Implementing an Undirected Graph and Breadth-First Search Algorithm

Homework #5

By Logan Miles

1. Objectives

The objective of this project was to implement an undirected graph using adjacency lists and a breadth-first search algorithm on the graph using a high-level programming language. The nodes of the undirected graph had to be read from a text file, then added into an undirected graph via a linked list of connected nodes to each node. The text files that were given in this assignment were “mediumG.txt” and “largeG.txt.” Each listed the number of vertices in line one and the number of edges in line two. The “mediumG” file contains 1275 lines, representing 250 nodes and 1273 edges. While the “largeG” file contains 7586065 lines, representing 1 million nodes and 7586063 edges. The graph had to then be verified by a print function that displays the adjacency list of each node in the graph. The breadth-first search algorithm must then search for the path that can access all other nodes and then print said path node-by-node.

2. Program Design

To implement the required functionality of the assignment, two class files were developed: a HW5 class containing the driver code and a Graph class containing the functions necessary to construct the graph, print the adjacency lists, and perform breadth-first search. The following functions are contained within the two classes:

Graph()

This constructor initializes the graph object when called. The vertices attribute is assigned the number of nodes that the constructor is passed as a parameter. The edges is initialized as 0 because the number of edges is iterated forward every time the addEdge() function is called. A new array list object that contains array lists is also initialized to contain array lists that represent the adjacency list of each node. A for loop is used to add a new array list for each node.

addEdge()

This function is passed two integers, node1 and node2, that are read from the text file in main. It then calls the get() function on adjList to get the current adjacency list for that each node and the add() function on that list to each node to the others adjacency list. This adds an edge in the graph between both nodes in both directions. The nodes are added to each other's list because the graph is supposed to be unidirectional.

printAdjList()

This function is responsible for representing the graph as an adjacency list for each node. The function first prints a statement containing the number of edges and vertices. Next, it iterates through the list of adjacency lists for each node by iterating from 0 to the number of vertices in the graph, which corresponds to the number of lists contained within the adjacency list. A nested for loop then iterates through each neighbor contained in the current adjacency list. The function then appends the neighbor to the string builder object. Once the nested for loop exits, a newline is appended to the string. After the outer for loop exits, the function prints the string.

getVertices(), getEdges(), getAdjList()

These getter functions return the vertices integer, edges integer, and adjacency list array list object from the graph object. They are called throughout the other functions in the Graph and HW5 class.

setVertices() and setEdges()

These functions set the vertices integer and the edges integer of the graph object initialized in the HW5 class.

BFS()

The BFS() function is passed an integer called startNode from which it performs breadth-first search. The function accomplishes this using an array of Boolean values corresponding to each vertex and a queue of integers used to queue and print neighbors. The function first adds the starting node to the queue and assigns it as visited in the Boolean matrix as it is the starting node and must be visited. While the queue is not empty, each node is first dequeued then printed to indicated which node is currently being processed. A for loop then iterate through each neighbor contained in the adjacency list of that node. If the node has not been visited, the node is marked as visited and it is added to the queue. This continues until the while loop exits due to the queue being empty. A flag variable called hasUnvisitedNodes is then initialized as false, and a for loop iterates through the vertices again, this time checking their index in the Boolean matrix to see if they have been visited. If they have not, the flag is set to true, and the loop exits. A final if statement checks if the flag is true, printing a statement that some nodes were inaccessible.

main()

The driver code contained in main() is responsible for reading the text file and then calling the above functions from the Graph class to achieve the codes desired functionality. First the file name is initialized as a string. The count() function is then called on the lines in the file to determine how many lines, and therefore how many nodes should exist in the graph when it is constructed. The constructor from the Graph class is then called. It is passed the numberOfLines variable mentioned before, only casted to an integer. A buffered reader object is then initialized and passed the filename variable. A while loop continues as long the line that the reader reads is not null, meaning it stops at the end of the file. An integer lineCount is also initialized and then iterated forward with each iteration of the loop to keep track of the current line. If the current line is the first one, the setVertices() function is called from the graph which sets the number of vertices attribute in the graph object to the number specified at the beginning of the text file. The second if statement checks if the current line is two, continuing if so. This is because the number of edges is determined by the addEdge() function. Finally, the printAdjList() and BFS() functions are called from the Graph class. The nanoTime() function from the System class records the time before and after execution of the BFS() function, then calculates the difference and prints it in different units of time. This was used in testing to compare the time complexity of the algorithm among different text file sizes and starting nodes.

Code Screenshots:

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

You, 7 minutes ago | 2 authors (You and others)
public class HW5 {

    /*
    Description: Executable function that is responsible for reading the text file and calling the functions in the Graph cl
    Parameters:
    String[] args - Runtime arguments
    Returns: Nothing
    Sources:
    https://stackoverflow.com/questions/26448352/counting-the-number-of-lines-in-a-text-file-java
    https://www.youtube.com/watch?v=X1LdtRW88c0        You, 7 minutes ago • Uncommitted changes
    */
    Run | Debug
    public static void main(String[] args) {
        String filename = "mediumG.txt"; //initialize filename string variable
        try {
            long numberOfLines = Files.lines(Paths.get(filename)).count(); //count the number of lines in the text file
            Graph graph = new Graph((int)numberOfLines); //initialize graph object with the number of lines in the text file
            BufferedReader reader = new BufferedReader(new FileReader(filename)); //initialize reader object
            String line;
            int lineCount = 0; //initialize line count
            while ((line = reader.readLine()) != null) { //iterate through the text file with a while loop
                lineCount++; //iterate the line count variable forward 1 with each loop
                if (lineCount == 1) { //case for first line
                    graph.setVertices(Integer.parseInt(line)); //parse the int value representing the number of vertices and
                }
                else if (lineCount == 2) { //case for second line
                    continue; //skip because addEdge() adds edges
                }
                else {
                    String[] nodes = line.split(regex:" "); //split each line at the space using a regex
                    int node1 = Integer.parseInt(nodes[0]); //parse the int value and assign it to node1
                    int node2 = Integer.parseInt(nodes[1]); //parse the int value and assing it to node2
                    graph.addEdge(node1, node2); //call addEdge() on both nodes to add them to the graph
                }
            }
            reader.close();
            graph.printAdjList(); //call printAdjList() to print the adjacency list representation of the graph
            long timeInit = System.nanoTime(); //records initial system time in nanoseconds
            graph.BFS(startNode:10); //call BFS on following nodes
            long timeFinal = System.nanoTime(); // records final system time in nanoseconds
            long time = timeFinal - timeInit; //calculates time taken for BFS algorithm
            System.out.println("Breadth-First Search Time: " + time + " nanoseconds, " + (float)time/1000000 + " millisecond
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 1: HW5.java

```java
You, 3 minutes ago | 2 authors (You and others)
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

You, 3 minutes ago | 2 authors (You and others)
public class Graph {

    private int vertices; //number of vertices in the graph
    private int edges; //number
    private List<List<Integer>> adjList;

    /*
    Description: This is the constructor for the graph object
    Parameters:
    int nodes - The number of nodes the graph will contain
    Returns: Nothing
    Sources:
    https://chat.openai.com/share/8b4c2e60-b4e3-4b2a-909b-4a3300ec4287
    https://www.youtube.com/watch?v=X1LdtRW88c0
    */
    public Graph(int nodes) { //constructor for Graph object
        this.vertices = nodes; //initializes the number of vertices as the number of nodes
        this.edges = 0; //initializes the number of edges to 0
        this.adjList = new ArrayList<>(); //initilizes ArrayList for adjacency lists for each node
        for (int i = 0; i < nodes; i++) { //for each node within the graph a new adjacency list is created
            adjList.add(new ArrayList<>()); //adds the list for each node to adjList
        }
    }

    /*
    Description: This function adds undirected edges to the graph represented as a 1 in the adjacency matrix both ways
    Parameters:
    int node1 - A node read from the text file to be connected to node2 via an edge
    int node2 - A node read from the text file to be connected to node1 via an edge
    Returns: Nothing
    Sources:
    https://chat.openai.com/share/8b4c2e60-b4e3-4b2a-909b-4a3300ec4287
    https://www.youtube.com/watch?v=X1LdtRW88c0
    */
    public void addEdge(int node1, int node2) {
        adjList.get(node1).add(node2); //gets the adjacency list for node1 and adds node2 to that list
        adjList.get(node2).add(node1); //gets the adjacency list for node2 and adds node1 to that list
        edges++;
    }


    /*
    Description: This function iterates through the list of adjacency lists for each node, appending each element to a strin
    Parameters: None
    Returns: Nothing
    Sources:
    https://www.youtube.com/watch?v=X1LdtRW88c0
    */
    public void printAdjList() {
        StringBuilder string = new StringBuilder(); //initialize StringBuilder object
        string.append("The graph contains "+ edges + " edges and " + vertices + " vertices. \n");
        for (int i = 0; i < vertices; i++) { //iterates through the list of adjacency lists using using a for loop stopping
            string.append("Adjacency list for node " + i + ": ");
            for (int neighbor : adjList.get(i)) { //for each neighbor contained in the nested adjacency list
                string.append(neighbor + " ");
            }
            string.append(str:"\n");
        }
        System.out.println(string);
    }
```
Figure 2: Graph.java (1)

```java
    public int getVertices() { //getter for vertices
        return this.vertices;
    }

    public void setVertices(int vertices) { //setter for vertices
        this.vertices = vertices;
    }

    public int getEdges() { //getter for edges
        return this.edges;
    }

    public void setEdges(int edges) { //setter for edges
        this.edges = edges;
    }

    public List<Integer> getAdjList(int node) { //getter for adjList of a certain node
        return adjList.get(node);
    }

    /*
    Description: This function uses breadth-first search to traverse the graph and print each node
    Parameters: None
    Returns: Nothing
    Sources:
    https://chat.openai.com/share/3615b67f-45b3-411b-80d5-4ac6984224e1
    https://stackoverflow.com/questions/5262308/how-do-implement-a-breadth-first-traversal      You, 3 minutes ago • Uncomm
    */
    public void BFS(int startNode) {
        boolean[] visited = new boolean[vertices]; //matrix containing boolean values for each node; 1 means visited, 0 mea
        Queue<Integer> queue = new LinkedList<>(); //creates a queue of integers using a LinkedList as the underlying data

        visited[startNode] = true; //set the startNode variable to visited in the boolean matrix as it is the starting node
        queue.add(startNode); //add starting node to the queue

        System.out.print("Breadth-First Search starting from node " + startNode + ": ");
        while (!queue.isEmpty()) { //while the queue is not empty
            int node = queue.poll(); //dequeues the front node and assigns it to the node variable
            System.out.print(node + " "); //prints node to indicate which node is currently being processed

            for (int neighbor : adjList.get(node)) { //iterates through neighbors of the current node
                if (!visited[neighbor]) { //if the neighbor has not been visited yet
                    visited[neighbor] = true; //set visited to true in matrix
                    queue.add(neighbor); //add neighbor to que for printing
                }
            }
        }

        boolean hasUnvisitedNodes = false; //unvisited nodes flag
        for (int i = 0; i < vertices; i++) { //iterates through vertices to check if any node remained unvisited
            if (!visited[i]) { //if the current node is unvisited
                hasUnvisitedNodes = true; //set flag to true
                break;
            }
        }

        if (hasUnvisitedNodes) { //case for there being no path to some nodes
            System.out.println(x:"\nThere is no path to some nodes.");
        } else {
            System.out.println(); //move to the next line after printing the BFS traversal
        }
    }
}
```

Figure 3: Graph.java (2)

3. Testing

Testing consisted of executing the code on both files and recording the execution time of the BFS() algorithm on different starting nodes and verifying printAdjList() outputs.

Testing Screenshots:

```
Breadth-First Search starting from node 10: 10 105 106 123 175 246 131 143 193 243 179 84 85 82 11 244 152 8 212 30 192 174 103 19
 79 70 100 207 43 210 221 214 219 156 110 162 13 51 2 133 205 122 139 196 108 101 147 140 99 129 86 18 14 42 141 166 92 181 135 12
5 157 7 117 54 16 6 178 236 35 94 132 171 172 235 184 188 230 148 71 65 57 197 167 98 36 41 88 198 28 12 242 154 238 180 213 245 1
65 155 142 124 118 27 21 233 240 138 62 151 208 224 47 29 182 81 121 170 17 223 113 90 195 191 68 9 239 102 78 77 26 4 226 128 168
 187 231 52 32 218 146 137 91 64 109 227 119 134 229 53 158 249 200 173 23 202 204 222 225 176 160 114 97 93 58 49 44 0 33 159 112
 55 5 217 136 69 248 144 104 185 201 145 183 215 126 74 38 120 56 161 73 34 22 203 189 186 177 72 107 1 209 211 163 80 15 59 149 2
4 50 234 67 232 83 48 45 87 216 169 46 150 220 130 164 194 206 66 39 61 3 95 76 111 190 40 247 116 75 20 89 37 115 153 228 241 60
25 127 31 63 96 199 237
Breadth-First Search Time: 14974300 nanoseconds, 14.9743 milliseconds, or 0.0149743 seconds
```

Figure 4: BFS() on 10 in mediumG.txt

```
Breadth-First Search starting from node 100: 100 103 133 174 192 84 70 19 13 243 166 129 51 14 179 79 162 106 131 193 212 214 244
30 105 236 147 16 6 178 99 86 110 18 2 152 143 8 140 123 246 10 219 221 210 207 156 82 43 11 205 196 139 122 108 101 117 98 54 135
 141 42 35 94 85 175 92 181 125 157 7 167 88 36 230 71 41 198 28 12 242 132 171 172 235 184 188 148 65 57 197 224 47 29 182 27 21
233 240 62 81 121 170 17 223 113 90 154 238 180 213 245 165 155 142 124 118 138 151 208 218 146 137 91 64 109 227 239 102 78 77 26
 4 128 119 134 229 53 158 249 200 173 195 191 68 9 226 168 187 231 52 32 145 183 215 126 74 38 159 112 55 5 217 136 69 120 56 161
73 34 22 203 189 186 177 72 107 1 23 202 204 222 225 176 160 114 97 93 58 49 44 0 33 248 144 104 185 201 234 67 232 83 48 45 87 16
9 46 150 220 130 164 194 209 211 163 80 15 59 149 24 50 216 61 3 95 76 111 190 40 247 116 75 20 89 206 66 39 37 115 153 228 241 60
 25 127 31 63 96 199 237
Breadth-First Search Time: 17306000 nanoseconds, 17.306 milliseconds, or 0.017306 seconds
```

Figure 5: BFS() on 100 in mediumG.txt

```
7760 217058 992314 459556 589088 263149 398303 317980 549283 980157 518237 128836 64535 637659 839174 563782 475882 916603 468791
124563 276852 29933 388578 820303 956599 643540 918838 968895 575481 611521 811421 945253 88441 548756 348770 176893 293277 368057
 239937 253091 272705 189826 68928 56692 156621 422853 448906 652609 784779 873098 908577 996443 845772 118000 888500 886099 94236
6 787636 647511 227643 895675 445942 427734 1514 123363 336434 133593 80062 432096 475069 730271 294149 232360 641679 260252 14536
4 51268 333546 685421 945667 458755 383580 908948 752276 901979 488429 334339 499952 391313 588873 917326 387777 115796 460269 353
501 536844 879836 79083 57747 242499 826103 422346 742719 364548 831466 70027 97586 731193 967171 457443 827817 366763 163277 6392
72 522037 93663 496182 66300 596817 572300 410340 777224 564410 484631 455185 261234 536134 917073 370791 934246 943051 978438 984
472 536909 453843 269666 329885 750994 675103 107368 641866 621731 809725 485311 142061 100346 685119 309680 932691 55065 160707 7
78326 494579 301397 153467 125239 803940 686656 598706 457839 745571 390459 925712 823864 942301 561252 503240 59645 525709 353994
 114394 105984 951613 867630 778925 658446 539903 754093 761659 301378 83506 55737 77680 558288 640581 127520 316336 315201 506518
 466246 849467 89955 747951 690716 680270 521066 960569 868440 765377 401635 227974 43042 985946 800180 34558 982555 204004 171625
 646412 657802 226371 56359 511655 398915 650807 843291 427270 78529 967095 698151 325591 652331 602684 2415 746530 565299 722731
279152 181947 901196 431556 289847 678013 200257 12199 799826 798143 903774 637049 511861 415118 320313 304054 298309 283652 20790
5 159771 866550 412489 317209 111489 6542 681477 344756 216487 579535 560739 542367 300515 98423 509408
Breadth-First Search Time: 36588461800 nanoseconds, 36588.46 milliseconds, or 36.588463 seconds
```

Figure 6: BFS() on 10 in largeG.txt

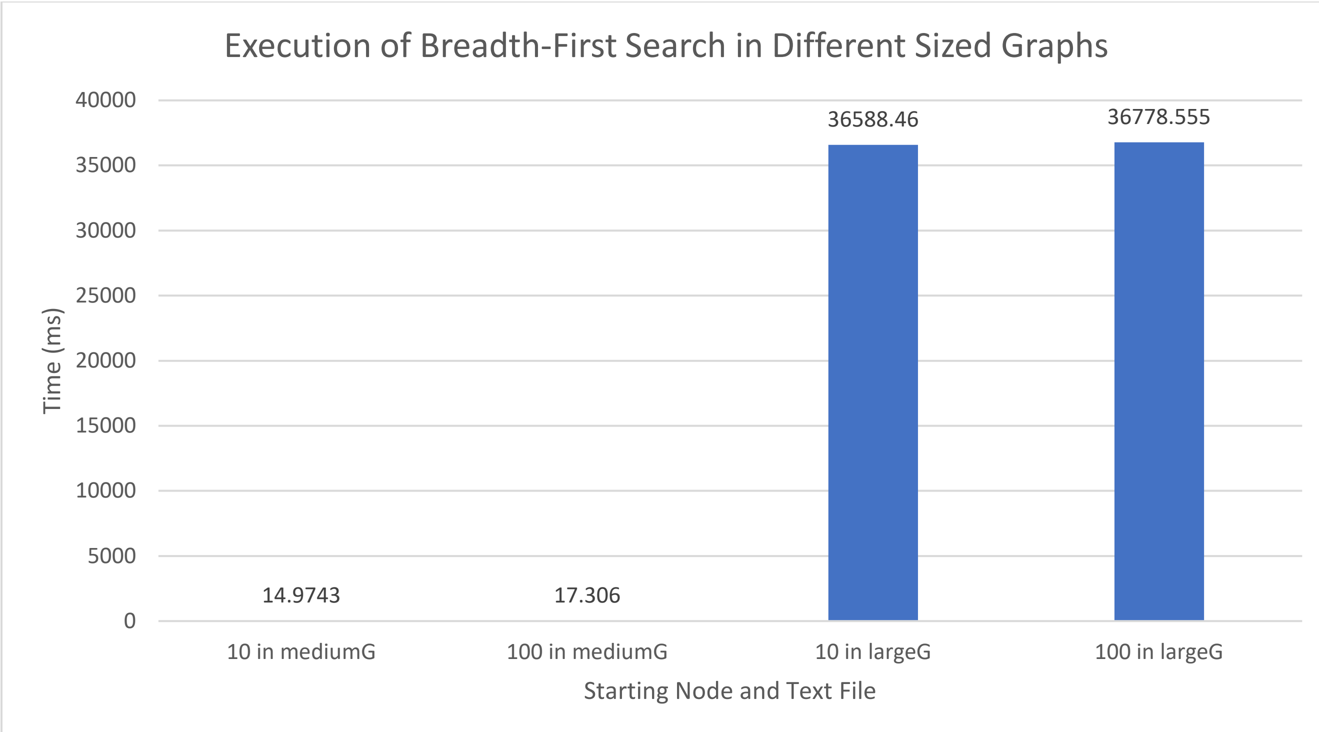
```
96 149711 115102 938352 726776 173568 325500 113509 311722 187567 97173 940219 115668 668489 427122 424767 181197 167951 428290 41
0877 851334 844775 812879 733778 691140 92408 118417 378595 890359 420143 21913 21710 99743 775763 74509 893183 986239 453165 1972
85 190680 674229 897026 185775 66882 597093 766931 976811 79896 321804 695112 701276 909973 858079 698756 447963 68116 210434 6384
27 551157 185781 483478 629965 818296 859859 891567 901161 207537 666245 167099 378200 117099 654685 867549 146265 161610 34452 13
9624 544938 794855 769610 951476 836624 817435 359619 131189 954570 158168 121478 734630 639360 217115 130200 310698 895143 961629
 1681 959040 632251 789768 951121 377752 938186 254737 35296 465138 458738 409751 304875 279781 71614 960524 834283 74991 712958 9
84316 454374 884714 63696 398264 235919 583282 743262 51658 861417 847086 574175 917634 616444 288381 276778 524292 679071 709743
758144 119592 726797 674540 304391 746799 643434 650620 500939 939606 974621 188877 700488 971896 100193 839950 788241 915678 1974
87 546068 156778 139679 195641 201260 234342 897345 378035 556656 853315 987490 162961 155251 669372 971044 625374 990569 219566 1
22693 569077 193533 608455 555789 525842 716029 75612 251923 346372 328728 930376 226592 684973 819655 846676 602703 276858 425874
 210324 502793 737964 869517 861924 563781 498353 469886 313877 205393 486213 721290 753741 890329 96893 206403 730850 971629 1811
79 131042 590762 327039 403861 332742 706651 483136 163476 235996 355147 695038 771030 235814 17228 513499 410225 190205 211841 36
4750 880558 201347 221170 492252 785122 973470 380237 528968 255527 992175 404733 408217 476564 870163 915488 308875 165912 858701
 503521 8508 954487 232390 144233 341351 502748 625085 282791 838122 930500 673032 787411 901768 260964 245869 191250 144269 46882
4 173076 237372 920214 854466 500917 201727 481967 798188 289732 829196 375777 121711 832026 5236 273776
Breadth-First Search Time: 36778555000 nanoseconds, 36778.555 milliseconds, or 36.778553 seconds
```

Figure 7: BFS() on 100 in largeG.txt

```
The graph contains 1273 edges and 250 vertices.
Adjacency list for node 0: 15 24 44 49 58 59 68 80 97 114 149 160 163 176 191 202 204 209 211 222 225
Adjacency list for node 1: 72 107 130 150 164 189 194 200 203 220
Adjacency list for node 2: 14 18 42 51 79 86 108 110 141
Adjacency list for node 3: 37 45 67 76 115 153 228 241
Adjacency list for node 4: 5 26 55 77 78 112 128 138 159 239 240
Adjacency list for node 5: 26 32 55 67 77 102 104 217 226 4
Adjacency list for node 6: 16 54 98 99 117 129 140 147 166 178 236
Adjacency list for node 7: 42 57 65 71 101 125 148 157 181 184 188 197 230
Adjacency list for node 8: 11 30 43 82 85 143 152 179 207 210 212 221 244 246
Adjacency list for node 9: 23 33 58 68 114 142 195
Adjacency list for node 10: 105 106 123 175 246
Adjacency list for node 11: 30 43 82 85 143 152 175 207 212 244 246 8
Adjacency list for node 12: 28 35 36 41 88 94 113 121 170 182 198 242
Adjacency list for node 13: 19 100 103 129 133 162 174 192
Adjacency list for node 14: 18 51 86 129 133 166 2
Adjacency list for node 15: 24 39 49 58 66 80 114 149 163 202 204 209 211 222 225 0
Adjacency list for node 16: 54 98 99 117 129 140 147 166 178 236 6
Adjacency list for node 17: 41 81 121 134 158 170 182 223 229
Adjacency list for node 18: 35 51 86 94 141 14 2
Adjacency list for node 19: 70 79 84 100 103 174 179 192 243 13
Adjacency list for node 20: 40 75 89 116 127 164 190 194 220 247
Adjacency list for node 21: 27 62 65 71 138 184 188 230 233 240
Adjacency list for node 22: 34 53 56 73 120 145
Adjacency list for node 23: 33 58 68 114 176 195 222 9
Adjacency list for node 24: 39 66 80 114 149 163 206 209 211 222 225 15 0
Adjacency list for node 25: 60 63 96 111 199
Adjacency list for node 26: 55 77 78 102 138 217 226 239 240 5 4
Adjacency list for node 27: 62 65 71 138 184 188 230 233 240 21
Adjacency list for node 28: 35 41 94 113 121 170 182 198 223 242 12
Adjacency list for node 29: 47 64 91 109 137 146 167 218 224 227
Adjacency list for node 30: 43 70 79 82 143 152 156 179 207 210 212 214 219 221 244 11 8
Adjacency list for node 31: 37 115 153 228 241
Adjacency list for node 32: 52 77 93 102 104 144 151 160 168 185 187 201 208 226 231 248 5
Adjacency list for node 33: 58 114 163 222 23 9
Adjacency list for node 34: 53 56 73 120 145 22
Adjacency list for node 35: 36 41 88 94 141 198 28 18 12
Adjacency list for node 36: 41 88 98 182 35 12
Adjacency list for node 37: 76 95 115 153 228 241 31 3
Adjacency list for node 38: 74 109 126 183 215
Adjacency list for node 39: 66 80 149 206 209 211 24 15
Adjacency list for node 40: 75 89 116 150 164 190 194 220 247 20
Adjacency list for node 41: 81 88 121 170 182 198 36 35 28 17 12
Adjacency list for node 42: 86 101 108 135 141 157 181 196 7 2
Adjacency list for node 43: 82 152 156 207 210 212 219 221 244 30 11 8
Adjacency list for node 44: 49 59 68 80 93 97 144 160 168 176 185 191 202 204 222 225 231 248 0
Adjacency list for node 45: 48 67 76 83 95 104 217 232 3
Adjacency list for node 46: 161 169 177 186
Adjacency list for node 47: 64 91 109 137 146 167 218 224 29
Adjacency list for node 48: 50 83 104 144 185 201 216 217 232 248 45
Adjacency list for node 49: 59 80 93 97 144 160 176 185 191 202 204 222 225 248 44 15 0
Adjacency list for node 50: 59 80 97 104 144 185 201 232 248 48
Adjacency list for node 51: 70 79 86 110 133 214 18 14 2
Adjacency list for node 52: 77 93 102 151 168 187 208 226 231 32
Adjacency list for node 53: 56 73 81 119 120 134 145 229 34 22
Adjacency list for node 54: 99 117 140 147 16 6
Adjacency list for node 55: 67 78 112 128 136 159 217 239 26 5 4
Adjacency list for node 56: 73 119 120 145 161 53 34 22
Adjacency list for node 57: 65 118 125 148 151 157 172 181 184 188 197 208 230 7
Adjacency list for node 58: 68 114 163 176 191 202 204 209 211 222 33 23 15 9 0
Adjacency list for node 59: 80 97 144 185 204 225 248 50 49 44 0
Adjacency list for node 60: 63 96 111 199 237 25
Adjacency list for node 61: 87 89 111 130 194 234
Adjacency list for node 62: 71 78 90 128 138 188 233 239 240 27 21
Adjacency list for node 63: 96 199 237 60 25
Adjacency list for node 64: 91 109 119 134 137 145 146 183 215 218 227 47 29
Adjacency list for node 65: 71 125 138 148 151 157 181 184 188 197 208 230 240 57 27 21 7
Adjacency list for node 66: 149 206 209 39 24 15
Adjacency list for node 67: 83 112 217 55 45 5 3
Adjacency list for node 68: 114 160 165 176 191 202 204 222 58 44 23 9 0
Adjacency list for node 69: 107 128 173
Adjacency list for node 70: 79 84 100 174 179 212 214 244 51 30 19
Adjacency list for node 71: 135 148 157 181 184 188 230 233 240 65 62 27 21 7
Adjacency list for node 72: 107 150 177 186 189 200 203 220 249 1
Adjacency list for node 73: 120 145 56 53 34 22
Adjacency list for node 74: 109 126 183 215 38
Adjacency list for node 75: 89 116 164 190 194 220 247 40 20
Adjacency list for node 76: 95 115 153 228 241 45 37 3
Adjacency list for node 77: 78 102 138 151 187 208 226 240 52 32 26 5 4
Adjacency list for node 78: 112 128 138 159 239 240 77 62 55 26 4
Adjacency list for node 79: 84 110 174 179 212 214 70 51 30 19 2
Adjacency list for node 80: 97 149 202 204 225 59 50 49 44 39 24 15 0
Adjacency list for node 81: 119 134 146 227 229 53 41 17
Adjacency list for node 82: 85 152 175 207 212 244 246 43 30 11 8
Adjacency list for node 83: 95 104 201 217 232 67 48 45
Adjacency list for node 84: 100 103 106 131 174 179 192 193 243 79 70 19
Adjacency list for node 85: 152 175 246 82 11 8
Adjacency list for node 86: 108 135 141 51 42 18 14 2
Adjacency list for node 87: 111 130 136 194 234 61
Adjacency list for node 88: 98 182 41 36 35 12
Adjacency list for node 89: 116 127 130 164 194 75 61 40 20
Adjacency list for node 90: 113 173 233 242 62
Adjacency list for node 91: 109 119 134 137 145 146 218 224 227 64 47 29
Adjacency list for node 92: 122 132 139 171 172 205 235
Adjacency list for node 93: 97 144 160 168 176 185 187 191 202 204 226 231 248 52 49 44 32
Adjacency list for node 94: 141 198 242 35 28 18 12
Adjacency list for node 95: 115 153 216 83 76 45 37
Adjacency list for node 96: 199 237 63 60 25
Adjacency list for node 97: 144 160 168 176 185 191 202 204 225 231 248 93 80 59 50 49 44 0
Adjacency list for node 98: 117 178 236 88 36 16 6
Adjacency list for node 99: 129 140 147 162 54 16 6
Adjacency list for node 100: 103 133 174 192 84 70 19 13
Adjacency list for node 101: 108 110 122 125 139 156 157 181 196 205 214 219 42 7
Adjacency list for node 102: 138 187 226 240 77 52 32 26 5
```

Figure 8: printAdjList() output from mediumG.txt

```
Adjacency list for node 2730: 287733 459092 531810 663737 272174 342307 356885 366350 383662 415512 545480 791370 906244 951164 10
1961 801358 862134 928164
Adjacency list for node 2731: 14457 50679 59060 101129 127200 233425 342181 349027 519041 553560 588961 624514 733857 801231 83246
6 859636 900945 646610
Adjacency list for node 2732: 122654 129753 284474 289774 327356 482824 565318 570625 610554 639018 669878 770515 805478 818191 85
9582 885580 892561 929315
Adjacency list for node 2733: 17682 16640 252302 290849 493830 554629 587654 860479 913506 50804 129245 366130 709528 455236 53647
8 588888 818786
Adjacency list for node 2734: 53821 68587 99439 222046 236772 331269 349430 458930 468363 490163 522255 711430 719411 810298 81465
0 841812 884367 913537
Adjacency list for node 2735: 29682 112372 124630 136848 218324 219702 449146 454035 528646 696028 711192 788888 833657 994103
Adjacency list for node 2736: 186080 223563 294333 327652 479592 484228 499090 598840 614245 663911 692739 704534 707123 721845 77
8417
Adjacency list for node 2737: 83145 156317 210933 214805 245679 254928 277497 382642 404585 410128 413544 612484 650987 974579 508
792 494192
Adjacency list for node 2738: 24981 53629 104387 131137 153322 321880 324648 347006 356537 371082 378570 397732 491771 593452 6118
17 696376 751487 767787 820887 840642 932079 988154
Adjacency list for node 2739: 106019 152404 184865 202638 252414 346016 473559 833541 879639
Adjacency list for node 2740: 40783 278474 351861 391630 995369 233754 269496 340890 646087 785556 813647 879233 894551 950216
Adjacency list for node 2741: 391410 819006 99123 137443 184738 185878 188030 307846 361650 399330 420670 463824 464487 740034 804
236 864075 956410
Adjacency list for node 2742: 76982 235811 335242 381961 390942 445659 510084 548535 656804 672171 702638 835448 939679 54206 1118
84 167158 320439 338134 379486 391683 578048 619221 648164 694168 871863
Adjacency list for node 2743: 765572 849568 927143 39820 136815 147557 191907 210807 211730 351251 369802 482303 488770 532709 585
837 698211 842798 959728 987971
Adjacency list for node 2744: 869231 118307 236656 342126 382320 425239 606859 635521 690933 710777 752304 818135 827662 842153 92
7789 945133
Adjacency list for node 2745: 299442 564734 711194 58596 311036 393052 451743 673065 876069 882179 886460 997921 926865 998013
Adjacency list for node 2746: 8434 134038 324219 417229 809404 865330 38542 65049 113481 402592 470451 528004 554845 627128 754968
 785530 838746 240245
Adjacency list for node 2747: 70841 192513 281798 537204 543269 619541 693470 835396 899246 925622 947131 287037 377151 389055 490
965 560532 677617
Adjacency list for node 2748: 78678 359898 480371 507926 521566 553977 593990 595224 631496 680925 762491 882656 933290 995679 722
538
Adjacency list for node 2749: 299659 299786 343326 343973 374428 441687 520668 606596 647024 666685 676393 702242 707382 757458
Adjacency list for node 2750: 48981 122858 230432 352542 565965 711370 796234 826937 837788 895929
Adjacency list for node 2751: 797629 102308 297228 316199 588874 629271 734707 962714
Adjacency list for node 2752: 49793 767301 859174 9109 33164 47074 102737 122083 135595 236446 254278 267236 304503 360775 452279
529210 541147 624585 637533 790197 849084 931077 932950
Adjacency list for node 2753: 448253 59472 85671 171389 221130 224202 327105 373974 379839 515191 521442 616958 672325 789132 8040
19 829264 840851
Adjacency list for node 2754: 439414 598157 602044 697964 812823 843508 75285 102557 216465 341182 471128 615260 654027 781363 802
288
Adjacency list for node 2755: 52719 63004 202146 273734 385164 401506 418359 605816 673555 689184 703627 816233 885555 982760 9978
85
Adjacency list for node 2756: 309644 391853 401710 428854 480776 548078 557497 728736 943638 982872 169536 219661 459261 564998 82
5513
Adjacency list for node 2757: 457326 757125 942772 944719 353397 356388 396189 619697 715246 733915 40268 476464 959251 108682 275
312 285584 630485
Adjacency list for node 2758: 37778 97932 131681 230069 258207 305134 396960 636837 660837 757846 809968 116905
Adjacency list for node 2759: 620460 641123 742674 786400 793693 796896 22567 324397 433357 472917 684116 848986 927850 964862
Adjacency list for node 2760: 221913 223312 242609 313779 326119 385760 497861 511409 527328 538905 574120 596491 636710 864990 93
6820 982131 13276 94088 301634 419195 816707 337
Adjacency list for node 2761: 54522 55419 84994 108527 133343 166781 192598 207565 235675 257189 313954 343577 361215 370945 40857
4 433013 470873 475803 515262 542665 546336 575983 581497 589052 589924 785772
Adjacency list for node 2762: 7118 86122 88350 213258 375717 382299 393031 453462 457703 536978 626857 772740 943465 987837
Adjacency list for node 2763: 54962 444074 679124 715153 909413 944735 339084 395443 659138
Adjacency list for node 2764: 287828 538729 576871 980447 985689 9536 253152 454262 526032 561319 580698 600460 617916 723015 8126
84 936110 957710
Adjacency list for node 2765: 632474 57542 134761 295110 512708 528307 597157 652060 737318 936525 960271 85483 128481 158946 2952
56 424694 972706
Adjacency list for node 2766: 694852 940904 60600 557781 129247 160701 600369 705414 874708 956203 51556
Adjacency list for node 2767: 47886 114600 233022 242766 265924 275320 311144 643656 650935 689811 806044 807140 828864 878821 890
565 911248 997051 748341 809721 962885 990120
Adjacency list for node 2768: 71084 204545 302565 992880 97421 222373 336308 610246 907208 702998 223375 476022 738576
Adjacency list for node 2769: 45384 211530 217176 495099 499739 519032 628244 784470 799630 814193 815895 845633 907005 943490 966
565 992047
Adjacency list for node 2770: 99507 256103 301741 322415 548124 699229 295754 706130 739853
Adjacency list for node 2771: 15686 203438 282894 593648 627306 643092 870254 890319 971771 432370 661462 694211 776636
Adjacency list for node 2772: 39580 110818 125146 327918 498175 590027 619280 637133 688206 736831 812409 902637 953007 965213 978
930 992684
Adjacency list for node 2773: 31155 83568 189894 195329 401909 892303 910746 73899 112065 117814 171290 247471 469738 627109 69787
3 865361
Adjacency list for node 2774: 11384 18906 62314 80947 116925 204253 222561 308592 337982 434152 648283 674756 770075 773293 827884
 847974 929725
Adjacency list for node 2775: 43737 107055 107756 282738 352038 410660 674562 680933 760823 782261 799075 827076 921024 940937 964
008
Adjacency list for node 2776: 15845 326249 448945 550274 582474 690974 817242 868123 127539 366244 666307 289352 590864 523403
Adjacency list for node 2777: 7787 35177 57863 151146 207688 217251 291784 343550 346068 404309 526306 560417 879875 94904 805320
Adjacency list for node 2778: 21051 310434 424555 581911 804279 859594 20442 585174 632765 687083 796656 1452
Adjacency list for node 2779: 61539 560179 708268 890066 893342 3176 420849 516880 588158 638576 891601
Adjacency list for node 2780: 145301 202438 215182 312606 332809 361555 427049 453950 471749 569382 588382 671422 752975 775808 99
7817
Adjacency list for node 2781: 91088 120494 123841 747748 833786 156225 227943 380092 411207 414785 433779 626298 713967 731705
Adjacency list for node 2782: 23686 56661 268578 301990 432453 491840 499675 621996 777614 815699 825180 893725 935355 413832 9406
10 14745 148933 781788
Adjacency list for node 2783: 152126 283958 484189 518751 588384 82173 180823 269430 306991 590968 730759 753965 809871 846486 928
763
Adjacency list for node 2784: 113998 125201 319185 387129 421757 429626 529587 554041 555479 611453 612861 704976 58120
Adjacency list for node 2785: 853884 72446 445199 776485 812207 492420 639342
Adjacency list for node 2786: 161245 279979 382978 603683 652516 779444 65333 263969 368318 565153 659698 779034 808598
Adjacency list for node 2787: 5684 119953 147758 243579 264152 326919 660436 728802 772795 790269 802851 1965
Adjacency list for node 2788: 36865 244497 622162 639073 739312 808465 976454
Adjacency list for node 2789: 300896 778181 900533 126713 302093 325928 367335 372870 478752 552129 715007 750587 909960
Adjacency list for node 2790: 551205 376793 968076 126370 773066 783592 835322 901036 996990 488121 723693 932823 940521
Adjacency list for node 2791: 181373 770926 23930 272107 708871 758166 156529 349423 196285 506017 705648 754147
Adjacency list for node 2792: 33634 57258 121235 130604 207501 230045 316965 364835 390040 400907 526151 534838 645702 711198 7526
55 803980 852199 979251
Adjacency list for node 2793: 235032 492195 497311 750933 831478 844577 16994 184621 215306 263922 376149 433026 453360 686628 731
145 757445 759391 862927 949646
Adjacency list for node 2794: 58122 262133 304375 341056 355103 428969 457035 481757 515193 553383 565090 623041 654011 699653 873
639 918532 920998 978075
Adjacency list for node 2795: 444618 766886 287616 298701 484675 544302 869839 898890 987207 85443 492746 516212 697974 744698 771
231
```
Figure 9: printAdjList() output from largeG.txt



The graph above displays the difference in execution time of the breadth-first search algorithm when it is called on different starting nodes within the medium and large text files. The largeG text file represents four thousand times as many nodes within the constructed graph. Therefore, the algorithm must traverse many more edges between these nodes to find the desired path. The time complexity of this algorithm is represented by O(V+E) where V is the number of vertexes (or nodes) and E is the number of edges. This is supported by the above graph, as the number of represented vertices and edges in largeG is many more than that of medium, causing the time complexity of the algorithm to increase dramatically. There is also a very slight difference in the execution time between the two different starting nodes in each text file. This could be due to the resulting differences in the path taken by the algorithm which results in a minor change in the execution time for the algorithm in mediumG and a slightly larger but still ultimately negligible change in execution time in largeG.

4. Sources:

https://chat.openai.com/share/8b4c2e60-b4e3-4b2a-909b-4a3300ec4287
https://chat.openai.com/share/3615b67f-45b3-411b-80d5-4ac6984224e1
https://www.geeksforgeeks.org/convert-adjacency-matrix-to-adjacency-list-representation-of-graph/
https://www.youtube.com/watch?v=X1LdtRW88c0
https://stackoverflow.com/questions/26448352/counting-the-number-of-lines-in-a-text-file-java
https://stackoverflow.com/questions/5262308/how-do-implement-a-breadth-first-traversal