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:

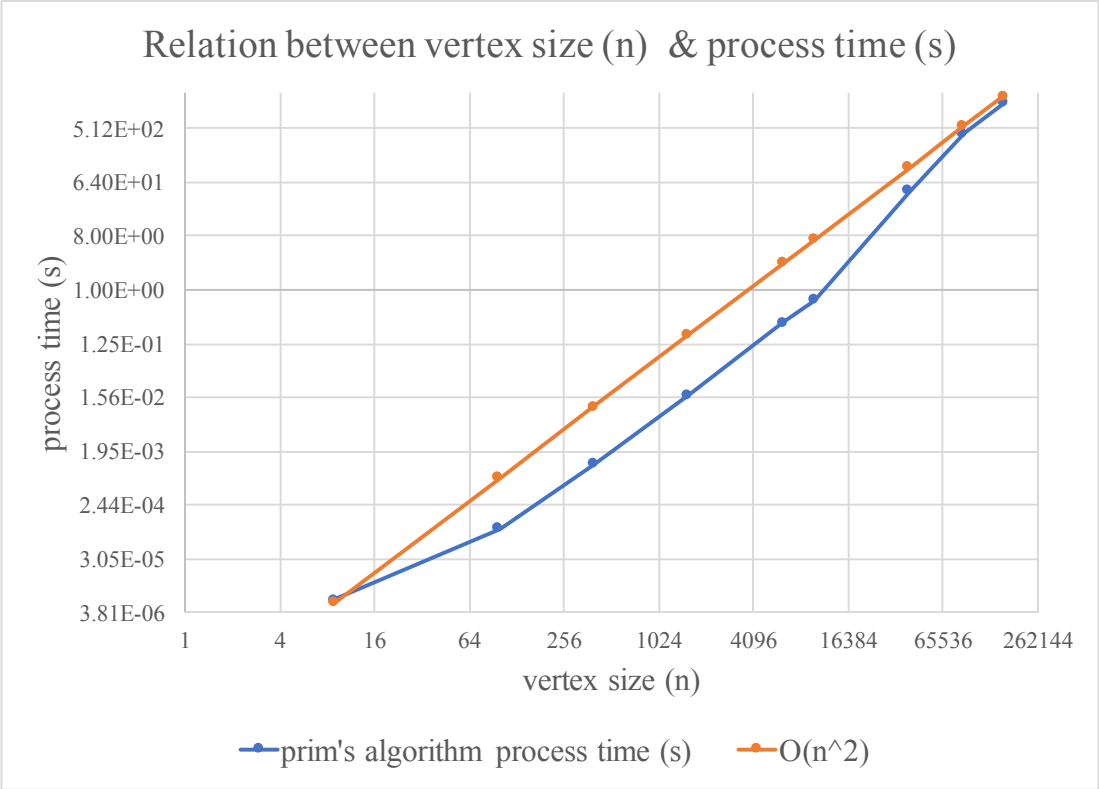
```
// input: vertex size 'V', weight function of any two vertex 'w'
// output: accumulated minimum cost of spanning tree 'mincost' , spanning tree 'T'
1  Algorithm Prim {
2       for I := 1 to V do {                    // initialize near array
3               near[I].index := -1;
4               near[I].weight := 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.

```
$ gcc hw08.c
$ a.out < g8.dat
Minimum-cost spanning tree:
  1: <1 2> 0.01
  2: <1 101> 0.02
  3: <1 102> 0.03
  4: <2 3> 0.04
  ...
  ...
  9995: <9895 9996> 392.85
  9996: <9896 9997> 392.89
  9997: <9897 9998> 392.93
  9998: <9898 9999> 392.97
  9999: <9899 10000> 393.01
|V| = 10000 |E| = 39402
Minimum cost: 1945547.02
CPU time: 0.161809 seconds
```

Good, solution is correct.
Can be more efficient.

---

score: 70.0

- Overall report writing
  - English writing needs more practice.
- Introduction
  - Introduction can be strengthened
- Approach
  - Can argue that the algorithm correctly solve the problem.
  - Can use Kruskal's algorithm for better efficiency.
- Time/Space complexity
  - Both space and time complexity analyses should be more complete.
- Program format can be improved

hw08.c

```c
// EE3980 HW08 Grouping Friends
// 106061225, 楊宇羲
// 2021/5/3

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

typedef struct node {                   // store adjacent list
    int index;                          // store adjacent vertex
    double weight;                      // store edge weight
    struct node *next;                  // points to next vertex
} node;

typedef struct RSLT {                   // store spanning tree
    int i1;                             // two indexes for an edge
    int i2;                             // two indexes for an edge
    double weight;                      // weight of the edge
} RSLT;

int V;                                  // number of vertex
int E;                                  // number of edge
double mincost;                         // total weight
RSLT *T;                                // store spanning tree
node **adlist;                          // store adjacent list

double GetTime(void);                   // get local time in seconds
void prim(void);                        // prim's algorithm
void ReadData(void);                    // store data
void PrintData(double t);               // print result

int main(void)
{
    double t;                           // for time

    ReadData();
    t = GetTime();                      // Get CPU time
    prim();                             // execute algorithm
    t = GetTime() - t;                  // Calculate CPU runtime
```

```
40      PrintData(t);                                   // print result
41
42      return 0;
43  }
44
45  void prim(void)                                     // perform prim's algorithm
46  {
47      int i, j, lminp = V;                            // loop, local min position
48      int i1, i2;                                     // store in spanning tree
49      double lmin;                                    // determine local minimum
50      node near[V];                                   // vertex status
51      node *curr;                                     // travel through list
52
53      for (i = 0; i < V; i++) {                       // near array initialization
54          near[i].index = -1;
55          near[i].weight = 0.0;
56          near[i].next = NULL;
57      }
58
59      /* first time assign each vertices' near array value */
60      for (curr = adlist[T[0].i1]->next; curr != NULL; curr = curr->next) {
61          near[curr->index].index = T[0].i1;
62          near[curr->index].weight = curr->weight;
63      }
64      for (curr = adlist[T[0].i2]->next; curr != NULL; curr = curr->next) {
65          if (near[curr->index].weight != 0.0 &&
66              curr->weight < near[curr->index].weight) {
67              near[curr->index].index = T[0].i2;
68              near[curr->index].weight = curr->weight;
69          }
70          else if (near[curr->index].weight == 0) {
71              near[curr->index].index = T[0].i2;
72              near[curr->index].weight = curr->weight;
73          }
74      }
75
76      near[T[0].i1].index = -2;           // vertex that is in the spanning tree
77      near[T[0].i2].index = -2;
78
79      for (i = 1; i < V - 1; i++) {       // find the rest spanning tree edges
80          lmin = 0.0;                     // initialize local minimum
```

```
81
82          /* find the vertex with the smallest weight edge */
83          for (j = 0; j < V; j++) {
84              if (near[j].index >= 0) {
85                  if (near[j].weight < lmin || lmin == 0) {
86                      lminp = j;
87                      lmin = near[j].weight;
88                  }
89              }
90          }
91          if (lminp > near[lminp].index) {// compare edge endpoints indexes
            if (lminp > near[lminp].index) { // compare edge endpoints indexes
92              i1 = near[lminp].index;
93              i2 = lminp;
94          } else {
95              i1 = lminp;
96              i2 = near[lminp].index;
97          }
98
99          T[i].i1 = i1;                   // store the edge to spanning tree
100         T[i].i2 = i2;
101         T[i].weight = lmin;             // edge weight
102         mincost = mincost + lmin;       // accumulated weight
103         near[lminp].index = -2;         // vertex is in spanning tree
104
105         /* update the new near array */
106         for (curr = adlist[lminp]; curr != NULL; curr = curr->next) {
107             if (near[curr->index].index != -2) {
108                 if (near[curr->index].weight != 0.0 &&
109                     curr->weight < near[curr->index].weight) {
110                     near[curr->index].index = lminp;
111                     near[curr->index].weight = curr->weight;
112                 }
113                 else if (near[curr->index].weight == 0) {
114                     near[curr->index].index = lminp;
115                     near[curr->index].weight = curr->weight;
116                 }
117             }
118         }
119     }
120 }
```

```
121
122 void ReadData(void)                          // build adjacent list
123 {
124     int i, v1, v2;                           // loop, store vertex
125     double weight;                           // store weight
126
127     scanf("%d %d", &V, &E);
128
129     adlist = (node**)malloc(V * sizeof(node*)); // dynamic allocation
130     T = (RSLT*)malloc((V - 1) * sizeof(RSLT));
131     mincost = 0.0;                           // total weight initialization
132
133     for (i = 0; i < V; i++) {                // initialize adjacent list
134         adlist[i] = (node*)malloc(sizeof(node));
135         adlist[i]->index = i;
136         adlist[i]->weight = 0.0;
137         adlist[i]->next = NULL;
138     }
139
140     for (i = 0; i < E; i++) {                // start store edges
141
142         node *build1;
143         node *build2;
144         node *curr;
145
146         scanf("%d %d %lf",&v1, &v2, &weight);
            scanf("%d %d %lf", &v1, &v2, &weight);
147
148         if (mincost == 0 || mincost > weight) { // find smallest weight
149             mincost = weight;
150             T[0].i1 = v1 - 1;
151             T[0].i2 = v2 - 1;
152             T[0].weight = weight;
153         }
154
155         build1 = (node*)malloc(sizeof(node));   // store edge to one endpoint
156         build1->index = v2 - 1;
157         build1->weight = weight;
158         build1->next = NULL;
159
160         curr = adlist[v1 - 1];
```

```
161         build1->next = curr->next;
162         curr->next = build1;
163 
164         build2 = (node*)malloc(sizeof(node));   // store to another endpoint
165         build2->index = v1 - 1;
166         build2->weight = weight;
167         build2->next = NULL;
168 
169         curr = adlist[v2 - 1];
170         build2->next = curr->next;
171         curr->next = build2;
172 
173     }
174 }
175 
176 void PrintData(double t)                        // print result
177 {
178     int i;
179 
180     printf("Minimum-cost spanning tree:\n");
181     for (i = 0; i < V - 1; i++) {
182         printf("  %d: <%d %d> %g\n", i + 1,
183                                      T[i].i1 + 1,
184                                      T[i].i2 + 1,
185                                      T[i].weight);
186     }
187     printf("|V| = %d |E| = %d\n", V, E);
188     printf("Minimum cost: %.2f\n", mincost);
189     printf("CPU time: %g seconds\n", t);
190 }
191 
192 double GetTime(void)                            // get local time
193 {
194     struct timeval tv;
195 
196     gettimeofday(&tv, NULL);
197     return tv.tv_sec + 1e-6 * tv.tv_usec;       // sec + micro sec
198 }
199 
200 
```