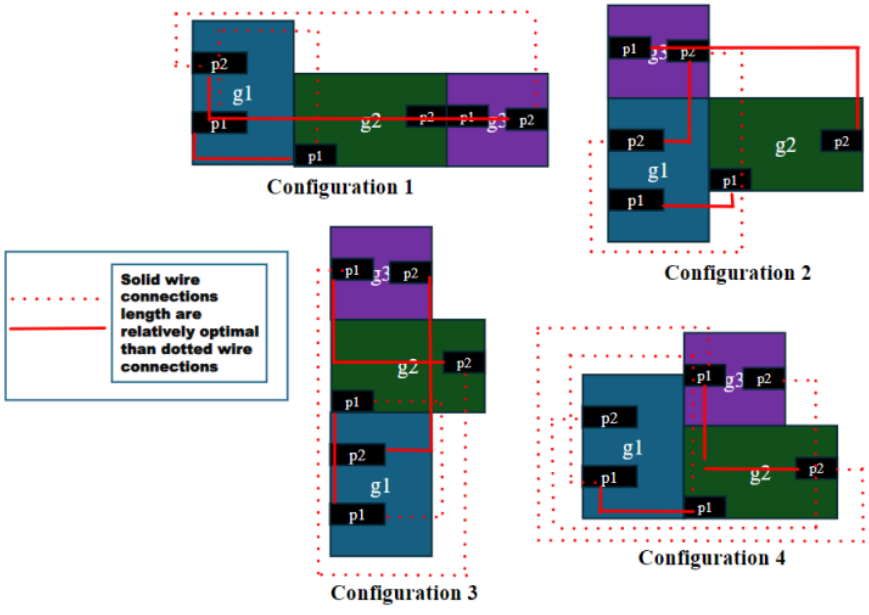
COL 215 Software Assignment 2: Wiring-aware Gate Positioning

Submission By: -

Aaditya Sharma(2023CS10420)

Namit Jain(2023CS10483)

October 5, 2024



Introduction :-

In this assignment, the problem of **Wiring-aware Gate Positioning** is addressed. The objective is to assign positions to rectangular logic gates on a 2D plane in such a way that the total wiring length between the connected pins is minimized, while also ensuring that no gates overlap.

Each gate has fixed dimensions and a set of pins located on its boundary. The wiring length is estimated using the **Manhattan distance** between the connected pins. This approach, known as the **semi-perimeter method**, forms a rectangular bounding box around the connected pins, with the wire length being half the perimeter of this rectangle.

To solve this problem efficiently, the **Simulated Annealing** optimization algorithm is employed. Simulated Annealing allows exploration of different configurations, accepting worse solutions with a probability that decreases over time, helping the algorithm escape local minima. The goal is to find the optimal gate configuration that minimizes the total wire length.

Design Decisions :-

Several key design decisions were made in the development of the solution:

1. **Data Structures:**
    - **Pins and Gates:** The pins dictionary stores instances of the Pin class for each pin, with the key being the pin's name (e.g., "g1.p1"), allowing direct access to the properties of individual pins.
    - Similarly, the gates dictionary stores instances of the Gate class, with the key being the gate's name (e.g., "g1"), allowing easy access to gate dimensions, positions, and associated pins. This structure allows quick and efficient retrieval of both gate and pin properties using their respective names, such as gates["g1"] for accessing the Gate object for gate g1 or pins["g1.p1"] for accessing the pin p1 on gate g1.

2. **Initial Placement of Gates on a Grid:**

- Gates are initially placed in a grid-like structure to ensure that no two gates overlap. The grid size is calculated based on the maximum width and height of the gates, as well as the number of gates. The gates are filled in the grid using a row-by-row scheme starting from the bottom left corner.
- This strategy ensures that all gates are placed with sufficient space between them, avoiding initial overlap.

3. **Simulated Annealing for Optimization:**
    - **Initial Temperature:** The starting temperature controls how likely the algorithm is to accept worse solutions. A higher temperature allows for more exploration, while a lower temperature restricts exploration.
    - **Temperature Decay:** The temperature is decreased after each iteration using an exponential decay factor, reducing the likelihood of accepting worse solutions as the algorithm progresses.
    - **Acceptance Probability:** Worse solutions are accepted with a probability based on the difference in wire length between the current and new configuration. This probability decreases as the temperature drops.
4. **Random Swaps in Simulated Annealing:**

- During each iteration of the Simulated Annealing algorithm, two gates are randomly selected, and their positions are swapped. This randomization helps the algorithm explore different configurations and move toward an optimal solution. The swap occurs both for the x and y positions of the two gates.
- The new configuration is evaluated, and if it results in a lower wire length, it is accepted. If not, it might still be accepted based on a probability that decreases with the temperature, allowing the system to escape local minima.

5. **Bounding Box Calculation:**
    - The wire length between connected pins is calculated using the **semi-perimeter method**, where the bounding box is formed around the connected pins, and the wire length is half the perimeter of the box.
6. **Gate Placement and Compression:**
    - Gates are initially placed in a grid-like manner to ensure no overlap. Afterward, the gates are compressed, where possible, to minimize empty space without causing overlap.

Code Snippets :-

1. Gate placement on a grid :

```
def place_gates(gates_list, max_height, max_width, dim_grid):
    ct = 0
    for i in gates_list:
        gates[i].x = max_width*(ct%dim_grid)
        gates[i].y = max_height*(ct//dim_grid)
        ct+=1
```

The first step is to place the gates on a grid-like structure to avoid initial overlaps. This function ensures that each gate has a starting position, which will later be optimized.

Explanation:

- **Row-major Grid Placement:** Gates are placed in a row-major order on a grid. This ensures that all gates are spaced out initially, with sufficient space to avoid overlaps starting from bottom left corner.
- **Grid Dimensions:** The grid's width and height are determined by the maximum width and height of the gates, and the total number of gates.

- The width of each cell in grid is maximum of all widths and height of each cell is maximum of all heights so that each cell is capable of holding every gate. Later the spaces will be compressed for further optimization.

2. Random Gate Swap :-

```
def swap_gates(order):
    i, j = random.sample(range(len(gates)), 2)
    new_order = order[:]
    gates[order[i]].x, gates[order[j]].x = gates[order[j]].x, gates[order[i]].x
    gates[order[i]].y, gates[order[j]].y = gates[order[j]].y, gates[order[i]].y
    new_order[i], new_order[j] = new_order[j], new_order[i]
    return new_order
```

Once the gates are placed, the optimization process begins with random swaps of gate positions. This is part of the exploration phase of the Simulated Annealing algorithm, where new configurations are generated.

Explanation:

- **Random Selection:** Two gates are randomly selected from the list of gates.
- **Swapping Positions:** The x and y coordinates of the two gates are swapped, resulting in a new configuration. This new configuration is evaluated for its wire length and potential improvements in the optimization process.

3. Bounding Box Calculation(Wire Length Estimation) :-

```
def estimate_total_wire_length():
    estimated_wire_length = 0
    for i in clusters:
        min_x = gates[clusters[i][0].gate_name].x + clusters[i][0].x
        min_y = gates[clusters[i][0].gate_name].x + clusters[i][0].x
        max_x = gates[clusters[i][0].gate_name].y + clusters[i][0].y
        max_y = gates[clusters[i][0].gate_name].y + clusters[i][0].y
        for j in clusters[i]:
            min_x = min(min_x, gates[j.gate_name].x + j.x)
            max_x = max(max_x, gates[j.gate_name].x + j.x)
            min_y = min(min_y, gates[j.gate_name].y + j.y)
            max_y = max(max_y, gates[j.gate_name].y + j.y)
        estimated_wire_length += max_x-min_x + max_y-min_y

    return estimated_wire_length
```

At each iteration, after gates have been swapped, the wire length between the connected pins must be recalculated. The **bounding box** method is used to estimate the wire length between connected pins.

Explanation:

- **Bounding Box Calculation:** For each cluster of connected pins, the minimum and maximum x and y coordinates are used to create a bounding box that encloses all pins in the cluster.
- **Wire Length Estimation:** The wire length is estimated by calculating the semi-perimeter of the bounding box. This is done for all clusters of connected pins, and the total wire length is accumulated.

4. Simulated Annealing Algorithm :-

```
def solve(max_iter, start_temperature, alpha, init_order, max_height, max_width, dim_grid):
    curr_temperature = start_temperature
    place_gates(init_order, max_height, max_width, dim_grid)
    curr_orientation = [estimate_total_wire_length(), init_order]
    curr_order = init_order[:]
    best_orientation = [curr_orientation[0], init_order] # O(1000)
    iteration = 0
    interval = (int)(max_iter / 10)
    while iteration < max_iter: # O(1000)
        new_order = swap_gates(curr_order) # O(1)
        new_orientation = estimate_total_wire_length() # O(40000)
        if new_orientation < curr_orientation[0]:  # better route so accept
            curr_orientation = [new_orientation, new_order] # O(1000)
            curr_order = new_order[:]
            if curr_orientation[0] < best_orientation[0]:
                best_orientation = [new_orientation, new_order] # O(1000)

        else:          # adjacent is worse
            accept_p = np.exp((curr_orientation[0] - new_orientation) / curr_temperature)
            p = random.random()
            if p < accept_p:  # accept anyway
                curr_orientation = [new_orientation, new_order] # O(1000)
                curr_order = new_order[:]
            # else don't accept
        if iteration % interval == 0:
            print("iter = %6d | curr error = %7.2f | temperature = %10.4f " % (iteration, best_orientation[0], curr_temperature))

        if curr_temperature < 0.00001:
            curr_temperature = 0.00001
        else:
            curr_temperature *= alpha
        iteration += 1

    place_gates(best_orientation[1], max_height, max_width, dim_grid) # O(1000)
    compress(best_orientation) # O(1000*1000*1000*100)
    best_orientation[0] = estimate_total_wire_length() # O(40000)

    return best_orientation
```

The **Simulated Annealing** algorithm optimizes the gate positions by swapping gates, calculating the new wire lengths, and deciding whether to accept the new configuration based on temperature and improvement. This process continues until the system converges to an optimal solution.

Explanation:

- **Initial Setup:** The gates are placed on the grid, and the initial wire length is calculated. The starting temperature and cooling schedule are set.
- **Swapping and Evaluation:** In each iteration, two gates are swapped using swap_gates(), and the total wire length is recalculated. If the new configuration has a lower wire length, it is accepted. If not, it is accepted with a probability that decreases as the temperature drops.

- **Cooling Schedule:** The temperature decreases exponentially at each iteration, which reduces the likelihood of accepting worse configurations as the algorithm progresses.
- **Final Placement and Compression:** Once the algorithm converges, the gates are placed using the best configuration, and the compression step is applied to reduce gaps between gates.

5. Compression Step :-

```
def compress(best_orientation):
    for i in best_orientation[1]:
        while gates[i].x>=0 and not check_for_overlaps(i):
            gates[i].x-=1
        gates[i].x+=1
    for i in best_orientation[1]:
        while gates[i].y>=0 and not check_for_overlaps(i):
            gates[i].y-=1
        gates[i].y+=1
```

After the Simulated Annealing process, a **compression step** is performed to reduce gaps between gates. This step helps compact the layout by shifting gates left and down as much as possible without causing overlaps.

Explanation:

- **Horizontal Compression:** Each gate is shifted to the left as far as possible, until an overlap is detected. The gate is then moved one unit back to prevent the overlap.
- **Vertical Compression:** Similarly, each gate is shifted down to remove unnecessary vertical gaps, again stopping when an overlap is detected.

Time Complexity Analysis :-

1.

```
def estimate_total_wire_length(): # O(pins)
    estimated_wire_length = 0
    for i in clusters:
        min_x = gates[clusters[i][0].gate_name].x + clusters[i][0].x
        min_y = gates[clusters[i][0].gate_name].x + clusters[i][0].x
        max_x = gates[clusters[i][0].gate_name].y + clusters[i][0].y
        max_y = gates[clusters[i][0].gate_name].y + clusters[i][0].y
        for j in clusters[i]:
            min_x = min(min_x, gates[j.gate_name].x + j.x)
            max_x = max(max_x, gates[j.gate_name].x + j.x)
            min_y = min(min_y, gates[j.gate_name].y + j.y)
            max_y = max(max_y, gates[j.gate_name].y + j.y)
        estimated_wire_length += max_x-min_x + max_y-min_y

    return estimated_wire_length
```

The time complexity of the function to find semi-perimeter is O(no. of pins) as every pin in every cluster is being visited once.

2.

```
def swap_gates(order): # O(1)
    i, j = random.sample(range(len(gates)), 2)
    new_order = order[:]
    gates[order[i]].x, gates[order[j]].x = gates[order[j]].x, gates[order[i]].x
    gates[order[i]].y, gates[order[j]].y = gates[order[j]].y, gates[order[i]].y
    new_order[i], new_order[j] = new_order[j], new_order[i]
    return new_order
```

Swapping 2 gates randomly will require constant amount of time and hence time complexity of this function is O(1).

3.

```
def check_for_overlaps(gate): #O(g)
    for i in gates:
        if i!=gate and gates[i].overlaps(gates[gate]):
            return True
    return False
```

To check if a gate overlaps with any of the other gates, we will check the possibility of overlap for each of the remaining gates which is done in constant time and hence the overall time complexity is O(no. of gates).

4.

```
def check_all_overlaps(): #((og2))
    for i in range(len(gates)):
        for j in range(i+1, len(gates)):
            i1 = "g"+str(i+1)
            j1 = "g"+str(j+1)
            if gates[i1].overlaps(gates[j1]):
                return True
    return False
```

To check that no 2 gates overlap, the total ways of choosing 2 different gates is gC2.

And to check for each pair, it requires constant amount of time and hence overall complexity of this function is O(no. of gates^2).

5.

```
def place_gates(gates_list, max_height, max_width, dim_grid): #O(g)
    ct = 0
    for i in gates_list:
        gates[i].x = max_width*(ct%dim_grid)
        gates[i].y = max_height*(ct//dim_grid)
        ct+=1
```

This function places the gates in the grid row-wise starting from the bottom left corner and hence the time complexity of this function is O(no. of gates).

6.

```
def compress(best_orientation):
    for i in best_orientation[1]:
        while gates[i].x>=0 and not check_for_overlaps(i):
            gates[i].x-=1
        gates[i].x+=1
    for i in best_orientation[1]:
        while gates[i].y>=0 and not check_for_overlaps(i):
            gates[i].y-=1
        gates[i].y+=1
```

This function compresses the gates so as to remove the unnecessary blank spaces.

The for loop runs for O(g) times. The inner while loop runs for root(g)*max_dim times where max_dim = max(max_height, max_width) and the condition of while loop takes O(g) time where g is number of gates.

Hence the time complexity of this function is O(no. of gates^2.5 * max_dim)

7.

```
def calc_bounding_box(): # O(g)
    min_x = gates["g1"].x
    max_x = min_x + gates["g1"].width
    min_y = gates["g1"].y
    max_y = min_y + gates["g1"].height
    for i in gates:
        min_x = min(min_x, gates[i].x)
        max_x = max(max_x, gates[i].x+gates[i].width)
        min_y = min(min_y, gates[i].y)
        max_y = max(max_y, gates[i].y+gates[i].height)
    return (max_x - min_x), (max_y - min_y)
```

To calculate the dimensions of the bounding box, we just have to keep track of maximum and minimum x and y co-ordinates of the gates and hence the time complexity of this code is O(no. of gates).

8.

```
def solve(max_iter, start_temperature, alpha, init_order, max_height, max_width, dim_grid):
    curr_temperature = start_temperature
    place_gates(init_order, max_height, max_width, dim_grid)
    curr_orientation = [estimate_total_wire_length(), init_order]
    curr_order = init_order[:]
    best_orientation = [curr_orientation[0], init_order] # O(1000)
    iteration = 0
    interval = (int)(max_iter / 10)
    while iteration < max_iter: # (10^6/root g * p)
        new_order = swap_gates(curr_order) # O(1)
        new_orientation = estimate_total_wire_length() # p = number of pins
        if new_orientation < curr_orientation[0]:  # better route so accept
            curr_orientation = [new_orientation, new_order] # O(1)
            curr_order = new_order[:]
            if curr_orientation[0] < best_orientation[0]:
                best_orientation = [new_orientation, new_order] # O(1000)

        else:          # adjacent is worse
            accept_p = np.exp((curr_orientation[0] - new_orientation) / curr_temperature)
            p = random.random()
            if p < accept_p:  # accept anyway
                curr_orientation = [new_orientation, new_order] # O(1000)
                curr_order = new_order[:]
            # else don't accept
        if iteration % interval == 0:
            print("iter = %6d | curr error = %7.2f | temperature = %10.4f " % (iteration, best_orientation[0], curr_temperature))

        if curr_temperature < 0.00001:
            curr_temperature = 0.00001
        else:
            curr_temperature *= alpha
        iteration += 1

    place_gates(best_orientation[1], max_height, max_width. dim grid) # O(1000) O(g)
    compress(best_orientation)
    best_orientation[0] = estimate_total_wire_length()

    return best_orientation
```

In this function, we are performing simulator annealing. The number of cells in the grid is O(no. of gates). For a given area, the bounding rectangle with minimum perimeter is a square. So to minimize the length of wires (approximated using semi-perimeter), each dimension of the grid is O(under_root(no. of gates)).

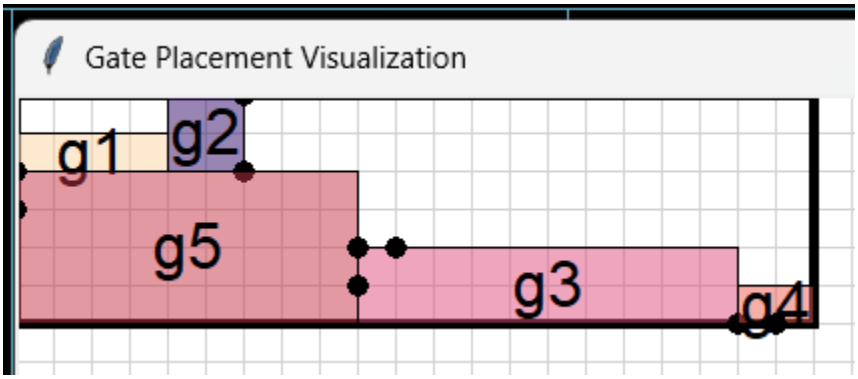
We have set no. of iterations to be 10^6/(dim of grid).

And hence O(10^6/(under_root(no. of gates))).

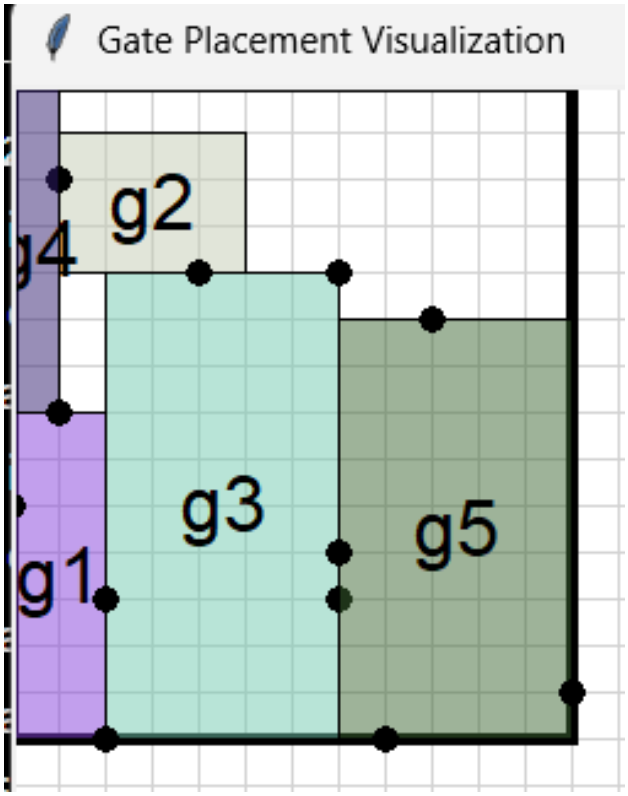
The complexity of each iteration is O(no. of pins) because of the function call “estimate_total_wire_length”.

The complexity of simulator annealing (solve function) is max(while loop, place_gates, compress) Which equals max(10^6/(under_root(no. of gates))*no. of pins, g^2.5*max_dim) where g = number of gates.

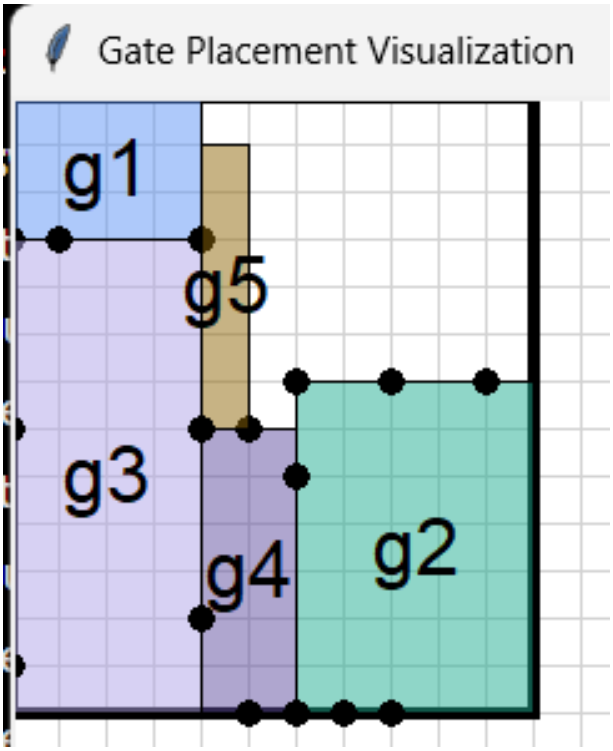
Test Cases :- 1. Randomly generated test cases with max gates = 5, max height/width = 10, max pin density on an edge = 3




```
Number of Gates: 5
Number of Pins: 12
Number of Wires: 6
Wire Length: 50
```



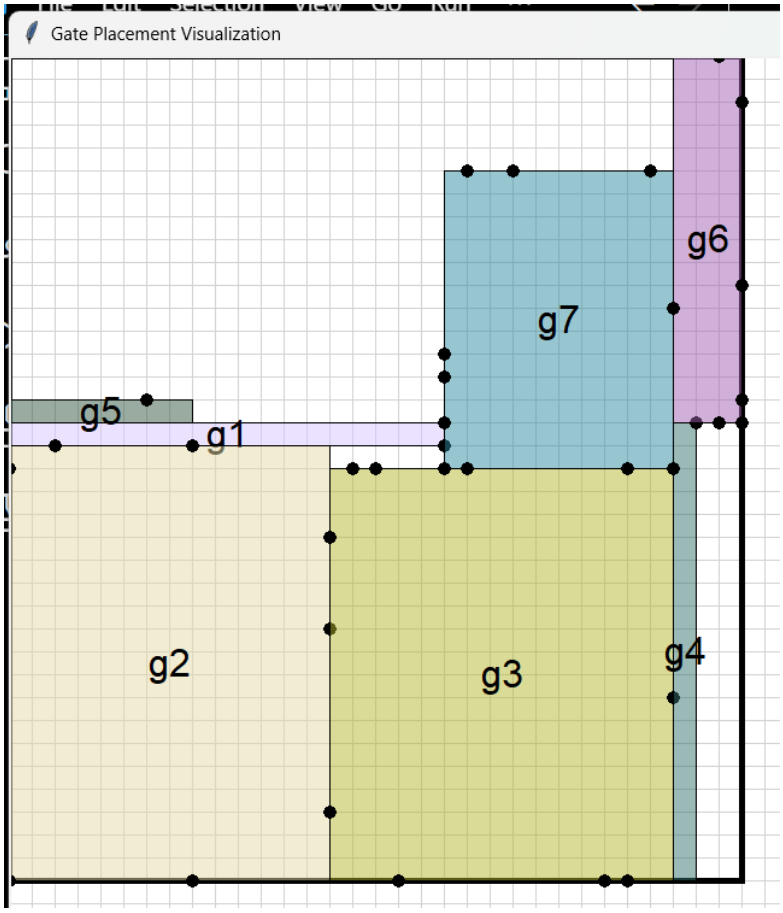

```
Number of Gates: 5
Number of Pins: 12
Number of Wires: 6
Wire Length: 64
```



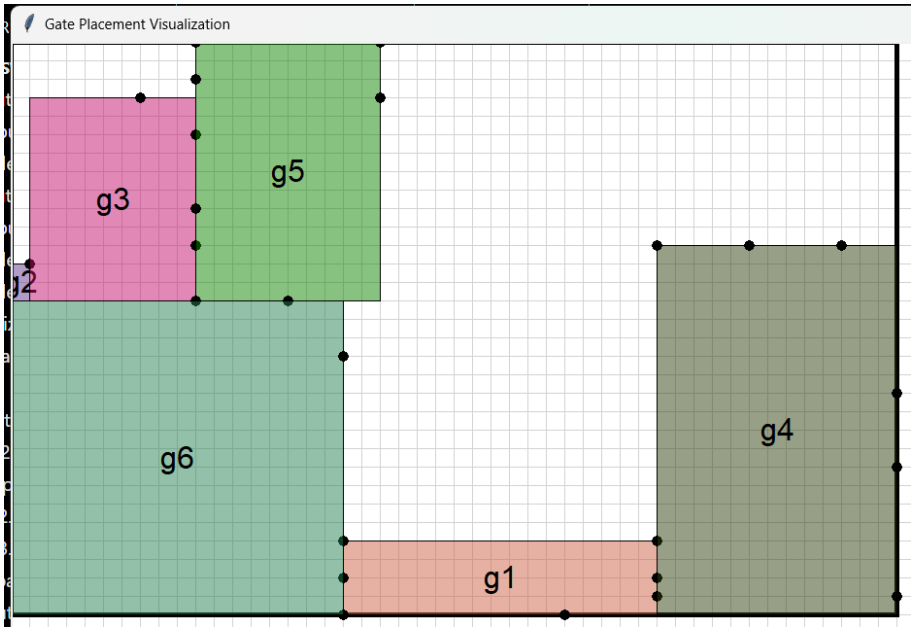

```
Number of Gates: 5
Number of Pins: 18
Number of Wires: 9
Wire Length: 76
```

2. Randomly Generated Test cases with max no. of gates = 10,

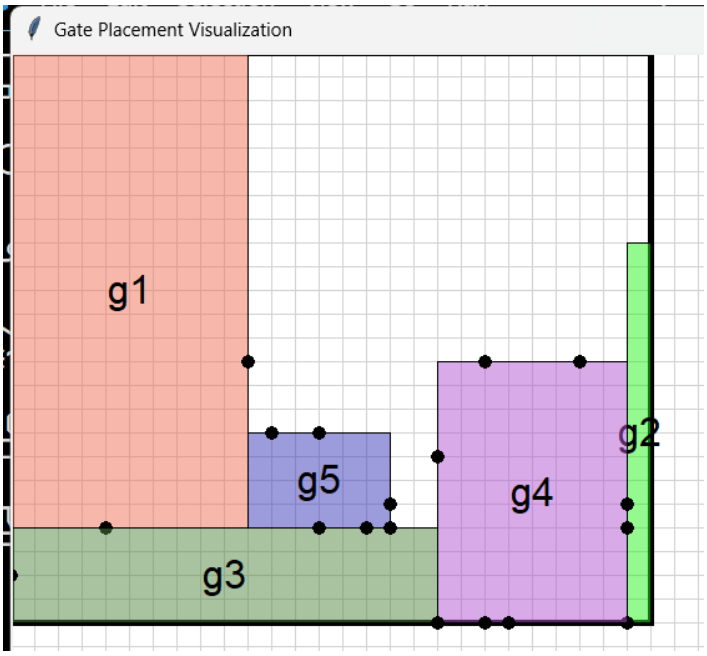
max height/width = 20, max pins per edge = 6




```
Number of Gates: 7
Number of Pins: 36
Number of Wires: 18
Wire Length: 376
```



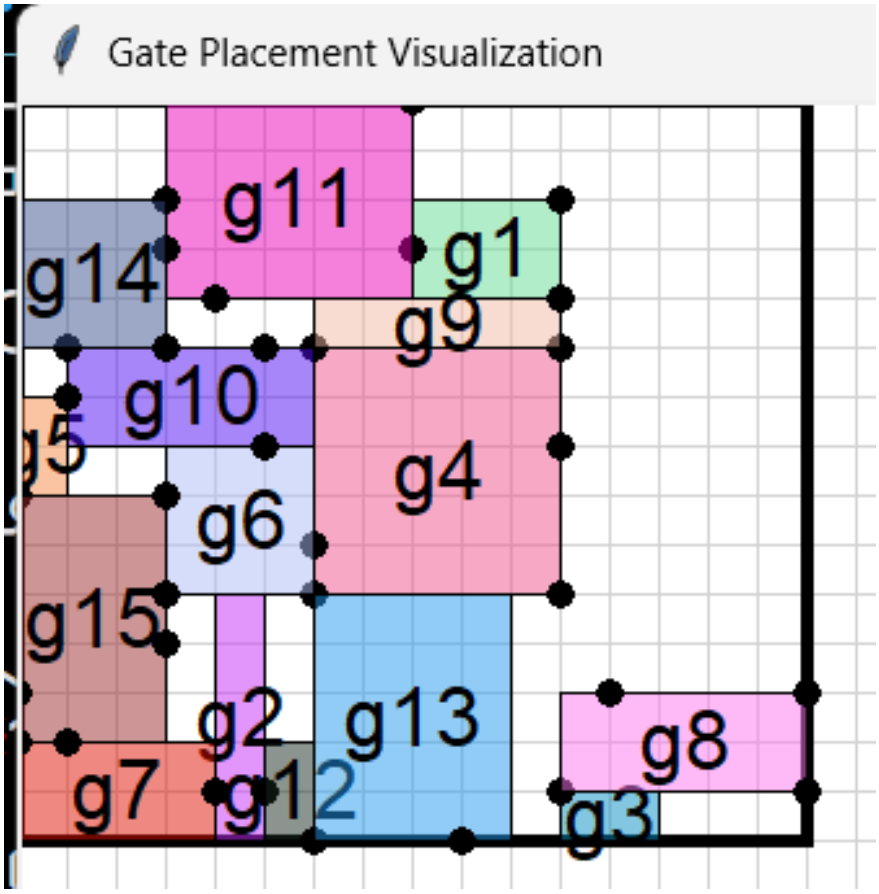

```
Number of Gates: 6
Number of Pins: 26
Number of Wires: 13
Wire Length: 348
```



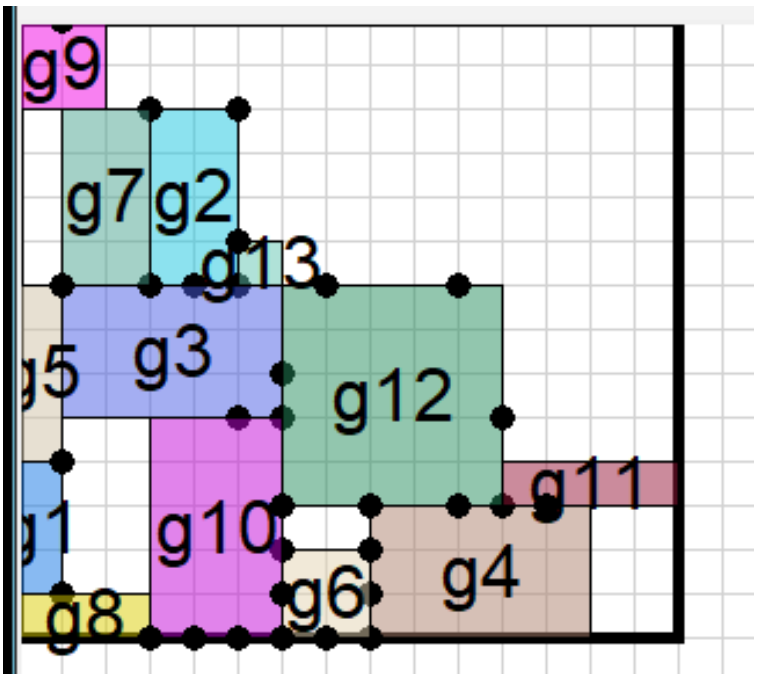

```
Number of Gates: 5
Number of Pins: 18
Number of Wires: 9
Wire Length: 129
```

```
g1 10 20
pins g1 4 0 10 7
g2 1 16
pins g2 0 0
g3 18 4
pins g3 0 2
g4 8 11
pins g4 0 0 0 7 2 0 2 11 3 0 6 11 8 4 8 5
g5 6 4
pins g5 1 4 3 0 3 4 5 0 6 0 6 1
wire g4.p1 g2.p1
wire g4.p2 g3.p1
wire g5.p1 g1.p1
wire g4.p3 g1.p2
wire g5.p2 g4.p4
wire g5.p3 g4.p5
wire g5.p4 g4.p6
wire g5.p5 g4.p7
wire g5.p6 g4.p8
```

3. Randomly generated test cases with max gates = 15, Max width/height = 5, max pins per edge = 3

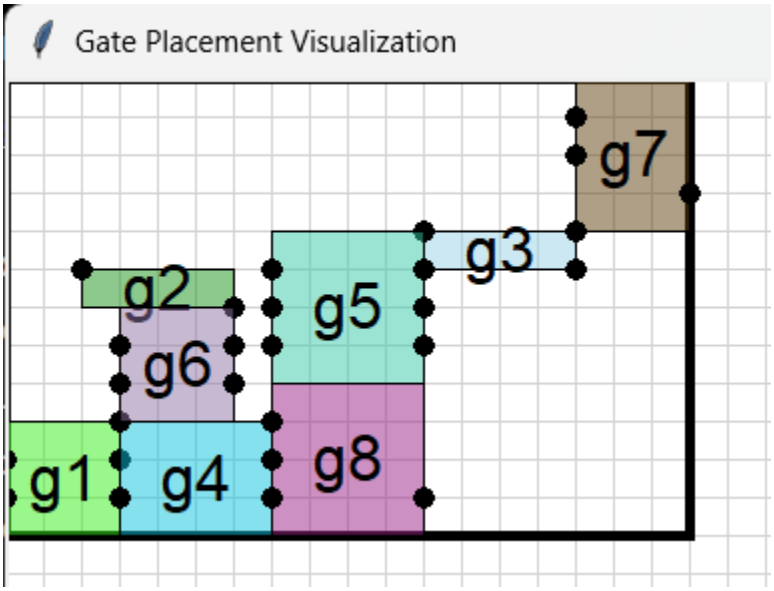



```
Number of Gates: 15
Number of Pins: 36
Number of Wires: 18
Wire Length: 239
```

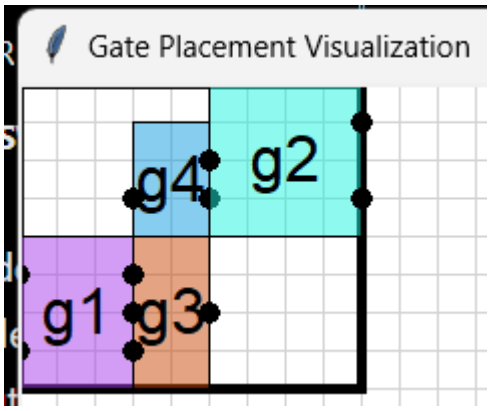



```
Number of Gates: 13
Number of Pins: 36
Number of Wires: 18
Wire Length: 127
```

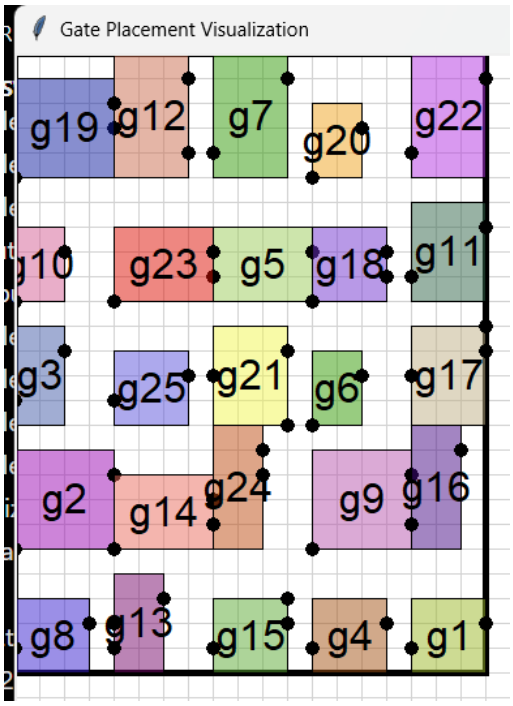
4. Sample Test cases (Moodle)



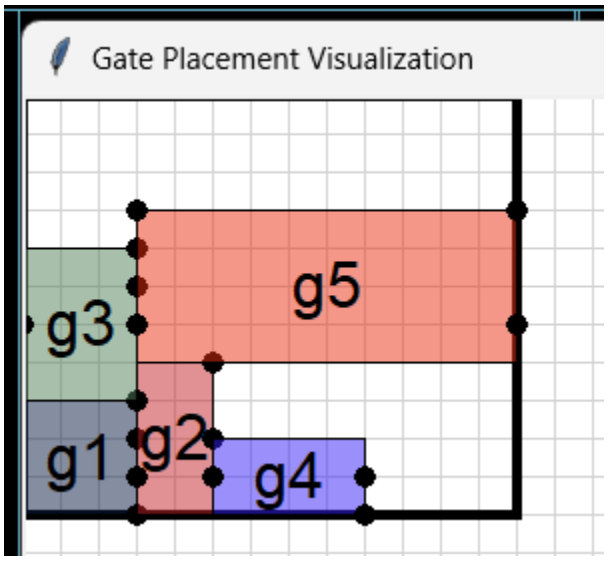
Visualization of tc_1



Visualization of tc_2



Visualisation of tc_3



Visualization of tc_4

Tests have also been done for number of gates = 1000, 500 and 100.

As their visualization cannot be put, the input/output files been added in the submission.

Conclusion :-

The **Wiring-aware Gate Positioning** project tackled the challenge of placing gates on a 2D plane while minimizing wire length between connected pins. By employing **Simulated Annealing**, the algorithm was able to explore a variety of configurations and escape local minima, allowing for near-optimal placement of gates. The use of **Manhattan distance** and the **semi-perimeter method** provided an efficient way to estimate wire lengths, while the **compression step** reduced gaps between gates, ensuring a compact layout.

The proposed **Simulated Annealing** approach, combined with grid-based envelopes and efficient pin mapping, significantly reduced gate overlaps and minimized wire length. The dictionary-based lookup for pin positions further optimized performance, ensuring the algorithm scales well with larger circuits. This flexibility and efficiency make the solution well-suited for handling complex layouts in VLSI design.

Simulated Annealing was particularly effective due to its ability to balance exploration and convergence. Unlike greedy algorithms that risk getting trapped in local optima, Simulated Annealing accepts worse solutions early on, exploring a wider solution space before fine-tuning towards a better configuration. Overall, the algorithm successfully minimized wire length, prevented overlaps, and demonstrated robustness in solving VLSI optimization challenges efficiently.

References :-

Simulated Annealing Optimization Using C# or Python :-

https://visualstudiomagazine.com/Articles/2021/12/01/traveling-salesman.aspx