COMP0005 Group Coursework

Academic Year 2023-24

Group Number: 025

1 Overview of Experimental Framework

1.1 Framework Design/Architecture

To analyse the performance of our 3 convex hull algorithms, we needed to be able to graph the performance of each algorithm with different amounts of points and consider the various edge cases. Our framework has an attribute called dataGenerator which is an instance of the class TestDataGenerator, and this object has its own attribute, points, that is a list of tuples containing the coordinates that we test our algorithms with. The TestDataGenerator class has methods that can either generate random points, integer coordinates that correspond to the points of an n-sided polygon (the n can be varied), integer coordinates that lie within the points of an n-sided polygon, or collinear points of 3 different cases – the lines $y = x$, any line parallel to the horizontal axis, and any line parallel to the vertical axis. With these methods, most edge cases and scenarios can be triggered by calling these methods on the dataGenerator attribute in the ExperimentalFramework class.

In the ExperimentalFramework, most of the methods allow us to get the necessary data for our experiments, while some methods are used to display the data in graphs, and we also included a method to view a scatter plot of the points and the convex hull to ensure that we were generating the correct convex hull, and so that we could test all edge cases. After generating points from the dataGenerator attribute for our input set, we ran an experiment and used the timeit library by subtracting the start time from the time after the algorithm finishes forming the hull to measure the runtime of each algorithm to produce the output set. For every trial (number of points / what we were testing for that experiment) and algorithm, we found the runtime 10 times each (which can be adjusted via the attribute numTrials), and then found the median and recorded that result. We decided a median would be better suited for these experiments than a mean because it would help us avoid considering outliers in our results, while still giving us a good approximation of an average runtime. To compare our algorithms, we plotted line graphs of the median runtimes, this was done with the library matplotlib and the utilisation of the dictionary attribute results, which keeps track of algorithm names with corresponding test size and time taken for test.

The experiments we conducted began by comparing the 3 algorithms' runtime as we increased the number of points (n) which were generated randomly. Following that, we looked at cases where the number of points on the convex hull (h) was equivalent to the number of points generated ($n = h$). This was done by simply generating the coordinates of the points of an n-sided polygon up to 600 integer points. The reason for limiting the polygon to a maximum of 600 sides is based on computational precision and practicality. When a polygon has more than 600 sides, the central angles (subtended by a line between 2 edges) become extremely small, and when the coordinates are rounded to integers, several adjacent points might become collinear (fall on the same line). This results in a reduction in the number of unique points that can contribute to the convex hull. We could have created a polygon with more than 600 integer points by using a convex hull algorithm to generate the h points themselves and then taking the output set produced by one of them (e.g. Graham Scan) as the input set. In this context, we would shuffle the order of points using random.shuffle() so it wouldn't guarantee an ordered list or any other particular case, but we didn't think doing this was necessary because we were able to see the general trend of the 3 algorithms with the n-sided polygon method. We then looked at varying h in a small range (3 – 10) for a fixed n – we tried this with a relatively large n (10000) and then a relatively small n (1000) as they gave us slightly different results. Following that, we looked at each algorithm separately, looking at how their behaviour changes as the ratio of h to n varies, while still varying n. Finally, we ensured to test edge-cases with collinear points ($y = x$, parallel to the x-axis, and parallel to the y-axis) because, since each algorithm relies on finding either the left-most or bottom-most point, there could be flaws if all the points lie on a line parallel to either axis, but we didn't find any errors in our implementation.

1.2 Hardware/Software Setup for Experimentation

The hardware we used to test our algorithms was a MacBook Pro 14 with the M2 Max chip. It had 12 cores, with a 2.42GHz base clock speed and a boost clock speed of 3.7GHz when the chip is in heavy load, as well as a 30 core GPU and 32GB RAM. The performance would vary with machines of different specification for each algorithm; hence we used the same machine to run all our tests. Additionally, the computer was running on macOS Sonoma 14.2.1, and we used Visual Studio Code as our text editor. We ran it using Jupyter Notebooks and Python version 3.11.5 from anaconda3 running in the base environment. The imported libraries included the math, random, and timeit libraries, as well as the pyplot module from matplotlib.

2 Performance Results

2.1 Jarvis march algorithm

We implement this by starting at the leftmost point and moving counter-clockwise, selecting the point that is the most counter-clockwise to the previous point until it returns to the starting point. The complexity is $O(nh)$ where n is the number of points in the input set and h is the number of points on the convex hull. This means that the algorithm is output-sensitive; in other words, performance depends on the size of the output as well as the input.

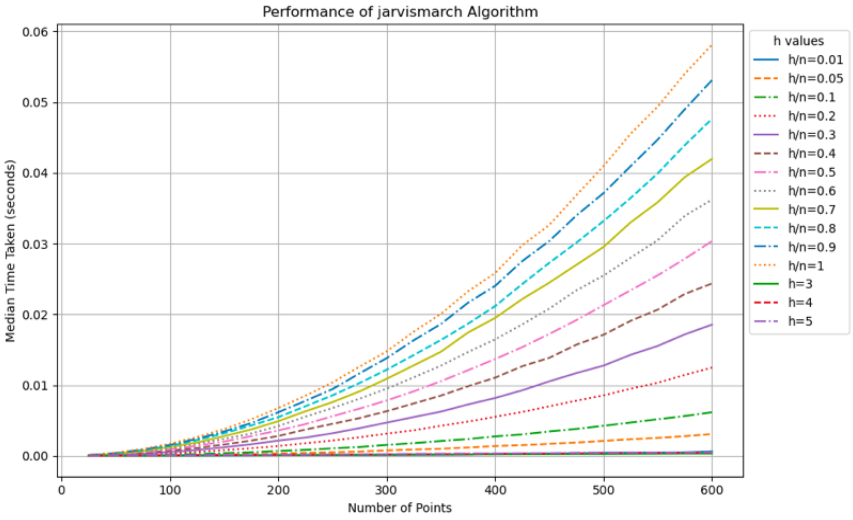

Figure 1. The Performance of Jarvis March Algorithm for Different Values of n and h

Figure 1 illustrates the median time taken for the computations across different configurations of h and varying numbers of points (n). The algorithm's performance linearly declines with fixed h values {3, 4, 5} and lower h/n ratios {0.01, 0.05}. Conversely, higher h/n ratios lead to an exponential increase in the computation time. Optimal performance is observed when the convex hull includes a minimal number of points, with h = 3 being the ideal scenario (as h = 2 is a line segment). In contrast, the higher h/n values correlate with steeper increases in the computation time as the number of points grows. Notably, when every point in the input set lies on the convex hull (i.e. $h = n$), the algorithm exhibits its least efficient performance as this is the worst-case because every point will have to be compared against every other point. In this context, the algorithm's complexity becomes $O(n^2)$, which implies that the algorithm is less suited for datasets where a large proportion, or indeed all the input points, lie on the convex hull.

2.2 Graham Scan algorithm

For this algorithm, we initially select the point with the lowest y coordinate as the pivot and sort the remaining points based on their polar angle with respect to the pivot. The computational complexity of the algorithm is $O(n \log n)$, and this is primarily due to the complexity of the sorting step. Unlike Jarvis March, the algorithm's performance is not influenced by the size of the output. In our implementation, we use Python's Timsort algorithm – a combination of merge sort and insertion sort – for sorting, which has a time complexity of $O(n \log n)$.

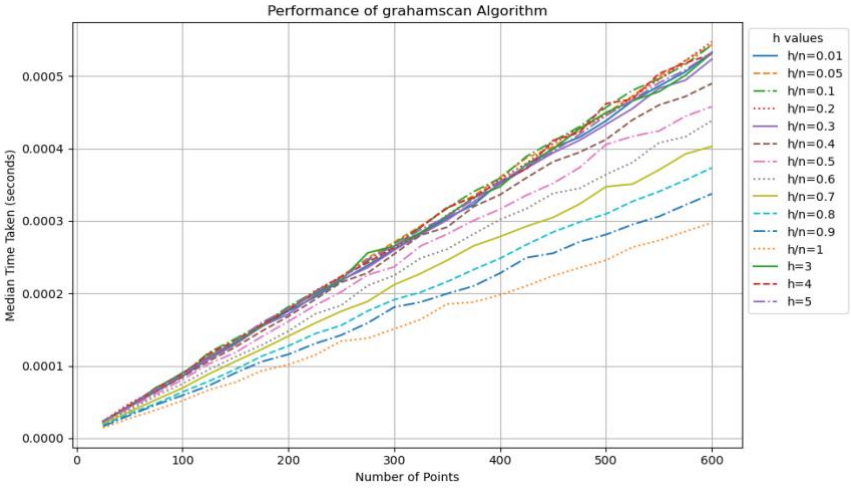

Figure 2. The Performance of Graham Scan Algorithm for Different Values of n and h

Figure 2 demonstrates the median time taken for our computations by this algorithm as we vary n and h. As shown, the algorithm maintains a consistent

increase in computation time as n grows, which matches with its $O(n \log n)$ complexity. The performance impact of different h/n ratios is observable, however, more subdued compared to algorithms with a higher dependence on h. Interestingly, the higher h/n values result in quicker computation times. This demonstrates the algorithm's efficiency in handling cases with a higher ratio of points on the convex hull, with $h = n$ representing the best case. As h/n ratio decreases, there is a noticeable but gradual increase in the computation time. The theoretical worst-case for Graham Scan should be the worst-case for Timsort, which is when there are a maximum number of runs, however, this can't exactly be generated since its minimum run size is chosen from a range of values. Instead, in practice, it is when $h = 3$, where 3 points are part of the convex hull, that the algorithm performs least efficiently.

2.3 Chen's algorithm

Chan's algorithm, a hybrid approach for computing the convex hull of a set of points, combines the previous 2 algorithms to achieve a complexity of $O(n \log h)$. This should theoretically make Chan's algorithm efficient for large datasets, especially when the size of the convex hull is significantly smaller than the number of input points.

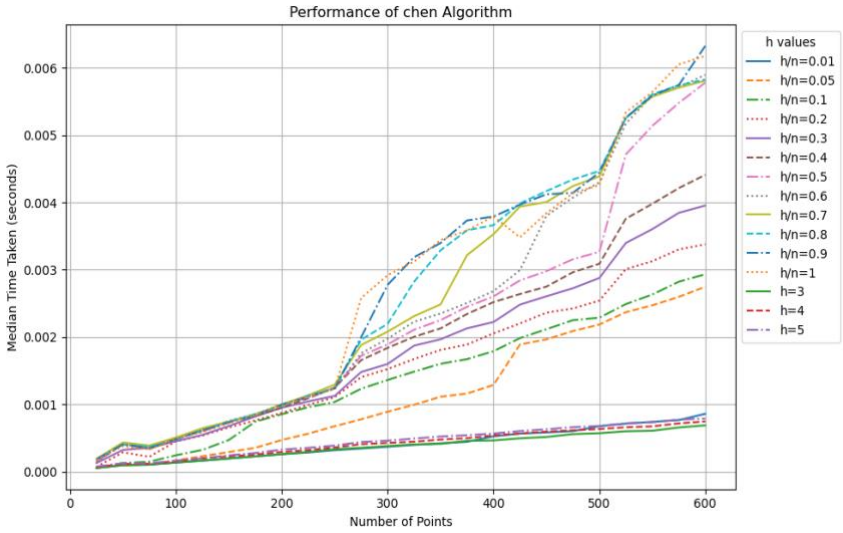

Figure 3. The Performance of Chan's Algorithm for Different Values of n and h

Figure 3 illustrates the median time taken for computations by Chan's algorithm across a variety of h/n ratios and different numbers of points (n). Notably, the algorithm maintains a stable and low computation time for smaller datasets, with a noticeable increase in time as n grows, particularly for higher h/n ratios. Importantly, for the fixed h value of 3, where the convex hull is minimal, Chan's algorithm performs most efficiently, while $h = n$ is the worst-case scenario for the algorithm to compute the convex hull points and the time complexity becomes $O(n \log n)$. This suggests that while Chan's algorithm is designed to be efficient for larger datasets, its performance can be influenced by the proportion of points on the convex hull. However, even at higher h/n ratios, the increase in computation time follows a more gradual curve compared to purely output-sensitive Jarvis March.

3 Comparative Assessment

Our experimental evaluation identified the performance characteristics of each of the three convex hull algorithms. We recognised the strengths and weaknesses of each algorithm, across best, average, and worse-case scenarios, and have identified the circumstances under which some algorithms are better suited than others.

Jarvis March offers simplicity and performed reasonably well in scenarios where the input size was small. It also has a constant space complexity, making it useful in situations where memory is limited. However, due to its time complexity of $O(nh)$, it quickly becomes a bottleneck in scenarios with larger datasets (as seen in **Figure 4**) or when points were evenly distributed along the convex hull. If all points lie on the convex hull ($n = h$), the time complexity becomes quadratic, making it very inefficient.

Graham Scan, which takes advantage of efficient sorting techniques, highlighted superior performance in handling larger datasets. Its time complexity of $O(n \log n)$ ensured scalability, making it an ideal choice for scenarios where computational resources are not a limiting factor (as seen in **Figure 4**). Nevertheless, Graham Scan's performance might degrade when faced with highly irregular point distributions or datasets with a small

number of points. Additionally, the Graham Scan algorithm requires linear memory space due to the use of a stack, which means it may not be useful in scenarios where memory is limited.

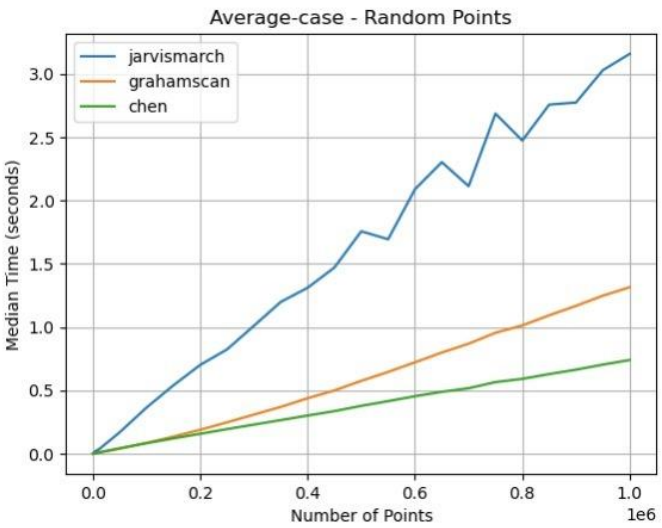

Figure 4. The Performance of all 3 Algorithms for Values of n 1-1,000,000

Figure 5. The Performance of Graham Scan and Chan's for Values of n 1-100,000

Chan's Algorithm, with a time complexity of $O(n \log h)$, emerged as a robust contender, striking a balance between efficiency and adaptability. By using a divide-and-conquer approach, it demonstrated remarkable performance across various scenarios (as seen in **Figure 4**), particularly excelling in datasets with non-uniform distributions. However, like the Graham Scan algorithm, Chan's algorithm also has a linear space complexity, which can be a drawback in situations where memory is limited. While the implementation of the algorithm can be quite complex, especially compared to Jarvis March and Graham Scan, Chan's Algorithm's efficiency makes it a compelling choice for many practical applications.

As shown in **Figure 5**, Graham Scan showed slightly better performance than Chan's Algorithm for datasets containing fewer than 80,000 points. This is due to Graham Scan's efficient sorting, while Chan's algorithm divides the input into subsets, which introduced an overhead.

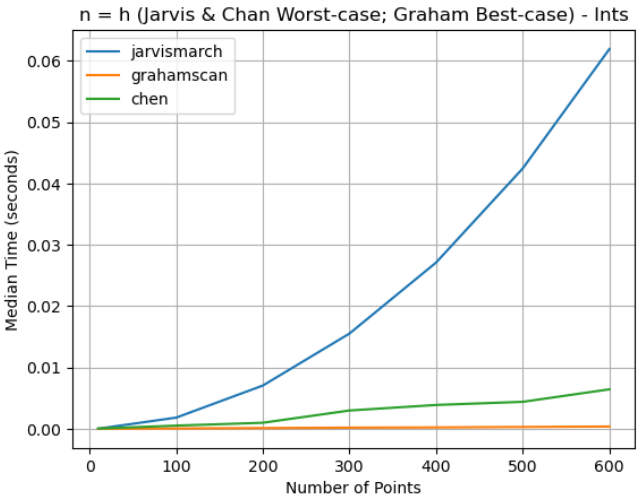

Figure 6. The Performance of all 3 Algorithms when n = h

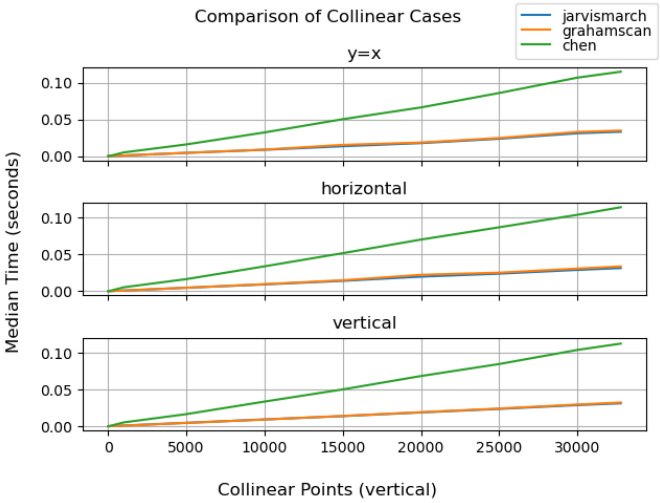

Figure 7. The Performance of all 3 Algorithms for Collinear Coordinates

As seen in **Figure 6**, scenarios where the number of points overall is equal to the number of points on the convex hull ($n = h$) proved to be the worst-case for Jarvis March and Chan's Algorithm, but the best-case for Graham Scan. Here, Jarvis March had a quadratic complexity, resulting in a significantly less efficient execution. In the case where $n = h$, Chan's Algorithm performed similarly to Graham Scan, however Graham Scan emerged as the most efficient algorithm in this case. This is likely to be because the number of points on the convex hull has no impact on its performance.

As you can see in **Figure 7**, when points are all collinear, Chan's Algorithm is the worst-performing, while both Jarvis March and Graham Scan both perform efficiently and similar to each other. This is the case for all types of collinear coordinates, including $y = x$, horizontal and vertical points.

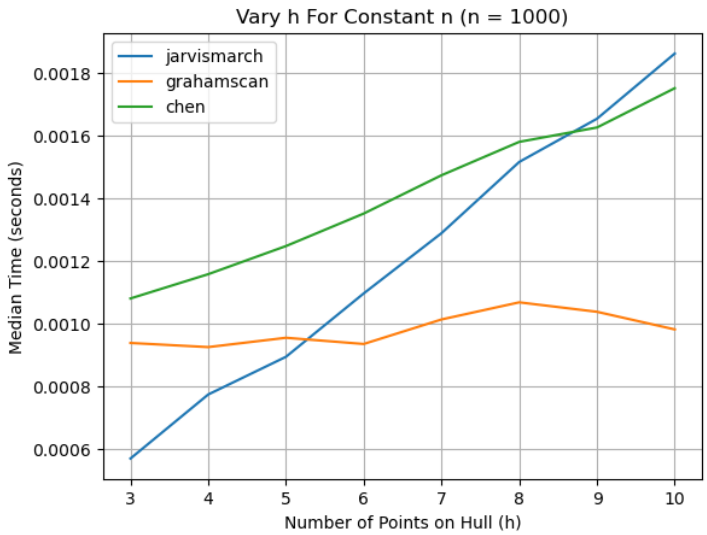

Figure 8. The Performance of all 3 Algorithms for Values of n = 1000 and small h

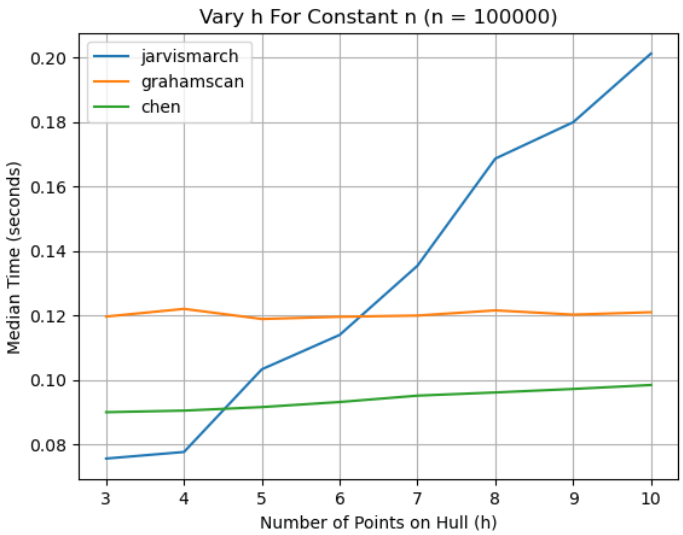

Figure 9. The Performance of all 3 Algorithms for Values of n = 100,000 and small h

In **Figures 8 and 9**, when testing the performance for small values of h, there are observable differences in the execution time. While Jarvis March remains relatively unchanged, Graham Scan performs best for small values of n, while Chan's algorithm performs best for large values of n. This is likely due to Chan's time complexity of $O(n \log n)$ which performs best for small values of h, while Graham Scan's complexity of $O(n \log n)$ is unchanged for small values of h.

In conclusion, there is no one-size-fits-all solution when it comes to these convex hull algorithms. The suitability of each algorithm depends on several factors, including input size, point distribution, and computational resources available. For scenarios with small datasets and a few points on the convex hull, Jarvis March and Graham Scan are well-suited. On the other hand, Chan's Algorithm is preferrable for large datasets.

4 Team Contributions

Each member of the team contributed equally: Raghav Awasthi (SN 23022439), Rasul Bakhodirov (SN 23158334), Prantanil Bhowmick (SN 23009912), Umar Ali (SN 23080656)