ADAPTIVE TRAFFIC ROUTING SOLUTION

A mini project

IT23302- Data Structures

Submitted By:

KUMARAN S	- 2023506091
DILLI DIKSHITH D	- 2023506089
VIJESH T	- 2023506087

Under the guidance of:

Dr. P. KOLA SUJATHA

DEPARTMENT OF INFORMATION TECHNOLOGY

MADRAS INSTITUTE OF TECHNOLOGY

ANNA UNIVERSITY

CHENNAI -600 044

August / December 2024

ABSTRACT

This project focuses on analyzing and modeling transportation or social networks using graph theory concepts. The network is represented using an adjacency matrix, enabling efficient storage and traversal of connections between cities. Key functionalities include:

1. **Shortest Path Analysis** using Dijkstra's Algorithm to determine the least-cost path between cities.
2. **Dynamic Network Management** for interactive updates to the network.
3. **Real-Time Path Tracing**, enhancing usability and decision-making.

The system is implemented in C, featuring a menu-driven interface for network management. It demonstrates practical applications of graph algorithms to solve real-world problems like transportation optimization. Future enhancements include support for larger networks, integration of advanced pathfinding algorithms, and transitioning to adjacency list representation for scalability.

TABLE OF CONTENTS

INTRODUCTION:

Efficient route planning and analysis are essential for connecting cities or nodes in transportation networks. This project leverages graph theory to represent cities as nodes and their interconnections as edges. The adjacency matrix approach is used for representing the network, offering straightforward implementation and clarity for smaller networks.

By combining graph algorithms with user-friendly interfaces, this system enables real-time management and visualization of connections. It highlights the power of graph-based data structures to tackle practical problems like shortest-path computation, dynamic updates, and connectivity analysis.

OBJECTIVES:

1.Graph Representation:
Represent transportation networks using graphs with an adjacency matrix.

2.Shortest Path Analysis:
Use Dijkstra's Algorithm to calculate the shortest path between two cities.

3.Dynamic Network Management:
Allow interactive addition, removal, and modification of connections.

4.Path Visualization:
Display paths between cities, enhancing usability.

5.Practical Demonstration of Graph Algorithms:
Showcase how algorithms like Dijkstra's can optimize real-world scenarios.

6.Future Scalability:
Lay a foundation for scaling to larger networks and incorporating advanced pathfinding or recommendation systems.

FEATURES OF THE MODEL:

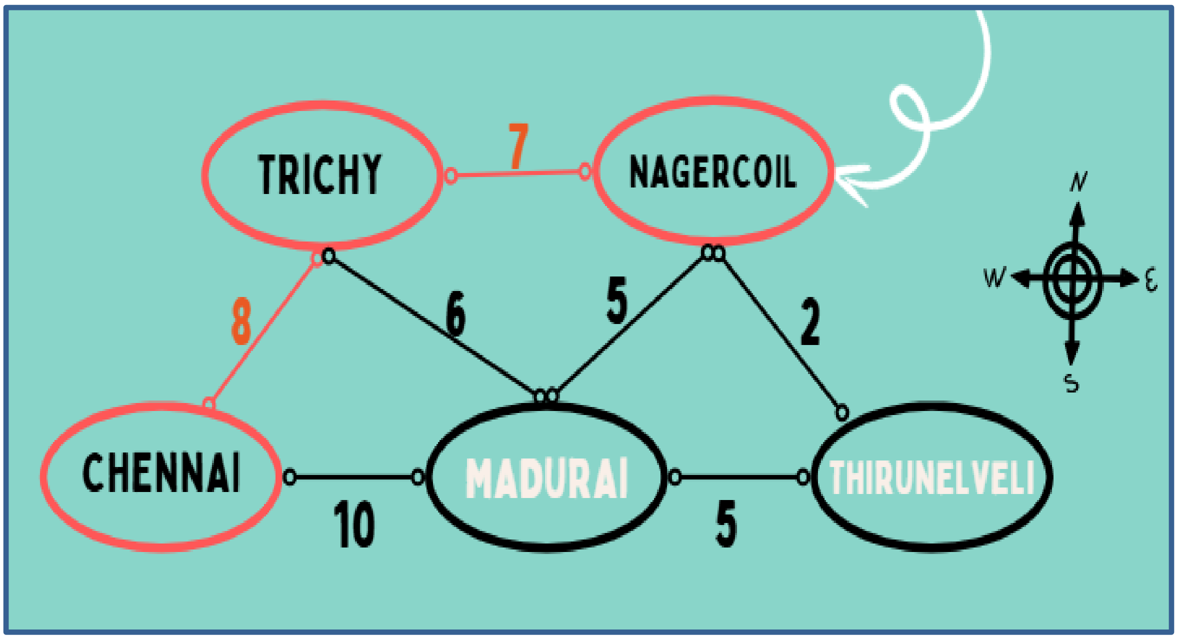
1.Graph Representation:
Uses adjacency matrix to simplify the visualization of connections.

2.Shortest Path Analysis:
Implements Dijkstra's Algorithm for real-time computation of the least-cost path.

3.Dynamic Network Updates:
Allows adding or removing cities and connections interactively.

4.Interactive Output:
Provides clear path tracing and distance computation.

5.Modular Design:
Enables future scalability with additional algorithms or data structure optimizations.



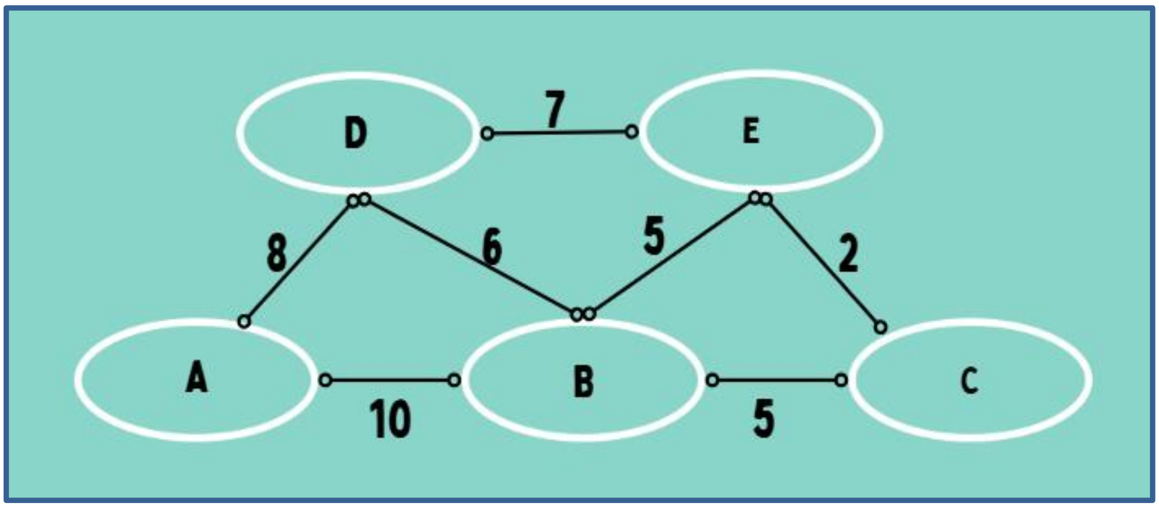
DATA STRUCTURES USED:

1.Adjacency Matrix:

Purpose: Represent connections between cities.

Use Cases:

- Storing distances between cities.
- Input for Dijkstra's Algorithm.



Adjacency Matrix:

V	A	B	C	D	E
A	0	10	0	8	0
B	10	0	5	6	5
C	0	5	0	0	2
D	8	6	0	0	7
E	0	5	2	7	0

2.Integer Arrays:

dist[]: Stores the shortest distance from the source city to other cities.

prev[]: Tracks the predecessor node for reconstructing paths.

visited[]: Tracks which nodes have been processed.

SOURCE CODE:

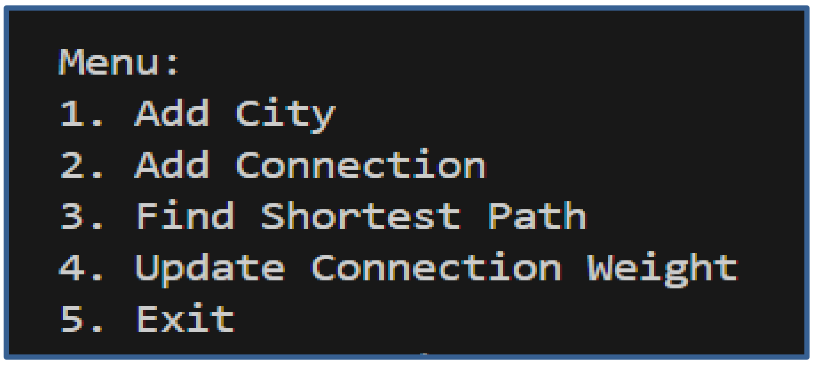
```
#include <stdio.h>
#include <string.h>
#include <limits.h> // For INT_MAX (represents infinity)

// Structure to represent a city
typedef struct {
   char name[20]; // City name
} City;

// Function prototypes
void dijkstra(int n, int cost[10][10], int dist[10], int prev[10], int src_ind);
int find_city_ind(City cities[], int n, char name[]);
void path(int prev[], int dest_ind, City cities[]);

int main() {
   int n = 0; // Number of cities
   int i, j, dist[10], cost[10][10], prev[10]; // Distance, cost matrix, and predecessor array
   char sourceCity[20], destCity[20]; // Source and destination city names
   int distance, src_ind, dest_ind; // Distance, source index, and destination index
   City cities[10]; // Array to store city names

   // Initialize cost matrix with INT_MAX (no direct connections) and 0 for self-loops
   for (i = 0; i < 10; i++) {
      for (j = 0; j < 10; j++) {
         cost[i][j] = (i == j) ? 0 : INT_MAX;
      }
   }

   // Menu-driven program for user interaction
       printf("\nMenu:\n");
      printf("1. Add City\n");
      printf("2. Add Connection\n");
      printf("3. Find Shortest Path\n");
      printf("4. Update Connection Weight\n");
         printf("5. Exit\n");
   while (1) {
      printf("Enter your choice: ");
      int choice;
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            // Add a new city
            if (n >= 10) {
               printf("Maximum number of cities reached.\n");
```

```
    } else {
        printf("Enter city name: ");
        scanf("%s", cities[n].name);
        n++;
        printf("City added successfully.\n");
    }
    break;

case 2:
    // Add a connection between two cities
    if (n < 2) {
        printf("Add at least two cities first.\n");
    } else {
        printf("Enter connection (source city, destination city, distance): ");
        scanf("%s %s %d", sourceCity, destCity, &distance);

        // Find indices of the source and destination cities
        int srcIndex = find_city_ind(cities, n, sourceCity);
        int destIndex = find_city_ind(cities, n, destCity);

        // Validate city names and distance
        if (srcIndex == -1 || destIndex == -1) {
            printf("Invalid city name(s). Please try again.\n");
        } else if (distance <= 0) {
            printf("Invalid distance. Please enter a positive value.\n");
        } else {
            // Update cost matrix for the connection (undirected graph)
            cost[srcIndex][destIndex] = distance;
            cost[destIndex][srcIndex] = distance;
            printf("Connection added successfully.\n");
        }
    }
    break;

case 3:
    // Find the shortest path between two cities
    if (n < 2) {
        printf("Add at least two cities and a connection first.\n");
    } else {
        printf("Enter source city: ");
        scanf("%s", sourceCity);
        src_ind = find_city_ind(cities, n, sourceCity);

        // Validate source city
        if (src_ind == -1) {
            printf("Invalid source city.\n");
            break;
        }
        printf("Enter destination city: ");
        scanf("%s", destCity);
        dest_ind = find_city_ind(cities, n, destCity);

        // Validate destination city
        if (dest_ind == -1) {
            printf("Invalid destination city.\n");
            break;
        }

        // Perform Dijkstra's algorithm
        dijkstra(n, cost, dist, prev, src_ind);

        // Check if a path exists
        if (dist[dest_ind] == INT_MAX) {
            printf("No roads found between %s and %s.\n", sourceCity, destCity);
        } else {
            printf("\nShortest path from %s to %s (distance = %d):\n",
                   sourceCity, destCity, dist[dest_ind]);
            path(prev, dest_ind, cities); // Print the path
            printf("\n");
        }
    }
    break;

case 4:
    // Update the weight of an existing connection
    if (n < 2) {
        printf("Add at least two cities first.\n");
    } else {
        printf("Enter connection to update (source city, destination city, new distance): ");
        scanf("%s %s %d", sourceCity, destCity, &distance);

        // Find indices of the source and destination cities
        int srcIndex = find_city_ind(cities, n, sourceCity);
        int destIndex = find_city_ind(cities, n, destCity);

        // Validate city names and new distance
        if (srcIndex == -1 || destIndex == -1) {
            printf("Invalid city name(s). Please try again.\n");
        } else if (distance <= 0) {
            printf("Invalid distance. Please enter a positive value.\n");
        } else {
            // Update the cost matrix
            cost[srcIndex][destIndex] = distance;
            cost[destIndex][srcIndex] = distance;
            printf("Connection weight updated successfully.\n");
        }
    }
```

```
            break;

         case 5:
            // Exit the program
            printf("Exiting program.\n");
            return 0;

         default:
            // Handle invalid menu choice
            printf("Invalid choice. Please try again.\n");
      }
   }

   return 0;
}

// Function to implement Dijkstra's algorithm
void dijkstra(int n, int cost[10][10], int dist[10], int prev[10], int src_ind) {
   int visited[10] = {0}; // Array to track visited nodes
   int count, i, v, min;

   // Initialize distances and predecessors
   for (i = 0; i < n; i++) {
      dist[i] = cost[src_ind][i];
      prev[i] = (cost[src_ind][i] != INT_MAX && i != src_ind) ? src_ind : -1;
   }
   visited[src_ind] = 1; // Mark the source as visited
   dist[src_ind] = 0; // Distance to the source is 0

   count = 1; // Count of visited nodes

   // Process nodes until all are visited
   while (count < n) {
      min = INT_MAX;
      for (i = 0; i < n; i++) {
         if (!visited[i] && dist[i] < min) {
            min = dist[i];
            v = i; // Nearest unvisited node
         }
      }

      if (min == INT_MAX) break; // No reachable unvisited nodes

      visited[v] = 1; // Mark the nearest node as visited
      count++;

      // Update distances to unvisited nodes
      for (i = 0; i < n; i++) {
         if (!visited[i] && cost[v][i] != INT_MAX && dist[v] + cost[v][i] < dist[i]) {
                dist[i] = dist[v] + cost[v][i];
                prev[i] = v; // Update predecessor
            }
        }
    }
}

// Function to find the index of a city by its name
int find_city_ind(City cities[], int n, char name[]) {
    for (int i = 0; i < n; i++) {
        if (strcmp(cities[i].name, name) == 0) {
            return i; // Return index if found
        }
    }
    return -1; // City not found
}

// Function to print the shortest path recursively
void path(int prev[], int dest_ind, City cities[]) {
    if (prev[dest_ind] == -1) {
        // Base case: no predecessor (starting city)
        printf("%s", cities[dest_ind].name);
        return;
    }
    path(prev, prev[dest_ind], cities); // Recursive call for previous nodes
    printf(" -> %s", cities[dest_ind].name); // Print current city
}
```

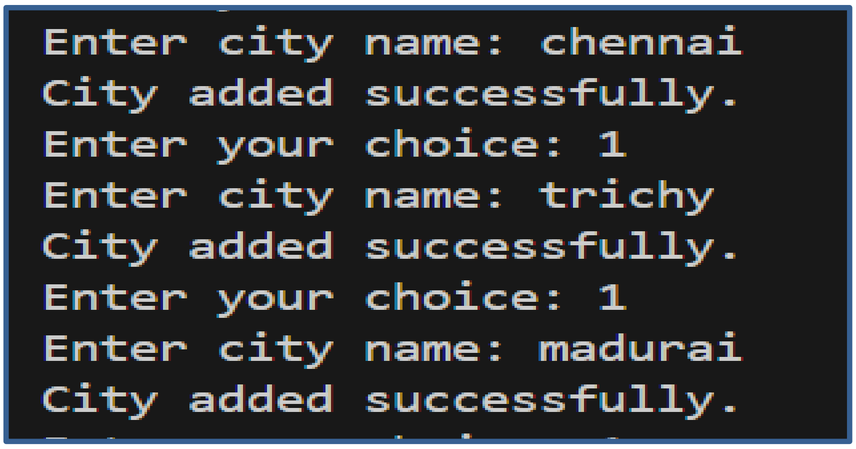
EXECUTION AND OUTPUT:

1. **MENU:**



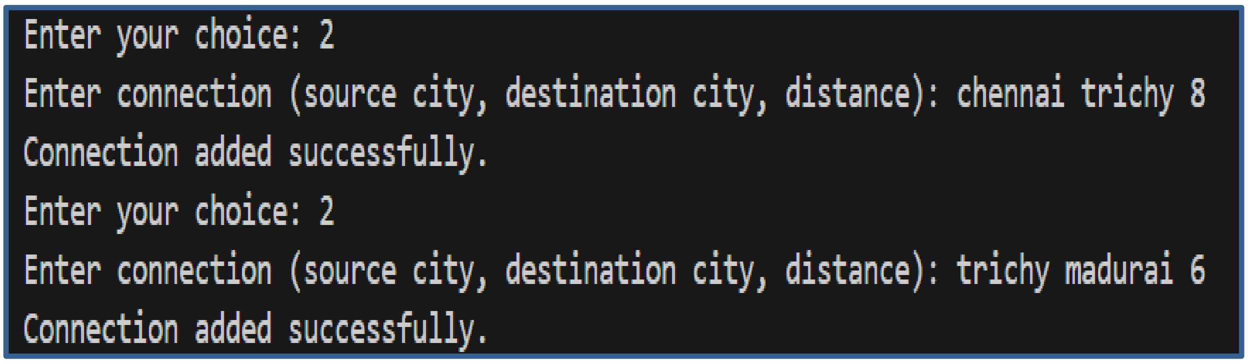
```
Menu:
1. Add City
2. Add Connection
3. Find Shortest Path
4. Update Connection Weight
5. Exit
```

2. **ADD CITIES:**



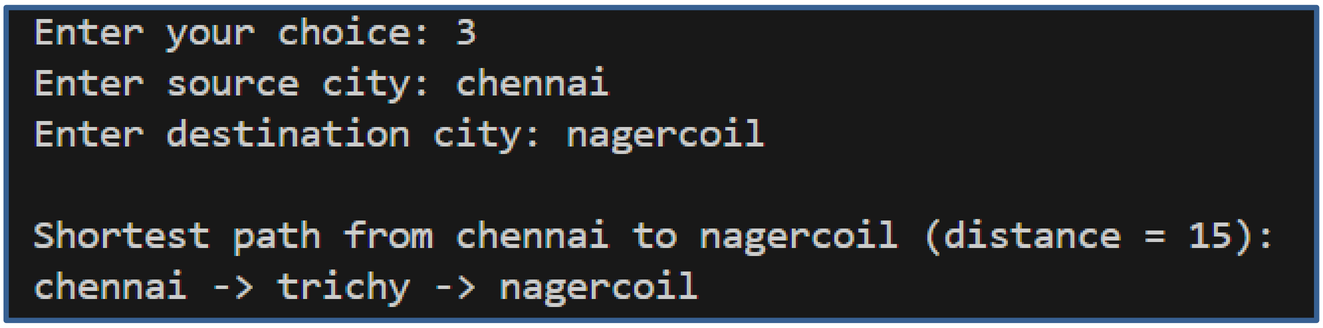
```
Enter city name: chennai
City added successfully.
Enter your choice: 1
Enter city name: trichy
City added successfully.
Enter your choice: 1
Enter city name: madurai
City added successfully.
```

3. **INPUT CONNECTIONS:**



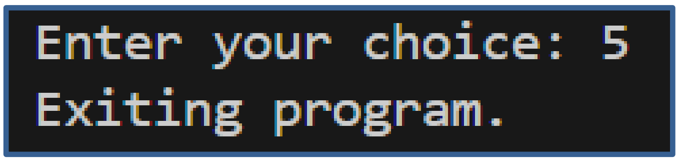
```
Enter your choice: 2
Enter connection (source city, destination city, distance): chennai trichy 8
Connection added successfully.
Enter your choice: 2
Enter connection (source city, destination city, distance): trichy madurai 6
Connection added successfully.
```

4. **FIND THE MOST ADAPTIVE ROUTING FROM SOURCE TO THE DESTINATION:**



```
Enter your choice: 3
Enter source city: chennai
Enter destination city: nagercoil

Shortest path from chennai to nagercoil (distance = 15):
chennai -> trichy -> nagercoil
```

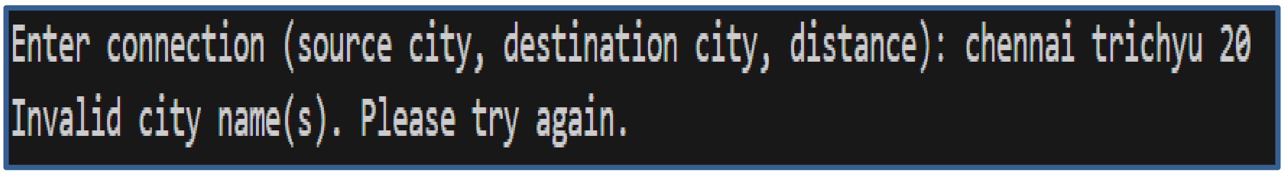
5. **EXIT THE PROGRAM:**



```
Enter your choice: 5
Exiting program.
```

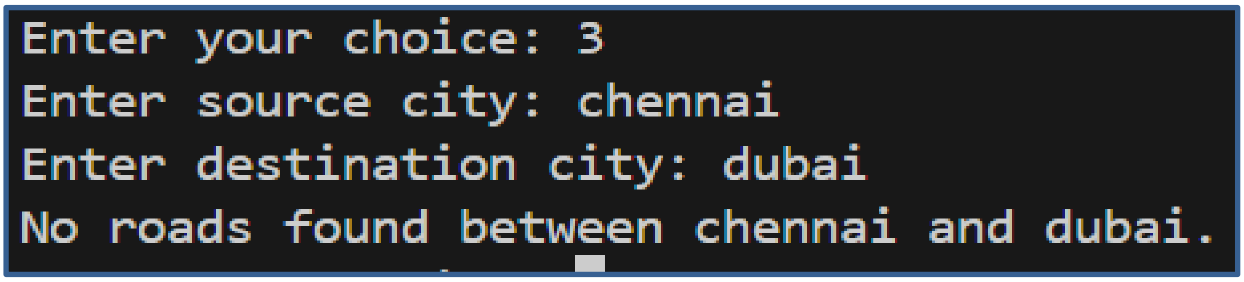
<u>HANDLING EXCEPTIONS:</u>

1. **IF USER ENTERS INVALID CITY NAME**



```
Enter connection (source city, destination city, distance): chennai trichyu 20
Invalid city name(s). Please try again.
```

2. **IF NO CONNECTION OCCURS**



```
Enter your choice: 3
Enter source city: chennai
Enter destination city: dubai
No roads found between chennai and dubai.
```

CONCLUSION:

1. **Input Cities and Connections**

   The program lets you enter city names and the distances between them, creating a network of cities.

2. **Find the Shortest Path**

   Using Dijkstra’s Algorithm, it calculates the shortest path between two cities and shows the total distance and the route.

3. **Handles Disconnected Cities**

   If there’s no path between the cities, it informs the user.

4. **Easy to Use Menu**

   A simple menu lets you add cities, connections, find paths, or exit the program.

5. **Error Checking**

   The program checks for invalid inputs and asks you to correct them.

6. **Real-Life Use**

   It’s useful for solving simple problems like finding the shortest route in a transportation system.

7. **Room for Improvement**

   The program can be extended with features like larger networks, visualizations, and advanced algorithms.

REFERENCES:

- C Programming by Balagurusamy, Schaum series, reema theraja.
- Let us C by yeshwant kanetkar
- Data strucutres by schaum series, weiss.
- GeeksforGeeks
- W3Schools
- Tutorialspoint
- Javatpoint