Assignment Four

Veronica Longley

November 19, 2021

Our Goal: In this assignment we will read a given file and create graphs based on the listed instructions. Once a graph is done being described and built, we will print the graphs adjacency list as well as it's matrix form. Then we will print the vertex ids in breadth first order as well as in depth first order. We will do this for each graph described. We will also look at Binary Search trees. We will construct a tree of magic item strings, print out the path to insert them into the tree, perform an in order traversal which will print the items in alphabetical order, then search for 42 items in the tree and look at the number of comparisons needed to locate each one. Lastly, we will look at the average number of comparisons for the 42 lookup items and establish why that is the case.

1 Vertex Class

```
public class Vertex {
	//Vertexes are made up of an id, a boolean is processed to
	//aid with traversals and a list of neighbors
	private int id;
	private boolean processed;
	public ArrayList<Vertex> neighbors;

	public Vertex()
	{
		id = 0;
		processed = false;
		neighbors = new ArrayList<Vertex>();
	}//Vertex

	public Vertex(int newid)
	{
		id = newid;
		processed = false;
		neighbors = new ArrayList<Vertex>();
	}//Vertex

	public ArrayList<Vertex> getNeighbors()
	{
		return neighbors;
	}//getNeighbors

	public void setNeighbors(ArrayList<Vertex> newNeighbors)
	{
		neighbors = newNeighbors;
	}//setNeighbors

	public boolean getprocessed()
	{
		return processed;
	}//getprocessed

	public void setprocessed(boolean newprocessed)
	{
		processed = newprocessed;
	}//setprocessed

	//getter for id
	public int getid()
```

```
        {
                return id;
        }//getid


        //setter for myData
        public void setid(int newid)
        {
                id = newid;
        }//setid

        //add vertex to other vertex's list of neighbors to create an edge
        public void addEdge(Vertex destination)
        {
                //System.out.println("... destination is vertex " + destination.id);
                this.neighbors.add(destination);
                //System.out.println("added destination vertex " + destination.id
                + "from Vertex " + this.id);
        }//add edge
}//Vertex
```

First we must look at the vertex class. The vertices are what make up the graph and allow edges to be formed. Vertices are made up of an id or integer value, a Boolean isProcessed to aid with traversals, and an array list of neighbors to the given vertex. In addition to constructors and getters and setters for each of the mentioned attributes, we also have a method named addEdge() which when called takes the vertex it is called on and adds the vertex in parenthesis to the list of neighbors of the first vertex. This must be called in main twice since the graph is undirected – if vertex a is neighbors with vertex b, then vertex b must be neighbors with vertex a. By adding these two as neighbors we are essentially creating an edge between them.

2 Graph Class

```
          public class Graph {

        private int numOfVertices;
        public ArrayList<Vertex> theVertices;

        public Graph()
        {
                theVertices = new ArrayList<>();
                numOfVertices = 0;

        }//Graph

        public void addVertex(int num)
        {
                Vertex newVertex = new Vertex(num);
                theVertices.add(newVertex);
                numOfVertices++;
        }//addVertex

        public int getnumOfVertices()
        {
                return numOfVertices;
```

```
        }//getnumOfVertices

        public void printAdjList(Graph graph)
        {
                for(int j = 0; j < graph.numOfVertices; j++)
                {
                        System.out.print(graph.theVertices.get(j).getid()+ " | ");
                        for(int q = 0; q < graph.theVertices.get(j).neighbors.size()
                        ; q++)
                        {
                                System.out.print(graph.theVertices.get(j).neighbors
                                .get(q).getid() + " ");
                        }//for
                        System.out.println();
                }//for

        }//printAdjList


        public void printMatrix(Graph graph)
        {
                boolean [][] matrix = new boolean [numOfVertices][numOfVertices];

                //initialize matrix to false
                for (int i = 0; i < numOfVertices; i++)
                {
                        for( int k= 0; k< numOfVertices; k++)
                                matrix[i][k] = false;
                }//for

                //change from false to true if in list of neighbors
                //if vertex id's start a 0 we do not have to adjust for the
                //matix positions
                if(graph.theVertices.get(0).getid() == 0)
                {
                        for (int h = 0; h < graph.theVertices.size() ; h++)
                                   for (int p = 0; p < graph.theVertices.get(h)
                                   .getNeighbors().size(); p++)
                                      matrix[h][graph.theVertices.get(h)
                                      .getNeighbors().get(p).getid()] = true;
                }//if
                else
                {
                        for (int h = 0; h < graph.theVertices.size() ; h++)
                           for (int p = 0; p < graph.theVertices.get(h)
                           .getNeighbors().size(); p++)
                              matrix[h][graph.theVertices.get(h).getNeighbors()
                              .get(p).getid()-1] = true;
                }//else

                //print out 1 for true and 0 for false
                for (int i = 0; i < graph.numOfVertices; i++)
                {
                        System.out.println( );
                   for (int j = 0; j < graph.numOfVertices; j++)
                   {
                           if(matrix[i][j] == true)
                                   System.out.print(" 1 ");
```

```
                                else
                                        System.out.print(" 0 ");
                        }//for
                }//for

         }//printGraph

        //depth first traversal
        public void depthFT(Vertex v)
        {
                if(v.getprocessed() == false)
                {
                        v.setprocessed(true);
                        System.out.print(v.getid() + " ");

                }// if
                for (int n = 0; n < v.neighbors.size(); n++)
                {
                        if(v.neighbors.get(n).getprocessed() == false)
                        {
                                //recursion
                                depthFT(v.neighbors.get(n));
                        }//if
                }//for
        }//DepthFT

        //breadth first traversal
        public void breadthFT(Vertex v)
        {
                Queue q = new Queue();
                q.enqueue(v);
                v.setprocessed(true);
                Vertex cv;
                while(!q.isEmpty())
                {
                        cv = q.dequeue();
                        System.out.print(cv.getid() + " ");
                        for(int n = 0; n < cv.neighbors.size(); n++)
                        {
                                if(cv.neighbors.get(n).getprocessed() == false)
                                {
                                        q.enqueue(cv.neighbors.get(n));
                                        cv.neighbors.get(n).setprocessed(true);
                                }//if
                        }//for
                }//while
        }//breadthFT

        //reset processed to be false again after one traversal has been done
        //so second traversal and start fresh
        public void reset(Vertex v)
        {
                if(v.getprocessed() == true)
                {
                        v.setprocessed(false);
                }// if

                for (int n = 0; n < v.neighbors.size(); n++)
```

```
                    {
                            if(v.neighbors.get(n).getprocessed() == true)
                            {
                                    reset(v.neighbors.get(n));
                            }//if
                    }//for
            }//reset

            //reset number of vertices and the vertices when we are moving to next graph
            public void resetGraph(Graph graph)
            {
                    graph.numOfVertices = 0;
                    graph.theVertices.clear();
            }//resetGraph
}//Graph
```

The graph class holds most of the methods to manipulate and traverse the graph. First we have addVertex() which creates a new vertex, adds it to the list of vertices and increments the counter for the number of vertices. Next we have a method to print the adjacency list for the graph. To do this we have a nested for loop where the first loop goes through the list of theVertices and for each vertex we print the list of neighbors for that vertex. We then print a blank line to properly align the list. Next we have printMatrix() which is responsible for constructing and printing the matrix form of the graph. To do this we initialize a two dimensional array of boolean's the size of the number of vertices for the graph. We then have a nested for loop to initialize the entire matrix to false. Next we have an if else statement that determines if the vertices start at zero (meaning we done have to adjust for a zero based list) or if they start at one (we will have to subtract one from the id value of the vertex to account for zero based arrays). Once this is determined, we have another nested for loop that goes through the list of vertices and for each we go through the list of neighbors and set the appropriate matrix block to be true where the vertex has a neighbor. Lastly, we have yet another nested for loop that goes through the entire matrix printing one if the cell is true and zero if the cell is false. As a result we have a matrix the size of the number of vertices in the graph where we can use zeros and ones to determine if the vertex is neighbors with the other vertices. Next is depthFT() which gets passed a vertex to start at and recursively calls itself until all the vertices in theVertices list have processed values that are now true. We start with the first vertex and go through its list of neighbors calling depthFT() on each of them until they are all processed and printing their id out as they are processed. In other words, depth first traversals will go as deep in the graph as possible before back tracking to the width of the graph. Conversely, we have breadth first traversals. This works by first processing all the one step away vertices before moving to the two steps away vertices and so on. This works by queuing the neighbors of the vertices and processing them in that order. Once the queue is empty all the vertices have been processed and printed and the traversal is complete. To aid with these two traversals we also have reset() which goes through the graph yet again undoing the traversal and setting all processed boolean's back to false. This will be called between depthFT() and breadthFT() to allow the second traversal to start fresh with all processed boolean's set to false. Lastly for the Graph Class we have resetGraph() which is called when the adjacency list, matrix form and both traversals have been completed and we are ready to move to the next graph. This sets the number of vertices to be zero and clears the list of theVertices so they are not added twice when the construction of the next graph begins. This completes everything needed for the graph portion of the assignment. The actual construction of the graphs and reading of the file will happen in main, so we will look at that later.

3 Binary Search Tree

```
public class BinarySearchTree
{

        //comparison counter
        public static float BSTComparisons;

        class Node
        {
                String key;
                Node left, right;

                public Node(String item)
                {
                        key = item;
                        left = right = null;
                }//Node
        }//Node

        Node root;

        BinarySearchTree()
        {
                root = null;
        }//BinarySearchTree

        public void insert (String key)
        {
                root = insertRecurssive( root,  key);
        }//insert

        public Node insertRecurssive(Node root, String key)
        {
                if (root == null)
                {
                        root = new Node(key);
                        return root;
                }//if

                if(key.compareToIgnoreCase(root.key)<0)
                {
                        root.left = insertRecurssive(root.left, key);
                        System.out.print("L");
                }//if
                else if(key.compareToIgnoreCase(root.key)>=0)
                {
                        root.right = insertRecurssive(root.right, key);
                        System.out.print("R");
                }//elseif
                return root;
        }//insertRecurssive
```

```

	public void inOrder()
	{
		inOrderRecurissive(root);
	}//inOrder

	public void inOrderRecurissive(Node root)
	{
		if(root != null)
		{
			inOrderRecurissive(root.left);
			System.out.println(root.key);
			inOrderRecurissive(root.right);
		}//inOrderRecurissive
	}//inOrderRecurissive

	public Node search( Node root, String key)
	{
		//if the tree is empty or the one we are looking at matches the one
		//we are searching for return the one we are looking at (root)
		if(root == null || root.key.compareToIgnoreCase(key) == 0)
		{
			BSTComparisons++;
			return root;
		}//if

		//if key comes after the one we are looking at go the right
		if (root.key.compareToIgnoreCase(key)< 0)
		{
			BSTComparisons++;
			System.out.print("R");
			return search(root.right, key);
		}//if

		//if key comes before the one we are looking at go the left
		else
		{
			BSTComparisons++;
			System.out.print("L");
			return search(root.left, key);
		}//else
	}//search

	public float countComps()
	{
		return BSTComparisons;
	}//countComps

	public void resetComps()
	{
		BSTComparisons = 0;
	}//resetComps
}//BinarySearchTree
```

A Binary Search Tree is made up of Nodes of magic items. When constructing the tree, all magic items alphabetically after the root node are found branched to the right while all magic items alphabetically before the root node are found to the left. With every right or left movement we have a similar construction using each new node as the 'root' and moving left or right depending on the alphabetical order. To insert into the tree we have a recursive method as well as a non recursive method. The non recursive method is what we call in main which then calls the recursive method. When inserting an item we send the item as a string then call the recursive method sending it the root/node we left off at as well as the item were adding. In the recursive method, if the root is null we know the tree is empty and add the item to be the root. If the item were adding is less than the root we print an "L" to signify we are moving to the left and call the recursive insert function on the node to the left of the root, otherwise we do the same on the right if the item were adding comes after the root or is the same word as the root. This is done over and over again until we find the proper null space where we can add the item. We do this for each of the 666 magic items in the file. To print the items in alphabetical order we have inOrder() and inOrderRecurissive(). InOrder() is called in main which then calls inOrderRecurissive() on all the nodes to the left of the root and prints them in order, prints the root node, then calls inOrderRecurissive() on all the nodes to the right of the root node and prints them. This results in all the items printed in alphabetical order. Next we have search(). This is a recursive function that first determines if the root is null (the tree is empty) or if the root is equal to the item we are looking for. For both cases we return the root node to main and increment the global counter to establish the number of comparisons for a binary search tree. If the root/node we are looking for comes before the item we are looking for we increment comparisons, print out an "R" to signify we are moving right and recursively call search on the node to the right of the one we were looking at. We do the same thing to the left if the item we are searching for comes before the root node/the one we are looking at. This is done until the node we send equals the item we are looking for and we can print it along with the number of comparisons used to find it. Lastly, we have two helper methods to aid with counting comparisons. CountComps() is called in main and returns the global variable BSTComparisons. ResetComps() sets BSTComparisons back to zero to start fresh with the next search count.

4 ...In Main Method

```
public class main
{
        public static float totalComparisons;
        public static Graph theGraph = new Graph();

        public static void main(String[] args)
        {
                //Read instructions .txt file
                String fileName = "graphs1.txt";
                ArrayList<String> commands = new ArrayList<>();
                try {
                Scanner readFile = new Scanner (new File (fileName));

                while (readFile.hasNextLine())
                   {
                                commands.add(readFile.nextLine());

                   }//while
                }//try
```

```
        catch(FileNotFoundException ex)
                {
                System.out.println("Failed to find file: "+ fileName);
                }//catch
        catch(InputMismatchException ex)
                {
                System.out.println("Type mismatch. ");
                System.out.println(ex.getMessage());
                }//catch
        catch(NumberFormatException ex)
                {
                System.out.println("Failed to convert String into integer");
                System.out.println(ex.getMessage());
                }//catch
        catch(NullPointerException ex)
                {
                System.out.println("Null pointer exception. ");
                System.out.println(ex.getMessage());
                }//catch
        catch(Exception ex)
                {
                System.out.println("Oops, something went wrong. ");
                ex.printStackTrace();
                }//catch

                //call makeGraph and send it the ArrayList of instructions
                //from the .txt file
                makeGraph(commands);


                //For the Binary Search Tree
                //Read MagicItems
                String file2Name = "";
                final int NUMOFITEMS = 666;
                String [] magicItems = new String [NUMOFITEMS];
                String theitem = null;
                file2Name = "magicitems.txt";

                //input item names from file and store in array
                try
                        {
                        Scanner readFile = new Scanner (new File (file2Name));
                        int g = 0;
                        while(readFile.hasNextLine())
                        {
                                theitem = readFile.nextLine();
                                magicItems[g] = theitem;
                                g++;
                        }//while
                        readFile.close();
                        }//try
                catch(FileNotFoundException ex)
                        {
                        System.out.println("Failed to find file: "+ file2Name);
                        }//catch
                catch(InputMismatchException ex)
                        {
```

```
				System.out.println("Type mismatch. ");
				System.out.println(ex.getMessage());
				}//catch
		catch(NumberFormatException ex)
				{
				System.out.println("Failed to convert String into integer. ");
				System.out.println(ex.getMessage());
				}//catch
		catch(NullPointerException ex)
				{
				System.out.println("Null pointer exception. ");
				System.out.println(ex.getMessage());
				}//catch
		catch(Exception ex)
				{
				System.out.println("Oops, something went wrong. ");
				ex.printStackTrace();
				}//catch

		//fill the Binary Search Tree
		BinarySearchTree tree = new BinarySearchTree();
		for(int k = 0; k < NUMOFITEMS; k++)
		{
				System.out.print(magicItems[k] + ": ");
				tree.insert(magicItems[k]);
				System.out.println();
		}//for

		System.out.println();
		//Traverse it in order to get an alphabetized list of magic items
		System.out.println("In-order Traversal:");
		tree.inOrder();
		System.out.println();
		System.out.println("Searches and Comparisons:");

		//read the file with 42 items to search for and store in a new array
		final int NUMOFSEARCHES = 42;
		String file3Name = "magicitems-find-in-bst.txt";
		String [] itemsToFind = new String [NUMOFSEARCHES];
		//input item names from file and store in array
		try
				{
				Scanner readFile = new Scanner (new File (file3Name));
				int r = 0;
				while(readFile.hasNextLine())
				{
						itemsToFind[r] = readFile.nextLine();
						r++;
				}//while
				readFile.close();
				}//try
		catch(FileNotFoundException ex)
				{
				System.out.println("Failed to find file: "+ file2Name);
				}//catch
		catch(InputMismatchException ex)
				{
				System.out.println("Type mismatch. ");
```

```
            System.out.println(ex.getMessage());
            }//catch
        catch(NumberFormatException ex)
            {
            System.out.println("Failed to convert String into integer.");
            System.out.println(ex.getMessage());
            }//catch
        catch(NullPointerException ex)
            {
            System.out.println("Null pointer exception. ");
            System.out.println(ex.getMessage());
            }//catch
        catch(Exception ex)
            {
            System.out.println("Oops, something went wrong. ");
            ex.printStackTrace();
            }//catch

        //loop through list of items to search for
        //search for each one
        //print it out along with its path and number of comparisons needed
        //to find it
        for(int j = 0; j < NUMOFSEARCHES; j++)
        {
            System.out.print(itemsToFind[j] + " -- ");

            tree.search(tree.root, itemsToFind[j]);

            totalComparisons += tree.countComps();
            System.out.print(" -- " );
            System.out.println(tree.countComps());
            tree.resetComps();
        }//for
        System.out.println();
        //Print the average number of comparisons for a Binary Search Tree
        System.out.println("The average number of comparisons "
        + totalComparisons / NUMOFSEARCHES);

      }//main

    public static void makeGraph(ArrayList<String> list)
    {
        //loop through each line of the array
        for (int i = 0; i<list.size(); i++)
        {
            //if line contains "undirected" we know a graph is being
            //announced, so we can
            //print out that line as it acts as a title for the graph
            if(list.get(i).contains("undirected"))
            {
                System.out.println(list.get(i));
            }//if

            //if line contains "new graph" we know we want to make a new
            //graph and not add to the last
            else if(list.get(i).contains("new graph"))
            {
                Graph theGraph = new Graph();
```

```
				}

				//if line contains "vertex" we want to add a new vertex to
				//the graph
				//We locate the last space in the line and take the number
				//from the string
				//and convert it to an integer, so it can be added to the
				//graph of integers
				else if (list.get(i).contains("vertex"))
				{
					int val = Integer.parseInt(list.get(i).substring(
					list.get(i).lastIndexOf(" ") + 1 ));
					//System.out.println("Adding Vertex");
					theGraph.addVertex(val);
				}//else if

				//if line contains edge we are linking two vertices
				//start finds the last index of the letter e and the index
				//of "-" and converts the number between it to an integer
				//end takes the number after the "-" and converts it
				//to an integer
				else if (list.get(i).contains("edge"))
				{
					int start = Integer.parseInt(list.get(i).substring(
					list.get(i).lastIndexOf("e") + 2,
					list.get(i).indexOf("-") -1 ));
					int end = Integer.parseInt(list.get(i).substring(
					list.get(i).indexOf("-") + 2 ));

					//System.out.println("Adding edge" + start+ " - "
					//+ end);

					if(theGraph.theVertices.get(0).getid() == 0)
					{
						theGraph.theVertices.get(start).addEdge(
						theGraph.theVertices.get(end));
						theGraph.theVertices.get(end).addEdge(
						theGraph.theVertices.get(start));
					}//if
					else
					{
						//System.out.println("Adding edge to vertex"
						//+ theVertices.get(start -1 ).getid());
						theGraph.theVertices.get(start - 1 )
						.addEdge(theGraph.theVertices.get(end - 1));
						theGraph.theVertices.get(end - 1 )
						.addEdge(theGraph.theVertices.get(start -1));

					}//else
				}//else if

				//if the line is empty we know the graph is done being
				//described and we can go about
				//making and printing the matrix and the adjacency list
				//as well as doing two traversals
				else if (list.get(i).equals("") )
				{
					System.out.println();
```

```
                                        System.out.println("The Adjacency List: ");
                                        theGraph.printAdjList(theGraph);
                                        System.out.println();
                                        System.out.println("The Matrix: ");
                                        theGraph.printMatrix(theGraph);
                                        System.out.println();
                                        System.out.println();
                                        System.out.println("Depth First Traversal ");
                                        theGraph.depthFT(theGraph.theVertices.get(0));
                                        System.out.println();
                                        //System.out.println(theGraph.theVertices.get(2)
                                        //.getprocessed());
                                        theGraph.reset(theGraph.theVertices.get(0));
                                        //System.out.println(theGraph.theVertices.get(2)
                                        //.getprocessed());
                                        System.out.println();
                                        System.out.println("Breadth First Traversal ");
                                        theGraph.breadthFT(theGraph.theVertices.get(0));
                                        System.out.println();
                                        theGraph.resetGraph(theGraph);
                                        System.out.println();
                                        System.out.println();
                                        System.out.println();

                              }//elseif
                              else
                              {
                              }//else
                    }//for
          }//makeGraph

}//main
```

We bring all of these classes together in Main. Main begins by reading the command file for creating the graph and storing it into an Array List. We then call makeGraph() to construct the graph and print the appropriate attributes. Skipping down to makeGraph() we see we loop through the command list and if lines contain certain words we do certain tasks. First, if the line contains "undirected" we know a graph is bring announced, so we print it out to label the graphs attributes that will later be printed. Next, if the line contains "new graph" we know we have to construct a new graph, so we do that. If the line contains "vertex" we know we are adding a vertex to the graph, so we locate the id in the line and convert it to an int then add it to the graph. If the line contains "edge" we know we are adding an edge, so we first locate the start and end values of the edge, then if it is the first edge being added we can do so without adjusting with minus one, otherwise we must subtract one from the start and end values before calling add edge. Lastly, if the line is empty we know we are done describing a graph, so we can call methods to print the adjacency list, the matrix form, complete the depth first traversal and breadth first traversal, and lastly reset the entire graph to prepare to make another graph if needed. This completes our exploration of an undirected graph. Moving back to the binary search tree, we start by reading the list of 666 magic items into an array. We then populate the array with those items using the insert() method we discussed earlier. Next we call the inOrder() method from earlier to print the magic items in alphabetical order. Lastly, we read in the text file that contains 42 magic items that we want to look up and count the comparisons of. We store these in another array. We then call search() from the Binary Search Tree class on each of these items, print the number of comparisons needed to find them from countComps(), add this to the totalComparisons counter, and reset the countComps() counter in the Binary Search Tree class. To finish we divide the totalComparisons by the number of searches, 42, to find the average look up time for a binary search tree.

5 Conclusion

In terms of complexity for the two graph traversals, we can see that they are both

$$\text{O(number of edges + number of vertices)}$$

This is because we of course must reach each vertex, and in order to do so we must travel across each edge to ensure that each vertex is being processed. As a result we have a running time for both depth first traversals and breadth first traversals of O(e + v).

As for Binary Search tree look ups, we have a complexity of

$$\text{O(log base 2 (n))}$$

This is the asymptomatic running time for both look ups and insertion. This is because when we move right or left we are cutting out half of the tree. On average we will have a complexity of log base 2 of n as sometimes the item will be in the first few levels of the tree and sometimes it will be deeper. Log base 2 of 666 is about 9.3, so for us getting 10.6 is in the appropriate realm of comparisons.