QUICKER TO DESTINATION

LIBRARIES USED

1. Graphviz Library
2. Standard Template Library

WHAT IS GRAPHVIZ?

Graphviz is open source graph visualization software. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. It has important applications in networking, bioinformatics, software engineering, database and web design, machine learning, and in visual interfaces for other technical domains.

WHAT IS DOT LANGUAGE?

Dot language, also known as Graphviz DOT language, is a simple textual language used to describe graphs and networks. It is primarily used with the Graphviz software, a popular open-source graph visualization tool developed by AT&T Labs.

WHAT IS STL?

STL stands for the "Standard Template Library." It is a collection of template classes and functions in C++ that provides common data structures and algorithms to work with those data structures. The STL was introduced as part of the C++ Standard Library, making it an essential and integral part of the C++ programming language

DATA STRUCTURES & AIGORITHMS USED

- *Graph [Adjacency List]*
- *Dijkstra's Algorithm*
- *Multi-Dimensional Vectors*
- *Min-Heap [Priority Queue]*
- *Map*
- *StringStream*
- *Arrays*
- *Primitive Data Types*

Header Files

RoadNetwork.h > ...

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <limits.h>
#include <map>
#include <algorithm>
using namespace std;

class Path{
public:
    vector<int> shortestPath(vector<int>& arr, int source, int destination);

    void printPath(const vector<int>& path, double distance, const map<int, string>& mpp);

    void allPathsUtil(vector<vector<pair<int, double>>>& adj, int source, int destination, vector<bool>& visited,
                      vector<int>& path, double distance, const map<int, string>& mpp);

    void allPaths(vector<vector<pair<int, double>>>& adj, int source, int destination, const map<int, string>& mpp);

    vector<int> shortestDistance(vector<vector<pair<int, double>>>& adj, int V,
                                 int source, int destination, map<int, string>& mpp);
};
```

Class Name: Path

Description: A class that provides functions for finding the shortest path and all paths between nodes in a weighted graph.

Public Member Functions:
1. `vector<int> shortestPath(vector<int>& arr, int source, int destination)`: Finds the shortest path between nodes using Dijkstra's algorithm.
2. `void printPath(const vector<int>& path, double distance, const map<int, string>& mpp)`: Displays the path and its distance based on node IDs and a map of node names.
3. `void allPaths(vector<vector<pair<int, double>>>& adj, int source, int destination, const map<int, string>& mpp)`: Finds and displays all paths between nodes in the graph.
4. `vector<int> shortestDistance(vector<vector<pair<int, double>>>& adj, int V, int source, int destination, map<int, string>& mpp)`: Finds the shortest distance and the path between nodes using Dijkstra's algorithm and returns the path as a vector of node IDs.

makeGraph.h > ...

```cpp
#include <iostream>
#include <map>
#include <fstream>
#include <set>
using namespace std;

class MakeGraph{
private:
    map<int,string> getLocationName;
    map<string,int> getLocationId;
public:
    void addEdge(vector<vector<pair<int, double>>>& adj);
    void addLocations();
    map<int, string> returnLocationNames();
    map<string, int> returnLocationId();
    vector<string> extractWords(const string& input);
    void displayMap(bool organized);
    vector<string> splitString(const string& inputStr, char delimiter);
    void displayLocationList();
    void runDotFile(string fileName, bool organized);
    void displayShortestPath(const vector<string>& path, bool organized);
};
```

Class Name: MakeGraph

Description: A class to create and manage a graph data structure representing locations. It provides functions to add edges between locations, extract location names and IDs, display the graph, generate Dot files for visualization, and find the shortest path between locations.

Public Member Functions:
- `void addEdge(vector<vector<pair<int, double>>>& adj)`: Add edges between locations in the graph.
- `void addLocations()`: Add locations to the graph.
- `map<int, string> returnLocationNames()`: Get the map of location names with their IDs.
- `map<string, int> returnLocationId()`: Get the map of location IDs with their names.
- `void displayMap(bool organized)`: Display the graph with all locations and connections.
- `void runDotFile(string fileName, bool organized)`: Generate a Dot file for graph visualization.
- `void displayShortestPath(const vector<string>& path, bool organized)`: Display the shortest path between two locations.

DESCRIPTION:

Prompts the user to enter source and destination location IDs, validates the input, and calculates the shortest path using Dijkstra's algorithm. The resulting path is stored in both integer and string formats.

```
int getUserInput(const string& prompt) {
    int userInput;
    cout << prompt;
    cin >> userInput;
    return userInput;
}

void getSourceAndDestination(int& source, int& destination, int& V, vector<int>& shortestPathInt,
vector<string>& shortestPathString, vector<vector<pair<int, double>>>& adjacencyList,map<int, string>& mpp, Path& p){
    source = getUserInput("Enter the ID of the source location: ");
    destination = getUserInput("Enter the ID of the destination location: ");

    if (mpp.find(source) == mpp.end() || mpp.find(destination) == mpp.end()) {
        cout << "Invalid location IDs entered. Please try again with valid IDs." << endl;
        return;
    }

    cout << "Source Location: " << mpp[source] << endl;
    cout << "Destination Location: " << mpp[destination] << endl;
    // Dijkstras will take care of the printig of Shortest path.
    shortestPathInt = p.shortestDistance(adjacencyList, V, source, destination, mpp);
    for (int i = 0; i < shortestPathInt.size(); i++) {
        shortestPathString.push_back(mpp[shortestPathInt[i]]);
    }
    return;
}
```

Make Graph Class

```cpp
vector<string> MakeGraph::splitString(const string& inputStr, char delimiter) {
    vector<string> result;
    stringstream ss(inputStr);
    string item;

    while(getline(ss, item, delimiter)) {
        result.push_back(item);
    }
    return result;
}
```

Description: A member function of the class MakeGraph that splits a given input string based on a specified delimiter character and returns the substrings as a vector of strings.

Parameters:
- inputStr (const string&): The input string to be split.
- delimiter (char): The character used to separate the substrings in the input string.

Functionality: The function uses the stringstream class to break down the input string inputStr into substrings, using the delimiter to identify the boundaries. Each substring is then added to the vector result, and the final vector containing the split strings is returned.

Description: A member function of the class MakeGraph that extracts individual words from a given input string based on a specified delimiter character (`,`), and returns them as a vector of strings. It also removes leading and trailing whitespaces from each extracted word.

Parameters:
- input (const string&): The input string from which words are to be extracted.

Return Type: vector<string>

Functionality: The function utilizes the stringstream class to break down the input string input into individual words, using the comma (,) as the delimiter. It then removes any leading and trailing whitespaces from each word before adding it to the vector words. The final vector containing the extracted words is returned.

```cpp
vector<string> MakeGraph::extractWords(const string& input) {
    vector<string> words;
    stringstream ss(input);
    //delimeter - ','
    string word;
    while (getline(ss, word, ',')) {
        // Remove leading/trailing whitespaces from the word
        size_t start = word.find_first_not_of(" ");
        size_t end = word.find_last_not_of(" ");
        if (start != string::npos && end != string::npos)
            words.push_back(word.substr(start, end - start + 1));
    }
    return words;
}
```

```cpp
void MakeGraph::addEdge(vector<vector<pair<int, double>>>& adj){
    ifstream inputFile("DataSet.txt", ios::in);
    if (!inputFile.is_open()){
        cout << "Error opening the file! DataSet.txt" << endl;
        return;
    }
    // cout << "At AddEdge, DataSet.txt\n";
    string line;
    vector<string> result = extractWords(line);
    while(getline(inputFile, line)){
        vector<string> result = extractWords(line);
        string place1 = result[0];
        string place2 = result[1];
        double distance = stod(result[2]);
        // cout << "Place 1 : " << place1 << ", Place 2 : " << place2 << ", Distance : " << distance << endl;
        adj[getLocationId[place1]].push_back({getLocationId[place2], distance});
        adj[getLocationId[place2]].push_back({getLocationId[place1], distance});
    }
    inputFile.close();
    return;
}
```

Description: Reads edge data from "DataSet.txt" file, extracts location IDs, names, and distances from each line using the extractWords function, and populates the adjacency list (adj) with the edges between locations.

Example Usage:
- Input File (DataSet.txt):
  Vandalur,Urappakkam,5.4
  Vandalur,Adhanur,6.5
  Vandalur,Perungalathur,3.0

Resulting Adjacency List:
  adj[1]]: { {2, 5.4}, {3, 6.5}, {4,3.0} }
  adj[2]: { {1,5.4} }
  adj[3]: { {1, 6.5} }

```cpp
void MakeGraph::addLocations(){
    ifstream inputFile("DataSetList.txt", ios::in);
    if (!inputFile.is_open()){
        cout << "Error opening the file! DataSetList.txt" << endl;
        return;
    }
    string line;
    while(getline(inputFile, line)){
        int id;
        string place;

        vector<string> words = splitString(line, '.');
        id = stod(words[0]);
        place = words[1];
        getLocationId[place] = id;
        getLocationName[id] = place;
    }
    inputFile.close();
    return;
}
```

Description: Reads location data from "DataSetList.txt" file, extracts location IDs and names from each line using a period ('.') as the delimiter, and populates the getLocationId and getLocationName maps.

Usage:
- Input File (DataSetList.txt):
  1.Vandalur
  2.Urappakkam
  3.Adhanur

- Resulting Maps:
  getLocationId: {"Vandalur"->1, "Urappakkam"->2, "Adhanur"->3}
  getLocationName: {1->"Vandalur", 2->"Urappakkam", 3->"Adhanur"}

Description: Reads and displays location data from "DataSetList.txt" file, which contains a list of locations with their IDs.

Example Usage:
- Input File (DataSetList.txt):
  1.Vandalur
  2.Urappakkam
  3.Adhanur

- Output: (This will be printed in CLI))
  1.Vandalur
  2.Urappakkam
  3.Adhanur

```
void MakeGraph::displayLocationList(){
    ifstream inputFile("DataSetList.txt");
    if (!inputFile) {
        cerr << "Error: Unable to open the input file." << endl;
        return;
    }

    string line;
    while(getline(inputFile, line)){
        cout << line << endl;
    }
    return;
}
```

```
map<int, string> MakeGraph::returnLocationNames(){
    return getLocationName;
}

map<string, int> MakeGraph::returnLocationId(){
    return getLocationId;
}
```

Function Name: returnLocationNames

Description: Returns a map containing location names as values and their corresponding IDs as keys.

Function Name: returnLocationId

Description: Returns a map containing location IDs as values and their corresponding names as keys.

Creating Dot File for Visualization

```
void MakeGraph::displayMap(bool organised){
    ifstream inputFile("DataSet.txt");
    if (!inputFile) {
        cerr << "Error: Unable to open the input file." << endl;
        return;
    }

    ofstream outputFile("Graph.dot");
    if (!outputFile) {
        cerr << "Error: Unable to create the output .dot file." << endl;
        return;
    }

    outputFile << "graph G {" << endl;

    string line;
    while(getline(inputFile, line)){
        vector<string> result = extractWords(line);
        string source = result[0];
        string target = result[1];
        double weight = stod(result[2]);
        outputFile << "    \"" << source << "\" -- \"" << target << "\" [label=\"" << weight << "\"];\n";
    }

    outputFile << "}" << endl;
    cout << "Graph data written to 'graph.dot' successfully." << endl;
    cout << "Dot file 'Graph.dot' created successfully." << endl;

    inputFile.close();
    outputFile.close();

    runDotFile("graph", organised);
    return;
}
```

Function Name: displayMap

Description: Reads edge data from "DataSet.txt" file, generates a Dot file ('Graph.dot') to visualize the graph using Graphviz, and runs the Dot file to create a graph visualization.

Parameters:
- organised (bool): A boolean parameter indicating whether the graph should be organized when visualized.

Return Type: void

Functionality: The function reads edge data from the "DataSet.txt" file, where each line represents a connection between two locations with a weight (distance). It creates a Dot file named 'Graph.dot', which defines the graph's structure and connections based on the provided edge data. The Dot file includes attributes such as edge labels to represent the weights of connections. Once the Dot file is generated, the function runs the Dot file using Graphviz to produce a graph visualization showing the connections between locations.

Notes:
- The "DataSet.txt" file should contain edge data in the format: "SourceLocation. TargetLocation. Weight"
- The graph visualization can be customized using Graphviz's layout options and styling.

INTERACTING WITH COMMAND PROMPT

```
void MakeGraph::runDotFile(string fileName, bool organized){
    string command;
    if(organized){
        command = "dot -Tpng -Gdpi=300 -o " + fileName + ".png " + fileName + ".dot";
    } else {
        command = "fdp -Tpng -Gdpi=300 -o " + fileName + ".png " + fileName + ".dot";
    }


    int result = system(command.c_str());

    if (result != 0) {
        cerr << "Error converting .dot to PNG." << endl;
        return;
    }

    cout << "Graph converted to 'graph.png' successfully." << endl;
    command = "start " + fileName + ".png";

    result = system(command.c_str());

    if (result != 0) {
        cerr << "Error opening the PNG file." << endl;
        return;
    }
    return;
}
```

Function Name: runDotFile

Description: Converts the Dot file ('Graph.dot') to a PNG image using Graphviz and opens the PNG image for graph visualization.

Parameters:
- fileName (string): The base name of the Dot file and the resulting PNG image file (without the extension).
- organized (bool): A boolean parameter indicating whether the graph should be organized when visualized. If true, the "dot" layout engine is used; otherwise, the "fdp" layout engine is used.

Functionality: The function generates a PNG image file visualizing the graph defined in the Dot file using Graphviz. It runs the Graphviz tool with the specified layout engine and DPI settings to create the PNG image. The generated image is saved as "fileName.png". The function then opens the PNG image for display.

Notes:
- The 'runDotFile' function requires Graphviz to be installed with the "dot" and "fdp" layout engines to work correctly.

```cpp
void MakeGraph::displayShortestPath(const vector<string>& path, bool organized) {
    ifstream inputFile("DataSet.txt");
    if (!inputFile) {
        cerr << "Error: Unable to open the input file." << endl;
        return;
    }

    ofstream outputFile("pathInGraph.dot");
    if (!outputFile) {
        cerr << "Error: Unable to create the output .dot file." << endl;
        return;
    }

    outputFile << "graph G {" << endl;

    string line;
    while (getline(inputFile, line)) {
        vector<string> result = extractWords(line);
        string source = result[0];
        string target = result[1];
        double weight = stod(result[2]);

        bool isHighlighted = false;
        for (size_t i = 0; i < path.size() - 1; i++) {
            if ((source == path[i] && target == path[i + 1]) || (source == path[i + 1] && target == path[i])) {
                isHighlighted = true;
                break;
            }
        }
        if (isHighlighted) {
            outputFile << "    \"" << source << "\" -- \"" << target << "\" [label=\"" << weight << "\", color=\"red\", penwidth=2.0];\n";
        } else {
            outputFile << "    \"" << source << "\" -- \"" << target << "\" [label=\"" << weight << "\"];\n";
        }
    }

    // Set different shapes for nodes in the path
    set<string> highlightedNodes(path.begin(), path.end()); // Convert the path vector to a set
    for (const auto& node : getLocationName) {
        string nodeName = node.second;
        if (highlightedNodes.find(nodeName) != highlightedNodes.end()) {
            outputFile << "    \"" << nodeName << "\" [shape=box];\n";
        } else {
            outputFile << "    \"" << nodeName << "\";\n";
        }
    }

    outputFile << "}" << endl;

    cout << "Graph data written to 'pathInGraph.dot' successfully." << endl;
    cout << "Dot file 'pathInGraph.dot' created successfully." << endl;

    inputFile.close();
    outputFile.close();

    runDotFile("pathInGraph", organized);
    return;
}
```

Function Name: displayShortestPath

Description: Generates a Dot file ('pathInGraph.dot') to visualize the graph with the shortest path highlighted in red, and then runs the Dot file to create a graph visualization.

Parameters:
- path (const vector<string>&): A vector representing the shortest path consisting of location names.
- organized (bool): A boolean parameter indicating whether the graph should be organized when visualized. If true, the "dot" layout engine is used; otherwise, the "fdp" layout engine is used.

Functionality: The function reads edge data from the "DataSet.txt" file, where each line represents a connection between two locations with a weight (distance). It creates a Dot file named 'pathInGraph.dot', which defines the graph's structure and connections based on the provided edge data. The Dot file includes attributes such as edge labels, edge color (red for the shortest path), and node shapes (box for nodes in the path). Once the Dot file is generated, the function runs the Dot file using Graphviz to produce a graph visualization. The shortest path in the graph is highlighted in red.

Notes:
- The 'displayShortestPath' function requires that Graphviz (with the "dot" and "fdp" layout engines) is installed on the system to work correctly.

ROAD NETWORK CLASS

Function Name: shortestDistance

Description: Finds the shortest distance and the shortest path between a source and destination node in a weighted graph represented by an adjacency list.

Parameters:
- adj (vector<vector<pair<int, double>>>&): The adjacency list representing the graph with weighted edges.
- V (int): The number of nodes (vertices) in the graph.
- source (int): The index of the source node.
- destination (int): The index of the destination node.
- mpp (map<int, string>&): A map that stores node indices as keys and corresponding location names as values.

Functionality: The function uses Dijkstra's algorithm to find the shortest distance and the shortest path between the source and destination nodes in the graph. It employs a Min-Heap (priority_queue) to efficiently select the next closest node for exploration. The shortest distance to each node is stored in the 'distance' vector, and the previous node on the shortest path to each node is recorded in the 'prevLocation' vector. The shortest path is reconstructed from the 'prevLocation' vector. The function also prints the source, destination, shortest distance, and the shortest path between them using the 'mpp' map for node index to location name conversion. Finally, it returns the shortest path as a vector of node indices.

```cpp
vector<int> Path::shortestDistance(vector<vector<pair<int, double>>>& adj,
                                   int V, int source, int destination, map<int, string>& mpp) {

    // Min-Heap
    priority_queue<pair<int, double>, vector<pair<int, double>>, greater<pair<int, int>>> pq;
    vector<double> distance(V, INT_MAX);
    distance[source] = 0;
    pq.push({0, source});
    vector<int> prevLocation(V);
    for(int i = 0; i < V; i++){
        prevLocation[i] = i;
    }
    while(!pq.empty()){
        double dist = pq.top().first;
        int location = pq.top().second;
        pq.pop();

        for(auto ele : adj[location]){
            int nearestPlace = ele.first;
            double nearestDistance = ele.second;

            if(nearestDistance + dist < distance[nearestPlace]){
                distance[nearestPlace] = nearestDistance + dist;
                pq.push({distance[nearestPlace], nearestPlace});
                prevLocation[nearestPlace] = location;
            }
        }
    }
    cout << endl;
    cout << "Source : " << mpp[source] << endl;
    cout << "Destination : " << mpp[destination] << endl;
    cout << "Shortest Distance : " << distance[destination] << endl;
    cout << "Shortest Path : ";
    vector<int> shortestPathInt = shortestPath(prevLocation, source, destination);
    printPath(shortestPathInt, distance[destination], mpp);
    return shortestPathInt;
}
```

Notes:
- The 'adj' parameter represents a directed graph with weighted edges. Each element of the 'adj' vector is a pair containing the node index and the corresponding edge weight to neighboring nodes.
- The 'mpp' map is used to convert node indices to location names for better readability.
- The function assumes that the graph is connected, and there exists a path between the source and destination nodes.

```cpp
vector<int> Path::shortestPath(vector<int>& arr, int source, int destination){
    int node = destination;
    vector<int> path;
    while(arr[node] != node){
        path.push_back(arr[node]);
        node = arr[node];
    }
    reverse(path.begin(), path.end());
    path.push_back(destination);
    return path;
}
```

Description: Reconstructs the shortest path between a source and destination node using the 'arr' vector, which represents the previous node on the shortest path.

Parameters:
- arr (vector<int>&): A vector that stores the previous node on the shortest path for each node.
- source (int): The index of the source node.
- destination (int): The index of the destination node.

Return Type: vector<int>

Functionality: The function reconstructs the shortest path from the 'source' to the 'destination' node using the 'arr' vector, which stores the previous node on the shortest path for each node. Starting from the 'destination' node, it iteratively moves backward through the 'arr' vector, pushing the previous node into the 'path' vector until it reaches the 'source' node. The 'path' vector represents the shortest path from the 'source' to the 'destination'. The resulting path is then reversed to get the correct order, and the 'destination' node is appended at the end. Finally, the function returns the vector representing the shortest path.

Notes:
- The 'arr' vector is generated by the 'shortestDistance' function, which uses Dijkstra's algorithm.
- The 'shortestPath' function assumes that a valid path exists between the 'source' and 'destination' nodes in the graph.

```cpp
void Path::printPath(const vector<int>& path, double distance, const map<int, string>& mpp){
    for (int i = 0; i < path.size(); i++) {
        cout << mpp.at(path[i]);
        if (i != path.size() - 1)
            cout << " -> ";
    }
    cout << endl;
    cout << "Distance: " << distance << endl;
    cout << endl;
}
```

Function Name: printPath

Description: Prints the given path, its distance, and the corresponding location names.

Parameters:
- path (const vector<int>&): A vector representing the path, containing node indices.
- distance (double): The total distance of the path.
- mpp (const map<int, string>&): A map that stores node indices as keys and corresponding location names as values.

Return Type: void

Functionality: The function prints the given 'path' vector, which represents a sequence of node indices. It converts each node index to its corresponding location name using the 'mpp' map and prints the location names in the order of the path. The function also prints the 'distance', which represents the total distance covered along the path.

Notes:
- The 'mpp' map is used to convert node indices to location names for better readability.

PRINTING ALL THE POSSIBLE PATHS

Function Name: allPaths

Description: Finds and prints all possible paths between a source and destination node in a weighted graph.

Parameters: adj (vector<vector<pair<int, double>>>&), source (int), destination (int), mpp (const map<int, string>&)

Functionality: Initializes data and calls 'allPathsUtil' for path exploration. Displays paths using 'printPath' function.

```
void Path::allPaths(vector<vector<pair<int, double>>>& adj, int source,
        int destination, const map<int, string>& mpp) {

    vector<bool> visited(adj.size(), false);
    vector<int> path;
    double distance = 0.0;
    cout << "All possible paths between " << mpp.at(source)
        << " and " << mpp.at(destination) << " are:\n";
    allPathsUtil(adj, source, destination, visited, path, distance, mpp);
}
```

```
void Path::allPathsUtil(vector<vector<pair<int, double>>>& adj,
            int source, int destination, vector<bool>& visited,
            vector<int>& path, double distance, const map<int, string>& mpp) {

    visited[source] = true;
    path.push_back(source);

    if (source == destination) {
        printPath(path, distance, mpp);
    } else {
        for (auto ele : adj[source]) {
            int nearestPlace = ele.first;
            double edgeWeight = ele.second;
            if (!visited[nearestPlace]) {
                allPathsUtil(adj, nearestPlace, destination,
                visited, path, distance + edgeWeight, mpp);
            }
        }
    }
    path.pop_back();
    visited[source] = false;
}
```

Function Name: allPathsUtil

Description: Recursive utility function to find all possible paths between a source and destination node in a weighted graph.

Parameters: adj (vector<vector<pair<int, double>>>&), source (int), destination (int), visited (vector<bool>&), path (vector<int>&), distance (double), mpp (const map<int, string>&)

Return Type: void

Functionality: Uses DFS to explore paths from 'source' to 'destination'. Prints paths and distances using 'printPath' function.

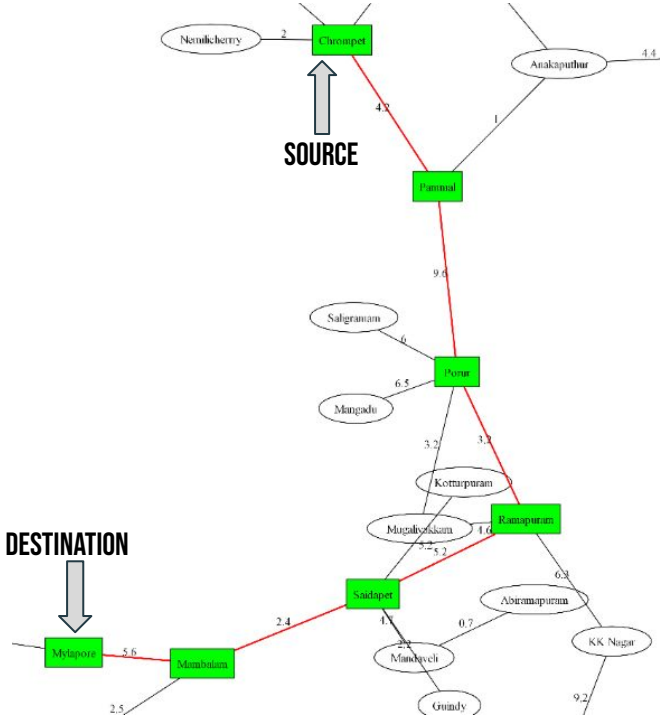
SAMPLE OUTPUT

```
1.Display all the available Locations list
2.Display the whole Map
3.Shortest Path in text
4.Shortest Path in Map
5.Print All Possible Paths
6.Close the map

Enter your option : 1

1.Vandalur
2.Urappakkam
3.Adhanur
4.Perungalathur
5.Mudichur
6.Tambaram
7.Sanatorium
8.Chitlapakkam
9.Chrompet
```

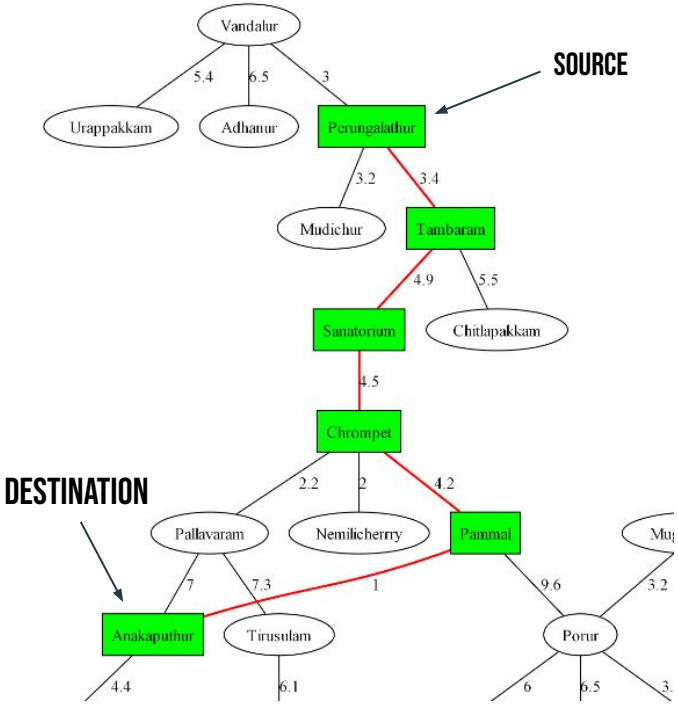
```
47.Chetpet
48.Arumbakkam
49.Kilpauk
50.Egmore
51.Park Town
52.Chennai Central
53.Chennai Fort
54.Porur

Location List is displayed Successfully!
```

```
Enter your option : 3

Enter the ID of the source location: 9
Enter the ID of the destination location: 48
Source Location: Chrompet
Destination Location: Arumbakkam

Source : Chrompet
Destination : Arumbakkam
Shortest Distance : 36.9
Shortest Path : Chrompet -> Pammal -> Porur -> Ramapuram -> Saidapet -> Mambalam -> Kodampakkam -> Nungampakkam -> Chetpet -
> Arumbakkam
Distance: 36.9

Displayed Shortest Path in Text SuccessFully
```

```
Enter your option : 5

All possible paths between Chrompet and Arumbakkam are:
Chrompet -> Pallavaram -> Anakaputhur -> Pammal -> Porur -> Mugalivakkam -> Ramapuram -> KK Nagar -> Velachery -> Guindy ->
Saidapet -> Mambalam -> Kodampakkam -> Nungampakkam -> Chetpet -> Arumbakkam
Distance: 64.5

Chrompet -> Pallavaram -> Anakaputhur -> Pammal -> Porur -> Mugalivakkam -> Ramapuram -> Saidapet -> Mambalam -> Kodampakkam
 -> Nungampakkam -> Chetpet -> Arumbakkam
Distance: 47.5

Chrompet -> Pallavaram -> Anakaputhur -> Pammal -> Porur -> Ramapuram -> KK Nagar -> Velachery -> Guindy -> Saidapet -> Mamb
alam -> Kodampakkam -> Nungampakkam -> Chetpet -> Arumbakkam
Distance: 59.9

Chrompet -> Pallavaram -> Anakaputhur -> Pammal -> Porur -> Ramapuram -> Saidapet -> Mambalam -> Kodampakkam -> Nungampakkam
 -> Chetpet -> Arumbakkam
Distance: 42.9

Chrompet -> Pammal -> Porur -> Mugalivakkam -> Ramapuram -> KK Nagar -> Velachery -> Guindy -> Saidapet -> Mambalam -> Kodam
pakkam -> Nungampakkam -> Chetpet -> Arumbakkam
Distance: 58.5

Chrompet -> Pammal -> Porur -> Mugalivakkam -> Ramapuram -> Saidapet -> Mambalam -> Kodampakkam -> Nungampakkam -> Chetpet -
> Arumbakkam
Distance: 41.5

Chrompet -> Pammal -> Porur -> Ramapuram -> KK Nagar -> Velachery -> Guindy -> Saidapet -> Mambalam -> Kodampakkam -> Nungam
pakkam -> Chetpet -> Arumbakkam
Distance: 53.9
```



Random Map



Organized Map

CONCLUSION:

The road network mapping project successfully implemented a program to model and navigate a road network using C++ and graph algorithms. The project allowed users to visualize the road map, find the shortest path between two locations, and display all possible paths between locations. It also provided features to display the road network on both organized and random maps, making it convenient for users to understand and plan their routes effectively.

LEARNING OUTCOMES:

1. Graph Representation: Learnt how to represent a road network using an adjacency list graph data structure. This allowed for efficient storage and traversal of the road connections.

2. Dijkstra's Algorithm: Gained an understanding of Dijkstra's algorithm for finding the shortest path between two locations in a weighted graph. This algorithm was used to calculate the shortest distance and path between any two given locations in the road network.

3. File Handling: Learned how to read data from external text files and store it in data structures. The project involved handling multiple files to create and display the road network effectively.

4. Object-Oriented Programming: Implemented the project using object-oriented programming concepts, such as classes and encapsulation, to organize and manage different functionalities effectively.

5. Error Handling: Developed skills in handling potential errors, such as file opening errors, invalid location IDs, or out-of-range exceptions, to provide users with informative error messages.

6. Visualization with GraphViz: Utilized the GraphViz library to visualize the road network in organized and random maps. This enhanced the user experience by providing visual representations of the graph.

7. Problem-Solving: Explored how to approach and solve a complex problem, such as finding the shortest path in a road network, breaking it down into manageable tasks, and implementing efficient algorithms to achieve the desired results.

8. User Interface (CLI): Implemented a simple Command Line Interface (CLI) to interact with the road network, providing users with a user-friendly experience and various options to explore the road map.

By working on this project, I have gained valuable experience in C++ programming, data structures, graph algorithms, and file handling. Additionally, I have improved my problem-solving skills and learned how to implement a real-world project from conception to completion. The road network mapping project has been an enriching learning experience that has broadened my knowledge and skills in programming and algorithm design.

THANK YOU!