Design and Analysis of Algorithms

Presented by Dr. Li Ning

Shenzhen Institutes of Advanced Technology, Chinese Academy of Science
Shenzhen, China

Algorithms on Directed Graphs

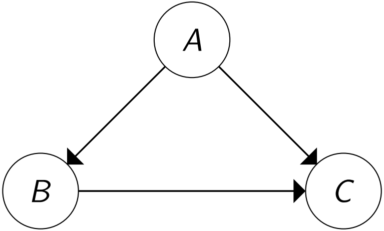
Directed Graph

Directed Graph

Directed Graph: a set of **nodes** connected by the **directed edges**.

$G = (V, E)$

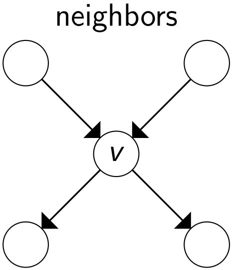
- V: the set of nodes
  - A, B, and C
- E: the set of edges
  - (A, B), (B, C), and (A, C)



Neighbors

Given a graph $G = (V, E)$, for node $v \in V$

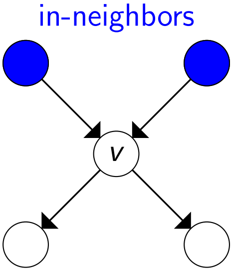
- in-neighbors: $u \in V$, s.t. $(u, v) \in E$
- out-neighbors: $u \in V$, s.t. $(v, u) \in E$



Neighbors

Given a graph $G = (V, E)$, for node $v \in V$

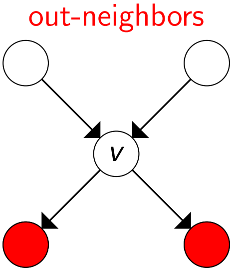
- in-neighbors: $u \in V$, s.t. $(u, v) \in E$
- out-neighbors: $u \in V$, s.t. $(v, u) \in E$



Neighbors

Given a graph $G = (V, E)$, for node $v \in V$

- in-neighbors: $u \in V$, s.t. $(u, v) \in E$
- out-neighbors: $u \in V$, s.t. $(v, u) \in E$



Traversal on Directed Graphs

DFS: Visit node v

- For every **out-neighbor** u of v
  - If u is not visited
    - DFS on u

BFS: pop node v from the queue

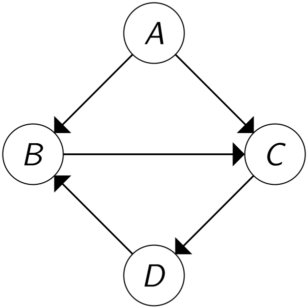
- For every **out-neighbor** u of v
  - If u is not visited
    - Visit u
    - add u to the queue

Connectivity

Connected: two nodes u and v are connected, iff.

- there is a path from u to v;
- there is a path from v to u.

Strongly Connected: all pairs of nodes are connected, i.e. for any pair of nodes u and v, there is a directed path from u to v.

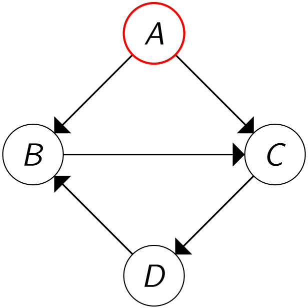


Connectivity

Connected: two nodes u and v are connected, iff.

- there is a path from u to v;
- there is a path from v to u.

Strongly Connected: all pairs of nodes are connected, i.e. for any pair of nodes u and v, there is a directed path from u to v.



Connectivity

"**Connected**" is an equivalence relation.

- Reflexive: v is connected to v, for all $v \in V$.
- Symmetric: v is connected to $u \Rightarrow u$ is connected to v
- Transitive: if
    - v is connected to u
    - u is connected to w

  then v is connected to w

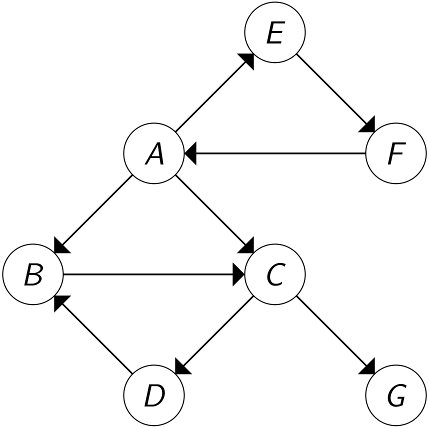
Connectivity

With an equivalence relation, the set can be divided into disjoint parts: in each part, the elements are related.

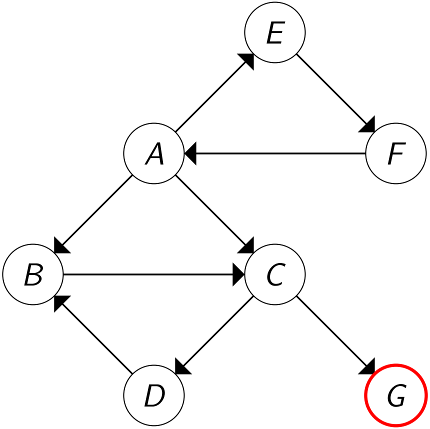
Strongly connected components: V is divided into several disjoint parts $V_0, V_1, \ldots$, such that

- all nodes in V_i are connected
- for any pair of nodes $u \in V_i$ and $v \in V_j$ with $i \neq j$, u is not connected to v.

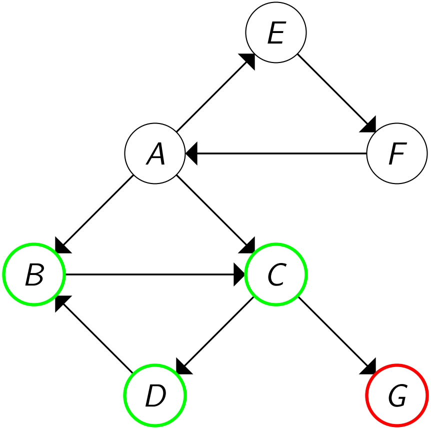
Connectivity



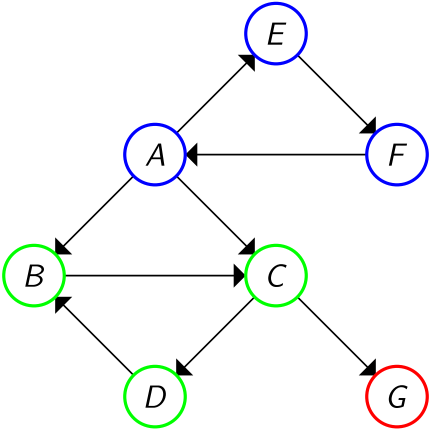
Connectivity



Connectivity



Connectivity



Strongly Connected Component

Strongly Connected Component

problem: given a directed graph $g = (v, e)$, find the strongly connected components, i.e. dividing v into $v_0, v_1, \ldots$, such that

- all nodes in v_i are connected
- for any pair of nodes $u \in v_i$ and $v \in v_j$ with $i \neq j$, u is not connected to v.

Component Graph

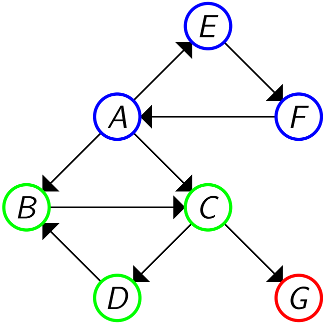
Component Graph: given a directed graph $G = (V, E)$, the components graph is defined as $G^* = (V^*, E^*)$ where

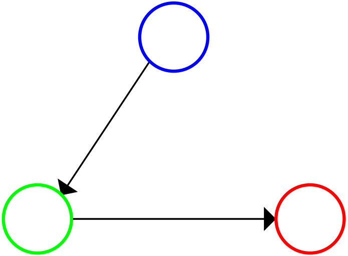
- V^*: one node for each strongly connected component
- E^*: there is an edge from node V_i to node V_j iff. there exists $u \in V_i$ and $v \in V_j$ with $(u, v) \in E$

Component Graph

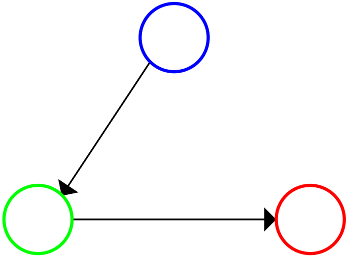
Component Graph: given a directed graph $G = (V, E)$, the components graph is defined as $G^* = (V^*, E^*)$ where

- V^*: one node for each strongly connected component
- E^*: there is an edge from node V_i to node V_j iff. there exists $u \in V_i$ and $v \in V_j$ with $(u, v) \in E$



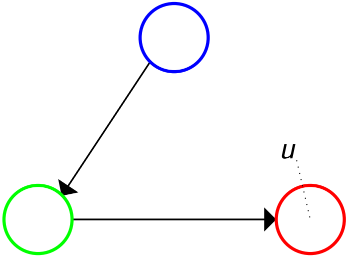


Component Graph



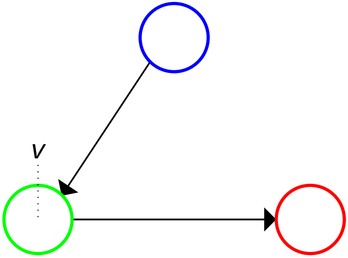
- The component graph has no cycles.

Component Graph



- The component graph has no cycles.
- For a node u in the sink component, all nodes v connected from u are all in the sink component.

Component Graph



- The component graph has no cycles.
- For a node u in the sink component, all nodes v connected from u are all in the sink component.
- Then we consider the component previous to the sink component.

Component Graph

Sink: the component that has no outgoing edge.

Algorithm: Components(G)

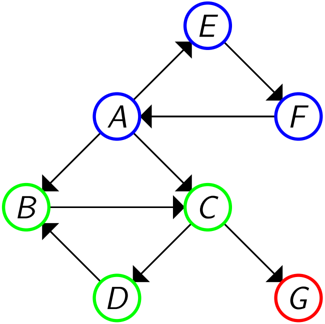
$S = [\,]$;
while $\cup_{C \in S} C \neq V$ **do**
    let s be any node in the sink component;
    run BFS from s;
    let C be the set of visited nodes;
    append C to S;
    remove nodes and edges adjacent to C from G;
end
Return S;

Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

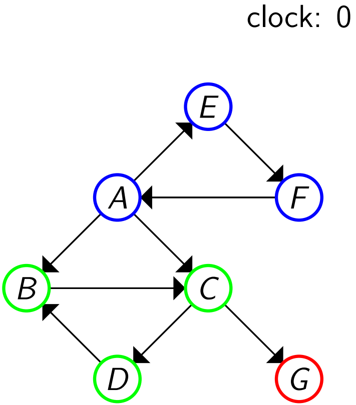


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

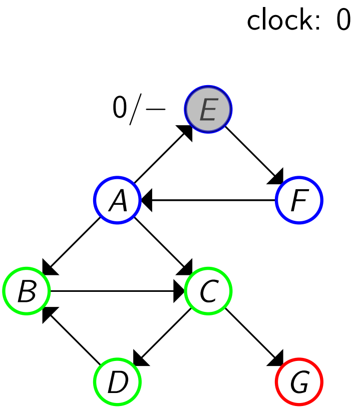


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$



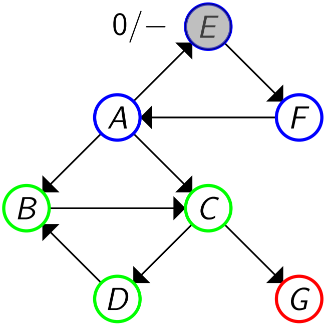
Find the Sink

clock $= 0$

clock: 1

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

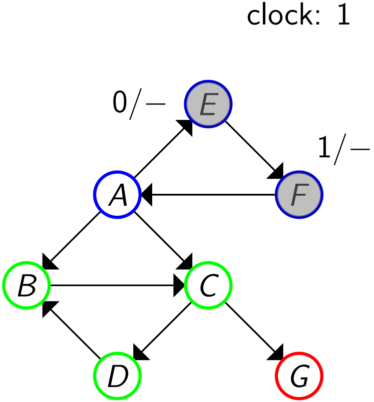


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

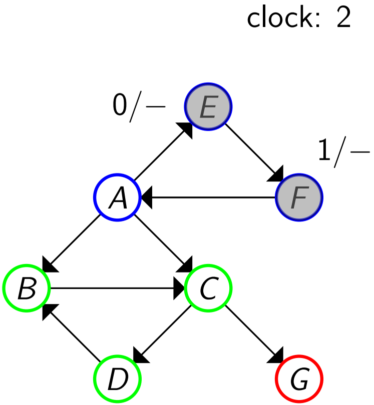


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

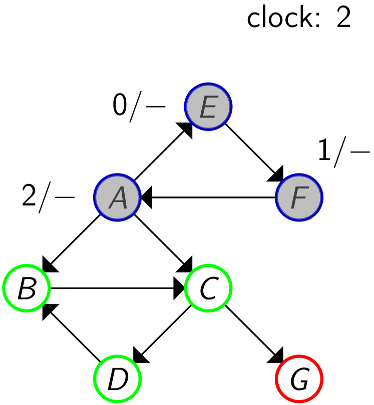


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

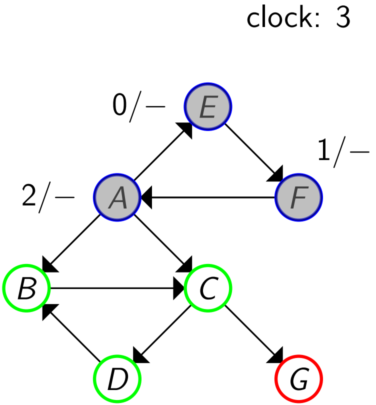


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

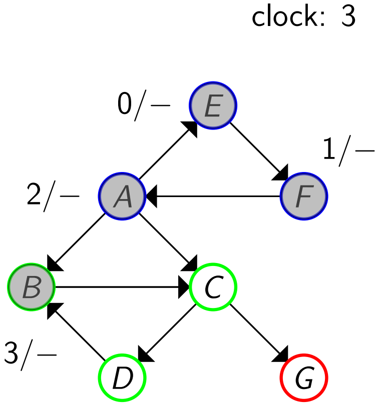


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

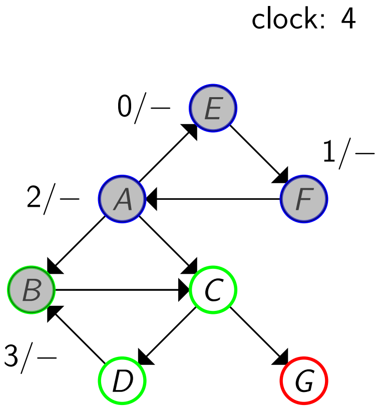


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$



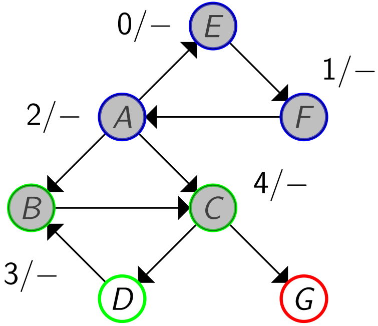
Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

clock: 4

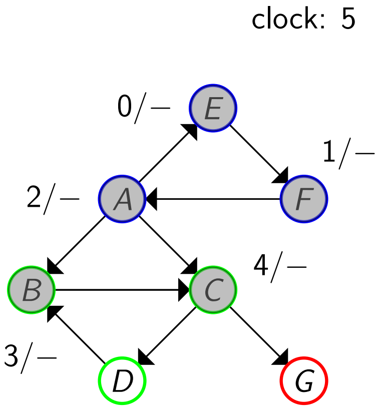


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

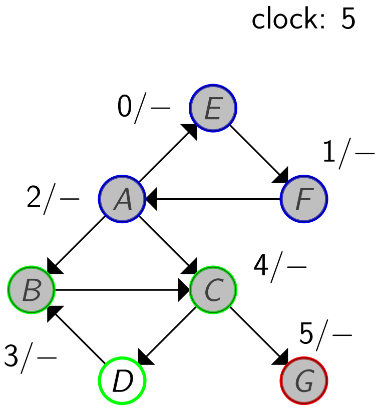


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

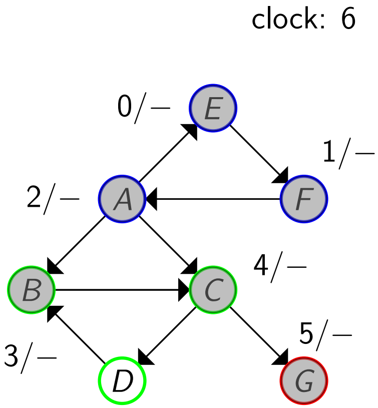


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
    - $pre(v) =$ **clock**
    - **clock** $+ = 1$
    - For every out-neighbor u of v
        - If u is not visited: Modified DFS on u
    - $post(v) =$ **clock**
    - **clock** $+ = 1$

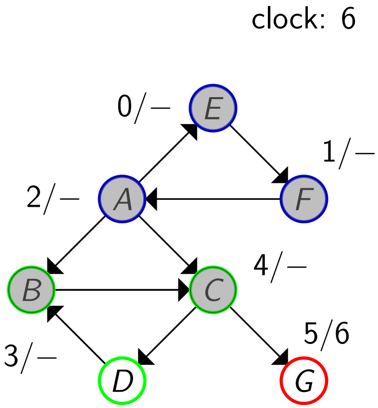


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

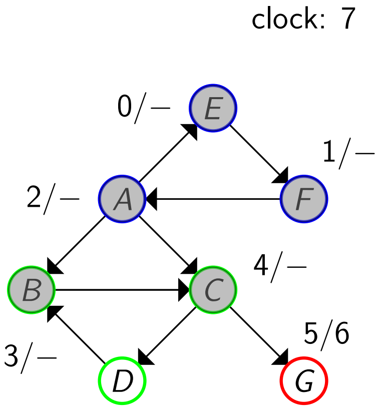


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

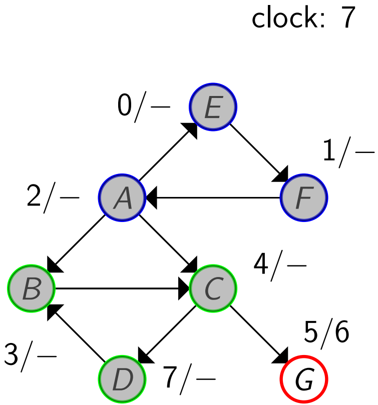


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

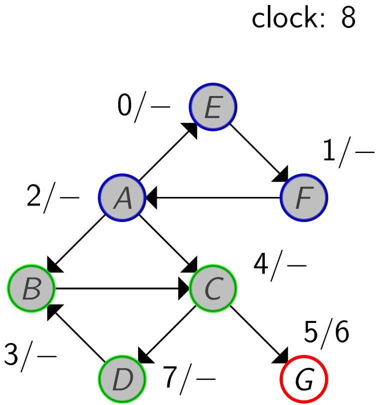


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+ = 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+ = 1$



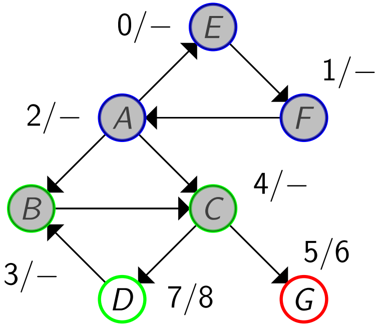
Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

clock: 8



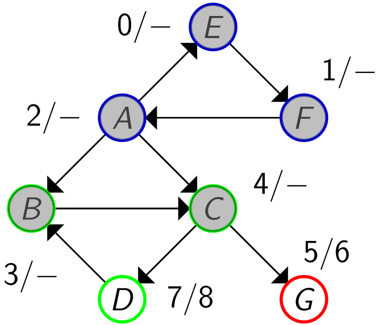
Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

clock: 9



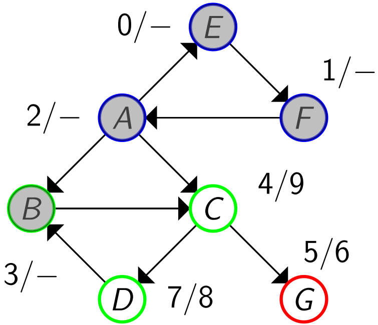
Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

clock: 9

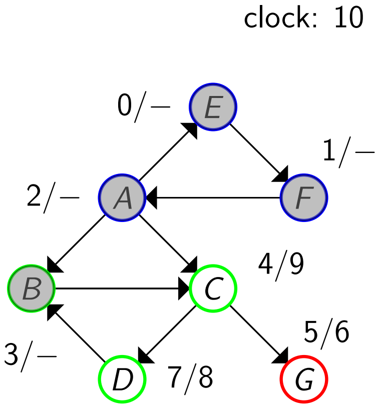


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

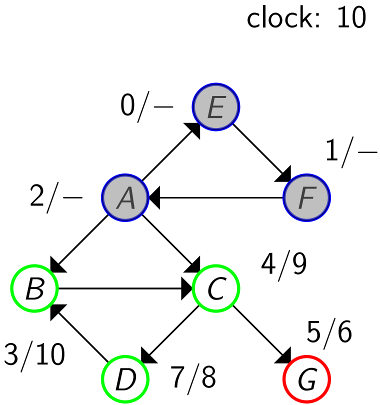


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

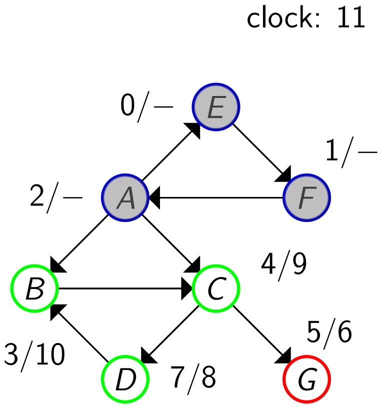


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

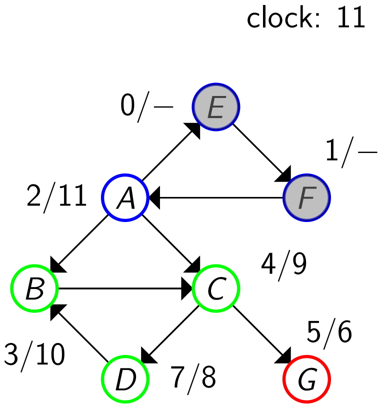


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+ = 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+ = 1$



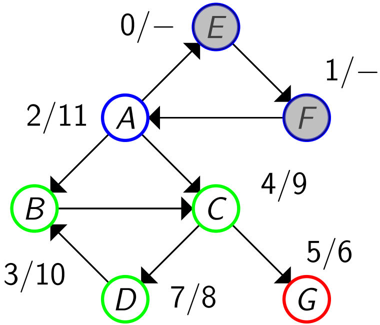
Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

clock: 12

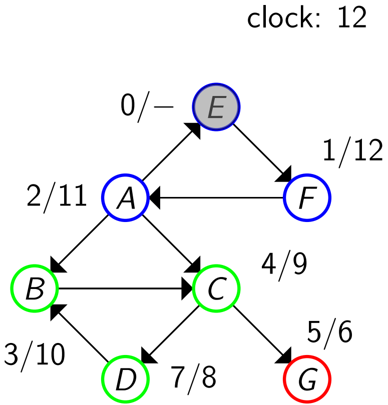


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+= 1$

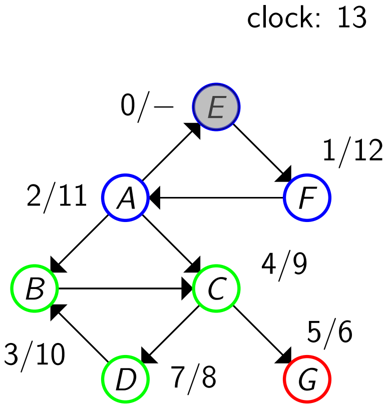


Find the Sink

clock $= 0$

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+ = 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+ = 1$

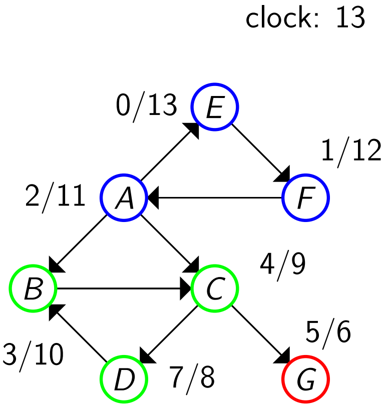


Find the Sink

clock $= 0$

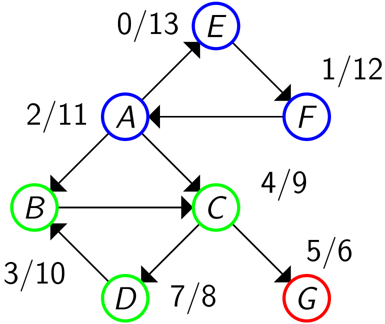
Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+ = 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(v) =$ **clock**
  - **clock** $+ = 1$



Post Clock

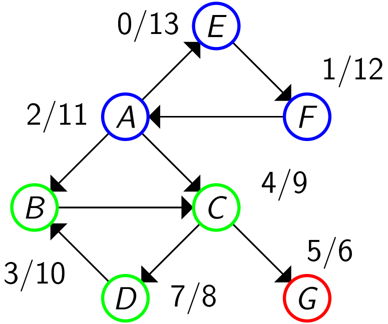
Largest *post*: node in a source component



Post Clock

Largest *post*: node in a source component

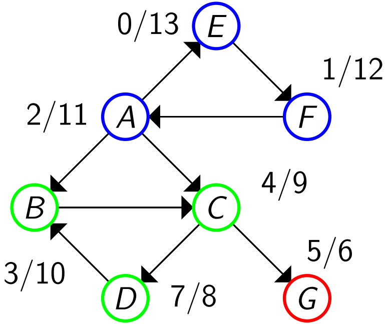
- pre(*source*)



Post Clock

Largest *post*: node in a source component

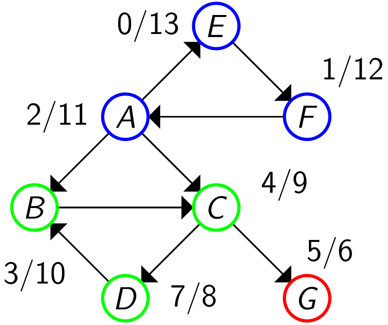
- pre(*source*)
- visit other components



Post Clock

Largest *post*: node in a source component

- pre(*source*)
- visit other components
- post(*source*)



Find Strongly Connected Components

Largest *post*: node in a source component

- pre(*source*)
- visit other components
- post(*source*)

Find Strongly Connected Components

Largest *post*: node in a source component

- pre(*source*)
- visit other components
- post(*source*)

1. G^R : reverse all edges of G

Find Strongly Connected Components

Largest *post*: node in a source component

- pre(*source*)
- visit other components
- post(*source*)

1. G^R : reverse all edges of G
2. **Modified DFS** on G^R

Find Strongly Connected Components

Largest *post*: node in a source component

- pre(*source*)
- visit other components
- post(*source*)

1. G^R : reverse all edges of G
2. **Modified DFS** on G^R
3. v : node of largest *post*

Find Strongly Connected Components

Largest *post*: node in a source component

- pre(*source*)
- visit other components
- post(*source*)

1. G^R : reverse all edges of G
2. **Modified DFS** on G^R
3. v : node of largest *post*
4. v in the sink component of G

Find Strongly Connected Components

Largest *post*: node in a source component

- pre(*source*)
- visit other components
- post(*source*)

1. G^R : reverse all edges of G
2. **Modified DFS** on G^R
3. v : node of largest *post*
4. v in the sink component of G
5. remove all nodes connected from v

Find Strongly Connected Components

Largest *post*: node in a source component

- pre(*source*)
- visit other components
- post(*source*)

1. G^R : reverse all edges of G
2. **Modified DFS** on G^R
3. v : node of largest *post*
4. v in the sink component of G
5. remove all nodes connected from v
6. next? repeat $1-5$ on the remaining graph

Find Strongly Connected Components

Largest *post*: node in a source component

- pre(*source*)
- visit other components
- post(*source*)

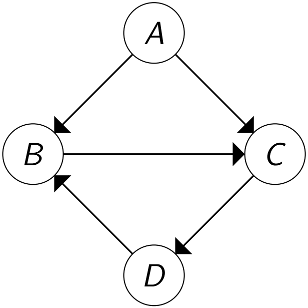
1. G^R : reverse all edges of G
2. **Modified DFS** on G^R
3. v : node of largest *post*
4. v in the sink component of G
5. remove all nodes connected from v
6. next? repeat $1-5$ on the remaining graph
7. **or**, largest *post* among the remaining nodes

Directed Cycle

Directed Cycle

Directed Cycle: a closed path consisting of directed edges.

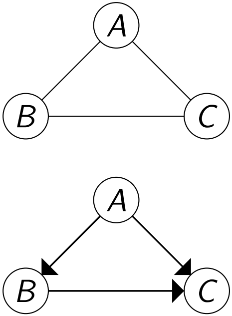
- $B - C - D - B$
  - (B, C) is in E
  - (C, D) is in E
  - (D, B) is in E



Directed Cycle

Directed Cycle: a closed path consisting of directed edges.

- Undirected: A - B - C - A
- Directed: A - B - C - A
    - (A, B) is in E
    - (B, C) is in E
    - (C, A) is not in E

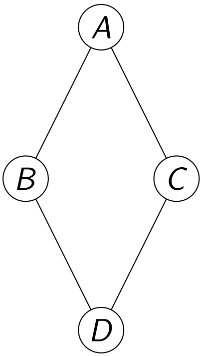


Cycle in Undirected Graphs

Problem: check if the given undirected graph contains a cycle.

DFS

- explore the children after visiting a node
- **cycle** $\Leftrightarrow$ visited child

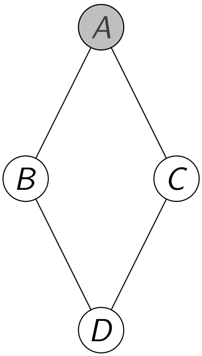


Cycle in Undirected Graphs

Problem: check if the given undirected graph contains a cycle.

DFS

- explore the children after visiting a node
- **cycle** $\Leftrightarrow$ visited child

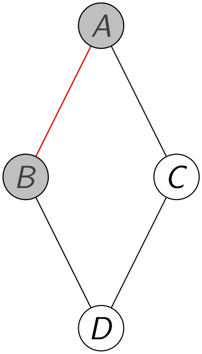


Cycle in Undirected Graphs

Problem: check if the given undirected graph contains a cycle.

DFS

- explore the children after visiting a node
- **cycle** $\Leftrightarrow$ visited child

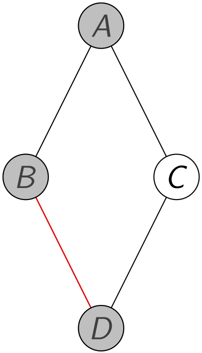


Cycle in Undirected Graphs

Problem: check if the given undirected graph contains a cycle.

DFS

- explore the children after visiting a node
- **cycle** $\Leftrightarrow$ visited child

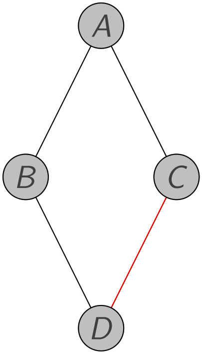


Cycle in Undirected Graphs

Problem: check if the given undirected graph contains a cycle.

DFS

- explore the children after visiting a node
- **cycle** $\Leftrightarrow$ visited child

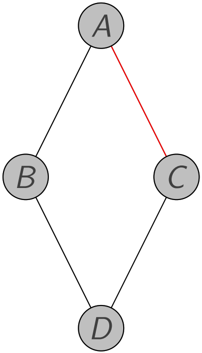


Cycle in Undirected Graphs

Problem: check if the given undirected graph contains a cycle.

DFS

- explore the children after visiting a node
- **cycle** $\Leftrightarrow$ visited child

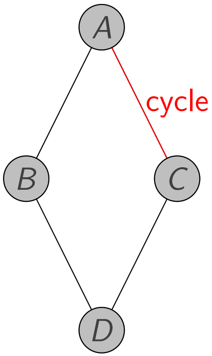


Cycle in Undirected Graphs

Problem: check if the given undirected graph contains a cycle.

DFS

- explore the children after visiting a node
- **cycle** $\Leftrightarrow$ visited child

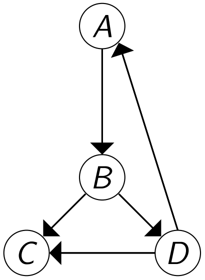


Cycle in Directed Graphs

Problem: check if the given directed graph contains a cycle.

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(r) =$ **clock**
  - **clock** $+= 1$

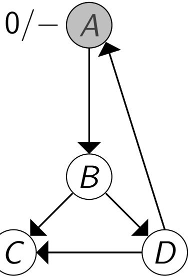


Cycle in Directed Graphs

Problem: check if the given directed graph contains a cycle.

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(r) =$ **clock**
  - **clock** $+= 1$

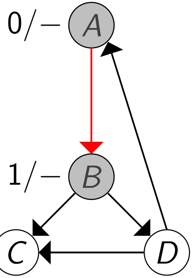


Cycle in Directed Graphs

Problem: check if the given directed graph contains a cycle.

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(r) =$ **clock**
  - **clock** $+= 1$

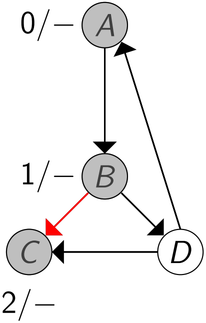


Cycle in Directed Graphs

Problem: check if the given directed graph contains a cycle.

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(r) =$ **clock**
  - **clock** $+= 1$

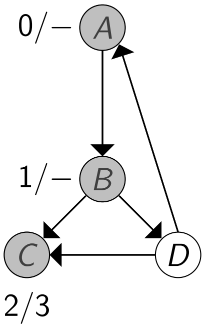


Cycle in Directed Graphs

Problem: check if the given directed graph contains a cycle.

Modified DFS

- Visit node v
    - $pre(v) =$ **clock**
    - **clock** $+= 1$
    - For every out-neighbor u of v
        - If u is not visited: Modified DFS on u
    - $post(r) =$ **clock**
    - **clock** $+= 1$

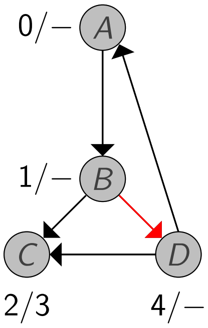


Cycle in Directed Graphs

Problem: check if the given directed graph contains a cycle.

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited:
      Modified DFS on u
  - $post(r) =$ **clock**
  - **clock** $+= 1$

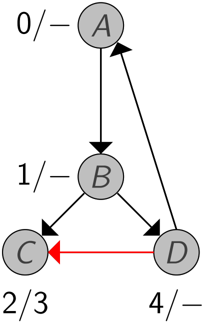


Cycle in Directed Graphs

Problem: check if the given directed graph contains a cycle.

Modified DFS

- Visit node v
  - $pre(v) =$ **clock**
  - **clock** $+= 1$
  - For every out-neighbor u of v
    - If u is not visited: Modified DFS on u
  - $post(r) =$ **clock**
  - **clock** $+= 1$



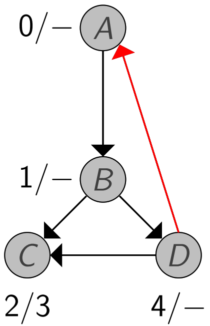
cross edge; not a cycle

Cycle in Directed Graphs

Problem: check if the given directed graph contains a cycle.

Modified DFS

- Visit node v
    - $pre(v) =$ **clock**
    - **clock** $+= 1$
    - For every out-neighbor u of v
        - If u is not visited: Modified DFS on u
    - $post(r) =$ **clock**
    - **clock** $+= 1$



back edge; found a cycle

DAG: Directed Acyclic Graph

Conclusion: A directed graph has a cycle if and only if the DFS reveals a **back edge**.

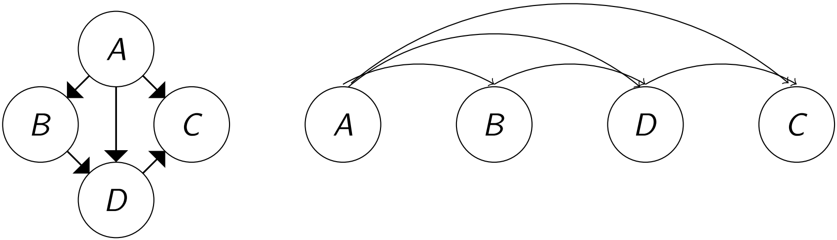
Directed Acyclic Graph: a directed graph contains no cycles.

Topological Sort

Directed Acyclic Graph

Problem: given a directed acyclic graph (DAG) $G = (V, E)$, list the nodes in a sequence, such that for any edge $(u, v) \in E$, node u precedes node v, i.e.

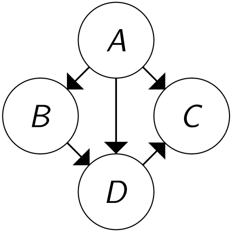
$$u \prec v, \forall (u, v) \in E$$

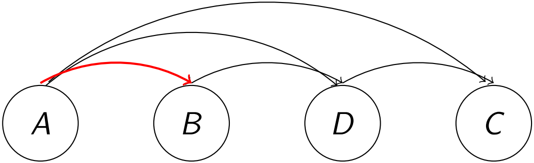


Directed Acyclic Graph

Problem: given a directed acyclic graph (DAG) $G = (V, E)$, list the nodes in a sequence, such that for any edge $(u, v) \in E$, node u precedes node v, i.e.

$$u \prec v, \forall (u, v) \in E$$

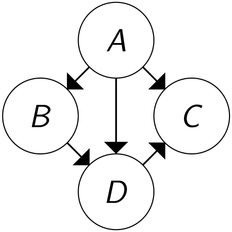


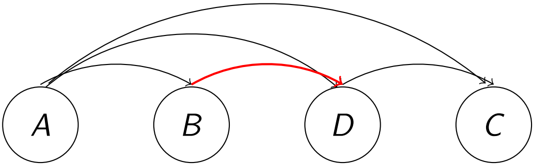


Directed Acyclic Graph

Problem: given a directed acyclic graph (DAG) $G = (V, E)$, list the nodes in a sequence, such that for any edge $(u, v) \in E$, node u precedes node v, i.e.

$$u \prec v, \forall (u, v) \in E$$

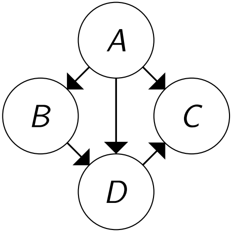


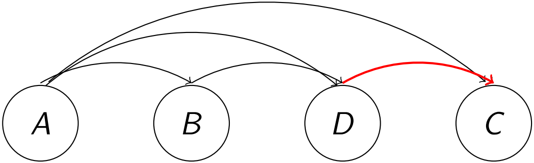


Directed Acyclic Graph

Problem: given a directed acyclic graph (DAG) $G = (V, E)$, list the nodes in a sequence, such that for any edge $(u, v) \in E$, node u precedes node v, i.e.

$$u \prec v, \forall (u, v) \in E$$

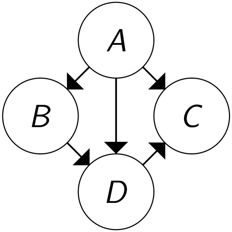


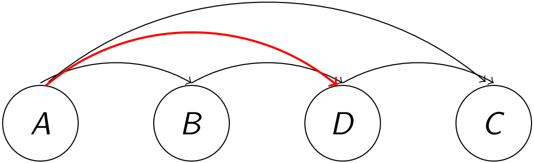


Directed Acyclic Graph

Problem: given a directed acyclic graph (DAG) $G = (V, E)$, list the nodes in a sequence, such that for any edge $(u, v) \in E$, node u precedes node v, i.e.

$$u \prec v, \forall (u, v) \in E$$

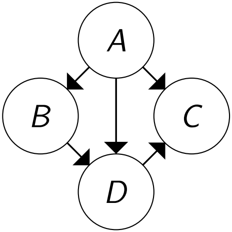


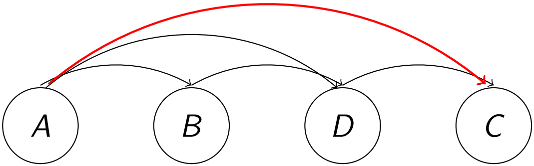


Directed Acyclic Graph

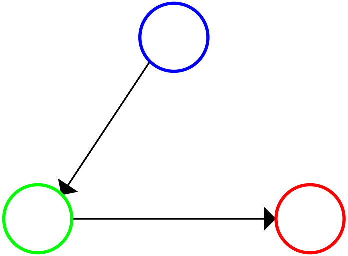
Problem: given a directed acyclic graph (DAG) $G = (V, E)$, list the nodes in a sequence, such that for any edge $(u, v) \in E$, node u precedes node v, i.e.

$$u \prec v, \forall (u, v) \in E$$

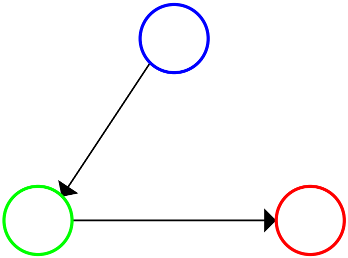




Strongly Connected Components

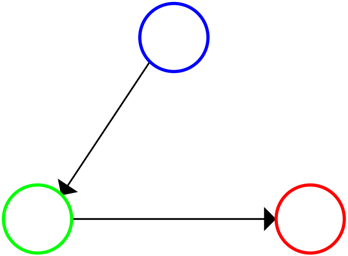


Strongly Connected Components



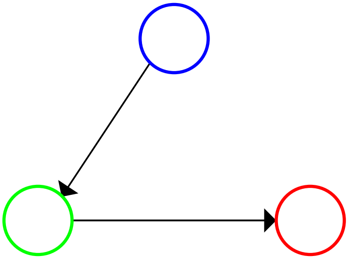
- Find the node of the largest post time.

Strongly Connected Components



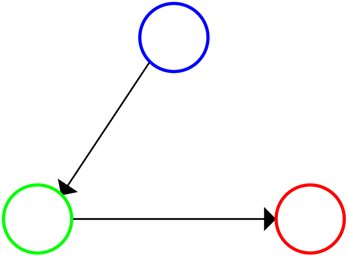
- Find the node of the largest post time.
- Remove adjacent nodes and edges?

Strongly Connected Components



- Find the node of the largest post time.
- Remove adjacent nodes and edges?
  - Not necessary!

Strongly Connected Components

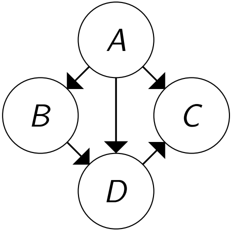


- Find the node of the largest post time.
- Remove adjacent nodes and edges?
    - Not necessary!
- Find the node of the largest post time **in the remaining ones**.

Topological Sort

DFS: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - DFS on v



$S = [\,]$, the sorted the sequence

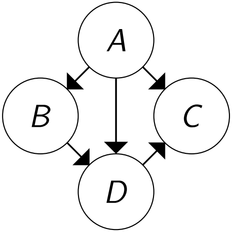
TopologicalSort: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - TopologicalSort on v
- put r at the front of S

Topological Sort

DFS: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - DFS on v



$S = [\ ]$

$S = [\ ]$, the sorted the sequence

TopologicalSort: Start at node r

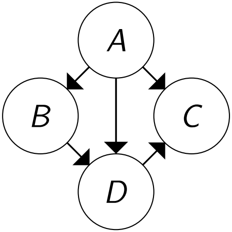
- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - TopologicalSort on v
- put r at the front of S

Visit A

Topological Sort

DFS: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - DFS on v



$S = [\ ]$

$S = [\ ]$, the sorted the sequence

TopologicalSort: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - TopologicalSort on v
- put r at the front of S

Visit A

- Visit B

Topological Sort

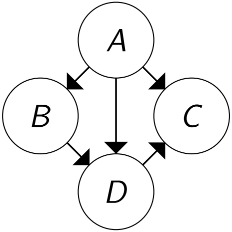
DFS: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - DFS on v

$S = [\,]$, the sorted the sequence

TopologicalSort: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - TopologicalSort on v
- put r at the front of S



$S = [\,]$

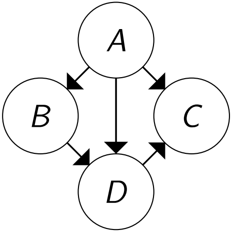
Visit A

- Visit B
  - Visit D

Topological Sort

DFS: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - DFS on v



$S = [\,]$

$S = [\,]$, the sorted the sequence

TopologicalSort: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - TopologicalSort on v
- put r at the front of S

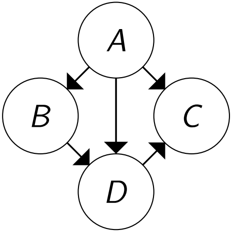
Visit A

- Visit B
  - Visit D
    - Visit C

Topological Sort

DFS: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - DFS on v



$S = [C]$

$S = [\,]$, the sorted the sequence

TopologicalSort: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - TopologicalSort on v
- put r at the front of S

Visit A

- Visit B
  - Visit D

Topological Sort

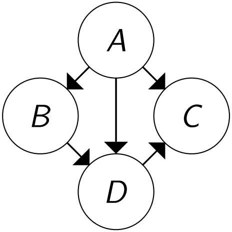
DFS: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - DFS on v

$S = [\ ]$, the sorted the sequence

TopologicalSort: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - TopologicalSort on v
- put r at the front of S



$S = [D, C]$

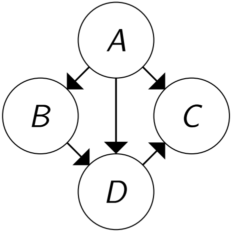
Visit A

- Visit B

Topological Sort

DFS: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - DFS on v



$S = [B, D, C]$

$S = [\ ]$, the sorted the sequence

TopologicalSort: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - TopologicalSort on v
- put r at the front of S

Visit A

Topological Sort

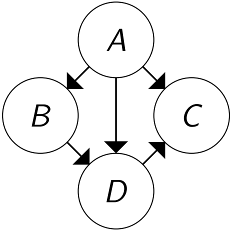
DFS: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - DFS on v

$S = [\ ]$, the sorted the sequence

TopologicalSort: Start at node r

- Visit node r
- For every **out-neighbor** v of r
  - If v is not visited
    - TopologicalSort on v
- put r at the front of S



$S = [A, B, D, C]$

THANK YOU

中国科学院深圳先进技术研究院
SHENZHEN INSTITUTES OF ADVANCED TECHNOLOGY
CHINESE ACADEMY OF SCIENCES