Leland Wendel

CS-260

Final Project (Graph)

I used the files found here as a project template.

(https://github.com/Joseph-I-Jess/cs260_winter_2024/tree/main/module_final)

1. A function to add a new vertex to the graph

```
18      // add node
19      // takes a single parameter "new_node"
20      void add_node(GraphNode *new_node)
21      {
22          nodes.push_back(new_node); // adds element new_node to the 'nodes' vector
23      }
```

The add_node function takes a single parameter "new_node", which is a pointer to a GraphNode object. "new_node" represents the node that will be added to the graph. At line 22, the push_back() function from the vector class adds the element to the end of the vector "nodes". This function essentially adds a new node to the graph.

Testing/Evaluation:

The GraphNode struct is defined as:

```
struct GraphNode
{
    std::string name;
    std::vector<Edge *> neighbors;
};
```

First, I need to create GraphNode objects:

```
    // create nodes
    GraphNode A{"Corvallis"};
    GraphNode B{"Salem"};
    GraphNode C{"Eugene"};
    GraphNode D{"Portland"};
    GraphNode E{"Mt Hood"};
    GraphNode F{"Bend"};
```

The GraphNode objects A – F are initialized with the corresponding names: Corvallis, Salem, Eugene, Portland, Mt Hood, and Bend . These objects are defined locally in the main function.

Next, I need to add nodes A-F to the graph:

```
    g.add_node(&A);
    g.add_node(&B);
    g.add_node(&C);
    g.add_node(&D);
    g.add_node(&E);
    g.add_node(&F);
```

Here, 'g' is an instance of the Graph class. The add_node function takes a pointer to GraphNode and appends it to the 'nodes' vector, adding the node to the graph.

Next, I need to print the graph's current nodes to verify that everything is working:

```
    // print current nodes in the graph
    void print_nodes()
    {
        cout << "\nCurrent nodes in the graph:" << endl;
        for (GraphNode *node : nodes)
        {
            cout << node->name << endl;
        }
    }
```

The 'print_nodes()' function iterates through the nodes vector and prints the name of each node.

Finally, call the 'print_nodes()' function in main to print the current nodes of the graph:

```
Current nodes in the graph:
Corvallis
Salem
Eugene
Portland
Mt Hood
Bend
```

All nodes have been successfully added to the graph!
This matches my result from the design phase.

2. A function to add a new edge between two vertices of the graph

```
25     // add edge
26     // takes a single parameter "new_edge" (pointer to edge object)
27     void add_edge(Edge *new_edge)
28     {
29         // access source member of the new_edge object
30         // add new_edge pointer to the neighbors vector of source
31         new_edge->source->neighbors.push_back(new_edge);
32     }
```

The add_edge function takes one parameter "new_edge", which is a pointer to an Edge object. "new_edge" represents the edge that will be added to the graph. At line 31, the source member of the new_edge object is accessed and the push_back() function from the vector class adds the element to the end of the vector "neighbors" of the source node. This function essentially records the connection from the source node to the destination node as an edge.

Testing/Evaluation:

The Edge struct is defined as:

```
struct Edge {
    int weight;
    GraphNode *source;
    GraphNode *destination;
};
```

First, I need to create an Edge object:

```
34     // create edges
35     Edge AB{36, &A, &B};
36     Edge BA{36, &B, &A};
37     Edge AC{47, &A, &C};
38     Edge CA{47, &C, &A};
39     Edge AF{128, &A, &F};
40     Edge FA{128, &F, &A};
```

The Edge objects are created and named AB, BA, AC, CA, AF, & FA. These objects represent a connection between two nodes on the graph and each have a defined weight (distance in this case). The elements within the brackets are initialized to the Edge object, and they correspond to members of the Edge struct (shown above). On line 35, for example, '36' initializes the weight member of the struct, '&A' initializes the source member of the edge struct, and '&B' initializes the destination member of the edge struct.

```
// hook up the edges into the nodes
g.add_edge(&AB);
g.add_edge(&BA);
g.add_edge(&AC);
g.add_edge(&CA);
g.add_edge(&AF);
g.add_edge(&FA);
```

Here, 'g' is an instance of the Graph class. The add_edge function adds an edge to the graph by updating the source node's list of neighbors. It takes a pointer to an Edge object and appends that Edge to the neighbors vector.

Next, I need to verify that the edges have been added:

```
// source, destination complication might come up here!
cout << "A.neighbors[2]->destination->name: " << A.neighbors[2]->destination->name << endl;
cout << "A.neighbors[1]->destination->name: " << A.neighbors[1]->destination->name << endl;
cout << "A.neighbors[0]->destination->name: " << A.neighbors[0]->destination->name << endl;
```

```
A.neighbors[2]->destination->name: Bend
A.neighbors[1]->destination->name: Eugene
A.neighbors[0]->destination->name: Salem
```

The edges have been successfully added to the graph! (at least for this example)
This result matches the expected result from my design

3. A function for a shortest path algorithm

**Broken up into smaller sections:*

```
// shortest path (Dijkstra's algorithm)
// returns a vector of 'pair' objects where each pair consists of a GraphNode and an int
// dijkstra() takes one argument: a pointer to the starting GraphNode
vector<pair<GraphNode *, int>> dijkstra(GraphNode *start)
{
    // create a hash table (distances) that maps each GraphNode to an int
    // store the shortest known distance to each node from start node
    unordered_map<GraphNode *, int> distances;

    // initializes 'distances' for each node
    for (GraphNode *node : nodes)
    {
        // distances to each node is infinity (INT_MAX)
        distances[node] = INT_MAX;
    }
    // start distance should be zero (distance to itself)
    distances[start] = 0;

    // lambda function 'compare' to compare pairs (left vs right)
    auto compare = [](pair<int, GraphNode *> left, pair<int, GraphNode *> right)
    { return left.first > right.first; };  // if 1st ele of left pair > 1st ele of right pair - return true
    // priority queue (min-heap) to store pairs (GraphNode & int)
    // ordered using 'compare'
    priority_queue<pair<int, GraphNode *>, vector<pair<int, GraphNode *>>, decltype(compare)> pq(compare);
    // push start node on top of priority queue (PQ) because distance is zero
    pq.push({0, start});
```

This function returns a vector of 'pair' objects, consisting of a GraphNode (GN) pointer and an integer. The pair represents the shortest distance from the start node to each node in the graph. The Dijkstra function takes one argument 'start', which is a pointer to the starting GraphNode. At line 41, a hash table 'distances' is created to map each GN pointer to an int (distance). The shortest known distance from the start node to each node will be stored here. At line 44, a for loop iterates over distances and sets each node's distance to infinity (or just a really large number). At line 50, the starting node's distance is set to zero (or distance to itself). At lines 53 and 54, a 'compare' function is declared to compare pairs – returns true is the first element of the left pair is greater than the first element of the right pair (constructor for min-heap). At line 57, the priority queue is declared to store the pairs of 'int' and GN and it is ordered using the compare function. At line 59, the start node is pushed on top of the PQ with a distance of zero.

```
61      // loop until PQ is empty
62      while (!pq.empty())
63      {
64          // extract node with smallest distance from PQ
65          auto [current_distance, current_node] = pq.top(); // structured binding unpacks pair returned by 'pq.top()'
66          // pop off PQ
67          pq.pop();
68
69          // if current distance > known/recorded distance:
70          if (current_distance > distances[current_node])
71              // skip this node
72              continue;
73
74          // iterate through each neighbor (edge connected) of current_node
75          for (Edge *edge : current_node->neighbors)
76          {
77              GraphNode *neighbor = edge->destination;
78              // calculate distance to neighboring node for each edge
79              int distance = current_distance + edge->weight;
80
81              // if new distance < known/recorded distance
82              if (distance < distances[neighbor])
83              {
84                  // update distance
85                  distances[neighbor] = distance;
86                  // push new distance and node onto PQ
87                  pq.push({distance, neighbor});
88              }
89          }
90      }
```

At line 62, a while loop is set to run until the PQ is empty. At line 65, the node with the smallest distance is extracted from the PQ using a structured binding that unpacks the pair returned by 'pq.top()'. At line 70, the node is skipped if its current distance is greater than the distance already recorded. At line 75, a for loop iterates through neighbors (connected by an edge) of the current node and calculates the distance to each neighboring node. At line 82, if the calculated distance to the neighbor is less than the known distance, the distance is updated, and the node and new distance is pushed onto the PQ.

```cpp
        // declares a vector 'result' to store pair of (GraphNode and int)
        // int represents shortest distance from start node to GraphNode pointer
        vector<pair<GraphNode *, int>> result;
        // iterate over distances hash table
        for (auto &[node, distance] : distances) // unpack each pair from distances into 'node' and 'distance'
        {
            // push a new pair to the end of the result vector
            result.push_back({node, distance});
        }
        return result;
    }
```

After exiting the while loop (PQ is empty), a result vector is constructed from the distances map. The result vector contains pairs of each node and its corresponding shortest distance from the start node.

Testing/Evaluation:

I need to test the algorithm and compare the results to my design's results:

```cpp
86      // Test Dijkstra's algorithm from Corvallis
87      cout << "\nDijkstra's shortest paths from Corvallis:\n";
88      vector<pair<GraphNode *, int>> distances = g.dijkstra(&A);
89      for (auto &[node, distance] : distances)
90      {
91          cout << node->name << ": " << distance << endl;
92      }
93
94      // Test Dijkstra's algorithm from Eugene
95      cout << "\nDijkstra's shortest paths from Bend:\n";
96      distances = g.dijkstra(&F);
97      for (auto &[node, distance] : distances)
98      {
99          cout << node->name << ": " << distance << endl;
100     }
```

The Dijkstra function returns a vector of pairs (pointer to a GN and an int). At line 88, the function is called on graph 'g', starting from the node pointed to by '&A' (Corvallis). In the for loop, each pair is unpacked into 'node' and 'distance' and then each pair is printed to the console. At line 96, this process is repeated starting from the node pointed to by '&F' (Bend). The results are shown below:

```
Dijkstra's shortest paths from Corvallis:
Bend: 128
Mt Hood: 159
Portland: 82
Eugene: 47
Salem: 36
Corvallis: 0

Dijkstra's shortest paths from Eugene:
Bend: 129
Mt Hood: 189
Portland: 112
Eugene: 0
Salem: 66
Corvallis: 47
```

The results from the algorithm match the results from my design!

4. A function for a minimum spanning tree algorithm

Broken up into smaller sections:

```
    // minimum spanning tree (Prim's algorithm)
    // returns a vector of pointers to Edge objects
    vector<Edge *> prim()
    {
        // check if the graph has nodes
        if (nodes.empty())
            // if true, return empty vector
            return {};
        // inMST is a hash table to track if nodes are to be included in MST
        unordered_map<GraphNode *, bool> inMST; // track if included, True/False
        // vector to store edges that are part of the MST
        vector<Edge *> mst;

        // lambda function 'compare' to prioritize edges with smaller distances
        auto compare = [](pair<int, Edge *> left, pair<int, Edge *> right)
        { return left.first > right.first; };
        // PQ to store pairs (pointers to edges and edge weights) ordered with compare function
        priority_queue<pair<int, Edge *>, vector<pair<int, Edge *>>, decltype(compare)> pq(compare);

        // helper function
        // lambda function to add all edges associated with the node to PQ, if not in MST
        auto add_edges = [&pq, &inMST](GraphNode *node)
        {
            // node included in MST = true
            inMST[node] = true;
            // for each Edge pointer in the neighbors vector of node
            for (Edge *edge : node->neighbors)
            {
                // if destination node of the edge is not in MST
                if (!inMST[edge->destination])
                {
                    // then add edge pair into PQ
                    pq.push({edge->weight, edge});
                }
            }
        };
```

*The prim function returns a vector of pointers to Edge objects, which represent the edges in the minimum spanning tree (MST). At lines 109-111, I check if the graph is empty and return an empty vector if nodes.empty() is true. At lines 113 and 114, a hash table (inMST) is created to keep track of whether a node is included in the MST (true if it is, false if it isn't) and a vector (MST) is created to store edges that are a part if the MST. At line 125, the lambda function 'compare' defines the comparison for the priority queue (PQ), prioritizing edges with smaller weights (shorter distances). The function takes two parameters 'left' and 'right', both of which are pairs of (int, Edge *). At line 119, the statement is true if the first element (weight) of the 'left' pair is greater than the first element of the 'right' pair. At line 121, a PQ is created to store the pairs (weight, Edge*) that have been ordered by weight using the 'compare' function. At line 125, the lambda function 'add_edges' adds all edges from the given node to the PQ if the destination node is not already in MST. This function marks the node as included in inMST and iterates over the node's neighbors to push other eligible edges into the PQ.*

```
141         // Start with the first node
142         // add edges to PQ
143         add_edges(nodes[0]);
144 
145         // loop until PQ is empty and MST contains 'n-1' edges
146         while (!pq.empty() && mst.size() < nodes.size() - 1)
147         {
148             // extracts top element of PQ (weight and pointer to Edge)
149             auto [weight, edge] = pq.top(); // structured binding
150             pq.pop(); // removes element
151 
152             // if destination node is already in MST (true in the inMST map)
153             if (inMST[edge->destination])
154                 // skip current iteration
155                 continue;
156 
157             // edge is added to the MST
158             mst.push_back(edge);
159             // mark destination node as included in MST
160             // add all its edges (not yet in MST) to the PQ
161             add_edges(edge->destination);
162         }
163 
164         return mst;
165     }
```

At line 143, MST is initialized by calling 'add_edges' for the first node of the graph, adding its edges to the PQ. The while loop at like 146 will run until the PQ is empty and the MST contains 'n-1' edges (an MST for a connected graph contains 'n-1' edges). At lines 149 and 150, the edge on top of the PQ is selected and removed from the PQ. At lines 153 and 155, if the destination node of the edge is already in the MST, the edge is skipped. At line 158, if the edge is not in MST, then edge is added to MST. Finally, at line

161, the 'add_edges' function is called for the destination node to add its edges to the PQ.

Testing/Evaluation:

```
    // Test Prim's algorithm for MST
    cout << "\nPrim's minimum spanning tree:\n";
    vector<Edge *> mst = g.prim();
    for (Edge *edge : mst)
    {
        cout << "(" << edge->source->name << " - " << edge->destination->name << ", " << edge->weight << ")\n";
    }
```

The prim function returns a vector 'MST' containing edge pointers which represent the minimum spanning tree. The prim function is called on the current graph and the for loop iterates over each edge in MST. The details of each edge in the MST are then printed.

A second smaller graph was created to provide a second test:

```
    // Create a second graph
    Graph g2;
    GraphNode G{"Brookings"};
    GraphNode H{"Coos Bay"};
    GraphNode I{"Roseburg"};
    GraphNode J{"Medford"};

    g2.add_node(&G);
    g2.add_node(&H);
    g2.add_node(&I);
    g2.add_node(&J);

    Edge GH{107, &G, &H};
    Edge HG{107, &H, &G};
    Edge GI{162, &G, &I};
    Edge IG{161, &I, &G};
    Edge HJ{165, &H, &J};
    Edge JH{165, &J, &H};
    Edge IJ{97, &I, &J};
    Edge JI{97, &J, &I};

    g2.add_edge(&GH);
    g2.add_edge(&HG);
    g2.add_edge(&GI);
    g2.add_edge(&IG);
    g2.add_edge(&HJ);
    g2.add_edge(&JH);
    g2.add_edge(&IJ);
    g2.add_edge(&JI);

    cout << "\nPrim's minimum spanning tree (second graph):\n";
    mst = g2.prim();
    for (Edge *edge : mst)
    {
        cout << "(" << edge->source->name << " - " << edge->destination->name << ", " << edge->weight << ")\n";
    }
```

```
Prim's minimum spanning tree:
(Corvallis - Salem, 36)
(Salem - Portland, 46)
(Corvallis - Eugene, 47)
(Portland - Mt Hood, 77)
(Corvallis - Bend, 128)

Prim's minimum spanning tree (second graph):
(Brookings - Coos Bay, 107)
(Brookings - Roseburg, 162)
(Roseburg - Medford, 97)
```

The final result matches my prediction from the design portion!

5. Analyze the complexity of all of your graph behaviors

The add_node and add_edge functions both have a complexity of O(1) if their vectors don't need resizing, in which case the complexity is O(n).

In the Dijkstra function:

- *The distance initialization has a complexity of O(n), where 'n' is the number of nodes in the graph.*
- *The PQ initialization and push has a complexity of O(log n)*
- *Edge processing occurs once for each edge, so O(n) where n is the number of edges*
- *For each edge, the PQ operation is O(log n).*

Overall, the algorithm has a complexity of O(E log N), where E is the number of edges and N is the number of nodes .

In the prim function:

- *Empty graph check is O(1)*
- *Hash table and MST vector initialization is O(1)*
- *PQ initialization is O(1)*
- *Add_edges function is O(log n), where n is the number of edges*

Overall, the complexity of the prim function is O(E log N), where E is the number of edges and N is the number of nodes.