COL215 SW Assignment 1 - Gate Packing

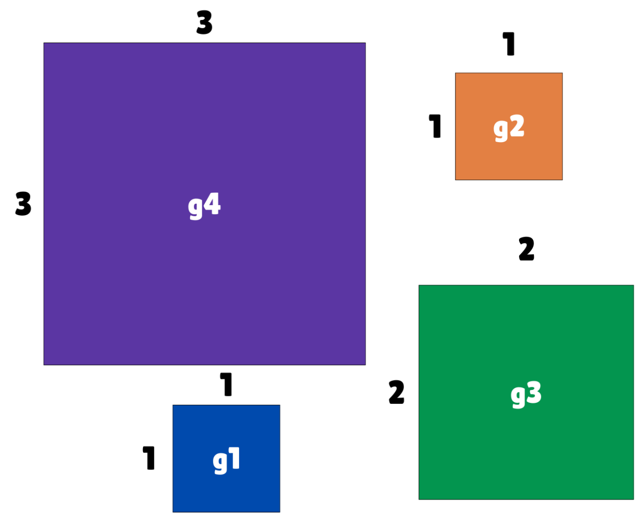
Submission By :

Yash Rawat **2023CS50334**

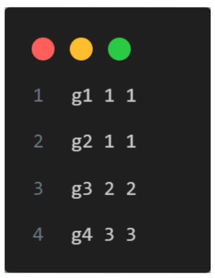
Priyanshi Gupta **2023CS10106**

Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

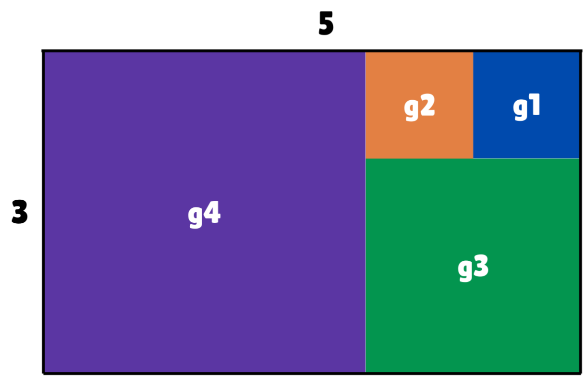
August 23, 2024



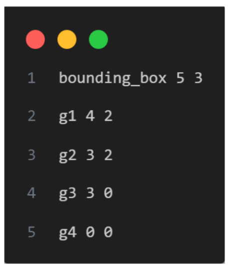
Input File



```
g1 1 1
g2 1 1
g3 2 2
g4 3 3
```



Output File



```
bounding_box 5 3
g1 4 2
g2 3 2
g3 3 0
g4 0 0
```

1 Modelling Gate Packing

1.1 What is Gate Packing ?

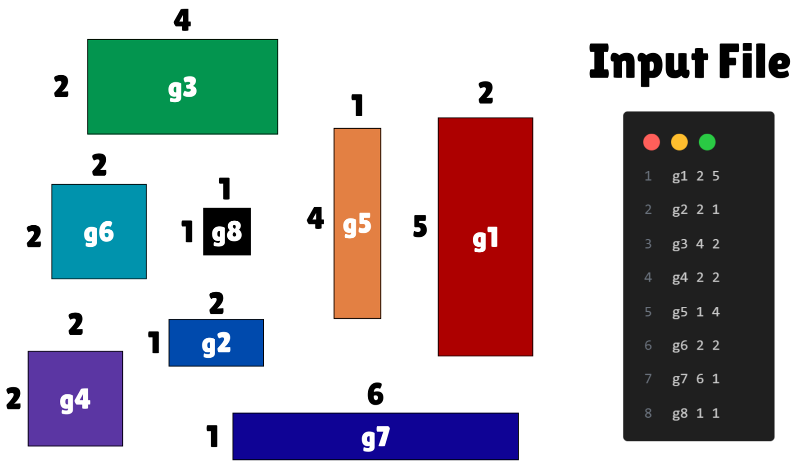
In the context of gate level circuit designing refers to the process of arranging logic gates on a circuit board in order to minimize wasted space, reduce interconnection length and optimize the overall layout.

Generalised gate packing is a very complex problem and involves multiple challenges such as placement and routing complexity, heat dissipation, design constraints due to fabrication processes, etc. but we will be tackling a simplified problem in this assignment.

1.2 Understanding the Problem Statement

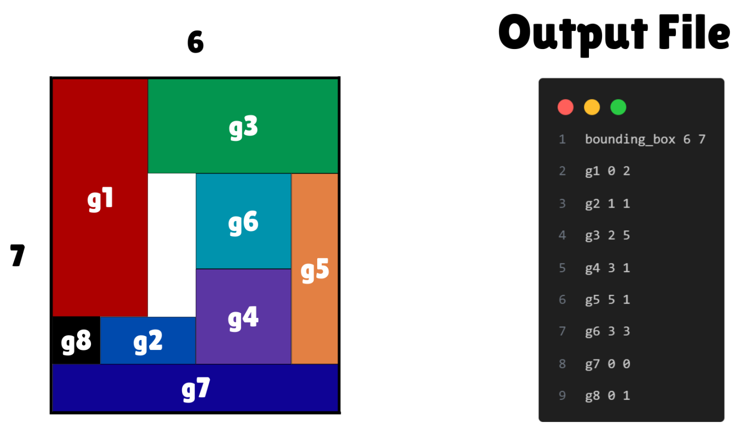
The problem statement models the gates as a set of n rectangles (provided as input for each test case) : $\{g_1, g_2, ..., g_n\}$ each represented a pair of integers : $g_i = (w_i, h_i)$, where w_i and h_i are the width and the height of the i^{th} board. A given set of gates is said to be "correctly assigned" if no two gates have a overlapping area. The bounding rectangle is defined to be the smallest rectangle that encloses all gates and has the minimum area (out of all the possible "correctly assigned" cases).

The program is supposed to output 2 things - The w and h of the bounding rectangle and the set of coordinates : $\{(x_i, y_i)\}_{i=1}^{i=n}$ - where (x_i, y_i) denote the coordinate of the bottom left corner of the g_i. A sample test case is given below : (Note that every gate placed is in the original orientation provided by test case, i.e. re-orientation by rotation is disallowed)



```
g1 2 5
g2 2 1
g3 4 2
g4 2 2
g5 1 4
g6 2 2
g7 6 1
g8 1 1
```

Figure 1: Sample Test Case with 8 gates



```
bounding_box 6 7
g1 0 2
g2 1 1
g3 2 5
g4 3 1
g5 5 1
g6 3 3
g7 0 0
g8 0 1
```

Figure 2: Output of above sample case

2 Algorithm Conceptualization and Design

Our all three Algorithms solve the problem in a Y Direction Flipped manner. Since the $(0, 0)$ of our grid co-ordinates are taken in the top left (Standard Matrix Notation) (which is different from the give problem statement where $(0, 0)$ is taken in bottom right) if any correct packing is found by our algorithm then it will also work in the original problem statement.

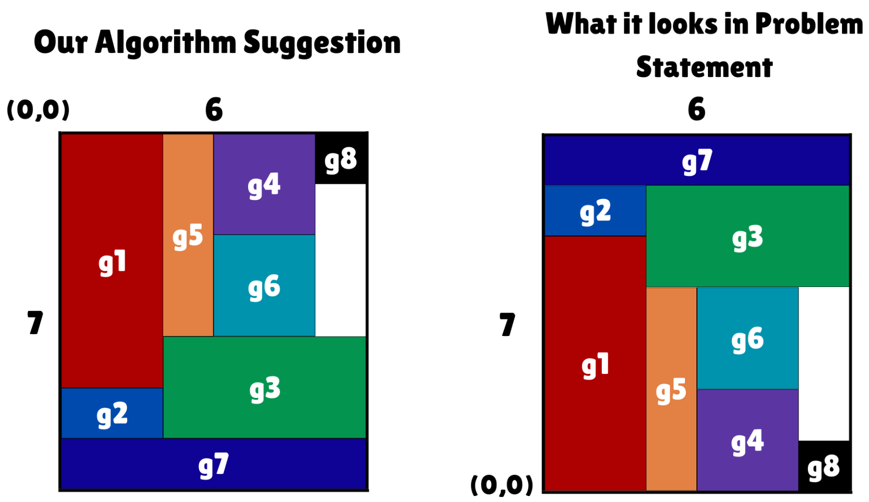


Figure 3: Y Flipped Output by our Algorithm

2.1 Naive Row Packing Algorithm

This is naive and the simplest way of packing rectangles.

1. Rectangles are sorted in decreasing order of height and width in respective priorities.
2. Rectangles are added to a row from left to right until next rectangle does not fit.In such case the next rectangle is placed in the next row.

2.2 Pixel Scan Algorithm

The naive row packing algorithm can be improved by manually scanning the entire image for a suitable location for every rectangle to pack. This process would be slower, but that can be improved later on. This is still extremely naive in terms of the actual packing logic.

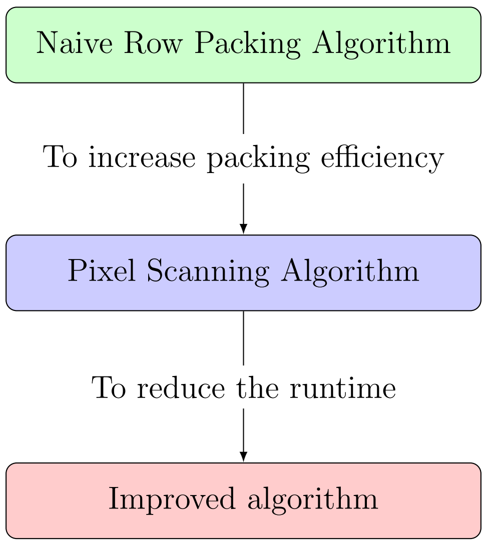
1. Rectangles are sorted in decreasing order of height (and then by width).
2. The algorithm implements a grid which stores value of every pixel.
3. We then loop over all our rectangles, and for each one go through every pixel and check if the top left and bottom right of the rectangle (some other pixel) will fit at that location. We also check if this rectangle will go outside the boundary.
4. We check all the pixels inside that rectangle to make sure we don't overlap with some other already placed rectangle.
5. Once we find a valid location we store location in the rectangle and mark those pixels as occupied in the grid.

2.3 Predictive-Pixel Scan Algorithm

1. The improved algorithm involves sorting rectangles by height (and then by width) just like the previous algorithms.
2. The algorithm implements a grid which stores 0 for unoccupied pixels and index value i of rectangle which occupies it.
3. We iterate through the grid but instead of looping through every pixel and checking its state , we check if a pixel is occupied or not. If it is occupied then we fetch the rectangle width using the index and calculate the nearest column to the right which is not being covered by that rectangle and move our checking coordinates to that column (same row).If we exceed the width of image then we just move to the first column of the next row and continue with the iteration.(This step makes the algorithm more efficient).

4. If empty pixel is returned then the grid is checked to ensure that the next rectangle can be placed in it or not.If it encounters a covered pixel the outer loop is continued(CHECK).

5. Once we find suitable position for the rectangle it fills the cells with index of the covering rectangle , sets the packed state of rectangle object, as well as its coordinates.



2.4 Proving Correctness of algorithm

2.4.1 Selection of Initial Width of Packing

1. The first choice of number of rows and columns in the grid is done by calculating the total area of the rectangles to be packed.Both the height and width are taken to be $\lfloor 1.1 \cdot \sqrt{A} \rfloor$ where A is the total area of gates to be packed.

$$W_0 = \lfloor (1.1 \cdot \sqrt{A} \rfloor$$

2. In case there is no possible packing arrangement we reset the width to 1.5 of width in previous iteration until the packing is done.

$$W_{i+1} = \lfloor 1.5 \cdot W_i \rfloor$$

3. If we ensure that our rows are more than the maximum height of the gates then this process will terminate in finite steps since a feasible upper bound on the number of columns is the sum of widths of all gates (as in the Naive Row Packing Algorithm).

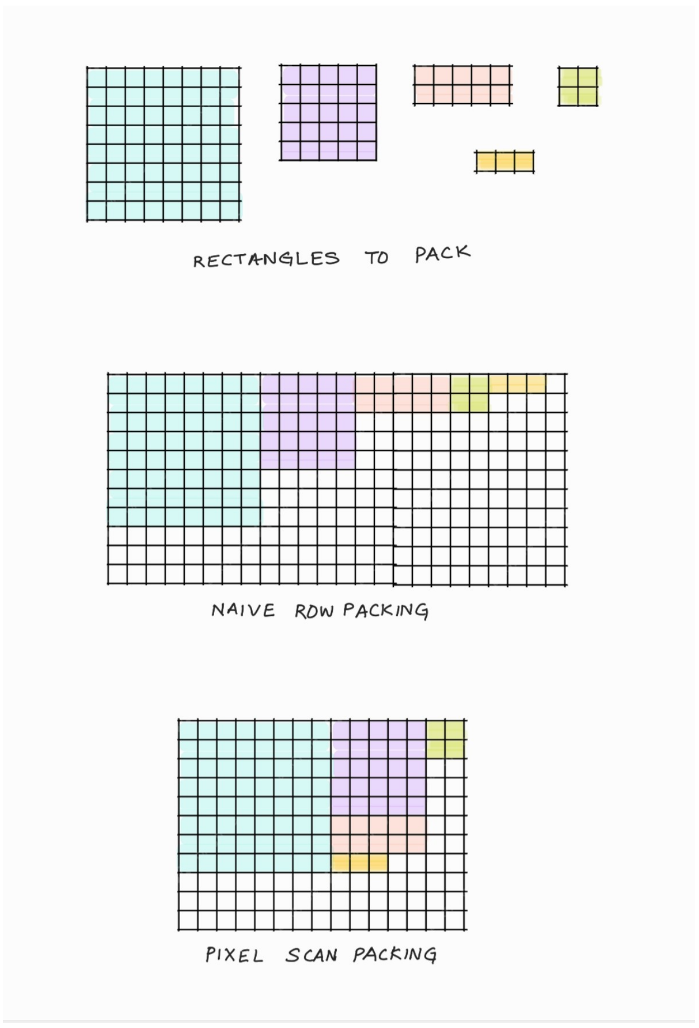


Figure 4: Gate Packing

2.4.2 Optimizing the Algorithm

After a satisfactory packing arrangement is found we try to find a better arrangement by varying the height and width of the grid.

1. We vary the width of the grid in intervals of 2.5% of initial width and adjust height of grid accordingly.
2. The time taken and packing efficiency are calculated for the adjusted width and height. A maximum of 20 iterations are carried out to find the best packing.
3. The iterations stop once packing efficiency exceeds 95% and runtime is below 2-3 sec.

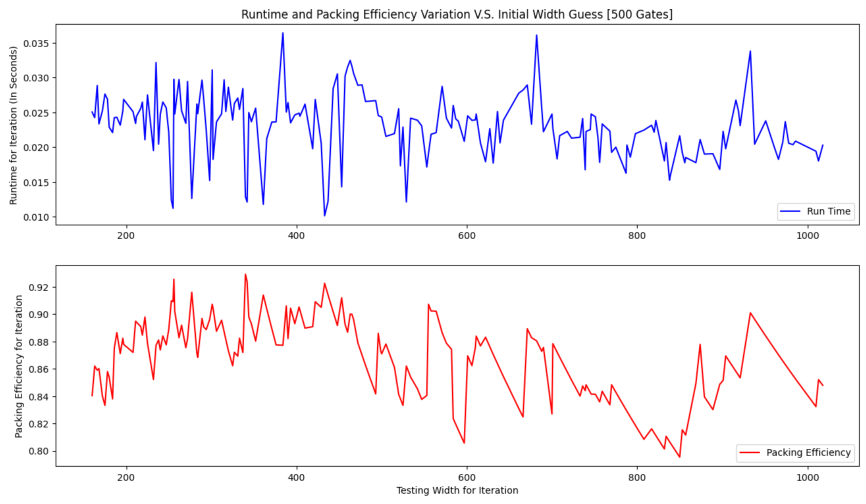


Figure 5: Runtime & Packing Efficiency Variation vs Width (32 gates)

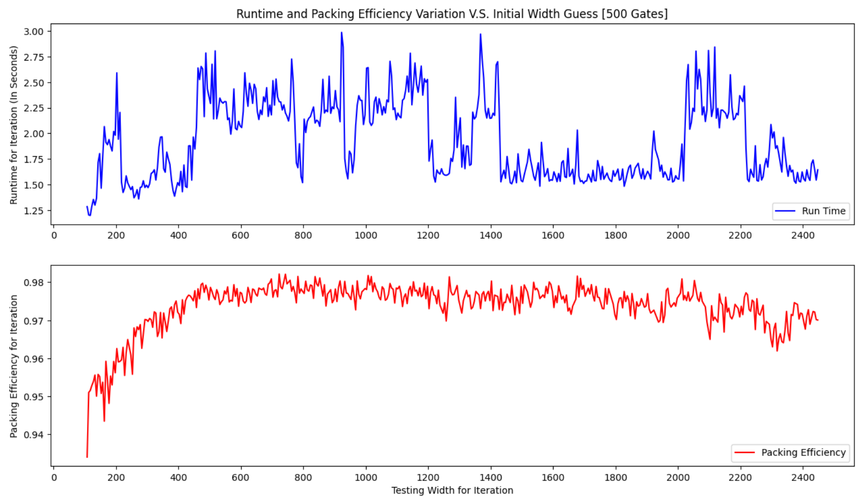


Figure 6: Runtime & Packing Efficiency Variation vs Width (500 gates)

2.4.3 Correctness of Algorithm

1. In this variant, the small rectangles can have varying lengths and widths, and their orientation is fixed (they cannot be rotated). The goal is to pack them in an enclosing rectangle of minimum area, with no boundaries on the enclosing rectangle's width or height.The problem is NP-complete in general which means that the time required to solve the problem using any currently known algorithm increases rapidly as the size of the problem grows.

2. The methods of solving this problem can be categorized into two types: exact

algorithms and heuristic algorithms. It is well-known that the exact algorithms can only solve small-scale instances within a reasonable computational time.

3. Therefore, we have proposed a new heuristic rectangle packing algorithm to maximize the area usage of the box.The algorithm involves greedy placement of rectangles from large to small.

4. If the largest rectangle remaining can't fit anywhere, we place it in a place that extends the pack region as little as possible.It is not perfect but it is easy to implement gives near perfect packing and decent runtime.

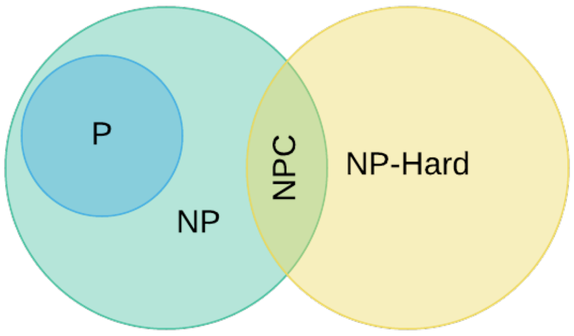


Figure 7: P vs NP Problems

3 Time Complexity Analysis

No. of rectangles	TC 1	TC 2	TC 3	TC 4	TC5
Time Taken (in sec)	0.005484	0.009657	0.008263	0.004607	0.004479
Packing Efficiency	0.818181	0.933333	0.902778	0.845238	0.952381

3.1 Plot of n vs T (n is the size of the test case

4 Visualising Output on Multiple Test Cases

4.1 Test Case 1

Gate No.	Input (w, h)	Output (x, y)
1	$(3, 10)$	$(0, 0)$
2	$(8, 3)$	$(3, 0)$
3	$(6, 6)$	$(3, 6)$

Bounding Box Dimension : Width $= 11$ **, Height** $= 10$

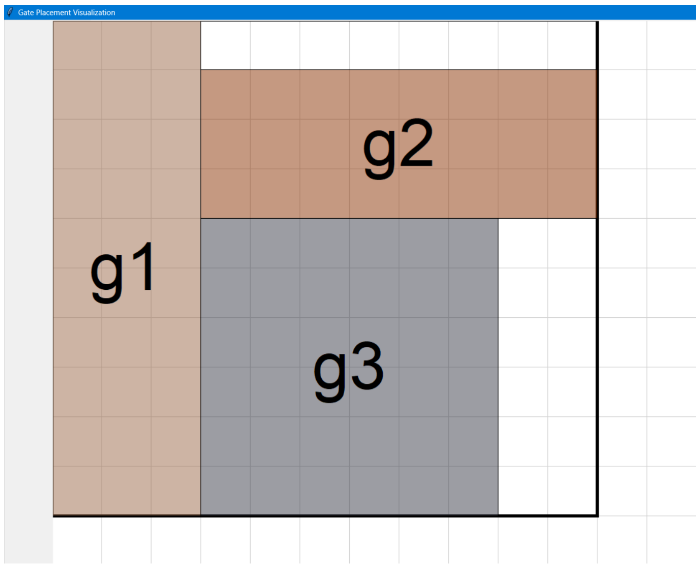
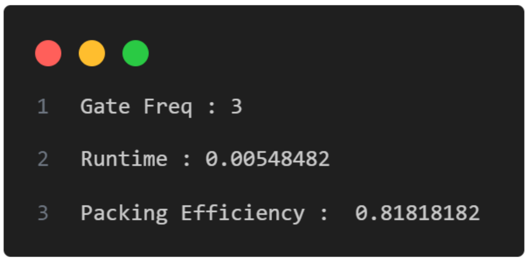


Figure 8: Gate Packing



```
Gate Freq : 3
Runtime : 0.00548482
Packing Efficiency :  0.81818182
```

Figure 9: Report

4.2 Test Case 2

Gate No.	Input (w, h)	Output (x, y)
1	$(3, 4)$	$(0, 0)$
2	$(5, 2)$	$(3, 0)$
3	$(2, 3)$	$(0, 4)$

Bounding Box Dimension : Width $= 5$, **Height** $= 6$

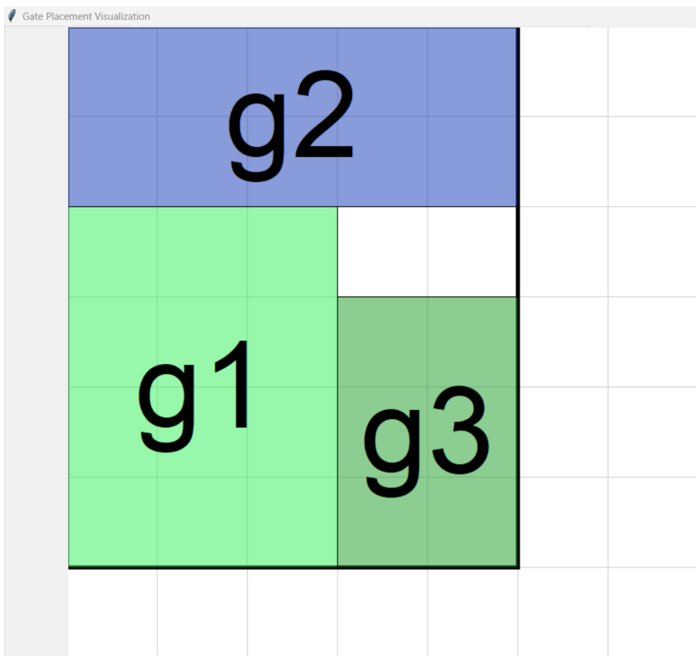
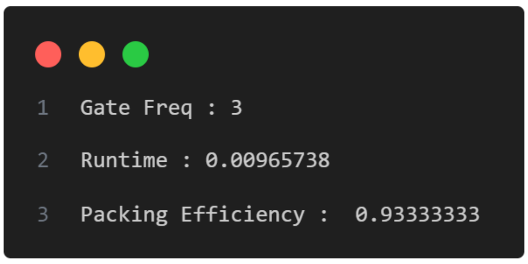


Figure 10: Gate Packing



```
Gate Freq : 3
Runtime : 0.00965738
Packing Efficiency :  0.93333333
```

Figure 11: Report

4.3 Test Case 3

Gate No.	Input (w, h)	Output (x, y)
1	$(10, 10)$	$(45, 10)$
2	$(20, 5)$	$(35, 25)$
3	$(5, 20)$	$(15, 0)$
4	$(15, 10)$	$(30, 10)$
5	$(10, 15)$	$(20, 0)$
6	$(25, 5)$	$(10, 25)$
7	$(5, 25)$	$(100, 0)$
8	$(30, 10)$	$(30, 0)$
9	$(10, 30)$	$(0, 0)$
10	$(35, 5)$	$(15, 20)$

Bounding Box Dimension : Width $= 60$ **, Height** $= 30$

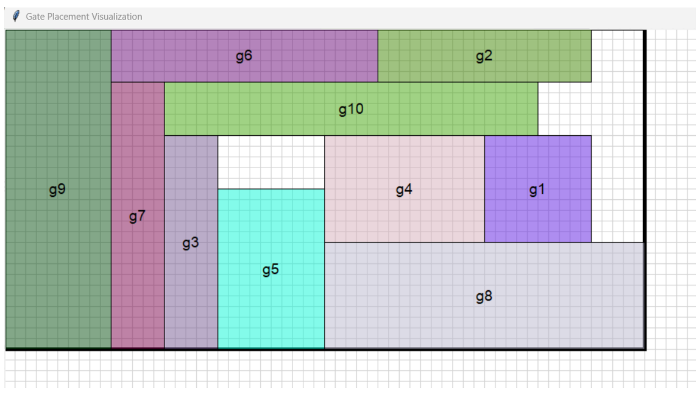
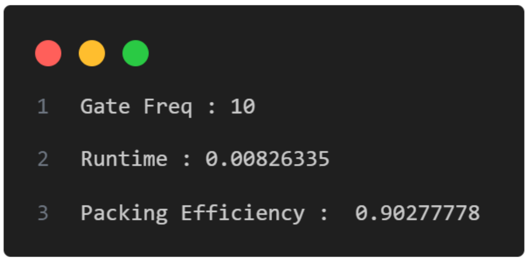


Figure 12: Gate Packing



```
Gate Freq : 10

Runtime : 0.00826335

Packing Efficiency :  0.90277778
```

Figure 13: Report

4.4 Test Case 4

Gate No.	Input (w,h)	Output(x,y)
1	$(4, 5)$	$(2, 0)$
2	$(6, 2)$	$(6, 4)$
3	$(6, 0)$	$(0, 0)$
4	$(3, 4)$	$(6, 0)$
5	$(5, 3)$	$(9, 0)$

Bounding Box Dimension : Width $= 14$ **, Height** $= 6$

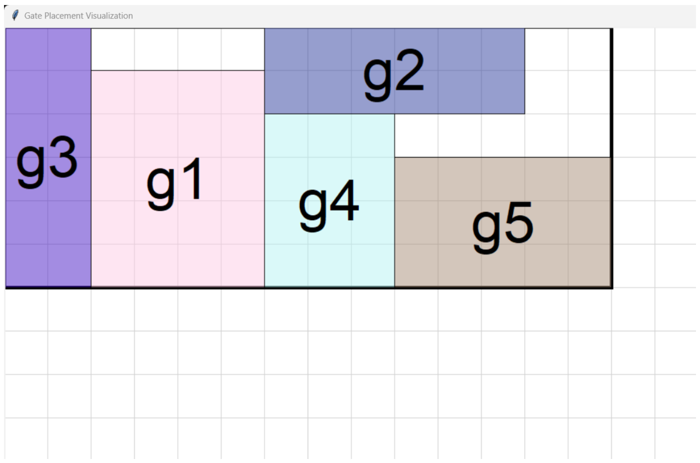
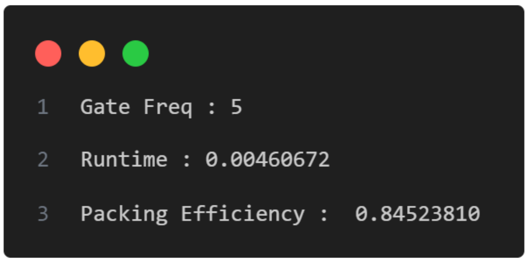


Figure 14: Gate Packing



```
Gate Freq : 5
Runtime : 0.00460672
Packing Efficiency :  0.84523810
```

Figure 15: Report

4.5 Test Case 5

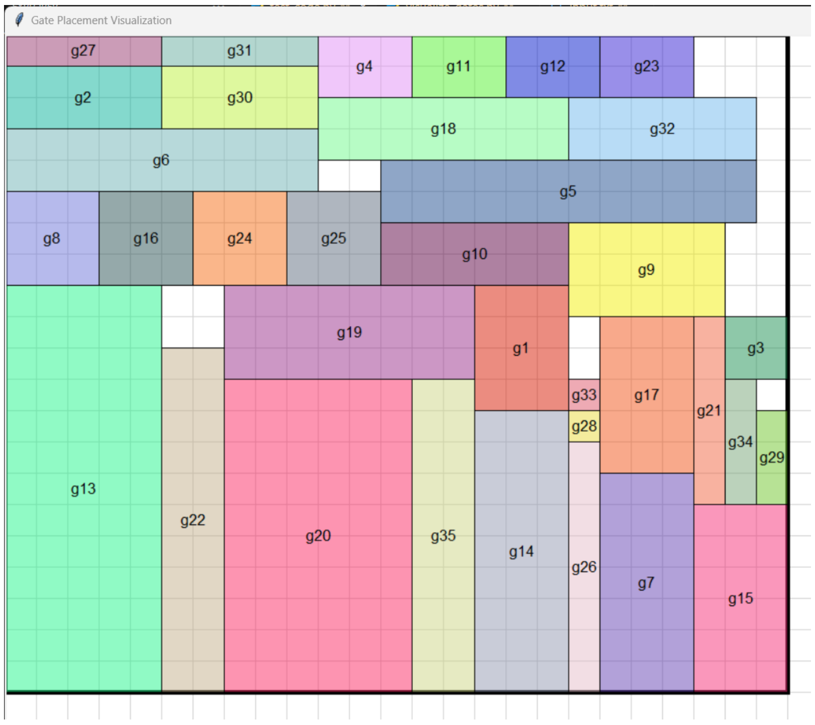
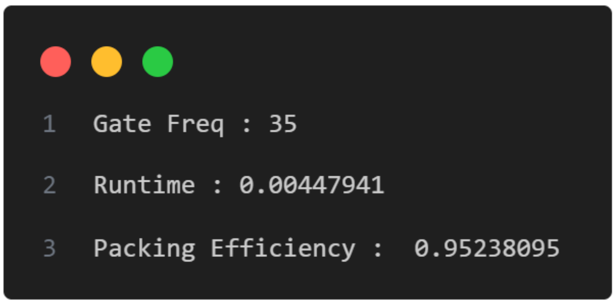


Figure 16: Gate Packing



```
Gate Freq : 35
Runtime : 0.00447941
Packing Efficiency :  0.95238095
```

Figure 17: Report