CMPT 435 - Fall 2021 - Dr. Labouseur

Assignment Four – LaTeX Graphs and Trees

Tyler Vultaggio

November 2021

1 Summary

Given a text file containing graph commands on each line, the program will assemble Adjacency Matrices, Adjacency Lists, and Linked Objects that represent each of these graphs, respectively. The program will then display the representation to the user in the form of text in the console. It will then perform Depth First Traversals and Breadth First Traversals on the linked object representations. All results will be output to the console.

The second part of the program will take in a given text file that contains a String on each line. Each String will sorted into a Binary Search Tree. 42 Strings will be selected at "random" and will be searched for by using the Binary Search Tree. All results will be output to console, including the number of comparisons made while searching.

2 Matrix

is a square matrix that is used to represent a finite graph in finite space. The elements represented in the matrix are pairs of vertices that are adjacent to each other in a given graph. Those pairs that are adjacent can be expressed by the integer "1". While those pairs that are not adjacent can be expressed by the integer "0". It is important to notice the properties of this graph representation. For example, an undirected graph can actual help save some space. You'll notice that a diagonal of "1"s will appear and you'll be able to cut your matrix in half, keeping on part of it. (Because you already have the identical half, you can simply expand this out if you need a clearer picture)

2.1 Matrix

```
public class Matrix
{
        private int LastVertex;
        private int matrix[][];

        public Matrix(int vertex)
        {
                LastVertex = vertex + 1;
                matrix = new int[LastVertex][LastVertex];
        }

        public void addEdge(int start, int end)
        {
                matrix[start][end] = 1;
                matrix[end][start] = 1;
        }

        public void printMatrix()
        {
                for(int i = 0; i < LastVertex; i++)
                {
                        for(int j = 0; j < LastVertex; j++)
                        {
                                System.out.print(matrix[i][j] + " ");
                        }
                        System.out.println();
                }
        }

        public static void main(String[] args)
        {
                // TODO Auto-generated method stub

        }
}
```

3 Adjacency List

is a collection of unordered lists that are used to represent a finite graph in finite space. Essentially, a single list will have a list in each cell which contains information describing that cell. Think of each cell as the vertex, and the list in that cell contains its edges. It best visualized as a hashmap, although the actual function of a hashmap here is not necessary.

3.1 Adjacency List

```
public class AdjacencyList
{

        private AdjacencyNode head = null;
```

```java

	public AdjacencyList()
	{

	}

	//adds to the head of the list
	public void addHead(int Source, int Destination)
	{
		AdjacencyNode next = head;
		head = new AdjacencyNode(Source, Destination);
		head.setNext(next);
	}

	public AdjacencyNode getHead()
	{
		return head;
	}

	public boolean isEmpty()
	{
		return head == null;
	}

	public void printList(AdjacencyNode edge)
	{
		System.out.println(edge.getStart() + " -> :");
		while(edge != null)
		{
			System.out.print(" " + edge.getEnd());
			edge = edge.getNext();
		}
	}

	public static void main(String[] args)
	{
		// TODO Auto-generated method stub

	}
}
```

4 Graphs

is a collection of vertices (nodes) that contain lists of edges, where each edge references its end vertex. Almost like a linked list, but not so linearly depicted. I like to think of it as Object Oriented vertices.

4.1 Graphs

```java
import java.util.ArrayList;

public class Graphs
{

        private ArrayList<Vertex> vertices;

        //This is the graph itself
        public Graphs()
        {
                vertices = new ArrayList<>();
        }

        //Adds a Vertex
        public void addVertex(int vertexID)
        {
                Vertex newVertex = new Vertex(vertexID);
                vertices.add(newVertex);
        }

        //Gets the vertices
        public ArrayList<Vertex> getVertices()
        {
                return vertices;
        }

        //Adds an edge to the graph
        public void addEdgetoGraph(int startId, int endId)
        {
        int startLoc = getVertexLocation(startId);
        int endLoc = getVertexLocation(endId);
        if( (startId != -1) && (endId != -1))
        {
            // If exists, lets add it
                vertices.get(startLoc).setEdge(vertices.get(endLoc));
                vertices.get(endLoc).setEdge(vertices.get(startLoc));
            }
            else
            {
                System.out.println("Vertex cannot be connected, make sure it exists");
            }
        }

        //Gets the location of the Vertices
        public int getVertexLocation(int someVertexIdToFind)
        {

                int location = 0;
                boolean found = false;
                while ( found == false && (location < vertices.size()) )
            {
                        if (someVertexIdToFind == vertices.get(location).getID())
                {
                                found = true;
                }
                        else
                {
                                location ++;
                }
            }
                return location;
```

```
         }

        public void GraphDetails()
        {
                for(int i = 0; i <vertices.size(); i++)
                {
                        vertices.get(i).printVertexInfo();
                }
        }

        public static void main(String[] args)
        {
                // TODO Auto-generated method stub

        }

}
```

5 Binary Search Tree

is a way to store items in memory, a really efficient way actually. A parent of the tree has at most 2 children. All items are in sorted order which makes look up times ridiculously fast. On average, look up time takes about Ø(log n), (which is the height of the tree, since a binary tree with n nodes has a minimum of log n levels it must take at least log n comparisons to find a particular node). Worst case look up time is Ø(n) and this is awful because it can basically degenerate to a linked list.

5.1 Binary Search Tree

```
import java.util.ArrayList;

public class BSTree
{

        private TreeNode root;
        private int count;
        private ArrayList<String> LeftsRights;

        public BSTree()
        {

        }

        public void setRoot(TreeNode root)
        {
                this.root = root;
        }

        public void setCount(int count)
        {
                this.count = count;
        }

        public TreeNode getRoot()
        {
                return root;
        }

        public int getCount()
        {
                return count;
        }

        public void addNode(String key)
        {
                TreeNode node = new TreeNode();
                node.magicItem = key;
                if(root == null)
                {
                        root = node;
                }
                else
                {
                        TreeNode nodeToLookAt = new TreeNode();
                        nodeToLookAt = root;
                        TreeNode parent;
            while(true)
            {
                parent = nodeToLookAt;
                if(key.compareTo(nodeToLookAt.getmagicItem()) < 0)
                {
                    nodeToLookAt = nodeToLookAt.getLeft();

                    if(nodeToLookAt == null)
                    {
                        parent.setLeft(node);
                        return;
                    }
                }
                else
                {
                    nodeToLookAt = nodeToLookAt.getRight();
                    if(nodeToLookAt == null)
                    {
                        parent.setRight(node);
                        return;
                    }
                }
            }
                }
        }

        public TreeNode findNode(String target)
        {

                TreeNode magicitem = root;
                count = 0;
                while(magicitem.getmagicItem() != target)
```

```
				{
					if(target.compareToIgnoreCase(magicitem.getmagicItem())<0)
					{
						magicitem = magicitem.getLeft();
						LeftsRights.add("L");
						count++;
					}
					else
					{
						magicitem = magicitem.getRight();
						LeftsRights.add("R");
						count++;
					}
				}
				System.out.print(LeftsRights);
				return magicitem;
		}

		public void TreeTraverseinOrder(TreeNode target)
		{
				if(target !=null)
				{
					TreeTraverseinOrder(target.getLeft());
					System.out.println(target);
					TreeTraverseinOrder(target.getRight());
				}
		}

		public static void main(String[] args)
		{
				// TODO Auto-generated method stub

		}

}
```

6 Depth First Traversals

– is an algorithm for traversing through a tree or a graph. It will go deep, not wide. This means that the algorithm will explore as far as possible along each branch before eventually backtracking. Worst case performance is Ø(Card(V) + Card(E)) and worst case space performance is Ø(Card(V)). The explanation of this is as follows: To touch each node just once, we see that it is Ø(Card(V)), to get to that node varies so any additional complexity comes from how that node is found. For example, if an edge leads you to some node that has been touched, we count the edge, not the node, and move on until we found the node that has not been touched. So we account for that with Ø(Card(E)). Using this, we can see that the best case is about Ø(1), as you may only have one vertex on a given graph. Note: that when implementing this, it is often good practice to use a stack to aid in process

6.1 Depth First Traversals

```java
import java.util.ArrayList;

public class DFS
{
	private ArrayList<Vertex> vertices;
	private MyStack stack = new MyStack();

	@SuppressWarnings("unchecked")
	public DFS(ArrayList<Vertex> vertices)
	{
		this.vertices = (ArrayList<Vertex>)vertices.clone();
		for(Vertex root : this.vertices)
		{
			if(!(root.getDFSisProcessed()))
			{
				DFStraverse(root);
			}
		}
	}

	public void DFStraverse(Vertex root)
	{
		stack.push(root);
		root.setDFSisProcessed(true);
		while(!stack.isEmpty())
		{
			Vertex vertex = stack.pop();
			System.out.print(vertex.getID() + " ");
			for(Vertex newRoot : vertex.getEdges())
			{
				if(!(newRoot.getDFSisProcessed()))
				{
					newRoot.setDFSisProcessed(true);
				}
				stack.push(newRoot);
			}
		}
	}

	public static void main(String[] args)
	{
		// TODO Auto-generated method stub

	}

}
```

7 Breadth First Traversals

is another algorithm for traversing through a tree or a graph. However, this one will go wide, not deep. This means that algorithm will explore the neighbors of nodes at the present depth prior to moving on to nodes at the next depth. The performance is the same as the Depth First Traversal, and so is the reasoning. However the difference is that we must use a queue, instead of stack when implementing this algorithm.

7.1 Breadth First Traversals

```java
import java.util.ArrayList;

public class BFS
{

	private MyQueue queue = new MyQueue();

	public BFS(ArrayList<Vertex> vertices)
	{
		for(Vertex v: vertices)
		{
			if(!(v.getBFSisProcessed()))
			{
				BFStraverse(v);
			}
		}
	}


	public void BFStraverse(Vertex root)
	{
		queue.enqueue(root);
		root.setBFSisProcessed(true);
		while(!queue.isEmpty())
		{
			Vertex vertex = queue.dequeue();
			System.out.print(vertex.getID() + " ");
			ArrayList <Vertex> neighbours = root.getEdges();
			for(int i = 0; i < neighbours.size(); i++)
			{
				Vertex n = neighbours.get(i);
				if(n != null && !n.getBFSisProcessed())
				{
					queue.enqueue(n);
					n.setBFSisProcessed(true);
				}
			}
		}
	}

	public static void main(String[] args)
	{
		// TODO Auto-generated method stub

	}

}
```

8 GraphandBtree

This class is my file reader and my format for printing in one. It both creates the set of graphs from the graph1.txt file and my Binary Search Tree from the magicitems.txt file.

8.1 GraphandBtree

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.InputMismatchException;
import java.util.Scanner;
import java.lang.Math;
public class GraphandBtree
{
        private static final String FILE_NAME = "graphs1.txt";
        private static final File myFile = new File(FILE_NAME);

        private static final String FILE_NAME_2 = "magicitems.txt";
        private static final File theFile = new File(FILE_NAME_2);

        private static final int TOTAL_LINES = 666;
        private static String[] magicItemsList = new String[TOTAL_LINES];

        private static void readFileVertexAndMakeGraph() throws FileNotFoundException
        {
            String[] regVertex;
            String[] regEdges;
            Scanner input = new Scanner(myFile);
            String currentLine;
            Graphs graph = new Graphs();
            GraphAL adj = new GraphAL();
            Matrix matrix = new Matrix(0);
            int numGraph = 0;
            while (input.hasNextLine())
            {
                currentLine = input.nextLine();
                if (currentLine.equals("new graph"))
                {
                        graph = new Graphs();
                    matrix = new Matrix(0);
                    adj = new GraphAL();
                    numGraph++;
                }
                else if (!(currentLine.equals("")) && (currentLine.charAt(0) != '-'))
                {
                    if (currentLine.charAt(4) == 'v')
                    {
                        // add vertex
                        regVertex = currentLine.split("add vertex ");
                        graph.addVertex(Integer.parseInt(regVertex[1]));
                        adj.addVertex();
                        matrix = new Matrix(Integer.parseInt(regVertex[1]));
                    }
                    else if (currentLine.charAt(4) == 'e')
                    {
                        // add edge
                        regEdges = currentLine.split("\\s+");
                        graph.addEdgetoGraph(Integer.parseInt(regEdges[2]), Integer.parseInt(r
                        adj.addEdge(Integer.parseInt(regEdges[2]), Integer.parseInt(regEdges[4
                        matrix.addEdge(Integer.parseInt(regEdges[2]), Integer.parseInt(regEdge
                    }
                }
                else if (currentLine.trim().isEmpty()) {
                    System.out.println("\n============================================");
                    System.out.println("Graph # " + numGraph);
                    System.out.println("============================================");
                    System.out.println("Adjacency Matrix");
                    System.out.println("--------------------------------------------");
                    matrix.printMatrix();
                    System.out.println("--------------------------------------------");
                    System.out.println("Adjacency List");
                    System.out.println("--------------------------------------------");
                    adj.printGraph(adj);
                    System.out.println("\n--------------------------------------------");
                    System.out.println("Linked Object");
                    System.out.println("--------------------------------------------");
                    graph.GraphDetails();
                    System.out.println("--------------------------------------------");
                    System.out.println("DFS");
                    System.out.println("--------------------------------------------");
                    DFS dfs = new DFS(graph.getVertices());
                    System.out.println("\n--------------------------------------------");
                    System.out.println("BFS");
                    System.out.println("--------------------------------------------");
                    BFS bfs = new BFS(graph.getVertices());
                    System.out.println("\n--------------------------------------------\n");
                }
            }
    }

        private static String[] randomItems()
        {
        String[] listOfRandomItems = new String[42];
        int numberOfElements = 42;
        for (int i = 0; i < numberOfElements; i++)
        {
            listOfRandomItems[i] = magicItemsList[(int)(Math.random() * magicItemsList.length)
        }
        return listOfRandomItems;
    }

        private static void readFileMagicItems() throws FileNotFoundException
        {
            int magicItemPosition = 0;
            String currentLine;
            Scanner input = new Scanner(theFile);
            while (input.hasNext())
            {
                currentLine = input.nextLine();
                currentLine = input.nextLine();
                currentLine = currentLine.replaceAll("[0-9+,_()/.]", "");
```

```
                currentLine = currentLine.replaceAll("\\s", "");
                currentLine = currentLine.replaceAll("'", "");
                currentLine = currentLine.replaceAll("-", "");
                currentLine = currentLine.toLowerCase();
                currentLine = currentLine.substring(0,1).toUpperCase() + currentLine.substring
                magicItemsList[magicItemPosition] = currentLine;
                magicItemPosition++;
            }
            input.close();
    }

    // Perform Binary Search Tree operations: get 42 random items and search for them while
    // calculating results
    private static void binarySearchTreeResults()
    {
        int avgcomp = 0;
        String[] list = randomItems();
        BSTree Tree = new BSTree();
        for (int i = 0; i < list.length; i++)
        {
                Tree.addNode(list[i]);
        }
        System.out.println("===============BST==================");
        System.out.println("You touched the following nodes, in that order: ");
        Tree.TreeTraverseinOrder(Tree.getRoot());
        System.out.println("----------------------------------");
        for (int j = 0; j < list.length; j++) {
            System.out.println("Search for " + list[j]);
            System.out.println("You found " + Tree.findNode(list[j]) +
                    " with " + Tree.getCount() + " comparisons");
            avgcomp = avgcomp + Tree.getCount();
        }
        System.out.println("----------------------------------");
        System.out.println("Average number of comparisons: " + avgcomp / list.length);
        System.out.println("==================================");
    }

    // Not a fan of how I made this class, but I don't have time to make it look any cleaner
    public static void main(String[] args) throws FileNotFoundException
    {
        readFileVertexAndMakeGraph();
        readFileMagicItems();
        binarySearchTreeResults();
    }


}
```