EC440 VLSI CAD

Project (181EC120) Documentation

Release 2.0

Karthik R Rao

November 21, 2020

Contents

Packages Imported

The complete assignment was implemented on Python3. Some of the additional packages used were-

1. NetworkX: a Python package which is very helpful in the creation and manipulation of complex graphs.
2. Matplotlib: helpful for visualizing graphs.
    a. pyplot
    b. patches
    c. colors
    d. collections
3. Math
4. NumPy: efficient array manipulation.
5. Queue: module used in routing to implement a queue data structure.
6. Abstract Syntax Trees (ast): used to read Python dictionaries from text files efficiently.

Parsing

The Kernighan-Lin algorithm for bi-partitioning, Simulated Annealing for floor planning and placement, and Lee's algorithm for routing are implemented on a 4-bit adder circuit (ISCAS-85 74283 Fast Adder Circuit). This 4-bit adder has 9 inputs, 5 outputs and 36 gates. The netlist of the circuit can be obtained here. This file is downloaded with a .isc extension.

This netlist is in text format. But the KL Algorithm requires the input to be a well-defined graph with vertices and edges. The **parsing.py** converts this netlist in text form to an undirected graph. This is achieved in two steps-

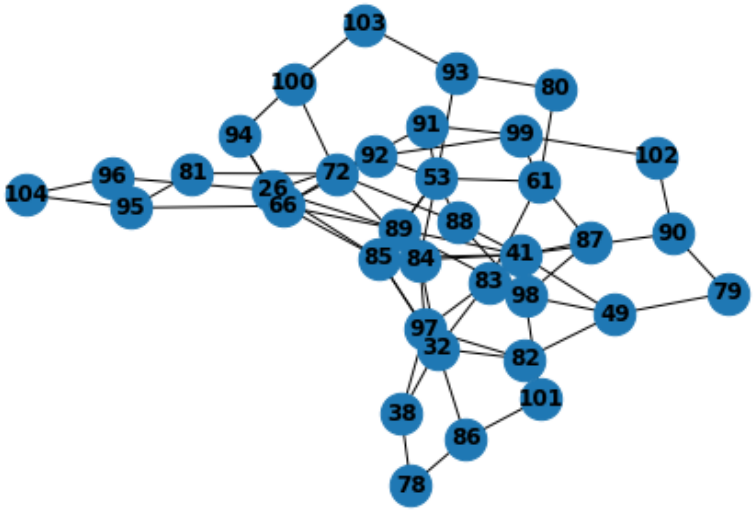
1. Create a graph using brute force utilising the connections mentioned in the .isc file. According to the ISCAS-85 Netlist Format, there are 10 types of nodes- *inpt, not, and, or, nand, nor, xor, xnor, buff, from*. In this step, even the *from* and *inpt* nodes are treated as vertices in the graph and an intermediate graph is obtained. There are 104 vertices in this graph.
2. However, for feeding the graph to the KL Algorithm, we require the number of vertices in the graph to be equal to the number of gates in the circuit. This can be achieved by contracting the intermediate graph such that *inpt* and *from* nodes are merged into one of the neighbouring nodes. The end graph has only gates as the vertices in the graph, i.e., it has 36 vertices.

Details about **parsing.py**:

- Input: Netlist with .isc extension
- Outputs: An undirected graph without weights.
    - **graph_nodes.txt** contains nodes in the graph.
    - **graph_edges.txt** contains edges in the graph. An entry *a, b* is made if there exists an edge between vertex *a* and vertex *b*.
    - **heights.txt** contains approximate heights of each gate. This along with **widths.txt** will be of use during the floor planning and placement stages.
    - **widths.txt** contains approximate widths of each gate.
    - **inputs.txt** contains the input nodes to the adder circuit. (can be used later for pin assignment)

Obtained outputs and graphs of **parsing.py**:

```
In [1]: runfile('D:/fifth-sem/vlsi-cad/assignment/parsing.py', wdir='D:/fifth-sem/
vlsi-cad/assignment')
The number of nodes in the graph are: 36

The nodes in the graph are:
['26', '32', '38', '41', '49', '53', '61', '66', '72', '78', '79', '80', '81',
'82', '83', '84', '85', '86', '87', '88', '89', '90', '91', '92', '93', '94',
'95', '96', '97', '98', '99', '100', '101', '102', '103', '104']

The number of edges in the graph are: 69

The edges in the graph are:
[('26', '85'), ('26', '89'), ('26', '92'), ('26', '94'), ('26', '96'), ('32',
'38'), ('32', '82'), ('32', '83'), ('32', '84'), ('32', '85'), ('32', '86'),
('38', '97'), ('38', '78'), ('41', '49'), ('41', '83'), ('41', '84'), ('41',
'85'), ('41', '87'), ('41', '88'), ('41', '89'), ('41', '90'), ('49', '82'),
('49', '98'), ('49', '79'), ('53', '61'), ('53', '84'), ('53', '85'), ('53',
'88'), ('53', '89'), ('53', '91'), ('53', '92'), ('53', '93'), ('61', '83'),
('61', '87'), ('61', '99'), ('61', '80'), ('66', '72'), ('66', '85'), ('66',
'89'), ('66', '92'), ('66', '94'), ('66', '95'), ('72', '84'), ('72', '88'),
('72', '91'), ('72', '100'), ('72', '81'), ('78', '86'), ('79', '90'), ('80',
'93'), ('81', '95'), ('82', '97'), ('83', '97'), ('84', '97'), ('85', '97'),
('86', '101'), ('87', '98'), ('88', '98'), ('89', '98'), ('90', '102'), ('91',
'99'), ('92', '99'), ('93', '103'), ('94', '100'), ('95', '104'), ('96', '104'),
('98', '101'), ('99', '102'), ('100', '103')]
```



Partitioning

Two-way Partitioning is done using the Kernighan-Lin (KL) algorithm. The KL algorithm takes an undirected (and unweighted in this implementation) graph as input and returns two sets of vertices such that the number of edges crossing the partitions is minimum.

This is achieved by taking a random initial partition and iteratively improving the solution by swapping vertices across the two partitions. This swapping process is determined by how much the current cutsize (number of edges crossing the partition boundary) will decrease after every swap. At every swap, the pair with the largest decrease/ smallest increase in cutsize is chosen. Once swapped, they are 'locked', meaning they should not be considered in the future swapping process. The rest of the vertices undergo the same process and when all the vertices have been swapped, we say that one 'pass' has finished. At the end of each pass, we accept the first 'k' moves which had led to the minimum cutsize during the entire pass.

The KL algorithm is as fast as a Greedy algorithm, but it also has the ability to accept some worse moves in search of a better local minima. This is the reason why the KL heuristic is so popular.

Details about **partitioning.py**:

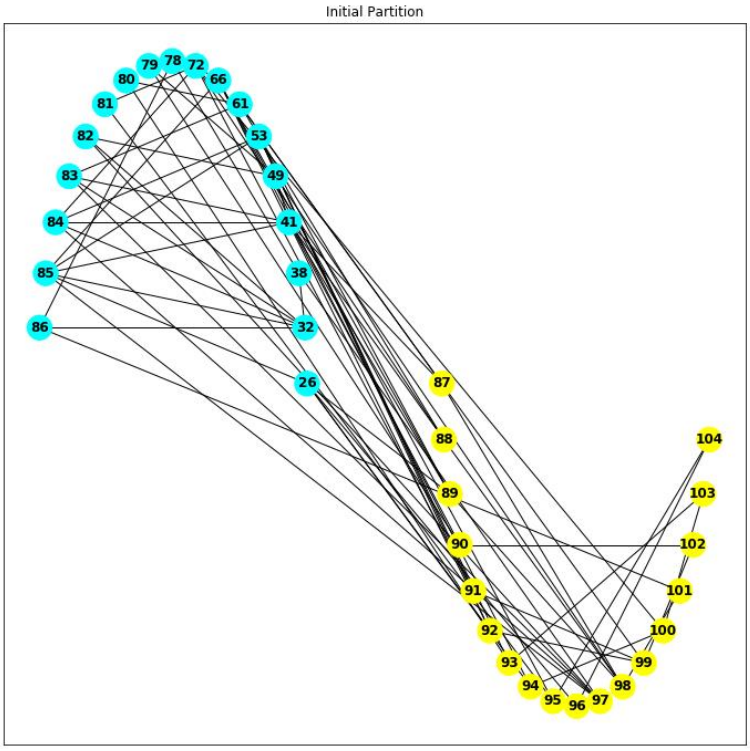
- Inputs: Information about the graph.
  - **graph_nodes.txt** has the nodes of the netlist.
  - **graph_edges.txt** has the edges of the netlist in the form *a, b* where *a* and *b* are nodes in the graph.
- Outputs:
  - Nodes in initial partition, initial cutsize.
  - Nodes in final partitions, final cutsize. This information is stored in **partition1.txt** and **partition2.txt**.

The following functions briefly explain the implementation:

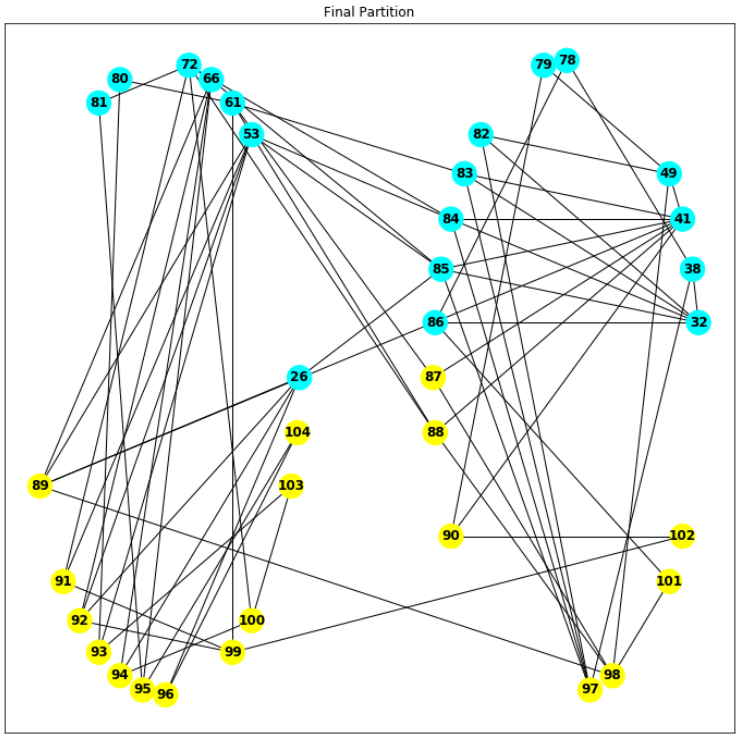
- construct_graph (): Given the nodes and edges of a graph, construct a NetworkX graph.
- get_init_partition (): Given a list of nodes, get two partitions of almost equal sizes.
- kernighan_lin_bisection ():
  - compute_cutsize (): Calculates the cutsize, i.e., the number of edges crossing between the two partitions.
  - computeD (): Computes the difference between the external costs (number of edges across partitions) and internal costs (number of edges within the same partition) of a given node. For a node *s*, $D = E_s - I_s$.
  - compute_gain (): Computes the reduction in cost of a possible swap. For two nodes *a* and *b*, the equation is given by, $G = D_a + D_b - 2*c_{a,b}$, where $c_{a,b}$ is the cost of possible edge between *a* and *b*.
  - swap_from_list (): The pairs corresponding to maximum gain is swapped.
- write_partitions (): Write the final partitions to text files.

Obtained outputs and graphs of **partitioning.py**:

```
In [44]: runfile('D:/fifth-sem/vlsi-cad/assignment/partitioning.py', wdir='D:/
fifth-sem/vlsi-cad/assignment')
Reloaded modules: construct_graph, kernighan_lin_bisection, util, write_partitions
Initial Partition1: ['26', '32', '38', '41', '49', '53', '61', '66', '72', '78',
'79', '80', '81', '82', '83', '84', '85', '86']
Initial Partition2: ['87', '88', '89', '90', '91', '92', '93', '94', '95', '96',
'97', '98', '99', '100', '101', '102', '103', '104']
Initial Cutsize: 32


Final Partition1: ['89', '61', '80', '93', '103', '99', '91', '53', '100', '72',
'81', '95', '104', '96', '92', '26', '94', '66']
Final Partition2: ['85', '88', '102', '84', '79', '90', '87', '98', '101', '78',
'86', '38', '49', '41', '82', '32', '83', '97']
Final Cutsize: 12
```



Initial cutsize = 32



Final cutsize = 12

Floor Planning

Floor planning is done using the Simulated Annealing algorithm. Simulated Annealing is a powerful heuristic that can be used when a global optimum is needed to be found in the presence of large numbers of local optima. The algorithm is basically hill-climbing except instead of picking the best move, it picks a random move. If the move improves the solution, it is always considered. However, if the move makes the solution worse, it is still considered, but the probability of accepting this wrong move decreases as temperature reduces. Simulated Annealing for floor planning is implemented in **floorplanning.py**.

Details about **floorplanning.py**:

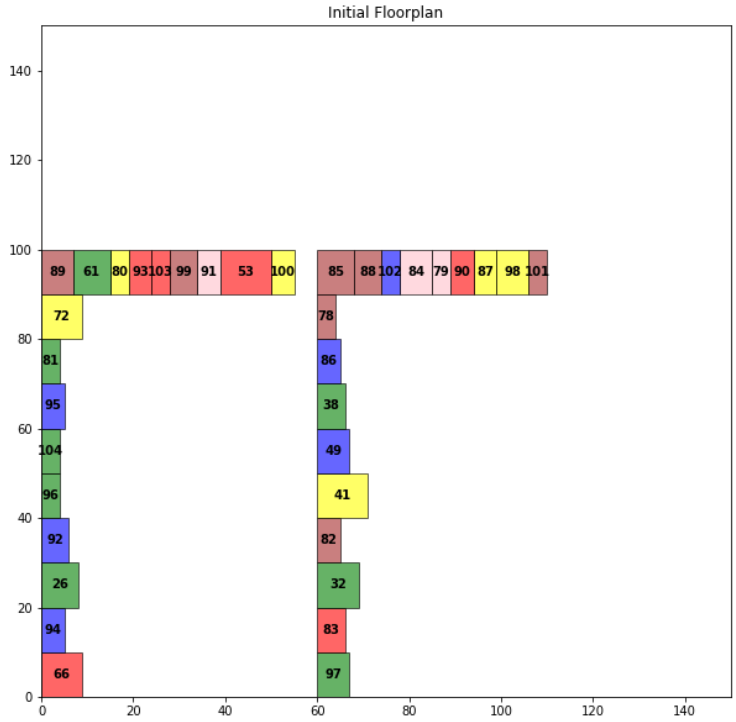
- Inputs:
  - The partitions obtained in the partitioning step. This information is stored in **partition1.txt** and **partition2.txt**.
  - The sizes of each gate. This was approximated during the parsing step and is stored in **widths.txt** and **heights.txt**.
- Outputs:
  - Initial Cost and Initial Polish Expression.
  - Final Cost and final Polish Expression.
  - Plot of area vs iteration.

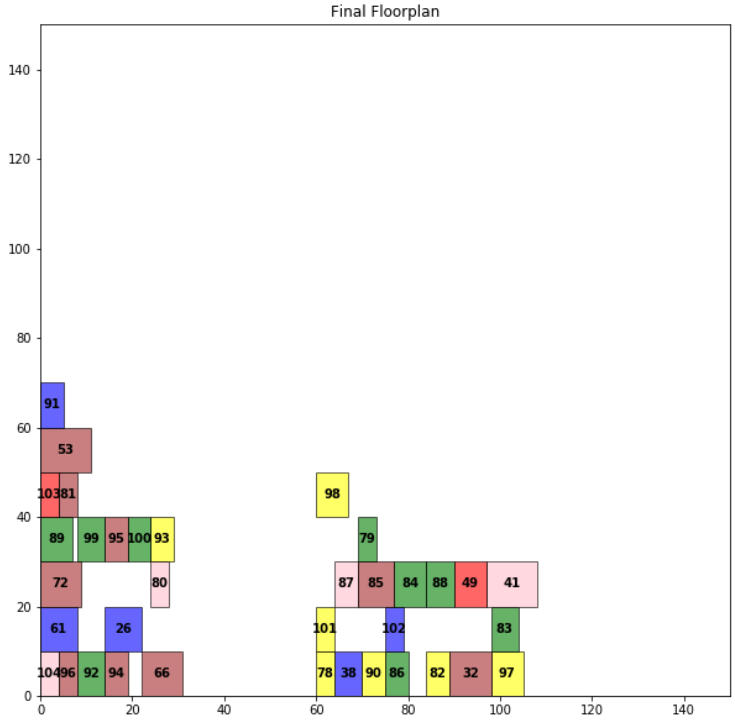
The following functions briefly explain the implementation:

- generate_polish_expression (): A valid Polish Expression is obtained when this function is called.
- get_cost_FP (): An estimate of the total area corresponding to a valid Polish Expression is obtained here.
- moves (): Defines the moves that can be made during the annealing procedure.
  - move1 (): Swaps two adjacent operands.
  - move2 (): Complement a series of operators in a sublist.
  - move3 (): Swap an adjacent operand-operator pair.
- simulated_annealing_FP (): Implements the Simulated Annealing heuristic. Initially, the temperature is set to a high value (1000). At every temperature, a random choice is made among the three moves and is implemented. This random choice is made 500 times in each temperature. After the 500^{th} move, the temperature is updated to 95% of the current temperature. This process is repeated till the temperature falls to 0.01. The costs (approx. area of the floorplan) is plotted against iterations. The Polish Expression corresponding to the lowest cost is considered as the best Floor Plan.
- get_coordinates_FP (): Get the bottom left coordinates of each block in the floor plan of the optimal Polish Expression.
- get_patches (): Used to generate rectangular patches for visualization.

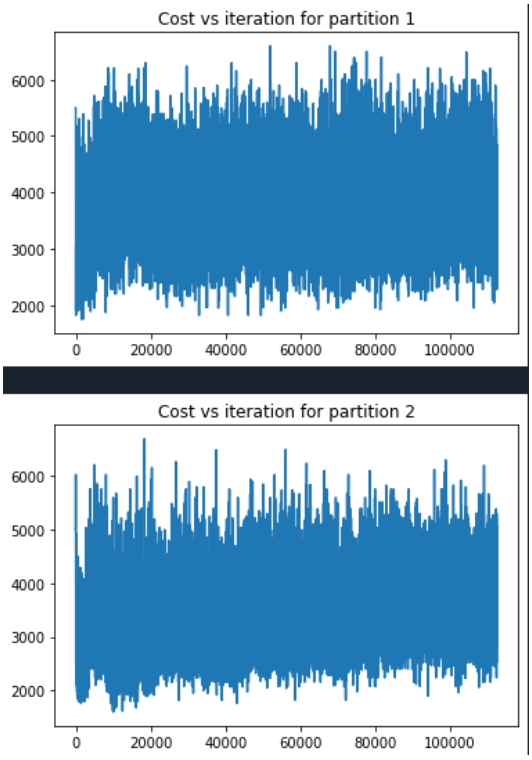
Obtained graphs and outputs of **floorplanning.py**:

```
In [12]: runfile('D:/fifth-sem/vlsi-cad/assignment/floorplanning.py', wdir='D:/
fifth-sem/vlsi-cad/assignment')
Reloaded modules: simulated_annealing_FP, moves, tests, get_cost_FP,
generate_polish_exp, get_coordinates_FP, get_patches_FP
Initial Polish Expression for partition 1:
 ['89', '61', 'V', '80', 'V', '93', 'V', '103', 'V', '99', 'V', '91', 'V', '53',
'V', '100', 'V', '72', 'H', '81', 'H', '95', 'H', '104', 'H', '96', 'H', '92',
'H', '26', 'H', '94', 'H', '66', 'H']

Initial Cost for partition 1: 5500

Final Polish Expression for partition 1:
 ['91', '53', 'H', '103', '81', 'V', '89', 'H', '99', 'V', '95', '100', 'V', 'V',
'72', 'H', '93', '80', 'H', 'V', 'H', '61', '104', '96', 'V', 'H', '92', 'V',
'26', '94', 'H', 'V', '66', 'V', 'H']

Final Cost for partition 1: 1750




Initial Polish Expression for partition 2:
 ['85', '88', 'V', '102', 'V', '84', 'V', '79', 'V', '90', 'V', '87', 'V', '98',
'V', '101', 'V', '78', 'H', '86', 'H', '38', 'H', '49', 'H', '41', 'H', '82', 'H',
'32', 'H', '83', 'H', '97', 'H']

Initial Cost for partition 2: 5000

Final Polish Expression for partition 2:
 ['98', '101', '78', 'H', '87', '79', '85', 'H', 'V', '84', 'V', '38', '90', 'V',
'102', '86', 'H', 'V', 'H', 'V', '88', '49', 'V', '41', 'V', '82', '32', 'V',
'83', '97', 'H', 'V', 'H', 'V', 'H']

Final Cost for partition 2: 1600
```





Note that there are two separate sets of blocks. The left set corresponds to partition 1 and the right set corresponds to partition 2. These partitions were obtained in the partitioning step. The space between the two partitions (x=25 to x=60) is deliberately introduced to help in visualization of the two partitions. In the real world, there will be no such wide gaps. The cost vs iteration plots for the two partitions are given below.



Placement and Routing

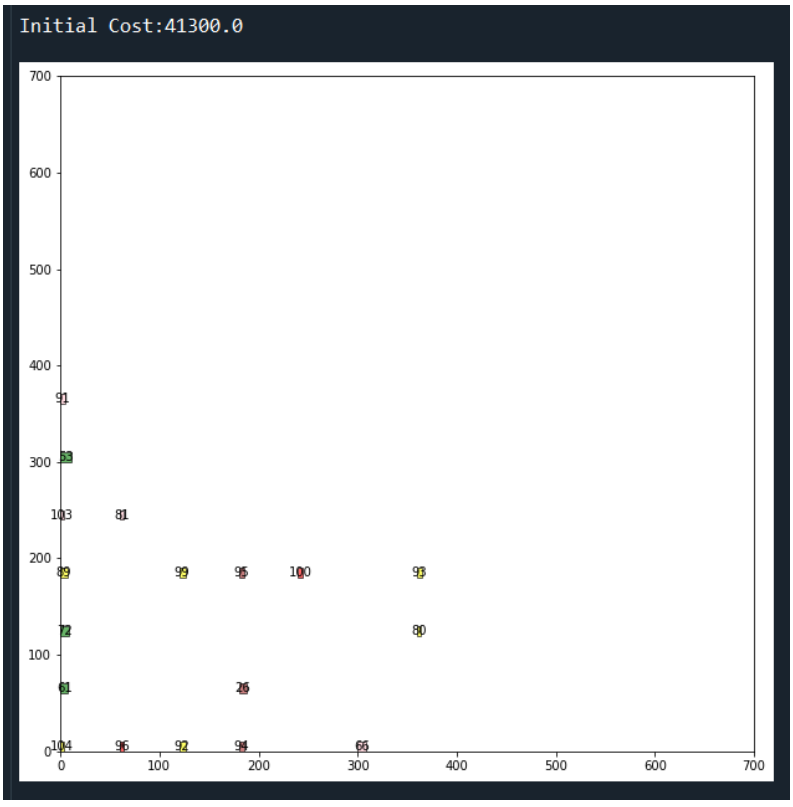
Since the placement and the routing steps are very closely related, they are implemented together. Also, the implementation of placement and routing are done partition-wise, i.e. the partitions are considered separately. They are implemented in **place_n_route.py**.

Simulated Annealing is used for the placement step. This step takes the bottom-left coordinates of each block that was generated in the floor planning step and makes improvements to get a good placement.

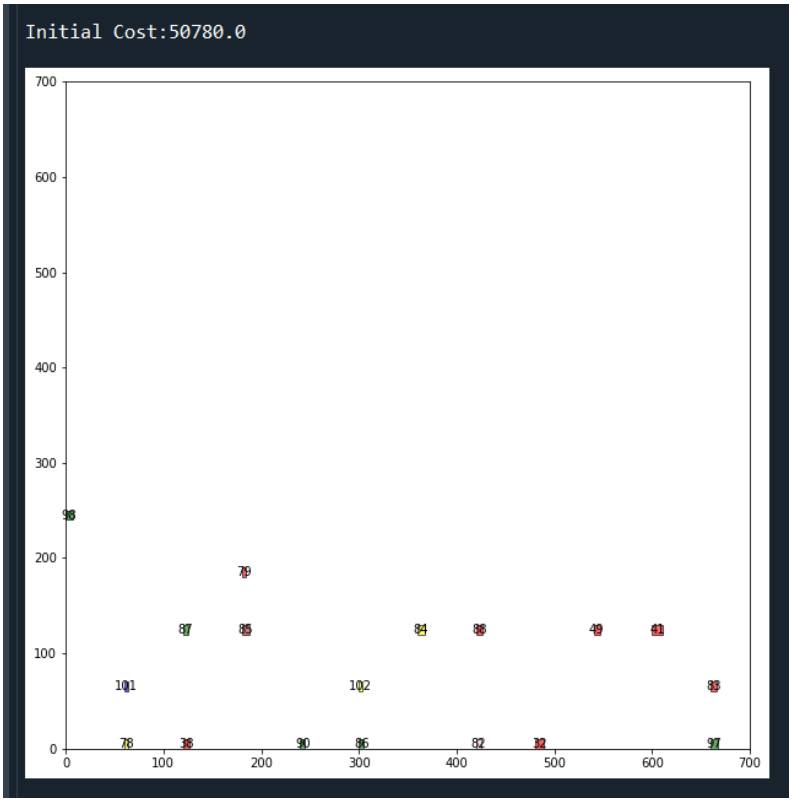
Details about the placement step:

- Inputs:
  - Bottom left coordinates of each block generated in the floor planning stage. They are stored in **floorplan1.txt** and **floorplan2.txt**.
  - The sizes of each gate stored in **widths.txt** and **heights.txt**.
  - Connections between the cells. This info is stored in **graph_edges.txt**.
- Outputs:
  - Initial Cost and Final Cost of placement
  - Final coordinates stored in **placement_final_p1.txt** and **placement_final_p2.txt**.

Initially, the blocks are placed in a checker-board model, with each block of size 10x10. The initial placement and initial cost of placement is given below:



For partition 1. Cost = 41300



For partition 2. Cost = 50780

The following functions briefly explain the implementation:

- get_initial_placement (): Get an initial placement from the floorplan. This is done by placing each block in a checker board model as mentioned before.
- get_boundaries (): Used to get the maximum x and y coordinates in the placement.
- get_cost_placement (): Cost function for the placement procedure is dependent on both the wirelength and the area occupied. The equation is given by Cost = a1*Wirelength + b1*Area. This function makes use of the following-
    - getHPWL (): Used to get the Half Perimeter Wire Length. This can be an estimate for the wire length between any two blocks.
    - getWireLength (): Get the total wirelength of a particular placement.
- moves_placement (): Defines the moves that can be made during the annealing procedure.
    - move1 (): Randomly pick a block and move to a vacant position.
    - move2 (): Pick two blocks randomly and swap their positions.
- simulated_annealing_placement (): Implements the Simulated Annealing heuristic for placement. This is similar to the Simulated Annealing algorithm explained in the floor planning stage. A random move is made at each temperature. If the new move leads to a lower cost, it is always accepted. If the new move leads to a higher cost, it is accepted with a probability that decreases as temperature reduces. The probability function is given by:

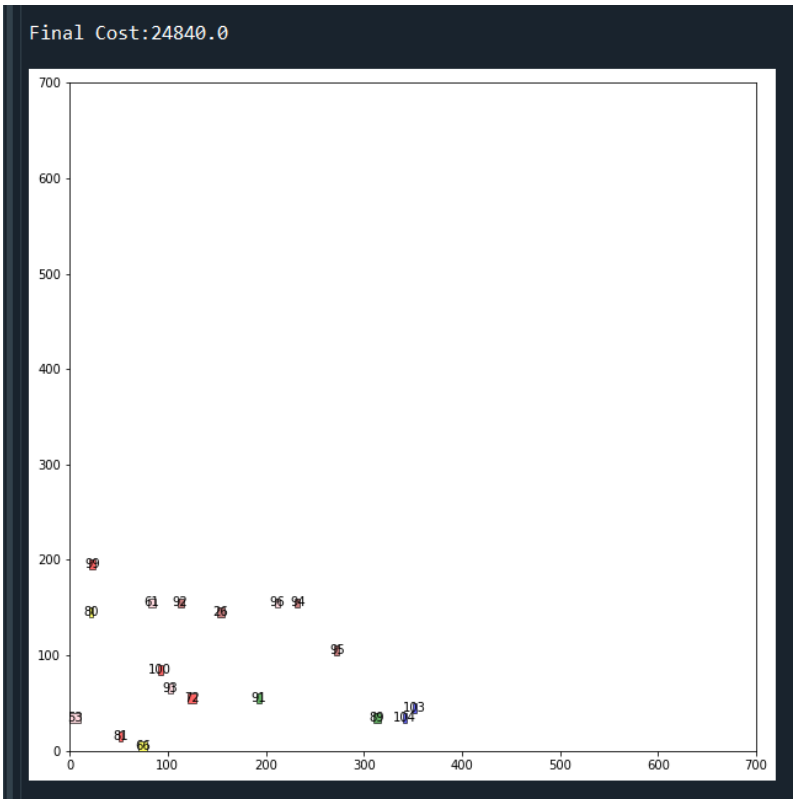
$$P \text{ (accepting wrong move)} = e^{-\delta/T}$$

Where δ = difference between new cost and old cost.

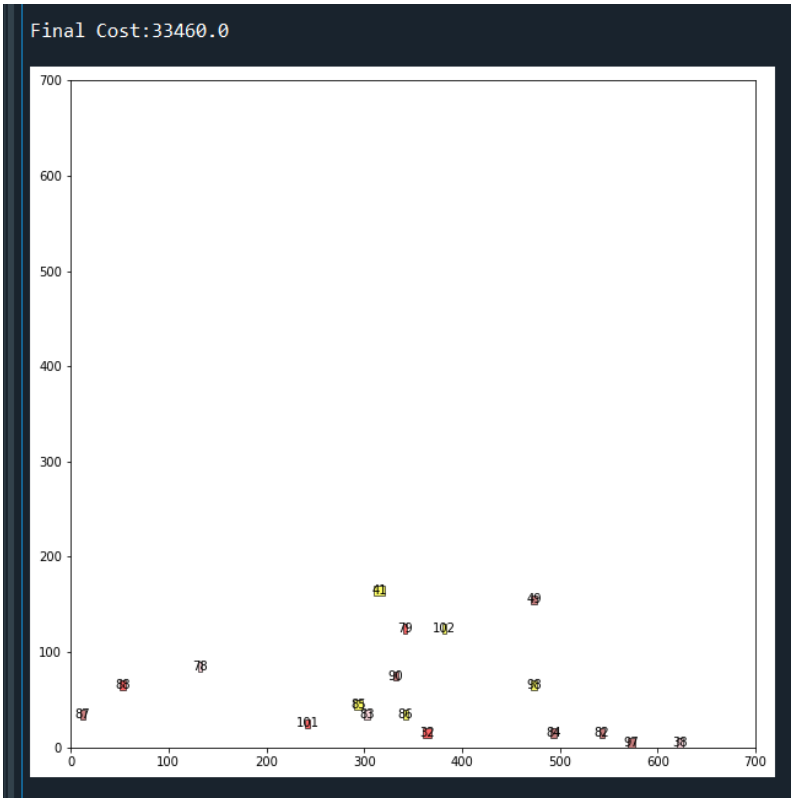
T = current temperature.

We can clearly see that as T reduces, δ/T increases, $e^{-\delta/T}$ reduces, which means P reduces.

The final placement of partition 1 and partition 2 and their associated costs are given below:



For partition 1. Cost = 24840



For partition 2. Cost = 33460

Maze Router Lee algorithm is used for routing. It is based on Breadth-First-Search. It always finds a path between any two given points, if it exists. However, it is quite slow and requires considerable memory. The multi-layer extension of the Lee algorithm is implemented here.

Details about the routing step:

- Inputs:
  - Coordinates of each cell generated in the placement step. These are stored in **placement_final_p1.txt** and **placement_final_p2.txt**.
  - Connections between blocks (gates). This is stored in **graph_edges.txt**.
- Outputs:
  - Wire paths and cost of wiring.
  - In case the routing is unsuccessful, it notifies that the placement step should be repeated again. This is the reason placement and routing are implemented together.

The following functions briefly explain the implementation:

- get_boundaries_routing (): Used to get the maximum x and y coordinates in the grid.
- check_cell (): Checks if a given cell in the grid is occupied or vacant.
- get_cost (): Returns cost of the current routing.
- find_path (): Finds a path between two points if it exists. It has three phases- Wave propagation, Backtrace and Clearance.
  - Wave propagation: In this phase, the cells are marked from the source to the destination.
  - Backtrace: In this phase, the path is retraced from the destination back to the starting point. The visited cells during backtrace are marked as well.
  - Clearance: The cells marked during the backtrace phase are marked ‘occupied’ and are not considered in the future.

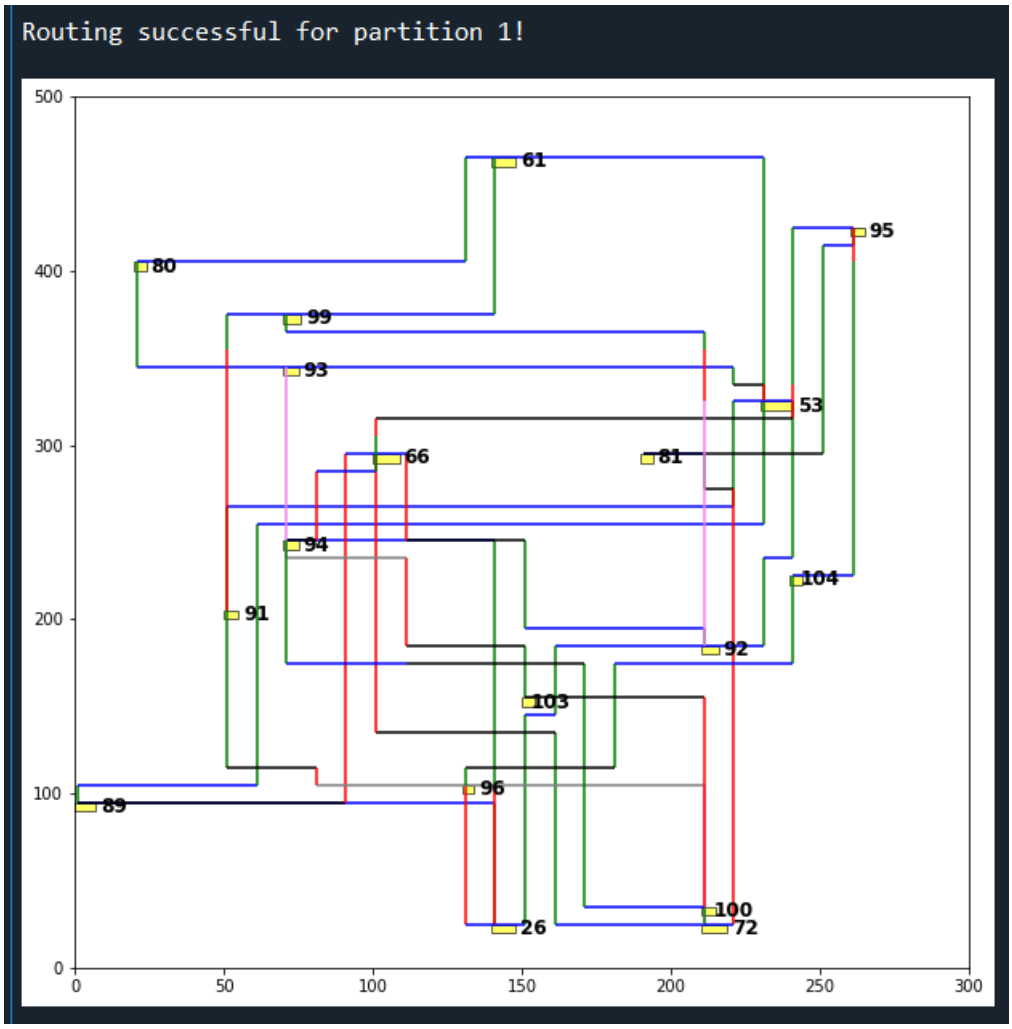
Every two cells that have a connection between them are named as source and destination. The vacant cells adjacent to the source (up, down, left, right, top, bottom) are marked 1. The vacant cells adjacent to these cells are marked 2. This goes on until the destination is reached. Once the destination is reached, a move is made from the destination to the cell marked with the next lower number. In this way, a path is retraced back to the source and this will be the path used for routing. Finally, the blocks over which the current wire has passed are marked ‘occupied’ and are not considered in the future.

Routing may not be successful in cases where the placement doesn’t offer enough place for the wires to be routed. These are examples of unsuccessful routing.

```
Routing Failed! Go back to placement again.
24 out of 28 routed successfully.
Routing Cost for partition 1: 3640
```

```
Routing Failed! Go back to placement again.
22 out of 29 routed successfully.
Routing Cost for partition 2: 4650
```

This is an example of a successful routing:



For partition 1.

The colors are used for the respective layers.

1. Blue
2. Green
3. Black
4. Red
5. Grey
6. Violet