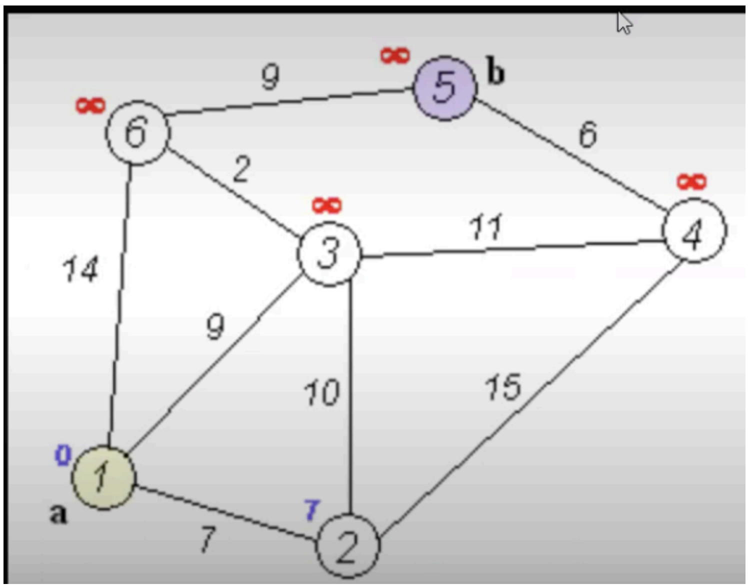
Quentin Creese
Jess
CS 260
DUE: 03/19/24

# Final Exam (Graphs)

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

Here is the graph I will be implementing into my code!
It can be found at: https://upload.wikimedia.org/wikipedia/commons/5/57/Dijkstra_Animation.gif




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

- Want to test:
    - I can add vertices to the graph
    - I can display the added vertices and the “neighbors”
    - Show empty neighbors if edges haven’t been created
    - I can add edges
    - I can display the added edges and the vertices “neighbors”
    - I can test the shortest path algorithm
    - I can test the minimum spanning tree algorithm

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

- (5%) a function to add a new vertex to the graph (perhaps add_vertex(vertex_name))

```
    void addNode(GraphNode node) {
        nodes.push_back(node);
    }
```

- (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
    void addEdge(Edge edge) {
        edges.push_back(edge);

        // Update neighbors of source and destination nodes
        nodes[edge.source->name - 'A'].neighbors.push_back(edges.size() - 1);
        nodes[edge.destination->name - 'A'].neighbors.push_back(edges.size() - 1);
    }

    // Function to add a new vertex to the graph
    void addVertex(char vertexName) {
        GraphNode newNode{vertexName};
        addNode(newNode);
    }

    // Function to add a new edge between two vertices of the graph
    void addEdgeBetweenVertices(char sourceName, char destinationName, int weight) {
        GraphNode* sourceNode = nullptr;
        GraphNode* destinationNode = nullptr;

        // Find source and destination nodes
        for (auto& node : nodes) {
            if (node.name == sourceName)
                sourceNode = &node;
            else if (node.name == destinationName)
                destinationNode = &node;
        }

        if (sourceNode && destinationNode) {
            Edge newEdge{weight, sourceNode, destinationNode};
            addEdge(newEdge);
        } else {
            std::cerr << "Error: Source or destination node not found." << std::endl;
        }
    }
```

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

```cpp
//Function to find the shortest path
std::vector<GraphNode*> shortestPath(char sourceName, char destinationName) {
    std::unordered_map<char, int> distance;
    std::unordered_map<char, char> previous;
    std::priority_queue<std::pair<int, char>, std::vector<std::pair<int, char>>, std::greater<std::pair<int, char>>> pq;

    // Initialize distances
    for (auto& node : nodes) {
        distance[node.name] = (node.name == sourceName) ? 0 : INT_MAX;
    }

    pq.push({0, sourceName});

    while (!pq.empty()) {
        char currentName = pq.top().second;
        pq.pop();

        if (currentName == destinationName) {
            break; // Reached the destination, exit loop
        }

        for (size_t neighborIndex : nodes[currentName - 'A'].neighbors) {
            const Edge& edge = edges[neighborIndex];
            char neighborName = (edge.source->name == currentName) ? edge.destination->name : edge.source->name;
            int totalDistance = distance[currentName] + edge.weight;
            if (totalDistance < distance[neighborName]) {
                distance[neighborName] = totalDistance;
                previous[neighborName] = currentName;
                pq.push({totalDistance, neighborName});
            }
        }
    }

    // Reconstruct path
    std::vector<GraphNode*> path;
    char currentName = destinationName;
    while (currentName != sourceName) {
        path.push_back(&nodes[currentName - 'A']);
        currentName = previous[currentName];
    }
    path.push_back(&nodes[sourceName - 'A']); // Add source node
    std::reverse(path.begin(), path.end());

    return path;
}
```

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

```cpp
    // Function for a minimum spanning tree algorithm (Prim's algorithm)
    std::vector<Edge> minSpanningTree() {
        std::vector<Edge> mst;
        std::unordered_set<char> visitedNodes; // Track visited nodes by their names
        std::priority_queue<std::pair<int, char>, std::vector<std::pair<int, char>>, std::greater<std::pair<int, char>>> pq;

        // Start from the first node
        visitedNodes.insert(nodes[0].name); // Assume nodes are added in the order of their names
        for (size_t neighborIndex : nodes[0].neighbors) {
            pq.push({edges[neighborIndex].weight, edges[neighborIndex].destination->name});
        }

        while (!pq.empty()) {
            char nodeName = pq.top().second;
            pq.pop();

            if (visitedNodes.find(nodeName) != visitedNodes.end()) {
                continue; // Skip if the node is already visited
            }

            visitedNodes.insert(nodeName);

            // Find the edge corresponding to the current node
            const Edge* edge = nullptr;
            for (const auto& e : edges) {
                if (e.source->name == nodeName || e.destination->name == nodeName) {
                    edge = &e;
                    break;
                }
            }

            if (edge) {
                mst.push_back(*edge); // Add edge to the spanning tree

                // Add edges incident to the current node to the priority queue
                for (size_t neighborIndex : nodes[nodeName - 'A'].neighbors) {
                    const Edge& neighborEdge = edges[neighborIndex];
                    char neighborName = (neighborEdge.source->name == nodeName) ? neighborEdge.destination->name : neighborEdge.source->name;
                    pq.push({neighborEdge.weight, neighborName});
                }
            }
        }

        return mst;
    }
```

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

Adding a new vertex would be O(1) because adding a new vertex involves appending a new node to the nodes vector. Since adding an element to the end of a vector has constant time complexity, the overall complexity of adding a new vertex is O(1).

Adding edges between vertices would also be O(1) because adding a new edge involves creating a new Edge object and updating the neighbor lists of the source and destination nodes. Since updating a vector has constant time complexity, the overall complexity of adding a new edge is O(1).

Finding the shortest path between two vertices has a complexity of O((V+E) * (log(V)) because the implementation uses Dijkstra's algorithm with a priority queue. In the worst case, all edges and vertices may need to be explored, resulting in O(V + E) iterations of the algorithm. Each iteration involves updating the priority queue, which has a cost of O(log(V)). Therefore, the overall complexity is O((V + E) * log(V)).

Finding the minimum spanning tree has a complexity of O((V + E) * log(V)) because the implementation uses Prim's algorithm with a priority queue. Similar to Dijkstra's algorithm, the worst-case complexity is O((V + E) * log(V)), where V is the number of vertices and E is the number of edges. This complexity arises from the repeated selection of the minimum-weight edge and updating the priority queue.

(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

Output from main.cpp:

```
quentin@Quentins-MacBook-Pro CS260_final % ./main
Graph Structure:
Node A neighbors: (A - B Weight: 7) (A - C Weight: 9) (A - F Weight: 14)
Node B neighbors: (A - B Weight: 7) (B - C Weight: 10) (B - D Weight: 15)
Node C neighbors: (A - C Weight: 9) (B - C Weight: 10) (C - D Weight: 11) (C - E Weight: 2)
Node D neighbors: (B - D Weight: 15) (C - D Weight: 11) (D - E Weight: 6)
Node E neighbors: (C - E Weight: 2) (D - E Weight: 6) (E - F Weight: 9)
Node F neighbors: (A - F Weight: 14) (E - F Weight: 9)
Shortest path from A to E:
A C E
Minimum spanning tree:
A - B (Weight: 7)
A - C (Weight: 9)
C - E (Weight: 2)
B - D (Weight: 15)
A - F (Weight: 14)
```

Output from graph_testing.cpp:

```
quentin@Quentins-MacBook-Pro CS260_final % ./graph_test
Testing adding vertices to the graph:
Graph Structure:
Node A neighbors: No neighbors
Node B neighbors: No neighbors
Node C neighbors: No neighbors

Testing adding edges to the graph:
Graph Structure:
Node A neighbors: (A - B Weight: 7) (A - C Weight: 9)
Node B neighbors: (A - B Weight: 7) (B - C Weight: 10)
Node C neighbors: (A - C Weight: 9) (B - C Weight: 10)

Testing finding the shortest path:
Shortest path from A to E:
A C E

Testing finding the minimum spanning tree:
Minimum spanning tree:
A - B (Weight: 7)
A - C (Weight: 9)
C - E (Weight: 2)
B - D (Weight: 15)
A - F (Weight: 14)
```