SW Assignment-3: Timing Optimisation in Gate Positioning

Given:

• a set of rectangular logic gates g1, g2...gn
• width and height of each gate gi
• the input and output pin locations (x and y co-ordinates) on the boundary of each gate gi .p1, gi .p2, ..., gi .pm (where gate gi has m pins)
• gate delay D_{g1} , D_{g2} ...D_{gn}
• 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 whole circuit is minimised.

Design Decisions:

We have expanded our code in SW-1 to accommodate pin connections.

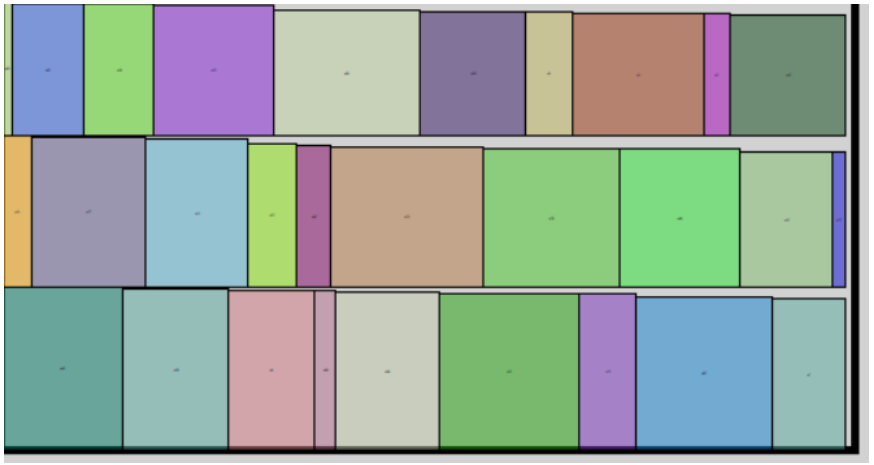
Data structures:

1. Dictionary for storing (width, height) of each gate, i.e., 'g{}' : (width, height). When sorted by decreasing height dictionary is called gh and gw when sorted by decreasing width.
2. Dictionary for storing bottom-left coordinates of each gate, i.e., 'g{}' : (x,y). When sorted by decreasing height, dictionary is called gxyh and gxyw for decreasing width.
3. Nested list conn for storing connection between gates as [['g{}', 'p{}'], ['g{}', 'p{}']]
4. Dictionary g_conn for storing each gate's connection with the other gate as {'g{}': {'g{}', ….. and so on}}
5. Dictionary delay to store individual gate delays.
6. Function wire_length to calculate Manhattan distance between 2 pin connections [['g{}', 'p{}'], ['g{}', 'p{}']].
7. Function total_wire(conn) to calculate total wire length.
   Input: [[['g1', 'p1'], ['g2', 'p2']], [['g2', 'p2'], ['g3', 'p3']], ……., [['gn', 'pn'], ['gm', 'pm']]]
8. Class gate is defined to store all the information of specific gate,
   i.e., final x-coordinate = x + width of gate; final y-coordinate = y + height of gate.
9. Class bdd is defined to dynamically create bounding box as gates are added. It computes final coordinates (by taking the maximum of final coordinates of gates), area of the bounding box and total_wire computed by summing all wire_lengths (from function).
10. Class Level is defined to keep track of the level at which the gate is placed. It stores x0 and y0 (starting of level), h (height of level) and w (width of level). We also added function add_rect() to store gates at each level and remove_rect() to remove gates at each level.
11. Function swap_levels(l1, l2) that swaps levels l1 and l2.
12. Function swap_gates_in_level(g1, g2, l) that swaps gates g1 and g2 in certain level l.
13. Function gate_connectivity() that returns dictionary containing the count of how strongly 2 gates are connected.
14. List paths to store all possible paths from input pin to output pin.
15. List paths_conn to store connections between two gates.
16. List only_conns to store connections between two gates excluding connections between pins in a gate.

In our implementation, we first designed the initial case (carried from SW-1, but does not place rectangle on top of another rectangle in a level).
The algorithm sorts the rectangle in decreasing height.

1. Sort by decreasing height (in case heights of gates are equal, sorted by decreasing width)
   The gate with the maximum height is placed at (0, 0). The first level begins at lh= 0. Hence, height of level lh is taken to be the maximum height of gates at that level. The next gate is placed to the right and so on as the coordinates of the next gate are fixed at the final x-coordinate and initial y-coordinate of the previous gate.
   When no more gates can be fitted at that level, a new level lh+1 is created at the top of the previous level and the process restarts.

Capacity of each level: A new level is created when the final x-coordinate of the gates exceeds the square root of the minimum possible area of bounding box i.e. the sum of all the gate areas. We chose this value after many guesses for optimal level capacity, e.g. max(gate widths). For a large number of gates it tries to 'square' the bounding box area, hence adjusting both height and width of the final bounding box.



The paths list is created by iteration over wire connections (conn).

1. We created primary_inputs and primary_outputs list, wherein pins that are unconnected but belong to gates that are present in the input gates but absent in the output pins are deemed to be primary_inputs while gates in the output gates but unconnected otherwise become primary_outputs.
2. After that, we iterated over connections and created a new path (list) beginning with one of the multiple primary_inputs. Connections with the same input pin are appended. Then we iterated over wire connections again and appended new connections to the path if the output_pin gate corresponded with the input_pin gate of another connection. The path is terminated when the output pin gate corresponds with one of the primary_outputs, and it ends with the addition of this primary_output. This path is appended to main list paths.

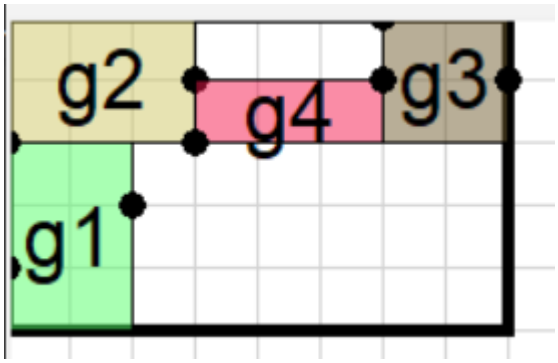
3. We created paths_conn list to store connections between two gates selected at a time. Also, we created only_conns list to store connections wherein connections between the same gate are not included.
4. This is for function total_wire(conn) that takes in nested loop of two gate connections and calculated Manhattan distance between two gates, sums all values and returns total wirelength.

Wire length is minimised when gates are placed as compactly as possible.
So, for rearrange() function,

- First, we iterated over dictionary conn. Then we accessed gates g1, g2 from each connection.
- If g1 belongs to Level l1 and g2 belongs to Level l2, then we swapped levels such that gates connected to each other get closer. E.g., swapping (l1+1) and l2, or (l2-1) and l1. This is done before the third step.
- We defined wire_priority that accesses gate connectivity value. If it is greater than 1, then we place g2 to the right of the last element in level l1. Then, we swap g2 with an element next to g1. This process takes place only when gate connectivity is high, or more than the average value of gate_connectivity count.
- Accordingly we adjusted the level and gate positions.

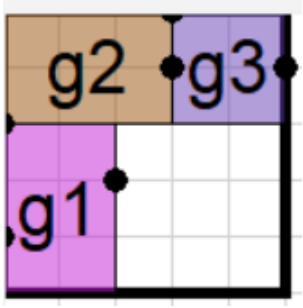
After rearranging for every connection, we implemented function critical() to find out which path causes maximum delay.

Test cases:
For 1st test case:



critical_path g1.p1 g1.p2 g3.p1 g3.p3 g4.p2 g4.p3
critical_path_delay 39

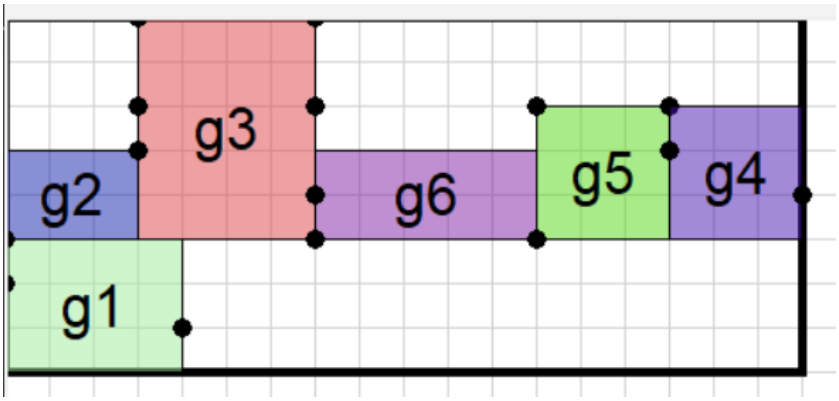
For 2nd test case:



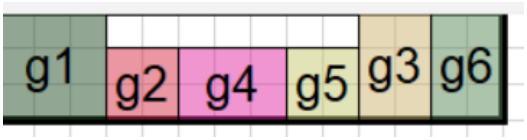
critical_path g1.p1 g1.p2 g3.p1 g3.p3

critical_path_delay 23

For 3rd test case



bounding_box 18 8
critical_path g1.p1 g1.p2 g3.p2 g3.p5 g6.p1 g6.p2
critical_path_delay 23
For 4th test case:



bounding_box 14 3
critical_path g1.p1 g1.p2 g2.p1 g2.p2 g3.p2 g4.p2 g5.p1 g5.p2 g6.p2 g6.p1
critical_path_delay 34

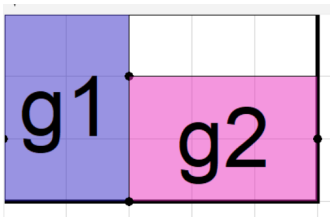
Generated test cases
Corner cases:

1. 1 gate

g1 2 3 5
pins g1 0 1 2 2
wire_delay 3
wire g1.p2 g1.p1

bounding_box 2 3
critical_path g1.p2 g1.p1
critical_path_delay 5
g1 0 0

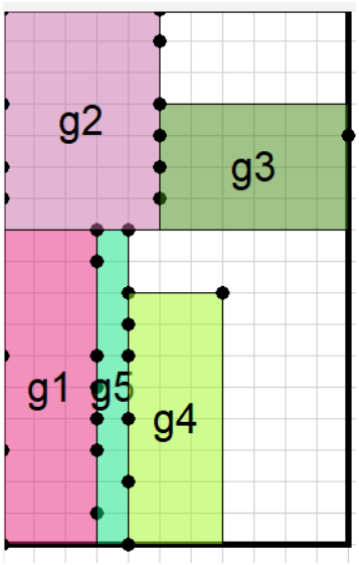
2. 2 gates



g1 2 3 5
pins g1 0 1 2 2
g2 3 2 3
pins g2 0 0 3 1
wire_delay 3
wire g1.p2 g2.p1
wire g2.p2 g1.p1

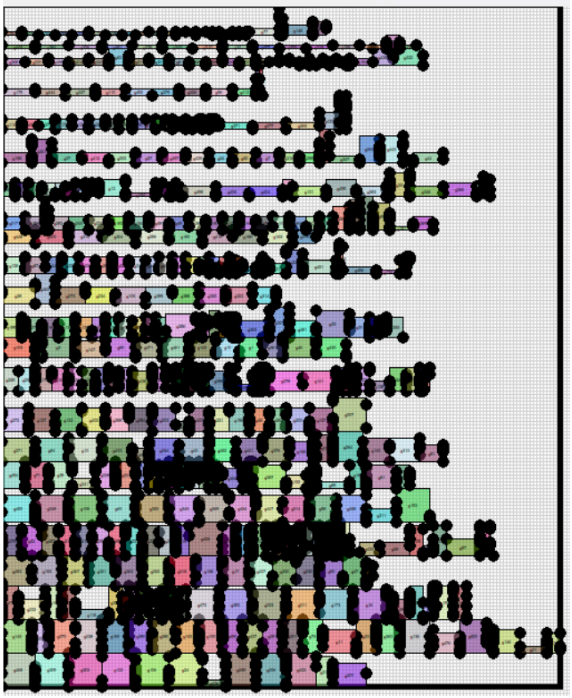
bounding_box 5 3
critical_path g2.p2 g1.p1
critical_path_delay 23

For 5 gates



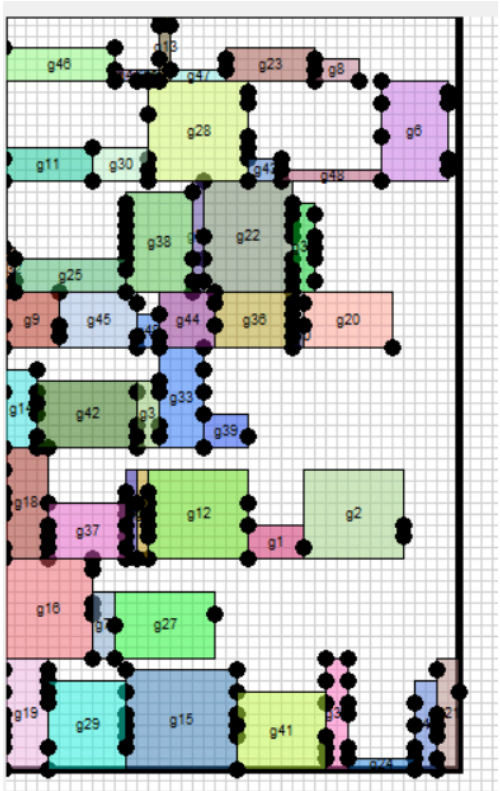
bounding_box 11 17
critical_path g1.p5 g1.p2 g4.p1 g4.p2
critical_path_delay 121

For 500 gates:



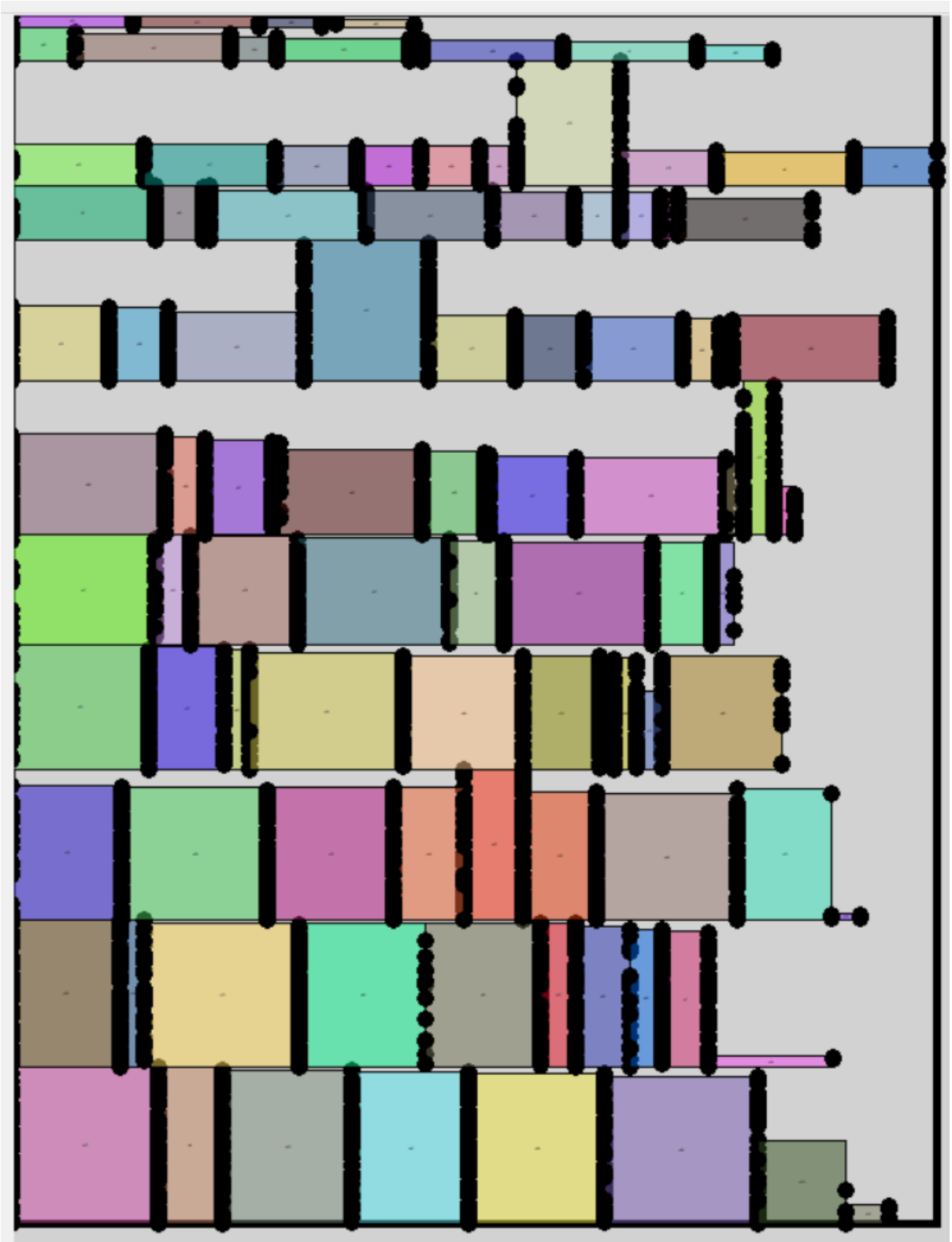
bounding_box 167 203
critical_path g490.p10 g490.p1 g125.p2 g125.p5 g411.p2 g411.p3
critical_path_delay 1600

For 50 gates:



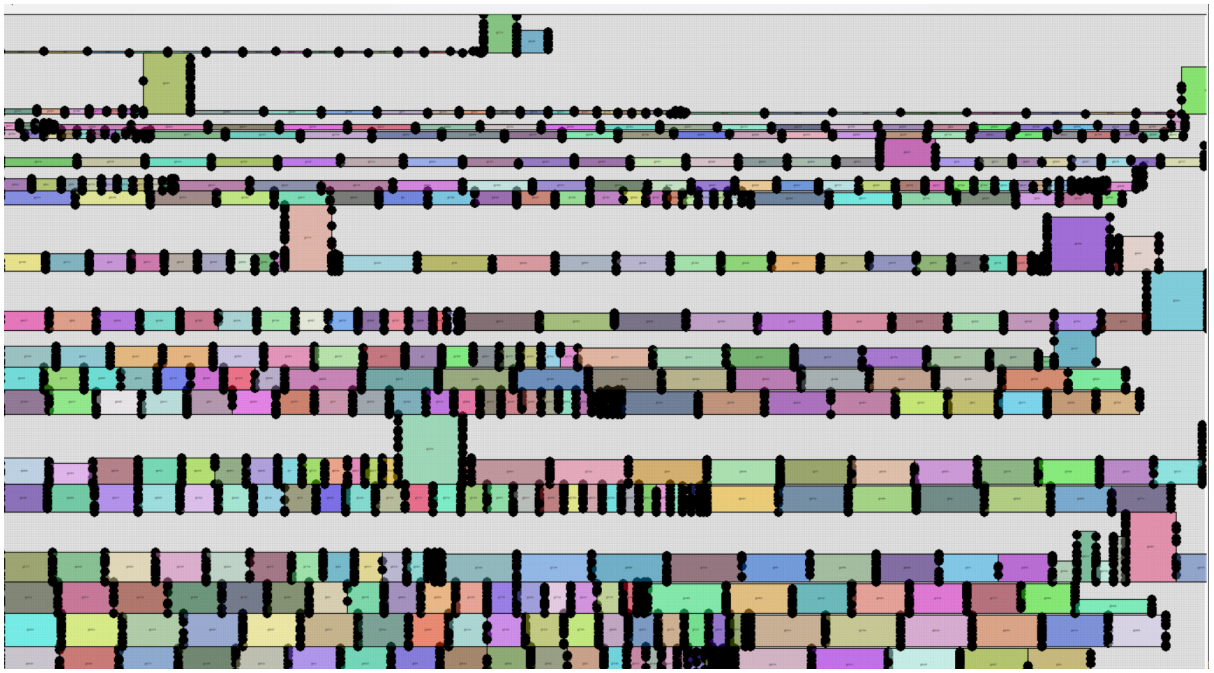
bounding_box 41 68
critical_path g21.p5 g21.p3 g35.p2 g35.p1 g34.p4 g24.p1 g24.p2
critical_path_delay 168

For 100 gates:

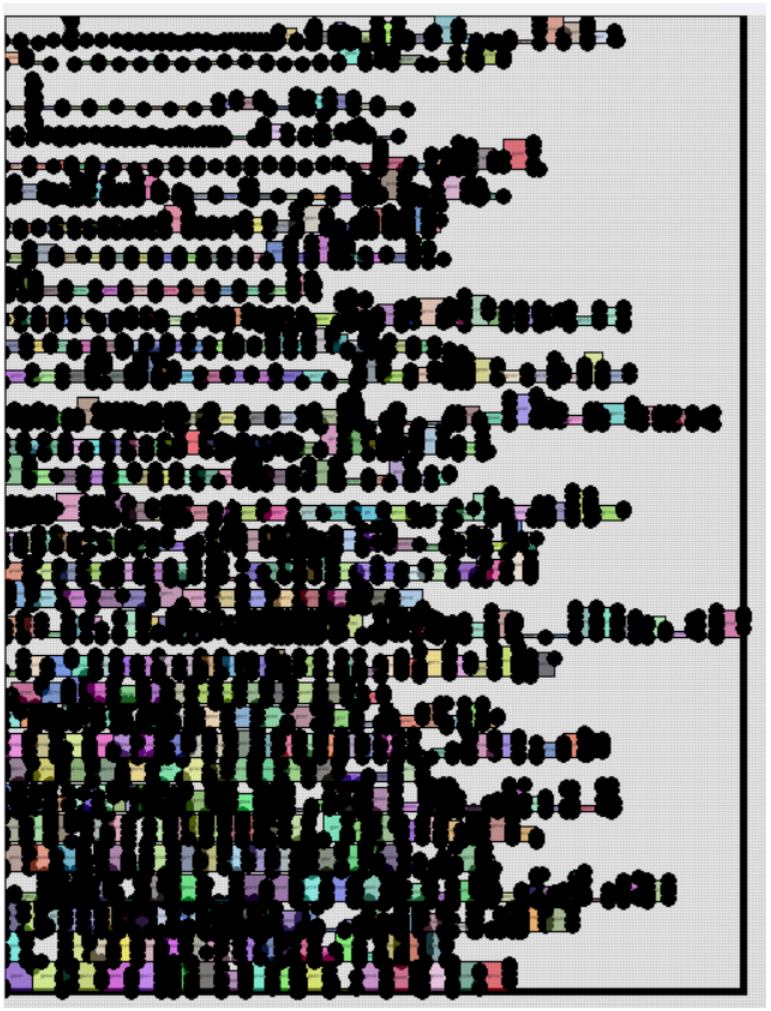


bounding_box 579 757
critical_path g64.p49 g64.p7 g51.p2 g51.p1 g72.p8 g72.p13 g93.p6 g93.p1
critical_path_delay 1174
For 1000 gates (with width and height ranging from 1 to 100)

bounding_box 782 938
critical_path g41.p25 g41.p6 g725.p1 g725.p2 g1000.p4 g1000.p1
critical_path_delay 5440



For 1000 gates (with width and height ranging from 1 to 10)



bounding_box 247 308
critical_path g421.p4 g421.p2 g832.p6 g832.p4 g488.p2 g506.p1 g913.p3 g913.p1
critical_path_delay 1691

There is no overlapping in any case.

For generating test cases we created a random text file generator randomtxt.py

```
import random
with open('input_15.txt', 'w') as f:
    n=5
    lst=[]
    pins={}
    for i in range(1, n+1):
        w=random.randint(1, 10)
        h=random.randint(1, 10)
        lst.append('g'+str(i))
              f.write('g'+str(i)+'         '+str(w)+'         '+str(h)+'
'+str(random.randint(1, 20))+'\n')
        s='pins '+'g'+str(i)+' '
```

```
        no=random.randint(1, h)
        n1=random.randint(1, h)
        coord=[]
        pins['g'+str(i)]=[]
        for j in range(1, no+1):
            y=str(random.randint(0, int(h)))
            if str(0)+' '+y not in coord:
                pins['g'+str(i)].append('p'+str(j))
                s+=str(0)+' '+y+' '
            coord.append(str(0)+' '+y)

        for j in range(1, n1+1):
            y=str(random.randint(0, int(h)))
            if str(int(w))+' '+y not in coord:
                pins['g'+str(i)].append('p'+str(j))
                s+=str(w)+' '+y+' '
            coord.append(str(int(w))+' '+y)

        f.write(s+'\n')

    conn=random.randint(1, n)
    f.write('wire_delay '+ str(random.randint(1, 10))+'\n')
    g=random.sample(lst, 2)
    memo=[g]
    for i in range(conn):
        g1=random.sample(lst, 2)
        if g1 in memo:
            continue
        memo.append(g1)
        memo.append(g1[::-1])
           f.write('wire '+ g1[0]+ '.'+ str(random.sample(pins[g1[0]],
1)[0])+' '+g1[1]+'.'+str(random.sample(pins[g1[1]], 1)[0])+'\n')
```

Time complexity analysis:

- N is the number of gates.
- M is the number of connections between gates.
- L is the number of levels.
- G is the number of gates in each level

For initial case (where gates are placed according to logic implemented in SW-1), the time complexity is O(N log N).

For appending all possible paths to list paths, the worst case time complexity is $O(N^2)$, since we iterated over the wire connections twice, and then over primary inputs and outputs.

For rearrange() function, we iterated over the dictionary conn, which in the worst case scenario has N(N-1)/2 elements and N elements in the best case scenario. Hence, let $O(M) \sim O(N^2)$. Inside this for loop we iterate over levels (L) and gates (G) after that. L and G are inversely related, hence we can include either one in the time complexity analysis. For worst case scenario, L ~ N (G ~ 1) or G ~ N (L ~ 1).

Worst case time complexity comes out to be $O(M*N) \sim O(N^3)$