Report on Convex Hull

Sabal Subedi
Master's in Computer Science
Idaho State University
Pocatello, ID, 83209 USA
sabalsubedi@isu.edu

February 10, 2024

Abstract

In this report, I will summarize the use of divide and conquer algorithm to obtain the convex hull of a set of points. I will give a pseudocode and analyze the asymptotic notation.

1 Introduction

The convex hull of a set of points is defined as the smallest convex polygon, that encloses all of the points in the set. Convex means that the polygon has no corner that is bent inwards.
The divide and conquer algorithm is a strategy of solving a large problem by breaking the problem into smaller sub-problems, solving the sub-problems and finally combining them to get the desired output. To implement this algorithm, we use recursion.
Here, we use divide and conquer algorithm to compute the convex hull of a set of points. First, I divide the set of points into equals halves recursively and then merge those halves using the logics that gives the final convex hull of the given set of points.

2 Pseudocode and Asymptotic Analysis

2.1 Pseudocode to compute convex hull

Algorithm 1 Convex Hull

```
function CONVEXHULL(points)
    if len(points) <= 3 then
        return points
    mid = len(points)//2
    left_hull ←CONVEXHULL(points[: mid])
    right_hull ←CONVEXHULL(points[mid :])
    right_most_left_hull ← maximum x-coordinate in left hull
    left_most_right_hull ← minimum x-coordinate in right hull
    left_decreasing = right_decreasing = False
    left_point, right_point = right_most_left_hull, left_most_right_hull
    slope ← SLOPE(right_most_left_hull, left_most_right_hull)
    while left_decreasing = False or right_decreasing = False do
        left_decreasing = right_decreasing = True
        while left_decreasing = True do
            left_point ← COUNTER_CLOCKWISE(left_hull, left_point)
            new_slope ← SLOPE(left_point, right_point)
            if next_slope < slope then
                slope = new_slope
            else
                left_point ← CLOCKWISE(left_hull, left_point)
                left_decreasing = False
            if SLOPE(left_point, CLOCKWISE(right_hull, right_point)) ≤
    slope then Break
```

```
        while right_decreasing = True do
            right_point ← CLOCKWISE(right_hull, right_point)
            new_slope ← SLOPE(left_point, right_point)
            if next_slope > slope then
                slope = new_slope
            else
                right_point ← COUNTER_CLOCKWISE(right_hull, right_point)
                right_decreasing = False
            if SLOPE(COUNTER_CLOCKWISE(lelft_hull, left_point), right_point ) ≥ slope then Break
    top_left_tan_point, top_left_tan_point = left_point, right_point
    left_decreasing = right_decreasing = False
    left_point, right_point = right_most_left_hull, left_most_right_hull
    slope ← SLOPE(right_most_left_hull, left_most_right_hull)
    while left_decreasing = False or right_decreasing = False do
        left_decreasing = right_decreasing = True
        while left_decreasing = True do
            left_point ← CLOCKWISE(left_hull, left_point)
            new_slope ← SLOPE(left_point, right_point)
            if next_slope > slope then
                slope = new_slope
            else
                left_point ← COUNTER_CLOCKWISE(left_hull, left_point)
                left_decreasing = False
            if SLOPE(left_point, COUNTER_CLOCKWISE(right_hull, right_point)) ≥ slope then Break
        while right_decreasing = True do
            right_point ← COUNTER_CLOCKWISE(right_hull, right_point)
            new_slope ← SLOPE(left_point, right_point)
            if next_slope < slope then
                slope = new_slope
            else
                right_point ← CLOCKWISE(right_hull, right_point)
                right_decreasing = False
            if SLOPE(CLOCKWISE(lelft_hull, left_point), right_point ) ≤ slope then Break
```

```
    lower_left_tan_point = left_point
    lower_right_tan_point = right_point
    final_hull = []
    right_hull_point = top_right_tan_point
    while right_hull_point ≠ lower_right_tan_point do
        final_hull.APPEND(right_hull_point)
        right_hull_point ← CLOCKWISE(right_hull, right_point)
    final_hull.APPEND(lower_right_tan_point)
    left_hull_point = lower_left_tan_point
    while left_hull_point ≠ top_left_tan_point do
        final_hull.APPEND(left_hull_point)
        left_hull_point ← CLOCKWISE(left_hull, left_point)
    final_hull.APPEND(lower_left_tan_point)
    return final_hull
```

Time complexity and space complexity
Given,
points: a list of tuples(x,y)

Space Complexity:

1. Since the depth of recusion for this algorithm is $O(\log n)$ space.
2. To store variables like left_hull, right_hull, final_hull is atmost the size of input. So, it takes $O(n)$ space.

$$S(n) = O(\log n) + O(n) \approx O(n) \tag{1}$$

Time Complexity:

1. The recursive division of a set of points into halves talkes $T(n) = 2T(n/2) + O(n)$. Using Master Theorem, we get $O(n \log n)$
2. To find the left hull and right hull, i.e. dividing the points at each recursion, takes $O(n)$
3. To find the upper tangent, the while loop runs $O(n)$ times
4. To find the lower tangent, the while loop runs $O(n)$ times

5. Finally, to merge the hull points it takes $O(n)$ time

$$\begin{aligned} T(n) &= O(n \log n) + O(n) + O(n) + O(n) + O(n) \\ &\approx O(n * \log n) \end{aligned} \tag{2}$$

Analysis of algorithm:
The pseudocode takes set of points and outputs a single set. It uses divide and conquer algorithm to divide the given set of points into two halves recursively. Finally, it merges the hull points to get the final convex hull.

2.2 Pseudocode to get next point based on clockwise order

Algorithm 2 Get points on Clockwise order

function CLOCKWISE($points, point$)
    $index \leftarrow$ index of the point in points
    if $index = len(points) - 1$ **then**
        return $points[0]$
    else
        return $points[index + 1]$

Time complexity and space complexity
Given,
points: a set of points
point: a valid tuple(x,y)
Space Complexity:

$$S(n) = O(1) + O(1) + O(1) = O(3) \approx O(1) \tag{3}$$

Time Complexity:

$$\begin{aligned} S(n) &= O(1) + O(1) + O(1) + O(1) + O(1) + O(1) \\ &= O(6) \approx O(1) \end{aligned} \tag{4}$$

Analysis of algorithm:
The pseudocode takes two inputs: a set of points and a valid tuple(x,y) and outputs a tuple(x,y). It returns next point in clockwise order of given point in the set of points.

2.3 Pseudocode to get next point based on counterclockwise order

Algorithm 3 Get points on Counter-Clockwise order

```
function COUNTER_CLOCKWISE(points, point)
    index ← index of the point in points
    if index = 0 then
        return points[len(points) − 1]
    else
        return points[index − 1]
```

Time complexity and space complexity
Given,
points: a set of points
point: a valid tuple(x,y)
Space Complexity:

$$S(n) = O(1) + O(1) + O(1) = O(3) \approx O(1) \tag{5}$$

Time Complexity:

$$\begin{aligned} S(n) &= O(1) + O(1) + O(1) + O(1) + O(1) + O(1) \\ &= O(6) \approx O(1) \end{aligned} \tag{6}$$

Analysis of algorithm:
The pseudocode takes two inputs: a set of points and a valid tuple(x,y) and outputs a tuple(x,y). It returns next point in counter-clockwise order of given point in the set of points.

2.4 Pseudocode to compute slope

Algorithm 4 Slope

```
function SLOPE(point1, point2)
    x1, y1, x2, y2 = point1[0], point1[1], point2[0], point2[1]
    return ((point2[1] − point1[1])/(point2[0] − point1[0]))
```

Time complexity and space complexity
Given,
point1: a valid tuple(x,y)
point2: a valid tuple(x,y)
Space Complexity:

$$S(n) = O(1) + O(1) + O(1) + O(1) + O(1) = O(5) \approx O(1) \tag{7}$$

Time Complexity:

$$T(n) = O(1) + O(1) \approx O(1) \tag{8}$$

Analysis of algorithm:
The pseudocode takes two points as input and outputs a single value. It computes the slope between two points and return the result.

3 Observations and Results

3.1 Observation Table

Sample Size	Distribution	Time
10	Gaussian	0.000 sec
100	Gaussian	0.001 sec
1000	Gaussian	0.004 sec
10000	Gaussian	0.056 sec
100000	Gaussian	0.667 sec
500000	Gaussian	2.983 sec
1000000	Gaussian	5.502 sec

Here, I have chosen Gaussian distribution to collect the elapsed time (i.e. time taken to get the convex hull) of seven different sample size. The elapsed time increases as the size of the sample increases.
To give more insight on this, I have collected 5 different samples for each sample size and get the data to compute the mean time needed for each sample size. Using the obtained data, I have plotted a graph shown in figure 1.

.

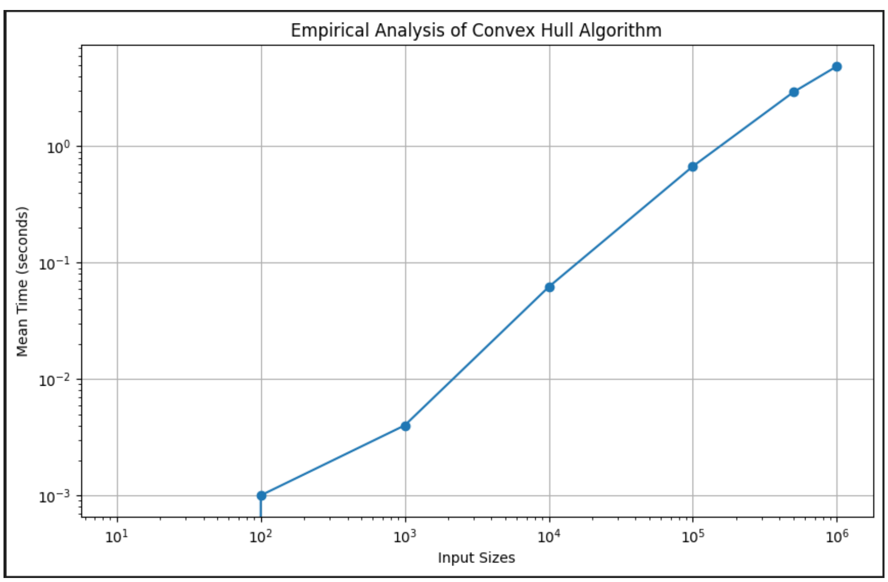

Figure 1: sample size vs mean time required

In the graph (in figure 1), we can see a increasing curve i.e. the mean time required to compute the convex hull with smaller sample size takes less time and so on. The change in size and the mean time follows logarithmic pattern. This mean that the changes either increasing or decreasing in sample size changes the mean time by increasing or decreasing rate.
According to my analysis on time complexity of CONVEXHULL algorithm, it takes $O(n * \log n)$ time. But, my graph (empirical) analysis differs from the theoretical analysis. The graph looks identical with $1/3(n * \log n)$. Therefore, the constant of proportionality(k) is $1/3$ so that $CH(Q) = K * g(n)$.
Due the presence of constant of proportionality value of $1/3$, the empiricial curve is progressing consisitently lower than expected theoretical curve i.e the curve continues to widens as we plot more data.

Reasons for the difference between theoretical and empirical analysis:

1. The hardware capacity of the machine can be a reason.
2. The sparseness of the data can be a valid reason.
3. Theoretical analysis focus on algorithm efficiency and estimates its performance based on input size and its growth. However, this can differ on empirical analysis. Implementation details, hardware limitations, etc can influence the empirical analysis.

3.2 Results

.

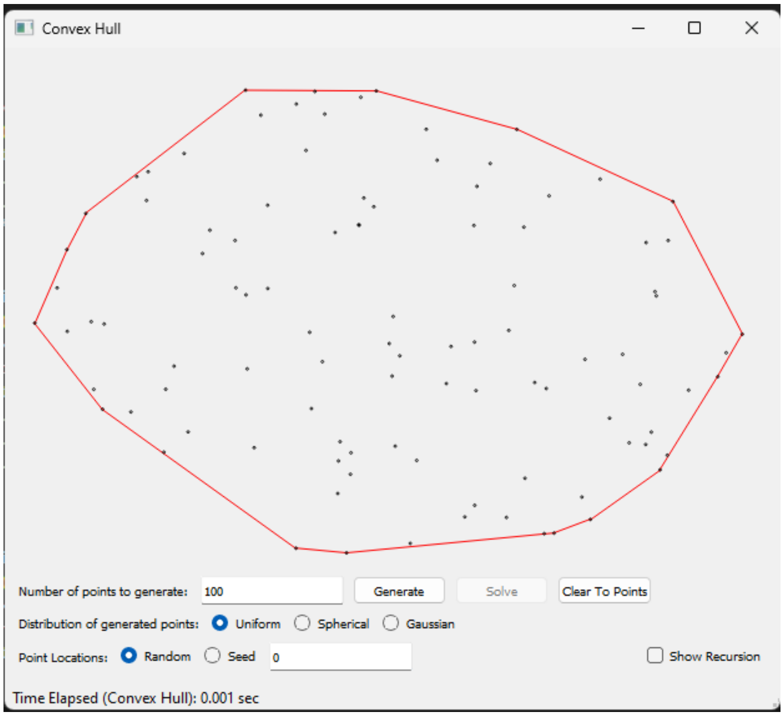

Figure 2: Convex hull for 100 points

I used 100 sample size (in figure 2) to compute hull. As expected the program was able to compute and display convex hull. The program took 0.001 second to compute the convex hull under Uniform distribution and random seed.

Again, I used 1000 random points to test the program under same setup and the program was able to generate the convex hull as expected. It took 0.010 seconds. The change in elpased time (required time) is significant as the sample size increases.

.

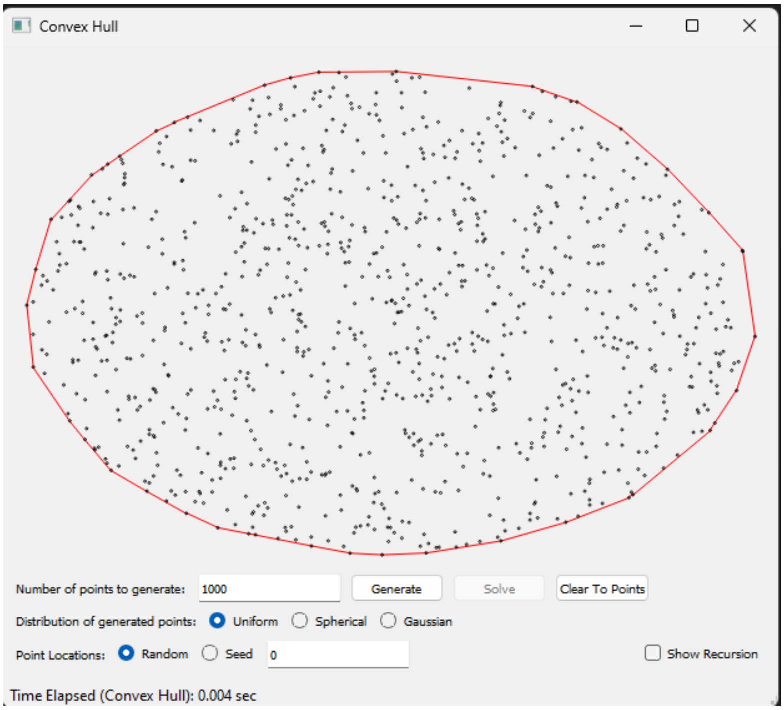

Figure 3: Convex hull for 1000 points

4 Conclusion

In summary, I was able to compute a convex hull using divide and conquer algorithm along with orientation and distance. I was able to analyze the pseudocode and get the time and space complexity.