12-1-1993

Convex Hulls: Complexity and Applications (a Survey)

Suneeta Ramaswami
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-97.

Convex Hulls: Complexity and Applications (a Survey)

Abstract
Computational geometry is, in brief, the study of algorithms for geometric problems. Classical study of geometry and geometric objects, however, is not well-suited to efficient algorithms techniques. Thus, for the given geometric problems, it becomes necessary to identify properties and concepts that lend themselves to efficient computation. The primary focus of this paper will be on one such geometric problems, the Convex Hull problem.

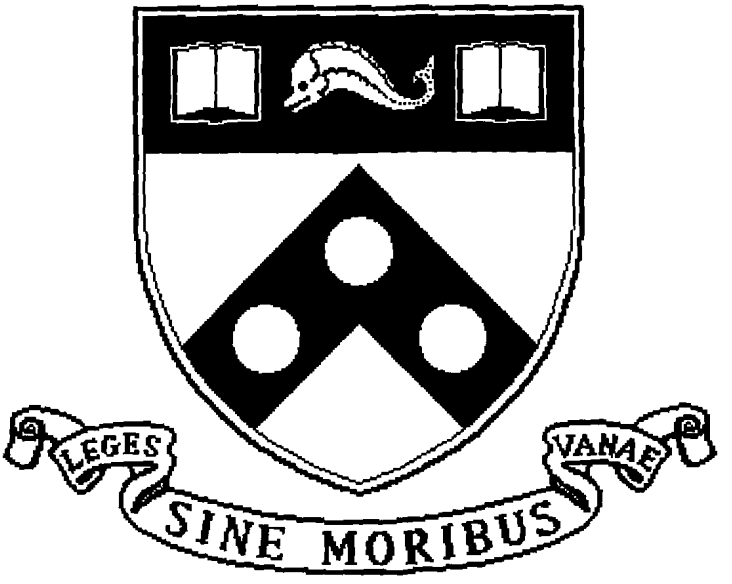
Disciplines
Theory and Algorithms

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-97.

Convex Hulls: Complexity and Applications (A Survey)

MS-CIS-93-97

Suneeta Ramaswami



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

December 1993

Convex Hulls: Complexity and Applications (A Survey)

Suneeta Ramaswami

April 1990

Contents

List of Figures

1 Introduction

Computational geometry is, in brief, the study of algorithms for geometric problems. Classical study of geometry and geometric objects, however, is not well-suited to efficient algorithmic techniques. Thus, for the given geometric problem, it becomes necessary to identify properties and concepts that lend themselves to efficient computation. The primary focus of this paper will be on one such geometric problem, the Convex Hull problem.

It is safe to say that the convex hull problem is one of the most extensively studied and well-understood problems in computational geometry. The study of efficient algorithms to compute convex hulls had started even before the emergence of computational geometry as an area of research in its own right. The reason for the extensive study of this problem is its theoretical and practical significance. Problems in computer graphics, image processing, pattern recognition, and statistics are, to mention but a few, some of the areas in which the convex hull of a finite set of points is routinely used. In addition, the computation of the convex hull arises as an intermediate step in many problems in computational geometry.

The concept of the convex hull of a set of points is intuitive and easy to grasp, and finding it seems to be a fairly straightforward task. However, in order to develop algorithms, we need a *constructive* approach for solving the problem, and it is not immediately obvious what this approach might be. In this paper, we will discuss some of the representative approaches. Before we do that, however, we address the important question of lower bounds to the complexity of the problem.

It is straightforward to show that sorting is linear time reducible to the problem of finding convex hulls, which immediately gives us an $O(N \log N)$ lower bound. More importantly, it can be shown that just identifying the points that lie on the hull (i.e. without specifying the order in which they occur) also takes at least $O(N \log N)$ time. However, the argument to prove this fact is not as elementary. This leads us to an extremely important result by Steele and Yao [22] which, when combined with the powerful algebraic decision-tree model of Ben-Or [5], gives us the stated lower bound. The proof for this result is fairly complex. We have devoted considerable attention to it because the proof technique involved is applicable to a large class of algorithms, and is particularly well-suited to problems in computational geometry.

We will restrict our attention to the planar and 3-dimensional convex hull algorithms. There is a vast amount of literature on algorithms for planar convex hulls; most of these attain the lower bound. Three algorithms for the planar case have been surveyed. We start with Graham's Scan [14], which was one of the first papers specifically concerned with finding an efficient algorithm. More than a decade later, this technique continues to be a powerful tool in computational geometry. Jarvis's March, the next algorithm surveyed, is the two-dimensional version of the *gift-wrapping method*, which is a constructive method of finding convex hulls in arbitrary dimension [9]. The

familiar technique of divide-and-conquer is applicable to the convex hull problem, a variation of which is the Kirkpatrick-Seidel algorithm [16]. It uses the novel idea of reversing the order of the divide and conquer stages; in particular, this method is known as the *prune and search method.* Among all the convex hull problems in dimensions greater than 2, the three-dimensional instance is of particular importance because of its relevance to a host of applications, ranging from computer graphics to design automation, to pattern recognition and operations research [19]. We describe the three-dimensional version of the gift-wrapping method as the final algorithm in our survey.

The convex hull problem is fundamental to computational geometry; this explains, and justifies, the amount of attention that has been paid to this problem. In order to lend some credence to this claim, it is important to consider some applications of the problem. The penultimate section of this paper attempts to do this. Giving a truly representative sample of such applications is a formidable task, quite beyond the scope of this paper. Instead, we restrict our attention to the convex layers problem, the farthest pairs problem, and Voronoi diagrams.

2 Preliminaries and Lower Bounds

Given k distinct points $p_1, p_2, \ldots, p_k$ in E^d (the d-dimensional Euclidean space), the set of points

$$p = \alpha_1 p_1 + \alpha_2 p_2 + \ldots + \alpha_k p_k \quad (\alpha_j \in \mathbf{R},\ \alpha_j \geq 0,\ \alpha_1 + \alpha_2 + \ldots + \alpha_k = 1)$$

is the *convex set* generated by $p_1, p_2, \ldots, p_k$, and p is a *convex combination* of $p_1, p_2, \ldots, p_k$. Intuitively speaking, we say that a subset $D \subset E^d$ is *convex* if for every $q_1, q_2 \in D$, each point on their line segment is also in D.

Definition 2.1 Given a set S of points in E^d, the *convex hull* $\mathrm{conv}(S)$ of S is the smallest convex set containing S.

The term "convex hull" is used interchangeably to mean either the convex set or the boundary of the convex set. Following Preparata and Shamos [20], we will use the notation $\mathrm{CH}(S)$ when referring to the boundary of the convex hull $\mathrm{conv}(S)$.

The convex hull, then, is just the intersection of all convex sets in E^d containing S. It is guaranteed to exist since the intersection of an arbitrary non-empty family of convex sets is convex ([17], Theorem 3, p8). Since E^d ($\supseteq S$) is convex, such a family for S will be non-empty, and the intersection will certainly contain S. The convex hull of a finite set of points in E^d is called a *convex d-polytope* (or, briefly, a polytope).

Definition 2.2 A *polyhedral set* in E^d is the intersection of a finite number of closed half-spaces.

A polyhedral set may be unbounded. The relationship between convex hulls and polyhedral sets is provided by the following theorem (stated without proof):

Theorem 2.3 ([17], pp. 43-47) *Every convex polytope is a* bounded *polyhedral set. Conversely, every bounded polyhedral set is a convex polytope.*

The description of the convex polytope is obtained from its boundary, which consists of *faces*. We use the term *k-face* to refer to a k-dimensional face. If a polytope P is d-dimensional, then its $d-1$-faces are called *facets*, its $d-2$-faces are called *subfacets*, its 1-faces are called *edges*, and its 0-faces are called *vertices*. For technical reasons and for uniformity, the given d-polytope P is sometimes referred to as the d-face, and the empty set is called the (-1)-face. It can be shown ([17], p40) that a convex polytope has only a finite number of distinct faces, and each face is a convex polytope.

Specifically, for the case of $d = 2$, the polytope is a convex polygon. Its facets are the edges, and its subfacets are the vertices. The polygon can be represented as a bidirectional list consisting of the ordered sequence of its vertices. For the case $d = 3$, the polytope is a polyhedron, its facets are planar polygons, and its subfacets are the edges. The number of vertices (v), edges (e) , and faces (f) of a polyhedron are linearly related, as given by Euler's formula $v - e + f = 2$. Thus a polyhedron with N vertices can be represented in $O(N)$ space ([20], p93). Thus, representation issues for polytopes in the important cases of $d = 2, 3$ do not pose any difficulties.

In what follows, we shall first define our problem formally, and then establish some lower bounds on its computational complexity.

2.1 Problem Statement and Establishing Lower Bounds

A simple observation about convex hulls will assist us in formalizing the statement of the problem.

Definition 2.4 A point p of a convex set T is called an *extreme point* if there do not exist points $a, b \in T$ such that p lies on the open line segment $\overline{ab}$.

If P is a convex polytope, i.e. $P = \text{conv}(S)$ for some finite S, and K is the set of vertices of P, then $K \subseteq S$, and K is the smallest subset such that $P = \text{conv}(K)$ ([17], p41). Moreover, the set K is precisely the set of extreme points of P.

We now state two versions of the convex hull problem, as given in [20].

- Problem EXTREME POINTS (EP): Given a set S of N points in E^d identify those points of S that are the vertices of conv(S).

However, in order to obtain a complete description of the convex hull, we will need to know how the extreme points are connected to each other i.e. we will need to obtain the description of the faces of the convex polytope.

- Problem CONVEX HULL (CH): Given a set S of N points in E^d, construct the complete description of the boundary CH(S).

It is clear that the first problem is linear time reducible to the second.

We now proceed to establish lower bounds for these two problems. We obtain lower bound results for $d = 2$, since any set of points in two dimensions is trivially embedded in E^d for arbitrary d, and hence any lower bound results obtained for $d = 2$ remain valid for higher dimensions[1].

First we consider problem CH. In the planar instance of this problem, we need to know the *order* in which the vertices of the convex polygon occur, so that we may get a complete description of its boundary. This suggests an immediate parallel between the problem of sorting and problem CH for $d = 2$.

Theorem 2.5 ([20], Theorem 3.2., p94) *Sorting is linear-time transformable to the convex hull problem; therefore, finding the ordered convex hull of N points in the plane requires $\Omega(N \log N)$ time*[2].

Proof: Given a set of n real numbers $x_1, x_2, \ldots, x_n$, we want to sort them using an ordered planar convex hull algorithm. Assume, without loss of generality, that the convex hull algorithm gives the vertices of the convex polygon in counter-clockwise order, starting at some given vertex. Now, we provide the CH algorithm with the input points $\{(x_i, x_i^2) : 1 \leq i \leq n\}$, and find the point p with the least x-coordinate (which we use as the starting point). This can be done in linear time. Observe that all points in the input to CH lie on the parabola $y = x^2$, and all of them actually lie on the convex hull (see Figure 1). So when the CH algorithm outputs the vertices of the hull in counter-clockwise order starting at p, the points will effectively be sorted by increasing abscissa. Reading the x coordinates of these points in order will give us the sorted list we need. □

Since the preceding transformation of sorting to planar CH involves only arithmetic operations, the lower bound of $\Omega(N \log N)$ for problem CH holds in all computational models in which multiplication is permitted and sorting is known to require $\Omega(N \log N)$ time ([20], p95).

Let us now consider problem EP. We do not know any linear transformation of sorting to this weaker problem, and no simple argument to establish a lower bound is known. In fact, this problem

[1] The convex hull of a set of points in one dimension is the smallest interval that contains them, and can be found in $\theta(N)$ time.

[2] Here, and in the rest of the paper, log means $\log_2$.

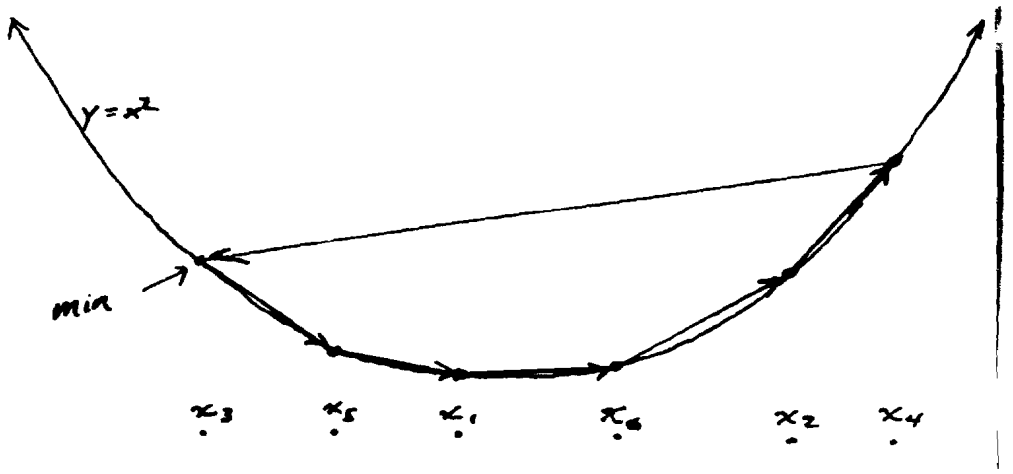


Figure 1: A linear time reduction from sorting to problem CH.

remained open for a long time, until the preliminary work of A. C. Yao [23], and, a little later, the powerful algebraic decision-tree results of Ben-Or [5] and Steele and Yao [22] definitively settled the question. As it turns out, even the unordered convex hull problem, i.e. just identifying the vertices on the planar convex hull, takes $\Omega(N \log N)$ time. In order to prove this, we will need some definitions and a few relevant results.

If we have some problem $\mathcal{A}$, and its associated decision problem is $D(\mathcal{A})$, then it is easy to recognize that $D(\mathcal{A}) \propto_{O(N)} \mathcal{A}$. This is a crucial observation, since it means that when we want to find a lower bound to a problem, we may restrict our attention to the lower bound of the corresponding decision problem. The statement of the decision problem that corresponds to problem EP is formulated as follows:

- Problem PLANAR EXTREME POINTS TEST (EP-test): Given N points in the plane, are they vertices of their convex hull?

Before going on to the results of Ben-Or and Steele-Yao that give us lower bounds to the above problem, we will need to know what algebraic decision trees are.

Definition 2.6 ([20], p30) An *Algebraic Decision Tree* on a set of variables $\{x_1, x_2, \ldots, x_n\}$ is a program with statements $L_1, L_2, \ldots, L_p$ of the form:

1. L_r: Compute $f(x_1, x_2, \ldots, x_n)$; if f : 0 then go to L_i else go to L_j (where : denotes any comparison relation such as $\leq$).
2. L_s: Halt and output YES (accepted input in decision problem).
3. L_t: Halt and output NO (rejected input in decision problem).

In definition 2.6, f is an *algebraic function* i.e. a polynomial of degree $degree(f)$. Furthermore, the program is assumed to be loop-free, so that it has the structure of a tree. Each non-leaf node in the tree is described by $f_v(x_1, \ldots, x_n) : 0$, where f_v is a polynomial in the variables x_i for $1 \leq i \leq n$, and : a comparison relation . The *order* of the algebraic decision tree $\mathcal{D}$ is the maximum of the degree of the polynomials $f_v(x_1, \ldots, x_n) : 0$, for each node v of $\mathcal{D}$.

The computation of a RAM can be represented by an algebraic decision tree $\mathcal{D}$. If the answer output by the RAM program for a given input is YES (NO), then the leaf of the decision tree that the input goes to is classified as accepting(rejecting). The depth of the tree gives a lower bound on the time $\mathcal{T}$ the computation takes. Now, suppose we know that there are H accepting leaves in the tree, and that the tree is binary[3]. Then, clearly $\mathcal{T} \geq \text{depth}(\mathcal{D}) \geq \log H$ (the $=$ case holds if all leaves are accepting leaves). Thus, if we get a handle on the number of accepting leaves in the tree, we immediately have a lower bound for T.

The motivation for the proof technique for obtaining a lower bound on $\mathcal{T}$ is as follows: If $x_1, x_2, \ldots, x_n$ are the parameters of the decision problem, then each such input to the decision problem can be viewed as a point in E^n Euclidean space. The decision problem gives a YES answer for all points in some $W \subseteq E^n$ i.e. it outputs YES if and only if $(x_1, x_2, \ldots, x_n) \in W$ (the *decision tree $\mathcal{D}$ solves the membership problem for W*). Each leaf l_j of the tree has a region $D_j \subseteq E^n$ associated with it, namely, the region that is the set of all points that satisfy the constraints $f_v : 0$ for each node v in the path from the root to the leaf l_j. Since each input $x_1, x_2, \ldots, x_n$ corresponds to a computation that traces a unique path in the decision tree and which takes it to either an accepting leaf or a rejecting one, leaf L_j is classified as:

$$\begin{cases} \text{accepting}, & \text{if } D_j \subseteq W \\ \text{rejecting}, & \text{otherwise} \end{cases}$$

Observe that for any two distinct leaves l_i, l_j, D_i and D_j must be disjoint because each input $x_1, x_2, \ldots, x_n$ traces a unique path from the root to a leaf. Now, suppose that, by the nature of the problem, we know the number of *disjoint, connected components*, $\#(W)$, of W. We then try to establish, for *accepting* leaf l_j, the maximum number of disjoint, connected components of W that D_j could possibly have i.e. we establish that $\#(D_j) \leq M$ for some M. From this, and from the previous observation, it follows that

$$\#(W) \leq MH$$

$$\text{i.e. } \log \#(W) - \log M \leq \log H \leq \text{depth}(\mathcal{D}) \leq \mathcal{T} \tag{1}$$

Thus $\log \#(W) - \log M$ is the lower bound for $\mathcal{T}$.

[3]This is a reasonable assumption , since any tree with branching > 2 can be replaced by a corresponding binary tree that performs the same computation

In the case of a linear decision tree model (i.e. an algebraic decision tree of order 1), it can be proved that M has to be 1, and, as a consequence, we have the following theorem (due to Dobkin and Lipton (1979), and stated here without proof).

Theorem 2.7 *[11] Any linear decision tree algorithm that solves the membership problem in* $W \subseteq E^n$ *must have depth at least* $\log \#(W)$, *where* $\#(W)$ *is the number of disjoint, connected components of* W.

Unfortunately, the elegant and straightforward proof technique used for the above theorem is restricted to linear decision tree models. The proof rests crucially on the fact that the region $D_j \subseteq E^n$ associated with a leaf l_j of the tree is convex, because it is the intersection of half-spaces (the polynomial f_v at node v in the tree defines a hyper-plane). However, this very useful property no longer holds when the maximum degree of the polynomial f_v is ≥ 2. When such polynomials are used, the region associated with a leaf may consist of *several* disjoint, connected components of W ([20], p33). Hence, more sophisticated concepts are needed to figure out what M should be for d-degree algebraic trees ($d \geq 2$).

This problem was solved by Steele and Yao (1982) and Ben-Or (1983), using a clever adaptation of a classical result in algebraic geometry, proved independently by Milnor (1964) and Thom (1965). The Milnor-Thom theorem is as follows:

Theorem 2.8 (Milnor-Thom) *Let* V *be the set of points in the* m*-dimensional cartesian space* E^m *defined by the simultaneous solution to the following* p *polynomial equations:*

$$\begin{aligned} g_1(x_1, \ldots, x_m) &= 0 \\ g_2(x_1, \ldots, x_m) &= 0 \\ &\vdots \\ g_p(x_1, \ldots, x_m) &= 0 \end{aligned}$$

Then, if the degree of each polynomial g_i $(i = 1, \ldots, p)$ *is* $\leq d$, *the number* $\#(V)$ *of disjoint, connected components of* V *is bounded above by*

$$\#(V) \leq d(2d - 1)^{m-1},$$

independent of the number of equations.

Theorem 2.8 gives us a way to bound the number of disjoint, connected components that can be associated with a leaf; the constraints encountered along the path from the root of the tree $\mathcal{D}$ to the leaf give us the p polynomials in the hypothesis of the theorem. However, since V above is defined in terms of equations, and a path in $\mathcal{D}$ will typically consist of equations as well as inequalities, the Milnor-Thom theorem cannot be applied directly. Ben-Or found a way to circumvent this difficulty, and proved the following result.

Theorem 2.9 ([20], Theorem 1.2., p35) *Let W be a set in the cartesian space E^n and let $\mathcal{D}$ be an algebraic decision tree of fixed order d ($d \geq 2$) that solves the membership problem in W. If h^* is the depth of $\mathcal{D}$, then $h^* = \Omega(\log \#(W) - n)$.*

Proof: Let $\mathcal{D}$ be an algebraic decision tree of order d ($d \geq 2$) that solves the membership problem in W, and $\#(W)$ the number of disjoint, connected components of W.

Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Suppose

$$\begin{aligned} &q_1(\mathbf{x}) = 0, \ldots, q_r(\mathbf{x}) = 0 \\ &p_1(\mathbf{x}) > 0, \ldots, p_s(\mathbf{x}) > 0 \\ &p_{s+1}(\mathbf{x}) \geq 0, \ldots, p_h(\mathbf{x}) \geq 0 \end{aligned}$$

are the constraints along the path from the root to some leaf in $\mathcal{D}$, where the q_i's and the p_j's are polynomials, and $d = \max\{2, \text{degree}(q_i), \text{degree}(p_j)\}$. Let U be the solution space of these constraints.

To convert the inequalities into equalities, Ben-Or introduced *slack variables*, an idea commonly used in linear programming [1]. The first step in the conversion is to replace the open inequalities with closed inequalities. Let $\#(U) \stackrel{\text{def}}{=} t$ ($\#(U)$ is finite), and pick a point from each of the t components of U. Let these points be $v_1, v_2, \ldots, v_t$. Define

$$\varepsilon = \min\{p_i(v_j) : i = 1, \ldots, s; j = 1, \ldots, t\}.$$

Since each v_i lies in the solution space U, $p_i(v_j) > 0$ (range of i and j as above) and hence $\varepsilon > 0$. Thus, the solution space U_ε of the following set of equations

$$\begin{aligned} &q_1(\mathbf{x}) = 0, \ldots, q_r(\mathbf{x}) = 0 \\ &p_1(\mathbf{x}) \geq 0, \ldots, p_s(\mathbf{x}) > 0 \\ &p_{s+1}(\mathbf{x}) \geq 0, \ldots, p_h(\mathbf{x}) \geq 0, \end{aligned}$$

is contained in U, and, clearly, $\#(U_\varepsilon) \geq \#(U)$.

The final set of (converted) constraints, then, with $y_1, \ldots, y_s, y_{s+1}, \ldots, y_h$ as the slack variables, is as follows (note that we now view E^n as a subspace of E^{n+h}):

$$\begin{aligned} &q_1(\mathbf{x}) = 0, \ldots, q_r(\mathbf{x}) = 0 \\ &p_1(\mathbf{x}) - \varepsilon - {y_1}^2 = 0, \ldots, p_s(\mathbf{x}) - \varepsilon - {y_s}^2 = 0 \\ &p_{s+1}(\mathbf{x}) - {y_{s+1}}^2 = 0, \ldots, p_h(\mathbf{x}) - {y_h}^2 = 0 \end{aligned}$$

The Milnor-Thom theorem can be applied to the above set of equations, since they are all equalities (in E^{n+h}). The order of the polynomials is still d (since we assumed $d \geq 2$). Let U_{n+h}

be the solution set of the above set of equations in E^{n+h}. By the Milnor-Thom theorem, we have

$$\#(U_{n+h}) \leq d(2d - 1)^{n+h-1}.$$

Since U_ε is the projection of U_{n+h} into E^n, we must have $\#(U_\varepsilon) \leq \#(U_{n+h})$[4]. Therefore, we can conclude that

$$\#(U) \leq \#(U_\varepsilon) \leq \#(U_{n+h}) \leq d(2d - 1)^{n+h-1}.$$

Since the set of equations we started off with were the equations along a path from a root to a leaf, h (the number of inequalities) can be at most as large as the path length. If h^* is the length of the longest path in the tree $\mathcal{D}$ (i.e. the depth of $\mathcal{D}$), then $\mathcal{D}$ will have at most 2^{h^*} leaves, and each leaf will have at most $d(2d - 1)^{n+h^*-1}$ disjoint, connected components associated with it. Referring now to Equation 1, we have

$$\begin{aligned} M &= d(2d - 1)^{n+h^*-1} \\ \Rightarrow M &< (2d)^{n+h^*} \\ \Rightarrow \log M &< (n + h^*) + (n + h^*)\log d \\ \Rightarrow \log M &= O(n + h^*) \end{aligned}$$

This means that $\log \#(W) - O(n + h^*) \leq h^*$ (again, from Equation 1), and since d is a constant, we have

$$h^* = \Omega(\log \#(W) - n),$$

and this concludes the proof of the theorem. □

Let us now see why the above results pertaining to quadratic and higher degree decision-tree models are relevant to the problem we are concerned with, namely that of establishing a lower bound for problem EP-test. For a large number of geometric problems, the linear decision-tree model is much too restrictive. For instance, just computing the Euclidean distance metric requires quadratic polynomials. The linear model is inadequate for our problem also; we do not know of any planar convex hull algorithm that uses only linear tests, and, in fact, all known algorithms can be correctly modeled using quadratic decision trees. The primitive operation in these algorithms is of the following form: Given three points p_1, p_2, and p_3 in the plane, does p_1 lie to the *left*, to the *right*, or *on* the directed line segment joining p_2 and p_3? Mathematically, this test is expressed as $\Delta(p_1, p_2, p_3) : 0$, where Δ corresponds to the polynomial given by the determinant

$$\Delta(p_1, p_2, p_3) = \begin{vmatrix} p_{11} & p_{12} & 1 \\ p_{21} & p_{22} & 1 \\ p_{31} & p_{32} & 1 \end{vmatrix} \tag{2}$$

[4] If two points in E^{n+h} are connected by some path, then the projections of these two points in E^n are connected by the projection of that path. Thus, projection can only reduce the number of disjoint, connected components [1].

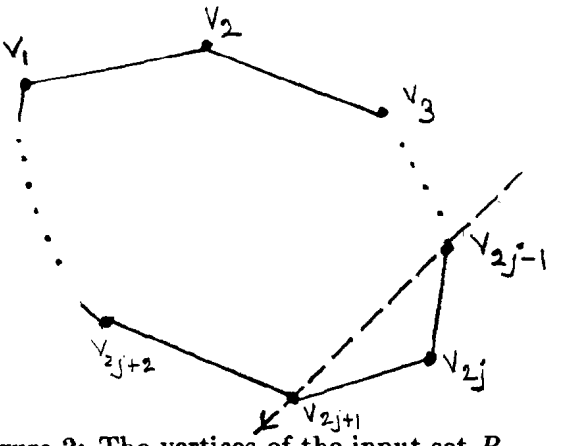


Figure 2: The vertices of the input set R.

where $p_1 = (p_{11}, p_{12})$, $p_2 = (p_{21}, p_{22})$, and $p_3 = (p_{31}, p_{32})$. Δ gives us twice the signed area of Triangle$(p_1p_2p_3)$[5]. It is clearly a quadratic polynomial.

We finally come to the last step of establishing a lower bound for problem EP-test. Since we know that the quadratic decision-tree model is the correct one for this problem, we can use Theorem 2.9 to give us our final result. However, in order to use this theorem, we first have to obtain a lower bound to $\#(W)$, and we do that as follows [1].

Let $R = \{v_1, v_2, \ldots, v_{2N}\}$ be the input set of points to the problem EP-test. Let $v_i = (x_i, y_i)$, $1 \leq i \leq 2N$. Each such set R can be regarded as a point $\mathbf{z} = (x_1, y_1, x_2, y_2, \ldots, x_{2N}, y_{2N})$ in E^{4N}. W in this case, then, is the set of all points in E^{4N} that satisfy the problem EP-test. Pick a set R for which EP-test gives the answer YES. Without loss of generality, let $v_1, v_2, \ldots, v_{2N}$ be the clockwise order of vertices on the hull, as shown in Figure 2. Let $\pi_1, \pi_2, \ldots, \pi_{N!}$ be the $N!$ permutations of the integers 1, 2, ..., N and let all possible permutations of the even numbered vertices $v_2, v_4, \ldots, v_{2N}$ be as given below:

$$\Pi_1 = (v_{2\pi_1(1)}, v_{2\pi_1(2)}, \ldots, v_{2\pi_1(N)})$$
$$\vdots$$
$$\Pi_{N!} = (v_{2\pi_{N!}(1)}, v_{2\pi_{N!}(2)}, \ldots, v_{2\pi_{N!}(N)})$$

Observe that EP-test gives a YES answer for each set

$$R_i = \{v_1, \Pi_i(1), v_3, \Pi_i(2), \ldots, v_{2N-1}, \Pi_i(N)\}, \; 1 \leq i \leq N!$$

since it is just a permutation of the input set R. Hence, for each such i, the point $\mathbf{z}_i$ that the set R_i corresponds to, will be in W.

[5]The *signed area* of the triangle $(p_1p_2p_3)$, as given by this determinant, is positive (negative) if and only if p_1, p_2, and p_3 form a counterclockwise (clockwise) cycle. This means that Δ is positive (negative) if and only if p_1 lies to the left (right) of the directed line segment from p_2 and p_3. Δ is zero when p_1 lies *on* the directed line segment from p_2 and p_3.

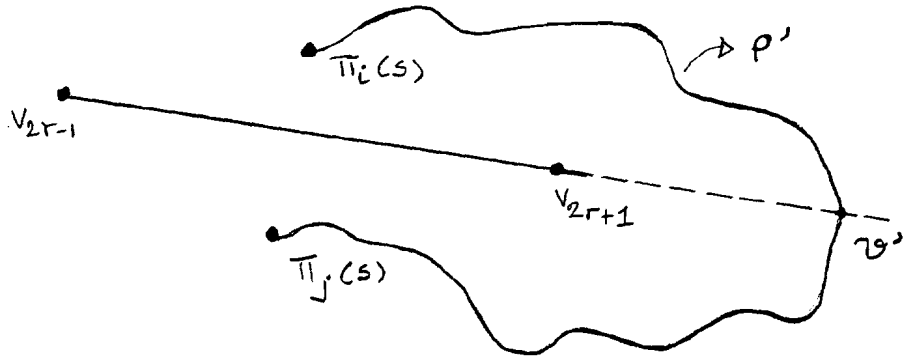


Figure 3: Projection ρ' of the path connecting $\mathbf{z}_i$ and $\mathbf{z}_j$.

Claim: Each $\mathbf{z}_i$, $1 \leq i \leq N!$ belongs to a distinct connected component of W.

Proof: Note that for any pair of sequential odd numbered vertices v_{2j-1} and v_{2j+1}, there is exactly one vertex from R, namely v_{2j}, that lies to the left of the directed line segment $\overline{v_{2j-1}v_{2j+1}}$ ($1 \leq j \leq N$, $v_{2N+1} = v_1$) (see Figure 2), and hence $\Delta(v_{2j-1}v_{2j+1}v_{2j}) > 0$ (where Δ is as in Equation 2). Now, for each Π_i, construct a two dimensional $N \times N$ array A_i as follows. For $1 \leq r \leq N$ and $1 \leq s \leq N$, associate with the r-th row of A_i the pair (v_{2r-1}, v_{2r+1}), and with the s-th column the vertex $\Pi_i(s)$. Define

$$A_i(r, s) = \begin{cases} -, & \text{if } \Delta(v_{2r-1}v_{2r+1}\Pi_i(s)) < 0 \\ +, & \text{otherwise} \end{cases}$$

Each row of the above array will contain exactly one $+$, by virtue of the observation made in the first sentence of this proof; $A_i(r, s') = +$ iff $\Pi_i(s') = v_{2r}$. Hence, if $i \neq j$, then $A_i \neq A_j$. The two arrays must differ in at least one location, otherwise they would correspond to the same permutation. Suppose A_i and A_j differ in location (r, s); for example, $A_i(r, s) = +$ and $A_j(r, s) = -$. Then, $\Pi_i(s)$ and $\Pi_j(s)$ must lie on opposite sides of the directed line segment from v_{2r-1} to v_{2r+1} (see figure 3).

Now, clearly $\mathbf{z}_i$ and $\mathbf{z}_j$ are two distinct points in E^{4N}. We want to show that $\mathbf{z}_i$ and $\mathbf{z}_j$ must lie on different connected components of W. Suppose not i.e. suppose that there is a path ρ in E^{4N} connecting $\mathbf{z}_i$ and $\mathbf{z}_j$ that lies entirely within W. Consider the projection that maps onto the plane the $2s$-th and $(2s+1)$-th coordinates of each point on the path ρ. Clearly, $\mathbf{z}_i$ gets mapped to $\Pi_i(s)$ and $\mathbf{z}_j$ gets mapped to $\Pi_j(s)$. Since $\mathbf{z}_i$ and $\mathbf{z}_j$ are connected in E^{4N}, the path ρ corresponds to a path ρ' in the plane that connects $\Pi_i(s)$ and $\Pi_j(s)$. As observed earlier, since $\Pi_i(s)$ and $\Pi_j(s)$ lie on opposite sides of $\overline{v_{2r-1}v_{2r+1}}$, by the intermediate value theorem, $\overline{v_{2r-1}v_{2r+1}}$ must intersect ρ' at some point, say v' (see Figure 3). Since v' is the projection of some point in W, it must be one of the vertices of the input set R. This means that, as illustrated in figure 4, one of v_{2r-1}, v_{2r+1}, or v'

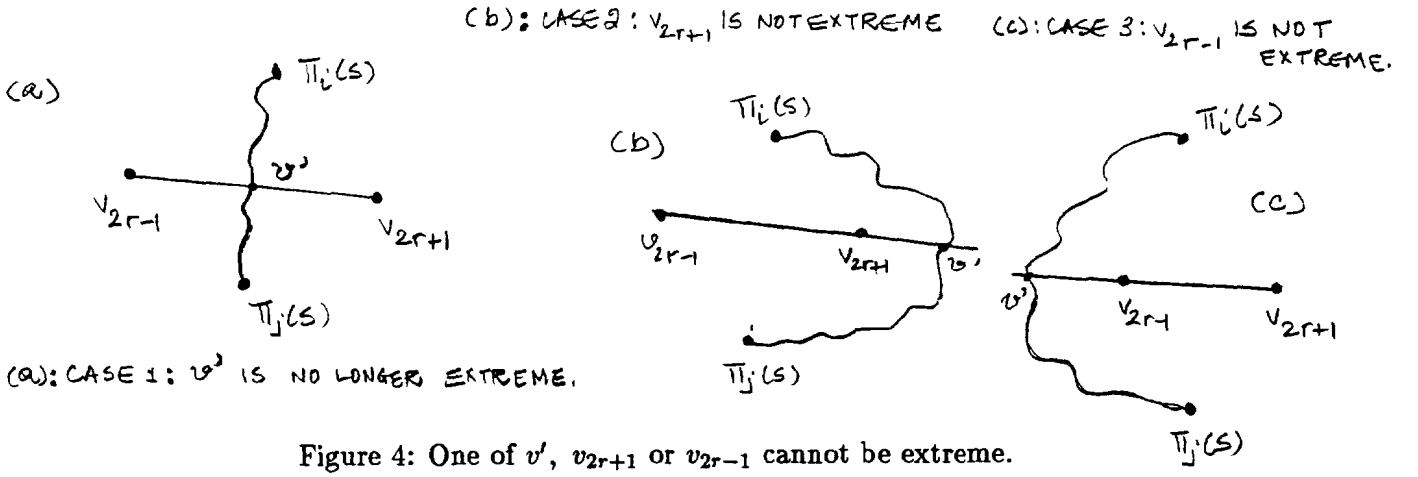


Figure 4: One of v', v_{2r+1} or v_{2r-1} cannot be extreme.

cannot be extreme. But obviously this is a contradiction, since our input set R is a YES instance of problem EP-test. Hence, *any* path from $\mathbf{z}_i$ to $\mathbf{z}_j$ must go out of W, which means that they must belong to two distinct, disjoint, connected components of W. □

Since we have $N!$ such $\mathbf{z}_i$'s, the above claim implies that W must have at least $N!$ disjoint, connected components, one for each permutation. Combining this result with Theorem 2.9, we get the following result (at last!).

Theorem 2.10 ([20], Theorem 3.3., p97) *In the fixed-order algebraic decision-tree model, the determination of the extreme points of a set of N points in the plane requires $\Omega(N \log N)$ time.*

Proof: Referring to Theorem 2.9, we have $n = 4N$ and $\#(W) \geq N!$. Therefore, by the result of the theorem, the computation time for the EP-test problem is $\Omega(\log \#(W) - 4N) = \Omega(\log N! - 4N) = \Omega(N \log N - 4N) = \Omega(N \log N)$. □

This concludes our section on lower bounds for the time complexity of the problem. Even though the proof for the lower bound to problem EP is much more elaborate and complex than that for problem CH, both problems bear an intimate relation to sorting, and this is apparent in many of the algorithms for the problem at hand. We now turn to the description of algorithms for the convex hull problem.

3 Algorithms for the Convex Hull Problem

We start with algorithms for the planar convex hull problem.

First we demonstrate a trivial (and extremely inefficient) algorithm to show that this problem is in P. The algorithm can be derived from the following theorem:

Theorem 3.1 ([20], Theorem 3.4, p98) *A point p fails to be an extreme point of a plane convex set S only if it lies in some triangle whose vertices are in S but is not itself a vertex of the triangle.*

Note that the triangles might deteriorate into three collinear points. Using the above theorem, we can eliminate all points that are not extreme from the input set of points, as follows: For a given point, there are $\begin{pmatrix} N-1 \\ 3 \end{pmatrix}$ triangles determined by the remaining $N-1$ points. For each of these triangles, determine if the point lies in the triangle or not, which can be done in a constant number of steps. So, in $O(N^3)$ time we can learn if a given point is extreme or not. This process is repeated for each of the N points, and hence we have an $O(N^4)$ algorithm to identify the extreme points of a given set of points. These points must be ordered for us to form the convex hull. Since a ray emanating from an interior point of a convex polygon intersects it in exactly one point ([20], Theorem 3.5, p99), and vertices of a convex polygon occur in sorted angular order about any interior point ([20], Theorem 3.6, p99), we can sort the extreme points by picking some point that is internal to the hull (the centroid of the extreme points, say, which can be found in $O(N)$ time). A ray starting at this point sweeps over all the extreme points, and the angle subtended by the ray with, say, the horizontal can be determined at each extreme point, which clearly takes constant time. The angles thus determined can then be sorted, and hence we have an $O(N \log N)$ procedure for this phase of our algorithm.

Aside from the theoretical significance of establishing that problem CH lies in P, an $O(N^4)$ algorithm does not buy us much in terms of computational feasibility. It is far from the $O(N \log N)$ lower bound, but, as we shall see now, there are a number of algorithms that actually match this lower bound.

3.1 Graham's Scan

One reason the algorithm described above is $O(N^4)$ may be that it is doing redundant computation. Graham's algorithm [14] amply illustrates that this is indeed the case. Performing the sorting step first enables the extreme points to be found in linear time.

(see diagram for Graham's Scan in notes).

Suppose $S = \{p_1, p_2, \ldots, p_N\}$ is the input set of points in the plane. Let O be the point from the input set that has the least y-coordinate (if there are two or more such points, pick the one with the largest x-coordinate). Assume that the coordinates of the n points have been transformed so that O is the new origin. We now sort these points with respect to polar angle from O about the horizontal axis. If two points have the same polar angle, the point closer to O is considered to be the smaller of the two points. Before we proceed to the rest of algorithm, a brief but important digression is necessary.

Suppose that we are given the input points in cartesian coordinates $(x,\ y)$. In order to do comparisons of the above form, we will need the polar coordinate information $(r,\ \theta)$, where $r = \sqrt{x^2 + y^2}$ and $\theta = \arcsin(y/r)$. In many systems with a restrictive set of primitives, however, both sqrt and arcsin are not unit-time operations. Since we will only be comparing distances from O when the polar angles for two points are the same, we do not have to actually compute the square root. We can work with the distance *r squared*, since all we are interested in is a magnitude comparison. We do not have to explicitly compute the polar angles either; the signed area of a triangle, defined earlier, will help us here. Given two points p_i and p_j from the input set of points, the polar angle subtended by $\overline{Op_i}$ with the horizontal is strictly smaller than that subtended by $\overline{Op_j}$ if and only if the signed area of the triangle $(O,\ p_i,\ p_j)$ is strictly positive.

So suppose now that the points have been sorted in the method described above, and arranged in a doubly linked circular list[6]. The basic idea of Graham's algorithm is to make a single pass around the sorted list of points, and eliminate all points that are not extreme from the circular list. The important to note is that if a point is not extreme, then it will be internal to some triangle $(O,\ p_i,\ p_j)$, where p_i and p_j are consecutive hull vertices (a point on the triangle boundary, excluding the end points, is also considered internal). Thus, what we are left with in the list are the hull vertices in the sorted order. Let $v_1,\ v_2,\ \ldots,\ v_N$ be the sorted order of the input set of points $p_1,\ p_2,\ \ldots,\ p_N$ (where v_1 obviously is O).

The actual method of elimination of points that are not extreme is as follows: We start the scan through the list at some point, say the origin O (note that O will be one of the extreme points, and hence a hull vertex). We then repeatedly consider three consecutive points $v_i,\ v_{i+1}$, and v_{i+2} $(1 \leq i \leq n-2)$ in the sorted list to determine whether or not internal $\angle v_i v_{i+1} v_{i+2}$ is a reflex angle (one that is $\leq \pi$. If internal $\angle v_i v_{i+1} v_{i+2}$ is reflex then $v_i v_{i+1} v_{i+2}$ is said to be a *right turn*, otherwise it is said to be a *left turn*. This can be determined by applying Equation 2; if $\Delta(v_i,\ v_{i+1},\ v_{i+2})$ is ≤ 0, then $v_i v_{i+1} v_{i+2}$ is a right turn; otherwise, it is a left turn. Note that if $v_i v_{i+1} v_{i+2}$ forms a right turn, then we can immediately eliminate v_{i+1} from our list, because v_{i+1} is internal to the triangle $(O,\ v_i,\ v_{i+2})$. Thus, our scan works as follows:

1. $v_i v_{i+1} v_{i+2}$ forms a right turn: Eliminate v_{i+1}, and check $v_{i-1} v_i v_{i+2}$.
2. $v_i v_{i+1} v_{i+2}$ forms a left turn: Advance the scan to the next vertex in the list, v_{i+1}, and check $v_{i+1} v_{i+2} v_{i+3}$.

We advance the scan until we hit v_1 again in the circular list. Also, since v_1 is an extreme point, we do not have to worry about eliminating it, and hence it is all right for the initial check to be $v_1 v_2 v_3$. The complete algorithm for Graham's Scan is given below:

[6]The data structures used for this algorithm are as described by Preparata and Shamos [20]. A stack may also be used for the same algorithm [1].

procedure GRAHAMHULL(S);

1. Find the point in S with the least y-coordinate. If there are two or more such points, pick from them the one with the largest x coordinate. Call this point O.
2. Transform all points in S so that O is the new origin.
3. Sort the points of S with respect to polar angle from O. If two points have the same polar angle, compute the distance of the points from O to determine their order.
4. (Scan)

```
begin
    v := O ;
    while (RLINK[v] ≠ O) do
        if (Δ(v, RLINK[v], RLINK[RLINK[v]]) ≤ 0) then
            begin
                Delete RLINK[v];
                v := LLINK[v];
            end
        else
            v := RLINK[v];
end
```

Let us determine the complexity of the algorithm. Note that only unit time arithmetic and comparison operations are used, since, as mentioned earlier, we do not have to explicitly compute the sqrt and arcsin functions. Step (1) and Step (2) clearly take $O(N)$ time (where N is the size of the input set of points). Step (3), which is the sorting step, takes $O(N \log N)$ time. Step (4) takes $O(N)$ time also, and this can be shown as follows. Computing *Delta* and doing the comparison takes a constant number of steps. After the comparison has been done, we either delete a point, or advance the scan. We can perform at most $N-1$ deletions ($N-1$ will be necessary in case all the input points are the same) and advance the scan at most $N-1$ times ($N-1$ will be necessary in case all the input points lie on the hull). Clearly, then, the last step takes $O(N)$ time. Thus, the entire algorithm takes $O(N \log N)$ time (and $O(N)$ space). At the end of the algorithm, all the points remaining in the circular list are the hull vertices, and, starting from O, we can output the points on the hull in counter-clockwise order as we traverse the list.

This algorithm clearly demonstrates the direct relation between sorting and the convex hull problem. However, Graham's scan has its own drawbacks. It rests crucially on the fact that the hull vertices occur in sorted angular order about any interior point, and this fact holds only in two dimensions. Hence this algorithm does not generalize to higher dimensions. Another important

goal for convex hull algorithms is parallelizability, and since this algorithm is not recursive (and hence does not divide the main problem into smaller subproblems), it cannot be used in a parallel environment.

3.2 Jarvis's March

Another approach to finding the convex hull is to identify the hull edges, as opposed to the hull vertices. This is exactly what Jarvis's algorithm [15] does. As it turns out, Jarvis's method is the two-dimensional version of a general algorithm, known as the "gift-wrapping" method, for finding the convex hull of a set of points in arbitrary dimension. This general method was proposed by Chand and Kapur [9] as early as 1970. The three-dimensional specialization of this algorithm will be mentioned later on in this paper.

The name Jarvis's March comes from the fact that we march around the convex hull, finding successive hull vertices in order[7]. We will now describe a version of Jarvis's algorithm that includes some minor corrections [4]. Let p_1 and q_1 be the points from the input set of points with the least and the largest y-coordinate, respectively (as usual, we break ties in each case by choosing the point with the largest x-coordinate). Clearly, both p_1 and q_1 are vertices on the hull. The idea is to start at p_1, and pick the next point p_2 on the hull. Clearly, p_2 is that point p from the input set such that $\overline{p_1 p}$ makes the least polar angle with the *positive* x-axis. After p_2 has been picked, we choose the next point on the hull p_3 in the same manner, and so on until we hit q_1. Now we start at q_1 and repeat the process until we hit p_1, only the polar angle is now calculated with respect to the *negative* x-axis. As mentioned earlier, the smallest angle can be determined using only arithmetic and comparison operations. The pseudocode for the algorithm is given below:

procedure JARVISMARCH(S);

1. Find the point in S with the least y-coordinate. If there are two or more such points, pick from them the one with the largest x-coordinate. Call this point p_1.

2. Find the point in S with the largest y-coordinate. If there are two or more such points, pick from the one with the largest x-coordinate. Call this point q_1.

3. Finding points on the hull from p_1 upto q_1 in the anti-clockwise direction:

```
n := 1 ;
While (p_n ≠ q_1)
    Begin
        n := n + 1;
```

[7]It also resembles the process of wrapping a two-dimensional package. This is the intuitive idea of the "gift-wrapping" method mentioned earlier.

Let p be the point from S such that the angle
made by $\overline{pp_{n-1}}$ with the positive
x-axis is minimum.
$p_n := p$;
End;

4. Finding points on the hull from q_1 down to p_1 in the anti-clockwise direction:

$m := 1$;
While $(q_m \neq p_1)$
Begin
$m := m + 1$;
Let q be the point from S such that the angle
made by $\overline{qq_{m-1}}$ with the negative
x-axis is minimum.
$q_m := q$;
End;

5. The points $p_1, p_2, \ldots, p_{n-1}, q_1, q_2, \ldots, q_{m-1}$ are the hull vertices. *Note that they are in anti-clockwise order, and hence we do not need to explicitly sort them.*

In the above algorithm, Steps (1) and (2) clearly take $O(N)$ time, where N is the size of the input set of points. Now, let us denote by h the number of vertices on the convex hull. We will use this measure to determine the complexity of Steps (3) and (4). The body of the while loop in both those steps will take $O(N)$ time. Since the total number of iterations of both those loops will be h, Steps (3) and (4) will take $O(Nh)$ time. Hence the total running time of the Jarvis March algorithm is $O(Nh)$. The worst case ($h = N$ i.e. all N input points lie on the convex hull) running time of this algorithm, then, is $O(N^2)$ which is worse than Graham's Scan. However, if h is known in advance to be small, Jarvis's algorithm is a very efficient one.

3.3 The Kirkpatrick-Seidel Algorithm

The idea of Divide and Conquer algorithms is to break up the problem into sub-problems (*divide*), solve each of the sub-problems recursively (*conquer*), and then combine the subsolutions to form the global solution (*merge*). Such algorithms for the convex hull problem will be of particular interest to us, because they might achieve the important goal of parallelizability. However, in order that such algorithms have good worst-case performances, it is important to divide the problem into sub-problems of nearly equal size.

Suppose we divide out input set of points S into two parts, S_1 and S_2, each of roughly the same size. Once we (recursively) find the convex hulls $\mathrm{CH}(S_1)$ and $\mathrm{CH}(S_2)$, is there an efficient method to find $\mathrm{CH}(S_1 \cup S_2)$ (i.e. the global solution)? The answer, fortunately, is yes. The following relation is of significance for this method:

$$\mathrm{CH}(S_1 \cup S_2) = \mathrm{CH}(\mathrm{CH}(S_1) \cup \mathrm{CH}(S_2)).$$

The above formula seems to suggest that the merge step of the divide and conquer method requires us to apply the convex hull algorithm all over again. However, it is not necessary to do that. We use the fact that $\mathrm{CH}(S_1)$ and $\mathrm{CH}(S_2)$ are in the form of *convex polygons* in order to efficiently find the convex hull of their union. How exactly this method works will become clear a little later in this section.

The standard divide-and-conquer algorithms for the convex hull problem work as follows:

procedure divide-and-conquer(S);

1. If $|S| \leq k_0$ (where k_0 is some small integer), then construct the convex hull of S by some other method. If not, go to step 2.
2. Partition S into two subsets S_1 and S_2 of approximately equal size.
3. Recursively find the convex hulls of S_1 and S_2.
4. Merge the two convex hulls $\mathrm{CH}(S_1)$ and $\mathrm{CH}(S_2)$ together to form $\mathrm{CH}(S)$.

The basic idea behind the merge step (Step (4)) is as follows (for now, we will not be rigorous, and will appeal to the intuition that Figure 5 provides). In order to combine the two convex hulls, we need to find two lines: one that is tangent to the top of both $\mathrm{CH}(S_1)$ and $\mathrm{CH}(S_2)$, and one that is tangent to the bottom of both. These are referred to as the *upper bridge* and the *lower bridge*, respectively. Let u_1 and l_1 be the vertices of S_1 that lie on the upper and lower bridge, respectively. Let u_2 and l_2 be the corresponding vertices for S_2. Then, as illustrated in Figure 5, all vertices in $\mathrm{CH}(S_1)$ going clockwise from u_1 to l_1 can be discarded. Similarly, all vertices in $\mathrm{CH}(S_2)$ going anti-clockwise from u_2 to l_2 can be discarded. All the remaining vertices of $\mathrm{CH}(S_1)$ and $\mathrm{CH}(S_2)$ form the vertices of $\mathrm{CH}(S)$. We will show later on that finding the upper and lower bridges can be done in $O(N)$ time.

The Kirkpatrick-Seidel algorithm [16] is based on a variation of the divide-and-conquer paradigm. The method used in this algorithm reverses the conquer and merge stages. Upon dividing the problem, *first* we determine how the sub-solutions will merge (without actually computing the sub-solutions), and then proceed to solve the sub-problems. The advantage of this method is that it allows us to remove beforehand, parts of the subproblems that upon merging turn out to be

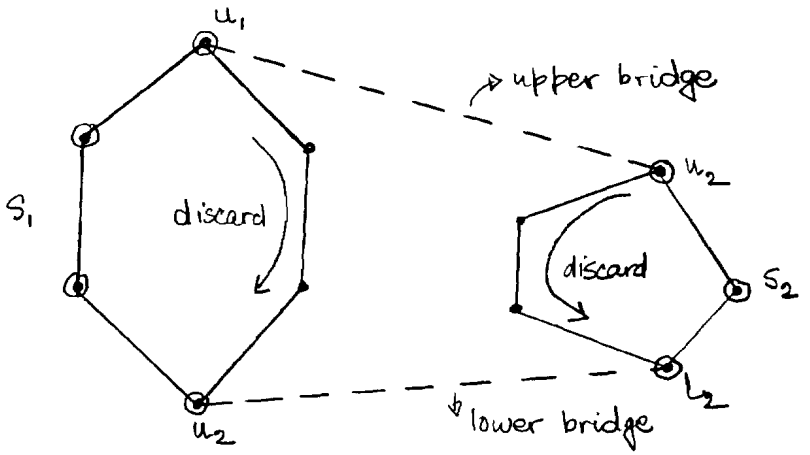


Figure 5: Merging two convex hulls by finding the upper and lower bridges.

redundant. Thus, it reduces the size of the subproblems to be solved recursively [[16], p288]. Such algorithms are also known as *prune and search* algorithms[8] [1].

The prune-and-search algorithm for the convex hull problem is as follows. Let S be the input set of (planar) points, whose members are p_1, p_2, ..., p_n.

procedure kirkpatrick-seidel(S);

1. Let p_{min} and p_{max} be the points of S with the least and largest x-coordinate, respectively
2. If there are two or more such p_{min} (p_{max}), pick from them the point with the largest y-coordinate and call it p_{umin} (p_{umax}).
   $T := \{p_{umin}, p_{umax}\} \cup \{p \in S | x(p_{umin}) < x(p) < x(p_{umax})\}$
   UPPER-HULL(p_{umin}, p_{umax}, T)[9];
3. If there are two or more such p_{min} or p_{max}, pick from them the point with the least y-coordinate as p_{lmin} or p_{lmax}, as the case may be.
   $T := \{p_{lmin}, p_{lmax}\} \cup \{p \in S | x(p_{lmin}) < x(p) < x(p_{lmax})\}$
   LOWER-HULL(p_{lmax}, p_{lmin}, T);
4. (UPPER-HULL (LOWER-HULL) will return the points on the upper (lower) hull of S in clockwise order, in the form of a doubly linked list.) Concatenate the list UPPER-HULL returns to the list that LOWER-HULL returns. We now have the convex hull of S in clockwise order.

procedure UPPER-HULL(p_{min}, p_{max}, T);

[8] In their paper, Kirkpatrick and Seidel refer to this approach as the *marriage-before-conquest* principle. I prefer *prune and search*.

[9] The *upper hull* (*lower hull*) refers to the convex hull of the points of S above (below) and on the line passing through p_{min} and p_{max}. Note that this is exactly the convex hull of S that lies above (below) that same line.

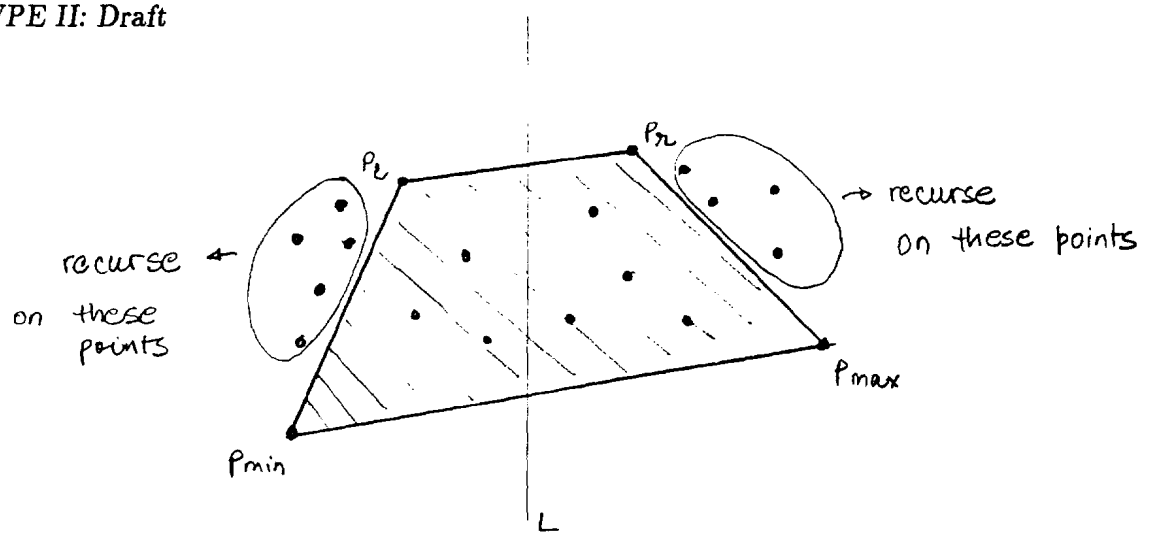


Figure 6: All points in the shaded region can be discarded before the recursive call of UPPER-HULL.

1. If $p_{min} = p_{max}$, then return the singleton (doubly linked) list containing p_{min}.
   Else, continue on with the next step.

2. Find the median of the x-coordinates of the points in T i.e. find an a such that

   $$x(p) \leq a \text{ for } |T|/2 \text{ points in } T \text{ and}$$

   $$x(p) \geq a \text{ for } |T|/2 \text{ points in } T$$

   Let L be the vertical line $x = a$. Let T_{left} be the points of T to the left of L. Let T_{right} be the points of T to the right of L.

3. Define the *upper bridge* to be the segment $\overline{p_l p_r}$ such that its left end-point p_l lies in T_{left} and its right end-point p_r lies in T_{right}, and it is a supporting line of T[10]. Clearly, the upper bridge is a part of the upper convex hull. We will show later that there is an algorithm to determine upper bridges efficiently.
   (p_l, p_r) := UPPER-BRIDGE(T, L);

4. Since p_{min}, p_l, p_r, and p_{max} are all points in T, and they form a convex quadrilateral, we can discard all points of T that fall within this quadrilateral (see Figure 6), since they cannot form the vertices of the upper hull[11].
   T_{left} := $\{p_l\} \cup$ all the points of T_{left} to the left of the line through p_{min} and p_l;
   T_{right} := $\{p_r\} \cup$ all the points of T_{right} to the right of the line through p_r and p_{max};

[10] A *supporting line* of a set S contains at least one point of S, and *all* points of S lie on one side of the line.

[11] This step is a little different in the paper by Kirkpatrick and Seidel [16]. They only discard the points that lie in the trapezoid determined by p_l, p_r and the $\overline{p_{min}p_{max}}$. They do, however, mention the approach taken here.

5. UPPER-HULL(p_{min}, p_l, T_{left}) * UPPER-HULL(p_r, p_{max}, T_{right}), where * represents list concatenation of doubly linked lists.

The procedure for the lower hull, LOWER-HULL(p_{max}, p_{min}, T), is very similar and uses exactly the same idea. We will not repeat the entire procedure for LOWER-HULL here. The only differences to note are that in Step (3) we find lower bridges instead of upper bridges. When assigning new values to T_{left} and T_{right} in Step (4), we only pick the points *below* the line in question. Finally, step (5) will be LOWER-HULL(p_{max}, p_r, T_{right}) * LOWER-HULL(p_l, p_{min}, T_{left}), so that we can get the points on the lower hull in clockwise order.

We have not yet specified how the upper and lower bridges are found. For now, let us assume that these bridges can be found in linear time. This will be proved a little later. With the given assumption, we can now show that the Kirkpatrick-Seidel algorithm runs in $O(N \log H)$ time, where N is the size of the input set of points, and H is the number of points on the hull.

Claim: UPPER-HULL takes $O(N \log H_u)$ time, where H_u is the number of vertices on the upper hull, and N is as above.

Proof: ([16], p290) From a result by Blum et al. ([3], p99) the median of a set of numbers can be found in linear time. Using this result, and our assumption stated above, it is clear that steps (2) to (4) take $O(N)$ time. Let f be the following recurrence relation (for $h \geq 2$):

$$f(n,h) \leq \begin{cases} cn & \text{if } h = 2 \\ cn + \max\{f(\frac{n}{2}, h_l) + f(\frac{n}{2}, h_r) \mid h_l + h_r = h\} & \text{otherwise} \end{cases}$$

where c is some positive constant and $n \geq h > 1$. Clearly, the run time of UPPER-HULL is given by $f(N, H_u)$, where h_l and h_r above will be the size of T_{left} and T_{right}, respectively, in step (5) in the algorithm. The claim is that $f(n,h) = O(n \log h)$ i.e. that $f(n,h) \leq cn \log h$ satisfies the above recurrence. Induction on h gives a straightforward proof. For the base case, this is obviously true. For $h > 2$, using the inductive hypothesis, we have

$$f(n,h) \leq cn + \max\left\{c\frac{n}{2}\log h_l + c\frac{n}{2} - log h_r \mid h_l + h_r = h\right\}$$

$$= cn + \frac{1}{2}cn\max\{\log(h_l h_r) \mid h_l + h_r = h\}$$

It can be easily established, using elementary calculus, that the maximum is realized when $h_l = h_r = h/2$. Thus,

$$f(n,h) \leq cn + \frac{1}{2}cn\log\left(\frac{h}{2}\right)^2$$

$$= cn + cn\log\left(\frac{h}{2}\right)$$

$$= cn + cn\log h - cn$$

$$= cn \log h$$

This concludes our proof. UPPER-HULL is $O(N \log H_u)$, and it obviously has a linear space bound. □

Similarly, LOWER-HULL will be $O(N \log H_l)$, where H_l is the number of points on the lower hull. The time bound for the main sub-routine (**procedure** kirkpatrick-seidel) follows immediately from the above claim. Since step (1) takes $O(N)$ time, the total time taken will be $O(N) + O(N \log H_u) + O(N \log H_l)$, which works out to be $O(N \log H)$.

We now demonstrate that finding the upper bridge takes $O(N)$ time. Recall that finding the upper bridge involves finding a supporting line that contains two points of S - one to the left of L and one to the right of L. Let b be the bridge, and m_b be the slope of b. One way in which bridge points can be identified is to successively eliminate points from S as candidates for bridge points. We pair up the points of S into $N/2$ couples, where N is the size of S. The following lemmas show how the formation of pairs can help us eliminate candidates for the upper bridge; analogous lemmas for the lower bridge are omitted here for the sake of brevity.

Lemma 3.2 *Let p, q be a pair of points of S. If $x(p) = x(q)$ and $y(p) > y(q)$ then q cannot be a bridge point.*

Lemma 3.3 *Let p, q be a pair of points of S with $x(p) < x(q)$, and let m_{pq} be the slope of the straight line h through p and q.*

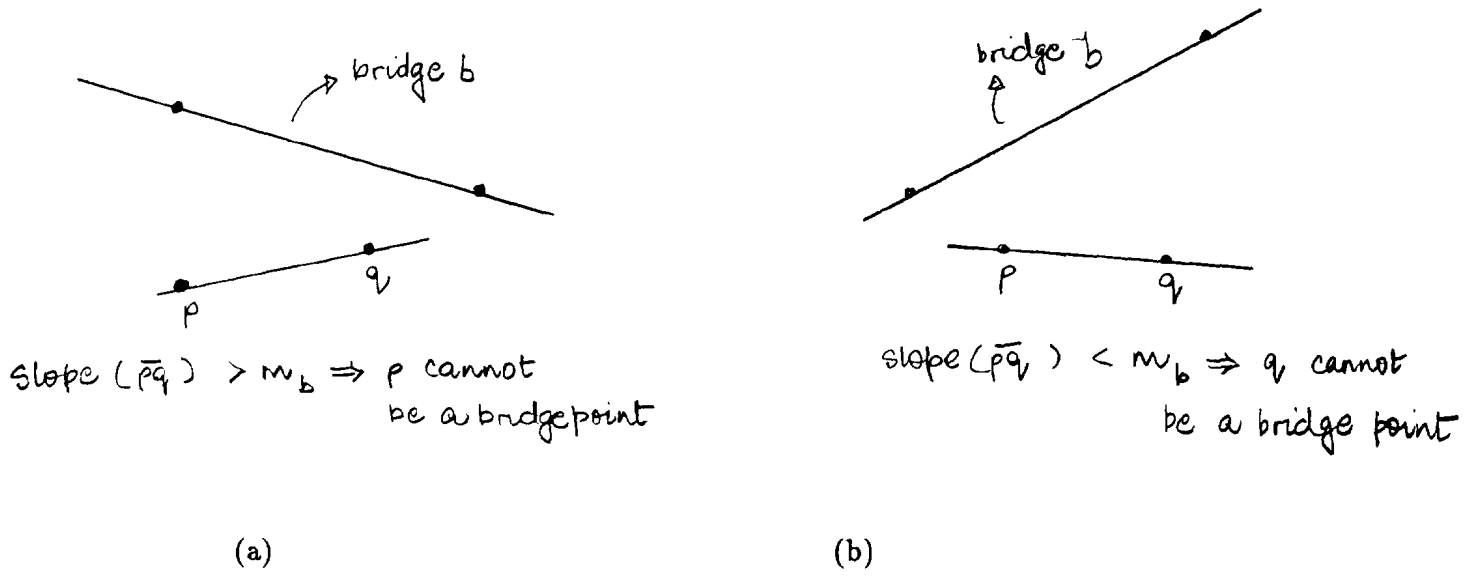
1. *If $m_{pq} > m_b$, then p cannot be a bridge point.*
2. *If $m_{pq} < m_b$, then q cannot be a bridge point.*

Instead of proving these simple lemmas formally, we refer the reader to figures 8-(a),(b) from which the validity of the lemmas is obvious. In particular, they allow us to eliminate one bridge point from each of the $N/2$ pairs by appealing to the following lemma.

Lemma 3.4 *Let h be a supporting line of S with slope m_h, and let the point(s) of S that it contains belong to the upper hull.*

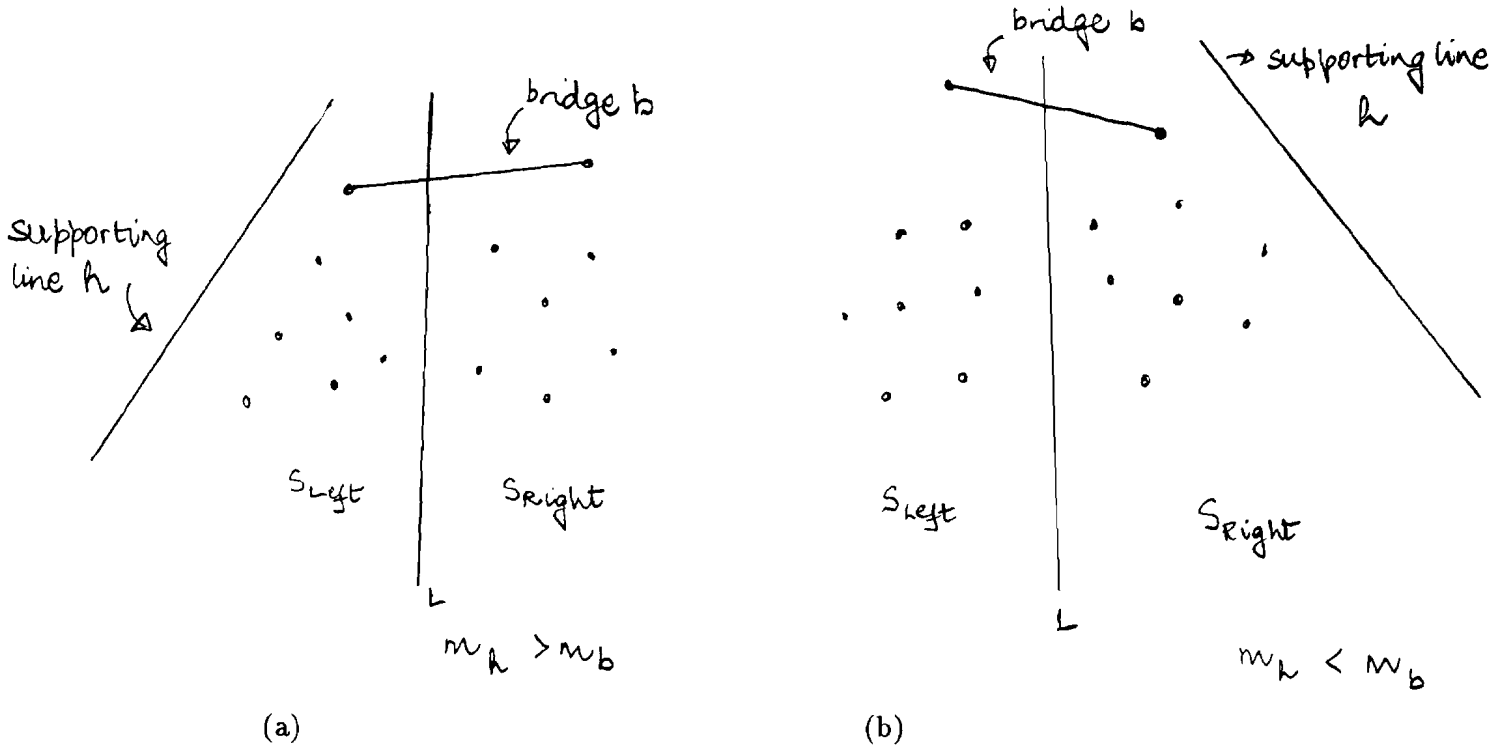
1. *$m_h > m_b$ iff h contains only points of S that are to the left of or on L.*
2. *$m_h < m_b$ iff h contains only points of S that are to the right of L.*

Figure 8 provides the necessary intuition behind the lemma. Combining it with lemmas 3.2 and 3.3, we can conclude the following.



(a) (b)

Figure 7: Elimination of candidates for bridge points.



(a) (b)

Figure 8: The upper bridge and its relation to supporting lines.

Corollary 3.5 *Let p, q be points in S with $x(p) < x(q)$, and h be any supporting line of S. Then*

- *if h contains only points of S to the left of or on L and $m_{pq} > m_h$, then p is not a candidate for the end point of the bridge, and*
- *if h contains only points of S to the right of L and $m_{pq} < m_h$, then q is not a candidate for the end point of the bridge.*

We are now ready to give the UPPER-BRIDGE algorithm. Let L be the vertical line $x = a$, and S the input set of points. If two edges of the convex hull intersect L and therefore qualify as upper bridges, i.e. L contains a vertex v of the upper hull, then this algorithm returns the bridge for which v is the left end-point.

function UPPER-BRIDGE(S, L); ([16], p292)

1. CANDIDATES := $\emptyset$;
2. **If** $|S| = 2$ **then** return (i, j), where $S = \{p_i, p_j\}$ and $x(p_i) < x(p_j)$.
3. Arbitrarily pair up points of S into pairs (p_i, p_j). There will be $\lfloor |S|/2 \rfloor$ such pairs.
   If a point of S remains, then insert it into CANDIDATES.
   PAIRS := the set of these ordered pairs (p_i, p_j) such that $x(p_i) \leq x(p_j)$.
4. *Determine the slopes of the straight lines defined by the pairs. If the slope does not exist, apply Lemma 3.2:*

   For all (p_i, p_j) in PAIRS **do**
   If $x(p_i) = x(p_j)$ **then**
   Begin
   If $y(p_i) > y(p_j)$ **then**
   insert p_i into CANDIDATES
   else
   insert p_j into CANDIDATES
   End
   else
   $k(p_i, p_j) := \frac{y(p_i) - y(p_j)}{x(p_i) - x(p_j)}$
5. Determine K, the median of $\{k(p_i, p_j) | (p_i, p_j) \in PAIRS\}$.
6. SMALL := $\{(p_i, p_j) \in \text{PAIRS} | k(p_i, p_j) < K\}$.
   EQUAL := $\{(p_i, p_j) \in \text{PAIRS} | k(p_i, p_j) = K\}$.

LARGE := $\{(p_i,\ p_j) \in \text{PAIRS} | k(p_i,\ p_j) > K\}$.

7. *Find a supporting line of S with slope K. Find the points of S that lie on this supporting line:*
   To do this we draw a line with slope K through each point of S. The line that has the highest intersection with the y-axis is the line we are looking for.
   MAX:= set of points $p_i \in S$ such that $y(p_i) - K * x(p_i)$ is maximum.
   p_k := point in MAX with minimum x-coordinate.
   p_m := point in MAX with maximum x-coordinate.

8. *Determine if h contains the bridge:*
   If $x(p_k) \leq a$ and $x(p_m) > a$ **then** return $(k,\ m)$.

9. *h contains only points of S to the left of or on L (Use Corollary 3.5):*
   If $x(p_m) \leq a$ **then**
   for all $(p_i,\ p_j) \in$ LARGE $\cup$ EQUAL insert p_j into CANDIDATES.
   for all $(p_i,\ p_j) \in$ SMALL insert p_i and p_j into CANDIDATES.

10. *h contains only points of S to the right of L (Use Corollary 3.5):*
    If $x(p_k) > a$ **then**
    for all $(p_i,\ p_j) \in$ SMALL $\cup$ EQUAL insert p_i into CANDIDATES.
    for all $(p_i,\ p_j) \in$ LARGE insert p_i and p_j into CANDIDATES.

11. **return**(UPPER-BRIDGE(CANDIDATES, L)).

Claim: UPPER-BRIDGE runs in $O(N)$ worst case time, using linear space.

Proof: From the linear median finding algorithm of Blum et al. ([3], p99), step (4) takes $O(N)$ time. All the other steps can clearly be executed in $O(N)$ time also. Furthermore, at least one quarter of the points of S are eliminated in step (3) and step (8) or step (9), and hence are not contained in CANDIDATES. Thus the worst case time for this algorithm is given by:

$$f(n) = \begin{cases} O(1) & \text{if } n = 2, \\ f\left(\frac{3n}{4}\right) + O(n) & \text{if } n > 2. \end{cases}$$

This is a well-known recurrence whose solution is $O(n)$ ([3], p64)/ □

This concludes the section on the Kirkpatrick-Seidel algorithm. Practically, as mentioned in [1], this algorithm has some nasty constants because of the median finding algorithm. So, for planar convex hulls, Graham's Scan is most commonly used. We now consider the problem of constructing the convex hull of a finite set of points in more than two dimensions.

3.4 Finding Convex Hulls in 3 dimensions: The gift-wrapping method

As mentioned in Section 1, the convex hull of a finite set of points in arbitrary dimensions is a convex polytope. The case we have considered so far, namely the planar convex hull (convex polygon), is the simplest geometric object in this class of objects, and its counterparts in higher dimensions are much more complicated. Since any convex hull algorithm must produce a complete description of the boundary of the polytope, it must organize the computation of the facets of the convex hull in such a manner as to minimize the likely overhead for the d-dimensional case. The first attempt toward this goal was proposed by Chand and Kapur as long back as 1970 [9], and their algorithm is known as the *gift-wrapping method.* However, it was not until 1982 that the analysis of this technique was produced by Bhattacharya ([20], p125).

The basic idea of this algorithm is to proceed from a facet of the convex polytope to the adjacent facet, in the manner of incrementally wrapping a sheet around the polytope. We will only mention the final result of the analysis of the gift-wrapping method for arbitrary dimensions. We will, however, discuss the algorithm in detail for the 3-dimensional case. The discussion of this method and its analysis by Preparata and Shamos [[20], p125-130] is based on the assumption that the resulting polytope is *simplical*[12]; hence, each facet of the d-polytope is determined by exactly d vertices. The following theorem provides the basis of this method.

Theorem 3.6 ([20], Theorem 3.13, p126) *In a simplical polytope, a subfacet is shared by exactly two facets and two facets F_1 and F_2 share a subfacet e if and only if e is determined by a common subset (with $(d-1)$ vertices) of the sets determining F_1 and F_2 (F_1 and F_2 are said to be* adjacent *on e).*

The gift-wrapping method uses a subfacet e of an already constructed facet F_1 to construct the adjacent facet F_2 that shares e with F_1. The final result is stated in the following theorem.

Theorem 3.7 ([20], Theorem 3.14, p140) *The convex hull of a set of N points in d-dimensional space can be constructed by means of the gift-wrapping technique in worst-case time $O(N^{\lfloor d/2 \rfloor + 1})$.*

Among all the convex hull problems in dimensions greater than 2, the three-dimensional instance is of particular importance because of its relevance to a host of applications, ranging from

[12] A d-polytope P is a *d-simplex* if it is the convex hull of exactly $(d+1)$ affinely independent points. A set of $(d+1)$ points $p_1, p_2, \ldots, p_{d+1}$ in E^d is said to be *affinely independent* if the d vectors $p_2 - p_1, p_3 - p_1, \ldots, p_{d+1} - p_1$ are linearly independent. Any subset of these points is itself a simplex, and is a face of P. A d-polytope is called *simplical* if each of its facets is a simplex. This means that each facet (which is a $d-1$-simplex) of the simplical polytope is determined by exactly d points of the input set of points. Also, each of the facets will contain exactly d sub-facets. Thus, for a simplical 3-polytope, each of its facets will be a triangle that will contain exactly 3 edges of the polytope.

computer graphics to design automation, to pattern recognition and operations research [19]. From Theorem 3.7 above, it follows that for $d = 3$, the computation of the convex hull will require $O(N^2)$ operations in the worst case. However, from the results in Section 1.1, the general lower bound is $\Omega(N \log N)$, so the best we can hope for is an $O(N \log N)$ algorithm. Fortunately, this objective can be achieved as demonstrated by Preparata and Hong [19].

Their algorithm is a specialization of the gift-wrapping algorithm for the 3-dimensional case. However, the reduction in complexity from $O(N^2)$ to $O(N \log N)$ is due to the property that, from Euler's formula, the number of edges of the convex polytope is linearly related to the number of vertices. This property no longer holds for dimensions greater than 3 because it can be shown that there exist convex polytopes with N vertices, for $d \geq 4$, whose number of edges have cardinality $O(N^2)$ ([19], p88).

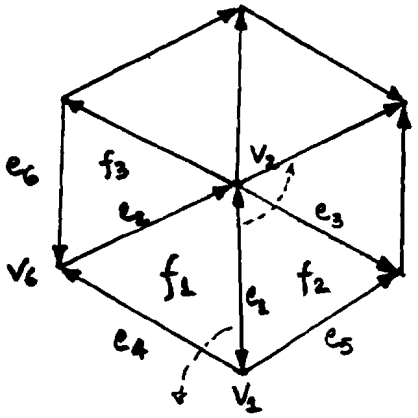
Let $S = \{p_1, p_2, \ldots, p_N\}$ be the input set of N points in E^3. For simplicity in the discussion of the algorithm, we will make a couple of assumptions about the given set of points. The first is that for any two points p_i and p_j in S, we have $x(p_i) \neq x(p_j)$, $y(p_i) \neq y(p_j)$, $z(p_i) \neq z(p_j)$. In other words, no two points of S can be on a common plane normal to the x, y, or z axis. The other assumption is that no four points of S are coplanar. What this means is that the convex polytope that the algorithm constructs will be simplical (See Section 1) i.e. it will have all its faces as triangles. As we will mention at the end of this section, the changes in the algorithm required for the unrestricted case are not very complicated.

Before we describe the algorithm, let us discuss, in brief, a data structure that is well-suited for the representation of convex polytopes in three dimensions, namely the *doubly-connected-edge-list (DCEL)*. The use of this data structure depends crucially on the following property of 3-D convex hulls, stated here without proof.

Theorem 3.8 *[1] The graph corresponding to a 3-dimensional convex polytope is a planar graph.*

Following Preparata and Shamos ([20], p15-16), a DCEL is described as follows. Let $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_M\}$ be the sets of vertices and edges of the planar graph. Assume that every edge in the graph is given an arbitrary direction. Each entry in the DCEL is indexed by an edge, and for each such edge, we have the following information:

- Two fields V_{tail} and V_{head}, that contain respectively the tail and head vertex of the edge.
- Two fields F_{left} and F_{right}, that contain the faces which lie respectively to the left and to the right of the edge oriented from V_{tail} to V_{head}.
- Two pointer fields P_{left} and P_{right}. Each contains a pointer to an edge index in the DCEL. P_{left} (P_{right}) points to the first edge encountered when we face the direction of orientation of the edge and proceed counter-clockwise around the tail (head), as shown in Figure 9.



Edge	V_{tail}	V_{head}	F_{left}	F_{right}	P_{left}	P_{right}
e_1	V_1	V_2	f_1	f_2	e_4	e_3
e_2	V_6	V_2	f_3	f_1	e_6	e_1
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 9: An example graph and the corresponding DCEL.

Thus, the above data structure can be represented as six arrays, each consisting of M entries. The reason the DCEL is doubly connected is that the reverse edge oriented from the head to the tail is implicitly stored in it. We read the three pairs of fields in the reverse order in order to get the correct information. Figure 9 shows an example graph and part of the corresponding DCEL. Suppose the graph has F faces f_1, f_2, ..., f_F. We can create two arrays called HV and HF, of sizes N and F respectively, such that $HV[i]$ contains some edge incident on v_i and $HF[i]$ contains some edge that surrounds the face f_i. These arrays can be created in $O(N)$ time by making a single scan through the first four fields of the DCEL. Using these arrays and the P_{left} and the P_{right} fields, it is straightforward to see that all the edges incident on a given vertex or all the edges enclosing a given face can be easily extracted from the DCEL in $O(N)$ time.

We are now ready to describe the algorithm for finding the convex hull in three dimensions using a DCEL. Divide and Conquer is the strategy used for solving the problem. Recall that S is the input set. We first sort the elements of S by their x-coordinates, and relabel them so that $i < j \iff x(p_i) < x(p_j)$. We then have the following recursive algorithm for finding the convex hull.

procedure 3D-ConvexHull(S);

1. **If** ($|S| \leq k_0$) (where k_0 is some small number) **then** construct CH(S) using some straightforward method. **Else** continue on with the next step.

2. (DIVIDE)
   $k := \lfloor N/2 \rfloor$;
   $S_1 := \{p_1, p_2, \ldots, p_k\}$;
   $S_2 := \{p_{k+1}, p_{k+2}, \ldots, p_N\}$;

3. (RECUR)
   $P_1 :=$ 3D-ConvexHull(S_1);

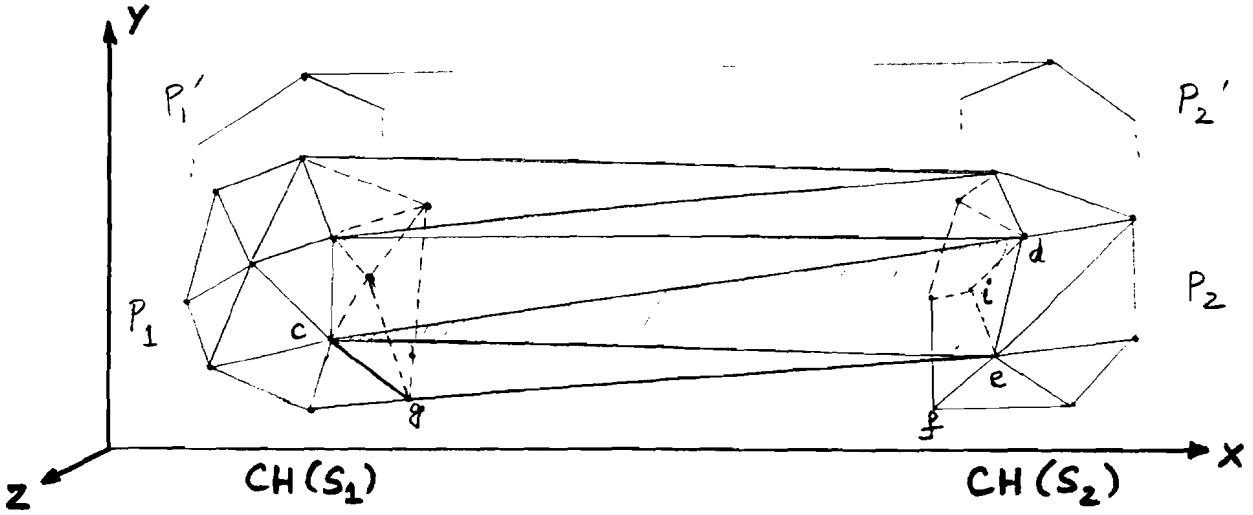


Figure 10: Merging the left and right convex hulls by constructing wrapping faces.

P_2 := 3D-ConvexHull(S_2);

4. (MERGE)
   Merge P_1 and P_2 to form CH($P_1 \cup P_2$);

Note that due to the pre-sorting step, the two convex polytopes P_1 and P_2 are non-intersecting. If the Merge step performs at most $M(N)$ operations, then an upper bound to the time $T(N)$ taken by 3D-ConvexHull is given by

$$T(N) = 2T(N/2) + M(N).$$

If the merge time $M(N)$ is $O(N)$, then $T(N)$ is $O(N \log N)$, and taking into account the initial sorting step, the overall complexity of the convex hull problem in three dimensions will be $O(N \log N)$.

The two polytopes P_1 and P_2 can be separated from each other by a plane h_0 that is normal to the x-axis. The final convex hull P will intersect this plane in a two-dimensional convex polygon. Every facet and edge of the final polytope P that is not already a facet or edge of either P_1 or P_2, must intersect h_0. Thus, the facets and edges of P can be constructed in a cyclic manner in the merge step ([12], p159). We now show that the merge step can indeed be performed in $O(N)$ time.

We first give the basic idea behind this step. Once the left and right polytopes have been obtained recursively, we merge the two in the manner of gift-wrapping them. We use a *cylinder of wrapping planes* [1] in such a way that each such plane goes through a hull edge of one polytope and an extreme vertex of the other (See Figure 10). Thus the left and right ends of this cylinder form cycles of edges from P_1 and P_2 respectively, and it will intersect the plane h_0 to form the convex polygon mentioned in the previous paragraph. Note that the wrapping faces[13] will all be triangles, and two adjacent wrapping faces will share an edge that is tangential to P_1 and P_2 [1].

[13]What we refer to as faces are actually *facets* (in rigorous terminology) of the three dimensional convex polytope.

There may, in general, be $O(N)$ such wrapping faces, and since finding each face may take $O(N)$ time, it seems like the merge step could $O(N^2)$ time. However, the nature of convex polytopes allows us to proceed from one face to the one adjacent to it in such a way that constructing the wrapping cylinder takes only $O(N)$ time.

In order to start building the wrapping cylinder, we need to start at an edge that is tangential to P_1 and P_2. An efficient way to find such an edge is as follows.

1. Project P_1 and P_2 onto a coordinate plane, say the $x - y$ plane. This can be easily done since we have access to all the vertex information from the DCEL. Let P_1' and P_2' be the projections of P_1 and P_2 respectively.

2. Now use one of the many known algorithms to find a common supporting line for P_1' and P_2'. For instance, we could use the UPPER-BRIDGE procedure described in Section 2.3, where S would be all the projected vertices, and any point on the open segment connecting the rightmost vertex of P_1 and the leftmost vertex of P_2, projected onto the $x - y$ plane will give us L.

Clearly, the edge joining the vertices of P_1 and P_2 that correspond to the two end-points of the bridge found above *supports* the two convex polytopes. This is the desired tangential edge in 3-dimensions, and finding it takes linear time since both step (1) and step (2) above take $O(N)$ time.

We now start constructing the wrapping faces. The following observations will help us do so [1].

1. At any point, only the two vertices that form the end points of the current tangential edge, say a and b, will be under consideration. We will need to look at all the edges incident on these two vertices in order to determine the next wrapping face.

2. The next wrapping face will be formed by the current tangential edge and by that edge of P_1 or P_2 which is incident on a or b, respectively, and offers the least amount of rotation to the current wrapping face. We can think of this as rotating the current wrapping face about the tangential edge until it hits one of the two convex polytopes along an edge incident on the vertices under consideration.

The DCEL plays a crucial role in the efficient implementation of the wrapping process. All the edge, vertex, and face information is readily available. We also assume that the array HV has been created so that all edges incident on a vertex can be obtained in a counter-clockwise direction or clockwise direction. As we find the tangential edges and the wrapping faces, we need to update the DCEL. The following steps [1] describe how this is accomplished. We will refer to Figure 10 throughout for concreteness of illustration.

1. Let $(c,\ d,\ e)$ be the current wrapping plane i.e. the most recently added one. Suppose that we advanced from edge $(c,\ d)$ to edge $(c,\ e)$ in order to do this, pivoting on vertex c. $(c,\ e)$ is the current tangential edge. First, we create a new node for $(c,\ e)$ in the DCEL. V_{tail} and V_{head} are c and e respectively. F_{left} is the current face $(c,\ d,\ e)$, and P_{left} is $(c,\ d)$. F_{right} and P_{right} will be added later. We now scan the edges incident on c and e.

2. First we scan the edges incident on e in P_2. We start at the edge $(d,\ e)$ in the edge incidence list for the current vertex e, and move in a counter-clockwise direction. We select a vertex e' connected to e, such that the face $(c,\ e,\ e')$ forms the largest *convex angle*[14] with the wrapping plane $(c,\ e,\ d)$ among the faces $(c,\ e,\ v)$, for v connected to e. This e' is f in Figure 10. *It is very important to observe that any intermediate edge* $(e,\ v)$ *(*$v \neq d$*) that we scan until we hit the edge* $(e,\ f)$ *is hidden by the face* $(c,\ e,\ f)$*, and hence becomes internal to the final hull* $CH(P_1 \cup P_2)$ *and need not be considered further as an edge incident on* e. These edges can then be removed from the incident edge list for e.

3. We now scan the edges incident on c in P_1. Since c was the pivoting vertex in the last stage, some of the edges incident on it have already been scanned (c was reached earlier than e was). Hence, we resume scanning, in the clockwise direction, at the last edge that was scanned in the incident edge list for vertex c. As in the previous step, we select a vertex c' among the vertices connected to c. This c' is g in Figure 10. Hidden edges are eliminated as described in the previous step.

4. Now, from the faces $(c,\ e,\ f)$ and $(c,\ e,\ g)$ found in steps (2) and (3), respectively, choose the one that makes a larger convex angle with $(c,\ e,\ d)$.

5. We can now add F_{right} and P_{right} in the node for $(c,\ e)$ in the DCEL, as promised in the first step.

   - For the case when the winning face in the previous step is $(c,\ e,\ g)$, F_{right} is $(c,\ e,\ g)$ and P_{right} is $(e,\ g)$. Also, P_{left} for $(c,\ g)$ is now $(c,\ e)$.
   - When the winning face is $(c,\ e,\ f)$, these fields are $(c,\ e,\ f)$ and $(e,\ f)$, respectively. P_{left} for $(f,\ e)$ is now $(c,\ f)$.

6. Advance the current wrapping plane and tangential edge to the newly created ones. In the first case (as above), these will be $(c,\ e,\ g)$ and $(e,\ g)$, and in the other case, $(c,\ e,\ f)$ and $(c,\ f)$, respectively.

7. The DCEL has now been updated to reflect all the changes as well as the new information. Go back to step 1 and repeat the process until the new tangential edge found in Step (6) is the one we started with, in which case Stop.

[14] A convex angle is one that is less than π.

Note that the edge deletion (of edges internal to the convex hull) process mentioned in step (2) may not entirely delete the edge from the DCEL. In fact, it is important that this does not happen because the other vertex of the edge being deleted, say e, may be one that is not yet known to be on the hull. Hence, e will need to be considered as one of the edges incident on that vertex, when it is being scanned. As a result of the manner in which the DCEL is updated, edges are only *effectively* deleted. The planar graph on the deleted edges will be disconnected from the final convex hull graph. At the end of the above process, we are automatically left with the DCEL of the merged convex hull. Note that the updated DCEL is still $O(N)$, even though the disconnected edges haven't been removed from the DCEL. Let us now analyze the time complexity of the above process.

Suppose we have a vertex v in P_1 under consideration and we are scanning all the edges incident on it. If after step (4), we choose an edge from the other polytope P_2 as the one that determines the wrapping face, we will have to rescan the edges incident on v again in the next iteration of the above process. However, we can eliminate edges that get hidden by the candidate for the wrapping face, since they cannot possibly lie on the convex hull. Hence, these edges do not get scanned again for the vertex v and scanning for v can resume at the last edge that was previously scanned. These hidden edges do not get looked up again, unless their other vertex gets scanned. It is clear that as we build the wrapping faces, every edge in P_1 and P_2 will be scanned at most twice, once for each of its vertices.

Let v_1 (e_1) and v_2 (e_2) be the number of vertices (edges) in P_1 and P_2 respectively. There are two types of angle comparisons (which take constant time) that we pay for.

1. In Steps (2) and (3), we make angle comparisons for all the scanned edges incident on the two vertices under consideration. From the observation in the previous paragraph, the total of these can be at most twice e_1 for P_1 and at most twice e_2 for P_2.

2. In step (4), we make angle comparisons between candidates for the wrapping face. This amounts to choosing between two vertices, one from P_1 and the other from P_2 (these are g and f in the process described above). Thus, the total number of such comparisons will be $O(v_1 + v_2)$.

The total number of comparisons, then, is $O(e_1 + e_2 + v_1 + v_2)$. *Due to the planar graph structure of* P_1 *and* P_2, e_1 *is* $O(v_1)$ *and* e_2 *is* $O(v_2)$, whence the total cost of the process of finding wrapping planes is $O(v_1 + v_2)$. Hence, $M(N)$ in Equation 3.4 is $O(N)$, leading to the result that the convex hull of N points in three-dimensional space can be computed in $O(N \log N)$ time.

Let us briefly mention the modifications necessary for the case of nonsimplical polytopes i.e. four or more points may be coplanar. The simplest way is to run the process as above, and let it introduce an artificial triangulation of nontriangular faces ([20], p139). This will happen if instead

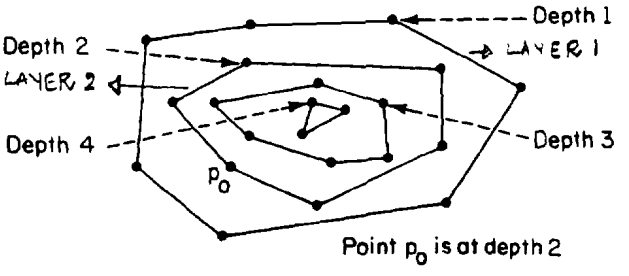


Figure 11: The depth of a point and the convex layers of a set.

of comparing convex angles (in steps (2), (3), and (4)), we compare angles that are less than or equal to π. Thus the final convex hull may have adjacent facets that are coplanar. After the merge step is completed, we can mark all edges that are coplanar with both its neighbors by making a single pass through the DCEL. We then delete all edges so marked. Clearly, this takes $O(N)$ time.

4 Some Applications of Convex Hulls

The claim that the convex hull problem is fundamental to computational geometry is justified because of the numerous applications it has to related problems. In this section, we will present a few such applications.

4.1 Convex Layers

We will first define the problem, and outline the algorithm for it. At the end of this section, we will state an application of the method to statistics.

Suppose we have a set of points of S. CH(S) forms the first layer of S. If we delete the points on this convex hull, and find the convex hull of the remaining points, we get the second layer of S. Continuing in this way, we can find the $(i + 1)$-th layer after finding the i-th layer (see Figure 11).

Definition 4.1 ([20], p166) The *depth of a point* p in a set S is the number of convex hulls (convex layers) that have to be stripped from S before p is removed. The depth of S is the depth of its deepest point.

This immediately leads to the following problem.

- Problem CONVEX LAYERS (CL): Given a set S of N points in the plane, what is the depth of each point in S.

Sorting is linear time reducible to Problem CL, as follows. Let the n points x_1, x_2, ..., x_n be the input to the sorting problem. Provide the CL algorithm with the input points $S = \{(x_i, 0) : 1 \leq i \leq n\}$. There will be two points for every depth j such that $1 \leq j < depth(S)$. At the layer depth(S), we may have one or two points. Clearly, the x-coordinates of the points that have depth i will be the i-th smallest and the i-largest of the sorted list. Thus, we will have to do one comparison for the points at each depth in order to obtain the final sorted list i.e. we need only $O(N)$ additional comparisons. This immediately gives us an $\Omega(N \log N)$ lower bound to problem CL.

The most obvious method of finding the convex hull of the N input points in $O(N \log N)$ time, discarding the hull points, and then finding the hull of the remaining points, and so on takes $O(N^2 \log N)$ time (this bound is reached when all the computed hulls are nested triangles). However, applying the Jarvis March method at each convex layer gives us a better algorithm ([20], p167). Let h_i denote the number of hull vertices in the i-th layer. At each layer, Jarvis's March takes $O(Nh_i)$ time, and hence the total running time will be $O(Nh_1 + Nh_2 + \ldots + Nh_k)$, where k is the depth of the input set. Since $h_1 + h_2 + \ldots + h_k = N$, this algorithm will take $O(N^2)$ time.

An algorithm reaching the $O(N \log N)$ lower bound was developed by Chazelle in 1983 [10]. In his paper, he shows that by organizing the deletions of points carefully, we can obtain the layers efficiently. The best known algorithm for finding convex layers in three dimensions is $O(N^{3/2} \log N)$.

The convex layers problem has an important application in statistics. As is well known, a central problem in statistics is to estimate a population parameter from a small, random sample drawn from the population. However, certain kinds of parameters are extremely sensitive to *outliers*, which are observations that lie abnormally far from most of the sample ([20], p165). It is important to reduce the effects of outliers, since they skew the sample data, thereby introducing significant error in the parameter estimate. The data is then said to be *robust*. There are several ways of accomplishing this goal. One method, known as the Gastwirth estimator, is based on the fact that in a random sample, the data that is closer to the center is more reliable and can be retained ([20], p165). In other words, the data that lies outside a certain fraction of the sample is discarded. Given the input set of points, we can use the algorithm for problem CL to peel off convex layers until the desired fraction of points remain.

4.2 The Farthest Pair Problem (Diameter of a Set)

The diameter of a set of points is used in *clustering* problems. Clustering is the grouping of similar objects ([20], p170). If we restrict our attention to objects in the plane, then "a measure of the *spread* of the cluster is the maximum distance between any two of its points", which is nothing but the *diameter* of the cluster ([20], p170). A well-known formulation of the clustering problem is as follows.

- Problem MINIMUM DIAMETER K-CLUSTERING [[20], p170]: Given N points in the plane, partition them into K clusters C_1, C_2, ..., C_K so that the maximum cluster diameter is as small as possible.

Regardless of the strategy used to compute a solution to this problem (whether exact or approximate), it is clear that we must have a method to determine the diameter of a set of points in the plane. In the rest of this section we will focus on this latter problem.

The Farthest Pair problem can be stated as follows.

- Problem FARTHEST PAIR (FP): Given a set S of N points, find a pair of points in S that are farthest[15]. If there are many such pairs, output any one of them.

The length of the edge connecting the farthest pair is known as the *diameter* of S. The brute force method for solving this problem is to compute the Euclidean distance between all possible pairs of points, and output any one of the pairs that has the largest such value. This gives us an $O(N^2)$ algorithm. It can be shown, however, that problem FP has an $\Omega(N \log N)$ lower bound[16].

There is an algorithm for problem FP in the plane that achieves this lower bound. In three dimensions, however, the best known algorithm is $O((N \log N)^{1.8})$ [24] and whether or not the gap between this bound and the $\Omega(N \log N)$ lower bound can be reduced is an open question. In this section, we will restrict our attention to the two dimensional case. Before we outline the algorithm, a couple of crucial observations are necessary. Let S be the input set of points in the plane.

Observation 1: [1] The diameter of S is formed by two extreme vertices of S.

Proof: Suppose not, i.e suppose that one of the vertices of the diameter of S is not extreme. Let $(d,\ a)$ be the farthest pair, and assume that a is not extreme. Then, a must lie in some triangle $(d,\ c,\ b)$, $c, b \in S$. Referring to Figure 12, the length of $\overline{da}$ is clearly less than that of $\overline{da'}$. Also,

[15] Here "farthest" means the largest Euclidean distance

[16] This can be done by showing a linear time reduction from the SET DISJOINTNESS PROBLEM to problem FP in the plane. If A and B are two sets of numbers, the set disjointness problem is to determine if they share an element. Using the algebraic decision tree model, this problem has been shown to have an $\Omega(N \log N)$ lower bound.

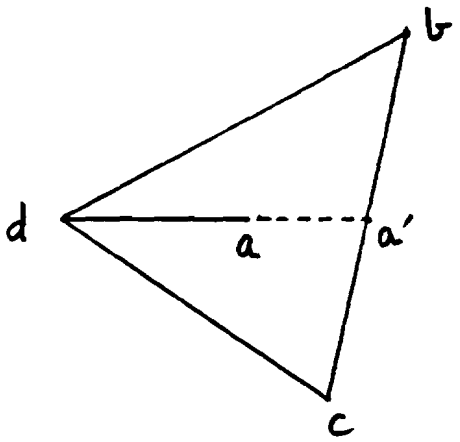


Figure 12: Figure for the proof of Observation 1.

since a' is closer to d than either b or c, it follows immediately that $|\overline{da'}|$ is less than $|\overline{db}|$ or $|\overline{dc}|$. But this contradicts the fact that $|\overline{da}|$ is the diameter. □

Definition 4.2 If l_1 and l_2 are supporting lines of S that go through p_1 and p_2 ($p_1, p_2 \in S$) respectively and l_1 is parallel to l_2, then (p_1, p_2) is said to be an *antipodal pair.*

The following theorem, which we state here without proof, provides the most important idea behind the efficient algorithm for problem FP in the plane.

Theorem 4.3 ([20], Theorem 4.18, p17) *The farthest pairs of a set S are the antipodal pairs of S of maximum length.*

Thus, in order to find the farthest pairs, we look at all possible antipodal pairs, and choose the ones that have maximum length. From Observation 1 and the above theorem, all antipodal pairs must be extreme points. Hence, we first construct the convex hull, and then determine the antipodal pairs from the hull vertices. We now give the outline of the algorithm, without going into implementation details.

procedure Farthest-Pairs-2D(S); [1]

1. Find the convex hull of S.
2. Let $p_0, p_1, \ldots, p_h$ be the hull vertices. Assume step (1) gives all the vertices of the convex hull in anti-clockwise order. We find all the antipodal pairs by making a rotational sweep around the convex hull.
   (a) Let l_1 be the supporting line determined by the first two points p_0 and p_1 on the hull. For all other points p_j, $2 \leq j \leq h$ on the hull, determine the distance between p_j

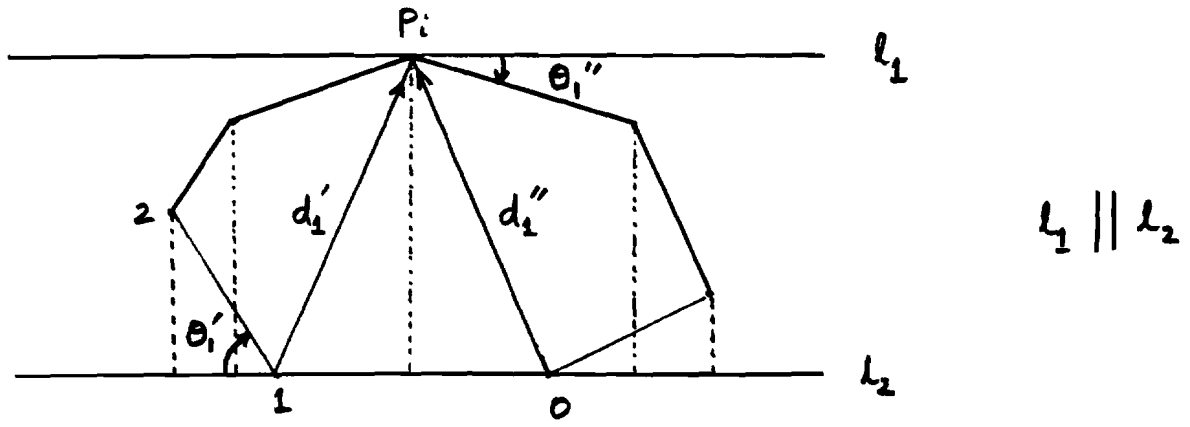


Figure 13: Figure for steps (2)-(a),(b) of the Farthest-Pairs-2D algorithm.

and l_1. Since all the points lie on a convex hull, this distance will increase as we make a counter-clockwise sweep around the hull until we hit a maximum, and then it will decrease (see Figure 13). Let p_i be the point that has the maximum such distance. Let l_2 be the line through p_i that is parallel to l_1 (if there are two such points, then l_2 is the line through both of them). Clearly, l_1 and l_2 are supporting lines through antipodal pairs of points.

(b) $\Theta_1' :=$ acute angle between l_1 and $\overline{p_1p_2}$;
$\Theta_1'' :=$ acute angle between l_2 and $\overline{p_ip_{i+1}}$;
(If there are two such p_i, then we consider the rightmost of them to determine Θ_1''.)
$\Theta_1 := \min(\Theta_1', \Theta'')$;
Let d_1' and d_1'' be as shown in Figure 13.
If $(d_1' > d_1'')$ **then**
$d_1 := d_1'$ and $e_1 := (p_1, p_i)$
else
$d_1 := d_1''$ and $e_1 := (p_0, p_i)$
Mark p_0, p_1 and p_i as already visited.
(If there are two such p_i, then compute the distance of p_0 and p_1 from each of them. The max of these will be d_1 and e_1 will be determined in the obvious way. Both the p_i will be marked.)

(c) We now rotate both l_1 and l_2, by the same amount, until one of them hits an edge in the convex hull. The amount of this rotation is Θ_1. Obviously, the rotated lines are still parallel supporting lines and they will now determine new antipodal pairs. As shown in Figure 14, suppose, without loss of generality, that l_2 is the first to hit the next edge on the convex hull. Θ_2', Θ_2'', d_2', and d_2'' are computed in a manner similar to that in the above step. They are shown in Figure 14.

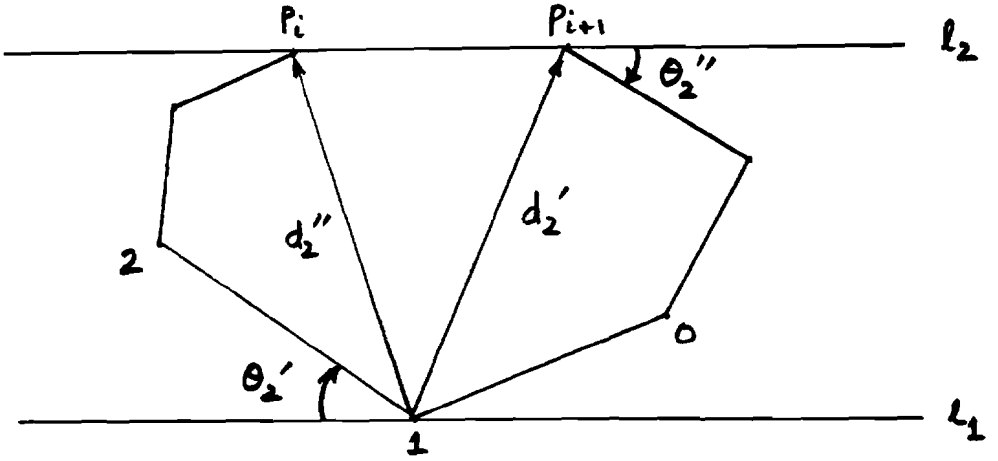


Figure 14: Figure for step (2)-(c) of the Farthest-Pairs-2D algorithm.

$\Theta_2 := \min(\Theta_2', \Theta_2'')$;
$d_2 := \max(d_2', d_2'')$ and e_2 is the corresponding edge (as in the above step).
Mark vertices as before

(d) Continue the above rotational sweep until every vertex on the hull has been marked. We will have h distances $d_1, d_2, \ldots, d_h$ and the corresponding h edges $e_1, e_2, \ldots, e_h$.

(e) The diameter of S will be $\max\{d_1, d_2, \ldots, d_h\}$ and the edge corresponding to the diameter will give us a farthest pair of S.

We can have at most h antipodal pairs, where h is the number of points on the hull. The above algorithm goes through all possible antipodal pairs as it does the rotational sweep. Let us analyze the complexity of the above algorithm. Step (1) takes $O(N \log N)$ time, as we have demonstrated many times in this paper. Consider step (2). (a) takes $O(h)$ time. Steps (b)-(d) are repeated for each antipodal pair, and hence this takes $O(h)$ time. Finally, step (e) is also $O(h)$. Since all N points may lie on the hull, step (2) in the worst case is $O(N)$[17]. Thus, problem FP in the plane can be solved in optimal $O(N \log N)$ time.

4.3 Voronoi Diagrams

The problem of constructing the Voronoi Diagram in the plane arises in a number of areas, one of the most important of which are the proximity problems. A few examples of such problems are (Euclidean) minimum spanning trees, clustering, and contour maps [8]. "The Voronoi diagram in fact expresses the proximity information of the set at hand in a very explicit and computationally useful manner." ([12], p293) The Voronoi diagram problem is defined for arbitrary dimension, but

[17]It follows immediately from this that the diameter of the vertices of a convex polygon can be found in linear time.

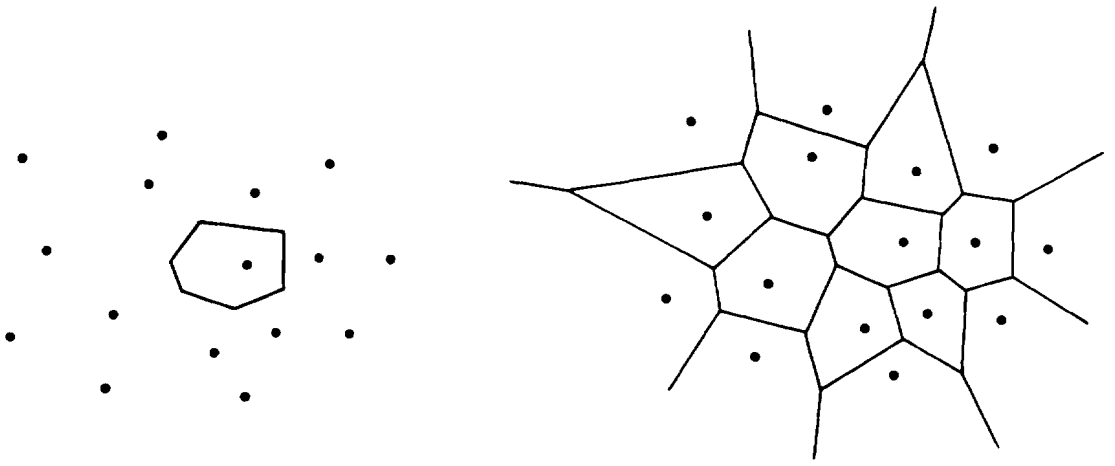


Figure 15: (a) A Voronoi polygon; (b) A Voronoi diagram.

we will restrict our attention to the planar case. It is appropriate to mention here that efficient algorithmic techniques for finding Voronoi diagrams are available only in the two-dimensional case.

Definition 4.4 *[1]* A *Voronoi Diagram* of a set S in the plane, where $S = \{p_1, p_2, \ldots, p_N\}$, is a partition of the plane into regions $r_1, r_2, \ldots, r_N$ such that any point in the region r_i is closer to p_i than to any other point in S.

Given any two points p_i and p_j in the plane, the set of all points that are closer to p_i than to p_j is just the half plane containing p_i and defined by the perpendicular bisector of the segment $\overline{p_i p_j}$. So, given a set of N points, the region that is closer to point $p_i \in S$ than to any other point in S will be the intersection of $N - 1$ half-planes. In E^2, this is a convex polygon of at most $N - 1$ sides. This polygon, $V(i)$, is called the *Voronoi polygon associated with* p_i. A Voronoi diagram is nothing but a mesh of N such convex regions. A Voronoi polygon and the Voronoi diagram for a set of points is shown in Figure 15. The vertices and the edges of the diagram are called the *Voronoi vertices* and *Voronoi edges*, respectively.

Note that some of the convex regions may be unbounded, as can be seen in the figure. It can be shown that these can only be the regions associated with the vertices on the convex hull of S. Numerous other important and interesting properties of Voronoi diagrams can be proved. We will not go into the details of these, since the primary purpose of this section is to see the relationship between convex hulls and Voronoi diagrams.

Constructing the Voronoi diagram of N points in the plane takes $\Omega(N \log N)$ time in the worst case, using the algebraic decision-tree computation model. A linear time reduction of sorting to this problem gives us this result ([20], p206), as follows. The Voronoi diagram of N points in one

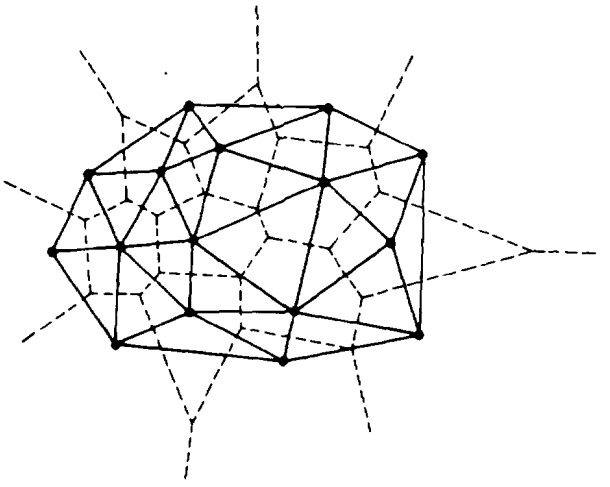


Figure 16: The straight-line dual of the Voronoi diagram (the Delaunay triangulation).

dimension is nothing but the sequence of $N - 1$ midpoints of the adjacent input points, which can be found in $O(N)$ time. Consecutive pairs of these midpoints can, in linear time, give us the input points in sorted order (assuming, of course, that the Voronoi algorithm outputs the midpoints in some particular order).

Shamos and Hoey [21] have given an $O(N \log N)$ divide-and-conquer algorithm for finding the Voronoi diagram. Using a technique known as the sweep plane technique, Fortune [13] also demonstrates an optimal algorithm. We will discuss here the connection between Voronoi diagrams and convex hulls. It can be shown that there is a relationship between the *dual* of the Voronoi diagram of N points in d-dimensions and the convex hull of those points in $d + 1$ dimensions, under a suitable projection. Thus, the planar Voronoi diagram can be obtained from the convex hull of a suitable set of points in three dimensions.

The straight-line dual of the Voronoi diagram of S in the plane is obtained by adding a straight line segment between every pair of points (p_i, p_j) of S such that $V(i)$ and $V(j)$ share an edge (See Figure 16).

Definition 4.5 ([20], p19) The *triangulation of a finite set* S of points in the plane is obtained by joining the points of S by nonintersecting straight line segments so that every region internal to the convex hull of S is a triangle.

It can be shown that the straight line dual of the Voronoi diagram gives a triangulation of S, and it is known as the *Delaunay triangulation.* We will now outline the important ideas behind the method of finding Delaunay triangulations from convex hulls[18] and will keep the discussion fairly intuitive [1]. Let $S = \{p_1, p_2, \ldots, p_N\}$ be a set of N points in the plane.

[18]When talking about properties of Voronoi diagrams, the assumption that no four points of the input set are co-circular is made. In the absence of such an assumption, lengthy details, that do not throw any new light on the property at hand, must be included in the statements and proofs of the properties.

1. Project each point of S onto the paraboloid U given by the equation $z = x^2 + y^2$. Let $p_i(U)$ denote the projection of p_i onto U and let $S(U) = \{p_1(U), p_2(U), \ldots, p_N(U)\}$.
2. Construct the convex hull, $\mathrm{CH}(S(U))$, of $S(U)$ in three dimensions.
3. Now "look up" at the convex hull from below it and project all the faces of $\mathrm{CH}(S(U))$ that can be "seen", vertically onto the $x-y$ plane (assume that we cannot look through faces i.e. that the faces are opaque). This gives us the Delaunay triangulation of S.

Since no four points of S are cocircular, no four points of $S(U)$ will be coplanar. Hence all the faces of the convex hull will be triangles, and the projection of these triangles gives us the triangulation. The following lemma is used crucially in proving that the final result is indeed a Delaunay triangulation. We state it here without proof.

Lemma 4.6 *[1] Let C be any circle in the $x-y$ plane and let p be a point in that plane. $p(U)$ is the projection of p onto U. Project C onto U and call it $C(U)$. Let $H(C)$ be the plane containing $C(U)$ (Note that $C(U)$ is an ellipse). Then*

1. *If p is outside C, then $p(U)$ is above $H(C)$.*
2. *If p is inside C, then $p(U)$ is below $H(C)$.*
3. *If p is on C, then $p(U)$ is on $C(U)$ (obviously).*

If we can show that the circumcircle of any triangle in the triangulation does not contain any points of S inside it, then we have proved that the triangulation is indeed a Delaunay triangulation (this follows immediately from some of the properties of Voronoi diagrams). Let C be the circumcircle in question. Let p be some point of S that is inside C. Then, from Lemma 4.6-(2), we have that $p(U)$ is below $H(C)$. However, since $H(C)$ is also a plane going through a face of $\mathrm{CH}(S(U))$, it is a supporting plane, and hence all points of $S(U)$ will lie above it. We have a contradiction, implying that p cannot lie inside C.

It can be shown that the number of Voronoi vertices and edges is linear in N. It follows immediately from this that the number of edges in the Delaunay triangulation must also be $O(N)$. Therefore, having found the triangulation by the given method, we can find the Voronoi diagram for S in $O(N)$ time. Steps $(1)-(3)$ clearly take $O(N \log N)$ time, and we have an optimal algorithm for planar Voronoi diagrams.

5 Conclusion

The $\Omega(N \log N)$ lower bound to the convex hull problem was shown, as well as the existence of efficient algorithms for the important cases of two and three dimensions. All the algorithms dis-

cussed in the paper are sequential and off-line. Off-line algorithms require all the data points to be available before any processing can begin. In many geometric applications that run in real-time, all the data may not be available at the same time and the convex hull needs to be constructed as and when the data is received. For this purpose, *on-line* convex hull algorithms that run in optimal $\theta(N \log N)$ time (i.e. real-time) can be shown [18]. In addition, a randomized algorithm that runs in $O(N)$ expected time has been demonstrated for the planar case [7]. There is no known efficient randomized algorithm for finding convex hulls in three dimensions. Finding approximations to the convex hull is useful in applications that need rapid solutions even at the expense of accuracy; statistical applications, where the observation points are themselves approximate, come immediately to mind. Efficient approximation algorithms exist for the two and three dimensional cases [6].

The development of sequential algorithms for geometric problems has been an active area of research since 1975. However, some of the first publications about parallel algorithms for such problems appeared only as recently as 1985. An optimal parallel algorithm for the planar convex hull problem appears in [2]. In this paper, Aggarwal *et al.* also demonstrate algorithms for finding convex hulls in three dimensions and Voronoi diagrams in two dimensions which, though suboptimal, are the best known parallel algorithms. There are very few problems in computational geometry for which parallel algorithms have been developed. For instance, it is not known if the important technique of prune and search can be parallelized and no parallel algorithm is known for the convex layers problem. Needless to say, parallel computational geometry is a burgeoning area of active research.

In summary, even though research endeavors relating to the convex hull problem have reached sophisticated levels, the problem, along with those related to it, continues to be the focus of vigorous research. Further developments in this area are bound to provide new insights fundamental to computational geometry.

References

[1] A. Aggarwal. Computational geometry. Lecture Notes, Research Seminar Series, MIT, Spring 1988.

[2] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3:293–327, 1988.

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

[4] S. G. Akl. Two remarks on a convex hull algorithm. *Info. Proc. Lett.*, 8(2):108–109, 1979.

[5] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proc. 15th ACM Annual Symp. on Theory of Computing*, pages 80–86, May 1983.

[6] J. L. Bentley, G. M. Faust, and F. P. Preparata. Approximation algorithms for convex hulls. *Comm. ACM*, 25:64–68, 1982.

[7] J. L. Bentley and M. I. Shamos. Divide and conquer for linear expected time. *Info. Proc. Lett.*, 7:87–91, Feb. 1978.

[8] K. Q. Brown. Voronoi diagrams from convex hulls. *Info. Proc. Lett.*, 9(5):223–228, Dec. 1979.

[9] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. *J. ACM*, 1(17):78–86, Jan. 1970.

[10] B. M. Chazelle. Optimal algorithms for computing depths and layers. In *Proc. 21st Allerton Conference on Comm., Control and Comput.*, pages 427–436, Oct. 1983.

[11] D. Dobkin and R. Lipton. On the complexity of computations under varying set of primitives. *Journal of Computer and Systems Sciences*, 18:86–91, 1979.

[12] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987.

[13] S. J. Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[14] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Info. Proc. Lett.*, 1:132–133, 1972.

[15] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Info. Proc. Lett.*, 2:18–21, 1973.

[16] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15(1):287–299, Feb. 1986.

[17] P. McMullen and G. C. Shepard. *Convex Polytopes and the Upper Bound Conjecture.* Cambridge University Press, Cambridge, England, 1971.

[18] F. P. Preparata. An optimal real-time algorithm for planar convex hulls. *Comm. ACM*, 22:402–405, 1979.

[19] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Comm. ACM*, 20(2):87–93, Feb. 1977.

[20] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag New York Inc., 1985.

[21] M. I. Shamos and D. Hoey. Closest-point problems. In *Sixteenth Annual IEEE Symposium on Foundations of Computer Science*, pages 151–162, Oct. 1975.

[22] J. M. Steele and A. C. Yao. Lower bounds for algebraic decision trees. *J. Algorithms*, 3:1–8, 1982.

[23] A. C. Yao. A lower bound to finding convex hulls. *J. ACM*, 28:780–787, 1981.

[24] A. C. Yao. On constructing minimum spanning trees in k-dimensional space and related problems. *SIAM J. Comput.*, 721–736, 1982.