🚙 City Path Finder: A DSA Project

Menu-driven program to find the shortest path between cities

Presented by: Amish
Roll No: 24/A11/030
Language used: Java

Project Overview:

- **Goal:** To find the shortest path between cities using a graph-based approach.

- **Key Features:**

- Add cities and roads dynamically.
- Visualize the map (adjacency list).
- Calculate shortest path using Dijkstra's Algorithm.
- Menu-driven for user-friendly interaction.

Technologies Used:

- **Programming Languages:**
- Java
- **Data Structures:**
- HashMap
- Priority Queue
- Graph (Adjacency List)
- Dijkstra's Algorithm

Graph Representation:

- **Class Used:** Graph.java
- **Data Structure:** HashMap
- **Why HashMap?**
- Fast lookup and insertion (O(1) average case).
- Efficient for sparse graphs.
- Easy representation of city-to-city distances.
- **Bidirectional Roads:**

Roads added in both directions for undirected graph.

Dijkstra's Algorithm:

- **Used In:** CityPathFinder.java
- **Purpose:** To find the shortest path from a source city to a destination.
- **Key Concepts:**
- dist → Stores current shortest distances.
- prev → Tracks the path.
- PriorityQueue → Picks the next closest city efficiently.

Why Priority Queue?

- Implements a **min-heap** based structure.
- Ensures that the city with the smallest current distance is processed next.
- Optimizes Dijkstra's algorithm to run in O(E log V) time.

Why HashMap over Array?

- Cities are strings, not indices.
- HashMap allows direct mapping from city names.
- Dynamic and memory-efficient.

Menu-Driven Program:

- **Implemented in Main.java**
- **Options available:**

1.Add city

2.Add road

3.Display map

4.Find shortest path

5.Exit

- Helps in real-time data input and testing.

Display Map Feature:

- Shows all cities and their connected neighbors.
- Helps in visual debugging and understanding graph structure.

```
Map Overview:
- A connected to: {B=4, C=6}
- B connected to: {A=4, D=3}
```

Example Path Calculation:

Input: Start = A, End = D

```
Shortest path: A -> B -> D
Total distance: 7
```

Time & Space Complexity:

- **Dijkstra's Time Complexity:** O((V + E) log V) with PriorityQueue.
- **Space Complexity:**
- adjList: O(V + E)
- dist, prev: O(V)

Limitations & Scope:

- ✅ Works for undirected graphs.
- ❌ Doesn’t handle negative weights.
- ❌ No GUI or real map integration (could be future scope).
- 🔁 Scope to add:
- City removal.
- One-way roads.
- Real-world APIs like Google Maps.

Conclusion:

- Efficiently finds shortest path using Dijkstra's algorithm
- Demonstrates effective use of graphs, maps, and queues.\
- Reinforces key DSA concepts with a real-world-like application.

THANK YOU

By Amish (24/A11/030)