COL 215-SOFTWARE ASSIGNMENT -2

Wiring-aware Gate Positioning

Introduction

- **Purpose** Extending on the gate packing assignment, the gates will contain pins as shown in below figure. Objective is to minimize the total wire length of the circuit.
- **Problem Statement:** write a program so that sum of estimated wire lengths for all wires in the whole circuit is minimised.:
- • no two gates are overlapping
- • 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

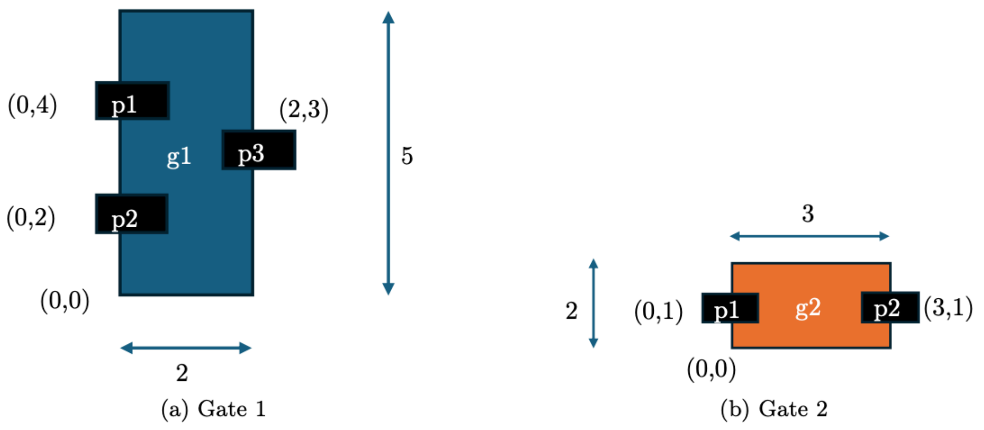
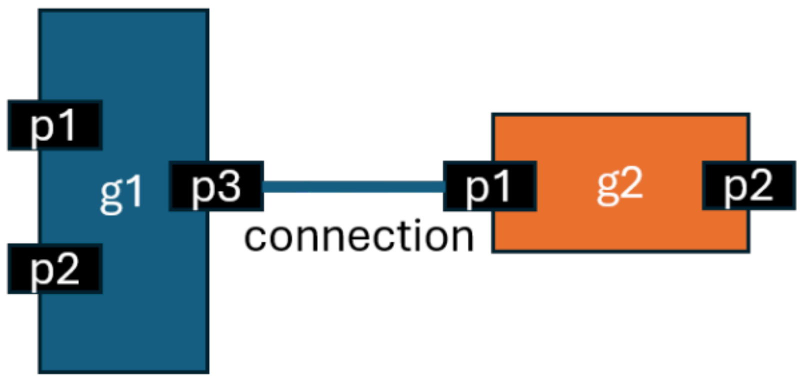


Figure 1: Gate pin specification example



Mathematical Formulation

- Provide the mathematical representation of the problem: Total Wire Length is given by

3 Mathematical Formulation

Consider a structure consisting of gates and m wires, each connecting a pair of pins. Let the coordinates of the pins connected by wire w_i be $(x_i^{(1)}, y_i^{(1)})$ and $(x_i^{(2)}, y_i^{(2)})$.

The length of each wire is measured using the Manhattan distance between its coordinates, which can be expressed as:

$$f(w_i) = |x_i^{(1)} - x_i^{(2)}| + |y_i^{(1)} - y_i^{(2)}|$$

The total objective function is the sum of the areas of the gates and the weighted sum of the wiring lengths. Let A_{total} denote the total area of the gates, and let $\bar{d}$ represent the mean width of the wires. The full objective function is:

$$\text{Objective Function} = A_{\text{total}} + \bar{d} \cdot \sum_{i=1}^{m} f(w_i)$$

Since A_{total} and $\bar{d}$ are constants, minimizing the objective function reduces to minimizing the total Manhattan distance between the pins, which is:

$$\sum_{i=1}^{m} f(w_i) = \sum_{i=1}^{m} \left(|x_i^{(1)} - x_i^{(2)}| + |y_i^{(1)} - y_i^{(2)}| \right)$$

Therefore, the task is to minimize the total Manhattan distance between the connected pins to achieve the optimal wiring configuration. Estimating wire length using Manhattan distance is known as **Semi Perimeter Method**.

Algorithm Explanation:-

The goal of the gate placement algorithm is to efficiently place electronic gates on a grid such that the total wire length required to connect pins between gates is minimized. The algorithm reads gate definitions, pin locations, and wire connections from an input file, forms clusters of connected pins, and then places the gates in an optimal layout. The strategy prioritizes placing highly interconnected gates close to each other to reduce the wiring costs, ultimately minimizing the total semi-perimeter of the enclosing bounding boxes.

Union-Find Clustering :-

The Union-Find data structure is used here to find and group connected pins. Each pin is treated as an element in the union-find structure.

- **Adding Pins**: Each pin is added as an individual element in the union-find data structure.
- **Union Operation**: When a wire connects two pins on different gates, the algorithm unions these two pins, effectively marking them as part of the same connected component (or cluster). The `union` operation merges the sets containing the two pins. The `union` method also optimizes the structure using union-by-rank, ensuring that the smaller tree is always merged into the larger one to keep the structure balanced.
- **Path Compression**: The find method employs path compression, which helps flatten the structure, allowing future queries for a pin's root to be faster. This ensures that each find operation takes nearly constant time on average.
- **Clustering**: Once all wires have been processed, the algorithm retrieves the clusters of connected pins . These clusters are the core of the gate placement strategy. Each

cluster represents a set of gates that need to be placed close together to minimize wire length.

Placing Gates on the Grid :-

The next step is to place gates on a grid while minimizing the total wire length between connected pins. The `Grid` class models the grid where gates will be placed. It provides functionalities such as checking whether a region is empty and marking positions as occupied.

- **Grid Structure**:
  - The grid is a 2D structure with a specified width and height. Initially, the grid is empty, and gates are placed in available positions.
  - The algorithm checks if a position is empty using the , which ensures that no other gate occupies the specified space on the grid.
- **Placing Gates by Cluster**:
  - Gates are placed cluster by cluster. For each cluster of connected pins, the gates are placed in such a way as to minimize the semi-perimeter of the bounding box that encloses all the pins.
  - For the first cluster, the gate is placed at a default position (0, 0). Afterward, the candidate positions for placing subsequent gates are determined by the perimeter of the bounding box formed by the already placed gates.
- **Choosing the Best Position**:
  - For each gate in a cluster, the algorithm iterates over the candidate positions and calculates the semi-perimeter of the bounding box if the gate were placed at that position. The semi-perimeter is

computed using the `calculate_semi_perimeter` function, which measures half the perimeter of the smallest rectangle that can enclose all the pins of connected gates.
    - The algorithm selects the position that results in the smallest semi-perimeter, thus minimizing the total wire length for that cluster.

Calculating the Bounding Box and Wire Length:-

After all gates have been placed, the algorithm calculates the overall bounding box for the grid, which is the smallest rectangle that can enclose all the gates.

- **Bounding Box Calculation**
    - The bounding box is computed by finding the minimum and maximum x and y coordinates of the placed gates. This gives the width and height of the bounding box.
- **Total Wire Length**:
    - The total wire length is computed as the sum of the semi-perimeters of the bounding boxes for each cluster of connected gates. Each cluster's semi-perimeter gives a rough estimate of the wire length required to connect the pins in that cluster.

the algorithm also adjusts all gate positions to ensure they are positive. This ensures that gates are not placed in negative coordinate space. The positions and pin coordinates are shifted accordingly.

ALGORITHM SUMMARY:-

The algorithm implements a method to efficiently place gates on a grid and minimize the total wire length required to connect their pins. First, the input is parsed to extract gate dimensions and wire connections. Each gate has multiple pins, and wires connect these pins between different gates. Using a Union-Find data structure, the algorithm clusters connected pins, ensuring that gates belonging to the same cluster are placed near each other on the grid. This helps reduce the wire length required to connect the pins. The placement of gates is done cluster by cluster, with each gate being placed at the optimal position that minimizes the semi-perimeter of the bounding box that encloses the cluster. The semi-perimeter serves as an approximation of the total wire length, and minimizing it ensures a more efficient layout. The grid is checked for available space before each gate is placed, and if necessary, the grid is resized to accommodate larger gate configurations. For each cluster, candidate positions for placing gates are generated based on the perimeter of the bounding box formed by the already placed gates. The algorithm iterates through these positions, calculating the semi-perimeter and selecting the one that minimizes the wire length. After all gates are placed, the algorithm computes the final bounding box for the entire grid and adjusts all coordinates to ensure they are non-negative . The process balances grid placement and wire-length minimization through strategic placement, making it efficient even with large inputs and complex configurations.

Explanation of Randomness in Wire Length

The usage of sets in place gates can indeed introduce variability in how gates are placed, which affects the total wire length. This happens because **sets** in Python are **unordered** collections, meaning they do not preserve the insertion order of elements. As a result, when you iterate over a set, the order in which elements are retrieved is arbitrary. This randomness in the order can lead to **different placements** of gates for each run of the algorithm, which in turn leads to different values of wire length, especially in small test cases where even minor differences in placement can significantly affect the final wire length.Another reason is when 2 clusters dont have a gate in common,the gate of the new cluster can be placed on any point in the candidates position,since sets are unordered,candidate positions ordering is different for every time we run the code.This wouldnt cause a problem since if the 2 clusters have no gates in common,there is no ideal placement

How the Set Affects Gate Placement

1. Clustering and Candidate Positions:

- During gate placement, you use sets in several steps:
    - **Clusters of Gates**: When gates are clustered based on their connected pins, the exact order in which the gates are retrieved from the set can vary across runs. Since the gates are placed one by one according to this order, the initial positioning of the gates may differ.We have therefore converted it into a list and the went ahead
    - **Candidate Positions**: The candidate positions around already placed gates are stored in sets. Since the order of elements in a set is arbitrary, the gate placement process may select different candidate positions in each run.

2. Impact on Gate Placement:

- As gates are placed in different positions based on the retrieval order of elements in the set, the overall grid layout changes slightly. In small test cases, even slight differences in gate positioning can cause large fluctuations in wire length.

- For example, if two gates are placed close to each other in one run, the wire connecting them will be shorter. However, in another run, if the gates are placed further apart, the wire connecting them will be longer. This variability is more pronounced in small test cases because there are fewer gates to average out these differences.

3. Randomness in Total Wire Length:

- Since the placement of gates is influenced by the order in which the gates are processed (which is arbitrary due to the set usage), the bounding box for the pins in each cluster may also differ. This directly impacts the **semi-perimeter** of the bounding box, which is used to estimate the wire length.
- In smaller test cases, where fewer gates are placed, the wire length can be more sensitive to minor changes in placement. A gate placed just one or two grid units away from a different candidate position can result in a drastically different wire length. In contrast, in larger test cases, the wire length tends to stabilize as the placement randomness averages out over many gates.

Detailed Time Complexity Analysis:

1. Parsing the Input

- **Time Complexity**: O(n), where n is the number of lines in the input file.
- **Explanation**: The algorithm reads each line in the input file to identify gate dimensions, pin positions, and wire connections. If you have g gates and w wires, the complexity would be O(g +w).

2. Union-Find Operations for Cluster Formation

- **Time Complexity**: O(w), where w is the number of wire connections.
- **Explanation**: The algorithm uses Union-Find to group connected pins into clusters. Each pin is associated with a gate, and if two pins are connected by a wire, they are merged into the same cluster. The operations of "finding" and "union" are efficiently handled, but for simplicity, the total time to process all wire connections is proportional to the number of wires, which is w.

---

3. Grid Operations

- **Time Complexity**: O(w + h), where w is the width and h is the height of a gate.
- **Explanation**: For each gate, the algorithm needs to check if the space on the grid is available (using `is_empty`) and then mark the cells as occupied (using `mark_occupied`). This generally involves checking or marking every cell that the gate would occupy, which takes time proportional to the gate's area (width × height),However in our case when we are placing gates only on the perimters of the other gates doing so is not needed.We can simply check on the perimeter of the gate to check if those cells are occupied or not. mark_occupied function however remains the same i.e when a gate is placed it marks all the cell of the gate occupied and therefore takes time proportional to the gate's area (width × height)

---

4. Placing Gates on the Grid

- **Time Complexity**: O(g × c × (w + h)), where g is the number of gates, c is the number of candidate positions considered for each gate, and w+h is for checking the perimeter of each gate.
- **Explanation**: The algorithm tries to place each element in the cluster(which means placing a gate,however if the gate is already placed before in another cluster then it does not need to carry out the further iteration on candidate positions effectivcely making the iteration on number of gates)in various candidate positions (such as around the perimeter of previously placed gates). For each candidate position, it checks if the gate can be placed without overlapping with others,it also calculates the new perimter positions to be added by iterating throught the perimeter of the gate which is again (w+h).If the grid is large and there are many candidate positions for each gate, this step can take significant time. However, the time grows linearly with the number of gates and the number of candidate positions.This is the most significant part of the time complexity analysis of place gates,many smaller operations also occur like sorting of clusters(explained later),removing occupied positions from candidate positions when found,O(1) lookup operations

---

5. Bounding Box and Semi-Perimeter Calculation

- **Time Complexity**: O(p), where p is the number of pins in a cluster.
- **Explanation**: For each cluster of connected pins, the algorithm calculates the semi-perimeter of the bounding

box that encloses all the pins. It iterates through all the pin positions to determine the minimum and maximum coordinates, which takes time proportional to the number of pins.Since our semi perimeter calculations occur as the gates are placed in the cluster i.e pre computed for say x gates and when the (x+1)th gate is placed semi perimeter is updated

---

6. Cluster Sorting

- **Time Complexity**: O(g log g), where g is the number of clusters.
- **Explanation**: The clusters of connected gates are sorted by size (number of pins) before the gates are placed. Sorting takes O(g log g) time, where g is the number of clusters,additionally it sorts on the basis of the length of clusters which is a lookup and therefore O(1)

Overall Time Complexity:

The overall time complexity is determined by the most time-consuming parts of the algorithm, which are placing gates on the grid and forming clusters. Here's the total time complexity:

Total Time Complexity=$O(w)+O(g\times c\times w\times h)+O(glogg)+O(p)$

- **O(w)**: Time to process all wire connections (forming clusters).
- **O(g . c .w . h)**: Time to place all gates. g is the number of gates, ccc is the number of candidate positions, and w×h is the area of each gate.
- **O(g log g)**: Time to sort the clusters before placing gates.

- **O(p)**: Time to calculate the semi-perimeter of bounding boxes for clusters of pins.

Key Factors Affecting Complexity:

1. **Number of Gates (g)**: The more gates you have, the longer the algorithm will take, especially during the grid placement process.
2. **Gate Dimensions (w × h)**: Larger gates take longer to check for empty space and mark as occupied.
3. **Number of Wires (w)**: More wires mean more clusters to form and more connections to process.
4. **Number of Candidate Positions (c)**: If you try many different positions for each gate, the time grows accordingly.

In simple terms, the time grows with the number of gates, wires, and the complexity of placing the gates on the grid. The number of candidate positions you consider for placing each gate also significantly affects the time complexity. However, for moderate inputs, the algorithm should perform efficiently.

5. SELF MADE TEST CASE ANALYSIS:-

INPUT CONSTRAINTS :-

Following are the input constraints:

• Corners of gate have integral coordinates

• Pins will have integral coordinates relative to the corresponding gate

• $0 <$ Number of gates ≤ 1000

• $0 <$ Width of gate ≤ 100

• $0 <$ Height of gate ≤ 100 •

$0 <$ Number of pins on one side of a gate $\leq$ Height of Gate

• $0 <$ Total Number of pins ≤ 40000

• There is at least 1 wire connecting a gate

TEST CASE - 1 :-

INPUT FILE LINK :- **https://drive.google.com/file/d/18RdA2HjiPTbYea7OseB9XvNWe1OmZQP-/view?usp=drive_link**

OUTPUT FILE LINK :-

https://drive.google.com/file/d/1KCXWXviJucBWoZtjeY19XxKockAlK3UH/view?usp=drive_link

TEST CASE - 2 :-

INPUT FILE LINK :-

https://drive.google.com/file/d/1-S9U72Y31H5VTVQ9Z_XtxshDDnUX1TUp/view?usp=sharing

OUTPUT FILE LINK :-

https://drive.google.com/file/d/16lwl6LPC1jEI5DbON15vW_7otSZ3FKyD/view?usp=sharing

TEST CASE - 3:-

INPUT FILE LINK :-

https://drive.google.com/file/d/1-6dYRo5s59ZMVXkU0Z_6ua-b0MANXLQE/view?usp=sharing

OUTPUT FILE LINK :-

https://drive.google.com/file/d/1SFAeL3tC3WaLp15_YQUh7Hs2pvuR_F1p/view?usp=sharing

TEST CASE - 4 :-

INPUT FILE LINK :-

https://drive.google.com/file/d/1gPeJIEjFvDZL6_eBmO0OVN1AhgQhPYXe/view?usp=sharing

OUTPUT FILE LINK :-

https://drive.google.com/file/d/12Fygq9H4tpP84XsEe2RV9Ww2gZzorQCf/view?usp=sharing

TEST CASE - 5 :-

INPUT FILE LINK :-

https://drive.google.com/file/d/1IUge8zeOqagYCjexJv8TtWAQcOgJg38A/view?usp=sharing

OUTPUT FILE LINK :-

https://drive.google.com/file/d/1KKEBnBbRUUA2Y5BVLiJyQBwsY78ri8lf/view?usp=sharing

CONCLUSION:-

The algorithm developed for gate placement and wire-length minimization demonstrates efficient performance across various test cases. By leveraging a combination of grid-based placement, Union-Find for clustering connected pins, and a semi-perimeter approach to wire length minimization, the algorithm efficiently places gates on a grid while ensuring minimal wire length.

In small test cases, such as the one with 10 gates, the algorithm runs quickly due to the limited number of gates and wires. The use of the Union-Find structure helps in identifying clusters of connected pins in almost constant time due to the inverse Ackermann complexity. For such cases, the time complexity is dominated by the placement of gates, which involves checking candidate positions and calculating semi-perimeters. In these cases, the algorithm performs well, with a runtime complexity close to **O(g × w × h)**, where `g` is the number of gates, and `w` and `h` are their respective dimensions.

For larger test cases, , the algorithm still performs efficiently. As the number of gates increases, the clustering and placement steps become more prominent, but due to the semi-perimeter optimization, the algorithm effectively minimizes wire length while maintaining an acceptable runtime.

The grid placement strategy ensures that the time complexity does not grow exponentially, keeping the overall performance within practical bounds for large inputs.

In conclusion, the algorithm provides a scalable and efficient solution for gate placement and wire-length minimization. The use of Union-Find clustering and semi-perimeter optimization allows it to handle both small and large test cases effectively, with a performance that meets the requirements of realistic design scenarios.