COMP0005 Group Coursework

Academic Year 2023-24

Group Number: 19

1 Overview of Experimental Framework

1.1 Framework Design/Architecture

1.1.1 Introduction to Design

In the test system, we include two main classes: TestDataGenerator, and ExperimentalFramework, one is for generating data, and the other for performance testing.

1.1.2 Test Data Generator

The *TestDataGenerator* class provides functionality for generating synthetic test data sets, which allows for repeated testing in ExperimentalFramework.

Class attributes:

- ***self.maxInt = 32767*** - represents the maximum coordinate value allowed for the coursework. All the data generator methods below will incorporate this.

Data generation methods:

- ***getRandData(self, n)*** - that can produce random data points with (x, y) integer values, where **n** is the total number of points. Generated example points are shown in **Figure 1**.

- ***getNHData(n)*** - generates the "worst-case" data sets for the Jarvis March Algorithm, where **n = h**. It's done by creating a circle of points, ensuring that all points are part of the convex hull. Due to the limitations of the design, data generated for **n = ~h** is only accurate up to n=1000. Generated example points are shown in **Figure 2**.

- ***GetChanData(n,h)*** - generates data used for the chan's algorithm test, where **h** is controlled for testing. This is used to demonstrate the average case scenario of **nlog(h)** for chan's algorithm. It's done by first generating a circle of points of size **h**, then generate **n – h** points within the circle. Generated example points are shown in **Figure 3**.

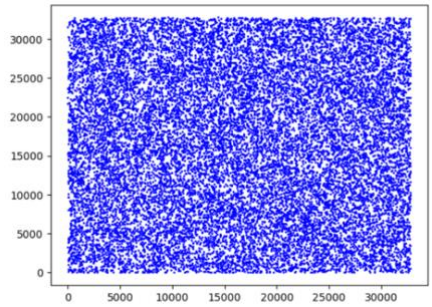

Figure 1: getRandData()

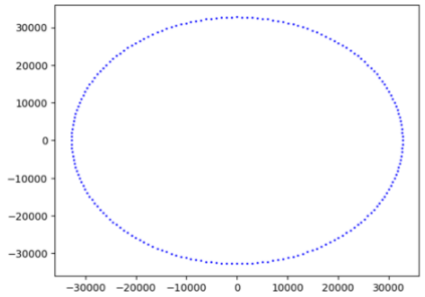

Figure 2: getNHData()

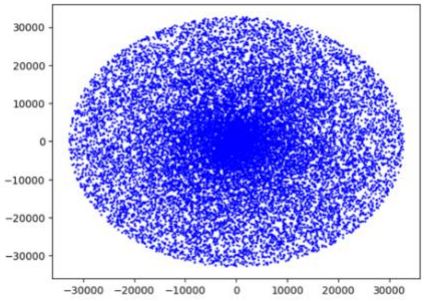

Figure 3: getChanData()

1.1.3 Experimental Framework

The *ExperimentalFramework* class provides a structured way to test and compare the performance of different algorithms. Test datasets of varying sizes are first generated from the *TestDataGenerator* class, then methods will be used to run the algorithms and measure the time taken. In the end, plotting graphs to show the performance.

The key methods:

- ***measureTime(self, algorithm, data, m=None)*** - Runs the given algorithm function on the input data set and returning the runtime measured. The parameter m is used for functions which require an additional parameter.

- ***formatNShow(self), formatNShow2(self)*** - Auxiliary functions used for plotting. it reduces code repetition.

- ***runAvgTest(self, numTest)*** - This runs the test which represents the average case for all the algorithms. It makes use of the *getRandData()* method for data generation. The algorithms are running *numTest* times, each time with varying *numPoints* (number of points), which is incremented by *step*. For each *numPoints*, the data are generated and algorithms are ran *repeatNum* times. This is to ensure a more stable and accurate measured time by taking the average. Finally, it plots a graph based on the data *numData* and *timeXX* (x-axis and y-axis respectively)

- ***runNHTest(self, numTest), runChanTest(self, numTest), runStbTest(self, numTest), runGSTest(self, numTest)*** - Similar to *runAvgTest()*, but using different test data and algorithms. However, some tests do not run *repeatNum* times as it is unnecessary.

1.2 Hardware/Software Setup for Experimentation

The Jupyter Notebook for implementing and evaluating the computational algorithm framework utilizes the following experimental hardware/software stack:

Hardware:

- 2.4 GHz 8-Core Intel Core i9
- AMD Radeon Pro 5500M 8GB GPU
- 32GB 2667 MHz DDR4 RAM

Software:

- macOS 14.3 (23D56)
- Python 3.8 (Anaconda)
- Jupyter Notebook Server 6.4.12

2 Performance Results

2.1 Jarvis March algorithm

Jarvis March exhibited reasonable efficiency when handling random data from *getRandData()* in **Figure 4**, with a linear growth in runtime **O(nh)**, where **h** = number of points on the hull. This suggests that the algorithm can be used to handle small datasets if efficiency is not the priority.

However, the algorithm faces challenges when **h** is large. In the worst-case scenario shown in **Figure 4**, where **n = h,** the time complexity becomes **O(n^2)** and the runtime grows exponentially. It takes **200ms+** to compute the convex hull at **n = 1000** compared to the average case of **40ms~** at **n = 25000**. Although this is an extreme case and most realistic datasets have a relatively small **h**, Jarvis March's inability to deal with a larger **h** is its biggest flaw.

Another problem with the Jarvis March Algorithm lies in its inconsistency in performance. This will be mentioned again in section 3, but it is already made apparent with **Figure 7**. Despite taking measurements repeatedly with the same **n** in *getRandData()* for better stability, the line is still very unstable.

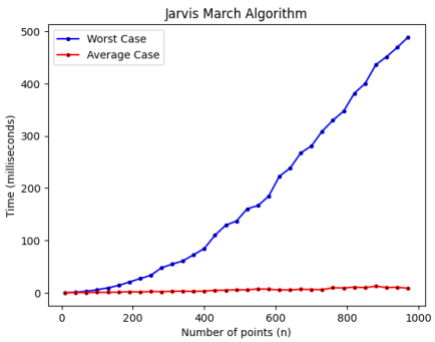

Figure 4: runJMTest: Jarvis March Test

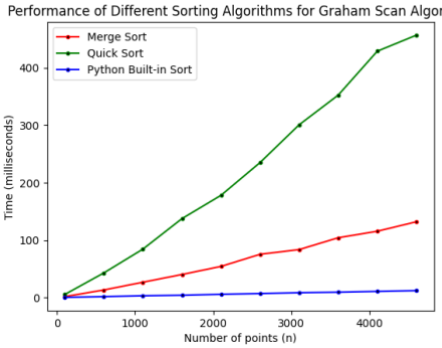

Figure 5: runGSTest(): Graham Scan Test

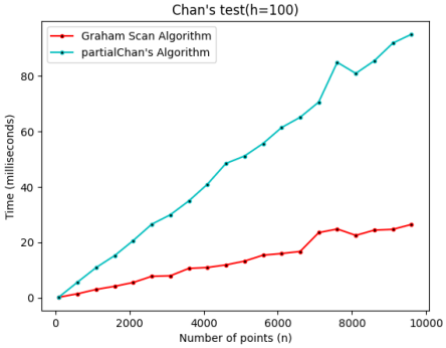

Figure 6: runChanTest(): Chan's Test

2.2 Graham Scan algorithm

In this report, unless otherwise specified, the sorting algorithm utilized for the Graham Scan algorithm is assumed to be the Python built-in sorting function.

Graham Scan is an algorithm with computational time heavily dependent on how it sorts collections of 2D coordinates instead of the characteristics of the points themselves. Therefore, Graham Scan always perform at the average case **O(nlog(n))** as shown in **Figure 7,8**, due to the nature of those tests.

In **Figure 5**, three sorting algorithms are compared: Merge Sort, Quick Sort, and Python's Built-in Sort, with respect to the time they take to sort points as part of the Graham Scan algorithm, which is used for convex hull finding. The Python Built-in Sort performs the best, exhibiting a nearly constant, linear time complexity, which is indicative of the efficient implementation of the Timsort algorithm in Python that adapts to various data patterns.

The Merge Sort demonstrates a stable increase in time with the increase in the number of points, maintaining a consistent time complexity that reflects its typical **O(nlogn)** behaviour.
Quick Sort shows a similar trend but with occasional spikes which may indicate certain data configurations causing the algorithm to perform sub optimally, approaching its worst-case performance of **O(n^2)**. As a result, this Quick Sort variation demonstrates the worst-case scenario of Graham Scan which is **O(n^2)**.

2.3 Chan's algorithm

Upon implementing Chan's algorithm, its performance is somehow even slower than Jarvis March for the average Test with random data as shown in **Figure 7**. While it is not completely understood by us on why this has occurred, this part will discuss potential causes:

1. **Repeated Computation Attempts:** in the implementation of Chan's algorithm, *partialChan(inputSet, m)* is the part which handles the computation of the convex hull while *Chan()* repeatedly calls *partialChan()* with different **m** values until a complete convex hull is found. The nature of repeatedly computing the hull increase runtime.
2. **Too Many Function Calls:** as python is an interpreted language, the action of calling a function is relatively costly. Within *chan(),* the auxiliary function *rTangent()* calls *turn()* many times, which causes a time overhead.
3. **Graham Scan Algorithm Efficiency:** as mentioned before, the built-in sort() function in python is very well-optimized. The performance difference between calling *grahamscan(n=100)* vs *grahamscan(n=10000*) is minimal. Chan's algorithm calls *grahamscan()* numerous times which is observed to be costly relative to other algorithms.

To demonstrate the real average case of chan's algorithm **O(nlog(h))**, we developed "Chan's test" utilizing *getChanData()* to generate a dataset with constant **h**. It is currently set at **h=100**, but different **h** values seem to exhibit the same behaviour. It is expected that by directly using the correct **h** in *partialChan()*, it would optimally compute the convex hull. However, as shown in **Figure 6**, it still is not possible due to the first 2 reasons above.

However, **Figure 7** demonstrates the worst case scenario of **O(nlog(n))**. This happens when **m** is too big, causing the Graham Scan part of Chan's algorithm to dominate. This is triggered due to directly using *partialChan(inputSet, m)* to compute the hull while setting **m** as very large in *runAvgTest()*. Therefore, the graph shows that the linear growth between Graham Scan and Chan's algorithm are nearly identical.

3 Comparative Assessment

3.1 Experimental Performance

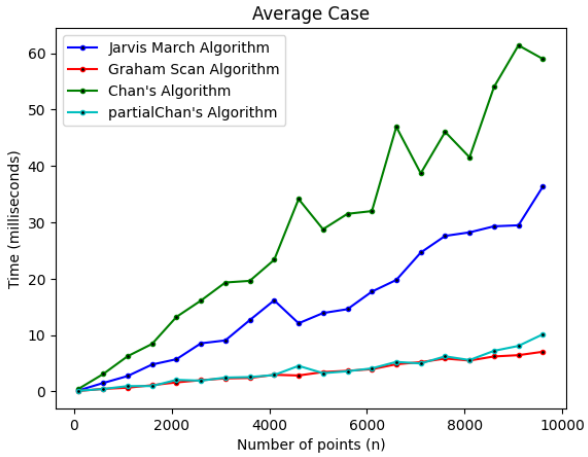

Figure 7: runAvgTest(): The average Test

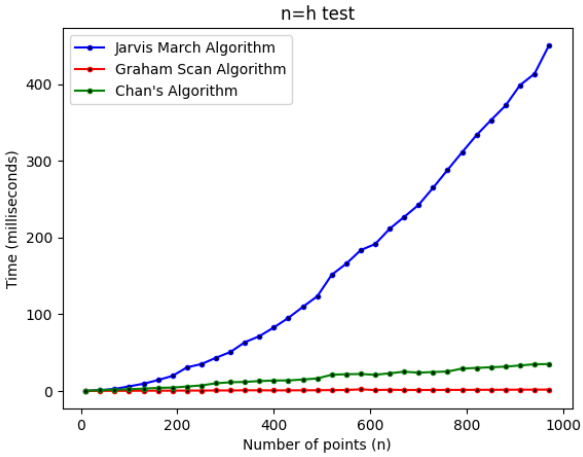

Figure 8: runNHTest(): N = H Test

3.1.1 Average-Case

As shown in **Figure 7**, Jarvis March performs with **O(nh)** and the difference in performance against Graham Scan grows as **n** grows larger, which is to be expected.
Graham Scan demonstrated the best performance with **O(nlog(n))**. Graham Scan exhibited consistent performance across different datasets due to its natural emphasis in sorting. Its runtime ends up always being very consistent and efficient.
While Chan's algorithm should theoretically have the best average case performance, it could not be showcased with our implementation unfortunately.
Based on the experimental results in the average case, the ranking of the algorithms from fastest to slowest is:

Graham Scan Algorithm > Jarvis March Algorithm > Chan's Algorithm

3.1.2 Worst-Case

As can be seen in **Figure 8**, in the worst-case scenario for Jarvis March, where **n = h**, it performs with **O(n^2)**. Although the worst-case time complexity for Graham Scan is also **O(n^2)**, it is easily avoidable unlike Jarvis March. This is done by simply not using sorting algorithms such as Quick Sort.

For Chan's Algorithm, its worst-case time complexity **O(nlog(n))** is exhibited by letting Graham Scan dominate the algorithm. Therefore, the worst-case performance of Chan's algorithm is almost the same as the average case performance of Graham Scan.

Based on the experimental results in the worst-case, the ranking of the algorithms from fastest to slowest is:

Chan's Algorithm > Graham Scan Algorithm > Jarvis March Algorithm

3.2 Stability of Performance

The stability of performance refers to how consistent an algorithm's runtime is across different input characteristics. The number of points is constant at **n = 1000** with *getRandData().*
The big spike seen in the graph is likely due to external factors.
In **Figure 9**, Graham Scan demonstrated the most consistent performance.
In **Figure 10**, Chan's algorithm exhibited good stability closer to Graham Scan.
In **Figure 11**, Jarvis March is observed to be less stable with very visible "wobbles" throughout.

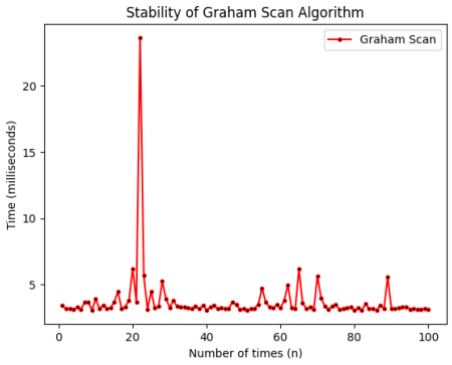

Figure 9: Stability of Graham Scan Algorithm

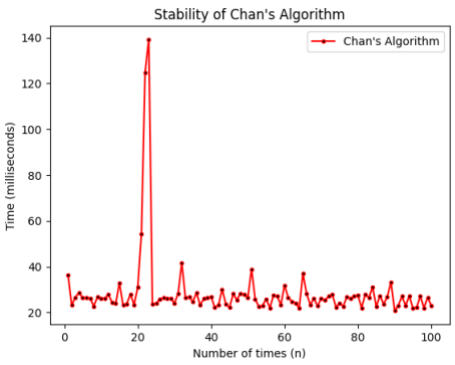

Figure 10: Stability of Chan's Algorithm

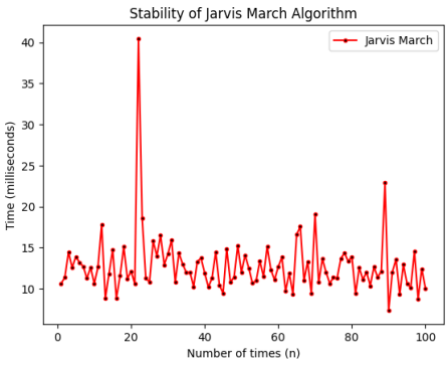

Figure 11: Stability of Jarvis March Algorithm

Overall, in terms of performance stability, the algorithms can be ranked as follows:

Graham Scan Algorithm > Chan's Algorithm > Jarvis March Algorithm

3.3 Conclusion

In conclusion, the experimental performance analysis, provides a comprehensive comparison of Jarvis March, Graham Scan, and Chan's Algorithm. Showing the average and worst case performances of all the algorithms, as well as discuss the situation which triggers them.

It is unfortunate however that our implementation of Chan's Algorithm did not perform as expected in terms of runtime. Although theoretically not the case, Graham Scan is the fastest, most consistent and most suitable algorithm in all cases. Jarvis March being the simplest, performed the worst overall.

4 Team Contributions

Student Name	**Student Portico ID**	**Key Contributions**	**Share of work[1]**
Aiden Li	23002328	Split the work with Ryan Li equally and focused more on Graham Scan and report.	45 %
Ryan Li	23161748	Split the work with Aiden Li equally, focused more on Jarvis March and Chan's algorithm.	45 %
Alicia Mak	22124895	Attempted to implement Chan's and write report section 2.	10 %
Zikai Li	22124895		0%

[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.