```
/*********************************************************************************
**
 * Filename: ass3.cpp
 * Name & Student No.: Yanhong Ben, 4845675
 * ASS No: 3
 * File Description: practice and improve Dijkstra's algorithm
 *********************************************************************************
**/

#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;


const int MAXSIZE = 50; //no more than 50 nodes in the graph

int countNodes = 0; //count how many nodes
int countEdges = 0; //count how many edges
int sourceNode = 0; //store the source node
int destNode = 0; //store the destination node

struct Node
{
    int index; //store the index of the nodes
    double nodeX; //store the x-coordinate of the nodes
    double nodeY; //store the y-coordinate of the nodes
};

struct Triples
{
    int fromNode; //store the start node of each edge
    int toNode; //store the end node of each edge
    double edgeCost; //store the edge cost of each edge
};

struct Node nodes[MAXSIZE];
struct Triples* trips; //dynamic array to store the edge

double graphic[MAXSIZE][MAXSIZE]; //temp dimensional array for Dijkstra algorith
m
double d[MAXSIZE]; //temp array for Dijkstra algorithm
double p[MAXSIZE]; //temp array for Dijkstra algorithm

void readFile()
{
    ifstream fin;
    char filename[20];
    cout << "Please input filename: ";
    cin.getline(filename, 20, '\n');
    fin.open(filename);
    while (!fin.good())
    {
       cout << "Wrong file name. Please try again: ";
       cin.getline(filename, 20, '\n');
       fin.open(filename);
    }
    // fin.open("ass03.txt");
    fin >> countNodes;
    if(fin.good())
    {
        for (int i=0; i<countNodes; i++)
        {
            fin >> nodes[i].index >> nodes[i].nodeX >> nodes[i].nodeY;
        }
        fin >> countEdges;
```

-1 Edge list representation
-0.1 magic nos

8.9/10

Well done!

Edge list
representation.
-1

```
        trips = new Triples[countEdges];
        for (int i=0; i<countEdges; i++)
        {
            fin >> trips[i].fromNode >> trips[i].toNode >> trips[i].edgeCost;
        }
    }
    fin >> sourceNode >> destNode;
    fin.close();
}


//================== Heap functions ==================
void swap(double &a, double &b)
{
    double temp;
    temp = a;
    a = b;
    b = temp;
}

void swap(int& id1, int& id2)
{
    int temp;
    temp = id1;
    id1 = id2;
    id2 = temp;
}

void sift_up(double *heap, int* index, int i)
{
    if (i == 0)
        return;
    int p = (i-1) / 2;
    if (heap[p] <= heap[i])
        return;
    else
    {
        swap(heap[i], heap[p]);
        swap(index[i], index[p]);
        sift_up(heap, index, p);
    }
}

void sift_down(double *heap, int *index, int i, int size)
{
    int c = i * 2 + 1;
    if (c < size-1)
        if (heap[c] > heap[c+1])
            c++;
    if (c <= size-1)
    {
        if (heap[i] > heap[c])
        {
            swap(heap[i], heap[c]);
            swap(index[i], index[c]);
            sift_down(heap, index, c, size);
        }
    }
    else
        return;
}

//================ Run ========================
void initialize()
{
    //initialize grahic array with -1 which is big enough to avoid sift down and
```

```
 up by heap sort
    for (int row=0; row<countNodes; row++)
    {
        for (int col=0; col<countNodes; col++)
            graphic[row][col] = 999;
    }

    //initialize graphic array with data read from the file
    int i = 0;
    while (i < countEdges)
    {
        graphic[trips[i].toNode−1][trips[i].fromNode−1] = trips[i].edgeCost;
        graphic[trips[i].fromNode−1][trips[i].toNode−1] = trips[i].edgeCost;
        i++;
    }

    for (int i=0; i<countNodes; i++)
    {
        d[i] = 0;
        p[i] = 0;
    }
}

void solution1()
{
    initialize();

    //set a boolean array to flag if node has been visited
    bool visited[countNodes];
    for (int i=0; i<countNodes; i++) //initialize visited array
        visited[i] = false;

    //set index array to identify the distance
    int index[countNodes];
    for (int i=0; i<countNodes; i++)
        index[i] = i;

    //marked source node to be visited
    visited[sourceNode−1] = true;

    //for i=1 to n do
    double copy_d[countNodes]; //make a copy d array to do heap sort to prevent
data change in d array
    for (int i=0; i<countNodes; i++)
    {
        d[i] = graphic[sourceNode−1][i]; //d[i] = L[source, i]
        copy_d[i] = graphic[sourceNode−1][i];
        sift_up(copy_d, index, i); //inset d array and index array into a heap a
nd sorted
        p[i] = sourceNode − 1; //p[i] = source
    }

    //repeat while destination is not visited
    int v = 0;
    int w = 0;
    int countNotSelected = countNodes;
    while (v != destNode − 1)
    {
        //v = the index of the minimum d[v] not yet selected
        v = index[0];
        //throw notifcation there is no path between the nodes
        if (d[v] == 999)
        {
            cout << "No edge between nodes." << endl;
            return;
        }
```

```
        //mark v as visited
        visited[v] = true;

        //for each w not visited do
        if(v != destNode - 1)
        {
            copy_d[0] = copy_d[countNotSelected-1];
            index[0] = index[countNotSelected-1];
            sift_down(copy_d, index, 0, countNotSelected);
            countNotSelected--;

            for (w=0; w<countNodes; w++)
            {
                if (visited[w] == false)
                {
                    if (d[w] > d[v] + graphic[v][w])
                    {
                        d[w] = d[v] + graphic[v][w];
                        int i = 0;
                        while (i < countNodes && index[i] != w) //find the w in
the index array
                            i++;
                        copy_d[i] = d[v] + graphic[w][v]; //change copy array as
 well
                        sift_up(copy_d, index, i);
                        p[w] = v;
                    }
                }
            }
        }
    }

    // get the path from the path array
    int shortestPath[countNodes];
    int k = 0;
    int j = destNode-1;
    while (j != sourceNode-1)
    {
        shortestPath[k] = j;
        j = p[j];
        k++;
    }
    shortestPath[k] = sourceNode - 1;

    cout << endl;
    cout << "============================== Result ==============================" <<
endl;
    cout << "The length of the shortest path from solution 1: " << d[destNode - 1] << endl;
    cout << "The Path from solution 1: ";
    for (int i=k; i>0; i--)
        cout << shortestPath[i]+1 << "-->";
    cout << shortestPath[0]+1;
    cout << endl;
    cout << "The number of additional nodes in the solution tree for solution 1: " << endl;
    for (int i=0; i<=k; i++)
        visited[shortestPath[i]] = false;
    for (int i=0; i<countNodes; i++)
        if (visited[i])
            cout << i + 1 << ' ';
    cout << endl;
    cout << "------------------------------------------------------------------" <
< endl;
}

//function for a* algorithm to count the distance between two nodes
```

```
double straightValue(double x1, double x2, double y1, double y2)
{
   return sqrt(pow((x1-x2),2) + pow((y2-y1),2));
}

void solution2()
{
    initialize();

    //store estimate of the remaining distance
    double estimate[countNodes];
    for (int i=0; i<countNodes; i++)
    {
        estimate[i] = straightValue(nodes[i].nodeX, nodes[destNode-1].nodeX, nod
es[i].nodeY, nodes[destNode-1].nodeY);
    }

    //set a boolean array to flag if node has been visited
    bool visited[countNodes];
    for (int i=0; i<countNodes; i++)
        visited[i] = false;

    //set index array to identify the distance
    int index[countNodes];
    for (int i=0; i<countNodes; i++)
        index[i] = i;

    //mark source node as visited
    visited[sourceNode-1] = true;

    //for i=1 to n do
    double aStar[countNodes];
    for (int i=0; i<countNodes; i++)
    {
        d[i] = graphic[sourceNode-1][i]; //d[i] = L[source, i]
        aStar[i] = d[i] + estimate[i];
        sift_up(aStar, index, i); //inset d array and index array into a heap an
d sorted
        p[i] = sourceNode - 1; //p[i] = source
    }

    //repeat while destination is not visited
    int v = 0;
    int w = 0;
    int countNotSelected = countNodes;
    while (v != destNode - 1)
    {
        //v = the index of the minimum d[v] not yet selected
        v = index[0];
        //throw notifcation there is no path between the nodes
        if (d[v] == 999)
        {
            cout << "No edge between nodes." << endl;
            return;
        }

        //mark v as visited
        visited[v] = true;

        //for each w hasn't been visited do
        if(v != destNode - 1)
        {
            aStar[0] = aStar[countNotSelected-1];
            index[0] = index[countNotSelected-1];
            sift_down(aStar, index, 0, countNotSelected);
            countNotSelected--;
```

```
            for (w=0; w<countNodes; w++)
            {
                if (visited[w] == false)
                {
                    if (d[w] > d[v] + graphic[v][w])
                    {
                        d[w] = d[v] + graphic[v][w];
                        int i = 0;
                        while (i < countNodes && index[i] != w) //find w in the
index array
                            i++;
                        aStar[i] = d[v] + graphic[w][v] + estimate[w]; //change
star array value as weel
                        sift_up(aStar, index, i);
                        p[w] = v;
                    }
                }
            }
        }
    }

    // get the path from the path array
    int shortestPath[countNodes];
    int k = 0;
    int j = destNode-1;
    while (j != sourceNode-1)
    {
        shortestPath[k] = j;
        j = p[j];
        k++;
    }
    shortestPath[k] = sourceNode - 1;

    cout << "The length of the shortest path from solution 2: " << d[destNode - 1]<< endl;
    cout << "The Path from solution 2: ";
    for (int i=k; i>0; i--)
        cout << shortestPath[i]+1 << "-->";
    cout << shortestPath[0]+1;
    cout << endl;
    cout << "The number of additional nodes in the solution tree for solution 2: " << endl;
    for (int i=0; i<=k; i++)
        visited[shortestPath[i]] = false;
    for (int i=0; i<countNodes; i++)
        if (visited[i])
            cout << i + 1 << ' ';
    cout << endl;
    cout << "-----------------------------------------------------------------" <
< endl;
}

void solution3()
{
    initialize();

    double d2[MAXSIZE] = {0};
    double p2[MAXSIZE] = {0};
    //set two boolean arrays to flag if nodes have been visited
    bool visited[countNodes];
    for (int i=0; i<countNodes; i++) //initialize
        visited[i] = false;

    bool visited2[countNodes];
    for (int i=0; i<countNodes; i++) //initialize
        visited2[i] = false;
```

```
    //set two index arrays to identify the distance
    int index[countNodes];
    for (int i=0; i<countNodes; i++)
        index[i] = i;

    int index2[countNodes];
    for (int i=0; i<countNodes; i++)
        index2[i] = i;

    //marked source and destination node to be visited
    visited[sourceNode-1] = true;
    visited2[destNode-1] = true;

    //for i=1 to n do
    double copy_d[countNodes]; //make two copy d arrays to do heap sort to preve
nt data change in d array
    double copy_d2[countNodes];

    for (int i=0; i<countNodes; i++)
    {
        d[i] = graphic[sourceNode-1][i]; //d[i] = L[source, i]
        d2[i] = graphic[destNode-1][i];
        copy_d[i] = graphic[sourceNode-1][i];
        copy_d2[i] = graphic[destNode-1][i];
        sift_up(copy_d, index, i); //inset d array and index array into a heap a
nd sorted
        sift_up(copy_d2, index2, i); //inset d2 array and index2 array into a he
ap and sorted
        p[i] = sourceNode - 1; //p[i] = source
        p2[i] = destNode - 1; //p[i] = destination
    }

    //repeat while v not visited in the other flag array is not visited
    int v = 0;
    int v2 = 0;
    int w = 0;
    int stopNode = 0;
    int countNotSelected = countNodes;
    int countNotSelected2 = countNodes;
    while (visited[v2] == false && visited2[v] == false)
    {
        //v = the index of the minimum d[v] not yet selected
        v = index[0];
        v2 = index2[0];
        //throw notifcation there is no path between the nodes
        if (d[v] == 999 || d2[v2] == 999)
        {
            cout << "No edge between nodes." << endl;
            return;
        }
        //mark v as visited
        visited[v] = true;
        visited2[v2] = true;

        //for each w not visited do
        if(visited[v2] == false && visited2[v] == false)
        {
            copy_d[0] = copy_d[countNotSelected-1];
            copy_d2[0] = copy_d2[countNotSelected2-1];
            index[0] = index[countNotSelected-1];
            index2[0] = index2[countNotSelected2-1];
            sift_down(copy_d, index, 0, countNotSelected);
            sift_down(copy_d2, index2, 0, countNotSelected2);
            countNotSelected--;
            countNotSelected2--;
```

Magic nos -0.1

```
            for (w=0; w<countNodes; w++)
            {
                if (visited[w] == false)
                {
                    if (d[w] > d[v] + graphic[v][w])
                    {
                        d[w] = d[v] + graphic[v][w];
                        int i = 0;
                        while (i < countNodes && index[i] != w) //find the w in
the index array
                            i++;
                        copy_d[i] = d[v] + graphic[w][v]; //change copy array as
 well
                        sift_up(copy_d, index, i);
                        p[w] = v;
                    }
                }
                if (visited2[w] == false)
                {
                    if (d2[w] > d2[v2] + graphic[v2][w])
                    {
                        d2[w] = d2[v2] + graphic[v2][w];
                        int i = 0;
                        while (i < countNodes && index2[i] != w) //find the w in
 the index array
                            i++;
                        copy_d2[i] = d2[v2] + graphic[w][v2]; //change copy arra
y as well
                        sift_up(copy_d2, index2, i);
                        p2[w] = v2;
                    }
                }
            }
        }
        else
            stopNode = (visited2[v] == true ? v : v2);
    }

    // get the path from the path array
    int shortestPath[countNodes];
    int k = 0;
    int j = stopNode;
    while (j != sourceNode-1)
    {
        visited[j] = false;
        shortestPath[k] = j;
        j = p[j];
        k++;
    }
    shortestPath[k] = sourceNode - 1;

    cout << "The length of the shortest path from solution 3: " << d[stopNode] + d2[stopNode] <<
endl;
    cout << "The Path from solution 3: ";
    for (int i=k; i>0; i--)
        cout << shortestPath[i]+1 << "-->";
    j = stopNode;
    while (j != destNode-1)
    {
        visited2[j] = false;
        cout << j+1 << "-->";
        j = p2[j];
    }
    visited[sourceNode-1] = false;
    visited2[destNode-1] = false;
    cout << destNode;
```

```
    cout << endl;
    cout << "The number of additional nodes in the solution tree for solution 3: " << endl;
    for(int i=0; i<countNodes; i++)
        if(visited[i] == true || visited2[i] == true)
            cout << i+1 << ' ';
    cout << endl;
    cout << endl;
}

int main()
{
    readFile();
    solution1();
    solution2();
    solution3();
    delete []trips;
    return 0;
}
```

```
*******************************************************************************
* Filename: ass3.txt
* Name & Student No.: Yanhong Ben, 4845675
* ASS No: 3
* File Description: discuss the three algorithms used in this assignment
*******************************************************************************

//brief discussion of my result
As assignment 3 required, three solutions were displayed.

Solution 1 is a Dijkstra’s algorithm for a specific start and end point. It is d
ifferent from standard Dijkstra’s algorithm since it suppose to find the shortes
t path between start node and destination node instead of start from the first n
ode. So once algorithm reached the destination node it will stop. It finds out t
he shortest path which length is 8.5 and path is 19−>4−>8−>16. However, it visit
ed most of the nodes in the graphic to find out the correct path.

Solution 2 is known as a A* algorithm, which is improved based on Dijkstra’s alg
orithm. This algorithm, I implemented it to find out the linear distance from al
l nodes to the destination node(16) and take it as the estimate of distance. It
finds out the length of shortest path and the path are the same as the result in
 solution 1. But it only visited 5,20 nodes excluding the shortest path. Pretty
much more efficient than solution 1.

Solution 3 is a proposed improved algorithm by assignment. It begins both the so
urce and the destination nodes together, and stops when one node is visited in t
he other process. It comes up with which is actually the second length of the sh
ortest path(8.6), the path is 19−>4−>20−>5−>16. The additional nodes it visited
is 1,3,8 and 15. It concludes a different answer from above two algorithms. 8.6
is not the shortest path.

//relative efficiency of each of the three proposed approaches
In solution 1, process compares and sorts out the actual edge cost, while soluti
on 2 compares and sorts out the actual cost plus a estimate linear distance whic
h accurate the real distance between nodes. By using solution 2, it makes the di
fference between each options more clearer and makes the algorithm much quicker
to find out the correct path. The more accurate estimate distance is the more ef
ficient solution 2 is.

Compare solution 1 and solution 3, definitely solution 3 is quicker than solutio
n 1, since solution 3 start from both source and destination nodes. However, it
comes up with the wrong answer, but the path is still a quiet short path just ma
y not be the shortest one.

//any problem that may arise each of them
For solution 1, DijkstraM−bM−^@M−^Ys algorithm does a blind search and spends a
lot of time wasting on the unnecessary nodes scanning.
For solution 2, since it calculate and take the linear distance between two node
s as a estimate data to sort, if the search is not about linear distance but som
ething like price, the algorithm need to be adjusted accordingly.
For solution 3, like in this assignment, it may make the process more efficient,
 but it will comes up with the wrong answer. Because it stops immediately when a
 node was found like here node 20 which is visited in the other process, but it
might be a possibility a better solution is yet behind.
```

```
testing 11 to 15
Please input filename:
============================== Result ==============================
The length of the shortest path from solution 1: 15.9
The Path from solution 1: 11-->1-->19-->4-->8-->16-->15
The number of additional nodes in the solution tree for solution 1:
2 3 5 6 7 9 10 12 13 14 17 18 20
--------------------------------------------------------------------
The length of the shortest path from solution 2: 15.9
The Path from solution 2: 11-->1-->19-->4-->8-->16-->15
The number of additional nodes in the solution tree for solution 2:
2 3 5 7 9 10 12 13 17 18 20
--------------------------------------------------------------------
The length of the shortest path from solution 3: 15.9
The Path from solution 3: 11-->1-->19-->4-->8-->16-->15
The number of additional nodes in the solution tree for solution 3:
2 5 6 7 9 10 12 14 18 20
```