COMP0005 Group Coursework
Academic Year 2023-24

Group Number: 12

1 Overview of Experimental Framework

1.1 Framework Design/Architecture

The experimental framework evaluates the runtime efficiency of Jarvis March, Graham Scan and Chan's Algorithm with varying number of points (n) and convex hull size (h), and performs the results with line charts.

- Framework Structure

The *ExperimentalFramework* class encapsulates the experiment setup, which takes seven parameters: four defining the ranges of n, h, x, and y for point generation, and three stating if the algorithm is tested for singular performance experiments. The framework supports fix n vary h, fix h vary n and n = h scenarios etc. By varying these parameters, simulations can cover the average and worst cases of each algorithm for comprehensive evaluations.

The imported *TestDataGenerator* class encapsulates the generation of points. The included methods support generating a convex hull with a fixed h value within the x and y range restricted by parameters and adding n-h random points in the convex hull.

- Experiment Execution and Visualisation

runExperiment() takes a parameter *count (= 1 by default)* and iterates n and h ranges, generates data, and measures the runtimes with *timeAlgorithms()*. *runWorstCase()* takes a parameter *count (= 1 by default)*, it runs experiments with h = n. The experiment procedure is repeated *count* times, and the mean values will be taken for results. Results are formatted as tuples (n, h, chans_time, graham_time, jarvis_time), and stored in instance variable *result*, a list, for visualisation and analysis.

The result can be visualised by *plotResult()* methods, which plot line charts of runtime against n for each h. Plot methods are called in *runExperiment()*. *results* can be printed with *printResult()* as a formatted table for precise values and mathematical analysis.

1.2 Hardware/Software Setup for Experimentation

The hardware required is simple. Any sort of computer (PC, laptop) with a non-outdated CPU will be suitable. For our experiment, we used a laptop with the AMD Ryzen 7 6800H CPU.

The software setup required for the experimentation would be Python and the necessary libraries to run this code, which are the 'timeit', 'random', 'math' and 'matplotlib' libraries. For 'matplotlib', it should be installed via pip. The command is "pip install matplotlib".

2 Performance Results

2.1 Jarvis March Algorithm

The 3D performance graph of the Jarvis March algorithm illustrates the relationship between the number of input points (n), the number of points on the convex hull (h) and the execution time of this algorithm. The x-axis represents n, the y-axis represents h, and the z-axis represents the execution time of the algorithm. This graph demonstrates that execution time increases with both n and h but more linearly with h. This visualisation helps in understanding the direct impact of hull size on performance, with colour gradients indicating longer execution times for larger values of n and h.

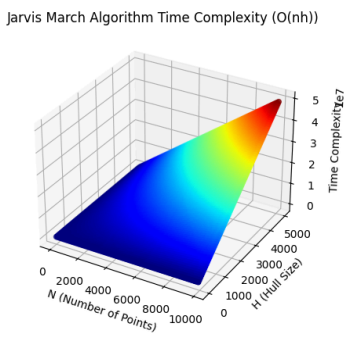

Figure 1: Expected Case

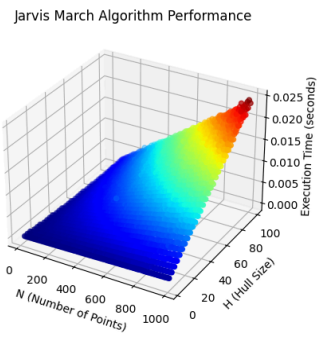

Figure 2: Experiment Small Values

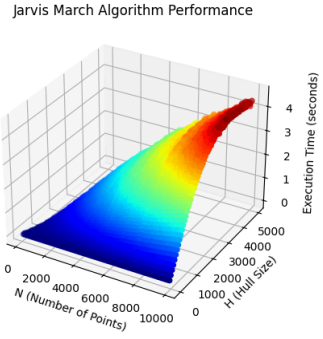

Figure 3: Experiment Huge Values

The time complexity of the Jarvis March algorithm is O(nh) which means its efficiency is directly related to both the total number of input points (n) and the number of points on the convex hull (h). This complexity indicates that the algorithm is suitable for datasets with a small h relative to n and it will meet the worst scenario when h tends to n. The execution time will depend linearly on h because each hull point requires scanning all n points to find the next hull point. Jarvis March has a space complexity of O(n) because it needs to store the set of points in the input and the convex hull points, which grows linearly with the number of points in the input.

For the average case, the average complexity can be considered as O(nh), with h being much smaller than n (then performance might be closer to linear). This is because the execution time is linearly affected by h.

The worst-case scenario occurs when h is very large and near to n (e.g. all points lie on the hull, n = h). Since the average complexity is O(nh), as h approaches to n, time complexity O(nh) tends to O(n^2). Thus, the worst complexity is O(n^2), as the algorithm must consider each point against every other point to construct the hull. This scenario provides the upper bound on the execution time.

To validate the theoretical complexity, we can do experiments by using various datasets with different n and h values. By plotting the 3D graph of the execution time against n and h and comparing the experimental results with the expected O(nh) trend, we can assess that the real-world performance aligns closely with theoretical predictions. By comparing these three graphs above, the experimental graph with a small sample (Figure 2) matches the theoretical graph (Figure 1) geometrically, but the experimental graph with a large sample (Figure 3) and the theoretical graph don't quite match. This is because all the points in our experiment are integers and the coordinates are no more than 32767, so h cannot reach a large number. Also, inaccuracy or error might arise due to some factors such as hardware performance, implementation details and data distribution. Overall, a linear relationship between the execution time with h could be observed, validating the O(nh) complexity.

2.2 Graham Scan Algorithm

The time complexity of the Graham Scan algorithm is O(nlog(n)) which means that only the total number of points (n) in the dataset affects the time taken to form the convex hull. The total number of points that form the convex hull (h) has no impact. This algorithm is split into 2 sections: point sorting and hull formation. In the initial point sorting section, points are sorted from the lowest polar angle to the greater polar angle. Our implementation of the algorithm uses Timsort, which has an average time complexity of O(nlog(n)). In the hull formation section, each point is traversed exactly once. Therefore the time complexity is O(n). Consequently, the Graham Scan algorithm has a time complexity of O(nlog(n)) due to sorting being the limiting factor. Graham's Scan algorithm also maintains a space complexity

of O(n), as it requires storing the sorted array of points by polar angle and the stack that holds the points of the convex hull, both of which depend linearly on the input size.

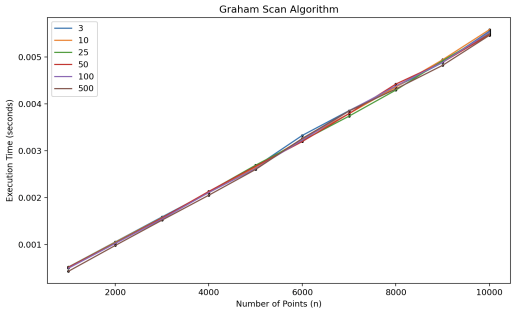

Figure 4: Graph of n with different h

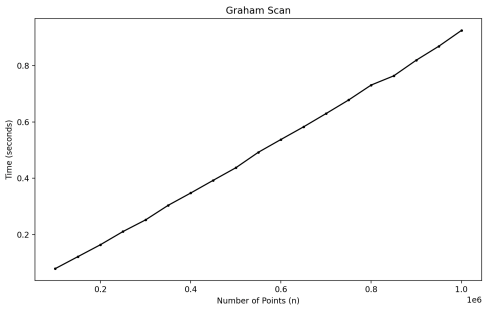

Figure 5: Graph of n with random points

Figure 4 illustrates the time taken (seconds) for the convex hull to form over the total number of points (n) across varied values of the points forming the convex hull (h). The time taken stays relatively the same throughout for different values for 'h', indicating that values for 'h' have no impact on the overall time complexity. The relationship between time and the number of points is almost linear, similar and resembles a graph of y = xlog(x).

Figure 5 shows the time taken (seconds) for convex hull formation across the total number of points (n) that are randomly arranged, representing the average case scenario. Similar to the previous graph, the relationship between time and the number of points is almost linear.

The average case scenario and worst case scenario for the Graham Scan algorithm are similar. The overall time complexity of the algorithm is dependent on the efficiency of the initial sorting algorithm. The Timsort algorithm has an average and worse-case time complexity of O(nlog(n)). However, we could use the Quicksort algorithm instead as it is more efficient for larger datasets leading to a more efficient Graham Scan.

2.3 Chan's Algorithm

Figure 8 is the 3D graph that illustrates Chan's Algorithm's performance in a larger range of n (number of points) and h (hull size). This illustrates the relationship between n, h and the time taken to execute the program. As we can see from the cross-section of the graph that takes h and time taken as the axes, it shows a log graph, which means the time complexity increases logarithmically as h increases. Also, when we see the cross-section

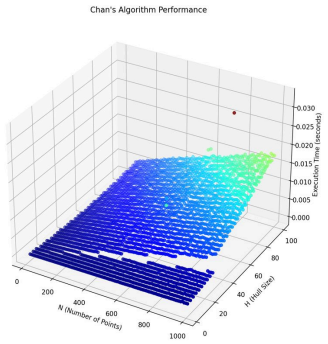

Figure 6 :Experiment Small Values

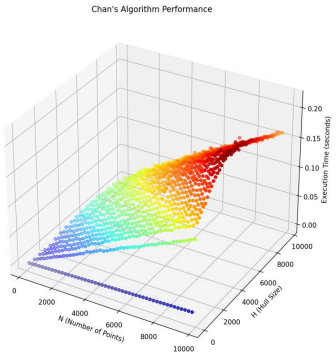

Figure 7 : Experiment Huge Values

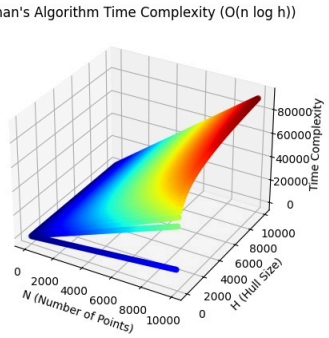

Figure 8: Expected Case

of the graph that takes n and time taken as the axes, it shows a linear graph, which means the time complexity increases linearly as n increases. As Chan's Algorithm's time complexity is O(n log h), the graph shows an accurate representation of the time complexity.

Both the average and worst-case scenarios' time complexities are O(n log h) for Chan's Algorithm. The worst case scenario happens when the h is too large compared to n, as the complexity is directly related to h. The average case has the same time complexity to the worst case as the algorithm's time complexity depends more on the nature of the dataset, such as how they are distributed, rather than the number of points in the dataset. As h affects the time complexity more than n, the average and worst cases' time complexities appear to be the same.

Jarvis march step is O($\frac{n}{m}h$ log m), and ideally, setting m = h gives the time complexity of n log h. This occurs because the Jarvis March step iterates through the points on the convex hull, and sets m to be the number of points on the convex hull ensuring that the time complexity scales with the size of the convex hull (h). Chan's Algorithm allows for flexibility in choosing the value of m and suggests starting with m = 2^{2^t} where t = 1 and increment t by one each step. In our implementation, we tried int(math.sqrt(n)) first as in most cases h is less than the square root of n. If this works in the first iteration, the algorithm will not have to iterate again to modify m and redo the partition.

Why are we using binary search? For each tangent found the time complexity is O(log m), O($\frac{n}{m}$) for each comparison and O($\frac{n}{m}$ log m) after comparison. It finds the most clockwise tangent as most_cw_tangent, which is guaranteed to be in the final hull.

We also use the 'Divide and Conquer' method. Each subset is computed in O(m log m) by Graham scan, considering that the total number of points in the input set is n and we are processing each point in subsets of size m, the number of subsets will be $\frac{n}{m}$. Therefore, the total time complexity of finding all the tangents from a single point is O($\frac{n}{m}$ log m) as mentioned above, and finding all points to the convex hull will be O($\frac{n}{m}$ h log m).

The space complexity of Chan's algorithms is O(n). In our implementation, the input points are stored in a list O(n), the total space complexity of calculating all subsets is no larger than O(n), the final hull is stored in a list and the complexity is O(n) as well, other than that no higher complexity auxiliary data structure is used. Therefore the overall space complexity is O(n).

3 Comparative Assessment

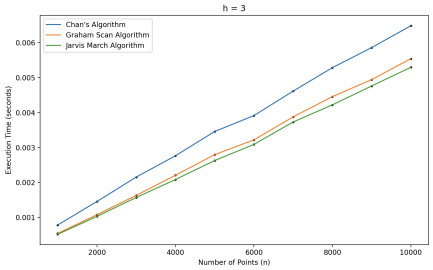

Figure 9 h=5 Graham

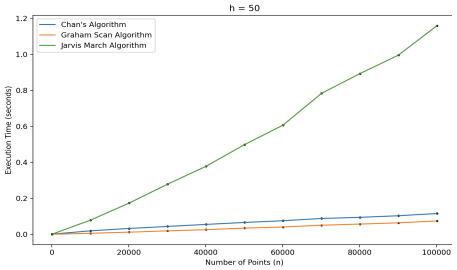

Figure 10: h=50 Graham

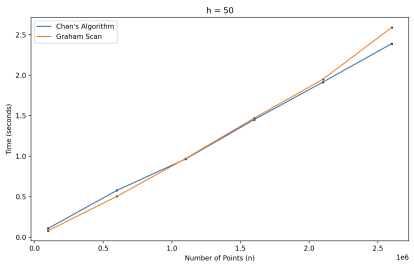

Figure 11: h=50 Graham

When the number of hull points (h) is low, Jarvis March is the most efficient algorithm because h < log(n) especially when the number of points (n) increases. Since Chan's algorithm implements Graham Scan, both time complexity is influenced by nlog(n). Once h increases, the efficiency of Jarvis March decreases linearly and the other two algorithms become more efficient. Chan's algorithm is slower than Graham Scan because h also influences Chan's. Eventually, when n becomes large enough, Chan's algorithm will be more efficient because the significance of O(h) on time complexity decreases as n becomes large. It is important to notice that as h increases, the point at which both algorithms cross

increases seen in Figure 11 above.

In the worst-case scenario when n = h, shown in Figure 12, the time taken to form the hull increases exponentially for Jarvis March because the time complexity would be $O(n^2)$. In theory, Graham Scan and Chan's algorithm should be very similar, however, Chan's algorithm runs slower due to many subsets being formed and merging them too. When n = h, it is best to use the Graham Scan algorithm as it closely follows a linear growth pattern.

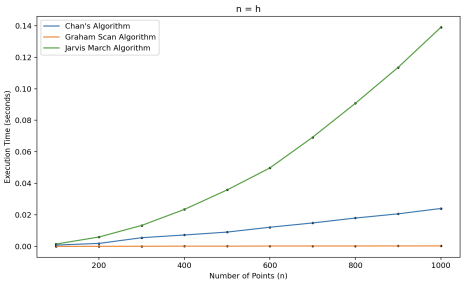

Figure 12

In many real-life cases, we will encounter the problem of having "small n and small h" or "big n and big h". In these "extreme" situations, we choose the proper algorithm to address the task efficiently and appropriately.

In the "small n and small h" scenarios, Jarvis March and Graham scans are suitable due to smaller overhead possibilities and easier implementation. Setup experiment: control n from 3 to 100, h from 3 to 10, run each algorithm 97×7 times, and repeat 1000 times to obtain the average. The average runtime of Jarvis March, Graham Scan and Chan's algorithm are 7.7831e-05, 4.868e-05 and 2.4055e-04 seconds respectively. Also, both Jarvis March and Graham Scan are easier to understand and implement, and combined with their good performance, they are preferred in scenarios such as robotics path planning.

In the "big n and big h" scenarios, Chan's algorithm outperforms, and Graham Scan performs the best when h is particularly large (h ~ n). The average runtime of Jarvis March, Graham Scan and Chan's algorithm in the experiment where n ranges from 100000 to 1000000 (step = 100000) and h ranges from 50 to 200 (step = 50) are 1.475, 0.0675, and 0.0615 seconds. As analysed above, the complexity of Chan's algorithm returns to O(n log n) when n=h, but because operations like partition are time-consuming, its actual runtime is greater than the Graham scan. In fields such as 3D modelling and image processing, Chan's algorithm and Graham Scan are more preferred.

Overall, the most effective algorithm is different in different situations. For small n with small h, as previous paragraphs discussed, generally, Graham Scan is the most effective algorithm, but for some special cases (e.g. h < 5), Jarvis March is better than the other two. For large n (n > 10000) with small h (h < 20), generally, Chan's algorithm is the most effective, but Jarvis March is better than the other two when h = 3. For large n with small h (20 < h < 50), Graham Scan is the best one but Chan's algorithm becomes faster when n is large enough (e.g. 5e6). For large n with large h (h > 50), Chan's algorithm outperforms, and Graham scan performs the best when h is particularly large (h ~ n).

[1] "Generating Random Convex Polygons," cglab.ca. *https://cglab.ca/~sander/misc/ConvexGeneration/convex.html (accessed Mar. 06, 2024).*
[2] Tak Hang Chan, "Optimal output-sensitive convex hull algorithms in two and three dimensions," vol. 16, no. 4, pp. 361–368, Apr. 1996, doi: https://doi.org/10.1007/bf02712873.

4 Team Contributions

Wing Ho Yeung (22006533) **25%** - Graham Scan (coding, report), Experimental Framework (coding), Comparative Assessment (report)
Aowei Zhang (23110228) **25%** - Graham Scan (coding), Chan's (coding, report), Experimental Framework (coding, report), Comparative Assessment (report)
Cindy Yun (23166290) **25%** - Chan's (coding, report), Hardware & Software (report), Graph Generation, Reformatting (code, report)
Meihui Zhao (23010321) **25%** - Jarvis March (coding, report), Data generation (coding), Comparative Assessment (report), Graph Generation