Shortest Path

Dijkstra's algorithm's time complexity is O((V+E)log(V)). Where V is the number of vertices and E is the number of edges in the graph. This time complexity is true when the graph stores nodes in a priority queue. My graph does not do that. Dijkstra's algorithm's space complexity is O(V). This is true when priorities queues are used and the distances and previous of each node is stored. My program does store the distance and previous of each node. My space complexity is probably slightly worse than the O(V). My shortest path function has a time complexity of $O(V^2)$. I have two loops that could iterate through the list of nodes once.

```
72          for(auto node : nodes) {
73              if(source_name == node->get_name()) { // finds source node's addresses
74                  source_node = node;
75              }
76              node->set_distance(numeric_limits<int>::max()); // sets all node's distances to the maximum integer
77              node->set_previous(nullptr); // sets all node's previous's to nullptr
78          }
```

```
122     // this for loop creates the output string
123     for (auto node : nodes) {
124         string str_distance = to_string(node->get_distance());
125         if (node->get_distance() == numeric_limits<int>::max()) {
126             output = output + node->get_name() + " cannot reach " + source_node->get_name() + ".";
127         } else {
128             output = output + node->get_name() + "'s shortest distance to " + source_node->get_name() + " = " + str_distance + " km";
129         }
130
131         if (node->get_previous() == nullptr) {
132             if (node == source_node) {
133                 output = output + "  |  Path from " + source_node->get_name() + " to " + node->get_name() + " is " + node->get_name() + " <- " + source_node->get_name
134             } else {
135                 output = output + "\n";
136             }
137         } else {
138             output = output + "  |  Path from " + source_node->get_name() + " to " + node->get_name() + " is " + node->get_name() + " <- " + node->get_previous()->get
139             if (node->get_previous() != source_node) {
140                 GraphNode *temp = node->get_previous();
141                 while (temp != source_node) {
142                     output = output + " <- " + temp->get_previous()->get_name();
143                     temp = temp->get_previous();
144                 }
145             }
146             output = output + "\n";
147         }
148     }
```

But the while loop has another for loop within it. Both of which could iterate through the list of nodes. Resulting in a time complexity of $O(V^2)$.

```
86      // finds the shortest distance from the source node to all other nodes
87      GraphNode *current = source_node;
88      while (queue.size() != 0) { // if all nodes have been visited the algorithm should stop
89
90          for (auto edge : current->get_neighbors()) { // iterates through "current's" neighbors
91              GraphNode *dest = edge->destination; // create destination node
92              if (dest->get_distance() > current->get_distance() + edge->weight) { // if a shorter path is found
93                  dest->set_distance(current->get_distance() + edge->weight); // update destination's distance
94                  dest->set_previous(current); // update destination's previous
95              }
96          }
97
98          int count = 0;
99          int current_index = 0;
100         for (auto node : queue) {
101             if (node == current) { // finds index of the current
102                 current_index = count;
103             }
104             count += 1;
105         }
106
107         queue.erase(queue.begin() + current_index); // removes current from unvisited list
108
109         GraphNode *next_current = new GraphNode("Next Current");
110
111         for (auto node : queue) {
112             if (node->get_distance() < next_current->get_distance()) { // next current should be unvisited and have the smallest distance
113                 next_current = node;
114             }
115         }
116
117         current = next_current;
118     }
```

Minimum Spanning Tree

Prim's algorithm's time complexity is $O(V^2)$. Its space complexity is O(V). I believe that my space complexity is worse than this because I do not use priority queues. The time complexity of my minimum spanning tree function is $O(V^2)$. I have a for loop which has a time complexity of O(V).

```
161         int floating_nodes = 0;
162         for (auto node : nodes) { // checking if there are any nodes not connected to the graph
163             int neighbor_size = node->get_neighbors().size();
164             if (neighbor_size < 1) {
165                 floating_nodes += 1;
166             }
167         }
```

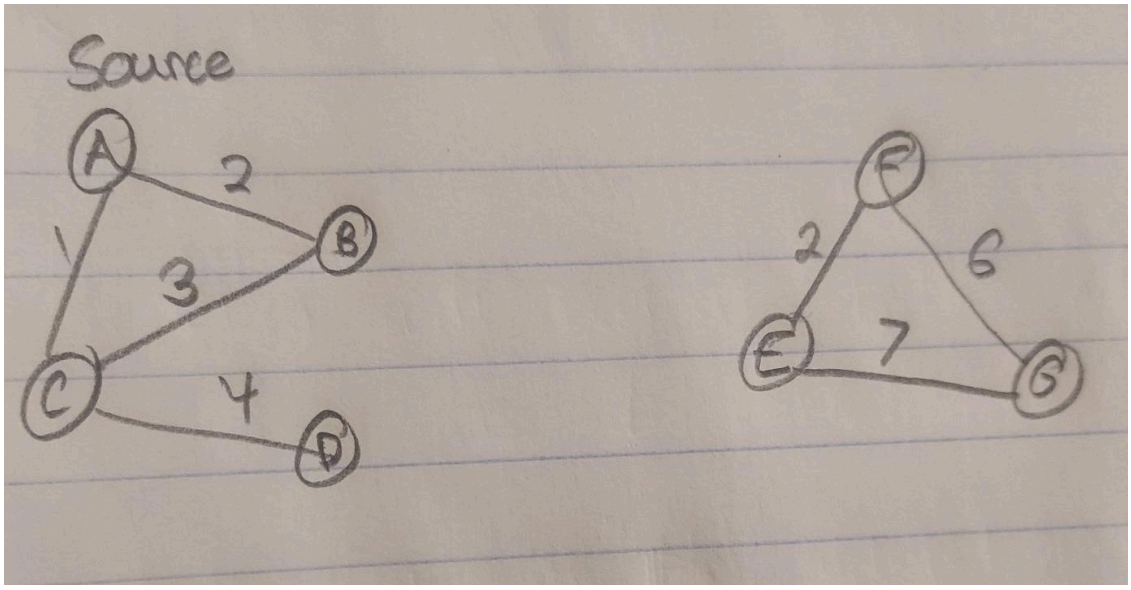
The for loop that generates the output string has a time complexity of O(E). E stands for the number of edges in the tree, not the graph.

```
219     // this for loop creates the output string
220     for (auto tree_edge : tree_edges) {
221         total_weight += tree_edge->weight;
222         count += 1;
223         one_two_count = (count % 2) + 1;
224         if (count % 2 == 0) {
225             count2 += 1;
226         }
227         string str_count = to_string(count);
228         string str_weight = to_string(tree_edge->weight);
229         string str_one_two_count = to_string(one_two_count);
230         string str_count2 = to_string(count2);
231         output = output + "Edge " + str_count2 + "." + str_one_two_count + ": weight = " + str_weight + " | source = " + tree_edge->source->get_name() + " | destinati
232     }
```

The main while loop has a time complexity of $O(V^2)$. In the worst case scenario the inner loop and the outer loop will have to iterate through all nodes in the graph.

```
169     // Prim's algorithm
170     while (visited.size() < nodes.size() - floating_nodes) { // the algorithm should stop when all connected nodes have been visited. (this loop might not work if the
171         edge *lightest_edge_1 = new edge(); // lightest edge 1
172         edge *lightest_edge_2 = new edge(); // lightest edge 2
173         if (visited.size() > 1) {
174             for (auto visited_node : visited) { // iterates through the visited nodes
175                 for (auto neighbor : visited_node->get_neighbors()) { // iterates through the visited node's neighbors
176                     bool contains = false;
177                     for (auto tree_edge : tree_edges) { // checks if neighbor is a tree edge
178                         if (tree_edge == neighbor) {
179                             contains = true;
180                         }
181                     }
182                     for (auto node : visited) { // checks if destination has already been visited
183                         if (node == neighbor->destination) {
184                             contains = true;
185                         }
186                     }
187                     if (contains == false) { // if the neighbor is not already in the tree and its destination has not been visited, then create two edges
188                         if (neighbor->weight < lightest_edge_1->weight) {
189                             lightest_edge_1 = neighbor;
190                             lightest_edge_2->weight = lightest_edge_1->weight;
191                             lightest_edge_2->destination = lightest_edge_1->source;
192                             lightest_edge_2->source = lightest_edge_1->destination;
193                         }
194                     }
195                 }
196             }
197         } else {
198             for (auto neighbor : source_node->get_neighbors()) { // finds the edge with the lowest weight on the first pass
199                 if (neighbor->weight < lightest_edge_1->weight) {
200                     lightest_edge_1 = neighbor;
201                     lightest_edge_2->weight = lightest_edge_1->weight;
```

My minimum spanning tree function will not work in all scenarios. It will only work for non-directed graphs. If there is a section of the graph that cannot be accessed by the source node, the function will not work. Example:



Add Vertex

The time complexity of my add vertex function is O(1) for most cases. This is because vectors are dynamic arrays. The push back function does not need to iterate through the entire vector to get to the end. However, if the vector's capacity is reached it may need to resize. Which will result in a time complexity of O(V).

```
33      // adds a GraphNode to the class
34      void add_node(string new_name) {
35          GraphNode *new_node = new GraphNode(new_name); // create node using inputted string
36          nodes.push_back(new_node); // add to end of nodes list
37      }
38
```

Add Edge between Vertices

The time complexity of my connect nodes function is O(V). The function takes in a string that represents the nodes. It must search for the nodes in the nodes vector. This could cause it to iterate through the entire vector.

```
39      // adds an edge between two GraphNodes
40      void connect_nodes(string source_name, string dest_name, int weight) {
41          GraphNode *source_node = nullptr;
42          GraphNode *dest_node = nullptr;
43
44          // find source and destination node addresses
45          for(auto current : nodes) {
46              if(source_name == current->get_name()) {
47                  source_node = current;
48              }
49
50              if(dest_name == current->get_name()) {
51                  dest_node = current;
52              }
53          }
54
55          // connect source to destination (undirected graph)
56          // this method creates two edge objects. So... not very efficient
57          if (source_node != nullptr && dest_node != nullptr) {
58              source_node->add_edge(dest_node, weight);
59              dest_node->add_edge(source_node, weight);
60          }
61      }
```

References

1. Jain, Sandeep. “Time and Space Complexity of Dijkstra’s Algorithm.” *GeeksforGeeks*, 9 February 2024, https://www.geeksforgeeks.org/time-and-space-complexity-of-dijkstras-algorithm/ Accessed 10 June 2024.
2. Mehta, Divyanshu, and Sandeep Jain. “Prim’s Algorithm for Minimum Spanning Tree (MST).” *GeeksforGeeks*, 16 February 2024, https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/ Accessed 10 June 2024.
3. Jain, Sandeep. “Vector in C++ STL.” *GeeksforGeeks*, 27 March 2024, https://www.geeksforgeeks.org/vector-in-cpp-stl/ Accessed 10 June 2024.