Rayner Sarmiento
Algorithms
May 3, 2025

Final Project Report: Al-GO-rithms Delivery Challenge

Overview

This Project aims to apply a different algorithm for a delivery company that faces the challenge of optimizing its delivery network during the high season. The network consists of locations (nodes) connected by roads (edges) with their corresponding travel costs. To address this problem, the implementation of three main algorithms is used:

- Algorithm 1: Finding the lowest-cost delivery path between two locations.
- Algorithm 2: Computing the minimum spanning tree (MST) to connect all locations from a hub with minimal cost.
- Algorithm 3: Adapting the MST to dynamic changes in the network, such as road closures or new connections.

Detailed Explanation of the Algorithms

Algorithm 1: Lowest Cost Delivery Between Two Locations

Objective: The goal of this first algorithm is to find the shortest (lowest-cost) path between an initial node and an end node.
Method: To solve this problem, Dijkstra's algorithm was implemented that maintains a priority queue to explore nodes with the shortest tentative distance, thus guaranteeing the search for the optimal route.

The logic implemented for this algorithm is as follows:

- Initialize distances to infinity, except for the start node (distance 0).
- Use a priority queue to select the node with the smallest current distance.
- For each unvisited neighbor, compute the new distance via the current node and update if shorter.
- Reconstruct the path using predecessors when the end node is reached.
- Return an empty path and infinite cost if no path exists.

Rayner Sarmiento
Algorithms
May 3, 2025

Efficiency:

- Time Complexity: O((V + E) log V), where V is the number of nodes and E is the number of edges, due to priority queue operations.
- Space Complexity: O(V) for distances, predecessors, and the visited set.

Algotithm 2: Minimum Spanning Tree MST

Objectives: Calculate the Minimum Spanning Tree (MST) to connect all locations from a hub with the lowest total cost.
Method: To solve this problem, the Prim algorithm is implemented, which starts from the central node and intelligently selects the edge with the least weight that connects a visited node with an unvisited one. A priority queue ensures efficient edge selection.

The logic implemented for this Algorithm is as follows:

- Initialize distances to infinity, except for the start node (distance 0).
- Use a priority queue to select the node with the smallest current distance.
- For each unvisited neighbor, compute the new distance via the current node and update if shorter.
- Reconstruct the path using predecessors when the end node is reached.
- Return an empty path and infinite cost if no path exists.

Efficiency:

- Time Complexity: O((V + E) log V), where V is the number of nodes and E is the number of edges, due to priority queue operations.
- Space Complexity: O(V) for distances, predecessors, and the visited set..

Algorithm 3: Dynamic Network Changes

Objectives: The objective of this algorithm is to update the MST after removing and adding borders to the chart.
Methods: The algorithm creates a copy of the graph, processes edge removals and additions, and reuses Prim's Algorithm (Algorithm 2) to

compute the new MST. It checks for graph connectivity by ensuring the MST includes all nodes.

The logic behind this algorithm consists of:

- Initialize distances to infinity, except for the start node (distance 0).
- Use a priority queue to select the node with the smallest current distance.
- For each unvisited neighbor, compute the new distance via the current node and update if shorter.
- Reconstruct the path using predecessors when the end node is reached.
- Return an empty path and infinite cost if no path exists. Efficiency:
- Time Complexity: O((V + E) log V), where V is the number of nodes and E is the number of edges, due to priority queue operations.
- Space Complexity: O(V) for distances, predecessors, and the visited set. Limitations: Assumes non-negative weights and may require optimization for very large graphs.

Code Implementation

The following code is a complete implementation of the algorithms written in Python from the final_project.py file:

```
"""
Final Project: Al-GO-rithms Delivery Challenge
Author: Rayner Sarmiento
"""
import heapq

from collections import defaultdict

def algorithm_1(graph:dict, start:str, end:str):
    """Finds the shortest path between start and end using Dijkstra's
Algorithm."""

    #Initialize distances with infinity for all nodes except start
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    #Tracking predecessors to reconstruct the path
    predecessors = {node: None for node in graph}
    #priority queue to store distance, node pairs
```

Rayner Sarmiento
Algorithms
May 3, 2025

```
    priority_queue = [(0, start)]
    #set to track visited nodes
    visited = set()

    while priority_queue:
        #Extract node with minimum distance
        current_distance, current_node = heapq.heappop(priority_queue)
        #skip if already visited
        if current_node in visited:
            continue
        visited.add(current_node)
        #stop if reached end node
        if current_node== end:
            #reconstruct path
            path = []
            current = end
            while current is not None:
                path.append(current)
                current = predecessors[current]
            path.reverse()
            return path, distances[end]

        #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
                #Update if a shorter path is found
                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance
                    predecessors[neighbor] = current_node
                    heapq.heappush(priority_queue, (new_distance, neighbor))
    #If no Path exists to the end node
    return [],float('inf')

#Algorithm 2
def algorithm_2(graph: dict, hub:str):
    """Computes the Minimum Spanning Tree(MST)"""
    #Check if hub exits in the graph
    if hub not in graph:
        return [], float('inf')

    #initialize MST and priority queue
    mst = []
```

Rayner Sarmiento
Algorithms
May 3, 2025

```
    priority_queue = [(0,None,hub)] #(weight, parent, node)
    visited = set()
    total_cost = 0

    while priority_queue:
        #Extract edge with minimum weight
        weight, parent, current_node = heapq.heappop(priority_queue)
        #Skip if already visited
        if current_node in visited:
            continue
        visited.add(current_node)
        # Add edge to MST skip for the hub as it has no parent
        if parent is not None:
            mst.append((parent, current_node, weight))
            total_cost += weight

            #explore neighbors
        for neighbor, weight in graph[current_node]:
            if neighbor not in visited:
                heapq.heappush(priority_queue, (weight, current_node, neighbor))

    # Check if graph is fully connected
    if len(visited) != len(graph):
        return [], float('inf')  # Graph is disconnected

    return mst, total_cost


#Algorithm 3

def algorithm_3(graph: dict, hub:str, edges_to_remove:list, edges_to_add:list):
    """Updates the graph by removing and adding edges, then computes the MST
using prim's Algorithm
    Args:
    graph: dictionary
    hub: string
    edges_to_remove: list of strings
    edges_to_add: list of tuples
    """
    #Create a copy of the graph
    updated_graph = defaultdict(list)
    for node in graph:
        for neighbor, weight in graph[node]:
            updated_graph[node].append((neighbor, weight))
            updated_graph[neighbor] #ensure all nodes exist in the graph
```

Rayner Sarmiento
Algorithms
May 3, 2025

```
    #Process for edges removal
    for edge in edges_to_remove:
        node_1, node_2 = edge.split('-')
        if node_1 in updated_graph and node_2 in updated_graph:
            updated_graph[node_1] = [(n, w) for n, w in updated_graph[node_1] if
n != node_2]
            updated_graph[node_2] = [(n, w) for n, w in updated_graph[node_2] if
n != node_1]

    #Process for edges additions
    for node_1, node_2, weight in edges_to_add:
        updated_graph[node_1].append((node_2, weight))
        updated_graph[node_2].append((node_1, weight))
    #convert to dict for compatibility with algorithm_2
    updated_graph = dict(updated_graph)
    #Compute MST on updated graph
    mst, total_cost = algorithm_2(updated_graph, hub)
    # Check if graph is still connected
    if len(mst) < len(updated_graph) - 1:
        return [], float('inf')  # Graph is disconnected

    return mst, total_cost


# testing
if __name__ == "__main__":

    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)]
    }

    # Test Algorithm 1
    print("=== Testing Algorithms fuctionalities ")

    path, cost = algorithm_1( graph, "A", "E")
    print(f"Algorithm 1 (A -> E): Path: {path}, Cost: {cost}")
    path, cost = algorithm_1( graph, "A", "B")
    print(f"Algorithm 1 (A -> B): Path: {path}, Cost: {cost}")

    # Test Algorithm 2
```

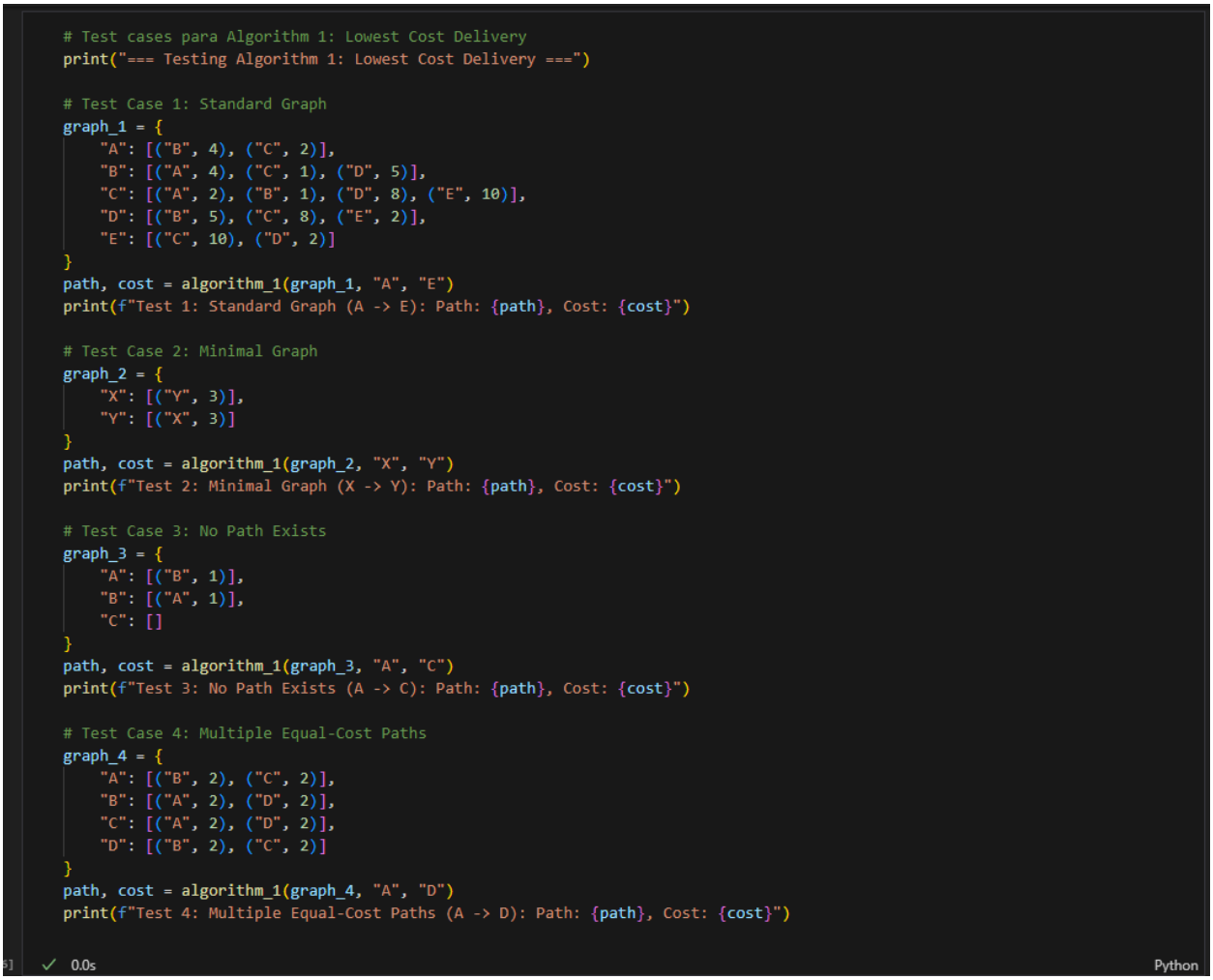
Rayner Sarmiento
Algorithms
May 3, 2025

```
    mst, cost = algorithm_2( graph, "A")
    print(f"Algorithm 2 (Hub A): MST: {mst}, Cost: {cost}")

    # Test Algorithm 3
    mst, cost = algorithm_3( graph, "A", ["C-E"], [("B", "E", 3)])
    print(f"Algorithm 3 (Hub A, remove C-E, add B-E=3): MST: {mst}, Cost:
{cost}")


    #Test Case from the part-1

            # Test cases para Algorithm 1: Lowest Cost Delivery
    print("=== Testing Algorithm 1: Lowest Cost Delivery ===")

    # Test Case 1: Standard Graph
    graph_1 = {
        "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)]
    }
    path, cost = algorithm_1(graph_1, "A", "E")
    print(f"Test 1: Standard Graph (A -> E): Path: {path}, Cost: {cost}")

    # Test Case 2: Minimal Graph
    graph_2 = {
        "X": [("Y", 3)],
        "Y": [("X", 3)]
    }
    path, cost = algorithm_1(graph_2, "X", "Y")
    print(f"Test 2: Minimal Graph (X -> Y): Path: {path}, Cost: {cost}")

    # Test Case 3: No Path Exists
    graph_3 = {
        "A": [("B", 1)],
        "B": [("A", 1)],
        "C": []
    }
    path, cost = algorithm_1(graph_3, "A", "C")
    print(f"Test 3: No Path Exists (A -> C): Path: {path}, Cost: {cost}")

    # Test Case 4: Multiple Equal-Cost Paths
    graph_4 = {
        "A": [("B", 2), ("C", 2)],
```

Rayner Sarmiento
Algorithms
May 3, 2025

```
        "B": [("A", 2), ("D", 2)],
        "C": [("A", 2), ("D", 2)],
        "D": [("B", 2), ("C", 2)]
    }
    path, cost = algorithm_1(graph_4, "A", "D")
    print(f"Test 4: Multiple Equal-Cost Paths (A -> D): Path: {path}, Cost:
{cost}")
```

Test Cases and Results:



```
# Test cases para Algorithm 1: Lowest Cost Delivery
print("=== Testing Algorithm 1: Lowest Cost Delivery ===")

# Test Case 1: Standard Graph
graph_1 = {
    "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)]
}
path, cost = algorithm_1(graph_1, "A", "E")
print(f"Test 1: Standard Graph (A -> E): Path: {path}, Cost: {cost}")

# Test Case 2: Minimal Graph
graph_2 = {
    "X": [("Y", 3)],
    "Y": [("X", 3)]
}
path, cost = algorithm_1(graph_2, "X", "Y")
print(f"Test 2: Minimal Graph (X -> Y): Path: {path}, Cost: {cost}")

# Test Case 3: No Path Exists
graph_3 = {
    "A": [("B", 1)],
    "B": [("A", 1)],
    "C": []
}
path, cost = algorithm_1(graph_3, "A", "C")
print(f"Test 3: No Path Exists (A -> C): Path: {path}, Cost: {cost}")

# Test Case 4: Multiple Equal-Cost Paths
graph_4 = {
    "A": [("B", 2), ("C", 2)],
    "B": [("A", 2), ("D", 2)],
    "C": [("A", 2), ("D", 2)],
    "D": [("B", 2), ("C", 2)]
}
path, cost = algorithm_1(graph_4, "A", "D")
print(f"Test 4: Multiple Equal-Cost Paths (A -> D): Path: {path}, Cost: {cost}")
```

0.0s Python

Rayner Sarmiento
Algorithms
May 3, 2025

```
[26]  ✓ 0.0s

...   === Testing Algorithm 1: Lowest Cost Delivery ===
      Test 1: Standard Graph (A -> E): Path: ['A', 'C', 'B', 'D', 'E'], Cost: 10
      Test 2: Minimal Graph (X -> Y): Path: ['X', 'Y'], Cost: 3
      Test 3: No Path Exists (A -> C): Path: [], Cost: inf
      Test 4: Multiple Equal-Cost Paths (A -> D): Path: ['A', 'B', 'D'], Cost: 4
```

```python
# Test cases para Algorithm 2: Best Path from the Hub
print("\n=== Testing Algorithm 2: Best Path from the Hub ===")

# Test Case 1: Standard Graph
mst, cost = algorithm_2(graph_1, "A")
print(f"Test 1: Standard Graph (Hub A): MST: {mst}, Cost: {cost}")

# Test Case 2: Linear Graph
graph_5 = {
    "A": [("B", 1)],
    "B": [("A", 1), ("C", 2)],
    "C": [("B", 2)]
}
mst, cost = algorithm_2(graph_5, "A")
print(f"Test 2: Linear Graph (Hub A): MST: {mst}, Cost: {cost}")

# Test Case 3: Disconnected Graph
graph_6 = {
    "A": [("B", 1)],
    "B": [("A", 1)],
    "C": []
}
mst, cost = algorithm_2(graph_6, "A")
print(f"Test 3: Disconnected Graph (Hub A): MST: {mst}, Cost: {cost}")

# Test Case 4: Dense Graph with Equal Weights
graph_7 = {
    "A": [("B", 1), ("C", 1)],
    "B": [("A", 1), ("C", 1)],
    "C": [("A", 1), ("B", 1)]
}
mst, cost = algorithm_2(graph_7, "A")
print(f"Test 4: Dense Graph with Equal Weights (Hub A): MST: {mst}, Cost: {cost}")
```

```
2]  ✓ 0.0s                                                    Python
```

```
=== Testing Algorithm 2: Best Path from the Hub ===
Test 1: Standard Graph (Hub A): MST: [('A', 'C', 2), ('C', 'B', 1), ('B', 'D', 5), ('D', 'E', 2)], Cost: 10
Test 2: Linear Graph (Hub A): MST: [('A', 'B', 1), ('B', 'C', 2)], Cost: 3
Test 3: Disconnected Graph (Hub A): MST: [], Cost: inf
Test 4: Dense Graph with Equal Weights (Hub A): MST: [('A', 'B', 1), ('A', 'C', 1)], Cost: 2
```

Rayner Sarmiento
Algorithms
May 3, 2025

```python
# Test cases para Algorithm 3: Dynamic Network Changes
print("\n=== Testing Algorithm 3: Dynamic Network Changes ===")

# Test Case 1: Standard Graph with Modifications
mst, cost = algorithm_3(graph_1, "A", ["C-E"], [("B", "E", 3)])
print(f"Test 1: Standard Graph with Modifications (Hub A): MST: {mst}, Cost: {cost}")

# Test Case 2: Remove All Edges to a Node
graph_8 = {
    "A": [("B", 1), ("C", 2)],
    "B": [("A", 1)],
    "C": [("A", 2)]
}
mst, cost = algorithm_3(graph_8, "A", ["A-B", "A-C"], [])
print(f"Test 2: Remove All Edges to a Node (Hub A): MST: {mst}, Cost: {cost}")

# Test Case 3: Add Redundant Edge
graph_9 = {
    "A": [("B", 1)],
    "B": [("A", 1)]
}
mst, cost = algorithm_3(graph_9, "A", [], [("A", "B", 2)])
print(f"Test 3: Add Redundant Edge (Hub A): MST: {mst}, Cost: {cost}")

# Test Case 4: Add New Node via Edge
mst, cost = algorithm_3(graph_9, "A", [], [("B", "C", 2)])
print(f"Test 4: Add New Node via Edge (Hub A): MST: {mst}, Cost: {cost}")
```

✓ 0.0s Python

```
=== Testing Algorithm 3: Dynamic Network Changes ===
Test 1: Standard Graph with Modifications (Hub A): MST: [('A', 'C', 2), ('C', 'B', 1), ('B', 'E', 3), ('E', 'D', 2)], Cost: 8
Test 2: Remove All Edges to a Node (Hub A): MST: [], Cost: inf
Test 3: Add Redundant Edge (Hub A): MST: [('A', 'B', 1)], Cost: 1
Test 4: Add New Node via Edge (Hub A): MST: [('A', 'B', 1), ('B', 'C', 2)], Cost: 3
```

Challenges Faced and Solutions:

One of the main challenges presented during the implementation of algorithm 1 was the handling of multiple equal-cost path correctly and to solve this I had to read a little more about the algorithm in these situations and I discovered that naturally Dijkstra's algorithm handles this behavior naturally and I was able to check this behavior with the case tests.

I also want to mention another challenge that arose handling the updates to the graph in Algorithm 3 without modifying the original graph, so I used *defaultdict* to create a copy of the graph and apply modification making sure that the original graph remained intact.

Finally, optimizing priority queue operations for efficiency, but fortunately I was able to handle it using the Python *heapq* module .

Rayner Sarmiento
Algorithms
May 3, 2025

Conclusions and Future Improvements

Conclusions: The implemented algorithms successfully optimize Al-GO-rithms' delivery network. Algorithm 1 efficiently finds the lowest-cost path, Algorithm 2 ensures minimal-cost connectivity from the hub, and Algorithm 3 adapts to dynamic changes. The test cases confirmed robustness across standard and edge cases, drawing parallels to network protocols like OSPF and STP.

Future Improvements:

- Implement Kruskal's Algorithm as an alternative to Prim's for sparse graphs, potentially reducing time complexity in certain scenarios.
- Optimize Algorithm 3 by incrementally updating the MST instead of recomputing it, improving performance for frequent updates.
- Add visualization tools (e.g., using Matplotlib or NetworkX) to display graphs and paths for debugging and presentation.
- Explore parallel processing for large graphs to reduce computation time.