# Assignment Four

Nicholas Fiore

Nicholas.Fiore@Marist.edu

November 17, 2022

## 1 Graphs

### 1.1 Graphing

Graphs are a data structure with varying uses. Its topology is useful for representations of networks, maps, or similar diagrams. Additionally, they can be useful for various data sets which have elements with multiple dependencies.

Graphs are akin to linked list and other similar data structures in the way that they function with node-like objects that are then linked to other node-like objects. The largest difference that separates it from lists and trees is the way that these nodes, called vertices in graphs, can connect to each other. With lists, there is a single connection from the parent and a single connection to the child. Trees are similar, but can have multiple "child" vertices. In graphs, however, vertices are not limited to the child and parent relationship. A vertex from any part of the graph can connect to any other vertex within the graph, regardless of what vertices it might already be attached to. This can create the more 3D structure of a graph, including things like loops which aren't possible on a tree (otherwise I wouldn't be a tree, it would be a graph).

Graphing in this assignment was implemented in a simple way. A $Vertex$ class was created, holding an integer id, a boolean to check if it was processed, and an ArrayList of Vertex objects to represent neighboring vertices. For simplicity, $Vertex$ objects that hold each other in their *neighbors* table are considered to have an edge between them. A *Graph* class was also created to represent a physical graph, composed of many Vertex objects. The *Graph* object holds an ArrayList of all the $Vertex$ objects within the graph and a name. There is also a String 2D array to be used for matrix printing, which will be touched on later. There are a few methods to this that allow it to function, including $addVertex()$ and $addEdge()$, but also things such as $printMatrix()$ and $printAdjacencyList()$, which print matrix and adjacency list representations of the graph, respectively.

### 1.2 Breadth-First and Depth-First Searching/Traversal

Linked lists and trees can be pretty easily traversed linearly, as there is only a forward and sometimes backwards to move. Graphs, however, are more complicated, as they can contain loops that make linearly searching impossible due to infinite looping potential. Therefore, there needs to be a way to keep track of

which vertices in the graph have already been considered. Breadth-first and depth-first searching are the two ways that these are done.

The ideas of breadth-first (BFS) and depth-first searching (DFS) are sort of opposite ways to achieve the same concept, which is to search or traverse a graph (or a tree, which these methods can also be implemented for, because a tree is technically a graph). Both BFS and DFS utilize the *processed* boolean in the $Vertex$ objects to keep track of which ones in the graph have already been processed.

### 1.2.1 Breadth-First

Breadth-first works based on the idea of finding and printing out all the adjacent vertices to the initial vertex, and then finding all the adjacent vertices of those vertices and printing the ones ***not yet processed***, repeating this process until all of the vertices in the graph have been processed. This is done by adding each vertex found as an adjacency to a Queue. As adjacencies are found, they are added to the Queue. Once no more adjacencies are found for the current node, the vertex at the front of the Queue is dequeued and printed, and its adjacencies are found, adding them to the Queue so long as they have not been processed already.

Breadth-first searching has an asymptomatic run-time that is a bit different from traditional Big-Oh classifications. Its runtime is considered to be $O(|V| + |E|)$, $V$ being the total number of vertices in the graph and $E$ being the total number of edges. This is due to the fact that every vertex and every edge will only ever be checked once, due to the nature of the processing. Simplifying this into more formal Big-Oh classes would make this a $O(n)$.

### 1.2.2 Depth-First

Depth-first searching takes the idea of breadth-first but works in the opposite direction. Instead of looking for the close adjacencies, it moves towards finding the furthest it can go by taking a single path and never repeating vertices. This is handled through a recursive function. It takes in a vertex and then prints that vertex and sets its *processed* boolean to *true*. Then it enters a for loop that iterates through all the neighbors of the current vertex. If the neighbor vertex it currently is looking at is not currently processed (*processed* is set to false), it will call the $depthFirstSearch()$ function with that neighbor vertex, repeating the process. This will happen recursively until every vertex that is connected in that loop or line is printed. As the recursive calls collapse again, any neighboring vertices that are not part of the branch or loop that had been printed already will be entered then, completely printing that branch/loop before collapsing again, until all connected vertices to the original starting vertex has been printed.

Like breadth-first searching, depth-first searching has a run-time of $O(|V| + |E|)$. This if for the same reason that it is in BFS: every edge and every vertex is checked only once. Simplified, this again gives the algorithm a run-time of $O(n)$.

## 1.3 Binary Trees

Trees are a sort of data structure that are similar to linked lists, but instead of only having one next value, they can have multiple. The number of "children" that a Node in a tree can have depends on the type of tree. In the case of binary trees, any Node can have 0, 1, or 2 children. Any more and the tree is no longer binary. This type of tree is binary because traversing the tree will always present two options to go, if there is anything else in either direction (the lack of an option is technically not a lack of choice but rather a forced choice). When putting a list of items into a binary tree, the general distribution of the items in the list will create something close to a balanced tree, which is a tree in which the left and right subtree of any particular tree has a height difference of no more than one level. This includes all subtrees of said original tree.

Due to the nature of the binary tree, searching the tree becomes very similar to the binary search algorithm used with normal arrays or lists. The algorithm itself looks similar to a linear search, however, each time a

Node is compared to the key, the algorithm will choose a branch to take. With a properly balanced binary tree, taking a branch will cause roughly half of the other options in the tree to be eliminated. This happens every single time, halving continuously until the item is found or an end of the tree is reached. For this reason, with a balanced binary tree, the asymptomatic run-time for a binary search tree is the same as a binary search, standing at $O(log(n))$. One of the benefits of a BST over normal binary search is that a BST does not need a sorted list to function (and actually prefers it not to be sorted), and therefore does not require the overhead of a sorting function.

As a caveat, however, poorly balanced binary tree can cause the run-time to degrade. The worst case scenario for this would be entering a sorted list into the binary tree. Since it is already sorted, the tree will always choose one option when placing the next element, causing a single, very long branch that always only has one child. In this worst case, the run-time of the algorithm degrades to $O(n)$ as the long branch is little more than a glorified linked list.

# 2 Appendix

## 2.1 Vertex.java

```java
import java.util.ArrayList;

public class Vertex {
    /* Data Fields */
    private int id;
    private boolean processed = false;
    private ArrayList<Vertex> neighbors = new ArrayList<Vertex>();

    /* Constuctors */
    public Vertex() {
        //default empty constructor
    }

    //constuctor with ID parameter
    public Vertex(int idNum) {
        this.id = idNum;
        this.processed = false;
    }

    /* Accessors/Mutators */
    public int getId() {
        return this.id;
    }

    public Vertex getNeighbor(int index) {
        return neighbors.get(index);
    }

    public int getNeighborSize() {
        return neighbors.size();
    }

    public boolean wasProcessed() {
        return this.processed;
    }

    public void setId(int newId) {
        this.id = newId;
    }

    public void setProcessed(boolean bool) {
        this.processed = bool;
    }

    /* Functions */
    //adds an adjacent vertex as a neighbor to the neighbors ArrayList
    public void addNeighbor(Vertex neighbor) {
        this.neighbors.add(neighbor);
    }
}
```

## 2.2 Graph.java

```java
import java.util.*;

public class Graph {
    /* Data Fields */
    private String name = "";
    private ArrayList<Vertex> verticies = new ArrayList<Vertex>();
```

```
7       private String[][] matrix;
8
9       /* Constructors */
10      //Default, empty constuctor
11      public Graph() {
12
13      }
14
15      //constuctor with name parameter
16      public Graph(String newName) {
17          this.name = newName;
18      }
19
20      /* Accesors/Mutators */
21      public String getName() {
22          return this.name;
23      }
24
25      public ArrayList<Vertex> getVerticies() {
26          return verticies;
27      }
28
29      public void setName(String newName) {
30          this.name = newName;
31      }
32
33      /* Functions */
34      //Adds a vertex to the verticies ArrayList
35      public void addVertex(Vertex vert) {
36          this.verticies.add(vert);
37      }
38      //Adds an edge between two verticies by adding each vertex to the other's list of neighbors
39      public void addEdge(Vertex vert1, Vertex vert2) {
40          vert1.addNeighbor(vert2);
41          vert2.addNeighbor(vert1);
42      }
43
44      //Performs a depth-first search of the graph
45      public void depthFirstSearch(Vertex vert) {
46          if (!vert.wasProcessed()) {
47              System.out.print(vert.getId() + "␣");
48              vert.setProcessed(true);
49          }
50          for (int i = 0; i < vert.getNeighborSize(); i++) {
51              Vertex neighbor = vert.getNeighbor(i);
52              if (!neighbor.wasProcessed()) {
53                  depthFirstSearch(neighbor);
54              }
55          }
56      }
57
58      //Performs a breadth-first search of the graph
59      public void breadthFirstSearch(Vertex vert) {
60          System.out.print("Breadth-First␣Traversal:␣");
61          VertQueue searchQueue = new VertQueue();
62          searchQueue.enqueue(new VertNode(vert));
63          vert.setProcessed(true);
64          while (!searchQueue.isEmpty()) {
65              Vertex currVert = searchQueue.dequeue().getMyVertex();
66              System.out.print(currVert.getId() + "␣");
67              for (int i = 0; i < currVert.getNeighborSize(); i++) {
68                  Vertex neighbor = currVert.getNeighbor(i);
69                  if (!neighbor.wasProcessed()) {
70                      searchQueue.enqueue(new VertNode(neighbor));
71                      neighbor.setProcessed(true);
```

```
72                  }
73              }
74          }
75          System.out.println();
76      }
77
78      //resets the "processed" status on all verticies in a graph.
79      public void resetProcessing() {
80          for (int i = 0; i < verticies.size(); i++) {
81              verticies.get(i).setProcessed(false);
82          }
83      }
84
85      //Prints the maxtrix representation of the graph
86      public void printMatrix() {
87          int size = verticies.size();
88          int i, j;
89          matrix = new String[size + 1][size + 1];
90          matrix[0][0] = "␣";
91          //creates the "labels" of the matrix
92          for (i = 0; i < matrix.length - 1; i++) {
93              matrix[i + 1][0] = verticies.get(i).getId() + "";
94              matrix[0][i + 1] = verticies.get(i).getId() + "";
95          }
96
97          //initializes the matrix to have no adjacencies, marked with a period
98          for (i = 1; i < matrix.length; i++) {
99              for (j = 1; j < matrix[0].length; j++) {
100                 matrix[i][j] = ".";
101             }
102         }
103
104         //adds all adjacencies to the matrix, marked by a 1
105         for (i = 0; i < verticies.size(); i++) {
106             Vertex vert = verticies.get(i);
107             for (j = 0; j < vert.getNeighborSize(); j++) {
108                 Vertex neighbor = vert.getNeighbor(j);
109                 int vertIndex = Search.linearSearchReturnIndex(verticies, vert.getId());
110                 int neighborIndex = Search.linearSearchReturnIndex(verticies, neighbor.getId());
111
112                 matrix[vertIndex + 1][neighborIndex + 1] = "1";
113                 matrix[neighborIndex + 1][vertIndex + 1] = "1";
114             }
115         }
116
117         //prints the matrix
118         for (i = 0; i < matrix.length; i++) {
119             for (j = 0; j < matrix[0].length; j++) {
120                 System.out.print(matrix[i][j] + "␣");
121
122             }
123             System.out.println();
124         }
125
126         System.out.println();
127     }
128
129     public void printAdjacencyList() {
130         System.out.println("Adjacency␣List");
131         for (int i = 0; i < verticies.size(); i++) {
132             Vertex vert = verticies.get(i);
133             System.out.print("[" + vert.getId() +"]␣");
134             for (int j = 0; j < vert.getNeighborSize(); j++) {
135                 System.out.print(vert.getNeighbor(j).getId() + "␣");
136             }
```

```
            System.out.println();
        }
        System.out.println();
    }
}
```

Graph and Vertex also utilize slightly modified Queue and Node class (VertQueue and VertNode), however the modifications are only to the datatype so I didn't bother adding them here.

## 2.3 TreeNode.java

```
/*
 * The Node class is used to created linked lists of objects in order to more variably control the
      size of said list without being locked into an array
 * This is a modified version of the Node class to be used to construct binary trees.
 */
public class TreeNode {
    /* Data Fields */
    private String myString = "";
    private TreeNode myParent = null;
    private TreeNode myLeft = null;
    private TreeNode myRight = null;

    /* Constructors */
    //No-arg (default) constructor for creating a default Node object
    public TreeNode() {}

    //Partial constructor for initializing only the Item within the Node
    public TreeNode(String str) {
        myString = str;
    }

    //Full constuctor for creating a Node object and assigning an character name and next Node
        object pointer for left and right
    public TreeNode(String str, TreeNode left, TreeNode right) {
        myString = str;
        myLeft = left;
        myRight = right;
    }

    /* Accessors and Mutators */
    //Returns the character String of the Node object
    public String getMyString() {
        return myString;
    }

    public TreeNode getMyParent() {
        return myParent;
    }

    //Returns the pointer for the next Node object to the left
    public TreeNode getMyLeft() {
        return myLeft;
    }

    //Returns the pointer for the next Node object to the right
    public TreeNode getMyRight() {
        return myRight;
    }

    //Changes the value of the character String of a Node object to a new String
    public void setMyString(String myString) {
        this.myString = myString;
```

```
51      }
52
53      public void setMyParent(TreeNode myParent) {
54          this.myParent = myParent;
55      }
56
57      //Changes the pointer of the left pointer of a Node object to a new pointer to a node Object (
            or NULL)
58      public void setMyLeft(TreeNode myLeft) {
59          this.myLeft = myLeft;
60      }
61
62      //Changes the pointer of the right pointer of a Node object to a new pointer to a node Object
            (or NULL)
63      public void setMyRight(TreeNode myRight) {
64          this.myRight = myRight;
65      }
66
67      /* Functions */
68      //Checks to see if the Node object has a myLeft or myRight value != null. Returns boolean
            depending on result.
69      public boolean hasLeft() {
70          if (this.getMyLeft() != null)
71              return true;
72          else
73              return false;
74      }
75
76      public boolean hasRight() {
77          if (this.getMyRight() != null)
78              return true;
79          else
80              return false;
81      }
82
83  }
```

## 2.4 Tree.java

```
1   public class Tree {
2       /* Data Field */
3       private TreeNode myRoot;
4
5       /* Constructors */
6       //empty default constructor
7       public Tree() {
8
9       }
10      //constructor for starting with the root node
11      public Tree(TreeNode root) {
12          myRoot = root;
13      }
14
15      /* Accessor/Mutator */
16      public TreeNode getMyRoot() {
17          return this.myRoot;
18      }
19
20      public void setMyRoot(TreeNode myRoot) {
21          this.myRoot = myRoot;
22      }
23
24      /* Functions */
```

```
25      public void insert(TreeNode newNode) {
26          TreeNode trailing = null;
27          TreeNode current = this.myRoot;
28          System.out.print("Inserting " + newNode.getMyString() + ": ");
29          while (current != null) {
30              trailing = current;
31              if ((newNode.getMyString().toUpperCase()).compareTo(current.getMyString().toUpperCase
                    ()) < 0) {
32                  current = current.getMyLeft();
33                  System.out.print("L");
34              }
35              else {//must be >=
36                  current = current.getMyRight();
37                  System.out.print("R");
38              }
39              System.out.print(" ");
40          }
41          newNode.setMyParent(trailing);
42          if (trailing != null) {
43              if ((newNode.getMyString().toUpperCase()).compareTo(trailing.getMyString().toUpperCase
                    ()) < 0) {
44                  trailing.setMyLeft(newNode);
45              }
46              else { // >=
47                  trailing.setMyRight(newNode);
48              }
49          }
50          else {
51              this.setMyRoot(newNode);
52              System.out.print("Root.");
53          }
54      System.out.println();
55      }
56  }
```

## 2.5 Search.java

Mostly just the addition of $searchBinaryTree()$.

```
1   import java.util.ArrayList;
2   /*
3    * A class used to maintain static methods for searching algorithms (namely linear search and
         binary search). Linear search is iterative,
4    * while binary search is recursive.
5    */
6   public class Search {
7
8       //sequentially searches the list for the key, returning the index of where it was found. If it
             was not found, returns -1.
9       public static int linearSearch(String[] arr, String key, int[] counter) {
10          int i = 0;
11          while (i < arr.length && arr[i] != key) {
12              i++;
13              counter[0]++;
14          }
15          if (i >= arr.length)
16              i = -1;
17          return i;
18      }
19
20      //sequentially searches the list for the key, returning the vertex with that value. Version
            for ArrayLists.
21      public static Vertex linearSearchReturnVertex(ArrayList<Vertex> list, int key) {
```

```
22          int i = 0;
23          while (i < list.size() && list.get(i).getId() != key) {
24              i++;
25          }
26          if (i >= list.size())
27              i = -1;
28          return list.get(i);
29      }
30
31      //sequentially searches the list for the key, returning the index of where it was found. If it
            was not found, returns -1. version for ArrayLists
32      public static int linearSearchReturnIndex(ArrayList<Vertex> list, int key) {
33          int i = 0;
34          while (i < list.size() && list.get(i).getId() != key) {
35              i++;
36          }
37          if (i >= list.size())
38              i = -1;
39          return i;
40      }
41
42      //recursively searches the list by comparing the key to the item at the middle of the list,
            then choosing half of the array to
43      //then search depending on whether the key is lesser or greater than the element at the middle
            . If the element is found, returns
44      //the index, otherwise returns -1.
45      public static int binarySearch(String[] arr, int startIndex, int endIndex, String key, int[]
            counter) {
46          int retVal;
47          int midIndex = (endIndex + startIndex) / 2;
48          if (startIndex > endIndex) {
49              retVal = -1;
50          } else if (key.toUpperCase().compareTo(arr[midIndex].toUpperCase()) == 0) {
51              counter[0]++;
52              retVal = midIndex;
53          } else if (key.toUpperCase().compareTo(arr[midIndex].toUpperCase()) < 0) {
54              counter[0]++;
55              retVal = binarySearch(arr, startIndex, midIndex - 1, key, counter);
56          } else {
57              counter[0]++;
58              retVal = binarySearch(arr, midIndex + 1, endIndex, key, counter);
59          }
60          return retVal;
61      }
62
63      public static int binarySearchIt(String[] arr, int startIndex, int endIndex, String key, int[]
             counter) {
64          int low = startIndex;
65          int high = (endIndex) - startIndex;
66          while (low < high) {
67              int mid = (low + high) / 2;
68              if (key.compareTo(arr[mid]) <= 0) {
69                  high = mid;
70              } else {
71                  low = mid + 1;
72              }
73              counter[0]++;
74          }
75          return high;
76      }
77
78      //searches a binary tree based on a list of keys to compare. Functions very similarly to
            binary search due to the nature of a binary tree.
79      public static boolean searchBinaryTree(Tree bst, String key, int[] comparisons) {
80          boolean retVal = false;
```

```
        TreeNode root = bst.getMyRoot();
        TreeNode current = root;
        comparisons[0] = 0;
        System.out.print("Searching for " + key + ":");
        while(current != null && (current.getMyString().toUpperCase()).compareTo(key.toUpperCase()
            ) != 0) {
            if ((current.getMyString().toUpperCase()).compareTo(key.toUpperCase()) > 0) {
                current = current.getMyLeft();
                System.out.print(" L");
                comparisons[0]++;
            } else {
                current = current.getMyRight();
                System.out.print(" R");
                comparisons[0]++;
            }
            System.out.print(",");
        }
        if (current != null && (current.getMyString().toUpperCase()).compareTo(key.toUpperCase())
            == 0) {
            retVal = true;
            System.out.print(" found. ");
        } else {
            System.out.print(" not found. ");
        }
        System.out.print("Comparisons made: " + comparisons[0] + "\n");
        return retVal;
    }
}
```

## 2.6 MainFour.java

```
//Utility imports
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;


/*
 * The purpose of this program is to explore both graphs and the ways to traverse them, as well as
      create a binary tree to be used for
 * the purpose of searching, similar to a binary search.
 */

public class MainFour {
    public static void main(String[] args) throws IOException {
        //variable to store comparisons
        int[] compCounter = new int[1];

        createGraph("graphs1.txt");


        //array to hold all the items in the magic items list
        String[] itemList = null;

        itemList = fileToArray("magicitems.txt", itemList);

        // //array to hold the list of items that will be used to search the binary tree
        String[] searchList = null;

        searchList = fileToArray("magicitems-find-in-bst.txt", searchList);

        Tree binarySearchTree = createBinaryTree(itemList);
```

```
32
33          System.out.print("In-Order Traversal: ");
34          inOrderTraversal(binarySearchTree.getMyRoot());
35          System.out.println();
36          System.out.println();
37
38          int average = searchTreeFromList(searchList, binarySearchTree, compCounter);
39
40          System.out.println("Average # of comparisons: " + average);
41
42      }
43
44      //takes a file name and a list and puts each line of the file into a String in an array
45      public static String[] fileToArray(String fileName, String[] list) throws IOException {
46          long totalLines = 0;
47          File file = new File(fileName);
48          BufferedReader input = null;
49          try {
50              //gets the path of the current file in order to get the # of lines
51              Path path = Paths.get(file.getName());
52
53              input = new BufferedReader(new FileReader(fileName));
54
55              totalLines = Files.lines(path).count();
56
57              list = new String[(int)totalLines];
58
59              for(int i = 0; i < totalLines; i++) {
60                  list[i] = input.readLine();
61              }
62
63          } catch(FileNotFoundException ex) {
64              System.out.println("Failed to find file: " + file.getAbsolutePath());
65          } catch(IOException ex) {
66              System.out.println(ex);
67          } catch(Exception ex) {
68              System.out.println("Something went wrong.");
69              System.out.println(ex.getMessage());
70              ex.printStackTrace();
71          } finally {
72              if (input != null) {
73                  input.close();
74              }
75          }
76
77          return list;
78      }
79
80      //takes in an array of Strings and adds the entirety of the array to a Tree, before returning
            that Tree object.
81      public static Tree createBinaryTree(String[] list) {
82          Tree tree = new Tree();
83          for (int i = 0; i < list.length; i++) {
84              tree.insert(new TreeNode(list[i]));
85          }
86          System.out.println();
87          return tree;
88      }
89
90      //calls the searchBinaryTree function for every String in the tree, then computes the average
            and returns it.
91      public static int searchTreeFromList(String[] list, Tree bst, int[] comparisons) {
92          int total = 0;
93          int average;
94          for (int i = 0; i < list.length; i++) {
```

```
95              Search.searchBinaryTree(bst, list[i], comparisons);
96              total += comparisons[0];
97          }
98          average = total / list.length;
99          return average;
100     }
101
102     //searches a tree using in-order traversal
103     public static void inOrderTraversal(TreeNode node) {
104         if(node == null)
105             return;
106
107         inOrderTraversal(node.getMyLeft());
108
109         System.out.print("[" + node.getMyString() + "]␣");
110
111         inOrderTraversal(node.getMyRight());
112     }
113
114
115     //Creates graphs from instructions given in a file
116     public static int createGraph(String fileName) throws IOException {
117         long totalLines = 0;
118         File file = new File(fileName);
119         BufferedReader input = null;
120         String line;
121         String nextLine;
122         try {
123             //gets the path of the current file in order to get the # of lines
124             Path path = Paths.get(file.getName());
125
126             input = new BufferedReader(new FileReader(fileName));
127
128             totalLines = Files.lines(path).count();
129
130             //instantiate variables for graph and vertex objects. There are two Vertex objects in
                    the case of adding edges.
131             Graph graph = null;
132             Vertex vert1 = null;
133             Vertex vert2 = null;
134
135             nextLine = input.readLine();
136
137             //Command parsing. Goes through each line and determines what command is being used
                    based on strings.
138             for (int i = 0; i < totalLines; i++) {
139                 line = nextLine;
140                 nextLine = input.readLine();
141                 if (line.split("␣")[0].compareTo("--") == 0) {
142                     //do nothing, ignore this line as it is a comment
143                 } else if (line.split("␣")[0].compareTo("new") == 0) { //new graph
144                     //create a new graph object held by the graph variable
145                     graph = new Graph();
146
147                 } else if (line.split("␣")[0].compareTo("add") == 0) { //enters add vertex/add
                        edge tree
148                     if (line.split("␣")[1].compareTo("vertex") == 0) { //add vertex x
149                         //adds a new vertex to the graph, parsing the ID from the line given
150                         vert1 = new Vertex(Integer.parseInt(line.split("␣")[2]));
151                         graph.addVertex(vert1);
152                     } else if (line.split("␣")[1].compareTo("edge") == 0) { //add edge x - y
153                         //adds a new edge to the graph, based on the IDs parsed from the line
154                             //the verticies are found by searching the graph's verticies ArrayList
                                    for a Vertex that matches the ID given in the line at both
                                    positions
```

```
155                         vert1 = Search.linearSearchReturnVertex(graph.getVerticies(), Integer.
                                parseInt(line.split("␣")[2]));
156                         vert2 = Search.linearSearchReturnVertex(graph.getVerticies(), Integer.
                                parseInt(line.split("␣")[4]));
157
158                         graph.addEdge(vert1, vert2); //the verticies are now added as neighbors,
                                forming an adjacency
159                     }
160                 }
161                 //final check to see if the next line is either empty or does not exist
162                 if (nextLine == null || nextLine.isEmpty()) { //empty space; commands for this
                        graph are done, begin processing
163                     graph.printMatrix();
164                     graph.printAdjacencyList();
165
166                     System.out.print("Depth-First␣Traversal:␣");
167                     graph.depthFirstSearch(graph.getVerticies().get(0));
168                     graph.resetProcessing();
169                     System.out.println();
170
171                     graph.breadthFirstSearch(graph.getVerticies().get(0));
172                     graph.resetProcessing();
173                 }
174             }
175
176         } catch(FileNotFoundException ex) {
177             System.out.println("Failed␣to␣find␣file:␣" + file.getAbsolutePath());
178         } catch(IOException ex) {
179             System.out.println(ex.getMessage());
180         } catch(NullPointerException ex) {
181             System.out.println(ex.getMessage());
182         } catch(Exception ex) {
183             System.out.println("Something␣went␣wrong.");
184             System.out.println(ex.getMessage());
185             ex.printStackTrace();
186         } finally {
187             if (input != null) {
188                 input.close();
189             }
190         }
191
192         return (int)totalLines;
193     }
194 }
```