Lab 4: Graphs and Trees

Nicholas Suchy
Nicholas.Suchy1@Marist.edu

April 30, 2023

1 Introduction

The goal of this lab was to implement graph and tree data structures, and to understand their performance.

For the first part of this lab, we were to read in data from graphs1.txt and then, using the data from within the file, create our own implementations of a matrix graph, adjacency list graph, and a linked objects graph. We were to print out each of these representations, and then print out the linked objects graph using a depth first and then a breadth first traversal.

For the second part of the lab, we were to create our own implementation of a binary search tree using data from the magicitems.txt file. Then we were to search for the items specified in magicitems-find-in-bst.txt and count the number of comparisons made and then compute the average number of comparisons.

2 Code Reuse and Changes from Previous Labs

This lab required the use of many previously made classes and code snippets from previous labs, even from outside of the course.

Previous classes such as the LinkedList and Node class were reused with minimal changes. LinkedList was given another method that allows it to print out its contents on one line rather than on seperate lines which helped with the presentation in this lab.

Additionally, the BSTNode and BinarySearchTree classes were reused but heavily modified from my Software Development I class. Some notable changes are adding a parent reference to BSTNodes instead of using a stack to traverse back up the tree, keeping track of the path a node is added/found at, and adjusting the tree to work with Stings rather than ints.

3 Structure

There are significantly more files that go into this lab than previous labs. this is because I decided that each graph implementation should have its own file. Additionally, several different types of

Nodes were being used, and each Node fulfilled a different purpose, resulting in several Node classes that are specialized for certain data structures.

This style of organization made sense to me as it better helped me categorize what went where and made debugging much easier as file sizes were smaller than having one giant file.

4 Graphs

The following section contains information such as my thought process, design choices, and much more when it came to implementing the Graphs.

4.1 Reading in the Data From graphs1.txt

```
63  try {
64              File graphFile = new File("graphs1.txt");
65              Scanner scanner = new Scanner(graphFile);
66
67              while (scanner.hasNextLine()) {
68                  String line = scanner.nextLine();
69                  String[] linePart = line.split(" ");
70
71                  // Finds where the requirements for one graph end and another graph begin
72                  if (linePart[0].equals("--")) {
73
74                      // Display Graphs when there is something in them
75                      if (!firstGraph) {
76                         presentation();
77                      }
78
...
89                  } else if (linePart[0].equals("add")) {
...
101                        matrixGraph.addVertex();
102
103                         // Adds the node read from the file to a LinkedList to be access later
104                         graphNodes.addBack(Integer.toString(vertex));
105
106                         // Reads from the file and adds an edge to the current graph
107                     } else if (linePart[1].equals("edge")) {
...
126                         // Adds edges to graphs
127                         matrixGraph.addEdge(vertex1, vertex2);
128                         adjListGraph.addEdge(vertex1, vertex2);
129                         linkObjGraph.addEdge(vertex1, vertex2);
130
131                     }
132                 }
133             }
134
135             scanner.close();
136
```

```
137         } catch (FileNotFoundException e) {
138             System.out.println("An error has occured");
139             e.printStackTrace();
140         }
```

The above code is a skeleton of the reading in code meant to highlight the importance of each part while abstracting more of the nuance.

When it came to reading in the graphs1.txt file, I struggled to figure out how to read and create the data in a reproducible way. I used W3Schools website to help me in reading in file lines as Strings, and then splitting the Strings at each space (" ") encounter, and then finally using these "lineParts" to decide what to do.

For example, the code reads in the line "add vertex 1" from the file. It then converts this line to a String and then splits the String into an array where each index is a sub-string broken where ever a space is encountered. For the above line, the array of sub-strings would look like ["add", "vertex", "1"]. The program then goes through logic statements to determine if the file is adding a vertex to a graph, adding an edge to a graph, or creating an entirely new graph.

Some of the nuance left out to better fit the code, was always setting the first node to have an id of 0. This allows for a more streamlined way of accessing and addressing Nodes in graphs later down the line.

4.2 Matrix Graph

This section revolves around adding vertices and edges to the Matrix Graph, the print function doesn't pertain the the core of the lab and therefore is left out of analysis.

4.2.1 Adding a Vertex

```
12      public void addVertex() {
13          this.size++;
14          int[][] newMatrix = new int[this.size][this.size];
15
16          // Copies over data from old matrix into the new, larger matirx
17          for (int i = 0; i < this.size - 1; i++) {
18              for (int j = 0; j < this.size - 1; j++) {
19                  newMatrix[i][j] = this.matrix[i][j];
20              }
21          }
22          this.matrix = newMatrix;
23
24          // Initalizes the new vertex to have no edges in the graph
25          for (int i = 0; i < this.size - 1; i++) {
26              this.matrix[i][this.size - 1] = 0;
27              this.matrix[this.size - 1][i] = 0;
28          }
29      }
```

When adding a vertex to a Matrix Graph, a new row and column are added to the existing matrix. As the implementation of the matrix in this case uses arrays, an entirely new matrix was made. Data from the older, smaller matrix was copied over to the newer, larger matrix. Then finally, as a new vertex would not be connected to any edges, each spot in the new row and column is initialized with a 0.

4.2.2 Adding an Edge

```
32     public void addEdge(int vertex1, int vertex2) {
33        this.matrix[vertex1][vertex2] = 1;
34        this.matrix[vertex2][vertex1] = 1;
35     }
```

In a matrix graph, vertices are denoted as the rows an columns of the matrix while edges fill in the area of the actual vertex. If there is a "0" where a row and column intersect, then those 2 nodes do not share an edge. If there is a "1" instead the two nodes do share an edge. When adding an edge between two vertices, the program goes to the row representing the first vertex then travels to the column representing the second vertex. It then sets the value at this location to "1" to signify that an edge from vertex one to vertex two has been added. Additionally, since the lab asked us to make undirected graphs the process is repeated in the reverse order so that vertex 2 can reach vertex 1.

4.3 Adjacency List Graph

```
1    public class AdjacencyListGraph {
2    LinkedList[] adjacencyList;
3    int size;
4
5    public AdjacencyListGraph(int size) {
6        this.adjacencyList = new LinkedList[size];
7        this.size = size;
8    }
```

This section revolves around the portion of the lab I had the most difficulty with: adding vertices and edges to create an adjacency list graph. Firstly, its notable to point out that although in class we had discussed using a hash table to implement this form of a graph, I used an array of Linked Lists which functions similarly. Additionally, The print function doesn't pertain the the core of the lab and therefore is left out of analysis.

4.3.1 Creating the Graph

```
35     public static AdjacencyListGraph adjListGraphCreation(LinkedList graphNodes) {
40         AdjacencyListGraph adjListGraph = new AdjacencyListGraph(graphNodes.length());
41
42         // Adds vertices to the graph from graphNodes
43         for (int i = 0; i < graphNodes.length(); i++) {
44             adjListGraph.addVertex(graphNodes.get(i).value);
45         }
46
47         return adjListGraph;
48     }
```

When it comes to creating the graph, the size of the graph is set on creation. After reading all "add vertex" statements from the file, the above function is called which creates the graph. This means vertices aren't added directly from the file to the graph, instead they are added to a Linked List. The above function uses the Linked List containing the vertices that need to be added and adds them to the graph. This implementation may not be optimal but it allowed me to tackle the part of the lab I struggled with the most.

4.3.2 Adding a Vertex

```
10    public void addVertex(String vertex) {
11        adjacencyList[Integer.parseInt(vertex)] = new LinkedList();
12    }
```

Adding a vertex to the graph involves taking the vertex read from the file, and then, using it's value, go to the matching index in the array and create a new Linked List in the location to hold neighbors in the future.

4.3.3 Adding an Edge

```
14     public void addEdge(int vertex1, int vertex2) {
15         adjacencyList[vertex1].addBack(Integer.toString(vertex2));
16         adjacencyList[vertex2].addBack(Integer.toString(vertex1));
17     }
```

Edges in adjacency lists are shown by what vertex has what neighbors. When adding an edge between two vertices in an adjacency list graph for the lab, one both vertices get added to the back of the Linked List at the other's index in the array.

4.4 Linked Objects Graph

```
4     public class LinkedObjectsGraph {
5     GraphNode[] vertices;
6     int size;
7
8     public LinkedObjectsGraph(int size){
9         this.vertices = new GraphNode[size];
10         this.size = size;
11     }
```

This section revolves around the linked objects graph.It has an array of Graph Nodes that holds all the vertices in the graph. Additionally, it has a size which is used when creating the array to determine how large to make the array.

4.4.1 Creating the Graph

```
47     public static LinkedObjectsGraph linkObjGraphCreation(LinkedList graphNodes) {
48         LinkedObjectsGraph linkObjGraph = new LinkedObjectsGraph(graphNodes.length());
49
50         // Adds verticies to the graph from graphNodes
51         for (int i = 0; i < graphNodes.length(); i++) {
52             linkObjGraph.addVertex(Integer.parseInt(graphNodes.get(i).value));
53         }
54
55         return linkObjGraph;
56     }
```

Creating the Linked Objects was conducted in a similar way to creating the adjacency list. Vertices are read from the file but are added to a Linked List rather than being immediately added to the graph. Then after all "add vertex" commands have been read, the above function creates the Linked Objects graph.

4.4.2 Graph Nodes

```
1       public class GraphNode {
2           int id;
3           GraphNode neighbors[];
4           boolean visited;
5
6           public GraphNode(int value){
7               this.id = value;
8               this.neighbors = new GraphNode[0];
9               this.visited = false;
10          }
11
12          public void addNeighbor(GraphNode vertex){
13              int newLength = (this.neighbors.length + 1);
14              GraphNode newArray[] = new GraphNode[newLength];
15              for (int i = 0; i < this.neighbors.length; i++) {
16                  newArray[i] = this.neighbors[i];
17              }
18              this.neighbors = newArray;
19              this.neighbors[this.neighbors.length - 1] = vertex;
20          }
21      }
```

The core of the Linked Objects Graph is the specialized nodes. These graph nodes have an id to be identified by, an array of neighbors, and a boolean value that is used for traversals to determine if the vertex has been processed.

Additionally, Graph Nodes have a method that allows them to add neighbors. The method takes another Graph Node as a parameter, resizes the current Graph Nodes neighbor array, and then adds the Graph Node passed as a parameter to the end of the array.

4.4.3 Adding a Vertex

```
13      public void addVertex(int vertex){
14          vertices[vertex]= new GraphNode(vertex);
15      }
```

When adding a new vertex, a new Graph Node is created at the index equal to the vertex's value. After first being created, the Graph Node has an id value equal to the vertex's value, an empty neighbors array, and the visited boolean in set to false.

4.4.4 Adding an Edge

```
17      public void addEdge(int vertex1, int vertex2) {
18          vertices[vertex1].addNeighbor(vertices[vertex2]);
19          vertices[vertex2].addNeighbor(vertices[vertex1]);
20      }
```

To add edges between graph nodes, neighbors are established. The above method takes two Graph Nodes and adds them to the other's neighbors array demonstrating that an undirected edge exists between them.

4.4.5 Depth First Traversal

```
29     public void depthFirstTraversal(GraphNode fromVertex) {
30         if (!fromVertex.visited) {
31            System.out.print(fromVertex.id);
32             fromVertex.visited = true;
33         }
34         for (int i = 0; i < fromVertex.neighbors.length; i++) {
35             if (!fromVertex.neighbors[i].visited) {
36                 System.out.print(" -> ");
37                 depthFirstTraversal(fromVertex.neighbors[i]);
38             }
39         }
40     }
```

The above code performs a Depth First Search. The algorithm is modeled after the Depth First Search we discussed in the class slides. When the method is called for the first time, it is passed a vertex in which it will begin its search from, stored as fromVertex. If fromVertex hasn't been processed before, it will print out the id of the Node. Then for all of fromVertex's neighbors, it checks if they've been processed. If a neighbor has not been processed, the method is called again, this time passing the neighbor as fromVertex.

As for the algorithms asymptotic run time, it has a linear time complexity. This is because in the worst case scenario, the algorithm will check every vertex and edge within the graph. Although the recursion and for loop seem to denote a quadratic run time, a single node can only be called as fromVertex a single time. This enforces a linear run time.

4.4.6 Breadth First Traversal

```
42     public void breadthFirstTraversal(GraphNode fromVertex) {
43         GraphNode currentVertex;
44         Queue<GraphNode> queue = new LinkedList<>();
45         queue.add(fromVertex);
46         fromVertex.visited = true;
47         while (!queue.isEmpty()) {
48             currentVertex = queue.remove();
49             System.out.print(currentVertex.id + " -> ");
50             for (int i = 0; i < currentVertex.neighbors.length; i++) {
51                 if (!currentVertex.neighbors[i].visited) {
52                     queue.add(currentVertex.neighbors[i]);
53                     currentVertex.neighbors[i].visited = true;
54                 }
55             }
56         }
57     }
```

The above method represents a Breadth First Traversal Algorithm. The method is modeled after the pseudo-code we went over in class. Additionally, It's important to point out that the algorithm doesn't use the queue we had built in a previous lab. It was mentioned in class that it was acceptable for this method to use the built in queue so I took advantage of this during a busy time of the year.

Unlike the Depth First Traversal, the method does not use recursion. When the method is called, a vertex is passed to the function as fromVertex. This vertex is where the method will start traversing the graph. Once passed to the function, fromVertex is then added to a queue. Then until the queue

is empty, the first Node in the queue is dequeued and denoted as processed. When a node is removed from the queue it then enters a for loop. For each of the Node's neighbors, if the neighbor has not been processed, add it to the queue and then mark the Node as processed. Once the while loop completes, the graph will have been printed out in a Depth First Traversal order.

As for the algorithms asymptotic run time, it has a linear time complexity. This is because in the worst case scenario, the algorithm will check every vertex and edge within the graph. Although there exist nested loops which would seem to indicate a quadratic run time, the use of the queue and only adding to queue if a Node as been unprocessed ensures that the runtime stays linear rather than quadratic as each node can only be added to the queue a max of one time.

5 Binary Search Tree

```
1     public class BinarySearchTree {
2     BSTNode root;
3     int size;
4
5     public BinarySearchTree(){
6         //this.root is the starting root of the tree
7         this.root = null;
8         this.size = 0;
9    }
```

This section covers the Binary Search Tree aspect of the lab. Methods and data structures specific to the implemented BST are explained. The above code show the constructor for the BST. When created, the tree is empty and therefore has a size of 0 and no root.

5.1 Reading from magicitems.txt

```
157    try {
158            // Reading in both magicitems and magicitems-find-in-bst
159            File file1 = new File("magicitems.txt");
160            File file2 = new File("magicitems-find-in-bst.txt");
161
162            // Scanners of each file
163            Scanner reader1 = new Scanner(file1);
164            Scanner reader2 = new Scanner(file2);
165
166            // Read in magicitems first to form a Binary Search Tree
167            while (reader1.hasNextLine()) {
168                String magicItem = reader1.nextLine().replace(" ", "").toLowerCase();
169                String path = magicItemsTree.add(magicItem);
170
171                // Prints out the path of where an item was placed in the tree
172                System.out.println(path);
173            }
174
175            reader1.close();
176
177            // Read in magicitems-find-in-bst to create a list of target from which to search
178            while (reader2.hasNextLine()) {
179                String magicItemTarget = reader2.nextLine().replace(" ", "").toLowerCase();
```

```
180                 targets.addBack(magicItemTarget);
181             }
182
183             reader2.close();
184
185             // Throw error if file path is not found
186         } catch (FileNotFoundException e) {
187             System.out.println("An error has occured");
188             e.printStackTrace();
189         }
```

When it comes to BST operations, the lab required us to read in data form two files. In order to do this I created two scanners one for each file and read them at separate times. Firstly, I read in data from the magicitems.txt file. For each line in the file, the line was converted to a String. Then the BST add() operation was called and recorded the path the Node took to find its new location was recorded. Then the path was printed out and the next line then followed suit. As for the second file, magicitems-find-in-bst.txt, the contents of this file were turned into Strings and then the String was added to a Linked List of targets to use to search the BST with.

5.2 BSTNode

```
1     public class BSTNode {
2         String value;
3         BSTNode left;
4         BSTNode right;
5         BSTNode parent;
6
7         public BSTNode(String value) {
8             this.value = value;
9             this.left = null;
10            this.right = null;
11            this.parent = null;
12        }
13     }
```

For the Binary Search Tree, a specialized Node class a made. These nodes hold a value and hold pointers for to up to three other nodes, a left child, a right child, and a parent.

5.3 Adding to The BST

```
11    public String add(String value) {
12        BSTNode tempNode;
13        BSTNode newNode = new BSTNode(value);
14        String path ="";
15
16        // If there is no root, creates a root for the tree
17        if(this.root == null){
18            this.root = newNode;
19            path = "This item is the root of the tree.";
20         } else {
21             tempNode = this.root;
22
23             // Loops when at least one child is found
```

```
24              while(tempNode.left != null || tempNode.right != null) {
25
26                  // Moves left if a child node is present and newNode comes before tempNode
27                  if(newNode.value.compareTo(tempNode.value) < 0 && tempNode.left != null){
28                      tempNode = tempNode.left;
29                      path += "L, ";
30
31                  // Moves right if a child node is present and newNode comes after tempNode
32                  } else if(newNode.value.compareTo(tempNode.value) > 0 && tempNode.right != null){
33                      tempNode = tempNode.right;
34                      path += "R, ";
35
36                  // Breaks out of the loop when a location is found
37                  } else {
38                      break;
39                  }
40              }
41
42              // Adds newNode to its correct location
43              if(newNode.value.compareTo(tempNode.value) < 0){
44                  tempNode.left = newNode;
45                  newNode.parent = tempNode;
46                  path += "L";
47              } else {
48                  tempNode.right = newNode;
49                  newNode.parent = tempNode;
50                  path += "R";
51              }
52          }
53          this.size += 1;
54          return (path);
55      }
```

The above method takes care of adding Nodes to the Binary Search Tree. After data is read in from magicitems.txt, the add() method is called. Firstly, it takes the String read from the file and and turns it into a BSTNode. It then checks tot see if the BST is empty. If so, it then marks the newly created node as the root of the tree. If the tree is populated, it then begins using Binary Search Tree logic to determine the correct spot to place the new node. The newly created Node is compared with the root of the tree: if its value is less (alphabetically before) the root and there is a left child the new node then begins comparisons with the left child, if its value is greater (aplhabetically after) the root and there is a right child the new node begins comparisons with the right child. If at any point, the new node is less or greater than the node it is comparing with and there is no child respective to where it would move, the new node is then added to the tree at that spot. While this is all happening the method is also keeping track of the path that the new node is taking to find it's spot. Once the new node is placed, it returns its path.

5.4 List BST In-Order

```
57      public LinkedList listInOrder() {
58          // Holds the values in order as the tree is traversed
59          LinkedList inOrderValues = new LinkedList();
60          BSTNode tempNode = this.root;
```

```
61
62        // Loops until the amount of Nodes in the list equals the amount of Nodes in the BST
63        while(inOrderValues.length < this.size){
64
65            // When there is a left child, go left
66            while(tempNode.left != null && inOrderValues.linearSearch(tempNode.left.value)[0] == 0
67                tempNode = tempNode.left;
68            }
69
70            // Once no more left children, add tempNode to list
71            if (inOrderValues.linearSearch(tempNode.value)[0] == 0) {
72                inOrderValues.addBack(tempNode.value);
73            }
74
75            // If there is a right child, go there
76            // Otherwise go back to the parent if it is not null
77            if((tempNode.right != null && inOrderValues.linearSearch(tempNode.right.value)[0] == 0
78                tempNode = tempNode.right;
79            } else if (tempNode.parent != null) {
80                tempNode = tempNode.parent;
81            }
82        }
83        return inOrderValues;
84    }
```

The above method prints out the BST in-order (alphabetically for a BST). It starts by traveling to the left most node in the tree. Then it adds this node to a Linked List, which, after the method is completed, will hold the Binary Search Tree in-order. Once the left most node is reached and added to the in-order list, the method then checks to see if the node has a right child, if so it sets the right child equal to tempNode, otherwise tempNode is set equal to the current tempNode's parent. Then using the new tempNode value, the function is repeated. It is important to note as well that the method constantly checks to makes sure it isn't adding the same node to the in-order list twice.

5.5 Find Element

```
86     public String[] findElement(String target){
87         String isInTree = "f";
88         String path = "";
89         Integer comparisons = 0;
90         BSTNode tempNode = this.root;
91
91         // Loops while tempNode exists
93         while(tempNode != null) {
94
95             // Checks if target has been found
96             comparisons++;
97             if(target.compareTo(tempNode.value) == 0){
98                 isInTree = "t";
99                 if (path == "") {
100                    path = "The target was found at the root of the tree";
101                }
102                break;
```

```
103             } else {
104 
105                 // Moves left when target comes before tempNode alphabetically
106                 // Moves right otherwise
107                 if(target.compareTo(tempNode.value) < 0){
108                     tempNode = tempNode.left;
109                     path += "L, ";
110                 } else {
111                     tempNode = tempNode.right;
112                     path += "R, ";
113                 }
114             }
115         }
116         if (isInTree == "f") {
117             path = "The target was not found in the tree";
118         }
119 
120         String[] returnArray = {isInTree, path, comparisons.toString()};
121         return returnArray;
122     }
```

As we discussed in class, finding items in a BST has a time complexity of O(h) where h is the height of the tree. In the worst case scenario, to find any item, you have to go from the root to the last item of the tree. If we look at a Linked List as a very poorly balanced BST the time complexity would be O(n), but since BSTs are typically much more balanced than that, we can say the time complexity is O(h).

The above function is passed a value that is to be searched for within the BST. It follows the same logic as adding a node to the tree, including keeping track of the path to find the target, but also counts the number of comparisons. It returns a returnArray which holds information regarding whether the target was actually found (index 0), the path the target was found at (index 1), and the number of comparisons the function performed to get to the target (index 3).