

CSE 317: Design and Analysis of Algorithms

Graphs

Shahid Hussain
Fall 2025

SMCS, IBA

Definitions

Graph

A graph is a discrete structure consisting of two sets:

- a set of *vertices* and
- a set of *edges*.

Undirected and Directed Graph

An *undirected graph/directed graph* G is a graph with the pair (V, E) where V is a set of vertices and E is a set of edges. Each edge e is a two-element subset of V , $e = \{u, v\}$. For directed graphs we associate a *direction* on the edges and write $e = (u, v)$, instead.

Definitions

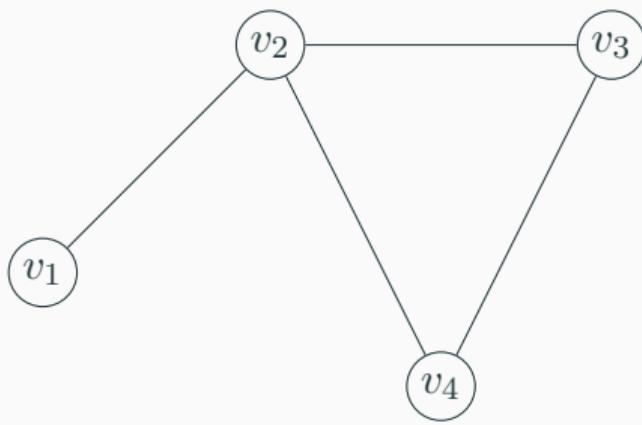
Paths

A *path* in undirected graph is a sequence $v_1, v_2, \dots, v_{k-1}, v_k$ of nodes such that $\{v_i, v_{i+1}\} \in E$, $i = 1, \dots, k - 1$. This path is called a *cycle* in undirected graph if $k > 3$, the first $k - 1$ nodes are pairwise different and $v_1 = v_k$.

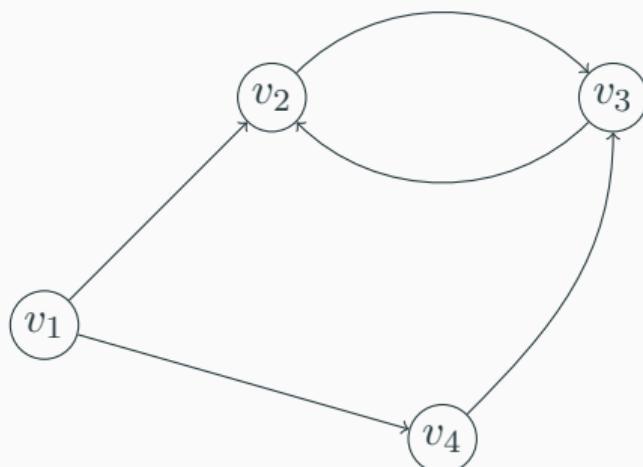
A *path* in directed graph is a sequence $v_1, v_2, \dots, v_{k-1}, v_k$ of nodes such that $(v_i, v_{i+1}) \in E$, $i = 1, \dots, k - 1$. This path is called a *cycle* in directed graph if $k > 2$, the first $k - 1$ nodes are pairwise different and $v_1 = v_k$.

Examples

Undirected graph $G = (V, E)$ where
 $V = \{v_1, v_2, v_3, v_4\}$ and $E =$
 $\{\{v_1, v_2\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}\}$



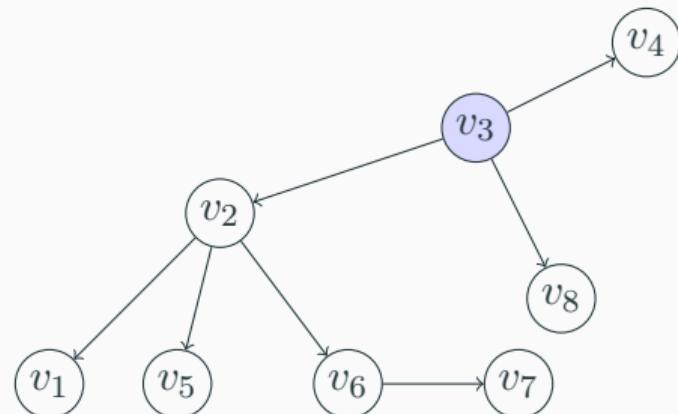
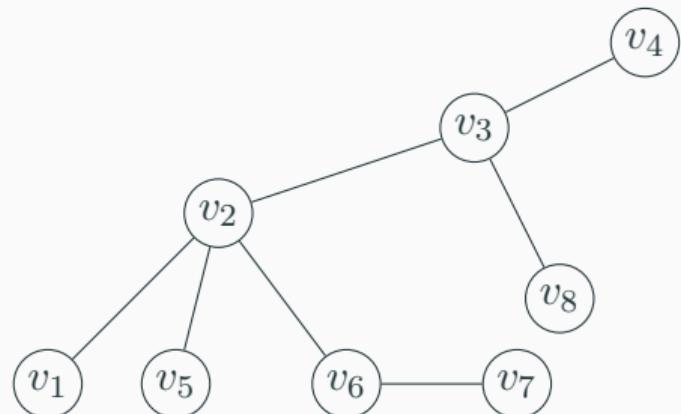
Directed graph (V, E) where
 $V = \{v_1, v_2, v_3, v_4\}$ and $E =$
 $\{(v_1, v_2), (v_2, v_3), (v_1, v_4), (v_4, v_3), (v_3, v_2)\}$



More definitions

- An undirected graph is connected if for every pair of nodes u and v , there is a path from u to v
- A directed graph is *strongly connected* if, for every two nodes u and v , there is a path from u to v and a path from v to u .
- An undirected graph is a *tree*, if it is connected and does not contain a cycle. If we choose one node of a tree as a *root* we obtain a rooted tree in which we “orient” each edge away from the root.

Trees



Theorem 1

Let G be an undirected graph with n nodes. Then the following three statements are equivalent:

- (a) *G is connected and does not contain a cycle.*
- (b) *G is connected and has $n - 1$ edges.*
- (c) *G has $n - 1$ edges and does not contain a cycle.*

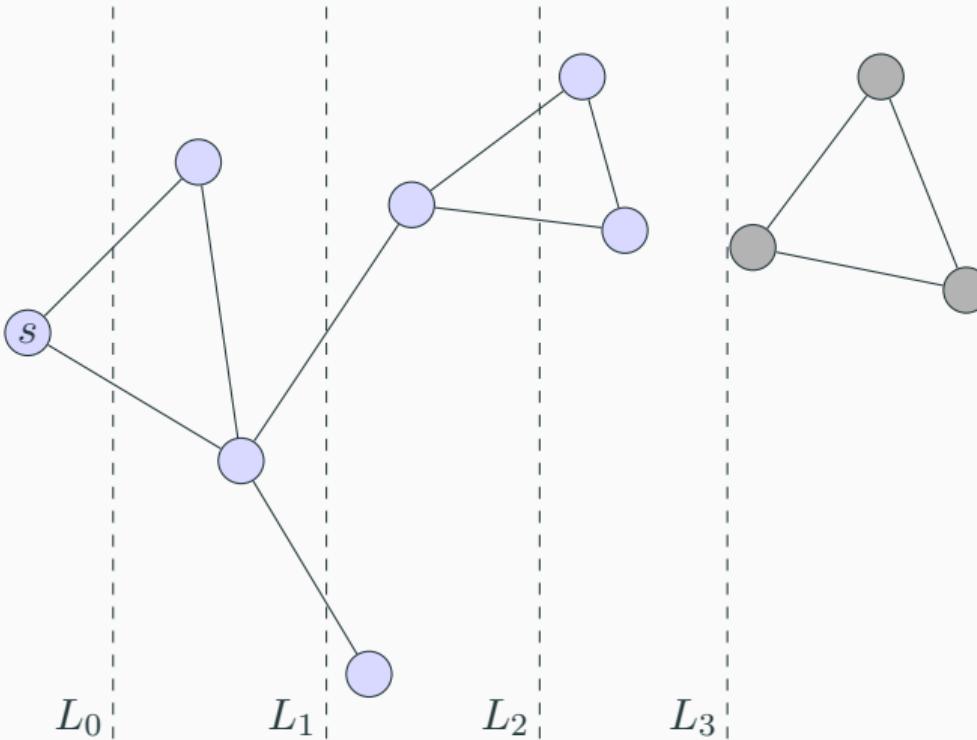
Representing Graphs

- To represent a graph we will use an adjacency list representation. For each node v of undirected graph we have a list of nodes u such that $\{u, v\} \in E$. For each node v of directed graph, sometimes it is useful to have two lists: the first one contains all nodes u such $(u, v) \in E$, and the second one contains all nodes u such that $(v, u) \in E$.

Breadth-First Search (Undirected Graphs)

- Let s be a node of an undirected graph $G = (V, E)$.
- Following is the BFS algorithm which allows us to find in G all nodes t such that there is a path from s to t .
- The considered algorithm constructs so called breadth-first search tree T for the graph G , all nodes of which are divided onto layers.
- The layer L_0 contains the only node s .
- Suppose, we have defined layers L_0, L_1, \dots, L_i . The layer L_{i+1} consists of all nodes v that do not belong to layers L_0, L_1, \dots, L_i and have an edge $\{u, v\}$ for a node u from L_i . We add each such node v and corresponding edge $\{u, v\}$ to the constructed tree T .
- It is clear that T contains all nodes t from G such that there exists a path from s to t , and only such nodes.

Example: BFS



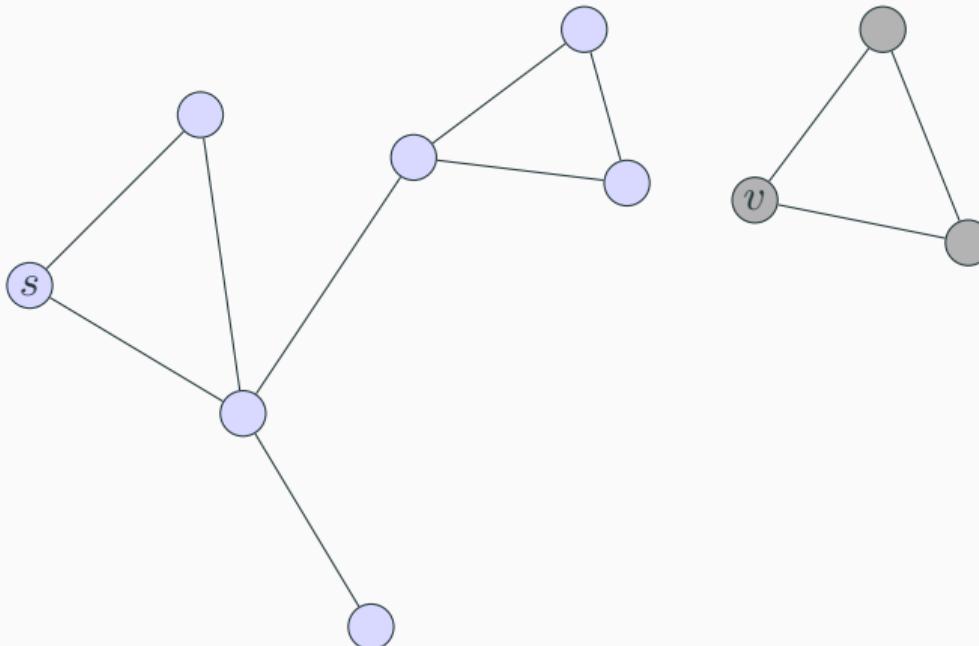
Implementation

- Let $|V| = n$ and $|E| = m$
- To implement the BFS we will use lists $L[0], L[1], \dots$ corresponding to layers L_0, L_1, \dots and array *Discovered* of length n such that $\text{Discovered}[v] = \text{True}$ if and only if v is already added to T
- Each node belongs to at most one list. Therefore we need $O(n)$ time to set up lists and manage the array *Discovered*
- Let $\{u, v\} \in E$, the algorithm will work with this edge at most two times: when the algorithm studies the node u , and when the algorithm studies the node v . In the first case we should recognize if $\text{Discovered}[v] = \text{False}$, and in the second case we should recognize if $\text{Discovered}[u] = \text{False}$
- Each such step requires $O(1)$ time. So the total time spent considering edges is $O(m)$, and the total time is $O(n + m)$

BFS and Connected Components

- We can use BFS to find all connected components of the undirected graph G
- A *connected component* of G is a maximal subgraph of G which is connected
- We can show that for any two nodes s and t connected components containing s and t respectively are either identical or disjoint
- We start with an arbitrary node s and use BFS to construct a tree T with the root s such that the set of nodes of T coincides with the set of nodes of connected component containing s
- We then find a node v (if any) that was not visited by the search from s , etc. We continue in this way until all nodes will be visited

Example



BFS in Directed Graphs

- Breadth-first search is almost the same in directed graphs as it is in undirected graphs
- Let $G = (V, E)$ be a directed graph and $s \in V$. The layer L_0 contains the only node s . If we have constructed layers L_0, \dots, L_i , then the layer L_{i+1} consists of all nodes v that do not belong to L_0, \dots, L_i , and have an edge (u, v) for a node u from L_i
- We can show that the constructed directed tree T with the root s contains all nodes t from G such that there exists a directed path from s to t , and only such nodes
- We can show also that the running time of this algorithm is $O(m + n)$ where $m = |E|$ and $n = |V|$

Strongly Connected Graphs

Strongly Connected Components

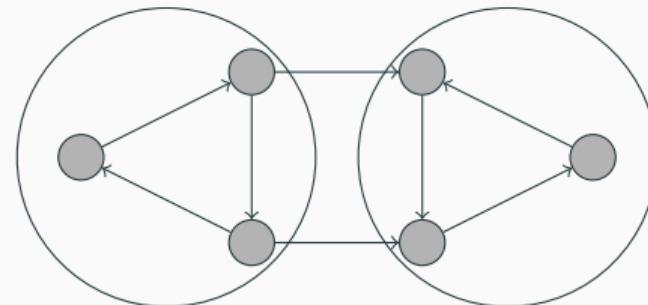
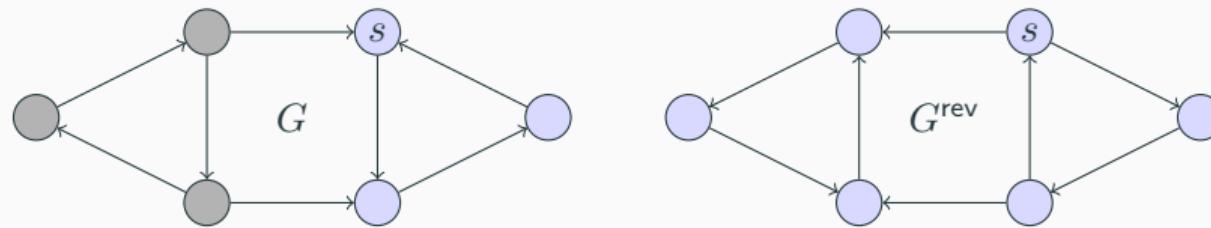
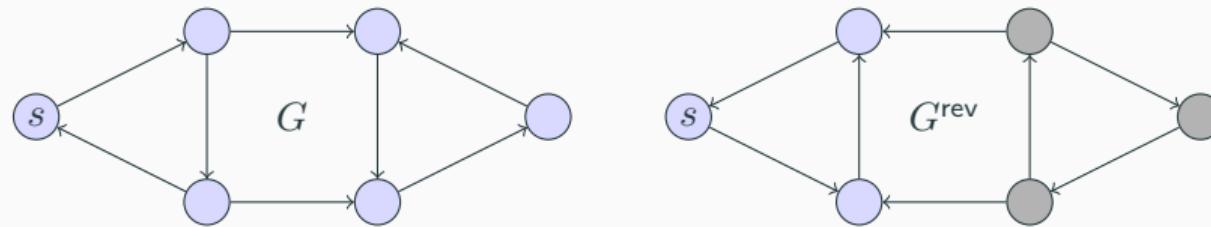
A *strongly connected component* is a maximal subgraph of G which is strongly connected. One can show that for any two nodes s and t , strongly connected components containing s and t respectively are either disjoint or identical.

- We can use BFS to find all strongly connected components of a directed graph G

BFS for SCC

- We denote by G^{rev} the directed graph which is obtained from G by reversing the direction of every edge. Let $s, t \in V$
- It is clear that there is a path from s to t in G^{rev} if and only if there is a path from t to s in G . We run BFS starting from s both in G and G^{rev}
- As a result we obtain two trees T_1 and T_2 with the set of nodes V_1 and V_2
- It is clear that the set of nodes in the strongly connected component containing s is equal to $V_1 \cap V_2$. If $V_1 \cap V_2 \neq V$ then we choose a node $v \in V \setminus (V_1 \cap V_2)$ and repeat the same operation with the node v , etc.

Example

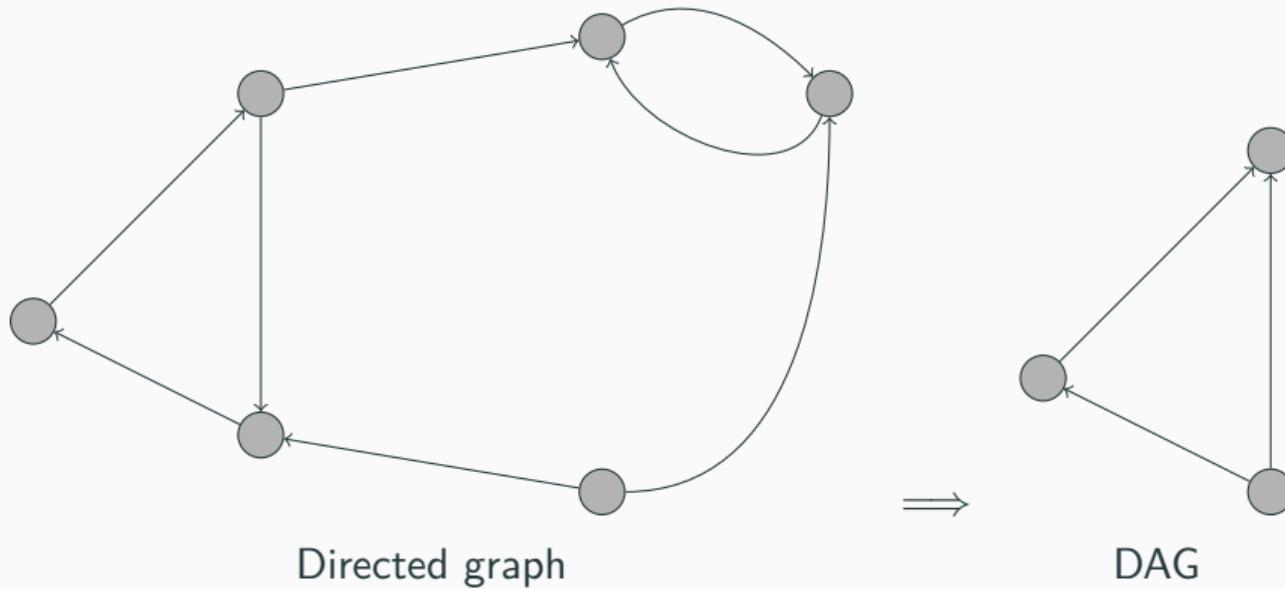


Directed Acyclic Graphs

Directed Acyclic Graphs

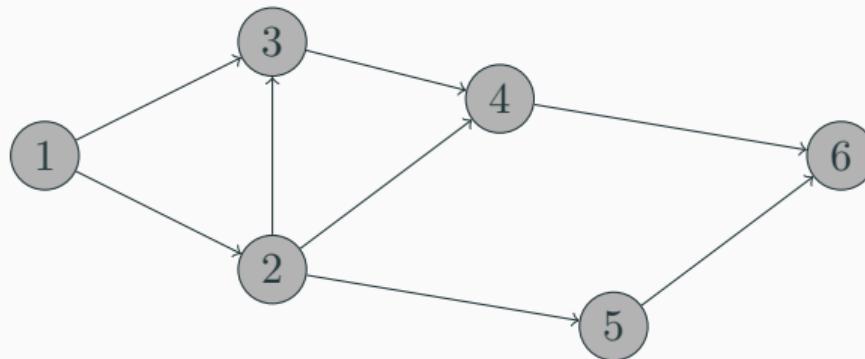
A *directed acyclic graph* (DAG for short) is a directed graph without cycles. Let G be a directed graph. We “join” each strongly connected component of G into a single meta-node, and draw an edge from one meta-node to another if there is an edge (in the same direction) between components of the considered meta-nodes. The resulting meta-graph is a DAG: a cycle containing several strongly connected components would “join” these components into a single strongly connected component.

Example



DAGs and Topological Ordering

- For a directed graph G , we say that a topological ordering of G is an ordering of its nodes as v_1, \dots, v_n so that for every edge (v_i, v_j) of G we have $i < j$. If G contains a cycle, then, obviously, G has no a topological ordering.



DAGs and Topological Ordering

- Let $G = (V, E)$ be a DAG
- We show that G has a topological ordering
- Since G has no cycles, there exists a directed path in G of maximum length (length here is the number of edges in the path)
- Let v be the first node in this path
- It is clear that v has no incoming edges
- This node will be considered as the first node v_1 in a topological ordering. We remove v_1 from G
- The obtained graph G' is also a DAG. In G' there exists a node u without incoming edges
- This node will be considered as the second node v_2 , etc

DAGs and Topological Ordering

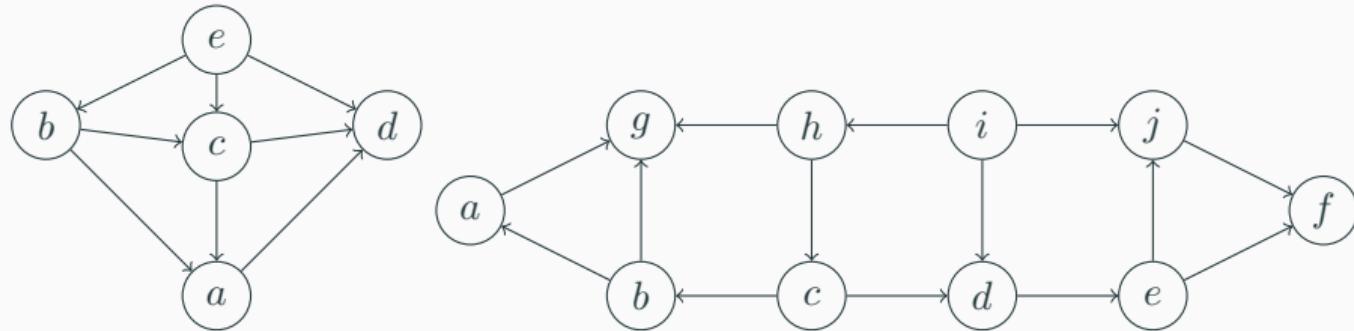
- To implement the considered algorithm efficiently, we consider the notion of active node, a node which was not deleted by the algorithm. We maintain two things:
 - (a) For each node w , the number of incoming edges that w has from active nodes.
 - (b) The set S of all active nodes in G that have no incoming edges from other active nodes.
- At the start, all nodes are active, so we can initialize (a) and (b) with a single path through nodes and edges
- Then each iteration consists of selecting a node v from the set S and deleting it
- After deleting v , we consider all edges of the kind (v, w) and subtract one from the number of active incoming edges for w
- If the obtained number is equal to 0 then we will add w to S
- During the iterations each edge will be considered at most one time
- Using this fact one can show that the overall time of the algorithm is $O(n + m)$ where $m = |E|$ and $n = |V|$

Depth-First Search

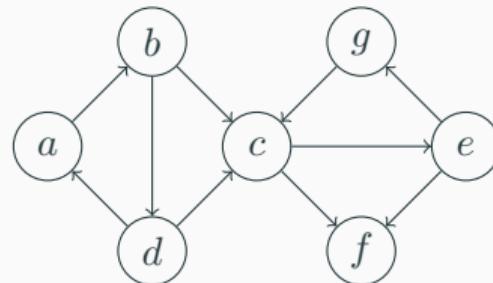
- We concentrate in this lecture on the consideration of BFS
- However, there exists another natural way to find nodes reachable from s
- We start from s and try the first edge leading out of s , to a node v
- We then follow the first edge leading out of v , and continue in this way until we reach a “dead-end”, a node for which we already explored all neighbors
- We then backtrack until we meet a node with a unexplored neighbor, etc
- This algorithm is called *depth-first search* (DFS)

Problems

1. Find topological ordering for the following two graphs:

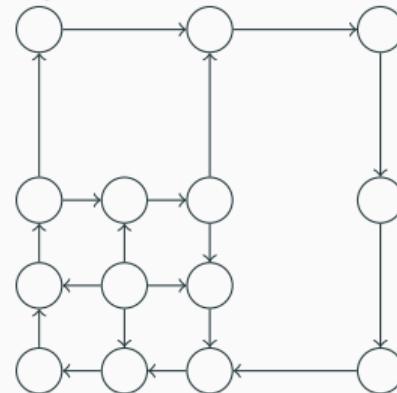


2. For the following directed graph use *breadth-first* search algorithm to find all strongly connected components.



Problems

3. Find all strongly connected components in the following directed graph:



4. An induced subgraph of a graph $G = (V, E)$ is a graph $H = (U, F)$ such that $U \subseteq V$, and $F = E \cap (U \times U)$. Given an undirected graph $G = (V, E)$ and an integer k , find the maximum induced subgraph H of G such that each vertex in H has degree at least k , or determine that it does not exist. The algorithm should run in time $O(|E| + |V|)$.

Problems

5. Let $G = (V, E)$ be a directed graph (not necessarily acyclic). Design an efficient algorithm to label the vertices of the graph with distinct labels from 1 to $|V|$ such that the label of each vertex v is greater than the label of at least one of v 's predecessors (if v has any), or to determine that no such labeling is possible (w is a *predecessor* of v iff $(w, v) \in E$). Your algorithm should run in time $O(|E| + |V|)$.
6. Let $G = (V, E)$ be a directed graph with n vertices. A *sink* is a vertex $s \in V$ such that for all $v \in V$, $(v, s) \notin E$. Devise an algorithm that, given the adjacency matrix of G , determines whether or not G has a sink in time $O(n)$.
7. Design an algorithm that for a given DAG $G = (V, E)$ checks/recognizes if G is semiconnected in time $O(|V| + |E|)$.
A directed graph $G = (V, E)$ is called *semiconnected* if and only if for every two vertices $u, v \in V$ there is a directed path from u to v or from v to u .

Problems

8. A graph is *triconnected* if there is no pair of vertices whose removal disconnects the graph. Design an algorithm that determines whether a graph with n vertices and e edges is triconnected in time $O(ne)$.
9. Let $G = (V, E)$ be a directed graph. A vertex v is called a *master-vertex* of G if there is a directed path from v to every other vertex in G . Design a linear-time algorithm to find the number of master-vertices in a given directed graph G .
[Note that this number can be equal to 0.]