Tamir Krief, Iaian Milton, Blessing Abumere
COSC 336
11/26/2024

Assignment 8

Exercise 1:

(a) When $m \leq 3n$, Variant (a)'s runtime will be $O(n^2)$ and Variant (b)'s runtime will be $O(n \log n)$. Since an $O(n \log n)$ runtime can handle larger values of n more efficiently, this means that Variant (b)'s runtime is faster than Variant (a)'s in this example with $m \leq 3n$.

(b) When $m \geq \frac{n^2}{3}$, Variant (a)'s runtime will be $O(n^2)$ and Variant (b)'s runtime will be $O(n^2 \log n)$. Since $O(n^2)$ grows faster than $O(n^2 \log n)$, this makes Variant (a)'s runtime faster than Variant (b)'s in this example with $m \geq \frac{n^2}{3}$.

(c) When $m = n^{\frac{3}{2}}$, Variant (a)'s runtime will be $O(n^2)$ and Variant (b)'s runtime will be $O(n^{\frac{3}{2}} \log n)$. Since an $O(n^{\frac{3}{2}} \log n)$ runtime can handle larger values of n more efficiently, this means that Variant (b)'s runtime is faster than Variant (a)'s in this example with $m = n^{\frac{3}{2}}$.

Exercise 2:

Analyzing the first case of a DAG ($u.d < v.d$), u will be discovered first, then the DFS will explore all possible reachable vertices from u before discovering v. When DFS explores the path from u to v, u cannot finish until all its descendents, including v, are finished, which makes v the first to finish making $u.f > v.f$.

For the second case of a DAG ($v.d < u.d$), v will be discovered before u. However, there is still a path from u to v, meaning v cannot finish until all vertices connecting to u leading to v have been explored, making $u.f > v.f$ in this case as well.

Whether u is discovered before or after v in a DAG, $u.f > v.f$ is consistent for both cases.

Programming Task 8

[1] based indexing for all

Graph G1 results:

Adjacency list of vertex1 head $\rightarrow 2 \rightarrow 3 \rightarrow 4$
Adjacency list of vertex2 head $\rightarrow 1 \rightarrow 5$
Adjacency list of vertex3 head $\rightarrow 1 \rightarrow 5$
Adjacency list of vertex4 head $\rightarrow 1 \rightarrow 6$
Adjacency list of vertex5 head $\rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 7$
Adjacency list of vertex6 head $\rightarrow 4 \rightarrow 5 \rightarrow 7$
Adjacency list of vertex7 head $\rightarrow 5 \rightarrow 6$

Adjacency Matrix:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

(3) Shortest paths of length (3) from 1 to 7

index	**npath[]**	**dist[]**
1	1	0
2	1	1
3	1	1
4	1	1
5	2	2
6	1	2
7	3	3

Graph G2 results:

Adjacency list of vertex1 head $\rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$
Adjacency list of vertex2 head $\rightarrow 1 \rightarrow 7$
Adjacency list of vertex3 head $\rightarrow 1 \rightarrow 7$
Adjacency list of vertex4 head $\rightarrow 1 \rightarrow 7$
Adjacency list of vertex5 head $\rightarrow 1 \rightarrow 7$
Adjacency list of vertex6 head $\rightarrow 1 \rightarrow 7$
Adjacency list of vertex7 head $\rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 9$
Adjacency list of vertex8 head $\rightarrow 7 \rightarrow 10$
Adjacency list of vertex9 head $\rightarrow 7 \rightarrow 10$
Adjacency list of vertex10 head $\rightarrow 8 \rightarrow 9$

Adjacency Matrix:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

(10) Shortest paths of length (4) from 1 to 10

index	**npath[]**	**dist[]**
1	1	0
2	1	1
3	1	1
4	1	1
5	1	1
6	1	1
7	5	2
8	5	3
9	5	3
10	10	4

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.Queue;

class Assignment8 {
    public static void main(String[] args) {
        // graphs are all undirected
        Adj_List_Graph G1 = graph_G1();
        Adj_List_Graph G2 = graph_G2();

        System.err.print("Graph G1 results:");
            G1.printGraph();
            G1.print_AdjacencyMatrix();
            BFS(G1, 1);

        System.err.print("\nGraph G2 results:");
            G2.printGraph();
            G2.print_AdjacencyMatrix();
            BFS(G2, 1);
    }

    /**
     * Breadth First Search sourced from notes 10
     * Program is required to <b>print</b> the two arrays dist[] and npath[].
     *
     * @param G the graph <br>
     * @param S the starting node with respect to the graphs <b>INDEX_OFFSET<b><br>
     * @return 2D array where index <b>0</b> is {@code dist[]} and index <b>1</b> is{@code npath[]}
     * @throws NullPointerException if graph is null
     */
    public static int[][] BFS(Adj_List_Graph G, final int S) throws NullPointerException {
        final int s = S - (G.INDEX_OFFSET); // the true starting index calculated from the index
offset

        if (s < 0 || s >= G.n) {
            System.err.printf("Starting Node (%d) is out of range: [%d, %d]\n", S, G.INDEX_OFFSET,
                G.n + 1 - G.INDEX_OFFSET);
            return null;
        }

        final int UNSEEN = Integer.MIN_VALUE; // used like infinity in notes 10
        final int N = G.n; // size

        int[] dist = new int[N]; // keeps track of how far from the starting node and whether the
node has been seen ; dist[v] = length of shortest path from s to v
        int[] npath = new int[N]; // n amount of shortest paths; npath[v] = number of shortest paths
from s to v

        Queue<Integer> queue = new LinkedList<>(); // holds the nodes. nodes are queued and dequenced
once

        // all nodes set to unseen / infinity
        for (int v = 0; v < N; v++)
            dist[v] = UNSEEN;

        dist[s] = 0; // distance from S to S is 0
        npath[s] = 1; // paths from S to S is 1
        queue.add(s); // enqueueing s

        int u;
        while (!queue.isEmpty()) {
            u = queue.remove(); // javas dequeue
```

```
            // visit u
            for (int v : G.adj.get(u)) {
                if (dist[v] == UNSEEN) { // check if node has not been seen
                    queue.add(v); // javas enqueue
                    dist[v] = dist[u] + 1; // set node to seen and recording the distance
                    npath[v] = npath[u]; // path from v to u stays the same
                }
                else if (dist[v] == dist[u] + 1)
                    npath[v] += npath[u]; // number of shortest paths from v to u goes up by the
number of shortest paths from u to v ; 1 + npath[u]
            }
        }

        System.out.printf("\n[%d] Based Indexing:\n(%d) Shortest paths of length (%d) from %d to %d
\n",
            G.INDEX_OFFSET,
            npath[N - 1],
            dist[N - 1],
            S, (N - 1) + G.INDEX_OFFSET
        );

        System.out.println("dist[]: " + Arrays.toString(dist));
        System.out.println("npath[]: " + Arrays.toString(npath));

        return new int[][] { dist, npath }; // for testing
    }

    /** creates the undirected graph of G1 from assign8 */
    public static Adj_List_Graph graph_G1() {
        Adj_List_Graph G1 = new Adj_List_Graph(7, 1);

        // 1 based indexing
        G1.addEdge(1, 2); // 1 -> 2
        G1.addEdge(1, 3); // 1 -> 3
        G1.addEdge(1, 4); // 1 -> 4

        G1.addEdge(2, 5); // 2 -> 5

        G1.addEdge(3, 5); // 3 -> 5

        G1.addEdge(4, 6); // 4 -> 6
        G1.addEdge(5, 6); // 5 -> 6
        G1.addEdge(5, 7); // 5 -> 7

        G1.addEdge(6, 7); // 6 -> 7

        return G1;
    }

    /** creates the undirected graph of G2 from assign8 */
    public static Adj_List_Graph graph_G2() {
        Adj_List_Graph G2 = new Adj_List_Graph(10);

        // 1 based indexing
        G2.addEdge(1, 2);// 1 -> 2
        G2.addEdge(1, 3);// 1 -> 3
        G2.addEdge(1, 4);// 1 -> 4
        G2.addEdge(1, 5);// 1 -> 5
        G2.addEdge(1, 6);// 1 -> 6

        G2.addEdge(2, 7);// 2 -> 7

        G2.addEdge(3, 7);// 3 -> 7

        G2.addEdge(4, 7);// 4 -> 7
```

```
    G2.addEdge(5, 7);// 5 -> 7

    G2.addEdge(6, 7);// 6 -> 7

    G2.addEdge(7, 8);// 7 -> 8
    G2.addEdge(7, 9);// 7 -> 9

    G2.addEdge(8, 10);// 8 -> 10

    G2.addEdge(9, 10);// 9 -> 10

    return G2;

}

public static class Adj_List_Graph {
    int n; // no of nodes
    ArrayList<ArrayList<Integer>> adj;

    /**
     * Exists for display purposes; <br>
     * assignment 8 uses 1 based indexing <br>
     * <code>
     * INDEX_OFFSET = 0 would be 0 based indexing <br>
     * INDEX_OFFSET = 1 would be 1 based indexing <br>
     * etc
     * </code>
     */
    final int INDEX_OFFSET; // added to control indexing when displaying graphs for assignment 8

    // constructor taking as the single parameter the number of nodes
    Adj_List_Graph(int no_nodes, final int INDEX_OFFSET) {
        n = no_nodes;
        adj = new ArrayList<ArrayList<Integer>>(n);
        for (int i = 0; i < n; i++)
            adj.add(new ArrayList<Integer>());

        this.INDEX_OFFSET = INDEX_OFFSET; // this was added for displaying assignment 8
    }

    public Adj_List_Graph(int no_nodes) {
        this(no_nodes, 1); // INDEX_OFFSET is set to 1 by default for assignment 8
    }

    // A utility function to add an edge in an
    // undirected graph; for directed graph remove the second line
    // adjusted for assignment 8
    public void addEdge(final int U, final int V) {
        int u = U - (this.INDEX_OFFSET); // adjusted u
        int v = V - (this.INDEX_OFFSET); // adjusted v
        adj.get(u).add(v);

        // undirected for assignment 8
        adj.get(v).add(u); // this line should be un-commented, if graph is undirected
    }

    /** A utility function to print the adjacency list representation of graph */
    // adjusted for assignment 8
    public void printGraph() {
        for (int i = 0; i < n; i++) {
            System.out.print("\nAdjacency list of vertex" + (i + this.INDEX_OFFSET) + "\thead");
            for (int j : adj.get(i)) {
                System.out.print(" -> " + (j + this.INDEX_OFFSET));
            }
```

```
            }
        }

        /** A utility function to print the adjacency matrix representation of graph */
        public void print_AdjacencyMatrix() {
            System.out.println("\n\nAdjacency Matrix:");
            for (int u = 0; u < n; u++) {
                for (int v = 0; v < n; v++) {
                    System.out.printf(
                            "%d  ",
                            adj.get(u).contains(v) ? 1 : 0);
                }
                System.out.println();
            }
        }
    }
}
```