COL-215
Software Assignment - 3 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 gate positioning in integrated circuits, focusing on minimizing the **critical path delay**. The task involves placing gates strategically on a two-dimensional plane while ensuring that gates do not overlap. By calculating the critical path delay, which is the longest delay from any primary input to a primary output, the goal is to optimize the gate positions to minimize the overall delay. The algorithm must consider gate and wire delays, using the **semi-perimeter method** to estimate wire lengths by calculating the bounding box of connected pins.

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 minimizing the critical path delay in gate placement optimization for integrated circuits. The solution starts by reading the gate dimensions, pin locations, and wire connections from the input. Gates are then initially placed on a grid using a greedy algorithm that ensures no overlap. The optimization process is performed using simulated annealing, which iteratively adjusts the gate positions while minimizing the maximum delay between input and output pins. This probabilistic method allows exploration of suboptimal solutions to avoid local minima. The implementation is parallelized, utilizing multiple CPU cores to speed up the optimization process, leading to an efficient circuit layout with reduced critical path delay.

2.1 Function Explanations

2.1.1 read_input(file)

- **Inputs:**
  - To read gates, pins, 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.
  - It captures their width, height, delay, and pin locations for gates.

- 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 primary_pins(gate_dict, wires)

- **Inputs:**
  - To identify all the primary input pins and the primary output pins.
- **Process:**
  - Identifies and returns the primary input and output pins of the circuit by analyzing the wires.
  - Any gate pin that is not a destination of any wire is considered a primary input.
  - Any gate pin that is not a source of any wire is considered a primary output.
- **Returns:**
  - main_inputs (set): Set of primary input pins.
  - main_outputs (set): Set of primary output pins.

2.1.3 find_all_paths_and_cycles(gates, wires, input_pins, output_pins)

- **Inputs:**
  - gates (dict): Dictionary of gates with attributes.
  - wires (list): List of wire connections between gates.
  - Input_pins (set): Set of primary input pins.
  - output_pins (set): Set of primary output pins.
- **Process**:
  - Searches for all possible paths for the pins of gates using the wire data.
  - Also checks for cyclic paths that could indicate errors in the circuit (a cycle means the circuit is invalid).
- **Returns**:
  - has_cycle (bool): Whether a cycle was found in the circuit.
  - cycle_path (list): The path of the cycle if one was found.
  - all_circuit_paths (list): All valid paths from input pins to output pins.
  - total_paths (int): The total number of paths found in the circuit.

2.1.4 greedy_grid_placement(gate_dict)

- **Inputs:**
  - gate_dict (dict): A dictionary of gates with attributes such as width, height, delay, and pin locations.
- **Process:**
  - Performs a greedy placement of gates on a 2D grid.
  - Places each gate on the grid with equal-sized cells based on the maximum width and height of the gates.
  - Assigns x and y coordinates to each gate based on its position in the grid.
- **Returns:**
  - placements (dict): Dictionary with the placement coordinates (x, y) of each gate.
  - grid_size (tuple): Dimensions of the grid used for placement.

2.1.5 generate_neighbours(gate_positions, grid_size, gates_dict)

- **Inputs:**
  - gate_positions (dict): Current positions of the gates.
  - grid_size (tuple): The size of the grid used for placement.
  - gates_dict (dict): Dictionary of gate attributes (width, height, etc.).
- **Process:**
  - Generates a new neighbours solution by slightly modifying the position of one randomly chosen gate.
  - Ensures that the new position is within the grid bounds by adjusting the x and y coordinates..
- **Returns:**
  - Returns the updated gate positions (dict) after moving one gate to a new location.

2.1.6 manhattan_distance

- **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 calculate_path_delay(path, gates, wire_delay_per_unit)

- **Input:**
  - path (list): The list of gate-pin connections that form a path.
  - gates (dict): Dictionary of gate attributes.
  - wire_delay_per_unit (int): Delay per unit of wire length.
- **Process:**
  - Calculates the total delay for a given path, accounting for the delays of the gates and the wire lengths between connected pins.

  - For connections between different gates, it calculates the Manhattan distance between their connected pins and adds the wire delay accordingly.
- **Returns:**
  - Returns the total delay (int) of the given path..

2.1.8 simulated_annealing(gates, initial_temp, final_temp, alpha, num_neighbourss, wire_delay, all_paths)

- **Inputs:**
  - gates (dict): Dictionary of gate attributes.
  - initial_temp (float): The starting temperature for the simulated annealing process.
  - final_temp (float): The temperature threshold to stop the algorithm.
  - alpha (float): The cooling rate, controlling how the temperature decreases.
  - num_neighbourss (int): Number of neighboursing solutions to generate at each step.
  - wire_delay (int): Delay associated with the wire connections.
  - all_paths (list): List of all valid paths between gates in the circuit.
- **Process:**
  - Implements a simulated annealing algorithm to optimize the placement of gates on the grid.
  - Tries to minimize the maximum path delay by exploring different gate placements.
  - At each iteration, a new gate placement is generated, and it may be accepted depending on the difference in delay and the current temperature.
  - Continues this process while cooling down the temperature until the final temperature is reached..
- **Returns:**
  - best_solution (dict): The best gate placement found.
  - best_delay (float): The lowest maximum path delay achieved.
  - best_path (list): The path with the highest delay (critical path).

2.1.9 Main

- **Inputs:**
  - 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 Critical Path Delay 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. **Data Structures**:
    - gates: Stores each gate's width, height, delay, and pin locations.
    - wires: Contains wire connections between pins on different gates.
2. **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).
3. **Total Critical Delay**:
    - The **total critical delay** is the minimum of all the critical delays for each arrangement of gates.
4. **Return**:
    - The function returns the total critical delay.

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. **primary_pins(gate_dict, wires)**:
    - **Time Complexity**: O(g+w)
    - Let g be the number of gates and w be the number of wires.
    - The function iterates over all gates and their pins in the dictionary, which takes O(g).
    - It also iterates over the wire list to build sets of input and output pins, which takes O(w).
3. **find_all_paths_and_cycles(gates, wires, input_pins, output_pins)**:
    - **Time Complexity**: O((g+w)*p)
    - p is the number of valid paths.
    - It finds all paths between inputs and outputs, in worst case it visits all the paths thus giving this complexity.
4. **greedy_grid_placement(gate_dict)**:
    - **Time Complexity**: O(g)
    - Let g be the number of gates.
    - The function iterates over the list of gates and assigns coordinates to each one. Each gate is processed in constant time, so the overall time complexity is O(g).
5. **generate_neighbours(gate_positions, grid_size, gates_dict)**:
    - **Time Complexity**: O(1)
    - The function randomly selects a gate and moves it by a small delta in the x and y directions. This involves constant-time operations, making the time complexity O(1).
6. **calculate_path_delay(path, gates, wire_delay_per_unit)**:
    - **Time Complexity**: O(p)
    - Let p be the number of pins in the path.
    - The function iterates through the path and performs a constant-time calculation for each pair of consecutive nodes. The overall time complexity is O(p), where p is the length of the path.
7. **give_max_path(all_paths, gates, wire_delay)**:
    - **Time Complexity**: O(a*p)
    - Let a be the total number of paths in all_paths and p be the average number of pins in each path.
    - The function iterates over all paths and calls calculate_path_delay for each one, which takes O(p) time. The overall time complexity is O(a * p).
8. **simulated_annealing(gates, initial_temp, final_temp, alpha, num_neighbourss, wire_delay, all_paths)**:
    - **Time Complexity:** O(i * n * (a * p))
    - Let i be the number of iterations (determined by the cooling schedule), n be the number of neighbours checked per iteration, a be the number of paths, and p be the average number of pins in a path.
    - For each iteration, the algorithm generates a neighbours solution and checks whether it improves the result by computing the maximum path delay. Generating the neighbours takes O(1), and calculating the delay for all paths takes O(a * p).
    - Since the loop runs for i iterations and checks n neighbourss per iteration, the overall complexity is O(i * n * (a * p).
9. **parallel_simulated_annealing(...)**:
    - **Time Complexity:** O(num_iterations * i * n * (a * p) / k)
    - Let k be the number of CPU cores available for parallel execution.
    - The function performs num_iterations independent runs of the simulated annealing algorithm in parallel.

- Each individual simulated annealing run takes O(i * n * (a * p)), and since these runs are distributed across k cores, the total time complexity becomes O(num_iterations * i * n * (a * p) / k).

10. **write_output(output_file, gate_positions, gates, critical_path, critical_delay)**:
    - **Time Complexity**: O(g)
    - Let g be the number of gates.
      The function iterates over all gates to write their positions and dimensions to the output file. Writing the critical path also involves iterating through the path list.

11. **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 + (g + w) * p + i * n * (a * p))

- n = number of lines in the input file
- g = number of gates
- w = number of wires
- p = number of paths
- i = number of iterations in simulated annealing
- n = number of neighbours checked per iteration
- a = number of paths in each iteration

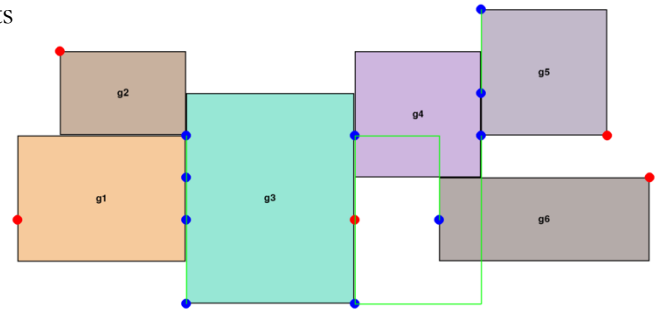
This complexity accounts for all major components of the program.

4 Result and Analysis

4.1 Provided Test Cases

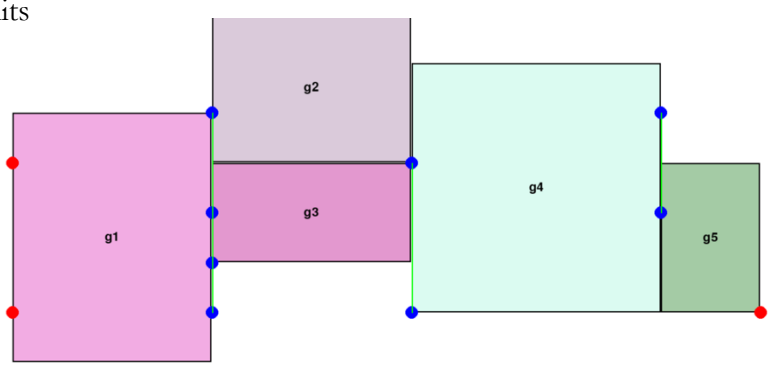
4.1.1 Test Case – 1

- Critical Delay – 26 Units



4.1.2 Test Case – 2

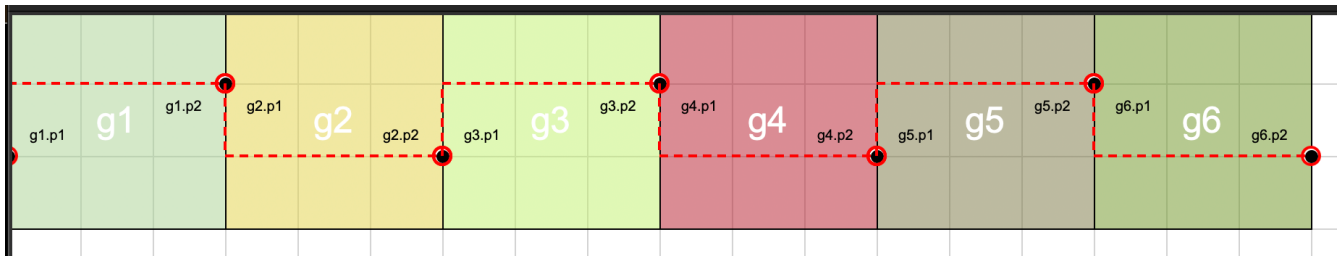
- Critical Delay – 30 Units



4.2 Self Test Cases

4.2.1 Test Case – 1. (5 Gates)

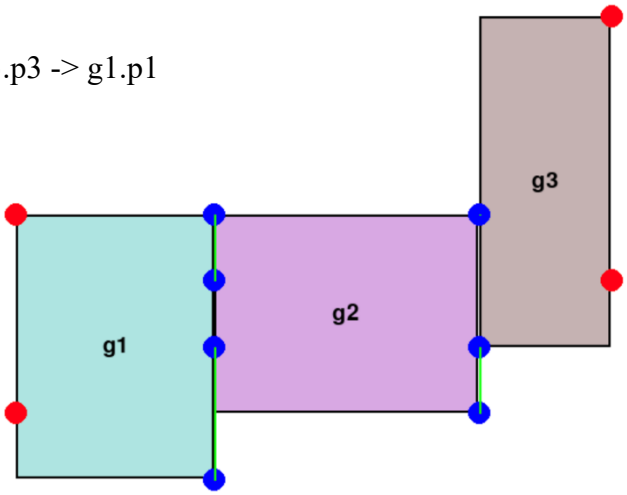
- Optimized Wire Length – 10 units



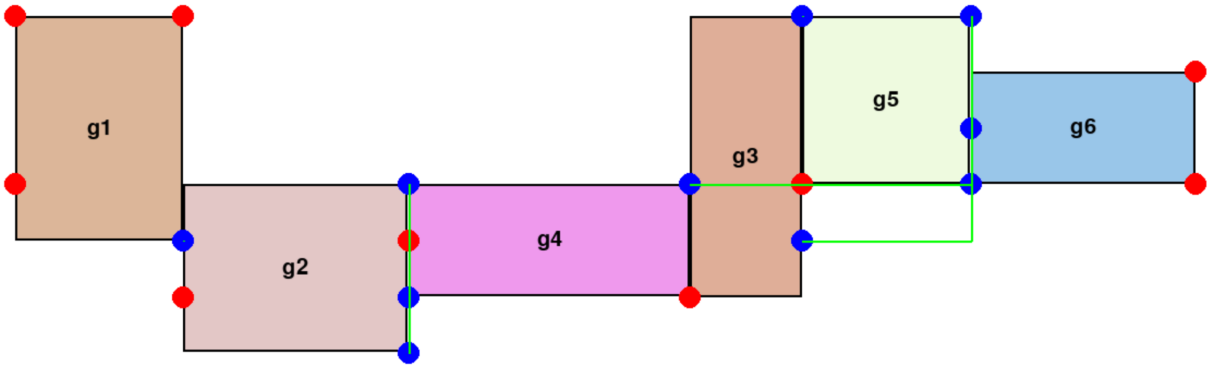
4.2.2 Test Case – 2. (cyclic Gates)

Error: Circuit contains a cyclic path
Cycle: g1.p3 -> g2.p1 -> g2.p3 -> g3.p1 -> g3.p3 -> g1.p1

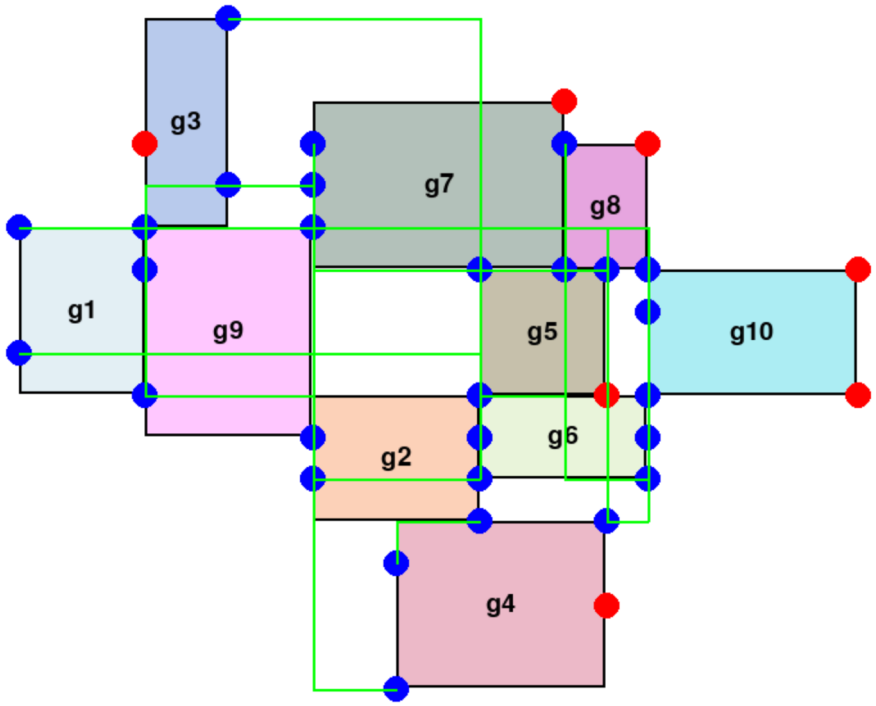
4.2.3 Test Case – 3. (5 Gates)



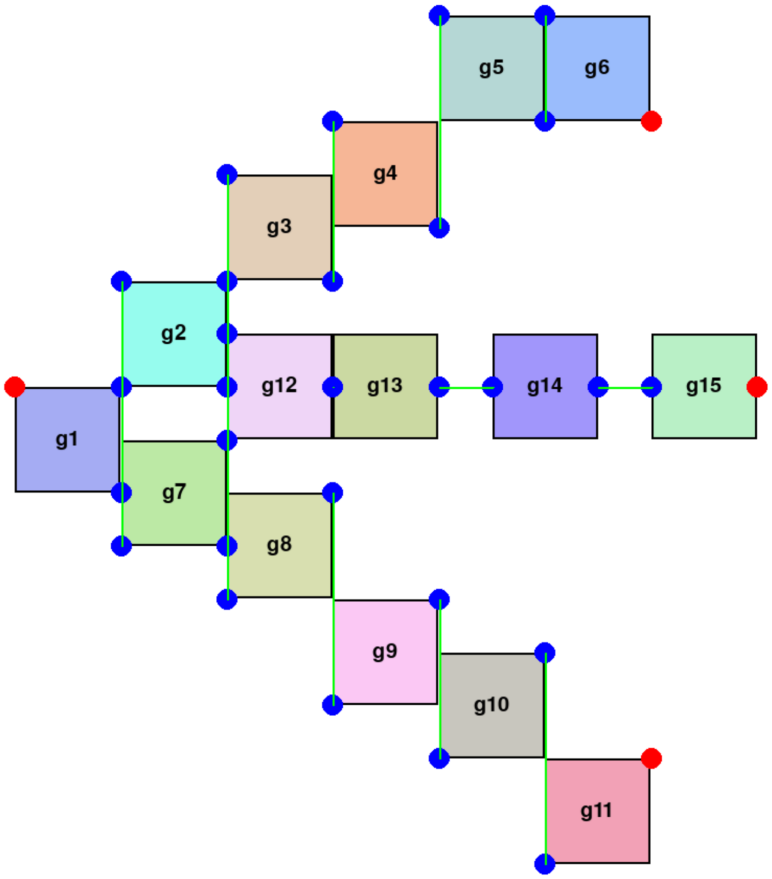
4.2.4 Test Case – 4. (6 Gates)



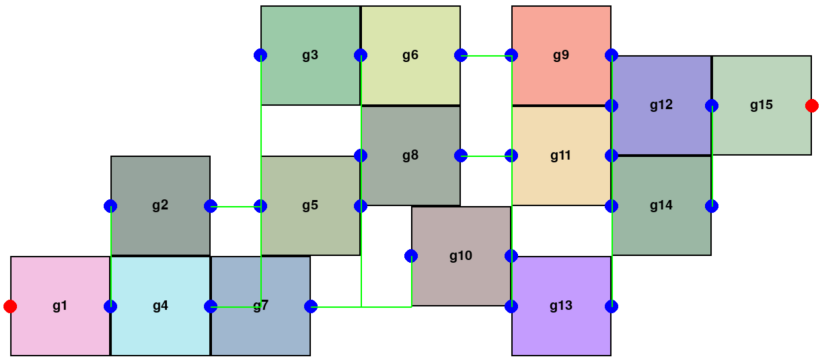
4.2.5 Test Case – 5. (10 Gates)



4.2.6 Test Case – 7. (15 Gates)



4.2.7 Test Case – 8 (15 Gates)



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