The 2nd-shortest Path

Date:2022-11-27

Content

Chapter 1: Introduction

There are **m points and n paths**, where the paths are all **unilateral paths**. Find **the 2nd-shortest path** weight from the first point to the nth point and output this path.

Input Specification:

Each input file contains one test case. For each case, the first line gives two positive integers $M(1 \leq M \leq 1000)$ and $N(1 \leq N \leq 5000)$ 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 $D(1 \leq D \leq 5000)$.

Output Specification:

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

2.1 The main function

First read in the data, I record the data in the form of an **adjacency list**.
Then use **Dijkstra's algorithm** to relax each edge to find the shortest path and the second shortest path.
Finally, the weight of the next shortest path and each point in the shortest path are output by **recursive method**.

2.2 The data structure

```
/*We use the adjacency table to represent the graph,
edge structure to represent an edge, next represents the
serial number of the next edge from the same starting point
of this edge, to represents the end point of this edge,
and w is the weight of this edge*/
struct edge {
    int next;
    int to;
    int w;
}Edge[MAXN_EDGE];
/*The head array contains the last
edge entered starting from the following punctuation*/
int Head[MAXN_POINT];
//Idx is a marking symbol that records how many edges are entered
int idx = 1;

/*The shortest path required from the starting
```

```
point to each point is stored in the min2Dis array,
and its value is stored in min2Dis every time the
secondary path is updated*/
int minDis[MAXN_POINT], min2Dis[MAXN_POINT];
/*The shortest path is recorded in the pathmin array,
and the updated minor path is recorded in pathmin2*/
int pathMin2[MAXN_POINT], pathMin[MAXN_POINT];

//Size is the number of elements in the current heap
int size;
/*Two values are stored in the heap:
the subscript of the point and the distance from
the starting point to this point (which may be the
shortest or the second shortest)*/
struct Heap {
    int dis;
    int endPoint;
} Elements[MAXN_POINT];
/*Change is used to pass the updated value
and record the shortest path that was changed last time*/
int change;
```

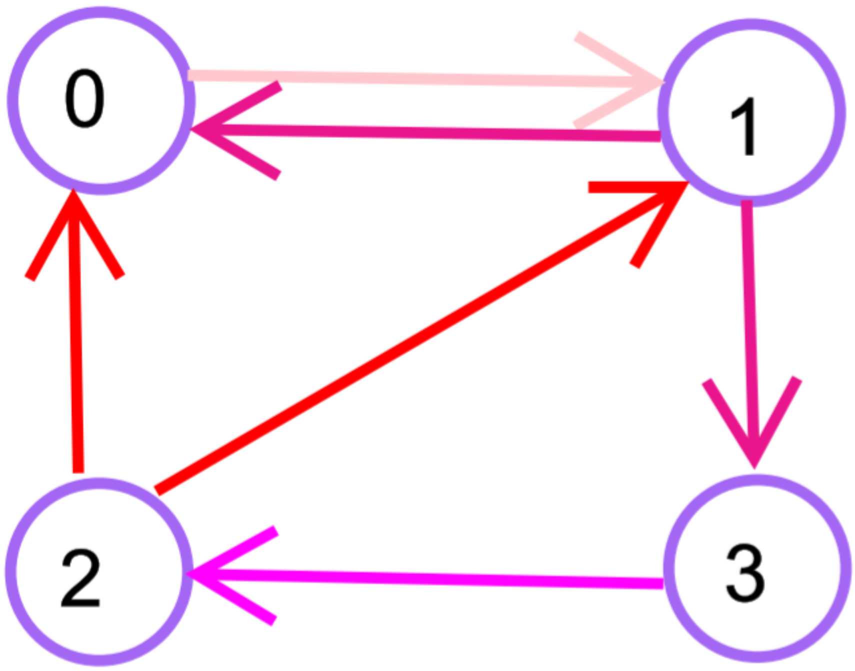
This time I mainly used two data structures: **adjacency list** and **heap**.

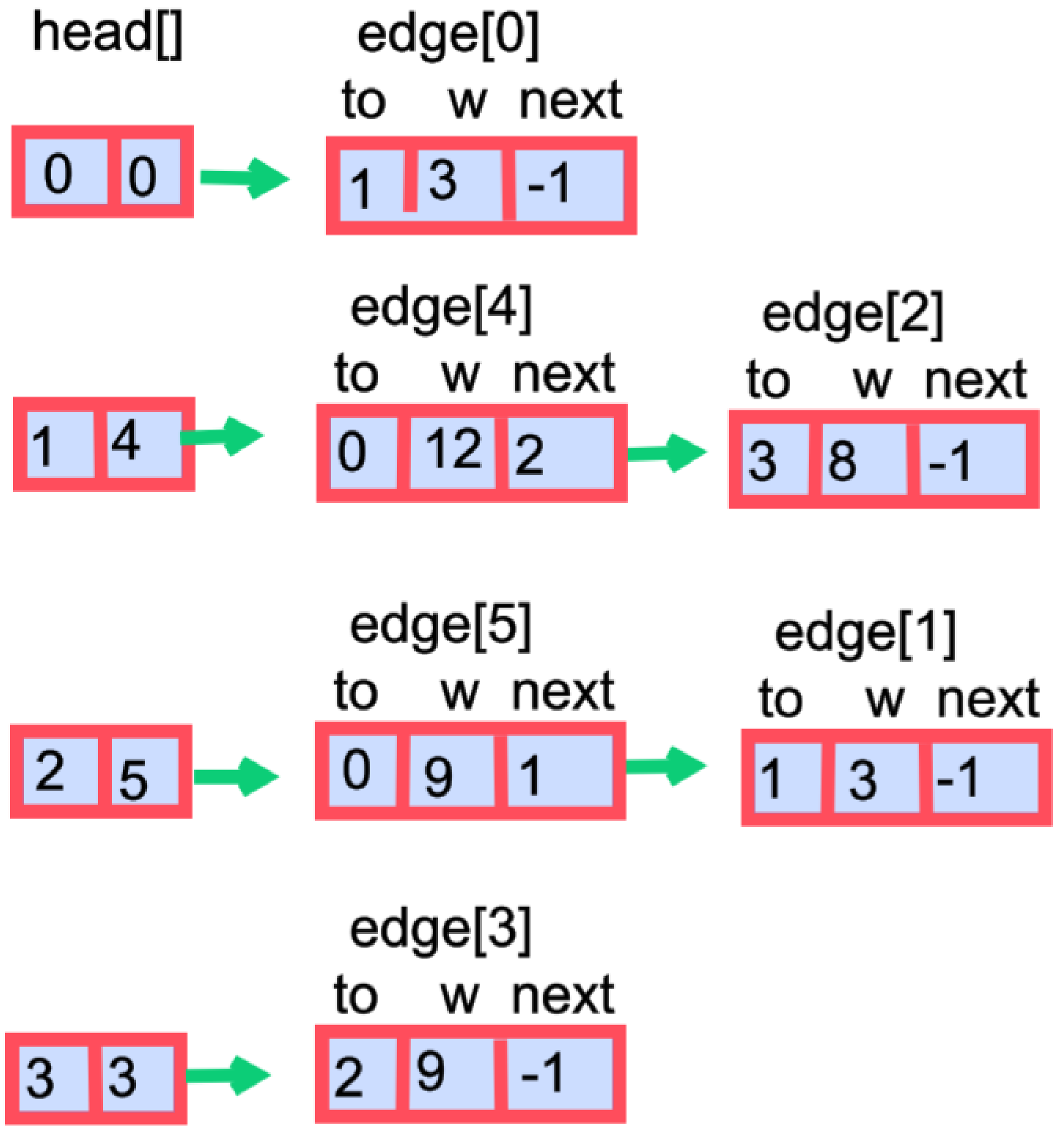
2.2.1 adjacency list

According to the title description, this graph is a **sparse graph**, so the general method of storing this graph (that is, using a two-dimensional array to record the edges between every two points) will make the **space complexity** too high. So I store this graph in the form of **adjacency table**.

The graph is stored as edges. Read in the information of each edge, store the edges in the array, and sort the edges in the array according to the **starting point order**.

A diagram and its corresponding storage structure are shown below:


0
1
2
3


head[]
0 0
1 4
2 5
3 3
edge[0]
to w next
1 3 -1
edge[4]
to w next
0 12 2
edge[2]
to w next
3 8 -1
edge[5]
to w next
0 9 1
edge[1]
to w next
1 3 -1
edge[3]
to w next
2 9 -1

2.2.2 heap

According to the general Dijkstra algorithm, it is necessary to find the minimum value in the undetermined array for each edge relaxation operation. If I use an array to store it, the **time complexity** will be very high, so I use the minimum heap to store the undetermined array, so that the smallest element can be directly taken from the heap each time.

2.3 read data

First, **initialize the array** of the first edge with each point as the starting point to - 1, then read in all edges, and set the corresponding end point of each edge, subscript and weight of the next edge in turn.

2.4 dijkstra algorithm

First, set the distance between the starting point and the starting point to 0, and put the starting point in the heap.

When the heap size is not zero, take out the minimum heap size and record it. If the minimum heap size is greater than the current secondary short path, skip it because there is no room for optimization (this operation is called **pruning**).

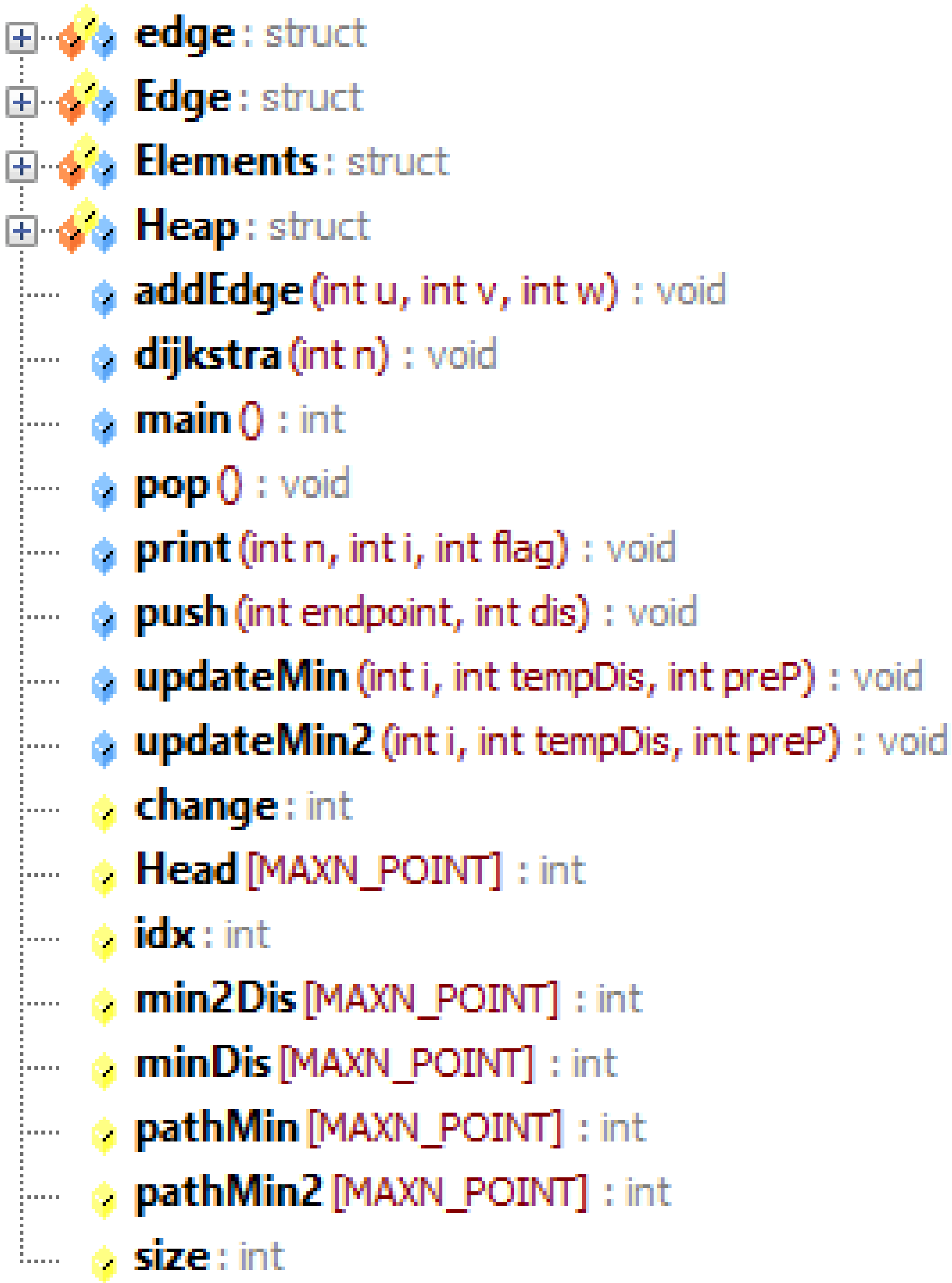
Traverse all edges starting from this point taken from the heap. If the value of this edge plus the path before the starting point is less than the existing minimum value, **update it and record the updated value**. If the recorded value is also less than the current value.

2.5 print result

Flag records whether switch from the secondary short path array to the secondary short path array. If the subscripts in the previous shortest path and the secondary path are the same, but this time they are different, output the value stored in the secondary path array, and mark the flag as 1.

When the flag is 1, the subscript stored in the shortest path is directly output.

2.6 structure



Chapter 3: Testing Results

3.1 TestCase1

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

result:

```
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
--------------------------------
```

3.2 TestCase2

```
4 5
1 2 100
2 4 200
2 3 100
3 4 100
1 4 301
```

result:

```
4 5
1 2 100
2 4 200
2 3 100
3 4 100
1 4 301
301 1 2 4
--------------------------------
```

3.3 TestCase3

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

result:

```
4 6
1 2 1
1 2 5
1 3 2
2 3 2
2 4 1
2 4 6
6 1 2 4
--------------------------------
```

Chapter 4: Analysis and Comments

4.1 The space complexity

Because we mostly use two-dimensional arrays, so the space complexity is $O(|E|+|V|)$, the heap's space complexity is $O(|E|*|V|)$.

4.2 The time complexity

4.2.1 read data

Because all edges are traversed when creating the adjacency table, the time complexity in this step is $O(|V|+|E|)$.

4.2.2 dijkstra algorithm

In the Dijkstra algorithm, we use the minimum heap, so the time complexity of this step is the time complexity of the minimum heap. So the time complexity is $O(|E|log|V|)$.

4.2.3 print result

The route is traversed recursively, so the time complexity is $O(|E|)$.

4.3 Summary

In summary, the space complexity of this program is $O(|E|*|V|)$, the time complexity of this program is $O(|E|log|V|)$.

Appendix: Source Code

```
#include<stdio.h>
#define MAXN_EDGE 4005
#define MAXN_POINT 1005
/*We use the adjacency table to represent the graph,
edge structure to represent an edge, next represents the
serial number of the next edge from the same starting point
of this edge, to represents the end point of this edge,
and w is the weight of this edge*/
struct edge {
    int next;
```

```
    int to;
    int w;
}Edge[MAXN_EDGE];
/*The head array contains the last
edge entered starting from the following punctuation*/
int Head[MAXN_POINT];
//Idx is a marking symbol that records how many edges are entered
int idx = 1;

/*The shortest path required from the starting
point to each point is stored in the min2Dis array,
and its value is stored in min2Dis every time the
secondary path is updated*/
int minDis[MAXN_POINT], min2Dis[MAXN_POINT];
/*The shortest path is recorded in the pathmin array,
and the updated minor path is recorded in pathmin2*/
int pathMin2[MAXN_POINT], pathMin[MAXN_POINT];

//Size is the number of elements in the current heap
int size;
/*Two values are stored in the heap:
the subscript of the point and the distance from
the starting point to this point (which may be the
shortest or the second shortest)*/
struct Heap {
    int dis;
    int endPoint;
} Elements[MAXN_POINT];
/*Change is used to pass the updated value
and record the shortest path that was changed last time*/
int change;

//This function is used to generate the adjacency list
void addEdge(int u, int v, int w) {
    Edge[idx].next = Head[u];
    Edge[idx].to = v;
    Edge[idx].w = w;
//  Update the value of the head array to point to the latest edge
    Head[u] = idx++;
}

//Update the value of the head array to point to the latest edge
void push(int endpoint, int dis) {
//  Heap size plus one
    size++;
//  Place the new value in the last digit
    Elements[size].endPoint = endpoint;
    Elements[size].dis = dis;
    int i = size;
//  Push the last bit up until the heap conforms to the property of the smallest heap
    for(; Elements[i/2].dis > dis; i /= 2) {
        Elements[i].dis = Elements[i/2].dis;
        Elements[i].endPoint = Elements[i/2].endPoint;
    }
```

```
    Elements[i].dis = dis;
    Elements[i].endPoint = endpoint;
}

//This function is used to delete the first element in the heap
void pop() {
//  Place the last element in the first
    Elements[1].dis = Elements[size].dis;
    Elements[1].endPoint = Elements[size--].endPoint;
//  Record the elements to be moved down
    int LastElementDis = Elements[1].dis;
    int LastElementP = Elements[1].endPoint;
    int i = 1, child;
//  Push the last bit up until the heap conforms to the property of the smallest heap
    for(; i*2 <= size; i = child) {
        child = i*2;
        if (child != size && Elements[child+1].dis < Elements[child].dis)
            child++;
        if ( LastElementDis > Elements[child].dis ) {
            Elements[i].dis = Elements[child].dis;
            Elements[i].endPoint = Elements[child].endPoint;
        }
        else     break;
    }
    Elements[i].dis = LastElementDis;
    Elements[i].endPoint = LastElementP;
}

//Update shortest path
void updateMin(int i, int tempDis, int preP) {
    /*If the distance between the edge
    and the starting point of the edge is less than
    the minimum distance between the starting point and
    the edge, the minimum value will be updated, and the
    updated minimum value will be recorded to update the
    secondary short path*/
    if(Edge[i].w + tempDis < minDis[Edge[i].to]) {
//      Record the updated value
        change = minDis[Edge[i].to];
        minDis[Edge[i].to] = Edge[i].w + tempDis;
        int temp = preP;
        preP = pathMin[Edge[i].to];
//      Put the updated node back into the heap
        push(Edge[i].to, minDis[Edge[i].to]);
        int j;
//      Record the shortest path
//      pathMin2[Edge[i].to] = pathMin[Edge[i].to];
        pathMin[Edge[i].to] = temp;
    }
}

/*If the updated path value recorded last is between
the current shortest path and the secondary path,
the secondary path will be updated*/
```

```
void updateMin2(int i, int tempDis, int preP) {
//  Judge whether the update conditions are met
    if(change > minDis[Edge[i].to] && change < min2Dis[Edge[i].to]) {
        min2Dis[Edge[i].to] = change;
//      Put the updated secondary short path into the heap
        push(Edge[i].to, min2Dis[Edge[i].to]);
        pathMin2[Edge[i].to] = preP;
    }
}

//This function is used to execute the Dijkstra algorithm
void dijkstra(int n) {
//  The distance from the starting point to the starting point is set to 0
    minDis[1] = 0;
//  Put the starting point in the pile
    push(1, minDis[1]);
//  When the heap is not empty
    while(size) {
//      Record the smallest element in the current heap
        int tempPoint = Elements[1].endPoint;
        int tempDis = Elements[1].dis;
//      Pop the smallest element out of the heap
        pop();
        /*If the smallest element is larger than the
        secondary short path, it is directly performed*/
        if(tempDis > min2Dis[tempPoint]) continue;
        int i;
        /*Traverse every edge starting from this point,
        and update the shortest path and secondary path*/
        for(i = Head[tempPoint]; i != -1; i = Edge[i].next) {
            change = Edge[i].w + tempDis;
            updateMin(i, tempDis, tempPoint);
            updateMin2(i, tempDis, tempPoint);
        }
    }
}

//Recursive output secondary short path
void print(int n, int i, int flag) {
    int res;
//  Judgment of recursive end point
    if(i > 1){
//      Whether the subscripts stored
//      in the secondary short path and the shortest path are different
        if(flag == 0) {
            /*If the subscripts in the previous shortest path
            and the secondary path are the same, but this time
            they are different, output the value stored in the
            secondary path array, and mark the flag as 1*/
            if(pathMin2[i] != pathMin[i] || pathMin2[i-1] == 0) {
                print(n, pathMin2[i], 1);
                res = pathMin2[i];
            }else{
                print(n, pathMin2[i], 0);
                res = pathMin2[i];
```

```
                }
            }else{
//              When the flag is 1, the subscript
//              stored in the shortest path is directly output
                print(n, pathMin[i], 1);
                res = pathMin[i];
            }
            printf(" %d", res);
        }
        else return;
}

int main() {
    int m, n, i;
//  Enter map size and number of paths
    scanf("%d%d", &n, &m);
    /*Initialize the path between
    the points so that they are disconnected from each other*/
    for(i = 0; i < MAXN_POINT; i++) {
//      -1 represents no edge starting from this point
        Head[i] = -1;
        minDis[i] = 65535;
        min2Dis[i] = 65535;
    }
//  Enter each edge
    for(i = 0; i < m; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        addEdge(u, v, w);
    }
//  Call Dijkstra function
    dijkstra(n);
    /*If the secondary short path is still initialized after execution,
    it means that there is no secondary short path*/
    if(min2Dis[n] == 65535)printf("There is no path!\n");
    else{
//      Output results
        printf("%d", min2Dis[n]);
        print(n, n, 0);
        printf(" %d", n);
    }
}
```

Declaration

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