COL215 Software Assignment 1: Gate Packing

Vanshika (2023CS10746)
Laksh Goel (2023CS10848)

August 25, 2024

1 Task

Task was to generate a compact physical layout of gate-level Circuit. Gates were assumed to be rectangular and cannot be re-oriented. In the compact layout, blank space is minimized.

2 Design Decisions

We begin by implementing basic algorithms. Our first implementation was to

1. Start placing blocks, starting from maximum width, one above the other.
2. Finding a block of remaining width using binary search and placing it in the gaps formed .
3. Shortcoming was finding a block of appropriate size may go on endlessly , and then height(of filled area in the gap formed) would increase significantly.
4. Then we fixed this height to the maximum height of gates , but this did not turn out to be a good approximation for a large number of gates.
5. But this does not give good enough results. As we can't keep track of all the blank spaces.

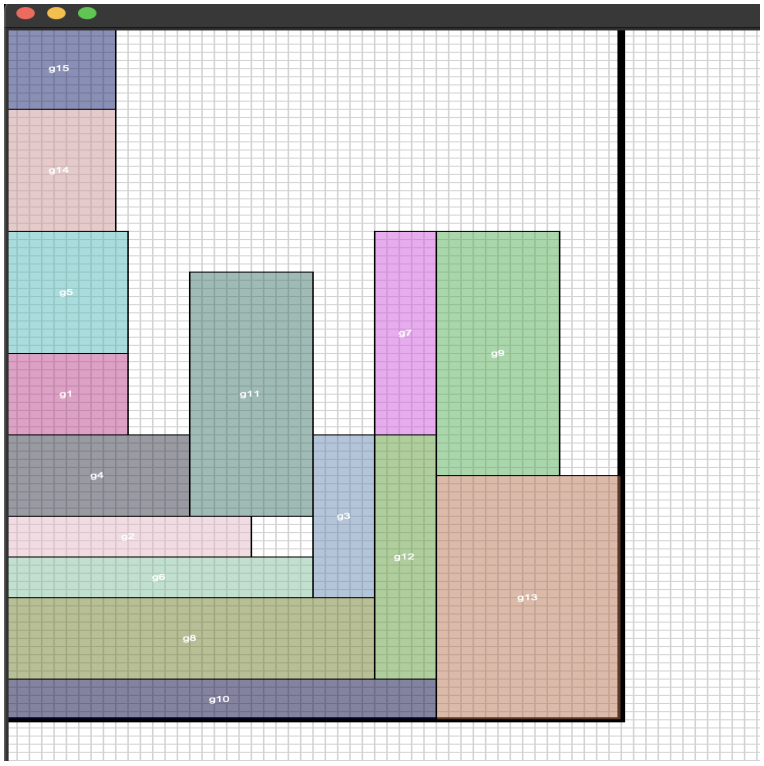

Figure 1: The very first naive approach result

So the first Question which came in our mind was: How to keep a track of blank spaces between blocks? And how to find an appropriate blank space to place the block?

2.1 Implementation-1

So we started with a list which could store the co-ordinates of the blank space, along with it's height and width. Whenever we placed a block, we removed that blank space from the list, added two new blank spaces, and sorted the list area-wise so that we could keep track of the smallest blank spaces available to us.

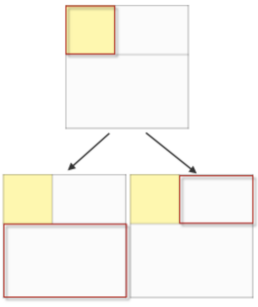

Figure 2: Adding empty spaces into list

Why to divide a blank space in such a way, that the smallest possible rectangle is made? We need to try to fit the gates in smallest possible area first. Hence we make

empty cells/spaces in such a way that one of the empty spaces made is as small as possible.

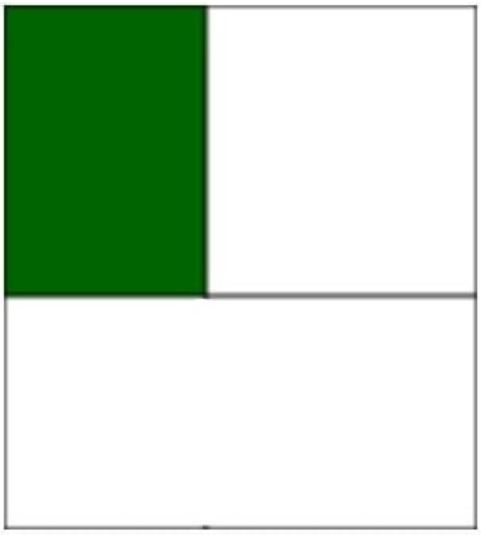

Figure 3: Horizontal cut larger sized empty spaces X

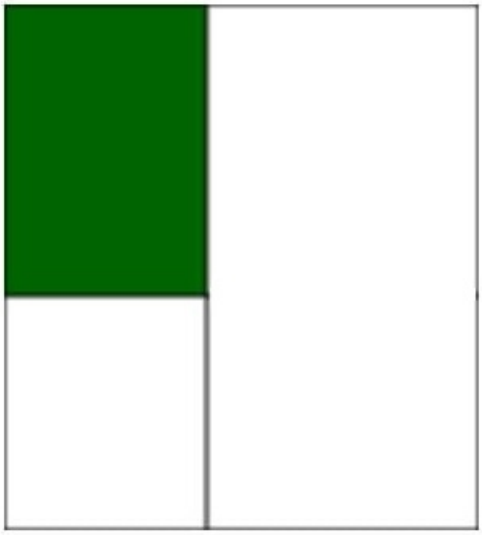

Figure 4: Vertical cut gives a smaller sized empty space ✔

The blank spaces must be of finite size, so we initially took the largest possible blank space of size $10^5 X 10^5$. Since we didn't know the size of the canvas required, we made a binary search algorithm to determine the size of the canvas. For each "size", we found if the blocks can be placed in that area. If it could, we reduced the size by a factor of 2, until I found an appropriate size.
But even this algorithm was not giving precise answer for the case of rectangular perfect fit. For example, Test Case:
g1 8 2
g2 4 4
g3 4 2
g4 4 2
g5 4 1

g6 4 1

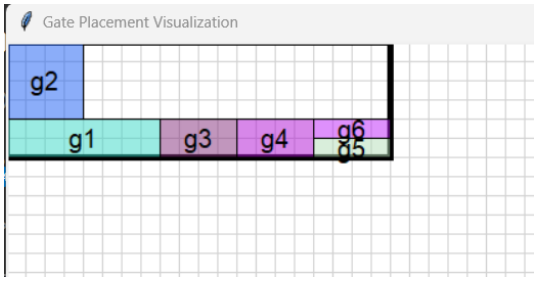

Figure 5: Output given by Implementation-1

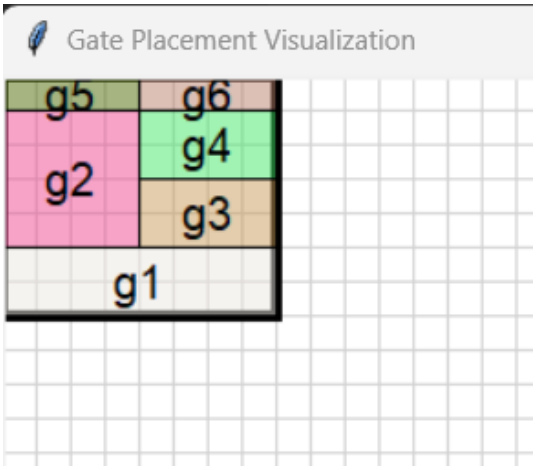

Figure 6: It can be perfectly fitted like this

2.1.1 Time complexity analysis

Binary Search = O($log(10^5)$)
Traversal through list and placing new empty spaces=O(n+n)=O(n)[1]
Iterating through all blocks, for a given size = O(n)
Overall = O($n^2 * log(10^5)$)

2.2 Implementation-2

On analyzing, we found that we were always taking a square canvas.
Then we separately applied binary search on height and width, which turned out to give better accuracy[2]. But this increased the time of execution.

2.2.1 Time complexity analysis

Binary Search=$O\,(\log(10^5)^2)$
Iterating through all blocks, for a given size = O(n)
Iterating in list of empty spaces + adding new block/adding empty spaces in list = O(n+n) = O(n)
Overall = O($n^2 * log(10^5)^2$)

2.3 Final Implementation

To further reduce time(which binary search was taking), we decided to increase the size of canvas from 0 rather than decreasing it from $10^5 X 10^5$. So we made a dynamic canvas, which

[1]Because we have added 2 elements in a sorted list, so complexity would be O(n)
[2]We calculated accuracy by dividing the sum of the area of blocks by the area bounding-box

increases size only when required i.e. when the block can't be fitted in any of the available blank spaces.

Figure 7: Growing canvas when there is no enough space

2.3.1 Time complexity analysis

Iterating through all blocks, for a given size = O(n)
Iterating through list of empty spaces + adding blocks/making empty spaces= O(n+n) = O(n)
Overall = O(n^2)

3 Test Cases

3.1 Random Test Cases

Note: These inputs and outputs are submitted as input files ;the corresponding names are in brackets

1. Test Case 1(custom_tc_1.txt and output_1.txt)
   Accuracy: 97.58%
   Time taken: 0.2709s

2. Test Case 2(custom_tc_2.txt and output_2.txt)
   Accuracy: 97.37%
   Time taken: 0.2904s

3. Test Case 3(custom_tc_3.txt and output_3.txt)
   Accuracy: 94.54%
   Time taken: 0.0859s

4. Test Case 4(custom_tc_4.txt and output_4.txt)
   Accuracy: 91.03%
   Time taken: 0.0390s

5. Test Case 5(custom_tc_5.txt and output_5.txt)
   Accuracy: 87.68%
   Time taken: 0.0156s

6. Test Case 6(custom_tc_6.txt and output_6.txt)
   Accuracy: 86.36%
   Time taken: 0.0129s

3.2 Some more random Test Cases

1. Test Case 1

Example Test Case:	Output:
g1 95 14	bounding_box 329 97
g2 72 97	g1 72 23
g3 74 2	g2 0 0
g4 77 58	g3 72 95
g5 97 23	g4 72 37
g6 84 74	g5 72 0
g7 76 51	g6 169 0
	g7 253 0

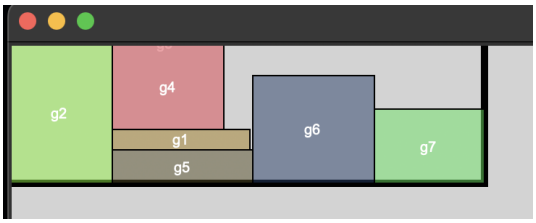

Figure 8: Our code output for above test case

2. Test Case 2

Example Test Case:	Output:
g1 99 17	bounding_box 99 108
g2 45 41	g1 0 0
g3 20 38	g2 15 17
g4 35 19	g3 60 17
g5 15 91	g4 15 58
	g5 0 17

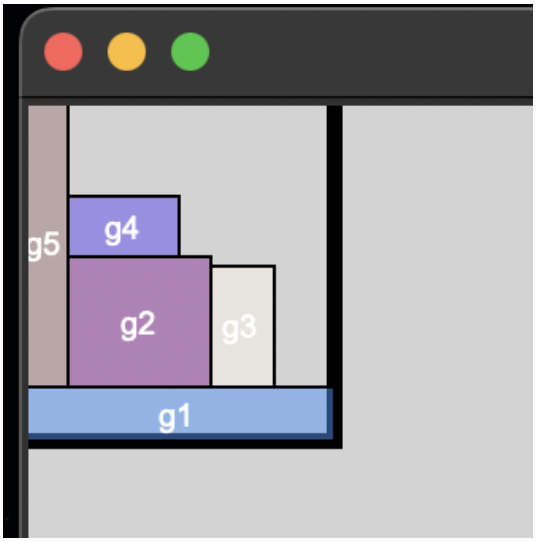

Figure 9: Our code output for above test case

3. Test Case 3

Example Test Case:
g1 18 67
g2 81 37
g3 42 89
g4 20 16
g5 95 23
g6 68 80
g7 99 24
g8 55 45
g9 47 47
g10 10 25

Output:
bounding_box 99 300
g1 68 173
g2 0 136
g3 0 47
g4 47 253
g5 0 24
g6 0 173
g7 0 0
g8 42 47
g9 0 253
g10 81 136

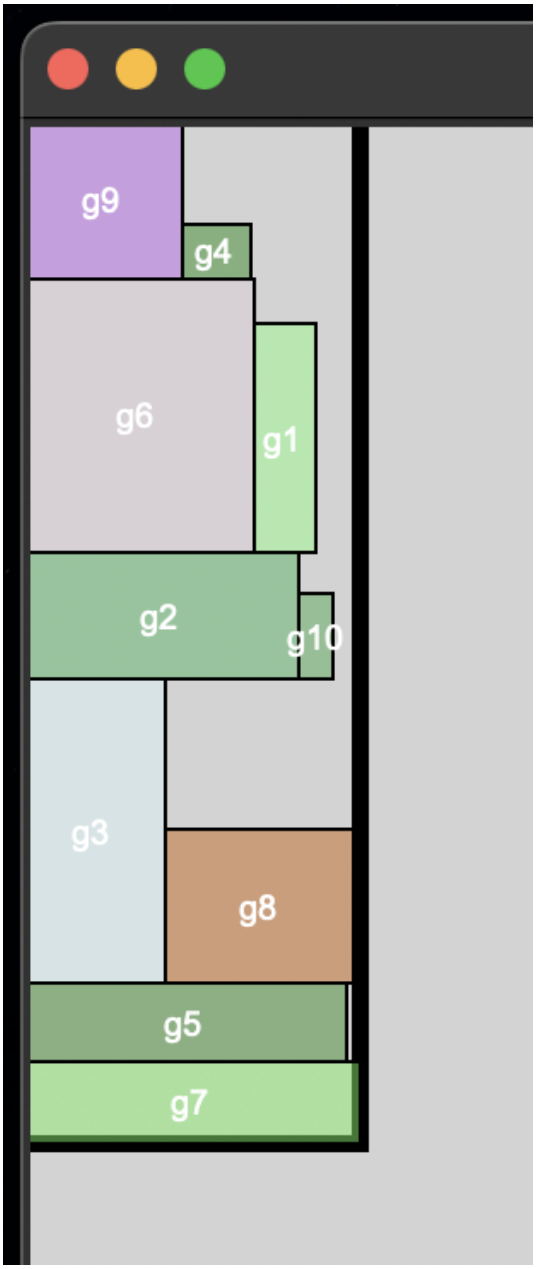

Figure 10: Our code output for above test case

4. Test Case 4

Example Test Case:
g1 36 18
g2 9 50
g3 30 31
g4 72 82
g5 19 66
g6 32 37
g7 74 86
g8 78 56
g9 18 42
g10 52 54

Output:
bounding_box 152 199
g1 102 147
g2 93 110
g3 0 168
g4 0 86
g5 74 56
g6 102 110
g7 0 0
g8 74 0
g9 74 122
g10 93 56

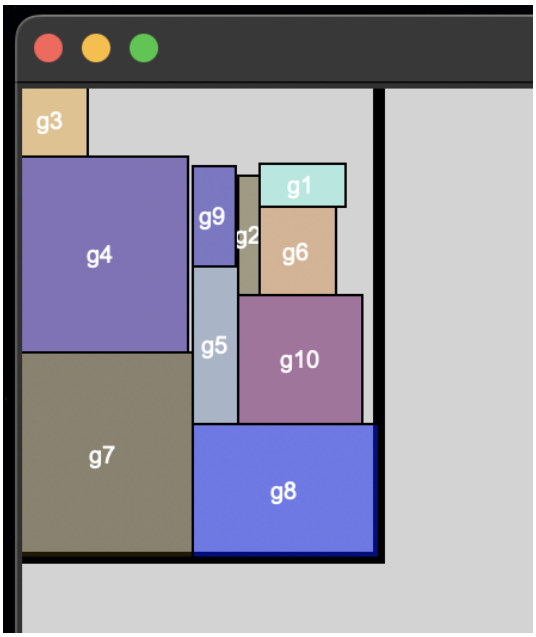

Figure 11: Our code output for above test case

3.3 Special Test Cases

3.3.1 Perfect Packing(i.e. cases where 100% filled space is possible)

1. Test Case 1
   Accuracy: 100%

Example Test Case:
g1 8 2
g2 4 4
g3 4 2
g4 4 2
g5 4 1
g6 4 1

Output:
bounding_box 8 7
g1 0 0
g2 0 2
g3 4 2
g4 4 4
g5 0 6
g6 4 6

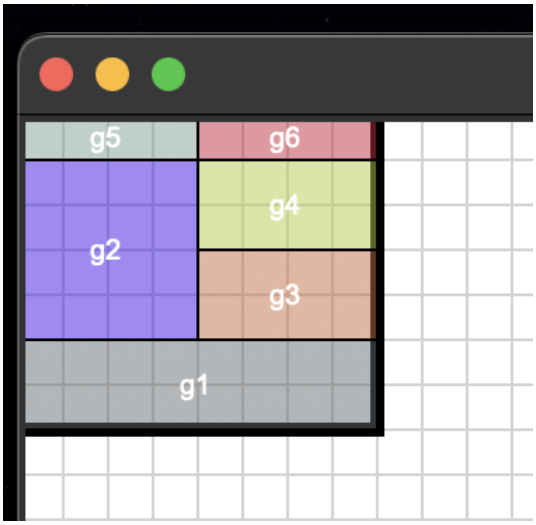

Figure 12: Our code output for above test case

2. Test Case 2
   Accuracy: 100%

Example Test Case:
g1 200 500
g2 50 50
g3 50 50
g4 50 50
g5 50 50
g6 50 50
g7 50 50
g8 50 50
g9 50 50
g10 50 50
g11 50 50
g12 50 50
g13 50 50
g14 50 50
g15 50 50
g16 50 50
g17 50 50
g18 50 50
g19 50 50
g20 50 50
g21 50 50

Output:
bounding_box 200 750
g1 0 0
g2 0 500
g3 50 500
g4 100 500
g5 150 500
g6 0 550
g7 50 550
g8 100 550
g9 150 550
g10 0 600
g11 50 600
g12 100 600
g13 150 600
g14 0 650
g15 50 650
g16 100 650
g17 150 650
g18 0 700
g19 50 700
g20 100 700
g21 150 700

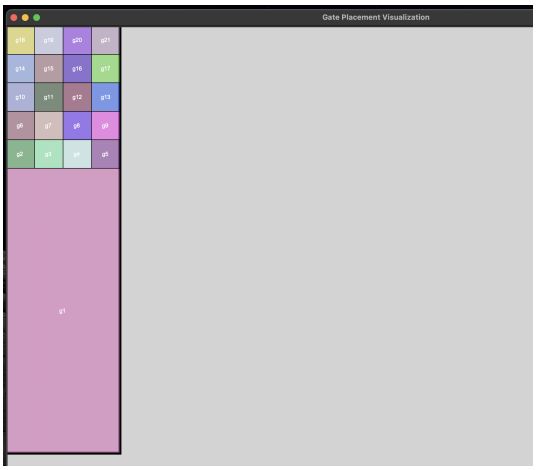

Figure 13: Our code output for above test case

3. Test Case 3
   Accuracy: 100%

Example Test Case:
g1 500 200
g2 250 200
g3 50 50
g4 50 50
g5 50 50
g6 50 50
g7 50 50
g8 50 50
g9 50 50
g10 50 50
g11 50 50
g12 50 50
g13 50 50
g14 50 50
g15 50 50
g16 50 50
g17 50 50
g18 50 50
g19 50 50
g20 50 50
g21 50 50
g22 50 50

Output:
bounding_box 1000 200
g1 0 0
g2 500 0
g3 750 0
g4 750 50
g5 750 100
g6 750 150
g7 800 0
g8 800 50
g9 800 100
g10 800 150
g11 850 0
g12 850 50
g13 850 100
g14 850 150
g15 900 0
g16 900 50
g17 900 100
g18 900 150
g19 950 0
g20 950 50
g21 950 100
g22 950 150

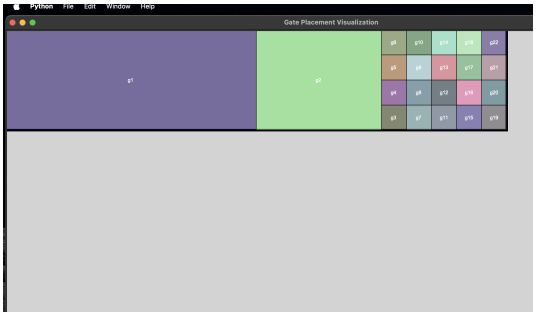

Figure 14: Our code output for above test case

4. Test Case 4
   Accuracy: 100%

Example Test Case:
g1 8 2
g2 4 4
g3 2 2
g4 2 2
g5 2 2
g6 2 2

Output:
bounding_box 8 6
g1 0 0
g2 0 2
g3 4 2
g4 6 2
g5 4 4
g6 6 4

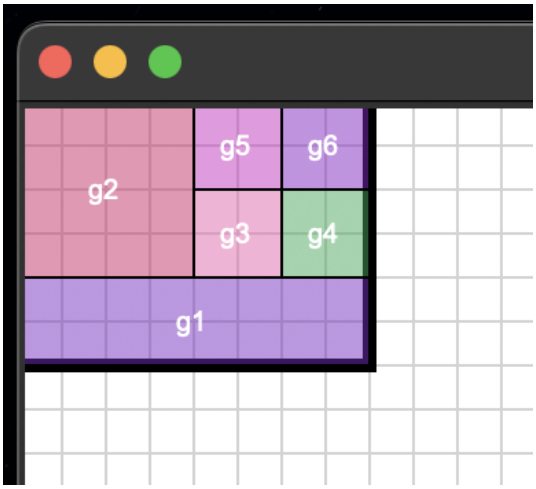

Figure 15: Our code output for above test case

3.4 Given Test Cases on moodle

1. Test Case 1
   Accuracy: 81 %
   Example Test Case:
   g1 3 10
   g2 8 3
   g3 6 6

   Output:
   bounding_box 11 10
   g1 0 0
   g2 3 0
   g3 3 3

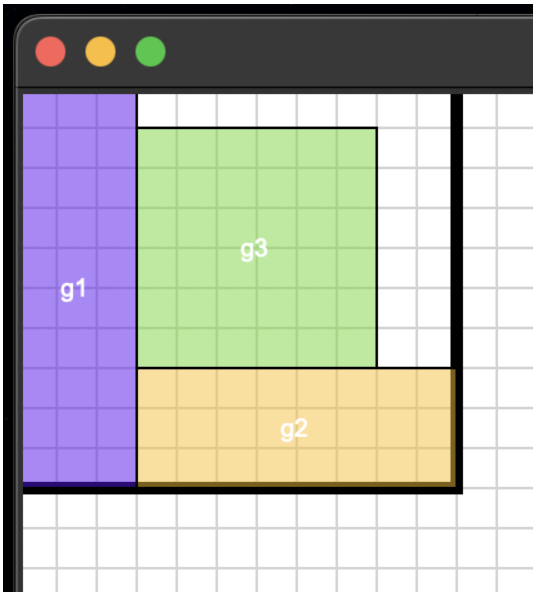

Figure 16: Our code output for above test case

2. Test Case 2
   Accuracy: 93.33%
   Example Test Case:
   g1 3 4
   g2 5 2
   g3 2 3

   Output:
   bounding_box 5 6
   g1 0 2
   g2 0 0
   g3 3 2

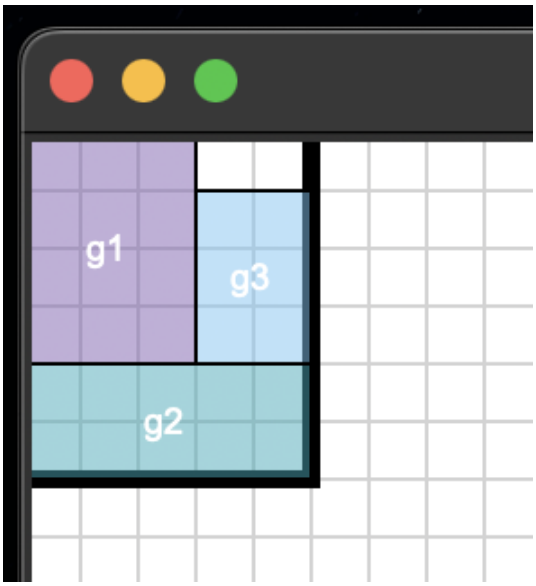

Figure 17: Our code output for above test case

3. Test Case 3
   Accuracy: 80.24%

Example Test Case:	Output:
g1 10 10	bounding_box 45 45
g2 20 5	g1 20 35
g3 5 20	g2 15 20
g4 15 10	g3 15 25
g5 10 15	g4 20 25
g6 25 5	g5 35 0
g7 5 25	g6 10 15
g8 30 10	g7 10 20
g9 10 30	g8 0 5
g10 35 5	g9 0 15
	g10 0 0

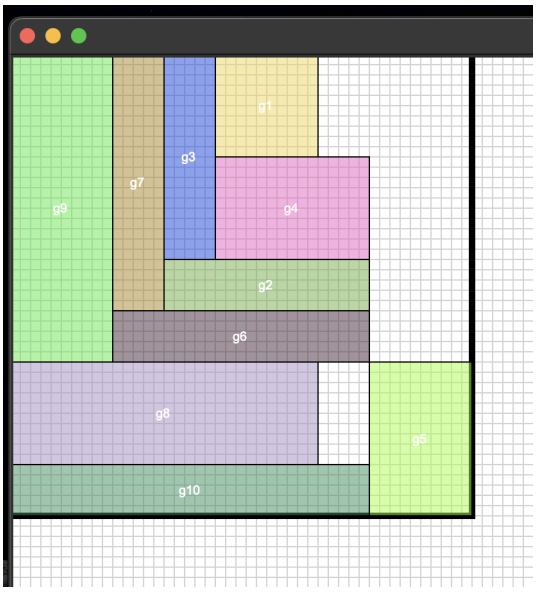

Figure 18: Our code output for above test case

4. Test Case 4
   Accuracy: 78.8%

Example Test Case:	Output:
g1 4 5	bounding_box 6 15
g2 6 2	g1 2 2
g3 2 6	g2 0 0
g4 3 4	g3 0 2
g5 5 3	g4 0 11
	g5 0 8

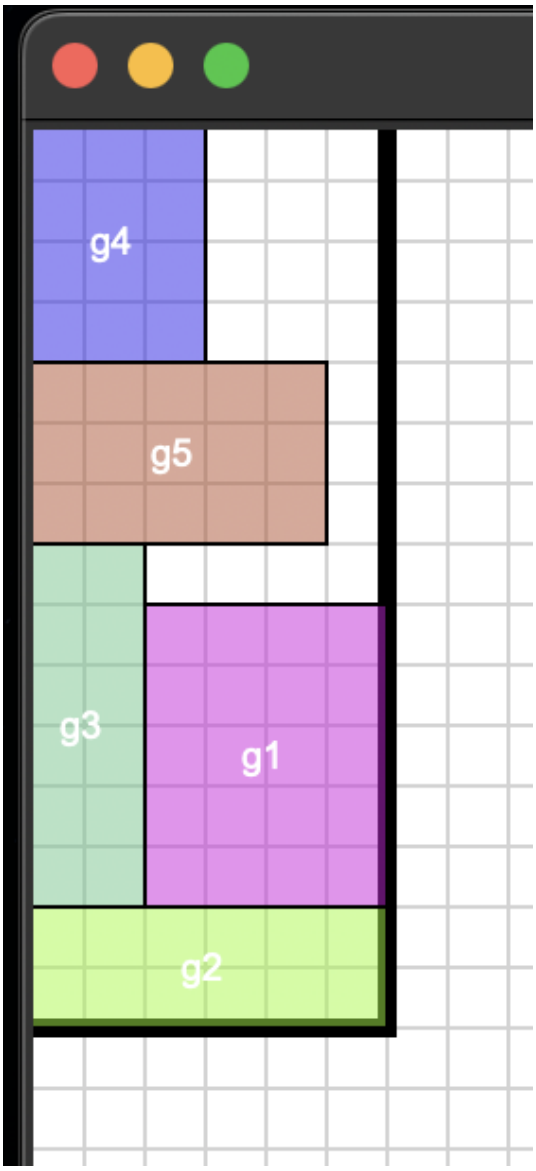

Figure 19: Our code output for above test case

5. Test Case 5(input file : input.txt)
   Accuracy: 95.78%
   Note: Due to large number of gates this testcase has been included in source files. Output file is named as: output.txt

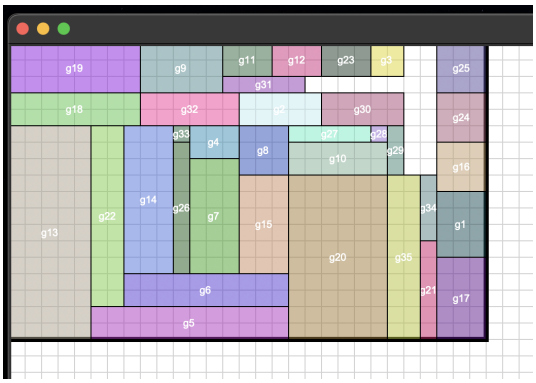

Figure 20: Our code output for above test case

4 Visualization for large number of gates

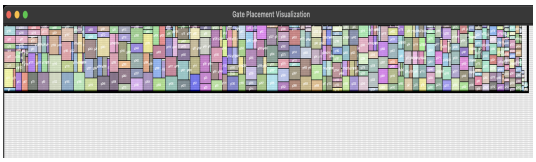

Figure 21: Our code output for random test case

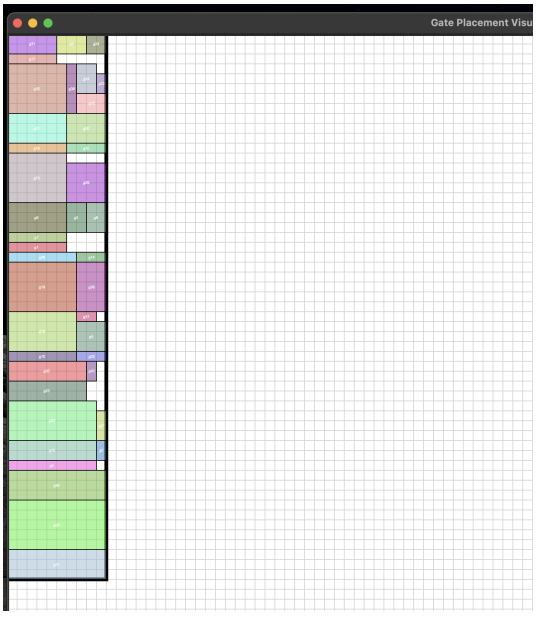

Figure 22: Our code output for random test case

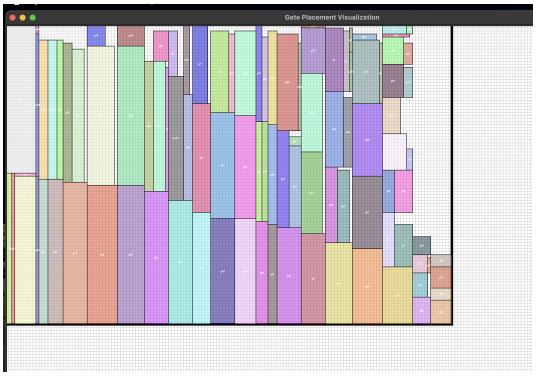

Figure 23: Our code output for random test case

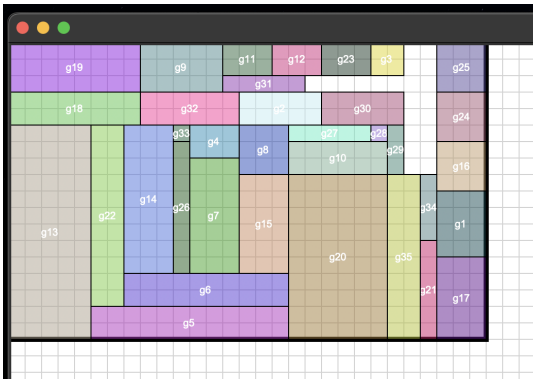

Figure 24: Our code output for random test case

5 Drawbacks of Final Implementation

1. For some test cases the output is maximised when gates are sorted by other criteria (like area, height, width) , while for some maxside (maximum of height and width) gives better results. On running over several test cases it turned out that maxside is efficient for more percentage of test cases.

Example Test Case:	Output:
g1 5 5	bounding_box 10 16
g2 10 5	g1 0 0
g3 6 3	g2 0 11
g4 9 2	g3 0 8
g5 9 1	g4 0 5
	g5 0 7

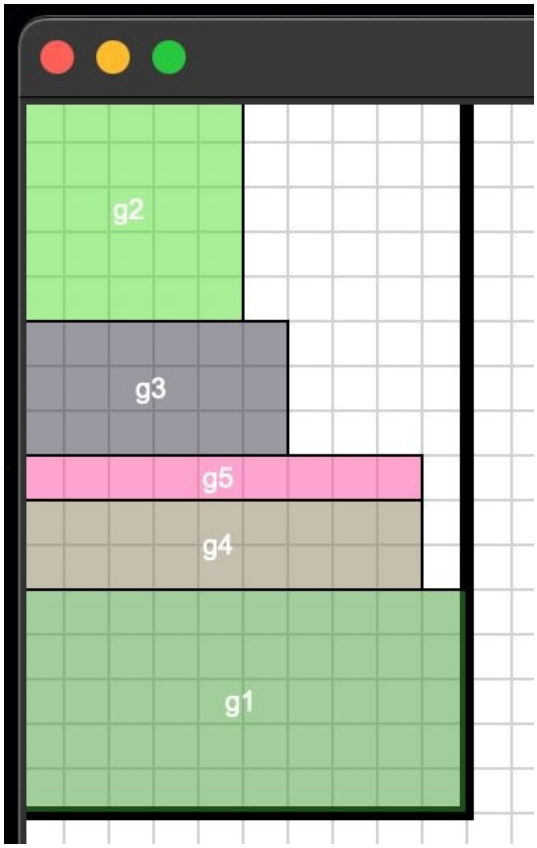

Figure 25: Our code output for above test case(75% filled space)

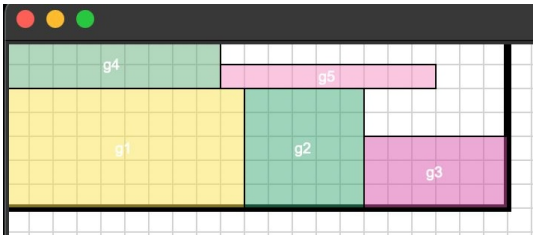

Figure 26: sorting by area gives better result (81 %)

2. In the add block function. We have fixed a criterion for deciding where to put a new block but different criteria may give better results for some test cases. Hence, our code does not give 100% for all cases where it is achievable.

Example Test Case:
g1 5 5
g2 10 5
g3 6 3
g4 9 2
g5 9 1

Output:
bounding_box 10 16
g1 0 0
g2 0 11
g3 0 8
g4 0 5
g5 0 7

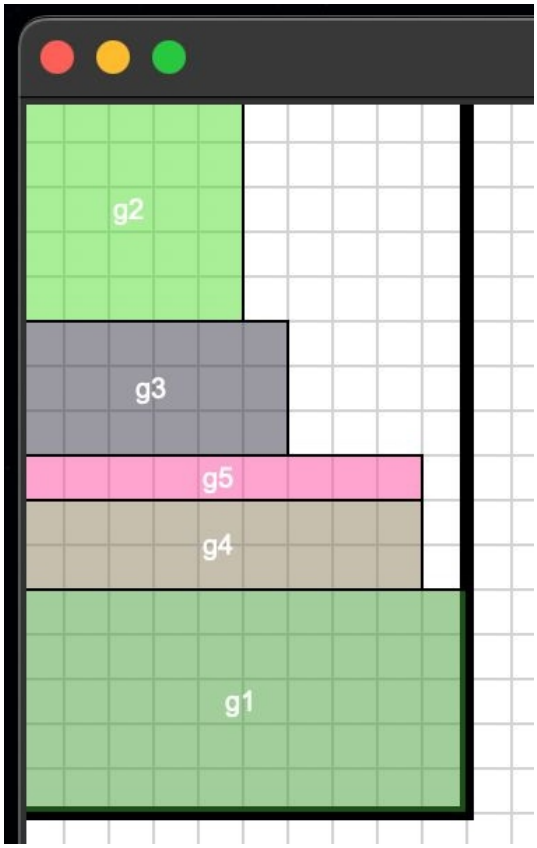

Figure 27: Our code gives only 75%

Figure 28: It can be perfectly fitted like this

6 Conclusion

Our code is having high accuracy reaching 90% or above for number of gates greater than 500 and between 70% to 90% for less . Time is always less than 1 sec even for the worst case.