**MINI PROJECT**

**On**

**MAP NAVIGATION**

**In**

**Data Structure & Algorithms**

**BACHELOR OF TECHNOLOGY**

**IN**

**Artificial Intelligence and Machine Learning**

SUBMITTED BY

**Shreya Chaturvedi**

**(PRN - 23070126123)**

**A red logo with a globe and text

Description automatically generated**

**Under the Supervision of**

**Ashwini Tande**

**SYMBIOSIS INSTITUTE OF TECHNOLOGY, PUNE – 412115**

**(A CONSTITUENT OF SYMBIOSIS INTERNATIONAL (DEEMED UNIVERSITY))**

**2024-25**

**Problem Statement**

Design a command-line map navigation system to calculate and display the shortest route between cities using Dijkstra's algorithm, offering travel time estimates based on transportation mode, simulated weather, traffic analysis, and road closure updates. The system supports real-time route modifications, displaying city connections and updating routes dynamically as per user interactions.

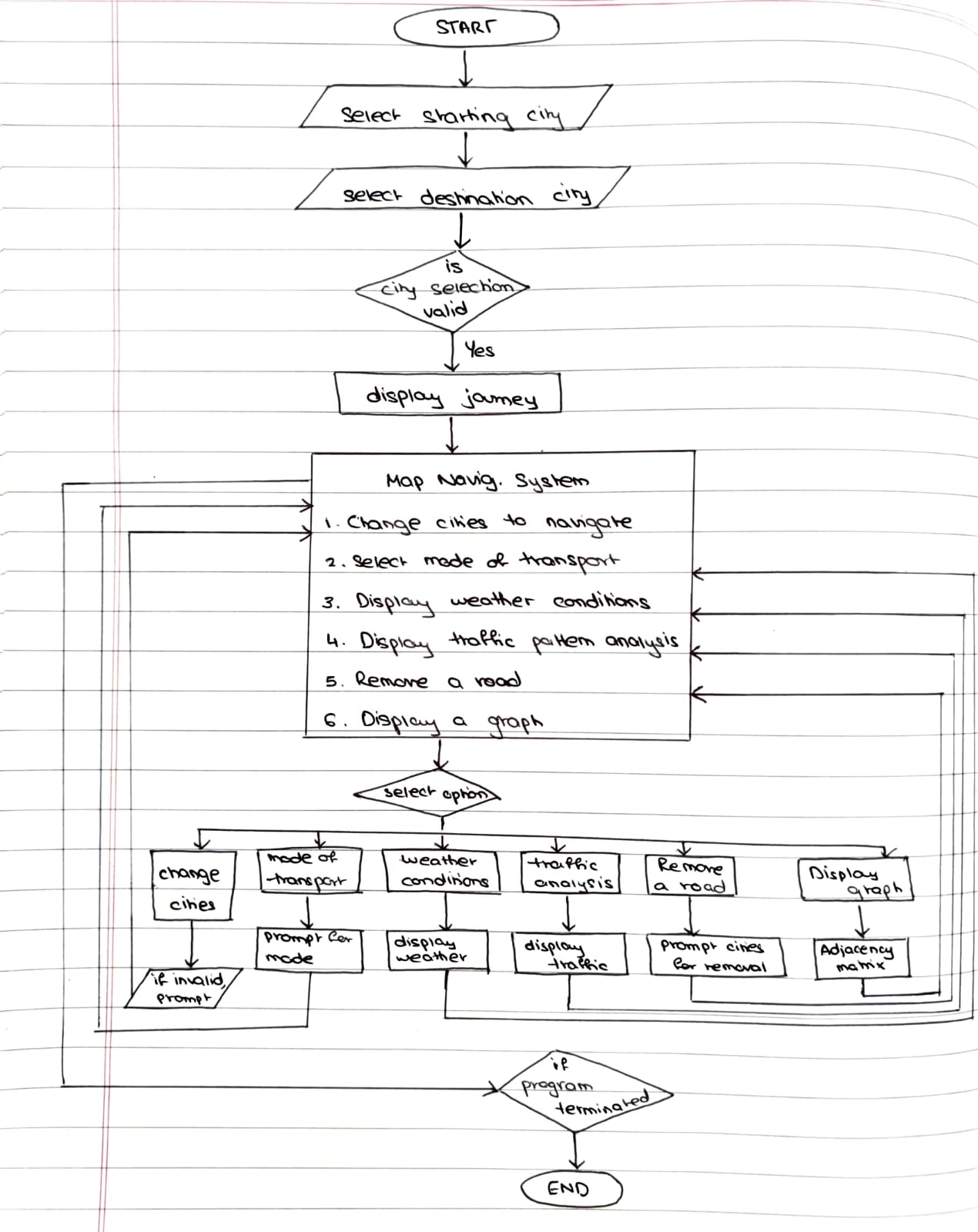
**Motivation**

* Applying Graph Algorithms: To practically implement Dijkstra's algorithm, enhancing understanding of shortest path algorithms and their real-world applications in routing systems.
* Simulating Real-World Navigation: To create a basic, functional navigation tool that models real-life navigation features like route optimization, weather, and traffic, making the project more relatable and realistic.
* Transport Mode Integration: To understand how travel time varies with different transportation modes, reflecting on how speed impacts route planning in diverse conditions.

**Objective**

* Implement Route Calculation: Accurately compute and display the shortest path between selected cities using graph-based algorithms, focusing on efficient navigation and pathfinding.
* Simulate Real-Time Navigation Features: Enable real-time interactions such as updating route availability, checking for weather impacts, and assessing traffic conditions, making the system adaptable to varying scenarios.
* User-Friendly Navigation and Information Display: Design a clear, interactive interface for selecting cities, choosing transportation modes, and visualizing city connections, enhancing usability for journey planning.

**Methodology of Implementation**

****

*FLOWCHART FOR MAP NAVIGATION SYSTEM*

**1. Graph Initialization**

Graph g; initializeGraph(&g);

The program starts by initializing the graph, which represents cities and roads. Each city is a node, and roads between cities are edges with weights (distances).

The adjacency matrix g.adjMatrix holds distance values between cities, where INF represents no direct connection.

**2. Selecting Starting and Destination Cities**

Input Section:

The program prompts the user to input the starting and destination cities (with city numbers ranging from 0 to MAX\_CITIES - 1). These cities serve as points for route planning, affecting calculations for travel time, weather conditions, and other metrics.

**3. Validating City Selection**

Input Validation:

The program checks if the selected cities are within the valid range. If the selection is invalid (either city is out of range), it displays an error message and prompts the user to reselect valid cities.

**4. Display Journey**

displayJourney(&g, city1, city2);

Once valid cities are selected, the program displays the journey or path from the starting city to the destination city. This function likely uses an algorithm (e.g., Dijkstra’s) to calculate the shortest or optimal path between the two cities.

**5. Main Options Menu**

After displaying the journey, the program presents a menu with various options for additional interactions. The user can select from the following:

* Option 1: Change Cities to Navigate From/To

Allows the user to reselect the starting and destination cities.

The program will then call displayJourney(&g, city1, city2); again to update the journey based on the new city selection.

* Option 2: Select Mode of Transport

Function: calculateTravelTime(g.adjMatrix[city1][city2], transport);

The user selects a mode of transport (Car, Flight, or Train).

Based on the mode of transport and the distance between cities, the program calculates and displays the estimated travel time. If the travel time calculation is successful, it prints the travel time; if the transport mode is invalid, it displays an error.

* Option 3: Display Weather Conditions

Function: displayWeather();

Displays weather conditions for the cities or along the journey path.

The weather data may be generated randomly or based on predefined conditions, helping the user make informed travel decisions.

* Option 4: Display Traffic Pattern Analysis

Function: displayTrafficPattern();

Shows traffic data between cities, which may include peak traffic times, congestion areas, or other relevant patterns. This helps the user assess how traffic could impact their journey.

* Option 5: Remove a Road (Real-Time Update)

Function: removeRoad(&g, rmCity1, rmCity2);

Allows the user to remove a road between two cities by setting the distance to INF, indicating no direct connection. This represents a real-time update in the graph, affecting subsequent route calculations and journey displays.

* Option 6: Display Graph

Displays the Adjacency Matrix:

Shows the adjacency matrix, which lists distances between cities. For city pairs with no direct road, the distance is shown as INF. This option helps the user visualize the entire graph structure and all connections.

**6. Looping and Continuation**

The program continuously loops through the menu, allowing the user to perform multiple actions without restarting. This loop gives the user flexibility to reselect cities, check conditions, update the graph, and view results until they manually exit the program.

**7. End of Program**

The program only terminates if the user exits manually. Otherwise, it remains in the loop to enable multiple operations in one session.

**Software Used**: Visual Studio Code

**Executable Code**

#include <stdio.h> // Required libraries

#include <stdlib.h>

#include <string.h>

#include <time.h>

#define MAX\_CITIES 5 // Maximum number of cities

#define INF 99999 // Constant for no direct path

// Graph structure with adjacency matrix for distances and array for city names

typedef struct {

    int adjMatrix[MAX\_CITIES][MAX\_CITIES];

    char\* cityNames[MAX\_CITIES];

} Graph;

// Initialize the graph with city names and distances

void initializeGraph(Graph\* g) {

    g->cityNames[0] = "City 0";

    g->cityNames[1] = "City 1";

    g->cityNames[2] = "City 2";

    g->cityNames[3] = "City 3";

    g->cityNames[4] = "City 4";

    // Set initial distances; 0 for same city, INF for others

    for (int i = 0; i < MAX\_CITIES; i++) {

        for (int j = 0; j < MAX\_CITIES; j++) {

            g->adjMatrix[i][j] = (i == j) ? 0 : INF;

        }

    }

    // Define specific distances between cities

    g->adjMatrix[0][1] = 50; g->adjMatrix[1][0] = 50;

    g->adjMatrix[1][2] = 70; g->adjMatrix[2][1] = 70;

    g->adjMatrix[1][3] = 60; g->adjMatrix[3][1] = 60;

    g->adjMatrix[3][4] = 40; g->adjMatrix[4][3] = 40;

    g->adjMatrix[0][2] = 150; g->adjMatrix[2][0] = 150;

    g->adjMatrix[0][3] = 90; g->adjMatrix[3][0] = 90;

    g->adjMatrix[0][4] = 200; g->adjMatrix[4][0] = 200;

    g->adjMatrix[2][3] = 100; g->adjMatrix[3][2] = 100;

}

// Display a random weather condition

void displayWeather() {

    const char\* weatherConditions[] = {"Sunny", "Rainy", "Windy", "Cloudy"};

    srand(time(0));

    int randomCondition = rand() % 4; // Random weather index

    printf("Weather condition is: %s\n", weatherConditions[randomCondition]);

}

// Display random traffic congestion level

void displayTrafficPattern() {

    srand(time(0));

    int trafficCongestion = rand() % 100; // Random congestion level

    printf("Traffic analysis: Current congestion is %d%%.\n", trafficCongestion);

}

// Remove a road between two cities by setting distance to INF

void removeRoad(Graph\* g, int city1, int city2) {

    g->adjMatrix[city1][city2] = INF;

    g->adjMatrix[city2][city1] = INF;

    printf("Road between %s and %s has been removed due to road closure.\n", g->cityNames[city1], g->cityNames[city2]);

}

// Dijkstra's algorithm to find the shortest path between two cities

int dijkstra(Graph\* g, int src, int dest, int\* path) {

    int dist[MAX\_CITIES], visited[MAX\_CITIES], prev[MAX\_CITIES];

    // Initializing distance, visited, and previous arrays

    for (int i = 0; i < MAX\_CITIES; i++) {

        dist[i] = INF;

        visited[i] = 0;

        prev[i] = -1;

    }

    dist[src] = 0; // Starting city's distance is 0

    // Finding shortest path to each city

    for (int i = 0; i < MAX\_CITIES - 1; i++) {

        int min = INF, u = -1;

        for (int j = 0; j < MAX\_CITIES; j++) {

            if (!visited[j] && dist[j] <= min) {

                min = dist[j];

                u = j;

            }

        }

        visited[u] = 1;

        for (int v = 0; v < MAX\_CITIES; v++) { // Updating distances for neighboring cities

            if (!visited[v] && g->adjMatrix[u][v] != INF && dist[u] + g->adjMatrix[u][v] < dist[v]) {

                dist[v] = dist[u] + g->adjMatrix[u][v];

                prev[v] = u; // Track path

            }

        }

    }

    int pathLength = 0; // Reconstructing path from destination to source

    for (int at = dest; at != -1; at = prev[at]) {

        path[pathLength++] = at;

    }

    for (int i = 0; i < pathLength / 2; i++) {  // Reversing path to display correctly

        int temp = path[i];

        path[i] = path[pathLength - i - 1];

        path[pathLength - i - 1] = temp;

    }

    return dist[dest];

}

// Displaying journey details (Dijkstra's algorithm)

void displayJourney(Graph\* g, int city1, int city2) {

    printf("Journey from %s to %s:\n", g->cityNames[city1], g->cityNames[city2]);

    printf("------------------------------------------------\n");

    int path[MAX\_CITIES];

    int distance = dijkstra(g, city1, city2, path);

    // Displaying path if reachable

    if (distance == INF) {

        printf("No direct or indirect route available between %s and %s.\n", g->cityNames[city1], g->cityNames[city2]);

    } else {

        printf("Shortest Path (Dijkstra's Algorithm):\n");

        for (int i = 0; path[i] != city2; i++) {

            printf("%s --[%d]--> ", g->cityNames[path[i]], g->adjMatrix[path[i]][path[i + 1]]);

        }

        printf("%s\n", g->cityNames[city2]);

        printf("Total Distance: %d km\n", distance);

    }

    printf("------------------------------------------------\n");

}

// Calculating travel time based on transport mode

float calculateTravelTime(int distance, const char\* transport) {

    float speed;

    if (strcmp(transport, "Car") == 0) speed = 60;

    else if (strcmp(transport, "Flight") == 0) speed = 800;

    else if (strcmp(transport, "Train") == 0) speed = 100;

    else return -1; // Invalid transport mode

    return (float)distance / speed;

}

int main() {

    Graph g;

    initializeGraph(&g);

    int city1 = -1, city2 = -1;

    int choice;

    do {

        // Selecting cities for navigation

        printf("Select starting city (0 to %d): ", MAX\_CITIES - 1);

        scanf("%d", &city1);

        printf("Select destination city (0 to %d): ", MAX\_CITIES - 1);

        scanf("%d", &city2);

        if (city1 < 0 || city1 >= MAX\_CITIES || city2 < 0 || city2 >= MAX\_CITIES) {

            printf("Invalid city selection. Please choose valid cities.\n");

            continue;

        }

        displayJourney(&g, city1, city2);

        do {

            // Displaying options menu

            printf("\n--- Map Navigation System ---\n");

            printf("1. Change Cities to Navigate From/To\n");

            printf("2. Select Mode of Transport\n");

            printf("3. Display Weather Conditions\n");

            printf("4. Display Traffic Pattern Analysis\n");

            printf("5. Remove a Road (Real-Time Update)\n");

            printf("6. Display Graph\n");

            printf("Enter your choice: ");

            scanf("%d", &choice);

            switch (choice) {

                case 1:

                    printf("Enter the cities to navigate from and to (0 to %d): ", MAX\_CITIES - 1);

                    scanf("%d %d", &city1, &city2);

                    displayJourney(&g, city1, city2);

                    break;

                case 2: {

                    char transport[10];

                    printf("Select mode of transport (Car/Flight/Train): ");

                    scanf("%s", transport);

                    float travelTime = calculateTravelTime(g.adjMatrix[city1][city2], transport);

                    if (travelTime >= 0) {

                        printf("Estimated travel time from %s to %s by %s is %.2f hours.\n", g.cityNames[city1], g.cityNames[city2], transport, travelTime);

                    } else {

                        printf("Invalid mode of transport selected.\n");

                    }

                    break;

                }

                case 3:

                    displayWeather();

                    break;

                case 4:

                    displayTrafficPattern();

                    break;

                case 5: {

                    int rmCity1, rmCity2;

                    printf("Enter the two cities for road removal (0 to %d): ", MAX\_CITIES - 1);

                    scanf("%d %d", &rmCity1, &rmCity2);

                    removeRoad(&g, rmCity1, rmCity2);

                    break;

                }

                case 6:

                    printf("Adjacency Matrix:\n");

                    for (int i = 0; i < MAX\_CITIES; i++) {

                        for (int j = 0; j < MAX\_CITIES; j++) {

                            if (g.adjMatrix[i][j] == INF) printf("INF\t");

                            else printf("%d\t", g.adjMatrix[i][j]);

                        }

                        printf("\n");

                    }

                    break;

                default:

                    printf("Invalid choice. Please try again.\n");

                    break;

            }

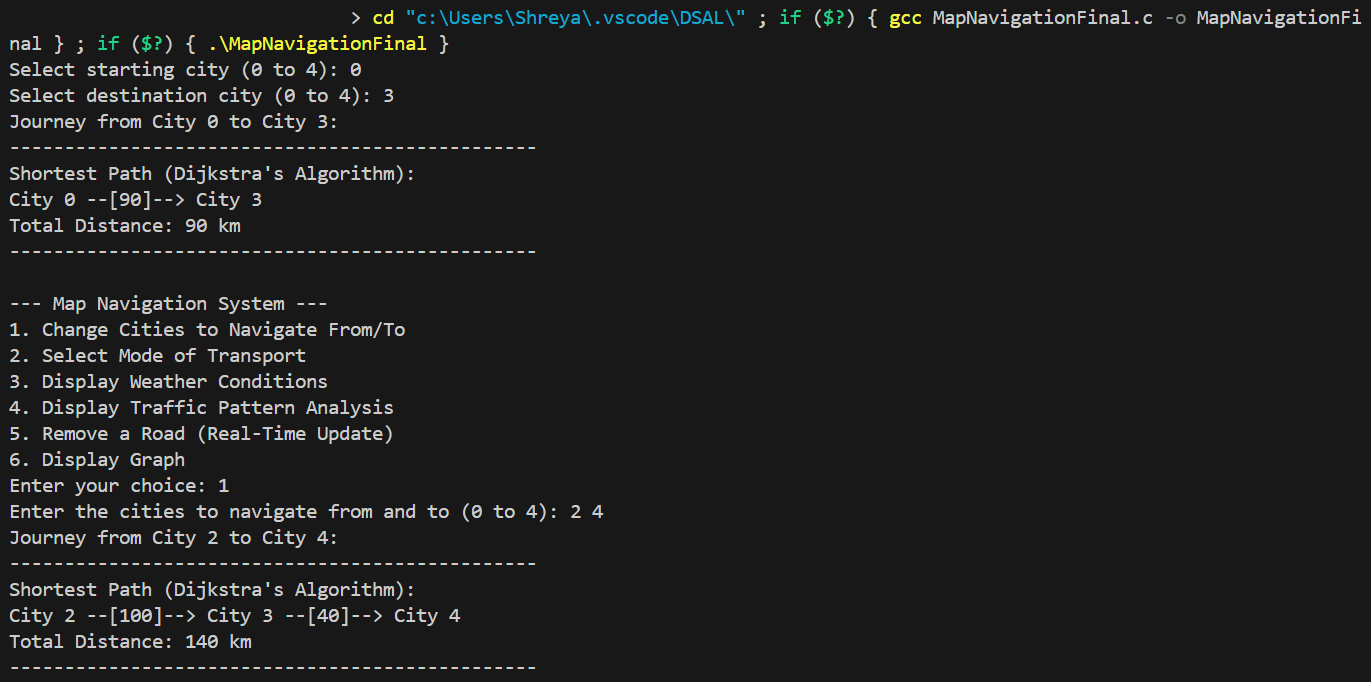
        } while (1);

    } while (1);

    return 0;

}

**Result Analysis with Output Screen Shot**



After selecting the starting and destination city, we are given the shortest path using Dijkstra’s Algorithm along with the total distance covered. This is the default output. We are then provided with 6 options within the Map Navigation system. These include: Changing of cities, selecting mode of transport, displaying weather conditions, displaying traffic pattern analysis, removing a road, displaying graph.

Selecting option 1 gives us the option to change cities for navigation. After choosing desired cities, we are once again given the shortest path and total distance.

A computer screen shot of a black screen

Description automatically generated

Selecting option 3 gives us random weather condition.

Option 2 allows us to choose the mode of transport and accordingly provides us with the estimated travel time from selected starting city to destination city.

A computer screen shot of a black screen

Description automatically generated

Option 4 gives us a randomized number for the current congestion in terms of traffic analysis.

Option 5 allowed for the real time update of the removal of a road

A black rectangle with white text

Description automatically generated

Option 6 displays the adjacency matrix of the graph we have created of the cities.

**Implementing Symbol Table and AVL Tree in your Projects?**

In a project like a Map Navigation System, **a Symbol Table** can be used to efficiently manage and retrieve information about cities and their respective data. A Symbol Table allows the storage of key-value pairs, where each city’s name can serve as the key, and the associated data (like coordinates, population, weather data, etc.) can be stored as the value. This can facilitate quick lookups when users select cities for navigation. Using hash-based or tree-based implementations, the Symbol Table would optimize access time, particularly beneficial in expanding the application to handle more cities and datasets. This design ensures that city data retrieval remains fast and efficient even as the project scales.

**An AVL Tree** (Adelson-Velsky and Landis Tree) can be implemented to maintain balanced routes or city distances for quick access, especially for dynamic scenarios where cities or roads may be added or removed. AVL Trees automatically balance after every insertion or deletion, maintaining an O(log n) time complexity for search operations. This feature is ideal for real-time updates, such as road closures, as it allows the program to quickly reorganize city routes and adjust travel times. By integrating an AVL Tree, the project can efficiently handle dynamic changes, improving performance in route recalculations and providing a scalable structure for managing expanding city networks.

**Learning Outcome**

* Understanding Graph Theory and Algorithms: Gain practical experience with graph data structures and algorithms like Dijkstra's, deepening knowledge in pathfinding and shortest route calculation applicable to navigation systems.
* Real-Time System Updates: Develop skills in handling real-time updates within data structures, such as modifying routes due to road closures, and experience the practical challenges of maintaining system performance under dynamic conditions.
* Simulating Real-World Conditions: Gain experience in simulating real-world elements like weather, traffic patterns, and transport-based travel estimates, enhancing the ability to model real-life systems through programming.

**References**

1. <https://github.com/Rohilkan/Map-Navigator-Project>
2. [**https://www.scribd.com/document/689691965/CS23D017-DSA-PROJECT**](https://www.scribd.com/document/689691965/CS23D017-DSA-PROJECT)
3. [**https://www.youtube.com/watch?v=FoJrnDSiUgE**](https://www.youtube.com/watch?v=FoJrnDSiUgE)
4. [**https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/**](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/)