Assignment Three – Graphs & Trees

Ryan Munger
Ryan.Munger1@marist.edu

November 14, 2024

1 Introduction

1.1 Goals

Assignment 3 focuses on undirected graphs and binary search trees. I was tasked with reading in a list of instructions such as *add vertex* and *add edge* and build an undirected graph with them. I was able to add vertexes, add edges, traverse the graph, and keep an adjacency list, a matrix, and linked objects to represent it. In my binary search trees, I was also able to place items, find items, and traverse them in order.

1.2 Write-up Format

In this report I will describe the logic being presented. Below the text explanation, relevant code will follow in C++. Unfortunately, as we approach finals season, I no longer have the spare time to also create the assignment in Ada :(.

1.3 Limerick of Luck

When life feels undirected
And your vertices become disconnected,
    One must traverse within,
    Letting one's edges win,
Rendering your attitude corrected.

How did the binary tree learn so much? - *It excelled at branching out...*

---

2 Undirected Graphs

2.1 Reading the Instructions

I was provided a file of instructions to create undirected graphs such as *add vertex* and *add edge*. Since I am in Dr. Norton's Formal Languages class, regex immediately came to mind. Capture groups make this task very easy. When I encountered the *new graph* command, I displayed the current graph and initialized a new one.

```
void createGraphs(const string& filename) {
    int graphCount = 1;
    Graph currentGraph(to_string(graphCount));
    // I am taking formal lang so regex it'll be!
    // this will take care of any small whitespace errors as well
    regex newGraphRe(R"(new graph)");
    regex addVertexRe(R"(add\s*vertex\s*(\S+))");
    regex addEdgeRe(R"(add\s*edge\s*(\S+)\s*-\s*(\S+))");

    ifstream file(filename); // input file stream
    if (!file) {
        cerr << "File opening failed." << endl;
    }
    string instruction;

    while (getline(file, instruction)) {
        // ignore any commands we don't know, empty lines, comments etc.
        // regex will allow some slack with white space, but assuming perfect syntax by user

        // case 1: start a new graph
        if (regex_match(instruction, newGraphRe)) {
            // check to see if the current graph has anything in it
            // if not, no need to start a new one
            if (!currentGraph.isEmpty()) {
                currentGraph.displayGraph();
                graphCount++;
                currentGraph = Graph(to_string(graphCount)); // start a new graph
            }
        } else {
            smatch match; // captures subexpressions/groups

            if (regex_match(instruction, match, addVertexRe)) { // case 2: new vertex
                string newVertex = match[1].str();
                currentGraph.addVertex(newVertex);
            } else if (regex_match(instruction, match, addEdgeRe)) { // case 3: new edge
                string v1 = match[1].str();
                string v2 = match[2].str();
                currentGraph.addEdge(v1, v2);
            };
        };
    };
```

```
    file.close();
    currentGraph.displayGraph(); // don't forget the last one!!
};
```

2.2 Graph Object

The graph object is pretty simple. Each graph has an ID, and several representations of it. First, we have an adjacency list. This is simply a list of neighbors mapped to each vertex. Next, we have our matrix. I used a map to keep track of the index I placed each vertex into the matrix since the user can name them whatever they want. This matrix is a 2d array that stores where edges are using a '1' or a '.' at the respective location. Finally, I stored each linked vertex object (that has an ID, a processed flag for traversals, and a list of neighbors) in a map corresponding to their ID so that I can look them up quickly.

```
class Graph {
    private:
        string graphID;
        // Each map vertex stores a list of neighbors
        map<string, vector<string>> adjacencyRep;

        // map whatever user named the vertex (string) to a matrix index (int)
        map<string, int> vertexToMatrixID;
        map<int, string> matrixToVertexID; // so we reverse lookup
        vector<vector<string>> matrixRep; // 2d matrix to represent relations

        // keep track of our vertex objects
        // vertex name -> vertex object
        // use a map so we can actually look them up without checking them all
        map<string, linkedVertex> linkedObjs;
```

```
struct linkedVertex {
    string id;
    bool processed;
    vector<linkedVertex*> neighbors; // no limit to neighbors!
};
```

2.3 Add a Vertex

When adding a vertex, we must update all three representations of the graph. My code works just fine if we add vertexes after edges (as long as those edges didn't reference a nonexistent vertex of course).

1. If the matrix is empty, start a matrix (1x1).
2. If the matrix is not empty, resize it by adding a column and a row.
3. Store the index we stored the new vertex at.

4. Start an adjacency list for this vertex.

5. Create a new linkedVertex object and store it.

```
    void addVertex(string vertex) {
        // add the vertex to the matrix using next available index
        if (vertexToMatrixID.count(vertex) == 0) { // make sure we haven't already added it!
            vertexToMatrixID[vertex] = matrixRep.size(); // keep track of where we put it
            matrixToVertexID[matrixRep.size()] = vertex;
            // default to no neighbors
            if(this->isEmpty()) { // start the matrix
                matrixRep.push_back(vector<string>(1, "."));
            } else { // increase matrix size
                this->matrixRep.push_back(vector<string>(matrixRep[0].size(), ".")); // new row
                // update relations for new row (no relations unless edge added) - new col
                for (vector<string>& vertex : this->matrixRep) {
                    vertex.push_back(".");
                }
            }
        }

        this->adjacencyRep[vertex]; // start an adj list for the vertex

        linkedObjs[vertex] = linkedVertex{vertex}; // store by value
    };
```

What do you call a vertex with no edges? *Lonely... you think that's funny?*

2.4 Adding an Edge

Since this graph is undirected, we must add two edges (one in the reverse direction as well).

1. Append the respective vertexIDs to the affected vertex's adjacency tables.

2. Update the row & column intersections of the vertices in the matrix with a 1.

3. Update the linkedVertex objects' neighbors list.

```cpp
    void addEdge(string vertex1, string vertex2) {
        // must do both as it is undirected
        this->adjacencyRep[vertex1].push_back(vertex2);
        this->adjacencyRep[vertex2].push_back(vertex1);

        this->matrixRep[vertexToMatrixID[vertex1]][vertexToMatrixID[vertex2]] = "1";
        this->matrixRep[vertexToMatrixID[vertex2]][vertexToMatrixID[vertex1]] = "1";

        this->linkedObjs[vertex1].neighbors.push_back(&linkedObjs[vertex2]);
        this->linkedObjs[vertex2].neighbors.push_back(&linkedObjs[vertex1]);
    };
```

2.5 Displaying the Graph

We must now display the different representations of the graph. The adjacency list is is simple, we can just print out each key in the map followed by the contents of the appropriate array. I printed them out in the order the user added them (because I am ~~too lazy to sort a combo of numbers and strings~~ experienced in UI/UX). The matrix is also easy to display, as it is a 2d array. I decided to skip the column headers as names above width 1 made the table hard to read. Finally, we must conduct a depth first and breadth first traversal of the linked objects.

To conduct a depth first traversal, we recursively visit each vertex, its first neighbor, and the neighbor of each neighbor... etc., effectively traveling deep before wide. Once we visit a node, we mark it as seen to prevent loops and printing the same ID multiple times. The time complexity of this is $O(n)$ because the execution is linearly related to the size of the graph. Since we visit each vertex and edge exactly once, our complexity is exactly $O(|V| + |E|)$ where V is the set of vertices and E is the set of edges.

To conduct a breadth first search, we must visit every vertex at each depth level before going further. We process each neighbor of the current vertex before the neighbors of the neighbors. Using a Queue, we can control the order of processing FIFO. This traversal is also $O(|V| + |E|)$, for the same reasons. We visit each vertex and edge exactly once. Keep in mind that we have twice the amount of edges we added since the graph is undirected.

```cpp
    void displayAdj() {
        cout << "\nAdjacency List:" << endl;
        // print them in the order user added the vertices
        // this is done bc the map will print 10 before 2 alphabetically
        for (const auto& pair : this->matrixToVertexID) {
            cout << setw(3) << pair.second << ": ";
            for (string neighbor : adjacencyRep[pair.second]) {
                cout << neighbor << " ";
            }
            cout << endl;
```

```
        }
    };

    void displayMatrix() {
        cout << "\nMatrix:" << endl;
        // column headers -- decided against this as the formatting did not look very good
        // cout << setw(5) << " ";
        // for(const auto& pair : matrixToVertexID) {
        //     cout << pair.second << " ";
        // }
        // cout << endl;
        // row headers and data
        for(size_t i = 0; i < this->matrixRep.size(); ++i) {
            cout << setw(5) << matrixToVertexID[i] << ": ";
            for(size_t j = 0; j < this->matrixRep[i].size(); ++j) {
                cout << this->matrixRep[i][j] << " ";
            }
            cout << endl;
        }
    };

    // recursively visit a vertex and then its children
    void depthFirstTraversal(linkedVertex* fromVertex) {
        if (fromVertex == nullptr) return;

        if (!fromVertex->processed) {
            cout << fromVertex->id << "->";
            fromVertex->processed = true;
        }
        for (linkedVertex* v : fromVertex->neighbors){
            if (v != nullptr && !v->processed) {
                depthFirstTraversal(v);
            }
        }
    };

    // use a queue to print vertexes in depth bands
    void breadthFirstTraversal(linkedVertex* fromVertex) {
        cout << "\nBreadth First Traversal: ";
        linkedVertex* cv;
        queue<linkedVertex*> q;
        q.push(fromVertex);
        fromVertex->processed = true;
        while (!q.empty()) {
            cv = q.front();
            q.pop();
            cout << cv->id << "->";
            for (linkedVertex* v : cv->neighbors) {
                if (!v->processed) {
                    q.push(v);
                    v->processed = true;
                }
            }
        }
        cout << "End" << endl;
```

```cpp
    };

    void displayGraph() {
        if (this->isEmpty()) {
            cout << "Graph " << this->graphID << " is empty silly!" << endl;
            return;
        }

        cout << "\n\nGraph " << this->graphID << " Display:" << endl;
        this->displayAdj();
        this->displayMatrix();

        // just start at the first vertex user created
        linkedVertex* defaultStart = &this->linkedObjs[this->matrixToVertexID.begin()->second];

        this->resetProcessedFlags(); // remove any flags from prior traversal
        cout << "\nDepth First Traversal: ";
        this->depthFirstTraversal(defaultStart);
        cout << "End" << endl;
        // also traverse any straggling or disconnected vertices
        for (auto& pair: linkedObjs) {
            if (!pair.second.processed) {
                cout << "\nDepth First Traversal: ";
                this->depthFirstTraversal(&pair.second);
                cout << "End" << endl;
            }
        }

        this->resetProcessedFlags(); // remove flags
        this->breadthFirstTraversal(defaultStart);
        for (auto& pair: linkedObjs) {
            if (!pair.second.processed) {
                this->breadthFirstTraversal(&pair.second);
            }
        }
    };
```

I have selected a smaller graph to show as output. Since the graph is undirected, we can see symmetry in the matrix! My nerd font makes the -> look great in my terminal...

```
Graph 1 Display:

Adjacency List:
  1: 2 5 6
  2: 1 3 5 6
  3: 2 4
  4: 3 5
  5: 1 2 4 6 7
  6: 1 2 5 7
  7: 5 6
```

```

Matrix:
    1: . 1 . . 1 1 .
    2: 1 . 1 . 1 1 .
    3: . 1 . 1 . . .
    4: . . 1 . 1 . .
    5: 1 1 . 1 . 1 1
    6: 1 1 . . 1 . 1
    7: . . . . 1 1 .

Depth First Traversal: 1->2->3->4->5->6->7->End

Breadth First Traversal: 1->2->5->6->3->4->7->End
```

3 Binary Search Trees

3.1 How it Works

Binary search trees store data based on their relation to the other data in the tree. To find a specific item or place an item, we can compare it to the other items stored in the nodes. If our item is less than a node, we should move to the left children of the node. If we are greater than or equal to, we go right. My tree has two "modes:" shorthand mode and regular mode. In shorthand mode, it will print the insertion/lookup path like: "L R L R !", to indicate the path. ! represents the final location. In regular mode, it will also print the value at the node we visited along with the action we took: ""Autumn" Insert Moves; (Root!):L -> (Alpha):R -> (Beta):L -> Nullptr -> !." This is great for visualization and debugging!

```cpp
template <typename T>
struct BinaryNode {
    T value;
    BinaryNode<T>* leftChild;
    BinaryNode<T>* rightChild;
};

class BinarySearchTree {
    private:
        BinaryNode<string>* root;

        // recursively destroy each child and its children
        void destroyTree(BinaryNode<string>* node) {
            if (node != nullptr) {
                destroyTree(node->leftChild);
                destroyTree(node->rightChild);
                delete node; // orphans created :(
            };
        };

    public:
        // constructor
        BinarySearchTree() {
```

```
            root = nullptr;
        };
        // alternate constructor
        BinarySearchTree(string str) {
            // I have just now learned that creating a new node
            // does NOT set left and right to nullptr but instead they
            // are totally uninitialized. led to some weird bugs.
            root = new BinaryNode<string>{str, nullptr, nullptr};
        };
        // destructor
        ~BinarySearchTree() {
            destroyTree(root);
            root = nullptr;
        };
```

3.2 Insertion

Inserting a new item into the tree will usually take $O(log(n))$ time. This is exactly like binary search! As we look for where to put our new item, we reduce the search space by half recursively (given the tree is balanced!!!). An unbalanced tree, such as one created from a sorted list, degrades all the way down to $O(n)$ time as we do not eliminate any of the search space as we move. We end up with a binary search stick. Not even a charlie brown tree, at least that has branches and leaves (needles)! If only someone smart taught us about AVL tree balancing or something like that...

1. Create a new binary node.
2. If we do not have a root, this node is now the root.
3. Until we find a nullptr (empty spot for item), we move down the child node structure.
4. If we are less than the current node, check its left child.
5. If we are greater or equal to current node, check its right child.
6. As we move, print out the paths we took.

```
        // can give detailed (each traversed node's value) output or just L R !; ! meaning placement
        string insert(string str, bool shorthand) {
            BinaryNode<string>* newItem = new BinaryNode<string>{str, nullptr, nullptr};
            string moves = "\"" + str + "\" Insert Moves; ";
            if (root == nullptr) {
                root = newItem;
                if (!shorthand) { return moves += "Nullptr -> !"; }
                else { return moves += "!"; };
            };

            BinaryNode<string>* searchLocation = root;
```

```
            // when it becomes null we can place our new item!
            while (searchLocation != nullptr) {
                if (toLowerCase(newItem->value) < toLowerCase(searchLocation->value)) {
                    if (!shorthand) { moves += "(" + searchLocation->value + "):L -> "; } // go left
                    else { moves += " L"; };

                    if (searchLocation->leftChild == nullptr) {
                        searchLocation->leftChild = newItem;
                        if (!shorthand) { return moves += "Nullptr -> !"; }
                        else { return moves += " !"; };
                    }
                    searchLocation = searchLocation->leftChild;
                } else { // less than equal to
                    if (!shorthand) { moves += "(" + searchLocation->value + "):R -> "; } // go right
                    else { moves += " R"; };

                    if (searchLocation->rightChild == nullptr) {
                        searchLocation->rightChild = newItem;
                        if (!shorthand) { return moves += "Nullptr -> !"; }
                        else { return moves += " !"; };
                    }
                    searchLocation = searchLocation->rightChild;
                };
            };
            delete newItem; // we have bigger problems than this if this runs...
            return "Something went VERY wrong...";
        };
```

3.3 Searching

Searching has the exact same time complexity as insertion for the same reasons, $O(log(n))$. Just as before, the same tree balancing risks apply.

1. Start at the root.
2. Until we find a nullptr (we didn't find our item), we move down the child node structure.
3. If we are less than the current node, check its left child.
4. If we are greater or equal to current node, check its right child.
5. As we move, print out the paths we took.

```
        // can give detailed (each traversed node's value) output or just L R !; ! meaning found
        string search(string str, bool shorthand) {
            BinaryNode<string>* searchItem = root;
            string path = "\"" + str + "\" Search Path; ";
```

```
            while (searchItem != nullptr) {
                if (searchItem->value == str) {
                    if (!shorthand) { return path += "(" + searchItem->value + ") -> !"; }
                    else { return path += " !"; };
                } else if (toLowerCase(str) < toLowerCase(searchItem->value)) {
                    if (!shorthand) { path += "(" + searchItem->value + "):L -> "; } // search left
                    else { path += " L"; };
                    searchItem = searchItem->leftChild;
                } else {
                    if (!shorthand) { path += "(" + searchItem->value + "):R -> "; } // search right
                    else { path += " R"; };
                    searchItem = searchItem->rightChild;
                };
            };
            if (!shorthand) { return path += "Nullptr -> Not Found"; }
            else { return path += "NF"; };
        };
```

3.4 In-Order Traversal

Left, root, right! An in-order traversal visits each node precisely once. Therefore, this is an $O(n)$ time operation, as the visits increase linearly with the size of the tree. Even if the tree is completely unbalanced, it remains linear time!

1. Start at the root.
2. Recursively in-order traverse the left side of the tree.
3. Print the root.
4. Recursively in-order traverse the right side of the tree.

So, the root ends up near the middle of the traversal! At each recursive call, the current node becomes the 'root' of a smaller tree. By always going left first, we get the items in sorted order!

```
        // awkward but need to start recursing
        // I used depth first so that it orders the items!
        void inOrderTraversal() { // left root right
            traverseInOrder(root);
        };

        // recursively visit left children, root, then right
        void traverseInOrder(BinaryNode<string>* root) {
            if (root == nullptr) { return; };
            traverseInOrder(root->leftChild);
            cout << "\"" << root->value << "\", ";
            traverseInOrder(root->rightChild);
        };
```

3.5 Tree in Action

Test cases with detailed output:
Code:

```
    cout << "\nBST Testing: " << endl;
    BinarySearchTree BST;
    cout << BST.insert("Root!", false) << endl; // root
    cout << BST.insert("Alpha", false) << endl; // L
    cout << BST.insert("Beta", false) << endl; // L R
    cout << BST.insert("Zebra", false) << endl; // R
    cout << BST.insert("Autumn", false) << endl; // L R L
    cout << BST.search("Root!", false) << endl;
    cout << BST.search("Alpha", false) << endl;
    cout << BST.search("Autumn", false) << endl;
    cout << BST.search("Not inserted", false) << endl;
    cout << "\nIn-order Traversal (Left Root Right):" << endl;
    BST.inOrderTraversal(); // they will be in order!
```

Output:

```
BST Testing:
"Root!" Insert Moves; Nullptr -> !
"Alpha" Insert Moves; (Root!):L -> Nullptr -> !
"Beta" Insert Moves; (Root!):L -> (Alpha):R -> Nullptr -> !
"Zebra" Insert Moves; (Root!):R -> Nullptr -> !
"Autumn" Insert Moves; (Root!):L -> (Alpha):R -> (Beta):L -> Nullptr -> !
"Root!" Search Path; (Root!) -> !
"Alpha" Search Path; (Root!):L -> (Alpha) -> !
"Autumn" Search Path; (Root!):L -> (Alpha):R -> (Beta):L -> (Autumn) -> !
"Not inserted" Search Path; (Root!):L -> (Alpha):R -> (Beta):R -> Nullptr -> Not Found
```

Loading up magic items:

```
Loading Magic Items into a BST:
"Saddle Blanket of Warmth" Insert Moves; !
"Cloak of the bat" Insert Moves;  L !
"Sword of Kings" Insert Moves;  R !
...
"Battle Axe +3, Earthshaker" Insert Moves;  L L L L R R R R L R !
"Book of Stealth" Insert Moves;  L L R L R L L L R !
```

In order traversal of magic items: (Its in order!)

```
In-order Traversal (Left Root Right):
"Aerewens armor", "Aerial's Dagger of magic missles", "Aibohphobia", ... "Ye Robe of Useless Things", "Zales Might",
```

Finding our sample of magic items:

```
Finding Requested Items:
"Kidnapper's Bag" Search Path;  L R L R R L R R L L L R L L R ! Comps: 16
"Eversol's Innebriator" Search Path;  L R L R R L R L L R R L L L ! Comps: 15
"Rope of climbing" Search Path;  L R R L L R R L ! Comps: 9
...
```

```
"Potion of the Hero's Heart" Search Path;  L R L R R R R R R !
    Comps: 10
"Link Tabbard" Search Path;  L R L R R L R R L L L R R L L L !
    Comps: 17
"Eyes of doom" Search Path;  L R L R R L R L L R R L L R L L R R !
    Comps: 19
```

```
    // load magic items
    cout << "\nLoading Magic Items into a BST:\n" << endl;
    vector<string> magicItems = getMagicItems(MAGICITEMS_PATH);
    BinarySearchTree* magicItemTree = new BinarySearchTree;
    for (string item : magicItems) {
        cout << magicItemTree->insert(item, true) << endl;
    };
    cout << "\nIn-order Traversal (Left Root Right):" << endl;
    magicItemTree->inOrderTraversal();

    // find the requested items
    cout << "\n\nFinding Requested Items:" << endl;
    vector<string> itemsToFind = getMagicItems(ITEMS_2_FIND_PATH);
    float totalComps = 0;
    int comps;
    for (string item : itemsToFind) {
        string searchPath = magicItemTree->search(item, true);
        comps = checkBSTComps(searchPath);
        totalComps += comps;
        cout << searchPath << " Comps: " << comps << endl;
    };
    cout << "\nAverage Comparisons Taken: " << fixed <<
    setprecision(2) << totalComps / itemsToFind.size() << endl;
```

4 Conclusion

Graphs are cool I guess. Graphs are a vital concept that drive a lot of the world as we know it. Social networks like we discussed in class aside, I use graphs every day! As a network technician here at Marist, everything such as our fiber link map, our topology, our firewall connections, our RF profiles, and routing are all graphs. You can think of almost anything as a graph if you try hard enough. The automata Dr. Norton has us create are graphs as well! My neural firings are a graph! Am I a graph? Do I even care if I am? Hopefully I'm directed...

One vertex to another: *V1: "I'm leaving you! I've found a better connection." V2: "Can we still be neighbors?"*

How did the graph get a job? *It was really good with networking...*
