Convex Hull Problem:
A comparison of Jarvis March and Graham Scan Algorithms

Alyssa Tamayo

CS584 Spring 2019

Contents:

Introduction to the Convex Hull Problem

The Convex Hull problem is by far the most interesting problem in computational geometry. I say this not only from personal preference but because of the influence convex hull has on solving real world problems. According to Levitin in textbook titled *Introduction to Design & Analysis of Algorithms*, "...in computer animation, replacing objects by their convex hulls speeds up collision detection; the same idea is used in path planning for Mars mission rovers."[1] Excuse my colloquialisms for a moment but, how amazingly cool is that?! So, what is the convex hull problem? The convex hull problem is described as, given a set of points, calculate the points that lay on the convex hull. A formal definition of the convex hull itself is also described in Levitin as "The convex hull is a set *Q* of points, denoted by CH(Q), is the smallest convex polygon *P* for which each point in *Q* is either on the boundary of *P* or in its interior."[2] See example of convex hull below.[3]

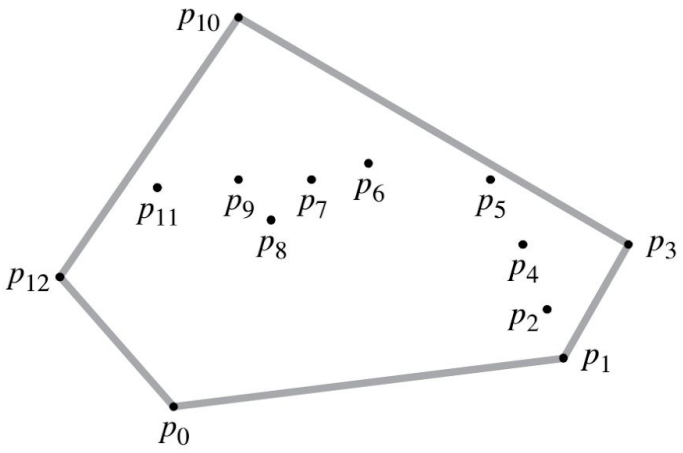


Figure 33.6 A set of points $Q = \{p_0, p_1, \ldots, p_{12}\}$ with its convex hull $\mathrm{CH}(Q)$ in gray.

Of the many algorithms that exist to solve this problem I've decided to implement, test and analyze two algorithms; Jarvis March and Graham Scan. I've chosen to compare these algorithms because I felt that the results would be interestingly different considering their complexities; Jarvis March ϵ *O(nh)*, *h* being the number of points on the hull, and Graham Scan ϵ *O(nlogn)*. I'm interested in how the two algorithms have different dependencies on the points on

[1] A. Levitin, *Introduction to the design and analysis of algorithms*, 3rd ed. Boston: Pearson, 2012, p. 110.
[2] T. Cormen, *Introduction to algorithms*, 3rd ed. Cambridge, Mass.: MIT Press, 2009, p. 1029.
[3] T. Cormen, *Introduction to algorithms*, 3rd ed. Cambridge, Mass.: MIT Press, 2009, p. 1029.

the hull and I'll talk more in depth about how these algorithms solve the convex hull problem in later sections.

I was excited to work with the following algorithms because the convex hull problem also serves as a creative outlet as I intend to demonstrate with this project. As a graphic design enthusiast I've realized that convex hull algorithms, along with a lot of other computational geometry algorithms, are used widely in creative applications that I use regularly such as Adobe Photoshop and the rest of Adobe's Creative Suite. To answer the question on how I decided on my topic, it was definitely a personally motivated choice and one I thought I would have the most fun learning more in depth about.

Introduction to Processing

As described on Processing's Wikipedia page, Processing is an open source graphical library and IDE.[4] I wanted to take this opportunity to re-learn Processing since applying my technical knowledge to creative endeavors was my real motivation in pursuing a graduate degree in computer science. I've been a hobbyist of the programming library since I was an undergraduate student studying saxophone performance in 2009. Back then, I used Processing to manipulate digital projections with musical input live from my saxophone. I also used Processing to build a vegetable piano and had the opportunity to perform my new instrument on stage[5]. Processing was initially introduced to the world as a library used with Java but since then the creators have adapted Processing to allow users to write code in Python. Considering how much I would need to code for this final project I decided to go with a scripting language and a language I enjoy the most writing in - Python.

Jarvis March Algorithm

The Jarvis March Algorithm, also known as the gift wrapping algorithm, was the easiest to implement. The algorithm is described below and is referenced from my study group's homework submission.

1. Initialize an empty list to store points on the hull
2. Find the lowest point in the set of points and add to the hull
3. For every point on the hull, compare to every point in the set

[4] "Processing (programming language)", *En.wikipedia.org*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Processing_(programming_language). [Accessed: 25- May- 2019].
[5] "8", *YouTube*, 2013. [Online]. Available: https://youtu.be/QZ6xPK16_d4.[Accessed: 25- May- 2019].

a. Pick an arbitrary point p_next on the list to be up for next on the hull
b. If there is a point p that results in a left turn or is collinear from p_next, reassign the p_next on the hull to p.
c. Once all points have been compared, if p_next is not the first point on the hull, add this point on the hull and repeat step 3

The toughest part about implementing this algorithm is revisiting the geometric formulas needed in helper functions to determine if a point results in a left turn. In the helper function is_left_turn() as part of my *Utils* module, I had to make a decision on how to handle collinear points. I've decided to let the algorithm pick the furthest collinear point to lessen the amount of points that will be on the hull. This decision on how to handle collinear points will ultimately help the performance of Jarvis March since it's complexity is heavily reliant on the total amount of points that are on the hull. This is due to the condition in the loop which iterates as long as there is a point on the hull we have yet to make comparisons to. See snippet of my Jarvis March implementation below.

```
def do_Jarvis(point_list):
    """
    Jarvis March method to solve Convex Hull
    :param point_list: set of Utils.Points
    :return: hull_list: list of Utils.Points on hull
    """
    hull_list = [Utils.get_lowest_point(point_list)]

    for hull_point in hull_list:
        next_point = point_list[0]
        for p in point_list:
            if Utils.is_left_turn(hull_point, p, next_point):
                next_point = p
        if next_point != hull_list[0]:
            hull_list.append(next_point)
    return hull_list
```

The complexity of Jarvis March is *O(n*h)* where h is the number of points on the hull.[6] The best case performance will be produced with minimal points on the hull. Later we'll see Jarvis March tested on random point sets where the points on the hull will vary, and deterministic point sets where there will only be 3 points on the hull or if every point is on the hull in a circle.

[6][2]"Gift wrapping algorithm", *En.wikipedia.org*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Gift_wrapping_algorithm. [Accessed: 25- May- 2019].

Graham Scan Algorithm

Graham Scan is an algorithm in which it's performance isn't reliant on the number of points on the hull. It first sorts the points in order of smallest polar angle in which *O(nlogn)* dominates the resulting complexity. The pseudocode, as referenced from the Graham Scan Wikipedia page[7], is as follows:

```
1. Initialize an empty stack to store points on hull
2. Find the lowest point p0 in the set of points and push to stack
3. Sort all of the points in order of smallest to largest polar angle from
   p0
4. For all p in sorted points
      a. While the addition of p results in a right turn from the top 2
         points in the hull, pop from hull stack
      b. Push p to hull stack
```

My implementation of Graham Scan utilizes the same `is_left_turn()` function as introduced in Jarvis March above. In addition, I've written another helper function, sort_polar which computes the polar angles and returns a sorted list in smallest polar angle to largest.

Test Cases

In this final project submission I have included a test suite which tests the algorithms with various points sets and various list lengths. To run the test suite you can type the following into the command prompt `py test_suite.py`. It's important to note that python3 is needed to execute the program. The test suite will then export the results of the tests into a csv file. The tests included in the suite can be described as follows:

```
1. A deterministic set of points where only 3 points are on the hull and all
   other points are inside of the hull spanning from list sizes 6,000 to
   15,000.
2. A deterministic set of points where all points are on the hull in a
   circle spanning from list sizes 1 to 3,000.
```

[7] [1]"Graham scan", *En.wikipedia.org*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Graham_scan. [Accessed: 25- May- 2019].

3. A randomly distributed set of points spanning from list sizes 1,000 to 11,000.

It was important to include point sets 1 and 2 because Jarvis March's performance is heavily dependant on the total points that make it on the hull. In the best case scenario with minimal points on the hull and the points within the hull is a total more than *logn,* Jarvis March will perform faster than Graham Scan. However, when all the points are on the hull, Graham Scan is the clear winner. Once I present the test results that were a product of the Test Suite, I'll take a closer look how the two algorithm performed with random data sets. In this case, it is unknown how many points are on the hull until the hull is created.

In order to rely on the data that will be presented in the following sections, I needed to ensure that the methods creating the point sets were accurately creating sets with either minimum points on the hull or maximum points on the hull. Along with the Test Suite, I've also provided `ImplementationTest.py` which tests the correctness of the utility methods. These tests will also check that each algorithm is returning the expected number of hull points depending on the data set. This is accomplished by creating various sized input lists, calling the algorithms on these sets and checking the length of the convex hull list to the expected total number of points, keeping track of any errors.

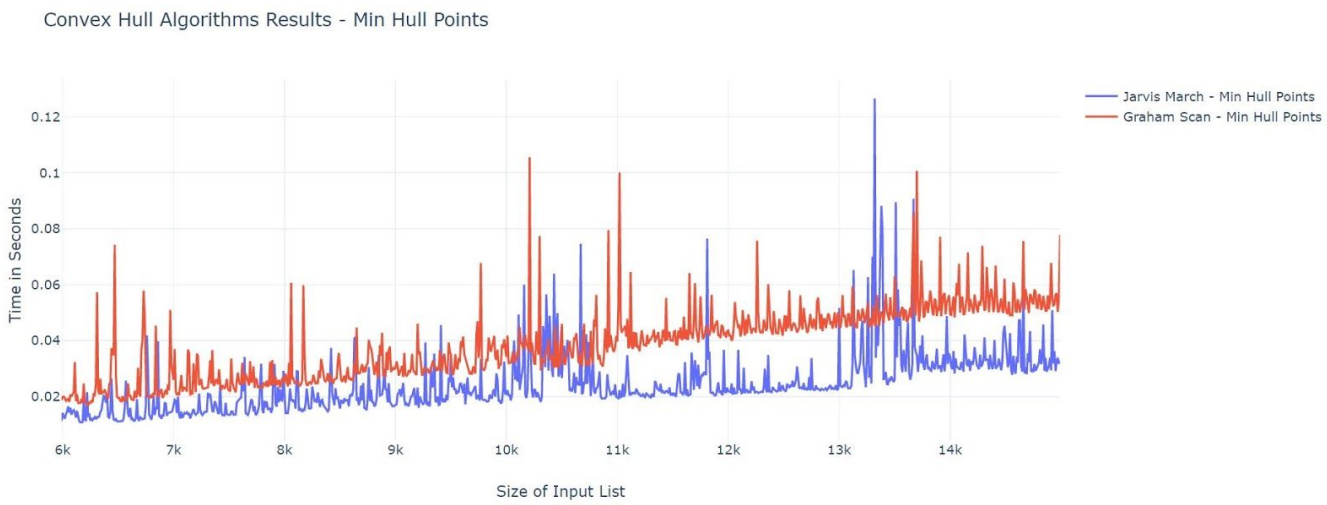
As a sanity check, run `python3 ImplementationTest.py` before proceeding to run the Test Suite. You should get the following message:

```
SUCCESS: Min Hull List resulted in correct # of hull points using Jarvis
SUCCESS: Max Hull List resulted in correct # of hull points using Jarvis
SUCCESS: Min Hull List resulted in correct # of hull points using Graham
SUCCESS: Max Hull List resulted in correct # of hull points using Graham
```

Repository Information

The following section will now present and analyze the data received from running the full Test Suite. Full results tables in which the graphs reference it's data has been omitted in interest of space in this document. For the full table of results and interactive Plot.ly line graphs please visit the project repository aly_tomato.github.com/ConvexHullProject. CSV files are uploaded containing all of the data and the README will contain links to the interactive Plot.ly line graphs. If reading this paper as a PDF, the images links are clickable.

Test Results - Min Hull Points Set



(Click above graph image to link to interactive Plot.ly graph)

Above are the results of both algorithms being run on set of points where exactly three points lie on the hull. Of the three test cases (min hull points, max hull points, and random points), this test case required a much larger data set in order to see significant differences between the two algorithms. This was surprisingly unexpected considering Graham Scan's complexity of *O(nlogn)* suggested a much quicker growth as long as *logn* is greater than *3.* Now that I have been given the opportunity to analyze Jarvis March and Graham Scan I would like to re-answer a questions from homework 2, "Give an example input on which Jarvis March will perform significantly better than Graham's scan and explain why it will perform better". I would first need to define *significantly better* as when Jarvis March performs 100% faster than Graham Scan. According to the min hull points results table below, this happens when the input size is at about 11,100 and the expected total of hull points is 3. Taking the average of a relatively small sample set, Jarvis March performed at about 0.02 seconds and Graham Scan performed at about 0.04 seconds.

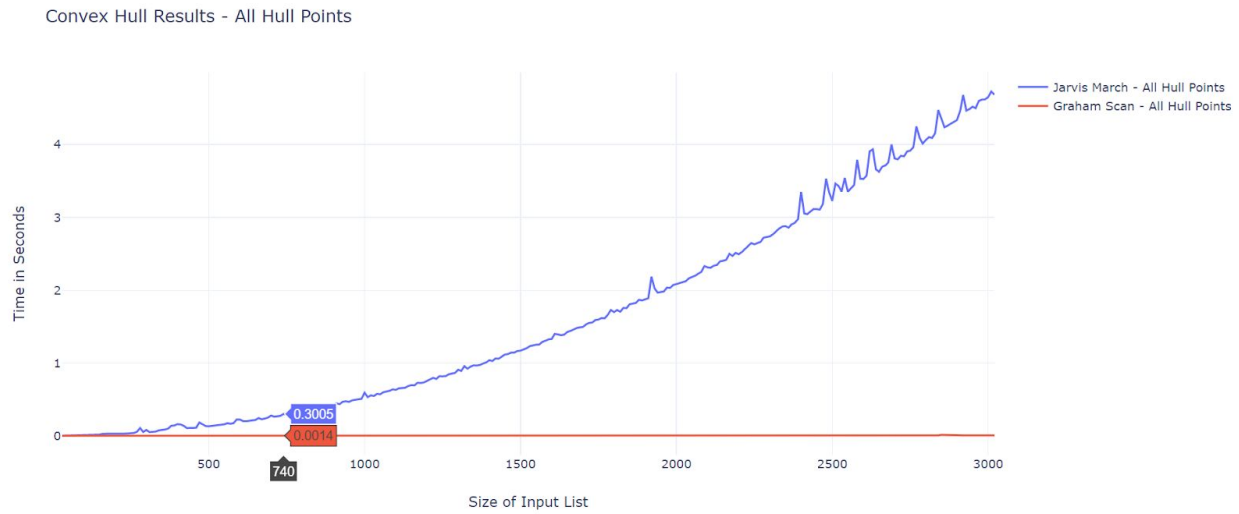
Between input size of 13,000 to 14,000 you can see examples of when Jarvis March will spike above Graham Scan. In fact, the longest run in this test is from Jarvis March at above 0.12 seconds. To be honest I don't have much of an explanation for this other than my implementation of creating a set of points where there is exactly three points on the hull has a non-deterministic efficiency.

	Input Size	Time in Seconds			Input Size	Time in Seconds
Jarvis Min Hull Points	11100	0.0274		**Graham Min Hull Points**	11100	0.0373
	11110	0.0204			11110	0.0379
	11120	0.0215			11120	0.0645
	11130	0.0209			11130	0.0382
	11140	0.0214			11140	0.037
	11150	0.0214			11150	0.0432
	11160	0.0209			11160	0.0378
	11170	0.0228			11170	0.0429
	11180	0.0221			11180	0.0375
	11190	0.0209			11190	0.0378
	11200	0.021			11200	0.0362
	11210	0.0202			11210	0.0401
	11220	0.0202			11220	0.0437
	11230	0.0208			11230	0.041
	11240	0.0201			11240	0.0382
	11250	0.0198			11250	0.0389
	11260	0.0213			11260	0.0379
	11270	0.0207			11270	0.0421
	11280	0.0206			11280	0.0388
	11290	0.0199			11290	0.0402
	11300	0.0197			11300	0.0403
	Avg. Time	**0.02114286**			**Avg. Time**	**0.040548**

(Click the above table image link for all Data CSVs for this project)

Test Results - All Points on Hull Points

Unlike the the results for min hull points in the previous section, I was not surprised by the results for all points on hull. It was expected that Jarvis March would perform significantly slower than Graham Scan considering Jarvis March benefits from having the least amount of points on the hull to be most efficient. In the graph below you can see that it doesn't take a large input size like the previous test to start seeing performance differences between the two algorithms.

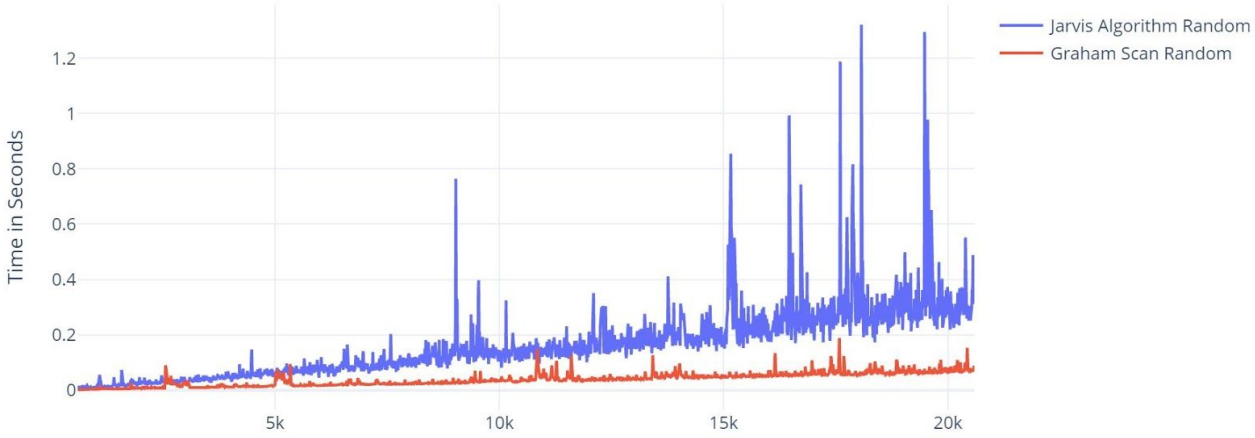


(Click above graph image to link to interactive Plot.ly graph)

At input size of 740 you can already see Jarvis March completing execution 215% slower than Graham Scan. If there is a high likelihood that the input set of points will have a maximum of points on the hull, perhaps shapes with curves, I would recommend to select Graham Scan as the algorithm of choice.

Test Results - Random Points Set

To create the random set of points, I utilized the random library as part of the python standard libraries. Below is an overview of the output of this test.



(Click above graph image to link to interactive Plot.ly graph)

Above you will see a graph containing the timed results of both algorithms performing on a random set of points. The number of hull points are unknown for each test and are randomly generated. It's interesting to see that Jarvis March is not the ideal algorithm to use with a random set of points. While Graham Scan will have peaks here and there for example at around an input size of 2,000, 5,000 and 11,000, they fail in comparison to the deviation and commonality of peaks that occur with the Jarvis March algorithm.

Results Summary

After looking at the results of the three test cases, I would conclude that Graham Scan is the overall best choice when selecting between the two algorithms studied in this project. Graham Scan excelled in performance in both random hull points set and max hull points set. In order to see much of a difference between Jarvis March and Graham Scan in the min points on hull test case we needed to have a much larger set of points. Even then the difference in time

were mere milliseconds off from each other. If there were a certainty that all points will be on the hull and the data sets were significantly greater than 11,000, Jarvis March would be the ideal algorithm. However, I believe this certainty is less likely and while we could be fairly certain that a shape will have minimal points, the benefits of Jarvis March are quite underwhelming when you look at Graham Scan's results for the test case involving random number of points on the hull or the test case where all points are on the hull.

Introduction to the Point Generator

Note that you'll need to have Java 7 in order for this application to work properly. It was easy to spend a lot of time using Processing to create a program that could help visualize and *artify* the convex hull problem. In all honesty, it turned to be less of a creative tool and more of a debugging tool especially as I ran into issues with implementing the algorithms. I used the Point Generator to see if and when my implementation of Jarvis March and Graham Scan had flaws by having the Point Generator randomly draw points on the screen and watching my implementations of the algorithms return expected shapes.

The Point Generator stores randomly generated coordinates for points in a list to then be passed into the Graham Scan algorithm. I've decided on Graham Scan because the results from the random points set tests suggested that Graham Scan performs better on average compared to Jarvis March. Once the algorithm returns back a set of points that lie on the hull, the Point Generator will then draw lines between the points to reveal the shape of the convex hull. Once complete, the hull drawing will rapidly cycle through pastel colors.

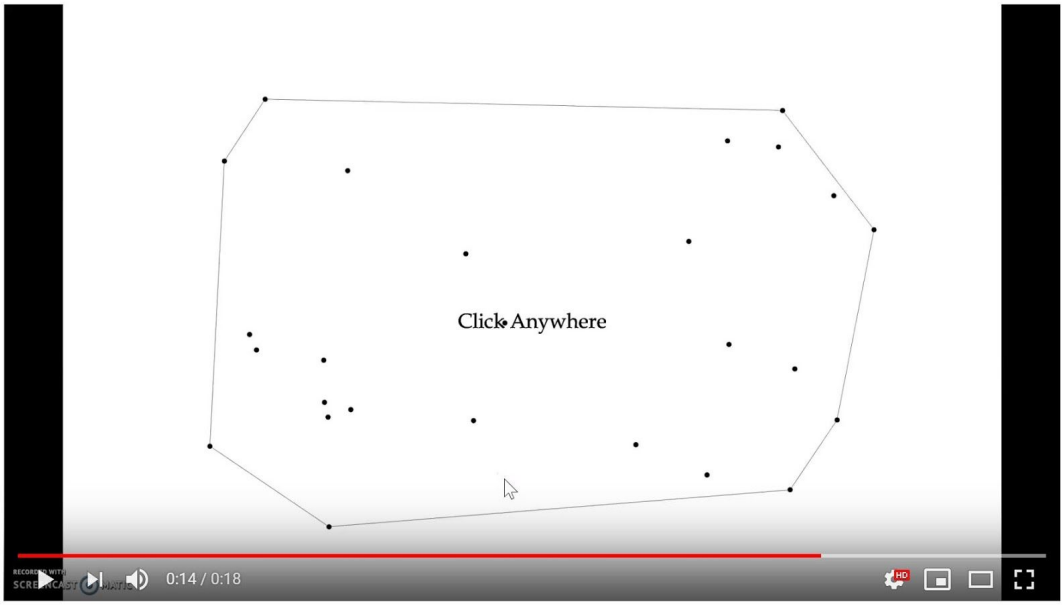
Ideally I would want to manipulate the shape in ways that could really utilize the power of Processing but I quickly found that there is a lot more math that I would have needed to brush up on in order to accomplish as such. Simply trying to inflate or deflate the shape took me be back to a geometry class I took over 13 years ago and unfortunately needed to prioritize my time towards writing and analyzing the Test Suite and results. Regardless, I'm happy with how the animations turned out. Iit was a fun experiment and I'm excited to utilize Processing in future side projects.

Using the Point Generator

Instructions for using the Point Generator is as follows:

1. In the Point Generator Applications folder, find the executable suitable to your machine

2. Watch as points are randomly generated on your screen
3. Click anywhere on the screen when you are ready for point generation to stop and to apply Graham Scan on the set of points
4. Once the shape is complete, the program will *artify* the shape
5. When you're ready for salvation, simply click the escape key on your keyboard.



Point Generator - Convex Hull - Processing

(Click the above image link to be directed to a youtube video demonstrating this process.)