EE3980 Algorithm

HW8. Minimum-Cost Spanning Tree

106061225 楊宇羲

1. Problem Description:

In this assignment, we are seeking for a method to find the minimum-cost spanning tree. The definition of minimum-cost spanning tree is an undirected graph that reaches every vertex in the graph without any cycle, also meet the requirement that the total weight of the tree is the minimum possibility. Since it is a tree that reaches every vertex, the total edges will be the number of vertex minus 1. There are two algorithms introduced in this course, which is the Prim's algorithm and Kruskal's algorithm. The general construction of spanning tree is to select a safe edge each time, for a total of |V| - 1 times, and by selecting the safe edge it means to select an edge that doesn't cause a cycle in the graph. Hence, the key to constructing minimum-cost spanning tree lies in the way of selecting edge. All these algorithms all construct optimal solution at each stage by applying greedy method, however, the order of picking edges and store it in the spanning tree may vary. In this report, I will implement the Prim's algorithm and analyze its time & space complexity.

2. Approach:

There are some global variables and structures required to be defined before we carry out the algorithm. The *node* structure is used to store the adjacent list of the original graph and store the *near* array, which is used to store the status of each vertex. It is consist of *index* and *weight* and the pointer that points to the next adjacent node *next.* There is an additional structure to define, which is used to store the final result of spanning tree *RSLT.* It has two integers *v1, v2* and one double *weight* in the structure to record the edge and the weight.

```
Typedef struct node {
    int index;
    double weight;
    node *next;
} node;
```

```
Typedef struct RSLT {
    int i1, i2;
    double weight;
    node *next;
} RSLT;
```

Status of *near*:

-2: the index is already in the spanning tree
-1: the index is not yet reachable to the spanning tree
> -1: the index *j* can reach the spanning tree with the minimum cost by going through *near*[*j*]

There are global variables that store the number of vertices 'V', edges 'E', the calculated minimum cost 'mincost', the adjacent list 'adlist' and the result 'T'. Since there are no need for repetition, the main function pseudo code will be like this:

```
Algorithm main {
    ReadData();
    t = GetTime();
    Prim();
    t = GetTime() - t;
    PrintData();
}
```

In the ReadData() function, we store the edges in the adjacent list, which is introduced in last assignment. It is a linked list array that uses pointer to store every vertex that is connected. If there is a fully connected undirected graph with 4 vertices, then the adjacent list will look like this:

```
Adlist[0] -> 1 -> 2 -> 3
Adlist[1] -> 0 -> 2 -> 3
Adlist[2] -> 0 -> 1 -> 3
Adlist[3] -> 0 -> 1 -> 2
```

After storing the adjacent list, since the Prim's algorithm starts with the edge with the minimum cost, so we record the minimum cost edge and make the two vertices as T[0].i1 and T[0].i2.

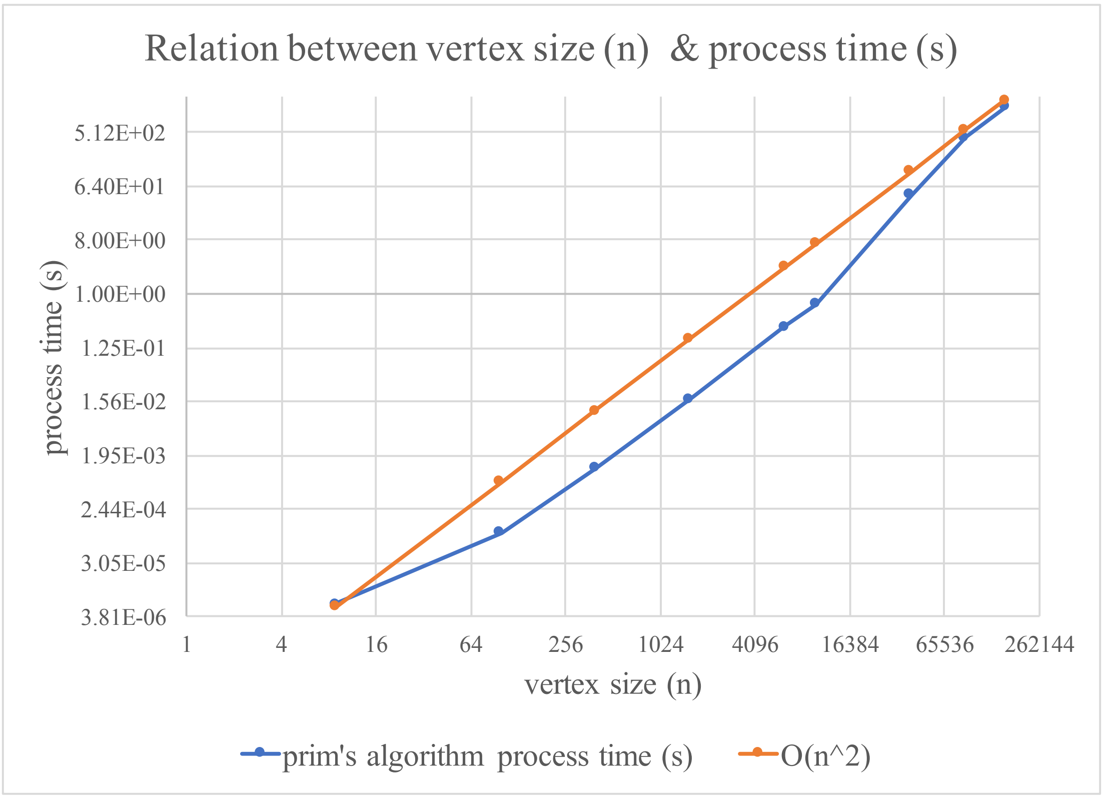
The Prim's algorithm's pseudo code is provided as below:

```
// input: vertex size 'V', weight function of any two vertex 'w'
// output: accumulated minimum cost of spanning tree 'mincost' , spanning tree 'T'
1  Algorithm Prim {
2      for I := 1 to V do {                    // initialize near array
3          near[I].index := -1;
4          near[I].weight := 0.0;
5          near[I].next = NULL;
6      }
7      for I := 1 to V do {      // find each reachable vertices' nearest neighbor with smaller weight
8          if (w[I, T[0].i1] < w[I, T[0].i2]) then near[I] = T[0].i1;
9          else near[I] = T[0].i2;
10     }
11     near[T[0].i1] := -2;                    // -2 means the vertex is in the spanning tree
12     near[T[0].i2] := -2;
13     for I := 2 to V - 1 do {                // look for the next nearest vertex for V - 2 times
14         find a vertex such that near[j] ≥ 0 and w[j, near[j]] is minimum;
15         T[I].i1 = j;                        // store the edge in the T array
16         T[I].i2 = near[j];
17         mincost = mincost + near[j].weight;  // update mincost
18         near[j] = -2;                       // mark the vertex as -2 means it is in the spanning tree
19         for k := 1 to V do {                // update remaining vertices' nearest vertex
20             if ((near[k] ≥ 0) and (w[k, near[k]] > w[k, j])) then near[k] = j;
21         }
22     }
23     return mincost;                         // return result
24 }
```

3. Analysis:

In the lecture, there are one more step described in the pseudo code, which is to find the minimum cost edge then store it in the spanning tree array. However, we done this part while executing the ReadData() function to avoid an extra O(n) execution since we must seek through the whole list to find the minimum. Line 7 to 9 demonstrates an O(n) execution which is aim to initialize the near array. Then from line 13 to 22, we will have to search the whole near array with size |V|, for a total of |V| - 2 times, which is the dominating part of the algorithm. In general, the time complexity will be $O(|V|^2)$ (despite it can be improved if we store the non-visited vertices in red-black tree), and the space complexity is $O(|E|+|V|)$ since there are near array and spanning tree array that has size |V| and the adjacent list that has size |E|.

4. Results:



Dataset name	vertex size (n)	edge size (n)	process time (s)	total weight
g3.dat	9	20	6.20E-06	0.56
g4.dat	100	342	9.82E-05	150.22
g5.dat	400	1482	0.0012	2781.42
g6.dat	1600	6162	0.0171812	47766.82
g7.dat	6400	25122	0.277497	791389.62
g8.dat	10000	39402	0.67	1945547.02
g9.dat	40000	158802	46.6348	31562194.02
g10.dat	90000	358202	426.71	160519941.02
g11.dat	160000	637602	1383.51	508488788.02

5. Observation and conclusion:

Overall, although the last three .dat file takes more time than speculated, the run time growing tendency is still n^2, which is represented in the graph.