**Algorithm Design and Analysis**

Student ID: B10907005

Name: 楊宜祥

**Algorithm**: Dijkstra’s Algorithm

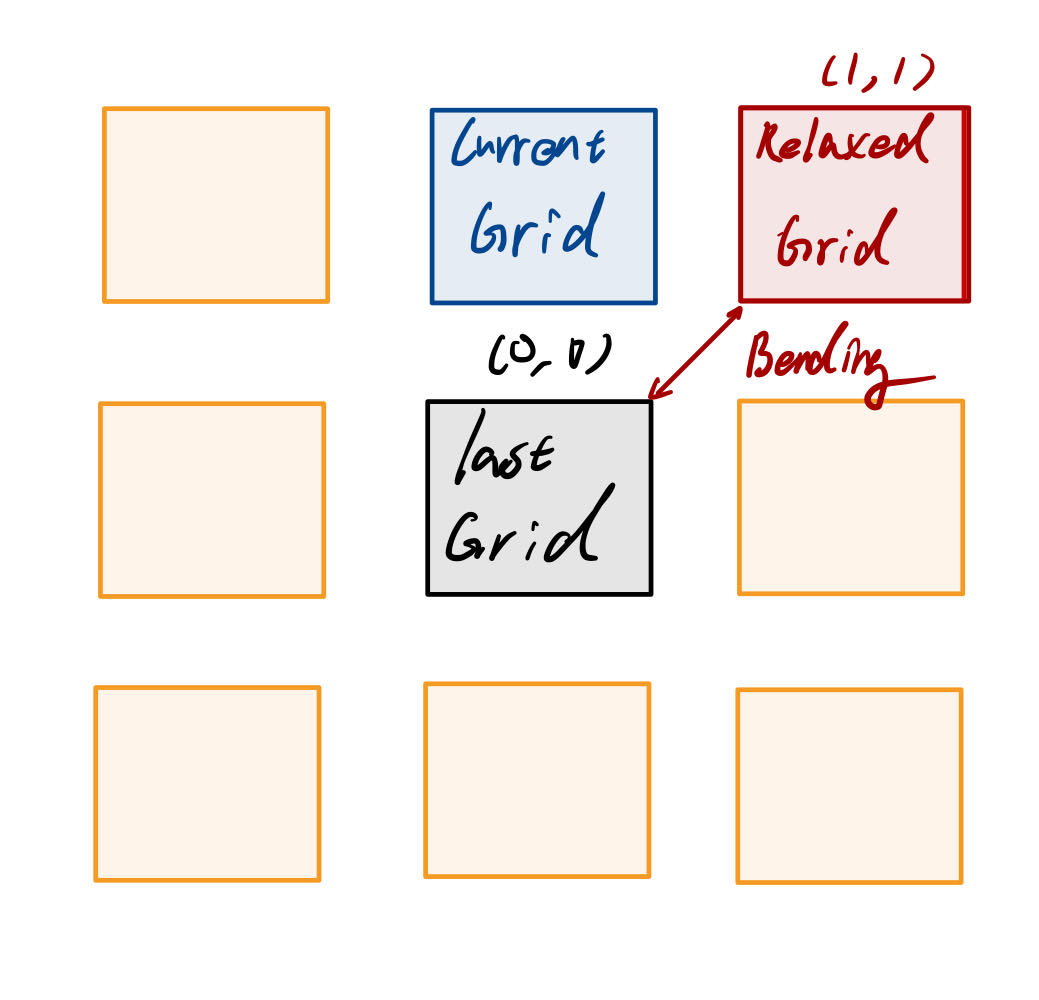
**Introduction:**  
I use Dijkstra’s Algorithm as basic algorithm to deal with this project. **Apply the algorithm to every net to find the path from the started point to the end point with minimal weight.** Most of the algorithm doesn’t change but I slightly modified the *Relax* function and *weight* to adapt the situation.

In the original Dijkstra’s Algorithm, the weight of all edges has been determined. In this problem, the weight will dynamically change if the current path is going to bend or to cross other connected lines. The basic weight is equal to propagation loss. If any of these situations occurs, its corresponding loss will be added to the weight.

**Check Bending**

Every grid of the graph has properties such as *row, col, weight, pre.* I use pre to record the last connected grid with the smallest weight. Following the rule shown below can help it check bending:

*If both of the differences of row and col between the relaxed grid and the last grid of current grid equal to one, consider relaxed grid is bending.*



**Check crossing**

After each time find out the minimal path of the net, the grids on the minimal path are recorded into 2D vector, *discover.* By means of inspecting *discover*, we can know whether the position of the relaxed grid has been dominated or not. Then we can make sure whether it is crossing.

**Potential Problem**

Under my solution, *discover* vector is the key character to determine the *weight* in each *Relax function.* Because *discover* vector is affected by the nets which have been discovered, **the sequence of discovering nets affects the final pattern of connecting nets**.

I have tried to use different sequences to discover nets including:   
1. Discover the net **in the order of reading each net from the input file**.  
2. The net with **smaller Euclidean distance** between two endpoints will be execute first.  
3. The net with **bigger Euclidean distance** between two endpoints will be execute first.

The experiment shows that using method 3 will get the worst pattern in each test .in file. The results from second and first method are almost the same.

Compare method 2 with method 1 in three test .in file:

1. pic5x5.out

|  |  |
| --- | --- |
| Method 1 | Method 2 |
|  |  |

2. pic20x20.out

|  |  |
| --- | --- |
| Method 1 | Method 2 |
|  |  |

3. pic60x60.out

|  |  |
| --- | --- |
| Method 1 | Method 2 |
|  |  |

The results are shown in the above table. The method 2 result from larger graph is slightly better than method1. So I choose method 2 to determine the process order. Make sure that the sequence will not fall into the worst case.

**Data Structure**

**struct Grid**

I model each grid as the customized struct, *Grid,* including variables such as int *row, int col, int weight, Grid \*pre.*

**vector<vector< Grid> > Graph**

Then use 2D vector *Graph* to contain each grid. The number of row and column are determined by the .in file.

**list<Grid \*> gridList**

In Dijkstra’s algorithm, it use a queue to contain all grids but it is time consuming when finding the grid with minimum weight each time. I decide to use a *list* to contain all grids. It can quickly insert and extract an element from the list. I also can use *list* member function, .*sort(),* to sort all elements from the one with smallest weight to the one with largest weight. Easily extract the grid with minimum weight from the front of the *list.*

**vector<vector<int>> discover**

This vector shows the current connected pattern of the graph. When *Relax* is executing, it can inspect this 2D vector to check whether crossing is occurred or not .

**vector<vector<Grid>> Path**

All minimal paths are save in this 2D vector. The path of i th net is saved in Path[i].