COL-215
Software Assignment - 2 Report

Shreeraj Jambhale	Pallav Kamad
2023CS50048	2023CS51067

Table of Contents

1 Introduction

1.1 Aim

The aim of this assignment is to develop a comprehensive algorithm for wiring gate-pin positioning in integrated circuits, with a primary focus on minimizing the total wire length between interconnected gates. This involves a thorough understanding of the gates' dimensions and pin configurations, allowing for the strategic placement of gates in a two-dimensional plane while ensuring they do not overlap. The algorithm will employ the semi-perimeter method to accurately estimate wire lengths by calculating the bounding box of connected pins, enabling an effective assessment of wiring efficiency. Furthermore, the report emphasizes the importance of generating output that adheres to specified formats, while also conducting a detailed time complexity analysis to evaluate the algorithm's performance and scalability across various scenarios involving multiple gates and pins. By achieving these objectives, the project aims to contribute to improved circuit design practices and enhanced optimization techniques in electronic engineering.

1.2 Constraints

- $0 < \text{Number of gates} \leq 1000$
- $0 < \text{Width of gate} \leq 100$
- $0 < \text{Height of gate} \leq 100$
- $0 < \text{Number of pins per gate} \leq 40$
- $0 < \text{Total Number of pins} \leq 40000$

2 Approach

The approach taken in this assignment focuses on optimizing gate placement to minimize total wire length in a circuit design. The solution utilizes a skyline packing algorithm to create an initial placement of gates based on their dimensions, efficiently managing the available space in a defined grid. Following this, a simulated annealing algorithm is employed, allowing for iterative refinement of the gate positions. This probabilistic technique explores neighboring solutions while accepting worse configurations with a certain probability to escape local minima. The implementation is further enhanced with parallel processing to leverage multiple CPU cores, significantly improving performance during the optimization phase. Overall, this method aims to achieve an efficient and effective layout while adhering to the constraints of the circuit design.

2.1 Function Explanations

2.1.1 read_input(file)

```
def read_input(file):
    gates = {}
    wires = []

    with open(file, 'r') as f:
        lines = f.readlines()

    i = 0
    while i < len(lines):
        line = lines[i].strip()
        if line.startswith("g"):
            parts = line.split()
            gate_name = parts[0]
            gate = {'width': int(parts[1]), 'height': int(parts[2]), 'pins': []}
            i += 1
            line = lines[i].strip()
            if line.startswith("pins"):
                pin_parts = line.split()[2:]
                gate['pins'] = [(int(pin_parts[j]), int(pin_parts[j + 1])) for j in range(0,
len(pin_parts), 2)]
            gates[gate_name] = gate
        elif line.startswith("wire"):
            wire_parts = line.split()
            wires.append(
                (wire_parts[1].split('.')[0], int(wire_parts[1].split('.')[1][1:]),
                 wire_parts[2].split('.')[0], int(wire_parts[2].split('.')[1][1:]))
            )
        i += 1
    return gates, wires
```

- **Purpose:**
  - To read gate and wire information from an input file and store it in structured data formats (dictionaries and lists).
- **Process:**
  - Opens the specified file for reading.
  - Iterates through each line to identify gates and wires.
  - For gates, it captures their width, height, and pin locations.
  - For wires, it captures the connections between gates and their respective pins.
- **Returns:**
  - A tuple containing:
    - A dictionary of gates with their properties (width, height, and pin coordinates).
    - A list of wires representing connections between gate pins.

2.1.2 find_valid_points(skyline, box_width)

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

    for i in range(1, box_width):
        if skyline[i] != prev:
            points.append([0 if prev > skyline[i] else 1, i, skyline[i]])
        prev = skyline[i]

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

- **Purpose:**
  - To identify potential placement points for a new rectangle based on the current skyline and box width.
- **Process:**
  - Initializes an empty list points to store valid points.
  - Sets the first element of the skyline as prev.
  - Iterates through the skyline from the second index to the end of the box width.
  - Checks for changes in height between consecutive skyline points:
    - If the current height is different from the previous height, a valid point is added to the list:
      - The first element indicates whether the previous height is greater or less than the current height (0 or 1).
      - The second element is the current index (i).
      - The third element is the height at the current index.
  - Finally, appends the last point representing the end of the skyline.

- **Returns:**
  - A list of valid points for rectangle placement, each represented as a list containing:
    - An indicator (0 or 1) for height comparison.
    - The index of the point.
    - The height at that index.

2.1.3 update_skyline(current_skyline, point, rect_height, rect_width)

```
def update_skyline(skyline, point, rect_height, rect_width):
    start = point[1] - rect_width if point[0] else point[1]
    skyline[start:start + rect_width] = [point[2] + rect_height] * rect_width
    return skyline
```

- **Purpose**: Updates the skyline after a rectangle is placed.
- **Process**:
  - Calculates the starting index based on the placement direction (left or right).
  - Updates the skyline segment to reflect the new height of the rectangle.
- **Returns**: The updated skyline.

2.1.4 is_valid_placement

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

- **Purpose:**
  - To check if a rectangle can be placed at a specified point without overlapping existing rectangles.
- **Process:**
  - Calculates the starting index for the rectangle based on the placement direction.
  - Verifies that the skyline heights in the designated area are sufficient for the rectangle's height.
- **Returns:**
  - A boolean indicating whether the rectangle can be placed validly.

2.1.5 pack_rectangles

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

    for rectangle in rectangle_sequence:
        for point in sorted(find_valid_points(skyline, box_width), key=lambda x: (x[2], x[1])):
            if is_valid_placement(rectangle[2], rectangle[1], point, skyline):
                x_pos = point[1] - rectangle[1] if point[0] else point[1]
                placements.append([rectangle[0], x_pos, point[2]])
                update_skyline(skyline, point, rectangle[2], rectangle[1])
                break

    return placements, max(skyline)
```

- **Purpose:**
  - To arrange a sequence of rectangles within a defined width using a skyline packing approach.
- **Process:**
  - Initializes a skyline list to represent the height profile of the placement area.
  - Iterates over each rectangle, finding valid points where it can be placed.
  - Updates the skyline after placing each rectangle to reflect the new heights.
- **Returns:**
  - A tuple containing:
    - A list of placements with the gate names and their coordinates.
    - The maximum height of the skyline after packing.

2.1.6 manhattan_distance

```
def manhattan_distance(x1, y1, x2, y2):
    return abs(x1 - x2) + abs(y1 - y2)
```

- **Purpose:**
  - To calculate the Manhattan distance between two points.
- **Process:**
  - Takes the coordinates of two points and computes the sum of the absolute differences of their respective x and y coordinates.
- **Returns:**
  - The calculated Manhattan distance as an integer.

2.1.7 total_wire_length

```python
def total_wire_length(gate_positions, wires, gates_dict):
    wire_length = 0

    # Calculate the total wire length based on the wires and gate positions
    for wire in wires:
        gate1, pin1, gate2, pin2 = wire

        g1_x, g1_y = gate_positions.get(gate1, (None, None))
        g2_x, g2_y = gate_positions.get(gate2, (None, None))

        if g1_x is None or g2_x is None:
            print(f"Warning: Gate {gate1} or {gate2} is missing in the positions.")
            continue

        if pin1 > len(gates_dict[gate1]['pins']) or pin1 <= 0:
            print(f"Warning: Invalid pin index {pin1} for gate {gate1}")
            continue
        if pin2 > len(gates_dict[gate2]['pins']) or pin2 <= 0:
            print(f"Warning: Invalid pin index {pin2} for gate {gate2}")
            continue

        p1_x, p1_y = gates_dict[gate1]['pins'][pin1 - 1]
        p2_x, p2_y = gates_dict[gate2]['pins'][pin2 - 1]

        pin1_pos = (g1_x + p1_x, g1_y + p1_y)
        pin2_pos = (g2_x + p2_x, g2_y + p2_y)

        distance = abs(pin1_pos[0] - pin2_pos[0]) + abs(pin1_pos[1] - pin2_pos[1])
        wire_length += distance

    return wire_length
```

- **Purpose:**
  - To compute the total wire length based on the positions of gates and their connections.
- **Process:**
  - Iterates over each wire, retrieves the gate positions and pin coordinates, and calculates the distance between connected pins using the Manhattan distance formula.
- **Returns:**
  - The total wire length as an integer.

2.1.8 generate_neighbor

```python
def generate_neighbor(gate_positions, grid_size, gates, gates_dict):
    new_positions = gate_positions.copy()
    gate_to_move = random.choice(list(new_positions.keys()))
    current_x, current_y = new_positions[gate_to_move]

    delta_x = random.randint(-5, 5)
    delta_y = random.randint(-5, 5)

    new_x = max(0, min(current_x + delta_x, grid_size[0] - gates_dict[gate_to_move]['width']))
    new_y = max(0, min(current_y + delta_y, grid_size[1] - gates_dict[gate_to_move]['height']))

    new_positions[gate_to_move] = (new_x, new_y)
    return new_positions
```

- **Purpose:**
  - To create a new random arrangement of gate positions by slightly moving one gate.
- **Process:**
  - Randomly selects a gate and adjusts its position within a limited range, ensuring it remains within grid boundaries.
- **Returns:**
  - A new dictionary of gate positions reflecting the adjusted coordinates.

2.1.9 no_gate_overlap

```
def no_gate_overlap(gate_positions, gates_dict):
    positions = [(gate, *gate_positions[gate], gates_dict[gate]['width'], gates_dict[gate]['height'])
                 for gate in gate_positions]

    for i, (g1, x1, y1, w1, h1) in enumerate(positions):
        for g2, x2, y2, w2, h2 in positions[i + 1:]:
            if not (x1 + w1 <= x2 or x1 >= x2 + w2 or y1 + h1 <= y2 or y1 >= y2 + h2):
                return False
    return True
```

- **Purpose:**
  - To verify that the current gate placements do not overlap with one another.
- **Process:**
  - Compares each pair of gates and checks if their rectangular areas intersect.
- **Returns:**
  - A boolean indicating whether there is no overlap (True) or if overlaps exist (False).

2.1.10 simulated_annealing

```
def simulated_annealing(gates, wires, grid_size, initial_temp, final_temp, alpha, num_neighbors):
    sorted_gates = sorted(gates.values(), key=lambda g: (g['height'], g['width']), reverse=True)
    gate_sequence = [(name, gate['width'], gate['height']) for name, gate in gates.items()]

    # Initial solution using skyline heuristic
    initial_placements, _ = pack_rectangles(gate_sequence, grid_size[0])
    gate_positions = {placement[0]: (placement[1], placement[2]) for placement in initial_placements}

    gates_dict = {gate: gates[gate] for gate in gates}
    current_solution = gate_positions
    current_wire_length = total_wire_length(gate_positions, wires, gates_dict)
    current_temp = initial_temp

    while current_temp > final_temp:
        for _ in range(num_neighbors):
            new_solution = generate_neighbor(current_solution, grid_size, gates, gates_dict)
            if no_gate_overlap(new_solution, gates_dict):
                new_wire_length = total_wire_length(new_solution, wires, gates_dict)
                delta_length = new_wire_length - current_wire_length

                if delta_length < 0 or random.uniform(0, 1) < math.exp(-delta_length / current_temp):
                    current_solution = new_solution
                    current_wire_length = new_wire_length

        current_temp *= alpha

    return current_solution, current_wire_length
```

- **Purpose:**
  - To optimize gate placements using the simulated annealing technique to minimize total wire length.
- **Process:**
  - Initializes gate positions using a skyline heuristic.
  - Iteratively generates neighbors and evaluates their wire lengths, accepting new positions based on a probability that decreases with temperature.
- **Returns:**
  - The best gate positions and the corresponding wire length.

2.1.11 parallel_simulated_annealing

```
def parallel_simulated_annealing(gates, wires, grid_size, initial_temp, final_temp, alpha,
num_iterations,
                                 num_neighbors):
    with Pool(cpu_count()) as pool:
        results = pool.starmap(simulated_annealing,
                               [(gates, wires, grid_size, initial_temp, final_temp, alpha,
                                 num_neighbors)] * num_iterations)

    return min(results, key=lambda x: x[1])
```

- **Purpose:**
  - To execute the simulated annealing process in parallel to speed up the optimization.
- **Process:**
  - Utilizes multiple CPU cores to perform independent simulated annealing runs, gathering the results to find the best solution.
- **Returns:**
  - The best gate positions and the lowest wire length found among all runs.

2.1.12 write_output

```
def write_output(output_file, final_solution, gates_dict, best_wire_length):
    min_x = min(x for x, y in final_solution.values())
    min_y = min(y for x, y in final_solution.values())

    shifted_positions = {gate: (x - min_x, y - min_y) for gate, (x, y) in final_solution.items()}
    max_x = max(x + gates_dict[gate]['width'] for gate, (x, y) in shifted_positions.items())
    max_y = max(y + gates_dict[gate]['height'] for gate, (x, y) in shifted_positions.items())

    with open(output_file, 'w') as f:
        f.write(f"bounding_box {max_x} {max_y}\n")
        for gate, (x, y) in shifted_positions.items():
            f.write(f"{gate} {x} {y}\n")
        f.write(f"wire_length {best_wire_length}")
```

- **Purpose:**
  - To write the final gate positions and total wire length to an output file.
- **Process:**
  - Calculates the bounding box dimensions and adjusts gate positions to fit within it, then writes the formatted results to the specified output file.
- **Returns:**
  - None (the function performs file I/O without returning values).

2.1.13 Main

```
st = time.perf_counter()
gates, wires = read_input("input.txt")
gates_num = len(gates)
wire_num = len(wires)
grid_size = (5000, 5000)
initial_temp = 1000
final_temp = 0.01
alpha = 0.99
num_iterations = 10
num_neighbors = 100

if gates_num > 150 or wire_num > 250:
    alpha = 0.99
    final_temp = 0.01

if gates_num < 30 or wire_num < 50:
    alpha = 0.999
    final_temp = 0.001

best_solution, best_wire_length = parallel_simulated_annealing(
    gates, wires, grid_size, initial_temp, final_temp, alpha, num_iterations, num_neighbors
)

write_output("output.txt", best_solution, gates, best_wire_length)
```

```
et = time.perf_counter()
print(f"Wire length: {best_wire_length} units")
print(f"Execution time: {et - st:.2f} seconds")
```

- **Purpose:**
  - To serve as the entry point for the program, orchestrating the execution of reading input, running the optimization process, and writing output results.
- **Process:**
  - Records the start time for performance measurement.
  - Calls the read_input function to load gate and wire data from a specified input file.
  - Sets up the grid size and initial parameters for the simulated annealing process.
  - Adjusts the cooling parameters based on the number of gates and wires.
  - Executes the parallel_simulated_annealing function to find the optimal gate placements.
  - Calls write_output to save the final gate positions and wire length to an output file.
  - Records the end time and prints the total wire length and execution time.
- **Returns:**
  - None (the function primarily orchestrates other functions without returning values).

2.2 Simulated Annealing:

Simulated annealing is a probabilistic optimization algorithm inspired by the physical process of heating and cooling materials. The algorithm begins at a high temperature, which allows it to explore a wide range of solutions, including those that may be worse than the current solution. This mechanism helps avoid getting stuck in local minima by allowing the acceptance of poorer solutions with a certain probability. As the temperature gradually decreases, the algorithm becomes more selective, focusing on refining the solution to find an optimal or near-optimal outcome. The rate of cooling is crucial, as it influences the algorithm's ability to escape local minima while converging towards a global minimum

2.3 Parallel Processing:

Parallel processing involves dividing a task into smaller sub-tasks that can be executed concurrently across multiple CPU cores or processors. In the context of simulated annealing, parallel processing is employed to enhance performance and efficiency. Multiple instances of the simulated annealing algorithm are run simultaneously, each exploring different solution paths independently. This approach allows for a more comprehensive search of the solution space in a shorter time. Once all parallel processes complete, the best solution among them is selected, leading to faster convergence to an optimal solution and improved overall performance of the algorithm.

2.4 Main Wire Length Calculating Function

This function is imported from different file provided along with main file names as wirelength.py and it utilizes the created intermediate output.txt and overwrites it again calculating the wirelength in prescribed way in assignment doc using semi perimeter method

Breakdown of the function:

1. **Reading Input**:
   - The function reads two files: dimensions_file (containing gate dimensions, pin positions, and wire connections) and coordinates_file (containing gate positions).
   - The data is parsed and stored in dictionaries for easy access later.
2. **Data Structures**:
   - gates: Stores each gate's width and height.
   - pins: Stores pin locations relative to their respective gates.
   - wires: Contains wire connections between pins on different gates.
3. **Gate and Pin Positions**:
   - gate_positions: Stores the (x, y) coordinates of each gate.
   - pin_positions: Uses the gate positions to calculate the absolute positions of the pins (by adding the pin offsets to the gate coordinates).
4. **Distance Matrix**:
   - A distance_matrix is constructed to calculate the Manhattan distance between all pairs of pins. Pins from the same gate are ignored (set to infinity).
5. **Connection Matrix**:
   - A connection_matrix is built to record which pins are connected based on the wire information. This matrix tracks pin-to-pin connections.
6. **Bounding Box Calculation**:
   - For each connected group of pins, the function calculates a **bounding box**, which is the smallest rectangle that can enclose all connected pins.
7. **Semi-Perimeter Calculation**:
   - Instead of calculating the full perimeter of the bounding box, the function computes the **semi-perimeter**, which is:

$$(x_{max} - x_{min}) + (y_{max} - y_{min})$$

This represents half the perimeter, providing a minimal wire length estimate for connecting the pins within the bounding box.

8. **Total Wire Length**:
   - The **total wire length** is the sum of the semi-perimeters of all bounding boxes enclosing connected pins. This provides the overall wire length required for the circuit layout.
9. **Return**:
   - The function returns the total wire length.

3 Time Complexity Analysis

1. **read_input(file)**:
    - **Time Complexity**: O(n)
    - This function reads all lines from the input file. Let n be the total number of lines in the input file. Each line is processed in constant time.
2. **find_valid_points(skyline, box_width)**:
    - **Time Complexity**: O(w)
    - This function scans through the skyline (which has a length of box_width) to find valid points. The complexity is linear with respect to box_width.
3. **update_skyline(skyline, point, rect_height, rect_width)**:
    - **Time Complexity**: O(w)
    - This function updates a segment of the skyline (length rect_width). The worst case is when it updates a segment equal to box_width.
4. **is_valid_placement(rect_height, rect_width, point, skyline)**:
    - **Time Complexity**: O(w)
    - In the worst case, this function checks all indices for validity, proportional to rect_width.
5. **pack_rectangles(rectangle_sequence, box_width)**:
    - **Time Complexity**: $O(m \cdot w)$
    - For each rectangle in rectangle_sequence (let's denote the number of rectangles as m), it calls find_valid_points, is_valid_placement, and updates the skyline, each taking O(w). Hence, the total complexity is $O(m \cdot w)$.
6. **total_wire_length(gate_positions, wires, gates_dict)**:
    - **Time Complexity**: O(k)
    - This function processes each wire (k wires), looking up positions and calculating distances, which takes constant time for each wire.
7. **generate_neighbor(gate_positions, grid_size, gates, gates_dict)**:
    - **Time Complexity**: O(1)
    - This function generates a new position for a randomly selected gate, taking constant time.
8. **no_gate_overlap(gate_positions, gates_dict)**:
    - **Time Complexity**: $O(g^2)$
    - This function checks for overlaps between gates, iterating over each pair of gates. If there are g gates, it involves a double loop, resulting in $O(g^2)$.
9. **simulated_annealing(…):**
    - **Time Complexity**: $O(\text{num\_neighbors} \cdot (g^2 + m \cdot w + k))$
    - The while loop runs until current_temp reaches final_temp, and for each of the num_neighbors, it generates a neighbor and checks for overlaps, resulting in complexities from the total_wire_length and pack_rectangles functions as well.
10. **parallel_simulated_annealing(...)**:
    - **Time Complexity**: $O(p \cdot (\text{num\_neighbors} \cdot (g^2 + m \cdot w + k)))$
    - Here, p is the number of parallel processes (determined by cpu_count()). Each process runs simulated_annealing, so the total complexity becomes p multiplied by the complexity of simulated_annealing.

11. **write_output(output_file, final_solution, gates_dict, best_wire_length)**:
    - **Time Complexity**: O(g)
    - This function writes the output, iterating through the gates, resulting in a linear complexity with respect to the number of gates.

12. Overall Time Complexity Calculation

Putting it all together, the overall time complexity is dominated by the parallel simulated annealing step, resulting in:

Overall Time Complexity:

$$O(n+m \cdot w+k+p \cdot num\_neighbors \cdot (g^2+m \cdot w+k)+g)$$

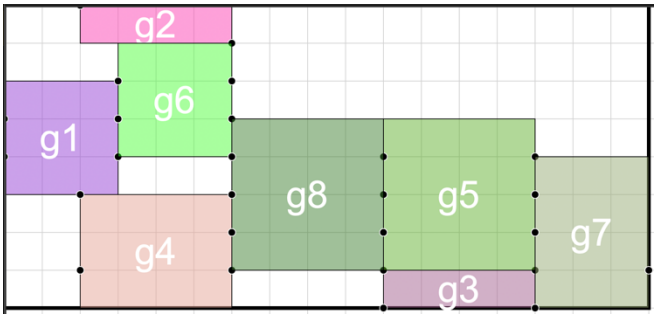
This complexity accounts for all major components of the program.

4 Result and Analysis

4.1 Provided Test Cases

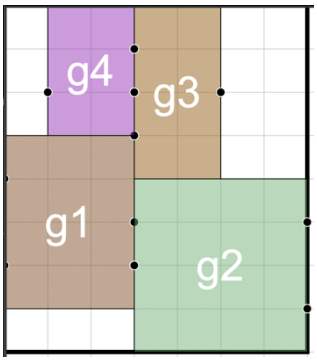
4.1.1 Test Case – 1

- Optimized Wire Length – 26 Units (Terminal Output - total wire length is: 26)



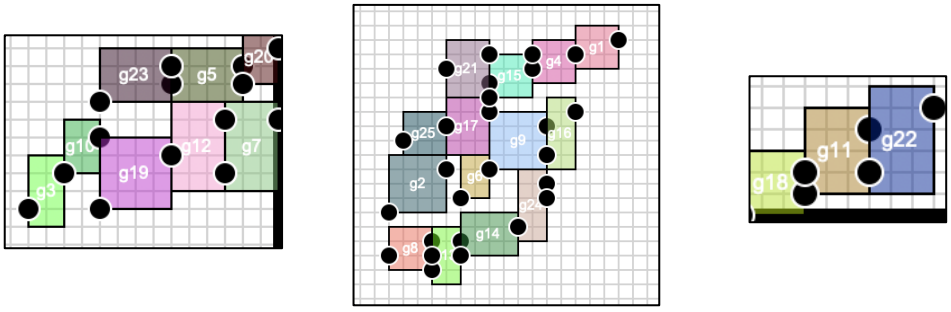
4.1.2 Test Case – 2

- Optimized Wire Length – 26 Units (Terminal Output - total wire length is: 26)



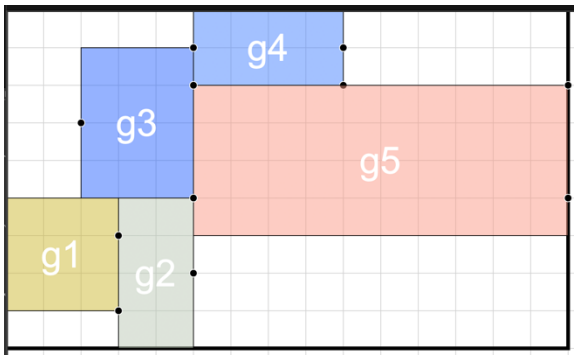
4.1.3 Test Case – 3

- Optimized Wire Length – 49 (The test case was designed to form three clusters where the distance between them does not affect the wire length, ensuring the focus remains solely on wire length minimization. Here are the three clusters formed)



4.1.4 Test Case – 4

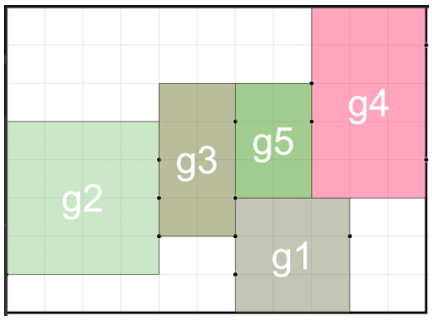
- Optimized Wire Length – 34 Units (Terminal Output - total wire length is: 31)



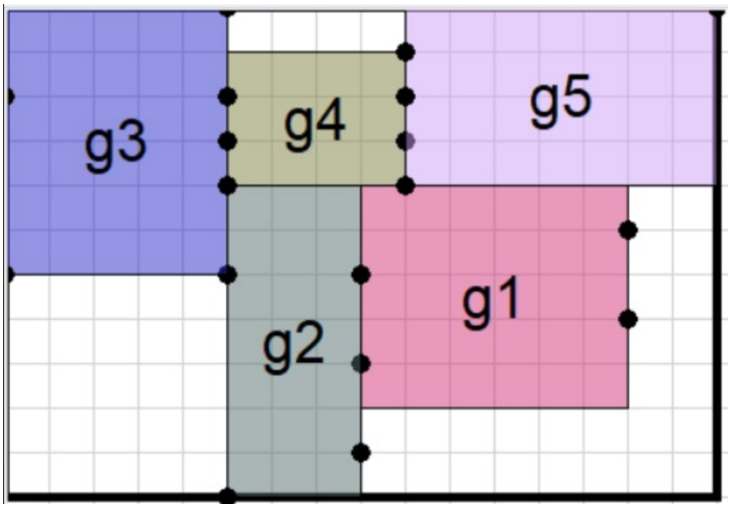
4.2 Self Test Cases

4.2.1 Test Case – 1. (5 Gates)

- Optimized Wire Length – 10 units
  Less connections and smaller gates

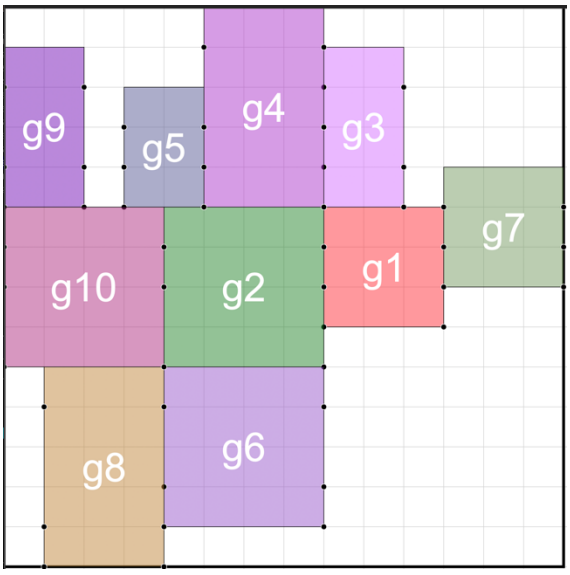


- Optimized Wire Length – 72 units
  High number of connections



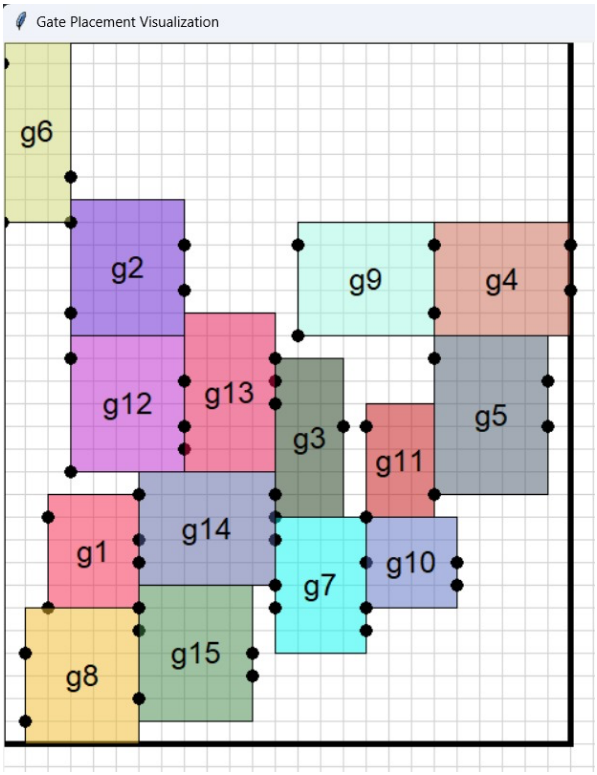
4.2.2 Test Case – 3 (10 Gates)

- Optimized Wire Length – 65



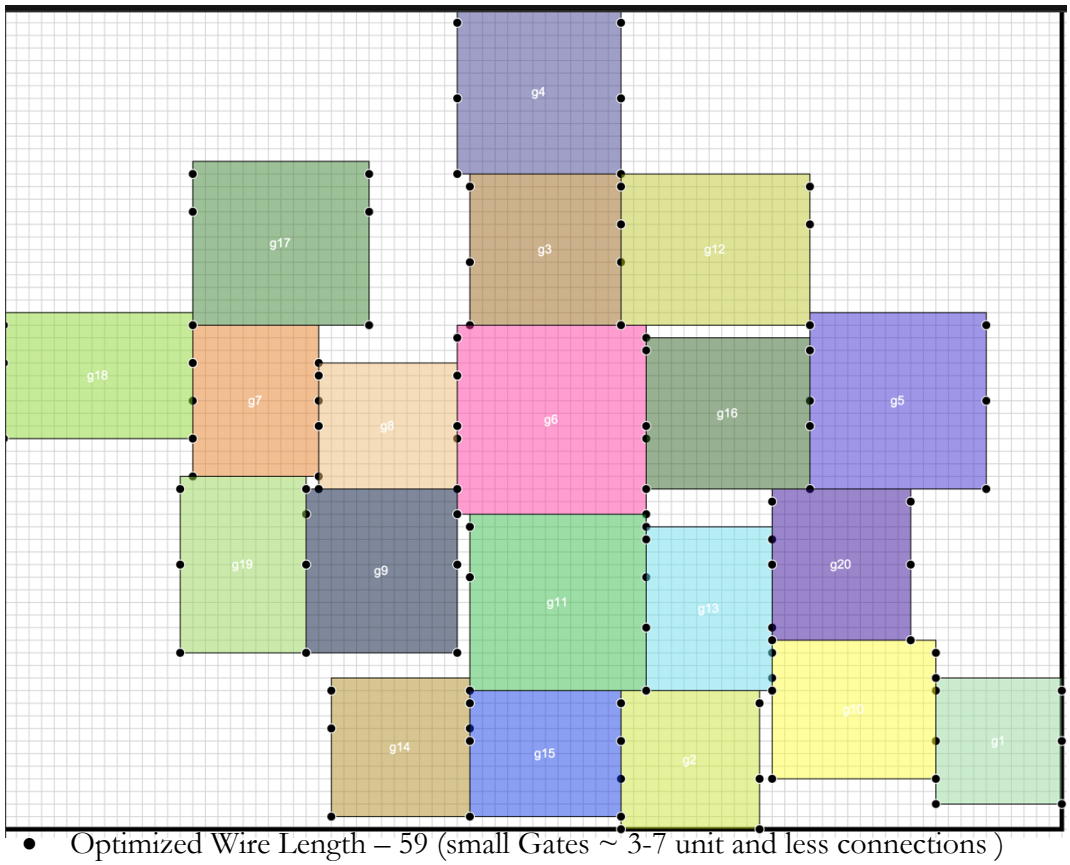
4.2.3 Test Case – 4 (15 Gates)

- Optimized Wire Length – 44

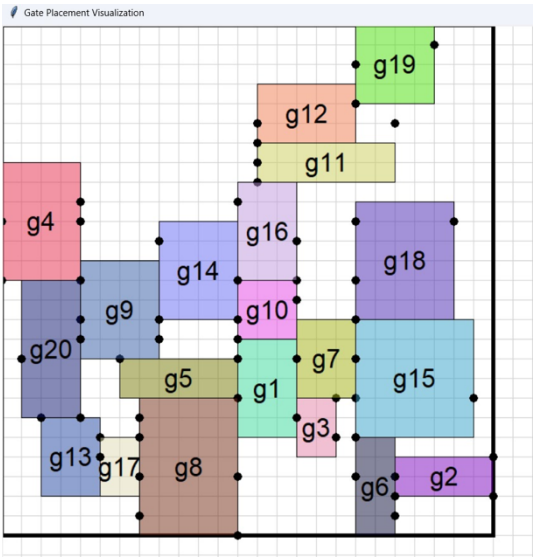


4.2.4 Test Case – 5(20 Gates)

- Optimized Wire Length – 1061(big Gates ~ 10-15 unit and More Connection~ 50 wires)

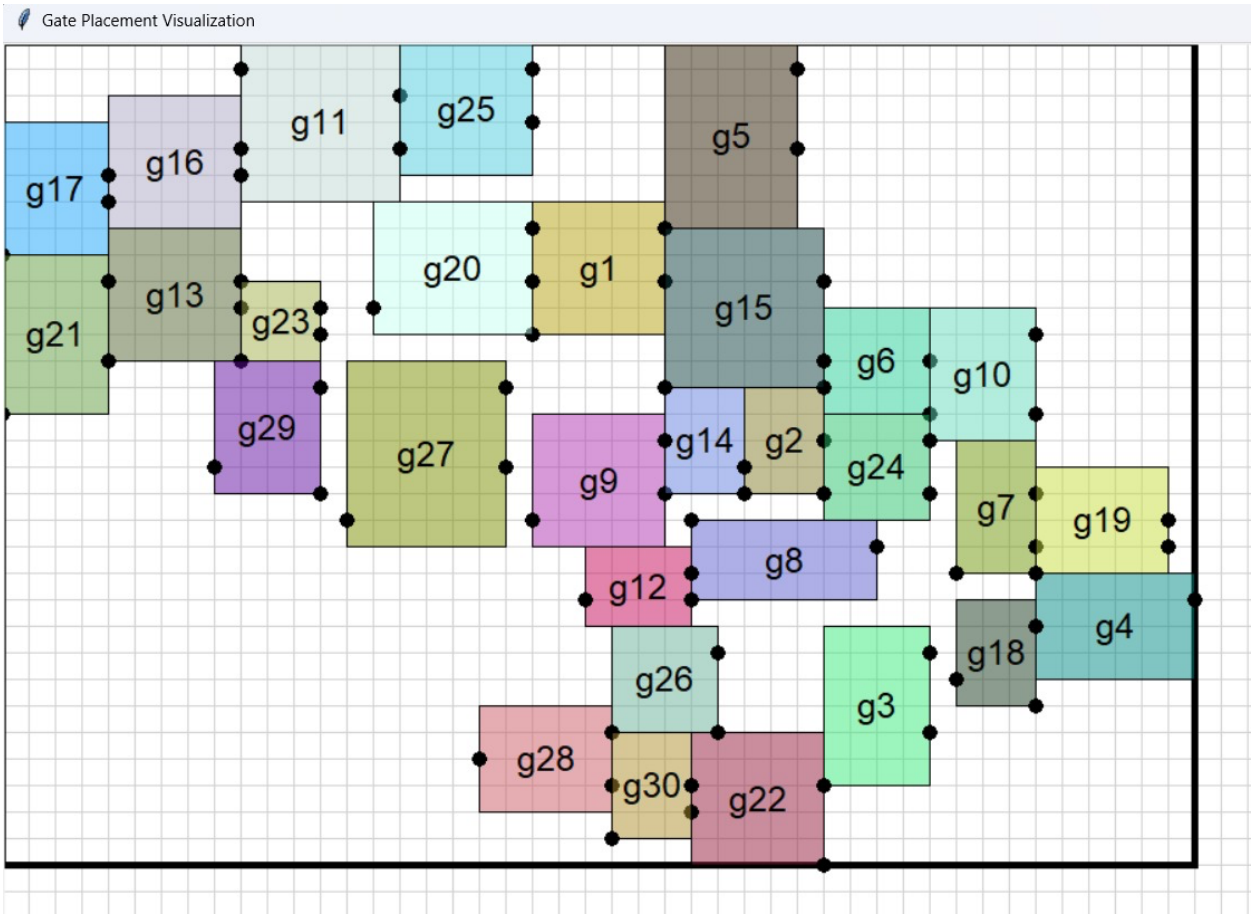


- Optimized Wire Length – 59 (small Gates ~ 3-7 unit and less connections)



4.2.5 Test Case – 7 (30 Gates)

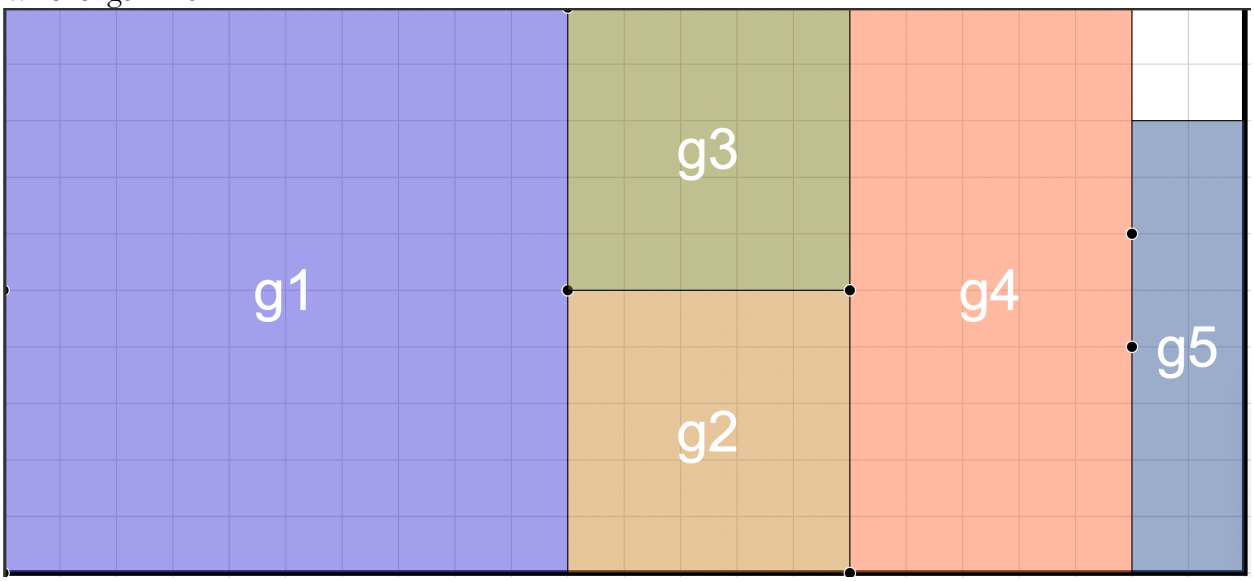
- Optimized Wire Length – 103
-



4.2.6 Edge Case

4.2.6.1 When wire length Is 0

Wire length = 0



4.2.7 No wire connections

Wire length = 0
Sample input -
g1 10 10
pins g1 10 5 10 10 0 0 0 5
g2 5 5
pins g2 0 5 5 0
g3 5 5
pins g3 0 5 5 0
g4 5 10
pins g4 0 0 0 5 5 4 5 6
g5 2 8
pins g5 0 4 0 6
****no wire connection****

output – no relation b/w gates so gates can be places anywhere on grid

bounding_box 1444 963
g1 1195 722
g2 1439 554
g3 327 958
g4 843 310
g5 0 0
wire_length 0

----------X----------