Amar Shukla A2 B2 27
Kedar Singh
Abhishekh Purohit

Route Finder Project - Complete Explanation

📋 Project Overview

This project finds the **shortest paths** between cities in a road network and visualizes them beautifully. It consists of two main files that work together.

🔧 File 1: `route_backend.py`

What It Does:

- Takes input of cities and roads from the user
- Calculates the shortest distance from a source city to all other cities
- Saves the results to CSV files for visualization
- Handles negative weight cycles (if they exist)

Algorithm Used: Bellman-Ford Algorithm

Why Bellman-Ford?

- ✅ Works with **negative edge weights** (unlike Dijkstra's)
- ✅ Detects **negative weight cycles**
- ✅ Simple to implement and understand
- ✅ Guaranteed to find shortest paths in V-1 iterations

🧮 How Bellman-Ford Algorithm Works

Step-by-Step Process:

1. INITIALIZE:
   - Set distance to source city = 0
   - Set all other distances = INFINITY (∞)

2. RELAX EDGES (V-1 times):

For each road (u → v with weight w):
If distance[u] + w < distance[v]:
distance[v] = distance[u] + w

This means: "If going through city u makes the path to city v shorter, update it!"

3. CHECK FOR NEGATIVE CYCLES:
Do one more relaxation:
If any distance can still be improved:
⚠️ Negative cycle exists!

4. MEMOIZE:
Store results so we don't recalculate for the same source

📊 Example Walkthrough

Let's trace through a simple example:

Cities: A, B, C, D
Roads:
A → B (weight 4)
A → C (weight 2)
B → C (weight 1)
C → D (weight 5)

Source: A

Iteration 0 (Initialization):

Distance: A=0, B=∞, C=∞, D=∞

Iteration 1:

Check A→B: 0+4 < ∞ ✓ B=4
Check A→C: 0+2 < ∞ ✓ C=2
Check B→C: 4+1 < 2 ✗ (no update)
Check C→D: 2+5 < ∞ ✓ D=7

Distance: A=0, B=4, C=2, D=7

Iteration 2:

Check B→C: 4+1 < 2 ✗
Check C→D: 2+5 < 7 ✗ (no change)

Distance: A=0, B=4, C=2, D=7 ✓ FINAL

🎨 File 2: `route_visualizer.py`

What It Does:

- Reads the CSV files generated by the backend
- Creates a beautiful graph visualization
- Shows the shortest path network
- Uses colors to indicate reachable/unreachable cities

Algorithm Used: Dijkstra's Algorithm (via NetworkX)

Why Dijkstra here?

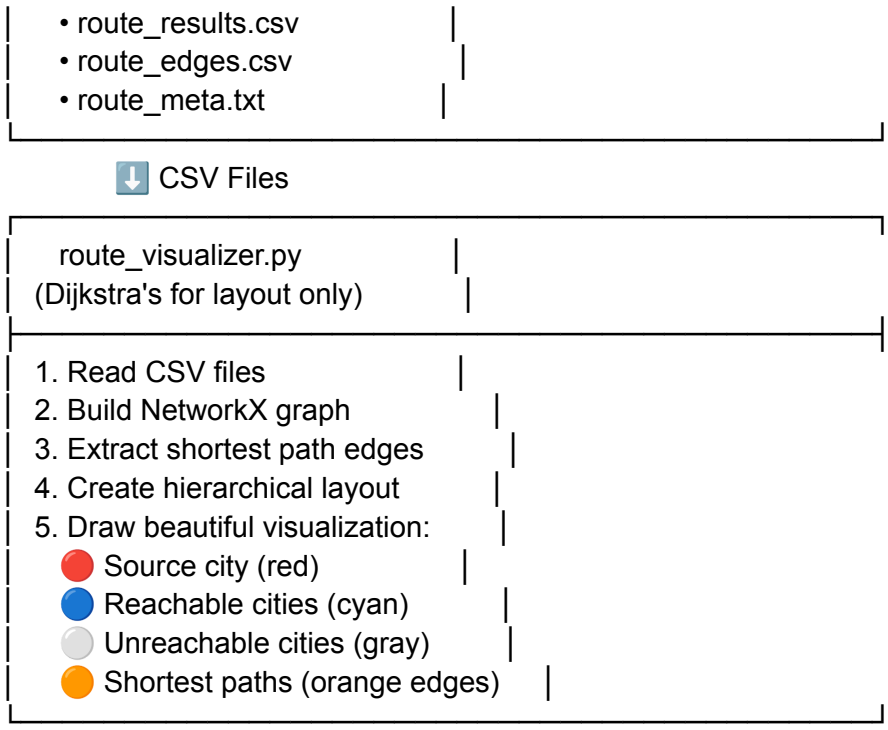
For visualization purposes only! We need to:

1. Extract which edges are part of shortest paths
2. Create a hierarchical layout based on distance from source

Note: The actual shortest path calculation was already done by Bellman-Ford. Dijkstra is only used here to determine graph structure for pretty visualization.

🔄 How the Two Files Work Together

```
┌─────────────────────────────────────────────┐
│      route_backend.py                       │
│  (Bellman-Ford Algorithm)                   │
├─────────────────────────────────────────────┤
│  1. Get user input (cities & roads)         │
│  2. Build graph structure                   │
│  3. Run Bellman-Ford from source            │
│  4. Calculate shortest distances            │
│  5. Detect negative cycles                  │
│  6. Save to CSV files:                      │
```



🎯 Key Features Implemented

1. Memoization

```
memo = [[INT_MAX for _ in range(MAX)] for _ in range(MAX)]
```

- Stores previously calculated results
- Avoids recalculating if you query the same source again
- Saves computation time!

2. Negative Cycle Detection

```
# After V-1 relaxations, check one more time
if dist[u] + w < dist[v]:
    print("Negative weight cycle detected!")
```

- Prevents infinite loops in cyclic graphs with negative weights
- Essential for algorithm correctness

3. Graph Classes

```
class Edge:
   src, dest, weight

class Graph:
   V (vertices), E (edges), edge[]
```

- Clean object-oriented design
- Represents the road network structure

4. Hierarchical Layout

```
def create_layout(G, source):
   # Group nodes by distance from source
   # Level 0: Source
   # Level 1: Direct neighbors
   # Level 2: 2 steps away
   # ...
```

- Creates beautiful tree-like visualization
- Shows logical flow from source to destinations

📈 Time & Space Complexity

Bellman-Ford Algorithm:

- **Time Complexity:** O(V × E)
  - V = number of cities
  - E = number of roads
  - For V=5, E=7: ~35 operations
- **Space Complexity:** O(V²)
  - Memoization table stores V×V distances
  - For V=50: 2,500 entries

Visualization:

- **Time Complexity:** O(E log V) for Dijkstra
- **Space Complexity:** O(V + E) for graph storage

🆚 Bellman-Ford vs Dijkstra

Feature	Bellman-Ford	Dijkstra
Negative weights	✅ Yes	❌ No
Negative cycle detection	✅ Yes	❌ No
Time complexity	O(V×E)	O(E log V)
Speed	Slower	Faster
Use case	General graphs	Non-negative weights

Our Choice: Bellman-Ford for robustness and cycle detection!

🎨 Visualization Color Scheme

🔴 Red (#FF6B6B) → Source city
🔵 Cyan (#4ECDC4) → Reachable destinations
⚪ Gray (#95A5A6) → Unreachable cities
🟠 Orange (#F39C12) → Shortest path edges
⬛ Dark (#1a1a1a) → Background

📝 CSV File Formats

`route_results.csv`

```
City,Distance
Mumbai,0
Delhi,1400
Bangalore,980
Chennai,1330
Kolkata,INF
```

`route_edges.csv`

```
Source,Destination,Weight
```

Mumbai,Delhi,1400
Mumbai,Bangalore,980
Delhi,Kolkata,1470

🚀 Real-World Applications

1. **GPS Navigation** - Finding shortest routes
2. **Network Routing** - Internet packet routing
3. **Logistics** - Delivery route optimization
4. **Transportation** - Flight/train connections
5. **Game Development** - Pathfinding for NPCs

💡 Key Takeaways

✅ **Bellman-Ford** calculates actual shortest paths (backend)
✅ **Dijkstra** helps visualize the graph structure (frontend)
✅ **Memoization** makes repeated queries faster
✅ **Negative cycle detection** prevents infinite loops
✅ **Dark theme** makes visualization look awesome! 🌙

🔍 Code Flow Summary

User Input → Graph Building → Bellman-Ford → CSV Export
↓
Visualization
↓
Beautiful Graph! 🎨