CS 180 HW2

Hanna Co

Due: April 29, 2021

1. Say that we have a graph, G, with n nodes, that has two disconnected components, c_1 and c_2. Take some node in c_1. If each node has at least degree $\frac{n}{2}$ then this means each node is connected to $\frac{n}{2}$ nodes. It follows that there must be at least $\frac{n}{2}+1$ nodes in c_1. A similar argument follows for c_2. However, this would mean that we have $n+2$ nodes in G, which contradicts our statement that G has n nodes. We can induct on this example, to m disconnected components within G. A similar argument follows, where if each node has degree $\frac{n}{2}$, then there must be $\frac{n}{2}$ nodes in each component, contradicting our initial conditions. Thus, if G has n nodes with degrees greater or equal to $\frac{n}{2}$, then G must be connected.

2. The claim is false.

Proof (by induction):

We want to show that for any constant c, we can construct some graph G such that $\frac{diam(G)}{apd(G)} > c$.

Let's start with some graph G with n nodes, connected "linearly" to maximize diameter (such as in the given example). We have

$$diam(G) = n - 1$$

and

$$apd(G) = \frac{x}{\binom{n}{2}}$$

where x is the sum of distances between all pairs of nodes. Let's call this ratio $\frac{diam(G)}{apd(G)}$, c.

Now add one more node, and an edge between this new node and the 'end' node of G, and call this new graph G' with $k = n + 1$ nodes. Now,

$$diam(G') = k - 1 = n$$

and

$$apd(G') = \frac{x + \sum_{i=1}^{n} i}{\binom{n}{2} + n}$$

Observe that the diameter increases by 1 with each node added, while the average pairwise distance increases by < 1. Thus, we can see that the following is true:

$$c < \frac{n}{\frac{x+\sum_{i=1}^{n} i}{\binom{n}{2}+n}}$$

This is true for any n and k, thus showing that for any graph G with n nodes and a maximal $\frac{diam(G)}{apd(G)} = c$, we are able to construct graph G' with $n + 1$ nodes and a maximal $\frac{diam(G')}{apd(G')} = c'$, such that $c' > c$. Thus, we have shown that the claim $\frac{diam(G)}{apd(G)} \leq c$ is false.

3.

a.

Say we are given the root node w. We can call A$(\cdot)$ on w, and it returns u. We then call A$(\cdot)$ on u, and it returns v. The distance between u and v is the diameter.

Proof (by contradiction):

Say our algorithm returns the path from u to v, but there exists some other unique path from x to y such that $dist(u, v) < dist(x, y)$, and $u \neq x \neq y$.

Our algorithm first picks the node that is furthest away from the root, and we know this node to have the greatest depth. This would mean x and u must have the same depth: in other words, $dist(u, r) = dist(x, r)$.

We are calling A$(\cdot)$ again on x and then u, so we can think of this as rooting our tree at x and u, and finding the node with the greatest depth. If $dist(u, v) < dist(x, y)$, then this means the tree rooted at x is taller than the tree rooted at u. However, given that u and x are both leaf nodes are the same depth, we know that rooting a tree at either would produce trees of the same height. Thus, there cannot be some node y whose depth is greater than the depth of v, so it follows that $dist(u, v) < dist(x, y)$ must be false. Therefore, $dist(u, v)$ is the diameter of the graph, and our algorithm is correct.

b.

Breadth first search would be ideal, as it goes down the nodes level by level. We know the node on the lowest level is farthest from the root node, so breadth-first search essentially keeps track of distance for you, compared to depth-first search, which can get messy with recursion. Since we are using breadth-first search twice, the time complexity for our algorithm is $2 * (E + V)$, as proved in lecture, simplified to $O(E+V)$,where E is the number of edges and V is the number of vertices in the tree.

4.
Algorithm):
for every $P_i, i = 1, ..., n$

Create a node P_{ib} and P_{id}, representing the birth and death of each P_i. There is a directed edge going from P_{ib} to P_{id}, because P_{ib} occurred before P_{id}.

for every fact

if P_i died before P_j was born (no overlap)

Create edge (P_{id}, P_{jb})

else if the life spans of P_i and P_j overlap

Create edges (P_{ib}, P_{jd}) and (P_{jb}, P_{id})

date = Jan. 1 1821
assign date to all nodes with indegree 0 and push them onto L
date++
for all nodes in the graph

u = L.pop()
append u to the topological ordering
for all nodes v such that $(u, v) \in E$

indegree[v]- -
assign date to v
if indegree[v] == 0

L.push(v)

date++

If the result is that the graph that we have formed, G, is not a DAG, then we report an inconsistency. Otherwise, we can keep track of the earliest available time in the two hundred year window, and assign that to the node with indegree 0.

Proof of correctness:
We know there exists an inconsistency if event i is said to have occurred before event j, but event j is also said to have occurred before event i. In our graph, this means there would exist edges (i, j) and (j, i). This would indicate the presence of a cycle. However, we proved in lecture that a graph does not have a topological ordering if there is a cycle. Thus if we aren't able to find a topological ordering, then our graph is *not* a DAG, indicating an inconsistency. Our method for assigning dates is correct, since we know that if we have nodes u and v with edge (u, v), event u must have occurred before event v. Our system adheres to that, assigning dates to nodes that have edges with the currently processed node.

Proof of time complexity:
Our first two loops run in $O(n)$ and $O(m)$ time complexity respectively. Our graph has $2n$ nodes and a maximum of $n + 2m$ edges, so we have a time complexity of $O(3n + 3m)$, which we simplify to $O(n+m)$. Since the topological sort has the greatest time complexity, we use that, so our algorithm has a time complexity of $O(n+m)$.

5.
keep two counter variables, i and j
while ($i < m$ and $j < n$)
    if (S′[i] = S[j])
        $i++$
    $j++$
return ($i == m$)

Proof of correctness:
This algorithm iterates over the elements of S′ and compares them to elements in S. If it is a match, we move on to the next elements in both S′ and S. Otherwise, we only increment j. If $i == m$ is true, this means that we iterated over all elements of S′ and were able to match them all, in which case S′ is a subsequence of S. If $i \geq m$, then that would mean the loop terminated because we ran out of elements in S to compare, thus S′ is not a subsequence of S.

Proof of time complexity:
In the worst case, we have to examine every element in S, in which case the while loop runs n times. Thus our time complexity is $O(n)$.