Name: Ananye Agarwal **Entry Number:** 2017CS10326
Name: Kabir Tomer **Entry Number:** 2017CS50410
Name: Rajat Jaiswal **Entry Number:** 2017CS50415

1. An undirected graph is said to be *connected* iff every pair of vertices in the graph are reachable from one another. Prove the following statement:

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

   We will prove the statement using Mathematical Induction. The first step in such a proof is to define the propositional function. Fortunately for this problem, this is already given in the statement of the claim.

   $P(n)$: 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) (5 points) Show that if the degrees of all nodes in G is at least 2, then G has at least k edges.

   Solution: Let us assume that the degree of each node in G is at least 2. The degree of a node is defined as the number of edges incident upon it. Edges are unordered 2-tuples therefore each edge is incident upon exactly two nodes. The sum of degrees of all nodes $\in V$ will therefore be twice the number of edges, since if we count edges for each node, each edge is counted twice (self edges are not allowed). Since for this case, $|V| = k+1$ and $\forall v \in V, deg(v) \geq 2$:

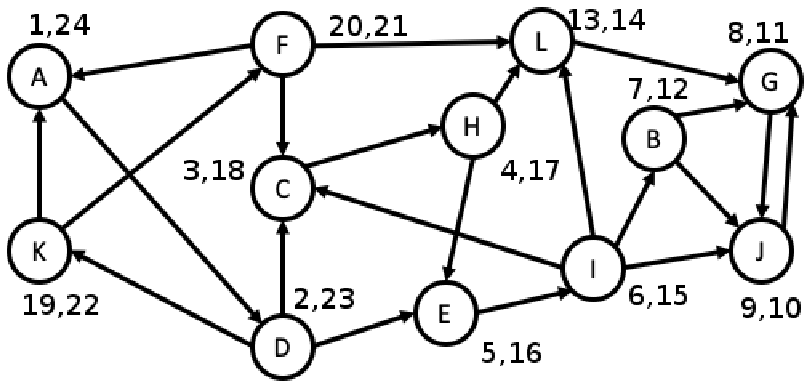
   $$\begin{aligned} 2|E| &= \sum_{v \in V} deg(v) \\ 2|E| &\geq \sum_{v \in V} 2 \\ 2|E| &\geq 2(k+1) \\ \Rightarrow |E| &> k \end{aligned}$$

   (b) (5 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: G' has k nodes, and is formed by removing vertex v along with its edge. Since G is connected, $\forall v_1, v_2 \in V, v_1, v_2 \neq v, \exists$ a path p between v_1 and v_2 in G.
   If $v \notin p$ then all nodes in p are present in G' as well. Since only the edge for v has been removed, p must exist in G' as well.

   If $v \in p$, then let v_1' and v_2' be the predecessor and successor of v in p respectively. Since $deg(v) = 1$, v has only one neighbour, which implies $v_1' = v_2'$ (successive nodes in a path must be neighbours). Therefore $v_1' \to v \to v_2'$ is a cycle and the path formed by removing v and v_2' from p is also a valid path from v_1 to v_2 in G. We can repeat this for every occurrence of v in p until we obtain path p' that does not contain v and so exists in G' as well. Therefore, $\forall v_1, v_2 \in V, v_1, v_2 \neq v, \exists$ a path p' between v_1 and v_2 in G', which means that G' is connected.
   Since G' is connected and has k nodes, the induction assumption applies, i.e. G' has at least $k-1$ edges. G' is formed by removing one edge from G, therefore G has atleast $(k-1)+1 = k$ edges.

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



(a) (1 point) Is the graph a DAG?

No, $A \rightarrow D \rightarrow K \rightarrow A$ is a cycle.

(b) (2 points) How many SCCs does this graph have?

5 SCCs: AFKD, HEIC, B, L, GJ. These may be found by `CreateMetaGraph(`G`)` and visually verified in reverse order.

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

1: AFKD. All nodes are reachable from A (as shown by DFS execution)

(d) (2 points) What is the distance of node B from the A?

By manual checking, nodes reachable from A in:
1 step: D
2 steps: D, C, E, K
3 steps: D, C, E, K, F, I, H
4 steps: D, C, E, I, F, I, H, L, B
Therefore the minimum steps required is 4, so the distance is **4**.

(e) (2 points) Suppose we run the DFS algorithm on the graph exploring nodes in alphabetical order. Given this, what is the pre-number of vertex F?

20 (see fig.)

(f) (2 points) Suppose we run the DFS algorithm on the graph exploring nodes in alphabetical order. Given this, what is the post-number of vertex J?

10 (see fig.)

3. (20 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:

```
MinQuarter(G):
  L ← TopologicalSort(G)
  Allocate array Q of size |V|
  Initialize Q to 0
  For each n in L:
     If Q[n] = 0:
       Q[n] ← 1
     For each out-neighbour m of node n:
       Q[m] ← Max(Q[n]+1, Q[m])
  Return Max(Q)
```

`TopologicalSort` runs in time $\mathcal{O}(V+E)$ and outputs a topologically sorted list of all nodes in G. The algorithm iterates through all the nodes and for each node also touches each outgoing edge. Each edge is visited once, and outermost for loop goes through each node once. Finding Max over the Q array touches each value once. Therefore the running time is $\mathcal{O}(V+E+V+E+V) = \mathcal{O}(V+E)$.

In a topologically sorted list, all edges go from left to right. The outermost loop processed each node from left to right, therefore, if a node is being processed, all nodes to the left of it have already been processed. This implies that all in-neighbours have been processed (since they must be to the left of the node being processed). $Q[n]$ is updated only when an in-neighbour of n or n itself is processed. We claim that after processing a node n, $Q[n]$ stores the earliest possible quarter in which the course corresponding to the node n can be taken. We prove this claim with induction over the length of L:

$P(i)$: $Q[L[i]]$ is the earliest possible quarter in which the course corresponding to node $L[i]$ can be taken.

$L[0]$ is the first element in the topologically sorted list, i.e. it has no in-neighbours, and therefore no prerequisites. The earliest its corresponding course may be taken is the first quarter. Since Q is initialised to zero, $Q[L[0]]$ is assigned 1 by the algorithm, and hence, $P(0)$ holds. We assume $P(0), P(1), ..., P(k)$ holds for arbitrary $k \geq 0$. If $Q[L[k]] = 0$ when $L[k]$ is being processed then L[k] does not have any in-neighbours (else they would have been processed and $Q[L[k]]$ would be atleast 1). Therefore its course can be taken in the first quarter, and $Q[L[k]]$ is assigned 1. If $L[k]$ does have in-neighbours, they would have been processed before $L[k]$. Each in-neighbour n would have assigned $Q[L[k]]$ as $max(Q[L[k]], Q[n]+1)$. Therefore when $L[k]$ is processed, $Q[L[k]]$ would have been assigned value 1+ maximum Q value for in-neighbours. By the induction assumption, each in-neighbour's Q value is the earliest quarter in which the corresponding course can be done. Since the course for $L[k]$ depends on in-neighbour courses, it can only be done after them. Therefore, the earliest quarter is 1+maximum Q value for in-neighbours, and $P(k+1)$ holds.
Therefore P holds $\forall i < |V|$ by the principle of mathematical induction.

The final result is the maximum Q value (which we may call R). Suppose the minimum number of quarters is less than R. Therefore, every course can be done in less quarters than R, and so every Q value is less than R. The maximum Q value is thus also less than R, which is a contradiction. Therefore R is less than or equal to the minimum number of quarters. Each course may be done in the quarter specified by the Q array since the only restriction is dependencies, and each course is taken only after all its dependencies have been taken (out-neighbour Q value $\geq$ node Q value + 1). Since R is greater than or equal to all Q values, it is enough for the completion of all courses, so we may conclude that R is the minimum number of quarters required.

4. A particular video game involves walking along some path in 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 that is possible to collect when starting from s. Give running time analysis and proof of correctness for both parts.

   (a) (10 points) Give a linear time algorithm that works for DAG's.

Solution:

Algorithm 1 Maximum amount of collectable money starting from s in a DAG

```
function MaxMoney-DAG($G, s, C$)
    $L \leftarrow TopologicalSort(G)$
    Allocate array $M$ of size $|V|$
    $M[v] \leftarrow 0$ for each $v \in V$

    for $i = n$ to 1 do
        $v \leftarrow L[i]$
        for each out-neighbour $u$ of $v$ do
            $M[v] \leftarrow Max(M[v], C[v] + M[u])$
        end for
    end for

    return M[s]
end function
```

<u>**Running time analysis:**</u> As discussed in the class `TopologicalSort` takes $O(V + E)$ time. Allocating and initialising the array $M[.]$ takes $O(V)$ time. Line 7, 8, 9, 10 takes $O(1 + outdeg(v))$ time for each vertex $v \in V$. Hence summing it over all vertices, gives $O(V + E)$ time. So, the overall running time of the algorithm is $O(V + E)$.

<u>**Proof of correctness:**</u> Let M[v] be the element of the array denoting the maximum amount of money collectable when starting the walk from vertex $v \in V$. Let $u_1, u_2, u_3, ..., u_k$ be the out-neighbors of v. If we compute M in the reverse topological order of the vertices, then the following claim holds, which we will prove using strong induction.

$$\textbf{Claim: } M[v] = C[v] + \max_{i \in \{1,2,3,...,k\}} M[u_i]$$

Induction Hypothesis: Let $P(i)$ denote that $M[R[i]]$ is the maximum amount of money collectable when starting the walk from $R[i]$, where R is the reverse-list of topological ordering L.

Base Case: $R[0]$ corresponds to a sink node in G since it is the last vertex appearing in the topological ordering L of the given G. If there was an out-neighbor of $R[0]$, it would have occured to right of $R[0]$ in L. Since, sink nodes have no out-neighbors, essentially $M[R[0]] = C[R[0]]$. Starting from a sink node we don't have any other place to go, therefore, the value of C corresponding to the sink node will be the maximum amount that we can collect. Hence, P(0) is true.

Inductive Step: Let $P(0), P(1), P(2), .., P(k)$ hold, then $P(k+1)$ is true. All the out-neighbors of $R[k+1]$ appear to the left of $R[k+1]$ in R because they appear to the right of $L[|V|-1-k-1]$ in topological ordering. Hence, the induction hypothesis holds for all the out-neighbors because their index in R is less than $k+1$. Using the induction hypothesis, we are guaranteed to know the maximum amount collectable at each of the out-neighbor. When we start from $R[k+1]$, we will be collecting the amount $C[k+1]$ and then walk along the edge to that out-neighbor which will give us the maximum amount. If we can collect more than that, then it violates the induction hypothesis for the out-neighbors, hence a contradiction. Therefore, P(k+1) is true.

(b) (10 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.*)

Solution: The idea is to reduce the given directed graph to its meta-graph, and since meta-graphs are $DAGs$, we will invoke the algorithm from the previous part to solve this problem. Let $V_1, V_2, V_3, .., V_k$ be the SCCs of G, and consider the meta-graph of G as G^m of the form $(\{1, 2, 3, ..., k\}, E^m)$. We define, cost corresponding to each vertex i in the meta-graph as the sum total of $C[u]$ for all $u \in V_i$. This cost is stored in $C^m[i]$ for all vertices i of meta-graph. If $s \in V_j$, we will run the algorithm of the previous part to compute $M[j]$ for the vertex j of the meta-graph. This will be maximum amount of money that will be collected when we start from s in G.

Algorithm 2 Maximum amount of collectable money starting from s in any directed graph.

```
function MaxMoney($G, s, C$)
    $G^m \leftarrow CreateMetaGraph(G)$
    Consider $G^m$ is of the form $(\{1, 2, 3, ..., k\}, E^m)$ and $V_1, V_2, V_3, .., V_k$ be the SCCs of $G$.

    Allocate array $C^m$ of size $k$
    $C^m[i] \leftarrow 0$ for all $i = 1$ to $k$

    for $i = 1$ to $k$ do
        for each $v \in V_i$ do
            $C^m[i] \leftarrow C^m[i] + C[v]$
        end for
    end for

    Let $s \in V_j$
    return MaxMoney-DAG($G^m, j, C^m$)
end function
```

Running time analysis: As discussed in the class, `CreateMetaGraph` takes $O(V+E)$ time. Allocating and initialising the array $C^m[.]$ takes $O(V)$ time at max, because G^m is a DAG with at most $|V|$ vertices and $|E|$ edges. Line $9, 10$ takes $O(|V_i|)$ time for each vertex i in G^m. Hence summing it over all vertices, gives $O(V)$ time. So, the overall running time of the algorithm is $O(V+E)$.

Proof of correctness: Let $s \in V_j$. Starting from s, we can collect all the money from all the vertices $v \in V_j$, because it is an SCC and a path for each pair of vertices $u, v \in V_j$ exists. $C^m[j]$ stores the value of this amount for vertex j of metagraph. Therefore, we can collect money from any vertex $v \in V_i$ such that vertex i of meta-graph lies along the path starting from vertex j of metagraph. Hence, the maximum amount of money collectable, starting from s, is the maximum money collectable starting at vertex j in DAG G^m, which is precisely the value returned by `MaxMoney-DAG(G`$^m, j, C^m$`)`.

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

   (a) (10 points) Give a linear time algorithm that works for DAG's.

Solution: Call graphs which satisfy this property *one-connectable*.
The function IsOneConnectable-DAG takes as input the DAG $G = (V, E)$ and returns whether the graph satisfies this property.

Algorithm 3 Check if a DAG is one-connectable

```
1: function IsOneConnectable-DAG($G = (V, E)$)
2:     $num\_sinks \leftarrow 0$
3:     $num\_sources \leftarrow 0$
4:     $G^R \leftarrow$ Reverse($(V, E)$)
5:
6:     for $v \in V$ do
7:         if $v$ is a source in $G^R$ then
8:             $num\_sinks++$
9:         end if
10:        if $v$ is a source in $G$ then
11:            $num\_sources++$
12:        end if
13:    end for
14:
15:    if $num\_sources = 1$ and $num\_sinks = 1$ then
16:        return yes
17:    else
18:        return no
19:    end if
20: end function
```

Note that checking each if condition in line 7 requires time $\mathcal{O}(1 + \text{indeg}(v))$ while checking condition in line 10 requires time $\mathcal{O}(1 + \text{outdeg}(v))$. Summing over all $v \in V$ we get that the for loop in lines 6-13 takes time $\mathcal{O}(V + E)$. Reversing the graph in line 4 takes $\mathcal{O}(V + E)$ time. Since all other statements take constant time, the overall time complexity is $\mathcal{O}(V + E)$. Note that line 7 is amounts to checking whether v is a sink in G. To prove correctness we show that the following holds

Claim 5.1: IsOneConnectable-DAG(V, E) returns **yes** iff (V, E) is one-connectable.

$\boxed{\Rightarrow}$ Suppose IsOneConnectable-DAG(V, E) returns **yes**. This means that there is exactly one source v and one sink u in G.

Lemma 5.1 : There exists a path from the source v to every vertex $w \in V$. Similarly, there exists a path from every $w \in V$ to the sink u.

Proof : Let us prove the first part of the lemma. Suppose L is a topological sorting of the vertices in G. Clearly, v is the first vertex in L. Suppose w is the first vertex in L (from the left) such that there is no path from v to w. Clearly, w is distinct from v and cannot be a source (v is the only source). Further, since w is not a source it has atleast one incoming edge, say $(t, w) \in E$. t must occur earlier than w in L. This implies that t is reachable from v. However, this contradicts the statement that w is not reachable from v.

The second part of the lemma is proved similarly. Let w be the first vertex in L (from the right) from which u is not reachable. Since w is distinct from u and is not a sink, there exists an outgoing edge $(w, t) \in E$. Clearly, t occurs later than w in L. This implies u is reachable from t. This contradicts our initial assumption that u is not reachable from w. $\square$
Consider the graph $G' = (V, E \cup \{(u, v)\})$. We claim that it is strongly connected. Indeed, consider any two vertices $s, t \in V$. It is sufficient to show that there is a path from s to t. By lemma 5.1, we have that there is a path P_1 from s to the sink u, and a path P_2 from the source v to t. Since (u, v) is an edge in G', this implies that $P_1 \oplus (u, v) \oplus P_2$ is a path connecting s and t. (here $\oplus$ denotes concatenation of paths).
$\boxed{\Leftarrow}$ We prove the contrapositive of this. Suppose IsOneConnectable-DAG(V, E) returns **no**. This means that there is either more than one source or more than one sink (or both). Suppose there is more than one source in G. After adding the edge (u, v) atmost one of these multiple sources can be converted to a non-source vertex (since (u, v) is an incoming edge for only one vertex). This means, G' contains atleast one source. Since a strongly connected graph cannot have any sources, G' is not strongly connected.
Similarly, if G has more than one sink, and because adding the edge (u, v) converts at most one of them to a non-sink, we conclude that G' contains at least one sink. This implies that G' cannot be connected.
The above reasoning implies that G is not one-connectable. This completes the proof of the claim 5.1

(b) (10 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.*)

Solution: Consider the meta-graph $G^M = (V^M, E^M)$ of G, where $V^M \subseteq \mathcal{P}(V)$ (powerset of V) and $E^M \subseteq V^M \times V^M$. We make the following claim,

Claim 5.2: *G is one-connectable iff G^M is one-connectable.*

Firstly, notice that this claim gives us a simple algorithm for checking whether a general graph is one-connectable (refer IsOneConnectable). Note that this is linear time ($\mathcal{O}(V + E)$) as well since both CreateMetaGraph and IsOneConnectable-DAG are linear time.

Algorithm 4 Check if a graph is one-connectable

```
1: function IsOneConnectable($V, E$)
2:     $V^M, E^M \leftarrow$ CreateMetaGraph($V, E$)
3:     return IsOneConnectable-DAG($V^M, E^M$)
4: end function
```

Let us prove claim 5.2.
$\boxed{\Rightarrow}$ Firstly, observe that a graph is strongly connected iff its meta-graph has only one node. Suppose G is one-connectable, then after adding an edge (u, v) we get a strongly connected graph $G' = (V, E \cup \{(u, v)\})$. The meta-graph $G^M = (V^M, E^M)$ of G has atleast two nodes (since G is one-connectable and therefore not strongly connected).
Let u^M, v^M be the SCCs containing u, v respectively. Add the edge (u^M, v^M) to the meta-graph to get $G^{M\prime} = (V^M, E^M \cup \{(u^M, v^M)\})$. We will show that $G^{M\prime}$ is strongly connected. Choose any two vertices $s^M, t^M \in V^M$ and let $s \in s^M, t \in t^M$ be any vertices of G' in the respective SCCs. Since G' is strongly connected there exists a path $e_1, e_2, \ldots, e_k$ from s to t where each $e_i \in E \cup \{(u, v)\}$. We can convert this to a path $e_1^M, e_2^M, \ldots, e_k^M$ from s^M, t^M, by replacing each vertex in each edge e_i by the vertices in V^M corresponding to their SCC. Note that if any $e_i = (a, b)$ connects vertices in different SCCs, the edge $e_i^M = (a^M, b^M) \in E^M \cup \{(u^M, v^M)\}$ by construction where a^M, b^M are SCCs of a, b.

This means that after removing any self-loops, $e_1^M, e_2^M, \ldots, e_k^M$ is a valid path from s^M to t^M. This implies that $G^{M\prime}$ is strongly connected i.e. G^M is one-connectable.

$\boxed{\Leftarrow}$ Suppose G^M is one-connectable. This means that there exists an edge (u^M, v^M) such that $G^{M\prime} = (V^M, E^M \cup \{(u^M, v^M)\})$ is strongly connected. Suppose $u, v \in V$ are any two vertices in u^M, v^M respectively. We claim that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected. Let $s, t \in V$ be two vertices. If s, t lie in the same SCC of G^M, there exists a path from s to t and we are done. Suppose s, t lie in distinct SCCs s^M, t^M respectively. Since $G^{M\prime}$ is strongly connected, there exists a path from s^M to t^M namely $v_1^M, v_2^M, \ldots, v_k^M$ where $v_i^M \in V^M$. Further, we know that for each edge v_i^M, v_{i+1}^M there exist vertices $a_i, b_i \in V$ such that $a_i \in v_i^M, b_i \in v_{i+1}^M$ and $(a_i, b_i) \in E \cup \{(u, v)\}$. We can choose $a_1 = s$ and $b_k = t$ since $a_1 \in v_1^M = s^M$ and $b_k \in v_k^M = t^M$. Since b_i, a_{i+1} lie in the same SCC v_{i+1}^M, we can find a path P_i between them. This implies that we can construct a path $(a_1, b_1) \oplus P_1 \oplus (a_2, b_2) \oplus P_2 \oplus \ldots \oplus P_{k-1} \oplus (a_k, b_k)$ between s and t. This implies that G' is connected and G is one-connectable.

6. (20 points) Let us call any directed graph $G = (V, E)$ *one-way-connected* iff for all pair of vertices u and v at least one of the following holds:

   (a) there is a path from vertex u to v,

   (b) there is a path from vertex v to u.

   Design an algorithm to check if a given directed graph is one-way-connected. Give running time analysis and proof of correctness of your algorithm.

Solution: Overload the term and call vertices u, v that satisfy the above property one-way-connected. Therefore, a graph is one-way-connected iff all pairs of vertices are one-way-connected. Further, call a DAG *linear* iff for every topological sorting $v_1, v_2, ..., v_n$ of its vertices, (v_i, v_{i+1}) is an edge in the DAG $\forall 1 \leq i < n$.

Algorithm 5 Check if a graph is one-way-connected

```
function IsOneWayConnected($V, E$)
    $V^M, E^M \leftarrow$ CreateMetaGraph($V, E$)
    $(v_1^M, v_2^M, \ldots, v_n^M) \leftarrow$ TopologicalSort($V^M, E^M$)
    for $i = 1$ to $n$ do
        if $(v_i^M, v_{i+1}^M) \notin E^M$ then
            return no
        end if
    end for
    return yes
end function
```

Consider the following claim that implies the correctness of the function IsOneWayConnected which takes as input a graph and returns yes iff it is one-way connected.

Claim 6.1: G is one-way-connected iff its meta-graph is linear.

Note that lines 3-8 correctly check if G^M is linear since it is sufficient to pick any *one* topological sorting $L = v_1, v_2, \ldots, v_n$ and verify if (v_i, v_{i+1}) is an edge for each i. This is because if a graph is linear, it can have only one topological sorting $L = v_1, v_2, \ldots, v_n$. Indeed, if there are two distinct sortings L, L' there will exist $v_i, v_j, i < j$ in L such that v_j occurs before v_i in L'. This would be a contradiction since $v_i, v_{i+1}, \ldots, v_j$ would become a path from a higher to a lower position in L'.

Notice each of lines 2, 3 is $\mathcal{O}(V+E)$. Checking the if condition in line 5 involves iterating over the linked list corresponding to v_i^M which takes $\mathcal{O}(1+\text{outdeg}(v_i^M))$. When summed over i this amounts to $\mathcal{O}(V+E)$. Thus, IsOneWayConnected runs in $\mathcal{O}(V+E)$ time.

We now tackle the proof for claim 6.1.
$\boxed{\Rightarrow}$ Suppose G is one-way-connected. Let $G^M = (V^M, E^M)$ be the meta-graph of G. Consider $u^M, v^M \in V^M$. Let $u, v \in V$ be such that $u \in u^M$ and $v \in v^M$. Since G is one-way-connected, assume WLOG that there is a path $w_1, w_2, \ldots, w_n$ between u and v. For each vertex w_i there exists a $w_i^M \in V^M$ such that $w_i \in w_i^M$. This implies there is a path (possibly with self-loops) $w_1^M, w_2^M, w_3^M, \ldots, w_n^M$ connecting u^M to v^M. Therefore, G^M is one-way-connected as well.
Now, suppose G^M has a topological sorting of vertices $L = v_1^M, v_2^M, \ldots v_n^M$ such that there exists an i for which $(v_i^M, v_{i+1}^M) \notin E^M$. Let i_0 be the smallest such i. Note that since $v_{i_0}^M$ occurs before $v_{i_0+1}^M$ in L, it cannot be reachable from $v_{i_0+1}^M$. Therefore, one-way-connectivity implies that $v_{i_0+1}^M$ is reachable from $v_{i_0}^M$. Suppose the length of this path is k. Since each edge in a DAG is directed strictly from left to right in the sorting L, we get that the difference in positions of the vertices at the ends of this path is atleast k. But since $v_{i_0}^M$ and $v_{i_0+1}^M$ are adjacent in L, k is atmost 1. Since k cannot be zero, we get that the path has length one. But this is a contradiction since $v_{i_0}^M$ and $v_{i_0+1}^M$ are not connected by an edge as per our assumption. This shows that G^M must be linear.v
$\boxed{\Leftarrow}$ Suppose G^M is linear. Consider a topological sorting $L = v_1^M, v_2^M, \ldots v_n^M$ such that $(v_i^M, v_{i+1}^M) \in E^M$ for each i. Consider two vertices $s, t \in V$. Suppose they lie in the SCCs v_i^M, v_j^M respectively. If $i = j$, s, t lie in the same SCC and are strongly connected and therefore one-way-connected as well. Assume WLOG $i < j$. Since G^M is linear we get a path between v_i^M and v_j^M, namely, $v_i^M, v_{i+1}^M, \ldots, v_j^M$. For each edge (v_k^M, v_{k+1}^M), we can find an edge $(a_k, b_k) \in V$ such that $a_k \in v_k^M$ and $b_k \in v_{k+1}^M$. Since a_k and b_{k-1} lie in the same SCC v_k, there is a path P_k between them in G. Therefore, we get the following path between s and t in G.

$$P_1 \oplus (a_1, b_1) \oplus P_2 \oplus (a_2, b_2) \oplus \ldots \oplus (a_{n-1}, b_{n-1}) \oplus P_n$$

where P_1 is a path between s and a_1 (both lie in v_1^M) and P_n is a path between t and b_{n-1} (both lie in v_n^M). $\oplus$ denotes concatenation of paths. This implies G is one-way-connected.