem is now to locate q among $O(\log n)$ convex layers. This can be done by computing the intersection of each of these layers with the ray $R(q)$, using the hive graph structure of [5]. This is a simple data structure that allows us to solve the following problem optimally. Given m nonintersecting segments in the plane, report all segments intersecting a query segment whose supporting line passes through the origin. The algorithm requires $O(m)$ storage and gives a response time $O(k + \log m)$, where k is the number of segments to be reported. This allows us to perform the computation needed in our case in $O(\log n)$ time. We can conclude with the main result of this section.

Theorem 2: It is possible to locate a point among the convex layers of n points, and hence compute its depth, in $O(\log n)$ time, using $O(n)$ storage.

We shall note once again that the novelty of this result resides in the fact that the algorithm is both optimal and practical. It is easy to see that a similar technique can be applied to handle planar subdivisions consisting of nonintersecting monotone polygonal lines running from $y = +\infty$ to $y = -\infty$. Whether this method can be adapted to arbitrary planar graphs is open.

V. Conclusions

The main contribution of this work is an optimal algorithm for computing the convex layers of a point set. The algorithm can be seen as a deletion mechanism with optimal amortized cost. The problem of allowing both insertions and deletions in logarithmic update time (amortized or not) remains open. We mention the extension of our techniques to higher dimensions as well as the dynamization of the two-dimensional case as two interesting open problems in the general area of convex hull maintenance.

Acknowledgment

I wish to thank the referees for their comments and suggestions, which helped to improve the readability of this paper.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley, 1974.

[2] S. G. Akl and G. T. Toussaint, "Efficient convex hull algorithms for pattern recognition applications," in *Proc. 4th Int. Joint Conf. Pattern Recognition*, 1978, pp. 483–487.

[3] J. L. Bentley and M. I. Shamos, "Divide-and-conquer for linear expected time," *Inform. Proc. Lett.*, vol. 7, pp. 87–91, 1978.

[4] G. Bilardi, "Average case analysis of an adjacency map searching technique," Tech. Rep. R-928, UILU-ENG-81-2259, Univ. of Ill. at Urbana-Champaign, Dec. 1981.

[5] B. Chazelle, "Filtering search: A new approach to query-answering," in *Proc. 24th IEEE Ann. Symp. Foundations of Computer Science*, 1983, pp. 122–132.

[6] B. Chazelle, L. Guibas, and D. T. Lee, "The power of geometric duality," in *Proc. 24th IEEE Ann. Symp. Foundations of Computer Science*, 1983, pp. 217–225.

[7] W. F. Eddy, "A new convex hull algorithm for planar sets," *ACM Trans. Math. Software*, vol. 3, no. 4, pp. 398–403, 1977.

[8] P. J. Green and B. W. Silverman, "Constructing the convex hull of a set of points in the plane," *Comput. J.*, vol. 22, pp. 262–266, 1979.

[9] P. J. Huber, "Robust statistics: a review," *Ann. Math. Statist.*, vol. 43, no. 3, pp. 1041–1067, 1972.

[10] R. A. Jarvis, "On the identification of the convex hull of a finite set of points in the plane," *Inform. Proc. Lett.*, vol. 2, 1973, pp. 18–21.

[11] D. G. Kirkpatrick, "Optimal search in planar subdivisions," *SIAM J. Comput.*, vol. 12, no. 1, pp. 28–35, 1983.

[12] R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem," *SIAM J. Comput.*, vol. 9, no. 3, pp. 615–627, 1980.

[13] M. H. Overmars and J. van Leeuwen, "Maintenance of configurations in the plane," *J. Comput. Syst. Sci.*, vol. 23, pp. 166–204, 1981.

[14] F. P. Preparata, "A new approach to planar point location," *SIAM J. Comput.*, vol. 10, no. 3, pp. 473–482, 1981.

[15] F. P. Preparata and S. J. Hong, "Convex hulls of finite sets of points in two and three dimensions," *Commun. ACM*, vol. 20, no. 2, pp. 87–93, 1977.

[16] R. Sedgewick, *Quicksort.* New York: Garland, 1978.

[17] M. I. Shamos, "Computational Geometry," Ph.D. dissertation, Yale Univ., New Haven, CT, 1978.

[18] G. T. Toussaint, "Pattern recognition and geometrical complexity," in *Proc. 5th. Int. Conf. Pattern Recognition*, 1980, pp. 1324–1347.

[19] D. E. Willard, "New data structures for orthogonal queries," *SIAM J. Comput.* to appear.

[20] A. C. Yao, "A lower bound to finding convex hulls," *J. ACM*, vol. 28, pp. 780–787, 1981.