Software Assignment – 1 Report

Nischay Patil, Parth Agrawal

24 August 2024

1 Introduction

In this assignment we were supposed to automatically generate a compact physical layout of a gate-level circuit. All the gates are rectangular and the final layout is also rectangular. In a compact layout, an attempt to minimise the blank space is made.

2 Design decisions

First approach:

Sort the rectangles in decreasing order with respect to width and then height. Set the width of the bounding box as the maximum width. Now keep adding rectangles in order (notice they are sorted in decreasing order of width). Only other thing we have to check is the heights so that they are not overlapped. But this approach was not effective because of the following problems:

- Bounding box's height was rapidly increasing as we were adding rectangles layer by layer, due to which much of the space was getting wasted, resulting in very low efficiency

If the width of all the rectangles is same, this logic would give most optimal layout for the gates/rectangles

Second Approach: (Implemented in submitted files)

We used the implementation of binary tree as our second approach. We took help from a online source for this algorithm:

https://codeincomplete.com/articles/bin-packing/

In this approach, we sort the rectangles according to a number of ways for e.g. Height, width, area in reverse order. Then we place the first rectangle of the list and add the next rectangle to the right of it. After that there will be a rectangle defined in which there will be some whitespace remaining if the second rectangle is not of the same width/height. Then we divide the whitespace into two regions – one on top and one on right and store it as children of a node as we do in a binary tree. If the next rectangle can be placed in one of the nodes, then we place it there, otherwise we grow the size of the bounding box either on the right or on top depending upon the dimensions of the rectangle we are adding. If we are able to place the rectangle in one of the nodes, then we mark the node as used and divide the whitespace region left after consuming the node into right and left. When this is implemented recursively, we form our desired bonding box.

Complexity for this approach: O(n*log(n))

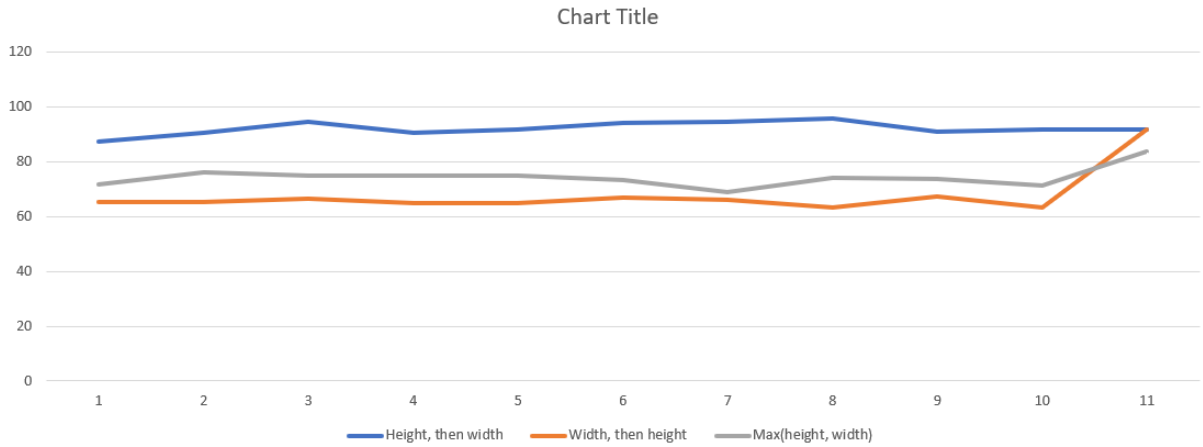
Reason: Because we are using the implementation of a binary tree in which we are searching space to fill a rectangle which we have divided into two ways- top and right

Obervations:

We observed that by sorting the rectangles in different ways, we are getting different efficiency of packing for various test cases.

Implementing this approach, we observed difference in efficiencies for different type of sortings – height then width, width then height, max(height,width). We took the data of efficiencies v/s no. of gates

used and plotted a graph for it. It looks like this:



Here, except the 11th point, all are for n=1000 gates and test cases were generated by importing random in our python file. 11th test case is for n = 35. So, we observed that efficiency is generally high if we are sorting height then width for higher values of n. And for low values of n, it can't be said.

Also, we observed that if the variance was high, efficiency was low in this approach. We checked that if the variance of height is in the order of 1e5, then efficiency was around 90 percent and if the variance was in the order of 1e4, then the efficiency was above 95%

random testcase of n = 1000, 1 to 100 random	
4783139.347	92.8764
4529748.105	89.6799
5059223.879	92.7699
random testcase of n=1000, 1 to 20 random	
8271.581701	97.7563
8062.020849	96.5007
8995.235251	95.8436

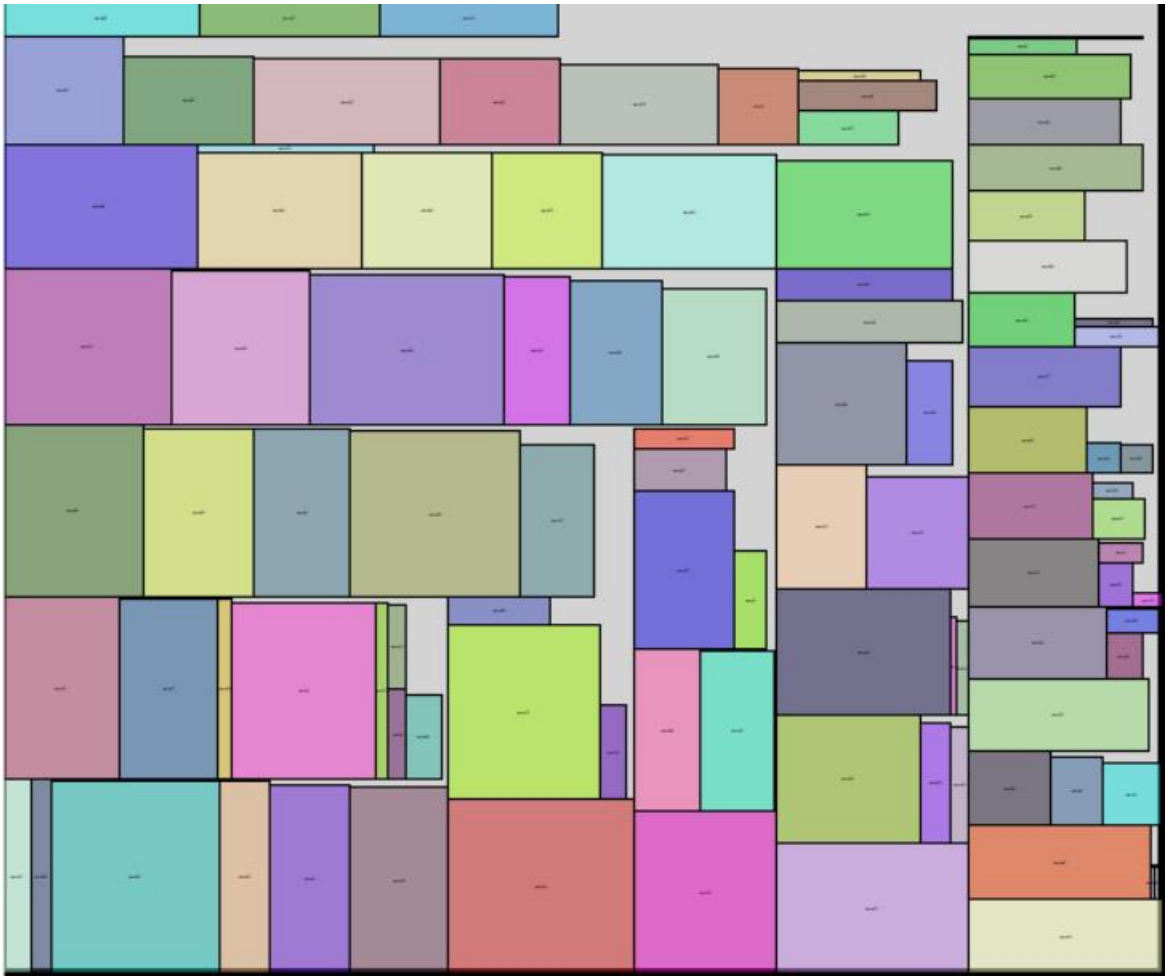
This table was variance v/s efficiency.

Advantages and Disadvantages:

This approach is not efficient enough because we are checking in this algorithm that if there is space on right, we place it in the right and

not see that it can be placed on the top or not. But this approach is the best if we want to optimise time in our code because its complexity is very low. Also, if the variance in height/width is low, then this approach gives high efficiency. This approach is the algo2.py file in the submission and is executed in main.py.

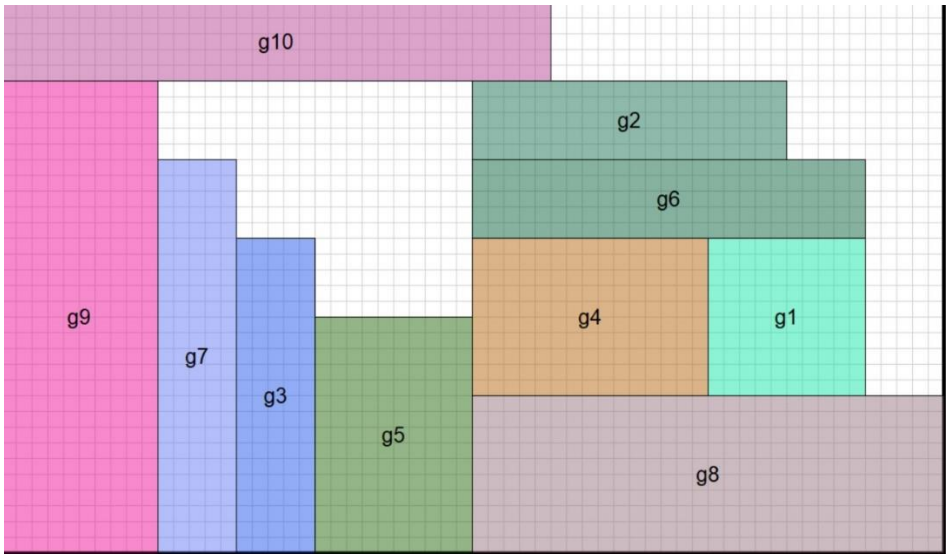
Visualised packing example:



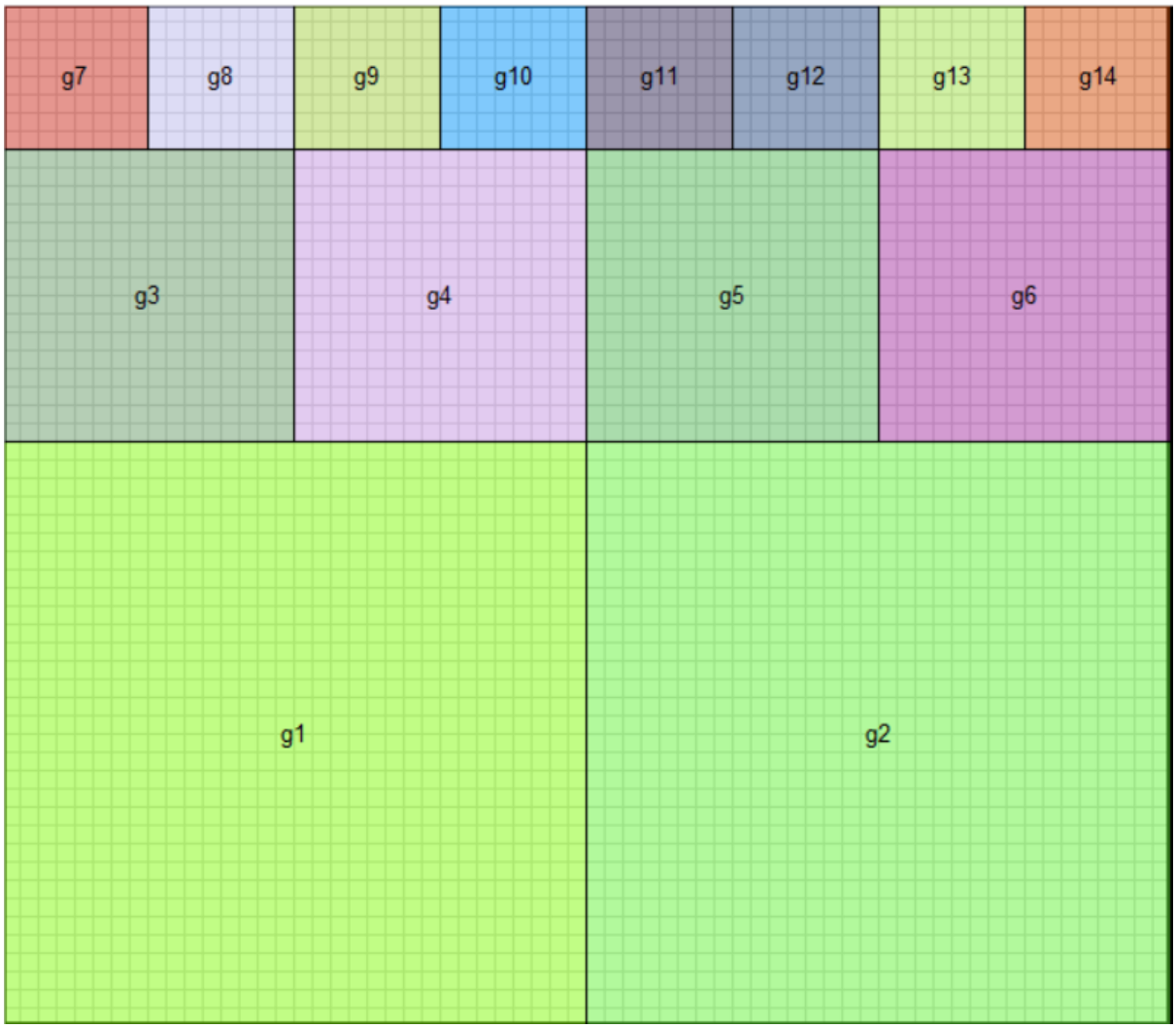
As we can see in the image, rectangles in the bottom are sorted according to height and are filling if there is space on the right or on

top. This is for n=100 and efficiency for this was 88%. (OWNTEST1)

This is for n=700, in which length and width are only from 1-20. Efficiency for this was 92.4% as the variance is low. (OWNTEST2)



And this was for n=10 (sample test case 3), for this efficiency was quite less. In the third approach, this is arranged in much compact way with a high efficiency.



We got 100% efficiency for this test case(OWNTEST 4), we can see that for this test case, binary approach is the most optimised approach.

Third Approach: (Implemented in submitted files)

In this approach, we are using a 2D grid to fill positions occupied by the rectangle. We took help from an online source for this algorithm:

https://www.codeproject.com/Articles/210979/Fast-optimizing-rectangle-packing-algorithm-for-bu

We take the max width among all the rectangles and max height among all the rectangles and set the initial size of the bounding box to be max width and max height. And initialise a 2D matrix of size max width*max height as false. Then we iterate in the list of

rectangles which is sorted according to width/height in reverse order. At each placing, we change the value of each cell of size 1*1 as true/visited. To place the rectangle, we check that if a size of the required width*required height rectangle is completely false or not. If yes, we place the rectangle there and append the coordinates in our output file. Otherwise, we increase the lengths of the bounding box by minimum height among the rectangles/ minimum width among the rectangles. And then again repeat the process to append the rectangle. Only increasing the lengths by 1 will result in fruitless results most of the time. After placing all the rectangles, we calculate the bounding box width and length iterating through the list of tuples of x,y coordinates. Overall, it is a time-consuming algorithm but gives highly efficient packing of the rectangles.
We observed that for around no. of gates = 500 and choosing the length and width among 1-20, it takes around 5 min for this algorithm to execute and that returns 99% efficiency.
Complexity of this code is very high
Time Complexity: O(no. of gates* max height * max width * average height of rectangle* average width of rectangle):

Reason: Firstly, we are creating a 2-D grid, so max height*max width and then we are checking for each gate so *no. of gates, also we are checking each cell, so it will be multiplied with the average width of the rectangles and average height of rectangles resulting in such complexity. This approach is not efficient for large inputs and takes a lot of time to execute.

Space Complexity would be O(max height * max width)

Observations:

This approach is very efficient packing wise but not time wise. For higher values of n, it does return high efficiency but takes >5-10min to run. We compared the results if we are implementing using this approach and using binary tree.

variance	efficiency	efficiency of binary
4486929	95.18	88.68370743
5115573	93.36	81.51662565
3837544	94.01	
8557.93	96.15	91.89609964
9488.61	95.71	80.1374872
10253.5	96.94	88.01963263

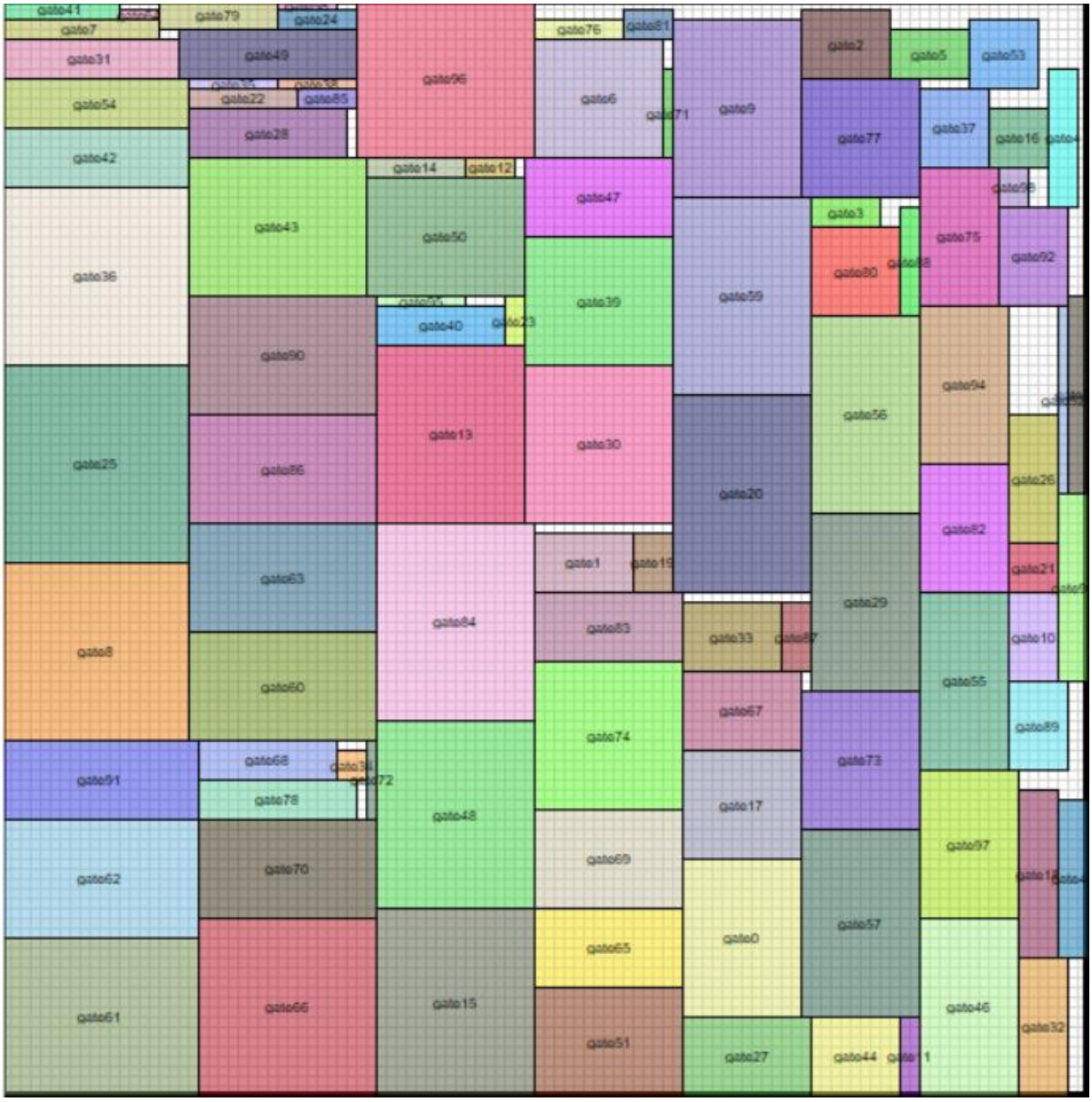
As we can see, efficiency for this approach is higher in comparison to the second approach.

We also observed that for low variation, n=1000 and if the height and width are only in range 1-20, out efficiency also reached 99.84.

Advantages and Disadvantages:

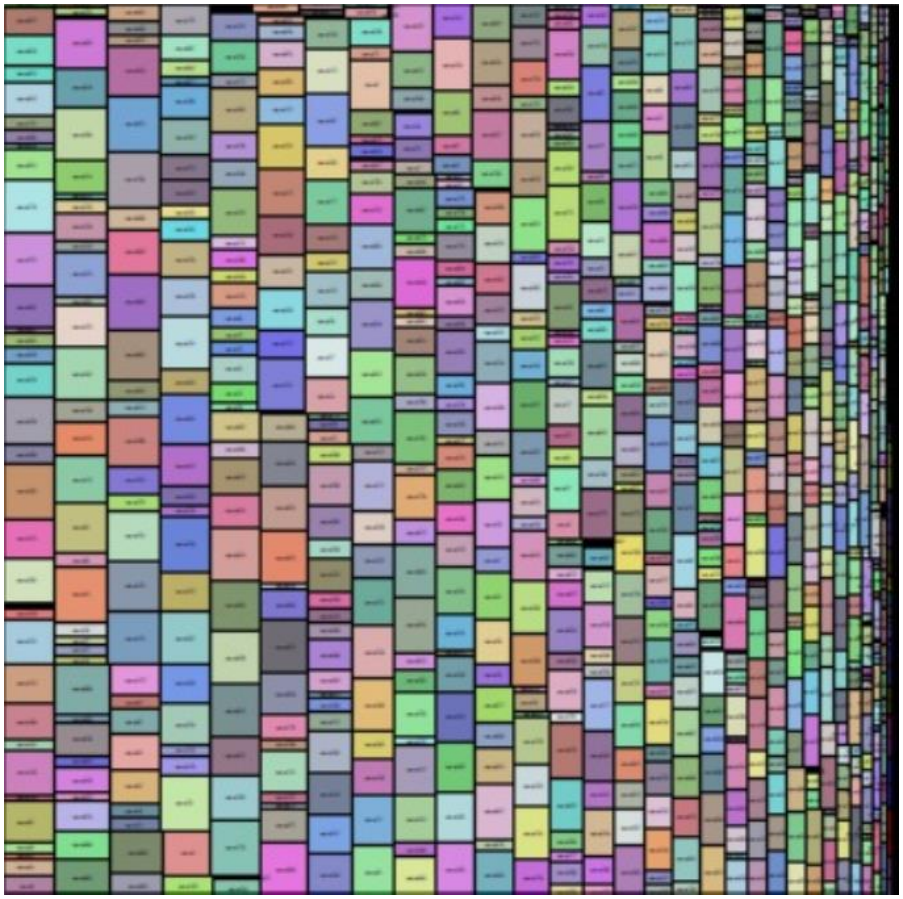
Considering the constraints, this code is not favourable if we want to optimise the time for this code as for higher values of n, it will execute in >10min but if our priority is optimising the packing efficiency, this code is more favourable as it checks every space possible.

Visualised Packing Example:

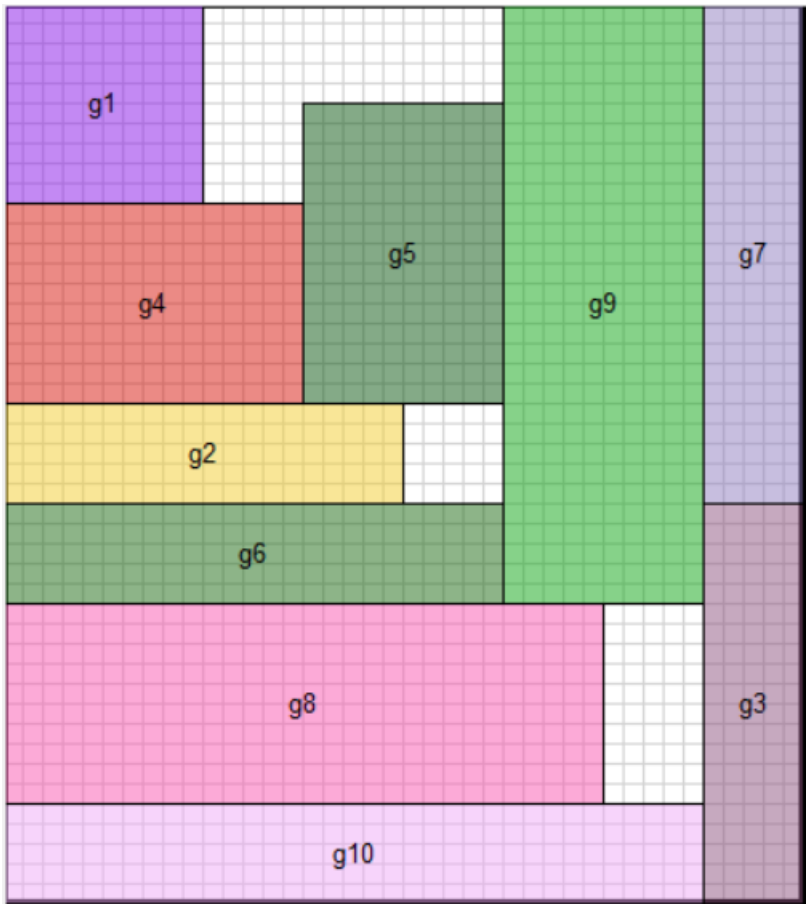


This was for n=100, we got 97% efficiency in this and as we can see, it is very compact. (OWNTEST 3)

This is for the OWNTEST 1 we tried on both the approaches. In this approach, we got 95% efficiency which is way better than the efficiency of the second approach. It took 5 min for this code to run.



We got this for n=1000 and we kept the length and height in range of 1-20. Efficiency for this packing was 99.8% which was the highest we got. But this test case took 20-25 min to run.



This was for the sample test case 3. In which we got 1800 as the bounding box area and is way more efficient than the second approach in which we got 2100.

We got this for OWNTEST 5 in which there were 1000 gates of size 1*1. For this, 97.66% efficiency came.

Conclusion:

Our second approach is more efficient in time and third approach is more efficient in packing. According to the need, we can use one of the two approaches in real-life scenario as well.

We have implemented both the algos in the files submitted and it gives the output according to the second approach for n>100 and for n<=100, it gives output according to the best efficiency among both the approaches. Also, we have attached some more test cases in the zip file which we tried along with the test cases for which visualized image is given above.

Remarks:

This was a great assignment and we learnt a lot about gate-packing and different type of algorithms which are used for this problem. The time spent debugging our code, figuring out the algorithm was very beneficial to us. I hope we get similar problems in the future too.