CS 180 Problem Set 2

Max Smiley

Spring 2021

1.) We prove a stronger version of the prompt - that for a graph G with n nodes, if every node of G has degree of at least $\lceil \frac{n}{2} \rceil$, then G is connected. We prove this by induction on n, the number of nodes in G.

Base Case: $n = 2$
Let $G = (V, E)$, where $V = \{v, w\}$. Since each node must have at least one edge, $E = \{(v, w)\}$ and thus G is connected, as $\exists$ a path between v and w.

Inductive Step: Assume $P(n-1)$
Let $G = (V, E)$. We partition the set of nodes V into one node v, and the set of all other nodes besides v, called V'. The minimum number of edges each node will have is $\lceil \frac{n}{2} \rceil$. Even if every node in V' has an edge with v, the "outsider" node, each node in V' still must delegate at least $\lceil \frac{n-1}{2} \rceil$ edges amongst the nodes in V'. However, as V' has size $n-1$, the graph formed by V' must be connected by the induction hypothesis. Since v also must have some edges, it is forced to connect to the graph formed by V', and thus all of G is connected, as V' is connected, and there is a path from v to at least one node in V'.

The original prompt is simply a special case of this result, namely when n is even, and $\lceil \frac{n}{2} \rceil = \frac{n}{2}$, as $\frac{n}{2}$ is an integer.

2.) Consider a graph $G = (V, E)$ with V split into two sets X and Y, such that $X \oplus Y = V$. Let $X = \{x_1, \ldots x_n\}$ be the complete graph of n nodes, i.e. $e = (x_i, x_j) \in E \; \forall x_i, x_j \in X$. Now let $Y = \{y_1, \ldots y_m\}$, such that $e = \{y_i, y_j\} \in E$ if and only if $|i - j| = 1$.

Now consider what happens as we fix m and let n grow arbitrarily large. Since all nodes in the "complete graph" subset of nodes are connected, the maximum path, i.e. the diameter of the graph is determined by the nodes in the set Y. In particular, the diameter will be m, as it will take m steps to reach the leaf node in Y from any of the nodes in X.

In contrast, the average pairwise distance of G will approach 1 as $n \to \infty$. This is because the number of adjacent nodes will vary with respect to n, while the number of nodes order m away will not grow, as they vary with m, which is fixed. In other words, for this particular graph

$$\lim_{n \to \infty} \frac{diam(G)}{apd(G)} = m$$

which is unbounded by any constant, as we can choose m to be an arbitrarily large value.

3a.) I will propose an algorithm, and show its correctness. Given a graph $G = (V, E)$

Pick any node $u \in V$
Run $A(u)$, and store the returned node as v
Run $A(v)$. The number returned $d(v, A(v))$ will be the diameter of G

Indeed, this algorithm calls $A(\dots)$ a constant number of times. Now, I will show its correctness.

Lemma A: *$A(u)$ returns a leaf node*
Proof: Suppose $A(u)$ returns a node v that is not a leaf node. Then, there is another edge protruding from v leading to a new node w that is not a part of the path. But then, we could have added that to the path, and created a new path longer than the one from u to v, which is a contradiction since $A(u)$ returns the node with the greatest distance from v. Note that since G is a tree, there exists a unique path between two nodes, and so there is no shorter path between w and u than the one that passes through v.

Lemma B: *The diameter of a graph G is between two leaf nodes*
Proof: Suppose the diameter of G is between two nodes u and v, and by way of contradiction, at least one of these nodes is not a leaf node, assumed to be u without loss of generality. Again, this implies there is another edge protruding from u that leads to a node w that is not a part of the path. However, this means a larger path could have been constructed between the nodes w and v. Note that since G is a tree, there exists a unique path between two nodes, and so there is no shorter path between w and v than the one that passes through u.

Suppose, by way of contradiction, that the node v returned by $A(u)$ is *not* part of a maximal path. We will define a maximal path, i.e. a path whose length is the diameter of G to have start and end nodes a and b, which are leaf nodes by Lemma B. Similarly, $A(u) = v$ is a leaf node by Lemma A. By our assumption, $v \neq a, b$.

There are two cases to consider here. First, what happens if the path from u to v intersects the path from a to b at a common junction w. If we consider the path from a to w, the algorithm has a chance to go to v or b from there. Since it chooses

b, as the maximal path is ultimately (a, b), this implies that $d(w, b) \geq d(w, v)$. But then, as we call $A(u)$ and the algorithm hits w, it should ultimately choose to go down the b route, since $A(u)$ is supposed to return the node farthest from u, which is a contradiction.

The second thing to consider is if the path from (a, b) and (u, v) don't intersect, i.e. their paths share no common nodes, i.e. w is not reached, where w is a node in the path between (a, b). Since v is defined to be the node farthest from u, then $d(u, x)$ is maximized when $x = v$. It holds that

$$\Rightarrow d(u, v) \geq d(u, w) + max(d(w, a), d(w, b))$$

$$\Rightarrow d(u, v) - d(u, w) \geq max(d(w, a), d(w, b))$$

$$\Rightarrow d(v, w) \geq max(d(w, a), d(w, b))$$

But if this were true, then the path that starts at a, passes through w, and ends at v is at least as big as the maximal path. Thus, we have arrived at a contradiction in any case, so $A(u) = v$ must return a or b, i.e. a leaf node part of a maximal path. Since $A(v)$ returns the node farthest from v, then by definition, the length returned by $A(v)$ is the diameter of G.

3b.) We can implement $A(\dots)$ by using breadth first search traversal. Since BFS returns "layers" of nodes, where L_i represents the set of all nodes with distance i away from the source node, we can simply run BFS($\dots$) and return an arbitrary node in the final set, and the value of i. This is guaranteed to return a node with the greatest distance from the source node, and the number of steps away it is. As shown in class, this algorithm will run in $O(m + n)$, where m and n are the numbers of edges and nodes in the graph.

4.) I will propose an algorithm, analyze its time complexity, and show its correctness.

```
create a directed graph $G = (V, E)$
for each person $P_i$
    add $b_i$, $d_i$ to $V$
for each fact $F_i$
    if $P_i$ died before $P_j$ was born
        add $(d_i, b_j)$ to $E$
    if $P_i$ and $P_j$'s lives overlapped
        add $(b_i, d_j)$ to $E$
        add $(b_j, d_i)$ to $E$
for each person $P_i$
    if $b_i$ has degree 0
        remove $b_i$ from $V$
    if $d_i$ has degree 0
        remove $d_i$ from $V$
topologically sort $V$
if $\exists$ cycle
    return facts are internally inconsistent
else
    return topological ordering
```

This algorithm runs in $O(n+m)$, where $|V| = n$, and $|E| = m$. Initializing an empty digraph is an $O(1)$ operation. We then have three loops, two of which iterate over all P, which is an $O(n)$ operation, as each iteration simply adds or removes P_i to/from V which is $O(1)$, and the third loop iterates over all F, which is an $O(m)$ operation, as each iteration simply adds or removes edges to E. Finally, topological sort is an $O(n+m)$ algorithm as shown in class, so the entire algorithm is $O(n+m) + 2O(n) + O(m) = O(n+m)$.

If G has a cycle, then this implies some events A and B both happened before each other, which means the facts as presented are inconsistent. On the other hand, if there exists a topological ordering, then this can be used as the order of birth and death dates of the people.

5.) I will propose an algorithm, analyze its time complexity, and show its correctness.

```
initialize a pointer to the first element of $S'$
for each (in order) item in $S$
    if this item = what is pointed to in $S'$
        if pointer is not on the last element in $S'$
            advance pointer to next item in $S'$
        else
            return true
return false
```

This algorithm runs in $O(n+m)$, where $|S| = n$ and $|S'| = m$. This is because in its worst case, it iterates through all items in S, which has size n, and advances the pointer in S' past its end, which is m instructions, as S' has size m.

We want to determine if S' is a subsequence of S. In other words, elements of S' must appear in S, in order but not necessarily consecutive. The algorithm I gave works, because once it finds the first element of S' in S, it searches *past* that point in S. If it doesn't find all the elements in order, it returns false, but it returns true if it does eventually find all the elements in order down the list.