Lab Assignment #8
Graph BFS

Full Name: Shreyas Srinivasa
BlazerId: SSRINIVA

CS 303 Algorithms and Data Structures

Spring 2023

1. Problem Specification

The purpose of this project was to create a programmatic representation of an undirected graph using the adjacency list as the data structure. All the details concerning the graph were read from a file and constructed according to the specifications. This graph was further put through a version of the breadth first search (BFS) algorithm which calculated parent nodes of all the vertices and distance from a singular source passed as one of the input parameters. This additional information was further used to detect the presence of a path between the chosen source and rest of the vertices of the graph. The real time performance was further measured with respect to execution time.

2. Program Design

```
public Node get(int i) {
    return vertices[i];
}

public void insert(int v1, int v2) {

    if (vertices[v1] == null)
        vertices[v1] = new Node((long) v1);
    if (vertices[v2] == null)
        vertices[v2] = new Node((long) v2);

    graph.get(v1).add(vertices[v2]);

}
```

The functions for retrieving a single vertex from the list of vertices and inserting a new vertex by specifying its neighbor.

```
public void bfs(Node s) {
    for (Node u : vertices) {
        Node v = u;
        if (!v.equals(s)) {
            v.setColor("WHITE");
            v.setD(null);
            v.setP(null);
        }
    }
    s.setColor("GRAY");
    s.setD(0l);
    s.setP(null);
    Queue<Node> Q = new ArrayDeque<>();
    Q.add(s);
    while (Q.size() > 0) {
        Node u = Q.remove();
        for (Node v: graph.get(Math.toIntExact(u.getValue()))) {
            if (v.getColor().equalsIgnoreCase("WHITE")) {
                v.setColor("GRAY");
                v.setD(u.getD() + 1);
                v.setP(u);
                Q.add(v);
            }
        }
        u.setColor("BLACK");
    }
}
```

The implementation of a variant of the BFS algorithm that stores the parent vertex and distance from a source within the Node object that represents a single vertex.

```
public void printPath(Node s, Node v) {
    if (v.equals(s)) {
        System.out.print("\n" + s.getValue());
    } else if (v.getP() == null) {
        System.out.format("\n%d X %d", s.getValue(), v.getValue());
    } else {
        printPath(s, v.getP());
        System.out.format(" -> %d", v.getValue());
    }
}

public Node[] getAllVertices() {
    return vertices;
}
```

The function which prints the path between the source and any vertex if exists.

```
public void print() {
    for (int i = 0; i < graph.size(); i++) {
        System.out.format("%d -> ", i);
        for (Node v : graph.get(i)) {
            System.out.format("%d -> ", v.getValue());
        }
        System.out.print("/ \n");
    }
}
```

The function to print the entire adjacency list that holds the graph information

```
public static Graph getGraphFromFile(String filename) {
    Graph g = null;
    try (Scanner fileScanner = new Scanner(Paths.get(filename))) {
        Integer vertices = fileScanner.nextInt();
        g = new Graph(vertices);
        Integer edges = fileScanner.nextInt();
        fileScanner.nextLine();
        while (fileScanner.hasNextLine()) {
            String line = fileScanner.nextLine();
            String elements[] = line.split(" ");
            g.insert(Integer.valueOf(elements[0]), Integer.valueOf(elements[1]));
        }
        fileScanner.close();
    } catch (Exception e) {
        System.out.println("Error: " + e.toString());
        System.exit(1);
    }
    return g;
}
```

The function which reads all the graph data from the passed filename and return a Graph object

```
public static void driver(String filename) {
    Instant start, finish;
    long timeElapsed;

    Graph g = getGraphFromFile(filename);
    g.print();

    start = Instant.now();
    g.bfs(g.get(0));
    finish = Instant.now();
    timeElapsed = Duration.between(start, finish).toNanos();

    System.out.format("Time taken for BFS:= %d\n", timeElapsed);

    for(Node v: g.getAllVertices()) {
        g.printPath(g.get(0), v);
    }
}
```

The function to perform the BFS with a chosen source and print all the paths

3. **Output**

```
0 -> 15 -> 24 -> 44 -> 49 -> 58 -> 59 -> 68 -> 80 -> 97 -> 114 -> 149 -> 160 -> 163 -> 176 -> 191 -> 202 -> 204 -> 209 -> 211 -> 222
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 -> /
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 -> /
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 -> /
15 -> 24 -> 39 -> 49 -> 58 -> 66 -> 80 -> 114 -> 149 -> 163 -> 202 -> 204 -> 209 -> 211 -> 222 -> 225 -> /
16 -> 54 -> 98 -> 99 -> 117 -> 129 -> 140 -> 147 -> 166 -> 178 -> 236 -> /
17 -> 41 -> 81 -> 121 -> 134 -> 158 -> 170 -> 182 -> 223 -> 229 -> /
18 -> 35 -> 51 -> 86 -> 94 -> 141 -> /
19 -> 70 -> 79 -> 84 -> 100 -> 103 -> 174 -> 179 -> 192 -> 243 -> /
20 -> 40 -> 75 -> 89 -> 116 -> 127 -> 164 -> 190 -> 194 -> 220 -> 247 -> /
21 -> 27 -> 62 -> 65 -> 71 -> 138 -> 184 -> 188 -> 230 -> 233 -> 240 -> /
22 -> 34 -> 53 -> 56 -> 73 -> 120 -> 145 -> /
23 -> 33 -> 58 -> 68 -> 114 -> 176 -> 195 -> 222 -> /
24 -> 39 -> 66 -> 80 -> 114 -> 149 -> 163 -> 206 -> 209 -> 211 -> 222 -> 225 -> /
25 -> 60 -> 63 -> 96 -> 111 -> 199 -> /
26 -> 55 -> 77 -> 78 -> 102 -> 138 -> 217 -> 226 -> 239 -> 240 -> /
27 -> 62 -> 65 -> 71 -> 138 -> 184 -> 188 -> 230 -> 233 -> 240 -> /
28 -> 35 -> 41 -> 94 -> 113 -> 121 -> 170 -> 182 -> 198 -> 223 -> 242 -> /
29 -> 47 -> 64 -> 91 -> 109 -> 137 -> 146 -> 167 -> 218 -> 224 -> 227 -> /
30 -> 43 -> 70 -> 79 -> 82 -> 143 -> 152 -> 156 -> 179 -> 207 -> 210 -> 212 -> 214 -> 219 -> 221 -> 244 -> /
31 -> 37 -> 115 -> 153 -> 228 -> 241 -> /
32 -> 52 -> 77 -> 93 -> 102 -> 104 -> 144 -> 151 -> 160 -> 168 -> 185 -> 187 -> 201 -> 208 -> 226 -> 231 -> 248 -> /
33 -> 58 -> 114 -> 163 -> 222 -> /
34 -> 53 -> 56 -> 73 -> 120 -> 145 -> /
```

Output 1 The adjacency list graph representation printed out on the console

```
Time taken for BFS:= 2660058

0
0 X 1
0 X 2
0 X 3
0 X 4
0 X 5
0 X 6
0 X 7
0 X 8
0 X 9
0 X 10
0 X 11
0 X 12
0 X 13
0 X 14
0 -> 15
0 X 16
0 X 17
0 X 18
0 X 19
0 X 20
0 X 21
0 X 22
0 X 23
0 -> 24
0 X 25
0 X 26
0 X 27
0 X 28
```

Output 2. The paths printed from a source to all the vertices in the graph. X represents an absence of an edge between the two vertices

```
0 X 215
0 -> 44 -> 144 -> 201 -> 216
0 -> 44 -> 144 -> 201 -> 217
0 X 218
0 X 219
0 X 220
0 X 221
0 -> 222
0 X 223
0 X 224
0 -> 225
0 -> 44 -> 93 -> 226
0 X 227
0 X 228
0 X 229
0 -> 68 -> 165 -> 172 -> 197 -> 230
0 -> 44 -> 231
0 -> 44 -> 144 -> 232
0 X 233
0 X 234
0 -> 68 -> 165 -> 171 -> 235
0 X 236
0 X 237
0 -> 68 -> 165 -> 171 -> 238
0 X 239
0 X 240
0 X 241
0 X 242
0 X 243
0 X 244
0 -> 68 -> 165 -> 171 -> 245
0 X 246
0 X 247
0 -> 44 -> 248
0 X 249
```

Output 3. The paths printed from a source to all the vertices in the graph. X represents an absence of an edge between the two vertices

4. Analysis and Conclusions

Analysis from GeeksForGeeks

Time Complexity: O(V+E)
Where V is the number of vertices and E is the number of edges in the graph.

Space Complexity: O(V)
We used an array of size V to store the BFS traversal. We also used an array of size V to keep track of visited vertices. We used a queue of size V to store the vertices.

Applications of BFS:

- *Shortest Path and Minimum Spanning Tree for unweighted graph*: In an unweighted graph, the shortest path is the path with the least number of edges. With Breadth First, we always reach a vertex from a given source using the minimum number of edges. Also, in the case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
- *Peer-to-Peer Networks*: In Peer-to-Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
- *Crawlers in Search Engines*: Crawlers build an index using Breadth First. The idea is to start from the source page and follow all links from the source and keep doing the same. Depth First Traversal can also be used for crawlers, but the advantage of Breadth First Traversal is, the depth or levels of the built tree can be limited.
- *Social Networking Websites*: In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
- *GPS Navigation systems*: Breadth First Search is used to find all neighboring locations.
- *Broadcasting in Network*: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
- *In Garbage Collection*: Breadth First Search is used in copying garbage collection using Cheney's algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of the better locality of reference:
- *Cycle detection in the undirected graph*: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. We can use BFS to detect cycle in a directed graph also,
- *Ford–Fulkerson algorithm*: In the Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst-case time complexity to $O(VE^2)$.
- *To test if a graph is Bipartite*: We can either use Breadth First or Depth First Traversal.
- *Path Finding*: We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
- *Finding all nodes within one connected component*: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Advantages of Breadth First Search:

- BFS will never get trapped exploring the useful path forever.
- If there is a solution, BFS definitely find it out.
- If there is more than one solution then BFS can find the minimal one that requires less number of steps. If there is a solution then BFS is guaranteed to find it.
- Low storage requirement: linear with depth.
- Easily programmed.

Disadvantages of Breadth First Search:

- The main drawback of BFS is its memory requirement. Since each level of the tree must be saved in order to generate the next level and the amount of memory is proportional to the number of nodes stored the space complexity of BFS is $O(b^d)$. As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.

As we already know from previous labs:-

The Time Complexity of an algorithm/code is **not** equal to the actual time required to execute a particular code, but the number of times a statement executes.

The **actual time required to execute code is machine-dependent**.

Instead of measuring actual time required in executing each statement in the code,Time Complexity considers how many times each statement executes.

Observations based on the lab experiment

- By executing the search with both mediumG.txt and largeG.txt, it is evident that the time taken for execution is directly proportional to the number of vertices and edges.
- Since our algorithm traverses the entire graph, it does use O(V) space. With the increase in the number of vertices the auxiliary space requirements will also drastically increase, along with the space required for storing the graph itself.
- The version of BFS used in this experiment is used to find the shortest path in an unweighted graph. The d property in each Node object to keep track of the distance from the source vertex. This is updated whenever we use an edge. At the same time, we also keep a p property to tell us what the parent of a vertex is in the shortest path that we have found using BFS. This allows us to compute the actual shortest path that we have found.

5. References

- https://web.archive.org/web/20171118044139/http://www.cs.cornell.edu/~wdtseng/icpc/notes/graph_part1.pdf
- https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
- https://medium.com/@yuhuan/covariance-and-contravariance-in-java-6d9bfb7f6b8e
- https://www.simplilearn.com/tutorials/data-structure-tutorial/bfs-algorithm
- https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/