MINI WASLNEY

002

Team ID

Team Members

عمر محمود علي سواكن
20231700134
SWE2

أحمد أيمن محمد يوسف
20231700302
CSec2

محمد عثمان أنور عثمان
20231700140
SWE2

عبد الرحمن بكر مبروك
20231700230
AI2

زياد ابراهيم محمد
20231700317
CSec2

أحمد محمد عبد الحميد
20231700205
AI2

يوسف محمد عبد الحميد عبد المالك
20231700150
SWE2

1. Introduction

Waslney is a graphical tool for finding optimal routes between cities based on distance or time using the shortest-path algorithm (Dijkstra's). Users can create, edit (insert/remove cities and edges), delete, and traverse maps, with changes saved only upon explicit exit confirmation.

2. Input and Output Scenarios

1. Graph Creation

Input

- **User Action:** Clicks "Add Graph" button.
- **Data Entry:**
    - Graph Name: Text field (e.g., "CompleteMap").
    - Constraints: Unique name.

Processing

1. **Validation:**
    - Check for duplicates in existing graphs.
2. **Graph Initialization:**
    - Create adjacency list in memory.

Output

- **Success:** New graph added to dropdown menu; status toast: "Graph added successfully."
- **Errors:**
    - Duplicate: "Graph name already exists. Use a unique name."
    - Empty name: "Graph name cannot be empty."

2. Graph Editing

2.1 Insert City

Input:

- City Name: Text field (e.g., "City A").

Processing:

1. **Validation:**
    - Check for duplicates in current graph's nodes object.
2. **Data Update:**
    - Add to adjacency list: nodes: {"City A": {}}.

Output:

- **Success:** City appears in node list; available for edge creation.
- **Errors:**
    - Duplicate city: "City already exists in this graph."
    - Empty name: "City name cannot be empty."

2.2 Insert Edge

Input:

- Source/Destination: Dropdowns (existing cities).
- Distance: Numeric field.
- Time: Numeric field.

Processing:

1. **Validation:**
    - Ensure source ≠ destination.
    - Ensure both cities exist in the graph.
    - Reject non-numeric values for time and distance.

2. **Data Update:**
    - Append to edges.

Output:

- **Success:** Edge visualized; updated in traversal algorithms.
- **Errors:**
    - Same nodes: "The two cities must be different."
    - Invalid values: "Please enter valid numerical values for time and distance."
    - Empty Values: Appropriate message. (e.g., “First/Second city cannot be empty.”)

2.3 Delete City

Input

- **User Action:**
    1. Selects city from dropdown/list (e.g., "City A").
    2. Clicks "Delete" button.

Processing

1. **Pre-Deletion Checks:**
    - **Check 1:** Verify city exists in graph (nodes object).
    - **Check 2:** Scan edges for connections to/from the city.
2. **Data Removal:**
    - **From** nodes**:** Remove key (e.g., delete nodes["City A"]).
    - **From** edges**:** Filter out all edges where from or to equals "City A".

Output

- **Success:**
    - City removed from UI lists and graph visualization.
    - Status toast: "City deleted successfully."

- **Errors:**
    - City not found: No action taken.
    - Empty name: "City name cannot be empty."

2.4 Delete Edge

Input

- **User Action:**
    1. Selects edge via: Dropdowns for First City and Second City.
    2. Clicks "Delete" button.

Processing

1. **Pre-Deletion Checks:**
    - Verify both cities exist in nodes.
    - Ensure there is a connection between the selected cities.
2. **Data Removal:**
    - Exclude the target edge.

Output

- **Success:**
    - Edge removed from UI and traversal algorithms.
    - Status toast: "Edge deleted successfully."
- **Errors:**
    - Edge not found: "No edge exists between the selected cities."
    - Invalid selection: "The two cities must be different."
    - Empty values: "Please select both cities."

3. Graph Deletion

Input

- Selection: Graph from dropdown → Click "Delete Graph".

Processing

1. **Confirmation Dialog:**
    - "Are you sure you want to delete the graph 'CompleteMap'? ."
2. **Data Removal:**
    - Remove graph from memory and dropdown.

Output

- **Success:** Graph list updated; status toast: "Graph deleted."
- **Cancel:** No action taken.

4. Save Protocol

Triggers

1. Explicit "Save and Exit" click.
2. Program closure attempt.
    - Only prompts if unsaved changes exist.

Processing

1. Check for Unsaved Changes:
    - Compare current state to last saved version.
2. Save Dialog:
    - "Do you want to save the changes you made?" (Yes/No/Cancel).
    - Yes: Write updated graph to file.
    - No/Cancel: Discard changes.

Output

- **Saved:** File updated.

- **Unsaved:** All edits since last save discarded.

5. Shortest Path Calculation

Input

1. **Source/Destination Selection:**
    - Dropdown menus for Source (e.g., "E") and Destination (e.g., "C").
2. **Mode Selection:**
    - Distance (optimizes for km) or Time (optimizes for hrs).
3. **Action Trigger:**
    - Click "Find The Shortest Path" button.

Processing

1. **Validation:**
    - Check if source/destination nodes exist in the graph.
    - Verify at least one path exists (graph connectivity check).
2. **Algorithm Execution (Dijkstra's).**
3. **Path Reconstruction:**
    - Trace the shortest path (e.g., E → D → C) and total cost.

Output

- **Success:**
    - Displays path + metrics.
    - Visual highlight of the path in the graph UI.
- **Errors:**
    - No path exists: "No path found."
    - Empty Values: Appropriate message. (e.g., "Dijkstra's mode cannot be empty.")

6. Graph Traversal

Input

1. **Start Node Selection:**
    - Dropdown for Start City (e.g., "B").
2. **Algorithm Choice:**
    - Breadth-First Search (BFS) or Depth-First Search (DFS).

Processing

- BFS or DFS implementation.

Output

- **Success:**
    - Displays traversal order (e.g., B → E → D → C → A).
    - Animates traversal step-by-step in the UI.
- **Errors:**
    - Empty values: "Start cannot be empty."
    - Disconnected graph: Traverses only reachable nodes.