Hard Performance Measurement(MSS)

2023-11-24

Chapter 1: Introduction

Holiday is coming. Lisa wants to go home by train. But this time, Lisa does not want to go home directly -- she has decided to take the second-shortest rather than the shortest path. Your job is to help her find the path.

There are *M* stations and *N* unidirectional railway between these stations. For simplicity Lisa starts at No.1 station and her home is at No.*M* station.

The second-shortest path can **backtrack**, i.e., use the same road more than once. The second-shortest path is the shortest path whose length is longer than the shortest path(s) (i.e. if two or more shortest paths exist, the second-shortest path is the one whose length is longer than those but no longer than any other path).

Each input file contains one test case. For each case, the first line gives two positive integers *M* (1≤*M*≤1000) and *N* (1≤*N*≤5000) which are specified in the above description. Then *N* lines follow, each contains three space-separated integers: *A*, *B*, and *D* that describe a road from *A* to *B* and has length *D* (1≤*D*≤5000)

For each test case, print in one line the length of the second-shortest path between node 1 and node *M*, then followed by the nodes' indices in order. There must be exactly 1 space between the numbers, and no extra space at the beginning or the end of the line.

Chapter 2: Algorithm Specification

1. adjacent matrix

To keep the paths among these vertices, I design an adjacent matrix `keep[10000][10000]` to record them. It is a global variable because it makes call it easily in many functions. It seems complex when every recursion passes it as a parameter. Additionally, if the input is `3 4 100`, `keep[3][4]=100` and `keep[4][3]` doesn't be changed. Because it is an directed graph.

2. dfs algorithm

To solve this problem, I think dfs is a good algorithm. When designing dfs, I consider recursion. I notice that the export of recursion. if `` `n==M ``,which means that we get the end, We should record the path of the road and the length of it and go out of the recursion.

What's more, to avoid infinite circulation of recursion, we should remember whether we have gone this road. I design `ifgo[10000][10000]` to remember. If we go from this road, `ifgo` will be set to 1. And we can only go the road if `ifgo[][]==0`.

Then we should notice another important thing. Different road must use the same road and if we don't reset `ifgo` to 0, we can't reach the same road in different path. So, it is very necessary to set `ifgo[][]=0` after each recursion. The program is below.

```
if (keep[n][i] != 0 && ifgo[n][i] == 0) // path exists and has not been visited
        {
            ifgo[n][i]++;                                // upload ifgo[n][i]
            Recursion(i, length + keep[n][i], num + 1); // continue recursion
            ifgo[n][i]--;                                // release ifgo[n][i]
after
        }
```

This program can find all the way of the map.

3. backtrack

To deal with backtrack, first I deal with each edge can go twice. For example, we can go this road if `ifgo[][]` <=1, but I find it is too complex and it has no necessity. The second shortest road can only be no backtrack or backtrack the shortest path. So I remember the shortest road between arbitrary 2 vertices. And after all the recursion, I compare the second shortest path without backtrack and the shortest path plus twice of the shortest road between arbitrary 2 vertices, which makes sure that the one of the 2 vertices is reached during the road. I think it's correct and easy.

4. remember paths

I consider for a long time because I can't get a good way to remember. Once I open a two-dimension array but it is not ok because this can't remember part of repeating way.

When I was sad, I got a good way. We can add a parameter in recursion to record the number of vertice and keep it in a temporary array. After I get the end. I transform this array into a two-dimension array and the first subscript is the same as the length. By this way I keep the road.

But it still has one problem, because I keep the path by a temporary array, it may effect next path. So before each recursion, I clear extra position of the array to make sure last road can't effect this road.

5. simplification

I think remembering all the road is too complex. So when reaching the end, I only keep the path that the length is smaller than current smallest plus twice than the smallest edge between 2 vertice. For example, the test 2 has more than 15 paths and when using this way, it only keeps 6 roads. It saves many space because remembering the path is complex.

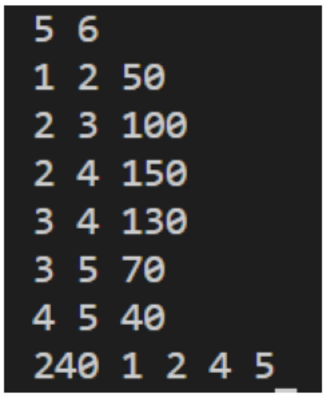
Chapter 3: Testing Results

You can copy these examples to test the program. They are as follows:

1

```
5 6
1 2 50
2 3 100
2 4 150
3 4 130
3 5 70
4 5 40
```

The example is given by PTA. We easily know the minimum path is 220(1-2-3-5) and the second shortest path is 240(1-2-4-5). I test it. The answer is below.

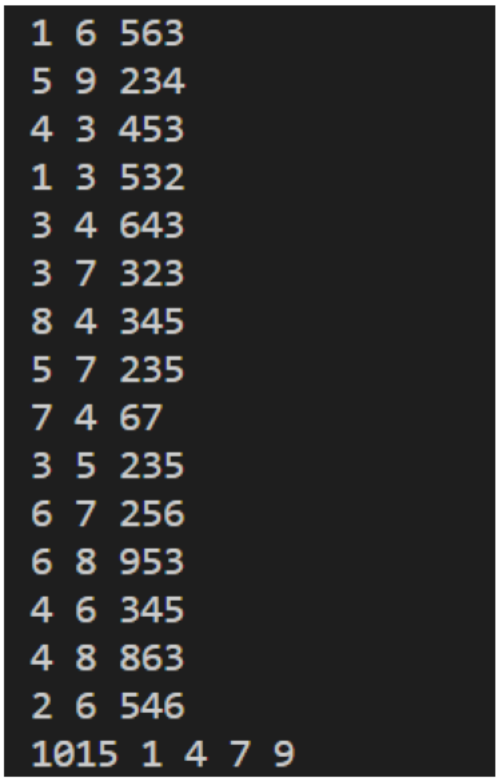


```
5 6
1 2 50
2 3 100
2 4 150
3 4 130
3 5 70
4 5 40
240 1 2 4 5
```

2

```
1 4 546
4 7 145
2 4 654
2 8 322
7 9 324
1 6 563
5 9 234
4 3 453
1 3 532
3 4 643
3 7 323
8 4 345
5 7 235
7 4 67
3 5 235
6 7 256
6 8 953
4 6 345
4 8 863
2 6 546
```

The example is a complex graph. We notice that it has circuit(3-4 4-3). And the minimum path is 1001(1-3-5-9) and the second minimum path is 1015(1-4-7-9). I test it. The answer is below.

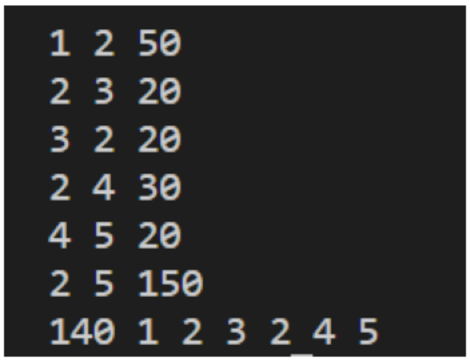


```
1 6 563
5 9 234
4 3 453
1 3 532
3 4 643
3 7 323
8 4 345
5 7 235
7 4 67
3 5 235
6 7 256
6 8 953
4 6 345
4 8 863
2 6 546
1015 1 4 7 9
```

3

```
5 6
1 2 50
2 3 20
3 2 20
2 4 30
4 5 20
2 5 150
```

The example is special. We notice that the path that reaches least vertices(1-2-5) is not the minimum path. And we know the minimum path is 100(1-2-4-5) and the second shortest path is 140(1-2-3-2-4-5). The second shortest path reaches the lab's require that going backtrack. I test it. The answer is below.



```
1 2 50
2 3 20
3 2 20
2 4 30
4 5 20
2 5 150
140 1 2 3 2 4 5
```

Chapter 4: Analysis and Comments

After the tests, I know that the program can pass during many complex testing point.

time and space complexity

The program has M vertice and N lines input. The space complexity is O(N^2), because we have two-dimension array to keep the path and each road may have M position. It will be changed but not less than O(M^2) because keeping the edges of vertices using at least M^2 array.

The time complexity is When the adjacency matrix is represented, the time required to find the adjacency points of each vertex is O(M), and to find the whole matrix, the total time degree is O(M^2). And in other places, time complexity isn't up to O(M^2). So time complexity is O(M^2).

Appendix: Source Code (in C)

```c
#include <stdio.h>

int keep[10000][10000];       // the path length of 2 edges
int ifgo[10000][10000] = {}; // if the path has been visited
int dis[10000] = {};          // the lengths of ways
int k = 0;                    // how many length has been kept
int M, N;                     // M is total number of vertice, N is the number of
lines
int left = 0, right = 0;      // the totalmin edge's left
int totalmin = 999999;        // the min path of total graph
int onemin = 999999;          // the min path of any 2 vertices
int way[10000][10000];        // all the way to be judged
int str[100];                 // temprorily record the way

void Read()
{
    int i;
    int a, b, distance;      // help read the length of edges
    for (i = 0; i < N; i++) // read the input
    {
        scanf("%d %d %d", &a, &b, &distance);
        keep[a][b] = distance;         // undirected path so only keep[a][b]
changes
        if (keep[a][b] && keep[b][a]) // both have road, can go back
        {
            if (onemin < distance) // going back only need a min path
            {
                onemin = distance; // the min path to go back
                left = a;          // the min path where to go back
                right = b;
            }
        }
    }
}

void Recursion(int n, int length, int num) // dfs: recursion to get all the
paths
{
    int ipp;
    for (ipp = num + 1; ipp <= 100; ipp++)
    {
        str[ipp] = 0; // reset the path (avoid situation that last road is
longer than this, but not delete)
```

```
    }
    str[num] = n; // upload current path
    if (n == M)   // reach end, end recursion
    {
        if (length < totalmin) // upload totalmin
        {
            totalmin = length;
        }
        if (length < totalmin + 2 * onemin)
        {
            // printf("%d ", length);
            dis[k] = length; // only keep possible second road
            int p;
            for (p = 0; p <= 100; p++) // copy current path to way
            {
                way[k][p] = str[p]; // number k length maps number k way
                if (str[p] == M)    // avoid copy other paths
                    break;
            }
            // printf("%d %d %d %d\n", way[k][0], way[k][1], way[k][2], way[k]
[3]);
            k++; // decrease the space occupation
        }
        return; // end
    }
    int i;
    for (i = 1; i <= M; i++) // recurse the adjacency matrix
    {
        if (keep[n][i] != 0 && ifgo[n][i] == 0) // path exists and has not been
visited
        {
            ifgo[n][i]++;                                    // upload ifgo[n][i]
            Recursion(i, length + keep[n][i], num + 1); // continue recursion
            ifgo[n][i]--;                                    // release ifgo[n][i]
after
        }
    }
    return; // end after all the recursion has been completed
}
void SortPrint() // sort dis[] and print second shortest path
{
    dis[k] = totalmin + 2 * onemin; // possible second shortest path(backtrack)
    int i, j;
    int min = dis[0];
    for (i = 1; i <= k; i++) // select the min road
    {
        if (dis[i] < min)
        {
            min = dis[i];
        }
    }
    for (i = 0; i <= k; i++)
    {
        if (dis[i] == min) // make sure clean all the min road
            dis[i] = 999999;
    }
    min = dis[0];
    for (i = 1; i <= k; i++) // choose strictly second shortest path
    {
        if (dis[i] < min)
        {
            min = dis[i]; // remember second shortest path
        }
    }
    for (i = 0; i <= k; i++)
    {
        if (dis[i] == min)
        {
            printf("%d ", min); // print second shortest path
            j = 0;
            if (j != k) // not backtrace
            {
                while (way[i][j] != 0)
                {
                    printf("%d ", way[i][j]); // print the path
                    j++;
                }
            }
            else // backtrace
            {
                while (way[i][j] != 0)
                {
                    printf("%d ", way[i][j]);
                    int ff = 1; // keep print only one backtrace
                    if (way[i][j] == left && ff)
                    {
                        printf("%d %d ", right, left);
                        ff = 0;
                    }
                    if (way[i][j] == right && ff) // maybe first reach left or
right
                    {
                        printf("%d %d", left, right);
                        ff = 0;
                    }
                    j++;
                }
            }
        }
    }
    return; // end the function
}
int main()
{
    scanf("%d %d", &M, &N);
    Read();                 // read lines
    Recursion(1, 0, 0); // dfs: recursion to get all the paths
    SortPrint();            // sort dis[] and print second shortest path
}
```

Declaration

I hereby declare that all the work done in this project titled "The 2nd-shortest Path" is of my independent effort.