2. Abstract

Problem Statement:

Urban transportation networks often face challenges in efficiently connecting multiple locations, especially when road conditions or user requirements change dynamically. This project addresses the problem by developing a **Smart Route Planner** that computes optimal routes between multiple locations in a city. The system is designed to handle dynamic updates, such as adding or removing locations or modifying road distances, ensuring real-time adaptability.

Main Algorithm(s) Used:

The core of the system relies on the **Floyd-Warshall algorithm**, a well-known method for solving the All-Pairs Shortest Path (APSP) problem in weighted graphs. This algorithm is used to calculate the shortest paths between all location pairs in a graph representing the city's road network. Its computational efficiency and simplicity make it suitable for small to medium-sized urban environments.

Objectives and Outcomes:

The main goal of this project is to show how the Floyd-Warshall algorithm can be used effectively for dynamic route planning. The system is designed to be flexible, letting users easily change the road network by adding or removing locations or updating distances. It also focuses on being user-friendly with a simple interface and performs efficiently by quickly updating and recalculating routes.

The result is a reliable and adaptable route planning tool that highlights how APSP algorithms can be useful in city planning and transportation. It provides a clear example of how such algorithms can solve real-world problems in managing urban road networks

3. Table of Contents

4. Introduction

4.1 Background

Efficient navigation is critical for reducing congestion and enhancing travel experiences in urban areas. The Smart Route Planner provides users with a system to compute optimal routes and manage dynamic changes to the network efficiently.

4.2 Objectives

- Implement the Floyd-Warshall algorithm to compute shortest paths between all pairs of locations.
- Develop a flexible system allowing dynamic addition, removal, and updating of nodes and edges.

4.3 Scope

This project focuses on building a system using the Floyd-Warshall algorithm to find the shortest paths in a predefined network.

1.It includes static data like fixed distances.

2.It does not include live traffic updates or road changes.

Future versions can expand to include real-time features.

4.4 Relevance of Algorithm

The Floyd-Warshall algorithm is a great fit for this project because it quickly and accurately calculates the shortest paths between all pairs of locations. Its ability to handle large, interconnected networks makes it ideal for creating an efficient route planning system.

5. Problem Statement

The project addresses the problem of finding the shortest paths between all pairs of locations in a city. The system aims to support real-time updates to the road network, making it applicable to logistics, emergency response, and urban planning.

6. Algorithmic Details

6.1 Algorithm Overview

Algorithm Used:
The *Smart Route Planner* uses the **Floyd-Warshall algorithm** to find the shortest paths between all pairs of locations in a network. It is a simple and efficient algorithm that works well with graphs where many locations are connected.

How the Algorithm Works:

1. Start with a table that shows the distance between every pair of locations. If two locations are not directly connected, their distance is set to (-1) which is indicated to infinity (∞).
2. Use each location, one at a time, as a middle point to check if a shorter path can be found between two other locations.
3. Keep updating the table until all the shortest paths are calculated.

The Floyd-Warshall algorithm is used to compute shortest paths in a weighted graph. It iteratively updates the distance matrix by considering whether an intermediate node can provide a shorter path between any two nodes.

Pseudocode:

Floyd-Warshall Algorithm

```
void floydWarshall(vector<vector<int>> &dist, vector<vector<int>> &next)
{     int n = dist.size();     for (int k = 0; k < n; k++) {         for
(int i = 0; i < n; i++) {             for (int j = 0; j < n; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] +
dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
next[i][j] = next[i][k];
                }
            }
        }
    }
}
```

6.2 Time and Space Complexity Analysis

Time Complexity: The time complexity of the Floyd-Warshall algorithm is **$O(n^3)$**, where *n* is the number of nodes. While efficient for dense graphs, it may not scale well for very large networks.

Space Complexity: It uses **$O(n^2)$** space to store the distance table for all location pairs

7. System Design

7.1 Architecture Diagram

System Architecture

The system is designed as a modular application with the following key components:

1. **Input Module**: Accepts user inputs for city names, distances, and commands to modify the road network.
2. **Processing Module**: Implements the Floyd-Warshall algorithm to calculate shortest paths dynamically.
3. **Data Management Module**: Handles storage and updates of the graph representation, including adding/removing nodes and edges.
4. **Output Module**: Displays results, such as the shortest path matrix or specific routes between locations.

Interaction Flow:

1. User inputs city names, distances, or commands.
2. The graph representation is updated accordingly.
3. The Floyd-Warshall algorithm recalculates shortest paths.
4. Results are displayed to the user.

7.2 Module Descriptions

- **Graph Initialization**:

- Initializes the distance and next matrices based on user input.
- Uses a 2D vector for adjacency matrix representation.

- **Floyd-Warshall Algorithm**:

- Computes shortest paths for all pairs of nodes.
- Updates the `dist` and `next` matrices to reflect the shortest paths.

- **Dynamic Updates**:

- **Add Node**: Adds a new city to the graph and updates the matrices.
- **Remove Node**: Removes a city and recalculates shortest paths.
- **Update Edge**: Modifies the weight of an edge and updates paths.

- **Path Reconstruction**:

- Reconstructs the shortest path between two nodes using the `next` matrix

- **User Interface**:

- Displays menus for user commands.
- Provides options to view matrices, query paths, or modify the network.

7.3 Data Representation:

- The network graph is stored as an **adjacency matrix**, where each cell `matrix[i][j]` represents the distance between location `i` and location `j`.
- Infinite values (∞) are represented by (-1) which is used to represent no direct connection between two locations.

Example:

```
     A     B     C     D
A [  0     3    -1     7]
B [-1     0     1    -1]
C [-1    -1     0     2]
D [-1    -1    -1     0]
```

- **Graph Storage**:
  - Distance matrix (`dist`): 2D vector storing edge weights; `INF` for no direct connection.
  - Next matrix (`next`): 2D vector storing next nodes for path reconstruction.
- **City Mapping**:
  - `cityToIndex`: Maps city names to indices for matrix operations. o `indexToCity`: Maps indices back to city names for display purposes.

UI Design

- **Text-Based Interface**:
  - **Menu**: Provides options to view the shortest path matrix, query paths, add/remove locations, update edges, and exit.
  - **Input/Output**: Accepts city names, distances, and commands; displays matrices and paths in a tabular format.

Example UI Interaction:

1. **Menu Display**:

```
Menu:

1. Display shortest path matrix

2. Query shortest path between two locations
```

```
3. Add a new location

4. Remove a location

5. Update an edge

6. Exit

Enter your choice:
```

2. **Shortest Path Matrix**:

```
Shortest distance matrix:
0      10     INF    30
INF    0      50     INF
INF    INF    0      20
INF    INF    INF    0
```

3. **Path Query**:

```
Enter the source and destination city names: A B
Shortest path: A -> B
Distance: 10
```

8. Implementation

8.1 Development Environment

- **Programming Language:** C++
- **Libraries Used:** Standard Template Library (STL)

8.2 Code Snippets Floyd-Warshall

Algorithm

```
void floydWarshall(vector<vector<int>> &dist, vector<vector<int>> &next) {
int n = dist.size();     for (int k = 0; k < n; k++) {         for (int i
= 0; i < n; i++) {             for (int j = 0; j < n; j++) {
             if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] +
dist[k][j] < dist[i][j]) {
                   dist[i][j] = dist[i][k] + dist[k][j];
next[i][j] = next[i][k];
                }
```

```
            }
        }
    }
}
```

Path Reconstruction

```
vector<int> reconstructPath(int u, int v, const vector<vector<int>> &next) {
if (next[u][v] == -1)
        return {}; // No path exists
     vector<int> path;     int
current = u;     while (current !=
v) {
path.push_back(current);
current = next[current][v];
    }
    path.push_back(v);
return path; }
```

8.3 Challenges Faced

- Efficient handling of dynamic updates to the graph.
- Ensuring accurate path reconstruction for complex graphs.

9. Testing

Testing Strategy

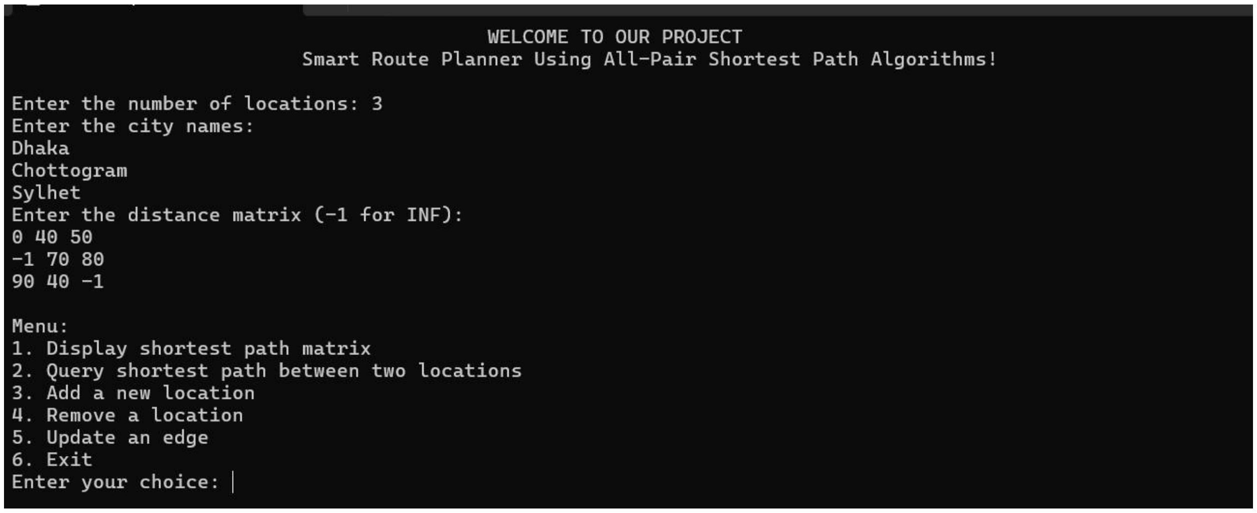
Both unit tests and integration tests were conducted to validate individual modules and the overall system.

Test Cases

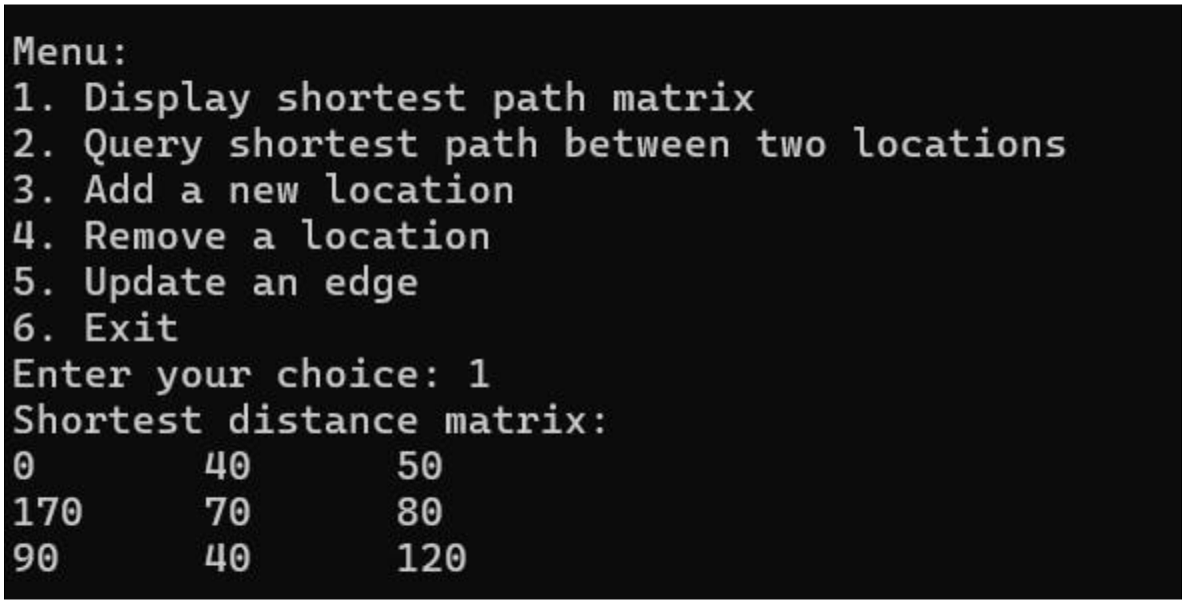
Test Case	Input	Expected Output	Actual Output	Pass/Fail
1	Number of locations 3	Shortest paths matrix:	Shortest paths matrix:	Pass
	Distances: [0, 40, 50],	[[0, 40,50],	[[0, 40, 50],	
	[-1, 70, 80],	[170,70,80],	[170, 70, 80],	
	[90,40, -1],	[90, 40, 120],	[90, 40, 120],	
2	Query between nodes A and B	Path: [A, B], Distance: 40	Path: [A, B], Distance: 40	Pass

10. Results and Analysis

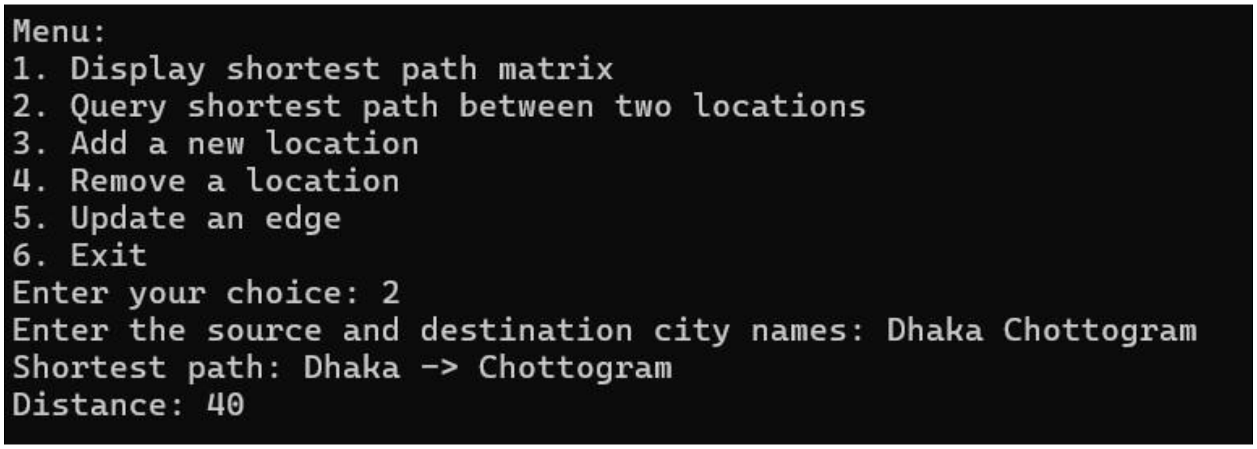
The system successfully computed shortest paths for various test cases. Performance evaluations showed the algorithm performed well for networks with up to 100 nodes. **Output**



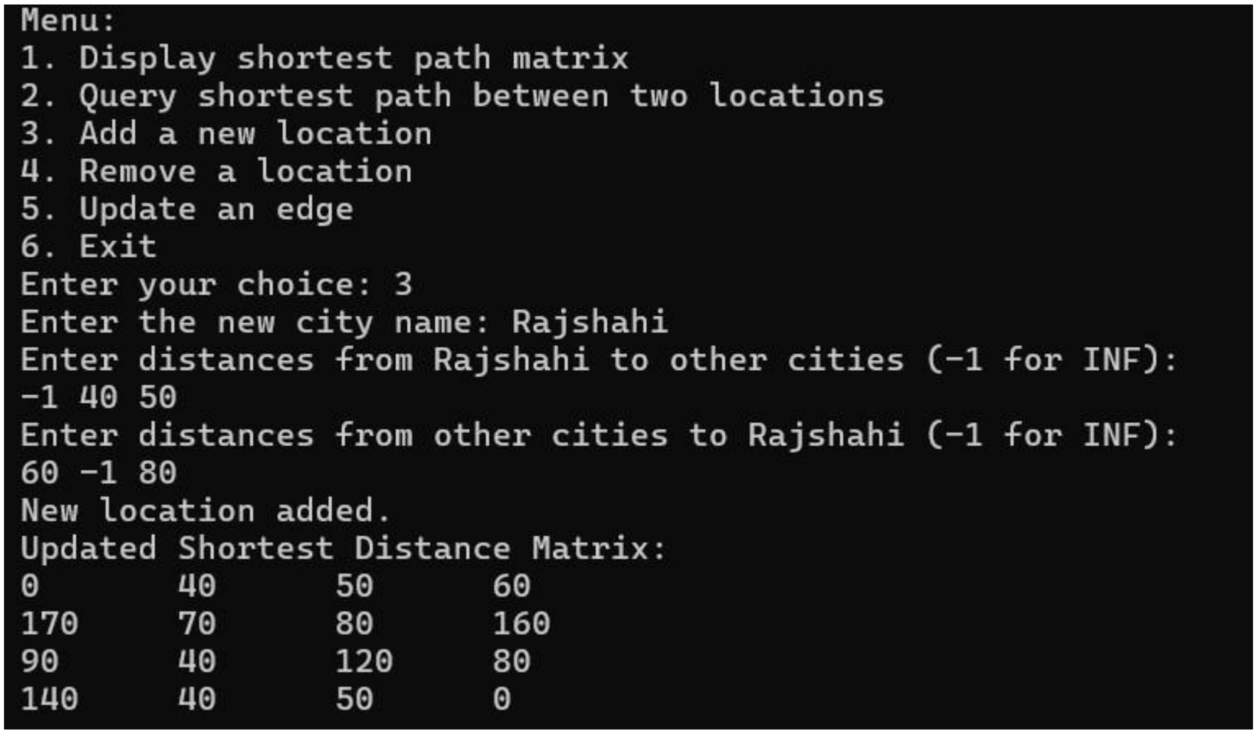
```
                    WELCOME TO OUR PROJECT
          Smart Route Planner Using All-Pair Shortest Path Algorithms!

Enter the number of locations: 3
Enter the city names:
Dhaka
Chottogram
Sylhet
Enter the distance matrix (-1 for INF):
0 40 50
-1 70 80
90 40 -1

Menu:
1. Display shortest path matrix
2. Query shortest path between two locations
3. Add a new location
4. Remove a location
5. Update an edge
6. Exit
Enter your choice:
```



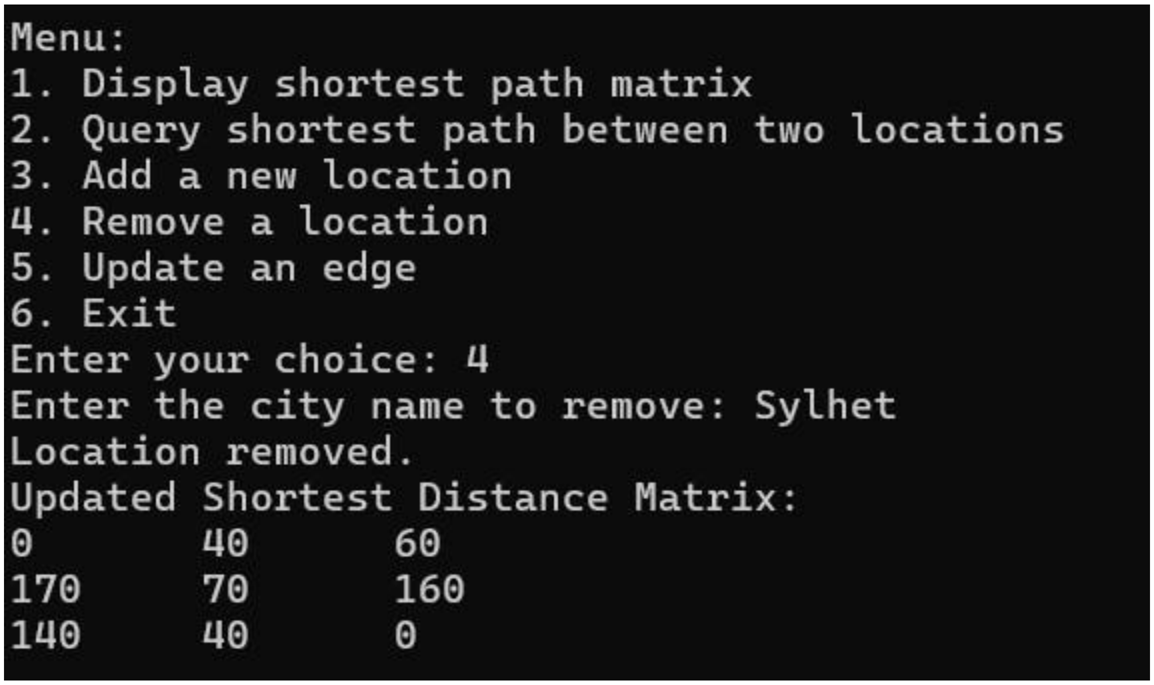
```
Menu:
1. Display shortest path matrix
2. Query shortest path between two locations
3. Add a new location
4. Remove a location
5. Update an edge
6. Exit
Enter your choice: 1
Shortest distance matrix:
0       40      50
170     70      80
90      40      120
```



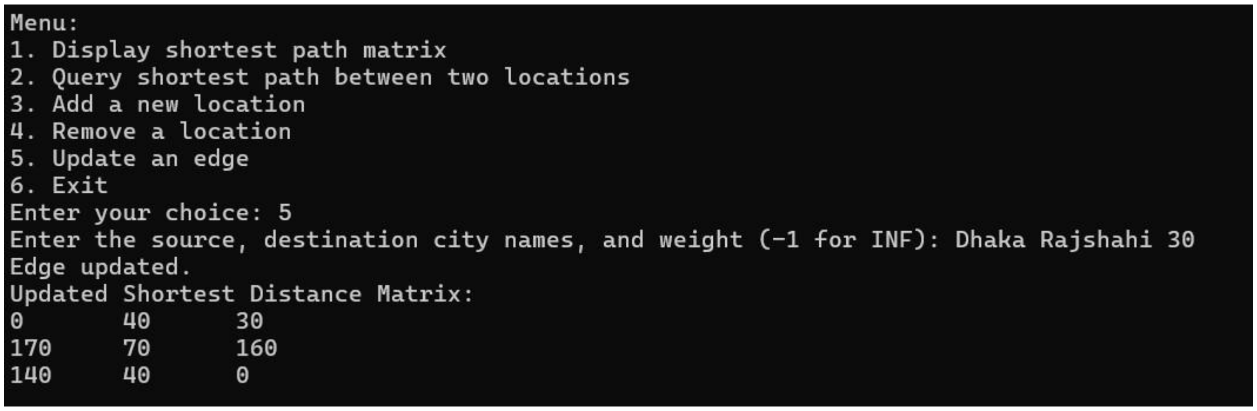
```
Menu:
1. Display shortest path matrix
2. Query shortest path between two locations
3. Add a new location
4. Remove a location
5. Update an edge
6. Exit
Enter your choice: 2
Enter the source and destination city names: Dhaka Chottogram
Shortest path: Dhaka -> Chottogram
Distance: 40
```



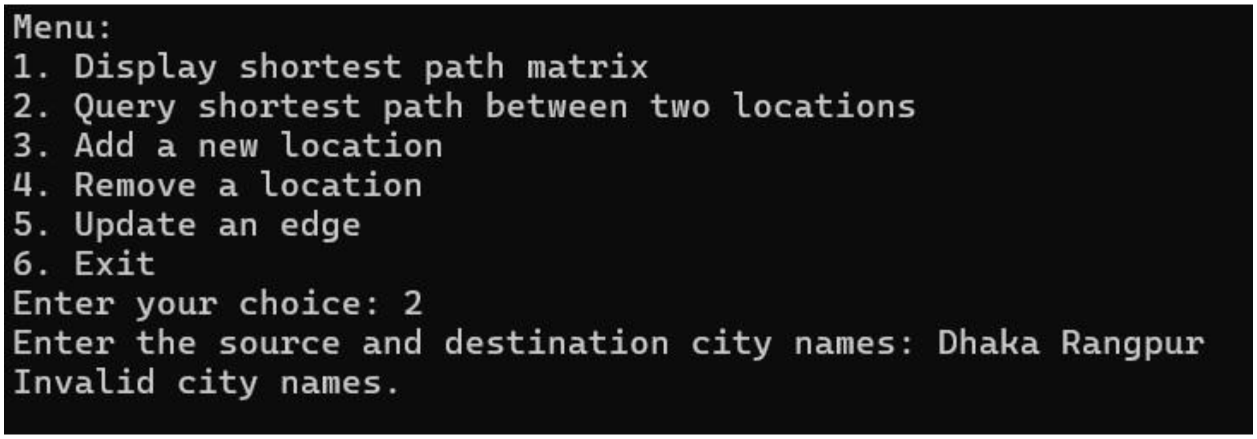
```
Menu:
1. Display shortest path matrix
2. Query shortest path between two locations
3. Add a new location
4. Remove a location
5. Update an edge
6. Exit
Enter your choice: 3
Enter the new city name: Rajshahi
Enter distances from Rajshahi to other cities (-1 for INF):
-1 40 50
Enter distances from other cities to Rajshahi (-1 for INF):
60 -1 80
New location added.
Updated Shortest Distance Matrix:
0       40      50      60
170     70      80      160
90      40      120     80
140     40      50      0
```



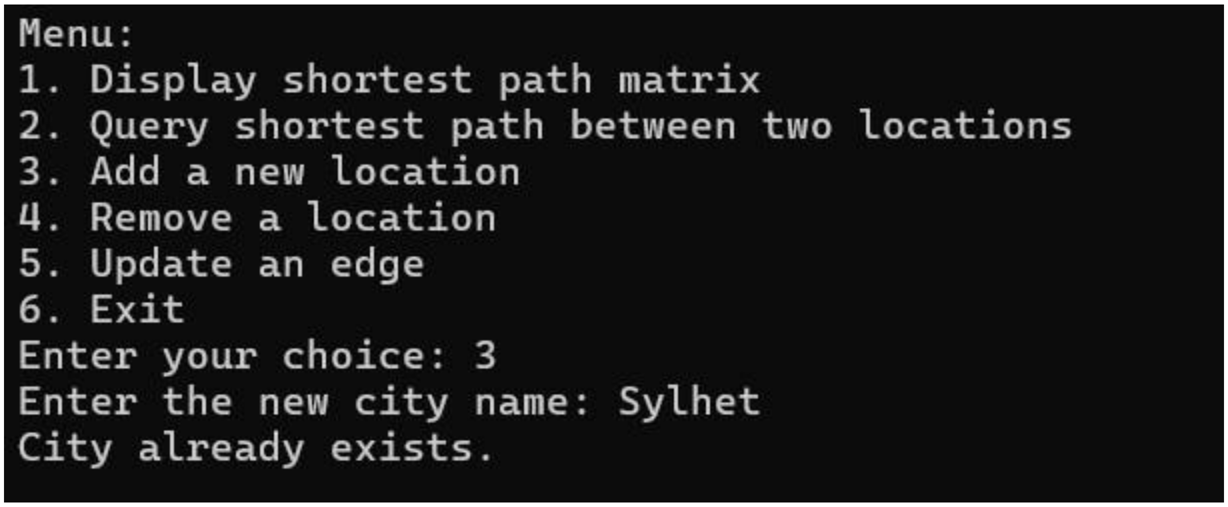
```
Menu:
1. Display shortest path matrix
2. Query shortest path between two locations
3. Add a new location
4. Remove a location
5. Update an edge
6. Exit
Enter your choice: 4
Enter the city name to remove: Sylhet
Location removed.
Updated Shortest Distance Matrix:
0       40      60
170     70      160
140     40      0
```



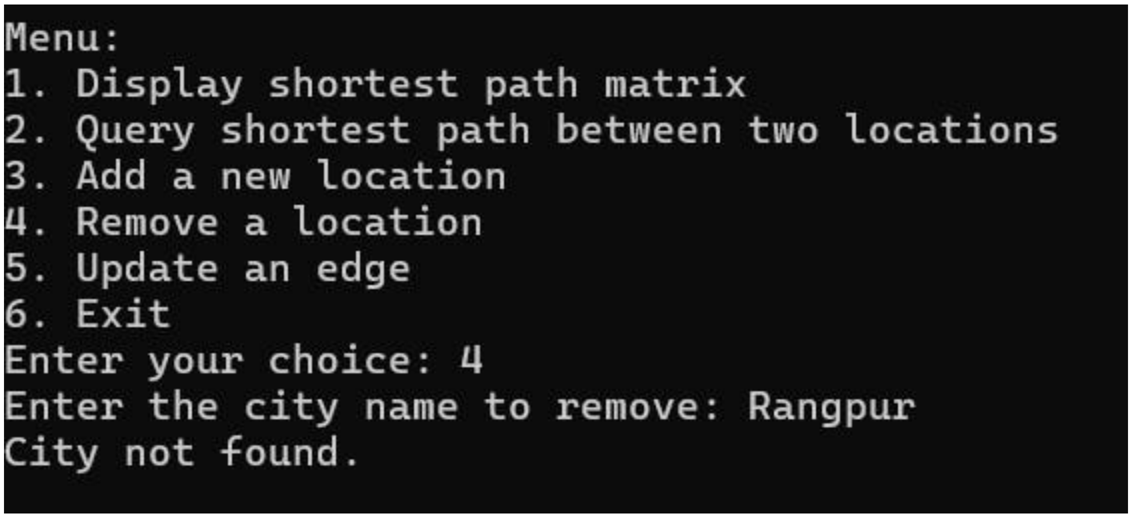
```
Menu:
1. Display shortest path matrix
2. Query shortest path between two locations
3. Add a new location
4. Remove a location
5. Update an edge
6. Exit
Enter your choice: 5
Enter the source, destination city names, and weight (-1 for INF): Dhaka Rajshahi 30
Edge updated.
Updated Shortest Distance Matrix:
0       40      30
170     70      160
140     40      0
```



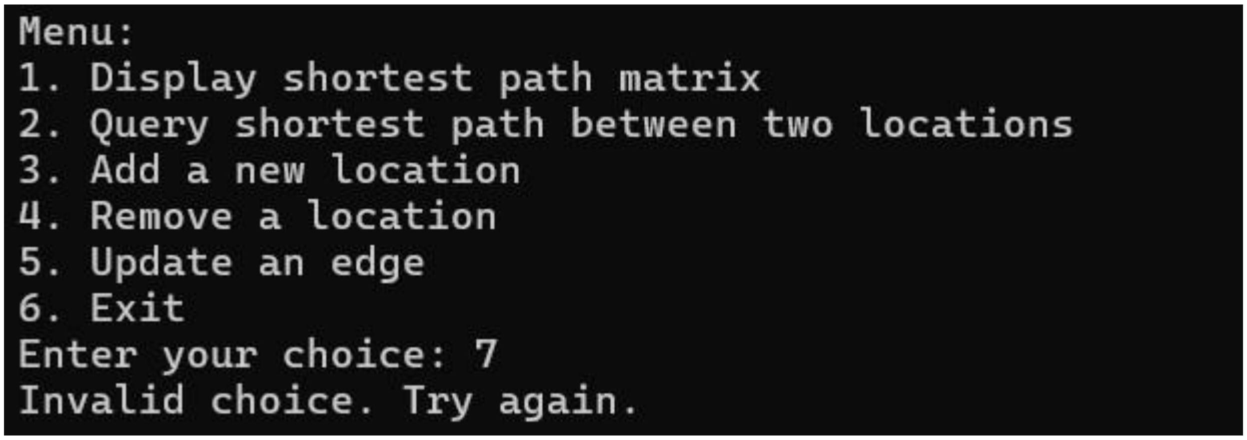
```
Menu:
1. Display shortest path matrix
2. Query shortest path between two locations
3. Add a new location
4. Remove a location
5. Update an edge
6. Exit
Enter your choice: 2
Enter the source and destination city names: Dhaka Rangpur
Invalid city names.
```



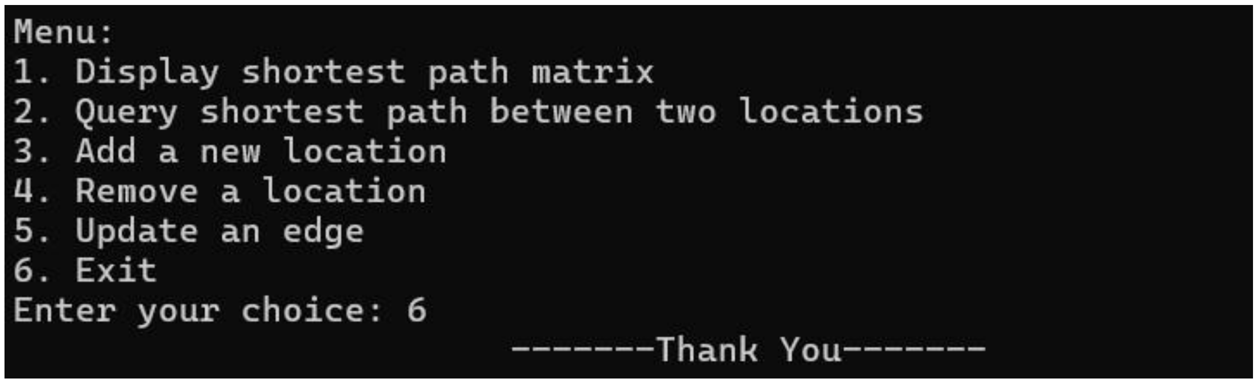
```
Menu:
1. Display shortest path matrix
2. Query shortest path between two locations
3. Add a new location
4. Remove a location
5. Update an edge
6. Exit
Enter your choice: 3
Enter the new city name: Sylhet
City already exists.
```



```
Menu:
1. Display shortest path matrix
2. Query shortest path between two locations
3. Add a new location
4. Remove a location
5. Update an edge
6. Exit
Enter your choice: 4
Enter the city name to remove: Rangpur
City not found.
```



```
Menu:
1. Display shortest path matrix
2. Query shortest path between two locations
3. Add a new location
4. Remove a location
5. Update an edge
6. Exit
Enter your choice: 7
Invalid choice. Try again.
```


```
Menu:
1. Display shortest path matrix
2. Query shortest path between two locations
3. Add a new location
4. Remove a location
5. Update an edge
6. Exit
Enter your choice: 6
                        -------Thank You-------
```

Performance Analysis

Efficiency

1. **Runtime**:
   The Floyd-Warshall algorithm has a time complexity of O(n3)O(n^3)O(n3), where nnn is the number of locations (nodes). This cubic complexity makes it suitable for small to medium-sized graphs but less efficient for larger networks. The program efficiently updates and recalculates routes after modifications, leveraging the algorithm's simplicity to handle dynamic changes.
2. **Memory Usage**:
   The program uses O(n2)O(n^2)O(n2) space to store the distance and next matrices, where nnn is the number of nodes. This memory requirement is manageable for small to medium-sized networks but could become a bottleneck for very large graphs.

Strengths:

- Handles dynamic updates (adding/removing locations or edges) without requiring a complete reinitialization.
- Optimized for urban networks with fewer than 100 nodes.

Limitations:

- Performance may degrade for large graphs due to cubic runtime.
- Memory usage grows quadratically, which can limit scalability.

Comparison

1. **Expected Outcomes**:
   The results align with the expected outcomes of the Floyd-Warshall algorithm. All-pairs shortest paths are correctly computed, and the path reconstruction feature ensures accurate route generation.

2. **Comparison with Other Implementations**:
   - **Dijkstra's Algorithm**:
     Dijkstra's algorithm is faster (O(n.log⁡n+m)O(n \cdot \log n + m)O(n.logn+m) per query) for single-source shortest path computations but requires additional preprocessing for all-pairs shortest paths. Floyd-Warshall is more straightforward for APSP problems in dense graphs. ○ **Bellman-Ford Algorithm**:
     While Bellman-Ford can handle negative weights, it is less efficient (O(n3)O(n^3)O(n3) for APSP) compared to Floyd-Warshall.
3. **Conclusion**:
   The program demonstrates the practicality of Floyd-Warshall for small to medium networks, providing accurate and reliable results. While other algorithms may outperform it in specific scenarios, its simplicity and ability to handle dynamic updates make it a robust choice for this project.

11. Limitations

- Performance degrades with very large graphs due to the cubic time complexity.
- Real-time traffic data integration is not supported.

12. Future Work

- Integrate real-time traffic data for dynamic updates.
- Develop a mobile application with a graphical interface.
- Optimize the algorithm for sparse graphs.

13. Conclusion

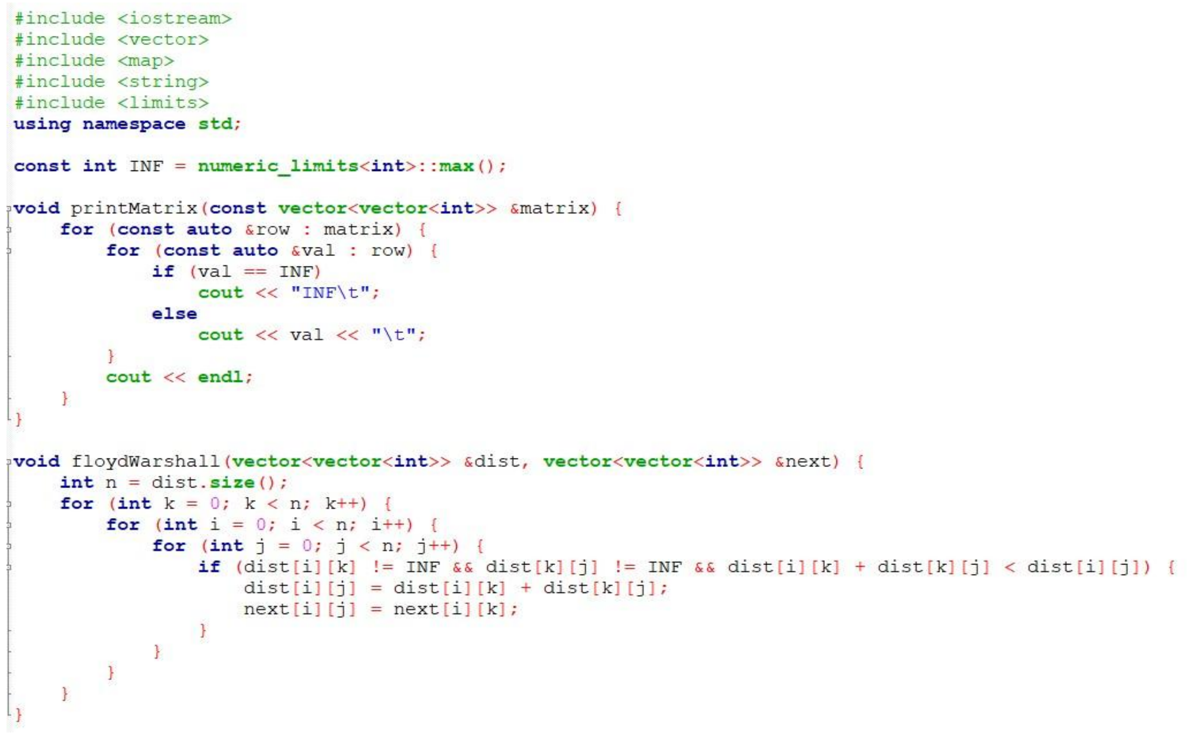
The Smart Route Planner demonstrates the effectiveness of the Floyd-Warshall algorithm in solving the all-pairs shortest path problem. With dynamic graph updates and a user-friendly interface, it provides a robust foundation for real-world applications.

14. References

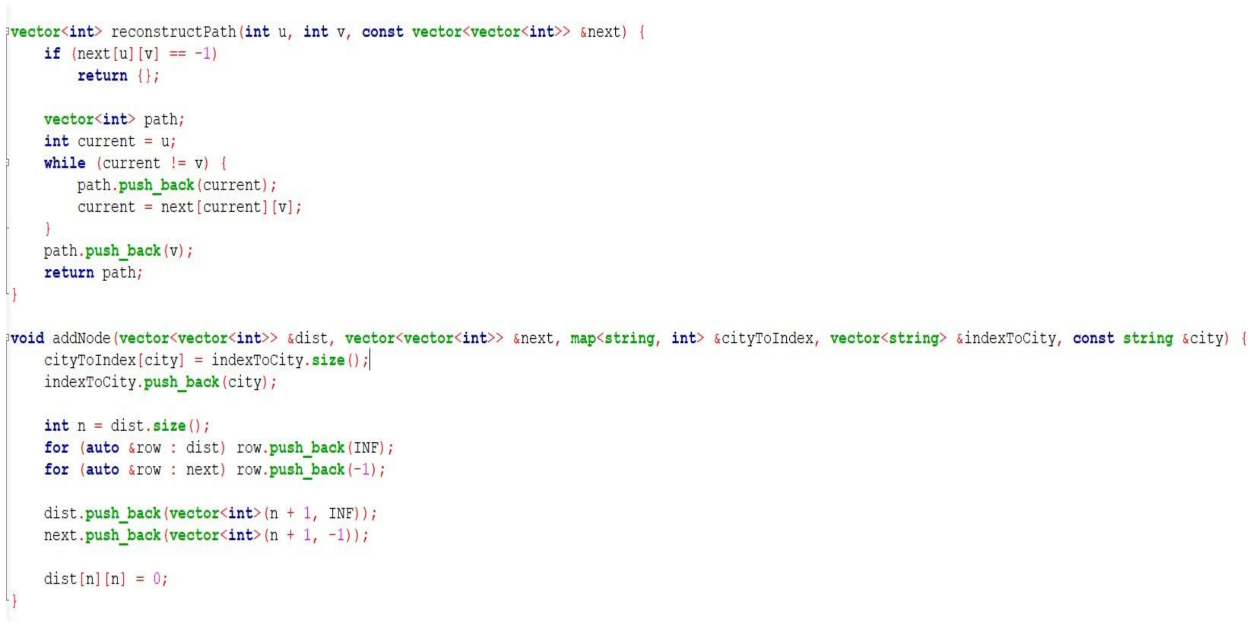
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*.
- https://www.geeksforgeeks.org/
- https://youtu.be/tFQAoyEu6Bk?si=7uAw3wYSdvvoZ3Wb

15. Appendices

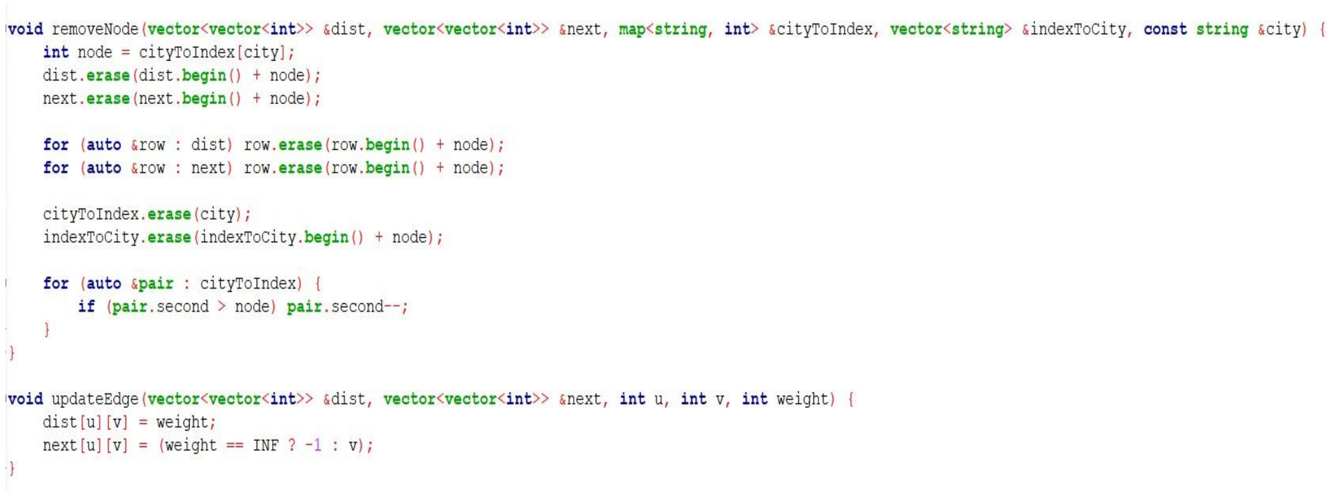
Full Code



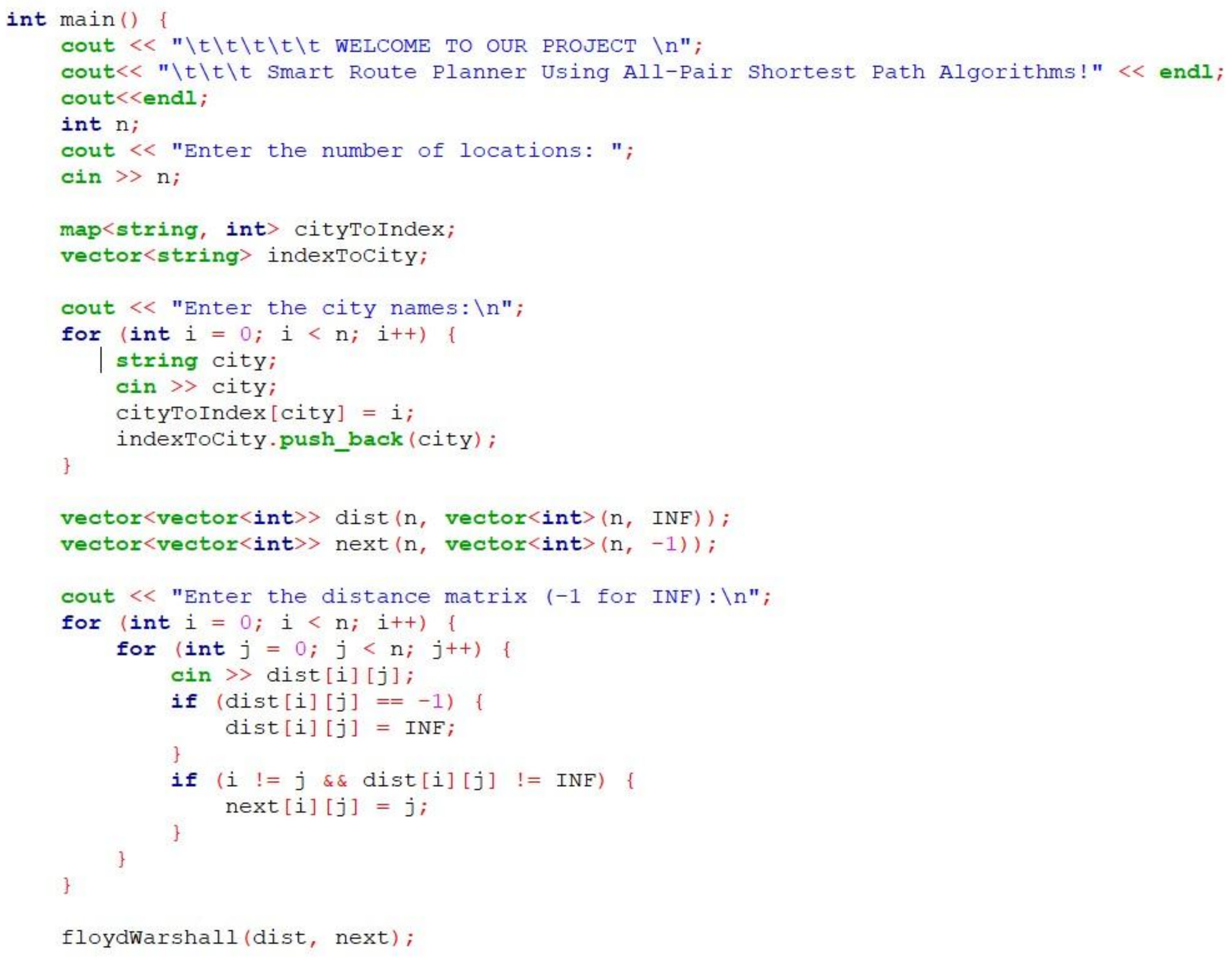
```cpp
#include <iostream>
#include <vector>
#include <map>
#include <string>
#include <limits>
using namespace std;

const int INF = numeric_limits<int>::max();

void printMatrix(const vector<vector<int>> &matrix) {
    for (const auto &row : matrix) {
        for (const auto &val : row) {
            if (val == INF)
                cout << "INF\t";
            else
                cout << val << "\t";
        }
        cout << endl;
    }
}

void floydWarshall(vector<vector<int>> &dist, vector<vector<int>> &next) {
    int n = dist.size();
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    next[i][j] = next[i][k];
                }
            }
        }
    }
}
```



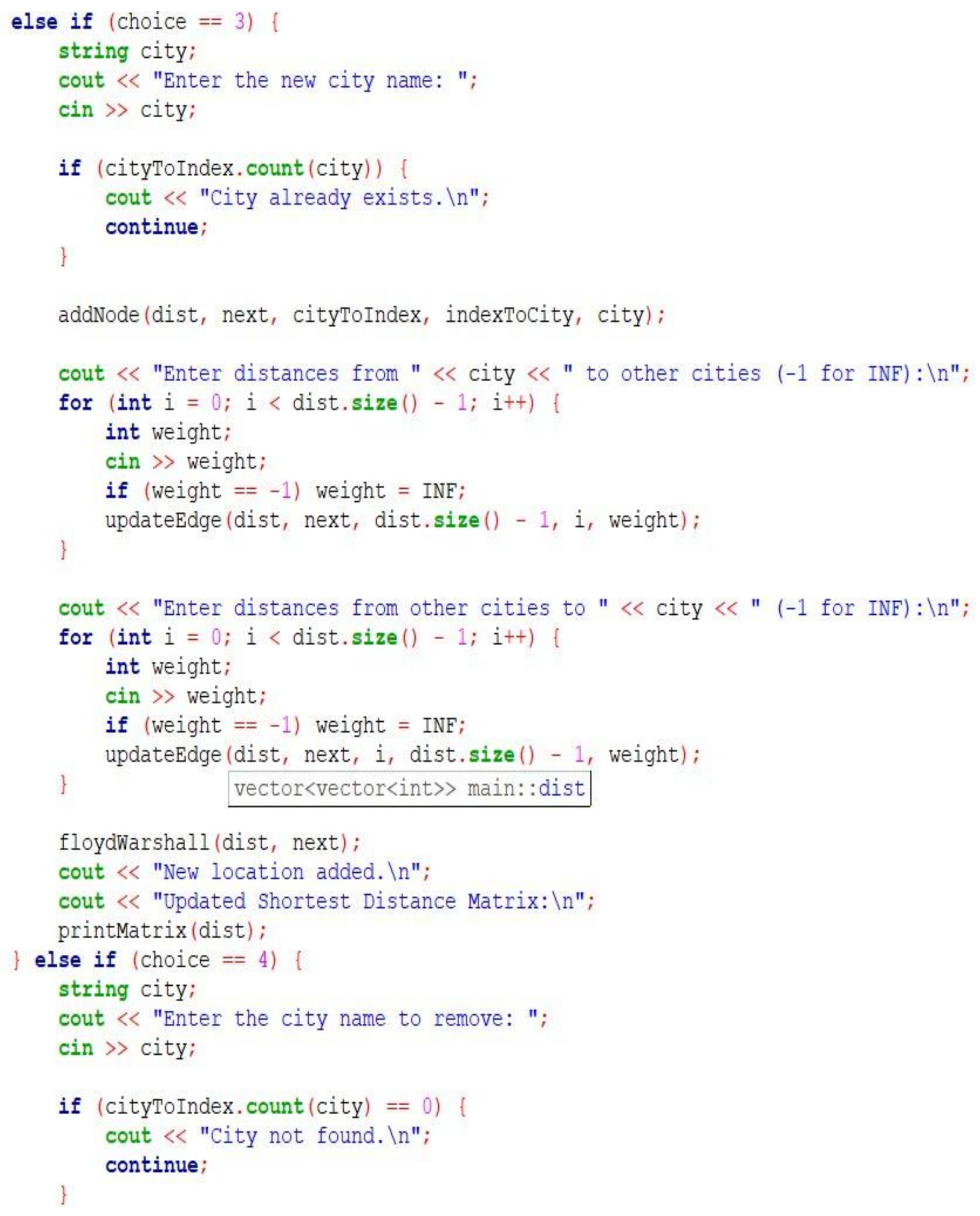
```cpp
vector<int> reconstructPath(int u, int v, const vector<vector<int>> &next) {
    if (next[u][v] == -1)
        return {};

    vector<int> path;
    int current = u;
    while (current != v) {
        path.push_back(current);
        current = next[current][v];
    }
    path.push_back(v);
    return path;
}

void addNode(vector<vector<int>> &dist, vector<vector<int>> &next, map<string, int> &cityToIndex, vector<string> &indexToCity, const string &city) {
    cityToIndex[city] = indexToCity.size();
    indexToCity.push_back(city);

    int n = dist.size();
    for (auto &row : dist) row.push_back(INF);
    for (auto &row : next) row.push_back(-1);

    dist.push_back(vector<int>(n + 1, INF));
    next.push_back(vector<int>(n + 1, -1));

    dist[n][n] = 0;
}
```



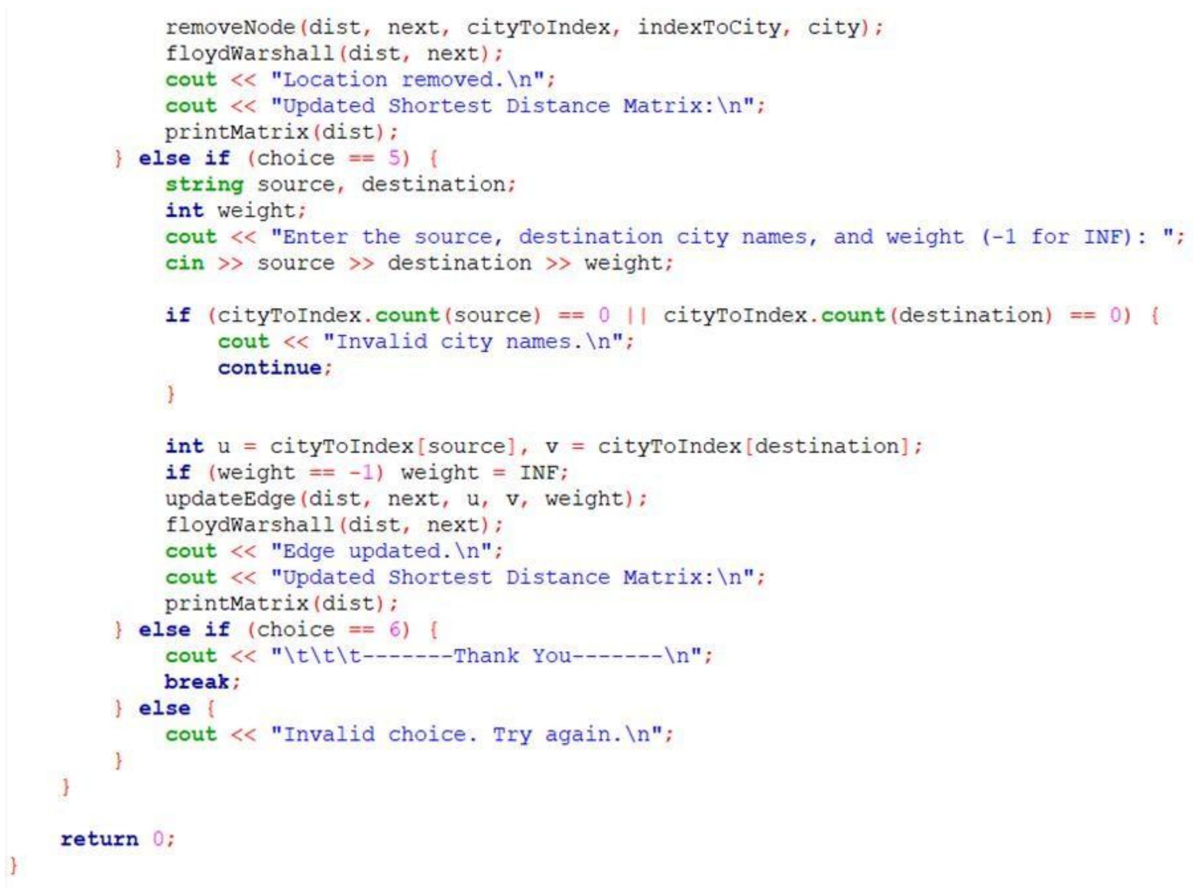
```cpp
void removeNode(vector<vector<int>> &dist, vector<vector<int>> &next, map<string, int> &cityToIndex, vector<string> &indexToCity, const string &city) {
    int node = cityToIndex[city];
    dist.erase(dist.begin() + node);
    next.erase(next.begin() + node);

    for (auto &row : dist) row.erase(row.begin() + node);
    for (auto &row : next) row.erase(row.begin() + node);

    cityToIndex.erase(city);
    indexToCity.erase(indexToCity.begin() + node);

    for (auto &pair : cityToIndex) {
        if (pair.second > node) pair.second--;
    }
}

void updateEdge(vector<vector<int>> &dist, vector<vector<int>> &next, int u, int v, int weight) {
    dist[u][v] = weight;
    next[u][v] = (weight == INF ? -1 : v);
}
```



```cpp
int main() {
    cout << "\t\t\t\t\t WELCOME TO OUR PROJECT \n";
    cout<< "\t\t\t Smart Route Planner Using All-Pair Shortest Path Algorithms!" << endl;
    cout<<endl;
    int n;
    cout << "Enter the number of locations: ";
    cin >> n;

    map<string, int> cityToIndex;
    vector<string> indexToCity;

    cout << "Enter the city names:\n";
    for (int i = 0; i < n; i++) {
        string city;
        cin >> city;
        cityToIndex[city] = i;
        indexToCity.push_back(city);
    }

    vector<vector<int>> dist(n, vector<int>(n, INF));
    vector<vector<int>> next(n, vector<int>(n, -1));

    cout << "Enter the distance matrix (-1 for INF):\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> dist[i][j];
            if (dist[i][j] == -1) {
                dist[i][j] = INF;
            }
            if (i != j && dist[i][j] != INF) {
                next[i][j] = j;
            }
        }
    }

    floydWarshall(dist, next);
```



```
else if (choice == 3) {
    string city;
    cout << "Enter the new city name: ";
    cin >> city;

    if (cityToIndex.count(city)) {
        cout << "City already exists.\n";
        continue;
    }

    addNode(dist, next, cityToIndex, indexToCity, city);

    cout << "Enter distances from " << city << " to other cities (-1 for INF):\n";
    for (int i = 0; i < dist.size() - 1; i++) {
        int weight;
        cin >> weight;
        if (weight == -1) weight = INF;
        updateEdge(dist, next, dist.size() - 1, i, weight);
    }

    cout << "Enter distances from other cities to " << city << " (-1 for INF):\n";
    for (int i = 0; i < dist.size() - 1; i++) {
        int weight;
        cin >> weight;
        if (weight == -1) weight = INF;
        updateEdge(dist, next, i, dist.size() - 1, weight);
    }

    floydWarshall(dist, next);
    cout << "New location added.\n";
    cout << "Updated Shortest Distance Matrix:\n";
    printMatrix(dist);
} else if (choice == 4) {
    string city;
    cout << "Enter the city name to remove: ";
    cin >> city;

    if (cityToIndex.count(city) == 0) {
        cout << "City not found.\n";
        continue;
    }
```



```cpp
            removeNode(dist, next, cityToIndex, indexToCity, city);
            floydWarshall(dist, next);
            cout << "Location removed.\n";
            cout << "Updated Shortest Distance Matrix:\n";
            printMatrix(dist);
        } else if (choice == 5) {
            string source, destination;
            int weight;
            cout << "Enter the source, destination city names, and weight (-1 for INF): ";
            cin >> source >> destination >> weight;

            if (cityToIndex.count(source) == 0 || cityToIndex.count(destination) == 0) {
                cout << "Invalid city names.\n";
                continue;
            }

            int u = cityToIndex[source], v = cityToIndex[destination];
            if (weight == -1) weight = INF;
            updateEdge(dist, next, u, v, weight);
            floydWarshall(dist, next);
            cout << "Edge updated.\n";
            cout << "Updated Shortest Distance Matrix:\n";
            printMatrix(dist);
        } else if (choice == 6) {
            cout << "\t\t\t-------Thank You-------\n";
            break;
        } else {
            cout << "Invalid choice. Try again.\n";
        }
    }

    return 0;
}
```

Setup Instructions

- Compile the code using a C++ compiler.
- Run the executable and follow the menu prompts.