Net Open Location Finder with Obstacles

第六組
組員：
B02901005 李瓖芸
B02901038 周昀
B02901037 吳宜凡

I. Problem Formulation

Given a set of routed net shapes $R = \{R_1, R_2, ..., R_j\}$, a set of routed net vias $V = \{v_1, v_2, ..., v_k\}$, a set of obstacles $O = \{O_1, O_2, ..., O_m\}$ on routing layers $L = \{M_1, V_1, M_2, V_2, ..., V_{n-1}, M_n\}$, with a design boundary B, spacing S, and via cost C_v, our problem is to find a set of paths $p = \{p_1, p_2, ..., p_l\}$ to connect all routed net shapes and routed net vias v to form a single connected component, while all the paths keep a minimum spacing of S from obstacles and the design boundary. Every path in the set of paths, either be a line or a via, forms connection only when the its endpoints touch an endpoint of another line, via or any point of net shapes. All the net shapes obstacles are presumed to be rectangular and overlapping between lines, vias, shaped will not be counted as connection. The goal is to find an efficient and stable means to minimise the overall cost

$Cost = \sum_{q=1}^{t} Cost(p_q) + Disjoint\ cost$, where

$Cost(p_q) = \begin{cases} length & p_q \ is\ V-line\ or\ H-line \\ C_v & p_q\ is\ via \end{cases}$ and

$Disjointcost = 2 \times (\#components - 1) \times (boundary\Delta X + boundary\Delta Y + \#ViaLayers \times C_v)$
$(boundary\Delta X = UR_x - LL_x\ of\ B,\ boundary\Delta Y = UR_y - LL_y\ of\ B,\ and\ \#ViaLayers = \#MetalLayers - 1)$

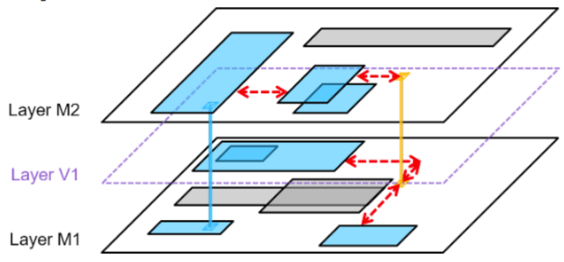


Fig. 1 An example of a single net with large scattered routed net shapes

II. Overview of Our Algorithm

Our algorithm can be divided into 3 parts, overviewed as follows:

Step 1:
We parse the routed shapes, obstacles, and vias from the input file and are then converted to nodes to construct a graph for each layer. To flatten the 3D structure into 2D graphs, shapes are duplicated to the neighbouring layer connected by vias, regardless of obstacles. These shapes are sorted by the y-axis of their points and connected with vias and lines if there is overlap.
Step 2:
Because the structure is now flattened, we can then apply multi-threading to separate layers and do operations in each graph. Inside every layer, the nodes are sorted by x-axis and y-axis values, so as to find trivial connections without intersecting obstacles, and the valid connections are then added to the graph as edges of two nodes. From the completed weighted graph, we construct minimum spanning subtrees connecting the nodes.
Step 3:

Now we have the disjoint set of MST subtrees and their roots, for each two local MSTs we scan their children to find the nearest connection if an L shaped segments can connect the shapes without intersecting an obstacle.

III. Detail of Our Algorithm

A. Parsing and data structure

The net-open location finder can be seen as a rectilinear Steiner minimal tree (RSMT) problem at its core, which, even without obstacles, is a well-known NP-complete problem. Taking obstacles into consideration, a obstacle avoiding rectilinear Steiner minimal tree (OARSMT) is often constructed to employ computational geometry techniques for practical applications. What this problem differs from RSMT lies in the routed shapes: we are no longer given terminal points, which makes it very difficult to build a complete MST because the Manhattan distance and obstacle penalty becomes increasingly hard to calculate due to the overlapping net shapes and obstacles. Instead of constantly updating the weights of nodes of the non-rectangular objects for each connection, we chose to sort values on the X and Y-axis for line connections, and Y-axis for vias alike to get the local minimum.

To map the input objects to a graph, we built our data structure as follows:

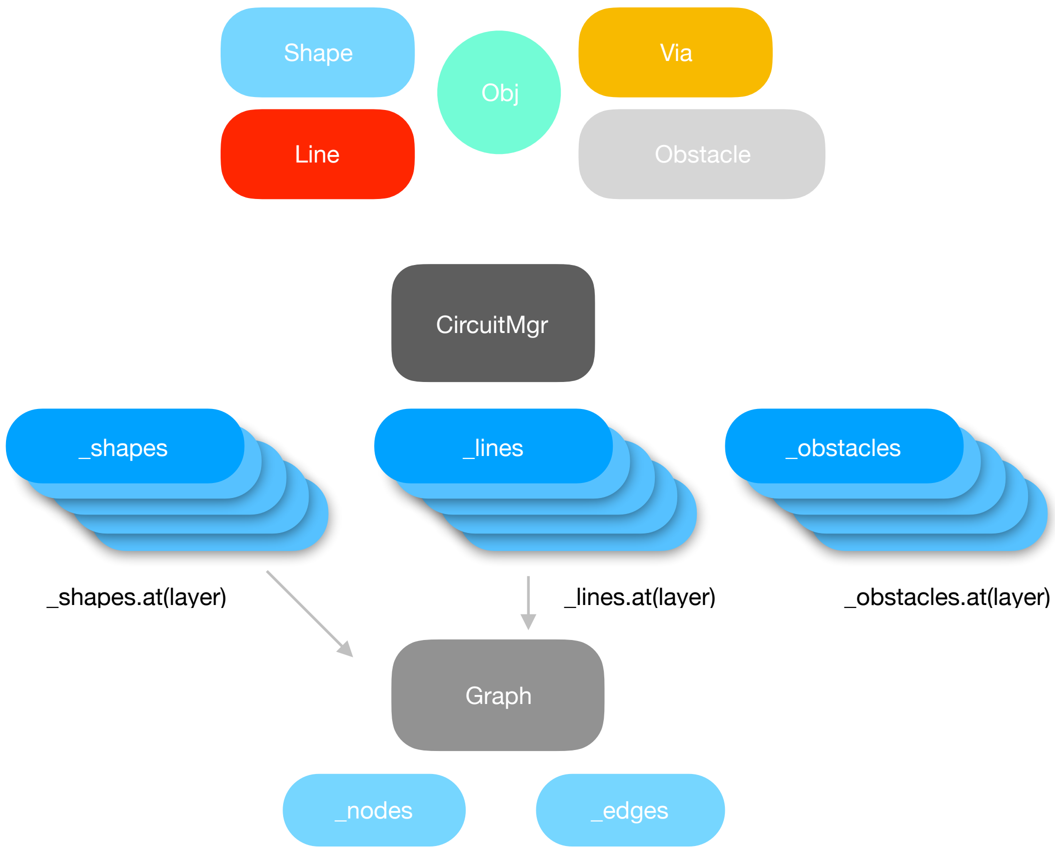
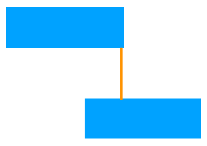
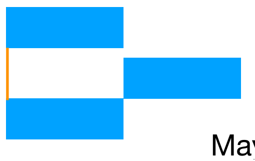


Fig. 2 Data structure

The classes Shape, Line, Via, and Obstacle inherited from class Obj, and then the CircuitMgr stores a vector of vectors of each class except for via, so objects can by queried by given layer to construct a graph for a single layer. After graph construction, we apply initial trivial connection for vias by sorting the Y-axis values of shapes. For each shape S at layer l, if the Y-axis values overlaps with another shape S' at layer $l + 1$ (except for boundary conditions), if a via is able to point touch the two objects at the same time, add the via to our result. Note that only comparison between two neighbouring layers are required since a via penetrating more two layers takes more risks to produce line waste or disconnection.



Adding trivial vias



May produce line waste

Fig. 3 Via connection

B. Constructing spanning graph of a layer

After getting the routed shapes and obstacles of a layer as nodes of a graph, we set the weight of edges with the distance between the two shapes if there are no obstacles between. We hereby assume that the shortest distance between two shapes to be the non-overlapping coordinate value differences. In our input case, there are so many shapes overlapping with one another or obstacles overlapping with other obstacles, so this is a good way to find the connected components without further hassle. With these edges, we use Prim's algorithm to find a minimum spanning tree to find the necessary lines to connect a subset of routed shapes and attach a set id to each subset for the next step. For the provided test cases, this step effectively reduces the number of disjoint set. The separation of layers allows for multi-threading to speed up the process.

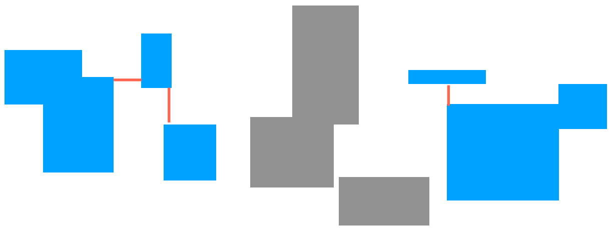


Fig. 4 Line connection

C. Merging the disjoint sets

To minimise the edge weight or path length between two disjoint sets, we first traverse each element of a MST tree generated by the last step to check if both L-shaped segments of an edge intersects an obstacle. After traversing all the elements, we can get the shortest path between disjoint sets.

IV. Experiment Results

case/# of layers	# of shapes/ obstacles	CPU Time(ms)	Memory(MB)	Path Cost	Disjoint Cost	Total Cost
case1 / 3	1503 / 414	1360.08	12.4805	24403	0*20040	24403
case2 / 5	4518 / 4773	43154.7	13.2656	566334	3*110240	897054
case3 / 8	97146 / 79012	6.45072E+06	722.738	17277389	40*13200420	69294189

From our table we can see that our algorithm generally achieves adequate results. However the runtime increase is severe because the time complexity of indexing the coordinates of objects is $O(n^2)$ and building a minimum spanning tree is $O(E\ lgE)$. We need to construct a complete graph among the nodes, and this has become the bottleneck of our algorithm.

Visualised Results
case1:

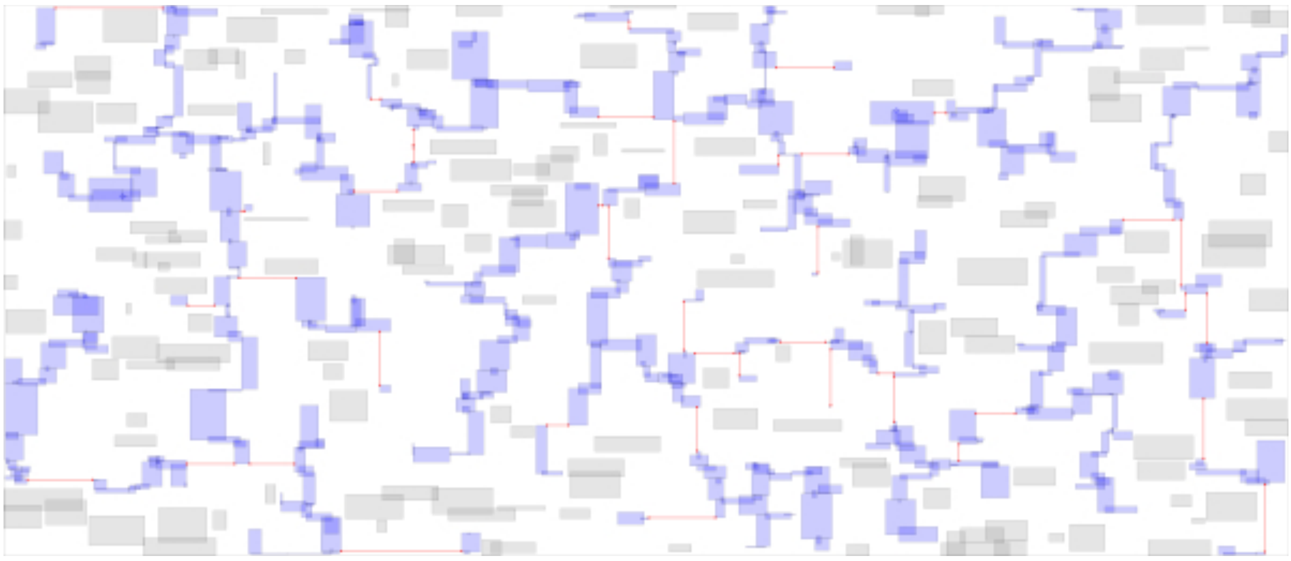


Fig. 5-1 Visualised result of case1

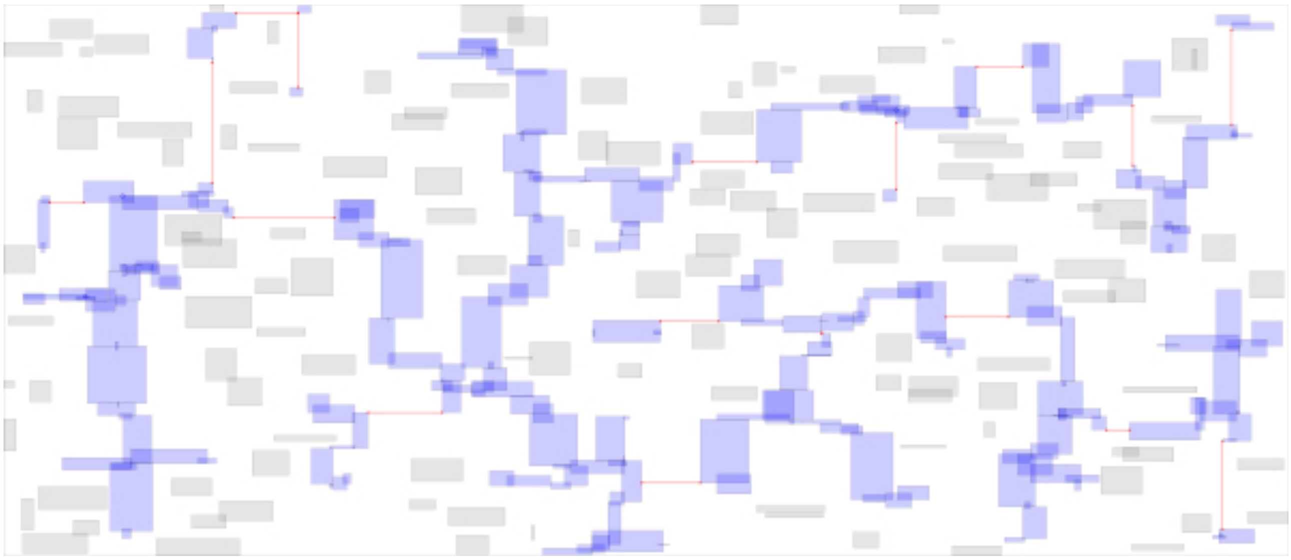


Fig. 5-2 Visualised result of case1

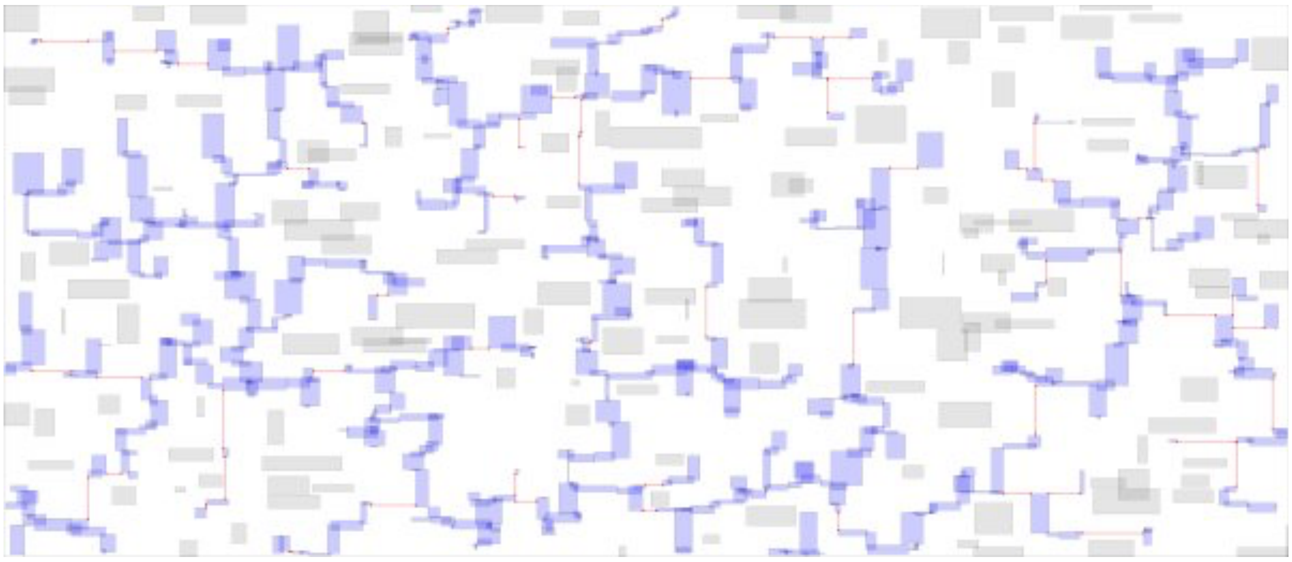


Fig. 5-3 Visualised result of case1

case2:

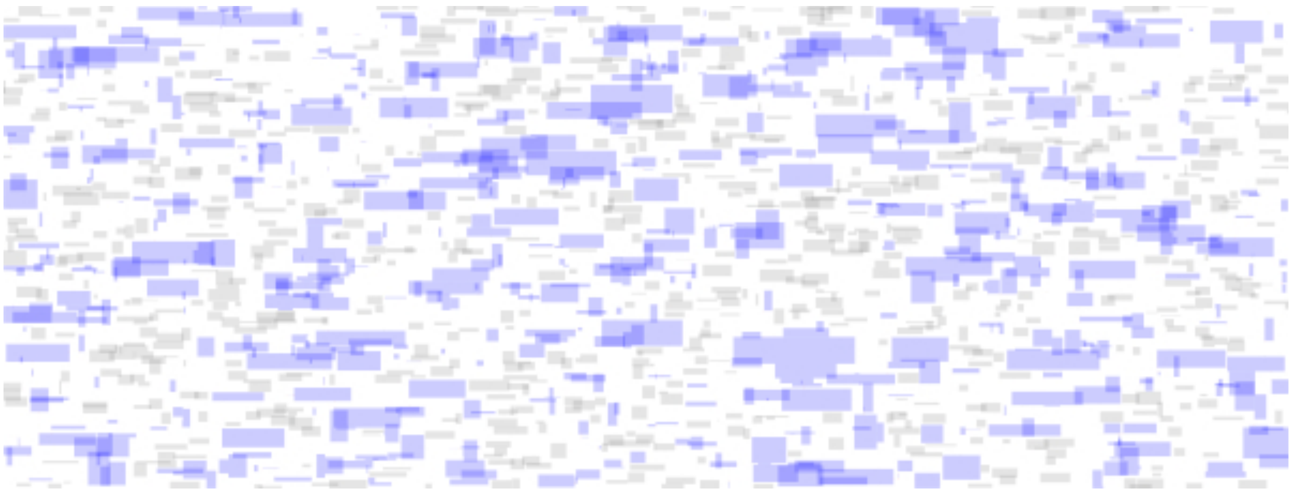


Fig. 6-1 Visualised result of case2

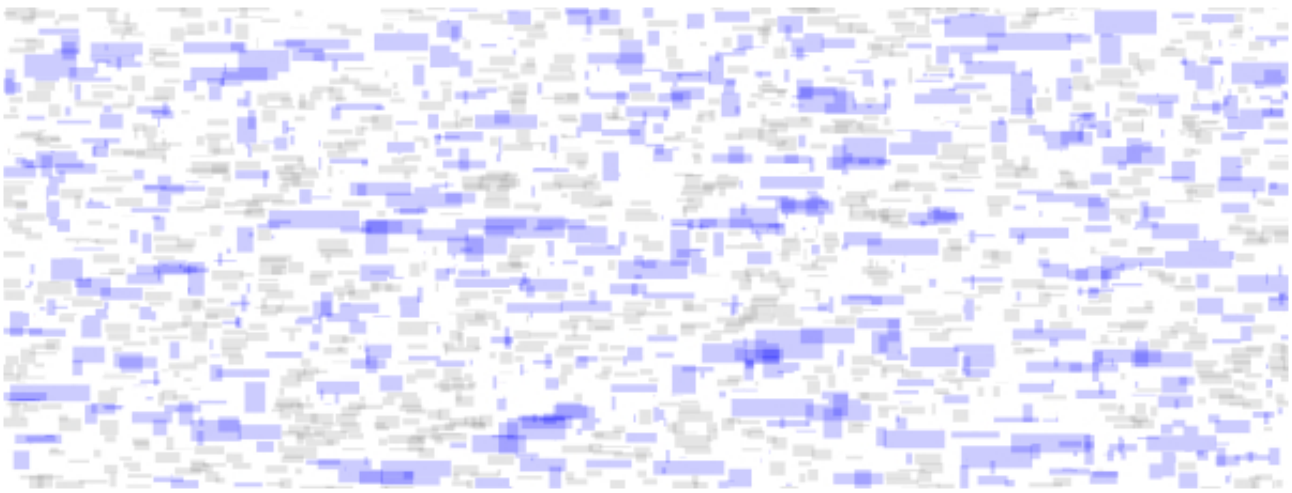


Fig. 6-2 Visualised result of case2

Fig. 6-3 Visualised result of case2

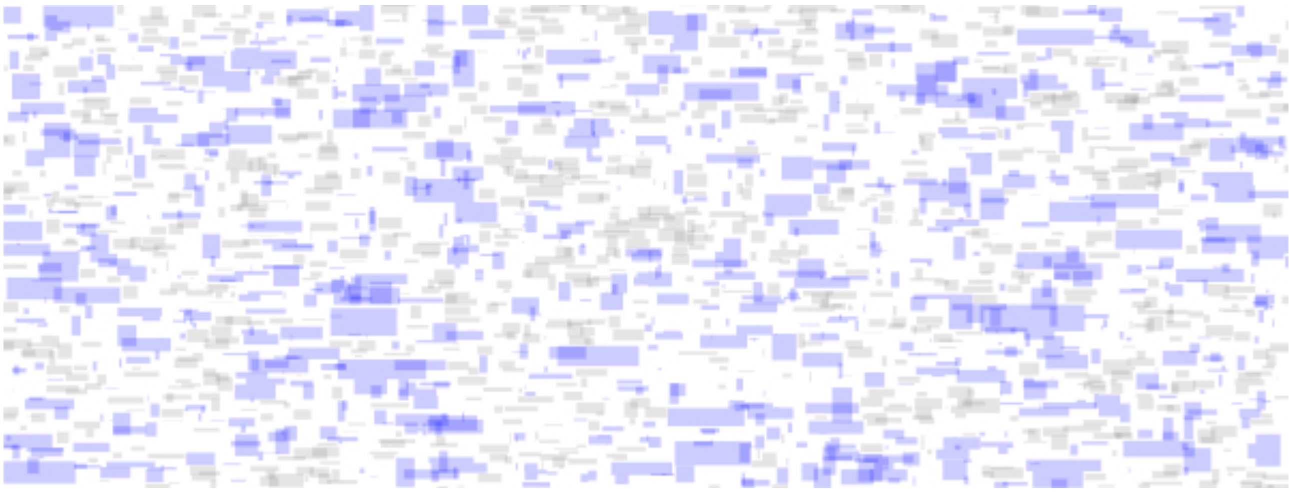


Fig. 6-4 Visualised result of case2

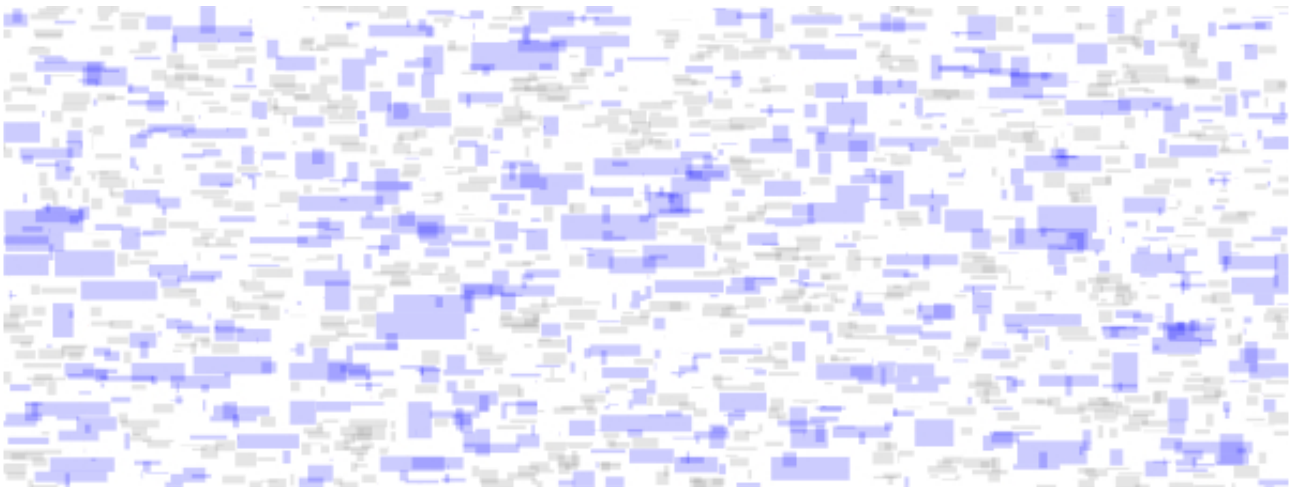


Fig. 6-5 Visualised result of case2

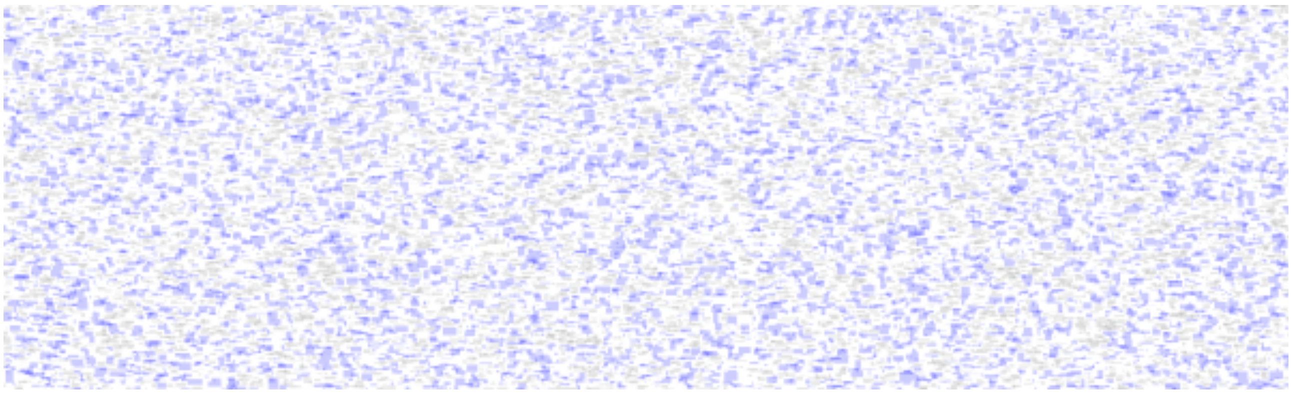


Fig. 7-1 Visualised result of case3_1

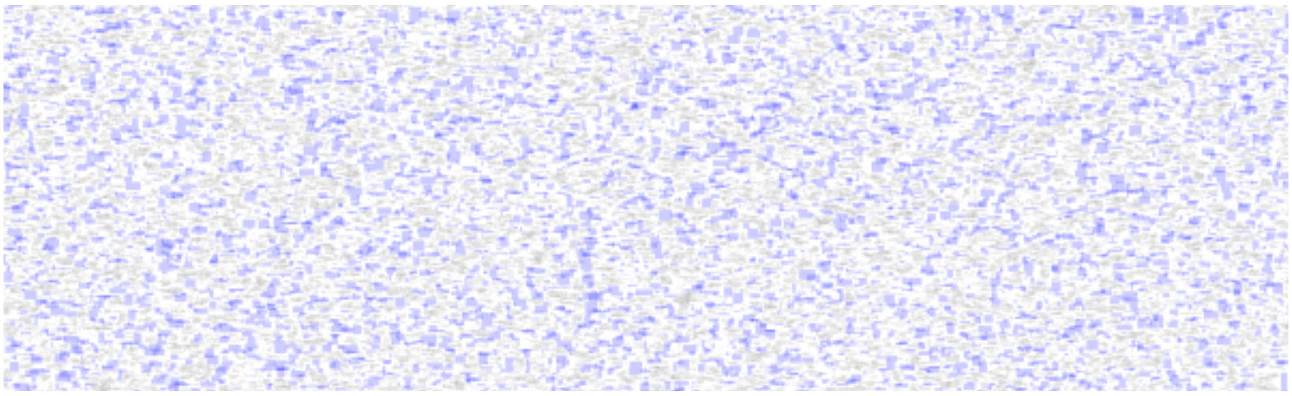


Fig. 7-2 Visualised result of case3_2

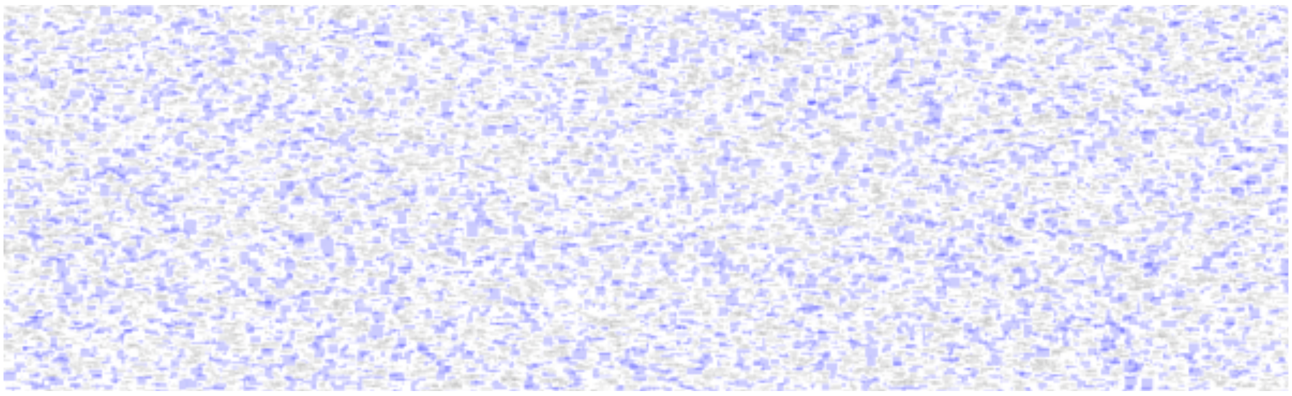


Fig. 7-3 Visualised result of case3_3

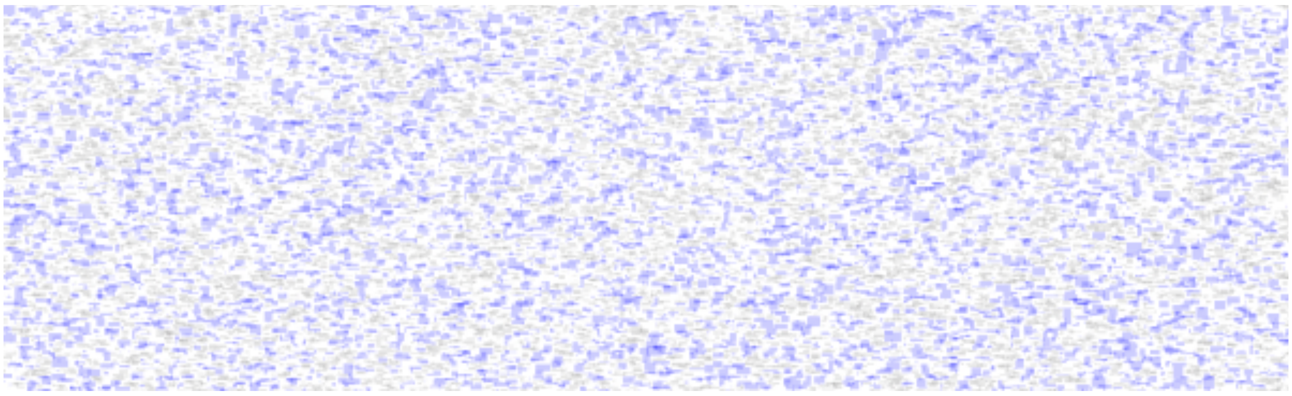


Fig. 7-4 Visualised result of case3_4

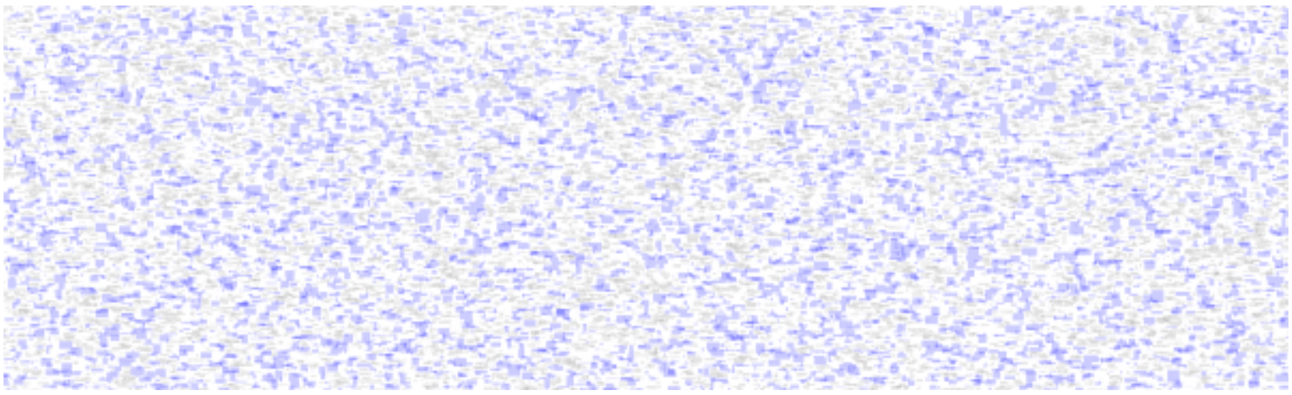


Fig. 7-5 Visualised result of case3_5

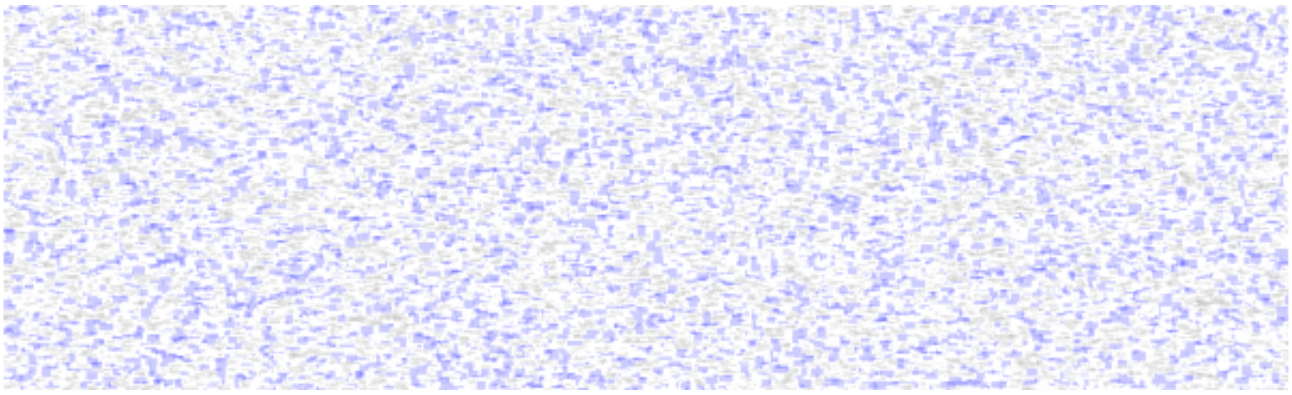


Fig. 7-6 Visualised result of case3_6

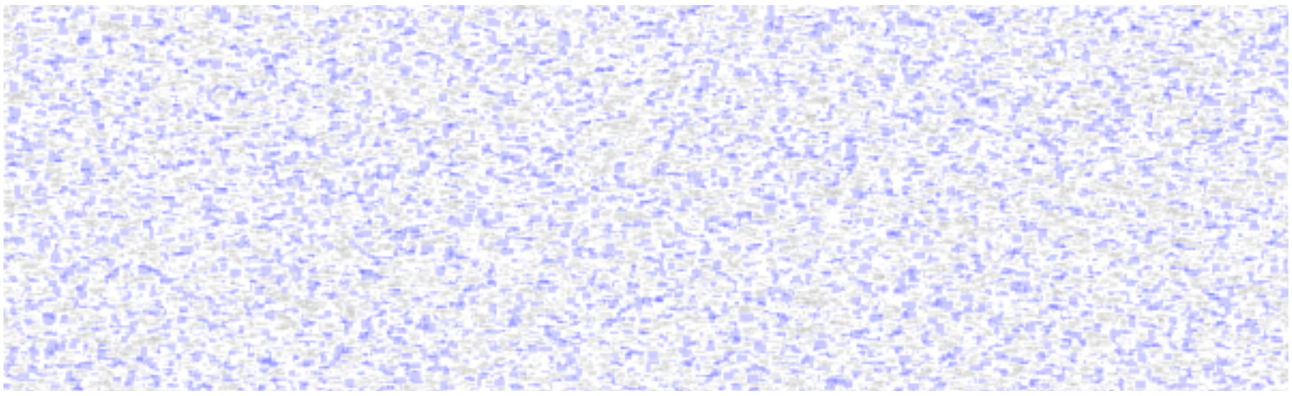


Fig. 7-7 Visualised result of case3_7

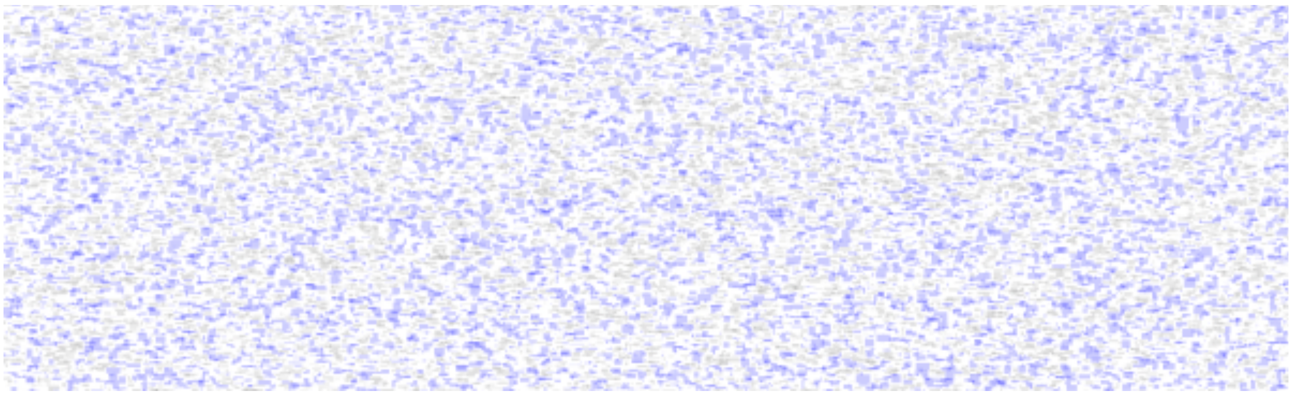


Fig. 7-8 Visualised result of case3_8

V. Conclusion

We tried to use a seemingly straightforward approach to solve the problem of net open location finder by first reducing the 3D net into 2D layers by adding trivial vias, and then constructed MST to find the shortest path between the routed shapes. For disjoint sets inside a layer, we iterated through their elements to find the shortest path to join sets. As a result the runtime is comparably large but produces good results. To improve the performance we'll have to reduce the complexity to make it more practical in the routing process.