Homework 3

Tags	Homework
Date	

Problem 1

Proof

- The tree that is obtained from a breadth first search of graph G is oriented in a manner where each node of G has an associated level that is determined by the shortest number of edges in between the s node to each node in G.
- Given that the distance between nodes s and t is strictly greater than n/2, that implies that shortest $path(s,t) = l_t > l_{n/2}$
- Since s is the root node, there does not exist exist any level from l_1 —> $l_{n/2}$ that has either the s or t node
- There cannot exist more than 2 nodes within each of these levels or the number of nodes in G would be greater n, thus there has to exist at least one level in between l_1 —> $l_{n/2}$ where there is one node v in its respective level l_v that connects edges l_{v-1} and l_{v+1} that exist in the shortest $path(s,t)$.
- Therefore if that node v were deleted, there would no longer exist a path from s to t

Algorithm

We want to find the level where there exists one node between levels 1 and n/2.

- set discovered[s] = true and discovered[v] = false for all other v
- initialize L[0] to consist of single element s
- set layer counter i = 0
- set current BFS tree T = null
- while L[i] is not empty
    - initialize empty list L[i+1]
    - for each node u in L[i]

        - consider each edge (u,v) incident to u
        - if discovered[v] = false then
            - set discovered[v] = true
            - add edge(u,v) to the tree T
            - add v to list L[i+1]
    - if L[i+1] > 0 and L[i+1] ≤ n/2 and length of L[i+1] = 1
        - return element in L[i+1] as v
    - increment i by 1

Time complexity

This algorithm uses a traditional BFS algorithm with an extra if statement that computes in O(1) time, hence the runtime of this algorithm is the runtime of BFS, that is O(m+n).

Problem 2

A binary tree is a rooted tree in which each node has at most two children.
Show by induction that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

Let n_2 = the number of nodes with 2 children, and l = number of leaves in a given tree

Base Case

Let's say T is a binary tree with 1 node, that is the root node. The root node has no children, and thus is considered a leaf node. Hence, $n_2 = 0$ and $l = 1$.

For the base condition of a binary tree, the statement retains true that there exists exactly one less node with 2 children than the number of leaves.

Induction Case

Assume there is a binary tree T that has a number of nodes with two children exactly one less than the number of leaves. We must prove that for each additional node w added to the tree, $n_2 + 1 = l$ retains true.

There are 3 cases to which the additional node is added to a node v:

1. v is a node with no children (thus, a leaf node). Therefore, adding a node to v results in
    a. n_2 = n_2 ⇒ the number of nodes with 2 children retain the same
    b. node v is no longer a leaf node, and node w is now a leaf node ⇒ $l + 1 - 1 = l$ ⇒ number of leaf nodes retain the same
    c. both n_2 and l retain the same, so $n_2 + 1 = l$ retains true.
2. v is a node with 1 child. Therefore, adding a node to v results in
    a. v now has 2 children ⇒ $n_2 = n_2 + 1$
    b. node w is a leaf node ⇒ $l = l + 1$
    c. $n_2 + 1 = l + 1 + 1 \Rightarrow n_2 = l + 1$ ⇒ both number of nodes with 2 children and numbers of leaves increment by 1, so it is offset and the statement retains true
3. v is a node with 2 children. By definition of a tree, w cannot be added to v so only the first 2 cases are viable cases.

Hence, by induction, the number of nodes with two children in a binary tree is exactly one less than the number of leaves.

Problem 3

Algorithm

- for each person n_i, create a graph G with 2 child nodes b_i and d_i such that there exists an edge from b_i and d_i
- let i represent an arbitrary person, and j another unique arbitrary person
- if d_i < b_j
    - add edge (d_i, b_j)
- if $d_i \geq b_j$
    - add edge (b_j, d_i)
    - add edge (b_i, d_j)
- if graph has a cycle

    - data is not internally consistent
- else
    - data is consistent

Proof

- BWOC assume that it is possible for both of the forms to coexist for a a person P_i
- if a person died before some other person was born but also partially overlapped in their lives, that would create a cycle between the two vertices, which is a logical contradiction—a person can't be dead and living unless they are schrodinger's cat
- hence, if there is a cycle in our graph, that means the internal dates are not consistent because at least one person will follow both forms from the question, which we've proven as a contradiction.

Problem 4

When BFS and DFS result in the same tree, that implies that the topology of G is that it itself is that tree, where the nodes of G have no other edges other than those that are represented in the trees found by the 2 search algorithms.

By way of contradiction, suppose G is not a tree

- that assumes that there exists at least one edge in G that does not exist in the tree found by BFS and DFS
- for BFS and DFS to have the same graph, we know that each arbitrary node must have an edge that connects to another node such that there is one level of distance between the two nodes
- for DFS to have such nodes that are represented in that manner, each node must have an edge that extends from a parent node to a child node that is one level's distance apart
- but since G is not a tree there exists an edge that connects a node to another node in the same level. This is a contradiction.

Therefore, G must be a tree.

Problem 5

Let T be the spanning tree rooted at the start vertex produced by the depth-first search of a connected, undirected graph, G. Argue why every edge of G, not in T, goes from a vertex in T to one of its ancestors, that is, it is a back edge

- By definition, a spanning tree of an undirected graph G is a tree that includes all the vertices of G such that each vertex connects from a vertex in G to its child exactly one level below, call it a tree edge.
- Since we know the graph is undirected, we know that the only type of edges we may have are tree edges, back edges, or level edges (edges that connect 2 vertices in the same level).
- Assuming the DFS was performed correctly, all the tree edges of G are in the set of tree edges of T and vice versa, by definition of a DFS spanning tree. Hence we know G must be left with either back edges or level edges
- We must prove there are no level edges in G. BWOC, assume that there are level edges in G
    - if there is a level edge that connects vertices v and w, there will exist a common ancestor between the two of them, call it vertex a
    - however, when DFS is performed, when vertex a is visited, DFS then visits v and then w in a manner where v is a parent vertex to w, which we said should have been a level edge above. This is a contradiction.
- Therefore, all level edges are in essence a tree edge because G is undirected.
- Hence, we are only left with back edges in G that aren't in T

Problem 6

The following two part question relates to Directed Graphs

1. **Can you design an algorithm that finds the longest path in a directed graph (DG)? (in a longest path you can use an edge at most once)? If yes, describe the algorithm and analyze its time complexity.**

Since the directed graph may have internal cycles, we must write an algorithm that traverses these cycles without running into a loop.

Algorithm

- let s be the root node of this graph
- let d[] be the distance from the s node to any node v
- initialize d[s] = 0, and all other node v d[v]=0
- function takes (node v, distance)
    - if v is visited
        - return
    - mark v as visited
    - if d[v] < distance
        - d[v] = distance
    - for each adjacent vertex of v, say w
        - recursively pass node w and its distance+=1

Proof

By way of contradiction, assume that there is a path that is longer than the one found from the algorithm

This would mean that the distance of the last child of v < max distance where v is the vertex with the longest path, which itself is a contradiction because distance is only incremented once a deeper node is found

Time Complexity

The algorithm has a time complexity of O(infinitely complex)

- This algorithm recursively iterates through each of the vertexes v and each of its adjacent vertexes w, which represent each of the vertexes in n and each of its edges
- henceforth, the time complexity would be O(n+m) where n is the number of nodes in the graph and m is the number of edges in the graph because each edge can only be traversed at most once.

- But since the graph gets more complex/connected with the addition of each node, the best case would be an infinitely long runtime since every edge and node would have to be explored even in the best case

2. **Can you design an algorithm that finds the longest path in a directed acyclic graph (DAG)? If yes, describe the algorithm and analyze its time complexity.**

Algorithm

- let G = DAG
- find topological ordering of the DAG
- let d[] equal to the longest path of each node
- initialize d[s] where s is the source node to 0, and all other nodes to a minimum value
- for each node v in G
    - for each adjacent edge from v to node w
        - if d[v] < d[w] + 1
            - d[v] = d[w] + 1
- return node with highest value d[v] in list

Proof

Since we have topologically ordered the DAG, we know each node will have another node one greater distance away, so by searching through the array we will find the node with the most amounts of nodes one greater distance away from each of its child nodes, hence, our algorithm finds the largest path.

Time Complexity

This algorithm has a time complexity of O(n+m).

- The topological sort has runtime O(n+m)
- searching through all nodes v in G requires O(n)
- search through all nodes w in node v requires O(m)

- Therefore we have $O(n+n+m+m) \Rightarrow O(n+m)$