<u>Requirements</u>

1) In the folder “designs”, I have 4 designs. The image “problem.jpg” lays out the problem I am attempting to solve. Outside of the folder there is an image called “vvardenfellmap.png”, which is the picture of the map/graph from the video game. The image “GraphStructue.jpg” is the design behind how the code files work together to represent a graph. “ShortestPathDesign.pdf” is my design for the shortest path function. “MinSpanningTreeDesign.pdf” is my design for the minimum spanning tree function.

2) I have at least two tests for each piece of functionality.

add_node

```
cout << "\n------Adding nodes: A, B, C, D, E, F------" << endl;
alphabet_graph.add_node("A");
alphabet_graph.add_node("B");
alphabet_graph.add_node("C");
alphabet_graph.add_node("D");
alphabet_graph.add_node("E");
alphabet_graph.add_node("F");

cout << "\n------Testing that the nodes have been added to alphabet_graph------" << endl;
cout << "alphabet_graph.get_size(): " << alphabet_graph.get_size() << endl;
cout << "alphabet_graph.get_nodes().at(0)->get_name(): " << alphabet_graph.get_nodes().at(0)->get_name() << endl;
cout << "alphabet_graph.get_nodes().at(5)->get_name(): " << alphabet_graph.get_nodes().at(5)->get_name() << endl;
```

```
cout << "\n------Adding nodes: A, B, C, D, E------" << endl;
alphabet_graph_2.add_node("A");
alphabet_graph_2.add_node("B");
alphabet_graph_2.add_node("C");
alphabet_graph_2.add_node("D");
alphabet_graph_2.add_node("E");

cout << "\n------Testing that the nodes have been added to alphabet_graph_2------" << endl;
cout << "alphabet_graph_2.get_size(): " << alphabet_graph_2.get_size() << endl;
cout << "alphabet_graph_2.get_nodes().at(0)->get_name(): " << alphabet_graph_2.get_nodes().at(0)->get_name() << endl;
cout << "alphabet_graph_2.get_nodes().at(3)->get_name(): " << alphabet_graph_2.get_nodes().at(3)->get_name() << endl;
```

connect_nodes

```
cout << "\n------Connecting nodes------" << endl;
alphabet_graph_2.connect_nodes("A", "B", 10);
alphabet_graph_2.connect_nodes("A", "C", 5);
alphabet_graph_2.connect_nodes("A", "D", 3);
alphabet_graph_2.connect_nodes("B", "C", 4);
alphabet_graph_2.connect_nodes("B", "D", 1);
alphabet_graph_2.connect_nodes("C", "E", 1);
alphabet_graph_2.connect_nodes("D", "E", 2);

cout << "\n------Testing that the nodes have been connected. Expected: source=D ; destination=B ; weight=1------" << endl;
cout << "alphabet_graph_2.get_nodes().at(3)->get_neighbors().at(1)->source->get_name(): " << alphabet_graph_2.get_nodes().at(3)->get_neighbors().at(1)->source->get_name() << endl;
cout << "alphabet_graph_2.get_nodes().at(3)->get_neighbors().at(1)->destination->get_name(): " << alphabet_graph_2.get_nodes().at(3)->get_neighbors().at(1)->destination->get_name() << endl;
cout << "alphabet_graph_2.get_nodes().at(3)->get_neighbors().at(1)->weight: " << alphabet_graph_2.get_nodes().at(3)->get_neighbors().at(1)->weight << endl;
```

```
cout << "\n------Connecting nodes------" << endl;
alphabet_graph.connect_nodes("A", "C", 3);
alphabet_graph.connect_nodes("A", "D", 1);
alphabet_graph.connect_nodes("B", "C", 2);
alphabet_graph.connect_nodes("B", "D", 2);
alphabet_graph.connect_nodes("B", "E", 2);
alphabet_graph.connect_nodes("C", "D", 1);
alphabet_graph.connect_nodes("C", "E", 6);
alphabet_graph.connect_nodes("D", "E", 5);

cout << "\n------Testing that the nodes have been connected. Expected: source=A ; destination=C ; weight=3------" << endl;
cout << "alphabet_graph.get_nodes().at(0)->get_neighbors().at(0)->source->get_name(): " << alphabet_graph.get_nodes().at(0)->get_neighbors().at(0)->source->get_name() << endl;
cout << "alphabet_graph.get_nodes().at(0)->get_neighbors().at(0)->destination->get_name(): " << alphabet_graph.get_nodes().at(0)->get_neighbors().at(0)->destination->get_name() << endl;
cout << "alphabet_graph.get_nodes().at(0)->get_neighbors().at(0)->weight: " << alphabet_graph.get_nodes().at(0)->get_neighbors().at(0)->weight << endl;
```

I test shortest_path and min_spanning_tree on two different graphs.

3) In the folder “code_files” there is a file “graph.h” which defines the Graph class.

3.1) In the Graph class there is a function called add_node. The function takes in a string, makes that input the name of a new node, and adds that node to a list of nodes.

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

3.2) In the Graph class there is a function called connect_nodes. This function adds two edges between nodes. It ensures that the graph is not directed.

```
    // adds an edge between two GraphNodes
    void connect_nodes(string source_name, string dest_name, int weight) {
        GraphNode *source_node = nullptr;
        GraphNode *dest_node = nullptr;

        // find source and destination node addresses
        for(auto current : nodes) {
            if(source_name == current->get_name()) {
                source_node = current;
            }

            if(dest_name == current->get_name()) {
                dest_node = current;
            }
        }

        // connect source to destination (undirected graph)
        // this method creates two edge objects. So... not very efficient
        if (source_node != nullptr && dest_node != nullptr) {
            source_node->add_edge(dest_node, weight);
            dest_node->add_edge(source_node, weight);
        }
    }
```

3.3) In the Graph class there is a function called shortest_path. It takes in a string that should be the name of a node in the graph. It then finds the shortest path and distances from the node to all other nodes. It will return a string of that information.

```cpp
    // shortest path (implements Dijkstra's algorithm) returns a string that represents th
    string shortest_path(string source_name) {
        GraphNode *source_node = nullptr;

        for(auto node : nodes) {
            if(source_name == node->get_name()) { // finds source node's addresses
                source_node = node;
            }
            node->set_distance(numeric_limits<int>::max()); // sets all node's distances t
            node->set_previous(nullptr); // sets all node's previous's to nullptr
        }

        // set source node's distance to 0
        source_node->set_distance(0);

        // queue of unvisited nodes
        vector<GraphNode *> queue = nodes;

        // finds the shortest distance from the source node to all other nodes
        GraphNode *current = source_node;
        while (queue.size() != 0) { // if all nodes have been visited the algorithm shoul

            for (auto edge : current->get_neighbors()) { // iterates through "current's" n
                GraphNode *dest = edge->destination; // create destination node
                if (dest->get_distance() > current->get_distance() + edge->weight) { // i
                    dest->set_distance(current->get_distance() + edge->weight); // update
                    dest->set_previous(current); // update destination's previous
                }
            }

            int count = 0;
            int current_index = 0;
            for (auto node : queue) {
                if (node == current) { // finds index of the current
```

3.4) In the Graph class there is a function called min_spanning_tree. It will return the minimum spanning tree as a string.

```cpp
    // minimum spanning tree (implements Prim's algorithm) returns a string that repr
    string min_spanning_tree() {
        GraphNode *source_node = nodes[0]; // source node is the first node added to
        vector<GraphNode *> visited; // list of visited nodes
        visited.push_back(source_node);
        vector<edge *> tree_edges; // list of edges in the minimum spanning tree
        GraphNode *current = source_node;

        int floating_nodes = 0;
        for (auto node : nodes) { // checking if there are any nodes not connected to
            int neighbor_size = node->get_neighbors().size();
            if (neighbor_size < 1) {
                floating_nodes += 1;
            }
        }

        // Prim's algorithm
        while (visited.size() < nodes.size() - floating_nodes) { // the algorithm sho
            edge *lightest_edge_1 = new edge(); // lightest edge 1
            edge *lightest_edge_2 = new edge(); // lightest edge 2
            if (visited.size() > 1) {
                for (auto visited_node : visited) { // iterates through the visited n
                    for (auto neighbor : visited_node->get_neighbors()) { // iterates
                        bool contains = false;
                        for (auto tree_edge : tree_edges) { // checks if neighbor is
                            if (tree_edge == neighbor) {
                                contains = true;
                            }
                        }
                        for (auto node : visited) { // checks if destination has alre
                            if (node == neighbor->destination) {
                                contains = true;
                            }
                        }
```

4) In “complexity.pdf”, I analyze the complexity of the behaviors of the graph.