COL215 Software Assignment 2

Wiring-aware Gate Positioning

Pramod Kumar Meena 2023CS51175 Manvendra Rajpurohit 2023CS10936

Introduction:

The goal of this assignment is to implement an algorithm that optimizes the placement of rectangular gates on a circuit board, while minimizing the total wire length connecting the gates. The placement must ensure no overlaps, and the wiring length is minimized using the Manhattan distance metric. The algorithm uses simulated annealing to iteratively improve the gate placement by exploring different configurations and accepting worse solutions with a probability based on temperature to escape local minima.

Problem Statement :

This assignment addresses the problem of wiring-aware gate positioning, where the objective is to assign positions to a set of rectangular logic gates on a two-dimensional plane. Each gate has pins located on its boundaries, and the goal is to minimize the total wire length between connected pins. The total wire length is measured using the Manhattan distance between the pins. The solution to this problem incorporates simulated annealing as an optimization technique to iteratively reduce the wire length and find a near-optimal gate placement.

Approach

To solve the gate packing problem, the following approach was adopted:

1. **Input Parsing**: The input file contains details about the gates and their connections. Each gate is specified by its name, width, height, and the coordinates of its pins relative to the gate's bottom-left corner. Wire connections between the pins are also specified. The input is parsed and used to create instances of gate and wire objects within the CircuitBoard class.
2. **Initial Gate Placement:** The initial placement of gates on the plane is based on the number of connections each gate has. Gates with a higher number of connections are prioritized for placement, as they are more constrained in terms of placement flexibility. For the initial positioning, connected gates are placed adjacent to one another to reduce wire length.
   For each gate, we check possible placements around already placed gates to minimize the total distance between connected pins. If no valid placement is found, gates are placed in the next available open space.
3. **Simulated Annealing Optimization**: Simulated annealing is a probabilistic optimization technique that is well-suited for large combinatorial problems like gate positioning. The key idea behind simulated annealing is to allow the algorithm to occasionally accept worse placements (i.e., placements with higher wire length) in the hope of escaping local minima and finding a more globally optimal solution.
4. **Bounding Box and Wire Length Calculation**: The final solution ensures the bounding box containing all gates is minimized, and the total wire length is computed using the Manhattan distance.
5. **Visualization**:

A provided Python script (`visualize_gates.py`) can be used to visualize the final gate layout. Although we made our own visualization file which even shows the wires connected and show length when hovered over that wire. This helps in verifying the correctness of the placement visually.

Mathematical Formulation

The objective function to minimize is the total wire length. For each wire connecting two pins, the wire length is calculated using the Manhattan distance between the pins:

$$f(wi) = |x1-x2|+|y1-y2|$$

The total wire length for all connections is the sum of the wire lengths for each wire:

$$Ltotal = \sum mf(wi) = \sum m(|x1-x2|+|y1-y2|)$$

Simulated annealing is used to iteratively adjust the placement of gates, with the goal of minimizing this total wire length.

Simulated Annealing Approach

Simulated annealing is a probabilistic technique for approximating the global optimum of a cost function. In our case, the cost function is the total wire length.

1. **Initial Placement**: Gates are placed based on connectivity. A gate with the most connections is placed first, and others are placed near their connected gates.
2. **Temperature Scheduling**: The system starts at a high temperature, allowing greater exploration of the solution space. As the temperature decreases, the probability of accepting suboptimal placements reduces.
3. **Acceptance Criterion**: A new placement is accepted if it reduces wire length. If it increases the wire length, it is accepted with a probability dependent on the temperature and the change in wire length.
4. **Termination**: The process continues until the temperature reaches a predefined minimum, or a sufficiently good solution is found.

Explaining the Functions in code:

Gate and Wire Class

These classes store information about each gate and wire:

- The Gate class keeps track of the gate's name, width, height, and the positions of its pins.
- The Wire class defines connections between two gates and their specific pins.

CircuitBoard Class

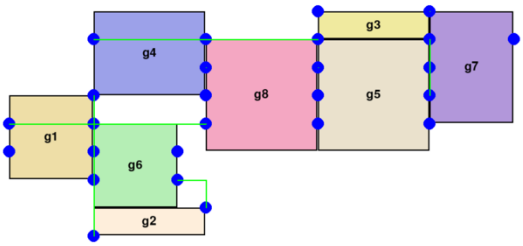
This class stores all gates and wires, and it manages the process of optimizing gate placement:

- **add_gate()** and **add_wire()** methods add gates and wire connections to the board.
- **calculate_wire_length()** calculates the total wire length based on Manhattan distances between connected pins.
- **optimize_placement()** initializes gate placement on a grid and ensures no overlaps. It calls **simulated_annealing()** to further optimize the placement.
- **simulated_annealing()** adjusts the placement iteratively, using a probabilistic acceptance criterion to avoid local minima. The algorithm cools down gradually, focusing on refining the solution.

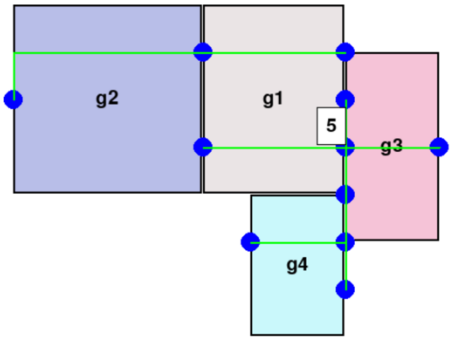
parse_input()
This function reads the input file and parses the gate dimensions, pin locations, and wire connections. It returns a CircuitBoard object populated with gates and wires.
main()
This function orchestrates the execution of the program. It parses the input, optimizes the gate placement, and generates the final output, which includes the bounding box dimensions and the minimized wire length.

Visualizations and total_wire_length on Provided test cases

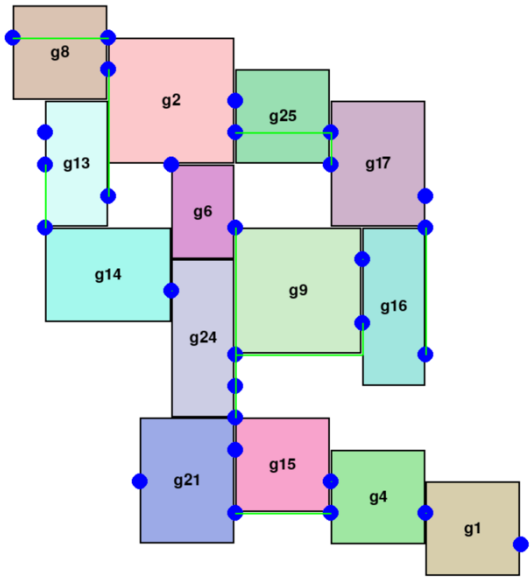
Sample Test Case 1: Total wire length = 26



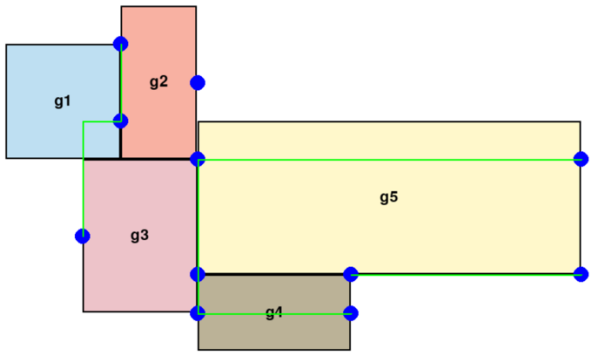
Sample Test Case 2: Total wire length = 26



Sample Test Case 3: Total wire length = 50



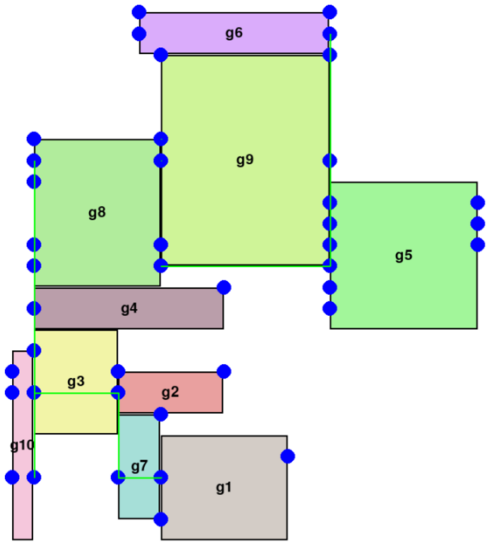
Sample Test Case 4: Total wire length = 31 (by your visualisation file)



Visualizations and total_wire_length on Custom test cases (made by us)

Custom_tc_1: Total wire length = 48 (this is the random generated test case)

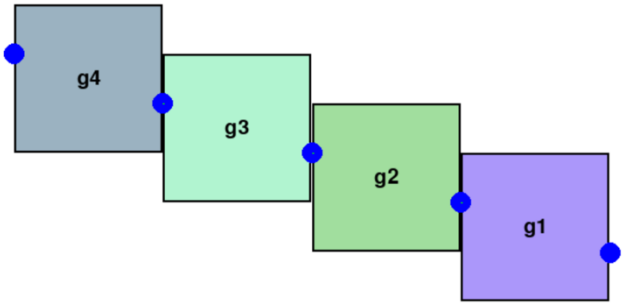
```
g1 6 5
pins g1 0 2 0 4 6 1
g2 5 2
pins g2 0 1 5 0
g3 4 5
pins g3 0 1 0 3 4 2 4 3
g4 9 2
pins g4 0 1 9 0
g5 7 7
pins g5 0 2 0 3 0 4 0 5 0 6 7 1 7 2 7 3
g6 9 2
pins g6 0 0 0 1 9 0 9 1
g7 2 5
pins g7 0 3 2 0
g8 6 7
pins g8 0 0 0 1 0 2 0 5 0 6 6 0 6 1 6 6
g9 8 10
pins g9 0 0 0 4 0 5 0 9 8 0 8 5 8 7 8 8 8 9
g10 1 9
pins g10 0 1 0 2 0 6 1 0 1 2 1 6
wire g6.p4 g9.p5
wire g3.p1 g10.p6
wire g8.p7 g9.p3
wire g3.p4 g10.p4
wire g3.p1 g8.p2
wire g8.p8 g9.p5
wire g8.p6 g9.p2
wire g5.p2 g9.p9
wire g2.p1 g3.p4
wire g4.p1 g8.p5
wire g7.p1 g1.p1
wire g7.p1 g2.p1
```



Custom_tc_2: Total wire length = 0

(this test case is made considering edge case
Where long Chain gates are present)

```
g1 3 3
pins g1 0 1 3 2
g2 3 3
pins g2 0 1 3 2
g3 3 3
pins g3 0 1 3 2
g4 3 3
pins g4 0 1 3 2
```

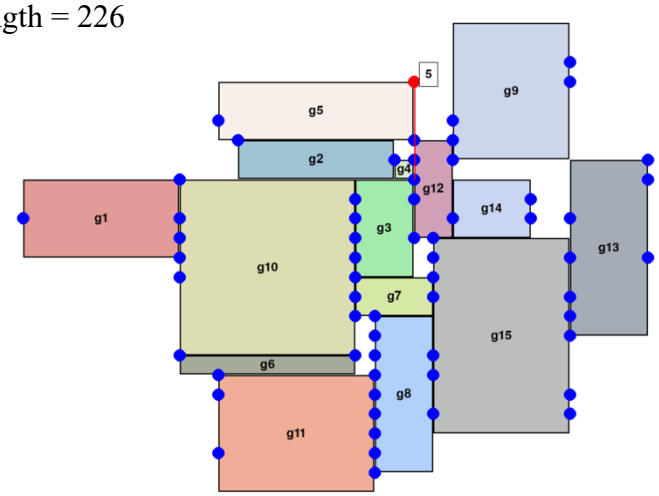


wire g1.p1 g2.p2
wire g2.p1 g3.p2
wire g3.p1 g4.p2

Custom_tc_3: Total wire length = 226

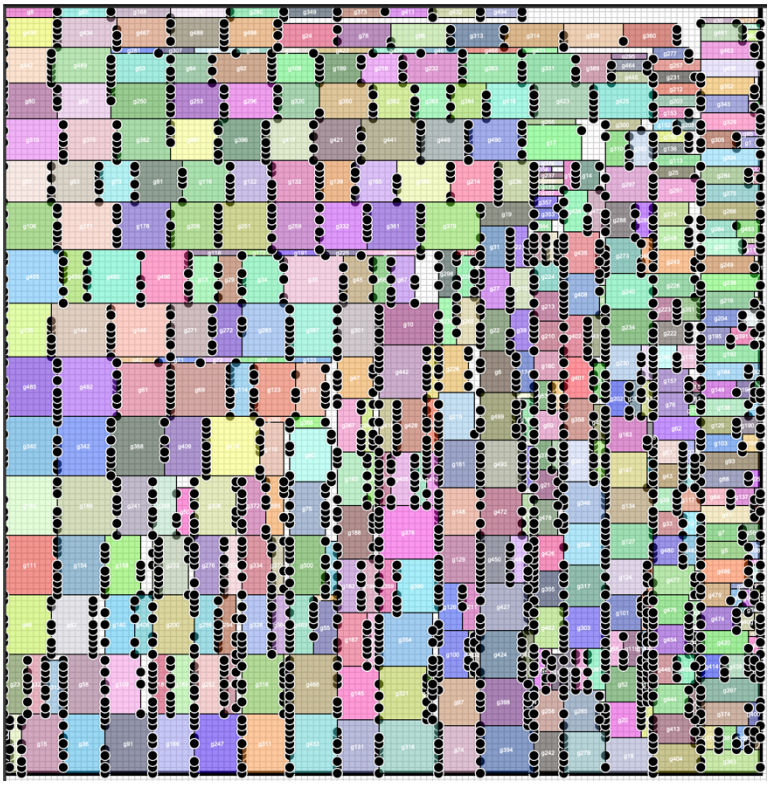
(input.txt file of this is very large
So can't be putted here in the report
Kindly please check the folder of
Custom test cases)

(this tc is made for the case when
gates have so many pins)



Custom_tc_4: Total wire length = 53928 (as per provided visualization file)

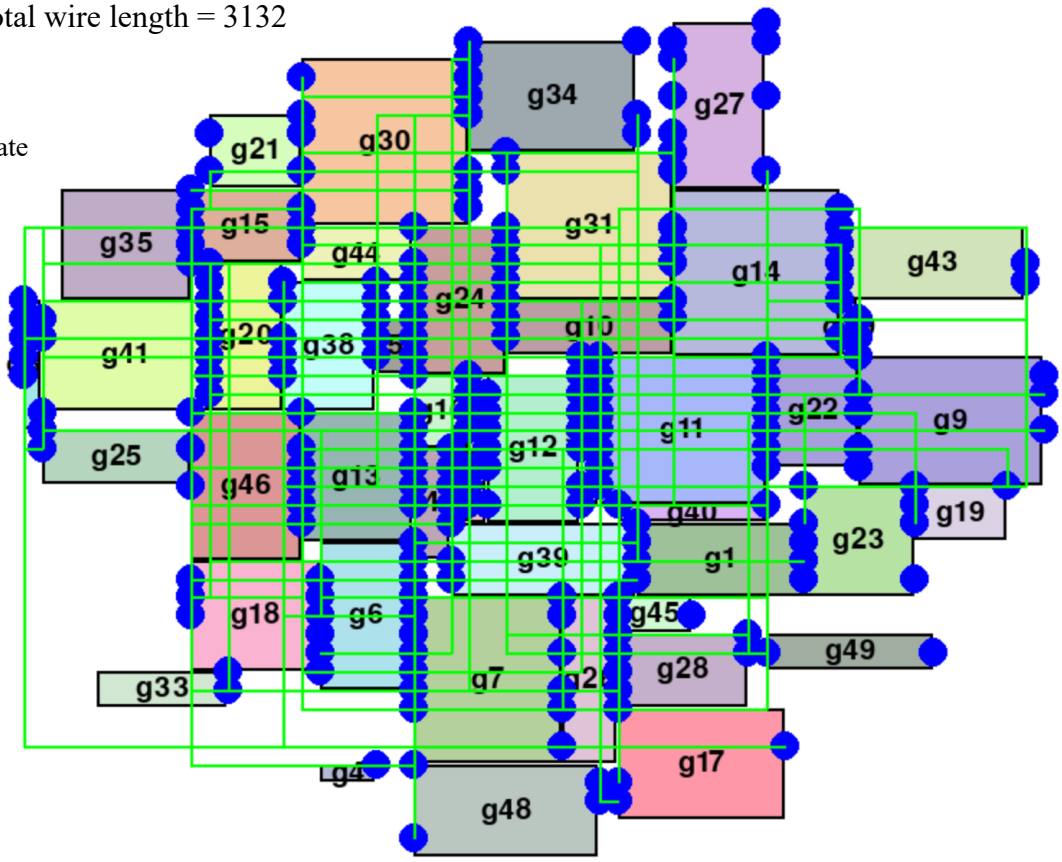
(This test case is where almost every size
Of gates are present and each are
Connected in some pattern)
(input.txt file of this is very large
So can't be putted here in the report
Kindly please check the folder of
Custom test cases)
(as our algo is based on simulated annealing
And for certain cases we also optimised our
Previous assignment.
Sometimes the output may be different but
Total wire length will be almost similar)



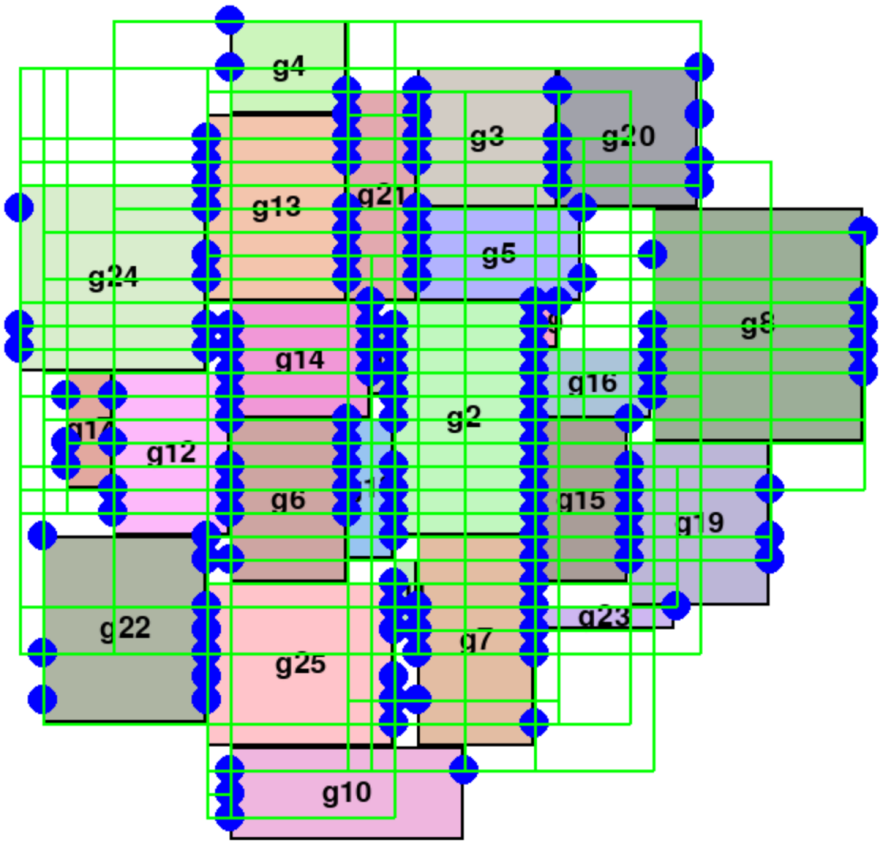
Custom_tc_5: Total wire length = 3132

(tc which contains moderate Number of wires)

(kindly please check Input.txt in folder)



Custom_tc_6: Total wire length =



Time Complexity Analysis

CircuitBoard Class

- add_gate: O(1) - Dictionary insertion
- add_wire: O(1) - List append and constant time operations
- calculate_wire_length: O(W) where W is the number of wires
    - Iterates through all wires once, performing constant time calculations for each
- optimize_placement: O(G*log G +G*G*(bounding_box) +SA) =O(G*G*(bounding_box) +SA)where:
    - SA is the complexity of simulated_annealing
- simulated_annealing: O((G+W)*I)where:
    - For each iteration:
        - Random gate selection: O(1)
        - Wire length calculation: O(W)
- is_valid_placement: O(G) where G is the number of gates
- generate_output: O(G + W) where:
    - W is needed for calculate_wire_length
- shift_to_first_quadrant: O(G) where G is the number of gates
    - Finding minimum coordinates: O(G)
    - Updating all gates: O(G)
- Place_connected_gates: O(G*W)
- Place_unconnected_gates: O(bounding_box * G)

parse_input

- Time Complexity: O(L + P + W) where:

Final time complexity = O(G*G*(bounding_box) +SA) = max (O(G*G*(bounding_box)), O((G+W)*I)) .

{ G is the number of gates,
I is the number of iterations (controlled by temperature and cooling rate)
L is the number of lines in the input file
P is the total number of pins across all gates
W is the number of wires }

Design Decisions

- **Sorting by Dimension**: Sorting gates by their larger dimension helps to minimize fragmentation, as larger gates are less flexible in placement compared to smaller gates.

- **Iterative Placement**: Rather than using a greedy algorithm that could lead to suboptimal packing, the algorithm checks all feasible positions to ensure that the smallest possible bounding box is achieved.
- **Dynamic Bounding Box**: Adjusting the bounding box as gates are placed prevents unnecessary expansion of the bounding area.

Conclusion

The simulated annealing approach to wiring-aware gate positioning provides an effective method for minimizing the total wire length in complex circuits. Although the method is probabilistic and may not always yield the global optimum, it consistently finds near-optimal solutions in a reasonable time for both small and large test cases.

Resources:

- https://tilos-ai-institute.github.io/MacroPlacement/CodeElements/SimulatedAnnealing/
- https://link.springer.com/chapter/10.1007/3-540-32539-5_49

also tried: we have also tried using the GREEDY, SKYLINE method but that was specifically designed for placing of gates so it was not able to minimize the wire length.