


Fernando Salazar

## Final Projects (Graphs)

- 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!),
- 2.● (20%) Create some tests (at least  for each piece of functionality) **before** you start coding...
- (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)`),
  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)`),
  3. (15%) a function for a shortest path algorithm (perhaps `shortest_path(source, destination)`),
  4. (15%) a function for a minimum spanning tree algorithm (example `min_span_tree()`).
- 4.● (10%) Analyze the complexity of all of your graph behaviors (effectively a part of our documentation for grading purposes),
- 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!).

### 1. Design Graph class

Going to need edges and vertices

It will need to have two algorithms, one using Dijkstra to find the shortest path and one for the minimum spanning tree.

Vertices

Add vertex function, name value

Edges

Add edge function, value for both beginning and end of edge, weight value for path.

Shortest Path

Function will need to have a starting node and an ending node

Some loop functions will go through all the edge weights to find the lowest path.

Minimum span tree

Function will need to have a starting node

Using prims algorithm

Check function

Check for vertex and edges

## 2. Testing for potential Functions (two tests per function)

// add vertex function tests

// test one, add one vertex, and check it in.

```
5 // Test functions
6 void testAddVertexOne() {
7     Graph g;
8     g.addVertex("A");
9
10    if (g.getVertexIndex("A") != -1) {
11        std::cout << "Test Add Vertex One Passed" << std::endl;
12    } else {
13        std::cout << "Test Add Vertex One Failed" << std::endl;
14    }
15 }
```

// test two, add more than one vertex, and check they are all in.

```
void testAddVertexMultiple() {
    Graph g;
    g.addVertex("A");
    g.addVertex("B");
    g.addVertex("C");

    if (g.getVertexIndex("A") != -1 && g.getVertexIndex("B") != -1 && g.getVertexIndex("C") != -1) {
        std::cout << "Test Add Vertex Multiple Passed" << std::endl;
    } else {
        std::cout << "Test Add Vertex Multiple Failed" << std::endl;
    }
}
```

// add edges functions tests

// test one, add one edge, and check that it has been added.

```
249
250 void testAddEdgeOne() {
251     Graph g;
252     g.addVertex("A");
253     g.addVertex("B");
254     g.addEdge("A", "B", 1);
255
256     if (g.adjacencyMatrix[g.getVertexIndex("A")][g.getVertexIndex("B")] != INF) {
257         std::cout << "Test Add Edge One Passed" << std::endl;
258     } else {
259         std::cout << "Test Add Edge One Failed" << std::endl;
260     }
261 }
262
```

// test two, add more than one edge and check that they have been added.

```
void testAddEdgeMultiple() {
    Graph g;
    g.addVertex("A");
    g.addVertex("B");
    g.addVertex("C");
    g.addEdge("A", "B", 1);
    g.addEdge("A", "C", 2);
    g.addEdge("B", "C", 3);

    if (g.adjacencyMatrix[g.getVertexIndex("A")][g.getVertexIndex("B")] != INF &&
        g.adjacencyMatrix[g.getVertexIndex("A")][g.getVertexIndex("C")] != INF &&
        g.adjacencyMatrix[g.getVertexIndex("B")][g.getVertexIndex("C")] != INF) {
        std::cout << "Test Add Edge Multiple Passed" << std::endl;
    } else {
        std::cout << "Test Add Edge Multiple Failed" << std::endl;
    }
}
```

//test for shortest path

//Test for a shorter path, maybe direct or two edges only.

```
void testShortestPathSimple() {
    Graph g;
    g.addVertex("A");
    g.addVertex("B");
    g.addEdge("A", "B", 1);

    g.shortestPath("A", "B");
    // Expected output: Shortest path from A to B: A B
    // You can manually verify the output.
}
```

// test with longer paths and multiple paths.

```
void testShortestPathComplex() {
    Graph g;
    g.addVertex("A");
    g.addVertex("B");
    g.addVertex("C");
    g.addVertex("D");
    g.addEdge("A", "B", 1);
    g.addEdge("B", "C", 2);
    g.addEdge("A", "C", 4);
    g.addEdge("C", "D", 1);

    g.shortestPath("A", "D");
    // Expected output: Shortest path from A to D: A B C D
    // You can manually verify the output.
}
```

// test for minimum span tree

// a short or simple tree

```
void testMinSpanTreeSimple() {
    Graph g;
    g.addVertex("A");
    g.addVertex("B");
    g.addVertex("C");
    g.addEdge("A", "B", 1);
    g.addEdge("B", "C", 2);
    g.addEdge("A", "C", 3);

    g.minSpanTree();
    // Expected output: Minimum Spanning Tree:
    // A - B
    // B - C
    // You can manually verify the output.
}
```

// a longer graph with more edges and

```
void testMinSpanTreeComplex() {
    Graph g;
    g.addVertex("A");
    g.addVertex("B");
    g.addVertex("C");
    g.addVertex("D");
    g.addEdge("A", "B", 1);
    g.addEdge("A", "C", 4);
    g.addEdge("B", "C", 2);
    g.addEdge("B", "D", 3);
    g.addEdge("C", "D", 5);

    g.minSpanTree();
    // Expected output: Minimum Spanning Tree:
    // A - B
    // B - C
    // B - D
    // You can manually verify the output.
}
```

## //Results

```
● crossoft-MIEngine-Error-dkrb4hz5.0rd' '--pid=Microsoft-MIEngine-Pid-logwwgic.wuv' '--dbgExe=C:\msys64\
Test Add Vertex One Passed
Test Add Vertex Multiple Passed
Test Add Edge One Passed
Test Add Edge Multiple Passed
Shortest path from A to B: A B
Shortest path from A to D: A B C D
Minimum Spanning Tree:
A - B
B - C
Minimum Spanning Tree:
A - B
B - C
B - D
PS C:\Users\ferna\Documents\CS-260\FinalProject\Graph> █
```

### 3. Code for graph class.

#### // vertex function

```
// Function to add a vertex to the array
// needs a name for the vertex
void addVertex(const std::string& vertexName) {
    // checks that we havent gone past the array max
    if (vertexCount < maxVertices) {
        // if not it will add it to the matrix and increase the count
        vertexNames[vertexCount++] = vertexName;
    } else {
        //let user know they cant add more edges.
        std::cout << maxVertices << " edges is the maximum." << std::endl;
    }
}
```

#### //edge function

```
// Function to add an edge
// needs two vertices and a weight for the edge.
void addEdge(const std::string& source, const std::string& destination, int weight) {
    // grabs in the index location for the vertex
    int srcIndex = getVertexIndex(source);
    int destIndex = getVertexIndex(destination);
    // -1 means it does not exist, return from getvertex function
    if (srcIndex != -1 && destIndex != -1) {
        // all entres are bidirectional
        //wight added
        adjacencyMatrix[srcIndex][destIndex] = weight;
        adjacencyMatrix[destIndex][srcIndex] = weight;
    }
}
```

#### //shortest path function

```
// Function to find the shortest path between two vertices
// dijkstra algorithm
void shortestPath(const std::string& source, const std::string& destination) {
    // grabs the index
    int srcIndex = getVertexIndex(source);
    int destIndex = getVertexIndex(destination);
    // the first thing it does is check that both vertices exist.
    if (srcIndex == -1 || destIndex == -1) {
        return;
    }

    // Array to store the shortest distance
    int distances[maxVertices];
    // Array to mark visited vertices false by default
    bool visited[maxVertices] = { false };
    // Array to store the previous vertex in the shortest path
    int previous[maxVertices];

    // sets the distances and previous arrays
    for (int i = 0; i < maxVertices; ++i) {
        // at set to inf , so no connection
        distances[i] = INF;
        // -1 will tell us that it has no path has been added
        previous[i] = -1;
    }

    // Distance is set to 0 at start
    distances[srcIndex] = 0;
```

```

// Main loop to find shortest paths
while (true) {
    int smallest = -1;
    for (int i = 0; i < maxVertices; ++i) {
        //will loop looking for an unvisited and smallest or distance shorter
        if (!visited[i] && (smallest == -1 || distances[i] < distances[smallest])) {
            smallest = i;
        }
    }
    // break the loop when they have all been visted
    if (smallest == -1 || distances[smallest] == INF) {
        break;
    }
    // sets the visted to true
    visited[smallest] = true;

    // Update distances to neighbors
    // loops through all the vertex possible
    for (int neighbor = 0; neighbor < maxVertices; ++neighbor) {
        //checks if has been visited first
        if (adjacencyMatrix[smallest][neighbor] != INF && !visited[neighbor]) {
            // gets the distance(wight)
            int alt = distances[smallest] + adjacencyMatrix[smallest][neighbor];
            // if its smaller it will update.
            if (alt < distances[neighbor]) {
                distances[neighbor] = alt;
                previous[neighbor] = smallest;
            }
        }
    }
}

//creates the shortest path liar
int path[maxVertices];
int pathLength = 0;
for (int at = destIndex; at != -1; at = previous[at]) {
    path[pathLength++] = at;
}

// reverses the list
for (int i = 0; i < pathLength / 2; ++i) {
    std::swap(path[i], path[pathLength - i - 1]);
}

// Output the list for user to see
std::cout << "Shortest path from " << source << " to " << destination << ": ";
for (int i = 0; i < pathLength; ++i) {
    std::cout << vertexNames[path[i]] << " ";
}
std::cout << std::endl;

```

// spanning tree function

```
// Function to find the minimum spanning tree using Prim's algorithm
void minSpanTree() {
    // Array to store the parent node
    int parent[maxVertices];
    // Array store if vertex is in array, start as false
    bool inMinSpanTree[maxVertices] = { false };
    // Array to store the min weight
    int key[maxVertices];
    // Array to store edges
    int minSpanTreeEdges[maxVertices][2];
    // Number of edges in tree
    int minSpanTreeSize = 0;

    //loops through each and set the key and parent values, to start
    for (int i = 0; i < maxVertices; ++i) {
        key[i] = INF;
        parent[i] = -1;
    }

    // sets value of first vertex
    key[0] = 0;

    // Main loop to create tree
    for (int count = 0; count < maxVertices - 1; ++count) {
        //grabs the location
        int keymin = minKey(key, inMinSpanTree);
        // save into array that it has been added to tree
        inMinSpanTree[keymin] = true;

        // Update key values and parent index
        for (int i = 0; i < maxVertices; ++i) {
            //
            if (adjacencyMatrix[keymin][i] != INF && !inMinSpanTree[i] && adjacencyMatrix[keymin][i] < key[i]) {
                parent[i] = keymin;
                key[i] = adjacencyMatrix[keymin][i];
            }
        }
    }

    // Store the tree edges
    // looping may array size
    for (int i = 1; i < maxVertices; ++i) {
        //when its not -1, added to array
        if (parent[i] != -1) {
            minSpanTreeEdges[minSpanTreeSize][0] = parent[i];
            minSpanTreeEdges[minSpanTreeSize][1] = i;
            minSpanTreeSize++;
        }
    }

    // Output the tree edges
    std::cout << "Minimum Spanning Tree: " << std::endl;
    for (int i = 0; i < minSpanTreeSize; ++i) {
        std::cout << vertexNames[minSpanTreeEdges[i][0]] << " - " << vertexNames[minSpanTreeEdges[i][1]] << std::endl;
    }
}
```

#### 4. Code analysis (complexity)

// add edges

It should be  $O(1)$ ; it does not need to sort through the array just added; only check its size. Time never increases.

// add vertex

It should be  $O(n)$ . It only needs to pass through once, but it will increase as it passes through the larger array.

//shortest path

I think it's  $O(n^2)$ . The process will repeat itself each time for each vertex, so the amount per vertex will increase. It has two parts that require going through the whole array to find a match.

// min span tree

$O(n)$  for the main loop that needs to go through everything. It has a couple of parts. Setting the initial values will also cycle through but only at the very beginning.