The 2nd-shortest Path

Project3 Hard Report

name

November 29, 2022

Project3 Hard

November 30, 2022

Abstract

The 2nd-shortest Path

Chapter 1:Introduction

Given a directed graph,find the second shortest path from the start point to the end point in this directed graph, output the corresponding path length and the corresponding path line,The second-shortest path is the shortest path whose length is longer than the shortest path(s) , 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, and the second-shortest path can backtrack,so that it can use the same road more than once.

Chapter 2:Algorithm Specification

Step1:Store the directed graph

In this project,I used adjacency list to store the information of Graph,Compared with the adjacency matrix, it can save more memory space, The relevant statement is as follows:

```
struct VertexPath {
    Vertex Adjv;
    //the number of the current point on the path
    struct VertexPath* Next;
    //point to the number of next point in this path
};

struct AdjVNode {
    Vertex AdjV;    //the number of this point
    PtrToAdjVNode Next; //point to next point that connect with the header point
    Length Value;
    int Know;   //Determine if the node is visited
    struct VertexPath* path1;
    //The shortest path connecting this node with a linked list
    struct VertexPath* path2;
    //The next-shortest path connecting this node with a linked list
};

typedef struct Vnode {
    PtrToAdjVNode FirstEdge;
} AdjList[MaxVertexNum];

typedef struct GNode* PtrToGNode;
struct GNode {
    int Nv; //the number of points
    int Ne; //the number of edges
    AdjList G;  //Adjacent table
};

typedef PtrToGNode LGraph;
```

Figure 1: The Declaration for Graph

The meaning of each content has been explained in the notes.

Step2:Find the path

In my project i use Dijkstra algorithm's revise to find the second shortest path,we use dist1 to store the shortest path and dist2 to store the second shortest path,Before the algorithm starts, except the dist1 of the starting point, the dist1 and dist2 of the other points are initialized to positive infinity, in each loop we find the smallest dist1's corresponding node(v1),Traverse its adjacent points,if

(1):dist1[v1]+value(v1->vk) less than dist1[vk],then changed vk's reference information, if the old dist1[k] less then the old dist2[k](According to the algorithm, dist2 may be assigned first, but dist1 is still positive and infinite, so this problem should be avoided), then we changed dist2[k] to dist1[k],and record the path(The change of the path is carried out by the change of the pointer),the changed for dist2[k] will The second shortest path affecting its critical point,then we let k into queue_dist2[] to change its adjacent points. Then we changed the old dist1[k],and and record the path。

(2):dist1[v1]+value(v1->vk) more than dist1[k] but more than dist2[k],then we should change dist2,And it is the same as the method of updating dist2 in (1)

(3):If the queue_dist2[] is not empty,then we should get a vertex from it,then Check and update the adjacent points of this point. If its adjacent points are updated, the changed points must also be enqueued. In the process of modifying dist2, there is no restriction on whether the point has been visited, so the problem of repeating edges can be solved.

Step3:Use Min-Heap to find dist1

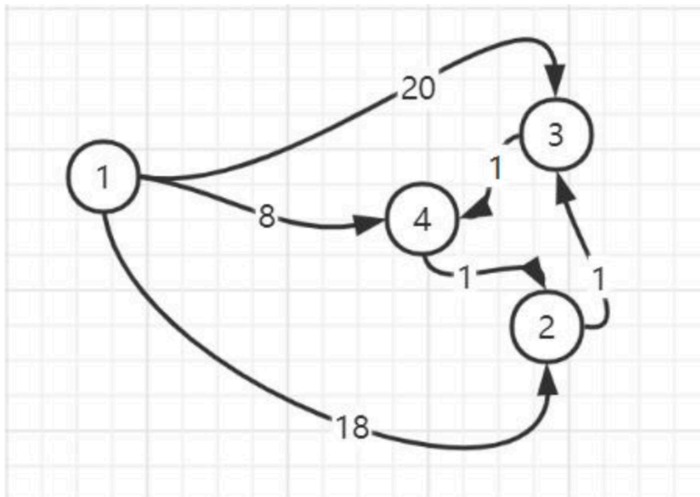
In order to obtain the smallest dist1 more efficiently,i use the min-heap to store dist1,and adjust the heap every time dist1 is modified.

Step4:Print the Path

Because I store the path by a linked list in 'struct VertexPath* path2',then use it we can find the second-shortest path easily.

Chapter 3:Testing Results

Test1:



Graph of test case 1

Figure 2: test1

The output is:11 1 4 2 3 4

The purpose of this test case is to test the correctness of the algorithm when there exists one small loop in the graph. From the graph, we can obviously find that the shortest path from 1 to 4 is 1-4 directly. However, when it comes to the second-shortest path, because the path from 1 to 3 and the path from 1 to 2 are too long and the loop 4-2-3-4 is short, so the second-shortest path from 1 to 4 is 1-4-2-3-4.

My program can deal with this situation coreectly.

Test2:

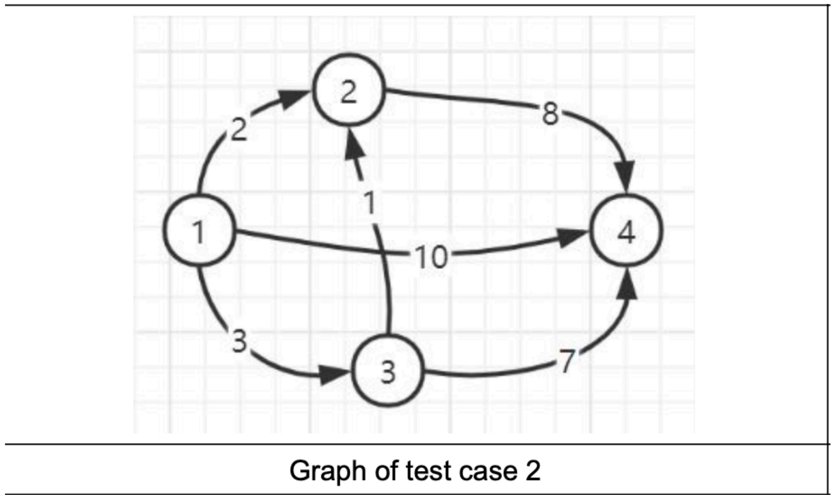


Figure 3: test1

The output is:12 1 3 2 4

In this case there are more than one shortest path,from 1 to 4,and my programm find the second-shortest coreectly.

Test3:

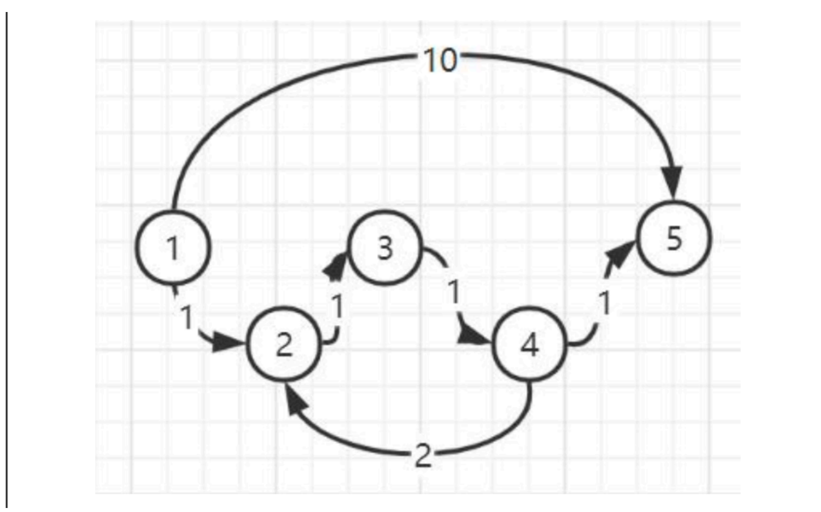


Figure 4: test1

The output is:8 1 2 3 4 2 3 4 5

In this case there are 'backtrack' edges in this answer,such as '2->3','3->4', in this case my program can solve it coreectly.

Test4:

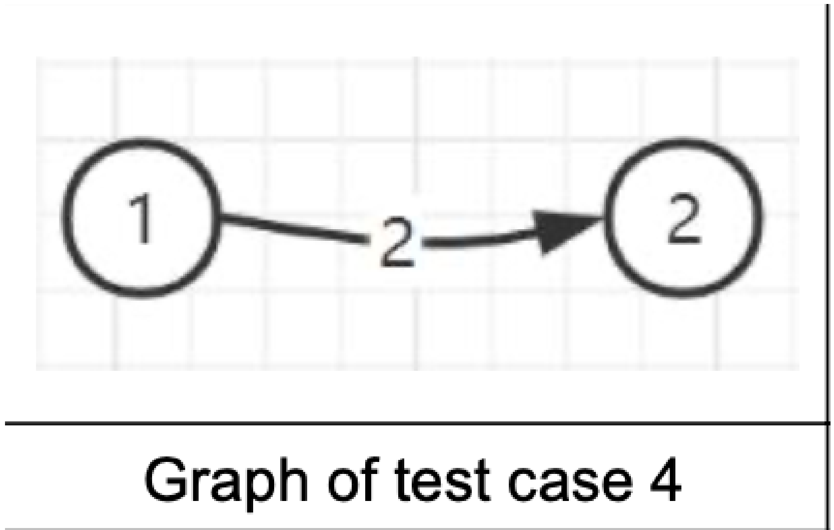


Figure 5: test1

The output is:No the second shortest path

In this case there is no the second-shortest path,my program can deal with it coreectly.

Test5:

```
1000 1000
1 2 1387
2 3 2589
3 4 3511
1 5 3120
1 6 3465
1 7 3571
3 8 3702
2 9 2997
7 10 3226
6 11 2407
6 12 4902
1 13 4557
10 14 1732
2 15 1454
```

Figure 6: test1

The output is:19567 1 6 11 277 531 582 759 993 1000

In this case my program can deal with the problem that has many nodes.

Test6:

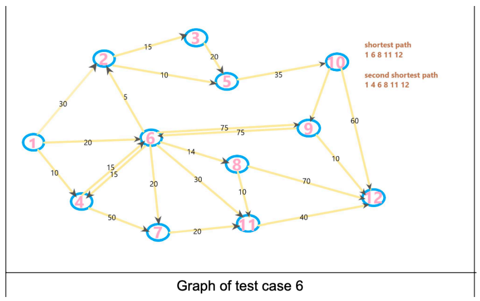


Figure 7: test1

The output is:89 1 4 6 8 11 12

In this case there are multiple roads between two vertexs,my programm can deal with it coreectly.

Chaper 4:Analysis and Comments

Space Complexity:

In my project I use Adjacency list to store this Graph,so the Space Complexity is O(V+E).

Time complexity:

The operations of heap:

To make a heap,because the initialize value of each points except Source are infinite,so we should not do something else, the time complexity of it is O(1).

The operations of DeleteMin,and Change dist1 is O(logV),Using the method of filtering upwards, this is the basic operation of the heap.

Find the second shortest path:

The best case: In a best case,if we almost not change dist2 in Dijkstra loop,then we only need to end the program after traversing all the points, the time complexity is O(V*logV).

The worst case: In a worst case,we should change dist2 for lots of times,when doing a change for dist2,our time complexity is O(V), and the change for dist2 happens many times,so the time complexity is O($2 * V * E + V * logV$).If V » E,the time complexity is O(V*logV),if E » V, the time complexity is O(V*E).

Print the Path:

Because i store the path in struct node by pointer path2,so we can get road easily by O(V).

Appendix:Source (in C)

You can see the Source code at the end of Report.

Declaration

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

```
//Dijkstra.h


#ifndef _DIJKSTRA_H_
#define _DIJKSTRA_H_

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

#define MaxVertexNum 999999
#define MaxPathNum 999999
#define ZERO 0
#define INFINITY 99999999

typedef int Vertex;
typedef int Length;
typedef struct AdjVNode* PtrToAdjVNode;
typedef struct VertexPath VertexPath;


struct VertexPath {
    Vertex Adjv;
    //the number of the current point on the path
    struct VertexPath* Next;
    //point to the number of next point in this path
};

struct AdjVNode {
    Vertex AdjV;    //the number of this point
    PtrToAdjVNode Next; //point to next point that connect with the header
point
    Length Value;
    int Know;   //Determine if the node is visited
    struct VertexPath* path1;
    //The shortest path connecting this node with a linked list
    struct VertexPath* path2;
    //The next-shortest path connecting this node with a linked list
};

typedef struct Vnode {
    PtrToAdjVNode FirstEdge;
} AdjList[MaxVertexNum];

typedef struct GNode* PtrToGNode;
struct GNode {
    int Nv; //the number of points
    int Ne; //the number of edges
    AdjList G;  //Adjacent table
};

typedef PtrToGNode LGraph;
```

```
typedef struct heap {
    int heap[6000];     //Use the heap to store each point
    int size;   //the size of heap
    int heapIndex[6000];    //Match the number of the point to the
subscript in the heap
} Heap;


extern int dist1[MaxVertexNum];     //store the shortest path length for
each point
extern int dist2[MaxVertexNum];     //store the second shortest path
length for each point


void InitialDist(); //initial dist1[] and dist2[]
void MakePath(LGraph Graph);   //the function to find the second shortest
path
LGraph ReadGraph();     //read the information of this Graph

Heap* MakeHeap(int s,int N);
//Make a heap to store the dist1
void ChangeHeap(Heap* DistHeap, int p, int dist);
//if the dist1 of p has been changed then Change the heap
int DeleteMin(Heap* DistHeap);
//Delete the min dist1 in the heap


#endif
```

```
//Dijkstra.c

#include <stdio.h>
#include <stdlib.h>
#include "Dijkstra.h"

int dist1[MaxVertexNum];    //store the shortest path length for each
point
int dist2[MaxVertexNum];    //store the second shortest path length for
each point



/**
 * @brief Initial the value of dist1 and dist2
 * the dist except 1 is initialized to positive infinity
 */
void InitialDist() {
    for (int i=0; i<MaxVertexNum; i++) {
        dist1[i] = INFINITY;
        dist2[i] = INFINITY;
    }
```

```
    dist1[1] = 0;
}



/**
 * @brief Make a heap to store dist1
 *
 * @param s the sourse of road
 * @param N the number of all Vertexes
 * @return Heap*
 */
Heap* MakeHeap(int s,int N)
{
    Heap* DistHeap = (Heap*)malloc(sizeof(Heap));
    DistHeap->size = N;
    DistHeap->heap[0] = 0;  //dummy index of heap
    DistHeap->heapIndex[0] = 0;
    dist1[0] = -1;
    DistHeap->heap[1] = s;  //Initial the root of heap as 's'
    DistHeap->heap[s] = 1;
    int heap_point = 2;
    /**
     * @brief
     * store other Vertex
     */
    for (int i=1; i<=N; i++) {
        if(i!=s) {
            DistHeap->heap[heap_point] = i;
            DistHeap->heapIndex[i] = heap_point;
            heap_point++;
        }
    }


    return DistHeap;
}

/**
 * @brief
 * if the dist1 of p has been changed then Change the heap
 * @param DistHeap
 * @param p the index that has been changed its dist1
 * @param dist the changed dist
 */
void ChangeHeap(Heap* DistHeap, int p, int dist) {
    int i = DistHeap->heapIndex[p];
    /**
     * @brief
     * Basic heap operations PercolateUp
     */
    for(; dist1[DistHeap->heap[i/2]] > dist; i/=2) {
        DistHeap->heap[i] = DistHeap->heap[i/2];
        DistHeap->heapIndex[DistHeap->heap[i]] = i;
```

```
    }

    DistHeap->heap[i] = p;
    DistHeap->heapIndex[p] = i;
}

/**
 * @brief get the root of heap then adjust the heap
 *
 * @param DistHeap
 * @return int
 */
int DeleteMin(Heap* DistHeap) {
    int res = DistHeap->heap[1];

    int tempIndex = DistHeap->heap[DistHeap->size--];
    int tempValue = dist1[tempIndex];

    int parent,child;
    /**
     * @brief
     * Basic heap operations PercolateDown
     */
    for (parent=1; 2*parent<=DistHeap->size ; parent=child) {
        child = 2*parent;
        if(child+1<=DistHeap->size) {
            if(dist1[DistHeap->heap[child]] > dist1[DistHeap-
>heap[child+1]]) {
                child++;
            }
        }

        if(dist1[DistHeap->heap[child]] > tempValue) {
            break;
        }
        else {
            DistHeap->heap[parent] = DistHeap->heap[child];
            DistHeap->heapIndex[DistHeap->heap[parent]] = parent;
            //Change the index of the node in the heap
        }
    }

    DistHeap->heap[parent] = tempIndex;
    DistHeap->heapIndex[tempIndex] = parent;
    //Change the index of the tempIndex in the heap

    return res;
}

/**
 * @brief
 * The function to Solve the sub-shortest path problem
 * @param Graph
 */
```

```
void MakePath(LGraph Graph) {
    Heap* DistHeap = MakeHeap(1,Graph->Nv); //Make a heap to store dist1
    //Use queue to store the Vertex that dist2 has been changed
    Vertex queue_dist2[3000];
    int front = 0;
    int rear = 0;
    for (int i=0; i<Graph->Nv; i++) {
        Vertex start = DeleteMin(DistHeap); //get the shortest dist1's
index
        if(start == -1) {
            break;
        }
        PtrToAdjVNode tempV = Graph->G[start].FirstEdge;    //get this
node
        Graph->G[start].FirstEdge->Know = 1;    //markd this node as know

            //Modify the dist1 and dist2 of the adjacent points of the
recorded vertices
            tempV = Graph->G[start].FirstEdge;  //Used to traverse the
critical table
            while(tempV->Next != NULL) {
                //changed the dist1
                if(dist1[start]+tempV->Next->Value < dist1[tempV->Next-
>AdjV]) {

                    //use dist2 to store the dist1 that will be changed
                    if(dist2[tempV->Next->AdjV] > dist1[tempV->Next-
>AdjV]) {
                        dist2[tempV->Next->AdjV] = dist1[tempV->Next-
>AdjV];    //use dist2 to store dist1
                        int change_dist2 = dist2[tempV->Next->AdjV];
                        Graph->G[tempV->Next->AdjV].FirstEdge->path2 =
Graph->G[tempV->Next->AdjV].FirstEdge->path1;
                        //Enqueue the changed node of dist2 for subsequent
updates
                        queue_dist2[rear++] = tempV->Next->AdjV;
                    }

                    dist1[tempV->Next->AdjV] = dist1[start]+tempV->Next-
>Value;
                    //Adjust in heap if dist1 is changed
                    ChangeHeap(DistHeap,tempV->Next->AdjV,dist1[tempV-
>Next->AdjV]);
                    //Record the updated path
                    Graph->G[tempV->Next->AdjV].FirstEdge->path1 = Graph-
>G[start].FirstEdge->path1;
                    VertexPath* prePath =
(VertexPath*)malloc(sizeof(VertexPath));
                    prePath->Adjv = start;
                    prePath->Next = Graph->G[start].FirstEdge->path1;
                    Graph->G[tempV->Next->AdjV].FirstEdge->path1 =
prePath;

                }
```

```
                //If the updated dist1 can only make changes to dist2 then
do the following
                else if(dist1[start]+tempV->Next->Value>dist1[tempV->Next-
>AdjV] && dist1[start]+tempV->Next->Value<dist2[tempV->Next->AdjV]) {
                    dist2[tempV->Next->AdjV] = dist1[start]+tempV->Next-
>Value;
                    //Record the updated path
                    Graph->G[tempV->Next->AdjV].FirstEdge->path2 = Graph-
>G[start].FirstEdge->path1;
                    VertexPath* prePath =
(VertexPath*)malloc(sizeof(VertexPath));
                    prePath->Adjv = start;
                    prePath->Next =  Graph->G[tempV->Next-
>AdjV].FirstEdge->path2;
                    Graph->G[tempV->Next->AdjV].FirstEdge->path2 =
prePath;
                   //Enqueue the changed node of dist2 for subsequent
updates
                   queue_dist2[rear++] = tempV->Next->AdjV;


                }
                tempV = tempV->Next;
            }

            while(front<rear) {
                int changeVertex = queue_dist2[front++];
                    int change_dist2 = dist2[changeVertex];
                    PtrToAdjVNode changeV = Graph-
>G[changeVertex].FirstEdge;
                    PtrToAdjVNode temp_changeV = changeV;
                    /**
                     * @brief
                     * Update dist2 in the same way as in the previous
case
                     */
                    while(temp_changeV->Next!=NULL) {
                    if(change_dist2+temp_changeV->Next->Value<
                        dist2[temp_changeV->Next->AdjV]) {
                                dist2[temp_changeV->Next->AdjV] =
change_dist2+temp_changeV->Next->Value;
                                Graph->G[temp_changeV->Next-
>AdjV].FirstEdge->path2 = Graph->G[changeV->AdjV].FirstEdge->path2;
                                VertexPath* prePath =
(VertexPath*)malloc(sizeof(VertexPath));
                                prePath->Adjv = changeV->AdjV;
                                prePath->Next =  Graph->G[temp_changeV-
>Next->AdjV].FirstEdge->path2;
                                Graph->G[temp_changeV->Next-
>AdjV].FirstEdge->path2 = prePath;
                                queue_dist2[rear++] = temp_changeV->Next-
>AdjV;
                            }
                            temp_changeV = temp_changeV->Next;
```

```
                            }
                    }
                }

        }




/**
 * @brief
 * Read in the information of the graph
 * @return LGraph
 */
LGraph ReadGraph()
{
    int v1,v2,length;
    LGraph Graph = (struct GNode*)malloc(sizeof(struct GNode));
    //make a Graph
    scanf("%d %d",&Graph->Nv,&Graph->Ne);
    //read the number of vertexes and edges
    /**
     * @brief
     * Create critical table
     * @param i
     */
    for (int i=1; i<=Graph->Nv; i++) {
        Graph->G[i].FirstEdge = (struct AdjVNode*)malloc(sizeof(struct
AdjVNode));
        Graph->G[i].FirstEdge->AdjV = i;
        Graph->G[i].FirstEdge->Next = NULL;
        Graph->G[i].FirstEdge->Value = 0;

    }

    /**
     * @brief
     * Edges are read in and stored in the adjacency table
     * @param i
     */
    for (int i=0; i<Graph->Ne; i++) {
        scanf("%d %d %d",&v1,&v2,&length);
        PtrToAdjVNode new_road = (struct AdjVNode*)malloc(sizeof(struct
AdjVNode));
        new_road->AdjV = v2;
        new_road->Value = length;
        new_road->Next = Graph->G[v1].FirstEdge->Next;

        Graph->G[v1].FirstEdge->Next = new_road;



    }
```

```
    return Graph;
}
```