COMP0005 Algorithms

Convex Hull Analysis

Andrzej Szablewski Cheng Loo Orhun Dogan Vishaol Vignaraja

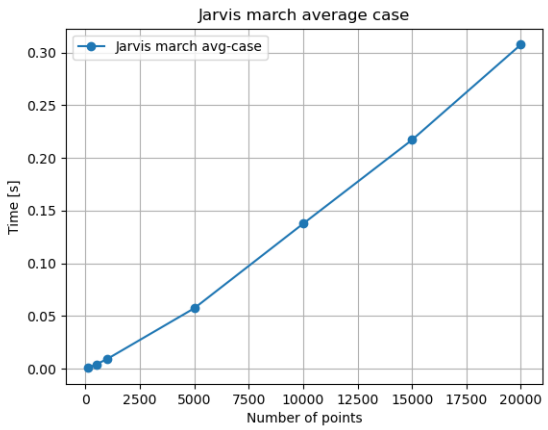
1 Jarvis March (Gift Wrapping) Algorithm

This algorithm first selects a starting point on the edge, then calculates the interior angle between the starting point and all the other points. The point that has the largest interior angle is then part of the convex hull. The entire set of N points is then cycled through until it returns to the starting point. For the purpose of this experiment, we use the set of $N = [100, 500, 1000, 5000, 10000, 15000, 20000]$.

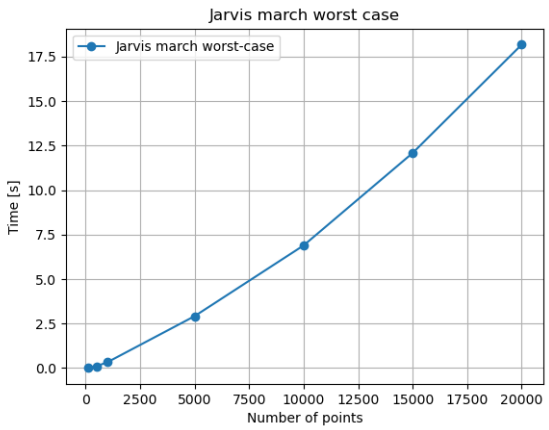
1.1 Random Input

In theory: When the points are randomly distributed, the average time complexity will be **O(NH)**, where **N** is the number of points and **H** is the number of convex hull vertices. This is due to the fact that the Jarvis March algorithm spends **O(N)** time on each convex hull vertex.

Experimentally: From running our experiments it was clear that as the number of points increased there were a lot of movements in the time taken but on average it indeed did contain a time complexity of **O(NH)**. Using Figure 1a, we can draw a line of best fit further demonstrating the linear relationship between the time taken and the number of points, and the complexity turns out to be more similar to **O(N)**, as **H** turns out to be sufficiently small.



(a) Jarvis March algorithm average case



(b) Jarvis March algorithm worst case

Figure 1: Jarvis March algorithm performance

1.2 Worst-case Scenario

In theory: The algorithm runs a function that loops through every point of the convex hull, hence if every point of the input set is also a point of the convex hull, and the complexity for the random case is **O(NH)**, **N** will be equal to **H**. The time complexity thus turns out to be **O($\mathbf{N^2}$)**.

Experimentally: Using the set of points N, we have plotted a graph of time against the number of points according to Figure 1b. The graph clearly shows that in the worst-case scenario, the graph increases in a quadratic-like manner, or $\mathbf{O(N^2)}$. While more points are needed to prove that it is of $\mathbf{O(N^2)}$ complexity, this gives a good approximation of the efficiency of the code in its worst-case scenario.

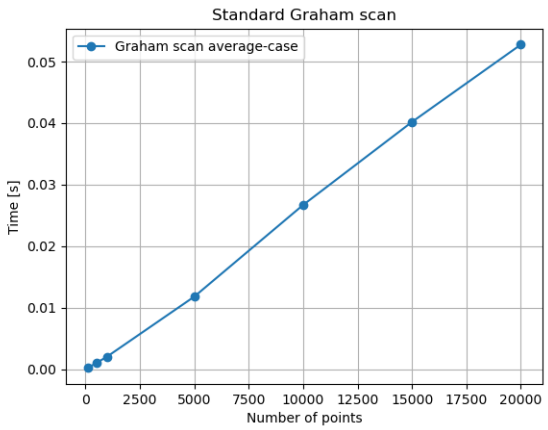
2 Graham Scan Algorithm

This algorithm finds the bottom most Y coordinate (if more than 1 found, it takes the one with the leftmost X coordinate) where this process will contain a time complexity of $\mathbf{O(N)}$. However once this point is found, the rest of the points would need to be sorted using a sort algorithm into a data structure, like a list. The points would need to be sorted with respect to the magnitude of the angle that is created between the first bottom most point and the X-axis, ordering them in an increasing fashion. The convex hull is produced when the algorithm traces all the sorted points. If the largest sorted point makes the convex hull turn counter-clockwise, the point is taken off the list, is added to the stack of convex hull points, and the next point would be traced. However, if the largest sorted point turns clockwise, then the algorithm pops the previous point from the list and continues checking the current largest point with remaining points on the top of the convex hull stack.

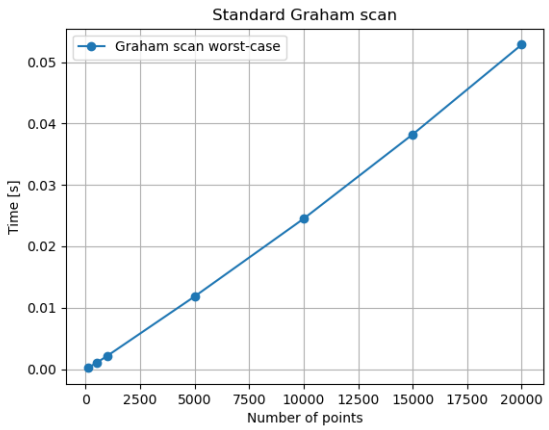
2.1 Random Input

In theory: For a set of randomly generated points, as the computation of the convex hull has minimal effect on the time complexity compared to the time complexity of the sorting algorithm, the time complexity of the graham scan on average is equivalent to the time complexity of the sorting algorithm, $\mathbf{O(N \log N)}$.

Experimentally: In order to experimentally determine the time complexity of the Graham Scan algorithm we had to determine a sorting algorithm to proceed further, hence for this experiment we decided to use the Timsort algorithm, which is a mixture of the Merge sort and Insertion sort. This has a time complexity of $\mathbf{O(N \log N)}$. However from Figure 2a it is evident that the time complexity is $\mathbf{O(N)}$. The reason for this we feel is due to the small number of points plotted only demonstrated this behaviour. With hardware limitation, producing this result took a lot of resources and time but we are confident with more points and better hardware it would be possible to produce a graph which demonstrated the behaviour of $\mathbf{O(N \log N)}$.



(a) Graham scan algorithm average case



(b) Graham scan algorithm worst case

Figure 2: Graham scan algorithm performance

2.2 Worst-case Scenario

In theory: As mentioned before, as the time to sort the points trumps the time to compute the convex hull, the worst-case for the Graham Scan is equivalent to the worst-case for the sorting algorithm, $\mathbf{O(N \log N)}$.

Experimentally: Since the sorting algorithm we use, Timsort, has its worst-case time complexity also set at $\mathbf{O(N \log N)}$), the points that we set only marginally affect the time taken to sort. This is shown by the graph in Figure 2b, which is very similar to the graph in Figure 2a. Due to the small number of points, the time complexity therefore stands more similar to $\mathbf{O(N)}$.

3 Extended Graham scan algorithm

3.1 Introduction

The complexity of the Graham scan algorithm depends on the sorting algorithm and generally (while using $\mathbf{O(N \log N)}$ sorting algorithm) can be expressed as $\mathbf{aN + bN \cdot log(cN)}$, where **a, b, c** are constants. Thus, we want to decrease the coefficients (**b** and **c**) and the idea for this is to eliminate the majority of the points before the sorting stage. The goal is to develop such a heuristic, which will have possibly minimal linear complexity, drastically decreasing **b, c** and slightly increasing the coefficient **a**.

3.2 Heuristic

To improve Graham scan, we have implemented the following heuristic. Firstly, it finds the 4 points that have the highest and lowest, X and Y coordinates respectively. The smallest and largest sums and differences of the X and Y coordinates are the next 4 points to be found. These 8 points are linked up to form a convex polygon (very often an octagon), and those points within are deemed to be not on the convex hull and are removed. The challenge here is to efficiently determine which points are inside a polygon and which are not.

3.2.1 Determining whether a point is in a polygon

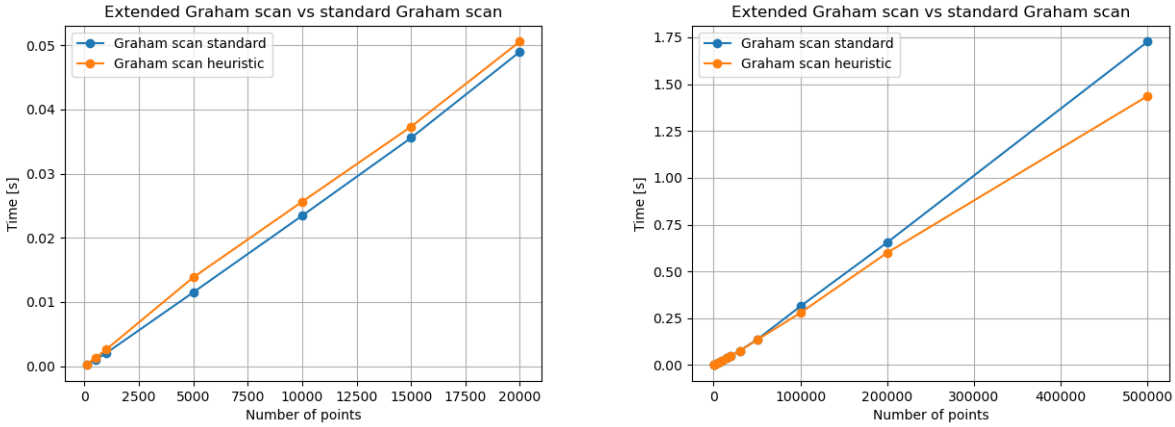
The obvious and probably the easiest way to determine the relation between a point and a polygon is to use the same method as the one used in the graham scan algorithm: calculating the vector cross product between a point and every edge of the polygon. However, this naïve approach may not be the best way, since most of the points are inside the polygon and calculating the cross product requires exactly 2 multiplication and 1 comparison operations for every point for each polygon vertex (multiplication is considered costly in terms of the time complexity compared to e.g. addition, subtraction). Hence, we decided to implement a more efficient method based on linear inequalities. Firstly, we determine linear equations for lines including each polygon edge. Then, we check the relations between every point and every line. This requires only 1 multiplication, 1 addition and 1 comparison for every point for every line (number of lines and number of vertices is the same), which should improve the algorithm performance both theoretically and practically.

3.3 Random Input

In theory: Heuristic complexity is linear ($\mathbf{O(N)}$ for finding corners, $\mathbf{O(N)}$ for removing points). We have experimentally shown that on the random input we can eliminate over 98% of 20,000 points and drastically lower the sorting time. Although we have deleted many points, the remaining points still need to be sorted so the time complexity of the extended graham scan will be still $\mathbf{O(N \log N)}$.

Experimentally: After gathering results from the experiments, the extended Graham scan seems to be working even worse than the standard one. The improvement is barely visible in the range of $100 - 20,000$ points, however, it is visible after reaching about $50,000$ points. The reason for such little improvement in range $100 - 20,000$ is that the standard Graham scan algorithm is already well implemented using efficient sorting mechanisms and $20,000$ points, which is roughly 10^4 in terms of the order of magnitude, is too small a number to show the actual improvement. However

with the heuristic of removing points that were certainly not in the convex hull we are confident that the algorithm performs better for larger numbers of points and it still has $\mathbf{O(N \log N)}$ complexity.



(a) Extended Graham up to 20,000 points

(b) Extended Graham up to 500,000 points

Figure 3: Extended Graham scan algorithm performance

Improvements: Overall it seems that there have been marginal improvements from the original Graham scan. However from Figure 3b, it demonstrates that the more points the algorithms has, the extended Graham scan tends to get to use less time compared to the original Graham scan algorithm. Hence the heuristics do have a substantial improvement to the algorithm, however results are only seen with greater number of points.

3.4 Worst-case Scenario

Fundamentally, our heuristic implemented simply removed certain points. However, if it does not find points to remove (e.g. **all points are collinear**) it still does additional passes through all points, which increases time complexity.

Improvements: The removal of points reduces the time taken to sort the points. While there are no points to remove, the algorithm still performs additional checking. According to Figure 4, it shows that there is a slightly visible deterioration in the time taken with the heuristic implemented.

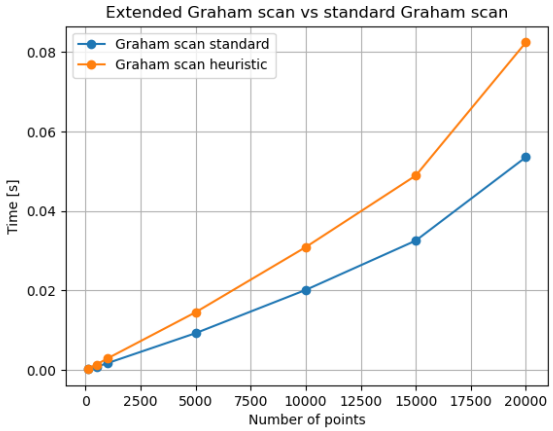


Figure 4: Extended Graham scan worst case performance

Note: All the testing was performed on a notebook running Windows 10, 16GB of RAM, i7-8750H CPU.