COL215 Software Assignment 3: Timing Optimisation in gate positioning

Samyak Sanghvi 2023CS10807
Vansh Ramani 2023CS50804

October 20, 2024

Contents

1 Introduction

In this assignment, we will address timing optimisation in gate positioning. Assume that the input circuit given to us in Assignment 3 is a combinational circuit (no loops). Note: Alot of technical details are in code with comments. Kindly refer to that for any implementations technicalities.

2 Critical Path

- **Primary input** refers to an input pin to a gate that is not connected to the output pin of some other gate (it is an external input).
- **Primary output** refers to an output pin from a gate that is not connected to the input pin of another gate (it is an external output). This is a simplification for our assignment – in reality, intermediate signals could also be primary outputs.
- **Path delay** from a point to another point is the sum of the gate delays and wire delays along that path.
  - Gate delay (units: nanoseconds) is specified in the input file separately for every gate.
  - Assume that wire delay is proportional to the estimated length of the wire, and the delay per unit length is specified in the input file.
- If a wire connects a source with multiple destinations, then the signal arrives at all destinations simultaneously.
- The **critical path delay** for a circuit is defined as the longest path delay from any primary input to primary output of the circuit.

3 Problem Statement

Given:

- A set of rectangular logic gates $g_1, g_2, \ldots, g_n$.
- Width and height of each gate g_i.
- The input and output pin locations (x, y) coordinates on the boundary of each gate $g_i.p_1, g_i.p_2, \ldots, g_i.p_m$ (where gate g_i has m pins).
- Gate delay $D_{g_1}, D_{g_2}, \ldots, D_{g_n}$.
- Wire delay per unit length D_{wire}.
- 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 critical path of the whole circuit is minimized.

3.1 Notes

- Assume that the gates cannot be re-oriented (rotated, etc.) in any way.
- Assume that all wiring is horizontal and vertical.
- A possible estimate for the wire length for a set of connected pins uses the semi-perimeter method: form a rectangular bounding box of all the pin locations; the estimated wire length is half the perimeter of this rectangle.

4 Mathematical Formulation

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 P is a sequence of gates and wires from the input pin to the output pin, and P denotes the set of all possible paths from input pins to output pins.

For a path P, the delay is calculated as

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

where g_i and w_m are the gates and wires in path P. Further, the critical path delay is represented as:

$$T_{\text{cp}} = \max_{p \in P} T_P$$

Finally, the objective function is

$$T_{\text{mincp}} = \min T_{\text{cp}} = \min \left(\max_{p \in P} \sum_{(g_i, w_m) \in P} (D_{g_i} + D_{\text{wire}} \cdot L_{w_m}) \right)$$

For estimating wire length, use the semi-perimeter method.

5 Design Decisions

The gates will be arranged such that the overlap constraint is satisfied, and the wiring length is minimized using the semi-perimeter method.

The following logic was used to optimize and find a good solution for the problem:

5.1 Grid Swapping Method

Here, we attempt to optimize the placement of components within a grid to minimize the critical delay required to connect them.

1. **Initial Setup:** It calculates the critical delay and determines a grid size based on the number of components. Each component is then placed in the grid with specific spacing.

2. **Optimization Process:** The function enters a loop where it randomly selects two components and swaps their positions. After each swap, the new critical delay is calculated. If the new configuration reduces the wire length, it keeps the new positions; otherwise, it reverts to the original positions.

3. **Result:** After a set number of iterations, the function updates the components' positions to the best-found configuration that minimizes the wire length and prints the final wire length.

This approach ensures that the gates do not overlap and optimally utilizes the available space, facilitating efficient connections while minimizing potential Critical delay. The overall layout promotes a compact arrangement, which can be further refined if needed.

5.2 Stochastic Relaxation Optimizer for Packing the Gates

Stochastic Relaxation Optimizer (SR) is an optimization technique inspired by the cooling process in metallurgy. The goal is to find an optimal or near-optimal solution by exploring the solution space. It initially allows sub-optimal solutions to avoid local minima but gradually becomes more selective as the "temperature" decreases.

- **Large Search Space**: SR explores various placements of circuit components effectively by perturbing gate positions at random.
- **Avoids Local Minima**: By accepting worse solutions probabilistically, SR avoids getting stuck in sub-optimal configurations.
- **Exploration and Exploitation**: It balances the broad search in the beginning with detailed refinement at lower temperatures.

We started with a initial tightly packed placement, taken from the bin packing algorithm of Software Assignment 1 and then proceeded to variate in a small range and then iterate over the other gates - but this didnt work for large cases.

Finally in the end, we place each of the gates discretely on the 2-D plane, to avoid overlaps in a grid pattern.

6 Implementation Details

6.1 Input Format

The input contains:

- For each gate: the width, height, delay and pin coordinates relative to the bottom-left corner of the gate.
- Wire delay per unit length
- The wire connections between gates, specifying which pins are connected.

6.2 Output Format

The output includes:

- The bounding box of the gate arrangement.
- The Critical Delay calculated.
- The Critical Path
- The positions of each gate, specified as the coordinates of the bottom-left corner.

6.3 Wire Length Approximation

The wire length between connected pins is approximated using the semi-perimeter of the bounding box that encloses all connected pins in a pin cluster. The following steps outline how the wire length is computed:

- For each pin p_i, access its Pin Cluster.
- If pin p_i is connected to one or more pins, calculate a bounding box that encloses all connected pins.
- The wire length for these connected pins is approximated by the semi-perimeter of the bounding box.

Bounding Box Calculation

Given the coordinates of the connected pins, the bounding box is defined by the minimum and maximum x-coordinates and y-coordinates of the pins. Let the coordinates of pin p_i be (x_i, y_i), and let S_i be the set of all pins connected to p_i.

The bounding box is determined by:

$$x_{\min} = \min_{p_j \in S_i} \{x_j\}, \quad x_{\max} = \max_{p_j \in S_i} \{x_j\}$$

$$y_{\min} = \min_{p_j \in S_i} \{y_j\}, \quad y_{\max} = \max_{p_j \in S_i} \{y_j\}$$

Semi-Perimeter and Wire Length Approximation

The wire length for pin p_i and its connected pins is approximated by the semi-perimeter of the bounding box. The semi-perimeter is calculated as:

$$\text{Semi-perimeter} = (x_{\max} - x_{\min}) + (y_{\max} - y_{\min})$$

This semi-perimeter represents the Manhattan distance required to connect the pins inside the bounding box.

1. **Adjacency List Construction**: The first step is to convert the list of wires between components into an adjacency list, which is a data structure that maps each component to its connected neighbors.

2. **DFS Function**: The DFS function is used to explore all components connected to a given component. When called on a starting component, the function marks it as visited, adds it to the current cluster, and recursively explores all its neighbors. Neighbors are other components that are connected to the starting component via wires. This process continues until all components in the current cluster are visited.

3. **Main Loop**: The main loop iterates over all components. If a component hasn't been visited yet, it means it's the starting point of a new cluster. DFS is called on this component to discover the entire cluster, and once the DFS is finished, the cluster is added to the list of clusters.

4. **Cluster Creation**: Each DFS call explores all components reachable from the starting component. After the DFS completes for a given component, the set of visited components forms a cluster. This set is added to the `clusters` list.

6.4 Pin Clustering

Algorithm 1 Pin Clustering

```
Input: List of wires connecting pins
Output: Pin clusters
Initialize an empty dictionary to store clusters
for each wire connecting pin1 to pin2 do
    if pin1 is not in the cluster dictionary then
        Create a new cluster and assign pin1 to it
    end if
    Add pin2 to the cluster of pin1
end for
for each cluster do
    for each pin in the cluster do
        Update the pin's cluster, and assign its prev and nxt pointers
    end for
end for
Return the list of clusters
```

1. **Group Pins by Wire Connections:** The algorithm processes each wire in the circuit and identifies the two connected pins—typically one output pin from a gate and one or more input pins. Each wire defines a relationship between these pins. Based on the wire connections, the algorithm groups connected pins into clusters, where each cluster represents a set of pins that are interconnected by wires.

2. **Update Clusters:** As the algorithm processes more wires, it updates the clusters dynamically. When new connections are identified, the relevant pins are added to the same cluster, ensuring that all pins in a cluster are part of the same connected component of the circuit. This process continues until all pins and wires have been processed and all clusters are fully formed.

6.5 Cycle Detection

Algorithm 2 Cycle Detection in Circuit

```
Input: List of circuit components and their pins
Output: True if a cycle is detected, False otherwise
Initialize visited set and recursion stack
for each gate in the circuit do
    if gate is not visited then
        Perform DFS on the gate
        if a cycle is detected during DFS then
            Return True
        end if
    end if
end for
Return False if no cycle is found
```

1. **Depth-First Search (DFS) on the Gate Graph:** In this step, the algorithm represents the circuit as a directed graph, where the gates are nodes and the wires between input and output pins are directed edges. To detect cycles, the algorithm performs a depth-first search (DFS) starting from each gate. The DFS explores the circuit by following the directed edges, marking gates as visited during the traversal.

2. **Detect Back Edges:** As the DFS progresses, the algorithm checks for back edges, which indicate edges that point to gates that have already been visited and are still active in the current DFS call stack. The presence of a back edge signals the existence of a cycle in the circuit. If no back edges are detected, the circuit is confirmed to be acyclic.

6.6 Critical Path Calculation

Algorithm 3 Critical Path Finding

```
Input: Circuit components, wires, and wire delay
Output: Critical path and its total delay
Find primary input and output gates
Initialize empty dictionaries for gate and pin delays
for each gate in primary output gates do
    for each input pin of the gate do
        Recursively compute the pin delay using the function calculate_pin_delay
        Store the maximum delay observed
    end for
end for
Trace back from the pin with the maximum delay to find the critical path
Return the critical path and total delay
```

1. **Identify Primary Inputs and Outputs:** The algorithm begins by traversing through all gates in the circuit to inspect each gate's input and output pins. A pin

is identified as a primary input if it does not have any predecessor pins connected to it. Similarly, a pin is identified as a primary output if it does not have any successor pins connected to it.

2. **Recursively Calculate Delays:** Once the primary inputs and outputs are identified, the algorithm recursively calculates the delays for each output pin. This is done by computing the delay of each output pin based on the delays of its corresponding input pins. To avoid recalculating delays multiple times, memoization is used to store the previously computed values.

3. **Determine the Critical Path:**

   After all delays are computed, the algorithm traces back through the circuit to identify the critical path. This critical path is the one with the maximum delay, representing the longest path from a primary input to a primary output in terms of signal propagation time.

6.7 Stochastic Relaxation Optimizer

Algorithm 4 Stochastic Relaxation Optimizer for Circuit Placement

Input: Components C, Wires W, Iterations K, Cooling Factor α
Output: Optimized positions of components, total wire length
Initialize component positions using a bin-packing algorithm.
Calculate initial total wire length L_0
Set initial temperature $T_{\max}$
for $i = 1$ to K **do**
    Select a random component $c \in C$
    Randomly move component c to a new position (x', y')
    Calculate the new wire length L_{new} based on the new position
    if New configuration is better ($L_{\text{new}} < L_{\text{current}}$) **then**
        Accept the new configuration
    else
        Accept with probability $P = \exp\left(\frac{L_{\text{current}} - L_{\text{new}}}{T_i}\right)$
    end if
    Update temperature $T_{i+1} = \alpha \times T_i$
end for
Return the final positions of the components and the optimized wire length

Stochastic Relaxation Optimizer is implemented to optimize the placement of circuit components in such a way that the total wire length between them is minimized. The key steps of the algorithm are as follows:

1. **Initialization**:

   - The algorithm starts with an initial placement via grid swapping method and a high temperature T_0.
   - The initial wire length, representing the current energy of the system, is calculated.

2. **Component Relocation**:
    - A component is randomly selected and its new position is computed by placing it at a random spot within the maximum possible boundaries.
    - The wire length is recalculated based on this new position.
    - If there is a collision, they we skip.
3. **Acceptance Criteria**:
    - If the new configuration has a lower wire length (lower energy), the new position is accepted.
    - If the new configuration has a higher wire length, it is accepted with a probability $P = \exp\left(-\frac{\Delta E}{T}\right)$, where ΔE is the increase in wire length and T is the current temperature.
4. **Cooling Schedule**:
    - After a certain number of iterations, the temperature is reduced according to a cooling factor α such that $T_{\text{new}} = \alpha \cdot T$.
    - This gradual cooling reduces the likelihood of accepting worse solutions over time, allowing the algorithm to focus on refining the best solutions.
5. **Stopping Condition**:
    - The process repeats for a fixed number of iterations, when a fixed number of iterations of annealing has occurred, it exits the loop.
6. **Final Solution**:
    - The algorithm outputs the best layout of components found during the optimization process, with the corresponding minimized wire length.

7 Test Cases

7.1 Given test Cases

We get good results on given test cases. 35 and 33 as delays for given testcases with solutions attached in submission.

We can observe that on all test cases our algorithm given enough iterations provides better results for each test case, than the sample output.

7.2 Generated Test Cases

We considered and analysed our code over multiple test cases and proved its robustness in multiple cases.
When there are multiple gate clusters our method performs very well.
Our method doesn't perform very well with large sequential chains as it cannot find good random positions.

The following type of test cases were generated and tested for: Below are cases, solutions and test case generator is attached.

1. **Random:** Generates a random directed acyclic graph (DAG) with no specific structure. (Delay = 1054)

2. **Sparse:** Generates a graph with sparse connections, ensuring very few edges between gates. (Delay = 253)

3. **Dense:** Generates a graph with dense connections, where many gates are interconnected. (Delay = 3538)

4. **Chain:** Produces a sequential chain of gates, where each gate connects to the next. (Delay = 1030)

5. **Star:** Creates a star-shaped graph where a central gate connects to all other gates. (Delay = 11064)

Example Test case of each are attached in submission.
The method used to generate all of these is attached in *generate_tc.py*. Kindly refer to that for specifics.

8 Time Complexity Analysis

With the inclusion of the bin-packing algorithm, the overall time complexity is divided into two parts:

- **Finding Critical Path**: The time complexity of finding the critical path is $O(n \times p_{\text{in}} \times p_{\text{out}})$, where n is the number of gates, p_{in} is the total number of input pins, and p_{out} is the total number of output pins. Each gate and its associated pins are processed once during delay calculations and path determination.

- **Grid Swapping**: Grid swapping runs for k iterations, thus the complexity is $O(k \times (n \times p_{\text{in}} \times p_{\text{out}}))$, where n is the number of components.

- **Simulated Annealing**: For each iteration of simulated annealing, recalculating the critical delay takes $O(n \times p_{\text{in}} \times p_{\text{out}})$, and checking for collisions takes $O(n)$. Since the algorithm runs for K iterations, the overall complexity is $O(K \times (n \times p_{\text{in}} \times p_{\text{out}}))$.

- **Complexity of Forming Pin Clusters**: The time complexity of forming pin clusters is $O(m)$, where m is the number of wires. Since m (the number of wires) is less than $p_{\text{in}} \times p_{\text{out}}$, it can often be neglected in overall complexity calculations.

- **Detecting Cycles**: The time complexity of detecting cycles is $O(n + m)$, where n is the number of gates (nodes) and m is the number of wires (edges). A depth-first search (DFS) is performed to explore the graph and detect cycles based on back edges.

Thus, the overall complexity is $O(K \times (n \times p_{\text{in}} \times p_{\text{out}}))$.
Where:

- n: number of components,

- m: number of wires,

- p_{in}: number of input pins,
- p_{out}: number of output pins,
- K: total number of iterations.

9 Conclusion

The Delay-aware Gate Positioning problem was approached using swapping grid search and Stochastic Relaxation Optimization for gate placement. This method effectively minimized critical delay while avoiding gate overlaps. The algorithm performed well on various test cases. Overall, this approach provides a robust solution to the circuit placement problem, balancing between optimization quality and computational efficiency.