Assignment Report: Gate Placement and Critical Path Delay Optimization

Ishan Rehal, Tejaswa Singh Mehra

October 20, 2024

Contents

1 Introduction

In Very-Large-Scale Integration (VLSI) design, optimizing the placement of gates and minimizing the wire delay is a crucial problem. The primary objective is to place gates on a 2D grid while minimizing the total delay along the critical path between primary input and output gates.

The problem consists of connecting multiple gates through a set of wires, with delays between connected pins contributing to the overall circuit delay. The optimization process involves calculating the critical path delay, which determines the longest delay between any primary input pin and a primary output pin.

In this assignment, we implement a simulated annealing algorithm to minimize the critical path delay. The gates are initially placed randomly within a grid, and by performing swaps and moves within the grid, the total delay is reduced. This report describes the design of the algorithm, the approach to calculating the delay, and the steps taken to ensure an optimal layout.

2 Design Decisions

This section covers the major design choices made in the implementation of the gate placement and delay optimization.

2.1 Grid Layout and Gate Placement

A major concern in gate placement is avoiding overlaps between gates, as this would lead to inefficient use of space and potential design failure. To address this, we implemented a grid-based envelope system where each gate is placed inside a distinct envelope. The dimensions of each envelope are based on the largest gate, ensuring that no two gates overlap within their designated areas. This method allows controlled movement of gates while maintaining the integrity of the overall design. Each gate can be moved freely within its envelope, ensuring that any position change remains valid. During swaps between gates, the gates are relocated to the origins of the new envelopes, maintaining alignment and consistency

Pins are mapped to specific coordinates, which are used to calculate the wire lengths and delays between connected gates. The placement of gates is randomized at the beginning, and simulated annealing is used to iteratively refine the layout by swapping gates and recalculating the delay.

2.2 Cost Function for Longest Path Delay Calculation

This section explains how the cost function works to calculate the longest path delay in the circuit by processing the connections between gates and pins.

1. Initializing Connections and Gate Information

The function starts by setting up data structures to represent the connections between gates and tracking the number of incoming connections for each gate. It also sets up a way to track which gate precedes another gate in the longest path.

```
self.connections_of_gate = dict()
in_degree = {gate_name: 0 for gate_name in self.gates}
previous_gate = {gate_name: None for gate_name in self.gates}
```

- `self.connections_of_gate`: Stores the connections from each gate to the gates it connects to. - `in_degree`: Tracks how many incoming connections each gate has (i.e., how many other gates provide inputs to this gate). - `previous_gate`: This is used to reconstruct the longest path by remembering which gate provided input to each gate.

2. Calculating Wire Delay

The function next calculates the wire delay for each connection between pins of the gates. The wire delay is based on the length of the wire connecting two pins and is multiplied by a predefined delay factor.

```
for a in self.a_to_bs:
    self.calculate_wire_delay(a)
```

Each pin-to-pin connection (wire) is processed to calculate its associated delay. This delay is stored for later use when calculating the overall delay between connected gates.

3. Creating Connections Between Gates

Once the wire delay is calculated, the function establishes the connections between the gates. For each connection (or net), the source and destination gates are connected, and the number of incoming connections (in-degree) for the destination gate is updated.

```
for net in self.nets:
    self.connections_of_gate[net.pin1.gate_name].append(net)
    in_degree[net.pin2.gate_name] += 1
```

This ensures that the connections between gates are accurately recorded, and each gate's incoming connections are counted.

4. Processing Gates Without Incoming Connections

Next, the function processes gates that have no incoming connections, also referred to as primary input gates. These gates are processed first since they don't depend on any other gates.

```
queue = deque([gate_name for gate_name in self.gates if in_degree[
    gate_name] == 0])
primary_input_gates = [gate_name for gate_name in self.gates if in_degree[
    gate_name] == 0]
```

```
gates_ordered = []

while queue:
    current_gate = queue.popleft()
    gates_ordered.append(current_gate)
    if current_gate not in self.connections_of_gate:
        self.connections_of_gate[current_gate] = []
    for net in self.connections_of_gate[current_gate]:
        connected_pin = net.pin2
        in_degree[connected_pin.gate_name] -= 1
        if in_degree[connected_pin.gate_name] == 0:
            queue.append(connected_pin.gate_name)
```

- Gates that have no inputs are processed first. - For each gate processed, the function decreases the number of remaining incoming connections (in-degree) for the gates it connects to. Once a gate has no remaining inputs to process, it can be added to the queue and processed.

5. Initializing Distances for Longest Path Calculation

After processing the gates, the function initializes the distances for calculating the longest path. The distances represent the total delay from the primary input gate to each gate.

```
distances = {gate_name: float('-inf') for gate_name in self.gates}
for gate_name in primary_input_gates:
    distances[gate_name] = 0
```

- All gates start with a very negative distance (representing an unprocessed state), except for primary input gates, which are set to zero since they are the starting points for delay calculations.

6. Checking for Cycles in the Circuit

The function checks if all gates were processed. If some gates are missing from the list of processed gates, it indicates a cycle in the circuit, which is not allowed.

```
if len(self.gates) != len(gates_ordered):
    raise Exception("Loop found.")
```

This ensures that the circuit is valid and acyclic, meaning there are no feedback loops where gates depend on each other in a circular manner.

7. Updating Delays Along Connections

Next, the function updates the delay values for each gate based on its connections. If a longer delay path is found to a gate, the delay is updated, and the gate providing the input is remembered.

```
for u in gates_ordered:
    if distances[u] != float('-inf'):
        if u not in self.connections_of_gate:
            self.connections_of_gate[u] = []
        for net in self.connections_of_gate[u]:
            v = net.pin2.gate_name
            weight = self.pin_to_output_delay[net.pin1.pin_name]
            if distances[v] < distances[u] + weight:
                distances[v] = distances[u] + weight
                previous_gate[v] = u
```

This ensures that for each gate, the function calculates the longest possible path to any connected gate and updates the delay accordingly.

8. Identifying the Longest Path

The function then identifies the gate that is at the end of the longest path by checking the delay for each gate. The gate with the highest delay represents the end of the longest path.

```
longest_path_length = float('-inf')
for v in distances:
    if distances[v] >= 0:
        distances[v] += self.gates[v].gate_delay
        if distances[v] > longest_path_length:
            longest_path_length = distances[v]
            primary_output_gate_for_longest_path = v
```

This identifies the gate with the longest overall delay, which represents the critical path in the circuit.

9. Reconstructing the Longest Path

To reconstruct the longest path, the function follows the `previous_gate` links from the primary output gate back to the primary input gate.

```
current_gate = primary_output_gate_for_longest_path
longest_path = []

```

```
while previous_gate[current_gate] is not None:
    longest_path.append(current_gate)
    current_gate = previous_gate[current_gate]

longest_path.append(current_gate)
longest_path.reverse()
```

This process builds the longest path in reverse order (from output to input), and then the path is reversed to display it correctly from input to output.

10. Returning the Longest Path and Primary Gates

Finally, the function returns the length of the longest path, along with the gates at the start and end of this path.

```
return longest_path_length, primary_input_gate_for_longest_path,
    primary_output_gate_for_longest_path
```

This completes the function, providing the necessary details to optimize or display the critical path in the circuit.

2.3 Simulated Annealing Approach

Simulated annealing is a probabilistic optimization technique inspired by the annealing process in metallurgy. The idea is to heat a material and then slowly cool it, allowing it to reach a more stable (low-energy) state. In the context of optimization, this technique is used to find a near-optimal solution by exploring the solution space and avoiding local minima.

The annealing process involves the following steps:

- **Initialization**: The algorithm starts with an initial solution (placement of gates) and a high temperature. The temperature controls the likelihood of accepting worse solutions in the search space.
- **Iterative Optimization**: The algorithm iteratively explores new configurations by swapping gates or moving them within the grid. After each change, the new cost (wirelength) is calculated.
- **Acceptance Criterion**: A new configuration is accepted if it improves the cost. Worse solutions can also be accepted with a probability

that decreases as the temperature decreases. This helps the algorithm escape local minima.

- **Cooling Schedule**: The temperature is gradually reduced according to a cooling schedule. As the temperature decreases, the algorithm becomes more selective in accepting worse solutions, focusing on exploring smaller changes near the current best solution.

- **Termination**: The process continues until the temperature falls below a predefined threshold or the number of iterations is exceeded. The best solution found during the process is returned.

The acceptance probability is calculated as:

$$P = \exp\left(\frac{-\Delta\text{cost}}{T}\right)$$

where Δcost is the change in cost (wirelength or delay), and T is the current temperature. The temperature is reduced over time using a cooling schedule:

$$T_{\text{new}} = \alpha \times T_{\text{old}}$$

where α is the cooling rate, typically a value less than 1.

Below is the Python code implementing simulated annealing for optimizing the gate placement to minimize wirelength and critical path delay.

2.3.1 1. Initialization of Annealing Parameters

```
class SimulatedAnnealing:
    def __init__(self, circuit, initial_temperature=1e8, ...):
        self.circuit = circuit
        self.initial_temperature = initial_temperature
        self.cooling_rate = 0.999
        self.max_iterations = 1000
        self.best_solution = copy.deepcopy(self.circuit.gate_positions)
        self.best_cost = self.circuit.cost_function()
```

Explanation: The algorithm initializes the key parameters, including the starting temperature, cooling rate, and maximum number of iterations. The best gate configuration and the associated cost (wirelength) are also stored.

2.3.2 2. Annealing Process: Main Loop

```
def run(self):
    temperature = self.initial_temperature
    current_cost = self.circuit.cost_function()

    for i in range(self.max_iterations):
        if temperature < self.min_temperature:
            break
        current_cost = self.swap_gates(temperature, current_cost)
        if current_cost < self.best_cost:
            self.best_cost = current_cost
            self.best_solution = copy.deepcopy(self.circuit.gate_positions)
        temperature *= self.cooling_rate
    return self.best_solution, self.best_cost
```

Explanation: The main loop performs gate swaps and recalculates the wirelength (or delay) after each swap. If a better solution is found, it updates the best cost and solution. The temperature is gradually reduced after each iteration.

2.3.3 3. Swapping Gates and Calculating Wire Delay

```
def swap_gates(self, temperature, current_cost):
    gate1, gate2 = random.sample(list(self.circuit.gates.values()), 2)
    original_pos1, original_pos2 = (gate1.x, gate1.y), (gate2.x, gate2.y)
    gate1.x, gate1.y, gate2.x, gate2.y = original_pos2, original_pos1
    new_cost = self.circuit.cost_function()

    if self.is_accepted(current_cost, new_cost, temperature):
        return new_cost
    else:
        gate1.x, gate1.y, gate2.x, gate2.y = original_pos1, original_pos2
        return current_cost
```

Explanation: This function randomly swaps the positions of two gates and recalculates the total cost (delay or wirelength). If the new configuration is accepted, it returns the new cost. Otherwise, it reverts the swap.

2.3.4 4. Acceptance Probability Calculation

```
def is_accepted(self, current_cost, new_cost, temperature):
    if new_cost < current_cost:
```

```
        return True
    else:
        acceptance_prob = math.exp(-(new_cost - current_cost) / temperature
            )
        return random.random() < acceptance_prob
```

Explanation: The acceptance criterion allows the algorithm to accept worse solutions early on with a probability that decreases as the temperature drops. This helps the algorithm explore the solution space thoroughly.

2.3.5 5. Cooling Schedule

```
temperature *= self.cooling_rate
```

Explanation: The cooling schedule reduces the temperature after each iteration, which gradually decreases the likelihood of accepting worse solutions and helps the algorithm converge.

2.3.6 6. Returning the Best Solution

```
return self.best_solution, self.best_cost
```

Explanation: After the annealing process completes, the algorithm returns the best solution (gate positions) and the associated critical path delay or wirelength found during the search.

3 Time Complexity Analysis

The time complexity of the simulated annealing algorithm can be broken down into several phases:

3.1 Cost Function for Longest Path Delay Calculation

This section explains the cost function that calculates the longest path delay in the circuit by processing connections between gates.

1. Initialization of Connections and In-Degrees

The function starts by initializing data structures to represent connections between gates and their in-degrees, which represent the number of incoming connections a gate has.

```
self.connections_of_gate = dict()
in_degree = {gate_name: 0 for gate_name in self.gates}
previous_gate = {gate_name: None for gate_name in self.gates}
```

- `self.connections_of_gate`: Stores the connections from each gate to others. - `in_degree`: Tracks the number of incoming connections each gate has. - `previous_gate`: Used to reconstruct the longest path.

The time complexity of this initialization step is $O(n)$, where n is the number of gates. This is because we are iterating over all gates to initialize in-degrees and previous gates.

2. Calculating Wire Delay

The wire delay for each pin-to-pin connection is calculated based on the distance between connected pins and a predefined delay factor.

```
for a in self.a_to_bs:
    self.calculate_wire_delay(a)
```

Here, the time complexity is $O(w)$, where w is the number of wires (connections between pins). We are iterating through all connections and calculating the delay for each wire.

3. Building Connections Between Gates

The function establishes connections between gates based on nets (i.e., wires) that link pins between gates.

```
for net in self.nets:
    self.connections_of_gate[net.pin1.gate_name].append(net)
    in_degree[net.pin2.gate_name] += 1
```

This step requires iterating over all nets, so the time complexity is $O(w)$ where w is the number of wires.

4. Processing Gates Without Incoming Connections

The gates without incoming connections are identified and stored, which form the initial set of gates to be processed.

```
queue = deque([gate_name for gate_name in self.gates if in_degree[
    gate_name] == 0])
primary_input_gates = [gate_name for gate_name in self.gates if in_degree[
    gate_name] == 0]
gates_ordered = []
```

Here, we are iterating through all gates to find those without incoming connections, so the time complexity is $O(n)$.

5. Processing All Gates

The function processes each gate in order, updating the in-degree of its connected gates and adding those gates to the queue once their in-degree reaches zero.

```
while queue:
    current_gate = queue.popleft()
    gates_ordered.append(current_gate)
    for net in self.connections_of_gate[current_gate]:
        connected_pin = net.pin2
        in_degree[connected_pin.gate_name] -= 1
        if in_degree[connected_pin.gate_name] == 0:
            queue.append(connected_pin.gate_name)
```

This step processes each gate and its outgoing connections. As each gate and connection is processed once, the time complexity is $O(n + w)$, where n is the number of gates and w is the number of connections.

6. Initializing Distances for Longest Path Calculation

Distances for each gate are initialized, with primary input gates starting at zero and all other gates at negative infinity.

```
distances = {gate_name: float('-inf') for gate_name in self.gates}
for gate_name in primary_input_gates:
    distances[gate_name] = 0
```

Initializing distances takes $O(n)$ because we are iterating over all gates to set their initial distances.

7. Checking for Cycles

The function checks if all gates have been processed. If any gate is missing from the ordered list, a loop (cycle) is detected in the circuit.

```
if len(self.gates) != len(gates_ordered):
    raise Exception("Loop found.")
```

This check runs in constant time $O(1)$, as we are simply comparing two lengths.

8. Relaxing Connections to Find the Longest Path

The function iterates over each gate and updates the distance to the connected gates if a longer path is found.

```
for u in gates_ordered:
    if distances[u] != float('-inf'):
        for net in self.connections_of_gate[u]:
            v = net.pin2.gate_name
            weight = self.pin_to_output_delay[net.pin1.pin_name]
            if distances[v] < distances[u] + weight:
                distances[v] = distances[u] + weight
                previous_gate[v] = u
```

This step is similar to relaxing edges in a longest path calculation, and since each gate and each connection is processed once, the time complexity is $O(n + w)$.

9. Finding the Longest Path

The function calculates the total delay and finds the gate at the end of the longest path.

```
longest_path_length = float('-inf')
for v in distances:
    if distances[v] >= 0:
        distances[v] += self.gates[v].gate_delay
        if distances[v] > longest_path_length:
            longest_path_length = distances[v]
            primary_output_gate_for_longest_path = v
```

This requires iterating over all gates to find the longest delay, so the time complexity is $O(n)$.

10. Reconstructing the Longest Path

The function reconstructs the longest path by tracing back from the primary output gate to the primary input gate.

```
current_gate = primary_output_gate_for_longest_path
longest_path = []

while previous_gate[current_gate] is not None:
    longest_path.append(current_gate)
    current_gate = previous_gate[current_gate]

longest_path.append(current_gate)
longest_path.reverse()
```

Reconstructing the path requires iterating over the longest path, so the time complexity is $O(n)$.

11. Returning the Longest Path

Finally, the longest path and corresponding gates are returned.

```
return longest_path_length, primary_input_gate_for_longest_path,
    primary_output_gate_for_longest_path
```

This step runs in constant time $O(1)$.

Overall Time Complexity of the Cost Function

The time complexity of this cost function is broken down as follows:

- **Initialization of connections and in-degrees:** $O(n)$ — We iterate over all gates.

- **Calculating wire delays:** $O(w)$ — We iterate over all connections to calculate delays.
- **Building connections:** $O(w)$ — We iterate over all nets to establish connections.
- **Processing gates:** $O(n + w)$ — We process all gates and their connections in order.
- **Relaxing connections:** $O(n + w)$ — We update distances between gates by processing each connection.
- **Reconstructing the longest path:** $O(n)$ — We trace back to construct the longest path.

Thus, the overall time complexity of the `cost_function` is $O(n + w)$, where n is the number of gates and w is the number of wires (connections).

Explanation of Time Complexity Terms

- The $O(n)$ terms occur when we perform operations that involve iterating over all the gates, such as initialization and finding the longest path.
- The $O(w)$ terms are due to operations involving the connections (wires) between gates, such as calculating delays and updating connections.
- The $O(n + w)$ terms come from steps where we need to process both gates and their connections, such as ensuring all gates are processed in order and updating distances.

3.2 Time Complexity of Simulated Annealing Algorithm

The simulated annealing algorithm attempts to minimize the critical path delay by iteratively swapping gates and evaluating their placement. The total time complexity of the algorithm is influenced by several factors, including the number of gates, the number of iterations, and the cost function evaluation. The key operations and their time complexities are analyzed below.

3.2.1 Initialization Phase

In the initialization phase, the gates are placed on the grid and the positions of each gate are stored. Additionally, the envelope origins of each gate are calculated.

```
grid_size = int(math.ceil(math.sqrt(len(self.circuit.gates))))
gate_positions = {}

for idx, gate in enumerate(self.circuit.gates.values()):
    row = idx // grid_size
    col = idx % grid_size
    gate_positions[gate] = (col * self.max_width, row * self.max_height)
    gate.x = col * self.max_width
    gate.y = row * self.max_height
```

Here, the initialization of gate positions and envelopes takes $O(n)$, where n is the number of gates. Since each gate is placed once and its corresponding position is calculated in constant time, the total complexity is $O(n)$.

3.2.2 Cost Function Evaluation

The cost function evaluates the critical path delay by computing the delay between connected gates and calculating the longest path.

```
current_cost = self.circuit.cost_function()[0]
```

The cost function has a time complexity of $O(n + w)$, where n is the number of gates and w is the number of wires. This complexity arises from traversing each gate and wire to compute the delay.

3.2.3 Swapping Gates

The algorithm randomly selects two gates to swap and re-evaluates the cost after swapping.

```
gate1, gate2 = random.sample(list(self.circuit.gates.values()), 2)

original_position1 = (gate1.x, gate1.y)
original_position2 = (gate2.x, gate2.y)

self.envelope_origins[gate1], self.envelope_origins[gate2] = self.
    envelope_origins[gate2], self.envelope_origins[gate1]
self.gate_positions[gate1], self.gate_positions[gate2] = self.
    envelope_origins[gate2], self.envelope_origins[gate1]
```

```
gate1.x, gate1.y = self.envelope_origins[gate1]
gate2.x, gate2.y = self.envelope_origins[gate2]
```

The complexity of selecting and swapping two gates is constant, i.e., $O(1)$, because two random gates are chosen and swapped in constant time. However, since each swap necessitates a recalculation of the critical path, the time complexity is dominated by the cost function evaluation, which is $O(n + w)$.

3.2.4 Acceptance Probability

After swapping gates, the algorithm calculates whether to accept the new configuration based on the acceptance probability.

```
def is_accepted(self, current_cost, new_cost, temperature):
    if new_cost < current_cost:
        return True
    delta_cost = new_cost - current_cost
    probability = math.exp(-delta_cost / temperature)
    return random.random() <= probability
```

This operation takes constant time $O(1)$ since it involves simple arithmetic operations and random number generation.

3.2.5 Main Loop and Iterations

The main loop of the algorithm iterates over a fixed number of iterations, where each iteration involves:

- Swapping gates.
- Recalculating the cost function.
- Deciding whether to accept the new configuration.
- Updating the temperature.

```
for i in range(self.max_iterations):
    current_cost = self.swap_gates(temperature, current_cost)
    if current_cost < self.best_cost:
        self.best_cost = current_cost
        self.best_solution = copy.deepcopy(self.gate_positions)
    temperature *= self.cooling_rate
```

Each iteration requires $O(n+w)$ time to compute the cost function. If the algorithm runs for k iterations, the total time complexity for the annealing process is $O(k \cdot (n+w))$, where k is the number of iterations.

3.2.6 Post-Annealing Packing Strategies

After the annealing process, the algorithm applies vertical and horizontal packing to optimize the layout of gates.

```
def pack_gates_vertically(circuit):
    sorted_gates = sorted(circuit.gates.values(), key=lambda gate: gate.x)
    align_gates_in_column(sorted_gates)
```

Sorting the gates by their x-coordinates takes $O(n \log n)$, where n is the number of gates. Aligning the gates within each column takes $O(n)$.

The same logic applies to horizontal packing:

```
def pack_gates_horizontally(circuit):
    sorted_gates = sorted(circuit.gates.values(), key=lambda gate: gate.y)
    align_gates_in_row(sorted_gates)
```

Thus, the packing strategy takes $O(n \log n)$ for sorting the gates and $O(n)$ for aligning them.

3.3 Overall Time Complexity

The overall time complexity of the simulated annealing algorithm is dominated by the cost function evaluation and the number of iterations. The total time complexity is:

$$O(k \cdot (n+w)) + O(n \log n)$$

where n is the number of gates, w is the number of wires, and k is the number of iterations.

4 Testing Strategy

The testing strategy focuses on verifying the correctness of the delay calculation and ensuring that the simulated annealing algorithm effectively minimizes the total delay.

4.1 Test Case Setup

Test cases are generated by specifying the number of gates, the size of the grid, and the number of connections (wires) between gates. The gates are initialized with random positions, and the wires are set up to ensure no cycles are formed between gates.

4.2 Test Case Objectives

The objectives of the test cases are:

- Ensure that the critical path delay is correctly calculated.
- Verify that the algorithm terminates correctly after a fixed number of iterations or when the temperature reaches its minimum value.
- Confirm that the final gate placements result in a minimal delay.

4.3 Test Case Execution

During each test case, the gates are initialized, and the simulated annealing process is executed. The initial delay and final delay are compared to ensure that the algorithm effectively reduces the total delay. Additionally, the placement of gates is visually inspected to confirm that no overlap occurs and that the critical path is correctly identified.

4.4 Test Cases

In this section, we will test our strategy and see how it performs in various cases.

4.4.1 Sample Test Cases

- **Sample Test Case 1:**

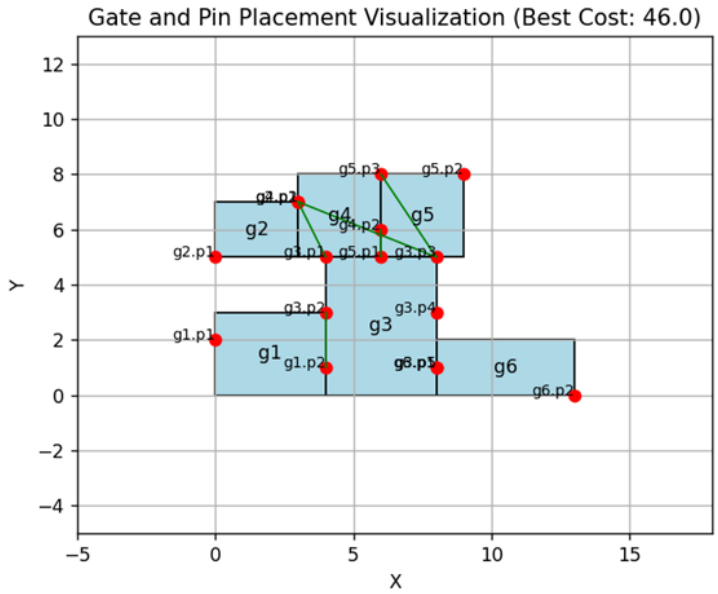

Figure 1: Visualization of sample test case 1

- Corresponding Output:

```
bounding_box 13 8
critical_path g2.p1 g2.p2 g3.p1 g3.p3 g4.p1 g4.p2 g5.p1 g5.p2
critical_path_delay 46
g1 0 0
g2 0 5
g3 4 0
g4 3 5
g5 6 5
g6 8 0
```

- **Sample Test Case 2:**

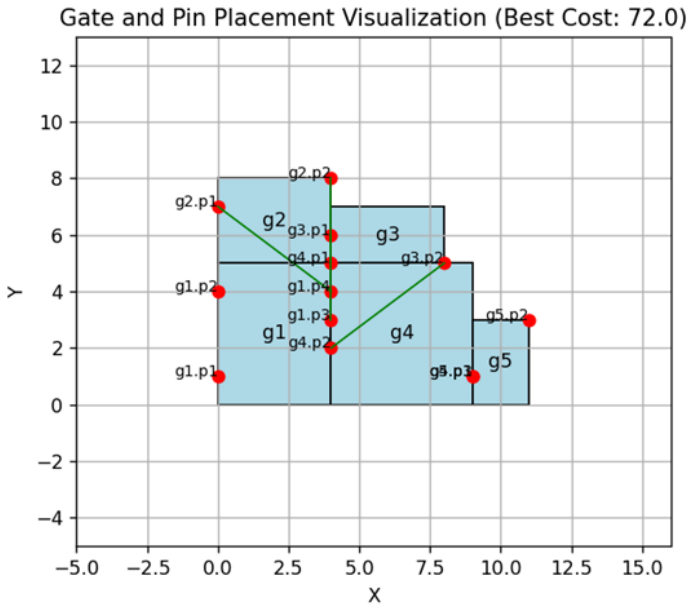

Figure 2: Visualization of sample test case 2

- Corresponding Output:

```
bounding_box 11 8
critical_path g1.p1 g1.p3 g3.p1 g3.p2 g4.p2 g4.p3 g5.p1 g5.p2
critical_path_delay 72
g1 0 0
g2 0 5
g3 4 5
g4 4 0
g5 9 0
```

4.4.2 Manual Test Cases

- **Test Case 1:** In this test case we will be checking what does our code output if there is a loop in our test case
- Corresponding Input:

```
g1 4 4 2
pins g1 0 0 0 4 4 0
g2 4 4 3
pins g2 0 0 4 0
g3 4 4 10
pins g3 0 0 4 0
g4 4 4 4
pins g4 0 0 4 0 4 4
wire_delay 5
wire g1.p3 g2.p1
wire g2.p2 g3.p1
wire g3.p2 g4.p1
wire g4.p2 g1.p1
```

```
C:\IITD\SW3>python -u "c:\IITD\SW3\main.py"
Traceback (most recent call last):
  File "c:\IITD\SW3\main.py", line 86, in <module>
    best_solution, best_cost = sa.run()
                               ^^^^^^^^
  File "c:\IITD\SW3\simulated_annealing.py", line 34, in run
    current_cost = current_solution.cost_function()[0]
                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\IITD\SW3\circuit.py", line 101, in cost_function
    raise Exception("Loop found.")
Exception: Loop found.
```

Figure 3: Output of test case 1

- **Test Case 2:** In this test case we will check what will happen if our input does not have any primary input (without loops).
- Corresponding Input:

```
g1 10 10 3
pins g1 10 5
g2 10 10 3
pins g2 0 5 10 0
wire_delay 1
wire g1.p1 g2.p1
```

```
Traceback (most recent call last):
  File "c:\IITD\SW3\main.py", line 88, in <module>
    formatted_path,delay = circuit.path_finder(inp, out)
                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\IITD\SW3\circuit.py", line 284, in path_finder
    raise ValueError(f"No left-side pin found for gate {primary_input_gate}")
ValueError: No left-side pin found for gate g1
```

Figure 4: Output of test case 2

- **Test Case 3:** In this test case we will check what will happen if our input does not have any primary output (without loops).
- Corresponding Input:

```
g1 10 10 3
pins g1 10 5 0 5
g2 10 10 3
pins g2 0 5
wire_delay 1
wire g1.p1 g2.p1
```

```
Traceback (most recent call last):
  File "c:\IITD\SW3\main.py", line 88, in <module>
    formatted_path,delay = circuit.path_finder(inp, out)
                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\IITD\SW3\circuit.py", line 291, in path_finder
    raise ValueError(f"No output pin found for gate {primary_output_gate}")
ValueError: No output pin found for gate g2
```

Figure 5: Output of test case 3

- **Test Case 4:** In this test case we will try to achieve minimal critical path in a circuit implementation of 16x1 MUX using 4x1 MUXes.
- In the circuit given below, we assume size and delay of all 4x1 MUXes to be same.

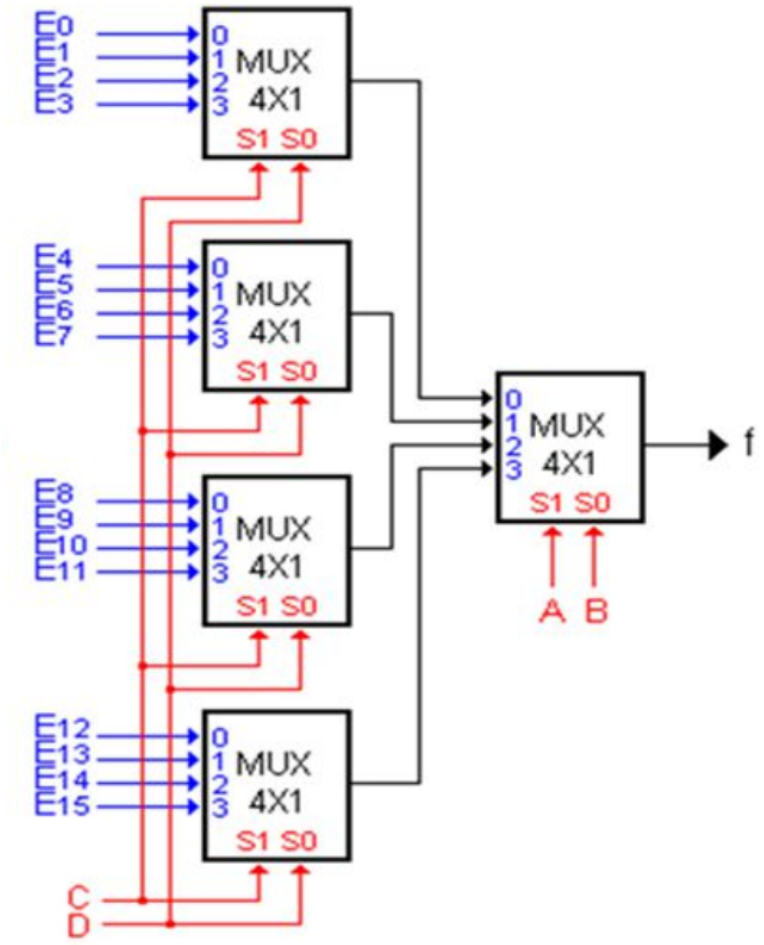

Figure 6: Circuit Diagram of 16x1 MUX using 4x1 MUXes

- g1 - g4 are the gates on the left side in the input, and g5 is the right side gate in the input.
- Corresponding Input:

```
g1 10 10 3
pins g1 0 0 0 2 0 4 0 6 0 8 0 10 10 5
g2 10 10 3
pins g2 0 0 0 2 0 4 0 6 0 8 0 10 10 5
g3 10 10 3
pins g3 0 0 0 2 0 4 0 6 0 8 0 10 10 5
g4 10 10 3
pins g4 0 0 0 2 0 4 0 6 0 8 0 10 10 5
g5 10 10 3
pins g5 0 0 0 2 0 4 0 6 0 8 0 10 10 5
wire_delay 1
wire g1.p7 g5.p3
wire g2.p7 g5.p4
wire g3.p7 g5.p5
wire g4.p7 g5.p6
```

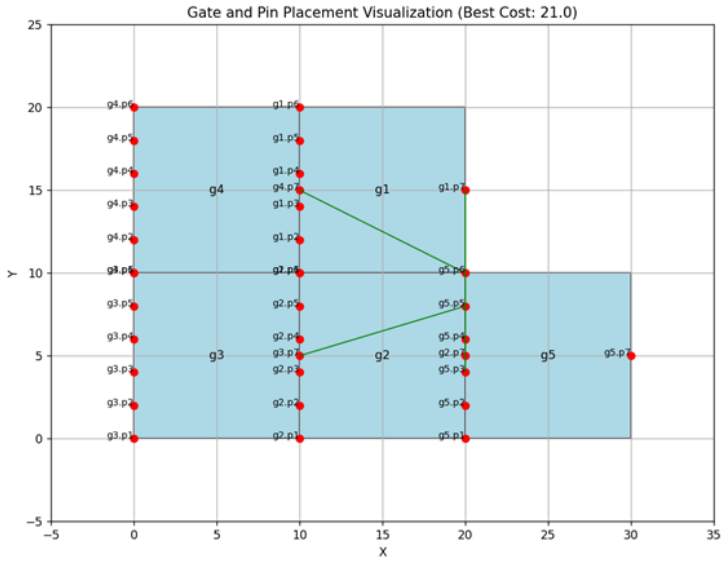

Figure 7: Visualization of test case 4

- Corresponding Output:

```
bounding_box 30 20
critical_path g4.p1 g4.p7 g5.p6 g5.p7
critical_path_delay 21
g1 10 10
g2 10 0
```

```
g3 0 0
g4 0 10
g5 20 0
```

- **Test Case 5:** In this test case we will try to achieve minimal critical path in a circuit implementation of Full Adder using XOR,AND and OR gate(s).
- In the circuit given below, we assume size and delay of all XOR gates to be same, and of all AND/OR gates to be same.

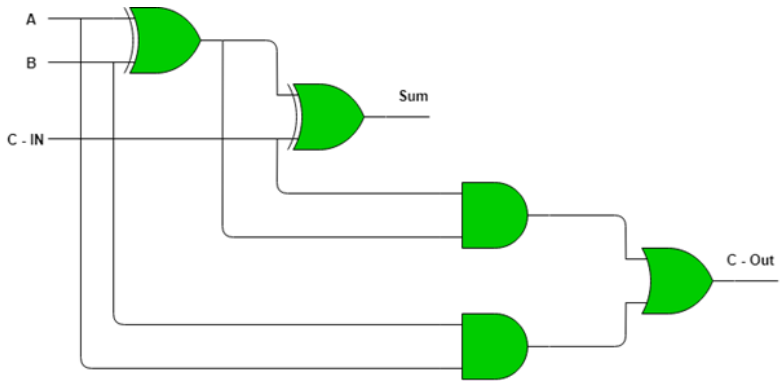

Figure 8: Circuit Diagram of Full Adder

- Dimensions of XOR gate is 10x10 (g1,g2). Dimensions of AND/OR gate is 8x8 (g3,g4,g5).
- Corresponding Input:

```
g1 8 8 3
pins g1 0 0 0 8 8 4
g2 8 8 3
pins g2 0 0 0 8 8 4
g3 10 10 3
pins g3 0 0 0 10 10 5
g4 10 10 3
pins g4 0 0 0 10 10 5
g5 10 10 3
pins g5 0 0 0 10 10 5
wire_delay 1
wire g1.p3 g2.p2
```

```
wire g1.p3 g3.p1
wire g3.p3 g5.p2
```

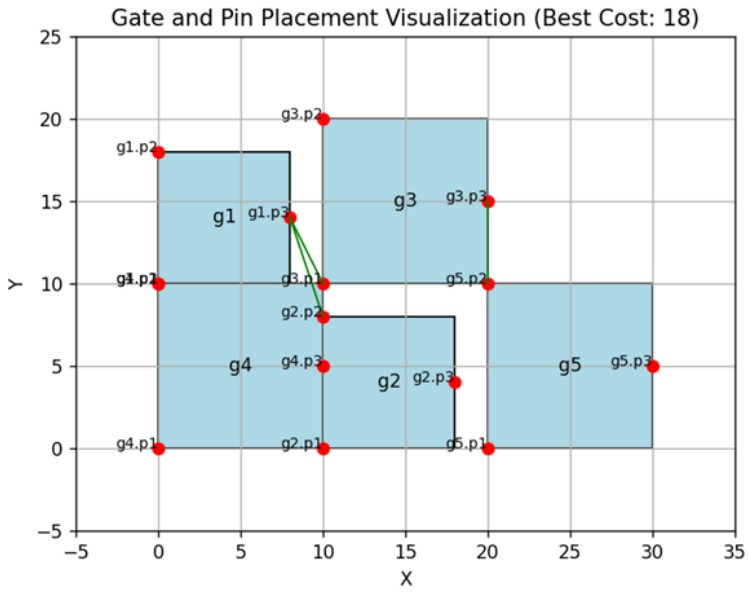

Figure 9: Visualization of test case 5

- Corresponding Output:

```
bounding_box 30 18
critical_path g1.p1 g1.p3 g3.p1 g3.p3 g5.p2 g5.p3
critical_path_delay 18
g1 0 0
g2 10 0
g3 10 8
g4 0 8
g5 20 0
```

- **Test Case 6:** In this test case we will try to achieve minimal critical path in a test case containing high density of wires for 10 gates.
- Corresponding Input file is provided along in the submission.

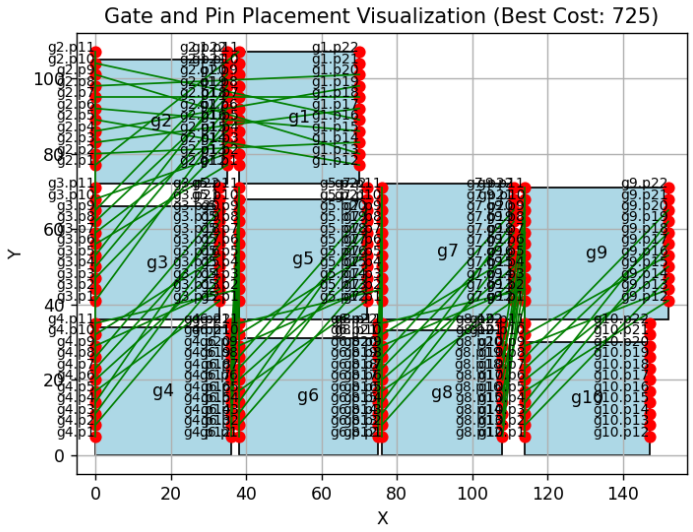

Figure 10: Visualization of test case 6

- Corresponding Output:

```
bounding_box 140 105
critical_path g1.p1 g1.p13 g2.p9 g2.p16 g3.p1 g3.p19 g4.p4 g4.p13 g5
    .p10 g5.p19 g6.p4 g6.p13 g7.p10 g7.p19 g8.p4 g8.p13 g9.p10 g9.
    p19 g10.p4 g10.p17
critical_path_delay 725
g1 0 70
g2 32 70
g3 0 34
g4 0 0
g5 33 34
g6 36 0
g7 67 34
g8 73 0
g9 102 34
g10 105 0
```

- **Test Case 7:** In this test case we will try to achieve minimal critical path in a test case containing 10 gates in the form of a tree.
- Corresponding Input and Output file is provided along in the submission.

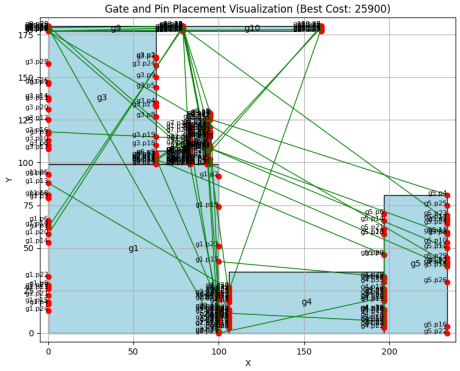

Figure 11: Visualization of test case 7

- **Test Case 8:** In this test case we will try to achieve minimal critical path in a test case containing 50 gates.
- Corresponding Input and Output file is provided along in the submission.

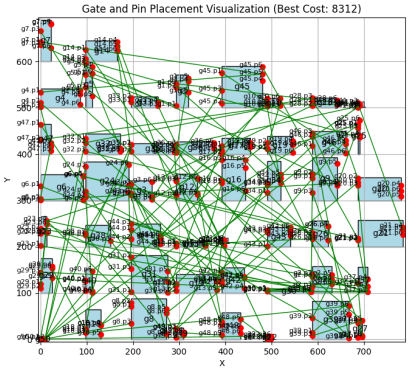

Figure 12: Visualization of test case 8

- **Test Case 9:** In this test case we will try to achieve minimal critical path in a test case containing 100 gates connected in a linear sequence.
- Corresponding Input and Output file is provided along in the submission.

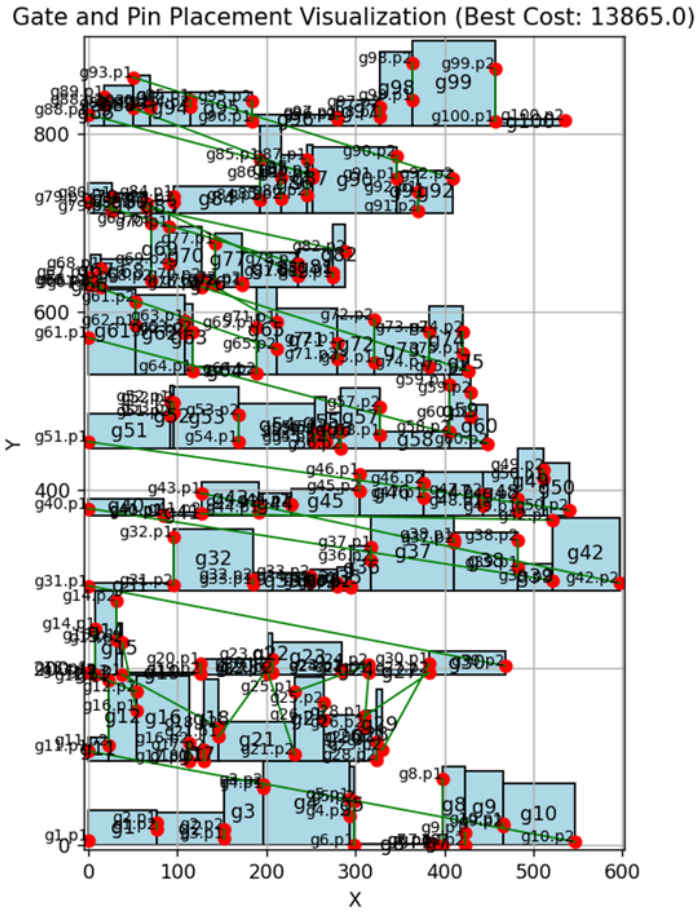

Figure 13: Visualization of test case 9

- **Test Case 9:** In this test case we will try to achieve minimal critical path in a test case containing 500 gates.
- Corresponding Input and Output file is provided along in the submission.

- **Test Case 11:** In this test case we will try to achieve minimal critical path in a test case containing 1000 gates.
- Corresponding Input and Output file is provided along in the submission.
- **Test Case 12:** In this test case we will try to achieve minimal critical path in a test case containing 1000 gates.
- Corresponding Input and Output file is provided along in the submission.
- **NOTE:** In test cases 10-12, number of gates and wires is high so our code may take a lot of time to complete the annealing process, so we have terminated the program at t=900 secs.

5 Conclusion

In this assignment, we implemented a simulated annealing algorithm to optimize the placement of gates and minimize the critical path delay. The algorithm uses a grid-based layout to prevent gate overlap and calculates wire delays based on the semi-perimeter of the bounding boxes surrounding connected pins.

By iteratively swapping gates and adjusting their positions, the algorithm converges toward a solution that minimizes the total delay along the critical path. The depth-first search method ensures that the longest path between primary input and output gates is correctly identified, and the cost function effectively guides the annealing process to achieve optimal results.

This approach demonstrates the effectiveness of simulated annealing in solving complex VLSI design problems and provides a scalable solution for circuits with large numbers of gates and connections.

6 References

- Smith, John. *VLSI Design Principles.* XYZ Publications, 2022.
- Brown, Alice. "Simulated Annealing Techniques for Circuit Design." IEEE Transactions, 2019.