CS 331: Algorithms and Complexity (Spring 2024)
Unique Number: 52765, 52770

Assignment 2

Due on Thursday, 1 Febrauary, by 11.59pm

Problem 1: Short Answers Section

For this section, restrict answers to no more than a few sentences. Most answers can be expressed in a single sentence. Unless otherwise stated, briefly justify. No proofs are necessary for this section.

(a) True, since at first there is a root node with no edges. Every subsequent node adds one edge.

(b) False, there may be cycles. However, DAG $\rightleftarrows$ Topological Ordering.

(c) True, BFS would iterate every edge set of every vertex, so $|V| \cdot |V|$ edges.

(d) True, iterate through the node's row of the matrix, which has $|V|$ elements.

(e) False, DFS will in general output deeper trees, but not always. Counterexample: a tree with only a root node.

(f) True, if there were multiple paths, then that means there's a cycle. Trees have no cycles.

Problem 2

(a) True, we can alternate the colors in each layer of the tree. Then, every edge in the tree will be touching a node on layer_n and one on layer_{n+1}, which are different colors. An algorithm is to run BFS starting from the tree's root.

(b) Nodes can't connect to other nodes on the same layer, as that would create an odd length cycle. They also can't connect to nodes an even number of layers away, as that would connect two nodes of the same color. So, they can only connect to nodes an odd number of layers away. We know that each node of a color can connect to every node of the other color and that a tree has n - 1 edges.

```
def add_edges(graph):
    black = 0
    white = 0
    Initialize queue as storing nodes with their associated depth
    Add root node to queue with depth 0
    while queue is not empty:
        node, depth = queue.poll()
        if depth % 2 == 0:
            black += 1
        else:
            white += 1
        for neighbor in node.neighbors:
                queue.add(neighbor, depth + 1)
    return black * white - (n - 1)
```

Problem 3

I would run a breadth-first search, which has O($| V | + | E |$).

Proof. **Inductive hypothesis:** At step k, all possible routes to nodes of distance k are stored in the path_count variable of each node and the queue stores all nodes k distance away from the s.
Initialization: At step 0, the path_count of the start node is 1, and all other nodes are unvisited. The queue stores only s.
Maintenance: At step k, we have a list of nodes of distance k. For each node, it stores the distance from s and the number of paths to it. For each node, we iterate through its edges. If the edge's node has not been visited, we set its distance to k+1, and by BFS, it's the shortest distance. We also set its path_count to the current node's + the node's path_count. If the edge's node has been visited, we don't need to add it again to the queue, but we do need to increment its path_count by the current node's path_count. We do this for all nodes of distance k, and then we increment k.
Now the queue contains all nodes of distance k+1 and the number of unique paths to it .
Termination: The algorithm terminates when the target node's distance is greater than the current node's distance. At this point, we return the target node's path_count. □

```
def bfs(graph, start):
    Initialize all nodes to path_count = 0 and distance = infinity
    Initialize s to path_count = 1 and distance = 0
    Initialize queue to list(s)
    while queue is not empty:
        node = queue.poll()
        if node.distance > t.distance:
```

```
        break
    for neighbor in node.neighbors:
        if neighbor.distance == infinity:
            neighbor.distance = node.distance + 1
            neighbor.path_count = node.path_count
            queue.add(neighbor)
        else if neighbor.distance == node.distance + 1:
            neighbor.path_count += node.path_count
return t.path_count
```