The 2nd-shortest Path

朱理真

Date: 2020-12-4

Chapter 1: Introduction

The 2-nd shortest path in a graph is the path from the source to its destination with traceback permitted, which is longer than the shortest path, but is no longer than any other path.

Actually, there has been a lot of algorithms that solve the kth-shortest path problem using Dijkstra, a*, d* or other methods to find the shortest paths, and the time complexity are also liner-logarithmic.

Our duty is to find the second shortest path with a given graph and get how long it is.

Input Specification:

Give M(1, 1000)(vertices), N(1, 5000)(edges), and for next each line, provide an edge with the form “vertex1 vertex2 weight”

Output Specification:

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

Data Structures

```
typedef int element; //used for elements in the stack, which are vertex
 numbers

NODE
typedef struct node // used in ADJ linked list as a node
{
    int vertex;
    int weight;
    struct node* next;
}*list;

GRAPH
typedef struct g{
    int v; //The number of the vertices in graph
    int e; //The number of edges in graph
    list* AdjList; // ADJ linked list, with edges storing in it.
}*graph;

HEAP
typedef struct h
```

```
{
    int capacity; //the max number of elements a heap can hold.
    int* vertices; //store the vertices number for every node. vertices
[0] = the number of elements in the heap.
    int* distance; //store the distance of each vertex.
}*heap;


STACK
typedef struct s
{
    int top; //stack top
    int capacity; //the max number of elements a heap can hold.
    element* container; //store the elements in the stack, that is, a v
ertex number.
}*stack;
```

Main Algorithm

```
main(){
    Create graph g and read the data from input.
    Create heap h.

    Calculate the shortest path from end vertex to start vertex and
return the shortest length and path information.

    Calculate the second shortest path form start vertex to end vertex
and return the second shortest length and path information.

    Print the second shortest length.

    Using the path information, print the path.
}
```

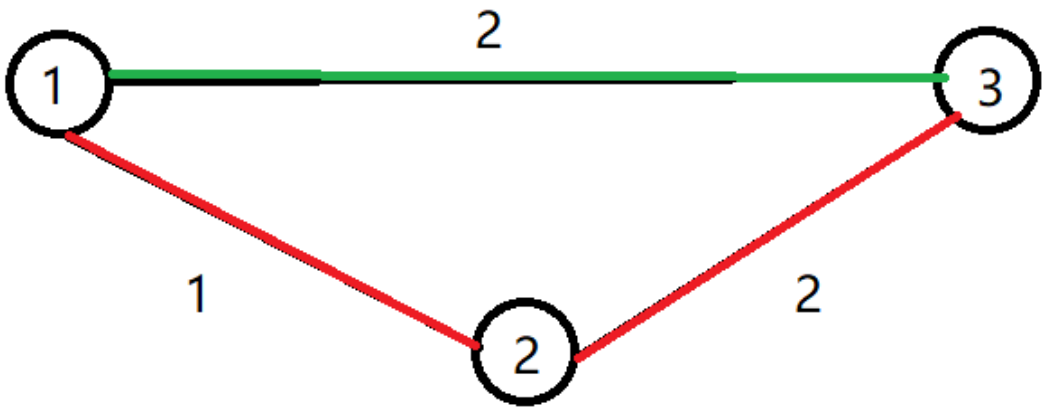
Key Algorithms

```
ShortestPath(heap, graph, distances array, source(start point),
destination, path array){
    Create an array named include, recording if each vertex has been
popped from the heap. All elements are not included at the beginning.

    Initialize the distance and path array.
    Make Heap Empty.

    insert the source into the heap.
```

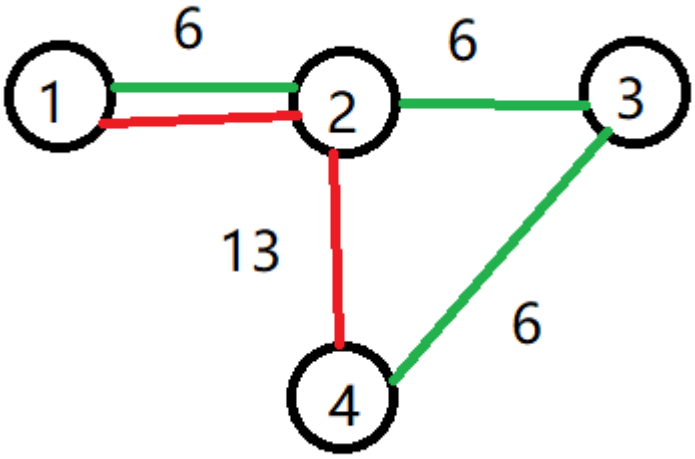
```
    while(the heap is not empty){
        vertex = the element popped from the heap top.
        if vertex has already been included
            continue.
        Include the vertex.

        for each neighbor vertex of current vertex{
            if (neighbor is not included yet){
                if (the distance of neighbor is larger
                  than the distance of current vertex plus
                  the weight between the them){

                    distances[neighbor]=distances[vertex]+weight.
                    path[neighbor]=vertex.
                    Insert the neighbor with its new distance into heap.
                 }
            }
        }
    }
    return the distances[destination]
}

SecondShortestPath(heap, graph, source, destination, distances array,
second_distance array, path array, second_path array){

    Initialize the distances array, path array.
    Initialize the second_distances array, second_path array.

    insert the source into the heap.
    while(the heap is not empty){
        vertex = the element popped from the heap top.
        distance = the distance of the vertex just poped.
        if distance > second_distances[vertex]
            continue.

        for each neighbor vertex of current vertex{
            NewDistance = distance + weight between vertex and neighbor.
            NewVertex = vertex.
            if (NewDistance < distances[neighbour]){

                 swap(NewDistance, distances[neighbor]
                 swap(NewVertex, path[neighbor])
                 Insert the neighbor with distances[neighbor] into the
heap.
```

```
            }
          if (NewDistance > distances[neighbor] and
           NewDistance < second_distances[neighbor]){

              second_distances[neighbor] = NewDistance.
              second_path[neighbor] = NewVertex.
              insert the neighbor along with second_distances[neighbor]
into the heap.
          }
      }
    }
    return second_distance[destination]
}

PrintPath(
    stack,
    end vertex,
    shortest length from start vertex to end vertex,
    distances array (stores the shortest distances from start vertex),
    reverse_distances array (shortest distances from end vertex),
    path array,
    reverse_path array,
    second path array
){
    vertex = end vertex
    backed=FALSE
    while(the vertex is not the break point “BEGIN”){
        if(
            the vertex is not in the shortest path OR
            the vertices in the shortest path between the current vertex
and last vertex just pushed are skipped OR
            the last vertex of last vertex is the same as the current
vertex
        )//if you don’t understand, see the pictures below.
            backed = TRUE
        push the vertex into stack

        if backed
            vertex = path[vertex]
        else
            vertex = sec_path[vertex]
    }
```

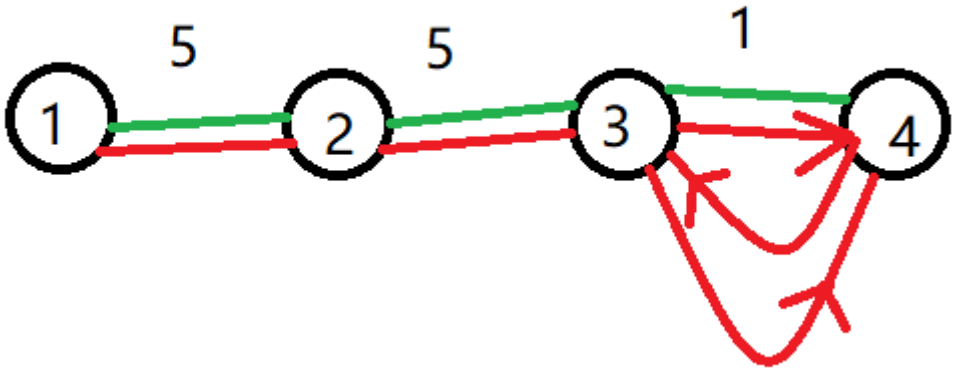
```
    while(the stack is not empty){
        pop the vertex from the stack.
        print vertex
    }
}
```



the case that one vertex is not in the shortest path



the case that vertices in the second shortest path are all in the shortest path, but some vertices in the shortest path are not included.



The case the last vertex of last vertex of a certain vertex is itself.

Other Algorithms

The algorithm for heap:

```
decrease(heap, index, NewDistance){
    vertex = vertex of index
    while(vertex is not the root of the min-heap){
        p= parent of index
        if(NewDistance < distance of index ){
            assign the information of p to the index.
            index = p.
        }
        else break.
    }
    Let vertex be the vertex of index, and NewDistance be the distance
of vertex.
}

increase(heap, index, NewDistance){
    vertex = vertex of the index

    while(index has children){
        child = the child of index whose distance is the smallest
        if(NewDistance > distance of child){
             assign the information of child to the index
             index = child
        }
        else break.
    }
```

```
    Let vertex be the vertex of index, and NewDistance be the distance
of vertex.
}

insert(heap h, vertex, distance){
    increase the size of heap by 1.
    let the last element have vertex as its vertex, distance as its
distance.

    calling function “decrease” to let it go up
}

Minimum(heap){
    if heap has no vertex, return FALSE.
    return the vertex and distance of top element.
    LastLeaf = the last element in the heap.
    decrease the size of heap by 1.
    assign the LastLeaf to the top element.
    calling function “increase” to let it down.
}
```

Chapter 3: Testing Results

All inputs are also stored in txt file in the folder “document”. You should copy the data there to test the program. Inputs here just for read.

TestCase 1	case purpose
	sample
	expected result
	240 1 2 4 5
	actual behavior
	240 1 2 4 5
	possible cause
	current status
	pass
	Input
	5 6

	1 2 50 2 3 100 2 4 150 3 4 130 3 5 70 4 5 40
TestCase 2	case purpose
	the case that vertices in the second shortest path are all in the shortest path, but some vertices in the shortest path are not included.
	expected result
	8 1 4
	actual behavior
	8 1 4
	possible cause
	current status
	pass
	Input
	4 4 1 2 6 1 4 8 2 3 5 2 4 1
TestCase 3	case purpose
	The case the last vertex of last vertex of a certain vertex in the second shortest path is itself.
	expected result
	14 1 2 3 2 4 5
	actual behavior
	14 1 2 3 2 4 5

	possible cause
	current status
	pass
	input
	5 5 1 2 6 2 3 1 2 4 2 2 5 10 4 5 4
TestCase 4	case purpose
	Minimum size
	expected result
	9 1 2 1 2
	actual behavior
	9 1 2 1 2
	possible cause
	current status
	pass
	input
	2 1 1 2 3
TestCase 5	case purpose
	the case that one vertex is not in the shortest path
	expected result
	7 1 4 5
	actual behavior
	7 1 4 5
	possible cause

	current status
	pass
	input
	5 6 1 2 1 1 4 3 2 3 2 2 4 4 3 5 3 4 5 4
TestCase 6	case purpose
	Larger Size
	expected result
	271 1 7 1 20
	actual behavior
	271 1 7 1 20
	possible cause
	current status
	pass
	Input
	20 87 1 2 268 1 7 64 1 8 2982 1 9 2325 1 10 3595 1 12 3249 1 15 1573

	1 17 2210 1 20 143 2 3 1190 2 4 3265 2 5 3459 2 6 1767 2 7 2413 2 11 620 2 14 4003 2 15 2802 2 18 2069 2 19 4060 2 20 2899 3 4 1408 3 6 2622 3 7 1311 3 13 174 3 14 4029 3 15 4460 3 16 3225 3 20 4922 4 5 3549 4 9 3920 4 10 3541 4 12 3637 4 18 560 5 7 3056 5 9 207 5 12 1216

	5 13 411 5 14 2393 5 15 131 5 16 3143 5 17 4734 5 20 3079 6 7 4109 6 10 4544 6 13 525 6 17 2376 6 18 1839 6 19 2795 6 20 1130 7 8 1488 7 9 882 7 13 887 7 14 3136 7 17 1850 7 18 2703 8 10 1913 8 13 4768 8 14 2745 8 15 3214 8 18 1751 8 20 891 9 11 581 9 12 4115 9 19 2800 10 14 2282

	10 18 1856 10 19 1341 10 20 1943 11 19 3613 12 13 3368 12 14 3994 12 15 3390 12 16 4304 12 17 2919 12 19 2441 13 16 731 13 18 4167 13 20 629 14 15 4034 14 17 4475 14 19 3299 14 20 1077 15 16 2699 15 18 1122 15 20 3792 16 18 174 16 19 140
TestCase 7	case purpose
	Larger Size
	expected result
	410 1 8 23 8 28 40
	actual behavior
	410 1 8 23 8 28 40
	possible cause

	current status
	pass
	input
	40 381 1 2 3494 1 3 3362 1 8 19 1 12 1263 1 14 3086 1 15 2690 1 20 1717 1 23 4151 1 24 2576 1 26 4244 1 31 369 1 36 3459 1 37 2113 1 38 2761 1 39 1468 2 3 2431 2 4 1822 2 6 362 2 7 3191 2 8 215 2 9 3078 2 10 4999 2 11 790 2 12 3271 2 13 1142

	2 18 355 2 25 3831 2 27 2300 2 28 3643 2 29 818 2 31 914 2 32 1731 2 33 4758 2 37 786 2 39 2674 3 4 3590 3 6 954 3 7 1211 3 9 1856 3 10 3925 3 13 3725 3 14 834 3 16 1470 3 18 3926 3 19 3589 3 20 159 3 21 4786 3 22 827 3 25 4688 3 26 2241 3 27 1030 3 29 1123 3 31 4854 3 34 4261

	3 35 3214 3 37 3017 3 39 4269 4 5 3496 4 6 2802 4 9 1036 4 11 1533 4 12 4156 4 15 3737 4 21 2294 4 27 4506 4 28 4676 4 29 4787 4 31 2705 4 32 866 4 33 3565 4 34 2956 4 37 922 4 39 4005 5 6 982 5 9 2147 5 12 3799 5 15 4332 5 17 3142 5 21 3954 5 27 2943 5 29 1517 5 31 4248 5 32 1231

	5 35 4232 5 38 512 5 39 4863 5 40 1187 6 8 1071 6 9 207 6 13 3007 6 15 4315 6 16 1977 6 20 4340 6 21 2908 6 27 4808 6 28 293 6 30 793 6 32 4333 6 33 2126 6 34 1685 6 36 4080 6 37 2890 6 39 2029 6 40 107 7 8 2191 7 9 2800 7 10 4590 7 11 3365 7 12 1691 7 13 940 7 16 3433 7 17 3583

	7 19 863 7 21 4631 7 22 2227 7 23 4905 7 24 3381 7 26 296 7 28 3212 7 31 2717 7 36 3427 7 38 1367 7 40 2799 8 9 2749 8 12 1200 8 14 1070 8 16 2544 8 19 1830 8 20 4803 8 22 1528 8 23 1 8 24 1629 8 25 4712 8 28 117 8 30 4843 8 31 2485 8 32 2443 8 33 1254 8 34 2589 8 38 665 8 40 2416

	9 13 3845 9 14 3874 9 15 2949 9 18 4833 9 19 1217 9 20 747 9 22 823 9 25 4295 9 31 1353 9 33 4007 9 36 3120 9 37 186 10 11 415 10 16 3222 10 20 2165 10 21 647 10 22 3957 10 24 2149 10 25 4057 10 26 4067 10 27 3050 10 28 1379 10 31 4319 11 12 4433 11 13 4875 11 14 2558 11 16 3314 11 17 2780 11 18 51

	11 21 2901 11 23 356 11 24 4743 11 25 1881 11 27 3424 11 28 805 11 29 2211 11 31 4524 11 34 2621 11 35 1766 11 36 332 11 38 4494 11 40 4345 12 13 4969 12 14 1051 12 15 1531 12 22 4705 12 23 3119 12 26 3530 12 27 2556 12 28 4044 12 29 4966 12 30 524 12 32 4017 12 33 1065 12 38 2358 12 40 4125 13 14 2794 13 16 2055

	13 18 4228 13 19 2318 13 20 4021 13 23 4139 13 24 1381 13 25 739 13 26 3104 13 27 4259 13 28 2327 13 30 4266 13 32 1531 13 34 544 13 38 2407 13 39 4449 13 40 80 14 16 44 14 17 253 14 19 4390 14 24 4477 14 25 3213 14 26 2761 14 27 3535 14 28 2273 14 32 757 14 33 3980 14 37 4192 14 39 3705 14 40 1278 15 16 776

	15 18 4852 15 20 2529 15 22 2251 15 23 3093 15 24 1323 15 26 3185 15 27 678 15 28 1793 15 34 3574 15 36 4997 15 37 59 15 39 2767 16 18 1535 16 19 3066 16 24 4837 16 27 4149 16 29 411 16 37 2432 16 38 1055 16 39 3018 16 40 3232 17 19 2474 17 21 1896 17 23 2386 17 25 2931 17 31 4373 17 32 2993 17 34 156 17 35 1205

	17 36 3022 17 37 3004 17 38 1947 17 40 1669 18 24 1696 18 26 3640 18 28 1733 18 29 3342 18 30 1376 18 32 1830 18 33 2148 18 35 1835 18 36 3808 18 37 3435 18 39 4547 19 21 1866 19 23 3032 19 28 2102 19 29 501 19 34 3342 19 39 3598 19 40 4951 20 21 1462 20 23 1183 20 25 2857 20 26 86 20 28 3679 20 29 1024 20 31 3256

	20 32 4474 20 34 2381 20 35 2093 20 37 1512 20 39 4673 20 40 3488 21 22 249 21 25 2445 21 27 2423 21 28 4412 21 29 2258 21 30 4107 21 32 755 21 34 758 21 36 1320 21 38 962 21 40 4177 22 25 1912 22 28 4026 22 29 271 22 30 451 22 35 2914 22 37 2490 22 40 4681 23 24 1009 23 25 2182 23 26 4956 23 28 1499 23 29 1584

	23 30 4925 23 31 4771 23 37 1742 23 40 518 24 28 570 24 34 1435 24 36 84 24 37 3058 24 39 3136 25 28 2155 25 33 4863 25 40 3398 26 27 3418 26 28 480 26 30 1333 26 33 3278 26 34 2722 26 36 775 26 37 3734 26 38 100 26 39 516 27 29 4390 27 31 634 27 33 264 27 34 3126 27 35 3240 27 38 4076 28 29 540 28 34 2278

	28 37 382 28 38 840 28 39 1979 28 40 272 29 30 3333 29 31 1049 29 32 2315 29 33 825 29 35 4879 29 39 2167 30 32 4516 30 35 1209 30 36 4116 30 38 862 30 39 1038 31 36 2696 31 37 807 31 39 1483 31 40 2651 32 33 1748 32 34 4640 32 35 4828 32 36 1098 32 38 523 32 39 2861 32 40 4579 33 34 3464 33 35 4697 34 35 3074

	34 36 1122 34 37 3923 34 38 2749 34 40 1241 35 36 2423 35 37 1009 35 38 4844 38 40 3005
TestCase 8	case purpose
	MAXSIZE(v=1000,e=5041)
	expected result
	2564 1 796 892 173 552 792 234 364 234 364 108 392 782 1000
	actual behavior
	2564 1 796 892 173 552 792 234 364 234 364 108 392 782 1000
	possible cause
	current status
	pass
	input
	Too much to show. The input data are put in the text file “t8” in folder “document”

Chapter 4: Analysis and Comments

Using the detailed pseudo code to analyze.

Time Complexity:

Analyze:

In function main:

Normal sequential execution statements take O(1) time.

Function CreateGraph, CreateHeap(O(1)), ShortestPath,

SecondShortestPath and PrintPath are called.

Therefore, T(main) = T(CreateGraph) + T(ShortestPath) + T(SecondShortestPath) + PrintPath

The following is the time complexity analysis of the functions mentioned above.

In function CreateGraph:

Normal sequential execution statements take O(1) time.

In while statement:

```
while (e--)
{
    scanf("%d%d%d",&va,&vb,&w);
    va--;
    vb--;
    g->AdjList[va]=CreateNode(vb,w,g->AdjList[va]);//Let the new node be the first element of the ADJ list.
    g->AdjList[vb]=CreateNode(va,w,g->AdjList[vb]);
}
```

Every statement in while loop takes O(1) time.

Therefore, the T(CreateGraph) = O(e).

In function ShortestPath:

Normal sequential execution statements take O(1) time.

In outer while loop:

it takes O(e) time to visit every edge, and to insert and delete them (by a heap), O(log v) time is taken (for each edge). Therefore, the while loop takes O(e* log v) time

Hence, T(ShortestPath)= O(e*log v)

In function SecondShortestPath:

The function has a similar structure with function ShortestPath, and they have the same time complexity O(e* log v), but it may have a larger constant for every edge will be inserted and deleted twice, one for shortest path, the other one for the second shortest path.

Hence, T(SecondShortestPath)= O(e*log v)

In function PrintPath:

Normal sequential execution statements take O(1) time.

Let L = the length of the second shortest path.

Clearly in the first while loop, statements will be executed O(L+1) times, the same as the second while loop.

Therefore, T(PrintPath) = O(L)

Conclusion:

Above all, T = T(main) = O(e) + $O(e \log v)$+ $O(e \log v)$ +O(L)

= $O(e \log v)$+O(L) =O $O(e \log v)$ + O(e+2) = $O(e \log v)$+O(e)= $O(e \log v)$

(

L be the length of the second shortest path,

e be the number of edges,

v be the number of vertices.

)

Space Complexity:

Analyze:

In function main:

graph g created, taking O(e) space

heap h created, taking O(e) space

stack s created, taking O(e) space

6 arrays created, taking O(v) space

And in function ShortestPath, array “include” is created taking another O(v) space.

Other local variables take O(1) space

And there’s no recursion in the program.

Above all, S = S(main) = O(e) + O(v) = O(e + v)

(e is the number of edges, v is the number of vertices)

Comments:

Possible Improvement:

1)

put the vertices and distances together forming a struct.

I regret that I didn't put the vertices field and distance field of structure heap into a smaller struct, and it is a little inconvenient when I pop and insert the element into the heap, for I need two parameters for every function, and I have to use pointers to get the result instead of "return".

2)

make the names of variable more understandable.

Following the tradition, I named the variable "distance" as "distance", which save the shortest distance from a vertex to the source, but when you also use the second shortest path, it may be confusing for still using this word, because you don't know which "distance" it is.

It is the same as variable "path" and "second_path". And I worried that the "reverse_dst" is also hard to understand.

Appendix: Source Code (in C)

```c
//TheSecondShortestPath.c

/*header files*/
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

/*Macro Definition*/
#define New(type,n) (type*)malloc(sizeof(type)*n) //Alloc a new space w
ith specific size and use.
#define NewArray(type,n) (type*)calloc(n,sizeof(type)) //Alloc a new sp
ace served as an array with specific size and use.
#define NULLPOS -
1 //which implies a variable is not existing, often used in stack, queu
e, heap and other structures.
#define INFINITY INT_MAX //Used to initialize the distance of vertices
from the start point.
#define ROOT 1 //point to the minimum element of the heap.
#define Parent(i) (i/2) //the parent of the node i in the heap
#define Left(i) (2*i) //the left child of the node i in the heap
#define Right(i) (Left(i)+1) //the right child of the node i in the hea
p
```

```c
#define TRUE 1 //Used to tell the status of a function or a variable, t
he same as FALSE
#define FALSE 0
#define BEGIN -2 //Used to simulate the last vertex of the start vertex

typedef int element; //used for elements in the stack, which are vertex
 numbers

typedef struct node // used in ADJ linked list as a node
{
    int vertex;
    int weight;
    struct node* next;
}*list;

typedef struct g{ //The structure of the whole graph.
    int v; //The number of the vertices in graph
    int e; //The number of edges in graph
    list* AdjList; // ADJ linked list, with edges storing in it.
}*graph;

typedef struct h //The structure of a heap
{
    int capacity; //the max number of elements a heap can hold.
    int* vertices; //store the vertices number for every node. vertices
[0] = the number of elements in the heap.
    int* distance; //store the distance of each vertex.
}*heap;

typedef struct s //The structure of a stack
{
    int top; //stack top
    int capacity; //the max number of elements a heap can hold.
    element* container; //store the elements in the stack, that is, a v
ertex number.
}*stack;

/* Function: Top_k
 * Used to get the k_th element away
 * from the stack top.
 *
 * k: the k_th number, 0 is on the top.
 * s: the stack need to provide
 */
```

```
int Top_k(int k,stack s);

/* Function: Minimum
 * Used to get top element of the heap.
 *
 * h: heap structure need to provide
 *
 * return_dst: provide the pointer,
 * and the function will put the
 * distance of the element in it.
 *
 * return_vertex:provide the pointer,
 * and the function will put the
 * vertex number of the element in it.
 *
 * If there's no element in the heap,
 * it will return FALSE;
 * else, return TRUE.
 */
int Minimum(heap h,int* return_dst,int* return_vertex);

/* Function: ShortestPath
 * Used to calculate the shortest
 * path in the graph and its path
 * given a source and a destination.
 * with Dijkstra Algorithm.
 *
 * h: heap you need to provide
 *
 * g: graph you need to provide
 *
 * dst: the function will save the shortest
 * distances of each vertex from the source
 * in it, please make sure it points
 * to a address in the memory with enough
 * space.
 *
 * source: start point
 *
 * dest: end point
 *
 * path: the function will save the shortest
 * paths of each vertex from the source
 * in it, please make sure it points
```

```
 * to a address in the memory with enough
 * space.
 */
int ShortestPath(heap h,graph g,int dst[],int source,int dest,int path[
]);

/* Function: ShortestPath
 * Used to calculate the second shortest
 * path in the graph and its path
 * given a source and a destination
 * with Dijkstra Algorithm.
 *
 * h: heap you need to provide
 *
 * g: graph you need to provide
 *
 * dst: the function will save the shortest
 * distances of each vertex from the source
 * in it, please make sure it points
 * to a address in the memory with enough
 * space.
 *
 * source: start point
 *
 * dest: end point
 *
 * dst_second: the function will save the
 * second shortest distances of each vertex
 * from the source in it, please make sure
 * it points to a address in the memory with
 * enough space.
 *
 * path: the function will save the shortest
 * paths of each vertex from the source
 * in it, please make sure it points
 * to a address in the memory with enough
 * space.
 *
 * sec_path: the function will save the second
 * shortest paths of each vertex from the source
 * in it, please make sure it points
 * to a address in the memory with enough
 * space.
 */
```

```
int SecondShortestPath(heap h,graph g,int source,int dest,int dst[],int
 dst_second[],int path[],int sec_path[]);

/* Function: ShortestPath
 * Used to make the heap empty.
 * h: the heap you need to provide.
 */
void dispose(heap h);

/* Function: exchange
 * Swap the values of two
 * integer.
 * a: operand 1
 * b: operand 2
 */
void exchange(int* a,int* b);

/* Function: push
 * push the element in
 * the stack.
 *
 * x: the element needed to
 * be provide, often being a
 * vertex number.
 */
void push(stack s,element x);

/* Function: insert
 * insert an element in heap
 *
 * h: heap
 * vertex: the vertex field of the element put in
 * dst: the distance field of the element put in
 */
void insert(heap h,int vertex,int dst);

/* Function: increase
 * increase the distance of the element in
 * heap, and let in sink in the heap, so that
 * we matain a min-heap's property
 *
 * h: heap
 * vertex: the index of the element to increase
 * NewDistance: refresh the distance value
```

```
 */
void increase(heap h,int index,int NewDistance);

/* Function: decrease
 * decrease the distance of the element in
 * heap, and let it get up from the heap, so that
 * we matain a min-heap's property
 *
 * h: heap
 * vertex: the index of the element to increase
 * NewDistance: refresh the distance value
 */
void decrease(heap h,int index,int NewDistance);

/* Function: SetSource
 * Often used before a Dijkstra Algorithm, to
 * Initialize the distance and path arrays given
 * a start point.
 *
 * v: the number of vertices
 * source: the start point
 * dst: distance array
 * path: path array
 */
void SetSource(int v,int source,int dst[],int path[]);

/* Function: PrintPath
 * print the second shortest path
 *
 * s: stack used in this procedure
 *
 * end: the end point
 *
 * short_path: the length of the shortest path
 *
 * dst: the shortest distances of vertices
 * away from the source (start point)
 *
 * reverse_dst: the shortest distances of vertices
 * away from the end (end point)
 *
 * path: the shortest path for source(basically, source=0)
 * reverse_path: the shortest path end(basically, end=v-1)
 * sec_path: the second shortest path for source
```

```
 */
void PrintPath(stack s,int end,int short_path,int dst[],int reverse_dst
[],int path[],int reverse_path[],int sec_path[]);

/* Function: push
 * pop the element from
 * the stack.
 *
 * return the element popped.
 */
element pop(stack s);

/* Function: push
 * Create a heap.
 * capacity: the maximum size of the heap
 * return the heap variable.
 */
heap CreateHeap(int capacity);

/* Function: push
 * Create a graph
 * v: the number of vertices
 * e: the number of edges
 * return the graph created.
 */
graph CreateGraph(int v,int e);

/* Function: CreateStack
 * Create a stack.
 * capacity: the maximum size of the stack
 * return the stack
 */
stack CreateStack(int capacity);

/* Function: CreateStack
 * Create a node.
 */
list CreateNode(int vertex,int weight,list next);


int main(){
    int v,e;
    scanf("%d%d",&v,&e);
    graph g=CreateGraph(v,e);
    heap h=CreateHeap(4*e+1);
```

```
    int* path=New(int,v);
    int* reverse_path=New(int,v);//the shortest path from end point, na
mely v-1.
    int* sec_path=New(int,v);
    int* dst=New(int,v);
    int* reverse_dst=New(int,v);//the shortest distances away from end
point, namely v-1.
    int* sec_dst=New(int,v);
    assert(path&&reverse_dst&&reverse_path&&sec_path&&dst&&reverse_dst&
&sec_dst);//assert all space alloced are valid.

    int shortest_path=ShortestPath(h,g,reverse_dst,v-1,0,reverse_path);
    int sec_shortest_path=SecondShortestPath(h,g,0,v-
1,dst,sec_dst,path,sec_path);

    printf("%d",sec_shortest_path);

    stack s=CreateStack(e+3);
    PrintPath(s,v-
1,shortest_path,dst,reverse_dst,path,reverse_path,sec_path);

    return 0;
}

int Top_k(int k,stack s){
    if(s->top-1-k<0) return NULLPOS;//the number of stack is too few.
    return s->container[s->top-1-k];
}

void PrintPath(stack s,int end,int short_path,int dst[],int reverse_dst
[],int path[],int reverse_path[],int sec_path[]){
    /* Since the path[] only
     * store the last vertex,
     * to print the path, we
     * need a stack to invert
     * the order we print.
     */
    int vertex=end;//start from end point.
    int backed=FALSE;//if backed=TRUE, we can straightly head to the so
urce with shortest path.
    int tmp;

    while (vertex!=BEGIN){
```

```
        if(
            //the case that the vertex is not in the shortest path.
            dst[vertex]+reverse_dst[vertex]!=short_path ||
            //the case that vertices in the shortest path between the v
ertex and last vertex are skipped.
            (tmp=Top_k(0,s))!=NULLPOS && tmp !=reverse_path[vertex] ||
            //the case that the last vertex of last vertex is the same
as the current vertex.
            Top_k(1,s) == vertex
        )
            backed=TRUE;//the detour has finished.

        push(s,vertex);

        if(backed){
            vertex=path[vertex];//go straight in the shoretest path.
        }
        else{
            vertex=sec_path[vertex];
        }

    }

    //print the vertices in the path.
    while((vertex=pop(s))!=NULLPOS){
        printf(" %d",vertex+1);
    }
}

void dispose(heap h){
    h->vertices[0]=0;
}

void push(stack s,element x){
    assert(s->top<=s->capacity);
    s->container[s->top++]=x;
}

element pop(stack s){
    if(!(s->top)) return NULLPOS;
    return s->container[--(s->top)];
}
```

```
void exchange(int* a,int* b){
    int tmp;
    tmp=*a;
    *a=*b;
    *b=tmp;
}

void SetSource(int v,int source,int dst[],int path[]){
    int i;
    for(i=0;i<v;i++){
        dst[i]=INFINITY;
        path[i]=NULLPOS;
    }

    dst[source]=0;
    path[source]=BEGIN;
}

list CreateNode(int vertex,int weight,list next){
    list p = New(struct node,1);
    assert(p);
    p->vertex=vertex;
    p->weight=weight;
    p->next=next;
    return p;
}

graph CreateGraph(int v,int e){
    graph g = New(struct g,1);
    assert(g);
    g->e=e;
    g->v=v;
    g->AdjList=NewArray(list,v);
    assert(g->AdjList);

    int va,vb,w;

    while (e--)
    {
        scanf("%d%d%d",&va,&vb,&w);
        va-
-;//Because the C array begins from 0, we substact every vertex by 1 to
 save the space.
        vb--;
```

```
        g->AdjList[va]=CreateNode(vb,w,g->AdjList[va]);//Let the new no
de be the first element of the ADJ list.
        g->AdjList[vb]=CreateNode(va,w,g->AdjList[vb]);
    }
    return g;
}

heap CreateHeap(int capacity){
    heap h = New(struct h,1);
    assert(h);
    h->capacity=capacity;

    h->vertices=New(int,capacity+1);
    assert(h->vertices);

    h->distance=NewArray(int,capacity+1);
    assert(h->distance);

    h->vertices[0]=0;// h->vertices[0] = the number of elements in the
heap.
    return h;
}


void decrease(heap h,int index,int NewDistance){//we make the distances
 the key field of elements in the heap.
    int p;
    int vertex = h->vertices[index];
    while(index > ROOT){
        p=Parent(index);
        if(NewDistance<h->distance[p]){
            h->vertices[index]=h->vertices[p];
            h->distance[index]=h->distance[p];
        }
        else break;
        index = p;
    }
    h->vertices[index]=vertex;
    h->distance[index]=NewDistance;
}

void increase(heap h,int index,int NewDistance){
    int child;
    int vertex = h->vertices[index];
    int LastLeaf=h->vertices[0];
```

```
    while(index <= Parent(LastLeaf)){
        child = Left(index);
        if(child!=LastLeaf && h->distance[child]>h->distance[Right(inde
x)])
            child=Right(index);

        if(NewDistance>h->distance[child]){
            h->vertices[index]=h->vertices[child];
            h->distance[index]=h->distance[child];
        }
        else break;
        index=child;
    }
    h->vertices[index]=vertex;
    h->distance[index]=NewDistance;
}

void insert(heap h,int vertex,int dst){
    int LastLeaf = ++(h->vertices[0]);
    assert(LastLeaf<=h->capacity);

    h->vertices[LastLeaf]=vertex;
    decrease(h,LastLeaf,dst);
}

int Minimum(heap h,int* return_dst,int* return_vertex){
    if(h->vertices[0]==0) return FALSE;//There's no element in the heap
.
    * return_vertex = h->vertices[ROOT];
    * return_dst = h->distance[ROOT];

    int LastLeaf = h->vertices[0]--;
    h->vertices[ROOT]=h->vertices[LastLeaf];

    increase(h,ROOT,h->distance[LastLeaf]);
    return TRUE;
}

int SecondShortestPath(heap h,graph g,int source,int dest,int dst[],int
 dst_second[],int path[],int sec_path[]){
    int v=g->v;
    int vertex,distance,NewDistance,neighbor,weight,NewVertex;
```

```
    dispose(h);

    SetSource(v,source,dst,path);
    SetSource(v,source,dst_second,sec_path);

    /* The second shortest distance from source
     * to source need to be initialized as INFINITY,
     * otherwise the algorithm won't work because the
     * if-statement in line 497.
     */
    dst_second[source]=INFINITY;

    insert(h,source,dst[source]);

    list p;

    while(Minimum(h,&distance,&vertex)){
        if(distance <= dst_second[vertex]){
            p=g->AdjList[vertex];
            while(p){
                neighbor=p->vertex;
                NewDistance=p->weight+distance;
                NewVertex=vertex;

                if(NewDistance < dst[neighbor]){
                    /* if the original distance is larger,
                     * it can be used for the second
                     * shortest distance.Therefore, we use
                     * a "exchange" to assign the NewDistance
                     * with original shortest distance for the
                     * next if-statement.
                     */
                    exchange(&dst[neighbor],&NewDistance);
                    exchange(&NewVertex,&path[neighbor]);
                    insert(h,neighbor,dst[neighbor]);
                }

                if(NewDistance > dst[neighbor] && NewDistance < dst_sec
ond[neighbor]){
                    dst_second[neighbor]=NewDistance;
                    sec_path[neighbor]=NewVertex;
                    //The second shortest distance should also be put i
nto the heap.
                    insert(h,neighbor,dst_second[neighbor]);
```

```
                }

                p=p->next;
            }
        }
    }
    return dst_second[dest];

}

stack CreateStack(int capacity){
    stack s=New(struct s,1);
    assert(s);
    s->top=0;
    s->capacity=capacity;
    s->container=New(element,capacity);
    assert(s->container);
    return s;
}

int ShortestPath(heap h,graph g,int dst[],int source,int dest,int path[
]){
    int vertex = source;
    int neighbor;
    int distance;
    int v=g->v;
    list p = NULL;

    int *include=New(int,v);//The array used to tell if an element has
been popped from the heap.

    assert(include);

    SetSource(v,source,dst,path);
    memset(include,FALSE,sizeof(int)*v);//All elements are not included
 at the beginning.

    dispose(h);

    insert(h,source,dst[source]);

    while(Minimum(h,&distance,&vertex)){
        if(!include[vertex]){
            include[vertex]=TRUE;
```

```
            p=g->AdjList[vertex];
            while(p){
                neighbor = p->vertex;

                if(!include[neighbor]){

                    if(dst[neighbor]>distance+p->weight){
                        dst[neighbor]=distance+p->weight;
                        path[neighbor]=vertex;
                        insert(h,neighbor,dst[neighbor]);
                    }
                }
                p=p->next;
            }
        }
    }

    free(include);
    return dst[dest];
}
```

Declaration

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