Rayner Sarmiento
Algorithms
04/20/25

Final Project Part 1: Al-GO-rithms Delivery Challenge – Algorithm Design & Test Plan

In the fast-paced world of e-commerce and data connectivity, optimizing logistics and network efficiency is paramount. Al-GO-rithms, a delivery company, faces the challenge of ensuring cost-effective routes during peak seasons, a problem closely aligned with data network optimization. As a network technician, I recognize parallels between this project and protocols like OSPF (Open Shortest Path First) and STP (Spanning Tree Protocol), which manage data packet routing and network topology. This project, spanning three parts, focuses on designing algorithms to address real-world delivery challenges: finding the lowest-cost path between two locations, connecting all locations from a hub with minimal cost, and adapting to dynamic network changes.

1. Algorithms Design:

For each of the selected algorithms, the following will be provided:

- **Pseudocode:** Detailed commentary will be provided explaining the logic behind each of the algorithms used.
- **Explanation:** Detail will be provided regarding efficiency, scalability and possible limitations.

Algorithm 1: Lower-cost delivery between two locations

Objective: Find the shortest path (lowest total cost) between an initial node and an end node in a weighted graph. When analyzing these algorithms I based a lot on how data networks work today, since I am a network technician and, for example, I related this part of the project a lot with a routing algorithm that I like a lot called OSPF (Open Shortest Path First), which uses the Dijkstra algorithm, one of the algorithms used for this project. This protocol considers metrics such as the cost of the link. In networks, routers exchange topology information to build routing tables, similar to how this algorithm scans nodes to find the optimal path.

Pseudocode:

```
// Algorithm 1: Dijkstra's Algorithm for Shortest Path

Function algorithm_1(graph, start, end)

        Initialize distance dictionary with infinity for all nodes
        except start +.

        distances = {node: INFINITY for node in graph}

        distances[start] = 0

        track predecessors to reconstruct path

        predecessors = {node: NULL for node in graph}
```

Rayner Sarmiento
Algorithms
04/20/25

```
Priority queue to store (distance, node) pairs
priority_queue = [(0, start)] // Set to track visited nodes
 visited = EMPTY_SET

WHILE priority_queue is not empty
      Extract node with minimum distance
      current_distance, current_node =
      HEAP_POP(priority_queue)

 Skip if already visited
IF current_node in visited
      CONTINUE

Mark as visited
ADD current_node to visited

Stop if reached end node
IF current_node == end
      BREAK

 Explore neighbors
 FOR neighbor, weight in graph[current_node]
       IF neighbor not in visited
            Calculate distance to neighbor via current node
            new_distance = current_distance + weight
      IF new_distance < distances[neighbor]
            Update distance and predecessor
            distances[neighbor] = new_distance
             predecessors[neighbor] = current_node
            Add to priority queue
            HEAP_PUSH(priority_queue, (new_distance,
            neighbor))
```

Rayner Sarmiento
Algorithms
04/20/25

```
Reconstruct path
 path = []
current = end
WHILE current is not NULL
      PREPEND current to path
      current = predecessors[current]
Return path and total cost
RETURN path, distances[end]
```

Logic:

- Dijkstra's algorithm It is used to find the shortest path in a graph. It maintains a priority queue to always explore the node with the least tentative distance, ensuring the most optimal path always.
- The algorithm tracks distances and predecessors to build the path and calculate the total cost.
- A set of visited avoids revisiting nodes that have already been visited, improving efficiency.

Efficiency, Scalability and Limitations:

- **Temporal Complexity:** O((V + E) log V), dominated by priority queue operations.
- Efficiency for graphs with non-negative weights.
- **Limitations:** May not perform well on very large graphs without optimizations.

Algorithm 2: Best Way from Downtown:

Objective: Calculate a minimum spanning tree (MST) to connect all locations from one hub with the lowest total cost. Like the previous algorithm, this algorithm is related to a protocol that is widely used in data networks called STP (Spanning Tree Protocol). It is used to avoid loops and guarantee a cycle-free topology. STP builds a minimum expansion arbor to connect all switches, minimizing the cost of links, like how this algorithm connects locations with the lowest total cost.

Pseudocode:

```
// Algorithm 2: Prim’s Algorithm for Minimum Spanning Tree
FUNCTION algorithm_2(graph, hub)
    Initialize MST as empty list
    mst = []
    Priority queue to store (weight, parent, node) pairs
```

```
        priority_queue = [(0, NULL, hub)]
        Track visited nodes
        visited = EMPTY_SET
        // Total cost of MST
        total_cost = 0

        WHILE priority_queue is not empty
            Extract edge with minimum weight
            weight, parent, current_node = HEAP_POP(priority_queue)

            Skip if already visited
            IF current_node in visited
                CONTINUE

            Mark as visited
            ADD current_node to visited

            Add edge to MST (if not the hub)
            IF parent is not NULL
                APPEND (parent, current_node, weight) to mst
                total_cost = total_cost + weight

            Explore neighbors
            FOR neighbor, weight in graph[current_node]
                IF neighbor not in visited
                    Add edge to priority queue
                    HEAP_PUSH(priority_queue, (weight, current_node,
    neighbor))

        Return MST and total cost
        RETURN mst, total_cost
```

Logic:

- Prim's algorithm is used to calculate the minimum spanning tree, starting from the center. It greedily selects the edge with the lowest weight that connects a visited node to an unvisited one, ensuring that all nodes are connected with the lowest total cost.
- The center is the starting point, but all nodes are included, not just the paths from the center.
- A priority tail ensures efficient selection of the heaviest edge.

Efficiency, Scalability and Limitations:

- Good for dense graphs and when a minimal connection from a hub is desired.
- **Temporal complexity:** O((V + E) log V)
- It does not return shorter routes, only the minimum total cost of connection.

Algorithm 3: Dynamic changes in the network

Rayner Sarmiento
Algorithms
04/20/25

Objectives: Compute an updated MST after removing and adding edges to the graph. This algorithm is responsible for adapting to changes in the graph, when there is a change, always ensuring the best possible optimization. Just as routing protocols update their tables in data networks when a link fails, or a new link is added.

Pseudocode:

```
// Algorithm 3: Update MST with Dynamic Changes
FUNCTION algorithm_3(graph, hub, edges_to_remove, edges_to_add)
    Create a copy of the graph
    updated_graph = COPY(graph)

    Process edge removals
    FOR edge in edges_to_remove
        node1, node2 = PARSE(edge) // e.g., "C-E" -> "C", "E"
        Remove edge in both directions
        REMOVE (node2, weight) from updated_graph[node1] where
node2 matches
        REMOVE (node1, weight) from updated_graph[node2] where
node1 matches

    Process edge additions
    FOR node1, node2, weight in edges_to_add
        Add edge in both directions
        APPEND (node2, weight) to updated_graph[node1]
        APPEND (node1, weight) to updated_graph[node2]

    Compute MST on updated graph using Prim's algorithm
    mst, total_cost = algorithm_2(updated_graph, hub)

    Check if graph is still connected
    IF number of nodes in mst < number of nodes in updated_graph -
1
        RETURN "Error: Graph is disconnected"

    Return updated MST and total cost
    RETURN mst, total_cost
```

Logic:

- Builds on Prim's Algorithm (from Algorithm 2) to compute the MST after modifying the graph.
- Edge removals and additions are processed by updating the graph's adjacency list.
- Checks for graph connectivity by ensuring the MST connects all nodes.

Efficiency, Scalability, and Limitations:

- Simple dynamic graph update before calculating MST.
- Allows real-time adjustment for network changes.
- Reuse Algorithm 2 after modifications.

- Efficient if changes are minimal, but recomputing can be expensive for large-scale upgrades.

2. Test case

Algorithm 1: Lowest Cost Delivery Between Two Locations

Input:

```
graph = {
    "A": [("B", 4), ("C", 2)],
    "B": [("A", 4), ("C", 1), ("D", 5)],
    "C": [("A", 2), ("B", 1), ("D", 8), ("E", 10)],
    "D": [("B", 5), ("C", 8), ("E", 2)],
    "E": [("C", 10), ("D", 2)]
}
start = "A", end = "E"
```

Output:

- Shortest Path: ["A", "C", "B", "D", "E"]
- Cost: 11

Test case 2:

Input:

```
graph = {
    "X": [("Y", 3)],
    "Y": [("X", 3)]
}
start = "X", end = "Y"
```

Expected Output:

- Shortest Path: ["X", "Y"]
- Cost: 3

Purpose: Tests a minimal graph with a single edge.

Test Case 3: No Path Exists

Input:

```
graph = {
    "A": [("B", 1)],
    "B": [("A", 1)],
    "C": []
```

```
}
start = "A", end = "C"
```

Expected Output:

- Shortest Path: []
- Cost: Infinity

Purpose: Tests edge case where start and end nodes are in disconnected components.

Test Case 4: Multiple Equal-Cost Paths

Input:

```
graph = {
    "A": [("B", 2), ("C", 2)],
    "B": [("A", 2), ("D", 2)],
    "C": [("A", 2), ("D", 2)],
    "D": [("B", 2), ("C", 2)]
}
start = "A", end = "D"
```

Expected Output:

- Shortest Path: ["A", "B", "D"] (or ["A", "C", "D"])
- Cost: 4

Purpose: Tests handling multiple shortest paths with equal cost.

Algorithm 2: Best Path from the Hub

Test Case 1: Provided Example

Input:

```
graph = {
    "A": [("B", 4), ("C", 2)],
    "B": [("A", 4), ("C", 1), ("D", 5)],
    "C": [("A", 2), ("B", 1), ("D", 8), ("E", 10)],
    "D": [("B", 5), ("C", 8), ("E", 2)],
    "E": [("C", 10), ("D", 2)]
}
hub = "A"
```

Rayner Sarmiento
Algorithms
04/20/25

Expected Output:

- MST: [("A", "C", 2), ("C", "B", 1), ("D", "E", 2), ("B", "D", 5)]
- Cost: 10

Test Case 2: Linear Graph

Input:

```
graph = {
    "A": [("B", 1)],
    "B": [("A", 1), ("C", 2)],
    "C": [("B", 2)]
}
hub = "A"
```

Expected Output:

- MST: [("A", "B", 1), ("B", "C", 2)]
- Cost: 3

Purpose: Tests a simple, linear graph structure.

Test Case 3: Disconnected Graph

Input:

```
graph = {
    "A": [("B", 1)],
    "B": [("A", 1)],
    "C": []
}
hub = "A"
```

Expected Output:

- **Error:** "Graph is disconnected"

Purpose: Tests edge case where the graph is not fully connected.

Test Case 4: Dense Graph with Equal Weights

Input:

```
graph = {
  "A": [("B", 1), ("C", 1)],
```

```
    "B": [("A", 1), ("C", 1)],
    "C": [("A", 1), ("B", 1)]
}
hub = "A"
```

Expected Output:

- MST: [("A", "B", 1), ("A", "C", 1)] (or equivalent)
- Cost: 2

Purpose: Tests handling a fully connected graph with equal edge weights.

Algorithm 3: Dynamic Network Changes

Test Case 1: Provided Example

Input:

```
graph = {
    "A": [("B", 4), ("C", 2)],
    "B": [("A", 4), ("C", 1), ("D", 5)],
    "C": [("A", 2), ("B", 1), ("D", 8), ("E", 10)],
    "D": [("B", 5), ("C", 8), ("E", 2)],
    "E": [("C", 10), ("D", 2)]
}
hub = "A"
edges_to_remove = ["C-E"]
edges_to_add = [("B", "E", 3)]
```

Expected Output:

- MST: [("A", "C", 2), ("C", "B", 1), ("D", "E", 2), ("B", "E", 3)]
- Cost: 8

Test Case 2: Remove All Edges to a Node

Input:

```
graph = {
    "A": [("B", 1), ("C", 2)],
    "B": [("A", 1)],
    "C": [("A", 2)]
```

```
}
hub = "A"
edges_to_remove = ["A-B", "A-C"]
edges_to_add = []
```

Expected Output:

- **Error:** "Graph is disconnected"

Purpose: Tests edge case where removals disconnect the graph.

Test Case 3: Add Redundant Edge

Input:

```
graph = {
    "A": [("B", 1)],
    "B": [("A", 1)]
}
hub = "A"
edges_to_remove = []
edges_to_add = [("A", "B", 2)]
```

Expected Output:

- MST: [("A", "B", 1)]
- Cost: 1

Purpose: Tests handling adding an edge that doesn't affect the MST (higher weight).

Test Case 4: Add New Node via Edge

Input:

```
graph = {
    "A": [("B", 1)],
    "B": [("A", 1)]
}
hub = "A"
edges_to_remove = []
edges_to_add = [("B", "C", 2)]
```

Expected Output:

- MST: [("A", "B", 1), ("B", "C", 2)]
- Cost: 3

Purpose: Tests dynamic addition of a new node and edge.

3. Justification of Test Cases

- **Algorithm 1:** Multiple paths are tested to verify cost accuracy and the case where the start node is equal to the destination node is included.
- **Algorithm 2:** Evaluate total cost efficiency from a central hub. The limit case ensures robustness with minimal input.
- **Algorithm 3:** Assesses the algorithm's ability to handle modifications in real-time. Borderline cases validate their behavior in the face of total disconnections.