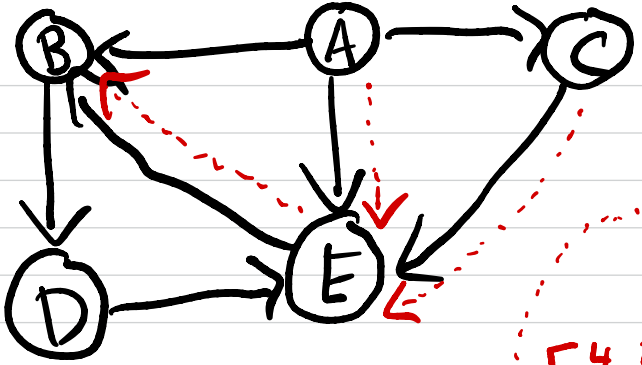
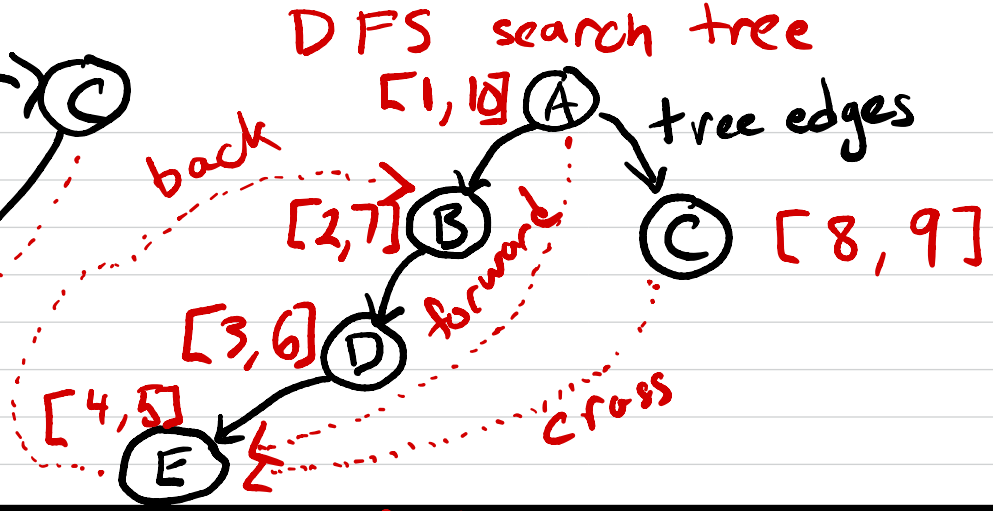
Directed graphs






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

  pre[v] = clock
  clock = clock+1

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

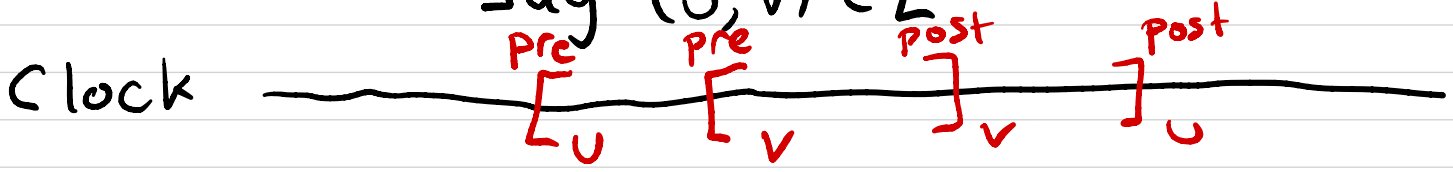
```
dfs(G)
  boolean array visited[n]
      (init to all 0's)
  clock = 1

  int array pre[n], post[n]

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

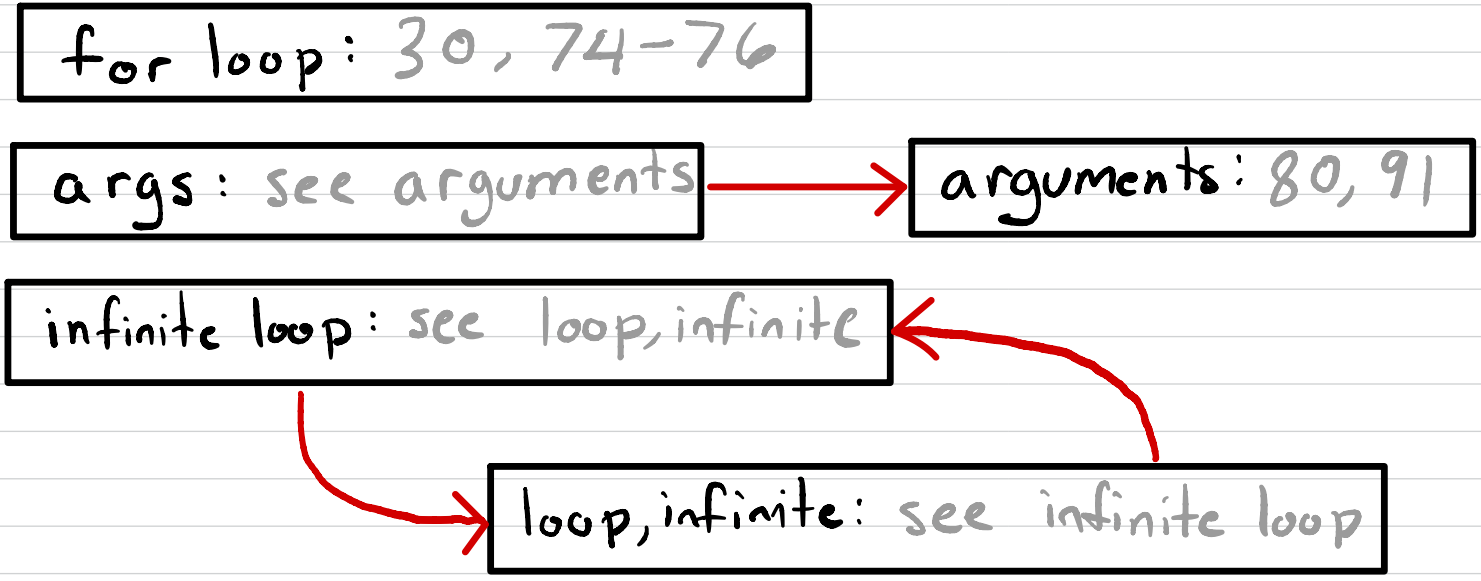
Classifying edges using pre & post #'s

Say $(u, v) \in E$



Application #1: Cycle detection

Book index:

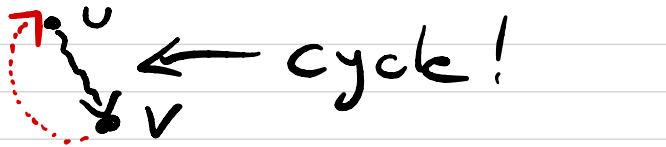


Q: Does my graph have a cycle?

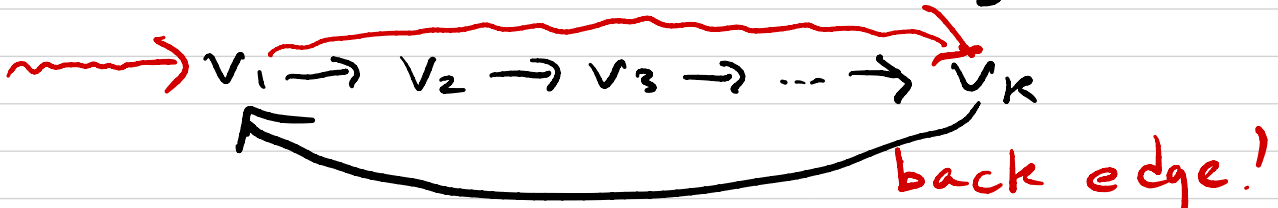
Def: A directed acyclic graph (DAG) is a directed graph w/ no cycles.

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.

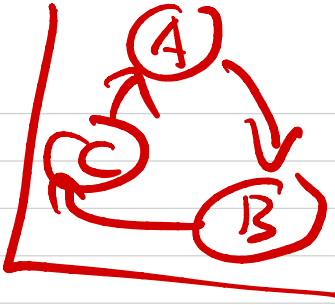


(2) G is not a DAG $\Rightarrow$ back edge



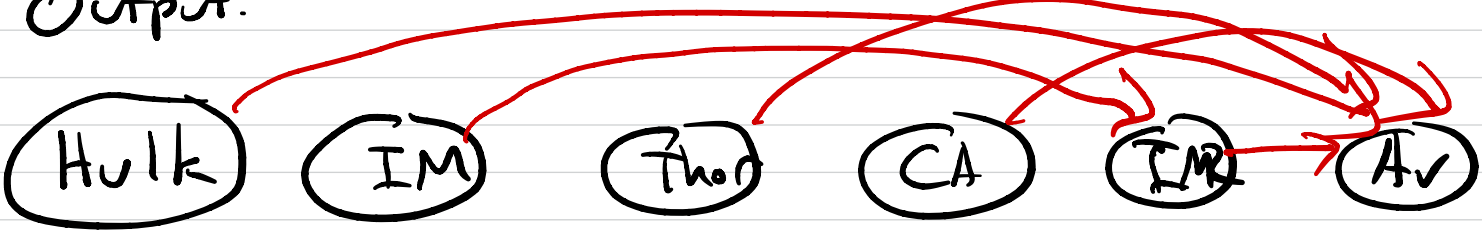
Cycle detection alg: Run DFS.
Output "DAG" if no back edges
($\forall (u,v) \in E$, check post(u) > post(v))

Application #2: Topological Sort



dependency graph

Output:




Spider-Man: No Way Home
Jessica Jones S3
Spider-Man 3
The Amazing Spider-Man 2
Venom 2
The Punisher S2
Spider-Man 2
The Amazing Spider-Man
Venom
Daredevil S3
Spider-Man
Spider-Man: Far From Home
Iron Fist S2
Hawkeye S1
Luke Cage S2
Black Widow
Shang-Chi and the Legend of the Ten Rings
Eternals
The Falcon and the Winter Soldier S1
WandaVision S1
Loki S1
What If…? S1
Jessica Jones S2
Avengers: Endgame
The Punisher S1
Captain Marvel
Ant-Man and the Wasp
The Defenders
Avengers: Infinity War
Iron Fist S1
Thor: Ragnarok
Spider-Man: Homecoming
Black Panther
Guardians of the Galaxy Vol. 2
Luke Cage S1
Doctor Strange
Captain America: Civil War
Guardians of the Galaxy
Daredevil S2
Ant-Man
Jessica Jones S1
Avengers: Age of Ultron
Daredevil S1
Thor: The Dark World
Captain America: The Winter Soldier
Iron Man 3
Marvel's The Avengers
The Incredible Hulk
Iron Man 2
Thor
Captain America: The First Avenger
Iron Man
(Note: edges are reversed)

Topological sort

Input: DAG $G=(V,E)$

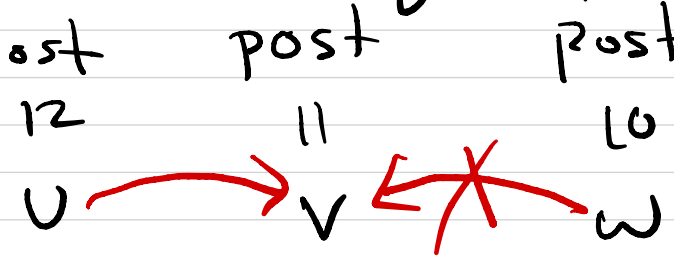
Output: Ordering of vertices $v_1, \ldots, v_n$ s.t. all edges go left $\rightarrow$ right

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

Pf: G is a DAG $\Rightarrow$ no back edges $\Rightarrow$ $post(u) > post(v)$ for all edges. □

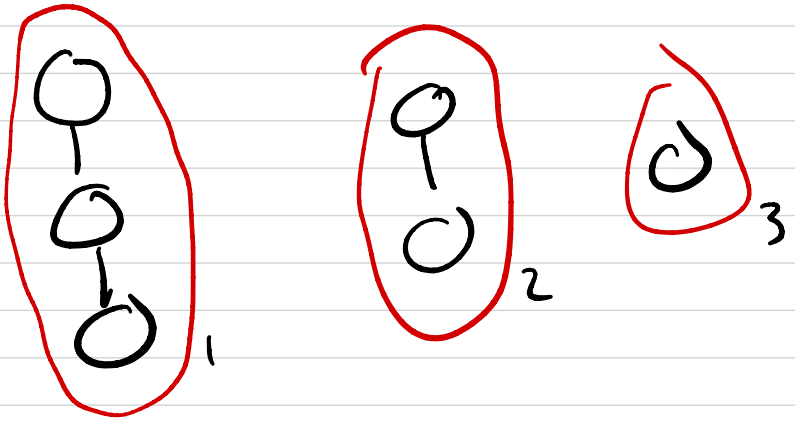
Alg: Run DFS on G. Sort vertices from highest post to lowest.

Output:



Connected components

Undirected:



Directed:

(Connected?)

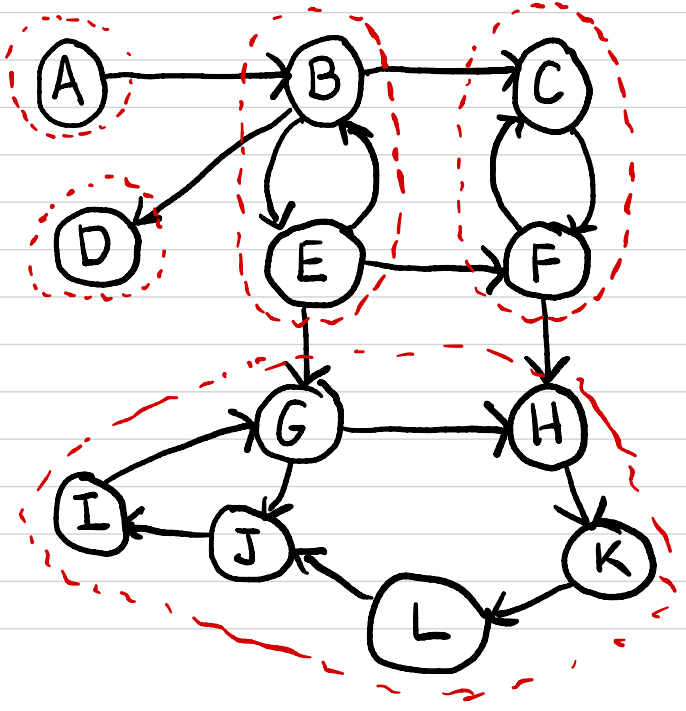
Questions:

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

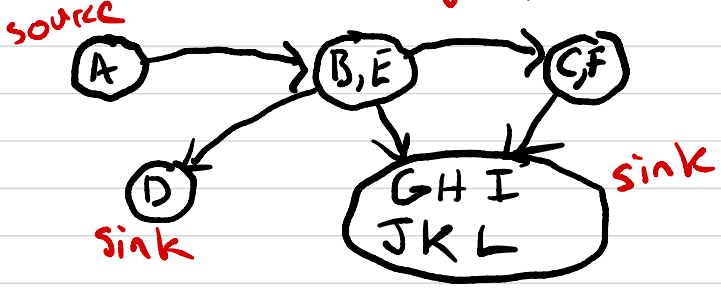
Strongly Connected Components (SCCs)

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

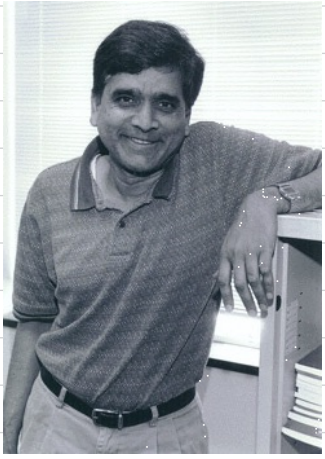


The Meta-graph



Today: Algorithm to compute SCCs

Kosaraju

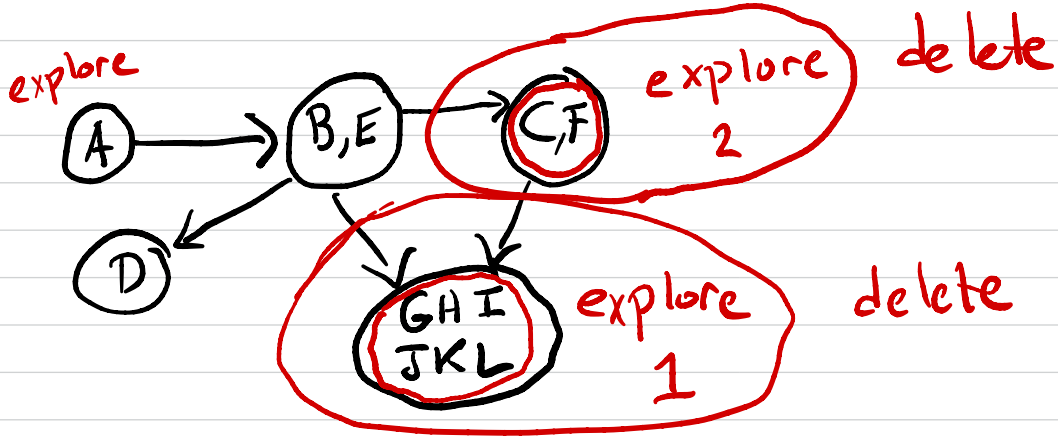


1978

Sharir

1981

Computing SCCs



Suppose we had a magic algorithm which could tell us a vertex v in a sink SCC

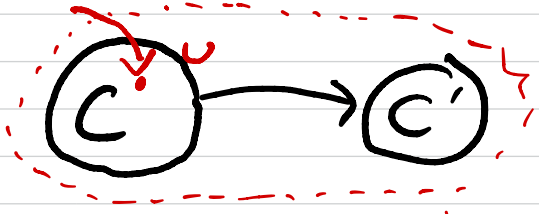
If we explore starting at v,
we explore all vertices in that SCC

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

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

Claim: Let C → C' be SCC's.
Then finish(C) > finish(C').

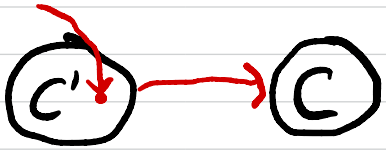
Pf: (i) Suppose DFS visits C first.
Then post(v) > finish(C').



v explores these

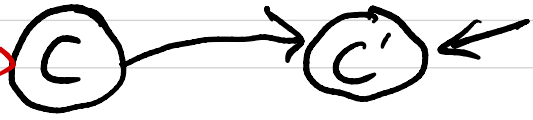
(ii) Suppose DFS visits C' first.
Then only visits C after done w/ C'.

∴ all posts in C > finish(C').

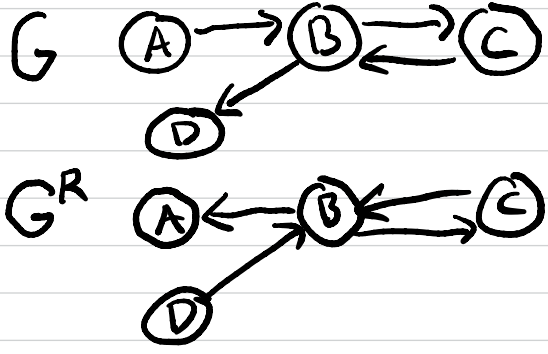


Can't happen!
Graph is DAG.

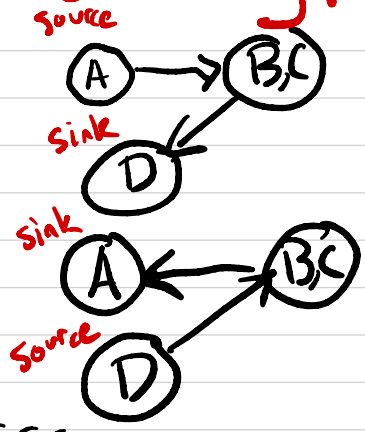
Claim: The highest post(v) is in source SCC.
Assume not.



The reverse graph



Meta graph



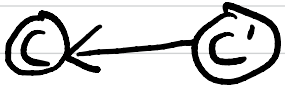
Claim: G and G^R have same SCCs.
In meta graphs:
- edges are reversed
- sources and sinks are swapped

Run DFS on G^R. Compute $post_R$ values.
∪ w/ highest $post_R$:
- (in G^R) in source SCC
- (in G) in sink SCC

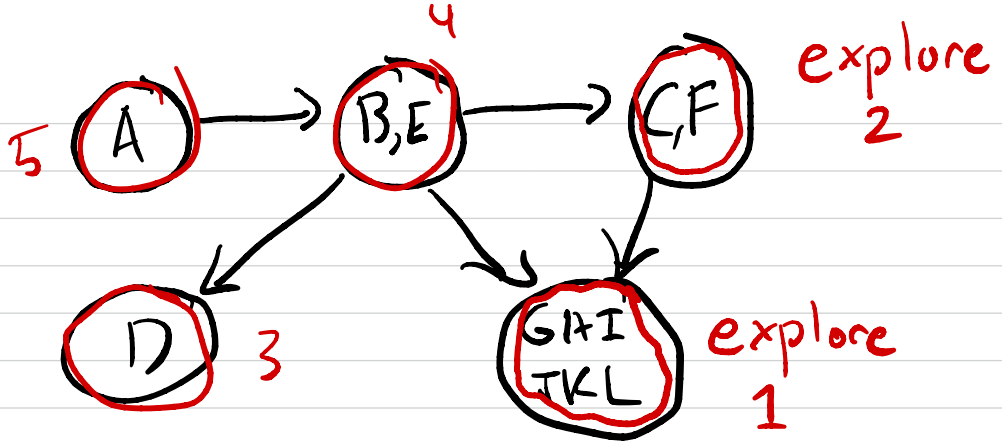
If  (in G^R) then highest $post_R$ in C > in C'

 (in G)

SCC algorithm



highest $post_R(u)$ in C
> in C'



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

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

```
Find SCCs (G)
    for all u, visited[u] = false

    Run DFS on G^R
        to compute post_R

    count = 1
    for u ∈ V (in reverse post_R order)
        if visited[u] = false
            explore (G, u)
            count = count + 1
```