Algorithm Final project
- Net Open Location Finder with Obstacles

Team 10: B03202040 Chun-Ju Wu 0960573434, B04901009 Meng-Jin Lin 0936141249, B04901025 Hong-Chi Chen 0905379256

June 20, 2017

1 Introduction

Finding the shortest path for several terminals is a well known unsolvable problem. For example, problem such as "Steiner Tree Problem", the decision variant of the problem in graphs is NP-complete (which implies that the optimization variant is NP-hard). In our problem, we have several layers rectangle routed net shapes that need to be connected, and some rectangle obstacles that can't be crossed. This should also not be able to solved by polynomial time. Thus, the only way we can process this is to use some heuristics to approach the minimum path. There are some past researches which give some good approach to the problems such as obstacle-avoiding RMST(Rectilinear Steiner minimal tree) problem and multilayer OARSMT(Obstacle avoiding rectilinear Steiner minimal tree)[2]. These past researches give us some idea to approach this problem and also let us proposed some future direction of improvements.

2 Problem Formalism

In this problem, we are proposed to connect large scattered routed net shapes. We need to connect all the routed net shapes and the vias while having minimum cost. There are three kinds of objects in this problem. They are routed net shapes R, routed net vias V, and obstacles O. They are blue, yellow, and grey in Figure 1, respectively. There are designed boundary B which is the range where our paths can located, and the minimum spacing S which is the spacing that we need to keep with the obstacles and the boundary. In the problem, there are several layers, the cost that required to cross the layers are Cv. The path are required to be horizontal or vertical. Steiner points are allowed. Points have no area and lines have no width. The total cost can be calculated by simple equation below:

$$Cost = \Sigma Cost(Pq) + Disjointcost \tag{1}$$

where the cost of Pq for horizontal and vertical lines are the length, and the cost of via is Cv.

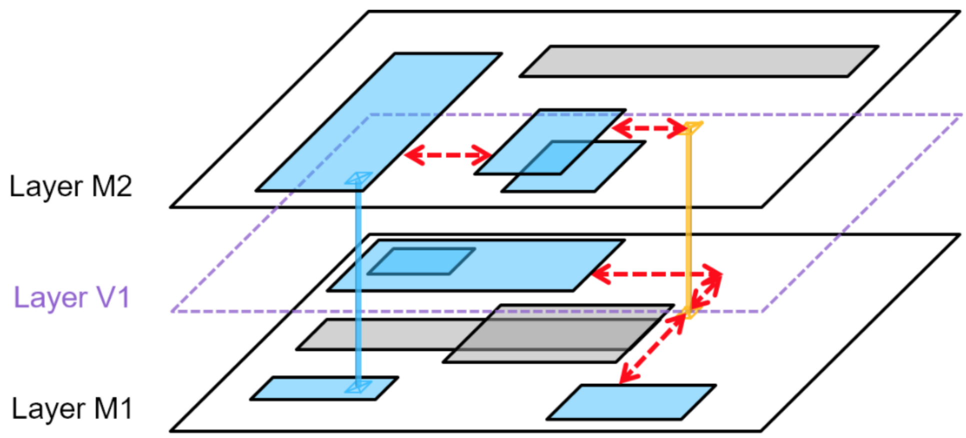

Figure 1: Problem example.

3 Algorithm and Time Complexity Analysis

3.1 Graph Construction

3.1.1 General Idea

In the graph construction, we want to change the input data into several layers' corresponding graphs. In this project, the overlapped routed shapes and obstacles are first detected. Then we construct the vertexes that we need for the graph. The vertexes are chosen if it is routed shapes' or obstacles' vertex and not in other routed shapes or obstacles. Then the edge between each vertex is constructed if there is no obstacle or routed shape inside the rectangle formed by these two points.

The process can be divided into two main tasks. The first task is to find the overlapped routed net shapes and the obstacles. The second task is to find the edge that we want to connect correspond to the graph.

3.1.2 Find the Overlapped Routed Net Shapes and the Obstacles

The algorithm can be divided into three steps. In the first step, we use balanced binary search tree to store what shapes are at a specific x-coordinate (which may be a shape's right or left x-coordinate). After this step, we now can construct connected shapes by unioning shapes which are overlapped. Now we have connected shapes and their points including points which are in the overlapping part of shapes. In the third step, We want to construct a new class Polygon to store polygons which are corresponding to connected shapes and their points on corners. The points in the overlapping part will be deleted after the determination of whether the point is inside the connected shapes. The remaining points then will be store in a polygon. In the end, a polygon which carries the information of its points will be add to the vector of polygons.

Data: originally we have some shapes that are rectangular on the 2D space which are called R1 R2....... Rn. They have coordinates of four points R1_ld, R1_lu, R1_rd, R1_ru (left right up down) x_coord(R1_l) is the x coordinate of the left edge..... R1 to Rn is already sorted by their Ri_l we want to have output a set with many polygons which are the overlapped R.

Result: Inputs for the second part

```
//We need to construct points projected to the x axis // points contain information that it
 is belong to a set of Ri_l..... , Rj_r....... // points should store in balanced binary tree
 point has vector R1_l, R2_r......
Balanced Binary Search Tree(BBST) B int x, vector<shape> Ri_l for i from 1 to n do
|  if x_coord(Ri_l) not in B then
|  |  insert (x_coord(Ri_l), Ri_l) into B
|  end
|  else if x_coord(Ri_l) in B then
|  |  B.find(x_coord(Ri_l))->second.pushback(Ri_l)
|  if x_coord(Ri_r) not in B then
|  |  insert (x_coord(Ri_r), Ri_r) into B
|  end
|  else if x_coord(Ri_r) in B then
|  |  B.find(x_coord(Ri_r))->second.pushback(Ri_r)
end
```

Algorithm 1: First Step: projection

Data: output from first part
Result: connected shapes
//now we have a balanced binary search tree B with "projected x coordinates and its belong to which rectangles" //we want to find the overlapped components now.//B.size = m
New BBST B1//store the rectangles that have are on the line x = l, l will increase from low to high Vector set A // store the connected shapes. Here we think the connected components should not have size too large, thus we simply store it in a list or some other things but not using a graph to store it

```
for i from 1 to m do
    List Need_to_delete x = extract min(B) for object in x do
        if object = Rk_l then
            //k doesn't matter. k can be all number from 1 to n. B1.pushback(Rk) construct
             new set SRk for object_inter in B1/Rk // / means without do
                if object_d <= object_inter_u or // for obstacle, this should be < then
                    Acobject_inter = Acobject_inter ∪ S //c is the index of vector of A
                     which contains C (U is union) S = Acobject_inter
                end
                else if object_u >= object_inter_d // for obstacle, this should be > then
                    Acobject_inter = Acobject_inter ∪ S //c is the index of vector of A
                     which contains C S = Acobject_inter
                else if object_u >= object_inter_u and object_d <= object_inter_d // for
                 obstacle, this should be > then
                    Acobject_inter = Acobject_inter ∪ S //c is the index of vector of A
                     which contains C S = Acobject_inter
                else if object_d >= object_inter_d and object_u <= object_inter_u // for
                 obstacle, this should be > then
                    Acobject_inter = Acobject_inter ∪ S //c is the index of vector of A
                     which contains C S = Acobject_inter
            end
            if S only contains one shape then
                A.pushback(S)
            end
            else if object = Rk_r then
                Need_to_delete.pushback(Rk_r)
        end

        for object in Need_to_delete do
            B1.delete(object) //this need to first check and delete at first in obstacle case
        end
    end
end
```

Algorithm 2: Second Step: find overlapped

Data: output from second part
Result: polygons with the information of its points
//Now we have connected shapes and their points including points which are in the overlapping part of shapes. We want to construct a new class Polygon to store polygons which are corresponding to connected shapes and their points on corners.
The points in the overlapping part will be deleted after the determination of whether the point is inside the connected shapes. The remaining points then will be store in a polygon.
In the end, a polygon which carries the information of its points will be add to the vector of polygons.
//vector of Polygons P

```
for i from 1 to number_of_connected shapes do
    List L //store the ran rectangles
    construct new Polygon poly
    new vector <Point> poi//store the points of a connected shape
    for object in A do
        for object_other in L do
            if A's points are not in object_other then
                poi.pushback(Point)
            end
            if object_other's point are in object then
                poi.erase(Point)//delete the points inside the connected shape
            end
        end
        L.push_back(object)
    end
    for j from 1 to poi.size do
        add point in poi[ j ] to poly
        //one poly stores points of one shape
    end
    P.push_back(poly)//construct a vector of polygons
end
```

Algorithm 3: Third Step: construct polygons

3.1.3 Construct the Edge

In this part, we first detect the possible x values for all the corners of the routed net shapes and the obstacles, which is O(NlgN). Then we insert the region that can't be passed for each x value, which is O(N^2). After finished creating the regions that can't be passed, we use the result in Section 3.1.2 to construct the "pins"(node) in the graph. The pins are the vertex of the routed net shapes(obstacles) which does not covered by other shapes. This step is O(N), because these points are already constructed in Section 3.1.2. At the last part, we construct edge between two point if there is no object inside the rectangle constructed by these two points. To do this, we construct edge for each pins from low x value to high x value. For each pin, we run from the low x value and store the point on that x in a list. If the point stored is blocked by the routed net shapes or obstacle, then we remove it from the list(maybe other data structure). To see if it is blocked, we could use the information of the region can't be passed that we stored for each x value. The details are describe in the pseudo code below. It is divided into two parts because of its length. The first part contains the algorithm of the preprocess:

```
Data: In this pseudo code, we want to construct the edge for each points. We have routed
      net shapes R1 R2....... Rn. Each shape has coordinates of four points R1_ld, R1_lu,
      R1_rd, R1_ru (left right up down). Obstacles Oz.......... Om. We contruct edge for
      every unblocked points(unblocked means that they do not have a obstacle or polygon
      inside the rectangle produced between the points). The first quadrant is denoted
      R1,.. and so on. Now we have several connected component polygons P1....
      Pk(routed net shapes), PO1..... POl(connected obstacle) and two connected sets of
      shapes(obstacles) A1...Ak, B1....Bl which are the sets that store the connected
      shapes(obstacles). Vi_1...Vj_z are the vias that is between i and i+1 layer
Result: Inputs for the second part
first construct the x projection of each points, and make the region of y values can't pass for
 each x value
//******Note that we need to sort the x value of the points of each polygon and obstacle
 first.
map B(coordinate_x, vector <cannot_pass_region>) //finally x1 to xj,
 cannot_pass_region store (yd,yu) which is the range can't be passed. Contain lots of
 different x coordinates. Each x coordinates have one vector of cannot_pass_region
for i from 1 to n do
    if Ri_l not in B //check if it is contains in the first elements in B then
        B.insert(Ri_l, )
    end
    if Ri_r not in B then
        B.insert(Ri_r, )
    end
end
for i from 1 to l do
    if Oi_l not in B then
        B.insert(Oi_l, )
    end
    if Oi_r not in B then
        B.insert(Oi_r, )
    end
end
for i from 1 to n do
    yu = Ri_u
    yd = Ri_d
    for B.node->first in [ Ri_l,Ri_r) do
        B.node-> second.pushback(yd, yu)
    end
end
for i from 1 to l do
    yu = Oi_u
    yd = Oi_d
    for B.node->first in[ Oi_l,Oi_r) do
        B.node-> second.pushback(yd, yu)
    end
end
//get all the points
vector point Point
for i from 1 to k do
    for p in P[ i ] do
        Point.pushback(p)
    end
end
for i from 1 to l do
    for p in PO[ i] do
        Point.pushback(p)
    end
end
//Same for Vias, we push back the points of vias
sort Point by x value
```

Algorithm 4: Preprocess

The second part is the algorithm of finding the edge and construct the graph.

```
Data: Outputs from the preprocess
Result: Graph(V,E)
for point p in Point do
    L_point_1//store the points not blocked and upper than the point
    L_point_2//store the points not blocked and down or equal than the point
    now_x_1 = 0
    pre_x_1 = 0
    now_x_2 = 0
    pre_x_2 = 0
    for p_left in Point //from low x value to high x value because it is already sorted do
        if p_left.x >= p.x then
            break
        end
        else
            //delete the points blocked by the obstacles between [ pre_x,now_x], and add
             the new point into the list
            if p_left.y > p.y then
                now_x_1 = p_left.x for B.node.coordinate_x in [ pre_x,now_x) do
                    for ppoint in L_point_1 do
                        if B.node.cannot_pass_region.yd <= ppoint.y&&
                         B.node.cannot_pass_region.yu > p.y //u mean the up point, which
                         has larger y value; d means down then
                            L_point_1.delete(ppoint)
                        end
                    end
                    pre_x_1 = p_left.x L_point_1.pushback(p_left)
                end
            end
            else
                now_x_2 = p_left.x for B.node.coordinare_x in [ pre_x,now_x) do
                    for ppoint in L_point_2 do
                        if B.node.cannot_pass_region.yu >= ppoint.y&&
                         B.node.cannot_pass_region.yd < p.y then
                            L_point_2.delete(ppoint)
                        end
                    end
                    pre_x_2 = p_left.x L_point_2.pushback(p_left)
                end
            end
        end
    end
    for point in L_point_1 do
        construct edge(point,p)
    end
    for point in L_point_2 do
        construct edge(point,p)
    end
end
```

Algorithm 5: Construct Edge

This process can be summarized in Figure 2. The result graph is shown in Figure 3.

3.1.4 Time Complexity Analysis

For findindg the overlapped routed shapes, the algorithm is O(N) + O(N^2) + O(N^2) . The total time complexity of this step is O(N^2). For constructing the edge, we can see that the algorithm is O(NlgN) + O(N^2) + O(N) + O(N^3) from the above discussion. This leads to the time complexity which is O(N^3)

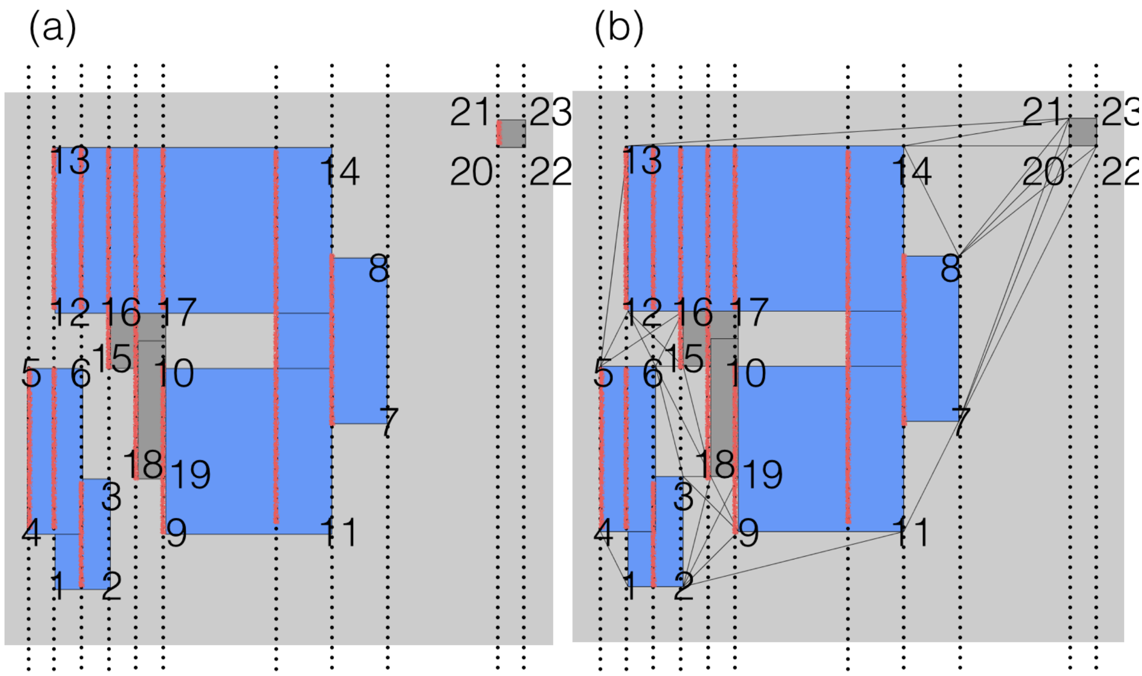

Figure 2: Find the edge. The dash line denotes the possible x values. The red region denotes the region can't be passed which is stored in the possible x values. In (a), we can see that how the possible x values and the region can't be passed been constructed. Also, we construct the pins. In (b), we construct the edge correspond to the pins.

3.2 Minimum Spanning Tree

3.2.1 construct MST

To construct MST, we first run through every connected-components. For each vertex in a connected-component, we use Dikstra's Agorithm to find the distance from vertex to every other point in the graph. after finishing Dikstra in a connected-component, we can find the distance from this connected-component to every other connected-component. Then we find distance between every two connected-component, thus we finished construct MST.

And we can treat every connected-components as single vertex. We solve it by using Prim's Algorithm.

3.2.2 Time Complexity Analysis

Construct MST: because we have to run through every vertex to find SSSP, the complexity is (V * complexity of Dikstra Algorithm) and Diksktra complexity = E lg V

MST : complexity = V^2 lg V using Prim's Algorithm.

4 Result

This section contains some selected results and the pictures of the result connecting figures. There are still things that can be improved, such as the connection between different layers, and the overlaid connecting lines. Here are the result of case 1. Number of disjoint components before inserting paths: 113
Number of components after inserting paths: 2
Path Cost = 36739
Disjoint Cost = 1 * 20040
Total Cost = 56779
case2:
Number of disjoint components before inserting paths: 1602
Number of components after inserting paths: 1
Path Cost = 755421
Disjoint Cost = 0 * 110240

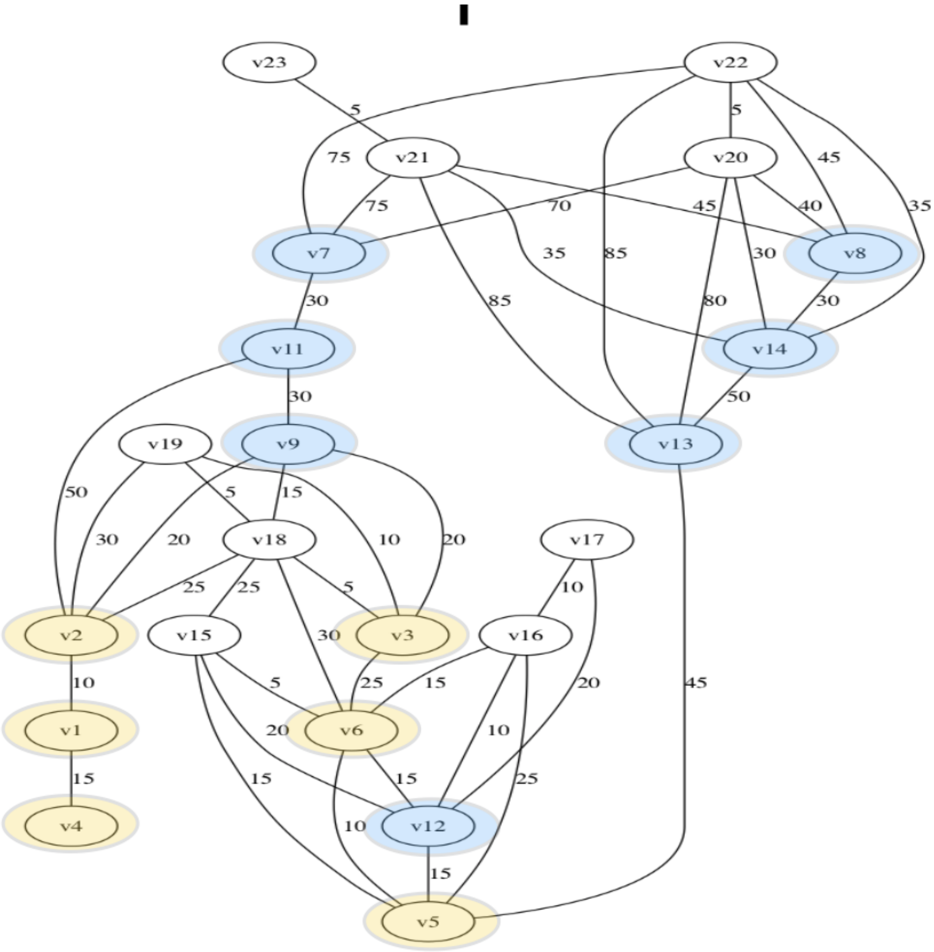

Figure 3: The constructed graph from Figure 2.

Total Cost = 755421

5 Future Method and Possible Improvements

5.1 Concept of the Algorithms

In this part we present a possible better approach for this problem. It could be decomposed into three steps. First, construct the ML-VG(V,E) graph, which can be derived from the problem input. Second, find the steiner points inside the ML-VG(V,E) graph. Third, find the minimum spanning tree for the steiner points and the originally connected pins(the definition of connected pins will be defined below).

5.2 ML-VG Construction

In this step, we need to construct the 3-D graph corresponding to the image. The first idea that comes into my mind is that we could consider only the four corner points as the initial vertex that we want to connect. The according to the paper, we can get R(P) and R(O). Then we can construct the 3-D VG.

5.2.1 The Construction Process

For the construction of the graph, we follow the method described in previous paper[1][2]. Here we first transform the graph on 2-D multilayer into 3-D cases. The transformation can described as the following(N_l is the number of layers):

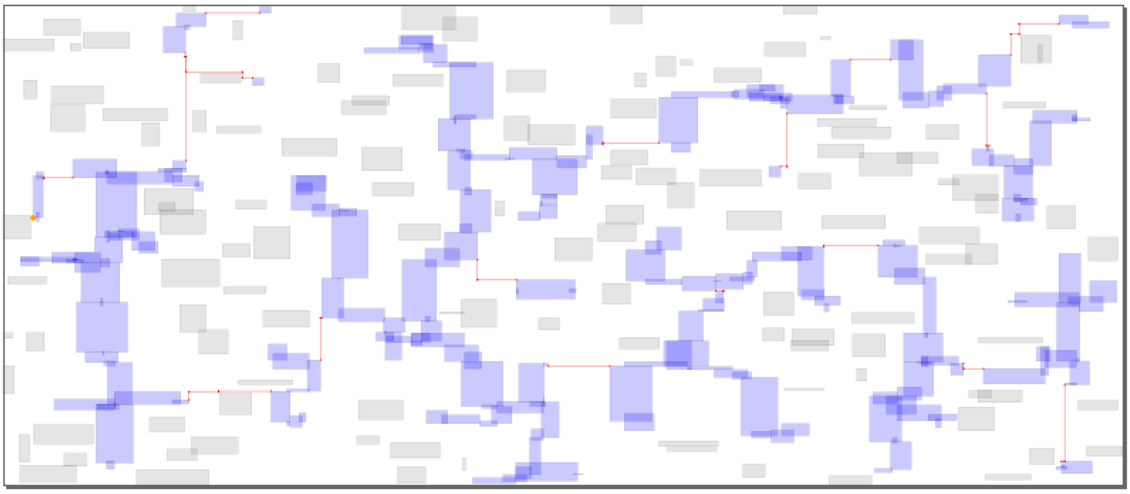

Figure 4: selected result after routing

1. For each pin vertex p = (x,y,l), R(p) = (x,y,$l\times$C$_v$), where the pin is the vertexes of routed net shapes.
2. For each obstacle o, R(o) is a rectilinear box. If $2 \le l \le N_l - 1$, R(o) is constructed by projecting o to $z = (l + 1) \times C_v$ and $z = (l - 1) \times C_v$, and then connect the line segments parallel to the z-axis. If the obstacle is on the first or last layer, we arbitrarily make two layers $z = -\infty$ and $z = \infty$, then the above projection method can be extended to the first layer and the last layer.

After transforming it into 3-D, we can start to construct the ML-VG(V,E) graph. The construction process can be described as following(O is the set containing all the obstacles, and P is the set containing all the vertexes):

1. Let U be the union of R(P) and the obstacle corners in R(O). Let (x_1,....,$x_{|U|}$),(y_1,....,$y_{|U|}$), and (z_1,....,$z_{|U|}$) be the x, y and z coordinates of the vertexes. Let V$_I$ be the set of all vertexes obtained by intersecting the edges of obstacles in R(O) with the plane $x = x_1$,....,x =$x_{|U|}$, y = y_1,....,y =$y_{|U|}$, z = z_1,....,z =$z_{|U|}$. Let the union of U and V$_I$ be V$_S$.
2. Let z$_m$ be the median of the z coordinates of vertexes in V$_S$, and P$_{zm}$ be the plane z = z$_m$. For each vertex (x,y,z) in V$_S$, create u = (x,y,z$_m$) if the line segment $\overline{uv}$ does not intersect obstacle.
3. In P$_{zm}$ plane, we need to construct essential points and edges. The essential points can be created by the following description:
   (a) Let x$_m$ be the median of the x coordinates of the obstacle line segment endpoints. For each endpoint p, include in V$_S$ the point p' = (x$_m$, p$_y$) if p is visible from p' ($\overline{pp'}$ does not intersect the obstacle).
   (b) Apply the first step recursively for the points on the left of x$_m$ and right of x$_m$. Also apply the same method for the y axis.
4. Recursively do steps 2 and 3 for the upper vertexes and lower vertexes for P$_{zm}$ until there is only one plane remain.
5. construct the edge for uv if $\overline{uv}$ is rectilinear and does not intersect any other vertex or obstacle.

5.3 Steiner Point Selection

In this part, we need to select the needed steiner points. We could start with connected component(the method of finding this can be referred to Section 3.1). For this part, maybe we can directly find the code of Dijkstra algorithms online.

5.3.1 The Selection Process

After the algorithm described in section 5.2, we now have an ML-VG G(V,E). To minimize the cost of the solution, Steiner points construction is necessary. In this project, we use the method similar to the method described in two of the previous research papers[2] [3] to construct the Steiner points according the the ML-VG graph obtained. Before going into the algorithm, here we define SPR(p,q), where p and q are vertexes in the graph, to be the minimal region including all the obstacle-avoiding paths between p and q. The steps of selecting Steiner points is described as follows.

1. First we need to group the overlapped routed net shapes' pins $P_i \subset V$ into connected components C_i .In fact, the connected components are simply connected rectilinear polygons.
2. Find the nearest pairs of connected components C_i and C_j. Assume that the nearest pair pins $u \in C_i$ and $v \in C_j$(vertexes on the boundary of the rectilinear polygons).
3. Construct SPR(u,v). Union C_i, SPR(u,v), and C_j into a new connected component. If u or $v \notin P$, select u or v as Steiner point.
4. Repeat 2. and 3. until all the components are connected to each other.

5.4 MTST Construction

In this part, we need to construct a minimum spanning tree for the Steiner points and the vertices which represent original connected components. We should construct a graph that with the original connected components and Steiner points as the vertices, and find the edges connecting each connected component by choosing the shortest edges from the edge connection in ML-VG construction. Then we find a minimum spanning tree for it.

5.4.1 The Construction Process

Let G = (V, E, d) be the connected, undirected distance graph, and $S \subset V$ the set of vertices which includes vertices representing original connected components and Steiner points for which a Steiner tree is desired. E is the set of edges choosing from the shortest edges which connect vertices of original connected components in ML-VG construction. And d is a distance function which maps E into the set of nonnegative numbers.

1. $G_1 = (V_1, E_1, d_1)$ be the complete distance graph where V1 = S and, for every $(v_i, v_j) \in E_1$, $d_1(v_i, v_j)$ is equal to the distance of a shortest path from v_i to v_j in G.
2. Find a minimum spanning tree G_2 of G_1.
3. Construct a subgraph G_3 of G by replacing each edge in G2 by its corresponding shortest path in G. (If there are several shortest paths, pick an arbitrary one.)
4. Find a minimum spanning tree G_4 of G_3.

5.4.2 Time Complexity

In step 1, we can compute the partition N(s) by adjoining an auxiliary vertex s_0 and edges (s_0, s), $s \in S$, of length 0 to G and then perform a single source shortest path computation with source s0. This takes O(|V| log|v| + |E|) time and yields for every vertex v the vertex $s(v) \in S$ with $v \in N(s(v))$ and the distance $d_1(v, s(v))$.
In the remaining steps, it takes O(|S| log|S| + |E|) time in each step because the graph G_1' has only O(|E|) edges.

Finally, we go through all the edges (u, v) in E and generate the triples (s(u), s(v), d1(s(u), u) + d(u, v) + d1(v, s(v))). We sort the triples by bucket sort and then select for each edge of G1' the minimum cost. All of this takes O(|E|) time.

Overall, this minimum spanning tree construction can be computed in time O(|V| log|v| + |E|).

6 Contribution

Chun-ju Wu. Construct the edge part of the report(also the code).
Meng-Jin Lin, Find the Overlapped Routed Net Shapes and Obstacles(also the code)
Hong-Chi Chen, Minimum spanning tree and the output(also the code).

References

[1] K. L. Clarkson, S. Kapoor, and P. M. Vaidya, *Rectlinear shortest paths through polygonal obstacles in O(n log2 n) time*, in Proc. 3rd Annu. ACM Symp. Comput. Geom. SCG, New York, NY, USA, 1987, pp. 251–257.

[2] C.-H. Liu et al. *Efficient Multilayer Obstacle-Avoiding Rectilinear Steiner Tree Construction Based on Geometric Reduction.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 33.12 (2014): 1928-1941.

[3] C.-H. Liu et al. *Obstacle-avoiding rectilinear Steiner tree construction: A Steiner-point-based algorithm.* IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 31, no. 7, pp. 1050–1060, Jul. 2012.