```
//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;
}
```