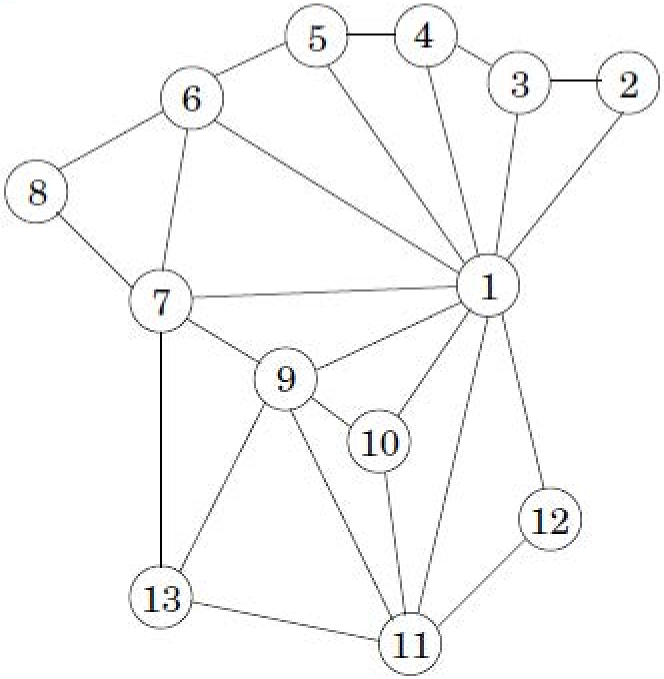
Graphs

Why graphs?

- A wide range of problems can be expressed as graphs.
- Ex) Coloring a map -> graph coloring

(a)

(b)



Graph coloring

- University needs to schedule examinations for all classes using fewest time slots possible.
- Constraint : two exams cannot be scheduled concurrently if a student takes both.

Graph

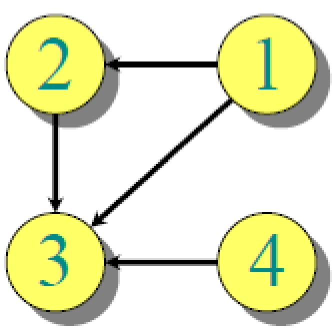
- A graph $G = (V, E)$ is specified by a set of *vertices* (also called *nodes*) V and by *edges* E between select pairs of vertices.
- Directed graphs (digraphs) vs. undirected graphs

  ex) a graph of all links in WWW
- In either case, $|E| = O(V^2)$.
- If G is connected, $|E| \geq |V| - 1$.
- If $|E|$ is close to $|V|$, we say that the graph is *sparse*.
- If $|E|$ is close to $|V|^2$, we call the graph *dense*.

Adjacency matrix

The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,.\,.\, n, 1 \,.\,.\, n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$

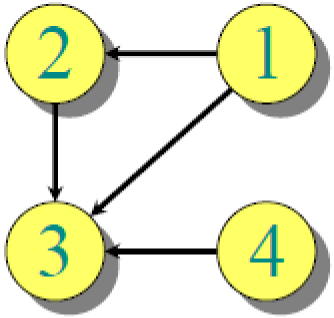


A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

$\Theta(V^2)$ storage $\Rightarrow$ ***dense*** representation.

Adjacency list

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

- $\Sigma_{v \in V}$ degree(v) = $2|E|$.
- $\Sigma_{v \in V}$ out-degree(v) = $|E|$.
- Adjacency lists use $\Theta(V+E)$ storage – a ***sparse*** representation.

Explore from a vertex

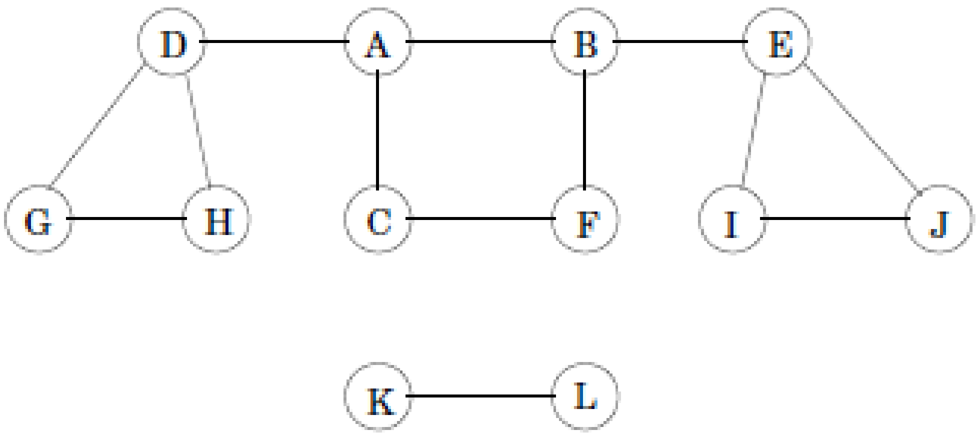
- What parts of the graph are reachable from a given vertex?

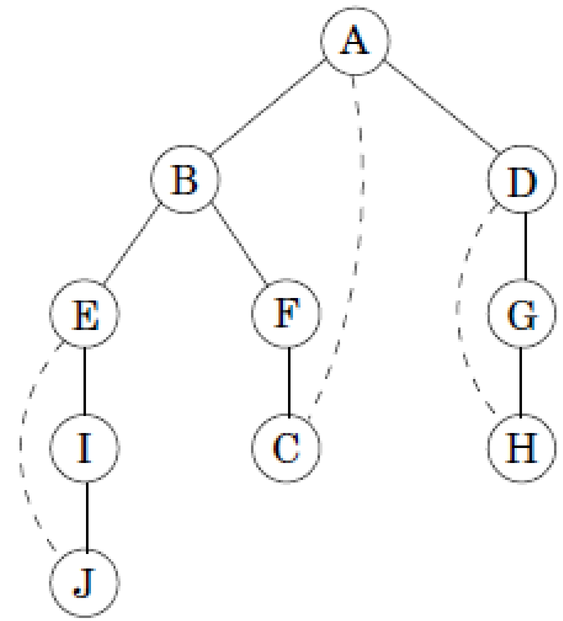
```
procedure explore(G, v)
Input:    G = (V, E) is a graph; v ∈ V
Output:   visited(u) is set to true for all nodes u reachable from v

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

- previsit(v), postvisit(v) : optional (to perform operations on a vertex v when it is first discovered and when it is being left for the last time)

Example of explore

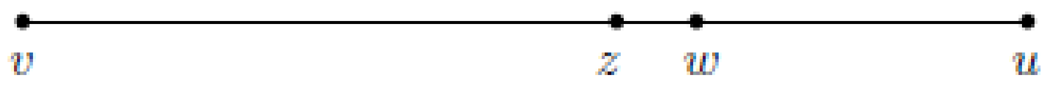




solid edges – tree edges
dotted edges – back edges

Correctness

- How to prove that it visits all vertices reachable from *v*?
    - Suppose there is a vertex *u* that it misses.



- Consider a path from *v* to *u* and call *z* the last vertex on this path that explore procedure visits.
- Let *w* be the vertex on this path immediately after *z*.
- *z* was visited but *w* was not visited.
- Contradiction!

Depth-first search

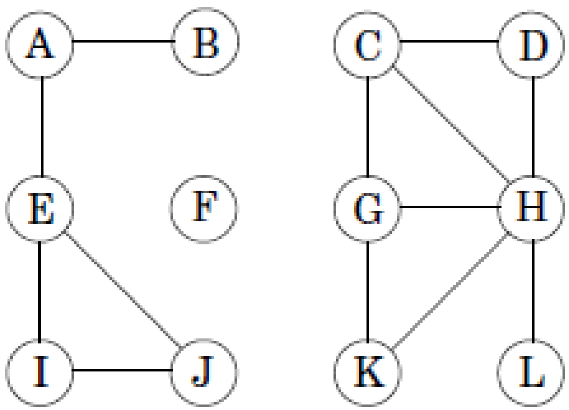
- *explore* procedure visits only the portion reachable from the starting point.
- Need to restart from unvisited vertices.

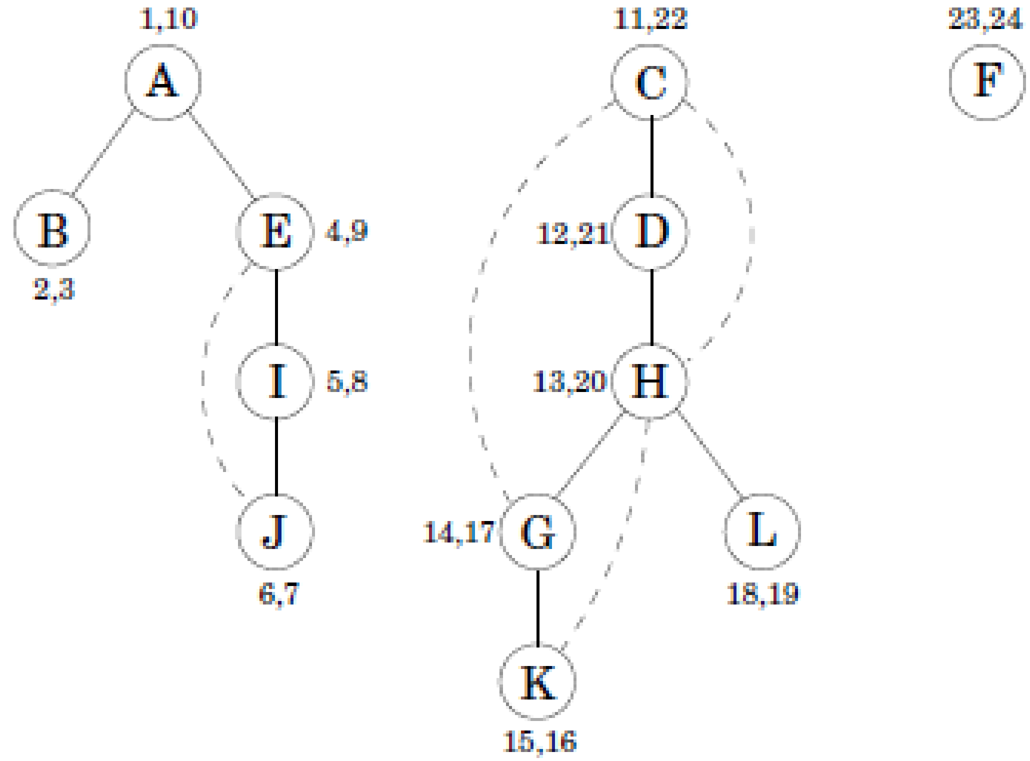
```
procedure dfs(G)

for all v ∈ V:
    visited(v) = false

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

Graph and *dfs forest*





Analysis

- Each vertex is explored just once thanks to the “visited”
  - For each vertex, we have to scan adjacent edges.

- Each edge (x, y) is examined exactly twice, once during explore(x) and once during explore(y).

- Thus, the overall running time for dfs is $O(|V| + |E|)$.

Connectivity in undirected graphs

- An undirected graph is ***connected*** if there is a path between any pair of vertices.
- We can adapt depth-first search to check if a graph is connected and to assign each node *v* an integer ccnum[*v*] identifying *the connected component* to which it belongs.

- Just add the following previsit procedure where cc is initialized to 0 and to be incremented each time the DFS calls explore.

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

Previsit and postvisit orderings

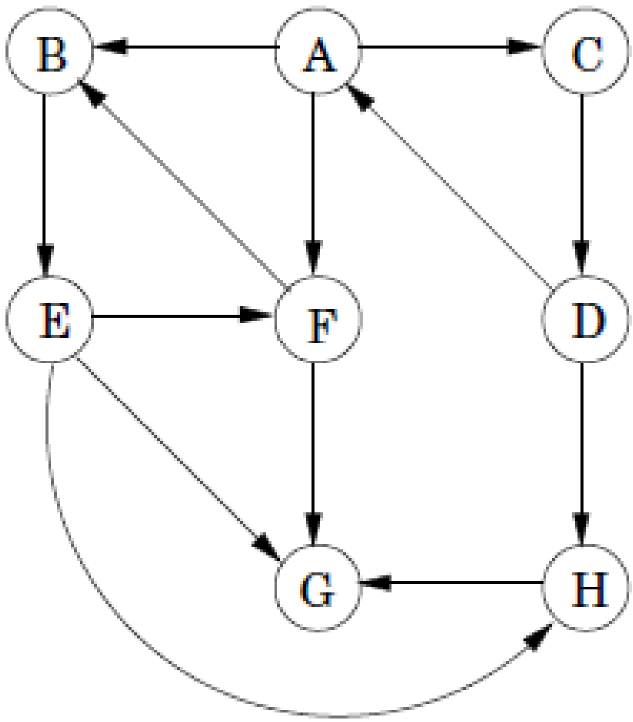
- For each vertex, record the time of first discovery and the time of final departure.
- Define a simple counter clock, initially set to 1.

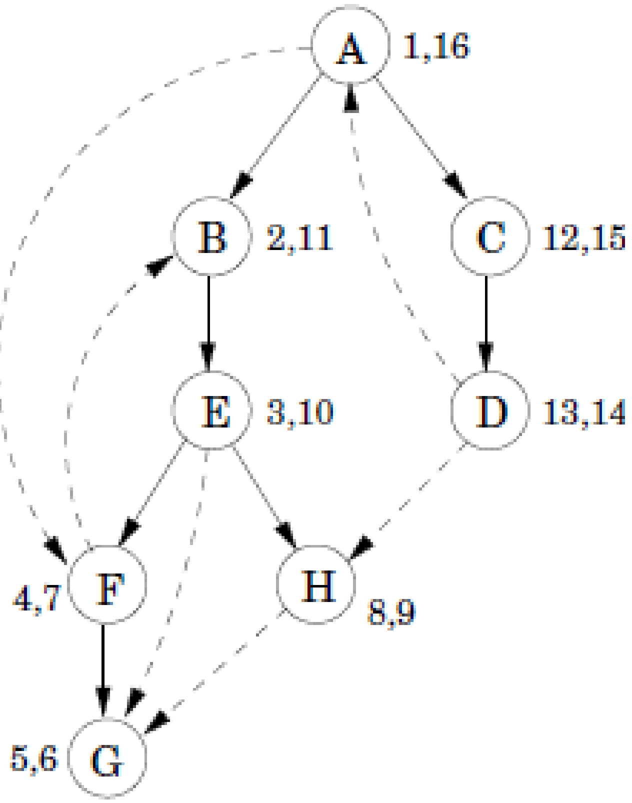
```
procedure previsit(v)
pre[v] = clock
clock = clock + 1

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

- For any nodes *u* and *v*, the two intervals [pre(*u*), post(*u*)] and [pre(*v*), post(*v*)] are either disjoint or one is contained within the other.

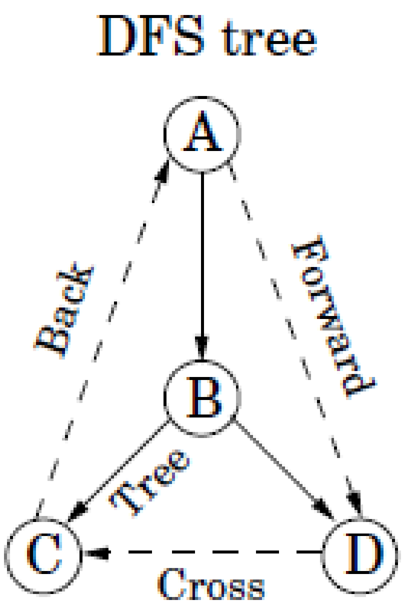
DFS in directed graphs





Types of edges

- Tree edges : edges of DFS forest
- Forward edges : from a node to a nonchild descendant in DFS tree
- Back edges : lead to an ancestor in DFS tree
- Cross edges : lead to neither descendant nor ancestor (lead to a node postvisited.)



Pre/post ordering and edge types

- *u* is an ancestor of *v* when *u* is discovered first and *v* is discovered during explore(*u*).
- pre(*u*) < pre(*v*) < post(*v*) < post(*u*)
- [[ ]]
- *u* *v* *v* *u*

pre/post *ordering for* (u, v)	*Edge type*
$\underset{u}{[}\quad\underset{v}{[}\qquad\underset{v}{]}\quad\underset{u}{]}$	Tree/forward
$\underset{v}{[}\quad\underset{u}{[}\qquad\underset{u}{]}\quad\underset{v}{]}$	Back
$\underset{v}{[}\qquad\underset{v}{]}\quad\underset{u}{[}\qquad\underset{u}{]}$	Cross

Directed acyclic graphs

- Directed acyclic graph (dag) is good for modeling relations like causalities, hierarchies, and temporal dependencies.
- A directed graph has a cycle if and only if its depth-first search reveals a back edge.
- Pf)
  - $\leftarrow$ if (u,v) is a back edge, then there is a cycle consisting of this edge together with the path from v to u in the search tree.
  - $\rightarrow$ if the graph has a cycle $v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$, look at the first node on this cycle to be discovered. Suppose it is v_i.

    All the other nodes in the cycle are reachable from it and will be its descendants in the search tree. Thus the edge to v_i in the cycle is a back edge.

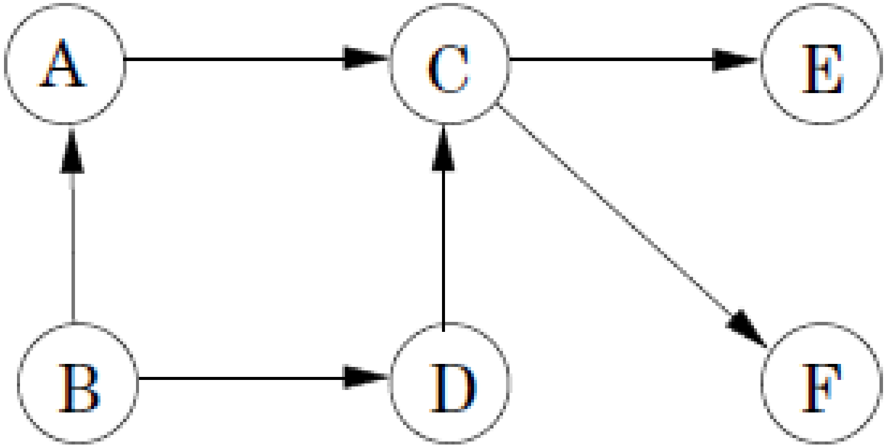
<u>Topological sort</u>

- Directed acyclic graph (dag) – a directed graph with no cycle.
- Good for modeling processes that have a ***partial order***.
  - $a > b$ and $b > c \Rightarrow a > c$
  - But may have a and b s.t. neither $a > b$ nor $b > a$.
- Can always make a ***total order*** from a partial order
- ***Topological sort*** of a dag : a linear ordering of vertices s.t. if $(u,v) \in E$, then u appears somewhere before v.

Topological sort

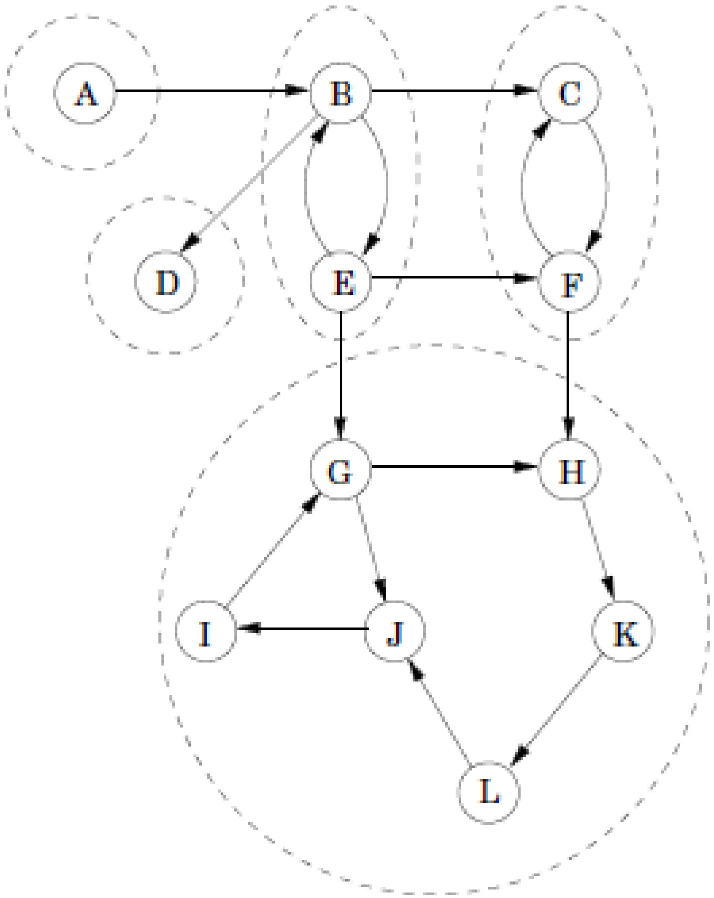
- Find a linear ordering of nodes where all edges are from earlier to later node.
- In a dag, every edge leads to a vertex with a lower post number (since there is no back edge.)
- Acyclicity, linearizability, and the absence of back edges during dfs are the same.
- We can linearize nodes by decreasing post numbers.
- Every dag has at least one source and at least one sink.
  - sink : the vertex with smallest post number
  - source : the vertex with highest post number

- A dag with one source, two sinks and 4 possible linearizations.

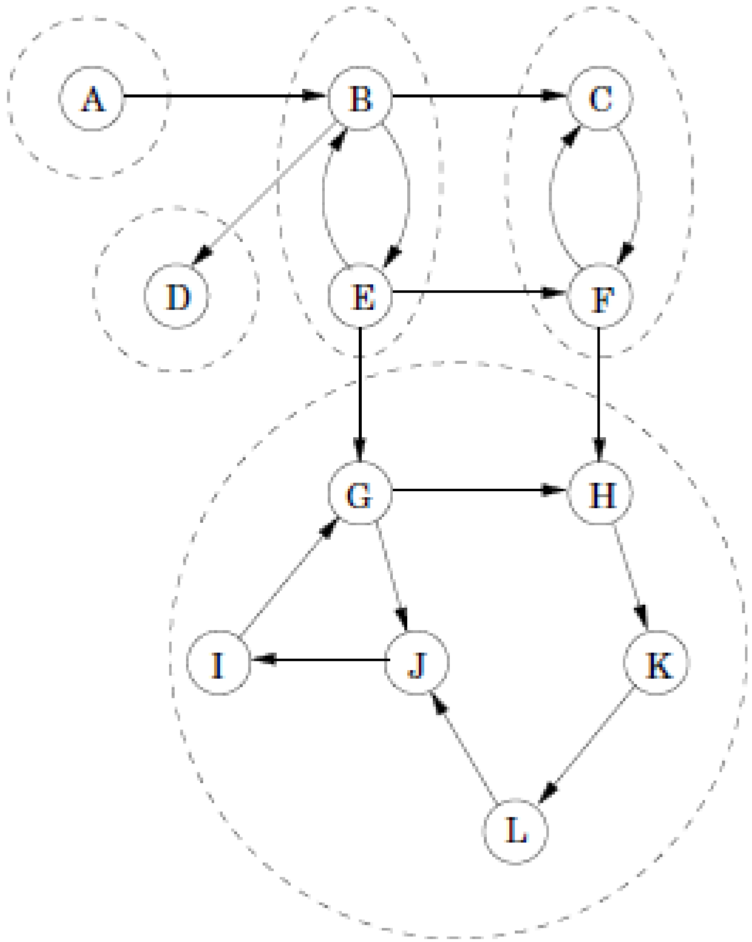


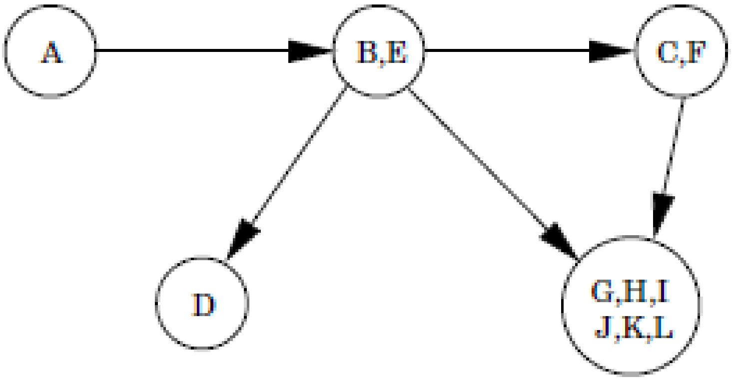
Strongly connected components

- Two nodes *u* and *v* of a directed graph are *connected* if there is a path from *u* to *v* and a path from *v* to *u*.
- This relation partitions *V* into disjoint sets called *strongly connected components (SCC)*.



Meta-graph





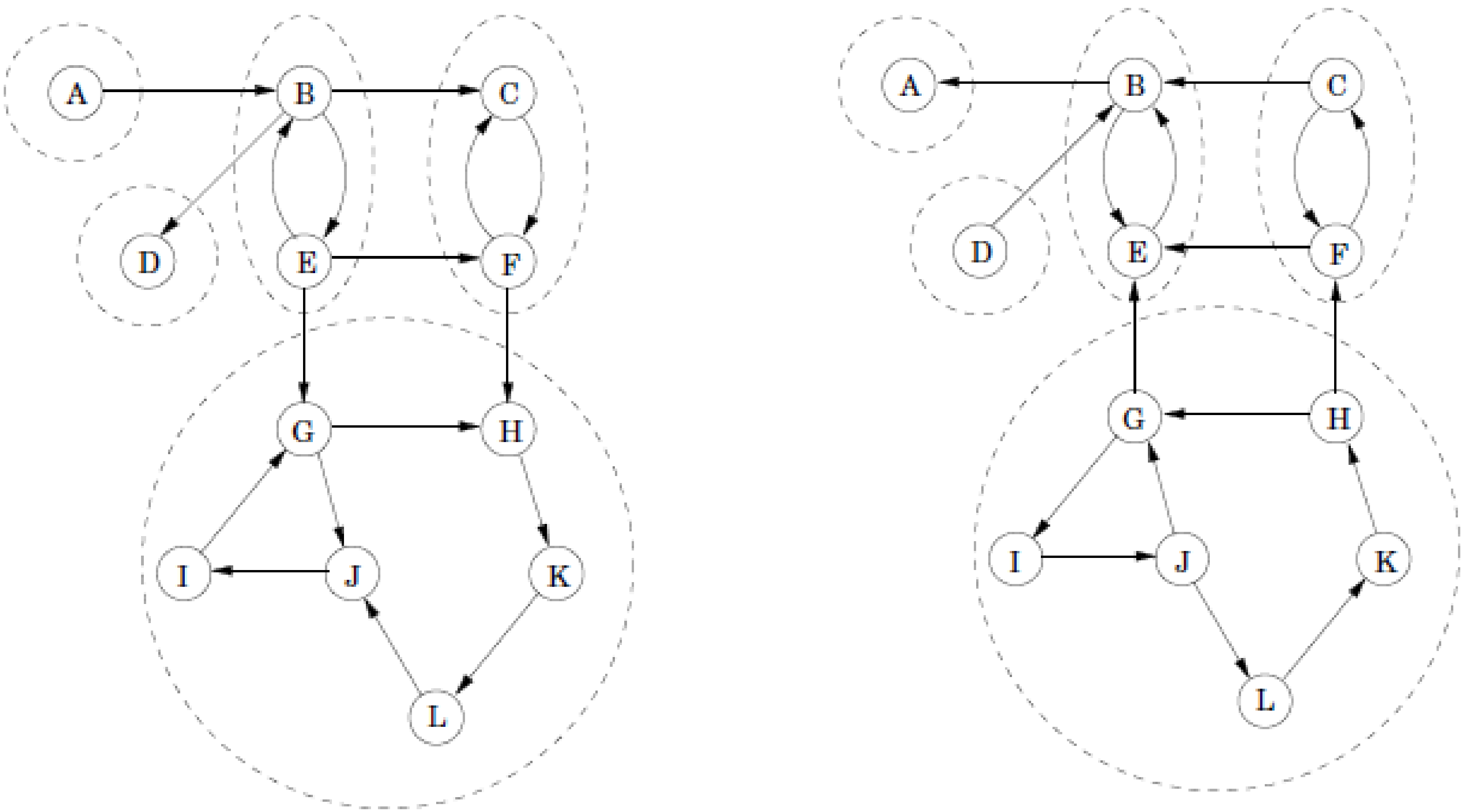
- ***Property 1*** : If the explore subroutine is started at node *u*, then it will terminate precisely when all nodes reachable from *u* have been visited.
- Thus, if we call explore on a node in a sink in the meta-graph, then we get the sink strongly connected component.
- (A) How do we find the sink?
- (B) How do we continue after the first (sink) component is found?

- ***Property 2*** : The node that receives the highest post number in a depth-first search must lie in a source strongly connected component.

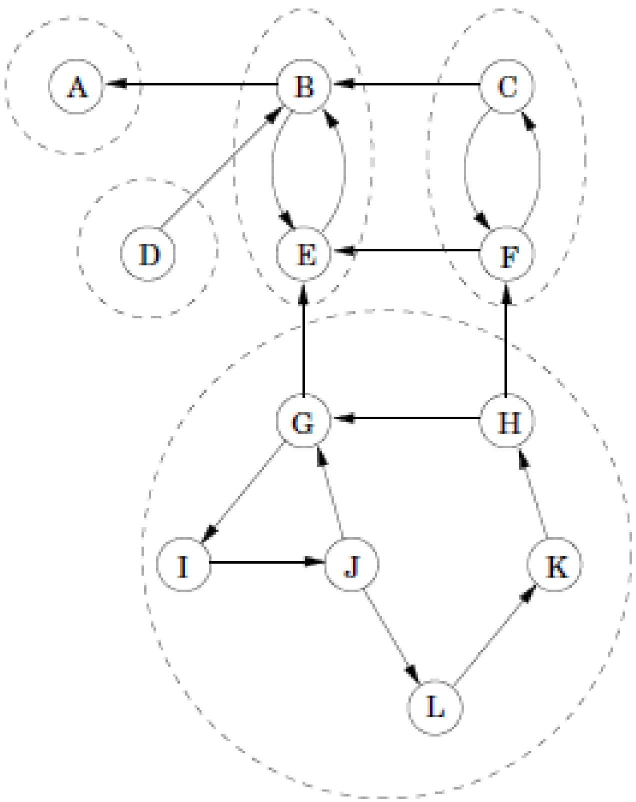
- ***Property 3*** : If *C* and *C'* are strongly connected components, and there is an edge from a node in *C* to a node in *C'*, then the highest post number in *C* is bigger than the highest post number in *C'*.
- Pf)
  - case 1 : dfs visits *C* before *C'*
  - case 2 : dfs visits *C'* before *C*
- Thus, we can linearize SCC in decreasing order of their highest post numbers.

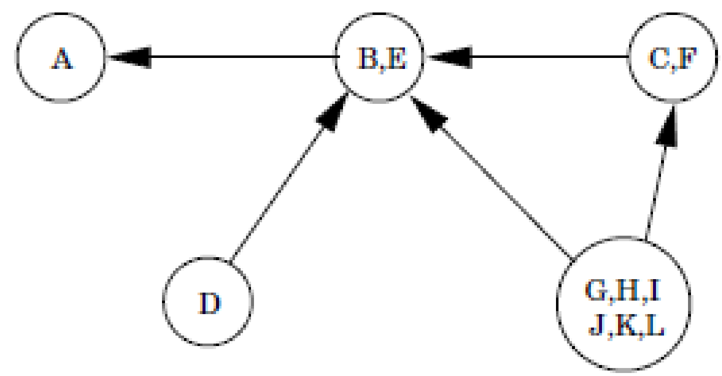
Reverse graph

- Reverse graph G^R of G : all edges of G reversed
- Observation : G and G^R have the same SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^R)



Reverse graph and its meta-graph





Algorithm

1. Run depth-first search on G^R
2. Run depth-first search on G (to obtain connected component) processing vertices in decreasing order of their post numbers from step 1.

Running time?