CS 361
Data Structures & Algs
Lecture 15

Prof. Tom Hayes
University of New Mexico
10-12-2010

Last Time

Identifying BFS vs. DFS trees

- Can they be the same?

Problems 3.6, 3.9, 3.2

- details left as homework. email.

DFS: recursive vs iterative. 2 kinds of stack

Digraphs, directed paths, oriented cycles, DAGs, topological ordering, DFS in digraph

Today

strongly and weakly connected digraphs

new equivalence relation: “strongly connected to”. Strongly connected components. Structure Theorem.

DFS for digraphs. Applications.

Connected Digraphs

An undirected graph is connected when?

Connected Digraphs

Undirected graph G=(V,E). “Connected”?

For every u, v in V, there exists a path from u to v.

Connected Digraphs

Undirected graph G=(V,E). "Connected"?

For every u, v in V, exists path from u to v.

Directed graph G=(V,E). What should "connected" mean?

Connected Digraphs

Undirected graph G=(V,E). “Connected”?

For every u, v in V, exists path from u to v.

Directed graph G=(V,E). What should “connected” mean?

2 versions: “Strongly connected”

“Weakly connected”

Connected Digraphs

Undirected graph G=(V,E). "Connected"?

For every u, v in V, exists path from u to v.

Directed graph G=(V,E). "Strongly connected" means for every u, v in V, there exists an oriented path from u to v, and an oriented path from v to u.

"Weakly connected" means, ignoring edge directions, the undirected graph is connected.

Components

Undirected G=(V,E). "Component of v"?

Components

Undirected G=(V,E). “Component of v”?
All vertices that have a path to/from v.
Recall: “a has a path to b” is an **equivalence relation** on V.

Components

Undirected G=(V,E). “Component of v”? All vertices that have a path to/from v. Recall: “a has a path to b” is an **equivalence relation** on V.

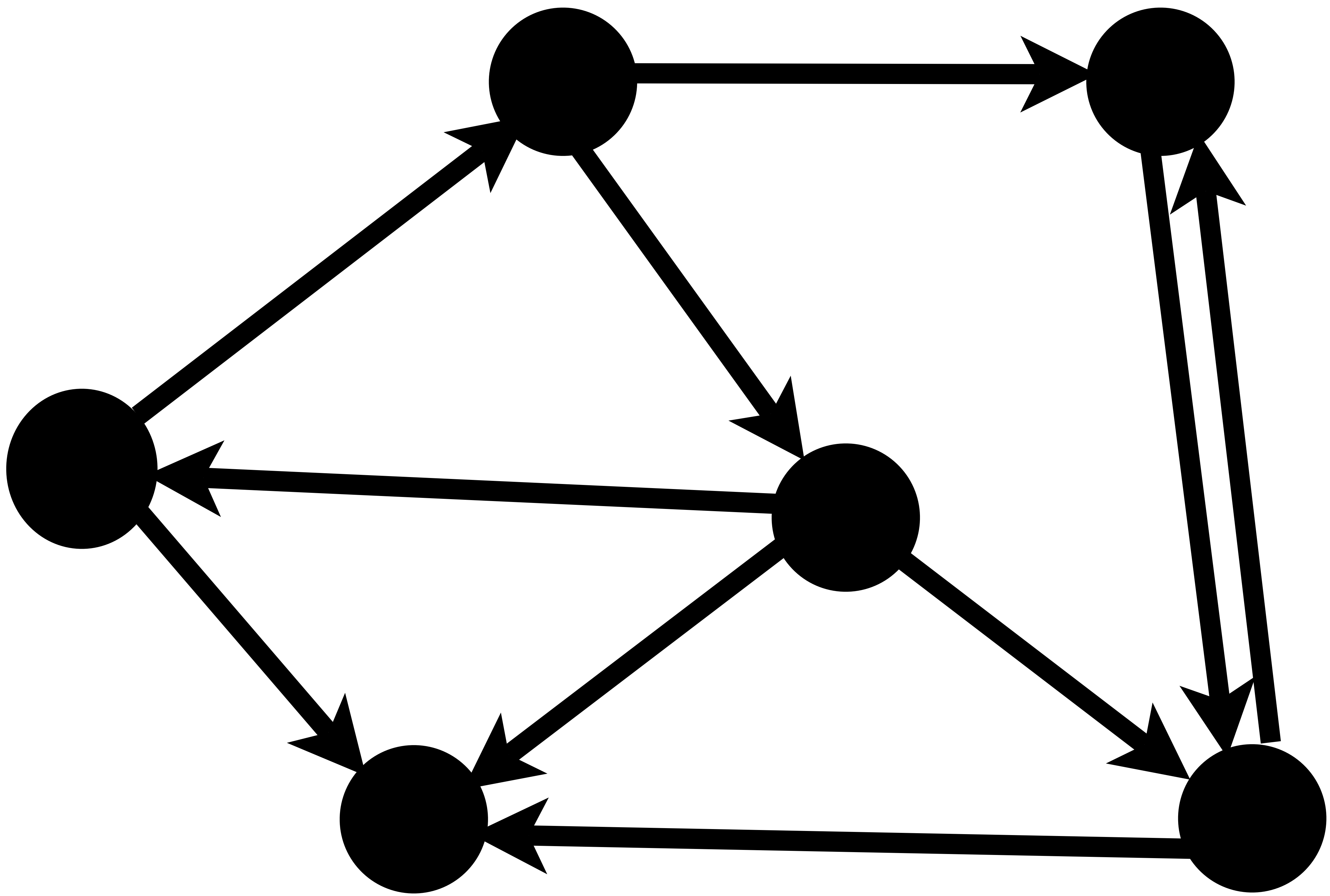
Directed G = (V,E). “Strong component of v”?

Components

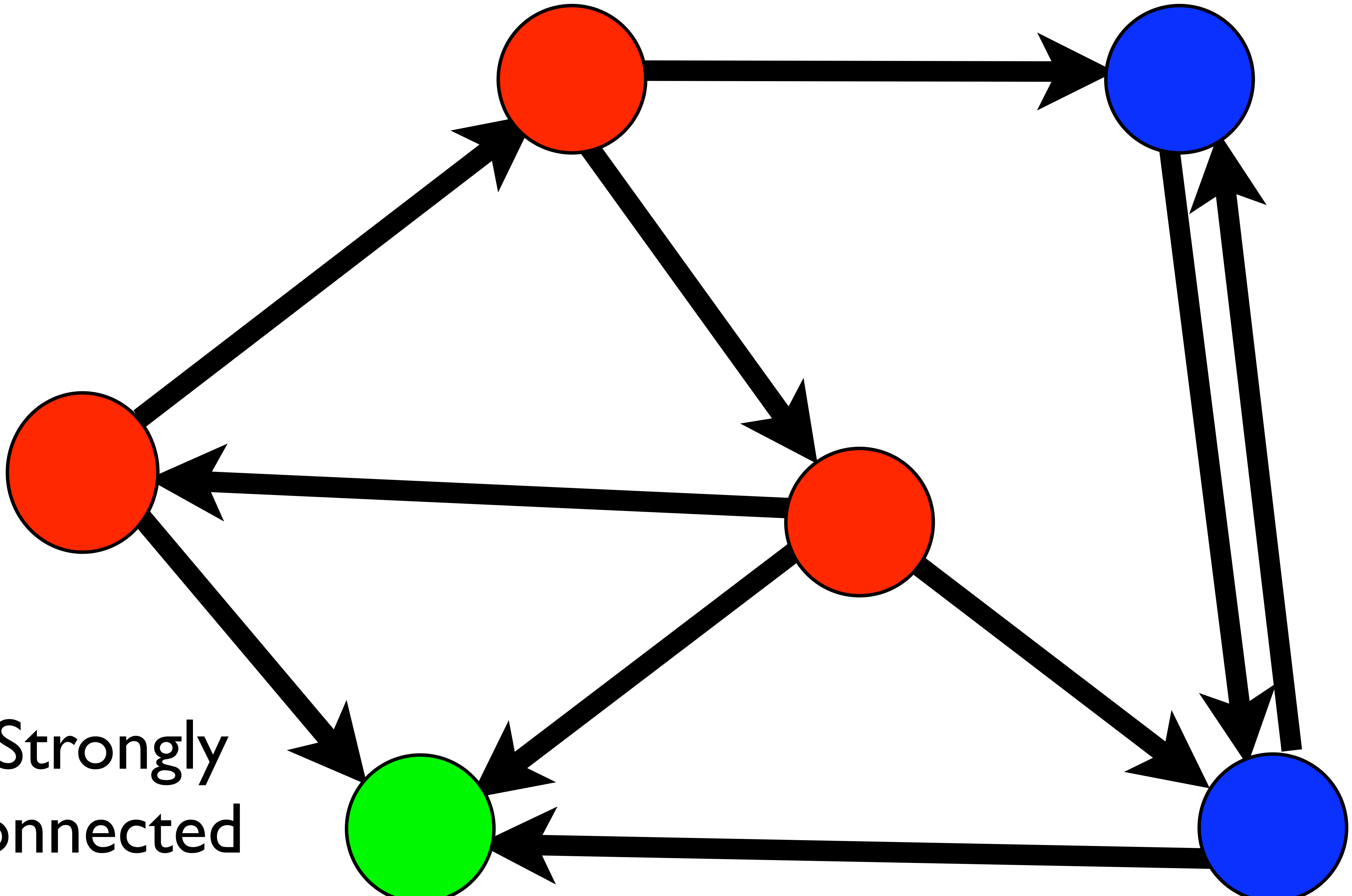
Undirected G=(V,E). "Component of v"? All vertices that have a path to/from v. Recall: "a has a path to b" is an **equivalence relation** on V.

Directed G = (V,E). "Strong component of v"? All vertices w such that w has both an oriented path to v, and from v. "a is in the same strong component as b" is an equivalence relation too.

Example



Example



3 Strongly
Connected
Components

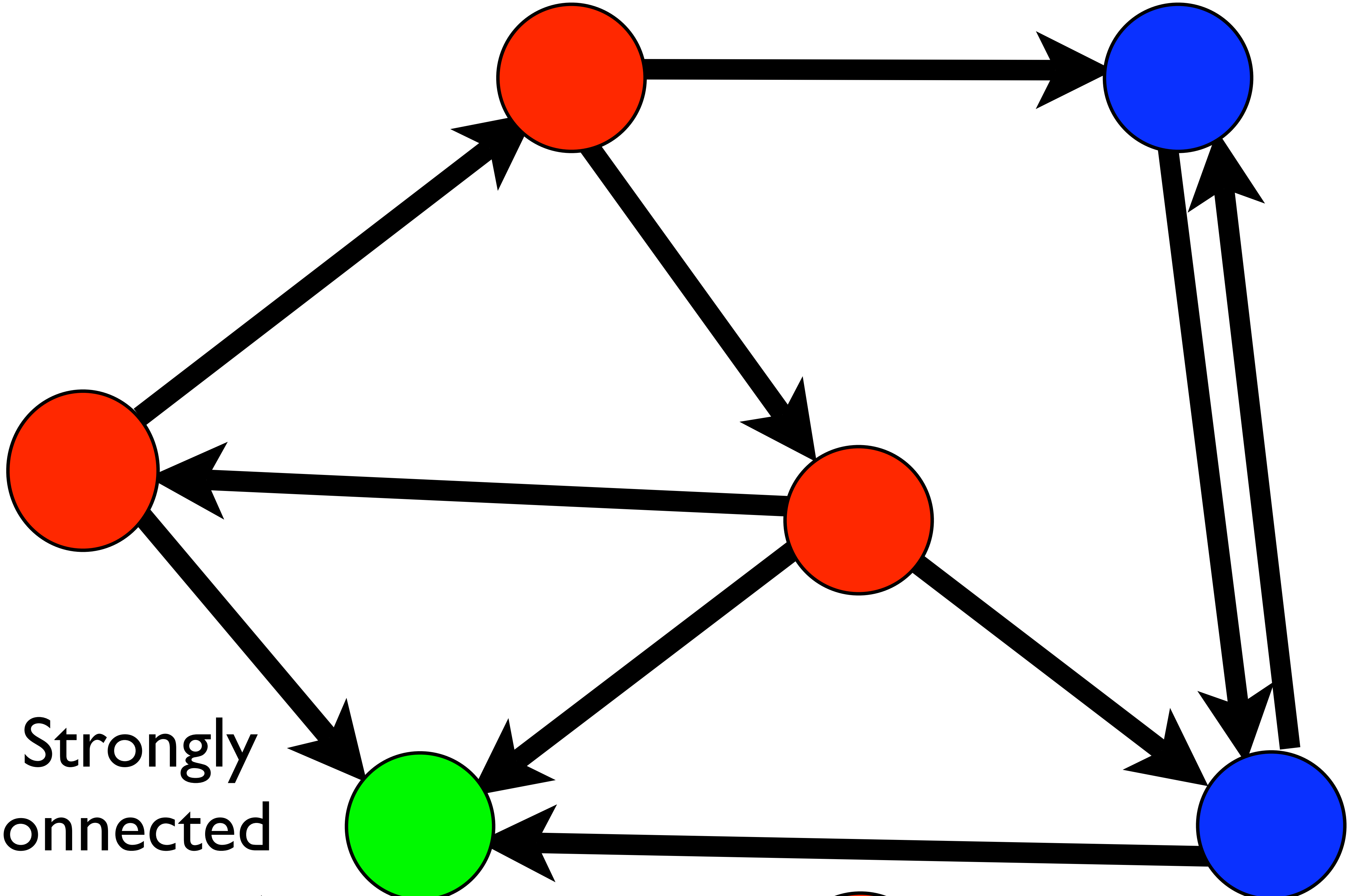
Structure Theorem

The Strong Components graph of G is obtained by “contracting” each strong component of G to a single vertex. Self-loops and multiple edges may result, and are discarded.

1: Strong Components graph is acyclic.

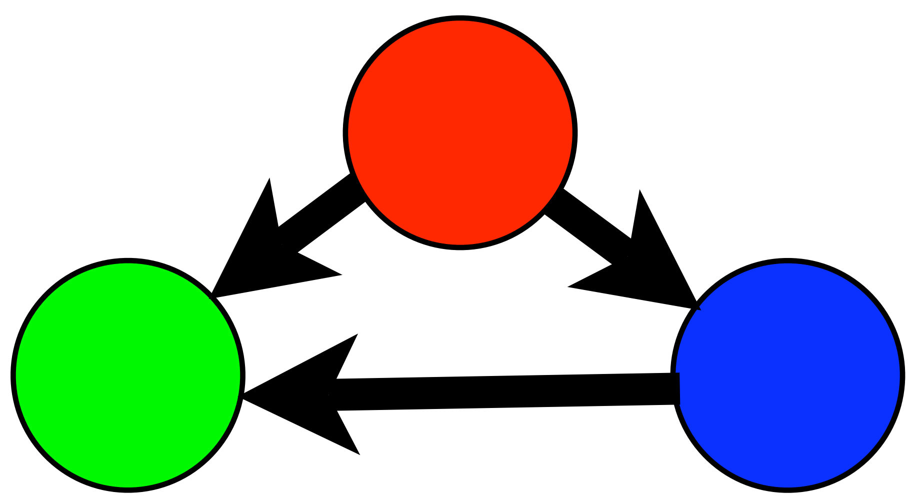
2: G has an oriented path from u to v iff scg(G) has an oriented path from [u] to [v]. (paths of length 0 count).

3: G is acyclic iff G = scg(G).



3 Strongly Connected Components

contracted version:



One Technicality

In an undirected graph, the shortest a cycle can be is length 3. Why? No edge or node may be repeated.

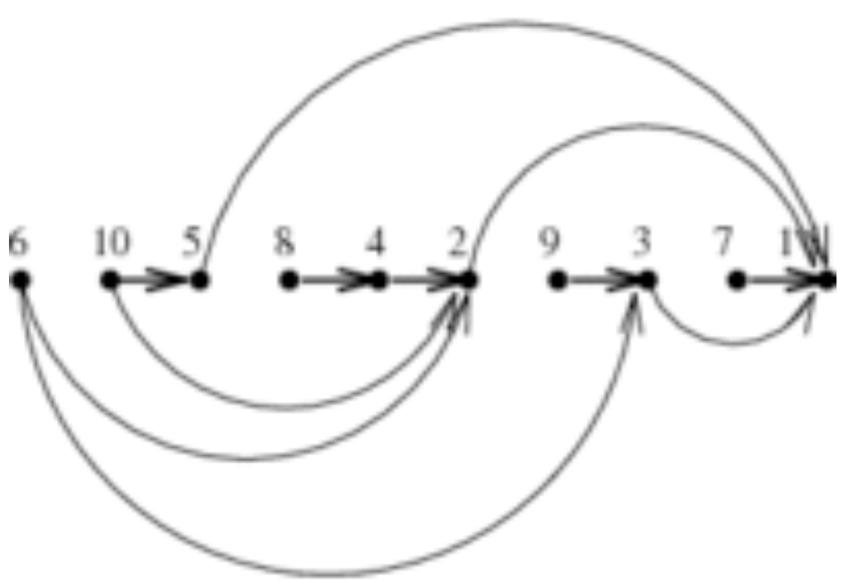
In a directed graph, there can be cycles of length 2, or even 1. Why? Edges in opposite directions don't count as repeats

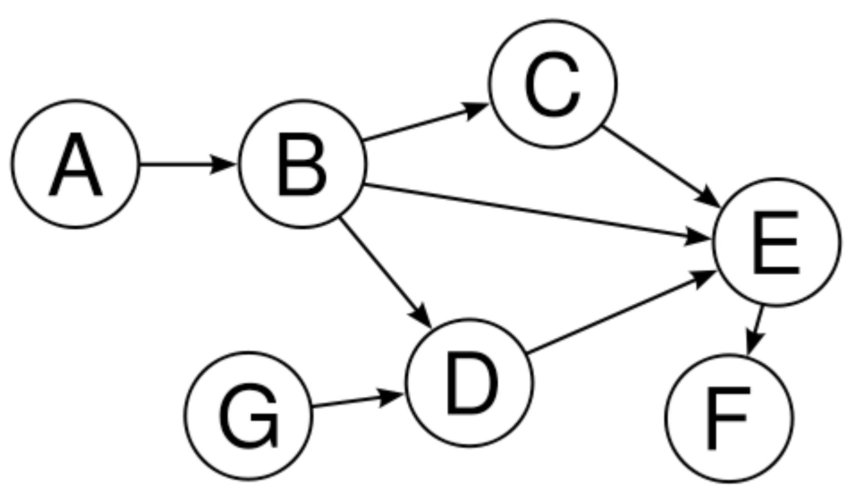
DAGs

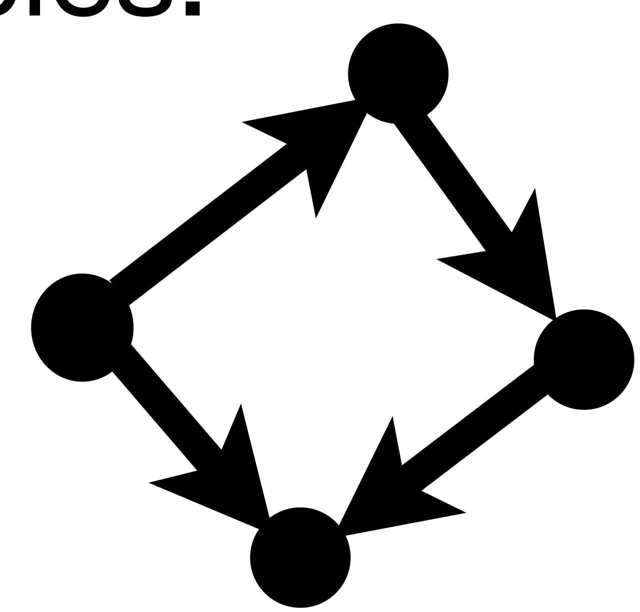
A directed graph is called acyclic if it has no oriented cycles.

Meaning: you can't get back where you start if you always follow arrows.

The "underlying graph" (just lose all the orientation info) may have cycles.



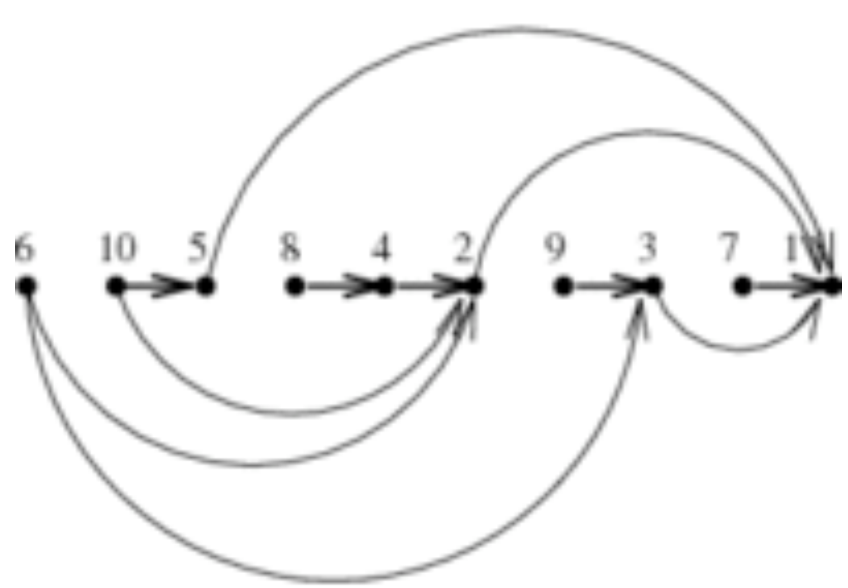


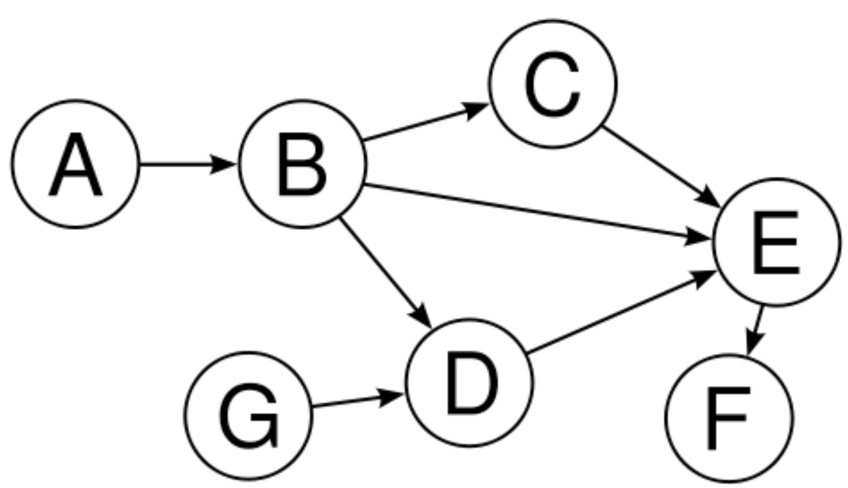


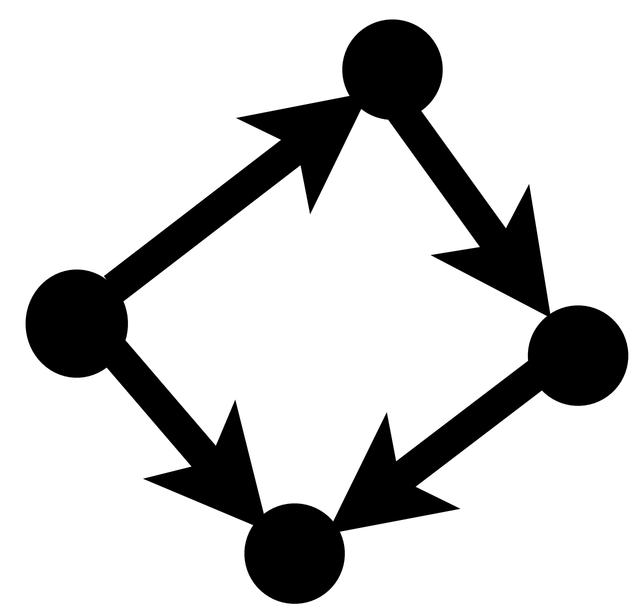
DAGs

Q: How do we tell if a graph is a DAG? Alternatively, how do we find an oriented cycle if there is one?

Look at the Left example. The nodes are in a line, all the edges go left-to-right. This is called a topological sort.





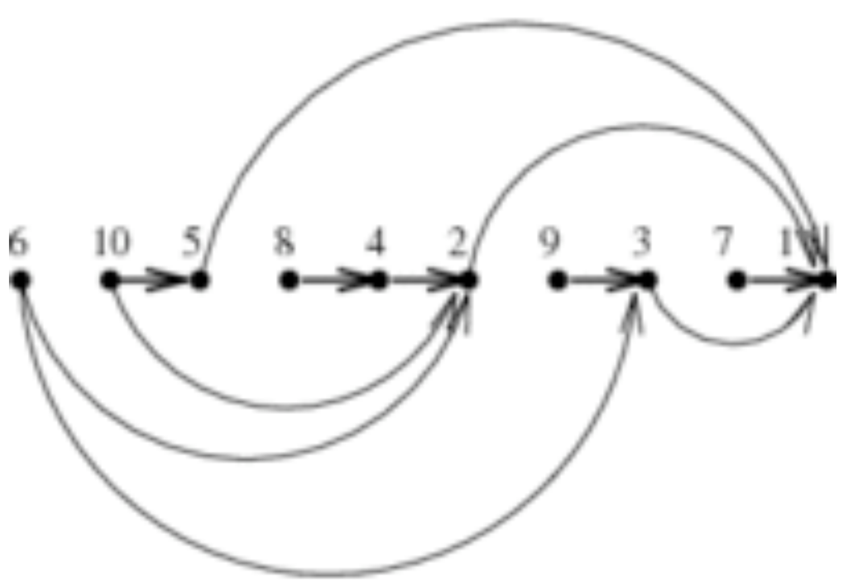


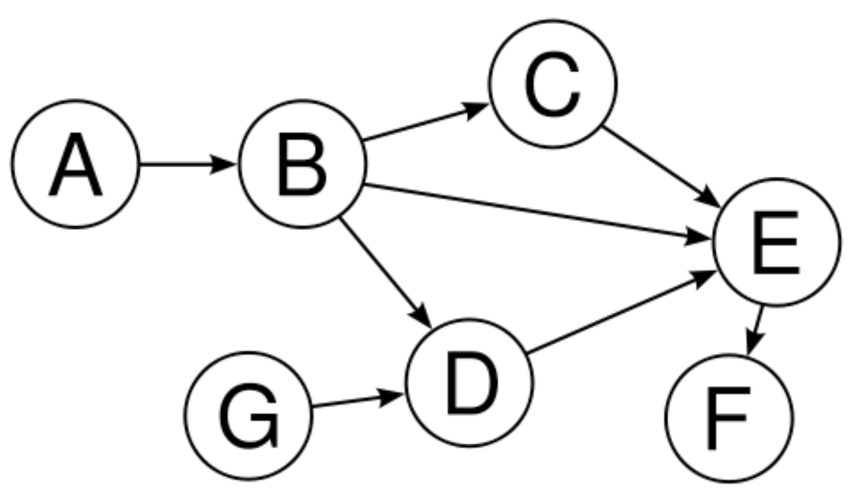
DAGs

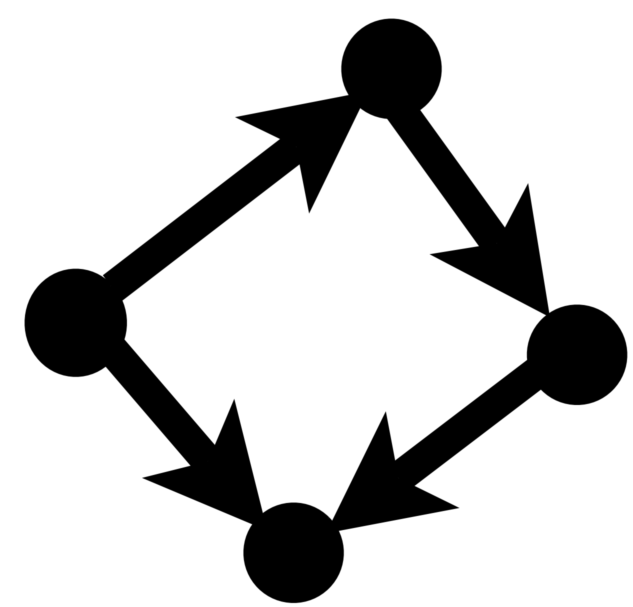
Q: How do we tell if a graph is a DAG? Alternatively, how do we find an oriented cycle if there is one?

Look at the Left example. The nodes are in a line, all the edges go left-to-right. This is called a topological sort.

Thm: G has a topological sort iff G is a DAG



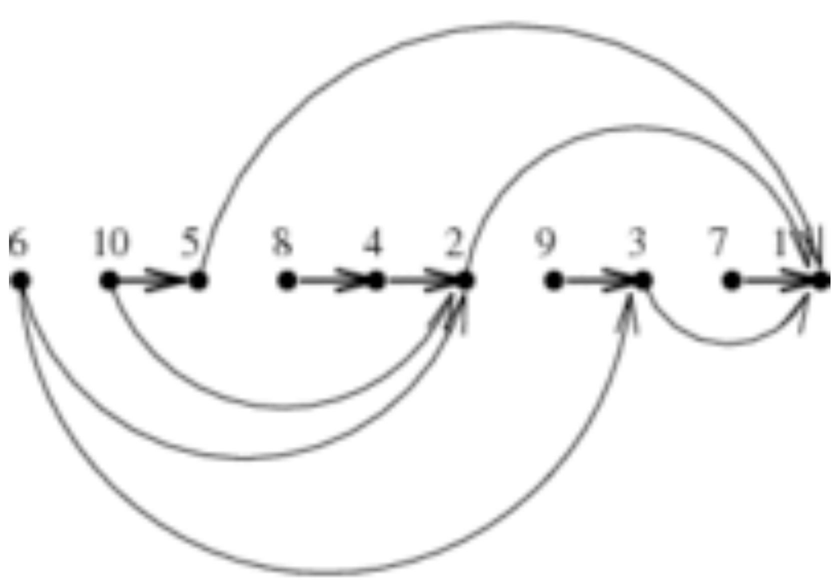


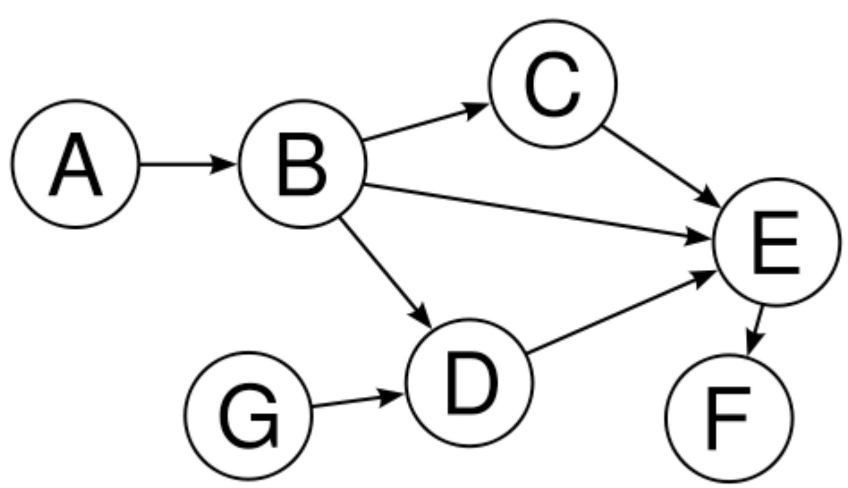


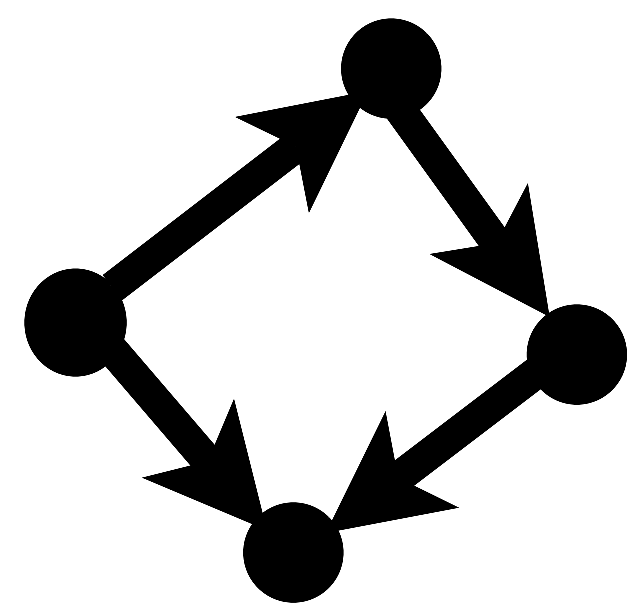
Topological Sorting

Q: How do we tell if a graph has a topo. sort?

Look at the left example. The leftmost node has no in-edges. Is this always the case?





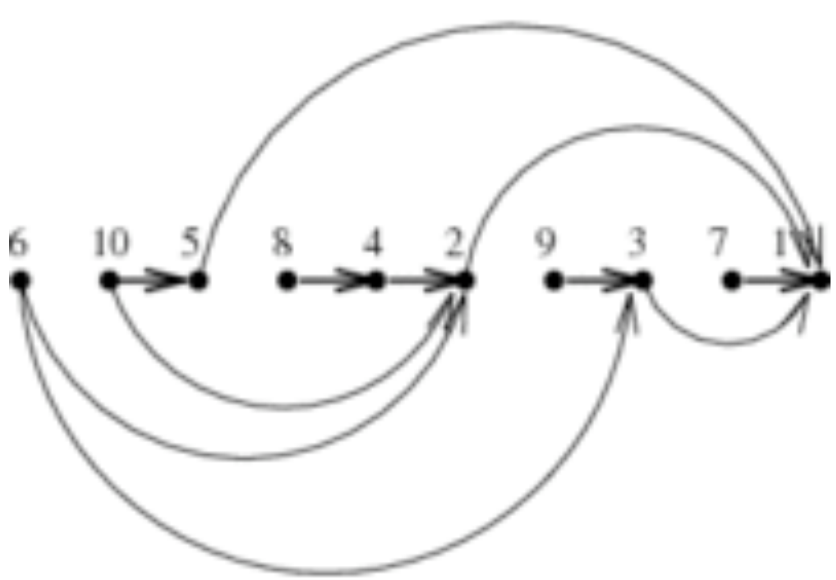


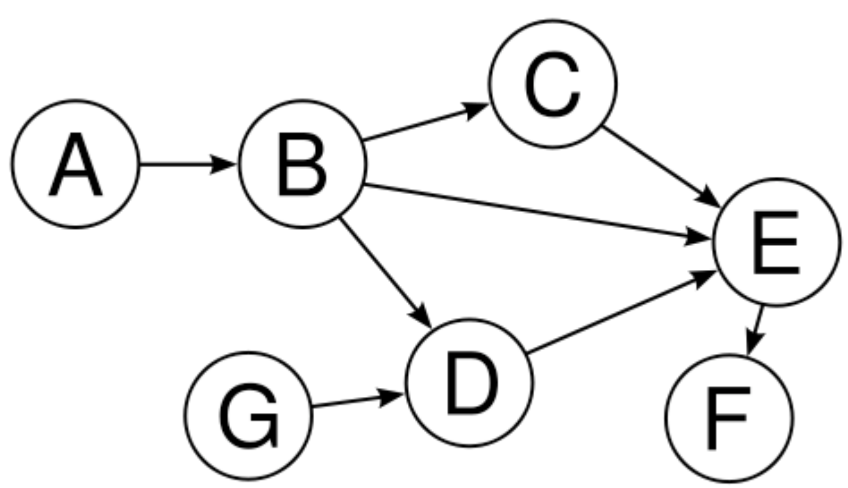
Topological Sorting

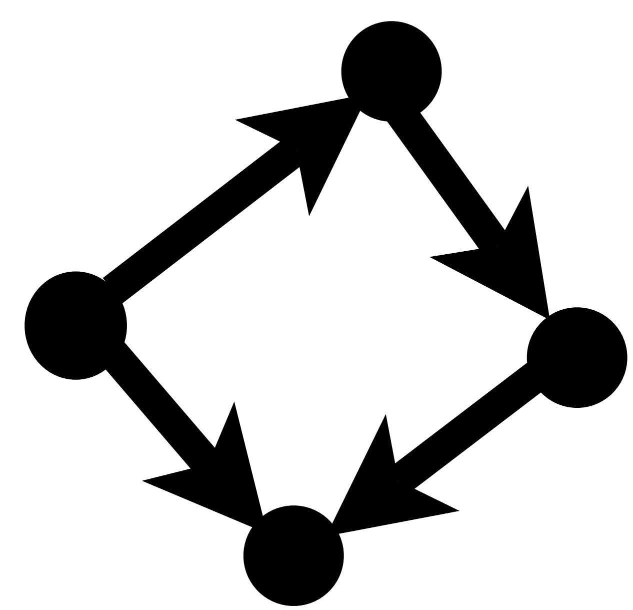
Q: How do we tell if a graph has a topo. sort?

Look at the left example. The leftmost node has no in-edges. Is this always the case? Yes.

Idea: Find the leftmost node. Recurse!







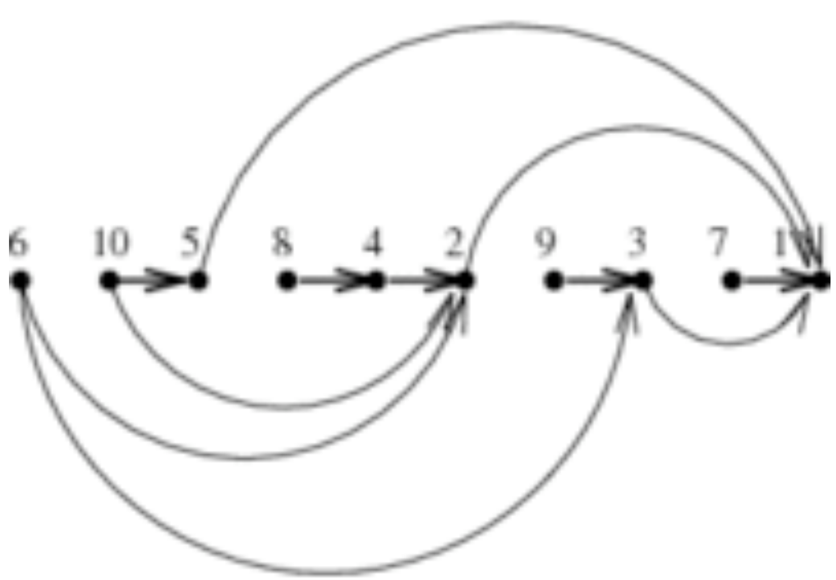
Topological Sorting

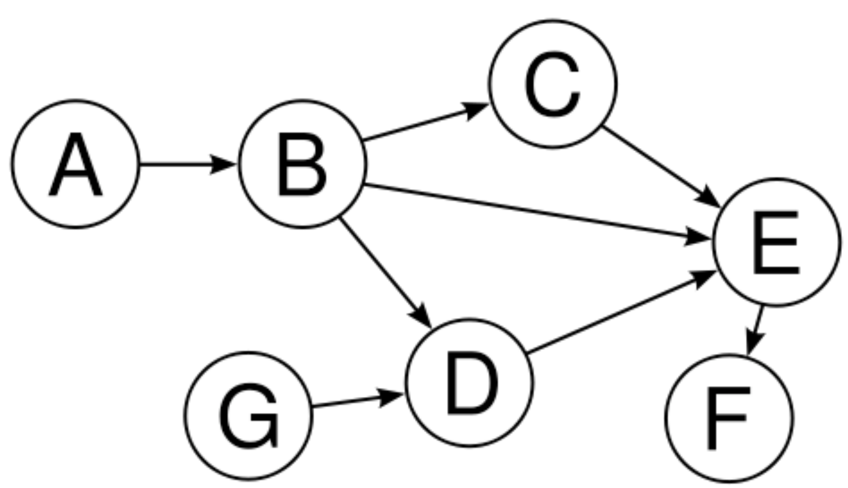
Q: How do we tell if a graph has a topo. sort?

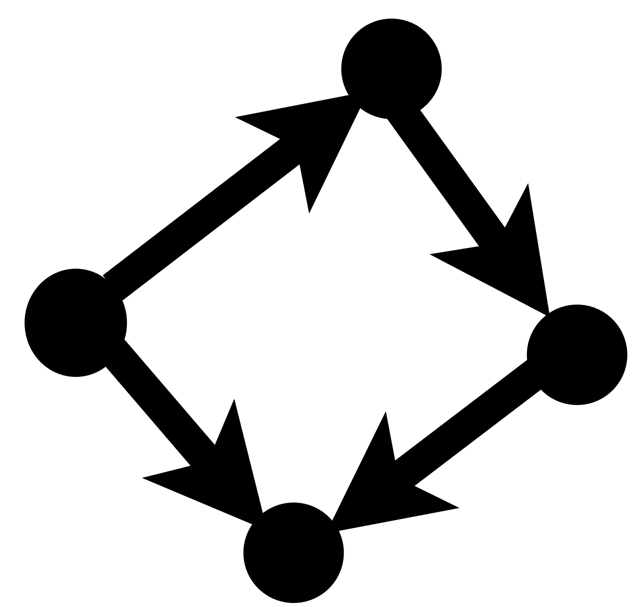
Look at the left example. The leftmost node has no in-edges. Is this always the case? Yes.

Idea: Find the leftmost node. Recurse!

Implementation issues?

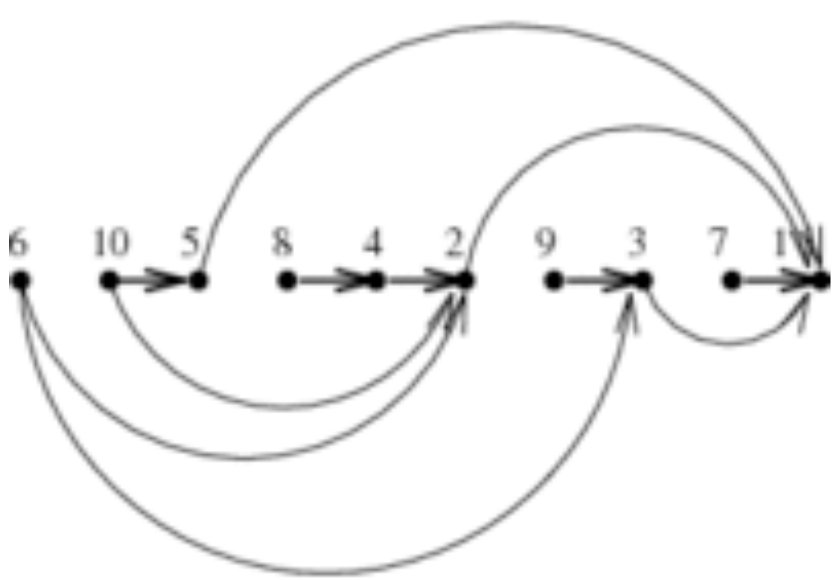


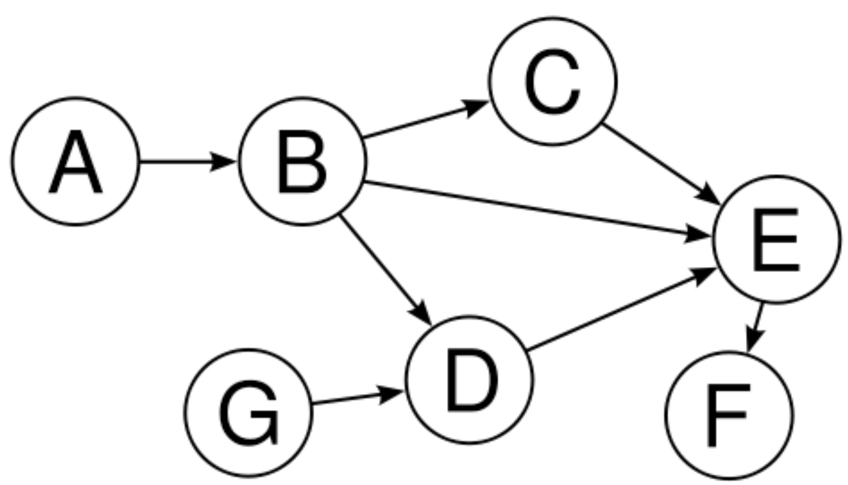


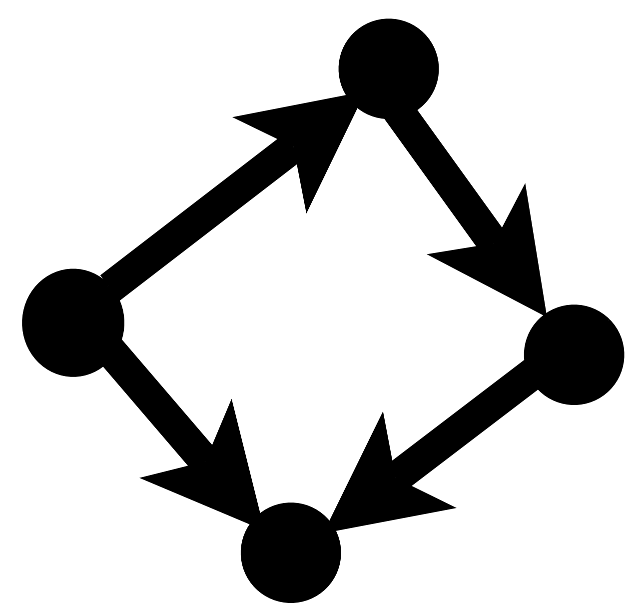


Searching DiGraphs

Question: Find all nodes reachable from s in a directed graph G.



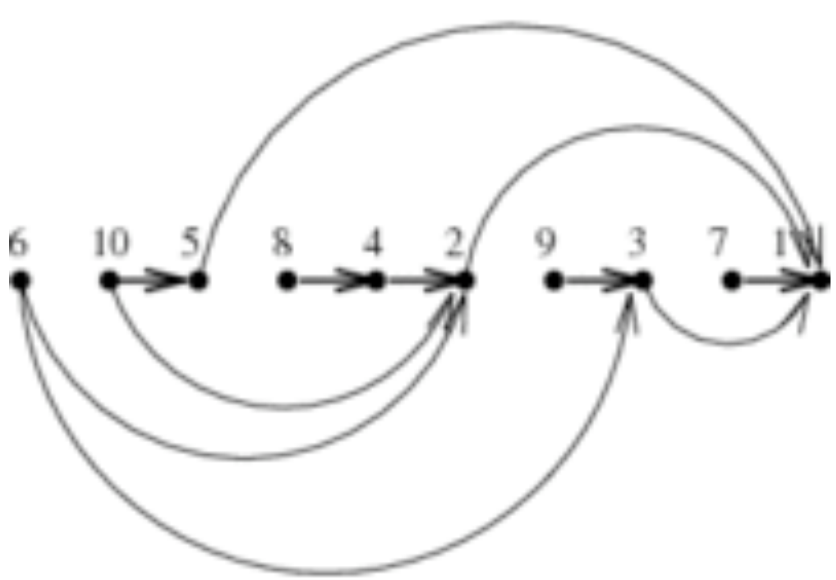


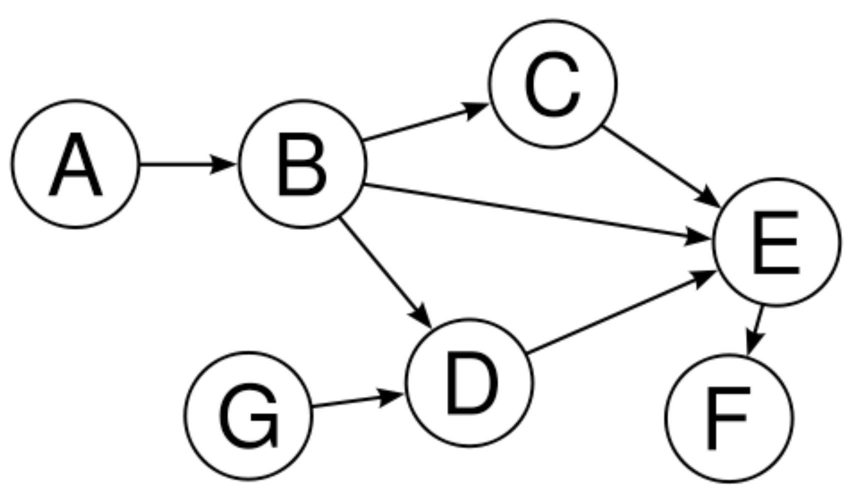


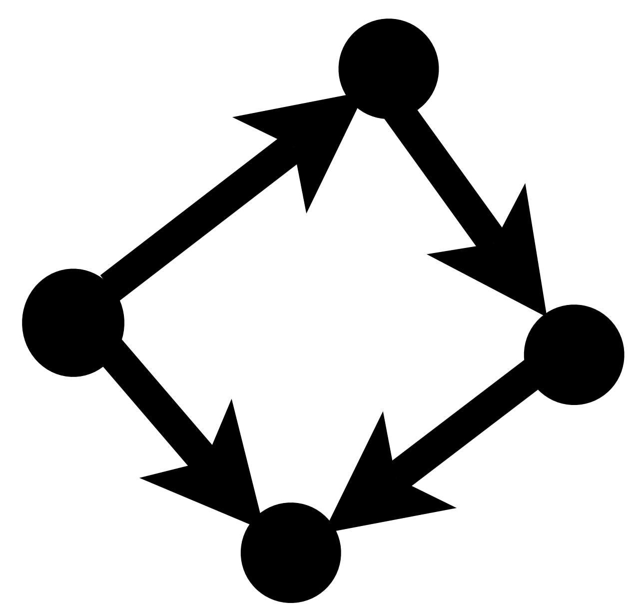
Searching DiGraphs

Question: Find all nodes reachable from s in a directed graph G.

Idea: Slightly modify BFS and DFS algorithms: they should only “find” nodes along out-edges.

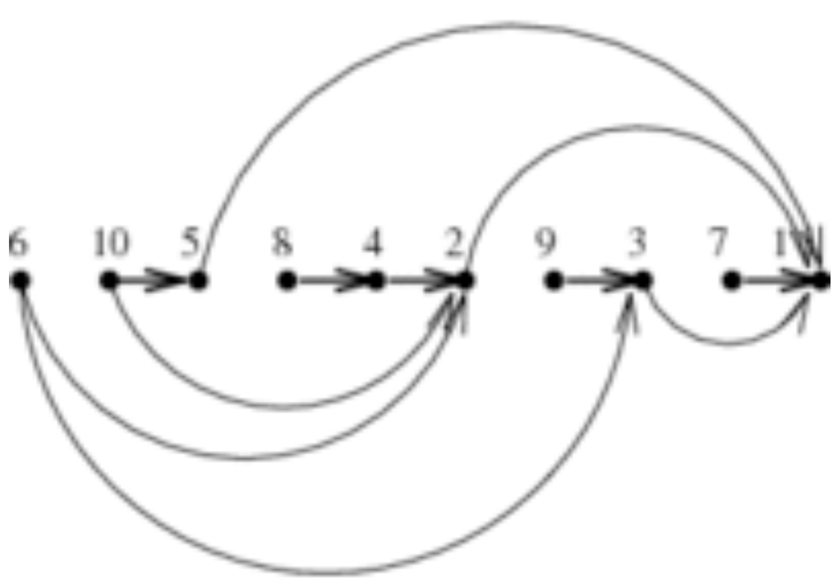


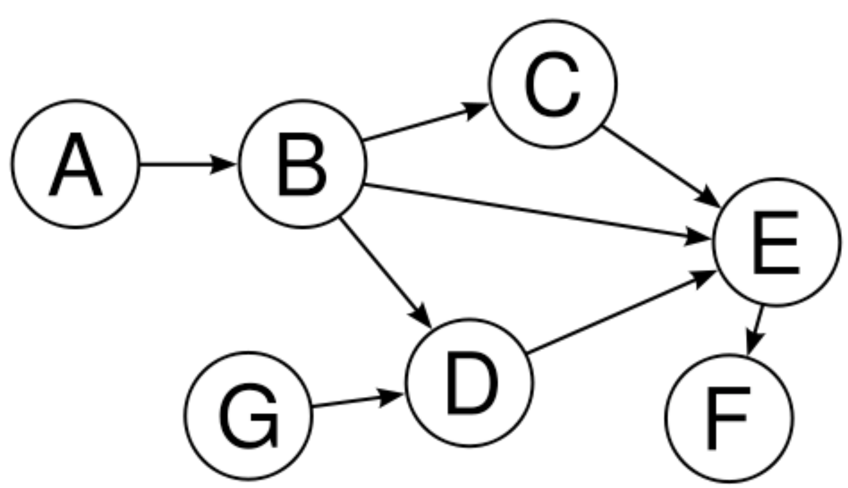


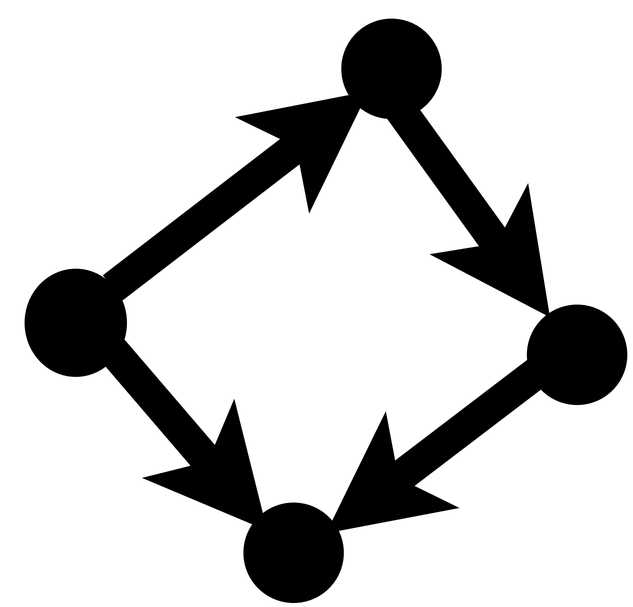


DFS and topo sort

Suppose we run DFS on a directed graph. What can we say about a node v when we mark it as INACTIVE?



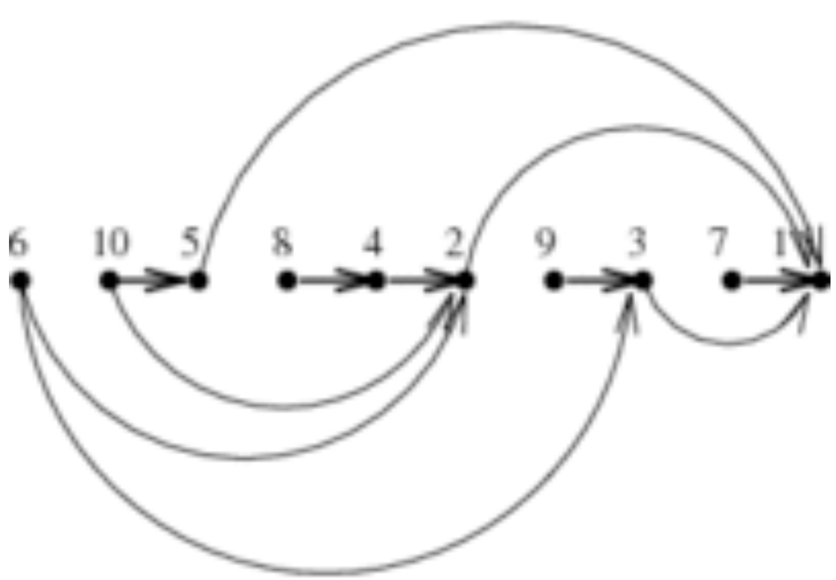


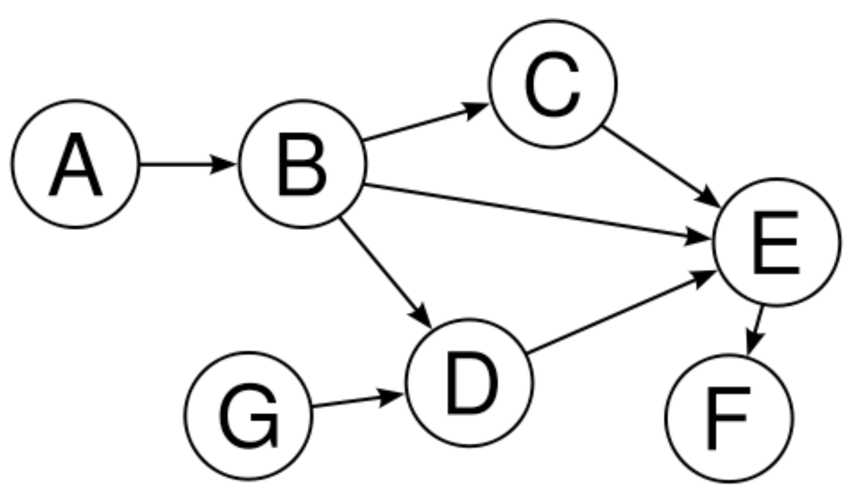


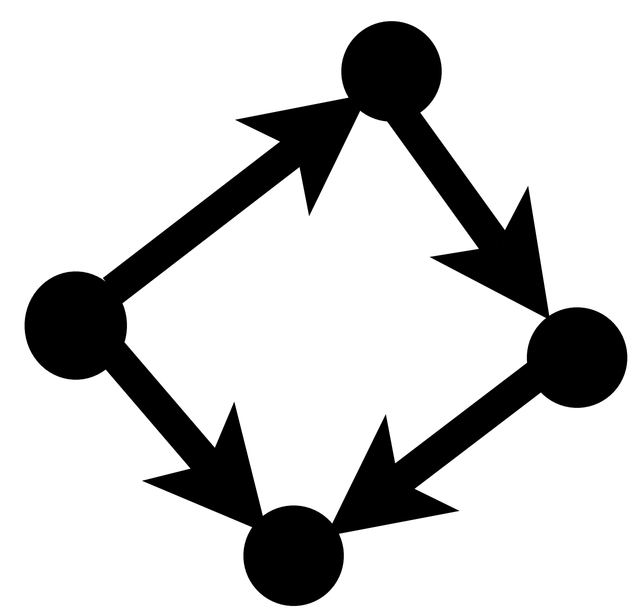
DFS and topo sort

Suppose we run DFS on a directed graph. What can we say about a node v when we mark it as INACTIVE?

Did we fully explore all nodes reachable from v?



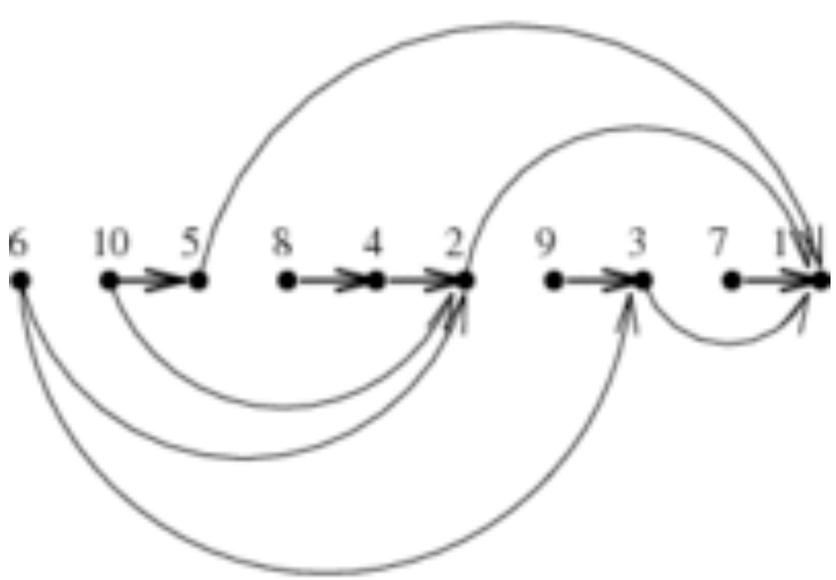


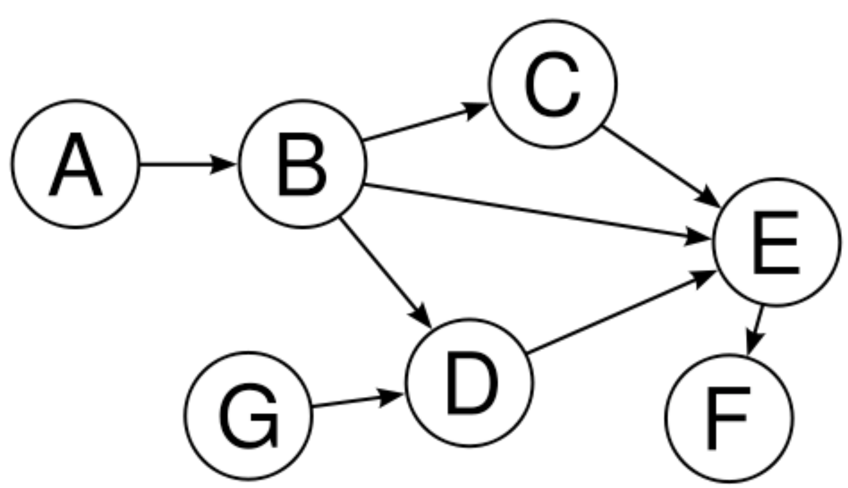


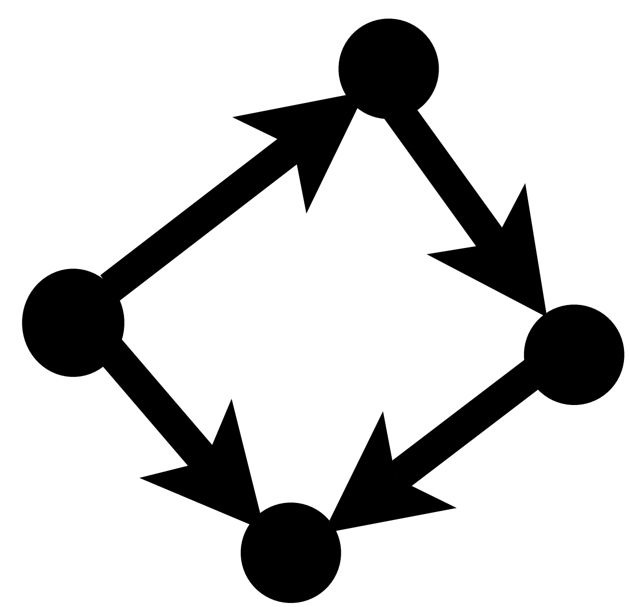
DFS and topo sort

Suppose we run DFS on a directed graph. What can we say about a node v when we mark it as INACTIVE?

Did we fully explore all nodes reachable from v? (to whiteboard)



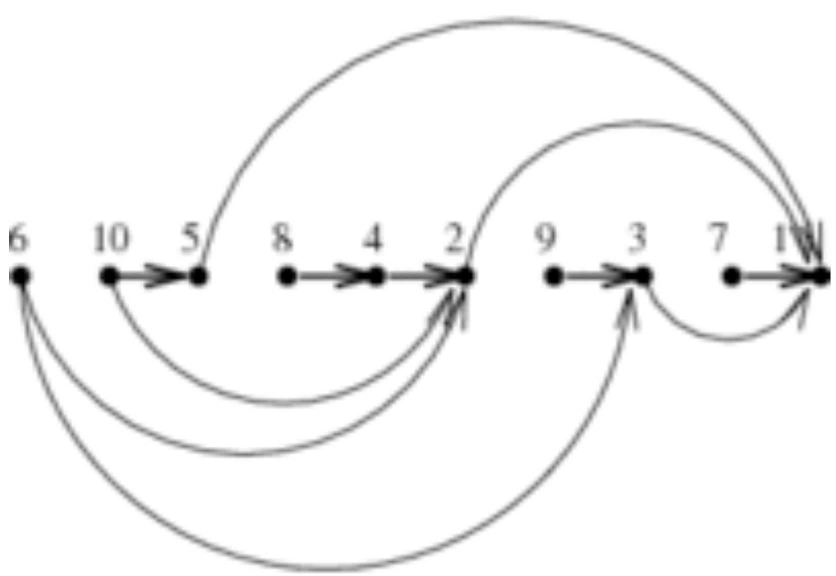


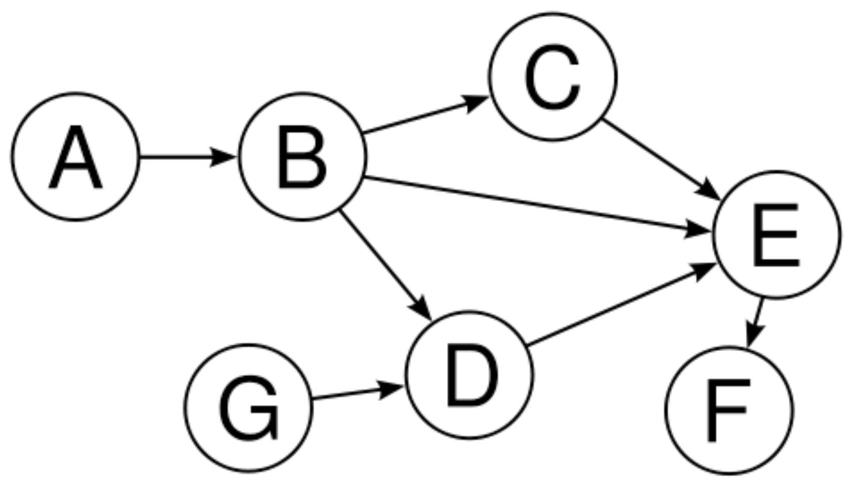


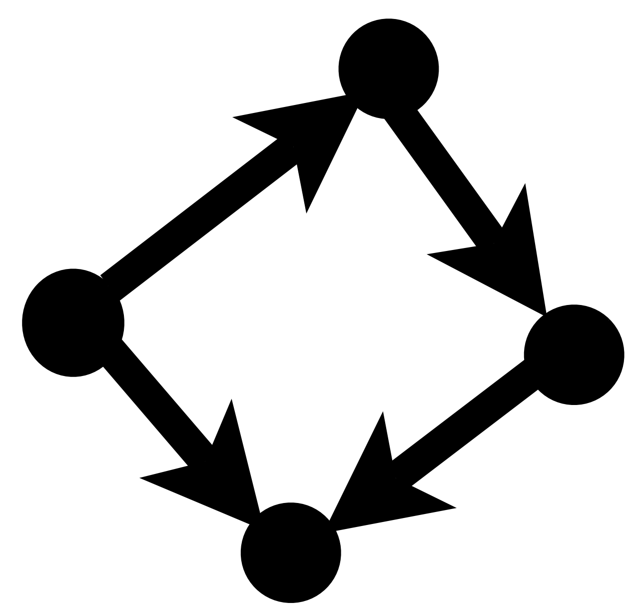
DFS and topo sort

Suppose we run DFS on a directed graph. What can we say about a node v when we mark it as INACTIVE?

Did we fully explore all nodes reachable from v? YES, unless there is a back-edge to an active node (an ancestor of the current node). In this case there is an oriented cycle!.







DFS and topo sort

Suppose we run DFS on a directed graph. What can we say about a node v when we mark it as INACTIVE?

Did we fully explore all nodes reachable from v? YES, unless there is a back-edge to an active node (an ancestor of the current node). In this case there is an oriented cycle!

So: IF we ever find a back-edge to an active node, output the cycle. Otherwise, we know all nodes reachable from v were marked INACTIVE before v was.

Algorithm

Start at a node s. Begin the DFS algorithm, following only out-edges.

Algorithm

Start at a node s. Begin the DFS algorithm, following only out-edges.

If we ever find an edge to an active node, there is a cycle. Each time we mark a node inactive, pre-pend it to the output.

Algorithm

Start at a node s. Begin the DFS algorithm, following only out-edges.

If we ever find an edge to an active node, there is a cycle. Each time we mark a node inactive, pre-pend it to the output.

If graph is a DAG, output will be a topological sort of the nodes reachable from s.

Algorithm

Start at a node s. Begin the DFS algorithm, following only out-edges.

If we ever find an edge to an active node, there is a cycle. Each time we mark a node inactive, pre-pend it to the output.

If graph is a DAG, output will be a topological sort of the nodes reachable from s.

If not given s: start by finding a source (node with no in-edges). A DAG always has one.

Algorithm

Start at a node s. Begin the DFS algorithm, following only out-edges.

If we ever find an edge to an active node, there is a cycle. Each time we mark a node inactive, pre-pend it to the output.

If graph is a DAG, output will be a topological sort of the nodes reachable from s.

If not given s: start by finding a source (node with no in-edges). A DAG always has one.

If not all nodes reached, go to next component.