1 Reachable nodes (50 points)

Let $G$ be a directed graph represented by an adjacency list. Suppose each node $u$ has a weight $w(u)$, which might be different for each node. Give an algorithm that computes, for every node $u$, the maximum weight of any node that is reachable from $u$. So, for example, if $G$ is strongly connected then every node can reach every other node, so for every node the maximum reachable weight is the same (the largest weight of any node in the graph).

More formally, for each vertex $u$ let $R(u)$ denote the vertices reachable from $u$. Then when your algorithm is run on $G$, it should return an array of values where the value for node $u$ is $\max_{v \in R(u)} w(v)$.

Your algorithm should run in $O(m + n)$ time. Prove correctness and running time.

Solution: Our algorithm has 4 steps. First we will run a modified Kosaraju’s algorithm to find all the strongly connected components. Then we find the graph $\hat{G}$ of SCCs. Since $\hat{G}$ is a directed acyclic graph, we can do a topological sort. Finally, we use a dynamic programming to calculate the maximal reachable weight. The algorithm is as follows:

Do Kosaraju’s algorithm on $G$ and get SCCs $\{C_1, \ldots, C_t\}$
For $i = t$ to 1:
    For every vertex $u$ in $C_i$:
        $c_u \leftarrow i$  //mark $u$ as in $C_i$
    For every edge $(u, v)$ in $G$:
        Add an edge $(C_{c_u}, C_{c_v})$ to $\hat{G}$
Do topological sort on $\hat{G}$ and get $(C_{a_1}, \ldots, C_{a_t})$
    //There is no edge from $C_{a_i}$ to $C_{a_j}$ with $i > j$
For $i = t$ to 1:
    $m_{a_i} \leftarrow \max\{\max_{v \in C_{a_i}} w(v), \max_{j:(C_{a_i}, C_{a_j}) \in G} m_{a_j}\}$
For every vertex $u \in G$:
    Output $m_{c_u}$

We first prove the running time. The first step takes $O(n + m)$ time, which is proven in class. The second step takes $O(n + m)$ times because marking takes $O(n)$ time and adding edges takes $O(m)$ time. The third step also takes $O(n + m)$ time because the number of vertices in $\hat{G}$ is $t \leq n$, and the number of edges in $\hat{G}$ is $m$. (We allow multiedges in our construction of $\hat{G}$.) The fourth step still takes $O(n + m)$ time because every vertex in $G$ is visited only once, and every edge in
\( \hat{G} \) is also visited only once. The outputting step obviously takes \( O(n) \) time. Therefore the total running time is \( O(n + m) \).

For the correctness, in class we have proven that Kosaraju’s algorithm correctly gives us the SCCs, and the next step gives us the graph of SCCs, by definition. We have also proven in class that graph \( G \) is a DAG, so we can do a topological sort correctly.

Now we claim that, for all \( i \in [t] \) and all \( u \in C_{a_i} \), we have \( m_{c_i} = m_{a_i} = \max_{v \in R(u)} w(v) \), which immediately shows the algorithm outputs the right array.

The first equation is easy to prove, because \( c_u \) is set to be \( a_i \) if \( u \) is in \( C_{a_i} \). For the second equation, we will prove the claim by induction on \( i \).

The base case is \( i = t \). Since \( C_{a_t} \) does not have out edges (both in \( G \) and in \( \hat{G} \), because of the topological sort) and \( C_{a_t} \) is a SCC, we know that for every vertex \( u \in C_{a_t} \), \( R(u) = C_{a_t} \). We also know that \( \{j \mid (C_{a_t}, C_{a_j}) \in \hat{G} \} \) is an empty set. Thus \( m_{a_t} \) is set to be \( \max_{v \in C_{a_t}} w(v) = \max_{v \in R(u)} w(v) \).

Assume the claim is established for \( t, t - 1, \ldots, i + 1 \), consider the inductive step of \( i \). Because \( R(u) \) is the reachable set of \( u \), and \( u \) is in \( C_{a_i} \), which is a SCC, we know that

\[
R(u) = C_{a_i} \cup \bigcup_{p: (p, q) \in G, p \in C_{a_i}} R(q)
\]

because the reachable set is the SCC and the union of all the reachable set of the reachable nodes outside of this SCC. Thus

\[
\max_{v \in R(u)} w(v) = \max \left\{ \max_{v \in C_{a_i}} w(v), \max_{p: (p, q) \in G, p \in C_{a_i}} \max_{v \in R(q)} w(v) \right\}
\]

\[
= \max \left\{ \max_{v \in C_{a_i}} w(v), \max_{j: (C_{a_i}, C_{a_j}) \in \hat{G}} m_{a_j} \right\}
\]

because of the inductive hypothesis and there is no edge from \( C_{a_i} \) to \( C_{a_j} \) with \( i > j \).

Therefore, the algorithm successfully set \( m_{a_i} \) to be \( \max_{v \in R(u)} w(v) \) for all \( u \in C_{a_i} \), which finishes the proof.

2 Shortest Paths (50 points)

(a) (25 points) Let \( G = (V, E) \) be a directed graph with weights \( w : E \to \mathbb{R} \) with no negative-weight cycles. Let \( v \in V \), and let \( \ell \) denote the maximum, over all \( u \in V \), of the number of edges on a shortest path from \( v \) to \( u \) (where the shortest-path is defined with respect to the weights). Given \( G \), \( w \), and \( v \) (but not \( \ell \)), give an algorithm that computes shortest paths from \( v \) in \( O(m\ell) \) time. You may assume that \( m = \Omega(n) \). Prove correctness and running time.

(b) (25 points) We are given a directed graph \( G = (V, E) \) where every edge \( e \in E \) has a value \( r(e) \) with \( 0 \leq r(e) \leq 1 \). We think of each edge as a communication channel, and we interpret \( r(e) \) as the reliability of edge \( e \): it is the probability that a message sent across \( e \) reaches the other side. We assume that these probabilities are independent for each edge (whether \( e \) fails or not does not affect whether \( e' \) fails, for \( e' \neq e \)). Give an algorithm (as fast as you can) that takes as input the graph, the reliabilities, and two vertices \( u, v \in V \), and computes the most reliable path from \( u \) to \( v \). Prove correctness and running time.
Solution:

(a) We can use a slight modification of Bellman-Ford. In the original Bellman Ford algorithm, we filled in $OPT(u, k)$ for every $1 \leq k \leq n$. But since every shortest path has at most $\ell$ edges, we only really need to fill in $OPT(u, k)$ for every $1 \leq k \leq \ell$. This would be easy if we knew $\ell$ in advance, but since we don’t, we will need to check after every iteration whether $k > \ell$. We can do this by seeing whether, for all $u \in V$, $OPT(u, k - 1) = OPT(u, k)$. If they are equal for all $u$, then $k > \ell$ and we can stop iterating.

Here is the algorithm in pseudocode:

```plaintext
for k from 1 to n {
    for u in V {
        fill in OPT(u, k) as in Bellman-Ford
    }
    if OPT(u, k - 1) == OPT(u, k) for all u in V
        break
}
```

Note that the backtracking information can be added while filling $OPT(u, k)$, so that we can actually find the shortest paths.

If we fill in the chart for $1 \leq k \leq \ell$, we are guaranteed to find all the shortest paths, because every shortest path has at most $\ell$ edges, and we proved in class that $OPT(u, \ell)$ is the length of the shortest path to $u$ using at most $\ell$ edges. Thus, all that remains to be shown is that our algorithm finds $\ell$ correctly. (To be specific, the outer loop of the algorithm will run exactly $\ell + 1$ times.)

Suppose $k \leq \ell$. Then there exists $w \in V$ such that $OPT(w, k - 1) \neq OPT(w, k)$. To see why this is true, consider $u \in V$ where the shortest path from $v$ to $u$ has exactly $\ell$ edges, and suppose this shortest path is $v = w_0, w_1, \ldots, w_{\ell-1}, w_\ell = u$. Now consider $OPT(w_k, k)$. If $OPT(w_k, k) = OPT(w_k, k - 1)$, then there would exist some other path $v = w_0', w_1', \ldots, w_i' = w_k$ with $i < k$ which had the same length as $v = w_0, w_1, \ldots, w_k$. But then we could substitute this into our shortest path to $u$, giving us a new path $v = w_0', w_1', \ldots, w_i' = w_k, w_{k+1}, \ldots, w_\ell = u$ that used fewer than $\ell$ edges. But this contradicts our assumption that the shortest path from $v$ to $u$ uses exactly $\ell$ edges! Therefore, $OPT(w_k, k) \neq OPT(w_k, k - 1)$, and so our algorithm keeps going.

Now suppose $k > \ell$. Then $OPT(u, k) = OPT(u, k - 1)$ for all $u \in V$. To see why, consider what it would mean if $OPT(u, k) \neq OPT(u, k - 1)$ for some $u \in V$. It would mean that we could achieve a shorter path to $u$ using $k$ edges than we could using $k - 1$. But this contradicts our assumption that all shortest paths use at most $\ell$ edges! So we know that $OPT(u, k) = OPT(u, k - 1)$ for all $u \in V$.

Thus, the if statement becomes true for the first time when $k = \ell + 1$, at which point the break is triggered and the algorithm finishes. So the algorithm correctly discovers the value of $\ell$.

The runtime is clearly $O(m\ell)$, since the outer loop runs $\ell + 1$ times, the for loop takes $O(m)$ because every edge gets checked at most once (as discussed in class), and the if statement takes $O(n)$ time to compute.

(b) We will design a set of weights $w : E \rightarrow \mathbb{R}_{\geq 0}$ where $w(e) = -\log(r(e))$. We can then run Dijkstra’s algorithm on $G$ with weights $w$ to give us the most reliable path. The backtracking information can be added as in the Dijkstra’s algorithm so that we can find the actual path.
The running time is the same as Dijkstra’s algorithm, which can be $O(m + n \log n)$ using Fibonacci heaps.

Note that all weights are indeed nonnegative, so Dijkstra’s algorithm will return the shortest path $P$ under the new weights $w$. We just need to prove that this is the most reliable path. Note that the length of $P$ is

$$\sum_{e \in P} w(e) = \sum_{e \in P} \log \left( \frac{1}{r(e)} \right) = \log \left( \frac{1}{\prod_{e \in P} r(e)} \right) \quad (1)$$

Suppose that there is a more reliable path $P'$, so $\prod_{e \in P'} r(e) > \prod_{e \in P} r(e)$. Then the length of $P'$ is

$$\sum_{e \in P'} w(e) = \sum_{e \in P'} \log \left( \frac{1}{r(e)} \right) = \log \left( \frac{1}{\prod_{e \in P'} r(e)} \right) < \log \left( \frac{1}{\prod_{e \in P} r(e)} \right) = \sum_{e \in P} w(e),$$

where we used (1) and the monotonicity of the log function.

This is a contradiction, since by construction $P$ is the shortest path under weights $w$ so the length of $P'$ cannot be shorter. Hence $P$ is the most reliable path.

Note that it’s possible to solve this problem directly (without doing the log transformation) by designing a new version of Dijkstra’s that multiples the edge weights in instead of adding them. However, that solution requires a more in-depth proof of correctness, which we do not include here.