Olivia Pita Final Project Report 4/21/2025
CSM IV

Problem Overview

"The delivery company Al-GO-rithms has hired you to optimize their delivery network during the busiest season of the year. They operate across a network of locations (nodes) connected by roads (edges). Each road has an associated travel cost based on distance, time, or fuel usage (weight)."
Three algorithms are needed. One to find the most efficient route from one location to another. A second to find the most efficient route between all locations in the network, starting from a specified hub location. A third to remove or add roads to the network, as traffic conditions change. The network will be represented by a weighted graph.

Algorithms Explained

Algorithm 1: most efficient route from one location to another

Input: weighted graph, start location, end location.
Output: shortest path from start to end, total cost.
Process:

- Create variables:
    - Create "visited" array of length = size of graph, to track which nodes have been visited.
    - Create "previous" array of length = size of graph, to track the path taken from start to end.
    - Create "distances" array of length = size of graph, to track the currently found shortest distance from the start to *each* other node. The distance from start to itself is 0.
    - Create "queue" array initialized with the tuple (start, 0), to track which edges still need processing.
        - Tuple "edges" are in the form (node2, cost), with node1 being …
    - Create "idx_map" dictionary, which acts as a translator between the graph's keys and their index equivalents in the above arrays.
        - E.g. *idx_map = {'A': 0, 'B': 1, 'C': 2, 'D': 3}*
        - *visited[idx_map.get('A')] == visited[0]*
- While there are items in the queue, and the next item to be processed != end:
    - *node,dis_from_start = queue.pop(0)*

        - *visited[idx_map.get(node)] = True*
        - *connections = graph[node]* # list of tuples of form (node2, cost)
        - In the list of node's direct connections, for each connection, if it provides a cheaper path to a node2 than currently available, record the shorter path.
            - Update *previous[idx_map.get(node2)]* and *distances[idx_map.get(node2)]*.
            - *queue.append((node2, new_distance))* # next node and its distance from start
- Create string describing path, working backworks to create a stack of nodes visited from end to start:
    - *stack = [end]*
    - *while stack[-1] != start:*
        - Append to stack the "previous" of the current last item of the stack. Note: because this adds a new last item, an iterator is not needed.
    - Reverse the stack, and convert it to a string with each item separated by " -> ". Now the path in in start-to-end order and ready to print.
- *return distances[idx_map.get(end)], path_string*

Algorithm 2

Input: weighted graph, start location.
Output: MST of most efficient path between all locations.
Process:

- Create dictionary that translates each node's string name into an index number.
- Create variables:
    - Create "visited" array of length = size of graph, to track which nodes have been visited.
    - Create empty "stack" array, to process edges FILO. An "edge" is a tuple of form (node1, node2, cost).
    - Create empty "MST" array.
    - Create "node" variable, initialized to start.
    - Create "idx_map" dictionary, which acts as a translator between the graph's keys and their index equivalents in the above arrays.
        - E.g. *idx_map = {'A': 0, 'B': 1, 'C': 2, 'D': 3}*
        - *visited[idx_map.get('A')] == visited[0]*
- *while not all(visited):*
    - *visited[idx_map.get(node)] =True*

  - For each unvisited node2, add the corresponding edge to the stack.
  - Find the shortest edge in the stack, working backwards to avoid index errors:
    - If node1 and node2 are both already in the MST, delete their edge from the stack.
    - Else, compare the cost of the edge and the current shortest_edge, and keep the cheaper.
  - Once the cheapest edge is found, remove it from the stack and add it to the MST.
  - *node* = node2 of *smallest_edge*
- Return MST.

Algorithm 3

Input: weighted graph, start location, edges to remove (as list of strings), edges to add (as list of tuples of format (string, string, int)).

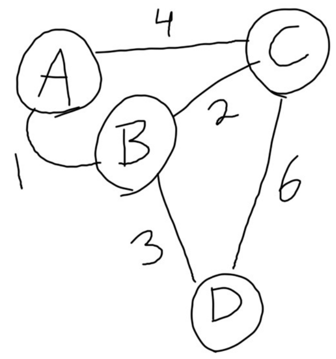
- "remove" string is of form "node1-node2", e.g. "A-B".
- "add" tuple is of form (node1, node2, cost), e.g. ("A", "B", 5).

Output: MST post-changes.

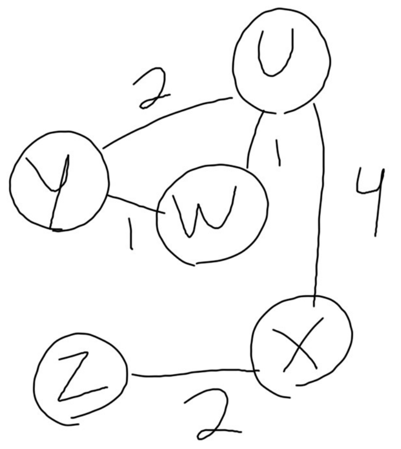
Process:

- For each edge to be removed:
  - Check each edge of node1. Delete the edge if it connects to node2.
  - Check each edge of node2. Delete the edge if it connects to node1.
- For each edge to be added:
  - Check if the edge already exists. If not, append the edge to node1's list of edges and to node2's list of edges.
- *return algorithm_2(graph, start_node)* # send the modified graph to the MST-finding algorithm, and return the result of that function as the result of this function

Test Cases



```
graph_1 = {
  "A": [("B", 1), ("C", 4)],
  "B": [("A", 1), ("C", 2), ("D", 3)],
  "C": [("A", 4), ("B", 2), ("D", 6)],
  "D": [("B", 3), ("C", 6)],
}
```



```
graph_2 = {
  "Z": [("X", 2)],
  "Y": [("W", 1), ("U", 2)],
  "X": [("Z", 2), ("U", 4)],
  "W": [("Y", 1), ("U", 1)],
  "U": [("Y", 2), ("X", 4), ("W", 1)],
}
```

	Input	**Output**	**Purpose**
Algorithm1, Test1	graph_1, “A”, “C”	Cost: 3 Path: A -> B -> C	Does the algorithm recognize that the most direct path is not always the most efficient?
Algorithm1, Test2	graph_1, “A”, “B”	Cost: 1 Path: A -> B	Does the algorithm recognize that the most direct path *can* be the most efficient?
Algorithm1, Test3	graph_1, “D”, “C”	Cost: 5 Path: D -> B -> C	Does the algorithm work when a decision between "shortest" and "directest" is not from the start node?
Algorithm2, Test1	graph_1, “A”	MST: [(A, B, 1), (B, C, 2), (B, D, 3)]	Generic test.
Algorithm2, Test2	graph_1, “D”	MST: [(D, B, 3), (B, A, 1), (B, C, 2)]	Same graph but starting from a different node.

Algorithm2, Test3	graph_2, “U”	MST: [(U, W, 1), (W, Y, 1), (U, X, 4), (X, Z, 2)]	Different graph, where the start node ends up in the middle of the path.
Algorithm3, Test1	graph_1, “A”, [“A-B”], [(“A”, “D”, 3)]	MST: [(A, D, 3), (D, B, 3), (B, C, 2)]	Remove one route, add one route.
Algorithm3, Test2	graph_1, “A”, [“A-B”, “B-D”], []	MST: [(A, C, 4), (C, B, 2), (C, D, 6)]	Remove two routes.
Algorithm3, Test3	graph_2, “U”, [], [(“Z”, “Y”, 3), (“W”, “X”, 1)]	MST: [(U, W, 1), (W, X, 1), (W, Y, 1), (X, Z, 2)]	Different graph, add two routes.

Challenges

Multiple times, I found that my algorithms for finding the shortest path ended too soon. Such as in Algorithm 2, test 3, using graph 2, only the first two edges were in the output, because the process traveled from U to W to Y, then got stuck because U was already visited. I had been deleting all edges from the stack while searching for the cheapest. That meant that U’s connection to X was forgotten. To fix this, I moved *stack.pop(i)* outside of the search loop, to only delete the cheapest edge. That caused another challenge. *stack.pop(i)* no longer targeted the correct edge. I found another method that deletes an entry in a list based on its contents. And so the line of code changed to *stack.remove(cheapest_edge)*.

Conclusion

I am proud of the work I accomplished. It was complicated to write a multifaceted algorithm. I’m particularly proud for recognizing from the start that I’d need a “translator” between the graph dictionary and the array indexes. And I learned that there’s a one-line code to achieve that! I thought I’d have to write a whole separate function.

I’m sure improvements could be made in efficiency. I was prioritizing writing short, clean code over understanding the time complexity. Maybe there are ways to speed up the while loops that search through the stacks/queues.