COMP0005 Group Coursework

Academic Year 2023-24

Group Number: 30

1 Overview of Experimental Framework

1.1 Framework Design/Architecture

The methods in the experimental framework include instances of the TestDataGenerator class, where its methods are used to generate points based on the parameters passed to the framework. The framework also tests the performance of the three convex hull algorithms: Jarvis March, Graham Scan and Chan's Algorithm.

The framework is parameterised with the following arguments: integer T , the number of trials; integer M , the maximum number of points; integer S , the step increment of the interval for the points. In the methods defined by the Experimental Framework, the parameters: list H ; the list of specified number of points in the convex hull; integer h , the specified number of points in the convex hull; integer n , the number of points generated; string t ; the type of point to be generated by the TestDataGenerator, where t is only either "random", "collinear", "circle" or "controlled" , to test the algorithms edge cases and compare the algorithms based on point type; list A , the list of algorithms tested by the framework are used for to run tests with the three algorithms. For a T number of trials, the framework starts generating a point range, P up to and excluding M with increments of S , where the time taken for the algorithm A_i to run for number of points, n is logged and totalled. The sum of the times is then divided by T to obtain the average times of algorithms A which are then plotted and visualised by graphs via matplotlib.

The values chosen for the framework for random points are: T = 20, M =1000001, S = 50000. The values chosen for the framework for circle points are: T = 50, M =1001, S =50. The values chosen for framework for controlled points are: T = 10, M = 1000001, S = 50000. The values chosen for framework to test and compare the Graham Scans on list of n based on sorting algorithms for random points are: T =10, M = 10001, S = 500. The values chosen for framework for collinear points are: T = 10, M = 500001, S = 50000. These values were chosen to produce distinguishing graphs to compare the average and worst case of the algorithms, as well as comparing the performance between the three algorithms based on t .

1.2 Hardware/Software Setup for Experimentation

To use the Experimental Framework, the file must have pip installed where pip can be used to install anaconda, together with the Jupyter Notebook and Python kernel with version above Python 3.11 which is supported by the Experimental Framework. The modules that are required to be installed are: matplotlib, timeit, math and random. To import matplotlib, the line "import matplotlib.pyplot as plt" must be specifically included before calling the Experimental Framework or Test Data Generator. The random and math libraries need to be imported for the algorithm functions to work, and timeit module needs to be imported before calling the Experimental Framework.

2 Performance Results

2.1 Jarvis March Algorithm

Theoretical Computational complexity

The Jarvis march algorithm in the average case follows the time complexity of $O(nh)$. This is because it will firstly iterate all the points and compare with the leftmost point of the data, and get the polar angles of all points, which causes time complexity of $O(n)$. The next step involves, the use these polar angles and iterate through the convex hull points, h, and form a convex hull, which causes time complexity of $O(h)$. The theoretical time complexity for average case, will thus be $O(nh)$ in the average case. In the worst-case scenario, all the points will form a convex hull, which will let $n = h$ and hence, overall time complexity will be $O(n^2)$. Moreover, as $h \to 3$ from a large value, the time complexity of Jarvis March follows towards $O(n)$.

Experimental Computational Complexity

Average case: Our implementation of the algorithm followed the theoretical complexity of $O(nh)$. For example, according to Figure 1, the 4 lines drawn showed that for a constant h value, the graph of n vs time, increased linearly. Furthermore, Fig 2.1.1 shows that, as value of h gets doubled, the time taken for the algorithm nearly doubles.

Worst case: Our data shows us that Jarvis march follows the worst-case theoretical complexity of $O(n^2)$ when the input data forms a circle. If every point, form a circle, all the points will lie on the convex hull and hence create the worst-case scenario for Jarvis march algorithm where $n = h$ .Figure 2.1.2 shows that, n and time has a quadratic relationship in the worst case. For example, the number of circle points doubled from 200 to 400, the time increased by 4 times from about 0.008 seconds to about 0.035 seconds.

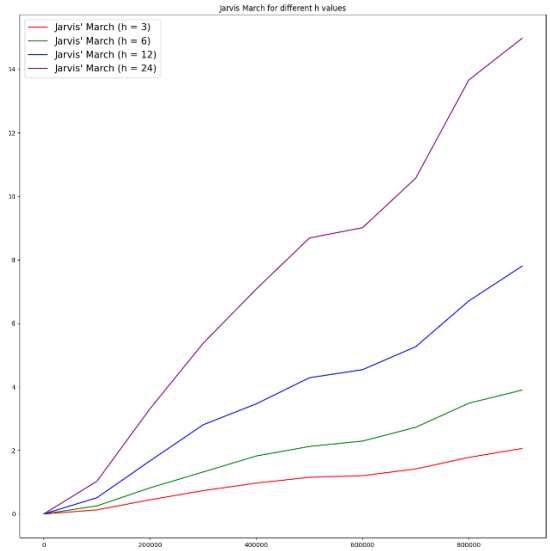


Fig 2.1.1 Visualisation of Jarvis March for h values of 3,6,12 and 24, for n until a million.

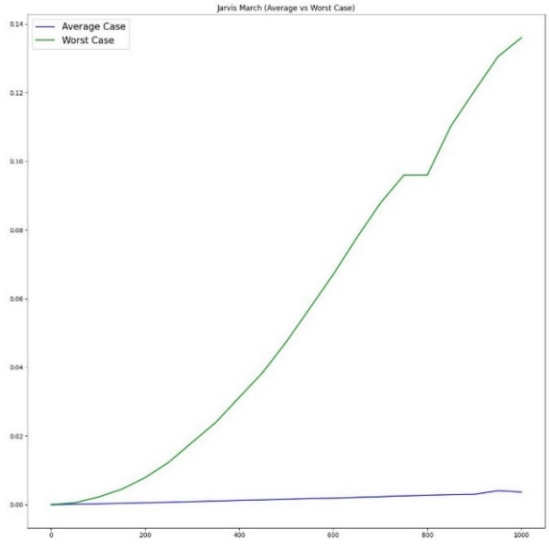


Fig 2.1.2 Visualisation of average vs worst case for Jarvis March

2.2 Graham Scan algorithm

Theoretical Complexity:

In the average and worst-case scenario, the algorithm first finds the bottom-most point for reference, and next it will calculate the polar angle for each point in the input. The list is sorted based on polar angle, or distance, if the polar angles are same, via timsort, the built-in sorting algorithm in Python. And finally, it will iterate through all the points and find the convex hull points. The sorting step has the time complexity of $O(n \log n)$, while for the rest of the steps, it has the time complexity of $O(n)$. Hence, the overall time complexity of the average and worst case will be $O(n \log n)$.

Experimental Complexity:

In both average and worst-case scenarios, our graph showed that our implementation followed the theoretical complexity of $O(n \log n)$. For example, in Fig 2.2.1, when data input size increased by 4 times from 200 thousand to 800 thousand, the time taken increased from about 0.3 seconds to around 1.6 seconds, which is an increase by about 5.3 times.

Comparing sorting algorithms for Graham Scan.

For the main Graham Scan algorithm, Python's in-built sort algorithm (timsort) is used where it achieves a time complexity of $O(n \log n)$ for both average and worst case. Different sorting algorithms (Merge Sort and Insertion Sort) are used to compare Graham Scan with our implementation with timsort. Fig 2.2.2 showed that when Graham Scan used insertion sort, the time taken for the Graham Scan to complete is much higher than Graham Scan that uses timsort or merge sort. The results shows that the Graham Scan algorithm is dependent on the sorting algorithm it uses, as an average time complexity of $O(n^2)$ is seen when implementing Graham Scan with an insertion sort, and $O(n \log n)$ for timsort and merge sort.

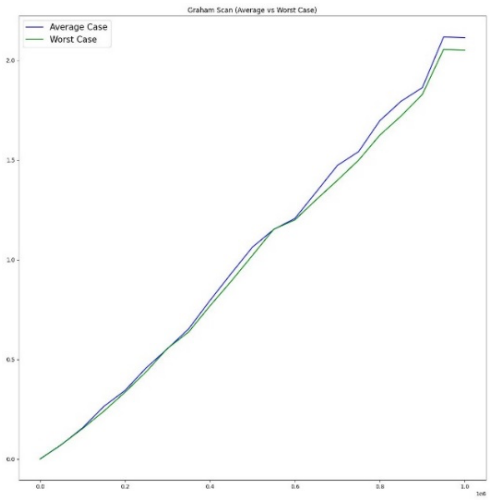


Fig 2.2.1 Visualisation of average and worst case of Graham Scan

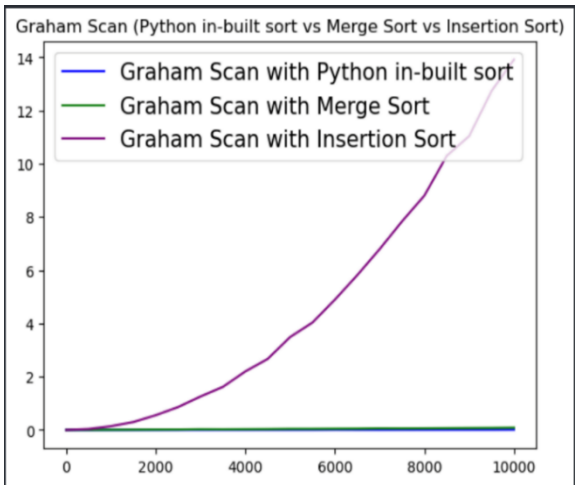


Fig 2.2.2 Visualisation of Graham Scan using with Timsort, merge sort and insertion sort

2.3 Chan's Algorithm

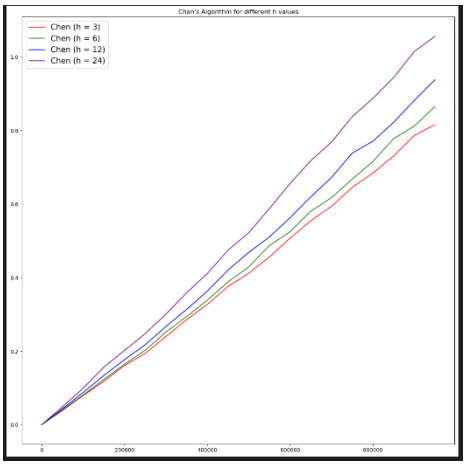
Theoretical Complexity:

Chan's Algorithm firstly partitions the set of points into subsets, each containing at most k points. For each subset, the mini convex hull is computed using Graham Scan algorithm. This phase takes $O(n \log k)$ time complexity. Using the points on mini convex hulls, Jarvis march algorithm will be run and allows the final convex hull to be formed. The algorithm repeats through different values of k and terminates when the final h value is known. Thus, average time complexity of Chan's Algorithm is $O(n \log h)$. For worst case, the Chan's Algorithm has time complexity of $O(n \log n)$, and this happens when $n = h$, all points forming a convex hull.

Experimental complexity

From Fig 2.3.1, it was shown that, with respect to n , time increases linearly. For example, when $h = 24$, when n doubled from 200 thousand to 400 thousand, time taken doubled from about 0.2 seconds to 0.4 seconds. It was also shown that with respect to h, time increased logarithmically. Compared to the earlier Figure 2.1.1 from Jarvis March, the gap between graphs is less widening for Chan's case for same increase in h. This can show that the increase of time with respect to h is logarithmic. Thus, the performance result show that the graph follows $O(n \log h)$.

In the worst-case scenario, it does not follow the expected time complexity of $O(n \log n)$. Fig 2.3.2 showed that our algorithm has performed worse than the expected time complexity, and this may be due to the modified Jarvis March where a binary search algorithm in the Chan's algorithm taking up the time complexity.



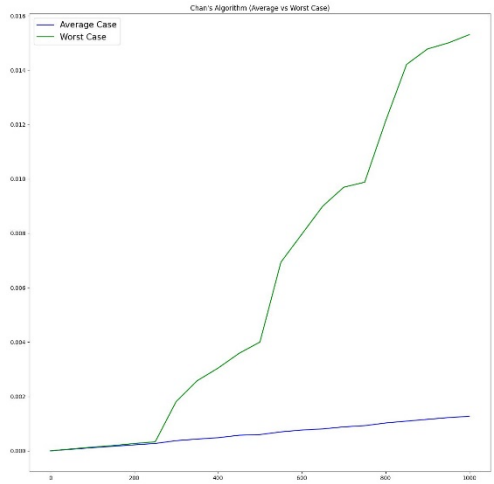


Fig 2.3.1 Visualisation of varying h for Chan's Algorithm

Fig 2.3.2 Visualisation of average and worst case of Chan's Algorithm

3 Comparative Assessment

3.1 Varying h (Fig 3.1)

Based on the graph, Jarvis' March is consistent with its expected time complexity of $O(nh)$, where the graph demonstrated that the time taken is directly proportional to h . This has indicated it is highly efficient for datasets with a small h . Graham Scan is stable across various dataset, where it has performed the correct computational complexity with $O(n \log n)$. Chan's Algorithm does fluctuate as Jarvis march but a much lesser extent. the overall trend shows a gradual increase in the graph with increment of h values, as shown by its time complexity of $O(n \log h)$. As h increases, Jarvis March becomes more inefficient compared to the other two algorithms, therefore Chan's Algorithm is better suited for greater values of h . However, at h =3, Graham Scan performed the slowest compared to Jarvis March and Chan's Algorithm, making Graham Scan least efficient for smaller values of h .

3.2 Random Points (Fig 3.2)

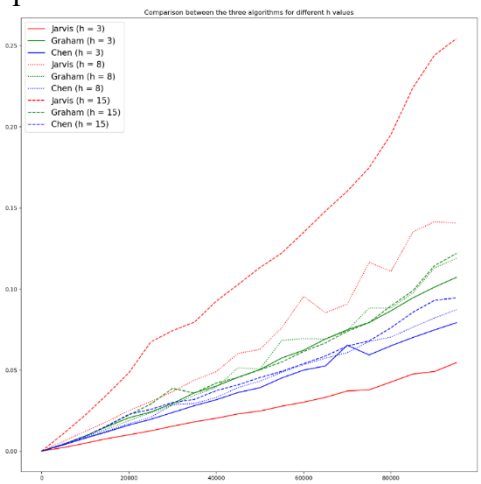
As the number of points increases when generating points randomly, the Chan's algorithm performs the best, while the Jarvis March algorithm the worst. The Jarvis march exhibits a sharp increase in time complexity as the n increase. Graham Scan shows a gradual increase which is more efficient than Jarvis March with larger dataset. The Chan's algorithm shows a relative flat as the time complexity is rely on both number of points and the number of convex hulls. It is better to use the Chan's algorithm when n goes up to extreme values for random points, as it outperforms both Graham Scan and Jarvis March based on the graph.

3.3 Circle Points (Fig 3.3)

When all the points form a circle, $(n = h)$, as n increases, Graham Scan algorithm performs the best, while Jarvis performs the worst. The time taken for Jarvis march increases quadratically and hence will take much longer time than the other two algorithms. For Chan's Algorithm, its time complexity gets affected by the number of points on convex hull and hence, as n increases, time will grow at faster rate than Graham Scan which is independent to h . Therefore, when all points are situated around a circle, Graham Scan is the best algorithm to be used.

3.4 Collinear Points (Fig 3.4)

When all the points are collinear, it was shown from the figure that Chan's Algorithm performs the worst out of the three while Jarvis March and Graham Scan algorithm performs similar. The difference between the time takes for Chan's Algorithm and time taken for Jarvis March or Graham Scan was significantly large. Based on the graph, Jarvis performs slightly more efficiently, thus it is better to use Jarvis March for collinear points.



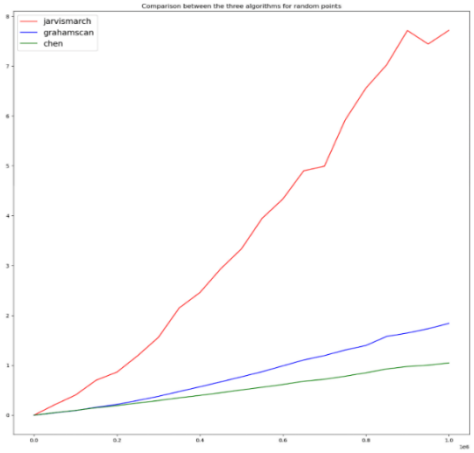


Fig 3.1 Visualisation of all algorithms for varying h

Fig 3.2 Visualisation of all algorithms when data inputs are random points

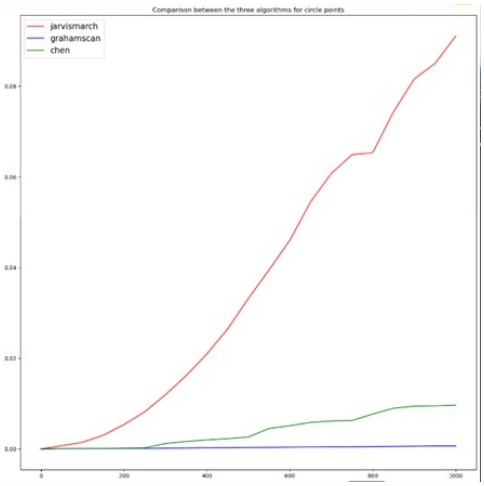


Fig 3.3 Visualisation of all algorithm performance when data inputs form a circle

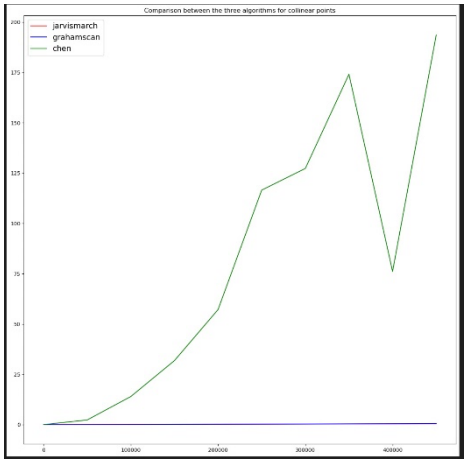


Fig 3.4 Visualisation of all algorithm performance when data inputs are collinear

4 Team Contributions

Student Name	**Student Portico ID**	**Key Contributions**	**Share of work**[1]
Declan Loo	23152800	Wrote implementations for Graham Scan, Jarvis March, and Chan's algorithm, also helped with the	28 %
Junwoo Lee	23156900	Authored the report sincerely and implemented the Test Data Generator. Ran the algorithm visualisations.	24 %
Cici Liu	23046190	Implemented the Experimental Framework. Ran the algorithm visualisations.	24 %
Runfeng Lin	23157804	Authored the report. Ran the algorithm visualisations.	24 %

[1] This should be a **percentage**. For example, in a group of 4 students, if all members contributed equally (i.e., the ideal scenario), their share of work would be 25% each.