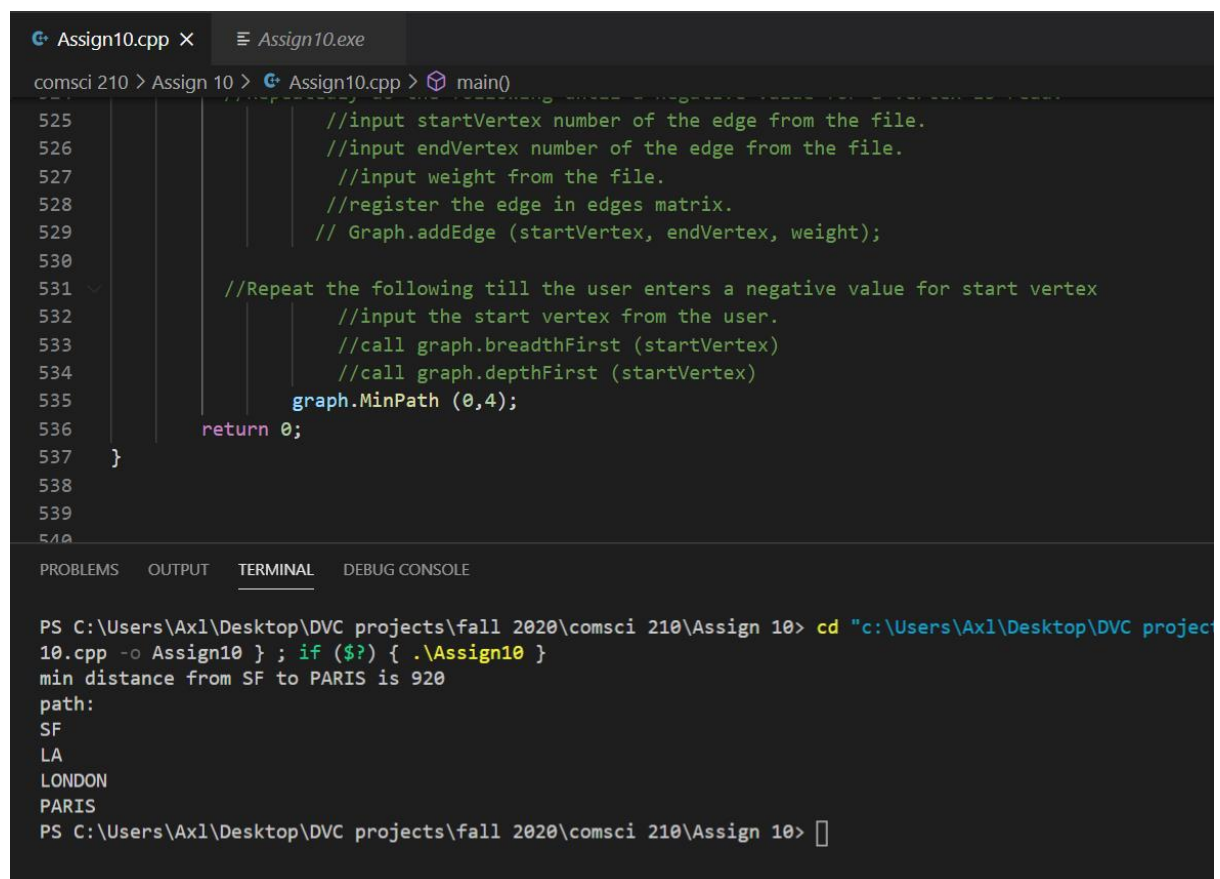


So the first thing the function does it create a graph object using the previously defined class. Afterwards it creates an array of strings, a variable to hold the vertex number as well as a string variable to hold the vertex name. The program then uses separate function types (one for the item and one for the number) and puts it in a queue to hold their value. It then starts using the graph function to add in all the values from the item and int queue into a graph, all while making sure that their edge and vertexes are within the right order. Afterwards, it starts inputting all the edges using the addEdge function using the data from the same file. It then arranges the graph as well as displays the output in a min path format using the function MinPath. This is done by iterating through the graph, putting down a mark after each node iterated as well as updating the min path whenever a shorter path is discovered within the graph. In easier terms, the program basically takes the contents of a user generated file, creates a graph with it, and then allows the user to input the name of the place they would like to travel to and it then shows the minimum total distance of the path for you to be able to get to that certain place.

(I used your code professor)

SAMPLE OUTPUT FROM SF TO PARIS 0 – 4 (in MinPath1.txt)



```

Assign10.cpp X  Assign10.exe
comsci 210 > Assign 10 > Assign10.cpp > main()
525 //input startVertex number of the edge from the file.
526 //input endVertex number of the edge from the file.
527 //input weight from the file.
528 //register the edge in edges matrix.
529 // Graph.addEdge (startVertex, endVertex, weight);
530
531 //Repeat the following till the user enters a negative value for start vertex
532 //input the start vertex from the user.
533 //call graph.breadthFirst (startVertex)
534 //call graph.depthFirst (startVertex)
535 graph.MinPath (0,4);
536 return 0;
537 }
538
539
540
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
PS C:\Users\Ax1\Desktop\DVC projects\fall 2020\comsci 210\Assign 10> cd "c:\Users\Ax1\Desktop\DVC projects\fall 2020\comsci 210\Assign 10" & .\Assign10.exe
min distance from SF to PARIS is 920
path:
SF
LA
LONDON
PARIS
PS C:\Users\Ax1\Desktop\DVC projects\fall 2020\comsci 210\Assign 10>

```

```
#include <iostream>
```

```
#include <fstream>
```

```
using namespace std;
```

```
struct IntNodeType;
typedef IntNodeType * IntNodePtr;
struct IntNodeType
{
    int i; //accumulated distance from the start vertex to this vertex.
    IntNodePtr link; //Link to the next item node.
};
```

```
class IntQue
{
private:
    IntNodePtr head;
    IntNodePtr tail;
public:
    IntQue( );
    void enqueue (int i);
    void dequeue (int & i );
    int isEmpty ( );
};
IntQue :: IntQue ( )
{
    head = NULL;
    tail = NULL;
}
```

```
void IntQue :: enqueue (int i)
{
    //Prepare a node
```

```
//Create an node
IntNodePtr newPtr = new IntNodeType ();

//fill up the node
newPtr->i = i;
newPtr->link = NULL;

//link the node

//case empty queue
if (head == NULL && tail== NULL)
{
    head = newPtr;
    tail = newPtr;
}

//case end queue
else
{
    tail->link = newPtr;
    tail = newPtr;
}

}

//pre: que is NOT empty
void IntQue::deque(int & i)
{

    //fill out the rec using head.
    i = head->i;

    //Remove the node
```

```
        head = head->link;

        //Test to see if the que became empty after removal
        //In that case, make sure to set the tail to NULL
        if (head == NULL)
            tail = NULL;

    }

    int IntQue ::isEmpty( )
    {
        return (head==NULL && tail==NULL)? true: false;
    }

    struct ItemType
    {
        int dist; //accumulated distance from the start vertex to this vertex.
        int path [20]; //An array containing a list of vertices along the path
                       //from the start vertex to this vertex.
        int pathlen; //The length of the occupied part of the above array i.e.
                    //the length of the path stored in the above array.

    };

    struct NodeType;
    typedef NodeType * NodePtr;

    struct NodeType
    {
        int dist; //accumulated distance from the start vertex to this vertex.
        int path [20]; //An array containing a list of vertices along the path
                       //from the start vertex to this vertex.
        int pathlen; //The length of the occupied part of the above array i.e.
                    //the length of the path stored in the above array.
        NodePtr link; //Link to the next item node.
```

```
};
```

```
class ItemQue
```

```
{
```

```
private:
```

```
    NodePtr head;
```

```
    NodePtr tail;
```

```
public:
```

```
    ItemQue( );
```

```
    void Enque (ItemType r);
```

```
    void dequeue(ItemType & rec);
```

```
    int isEmpty ( );
```

```
    void penque (ItemType r);
```

```
};
```

```
ItemQue :: ItemQue ( )
```

```
{
```

```
    head = NULL;
```

```
    tail = NULL;
```

```
}
```

```
void ItemQue::penque(ItemType r)
```

```
{
```

```
    //Create an node
```

```
    NodePtr newPtr = new NodeType ;
```

```
    //fill up the node
```

```
    newPtr->dist = r.dist;
```

```
    int i;
```

```
        for (i=0;i<20;i++)
    {
        newPtr->path[i] = r.path[i];
    }

    newPtr->pathlen = r.pathlen;
    newPtr->link = NULL;

    //Link the entry;
    //case: empty insertion

    //case: head insertion

    //case: mid and end insertion

    //case: empty list
    if (head == NULL)
    {
        head = newPtr;
    }

    //case: head insertion
    else if (r.dist < head->dist)
    {
        newPtr->link = head;
        head = newPtr;
    }

    //case: mid and end insertion:
```

```
    else
    {
        //use the cur to locate the point of insertion
        //let prev walk behind it.
        NodePtr cur, prev;
        for (prev=head, cur=head->link; cur != NULL;
            prev=prev->link, cur=cur->link)
        {
            if (newPtr->dist < cur->dist)
                break;
        }
        //do the insertion using prev
        newPtr->link = prev->link;
        prev->link = newPtr;
    }

}

void ItemQue :: Enque (ItemType r)
{
    //Prepare a node
    //Create an node
    NodePtr newPtr = new NodeType ;
    //fill up the node
    newPtr->dist = r.dist;
    int i;
```

```
        for (i=0;i<20;i++)
        {
            newPtr->path[i] = r.path[i];
        }

        newPtr->pathlen = r.pathlen;
        newPtr->link = NULL;

        //link the node

        //case empty queue
        if (head == NULL && tail== NULL)
        {
            head = newPtr;
            tail = newPtr;
        }

        //case end queue
        else
        {
            tail->link = newPtr;
            tail = newPtr;
        }
    }

    //pre: que is NOT empty
    void ItemQue::dequeue(ItemType & rec)
    {
        //fill out the rec using head.
        rec.dist = head->dist;
        int i;
```



```
        for (i=0;i<20;i++)
    {
        rec.path[i] = head->path[i];
    }

    rec.pathlen = head->pathlen;
    //Remove the node
    head = head->link;
    //Test to see if the que became empty after removal
    //In that case, make sure to set the tail to NULL
    if (head == NULL)
        tail = NULL;

}

int ItemQue ::isEmpty( )
{
    return (head==NULL && tail==NULL)? true: false;
}

//The code below assumes IntQue and IntStack as user provided classes.
//Please write those classes first.

//write this class

class Graph
{
private:
    string vertices [20];
    int edges [20][20];
    int vSize;
```

```
public:
```

```
    Graph ( );
```

```
    void addVertex (string vertex);
```

```
    void addEdge (int fromV, int toV, int weight);
```

```
    void getAdjacent (int vertex, IntQue & adjQ);
```

```
    void MinPath (int from, int to);
```

```
};
```

```
Graph::Graph( )
```

```
{
```

```
    vSize = 0;
```

```
    for (int i=0; i<20; i++)
```

```
    {
```

```
        for (int j=0; j<20; j++)
```

```
            edges [i][j] = 0;
```

```
    }
```

```
}
```

```
void Graph::addVertex (string vertex)
```

```
{
```

```
    //add the next vertex at the next index in the array
```

```
    vertices [vSize] = vertex;
```

```
    vSize ++;
```

```
}
```

```
void Graph::addEdge (int fromV, int toV, int weight)
{
    //set to weight the appropriate element in the matrix.
    //Since it is non-directed graph.
    //Set to weight the other corresponding element to 1
    edges [fromV][toV] = weight;
    edges [toV][fromV] = weight;
}

void Graph::getAdjacent (int vertex, IntQue & adjQ)
{
    //Any vertex to which I have an edge present is my adjacent vertex.
    for (int i=0; i< vSize; i++)
    {
        if (edges[vertex][i] > 0)
            adjQ.enqueue (i);
    }
}

void Graph::MinPath (int from, int to)
{
    //ItemQue is a priority que that queus ItemType items.
    //IntQue is a regular que that queues ints.
    //It's used to store adjacent vertices.
    ItemQue mq;
    IntQue adjq;
    ItemType item;
```

```
//The following variables are used to save values from an item dequeued.
//The item dequeued is considered the parent item.
//The values saved from the working (parent) item are used
//to create next level adjacent (child) items.

//
//We save the parent vertex number.
//We will use it to pass it to method edges ( )to find each of its
//child's distance from the parent vertex.

//We obtain this from the vertex path list stored in the parent item.
//It is stored as the last vertex in the path list.
//
//We save the parent distance from the start vertex because
//we will use this value to find each child's distance from the
//start vertex.
//We will do this by adding the child vertex distance
//from the parent vertex to the parent vertex distance from the
//start vertex.

//We save the parent path length.
//We will use this value to generate the path length of each child.
//The path list of each child will be one greater than the parent.
//
//We do not save the parent path. Because, every child
//vertex path has the same path as the parent vertex except that an
//additional vertex is added to the path.

int p_vertex; //parent vertex # (i.e. the vertex just dequeued).
int p_dist; //total distance of the parent vertex from the start vertex.
```

```
int p_pathlen;//the length of the occupied part of the path array i.e.  
    //the length of path.
```

```
int adjver; //a vertex
```

```
int marked [20]; //array to keep track of which vertices are marked.
```

```
//unmark all vertices.
```

```
for (int i=0;i<20;i++)  
    marked[i] = 0;
```

```
//prepare the first item to be queued.
```

```
//Initialize the path array to be all zeros.
```

```
for (int i=0;i<20;i++)  
    item.path[i] = 0;
```

```
//set the distance of the start vertex to the start vertex to be 0.
```

```
item.dist = 0; //its distance from from start vertex.
```

```
//set the path of the start vertex to the start vertex to contain
```

```
//the start vertex.
```

```
item.path[0] = from; //first entry in the vertex array list.
```

```
//set the path length to be 1 because there is only one vertex in
```

```
//the path.
```

```
item.pathlen = 1;
```

```
//enqueue the item in the priority queue

//we enqueue this to get the algorithm started
//we will soon deque it and then enqueue its adjacents (i.e.children).

mq.penqueue (item);

//start the deque/enqueue loop

while ( ! (mq.isEmpty() ) )
{
    //Deque the item.
    mq.dequeue (item);

    //break if target is reached.
    if (item.path[item.pathlen-1] == to)
        break;

    //Save its values (call them parent values. These will be needed
    //to generate (child) next level items

    //For this purpose,
    //Save the distance of this vertex from the start vertex.
    //Save the vertex number of this vertex. This vertex is the
    //last vertex in the path list.
    //Save the path length.

    //We save them here because the variable item will be reused
```

```
//preparing a next level item.

p_dist = item.dist;
p_vertex = item.path[item.pathlen-1];
p_pathlen = item.pathlen;

//if the item is not yet marked. find the next level items.
if(marked[item.path[item.pathlen-1]] == 0 )
{
    //mark the item
    marked[item.path[item.pathlen-1]] = 1;

    //Find the next level vertices. receive them in an int que
    //Call method findAdj and pass it an int queue.
//findAdj method will return list of adjacent (child)
//vertices in the int queue passed to it.

    //findAdj (item.path[item.pathlen-1], adjq);
    getAdjacent (item.path[item.pathlen-1], adjq);

    //enqueue next level items in the priority que
    while ( !adjq.isEmpty() )
    {
        //deque a next level (i.e. child) vertex
        //the vertex number of one of the next level vertices
// will be returned in an int vertex
        adjq.deque (adjver);

        //if the next level vertex is not marked.
```

```
        //Prepare an item for it and
    //enqueue the item in the priority queue

        //Use the same item variable as above.
    //But modify it as below.

    if (marked[adjver] == 0)
    {
        //prepare an item for it by
        //reusing the item variable.

        //calculate its accumulated distance.
        item.dist = p_dist + edges [p_vertex][adjver];

        //add the vertex to the path list in the item
        item.path[p_pathlen] = adjver;

        //update the length of the used verix array list
        item.pathlen = p_pathlen + 1;

        //enqueue the item in the priority que
        mq.penqueue (item);
    }
}

}

}

//write code here to display the item
//display the target vertex distance from the start vertex.
//display the path from the start vertex to the target vertex.

cout << "min distance from " << vertices[from] << " to " << vertices[to] << " is "<< item.dist << endl;
```



```
        cout << "path:" << endl;

        int i;

        for (i=0;i<item.pathlen;i++)
        {
            cout << vertices[item.path[i]] << endl;

        }

    }

}

int main ( )
{
    //cout << "Hi" << endl;

    //Create a Graph object
    Graph graph;

    string vertexName [20];

    int verNum;

    string verName;

    //Input vertices number and names from the file.

    //Repeatedly do the following until a negative value for a vertex is read.

    //input vertex number from the file.

    //input vertex name from the file.

    ifstream fin;

    fin.open ("minpath");

    if(!fin)

        {cout <<"error opening the file, exiting the program";

return 1;}

    fin >> verNum;

    while (verNum >= 0)
```

```
{
    fin >> verName;
    //add the vertex to the graph
    graph.addVertex (verName);
    fin >> verNum;

}

int startVertex, endVertex, weight;
fin >> startVertex;

while (startVertex >= 0)
{
    fin >> endVertex;
    fin >> weight;
    //add the vertex to the graph
    //register the edge in edges matrix.
    graph.addEdge (startVertex, endVertex, weight);
    fin >> startVertex;

}

//Input vertices edges from the input file.
//Repeatedly do the following until a negative value for a vertex is read.
    //input startVertex number of the edge from the file.
    //input endVertex number of the edge from the file.
    //input weight from the file.
    //register the edge in edges matrix.
    // Graph.addEdge (startVertex, endVertex, weight);
```

```
//Repeat the following till the user enters a negative value for start vertex
    //input the start vertex from the user.
    //call graph.breadthFirst (startVertex)
    //call graph.depthFirst (startVertex)
    graph.MinPath (0,4);
return 0;
}
```