**COSC 262 – Convex Hulls**

1. **Algorithm Implementation:**

* Explain your program this includes the main algorithms, the sub-functions used to help those algorithms and anything you found useful for your algorithms
* Explain your additional algorithm (Monotone Chain) and how it works.

Throughout this assignment we are told to program three algorithms in relation to computing convex hulls. A convex hull of a graph is all the outermost vertices of that graph. There are several ways to achieve this by a computer program, the algorithms I have covered in the Gift Wrapping algorithm, Graham Scan and Monotone Chain. Although I also had a go at Chan’s algorithm but couldn’t get correct output.

The Gift Wrapping algorithm works by which works by starting with the lowest vertex on the y-axis to start and then calculates the angle between that vertex and all other vertices to find the rightmost vertex, which it then uses to calculate the next vertex and so on until we create a complete convex hull. This algorithm is one of the most straightforward algorithms but in the also is one of the most inefficient way of computing a convex hull.

The Graham Scan starts the convex hull computations much like the other one, by choosing the min vertex on the y-axis to start its calculations. From there on it checks that vertex with the other vertices of the graph sorted by the angle between that vertex and the rest. The program then checks if the vertex that is it comparing with is clockwise or counter-clockwise to comparison vertex. If a vertex is counter-clockwise then it will add that vertex to the convex hull, although if a vertex is clockwise then the previous vertex must not be a part of the convex hull and thus removes that vertex. The algorithms continue in this manner until the complete convex hull is formed. The implementation of this algorithm isn’t too difficult whilst also returning fast results at a time complexity of O(n log n).

And finally, we have the Monotone Chain which creates a convex hull by firstly ordering its vertices by their x-coordinate and then splitting the graph into two sections and running two loops, one for the upper-half of the graph and one for the lower-half of the graph. In each loop it checks the angles much like the Graham Scan. Removing clockwise vertices and adding counter-clockwise vertices. It then combines both sub-hulls together to form the complete convex hull. To my surprise this was one of the easiest algorithms I had created for convex hulls and with that it also has a good time complexity, one close to what the Graham Scan can get up to O(n log n).

Within my python file you will find I also tried to implement the Chan’s algorithm I originally wanted to implement this as it ran in O(n log h) time where n is the number of vertices and h is the number of convex hull vertices. The Chan’s algorithm works by splitting the graph into sub-graphs of size m (m = 2^2^**i**, where **i** is the index number of the primary function) then runs a Graham Scan on each of those sets to create sub-hulls of our sub-sets. After we have received the sub-hubs we then want to find the extreme points of those sub-hulls in comparison to our minimum point and its angle. We do this by using a special case of the Gift Wrapping algorithm. By the end of this algorithm we should have a complete convex hull, although if a convex hull was not created or the algorithms took too long the nit would break and increment the magic number by increasing the value of **i** by 1. I had several attempts at creating the Chan’s algorithm but my result never seemed to make it. The first half of my program was working fine where it creates sub-hulls and runs a Graham Scan, but as soon as I needed to start calculating critical points my program stopped working. There also there aren’t many implementations of Chan’s algorithm out there on the web and the pseudo code wasn’t very helpful either so I changed my algorithm to the Monotone Chain.

1. **Algorithm Analysis:**

* Preform a set of comparisons between all your algorithms, showing how fast they each were to work on each problem and explaining their complexity.
* How did the results vary for each algorithm? Unexpected variations? Similarities?
* Which algorithm gave the best result? Complexity?

To test the complexity of our algorithms I created a separate python file (convexhulls\_time.py). Within this python file I created a series of tests to run on the algorithms I have created. We were given twenty different test cases to trial on, ten of which are a part of set-A and ten are a part of set-B. Each problem starts off easy and becomes sequentially harder as the number of vertices and convex hull vertices go up. To test an algorithm’s running time in python I started off by importing a couple modules to help, glob and time. Glob just creates a list containing all the test files under a directory, I split these files into to lists one for set-A and one for set-B. I then ran a call to an algorithm test, recorded the start time, tested set-A, record end time, then write to a CSV file with the results. Then through the same process tested for set-B and for the following algorithms. All outputting their total run time into a CSV. The order of which is shown below:

Gift Wrapping Algorithm

🡪 Test set-A

🡪 Test set-B

Graham Scan Algorithm

🡪 Test set-A

🡪 Test set-B

Monotone Chain Algorithm

🡪 Test set-A

🡪 Test set-B

Rinse and repeat for five tests.

These following tests can be seen in the three following tables below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Gift Wrapping** | | | | | | |
| **Name** | **Time, Run 1** | **Time, Run 2** | **Time, Run 3** | **Time, Run 4** | **Time, Run 5** | **Average** |
| **A\_3000.dat** | 0.011970043 | 0.017226219 | 0.011431217 | 0.022658348 | 0.036580801 | 0.019973326 |
| **A\_6000.dat** | 0.022940636 | 0.034285069 | 0.050323725 | 0.045897961 | 0.061262369 | 0.042941952 |
| **A\_9000.dat** | 0.039928436 | 0.039721489 | 0.064706326 | 0.062961102 | 0.069992542 | 0.055461979 |
| **A\_12000.dat** | 0.054818392 | 0.068067074 | 0.100920677 | 0.098943472 | 0.106305361 | 0.085810995 |
| **A\_15000.dat** | 0.066820145 | 0.066458941 | 0.120938778 | 0.088796616 | 0.133679628 | 0.095338821 |
| **A\_18000.dat** | 0.080827951 | 0.098746300 | 0.163236141 | 0.111293316 | 0.178956747 | 0.126612091 |
| **A\_21000.dat** | 0.093725681 | 0.101514578 | 0.149610519 | 0.149297714 | 0.101008415 | 0.119031382 |
| **A\_24000.dat** | 0.123649359 | 0.158179283 | 0.165965796 | 0.211520672 | 0.190931559 | 0.170049334 |
| **A\_27000.dat** | 0.145652056 | 0.250208139 | 0.251326561 | 0.288501024 | 0.200167894 | 0.227171135 |
| **A\_30000.dat** | 0.155543089 | 0.299271822 | 0.184167147 | 0.210258961 | 0.309299707 | 0.231708145 |
| **B\_3000.dat** | 0.073802471 | 0.082775116 | 0.124733925 | 0.102184534 | 0.114027977 | 0.099504805 |
| **B\_6000.dat** | 0.387422085 | 0.480361700 | 0.434331656 | 0.427898645 | 0.420109272 | 0.430024672 |
| **B\_9000.dat** | 0.595429420 | 0.746964216 | 0.831114531 | 0.920100451 | 0.850023746 | 0.788726473 |
| **B\_12000.dat** | 1.208276033 | 1.536751986 | 1.416501999 | 1.471173525 | 1.732015133 | 1.472943735 |
| **B\_15000.dat** | 1.947947264 | 2.286554575 | 2.436725140 | 2.749589682 | 2.725796223 | 2.429322577 |
| **B\_18000.dat** | 4.438503981 | 3.394481897 | 3.665191412 | 3.388609886 | 3.920180321 | 3.761393499 |
| **B\_21000.dat** | 3.404277086 | 5.100181103 | 5.420728207 | 5.702098608 | 5.251897812 | 4.975836563 |
| **B\_24000.dat** | 4.203809023 | 6.183103561 | 8.492543459 | 6.870060682 | 6.632013321 | 6.476306009 |
| **B\_27000.dat** | 5.275710583 | 8.498424053 | 16.397583246 | 8.893428564 | 9.037980556 | 9.620625401 |
| **B\_30000.dat** | 6.966216803 | 10.090911627 | 11.330366611 | 10.716295958 | 10.892482519 | 9.999254704 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Graham Scan** | | | | | | |
| **Name** | **Time, Run 1** | **Time, Run 2** | **Time, Run 3** | **Time, Run 4** | **Time, Run 5** | **Average** |
| **A\_3000.dat** | 0.020941019 | 0.056545019 | 0.016984940 | 0.039939165 | 0.053866148 | 0.037655258 |
| **A\_6000.dat** | 0.047872543 | 0.068066597 | 0.083578348 | 0.079271317 | 0.060822487 | 0.067922258 |
| **A\_9000.dat** | 0.046874046 | 0.131992817 | 0.115377903 | 0.121638775 | 0.040980101 | 0.091372728 |
| **A\_12000.dat** | 0.052859068 | 0.119891405 | 0.118870974 | 0.118438005 | 0.060082912 | 0.094028473 |
| **A\_15000.dat** | 0.065862179 | 0.063770533 | 0.064422607 | 0.130502939 | 0.079763412 | 0.080864334 |
| **A\_18000.dat** | 0.075799227 | 0.116845846 | 0.099819899 | 0.119881868 | 0.119349480 | 0.106339264 |
| **A\_21000.dat** | 0.087769270 | 0.117325068 | 0.166559696 | 0.089327812 | 0.158743143 | 0.123944998 |
| **A\_24000.dat** | 0.103719473 | 0.167273521 | 0.195341587 | 0.110671997 | 0.132161617 | 0.141833639 |
| **A\_27000.dat** | 0.113697529 | 0.255452156 | 0.212285519 | 0.187909603 | 0.189791918 | 0.191827345 |
| **A\_30000.dat** | 0.139737368 | 0.159539461 | 0.160342693 | 0.192178726 | 0.219863415 | 0.174332333 |
| **B\_3000.dat** | 0.019949198 | 0.016012907 | 0.027614832 | 0.020704746 | 0.015328884 | 0.019922113 |
| **B\_6000.dat** | 0.051859379 | 0.033372641 | 0.033381462 | 0.032202959 | 0.039433479 | 0.038049984 |
| **B\_9000.dat** | 0.121675491 | 0.070996761 | 0.082254410 | 0.047949314 | 0.059723616 | 0.076519918 |
| **B\_12000.dat** | 0.095744848 | 0.084186316 | 0.081624985 | 0.078992128 | 0.078715801 | 0.083852816 |
| **B\_15000.dat** | 0.113223076 | 0.100223303 | 0.076701880 | 0.112113476 | 0.120609283 | 0.104574203 |
| **B\_18000.dat** | 0.106713295 | 0.077266455 | 0.124750614 | 0.139999390 | 0.139625311 | 0.117671013 |
| **B\_21000.dat** | 0.125485897 | 0.100792408 | 0.116410494 | 0.108750105 | 0.127837896 | 0.115855360 |
| **B\_24000.dat** | 0.110132694 | 0.099541664 | 0.149422169 | 0.142221928 | 0.111847639 | 0.122633219 |
| **B\_27000.dat** | 0.121098757 | 0.218641043 | 0.199965477 | 0.200011730 | 0.193991899 | 0.186741781 |
| **B\_30000.dat** | 0.140922785 | 0.182892799 | 0.234416246 | 0.133342266 | 0.176613808 | 0.173637581 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Monotone Chain** | | | | | | |
| **Name** | **Time, Run 1** | **Time, Run 2** | **Time, Run 3** | **Time, Run 4** | **Time, Run 5** | **Average** |
| **A\_3000.dat** | 0.016960144 | 0.017055273 | 0.040652752 | 0.027701616 | 0.029361486 | 0.026346254 |
| **A\_6000.dat** | 0.032950401 | 0.070644379 | 0.072205305 | 0.061381817 | 0.059865713 | 0.059409523 |
| **A\_9000.dat** | 0.051859856 | 0.090050697 | 0.088989496 | 0.061857462 | 0.088558912 | 0.076263285 |
| **A\_12000.dat** | 0.065827370 | 0.101438284 | 0.438152313 | 0.101473570 | 0.119972229 | 0.165372753 |
| **A\_15000.dat** | 0.088762283 | 0.098651886 | 0.233987093 | 0.148570538 | 0.158086061 | 0.145611572 |
| **A\_18000.dat** | 0.096701384 | 0.131184101 | 0.281697273 | 0.152089834 | 0.174361944 | 0.167206907 |
| **A\_21000.dat** | 0.114807367 | 0.175915241 | 0.226244688 | 0.207911730 | 0.150244474 | 0.175024700 |
| **A\_24000.dat** | 0.132982016 | 0.177069187 | 0.220929146 | 0.229929447 | 0.166301966 | 0.185442352 |
| **A\_27000.dat** | 0.175395727 | 0.311542511 | 0.270968437 | 0.210021496 | 0.193288565 | 0.232243347 |
| **A\_30000.dat** | 0.188453913 | 0.353291512 | 0.299080133 | 0.198092461 | 0.187674046 | 0.245318413 |
| **B\_3000.dat** | 0.015955210 | 0.030510187 | 0.039209604 | 0.022716045 | 0.019949675 | 0.025668144 |
| **B\_6000.dat** | 0.031918764 | 0.060817719 | 0.077346087 | 0.048414946 | 0.031647921 | 0.050029087 |
| **B\_9000.dat** | 0.046870947 | 0.066053629 | 0.113059521 | 0.117836952 | 0.054776669 | 0.079719543 |
| **B\_12000.dat** | 0.064828634 | 0.143696547 | 0.161778688 | 0.082161665 | 0.084068775 | 0.107306862 |
| **B\_15000.dat** | 0.084772825 | 0.137765646 | 0.175454378 | 0.089966536 | 0.147865772 | 0.127165031 |
| **B\_18000.dat** | 0.098736286 | 0.158543825 | 0.212340355 | 0.160107136 | 0.122148991 | 0.150375319 |
| **B\_21000.dat** | 0.132642746 | 0.117470264 | 0.250306845 | 0.220004797 | 0.198058605 | 0.183696651 |
| **B\_24000.dat** | 0.152631998 | 0.182972908 | 0.337704420 | 0.260069132 | 0.169011116 | 0.220477915 |
| **B\_27000.dat** | 0.173536301 | 0.184217930 | 0.300958395 | 0.266865015 | 0.260935068 | 0.237302542 |
| **B\_30000.dat** | 0.168508291 | 0.250455856 | 0.318091631 | 0.213325500 | 0.250105858 | 0.240097427 |

After running five tests on every algorithm I investigated the data I received. Starting with set-A I calculated these calculates and graphs:

Gift Wrapping, had an average run time of 0.0117 seconds overall.  
Graham Scan, had an average run time of 0.1110 seconds overall.  
Monotone Chain, had an average run time of 0.1478 seconds overall.

And from looking at our graph we can also see how these algorithms performed, and we can confirm that for set-A the best algorithm overall was the Graham Scan with an overall running time of 0.111 seconds, followed by Gift Wrapping at an overall running time of 0.117 seconds and finally Monotone Chain comes in last at 0.148 seconds. This is still kind of what I expected as the Gift Wrapping algorithm is not the most effective or efficient algorithm to run. Although the Monotone Chain was the least effective here which was to my surprise.

Now for the data of set-B:

Gift Wrapping, has an average run time of 9.9993 seconds overall.  
Graham Scan, had an average run time of 0.1736 seconds overall.  
Monotone Chain, had an average run time of 0.2401 seconds overall.

Looking at these graphs we can see that the Gift Wrapping algorithm is suffering compared to the time complexity of the other two. With an average of 10 seconds overall, we can say that Gift Wrapping is out of the question. Although when comparing the Graham Scan and Monotone Chain we can see that they both have quite similar run times with 0.17 seconds for the Graham Scan and 0.24 seconds for the Monotone Chain.

Altogether with both set-A and set-B the time complexity is as follows:

Gift Wrapping, has an average run time of 2.0614 seconds overall.  
Graham Scan, had an average run time of 0.1075 seconds overall.  
Monotone Chain, had an average run time of 0.1450 seconds overall.

From all this data we can determine the complexity of our algorithms.

The Gift Wrapping algorithm is by far the worst of the three algorithms with an exponential time complexity as we can see on the graph. The algorithm was quite efficient for problems smaller than (12000, 4) \*(Vertices in graph, Vertices in hull) in size (in accordance to our results). When the number of convex hull vertices overtook this value, we found a large degradation on the time complexity of the Gift Wrapping algorithm. On much larger problems, like the set-B files we found that this algorithm suffered with a near exponential time complexity. This matches up with the algorithms actual time complexity which is O(nm) where n is the number of graph vertices and m is the number of convex hull vertices.

The Graham Scan was by far one of the fastest algorithms that we tested coming in at a logarithmic time. The algorithm had trouble getting an efficient time for the small problems but quickly made its mark on the quicker problems with a running time that never exceeded 0.25 seconds in our tests. The actual time complexity of this algorithm is O(n log n) where n in the number of vertices in the graph.

The Monotone Chain was far faster than Gift Wrapping but didn’t quite catch up with the Graham Scan in its time complexity. Monotone Chain ran in a logarithmic time much like the Graham Scan did where it struggled at first but with larger problems became more efficient, the running time never exceeded 0.32 seconds within our tests. The actual time complexity for this algorithm is O(n log n) where n is the number of vertices in the graph.

1. **References:**

These are the following sites that I used to help implement and understand convex hulls. I created my programs mostly from pseudo code from Wikipedia and found informative PDF Lectures from other universities to figure out how my algorithms worked. The following is the Wikipedia pages I used to implement y code:

<https://en.wikipedia.org/wiki/Gift_wrapping_algorithm>

<https://en.wikipedia.org/wiki/Graham_scan>

<https://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monotone_chain>

<https://en.wikipedia.org/wiki/Chan%27s_algorithm>

The following pages I used for a more informative explanation around how the algorithms functioned:

<https://startupnextdoor.com/computing-convex-hull-in-python/>

<https://www.geeksforgeeks.org/convex-hull-set-1-jarviss-algorithm-or-wrapping/>

<https://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/>

<http://www.algorithmist.com/index.php/Monotone_Chain_Convex_Hull>

<https://gist.github.com/tixxit/252229>

**Created by,**

C:\Users\Zachary\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Sketch - Copy.png

Student ID: 58520526