COL215 Software Assignment 2: Wiring-aware Gate Positioning Report

Submitted by:
Harshit Kansal: 2023CS10498
Aadl Govil: 2023CS10490

Design decisions:

Overall:
We divided the gates into components connected to each other. IE if G1 is connected to G2, G3 and G4 to G5 but there is no connections between {G1,G2,G3} and {G4,G5}, we have evaluated them independently as no wire exists between them thus evaluating independently won't cause any problems.

Increased modularity of code as compared to first assignment to help with easy manipulation, combination, additions, etc to the code.

Algorithms working(These algorithms work on each connected component of gates independently)

Algorithm1: Greedy packing

The algorithm works by first selecting the gate with the most number of connections(by checking the number of gates connected to each gate), after that, we place that gate at the coordinates (sum of widths, sum of heights) to allow for sufficient expansion in every direction. Whenever we place a gate, we add the points occupied by the gate in a set of occupied points and the corner points of the gate in a set of points to be checked when placing new gates. Whenever we place a new gate, we try to place it in every possible position in the list of points maintained for trying to place the gates. For each position we try to place the gate in four different ways such that one of its corners touches the point we are trying to place it at. We check overlap; if there is no overlap we check the sum of semi-perimeters of all the pins of that gate, and we choose the position with no overlap and the least sum of perimeters to place the gates. To place the gates, we follow an order such that after placing the gate with most connections we place all the gates connected to it, then we place all the gates connected to those gates we placed earlier, for doing this we use a queue data structure. After we are done with a connected component of gates(gates which are connected with each other and not with other gates). We shift it so that the bottom most gate starts from y = 0 and the left most gate starts just after the previous connected component of gates. Some notable design decisions are as follows:

1) Each gate is placed on the boundary of the already existing region, so for checking overlap we maintain a set of all occupied points and need to only check if any point on the layer just beneath the perimeter is occupied, because for any overlap to occur the overlap must pass through the perimeter as it is placed on the boundary of a connected region.
2) For picking the optimal position and calculating and updating the semi perimeter we maintain dimensions of bounding boxes of all the pins and just extend those bounding boxes with the pins of the gate we are trying to place making it efficient.

3) By using a queue, we make sure that we are trying to place gates connected with a gate close to it.

Algorithm 2 : Clustering and Greedy Square packing

Design Decision 1:Usage of Algorithm from 1st assignment, with certain modifications to use greedy square packing

The reason we decided to do square packing was basically as we are minimising the semi-perimeter of the bounding boxes for a set of connected pins, and if we pack the rectangles into a square, we can say we are effectively packing it in a way such that maximum area is fitting into a rectangle of the smallest perimeter. Hence the idea. This was achieved by limiting height iteration in the algorithm of the first assignment to certain values and using semi_perimeter as the function to be maximised instead of area.

Design Decision 2: Clustering

Another idea we had was clustering the graphs into small clusters sequentially. IE if there are 1000 gates, we divide them into clusters of say 4, and thus we have 250 clusters of 4 gates each. We efficiently pack the gates within each of the clusters. Then we take these 250 clusters and divide them into clusters of 10 each, and say get 25 such clusters of clusters. We then square pack the 10 clusters within each clusters of clusters. And then we square pack these 25 clusters together to give final output for 1000 gates.

The reason for this decision was to place gates which are connected more proximately, to get a better ordering than just square packing on a 1000 gates. This was backed by testing we did, which showed that indeed such clustering improved performance over normal square clustering.

Design Decision 3 : Clustering Algorithms

Through some googling we figured out that the clustering of graphs in itself is an NP Hard Problem, with different approaches. Thus we used one approach according to the internet and one approach to make a similar clustering algorithm ourselves.

Label Propagation:

Assign each node its own label, then for each node in the graph, figure out the label that occurs the most in its neighbours, and we do this for many iterations. We hope to get a final answer that has clustered graphs in a network of gates.

Adjacent Clustering:

Take all the given gates and partition it using a random algorithm which makes clusters by searching in all the gates connected to a certain gate and choosing max_cluster_size amount of gates with a higher priority given to gates having more number of connections to the first gate.

Time Complexity Analysis:

n = number of gates
p=number of pins
c=number of components
W = Number of wires
w = maximum width of gates
h = maximum height of gates

Algorithm1: Greedy packing

For doing the time complexity analysis, let's try to understand the time taken by each function individually and then do the entire time complexity analysis.

1) Init : In the init function we are just initializing some data structures,it takes O(1) time
2) set_pin_groups(): For each pin which is an input pin it goes over all the output pins and assigns them this input pin, since each pin can take input only once, maximum that many iterations will be there, so Tc is O(number of pins) in worst case
3) find_gate_with_most_connections(): We are just iterating over gates, so it takes O(n) time.
4) place_first_gate(): Here we are iterating over all the pins of the gate, calling functions like gate with most connections, set pin groups and update occupied points and bounding box, time taken of the order of maximum of these functions, which is set_pin_groups() in worst case which takes O(pins) time.
5) update_bounding_box_and_occupied(): We are iterating over the width and height of the gate so time complexity is O(wh)
6) is_overlap(): We are just iterating over the width and height of the gate in different loops so time complexity is O(max(w , h))
7) calculate_semi_perimeter(): in this, we are iterating over each pin of the gate and adding semi perimeter for each of the bounding boxes(at max 2), which takes it to be about O(pins per gate)
8) try_placement(): In the we iterate over each point in boundary points which are of the order of n since for each gate we are adding 4 points, in that loop we are calling is_overlap and calculate_semi_perimeter() functions, the time taken will be n times the maximum of time taken by is overlap and calculate_semi_perimeter, which is O(n*max(pins per gate , max(w , h))), pins per gate can be at max h, so tc is O(n*h) in worst case.
9) update_pin_group_bounding_box(): It involves iterating over all the pins of a gate and updating its bounding boxes, which takes about O(pins per gate) time
10) greedy_packing(): The main time-consuming part of this is the try_placement() method, which takes about O(n*h) time, and we are iterating over all gates in a queue and calling this, so it takes O(n^2*h) time
11) greedy_multiple(): we call the greedy_packing function on all the gates connected components, in the worst case, there is one connected component so it will take O(n^2*h) time.

Greedy_multiple is called for using this algorithm, which has a time complexity of **O(n^2*h)**

*Algorithm 2 : Clustering and Greedy Square packing O(30 * depth*c*(g+W+sum_of_heights*n^2logn))*

In this approach we have taken the best out of a certain number of iterations on both clustering functions, while changing parameters. I shall find the time complexity for each of these iterations.
In every such iteration:

1. Init : O(1) - we just define some literals
2. Read_input : O(n+p+W): Reading all pins, wires, gates take O(n+p+W) amount of time at least. Add to that the time required to make a pins dictionary, and time to take out connected components of gates, we get an additional O(p+n+W) time, which results in a total complexity of O(n+p+W).
3. Clustering_recursion: c*TC(recursive_clustering) + O(n) + O(w) = O(depth*c*(g+W+sum_of_heights*n^2logn))

For every connected component we call recursive_clustering once. We also set up the x,y of every gate so that adds another O(n), and using calculate wire in the end adds another O(w).

4. Recursive_clustering(): O(depth * (g+W+sum_of_heights*n^2logn))
   Runs for utmost depth no. of iterations, which we have defined to be 1000 maximum.
   In every iteration, we run:
   i) Clustering algorithm:
      a) Label_propogation : O(g+W) * 100
         We see all gates of the connected component and check all its connections to find the maximum label
      b) Adj_clustering : O(W + log(g))
         We see every gate if it has not been visited and add to a priority queue, and take out constant number of elements resulting in a W+log(g) time complexity
   ii) Optimal_packing() : O(sum of heights*n^2*logn)
      Complexity of algorithm from first assignment, can be inferred from there itself
   iii) Modifications of gates and wires: O(g+w):
      We are iterating over the total gate and wires a couple of times, which leads to added O(g+w) complexity.

Testing:

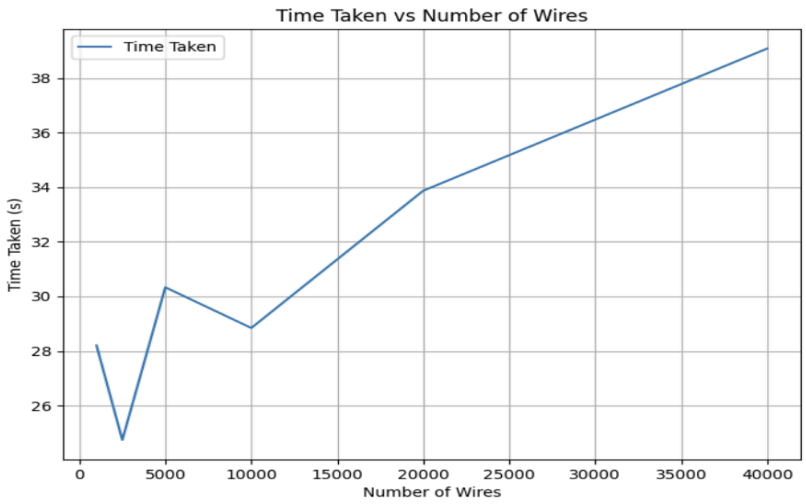
For testing, we tried to change various parameters such as number of gates, maximum number of pins per gate, number of wires and the mean of the dimensions of the gate and plotted the graphs for both algorithms.
After that, we ran our algorithms for some cases with different merits, like very large number of pins, small gates, etc. and showed the results of both the algorithms for all of these case

Test cases 0-5: Varying no. of wires

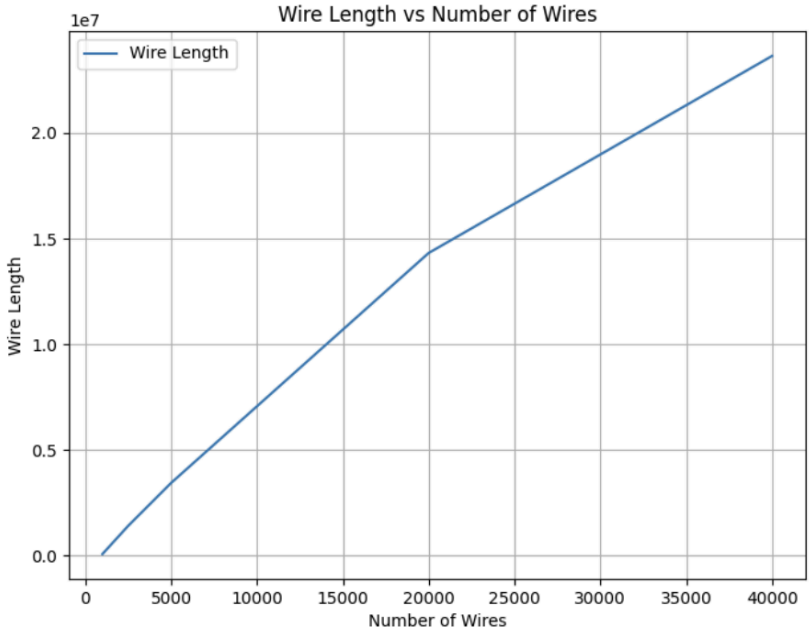
We kept the number of gates as 1000, the max pins of gates as 80 and chose a random number of pins from 2 to 80, the max dimensions of gates as 100, and randomly chose dimensions between 1 and 100 and varied the number of wires from 1000 to 40000, almost doubling each time.

Algorithm 1:

Time taken vs No of wires:

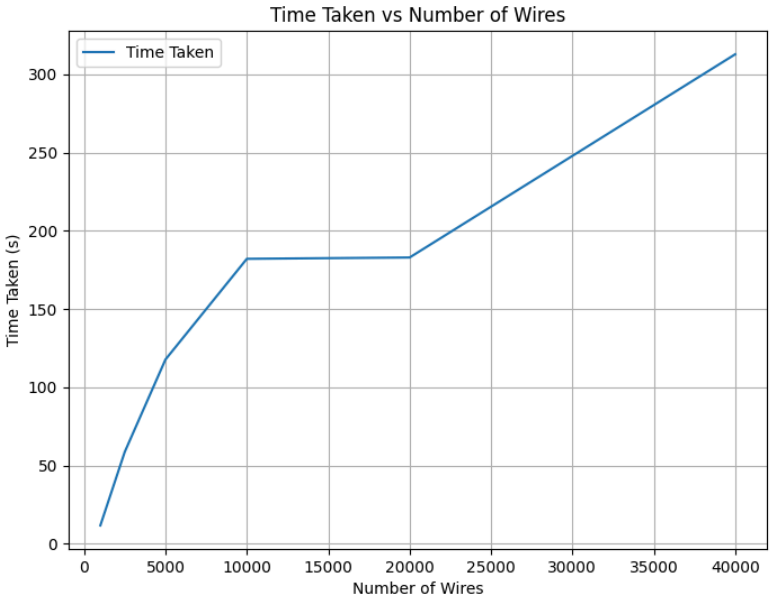


Wirelength vs No of wires:

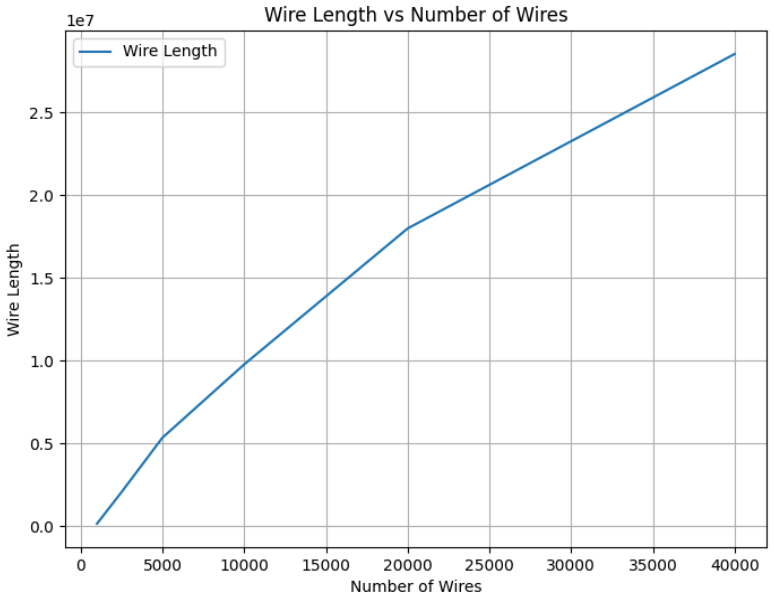


Algorithm2:

Time taken vs number of wires



Wire length vs number of wires

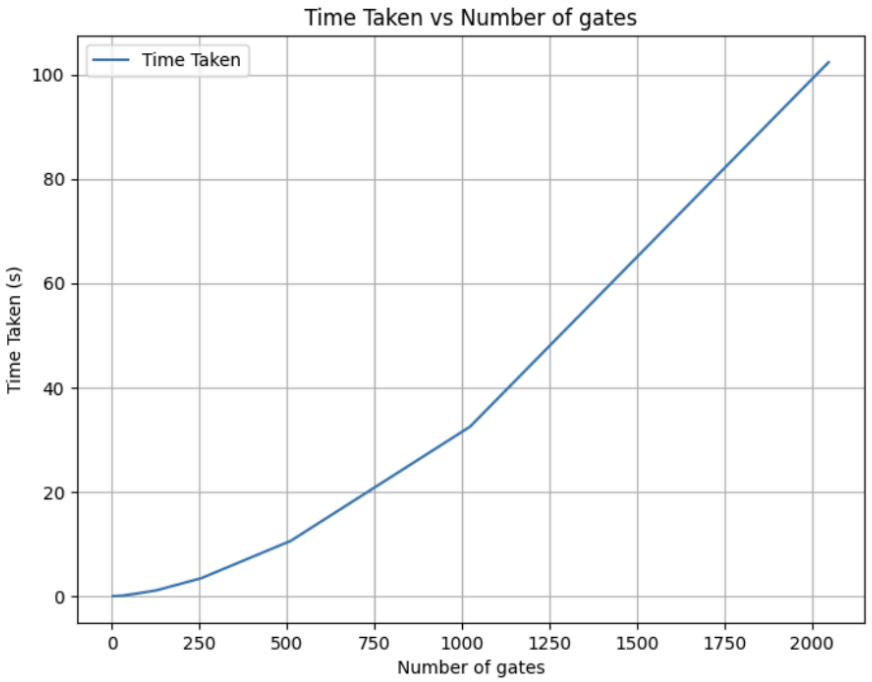


Test cases 6-15: No. of gates

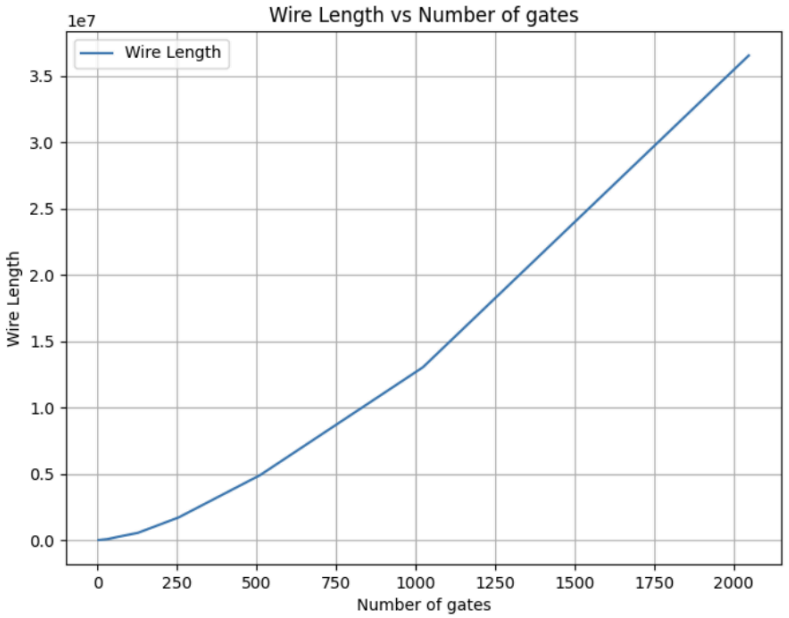
We fixed the maximum number of pins per gate as 40, dimensions for gates are random between 1 and 100, we vary the number of gates from 4 to 2048, doubling each time, we also double the number of wires each time we double the number of gates to have similar connections each time.

Algorithm 1:

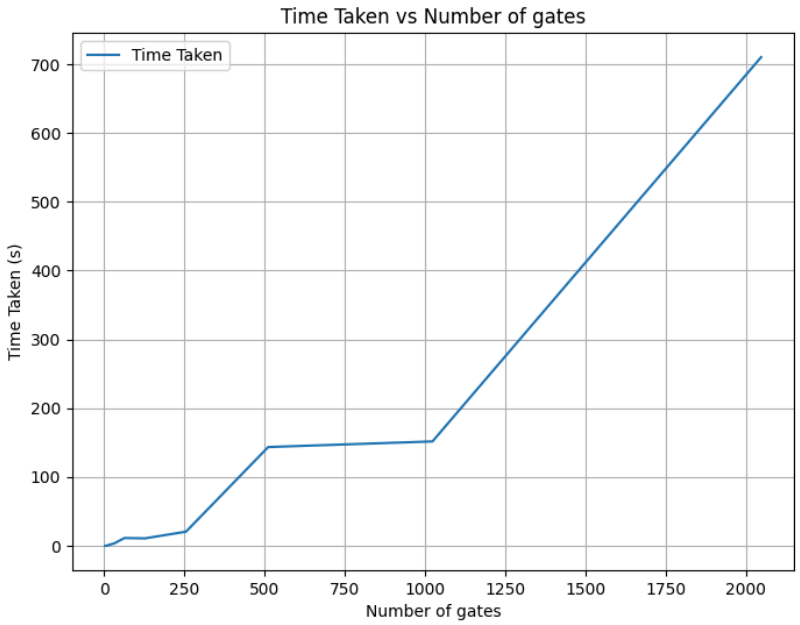
Time taken VS No. of Gates



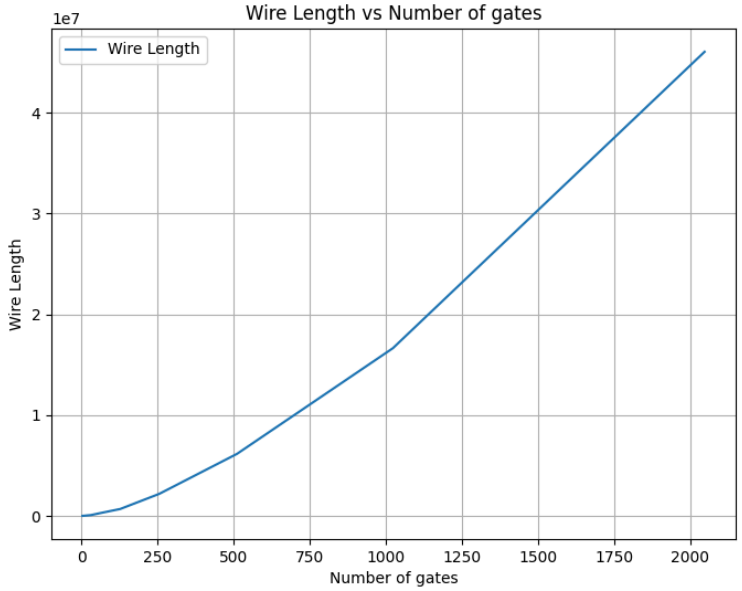
Wire Length vs No.of Gates:



Algorithm 2: Time taken vs number of gates



Wire length vs number of gates

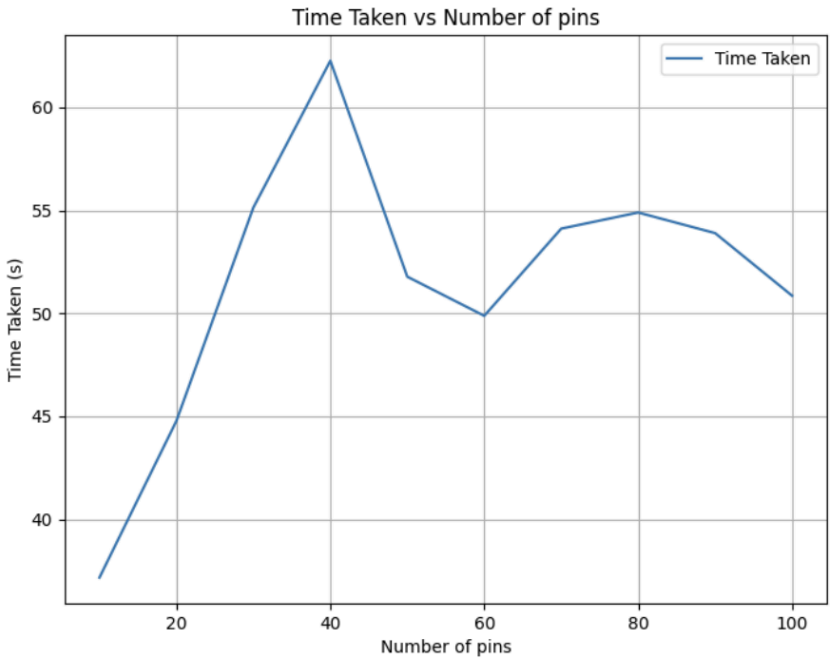


Test Cases 16-25: Varying No. of pins

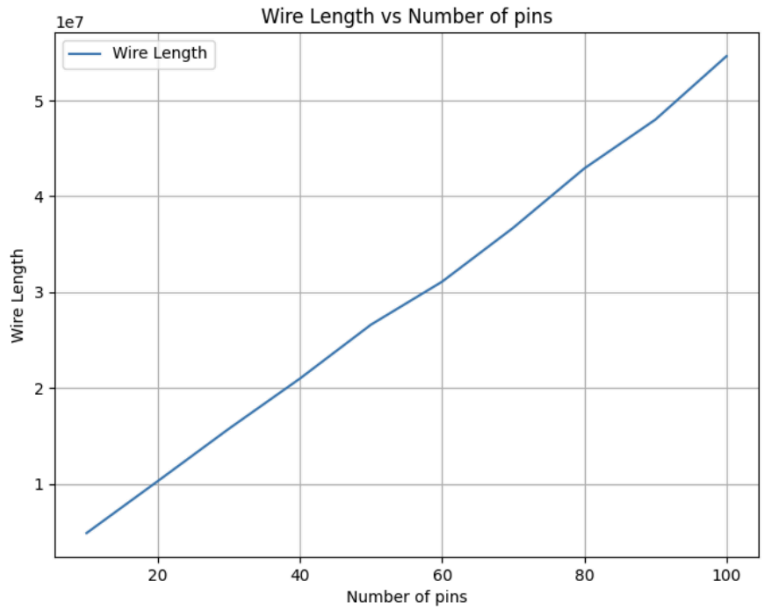
Here, we fix the number of gates as 1000, keep the dimensions as a random number between 50 and 100, vary the number of pins from 10 to 100 in intervals of 10, we also increase the number of wires as we increase the number of pins to have similar number of connections between the pins.

Algorithm 1:

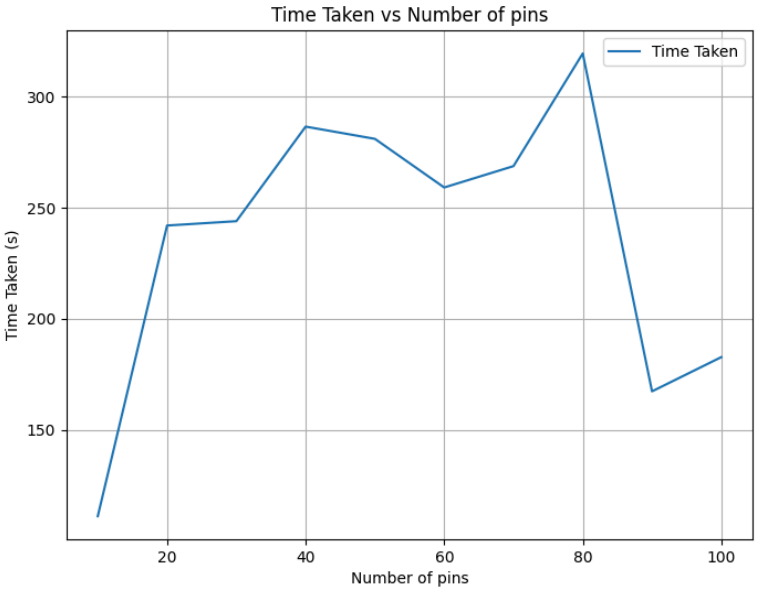
Time Taken Vs No. of pins:



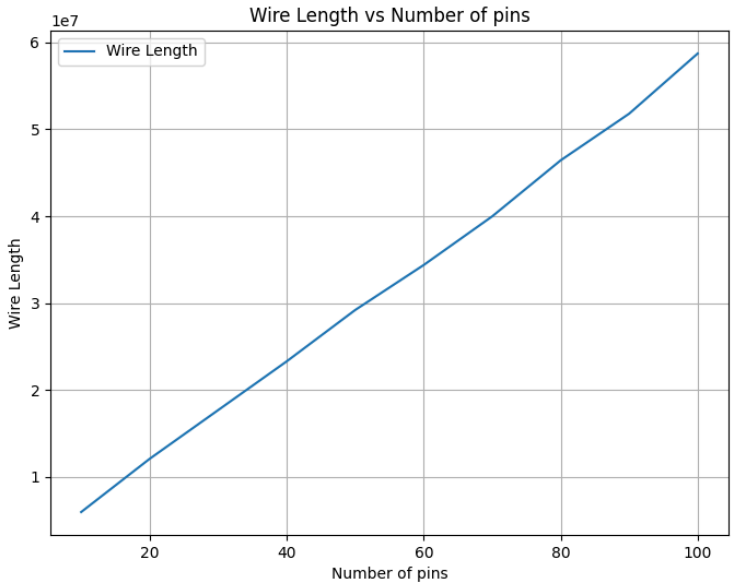
Wire Length Vs No. of pins:



Algorithm 2:
Time taken vs number of pins



Wire length vs number of pins

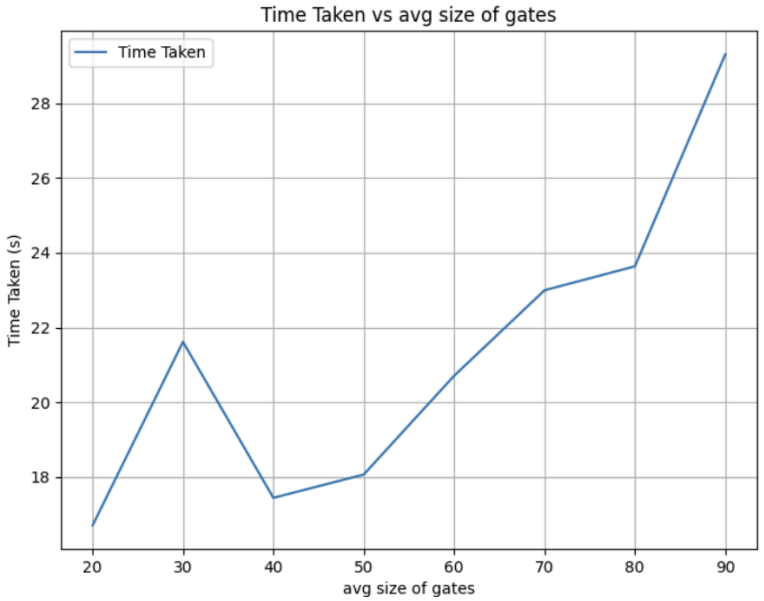


Test Cases 26-33: Varying Size of gates:

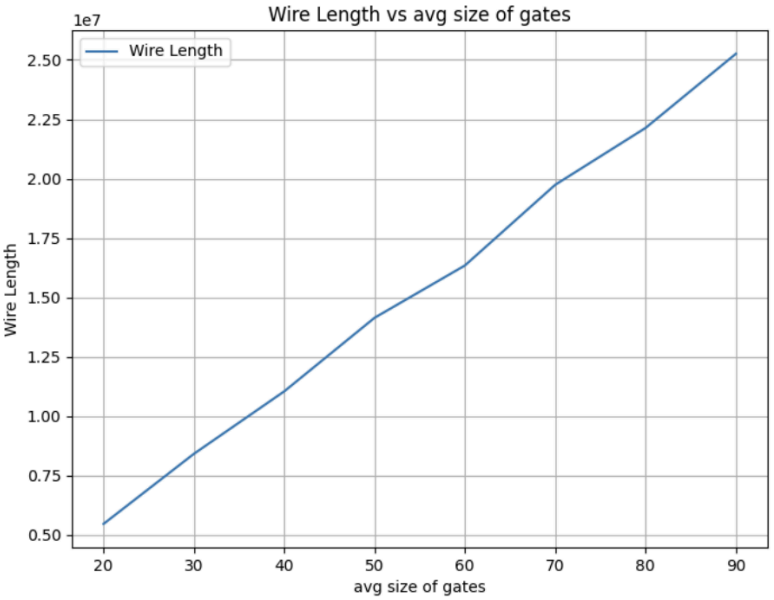
Here, we keep the number of gates as 1000, max. Number of pins per gate as 40, number of wires as 20000 and try to vary the avg size of the gates to see variation with sizes of gates. We vary the sizes from 10-30, 20-40 and so on till 80-100

Algorithm 1:

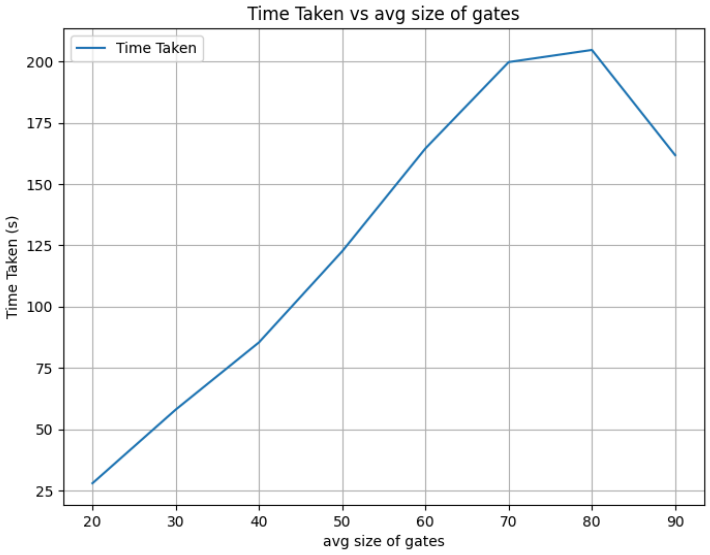
Time Taken VS Avg. size of gates:



Wire Length vs Avg. size of gates:



Algorithm 2
Time taken vs avg. size of gates



Wire length vs avg. size of gates

Test cases 34 and 35: Here, we try to have an exact number of pins per gate rather than selecting a random number of pins till a certain amount as in the cases earlier. Test case 34 has 1000 gates, 40 pins per gate, dimension random from 1 to 100, 40000 wires and test case 35 has 1000 gates, 10 pins per gate, dimension random from 1 to 1000, 10000 wires. We have kept cases with a lot of and somewhat less amount of pins.

Test case 36: Here we have kept the sizes of gates very small and each gate has maximum number of pins(pins on each point possible), gate dimensions are from 1 to 10, 1000 gates, 20000 wires

Test case 37 and 38: These test cases have a lot of pins for bigger gates, test case 37 has 1000 gates of sizes between 80 and 100, and each has 40 pins, so total pins are 40000 and there are 40000 wires. For test case 38, there are 1000 gates of sizes between 80 and 100 with 100 pins each, making 100000 total pins (much more than the given constraints) and there are 100000 wires.

Test cases 39-43: These test cases test the algorithms on gates of different sizes with almost max pins for each gate. Test case 39 has 100 gates with sizes between 80 and 100, a maximum number of pins per gate(close to 200), and around 20000 wires. Test cases 40-43 also have 100 gates with a maximum number of pins and wires almost equal to a total number of pins in each, we are decreasing the range of sizes from 80-100 to 60-80, 40-60 and so on until 1-20.

Test cases 44 and 45: Here, we take 1000 small gates(sizes at random from 1 to 10), in one case, we take a random number of pins to 40, which will make most gates have a pin and every possible spot, and some gates with lesser pins, in the other we take a random number of pins till 10.

The results for the test cases are shown below in the form of wire length and then the time taken.

Algorithm1 : Greedy

Test case 34: 27545339 48.34311842918396
Test case 35: 6548062 33.41367316246033
Test case 36: 1189039 18.75363039970398
Test case 37: 57308989 57.930023193359375
Test case 38: 146763089 77.2291886806488
Test case 39: 8671543 3.2183103561401367
Test case 40: 5177954 2.506685495376587
Test case 41: 2569040 1.665933609008789
Test case 42: 964345 1.1550543308258057
Test case 43: 108194 0.424360990524292
Test case 44: 923198 17.43991732597351
Test case 45: 360069 15.959906101226807

Algorithm 2:

Test case 34: 32408316 148.8607439994812
Test case 35: 8718472 120.50190377235413
Test case 36: 1388926 8.64633059501648
Test case 37: 60466773 149.99879717826843
Test case 38: 150363080 187.38080549240112
Test case 39: 8891277 14.263546705245972
Test case 40: 5324953 11.802931308746338
Test case 41: 2676208 13.597596406936646
Test case 42: 1004540 4.9552998542785645
Test case 43: 125800 1.6493537425994873
Test case 44: 1120227 11.546499013900757
Test case 45: 505743 7.061613321304321

Final Conclusions:

We have compared 2 algorithms, and in one of the algorithms we have also compared 2 clustering functions:

<u>Comparison between clustering functions:</u>

Label propagation is the better clustering function in terms of wire length produced, but takes a lot more time to run than one iteration of the Adj clustering algorithm. But for the Adj_clustering algorithm we need to iterate over the max_cluster_size so the time taken becomes equivalent only. The only benefit provided by the Adj Clustering algorithm comes in the case where we are taking smaller

number of gates. In our testing TC 6,7,8,9,11,42, all of which had number of gates<=150 were the only ones which showed better results for Adj_Clustering over Label Propagation.

Comparison between algorithm 1 and algorithm 2:

Through extensive testing as done above we have been able to see that in none of the 45 test cases used do we get a better wire length for algorithm 2, and also algorithm 2 takes much more time to run than algorithm, barring cases in which there are a small number of gates, in which case the running times are comparable. Hence we have chosen to use algorithm 1 in our main file, given its time efficiency as well as better wire lengths produced by it.