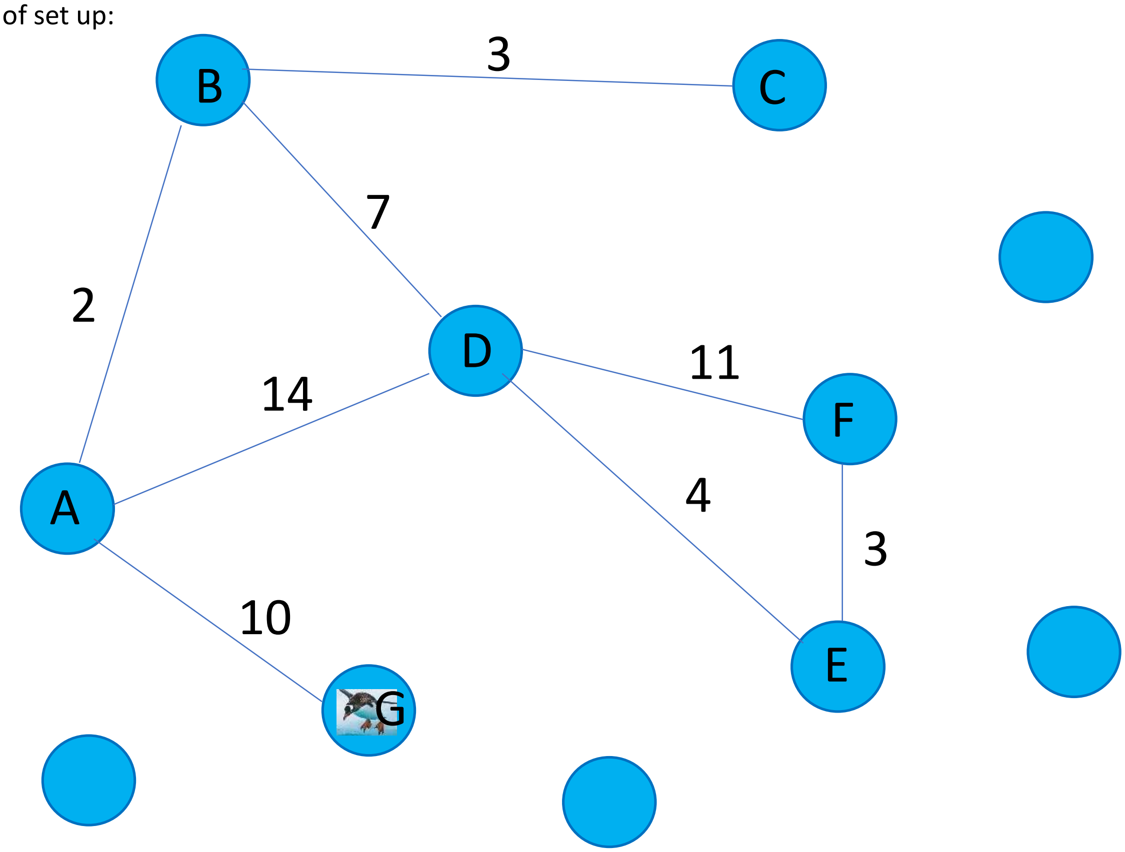
1. **(20%) Create a design** before **you start coding that describes or shows how a graph structure could be used to store some kinds of data and attempt to solve some kind of problem (yes, this can be a game that needs a graph to represent a map!)**

   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.GetVertic
        return true;
    } else {
        std::cout << "Number of Vertices: Failed (Expected: 3, Actual: " << g.GetVertic
        return false;
    }
}
```

```
Testing adding three vertices.
Number of Vertices: Passed (Expected: 3, Actual: 3)
```

- TestAddVertex2: Verifies if the adjacency matrix size increases by one row and one column after adding a vertex.

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

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

```
Testing iceberg matrix size after adding a vertex.
Iceberg Matrix 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.

```cpp
// Test case for adding an edge
bool GraphTest::TestAddEdge1() {
    std::cout << "\nTesting adding an edge.\n";
    Graph g;
    g.AddVertex(); // A
    g.AddVertex(); // B
    g.AddEdge("A", "B", 10);
    if (g.GetEdgesCount() == 1 && g.GetWeight("A", "B") == 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.
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.

```cpp
// Test case for 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("A", "C", 10);
    if (g.GetEdgesCount() == 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.
Error: One or both vertices not found!
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).

```
// Test case for 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).\n";
    Graph g;

    // Add vertices and edges
    g.AddVertex(); // A
    g.AddVertex(); // B
    g.AddVertex(); // C
    g.AddEdge("A", "B", 10);
    g.AddEdge("B", "C", 15);
    g.AddEdge("A", "C", 20);

    // Calculate shortest path
    int result = g.ShortestPath("A", "C");

    // Check the result
    if (result == 20) {
        std::cout << "Shortest Path from A to C: Passed (Expected: 20, Actual: " << result << ")\n";
        return true;
    } else {
        std::cout << "Shortest Path from A to C: Failed (Expected: 20, Actual: " << result << ")\n";
        return false;
    }
}
```

```
Testing Shortest Path with three vertices and three edges (A->B->C).
Shortest Path from A to C: A -> C (Distance: 20)
Shortest Path from A to C: Passed (Expected: 20, Actual: 20)
```

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

```
// Test case for shortest path between disconnected vertices
bool GraphTest::TestShortestPath2() {
    std::cout << "\nTesting Shortest Path between disconnected vertices.\n";
    Graph g;

    // Add vertices and edges
    g.AddVertex(); // A
    g.AddVertex(); // B
    g.AddVertex(); // C
    g.AddEdge("A", "B", 10);

    // Calculate shortest path
    int result = g.ShortestPath("A", "C");

    // Check the result
    if (result == INF) { // Since there is no path, it should return INF
        std::cout << "Shortest Path from A to C (Disconnected): Passed (Expected: INF, Actual: " << result << ")\n";
        return true;
    } else {
        std::cout << "Shortest Path from A to C (Disconnected): Failed (Expected: INF, Actual: " << result << ")\n";
        return false;
    }
}
```

```
Testing Shortest Path between disconnected vertices.
No path exists between A and C
Shortest Path from A to C (Disconnected): Passed (Expected: INF, Actual: 1061109567)
```

MinSpanTree:

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

```
// Test case for minimum spanning tree 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("A", "B", 10);
    g.AddEdge("B", "C", 15);
    g.AddEdge("A", "C", 20);
    int result = g.MinSpanTree();
    if (result == 25) { // MST weight should be 25 (10 + 15)
        std::cout << "Minimum Spanning Tree Weight: Passed (Expected: 25, Actual: " <<
        return true;
    } else {
        std::cout << "Minimum Spanning Tree Weight: Failed (Expected: 25, Actual: " <<
        return false;
    }
}
```

```
Testing Minimum Spanning Tree with three vertices and three edges.
Edge: A - B (Weight: 10)
Edge: B - C (Weight: 15)
Total weight of the Minimum Spanning Tree is 25
Minimum Spanning Tree Weight: Passed (Expected: 25, Actual: 25)
```

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

```
// Test case for minimum spanning 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
    int result = g.MinSpanTree();
    if (result == 0) { // MST of a completely disconnected graph should be 0
        std::cout << "Minimum Spanning Tree Weight (Disconnected): Passed (Expected: 0
        return true;
    } else {
        std::cout << "Minimum Spanning Tree Weight (Disconnected): Failed (Expected: 0
        return false;
    }
}
```

```
Testing Minimum Spanning Tree with disconnected graph.
Total weight of the Minimum Spanning Tree is 0
Minimum Spanning Tree Weight (Disconnected): Passed (Expected: 0, Actual: 0)
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))**

        Adds a new vertex to the graph by incrementing the vertex count, reallocating memory for the adjacency list and vertices array, and assigning a unique name to the new vertex. It initializes the adjacency list entry for the new vertex and confirms the addition with a printed message.

```
void Graph::AddVertex() {
    // Increment the number of vertices
    ++V;

    // Reallocate memory for the adjacency list with increased size
    Node** newAdj = new Node*[V]; ···
    vertices[V - 1] = std::string(1, newName);

    // Initialize the new adjacency list entry
    adj[V - 1] = nullptr;

    std::cout << "Vertex " << vertices[V - 1] << " added successfully." << std::endl;
}
```

    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))**

        Adds an edge between two existing vertices with a specified weight. It locates the indices of the source and destination vertices, updates their adjacency lists, and records the new edge in the edges vector for efficient access.

```
void Graph::AddEdge(const std::string& source, const std::string& destination, int weight) {
    // Validate the weight of the edge
    if (weight < 1 || weight > 40) {
        std::cerr << "Error: Edge weight must be between 1 and 40." << std::endl;
        return;
    }

    int u = -1, v = -1;
    // Find index of source and destination vertices
    for (int i = 0; i < V; ++i) {
        if (vertices[i] == source)
            u = i;
        if (vertices[i] == destination)
            v = i;
    }
    // Check if both vertices exist
    if (u != -1 && v != -1) {
        // Add edge from source to destination
        Node* newNode = new Node(v, weight);
        newNode->next = adj[u];
        adj[u] = newNode;
        // Add edge from destination to source for undirected graph
        newNode = new Node(u, weight);
        newNode->next = adj[v];
        adj[v] = newNode;

        // Add the edge to the edges vector
        edges.push_back({weight, {u, v}});
        std::cout << "Edge added successfully!" << std::endl;
    } else {
        std::cerr << "Error: One or both vertices not found!" << std::endl;
    }
```

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

Uses Kruskal's algorithm to find the Minimum Spanning Tree (MST) by iterating through sorted edges, adding them to the MST if they don't form a cycle, and merging vertex sets. It prints the MST edges and total weight after processing all edges.

```cpp
// Method to find the shortest path between two vertices and return the total weight
int Graph::ShortestPath(const std::string& src, const std::string& dest) {
    int source = -1, destination = -1;
    // Find index of source and destination vertices
    for (int i = 0; i < V; ++i) {
        if (vertices[i] == src)
            source = i;
        if (vertices[i] == dest)
            destination = i;
    }
    // Check if both vertices exist
    if (source != -1 && destination != -1) {
        // Call Dijkstra's algorithm to find shortest path
        std::vector<int> shortestPath;
        int distance = DijkstraShortestPath(source, destination, shortestPath);
        // Print the shortest path if it exists
        if (distance != INF) {
            std::cout << "Shortest Path from " << src << " to " << dest << ": ";
            for (size_t i = 0; i < shortestPath.size(); ++i) {
                std::cout << vertices[shortestPath[i]];
                if (i != shortestPath.size() - 1)
                    std::cout << " -> ";
            }
            std::cout << " (Distance: " << distance << ")" << std::endl;
            return distance; // Return the total weight of the shortest path
```

```cpp
// Dijkstra's algorithm to find shortest path from source to destination
int Graph::DijkstraShortestPath(int src, int dest, std::vector<int>& shortestPath) {
    // Priority queue to select the next vertex with the smallest distance
    std::priority_queue<iPair, std::vector<iPair>, std::greater<iPair>> pq;
    // Initialize distances to all vertices as infinite
    std::vector<int> dist(V, INF);
    std::vector<int> parent(V, -1); // Parent array to store shortest path
    // Distance to the source is 0
    pq.push(std::make_pair(0, src));
    dist[src] = 0;
    while (!pq.empty()) {
        int u = pq.top().second; // Get the vertex with the smallest distance
        pq.pop();
        // If reached destination, construct shortest path and return distance
        if (u == dest) {
            int v = dest;
            while (v != -1) {
                shortestPath.push_back(v);
                v = parent[v];
            }
            std::reverse(shortestPath.begin(), shortestPath.end());
            return dist[dest];
        }
        // Iterate through all adjacent vertices of u
        for (Node* neighbor = adj[u]; neighbor; neighbor = neighbor->next) {
            int v = neighbor->vertex;       // Adjacent vertex
            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 the distance to v
                pq.push(std::make_pair(dist[v], v)); // Push the updated distance to the priority queue
                parent[v] = u; // Update parent for constructing shortest path
            }
        }
    }
    return INF; // No path exists
}
```

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

Finds the shortest path between two vertices using Dijkstra's algorithm, identifying the indices of the source and destination vertices. It computes and prints the shortest path and its distance if a path exists, or notifies the user if no path is found.

```
//Minimum Spanning Tree usking Kurskal's algorithm.
int Graph::MinSpanTree() {
    // Create a disjoint set to keep track of connected components
    DisjointSets ds(V);
    // Initialize MST weight
    int mstWeight = 0;
    // Container to store the edges of the MST
    std::vector<std::pair<int, int>> mstEdges;

    // Iterate through all sorted edges
    for (size_t i = 0; i < edges.size(); ++i) {
        int u = edges[i].second.first;
        int v = edges[i].second.second;
        // Find the sets of u and v
        int set_u = ds.find(u);
        int set_v = ds.find(v);
        // Check if including this edge creates a cycle
        if (set_u != set_v) {
            // Merge the sets of u and v
            ds.merge(set_u, set_v);
            // Increment MST weight
            mstWeight += edges[i].first;
            // Add edge to the MST edges list
            mstEdges.push_back({u, v});
        }
    }

    // Print the edges of the MST
    std::cout << "Edges in the Minimum Spanning Tree:\n";
    for (const auto& edge : mstEdges) {
        std::cout << vertices[edge.first] << " - " << vertices[edge.second] << "\n";
    }

    // Print the total weight of the minimum spanning tree
    std::cout << "Total weight of the Minimum Spanning Tree is " << mstWeight << std::endl;
    return mstWeight;
```

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 for most of my error handling. I would throw it in there to see what was wrong with it. This is especially true for Dijkstra's algorithm.

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!).**

Done!