Software Assignment 1 - COL 215

Report by:

Aadi Govil - 2023CS10490

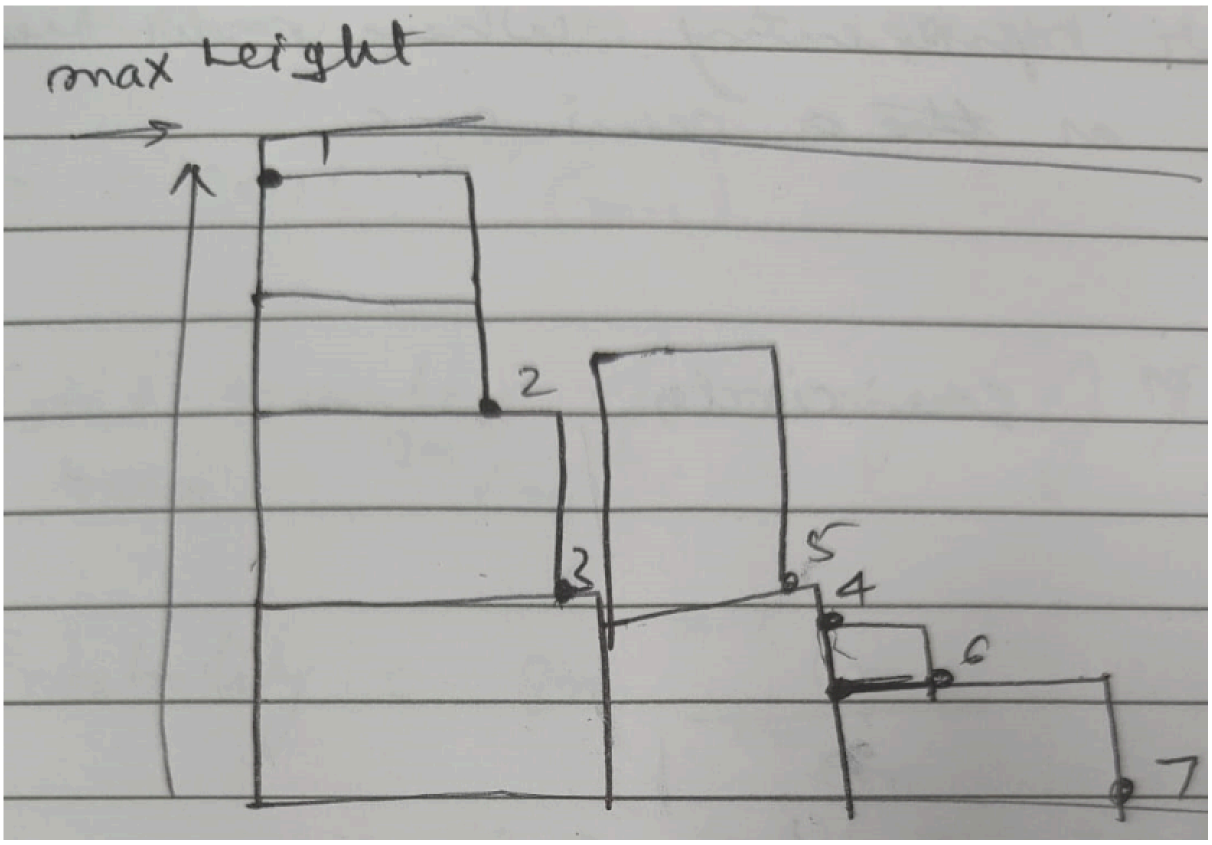
Harshit Kansal - 2023CS10498

Design Decisions:

Algorithm 1: Optimal packing

Explanation of algorithm:

For each iteration of the code, we define a maximum height. We first sort all of the rectangles by their widths. Then, we start placing the rectangles from the left corner. The algorithm for placing:



We store coordinates and max height and width allowed at the marked points into a priority queue-type data structure, which is sorted based on their X-axis and Y-axis coordinates. We start iterating over the rectangles and try placing rectangles at the leftmost and highest of the points available first to improve packing efficiency. If we cannot put it we move on to the next point and so on, till we can all the rectangles.

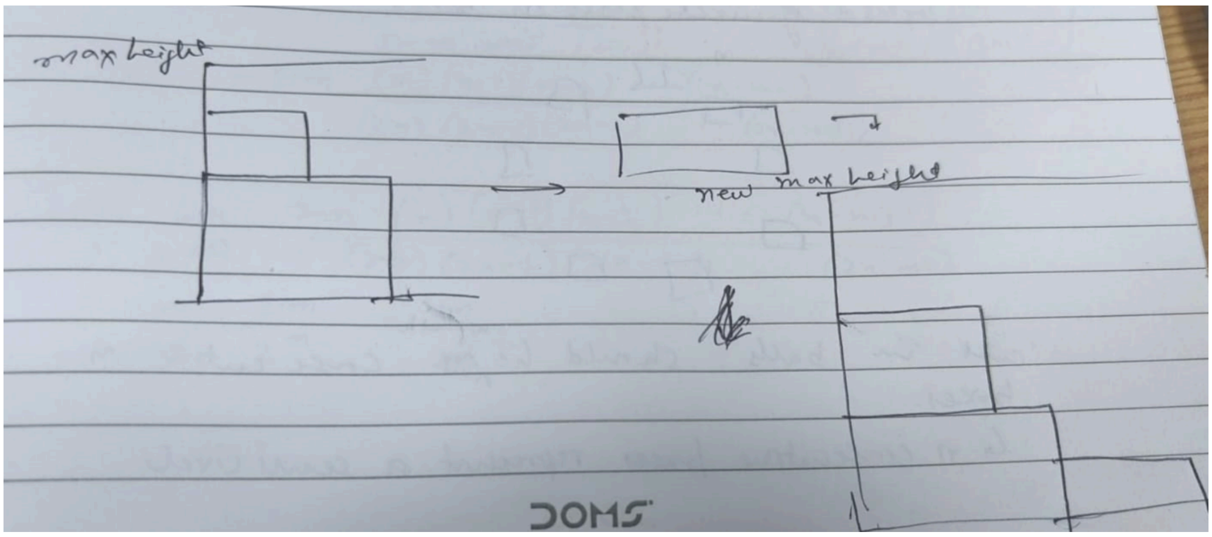
We then change the max height allowed and iterate the same algorithm overall allowed heights, which is the maximum height of the set, to the sum of all heights, and output the best answer we get.

Design Decision:

To save time, we made the step value, which increases height iteratively to 10 when the number of gates is high, as we found that does not lead to a lot of drop-in packing efficiency according to initial testing.

Variation 1 of the algorithm:
We introduced a bound of 500 to the maximum height to increase the speed of operation, but this worked out in our favor in some circumstances and not so much in others in terms of packing efficiency, which will be explored in the testing section.

Algorithm 2: optimal_packing_square
In an attempt to reduce the time taken while keeping up the efficiency of our algorithm, we decided to make an algorithm such that whenever the width of the bounding box increased, the height of the bounding box would equal the new width.



This however, as we see in testing, significantly decreased the packing efficiency but greatly sped up the time.

During Testing:
We decided to use 3 different varying factors:

- No of gates
- Spread of the width and height values
- Mean of the width and height value

So, for testing each, we kept the other 2 to be the same:

- While varying the number of gates, we kept ranges of width and height from 1 to 100 for both
- While varying the mean, we kept the number of gates to be 1000, while we kept the range to be 10 for both width and height
- While varying range, we kept a mean of 50-51, and number of gates to be 1000

And we plotted 2 different factors with it:

- Time taken to run
- Packing efficiency

We performed this for both the algorithms, taking averages over various randomly generated test cases to get an approximate idea of what is ensuing.

We also performed some other standard tests on our algorithms, the results of which will be seen in the testing.

Testing process and inferences

In The testing phase, firstly, we tried running our algorithms on the given sample test cases. After this, we tried to run our algorithms on a few standard benchmarks we found in research papers like the test case involving squares of side lengths varying from 1 to 100, and found the following results.
Algorithm 1: percentage white space - 3.77 percent, time taken - 0.22 seconds
Variation of algorithm 1: percentage white space- 3.77 percent, time taken - 3 seconds
Algorithm 2 : percentage white space - 23.7 percent, time taken - 0.001 seconds

From these results we can see that algorithm 1 is performing fairly well with only 3.77 percent white space and running in 0.22 seconds, whereas algorithm 2 runs quite faster but the percentage white space is higher

After this, we tested our code with another standard benchmark which is a test case width rectangles of sizes 1X100 , 2X99, …. 100X1. This gives us a good idea of how the code functions on rectangles with varying widths and sizes.
Algorithm 1: percentage white space - 15.84 percent, time taken - 0.44 seconds
Variation of algorithm 1: percentage white space- 3.77 percent, time taken -2.87 seconds
Algorithm 2 : percentage white space - 30.3 percent, time taken - 0.001 seconds

Here we observe that the algorithm work significantly worse in comparison to the above test with 15.84 percent white space and runs in 0.44 seconds, whereas the second algorithm takes very less time to run but gives around 30 percent error

We also tested the code with a slight variation of this test case to see if there was much difference, namely, modifying it so that the rectangle widths and heights change by two instead of one to see the effects.
Algorithm 1: percentage white space - 20.70 percent, time taken - 0.094 seconds
Variation of algorithm 1: percentage white space- 20.70 percent, time taken -0.35 seconds
Algorithm 2 : percentage white space - 35.3 percent, time taken - 0.001 seconds
The algorithms perform a little worse as the n value is lesser and it doesn't work well on small number of rectangle with small changes in widths

We also tested a practical case in which the gates with larger dimensions are lesser in number in comparison to the smaller number of gates

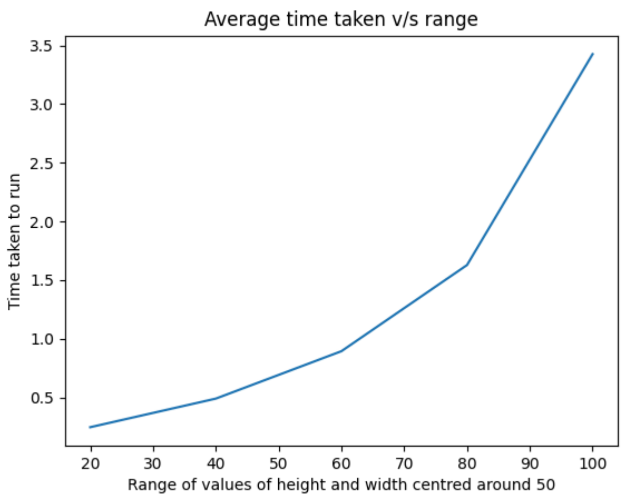
Algorithm 1: percentage white space - 0.95 percent, time taken - 0.82 seconds
Variation of algorithm 1: percentage white space- 0.95 percent, time taken -20.49 seconds
Algorithm 2: percentage white space - 29.6 percent, time taken - 0.05 seconds
Here we observe that algorithm 1 performs really well, having percentage white space less than 1 percent and running within 1 second, algorithm 2 doesn't do too well with 30 percent white space.

After this, we followed three types of testing, graphed their results, and made some interesting observations:

1) During this type of testing we kept the mean of the value of width and height around 50, but we varied the ranges of the values from 40-60, 30-70, 20-80, 10-90, 1-100 and observed the results. We wanted to see the change in error and mean with varying ranges of values. To ensure proper results we did 50 iterations with randomly generated test cases for respective ranges and then took average of all the errors and means for all the algorithms.

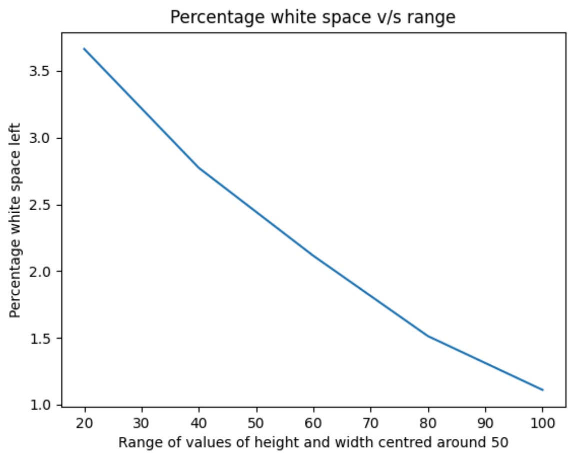
   Algorithm 1:

   The graph obtained for the time taken v/s range is shown below



   We observe from the graph that as the range of values increases from 20 to 100, the average time taken follows a polynomial curve, possibly proportional to the square of range, for small range of values the time is quite less- close to 0.1 seconds but increases to 3.5 seconds for varying the range to about 100. This is useful because the gate's widths and heights are closer in practical applications, and the range of values is less.

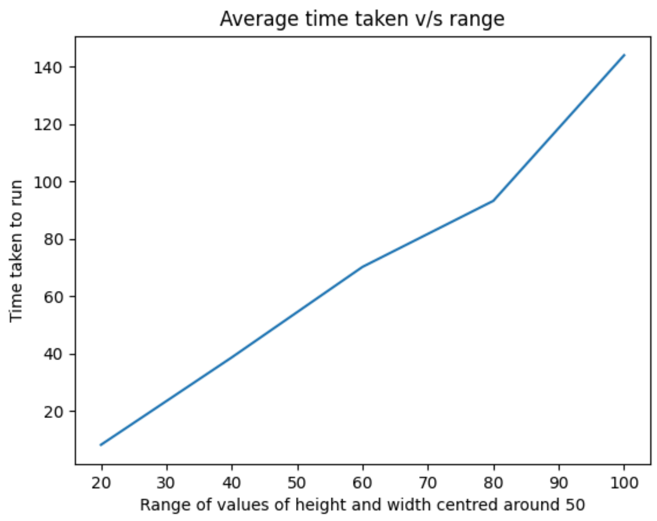
   The graph obtained for the percentage white space with increasing range is given below



We can see that the white space decreases with the increasing range in what seems like a close to linear fashion, however for the small values of range also, the white space is not that high, only about 3.5 percent, with high values of the range of data it drastically reduces to a minimal 1 percent.

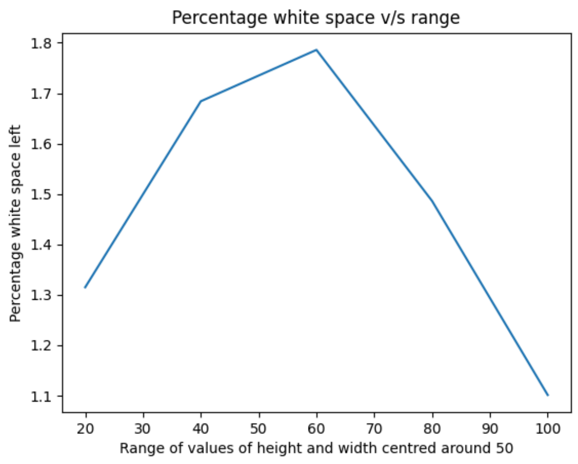
Variation of algorithm 1(algorithm 3):

The graph for time v/s range is given below



We can see from the graph that the time taken is increasing in a similar fashion to the normal algorithm with a limit of 500 with an increasing range of values. However, the graph is not too smooth because we could only average this over 5 iterations instead of 50, as this algorithm takes a little longer to run. For close values, it runs in less than 20 seconds, which is great, and goes to about 140 seconds for a range of values of 100.

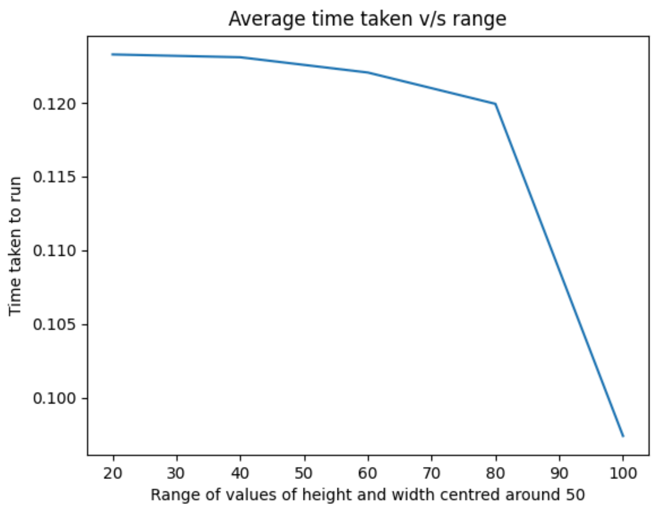
The graph for the percentage whitespace v/s range is shown below



We can see that this graph is quite different from the graph of this algorithm with the bound. The algorithm performs much better on gates with close widths and heights, the ranges of values being around 20, giving white space only about 1.3 percent. On increasing range, it goes to a maximum error of about 1.8 percent near 60 and then goes down again to about 1 percent for varying widths and heights. One notable thing is that it improves accuracy from the normal algorithm from 3.5 to 1.3 percent.

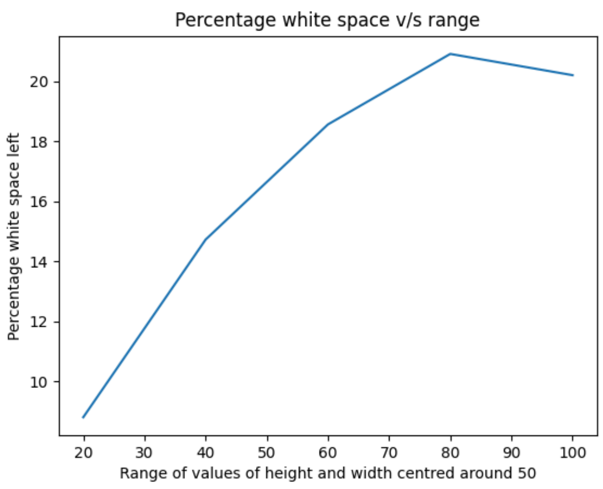
Algorithm 2: Optimal packing in square

The graph for the time v/s range of values is shown below



We can see that this algorithm takes very little time for all varying ranges of values, around 0.1 seconds. It stays constant around 0.12 seconds for most ranges and decreases to less than 0.1 seconds for the range of values closing to 100.

The graph for the percentage whitespace v/s range of values is shown below

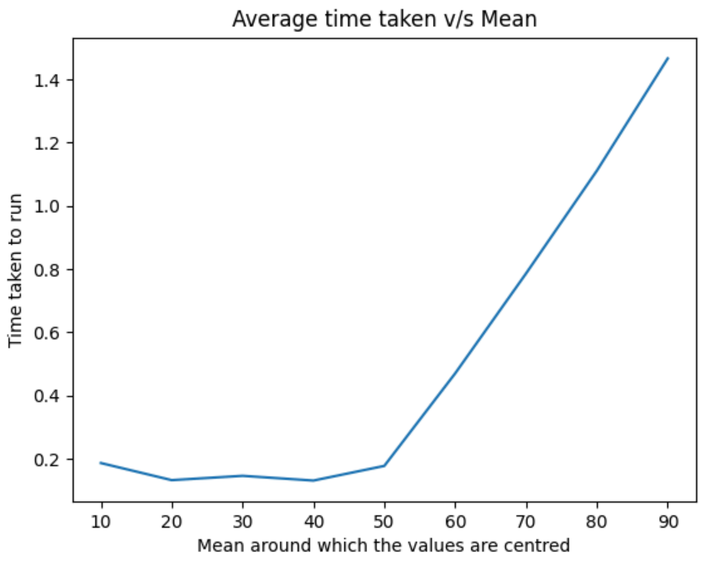


We can see that this is quite different from the other algorithms as the percentage of white space increases with an increasing range of values. It is less than ten percent for close widths and heights and increases to about 20 percent with higher differences in heights and widths of gates. This is quite beneficial as for the practical case of gates being close it gives white space to be less than 10 percent and runs quite fast.

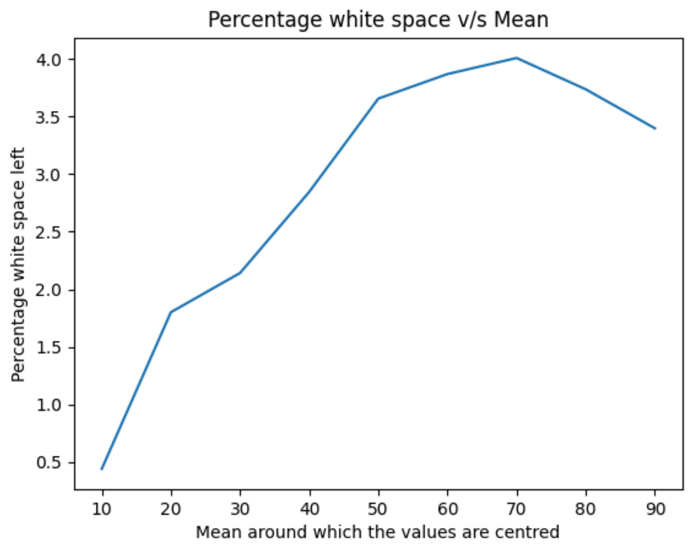
2) During this testing, we kept the range of values constant at 20 and tried to change the mean around which the values are centered from 10 to 90 and observed the results. Here also, we averaged the data over 50 iterations to get better results while only doing it 5 times for the variation in algorithm 1 as it takes longer to run.

Algorithm 1: Optimal packing

The graph for average time takes v/s mean is shown below



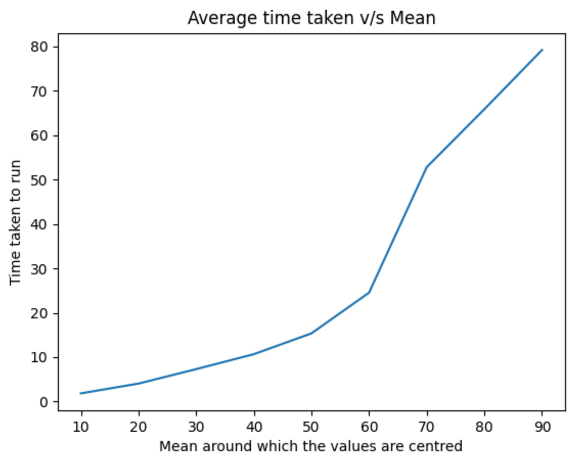
From this graph, we see that the time taken remains less(about 0.2 seconds) for mean values till 50, and after that, it increases in a linear fashion till it reaches a mean of 90, taking about 1.4 seconds there. So, as the size of the gates increases, the time taken also increases.
The graph for the percentage white space v/s mean is shown below



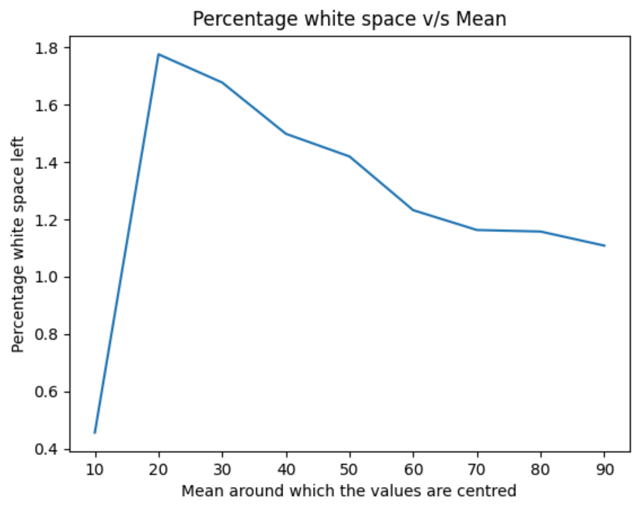
We observe that the percentage follows an increasing trend, reaching a peak of 4 percent at about a mean of 70 and then shows a little decrease till it reaches 90. We also see that for smaller gates, the white space is very less(close to 0.5 percent), which is a good thing, and it reaches a maximum of about 4 percent for larger gates.

Variation of algorithm 1:

The graph for the time taken v/s mean is shown below



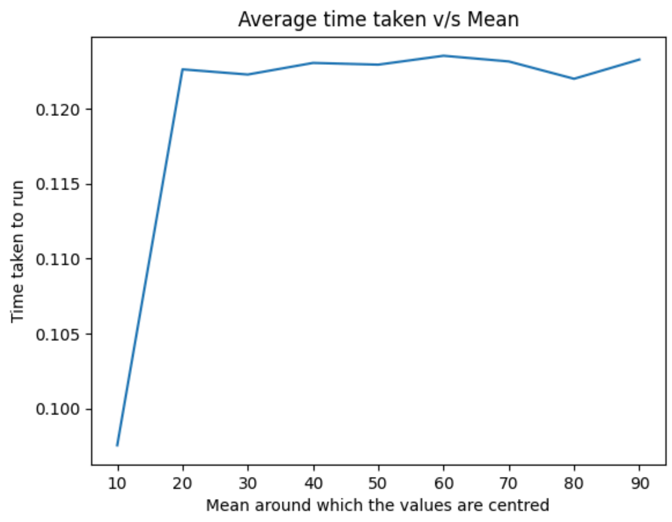
We see that this graph is quite different from the above one because it depends on the heights of gates, and there is no limit of 500, so the time increases as the mean increases and gates become larger. The graph is not so uniform as we only took an average of over 5 iterations as it takes a long time to run. The graph ranges from quite less time for smaller gates and increases to about 80 seconds for larger gates.
The graph for the percentage white space v/s mean is shown below



We observe that the percentage white space is very less for smaller gates (with a mean around 10), after which the percentage white space decreases at a small pace as the mean increases, starting at 1.8 and going down to 1.2

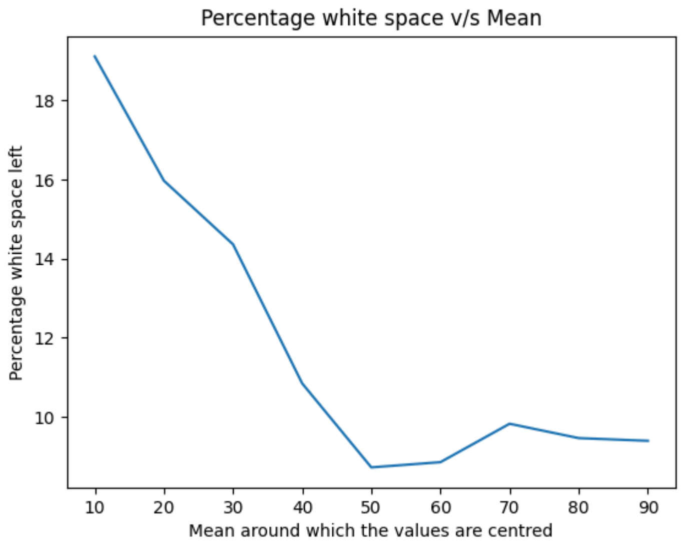
Algorithm 2:Optimal packing square

The graph for the time taken v/s mean is shown below



We observe that the time taken is relatively low(around 0.12 seconds) and remains almost constant for all values of mean but is lower than the rest at about 0.1 percent for gates, which are very small in size with a mean of around 10.

The graph for the percentage white space v/s mean is shown below

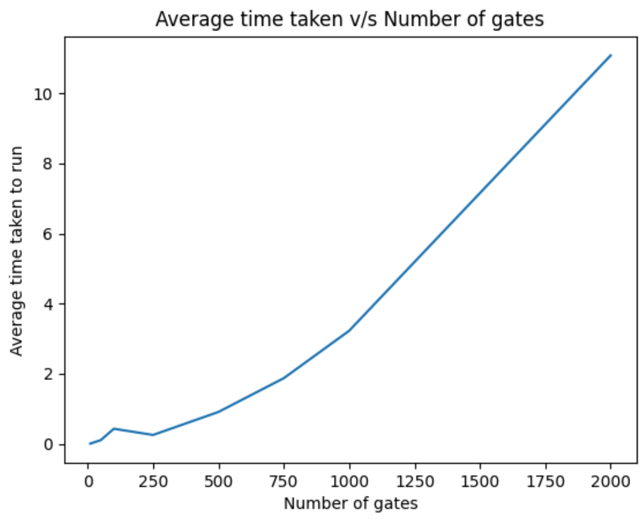


We see that the percentage of white space decreases from 20 to 10 as the mean increases from 10 to 50, after which the percentage of white space stays close to 10 percent.

3) During this type of testing, we randomly varied the heights and widths between 1 and 100, but the value of the number of gates changed from 10 to 2000, giving us an idea of how the algorithms perform when the number of gates is changed.

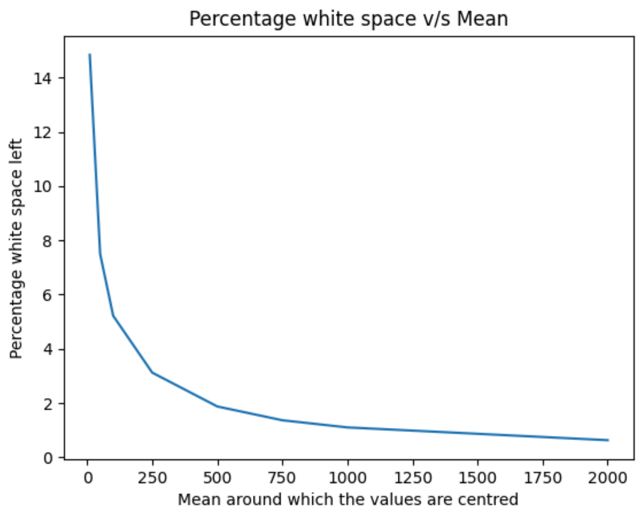
Algorithm 1: Optimal packing

The graph for the time taken v/s number of gates is given below



We see that the time increases as the number of gates increases, and it seems like a graph that is polynomial in terms of the number of gates. However, there is a spike near n=100. This is because in our algorithm, when the number of gates exceeds 100, we don't check all heights but only in intervals of 10 to reduce time which reduces the time suddenly by a factor of 10, and hence the spike near n = 10. For the given constraints of n=1000, the algorithm takes around 2 seconds to run.

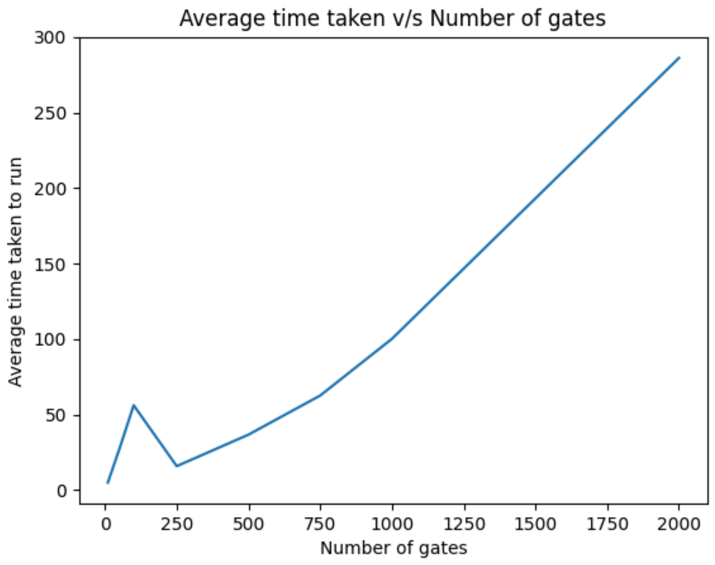
The graph for the percentage white space v/s number of gates is shown below



We observe that as the value of the number of gates increases, the percentage white space decreases drastically for smaller values of number of gates(less than 250), after which the percentage white space decreases slightly and reaches almost constant(around 1 percent) for larger values of the number of gates.

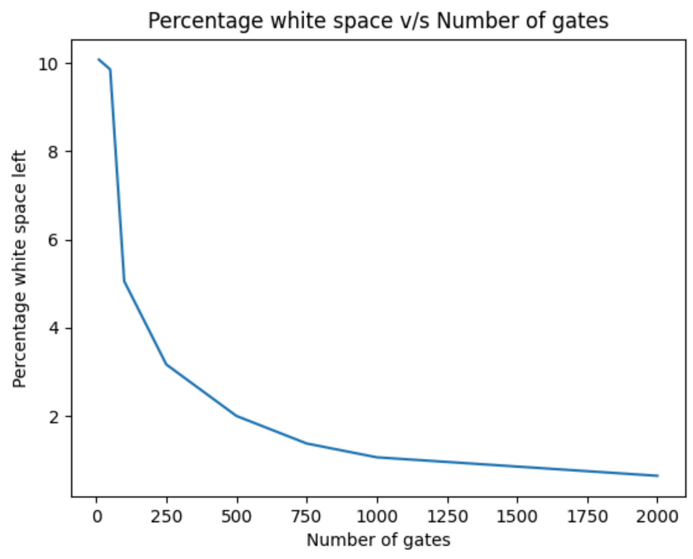
Variation of Algorithm 1

The graph of time taken v/s number of gates is given below



This graph is quite similar to the algorithm without the variation, with the only notable change being a more considerable spike near n=100.

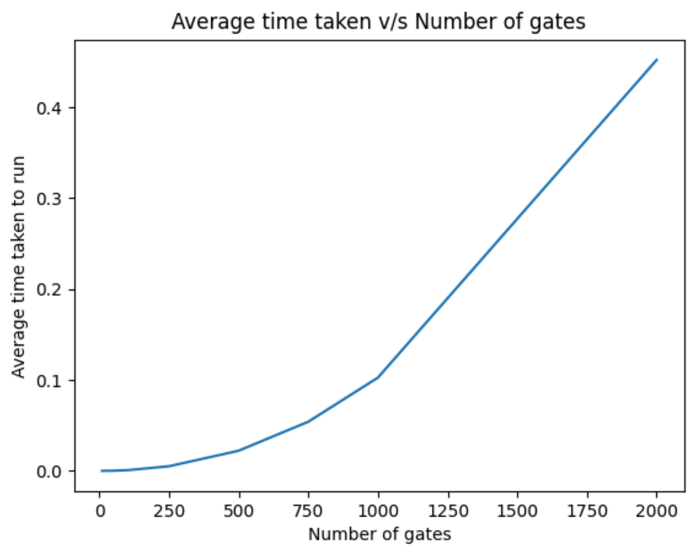
The graph of percentage white space v/s Number of gates is shown below



This graph is very similar to the algorithm without variation and with no noticeable difference.

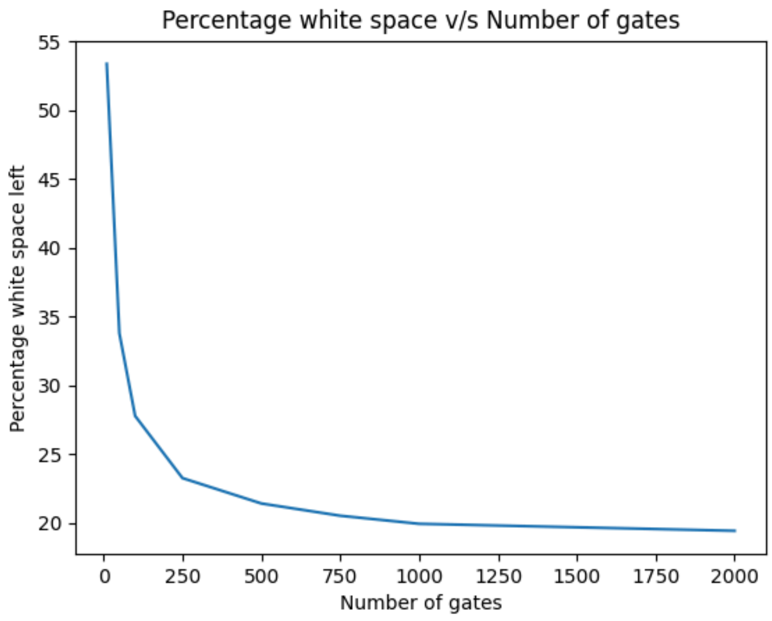
Algorithm 2: Optimal packing square

The graph for the time taken v/s number of gates is shown below



The graph follows a similar trend to algorithm 1, with it being proportional to the square of the number of gates, with the only difference being that there is no spike near n=100 as we are not iterating over heights, and the time is relatively less in comparison to the above algorithm

The graph for percentage white space v/s number of gates is shown below



This graph is also the same in shape in comparison to algorithm 1, with the only difference being that the percentage white space values start above 50 percent and go down to 20 percent, which is much more in comparison to the above algorithm.

Finally, if there is no problem with the algorithm running for a longer time, we should use variation of algorithm 1 with no bound, in the cases of low variance, algorithm 1 especially performs better than the normal algorithm 1. In other cases, algorithm 1 mostly works with great accuracy and time in order of seconds. If there is a need for a way faster algorithm, then we need to use algorithm 2.

Time Complexity Analysis

Algorithm 1: O(50*N^2LogN for N>100, 500* N^2logN, for N<=100)
in this algorithm, we are iterating over heights in steps of 10 to a maximum of 500 which multiplies the overall complexity by around 50. Then, in each iteration, after fixing the height, we sort the rectangles in decreasing order of width, which takes around NlogN time then we iterate over all the rectangles, which contributes a factor of n, and in each iteration which checks for points in the priority queue starting from the leftmost and topmost point, in the worst case we have to check points which are in the order of n and each time we are accessing a point it takes LogN time since it is a priority queue, so overall complexity comes to be about N^2logN for a fixed height.
Variation of algorithm 1: O((sum of heights)(N^2LogN))
The only difference from algorithm 1 is that we are iterating over all heights from max height to the sum of heights, which increases the time complexity by around 20 times in the worst-case

Algorithm 2: O(N^2LogN)
The algorithm is quite similar to algorithm 1, but it does not involve iterating over heights. Instead, we do the same process in N^2LogN of adding rectangles and checking points, but whenever the width is greater than the height, we make the height equal to the width eliminating the need to fix a height and iterate over the heights and greatly improving time complexity