Algorithm Design

Algorithm 1

Input: weighted graph, start location, end location.

Output: shortest path from start to end, total cost.

Process:

- Create dictionary that translates each node's string name into an index number.
  - # Note: put the creation of the dictionary in another function that returns the dictionary object. This way, Algorithm 2 can utilize the same function. Scalability.
  - # Note: in this design doc, I refer to the dictionary as if it were a function called translate().
  - For each key in graph, assign that key as the value of an integer key. # this way, data saved in the arrays (created below) can be referenced with by index instead of by name.
- 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 "node" variable, initialized to start_node.
  - Create "cost" variable, initialized to 0.
- *while node != end_node:*
  - *visited[node] =True*
  - In the list of node's direct connections, identify the cheapest unvisited connection.
  - *previous[translate(connected_node)] = node* # current node is the previous of the node it is connected to
  - *cost += connection_cost*
  - *node = connected_node*
- Create string describing path:
  - *stack = [previous[end_node]]*
  - *while stack[-1] != start_node:*
    - *stack.append(previous[stack[-1]])* # work backwards to create a list of nodes visited from end to start
  - path_string = stack's values FILO, separated by "->".
- Return path_string and cost.

Algorithm 2

Input: weighted graph, start location.

Output: MST of most efficient path between all locations.
Process (Prim's Algorithm):
- 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_node.
- *while not all(visited):*
  - *visited[node] =True*
  - *smallest_edge = (None, None, float("inf"))* # this is a new variable
  - For each unvisited connected node, add the edge to that node to the stack:
    - For each of the node's direct connections, *if visited[connected_node] == False:*
      - *stack.append([node, connected_node, connection_cost])*
  - Find smallest untraveled edge in stack, working backwards to avoid index errors:
    - *i = len(stack) - 1* # i holds the tuple's index
    - *while i >= 0:*
      - *node1 = stack[i][0]* # first item in tuple
      - *node2 = stack[i][1]*
      - *cost = stack[i][2]*
      - *if visited[node1] and visited[node2]:* # if node1 and node2 are already connected in the MST, delete their edge from stack
        - *stack.pop(i)*
      - *elif cost < smallest_edge[2]:*
        - *smallest_edge = (node1, node2, cost)*
      - *i -= 1*
  - *MST.append(smallest_edge)*
  - *node = smallest_edge[1]*
- After completing the MST, there will be one null edge at the end of the stack, so delete it.
- Return MST.

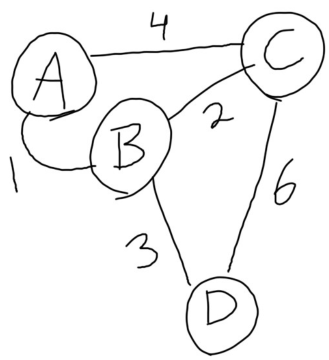
Algorithm 3
Input: weighted graph, start location, edges to remove (as list of strings), edges to add (as list of tuples of format (string,int)).
Output: MST post-changes.
Process:
- For each edge to be removed:

- Create variables:
  - node1 = first char in string
  - node2 = last char in string
- In graph, key == node1, check the first item in each tuple, and:
  - if it is a string == node2, delete that tuple.
  - Break out after deletion to avoid unnecessary checks.
- In graph, key == node2, check the first item in each tuple, and:
  - if it is a string == node1, delete that tuple.
  - Break out after deletion to avoid unnecessary checks.

- For each edge to be added:
  - Create variables:
    - node1 = first item in tuple (a string)
    - node2 = second item in tuple (a string)
    - cost = third item in tuple (an int)
  - In graph, key == node1:
    - Check the first item in each tuple; if it is a string == node2, break out to avoid having multiples edges between a pair of nodes.
    - If a match is not found, append a new tuple (node1, node2, cost). Then:
      - In graph, key == node2, append a new tuple (node1, node2, cost).
      - # Notice that the algorithm does not check to see if node2 already has a tuple with node1. It is assumed after verifying that node1 does not have a tuple with node2. This makes the code more efficient as long as the input data is good. However, bad input data would cause the algorithm to miscalculate.
- *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

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



Algorithm 1

#Test 1: Does the algorithm recognize that the most direct path is not always the most efficient?

```
# start at A and end at C
print(algorithm_1(example_graph, "A", "C"))
# Shortest Path: A -> B -> C
# Cost: 3
```

#Test 2: Does the algorithm recognize that the most direct path *can* be the most efficient?

```
# start at A and end at B
print(algorithm_1(example_graph, "A", "B"))
# Shortest Path: A -> B
# Cost: 1
```

#Test 3: Does the algorithm work when *not* starting from "A"?

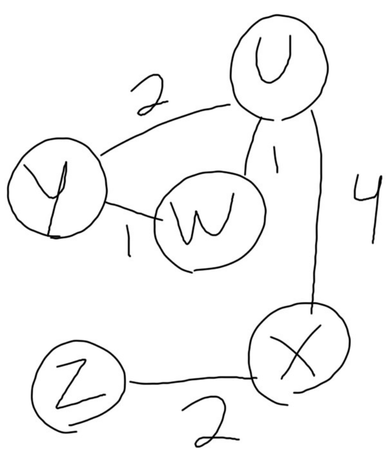
```
# start at D and end at C
print(algorithm_1(example_graph, "D", "C"))
# Shortest Path: D -> B -> C
# Cost: 5
```

Algorithm 2

#Test 1: Generic test.
```
# start at A (the hub location)
print(algorithm_2(example_graph, "A"))
# MST: [(A, B, 1), (B, C, 2), (C, D, 6)]
```

#Test 2: Same graph but starting from a different node.
```
# start at D (the hub location)
print(algorithm_2(example_graph, "D"))
# MST: [(D, B, 3), (B, A, 1), (A, C, 4)]
```

#Test 3: Different graph, where the start node ends up in the middle of the path.
```
example_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)],
}
```




```
# start at U (the hub location)
print(algorithm_2(example_graph, "U"))
# MST: [(U, W, 1), (W, Y, 1), (U, X, 4), (X, Z, 2)]
```

Algorithm 3

#Test 1: Remove one route, add another.

```
# initial graph is the first one in this document
# starts at A (the hub location), removed A-B edge, and adds A-D
edge with a weight of 3
print(algorithm_3(example_graph, "A", ["A-B"], [("A", "D", 3)]))
# MST: [(A, D, 3), (D, B, 3), (B, C, 2)]
```

#Test 2: Remove two routes.

```
# initial graph is the first one in this document
# starts at A (the hub location), removed A-B edge and B-D edge
print(algorithm_3(example_graph, "A", ["A-B"], ["B-D"]))
# MST: [(A, C, 4), (C, B, 2), (C, D, 6)]
```

#Test 3: Different graph, add one route.

```
# initial graph is the second one in this document
# starts at U (the hub location), added Z-Y edge with weight 3
print(algorithm_3(example_graph, "U", [("Z", "Y", 3)]))
# MST: [(U, W, 1), (W, Y, 1), (Y, Z, 3), (Z, X, 2)]
```