- Please note that one of the main goals of this course is to design efficient algorithms. So, there are points for efficiency even though we may not explicitly state this in the question.
- Unless otherwise mentioned, assume that graphs are given in adjacency list representation.
- In the lectures, we have discussed an $O(V+E)$ algorithm for finding all the SCCs of a directed graph. We can extend this algorithm to design an algorithm `CreateMetaGraph(G)` that outputs the meta-graph of a given directed graph in $O(|V| + |E|)$ time. For this homework, you may use `CreateMetaGraph(G)` as a sub-routine.
- The other instructions are the same as in Homework-0.

There are 6 questions for a total of 82 points.

---

1. An undirected graph is said to be connected iff for every pair of vertices, there is a path between them. For this question, you have to show the following statement:

   Any connected undirected graph with n nodes has at least $(n-1)$ *edges.*

   We will prove the statement using Mathematical Induction. First, we define the propositional function.

   $P(n)$: *Any connected undirected graph with n nodes has at least* $(n-1)$ *edges.*

   The base case is simple. $P(1)$ holds since any connected graph with 1 node having at least 0 edges is indeed true. For the inductive step, we assume that $P(1), P(2), ..., P(k)$ holds for an arbitrary $k \geq 1$, and then we will show that $P(k+1)$ holds. Consider any connected graph G with $(k+1)$ nodes and k edges. You are asked to complete the argument by doing the following case analysis:

   (a) (3 points) Show that if the degrees of all nodes in G is at least 2, then G has at least k edges.

   (b) (2 points) Consider the case where there exists a node v with degree 1 in G. In this case, consider the graph G' obtained from G by removing vertex v and its edge. Now use the induction assumption on G' to conclude that G has at least k edges.

Solution: SOLUTION:

We will be using the handshaking theorem for proving it: Handshaking theorem states that$----->$ For any undirected graph $G(V, E)$, sum of the degree of all the vertices of the graph is $2 * |E|$. We first proof this theorem.

We will use proof by induction. The proposition for our inductive proof is:

P(n): For any graph with n edges, we have:

$$\Sigma_{i=1}^{v} deg(v_i) = 2 * n$$

where v is the number of vertices in the graph. and $deg(v_i)$ represents the degree of i^{th} vertex For $n = 0$:

In a graph with no edges indegree of all vertices is 0 , therefore:

$$\Sigma_{i=1}^{v} deg(v_i) = 0 = 2 * 0$$

So the statement is true for $n = 0$

Let the statement be true till $n = k$, i.e.,:

$$\Sigma_{i=1}^{v} deg(v_i) = 2 * k$$

Now lets see it for $n = k + 1$:

Lets remove one edge from the graph (note that no vertex is being removed). Then we end up with a graph with k edges. For that case, since $P(k)$ holds, we have:

$$\Sigma_{i=1}^{v} deg(v_i) = 2 * k$$

Now if we add and an edge , it will just increase the degree of 2 vertices by 1, therefore:

$$\Sigma_{i=1}^{v} deg(v_i) = 2 * k + 1 + 1$$

$$\Sigma_{i=1}^{v} deg(v_i) = 2 * (k + 1)$$

therefore, the statement is true in general.

Now coming back to the question, we need to proof that:

$P(n)$: *Any connected undirected graph with n nodes has at least* $(n - 1)$ *edges.*

$P(1)$ holds since any connected graph with 1 node have at least 0 edges is indeed true. For the inductive step, we assume that $P(1), P(2), ..., P(k)$ holds for an arbitrary $k \geq 1$.

Now lets analyse for $n = k + 1$:

Here we end up with two cases:

Case1: The degrees of all nodes in G is at least 2.

$$\Sigma_{i=1}^{k+1} deg(v_i) \geq 2 * (k + 1)$$

Now using the hand shaking theorem,

$$2 * |E_{k+1}| \geq 2 * (k + 1)$$

where $|E_{k+1}|$ is number of edges in any arbitrary connected graph with $k + 1$ nodes.

Hence,

$$|E_{k+1}| \geq (k + 1)$$

$$|E_{k+1}| \geq k$$

Case 2a: There exists a node v with degree 1 in G. In this case, lets consider the graph G' obtained from G by removing vertex v and its edge. After that, we will still endup with a connected graph with k vertices (After removing the pendant vertex from a connected undirected graph, we still get a connected graph with 1 less vertex). Since we have assumed that $P(k)$ holds, number of edges in the connected graph obtained after removal have to be atleast $k - 1$. :

$$|E_k| \geq (k - 1)$$

where E_k is the number of edges in undirected connected graph with k vertices, obtained after removing a pendant vertex. Now, by handshaking theorem:

$$\Sigma_{i=1}^{k} deg(v_i) = 2 * |E_k| \geq 2 * (k-1)$$

Therefore,:

$$\Sigma_{i=1}^{k} deg(v_i) \geq 2 * (k-1)$$

Adding 2 both sides:

$$\Sigma_{i=1}^{k} deg(v_i) + 2 \geq 2 * (k-1) + 2$$

$$\Sigma_{i=1}^{k} deg(v_i) + 2 \geq 2 * k$$

Now the quantity $\Sigma_{i=1}^{k} deg(v_i) + 2$ is as good as sum of the degree of the vertices in graph G (from which we obtained G' by removing the pendant vertex, that is vertex with degree 1). This is because on removal of edge , the total degree was reduced by 2. Hence :

$$\Sigma_{i=1}^{k} deg(v_i) + 2 = \Sigma_{i=1}^{k+1} deg(v_i) \geq 2 * k$$

(Note the G have $k+1$ vertices!!) Now by handshaking theorem $\Sigma_{i=1}^{k+1} deg(v_i)$ is $2 * |E_{k+1}|$ ($|E_{k+1}|$ is number of edges in any arbitrary connected graph with $k+1$ vertices). Therefore:

$$2 * |E_{k+1}| \geq 2 * k$$

$$|E_{k+1}| \geq k$$

Therefore its true!!

Case 2b: This case is just generalization of the case 2a. There exists x vertices ($x < k$) with degree 1 in G. In this case, lets consider the graph G' obtained from G by removing x vertices and its edge. After that, we will still endup with a connected graph with $k-x$ vertices (After removing the pendant vertices from connected undirected graph, we still get a connected graph with less vertices). Since we have assumed that $P(1), P(2), P(3)...P(k)$ holds, therefore $P(k-x)$ holds. The number of edges in the connected graph obtained after removal have to be atleast $k-x-1$. :

$$|E_{k-x}| \geq (k-x-1)$$

where E_{k-x} is the number of edges in undirected connected graph with $k-x$ vertices. Now, by handshaking theorem:

$$\Sigma_{i=1}^{k-x} deg(v_i) = 2 * |E_{k-x}| \geq 2 * (k-x-1)$$

Therefore,:

$$\Sigma_{i=1}^{k-x} deg(v_i) \geq 2 * (k-x-1)$$

Adding $2 * (k-x)$ both sides:

$$\Sigma_{i=1}^{k-x} deg(v_i) + 2 * (k-x) \geq 2 * (k-x-1) + 2 * (k-x)$$

$$\Sigma_{i=1}^{k-x} deg(v_i) + 2 * (k - x) \geq 2 * k$$

Now the quantity $\Sigma_{i=1}^{k-x} deg(v_i) + 2 * (k - x)$ is as good as sum of the degree of the vertices in graph G (from which we obtained G' by removing the pendant vertices, that is vertices with degree 1). This is because on removal of edge , the total degree was reduced by $2 * (k - x)$. Hence :

$$\Sigma_{i=1}^{k-x} deg(v_i) + 2 * (k - x) = \Sigma_{i=1}^{k+1} deg(v_i) \geq 2 * k$$

(Note the G have $k + 1$ vertices!!) Now by handshaking theorem $\Sigma_{i=1}^{k+1} deg(v_i)$ is $2 * |E_{k+1}|$ ($|E_{k+1}|$ is number of edges in any arbitrary connected graph with $k + 1$ vertices). Therefore:

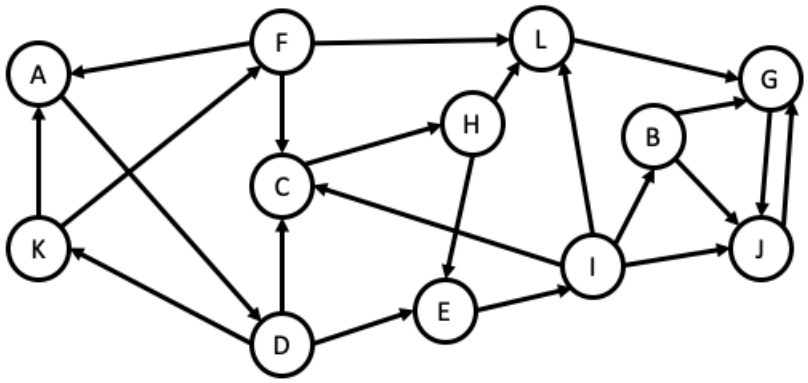
$$2 * |E_{k+1}| \geq 2 * k$$

$$|E_{k+1}| \geq k$$

Therefore its true!!

Hence proved by induction.

2. Consider the following directed graph and answer the questions that follow:



(a) (½ point) Is the graph a DAG?

Solution: Solution: The graph is NOT Directed acyclic graph as their exist a cycle. Example: $A- > D- > K- > A$.

(b) (1 point) How many SCCs does this graph have?

Solution:
Solution: The graph have 5 strongly connected component, namely:

$$\{G, J\}, \{B\}, \{L\}, \{I, C, H, E\} and \{A, D, K, F\}$$

(c) (½ point) How many source SCCs does this graph have?

Solution: Solution: We have only one source SCC, i.e.,

$$\{A, D, K, F\}$$

(d) (1 point) Suppose we run the DFS algorithm on the graph exploring nodes in alphabetical order. Given this, what is the pre-number of vertex F?

Solution: Solution: Prenumber of F will be 20 (assuming that clock starts with 1)

(e) (1 point) Suppose we run the DFS algorithm on the graph exploring nodes in alphabetical order. Given this, what is the post-number of vertex J?

Solution: Solution: Post number of J will be 10 (assuming that clock starts with 1)

(f) (1 point) Is it possible to add a single edge to this graph so that the graph becomes a strongly connected graph? If so, which edge would you add?

Solution: Solution: Yes, it is possible !! We have to add an edge from SCC $\{G, J\}$ to SCC $\{A, D, K, F\}$. Say an edge from G to A, i.e., (G, A).

3. (18 points) Suppose a degree program consists of n mandatory courses. The *prerequisite graph* G has a node for each course, and an edge from course u to course v if and only if u is a prerequisite for v. Design an algorithm that takes as input the adjacency list of the prerequisite graph G and outputs the minimum number of quarters necessary to complete the program. You may assume that there is no limit on the number of courses a student can take in one quarter. Analyse running time and give proof of correctness of your algorithm.

Solution: Solution:

Let the graph be represented by $G(V, E)$, where V is set of all vertices and E is the set of edges (which is represented in the form of adjacency list datastructure).

Few definitions which we are going to use:

1. Path: Sequence of vertices $(v_1, v_2...)$ such that consecutive vertices v_i and v_{i+1} are connected by edges (directed from v_i to v_{i+1}). Can be also represented as $(v_1 \rightarrow v_2 \rightarrow v_3.....)$
2. Length of a path: Number of edges in the path, or *number of vertices in the path* -1
3. In-degree of a vertex: Number of incoming edges to the vertex.
4. Null graph: Graph with no vertices.
5. Void graph: Graph with no edges.
6. If we have an edge from u to v then this directed edge can be represented as tuple (u, v)

Note that these definitions are in the context of directed graphs.

Note that these definitions are not universal, and vary from author to author.

Please keep all these definitions in mind while reading

In this probelem, number of quaters required to complete the degree program is given by= length of the longest path in DAG + 1

So if we are able to find the length of the longest path in DAG, we can easily find the number of quaters required to complete the course.

The algorithms goes like:

1. Initialize the in-degree array (an array which tells the in-degree of all the vertices) using the adjacency list. Also initialize "path" variable with 0 (which tells the length of the longest path in the DAG)
2. If the no nodes with zero indegree does not exist in the graph, or if we have a null graph, we return 0 (as either no quarters is required to complete the programme or it is not a valid DAG, as DAG have atleast one node with 0 degree)
3. Remove all the nodes with zero in-degree from the graph (as they can be completed in 1 quarter) and update the in-degree all vertices adjacent to nodes with 0 in-degree.
4. Increment the path by 1
5. If we are left with Null graph, return $path + 1$, else go to step 3 again.

(Proof of correctness is given after psuedo code)

Psuedo code for the algorithm is:

Note- We have used 1 based indexing

Note- By updating the indegree of adjacent vertices, it implicitly means we are removing edges from the graph, without actually modifying the adjacency list. And by poping a node from stacks, it implicitly means that node is being removed from the graph without modifying the set V

Note- In $G(V, E)$, E represent the adjaceny list and V represent the set of edges

Note- We assume that reader know standard data structures and programming constructs

```
return_quarters(G(V,E)):
    n=|V|
    ARRAY indegree[n]
    for each vertex v:
        indegree[v]=0

    for all (u,v) in E:
        indegree[v]=indegree[v]+1

    STACK st1
    STACK st2


    for each vertex v:
        if indegree[v]==0:
            st1.push(v)
    //pushing all the nodes with 0 indegree on st1
```

```
if st1.empty()
    return 0

/*if st1 is empty, it implies that either we have a null graph
or we don't have a valid DAG, so we return 0 */


path=0
//will tell the length of the longest path in DAG

stacknumber=1

/*The variable stacknumber tells which stack of the  stores the nodes with 0 indegree
which are being currently removed
and which stack stores the next layer of indegree 0 nodes.
If stacknumber is 1, it means we have 0 indegree nodes which are
currently being removed are in st1 and next layer of indegree 0
nodes are being accumulated in st2
else if stacknumber is 2, it means we have 0 indegree nodes which are
currently being removed are in st2 and next layer of indegree 0
nodes are being accumulated in st1*/



while not st1.empty() and not st2.empty():

    /*If the stack storing indegree zero nodes
    which were being removed
    becomes empty
    we update the length of the path.
    and change the stacknumber to the stack
    which store next level of indegree 0 nodes */

    if st1.empty() and stacknumber == 1 :
        path=path+1
        stacknumber=2

    if st2.empty() and stacknumber == 2 :
        path=path+1
        stacknumber=1


    /*Depending on which of the two stack stores the nodes
    with 0 indegree, which are to be removed,
    we remove the nodes and
    update the indegree of the neighbouring nodes
    and if after updating the indegree of the neighbouring nodes
    it turns out that their indegree becomes zero, we store
    those nodes in the opposite stack, which will store
    the next layer of indegree 0 nodes*/
```

```
        if stacknumber==1 :
            u=st1.pop()
            for all (u,v) in E:
                indegree[v]=indegree[v]-1
                if indegree[v]==0:
                    st2.push(v)

        if stacknumber==2 :
            u=st2.pop()
            for all (u,v) in E:
                indegree[v]=indegree[v]-1
                if indegree[v]==0:
                    st1.push(v)




    return path+1
```

The algorithm assumes that the length of the longest path in a void graph is 0 **(Doesn't matter the graph is null or non null)**

Note that null graph is also assumed to be a DAG. Also, since a null graph will also won't have any edges, it is also a void graph. So the length of the longest path in a null graph is 0

To prove the correctness of the algorithm we have to prove that:

1. A non-null DAG (a DAG with atleast 1 vertex) will always have a vertex with 0 in degree.
2. If we remove the vertex with 0 indegree from a non null DAG, we are still left with DAG.
3. Longest path in DAG will always start with a node with 0 indegree.
4. If $(v_1, v_2, v_3...., v_i)$ is the longest path(of length $i-1$) for a DAG, then their can't be a path longer than $(v_2, v_3...., v_i)$ (of length $i-2$) in a DAG obtained after removing all the nodes with 0 indegree.
5. For a non-void DAG, length of the longest path is : 1 + *length of the longest path in the DAG obtained after removing all nodes with* 0 *indegree.*

Note that we have assumed that we have a "finite" DAG

Lets prove them one by one:

Argument 1: A non-null DAG (a DAG with atleast 1 vertex) will always have a vertex with 0 in degree.

We will prove it via contradicton:

Lets say we have a non-null DAG in which all the vertices have a non-zero in-degree.

Let us pick up a vertex say, v_i. Now since v_i have a non-zero in-degree, we will have a vertex v_{i-1} such that we have a directed edge from v_{i-1} to v_i. Now since v_{i-1} also have a non-zero in-degree, so we have another vertex v_{i-2} such that we have a directed edge from v_{i-2} to v_{i-1}. It will keep on repeating.

Now since the graph is finite, some vertex will repeat in the sequence of path ($... \rightarrow a \rightarrow b.... \rightarrow v_{i-2} \rightarrow v_{i-1} \rightarrow v_i$). This implies existence of a cycle. But our graph is acyclic, which is a contradiction!!

Hence it cannot be the case that we have a non-null DAG in which all the vertices have a non-zero in-degree. Their have to be some vertex in a non-null DAG with 0 in-degree

Argument 2: If we remove the vertex with 0 in-degree from a non-null DAG than we are still left with a DAG.

Lets prove it via contradiction:

After removing a vertex with in-degree 0 from a non-null graph, we could have two cases:

case 1: We are left with a Null graph. Now a null graph is assumed to be a DAG for our algorithm

case 2: We are left with a non null graph: Lets say that we are left with a cycle in the graph after removing the vertex with in-degree 0 from DAG. In that case, even if we re-add the removed vertex with in-degree 0 and its corresponding edge, we still end up with graph with a cycle. So it was not a DAG to begin with!! Which is a contradiction!!

So if we remove the vertex with 0 in-degree from a non-null DAG than we are still left with a DAG.

Argument 3: Longest path in DAG will always start with a node with 0 indegree.

We can prove it via contradiction:

Lets say, we have a longest path $(v_1, v_2, v_3...., v_k)$ in a DAG (which have length $k-1$)and v_1 does not have a 0 indegree. Now since it is a path, every of the consecutive pair v_i and v_{i+1} will have a directed edge between them. So non of the $\{v_2, v_3....v_k\}$ can have a 0 indegree. Now since v_1 does not have a zero indegree, their exist a vertex say x such that we have a directed edge from x to v_1. So we could have another sequence of vertices $(x, v_1, v_2, v_3...., v_k)$ forming a path which is greater(length k) than assumed longest path(length $k-1$). Which is a contradiction!! Therefore, longest path in DAG will always start with a node with 0 indegree.

Argument 4: If $(v_1, v_2, v_3...., v_i)$ is the longest path(of length $i-1$) for a DAG, then their can't be a path longer than $(v_2, v_3...., v_i)$ (of length $i-2$) in a DAG obtained after removing all the nodes with 0 indegree.

Lets say that longest path in non-null DAG is $(v_1, v_2, v_3...., v_i)$. Since it is a path, every of the consecutive pair v_i and v_{i+1} will have a directed edge between them. So non of the $\{v_2, v_3....v_k\}$ can have a 0 indegree. Now if we remove all the vertices with 0 indegree, we will endup with a DAG with longest path $(v_2, v_3...., v_i)$

We can prove it via contradiction:

Lets say, after removing all the nodes with indegree 0 from the DAG, we end up with a DAG with a path longer than $(v_2, v_3...., v_i)$, say $(u_1, u_2, u_3, ..., u_j)$, i.e., $i-2 < j-1$ (u_1 will have indegree 0 now, via argument 3). Now u_1 cannot have indegree 0 in the original DAG (Otherwise it would have been removed in the first place). Therefore, u_1 must have atleast an incoming edge from some other nodes in the original DAG. Also all of the incoming edges to u_1 have to be from indegree 0 nodes because only then indegree of $u1$ can be 0 after removal of indegree 0 nodes from the original DAG.

Hence their exist a path $(x, u_1, u_2, u_3, ..., u_j)$ which is of length j in original DAG which is longer than the path $(v_1, v_2, v_3...., v_i)$ which is of length $i-1$ (since $i-2 < j-1$ so, $i-1 < j$), which is a contradiction!!

So it can't be the case that after removing all the nodes with indegree 0 from the DAG, we end up with a DAG with a path longer than $(v_2, v_3...., v_i)$.

Argument 5: For a non-void DAG, length of the longest path is : 1 + *length of the longest path in the DAG obtained after removing all nodes with* 0 *indegree.*

By argument 4, if $(v_1, v_2, v_3...., v_i)$ is the longest path(of length $i-1$) for a non null DAG, then their can't be a path longer than $(v_2, v_3...., v_i)$ (of length $i-2$) in a DAG obtained after removing all the nodes with 0 indegree.

Therefore, for a non-null DAG, length of the longest path is : 1 + length of the longest path in the DAG obtained after removing all nodes with 0 indegree.

(You can see that this argument is valid even if we end up with a void graph after removal)

All 5 arguments proved!!

Now talking about the time complexity:

1. Calculating the in-degree of all the vertices will traversal across whole adjacency list which take $O(|V| + |E|)$ time
2. Finding the nodes with 0 in-degree and pushing them onto a stack would take $O(|V|)$ time
3. Now each vertex and its outgoing edges are explored atmost once, so it will take $O(|V| + |E|)$ time.(More precisely,each vertex is pushed only on one of the two stacks, and for every vertex poped out of a stack, we only explore its outgoing edges, so for every vertex v, time taken is $1 + outdegree(v)$ (where outdegree of a vertex is number of outgoing edges). Now if we sum this quantity over all the vertices, we end up with a function which is $O(|V| + |E|)$).

So overall time complexity of the algorithm is $O(|V| + |E|)$.

4. A particular video game involves walking along some path on a map that can be represented as a directed graph $G = (V, E)$. At every node in the graph, there is a single bag of coins that can be collected on visiting that node for the first time. The amount of money in the bag at node v is given by $c(v) > 0$. The goal is to find what is the maximum amount of money that you can collect if you start walking from a given node $s \in V$. The path along which you travel need not be a simple path.

   Design an algorithm for this problem. You are given a directed graph $G = (V, E)$ in adjacency list representation and a start node $s \in V$ as input. Also given as input is a matrix C, where $C[u] = c(u)$. Your algorithm should return the maximum amount of money possible to collect starting from s.

   (a) (9 points) Give a linear time algorithm that works for DAGs.

   Solution: The algorithm uses the fact that, optimal cost from a vertex

   $$v = Cost\ at\ that\ v + maximum\ of\ the\ optimal\ costs\ from\ all\ neighbouring\ vertices$$

   Where neighbouring vertices of v are all the vertices to which there is a direct path from v.
   Where *maximum of the optimal costs from all neighbouring vertices* is 0 if a node have no neighbours.
   (By optimal cost from/at any node, we mean maximum money that can be collected if we start from that node)
   The algorithm for the problem is :

   1. Pick the nodes in reverse order of topological ordering
   2. If the node have any outgoing edge, find the maximum of the accumulated money among all the neighbours and update the maximum money that can be accumulated at the given node as : maximum of the accumulated money among all the neighbours + money at that node.

3. Do it till we are left with no nodes.
4. Find the maximum accumulated cost among all the nodes

Note- We have used Linearize(G(V,E)) as a subroutine for topological sort, disscused in class
Psuedo code :

```
max_money(G(V,E),C,s):

    ARRAY list[V.size()]

    list = Linearize(G(V,E))

    list=reverse(list)

    max_acc_money[V.size()]



    for each vertex v:
        max_acc_money[v]=0

    for each vertex v in list:
        cmax=0
        for each edge(v,u) in E:
            if cmax < max(max_acc_money[u],cmax):
                cmax=max_acc_money[u]
        max_acc_money[v]=C[v]+cmax


    return max_acc_money[s]
```

Proof of Correctness:

1. Picking the nodes in reverse topological order ensures that before calculating the optimal cost at any nodes, optimal cost from all the nodes reachable from that node have already been calculated.
2. The problem exhibits optimal substructure as the maximum amount of money that can be collected from any node depends on the maximum amount of money that can be collected from its neighbouring nodes.
3. We have to prove that for this algorithm correctly computes optimal cost for all nodes in the reverse topological order.

We will be using an inductive proof:
Inductive hypothesis:
$P(n)$: Algorithm returns correct result for the optimal cost if we start with n^{th} node of the DAG in reverse topological ordering
Lets check for the base case, i.e., $P(1)$:
The only money that can be collected starting from the first node of the reverse topological ordering is the money at the node itself, as no other nodes are reachable from first node. Now computation that our algorithm does for the first node:

Optimal cost from $(1)^{st}$ vertex = Cost at $(1)^{st}$ + *maximum of the optimal costs from all neighbouring vertices*

Since it have no neighbours, maximum of the optimal costs from all neighbouring vertices is 0.

Therfore, optimal cost from a $(1)^{st}$ vertex = Cost at $(1)^{st}$.

Which is indeed true !!

Let the statement be true for all nodes of reverse topological ordering till $n = k$.

Now we analyze the $n = (k+1)^{th}$ node of the topological order:

By algorithm, optimal cost from a $(k+1)^{th}$ vertex = Cost at $(k+1)^{th}$ + *maximum of the optimal costs from all neighbouring vertices*

Now all the neighbouring vertices of $(k+1)^{th}$ will be the nodes appearing at k^{th} or lesser place in the reverse topological order. Since optimal costs have been assumed to be correctly calculated.

Therefore algorithm correctly calculates the optimal cost from $(k+1)^{th}$ node of the topological order!!

Now talking about the time complexity:

1. Time it takes sort the DAG topologically (Linearize) is $O(|V| + |E|)$
2. Time it takes reverse topological order is $O(|V|)$
3. Time it takes to initialize the max_acc_money is $O(|V|)$
4. Time it takes to find the max_acc_money from each node in reverse topological order is $O(|V|+|E|)$ (more precisely, time it takes for each node is $k(1+number of outgoing edges)$, therefore, if we sum it over all the nodes, it turns out to be $O(|V| + |E|)$)
5. Time to find maximum of max_acc_money from each node is $O(|V|)$

Therefore, overall time complexity for the algorithm is $O(|V| + |E|)$

(b) (9 points) Extend this to a linear time algorithm that works for any directed graph. (*Hint: Consider making use of the meta-graph of the given graph.*)

Give running time analysis and proof of correctness for both parts.

Solution:

1. Any directed graph can be decomposed into meta graph of SCCs.
2. We can collect all the money at nodes inside SCC if we are able to visit any node inside that SSC, because all the vertices are reachable to each other inside a SCC.
3. Also once we enter a SCC we can exit from any node in that SCC that has an exit because the path of our traversal need not be simple.
4. Also since values of money are positive we need to collect all the money inside an SCC to get maximum accumulation of money we need to take all the money.

So if we draw the meta graph of our strongly connected components we can treat the SCC as single node with money at that node equal to the sum of money on all the nodes inside that SCC.

So this meta graph will act as DAG and we can find the optimal cost using the previous algorithm in 4(b).

5. (18 points) You are given a DAG $G = (V, E)$ and want to determine if a path in G exists that visits every vertex exactly once. See Figure 1 for examples.

Figure 1: The DAG on the left does not have any path that visits all nodes but the DAG on the right has such a path $a \to c \to b \to d$.

(a) Design an algorithm for this problem. Your algorithm should output "yes" if G has such a path and "no" otherwise. Give running time analysis and proof of correctness.

Solution:

The question asks for the existence of the Hamiltonian path in the DAG.

Few definitions which we are going to use:

1. Path: Sequence of vertices $(v_1, v_2...)$ such that consecutive vertices v_i and v_{i+1} are connected by edges (directed from v_i to v_{i+1}). Can be also represented as $(v_1 \to v_2 \to v_3.....)$
2. Hamiltonian path: A path that visits all the vertices of the graph
3. In-degree of a vertex: Number of incoming edges to the vertex.
4. Null graph: Graph with no vertices.
5. If we have an edge from u to v then this directed edge can be represented as tuple (u, v)

Note that these definitions are in the context of directed graphs.

The algorithm goes like:

(a) Initialize the in-degree array (an array which tells the in-degree of all the vertices) using the adjacency list.

(b) Count the number of vertices with 0 indegree in the graph

(c) If we have more than 1 vertices in the graph with 0 in-degree, then the hamiltonian path does not exist, hence return "NO!!".

(d) Else remove the vertex with 0 indegree from the graph (and the corresponding edges) and update (reduce) the in-degree of all the vertices which are adjacent to the vertex with 0 indegree.

(e) If we are not left with any vertex (after removal) in the graph, return "Yes!!", else go to the step 3

(Proof of correctness is given after psuedo code)

Psuedo code of the algorithm is:

Note- We have used 1 based indexing

Note- By updating the indegree of adjacent vertices, it implicitly means we are removing edges from the graph, without actually modifying the adjacency list. And by dequeing a node, it implicitly means that node is being removed from the graph without modfying the set V

Note- In $G(V, E)$, E represent the adjaceny list and V represent the set of edges

Note- We assume that reader know standard data structures and programming constructs

```
return_answer(G(V,E)):
    n = V.size()

    ARRAY indegree[n]
    for each vertex v:
        indegree[v]=0


    for all (u,v) in E:
        indegree[v]=indegree[v]+1

    QUEUE q;
    for each vertex v:
        if indegree[v]==0:
            q.enqueue(i)


    while not q.empty():
        if q.size()>=2
            return "NO"
        u=q.dequeue()
        for all (u,v) in E:
            indegree[v]=indegree[v]-1
            if indegree[v]==0:
                q.enqueue(v)


    return "YES"
```

Note that the algorithm assumes that the hamiltonian path always exists for a null graph (graph with no nodes)

Also note that the algorithm assumes that only a DAG is given as input

Also note that we have assumed that a null graph is DAG

Please keep all these notes in mind while reading

To argue about the correctness of the algorithm we have to prove the following things:

(a) Argument 1: A non-null DAG (a DAG with atleast 1 vertex) will always have a vertex with 0 in degree.

(b) Argument 2: If we have more than two vertices in the DAG with 0 in-degree, than we cannot have a hamiltonian path in DAG.

(c) Argument 3: If we remove the vertex with 0 indegree from a non-null DAG than we are still left with a DAG.

(d) Argument 4: If a hamiltonian path exist in a non-null DAG, then their exist a hamiltonian path in the graph obtained after removing the vertex with 0 in-degree.

Note that we have assumed that we have a "finite" DAG

Lets prove them one by one:

Argument 1: A non-null DAG (a DAG with atleast 1 vertex) will always have a vertex with 0 in degree.

We will prove it via contradicton:

Lets say we have a non-null DAG in which all the vertices have a non-zero in-degree.

Let us pick up a vertex say, v_i. Now since v_i have a non-zero in-degree, we will have a vertex v_{i-1} such that we have a directed edge from v_{i-1} to v_i. Now since v_{i-1} also have a non-zero in-degree, so we have another vertex v_{i-2} such that we have a directed edge from v_{i-2} to v_{i-1}. It will keep on repeating.

Now since the graph is finite, some vertex will repeat in the sequence of path ($... \rightarrow a \rightarrow b.... \rightarrow v_{i-2} \rightarrow v_{i-1} \rightarrow v_i$). This implies existence of a cycle. But our graph is acyclic, which is a contradiction!!

Hence it cannot be the case that we have a non-null DAG in which all the vertices have a non-zero in-degree. Their have to be some vertex in a non-null DAG with 0 in-degree

Argument 2: If we have more than two vertices in the DAG with 0 in-degree, than we cannot have a hamiltonian path in DAG

We will prove the contraposition of this argument, which is given by:

"If we have a hamiltonian path in DAG then we cannot have more than two vertices with in-degree 0 in the graph "

We will prove it in a straightforward way:

Lets suppose we have a hamiltonian path in the DAG. Now the existence of a hamiltonian path implies that we have a sequence of vertices, $(v_1, v_2, v_3, v_4....v_n)$ such that we have edges $\{(v_1, v_2), (v_2, v_3), (v_3, v_4), ...(v_{n-1}, v_n)\}$ in the graph (here n is the number of vertices in the graph). This implies that non of the vertices in $\{v_2, v_3, v_4, ..., v_{n-1}, v_n\}$ can have in-degree as 0 (since, every incoming edge to a vertex contributes to its indegree, and every of these vertices must have an incoming edge for the sequence of path to exist). Therefore, we can't two or more vertices with indegree 0!!

Argument 3: If we remove the vertex with 0 in-degree from a non-null DAG than we are still left with a DAG.

Lets prove it via contradiction:

After removing a vertex with in-degree 0 from a non-null graph, we could have two cases:

case 1: We are left with a Null graph. Now a null graph is assumed to be a DAG for our algorithm

case 2: We are left with a non null graph: Lets say that we are left with a cycle in the graph after removing the vertex with in-degree 0 from DAG. In that case, even if we re-add the removed vertex with in-degree 0 and its corresponding edge, we still end up with graph with a cycle. So it was not a DAG to begin with!! Which is a contradiction!!

So if we remove the vertex with 0 in-degree from a non-null DAG than we are still left with a DAG.

Argument 4: If a hamiltonian path exist in a non null DAG, then their exist a hamiltonian path in the graph obtained after removing the vertex with 0 in-degree.

To prove this argument, we first have to prove that **" for any directed graph if a hamiltonian path exists than node with 0 in-degree cannot appear after any other nodes in the hamiltonian path"** . This statement will be used in the proof of the **argument** 4, so lets prove it first!!

We will again use prove by contradiction:

Lets suppose we have hamiltonian path in any arbitrary directed graph such that node with in-degree 0, say u, appear "just" after some other node, say v in the hamiltonian path. In that case we have an edge (v, u) in the graph. So in-degree of of u cannot be zero. Hence contradiction!!

So for any directed graph if a hamiltonian path exists than node with 0 in-degree cannot appear after any other nodes in the hamiltonian path

Now coming back to the prove of **arguement** 4:

"If a hamiltonian path exist in a non null DAG, then their exist a hamiltonian path in the graph obtained after removing the vertex with 0 in-degree. "

Now in-degree 0 node always exist in a non null DAG (say u) and if their exists a hamiltonian path in the DAG than, u cannot appear after some other node in the hamiltoian path. So if a hamiltonian path exists in DAG and if u is the start node of hamiltonian path, it will look like $(u, a, b, c....)$. So even if we remove all nodes with indegree 0 and their corresponding edges in the graph, remaining graph will still have the hamiltonian path of the form $(a, b, c, ...)$, which visits all the vertices. And in case if we are left with a null graph, then our algorithms assumes it to be have a hamiltonian path.

All four arguments proved!!

Now talking about the time complexity:

1. Initializing the indegree array will take $O(|V|)$
2. Calculating the in-degree of all the vertices will traversal across whole adjacency list which take $O(|V| + |E|)$ time
3. Finding the nodes with 0 in-degree and enqueuing them would take $O(|V|)$ time
4. Now each vertex and its outgoing edges are explored atmost once, so it will take $O(|V| + |E|)$ time.(More precisely, for every vertex dequeued, we only explore its outgoing edges, so for every vertex v, time taken is $1 + outdegree(v)$ (where outdegree of a vertex is number of outgoing edges). Now if we sum this quantity over all the vertices, we end up with a function which is $O(|V| + |E|)$).

So overall time complexity of the algorithm is $O(|V| + |E|)$.

(b) Argue that there exists a path that visits every node in a DAG if and only if the DAG has a unique topological ordering of nodes.

Solution:

We will be using following properties of DAGs and topological ordering discussed in lecture slides (please click on the link and see slide 42 and 43):

1. Every edge in a DAG goes from a higher post number vertex to lower post number vertex (Post numbers and pre-numbers are assigned to a vertex after we call Depth first search on a DAG).
2. Linearization of a DAG (Topological ordering): Since we know that edges go in the direction of decreasing post number vertices, if we order the vertices by decreasing post numbers then we will have a linearization. Linearization will be in the form of list of vertices in decreasing order of post numbers.

Now coming back to the question:

"There exists a path that visits every node in a DAG if and only if the DAG has a unique topological ordering of nodes."

Note the definition of the hamiltonian path: "A path in the graph which visits all the vertices!!"

So the argument can be re-written as:

"There exists a hamiltonian path in a DAG if and only if the DAG has a unique topological ordering of nodes."

The argument can be broken down into two parts:

1. If their is a hamiltonian path in a DAG then DAG has a unique topological ordering of nodes.
2. If DAG has a unique topological ordering of nodes then their is a hamiltonian path in a DAG.

Now lets prove two parts:

case 1. If their is a hamiltonian path in a DAG then DAG has a unique topological ordering of nodes:

Let the directed acyclic graph have hamiltonian path $v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to ... \to v_n$. Where n is the number of vertices in the graph. Now this hamiltonian path will be a valid topological ordering. This is because, since it is a DAG, all the edges appearing in the hamiltonian path have to go from the vertices from with higher post number to the vertices with lower post number. So vertices will appear in the hamiltonian path in decreasing order of post numbers, hence sequence of vertices in the hamiltonian path represent a valid topological ordering.

Now this topological ordering would be unique (Lets use contradiction). To argue about the uniqueness of the this topological ordering, lets say we have different valid topological ordering λ different from the hamiltonian path vertices. Then their must be atleast one pair of consecutive vertices of hamiltonian path, v_i and v_{i+1} such that v_{i+1} appear before v_i in λ (because λ is assumed to be a different from the ordering of hamiltonian path). But since all the edges in hamiltonian path of a DAG have go from a vertices of higher post number to lower post number, it can't be the case that v_{i+1} appear before v_i in any valid topological ordering (by definition, a topological ordering is valid if vertices appear in decreasing order of post number). So λ is not a valid toplogical ordering. Which is a contradiction.

Hence, if their is a hamiltonian path in a DAG then DAG has a unique topological ordering of nodes!!!

case 2. If DAG has a unique topological ordering of nodes then their is a hamiltonian path in a DAG.

We will be using the fact that "For any DAG if we have a unique topological order $(v_1, v_2, v_3..., v_n)$, than for every consecutive vertices, v_i and v_{i+1} in topological order, we have a directed edge (v_i, v_{i+1}) in the graph". So first we will prove this fact:

For a DAG lets say we have a unique topological ordering $(v_1, v_2, v_3..., v_n)$. and lets say that we have an arbitrary pair of consecutive vertices v_i and v_{i+1} in this unique topological ordering such that we do not have an edge (v_i, v_{i+1}) in the DAG. We have following two scenarios:

1. v_{i+1} is reachable from v_i : In that case v_i will have an out going edge to some other vertex, say a, from which v_{i+1} is reachable. So in the topological ordering a will have to appear after v_i and v_{i+1} have to appear after a. So topological ordering will be: $(v_1, v_2, v_3, .., v_i, a, ..v_{i+1}..., v_n)$. But since topological ordering is unique. Hence, a contradiction
2. v_{i+1} is not reachable from v_i: In that case v_{i+1} and v_i can appear in any order in the topological ordering. Therefore, another valid topological ordering could be $(v_1, v_2, v_3, ..v_{i+1}, v_i, ..., v_n)$. But since topological ordering is unique. Hence, a contradiction

So "For any DAG if we have a unique topological order $(v_1, v_2, v_3..., v_n)$, than for every consecutive vertices, v_i and v_{i+1} in topological order, we have a directed edge (v_i, v_{i+1}) in the graph"

Now coming back to the case 2 : "If DAG has a unique topological ordering of nodes then their is a hamiltonian path in a DAG."

If a DAG have a unique topological ordering $(v_1, v_2, v_3..., v_n)$, than for every consecutive vertices, v_i and v_{i+1} in topological order, we have a directed edge (v_i, v_{i+1}) in the graph. So we have a path $(v_1 \to v_2 \to v_3, ... \to v_n)$ which visits all the vertices. Hence we have a hamiltonian path!!

6. Given a directed graph $G = (V, E)$ that is not a strongly connected graph, you have to determine if there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected. In other words, you have to determine whether there exists a pair of vertices $u, v \in V$ such that adding a directed edge from u to v in G converts it into a strongly connected graph. Design an algorithm for this problem. Your algorithm should output "yes" if such an edge exists and "no" otherwise.
   (a) (9 points) Give a linear time algorithm that works for DAGs.

Solution: Few definitions we will be using:

1. If we have an edge from u to v then this directed edge can be represented as tuple (u, v)
2. Incoming and outgoing edge: An edge (u, v) is an incoming edge for v and an outgoing edge for u.
3. In-degree of a vertex: Number of incoming edges to the vertex.
4. Out-degree of a vertex: Number of outgoing edges from a vertex
5. Path: Sequence of vertices $(v_1, v_2...)$ such that consecutive vertices v_i and v_{i+1} are connected by edges (directed from v_i to v_{i+1}).
6. Source vertex: The vertex with 0 in-degree.
7. Sink vertex: The vertex with 0 outdegree.
8. Strongly connected graph: A directed graph is strongly connected if for all pair of vertices u and v of the graph, their is a path from u to v and their is a path from v to u.

Note that these definitions are in the context of directed graphs.
Note that these definitions are not universal, and vary from author to author.
Please keep all these definitions in mind while reading

Such pair of vertices in DAG only exists if we have only one node with in-degree 0 and only one node with out-degree 0, i.e., we have only one source and only one sink.

Following is a linear time algorithm for it:

step 1. Initialize the indegree and outdegree of all vertices

step 2. Count the number of vertices with indegree 0

step 3. If the number of vertices with indegree 0 is not equal to one, return "NO".(if number of such vertices with indegree 0 is 0, than its not a DAG in the first place. And if number of vertices with indegree 0 is greater than 1, than we have more than two sources)

step 4. Count the number of vertices with outdegree 0

step 5. If the number of vertices with outdegree 0 is not equal to one, return "NO".(if number of such vertices with outdegree 0 is 0, than its not a DAG in the first place. And if number of vertices with outdegree 0 is greater than 1, than we have more than two sinks)

step 6. return "Yes".

(Proof of correctness is given after psuedo code)

Psuedo code of the algorithm is:

Note- We have used 1 based indexing

Note- In $G(V, E)$, E represent the adjaceny list and V represent the set of edges

Note- We that reader know standard data structures and programming constructs

```
return_answer(G(V,E)):
    n=V.size()

    ARRAY indegree[n]
    ARRAY outdegree[n]

    for each vertex v:
        indegree[v]=0
        outdegree[v]=0

    for all (u,v) in E:
        indegree[v]=indegree[v]+1
        outdegree[u]=outdegree[u]+1

    count_indegree=0

    for each vertex v:
        if indegree[v]==0:
            count_indegree=count_indegree+1

    if count_indegree != 1:
        return "No"

    count_outdegree=0

    for each vertex v:
        if outdegree[v]==0:
            count_outdegree=count_outdegree+1

    if count_outdegree != 1:
        return "No"

    return "YES"
```

To prove the correctness of the algorithm we have to prove the following:

Argument 1. If their exist more than one sources in a DAG $G(V, E)$ then it is not possible that there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected.

Argument 2. If their exist more than one sinks in a DAG $G(V, E)$ then it is not possible that there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected.

Argument 3. For a DAG, every vertex with a non-zero in-degree have a path from a source vertex .

Argument 4. For a DAG, every vertex with a non-zero out-degree have a path to a sink vertex.

Argument 5. For a DAG $G(V, E)$, if it have only one source and only one sink than there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected.

Argument 6. For a DAG $G(V, E)$, if there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected then it will have only one source and only one sink.

Note-We are considering Finite DAGs

Lets prove them one by one:

Argument 1. If their exist more than one sources in a DAG $G(V, E)$ then it is not possible that there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected.

Lets say we have a DAG with with more than one source vertices. Now in-order to add an edge to make it strongly connected, their are only following 4 possibilities:

1. We add an edge between any two sources: Lets add an edge from a source, say a, to another source, say b.(That is, we add an edge (a, b) to the DAG). In that case, a still have 0 indegree (no incoming edges). So non of the other verties have path from them to a. So the graph is still not strongly connected.

2. We add an edge from a source, say a, to a vertex with non-zero indegree: Adding such an edge will not increase the indegree of a. Since a don't have any incoming edges, we will not have anypath from any of the other vertices of the DAG to a. So the graph is still not strongly connected.

3. We add an edge from vertex with non-zero indegree to source: In this case also, since we have more than one source, one of the source, say b, will not have any incoming edge into itself. So non of the vertices of the graph will have a path from them to b. So the graph is still not strongly connected

4. We add an edge between two vertices with non-zero indegree: In this case, it won't be possible that their exist a path from any other vertex of the graph to any of the source.(As sources don't have any incoming edges) So the graph is still not strongly connected.

Argument 2. If their exist more than one sinks in a DAG $G(V, E)$ then it is not possible that there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected.

Lets say we have a DAG with with more than one sink vertices. Now in-order to add an edge to make it strongly connected, their are only following 4 possibilities:

1. We add an edge between any two sinks: Lets add an edge from a sink, say a, to another sink, say b.(That is, we add an edge (a, b) to the DAG). In that case, b will still have out-degree 0 (no out-going edges). So we will not have path from b to any of the other vertices of the graph. So the graph is still not strongly connected.

2. We add an edge from a sink to a vertex with non-zero out-degree: Since we have more than one sink, one of the sinks, say b, still won't have any outgoing edges.So we will not have path from b to any of the other vertices of the graph. So the graph is still not strongly connected.

3. We add an edge from vertex with non-zero outdegree to sink : Since sinks will not have any outgoing edges, we still will not have path from sinks to any other vertices of the graph. So the graph is still not strongly connected.

4. We add an edge between two vertices with non-zero out-degree: Since sinks will not have any outgoing edges, we still will not have path from sinks to any other vertices of the graph. So the graph is still not strongly connected.

Argument 3. For a DAG, every vertex with a non-zero in-degree have a path from a source vertex.

We will prove it via contradiction:

Lets say we have a vertex, say v_i in a DAG, such that it have a non-zero indegree and does not have a path from a source vertex.

Now since v_i have a non-zero indegree, it must have an edge incoming into it. Lets say we have an incoming edge from v_{i-1} to v_i. Now v_{i-1} can't have a zero indegree (as we can't have a path from a source to v_i).

Hence v_{i-1} will also have an incoming edge from a vertex say v_{i-2}. Now same goes for v_{i-2}, it can't also have zero indegree.(otherwise we will have a path from a source to v_i).

It will keep on happening. Since our graph is finite, some vertex in the path $(....v_{i-2}, v_{i-1}, v_i)$ will repeat. It implies the existence of a cycle. But our graph is DAG. Hence a contradiction.

Therefore, for a DAG, every vertex with a non-zero in-degree have a path from a source vertex.

Argument 4. For a DAG, every vertex with a non-zero out-degree have a path to a sink vertex.

We will prove it via contradiction:

Lets say we have a vertex, say v_i in a DAG, such that it have a non-zero out-degree and does not have a path to a sink vertex.

Now since v_i have a non-zero out-degree, it must have an edge outgoing from it. Lets say we have an outgoing edge from v_i to v_{i+1}. Now v_{i+1} can't have a zero out degree (as we can't have a path from v_i to a sink).

Hence v_{i+1} will also have an outgoing edge from it to vertex say v_{i+2}. Now same goes for v_{i+2}, it can't also have zero out degree.(otherwise we will have a path from a v_i to sink).

It will keep on happening. Since our graph is finite, some vertex in the path $(v_i, v_{i+1}, v_{i+2}, ...)$ will repeat. It implies the existence of a cycle. But our graph is DAG. Hence a contradiction.

Therefore, for a DAG, every vertex with a non-zero out-degree have a path to a sink vertex.

Argument 5. For a DAG $G(V, E)$, if it have only one source and only one sink than there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected.

Consider a DAG $G(V, E)$, which have only one source, say a, and only one sink, say b.

Lets add an edge (b, a) to the graph. If we can prove that $G' = (V, E \cup \{(b, a)\})$ is strongly connected, we can prove argument 5.

Since b was the only sink in G, so every vertex in G will have path to b (by argument 4). Since $E \cup \{(b, a)\} \subseteq E$, so in G' as well, every vertex will have path to b.

Also, since a was the only source in G, so every vertex in G will have path from a (by argument 3). Since $E \cup \{(b, a)\} \subseteq E$, so in G' as well, every vertex will have path from a.

Since we have an edge (b, a) in G' so we have a path from b to a in G'.

Therefore, for all pair of vertices u and v of the G', their is a path from u to v and their is a path from v to u. The path from u to v can be in the form of $(u, ..., b, a, ...v)$ and the path from v to u can be in the form of $(v, ..., b, a, ...u)$.

Hence G' is strongly connected.Argument 5 is proven!!.

Argument 6. For a DAG $G(V, E)$, if there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected then it will have only one source and only one sink.

It is equivalent (contraposition of) to argument 1 and argument 2. But for sake of completness we prove it.

We can prove it by contradiction:

Lets say for a DAG, there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected and it more than one source and more than one sink.

Now by argument 1 and argument 2, if we have more that one sinks or more than one sources, it is not possible that for a DAG, there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected.

Hence a contradiction.

So, for a DAG $G(V, E)$, if there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected then it will have only one source and only one sink.

Now talking about the time complexity of the algorithm:

1. Initializing the indegree and outdegree araay will take $O(|V|)$
2. Calculating the indegree and outdegree of each vertex requires traversing entire adjacency list, and will take $O(|O| + |E|)$
3. Counting the indegree and outdegree will take $O(|V|)$ time
4. Rest of the operations will take constant time

Hence the time complexity of the algorithm is $O(|V| + |E|)$

(b) (9 points) Extend this to a linear time algorithm that works for any directed graph. (*Hint: Consider making use of the meta-graph of the given graph.*)

Give running time analysis and proof of correctness for both parts.

Solution:

All the definitions and arguments used in part *a* are used here.

Before reading the algorithm, we have assumed that `CreateMetaGraph`(G) return a meta-graph of SCCs of the given graph. Also we will be using `return_answer`($G(V, E)$) as a subroutine defined in previous part.

The algorithm goes like :

Step 1. Create a meta graph G' from the input graph G

Step 2. The obtained graph will be a DAG of SCCs

Step 3. Call the subroutine `return_answer`($G'(V, E)$)

Step 4. Return what ever the answer is returned by subroutine `return_answer`

Pseudo code for the algorithm is :

```
return_answer_2(G(V,E)):

    G'=CreateMetaGraph(G)
    return return_answer(G'(V,E))
```

We have assumed that we are given a directed graph which is not strongly connected

We have already argued about the correctness of the algorithm used in the routine `return_answer` (read argument $1, 2, 3, 4, 5, 6$). To prove the correctness of the whole algorithm, we have to prove that:

Argument 7. Every directed graph is a DAG of strongly connected components.

Argument 8. If meta graph of SCCs(which will be a DAG, hence not strongly connected) have a pair of vertices, a and b, such that adding an edge (a, b) make it strongly connected, than original graph have a pair of vertices, u and v, such that adding an edge (u, v) make it strongly connected.

Arguement 9. If the original graph have a pair of vertices , u and v, such that adding an edge (u, v) make it strongly connected than meta graph of SCCs(which will be a DAG, hence not strongly connected) have a pair of vertices, a and b, such that adding an edge (a, b) make it strongly connected

Lets prove them one by one:

Argument 7. Every directed graph is a DAG of strongly connected components (SCCs).

For every directed graph, we could have two cases:

1. Directed graph is Strongly connected: In that case, graph of SCCs of the graph will have just a single node with no edges. Which is a graph without a cycle, hence a DAG.
2. Directed graph is not strongly connected: In that case, graph of SCCs of the graph cannot have a cycle. Because if their is a cycle, then path exists between every pair of vertices in the original graph. Hence graph of SCCs here as well is a DAG

Argument 8. If meta graph of SCCs (which will be a DAG, hence not strongly connected) have a pair of vertices, a and b, such that adding an edge (a, b) make it strongly connected, than original graph have a pair of vertices, u and v, such that adding an edge (u, v) make it strongly connected.

If meta graph of SCCs (which will be DAG, hence not strongly connected) have a pair of vertices, a and b, such that adding an edge (a, b) make it strongly connected, it means that after adding such an edge in the meta graph results in a graph such that we have a path from every vertices to every other vertices of the meta graph. Now we know that vertices of the meta graph are nothing but the collection of vertices of the original graph which are strongly connected. So adding the edge (a, b) in the graph will imply adding an edge from any of the vertex of the SCC represented by a to any of the vertex of the SCC represented by b. So after adding such an edge in the original graph,the resulting graph is strongly connected. (Because if after adding such an edge, the original graph is not stongly connected, so will be the SCCs of the graph, hence vertices of the meta graph will not be strongly connected, which will be a contradiction).

Argument 9. If the original graph have a pair of vertices , u and v, such that adding an edge (u, v) make it strongly connected than meta graph of SCCs(which will be a DAG, hence not strongly connected) have a pair of vertices, a and b, such that adding an edge (a, b) make it strongly connected.

If original graph have a pair of vertices, u and v, such that adding an edge (u, v) make it strongly connected, it means that after adding such an edge in the original graph results in a graph such that we have a path from every vertices to every other vertices of the original graph. Now we know that vertices of the meta graph (which will be a DAG, hence not strongly connected) are nothing but the collection of vertices of the original graph which are strongly connected. So adding the edge (u, v) in the graph will imply adding an edge from the vertex of the SCC which contains u to any of the vertex of the SCC which contains v. So after adding such an edge in the meta graph,meta graph becomes strongly connected.(because, if now meta graph is not strongly connected, then so will be SCCs of orginal graph, hence vertices of the original graph will not be stongly connected, which will be a contradiction).

Talking about the time complexity:

1. `CreateMetaGraph(`G`)` is assumed to be taking $O(|V| + |E|)$ time
2. `return_answer(`G`)` subroutine will take $O(|V'| + |E'|)$ time, where $|V'|$ and $|E'|$ is the number of vertices and edges in the meta graph of the SCCs. Now since $|E'| < |E|$ and $|V'| < |V|$, we can say that, it will take $O(|V| + |E|)$ time.

So the total time complexity of the algorithm is : $O(|V| + |E|)$