COL215 SW Assignment 2 - Wire Aware Gate Packing

Submission By :

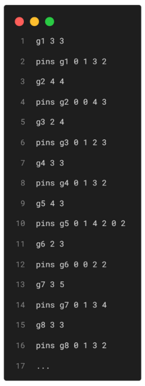
Yash Rawat **2023CS50334**

Priyanshi Gupta **2023CS10106**

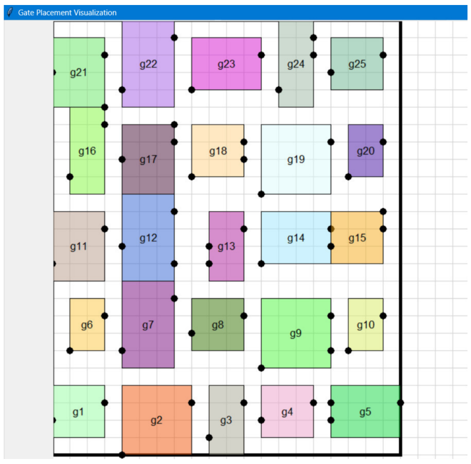
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

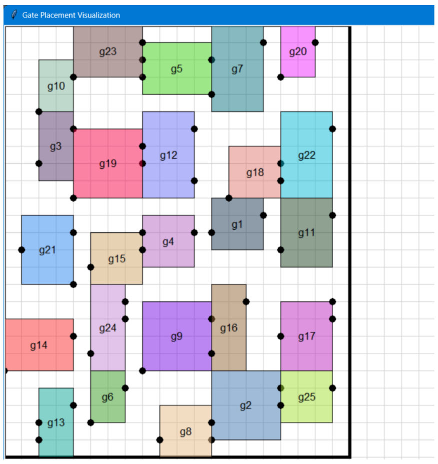
October 4, 2024

Input File

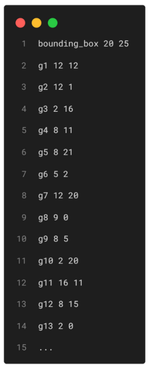


```
g1 3 3
pins g1 0 1 3 2
g2 4 4
pins g2 0 0 4 3
g3 2 4
pins g3 0 1 2 3
g4 3 3
pins g4 0 1 3 2
g5 4 3
pins g5 0 1 4 2 0 2
g6 2 3
pins g6 0 0 2 2
g7 3 5
pins g7 0 1 3 4
g8 3 3
pins g8 0 1 3 2
...
```





Output File



```
bounding_box 20 25
g1 12 12
g2 12 1
g3 2 16
g4 8 11
g5 8 21
g6 5 2
g7 12 20
g8 9 0
g9 8 5
g10 2 20
g11 16 11
g12 8 15
g13 2 0
...
```

1 Modelling Wire-aware Gate Packing

1.1 What is wire aware gate packing?

Wire-aware gate packing denotes the simultaneous arrangement of logic gates, represented as rectangles, on a circuit board while concurrently forming pin-level connections to produce a net-list. The aim of this assignment is to decrease the aggregate projected total wire length.

1.2 Understanding the problem statement

The problem statement models

1. The gates as rectangles (provided as input for each test case): $\{g_1, g_2, ..., g_n\}$ each represented by a pair of integers: $g_i = (w_i, h_i)$, where w_i and h_i are the width and the height of the i^{th} board.

2. The input and output pin locations (x and y coordinates relative to bottom left origin) on the boundary of each gate represented by its gate and PIN: $\{g_i.p_1, g_i.p_2, ..., g_i.p_m\}$ (where gate $\{g_i\}$ has m pins).

3. The wires (pin-level connections), represented by $\{(g_i.p_x, g_j.p_y)\}$ where x^{th} pin of i^{th} gate is connected to y^{th} pin of j^{th} gate.

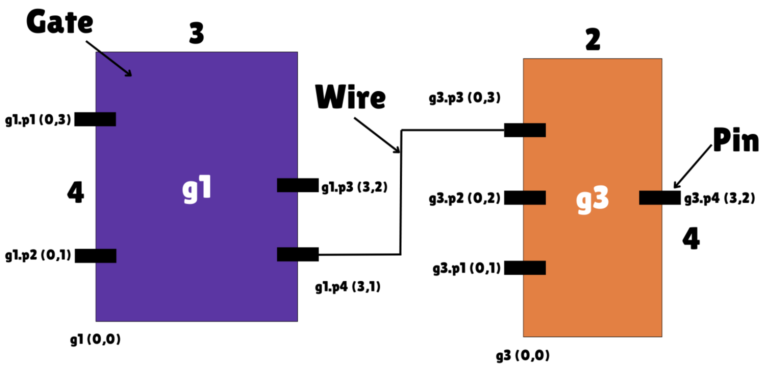


Figure 1: Modelling the Problem Statement

A given set of gates is said to be "correctly assigned" if no two gates have overlapping areas. Our aim is to pack the gates and establish pin level connections so that sum of estimated wire lengths for all wires in the whole circuit is minimized. We assume that gates cannot be reoriented and all wiring is horizontal and vertical.

1.2.1 What are we minimizing?

Our objective is to minimize the overall wire length necessary for connecting the specified pins and gates.

1.2.2 How are we calculating the length?

Overlapping wire connections are treated as a single wire.The method involves calculating the semi-perimeter of the bounding box surrounding a collection of connected pins.
The semi-perimeter is calculated for each set of interconnected pins, and the total length of wire is determined by summing these values. Thus we construct the objective function as follows
Let wire w_i connect two pins whose coordinates are $(x_i^{(1)}, y_i^{(1)})$ and $(x_i^{(2)}, y_i^{(2)})$. Let A_{total} denote the total area of gates, d represents the mean width of wires and $f(w_i)$ is the Manhattan distance between pin coordinates of two pins connected through a wire.

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

$$Objective function = A_{total} + d. \sum_{i=1}^{m} f(w_i)$$

We do not use the combined wiring case because it makes the calculation complicated and the a polynomial time solution for minimizing this length does not exist.

1.2.3 What are the constraints/bounds which are being taken into consideration?

1. The corners of gate have integral coordinates.
2. Pins will have integral coordinates relative to the corresponding gate
3. $0 <$ Number of gates ≤ 1000
4. $0<$ Width of gate ≤ 100
5. $0 <$ Height of gate ≤ 100
6. $0 <$ Number of pins on one side of a gate $\leq$ Height of Gate
7. $0 <$ Total Number of pins ≤ 40000
8. There is atleast 1 wire connecting a gate
9. A gate has left side and right side pins. For right hand side pins, multiple wires can emanate from a single pin. For left hand side pins, atmost 1 wire can enter into a single pin.

2 Algorithm Conceptualization

2.1 Initial Ideas

Initially, we tried approaching this problem using a naive method, knowing that our Naive Packing and the Pack by Pixels method would produce a given configuration for this problem. While our initial idea worked for smaller cases, we were not able to scale our initial idea for larger number of gates, pins and wire connections and thus decided to pivot into using an approach more closely related to metallurgical and Probabilistic methods inspired by these resources[1][2].

2.2 What is Annealing ?

In metallurgy, annealing refers to the process of heating a material to a high temperature and then gradually cooling it down to remove defects and achieve a more stable, low-energy configuration. This process allows the material to explore various configurations before settling into a state with minimal internal energy.

Thus, simulated annealing is a probabilistic process that tries to approximate the global optima of a cost function (which can be the sum of lengths of wires in our case). It does so by generating an initial configuration of gates at a very high temperature, perturbing these configurations and accepting them based on a probability function, which depends on the current temperature as well as the change in cost from the older configuration.

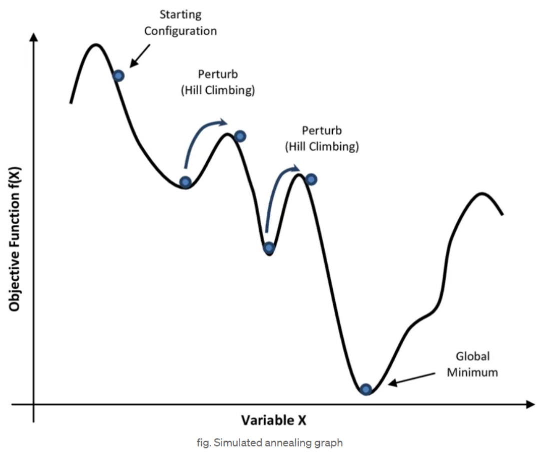


Figure 2: Visualization of Simulated Annealing

[1] VLSI Cell Placement - Shahookar & Mazumder

[2] Placement and Channel Routing by Simulated Annealing Some Recent Developments

2.3 Why do Annealing ?

Generalised VLSI gate placement is proven to be an NP hard problem, i.e. producing global optimum will take non-polynomial time. SA[3] has been used to solve many computationally hard problems with discrete solution spaces like Travelling Salesman Problem (TSP), Job Shop Scheduling (JSS), etc.

It performs better at approximating the global optima where exact global optima is not needed and we need the algorithm to run in feasible time. Also "cooling" of the system can be seen as decrease on the probability (implementation based on cooling schedule as well as `acceptance_probability` function) of accepting worse configurations as our configuration is achieving a optimum value.

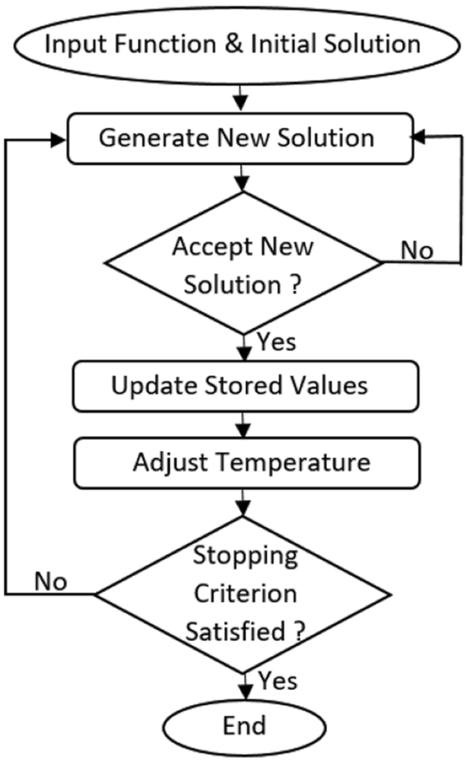


Figure 3: Flowchart of Simulated Annealing

[3] SA - Simulated Annealing (abbreviation used throughout document)

3 Project Structure

3.1 Object Oriented Programming (oops.py)

To store the data of gates, its bounding envelope, pin connections as well as performing the Annealing process, we have implemented multiple classes which include the `Gate_Data`, `Gate_Env`, `Pin`, and `Simulated_Annealing` classes. We will tackle them in increasing complexity.

3.1.1 Pin Class

An instance of `Pin` class is an abstract object that represents the pin of a gate. It stores it's parent gate's index (used for referencing), its own index, the relative coordinate to the parent gate's origin.

The most important functionality of Pin is that when wires are added, it stores the the relevant connections to itself in a dictionary (where the key is index of the other gate (to which it is connected) and the value is a list of indices of the pins of the other gate connected to said pin). This is implemented using `connected_to` method and plays a vital role in `const_delta_function` method of `Simmulated_Annealing` class.

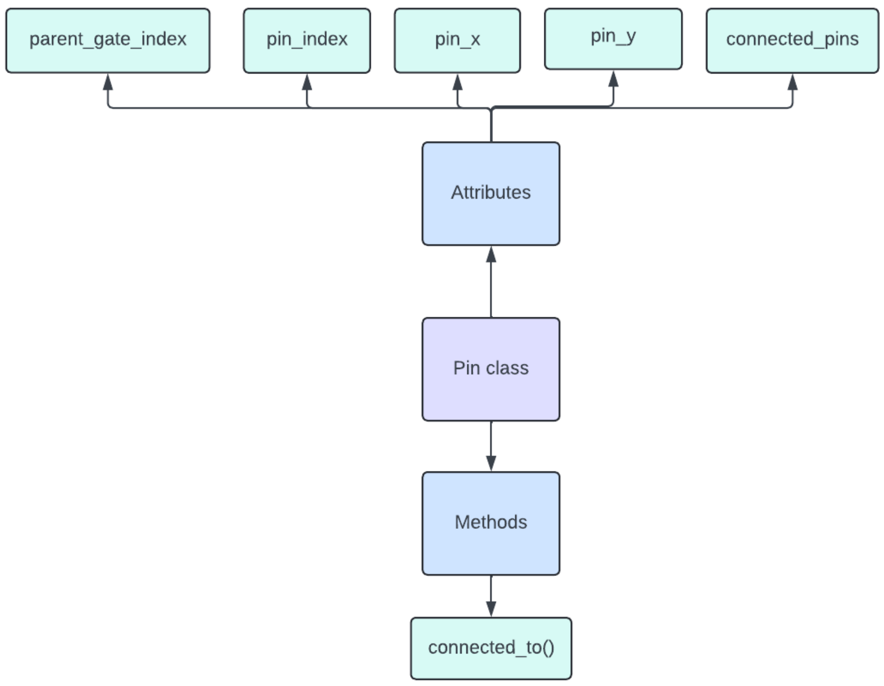


Figure 4: Pin class

3.1.2 Gate_Env Class

An instance of `Gate_Env` class stores the environment of a gate, which includes : Dimension's and global position of the Envelope of given gate, index, dimensions and relative (to the envelope) as well as global coordinates of the gate.

`pins` is an important attribute of `Gate_Env` , a dictionary with key as the pin and value as the `Pin` object. These are used for cost calculations in the `Simulated_Annealing` class.

`Gate_Env` supports various methods such as `set_env`, `set_coord_env`, `set_coord_rel_env` to set the dimensions of the envelope, set the global coordinates of envelope, set coordinates of gate relative to envelope, etc.`get_global_coord` returns the global coordinate of gates for annealing purposes.

Additional methods such as `add_pin`, `get_global_coord_pin` are required to add instances pin object and retrieve global coordinates of pins that are required during wire calculations during annealing process.

Implementation of remaining attributes, methods as well as the `__init__` method can be referred to from the doc-strings of code.

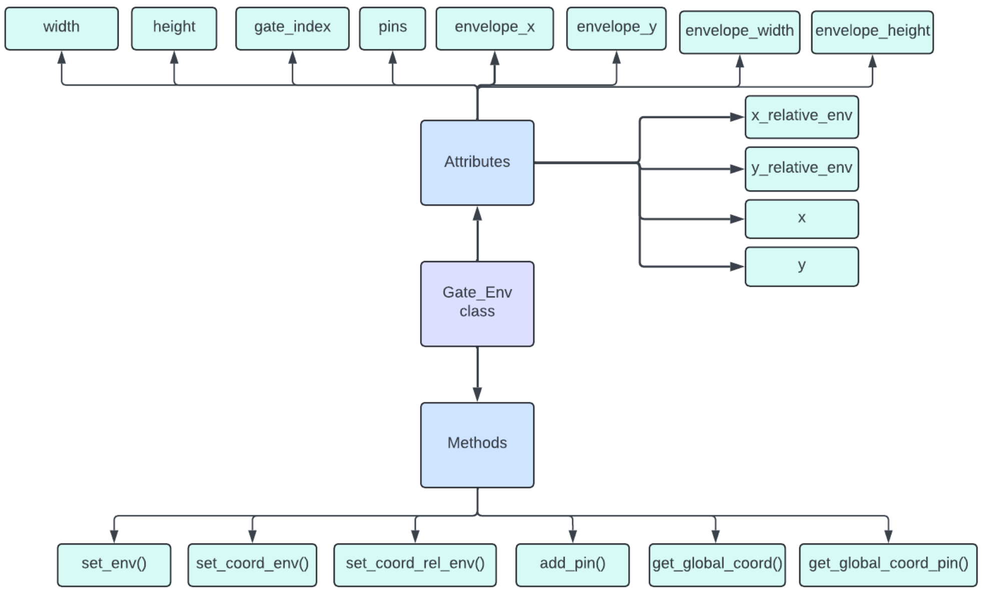


Figure 5: Gate Env class

3.1.3 Gate_Data Class

An instance of `Gate_Data` is used to keep track of the net-list data of a given test-case. This was done to prevent spaghetti code and in layman's terms allows user to store and operate upon different input test-cases/net-list configurations from one python program.

Attributes such as `gates`, `wires` store instances of `Gate_Env` and wire connections (in form of tuple) in a dictionary. It also keeps track of `max_width` and `max_height` which are used to set the envelope sizes for all the gates once the entire input has been read by `Parse_Input` using `set_gate_env` method. `bbox` and `wire_length` store the dimensions of Bounding box and twice the wire length[4]of current configuration.

`Gate_Data Class` supports various functionalities such as `add_gate` ,`get_gate`, `add_pin`, `get_pin` for adding and referencing, gates and pin (using indexing), `add_wire` for adding a wire, etc.`set_bbox` and `get_bbox`

`set_gate_env` is called after the entire input is read and calls the `set_env` on all the gates, setting their envelope dimensions to the `max_width` and `max_height`. Implementation of remaining attributes, methods can be referred to from the documentation of code.

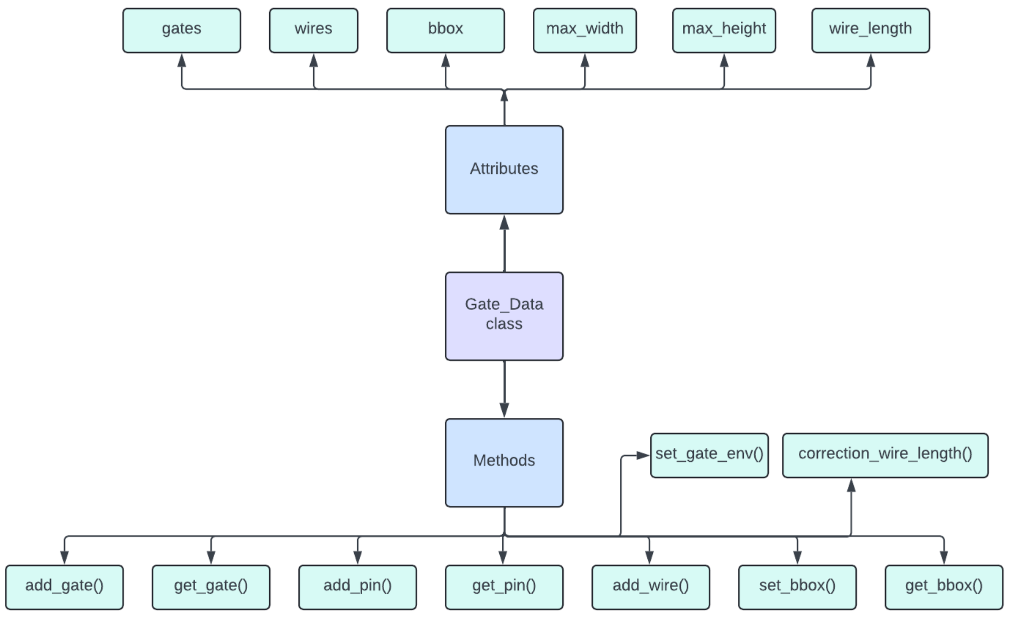


Figure 6: Gate Data class

[4] This implementation detail is explained here

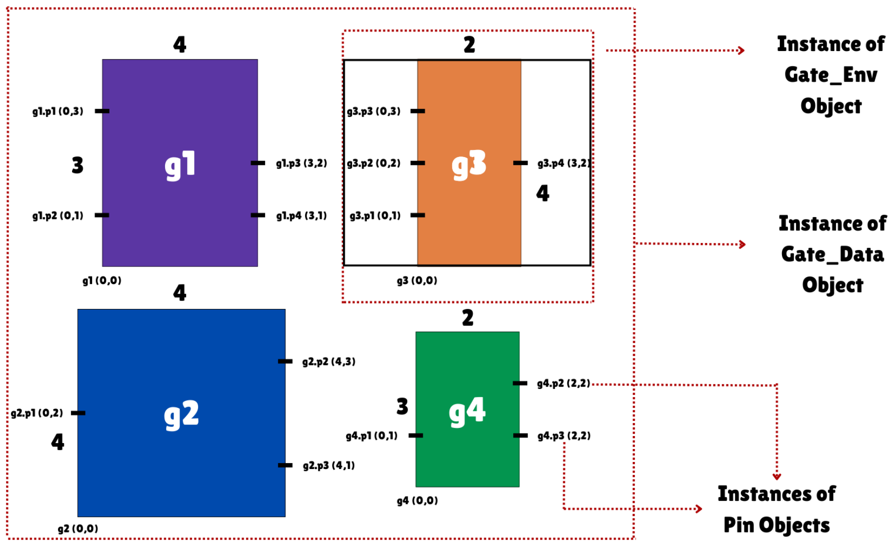


Figure 7: Schematic of the above aforementioned classes

3.2 Simulated_Annealing Class

The `Simulated_Annealing Class` encapsulates multiple methods which perform simulated annealing to optimize gate placement to minimize wire length. These functions and their descriptions are as follows:

1. `acceptance_probability`: This method calculates the acceptance probability of a new perturbed solution based on the cost difference and current temperature.If the new cost is less than the previous value,it is always accepted. The method returns a boolean value.

2. `update_wire_cost`: This method modifies the existing wire cost in the Annealing class instance, updates the `gate_data` attribute, and returns the result.

3. `wire_cost_function`:This method calculates the aggregate wire length for the existing gate configuration, it is costly in terms of the number of wires and should be used solely when essential.

4. `cost_delta_function`:This method determines the variation in wire length resulting from the interchange of two gates.It is a very crucial method as recalculating the wire length for all wires is unnecessary.

5. `gen_init_packing`:This method produces a deterministic initial packing of gates, determined by their order, along with the corresponding bounding box.It establishes the initial wire cost and is invoked whenever a new annealing process commences.

6. `perturb_packing_swap`: Executes a perturbation by exchanging two randomly chosen gates and computes the new wire cost using the `wire_cost_function`

method.The perturbation is accepted or rejected according to the `acceptance_probability` method.If rejected, the gate environments revert to their previous values.

7. `perturb_packing_swap_v2`: It executes a perturbation by exchanging two randomly chosen gates and computes the delta wire cost utilising the `cost_delta_function`.

   The perturbation is accepted or rejected according to the `acceptance_probability` method.If rejected, the gate environments revert to their previous values.(This method is substantially more efficient than `perturb_packing_swap`)

8. `perturb_packing_move`:This executes a perturbation by relocating a randomly chosen gate to a new position.

9. `anneal_to_pack(perturb_freq_per_iter=5)`: This method executes the simulated annealing algorithm to enhance gate placement optimisation.

10. `anneal_routine`: This procedure designed to execute the annealing process. This process establishes the invocation of the annealing function through a single call to the `anneal_to_pack` method, utilising `perturb_freq_per_iter` to estimate runtime and determining the iteration count based on the available time for packing optimisation.

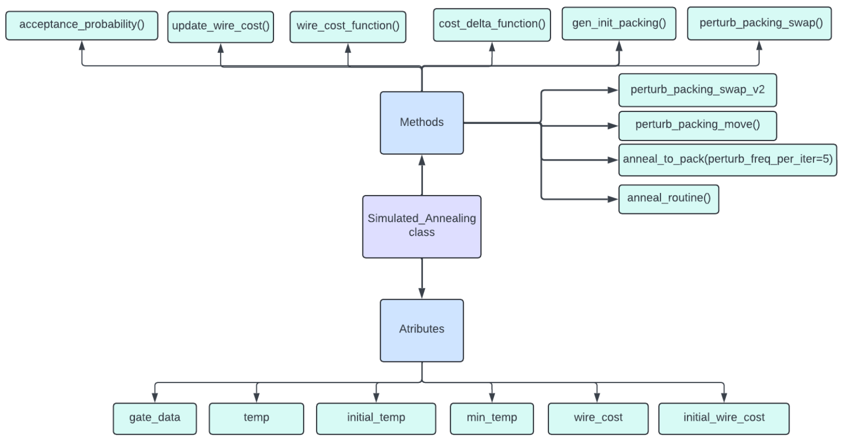


Figure 8: Simulated Annealing class

4 Algorithm Design

Since the $(0, 0)$ of our grid coordinates are taken in the bottom left (similar to the given problem statement).Thus if our algorithm finds a packing, it works in the original problem statement.The following are the steps of the algorithm:

4.1 Input Parsing

The `Parse_Input` method analyses the input from our text file, which includes information about the netlist's gates, pins, and wires.The system stores this information in a Gate_Data object and, upon parsing all input, invokes the methods to configure the gates' envelopes.It also identifies groups of interconnected pins, specifically those connected by the same wires.

4.2 gen_init_packing method

This is the initial phase of the annealing process, utilised to establish a preliminary configuration for the gates on which the simulation would be conducted:

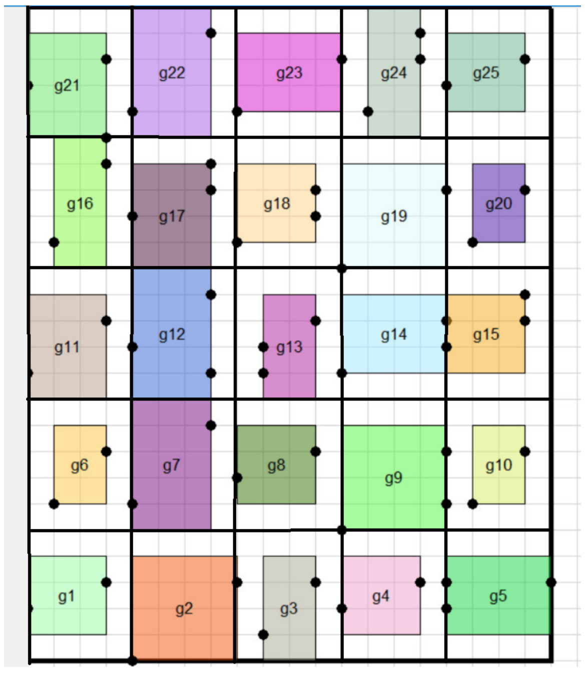


Figure 9: Initial Packing

1. An upper bound is established on the dimensions of the bounding box by using the number of gates to be packed.
2. Since each gate is always enclosed in its envelope (initially and ensured throughout the implementation, explained here). The envelopes are in a square-shaped grid in a row-major order.

3. Thus for each gate the `set_coord_env`, `set_coord_rel_env` are called and each gate is initially assigned to be in equidistant from both the pair of boundaries, centered to the envelope.
4. Calls are made to `wire_cost_function`,`update_wire_cost` to set the wire length data for our initial packing.
5. As this the bounding box generated is going invariant due to our further implementation details and hence it's dimensions are updated in the `gate_data` attribute of `SA` class.

4.3 anneal_to_pack method

This is the method that simulates the Annealing process from starting from a high temperature of $\approx 10^5$ till it cools down below a certain threshold level :

1. Generates an initial packing using the `gen_init_packing` method (This implementation was later changed to occur only if `call_init_pack` parameter was True.)
2. Enters a While loop that executes till the `temp` of the system is greater than the `min_temp`.
3. Calls the `perturb_packing_swap` method to perturb the packing of the system by swapping the two random gates and accepting the swapped state[5]based on the`acceptance_probability` function.
4. Cools down the state system based on `cooling_rate` function and the enters the next iteration of While loop.

To take a look at how the cost of the packing is expected to decrease after annealing we will take a look at `perturb_packing_swap` and `acceptance_probability` methods of `SA` class mentioned below.

4.4 perturb_packing_swap method

This method helps in annealing simulation since it goes from current packing to a new packing state using following implementation details :

1. Initializes the seed of random function (explanation here) and selects two random gates (by indices) which will be swapped.
2. Stores the initial coordinates of the envelope as well as global coordinates of both the gates and their envelopes (which may be needed in case we want to revert back from new state to old state)

[5] State and Packing of the net-list have been used interchangeably henceforth

3. Swap the position of the envelopes of both the gates and recalculate their new global positions.

4. At this point no new wire cost calculations have been made so we store the `old_wire_cost` and calculate the `new_wire_cost` using `wire_cost_function`[6] method of `SA`.

5. `old_wire_cost` and `wire_cost_function` are passed onto the `acceptance_probability` method (implementation discussed in next sub-section) which returns whether or not to accept the new packing.

6. Based on above either the `wire_cost` of the system is updated (using `update_wire_cost`) or the state is reverted back to the original state.

4.5 wire_cost_function method

This approach traverses all groups of interconnected pins and estimates the wire length for each group as the semi-perimeter of the bounding box encompassing the pins in that group. The total wire length is the aggregate of the semi-perimeters computed for each group.

4.6 Improvements made in above algorithm

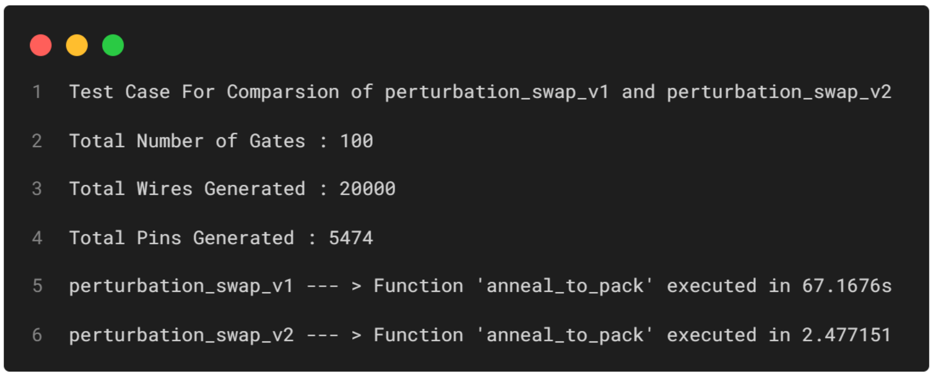
After running and testing our initial implementation we made the following progressive improvements to achieve better packing:

1. Implemented the `cost_delta_function` that calculates the change in cost caused by swapping two gates.It carefully recalculates for those group of interconnected pins that whose pins are connected to the swapped gates. It is better to use than the costly `wire_cost_function` that iterate over all the pin groups as is visible below (also discussed further here): As visible, the difference between `v1` using `wire_cost_function` and `v2` using `cost_delta_function` is significant.

2. Implemented `perturb_freq_per_iter` parameter for `anneal_to_pack`, controlling the number of calls to `perturb` methods in a single temperature transition. It was observed that varying this (by default 1 in original implementation) gave better cost reduction, albeit at a linear runtime trade-off (graphs added in analysis).

3. Implemented `perturb_packing_move`, a new way to perturb our configuration in which a random gate is selected and moved inside its envelope to one of the predetermined 8 locations and performing rest operations similar to `perturb_packing_swap`. Including this during annealing operation made the function achieve a lower minima since their was a new way to change the cost

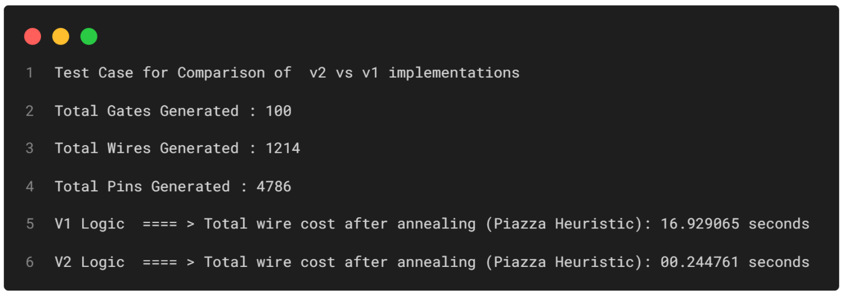
[6] This is a major point of improvement, reasons of which are explained in later parts



```
Test Case For Comparsion of perturbation_swap_v1 and perturbation_swap_v2
Total Number of Gates : 100
Total Wires Generated : 20000
Total Pins Generated : 5474
perturbation_swap_v1 --- > Function 'anneal_to_pack' executed in 67.1676s
perturbation_swap_v2 --- > Function 'anneal_to_pack' executed in 2.477151
```

Figure 10: Comparison between wire_cost and cost_delta based implementations

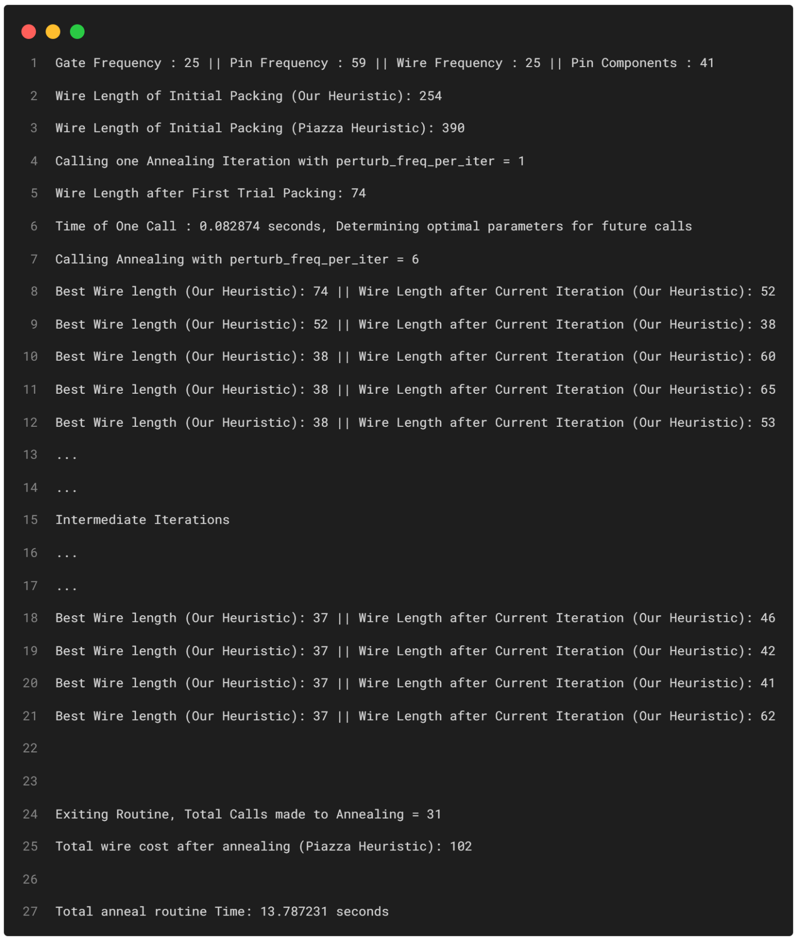
function.Our analysis revealed that `perturb_pack_move` does not yield substantial enhancements when the number of pins exceeds 20,000; hence, we opted not to utilise this function with a high pin count.



```
Test Case for Comparison of  v2 vs v1 implementations
Total Gates Generated : 100
Total Wires Generated : 1214
Total Pins Generated : 4786
V1 Logic  ==== > Total wire cost after annealing (Piazza Heuristic): 16.929065 seconds
V2 Logic  ==== > Total wire cost after annealing (Piazza Heuristic): 00.244761 seconds
```

Figure 11: Comparison between different perturbing logic and their combination

4. Implemented `anneal_routine` method - Since the `anneal_pack` process is probabilistic, running it multiple times can change the outcome. Hence this method estimates the runtime of one call and depending upon that chooses the optimal `perturb_freq_iter` parameter for subsequent calls. This process takes place till our estimated runtime exceeds a certain threshold, during which the best state's data is stored in the `final_packed_data` attribute of `SA` class. Care is taken that the correct cost is implemented by `wire_cost_piazza` in the end.



```
Gate Frequency : 25 || Pin Frequency : 59 || Wire Frequency : 25 || Pin Components : 41
Wire Length of Initial Packing (Our Heuristic): 254
Wire Length of Initial Packing (Piazza Heuristic): 390
Calling one Annealing Iteration with perturb_freq_per_iter = 1
Wire Length after First Trial Packing: 74
Time of One Call : 0.082874 seconds, Determining optimal parameters for future calls
Calling Annealing with perturb_freq_per_iter = 6
Best Wire length (Our Heuristic): 74 || Wire Length after Current Iteration (Our Heuristic): 52
Best Wire length (Our Heuristic): 52 || Wire Length after Current Iteration (Our Heuristic): 38
Best Wire length (Our Heuristic): 38 || Wire Length after Current Iteration (Our Heuristic): 60
Best Wire length (Our Heuristic): 38 || Wire Length after Current Iteration (Our Heuristic): 65
Best Wire length (Our Heuristic): 38 || Wire Length after Current Iteration (Our Heuristic): 53
...
...
Intermediate Iterations
...
...
Best Wire length (Our Heuristic): 37 || Wire Length after Current Iteration (Our Heuristic): 46
Best Wire length (Our Heuristic): 37 || Wire Length after Current Iteration (Our Heuristic): 42
Best Wire length (Our Heuristic): 37 || Wire Length after Current Iteration (Our Heuristic): 41
Best Wire length (Our Heuristic): 37 || Wire Length after Current Iteration (Our Heuristic): 62


Exiting Routine, Total Calls made to Annealing = 31
Total wire cost after annealing (Piazza Heuristic): 102

Total anneal routine Time: 13.787231 seconds
```

Figure 12: Example of anneal_routine

4.7 Correctness of Algorithm

4.7.1 No overlapping of gates

Since we have implemented gate envelopes, each gate is always packed inside a unique envelope and envelopes are placed non intersecting during the `gen_init_packing` and careful swapping.Thus two gates can never overlap.

4.7.2 anneal_pack method

The system will exit after a finite number of iterations due to cooling occurring after each temperature step, regardless of cost improvement during perturbing operations.

4.7.3 anneal_routine

The process is designed to terminate when the time_it wrapper is employed to assess the runtime of each anneal_pack invocation, maintaining a record of the estimated runtime across calls. Termination occurs based on global parameters, including TIME_BOUND_TOTAL_SEC and TIME_BOUND_TOTAL_BUFFER.

5 Time Complexity Analysis

5.1 Finding pins connected by same wires

The time complexity arises from traversing every wire and pins to find pins connected by the same wires. $O(P+W)$, where P represents the quantity of pins and W signifies the quantity of wires in the netlist.

5.2 Generation of initial packing

The `gen_init_packing` method iterates through the complete set of gates, updating both their envelope coordinates and relative coordinates. The time complexity of this procedure is $O(G)$, where G is the number of gates. This method includes the preliminary wire length computation performed by the `wire_cost` function.

5.3 wire_cost function

The approximation of wire length involves determining the semi-perimeter of interconnected pins. Consequently, the function traverses each pin inside a group and performs constant time evaluations for $x_{min}, x_{max}, y_{min}$, and y_{max} corresponding to the current group, resulting in an effective time complexity of $O(P)$, where P is the number of pins.

5.4 cost_delta function

The `cost_delta` function separates all the groups which have pins which are attached to swapped gates. Then it iterates over each group and every pin in the group. It updates the maximum and minimum x and y coordinates for the current group and then calculates the semi-perimeter. Thus,the effective time complexity of this function is $O(P)$ where P is the number of pins.

5.5 perturb_swap function

Under the assumption of minimal collisions in the pseudo-random number generator, the process of selecting two random gates for swapping operates in constant time. Following the envelope swap executed in constant time, a subsequent call is initiated to `cost_delta`.Thus time complexity of function is $O(P)$ where P is the number of pins.

5.6 wire_cost_piazza function

After going through the updated information of wire cost on Piazza, we implemented the corrected wire cost function. Since it calculates cost by making a bounding box for the set of wires : $S_{g_i.p_j} = \{g_i.p_j\} \cup \{g_p.p_q | \exists\, g_i.p_j - -wire - -g_p.p_q\}$ for all such pins , the worse case time complexity goes to $O(P^2)$.

This makes it extremely inefficient to be called during the annealing simulation. Hence we have used our own heuristic throughout during the intermediate calculations , just updating the cost using this at the time of final parsing

It was experimentally observed that our heuristic has a correlation with the the piazza heuristic, as the packing progresses both of them decreases during annealing.

5.7 Time Complexity as a Function of Pins

As expected from the above methods, out time complexity during the one anneal call is $O(P)$ as can be visualised from the graph below (the number of gates was fixed to be 1000 and wires were kept to be 10000) :

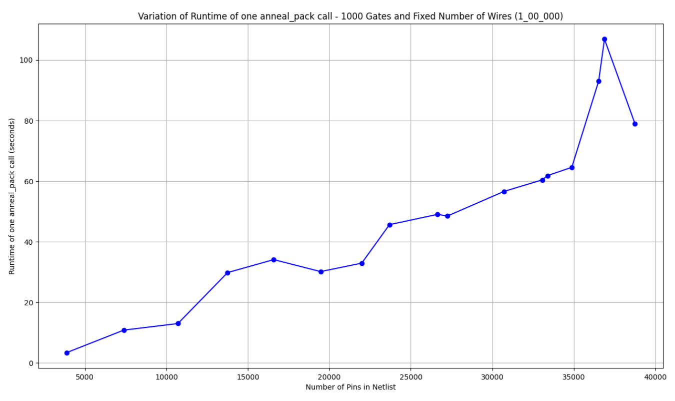


Figure 13: Run-Time vs Number of Pins

Hence by combining observations from up-coming section 5.8 and 5.9 as well as theoretical explanations of `anneal_to_pack` and `cooling_rate`, The Overall time complexity of one Anneal to pack call comes out to be :
$O(P \cdot \alpha \cdot \log_R \frac{T_{min}}{T_O} + P + W + G)$, where P is the number of Pins, G is the number of Gates, W is the number of wires, and $\log_R \frac{T_{min}}{T_O}$ denotes the number of iterations of while loops (based on cooling, R denoting the cooling rate, T_{min} being the minimum temperature and T_O the starting temperature), α denoting the `perturb\_freq` parameter. Analysis is done in a manner in keeping most of the above constant and varying just one parameter. (An additional $O(n^2)$ shows up during `anneal_routine` due to 5.6)

5.8 Time Complexity as a Function of Wires

The runtime is not impacted heavily by the number of wires in our configuration, as demonstrated by the graph and the aforementioned methods. (the number of gates was fixed to be 500 and pins were kept to around 12000) indicating the practical constants involved with G are much less compared to others in big O analysis.

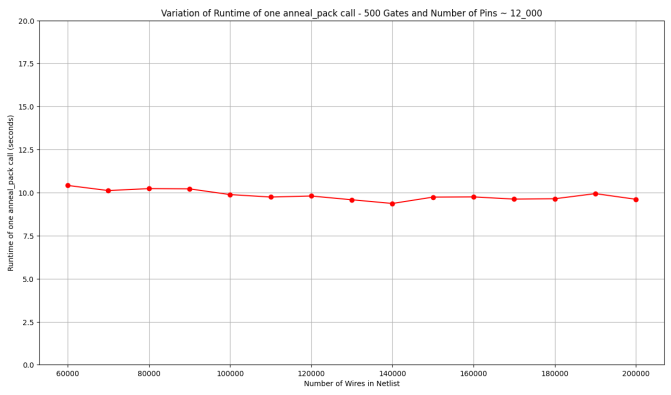


Figure 14: Runtime of anneal_pack vs Number of Wires

5.9 Time Complexity as a Function of perturb_freq Parameter

Using a loop, the `anneal_pack` method executes `perturb_freq` numbers of times. The `perturb_freq` parameter results in a linear increase in runtime, as expected.

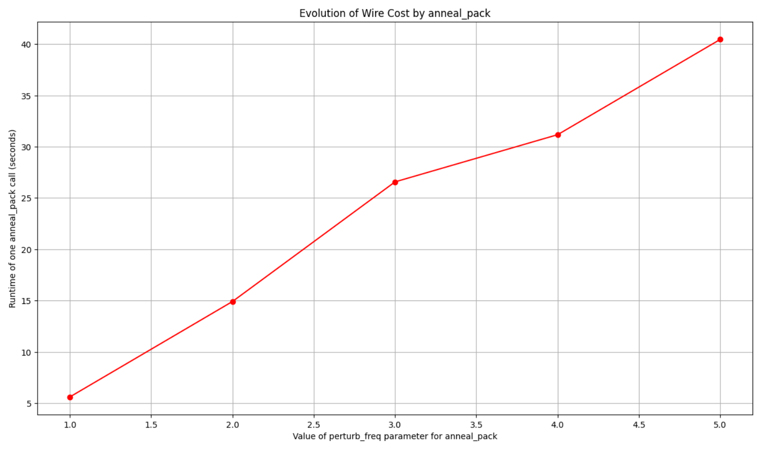


Figure 15: Runtime of anneal_pack vs perturb_freq

6 Wire_Cost Analysis

6.1 Wire Cost Evolution throughout an anneal pack Call

The number of iterations of the while loop inside an anneal_pack call is dependent upon the initial temperature of the system as well as the cooling rate.

On our initial parameters, we get about $\approx$ 1200 calls to perturb logic. The evolution of the wire cost throughout this is show below for a test case : It shows expected behaviour as it initially accepts poor packing in-order to attain a better minima and later on rejects such packing as the "temperature" of the system cools down.

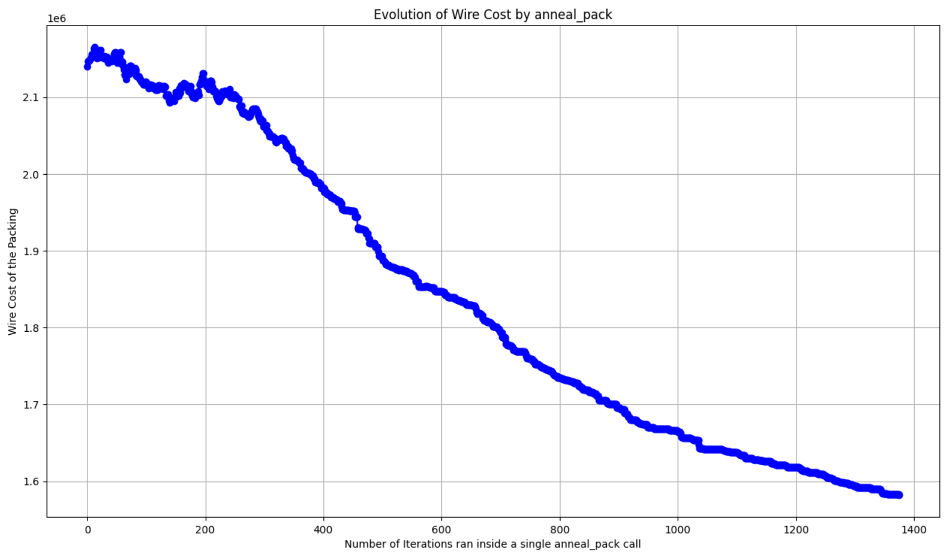


Figure 16: Evolution of wire cost throughout the anneal_pack call

The below graph shows the variation in the cost-evolution as a function of `perturb_freq_per_iter` parameter , which controls the number of times perturb logic is called at a single temperature step. As expected if we allow the system to perturb more, it is able to find better packing, albeit at a linear runtime trade-off explained before.

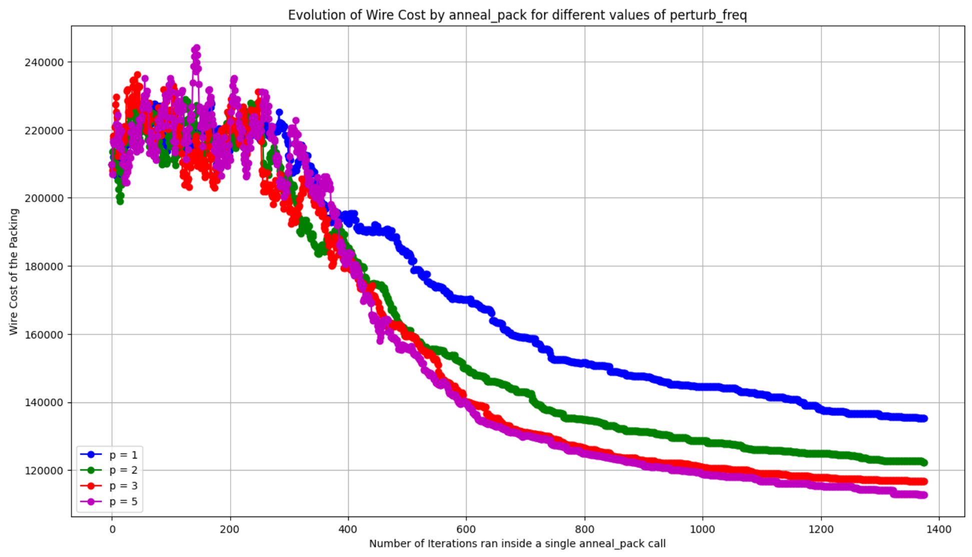


Figure 17: Evolution of wire cost throughout the anneal_pack call

We also ran the simulation for higher values of `perturb_freq_per_iter`. We saw that the further improvements in the cost of the system were not significant as compared to the run-time trade-off.

Thus we carefully chose our perturb `perturb_freq_per_iter` depending on the `select_perturb_freq` helper function available in `utils.py` for the `anneal_routine`.

7 Test Cases

7.1 Moodle Test Cases

7.1.1 Test Case 1

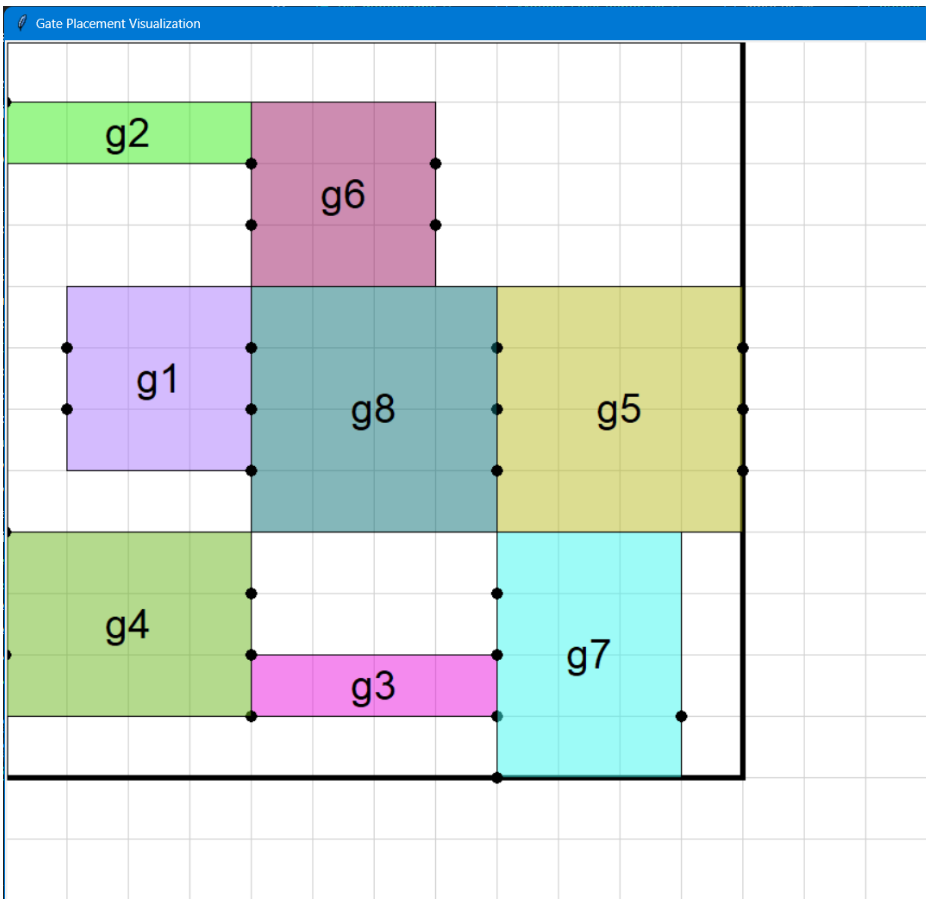


Figure 18: Sample Test Case 1 - Output

Metric for Test Case	Amount
No. of Gates	8
No. of Pins	32
No. of Wires	11
Final Cost	40

Table 1: Details of above Test Case

7.1.2 Test Case 2

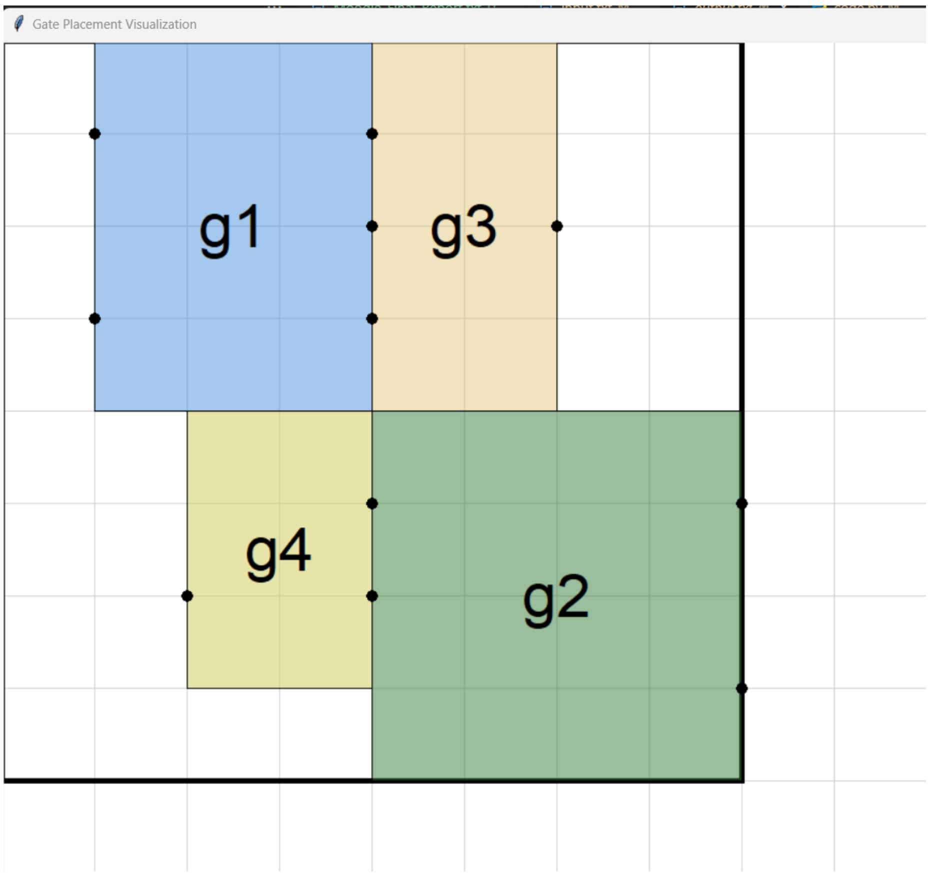


Figure 19: Sample Test Case 2 - Output

Metric for Test Case	Amount
No. of Gates	4
No. of Pins	14
No. of Wires	7
Final Cost	36

Table 2: Details of above Test Case

7.1.3 Test Case 3

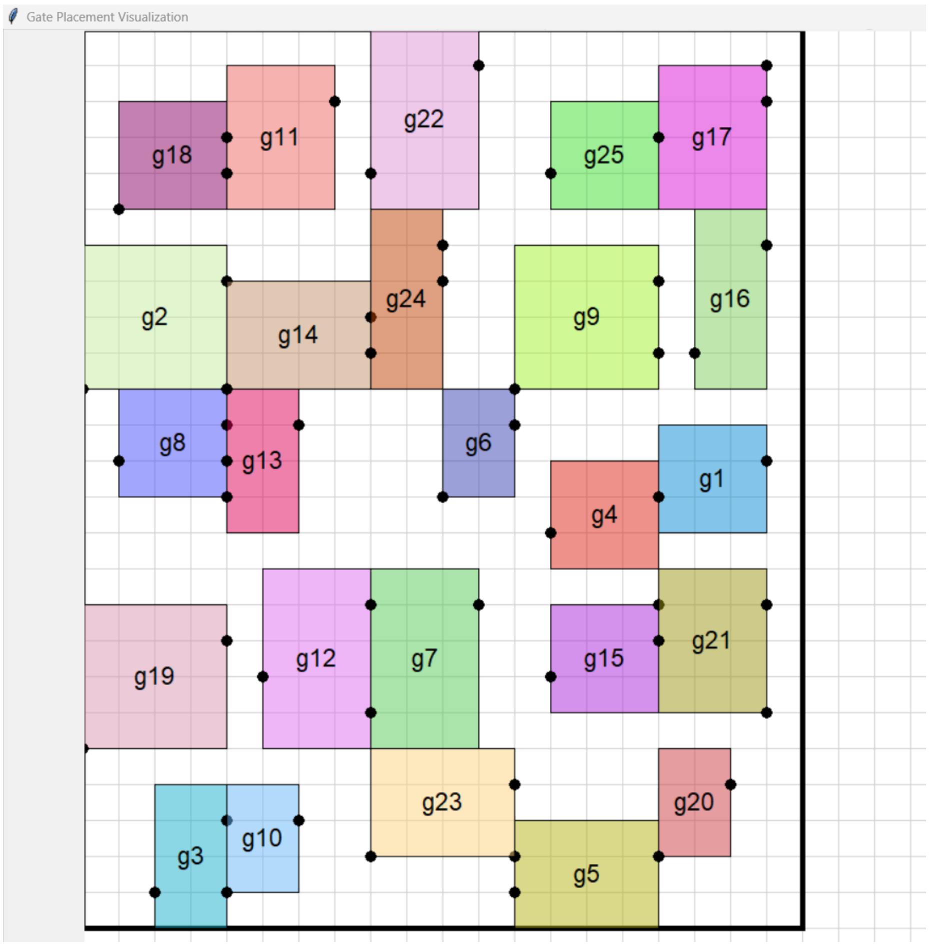


Figure 20: Sample Test Case 3 - Output

Metric for Test Case	Amount
No. of Gates	25
No. of Pins	59
No. of Wires	25
Final Cost	98

Table 3: Details of above Test Case

7.1.4 Test Case 4

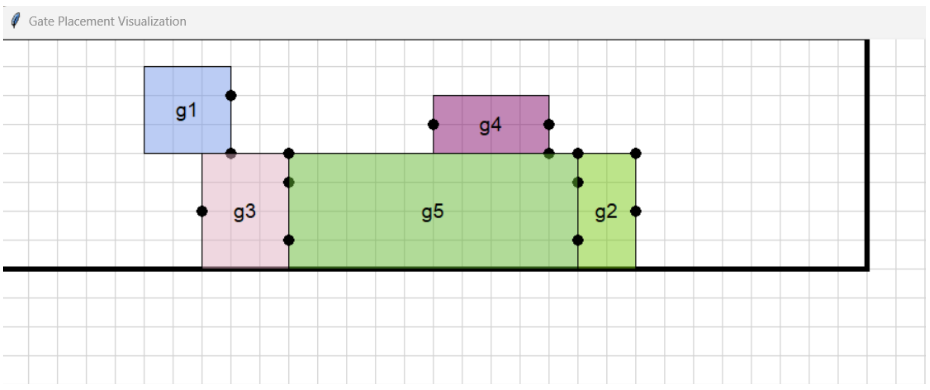


Figure 21: Sample Test Case 4 - Output

Metric for Test Case	Amount
No. of Gates	5
No. of Pins	16
No. of Wires	9
Final Cost	45

Table 4: Details of above Test Case

7.2 Self-Generated Test Cases

7.2.1 Test Case 1

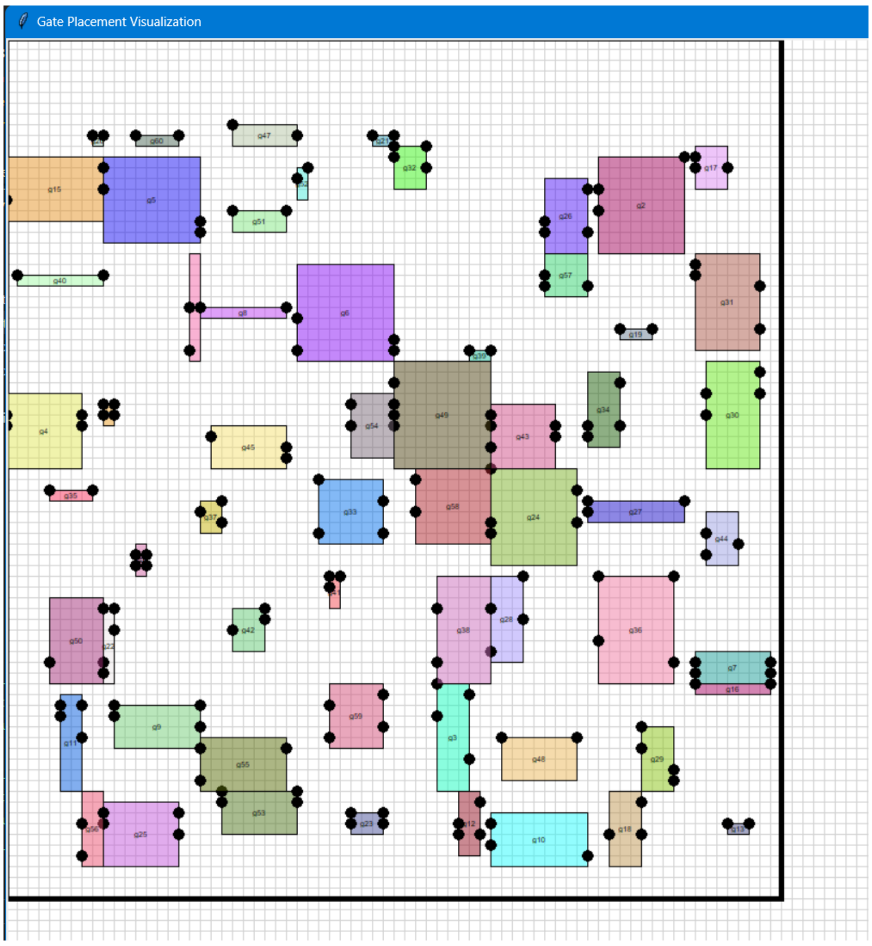


Figure 22: Test Case 1 - Output

Metric for Test Case	Amount
No. of Gates	60
No. of Pins	194
No. of Wires	8946
Final Cost	14534

Table 5: Details of above Test Case

7.3 Self-Generated Test Cases

7.3.1 Test Case 2

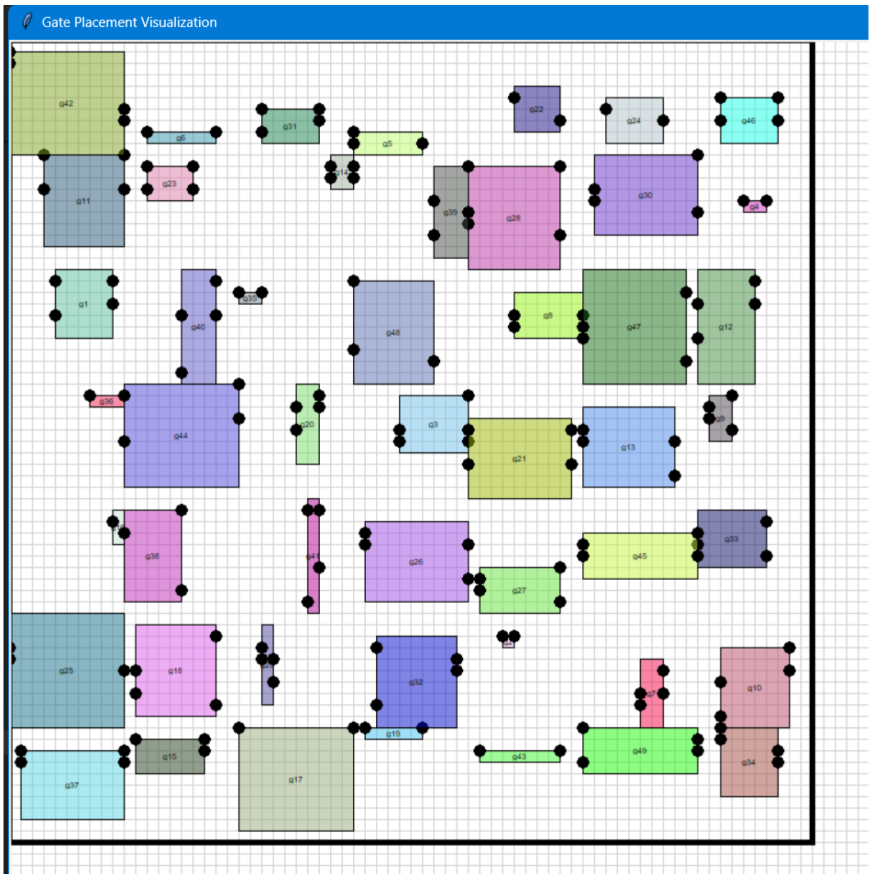


Figure 23: Test Case 2 - Output

Metric for Test Case	Amount
No. of Gates	49
No. of Pins	169
No. of Wires	6951
Final Cost	11109

Table 6: Details of above Test Case

7.4 Self-Generated Test Cases

7.4.1 Test Case 3

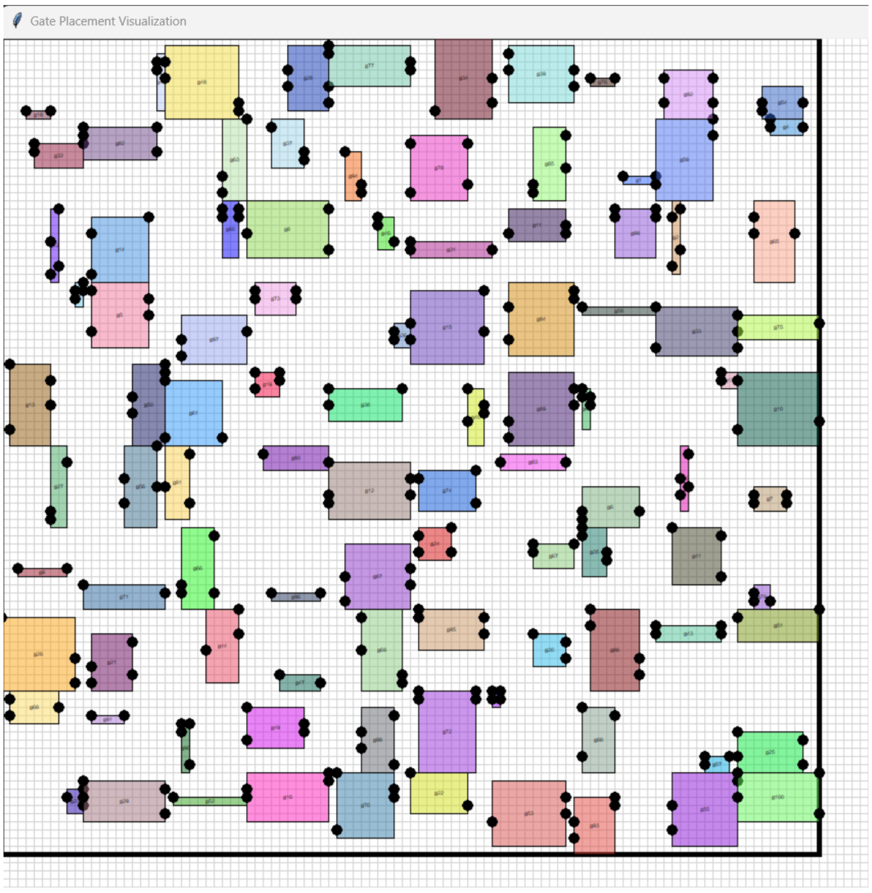


Figure 24: Test Case 3 - Output

Metric for Test Case	Amount
No. of Gates	100
No. of Pins	335
No. of Wires	27070
Final Cost	36116

Table 7: Details of above Test Case

7.5 Self-Generated Test Cases

7.5.1 Test Case 4

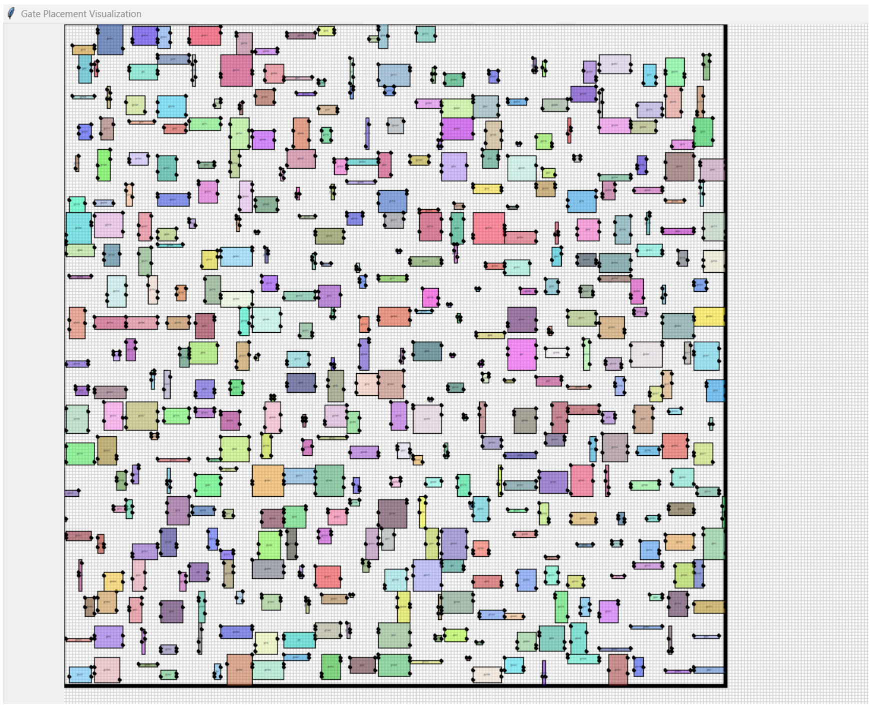


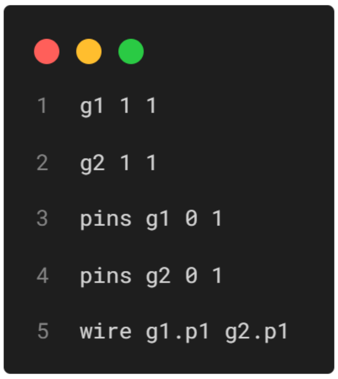
Figure 25: Test Case 4 - Output

Metric for Test Case	Amount
No. of Gates	432
No. of Pins	1448
No. of Wires	51585
Final Cost	338412

Table 8: Details of above Test Case

7.5.2 Edge Cases: Two Gates

The following test cases check the gate packing for various configurations of two unit-length square gates.



```
g1 1 1
g2 1 1
pins g1 0 1
pins g2 0 1
wire g1.p1 g2.p1
```

Figure 26: Input

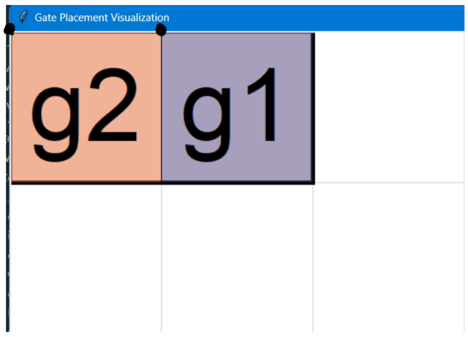
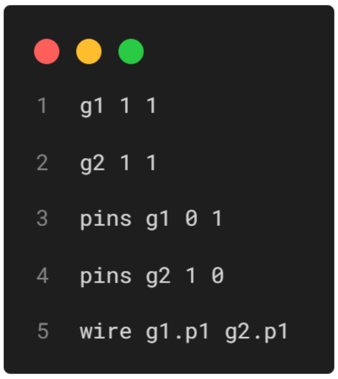


Figure 27: Visualization



```
g1 1 1
g2 1 1
pins g1 0 1
pins g2 1 0
wire g1.p1 g2.p1
```

Figure 28: Input

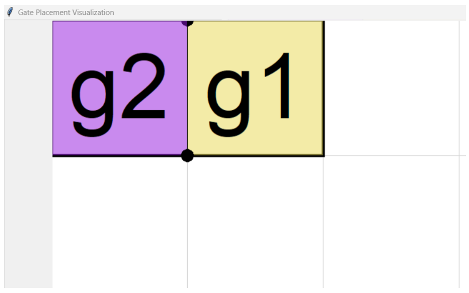
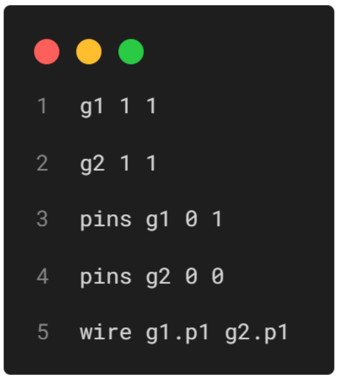


Figure 29: Visualization



```
g1 1 1
g2 1 1
pins g1 0 1
pins g2 0 0
wire g1.p1 g2.p1
```

Figure 30: Input

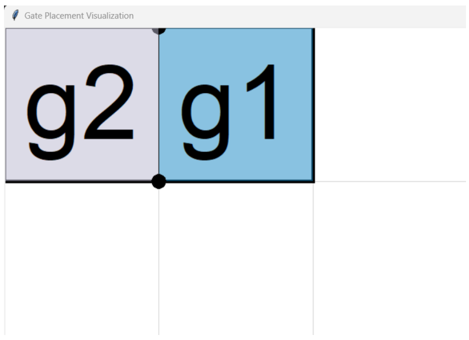
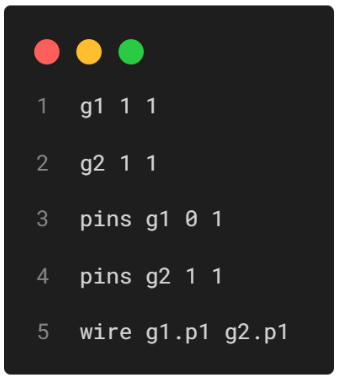


Figure 31: Visualization



```
g1 1 1
g2 1 1
pins g1 0 1
pins g2 1 1
wire g1.p1 g2.p1
```

Figure 32: Input

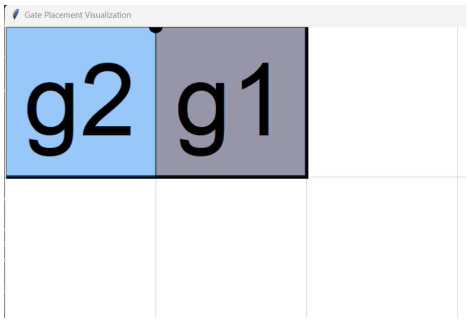
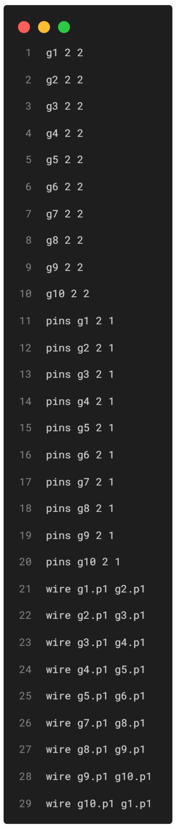


Figure 33: Visualization

7.5.3 Edge Cases: Gates connected in cyclic pattern



```
g1 2 2
g2 2 2
g3 2 2
g4 2 2
g5 2 2
g6 2 2
g7 2 2
g8 2 2
g9 2 2
g10 2 2
pins g1 2 1
pins g2 2 1
pins g3 2 1
pins g4 2 1
pins g5 2 1
pins g6 2 1
pins g7 2 1
pins g8 2 1
pins g9 2 1
pins g10 2 1
wire g1.p1 g2.p1
wire g2.p1 g3.p1
wire g3.p1 g4.p1
wire g4.p1 g5.p1
wire g5.p1 g6.p1
wire g7.p1 g8.p1
wire g8.p1 g9.p1
wire g9.p1 g10.p1
wire g10.p1 g1.p1
```

Figure 34: Input

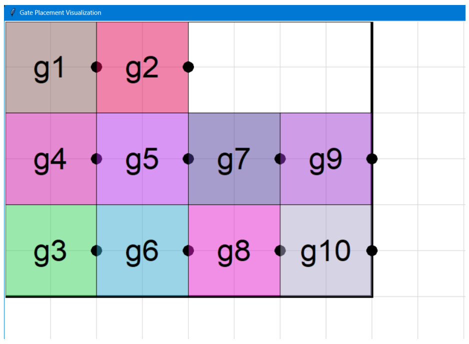


Figure 35: Visualization

Note that since provided `visualisation.py` was working best on smaller number of pins as well as gates, test cases with more challenging conditions could not be attached in the report but they were also tested and are attached in the submission.