

- 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 $P(1), P(2), \dots, P(k)$ 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.

Solution: Let $G = (V, E)$. If all vertices have a degree at least 2, then we have:

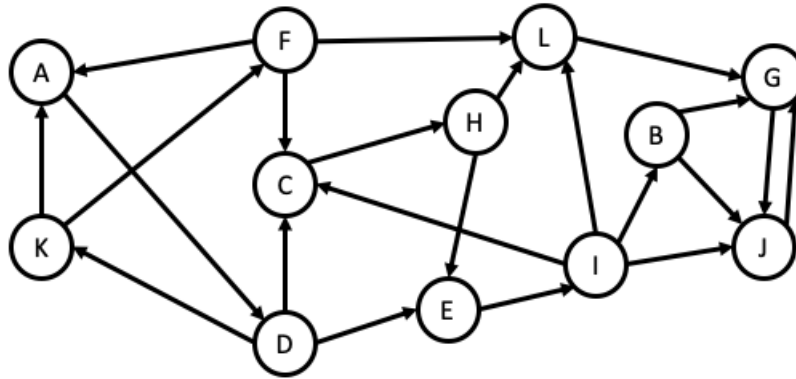
$$2|E| = \sum_{v \in V} \deg(v) \geq \sum_{v \in V} 2 = 2(k + 1)$$

The first equality above is because the sum of the degree of vertices in any undirected graph equals twice the number of edges. So, from the above, we get that $|E| \geq (k + 1) > k$.

- (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: We know that G' is a connected graph with k nodes since removing a degree one vertex along with its edge will not disconnect any other vertices. Now, from the induction assumption, we have that G' has at least $k - 1$ edges. This means that G has at least $(k - 1) + 1 = k$ edges.

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



- ($\frac{1}{2}$ point) Is the graph a DAG?
- (1 point) How many SCCs does this graph have?
- ($\frac{1}{2}$ point) How many source SCCs does this graph have?
- (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 ?
- (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 ?
- (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?

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:

Main idea: The courses we can take in the first semester are precisely the sources of the prerequisite graph G ; we might as well take them all. We can then remove them from the graph and take the sources of the remaining graph in the subsequent semester.

```
function ComputeMinQuarters( $G$ ):
  Compute indegrees in[.] and let currL be the list of source nodes

  time = 0
  repeat until currL is empty:
    time = time + 1
    nextL = empty list
    for all  $u$  in currL:
      for all neighbors  $w$  of  $u$  in  $E$ :
        in[ $w$ ] = in[ $w$ ] - 1
        if in[ $w$ ] is 0:
          add  $w$  to nextL
    currL = nextL
  Output time
```

The first part is the linear time indegree computation and source identification. Then the main loop begins, with the integer variable `time` containing the current semester number and `currL` containing the current list of courses. When these courses are removed from the graph, the indegrees of the nodes they point to are reduced; the main body of the loop performs the corresponding reductions and identifies the next set of source nodes, `nextL`.

Each course u is on `currL` during exactly one semester; at that point, its in-degree is already zero, and thus we never again consider edges into u . While u is on the current list, all the edges out of it are examined. Thus, in the repeat loop, all edges in the graph are examined exactly once over the entire execution of the algorithm. Therefore, the total time taken by the loop is $O(|V| + |E|)$.

We now give proof of correctness. Consider the solution where in quarter t , the student takes all the courses corresponding to the nodes with degree 0 in iteration t . This is a valid solution since all the pre-requisite courses have already been taken in one of the previous quarters for every course. We will also argue that for any course taken in quarter t per this solution, t is the earliest quarter in which this course can be taken. Let $P(t)$ denote this proposition. We will use mathematical induction to show $\forall i, P(i)$.

Base case: $P(1)$ holds trivially since quarter 1 is the least quarter that any course can be done.

Inductive step: Assume that $P(1), P(2), \dots, P(t)$ holds for an arbitrary $t \geq 1$. We will show that $P(t+1)$ holds. Consider any course c that has degree 0 in iteration $t+1$. Then there is a course c' that had degree 0 in quarter t that is a pre-requisite for c . This is because if there is no such

course, then the node for c will have degree 0 in the previous iteration (i.e., iteration t), which is a contradiction. From the induction hypothesis, we have that the earliest quarter that c' can be done is quarter t ; hence, the earliest quarter that c can be taken is quarter $t + 1$.

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: Main idea: We will design an algorithm that, for every vertex $v \in V$, computes the maximum money that can be collected when starting from v . In particular, this will give the maximum money when starting from s . Let the maximum amount of money that can be collected starting from vertex v be denoted by $p(v)$. For any vertex u in the DAG G , let $v_1, v_2, \dots, v_l$ denote all the vertices on the out-going edges of u . Then the following holds

$$p(u) = c(u) + \max_{i \in \{1, \dots, l\}} (p(v_i)).$$

We can use this equation to compute the p value for every vertex, but the order in which we compute these values is important. Suppose we compute the p values of all vertices in reverse topological order. In that case, we are guaranteed that for any vertex u and its neighbours $v_1, \dots, v_l$, the p values of $v_1, \dots, v_l$ are already known when we get to u as per the reverse topological ordering. The pseudocode for the algorithm based on this idea is given below:

MaximiseMoney(G, s, C)

- Initialize $p[v] \leftarrow 0$ and for all vertices $v \in V$
- Run DFS with pre-post numbers and use the post numbers to obtain a list L of vertices in topological order
- For $i = n$ to 1:
 - Let u be the i^{th} element of list L
 - For every neighbor v of u :
 - $p[u] \leftarrow \max(p[u], c[u] + p[v])$
- return($p[s]$)

Running time: Initialising array $p[\cdot]$ takes $O(V)$ time. Finding the topological ordering using DFS costs $O(V + E)$ (as per discussion in class). After this, the algorithm goes over the vertices, and for every vertex, it goes over its neighbours. This is essentially going over the graph's adjacency list, the running time of which is $O(V + E)$. So, the overall running time 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.*)

Solution: We will solve the problem on directed graphs by reducing it to the problem on DAGs (the algorithm we have already designed in the previous part). Let $V_1, \dots, V_k$ denote the strongly connected components of a given directed graph $G = (V, E)$. Consider the meta-graph $G' = (\{1, \dots, k\}, E')$ of G . We define the money at the meta-node i of the SCC V_i as the total of the money at nodes in SCC V_i . Now we can run the algorithm of the previous part on graph G' to compute the cost $p[1], \dots, p[k]$ of all the meta-nodes $1, \dots, k$. Let s belong to SCC V_i . Then the maximum money that can be collected is $p[i]$. This idea can be summarised as the following pseudocode:

```

MaximiseMoneyGeneral( $G, price$ )
- Run CreateMetaGraph( $G$ ) to obtain SCCs  $V_1, \dots, V_k$  and
  meta graph  $G' = (\{1, \dots, k\}, E')$ 
- For all  $i \in \{1, \dots, k\}$ :
  -  $C'[i] \leftarrow \sum_{v \in V_i} C[v]$ 
- Let  $j$  be the index such that  $s \in V_j$ 
- return(MaximiseMoney( $G', j, C'$ ))

```

Running time: Creating the meta-graph G' takes $O(|V| + |E|)$ time. Computing the money at the meta node, i takes $O(|V_i|)$ time. So, computing the money at all meta nodes takes $\sum_i O(|V_i|) = O(|V|)$ time. Since G' is a DAG with at most $|V|$ vertices and $|E|$ edges, the time for running **MaximiseMoney** is $O(|V| + |E|)$. So, the overall running time is $O(|V| + |E|)$.

The correctness of the algorithm follows from the following argument. One can collect money from all nodes in any SCC that lies along any path starting from the node s . So, the maximum amount of money that can be collected is the maximum money that can be collected starting at meta-node j in DAG G' .

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

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.

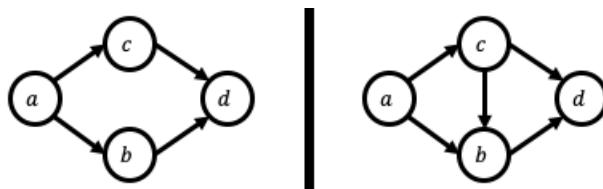


Figure 1: The DAG on the left does not have any path that visits all nodes but the DAG on the right has such a path $a \rightarrow c \rightarrow b \rightarrow d$.

- (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: Consider the following algorithm for the problem:

```

VisitAllVertices( $G$ )
- For  $i = 1$  to  $n$ 
  - If there are more than one vertex with in-degree 0 in  $G$ ,
    then return("No such path exists")
  - Let  $v$  be the unique vertex with in-degree 0 in  $G$ 
  - Let  $G'$  be the graph obtained from  $G$  by removing  $v$  and its outgoing edges
  -  $G \leftarrow G'$ 
- return("such a path exists")

```

Proof of correctness: The proof of correctness for the above algorithm follows from the next two claims.

Claim 1: If the algorithm outputs "no such path" exists, then there is no path in the graph that visits every vertex once.

Proof. For sake of contradiction, assume that there is a path P in the graph that visits every vertex once. The algorithm outputs "no such path exists" only when there is an iteration i such that the number of in-degree 0 vertices in that iteration is at least 2. Suppose u and v are two vertices in iteration i that have in-degree 0 in iteration i . Without loss of generality, let u be the vertex that is visited before v as per the path P . This means that there is a path from u to v . This means that in the beginning of the i^{th} iteration the in-degree of v cannot be 0. This is a contradiction. $\square$

Claim 2: If the algorithm outputs "such a path exists", then there is a path in the graph that visits every vertex.

Proof. The algorithm outputs "such a path exists" only when in every iteration of the for loop, it finds a unique vertex with 0 in-degree. Let the unique vertex in the i^{th} iteration be v_i . Then for all i , there is a directed edge from vertex v_i to vertex v_{i+1} in the graph (otherwise v_i will not be the unique vertex in iteration i with in-degree 0). This means that there is a path $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_n$ in the graph. $\square$

Implementation and running time: In every iteration, we need to find vertices with in-degree 0. Suppose we maintain a set S that contains all the 0 in-degree vertices at the beginning of

iteration i for all i . We will also maintain the in-degree of all the vertices in graph G at the beginning of iteration i for all i . Initially, we run the algorithm for finding the in-degrees for one of the previous homework problems to compute the in-degrees of all vertices of G and pick the ones with in-degree 0 to be put in set S .

Picking a vertex from the set S is an $O(1)$ operation. However, we will have to update this set S and in-degree information at the end of every iteration. How much does that cost? Suppose in the i^{th} iteration, we pick a vertex v . For the next iteration, we remove this vertex and all the directed edges from v to vertices in G . This changes the in-degree of vertices in G' . So, if v has an edge to $u_1, \dots, u_l$, then we decrease the in-degree of these vertices and if after decreasing the in-degree of a vertex, the in-degree becomes 0, then we add this vertex in the set S . The running time is proportional to the number of vertices plus the number of edges, which is $O(n + m)$.

- (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: Written solution not available.

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: The algorithm follows from the following main idea stated as a claim.

Claim 1: Let G be a DAG. There is a pair of vertices u, v such that adding a directed edge from u to v converts G into a strongly connected graph if and only if G has a unique source and sink vertex. Moreover, if G has a unique source and sink vertices, adding an edge from the unique sink vertex to the unique source vertex converts G into a strongly connected graph.

We break the if and only if part into the following two claims:

Claim 1.1: If a DAG G has a unique source s and a unique sink t , then adding a directed edge from t to s converts G to a strongly connected graph.

Proof. We claim that every vertex in G is reachable from the unique source s . Suppose, for the sake of contradiction, there is a vertex v which is not reachable from s . Let u_1 be the vertex with a directed edge to v . Note that such an edge exists; otherwise, v is also a source vertex. Again consider a vertex u_2 that has a directed edge to u_1 and keep traversing back along the incoming edges. This process will eventually terminate in a vertex u with no incoming edges. This vertex u cannot be equal to s since then v becomes reachable from s . Hence we get a contradiction with the fact that s is the unique source.

Similarly, we can argue that t is reachable from all vertices in G . So, adding a directed edge from t to s connects all pairs of vertices. Since then, one can reach t from any vertex u , take the edge (t, s) and then reach any other vertex v in the graph. $\square$

Claim 1.2: If adding a directed edge from t to s in a DAG G converts it to a strongly connected graph, then s is the unique source and t is the unique sink of G .

Proof. Suppose, for the sake of contradiction, s is not the unique source (t not being the unique sink will be similar). Then there is a topological ordering of the vertices such that s is not the first vertex in this topological ordering. Consider the first vertex v in this topological ordering. Even after adding the edge (t, s) , there is no path from s to v since otherwise, the same path will exist in G . $\square$

Claim 1 suggests that all we need to check is whether the given DAG has a unique source and a unique sink. This can be done by checking if there is a unique vertex with no incoming edges (this is the unique source) and a unique vertex with no outgoing edges (this is the unique sink). So, the algorithm simply computes the in-degree and out-degree of all vertices. The algorithm is summarised in the following pseudocode:

```

CheckConnectivity( $G = (V, E)$ )
- Compute  $indegree(u)$  and  $outdegree(u)$  for every vertex  $u \in V$ 
- If there is a unique vertex with 0 indegree and a unique vertex with 0 outdegree
  return(“yes”)
- else
  return(“no”)

```

Running time: The out-degree of a vertex v can be computed by counting the number of vertices in the linked list for v in the adjacency list of G . So, computing the out-degrees of all the vertices costs $O(V + E)$ time. Similarly, the in-degrees can be computed by doing the same

on the reverse graph, and we have seen that creating the reverse graph takes $O(V + E)$ time. So, the overall running time 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.*)

Solution: Consider the meta-graph G' of the given directed graph. Note that G' is a DAG. If G' has a unique source and unique sink, then adding a directed edge from a node in the sink SCC to a source SCC converts G' and hence G to a strongly connected graph. G' does not have a unique source and sink; then, by the same argument as before (claim 1.2), we conclude that it is impossible to create a strongly connected graph by adding a single edge. The algorithm based on the above idea is now simple. We construct the meta-graph DAG G' and check if this has a unique source and a unique sink vertex. If so, the algorithm outputs “yes”; otherwise, it outputs “no”. The pseudocode is given below.

CheckConnectivityGeneral(G)

- Use the algorithm developed in class to construct the meta-graph G'
- return(**CheckConnectivity**(G'))

Running time: Constructing G' takes $O(V+E)$ time. After this, we make a call to **CheckConnectivity**(G') that has $|V|$ vertices and $|E|$ edges in the worst case. So, the overall running time is $O(V + E)$.

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