Smart Delivery Route Planner - Design Document

A. Overview

Purpose and Scenario

The Smart Delivery Route Planner is a logistics optimization tool designed to help delivery companies find the most efficient routes for their drivers. The system models a city's road network as a weighted graph where intersections are nodes and roads are edges with distances as weights.

Inputs

- A CSV file containing road network data (node connections and distances)
- Depot location (starting point)
- List of delivery locations to visit

Outputs

- Optimal delivery route sequence
- Total travel distance
- Step-by-step directions

Manual Worked-Out Example

- Consider a simple city network: A —4— B | | 2 3 | | C —5— D —2— E

Input: - Depot: A - Deliveries: B, D, E

Process: 1. Check if all deliveries are reachable from depot A 2. Find shortest paths between all points 3. Determine optimal visiting order

Output: - Route: A → B → D → E - Total distance: 4 + 5 + 2 = 11 km

B. Function Designs

1. build_graph(filename)

Purpose: Construct a graph representation of the road network from a CSV file

Parameters: - `filename`: a string indicating the path to the CSV file

Return: a dictionary representing the graph where keys are nodes and values are dictionaries of connected nodes with distances

Pseudocode:

```
1. Create an empty graph dictionary
2. Open and read the CSV file
3. For each row in the file:
   - Extract source node, destination node, and distance
   - Add edge from source to destination with distance
   - Add edge from destination to source with distance (undirected graph)
4. Return the completed graph
```

2. is_route_possible(graph, start, end)

Purpose: Determine if a route exists between two locations using depth-first search

Parameters: - `graph`: dictionary representing the road network - `start`: string indicating the starting location - `end`: string indicating the destination location

Return: boolean value (True if route exists, False otherwise)

Pseudocode: 1. Create a set to track visited nodes 2. Create a stack and push the start node 3. While stack is not empty: - Pop a node from stack - If node equals end, return True - Mark node as visited - For each unvisited neighbor: - Push neighbor to stack 4. Return False (no route found)

3. find_shortest_path(graph, start, end)

Purpose: Find the shortest path between two locations using Dijkstra's algorithm

Parameters: - `graph`: dictionary representing the road network - `start`: string indicating the starting location - `end`: string indicating the destination location

Return: tuple containing (shortest distance, path as list of nodes)

Pseudocode: 1. Initialize distances dictionary with infinity for all nodes except start (0) 2. Initialize previous dictionary to track path 3. Create priority queue with (distance, node) pairs 4. Add (0, start) to priority queue 5. While priority queue is not empty: - Get node with minimum distance - If node is end, reconstruct and return path - For each neighbor: - Calculate new distance through current node - If new distance is shorter: - Update distance - Update previous node - Add to priority queue 6. Return (infinity, empty list) if no path found

4. plan_delivery(graph, depot, deliveries)

Purpose: Plan the optimal delivery route visiting all locations

Parameters: - `graph`: dictionary representing the road network - `depot`: string indicating the starting depot location - `deliveries`: list of strings indicating delivery locations

Return: tuple containing (total distance, ordered list of locations to visit)

Pseudocode: 1. Verify all deliveries are reachable from depot 2. Create distance matrix between all locations (depot + deliveries) 3. Start from depot 4. While unvisited deliveries exist: - Find nearest unvisited delivery - Add to route - Update current location - Mark as visited 5. Calculate total distance 6. Return (total distance, route)

Author

Group 14 (Ju Ho Kim, Sangmin Kim)
CS 034 - Data Structures and Advanced Python
Spring 2025