SOFTWARE ASSIGNMENT 2

Name	Entry no	Group
Nishant Kumar	2023CS10522	2
Aayushya Sahay	2023CS10543	2

AIM:

Given:

- a set of rectangular logic gates g1, g2...gn
- width and height of each gate gi
- the input and output pin locations (x and y co-ordinates) on the boundary of each gate gi.p1, gi .p2, ..., gi .pm (where gate gi has m pins)
- the pin-level connections between the gates

Write a program to assign locations to all gates in a plane so that:

• no two gates are overlapping

• the sum of estimated wire lengths for all wires in the whole circuit is minimized.

DESIGN DECISION:

For our solution, we have combined two algorithms.

First, we have used our algorithm used in software assignment 1 which is the Skyline Algorithm.

We went for a heuristic method for packing rectangles onto a sheet using a rectilinear skyline representation. This skyline is depicted as a sequence of horizontal line segments that form the contour of vertical bars.

The key properties of the skyline are:

- Consecutive segments have different y-coordinates.
- The x-coordinate of the right endpoint of each segment aligns with the x-coordinate of the left endpoint of the next.
- Initially, the skyline is a single segment representing the bottom of the sheet.

Rectangles are placed one by one, either with their bottom left corner touching a left endpoint or their bottom right corner touching a right endpoint of a skyline segment.

The placement is determined by whether adjacent segments are higher or lower. After placing a rectangle, the skyline is updated in two steps:

- A new segment is created for the top edge of the rectangle, and affected segments are adjusted based on the rectangle's width.
- The algorithm identifies locally lowest segments—those lower than their neighbors—and raises and merges them if no unplaced rectangles can fit. This process continues until all segments are checked.

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

Second, we went for Simulated Annealing algorithm.

Simulated Algorithm explores the solution space by randomly accepting worse solutions with a probability based on temperature, which gradually decreases, ensuring the system settles into an optimal or near-optimal solution.

Steps of the Algorithm:

1. **Initial Solution**: Start with a random or heuristic-based layout of gates.
2. **Cost Function**: The cost function combines:
   - **Wire length**: The total Manhattan distance between connected gates.
   - **Penalties**: Applied for overlapping gates or boundary violations.

$$\text{Cost} = \text{Wire Length} + \text{Penalty}$$

3. **Neighbor Solution**: A new layout is generated by making small changes (moving a gate) to the current solution.
4. **Acceptance Probability**: Even if the new solution is worse, it may still be accepted based on a probability that decreases with temperature:

$$P = \exp(-\Delta\text{Cost} / T)$$

5. **Cooling Schedule**: The temperature decreases over time, allowing for more exploration early on and more refinement later. The cooling schedule is divided into phases:
   - **Stochastic Search**: Early acceptance of worse solutions to explore the solution space.
   - **Local Search**: Focus on improving the current solution.
   - **Uphill Search**: Periodic temperature increases to escape local minima.

6. **Termination**: The process continues until the temperature reaches a minimum value, resulting in an optimized gate layout.

[Reference: https://tilos-ai-institute.github.io/MacroPlacement/CodeElements/SimulatedAnnealing/]

TIME COMPLEXITY ANALYSIS:

SKYLINE ALGORITHM

Finding Position for Rectangle:

- To find a position for placing a rectangle, the algorithm scans the skyline segments. In the worst case, this involves iterating over all segments.
- Time Complexity: O(n), where n is the number of skyline segments.

Updating the Skyline:

- Updating the skyline involves removing old segments and adding new segments, which involves scanning and modifying the list of skyline segments.
- Inserting and removing segments typically involves linear operations with respect to the number of segments.
- Time Complexity: O(n), where n is the number of skyline segments.

Overall, for Placing All Rectangles:

- For each rectangle, the place_rectangle function needs to find its position and update the skyline. If there are m rectangles and each rectangle operation involves scanning and updating the skyline, then:
- Time Complexity: O(m*n), where m is the number of rectangles and n is the number of skyline segments

SIMULATED ANNEALING:

1. Pin and Gate Initialization:

- Creating a Pin and Gate object each takes constant time: O (1).

2. parse_input():

This function reads and processes an input file line by line. The time complexity depends on the number of gates G, pins P, and wires W in the file.

- Gate parsing: For each gate, parsing and creating pin objects takes O(P) where P is the number of pins. If there are G gates, the total complexity for gate parsing is O (G×P)
- Wire parsing: For each wire, processing is O (1), so for W wires, the complexity is O (W)

- Connection creation: For each wire, connecting gates takes O(1). For W wires, it takes O(W)

Thus, the total time complexity of parse_input() is: O (G×P+W)

3. is_overlapping() and check_overlap():

- is_overlapping() checks whether two gates overlap, which is a constant-time operation: O(1)
- check_overlap() iterates over all placed gates to check for overlap. If there are N placed gates, the time complexity is O(N)

4. nishant() / use_wire_estimator():

- Wire Length Calculation: The function nishant() computes the wire length by iterating over all wires W and looking up the coordinates of the gates involved. Looking up a gate takes O (1), so the total complexity for wire length calculation is O (W)

5. place_gate():

- This function attempts to place a gate in several directions (up, down, left, right) and checks for overlap.
    - The overlap check takes O(N), where N is the number of placed gates.
    - Estimating wire length using use_wire_estimator() takes O(W).
    - This process is repeated for each direction (4 directions).

Thus, the total complexity of place_gate() is: O (4×(N+W)) = O (N+W)

6. base_packing():

- Sorting Gates: Sorting the gates based on the number of connections takes O (G * log G).
- Placing Gates: For each gate, place_gate() is called, which takes O (N+W) for each gate. Since each gate is placed once, this takes O(G×(N+W))
- Thus, the time complexity of base_packing() is: O(G * log G+G*(N+W))

7. generate_neighbor():

- This function selects a gate randomly and tries several possible moves. For each move, it checks for overlap and estimates wire length.
    - Overlap check takes O(N)
    - Estimating wire length takes O(W)
    - The number of possible moves is constant (8).

Thus, the time complexity is: O(8×(N+W)) = O(N+W)

8. local_search():

- This function tries small adjustments for each gate to reduce the wire length.
    - For each gate, it checks all 8 possible positions, and for each position, it checks for overlap O(N) and estimates wire length O(W).
    - For G gates, the total complexity is O(G×8×(N+W))=O(G×(N+W))

9. simulated_annealing():

- Main loop: The loop runs until the temperature reaches the final temperature. The number of iterations depends on the cooling schedule and is typically O (log (final_temp/initial_temp))
- Each iteration: In each iteration:
    - Neighbour generation takes O(N+W)
    - Local search takes O(G×(N+W))
    - Cooling takes O(1)

Thus, the total complexity for simulated_annealing() is:
O(log(final_temp/initial_temp)×(G×(N+W)))

10. Bounding Box Calculation:

- Calculating the bounding box takes O(G) as it involves iterating through all placed gates to find the minimum and maximum coordinates.

WIRELENGTH MEASUREMENT (taken from the visualization.py)

1. calculate_pin_coordinates function:

For each gate, it computes the pin positions relative to the gate's position. If there are n gates and each gate has on average p pins:

- The function computes O(p) operations for each gate.
- **Time complexity**: O(np).

2. calculate_all_pin_distances function:

- **Storing pin coordinates**: The function iterates through all pins across all gates. If there are n gates and each gate has p pins, there are np pins in total.
- **Distance matrix computation**: The function calculates the Manhattan distance between each pair of pins, which is O((np)^2) as it involves a double loop for all pin pairs.

- **Time complexity**: O((np)^2).

3. calculate_connection_matrix function:

- A connection matrix is initialized of size (np x np) for np pins.
- For each wire, it updates the matrix by checking which pins are connected.
- **Time complexity**: Building the connection matrix involves iterating over all np pins and processing w wires, leading to O (w + np^2) complexity.
- **Overall:** O (np^2 + w).

4. get_pin_coordinates_in_order function:

Flattens the pin coordinates into a single list.

- **Time complexity**: If there are np pins in total, this takes O(np) time.

5. main function:

- Calls all the above helper functions.
- Iterates over all pins and computes the wire lengths:
  - The function iterates over the connection matrix of size np x np.
  - For each pair of connected pins, it computes the bounding box dimensions.
- **Time complexity of wire length computation**: O(np^2), as it involves looping over all connections.

SUMMARY OF TIME COMPLEXITY:

Simulated Annealing:

1. **Simulated Annealing Loop**: Iterating over temperature cycles and generating candidate solutions—this complexity was mainly **O(n²)** per temperature iteration due to the random selection of neighboring solutions.
2. **Wirelength Calculation**: This part involved computing distances between pairs of pins or gates. The complexity here was **O(n²)** due to the pairwise distance checks and bounding box calculations.

Total Complexity for Simulated Annealing:
Given that T is the number of temperature cycles, and n is the number of gates or pins, the overall complexity was **O (T × n²)**, where T is typically logarithmic in practice due to the cooling schedule.

Wirelength Minimization:

1. **Parsing Input** (parse_input1): **O(n + np + w)**.

2. **Gate Position Parsing** (parse_gate_positions): **O(n)**.
3. **Pin Coordinates Calculation** (calculate_pin_coordinates): **O(np)**.
4. **Distance Matrix Calculation** (calculate_all_pin_distances): **O((np)²)**.
5. **Connection Matrix Calculation** (calculate_connection_matrix): **O((np)² + w)**.
6. **Wirelength Calculation**: **O((np)²)**.

Total Complexity for Second Code:
Given that n is the number of gates, p is the average number of pins per gate, and w is the number of wires, the total time complexity is **O((np)² + w)**.

TESTING STRATEGY:

We have tried changing the parameters and comparing them. We tried to form clusters of gates which are interconnected with each other for applying simulated annealing easily. This is done in the base_case configuration.

[Because simulated annealing is a probabilistic algorithm, it sometimes gives different answer.
But the final answer is near about same each time.]

TEST CASE:

[In our output, we may get negative coordinates of the gates because we were concerned with only wirelength. There is a function for correcting it (shifting the coordinates) in our code, but we have not used it]

Sample tc1: Wirelength = 30

Sample tc2: Wirelength = 27

Sample tc3: Wirelength = 91

Sample tc4: Wirelength = 36

OUR TEST CASES:

TC1: Wirelength = 29

TC2: Wirelength = 39

TC3: Wirelength = 27

TC4: Wirelength = 44071

TC5: Wirelength = 0