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) It is $\theta(|V|)$ to iterate through the node's row of the matrix, which has $|V|$ elements. It is thus $\theta(|V^2|)$ to iterate the entire matrix with all the edges.

(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. We alternate the colors of the nodes at each layer. Since we visit every node, the algorithm will run in O(n).

(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. An algorithm would be to count up the number of nodes of each color by going through the tree starting from the root using BFS. Each layer will alternate colors, and then we multiply the count of each color together. After, we would subtract (n - 1) to account for the original edges. This yields us the maximum number of edges we can add to keep the graph bipartite. As we have to iterate through every node, the algorithm will run in O(n).

Problem 3

I would run a breadth-first search, which has O($| V | + | E |$).
First, I would initialize all nodes to have a distance of infinity and a path_count of 0. Then, I would initialize the start node to have a distance of 0 and a path_count of 1. I would then initialize a queue with the start node.
While the queue is not empty, I would pop the first node from the queue. If the node's distance is greater than the target node's distance, I would break. Otherwise, I would iterate through the node's neighbors. If the neighbor's distance is infinity, I would set the neighbor's distance to the current node's distance + 1 and its path_count to the current node's path_count. I would then add the neighbor to the queue. If the neighbor 's distance is equal to the current node's distance + 1, I would increment the neighbor 's path_count by the current node's path_count. After the loop exits, I would then return the target node's path_count.

Proof. The queue stores each node only once and polls one each iteration, $\therefore$ the algorithm will terminate.
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. The queue stores only s, which is correct.
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 neighbors. If the neighbor has not been visited, i.e. distance = ∞, 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 won't add nodes already visited, since their distance wouldn't be k + 1. 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 them .
Termination: The algorithm terminates when the target node's distance is less than the current node's distance, which means we are done processing shortst paths to the target. At this point, we return the target node's path_count. □