Assignment Three

Christian Sarmiento
christian.sarmiento1@marist.edu

November 16, 2024

1 Introduction

Assignment Three focuses on creating a program that parses files to create a graph and a binary search tree. The graph is complete with an adjacency matrix, adjacency list, and traversals. The binary search tree has an in-order traversal as well as keeping track of comparisons.

2 Graphs

2.1 Vertex.cpp

The file graphs1.txt contains a few different graphs to build. First, we make a Vertex class to store our data. Our header file gives us a nice overview of what our class contains:

```
class Vertex {

    public:

        // Instance Variables
        std::string myID = " ";
        bool proccessed = false;
        std::vector<Vertex*> myNeighbors;

        // Null & Full Constructors
        Vertex();
        Vertex(std::string data);

        // Getters
        std::string getID();
        bool isProccessed();
        std::vector<Vertex*> getNeighbors();
        std::string getNeighborIDs();

        // Setters
        void setID(std::string newID);
        void addNeighbor(Vertex* newVertex);

}; // Class Vertex
```

The class contains a few different attributes. We have a vertex ID (Line 6) to keep track of the vertex. We also have a boolean flag (Line 7) to help with future traversals. Most importantly, we have a vector that saves our neighbors, or the vertices that share an edge with the given vertex, to help traverse the graph. All of the methods to basic operations like returning some of these attributes. Our function getNeighborIDs() (line 18) iterates through our vector of vertices to return a full list as a string of vertex IDs to help with adjacency list output.

2.2 Graph.cpp

Our Graph class uses the Vertex class to build our graphs. We add vertices using this function:

```
 1 void Graph::addVertex(Vertex* newVertex) {
 2
 3     // Add vertex to the graph
 4     myVertices.push_back(newVertex);
 5     numVertices++;
 6
 7     // Add a row in the graph matrix
 8     myGraphMatrix.push_back(std::vector<std::string>(numVertices + 1, "0"));
 9
10     // Add column to the graph matrix
11     for (int i=0; i <= numVertices; i++)
12         myGraphMatrix[i].push_back("0");
13
14     // Set vertex ID for the matrix
15     myGraphMatrix[numVertices][0] = newVertex->getID();
16     myGraphMatrix[0][numVertices] = newVertex->getID();
17
18 } // addVertex()
```

First, we add the new vertex to our list of vertices for our graph object (Line 4). For adjacency matrix output purposes, we also keep track of the number of vertices we have with numVertices (Line 5). After we acknowledge the existence of the new vertex, we update our graph matrix for ourput purposes. This involves adding a row and column for the new vertex (Lines 8,11-12) and saving the vertex ID in the matrix after adding the row/column (Line 15, 16).

For adding edges, we have a similar type of function:

```
 1 void Graph::addEdge(std::string vertex1ID, std::string vertex2ID) {
 2
 3     // Variables
 4     Vertex* vertex1;
 5     Vertex* vertex2;
 6     int start = -1;
 7     int end = -1;
 8
 9     // Get both of the verticies
10     vertex1 = getVertex(vertex1ID);
11     vertex2 = getVertex(vertex2ID);
12
13     // Add an edge between them if both vertices exist
14     if (vertex1 && vertex2) {
15
16         vertex1->addNeighbor(vertex2);
17         vertex2->addNeighbor(vertex1);
18
19     } // if
20
21     else
22         std::cout << "Could not add an edge between " << vertex1ID << " and " << vertex2ID
               << "." << std::endl;
23
24     // Find the index at where the vertices exist in the adjacency matrix
25     for (int i = 1; i <= numVertices; i++) {
26
27         if (myGraphMatrix[i][0] == vertex1ID)
28             start = i;
29
30         if (myGraphMatrix[0][i] == vertex2ID)
31             end = i;
32
33     } // for i
34
35     // Add edge to the adjacency matrix
36     myGraphMatrix[start][end] = "1";
37     myGraphMatrix[end][start] = "1";
38
39 } // addEdge()
```

We first get the vertices to add edges to by iterating through our vertex list in the graph class using a helper method (lines 10,11). Once we have these vertices, we can add each of the vertices to the neighbor vector for each vertex to signify that they share an edge (Lines 16,17). After that, we do something similar that we did for adding a vertex so that we can update the adjacency matrix. This involves finding where the vertices exist in the adjacency matrix (Lines 25-33) and then adding an edge between them (Lines 36,37).

For our output, we have a few functions to deal with building our adjacency list, adjacency matrix, and traversals.

For our adjacency list, we have:

```
 1 void Graph::adjacencyList() const {
 2
 3     // Variables
 4     std::string currID = " ";
 5     std::string neighbors = " ";
 6
 7     // Iterate through each vertex in the graph to output the IDs of each vertex
 8     std::cout << "Adjacency List" << std::endl;
 9     for (int i = 0; i < myVertices.size(); i++) {
10
11         // Get the vertex ID
12         currID = myVertices[i]->getID();
13
14         // Get vertex neighbors
15         neighbors = myVertices[i]->getNeighborIDs();
16
17         // Print out the row for the adjacency list
18         std::cout << currID << ": " << neighbors << std::endl;
19
20     } // for i
21
22     std::cout << std::endl;
23
24 } // adjacencyList()
```

We first iterate through all the vertices that we have in our graph (Line 9). After doing this, we get the ID at the vertex we are at (Line 12). Then, we get all IDs of the neighbors for the current vertex using the aformentioned function getNeighborIDs() (Line 15). Remember, this function will iterate through the neighbors vector and will make one nice string with all the IDs. After doing this, we output the current vertexID and its neighbor's IDs and repeat the process for the rest of the vertices (Line 18).

For our adjacency matrix, we use this function:

```
 1 void Graph::adjacencyMatrix() const {
 2
 3     // Iterate through each row
 4     std::cout << "Adjacency Matrix:" << std::endl;
 5     for (const auto& row : myGraphMatrix) {
 6
 7         // Iterate through each cell and print out the value
 8         for (const auto& cell : row) {
 9
10             std::cout << (cell.empty() ? "0" : cell) << " ";
11
12         } // for cell
13
14         std::cout << std::endl;
15
16     } // for row
17
18     std::cout << std::endl;
19
20 } // adjacencyMatrix()
```

Most of the work for this function was done when we added our vertices and edges to the graph. All that is left to do is iterate through the rows and the cells (Lines 5, 8) and output each value (Line 10). The only issue that I could not fix in time is that this function will generate an extra column of 0s at the end of the matrix. This is most likely due to how the rows and columns are added in the addVertex() function, specifically with the conditions with the vector sizes (Line 8, 11-12 in addVertex()). I was not able to fix that, but the output for the graph before the row of 0s is complete and correct.

For Depth-First Traversal, we implement this function:

```
void Graph::depthFirstTraversal(Vertex* currVertex) {

    // Check that the vertex is valid
    if (currVertex) {

        // Check if the vertex is processed
        if (!currVertex->isProccessed()) {

            // Print out the ID and recognize that the vertex has been proccessed
            std::cout << currVertex->getID() << " ";
            currVertex->proccessed = true;

        } // if

        // Move on to the neighbors
        for (int i = 0; i < currVertex->myNeighbors.size(); i++) {

            // If the current vertex isn't processed yet, recursively process it
            if (!(currVertex->myNeighbors[i]->isProccessed()))
                depthFirstTraversal(currVertex->myNeighbors[i]);

        } // for

    } // if

    else
        std::cout << "That vertex does not exist." << std::endl;

} // depthFirstTraversal
```

This traversal performs in $O(V + E)$ time, where V is the number of vertices and E is the number of edges. We have V since we are visiting each vertex only once for each recursive call. We can only reach the vertex if it is connected by an edge, so E must be added.

Although not recursive, we use a queue for Breadth-First-Traversal:

```
oid Graph::breadthFirstTraversal(Vertex* initVertex) {

    // Variables
    Queue traversalQueue;
    Vertex* currVertex;

    if (initVertex) {

        // Load vertex into the queue
        traversalQueue.enqueue(initVertex);
        initVertex->proccessed = true;

        // Iterate through the rest of the vertices
        while (!traversalQueue.isEmpty()) {

            // Get the vertex in the queue to process the neighbors
            currVertex = traversalQueue.dequeue();
            std::cout << currVertex->getID() << " ";

            // Iterate through the neighbors
            for (int i = 0; i < currVertex->myNeighbors.size(); i++) {

                // If the current vertex isn't processed yet, add the neighbor vertex to the
                     queue
                if (!(currVertex->myNeighbors[i]->isProccessed())) {

                    traversalQueue.enqueue(currVertex->myNeighbors[i]);
                    currVertex->myNeighbors[i]->proccessed = true;

                } // if

            } // for

        } // while

    } // if

    else
        std::cout << "That vertex does not exist." << std::endl;

} // depthFirstTraversal
```

This function uses a queue since we need to look at the immediate neighbors first before going to the neighbors of the neighbors, otherwise known as the neighborhood. Like DFT, this algorithm performs in $O(V + E)$ time since each vertex is enqueued and dequeued once and are connect by edges in order to be reached.

2.3 Binary Search Tree

For our BinarySearchTree class, we use a NodeBinaryTree class to build the nodes for the tree. This is what the header file looks like for the nodes of the binary tree, as a overview for the class:

```
class NodeBinaryTree {

     // Instance Variables
    public:

        std::string myData = " ";
        NodeBinaryTree* myLeft = nullptr;
        NodeBinaryTree* myRight = nullptr;
```

```
 9 
10 
11         // Null & Full Constructors
12         NodeBinaryTree();
13         NodeBinaryTree(std::string data);
14 
15         // Getters
16         std::string getData();
17         NodeBinaryTree* getLeft();
18         NodeBinaryTree* getRight();
19 
20         // Setters
21         void setData(std::string data);
22         void setLeft(NodeBinaryTree* node);
23         void setRight(NodeBinaryTree* node);
24 
25 }; // Class NodeLinkedList
```

This basically outlines what each node looks like. Each node contains a string data variable (Line 6), and pointers to other binary tree nodes for the left and right nodes (Lines 7,8). This also contains basic getter and setter methods to help retrieve things from the nodes.

For our binary tree class, we can insert data using this function:

```
 1 NodeBinaryTree* BinarySearchTree::insertNode(NodeBinaryTree* node, std::string newData) {
 2 
 3     // Check if root node is null
 4     if (node == nullptr)
 5         node = new NodeBinaryTree(newData);
 6 
 7     // If the data is greater than or equal to the node, add to the right
 8     else if (newData >= node->getData()) {
 9 
10         std::cout << "R" << " ";
11         node->myRight = insertNode(node->myRight, newData);
12 
13     } // if
14 
15     // If the data is less than the node, add to the left
16     else if (newData < node->getData()) {
17 
18         std::cout << "L" << " ";
19         node->myLeft = insertNode(node->myLeft, newData);
20 
21     } // if
22 
23     return node;
24 
25 } // insertNode()
```

To insert nodes, we use recursion to traverse through the trees depending on id the data value is greater/equal to or less than the current node (Lines 8-21). Our base case is when we hit a null pointer, or a leaf node (Lines 4,5). This results in a $O(log_2 n)$ insertion time.

We search for items in a similar fashion:

```
1 NodeBinaryTree* BinarySearchTree::search(NodeBinaryTree* node, const std::string& value, int
      & comparisons) const {
2 
3     // Variables
4     NodeBinaryTree* result = nullptr;
5 
6     // Make sure node is valid
```

```
 7     if (node != nullptr) {
 8
 9         // Found the result
10         comparisons++;
11         if (node->getData() == value) {
12
13             std::cout << "Found " << value << std::endl;
14             result = node;
15
16         } // if
17
18         // Result is less than
19         else if (value < node->getData()) {
20
21             std::cout << "L" << " ";
22             result = search(node->getLeft(), value, comparisons); // Search in left subtree
23
24         } // else if
25
26         // Result is greater than or equal to
27         else {
28
29             std::cout << "R" << " ";
30             result = search(node->getRight(), value, comparisons); // Search in right
                   subtree
31
32         } // else
33
34     } // if
35
36     return result; // Return the result at the end
37
38 } // search()
```

Similar to insertion, we use recursion to go through the tree to find our target value (Lines 19-32). The base case changes to stop when the target is found. This function performs in $O(log_2 n)$ time due to the recursion. To find an item, we have an average of 11.95 comparisons to find an item in a binary tree given 666 magic items and 42 items to find in the tree.

For an in-order traversal, we use the following algorithm:

```
void BinarySearchTree::inOrderTraversal(NodeBinaryTree* node) {

    // Check if the node exists
    if(node != nullptr) {

        // Traverse the tree
        inOrderTraversal(node->getLeft());
        std::cout << node->getData() << " ";
        inOrderTraversal(node->getRight());

    } // if

} // inOrderTraversal()
```

This again uses recursion to get through the entirety of the tree. The print statement in between the two recursion calls ensures an in order traversal (left, root, right). This has a time complexity of $O(N+E)$, where N is the number of nodes and E is the number of edges connecting the nodes of the tree. This is because to traverse the tree, we must visit each node once (N) and each node is connected with an edge (E). Sounds familiar.

3 References

- GeeksForGeeks - Binary Search Tree in C++
- GeeksForGeeks - Implementing an Adjacency Matrix
- Labouseur.com - Algorithms Slides
- ChatGPT - Personal Chat for C++ Syntax & Help