**APPENDIX 1**

**SHORTEST PATH - DIJKSTRA**

**A PROJECT REPORT**

***Submitted by***

***Submitted by***

**Abdul Ghaffar Kalhoro -194699**

**Hamad Nasir-120312**

**Muhammad Maubashirullah Durrani-202209**

**Madiha Urooj-190833**

***in partial fulfillment for the award of the degree***

***of***

**Bachelors**

**IN**

**Computer Science**

**BSCS-6C**

**NATIONAL UNIVERSITY OF SCIENCES**

**& TECHNOLOGY, SEECS**

JANUARY 2018

**APPENDIX 2**

**NATIONAL UNIVERSITY OF SCIENCES &**

**TECHNOLOGY, SEECS**

**BONAFIDE CERTIFICATE**

Certified that this project report **“…Shortest Path - Dijkstra** **……………..”** is

the bonafide work of **“……Abdul Ghaffar Kalhoro, Hamad Nasir, Muhammad**

**Mubashirullah Durrani, Madiha Urooj.…………”** who carried out the project work under my supervision.

**APPENDIX 3**

**TABLE OF CONTENTS**

**CHAPTER NO. TITLE PAGE NO.**

**ABSTRACT 5**

**1# INTRODUCTION 6**

**2# DIJKSTRA’S ALGORITHM 8**

**3# IMPLEMENTATION 11**

**4# PSUEDO CODE 12**

**5# DIJKSTRA’S –A GREEDY ALGORITHM 13**

**6# RESULT 14**

**7#** **DISADVANTAGES 15**

**8#** **IMPROVEMENTS 16**

**9# DIJKSTRA’S –APPLICATION 18**

**APPENDIX 4**

**TABLE OF CONTENTS**

**1 INTRODUCTION**

* **SHORTEST PATH**
* **ALGORITHM SELECTION**
* **INSERTION OF PAKISTAN CITIES GRAPH**

**2 DIJKSTRA’S ALGORITHM**

* **SHORTEST PATH PROBLEM**
* **DIRECTED GAPHS**
* **UNDIRECTED GRAPHS**
* **DIRECTED WEIGHTED GRAPHS**
* **UNDIRECTED WEIGHTED GRAPHS**
* **WHY DIJKSTRA’SALGORITHM**

**3 IMPLEMENTATION**

* **GRAPHS**
* **ARRAYS**
* **FILE 1/O**
* **MAPS AS HASH TABLES**
* **VECTORS**
* **OPEN GL**

**4 PSUEDO CODE**

**5 DIJKSTRA’S –A GREEDY ALGORITHM**

**6 RESULT**

* **RUNNING TIME**
  + - * + **FOR ARRAYS**
        + **FOR PRIORITY QUEUES**

**7 DISADVANTAGES**

**8 IMPROVEMENTS**

**9 DIJKSTRA’S –APPLICATION**

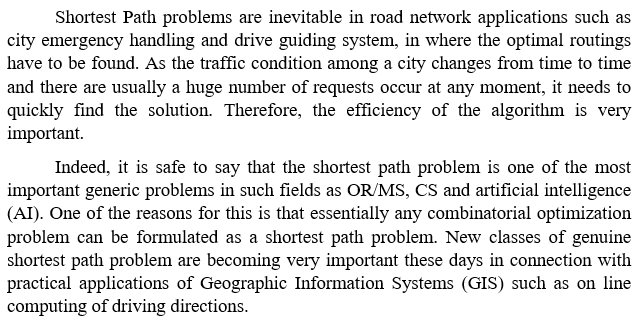
* **MAPS IN STANDARD TEMPLATE LIBRARY**
* **WHY TO USE MAPS**
* **SOURCE CODE**
* **OUTPUT**

**Abstract:**

In this project we are implementing Dijkstra algorithm with real world application of maps in which we are dealing with different cities and would prompt user to input source and destination cities and provide the shortest path shown by using Graphic User Interface. The vast use of Geographic Information Systems(GIS) has been increased rapidly since eighties and nineties. The shortest path search can be considered as their most demanding applications in search field. The most of the studies regarding to the shortest path showed that graphs are the most feasible for this purpose. Dijkstra’s Algorithm is one of the most popular algorithms in computer science. It is also popular in operations research. It is generally viewed and presented as greedy algorithm.

**Introduction:**

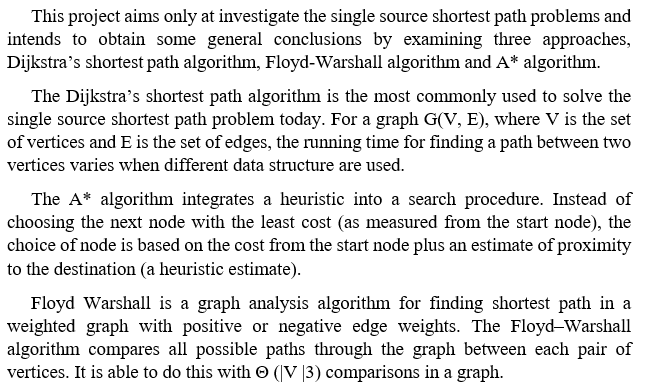
# **Shortest Path actual Problem:**



# **Algorithm Selection:**

One of major problem that we faced were the selection of algorithm from three well known algorithms:

* **Dijkstra’s Algorithm**
* **A\* Algorithm**
* **Floyd Warshall**

We selected Dijkstra’s algorithm to solve our problem, because it was very popular used algorithm and we knew little bit about its working too.

# **Insertion of the Pakistani cities graph:-**

It was the main problem that rose later on, because we have to insert Pakistani major cities graph as a map with vertices as the cities and edges distance between them. It was quite difficult to collect the information of Pakistani major highways connecting different cities with different distance, because its data was not available at internet. So we solve this problem by making text file having cities connecting to others with certain distance and utilized it as our graph.

**Dijkstra’s Algorithm:**

Edsger Dijkstra a Dutch computer scientist in 1959 created Dijkstra’s Algorithm. During his employment in Mathematical Centre, he was asked to present ARMAC an advanced computer system. During his presentation he described an efficient algorithm to determine shortest path between two points later on in honor of his the algorithm was called Dijkstra’s Algorithm.

Dijkstra's algorithm is used to calculate shortest path between two vertices or points in any graph like structure and is called the single-source shortest path. It is also known as the single source shortest path problem. It computes length of the shortest path from the source to each of the remaining vertices in the graph.

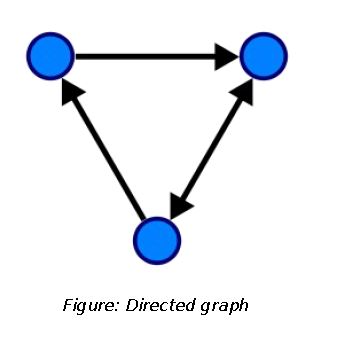
**Single source shortest path problem** is described as follows:

Let G= {V, E} be a weighted graph there is a source vertex S in a V. The problem is to find shortest path and distance of every vertex from that source vertex S.

All the weights in the graph should be non-negative and it can be both directed or un-directed graph.

Before going in detail of Dijkstra’s algorithm there are some important concepts whose understanding is necessary.

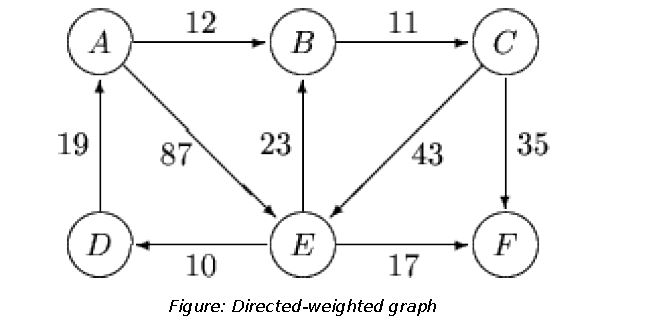
**Directed graph** is a set of vertices or nodes that are connected together, where all the edges are directional. Directed graphs are also called as digraph.

Figure: Directed graph

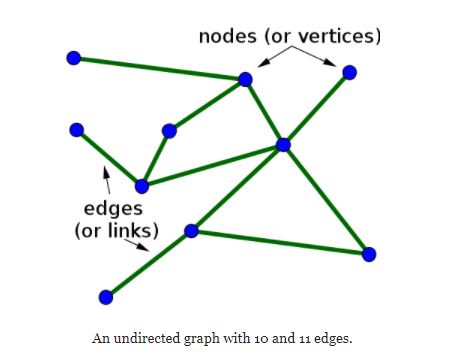
Directed-weighted graph is a graph which is directed and has a numerical value attached to each of the edges.

the edge of the graph.

Figure: Directed-weighted graph



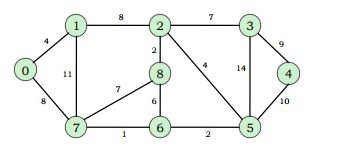
An **undirected graph** is [graph](http://mathinsight.org/definition/graph) which consists of set of vertices or nodes that are connected together and all the edges are bidirectional. An undirected graph is also called as undirected [network](http://mathinsight.org/definition/network), as illustrated in the following figure.



Undirected graph is defined as consisting of the set N of nodes and the set E of edges which are not directed.

Un-Directed weighted graph is an un-directed graph in which a numerical value

is attached to each of the edge of the graph.

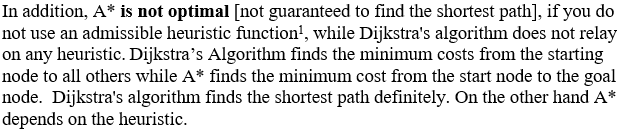


**Why Dijkstra’s algorithm:**

WeUsed Dijkstra’s Algorithm because of below points:

* **Dijkstra is uninformed algorithm** - it should be used when you have no knowledge on the graph, and cannot estimate the distance from each node to the target:-
* Its time complexity can be reduced using different data structures

We preferred it over A\* because:

****

**Implementation:**

In the implementation of our project we have used following data structures and techniques.

**Graphs:**

We had to use 2D array to store the graph representing adjacency matrix[[1]](#footnote-1).

**Arrays:**

Used arrays to store list of cities, their distances and many others information.

**File I/O:**

We have collected cities connection and their distance information in a text file, and other text file representing cities as a indices. In order to make a adjacency matrix we have to read the file so we used File/IO.

**Maps as a hash table:**

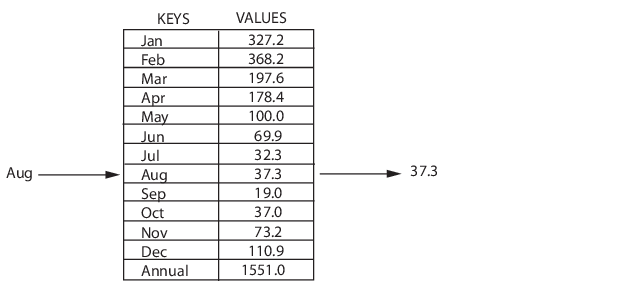
It was one the important data structure in our project. We had assigned every city a unique index like a[“Rawalpindi”]=1; etc that index we use to represent the cities so we stored that in STL: MAPS.

**Vectors:**

We had to store the Points and Vertices information in opengl, so we used Vectors in our project

**OpenGL:**

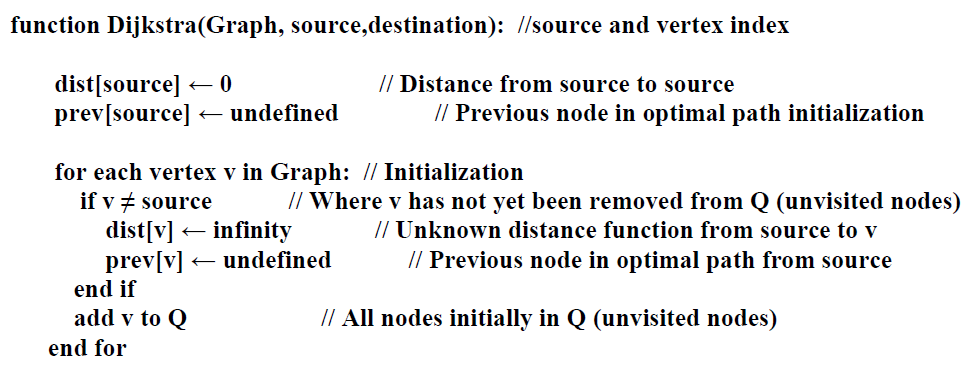
Used for the graphics:

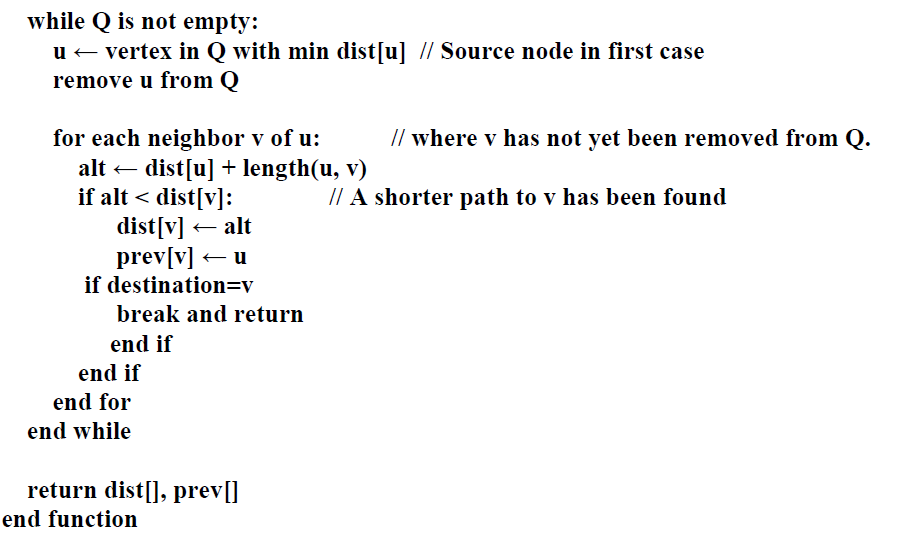


**Pseudo Code of Dijkstra’s algorithm:**

The following pseudo-code describes full working of the

Dijkstra’s algorithm.

****

****

This algorithm is tested by using simple unweighted graphs, it gave correct answers and also tested using google maps:

**Dijkstra’s –A Greedy Algorithm:**

A greedy algorithm follows the problem answering heuristic of making the.

In general it is provided that the greedy algorithm does not provide an optimal solution, but in spite of that, greedy heuristic may yield such locally optimal solutions that approximate a general optimal solution in a sensible time.

Greedy algorithms use methods of problem solving based on actions to see if there’s a better long-term strategy.

Dijkstra’s algorithm uses Greedy Approach to solve the single source shortest problem. After each iteration it selects from the unselected vertices, vertices nearest to source S and declares the distance that it is the actual shortest distance from source to that vertex.

**Result:**

**Running Time:**

The simplest implementation of the Dijkstra's algorithm stores vertices in an ordinary linked list or array

Good for dense graphs (many edges)

If you go through it linearly then finding a vertex will cost O(|V|) which in the total scheme of things will cost O(|V^2|).

**for arrays**

* |V| vertices and |E| edges
* Initialization O(|V|)
* While loop O(|V|)
* Find and remove min distance vertices O(|V|)
* Potentially |E| updates
* Update costs O(1)
* Reconstruct path O(|E|)
* Total time O(|V2| + |E|) = O(|V2| )
* Time Complexity: Using Lists

We can not use queues since each adjacent vertex now have a different cost. If we use a priority queue implemented through Binary heap, this brings down the complexity from

O(|E|+|V|^2) i.e. O(|V|2) to O(|E|log|V|).

A simple approach to this would be to add the adjacent vertex and its distance to a priority queue every time the distance (w.path) is updated. This may result in more than one representation for the same vertex. What is required is to delete subsequent versions once the smaller ones are used and made known.

**Priority Queue**

For sparse graphs, (i.e. graphs with much less than |V2| edges) Dijkstra's implemented more efficiently by *priority queue*

• Initialization O(|V|) using O(|V|) build Heap

• While loop O(|V|)

– Find and remove min distance vertices O(log |V|) using O(log |V|) deleteMin

– Potentially |E| updates

• Update costs O(log |V|) using decrease Key

• Reconstruct path O(|E|)

 Total time O(|V|log|V| + |E|log|V|) = O(|E|log|V|)

• |V| = O(|E|) assuming a connected graph

--Another data-structure , priority queue through Fibonacci heaps could bring the complexity down to

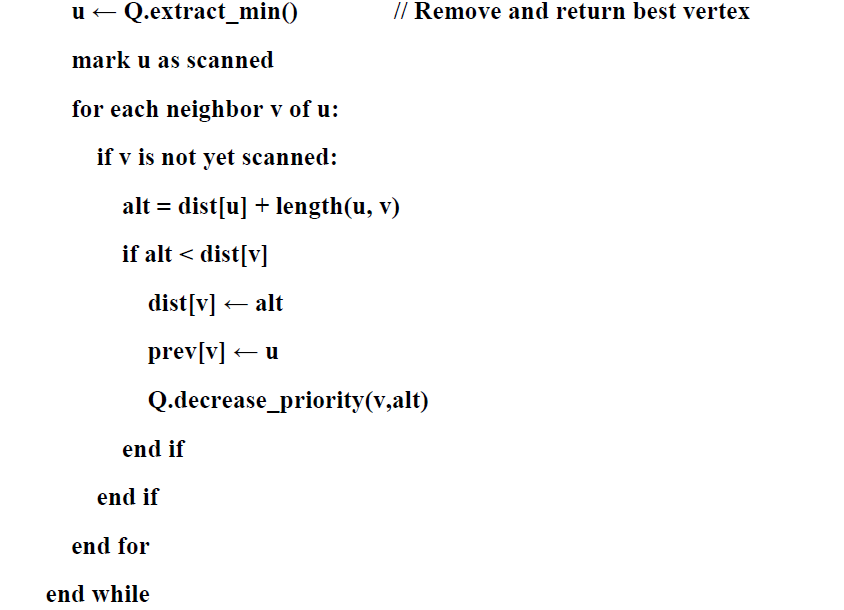
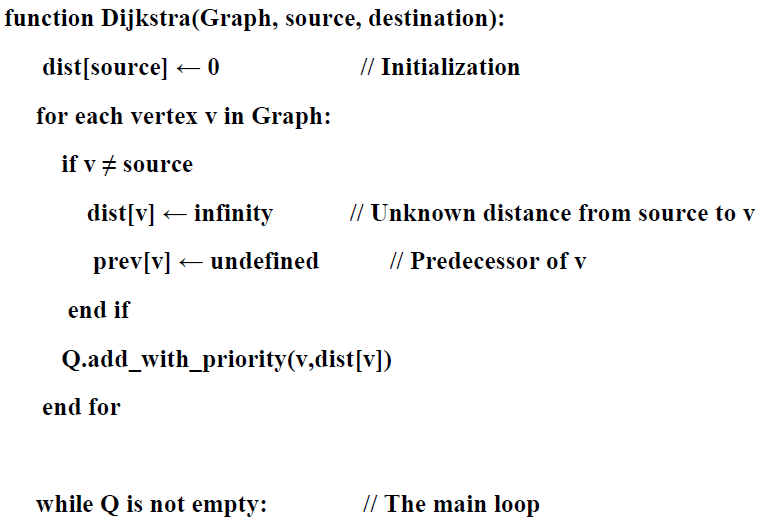
O(|E|+|V|log|V|).

Given there is a large overhead involved, it is not clear if this version is better. To-date, there are no meaningful average case results for this problem.

**Dis-advantages:**

The biggest disadvantage of Dijkstra’s algorithm is that it consumes a lot of time, in this way wasting required resources because it conducts blind search. This results in acyclic graphs and more often the right shortest path is not obtained. Another drawback is that it does not support negative weights of the edges.

**Improvements:**

We can further improve the efficiency of our program by using priority queues instead of arrays. The implementation with priority queues is as follows with some of minor changes like we will use queues instead of checked array: our approach would be like 

**Source Code approach of Making an application that uses Dijkstra's algorithm:**

As mentioned in our abstract of project, we have implemented this algorithm in making a prototype of map that uses some selected cities. In the provided source code, we have used 24 cities on that basis our whole code works. The steps that our code follows are mentioned below:

* **Reading strings of city values and their corresponding hash code from file through file handling approach.**
* **Function for separating string type city name and the integer type hash code into an array.**
* **Now reading from the file and storing into graph adjacency matrix that contains the distances from given city.**
* **Calculating the shortest distance from source to the destination city by using Dijkstra's algorithm and printing the shortest path.**
* **Now showing the obtained shortest path via Graphic user interface.**
* **For GUI approach the different functions of the OpenGl are used.**

**Maps in Standard Template Library:**

A Map is one type of fast key searching data structure in which we get the flexibility by means of indexing into its separate individual elements. Indices into the elements in a Map are named as **keys**. These key values are stored within the Map along with the unique associated data values means there are no any mapped value with same key values.

It is possible for a map to have different indices. In a Map, each entry contains exactly one unique key with related value which corresponds the key. The key value can be of different data types which includes integers, string of characters etc. and the value stored in the map also includes the data types of arrays of numeric values, character, objects arrays, or other Maps.

**Why to use STL’s maps in this project:**

The main reason of using maps is the flexibilities that it provides in terms of indices through which we can access associated data values easily. As this project includes different cities in which we are dealing with the strings of cities as the key values in order to access its associated information. For resolving this issue, we had one option to use our own hash table. Creating own hash table will require including the hash function in our code and writing it and handling collisions, and dealing with lots of testing to make sure that we got it right or not.

On the other hand, we are introduced with the Standard Template Library (STL) that includes a templated class to handle such these sort of situations: which you can conceptually think of as an associative array where key names are associated with the specified values

e.g., you can use a string as a key, and numerical values as the data, as in below example we are using months as the key which then identifies the numerical data type values.

**Source Code:**

//our project finds the shortest distance and path for given any two vertices

#include "Dependencies\glew.h"

#include "Dependencies\freeglut.h"

#include<iostream>

#include <stdio.h>

#include <stdlib.h>

#include<string>

#include<cstring>

#include<fstream>

#include<map>

#include <vector>

#include <sstream> //used to convert integer into string

#include "DrawGraph.h"

using namespace std;

int V = 0; //total vertices present in glass

string\* Store\_city = new string[100];//stores all the available cities

int\* city\_distance = new int[100];//store distance between all the vertices

string\* city\_path = new string[100];//stores all the cities that comes in the path of source to destination

int city\_count = 0;//counts all the path cities

string\* g = new string[100];//stores graph vertices

using namespace std;

DrawGraph graph;//main object to control the draGraph functions

class Shortest\_path{

public:

//constructor

map<string, int> index;

map<int, string> reverse\_index;

//functions

Shortest\_path(){}

string\* token(std::string str, char delimiter);

void readingIndex();

void reading(int\*\* graph);

void printShortestDistance(int\* weight, int n, int target\_index);

void printShortestpath(int\* prev, int n, int target\_index);

void dijkstra(int\*\* graph, int source\_index, int target\_index);

~Shortest\_path(){}//destructor

};

//token function

string\* Shortest\_path::token(std::string str, char delimiter)

{

if (str.find(delimiter) != std::string::npos) {

int countDel = 0;

std::string \*strArr;

for (int i = 0; i < str.length(); i++) {

if (str[i] == delimiter) {

countDel++;

}

}

strArr = new std::string[countDel + 1];

std::string temp;

for (int i = 0; i < (countDel + 1); i++) {

strArr[i] = str.substr(0, str.find(delimiter));

temp = str.substr(str.find(delimiter) + 1, str.length());

str = temp;

}

return strArr;

}

else {

return NULL;

}

}

void Shortest\_path::readingIndex(){

string line;

string\* tokens;

//reading from indices

ifstream file;

//initialize diagonal elements of the matrix by infinity;

file.open("new.txt");

if (file.good())

{

while (getline(file, line))

{

tokens = token(line, '-');

int index\_number = atoi(tokens[1].c\_str());

index[tokens[0]] = index\_number;

reverse\_index[index\_number] = tokens[0];

V++;

}

}

else

file.close();

//v = 24

}

//reading from the file and storing into graph adjacency matrix

void Shortest\_path::reading(int\*\* graph){

string line;

string\* tokens;

//reading from indices

ifstream file;

//initialize diagonal elements of the matrix by infinity;

for (int i = 0; i < V; i++){

for (int j = 0; j < V; j++){

graph[i][j] = 9999;

}

}

int counter = 0;

file.open("new.txt");

if (file.good())

{

while (getline(file, line))

{

tokens = token(line, '-');

int index\_number = atoi(tokens[1].c\_str());

index[tokens[0]] = index\_number;

reverse\_index[index\_number] = tokens[0];

Store\_city[counter] = tokens[0];

counter++;

}

}

else

file.close();

//reading from graph.txt and storing in 2 dimensional graph

ifstream f;

f.open("testing.txt");

if (f.good())

{

f.seekg(0);

while (getline(f, line))

{

tokens = token(line, '-');

int src\_index = index[tokens[0]];

int destination\_index = index[tokens[1]];

int distance = atoi(tokens[2].c\_str());

graph[src\_index][destination\_index] = distance;

graph[destination\_index][src\_index] = distance;

}

}

else

cout << "file distances not open\n";

f.close();

}

//prints the shortest path

void Shortest\_path::printShortestDistance(int\* weight, int n, int target\_index)

{

printf("Shortest distance from Source\n");

cout << weight[target\_index] << "KM";

}

//Function that gives the shortest path between two vertices when adjacency matrix is given

void Shortest\_path::printShortestpath(int\* prev, int n, int target\_index){

cout << "\n";

//printing the path

int i = target\_index;

{

int counter = 0;

int j;

cout << "Path = " << reverse\_index[i];

city\_path[counter] = reverse\_index[i];

counter++;

city\_count++;

j = i;

do{

j = prev[j];

cout << " <- " << reverse\_index[j];

city\_path[counter] = reverse\_index[j];;

counter++;

city\_count++;

} while (j != n);

cout << "\n";

}

}

//gives shortset distance and path between two vertices

void Shortest\_path::dijkstra(int\*\* graph, int source\_index, int target\_index)

{

int\* distance = new int[V];

int\* visited = new int[V];

int\* preD = new int[V];

int min;

int nextNode = 0;

for (int i = 0; i < V; i++){

visited[i] = 0;

preD[i] = source\_index;

distance[i] = graph[source\_index][i];

}

distance[source\_index] = 0;

visited[source\_index] = 1;

for (int i = 0; i < V; i++){

min = 9999;

for (int j = 0; j < V; j++){

if (min > distance[j] && visited[j] != 1){

min = distance[j];

nextNode = j;

}

}

visited[nextNode] = 1;

for (int c = 0; c < V; c++){

if (visited[c] != 1){

if (min + graph[nextNode][c] < distance[c]){

distance[c] = min + graph[nextNode][c];

preD[c] = nextNode;

}

}

}

if (visited[nextNode] == target\_index)

break;

}

printShortestDistance(distance, source\_index, target\_index);

printShortestpath(preD, source\_index, target\_index);

}

// driver program to test above function

void displayCities(Shortest\_path\* access){

for (int i = 0; i < V; i++){

cout << access->reverse\_index[i] << endl;

}

}

//constructor that sets radius of the circle

DrawGraph::DrawGraph() {

radius = .83;

sides;

}

//saves the co ordinates of the graph into .gph file

void DrawGraph::save(string fileName, vector <string> input) {

ofstream myfile;

//append the fileName to add the extention and open the file

fileName += ".gph";

myfile.open(fileName);

//write all of the accepted input into a file

for (int i = 0; i < input.size(); i++)

myfile << input[i] << endl;

//complete!

myfile.close();

}

//sets the vertices for the given points and stores into vertices object

void DrawGraph::setVerticesComplete() {

//This method draws all vertices for a complete graph

Vertices testVert;

for (int i = 0; i < pts.size() - 1; i++)

for (int j = 0; j < pts.size(); j++) {

if (i != j) {

testVert.x1 = pts[i].x;

testVert.y1 = pts[i].y;

testVert.x2 = pts[j].x;

testVert.y2 = pts[j].y;

vertices.push\_back(testVert);

}

}

}

//stores the points that needs to be drawn

void DrawGraph::setPoints() {

//run through all the points and set them

Point newPoint;

for (double i = 0; i < 2 \* PI; i += PI / (sides / 2.0)) {

newPoint.x = cos(i) \* radius;

newPoint.y = sin(i) \* radius;

pts.push\_back(newPoint);

}

}

//This method draws all of the points inside the DrawGraph object

void drawPoints() {

glPointSize(3);

//draw all of the points

glBegin(GL\_POINTS);

for (int i = 0; i < graph.pts.size(); i++)

glVertex2d(graph.pts[i].x, graph.pts[i].y);

glEnd();

}

//converts integer into string for the distances

string convertInt(int number) {

//converts an int into a string and returns it

stringstream ss;

ss << number;

//this fixes a bug where an extra number is sometimes printed

if (number > graph.sides)

return "";

return ss.str();

}

//turns integers into string for city indices

string convertInt2(int number) {

//converts an int into a string and returns it

stringstream ss;

ss << number;

return ss.str();

}

//draw string distances in opengl form

void drawString(float x, float y, string str) {

//Color and set position

glColor3f(0.0, 0.0, 1.0);

glRasterPos2f(x, y);

char scoreArr[300];

//converts the string to a char \*

sprintf\_s(scoreArr, str.c\_str());

//draw out each character

for (const char\* c = scoreArr; \*c != '\0'; c++) {

glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, \*c);

}

}

//draw string indices in opengl form

void drawString2(float x, float y, string str) {

//Color and set position

glColor3f(2.0, 0.0, 1.0);

glRasterPos2f(x, y);

char scoreArr[300];

//converts the string to a char \*

sprintf\_s(scoreArr, str.c\_str());

//draw out each character

for (const char\* c = scoreArr; \*c != '\0'; c++) {

glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, \*c);

}

}

//checks the cities if they are in the shortest path return true else return false

bool checkPath(string checkCityName){

for (int i = 0; i < city\_count;i++){

if (city\_path[i] == checkCityName)

return true;

}

return false;

}

//set color and paints all the points

void drawNames(){

//because it's drawn onto the screen an offset is needed

//enlarge makes the radius bigger so it doesn't overlap with the points

Shortest\_path object;

float offset = .03;

float enlarge = 1.08;

//convertInt(i + 1)

//loop through all the points and draw the point #'s

cout << graph.pts.size();

for (int i = 0; i < graph.pts.size(); i++){

//convertInt(i+1) converts the current point to a string and adds one (so we go from 1-22 etc.)

//drawString (x, y, string) x and y are based upon the point in which we want the string to be displayed

if (checkPath(Store\_city[i])==false)

drawString((graph.pts[i].x \* enlarge - offset), (graph.pts[i].y \* enlarge - offset), Store\_city[i]);

else

drawString2((graph.pts[i].x \* enlarge - offset), (graph.pts[i].y \* enlarge - offset), Store\_city[i]);

}

int temp = 0;

offset = .01;

enlarge = 1.01;

for (int i = 0; i < graph.vertices.size(); i++){

double xtemp = (graph.vertices[i].x1 + graph.vertices[i].x2) / 2;

double ytemp = (graph.vertices[i].y1 + graph.vertices[i].y2) / 2;

drawString((xtemp \* enlarge - offset), (ytemp \* enlarge - offset), convertInt2(city\_distance[temp])+"km");

++temp;

}

}

//set color and paints all the vertices

void drawVertices() {

//Shockingly this method draws all the vertices of a DrawGraph object

glLineWidth(1.0);

glColor3f(1, 0, 0);

glBegin(GL\_LINES);

for (int i = 0; i < graph.vertices.size(); i++) {

glVertex2f(graph.vertices[i].x1, graph.vertices[i].y1);

glVertex2f(graph.vertices[i].x2, graph.vertices[i].y2);

}

glEnd();

}

//main function to control all the display functions

void graphicsDisplay() {

//clear window

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

//Draw the points, vertices, and point names

drawPoints(); drawVertices(); drawNames();

//finish drawing

glutSwapBuffers();

}

//sets all the input and calls main graphic function to paint whole graph

void displayGraph(int\*\* gr) {

//Get input from the user to make a graph

Vertices testVert;

string testInput;

vector<string> input;

int counter = 0;

for (int i = 0; i < V; i++){

g[i] = "";

for (int j = 0; j < V; j++){

if (gr[i][j] == 9999)

g[i] = g[i] + "0";

else{

g[i] = g[i] + "1";

city\_distance[counter] = gr[i][j];

counter++;

}

}

}

//graph via keyboard

cout << "\n";

cout << "do you want to see the graph\n: ";

graph.sides = V;

graph.setPoints();

cout << "You chose " << graph.sides << " sides" << endl;

cout << "Please ensure each matrix row has a length of " << graph.sides << endl;

//this giant loop gets the matrix from the user

for (int i = 0; i < graph.sides; i++) {

//puts the current point for the entire line into the vertices

testVert.x1 = graph.pts[i].x;

testVert.y1 = graph.pts[i].y;

//clear input

testInput = "";

//ensure the input has enough characters

while (testInput.length() < graph.sides) {

//cout << "Please enter the " << i + 1 << " line of the matrix : ";

testInput = g[i];

}

//now that we've gotten a valid input, put it in the vector

input.push\_back(testInput);

//loop through all inputs, if a 1 is found then it needs to be connected

for (int j = 0; j < graph.sides; j++) {

if (testInput[j] == '1') {

//places the vertices requested, attached to the current point (eg 3)

testVert.x2 = graph.pts[j].x;

testVert.y2 = graph.pts[j].y;

//put into the graph

graph.vertices.push\_back(testVert);

}

}

} //end matrix input

//ask user to save graph

cout << "Would you like to save this graph? Y or N : ";

cin >> testInput;

if (testInput == "Y" || testInput == "y") {

cout << "Please enter a file name: (ex: myGraph)";

cin >> testInput;

graph.save(testInput, input);

}

}

//menu to call displayGraph

void menu(int\*\* gr) {

//MENU

string option;

cout << "Welcome to DrawGraph" << endl;

cout << "Press 1 to draw a Graph " << endl;

do{

cin >> option;

if (option == "1")

displayGraph(gr);

} while (option!="1");

}

//control function

int main(int argc, char \*argv[])

{

//adjacency matrix //9999 shows 0 input meaning no edge

Shortest\_path\* obj = new Shortest\_path;

string choice;

obj->readingIndex();

int\*\* grap = new int\*[V];

for (int i = 0; i < V; i++){

grap[i] = new int[V];

}

string source, destination;

int source\_index, destination\_index;

obj->reading(grap);

cout << endl;

while (1){

cout << "\*\*\*\*\*\*\*\*\*\*\* Shortest Path \*\*\*\*\*\*\*\*\*\*\n" << endl;

do{

cout << "Do you want to see the list of available Cities\n"

<< "Pree 1 for yes or 0 for No\n" << endl;

cin >> choice;

if (choice == "1")

displayCities(obj);

} while (choice != "0");

do{

cout << "Enter the Source city" << endl;

cin >> source;

source\_index = obj->index[source];

} while (source != obj->reverse\_index[source\_index]);

do{

cout << "Enter the Destination city" << endl;

cin >> destination;

destination\_index = obj->index[destination];

} while (destination != obj->reverse\_index[destination\_index]);

cout << endl;

obj->dijkstra(grap, source\_index, destination\_index);

cout << endl;

do{

cout << "Press 1 to continue , 0 to exit" << endl;

cin >> choice;

if (choice == "0")

break;

} while (choice != "1");

if (choice == "0")

break;

menu(grap);

//initialize OpenGL

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA | GLUT\_DEPTH);

//window attributes

glutInitWindowSize(500, 500);

glutInitWindowPosition(0, 0);

glutCreateWindow("DrawGraph");

//register display callback

glClearColor(1, 1, 1, 1);

glutDisplayFunc(graphicsDisplay);

//turn control over to event loop

glutMainLoop();

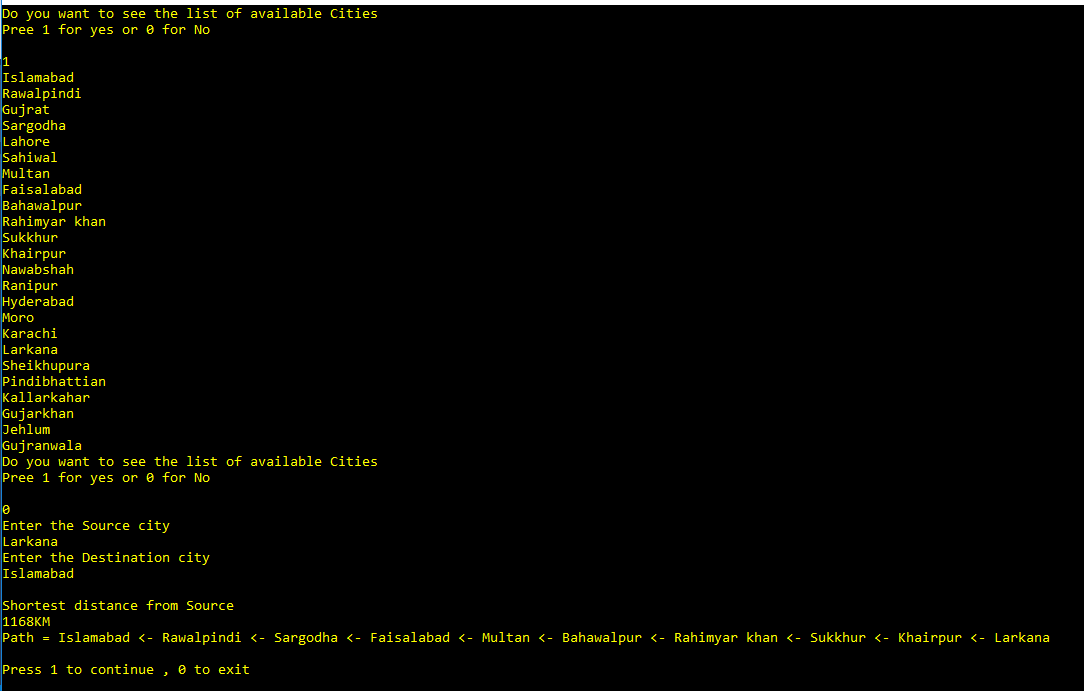
}

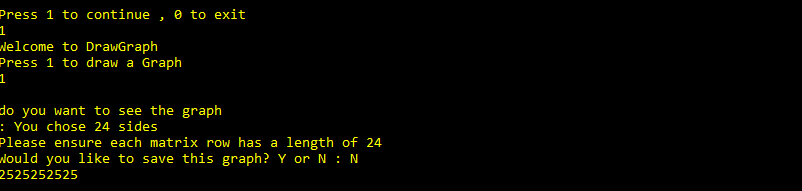
system("pause");

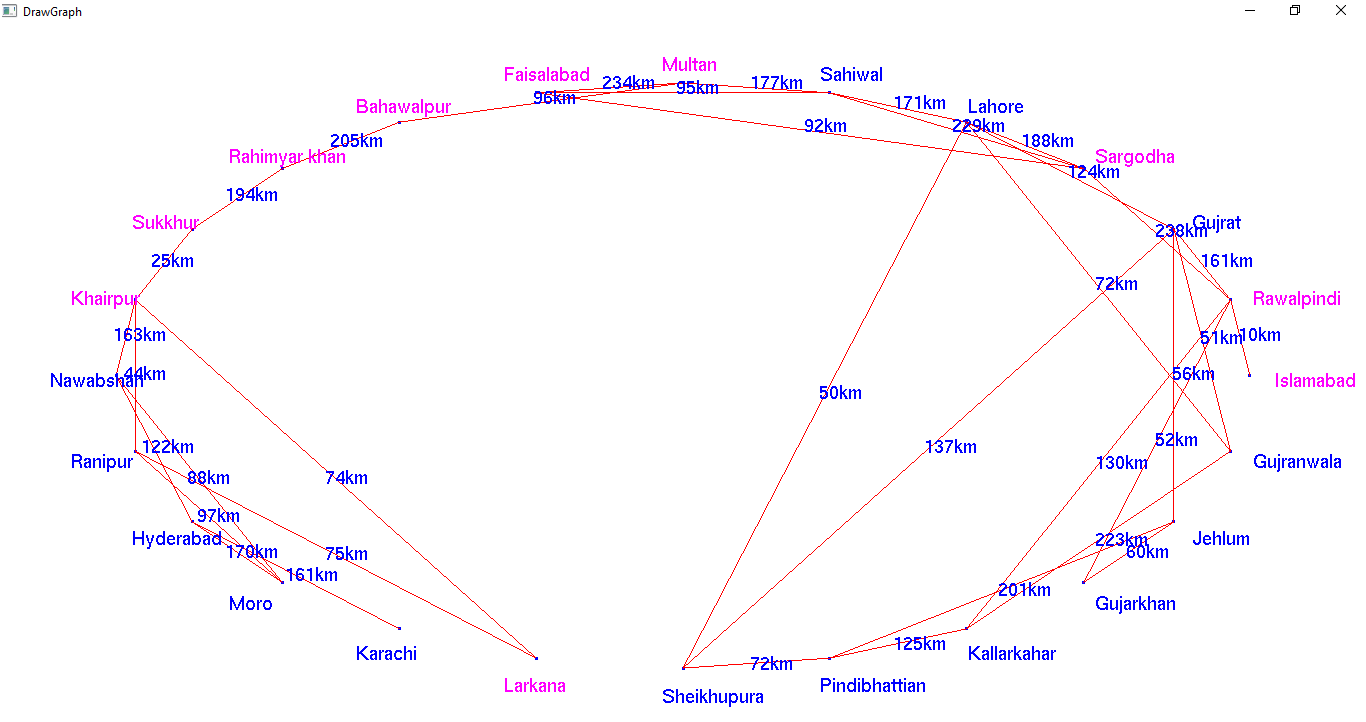
return 0;

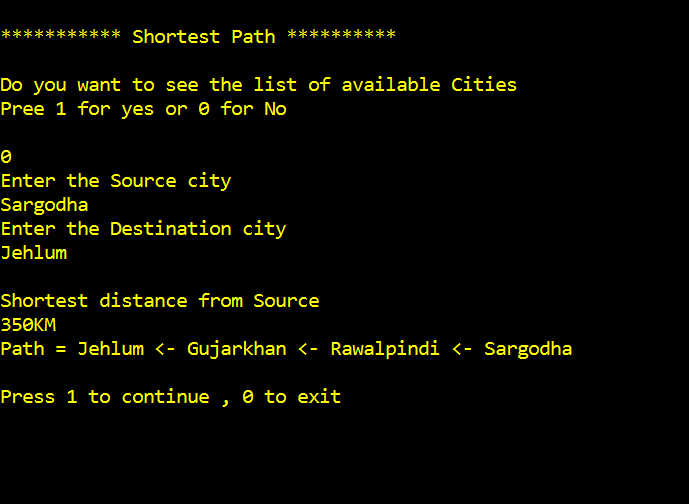
}

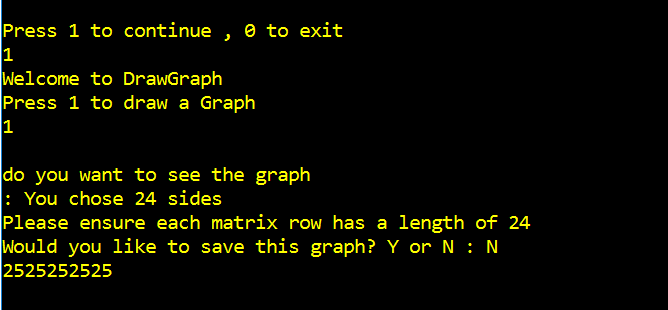
**Output:**

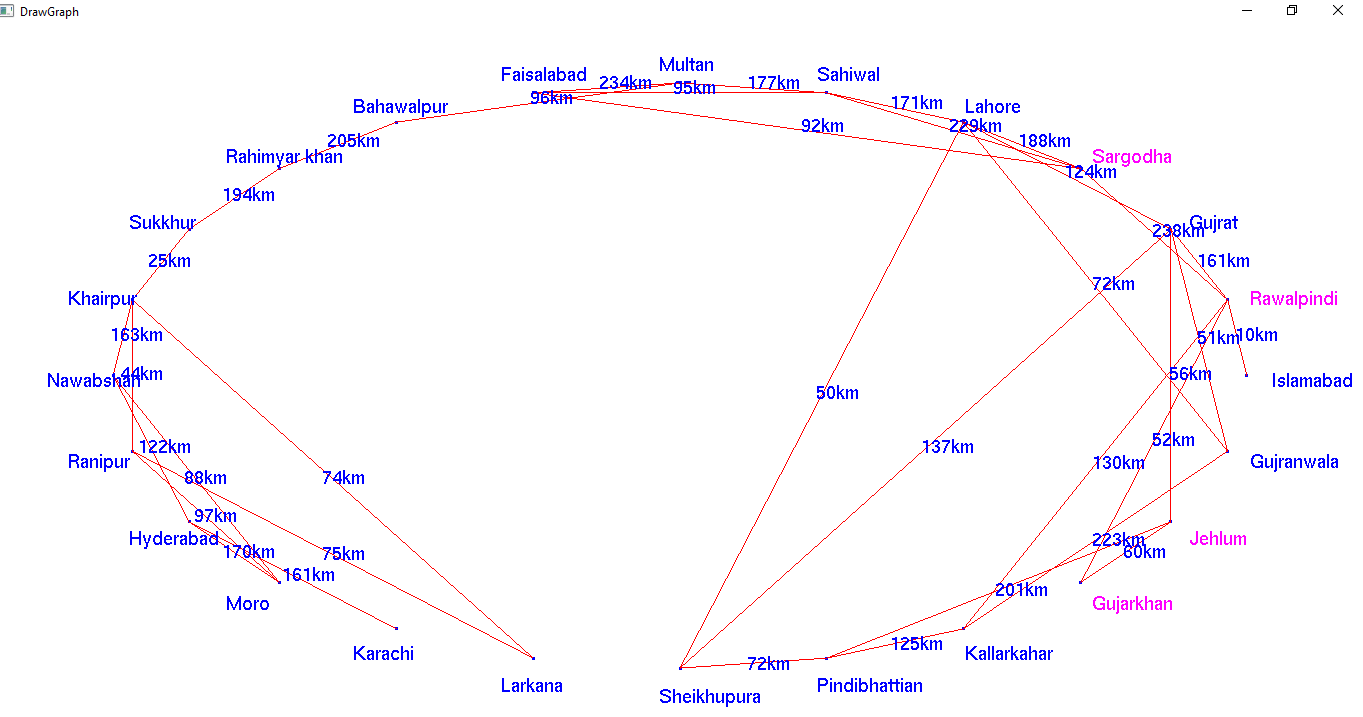
****

****

****

****

****

****

1. https://en.wikipedia.org/wiki/Adjacency\_matrix [↑](#footnote-ref-1)