Fundamentals of Data Structures

Project 3 : Transportation Hub

Date : 2024/04/25

2023–2024 Spring & Summer Semester

Table of Contents

Chapter 1 : Introduction

Problem

Description

For a map of a country, there might be more than one shortest path from the starting city to the destination. For a city that appears in the shortest paths for more than k times, we call it a transportation hub. Our task is to find the transportation hubs for each query given a map of a country.

Input

The first line provides the total number of cities n, the number of roads m, and the threshold for a transportation hub k.

Then we are provided m lines in the format $c_1\ c_2\ length$, where c_1 and c_2 are the city indices (from 0 to $n-1$) of the two ends of the road (two-way roads) and $length$ is the length of the road.

Then the next line gives a positive integer T, followed by T lines, each of which gives a pair of source and destination.

Output

The output contains T lines. For each pair of source and destination, list all the transportation hubs on the way in the ascending order in the line.

If there is no transportation hub on the way, print $None$ in the line.

Sample Input

```
10 16 2
1 2 1
1 3 1
1 4 2
2 4 1
2 5 2
3 4 1
3 0 1
4 5 1
4 6 2
5 6 1
7 3 2
7 8 1
7 0 3
8 9 1
9 0 2
```

```
0 6 2
3
1 6
7 0
5 5
```

Sample Output

```
2 3 4 5
None
None
```

Algorithm Analysis

Dijkstra Algorithm

The core algorithm is called Dijkstra Algorithm.

The common Dijkstra Algorithm uses greedy strategy and is comprised of 4 steps:

Firstly declare an array dis to store the shortest distance from the source point to each vertex and a set of vertices named T that have found the shortest path. Initially, the path weight of the origin s is assigned to 0 ($dis[s] = 0$). If there are edges (s, m) that can be directly reached for vertex s, set $dis[m]$ as the weight of the edge, and set the path length of all other vertices (which s cannot directly reach) to infinity.

Secondly, the set T only has vertices s.Then, select the minimum value from the dis array, which is the shortest path from the source point s to the corresponding vertex, and add the point to T. Then a vertex is completed.

Thirdly, we need to see if the newly added vertices can reach other vertices and if the path length to reach other points through that vertex is shorter than directly reaching the source point. If so, then replace the values of these vertices in dis.This operation is called Loose Operation.

Lastly, find the minimum value from dis and repeat the Loose Operation until T contains all the vertices of the graph.

DFS Algorithm

Also in this task we use DFS Algorithm to find all smallest distance path.

To obtain a solution to the problem, DFS Algorithm firstly choose a possible scenario to explore forward.

During the exploration process, once it is discovered that the original choice was incorrect or we found one solution, one step back and choose again to continue exploring forward

Repeat this process until all solutions are obtained.

Chapter 2 : Algorithm Specification

Step 1 : Apply Dijkstra Algorithm

To solve this problem, we only need to make several little changes to the common Dijkstra Algorithm. In the Loose Operation, we also need to consider the situation when the path length to reach other points through that vertex is equal to directly reaching the source point. This means there is probably more than one minimum path. Store the path.

We use **heap** to find the smallest unknown distance vertex thus optimizing the dijkstra algorithm. Also an **array** dis is applied to store the smallest path from starting point to the destination. Pseudo-code is shown below:

```
dis[start]=0;
push the start point into the heap;
while(heap is not empty){
    V=smallest unknown distance vertex;
    pop V from the heap;
    if(V is visited)
        continue;
    Mark V as visited;
    for(each W adjacent to V){
        if(dis[V]+weight of the edge<dis[W]){//Loose Operation
            dis[W]=dis[V]+weight of the edge;
            clear the pre array;//The old path in pre is useless,so clear them and start storing new path again
            store V in the pre array;
            if(W is not visited)
                push W in the heap;
        }
        if(dis[V]+weight of the edge==dis[W]){//If there exists a path of same distance,we also store the previous vertex
            store V in the pre array;
            if(W is not visited)
                push W in the heap;
        }
    }
}
```

Step 2 : Get the path and judge transportation hub

Finally we use DFS Algorithm to count the times that the vertix appears in the path and judge whether it is a transportation hub or not.

We will store all possible previous vertex in the smallest distance path in an **array** pre. Then for every previous vertex $pre[i]$ in the pre, explore further to get the previous vertices of $pre[i]$. Dug deeper until we reach the starting point. This means we have found one smallest distance path. Step back to explore further until we have found all smallest distance path.

Lastly use an **array** $bucket$ to count and judge whether a vertix is a transportation hub or not. Pseudo-code is shown below:

```
void dfs(starting point,current point){
    if(current point==starting point){
        for(every vertix i in temp path)
            bucket[i]++;
    }
    push current point in temp path;
    for(every vertix i in pre[current point])
        dfs(starting point,i);
    pop current point in temp path;//step back to explore further
}
```

Chapter 3 : Testing Results

Test case 1

Test case 1 wants to test the graph of 10 vertices and 38 edges with the threshold being 1 and query times being 5.

Input

```
10 38 1
6 8 3
0 9 1
8 4 3
9 1 4
9 6 1
5 3 5
2 3 5
5 9 1
2 1 5
5 0 1
8 9 5
7 3 1
5 8 5
2 7 4
4 0 2
4 6 4
8 0 3
6 2 5
4 7 4
5 6 4
3 1 1
9 4 4
8 3 2
1 4 2
7 9 2
6 1 5
5 1 1
1 0 2
```

```
3 4 2
0 7 2
0 3 5
2 9 1
7 1 3
7 5 2
0 2 1
4 5 5
3 6 5
8 2 2
5
5 8
3 0
0 5
2 9
3 7
```

Output

```
6 9
1
7
None
0 1
```

Status : Passed

Test case 2

Test case 2 wants to test the graph of 20 vertices and 137 edges with the threshold being 2 and query times being 10.

Input

```
20 137 2
10 19 5
4 11 1
//134 lines being omitted
10 14 3
10
6 1
18 17
10 16
13 3
9 5
8 18
6 4
0 14
19 16
1 3
```

Output

```
4 16
None
None
None
None
None
None
None
3
7 8 12
```

Status : Passed

Test case 3

Test case 3 wants to test the graph of 50 vertices and 1089 edges with the threshold being 2 and query times being 20.

Input

```
50 1089 2
5 9 9
//1087 lines being omitted
29 14 4
20
27 20
36 23
46 27
41 12
30 43
29 37
9 33
21 47
48 13
28 17
38 1
28 9
29 7
47 10
45 18
9 10
11 26
35 18
13 12
30 15
```

Output

```
None
6 22
None
None
None
None
None
48
None
32
None
None
None
None
None
7
6
None
40
7 22
```

Status : Passed

Test case 4

Test case 4 wants to the graph of 200 vertices and 8558 edges with the threshold being 1 and query times being 100.

Input

```
200 8558 1
177 108 22
//8556 lines being omitted
96 22 15
100
180 37
//98 lines being omitted
23 152
```

Output

```
64 104 126
12 25 48 97 164 174 175 193
11 23 24 35 52 95 96 97 189 192 193 196
//96 lines being omitted
48 97
```

Status : Passed

Test case 5

Test case 5 wants to test the graph of 500 vertices and 124750 edges with the threshold being 4 and query times being 500. (Maximum test cases)

Input

```
500 124750 4
200 280 2
//124748 lines being omitted
456 386 94
500
166 406
//498 lines being omitted
426 206
```

Output

```
None
38
//497 lines being omitted
None
```

Status : Passed

Chapter 4 : Analysis and Comments

Time Complexity

For step 1 : Apply Dijkstra Algorithm, the time complexity of find the smallest unknown distance vertex through heap is $O(log\ n)$. Also we still need to traverse all the vertex, so the time complexity of step 1 is $O(nlog\ n)$.

For step 2 : Get the path and judge transportation hub. Firstly we need to traverse all the vertex. Then we need to traverse all the vertex's previous vertex in the smallest distance path. This means the time complexity of step 2 is $O(n * n) = O(n^2)$

To sum up, the total time complexity of the program is $O(nlog\ n + T * n^2) = O(T * n^2)$

Space Complexity

The whole program construct:

- Two arrays *edge* and *head* to store the information of the graphs : $O(n^2+n)=O(n^2)$
- A two-dimensional array *pre* to store the previous vertex of every vertex in the smallest distance path : $O(n^2)$
- A heap to get the smallest unknown distance vertex : $O(n)$
- An array *dis* to store the minimum distance : $O(n)$
- An array *bucket* to count the times every vertex in the minimum paths : $O(n)$
- An array *visit* to mark whether the vertex is in the set T : $O(n)$

To sum up the total space complexity of the program is $O(n^2)$

Comments

This program successfully complete the task, but the DFS algorithm slow down the whole program, making the optimizing part of the Dijkstra algorithm a bit useless. It can be possibly improved.

Appendix : Source Code

Source Code

```
#include<stdio.h>
#include<vector>
#include<queue>
#include<string.h>
using namespace std;

struct node{//We use Chain Forward Star to store the graph information
    int to,next,value;
}edge[125000];

struct node1{//Define a node to store distance to ind
    int ind,dist;
    bool operator < (const node1 &x) const{//Define the < operator of node1
        return x.dist<dist;
    }
};

std::priority_queue<node1> q;//We use heap to optimize dijkstra algorithm
vector<int> pre[501];//pre[i] stores previous vertex in the minimum path
vector<int> temp;//temp stores the temporary path

int head[501];//head[i] stores the index of the head edge starting from i
int dis[501];//dis[i] stores the minimum distance from staring point to i
int bucket[501];//bucket[i] stores the times of vertex i appearing in the minimum
path
```

```cpp
bool visit[501];//visit[i] stores whether we have calculated the minimum distance of the vertex i
int total,n,m,k,c1,c2,len,T,s,t;

void add(int u,int v,int w){//Add edges
    edge[++total].to=v;//The destination is v
    edge[total].value=w;//The weight of the edge is w
    edge[total].next=head[u];//The next edge is the old head edge
    head[u]=total;//The new head edge of u is the current edge
}

void dfs(int s,int t){
    if(s==t){//We have found one minimum path
        temp.push_back(t);
        for(int i=temp.size()-2;i>=1;i--)//For the vertex appearing in the path(s and t not included),plus 1
            bucket[temp[i]]++;
        temp.pop_back();//pop to find next minimum path
        return;
    }
    temp.push_back(t);//push the vertex in the path
    for(int i=0;i<pre[t].size();i++)
        dfs(s,pre[t][i]);//dfs previous vertex
    temp.pop_back();//pop to find next minimum path
}

int main(){
    /*Readin*/
    scanf("%d %d %d",&n,&m,&k);
    for(int i=1;i<=m;i++){
        scanf("%d %d %d",&c1,&c2,&len);
        add(c1,c2,len);//add edges
    }
    /*Operation*/
    scanf("%d",&T);
    for(int i=1;i<=T;i++){
        bool flag=false;//flag stores whether there is a transporation hub or not
        scanf("%d %d",&s,&t);
        /*Clear all the arrays*/
        memset(dis,0x7f,sizeof(dis));
        memset(visit,false,sizeof(visit));
        memset(bucket,0,sizeof(bucket));
        /*Dijkstra Algorithm*/
        dis[s]=0;q.push((node1){s,0});
        while(!q.empty()){
            node1 temp=q.top();//Get the minimum vertex to start loose operation
            q.pop();//Pop the vertex from the heap
            if(visit[temp.ind])//If the minimum distance of the vertex is calculated then continue
                continue;
            visit[temp.ind]=true;
            for(int j=head[temp.ind];j;j=edge[j].next){
                int des=edge[j].to;
                if(dis[des]>dis[temp.ind]+edge[j].value){//Loose operation
                    dis[des]=dis[temp.ind]+edge[j].value;
```

```
                    pre[des].clear();//The old path in pre is useless,so clear them and start storing new path again
                    pre[des].push_back(temp.ind);
                    if(!visit[des])
                        q.push((node1){des,dis[des]});
                }
                else if(dis[des]==dis[temp.ind]+edge[j].value){//If there exists a path of same distance,we also store the previous vertex
                    pre[des].push_back(temp.ind);
                    if(!visit[des])
                        q.push((node1){des,dis[des]});
                }
            }
        }
        /*Find the path and calculate the times of of vertex appearing in the minimum path*/
        dfs(s,t);
        for(int j=0;j<n;j++)
            if(bucket[j]>=k)//A transportation hub
            {
                if(flag==false)
                    printf("%d",j);
                else
                    printf(" %d",j);
                flag=true;
            }
        if(flag==false)
            printf("None\n");
        else
            printf("\n");
    }
    return 0;
}
```

Data Generator

```
#include<bits/stdc++.h>
using namespace std;
const int N=500,T=500,K=5,L=100;
bool visit[N+1][N+1];
int main(void){
    freopen("test9.in","w",stdout);
    srand(time(0));
    int limit=N*(N-1)/2;
    int m=limit;
    int k=rand()%K+1;
    memset(visit,false,sizeof(visit));
    fprintf(stdout,"%d %d %d\n",N,m,k);
    for(int i=1;i<=m;i++){
        int c1=rand()%N,c2=rand()%N;
        while((c1==c2)||(visit[c1][c2])){
            c1=rand()%N;
```

```
            c2=rand()%N;
        }
        visit[c1][c2]=visit[c2][c1]=true;
        int len=rand()%L+1;
        fprintf(stdout,"%d %d %d\n",c1,c2,len);
    }
    memset(visit,false,sizeof(visit));
    fprintf(stdout,"%d\n",T);
    for(int i=1;i<=T;i++){
        int c1=rand()%N,c2=rand()%N;
        while((c1==c2)||(visit[c1][c2])){
            c1=rand()%N;
            c2=rand()%N;
        }
        visit[c1][c2]=visit[c2][c1]=true;
        fprintf(stdout,"%d %d\n",c1,c2);
    }
    fclose(stdout);
    return 0;
}
```

Declaration

I hereby declare that all the work done in this project titled “Transportation Hub” is of my independent effort.