#1 OPTIMAL PATH

Modify Dijkstra's Algorithm to compute an optimal path from s to t in $O(m+n)$ time. An optimal path is both the shortest path, and the path with the least number of edges. The algorithm must be able to output the actual path.

PSEUDO CODE:

```
1  int findOptimalPath(int source, int destination) {
2      boolean[] visited = new boolean[V];
3      visited[source] = true;
4
5      int[] pathWeight = new int[V];
6      pathWeight[source] = 0;
7
8      int[] numEdges = new int[V];
9      numEdges[source] = 0;
10
11     LinkedList<Integer> queue = new LinkedList<>();
12     queue.add(source);
13
14     while (queue.size() != 0) {
15         source = queue.poll();
16         Iterator<Integer> iterator = adj[source].listIterator();
17
18         while (iterator.hasNext()) {
19             int node = i.next();
20
21             if (!visited[node]) {
22                 visited[node] = true;
23                 pathWeight[node] = pathWeight[source] + node.getWeight();
24                 numEdges[node] = numEdges[source] + 1;
25                 node.previous = source;
26                 queue.add(node);
27             }
28             else if (pathWeight[node] == pathWeight[source] + node.getWeight()
29             && numEdges[node] > numEdges[source] + 1) {
30                 numEdges[node] = numEdges[source] + 1;
31                 node.previous = source;
32             }
33             else if (pathWeight[node] > pathWeight[source] + node.getWeight()) {
34                 pathWeight[node] = pathWeight[source] + node.getWeight();
35                 numEdges[node] = numEdges[source] + 1;
36                 node.previous = source;
37             }
38         }
39     }
40
41     return pathWeight[destination];
42 }
```

DESCRIPTION:

This algorithm functions similarly to Dijktra's Algorithm. As it traverses the graph G it adds any unvisited nodes to the queue and updates the weight of the path traveled from the source s to that node, and the number of edges in that path. If a node has already been visited, the algorithm checks to see if the new path costs less than the original path. If it does, the weight and number of edges of the new path is saved. If the weight of the old path and the new path are the same, the algorithm checks if the new path has less edges than the old path. If so, the weight and edge info of the path is updated. The node used to get to the current node in the iteration is store in the *previous* data member of the current node. The algorithm will return the weight of the path used to get from s to t. In order to get the actual path, add t to a stack, the iterate through the *previous* data member, beginning with t, adding each node to the stack until s is reached. Popping each item on the stack will result in the optimal path beginning with s and ending with t.

ANALYSIS:

Only constant computations are added to Dijktra's original algorithm. Therefore the total complexity of $findOptimalPath$ will be the same. The edge of every node is visited once, and each node visited once. This results in $O(m+n)$ time, where m is the number of edges, and n the number of nodes. To obtain the actual path traveled will take at worst n time, if every node in the graph is visited along the path.

$$\begin{aligned} Time : O((m+n)+n) \\ O(m+2*n) \\ O(m+n) \end{aligned}$$

#2 CONCEPTUAL QUESTIONS - GRAPHS

Answer the following questions with *yes*, *no*, or *not necessarily*, and explain your reasoning.

If the weight of every edge in G is increased by a value $\delta > 0$, then...

a: Is $\pi(s, t)$ still the shortest path from s to t?

Not necessarily. Unless $\pi(s, t)$ was the optimal path to begin with, it will not necessarily be the shortest path after increasing the weight of each edge. If G originally had two shortest paths but one had three edges while the other only had two, when the weights are increased the two paths will no longer be equal. The path with three edges will now be longer than the path with two edges.

b: Is T still the minimum spanning tree of G?

Not necessarily. Because the shortest path from any node s to any node t can change when the weight of each edge is increased, it means that the minimum spanning tree (MST) also is at risk of changing. Considering the same situation as above the MST might have originally included the path with three edges in it, but when the weights are increased the minimum spanning tree would have use the path with two edges instead.

Commentary

In order for the answer of either of the two questions above to be always be *yes*, the shortest path and MST must be constructed using the definition of an optimal shortest path. If any two paths have the same length (or weight), the algorithm must select the one with the least number of edges. Then when the weight of each edge in G is increased, $\pi(s, t)$ and T will always be the shortest path from s to t, and the MST of G.

#3 BLUE SPANNING TREE

Given a graph where each edge is either red or blue, design an algorithm to find a spanning tree with as few red edges as possible. The algorithm should run in $O((m+n)*logn)$ time.

PSUEDO CODE:

```
1  int [] minimumRedTree(int source) {
2      boolean[] visited = new boolean[V];
3      visited[source] = true;
4      boolean[] isBlue = new boolean[V];
5      isBlue[source] = false;
6      int [] parent = new int[V];
7      parent[source] = -1;
8      LinkedList<Integer> queue = new LinkedList<>();
9      queue.add(source);
10
11     while (queue.size() != 0) {
12         source = queue.poll();
13         Iterator<Integer> iterator = adj[source]. listIterator ();
14         while (iterator.hasNext()) {
15             int node = i.next();
16             if (!visited[node]) {
17                 visited[node] = true;
18                 isBlue[node] = node.isBlue();
19                 queue.add(node);
20                 parent[node] = source;
21             }
22             else if (!isBlue[node] && node.isBlue()) {
23                 isBlue[node] = node.isBlue();
24                 parent[node] = source;
25             }
26         }
27     }
28     return parent;
29 }
```

DESCRIPTION:

The *minimumRedTree* functions much like Dijktra's or Prim's Algorithm, which both run in $O(m+n)$ time. Each edge is visited once, along with at least one visit to each node. If a node is visited a second time, but this time via a blue path instead of a red path, the new path is used in the spanning tree instead. The *parent* array stores the parent of the corresponding index of the array. This allows the minimum spanning tree to be printed in $O(n)$ time (although not in tree fashion).

ANALYSIS:

As mentioned above, the algorithm itself runs in $O(n+m)$ time with an additional $O(n)$ time required to print the *parent* array.

$$Time : O(m+n)$$