COL-215
Software Assignment - 1 Report

Shreeraj Jambhale 2023CS50048
Pallav Kamad 2023CS51067

Table of Contents

1 Introduction

1.1 Aim

The goal of this code is to efficiently pack a set of rectangles/gates into a bounding box of variable width such that the total area used is maximized while minimizing the bounding box area. The output provides the dimensions of the bounding box and the placement of each rectangle within it. Here rectangles signify packing of gates on a logic board and direct application of 2D-Packing

1.2 Constraints

- *0 < Number of gates <= 1000*
- *0 < Width of gate <= 100*
- *0 < Height of gate <= 100*

2 Approach

The algorithm uses a skyline-based packing approach, where rectangles are placed in a way that minimizes wasted space above and around them. The skyline is updated dynamically as rectangles are placed, optimizing for the smallest possible bounding box height

2.1 Function Explanations

2.1.1 read_input(file)

```
def read_input(file):
    max_width = 0
    total_area = 0

    with open(file, 'r') as f:
        data = f.readlines()
        rectangles = []
        for line in data:
            parts = line.split()
            identifier, width, height = parts[0], int(parts[1]), int(parts[2])
            rectangles.append([identifier, width, height])
            max_width = max(max_width, width)
            total_area += width * height

    return rectangles, max_width, total_area
```

- **Purpose**: Reads input from a file containing rectangle names and dimensions. It calculates the maximum width and sum of area of the rectangles.

- **Process**:
  - Opens the file and reads each line.
  - Splits each line into the rectangle name, width, and height.
  - Updates max_width if the current rectangle's width is larger.
  - Adds the area of the current rectangle to total_area.
- **Returns**: A list of rectangles along with the max_width and total_area.

2.1.2 sort_rectangles(rectangles)

```
def sort_rectangles(rectangles):
    return sorted(rectangles, key=lambda x: (x[1], x[2]), reverse=True)
```

- **Purpose**: Sorts the list of rectangles in descending order of width. If tie then rectangles sorted on basis of decreasing height.
- **Process**:
  - Uses the sorted function sort rectangles by width and then by height.
- **Returns**: A sorted list of rectangles.

2.1.3 update_skyline(current_skyline, point, rect_height, rect_width)

```
def update_skyline(current_skyline, point, rect_height, rect_width):
    if point[0] == 0:
        for i in range(point[1], point[1] + rect_width):
            current_skyline[i] = point[2] + rect_height
    else:
        for i in range(point[1] - rect_width, point[1]):
            current_skyline[i] = point[2] + rect_height

    return current_skyline
```

- **Purpose**: Updates the skyline after a rectangle is placed.
- **Process**:
  - Depending on whether the rectangle is placed to the right or left of a valid point, it updates the skyline to reflect the new height where the rectangle has been placed.
- **Returns**: The updated skyline.

2.1.4 optimize_skyline(current_skyline, rect_width, valid_points_list)

```
def optimize_skyline(current_skyline, rect_width, valid_points_list):
    for i in range(1, len(valid_points_list) - 2):
        if valid_points_list[i][0] == 0 and valid_points_list[i + 1][0] == 1:
            local_min_length = valid_points_list[i + 1][1] - valid_points_list[i][1]
            next_max_height = min(current_skyline[valid_points_list[i][1] - 1],
                                  current_skyline[valid_points_list[i + 1][1]])
            if local_min_length < rect_width:
                for j in range(valid_points_list[i][1], valid_points_list[i + 1][1]):
                    current_skyline[j] = next_max_height

    return current_skyline
```

- **Purpose**: Optimizes the skyline by leveling out local minima, helping to reduce wasted space. A local minima is that segment of skyline whose Y-coordinate is less than Y-coordinates of both the adjacent (left and right) skyline segments.
- **Process**:
  - Identifies segments of the skyline where a local minimum by comparing its Y-coordiante to its adjacent segment's Y-coordinates. Then this segment is raised to match the minimum of adjacent heights, reducing the gap that would otherwise be too small for any other rectangle.
- **Returns**: The optimized skyline.

2.1.5 find_valid_points(skyline, box_width)

```
def find_valid_points(skyline, box_width):
    points = [[0, 0, skyline[0]]]

    for i in range(box_width - 1):
        if skyline[i] > skyline[i + 1]:
            points.append([0, i + 1, skyline[i + 1]])
        elif skyline[i] < skyline[i + 1]:
            points.append([1, i + 1, skyline[i]])

    points.append([1, box_width, skyline[box_width - 1]])

    return points
```

- **Purpose**: Identifies points on the skyline where rectangles can potentially be placed.
- **Process**:
  - The function iterates over the skyline The list points stores list of bool value, index of box, height of skyline at that index. A point is valid if the height of the current segment is less than height of previous segment (and is marked as 0) or if its height is more than the next adjacent segment (in this case point is marked as 1).
- **Returns**: A list of valid points with their marked value(Boolean), index and heights.

2.1.6 is_valid_placement(rect_height, rect_width, point, skyline)

```
def is_valid_placement(rect_height, rect_width, point, skyline):
    if point[0] == 0:
        try:
            for i in range(point[1], point[1] + rect_width):
                if skyline[i] <= point[2]:
                    continue
                else:
                    return False
            return True
        except:
            return False
    else:
        try:
            for i in range(point[1] - rect_width, point[1]):
                if skyline[i] <= point[2]:
                    continue
                else:
                    return False
            return True
        except:
            return False
```

- **Purpose**: Checks if a rectangle can be placed at a given point on the skyline without overlapping existing rectangles.
- **Process**:
  - Depending on whether the rectangle is placed to the right or left of the point, checks if there is enough space in the skyline for the rectangle's width.
- **Returns**: True if the rectangle can be placed, False otherwise.

2.1.7 pack_rectangles(rectangle_sequence, box_width, total_area)

```
def pack_rectangles(rectangle_sequence, box_width, total_area):
    output = []
    skyline = [0] * box_width

    for rectangle in rectangle_sequence:
        valid_points_list = find_valid_points(skyline, box_width)
        should_optimize = True

        for i in range(len(valid_points_list) - 1):
            gap = valid_points_list[i][1] - valid_points_list[i + 1][1]
            if gap >= rectangle[1]:
                should_optimize = False
                break

        if should_optimize:
            skyline = optimize_skyline(skyline, rectangle[1], valid_points_list)

        valid_points_list = find_valid_points(skyline, box_width)
        sorted_valid_points = sorted(valid_points_list, key=lambda x: (x[2], x[1]))

        for point in sorted_valid_points:
            if is_valid_placement(rectangle[2], rectangle[1], point, skyline):
                if point[0] == 0:
                    output.append([rectangle[0], point[1], point[2]])
                    update_skyline(skyline, point, rectangle[2], rectangle[1])
                else:
                    output.append([rectangle[0], point[1] - rectangle[1], point[2]])
                    update_skyline(skyline, point, rectangle[2], rectangle[1])
                break

    height = max(skyline)
    efficiency = total_area / (height * box_width)

    return output, efficiency, height
```

- **Purpose**: Attempts to pack all rectangles into a bounding box of given width, optimizing for minimum height and maximum area usage.
- **Process**:
  - Initializes the skyline to a flat starting point that is 0. Skyline is a list containing heights of the highest rectangle edge at each index starting from 0 till box width –1.
  - Iterates over each rectangle, finding valid points on the skyline and checking if the rectangle can be placed.
  - Updates the skyline and records the placement of each rectangle.
  - Optimizes the skyline when necessary.
  - Calculates the efficiency of the packing (total_area divided by the bounding box area). This does not imply that minimal rectangular box will have the same area as toatal_area. Thus Actual packing efficiency is higher than calculated.
- **Returns**: The list of rectangle placements, the packing efficiency, and the final height of the bounding box.

2.1.8 Main Execution

```
best_efficiency = 0
best_index = 0
start_time = time.perf_counter()

input_rectangles, max_box_width, total_area = read_input("input.txt")

for width in range(max_box_width, max_box_width * 5):
    print("-",end='')
    placement_list, efficiency, final_height  = (
pack_rectangles(sort_rectangles(input_rectangles), width, total_area))
    if best_efficiency < efficiency:
        best_index = width
        best_efficiency = efficiency
print()
print("time(s) - ",time.perf_counter() - start_time)
print("best_efficiency - ",best_efficiency)
```

- **Purpose**: Finds the optimal bounding box width that minimizes height while maximizing efficiency.
- **Process**:
  - Reads the input and sorts the rectangles.
  - Iteratively tries different bounding box widths, recording the one that yields the best efficiency.
  - Outputs the final placement and bounding box dimensions to the output file.

2.1.9 Sorting the Placement List and Generating Output.txt

```
placement_list, efficiency, final_height =
pack_rectangles(sort_rectangles(read_input("input.txt")[0]), best_index,total_area)
placement_list = sorted(placement_list, key=lambda x: int(x[0][1:]))

with open("output.txt", "w") as file:

    file.write(f"bounding_box {best_index} {final_height}\n")
```

```
    for rect in placement_list:
        file.write(f"{rect[0]} {rect[1]} {rect[2]}\n")
```

- **Purpose**: Ensures the placement list is ordered by rectangle's name (e.g., g1, g2, g3, ...).
- **Process**:
  - Sorts the placement list by the numeric part of each rectangle's identifier.
  - Generate output.txt file.
- **Returns**: A sorted placement list that is then written to the output file.

3 Time Complexity Analysis

The skyline update operations are dependent on the box width w, the time complexity of the skyline-related functions should be adjusted accordingly. Here's the time complexity analysis:

1. **Reading Input**: The read_input function has a time complexity of O(n), where n is the number of rectangles.
2. **Sorting Rectangles**: The sort_rectangles function sorts the rectangles by width and height, taking O(nlog(n)) time.
3. **Updating Skyline**: The update_skyline function modifies the skyline for a width w, resulting in a time complexity of O(w)
4. **Optimizing Skyline**: The optimize_skyline function also works over a width w, giving it a time complexity of O(w).
5. **Finding Valid Points**: The find_valid_points function iterates through the skyline, leading to a time complexity of O(w)
6. **Validating Placement**: The 'is_valid_placement' function checks the skyline over the reactangles width, which results in O(w) complexity.
7. **Packing Rectangles**: The pack_rectangles function calls these skyline functions for each rectangle, leading to an overall time complexity of $O(n \times w \times n\log(n))$ because sorting is O(nlog(n)) and each rectangle placement involves skyline operations with complexity O(w)
8. **Main Loop**: The main loop iterates over possible box widths, which is bounded by w=100. This results in a final overall time complexity of $O(w \times n^2 \log(n))$

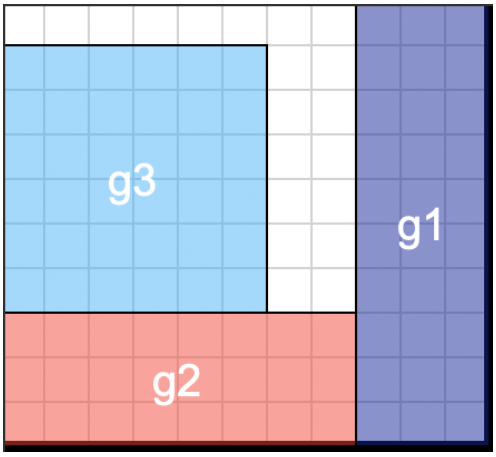
Given the constraint that w is 100, the final complexity simplifies to $O(w \times n^2 \log(n)) = O(n^2 \log(n))$, where w is constant and does not scale with n.

4 Result and Analysis

4.1 Provided Test Cases

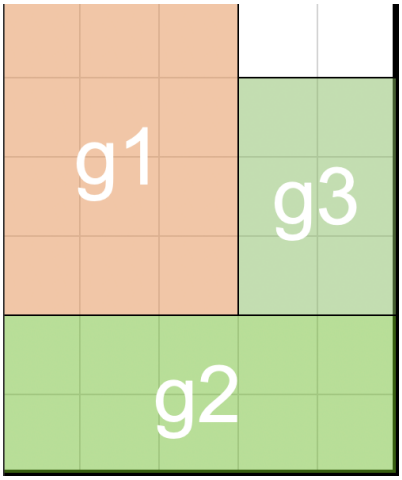
4.1.1 Test Case – 1

- Input –
  g1 3 10
  g2 8 3
  g3 6 6
- Code Output –
  bounding_box 11 10.
  g1 8 0
  g2 0 0
  g3 0 3
- Time(s) - 0.000749250000808388
- Best_efficiency of our code - 0.8181818181818182



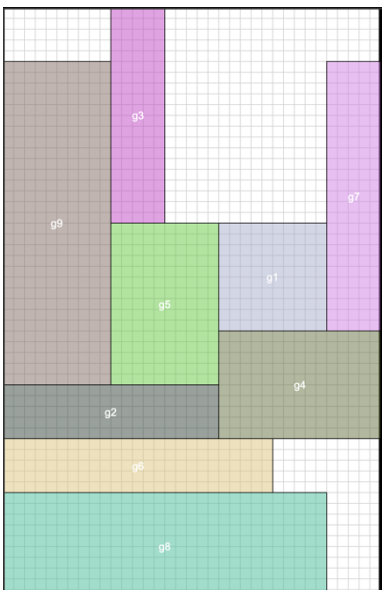
4.1.2 Test Case – 2

- Input –
  g1 3 4
  g2 5 2
  g3 2 3
- Code Output –
  bounding_box 5 6
  g1 0 2
  g2 0 0
  g3 3 2
- Time(s) - 0.0002788340061670169
- Best_efficiency of our code - 0.9333333333333333



4.1.3 Test Case – 3

- Input –
  g1 10 10
  g2 20 5
  g3 5 20
  g4 15 10
  g5 10 15
  g6 25 5
  g7 5 25

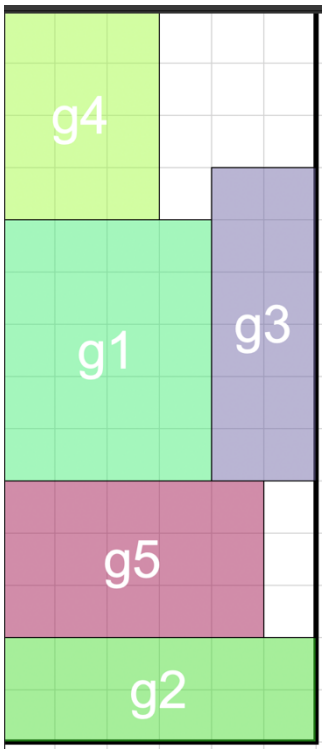


g8 30 10
g9 10 30
g10 35 5

- Code Output –

bounding_box 35 60
g1 20 30
g2 0 20
g3 10 40
g4 20 20
g5 10 25
g6 0 15
g7 30 30
g8 0 5
g9 0 25
g10 0 0

- Time(s) - 0.018845209007849917
- Best_efficiency of our code - 0.7738095238095238



4.1.4 Test Case – 4

- Input –

g1 4 5
g2 6 2
g3 2 6
g4 3 4
g5 5 3

- Code Output –

bounding_box 6 14
g1 0 5
g2 0 0
g3 4 5
g4 0 10
g5 0 2

- Time(s) - 0.00048658400191925466
- Best_efficiency of our code - 0.8452380952380952

4.1.5 Test Case – 5

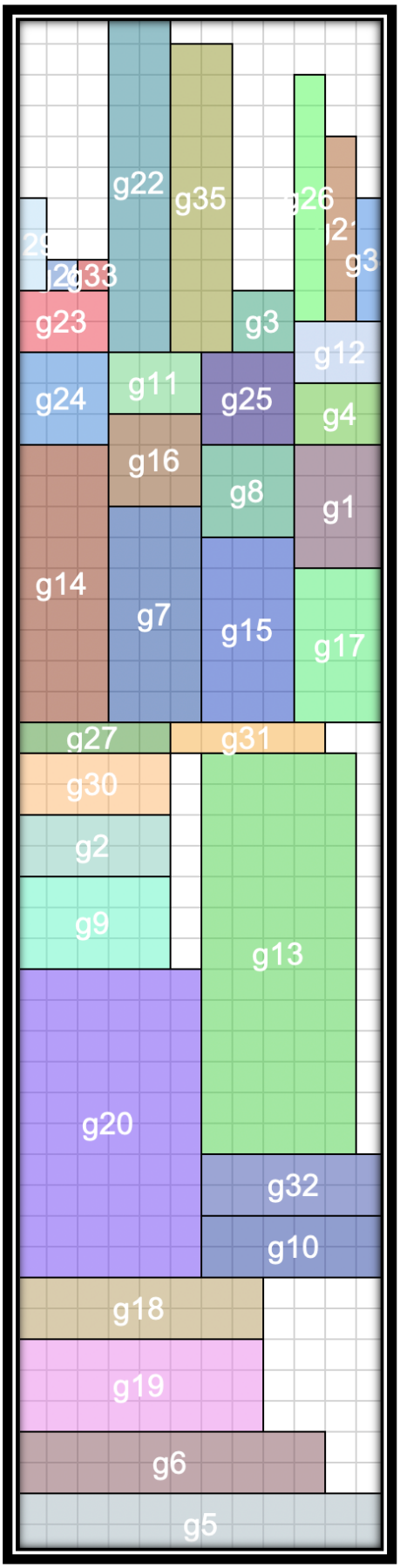
- Input –

g1 3 4
g2 5 2
g3 2 2
g4 3 2
g5 12 2
g6 10 2
g7 3 7
g8 3 3
g9 5 3
g10 6 2
g11 3 2
g12 3 2

g13 5 13
g14 3 9
g15 3 6
g16 3 3
g17 3 5
g18 8 2
g19 8 3
g20 6 10
g21 1 6
g22 2 11
g23 3 2
g24 3 3
g25 3 3
g26 1 8
g27 5 1
g28 1 1
g29 1 3
g30 5 2
g31 5 1
g32 6 2
g33 1 1
g34 1 4
g35 2 10

- Code Output –

bounding_box 12 50
g1 9 32
g2 0 22
g3 7 39
g4 9 36
g5 0 0
g6 0 2
g7 3 27
g8 6 33
g9 0 19
g10 6 9
g11 3 37
g12 9 38
g13 6 13
g14 0 27
g15 6 27
g16 3 34
g17 9 27
g18 0 7
g19 0 4
g20 0 9
g21 10 40
g22 3 39
g23 0 39
g24 0 36
g25 6 36
g26 9 40



g27 0 26
g28 1 41
g29 0 41
g30 0 24
g31 5 26
g32 6 11
g33 2 41
g34 11 40
g35 5 39

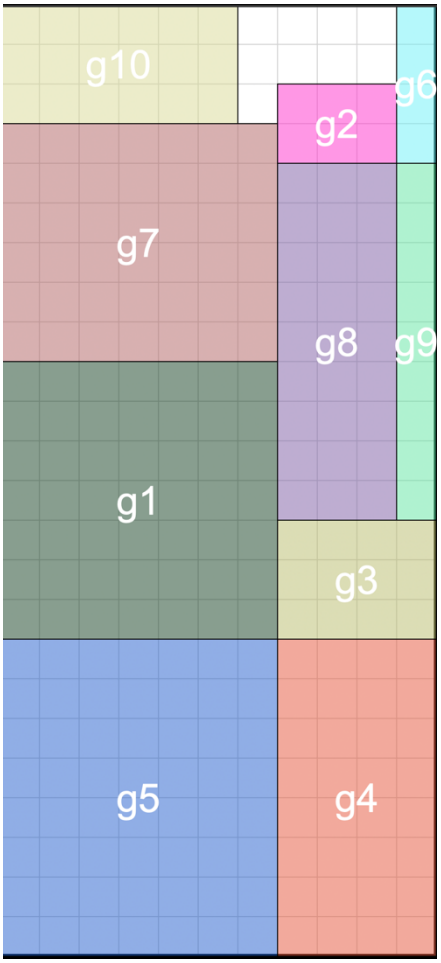
- Time(s) - 0.009657750008045696
- Best_efficiency of our code - 0.8333333333333334

4.2 Self Test Cases

4.2.1 Test Case – 1

- Input –

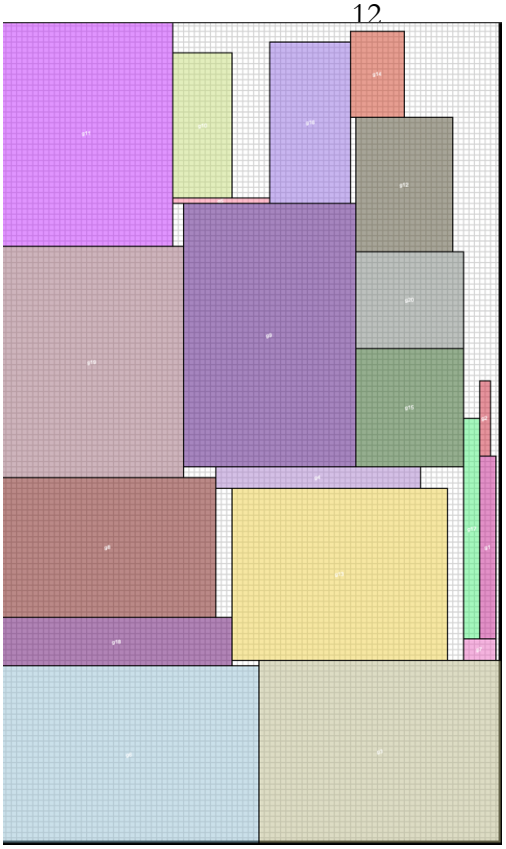
g1 7 7
g2 3 2
g3 4 3
g4 4 8
g5 7 8
g6 1 4
g7 7 6
g8 3 9
g9 1 9
g10 6 3



- Code Output –

bounding_box 11 24
g1 0 8
g2 7 20
g3 7 8
g4 7 0
g5 0 0
g6 10 20
g7 0 15
g8 7 11
g9 10 11
g10 0 21

- Time(s) - 0.001882874988950789
- Best_efficiency of our code - 0.9659090909090909



4.2.2 Test Case – 2

20 Gates

- Input – Input2.txt
- Code Output – Output2.txt
- Time(s) - 0.05274341600306798
- Best_efficiency of our code - 0.8995010190456111

4.2.3 Test Case – 3

60 Gates

- Input – input3.txt
- Code Output – output3.txt
- Time(s) - 0.001882874988950789
- Best_efficiency of our code - 0.9659090909090909

4.2.4 Test Case – 4

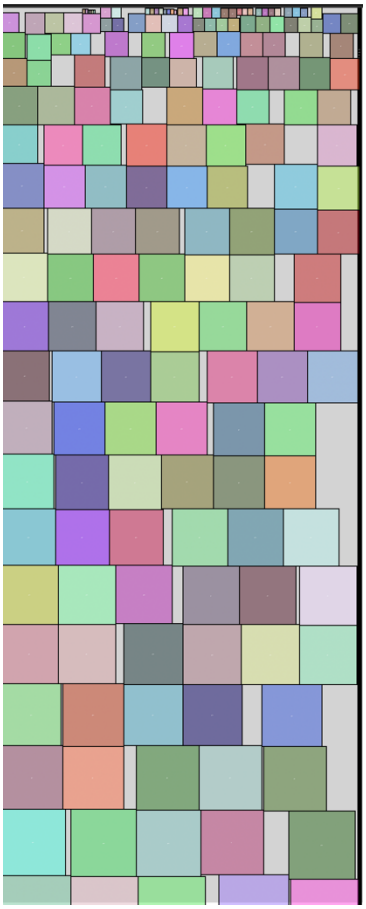
125 Gates

- Input – input4.txt
- Code Output – output4.txt
- Time(s) - 1.0456233750010142
- Best_efficiency of our code - 0.873861145827673

4.2.5 Test Case – 5

250 Gates

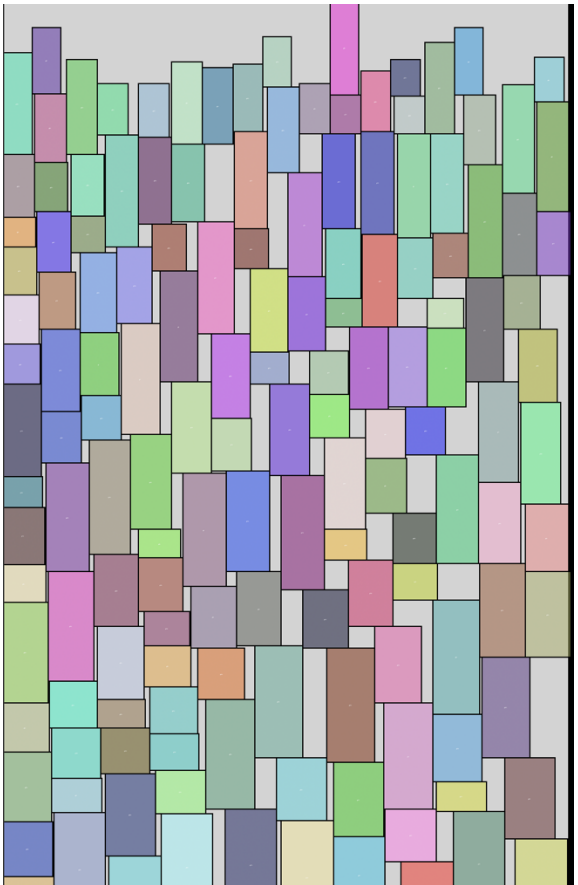
- Input – input5.txt
- Code Output – output5.txt
- Time(s) - 3.2227385000005597
- Best_efficiency of our code - 0.9135724548440066



4.2.6 Test Case – 6

500 Gates

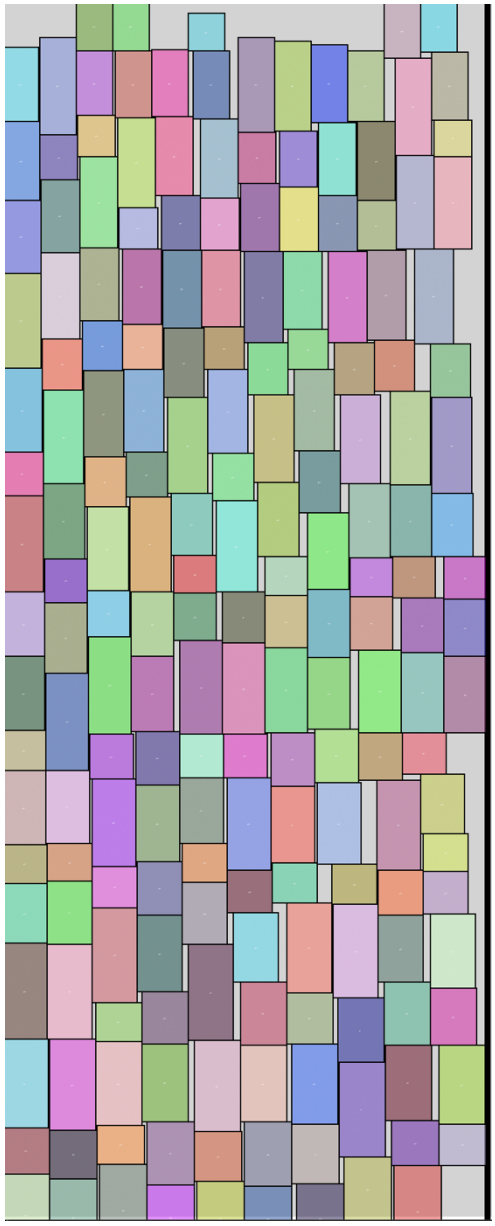
- Input – input6.txt
- Code Output – output6.txt
- Time(s) - 7.403824709006585
- Best_efficiency of our code 0.9059006598295702



4.2.7 Test Case – 7

1000 Gates

- Input – input7.txt
- Code Output – output7.txt
- Time(s) - 9.957367666997015
- Best_efficiency of our code 0.9245039872408294



5 Reference

https://www.david-colson.com/2020/03/10/exploring-rect-packing.html

https://www.researchgate.net/publication/221049934_A_Skyline-Based_Heuristic_for_the_2D_Rectangular_Strip_Packing_Problem

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

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

https://github.com/juj/RectangleBinPack/tree/master

https://www.csc.liv.ac.uk/~epa/surveyhtml.html#:~:text=In%20the%20two%2Ddimensional%20bin,the%20minimum%20number%20of%20bins.

https://www.jstor.org/stable/2582731?seq=1