Empirical Analysis of Convex Hull Algorithms

Edgar Aguilar Cruz, Christopher Pitte, Shane Brown, Katelynn Shelton

Overview

The convex hull creates the smallest convex set that encloses all available data points within that hull, similar to how a rubber band stretches around a set of nails on a board, as shown in Figure 1. This geometric concept is used from everyday applications such as the wand tool in picture programs, computing geographical data, to more complex operations such as bounding and object collision in modern cars as well as other instances across engineering and science as a whole.

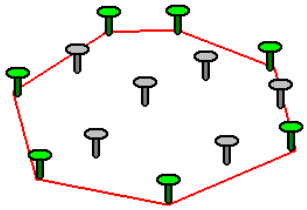

Figure 1: Graphical Image of Hull

The most direct way of computing the hull would be a brute force method, however that method is extremely taxing on operations and can take an exorbitant amount of time as the array of points increases. While it is the most straightforward method, other options to compute the hull are less demanding on the system and take considerably less time with the same amount of accuracy. These methods include the Incremental Randomized, Jarvis' March (also known as Gift Wrapping), and Divide-and-Conquer form of Convex hull.

Methods

Brute Force Convex Hull will be the reference method when analyzing these three algorithms. It meticulously checks each point within the hull to ensure that there are no outlying points beyond the hull border. Due to this complexity of checks the algorithm has an average time complexity of $O(n^3)$ which is also its worst case run time as well. Due its high time complexity compared to the other algorithms before it will be used as a base benchmark to ensure that the other algorithms are performing as expected or better.

Incremental Randomized works by randomly adding the points one by one to the hull until all points are encased. With each new point the hull is updated and all existing faces of the polygon that are visible from the new boundary are removed. Using this method often involves a small hull that rapidly grows and takes an average time complexity of $O(nlog(n))$ while having a worst case complexity of $O(n^2)$. The process of how the hull is computed can be seen below in Figure 2.

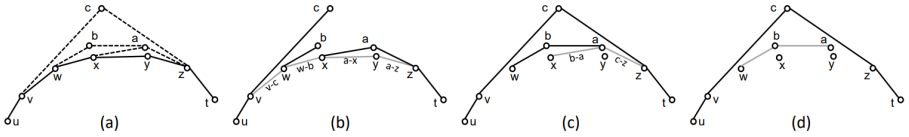

Figure 2: Incremental Randomized Computation

Divide-and-Conquer has the same time complexity for both average and worst case as Incremental Randomized but works by recursively dividing the array of points in half until only a small set remains. When the base case of a small set of points is reached, it computes the hull for those points and then begins to find the upper and lower bounds of the hull. Once the upper and lower bounds are determined, it merges the two smaller hulls to form a larger hull using

those bounds and removing the inner bounds. Figure 3 below shows a representation of how the smaller hulls merge to form larger ones.

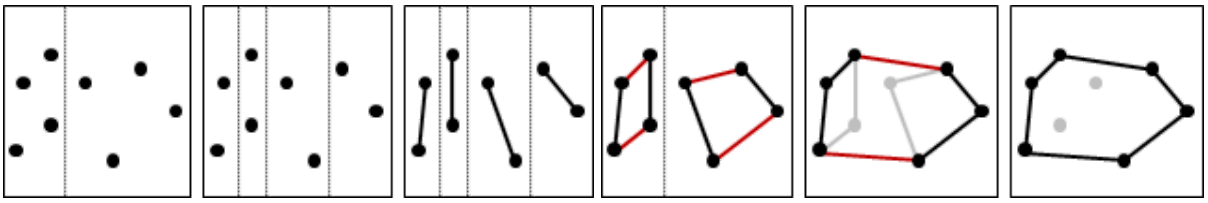

Figure 3: Divide-and-Conquer Hull Computation

Jarvis' March (Gift Wrap) has the same worst time complexity as the two algorithms mentioned above, however, has a time complexity of O(nh), where h is the number of points on the hull, instead. It functions by first finding the point that is the furthest left. Once the point that is to the furthest left is found begins computing the point that has the smallest angle while keeping all points towards the inside of the hull and continues to do this until it reaches the original starting point as shown in Figure 4.

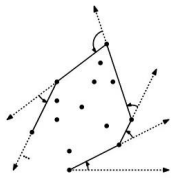

Figure 4: Jarvis' March Angle Wrap Implementation

Objectives

Incremental Randomized, Jarvis' March (Gift Wrap), and Divide-and-Conquer algorithms are expected to take less time and operations compared to their brute force counterpart. While these three all have a worst time complexity of $O(n^2)$, both Incremental Randomized and Divide-and-Conquer have an average complexity of O(nlog(n)) while Jarvis's March has a complexity of O(nh). With slightly different time complexities the scalability of these algorithms were brought into question. Does performance and accuracy degrade at increasing amounts of hull points for one algorithm over the other? How does different point distributions such as uniform random and Gaussian clusters, structured shapes and array size affect the performance between these algorithms? Does one data type hold an advantage over another? Lastly, how does each algorithm handle outliers and does one do so more efficiently than another?

Experiments

The first step in the experiment process was to implement the three algorithms to test as well as a base algorithm in a programming language to be able to test different array types. Another program was also created for the purpose of testing that created different array types and structured shapes for the purpose of testing. These programs were created with use of partial code both found from online sources and with generative AI, as well as created by members of the group. Implementation and changes to code were annotated as well as credited where original code came from if found from an external site. All programs were created in C++ with some minor python uses in between implementations for ease of use. Algorithms were checked for correctness with implementations for a variety of cases before experimental data was collected. Those correctness tests were not collected beyond testing means, and are not represented in the data that was used for these experiments. All experimental data was produced with the program that was created specifically for this experiment.

Computers with these specs were used for the various tests. First computer: i9-14900KF CPU, 64.0 GB Ram running Windows 11 Pro version 25H2 Desktop. Second Computer: AMD Ryzen 3 3250U with Radeon Graphics CPU,16 GB RAM Windows 11 Home 24H2.

Three major experiments were performed on these algorithms to test various factors of them. The first was to see how exponential data affected each of these algorithms. Since 2 points could only have a hull of those two, 2^1 results will not be included in the results, but were computed for some of the algorithms. Brute Force was only measured for 2^{10} - 2^{16} to see the vast difference in computation operations and time needed to complete to gauge the difference between the brute method as well as the other algorithms. Arrays up to size 2^{24} were tested for each algorithm within the scope to see the effects the size of the array had on correctness as well as complexity of the algorithm, and if diminishing returns were a factor as the array grew.

The second test involved Gaussian clusters, and these data points were tested with higher array indexes than expodential was. These data types were used to test for correctness between the algorithms of varying data and to ensure that all the algorithms were able to agree on the points that created the hull. In addition to correctness testing, the clusters were used to see the variance in run-time when higher degrees of varying arrays are used with a more real-world example.

Lastly various shapes were tested to benchmark timing and accuracy of these algorithms for shapes whose hull would be easily determined. Each of these shapes consisted of the same number of points. Of these benchmarks the shapes used were a circle, square, grid shape, and a disk (circle within a circle). Those arrays can be seen below in Figures 5-8.

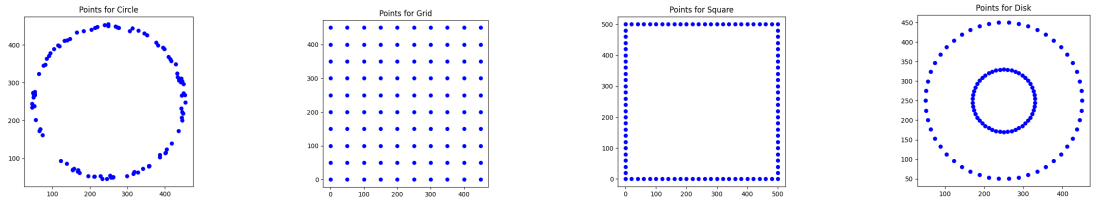

Figure 5-8: Shape Structures Used in Testing

Results

Empirical Analysis was used for the majority of these tests to see how the algorithms function in real-time on differing devices. The empirical data that had the most consistency and lowest run-time, or best-case time, was included in the data conclusion due to other processes on the systems being possible causes of congestion on computation.

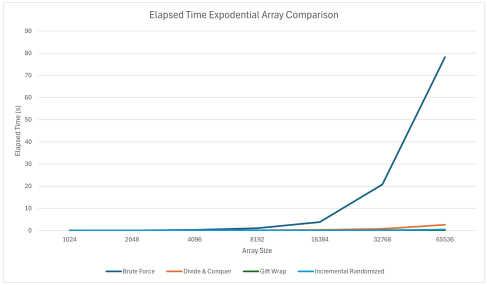

Figure 9: Time Comparison with Brute Force

The first set of tests were conducted using exponential data sets that were generated as uniform random. Theoretically, as the amount of data increases,Brute Force should take the longest amount of time and highest operation count to compute the hull with both Incremental Randomized and Divide-and-Conquer having roughly the same operation and time to do the same array. Jarvis' March would be expected to see the lowest in both operation count and time

computation when compared to the above. Figure 9 shows the time that each algorithm took to run the same arrays including the reference Brute Force method. Notice that as the array gets exponentially larger the time for the reference grows exponentially as well. This quickly becomes an unsuitable computation method when working with larger data sets. Figure 10 shows the outcome of the three algorithm methods for lower exponential and Figure 11 higher exponential growth arrays without the reference of brute force. While Figure 9 shows an overwhelming time that dwarfs the other algorithms, Figure 10-11 shows the differences between the three algorithms with more precision. Jarvis' March consistently performs the best regardless of the array size when working with exponential array sizes as can be shown by Figure 10-11, but Divide-and-Conquer and Incremental displayed similar outcomes that was expected with their time complexities. When the arrays reached an even higher level of exponential arrays there was a very clear distinction between the three algorithms run-time. While before both Incremental and Divide-and-Conquer had roughly the same run-time they had a divergence where one clearly became shorter to compute than the other that is clearly seen in Figure 11. Jarvis' March however still presented the shortest run-time despite the issues that it had noted later.

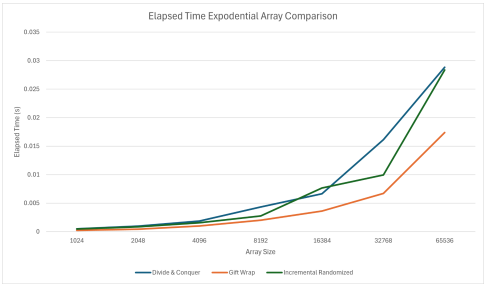

Figure 10: Lower Level Exponential Arrays

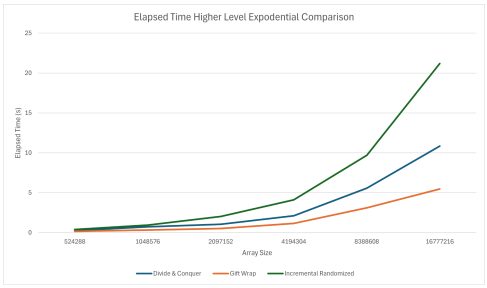

Figure 11: Higher Level Exponential Arrays

It is also worth noting that inconsistencies were noticed as the array grew into the much larger numbers as shown by Table 1. Until 2^{12} all algorithms had consistently agreed on the hull size at 2^{12} the first disagreement with the hull size was found with Divide-and-Conquer, after which consistently had a larger hull number than the other two and Jarvi's March often had either the same hull size or only slightly larger than the Incremental Random implementation. These inconsistencies could be the result of either a high array number for the implementation that was used or implementation that under certain circumstances were incorrect. Jarvi's March was also noticed on occasion with these comparisons giving points of the hull that were on the hull but expanded by another point and thus should not have been included. Additional programming for conditions such as that would have helped reduce the number of hull points to meet those of the Incremental Random considering the small differences between the two.

Table 1: Exponential Hull Size Inconsistencies

Array Size	Divide-and-Conquer	Jarvis' March	Incremental Random
2^{11}	17	17	17
2^{12}	22	21	21
2^{13}	26	22	20
2^{14}	27	25	25
2^{15}	24	23	23
2^{16}	37	27	23

The second set of tests was done using Gaussian clusters which were also randomly generated. This set of tests performed with the same expectations as the Exponential array size, however the size of these arrays tested were much larger due to the nature of the cluster. Despite the larger quantities there were no noticed discrepancies between the array sizes as the array grew. There was no noticeable difference between the Exponential and the Gaussian cluster computations run-time up till 10 million points was tested. At this point Incremental Random became much more time consuming that the other two methods. Similarly to the exponential arrays, Jarvis' March still performed the best out of all the algorithms. A small array of the timing taken for each algorithm while computing the Gaussian Clusters is displayed to the right in Figure 12.

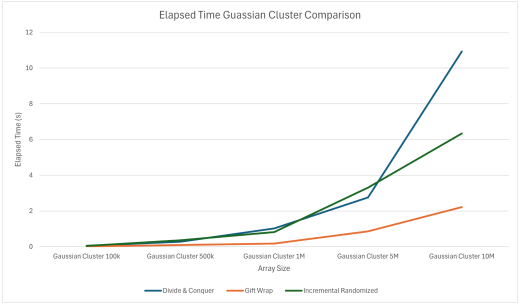

Figure 12: Gaussian Clusters Run-time

Finally the last set of tests was run on structured sets to look like different shapes. Those shapes were circle, grid, square, and disk shapes and all consisted of 100 points for each shape. Figures 5-8 in the Experiments section showcases how the points were laid out for each of these four tests while Figures 14-17 show those same shapes with the hull. Figure 13 shows the discrepancies between each of the three algorithms. For this test Divide-and-Conquer and Jarvis' March performed similarly while Incremental Randomized had vast differences with both circle and disk but had a more accurate output than the other algorithms overall. Similarly there were some inconsistencies between the hull sizes between each of these algorithms similar to how the exponential array shown at higher numbers. This shows that the problem wasn't with the size of the number but most likely the way the points fell on a grid. Incremental Random was the only algorithm to completely disregard any points that lied on the hull and only count the points where the hull shifted as what would be expected. Divide-and-Conquer and Jarvis' March both had instances where the hull held points that should not have been included as they were on the hull line.

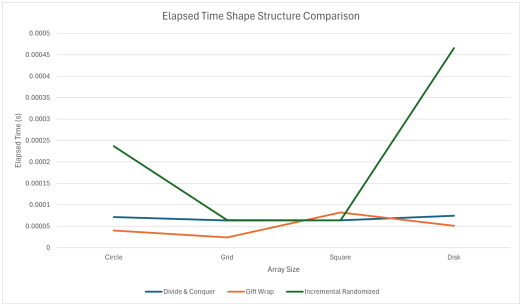

Figure 13: Shape Structure Run-time

Table 2: Shape Hull Size Inconsistencies			
Array Shape	Divide-and-Conquer	Jarvis' March	Incremental Random
Circle	37	38	39
Grid	10	20	4
Square	34	100	4
Disk	38	50	50

While each of these tests show innate faults that could have either been avoided or corrected with a higher level of implementation, they give a base line understanding of how each of these algorithms perform. Overall Brute force was by far the most system demanding and most time consuming and showcased the great advancement implementation and run-time when compared to any of the three algorithm

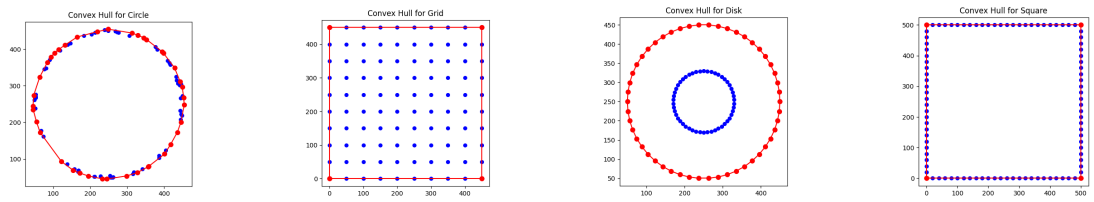

Figure 14-17: Shape Structure Hull Results

implementations attempted. Divide-and-Conquer and Incremental Randomized showed remarkably better marks than Brute Force but was still heavily intensive on operation time and still had more growth being an O(nlog(n)) algorithm when compared to Jarvis' March, which was an O(nh) algorithm. Jarvis' March, however, excelled in almost all categories, with Jarvis' March performing only slightly better than Incremental Random in most tests with run-time and complexity. Incremental Random performed with the greatest accuracy and correctness between all algorithms tested making it a viable choice for both accuracy and speed computations.

With each of these tests concluded and the results of them we can return back to the questions originally asked. Scalability of these algorithms vary greatly depending on the uses these are intended for. Testing shows that Divide-and-Conquer and Jarvis' March were inaccurate with the hull composition with these implementations. Incremental Randomized and Divide-and-Conquer could eventually become infeasible with arrays and data points that go higher than the tested measure for these experiments as well, especially if trends such as shown in Figure 11 continued. Jarvis' March, however, showed that their growth was remarkably slower than the reference Brute force or other two algorithms. Therefore for scalability both Jarvis' March and Incremental would be a viable option depending on the level of accuracy needed and available resources to compute that hull.

Unfortunately, some accuracy problems were noted within the experiments. These were shown to not be based on increasing size of the array, but more the shape of the array. Jarvis' March was shown to include all points even those that lied on the hull and Incremental was shown to only provide the points that formed the hull and not those that lied on it. Divide-and-conquer had the highest level of inaccuracy between all algorithms and that is probably due to the way that it stitches the hull together after dividing it into smaller problems. It's possible that the optimal reference point to reconnect the hull was missed and not recalculated properly. Distribution, not size, was the key factor in the inaccuracies of these algorithms and their agreement of hull size. Gaussian clusters showed the highest level of agreement with each of the algorithms, having had no disagreements on the hull size between tests, and the shapes structure had the highest level of disagreement with Square being the most extreme case. This could mean that Gaussian clusters would be the optimal data type for convex hulls as a whole. Lastly, outliers were not noticed to have any discernible impact on implementation or computation. Outliers were mostly used in exponential measurements but didn't seem to impact the data in a meaningful way.

Biggest Limitation for these experiments perhaps would be the time needed to complete each step of data collection and ensuring programing correctness. If repeating this experiment a second time more focus would be placed at testing higher limits of the program's correctness and ensuring that programs like Jarvis' March and Divide-and-Conquer don't have points that lie

on the hull but only points that make the hull. Doing so would probably guarantee a better runtime overall for both algorithms as well. Larger data sets would also be used within reason for exponential arrays as well as more shapes such as a star and cloud design as well.

Sources

Convex Hull | Brilliant Math & Science Wiki. (n.d.). https://brilliant.org/wiki/convex-hull/

- Figure 1 graphic

Blelloch, G. E., Carnegie Mellon University, Gu, Y., UC Riverside, Shun, J., MIT CSAIL, Sun, Y., & UC Riverside. (2020). Randomized Incremental Convex Hull is Highly Parallel. In Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20) (p. 13). https://jshun.csail.mit.edu/convexhull.pdf

- Figure 2 graphic

convex hull - 演算法筆記. (n.d.). https://web.ntnu.edu.tw/~algo/ConvexHull.html

- Figure 3 graphic

GeeksforGeeks. (2025b, July 23). Convex Hull Algorithm in C++. GeeksforGeeks. https://www.geeksforgeeks.org/cpp/convex-hull-algorithm-in-cpp/

- Original code for Divide and Conquer

Harshit Sikchi, Convex Hulls: Explained, April 22, 2017, https://medium.com/@harshitsikchi/convex-hulls-explained-baab662c4e94

- Figure 4 Graphic

C++ Program to Find the Convex Hull using Jarvis March, https://www.sanfoundry.com/cpp-program-implement-jarvis-march-find-convex-hull/

- Original code for Jarvis' March (Gift wrap)

OpenAI. (2023). ChatGPT, https://chatgpt.com/

- Various code implementations, between code corrections and correctness determinations. Minor uses for code implementations.

Autar, Avishkar. "Brute-Force Convex Hull Construction: Semi/Signal." Semisignal RSS, semisignal.com/brute-force-convex-hull-construction/. Accessed 30 Nov. 2025.

- Brute Force implementation

Table of team performance

Project Part	E. Aguilar Cruz	S. Brown	C. Pitte	K. Shelton
Brute Force	100%	0%	0%	0%
Divide-and-Conquer	80% Got working correctly	0%	3% Tested code	17% Originally wrote but wasn't correctly giving hull points
Gift Wrap	7% Fixed some errors with large array numbers	0%	3% Helped test code was working	90%
Incremental Randomized	0%	0%	99%	1% Only added timing and op count
Randomizer for Data Collection	100%	0%	0%	0%
Data Collection	50% Ran majority of tests again and helped fill in gaps in data, ran higher data points	0%	0%	50% Ran the majority of the tests for algorithms that were done at time
Graphs	0%	0%	20%	80%
Report	3%	3%	3%	91%
Presentation Preparation	10%	40%	40%	10%

All together the biggest difficulties for these experiments quickly became a lack of communication and time where all members could actively participate on the project. Members of the group were at different levels of programming skills and understanding of the algorithms, and some worked better in different languages than the one used for final computations. When some members were available to meet to discuss project completion, others were unable to attend. Remote work became the default for each member's part and communication between completed and needed parts suffered to some degree. If repeating the project, times and parts would definitely become a higher focus for each member as well as higher level of check in on project part completions.