Romerico David, Haley Elliott, Julian Halsey, Nkechiyem Molokwu
COSC 336
05/07/2024

Assignment 8

Exercise 1:

(a). Variant (b) would give the fastest runtime. Variant (a) would run in $\mathcal{O}(n^2)$ time while variant (b) would only take $\mathcal{O}(nlog(n))$ time.

(b). Variant (a) would give the fastest runtime. Variant (a) would run in $\mathcal{O}(n^2)$ time while variant (b) would take $\mathcal{O}(n^2log(n))$ time.

(c). Variant (b) would give the fastest runtime. Variant (a) would run in $\mathcal{O}(n^2)$ time while variant (b) would take $\mathcal{O}(n^{3/2}log(n))$ time.

Exercise 2:

Let u and v be two distinct nodes in a DAG such that there exists a path from u to v. Since discovery time is distinct $u.d \neq v.d \therefore$ either $u.d < v.d$ or $u.d > v.d$

Case 1: $u.d < v.d$

Since there exists a path $u \rightarrow v$ and $u.d < v.d$, then v is a descendant of u, so (u, v) is a forward edge. Then $u.d < v.d => v.f < u.f$

Case 2: $v.d < u.d$

By definition of DFS with timing, $v.d < v.f$ and $u.d < u.f$. Since v is a descendant of u, u is not discovered before v is finished, so $v.f < u.d < u.f$ which shows $v.f < u.f$.

Since $u.d < v.d$ implies $v.f < u.f$ and $v.d < u.d$ implies $v.f < u.f$, then for any two nodes u and v such that there exists a path from u to v, it holds that $u.f > v.f$.

Programming Task:

Description of Algorithm: We utilize an adjacency list representation to manage two undirected graphs that performs a modified BFS that computes the shortest paths from some starting node. The modifiedBFS method calculates and outputs the shortest distance (dist) and the number of such paths

(npath) from a specified starting vertex to all other vertices. dist is initially filled with "∞" to denote that vertices are unvisited, and npath is set to zero. The search starts by setting the distance to the starting vertex s to zero and the number of paths to one, then it proceeds to explore the graph layer by layer. As vertices are dequeued, each of their unvisited neighbors are updated with the new distance and takes the path count of the current vertex. If a neighbor has already been visited and a shorter path is found through the current vertex, the path count for that neighbor is incremented by the path count of the current vertex. This process ensures dist[v] contains the shortest distance from the starting vertex to vertex v, and npath[v] contains the number of distinct shortest paths from the starting vertex to vertex v.

Code:

```
public class Assign8 {
    public static void main(String[] args) {
        Adj_List_Graph g1 = new Adj_List_Graph(7 + 1);
        g1.addEdge(1, 2);
        g1.addEdge(1, 3);
        g1.addEdge(1, 4);
        g1.addEdge(2, 5);
        g1.addEdge(3, 5);
        g1.addEdge(4, 6);
        g1.addEdge(5, 7);
        g1.addEdge(6, 7);

        Adj_List_Graph g2 = new Adj_List_Graph(10 + 1);
        g2.addEdge(1, 2);
        g2.addEdge(1, 3);
        g2.addEdge(1, 4);
        g2.addEdge(1, 5);
        g2.addEdge(1, 6);
        g2.addEdge(2, 7);
        g2.addEdge(3, 7);
        g2.addEdge(4, 7);
        g2.addEdge(5, 7);
        g2.addEdge(6, 7);
        g2.addEdge(7, 8);
```

```
        g2.addEdge(7, 9);
        g2.addEdge(8, 10);
        g2.addEdge(9, 10);

        modifiedBFS(g1, 1);
        System.out.println
            ("-----------------------------------");
        modifiedBFS(g2, 1);
    }

    public static void modifiedBFS(Adj_List_Graph g,
        int s) {
        int n = g.n;
        int[] dist = new int[n];
        int[] npath = new int[n];

        Arrays.fill(dist, Integer.MAX_VALUE);
        Arrays.fill(npath, 0);
        Queue<Integer> queue = new LinkedList<>();
        dist[s] = 0;
        npath[s] = 1;
        queue.add(s);

        while (!queue.isEmpty()) {
            int u = queue.poll();
            for (int v : g.adj.get(u)) {
                if (dist[v] == Integer.MAX_VALUE) { //
                    v has not been visited
                    dist[v] = dist[u] + 1;
                    npath[v] = npath[u];
                    queue.add(v);
                } else if (dist[v] == dist[u] + 1) //
                    Another shortest path found
                    npath[v] += npath[u];
            }
        }

        for (int i = 1; i < n; i++)
```

```
                System.out.println("Vertex " + i + ": dist
                    = " + dist[i] + ", npath = " + npath[i])
                    ;
        }
}
```

Results Table: Graph G1

Node	dist	npath
1	0	1
2	1	1
3	1	1
4	1	1
5	2	2
6	2	1
7	3	3

Table 1: Results for Graph G1

Results Table: Graph G2

Node	dist	npath
1	0	1
2	1	1
3	1	1
4	1	1
5	1	1
6	1	1
7	2	5
8	3	5
9	3	5
10	4	10

Table 2: Results for Graph G2