Introduction

Here is part of the introduction from PTA:

> 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).

Since the railways are unidirectional, so I take the graph as a directed graph. In fact, since the road can be used more than once, the algorithm also works for a undirected graph(like Input 6). Here are some specifications based on my **own** comprehension of the introduction from PTA :

1. There is no self loop, for I think the distance of going from one station and return to one station, not passing any other stations, is hard to define. I think it is meaningless and not realistic;
2. The last station is allowed to appear within the path, not only the end of the path, which can be interpreted as you can take a walk from home and then return if you want to walk a little more;
3. If there is no second shortest path, which means that the length of the 'second shortest path' is equal to the length of the shortest path, the program will just output the result that is the same as the shortest path, for I think they are the same in this occasion. Or we can compare the shortest distance and the second shortest distance before output. If they are the same, make the program output "There is no the second shortest path". It's a simple modification;
4. The distance between two stations cannot be 0. If the distance between two stations is 0, then they are exactly one station. And for convenience, I just define the distance to be an integer. In reality, it can be floating point number and can be very large, but the modification takes little effort, so I just use type integer for convenience;

Algorithm Specification

Summary

To solve the problem, I modify the Dijkstra algorithm.

At first, we need to define the matrix representing the graph consisting of railways and stations. Each element in the matrix has two members. 'length' represents the distance between two stations. If two stations are unconnected, 'length' is 0. 'visit' is a signal convenient for quicker search for loops. The definition is as follow:

```
typedef struct graphnode{
    int length;
    int visit;
}gnode;
```

And the definition of the list storing the result is as follow. Compared to the original Dijkstra algorithm, I add some members to store the information about the second shortest path. The usage of each member has been declared:

```
typedef struct Node{
    int shortdist;      // the shortest distance
    int shortpath;      // last vertex of the shortest path
    int secshortdist;   // the second shortest distance
    int secshortpath;   // last vertex of the seconde shortest path
    int *backtrack;     // the vertices for backtrack (if necessary)
    int pathflag;       // to signal how to traverse the path
    int known;          // to signal whether the information of a vertex has
been settled
}node;
```

So, to solve the problem, using the modified Dijkstra algorithm, using majorly some arrays, the steps should be:

1. Get the input;
2. Initialize the matrix and the list(also an array for searching for loops);
3. Apply the modified Dijkstra algorithm;
4. Output the result;

```
int main()
{
    int M, N;
    scanf ("%d %d", &M, &N);    // get the input
    Initialization();
    getInput(M, N);

    moddij();

    Output();

    return 0;
}
```

Get the Input

It is easy and so it is omitted.

Initialize the matrix and the list

It is also easy. Set all the graph[i][j].length=graph[i][j].visit=0.

Set all the list[i].shortdist=list[i].secshortdist=2147483647(which can be taken as infinity), list[i].secshortpath=list[i].shortpath=list[i].known=list[i].pathflag= 0, allocate space for the backtrack array and then set list[i].backtrack[0]=0. It means that there is no loop yet.

Apply the modified Dijkstra algorithm;

This step is the most difficult to achieve.

First of all, we need three functions. One is the modified Dijkstra algorithm, one is the function setting the shortest loop and setting the backtrack array and one is a recursive function searching for the shortest loop.

```
void moddij();          // a modified Dijaska algorithm
int findminloop(int st);// find the shortest loop starting from st
int loop(int st, int end, int top); // a recursive function serving for
findminloop
```

The beginning steps of the modified algorithm are the same as the original one. There are only three possible values of the second shortest distance of ***w***: following the shortest path of ***v*** to ***w***, following the second shortest path of ***v*** to ***w***, following the shortest path of ***v*** to ***w*** with a shortest loop starting from ***w***. In pseudo-code :

```
void moddij()
{
    int v, w, dis=0, i;
    // initialize the information of the first station
    setList1();

    while(1)
    {
        v=the unknown vertex of the shortest distance;
        if(v==0)  //no unknown station
            break;
        list[v].known=1;
        for(each w adjacent to v)
        {
            if(!list[w].known)  // update the shortest path
            {
                if(list[v].shortdist+graph[v][w].length<list[w].shortdist)
                {
                    list[w].shortdist=list[v].shortdist+graph[v][w].length;
                    list[w].shortpath=v;
                }
            }
            int temp=findminloop(w);
            list[w].secshortdist=the minimum of list[v].shortdist+graph[v]
[w].length, list[v].secshortdist+graph[v][w].length and
list[v].shortdist+graph[v][w].length+temp;
            list[w].secshortpath=v;
            setPathFlag;
        }
    }
}
```

Then for the function setting the shortest loop, the function tries to find a loop starting from the start ***st***. If the newly found loop is shorter than last one, renew the according information. In pseudo-code:

```
int findminloop(int st)
{
    int minlength=infinity, lth=0,v;
    for(each v adjacent to st)
    {
        store v in array lp;
        lth=loop(v, st, 1);
        if(the loop exist and the loop is shorter than minlength)
            renew the array list[st].backtrack;
        else
        {
            lth=0;
            lp[0]=0;
        }
    }
    if(minlength==infinity) // no loop exists
        minlength=0;
    return minlength;       // the second shortest path is as long as the
shortest one
}
```

Then for the recursive function searching for the shortest loop, the algorithm traverses all the possible ways to form a loop. If all the stations adjacent to ***st*** has been check and the function doesn't return a meaningful value, it returns -1, which means there is no loop. In pseudo-code:

```
int loop(int st, int end, int top)
{
    int temp;
    for(each i adjacent to st)
    {
        if(i==end)
        {
            set the end of lp to 0;
            return the distance from st to end;
        }
        if(the edge between st and i has not been visited)
        {
            store i in lp;
            set the edge to be visited;
            if(temp=loop(i, end, top+1) exists)
            {
                set the edge to be not visited;
                return the distance from st to i plus temp;
            }
            else
            {
                set the end of lp to 0;
                set the edge to be not visited;
            }
        }
    }
    return -1;
```

```
}
```

Output the Result

It is relatively simple, a value named ***lock*** is set. Once a loop has been output, ***lock*** is cleared, because a second shortest path includes one loop at most. If ***lock*** is 0, then any loop is not allowed to be output and except the loop, the second shortest path goes the same as the shortest path.

Then, if ***pathflag*** is 3 and ***lock*** is not 0, a loop should be output.

For the next station to be output, the algorithm needs to check ***pathflag*** of the last station. If it is 1, the second shortest path goes along the shortest path to current station. If it is 2, the second shortest path goes along the second shortest path to current station.

And we should notice that, since the algorithm can only find the path reversely, it needs a stack to output the result in correct order.

In pseudo-code:

```
for(int i=the last station; i; )
{
    stack[++top]=i;
    if(a loop exists and pathflag==3 and lock==1)
    {
        store the loop into the stack;
        lock--;
    }
    if(lastflag==1 or lock==0)
    {
        i = the last staion of the shortest path;
    }
    else if(lastflag==2)
    {
        i = the last staion of the second shortest path;
    }
}
output the second shortest distance;
ouput the stack;
```

Testing Results

Input 1

The sample given by PTA.

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

Desired output:

```
240 1 2 4 5
```

Actual output:

```
240 1 2 4 5
```

Pass

Input 2

A case including a bidirectional railway and a loop.

```
5 6
1 2 10
2 3 20
3 2 20
3 4 40
4 2 10
4 5 10
```

Desired output:

```
120 1 2 3 2 3 4 5
```

Actual output:

```
120 1 2 3 2 3 4 5
```

Pass

Input 3

A case in which traveling from the last station and going back to the last station form the second shortest path.

```
3 3
1 2 10
2 3 20
3 2 20
```

Desired output:

```
70 1 2 3 2 3
```

Actual output:

```
70 1 2 3 2 3
```

Pass

Input 4

There is a loop incorporating the first station.

```
4 4
1 2 20
2 3 30
3 1 10
2 4 10
```

Desired output:

```
90 1 2 3 1 2 4
```

Actual output:

```
90 1 2 3 1 2 4
```

Pass

Input 5

A case in which part of several loops are overlapped.

```
6 7
1 2 10
2 3 20
3 4 10
3 5 30
4 5 5
4 6 5
5 2 10
```

Desired output:

```
90 1 2 3 4 5 2 3 4 6
```

Actual output:

```
90 1 2 3 4 5 2 3 4 6
```

Pass

Input 6

A bidirectional graph, also an undirected graph.

```
6 12
1 2 30
2 1 30
1 3 35
3 1 35
2 4 40
4 2 40
3 5 40
5 3 40
4 6 50
6 4 50
5 6 50
6 5 50
```

Desired output:

```
125 1 3 5 6
```

Actual output:

```
125 1 3 5 6
```

Pass

Input 7

A case inspired from the railway map of China, in which Lisa is going from Shenzhen to Beijing.

```
5 10
1 2 161
1 3 1512
1 4 1814
2 1 161
2 3 1440
2 5 2478
3 1 1512
3 2 1440
3 5 1493
4 5 1490
```

Desired output:

```
2961 1 2 1 2 5
```

Actual output:

```
2961 1 2 1 2 5
```

Pass

Analysis and Comments

Analysis

Assume that there are *M* stations and *N* unidirectional railways. Since `moddij()` and `findminloop()` use `loop()`, and `loop()` obviously has the largest time complexity, the three functions have the same time complexity.

Also, they all use the same arrays and have local variables of a constant number, so the three functions have the same space complexity.

- `loop()`

In worst case, the graph is a complete graph. The ways to form a loop is larger than *M!*. But in reality or in average, it is not that inefficient. The worst time complexity is too large and the average time complexity is hard to analyze, so I only give the space complexity.

The largest array used is the matrix ***graph***, so the space complexity is $O(M^2)$.

space complexity: $O(M^2)$

- `findminloop()`

space complexity: $O(M^2)$

- `moddij()`

space complexity: $O(M^2)$

Comments

As far as I can see, there are two possible improvements:

1. Modify the structure of array storing the result. Make it simpler and easier to use.
2. Create a better algorithm for finding the shortest loop. The current function `loop()` is not efficient in worst case.

Program

The code of the program is as follow, which can also be obtained in sec_shortest.cpp:

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

// the structure of the graph
typedef struct graphnode{
    int length;
    int visit;
}gnode;

// the sturcture of the nodes in the path
typedef struct Node{
    int shortdist;      // the shortest distance
    int shortpath;      // last vertex of the shortest path
    int secshortdist;   // the second shortest distance
    int secshortpath;   // last vertex of the seconde shortest path
    int *backtrack;     // the vertices for backtrack (if necessary)
```

```
    int pathflag;       // to signal how to traverse the path
    int known;          // to signal whether the information of a vertex has
been settled
}node;

gnode **graph;          // the matrix of the graph
node *list;             // the list storing the result
int Mnode, Nedge;       // the input M for the number of nodes and N for the
number of edges
int *lp;                // store the loop temporarily
void moddij();          // a modified Dijaska algorithm
int findminloop(int st);// find the shortest loop starting from st
int loop(int st, int end, int top); // a recursive function serving for
findminloop

int main()
{
    scanf("%d %d", &Mnode, &Nedge);
    // initialize the matrix representing the graph
    graph=(gnode **)malloc(sizeof(gnode*)*(Mnode+1));
    for (int i=1; i<=Mnode; i++)
    {
        graph[i]=(gnode*)malloc(sizeof(gnode)*(Mnode+1));
    }
    for (int i=1; i<=Mnode; i++)
    {
        // 0 stands for no connection and not visited
        for(int j=1; j<=Mnode; j++)
        {
            graph[i][j].length=graph[i][j].visit=0;
        }
    }

    // initialize the list storing the result
    list=(node*)malloc(sizeof(node)*(Mnode+1));
    for(int i=1; i<=Mnode; i++)
    {
        list[i].shortdist=list[i].secshortdist=2147483647;
        list[i].secshortpath=list[i].shortpath=list[i].known=list[i].pathflag=
0;
        list[i].backtrack=(int *)malloc(sizeof(int)*(Mnode+1));
        list[i].backtrack[0]=0;
    }

    // initialize the list storing the result temporarily
    lp=(int *)malloc(sizeof(int)*(Mnode+1));
    lp[0]=0;

    // input
    for (int i=0; i<Nedge; i++)
    {
        int v, w, length;
        scanf("%d %d %d", &v, &w, &length);
        graph[v][w].length=length;
    }

    // generate the path and find the result
    moddij();
```

```
    // just for debug
    /*
    for(int i=1; i<=Mnode; i++)
    {
        for(int j=1; j<=Mnode; j++)
            printf(" *%d* ", graph[i][j]);
        putchar('\n');
    }
    for(int i=1; i<=Mnode; i++)
    {
         printf("%d: %d %d %d %d %d %d\n", i, list[i].shortdist,
list[i].shortpath, list[i].secshortdist, list[i].secshortpath, list[i].pathflag,
list[i].known);
         for(int j=0;  list[i].backtrack[j]; j++)
            printf("%d ",  list[i].backtrack[j]);
        putchar('\n');
    }
    */

    // output, for the last node, it must go along the path of the second
shortest distance, so last =2
    // set lock to 1 for there is at most one loop in a second shortest path
    int *stack, top=-1, lock=1, last=2;
    stack=(int *)malloc(sizeof(int)*(2*Mnode));
    for(int i=Mnode; i; )
    {
        stack[++top]=i;                 // use a stack to get the correct path
starting from vertex 1
        // there is a backtrack
        if(list[i].backtrack[0] && list[i].pathflag==3 && lock)
        {
            int j;
            for(j=0; list[i].backtrack[j]; j++);
            for(j; j>0; )
            {
                stack[++top]=list[i].backtrack[--j];
            }
            stack[++top]=i;
            lock--;
        }
        // once a loop has been output, the path must go along the shortest path
        if(last==1 || !lock)
        {
            i=list[i].shortpath;
        }
        else if(last==2)
        {
            last=list[i].pathflag;      // renew the flag signaling how the path
goes
            i=list[i].secshortpath;
        }
    }
    printf("%d", list[Mnode].secshortdist);
    while(top!=-1)
    {
        printf(" %d", stack[top--]);
    }
```

```
    //putchar('\n');
    //system("pause");
    return 0;
}

void moddij()
{
    int v, w, dis=0, i;
    // initialize list[1]
    list[1].shortdist=0;
    if((list[1].secshortdist=findminloop(1))>0)
    {
        list[1].pathflag=2;
    }
    else
    {
        list[1].pathflag=1;
    }
    // generate the other results
    while(1)
    {
        // search for the unknown vertex with the smallest distance
        v=0;
        dis=2147483647;
        for(int i=1; i<=Mnode; i++)
        {
            if(!list[i].known && list[i].shortdist<dis)
            {
                v=i;
                dis=list[i].shortdist;
            }
        }
        if(!v)                      // no such unknown distance vertex
        {
            break;
        }
        list[v].known=1;
        for (w=1; w<=Mnode; w++)
        {
            if(!graph[v][w].length)
                continue;
            if(!list[w].known)
            {
                // update the shortest path
                if(list[v].shortdist+graph[v][w].length<list[w].shortdist)
                {
                    list[w].shortdist=list[v].shortdist+graph[v][w].length;
                    list[w].shortpath=v;
                }
                // update the second shortest path, check for all the
possibilities
                int temp=findminloop(w);
                // the second shortest path includes a loop
                if(temp>0 && list[v].shortdist+temp+graph[v]
[w].length>list[w].shortdist && list[v].shortdist+temp+graph[v]
[w].length<list[w].secshortdist)
                {
```

```
                    list[w].secshortdist=list[v].shortdist+temp+graph[v]
[w].length;
                    list[w].secshortpath=v;
                    list[w].pathflag=3;
                }
                // the second shortest distance of v plus the distance between v
and w
                if(list[v].secshortdist+graph[v][w].length>list[w].shortdist &&
(list[v].secshortdist+graph[v][w].length<list[w].secshortdist ||
list[w].shortdist==list[w].secshortdist))
                {
                    list[w].secshortdist=list[v].secshortdist+graph[v]
[w].length;
                    list[w].secshortpath=v;
                    list[w].pathflag=2;
                }
                // the shortest distance of v plus the distance between v and w
                if((list[v].shortdist+graph[v][w].length>list[w].shortdist &&
list[v].shortdist+graph[v][w].length<list[w].secshortdist) ||
list[w].secshortdist==2147483647)
                {
                    list[w].secshortdist=list[v].shortdist+graph[v][w].length;
                    list[w].secshortpath=v;
                    list[w].pathflag=1;
                }
            }
        }
    }
}

int findminloop(int st)
{
    int minlength=2147483647, lth=0, v;
    for (v=1; v<=Mnode; v++)
    {
        if(graph[st][v].length)                             // possibly there is
a loop
        {
            lp[0]=v;
            lth=loop(v, st, 1);
        }
        if(lth>0 && lth+graph[st][v].length<minlength)      // renew the
shortest loop
        {
            for(int i=0; list[st].backtrack[i]=lp[i]; i++);
            minlength=lth+graph[st][v].length;
            lth=0;
        }
        else
        {
            lth=0;
            lp[0]=0;
        }
    }
    // there is no loop, the second shortest path is as long as the shortest
path
    if(minlength==2147483647)
    {
```

```
        minlength=0;
    }
    return minlength;
}

int loop(int st, int end, int top)
{
    int temp;
    for(int i=1; i<=Mnode; i++)
    {
        if(i==end && graph[st][i].length)               // back to the start of
the loop
        {
            lp[top]=0;
            return graph[st][i].length ;
        }
        if(graph[st][i].length && !graph[st][i].visit)
        {
            lp[top]=i;
            graph[st][i].visit=1;
            if((temp=loop(i, end, top+1))<0)            // not a loop
            {
                lp[top]=0;
                graph[st][i].visit=0;
            }
            else                                        // a loop forms
            {
                graph[st][i].visit=0;
                return graph[st][i].length+temp;
            }
        }
    }
    return -1;                                          // no loop starting from
vertex end
}
```