I would like to follow the below syntax for few questions of this assignment.

Syntax: Please consider this CLRS Book terminology for the solutions to the questions of this assignment.

Graph (V, E) with n vertices and m edges OR n= |V|, m=|E| with weights of edges (u,v) represented as w (u,v).

For each vertex v ∈ V, we maintain an attribute d[v], which is an upper bound on the weight of a shortest path from source s to v. We call d[v] a shortest-path estimate.

Predecessor/parent of v is represented as π[v]

δ(u, v) is the shortest-path weight from u to v as defined in CLRS

Q1: Let G = (V, E) be a weighted, directed graph with weight function. Give an O(nm)-time algorithm that finds for every vertex v, the value $\delta^*(v) = \min_{u \in V} \{\delta(u, v)\}$.

Solution:

The Algorithm for this problem is identical to that of Bellman ford, except instead of initializing the values to be infinity, source to zero, we initialize every d value to be zero. We can even recover the path of minimum length for each 2 vertex by looking at their π values.

We initialize d[v] = 0 for each v ∈ V and then run the Bellman-Ford algorithm on this network, i.e., the relaxations are organized into n − 1 rounds and each edge is relaxed once in every round. As with the Single-Source shortest paths problem, if there exists a further relaxable edge at the end of (n−1) rounds, we declare that G has a negative cost cycle and our algorithm returns FALSE. Note that in the presence of negative cost cycles, the shortest path problem is undefined.

Thus, at the end of the (n − 1)th round of the for loop, we have $d[v] = \delta^*(v)$ for all vertices v ∈ V .

where d[v] represents the actual cost of a path from some vertex t ∈ V to v and hence it follows that

$d[v] = \delta * (v)$.

where d[v] represents the length of a path from some vertex u ∈ V to v. If G does not contain a negative cost cycle, then at the end $d[v] = \delta^*(v)$, for all v ∈ V.

Please see the Pseudocode in the next page.

Pseudo Code:

Modified-BF(G,w,s) //s is the source vertex

```
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to |V[G]| − 1
3       do for each edge (u, v) ∈ E[G]
4            do RELAX(u, v, w)
5  for each edge (u, v) ∈ E[G]
6       do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

INITIALIZE-SINGLE-SOURCE (G, s)

for each vertex v∈V **do**
    d[v]= 0

RELAX (u,v,w)

If d[v] > d[u]+w(u , v) then
    d[v]=d[u]+w(u , v)

Time Complexity:

This algorithm is a simple modification to Bellman-Ford so it will have the same time complexity of O(mn) or O(|V||E|)

I also would like to propose a second solution for this question:

2nd Approach:

Let G= (V , E).

Construct new graph G' by introducing a new vertex s and new edges of weight 0 from s to all vertices in V. Then, for all u∈V, shortest path weight from s to u in G' is the same as δ*(u) in G. So, simply run Bellman-Ford on G' with s as the source vertex; its output is indeed the desired output.

Pseudo Code:

Shortest-Paths-Ending-At-Each-Node(V , E , w)

V'= V ∪ {s}
E'= E ∪ {(s , u)|u∈V}
w'(s , u) = 0 for all u ∈ V, and w'(u , v) =w(u , v) for all u , v ∈ V
run Bellman-Ford(V', E ' , w ' , s) to compute δG'(s , u) for all u∈V
output δG'(s , u) as δ*G(u) for all u∈V

Time Complexity:

Since|V'|=|V|+ 1 and|E'|=|E|+|V|, the time complexity of running Bellman-Ford on G' is O((|V|+ 1)(|E|+|V|)) =O(|V| |E|) (assuming E= Ω(V)). Hence, the time complexity of the above algorithm is O(|V| |E|) or O(mn)

Q2. Suppose that we have a sequence of MakeSet-Find-Union operations in which no Find appears before any Union. What is the computational time for this sequence?

Solution:

Let us assume that we have **m** sequences of MakeSet-Find-Union operations and **n** number of edges.

Union by rank and path compression are used

Time Complexity: Each of these operations take the following time

MakeSet - O(1)

Union - O(1)

Find - O(logn)

We can have operations involving Makeset(denoted by M), Union(denoted by U) in a sequence like UM UM or MUMU or MMUU etc..Without considering **Find** the Computational time of the sequence of Makeset and Union would be O(m)

It is given that no Find appears before Union.

So the sequence may look like UUUUMMMMFFFF or MUMUMUMUMMFFFF etc..

As we are following path compression approach and all Find operations appear after Union, one Find operation compresses each edge along the path to the root, so that later Find operations on nodes on this path becomes direct children to the root and hence computational time is reduced to O(1).

So, Time complexity of m Find operations would be O(m+n). **Hence total Computational time for the given sequence is O(m+n), which is a linear-time.**

Q3: Design a linear-time algorithm for the following problem: given a directed acyclic graph G, and three vertices s, w, and t, construct a simple path (i.e., it does not repeat vertices) in G that starts from s, ends at t, contains the vertex w, and is the longest over all s-t paths in G that contain w.

If there is a path from vertex u to vertex v, then u precedes v in the topological sort. We make just one pass over the vertices in the topologically sorted order. As we process each vertex, we relax each edge that leaves the vertex.

Pseudo code: G(s,t) // n vertices and m edges //s is the source // t is the destination

1. **For** each vertex v **do** d[v] = Negative Infinity and π[v]=0 **//O(n)**
2. Topologically sort the graph G <-- TP[1….n] **//O(m+n)**
3. d[w]=0 **//O(1)**
4. For i=1 to n-1 do **//O(m+n)**
   If TP[i] is vertex 't' then break
   For each edge [TP[i],TP[j]] do
   If d[TP[j]]<d[TP[i]]+1 then
   d[TP[j]]=d[TP[i]]+1 and π[TP[j]]=TP[i]
5. d[s]=0 **//O(1)**
6. For i=1 to n-1 do **//O(m+n)**
   If TP[i] is vertex 'w' then break
   For each edge [TP[i],TP[j]] do
   If d[TP[j]]<d[TP[i]]+1 then
   d[TP[j]]=d[TP[i]]+1 and π[TP[j]]=TP[i]
7. If step 4 didn't find 't' (d[t]=Negative Infinity) or step 6 didn't find 'w'(d[w]=0) then REPORT Failure

A simple path is one with no repeated vertices. As we are considering directed acyclic graph, there should not be any cycles in the graph. A topological sort of a DAG is a linear ordering of all its vertices such that if G contains an edge (u,v), then u appears before v in the ordering. So If the graph is not acyclic, then this ordering is not possible. Hence paths in a directed acyclic graph are necessarily simple as they will not contain repeated vertices. Accordingly, In the topological sorting s will occur before t and w should occur in between s, t from the above definitions. Hence I have set d[w]=0 to find longest path from w to t then find longest path from s to w. If there exists a longest path from s to t and passing through w then path from w to v in that longest path of s-t should also be the longest.

Time Complexity:

Step 1 takes O(n)

Steps 2, 4, 6 are taking O(m+n)

Steps 3, 5 are taking O(1)

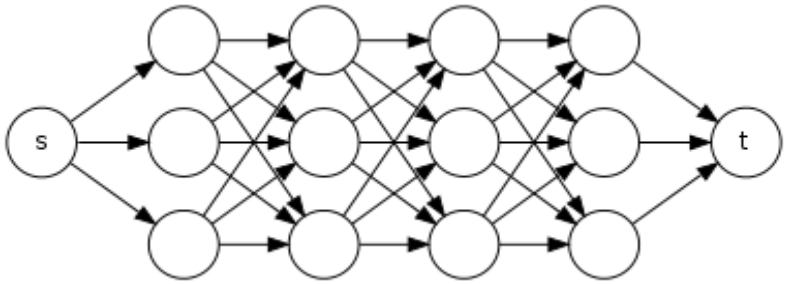
So Time complexity of this algorithm is O(m+n)

Q-4. Give a linear-time algorithm that takes as input a directed acyclic graph G and two vertices s and t, and returns the number of simple paths from s to t in G. Your algorithm needs only to count the simple paths, not list them. Note that different paths from s to t may share common vertices.

A simple path is one with no repeated vertices. As we are considering directed acyclic graph, there should not be any cycles in the graph. A topological sort of a DAG is a linear ordering of all its vertices such that if G contains an edge (u,v), then u appears before v in the ordering. So If the graph is not acyclic, then this ordering is not possible. Hence paths in a directed acyclic graph are necessarily simple. Accordingly, In the topological sorting s will occur before t from the above definitions. I would like to propose an algorithm for this problem using topological sorting and dynamic programming.

Approach:

Do a topological sort of the DAG, then scan the vertices from the target backwards to the source. For each vertex u, keep a count of the number of paths from u to the target. When you get to the source, the value of that count is the answer. This can be done in time no more than O(m+n).



The first time the vertex u is encountered, it computes the number of paths it has to the vertex t. This is stored in path_count[u]. As different paths from s to t may share common vertices, each subsequent call to u will not recompute its number of paths, but simply return path_count[u]. Pseudo code is as below:

SIMPLE-PATHS G(s,t) : //u is any vertex in the graph //G has m edges, n vertices
//s is the source and t is the destination.
//path_count[u] stores the number of simple paths from u to t.
//output: returns the number of simple paths from s to t in G

1. For each vertex v, Initialize path_count[u]=0 **//O(1)**
2. Set path_count[t]=1 **//O(1)**
3. Perform topological sorting on G **//O(m+n)**
4. **For** each vertex u in reverse topological order **do** **//O(m+n)**

```
    If path_count[u] != 0 then return path_count[u]
    Else
        For each successor w of u do
            path_count[u] = path_count[u] + path_count[w]
    If u=s then return path_count[s]
```

path_cost[u] tells the number of simple paths from u to t, where we assume that t is fixed throughout the entire process. To count the number of paths, we can sum the number of paths which leave from each of the neighbors of s. Since we have no cycles, we will never risk adding a partially completed number of paths.

Time complexity:

In our algorithm, if we start in reverse Topological order also, vertices counts doesn't change till we reach the vertex t because path_count[t]=1 and all other vertices has path_count=0 initially.

In this problem as we have assumed graph is unweighted DAG. Initially, set vertex t count (I mean path_count[u]) to 1 and other vertices count to 0. Each time Algorithm finishes a vertex v, set v's count to the sum of the counts of all vertices adjacent to v. When Algorithm reaches vertex s, stop and return the count computed for s.

It is not difficult to see that each edge is looked only once, hence algorithm has **runtime of O(|V|+|E|) or O(m+n).**

Thank you so much, Qin Huang