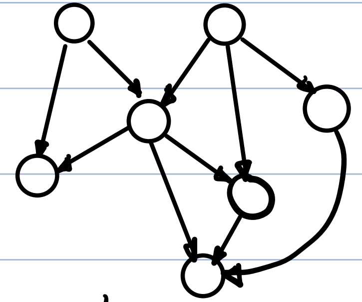
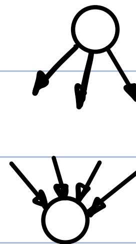
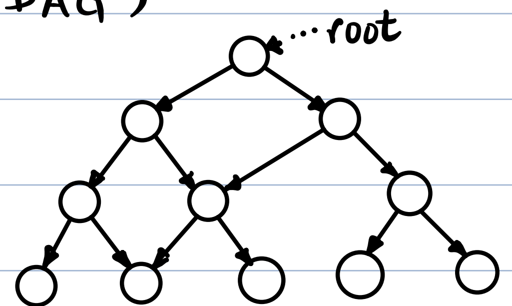
CMSC 451 - Algorithm Design
Lecture 3 - Cycles + Strong Components



Directed Acyclic Graphs (DAG):

- Arises in:
  - scheduling precedence constraints
  - data structures (rooted DAG)
  - ...
- Source - in-degree = 0
- Sink - out-degree = 0



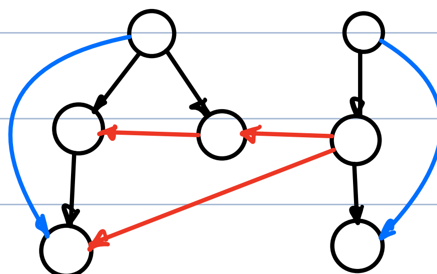


Acyclicity Test:

Given a directed graph, does it have a cycle?

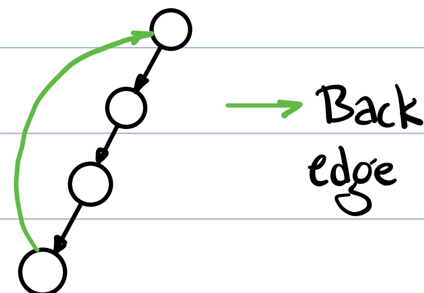
Ideas:

- Repeatedly delete sinks (or sources)
- DFS? How to detect cycles?

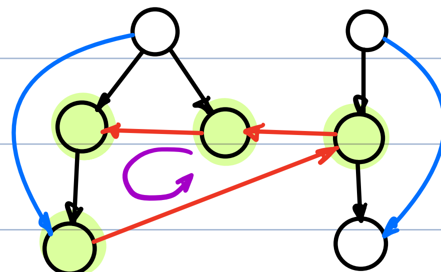


No cycles!

Back edge ⇒ cycle

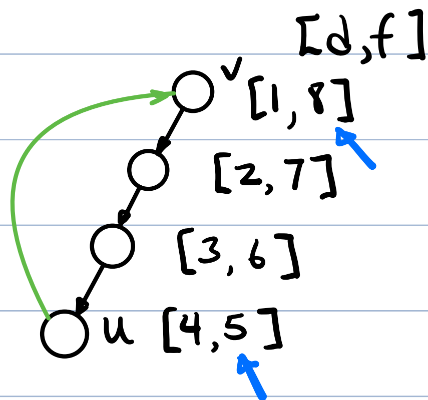


Huh? Can't cross edges create cycles?



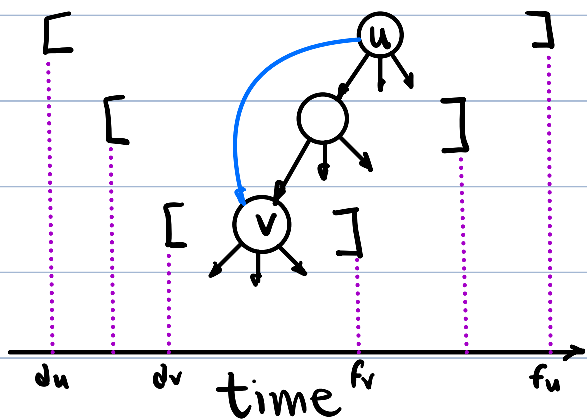
No! Try DFS on this graph + you'll see why

How to detect a back edge?



Claim:
(u,v) is back edge iff $f[u] \leq f[v]$

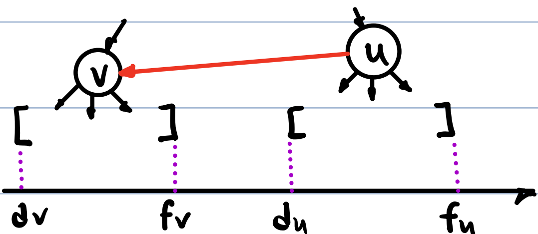
why "="? (self-loop)



If (u,v) is tree or forward

$[d[v], f[v]] \subset [d[u], f[u]]$

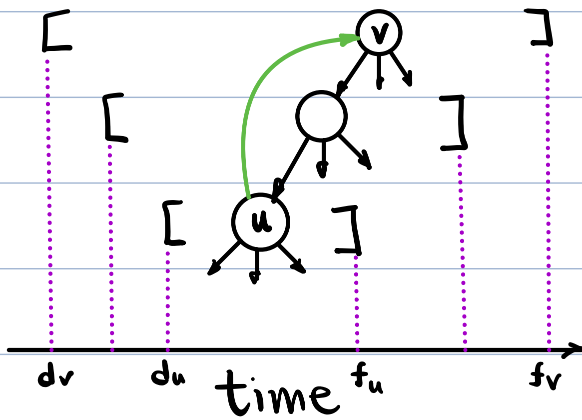
$\Rightarrow f[u] > f[v]$



If (u,v) is a cross edge

$[d[v], f[v]] < [d[u], f[u]]$

$\Rightarrow f[u] > f[v]$



If (u,v) is back edge
$[d[u], f[u]] \subseteq [d[v], f[v]]$
$\Rightarrow f[u] \leq f[v]$

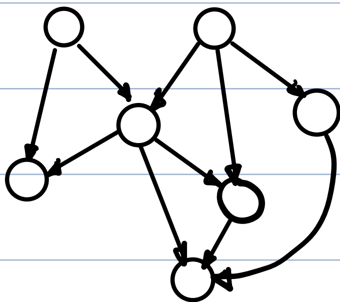
(See latex notes for formal proof)

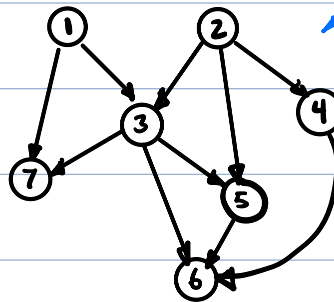
Topological Sorting (via DFS):

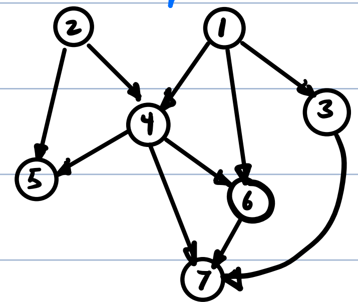
Fact: Given any DAG $G=(V,E)$, there exists a linear ordering of the vertices that respects edge directions

$(u,v) \in E \Rightarrow u < v$

Possibly many







How to compute?

Idea 1: Repeatedly find a sink (or source) v
Put v at end (or start) of order
Remove v

Time: $O(n(n+m))$ - trivial
$O(n+m)$ - more clever

Topological Sort:

Idea 2: DFS

- Order vertices in reverse order of finish times.
- DFS on DAG has no back edges
  by Claim $\Rightarrow \forall (u,v) \in E,\ f[u] > f[v]$
- Implementation:
  - Push vertices on stack when finished
  - Pop stack at end to reverse order

```
topSort(G=(V,E)):
    mark all vertices undiscovered
    S ← empty stack
    for each (undiscovered u ∈ V) topVisit(u)
    pop S + output

topVisit(u):
    mark u visited
    for each (v ∈ Adj[u])
        if (v undiscovered) topVisit(v)
    push u onto S        // finished with u
```

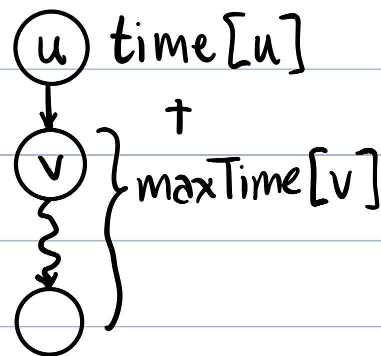
Time: $O(n+m)$ – standard DFS

Longest Path in a DAG:

- Given a DAG where each vertex u stores its duration time[u]. Compute the path that maximizes sum of durations.
- This is shortest completion time assuming:
  - Edges are precedence constraints
  - Allowing maximum parallelism

- Approach: Use DFS
  - Maintain maxTime[u] = length (time) of longest path starting at u.
  - For u, visit each neighbor v:

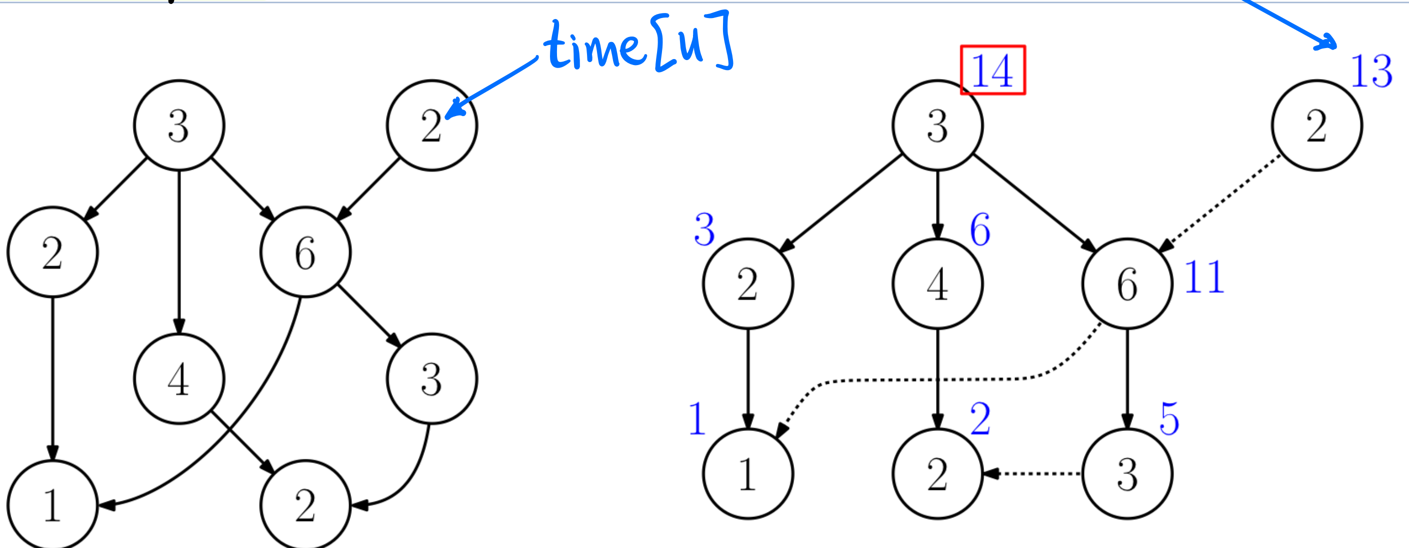
$$\text{maxTime}[u] \leftarrow \max \begin{cases} \text{maxTime}[u] \\ \text{time}[u] + \text{maxTime}[v] \end{cases}$$



```
LongPathVisit(u)
    mark[u] ← discovered;  maxTime[u] ← time[u]
    for each v ∈ Adj[u]
        if (v undiscovered)  LongPathVisit(v)
        maxTime[u] ← max { maxTime[u],
                           time[u] + maxTime[v] }
```

On termination, output $\max_{u \in V} \text{maxTime}[u]$

Example:

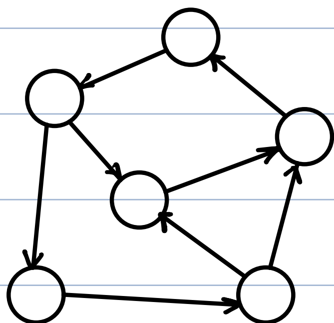


Does this only work on DAGs?
Can we use this to compute the longest simple path (no repeats) in any directed graph?

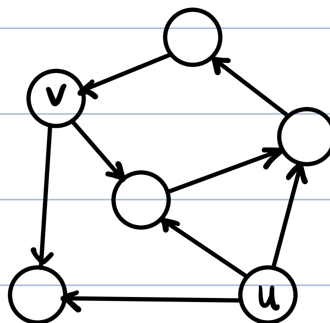
Ans: No – But why not?

Strong Components:

Def: A digraph is **strongly connected** if for every $u, v \in V$, there is a path from u to v and from v to u.

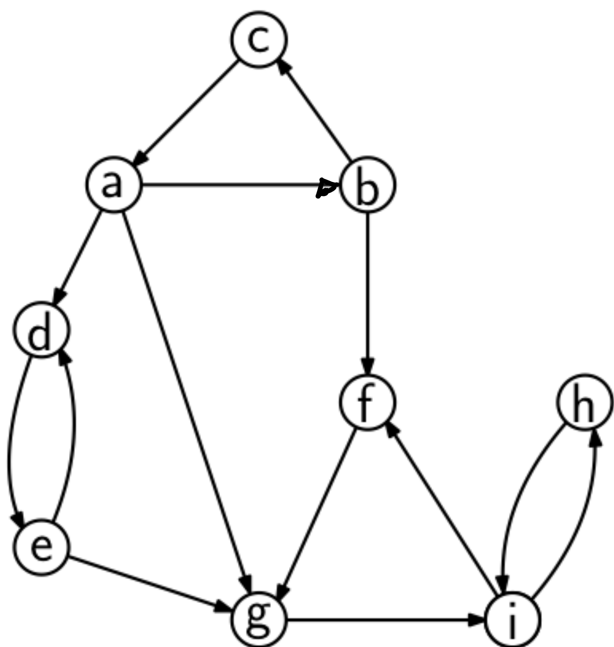


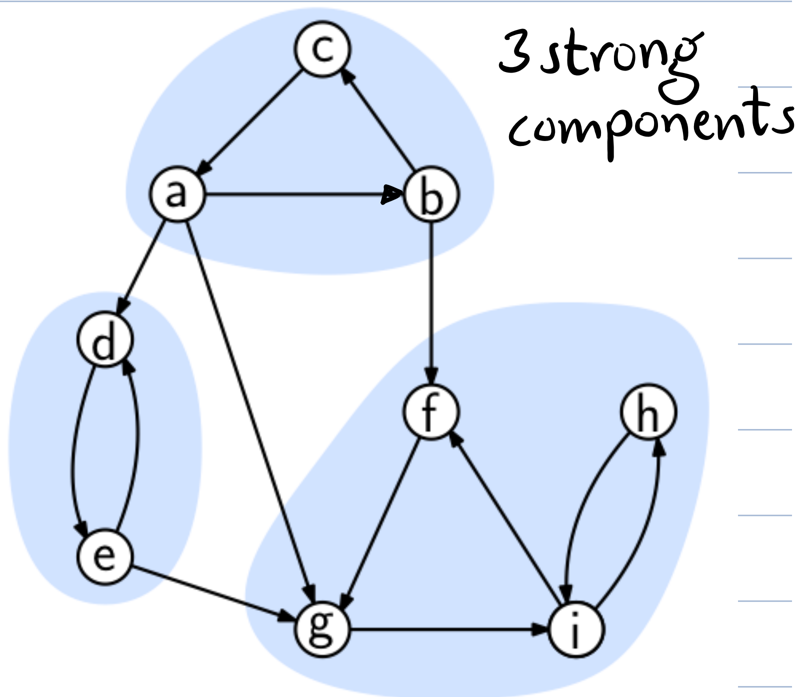
strongly connected



not
(No path from v to u)

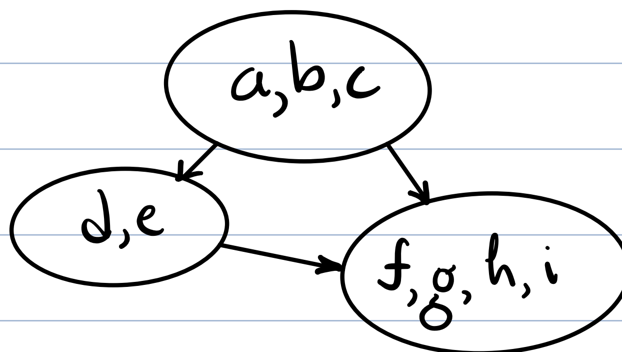
Def: Two vertices u + v are in the same **strong component** if $\exists$ path u to v + v to u





3 strong components

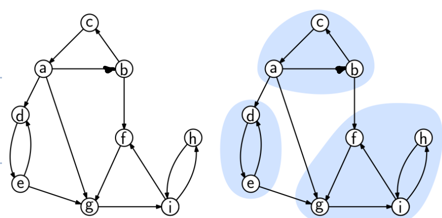
Def: If we "collapse" all the vertices in each strong component, we obtain the component digraph.



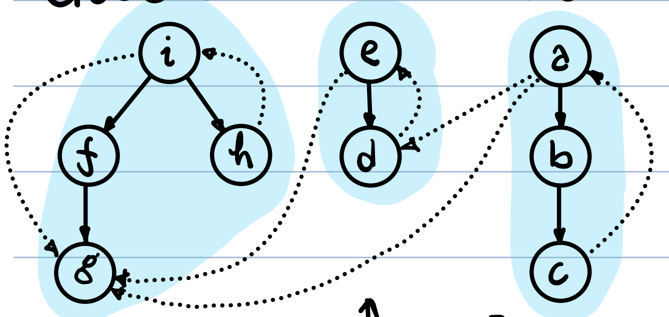
> **Claim:** The component digraph is a DAG

Computing the Strong Components in a digraph $G=(V,E)$

- Use DFS → $O(n+m)$
- Idea – If we visit vertices in the magic order the components appear in separate trees of DFS.



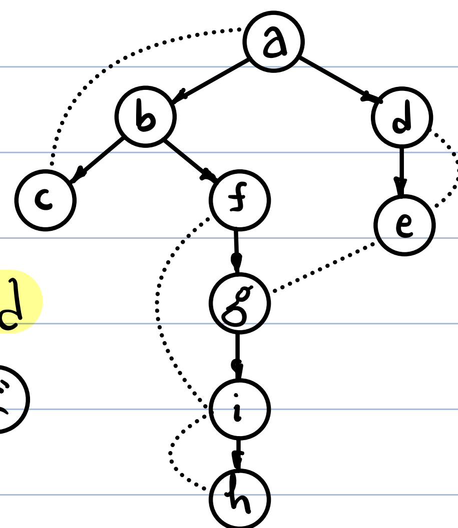
Good order: i…e…a



Each component a separate tree

Bad order: a…

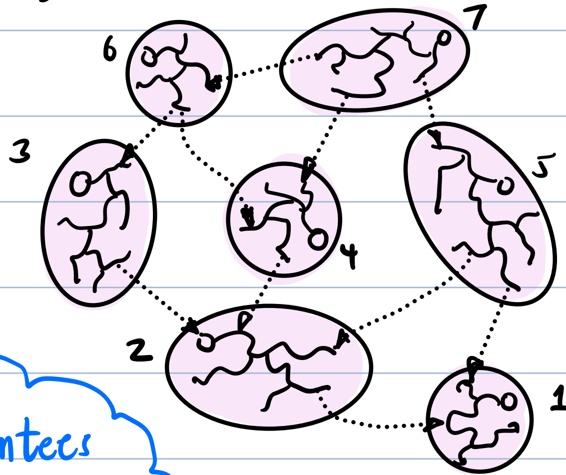
All smooshed together ☹



What makes an order good?

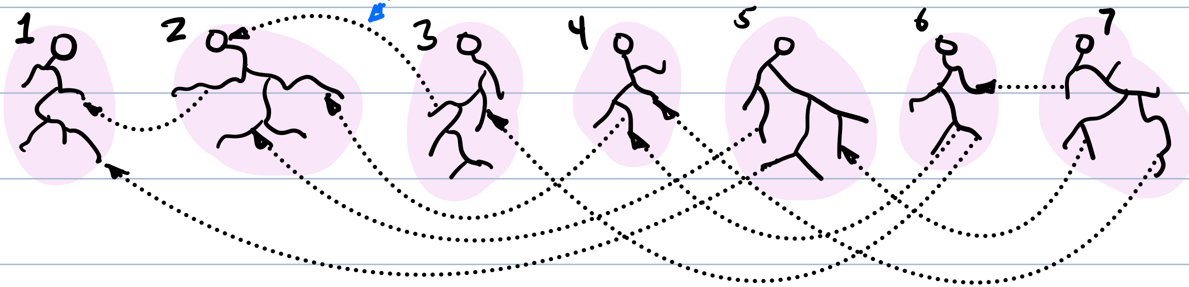
- When DFS visits a vertex, it does not finish until all reachable vertices are visited
- Key is to visit strong components in reverse topological order of component digraph.

DFS(G):



Shows DFS trees and edges between components

This order guarantees that edges between components are cross edges



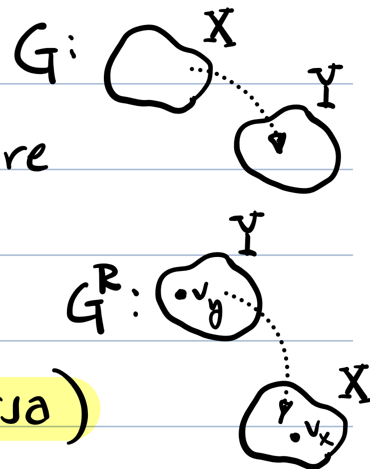
Huh? How can you order the components without knowing the strong components??

Insanely Clever Trick:

- Compute reverse graph G^R (reverse directions)
- DFS(G^R)
- Sort vertices inversely by finish times

Claim: This is the *magic order*
(Doesn't require knowledge of components)

Why?



- Consider components X + Y in G, where X can reach Y (but not vice versa)
- In G^R, these are strong components, and Y can reach X (but not vice versa)

- Let v_x + v_y be the last vertices to finish in X + Y, respectively (in DFS(G^R))
- **Observe:** v_x must finish before v_y

  Why? If DFS hits X first, it visits all of X before starting Y. $\Rightarrow f(v_y) > f(v_x)$

  If DFS hits Y first, it will leak into X, visit/finish everything in X, and then return + finish Y. $\Rightarrow f(v_y) > f(v_x)$

From this we have:

⇒ In DFS(G^R): $\max_{y \in Y} f(y) > \max_{x \in X} f(x)$

⇒ Component Y will be visited before X in DFS(G)

⭐ ⇒ DFS(G) visits strong components in reverse topological order of component digraph.

Final algorithm:

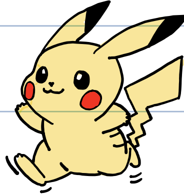
```
Strong Components(G=(V,E))
  - Compute G^R
  - Run DFS(G^R) + record finish times
  - Sort vertices reversely by finish times
  - Run DFS(G)
      - Whenever we start a new tree,
        use the above order
  - Output DFS trees as Strong Comps.
```

Correctness: Follow from previous observations.

Time: Let $n = |V|$, $m = |E|$

- Compute G^R : $O(n+m)$ (exercise)
- $DFS(G^R)$: $O(n+m)$ (G^R has same size)
- Sort vertices : $O(n)$ (radix sort)
- $DFS(G)$: $O(n+m)$

Total : $O(n+m)$



Summary:

- DAGs + Acyclicity Testing
- Topological Sorting (via DFS)
- Longest Path in a DAG
- Strong Components