Bonus-2 Battle Over Cities - Hard Version

王睿

3180103650

Contents

1. Introduction

1.1 Description

With the information of all the cities and the highways connected to them (destroyed or remaining), we're supposed to find the city that we'll take the largest cost to restore the connectivity of the graph if that city is conquered.

Actually, we are asked to find the largest **minimum spanning tree** for that graph with each time one vertex deleted.

2.1 Algorithm Specification

In this algorithm, I used the **Kruska's Algorithm** to find the minimum spanning tree for each condition. Then general implementation is as follows.

After finishing reading in the map, I firstly used `qsort` to sort the edges according the decreasing status. That is, after this sort, those highways (edges) which are in use will be put prior to those destroyed. The reason for this sort is that we will add the existing edges to the *minimum spanning tree* first (since we don't need to spend any cost to repair it). Then, I seperately sort the edges in use and edges that are destroyed according to the ascending cost. So after these three `qsort`, we just need to add edges to the minimum spanning tree according to the ascending index sequence.

The `MST` function is where the *Kruska's Algorihtm* takes place. Each time I omit one vertex and all the edges incident to it, and finding the minimum spanning tree by add the edges according to the previously sorted edge sequence. Note that if the total number of edges in the final minimum spanning tree is less than $N-2$ (N is the number of veritces in the graph), then the minimum spanning tree is not connected (The minimum number of edges needed to connect $N-1$ vertices is $N-2$). So that deleted vertex must be the answer (one of the answer).

Pseudocode is as follows. Note that N is the number of vertices while M is the number of edges, $result$[] is the array to store the minimum spanning trees and S[] is the array of the minimum spanning tree union. For example, $result[i]$ is the minimum spanning tree after deleting the vertex i and the edges incident to it, and $S[i]$ is the previous vertex in the union of vertex i.

```
read in the Graph

sort all the edges by putting the destroyed edgeslast
sort the used edges in ascending cost
sort the destroyed edges in ascending cost

for each i from 1 to N:
    result[i] <- 0
    count <- 0
    for each j from 1 to N:
        S[j] <- (-1)
    for each j from 0 to M-1:
        if one of the end points of edge[j] is i:
            continue
        root_source <- the root of the end_point_1 in the union
        root_dest <- the root of the end_point_2 in the union
        if root_source != root_dest:        // no cycle produced
            S[root_source] <- root_dest
```

```
                count++
                if edge.status == 0:        // add to the MST if it's destroyed
                    result[i] <- result[i] + edge[j].cost
        if count < N-2:
            result[i] <- INF
        ans <- MAX(ans, result[i])   // update the ans

    for each element in result[]:
        if element == ans
            output index
```

3. Testing Results

- **Test point 1: MAX N, connected**

> **Input :**
>
> See the file "MAX_N_connected.txt"

> **Output :**
>
> 0

- **Test point 2: MAX N, output all**

> **Input :**
>
> See the file "MAX_N_outputAll.txt"

> **Output :**
>
> 2 3 4 ... 499

4. Analysis and Comments

4.1 Time Complexity

The average time complexity for this algorithm is $O(ElogV)$, because of the Kruskal's Algorithm with union find operation. Other functions like `initGraph` and `printResult` are all $O(N)$.

4.2 Space Complexity

The space complexity for this algorithm is $O(E+2V)$, since in the function `initGraph`, I used the a structure that has an array of size E, in the function `MST`, there're two utility arrays of size E and size `V` respectively. So the total space complexity is $O(E+2V)$.

5. Appendix

> Source Code in C

```c
#include<stdio.h>
#include<stdlib.h>

#define MAX (501)
#define FINDMAX(x,y) ((x)>(y)?(x):(y))
#define ROOT (-1)                // to indicate the root if a union
#define INF (0x7FFFFFFF)         // to mark that if remove this vertex, the graph cannot be connected

/***********************************************
       Data Structure Used in This Code

* GNode: Used to store the information the graph.
        That is the all the cities(vertices) and
        the highways(edges) connected to them.

* Edge: Used to store the information of one edge
        (highway), including the end points, cost
        and the status

***********************************************/

typedef struct GNode* Graph;
typedef struct Edge* PtrToEdge;

struct GNode {
    int NumOfVertices;
    int NumOfEdges;
    PtrToEdge edge;
};

struct Edge {
    int source;
    int dest;
    int cost;
    int status;
};

/* S[i] represents the previous ponint in the union of the minimum spanning tree */
/* result[i] represents the cost we need to take if city[i] is being conquered*/
int S[MAX], result[MAX];

/**********************************************************
                Functions

* InitGraph: Read in all the information of the graph

* myCompare_cost: Compare function used in qsort.Determine
        the sort according to the ascending cost of rebuild
        -ing the highway.

* myCompare_status: Compare function used in qsort.Determine
        the sort according to the status in decreasing order.
        That is, put the highways that are in use prior to
        those destroyed.

```

```
* MST:  calculate the minimum spanning tree after deleting
        the vertex and the edges incident with it using
        Kruska's Algorithm.

* find: Union find operation. Used path compression.

* printResult: Print the cities that we must pay most attention
        to in increasing order.
**********************************************************/

Graph initGraph(int* destroy_cnt);
int myCompare_cost(const void* a, const void* b);
int myCompare_status(const void* a, const void* b);
int MST(Graph G);
int find(int vertex);
void printResult(Graph G, int* result, int ans);

int main()
{
    int i, ans, detroy_cnt = 0;      // ans is the final answer, destroy_cnt is the number of destroyed highways

    Graph G = initGraph(&detroy_cnt); // read in the graph

    qsort(G->edge, G->NumOfEdges, sizeof(G->edge[0]), myCompare_status); // sort the edges according to the status in decreasing order
    qsort(G->edge, G->NumOfEdges - detroy_cnt, sizeof(G->edge[0]), myCompare_cost); // sort the edges in use according to the ascending cost order
    qsort(G->edge + (G->NumOfEdges - detroy_cnt), detroy_cnt, sizeof(G->edge[0]), myCompare_cost);  // sort the destroyed edges according to ascending cost order

/*calculate the minimum spanning tree after deleting each vertex and the edges incident with it using Kruska's Algorithm. Store the result*/
    ans = MST(G);

    printResult(G, result, ans);    // output the answer

    return 0;
}

Graph initGraph(int* destroy_cnt)
{
    Graph G = (Graph)malloc(sizeof(struct GNode));
    scanf("%d %d", &G->NumOfVertices, &G->NumOfEdges);  // read in the number of cities and highways
    G->edge = (PtrToEdge)malloc(sizeof(struct Edge) * G->NumOfEdges);

    for (int i = 0; i < G->NumOfEdges; i++) {
        scanf("%d %d %d %d", &(G->edge[i]).source, &(G->edge[i]).dest, &(G->edge[i]).cost, &(G->edge[i]).status);   //read in the information for each highway
        if (!(G->edge[i]).status)  // count the number of destroyed highways
            (*destroy_cnt)++;
    }
    return G;
```

```c
}

int MST(Graph G)
{
    int i, j, ans, root_source, root_dest, count;

/***************************************************************************
                        Variables Used

    * i, j: Temporary counter.

    * ans: The maximum effort we need to take to restore the connectivity of
            the graph. (final answer)

    * root_source: The root of the source vertex of a highway in the MST union.

    * root_dest: The root of the destination vertex of a highway in the MST union.

    * count: A counter to count the number of edges in the final minimum spanning
            tree. If it's less than N-2 then it imply that the minimum spanning
            tree is not conneced. Then that vertex must be the answer.

***************************************************************************/
    ans = 0;

    for (i = 1; i <= G->NumOfVertices; i++) {   // loop for each vertex, calculate the MST without that vertex
        result[i] = 0;      // initialize the result[] array and the counter
        count = 0;
        for (j = 1; j <= G->NumOfVertices; j++)     // initialize the union array
            S[j] = ROOT;
        for (j = 0; j < G->NumOfEdges; j++) {
            if ((G->edge[j]).source == i || (G->edge[j]).dest == i)     // if the highway is from/toward that deleting city, continue
                continue;
            root_source = find((G->edge[j]).source);    // find the root of the end points
            root_dest = find((G->edge[j]).dest);
            // if the root of the end points is identical ,then adding this edge will produce a cycle
            if (root_source != root_dest) {
                S[root_source] = root_dest;     // connect the edge into the union
                count++;
                if (!(G->edge[j]).status)               // only the destroyed highway needs to be repaired
                    result[i] += (G->edge[j]).cost;
            }
        }
```

```
        if (count < G->NumOfVertices - 2)       // if the edge in the final MST is less than N-2, then the MST is not connected
            result[i] = INF;
        ans = FINDMAX(ans, result[i]);  // update the ans
    }
    return ans;
}

int find(int vertex)        // union find operation with path compression
{
    if (S[vertex] == ROOT)
        return vertex;
    else
        return S[vertex] = find(S[vertex]);
}


void printResult(Graph G, int* result, int ans)     // print the results
{
    int i, flag;

    if (!ans)
        printf("0\n");
    else {
        for (i = 1, flag = 0; i <= G->NumOfVertices; i++)
            if (result[i] == ans) {
                if (flag)
                    printf(" %d", i);
                else {
                    printf("%d", i);
                    flag = 1;
                }
            }
        printf("\n");
    }
}

int myCompare_cost(const void* a, const void* b) // sort according to the cost
{
    PtrToEdge a1 = (PtrToEdge)a;
    PtrToEdge b1 = (PtrToEdge)b;
    return a1->cost - b1->cost;
}

int myCompare_status(const void* a, const void* b) // sort according to the status
{
    PtrToEdge a1 = (PtrToEdge)a;
    PtrToEdge b1 = (PtrToEdge)b;
    return b1->status - a1->status;
}
```