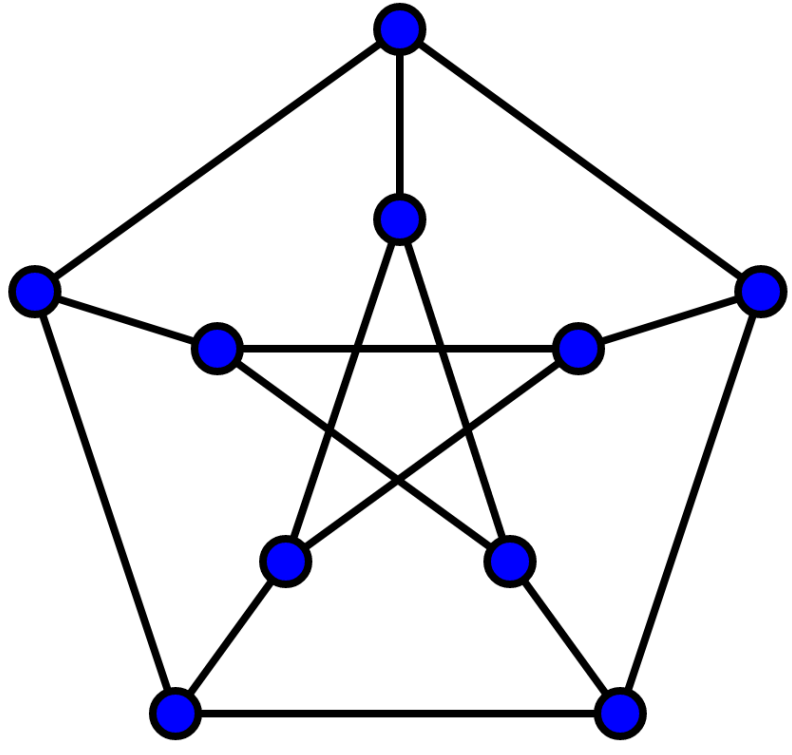
CS 344

LECTURE 7: GRAPHS

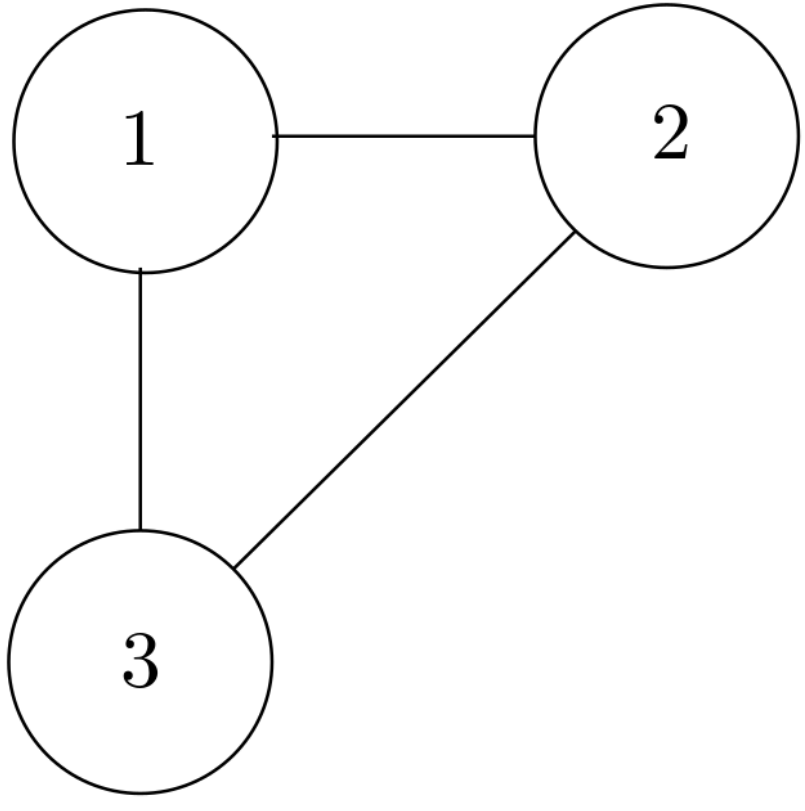
This is a graph:

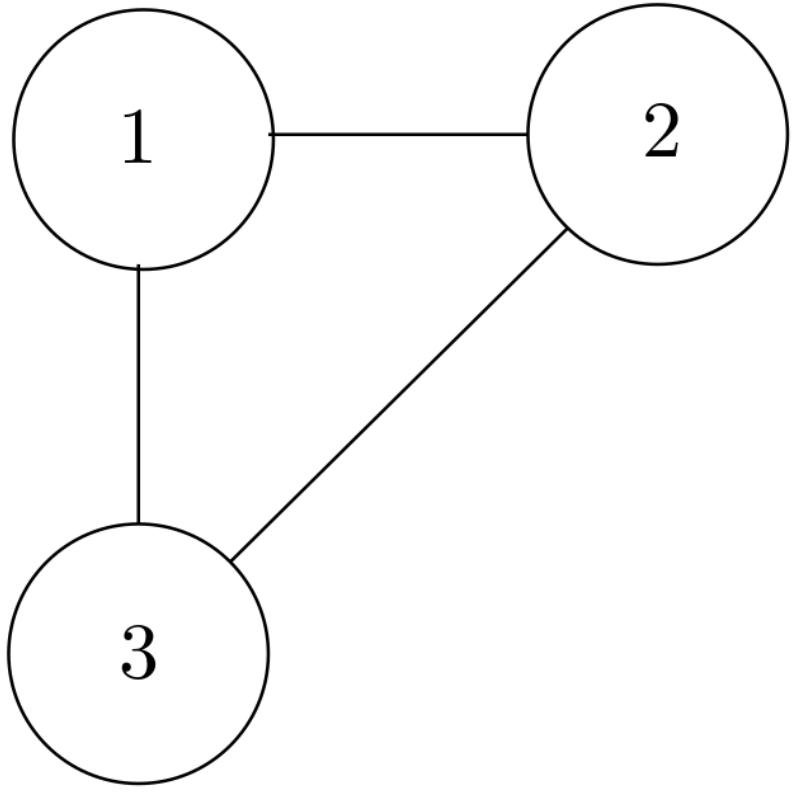


A graph is a pair $< V, E >$:

- V: vertices
- E: edges, where $E \subseteq V \times V$

This is a graph:





- $V = \{1, 2, 3\}$
- $E = \{(1, 2), (1, 3), (2, 3)\}$

GRAPH REPRESENTION

- adjacency matrix
- adjacency list

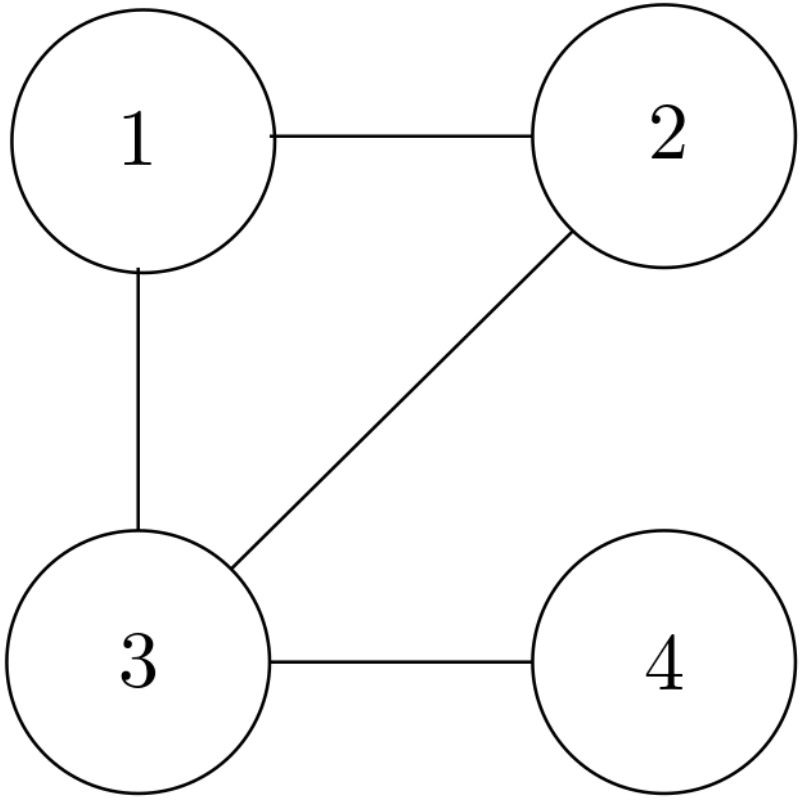
ADJACENCY MATRIX

If we have $|V| = n$ vertices,

the adjacency matrix is an $n \times n$ matrix:

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

ADJACENCY MATRIX



$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

ADJACENCY MATRIX

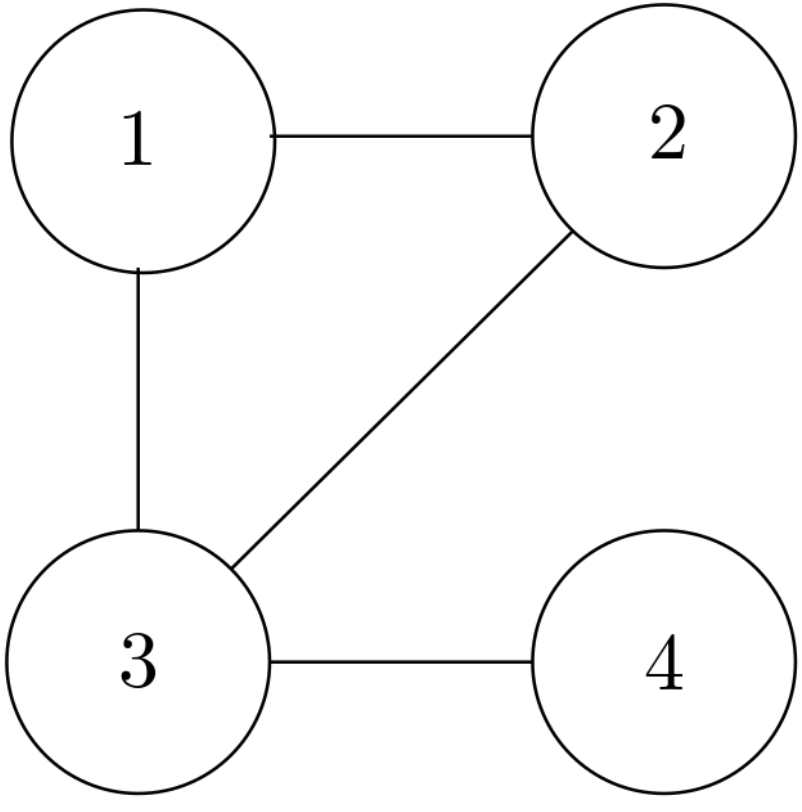
- Can check for an edge in $O(1)$ time
- Takes $O(n^2)$ space!

ADJACENCY LIST

Each vertex has a linked list of its (outgoing) neighbors

That is, for each vertex u, it has a list of every v for which $(u, v) \in E$

ADJACENCY LIST



Vertex	List
1	2, 3
2	1, 3
3	1, 2, 4
4	3

ADJACENCY LIST

- Have to walk a list to check for an edge
- Takes $O(|E|)$ space

SPARSITY

For a (connected) graph, we have that (roughly)

$$|V| \leq |E| \leq |V|^2$$

SPARSITY

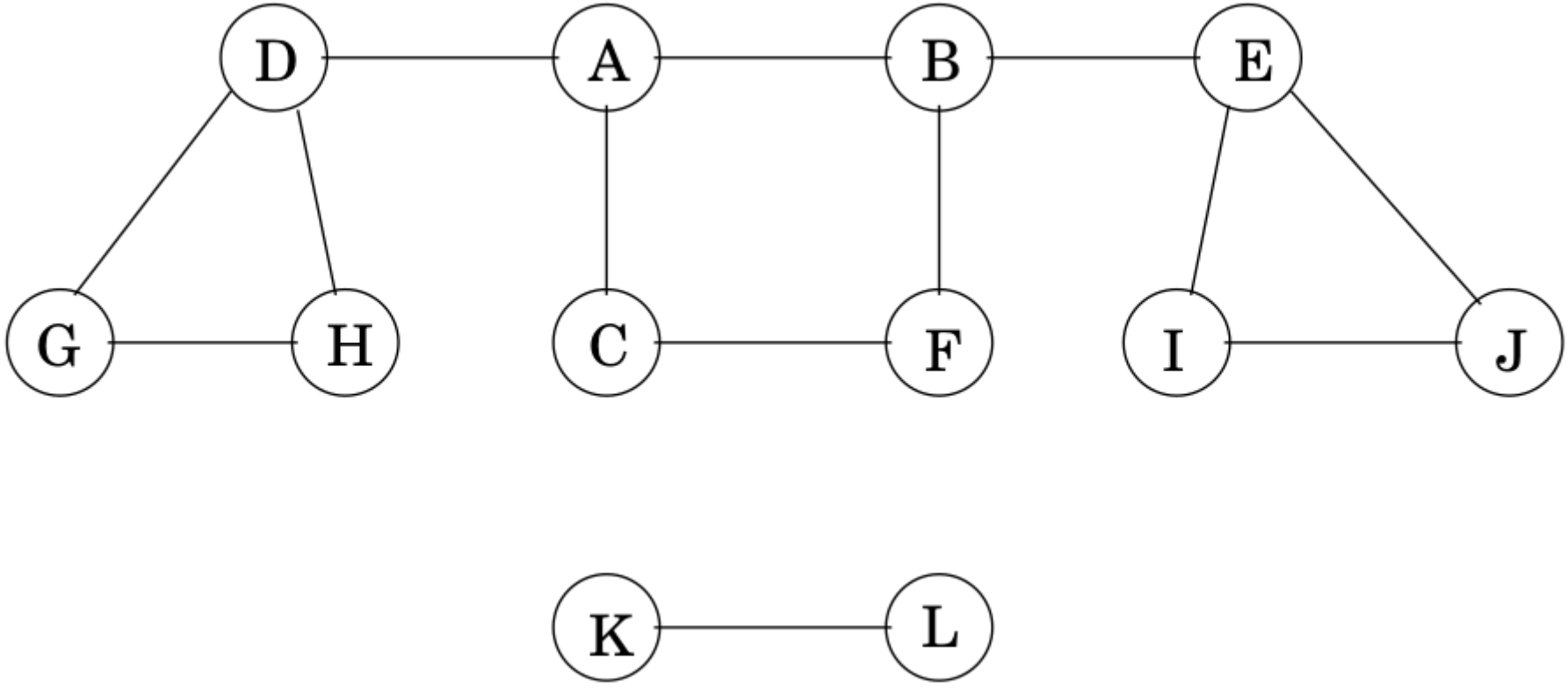
- Sparse: $|E|$ closer to $|V|$
- Dense: $|E|$ closer to $|V|^2$

SPARSITY

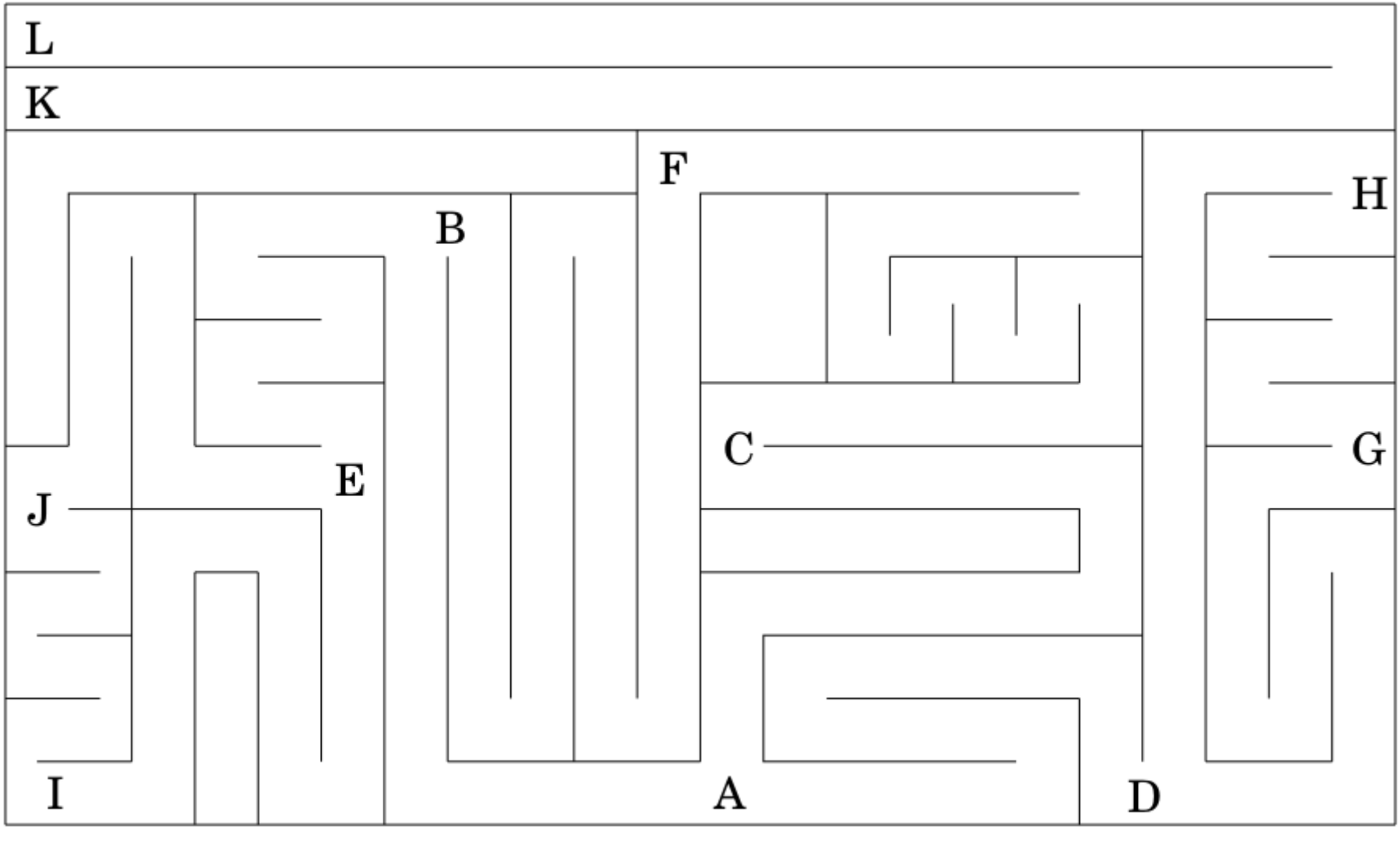
Which representation is better depends on the sparsity of the graph.

- For example, take the world-wide web
- Vertices are pages, edges are links
- $|V|$ is in the billions
- But most pages only link to a few other pages!

Suppose we have this graph:



Consider this as a maze:



To explore a maze we need two things:

- Chalk: to mark junctions you've visited
- String: to find your way back

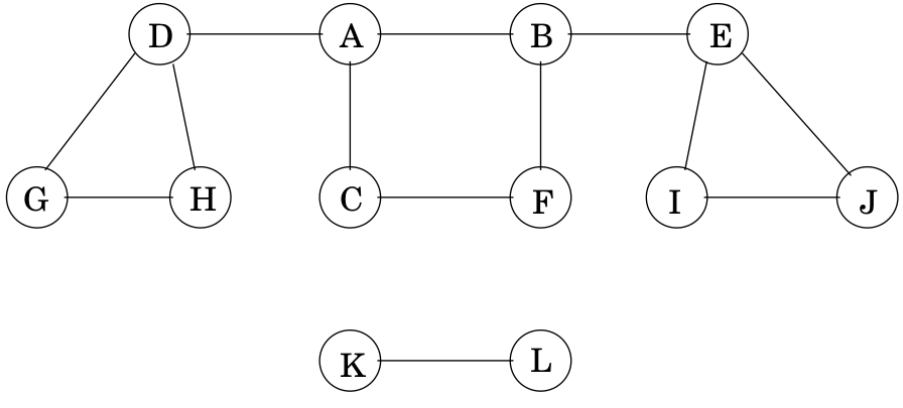
To explore a graph we need two things:

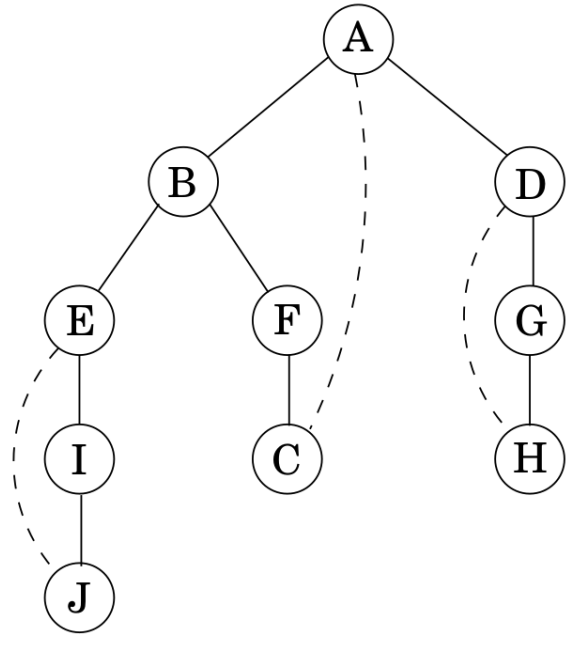
- A variable to mark nodes you've visited
- A stack to find your way back

Let's find all nodes reachable from a particular node:

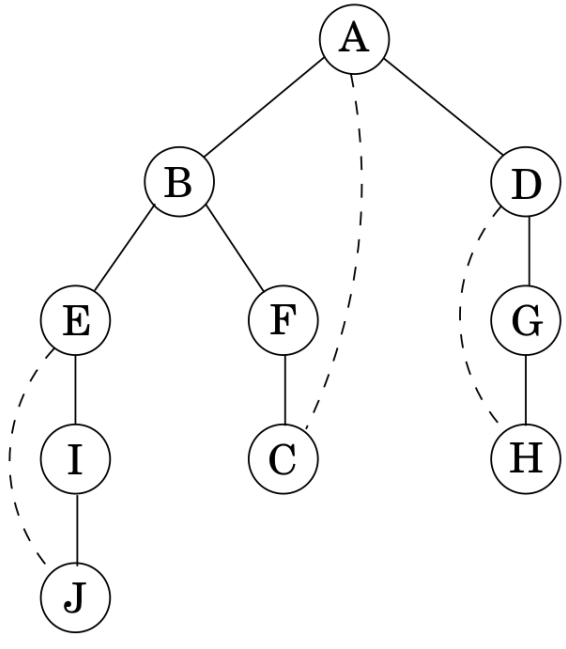
```
procedure explore(G, v)
Input:    G = (V, E) is a graph; v ∈ V
Output:   visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
   if not visited(u):  explore(u)
postvisit(v)
```

Let's run explore(A) on the graph on the left:





Then we have two kinds of edges:



- tree edges: black lines
- back edges: dotted lines

Is the explore function correct?

Does it find all vertices reachable from v?

Suppose not, that it misses vertex u:

More generally, correctness here means for any $k \geq 0$, all nodes within k hops of v will be visited.

- Base case: $k = 0$
- Inductive step: If all nodes k hops away are visited, then all nodes $k + 1$ hops away are too

DEPTH-FIRST SEARCH

The explore function only visits nodes reachable from the starting point

To examine the rest of the graph, we can repeatedly call explore

```
procedure dfs(G)

for all v ∈ V:
    visited(v) = false

for all v ∈ V:
    if not visited(v):  explore(v)
```

TIME TO RUN DFS

- $O(1)$ to mark node visited, call pre/postvisit
- Then loop through and scan adjacent edges

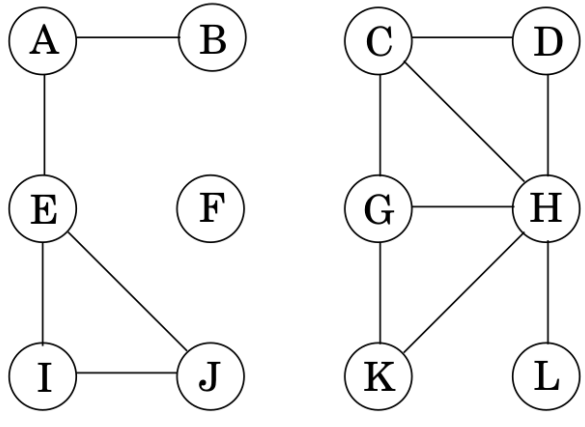
TIME TO RUN DFS

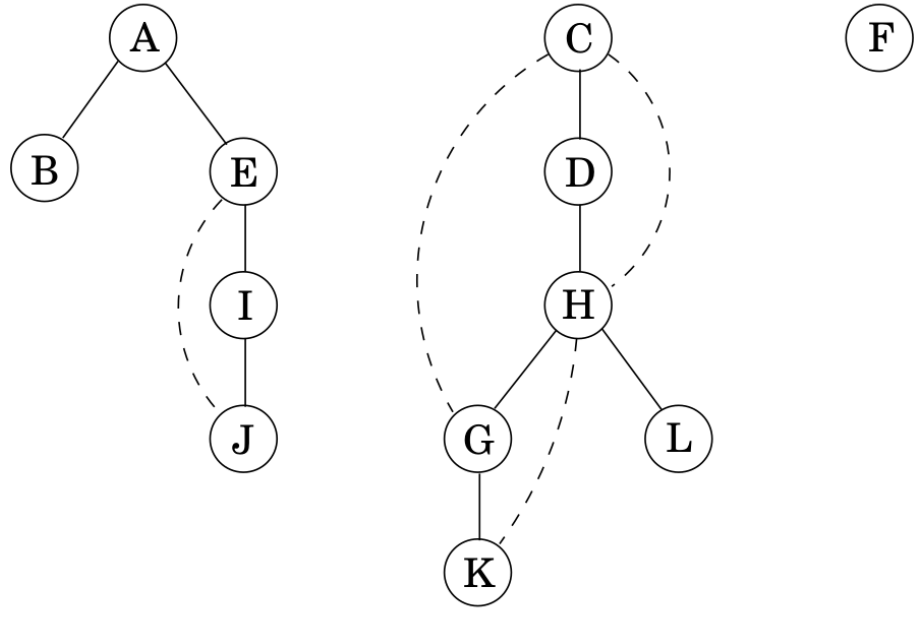
For all vertices together,

- $O(|V|)$ to mark nodes visited, call pre/postvisit
- Each edge (u, v) will be visited twice
  - once in explore(u)
  - once in explore(v)
- Therefore $O(|E|)$ work to scan edges

Total: $O(|V| + |E|)$

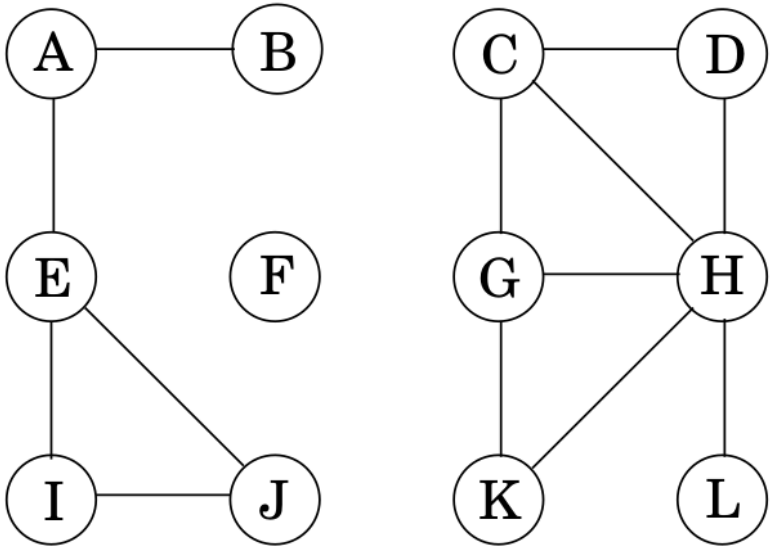
Running DFS on this graph (left) generates this forest (right):



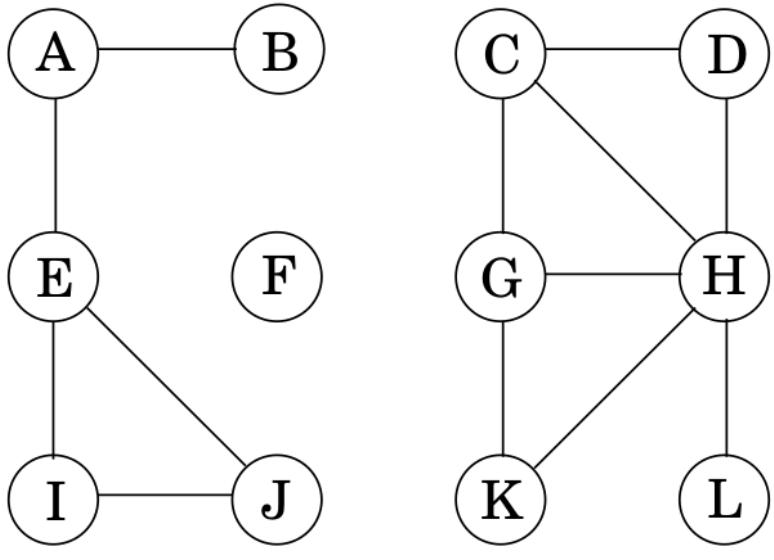


An undirected graph is **connected** if there is a path from any vertex to any other.

In a disconnected graph, each connected subgraph is called a **connected component**.

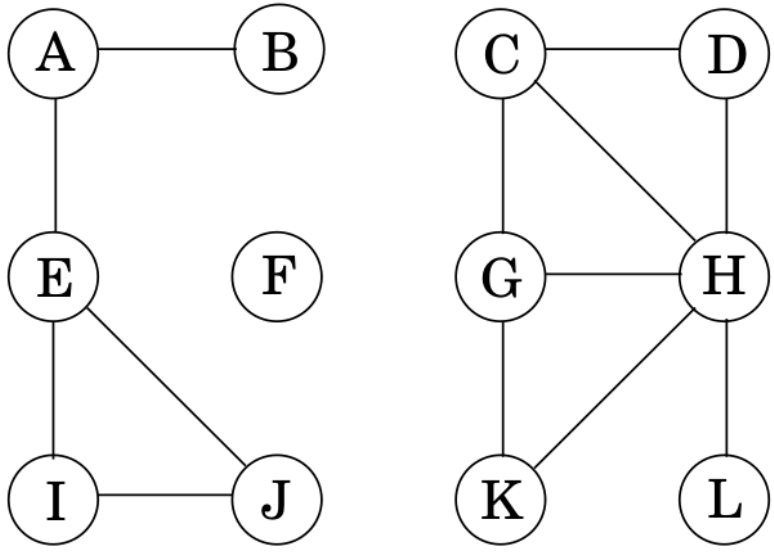


Connected components:



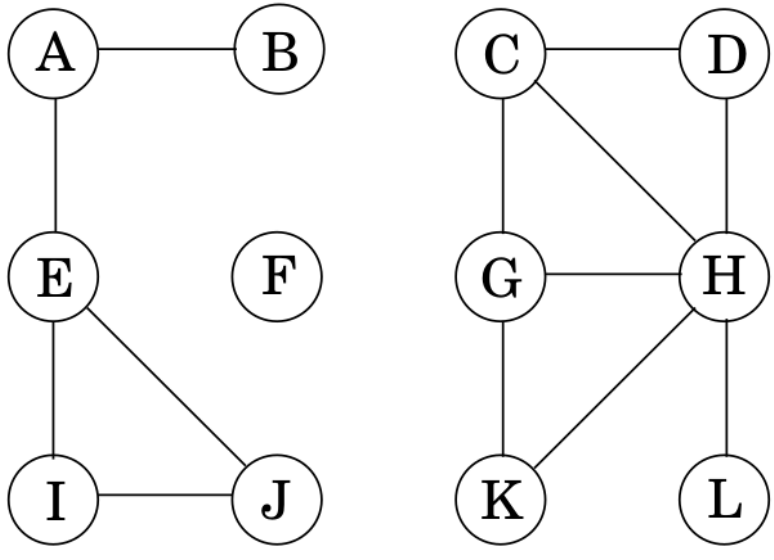
Connected components:

- $\{A, B, E, I, J\}$



Connected components:

- $\{A, B, E, I, J\}$
- $\{C, D, G, H, K, L\}$



Connected components:

- $\{A, B, E, I, J\}$
- $\{C, D, G, H, K, L\}$
- $\{F\}$

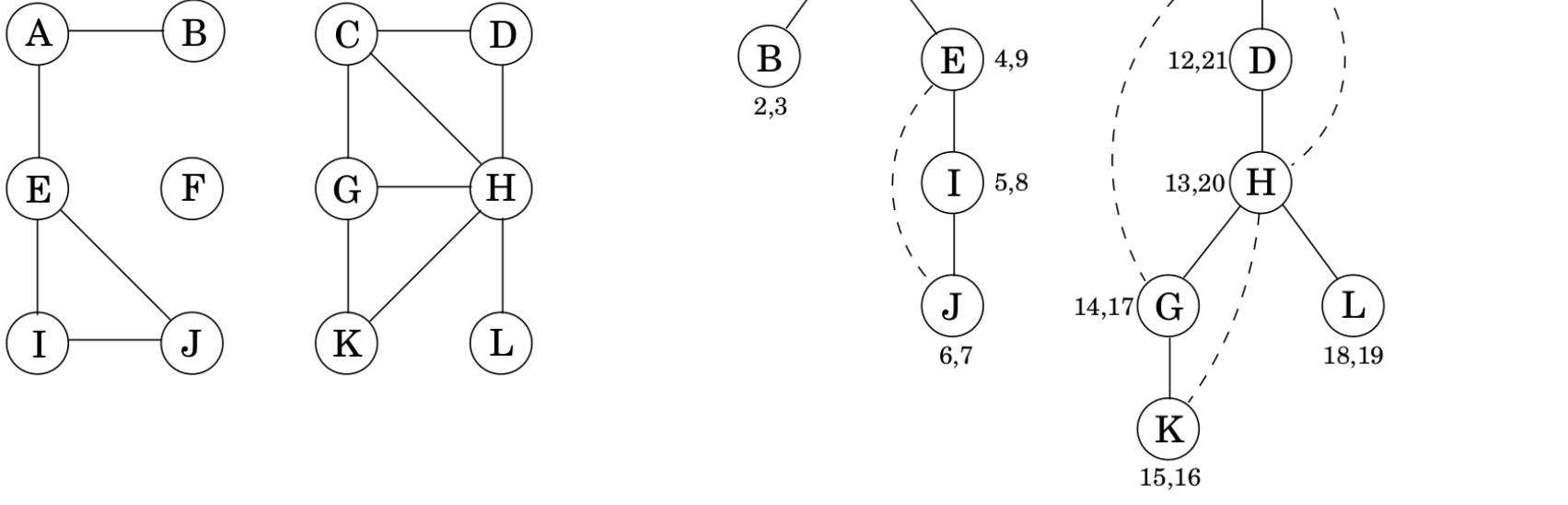
```
procedure previsit(v)
ccnum[v] = cc
```

- Initialize to 0
- Increment each time DFS calls explore

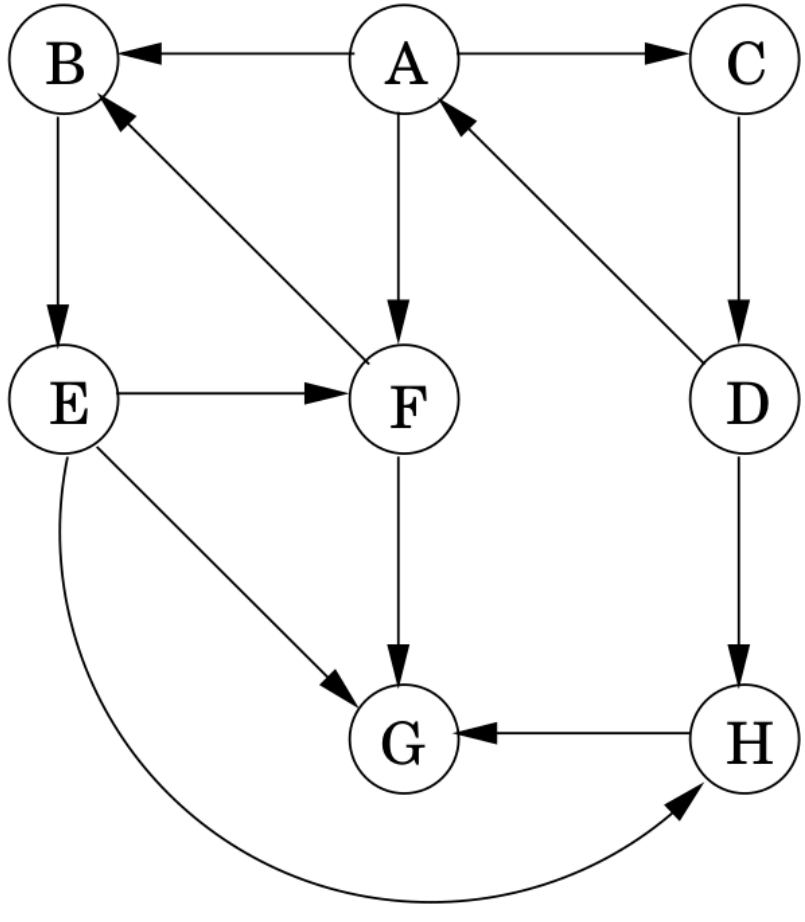
Let's add a counter to see when we enter/leave nodes:

```
procedure previsit(v)
pre[v] = clock
clock = clock + 1

procedure postvisit(v)
post[v] = clock
clock = clock + 1
```

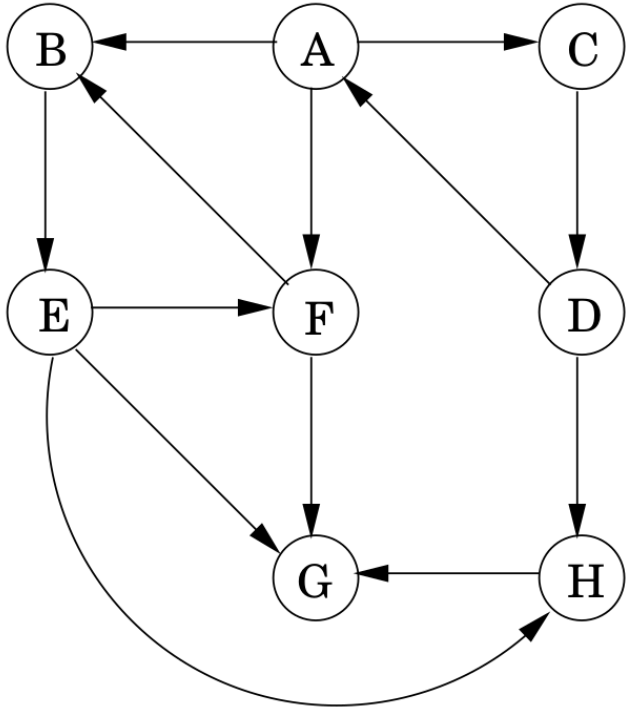
Then our forest now looks like this:

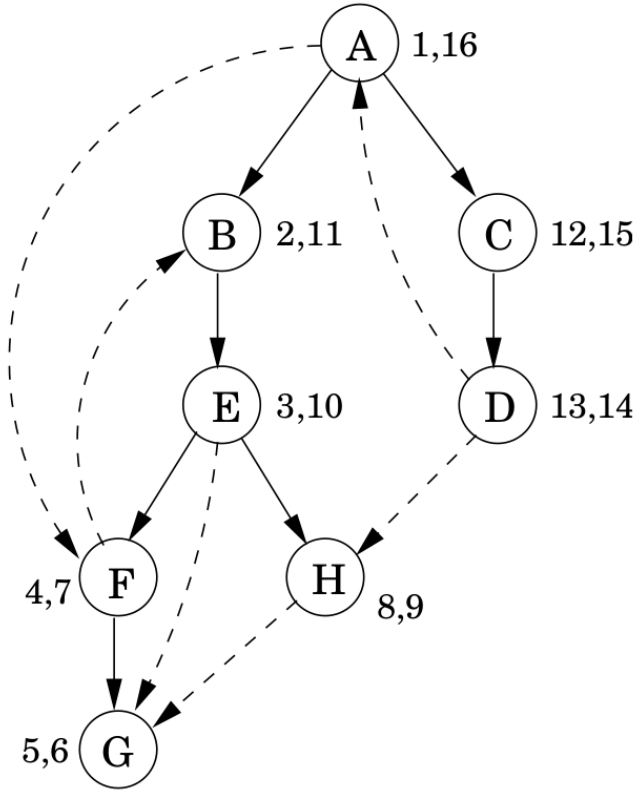


DIRECTED GRAPHS

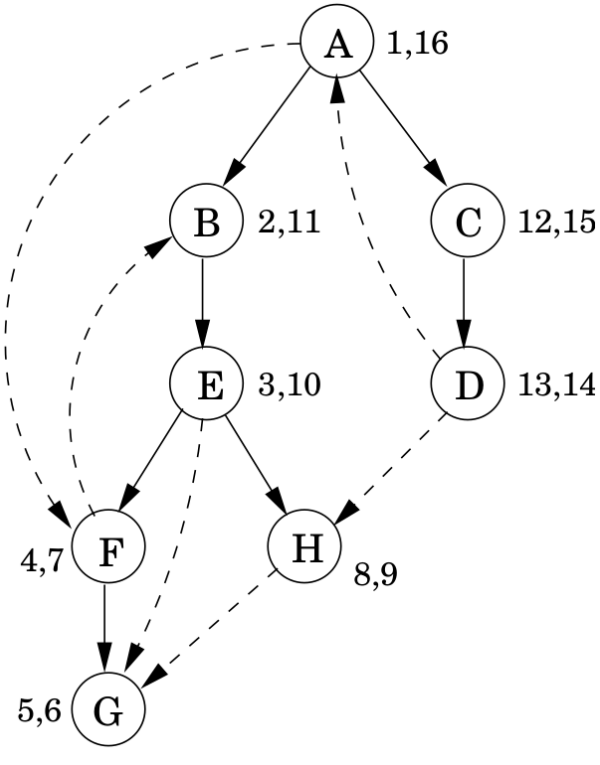


DFS ON DIRECTED GRAPHS



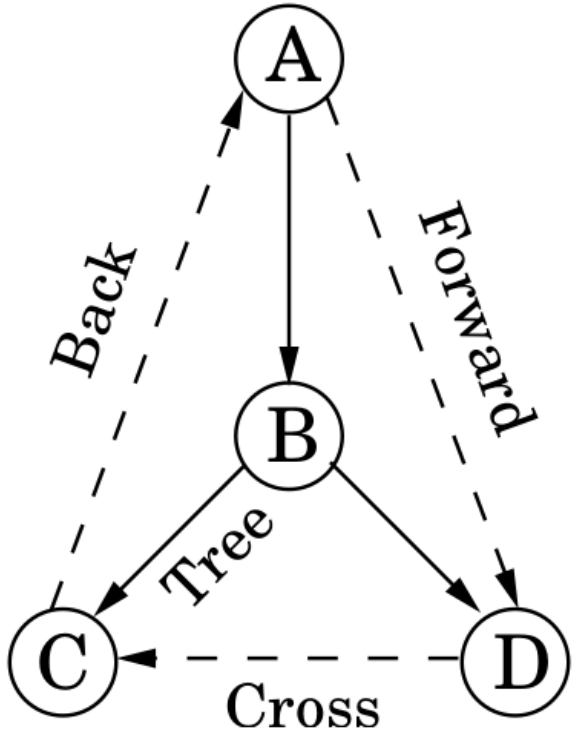


Terminology:

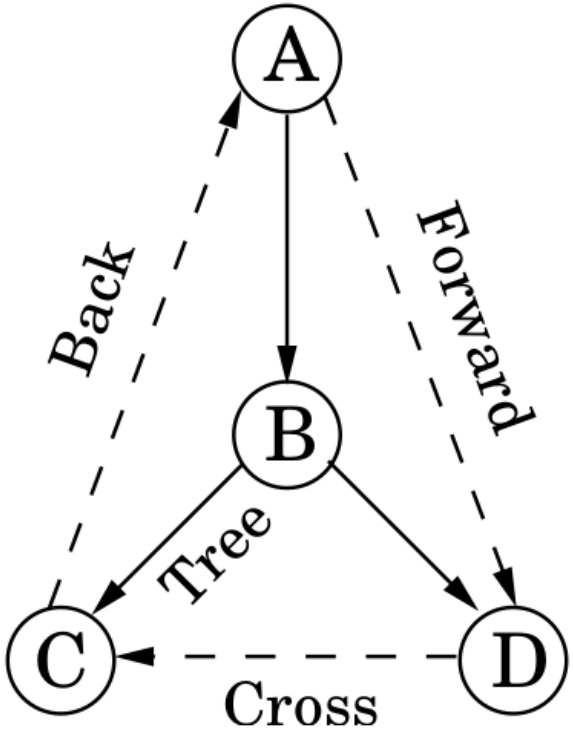


- A is the **root**
- E has **descendants** F, G, and H
- E is an **ancestor** of F, G, and H
- C is the **parent** of D
- D is the **child** of C

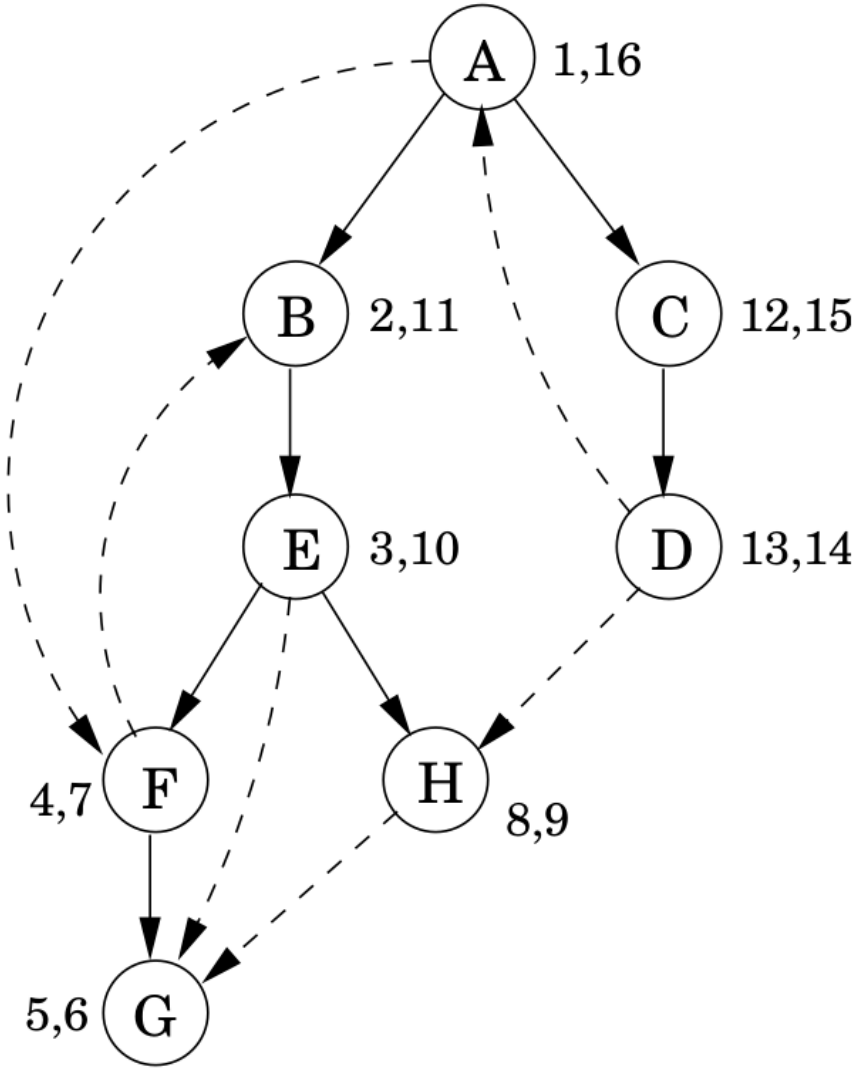
We can also have finer-grained distinctions on edges in the generated tree:



- tree edges
- forward edges
- back edges
- cross edges



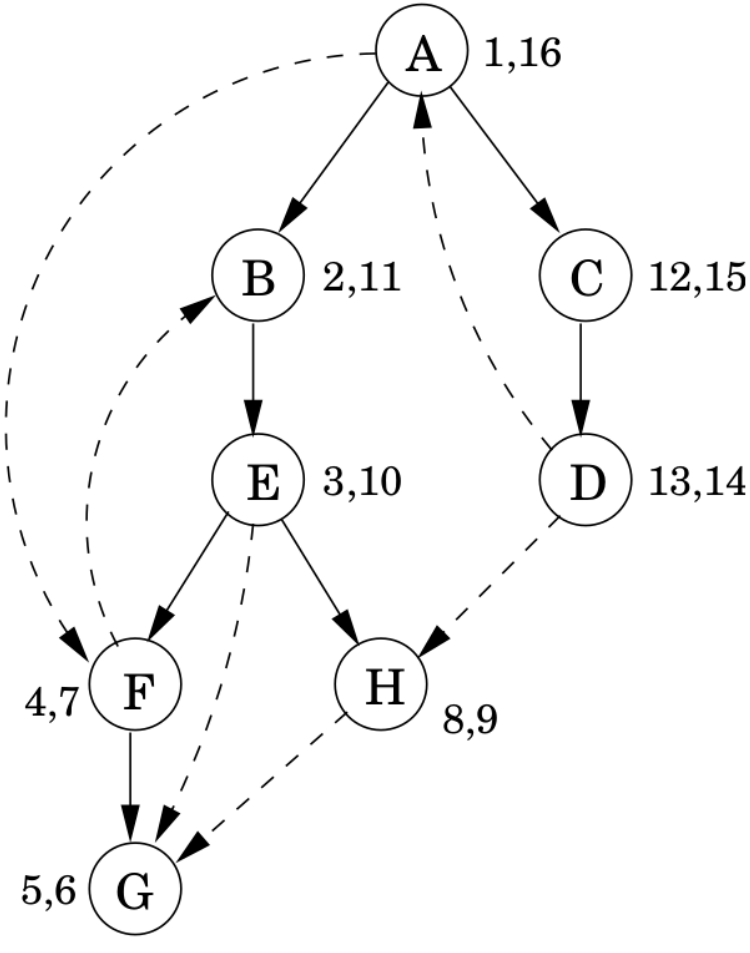
- tree edges: part of the DFS forest
- forward edges: node to nonchild descendent
- back edges: node to ancestor
- cross edges: lead to neither descendent nor ancestor



How many

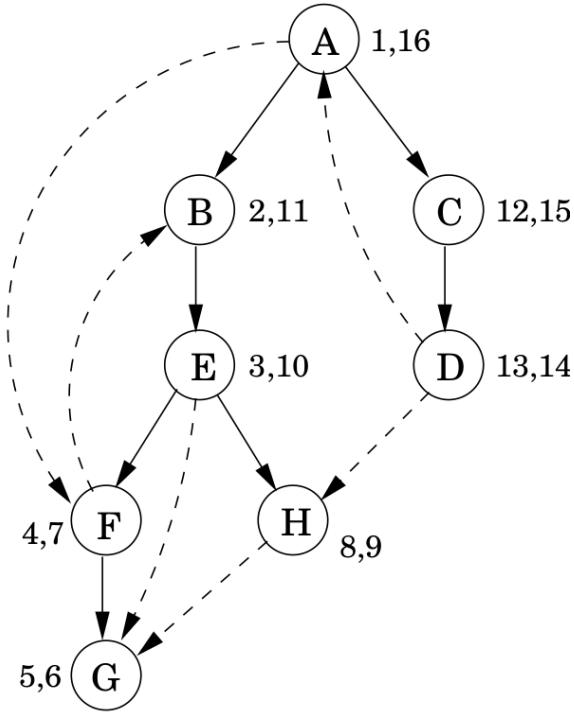
- Forward edges?
- Back edges?
- Cross edges?

These relationships can be inferred from the pre and post numbers!



Vertex u is an ancestor of vertex v:

$$[_u \quad [_v \quad]_v \quad]_u$$



Edge categories:

`pre/post` *ordering for* (u, v)	*Edge type*
$[_u \; [_v \;]_v \;]_u$	Tree/forward
$[_v \; [_u \;]_u \;]_v$	Back
$[_v \;]_v \; [_u \;]_u$	Cross

CYCLES

A cycle is a circular path $v_0 \to v_1 \to \ldots \to v_k \to v_0$.

A graph without cycles is acyclic.

DFS AND CYCLES

A directed graph has a cycle if and only if DFS reveals a back edge.

A directed graph has a cycle if and only if DFS reveals a back edge.

A directed graph has a cycle if and only if DFS reveals a back edge.

- ($\Leftarrow$) If (u, v) is a back edge,

A directed graph has a cycle if and only if DFS reveals a back edge.

- ($\Leftarrow$) If (u, v) is a back edge,
  - There is a path from v to u

A directed graph has a cycle if and only if DFS reveals a back edge.

- ($\Leftarrow$) If (u, v) is a back edge,
  - There is a path from v to u
  - That path plus the back edge is a cycle

A directed graph has a cycle if and only if DFS reveals a back edge.

- ($\Leftarrow$) If (u, v) is a back edge,
  - There is a path from v to u
  - That path plus the back edge is a cycle
- ($\Rightarrow$) If $v_0 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$ is a cycle:

A directed graph has a cycle if and only if DFS reveals a back edge.

- ($\Leftarrow$) If (u, v) is a back edge,
  - There is a path from v to u
  - That path plus the back edge is a cycle
- ($\Rightarrow$) If $v_0 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$ is a cycle:
  - Consider the first node v_i to be discovered

A directed graph has a cycle if and only if DFS reveals a back edge.

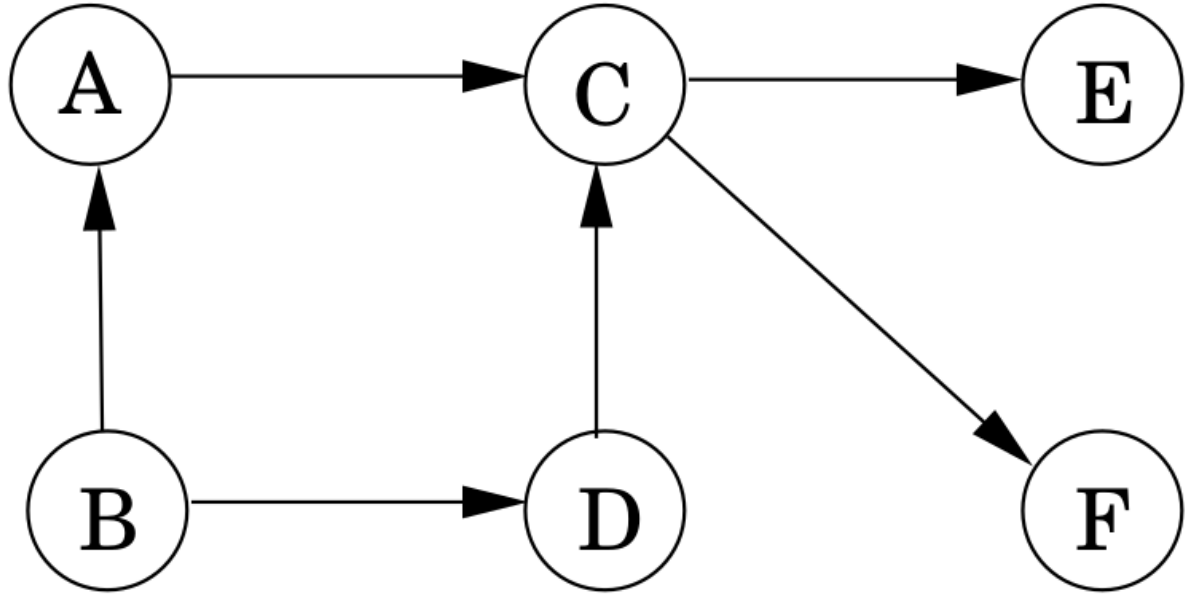
- ($\Leftarrow$) If (u, v) is a back edge,
  - There is a path from v to u
  - That path plus the back edge is a cycle
- ($\Rightarrow$) If $v_0 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$ is a cycle:
  - Consider the first node v_i to be discovered
  - It will reach all other nodes on the cycle

A directed graph has a cycle if and only if DFS reveals a back edge.

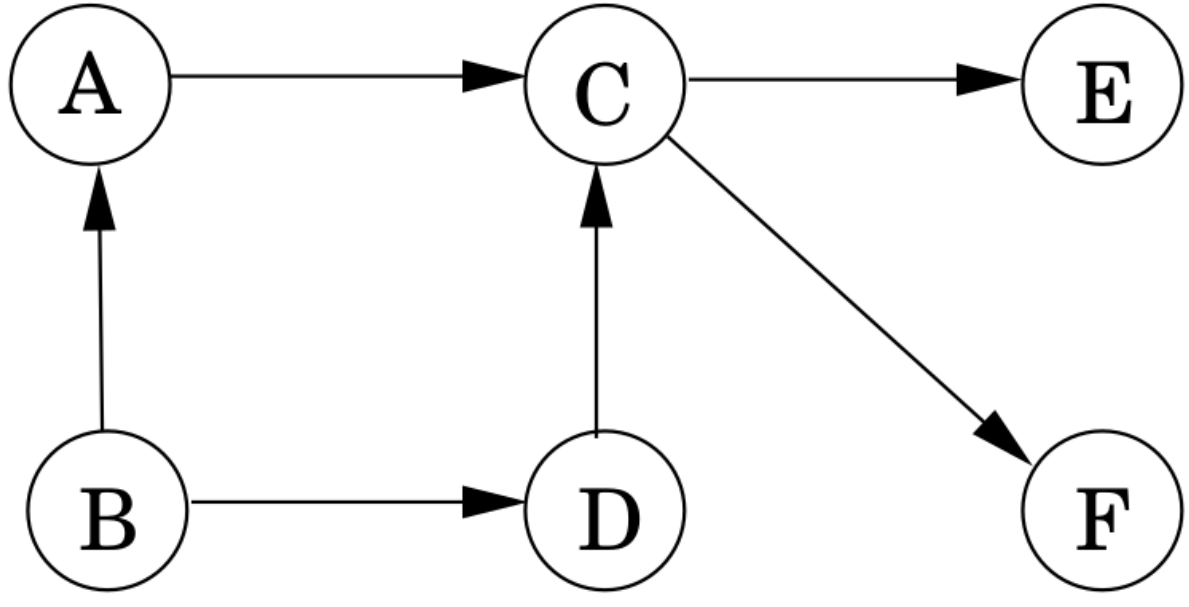
- ($\Leftarrow$) If (u, v) is a back edge,
  - There is a path from v to u
  - That path plus the back edge is a cycle
- ($\Rightarrow$) If $v_0 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$ is a cycle:
  - Consider the first node v_i to be discovered
  - It will reach all other nodes on the cycle
  - The edge $v_{i-1} \rightarrow v_i$ will be a back edge

DIRECTED ACYCLIC GRAPH (DAG)

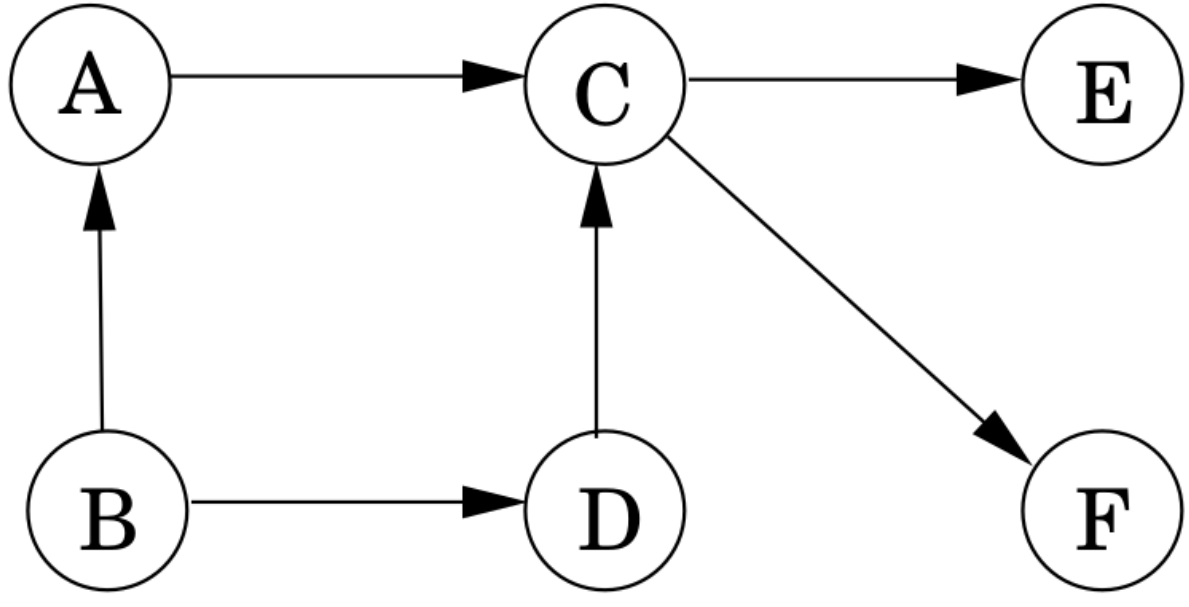
Dags are good for modeling hierarchies or dependencies (e.g., course prerequisites).



Given a dag, we may want to linearize (topologically sort) the nodes.



Given a dag, we may want to linearize (topologically sort) the nodes.



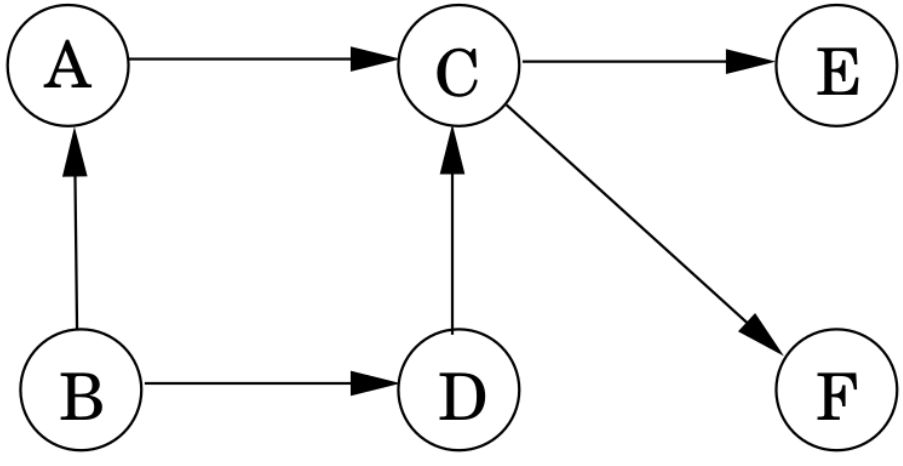
One possibility: B, A, D, C, E, F

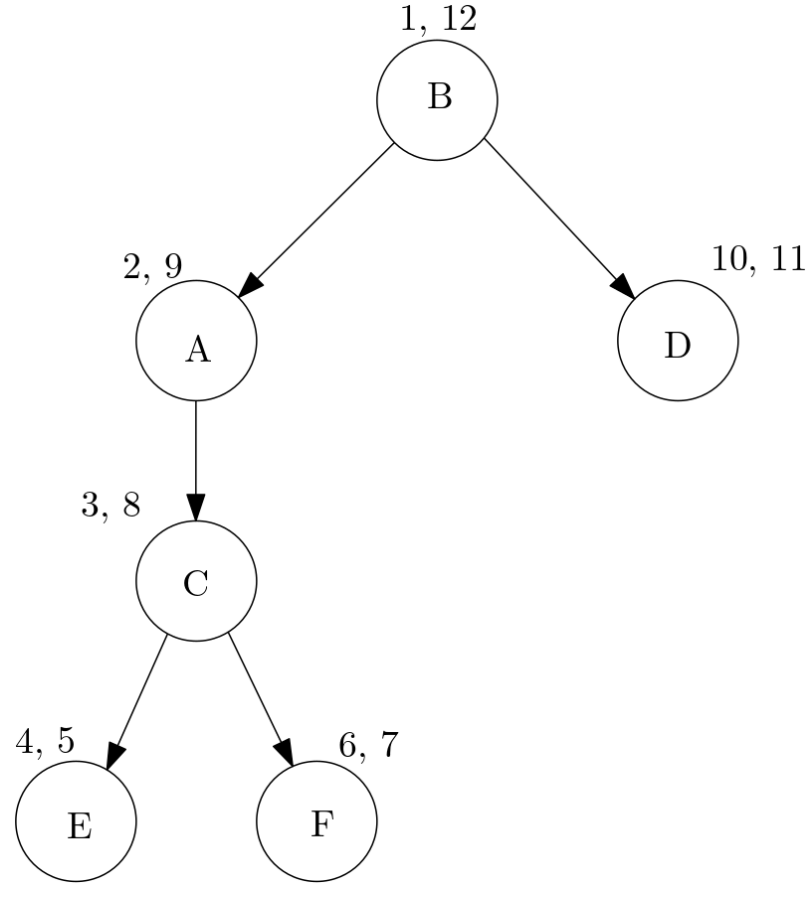
How can we linearize a dag algorithmically?

How can we linearize a dag algorithmically?

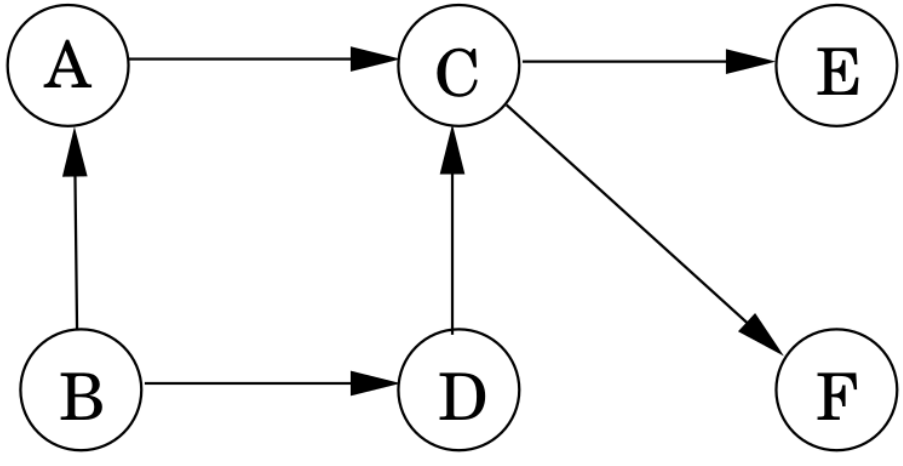
List nodes in decreasing order of post numbers

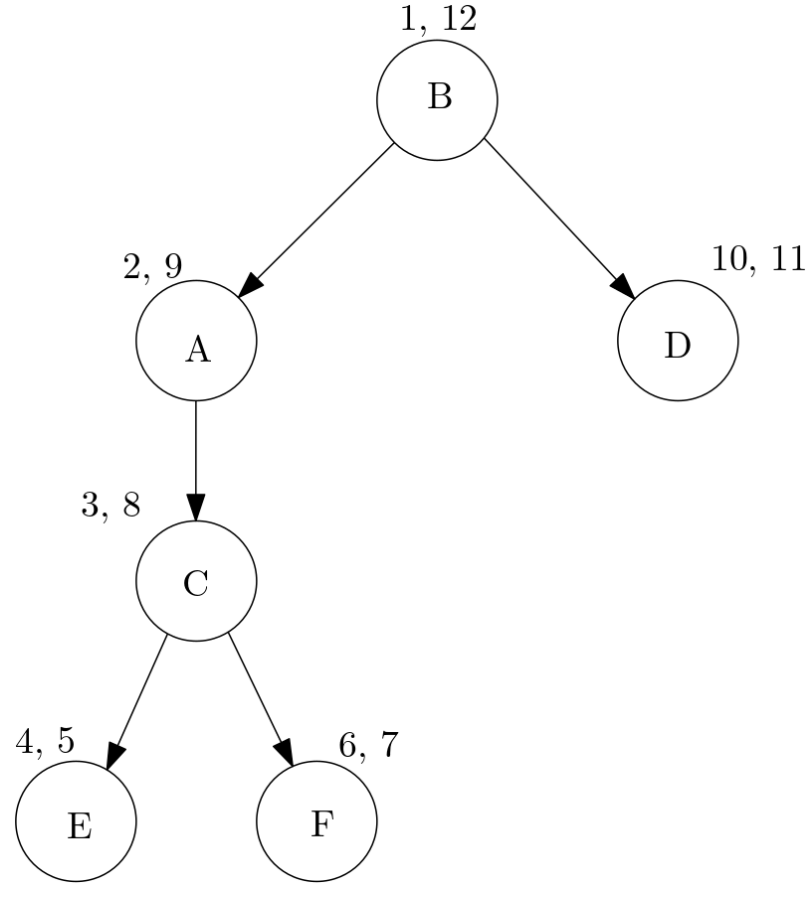
List nodes in decreasing order of post numbers





List nodes in decreasing order of post numbers





$$B, D, A, C, F, E$$

Given a dag, we can define two kinds of special nodes:

- source: a node with no incoming edges
- sink: a node with no outgoing edges

Given a dag, we can define two kinds of special nodes:

- source: a node with no incoming edges
- sink: a node with no outgoing edges

  Then a linearization must start with a source and end with a sink.

Another algorithm for linearization:

Another algorithm for linearization:

- find a source

Another algorithm for linearization:

- find a source
- output it

Another algorithm for linearization:

- find a source
- output it
- delete it

Another algorithm for linearization:

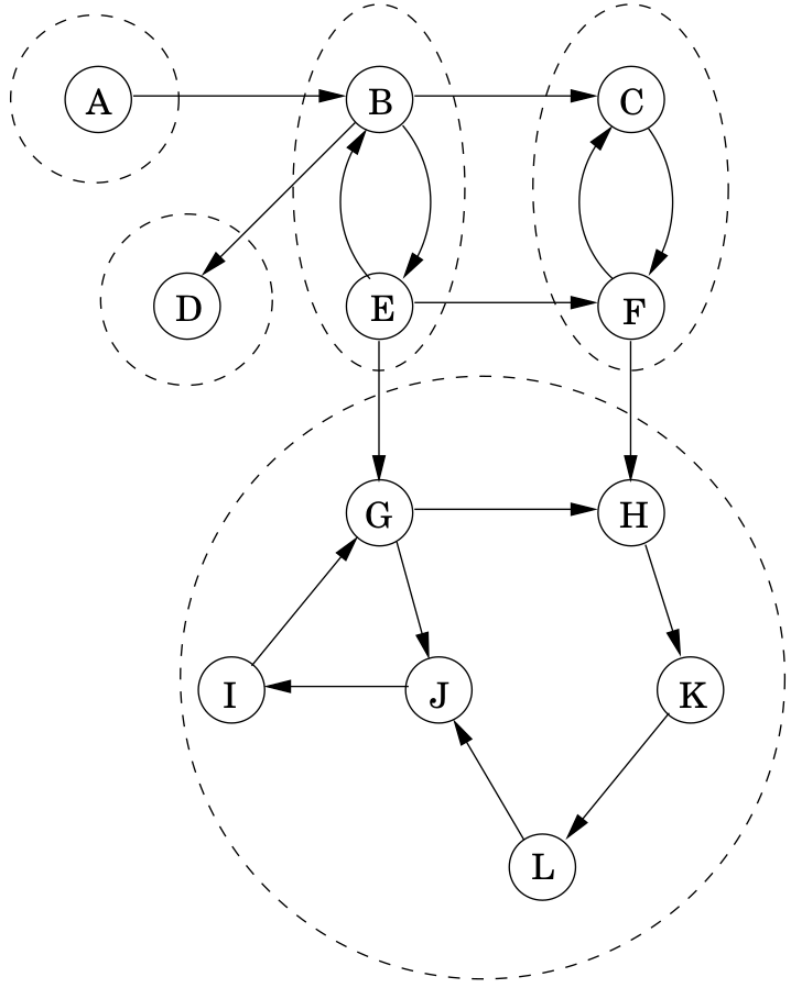
- find a source
- output it
- delete it
- repeat until graph is empty

STRONGLY-CONNECTED COMPONENTS

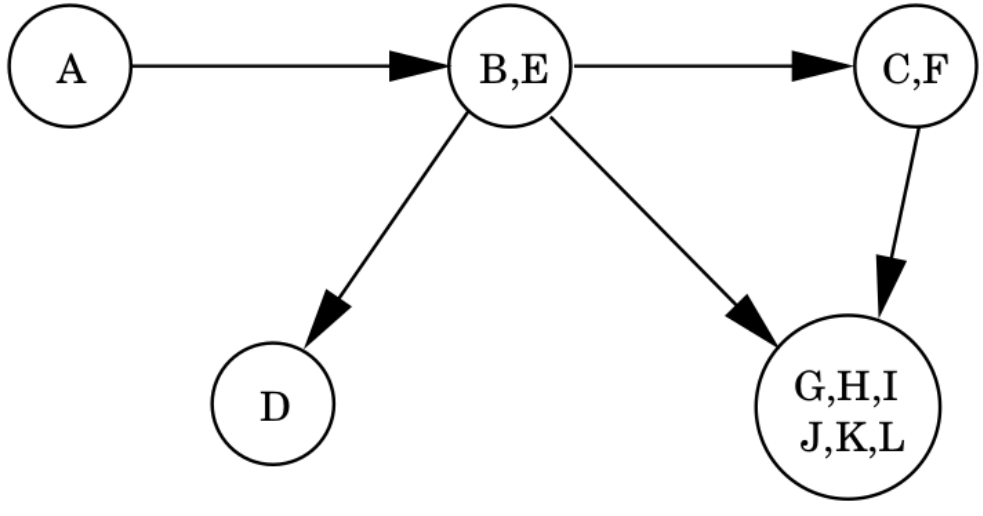
Two nodes u and v are connected if there is a u-v path as well as a v-u path.

We can partition a directed graph into a set **strongly-connected components** (SCCs), where all vertices are connected.

Dotted lines indicate the SCCs:



Then we can shrink each component to a single meta-vertex:



A directed graph is a dag of its strongly-connected components.

How can we decompose a graph to SCCs?

- If we found a node in a sink SCC
  - explore would find all nodes in its SCC
- Then we could remove it and repeat

How can we decompose a graph to SCCs?

But it's easier to find a node in a source SCC:

How can we decompose a graph to SCCs?

But it's easier to find a node in a source SCC:

node with the highest post number in DFS

More generally,

- if C and C' are SCCs,
- and there's a $C \to C'$ edge,
- then the highest post number in $C >$ highest in C'

This means SCCs can be linearized by descreasing highest post numbers.

(a generalization of linearization of dags)

How can we decompose a graph to SCCs?

We can find a source SCC, but we need a sink SCC

How can we decompose a graph to SCCs?

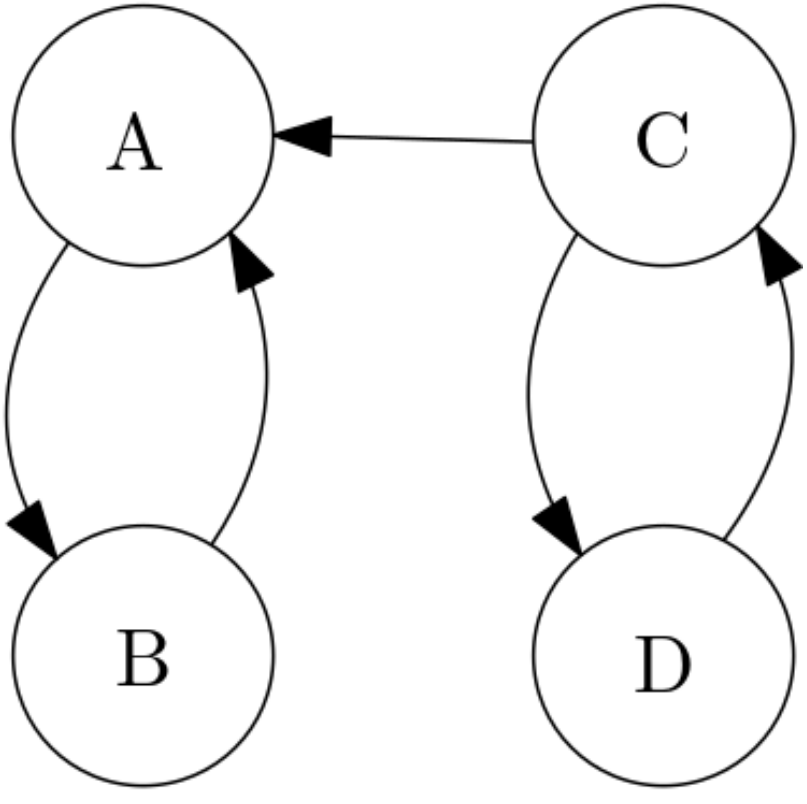
We can find a source SCC, but we need a sink SCC

Transform the graph G into the reverse graph G^R!

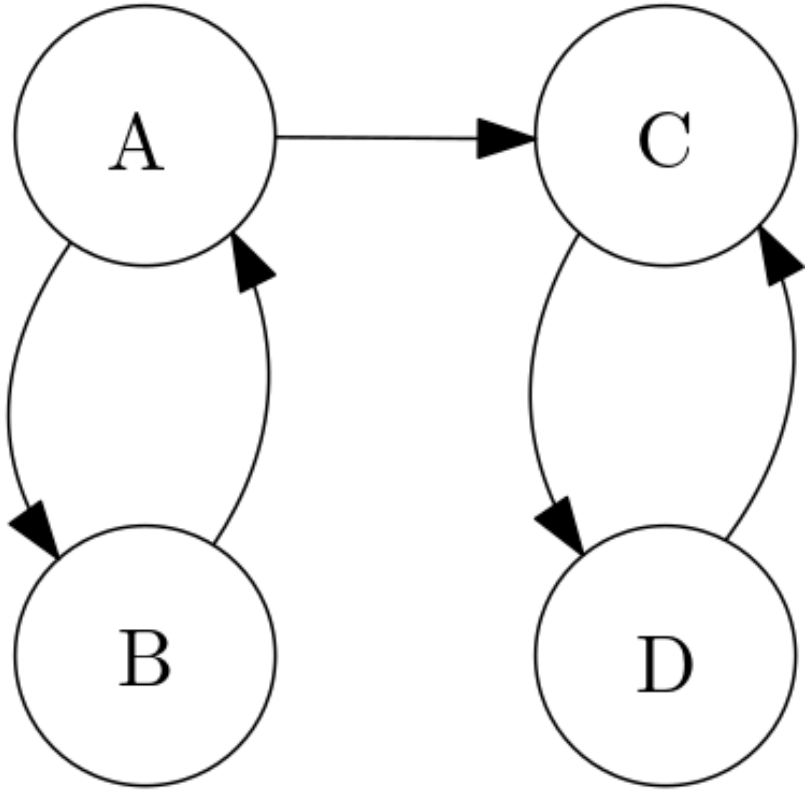
How can we decompose a graph to SCCs?

- Compute G^R
- Run DFS on G^R
- Find the connected components using explore, in decreasing order of post numbers

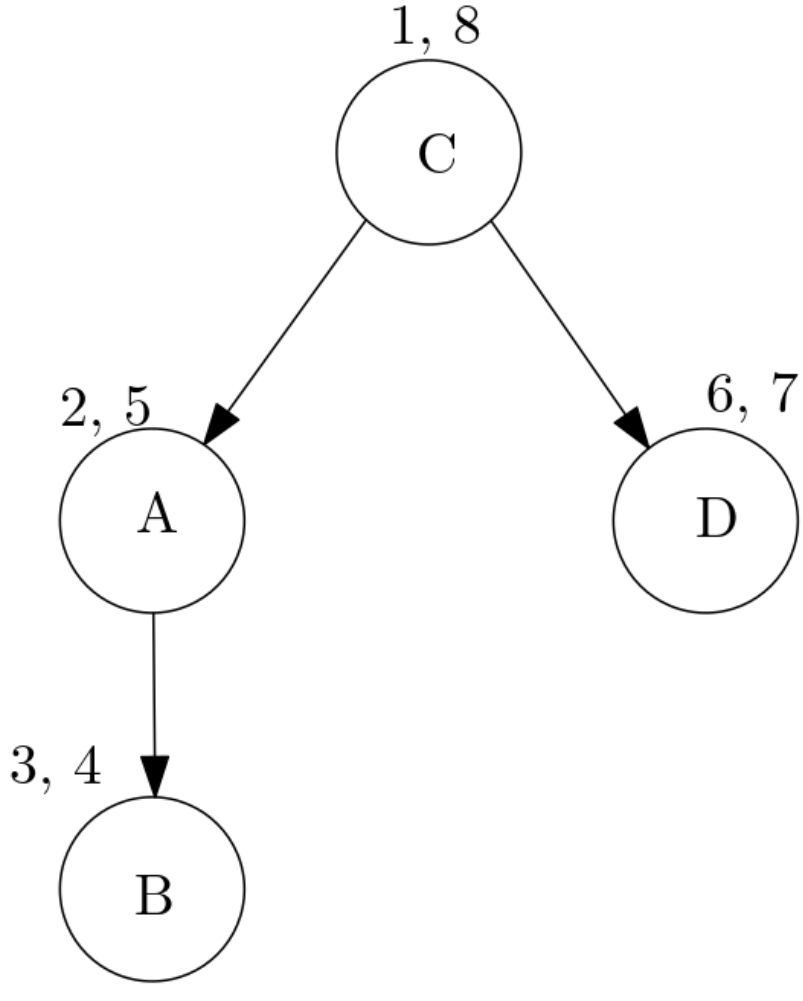
Suppose we have this graph:



Then G^R is:



And a DFS gives these post numbers:



Then we can separate this into SCCs $\{C, D\}$ and $\{A, B\}$: