Emily Monroe and Ethan Smith

CS 355

Fall 2024

Project 3

Due Date: November 15, 2024

Program Requirements:

Create a real-world project that demonstrates the shortest path problem. Your project will be composed of a front-end application that makes use of a back end. Your back end will solve the shortest path algorithm on a directed acyclic graph. The front end will show you that you back-end works. You will have to create a proposal, a design document, implementation, presentation, a final project portfolio and a team assessment.

Program Inputs:

CCP Locations

- Weslyann hall coordinates
- The Commons coordinates
- Flowers Hall Coordinates

Hospital Locations

- North Alabama medical center Coordinates
- Hellen Keller Hospital Coordinates

Paths

- At least 2 different routes to a hospital per building

UPDATED TEST PLAN

Test Plan:

Test Case 1.1: Add Edge

Description: Verify that addEdge correctly adds an edge to the adjacency list.

- **Steps**:
    1. Create a graph instance.
    2. Add an edge between nodes 1 and 2 with a weight of 5.
    3. Retrieve neighbors of node 1.
- **Result**: The neighbor list of node 1 contains node 2 with weight 5.

Test Case 1.2: Get Neighbors

Description: Ensure getNeighbors returns correct neighbors for a given node.

- **Steps**:
    1. Add edges: 1 -> 2 (weight 5), 1 -> 3 (weight 10)
    2. Retrieve neighbors of node 1.
- **Result**: The neighbor list of node1 contains nodes 2 and 3 with weights 5 and 10, respectively.

Dijkstra Class Tests

Test Case 2.1: Find Shortest Path in Simple Graph

Description: Verify that Dijkstra's algorithm finds the shortest path in a simple graph.

- **Steps**:
    1. Create a graph with nodes: 1 -> 2 (weight 5), 2 -> 3 (weight 3), 1 -> 3 (weight 10).
    2. Use Dijkstra's algorithm to find the shortest path from node 1.
- **Result**: The shortest path distance from node 1 to node 3 is 8 (1 -> 2 -> 3).

Test Case 2.2: **Find Closest Hospital**

Description: Ensure the nearest hospital is correctly identified.

- **Steps**:
    1. Run Dijkstra's algorithm from an CCP location "Commons" with hospitals at nodes North Alabama medical center and Hellen Keller.
    2. Check the nearest hospital from the computed distances.
- **Expected Result**: The hospital with the shortest distance to node 1 is identified as North Alabama medical center.

ccpLocations Class Tests

Test Case 3.1: **Add Hospital and CCP Location**

Description: Confirm that hospitals and emergency locations are added correctly.

- **Steps**:
    1. Add The North Alabama medical center as a node 3.
    2. Add Commons as node 1.
- **Result**: The hospital and emergency location are present in the respective sets.

Test Case 3.2: **Find Shortest Path to Nearest Hospital**

Description: Ensure the shortest path from an emergency location to the nearest hospital is found.

- **Steps**:
    1. Create a graph with several nodes and edges.
    2. Designate node 4 as Hellen Keller and node 1 as Commons.
    3. Run shortestPathToHospital for the emergency location.
- **Expected Result**: The path correctly lists the nodes and total distance to the nearest hospital.

Test Case 4.1: **Multiple Hospitals and CCP Locations**

Description: Test with multiple hospitals and emergency locations in a complex graph.

- **Steps**:
    1. Create a graph with nodes and multiple edges.
    2. Add 2 hospitals and 4 CCP locations.

3. Run the program to determine the shortest path to the closest hospital for each emergency location.

- **Result**: Each emergency location should output the correct shortest path and distance to the nearest hospital. This should be the North Alabama medical center for all locations. The correct path is different for every location

Test Case 4.2: **Disconnected Graph Segments**

Description: Test if the program handles cases where an emergency location has no path to any hospital.

- **Steps**:
  1. Create a graph with two disconnected segments: one containing hospitals and the other containing emergency locations.
  2. Try to find the shortest path for an emergency location in the segment without hospitals.
- **Expected Result**: The program returns with no hospital reachable.

UPDATED Pseudocode

Algorithm In Pseudocode:

Define Classes

Class Graph:

Vertex Class

Attributes:

- label: String

Methods:

- Constructor Vertex(vertexLabel):
  - Initializes label with vertexLabel.
- getLabel():
  - Return label.

Edge Class

Attributes:

- fromVert: Vertex (pointer to the starting vertex)
- toVert: Vertex (pointer to the destination vertex)

- weightSet: List of Doubles

Methods:

- Constructor Edge(from, to, edgeWeight):
    - Initializes fromVert with from, toVert with to, and weightSet with edgeWeight.
- getToVert():
    - Return toVert.
- getFromVert():
    - Return fromVert.
- getWeight():
    - Return the first value in weightSet (i.e., weightSet[0]).
- getWeightSet():
    - Return the entire weightSet.

Graph Class

Attributes:

- fromEdges: Map of Vertex → List of Edge pointers (edges starting from each vertex)
- toEdges: Map of Vertex → List of Edge pointers (edges ending at each vertex)
- undirEdges: Map of Vertex → List of Edge pointers (unused in the current context)

Methods:

1. Constructor Graph:
    - Initializes the data structures for edges (fromEdges, toEdges, undirEdges).
2. Destructor GraphDestructor():
    - For each entry in fromEdges and toEdges:
        - Collect distinct vertices and edges.
        - Delete all edges and vertices.
3. addVertex(newLabel):
    - Create a new Vertex with newLabel.
    - Add the vertex to fromEdges and toEdges maps.
    - Return the new vertex.
4. addDirectEdge(fromVertex, toVertex, weight = 1.0):

  - Check if an edge from fromVertex to toVertex already exists:
    - If yes, return null.
  - Create a new Edge between fromVertex and toVertex with the given weight.
  - Add the new edge to both fromEdges[fromVertex] and toEdges[toVertex].
  - Return the new edge.

5. getEdges():
   - Initialize an empty set of edges.
   - For each vertex in fromEdges:
     - Add each edge in the list of edges for that vertex to the set.
   - Return the set of edges.
6. getWeightPair(vertA, vertB):
   - For each edge originating from vertA:
     - If the edge leads to vertB, return the edge's weight and weight set.
   - If no such edge exists, return -1.0 and a default weight set.
7. getEdgesFrom(fromVertex):
   - If fromVertex exists in fromEdges, return the list of edges originating from that vertex.
   - If not, print an error message and return an empty list.
8. getEdgesTo(toVertex):
   - Return the list of edges that point to toVertex in toEdges.
9. getVertex(vertexLabel):
   - For each vertex in fromEdges, if the vertex's label matches vertexLabel, return that vertex.
   - If not found, return null.
10. getVertices():
    - Initialize an empty list of vertices.
    - For each vertex in fromEdges, add it to the list.
    - Return the list of vertices.
11. hasEdge(fromVertex, toVertex):
    - If fromVertex exists in fromEdges:

        - Check if any edge from fromVertex leads to toVertex.
    - Return true if such an edge exists, otherwise false

Class: Dijkstra

Attributes:

- dists: A map of Vertex to double (stores the shortest distance from the start vertex to each vertex).
- prev: A map of Vertex to Vertex (stores the predecessor vertex for each vertex).
- visited: A map of Vertex to bool (indicates whether a vertex has been visited).

Methods:

1. Constructor Dijkstra():
    - Initialize dists, prev, and visited as empty maps.
2. findDijkstra(graph, start, end):
    - Parameters:
        - graph: The graph on which Dijkstra's algorithm is to be run.
        - start: The starting vertex.
        - end: The destination vertex.
    - Steps:
        - Initialize all vertices in the graph:
            - Set dists[v] to infinity for each vertex v.
            - Set prev[v] to nullptr for each vertex.
            - Set visited[v] to false for each vertex (mark all as unvisited).
        - Set the distance for the start vertex to 0 (because the distance from the start to itself is 0).
        - Priority Queue Setup:
            - Define a comparison lambda compare to prioritize vertices with the smallest distance.
            - Initialize a priority queue pq of pairs of vertices and distances, using the custom comparator.
        - Dijkstra’s Algorithm Loop:
            - While the priority queue is not empty:
                - Pop the vertex currVert with the smallest distance from the queue.
                - If currVert has already been visited, skip it.
                - Mark currVert as visited.
                - If currVert is the end vertex, exit the loop early (the shortest path is found).
                - For each outgoing edge from currVert:
                    - Get the neighbor vertex vertNeighbor and the edge's weight.

            - If vertNeighbor is not visited and the new calculated distance (dists[currVert] + weight) is smaller than the existing distance for vertNeighbor:
                - Update dists[vertNeighbor] to the new smaller distance.
                - Set prev[vertNeighbor] to currVert (to track the path).
                - Push vertNeighbor and its updated distance into the priority queue.
    - Rebuild the Path:
        - Initialize an empty list path.
        - Starting from end, trace back using prev to build the path in reverse order.
        - Insert vertices at the front of the path list to reverse the order.
    - Return the Result:
        - If the path is empty or the first vertex in the path is not the start vertex, return an empty path and infinity (no valid path).
        - Otherwise, return the path and the distance to the end vertex.

Class: ccpLocations

1. **Private Members:**
    - graph: A pointer to the graph object containing all nodes (CCPs and hospitals).
    - hospitalNodes: A set to store hospital nodes.
    - ccpNodes: A set to store CCP nodes.

Public Methods:

- Constructor
- addHospital
- addCCPLocation
- shortestPathToHospital

Constructor

2. Purpose:
    - Initializes the ccpLocations object with a graph of CCPs and hospitals.
3. Inputs:
    - graph: Pointer to a Graph object.
4. Process:

- Assign graph to the class member.

Method: addHospital

5. Purpose:
    - Adds a hospital node to the hospitalNodes set.
6. Inputs:
    - hospitalLabel: The label of the hospital (string).
7. Process:
    - Use the graph to retrieve the node for the given hospitalLabel.
    - If the node exists:
        - Add it to the hospitalNodes set.

Method: addCCPLocation

8. Purpose:
    - Adds a CCP node to the ccpNodes set.
9. Inputs:
    - ccpLabel: The label of the CCP location (string).
10. Process:
    - Use the graph to retrieve the node for the given ccpLabel.
    - If the node exists:
        - Add it to the ccpNodes set.

Method: shortestPathToHospital

11. Purpose:
    - Finds the shortest path from a CCP to the nearest hospital using Dijkstra's algorithm.
12. Inputs:
    - ccpLabel: The label of the CCP location (string).
13. Outputs:
    - A list of labels (strings) representing the shortest path from the CCP to the nearest hospital.
14. Process:

- Retrieve the CCP node using its label:
    - If the node does not exist, return ["ccp location not found"].
- Check if the hospitalNodes set is empty:
    - If empty, return ["No hospitals available"].
- Initialize:
    - shortestDistance to infinity.
    - bestPath as an empty list.
- For each hospital in the hospitalNodes set:
    - Use Dijkstra's algorithm to find the path and distance to the hospital.
    - If a valid path is found and the distance is less than shortestDistance:
        - Update shortestDistance with the new distance.
        - Update bestPath with the new path.
- After checking all hospitals:
    - If bestPath is still empty, return ["No hospital reachable"].
- Convert bestPath (a list of vertex objects) to a list of their labels (strings).
- Return the converted list of labels.

Main Program

1. Create Graph:
    - Initialize a graph object.
    - Add vertices for all CCP locations (e.g., Commons, Muscle Shoals Sound Studio, etc.).
    - Add vertices for hospitals (e.g., North Alabama Medical Center, Helen Keller Hospital).
2. Add Edges:
    - For each CCP location, add directed edges to all hospitals with specified weights (distances).
    - Store alternate routes for flexibility in distance calculations.
3. Initialize CCP Locations:
    - Create a ccpLocations object linked to the graph.

    - Register all CCPs and hospitals in the ccpLocations object.

User Interaction

4. Menu Display:
    - Display a menu:
        - Option 1: Find the shortest path from a location to the nearest hospital.
        - Option -1: Exit the program.
5. Process User Choice:
    - If choice is 1:
        - Prompt the user to select a location from a list of CCPs.
        - Validate the user's input.
        - Calculate the shortest path using findShortestPath function.
            - Call the shortestPathToHospital method from the ccpLocations object.
            - Retrieve the shortest path as a list of vertices.
        - Display:
            - The shortest path (sequence of locations).
            - Total distance to the nearest hospital.
            - Alternate routes with distances.
    - If choice is invalid: Display an error and re-prompt.
6. Exit the Program:
    - If the user selects -1, display a goodbye message and terminate the program.

Helper Functions

7. Menu Function:
    - Display the menu and return the user's choice.
8. Prompt Location Function:
    - Display available CCP locations.
    - Continuously prompt the user until a valid location is entered.
9. Find Shortest Path Function:
    - Take the user's location as input.

    - If the location is empty, prompt the user again.
    - Use Dijkstra’s algorithm to calculate the shortest path to the nearest hospital.
    - Return the path and associated distances.

10. Print Edges Function (Optional):
    - Retrieve and display all edges in the graph with their weights for debugging.