Falk Sinke
Claartje Barkhof May 31st 2017
Lucas Fijen Heuristieken, Universiteit van Amsterdam

Solving a constraint optimisation problem found in integrated circuits with a Heatmap guided A* implementation

Abstract The goal of the Chips and Circuits case is to find an optimal configuration of wires, called *nets*, between gates on a logic board in a chip. This case deals with a constraint optimisation problem: a kind of problem that consists of both finding a valid solution (the constraint) and using the least amount of wire possible (optimisation). We approached this problem with a purpose-built Heatmap guided A* Search Algorithm, which we showed to be highly effective in finding a solution to this problem. The Heatmap forces nets to be laid down further from gates to prevent routes for other nets to be very limited or even closed off. This heatmap is distinct for every netlist. We managed to solve all the netlists given. After meeting the constraints we were able to optimise our results by reducing the total wire length by at least 12,2% in each valid solution.

Introduction

In this report we will propose a solution to the case "Chips and Circuits" from the subject Heuristics at the UvA. The goal of this case is to find an optimal configuration of wires, called *nets*, between gates on a logic board in a chip. Chips or integrated circuits are found in most electronic devices and the chips are crucial to the devices' performance. The case consists of six *netlists* of varying length: prescriptions of how the gates should be connected. There are two circuit boards structured as a Manhattan style grids and for each board there are three netlists to be solved.

The nets should follow the grid structure. The gates are situated on the bottom layer, layer 0. It is allowed to add up to 7 more layers in height. The grid this way becomes 3 dimensional and the nets can travel through this grid. To prevent short circuit, the nets can not intersect. Also they cannot run through gates. As longer nets mean more used material per chip - causing higher costs and slower chips - the nets should be kept as short as possible.

This kind of problem is called a *constraint optimisation problem* (COP). A COP can be divided in two subproblems: the constraint satisfaction and the optimisation. In this case the constraint satisfaction is finding a valid solution. A valid solution is a solution that meets all constraints: all gates are connected according to the netlist, the nets connecting them do not intersect or run through gates and are all within the bounds of the grid. The optimisation entails finding the optimal solution: the solution with the shortest cumulative wire length.

We used a combination of algorithms to solve and optimise this case. We developed a program using an *A* Search Algorithm* (A*) implemented with a purpose-built *Heatmap*. This combination appeared to be highly effective for meeting the constraints of this case. Optimisation was approached using only A*.

Materials

We used the circuit boards and netlists provided by the Heuristieken website (Berg, D. van den. 2015). Netlist 1, 2 and 3 are to be laid down on circuit board A and netlist 4, 5 and 6 on circuit board B.

Our program that tries to find the best solution to the case was written in Python 3.6.0 in Pycharm and Atom as main APIs. GitHub was used for the exchange of code and backup purposes. The following Python libraries were used: *The Python Standard Library, MatplotLib 2.0.2 and Plot.ly 2.0.8.*

Three different systems were used for writing and running the program: A MacBook Pro running macOS Sierra 10.12.1 with a 2,4 GHz Intel Core i5 processor and 4 GB DDR3 RAM, a MacBook Pro running OS X El Capitan 10.11.5 with a 2GHz Intel Core i7 processor and 8 GB DDR3 RAM and a HP EliteBook 8570W running Ubuntu 16.04LTS with a 2,4 GHz intel i7 and 8GB DDR3 RAM.

Methods

Our program is based on two main algorithms: A* and a Heatmap implementation. First we will discuss the two individually and secondly the integration of both to meet the constraints of the problem solution. Lastly, we will describe our optimisation method.

The A* search algorithm.

The A* algorithm is a search algorithm used for pathfinding and graph traversal. The algorithm finds the shortest path between two points without preprocessing the graph to work properly. In our case these points are gates on the circuit board grid. The algorithm uses an estimation of distance yet to travel (the *heuristic*) and the distance already traveled to efficiently explore the *state space*. The state space consists of all shortest paths from the first gate -a grid point or a node marking the beginning of a path - to all explored nodes in the grid. In case the end-node, second gate, is visited, exploration is completed and the shortest path from the first to the final gate can be returned. It is more efficient than a normal breadth-first search, since it explores the promising states first. In the worst case it still has to explore all states, just like breadth-first. Exploring this state space while looking for the desired path efficiently is expedient, because the case deals with an extensive state space growing disproportionately rapid when expanding the grid.

We implemented A* using a *Priority Queue* data structure. A Priority Queue (PQ) is a queue that automatically sorts its elements after each insertion using a score or penalty. The item with the highest priority is at the front of the list and will be popped first. For the use of A* the highest priority is granted to the element with the lowest penalty. For example: If the highest priority element A in the queue has a penalty of 8 and an element B with a penalty of 6 is inserted, B is granted higher priority and moves to the front of the queue, leaving A as the second element. In our case, the elements are the paths that are explored until that point.

A* implemented with a PQ works as follows. First the start node (in our case: first gate) is inserted in an empty PQ as a path with a length of one node. Since this is now the only path in the queue it will be popped. The path with an appurtenant penalty will then be *expanded*: all the nodes that are valid follow up states will be evaluated. In our case valid follow up states are all nodes that are one step in a direction on the x, y or z axis and exist in the grid. Also the nodes can not be occupied by a wire or a non-terminal gate. The valid follow up nodes will be added to copies of the currently expanding path and inserted as new paths in the PQ.

In the evaluation phase A* uses two guidelines to grant a penalty to a path: (1) the traveled path length and (2) a heuristic. The heuristic function calculates an estimation of the distance that still has to be traversed to reach the goal node, the final gate. This estimation has to be equal to or less than the actual distance (*admissible*). Otherwise items might end up further down the queue than they should be, which results in a non-optimal solution. For our purposes we used the Manhattan distance (dx + dy + dz) as heuristic function. The final penalty is calculated by taking the sum of the two guidelines. The nodes that are already visited are saved. If you encounter the same node twice, A* guarantees there is a shorter path already found. A* proceeds with popping and expanding the first path until the final gate is found. That path is guaranteed to be the shortest path.

The Heatmap implementation

Although A* finds the shortest paths possible, it does not take the order of laying the nets down into account. This can have two undesirable effects. First, it can cause situations in which gates are closed off causing that some nets can not be laid down and no solution is found. Secondly, it can cause some nets to be laid down in a cumbersome manner, because options for routes become disproportionately limited. Thus we need A* to find paths that do not run right next to or close to gates. To achieve this, we invented a Heatmap. The Heatmap implementation assigns a heat penalty to each gridpoint in the circuit board. The heat penalty is added to the heuristic function so paths that contain a higher cumulative penalty will end further down the priority queue.

The heat penalty is calculated in the following manner. First we assign an intensity constant of the heatmap, which we call the *heat constant*. Then for each gate on the logic board the heat penalty is determined using this formula:

Heat penalty =
Heat constant - (2 · Manhattan distance to gate)

The final heat penalty of a node (grid point) is the sum of all heat penalties given to that node from all gates. As said, we add the penalty of a node to the total score A* grants a node in the evaluation phase.

The new calculation of the penalty for A*, with a Heatmap implementation, is:

A penalty = Total heat penalty of traveled path +*
pathlength of traveled path +
Manhattan distance to final gate +
heat penalty of currently evaluated node

This results in A* avoiding paths that lie right next or close to gates to prevent it from blocking gates off and making new connections impossible. Though, if a path running next to the gate is the only option, A* will still find that path.

It is important to note that this updated calculation of the penalty for A* results in a path that is not per se the shortest, but has the lowest penalty. A traditional A* counts length as cost, but we also take the heat into account. Think of it like a mountainous landscape: the fastest, not per se the shortest, routes run through valleys.

To determine the right heat constant we did a pilot study where we tried heats ranging from 0 to 50. Our observation is that the results vary using heats constants ranging from 0 to ~30, but drop after ~30 (Figure 1). We surmise this is because the shape of the heatmaps doesn't seem to change by increasing the heat constant at a certain point. All heatmaps take a cone shape with a spherical top at great heat constants (Figure 2). Hence we use 30 as our maximum heat.

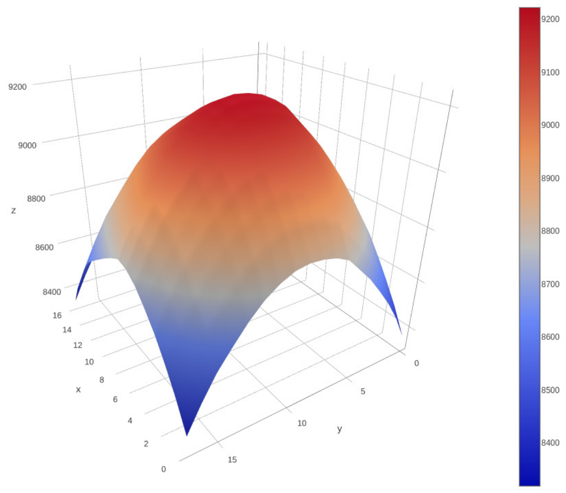


Figure 2: layer 0 of circuit board B with a heat constant of 2000

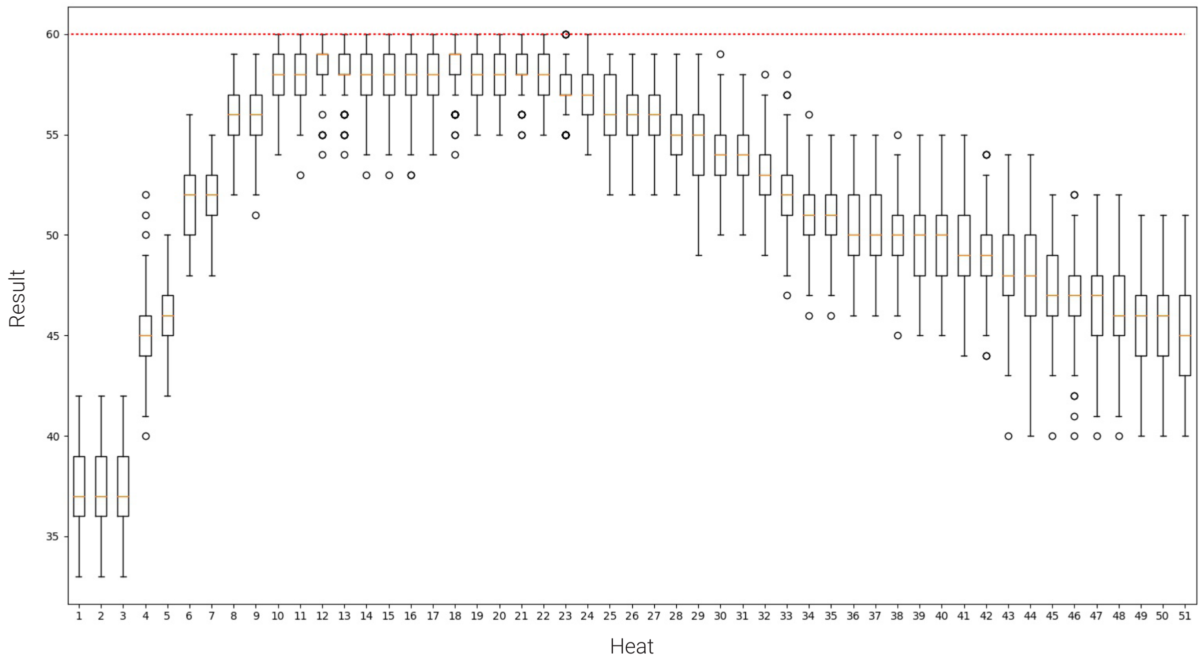


Figure 1: distribution of the results per heat (100 iterations netlist 5)

Constraint satisfaction

We satisfied the constraints using the Heatmap guided A* algorithm. Our program first produces a Heatmap for a given heat constant. It then tries to satisfy a given netlist with the Heatmap guided A*. If A* doesn't succeed to find a valid path for each net in the netlist, the whole process is restarted but with a heat incremented by 1. This means the Heatmap will change and A* will choose different routes. If the program does not find a solution after a set amount of heat constants, it will restart with a different order (*permutation*) of the nets in the netlist. Because the paths from gate to gate will now be searched for in different order, some nodes that would be previously closed off will now be open and vice versa. Remember that even with a Heatmap implementation, gates can still be closed off, if that's the only option left. This process is repeated until a valid solution for all netlist has been found.

Optimisation

Because our method using the heatmap made A* avoid paths that are right next to gates, a lot of unnecessary twists and detours are left over. These undesirable elongations of the paths result in an unnecessarily long total wire length. To remove this extra wire length , we removed and reinserted all paths one by one with regular A*. This means the heat constant is set to 0. This causes A* to find the shortest paths possible without obstructing other paths or making unnecessary detours.

After doing the optimisation for each solution that met the constraints, the solution with the shortest total wire length is picked as the final result.

Results

We managed to meet the constraints of 6 out of 6 netlists. An overview of these results produced by A* with a Heatmap implementation is shown below in Table 1. The outcomes shown in Table 1 are all computed similarly. Our algorithm was run for all netlists with 500 random permutations per netlist with a heat varying from 5 to 30 for each permutation. The algorithm, thus, is run 12500 times per netlist in total.

To compare these results with a regular A* algorithm, we also ran the regular A*, i.e. a heat constant of 0, for all netlists with 500 permutations per netlist. These results are shown in Table 2. Table 2 shows no netlists were solved using this strategy. Optimisation is not addressed in this table, since only valid solutions can be optimised and none are valid.

Table 1 does show optimisation results. We

Table 1: an overview of the results for all netlists produced by the Heatmap guided A* implementation, 500 permutations, heat constant 5-30 per permutation

	Circuit board 1			Circuit board 2		
Netlist	1	2	3	4	5	6
Number of nets laid down	30/30	40/40	50/50	50/50	60/60	70/70
Heat constant	5	19	14	29	18	22
Total length before optimisation	409	765	1003	1096	1416	1615
Total length after optimisation	359	455	743	818	910	1341
Decrease in total length after optimisation	12,2%	40,5%	25,9%	25,4%	35,7%	17,0%
Lower bound total length	291	341	475	600	578	761
Percentage of optimised total length exceeding the lower bound	23,4%	33,4%	56,4%	36,3%	57,4%	76,2%

Table 2: an overview of the results for all netlists produced by regular A*, 500 permutations, heat constant 0 for all permutations

	Circuit board 1			Circuit board 2		
Netlist	1	2	3	4	5	6
Number of nets laid down	27/30	32/40	37/50	40/50	45/60	47/70

showed the total wire length found after running the algorithm before optimisation, the optimised total wire length and the decrease of wire length in percentages caused by optimisation. To put the optimised total length found in perspective, a lower bound is added to Table 1. The lower bound is computed as the sum of all nets put down together at the bottom layer without any restrictions: the Manhattan distance of all nets added up together. Also reported is the percentage of total length that exceeds the lower bound. This is computed by the following equation:

$$\left(\frac{\textit{Optimised total length}}{\textit{Lower bound total length}} \cdot 100\%\right) - 100\%$$

The results in Table 1 show us that the optimisation decreased the length by at least 12,2% and at most 40,5%. The (non-realistic) lower bound is exceeded by at least 23,4% and at most 76,2%.

The permutations that yielded the best results with Heatmap implementation are shown in Appendix A and without Heatmap in Appendix B. See Appendix C for a visualisation for our best found results of each netlist.

Conclusion

In this report we tried to resolve the constraint optimisation problem of the total length of nets in chips. We used the A* search Algorithm, the Heatmap Algorithm and a randomiser of the netlist for solving the constraint satisfaction. As an optimisation approach we reused the A* search Algorithm without the Heatmap.

Using only the A* search Algorithm we were not able to satisfy the constraint of any netlist. After combining the Heatmap Algorithm with the A* search Algorithm and by randomising the order of the nets in the netlists we were able to satisfy every constraint of the six netlists. From these results we can conclude that all the six netlists are solvable. Also the A* search Algorithm in combination with the Heatmap Algorithm is proven to be a more effective tool to meet the constraints of this case, a classical example of a minimum cost path finding problem.

From the results we can also conclude that there is not an universal optimal heat value. For each netlist there is a different heat value at which we found our best result. The bigger netlists had a higher heat value at which we found the best result. We believe this is because a higher heat value forces the tracks to make more use of the altitudinal direction. This way the nets are more spread out, leaving more space for others to be laid down. Also we found that the order in which the nets are performed is crucial in both satisfying the constraint and optimising the total length.

As for the optimisation, we were able to reduce the total length of the paths in the netlists by at least 12,2%, compared to the lower bound, by running only the A* search Algorithm again on the netlists which were already solved.

Discussion

Although we showed that all netlists are solvable, the optimisation is debatable. We do not know how close the theoretical optimal solution in total wire length is, to the lower bound. Thus, we cannot judge how optimal our findings are. The only thing we do know for certain is that the optimal length must be higher than the lower bound, since the lower bound is computed without meeting the constraint: there are paths that intersect. However, our optimisation had effect: it notably shortened the total length.

Since it would very well be possible that there exist more optimal solutions, one might want to find this using a different approach of traversing through all possible permutations. The amount of possible permutations is !n, with n being the number of nets in the netlist. This number is too large to fully explore in our case and grows very fast by increasing the length of the netlist. Because we took random permutations, we cannot guarantee them to be unique, neither explore promising sets of permutations that are alike. Further research should be conducted into finding better ways to do this. We suggest trying algorithms like Simulated Annealing, which take smaller changes into account while looking for an optimal solution, rather than randomly changing the entire order of the netlist.

Also, we did not try to optimise the number of layers of the chip. There could exist a solution to the problem with more wire but less layers. This however is not certain to be more optimal due to lack of knowledge about the relative costs of wire in comparison to layers.

Appendix A

Final permutations netlists matching the results in Table 1:

Netlist 1:
[(‘23’, ‘12’), (‘4’, ‘16’), (‘4’, ‘24’), (‘6’, ‘8’), (‘23’, ‘17’), (‘10’, ‘14’), (‘21’, ‘11’), (‘12’, ‘25’), (‘7’, ‘15’), (‘4’, ‘1’), (‘4’, ‘6’), (‘4’, ‘5’), (‘16’, ‘6’), (‘16’, ‘18’), (‘14’, ‘19’), (‘23’, ‘14’), (‘20’, ‘3’), (‘3’, ‘21’), (‘24’, ‘5’), (‘2’, ‘1’), (‘8’, ‘10’), (‘16’, ‘22’), (‘21’, ‘20’), (‘8’, ‘14’), (‘11’, ‘5’), (‘16’, ‘9’), (‘24’, ‘9’), (‘17’, ‘10’), (‘11’, ‘8’), (‘20’, ‘6’)]

Netlist 2:
[(‘20’, ‘10’), (‘9’, ‘19’), (‘15’, ‘7’), (‘9’, ‘24’), (‘5’, ‘4’), (‘14’, ‘12’), (‘1’, ‘16’), (‘4’, ‘1’), (‘20’, ‘9’), (‘19’, ‘14’), (‘3’, ‘6’), (‘11’, ‘21’), (‘18’, ‘12’), (‘17’, ‘22’), (‘6’, ‘18’), (‘19’, ‘22’), (‘13’, ‘21’), (‘15’, ‘8’), (‘16’, ‘14’), (‘16’, ‘5’), (‘23’, ‘11’), (‘12’, ‘8’), (‘23’, ‘19’), (‘11’, ‘5’), (‘15’, ‘6’), (‘5’, ‘1’), (‘10’, ‘24’), (‘12’, ‘16’), (‘25’, ‘13’), (‘12’, ‘13’), (‘2’, ‘22’), (‘23’, ‘9’), (‘7’, ‘10’), (‘2’, ‘23’), (‘11’, ‘6’), (‘13’, ‘14’), (‘15’, ‘2’), (‘14’, ‘20’), (‘24’, ‘21’), (‘16’, ‘11’)]

Netlist 3
[(‘19’, ‘3’), (‘18’, ‘16’), (‘25’, ‘17’), (‘3’, ‘4’), (‘5’, ‘9’), (‘7’, ‘5’), (‘9’, ‘2’), (‘12’, ‘18’), (‘14’, ‘3’), (‘13’, ‘10’), (‘11’, ‘4’), (‘12’, ‘1’), (‘21’, ‘17’), (‘4’, ‘10’), (‘1’, ‘15’), (‘13’, ‘15’), (‘19’, ‘21’), (‘23’, ‘19’), (‘4’, ‘20’), (‘11’, ‘1’), (‘14’, ‘20’), (‘14’, ‘5’), (‘16’, ‘22’), (‘12’, ‘4’), (‘3’, ‘17’), (‘8’, ‘24’), (‘9’, ‘10’), (‘1’, ‘14’), (‘6’, ‘15’), (‘2’, ‘14’), (‘8’, ‘6’), (‘20’, ‘22’), (‘9’, ‘11’), (‘1’, ‘23’), (‘13’, ‘25’), (‘19’, ‘11’), (‘2’, ‘16’), (‘5’, ‘2’), (‘17’, ‘8’), (‘23’, ‘12’), (‘3’, ‘7’), (‘25’, ‘24’), (‘6’, ‘13’), (‘5’, ‘10’), (‘9’, ‘8’), (‘11’, ‘2’), (‘18’, ‘22’), (‘21’, ‘7’), (‘25’, ‘16’), (‘18’, ‘10’)]

Netlist 4
[(‘13’, ‘2’), (‘30’, ‘10’), (‘9’, ‘43’), (‘23’, ‘50’), (‘44’, ‘31’), (‘44’, ‘16’), (‘15’, ‘5’), (‘50’, ‘14’), (‘35’, ‘46’), (‘40’, ‘16’), (‘1’, ‘13’), (‘34’, ‘31’), (‘27’, ‘25’), (‘5’, ‘31’), (‘11’, ‘33’), (‘33’, ‘30’), (‘1’, ‘22’), (‘22’, ‘8’), (‘37’, ‘3’), (‘47’, ‘13’), (‘29’, ‘44’), (‘49’, ‘25’), (‘15’, ‘40’), (‘32’, ‘26’), (‘36’, ‘20’), (‘37’, ‘5’), (‘6’, ‘32’), (‘4’, ‘49’), (‘18’, ‘26’), (‘5’, ‘30’), (‘42’, ‘2’), (‘4’, ‘32’), (‘40’, ‘19’), (‘32’, ‘12’), (‘3’, ‘4’), (‘22’, ‘7’), (‘13’, ‘15’), (‘21’, ‘41’), (‘38’, ‘17’), (‘43’, ‘4’), (‘15’, ‘7’), (‘9’, ‘29’), (‘36’, ‘27’), (‘36’, ‘15’), (‘47’, ‘1’), (‘29’, ‘28’), (‘5’, ‘24’), (‘48’, ‘23’), (‘6’, ‘45’), (‘47’, ‘36’)]

Netlist 5
[(‘9’, ‘28’), (‘35’, ‘6’), (‘42’, ‘14’), (‘23’, ‘31’), (‘31’, ‘10’), (‘32’, ‘42’), (‘33’, ‘12’), (‘48’, ‘6’), (‘28’, ‘23’), (‘5’, ‘9’), (‘5’, ‘41’), (‘49’, ‘48’), (‘14’, ‘50’), (‘45’, ‘35’), (‘1’, ‘17’), (‘16’, ‘43’), (‘25’, ‘24’), (‘2’, ‘4’), (‘20’, ‘34’), (‘45’, ‘43’), (‘34’, ‘5’), (‘5’, ‘16’), (‘39’, ‘17’), (‘41’, ‘11’), (‘24’, ‘44’), (‘21’, ‘48’), (‘46’, ‘19’), (‘27’, ‘2’), (‘19’, ‘28’), (‘4’, ‘34’), (‘18’, ‘4’), (‘44’, ‘45’), (‘29’, ‘41’), (‘34’, ‘37’), (‘13’, ‘26’), (‘17’, ‘2’), (‘10’, ‘6’), (‘45’, ‘30’), (‘13’, ‘25’), (‘1’, ‘22’), (‘40’, ‘49’), (‘20’, ‘31’), (‘26’, ‘50’), (‘27’, ‘9’), (‘26’, ‘25’), (‘36’, ‘7’), (‘3’, ‘18’), (‘1’, ‘13’), (‘21’, ‘30’), (‘13’, ‘33’), (‘25’, ‘9’), (‘32’, ‘5’), (‘8’, ‘29’), (‘29’, ‘46’), (‘37’, ‘30’), (‘11’, ‘20’), (‘19’, ‘12’), (‘35’, ‘22’), (‘1’, ‘11’), (‘37’, ‘38’)]

Netlist 6:
[(‘28’, ‘39’), (‘26’, ‘18’), (‘38’, ‘41’), (‘33’, ‘3’), (‘22’, ‘43’), (‘27’, ‘42’), (‘30’, ‘38’), (‘2’, ‘12’), (‘41’, ‘48’), (‘49’, ‘3’), (‘47’, ‘30’), (‘31’, ‘33’), (‘4’, ‘22’), (‘13’, ‘6’), (‘25’, ‘30’), (‘28’, ‘48’), (‘2’, ‘14’), (‘39’, ‘8’), (‘2’, ‘21’), (‘34’, ‘8’), (‘44’, ‘32’), (‘4’, ‘15’), (‘35’, ‘3’), (‘49’, ‘35’), (‘39’, ‘50’), (‘40’, ‘4’), (‘28’, ‘37’), (‘6’, ‘40’), (‘26’, ‘7’), (‘45’, ‘29’), (‘40’, ‘39’), (‘6’, ‘42’), (‘13’, ‘10’), (‘2’, ‘38’), (‘9’, ‘5’), (‘12’, ‘46’), (‘13’, ‘37’), (‘22’, ‘34’), (‘1’, ‘40’), (‘45’, ‘35’), (‘12’, ‘11’), (‘23’, ‘45’), (‘31’, ‘41’), (‘47’, ‘1’), (‘35’, ‘20’), (‘37’, ‘21’), (‘1’, ‘4’), (‘17’, ‘11’), (‘33’, ‘26’), (‘4’, ‘11’), (‘27’, ‘40’), (‘15’, ‘17’), (‘6’, ‘18’), (‘7’, ‘34’), (‘31’, ‘8’), (‘15’, ‘35’), (‘46’, ‘39’), (‘38’, ‘3’), (‘14’, ‘46’), (‘50’, ‘13’), (‘34’, ‘32’), (‘38’, ‘44’), (‘48’, ‘45’), (‘18’, ‘36’), (‘17’, ‘9’), (‘50’, ‘29’), (‘15’, ‘44’), (‘15’, ‘47’), (‘16’, ‘23’), (‘19’, ‘43’)]

Appendix B

Final permutations netlists matching the results in Table 2:

Netlist 1:
[(‘16’, ‘22’), (‘11’, ‘8’), (‘4’, ‘6’), (‘23’, ‘17’), (‘21’, ‘11’), (‘14’, ‘19’), (‘6’, ‘8’), (‘8’, ‘10’), (‘7’, ‘15’), (‘4’, ‘5’), (‘2’, ‘1’), (‘8’, ‘14’), (‘23’, ‘12’), (‘20’, ‘6’), (‘10’, ‘14’), (‘23’, ‘14’), (‘16’, ‘6’), (‘20’, ‘3’), (‘21’, ‘20’), (‘4’, ‘16’), (‘11’, ‘5’), (‘16’, ‘18’), (‘4’, ‘1’), (‘24’, ‘9’), (‘17’, ‘10’), (‘4’, ‘24’), (‘24’, ‘5’), (‘16’, ‘9’), (‘12’, ‘25’), (‘3’, ‘21’)]

Netlist 2:
[(‘10’, ‘24’), (‘12’, ‘8’), (‘6’, ‘18’), (‘25’, ‘13’), (‘23’, ‘11’), (‘24’, ‘21’), (‘13’, ‘14’), (‘12’, ‘13’), (‘15’, ‘6’), (‘9’, ‘19’), (‘16’, ‘11’), (‘3’, ‘6’), (‘20’, ‘9’), (‘14’, ‘12’), (‘19’, ‘22’), (‘17’, ‘22’), (‘1’, ‘16’), (‘15’, ‘7’), (‘11’, ‘21’), (‘16’, ‘14’), (‘20’, ‘10’), (‘9’, ‘24’), (‘11’, ‘5’), (‘16’, ‘5’), (‘4’, ‘1’), (‘23’, ‘9’), (‘5’, ‘4’), (‘18’, ‘12’), (‘11’, ‘6’), (‘19’, ‘14’), (‘13’, ‘21’), (‘2’, ‘22’), (‘23’, ‘19’), (‘5’, ‘1’), (‘15’, ‘8’), (‘15’, ‘2’), (‘2’, ‘23’), (‘7’, ‘10’), (‘12’, ‘16’), (‘14’, ‘20’)]

Netlist 3:
[(‘5’, ‘9’), (‘17’, ‘8’), (‘23’, ‘12’), (‘11’, ‘1’), (‘19’, ‘3’), (‘9’, ‘2’), (‘4’, ‘20’), (‘9’, ‘8’), (‘8’, ‘6’), (‘5’, ‘2’), (‘3’, ‘17’), (‘14’, ‘3’), (‘6’, ‘13’), (‘8’, ‘24’), (‘11’, ‘4’), (‘25’, ‘24’), (‘13’, ‘25’), (‘20’, ‘22’), (‘12’, ‘1’), (‘5’, ‘10’), (‘6’, ‘15’), (‘3’, ‘4’), (‘4’, ‘10’), (‘13’, ‘15’), (‘25’, ‘17’), (‘1’, ‘14’), (‘18’, ‘22’), (‘18’, ‘10’), (‘9’, ‘10’), (‘12’, ‘18’), (‘13’, ‘10’), (‘7’, ‘5’), (‘11’, ‘2’), (‘25’, ‘16’), (‘14’, ‘5’), (‘1’, ‘23’), (‘1’, ‘15’), (‘14’, ‘20’), (‘2’, ‘14’), (‘3’, ‘7’), (‘16’, ‘22’), (‘9’, ‘11’), (‘19’, ‘11’), (‘21’, ‘17’), (‘2’, ‘16’), (‘23’, ‘19’), (‘18’, ‘16’), (‘21’, ‘7’), (‘12’, ‘4’), (‘19’, ‘21’)]

Netlist 4:
[(‘29’, ‘28’), (‘27’, ‘25’), (‘48’, ‘23’), (‘38’, ‘17’), (‘5’, ‘24’), (‘36’, ‘20’), (‘21’, ‘41’), (‘1’, ‘13’), (‘15’, ‘5’), (‘32’, ‘12’), (‘6’, ‘32’), (‘40’, ‘19’), (‘47’, ‘13’), (‘37’, ‘5’), (‘32’, ‘26’), (‘29’, ‘44’), (‘18’, ‘26’), (‘15’, ‘40’), (‘40’, ‘16’), (‘9’, ‘29’), (‘33’, ‘30’), (‘13’, ‘2’), (‘37’, ‘3’), (‘50’, ‘14’), (‘5’, ‘30’), (‘15’, ‘7’), (‘36’, ‘15’), (‘23’, ‘50’), (‘42’, ‘2’), (‘36’, ‘27’), (‘44’, ‘16’), (‘11’, ‘33’), (‘35’, ‘46’), (‘6’, ‘45’), (‘9’, ‘43’), (‘22’, ‘8’), (‘1’, ‘22’), (‘47’, ‘36’), (‘22’, ‘7’), (‘4’, ‘49’), (‘3’, ‘4’), (‘34’, ‘31’), (‘4’, ‘32’), (‘49’, ‘25’), (‘47’, ‘1’), (‘30’, ‘10’), (‘44’, ‘31’), (‘43’, ‘4’), (‘5’, ‘31’), (‘13’, ‘15’)]

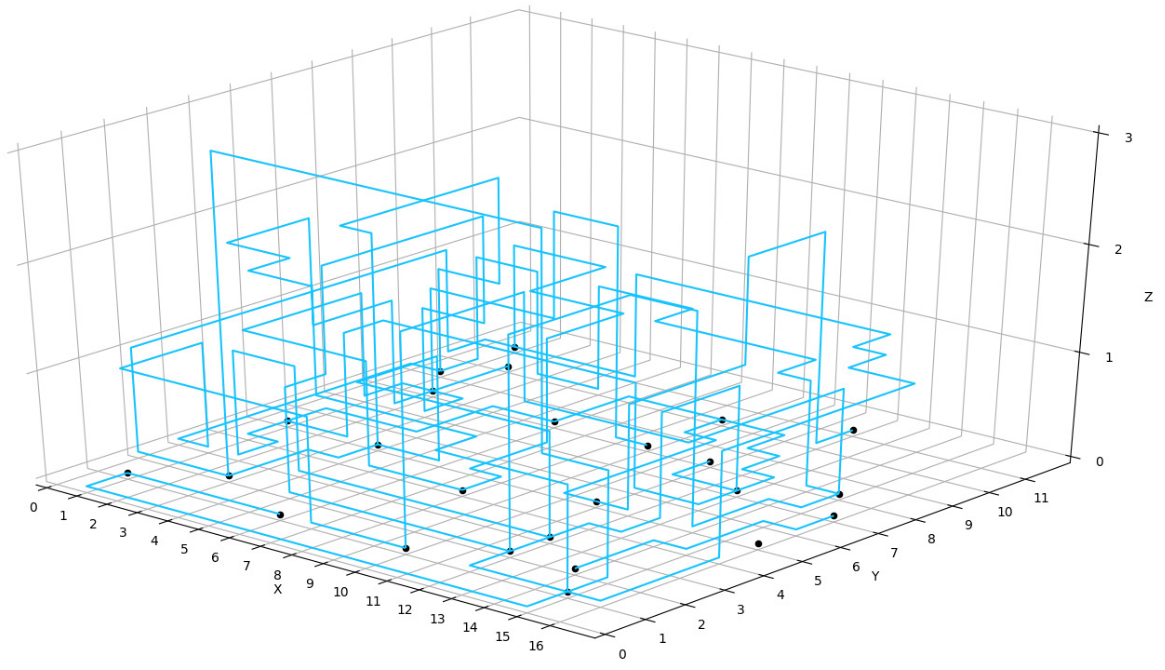
Netlist 5
[(‘27’, ‘9’), (‘8’, ‘29’), (‘20’, ‘34’), (‘34’, ‘37’), (‘26’, ‘25’), (‘48’, ‘6’), (‘18’, ‘4’), (‘13’, ‘25’), (‘45’, ‘30’), (‘25’, ‘24’), (‘45’, ‘35’), (‘4’, ‘34’), (‘23’, ‘31’), (‘41’, ‘11’), (‘34’, ‘5’), (‘1’, ‘17’), (‘5’, ‘16’), (‘28’, ‘23’), (‘16’, ‘43’), (‘35’, ‘22’), (‘32’, ‘5’), (‘1’, ‘22’), (‘21’, ‘48’), (‘14’, ‘50’), (‘19’, ‘12’), (‘39’, ‘17’), (‘45’, ‘43’), (‘9’, ‘28’), (‘13’, ‘26’), (‘35’, ‘6’), (‘24’, ‘44’), (‘17’, ‘2’), (‘31’, ‘10’), (‘1’, ‘13’), (‘20’, ‘31’), (‘26’, ‘50’), (‘29’, ‘41’), (‘5’, ‘41’), (‘49’, ‘48’), (‘1’, ‘11’), (‘36’, ‘7’), (‘10’, ‘6’), (‘46’, ‘19’), (‘3’, ‘18’), (‘33’, ‘12’), (‘19’, ‘28’), (‘29’, ‘46’), (‘42’, ‘14’), (‘37’, ‘30’), (‘40’, ‘49’), (‘32’, ‘42’), (‘2’, ‘4’), (‘27’, ‘2’), (‘25’, ‘9’), (‘5’, ‘9’), (‘37’, ‘38’), (‘11’, ‘20’), (‘44’, ‘45’), (‘21’, ‘30’), (‘13’, ‘33’)]

Netlist 6:
[(‘33’, ‘3’), (‘49’, ‘35’), (‘46’, ‘39’), (‘35’, ‘20’), (‘19’, ‘43’), (‘28’, ‘37’), (‘22’, ‘43’), (‘6’, ‘18’), (‘40’, ‘39’), (‘13’, ‘10’), (‘38’, ‘3’), (‘17’, ‘11’), (‘17’, ‘9’), (‘41’, ‘48’), (‘6’, ‘42’), (‘22’, ‘34’), (‘26’, ‘18’), (‘4’, ‘15’), (‘34’, ‘32’), (‘16’, ‘23’), (‘28’, ‘39’), (‘15’, ‘17’), (‘14’, ‘46’), (‘35’, ‘3’), (‘39’, ‘50’), (‘38’, ‘44’), (‘15’, ‘47’), (‘45’, ‘35’), (‘6’, ‘40’), (‘37’, ‘21’), (‘31’, ‘41’), (‘4’, ‘11’), (‘40’, ‘4’), (‘26’, ‘7’), (‘25’, ‘30’), (‘39’, ‘8’), (‘15’, ‘35’), (‘27’, ‘42’), (‘23’, ‘45’), (‘47’, ‘30’), (‘1’, ‘4’), (‘38’, ‘41’), (‘48’, ‘45’), (‘34’, ‘8’), (‘50’, ‘29’), (‘7’, ‘34’), (‘50’, ‘13’), (‘28’, ‘48’), (‘30’, ‘38’), (‘33’, ‘26’), (‘12’, ‘11’), (‘27’, ‘40’), (‘13’, ‘6’), (‘31’, ‘8’), (‘12’, ‘46’), (‘4’, ‘22’), (‘2’, ‘21’), (‘45’, ‘29’), (‘9’, ‘5’), (‘2’, ‘38’), (‘1’, ‘40’), (‘44’, ‘32’), (‘18’, ‘36’), (‘13’, ‘37’), (‘15’, ‘44’), (‘2’, ‘14’), (‘47’, ‘1’), (‘31’, ‘33’), (‘2’, ‘12’), (‘49’, ‘3’)]

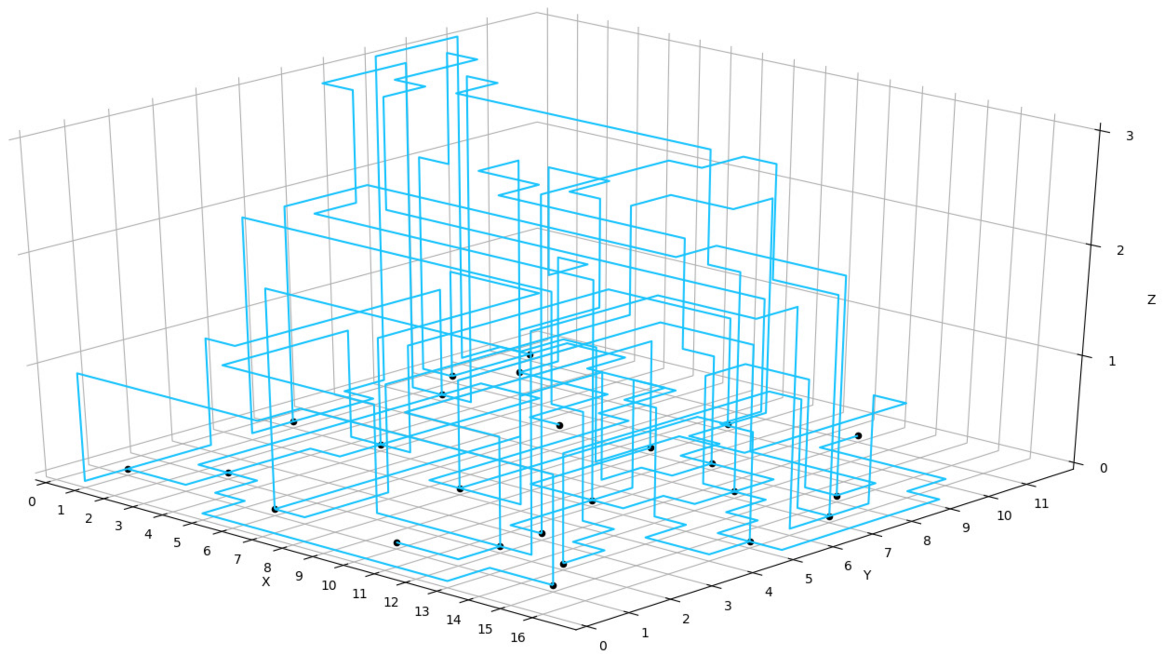
Appendix C

3D plots of the best solutions we found for all netlists matching the results in Table 1:

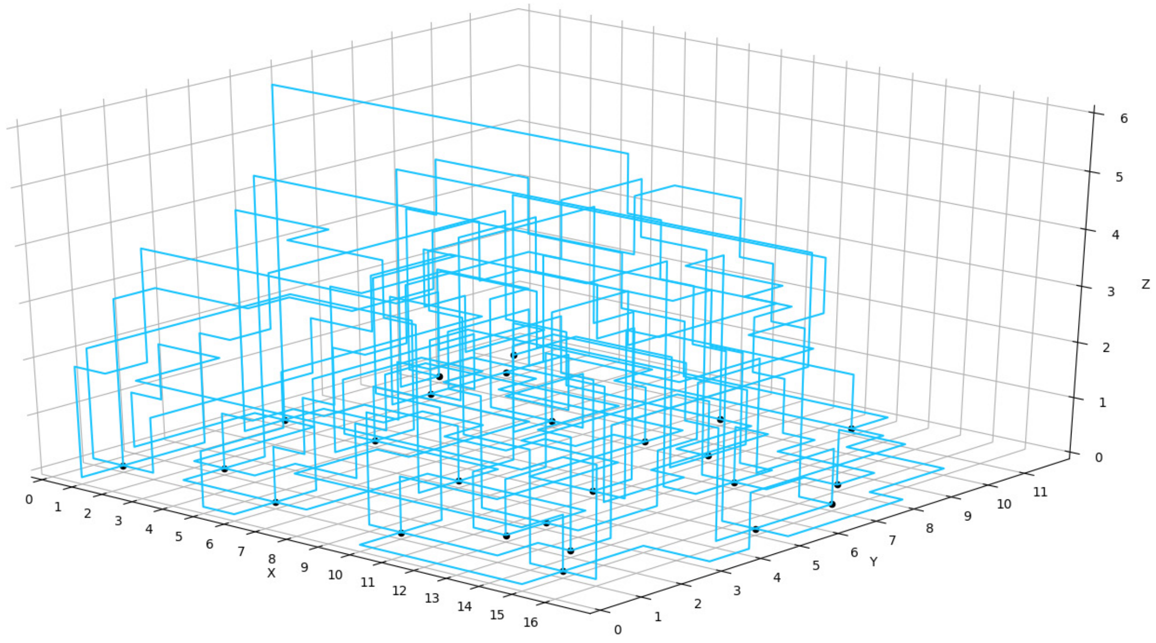
Netlist 1:



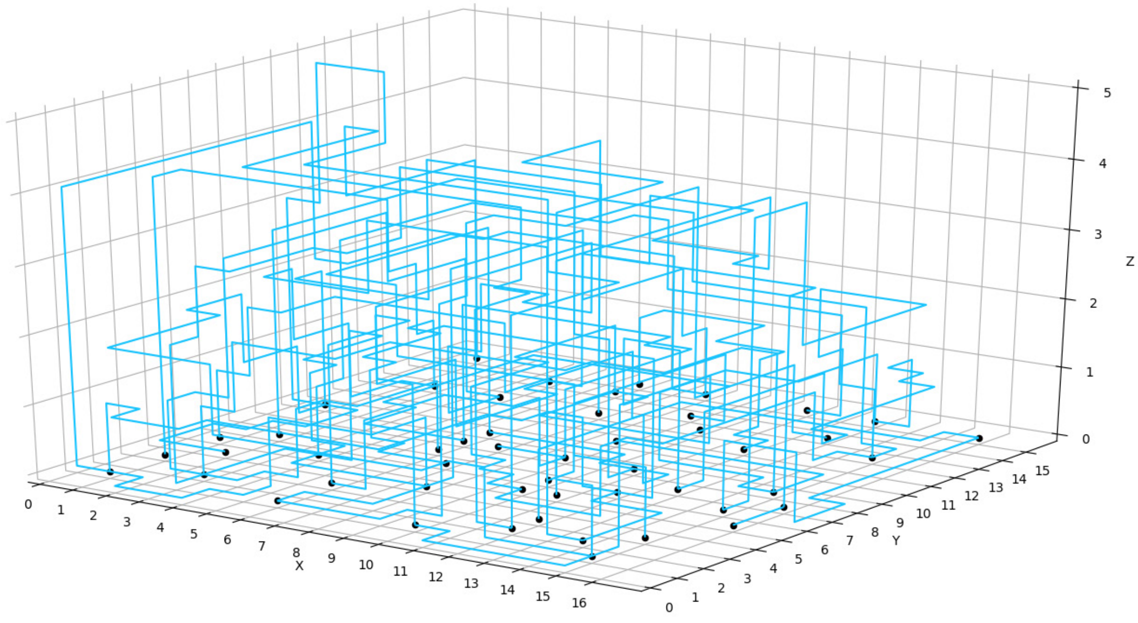
Netlist 2:



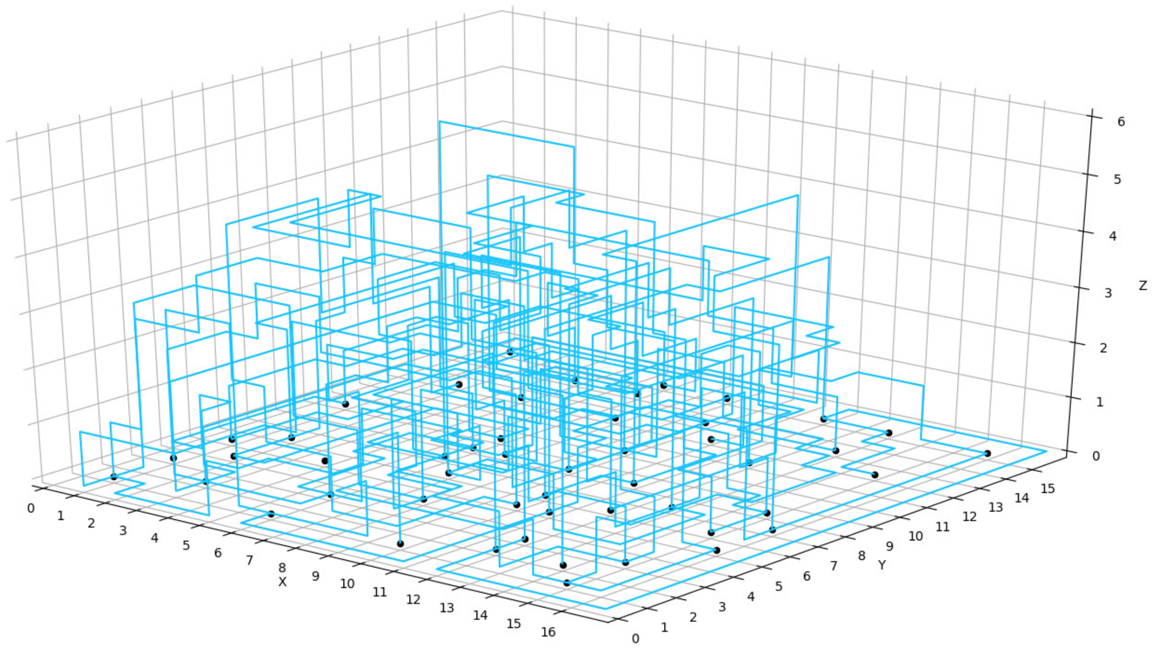
Netlist 3:



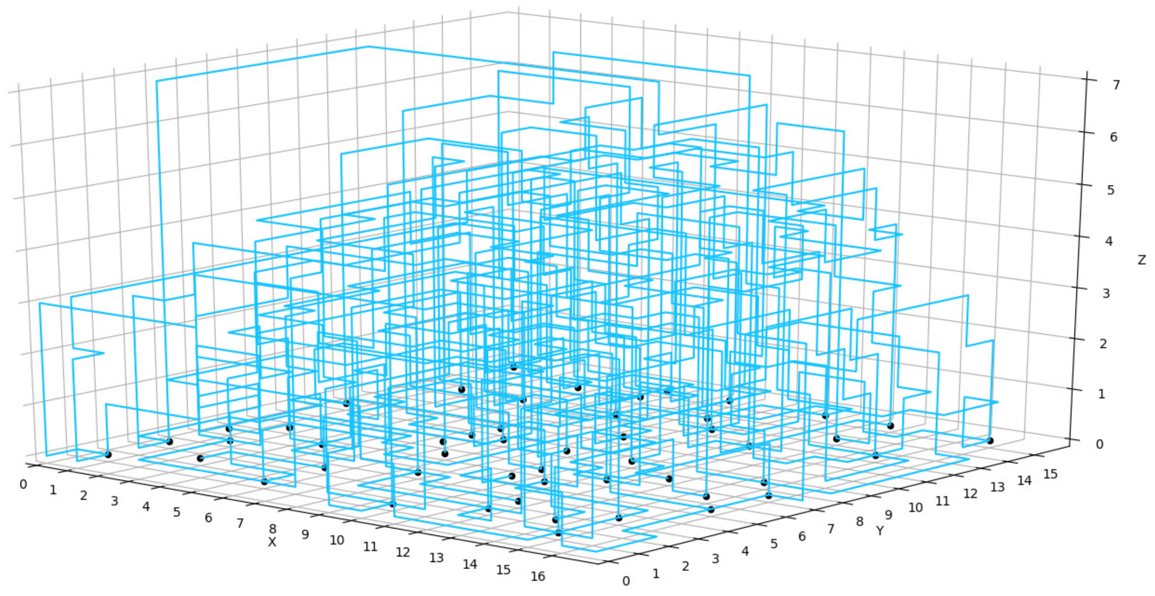
Netlist 4:



Netlist 5:



Netlist 6:


X
Y
Z
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16