Assignment Four – Graphs & Binary Search Trees

C. Marcus DiMarco
C.DiMarco1@Marist.edu

November 19, 2021

1 Objectives

- Create a program that reads a text file in order to create graphs of various numbers of vertices and edges.
- Print the matrix, adjacency list, depth-first traversal and breadth-first traversal for each of these graphs.
- Populate a binary search tree with a set of 666 Strings, printing the path to each String from the root as it is populated.
- Print an in-order traversal of the binary search tree.
- Search for 42 distinct items in the tree, printing the path to each and recording the number of comparisons for each search as well as the average of all searches.

2 Conditions

- The program must create the relevant graph Objects by parsing the commands and inputs of the file.
- The program must distinguish based on the format of the file when a graph has been fully constructed.

- The program must only use custom algorithms for the binary search tree. Any of these which already exist in the language may not be used.

3 Overview - Graphs

Our graphing is driven by two classes: MarcusVertex and MarcusGraphs.

3.1 MarcusVertex

This custom vertex class is the building block of the graphs. Neighboring vertices are stored in an ArrayList. For increased user-friendliness, it includes a method hasNeighbor() to return true if two vertices are neighbors, which greatly streamlines the calls to printMatrix().

```java
/**
 * A custom implementation of a vertex object for
 * representing graphs as linked objects.
 */
import java.util.ArrayList;

public class MarcusVertex {
  private int id;
  private boolean isProcessed;
  private ArrayList<MarcusVertex> neighbors;
  private MarcusVertex next;

  public MarcusVertex(int id) {
    this.id = id;
    this.isProcessed = false;
    this.neighbors = new ArrayList<MarcusVertex>();
    this.next = null;
  }

  public boolean hasNeighbor(MarcusVertex neighbor) {
    for (int i = 0; i < neighbors.size(); i++) {
      if (neighbors.get(i).getId() == neighbor.getId()) {
        return true;
      }
    }

    return false;
  }

  // Setters and getters for private fields
  public int getId() {
    return this.id;
```

```java
    }

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

    public boolean getIsProcessed() {
      return isProcessed;
    }

    public void setIsProcessed(boolean isProcessed) {
      this.isProcessed = isProcessed;
    }

    public void addNeighbor(MarcusVertex neighbor) {
      this.neighbors.add(neighbor);
    }

    public ArrayList<MarcusVertex> getNeighbors() {
      return this.neighbors;
    }

    public void printNeighbors() {
      for (MarcusVertex currentVertex : neighbors) {
        System.out.print(currentVertex.getId() + " ");
      }
      System.out.print("\n");
    }

    public void setNext(MarcusVertex next) {
      this.next = next;
    }

    public MarcusVertex getNext() {
      return this.next;
    }
}
```

3.2 MarcusGraphs

Using MarcusVertex, we can assemble MarcusGraphs - a class which defines the various print and traversal methods needed. For user-friendliness, MarcusGraphs includes a method getVertexById(), which is instrumental in the creation of the edges from the text file.

```java
/**
 * A custom implementation of graphs as an object containing
 * vertices and edges. Supports matrices, adjacency lists,
 * and linked objects, as well as both depth-first traversals
 * and breadth-first traversals.
 */
import java.util.ArrayList;

public class MarcusGraphs {
  private ArrayList<MarcusVertex> vertices;
  private MarcusVertex initialVertex;

  // Default constructor
  public MarcusGraphs() {
    this.vertices = new ArrayList<MarcusVertex>();
    this.initialVertex = null;
  }

  // Prints a matrix of all vertices, printing a 1 at the intersection
  // if there is an edge present and printing a . if not
  public void printMatrix() {
    for (int i = -1; i < vertices.size(); i++) {
      for (int j = -1; j < vertices.size(); j++) {
        if (i == -1 && j == -1) {
          // Top left corner is blank space
          System.out.printf("%3s", "");
        } else if (i == -1) {
          // Top row is vertex IDs
          System.out.printf("%3s", vertices.get(j).getId() + " ");
        } else if (j == -1) {
          // First column is vertex IDs
          System.out.printf("%3s", vertices.get(i).getId() + " ");
        } else if (vertices.get(i).hasNeighbor(vertices.get(j))) {
          // If the vertices are neighbords, print 1
          System.out.printf("%3s", "1 ");
        } else {
          // If not neighbors, print .
          System.out.printf("%3s", ". ");
        }
      }
      // New line
      System.out.print("\n\n");
    }
  }

  // Prints each vertex followed by its neighbors
```

```java
public void printAdjacencyList() {
  for (int i = 0; i < vertices.size(); i++) {
    System.out.print("[" + vertices.get(i).getId() + "] ");
    vertices.get(i).printNeighbors();
  }
  System.out.print("\n");
}

// Traverses a graph vertex-by-vertex, going as deep as possible from
// the source before moving on to the next vertex. Prints IDs as
// encountered.
public void depthFirstTraversal(MarcusVertex source) {

  if (!source.getIsProcessed()) {
    System.out.print(source.getId() + " ");
    source.setIsProcessed(true);
  }
  for (MarcusVertex currentVertex : source.getNeighbors()) {
    if (!currentVertex.getIsProcessed()) {
      depthFirstTraversal(currentVertex);
    }
  }
}

// Traverses a graph using a queue. Prints IDs as dequeued.
public void breadthFirstTraversal(MarcusVertex source) {
  MarcusVertex currentVertex;

  // Reset booleans from depth-first traversal
  this.resetBooleans();

  // Enqueue when encountered
  MarcusQueue queue = new MarcusQueue();
  queue.enqueue(source);
  source.setIsProcessed(true);
  while (!queue.isEmpty()) {
    currentVertex = queue.dequeue();
    System.out.print(currentVertex.getId() + " ");
    for (MarcusVertex each : currentVertex.getNeighbors()) {
      if (!each.getIsProcessed()) {
        queue.enqueue(each);
        each.setIsProcessed(true);
      }
    }
  }

```

```
    System.out.print("\n\n");
  }

  // Reset isProcessed for each vertex in the graph
  public void resetBooleans() {
    for (MarcusVertex currentVertex : vertices) {
      currentVertex.setIsProcessed(false);
    }
  }

  // Add vertex to ArrayList and set initialVertex if needed
  public void addVertex(MarcusVertex vertex) {
    this.vertices.add(vertex);
    if (this.initialVertex == null) {
      this.initialVertex = vertex;
    }
  }

  public MarcusVertex getVertexById(int vertexId) {
    MarcusVertex returnVertex = null;

    for (MarcusVertex currentVertex : vertices) {
      if (currentVertex.getId() == vertexId) {
        returnVertex = currentVertex;
        break;
      }
    }

    return returnVertex;
  }

  public MarcusVertex getInitialVertex() {
    return initialVertex;
  }
}
```

4 Overview - Binary Search Tree

Our binary search tree refines our existing MarcusNode class and defines a new MarcusBST class.

4.1 MarcusNode

MarcusNode shifted from having a private field "next" to having two private fields, "leftChild" and "rightChild". Included also is a private field "parent".

```java
/**
 * Custom Node class which will serve as the container for
 * individual characters from the strings to be tested.
 * Must point to another Node object or null.
 *
 * Update, Assignment4: Instead of @next, points to up to
 * two children and up to one parent.
 */
public class MarcusNode {
  private String item;
  private MarcusNode leftChild;
  private MarcusNode rightChild;
  private MarcusNode parent;

  // Not having a default constructor forces String input
  public MarcusNode(String item) {
    this.item = item;
    this.leftChild = null;
    this.rightChild = null;
    this.parent = null;
  }

  public MarcusNode(String item, MarcusNode parent) {
    this.item = item;
    this.parent = parent;
  }
```

4.2 MarcusBST

MarcusBST is the hub in which all binary search tree operations exist. It includes methods to insert nodes into the tree, search for a target within the tree, print an in-order traversal, track the path followed and count comparisons.

```java
  // insertNode finds the correct space in the tree for the node,
  // then sets parent/child relationships for the nodes involved
  public void insertNode(MarcusNode node) {
    MarcusNode currentNode = root;
    MarcusNode trailingNode = null;

    this.resetPath();

    // Find the correct space in the tree
    while (currentNode != null) {
      trailingNode = currentNode;
```

```java
      if (!this.path.equals("")) {
        this.path += ", ";
      }
      if (node.getItem().compareToIgnoreCase(currentNode.getItem()) < 0) {
        this.path += "L";
        currentNode = currentNode.getLeftChild();
      } else {
        this.path += "R";
        currentNode = currentNode.getRightChild();
      }
    }

    // Set parent/child relationships
    if (trailingNode == null) {
      this.root = node;
    } else {
      node.setParent(trailingNode);
      if (node.getItem().compareToIgnoreCase(trailingNode.getItem()) < 0) {
        trailingNode.setLeftChild(node);
      } else {
        trailingNode.setRightChild(node);
      }
    }
  }

  // Public-facing abstraction for proper counter and path tracking
  public void search(String target) {

    // Reset counter and path
    this.resetCounter();
    this.resetPath();

    // Execute private recursive method
    search(this.getRoot(), target);
  }


  // search recursively iterates through the BST to find the target
  // in log(n) time, counting comparisons and printing the path
  private String search(MarcusNode root, String target) {

    if (root == null) {
      return "Target not found.";
    } else if (root.getItem().compareTo(target) == 0) {
      counter++;
      return target;
```

```java
    } else {
      counter++;
      if (this.path != null) {
        this.path += ", ";
      }
      if (target.compareToIgnoreCase(root.getItem()) < 0) {
        this.path += "L";
        return search(root.getLeftChild(), target);
      } else {
        this.path += "R";
        return search(root.getRightChild(), target);
      }
    }
  }

  public void inOrderTraversal(MarcusNode node) {
    if (node == null) {
      return;
    }

    inOrderTraversal(node.getLeftChild());
    System.out.println(node.getItem());
    inOrderTraversal(node.getRightChild());
  }
```

5 Assignment4

With the classes we have just defined, we are ready to create our program. We'll need some imported libraries: namely, java.io.File to import a file, java.io.FileNotFoundException to account for errors finding the input file, and java.util.Scanner to read the file.

```java
/**
 * A program which completes two tasks. The first is creating graphs from
 * a file and printing the matrices, adjacency lists, depth-first and
 * breadth-first traversals.
 *
 * The second is reading a file of a constant number of Strings and
 * populating a binary search tree with the items, printing the paths
 * taken on the tree. It then prints an in-order traversal, followed
 * by the search results of finding 42 distinct items and the average
 * comparisons needed to search in the tree.
```

```java
 */
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Assignment4 {
  public static void main(String[] args) {
    final int NUM_OF_ITEMS = 666;          // Length of file as constant
    String[] magicItems = new String[NUM_OF_ITEMS]; // Array of file strings

    // Read graphs1.txt and create matrix, adjacency list, and linked objects
    try {
      File graphs = new File("graphs1.txt");
      Scanner graphRead = new Scanner(graphs);
      MarcusGraphs graph = null;
      String command = null;
      String item = null;
      while (graphRead.hasNextLine()) {
        command = graphRead.next();
        if (command.equals("--")) {
        // Skip comment if null, print and execute methods if exists
          if (graph == null) {
            graphRead.nextLine();
          } else {
            System.out.println("Matrix:");
            graph.printMatrix();
            System.out.println("Adjacency list:");
            graph.printAdjacencyList();
            System.out.println("Depth-first traversal:");
            graph.depthFirstTraversal(graph.getInitialVertex());
            System.out.print("\n\n");
            System.out.println("Breadth-first traversal:");
            graph.breadthFirstTraversal(graph.getInitialVertex());
          }
        } else if (command.equals("new")) {
        // Create new graph
          graph = new MarcusGraphs();
          graphRead.nextLine();
        } else if (command.equals("add")) {
          item = graphRead.next();
          if (item.equals("vertex")) {
          // Add new vertex to graph
            MarcusVertex vertex = new MarcusVertex(graphRead.nextInt());
            graph.addVertex(vertex);
          } else if (item.equals("edge")) {
          // Add new edge to graph
```

```
        int a = graphRead.nextInt();
        graphRead.next();
        int b = graphRead.nextInt();
        MarcusVertex first = graph.getVertexById(a);
        MarcusVertex second = graph.getVertexById(b);
        first.addNeighbor(second);
        second.addNeighbor(first);
      }
    }
  }
  System.out.println("Matrix:");
  graph.printMatrix();
  System.out.println("Adjacency list:");
  graph.printAdjacencyList();
  System.out.println("Depth-first traversal:");
  graph.depthFirstTraversal(graph.getInitialVertex());
  System.out.print("\n\n");
  System.out.println("Breadth-first traversal:");
  graph.breadthFirstTraversal(graph.getInitialVertex());
  graphRead.close();
} catch (FileNotFoundException e) {
  System.out.println("Whoops! Couldn't find graphs1.txt");
  e.printStackTrace();
}

// Try/catch block for file import and reading
try {
  File file = new File("magicitems.txt");
  Scanner read = new Scanner(file);
  for (int i = 0; i < NUM_OF_ITEMS; i++) {
    magicItems[i] = read.nextLine();
  }
  read.close();
} catch (FileNotFoundException e) {
  System.out.println("Whoops! Couldn't find magicitems.txt");
  e.printStackTrace();
}

// Populate BST with magicItems, printing the path from the root
MarcusBST binarySearchTree = new MarcusBST();
for (String item : magicItems) {
  MarcusNode node = new MarcusNode(item);
  binarySearchTree.insertNode(node);
  System.out.println(item + ": " + binarySearchTree.getPath());
}
System.out.print("\n\n");
```

```java

    // Print the entire BST with an in-order traversal
    binarySearchTree.inOrderTraversal(binarySearchTree.getRoot());

    // Read magicitems-find-in-bst.txt and lookup in BST, printing path
    try {
      File itemsToFind = new File("magicitems-find-in-bst.txt");
      Scanner bstRead = new Scanner(itemsToFind);
      String currentItem;
      double average = 0;
      while (bstRead.hasNextLine()) {
        currentItem = bstRead.nextLine();
        binarySearchTree.search(currentItem);
        System.out.println(currentItem + ": " + binarySearchTree.getPath());
        System.out.println("  Number of comparisons: "
                  + binarySearchTree.getCounter());
        average += binarySearchTree.getCounter();
      }
      bstRead.close();
      average /= 42.0;
      System.out.println("Average comparisons: " + average);
    } catch (FileNotFoundException e) {
      System.out.println("Whoops! Couldn't find magicitems-find-in-bst.txt");
      e.printStackTrace();
    }
  }
}
```

6 Results & Analysis

The asymptotic running times of both depth-first and breadth-first traversals are similar to $O(n)$, but could be better described as $O(|V+E|)$, where V is the set of all vertices in the graph and E is the set of all edges in the graph. This distinction is important as we need to track the change in the running time due to two inputs instead of just one.

The asymptotic running time of lookup in a binary search tree is $O(log(n))$. This is due to the logarithmic nature of searching the tree: since the tree is sorted by nature, we can discard half of the tree with each level we descend. This is further supported by our number of average comparisons to locate 42 items in a tree of 666 items: 10.64.