Breadth First Search Algorithm on Undirected Graphs using Adjacency Lists

Lab # 8

By

Kamila Jusino

CS 303 Algorithm and Data Structure

November 1, 2024

Problem Specification

The goal of this assignment was to create an undirected graph. This was done using adjacency lists. In this assignment we needed to implement a BFS algorithm to go through the graph. In the program a driver needed to read through two separate files (medium.txt and large.txt.) The also calls to functions to help construct the graph. In the output the adjency list is shown to show the algorithms correctness as well as the paths. In the BFS algorithm, the pseudocode was used. It

chooses a vertex (0) in the case and as mentioned and prints from the vertex to the other nodes. In the lab report, an analysis is done to portray the performance of the algorithm.

Program Design

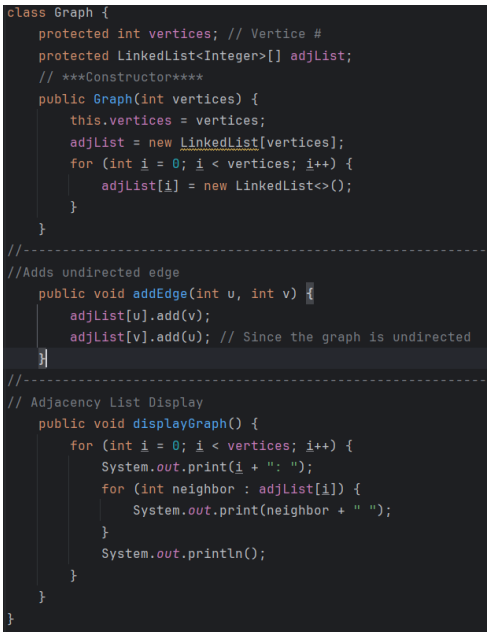
```
class BFSGraph extends Graph {
    private static final int WHITE = 0;
    private static final int GRAY = 1;
    private static final int BLACK = 2;

    public BFSGraph(int vertices) { super(vertices); }
//-----------------------------------------------------------------------
// BFS Algorithm per pseudocode
    public BFSResult bfs(int startVertex) {
        BFSResult result = new BFSResult(super.vertices);
        LinkedList<Integer> queue = new LinkedList<>();
        int[] color = new int[super.vertices]; // Tracks the colors
        Arrays.fill(color, WHITE); // white = all vertices
        result.getDistance()[startVertex] = 0;
        result.getPredecessor()[startVertex] = -1;
        color[startVertex] = GRAY; // Gray = start vertex
        queue.add(startVertex);
        while (!queue.isEmpty()) {
            int u = queue.poll();
            for (int v : adjList[u]) {
                if (color[v] == WHITE) { // White = process
                    color[v] = GRAY; // Gray = in queue
                    result.getDistance()[v] = result.getDistance()[u] + 1;
                    result.getPredecessor()[v] = u;
                    queue.add(v);
                }
            }
            color[u] = BLACK; // u = fully processed
        }
        return result;
    }
```

```
//-----------------------------------------------------------------------
// Prints Paths
    public void printPath(int startVertex, int endVertex, BFSResult result) {
        if (endVertex == startVertex) {
            System.out.print(startVertex + " ");
        } else if (result.getPredecessor()[endVertex] == -1) {
        } else {
            printPath(startVertex, result.getPredecessor()[endVertex], result);
            System.out.print(endVertex + " ");
        }
    }
}
```

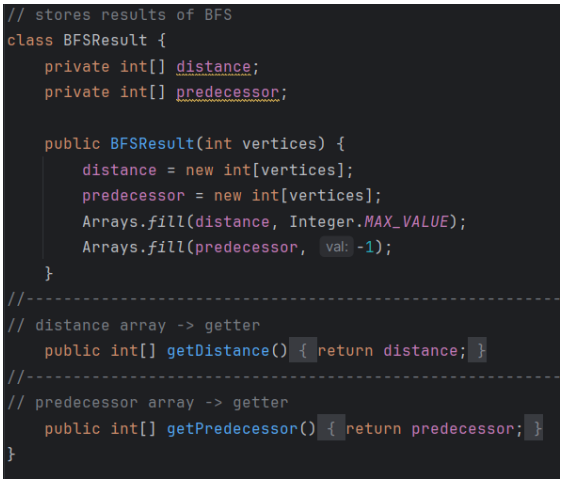
BFSGraph.java:

This code extends to Graph.java. This class performs the assignments instructed BFS algorithm. This is done on an undirected graph. The algorithm moves through the graph and stores what color each vertex is. The colors are dependent on “stage” so they might be white, gray, or black (process, in queue, fully processed.) It all works together by keeping a queue of what still needs

to be checked. All of this is stored in BFSResult. In this specific code the shortest distance from the starting vertex to each other is stored in the object. Print Path helps recursively print the starting vertex to any endpoint vertex using the arrays for path tracing.



```java
class Graph {
    protected int vertices; // Vertice #
    protected LinkedList<Integer>[] adjList;
    // ***Constructor****
    public Graph(int vertices) {
        this.vertices = vertices;
        adjList = new LinkedList[vertices];
        for (int i = 0; i < vertices; i++) {
            adjList[i] = new LinkedList<>();
        }
    }
//-------------------------------------------------------
//Adds undirected edge
    public void addEdge(int u, int v) {
        adjList[u].add(v);
        adjList[v].add(u); // Since the graph is undirected
    }
//-------------------------------------------------------
// Adjacency List Display
    public void displayGraph() {
        for (int i = 0; i < vertices; i++) {
            System.out.print(i + ": ");
            for (int neighbor : adjList[i]) {
                System.out.print(neighbor + " ");
            }
            System.out.println();
        }
    }
}
```



```java
// stores results of BFS
class BFSResult {
    private int[] distance;
    private int[] predecessor;

    public BFSResult(int vertices) {
        distance = new int[vertices];
        predecessor = new int[vertices];
        Arrays.fill(distance, Integer.MAX_VALUE);
        Arrays.fill(predecessor, val: -1);
    }
//-------------------------------------------------------
// distance array -> getter
    public int[] getDistance() { return distance; }
//-------------------------------------------------------
// predecessor array -> getter
    public int[] getPredecessor() { return predecessor; }
}
```

Graph.java

This snippet includes two main classes. BFSResult and Graph. The graph class “creates” the undirected graph using the adjacency lists. It also includes a couple helper functions. A constructor was created to help the functionality of the code. It sets the number of vertices. It also includes add Edge which adds edges between vertexes. Display Graph prints the lists for each vertex. BFSResult store the results from BFS with an array for tracking the distance. It also contains predecessor which stores previous paths between vertex. The helper functions get

Distance and get Predecessor allow into the array. This class helps store the results from the algorithm to more efficiently portray the results.

```
public class Main {
    public static void main(String[] args) {
// ------------------------------------------------------------------------------
// FILES names below ->
        String[] fileNames = {"src/mediumG.txt", "src/largeG.txt"};
// ------------------------------------------------------------------------------
// Reads FILES
        for (String fileName : fileNames) {
            System.out.println("File: " + fileName);
// ------------------------------------------------------------------------------
// Time Started!
            long startTime = System.nanoTime();
            try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
// ------------------------------------------------------------------------------
// Reads vertices + edges
                int vertices = Integer.parseInt(br.readLine().trim());
                int edges = Integer.parseInt(br.readLine().trim());
// ------------------- ----------------------------------------------------------
// Graph Started
                BFSGraph graph = new BFSGraph(vertices);

// ------------------- Adding Edges to the Graph ----------------------------------
// Edges -> Graph
                for (int i = 0; i < edges; i++) {
                    String[] edge = br.readLine().split( regex: " ");
                    int u = Integer.parseInt(edge[0]);
                    int v = Integer.parseInt(edge[1]);
                    graph.addEdge(u, v);
                }
// Prints Adjecency Lists
                System.out.println("\nAdjacency List:");
                graph.displayGraph();

// ------------------------------------------------------------------------------
// Runs BFS Algorithm
                int startVertex = 0;
                BFSResult bfsResult = graph.bfs(startVertex);
// ------------------------------------------------------------------------------
// Prints Paths
                System.out.println("\nBFS Paths from vertex " + startVertex + ":");
                for (int i = 0; i < vertices; i++) {
                    System.out.print("Path to " + i + ": ");
                    graph.printPath(startVertex, i, bfsResult);
                    System.out.println();
                }
// ------------------------------------------------------------------------------
// Time Ended -> Time Formula
                long endTime = System.nanoTime();
                double duration = endTime - startTime;
// ------------------- ------------------------------------------------------------
// Prints time
                System.out.println("\nFinished " + fileName);
                System.out.printf("Running Time: %.2f ns\n\n", duration);
            } catch (IOException e) {
                System.out.println("Error reading file: " + fileName);
                e.printStackTrace();
```

Main.java (Driver):

This snippet reads in the files provided to us (medium.txt, large.txt). It also performs the algorithm and prints the results. In this code each file is read for the vertex and edges. With the information gathered it starts BFSGraph which reads the edges to add them to the list. This is done with the help of the addEdge function. DisplayGraph is called to print the lists. BFS starts from vertex 0 and results BFSResult. With the printPath function the the results are printed and

well as the calculation for how long it took to process through the file. Error handling was added during the testing phase to implement good coding habits.

Test Cases

For the testing phase I foocused on the implementation of the pseudocode and debugged my way through ensuring it was functioning. I used smaller numbers and hardcoded a bit to further ensure the program. Once the program felt relatively close the main class implemented medium.txt. Print statements were added to show the entire process of the algorithm. Once that was re-worked out, large.txt was tested.

Analysis and Conclusions

File	Running Time
Medium.txt	53211200.00 ns

In conclusion, large.txt was not used for the lab report but can still be tested through the code. It does take a while to fully run through the entire file. Medium.txt does relatively well with a time of 53211200.00 ns as can be seen in the graph above. As we know, BFS hold a best, average, and worst time complexity of O(V+E). V is a representation of the number of vertices while E is the number of edges. BFS means breadth-first which occurs a “linear-time complexity proportional to the sum of vertices and edges.” (geeksforgeeks.) In true analysis, using distance information allowed the algorithm to ensure that it was indeed using the shortest path as well as going through all nodes. Adjency list provided a clear structure for the graph. All these

components together provided major insight into the direction the algorithm was going and in ensuring that it indeed was doing what the algorithm should be doing.

```
Adjacency List:
0: 15 24 44 49 58 59 68 80 97 114 149 160 163 176 191 202 204 209 211 222 225
1: 72 107 130 150 164 189 194 200 203 220
2: 14 18 42 51 79 86 108 110 141
3: 37 45 67 76 115 153 228 241
4: 5 26 55 77 78 112 128 138 159 239 240
5: 26 32 55 67 77 102 104 217 226 4
6: 16 54 98 99 117 129 140 147 166 178 236
7: 42 57 65 71 101 125 148 157 181 184 188 197 230
8: 11 30 43 82 85 143 152 179 207 210 212 221 244 246
9: 23 33 58 68 114 142 195
10: 105 106 123 175 246
11: 30 43 82 85 143 152 175 207 212 244 246 8
12: 28 35 36 41 88 94 113 121 170 182 198 242
13: 19 100 103 129 133 162 174 192
14: 18 51 86 129 133 166 2
15: 24 39 49 58 66 80 114 149 163 202 204 209 211 222 225 0
16: 54 98 99 117 129 140 147 166 178 236 6
17: 41 81 121 134 158 170 182 223 229
18: 35 51 86 94 141 14 2
19: 70 79 84 100 103 174 179 192 243 13
20: 40 75 89 116 127 164 190 194 220 247
```

```
BFS Paths from vertex 0:
Path to 0: 0
Path to 1: 0 160 187 77 78 112 234 130 1
Path to 2: 0 44 168 208 65 157 42 2
Path to 3: 0 44 144 232 45 3
Path to 4: 0 160 187 77 4
Path to 5: 0 160 32 5
Path to 6: 0 44 168 208 65 71 135 86 14 129 6
Path to 7: 0 44 168 208 65 7
Path to 8: 0 68 165 171 92 122 207 8
Path to 9: 0 58 9
Path to 10: 0 68 165 171 92 122 207 244 246 10
Path to 11: 0 68 165 171 92 122 207 11
Path to 12: 0 44 93 226 138 233 90 113 12
Path to 13: 0 68 165 171 92 122 214 79 174 13
Path to 14: 0 44 168 208 65 71 135 86 14
Path to 15: 0 15
Path to 16: 0 44 168 208 65 71 135 86 14 129 16
Path to 17: 0 44 93 226 138 233 90 113 121 17
Path to 18: 0 44 168 208 65 71 135 141 18
Path to 19: 0 68 165 171 92 122 214 79 19
Path to 20: 0 160 32 5 55 136 87 194 20
Path to 21: 0 44 93 226 138 21
Path to 22: 0 44 93 226 138 233 90 113 158 229 120 22
```

```
Finished src/mediumG.txt
Running Time: 53211200.00 ns
```

References

Running Time Help Credit:

StackOverFlow

https://stackoverflow.com/questions/5204051/how-to-calculate-the-running-time-of-my-program

BroCode

https://www.youtube.com/watch?v=dOYieTlItMM

File Input Help Credit:

https://chatgpt.com/share/66ee3d6a-a7b8-8011-90b6-abe7e6c76210

GeeksforGeeks

https://www.geeksforgeeks.org/different-ways-reading-text-file-java/

StackOverFlow

https://stackoverflow.com/questions/69356108/accessing-a-txt-file-in-the-src-folder

Proffesor Unan Slide Show

BFS/Adjacency Help Credit:

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

https://www.youtube.com/watch?v=6rDKLqFLfh0

https://www.youtube.com/watch?v=HZ5YTanv5QE&t=35s

https://www.youtube.com/watch?v=oDqjPvD54Ss

https://www.geeksforgeeks.org/adjacency-list-meaning-definition-in-dsa/

https://www.programiz.com/dsa/graph-adjacency-list

https://stackoverflow.blog/2022/05/26/the-complete-beginners-guide-to-graph-theory/

https://stackoverflow.com/questions/2505431/breadth-first-search-and-depth-first-search

Array Help Credit:

GeeksforGeeks

https://www.geeksforgeeks.org/arrays-in-java/

Professor Unan Slide Show

https://stackoverflow.com/questions/5785745/make-copy-of-an-array

https://www.youtube.com/watch?v=_86FMWNfGOc

https://chatgpt.com/share/66ee3ea9-5200-8011-97c5-fcdb0380596e

Extra

https://www.geeksforgeeks.org/polymorphism-in-java/

https://www.w3schools.com/java/java_inheritance.asp