# Isaiah Lugo

CSM IV – Algorithms

Week 11 – Final Project Part 1

# Algorithm Design and Test Cases for Delivery Optimization

# Algorithm 1: Lowest Cost Delivery Between Two Locations

## Pseudocode

Algorithm Dijkstra(graph, start, end):  
 Create a priority queue (min-heap) PQ  
 Create a dictionary DISTANCE initialized to infinity for all nodes, except START set to 0  
 Create a dictionary PREVIOUS to track path  
 Add (0, START) to PQ  
  
 While PQ is not empty:  
 Extract (current\_cost, current\_node) from PQ  
 If current\_node is the END node:  
 Break (Shortest path found)  
  
 For each neighbor, weight in graph[current\_node]:  
 new\_cost = current\_cost + weight  
 If new\_cost < DISTANCE[neighbor]:  
 DISTANCE[neighbor] = new\_cost  
 PREVIOUS[neighbor] = current\_node  
 Add (new\_cost, neighbor) to PQ  
  
 Construct the shortest path by backtracking using PREVIOUS  
 Return the shortest path and DISTANCE[END]

## Efficiency Considerations

- Uses Dijkstra’s Algorithm (O((V + E) log V) with a priority queue)  
- Handles weighted graphs effectively  
- Scales well for large networks  
- Limitation: Cannot handle negative weights

## Test Cases and Justification

Basic Case:  
Input: example\_graph, start=A, end=E  
Expected Output: Shortest Path: A -> C -> B -> D -> E, Cost: 11  
Justification: Tests the algorithm’s ability to find the optimal path with varying weights.

Edge Cases:  
1. Single Edge Path: Input: start=A, end=B  
Output: A -> B, Cost: 4  
Justification: Confirms algorithm recognizes direct routes.  
2. Disconnected Nodes: Input: Graph with isolated node F, start=A, end=F  
Output: No path or cost = infinity  
Justification: Ensures the algorithm handles unreachable destinations gracefully.  
3. Equal Weights: Input: Graph where all edge weights are equal  
Output: Any shortest path  
Justification: Tests algorithm's behavior with uniform edge weights.

# Algorithm 2: Best Path from the Hub (Minimum Spanning Tree - MST)

## Pseudocode

Algorithm Prim(graph, start):  
 Create an empty set MST\_edges  
 Create a priority queue (min-heap) PQ  
 Create a set VISITED  
 Add (0, START, None) to PQ (cost, node, parent)  
  
 While PQ is not empty and len(VISITED) < total\_nodes:  
 Extract (cost, current\_node, parent) from PQ  
 If current\_node is already visited, continue  
  
 Add current\_node to VISITED  
 If parent is not None, add (parent, current\_node, cost) to MST\_edges  
  
 For each neighbor, weight in graph[current\_node]:  
 If neighbor is not in VISITED:  
 Add (weight, neighbor, current\_node) to PQ  
  
 Return MST\_edges and total cost

## Efficiency Considerations

- Uses Prim’s Algorithm (O((V + E) log V) with a priority queue)  
- Guarantees minimum cost for connecting all nodes  
- Avoids cycles  
- Limitation: Requires connected graph

## Test Cases and Justification

Basic Case:  
Input: example\_graph, start=A  
Expected Output: MST: [(A, C, 2), (C, B, 1), (D, E, 2), (B, D, 5)], Cost: 10  
Justification: Validates full MST creation with the correct total cost.

Edge Cases:  
1. Single Node Graph: Input: graph={"A": []}, start=A  
Output: Empty MST  
Justification: Confirms algorithm can handle trivial input  
2. Graph Already an MST: Input: Tree structure input  
Output: Same edges as input  
Justification: Ensures no extra edges are added  
3. Graph with Cycles: Input: Graph with multiple loops  
Output: Cycle-free MST  
Justification: Tests cycle-avoidance property

# Algorithm 3: Dynamic Network Changes

## Pseudocode

Algorithm DynamicMST(graph, start, remove\_edges, add\_edges):  
 Remove edges from graph  
 Add new edges to graph  
 Run Prim(graph, start) to compute MST  
 Return updated MST

## Efficiency Considerations

- Efficient for batch updates via re-running MST  
- Trades off recomputation for simplicity  
- Scales well for real-world road changes  
- Limitation: Not incremental—rebuilds from scratch

## Test Cases and Justification

Basic Case:  
Input: example\_graph, start=A, remove\_edges=["C-E"], add\_edges=[("B", "E", 3)]  
Expected Output: MST: [(A, C, 2), (C, B, 1), (D, E, 2), (B, E, 3)], Cost: 8  
Justification: Validates dynamic changes are reflected in MST

Edge Cases:  
1. Removing Essential Edges: Input: remove edge critical to connectivity  
Output: New MST with different structure  
Justification: Ensures MST adapts correctly  
2. Adding Redundant Edges: Input: add edge that’s not used in MST  
Output: Same MST as before  
Justification: Confirms algorithm maintains optimality  
3. Removing All Edges: Input: remove all edges  
Output: Empty MST  
Justification: Confirms that no connectivity yields no MST

# Test Case Justification

Each test case ensures:  
- The correctness of the algorithm  
- Handling of edge cases (single nodes, cycles, disconnects)  
- Efficiency in real-world scenarios  
  
Algorithm 1 Justification:  
- Covers all scenarios from direct paths to unreachable nodes.  
- Ensures Dijkstra’s path selection is cost-based, not node-count based.  
  
Algorithm 2 Justification:  
- Proves Prim’s algorithm forms the minimal-cost connection.  
- Includes trivial graphs and potential cycles to confirm robustness.  
  
Algorithm 3 Justification:  
- Verifies system resilience to changes.  
- Demonstrates adaptability and proper rebuilding under updates.