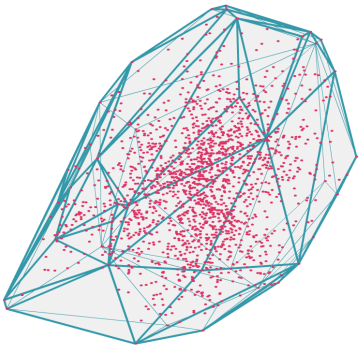


Habib University

CS 424: Joy of Theoretical Computer Science

Geometric Algorithms

The Convex Hull Problem in 2 and 3 Dimensions

Authors:
M. Usaid Rehman
Syed Anus Ali
Faraz Ozair

June 1, 2021

Contents

Abstract

Computational geometry is an important area of study in computer science that comprises several important problems. We present a survey of the well-known convex hull problem which is heavily applied in areas ranging from mathematics and economics to quantum physics We will define the problem and explore three algorithms in the planar case – Jarvis march, Graham scan, and Quickhull – while briefly discussing the 3-dimensional case and an algorithm to compute it.

1 Introduction to Computational Geometry

1.1 Historical Context

The Ancient Egyptians and Greeks were brilliant applied mathematicians especially when it came to geometry. The motivation for their attempt to solving geometric problems was the need to tax lands accurately and to perform architectural endeavors. Over the years, the use of geometry became more prevalent and was at the heart of mathematical thinking.

An important ancient figure in geometry was Euclid who is often referred to as the founder of geometry. Euclid's formulation of geometry prevailed for almost two thousand years until Rene Descartes introduced the idea of coordinates which greatly increased the computational power that mathematicians had to compute geometric problems. During the 17^{th} to 19^{th} centuries, many mathematicians worked on geometric problems – deriving various results through the method of construction. Gauss, Euler, Abel and many others connected various areas of mathematics with geometry. [7]

From the 1960s onwards, there was a lot of work being done in modern geometry. Geometric modelling by means of spline curves and surfaces was an important part of the study of geometry within a computational context. The term "computational geometry" was coined by Marvin Minsky and first used in his book called *Perceptrons*, [11] published in 1969. The book dealt with the idea of pattern-recognition and was an early work in artificial intelligence.

Furthermore, the advent of computer graphics and the growing sophistication of graphic tools required a paradigm of computation that would allow computer scientists to deal with geometry in an algorithmic sense and solve problems that arise in practical scenarios. The large number of application areas relying on geometry have been the incubation bed for the discipline of computational geometry. Some problems that inspired computational geometry include the Euclidean traveling salesman, minimum spanning tree, hidden line, linear programming and many others. However, it was only until the 1970s that the term computational geometry was properly applied and a paper was published [13], introducing the discipline.

1.2 An Overview

We still have not truly defined what computational geometry is. We shall do that in this section. Computational geometry is the term used to describe the subfield of algorithm theory that involves the design and analysis of algorithms for problems involving geometric inputs and outputs.

The discipline of computational geometry grew rapidly in the 1970s and the 1980s. It is still a fairly active field of research. Computational geometry has developed as a generalization of the study of algorithms for sorting and searching in 1-dimensional space to problems involving multi-dimensional inputs.

Research on computational geometry has mostly focused on 2-dimensional space – which will form the bulk of this paper – and to some extent on 3-dimensional space. Even in higher dimensional spaces, it is usually assumed that the dimension of the space is small constant usually lower than 10. However, approximation algorithms have been used in computational geometry to solve certain problems in very high dimensions.

Since computer scientists usually deal with discrete algorithms computational geometry has also mainly focused on the discrete aspects of geometric problem solving. This mixture of discrete algorithms with geometric problems gives rise to an interesting set of problems. Moreover, computational geometry primarily deals with straight or flat objects (lines, line segments, polygons, planes, polyhedra) or simple curved objects like circles. [12]

One of the major aims of computational geometry is to provide a set of basic geometric tools by which application areas can build algorithms and a set of theoretical analytical tools which can be used to analyze the performance of these algorithms.

1.3 Famous Problems in Computational Geometry

Computational geometry encompasses several different problems that utilize and manipulate different geometric structures. An example of a typical problem in computational geometry is the *shortest path problem*. The problem statement is: Given a set of polygonal obstacles in the plane, find the shortest obstacle-avoiding path from given start point to some end-point. One approach would be to reduce this problem to a graph problem and then use a shortest-path algorithm like Dijkstra's algorithm.

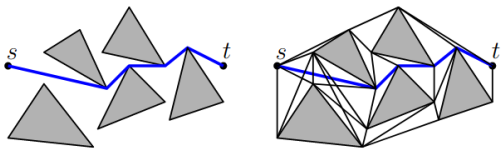


Figure 1: The Euclidean shortest path problem.

However, using a geometric approach would result in better algorithms. Dijkstra is usually implemented in $O(n^2)$. A geometric solution to the shortest path problem can be implemented in $O(n^2 \log n)$ using a simple algorithm and in $O(n \log n)$ using a more complex algorithm.

In the remainder of this section we will briefly mention some of the other famous problems in computational geometry.[2]

Convex Hulls: Computing the convex hull of a given a set of points was one of the first problems identified in computational geometry. Convexity is an important geometric property with a wide range of applications. It is also the focus of the rest of this paper.

Intersections: Determining when two sets of objects intersect one other is an important geometric problem. To determine the intersection of complex geometric structures such as polygons, we look at intersections of simpler entities within the complex structures such as line segments and points.

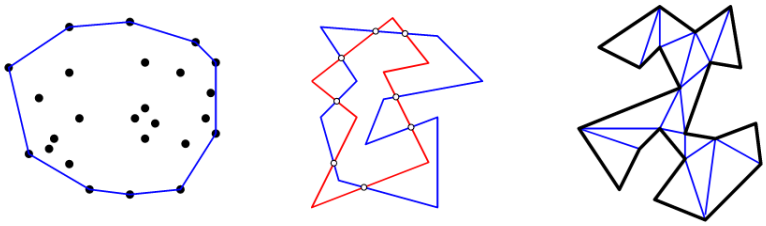


Figure 2: Convex hulls, intersections, and polygon triangulation.

Triangulation and Partitioning: Triangulation is the term used to describe the general problem of subdividing complex geometric structures into a disjoint collection of simpler objects. The simplest region that a planar object can divided into is a triangle. In the 3-dimensional case, polyhedra divide into tetrahedron and in n-dimensions – a simplex.

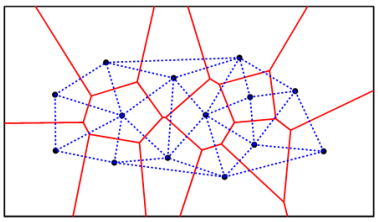


Figure 3: Voronoi diagram and Delaunay triangulation

Voronoi Diagrams and Delaunay Triangulations: Given a set of points S in space, which of the points in S, is closest to a point $s' \notin S$. A technique to solve this is to subdivide the space into regions according to which point is closest.

This partition is called a *Voronoi diagram*. The dual structure of this partitioning is called the *Delaunay triangulation* which also has interesting properties.

Geometric Search: Given a static data set, pre-process the data into a data structure in a way that some types of queries can be answered efficiently. Several different problems lie in this category such as *point location* or *range searching* – both of which have uses in areas like data analysis and querying. The famous *nearest neighbor* problem also lies in this category.

2 Geometric Preliminaries

Computational geometry is based heavily on Euclidean geometry. We mostly consider sets of points in Euclidean space. Since Euclidean space is based on real numbers which are continuous – uncountably infinite. Therefore, the point sets we consider have to be either finite or *finitely specifiable*. This means that they can be defined using finite strings of parameters. Examples of this include line segments, lines, planes, polygons – all objects that can be defined using finite collections of points.

2.1 Euclidean Geometry

By $\mathbb{R}^d$ we denote the *d-dimensional Euclidean space*, the metric space of the d-tuples $(x_1, \ldots, x_d)$ of real numbers $x_i, i = 1, \ldots, d$ with metric

$$\sqrt{\sum_{i=1}^{d} x_i^2}$$

Now let's look at definitions of some geometric structures considered in computational geometry.

Definition 1 (Point)**.** A d-tuple $(x_1, \ldots, x_d)$ denotes a point $p \in \mathbb{R}^d$. We can also think of this point as the d-component position vector from the origin to point p.

Definition 2 (Line)**.** Given two distinct points q_1 and q_2 in $\mathbb{R}^2$, the linear combination

$$\alpha q_1 + (1 - \alpha) q_2 \qquad (\alpha \in \mathbb{R})$$

is a *line* in E^d.

Definition 3 (Line Segment)**.** Given two distinct points q_1 and q_2 in $\mathbb{R}^2$, the convex combination

$$\alpha q_1 + (1 - \alpha) q_2 \qquad (\alpha \in \mathbb{R},\ \ 0 \leq \alpha \leq 1)$$

describes the *straight line segment* joining the two points q_1 and q_2.

Definition 4 (Polygon)**.** In $\mathbb{R}^2$, a *polygon* is described by a finite set of line segments such that every segment extreme is shared by exactly two edges and no subset of edges has the same property. The segments are the *edges* and their extremes are the *vertices* of the polygon.

Definition 5 (Polyhedron)**.** In $\mathbb{R}^3$, a *polyhedron* is defined by a finite set of plane polygons such that every edge of a polygon is shared by exactly one other polygon (adjacent polygon) and no subset of polygons has the same property. The vertices and the edges of the polygons are the vertices and the edges of the polyhedron, and the polygons are the *facets* of the polyhedron.

2.2 Convex Sets & Convex Hulls

Since this paper's focus is on the convex hull problem, it would be logical to look at definitions regarding convexity and convex sets. We do so in this section.

Definition 6. Convex Sets

A set $K \subset \mathbb{R}^d$ is *convex* if, for any two points q_1 and q_2 in K, the line segment joining q_1 and q_2 is entirely contained in K.
The intersections of convex sets is a convex set. We leave the proof as an exercise for the reader.

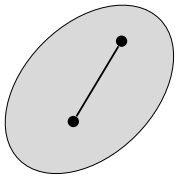


Figure 4: A convex set.

Definition 7. Convex Hull

The *convex hull* of a set of points S in $\mathbb{R}^d$ is the boundary of the smallest convex set containing S. Alternatively, it can be defined as the intersection of all the convex sets containing the points S.

It is also important to note that the intersection defined in Definition 7 is the intersection of uncountably infinite convex sets owing to the completeness of $\mathbb{R}$. Moreover, a polygon is convex if its interior is a convex set. Similarly, a polyhedron is convex if its interior is a convex set.

3 The Convex Hull Problem

We now look at the convex hull problem in computational geometry. It is one of the most central problems in computational geometry and the most widely studied. Recall that in Section 2 we stated that we work with finite or finitely specifiable sets of points. Let's define two versions of the convex hull problem.[12]

Problem (Convex Hull). Given a set S of $n \in \mathbb{N}$ points in $\mathbb{R}^d$, construct its convex hull (the complete description of the boundary $\mathrm{CH}(S)$).

Problem (Extreme Points). Given a set S of $n \in \mathbb{N}$ points in $\mathbb{R}^d$, identify those that are the vertices of $\mathrm{CH}(S)$.

The first version of the problem is asymptotically as hard as the second since the output of Convex Hull can be a valid solution to Extreme Points if we recopy the vertex set produced by Convex Hull as an unordered list of points, therefore one of the problems is reducible to the other.

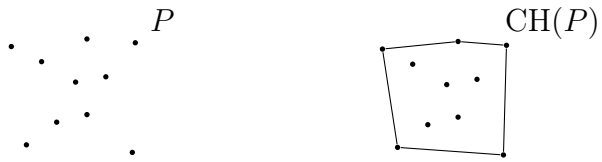


Figure 5: A point set and its convex hull.

3.1 A Motivating Example

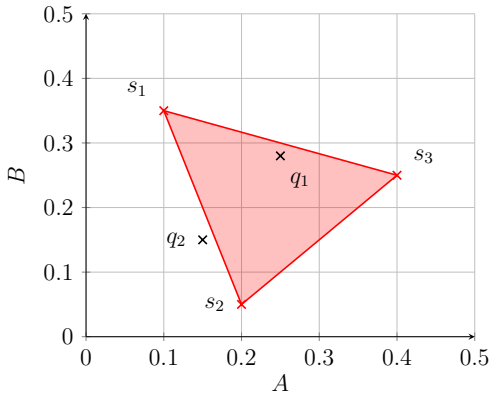
Before we move look at the algorithms for the convex hull problem, let's look at a very simple practical example that demonstrates the use of the convex hull.[2] Imagine you work at a factory that produces three substances s_1, s_2, s_3 from two chemicals A and B. These substances contain A and B in the following concentrations:

subst.	fract. A	fract. B
s_1	10%	35%
s_2	20%	5%
s_3	40%	55%

You are tasked with creating two new substances q_1 and q_2 using only s_1, s_2, s_3 which contain A and B in the following concentrations:

q_1	25%	28%
q_2	15%	15%

You want to know if this is possible. How would we solve this problem? The first thing we should do is plot all the points with respect to their concentrations of A and B.



Based on the plot, if we were to create a substance using s_1 and s_2, the concentrations of the new substance would lie on the line segment joining them. This idea would be symmetric for s_1 & s_3 and s_2 & s_3, or we can say that the new substances would be convex combinations of the three older substances. Similarly, if we were to use all three substances, the substances produced would be convex combinations of the three points, i.e. lying inside the shaded region. Therefore, you can produce q_1 using the three substances but you can't produce q_2. The shaded region is the convex hull of the three points.

Remark. We can observe that a substance $q \in \mathbb{R}^2$ can be created using substances in $S \subset \mathbb{R}^2$ if and only if $q \in \mathrm{CH}(S)$.

4 Convex Hull Algorithms

Computing the convex hull has been a heavily researched problem that has led to computer scientists develop many different algorithms. These algorithms also employ several different techniques and often borrow from various areas of algorithm design – including divide-and-conquer, prune-and-search, marriage-before-conquest, approximation algorithms, and others. Following are some of the famous algorithms for the convex hull problem, along with their time complexity:

Algorithm	**Time Complexity**
Jarvis March	$O(nh)$
Graham Scan	$O(n \log n)$
Quickhull	$O(n \log n)$
Andrew's Algorithm	$O(n \log n)$
Incremental Convex Hull Algorithm	$O(n \log n)$
Kirkpatrick-Seidel Algorithm	$O(n \log h)$
Chan's Algorithm	$O(n \log h)$

Table 1: Some convex hull algorithms and their complexities.

General Position: To make algorithm design easier, we make an important assumption about the position of the input points. We assume that the points of the input point set are in *general position*. This means that degenerate configurations – such as colinearity of three points – will not arise in the input set. A point set in general position remains in general position if the points are perturbed infinitesimally. This assumption is ony a convenience to simplify the algorithm design process by avoiding the need of dealing with lots of special cases. Assumptions about general position never have a critical impact on the performance of geometric algorithms.

4.1 Jarvis March

Jarvis march is one of the earliest and also one of the least efficient algorithms for computing convex hulls. It was introduced by R. A. Jarvis in 1973.[10] It is also known as the gift-wrapping algorithm and it extends to the 3-dimensional case. For now, we will only look at the planar case. The main idea behind Jarvis March is to iteratively find each point of the convex hull by comparison of polar angles between the current point and the potential next point.

The algorithm begins with a counter i set to 0, and a vertex p_0 (e.g. the leftmost point) that is known to be on the convex hull. Then, it selects the point p_{i+1} such that all points are to the right of the line $p_i p_{i+1}$. The point p_{i+1} can be found by comparing polar angles with respect to point p_i as the center of the polar coordinates as shown in Algorithm 1.

Algorithm 1: JARVIS MARCH

Input: A point-set S.
Output: The convex hull P

```
1  P ← ∅
2  pointOnHull ← leftmost point in S
3  i ← 0
4  repeat
5  |   endPoint ← S[0]
6  |   for j ← 0 to j ← |S| do
7  |   |   if S[j] is on the left of the line from P[i] to endPoint then
8  |   |   |   endPoint ← S[j]
9  |   i ← i + 1
10 |   pointOnHull ← endPoint
11 until pointOnHull = endPoint
12 return P
```

The algorithm starts by taking the left most point from the input set S and adding it into our output convex hull set P. We then iterate from that point over the rest of the n points in S. We compute the polar angle between the current point p_i and the rest of the n points p_{i+1} – which we use to see which point lies farthest to the left of p_i. Our leftmost point becomes our new p_i and gets added to our convex hull P. We repeat this process until we return to the initial point on the convex hull in a counterclockwise fashion – returning the convex hull in h steps where $h = |\text{CH}(S)|$. Here is a link to an animation for Jarvis march.

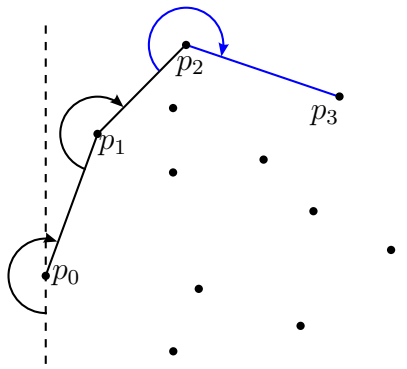


Figure 6: The working of Jarvis march.

4.1.1 Time Complexity of Jarvis March

There are two loops in the Jarvis march algorithm. We say that these points are h in quantity. There are two loops in the algorithm. The inner for loop checks for every point in the input set S and the outer loop repeats for each of the h points on the convex hull. So, the computational complexity is $O(nh)$.

Since h could vary for different inputs, the true efficiency of Jarvis March depends on the relative sizes of h and n. If h is smaller than $\log n$, then Jarvis march works better compared to $O(n \log n)$ algorithms. But in most cases Jarvis march is easily outperformed by several other algorithms.

4.2 Graham Scan

Graham scan[8] makes use of the *incremental construction* technique to find the convex hull of a set of input points. In this technique, objects are added one at a time, and the structure is updated with each new insertion. The order of insertion matters in incremental construction so that we don't have to check whether the object belongs in the structure upon each insertion.

This algorithm starts by picking lowest and leftmost point out of the input set S and then moving counterclockwise and incrementally finding rest of the points that lie on the convex hull. The key idea behind Graham scan is that it checks for a left turn every time we move from one point on the convex hull towards the next point. We state an outline[5] of the Graham scan algorithm in Algorithm 2.

Algorithm 2: GRAHAM SCAN

Input: A point set P
Output: The convex hull of P, CH(P)

```
1  let p_0 be the minimum and left-most point in P
2  let ⟨p_1, p_2, ..., p_n⟩ be the remaining points in P sorted by polar angle in
     counterclockwise order around p_0
3  let S be an empty stack
4  PUSH(p_0, s)
5  PUSH(p_1, S)
6  PUSH(p_2, S)
7  for i ← 3 to i ← n do
8  |  while the angle formed by points TOP(S), NEXT_TO_TOP(S), and p_i
   |    make a non-left turn do
9  |  |  POP(S)
10 |  PUSH(p_i, S)
11 return S
```

Graham scan maintains a stack to compute the convex hull. The algorithm first computes the polar angles of all the points with respect to the point p_0 and then sorts all the points in counterclockwise order based on the polar angles. The first two points are then pushed onto the stack. The algorithm then considers each point stored in the sorted array in sequence. Then for each point, it checks if traveling from the two points preceding this point – that are stored in the stack – constitutes making a left turn or a right turn.

If a right turn is made, the second-to-last point is not part of the convex hull and is therefore popped from the stack and then discarded. Then the same determination is made for the current point, and two points preceding the point that was just discarded. This is repeated until the stack contains only points on which left turns were made. The process stops when the algorithm reaches the starting point through counterclockwise movements.

4.2.1 Time Complexity of Graham Scan

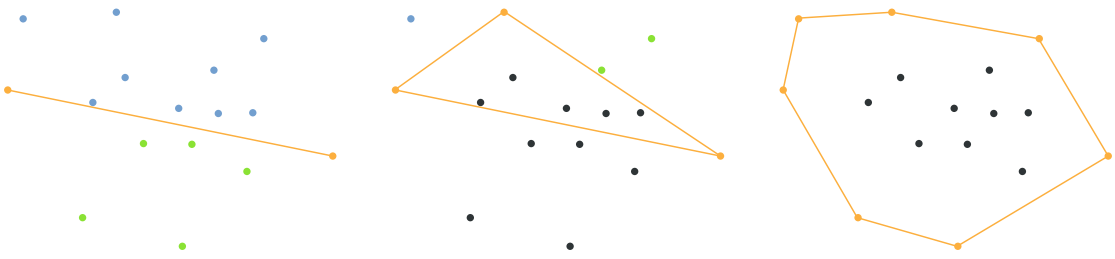
Graham scan performs sorting at the start of the algorithm which can be done using heapsort which is known to be in $O(n \log n)$. At first glance, in lines 7–10 in Algorithm 2, we have two nested loops which might indicate that Graham scan is running in $O(n^2)$. However, the inner while loop does not actually run n times for each point in the input set. Once a point is known to be in inside the convex hull, the algorithm does not consider it in computation again.

Therefore, each point in the input set is visited at most twice, once on the left turn and once on a right turn. Hence, the loops run in $O(n)$ time, implying that the time complexity of the sorting procedure outweighs the loop and so, the time complexity of Graham scan is $O(n \log n)$.

4.3 Quickhull Algorithm

The Quickhull algorithm makes use of the divide-and-conquer approach similar to that of quicksort. It was published in 1990 by Johnathan Greenfield.[9] It finds the convex hull using triangular expansion – constructing triangles and then concatenating them to obtain a larger structure. Quickhull can be extended to $\mathbb{R}^d$, however, we will only discuss the planar case here. The working of the algorithm can be visualized here. We also present more formal pseudocode for Quickhull in Algorithms 3 and 4. The algorithm can be broken into a few simple steps:

1. Find the rightmost and the leftmost points since they will always be in the convex hull.
2. Use the line segment (baseline) created by the two points to divide the input set into two subsets of points which will be processed recursively.
3. Determine a point on one side of the line that is the furthest from the line. This point will form a triangle with the baseline.
4. The points inside the triangle cannot be in the convex hull and are therefore, discarded.
5. Repeat steps 3 and 4 on the two new lines that are formed by the triangle.
6. Keep doing this until no more points are left, recursion has come to an end. The selected points constitute the convex hull.



(a) Step 1-2 (b) Step 3-5 (c) Step 6

Figure 7: The Quickhull procedure.

Algorithm 3: QUICKHULL

Input: A point set P

Output: The convex hull of P, S

let p_0 be left-most point in P
let p_1 be right-most point in P
Segment p_0 to p_1 divides our input points into two sets S_1 and S_2
$S \leftarrow p_0$
$S \leftarrow p_1$
$S \leftarrow$ FINDHULL(S_1 , p_0, p_1)
$S \leftarrow$ FINDHUl(S_2 , p_1, p_0)
return S

Algorithm 4: FINDHULL

Input: S_k,p_a,p_b

Output: The convex hull of S

if S_k is empty **then**
  return p_a,p_b
Find the point p_c with the greatest perpendicular distance from the line p_a,p_b
$S_1 \leftarrow$ points above p_c and p_a
$S_2 \leftarrow$ points above p_c and p_b
$S \leftarrow$ FINDHULL(S_1 , p_0, p_1)
$S \leftarrow$ FINDHULL(S_2 , p_1, p_0) **return** S

4.3.1 Time Complexity of Quickhull

The analysis of the time complexity for Quickhull is not as straightforward as it is for other algorithms since the time complexity can vary based on the distribution of the points with respect to the line that is drawn in the initial step. However, we are certain that finding the extreme points (farthest towards the left or right) takes $O(n)$ time since it is a simple search. After finding the initial baseline, recursion is performed. There are generally two cases we should consider when analyzing Quickhull.[9]

***Case* 1: every point is in the convex hull** — This is the case where $h = O(n)$. This is the worst case for all convex hull algorithms since all input points must be output. We look at two sub-cases:

- *The worst subcase:* This is the worst possible case for the algorithm. This would require $O(n)$ calls to QUICKHULL, each of which would needs an $O(n)$ computation. Therefore, the time complexity would be $O(n^2)$.
- *The best expected subcase:* Assuming that the partitioning of the points would be reasonably even, we can come up with the following recursion:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

We can easily solve the recurrence using the master theorem to obtain a time complexity of $O(n \log n)$.

***Case* 2: uniformly distributed points** — This is the expected case where the number of points on the convex hull h are significantly less than the number of input points n. There are two sub-cases to consider in this scenario:

1. *The worst subcase:* The points of the convex hull are distributed in such a way the partitioning of the points will almost always result in one empty call. In this case we will have $O(h)$ calls to QuickHull, and each of those calls would require no more than $O(n)$ computation. Therefore, the time complexity would be $O(nh)$.
2. *The best (expected) subcase:* We assume that most of the internal points are removed in the initial stages of the algorithm due to points being distributed uniformly across the region. Then after the initial stages, only $O(h)$ points are remaining.

   After the early stages, the algorithm would run in $O(h \log h)$ due to divide-and-conquer. The early stages of the algorithm – running in $O(n)$ – combined with the later stages would yield the time complexity $O(n + h \log h)$.

5 Lower Bound on the Convex Hull Problem

We have seen some algorithms that output results in $O(nh)$, $O(n \log n)$ and $O(n \log h)$ time in Table 1. We have also explored how these computational complexities are derived in the context of the three algorithms we discussed in Section 4. A natural question that should arise is that can we do better than $O(\log n)$?

The convex hull problem outputs a convex polygon which is essentially a cyclic enumeration of the boundary vertices of the convex hull. Therefore, in a way, to find the convex hull of a point set – we would need to sort the vertices of the hull. We already know that the lower bound for sorting n numbers is $\Omega(n \log n)$. We will use this intuition about sorting to deduce a formal lower bound for the convex hull problem.[15]

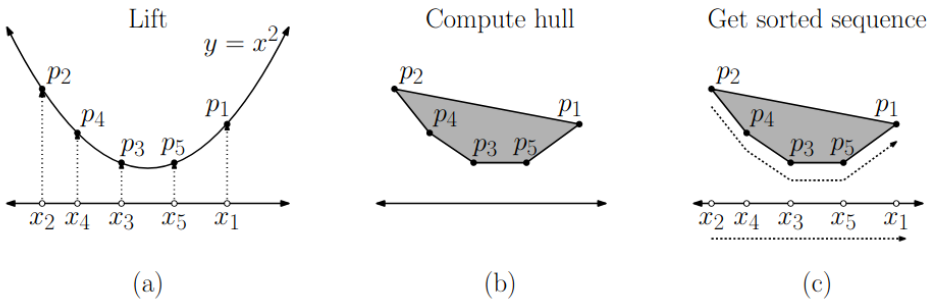
Theorem 1. Lower Bound for Convex Hull

Assuming computations based on comparisons, any algorithm for the convex hull problem requires $\Omega(n \log n)$ time in the worst case.

Proof 1.1

We will show that we can reduce the sorting problem to the convex hull problem in $O(n)$ time. This would imply that for any $O(f(n))$ algorithm for the convex hull problem implies an $O(n + f(n))$ algorithm for sorting. It is obvious that $f(n)$ cannot be smaller than $\Omega(n \log n)$, otherwise it would lead to a contradiction.

We perform this reduction by projecting the input points onto a convex curve. Let $X = \{x_1, \ldots, x_n\}$ be the n values that we want to sort. We "lift" these points onto a parabola $y = x^2$, by mapping x_i to the point $p_i = (x_i, x_i^2)$. Call the resulting set of points P, as shown in the diagram below.[12]



We can see that all points of P lie on the convex curve, and the sorted order of the points along the lower hull are the same as the sorted order of X. Obtaining the lower hull vertices can be done trivially in $O(n)$ time, and then traversing the boundary will produce the sorted order of X.

We can see this reduction can be performed in linear time, which means that in the general case, the convex hull of n points cannot be computed more quickly than sorting, i.e. the lower bound for the convex hull problem is $\Omega(n \log n)$. ■

5.1 Output Sensitivity

The complexity of finding a convex hull as a function of the input size n is lower bounded by $\Omega(n \log n)$. However, for some convex hull algorithms, we have seen that we can characterize their complexity in terms of both the input size n and the output size h – the number of points on the hull. These algorithms are called output-sensitive algorithms and can be asymptotically more efficient than $\Theta(n \log n)$ if $h = o(n)$ – h is significantly smaller than n.

We saw that the Jarvis March algorithm in Section 4.1 had a time complexity of $O(nh)$. The lower bound on the worst-case running time of output-sensitive convex hull algorithms was proven to be $\Omega(n \log h)$ in the 2-dimensional case. We also saw some algorithms in Table 1 that achieve this time complexity.

The earliest such algorithm was introduced by Kirkpatrick and Seidel in 1986 which used a technique that the authors called "marriage-before-conquest". Since

the algorithm is so complicated it is not used in practice. In 1996, Timothy Chan published the Chan's algorithm which is also an optimal output-sensitive algorithm with a time complexity of $O(n \log h)$. It is a simpler algorithm compared to the Kirkpatrick-Seidel algorithm. Chan's algorithm combines Jarvis march with the Graham scan algorithm and it also naturally extends to 3-space.

5.2 Akl-Toussaint Heuristic

A simple heuristic is often used as the first step in implementations of convex hull algorithms so that their performance is improved. The heuristic is based on an algorithm developed by Selim Akl and G. T. Toussaint in 1978. The approach is to quickly exclude many points that would not have been part of the hull anyway, thereby reducing the actual input size for the algorithm. [6]

The first step to implement this heuristic is to find the two points with the lowest and highest x-coordinates and the two points with the highest and lowest y-coordinates. This is an $O(n)$ operation. These four points form a convex quadrilateral Q. All the points that lie inside Q are not part of the convex hull. Finding the points that lie inside Q is an $O(n)$ operation, therefore, the entire operation is $O(n)$.

An additional optional step could be to add the points with the smallest and largest sums of x and y coordinates to the quadrilateral – turning it into an irregular convex octagon and then discarding the points contained in it. In many cases, this preprocessing step can make a convex hull algorithm run in linear expected time, even when the complexity could be quadratic.[14]

6 Convex Hulls in 3-Space

So far we have looked at three algorithms that found the convex hull – Jarvis march, Graham scan, and Quickhull. However, we were only computing the convex hull in $\mathbb{R}^2$, i.e. the planar convex hull. What would the convex hull look like if we were in a 3-dimensional space? If we are computing the convex hull of a finite set of points in 3-space, we get a convex polyhedron as shown in Figure 8

Figure 8: A 3-dimensional convex hull (polyhedron)

The polyhedron is a 3-dimensional polytope. When computing the convex hull in higher dimensions, we consider the *facets* of a polytope instead of the line segments that we would consider in the planar case. We state a few definitions that will be important in understanding the higher dimensional convex hull problem.

Definition 8 (Simplex)**.** A *simplex* is the generalization of the triangle or tetrahedron to an arbitrary number of dimensions. d-simplex is used to denote a simplex in $\mathbb{R}^d$.

Definition 9 (d-polytope)**.** A *polytope* is the generalization of a polyhedron to $\mathbb{R}^d$. A polytope has "flat" sides, which means that the sides – called *facets* of a d-polytope are $d-1$-polytopes. A polyope is called *simplicial* if all the facets of a *polytope* are simplices (see Definition 8).

Definition 10 (Affine combination)**.** Given k distinct points $p_1, p_2, \ldots, p_k$ in $\mathbb{R}^d$, the set of points

$$p = \alpha_1 p_1 + \alpha_2 p_2 + \cdots \alpha_k p_k \qquad (\alpha_j \in \mathbb{R},\ \ \alpha_1 + \alpha_2 + \cdots + \alpha_k = 1)$$

is the *affine set* generated by $p_1, \ldots, p_k$ and p is the *affine combination* of $p_1, \ldots, p_k$.

Definition 11 (Affine hull)**.** Given a subset L of $\mathbb{R}^d$, the *affine hull* $\text{aff}(L)$ of L is the smallest affine set containing L.

The 3-dimensional convex hull problem is of enormous relevance since it has a lot of applications in computer graphics, pattern recognition and many more. This problem is also in $\Theta(n \log n)$, however, the proof of this differs from the planar case and is outside the scope of this paper. Several algorithms can compute the convex

hull in 3 or more dimensions. Some of these include the gift-wrapping algorithm, d-dimensional Quickhull, the beneath-beyond method and Chan's algorithm.

6.1 The Gift-Wrapping Algorithm

The gift-wrapping method is among the earliest algorithms to compute the convex hull in 3 dimensions. It is the method upon which Jarvis march is based. The gift-wrapping method was proposed by Chand and Kapur in 1970[4], and analyzed by Bhattacharya in 1982. The idea is to go from one facet to an adjacent facet akin to how one wraps a sheet around an object, hence the name. We will look at the d-dimensional case of the gift-wrapping method, however, our explanation will mostly rely on the 3-dimensional case due to ease of visualization.

The algorithm rests on the assumption that the resulting polytope would be simplicial (see Definition 9). In this case, each subfacet e is shared by exactly two facets F_1 and F_2. We state the algorithm below in Algorithm 5.

Algorithm 5: GIFT-WRAPPING

Input: $(p_1, \ldots, p_n)$
Output: The convex hull F

$Q \leftarrow \emptyset$, $\mathscr{T} \leftarrow \emptyset$
$F \leftarrow$ find an initial convex hull facet
$\mathscr{T} \leftarrow$ subfacets of F
$Q \leftarrow F$
while $Q \neq \emptyset$ **do**
    $F \leftarrow Q$ \\ extract front element from queue
    $T \leftarrow$ subfacets of F
    foreach $e \in T \cap \mathscr{T}$ **do**
        $F' \leftarrow$ facet sharing e with F \\ gift-wrapping
        insert into $\mathscr{T}$ all subfacets of F' not yet present and delete all those already present
        $Q \leftarrow F'$
    return F

6.1.1 Analysis of the Gift-Wrapping Algorithm

The algorithm starts with finding a facet that already lies on the convex hull. The idea is to obtain a hyperplane containing a facet using successive approximations, i.e. by creating d supporting hyperplanes. After d iterations, the hyperplane π_d would contain a $(d-1)$-face of the convex hull. The construction of the initial face is done using iterative vector operations and can be done in $O(nd^2)$.

The gift-wrapping mechanism to move from a facet F to a facet F' comprises of seeking among all hyperplanes determined by subfacet e (shared between F and F') and a point of input set S not in F, the hyperplane that forms the largest angle with aff(F).

The step in line 7 generates subfacets of F. Each facet is determined by d vertices, and each subset of $(d-1)$ vertices of F determine a subfacet. Therefore, this step can be computed easily in $O(d^2)$.

In step 8, we check whether a subfacet e is a candidate for gift-wrapping. A subfacet is a candidate if it is contained in only one facet generated by the algorithm. We keep a store $\mathscr{T}$ of such subfacets. Since the subfacets also contain subfacets within them of lower dimensions all the way to a point, they can be stored in a height balanced binary tree. Searches can be done in this data structure in $O(d \log M)$ where M is size of the data structure.

We now estimate the overall complexity of the algorithm. The initialization process has a complexity $O(nd^2)$. We use φ_{d-1} and φ_{d-2} to represent the actual number of facets and subfacets of the polytope. Steps 6, 11 and 12 can be done in $O(d) \cdot \varphi_{d-1}$ and step 7 can be done in $O(d) \cdot \varphi_{d-2}$. The tree search to check for candidate solutions can be done in $O(d \cdot \log \varphi_{d-2}) \cdot \varphi_{d-2}$. The complexity of the gift-wrapping method would be $O(d^3) + O(nd)$. The maximum values for φ_{d-1} and φ_{d-2} are $O(n^{\lfloor d/2 \rfloor})$. Therefore, we reach the conclusion:

Theorem 2. *The convex hull of a set of N points in $\mathbb{R}^d$ can be constructed using the gift-wrapping methods in time $T(d,n) = O(n \cdot \varphi_{d-1}) + O(\varphi_{d-2} \cdot \log \varphi_{d_2})$. Therefore, $T(d,n) = O(n^{\lfloor d/2 \rfloor + 1}) + O(n^{\lfloor d/2 \rfloor} \log n)$.*

For a more detailed version of this proof along with more definitions, please see [12]. In the 3-dimensional case, this process is simpler and the complexity turns out be lower as well. The gift-wrapping method itself is $O(n)$ and the overall complexity is $O(n \cdot F)$ where F is the number of facets on the hull.

7 Conclusion

This paper sought to provide an overview of the paradigm geometric algorithms with a focus on the convex hull. To that end, we presented a summary of the convex hull problem along with some important geometric definitions and three algorithms for the planar case – Jarvis march, Graham scan and Quickhull – and the gift-wrapping algorithm for the higher dimensional case. There are a lot of algorithms that we have left out which employ different strategies to find efficient solutions to the convex hull problem such as Chan's algorithm and more. We encourage the reader to explore these algorithms in [1, 3].

Furthermore, we explored the algorithm design process for each of these geometric algorithms along with simple analyses of their time complexities. We did not include proofs of correctness for the algorithm for the sake of brevity, and we encourage the more mathematically inclined readers to explore proofs of correctness and time complexities in the courses cited.

References

[1] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. "The quickhull algorithm for convex hulls". en. In: *ACM Transactions on Mathematical Software* 22.4 (Dec. 1996), pp. 469–483. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/235815.235821. URL: https://dl.acm.org/doi/10.1145/235815.235821 (visited on 05/31/2021).

[2] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational geometry: algorithms and applications*. en. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. ISBN: 9783540779735 9783540779742. DOI: 10.1007/978-3-540-77974-2. URL: http://link.springer.com/10.1007/978-3-540-77974-2 (visited on 05/31/2021).

[3] T. M. Chan. "Optimal Output-Sensitive Convex Hull Algorithms in Two and Three Dimensions". In: *Discrete Comput. Geom.* 16.4 (Apr. 1996), pp. 361–368. ISSN: 0179-5376. DOI: 10.1007/BF02712873. URL: https://doi.org/10.1007/BF02712873.

[4] Donald R. Chand and Sham S. Kapur. "An Algorithm for Convex Polytopes". In: *J. ACM* 17.1 (Jan. 1970), pp. 78–86. ISSN: 0004-5411. DOI: 10.1145/321556.321564. URL: https://doi.org/10.1145/321556.321564.

[5] Thomas H. Cormen and Thomas H. Cormen, eds. *Introduction to algorithms*. 2nd ed. Cambridge, Mass: MIT Press, 2001. ISBN: 9780262032933.

[6] Luc Devroye and Godfried T. Toussaint. "A note on linear expected time algorithms for finding convex hulls". In: *Computing* 26.4 (1981), pp. 361–366. DOI: 10.1007/BF02237955. URL: https://doi.org/10.1007/BF02237955.

[7] Howard Eves. *A survey of geometry*. eng. Rev. ed., 2. print. OCLC: 256176456. Boston, Mass.: Allyn and Bacon, 1980. ISBN: 9780205032266.

[8] R.L. Graham. "An efficient algorith for determining the convex hull of a finite planar set". en. In: *Information Processing Letters* 1.4 (June 1972), pp. 132–133. ISSN: 00200190. DOI: 10.1016/0020-0190(72)90045-2. URL: https://linkinghub.elsevier.com/retrieve/pii/0020019072900452 (visited on 05/31/2021).

[9] Jonathan Greenfield. *A Proof for a QuickHull Algorithm*. Tech. rep. 65. 1990. URL: https://surface.syr.edu/cgi/viewcontent.cgi?article=1058&context=eecs_techreports.

[10] R.A. Jarvis. "On the identification of the convex hull of a finite set of points in the plane". en. In: *Information Processing Letters* 2.1 (Mar. 1973), pp. 18–21. ISSN: 00200190. DOI: 10.1016/0020-0190(73)90020-3. URL: https://linkinghub.elsevier.com/retrieve/pii/0020019073900203 (visited on 05/31/2021).

[11] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 2017. ISBN: 0262534770.

[12] Franco P. Preparata and Michael Ian Shamos. *Computational geometry*. New York, NY: Springer New York, 1985. ISBN: 9781461270102 9781461210986. DOI: 10.1007/978-1-4612-1098-6. URL: http://link.springer.com/10.1007/978-1-4612-1098-6 (visited on 05/29/2021).

[13] Michael Ian Shamos. "Geometric Complexity". In: *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*. STOC '75. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1975, pp. 224–233. ISBN: 9781450374194. DOI: 10.1145/800116.803772. URL: https://doi.org/10.1145/800116.803772.

[14] Jean Souviron. *Convex hull: Incremental variations on the Akl-Toussaint heuristics Simple, optimal and space-saving convex hull algorithms*. 2013. arXiv: 1304.2676 [cs.CG].

[15] Andrew Chi-Chih Yao. "A Lower Bound to Finding Convex Hulls". In: *J. ACM* 28.4 (Oct. 1981), pp. 780–787. ISSN: 0004-5411. DOI: 10.1145/322276.322289. URL: https://doi.org/10.1145/322276.322289.