COL215 SW Assignment 1 - Gate Packing

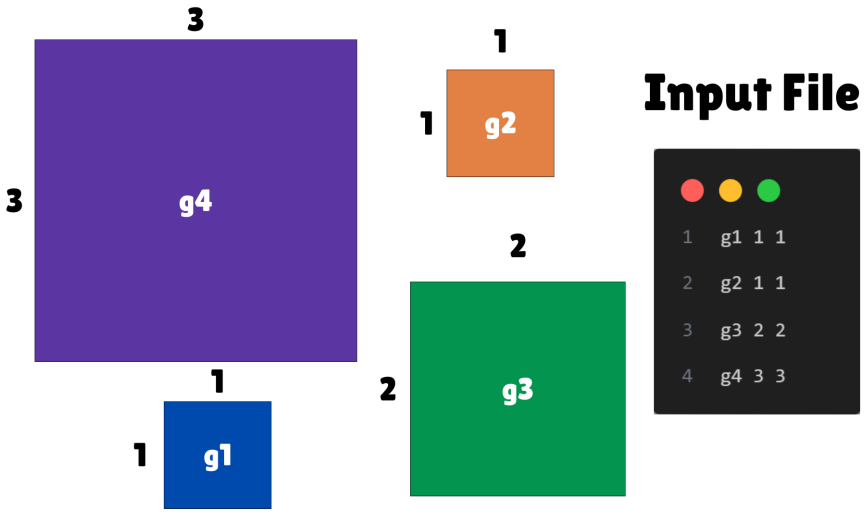
Submission By :

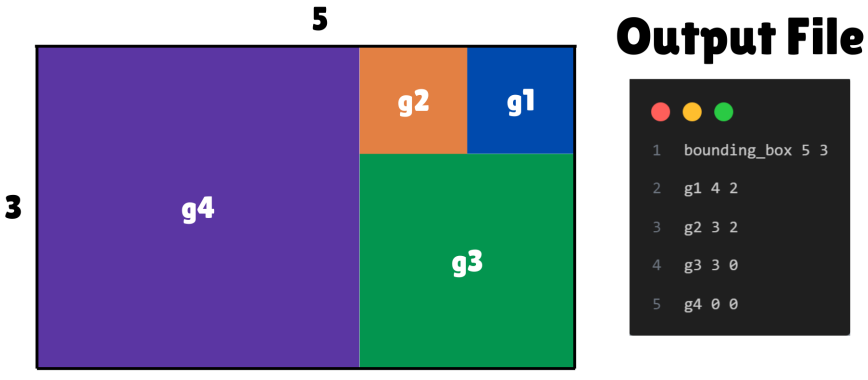
Yash Rawat **2023CS50334**

Priyanshi Gupta **2023CS10106**

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

August 24, 2024





1 Modelling Gate Packing

1.1 What is Gate Packing ?

In the context of gate level circuit designing, it 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 of the circuit board.

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 by 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 overlapping areas. 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)

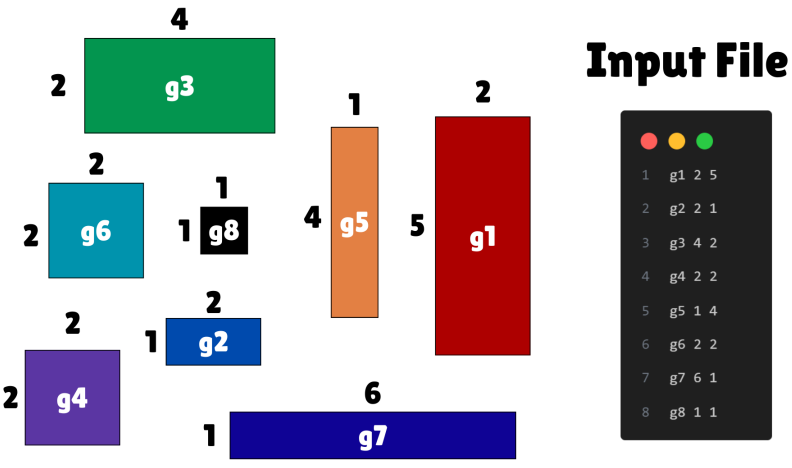


Figure 1: Sample Test Case with 8 gates

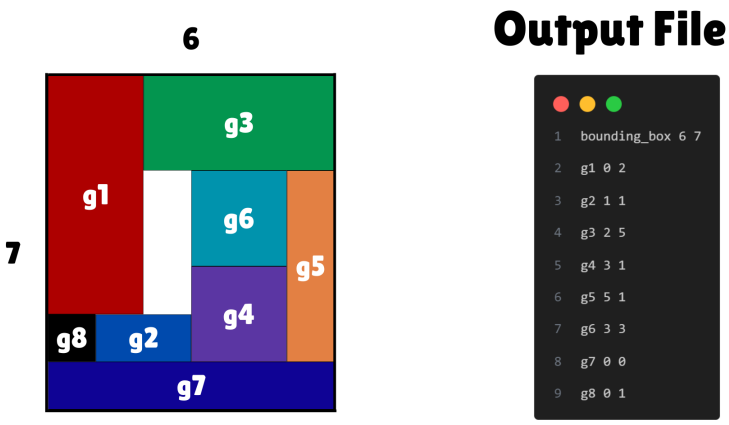


Figure 2: Output of above sample case

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 (allowing us to iterate the grid in row-major order) (which is different from the given problem statement where $(0, 0)$ is taken in bottom right). Hence if any correct packing is found by our algorithm then it will also work in the original problem statement.

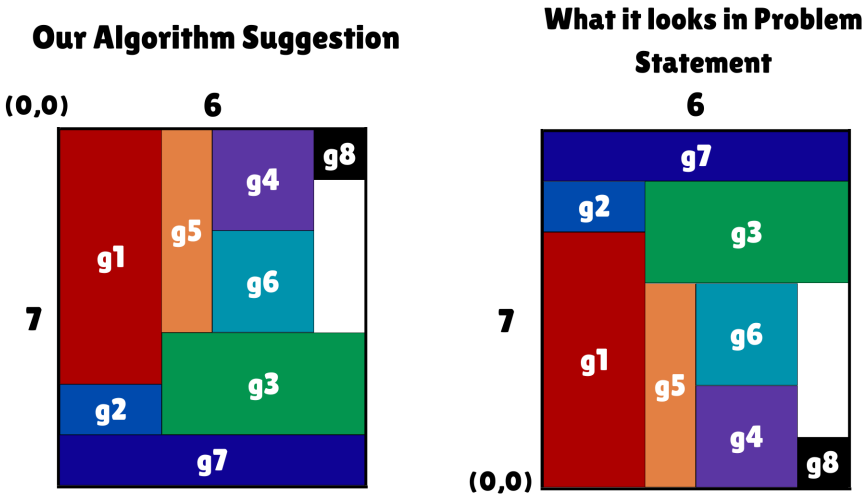


Figure 3: Y Flipped Output by our Algorithm

2.1 Naive Row Packing Algorithm

This is a naive and one of the simplest ways to pack rectangles and involves following steps :

1. Rectangles are sorted in decreasing order of height and width in respective priorities by the IO-Parser.
2. Rectangles are added to a row from left to right consecutively until all rectangles are placed side-by-side.

2.2 Pixel Scan Algorithm

The Pixel Scan algorithm involves generating a grid and scanning the it (pixel by pixel) to identify a suitable location for every rectangle to be packed. Although this method introduces a slower process, further optimizations can be implemented in subsequent iteration. However, the packing logic remains quite rudimentary in its current state :

1. Rectangles are sorted in decreasing order of height (and then by width) by the IO-Parser.
2. The algorithm implements a grid which stores value (1/0) of every pixel.
3. We then iterate through all the rectangles and, for each one, examine every pixel to determine whether the corresponding pixels to top-left and bottom-right corners of the rectangle are unoccupied. Additionally, we ensure that the rectangle does not extend beyond the boundaries defined by the grid.
4. We examine all the pixels within the potential rectangular region to ensure there is no overlap with any previously placed rectangles..
5. Once we find a valid location we store location in the rectangle and mark those pixels as occupied (1) in the grid.

2.3 Predictive-Pixel Scan Algorithm

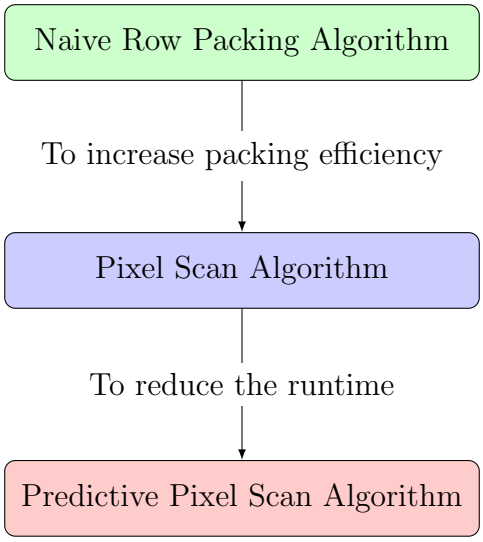
The Pixel Scan Algorithm can be optimized by reducing the number of iterations in the outermost loop. This can be achieved by leveraging the data from previously packed rectangles in the following manner :

1. The improved algorithm involves sorting rectangles by height (and then by width) by the IO-Parser, 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's width using the index and calculate the nearest right column 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 compared to before).

4. If an empty pixel is identified, the corresponding sub-grid is evaluated to determine whether the given rectangle can be placed within it. If a covered pixel is encountered, a similar calculation to that in Step 3 is performed, after which we return to Step 3 to continue iterating through the grid.

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 Completion of Algorithms

2.4.1 The Naive Row Packing algorithm

The correctness of this algorithm is trivial (from the way it is implemented) and can be shown by explicit construction in every case such as in Fig. 4.

2.4.2 Pixel Scan - Choice of Initial Bounding Box Dimensions

1. The first choice of number of rows and columns in the grid is done by calculating the total area of the gates to be packed. Initially, 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 :

$$W_0 = H_0 = \lfloor (1.1 \cdot \sqrt{A} \rfloor$$

Since there can be test cases where W_0 is less than the maximum width or H_0 is less than the maximum height - In which case we set either dimension to be floor of 1.5 times the corresponding maximum gate dimension.

2. In case no possible packing arrangement exists, we set the width to floor of 1.5 times the width of previous iteration until the packing is done :

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

3. Since we have ensured that our rows are more than the maximum height of the gates, this process (Step 2) will terminate in a 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).

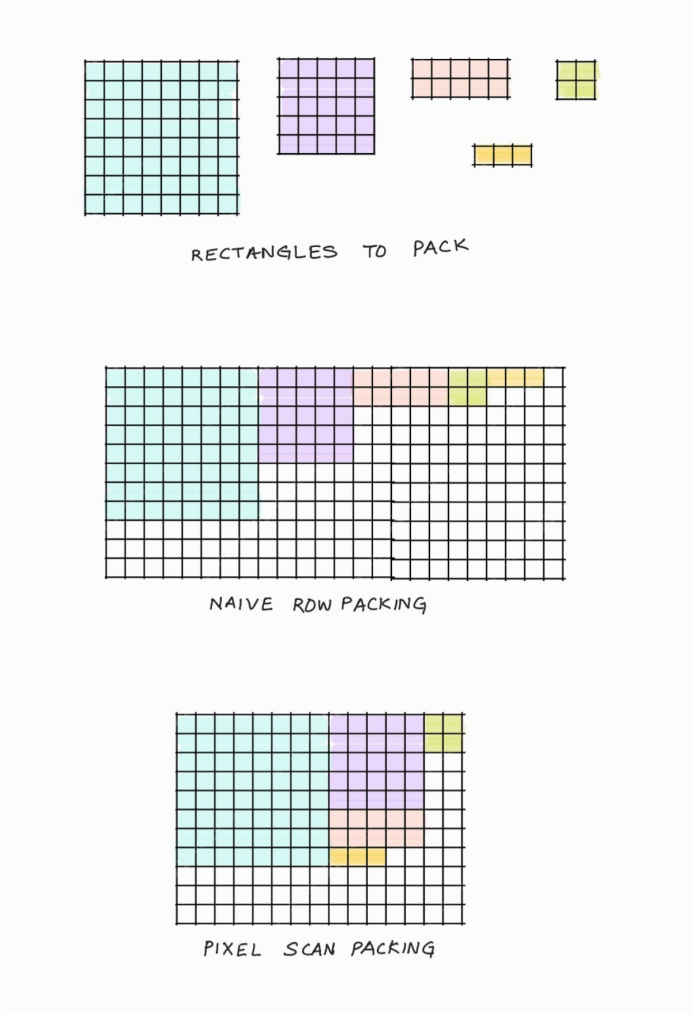


Figure 4: Visualising Output of different algorithms

2.4.3 Optimizing the Algorithm - Introducing Multiple Iteration Logic

After a satisfactory packing arrangement is found we may try to find a better arrangement by varying the height and width of the grid if the time of single iteration is very small.

1. We vary the width of the grid in intervals of 2.5% of initial width and adjust the height of grid accordingly (Up-to a maximum of 20 times or a bound calculated by estimating runtime from the first single packing iteration, the minimum of two.)

2. Then the algorithm runs another iteration on adjusted width and height. If the new output has a higher packing efficiency, then the maximum value and packing data is updated as a potential solution.

3. The iterations stop once packing efficiency exceeds 95% or runtime exceeds 2-3 seconds. The final packing with the highest packing efficiency is returned.

2.4.4 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 with number of rectangles and cannot be done in polynomial time.

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, after which they become unfeasible for larger input sizes.

3. Therefore, we have proposed a new heuristic rectangle packing algorithm to maximize the area usage of the box while having decent runtime.The algorithm involves greedy placement of rectangles from large to small as well as efficient grid-checking to improve run time.

4. If the current rectangle can't fit anywhere, the code exits and then we extend our packing region as little as possible. It is not perfect but it is easy to implement gives a good packing with a decent runtime.

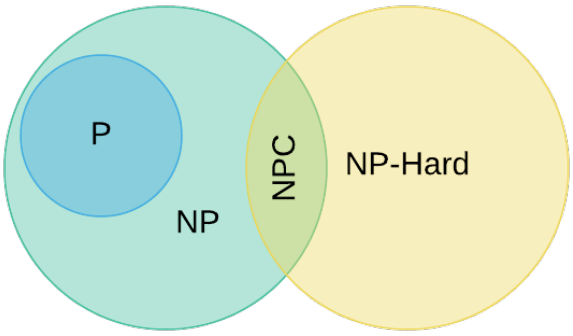


Figure 5: P vs NP Problems

3 Project Structure & Design Decisions

3.1 Files of Code Directory

In addressing the problem statement, which encompassed the Input/Output requirements, Test Case generation, and analysis, we adopted a structured approach. Multiple Python files were meticulously made to manage different aspects of the problem. Each file was documented and organized to handle specific components of the solution, ensuring clarity and readability in the overall implementation even for a third party. Detail of particular files are given below :

1. **code_timer.py** : This module defines the wrapper functions time_it and time_it_no_out, which are utilized for measuring the execution time of functions. These functions assist in graphical analysis by providing precise timing information.

2. **IO_Parser.py** : This module facilitates the parsing of input and output, converting data related to packed rectangles into a user-readable format and vice versa.

3. **Pack_by_Pixels.py** : This module contains the implementation of the Naive Row Packing, Pixel Scanning, and Predictive Pixel Scanning algorithms.

4. **project_constants.py** : As the number of constants used across the programs increased, a dedicated Python file was created to manage and track these constants. This organization facilitated the simultaneous testing of multiple test cases by generating the corresponding file paths. Additionally, it enabled adjustments to the implementation to optimize performance and achieve better results.

5. **Rect.py** : This module defines a custom class -*Rect*, which represents the gates utilized in the packing algorithms. It also includes the Rec_Data_Analysis method, which generates key statistics about the rectangles within a given test case.

6. **test_code.py** : This file defines the logic for both single and multiple iterations, includes various methods for analyzing different types of test cases and generating corresponding output files. It serves as the central component of the assignment, orchestrating the overall operation and ensuring cohesive functionality throughout the process.

7. **testcase_gen.py** : This component assists in generating both single and multiple test cases, tailored according to gate frequency. It incorporates three distinct modes for test case generation, utilizing the NumPy library. Additionally, it provides functionality to delete Input/Output files associated with previous test cases, ensuring a clean and organized testing environment.

8. **txt_analysis.py** : This file use Matplotlib to analyze the report files generated by the preceding files. This analysis involved visualizing the data through various graphs, facilitating a comprehensive understanding of the performance metrics and outcomes.

3.2 Multiple vs Single Iteration - Which is better ?

After implementing the Predictive Pixel Scan algorithm, we encountered a dilemma - whether to run multiple iterations for cases where run time is smaller in order to go for higher packing efficiency or let the implementation be naturally fast for all frequency of gates.

It is important to note that the behaviour of multi iteration logic becomes the same as that of single iteration above a certain threshold of frequency of gates. This is due to the implementation explained in Subsection 2.4.3 - Since the packing efficiency as well as time (for a single successful packing) grows with the number of gates , the code exits due to point 3 at the first iteration itself.

After crunching some data across multiple test cases we came to the conclusion that when the gate frequency is less than 250-300, the gain in packing efficiency is really noticeable only for < 50 gates.Thus the trade-off ≈ 20 times more runtime is only viable if any theoretical/practical (industrial) cost function (used for analysing output) weighs heavily towards packing efficiency.

Thus if we prioritise for Run Time, say we want to generate multiple designs very quickly then single iteration is useful. However if say a company wants to create one design which will be produced on a massive scale, the Δ gain in packing efficiency will improve the cost of manufacturing by a lot, in which case multi iteration is a better solution.

Since for our assignment purposes the final runtime wasn't affected much as and it was fast enough (2 to 3 seconds), from now on we use multi iteration logic for analysing all our test cases.

4 Time Complexity & Packing Efficiency Analysis

4.1 Naive Row Packing Algorithm

The time complexity of this naive algorithm standalone is $O(n)$ (where n is the number of gates) since it only involves finding the maximum height of a rectangle and packing them side-by-side in a single iteration over all rectangles, according to the widths (as visible in the Listing 1 below)

However our IO-Parser also sorts the rec_data as mentioned here while preparing it to pass to a packing algorithm. Hence the overall implementation has a time complexity of $O(n \log n)$. Since sorting doesn't provide any algorithmic benefit, the sorting can be suppressed by an argument if this algorithm has to be used.

```
1  def Pack_by_Pixel_v0(rec_data,h_max):
2      x,y = 0,0
3      cells_packed, max_rows_used, max_cols_used = 0, h_max, 0
4      rdata = deepcopy(rec_data)
5
6      for i in range(len(rdata)):
7          rdata[i].set_pos(x,y)
8          rdata[i].packed()
9          x += rdata[i].width
10         cells_packed += (rdata[i].width)*(rdata[i].height)
11
12     max_cols_used = x
13
14     return rdata,[cells_packed,max_rows_used,max_cols_used],True
```

Listing 1: Naive Row Algorithm's Python Implementation

4.2 Pixel Scan Algorithm

The time complexity of this heuristic algorithm is not easy to calculate systematically since the grid will change dynamically with rectangle placement. However we can claim a loose upper bound to be $O(n^2)$ by the following logic :

1. Suppose the outer loops (Listing 2 - Line 9 & 13) run over the entire grid for every rectangle object (Listing 2 - Line 7). Hence it has to run over A_0 pixels (an upper bound), where $A_0 = W_0 \cdot H_0$, is the area of initial bounding box guess.

2. Suppose for every possible empty pixel it has to scan A_r pixels , A_r being the area of the rectangle being packed.If it finds that all A_r pixels are empty then runs on them again,marking them as occupied. Hence running over $2 \cdot A_r$ pixels for each rectangle.

3. Therefore, a worst case upper bound of pixels which we need to iterate is given by :

$$\sum_{r=1}^{n} O(2 \cdot A_o \cdot A_r) \leq O(A_o \cdot n \cdot A_{max}) \approx O(A_{Tot} \cdot n \cdot 10^4) \approx O\left(n \cdot \frac{10^4}{4} \cdot n\right) = O(n^2)$$

Here A_o is taken approximately to be A_{Tot} (The total area of gates that have to be placed), which is further approximated as $\frac{A_{max}}{4}$ (Since our dimensions are drawn independently from a normal distribution with mean at half of maximum dimension). $A_{max} = 10^4$ by the constraints of problem statement.

```
def Pack_by_Pixel_v1(rec_data,Im_Width,Im_Height):

    rdata = deepcopy(rec_data)
    cells_packed, max_rows_used, max_cols_used = 0, 0, 0
    Im_Data = [[0]*Im_Width for r in range(Im_Height)]

    for i in range(len(rdata)):
        rec_done = False
        for y in range(Im_Height):
            if(y+rdata[i].height > Im_Height or rec_done):
                break
            else:
                for x in range(Im_Width):
                    if(x+rdata[i].width > Im_Width):
                        break
                    # [Internal Loops ...]
        if(not rec_done):
            return -1,None,None

    return rdata,[cells_packed,max_rows_used,max_cols_used],True
```

Listing 2: Outer loops of Pixel Scan Implementation

```
# [Internal Loops ...]
if(Im_Data[y][x] == 0 and Im_Data[y+rdata[i].height-1][x+rdata[i].
    width-1] == 0):
    isvalid = True
    for r in range(y,y+rdata[i].height):
        if(isvalid):
            for c in range(x,x+rdata[i].width):
                if(Im_Data[r][c] == 1):
                    isvalid = False
                    break
        else:
            break
    if(isvalid):
        # [Successful Packing Loops]
        # [Marks the grid location as occupied]
        # [Updates the attributes of Rec Object]
        rec_done = True
        break
```

Listing 3: Internal Loops of Pixel Scan Implementation

4.3 Predictive Pixel Scan

The time complexity of this heuristic algorithm is even harder to calculate systematically since now the pixel finding loop (Listing 4 - Line 10) has iterations dependent on the widths of already packed gates. However the bound can still proven to be $O(n^2)$ by the following logic :

1. For the worst case scenario, the only difference between this and the previous algorithm is that for a given row, instead of scanning it entirely it only scans at gaps of the already filled rectangle's widths.

2. Hence instead of Scanning W_0 pixels per row , It will on average scan $\approx \frac{W_0}{\varphi}$ pixels per row, φ being the average width of gates placed in a row.

3. φ can be approximated since we draw our samples from a normal distribution with mean at half of maximum dimension. Hence we can take $\varphi \approx 50$.

4. This has an interesting implication, since instead of going over A_0 pixels for each rectangle, we only run over $\frac{A_0}{\varphi}$ pixels. Since φ doesn't depend on order of n, hence we can further use the logic of Pixel Scan to calculate time complexity as $O(n^2)$ (Ignoring φ as a constant during the big-O calculation.)

5. Despite having same $O(n^2)$ time complexity, this performs much better than previous algorithm which is explained later.

```
1  def Pack_by_Pixel_v2(rec_data,Im_Width,Im_Height):
2      rdata = deepcopy(rec_data)
3      cells_packed,max_rows_used,max_cols_used = 0,0,0
4      Im_Data = [[0]*Im_Width for r in range(Im_Height)]
5      Im_Rec_Data = {rdata[i].index : rdata[i] for i in range(len(
     rdata))}
6
7      for i in range(len(rdata)):
8          rec_done = False
9          x,y = 0,0
10         for _inf_it in count(0,1):
11             if (y + rdata[i].height > Im_Height or rec_done):
12                 break
13             else:
14                 if(x + rdata[i].width > Im_Width):
15                     x,y = 0,y+1
16                     continue
17                 else:
18                     # [Internal Loops ...]
19
20         if(not rec_done):
21             return -1,None,None
22
23     return rdata,[cells_packed,max_rows_used,max_cols_used],True
```

Listing 4: Outer loops of Predictive Pixel Scan Implementation

```
# [Internal Loops ...]
if(Im_Data[y][x] == 0):
    if(Im_Data[y+rdata[i].height-1][x+rdata[i].width-1] == 0):
        isvalid = True
        for r in range(y,y+rdata[i].height):
            if(isvalid):
                for c in range(x,x+rdata[i].width):
                    if(Im_Data[r][c] != 0):
                        isvalid = False
                        row_notvalid,col_notvalid = r,c
                        break
            else:
                break
        if(isvalid):
            # [Successful Packing Loops]
            # [Marks the grid location as occupied]
            # [Updates the attributes of Rec Object]
            rec_done = True
            break
        else:
            w_enc = # Width of Rectangle stored at invalid position
            x_enc = # X of Rectangle stored at invalid position
            x = x_enc + w_enc
            continue
    else:
        x = x + rdata[i].width
        continue
else:
    w_enc = # Width of Rectangle stored at (x,y) [Line 2]
    x_enc = # X of Rectangle stored at (x,y) [Line 2]
    x = x_enc + w_enc
    continue
```

Listing 5: Inner loops of Predictive Pixel Scan Implementation

Here we must mention that the order of n in our problems is at max 1000, which isn't large enough for good asymptotic analysis. Thereby the constants of implementation matter and φ is the reason that our second algorithm is **practically much faster** than the previous implementation.

Additionally φ need not be a constant but the only important aspect (for theoretical calculations) is that it is independent of n. Practically we can see the improvement by plotting Run Time as we increase the number of gates (n) as shown below. (It is important to note that runtime of cases less than 200 seems to be higher than cases above 200 in the graph. This implementation decision is explained later)

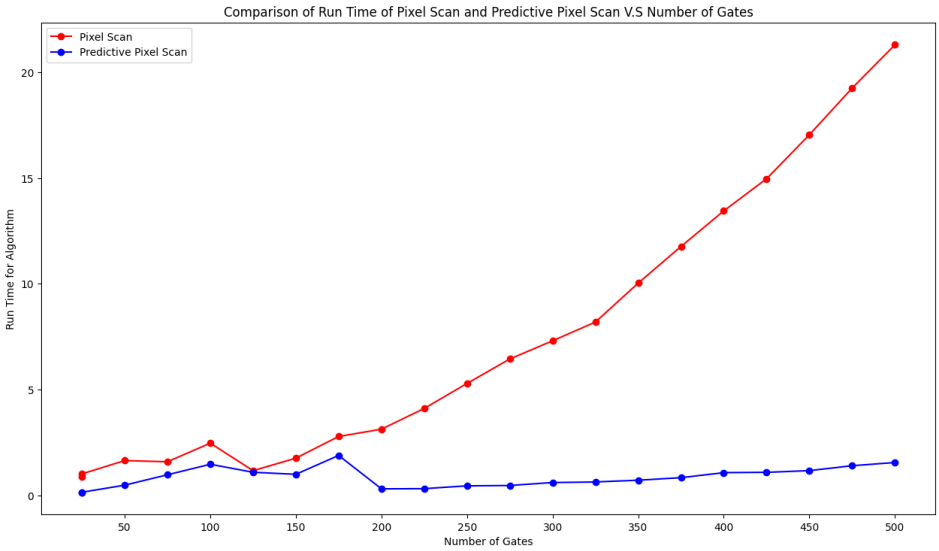


Figure 6: Run Time Comparison of 2 Algorithm Implementations

4.4 Plot of t vs n & η vs n for Test Cases

The following table shows the runtime and packing efficiency for the test cases given on Piazza. Our algorithm performs sufficiently fast and gives good packing efficiency for these test cases (Note that unit of time used is seconds throughout the graphs and our discussions).

Test Case	Number of Gates (n)	Runtime in Seconds (t)	Packing Efficiency (η)
1	3	0.005484	0.818181
2	3	0.009657	0.933333
3	10	0.008263	0.902778
4	5	0.004607	0.845238
5	35	0.004479	0.952381

Table 1: t and η on Sample cases (provided on Piazza)

To stress test our code, we automated creation of Test Cases using two-three methods. Firstly we ran our algorithm on test cases where the gate frequency ranges from 5 to 1000 (at a difference of 5 gates) to check how the packing efficiency as well as run time varies with increasing the number of gates (n) :

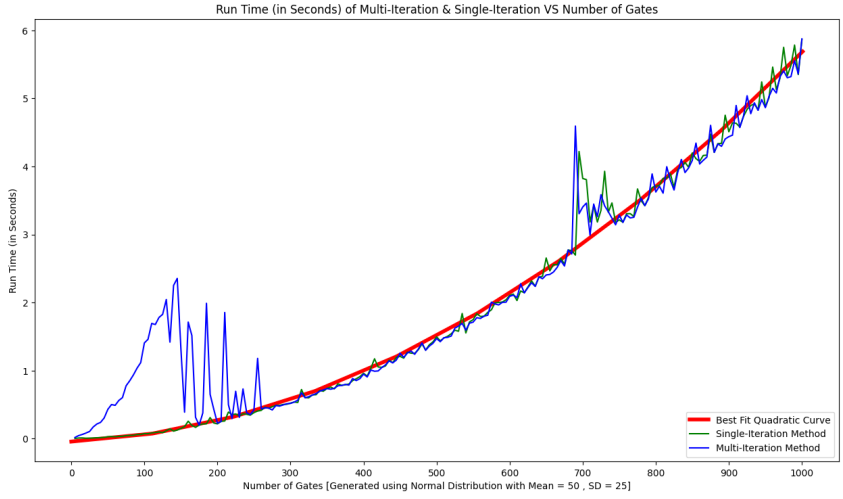


Figure 7: Run Time vs Number of Gates

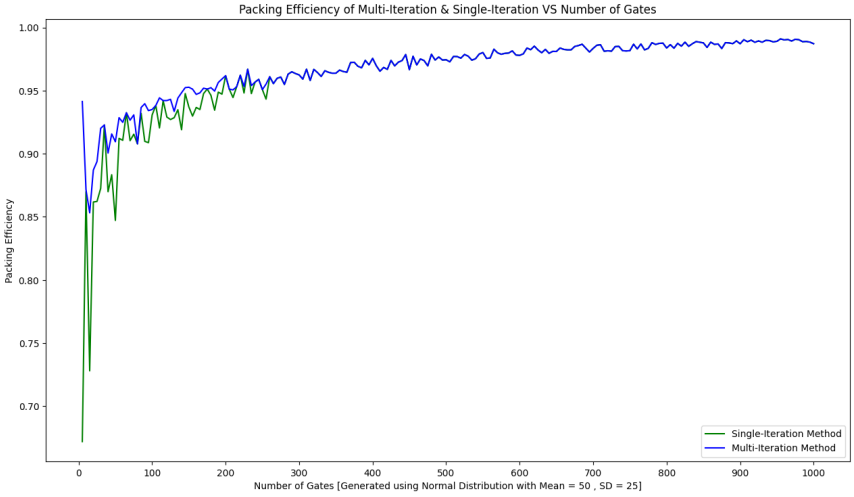


Figure 8: Packing Efficiency vs Number of Gates

The multi iteration method only runs 20 or so more iterations of the same algorithm (in nearby widths) in the hope of finding a better packing.The observations from the above graphs are:

4.4.1 Runtime :

1. The runtime for both single and multi iteration is approximately quadratic in nature with respect to number of gates. This is in accordance to our theoretical prediction and the best fit quadratic curve has following expression

(found using NumPy polyfit on the data, the red part of Fig 7) :

$$T(n) = 5.248 \cdot 10^{-6} \cdot n^2 + 4.773 \cdot 10^{-4} \cdot n - 0.04536$$

Where $T(n)$ is the runtime on n gates Test Case (in seconds)

2. There is deviation in multi and single iteration for less than 300 gates which is as expected as per the discussion in Section 3.2. For higher gate frequency they both tend to show same behaviour.

3. A peak is observed near $n = 700$ - 750 but in magnitude terms, the deviation from expected behaviour is only of 1-2 s. This can be accounted for randomness of the test case generation and the fact that for the given plot, we only generated single case for each gate frequency.

4.4.2 Packing Efficiency:

1. The packing efficiency is almost same(> 0.95) for multi and single iteration for more than 300 gates due to the explanation of Section 3.2.

2. For less than 300 gates the packing efficiency of multiple iteration is greater than single iteration. This is because of the logic of multiple iteration implementation discussed in Subsection 2.4.3.

4.5 Average Runtime and Packing Efficiency for Multiple Testcases

SNo.	Number of Gates (n)	Average Runtime in Seconds (t)	Packing Efficiency (η)
1	25	0.109190	0.898202
2	100	1.802440	0.939640
3	250	0.560971	0.956785
4	1000	7.327116	0.989691

Table 2: Average t & η for 100 cases for different n : $\mu = 50$, $\sigma = 25$

SNo.	Number of Gates (n)	Average Runtime in Seconds (t)	Packing Efficiency (η)
1	25	0.111734	0.880246
2	100	1.321254	0.927841
3	250	0.984085	0.956785
4	1000	4.771615	0.977463

Table 3: Average t & η for 100 cases for different n : $\mu = 50$, $\sigma = 10$

In the above test cases, gate dimensions have been generated using normal distribution. Since our algorithm revisits grid and packs shorter and less leaner rectangles in available spaces, when the σ is low, such rectangles generate with lesser probability hence the slightly worse packing efficiency.

5 Visualising Output on Multiple Test Cases

5.1 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** $= 10$

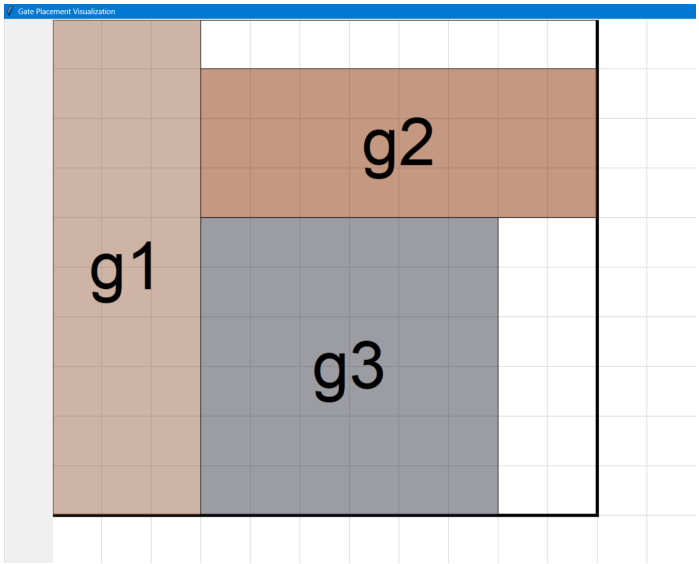


Figure 9: Gate Packing

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

Figure 10: Report

5.2 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** $= 6$

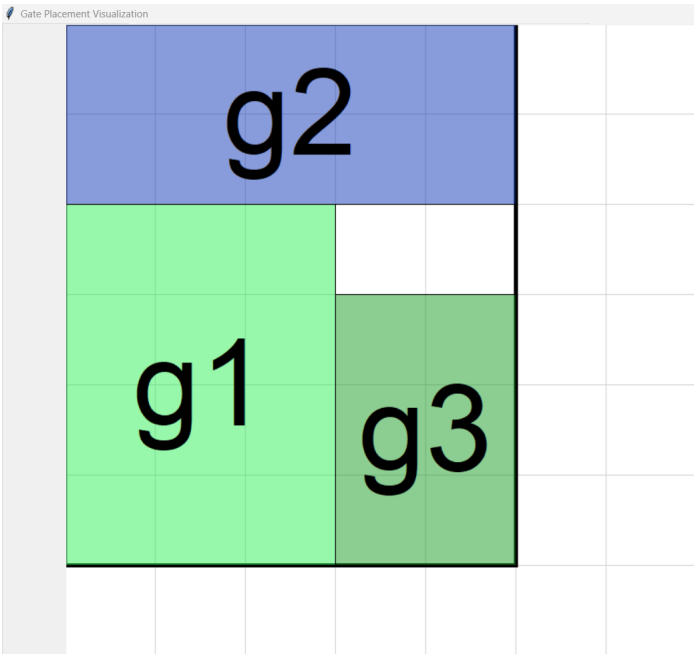


Figure 11: Gate Packing

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

Figure 12: Report

5.3 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** $= 30$

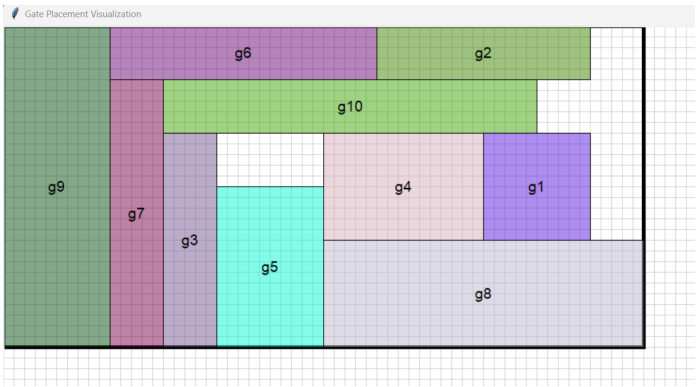


Figure 13: Gate Packing

```
Gate Freq : 10
Runtime : 0.00826335
Packing Efficiency :  0.90277778
```

Figure 14: Report

5.4 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** $= 6$

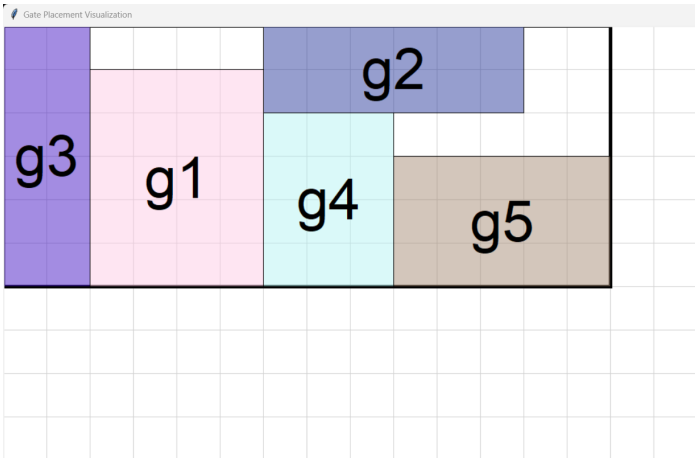


Figure 15: Gate Packing

```
1  Gate Freq : 5
2  Runtime : 0.00460672
3  Packing Efficiency :  0.84523810
```

Figure 16: Report

5.5 Test Case 5

Bounding Box Dimension : Width = 25 , Height = 21

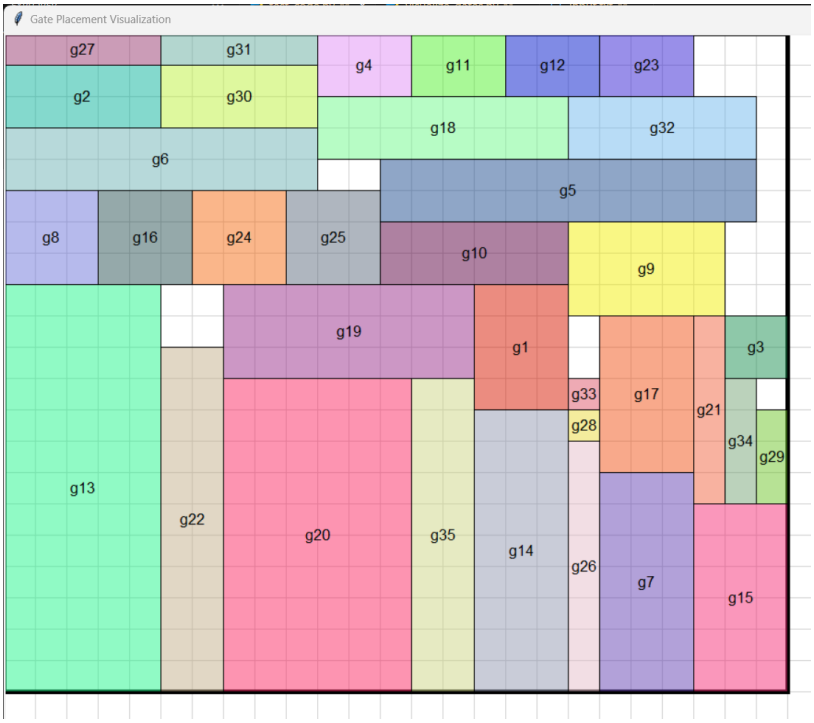


Figure 17: Gate Packing

```
1  Gate Freq : 35
2  Runtime : 0.00447941
3  Packing Efficiency :  0.95238095
```

Figure 18: Report