MINI PROJECT

Student Name: Anubhav Yadav **UID: 25MCD10057**

Branch: MCA (Data Science) **Section/Group: 25MCD-2(A)**

Semester: 1st Semester **Date of Performance: 03-11-25**

Subject Name: Design and Analysis Algorithms Lab **Subject Code: 25CAP-612**

TITLE: SMART DELIVERY ROUTE FINDER

1. Aim:

To design and develop a **Smart Delivery Route Finder** web application using **Flask (Python)** that helps users determine the **shortest and most efficient delivery route** between multiple locations. The project focuses on applying **graph algorithms** to real-world delivery route optimization problems.

2. Project Introduction:

In today's world, logistics and delivery companies need to ensure fast and efficient transportation of goods.
Choosing the shortest path between source and destination can save both **time and fuel**.

The *Smart Delivery Route Finder* aims to simplify this process by allowing users to:

- Select a **source** and **destination** location.
- Automatically calculate the **shortest possible route**.
- Display the **optimal path and total distance** visually.

The project uses **HTML and CSS** for the frontend interface, and **Flask (Python)** for the backend logic.
A **distance matrix** defines the distances between all locations, and the system dynamically computes the optimal path.

3. Software Requirements:

1. Hardware Requirements

- Processor: Intel i3 or above

- RAM: Minimum 4 GB
- Hard Disk: 500 MB free space

2. **Software Requirements**

- **Operating System:** Windows 10 / 11
- **Programming Language:** Python 3.10 or higher
- **Framework:** Flask (for backend)
- **Frontend:** HTML, CSS
- **Libraries Used:**
    - matplotlib – for graph plotting
    - flask – for web application
    - time**,** os – for cache and file handling
- **Browser:** Google Chrome or Microsoft Edge
- **Editor:** VS Code / PyCharm

4. Algorithm Used – Dijkstra's Algorithm:

The algorithm used in this project is a simplified version of **Dijkstra's Algorithm**, which is widely used for finding the **shortest path in a weighted graph**.
Each node represents a location (A, B, C, D), and edges represent distances between these locations.

Algorithm Steps

1. Initialize all nodes with infinite distance except the source (distance = 0).
2. Visit the unvisited node with the smallest known distance.
3. Update the neighboring nodes' distances if a shorter path is found.
4. Mark the current node as visited.
5. Repeat until the destination node is reached or all nodes are visited.
6. Output the path with the minimum total distance.

Why Dijkstra's Algorithm?

- Guarantees the **shortest path** in a weighted graph.
- Efficient and easy to implement for small-scale delivery networks.
- Commonly used in **GPS navigation**, **network routing**, and **transport systems**.

5. Code:

```
distances = {
    'Warehouse': {'Depot': 5, 'Supermarket': 9, 'Retail Store': 15, 'Cold Storage': 8, 'Distribution Hub': 12},
    'Depot': {'Warehouse': 5, 'Supermarket': 7, 'Retail Store': 10, 'Cold Storage': 6, 'Distribution Hub': 11},
    'Supermarket': {'Warehouse': 9, 'Depot': 7, 'Retail Store': 4, 'Cold Storage': 5, 'Distribution Hub': 9},
    'Retail Store': {'Warehouse': 15, 'Depot': 10, 'Supermarket': 4, 'Cold Storage': 7, 'Distribution Hub': 6},
    'Cold Storage': {'Warehouse': 8, 'Depot': 6, 'Supermarket': 5, 'Retail Store': 7, 'Distribution Hub': 4},
    'Distribution Hub': {'Warehouse': 12, 'Depot': 11, 'Supermarket': 9, 'Retail Store': 6, 'Cold Storage': 4}
}
def find_shortest_path(src, dest):
    routes = list(distances.keys())
    shortest_path_nodes = [src, dest]
    shortest_dist = distances[src].get(dest, float('inf'))

    # Check intermediate routes
    for mid in routes:
        if mid != src and mid != dest:
            total = distances[src].get(mid, float('inf')) + distances[mid].get(dest, float('inf'))
            if total < shortest_dist:
                shortest_dist = total
                shortest_path_nodes = [src, mid, dest]

    return shortest_path_nodes, shortest_dist
```

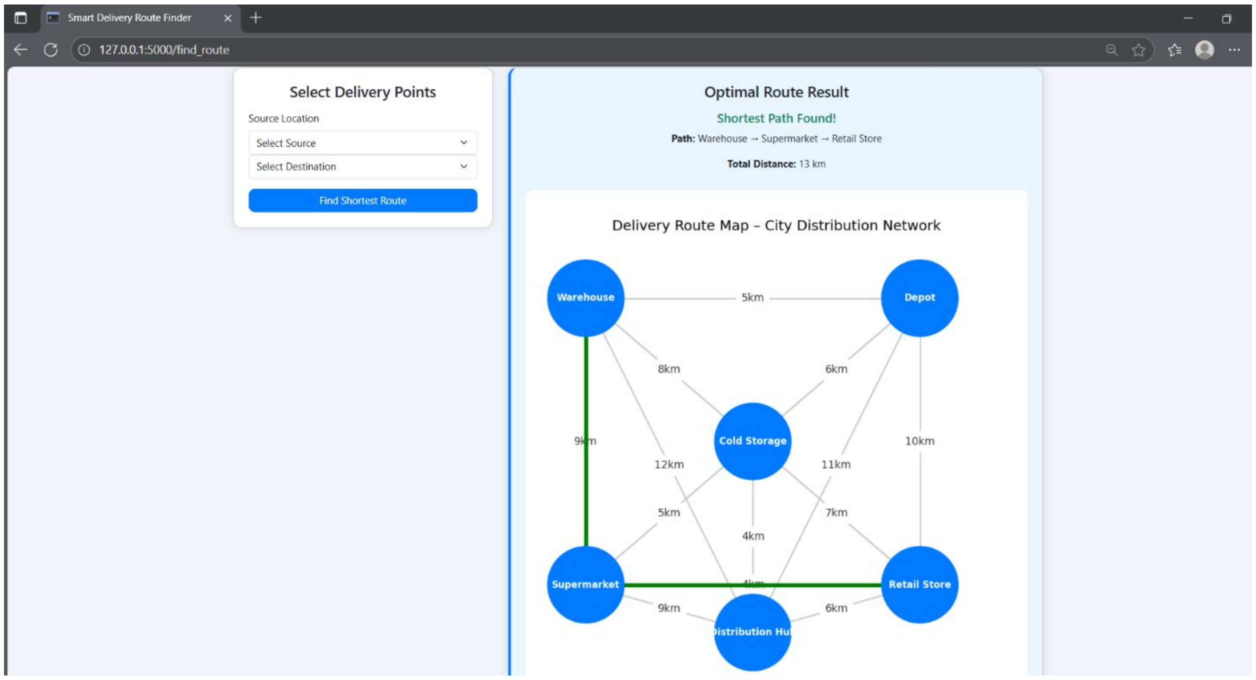
Explanation:

- The `distances` dictionary stores all connections between the locations.
- The function `find_shortest_path()` checks direct and indirect paths.
- It returns the **shortest path** and its **total distance**.

6. Working of the Project:

1. **User Input:**
   The user selects a **Source** and **Destination** location from dropdown menus in the webpage.
2. **Backend Processing (Flask):**
   Flask receives the data and runs the Python function `find_shortest_path()`.
3. **Algorithm Execution:**
   The function calculates all possible routes and determines the shortest path.
4. **Result Display:**
   The calculated path and total distance are displayed on the webpage.
   Additionally, a **visual graph/map** is shown to represent the selected route.

6. <u>Output:</u>



Optimal Route Result

Shortest Path Found!

Path: Warehouse → Supermarket → Retail Store

Total Distance: 13 km

7. Learning Outcome:

- **Understanding of Graph Algorithms:**
  Gained practical knowledge of how **Dijkstra's Algorithm** works for finding the shortest path between nodes.
- **Integration of Backend and Frontend:**
  Learned how to connect a **Flask backend** with **HTML/CSS frontend** to create a functional web application.
- **Data Visualization Skills:**
  Implemented **matplotlib** to visually represent delivery routes on a city map, improving graphical interpretation of data.
- **Problem-Solving and Optimization:**
  Developed logical thinking to optimize delivery networks and analyze route efficiency using algorithmic concepts.
- **Real-world Application of Python:**
  Applied **Python programming** in a real-world scenario involving logistics, route optimization, and automation.