SIAM J. COMPUT.
Vol. 20, No. 2, pp. 259–269, April 1991

005

AN $O(n \log^2 h)$ TIME ALGORITHM FOR THE THREE-DIMENSIONAL CONVEX HULL PROBLEM*

HERBERT EDELSBRUNNER† AND WEIPING SHI‡

Abstract. An algorithm is presented that constructs the convex hull of a set of n points in three dimensions in worst-case time $O(n \log^2 h)$ and storage $O(n)$, where h is the number of extreme points. This is an improvement of the $O(nh)$ time gift-wrapping algorithm and, if $h = o(2^{\sqrt{\log_2 n}})$, of the $O(n \log n)$ time divide-and-conquer algorithm.

Key words. computational geometry, convex hull, three dimensions, output sensitive

AMS(MOS) subject classifications. 68U05, 52-04, 52A15

1. Introduction. The *convex hull* of a set of points S in three-dimensional space is the smallest convex set that contains S. If S is finite then the convex hull of S is a convex polytope with vertices, edges, and facets making up its boundary. The *convex hull problem* is to determine the points in S that are vertices of this convex polytope (the *extreme points* of S, or ext (S)), possibly together with some ordering and adjacency information. Finding efficient convex hull algorithms is one of the most intensively studied problems in computational geometry (see Preparata and Shamos (1985) and Edelsbrunner (1987) for more information). Table 1.1 summarizes the current best results on the worst-case complexity of the convex hull problem not including the result of this paper. The $\Omega(n \log n)$[1] lower bounds in two and three dimensions are due to Yao (1981) and simple proofs follow from Ben-Or (1983). Matching upper bound in two and three dimensions can be found in Graham (1972) and in Preparata and Hong (1977). The $\Omega(n^{\lfloor d/2 \rfloor})$ lower bound in $d \geqq 4$ dimensions follows from the

TABLE 1.1

The current best results for finding the convex hull, where n is the number of input points, h is the number of extreme points, F is the number of faces of any dimension, and d is assumed to be a fixed constant. The first part of the table shows the best results that do not depend on h and assume the worst case over all values of h. The second part gives the output-sensitive results.

Dimension	Lower bound	Upper bound
2	$\Omega(n \log n)$	$O(n \log n)$
3	$\Omega(n \log n)$	$O(n \log n)$
$d \geqq 4$	$\Omega(n^{\lfloor d/2 \rfloor})$	$O(n^{\lceil d/2 \rceil})$
		$O(n^{\lfloor d/2 \rfloor} \log n)$
2	$\Omega(n \log h)$	$O(n \log h)$
3	$\Omega(n \log h)$	$O(nh)$
$d \geqq 4$	$\Omega(n \log h + F)$	$O(n^2 + F \log h)$

* Received by the editors February 7, 1990; accepted for publication (in revised form) July 20, 1990.

† Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. The research of this author was supported by National Science Foundation grant CCR-8714565.

‡ Coordinated Science Laboratory and Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. The research of this author was supported by the Semiconductor Research Corporation under contract 88-DP-109.

[1] Throughout the paper, log is base 2 unless otherwise specified.

same lower bound on the maximum number of faces of the convex hull of n points (see, e.g., Edelsbrunner (1987)). The $O(n^{\lceil d/2 \rceil})$ upper bound is a careful implementation of the beneath-beyond method due to Seidel (1981) and is also described in Edelsbrunner (1987). The $O(n^{\lfloor d/2 \rfloor} \log n)$ upper bound is due to Seidel (1986). The $\Omega(n \log h)$ lower bounds in two and three dimensions are given by Kirkpatrick and Seidel (1986). The currently best output-sensitive algorithm in three dimensions, which takes $O(nh)$ time, is the gift-wrapping algorithm of Chand and Kapur (1970). In $d \geqq 4$ dimensions, $O(F)$ is the size of the convex hull, and therefore a trivial lower bound. The $O(n^2 + F \log h)$ time algorithm is due to Seidel (1986).

It is interesting to note that the lower bounds of Table 1.1 in two and three dimensions also hold for the weaker problem of finding all extreme points, without any order and adjacency information. In higher dimensions the complexity of this problem is considerably less than that of constructing the convex hull itself. By solving n linear programs the extreme points can be found in $O(n^2)$ time using results of Megiddo (1984), if the number of dimensions is a fixed constant. At this point, it is worth mentioning that in three dimensions the hard part of the convex hull problem is to identify extreme points; thereafter, $O(h \log h)$ time suffices to construct the adjacency and order information. This allows us to be loose in explaining what exactly our algorithm outputs. We will design it such that it produces all extreme points plus the pairs that define edges of the convex hull, but we will ignore the order of edges around vertices. Because the edges define a three-connected planar graph, $O(h)$ time suffices to find the unique embedding (see Hopcroft and Tarjan (1974) and also Kirkpatrick (1987)).

For the two-dimensional convex hull problem, Kirkpatrick and Seidel (1986) give an $O(n \log h)$ time algorithm and prove it is asymptotically optimal if the complexity of the problem is measured in terms of input and output sizes. They also raise the question of whether there exists an $O(n \log h)$ time algorithm for constructing the convex hull in three dimensions. Up to now, the best deterministic three-dimensional convex hull algorithm whose complexity depends on n and h is the $O(nh)$ time "gift-wrapping" method of Chand and Kapur (1970). Using randomization, a concept we do not consider in this paper, Clarkson and Shor (1989) give an algorithm running in expected time $O(n \log h)$.

This paper presents an $O(n \log^2 h)$ worst-case time algorithm for the three-dimensional convex hull problem. Following Kirkpatrick and Seidel (1986), the algorithm uses the approach "marriage before conquest" in which it first determines how the solutions of the subproblems will combine and then proceeds to solve the subproblems recursively. The main idea of the algorithm is to first project S onto two carefully chosen planes. Then we use the $O(n \log h)$ time two-dimensional convex hull algorithm to find the convex hulls for the projected points. The two two-dimensional convex hulls are projections of edge sequences of the convex hull of S. They are used to partition S into subsets in a balanced way. By recursively finding the convex hulls for each of the subsets, we can get the convex hull of S.

Section 2 presents the algorithm, § 3 assesses its complexity, § 4 remarks on problems caused by points not in general position, and § 5 concludes this paper with a brief discussion of the results.

2. The algorithm. Let S be a finite set of points in three-dimensional space. We assume general position, that is, no four points are coplanar and no three points lie on a common vertical plane. This assumption is algorithmically justified by the conceptual perturbation technique of Edelsbrunner and Mücke (1990). With this assumption,

the convex hull of S is a simplicial polytope, that is, every facet is a triangle. In addition, no edge or facet is vertical.

We call the part of the convex hull of S that can be seen from $(0, 0, \infty)$ the *upper hull* of S (see Fig. 2.1). Similarly, the *lower hull* of S is the part of the convex hull that can be seen from $(0, 0, -\infty)$. Thus, both the upper hull and the lower hull are simply connected subsets of the boundary of the convex hull of S. Indeed, the boundary of the convex hull is the union of the upper hull and the lower hull, while the intersection of the upper hull and lower hull is the cycle of edges that admit vertical supporting planes.

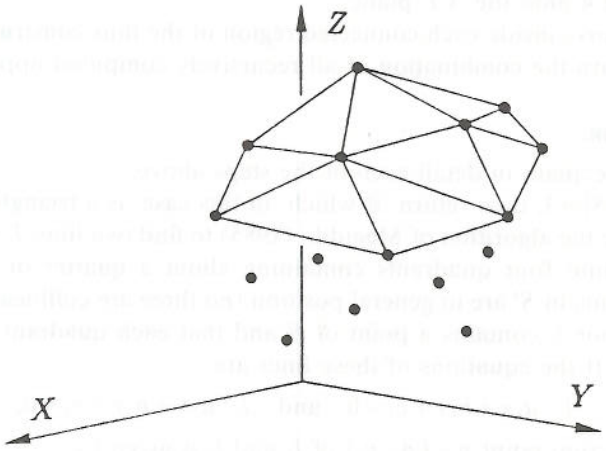

FIG. 2.1. *The upper hull of S.*

The following algorithm shows how to construct the upper hull; the method to construct the lower hull is symmetric. The union can be constructed in $O(h)$ time once we have both the upper and the lower hull. When we describe the procedure we assume that the reader is familiar with the $O(n)$ time three-variable linear programming algorithms of Dyer (1984) or Megiddo (1983), the $O(n)$ time planar ham-sandwich cut algorithm of Megiddo (1985), the $O(n \log h)$ time two-dimensional convex hull algorithm of Kirkpatrick and Seidel (1986), and one of the optimal planar point location algorithms of Kirkpatrick (1983), Cole (1986), Sarnak and Tarjan (1986), and Edelsbrunner, Guibas, and Stolfi (1986).

When we describe the algorithm we use the notational convention that a set primed means the projection of the set onto the XY-plane. For example, $S' = \{(x, y) | (x, y, z) \in S\}$. The nondegeneracy assumption that no three points of S lie in a vertical plane implies that no two points lie on a common vertical line which guarantees a bijection between S and S'.

ALGORITHM 3D_UPPER_HULL $(S, \mathcal{B})$.
Input. S is a set of points in space. $\mathcal{B}$ is a simple closed polygonal curve in space whose vertices form a subset of S. $\mathcal{B}'$, the projection of $\mathcal{B}$ onto the XY-plane, is the boundary of a simple polygon and all points of S' lie on the boundary or inside this polygon.
Output. The part of the upper hull of S whose relative boundary is $\mathcal{B}$, represented as an edge-point adjacency list.

Method
  if $|S| = 3$
0: **then** return $\mathcal{B}$
1: **else** do a 4-division of S' using two intersecting lines in the XY-plane;
2: use three-variable linear programming to find the triangle of the upper hull that intersects the vertical line through the center of the 4-division;
3: project S onto the vertical plane of the first line and compute the two-dimensional upper hull;
4: do the same for the second line;
5: project the two three-dimensional polygonal paths obtained in Steps 3 and 4 onto the XY-plane;
6: recurse inside each connected region of the thus constructed subdivision;
7: return the combination of all recursively computed upper hulls
  endif.
End of Algorithm.

Below, we explain in detail each of the steps above.

Step 0. If $|S| = 3$, then return $\mathcal{B}$ which, in this case, is a triangle in space.

Step 1. Use the algorithm of Megiddo (1985) to find two lines l_1 and l_2 that divide the XY-plane into four quadrants containing about a quarter of the points each. Because the points in S' are in general position (no three are collinear) we can assume that neither l_1 nor l_2 contains a point of S and that each quadrant contains at least $\lfloor |S'|/4 \rfloor$ points. If the equations of these lines are

$$l_1 : a_1 x + b_1 y + c_1 = 0 \quad \text{and} \quad l_2 : a_2 x + b_2 y + c_2 = 0,$$

then the intersection point $p = (p_x, p_y)$ of l_1 and l_2 is given by

$$p_x = -\frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} \quad \text{and} \quad p_y = -\frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}.$$

Let Γ_1 and Γ_2 be the vertical planes that intersect the XY-plane at l_1 and l_2 (see Fig. 2.2).

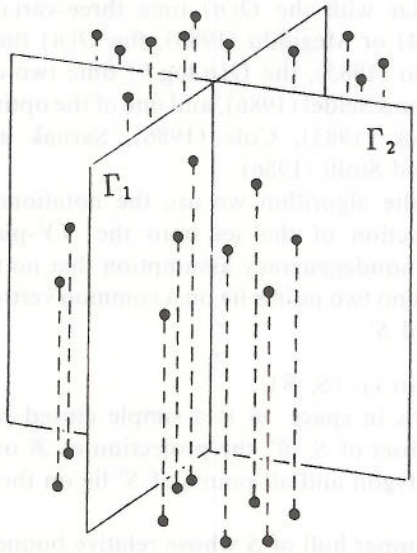

FIG. 2.2. *Cut by two vertical planes,* Γ_1 *and* Γ_2.

Step 2. Use the algorithm of Dyer (1984) or Megiddo (1983) to solve the following three-variable linear programming problem:

$$\text{minimize} \quad \gamma_1 p_x + \gamma_2 p_y + \gamma_3$$
$$\text{subject to} \quad \gamma_1 x_i + \gamma_2 y_i + \gamma_3 \geqq z_i \quad \text{for all } (x_i, y_i, z_i) \in S.$$

The solution to this linear program is a plane $\Gamma: z = \gamma_1 x + \gamma_2 y + \gamma_3$, such that every point of S lies on or below Γ and it has the lowest intersection point with the vertical line through point $p = (p_x, p_y)$ among all those planes. The 4-partition computed in step 1 is such that each quadrant contains at least one point (because $\lfloor |S'|/4 \rfloor \geqq 1$) which implies that, indeed, the vertical line through p intersects the upper hull. We can also assume that p is not collinear with any two points of S' and thus Γ is unique. By our nondegeneracy assumption Γ passes through exactly three points, $a = (a_x, a_y, a_z)$, $b = (b_x, b_y, b_z)$, and $c = (c_x, c_y, c_z)$ of S, and is given by the equation

$$\Gamma: \begin{vmatrix} x & y & z & 1 \\ a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \end{vmatrix} = 0$$

(see Fig. 2.3).

Step 3. Project S onto the plane Γ_1 where the direction of the projection is parallel to $\Gamma \cap \Gamma_2$. Call the obtained point set $S(\Gamma_1)$ and assume a bijection between S and $S(\Gamma_1)$. Using Kirkpatrick and Seidel's algorithm compute the two-dimensional upper hull of $S(\Gamma_1)$ and let H_1 be the corresponding polygonal path in space, that is, the vertices of H_1 are points of S and the projection of H_1 onto Γ_1, along $\Gamma \cap \Gamma_2$, is the upper hull of $S(\Gamma_1)$.

Step 4. Project S onto the plane Γ_2 along the direction parallel to $\Gamma \cap \Gamma_1$. Let the resulting set be $S(\Gamma_2)$ and compute H_2 in complete analogy to H_1 in Step 3.

Step 5. Initialize $H \leftarrow \mathcal{B}$. Add H_1, H_2 and the triangle *abc* to H; H' is a subdivision of the XY-plane. Discard duplicate edges and edges whose projections lie outside $\mathcal{B}'$.

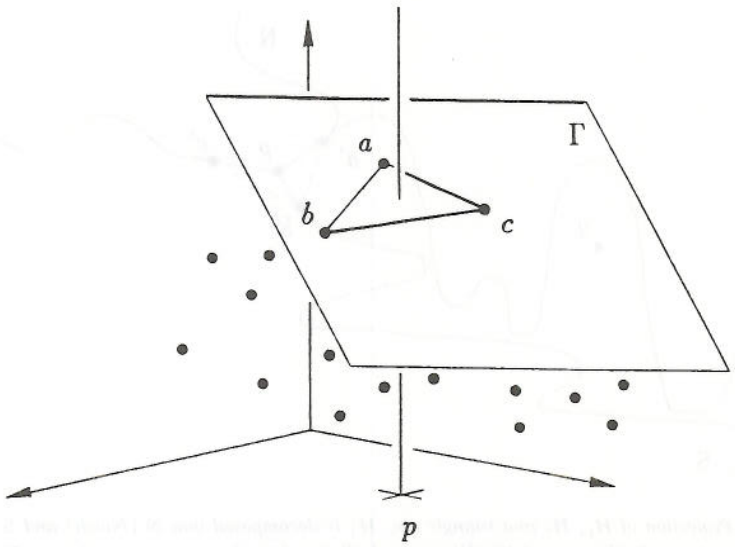

FIG. 2.3. *Use linear programming to find a supporting triangle.*

Build a planar point location data structure based on H'. The data structure can be any one of the optimal methods described by Kirkpatrick (1983), Cole (1986), Sarnak and Tarjan (1986), and Edelsbrunner, Guibas, and Stolfi (1986). Associate each region R_i' of H' with an initially empty set S_i. For every point $q \in S$, decide which region R_i' contains q'; put q into set S_i. We also add points of S to S_i whose projections lie on the boundary of R_i'. Discard points inside $a'b'c'$.

Step 6. For every region R_i' of H' do $H \leftarrow H \cup$ 3D_UPPER_HULL $(S_i, \mathcal{B}_i)$, where $\mathcal{B}_i'$ is the boundary of R_i'.

Step 7. Return H.

Before the first call of the 3D_UPPER_HULL algorithm, we find the convex hull of S' and let $\mathcal{B}$ be the three-dimensional polygonal curve that projects onto its boundary.

Remark. The fairly complicated general point location method in Step 5 can be avoided if we exploit special properties of H_1' and H_2'. The partitioning of S into subsets S_i can be done in linear time since the polygonal paths, H_1' and H_2', are monotone and the two-dimensional convex hull algorithm of Kirkpatrick and Seidel (1986) not only produces H_1 and H_2, but also "sorts" S into $|H_1|$ and $|H_2|$ buckets. We spend a paragraph explaining how this works.

Assume for simplicity that l_1 is the Y-axis and l_2 is the X-axis. Then H_1' is monotone in the Y-direction and H_2' is monotone in the X-direction. Consider the regions of the subdivision formed by H_1' and H_2' that lie to the left of H_1' and below H_2' (see Fig. 2.4 where H_1' consists of the branches labeled N and S and H_2' consists of the branches labeled E and W). Observe that there is a polygonal path Q that is X- and Y-monotone and separates H_1' from H_2' (dotted line in Fig. 2.4); it can be constructed in linear time from either H_1' or H_2'. If a point q lies in region R_i then the vertical line through q meets Q somewhere inside R_i or the horizontal line through q does so. Thus, after doing binary search twice, once in H_1' and once in H_2', we can narrow down the search to at most two regions. If q lies below or to the right of Q then the horizontal line has priority over the vertical line, and vice versa if q lies above or to the left of Q. The two binary searches are done implicitly in the two-dimensional convex hull construction and thus add no extra time to the algorithm.

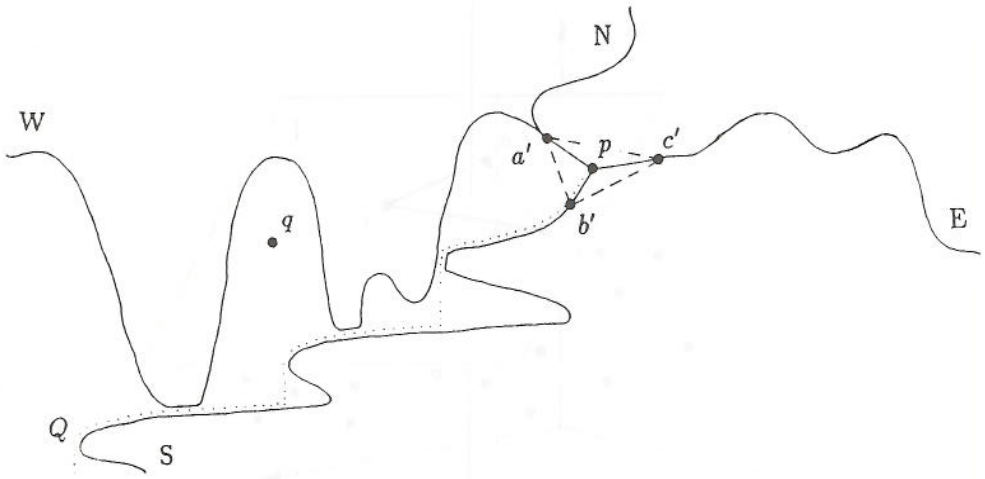

FIG. 2.4. *Projection of H_1, H_2 and triangle abc. H_1' is decomposed into* N (*North*) *and* S (*South*) *and H_2' is decomposed into* E (*East*) *and* W (*West*), *and all four branches are connected to p. Note that some branches share common parts.*

In the remainder of this section, we argue that the algorithm correctly finds the part of the upper hull of S whose projection onto the XY-plane lies in $\mathcal{B}'$. We use induction on the number of points in S. If $|S|=3$, the algorithm is trivially correct. When $|S|>3$, every edge e (and therefore every vertex) on H_1 and H_2 must also be on the upper hull of S since there exists a plane that passes through e and has all other points of S below it. For the same reason triangle abc must be on the upper hull. Points inside triangle $a'b'c'$ cannot be on the upper hull because all these points are under the facet abc and can therefore not be seen from $(0, 0, +\infty)$. It remains to show that by partitioning S into subsets the algorithm does not lose any edges of the upper hull.

Recall that every facet is a triangle because we assume that no four points are coplanar. If there were an edge p_ip_j on the upper hull and $p_i \in S_i$ and $p_j \in S_j$, $i \neq j$, then the projection onto the XY-plane of the edge p_ip_j must cross some edge of H_1', H_2', $\mathcal{B}'$ or triangle $a'b'c'$. We then have at least four points on the same plane, violating the nondegeneracy assumption.

Remark. It is not absolutely necessary that the points are in general position for our algorithm to work. However, with the nondegeneracy assumption it is significantly simpler to describe, to understand, and to code.

3. Analysis. Finally, we are ready to assess the time-complexity of the algorithm. We start by proving a general lemma on the total cost of certain trees. Then we show by a geometric argument that the cost of the algorithm can be modeled by such a tree and finally conclude with the main result of this paper.

DEFINITION. Let $T=(V, E)$ be a rooted tree with a cost function $c: V \to [0, +\infty)$. If there is a constant $\alpha \in (0, 1)$, such that $c(\mu) \leqq \alpha c(\nu)$ for every node μ and its parent ν, then we say the cost function c is *fading* and α is the *fading factor.*

LEMMA 3.1. *In a rooted tree $T=(V, E)$ with fading cost function c and fading factor α, if for each level the sum of the costs of all nodes at this level is bounded by C, then the sum of the costs of all nodes in the tree is at most $C(\log_{1/\alpha}|V|+O(1))$.*

Proof. The proof consists of two steps. We first change the shape of the tree by repeated application of a path compression operation without increasing its total cost nor violating the level cost bound. Then we claim that after changing the shape, the tree has height $\log_{1/\alpha}|V|+O(1)$ and finish the proof.

Path compression operation. Arbitrarily pick a node v from the bottommost level and make it a new child of one of its ancestors.

It is clear that after the application of this operation T is still a tree, c is still fading, and the total cost of the tree did not change. To change the tree we number its levels 0, 1, 2, etc. with the root at level zero. If some level i that is not the bottommost level has fewer than $\lfloor \alpha^{-i} \rfloor$ nodes, then we apply the path compression operation and make v a child of its ancestor at level $i-1$. This is done until level i has at least $\lfloor \alpha^{-i} \rfloor$ nodes, for each but the bottommost level i. The cost of level i increases only if in the end it has exactly $\lfloor \alpha^{-i} \rfloor$ nodes (except for the bottommost level which may have fewer nodes). These $\lfloor \alpha^{-i} \rfloor$ nodes have a cost that does not exceed the cost of the root and is therefore not greater than C.

Now consider the total number of nodes in the resulting tree, where l is the height of the tree (so level l is the bottommost level). We have

$$|V| \geqq \sum_{i=0}^{l-1} \lfloor \alpha^{-i} \rfloor \geqq \sum_{i=0}^{l-1} \alpha^{-i} - l = \frac{(1/\alpha)^l - 1}{1/\alpha - 1} - l \geqq \frac{(1/\alpha)^l - 1}{1/\alpha - 1} - |V|.$$

It follows that $l \leqq \log_{1/\alpha}|V| + \log_{1/\alpha} 2 + 1$, and that the total cost of the tree is at most $C(\log_{1/\alpha}|V|+O(1))$. □

In order to apply Lemma 3.1 to the algorithm of the preceding section, think of the algorithm as a rooted tree whose nodes correspond to recursive calls. The cost of a node is the time spent at this node. We will be able to argue that this cost is fading if we can show that the number of points decreases by a constant factor from one level of the recursion to the next.

LEMMA 3.2. *In Step 5 of the algorithm, each set S_i contains at most $\lceil 3|S|/4 \rceil$ points.*

Proof. Denote the four quadrants defined by l_1 and l_2 as NE, NW, SW, and SE. We claim that each set S_i contains points from at most three of the four quadrants. The assertion follows because each quadrant contains at least $\lfloor |S|/4 \rfloor$ of the points in S.

Note that no line parallel to l_1 intersects H_2' more than once, and that no line parallel to l_2 intersects H_1' more than once. Take a region R of the subdivision defined by H_1' and H_2'; its boundary can be decomposed into two connected chains, one in H_1' and one in H_2'. To show that R cannot intersect all four quadrants, we remove $a'b'c'$, and call the remaining branches of H_1' N (North) and S (South) and those of H_2' E (East) and W (West). Next, we connect each branch to point p by a straight line segment, as shown in Fig. 2.4. These modifications can only increase regions of the subdivision. Region R is bounded by only two of the four branches, say N and E, and cannot intersect SW because N intersects only NE and NW and E meets only NE and SE. □

Using the two lemmas we are ready to give the analysis of the algorithm still assuming general position of the points.

THEOREM 3.3. *The algorithm described in the previous section constructs the convex hull of a set S of n points in three-dimensional space in time $O(n \log^2 h)$ and storage $O(n)$ in the worst case, where $h = \text{ext}(S)$.*

Proof. Let $T(n, h)$ be the time-complexity of the algorithm, write S_i for the recursively considered subsets, and set $n_i = |S_i|$ and $h_i = |\text{ext}(S_i)|$. Then

$$T(n, h) = \begin{cases} O(n) & \text{if } h \leqq 3, \\ O(n \log h) + \sum_i T(n_i, h_i) & \text{otherwise.} \end{cases}$$

Think of $T(n, h)$ as a node in a tree, with cost $n \log h$ and children $T(n_i, h_i)$. Every time we recurse we find some new edges or facets on the convex hull, since otherwise there is no way to partition S into proper subsets which would contradict Lemma 3.2. So the number of nodes in the tree is at most the number of edges and facets on the hull which is $O(h)$.

Now, increase the cost of each node in the tree from $n_i \log h_i$ to $n_i \log h$. If c is the cost of a node then, by Lemma 3.2, the cost of each child is at most $\lceil 3c/4 \rceil$ and therefore the cost is fading.

At any one level of the tree, the algorithm works on different polygonal regions $R_1', R_2', \cdots, R_k'$ with boundaries $\mathcal{B}_1', \mathcal{B}_2', \cdots, \mathcal{B}_k'$. Let $|R_i'|$ be the number of points in R_i' (that is, inside and on $\mathcal{B}_i'$), and $|\mathcal{B}_i'|$ be the number of vertices of $\mathcal{B}_i'$. For the total cost at this level we have

$$\sum_{i=1}^{k} |R_i'| \log h \leqq \left(n + \sum_{i=1}^{k} |\mathcal{B}_i'| \right) \log h.$$

Since each $\mathcal{B}_i'$ is a simple polygon, each edge of the (planar) subdivision defined by

the $\mathcal{B}_i'$ occurs in at most two polygon boundaries on this level. Hence,

$$\sum_{i=1}^{k} |\mathcal{B}_i'| \leqq 6h - 12 \leqq 6n.$$

This implies that the cost at each level is bounded by $7n \log h$. Since the cost is fading, at each level it is bounded by $O(n \log h)$, and the total number of nodes in the tree is $O(h)$, we have $T(n, h) = O(n \log^2 h)$ by Lemma 3.1.

The $O(n)$ storage can be guaranteed if we declare H, $\mathcal{B}$, and S as global variables. For other subroutine calls, such as 4-partition, linear programming, two-dimensional convex hull and point location, $O(n)$ storage suffices and it is immediately returned after each call. □

Remark. To demonstrate that the above analysis is tight, we show that there are point sets for which the algorithm in § 2 indeed takes $\Theta(n \log^2 h)$ time. To see this consider the situation where the projections onto the XY-plane of extreme points form a $\sqrt{h}$-by-$\sqrt{h}$ grid and nonextreme points are evenly distributed in the grid. Since each call to the two-dimensional convex hull algorithm takes $\Theta(n \log h)$ time, and the depth of recursive calls is $\Theta(\log h)$, the algorithm described in § 2 runs in $\Theta(n \log^2 h)$ time.

4. Coping with degenerate point sets. As remarked earlier, point sets that are not in general position can be (conceptually) perturbed to satisfy the general position assumption with various definitions of this notion. For all details we refer to Edelsbrunner and Mücke (1990) where such a perturbation method is described. Still, there are two questions that need to be answered. First, how can we make sure that the perturbation does not change S in a way that significantly changes its convex hull and possibly increases the number of extreme points? Second, how does the perturbation affect the implementation of the necessary primitive operations?

We start with a brief review of the main features of the conceptual perturbation method, called SoS, of Edelsbrunner and Mücke (1990). SoS simulates an infinitesimal perturbation of the point coordinates that removes all degeneracies relevant to the algorithm of § 2. This is done by perturbing each coordinate differently, that is, there is a sequence of the coordinates so that the amount of perturbation of a coordinate is astronomically smaller than that of the preceding coordinates. Furthermore, because the perturbation is arbitrarily small everywhere, extreme points remain extreme and interior points remain interior. However, a point that is not extreme but lies on the boundary of the convex hull will end up either as an extreme point or in the interior of the convex hull. The decision made by SoS is based on the point and coordinate indices, arbitrarily assigned, and thus is by and large arbitrary.

If we accept this arbitrariness then the main theorem of this paper still applies, even if the point set S is not in general position. However, h must be redefined as the number of points that lie on the boundary of the convex hull, rather than the number of vertices of the convex hull. Another way to deal with nonextreme points on the boundary of the convex hull is to devise a sequence of the coordinates that guarantees that nonextreme points are perturbed below the upper hull. Such schemes are possible but tedious, and we refer to Rosenberger (1990, Chap. 5.2), where related ideas are explicated for the two-dimensional convex hull problem.

The next and related issue is how to simulate the perturbation. This is discussed at length in Edelsbrunner and Mücke (1990), except that they do not cover all primitive operations needed for our algorithm. The tricky part is in Steps 3 and 4 of our algorithm, where points are projected onto vertical planes Γ_1 and Γ_2. These planes as well as the directions of projection depend on the input points. As a consequence, the primitive

operations in the two-dimensional convex hull constructions are significantly more involved than in the plain plane case, although still of constant size. A detailed study of these operations together with possible simplifications is left as a future project.

5. Discussion. This paper presents an algorithm for constructing the convex hull of n points in three-dimensional space in worst-case time $O(n \log^2 h)$ and storage $O(n)$, where h is the number of vertices of the convex hull. It should be mentioned that the hidden constant in the big-O notation is rather large, although not astronomical. This is because the algorithms for the planar 4-partition, three-dimensional linear programming, and output-sensitive two-dimensional convex hull construction used in our algorithm all have large multiplicative constants.

It seems natural to ask if the time-complexity can be further reduced to $O(n \log h)$. The bottleneck of our method is the construction of two-dimensional hulls. All other operations can be done in time $O(n)$. It might also be interesting to see if the algorithm of this paper can be extended to four and higher dimensions; compare to the $O(n^2 + F \log h)$ algorithm of Seidel (1986).

Acknowledgments. The second author thanks Franco P. Preparata and Bernard Chazelle for discussions on the problem of this paper.

REFERENCES

M. BEN-OR (1983), *Lower bounds for algebraic computation trees*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, pp. 80–86.

D. R. CHAND AND S. S. KAPUR (1970), *An algorithm for convex polytopes*, J. Assoc. Comput. Mach., 17, pp. 78–86.

K. L. CLARKSON AND P. W. SHOR (1989), *Applications of random sampling in computational geometry*, II, Discrete Comput. Geom., 4, pp. 387–421.

R. COLE (1986), *Searching and storing similar lists*, J. Algorithms, 7, pp. 202–220.

M. E. DYER (1984), *Linear time algorithms for two- and three-variable linear programs*, SIAM J. Comput., 13, pp. 31–45.

H. EDELSBRUNNER, L. J. GUIBAS, AND J. STOLFI (1986), *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15, pp. 317–340.

H. EDELSBRUNNER (1987), *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, Germany, 1987.

H. EDELSBRUNNER AND E. P. MÜCKE (1990), *Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms*, ACM Trans. Graphics, 9, pp. 66–104.

R. L. GRAHAM (1972), *An efficient algorithm for determining the convex hull of a finite planar set*, Inform. Process. Lett., 1, pp. 132–133.

J. E. HOPCROFT AND R. E. TARJAN (1974), *An efficient algorithm for graph planarity*, J. Assoc. Comput. Mach., 21, pp. 549–568.

D. G. KIRKPATRICK (1983), *Optimal search in planar subdivisions*, SIAM J. Comput., 12, pp. 28–35.

——— (1987), *Establishing order in planar subdivisions*, in Proc. 3rd Annual Symposium on Computational Geometry, Association for Computing Machinery, New York, pp. 316–321.

D. G. KIRKPATRICK AND R. SEIDEL (1986), *The ultimate planar convex hull algorithm?*, SIAM J. Comput., 15, pp. 287–299.

N. MEGIDDO (1983), *Linear-time algorithms for linear programming in R^3 and related problems*, SIAM J. Comput., 12, pp. 759–776.

——— (1984), *Linear programming in linear time when the dimension is fixed*, J. Assoc. Comput. Mach., 31, pp. 114–127.

——— (1985), *Partitioning with two lines in the plane*, J. Algorithms, 3, pp. 430–433.

F. P. PREPARATA AND S. J. HONG (1977), *Convex hulls of finite sets of points in two and three dimensions*, Comm. Assoc. Comput. Mach., 20, pp. 87–93.

F. P. PREPARATA AND M. I. SHAMOS (1985), *Computational Geometry—an Introduction*, Springer-Verlag, New York.

H. ROSENBERGER (1990), *Degeneracy control in geometric programs*, Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, IL.

N. SARNAK AND R. E. TARJAN (1986), *Planar point location using persistent trees*, Comm. Assoc. Comput. Mach., 29, pp. 669–679.

R. SEIDEL (1981), *A convex hull algorithm optimal for point sets in even dimensions*, Tech. Report 81-14, Department of Computer Science, University of British Columbia, British Columbia, Canada.

——— (1986), *Constructing higher-dimensional convex hulls at logarithmic cost per face*, in Proc. 18th Annual ACM Symposium on Theory Computing, Association for Computing Machinery, New York, pp. 404–413.

A. C. YAO (1981), *A lower bound to finding convex hulls*, J. Assoc. Comput. Mach., 28, pp. 780–789.

SIAM J. COMPUT.
Vol. 20, No. 2, pp. 270–277, April 1991

006

A GENERAL SEQUENTIAL TIME-SPACE TRADEOFF FOR FINDING UNIQUE ELEMENTS*

PAUL BEAME†

Abstract. An optimal $\Omega(n^2)$ lower bound is shown for the time-space product of any R-way branching program that determines those values which occur exactly once in a list of n integers in the range $[1, R]$ where $R \geqq n$. This $\Omega(n^2)$ tradeoff also applies to the sorting problem and thus improves the previous time-space tradeoffs for sorting. Because the R-way branching program is such a powerful model, these time-space product tradeoffs also apply to all models of sequential computation that have a fair measure of space such as off-line multitape Turing machines and off-line log-cost random access machines (RAMs).

Key words. lower bounds, time-space tradeoff, computational complexity, sorting, branching programs

AMS(MOS) subject classifications. 68P10, 68Q10, 68Q25

1. Introduction. The goal of producing nontrivial lower bounds on the time or space complexity for specific computational problems in $\mathcal{NP}$ has largely been elusive Also, concentration on a single resource does not always accurately represent all of the issues involved in solving a problem. For some computational problems it is possible to obtain a whole spectrum of algorithms within which one can trade time performance for storage or vice versa. Thus the question of obtaining lower bounds that say something about time and space simultaneously has received considerable study as well.

The most interesting model for studying time-space tradeoff lower bounds that has been developed is the R-way branching program model. The R-way branching program is an unstructured model of computation that has unrestricted random access to its inputs and which makes no assumption about the way its internal storage is managed. The model is powerful enough that lower bounds proven in it apply to a wide variety of sequential computing models including off-line multitape Turing machines with random-access input heads. A particularly convenient model for which the lower bounds for R-way branching programs apply is that of a random access machine (RAM) with its input stored in a read-only memory, with a unit-cost measure of time and with its read–write storage charged on a log-cost basis.

The R-way branching program model was introduced by Borodin and Cook [BC82], who used it in showing the first nontrivial general sequential time-space tradeoff lower bound for any problem. They showed that any R-way branching program requires a time-space product of $\Omega(n^2/\log n)$ to sort n integers in the range $[1, n^2]$.

Since [BC82], time-space tradeoff lower bounds on R-way branching programs have been shown for a number of algebraic problems such as discrete Fourier transforms, matrix-vector products, and integer and matrix multiplication [Yes84], [Abr86]. In addition to these results, Reisch, in [RS82], has claimed an improvement of the sorting lower bound to $\Omega(n^2 \log\log n/\log n)$ using the same approach as in [BC82]. [RS82] presents an improvement of only one of the two key lemmas in [BC82]; this change appears to necessitate an overhaul of the second, more complex lemma as well in order to obtain the claimed bound. However, even this bound leaves a gap between the upper and lower bounds for sorting.

* Received by the editors March 16, 1989; accepted for publication (in revised form) June 28, 1990. This research was supported by National Science Foundation grant CCR-8858799.

† Computer Science Department, FR-35, University of Washington, Seattle, Washington 98195.