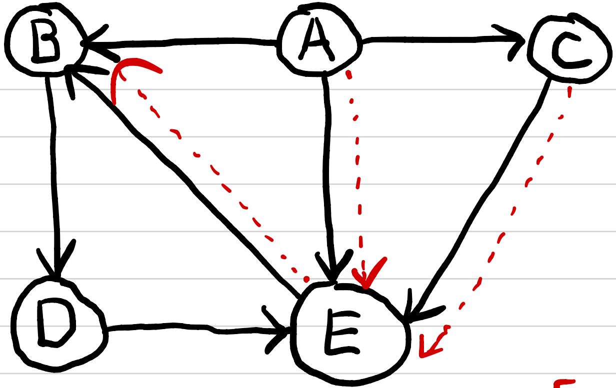
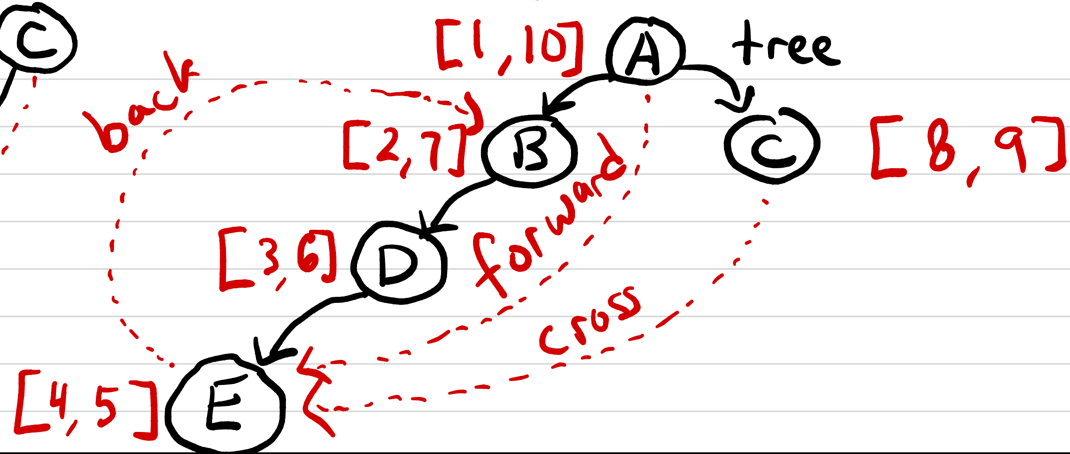
# Directed graphs




DFS search tree



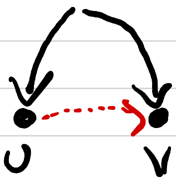

```
explore(G,u)
    visited[u] = true

    pre[u] = clock
    clock = clock+1

    for v s.t. (u,v) ∈ E
        if visited[v] = false
            explore(G,v)
    post[u] = clock
    clock = clock+1
```

```
dfs(G)
    boolean array visited[n]
        (init to all 0's)
    clock = 1

    int array pre[n], post[n]

    for v ∈ V
        if visited[v] = false
            explore(G,v)
```

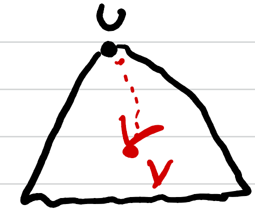
# Classifying edges using pre and post #s

Say $(u, v) \in E$

Clock: $\overset{pre}{[_u}\ \overset{pre}{[_v}\ \overset{post}{]_u}\ \overset{post}{]_v}$

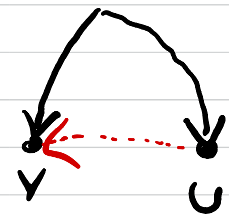



$[_u\ [_v\ ]_v\ ]_u$ : tree, forward




$[\ ]_u\ [\ ]_v$ : impossible

$[\ ]_v\ [\ ]_u$ : cross




$[_v\ [_u\ ]_u\ ]_v$ : back

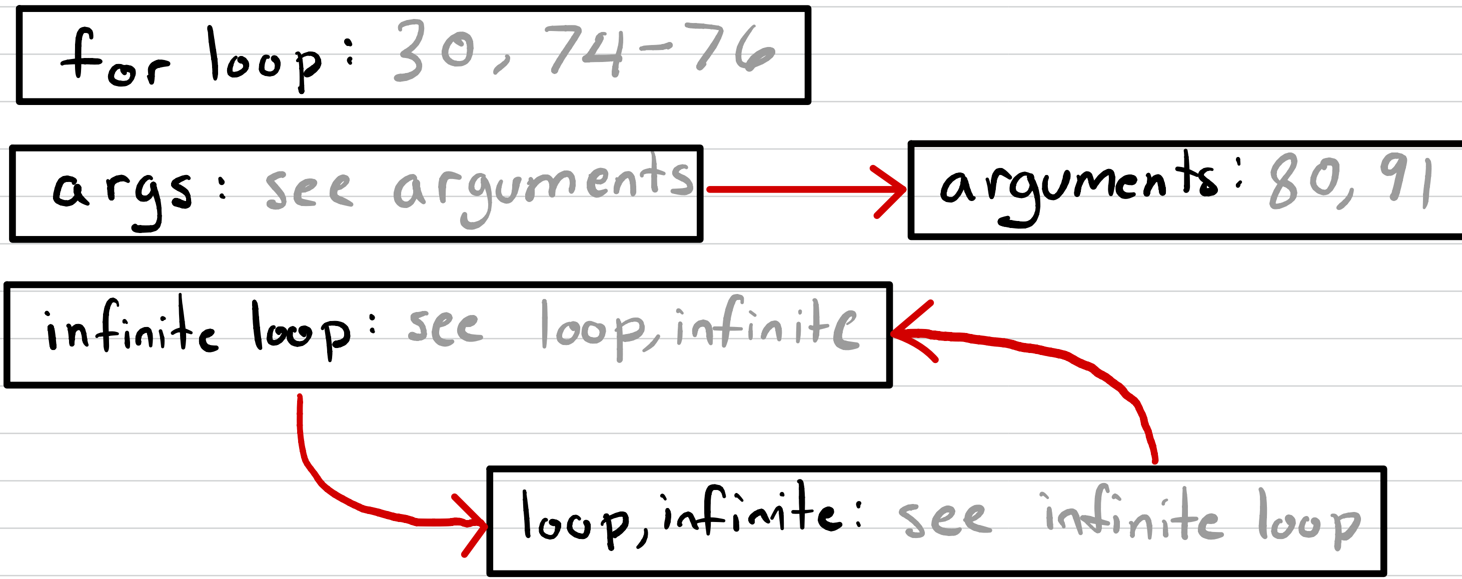

$[_u\ [_v\ ]_u\ ]_v$ : impossible

Fact: $(u,v)$ is back edge

$\iff \text{post}(u) < \text{post}(v)$

# Application # 1: Cycle detection

Book index:

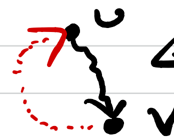



Q: Does my graph have a cycle?

**Def:** A directed acyclic graph (DAG) is a directed graph w/ no cycles.

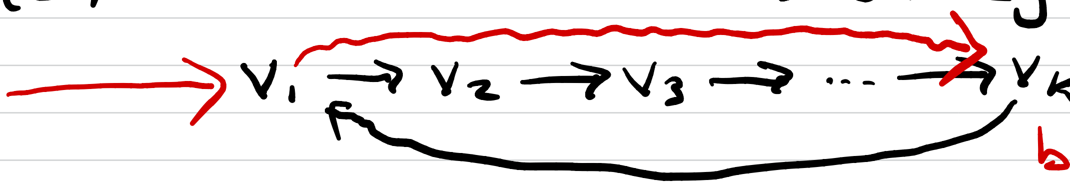
**Claim:** Suppose we run DFS on $G$. Then $G$ is a DAG iff no back edges.

**Pf:** (1) If back edge $\Rightarrow$ $G$ is not a DAG.




$\leftarrow$ a cycle!

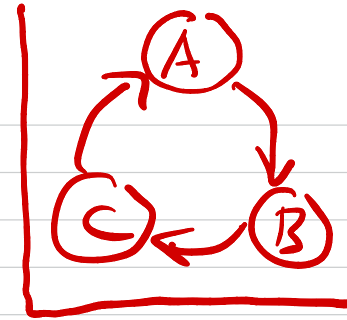
(2) $G$ is not a DAG $\Rightarrow$ back edge

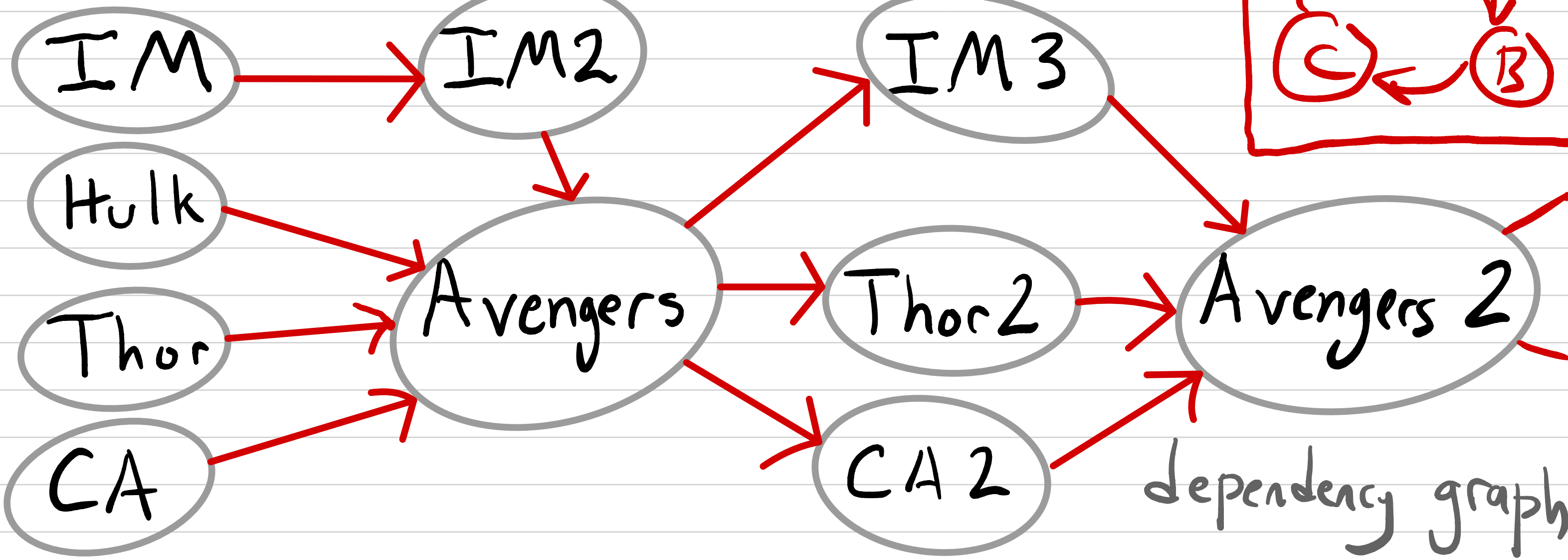



back edge!

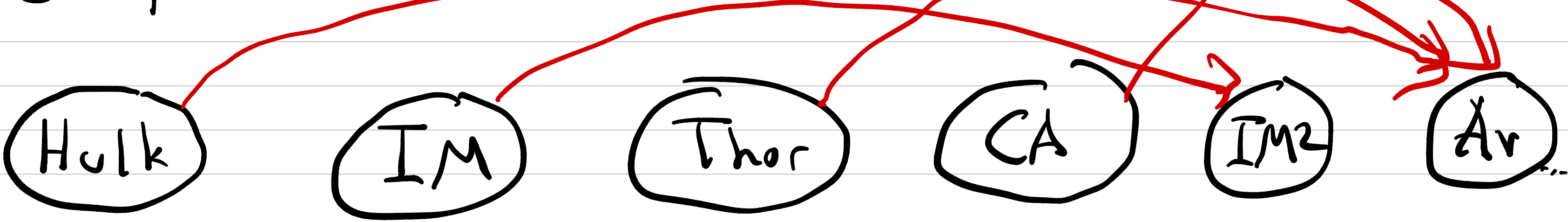
**Cycle detection alg:** Run DFS.
Output "DAG" if no back edges.
($\forall (u,v) \in E$, check $post(u) > post(v)$)

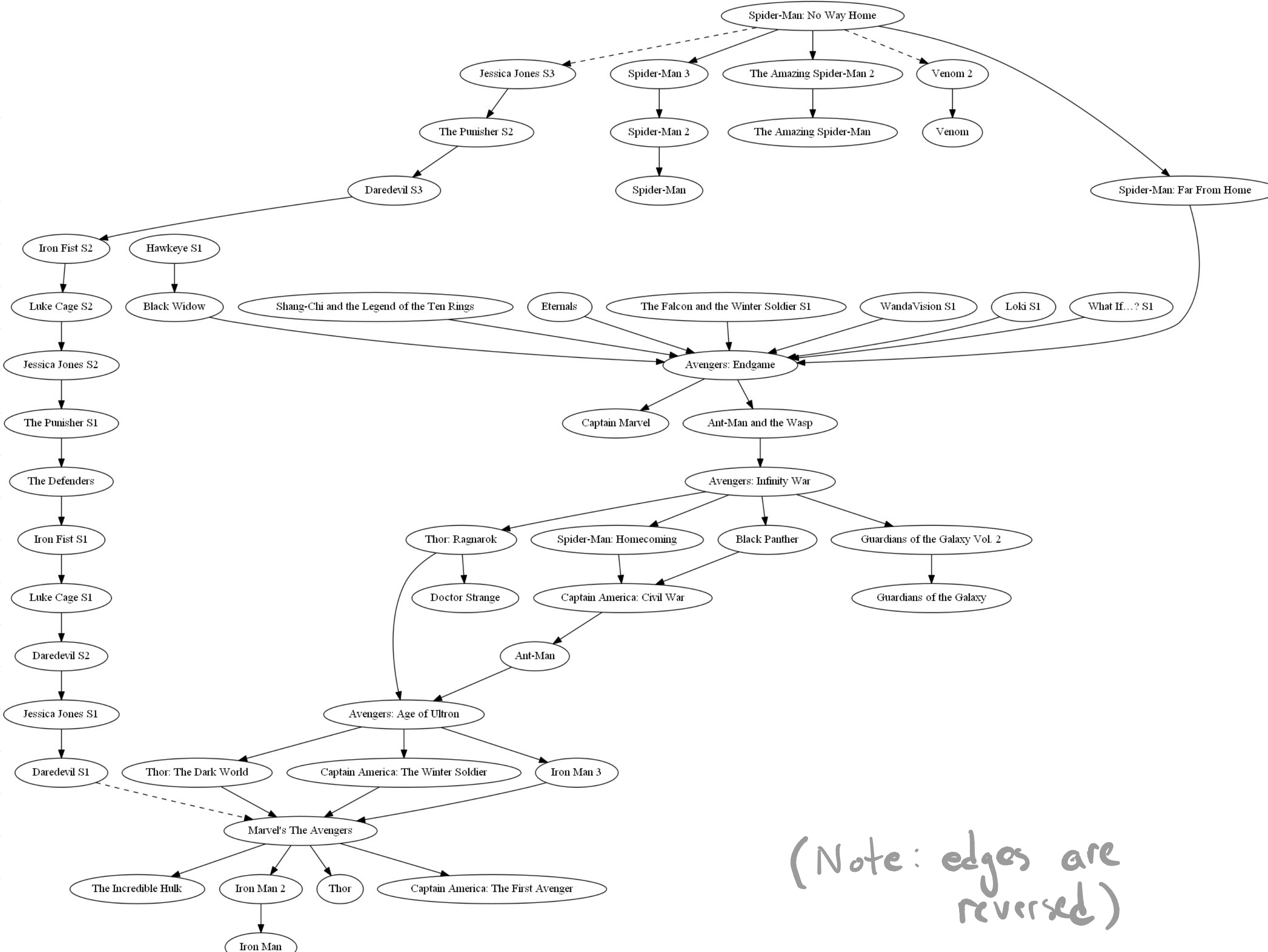
# Application #2: Topological Sort




Output






(Note: edges are reversed)

# Topological sort

**Input:** DAG $G=(V,E)$
**Output:** Ordering of vertices $v_1, \dots, v_n$ s.t. all edges go left $\longrightarrow$ right

**Claim:** Suppose we run DFS on $G$. Then for all $(u,v) \in E$, $\text{post}(u) > \text{post}(v)$.

**Pf:** $G$ is a DAG $\Rightarrow$ no back edges
$\Rightarrow \text{post}(u) > \text{post}(v)$ for all edges. $\square$

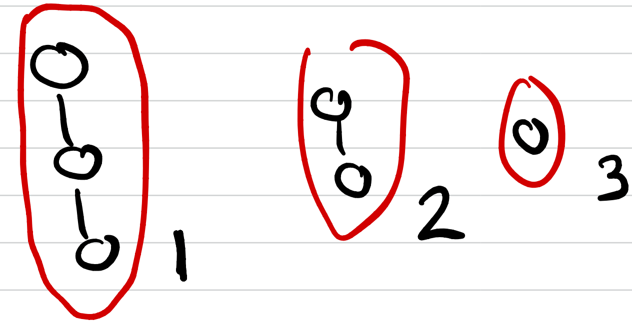
**Alg:** Run DFS on $G$.
Sort vertices from highest post number to lowest.

**Output:**

$$\overset{\text{post } 12}{u} \longrightarrow \overset{\text{post } 11}{v} \;\cancel{\longleftarrow}\; \overset{\text{post } 9}{w} \quad \dots$$

# Connected components

Undirected




Directed


(Connected?)

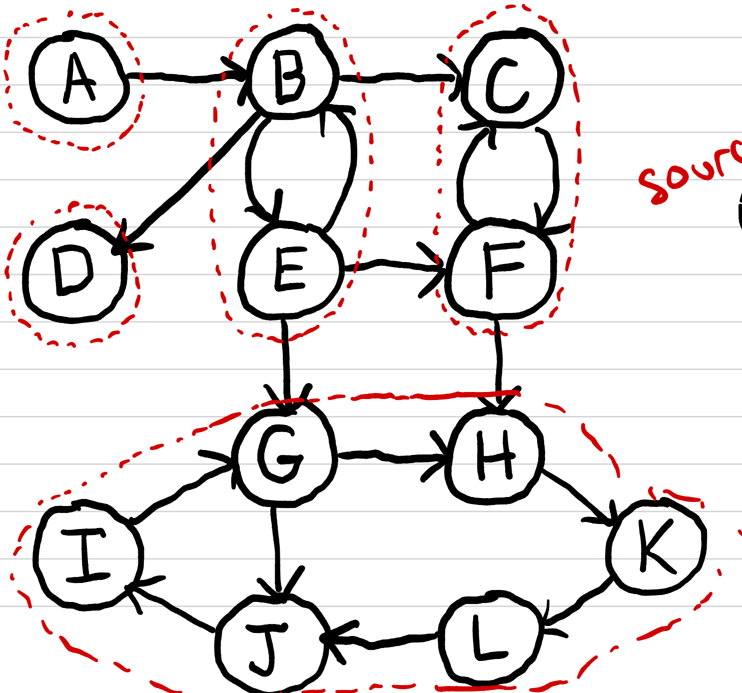
Questions:

1. How do we define connected components in digraphs?
2. How do we compute them?

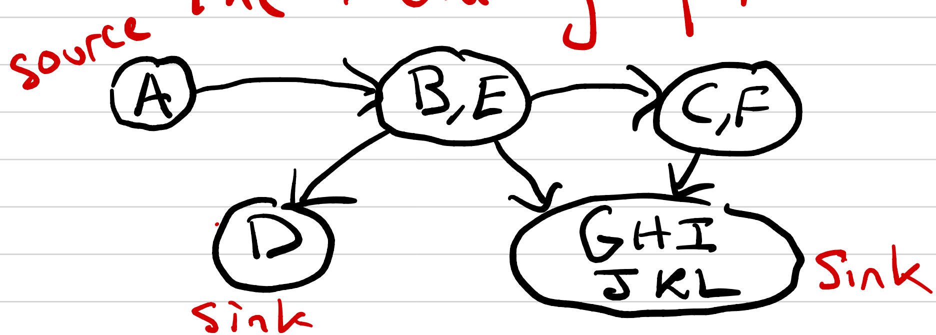
# Strongly connected components (SCCs)

Def: Vertices u and v are strongly connected if there is a path from u to v and from v to u

Claim: $u \sim v$ is an equivalence relation (i) reflexive
(ii) symmetric
(iii) transitive



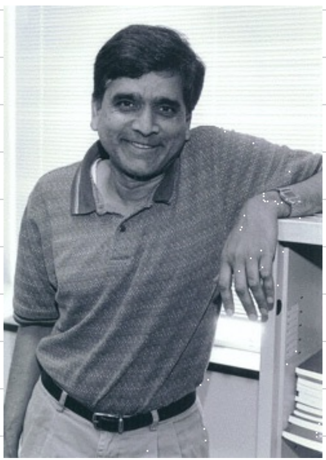

## The Meta-graph




Claim: The meta-graph is a DAG

Today: Algorithm to compute SCCs

Kosaraju

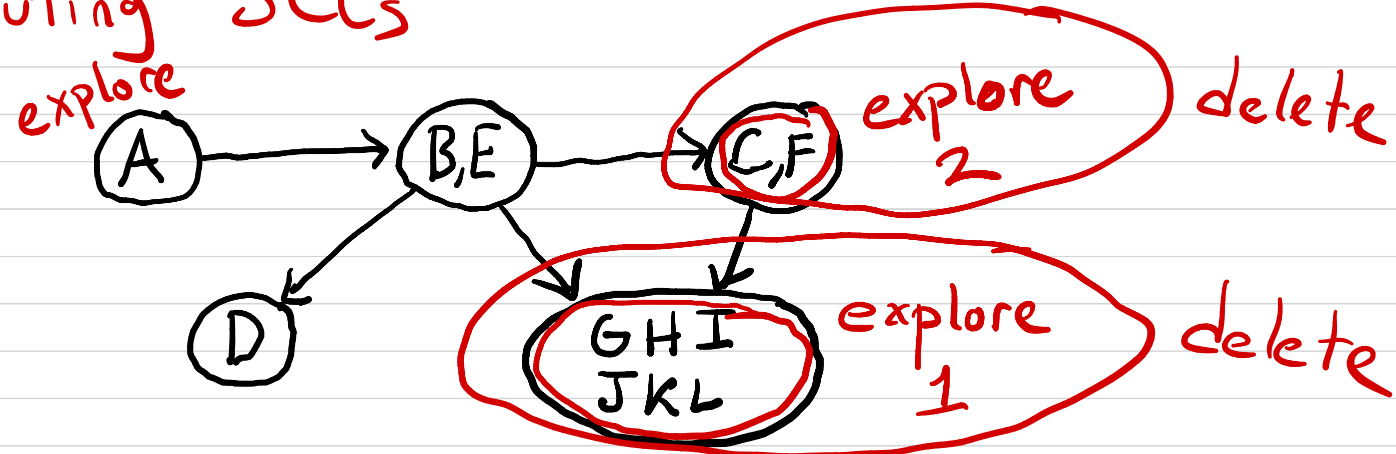


1978

Sharir

1981

# Computing SCCs




Suppose we had a magic algorithm which could tell us a vertex $v$ <u>in a sink SCC</u>

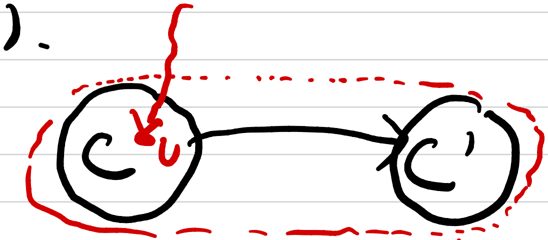
If we explore starting at $v$,
we explore all vertices in its SCC

Magic algorithm: DFS! (with a small twist...)

Suppose we run DFS.
For all SCCs $C$, define finish($C$) = $C$'s highest post($v$)

Claim: Let $C \rightarrow C'$ be SCCs.
Then finish($C$) > finish($C'$).

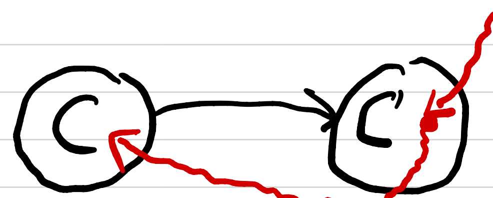
Pf: (i) Suppose DFS visits $C$ first.
Then post($u$) > finish($C'$).




u explores these

(ii) Suppose DFS visits $C'$ first.
Then only visits $C$ after done w/ $C'$.

∴ all posts in $C$
> finish($C'$)




Can't happen
Graph is DAG.

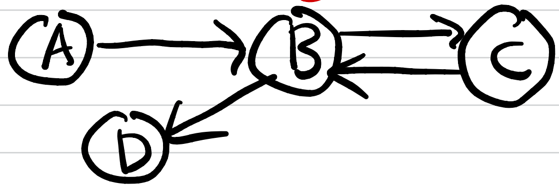
Claim: The highest post($u$) is in source SCC.
Assume not.


even bigger

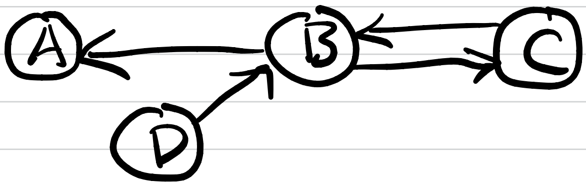
highest post

# The reverse graph

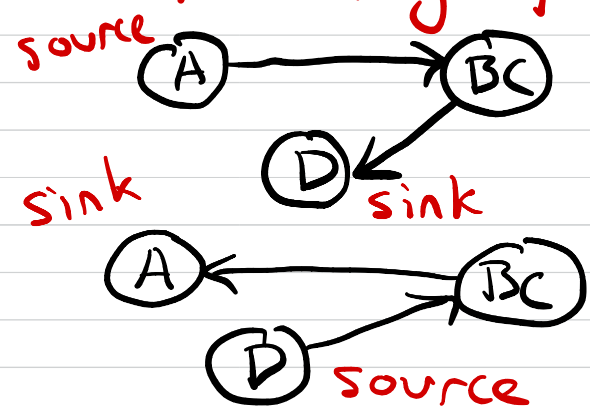
$G$




$G^R$




# Meta-graph

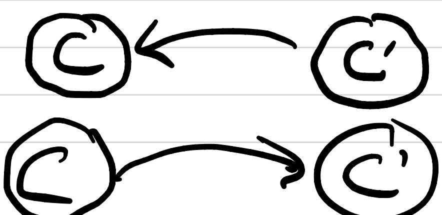



Claim: $G$ and $G^R$ have same SCCs.
In meta graphs: • edges are reversed
• sources and sinks are swapped

Run DFS on $G^R$. Compute $post_R$ values.

$u$ w/ highest $post_R$: • (in $G^R$) in source
• (in $G$) in sink

If



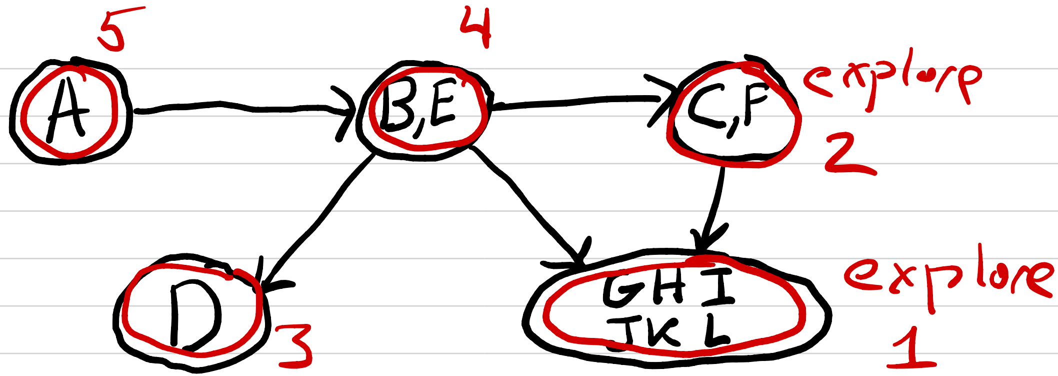

(in $G^R$)

(in $G^R$)

then highest $post_R$ in $C'$ > in $C$

# SCC algorithm

If $C \to C'$
highest $post_R$
in $C'$




explore(G, u)
  visited[u] = true
  sccnum[u] = count

  for v s.t. $(u,v) \in E$
    if visited[v] = false
      explore(G, v)

FindSCC(G)
  for all u, visited[u] = false
  Run DFS on $G^R$
    to compute $post_R$
  Count = 1
  for $u \in V$ (in reverse $post_R$ order)
    if visited[u] = false
      explore(G, u)
      count = count + 1.