COL215 Software Assignment 3: Timing Optimization in Gate Positioning

- Yash Shindekar 2023CS10592

1. Introduction

In this assignment, we will address timing optimisation in gate positioning. Assume that the input circuit is a combinational circuit (no loops). The goal is to place gates in such a way that the critical path delay of the entire circuit is minimized.

2. Design Decisions

1. The primary input refers to an input pin to a gate that is not connected to the output pin of another gate (external input). The primary output refers to an output pin from a gate that is not connected to the input pin of another gate (external output). Path delay is the sum of gate delays and wire delays along that path. The critical path delay for the circuit is the longest path delay from a primary input to a primary output.
   Class Structure:
   - The design uses a class-based approach to encapsulate the properties and behaviors of circuit elements, connections, and connection points. This promotes modularity and reusability.
   - **CircuitElement** class holds the properties of each gate, including its dimensions, connection points, and latency.
   - **Connection** class represents the wiring between different connection points of the gates.
   - **ConnectionPoint** class represents individual connection points on a gate, making it easier to manage connections and their positions.
2. **Connection Management**:
   - The **connect** method in the **CircuitElement** class ensures that incoming connections are counted only when the corresponding connection points are

not already positioned, which helps in tracking the number of connections for critical path calculations.

3. **Finding Entry and Exit Points**:
    - Separate functions (**find_entry_points** and **find_exit_points**) are defined to identify primary input and output points in the circuit. This separation enhances clarity and allows for easy adjustments or extensions in the future.
4. **Path Delay Calculation**:
    - The **compute_critical_delay** function calculates the critical path delay based on the latency of gates and the estimated wire delay. This function employs a breadth-first search algorithm to traverse through the circuit elements, ensuring that all paths from entry to exit points are considered.
5. **Simulated Annealing for Optimization**:
    - The **optimize_placement** function implements a simulated annealing algorithm to find an optimal layout of the circuit elements. This method allows the algorithm to escape local minima by probabilistically accepting worse solutions based on a temperature parameter, which decreases over time.
6. **Non-Overlapping Positioning**:
    - The **find_non_overlapping_position** function ensures that the randomly generated positions for the gates do not overlap, which is crucial for a valid circuit layout. This function attempts to find a valid position within a limited number of attempts to avoid infinite loops.
7. **Dynamic Position Updates**:
    - The **update_absolute_positions** function recalculates the absolute positions of connection points based on their respective circuit element positions. This ensures that all calculations related to connection lengths and delays are accurate and up-to-date.
8. **Input and Output Handling**:
    - The **parse_input** function is designed to read circuit specifications from a file format that is easy to understand and modify. This allows for flexible testing with different circuit configurations.
    - The **generate_output** function formats the results into a specified output format, including bounding box dimensions, critical path, and delays, making it easy to interpret the results.

9. **Randomness in Neighbor Solutions**:

    - The **create_neighbor_solution** function introduces randomness in relocating elements to explore different configurations. This randomness is essential for the simulated annealing process to search the solution space effectively.

10. **Performance Monitoring**:

    - Throughout the optimization process, the algorithm prints the current temperature and cost at each iteration, which aids in monitoring the optimization process and debugging.

3. Code Explanation

Classes

1. CircuitElement Class

- **Purpose: Represents a logic gate or any circuit element.**
- **Attributes:**
  - **identifier: A unique string representing the gate.**
  - **width, height: Dimensions of the gate.**
  - **connection_points: A dictionary storing the connection points on the gate.**
  - **latency: The delay introduced by the gate itself.**
  - **position: The (x, y) coordinates representing the location of the gate in the 2D plane.**
  - **incoming_connections: A count of the number of connections going into the gate.**
- **Methods:**
  - **connect(connection): Increments incoming_connections if the gate is connected via the connection object.**
  - **__str__(): Provides a string representation of the gate for easy debugging.**

2. Connection Class

- **Purpose: Represents a connection or wire between two gates.**
- **Attributes:**
    - **source: The source gate's identifier.**
    - **source_point: The specific connection point on the source gate.**
    - **destination: The destination gate's identifier.**
    - **dest_point: The specific connection point on the destination gate.**

3. ConnectionPoint Class

- **Purpose: Represents a point on a gate where connections (wires) can be made.**
- **Attributes:**
    - **element: The gate to which the connection point belongs.**
    - **point: The specific point identifier (e.g., p1, p2).**
    - **connections: A list of connections associated with this point.**
    - **position: The position of the point relative to the gate.**
    - **absolute_position: The absolute position of the point after placing the gate in the 2D plane.**
- **Methods:**
    - **connect(connection): Adds a Connection object to the connections list, representing a wire to/from this point.**

Functions

a. Helper Functions for Connection Points

- **find_entry_points(connection_points_dict):**
    - **Purpose: Finds gates that are entry points (gates that have no incoming connections).**
    - **Returns: A list of gate identifiers that are entry points.**
    - **How: It iterates through the connection points dictionary and looks for points with no connections and that have a position of (0, 0).**
- **find_exit_points(connection_points_dict):**
    - **Purpose: Finds gates that are exit points (gates that have no outgoing connections).**
    - **Returns: A list of gate identifiers that are exit points.**
    - **How: It checks connection points that are connected to others and that have a non-zero position.**
- **update_absolute_positions(connection_points_dict, elements_dict):**
    - **Purpose: Updates the absolute positions of all connection points based on the gate's position in the plane.**
    - **Returns: Updated dictionaries for connection points and elements.**

b. Functions Related to Delay Calculation

- **measure_connection_length(connection_points_dict, point):**

- **Purpose: Measures the wire length between connection points using the Manhattan distance (difference in x and y coordinates).**
- **Returns: The maximum distance between the x and y coordinates of the connected points.**

- **compute_critical_delay(elements, elements_dict, connection_points_dict, connection_delay):**
  - **Purpose: Computes the critical path delay of the circuit, which is the longest delay from any input to any output.**
  - **Returns: The maximum delay (critical delay) of the circuit.**
  - **How: It performs a breadth-first traversal of the gates using a queue and computes the delay by summing up gate and wire delays. It tracks the maximum delay across all paths.**

c. Functions for Critical Path Tracing

- **trace_critical_path(elements, elements_dict, connection_points_dict, connection_delay):**
  - **Purpose: Traces the critical path in the circuit.**
  - **Returns: A list of connection points representing the critical path (the longest delay path in the circuit).**
  - **How: Similar to compute_critical_delay(), but instead of just computing the delay, it also stores the path that causes the maximum delay.**

d. Functions for Layout Optimization

- **detect_overlap(elements_dict):**

- **Purpose: Detects if any gates overlap in the current placement.**
- **Returns: True if overlap is detected, False otherwise.**
- **How: It checks whether any gate's rectangular area overlaps with another gate's rectangular area based on their positions and dimensions.**

- **create_neighbor_solution(elements_dict, max_x, max_y):**
  - **Purpose: Generates a new candidate placement (neighbor solution) by randomly relocating one gate.**
  - **Returns: A new dictionary of gate placements.**
  - **How: It creates a copy of the current layout and randomly selects one gate to relocate within the plane's bounds.**

- **optimize_placement(elements, elements_dict, connection_points_dict, connection_delay, max_x, max_y, start_temp, cooling_factor, max_iterations):**
  - **Purpose: Optimizes the gate placement using simulated annealing to minimize the critical path delay.**
  - **Returns: The best layout (placement of gates) and the minimized critical delay.**
  - **How: It uses the simulated annealing approach to iteratively improve the placement. The algorithm randomly selects a neighbor layout and decides whether to accept it based on the temperature and cost (delay). The temperature decreases over iterations, reducing the chances of accepting worse layouts as the optimization proceeds.**

e. Utility Functions

- **get_single_entry_point(element, connection_points_dict) & get_single_exit_point(element, connection_points_dict):**
    - **Purpose: These functions find the entry and exit connection points of a gate in the connection_points_dict.**
- **find_non_overlapping_position(elements, width, height, max_x, max_y):**
    - **Purpose: Finds a non-overlapping position for a gate within the plane.**
    - **Returns: The (x, y) coordinates where the gate can be placed without overlap.**
    - **How: It attempts random positions and checks for overlap until a valid position is found, or throws an error if none can be found after multiple attempts.**
- **set_initial_positions(elements, max_x, max_y):**
    - **Purpose: Sets the initial positions of gates in the plane.**
    - **How: Uses find_non_overlapping_position() to place gates in non-overlapping positions randomly.**

f. File Parsing and Output

- **parse_input(filename):**
    - **Purpose: Parses an input file to create elements (gates), connections, and the necessary dictionaries (elements_dict and connection_points_dict).**
    - **Returns: Lists of elements, connections, and dictionaries for connection points and elements, as well as the connection delay.**

- **generate_output(filename, elements, elements_dict, connection_points_dict, connection_delay):**
    - **Purpose: Generates the output file based on the best layout found by the optimization process.**
    - **How: Writes the bounding box dimensions, the critical path, and the delay, as well as the positions of each gate in the output file.**

Main Function

- **main(input_file='input.txt', output_file='output.txt'):**
    - **Purpose: The main function that coordinates the entire process:**
        1. **Parses the input file to get the circuit details.**
        2. **Sets the initial positions of the gates.**
        3. **Runs the optimization algorithm to find the best layout.**
        4. **Writes the best layout and its critical delay to the output file.**

Summary of the Design

- **Data Representation: The circuit is represented by gates (CircuitElement), wires (Connection), and connection points (ConnectionPoint). This design allows for flexible modeling of connections and gate placements.**
- **Optimization Approach: Simulated annealing is used for gate placement optimization. This approach allows for probabilistic acceptance of worse solutions in early iterations to avoid local minima.**
- **Delay Calculation: Critical path delay is calculated using a breadth-first search through the gates, and the longest delay is found using gate and wire delays.**

- **Critical Path Tracing: The code can also trace the critical path through the circuit to identify the longest path contributing to the delay.**

This structure balances flexibility (in circuit modeling) with efficiency (in delay calculation and optimization).

4. Mathematical Formulation

The critical path delay is calculated based on gate and wire delays along all possible paths from input pins to output pins. The objective is to minimize the maximum delay across all paths.

Consider a structure consisting of gates and M wires with length L_m, each connecting a set of pins, gate delay as D_{g_i} and wire delay as D_{wire}. A path $\mathbf{P}$ is a sequence of gates and wires from the Input Pin to the Output Pin and $\mathcal{P}$ denote the set of all possible paths from input pins to output pins.

For a path $\mathbf{P}$, delay is calculated as

$$T_P = \sum_{(g_i, w_m) \in \mathbf{P}} (D_{g_i} + D_{wire} \cdot L_{w_m})$$

Where, g_i and w_m are the gates and wires in Path $\mathbf{P}$. Further, critical path delay is represented as:

$$T_{cp} = \max_{\mathbf{P} \in \mathcal{P}} T_P$$

Finally, the objective function is

$$T_{mincp} = \min T_{cp} = \min(\max_{\mathbf{P} \in \mathcal{P}} \sum_{(g_i, w_m) \epsilon \mathbf{P}} (D_{g_i} + D_{wire} \cdot L_{w_m}))$$

For estimating wire length use **Semi Perimeter Method**.

5.Time Complexity Analysis

The input file specifies the width, height, delay, and pin coordinates of each gate, as well as the wire connections between the pins. The output file contains the bounding box dimensions, the critical path, and the location of each gate.

Functions

a. Helper Functions for Connection Points

1. **find_entry_points(connection_points_dict)**:
    - **Time Complexity**: O(N), where N is the number of connection points.
    - **Reason**: The function iterates over all connection points once to check if they have any incoming connections.
2. **find_exit_points(connection_points_dict)**:

- **Time Complexity**: O(N), where N is the number of connection points.
- **Reason**: Similar to find_entry_points(), it iterates through all connection points to check if they are exit points.

3. **update_absolute_positions(connection_points_dict, elements_dict)**:
   - **Time Complexity**: O(N), where N is the number of connection points.
   - **Reason**: Each connection point's absolute position is updated based on the gate's position, requiring a single pass through the connection points.

b. Functions Related to Delay Calculation

4. **measure_connection_length(connection_points_dict, point)**:
   - **Time Complexity**: O(M), where M is the number of connections for the point.
   - **Reason**: The function computes the length by iterating over all connections of a given point.

5. **compute_critical_delay(elements, elements_dict, connection_points_dict, connection_delay)**:
   - **Time Complexity**: O(V + E), where V is the number of gates (elements) and E is the number of connections (wires).
   - **Reason**: This function performs a breadth-first traversal of the circuit, processing each element (gate) once, and for each gate, it processes its connections.
   - Updating delay for each gate involves checking all connections, which leads to O(V + E).

c. Functions for Critical Path Tracing

6. **trace_critical_path(elements, elements_dict, connection_points_dict, connection_delay)**:
   - **Time Complexity**: O(V + E), where V is the number of gates and E is the number of connections.
   - **Reason**: Similar to compute_critical_delay(), it traverses the gates and their connections, but it also stores the path, so the time complexity remains O(V + E).

d. Functions for Layout Optimization

7. **detect_overlap(elements_dict)**:
    - **Time Complexity**: O(V^2), where V is the number of gates (elements).
    - **Reason**: The function checks all pairs of gates for overlap, leading to a nested loop over all gates.
8. **create_neighbor_solution(elements_dict, max_x, max_y)**:
    - **Time Complexity**: O(V), where V is the number of gates.
    - **Reason**: The function creates a new layout by copying each gate's properties and randomly modifying one gate's position, which requires a single pass over the gates.
9. **optimize_placement(elements, elements_dict, connection_points_dict, connection_delay, max_x, max_y, start_temp, cooling_factor, max_iterations)**:
    - **Time Complexity**: O(max_iterations * (V + E + V^2)), where V is the number of gates and E is the number of connections.
    - **Reason**: The function performs simulated annealing for max_iterations iterations. In each iteration, it:
        - Creates a new layout in O(V) (via create_neighbor_solution()),
        - Detects overlap in O(V^2) (via detect_overlap()),
        - Computes the critical delay in O(V + E) (via compute_critical_delay()).

e. Utility Functions

10. **get_single_entry_point(element, connection_points_dict)**:
    - **Time Complexity**: O(N), where N is the number of connection points.
    - **Reason**: The function iterates over all connection points to find the one associated with the given element.
11. **get_single_exit_point(element, connection_points_dict)**:
    - **Time Complexity**: O(N), where N is the number of connection points.
    - **Reason**: Similar to get_single_entry_point(), it iterates over connection points to find an exit point.
12. **find_non_overlapping_position(elements, width, height, max_x, max_y)**:
    - **Time Complexity**: O(V), where V is the number of gates.

- **Reason**: It tries a maximum of max_attempts to find a valid position, and for each attempt, it checks for overlap with all gates in O(V).

13. **set_initial_positions(elements, max_x, max_y)**:
    - **Time Complexity**: O(V^2), where V is the number of gates.
    - **Reason**: It calls find_non_overlapping_position() for each gate, which checks for overlap with all other gates.

f. File Parsing and Output

14. **parse_input(filename)**:
    - **Time Complexity**: O(L), where L is the number of lines in the input file.
    - **Reason**: The function reads and processes each line of the input file. Each line involves creating a gate or connection, so the complexity is proportional to the size of the input.

15. **generate_output(filename, elements, elements_dict, connection_points_dict, connection_delay)**:
    - **Time Complexity**: O(V + E), where V is the number of gates and E is the number of connections.
    - **Reason**: It computes the critical path and critical delay, updates positions, and writes the results to the file. The critical delay and path tracing are O(V + E), and writing the output involves iterating over all gates and connections.

Main Function

16. **main()**:
    - **Time Complexity**: O(max_iterations * (V + E + V^2)), where V is the number of gates and E is the number of connections.
    - **Reason**: The main function combines all components, including the optimization process (optimize_placement), which dominates the complexity with O(max_iterations * (V + E + V^2)).

Summary Table

Function	Time Complexity
CircuitElement.connect()	O(1)

Function	Time Complexity
ConnectionPoint.connect()	O(1)
find_entry_points()	O(N)
find_exit_points()	O(N)
update_absolute_positions()	O(N)
measure_connection_length()	O(M)
compute_critical_delay()	O(V + E)
trace_critical_path()	O(V + E)
detect_overlap()	O(V^2)
create_neighbor_solution()	O(V)
optimize_placement()	O(max_iterations * (V + E + V^2))
get_single_entry_point()	O(N)
get_single_exit_point()	O(N)
find_non_overlapping_position()	O(V)
set_initial_positions()	O(V^2)
parse_input()	O(L)
generate_output()	O(V + E)
main()	O(max_iterations * (V + E + V^2))

In most cases, the dominating factors in the complexity are the number of gates (V), the number of connections (E), and the number of iterations (max_iterations) in the simulated annealing process.

7. Testing Strategy

1. **Unit Testing**: Each class and function is tested with isolated inputs to ensure correctness.

2. **Integration Testing**: Tests verify that different components work together to form the desired system behavior.
3. **Performance Testing**: The algorithm's efficiency is evaluated as the circuit size scales.
4. **Boundary and Edge Case Testing**: Unusual and extreme cases are tested to ensure the system handles them gracefully.
5. **Randomized Testing**: Randomly generated inputs ensure that the system works for a wide variety of circuits**.**

6. Test Case Format

Sample test cases provide gate dimensions, pin coordinates, and delays. The program needs to compute the optimal gate placement to minimize the critical path delay based on the input data.