# UNIVERSITY OF GONDAR

# COLLAGE OF INFORMATICS

# DEPARTMENT OF COMPUTER SCIENCE

## DSA PROJECT

# TITLE: TRAVEL PLANNER


| GROUP MEMBER | ID |
|---|---|
| 1. AMANUAL AZANAW | 02595/15 |
| 2. NATNAEL GETNET | 01640/15 |
| 3. ASHENAFI HABTE | 02264/15 |
| 4. GETAHUN NIGUSSIE | 02621/15 |
| 5. HILINA MEKURIAW | 01729/15 |


**SUBMITTE TO**: Mr. Kibret

**SUBMISSION DATE**:25/07/2017 E.C

Table of Contents

1. Introduction

The Travel Planner System is a C++ program designed to model a network of travel locations (nodes) and routes (edges) with weighted distances. It provides functionalities for:Adding, removing, and updating locations and route

-Graph traversals (BFS & DFS)

-Finding the shortest path using Dijkstra’s algorithm

-Displaying the complete travel network

-This documentation provides a detailed breakdown of the implementation, algorithms, and usage of the system.

2. System Overview

The system is built using:C++ Standard
Library (<unordered_map>, <vector>, <queue>, <stack>, <algorithm>)

Graph Representation: Adjacency list (unordered_map<string, vector<pair<string, int>>>)Menu-Driven Interface: User interacts via a console-based menuKey FeaturesUndirected Graph: Routes are bidirectional (A ↔ B)Dynamic Updates: Locations and routes can be modified at runtime

Pathfinding: Dijkstra’s algorithm for shortest path

Traversal Methods: BFS & DFS for exploration

3. Data Structures & Design

3.1 Graph Representation

unordered_map<string, vector<pair<string, int>>> graph;

Key: string (Location name)

Value: vector<pair<string, int>> (List of connected locations and weights)

3.2 Supporting Data Structures

queue<string> (BFS traversal)

stack<string> (DFS traversal)

priority_queue<pair<int, string>> (Dijkstra’s algorithm)

unordered_map<string, int> (Distance tracking) unordered_map<string, string> (Path reconstruction)

4. Class & Method Descriptions

4.1 TravelPlanner Class

Manages the travel network and provides core functionalities.

4.1.1 Location Management

| Method | Description | Parameters | Return |
|---|---|---|---|
| addLocation() | Adds a new location | const string& location | void |
| removeLocation() | Removes a location and all connected routes | const string& location | void |

4.1.2 Route Management

| Method | Description | Parameters | Return |
|---|---|---|---|
| addRoute() | Adds a bidirectional route | src, dest, weight | void |
| updateRoute() | Modifies route weight | src, dest, newWeight | void |
| removeRoute() | Deletes a route | src, dest | void |

4.1.3 Graph Traversals

| Method | Description | Parameters | Return |
|---|---|---|---|
| bfsTraversal() | Breadth-First Search | startLocation | void |
| dfsTraversal() | Depth-First Search | startLocation | void |

4.1.4 Pathfinding

| Method | Description | Parameters | Return |
|---|---|---|---|
| findShortestPath() | Dijkstra’s algorithm | src, dest | void |

4.1.5 Display

| Method | Description | Parameters | Return |
|---|---|---|---|
| displayGraph() | Prints the entire network | None | void |

5. Algorithm Implementations

5.1 Dijkstra’s Algorithm (findShortestPath

Priority queue (min-heap) for node selection

Distance map (unordered_map<string, int>) initialized to INT_MAX

Previous node map for path reconstruction

Execution:Extract node with minimum distance

Relax edges and update distances

Path Reconstruction:

Backtrack from destination using the previous map

Time Complexity:O((V + E) log V) (V = vertices, E = edges)

5.2 BFS (bfsTraversal)

Uses a queue to explore nodes level by level.

Time Complexity: O(V + E)

5.3 DFS (dfsTraversal)

Uses a stack (recursive alternative possible).

Time Complexity: O(V + E)

6. Input/Output Specifications

6.1 Input Handling

Menu-Driven: User selects options (1-10).

Validation: Checks for invalid inputs (non-integer choices, negative weights).

6.2 Expected Outputs

| Function | Output Example |
| --- | --- |
| addLocation() | "Location 'Paris' added successfully." |
| removeLocation() | "Location 'London' removed successfully." |
| addRoute() | "Route added between 'Paris' and 'London' with weight 300." |
| findShortestPath() | "Shortest path from 'Paris' to 'Rome': Total distance: 1800. Path: Paris → Berlin → Rome" |

7. Error Handling & Edge Cases

7.1 Common Errors

| Scenario | Handling |
|---|---|
| Invalid menu choice | "Invalid choice. Please enter a number between 1 and 10." |
| Non-existent location | "Location 'Tokyo' not found." |
| Duplicate route | "Route already exists. Use updateRoute to change weight." |
| No path exists | "No path exists between 'Paris' and 'Sydney'." |

7.2 Edge CasesEmpty Graph: displayGraph() shows no connections.Self-loops: Not explicitly prevented (could be added).Disconnected Components: findShortestPath() detects unreachability.

8. Performance Analysis

| Operation | Time Complexity | Space Complexity |
|---|---|---|
| addLocation | O(1) | O(1) |
| removeLocation | O(V + E) | O(1) |
| addRoute | O(1) avg | O(1) |
| bfsTraversal | O(V + E) | O(V) |
| findShortestPath | O((V + E) log V) | O(V) |

Optimizations:

Adjacency List: Efficient for sparse graphs.

Priority Queue: Ensures optimal node selection in Dijkstra’s.

9. Limitations & Known Issues

No Persistence: Data is lost after program exit.

No Multithreading: Large graphs may block execution.

No GUI: Console-based only.

No Negative Weights: Dijkstra’s fails with negative edges.

10. Future Enhancements

File I/O: Save/load travel networks.

A Algorithm:Optimized pathfinding for large graphs.

Visualization: Graph plotting using external libraries.

Travel Time Estimates: Incorporate real-world data.

## Conclusion

This Travel Planner System provides a robust implementation of graph algorithms for route planning. Future work includes persistence, optimization, and visualization.