Name: Jungbin Choi
Due November 3, 2023

COMP 550-001
Homework 4 (12 points)

Notes. Collaboration is allowed, and you may use any resources you'd like. You may call any algorithm given in the notes without providing details. However, you should write and submit your solutions individually, and each submission will be graded individually. In your submission, you should list your collaborators (including course staff) and cite every resource you use. For each problem, I encourage you to attempt it for at least 30 minutes before consulting any people or resources.

Your submission should be a single PDF file. I recommend using Overleaf; a template is on Canvas. If your answers are not clear, you might not get full credit. You should describe your algorithms in pseudocode similar to the examples given in the lectures/notes. I recommend also including a brief description in full sentences.

Problem 1 (5 points). Let $G = (V, E)$ be a directed acyclic graph on n vertices and m edges, where each edge $e = (u, v)$ has a (possibly negative) length $\ell_e = \ell(u, v)$, and let s be a vertex. Our goal is to find the length of the longest path from s to v for every vertex v. You may assume that every vertex is reachable from s. Note that there is only one path from s to itself, and it has length 0.

a. (2p) Give an $O(m)$-time DP algorithm for this problem. In particular, you should define a subproblem, state the recurrence, and write the corresponding pseudocode. Your algorithm should return an array d such that $d[v]$ is the length of the longest path from s to v.

b. (2p) Prove that your algorithm is correct. In particular, you only need to explain why your recurrence is true.

c. (1p) Analyze the running time of your algorithm.

Solution.

a. **Subproblem:** $OPT[v]$ = The length of longest path from s to $v \in V$

Base Case: $OPT[s] = 0$ (the longest possible path from a vertex to itself has length zero)

Recursion: $OPT[v] = \max_{u:(u,v)}(OPT[u] + \text{len}(u, v))$ (The maximum possible value for every in-neighbor edge plus that in-neighbor's optimal value)

Pseudocode:

```
ALG(G,s):
   d = [-inf] * n
   d[s] = 0
   for v in V:
       for each u in (u, v):
```

```
            d[v] = max(d[v], d[u] + len(u, v))
    return d
```

b. The reason why the recursion works is that it's building off the fact that the in-neighbors of a vertex will already have their optimal value (the max length of path from s) calculated, so by adding the edge length to the neighbor's value, and then seeing which one is the largest, then the calculated value will be the biggest length possible for a path from s to the vertex. Another factor is that since the graph is known to be acyclic, there won't be a situation where there is a loop of only positive edge lengths and would cause the length to become infinity.

c. Initialization of $d[]$ is $O(n)$, and then the base case of $d[s] = 0$ only takes constant time. Then the outside for-loop will take $O(n)$ iterations because it'll go through each vertex once. Then, per each iteration, the inner for-loop will take $O(\#$ in-neighbors of $v)$ time. Everything inside the nested for-loops will take constant time. Therefore, the running time for the nested loop will be $O(m)$ because the sum of each vertices' number of in-neighbors represents the total number of edges. Lastly, the final running time will be $O(m+n)$, or because the starting vertex can reach every other vertex and thus requires more edges than the number of vertices, simplified down to $O(m)$.

Problem 2 (3 points). Let $G = (V, E)$ be a directed graph on n vertices and m edges. Our goal is to return a graph $G' = (V, E')$ that is the reverse of G. More specifically, G' has the same set of vertices as G, and each edge of G' is the reverse of an edge in G. So for any two vertices $u, v \in V$, we have $(u, v) \in E$ if and only if $(v, u) \in E'$.

a. (2p) Give an $O(m + n)$-time algorithm that, given G, returns G'. Your pseudocode should specifically work with G in the adjacency list format. (So, for example, it should not say something like "for each edge in G".)

b. (1p) Analyze the running time of your algorithm. You may assume that creating an array of n elements takes $O(n)$ time, accessing an element of an array given its index takes $O(1)$ time, and appending to a list takes $O(1)$ time.

Solution.

a.
```
ALG(G):
    result = [{}] * n   \\ Adj. List: n-size Array of empty Lists
    for v in G:   \\ Go through G's array of vertices
        if not v.empty():
            u = v.getNext()   \\ Get first item in list
            while u != None:
                result[u].add(v)   \\ At index u , add reverse edge to v
                u = v.getNext()
    return result
```

b. Creating the *result* adjacency list will take $O(n)$ time. The for-loop will also take $O(n)$ time to go through the input graph's vertices (represented as v). Within the for-loop, everything other than the while loop will take $O(1)$ time. The while-loop will take $O(\text{out-degree}(v)$ per the for-loop's iteration. Therefore, the entire nested loop will take $O(\text{out-degree}(v1) + \text{out-degree}(v2) + ...)$, or simplified to $O(m)$. Finally, the total running time for the algorithm will be $O(m + n)$ time.

Problem 3 (4 points). Let G be a directed graph on n vertices and m edges, where each edge $e = (u, v)$ has a length $\ell_e = \ell(u, v) \geq 0$, and let x be a vertex of G. An *x-walk* is a walk that passes through x. Give an $O(n^2)$-time algorithm that finds the length of the shortest x-walk between every pair of vertices.

a. (2p) Give an $O(n^2)$-time algorithm for this problem. Your algorithm should return a 2-dimensional array d such that $d[u][v]$ is the length of the shortest x walk from u to v. (If there is no x-walk from u to v, then $d[u][v]$ should be ∞.)

b. (1p) Prove that your algorithm is correct.

c. (1p) Analyze the running time of your algorithm.

Solution.

a.
```
ALG(G, X):
    dij = Dijkstra(G, X)   \\ Run Dijkstra's using X as S

    G_rev = Problem2(G)   \\ Run Q2's reverse alg
    dij_rev = Dijkstra(G_reverse, X) \\ Run Dijkstra's using G reversed

    d = [inf * n][inf * n]
    for u in V:
        for v in V:
            d[u][v] = dij_rev[u] + dij[v]

    return d
```

b. Dijkstra's will get the shortest path (i.e. the distance) from the starting vertex (in this case X) to every other vertex. If the algorithm is ran with the same starting vertex but with a reversed graph, the values found are the distances from every vertex to the starting vertex. Now, looking at the properties of a x-walk, we can divide the walk into two portions: $u \to X$ and $X \to v$. Because we found the distance from every vertex to X as well as every distance from X to every other vertex, we can add the corresponding path values (which we know to be the shortest) and get the shortest path $u \to v$ that contains X.

c. The 1st Dijkstra's will take $O(n^2)$ time, the reversing algorithm will take $O(m+n)$ time, and the 2nd Dijkstra's value will also take $O(n^2)$ time. The 2D array initialization will

take $O(n^2)$ time, and the nested loops will take a total of $O(n^2)$ time (as the expression within the loop is constant time). Thus, adding it all up, the total running time is $O(4n^2 + m + n)$, which simplifies down to $O(n^2)$.