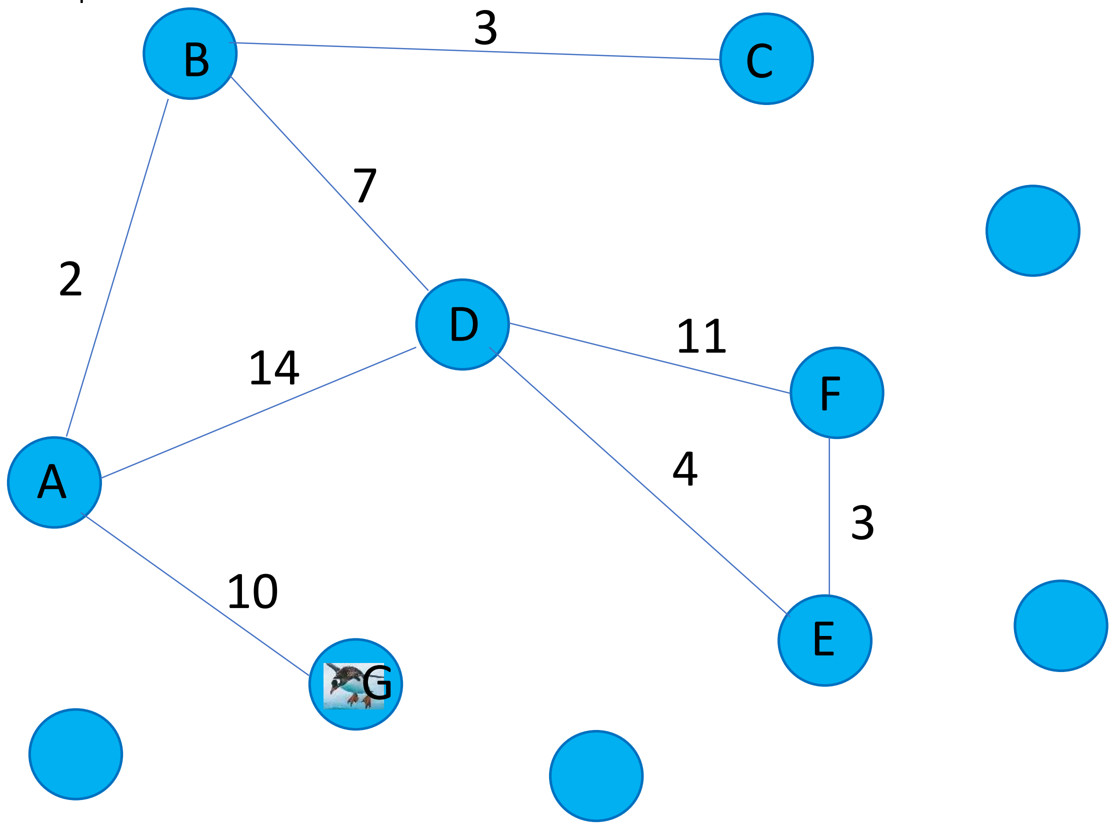
1. **(20%) Create a design** before **you start coding that describes or shows how a graph structure could be used to store some kinds of data and attempt to solve some kind of problem (yes, this can be a game that needs a graph to represent a map!)**

   Graph Structure for Penguin Routes:
   - **Vertices**: Icebergs with fun names.
   - **Edges**: Ocean paths connecting icebergs.
     - Only on edge per pair of vertices
     - Bidirectional
   - **Weight**: Length of the path (distance between icebergs).
     - Used for MST and shortest path

   Problem to Solve:
   - Add graphs and vertices as new currents and icebergs are found.
   - Find the shortest path for penguins to travel from one iceberg to another for optimal fishing spots. Finding the path with the minimum total distance between two icebergs.
     - Dijkstra's
     - https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
     - Wikipedia: Dijkstra's algorithm
   - Determine the minimum spanning tree to establish efficient routes for penguins to explore new fishing spots while covering all icebergs.
     - Kruskal's
     - https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-using-stl-in-c/
     - https://www.geeksforgeeks.org/union-by-rank-and-path-compression-in-union-find-algorithm/

Example of set up:

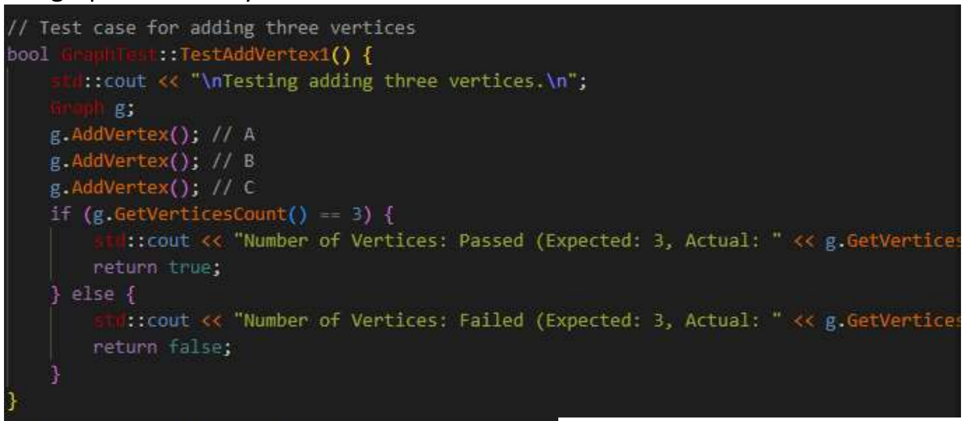


2. (20%) Create some tests (at least *two* for each piece of functionality) before **you start coding...**

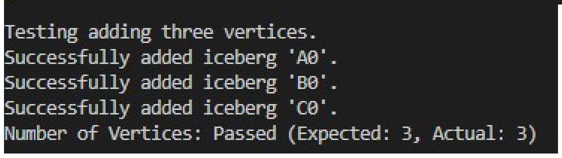
Was running into issues with my minimum spanning tree, so I used ChatGPT to help me with tests for it. Which then also helped fix my problem in the functions themselves. Other issues as well.

AddVertex:

- TestAddVertex1: Tests the addition of three vertices and verifies if the number of vertices in the graph is correctly incremented to 3.

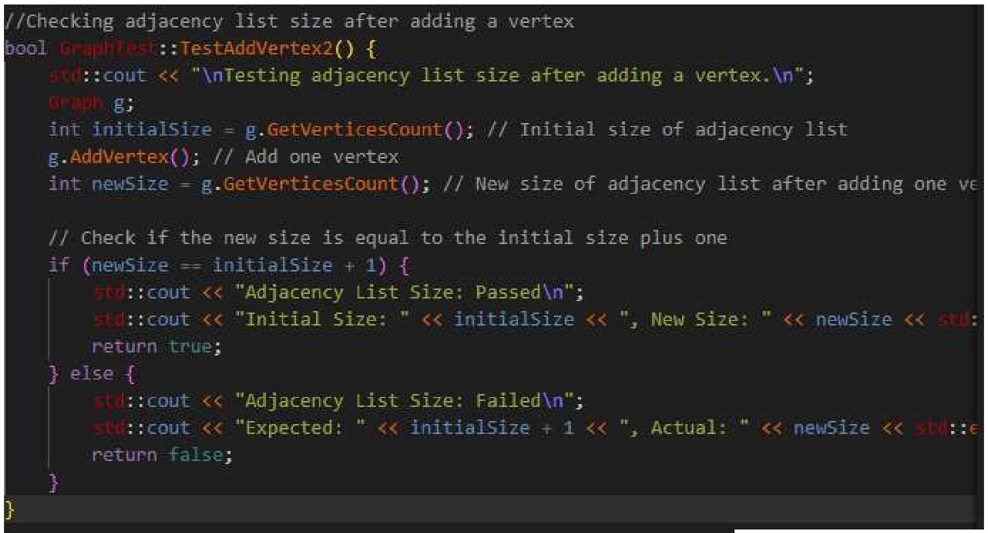


```
// Test case for adding three vertices
bool GraphTest::TestAddVertex1() {
    std::cout << "\nTesting adding three vertices.\n";
    Graph g;
    g.AddVertex(); // A
    g.AddVertex(); // B
    g.AddVertex(); // C
    if (g.GetVerticesCount() == 3) {
        std::cout << "Number of Vertices: Passed (Expected: 3, Actual: " << g.GetVertices
        return true;
    } else {
        std::cout << "Number of Vertices: Failed (Expected: 3, Actual: " << g.GetVertices
        return false;
    }
}
```

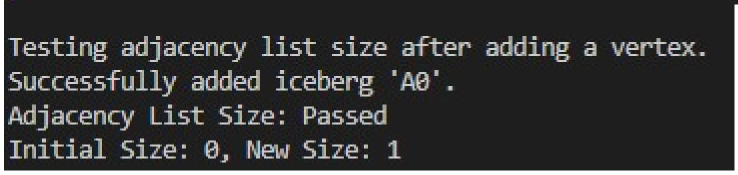


```
Testing adding three vertices.
Successfully added iceberg 'A0'.
Successfully added iceberg 'B0'.
Successfully added iceberg 'C0'.
Number of Vertices: Passed (Expected: 3, Actual: 3)
```

- TestAddVertex2: Verifies if the adjacency list size increases



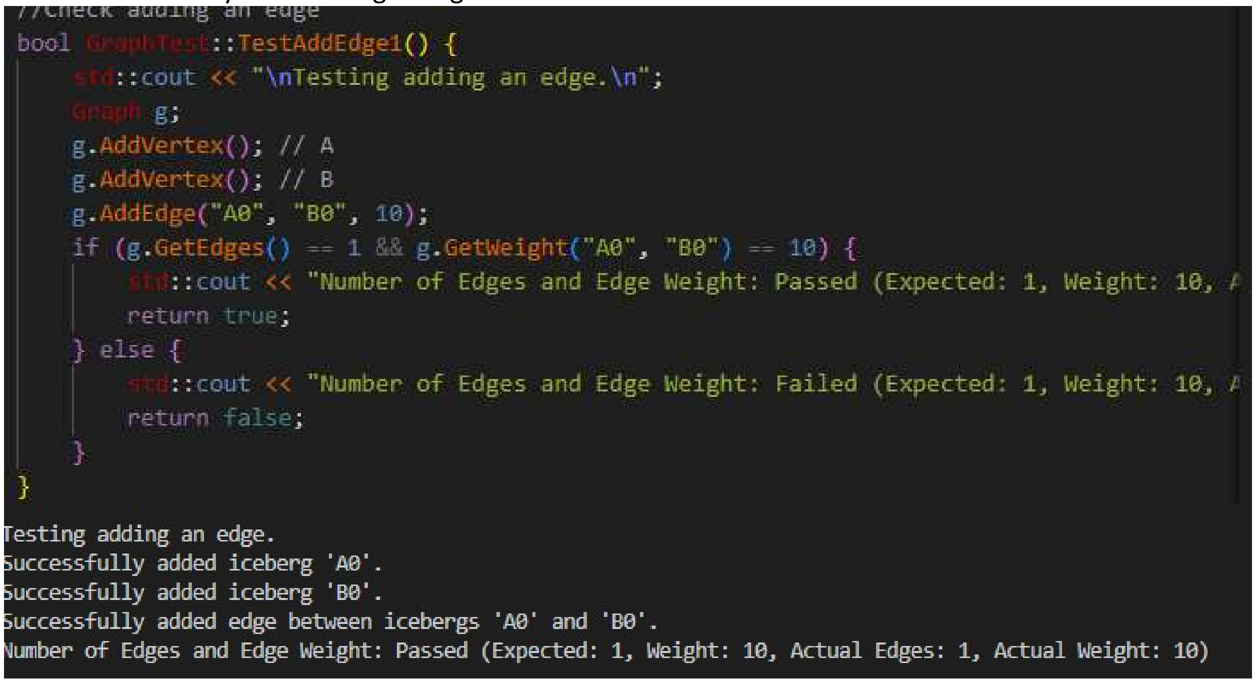
```
//Checking adjacency list size after adding a vertex
bool GraphTest::TestAddVertex2() {
    std::cout << "\nTesting adjacency list size after adding a vertex.\n";
    Graph g;
    int initialSize = g.GetVerticesCount(); // Initial size of adjacency list
    g.AddVertex(); // Add one vertex
    int newSize = g.GetVerticesCount(); // New size of adjacency list after adding one ve

    // Check if the new size is equal to the initial size plus one
    if (newSize == initialSize + 1) {
        std::cout << "Adjacency List Size: Passed\n";
        std::cout << "Initial Size: " << initialSize << ", New Size: " << newSize << std:
        return true;
    } else {
        std::cout << "Adjacency List Size: Failed\n";
        std::cout << "Expected: " << initialSize + 1 << ", Actual: " << newSize << std::e
        return false;
    }
}
```



```
Testing adjacency list size after adding a vertex.
Successfully added iceberg 'A0'.
Adjacency List Size: Passed
Initial Size: 0, New Size: 1
```

AddEdge:

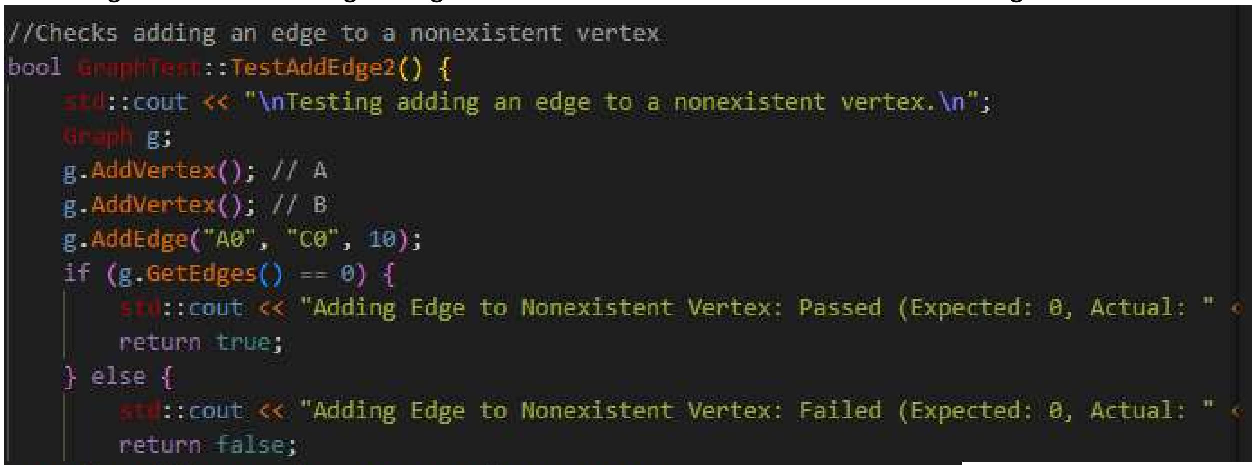
- TestAddEdge1: Ensures that adding an edge between two existing vertices updates the edge count and correctly sets the edge weight.



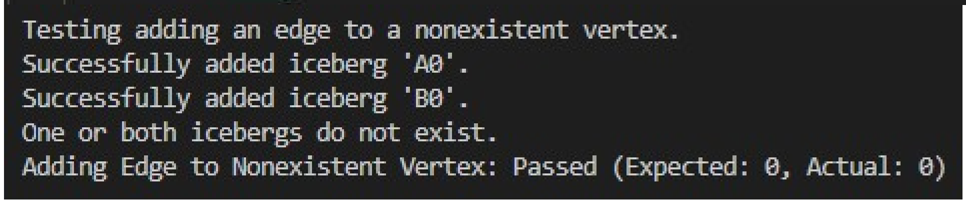
```
//Check adding an edge
bool GraphTest::TestAddEdge1() {
    std::cout << "\nTesting adding an edge.\n";
    Graph g;
    g.AddVertex(); // A
    g.AddVertex(); // B
    g.AddEdge("A0", "B0", 10);
    if (g.GetEdges() == 1 && g.GetWeight("A0", "B0") == 10) {
        std::cout << "Number of Edges and Edge Weight: Passed (Expected: 1, Weight: 10, A
        return true;
    } else {
        std::cout << "Number of Edges and Edge Weight: Failed (Expected: 1, Weight: 10, A
        return false;
    }
}
```

```
Testing adding an edge.
Successfully added iceberg 'A0'.
Successfully added iceberg 'B0'.
Successfully added edge between icebergs 'A0' and 'B0'.
Number of Edges and Edge Weight: Passed (Expected: 1, Weight: 10, Actual Edges: 1, Actual Weight: 10)
```

- TestAddEdge2: Checks if adding an edge to a nonexistent vertex does not affect the edge count.



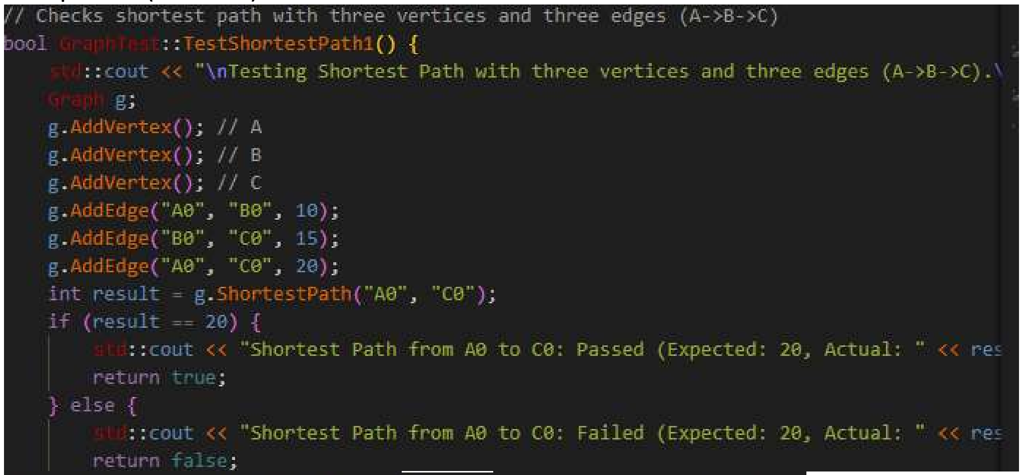
```
//Checks adding an edge to a nonexistent vertex
bool GraphTest::TestAddEdge2() {
    std::cout << "\nTesting adding an edge to a nonexistent vertex.\n";
    Graph g;
    g.AddVertex(); // A
    g.AddVertex(); // B
    g.AddEdge("A0", "C0", 10);
    if (g.GetEdges() == 0) {
        std::cout << "Adding Edge to Nonexistent Vertex: Passed (Expected: 0, Actual: " <
        return true;
    } else {
        std::cout << "Adding Edge to Nonexistent Vertex: Failed (Expected: 0, Actual: " <
        return false;
```



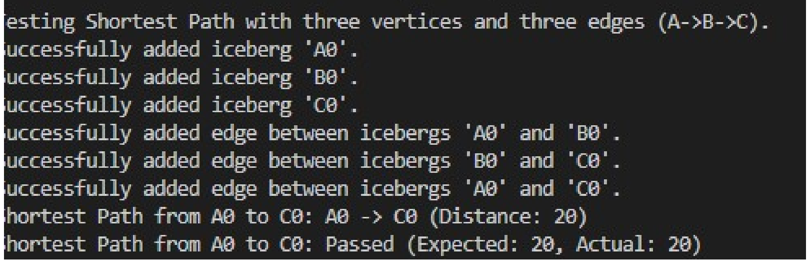
```
Testing adding an edge to a nonexistent vertex.
Successfully added iceberg 'A0'.
Successfully added iceberg 'B0'.
One or both icebergs do not exist.
Adding Edge to Nonexistent Vertex: Passed (Expected: 0, Actual: 0)
```

ShortestPath:

- TestShortestPath1: Validates the shortest path calculation between three vertices connected in a sequence (A->B->C).

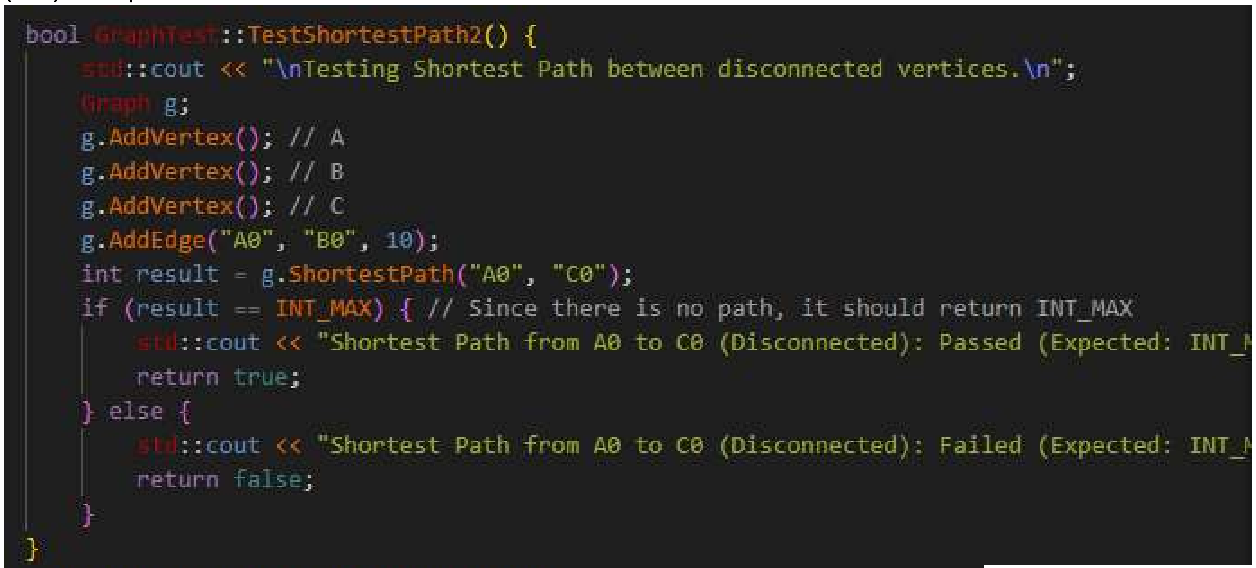


```
/ Checks shortest path with three vertices and three edges (A->B->C)
bool GraphTest::TestShortestPath1() {
    std::cout << "\nTesting Shortest Path with three vertices and three edges (A->B->C).\
    Graph g;
    g.AddVertex(); // A
    g.AddVertex(); // B
    g.AddVertex(); // C
    g.AddEdge("A0", "B0", 10);
    g.AddEdge("B0", "C0", 15);
    g.AddEdge("A0", "C0", 20);
    int result = g.ShortestPath("A0", "C0");
    if (result == 20) {
        std::cout << "Shortest Path from A0 to C0: Passed (Expected: 20, Actual: " << res
        return true;
    } else {
        std::cout << "Shortest Path from A0 to C0: Failed (Expected: 20, Actual: " << res
        return false;
```

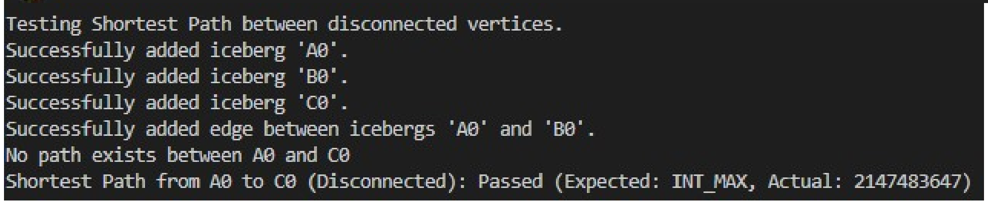


```
esting Shortest Path with three vertices and three edges (A->B->C).
uccessfully added iceberg 'A0'.
uccessfully added iceberg 'B0'.
uccessfully added iceberg 'C0'.
uccessfully added edge between icebergs 'A0' and 'B0'.
uccessfully added edge between icebergs 'B0' and 'C0'.
uccessfully added edge between icebergs 'A0' and 'C0'.
hortest Path from A0 to C0: A0 -> C0 (Distance: 20)
hortest Path from A0 to C0: Passed (Expected: 20, Actual: 20)
```

- TestShortestPath2: Checks if the shortest path between disconnected vertices returns infinity (INF) as expected.



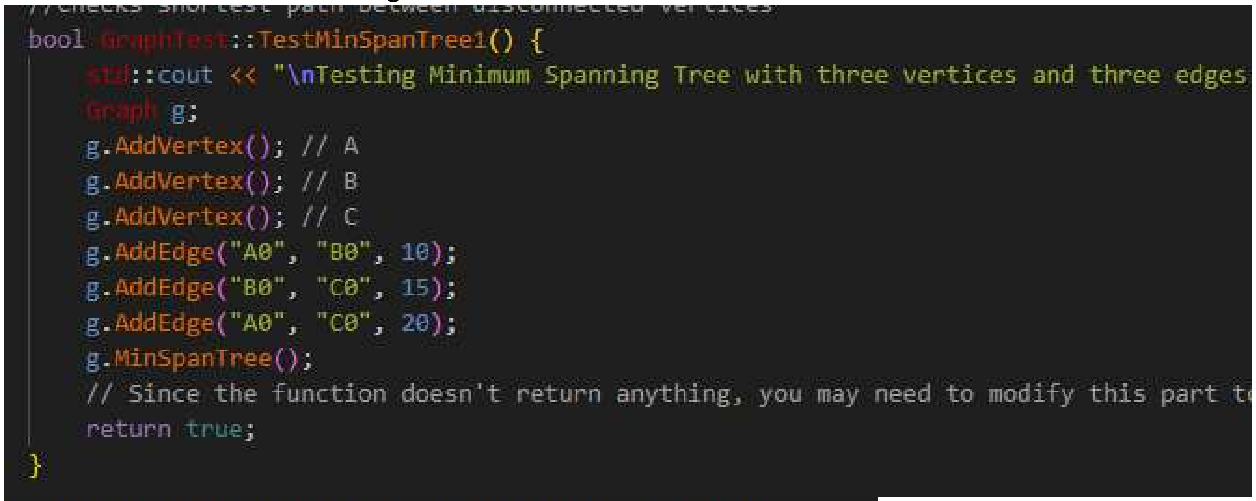
```
bool GraphTest::TestShortestPath2() {
    std::cout << "\nTesting Shortest Path between disconnected vertices.\n";
    Graph g;
    g.AddVertex(); // A
    g.AddVertex(); // B
    g.AddVertex(); // C
    g.AddEdge("A0", "B0", 10);
    int result = g.ShortestPath("A0", "C0");
    if (result == INT_MAX) { // Since there is no path, it should return INT_MAX
        std::cout << "Shortest Path from A0 to C0 (Disconnected): Passed (Expected: INT_M
        return true;
    } else {
        std::cout << "Shortest Path from A0 to C0 (Disconnected): Failed (Expected: INT_M
        return false;
    }
}
```



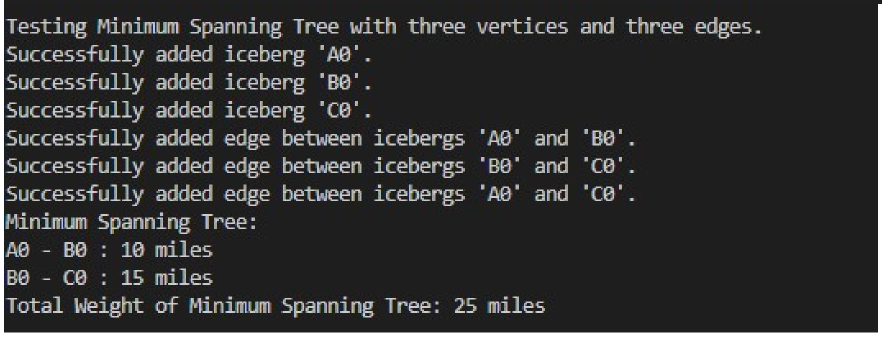
```
Testing Shortest Path between disconnected vertices.
Successfully added iceberg 'A0'.
Successfully added iceberg 'B0'.
Successfully added iceberg 'C0'.
Successfully added edge between icebergs 'A0' and 'B0'.
No path exists between A0 and C0
Shortest Path from A0 to C0 (Disconnected): Passed (Expected: INT_MAX, Actual: 2147483647)
```

MinSpanTree:

- TestMinSpanTree1: Tests the calculation of the minimum spanning tree (MST) weight in a graph with three vertices and three edges.

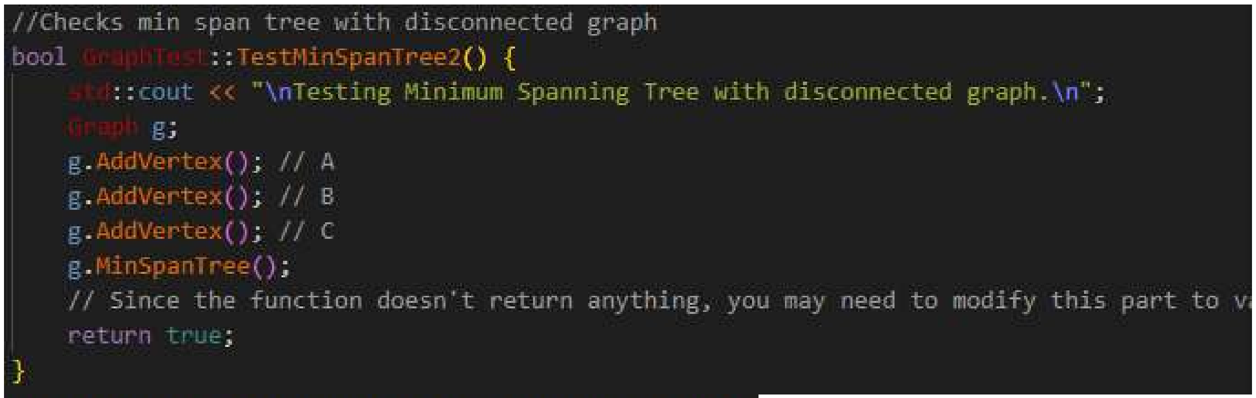


```
bool GraphTest::TestMinSpanTree1() {
    std::cout << "\nTesting Minimum Spanning Tree with three vertices and three edges
    Graph g;
    g.AddVertex(); // A
    g.AddVertex(); // B
    g.AddVertex(); // C
    g.AddEdge("A0", "B0", 10);
    g.AddEdge("B0", "C0", 15);
    g.AddEdge("A0", "C0", 20);
    g.MinSpanTree();
    // Since the function doesn't return anything, you may need to modify this part t
    return true;
}
```

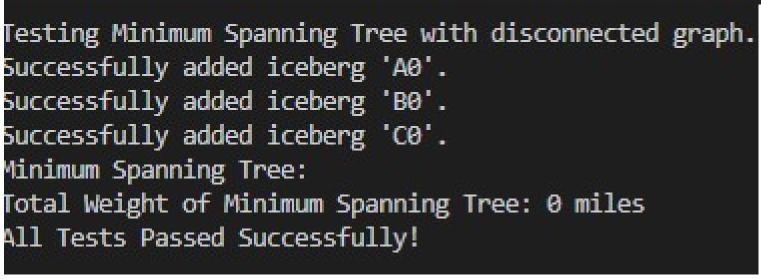


```
Testing Minimum Spanning Tree with three vertices and three edges.
Successfully added iceberg 'A0'.
Successfully added iceberg 'B0'.
Successfully added iceberg 'C0'.
Successfully added edge between icebergs 'A0' and 'B0'.
Successfully added edge between icebergs 'B0' and 'C0'.
Successfully added edge between icebergs 'A0' and 'C0'.
Minimum Spanning Tree:
A0 - B0 : 10 miles
B0 - C0 : 15 miles
Total Weight of Minimum Spanning Tree: 25 miles
```

- TestMinSpanTree2: Verifies that the MST weight of a disconnected graph is correctly calculated as 0.



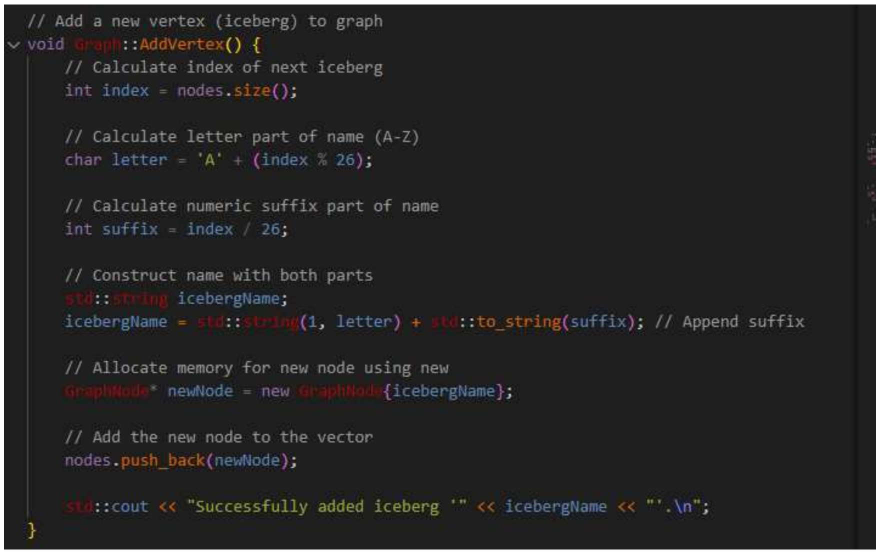
```
//Checks min span tree with disconnected graph
bool GraphTest::TestMinSpanTree2() {
    std::cout << "\nTesting Minimum Spanning Tree with disconnected graph.\n";
    Graph g;
    g.AddVertex(); // A
    g.AddVertex(); // B
    g.AddVertex(); // C
    g.MinSpanTree();
    // Since the function doesn't return anything, you may need to modify this part to v
    return true;
}
```



```
Testing Minimum Spanning Tree with disconnected graph.
Successfully added iceberg 'A0'.
Successfully added iceberg 'B0'.
Successfully added iceberg 'C0'.
Minimum Spanning Tree:
Total Weight of Minimum Spanning Tree: 0 miles
All Tests Passed Successfully!
```

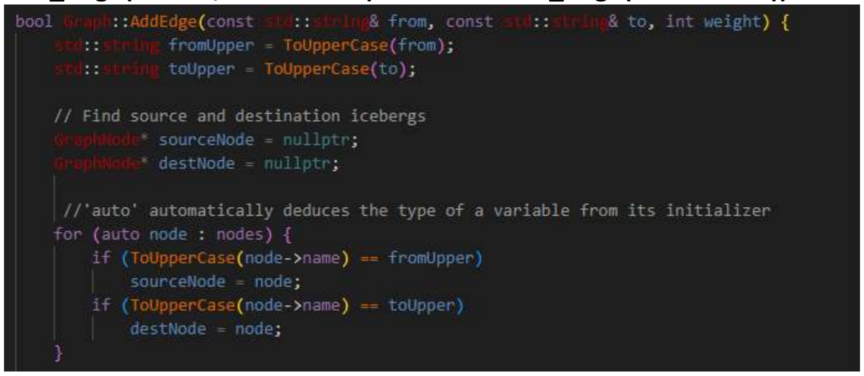
3. **(40%) Implement a graph class with at least (this category effectively combines implementation and specification, partly to emphasize getting the algorithms working!):**

    1. **(5%) a function to add a new vertex to the graph (perhaps add_vertex(vertex_name))**



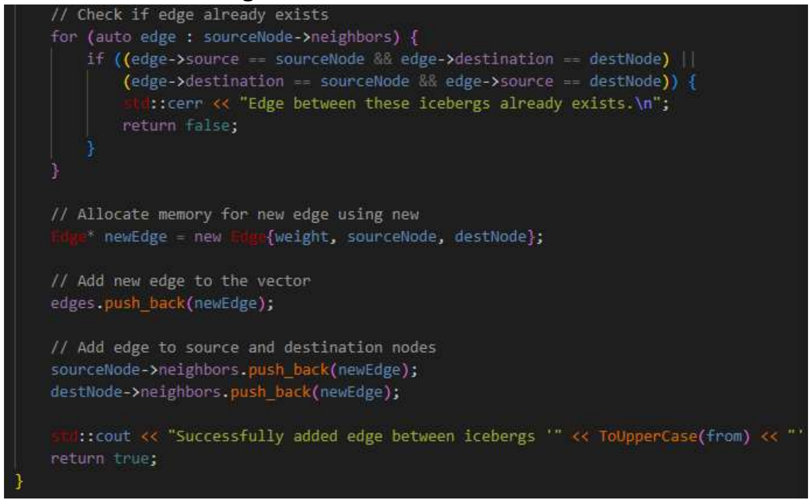
```cpp
// Add a new vertex (iceberg) to graph
void Graph::AddVertex() {
    // Calculate index of next iceberg
    int index = nodes.size();

    // Calculate letter part of name (A-Z)
    char letter = 'A' + (index % 26);

    // Calculate numeric suffix part of name
    int suffix = index / 26;

    // Construct name with both parts
    std::string icebergName;
    icebergName = std::string(1, letter) + std::to_string(suffix); // Append suffix

    // Allocate memory for new node using new
    GraphNode* newNode = new GraphNode{icebergName};

    // Add the new node to the vector
    nodes.push_back(newNode);

    std::cout << "Successfully added iceberg '" << icebergName << "'.\n";
}
```

    2. **(5%) a function to add a new edge between two vertices of the graph (perhaps add_edge(source, destination) or source.add_edge(destination))**



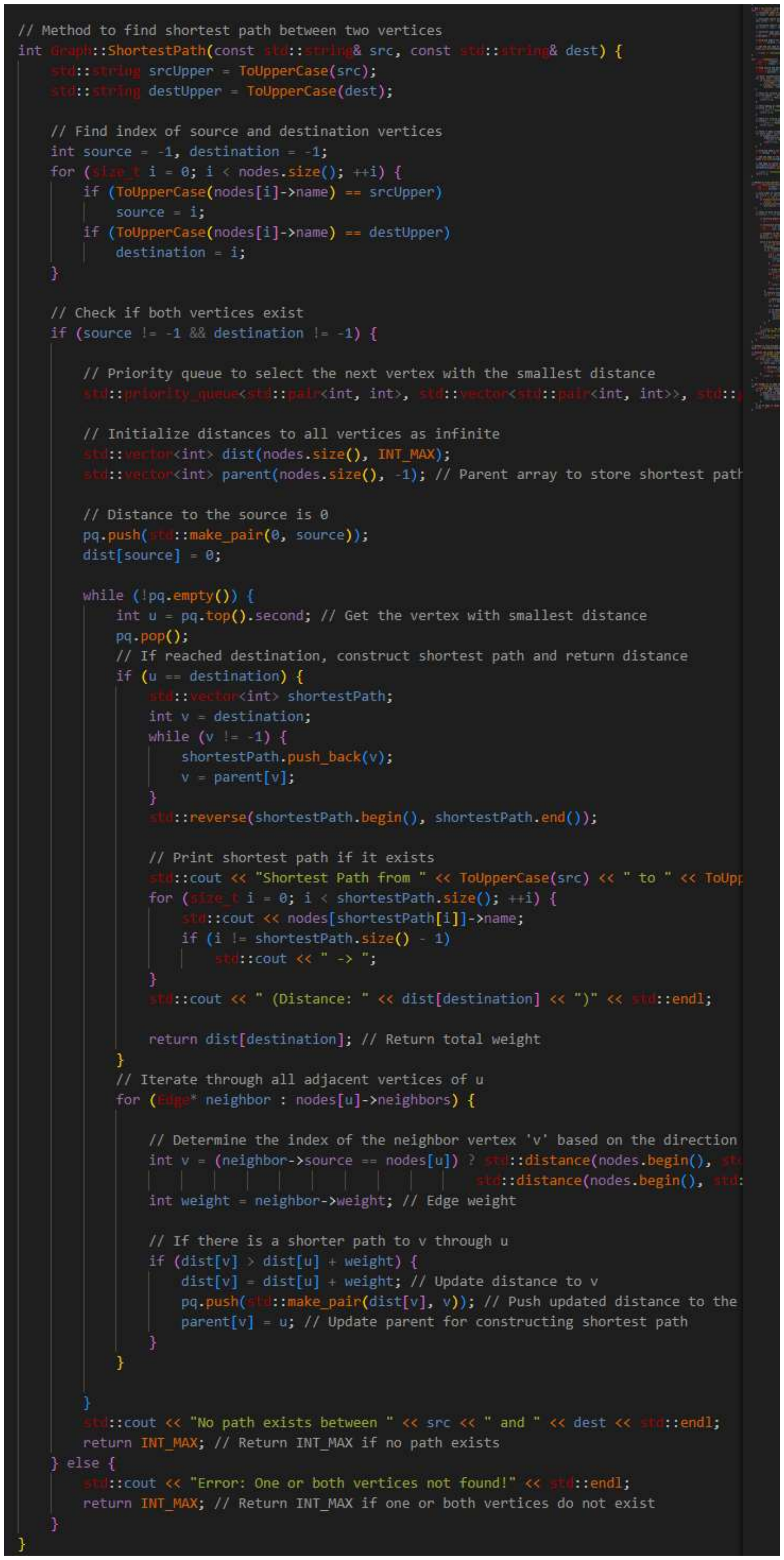
```cpp
bool Graph::AddEdge(const std::string& from, const std::string& to, int weight) {
    std::string fromUpper = ToUpperCase(from);
    std::string toUpper = ToUpperCase(to);

    // Find source and destination icebergs
    GraphNode* sourceNode = nullptr;
    GraphNode* destNode = nullptr;

    //'auto' automatically deduces the type of a variable from its initializer
    for (auto node : nodes) {
        if (ToUpperCase(node->name) == fromUpper)
            sourceNode = node;
        if (ToUpperCase(node->name) == toUpper)
            destNode = node;
    }
```

lots of error handling between these



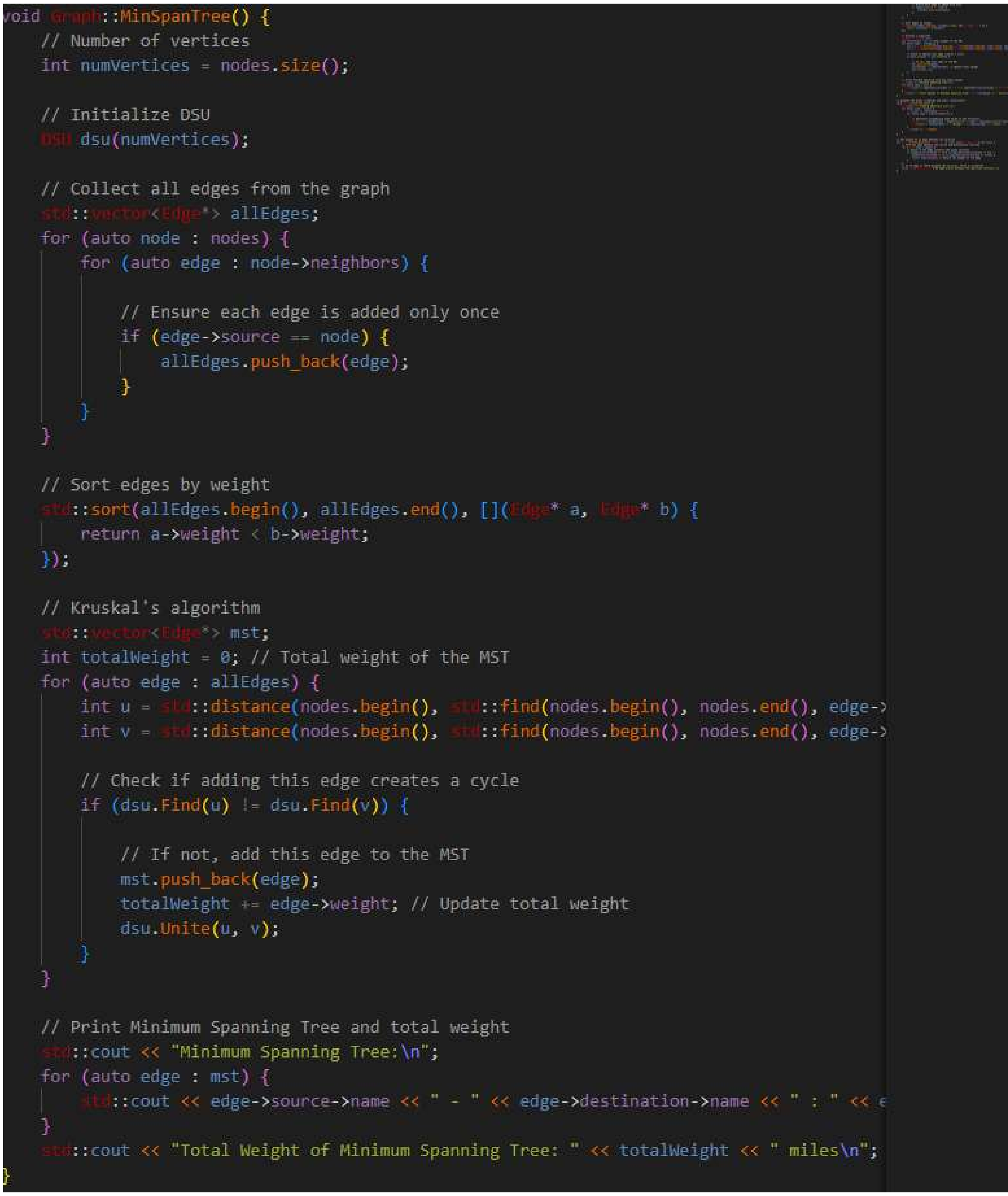
```cpp
    // Check if edge already exists
    for (auto edge : sourceNode->neighbors) {
        if ((edge->source == sourceNode && edge->destination == destNode) ||
            (edge->destination == sourceNode && edge->source == destNode)) {
            std::cerr << "Edge between these icebergs already exists.\n";
            return false;
        }
    }

    // Allocate memory for new edge using new
    Edge* newEdge = new Edge{weight, sourceNode, destNode};

    // Add new edge to the vector
    edges.push_back(newEdge);

    // Add edge to source and destination nodes
    sourceNode->neighbors.push_back(newEdge);
    destNode->neighbors.push_back(newEdge);

    std::cout << "Successfully added edge between icebergs '" << ToUpperCase(from) << "'
    return true;
}
```

3. (15%) a function for a shortest path algorithm (perhaps shortest_path(source, destination))



```cpp
// Method to find shortest path between two vertices
int Graph::ShortestPath(const std::string& src, const std::string& dest) {
    std::string srcUpper = ToUpperCase(src);
    std::string destUpper = ToUpperCase(dest);

    // Find index of source and destination vertices
    int source = -1, destination = -1;
    for (size_t i = 0; i < nodes.size(); ++i) {
        if (ToUpperCase(nodes[i]->name) == srcUpper)
            source = i;
        if (ToUpperCase(nodes[i]->name) == destUpper)
            destination = i;
    }

    // Check if both vertices exist
    if (source != -1 && destination != -1) {

        // Priority queue to select the next vertex with the smallest distance
        std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, std:::

        // Initialize distances to all vertices as infinite
        std::vector<int> dist(nodes.size(), INT_MAX);
        std::vector<int> parent(nodes.size(), -1); // Parent array to store shortest path

        // Distance to the source is 0
        pq.push(std::make_pair(0, source));
        dist[source] = 0;

        while (!pq.empty()) {
            int u = pq.top().second; // Get the vertex with smallest distance
            pq.pop();
            // If reached destination, construct shortest path and return distance
            if (u == destination) {
                std::vector<int> shortestPath;
                int v = destination;
                while (v != -1) {
                    shortestPath.push_back(v);
                    v = parent[v];
                }
                std::reverse(shortestPath.begin(), shortestPath.end());

                // Print shortest path if it exists
                std::cout << "Shortest Path from " << ToUpperCase(src) << " to " << ToUpp
                for (size_t i = 0; i < shortestPath.size(); ++i) {
                    std::cout << nodes[shortestPath[i]]->name;
                    if (i != shortestPath.size() - 1)
                        std::cout << " -> ";
                }
                std::cout << " (Distance: " << dist[destination] << ")" << std::endl;

                return dist[destination]; // Return total weight
            }
            // Iterate through all adjacent vertices of u
            for (Edge* neighbor : nodes[u]->neighbors) {

                // Determine the index of the neighbor vertex 'v' based on the direction
                int v = (neighbor->source == nodes[u]) ? std::distance(nodes.begin(), st
                                                         std::distance(nodes.begin(), std:
                int weight = neighbor->weight; // Edge weight

                // If there is a shorter path to v through u
                if (dist[v] > dist[u] + weight) {
                    dist[v] = dist[u] + weight; // Update distance to v
                    pq.push(std::make_pair(dist[v], v)); // Push updated distance to the
                    parent[v] = u; // Update parent for constructing shortest path
                }
            }

        }
        std::cout << "No path exists between " << src << " and " << dest << std::endl;
        return INT_MAX; // Return INT_MAX if no path exists
    } else {
        std::cout << "Error: One or both vertices not found!" << std::endl;
        return INT_MAX; // Return INT_MAX if one or both vertices do not exist
    }
}
```

4. **(15%) a function for a minimum spanning tree algorithm (example min_span_tree()).**



```
void Graph::MinSpanTree() {
    // Number of vertices
    int numVertices = nodes.size();

    // Initialize DSU
    DSU dsu(numVertices);

    // Collect all edges from the graph
    std::vector<Edge*> allEdges;
    for (auto node : nodes) {
        for (auto edge : node->neighbors) {

            // Ensure each edge is added only once
            if (edge->source == node) {
                allEdges.push_back(edge);
            }
        }
    }

    // Sort edges by weight
    std::sort(allEdges.begin(), allEdges.end(), [](Edge* a, Edge* b) {
        return a->weight < b->weight;
    });

    // Kruskal's algorithm
    std::vector<Edge*> mst;
    int totalWeight = 0; // Total weight of the MST
    for (auto edge : allEdges) {
        int u = std::distance(nodes.begin(), std::find(nodes.begin(), nodes.end(), edge->
        int v = std::distance(nodes.begin(), std::find(nodes.begin(), nodes.end(), edge->

        // Check if adding this edge creates a cycle
        if (dsu.Find(u) != dsu.Find(v)) {

            // If not, add this edge to the MST
            mst.push_back(edge);
            totalWeight += edge->weight; // Update total weight
            dsu.Unite(u, v);
        }
    }

    // Print Minimum Spanning Tree and total weight
    std::cout << "Minimum Spanning Tree:\n";
    for (auto edge : mst) {
        std::cout << edge->source->name << " - " << edge->destination->name << " : " << e
    }
    std::cout << "Total Weight of Minimum Spanning Tree: " << totalWeight << " miles\n";
}
```

This also uses the disjointed union method. The DSU is a class.

4. **(10%) Analyze the complexity of all of your graph behaviors (effectively a part of our documentation for grading purposes)**

   So, in my first run, I realized that I made something much too complex for what it needs to be because I was using a matrix, which made it an O(V^2). I wanted to avoid this, so I changed it. I've also found some much better examples as I dig though this, but I'm sticking with it.
   sources:
   https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

   https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/

   https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

Behavior	Time Complexity	Space Complexity	Description
Add Vertex	O(1)	O(V)	Adds a new vertex to the graph. Time complexity is constant as it involves simple memory allocation and assignment operations. Space complexity increases linearly with the number of vertices.
Add Edge	O(1)	O(1)	Adds a new edge between two vertices with a specified weight. Time and space complexities are constant since it involves direct manipulation of adjacency lists and edge data structures.
Shortest Path	O(E log V)	O(V)	Finds the shortest path from a source vertex to all other vertices using Dijkstra's algorithm. The time complexity is O(E log V) using a binary heap for priority queue implementation. The space complexity is O(V) for storing distances and maintaining priority queues.
Minimum Spanning Tree	O(E log V)	O(V)	Computes the Minimum Spanning Tree (MST) of the graph using Kruskal's algorithm. Time complexity is O(E log V) due to sorting of edges and disjoint set operations. Space complexity is O(V) for storing disjoint sets.

5. **(10%) Once you have implemented and tested your code, add to the README file what line(s) of code or inputs and outputs show your work meeting each of the above requirements (or better, include a small screen snip of where it meets the requirement!).**

   Sources: There were soooooo many https://www.geeksforgeeks.org/ pages I went through for this. I think I got them all? But each little section kind of gave me new ideas. I also use ChatGBT to check my er. I would throw it in there to see what was wrong with it. This is especially true for Dijkstra's algorithm. I think that I understand everything at this point.

Done!