Algorithms Assignment 3

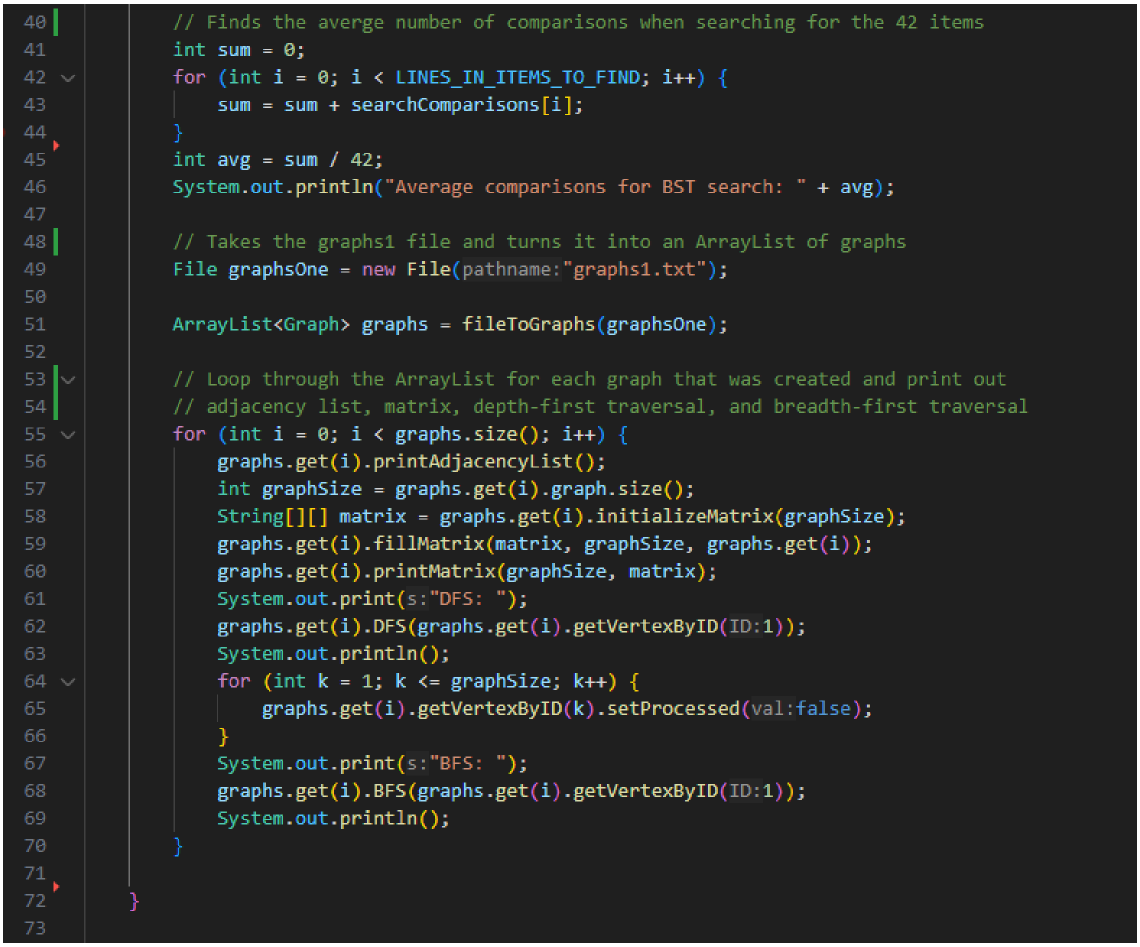
Thomas Breaton

November 2023

1 Main Function

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class Main3 {

    private static final int LINES_IN_MAGIC_ITEMS = 666;
    private static final int LINES_IN_ITEMS_TO_FIND = 42;

    Run | Debug
    public static void main(String[] args) throws FileNotFoundException {
        // Turn magicItemsFile into array (same as last two Asignments)
        File magicItemsFile = new File(pathname:"magicitems.txt");
        String[] magicItemsArray = fileToArray(magicItemsFile);

        // Implements the bst class and creates a new tree
        BST_class bst = new BST_class();
        String path = "";
        // Inserts each of the items of magicItems into a binary tree and keeps track of
        // their path
        for (int i = 0; i < LINES_IN_MAGIC_ITEMS; i++) {
            bst.insert(magicItemsArray[i], path);
        }
        // Prints out the tree in order
        bst.inOrder();

        // Turns the items to find file into an array
        File magicItemsToFindFile = new File(pathname:"magicitems-find-in-bst.txt");
        String[] magicItemsToFindArray = fileToArray(magicItemsToFindFile);

        int[] searchComparisons = new int[42];

        // Loops through all 42 items and finds all of them within the binary tree
        for (int i = 0; i < LINES_IN_ITEMS_TO_FIND; i++) {
            searchComparisons = bst.search(magicItemsToFindArray[i], path, searchComparisons, i);
        }

```

Figure 1: Image of main function



```java
        // Finds the averge number of comparisons when searching for the 42 items
        int sum = 0;
        for (int i = 0; i < LINES_IN_ITEMS_TO_FIND; i++) {
            sum = sum + searchComparisons[i];
        }
        int avg = sum / 42;
        System.out.println("Average comparisons for BST search: " + avg);

        // Takes the graphs1 file and turns it into an ArrayList of graphs
        File graphsOne = new File(pathname:"graphs1.txt");

        ArrayList<Graph> graphs = fileToGraphs(graphsOne);

        // Loop through the ArrayList for each graph that was created and print out
        // adjacency list, matrix, depth-first traversal, and breadth-first traversal
        for (int i = 0; i < graphs.size(); i++) {
            graphs.get(i).printAdjacencyList();
            int graphSize = graphs.get(i).graph.size();
            String[][] matrix = graphs.get(i).initializeMatrix(graphSize);
            graphs.get(i).fillMatrix(matrix, graphSize, graphs.get(i));
            graphs.get(i).printMatrix(graphSize, matrix);
            System.out.print(s:"DFS: ");
            graphs.get(i).DFS(graphs.get(i).getVertexByID(ID:1));
            System.out.println();
            for (int k = 1; k <= graphSize; k++) {
                graphs.get(i).getVertexByID(k).setProcessed(val:false);
            }
            System.out.print(s:"BFS: ");
            graphs.get(i).BFS(graphs.get(i).getVertexByID(ID:1));
            System.out.println();
        }

    }
```

Figure 2: Image of main function

This section of code consists of what is being imported, constants, and the code that I want to run. The main function in this assignment calls for files to be scanned, binary trees to be made and searched, and graphs to be made and traversed.

2 Reading File Into Array

```java
    public static String[] fileToArray(File file) throws FileNotFoundException {
        int iteration = 1;

        Scanner scan = new Scanner(file);

        // Creates the array to store the magic itmes in
        String[] itemsArray = new String[666];
        String[] itemsToFindArray = new String[42];
        int i = 0;

        // Goes through the whole text file and adds each line to the array
        while (scan.hasNextLine()) {
            String data = scan.nextLine();
            if (iteration == 1) {
                itemsArray[i] = data;
            } else if (iteration == 2) {
                itemsToFindArray[i] = data;
            }

            i++;
        }
        scan.close();
        iteration++;
        return itemsArray;
    }

```

Figure 3: Image of fileToArray function

This section of code is the same from Assignment 1 and 2. Here we are taking the magicItems file and going line by line putting each item into an array. Once there are no more lines remaining in the file, the function returns the full array.

3 Reading a File Into Graphs

```java
    public static ArrayList<Graph> fileToGraphs(File file) throws FileNotFoundException {
        Scanner scan = new Scanner(file);

        // Creates a new graph and an Arraylist to store the graph in
        Graph g = new Graph();
        ArrayList<Graph> graphArrayList = new ArrayList<Graph>();

        // Scans each line looking for a specific string
        while (scan.hasNextLine()) {
            String data = scan.nextLine();
            String delims = "[ ]+";
            String[] tempString = data.split(delims);
            // There is a new line so the graph is done being created, it can be added to
            // the ArrayList
            if (tempString[0].compareTo(anotherString:"") == 0) {
                graphArrayList.add(g);
                // Creates the new graph
            } else if (tempString[0].compareTo(anotherString:"new") == 0) {
                g = new Graph();
            } else if (tempString[0].compareTo(anotherString:"--") == 0) {
                // Adds the vertexes to the graph
            } else if (tempString[1].compareTo(anotherString:"vertex") == 0) {
                g.addVertex(Integer.parseInt(tempString[2]));
                // Adds the edges to the graph
            } else if (tempString[1].compareTo(anotherString:"edge") == 0) {
                g.addEdge(Integer.parseInt(tempString[2]), Integer.parseInt(tempString[4]));
            }
        }
        // Adds the final graph from the file to the ArrayList
        graphArrayList.add(g);
        scan.close();
        return graphArrayList;
    }

}
```

Figure 4: Image of fileToGraphs function

This section of code takes a file and reads line by line looking for the pattern of "add vertex," "add edge," and blank lines. With these three lines we are able to create a graph and then add those graphs into an ArrayList of graphs so they can be easily managed.

4 Binary Search Tree Class

```java
// Binary search tree class
class BST_class {
    class Node {
        String key;
        Node left, right;
        String path;

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

    // Root node for BST
    Node root;

    // BST constructor, makes empty tree
    BST_class() {
        root = null;
    }

    // Insert a node by calling the recursive insert function
    void insert(String key, String path) {
        path = "";
        root = BST_Insert(root, key, path);
    }

    Node BST_Insert(Node root, String key, String path) {
        // When tree is empty
        if (root == null) {
            root = new Node(key);
            System.out.println(key + ": path - " + path);
            return root;
        }

        if (key.compareToIgnoreCase(root.key) < 0) {
            path = path + "L,";
            root.left = BST_Insert(root.left, key, path);
        } else {
```

Figure 5: Image of BST class

```java
        if (key.compareToIgnoreCase(root.key) < 0) {
            path = path + "L,";
            root.left = BST_Insert(root.left, key, path);
        } else {
            path = path + "R,";
            root.right = BST_Insert(root.right, key, path);
        }
        // Return the pointer
        return root;
    }

    void inOrder() {
        inOrderTraversal(root);
    }

    // Traverse the tree recursively
    void inOrderTraversal(Node root) {
        if (root != null) {
            inOrderTraversal(root.left);
            // System.out.print(root.key + ", ");
            inOrderTraversal(root.right);
        }
    }

    // Call the search functin to recursively search the binary tree
    int[] search(String key, String path, int[] numComparisons, int placeInArray) {
        // Needed this to be a place holder for the real root
        Node rootToChange;
        int comparisons = 0;
        path = "";
        rootToChange = search_BST(root, key, path, comparisons, numComparisons, placeInArray);
        return numComparisons;
    }

    Node search_BST(Node root, String key, String path, int comparisons, int[] numComparisons, int placeInArray)
        // Base or if values are the same
        if (root == null || root.key.compareTo(key) == 0) {
            System.out.println(key + ": Path - " + path + ": Comparisons - " + comparisons);
            numComparisons[placeInArray] = comparisons;
            return root;
```

Figure 6: Image of BST class

This section of code creates the BST or binary search tree full of the information from magicItemsFile. The BST class has 3 major components, the insert function, the in order traversal function, and the search function. With these 3 functions we are able to create a tree with the objects we want, we can print out every object to know what we have, and we can search for objects within the tree. The running time for searching within the BST is O(n), however, since we are running through an additional loop our running time would be O(n^2).

5 Graph Class

```java

// Graph class
class Graph {
    ArrayList<GraphNode> graph;
    int v;

    // Vertex class
    class GraphNode {

        private int vertexID = 0;
        private boolean processed;
        private ArrayList<Integer> neighbors;

        public GraphNode(int newID) {
            vertexID = newID;
            processed = false;
            neighbors = new ArrayList<Integer>();
        }

        public void setProcessed(boolean val) {
            processed = val;
        }
    }

    // Initializes a new graph
    Graph() {
        graph = new ArrayList<GraphNode>();
    }

    // Allows to add a vertex with a given ID
    void addVertex(int vertexID) {
        GraphNode vertex = new GraphNode(vertexID);
        addToGraph(vertex);
    }

    // Adds a vertex to the graph
    void addToGraph(GraphNode newVertex) {
        graph.add(newVertex);
    }

    // Loops through all the vertexes in the graph untli it finds the one that
```

Figure 7: Image of Graph class

```
    GraphNode getVertexByID(int ID) {
        for (int i = 0; i < graph.size(); i++) {
            if (graph.get(i).vertexID == ID) {
                return graph.get(i);
            }
        }
        return null;
    }

    GraphNode getByID(int ID) {
        GraphNode result = getVertexByID(ID);
        return result;
    }

    // Adds an edge to the graph through the neighbors ArrayList
    void addEdge(int source, int dest) {
        GraphNode tempSource = getVertexByID(source);
        GraphNode tempDest = getVertexByID(dest);

        tempSource.neighbors.add(dest);
        tempDest.neighbors.add(source);
    }

    // Prints out the adjacency list
    void printAdjacencyList() {
        for (int i = 0; i < graph.size(); i++) {
            GraphNode tempVertex = graph.get(i);
            System.out.print("Vertex " + tempVertex.vertexID + " has neighbors: ");
            for (int k = 0; k < tempVertex.neighbors.size(); k++) {
                System.out.print(tempVertex.neighbors.get(k) + ", ");
            }
            System.out.println();
        }
    }

    // Creates the matrix
    String[][] initializeMatrix(int graphSize) {
        String[][] matrix = new String[graphSize + 1][graphSize + 1];
        for (int i = 0; i <= graphSize; i++) {
            matrix[i][0] = Integer.toString(i);
            if (i == 0) {
```

Figure 8: Image of Graph class

This section of code is dealing with everything regarding out graph class. This is how we initialize a graph, create a subclass for vertexes, I called it GraphNode, get vertexes based on ID, make edges, create the linked objects, create a matrix of the vertexes and edges, and create adjacency lists. On top of that, this is how we use our depth first traversal and breadth first traversal. The running time for our depth first traversal function is O(n), but our running time for our breadth first traversal function is O(V + E).

```java
                    matrix[i][k] = Integer.toString(k);
                }
            } else {
                for (int k = 1; k <= graphSize; k++) {
                    matrix[i][k] = ".";
                }
            }
        }
        return matrix;
    }

    // Fills the matrix with 1s where there is an edge
    void fillMatrix(String[][] matrix, int graphSize, Graph g) {
        int neighbor;
        GraphNode tempVertex;
        for (int i = 0; i <= graphSize; i++) {
            if (g.getVertexByID(i) != null) {
                tempVertex = g.getVertexByID(i);
            } else {
                i++;
                tempVertex = g.getVertexByID(i);
            }
            for (int k = 0; k <= graphSize; k++) {
                for (int j = 0; j < tempVertex.neighbors.size(); j++) {
                    neighbor = tempVertex.neighbors.get(j);
                    if (neighbor == k) {
                        matrix[i][k] = "1";
                    }
                }
            }
        }
    }

    // Prints out matrix
    void printMatrix(int graphSize, String[][] matrix) {
        for (int i = 0; i <= graphSize; i++) {
            for (int k = 0; k <= graphSize; k++) {
                System.out.print(matrix[i][k] + " ");
            }
            System.out.println();
        }
```

Figure 9: Image of Graph class

6 Results

```java
                System.out.print(matrix[i][k] + " ");
            }
            System.out.println();
        }
    }

    // Depth-first traversal
    void DFS(GraphNode fromVertex) {
        if (!fromVertex.processed) {
            System.out.print(fromVertex.vertexID + ", ");
            fromVertex.processed = true;
        }
        for (int i = 0; i < fromVertex.neighbors.size(); i++) {
            GraphNode neighborVertex = getVertexByID(fromVertex.neighbors.get(i));
            if (!neighborVertex.processed) {
                DFS(neighborVertex);
            }
        }
    }

    // Breadth-first traversal
    void BFS(GraphNode fromVertex) {
        Queue<GraphNode> q = new LinkedList<>();
        q.add(fromVertex);
        fromVertex.processed = true;
        while (!q.isEmpty()) {
            GraphNode currentVertex = q.remove();
            System.out.print(currentVertex.vertexID + ", ");
            for (int i = 0; i < currentVertex.neighbors.size(); i++) {
                GraphNode neighborVertex = getVertexByID(currentVertex.neighbors.get(i));
                if (!neighborVertex.processed) {
                    q.add(neighborVertex);
                    neighborVertex.processed = true;
                }
            }
        }
    }
}
```

Figure 10: Image of Graph class

```
Cloak of Blackshadows: path - L,R,L,R,L,R,L,L,L,L,L,R,L,L,
Lightning Totem: path - R,L,L,L,R,R,R,R,L,R,L,
The Scalp of Kung: path - R,R,L,R,R,R,L,L,R,
Was It A Rat I Saw: path - R,R,R,L,R,R,L,L,
Cloak of displacement, minor: path - L,R,L,R,L,R,L,L,L,L,L,R,R,
Incense of meditation: path - R,L,L,L,L,R,R,L,R,L,L,L,L,
Great Axe: path - L,R,R,R,L,L,L,L,R,R,R,
Pearl of power, 7th-level spell: path - R,L,L,R,R,R,R,R,R,L,
Borgir's Sugar Cubes: path - L,R,L,L,R,R,R,L,L,R,L,R,R,
Glaive, Dragon Slayer +1/+5: path - L,R,R,L,L,R,L,
Twig of a Dozen Uses or The Handy Stick: path - R,R,R,L,L,L,L,L,R,L,L,R,
Snail Liquid: path - R,L,R,L,R,R,L,L,L,L,
```

Figure 11: Image of Results of inserting items into the BST

These are some of the results that I got after running the code. You can see when inserting items into the BST, each item's path is displayed. When searching for an item in the BST the path is also displayed as well as the number of comparisons needed. Lastly, you can see the adjacency list, matrix, depth first traveral, and breadth first traverasl for one of the graphs that we had to create.

```
Cane of Evocation: Path - L,R,L,R,L,L,L,R,R,L,: Comparisons - 10
Scimitar: Path - R,L,R,L,R,L,R,L,R,R,R,: Comparisons - 11
Ring of Fiery Assistance: Path - R,L,R,L,L,R,L,R,L,R,: Comparisons - 10
Statuette of Nightveil: Path - R,L,R,L,R,R,R,L,: Comparisons - 8
Cloak of the Undead: Path - L,R,L,R,L,R,L,L,L,R,L,R,R,: Comparisons - 13
Short Bow: Path - R,L,R,L,R,L,R,L,R,R,R,: Comparisons - 11
Sack of Plunder: Path - R,L,R,L,R,L,R,L,R,R,R,: Comparisons - 11
Self-Loading Bow: Path - R,L,R,L,R,L,R,L,R,R,R,: Comparisons - 11
Potion of the Hero's Heart: Path - R,L,R,L,L,R,L,R,L,L,L,L,: Comparisons - 12
Link Tabbard: Path - R,L,L,L,R,R,R,R,L,R,: Comparisons - 10
Eyes of doom: Path - L,R,L,R,L,R,R,L,R,L,R,L,R,R,R,: Comparisons - 15
Average comparisons for BST search: 9
```

Figure 12: Image of Results of searching for items in the BST

```
Vertex 1 has neighbors: 2, 5, 6,
Vertex 2 has neighbors: 1, 3, 5, 6,
Vertex 3 has neighbors: 2, 4,
Vertex 4 has neighbors: 3, 5,
Vertex 5 has neighbors: 1, 2, 4, 6, 7,
Vertex 6 has neighbors: 1, 2, 5, 7,
Vertex 7 has neighbors: 5, 6,
0 1 2 3 4 5 6 7
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 .
DFS: 1, 2, 3, 4, 5, 6, 7,
BFS: 1, 2, 5, 6, 3, 4, 7,
```

Figure 13: Image of Results from graph functions