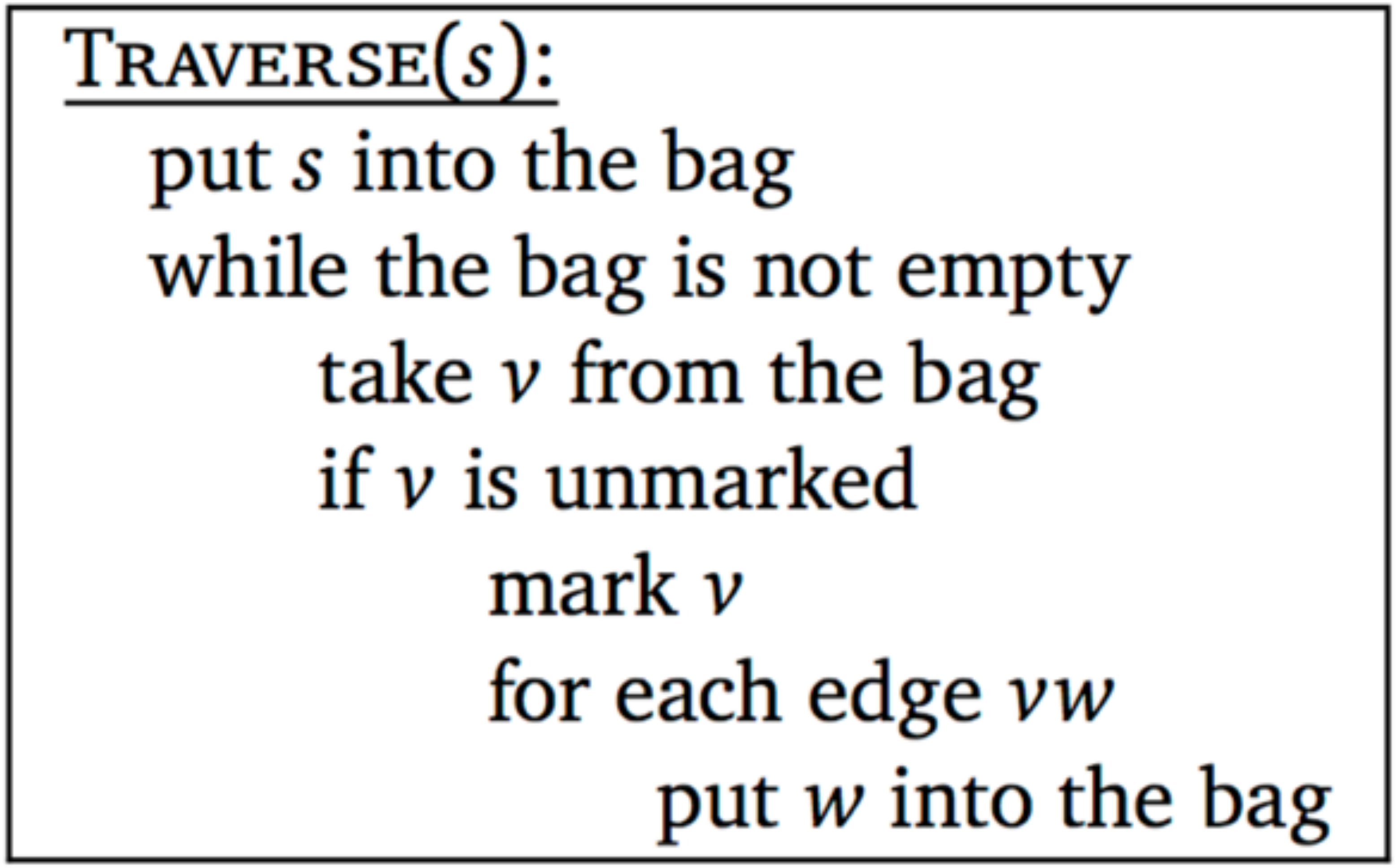
DFS, Topological Sort

Lecture18

- Mid-semester survey which can be accessed at https://illinois.edu/sb/sec/7058301.
- Midterm in two weeks!
- Review session the Thursday before.

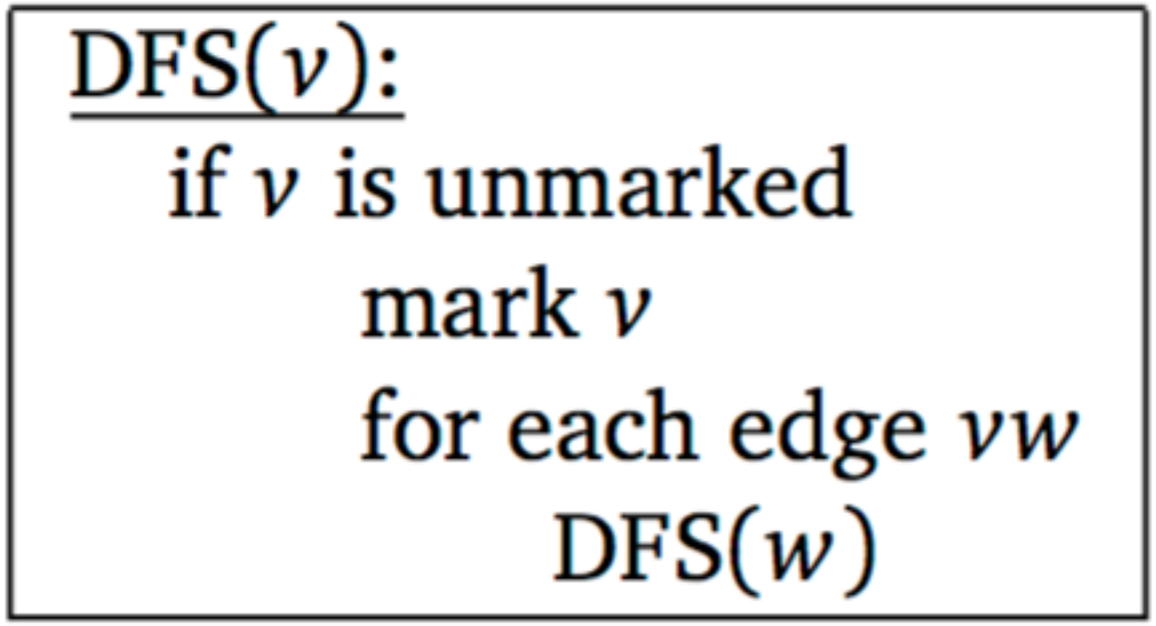
How to traverse a graph?



```
TRAVERSE(s):
    put s into the bag
    while the bag is not empty
        take v from the bag
        if v is unmarked
            mark v
            for each edge vw
                put w into the bag
```

stack =LIFO (DFS)
Queue = FIFO (BFS)
Priority Queue = lightest out
Random, etc

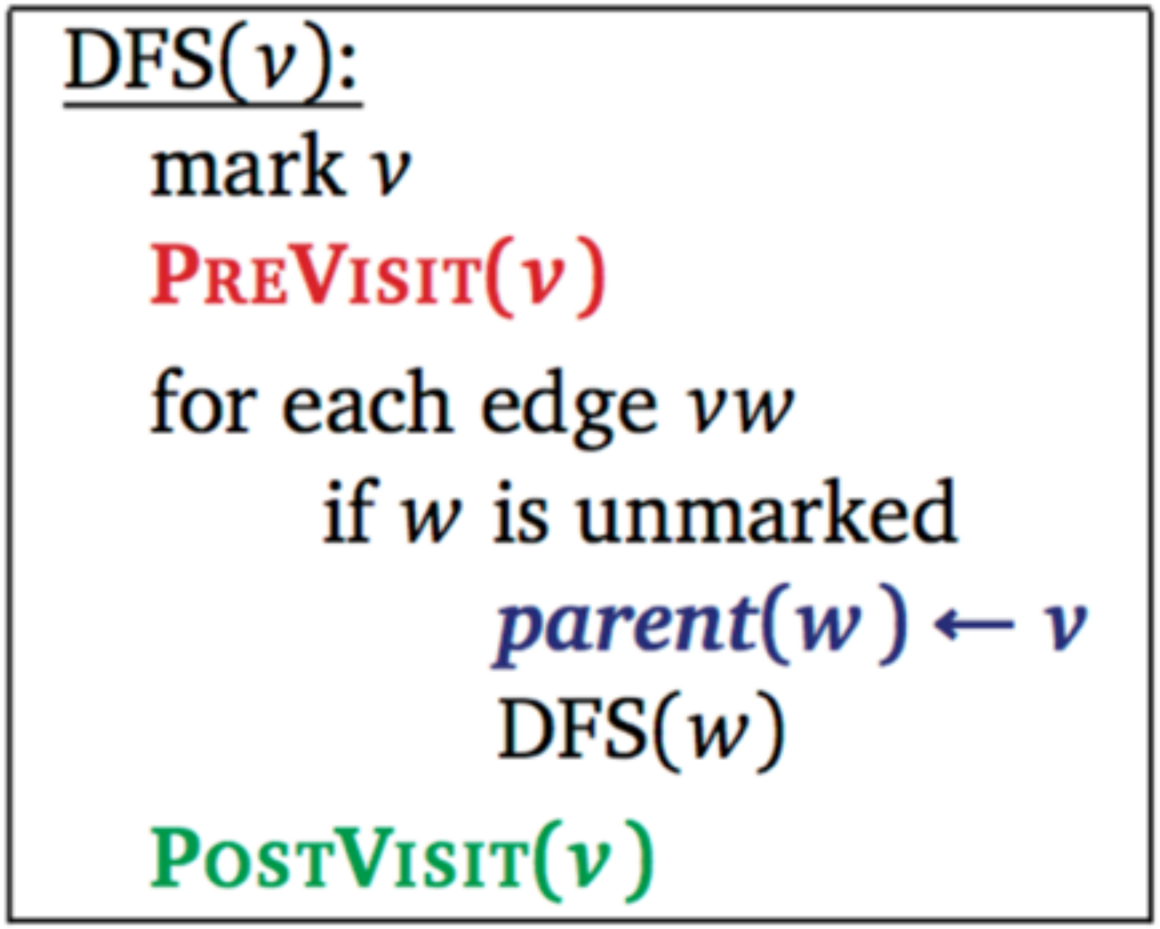
DFS



```
DFS($v$):
    if $v$ is unmarked
        mark $v$
        for each edge $vw$
            DFS($w$)
```

stack =LIFO (DFS)

DFS

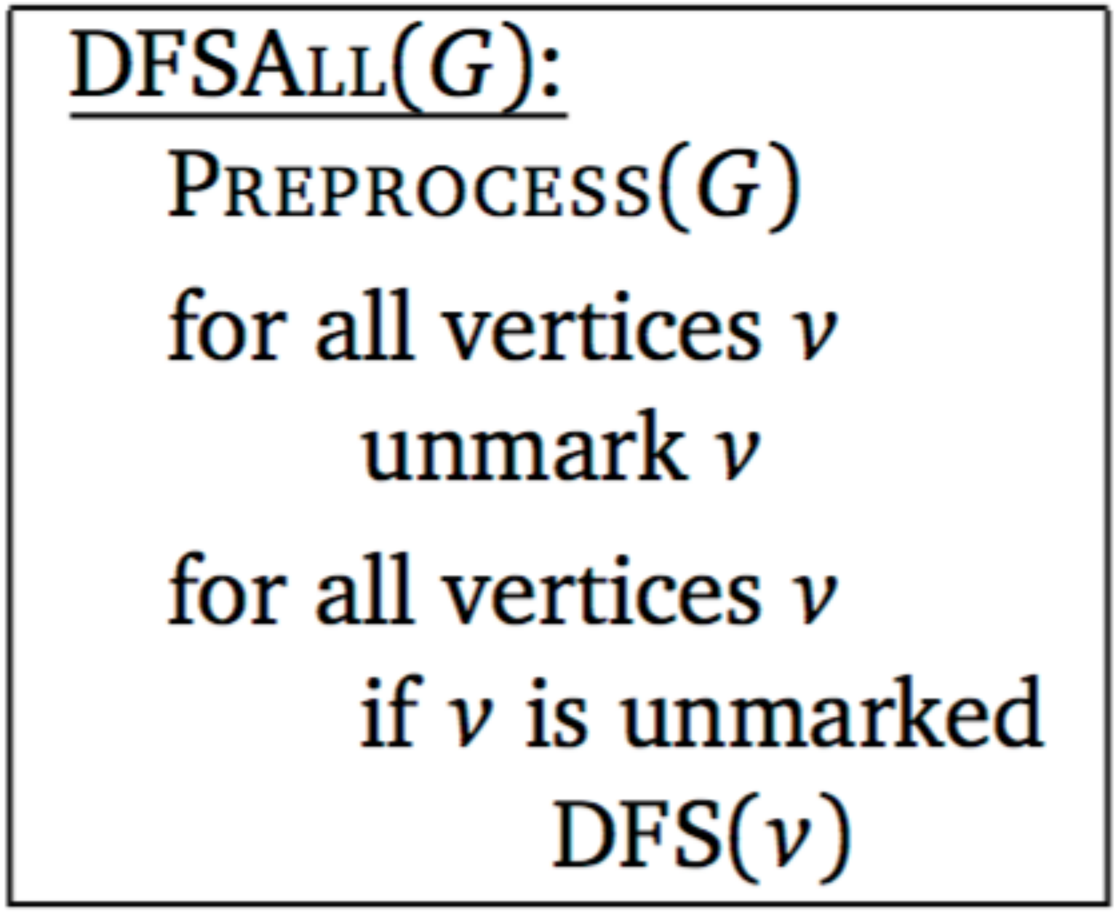


```
DFS(v):
    mark v
    PreVisit(v)
    for each edge vw
        if w is unmarked
            parent(w) ← v
            DFS(w)
    PostVisit(v)
```

check if a node is marked
before recursively exploring it.
DFS(v) called once for each v.

Disconnected graphs?

DFS



```
DFSAll(G):
    Preprocess(G)
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            DFS(v)
```

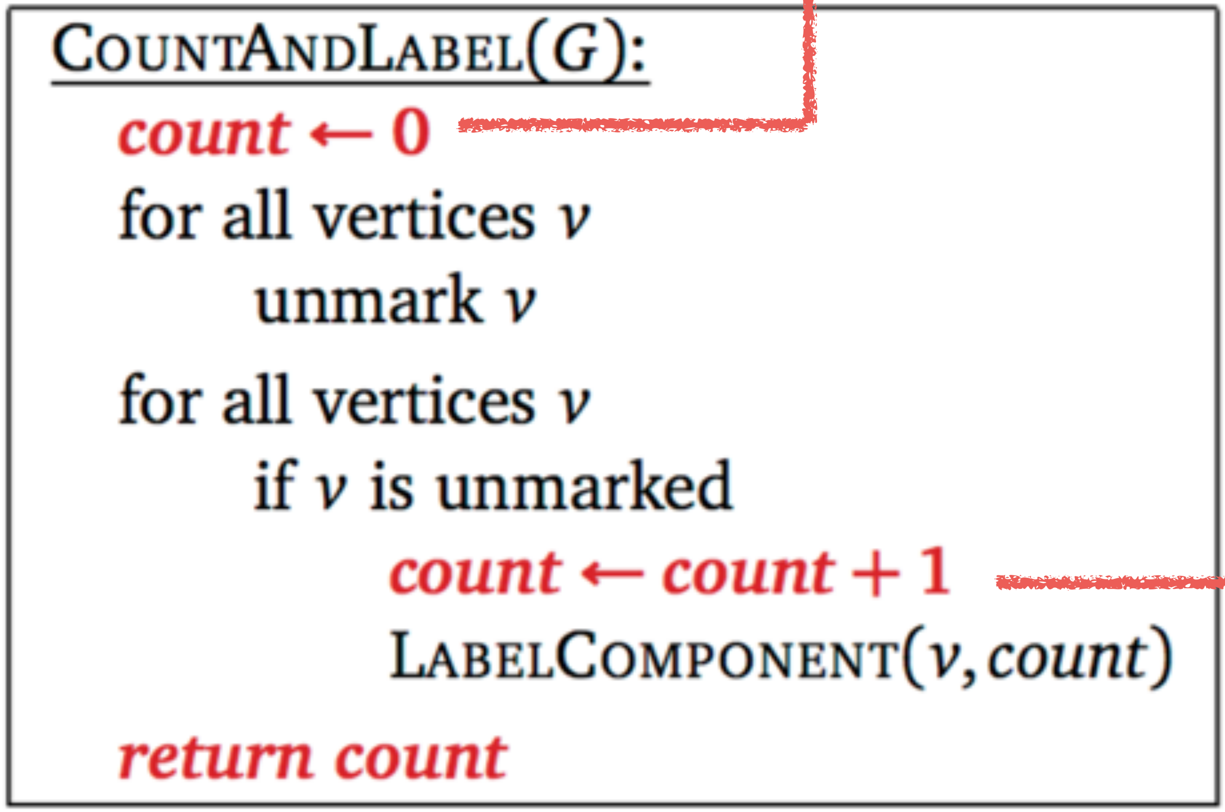
check if a node is marked before recursively exploring it. DFS(v) called once for each v.

Disconnected graphs?

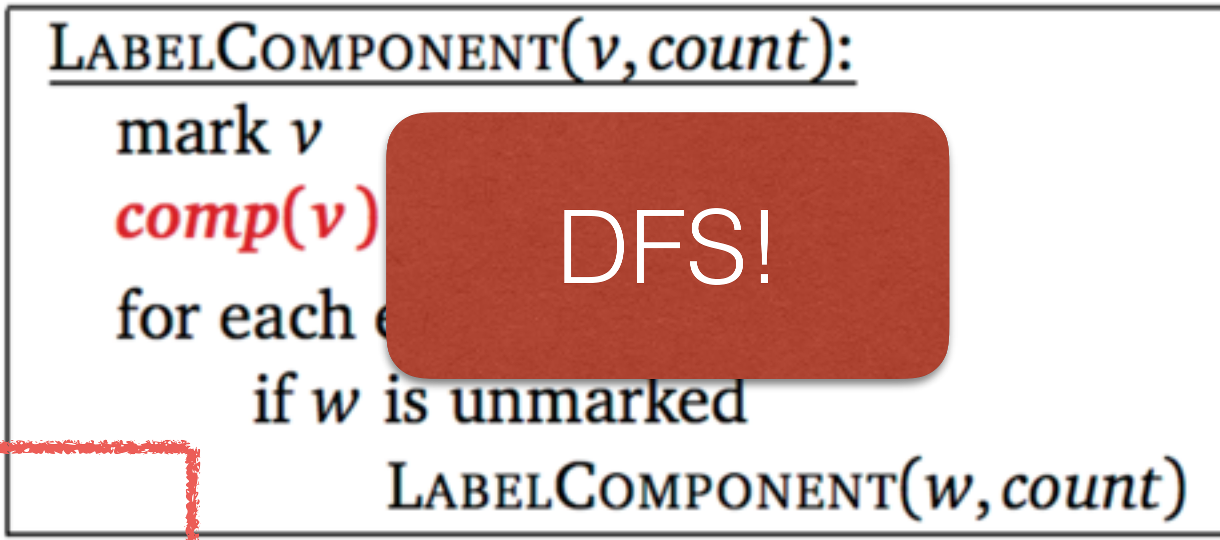
How to label all the vertices in each component with same label?

DFS

I have found no connected components yet



```
COUNTANDLABEL(G):
    count ← 0
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            count ← count + 1
            LABELCOMPONENT(v, count)
    return count
```



```
LABELCOMPONENT(v, count):
    mark v
    comp(v)
    for each
        if w is unmarked
            LABELCOMPONENT(w, count)
```

DFS!

Every time I find a new component increase counter

DFS

I have found no connected components yet

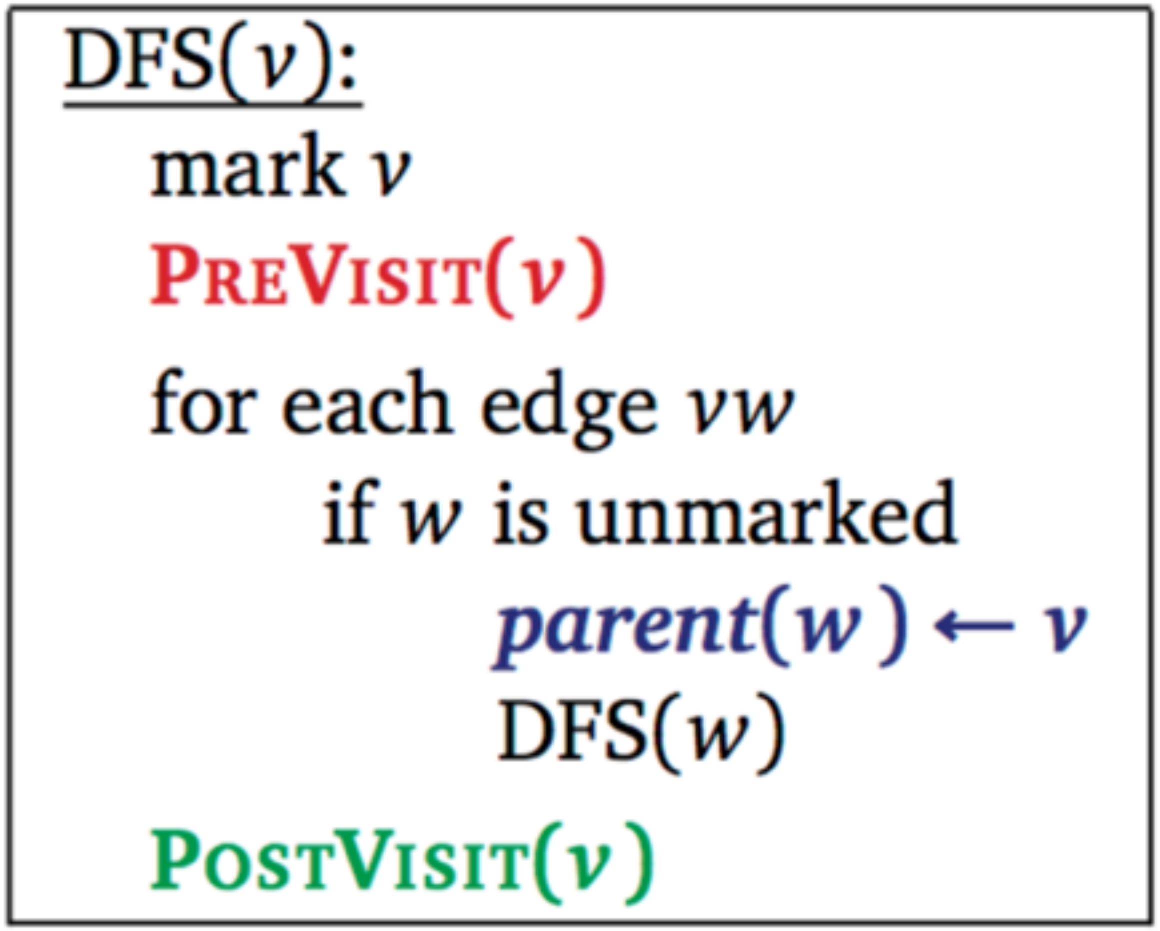
```
CountAndLabel(G):
    $count \leftarrow 0$
    for all vertices $v$
        unmark $v$
    for all vertices $v$
        if $v$ is unmarked
            $count \leftarrow count + 1$
            LabelComponent($v$, $count$)
    return $count$
```

```
LabelComponent($v$, $count$):
    mark $v$
    $comp(v) \leftarrow count$
    for each edge $vw$
        if $w$ is unmarked
            LabelComponent($w$, $count$)
```

Every time I fine a new component increase counter

Label each vertex in the component with the index of the component

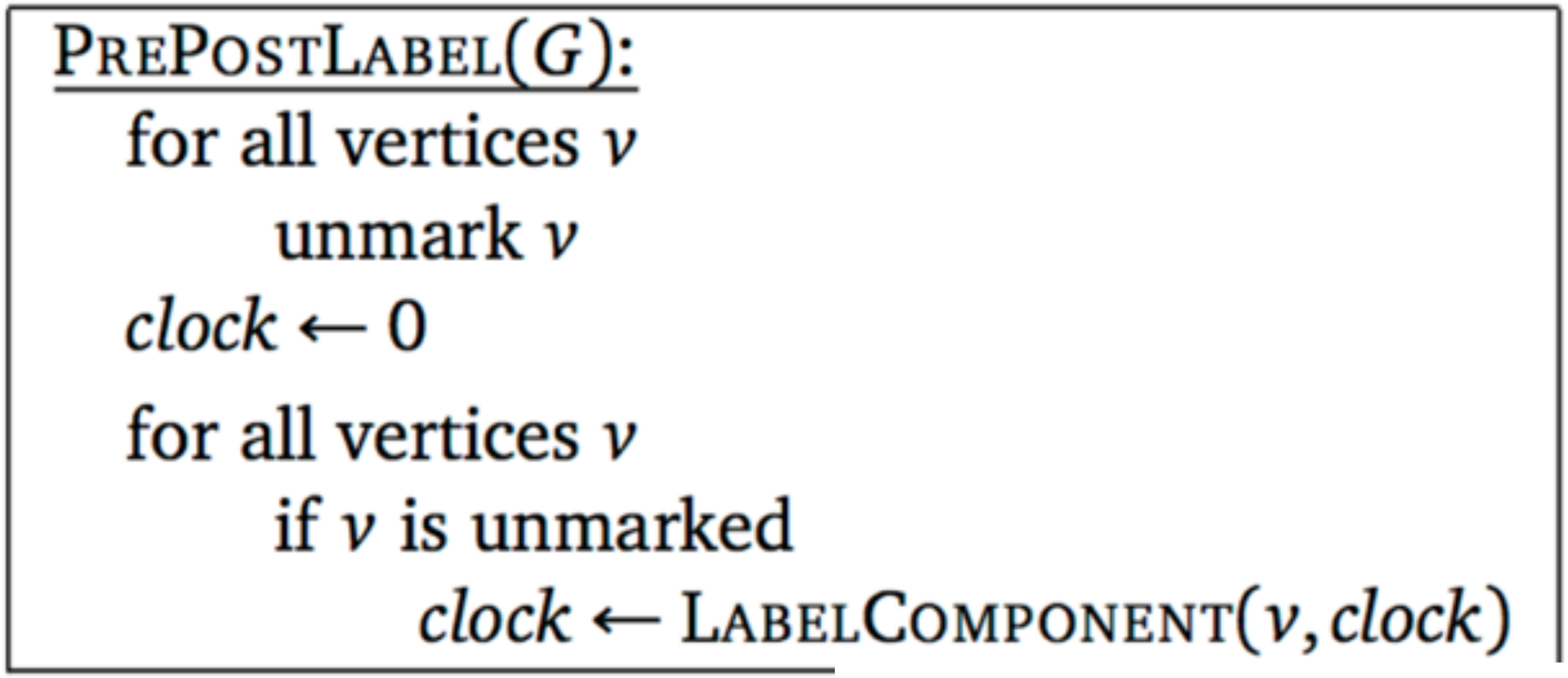
DFS



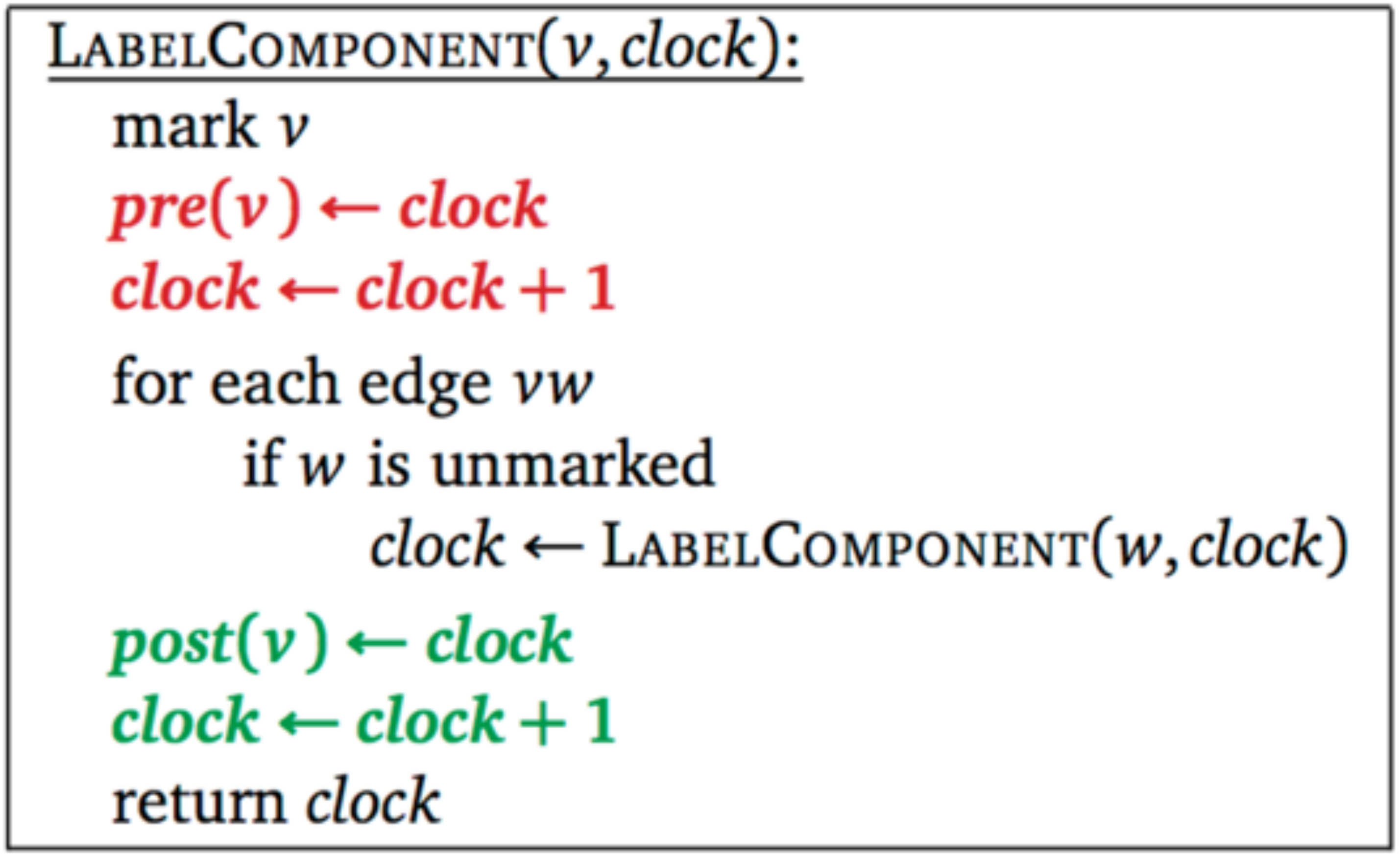
```
DFS(v):
    mark v
    PreVisit(v)
    for each edge vw
        if w is unmarked
            parent(w) ← v
            DFS(w)
    PostVisit(v)
```

Sometimes I want to compute an order of the vertices in a graph which is consistent with pre or postorder traversal e.g. DP

Preorder/Postorder

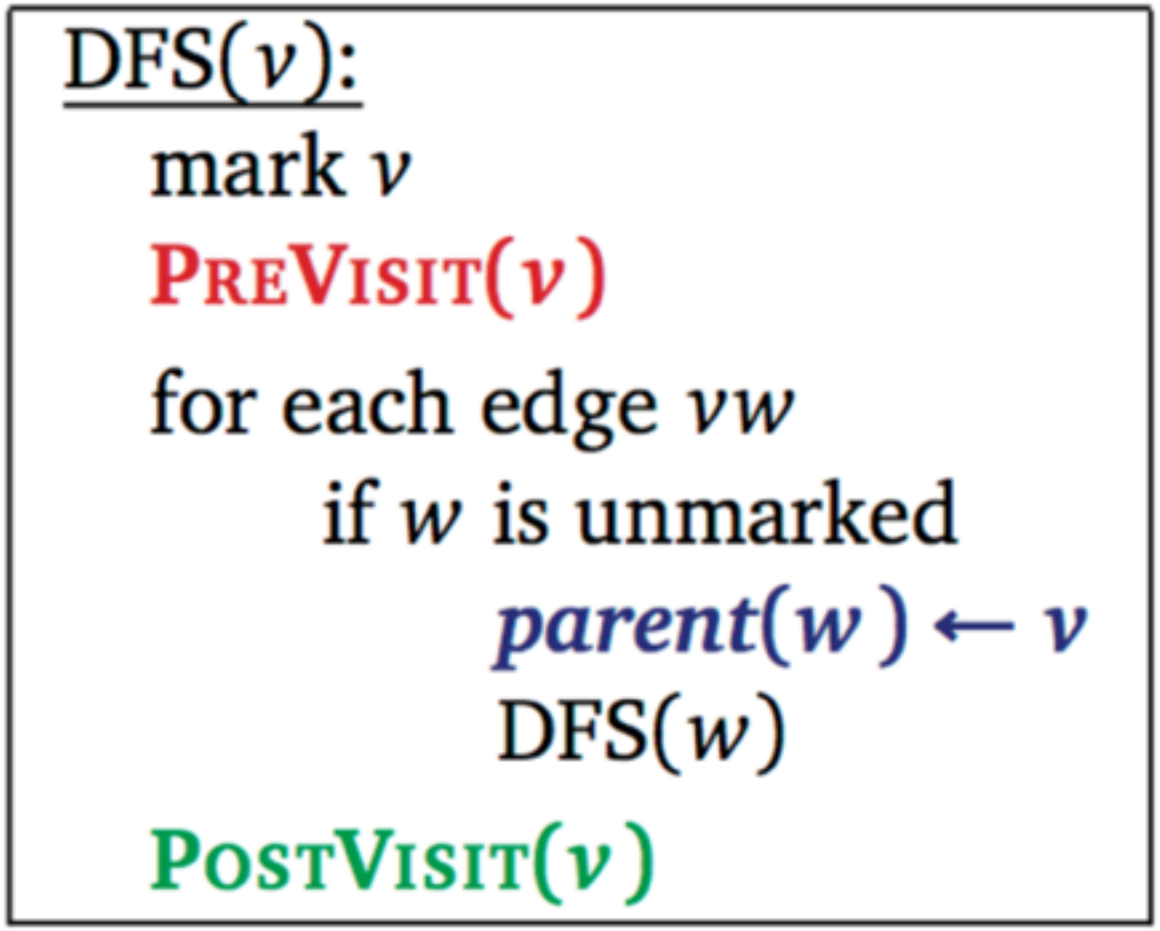


```
PrePostLabel(G):
    for all vertices v
        unmark v
    clock ← 0
    for all vertices v
        if v is unmarked
            clock ← LabelComponent(v, clock)
```

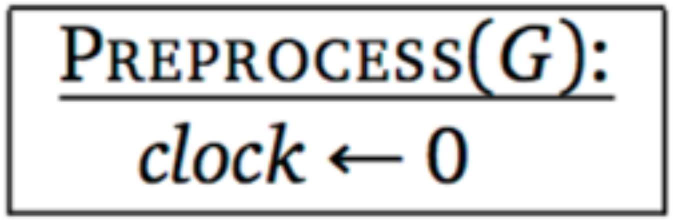


```
LabelComponent(v, clock):
    mark v
    pre(v) ← clock
    clock ← clock + 1
    for each edge vw
        if w is unmarked
            clock ← LabelComponent(w, clock)
    post(v) ← clock
    clock ← clock + 1
    return clock
```

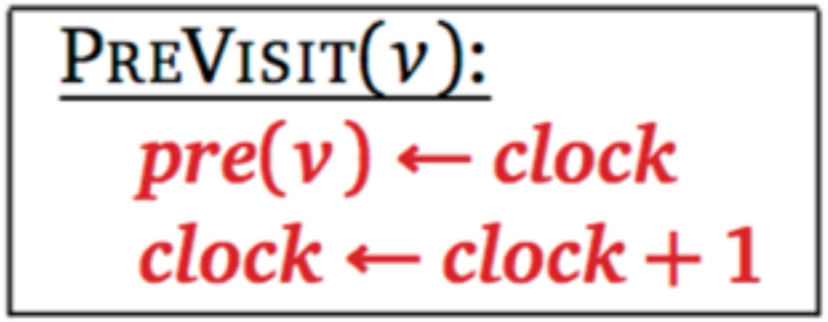
DFS



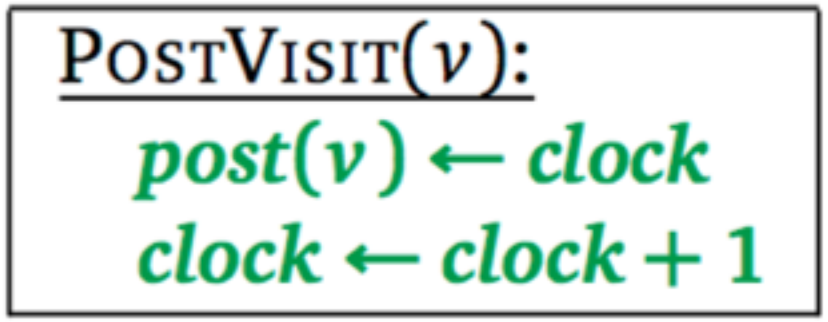
```
DFS($v$):
    mark $v$
    PreVisit($v$)
    for each edge $vw$
        if $w$ is unmarked
            $parent(w) \leftarrow v$
            DFS($w$)
    PostVisit($v$)
```



```
Preprocess($G$):
    $clock \leftarrow 0$
```

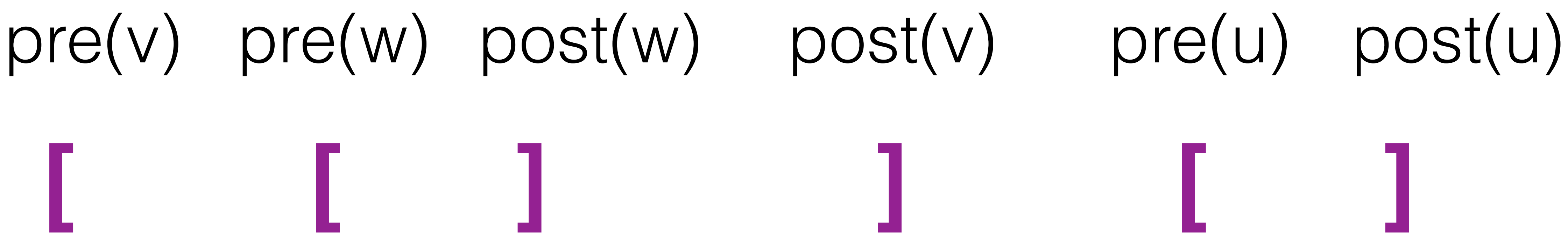


```
PreVisit($v$):
    $pre(v) \leftarrow clock$
    $clock \leftarrow clock + 1$
```



```
PostVisit($v$):
    $post(v) \leftarrow clock$
    $clock \leftarrow clock + 1$
```

DFS

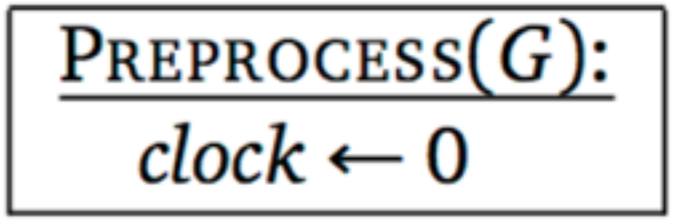


pre(v) pre(w) post(w) post(v) pre(u) post(u)

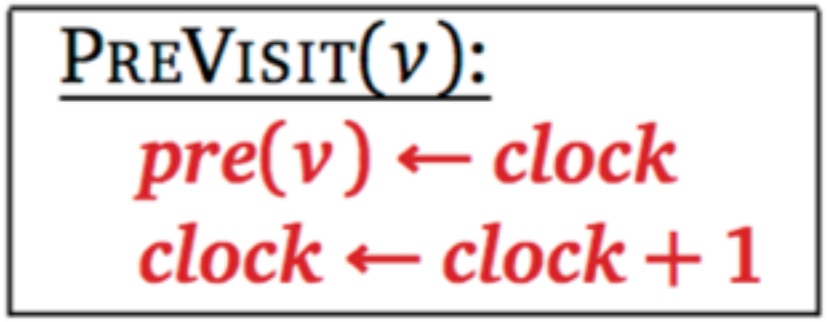
[[ ]] [ ]

(v,w) edge : intervals are nested

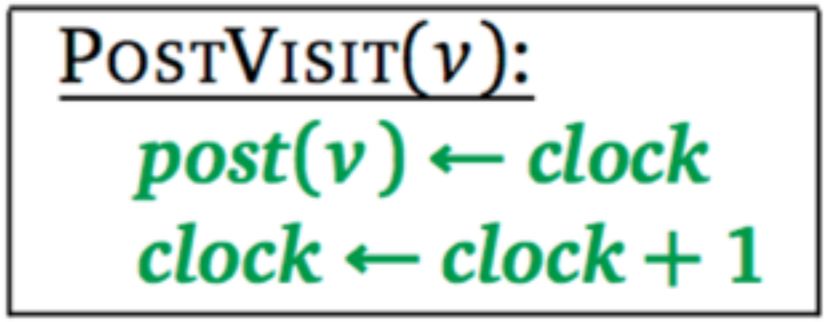
Different way of encoding the DFS tree for the recursive algorithm



```
PREPROCESS(G):
    $clock \leftarrow 0$
```

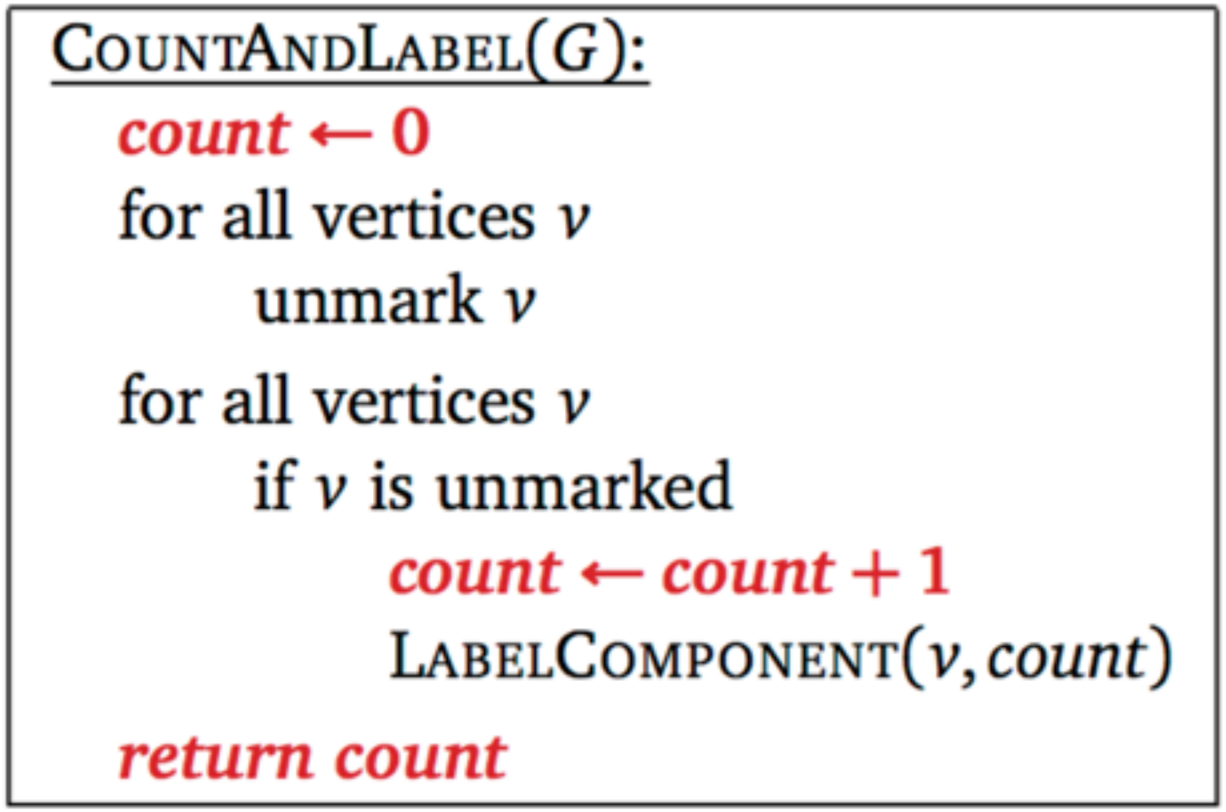


```
PREVISIT(v):
    $pre(v) \leftarrow clock$
    $clock \leftarrow clock + 1$
```

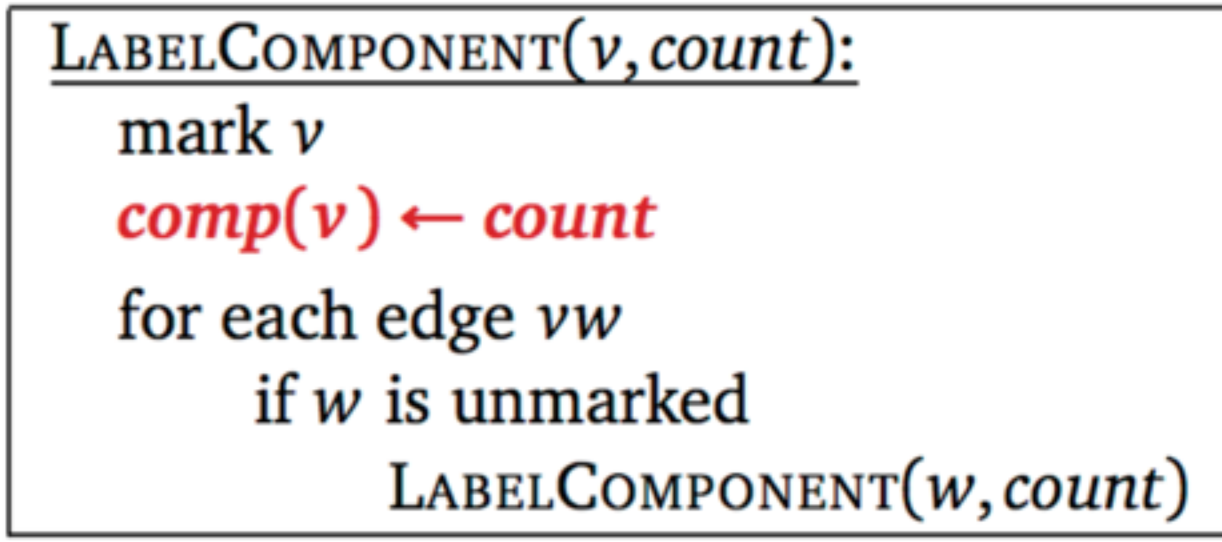


```
POSTVISIT(v):
    $post(v) \leftarrow clock$
    $clock \leftarrow clock + 1$
```

DFS



```
COUNTANDLABEL(G):
    $count \gets 0$
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            $count \gets count + 1$
            LABELCOMPONENT(v, count)
    return count
```

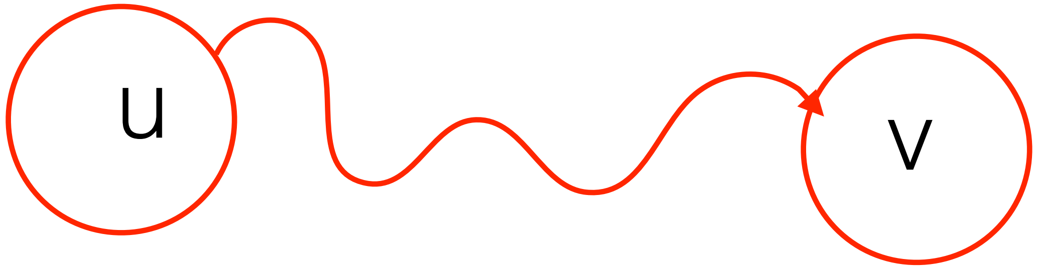


```
LABELCOMPONENT(v, count):
    mark v
    $comp(v) \gets count$
    for each edge vw
        if w is unmarked
            LABELCOMPONENT(w, count)
```

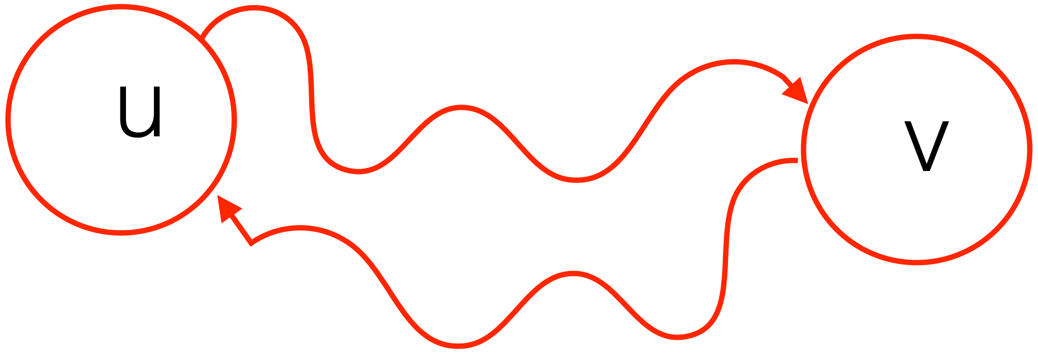
assumes the graph is undirected
What about directed graphs?
When is a graph connected?

DFS

In directed graph vertex u can reach vertex v iff there is a directed path from u to b

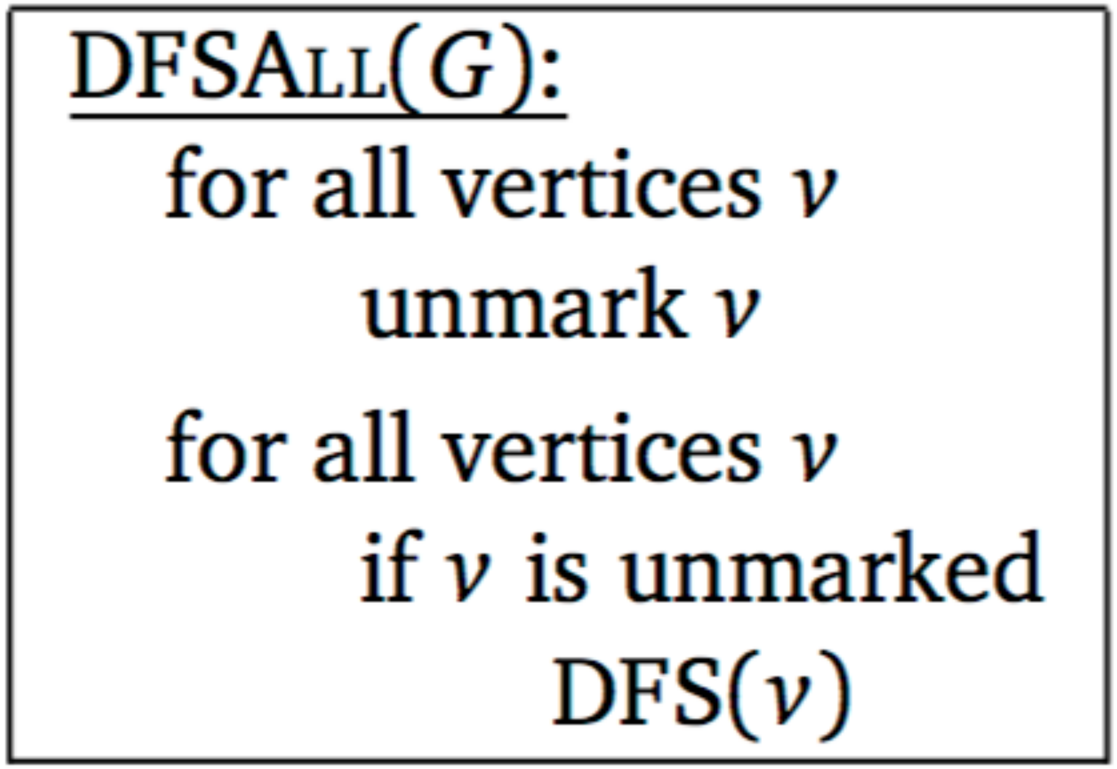


u and v are strongly connected if u can reach v and v can reach u

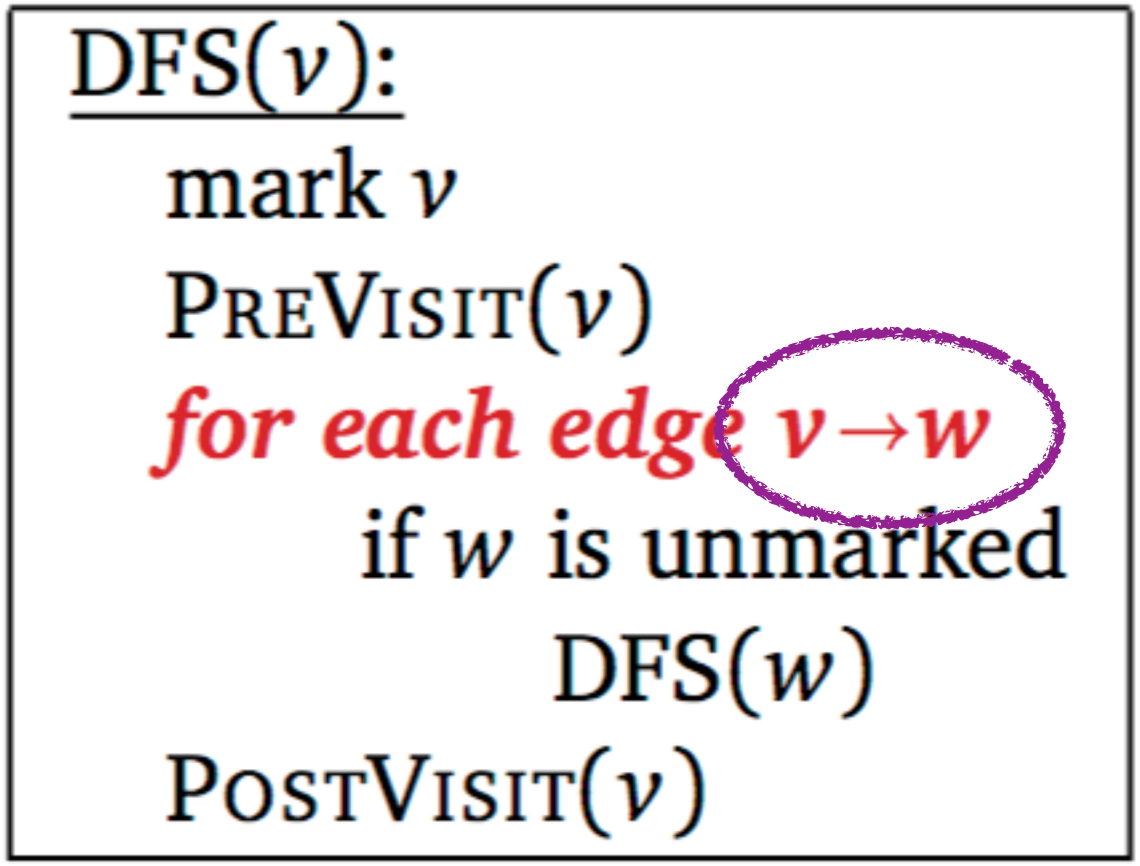


directed cycle!

DFS for directed graphs



```
DFSALL(G):
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            DFS(v)
```



```
DFS(v):
    mark v
    PREVISIT(v)
    for each edge $v \to w$
        if w is unmarked
            DFS(w)
    POSTVISIT(v)
```

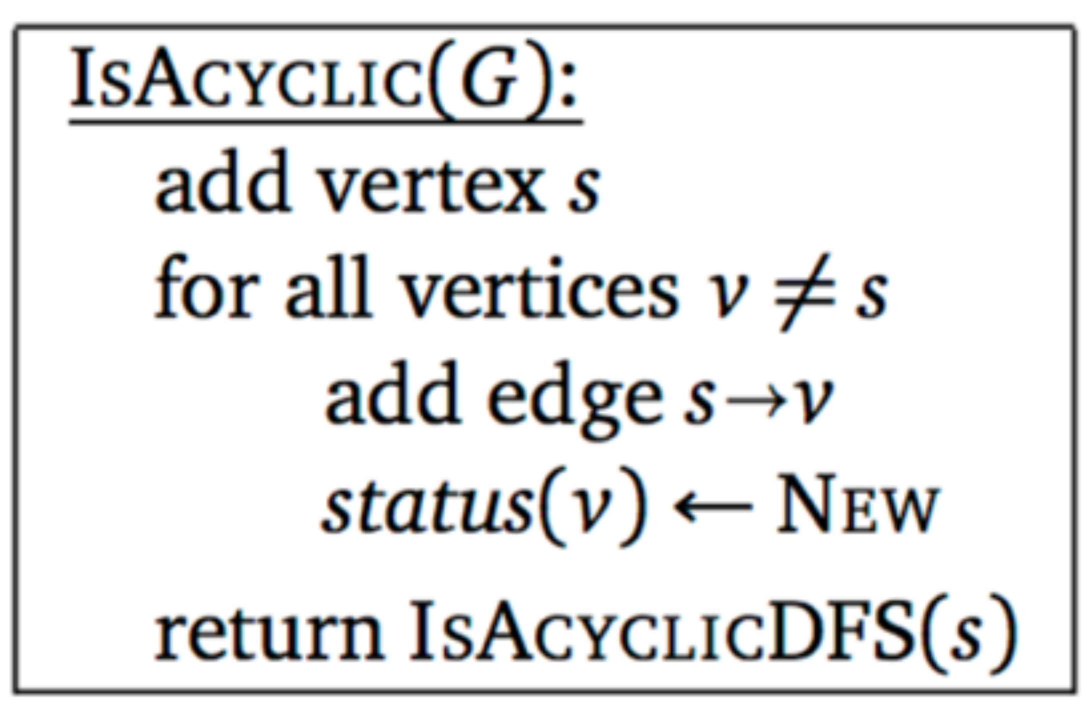
DFS

Think of two extremes

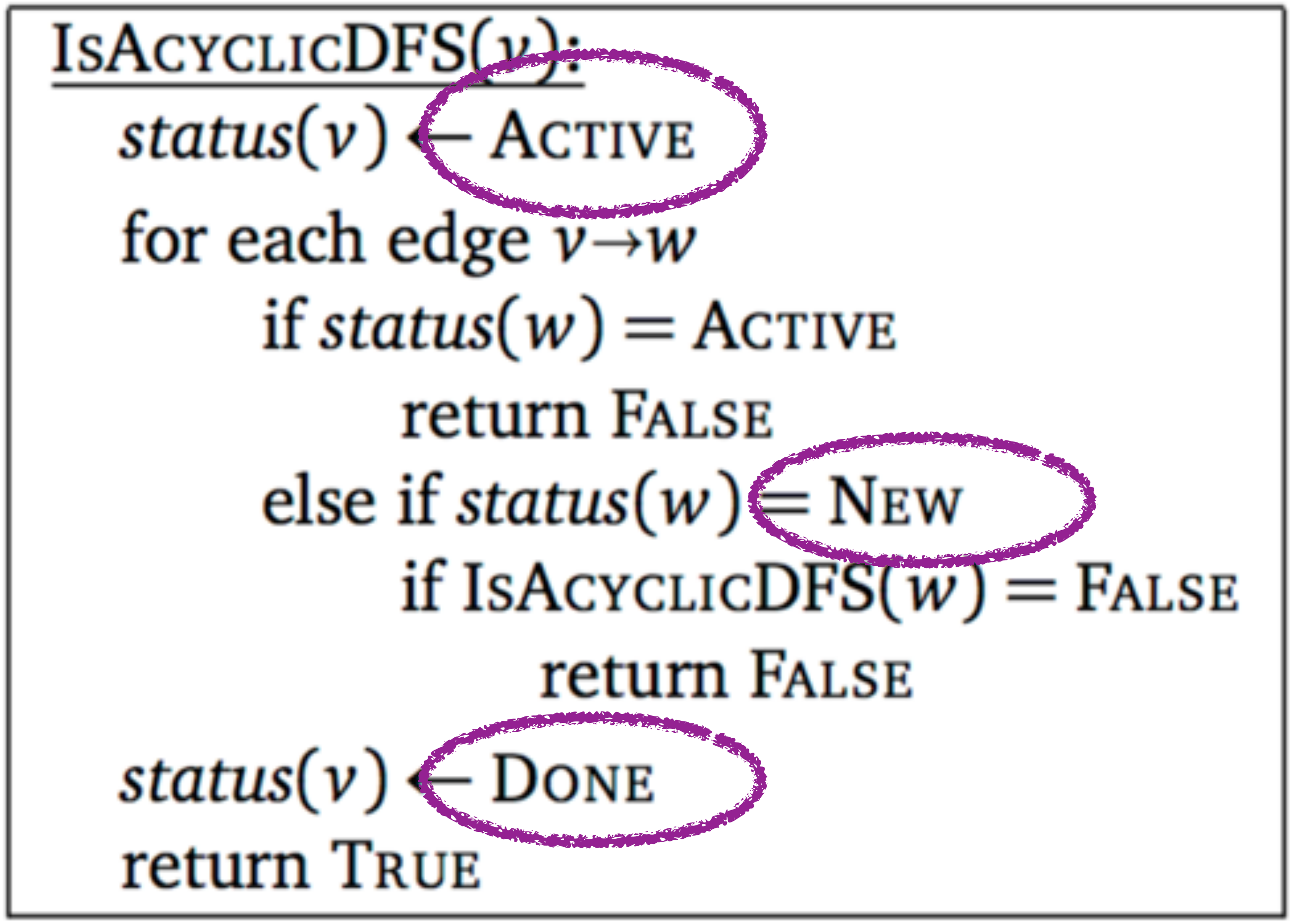
1) There are no directed cycles (DAG)
2) Every two vertices have a directed cycle between them

How do i decide if a graph is a DAG or strongly connected?

Is it a DAG?



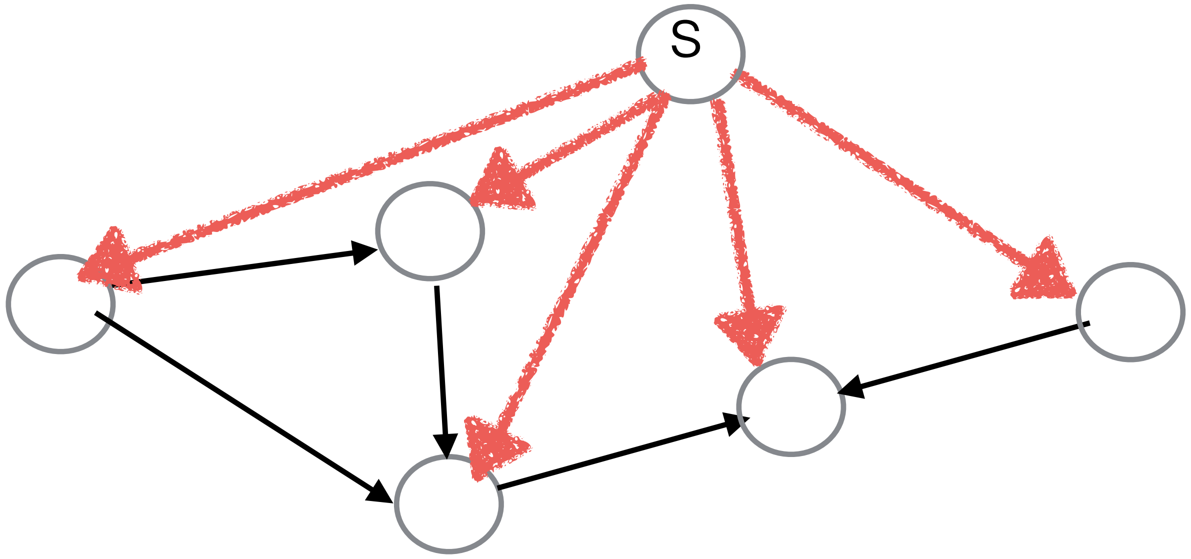
```
IsAcyclic($G$):
    add vertex $s$
    for all vertices $v \neq s$
        add edge $s \to v$
        $status(v) \leftarrow$ New
    return IsAcyclicDFS($s$)
```



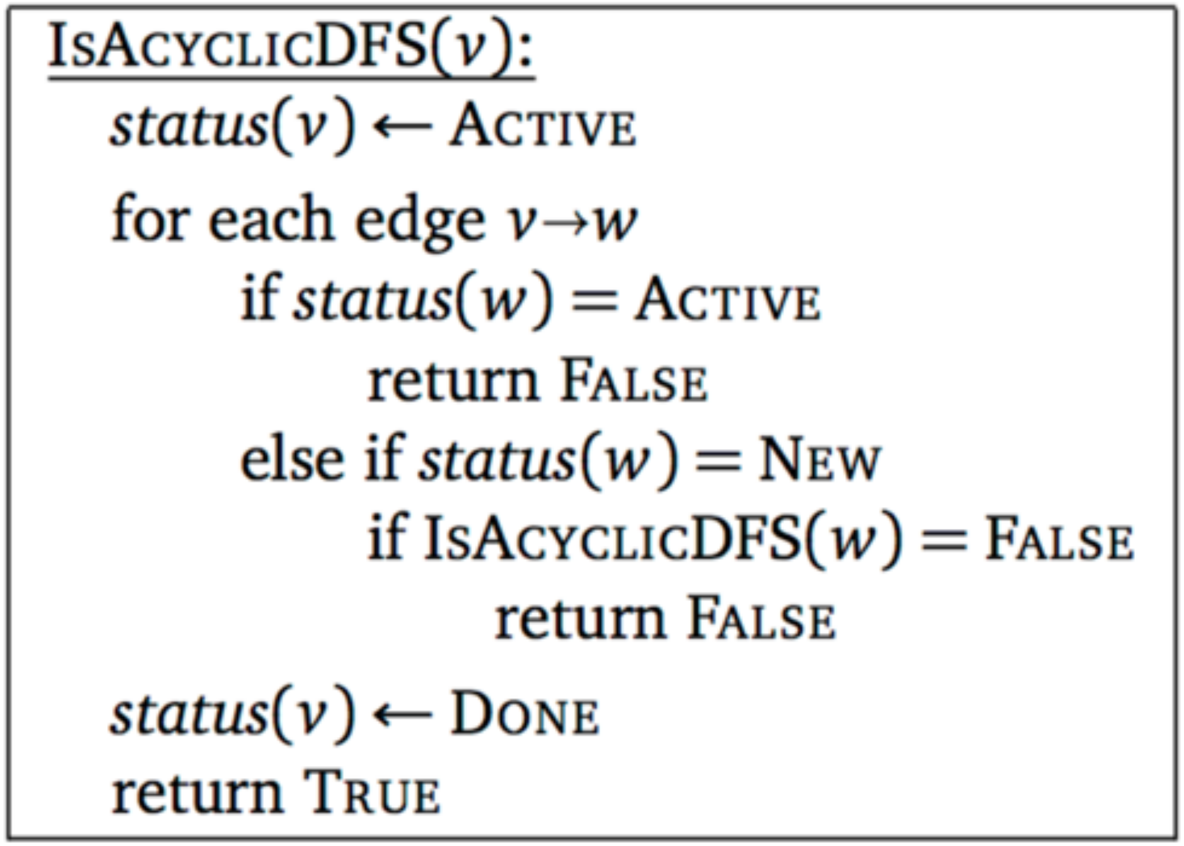
```
IsAcyclicDFS($v$):
    $status(v) \leftarrow$ Active
    for each edge $v \to w$
        if $status(w) =$ Active
            return False
        else if $status(w) =$ New
            if IsAcyclicDFS($w$) $=$ False
                return False
    $status(v) \leftarrow$ Done
    return True
```

a vertex can be
NEW, ACTIVE or DONE

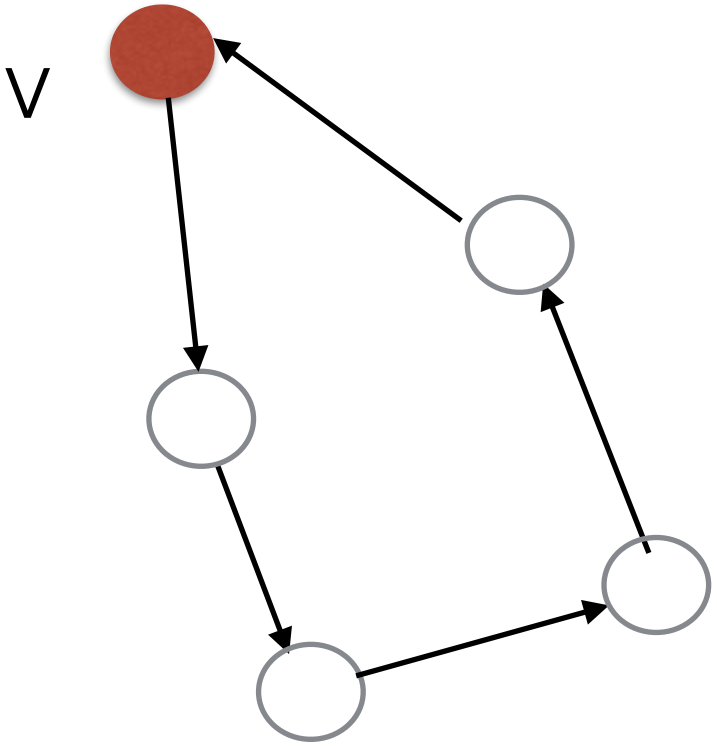
directed cycle if and only if
I reach an ACTIVE vertex
from an ACTIVE vertex



Is it a DAG?



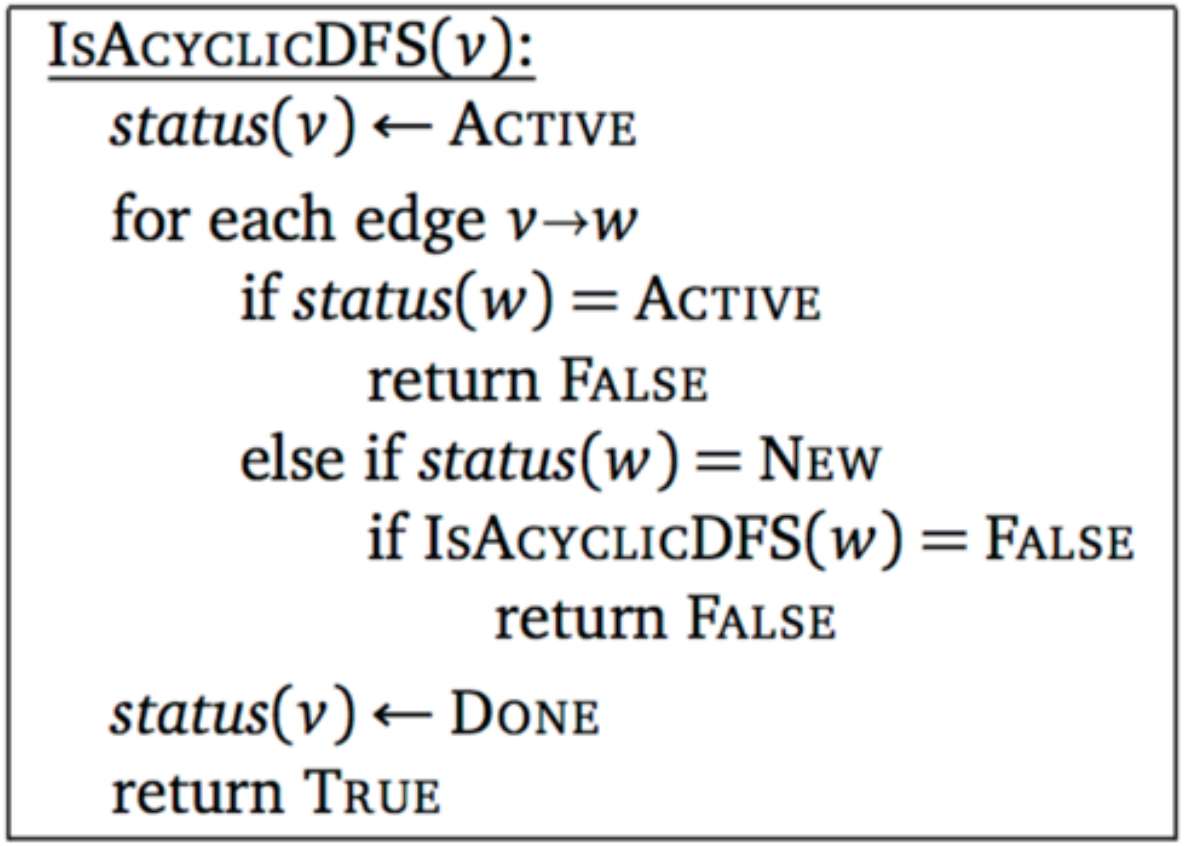
```
IsAcyclicDFS($v$):
    $status(v) \leftarrow$ ACTIVE
    for each edge $v \to w$
        if $status(w) =$ ACTIVE
            return FALSE
        else if $status(w) =$ NEW
            if IsAcyclicDFS($w$) = FALSE
                return FALSE
    $status(v) \leftarrow$ DONE
    return TRUE
```



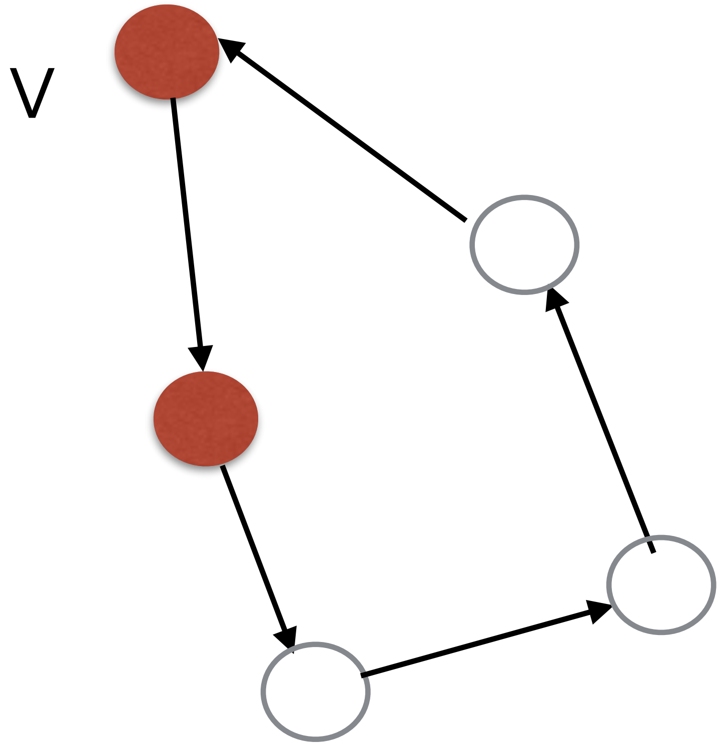
a vertex can be
NEW, ACTIVE or DONE

directed cycle if and only if
I reach an ACTIVE vertex
from an ACTIVE vertex

Is it a DAG?



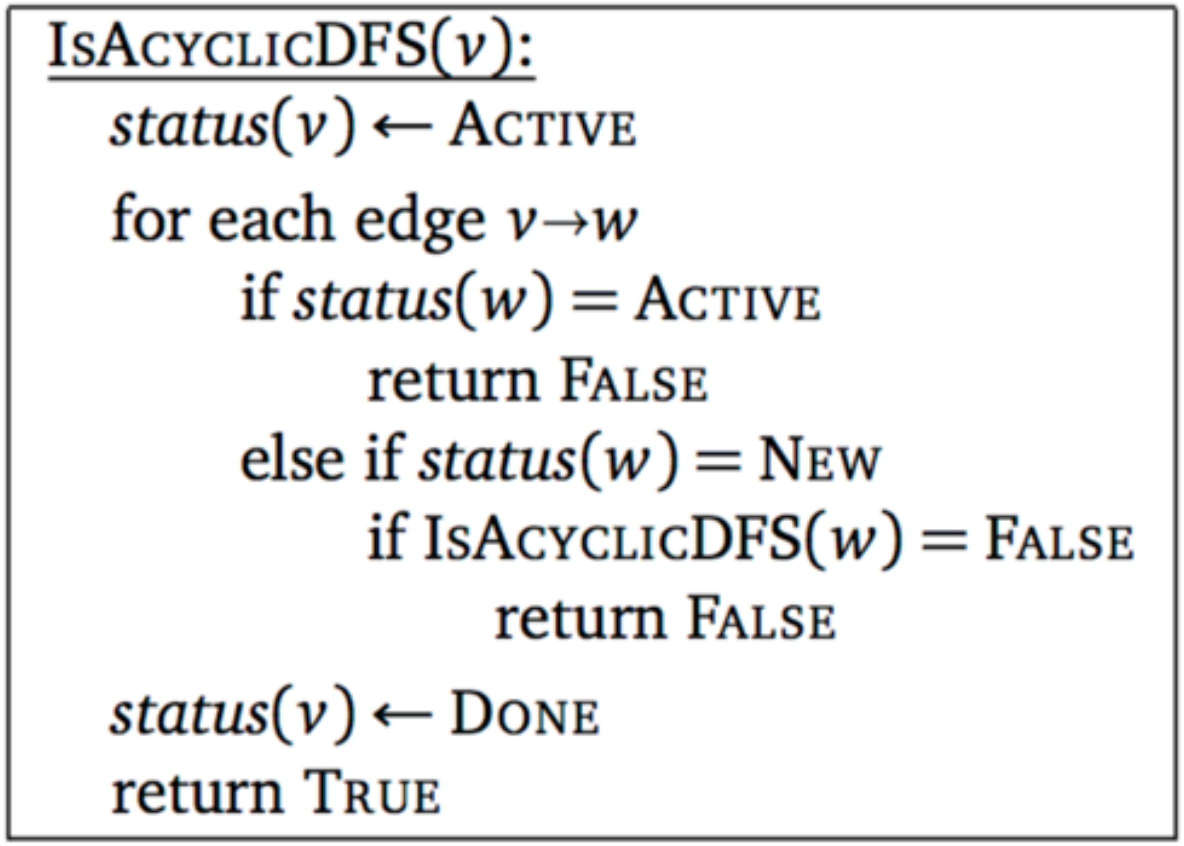
```
IsAcyclicDFS($v$):
    $status(v) \leftarrow$ Active
    for each edge $v \rightarrow w$
        if $status(w) =$ Active
            return False
        else if $status(w) =$ New
            if IsAcyclicDFS($w$) = False
                return False
    $status(v) \leftarrow$ Done
    return True
```



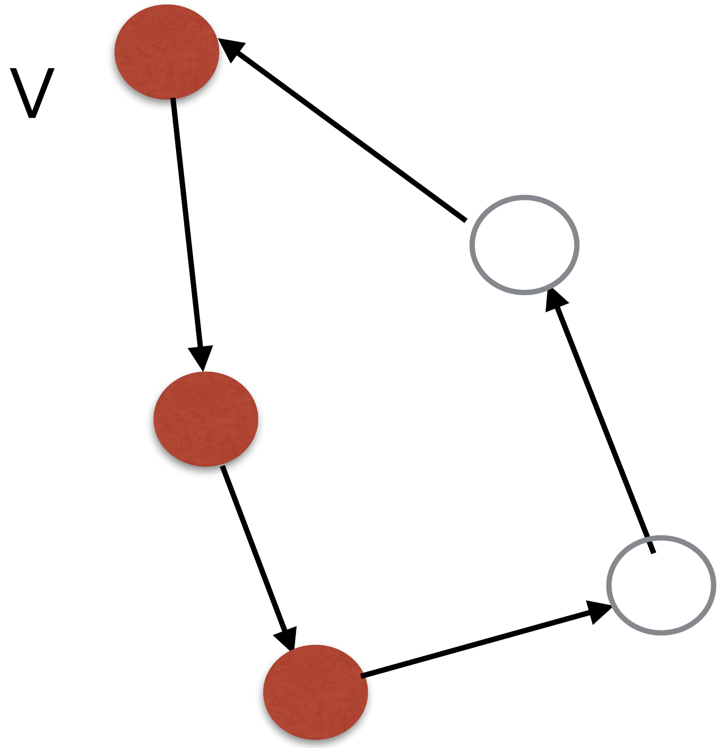
a vertex can be
NEW, ACTIVE or DONE

directed cycle if and only if
I reach an ACTIVE vertex
from an ACTIVE vertex

Is it a DAG?



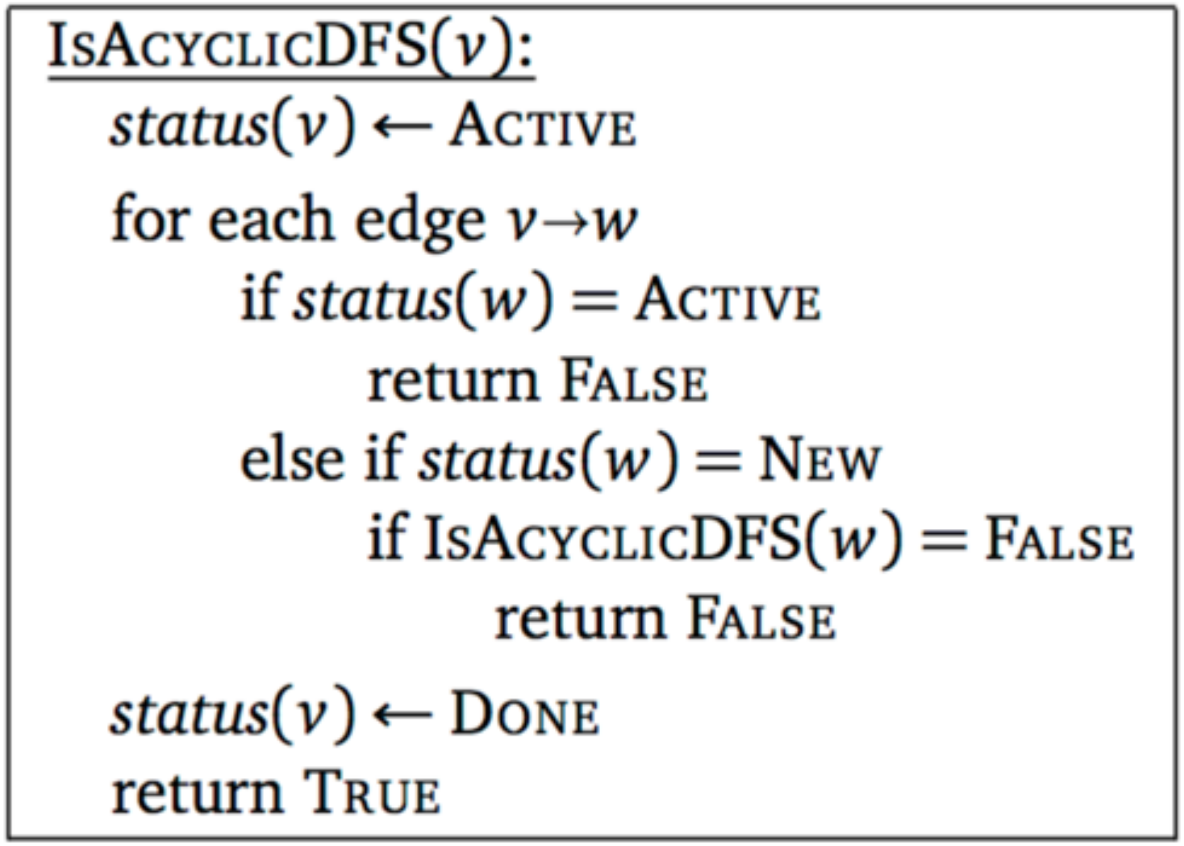
```
IsAcyclicDFS(v):
    status(v) ← Active
    for each edge v→w
        if status(w) = Active
            return False
        else if status(w) = New
            if IsAcyclicDFS(w) = False
                return False
    status(v) ← Done
    return True
```



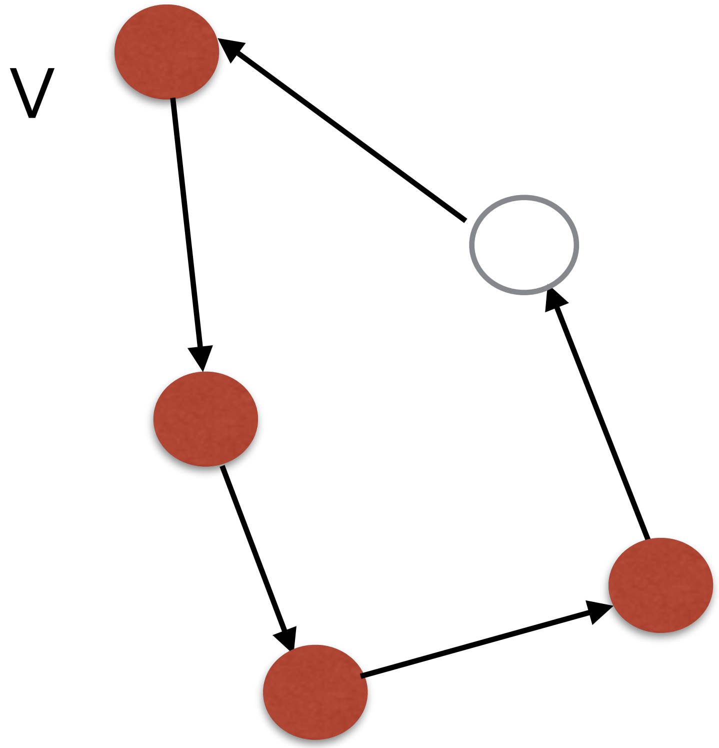
a vertex can be
NEW, ACTIVE or DONE

directed cycle if and only if
I reach an ACTIVE vertex
from an ACTIVE vertex

Is it a DAG?



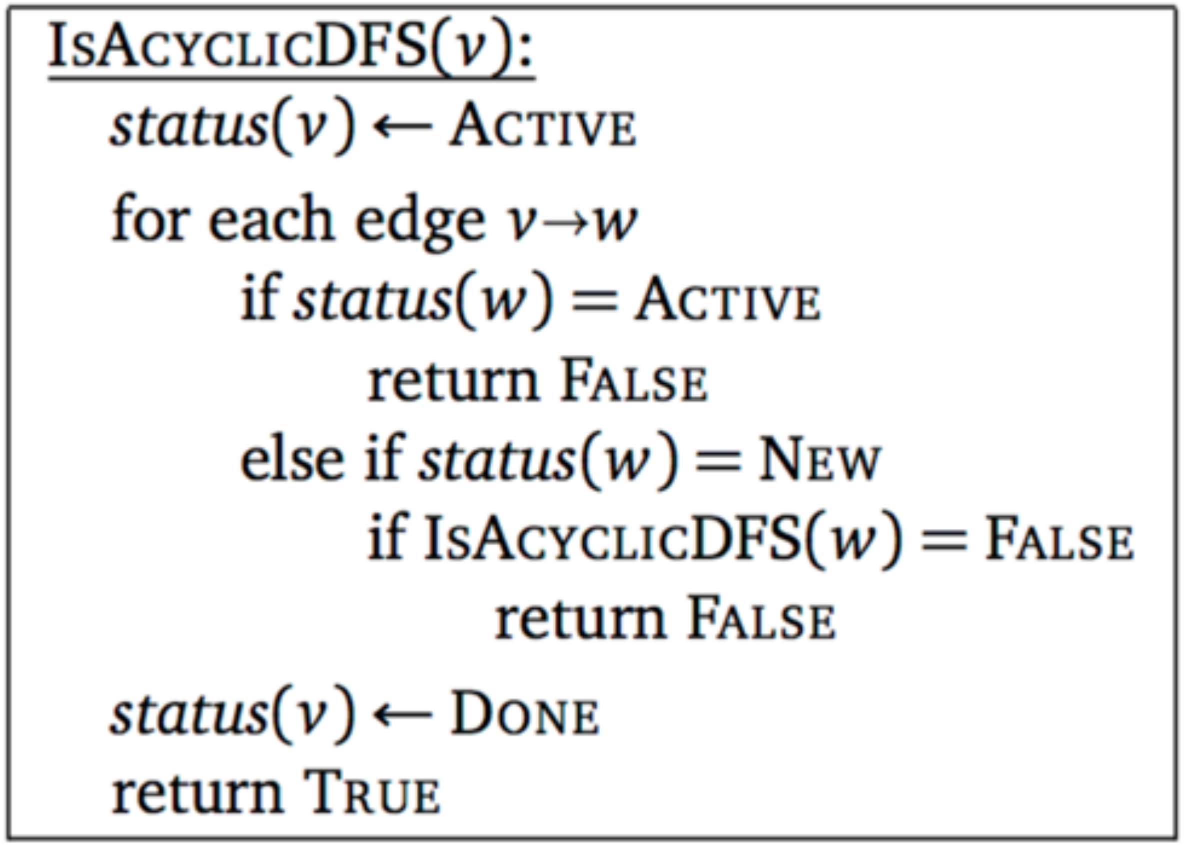
```
IsAcyclicDFS(v):
    status(v) ← Active
    for each edge v→w
        if status(w) = Active
            return False
        else if status(w) = New
            if IsAcyclicDFS(w) = False
                return False
    status(v) ← Done
    return True
```



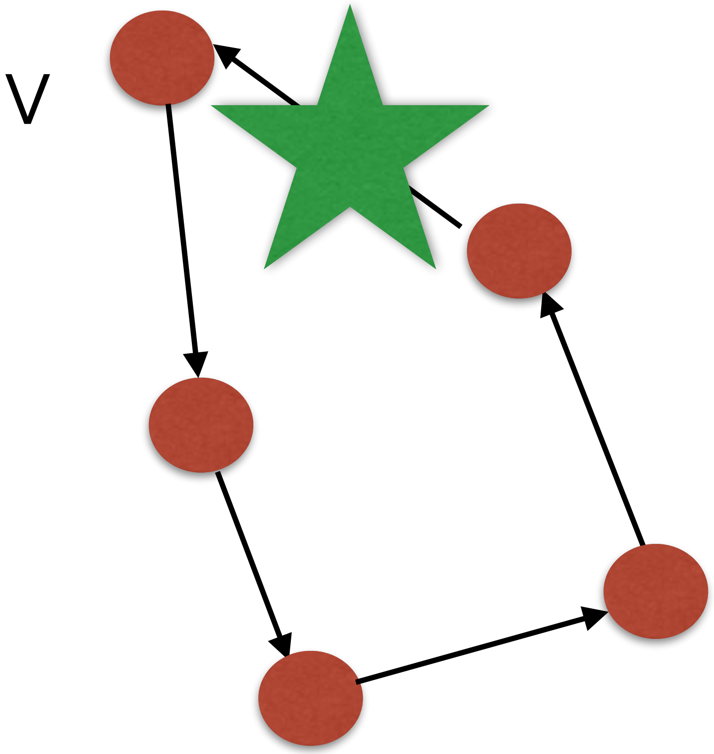
a vertex can be
NEW, ACTIVE or DONE

directed cycle if and only if
I reach an ACTIVE vertex
from an ACTIVE vertex

Is it a DAG?



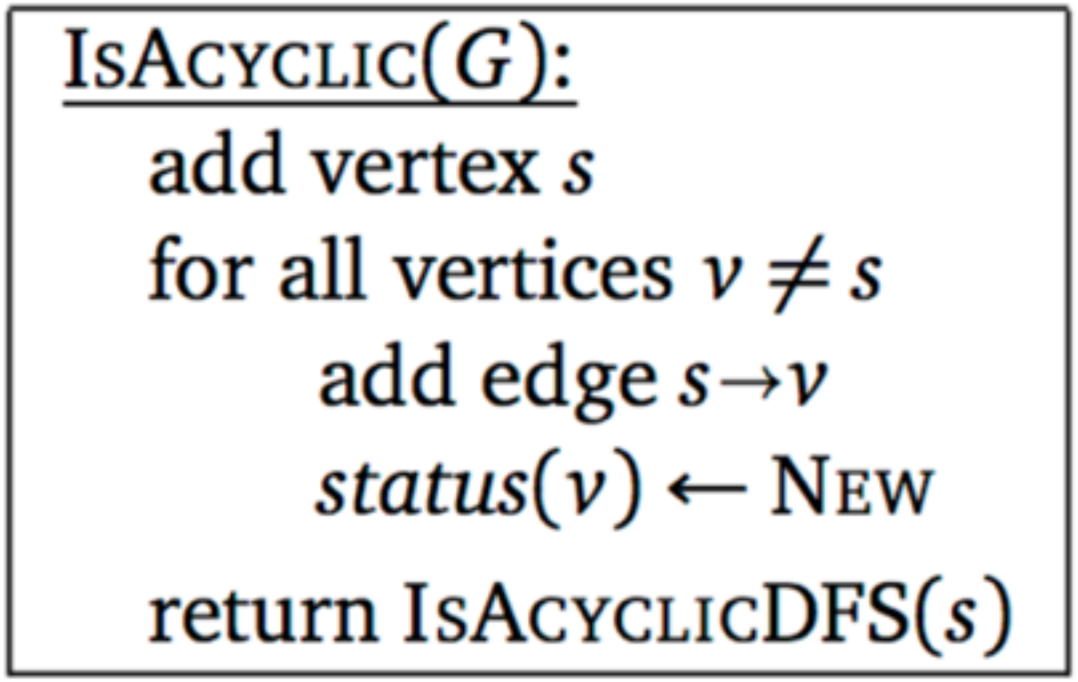
```
IsAcyclicDFS($v$):
    $status(v) \leftarrow$ Active
    for each edge $v \rightarrow w$
        if $status(w) =$ Active
            return False
        else if $status(w) =$ New
            if IsAcyclicDFS($w$) = False
                return False
    $status(v) \leftarrow$ Done
    return True
```



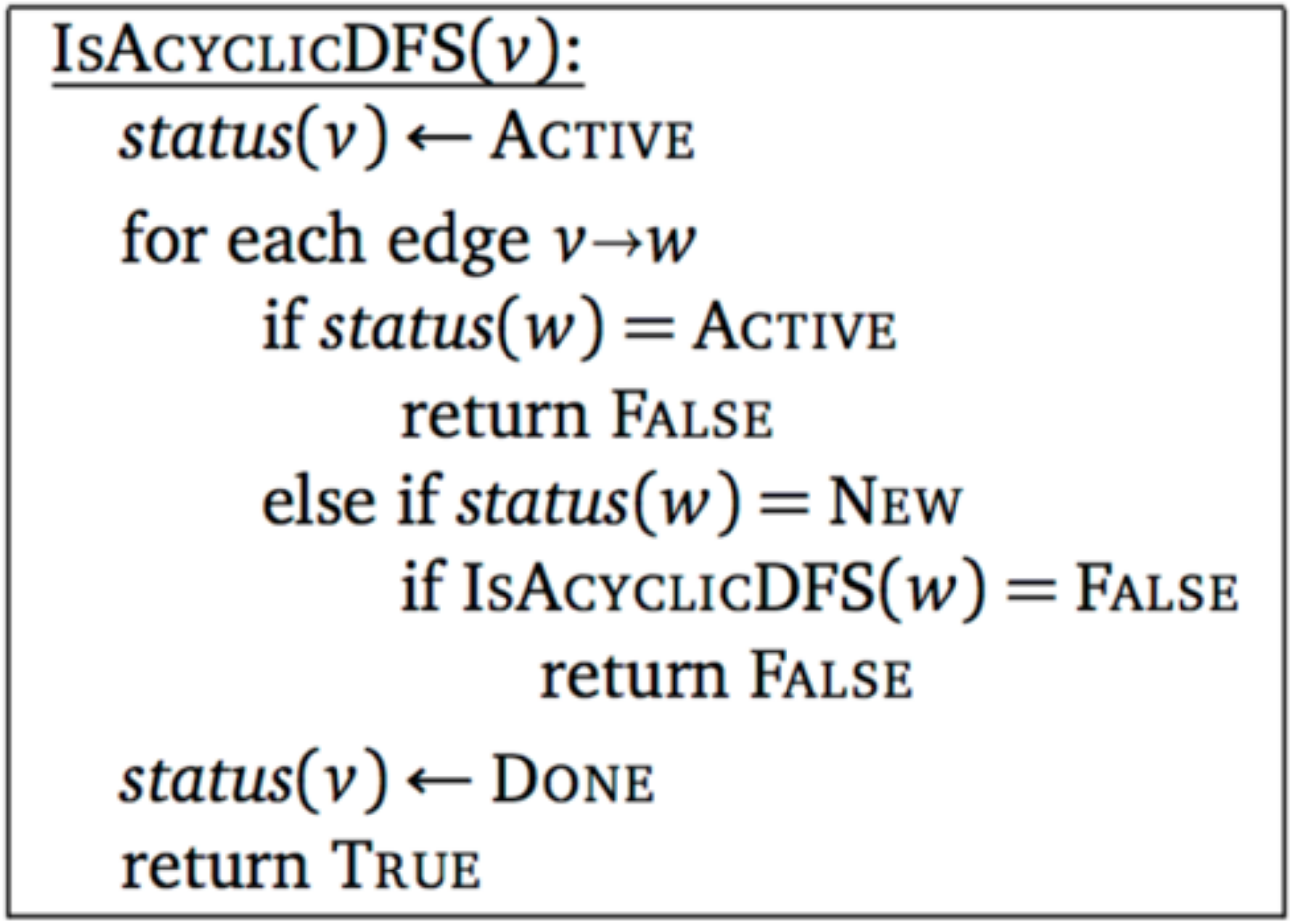
a vertex can be
NEW, ACTIVE or DONE

directed cycle if and only if
I reach an ACTIVE vertex
from an ACTIVE vertex

Is it a DAG?



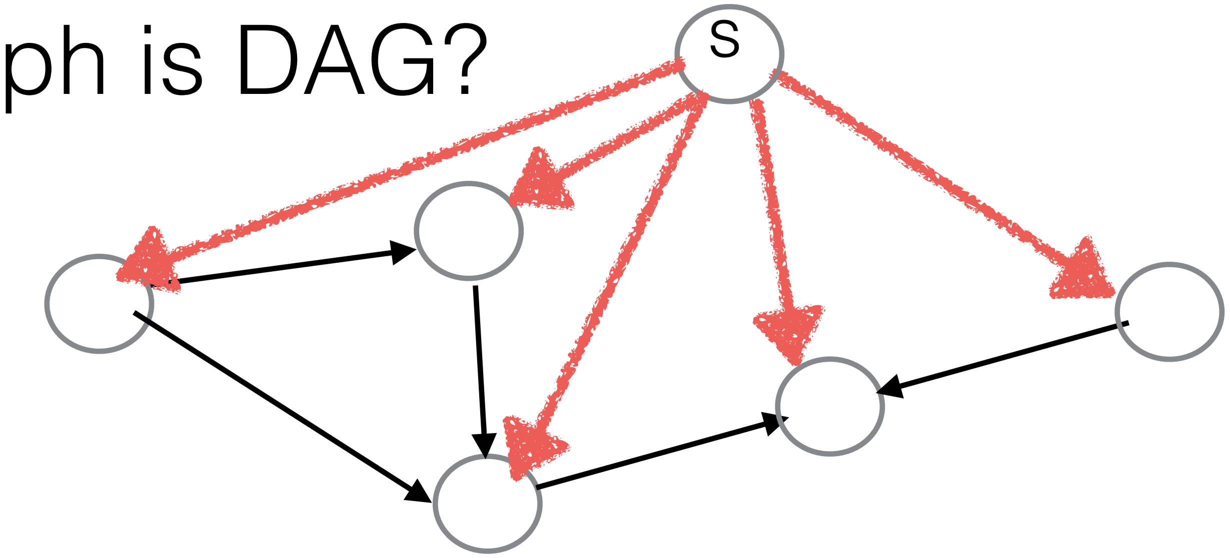
```
IsAcyclic(G):
    add vertex s
    for all vertices $v \neq s$
        add edge $s \to v$
        $status(v) \leftarrow$ New
    return IsAcyclicDFS(s)
```



```
IsAcyclicDFS(v):
    $status(v) \leftarrow$ Active
    for each edge $v \to w$
        if $status(w) =$ Active
            return False
        else if $status(w) =$ New
            if IsAcyclicDFS(w) = False
                return False
    $status(v) \leftarrow$ Done
    return True
```

time O(|V|+|E|)

Why do I want to decide if graph is DAG?

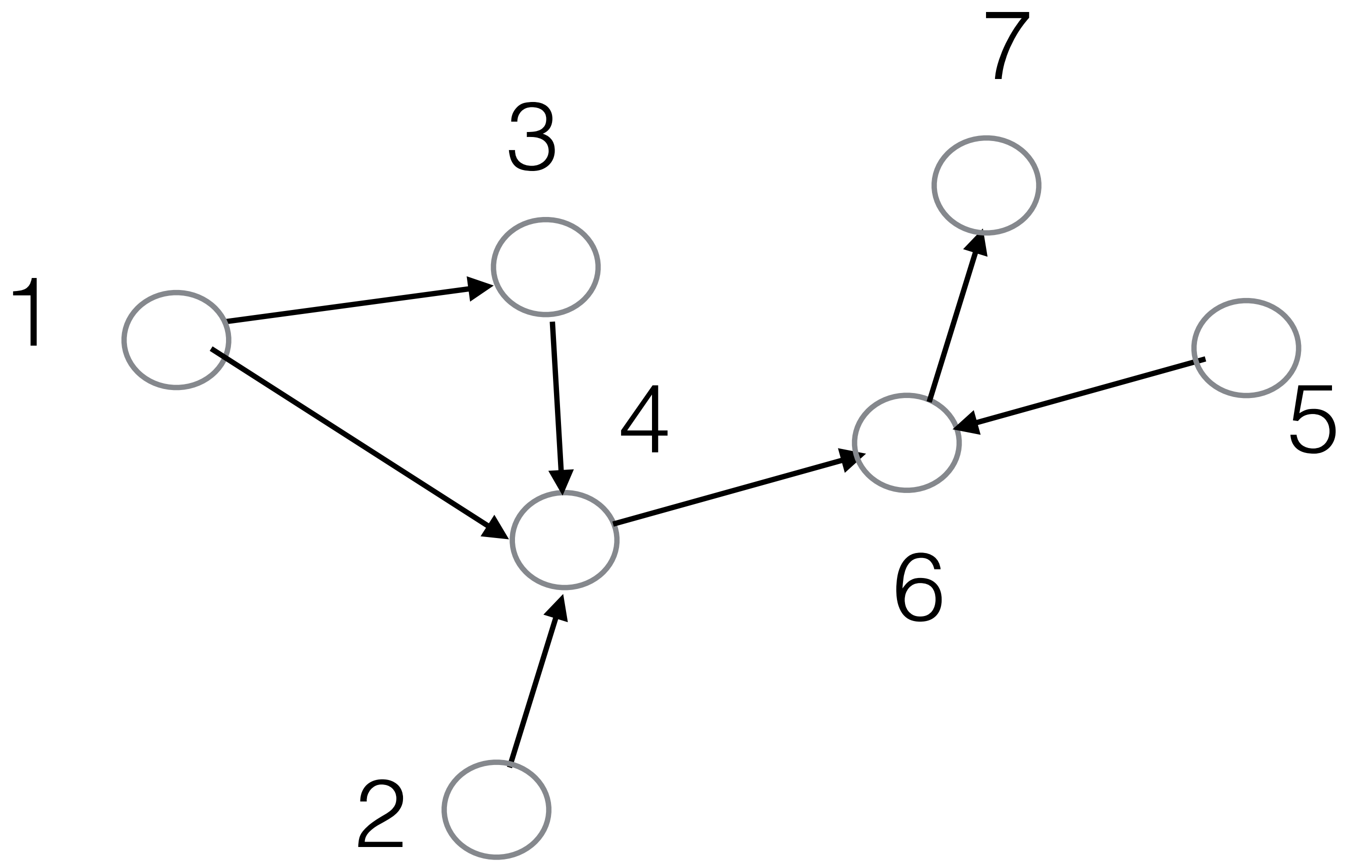


Why do I want to find if DAG?

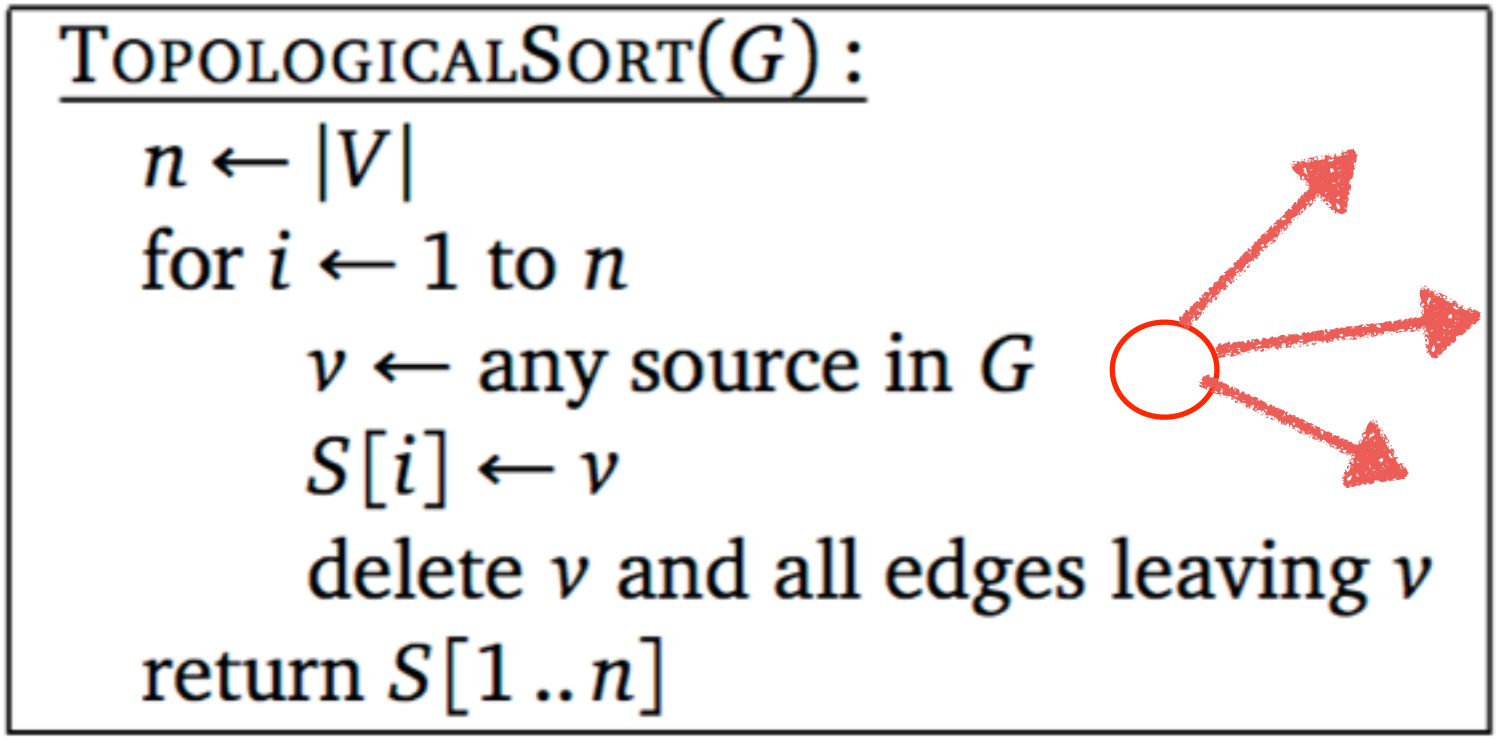
Make:

- huge database of files
- nodes are files,
- edges between files x and y: ifI change file x, i need to recompile file y
- Sometimes people create file system that have cycles!
- Make has to find those cycles and prevent that.
- It also has to execute the compilation commands in the correct order in order to produce the final executable.
- Just DFS of dependency graph
- See DP memoization

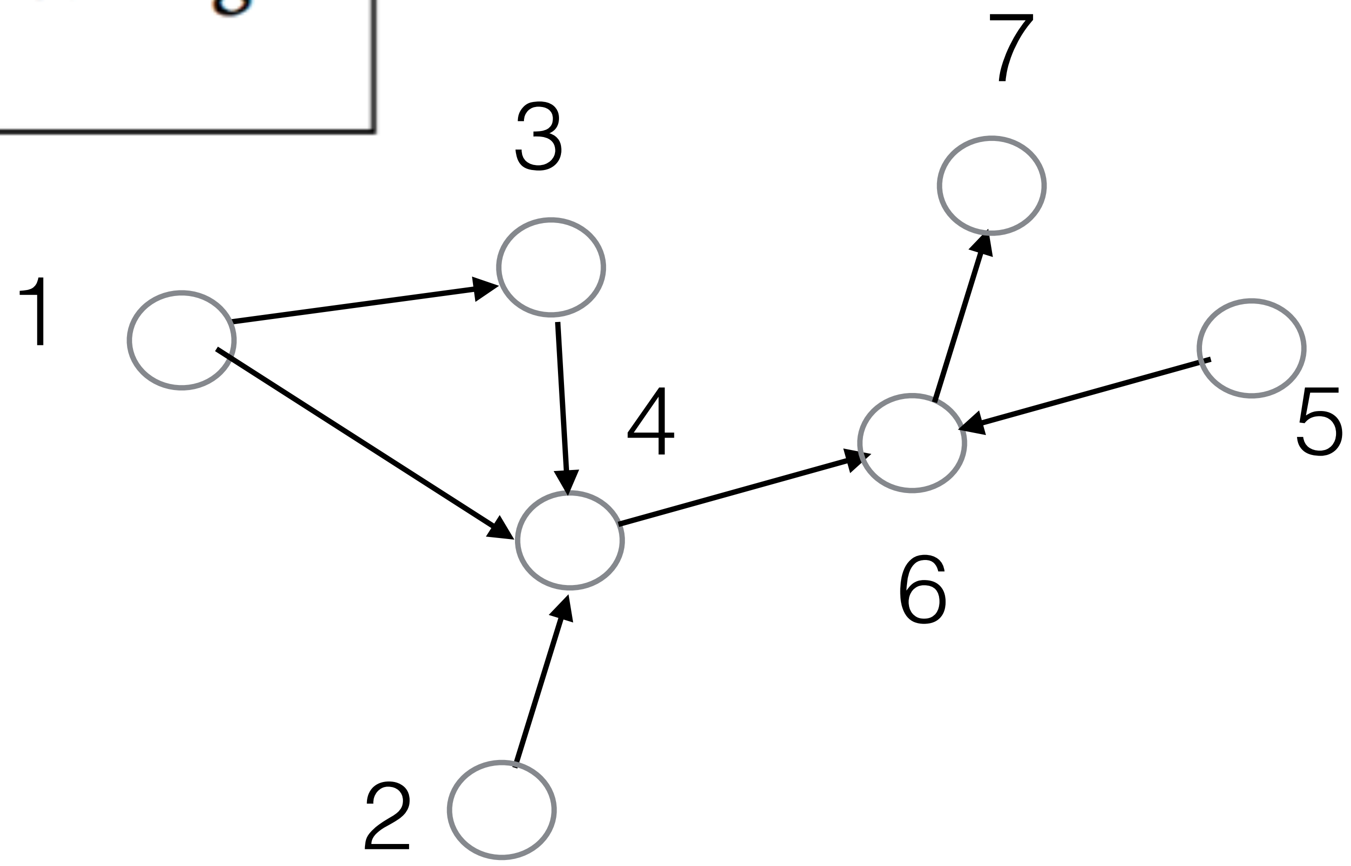
Topological Sort



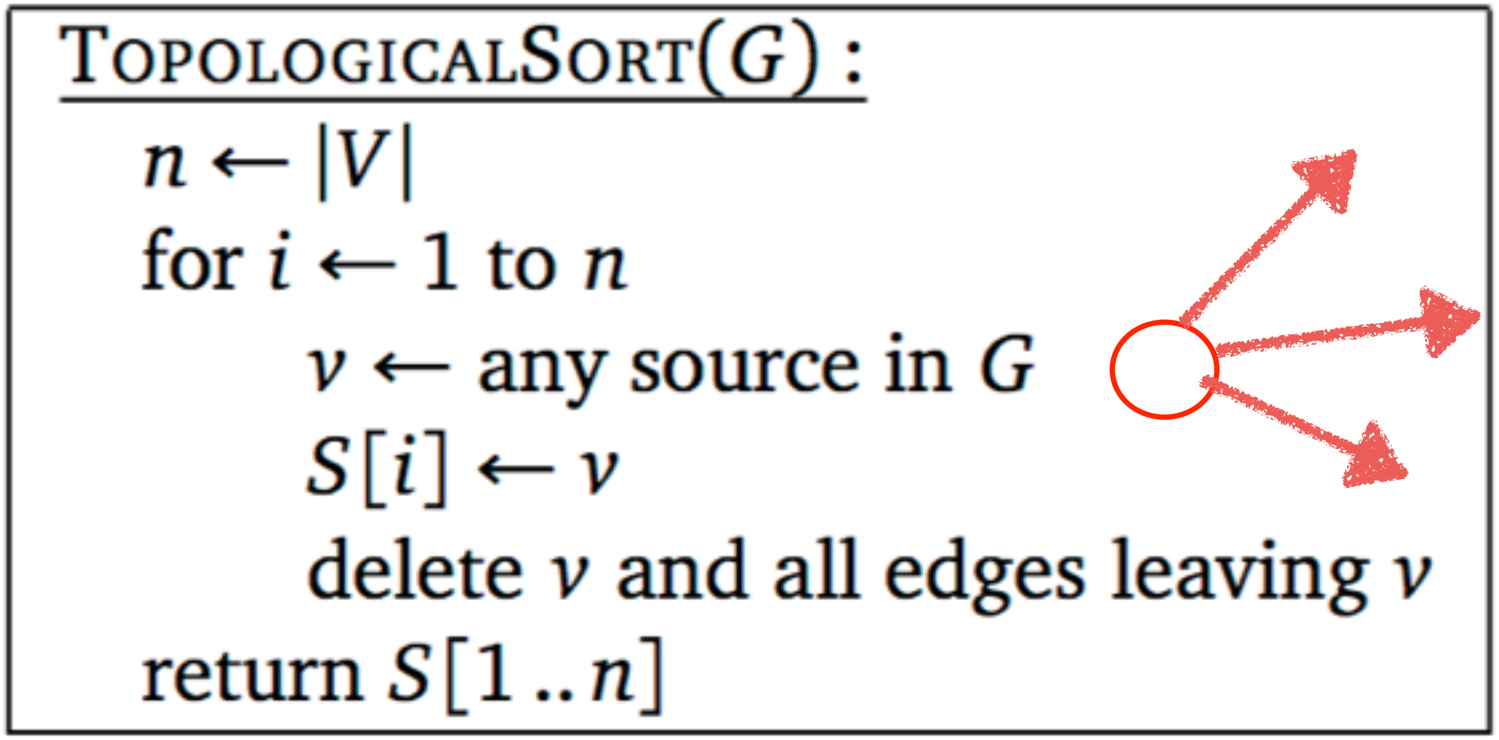
Topological Sort



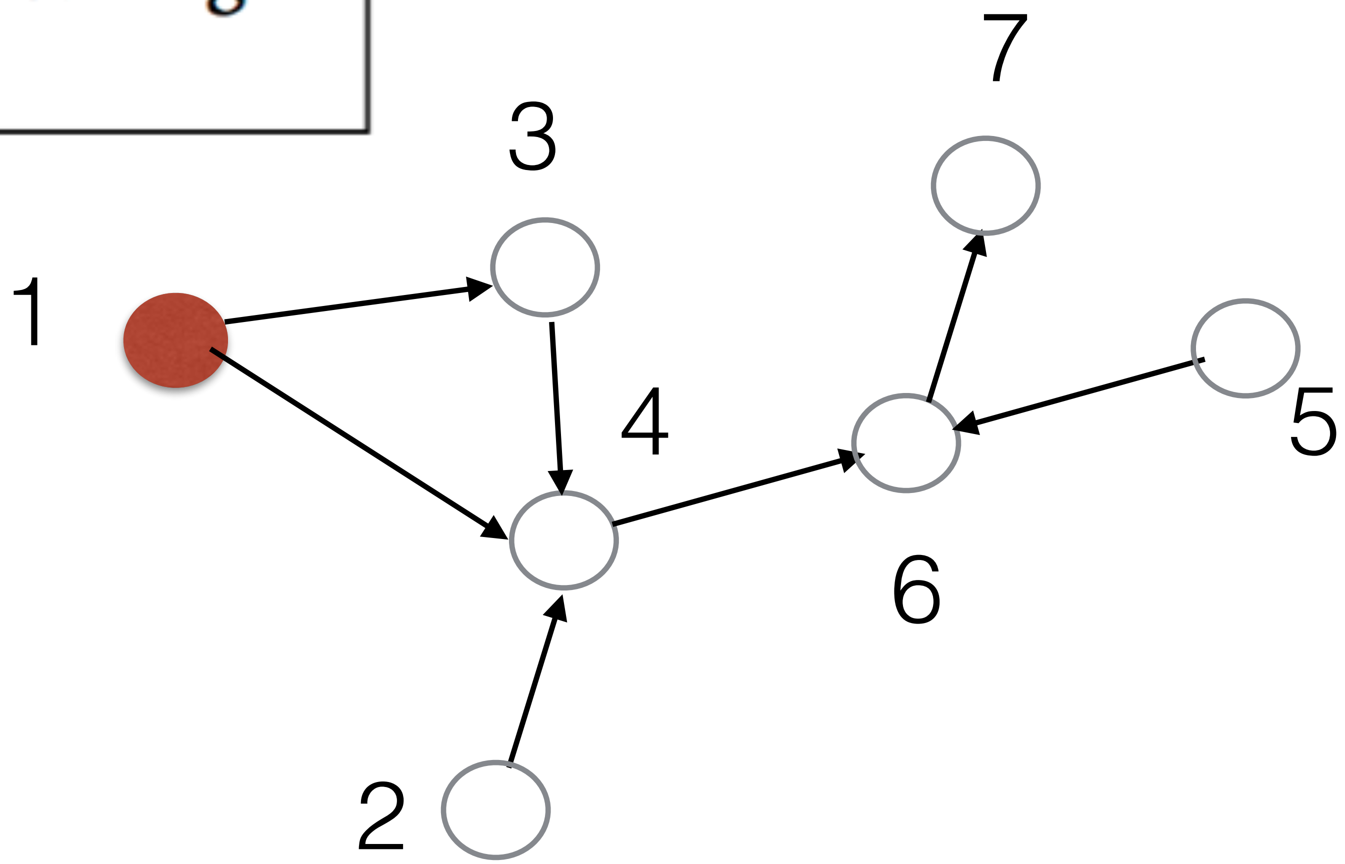
```
TopologicalSort(G):
    $n \leftarrow |V|$
    for $i \leftarrow 1$ to $n$
        $v \leftarrow$ any source in $G$
        $S[i] \leftarrow v$
        delete $v$ and all edges leaving $v$
    return $S[1..n]$
```



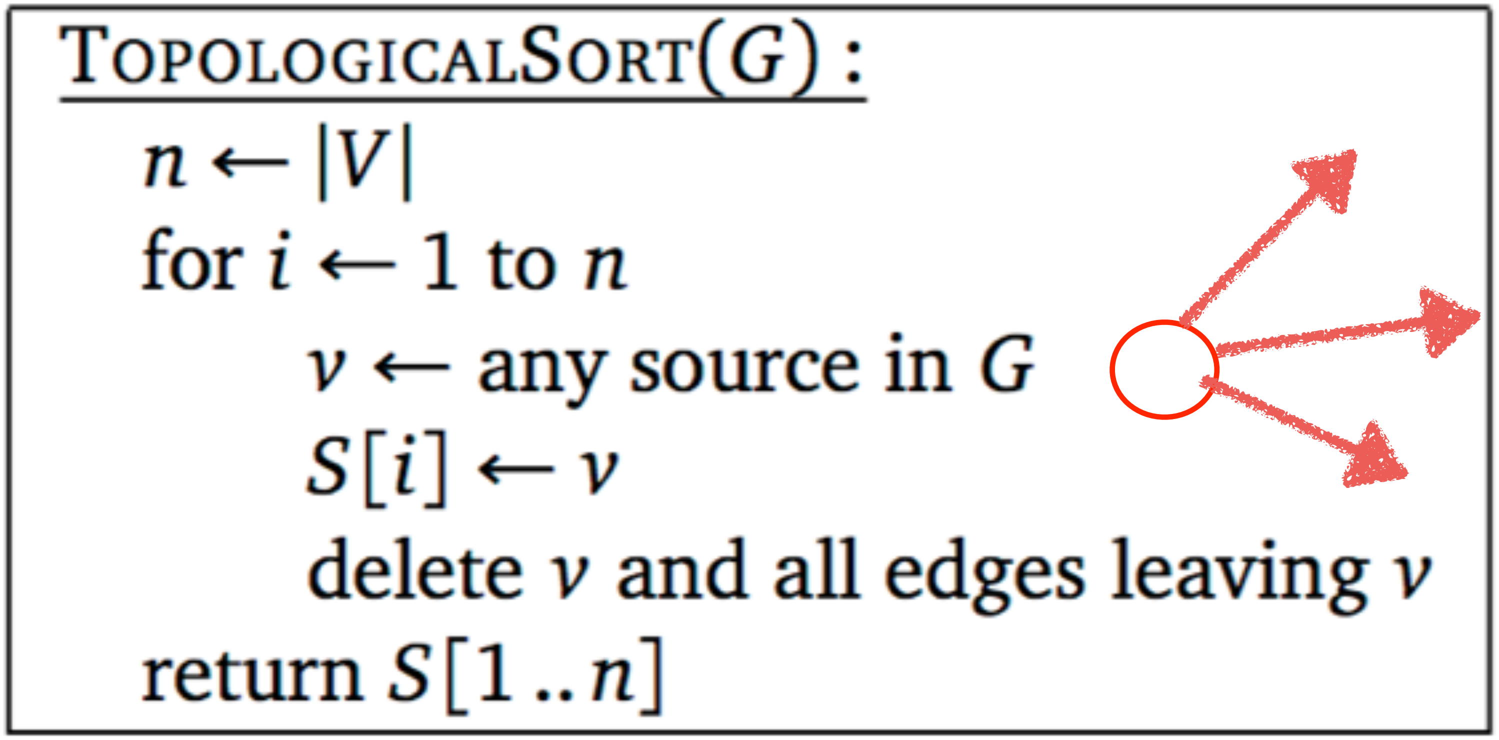
Topological Sort



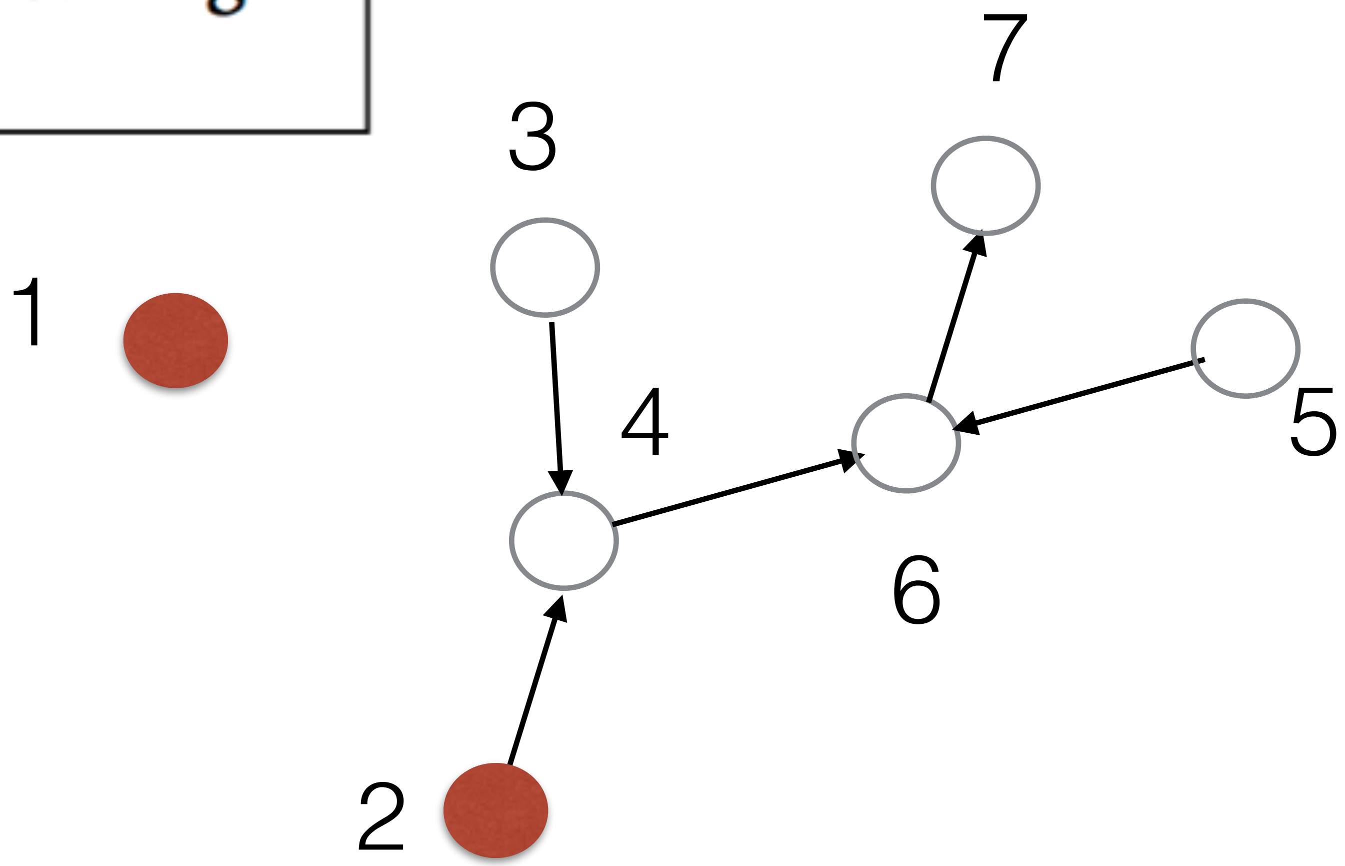
```
TopologicalSort(G):
  $n \leftarrow |V|$
  for $i \leftarrow 1$ to $n$
      $v \leftarrow$ any source in $G$
      $S[i] \leftarrow v$
      delete $v$ and all edges leaving $v$
  return $S[1..n]$
```



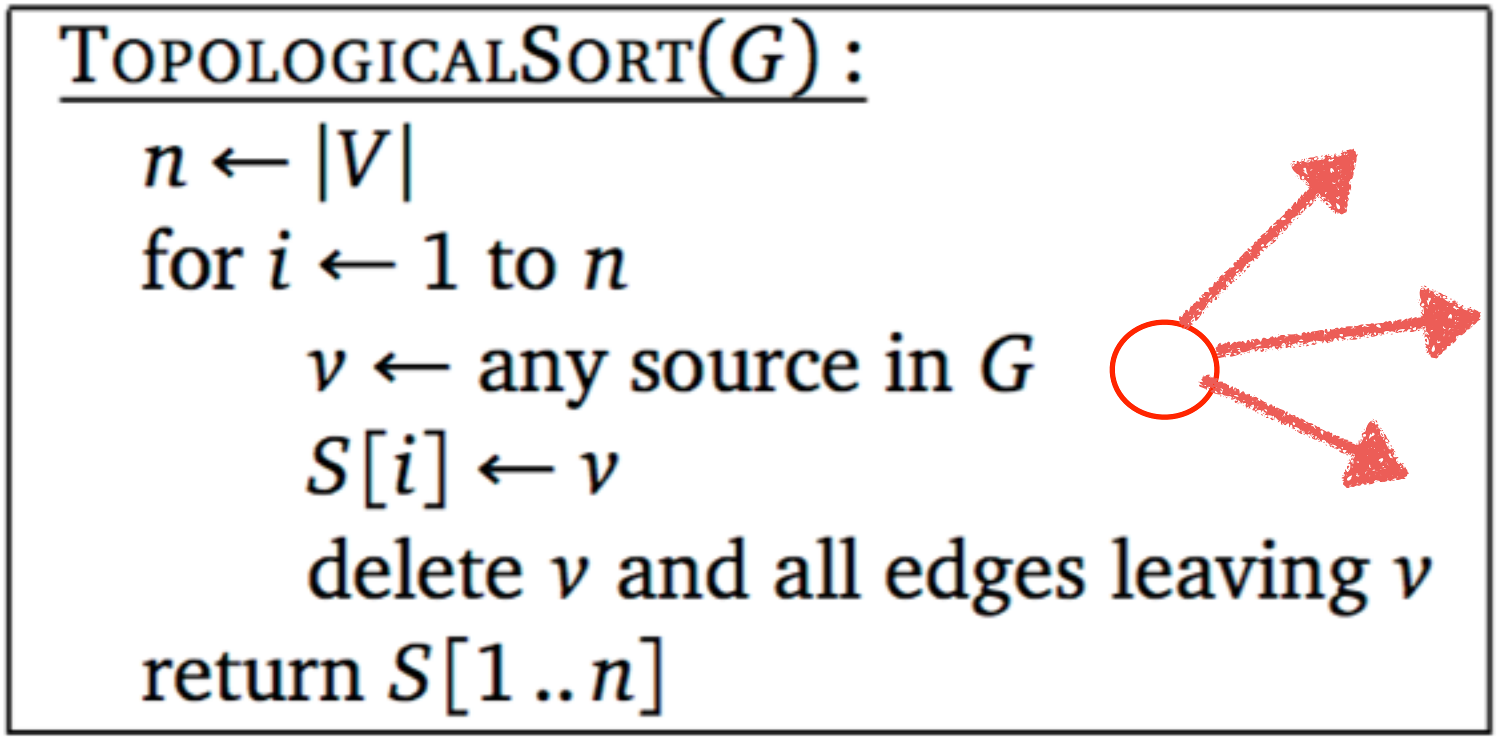
Topological Sort



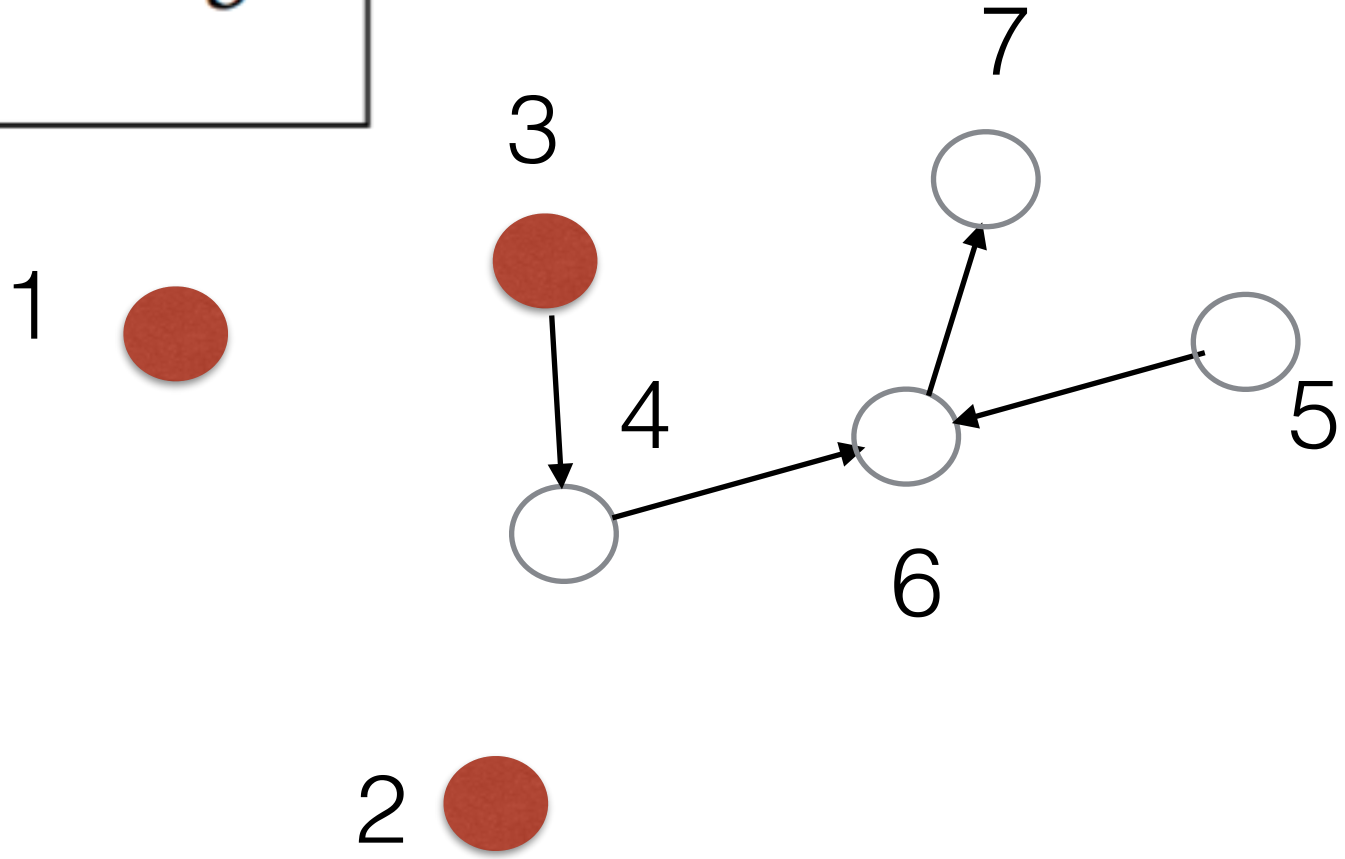
```
TOPOLOGICALSORT(G):
    $n \leftarrow |V|$
    for $i \leftarrow 1$ to $n$
        $v \leftarrow$ any source in $G$
        $S[i] \leftarrow v$
        delete $v$ and all edges leaving $v$
    return $S[1..n]$
```



Topological Sort

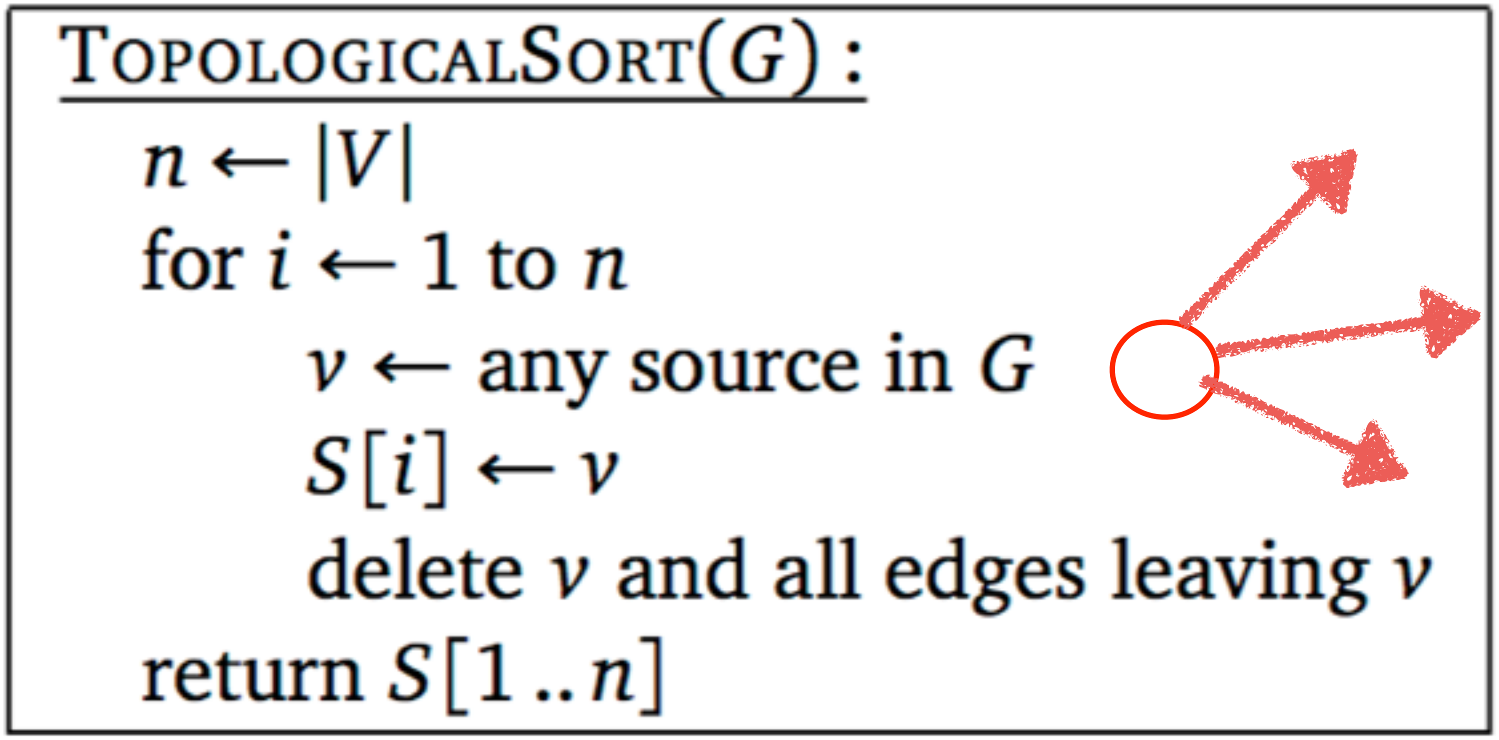


```
TopologicalSort($G$):
    $n \leftarrow |V|$
    for $i \leftarrow 1$ to $n$
        $v \leftarrow$ any source in $G$
        $S[i] \leftarrow v$
        delete $v$ and all edges leaving $v$
    return $S[1..n]$
```

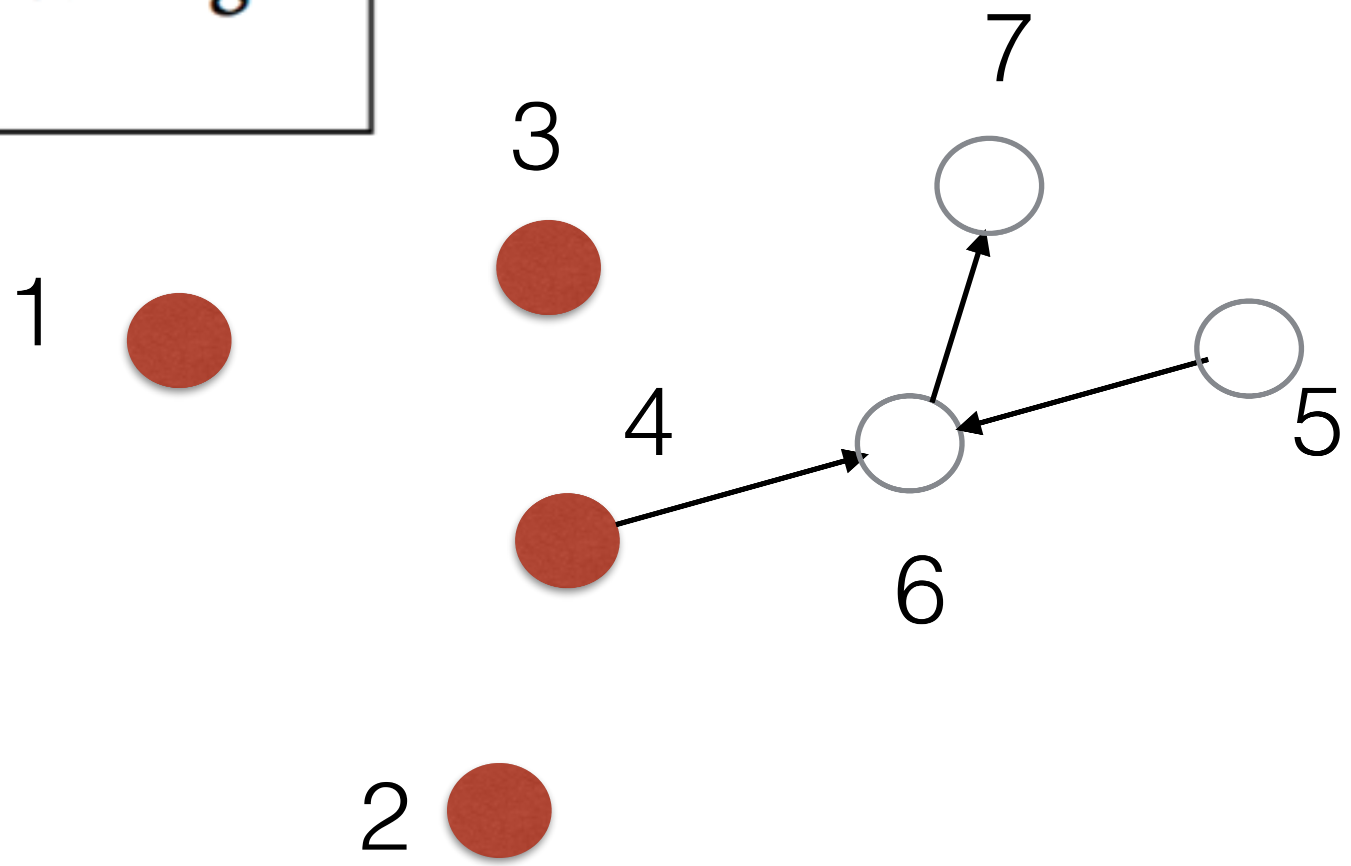


CS 374

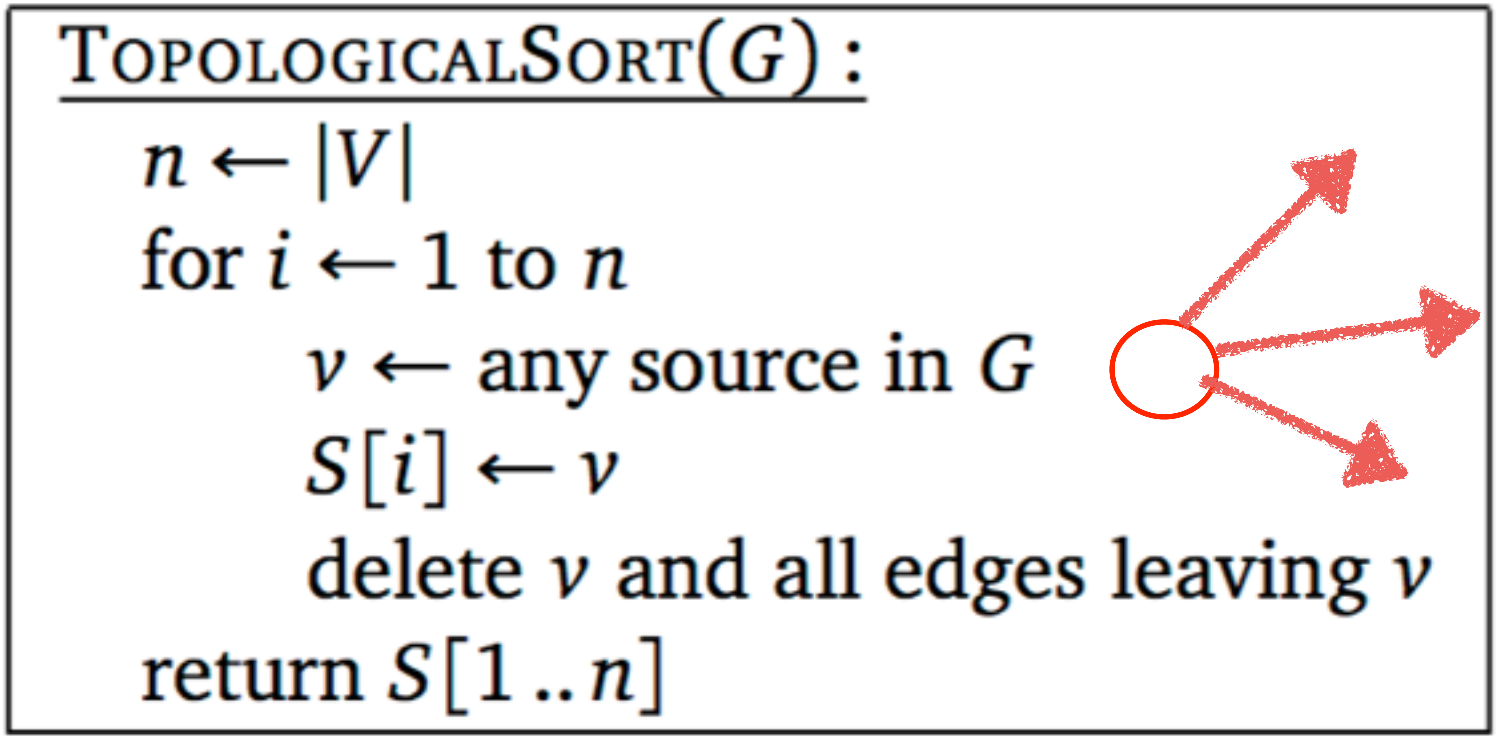
Topological Sort



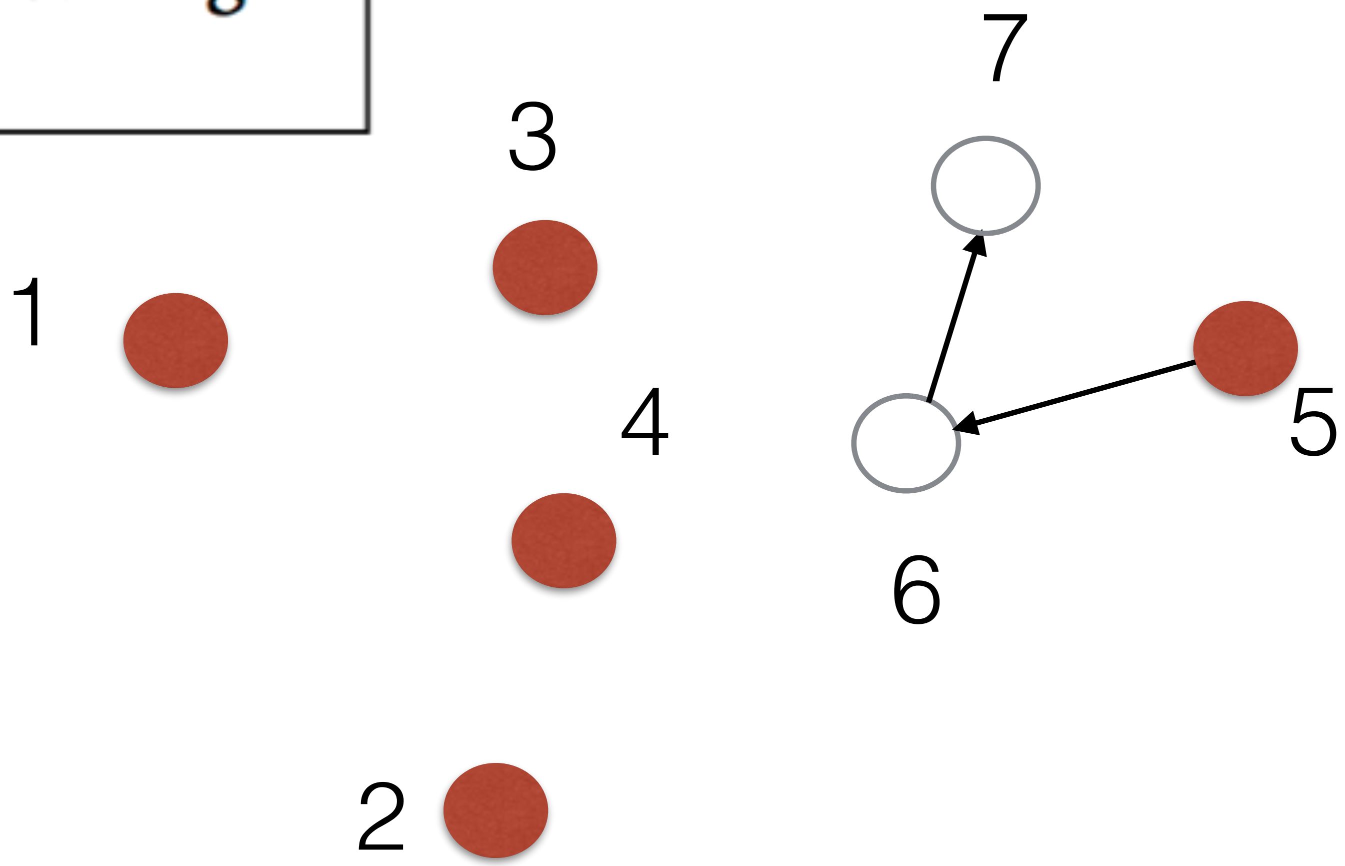
```
TOPOLOGICALSORT($G$):
    $n \leftarrow |V|$
    for $i \leftarrow 1$ to $n$
        $v \leftarrow$ any source in $G$
        $S[i] \leftarrow v$
        delete $v$ and all edges leaving $v$
    return $S[1..n]$
```



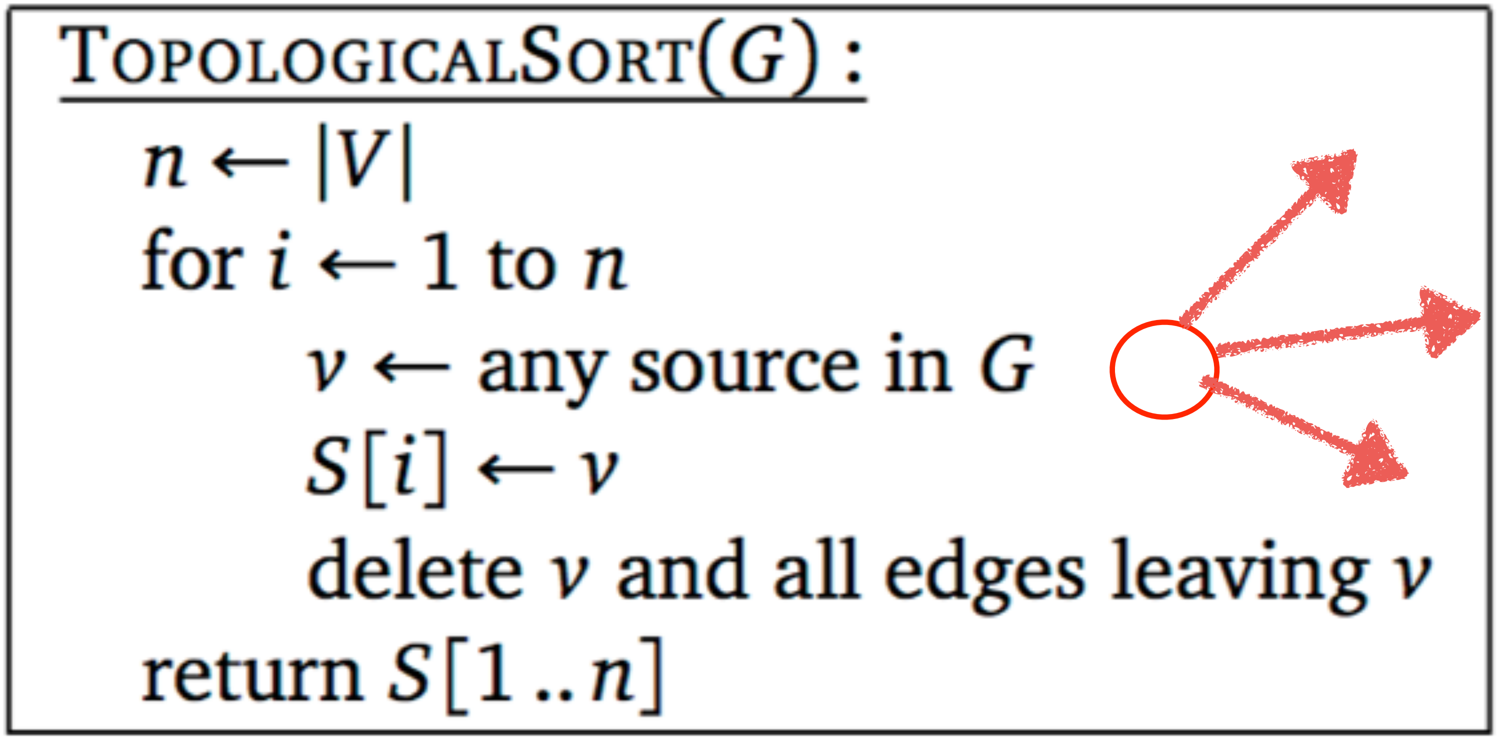
Topological Sort



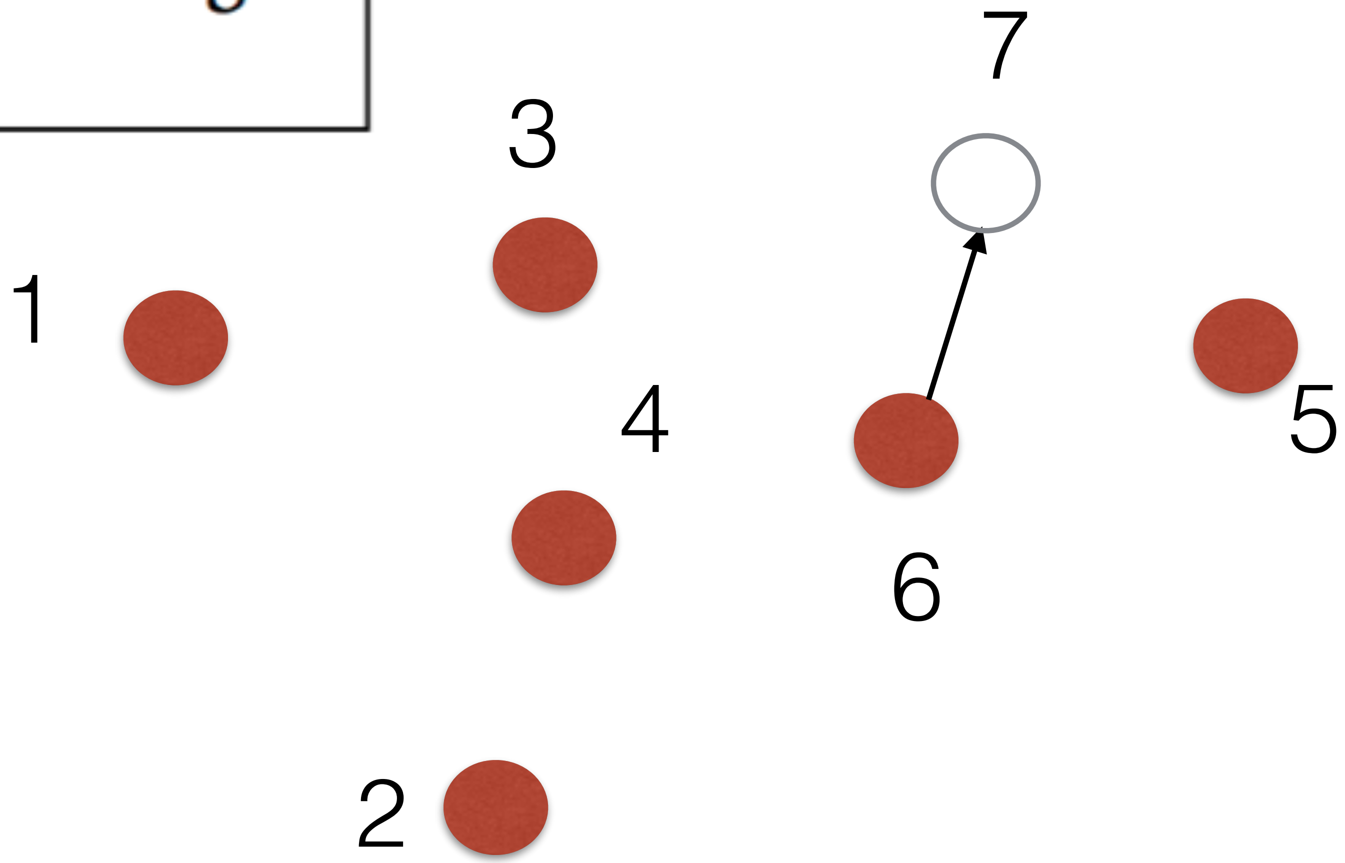
```
TopologicalSort(G):
    $n \leftarrow |V|$
    for $i \leftarrow 1$ to $n$
        $v \leftarrow$ any source in $G$
        $S[i] \leftarrow v$
        delete $v$ and all edges leaving $v$
    return $S[1..n]$
```



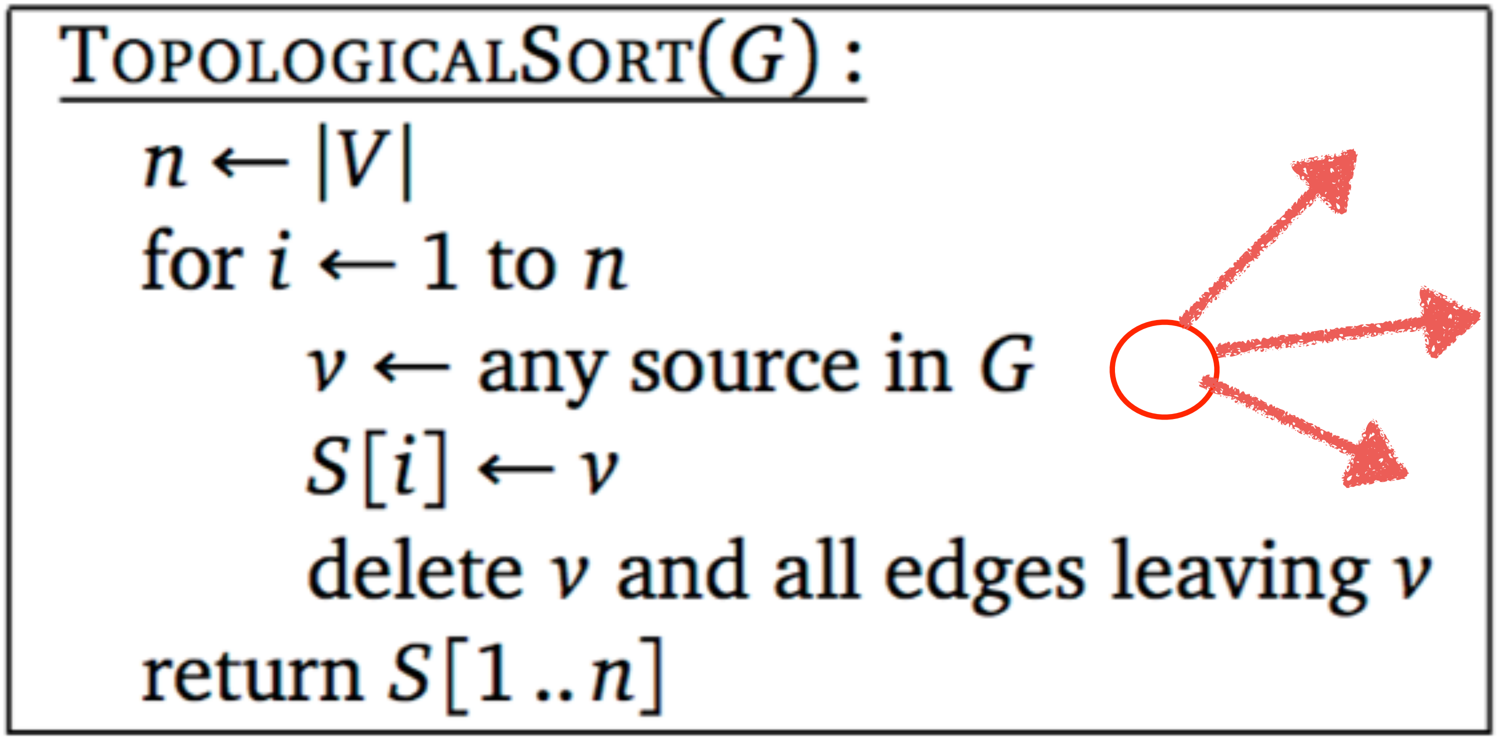
Topological Sort



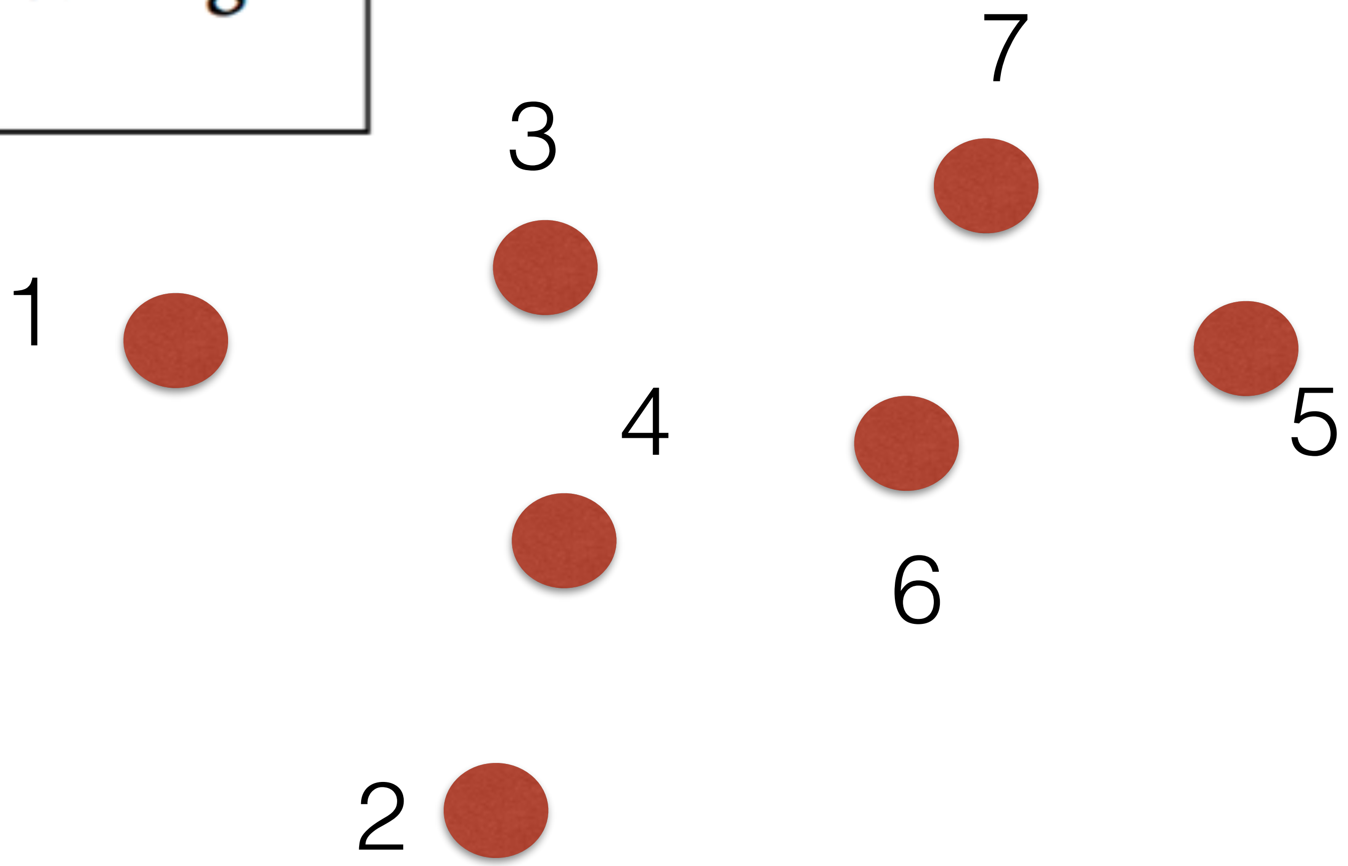
```
TopologicalSort(G):
    $n \leftarrow |V|$
    for $i \leftarrow 1$ to $n$
        $v \leftarrow$ any source in $G$
        $S[i] \leftarrow v$
        delete $v$ and all edges leaving $v$
    return $S[1..n]$
```



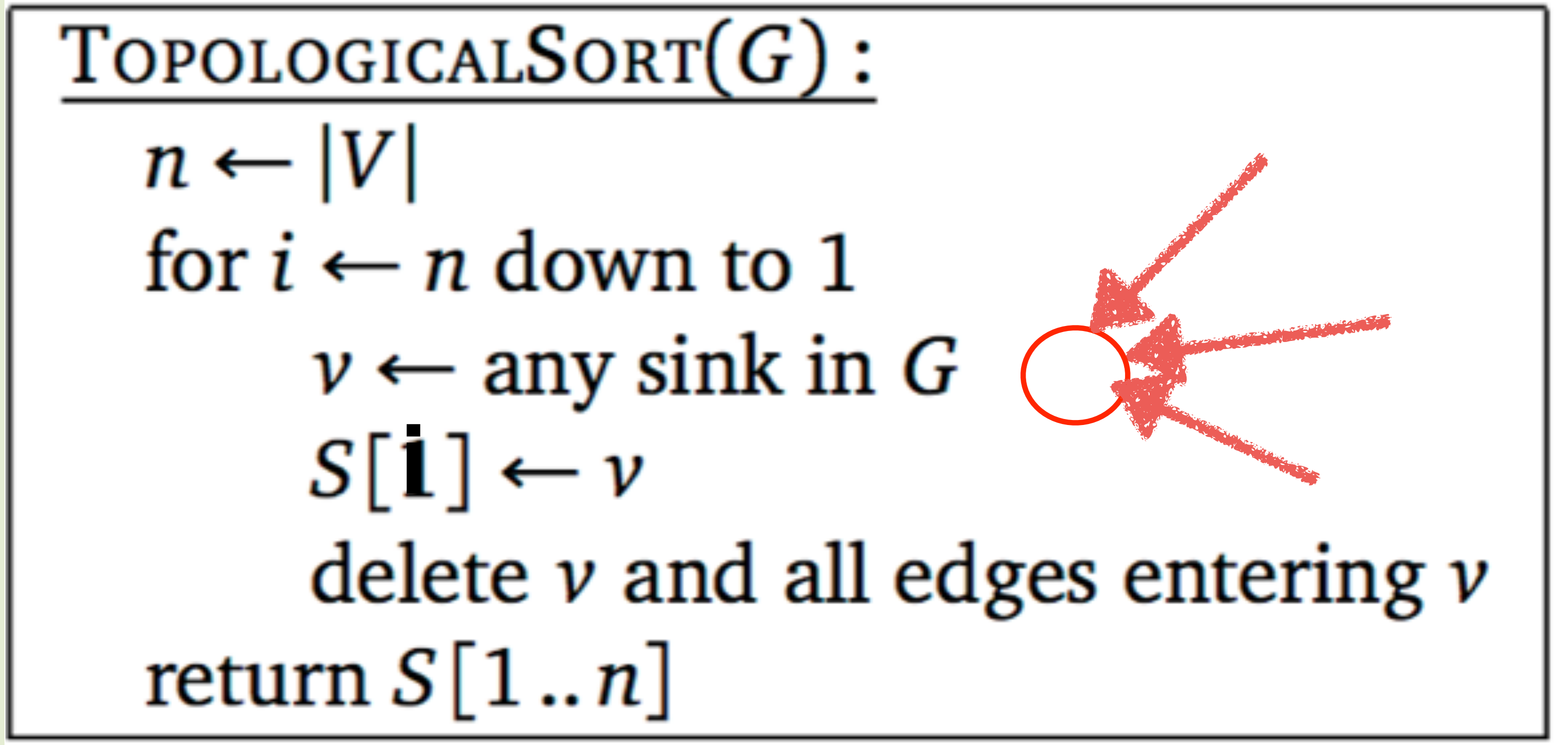
Topological Sort



```
TopologicalSort(G):
    $n \leftarrow |V|$
    for $i \leftarrow 1$ to $n$
        $v \leftarrow$ any source in $G$
        $S[i] \leftarrow v$
        delete $v$ and all edges leaving $v$
    return $S[1..n]$
```



Topological Sort



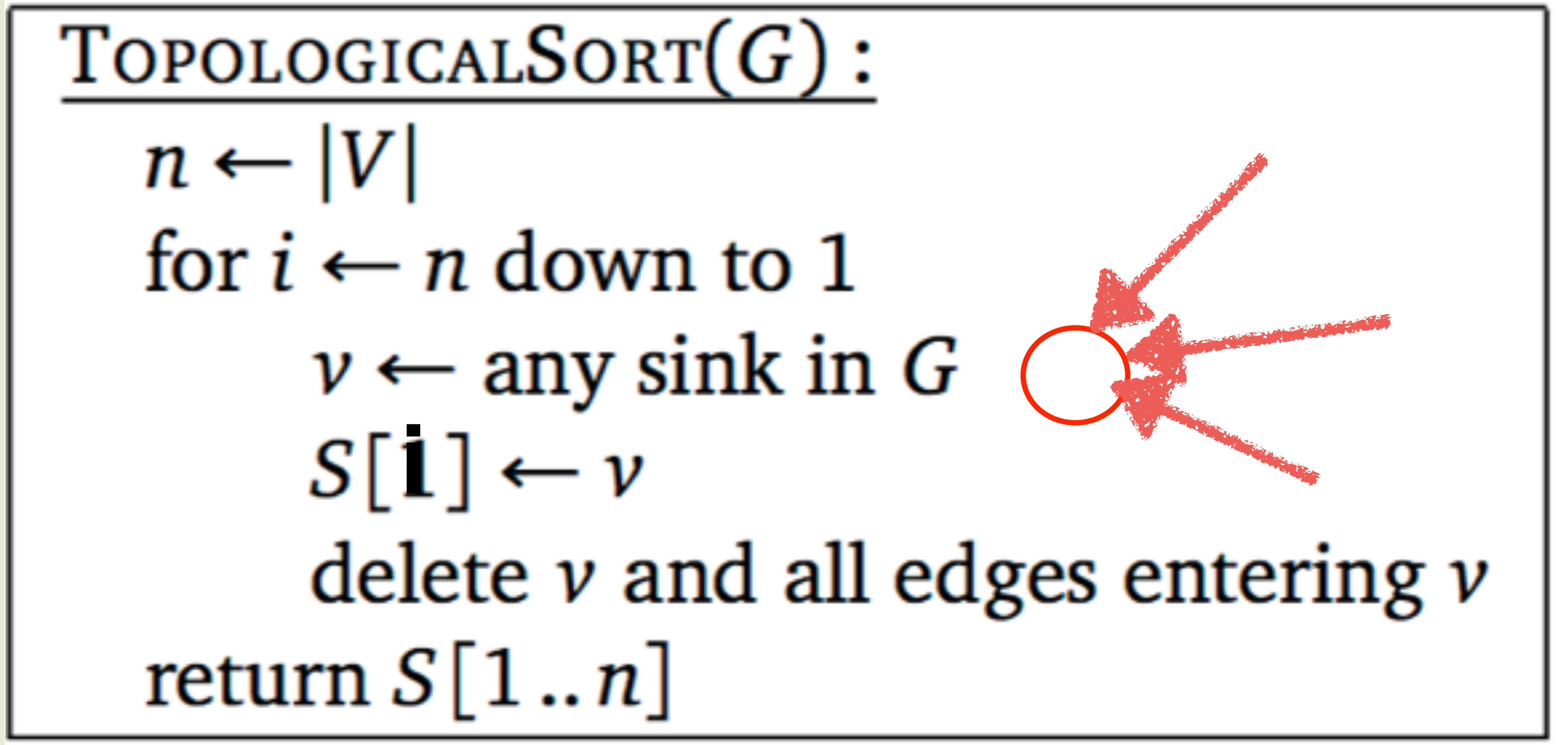
```
TopologicalSort(G):
    $n \leftarrow |V|$
    for $i \leftarrow n$ down to 1
        $v \leftarrow$ any sink in $G$
        $S[i] \leftarrow v$
        delete $v$ and all edges entering $v$
    return $S[1..n]$
```

Dependency graph of software

Running time?

- How to find sink?
- Naively O(n) for each sink, total $O(n^2)$
- For source, even worse, cause the adjacency list representation doesn't have pointers for incoming edge
- $O(n^2|E|)$ naively.

Topological Sort



```
TopologicalSort(G):
    $n \leftarrow |V|$
    for $i \leftarrow n$ down to 1
        $v \leftarrow$ any sink in $G$
        $S[i] \leftarrow v$
        delete $v$ and all edges entering $v$
    return $S[1..n]$
```

Dependency graph of software

Running time?

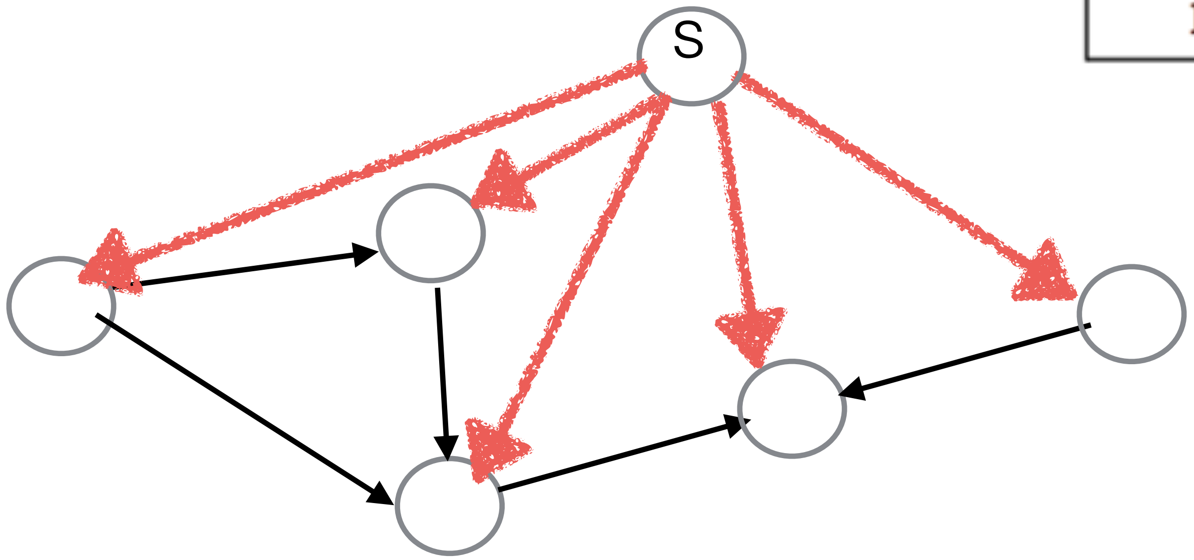
- Could do it with priority queue of out degrees in O(|V|+|E|).
- Reverse the dag to help delete edges etc…
- Is there another way?

Topological Sort

- **Claim**: First vertex DONE in DFS below is sink.

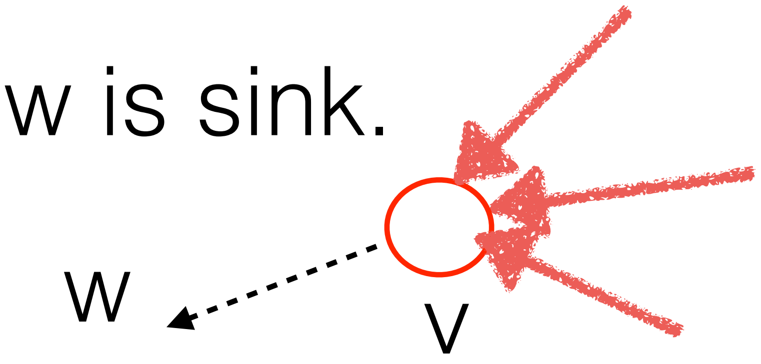
```
IsAcyclic(G):
    add vertex $s$
    for all vertices $v \neq s$
        add edge $s \to v$
        $status(v) \leftarrow$ New
    return IsAcyclicDFS($s$)
```

```
IsAcyclicDFS(v):
    $status(v) \leftarrow$ Active
    for each edge $v \to w$
        if $status(w) =$ Active
            return False
        else if $status(w) =$ New
            if IsAcyclicDFS($w$) = False
                return False
    $status(v) \leftarrow$ Done
    return True
```



Topological Sort

- **Claim**: First vertex DONE in DFS below is sink.
- **Proof**:

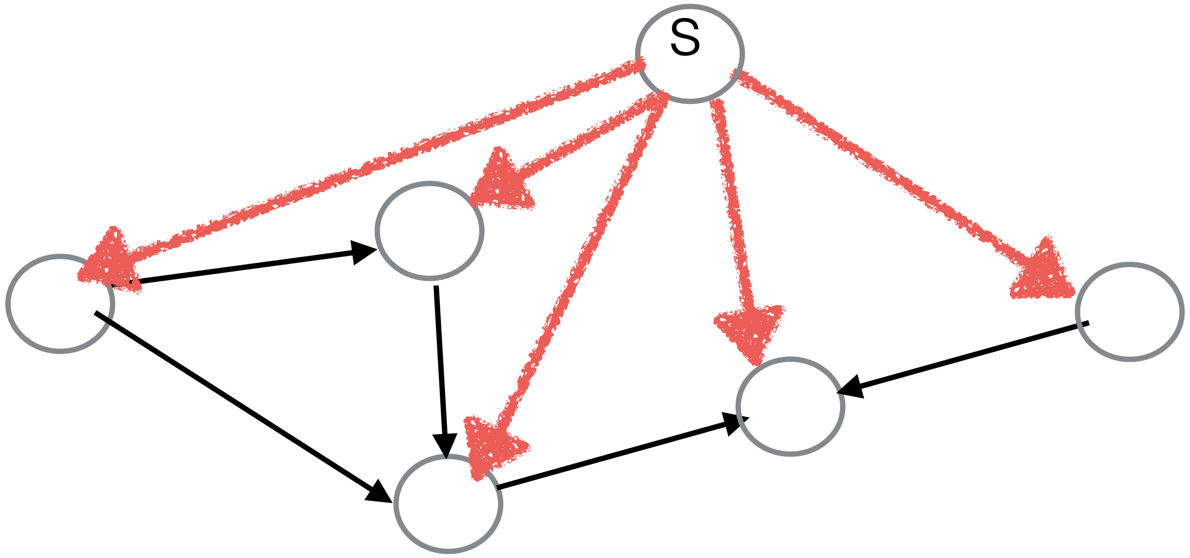


Assume, towards contradiction that v is DONE first

but there is w: v → w.

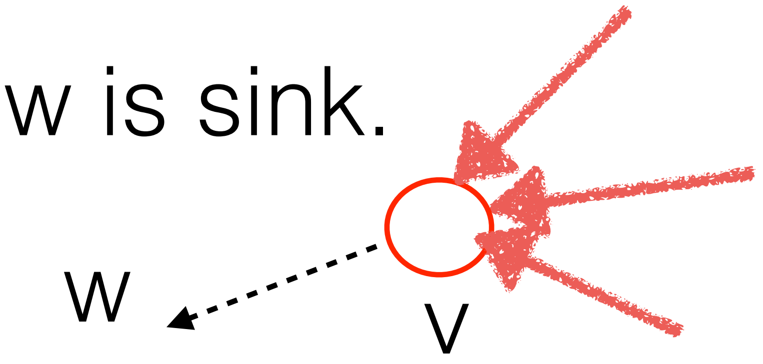
There are 3 cases:

- w is NEW
- w ACTIVE
- w DONE



Topological Sort

- **Claim**: First vertex DONE in DFS below is sink.



- **Proof**:

Assume, towards contradiction that v is DONE first

but there is w: v → w.

There are 3 cases:

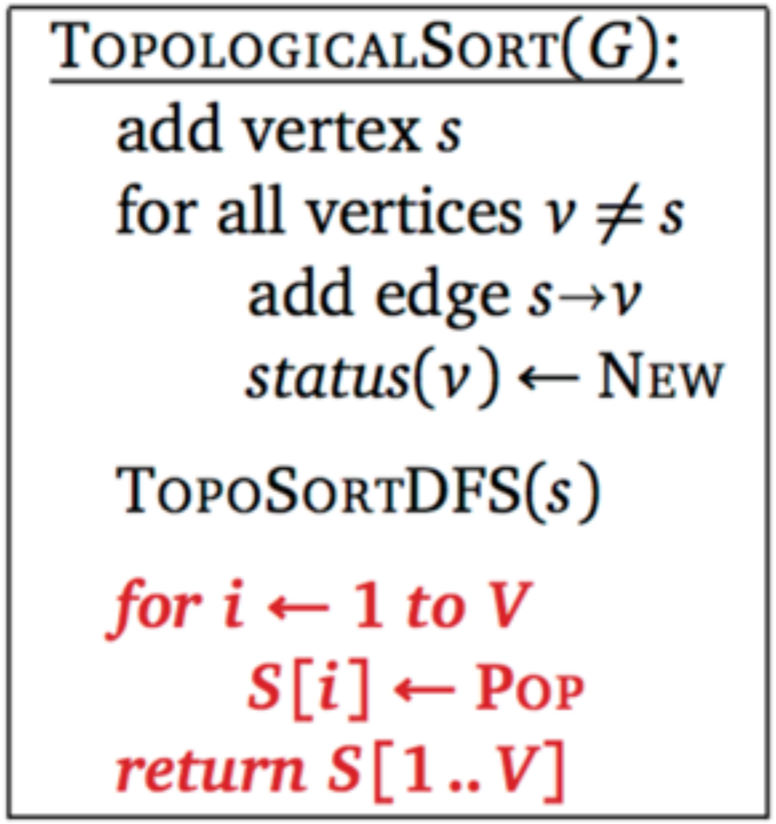
- w is NEW ✗ w would be marked active and then be DONE first (contradiction)
- w ACTIVE ✗ v is active still, so there is a cycle (contradiction of DAG)
- w DONE ✗

v is DONE first (contradiction)

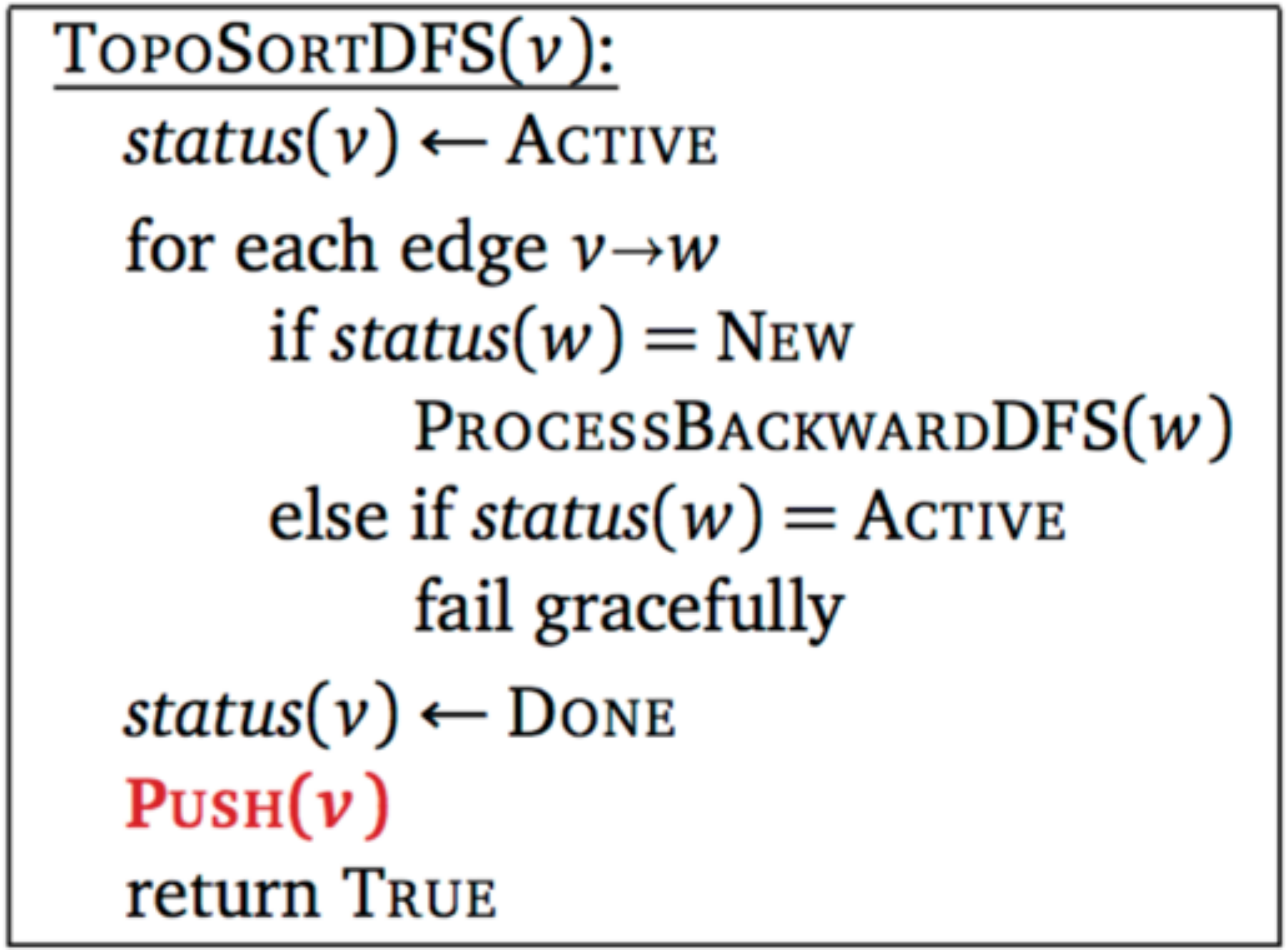
Topological Sort

- **Claim**: The order by which vertices are DONE in DFS is a reverse topological order (proof?).

- Can just sort those vertices in stack and pop them for topological order, put them in sorted array.



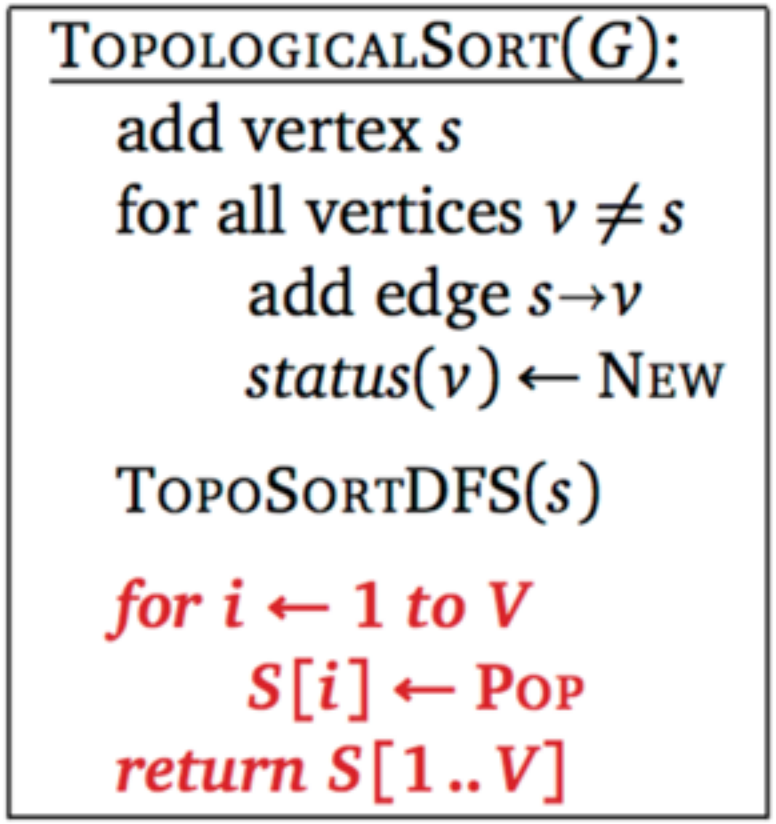
```
TopologicalSort(G):
    add vertex s
    for all vertices v ≠ s
        add edge s→v
        status(v) ← New
    TopoSortDFS(s)
    for i ← 1 to V
        S[i] ← Pop
    return S[1..V]
```



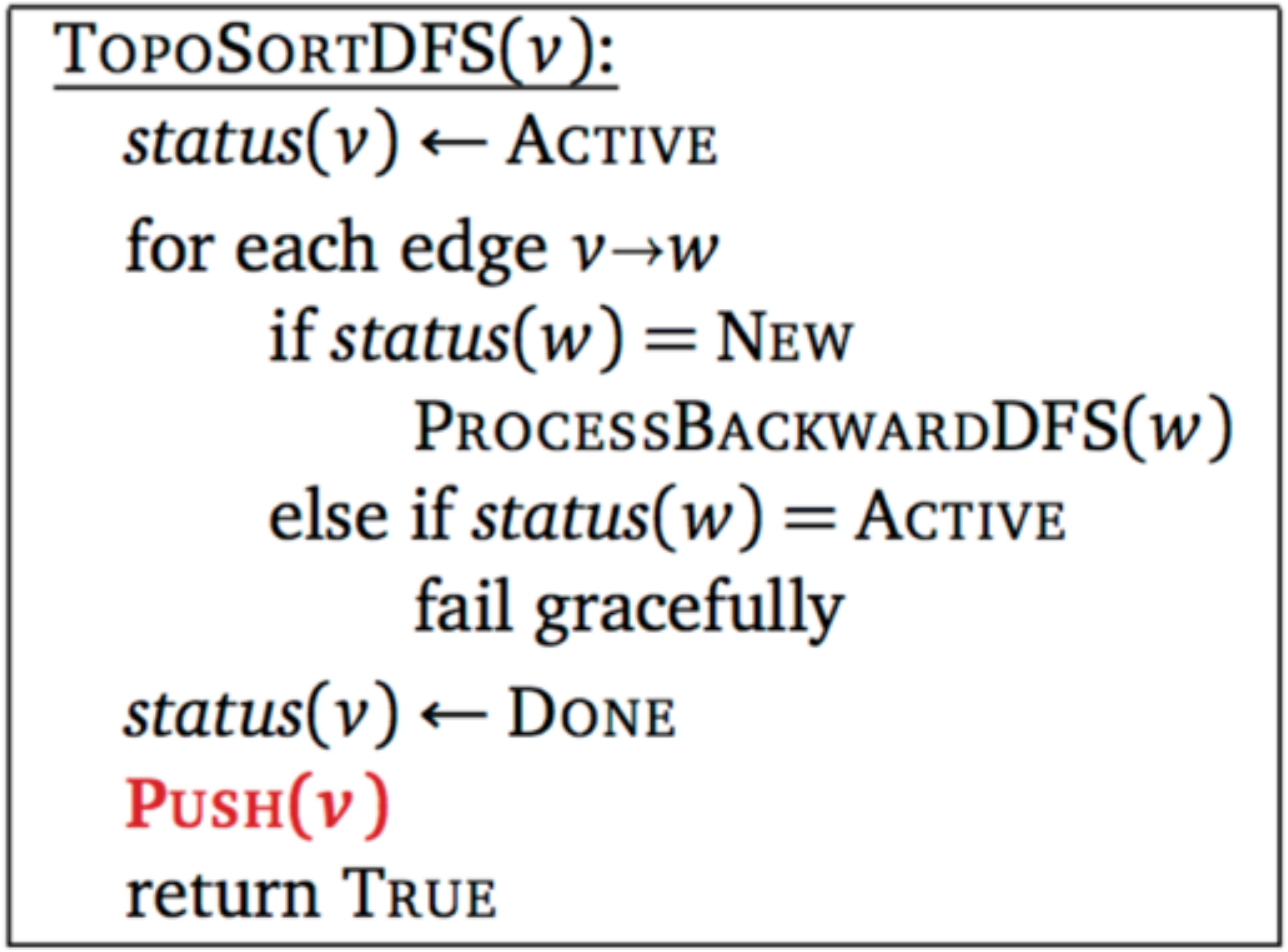
```
TopoSortDFS(v):
    status(v) ← Active
    for each edge v→w
        if status(w) = New
            ProcessBackwardDFS(w)
        else if status(w) = Active
            fail gracefully
    status(v) ← Done
    Push(v)
    return True
```

Topological Sort

- **Claim**: The order by which vertices are DONE in DFS is a reverse topological order (proof?).
- Overkill, all I need is to be able to do some computation so that we respect dependencies

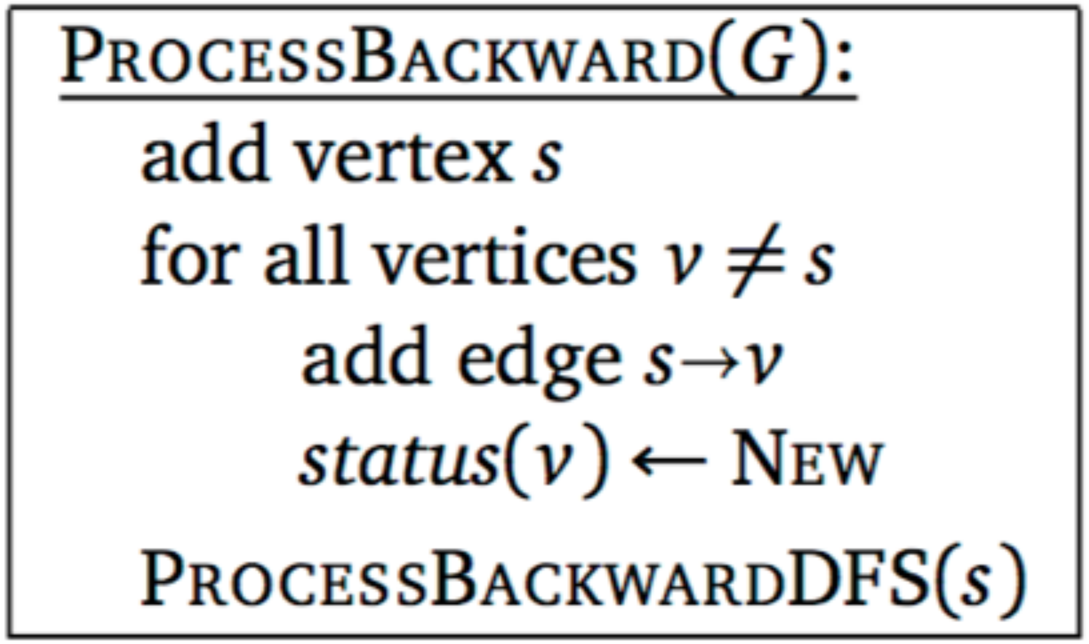


```
TOPOLOGICALSORT($G$):
    add vertex $s$
    for all vertices $v \neq s$
        add edge $s \to v$
        $status(v) \leftarrow$ NEW
    TOPOSORTDFS($s$)
    for $i \leftarrow 1$ to $V$
        $S[i] \leftarrow$ POP
    return $S[1..V]$
```

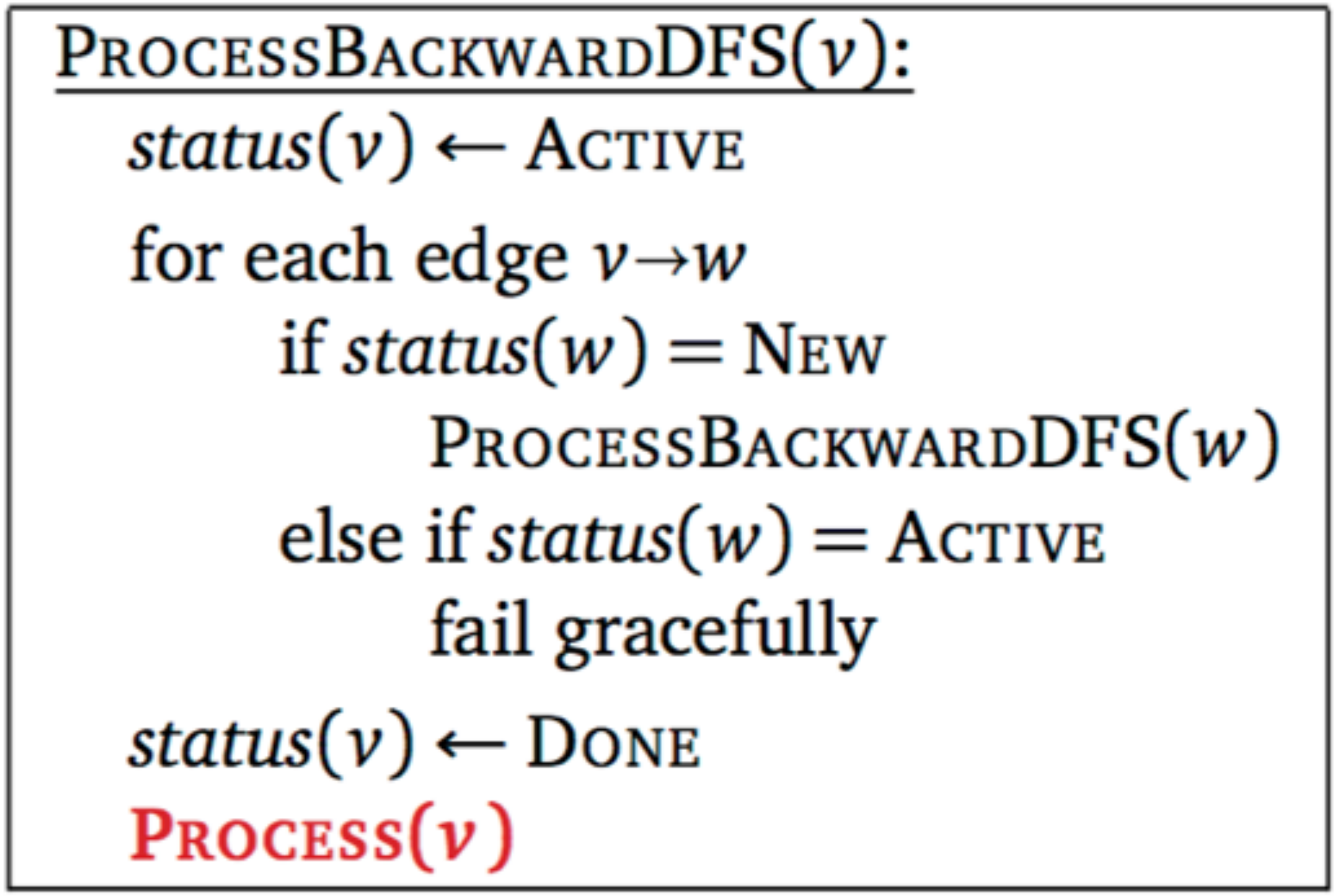


```
TOPOSORTDFS($v$):
    $status(v) \leftarrow$ ACTIVE
    for each edge $v \to w$
        if $status(w) =$ NEW
            PROCESSBACKWARDDFS($w$)
        else if $status(w) =$ ACTIVE
            fail gracefully
    $status(v) \leftarrow$ DONE
    PUSH($v$)
    return TRUE
```

Topological Sort



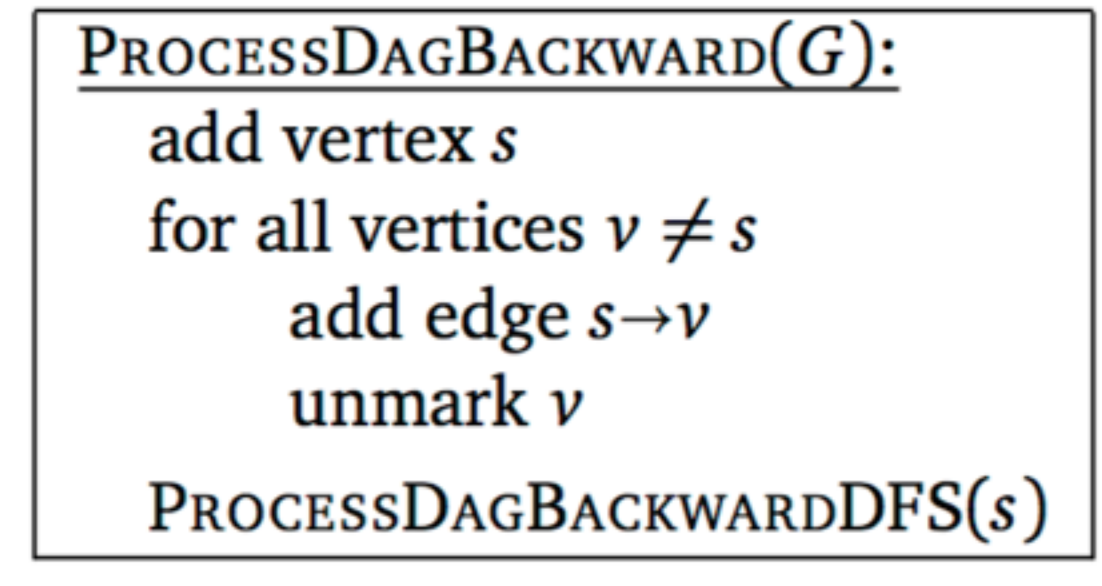
```
PROCESSBACKWARD($G$):
    add vertex $s$
    for all vertices $v \neq s$
        add edge $s \to v$
        $status(v) \leftarrow$ NEW
    PROCESSBACKWARDDFS($s$)
```



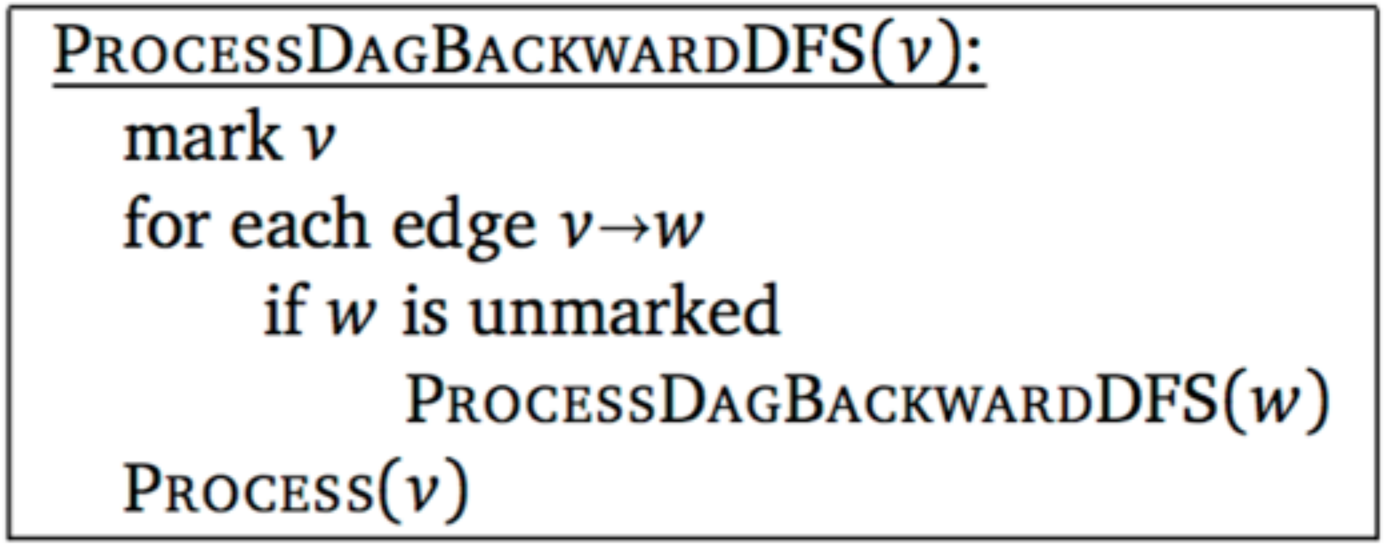
```
PROCESSBACKWARDDFS($v$):
    $status(v) \leftarrow$ ACTIVE
    for each edge $v \to w$
        if $status(w) =$ NEW
            PROCESSBACKWARDDFS($w$)
        else if $status(w) =$ ACTIVE
            fail gracefully
    $status(v) \leftarrow$ DONE
    PROCESS($v$)
```

- The order that I want to do commutation is the order I mark thing DONE
- I Process while I explore the node in DFS
- Processes every node in the graph in reverse topological order.
- Check for DAG in there, unless I know it is DAG.

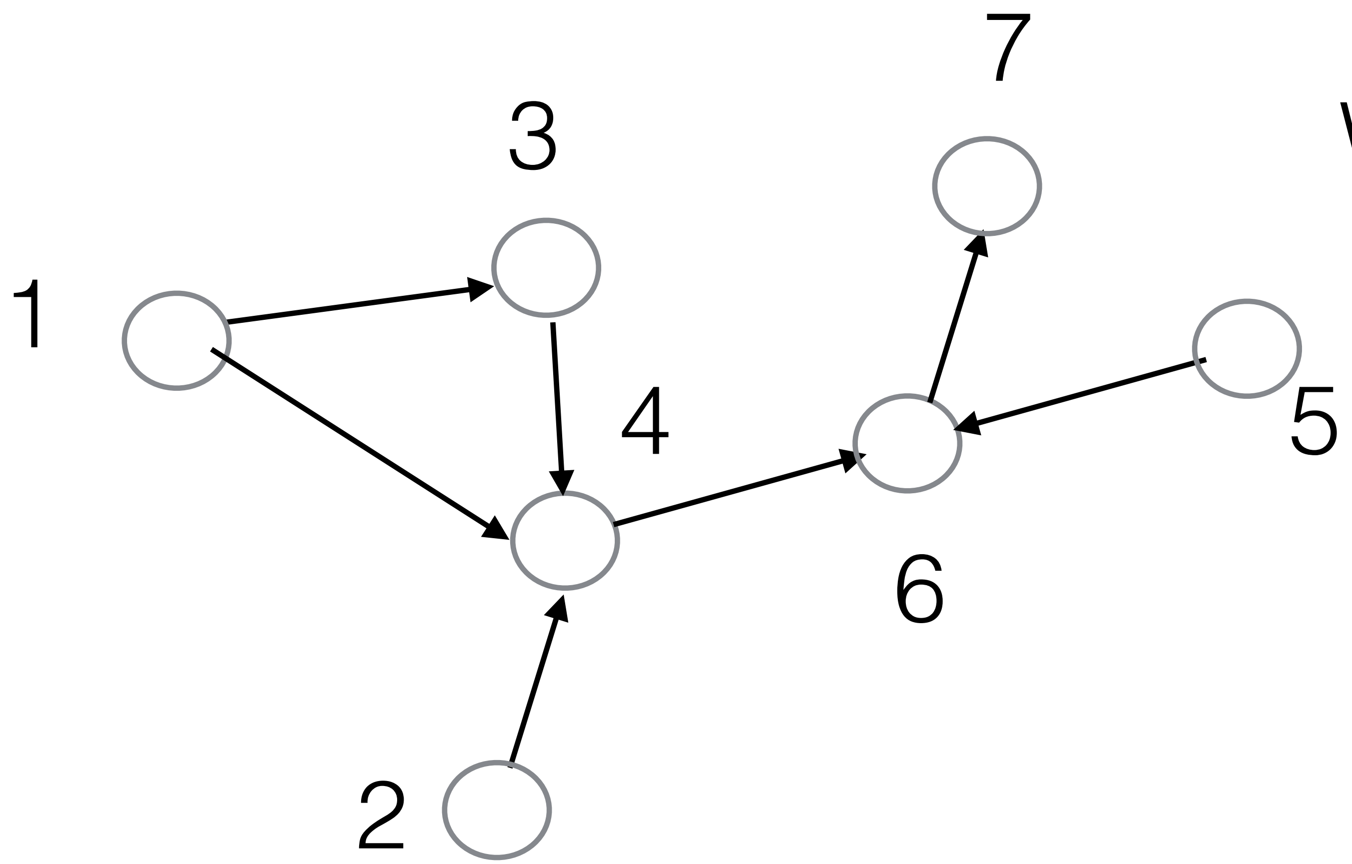
Topological Sort if DAG



```
PROCESSDAGBACKWARD(G):
    add vertex s
    for all vertices v ≠ s
        add edge s→v
        unmark v
    PROCESSDAGBACKWARDDFS(s)
```



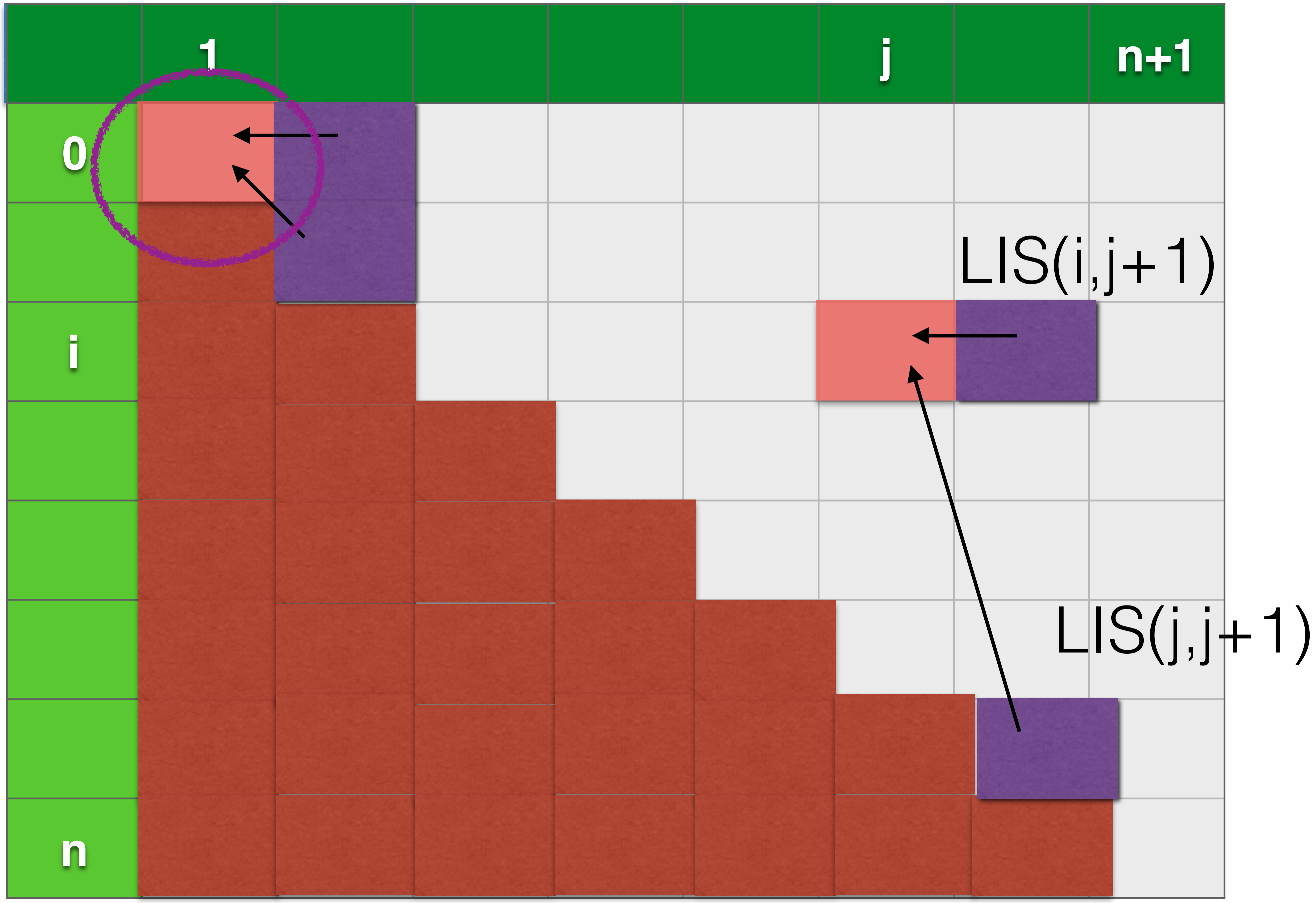
```
PROCESSDAGBACKWARDDFS(v):
    mark v
    for each edge v→w
        if w is unmarked
            PROCESSDAGBACKWARDDFS(w)
    PROCESS(v)
```



Where have we seen this before?

For i<j

$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i,j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i,j+1),\ 1 + LIS(j,j+1)\} & \text{otherwise} \end{cases}$$



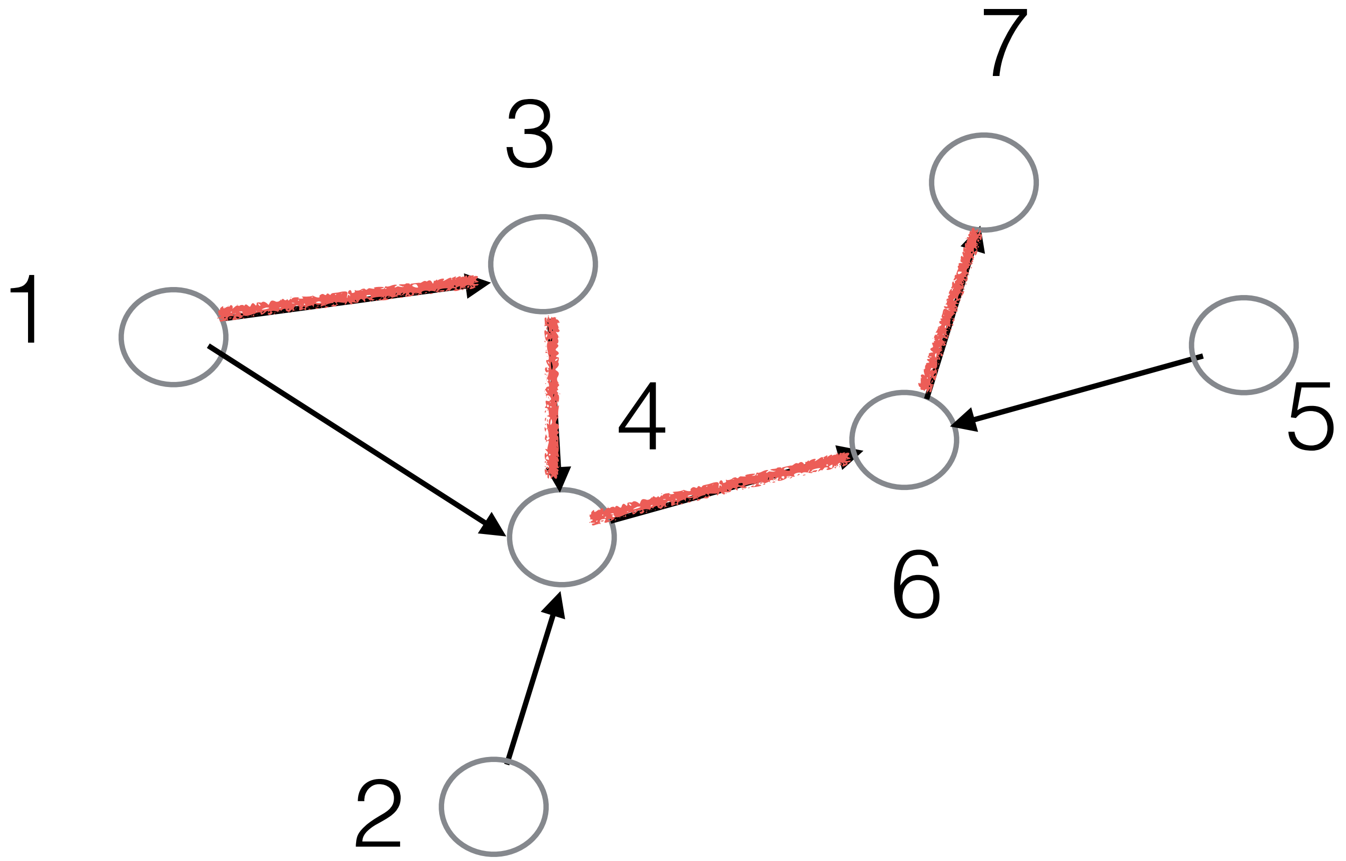
DP=DFS

Memoized recursion is DFS

Dynamic Programming uses topological sort

Longest Path in DAG

Given DAG and I am interested in finding the longest path.



Longest Path in DAG

Given DAG and I am interested in finding the longest path.

LLP(s,t) =length of longest path from s to t

LLP(s,t) =

0	if s=t
$\max_{s \rightarrow v} \{1+\mathrm{LLP}(v,t)\}$	o.w
$-\infty$	s sink

define max $\emptyset = -\infty$

t is constant throughout

Longest Path in DAG

Given DAG and I am interested in finding the longest path.

LLP(s,t) =length of longest path from s to t

LLP(s,t) =

0	if s=t
$\max_{s \rightarrow v} \{1+\mathrm{LLP}(v,t)\}$	o.w
$-\infty$	s sink

what data structure to use?
tha graph! Memoize LLP(s,t) into node s!

Longest Path in DAG

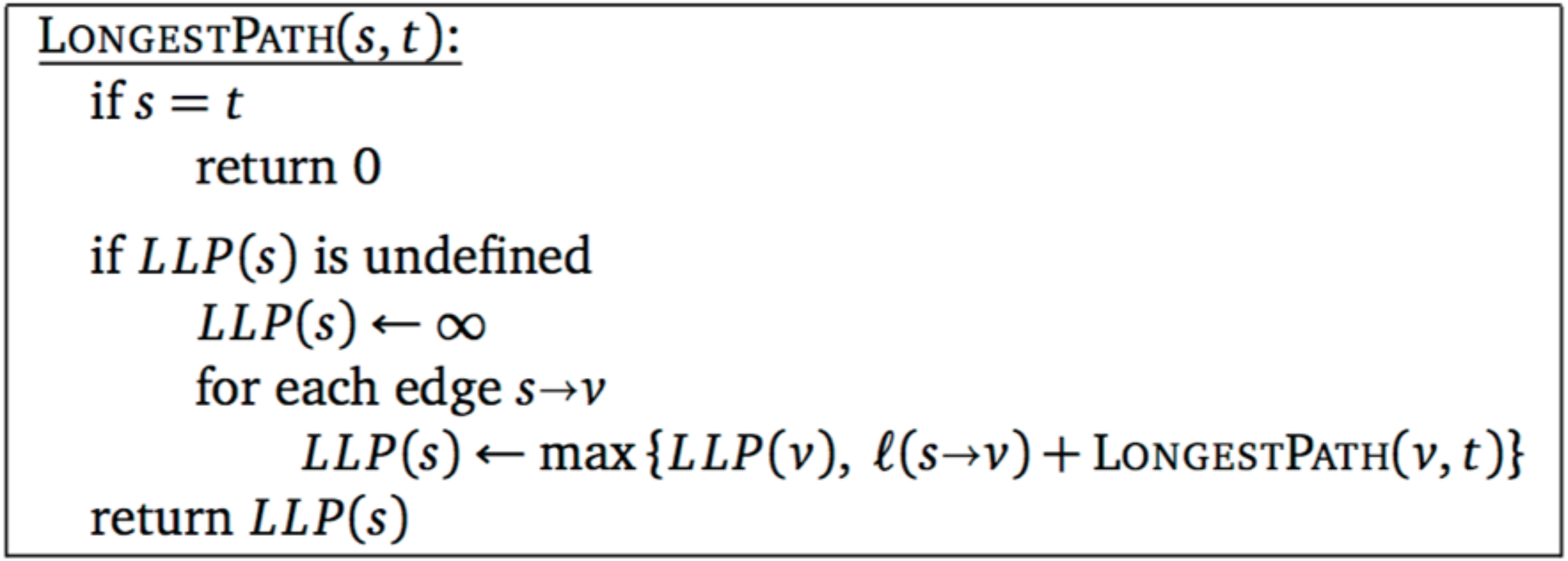
Given DAG and I am interested in finding the longest path.

LLP(s,t) =length of longest path from s to t

$$LLP(s,t) = \begin{cases} 0 & \text{if } s=t \\ \max_{s \to v} \{1+LLP(v,t)\} & \text{o.w} \\ -\infty & s \text{ sink} \end{cases}$$

What order?
Reverse topological sort order!

Longest Path in DAG

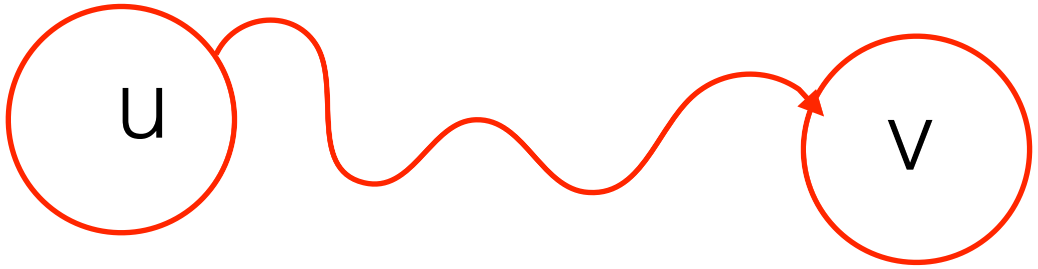


```
LONGESTPATH($s, t$):
    if $s = t$
        return 0
    if $LLP(s)$ is undefined
        $LLP(s) \leftarrow \infty$
        for each edge $s \rightarrow v$
            $LLP(s) \leftarrow \max\{LLP(v),\ \ell(s \rightarrow v) + \textsc{LongestPath}(v, t)\}$
    return $LLP(s)$
```

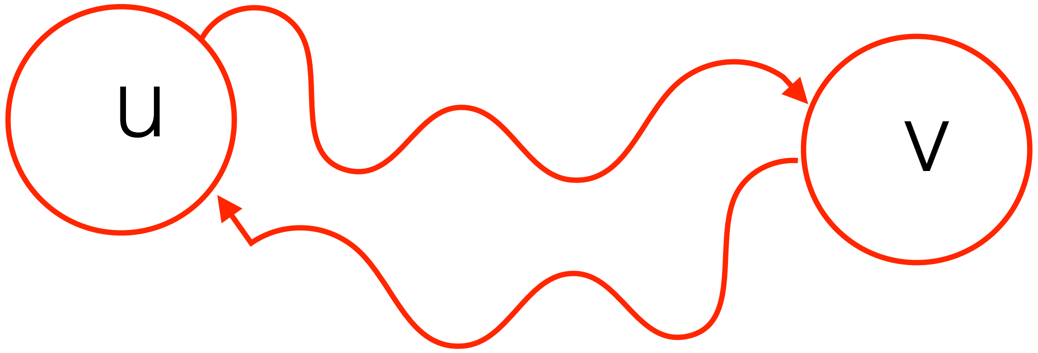
What is reverse topological order?
Just do DFS for reverse topological order!
it is also the naive recursive algorithm

Strong Connectivity

In directed graph vertex u can reach vertex v iff there is a directed path from u to b
reach(u) = set of vertices u can reach

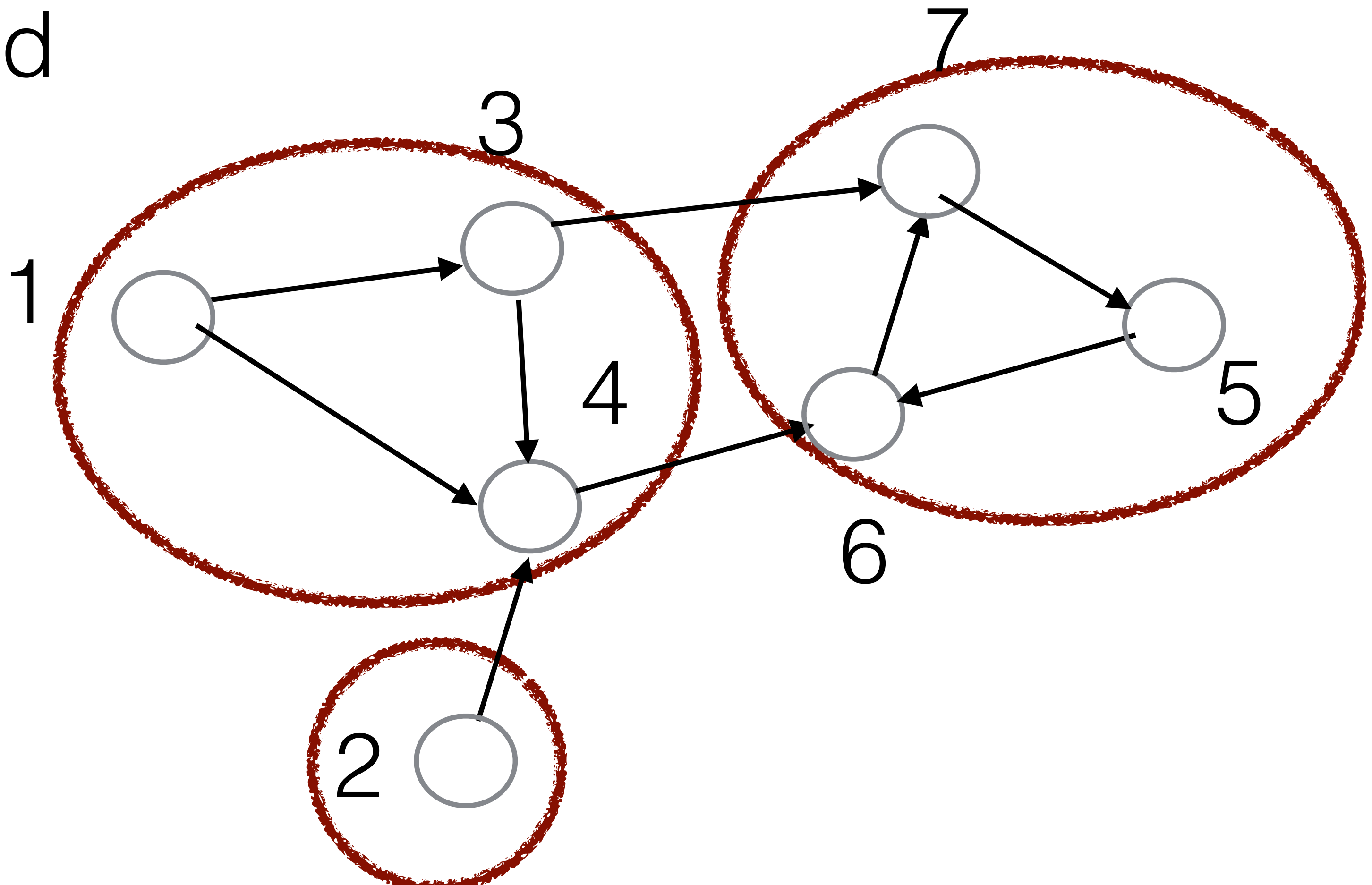


u and v are strongly connected if u can reach v and v can reach u



Strong Connectivity, SCC

- Strong connectivity is an equivalence relation
- Equivalence classes are called strongly connected components
- If G has a single strongly connected component: strongly connected



Strong Connectivity, SCC

- Strong connectivity is an equivalence relation
- Equivalence classes are called strongly connected components
- If G has a single strongly connected component: strongly connected
- When is G a DAG?
- Every SCC is a single vertex

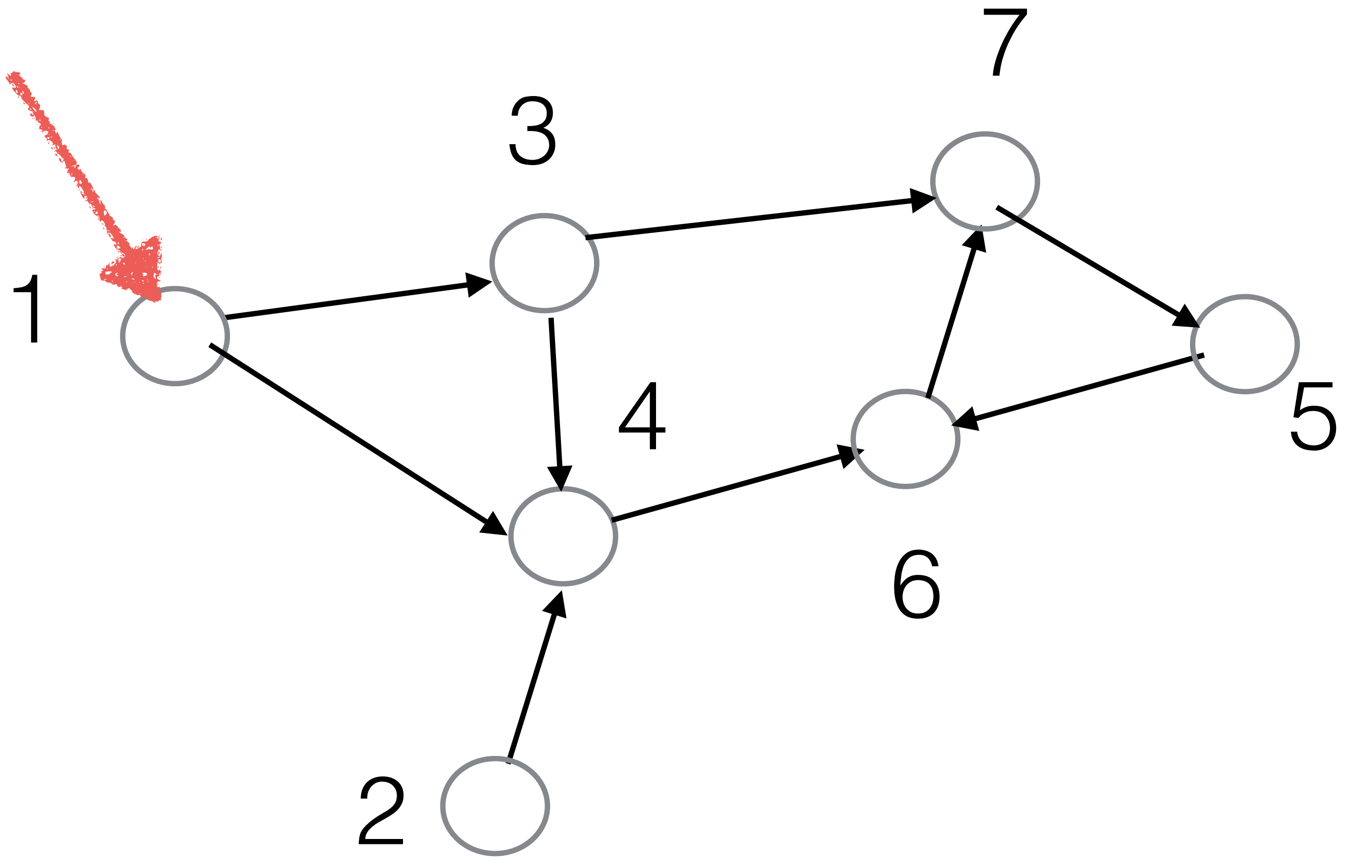
Strong Connectivity, SCC

- How to compute SCC of vertex u in O(|V|+|E|) time?

DFS(G,u) gives us Reach(u)

DFS(G^{rev},u) gives us all the stuff that can reach u

Take intersection of both for SCC

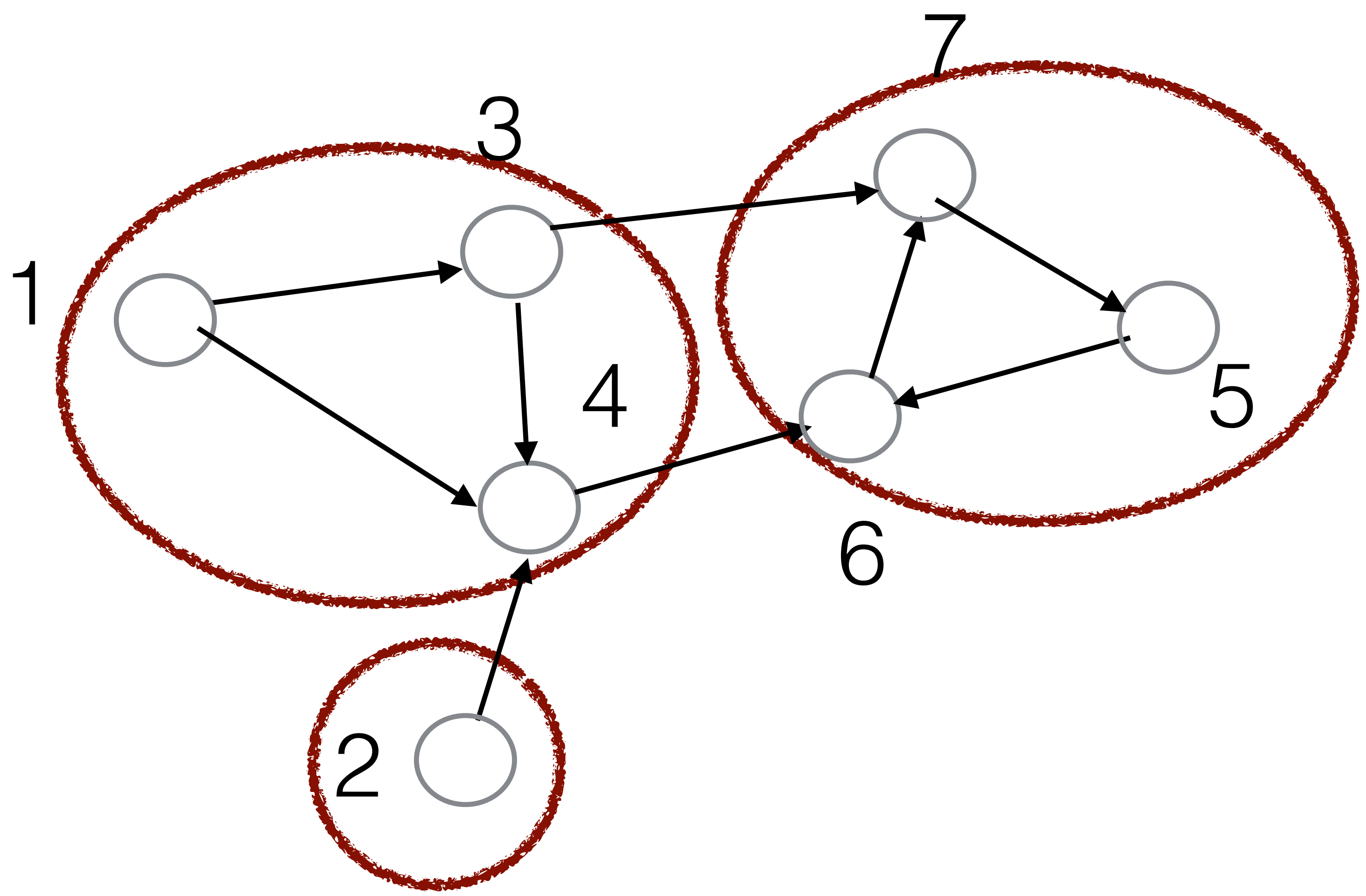


Strong Connectivity, SCC

- How to compute SCC of vertex u in O(|V|+|E|) time?
- Compute Reach(u) with DFS
- Compute Reach^{-1}(u) ={v: u is in Reach(v)} with DFS on reverse graph
- SCC is the intersection of the two sets.
- How to compute all SCC of a graph?
- Naive: O(|V||E|) time.
- Can we do better?

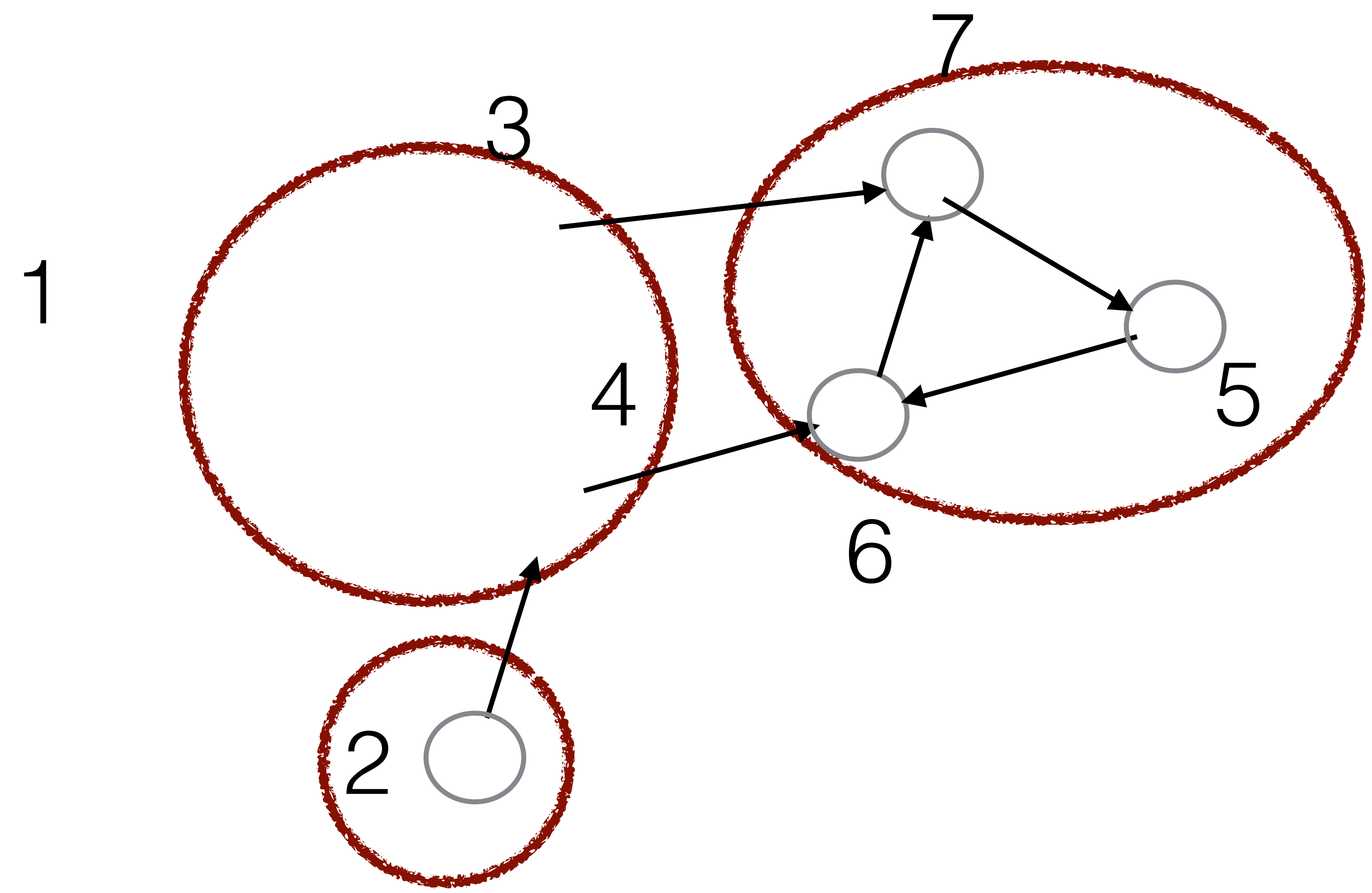
SCC Graph

For every directed graph G, scc(G) is another (meta)graph: Contract each SCC of G in one vertex and collapse parallel edges



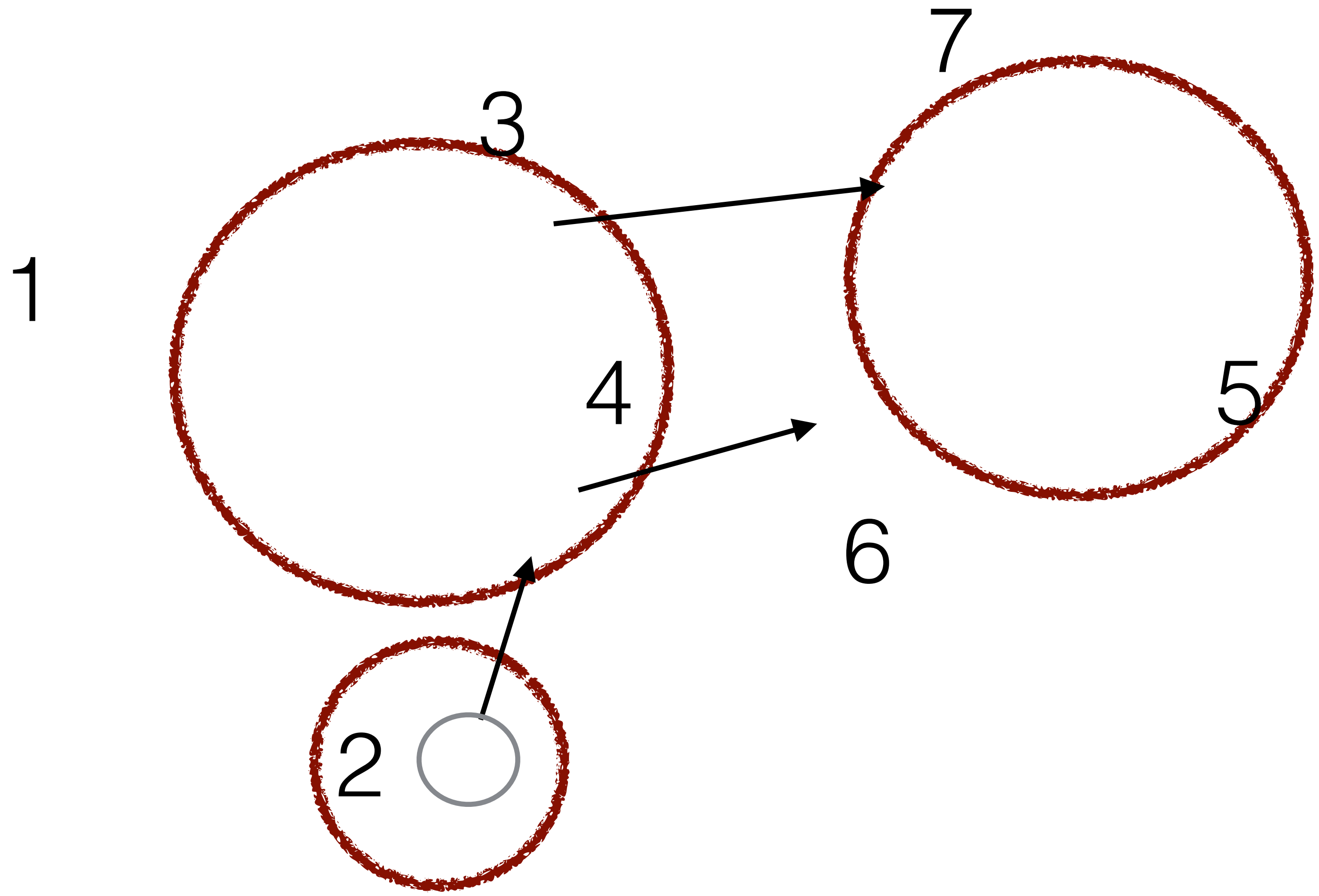
SCC Graph

For every directed graph G, scc(G) is another (meta)graph: Contract each SCC of G in one vertex and collapse parallel edges



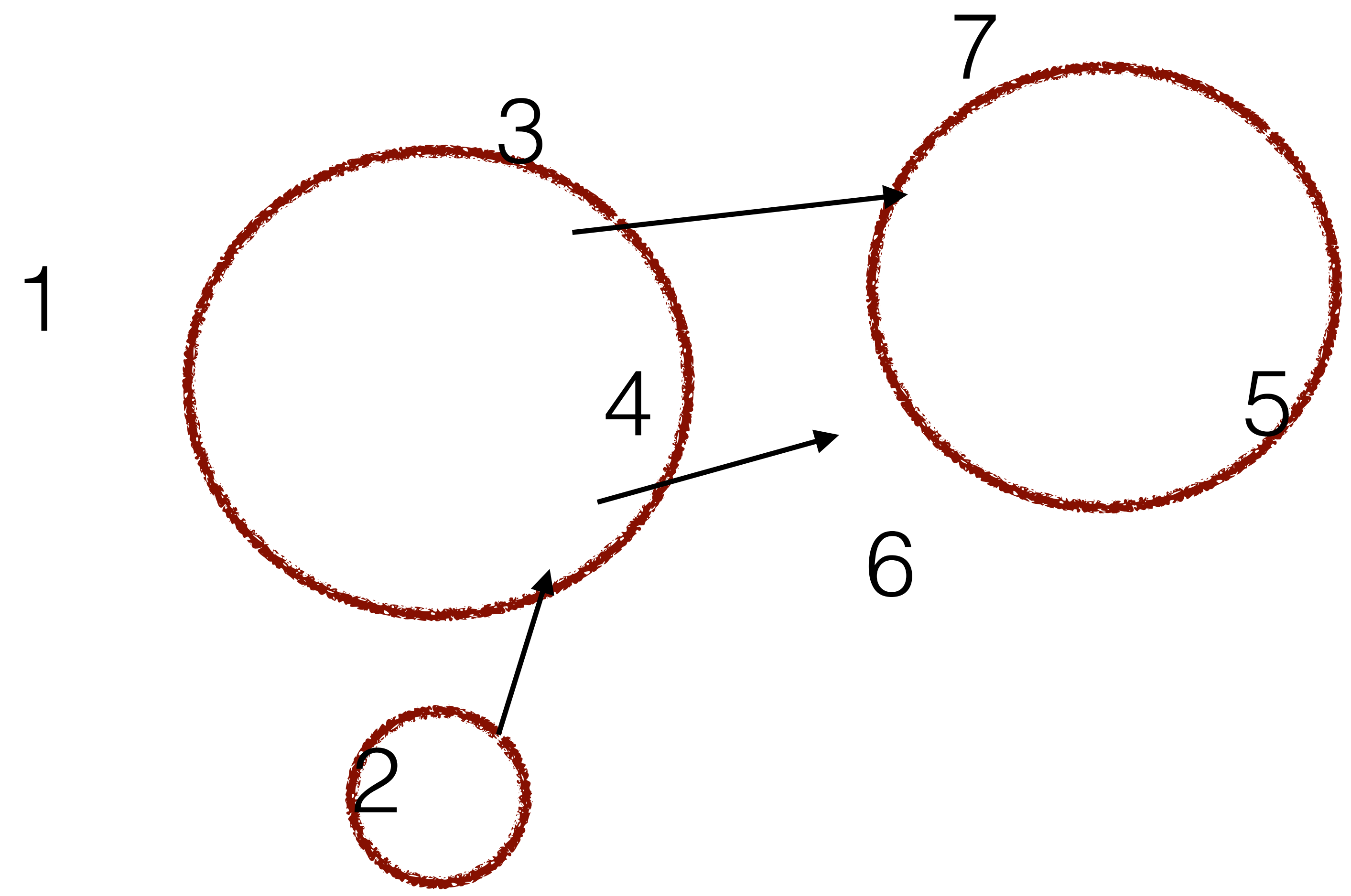
SCC Graph

For every directed graph G, scc(G) is another (meta)graph: Contract each SCC of G in one vertex and collapse parallel edges



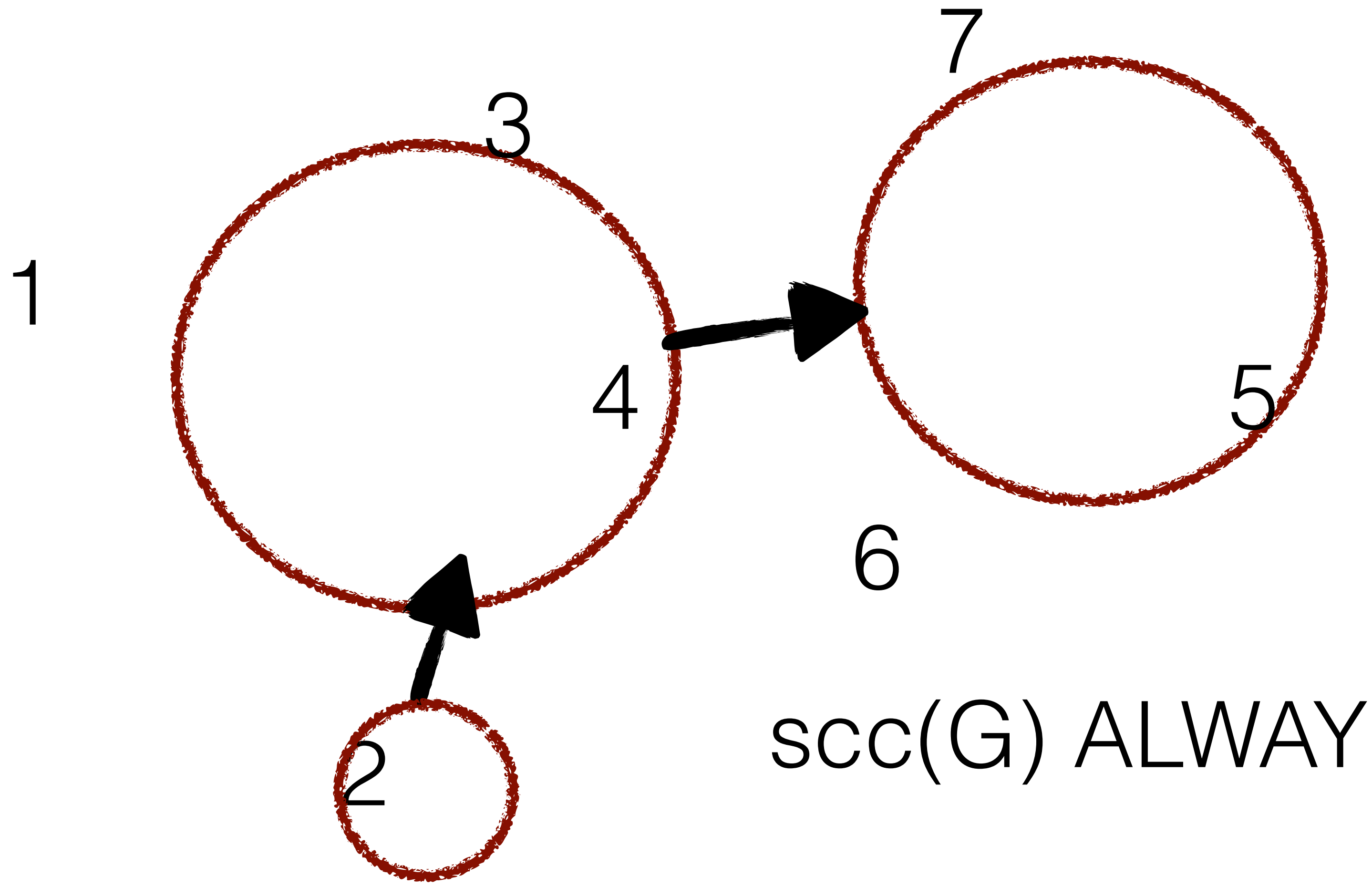
SCC Graph

For every directed graph G, scc(G) is another (meta)graph: Contract each SCC of G in one vertex and collapse parallel edges



SCC Graph

For every directed graph G, scc(G) is another (meta)graph: Contract each SCC of G in one vertex and collapse parallel edges

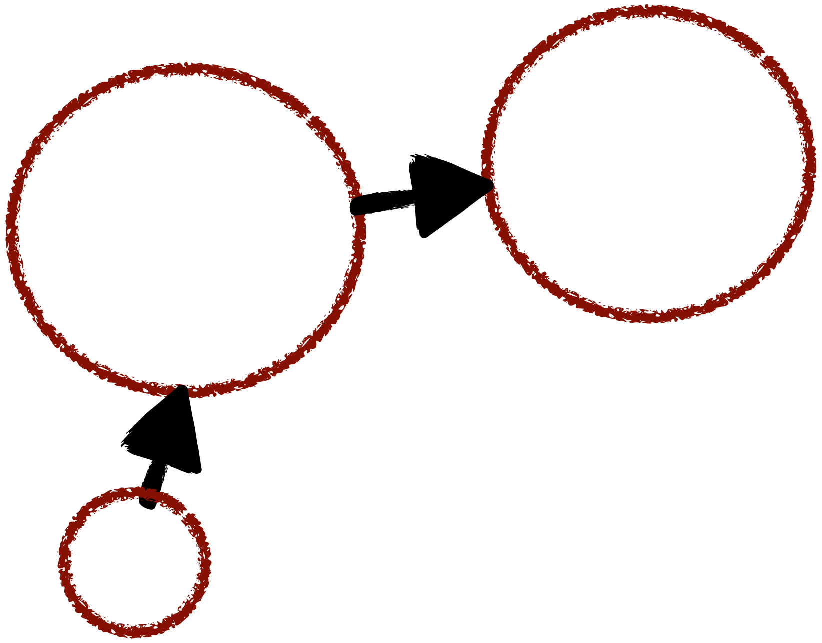


scc(G) ALWAYS A DAG!

SCC Graph

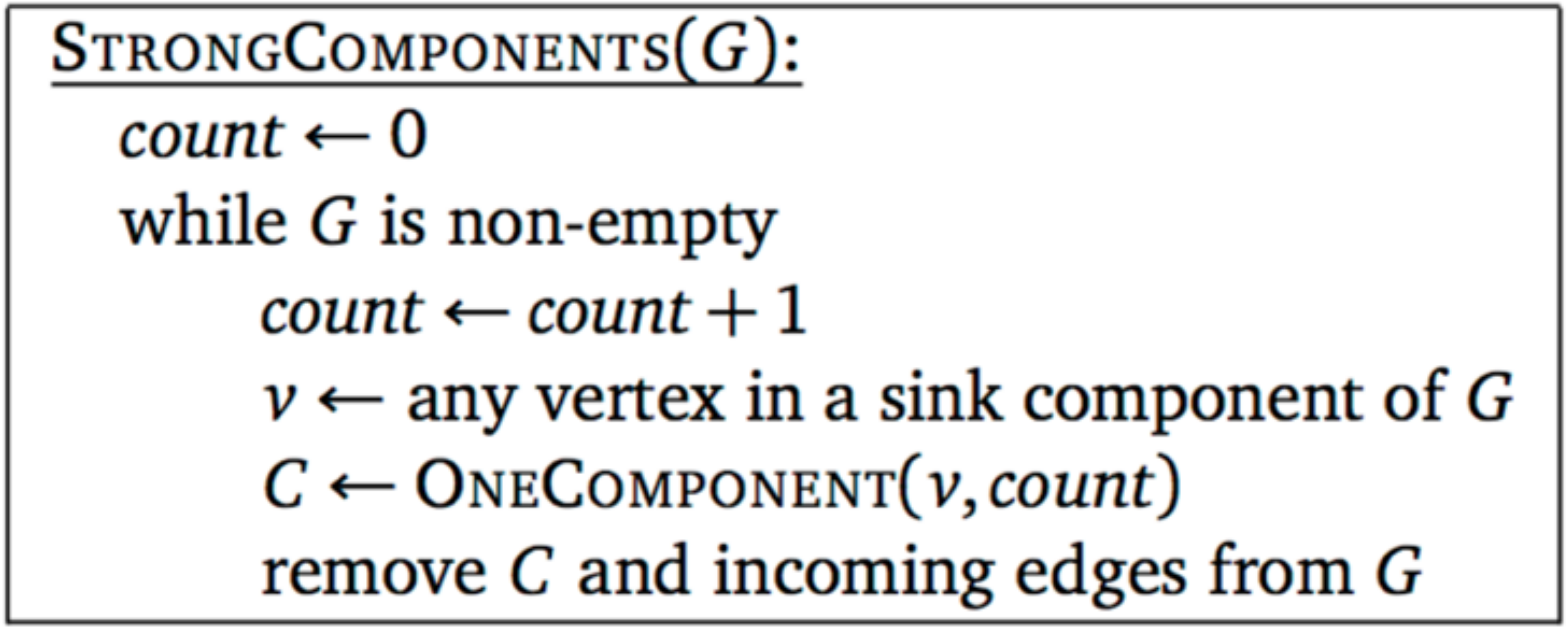
For every directed graph G, scc(G) is another (meta)graph: Contract each SCC of G in one vertex and collapse parallel edges

- I can topologically sort the SCC.
- There is a sink SCC
- DFS starting from a vertex in C, reaches only vertices in C and nothing else



SCC Graph

Can compute all the SCC:



```
StrongComponents(G):
    $count \gets 0$
    while $G$ is non-empty
        $count \gets count + 1$
        $v \gets$ any vertex in a sink component of $G$
        $C \gets$ OneComponent($v, count$)
        remove $C$ and incoming edges from $G$
```

How to find a vertex in a sink component? (next time)