Convex Hull Algorithms

CS584 – Spring 2019 – Final Project

By Chad Tolleson

Contents

Convex Hull Preview

A convex hull is the smallest convex polygon containing a given set of points on a plane. Each vertex of the convex hull represents the point at the extreme edge of the set of points in a particular direction. Connecting each of these vertexes creates a boundary, or hull that entirely encompasses all of the points on the plane. A convex hull is commonly described using the following analogy:

- Start a wooden board with a bunch of nails driven into it.
- Grab a rubber band,
- stretch the rubber band around all of the nails and let go.
- The rubber band now encompasses all of the nails,
- thus creating a convex hull.
- Nails touching the rubber band are the vertexes along the convex hull.

Levitin provides the following explanation as to why convex hull is such an important concept: "Several such applications are based on the fact that convex hulls provide convenient approximations of object shapes and data sets given. For example, 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. Convex hulls are used in computing accessibility maps produced from satellite images by Geographic Information Systems. They are also used for detecting outliers by some statistical techniques. An efficient algorithm for computing a diameter of a set of points, which is the largest distance between two of the points, needs the set's convex hull to find the largest distance between two of its extreme points (see below). Finally, convex hulls are important for solving many optimization problems, because their extreme points provide a limited set of solution candidates." [1]

For the purposes of this paper, I stick to the 2-Dimensional plane. I will use the terms vertex and point interchangeably. Sets of data will consist of n points $P = \{p_1, p_2, p_3 \dots\}$, where each $p_i = (x_i, y_i)$ and $x_i, y_i \in \mathbb{R}$. H consists of the h vertexes on the convex hull.

For a set of n points, any of the 2^n subsets of points may be the vertexes of the convex hull. The order in which the vertexes are located on the convex hull is important. [2] Using a Brute Force approach to finding the convex hull is very inefficient, as we will see. So several clever algorithms were developed to find a convex hull much quicker.

I will discuss and assess four different algorithms for solving the convex hull problem:

1) Brute Force, expected performance $\mathrm{O}(n^3)$
2) Graham's Scan, expected performance $\mathrm{O}(n \lg n)$
3) Jarvis's March, expected performance $\mathrm{O}(n * h)$
4) Chan's Algorithm, expected performance $\mathrm{O}(n \lg h)$

For each algorithm, I will provide a brief description of how it works, some concise pseudo code, the expected and observed performance, and the full code (in separate files). To test each algorithm, I will run it on several different sample data sets (every algorithm will be given the same samples) and track the total running time. To account for variations in the environment (OS processes, garbage collecting, etc.) I will run the algorithm on multiple servers at different times of day and compare the results. To ensure garbage collection is not a factor, I did the coding in C and placed the garbage collecting activities outside of the statements that end the operation timing. Finally, I will chart the results of each algorithm's performance for an easier representation of the relative performance.

Brute Force

How it Works

One brute force approach to solving the convex hull problem consists of the following steps:

1) Take an input of P points.
2) For every combination of two points, p_1,$p_2 \in P$, calculate the line defined by the points as $L_{p_1p_2} = \{(p_1, p_2): ax + by = c\}$, where:
   a. $a = y_2 - y_1$,
   b. $b = x_1 - x_2$, and
   c. $c = x_1y_2 - y_1x_2$.
3) L then divides the plane into two halves, defined as:
   a. $ax + by - c < 0$ $(lesser)$
   b. $ax + by - c > 0$ $(greater)$
   c. $ax + by - c = 0$ $(point\ is\ on\ the\ line)$
4) For every point $p_i \neq p_1, p_2$ determine if the point is "lesser", "greater", or "on" $L_{p_1p_2}$.
5) If all p_i are to one side of, or on $L_{p_1p_2}$ then points p_1, p_2 are on the convex hull, so add the points to H.

Pseudo-Code

```
BRUTE_CH($P, n$)
1   $for\ i\ from\ 0\ to\ (n-2)$
2       $for\ j\ from\ (i+1)\ to\ (n-1)$
3           $left := 0,\ right := 0,\ on := 0$
4           $a := y_j - y_i$
5           $b := x_i - x_j$
6           $c := x_iy_j - y_ix_j$
7           $for\ k\ from\ 0\ to\ (n-1)$
8               $if\ p_k \neq p_i, p_j$
9                   $if\ ax_k + by_k - c < 0$
10                      $lesser += 1$
11                  $if\ ax_k + by_k - c > 0$
```

```
                        $greater += 1$
                    $if\ ax_k + by_k - c = 0$
                        $on += 1$
        $if\ (lesser = 0)\ or\ (greater = 0)$
            $add\ p_1, p_2\ to\ H$
    $return$
```

Expected Performance

This approach must calculate the "sidedness" of each point (n) for the line defined for every other combination of two points $(n) * (n)$. Thus, the time complexity of this algorithm is $O(n^3)$.

Graham's Scan

History

This algorithm is named for Ronald Graham who developed and published the algorithm while at Bell Telephone Laboratories, Inc. in 1972. [3] This Ronald Graham person is interesting; he has accomplished far more than this algorithm in his career. I would suggest at least taking a quick look at his Wikipedia page. [4]

How it Works

Graham's Scan uses a stack S to keep track of the points in P. When the algorithm is finished, S contains only the vertexes comprising H.

The algorithm first sorts all of the points in P in increasing order by y-coordinate, and then x-coordinate. It selects the point with the lowest value (primarily y-coordinate, and then x-coordinate to break a tie) as the starting point p_0. Then, for all other points in P, the algorithm calculates the polar angle in relation to p_0. These remaining points are sorted by their polar angles, in increasing order. If there are multiple points along the same polar angle, all but the point furthest away from p_0 are removed from the analysis because they could not possibly be on the convex hull.

The algorithm now has the information it needs to assess each point to determine whether it falls on the convex hull. It does this by assessing three points at a time (utilizing special functions called TOP and NEXT-TO-TOP that allows it to peak into S), where the first point is the most recently added point on H, to determine if the angle they create $(\angle p_0 p_i p_{i+1})$ is clockwise or counterclockwise. If the angle is clockwise (right hand turn), then the middle point (p_i) could not possibly be on the convex hull. Therefore, the algorithm pops that point off S and tries the next series of three points. The algorithm continues in this manner until it reaches p_0 again. When the algorithm reaches p_0 again, all of the remaining points in S are the vertexes on H. Some elements of the implemented code were inspired by pseudocode and code sample found online. [5]

Pseudo-Code

```
GRAHAM_CH(P, n) [2]
1   let $p_0$ be the point in $P$ with the minimum y-coordinate,
        or the leftmost such point in case of a tie
2   let $\langle p_1, p_2, \dots, p_n \rangle$ be the remaining points in $P$,
        sorted by polar angle in counterclockwise order around $p_0$
        (if more than one point has the same angle, remove all but
        the one that is farthest from $p_0$)
3   let $S$ be an empty stack containing the vertexes on the hull
4   PUSH($p_o$, $S$)
5   PUSH($P_1$, $S$)
6   PUSH($P_2$, $S$)
7   for $i$ from 3 to $n - 1$
8       while the angle formed by points NEXT-TO-TOP($S$), TOP($S$),
            and $p_i$ makes a nonleft turn
9           POP($S$)
10      PUSH($p_i$, $S$)
11  return $S$
```

Expected Performance

Finding the point with the lowest y-coordinate requires inspecting every element, so the time complexity is $\mathrm{O}(n)$. The second main function of the algorithm is to calculate the polar angle of each point in relation to the starting point. This takes $\mathrm{O}(n)$ as well. The algorithm then sorts all of the points in increasing order by their polar angle. I use $Quicksort$, so the sorting takes $\mathrm{O}(n \lg n)$. Finally the algorithm does the actual scanning of all points to find those on the hull via the clockwise/counterclockwise analysis. This function takes time of $(2 * n) \in \mathrm{O}(n)$.

Graham's Scan is interesting in that the upper bound for time complexity is established not on the "Graham scanning" itself, rather the pre-sorting of points by their polar angle in relation to the lowest point, p_0. Since we know that sorting, as a class of functions, has a best case lower bound of $\Omega(n \lg n)$, by application we know that Graham's Scan is $\in \mathrm{O}(n \lg n)$.

Jarvis's March

History

Jarvis's March is named for R.A. Jarvis, who published it in the journal "Information Processing Letters" in March 1973. There is not much more information readily available about R.A. Jarvis, unfortunately. [6]

How it Works

The Jarvis March algorithm takes a set of n points, P, on a plane as input. The algorithm works by first finding the point in the set with the smallest x-coordinate. This point is noted as the starting point (p_0) in the search for the convex hull H. Next, the algorithm looks at three consecutive points (p, q, r) in P, using p_0 as the first value of p and the following two points in P as points q, r, respectively. For each set of three points, the algorithm calculates whether the angle $\angle pqr$ is "clockwise" or "counterclockwise" using the cross-product of the vectors $\overrightarrow{pq}, \overrightarrow{qr}$. If an angle is found to be clockwise, then q is updated to r, and the search resumes with the remaining points in P. If no clockwise angles are found from a particular $\overrightarrow{pq}$, then q is added to H (p would, by design, already be a part of H). The algorithm halts when it works its way back to p_0. Parts of the actual implementation code are derived from in-class coding exercises in the Winter 2019 term [7]

Pseudo-Code

```
JARVIS_CH($P$,$n$) [8]
    let $p$ = leftmost point in $P$, let $H$ hold the vertexes on the hull
    $i$ = 0
    repeat (
        $H[i] = p$
        $r = P[0]$     // initial endpoint for a candidate edge on the hull
        for $j$ from 1 to $n$
            $q = P[j]$
            if either ($r == p$) or ($q$ is on right of line between $p$ to $r$)
                $r = q$   // found greater left turn, update endpoint
        $i = i + 1$
        $p = r$
        )
    until $r == H[0]$     // wrapped around to first hull point
    return $H$
```

Expected Performance

Jarvis's March looks at each of the n items in P once for every vertex already placed on the hull H. As such, the worst case running time for the algorithm is **$O(n * h)$.**

It is worth noting that Jarvis's March is ideally used when analyzing a large number of points surrounded by relatively fewer hull vertexes. I like to imagine a triangular perimeter around a dense cluster of points… like three sentinel llamas positioned around a herd of blue ribbon winning alpacas. [9] On the other hand, the worst-case scenario for Jarvis's March would be a situation where all n points are located on the hull H. For this scenario imagine a hundred electric fence posts arranged in a near perfect circle around the center-pivot-irrigated-hay-field [10] in which the alpacas often graze. For this scenario the algorithm's performance is still $O(n * h)$, but since $h = n$, the actual performance is really $O(n^2)$.

Chan’s Algorithm

History

The above two algorithms are cool, but a kid named Timothy M. Chan combined the two to create his own ice cold algorithm, appropriately named Chan’s algorithm. He published the related paper in “Discrete & Computational Geometry” in 1996 [11], when Chan was a fresh 20 years of age! And this is just one incredible achievement early in his impressive career. [12]

How it Works

Chan’s algorithm is ingenious in that it leverages Graham’s Scan to find the sub-convex-hulls $H_1 H_2, \dots H_k$ of multiple (let’s say the count is K) sub-sets $P_1, P_2, \dots P_K \subset P$ of m points first, then combines the vertexes of those sub-hulls into a new population of points, P', and finally uses Jarvis’s March to find the convex hull H' of P', which is also the convex hull H of the original set of points P. What a ride! While I am doing all my calculations within the 2D space, Chan’s Algorithm is naturally extendable into 3D space too. Since Graham’s Scan and Jarvis’s March are discussed in more detail above, I will not delve back into them here. As we will see below, the divide & conquer nature of the Graham’s Scan application, combined with the final Jarvis’s March allows Chan’s Algorithm to perform in $O(n \log h)$ time. [13] [14]

Pseudo-Code

```
CHAN_CH($P, n, m$)
1    let $K = n \div m$, let $H$ hold the vertexes on the hull, let $h$ be the number of hull vertexes
2    divide $P$ into $K$ subsets of $m$ items
3    for $i$ from 1 to $K$
4        GRAHAM_CH($P_i, m$) // returns $H_i$ & $h_i$
5        add $H_i$ to $P'$
6        add $h_i$ to $n'$
7    $H$ = JARVIS_CH($P', n'$)
8    return $H$
```

Expected Performance

Splitting the original set of points P into K sub-sets $P_1, P_2, \dots P_K$ takes $O(n)$ time. Then finding the sub-hull H_i of each sub-set P_i using Graham’s Scan takes $O(m \lg m)$. Since we are finding the sub-hull for K sub-sets, the total cost of these operations is $O(K * m \lg m) = O(n \lg m)$. For the Jarvis’s March phase of the algorithm, we know that the “outer loop” of the algorithm only runs once per vertex on the hull, or h times. The inner part of the loop is typically bounded by the number of points n in the set P, but in this scenario we are looking only at the vertexes of each sub-hull H_i. As such, the cost of the inner loop is estimated by the number of sub-sets times the reduced number of points in each subset, so approximately $O(K) \; or \; O\left(\frac{n}{m}\right) * O(\lg m) = O\left(\frac{n}{m} \lg m\right)$. Since we are doing this h times (for the outer loop), the combined cost of the Jarvis’s March portion of the algorithm is $O(h * \left(\frac{n}{m}\right) \lg m)$. Then when $m \approx h$, the cost becomes $O(h * \left(\frac{n}{h}\right) \lg h) = O(n \lg h)$.

Summary of Findings

I coded each of the four algorithms using C. Then I ran each algorithm on eight different data samples: two I already had on hand, five were randomly generated, and one is the largest of the randomly generated sets modified specifically to show that Jarvis's March is more optimal for data sets with few vertexes on the hull in relation to total number of points. Jarvis gotta flex. All of the code and sample data are included separately.

Some specific noteworthy items are:

- Brute Force, taking $O(n^3)$ is simply too costly to test on sample sizes beyond 1,000 points. Therefore, I did not; I set a rule within the code to prevent the Brute Force algorithm from working on any data sets larger than 1,000 points. Funnily enough, Brute Force actually beat Graham's Scan in the most trivial of data sets.
- Generally speaking, given random data we can rank the performance of the algorithms both in terms of expected time complexity and actual performance as follows:

1)	Chan's Algorithm	$O(n \lg h)$
2)	Graham's Scan	$O(n \lg n)$
3)	Jarvis's March	$O(n * h)$
4)	Brute Force	$O(n^3)$

- As expected, the Jarvis's March absolutely killed the competition when it came to the scenario with few vertexes on the hull relative to total points in the set… the llamas are in position, those alpacas can rest easy tonight! On the other hand, when we analyzed far fewer points (1,000 vs 10,000,000) Jarvis's March and Chan's Algorithm were markedly less efficient that Graham's Scan.
- While Chan's Algorithm does indeed outperform the other three in most scenarios, we do not really see the dramatically reduced actual time cost that we would expect. This is likely due to some combination of implementation choices and the processing-time overhead required to handle the multiple function calls within Chan's Algorithm.
- As expected, the gap in total processing time grows noticeably between the different algorithms as the input size grows. If you were to imagine much larger data sets, and compounded with repeating processes (i.e. tasks performed every hour or minute), these differences in processing time would be immense. This certainly supports the notion that picking the appropriate algorithm for the problem has a meaningful impact on the results.

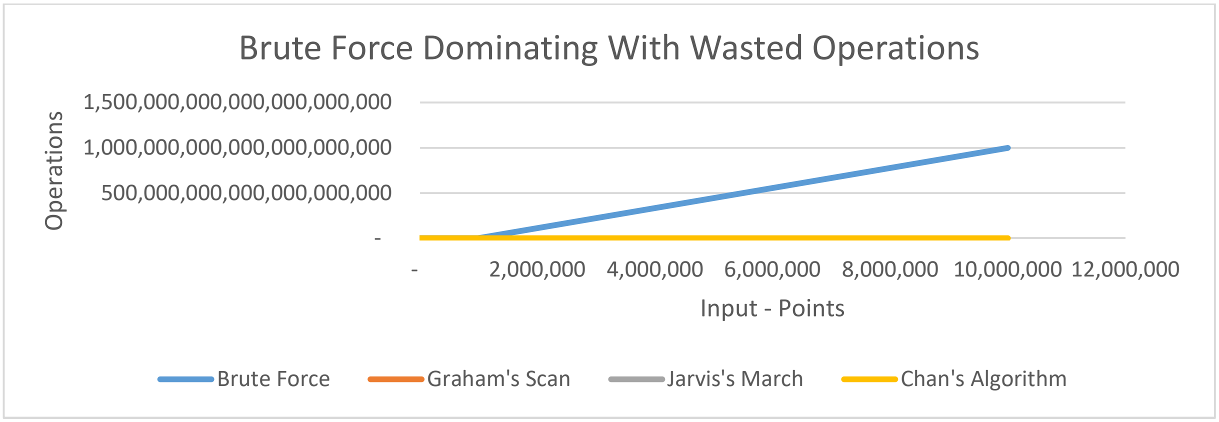
This table summarizes the maximum number of operations expect to see from each algorithm based strictly off the number of input points (n) and number of vertexes on the hull (h).

Sample Size *n*	Points on Hull *h*	Brute Force $O(n^3)$	Graham's Scan $O(n \lg n)$	Jarvis's March $O(n * h)$	Chan's Algorithm $O(n \lg h)$
16	**5**	**4,096**	**64**	**80**	**37.15**
1,000	19	1.00E+09	9,966	19,000	4,248
1,000	**979**	**1.00E+09**	**9,966**	**979,000**	**9,935**
10,000	25	1.00E+12	132,877	250,000	46,439
100,000	**32**	**1.00E+15**	**1,660,964**	**3,200,000**	**500,000**
1,000,000	39	1.00E+18	19,931,569	39,000,000	5,285,402
10,000,000	**43**	**1.00E+21**	**232,534,967**	**430,000,000**	**54,262,648**
10,000,003	3	1.00E+21	232,535,041	30,000,009	15,849,630

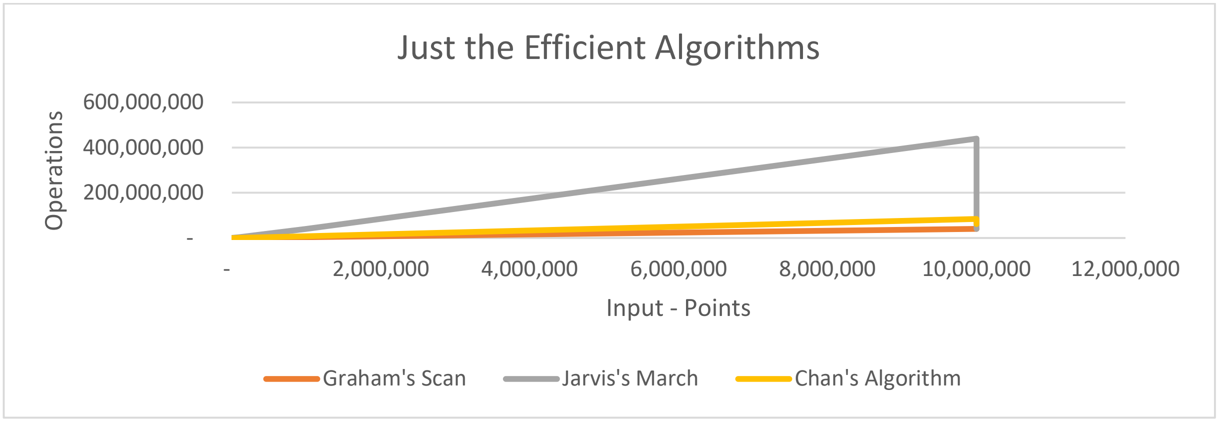
The following table and graphs depict the performance of the four algorithms across the eight different data sets.

Sample Size	Metric	Brute Force	Graham's Scan	Jarvis's March	Chan's Algorithm
16	# points on hull	5	5	5	5
16	# of operations	1,680	58	101	130
16	time in seconds	0.000027	0.000045	0.000006	0.000011
1,000	# points on hull	19	19	19	19
1,000	# of operations	498,501,000	3,980	20,019	7,235
1,000	time in seconds	5.917939	0.000378	0.000303	0.000419
1,000	# points on hull	981	979	979	979
1,000	# of operations	498,501,000	3,020	980,979	979,135
1,000	time in seconds	5.298564	0.000306	0.014139	0.015023
10,000	# points on hull	-	25	25	25
10,000	# of operations	-	39,974	260,025	74,701
10,000	time in seconds	-	0.003737	0.003919	0.003826
100,000	# points on hull	-	32	32	32
100,000	# of operations	-	399,967	3,300,032	787,081
100,000	time in seconds	-	0.043248	0.047031	0.0394
1,000,000	# points on hull	-	39	39	39
1,000,000	# of operations	-	3,999,960	40,000,039	8,251,292
1,000,000	time in seconds	-	0.502699	0.639264	0.446002
10,000,000	# points on hull	-	43	43	43
10,000,000	# of operations	-	39,999,956	440,000,043	84,841,088
10,000,000	time in seconds	-	6.534802	7.739181	6.466264
10,000,003	# points on hull	-	3	3	3
10,000,003	# of operations	-	40,000,008	40,000,015	62,222,054
10,000,003	time in seconds	-	7.116797	0.892684	4.075923

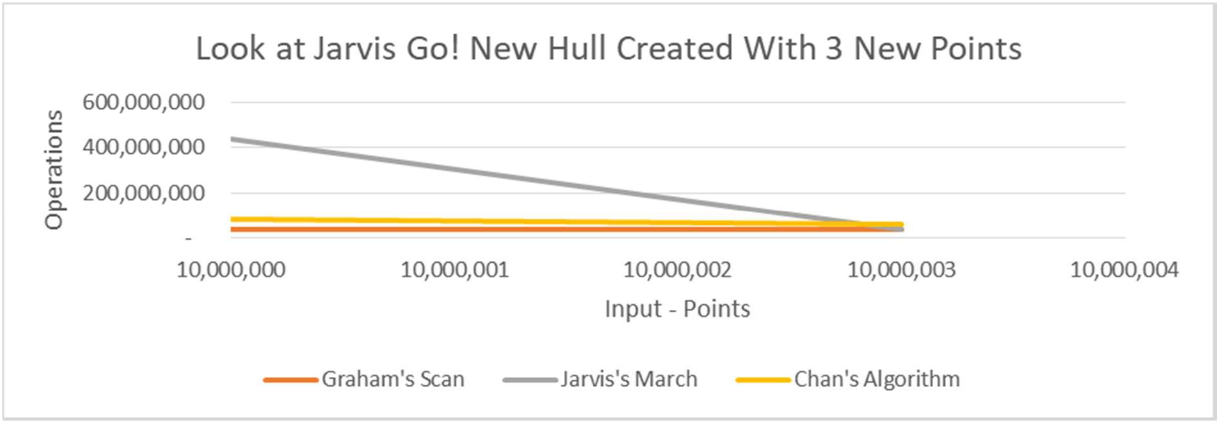
The following graph shows how inefficient Brute Force is compared to the other algorithms, it will be excluded from the remaining graphs.



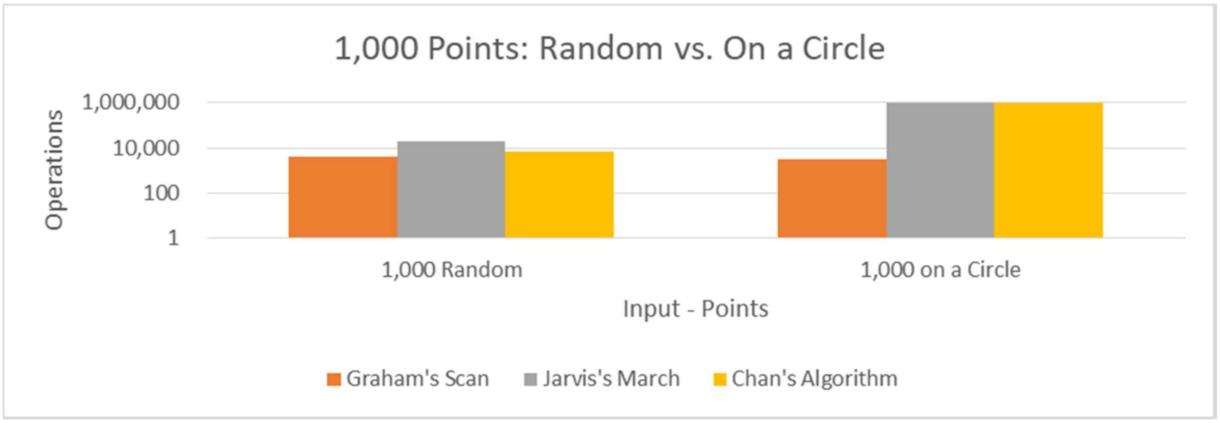
The following graph shows how, for a quasi-random points, Jarvis's March becomes increasingly inefficient in relation to both Graham's Scan and Chan's Algorithm.



The following graph shows how adding just three points to create a new hull entirely enclosing the original points dramatically increases the speed of Jarvis's March (and Chan's Algorithm, but to a much less impressive degree).



This final graph shows the abysmal performance of both Jarvis's March and Chan's Algorithm when they are given sub-optimal data. The data set on the left is simply 1,000 random points, and the data set on the right is also 1,000 points, but they are all arranged in a circle around point (0,0). I will note that due to rounding errors, not all of the points are considered to be on the hull, furthermore the Brute Force algorithm came up with a slightly different hull count (also due to rounding errors.



References

[1] A. Levitin, Introduction to The Design and Analysis of Algorithms, 3rd Edition, Boston: Addison-Wesley, 2012.

[2] Cormen, Leiserson, Rivest and Stein, Introduction to Algorithms, Boston, Masschusetts: The MIT Press, 2009.

[3] R. Graham, "An efficient algorithm for determining the convex hull of a finite planar set," *Information Processing Letters,* vol. 1, no. 4, p. 2, June 1972.

[4] Wikipedia, contributors, "Wikipedia - Ronald Graham," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Ronald_Graham. [Accessed 28 05 2019].

[5] GeeksForGeeks, "GeeksForGeeks," [Online]. Available: https://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/. [Accessed 28 05 2019].

[6] R. Jarvis, "On the identification of the convex hull of a finite set of points in the plane," *Information Processing Letters,* vol. 2, no. 1, pp. 18-21, March 1973.

[7] B. York, *Convex Hull Lecture,* Portland, OR: Bryant York, 2019.

[8] Wikipedia, contributors, "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Gift_wrapping_algorithm. [Accessed 27 05 2019].

[9] A. Amelinckx, "Modern Farmer," Modern Farmer Media Inc., 25 09 2015. [Online]. Available: https://modernfarmer.com/2015/09/difference-between-llama-and-alpaca/. [Accessed 30 05 2019].

[10] Wikipedia, contributors, "Wikipedia," Wikipedia Foundation, [Online]. Available: https://en.wikipedia.org/wiki/Center_pivot_irrigation. [Accessed 30 05 2019].

[11] T. M. Chan, "Optimal output-sensitive convex hull algorithms in two and three dimensions," *Discrete & Computational Geometry,* vol. 16, no. 4, pp. 361-368, April 1996.

[12] Wikipedia, contributors, "Wikipedia," Wikipedia Foundation, [Online]. Available: https://en.wikipedia.org/wiki/Timothy_M._Chan. [Accessed 01 06 2019].

[13] Wikipedia, contributors, "Wikipedia," Wikipedia Foundation, [Online]. Available: https://en.wikipedia.org/wiki/Chan%27s_algorithm. [Accessed 30 05 2019].

[14] O. Daescu, "UT Dallas," [Online]. Available: http://www.utdallas.edu/~daescu/convexhull.pdf. [Accessed 30 05 2019].