CS 341: ALGORITHMS

Lecture 12: graph algorithms III – DAG testing, topsort, SCC

Readings: see website

Trevor Brown

https://student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca

DFS APPLICATION: **TESTING** WHETHER A GRAPH IS A **DAG**

A directed graph G is a **directed acyclic graph**, or **DAG**, if G contains no directed cycle.

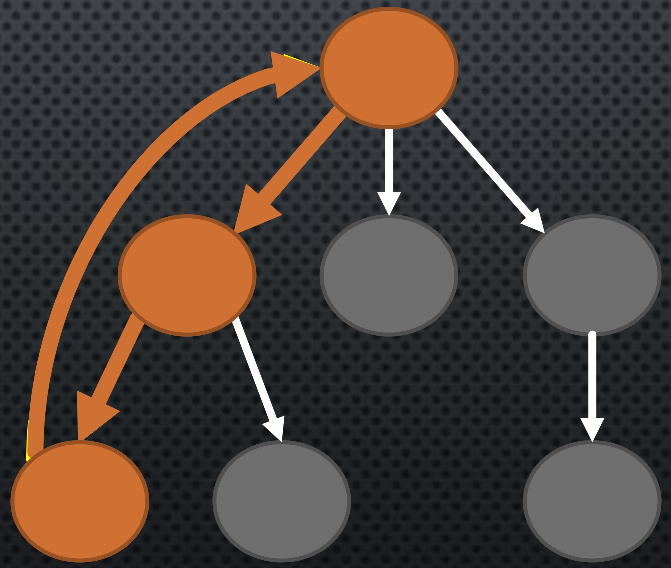
Lemma 6.7

A directed graph is a DAG if and only if a depth-first search encounters no back edges.

Proof.

($\Rightarrow$): Any back edge creates a directed cycle.

Back edge:
points to an ancestor
in the DFS forest



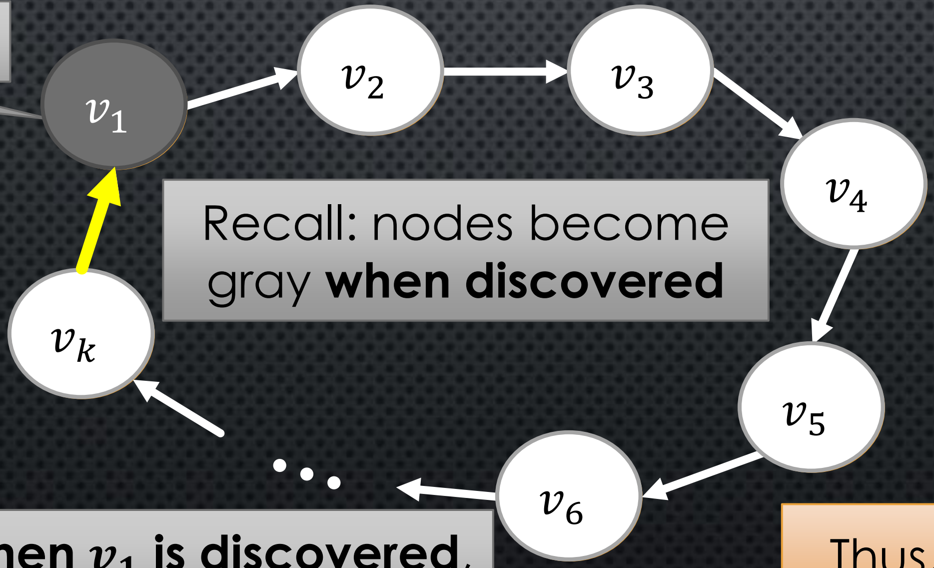
- Case (⇐): Suppose ∃ directed cycle. Show ∃ back edge.
  - Let $v_1, v_2, \ldots, v_k, v_1$ be a directed cycle
  - WLOG let v_1 be *earliest* discovered node in the cycle

	$v_k \rightarrow v_1$
edge type	discovery/finish times
tree	$d[v_k] < d[v_1] < f[v_1] < f[v_k]$
forward	$d[v_k] < d[v_1] < f[v_1] < f[v_k]$
back	$d[v_1] < d[v_k] < f[v_k] < f[v_1]$
cross	$d[v_1] < f[v_1] < d[v_k] < f[v_k]$

Discovered **before** $v_2, \ldots, v_k$

Consider edge $\{\boldsymbol{v_k}, \boldsymbol{v_1}\}$

Since $d[v_1] < d[v_k]$, $\{\boldsymbol{v_k}, \boldsymbol{v_1}\}$ must be a **back** or **cross** edge. Why?



Recall: nodes become gray **when discovered**

So **when $\boldsymbol{v_1}$ is discovered**, $v_2, \ldots, v_k$ are **all white**

Recall: every node v_i that is **white-reachable** from v_1 when we discover v_1 (call $DFSVisit(v_1)$) turns **black before $\boldsymbol{v_1}$** ($f[v_i] < f[v_1]$)

So v_k must turn black **before** v_1, and we have $f[v_k] < f[v_1]$.

Thus, $\{\boldsymbol{v_k}, \boldsymbol{v_1}\}$ must be a **back edge**. QED

TURNING THE **LEMMA** INTO AN **ALGORITHM**

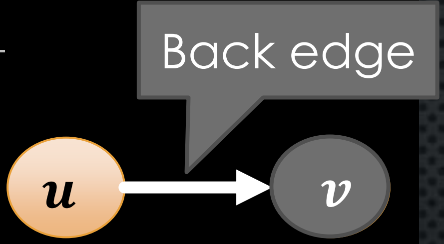
Lemma 6.7

A directed graph is a DAG if and only if a depth-first search encounters no back edges.

- Search for back edges
- How to identify a back-edge?

When we observe an edge from u to v, check if $\boldsymbol{v}$ **is gray**

edge type	colour of v	discovery/finish times
tree	**white**	$d[u] < d[v] < f[v] < f[u]$
forward	**black**	$d[u] < d[v] < f[v] < f[u]$
back	**gray**	$d[v] < d[u] < f[u] < f[v]$
cross	**black**	$d[v] < f[v] < d[u] < f[u]$

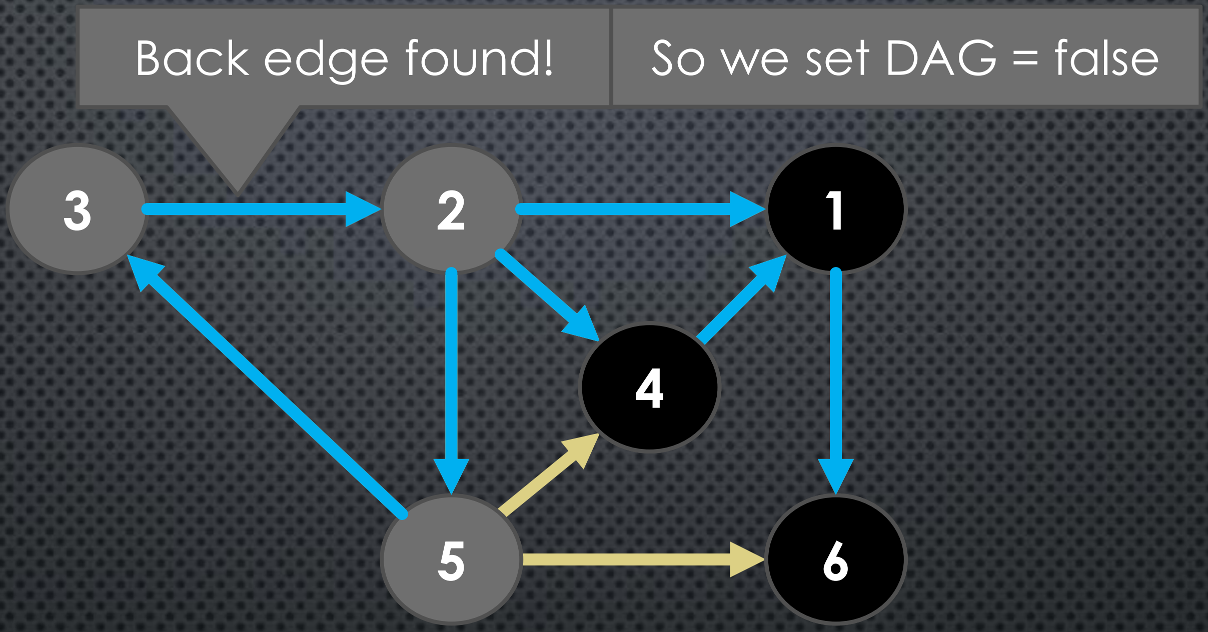


DFS: TESTING WHETHER A GRAPH IS A DAG

```
global variables:
    pred[1..n] = [null, null, ..., null]
    colour[1..n] = [white, white, ..., white]
    d[1..n] = [0, 0, ..., 0] // discovery times
    f[1..n] = [0, 0, ..., 0] // finish times
    time = 0
    DAG = true

IsDAG(adj[1..n])
    for v = 1..n
        if colour[v] == white
            DFSVisit(adj, v)
    return DAG
```

```
DFSVisit(adj[1..n], v)
    colour[v] = gray
    time = time + 1
    d[v] = time

    for each w in adj[v]
        if colour[w] == white
            pred[w] = v
            DFSVisit(w)
        if color[w] == gray
            DAG = false

    colour[v] = black
    time = time + 1
    f[v] = time
```

EXAMPLE



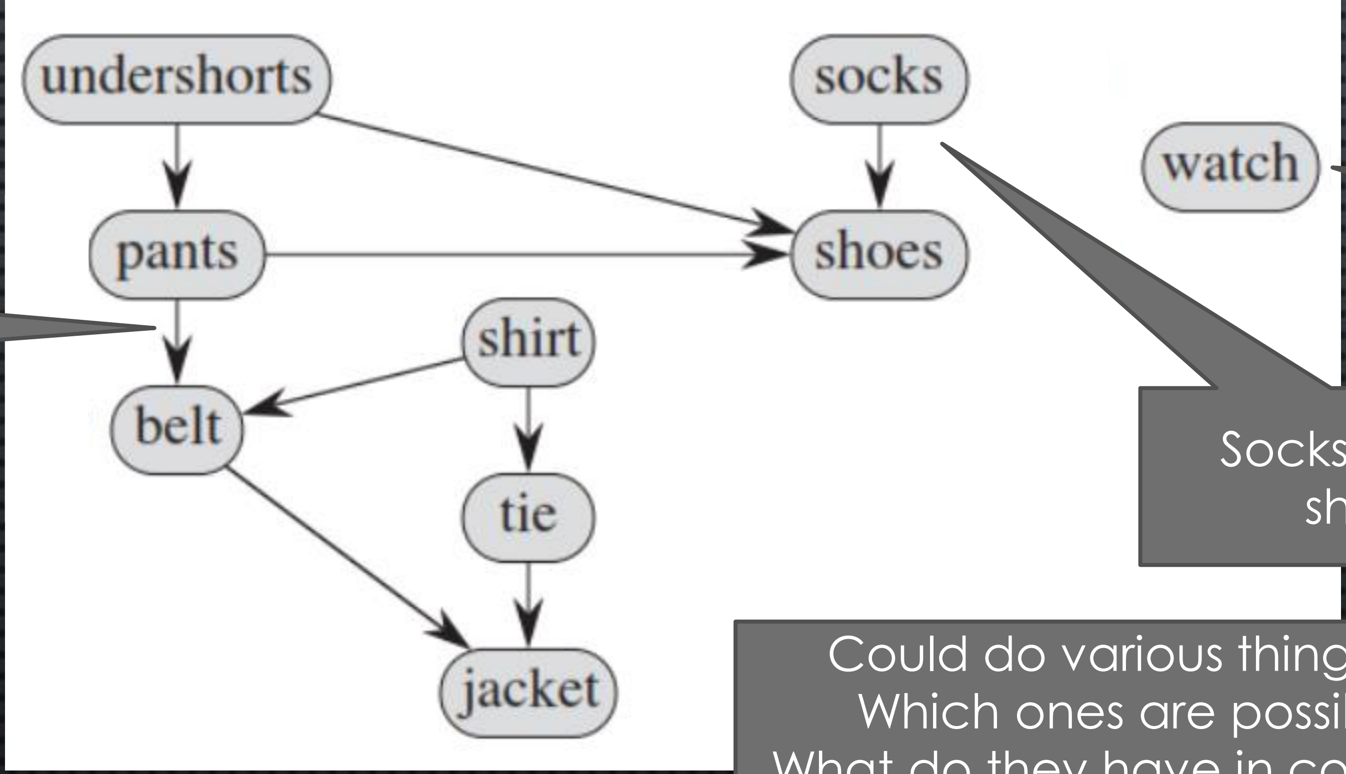
TOPOLOGICAL SORT

Finding node orderings that satisfy given constraints

DEPENDENCY GRAPH

- Edge $\{u, v\}$ means u must be completed **before** v

Example problem: getting dressed in the morning

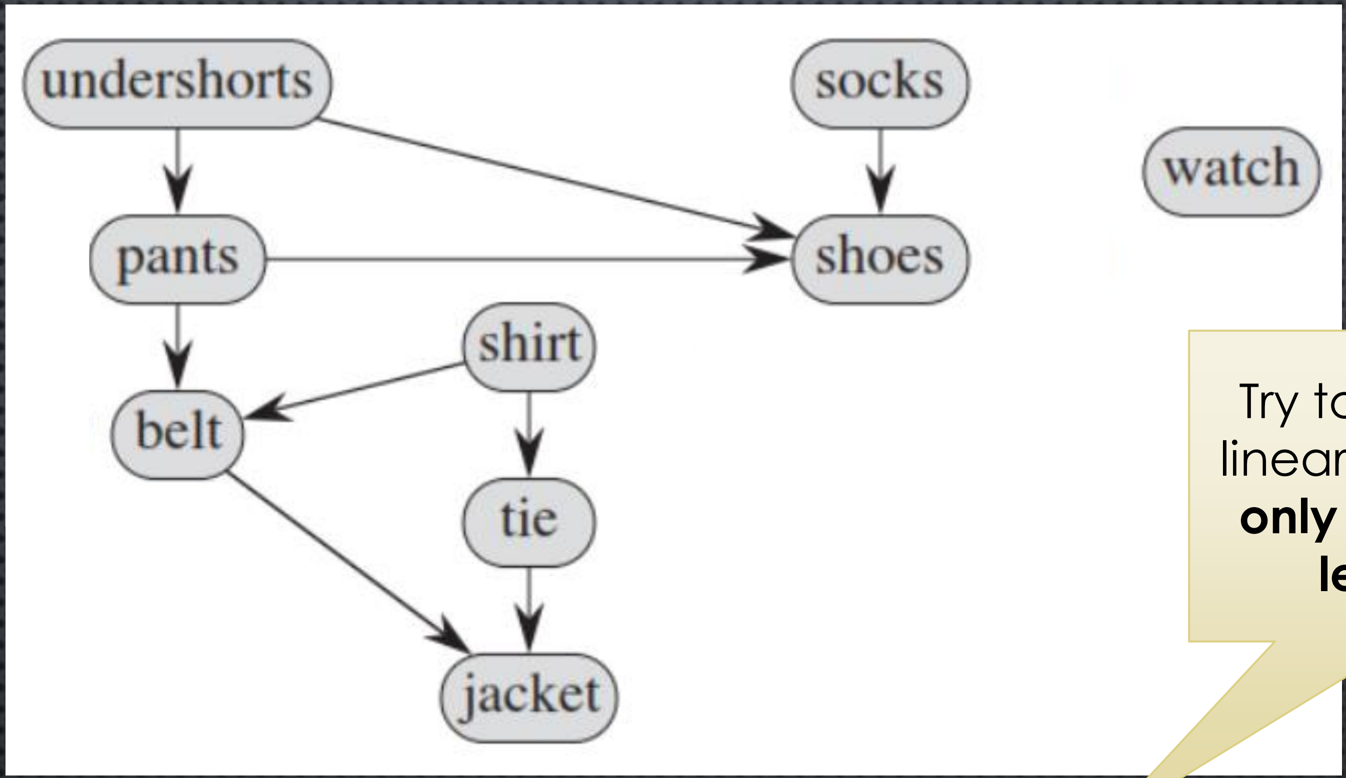


Pants before belt

Socks before shoes

Watch any time

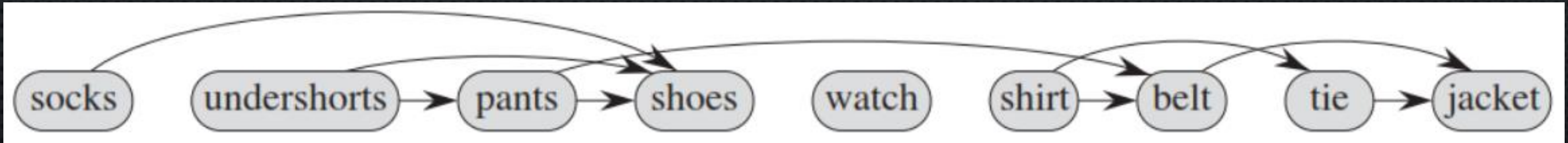
Could do various things first.
Which ones are possible?
What do they have in common?



Try to order nodes linearly so there are **only** pointers from **left to right!**

Possible IFF **graph is a <u>DAG</u>**

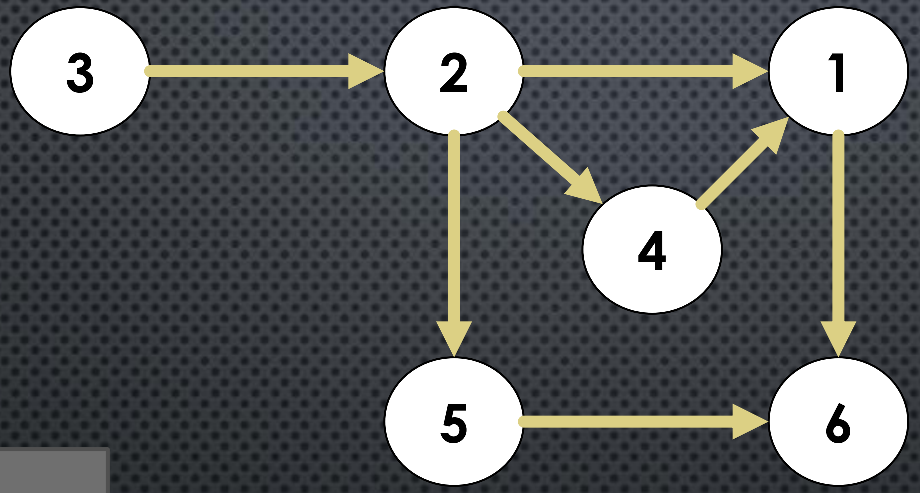
Topological sort



FORMAL DEFINITION

A directed graph $G = (V, E)$ has a **topological ordering**, or **topological sort**, if there is a linear ordering $<$ of all the vertices in V such that $u < v$ whenever $uv \in E$.

Graph G



Topological sort of G

$v_3 < v_2 < v_5 < v_4 < v_1 < v_6$

Edges are directed **only left-to-right** in this ordering

USEFUL FACT

Lemma 6.5

A DAG contains a vertex of indegree 0.

Proof.

Suppose we have a directed graph in which every vertex has positive indegree. Let v_1 be any vertex. For every $i \geq 1$, let $v_{i+1}v_i$ be an arc. In the sequence $v_1, v_2, v_3, \ldots,$ consider the first repeated vertex, $v_i = v_j$ where $j > i$. Then $v_j, v_{j-1}, \ldots, v_i, v_j$ is a directed cycle. □

One of these must be **repeated**. So there is a cycle!

$$v_{n+1} \to v_n \to \cdots \to v_3 \to v_2 \to v_1$$

TOPOLOGICAL SORT VIA DFS

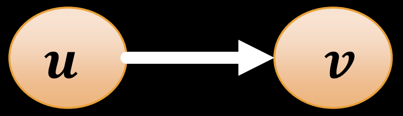
- We can implement topological sort by using **DFS**!
- The **finishing times** of nodes help us
- Understanding this algo will be **key** for understanding **strongly connected components**

Lemma 6.8

Suppose D is a DAG. Then $f[v] < f[u]$ for every arc uv.

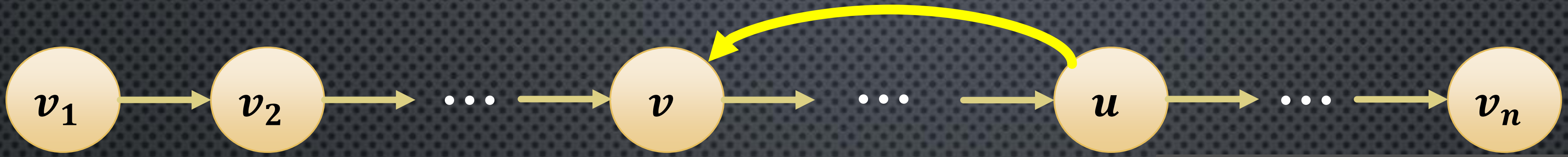
Recall from DAG-testing: there are **no back edges** in a DAG

edge type	colour of v	discovery/finish times
tree	white	$d[u] < d[v] < \boxed{f[v] < f[u]}$
forward	black	$d[u] < d[v] < \boxed{f[v] < f[u]}$
~~back~~	~~gray~~	$\cancel{d[v] < d[u] < f[u] < f[v]}$
cross	black	$d[v] < \boxed{f[v] < d[u] < f[u]}$



Theorem: if D is a DAG, and we order vertices in **reverse order of finishing time**, (i.e., by largest to smallest finish time) then we get a topological ordering!

To see **why**, suppose D is a DAG and we order nodes in this way, so $f_{v_1} > f_{v_2} > \cdots > f_{v_{n-1}} > f_{v_n}$



For contradiction, suppose a **right-to-left** edge $\{u, v\}$ exists

By our node ordering, $f_v > f_u$

But the lemma says for every edge $\{u, v\}$, we must have $f_v < f_u$

Lemma 6.8

Suppose D is a DAG. Then $f[v] < f[u]$ for every arc uv.

Contradiction! Right-to-left edge cannot exist.
So is is a topological ordering.

TOPOLOGICAL ORDERING VIA DFS

$O(n+m)$ w/adj. lists

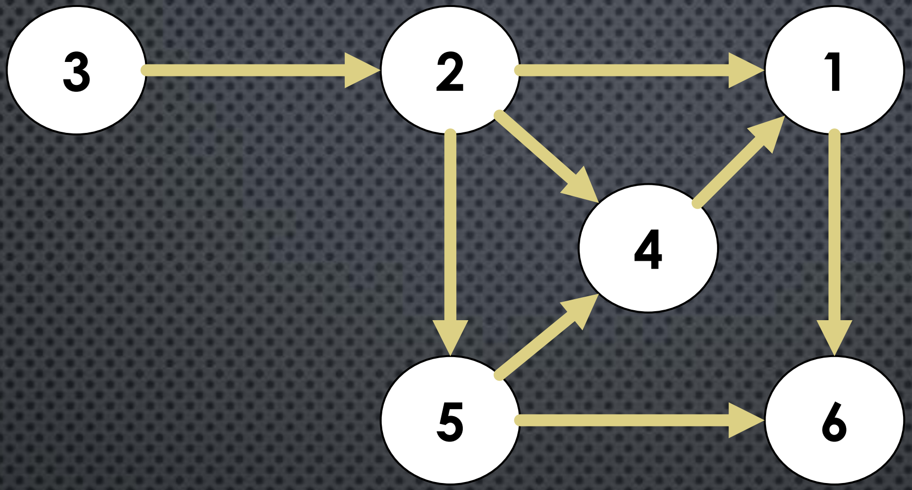
```
global variables:
    pred[1..n] = [null, null, ..., null]
    colour[1..n] = [white, white, ..., white]
    d[1..n] = [0, 0, ..., 0] // discovery times
    f[1..n] = [0, 0, ..., 0] // finish times
    time = 0
    DAG = true

TopologicalSort(adj[1..n])
    S = new stack
    for v = 1..n
        if colour[v] == white
            DFSVisit(adj, v, S)
    if DAG then return S
    return null
```

```
DFSVisit(adj[1..n], v, S)
    colour[v] = gray
    time = time + 1
    d[v] = time

    for each w in adj[v]
        if colour[w] == white
            pred[w] = v
            DFSVisit(w)
        if color[w] == gray
            DAG = false

    colour[v] = black
    S.push(v)
    time = time + 1
    f[v] = time
```

Save each node when it **finishes**

Push smallest finishing time first → **pop largest** first

HOME EXERCISE: RUN ON THIS GRAPH



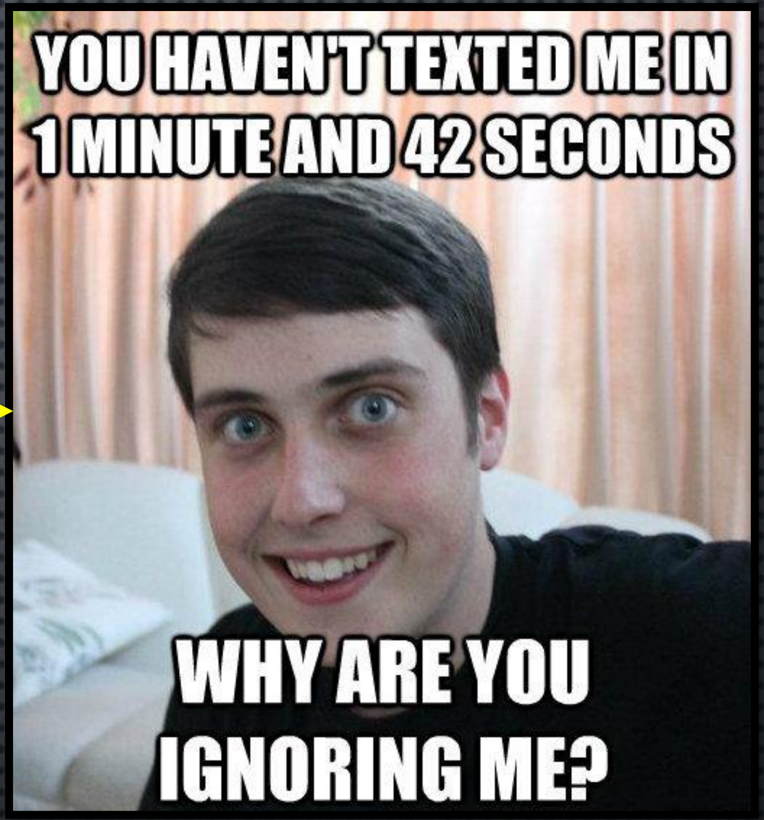
The initial calls are *DFSvisit*(1), *DFSvisit*(2) and *DFSvisit*(3).

The discovery/finish times are as follows:

v	$d[v]$	$f[v]$
1	1	4
2	5	10
3	11	12

v	$d[v]$	$f[v]$
4	6	7
5	8	9
6	2	3

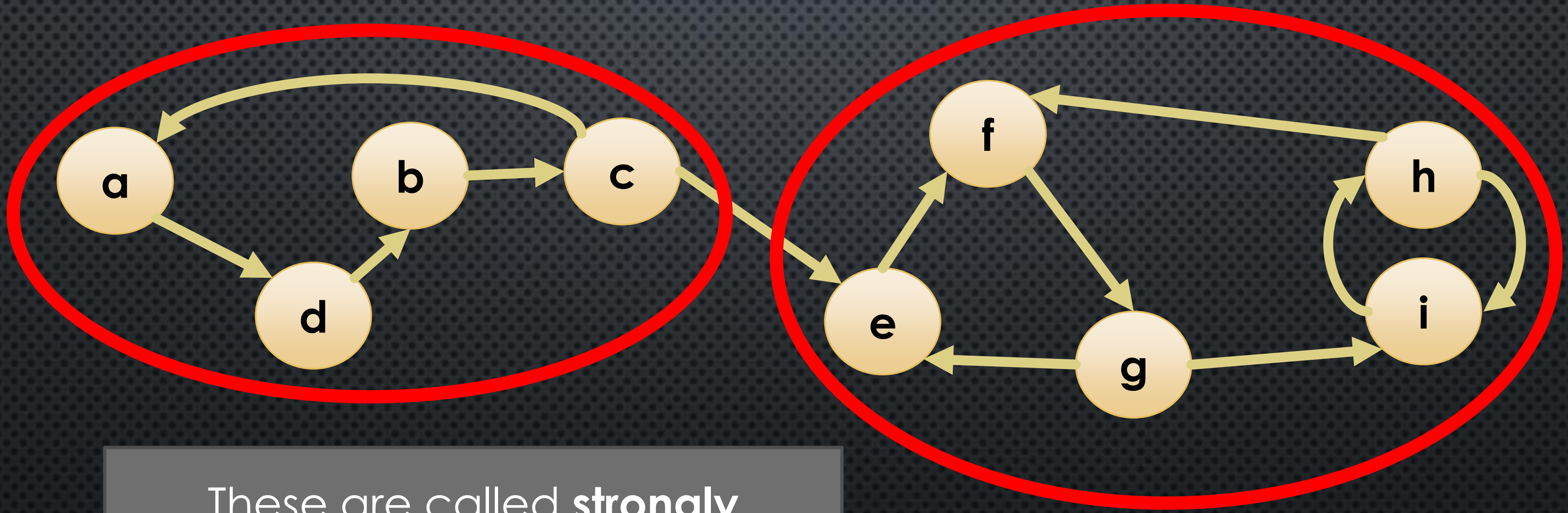
The topological ordering is $3, 2, 5, 4, 1, 6$ (reverse order of finishing time).



STRONGLY CONNECTED COMPONENTS

STRONGLY CONNECTED **COMPONENTS**

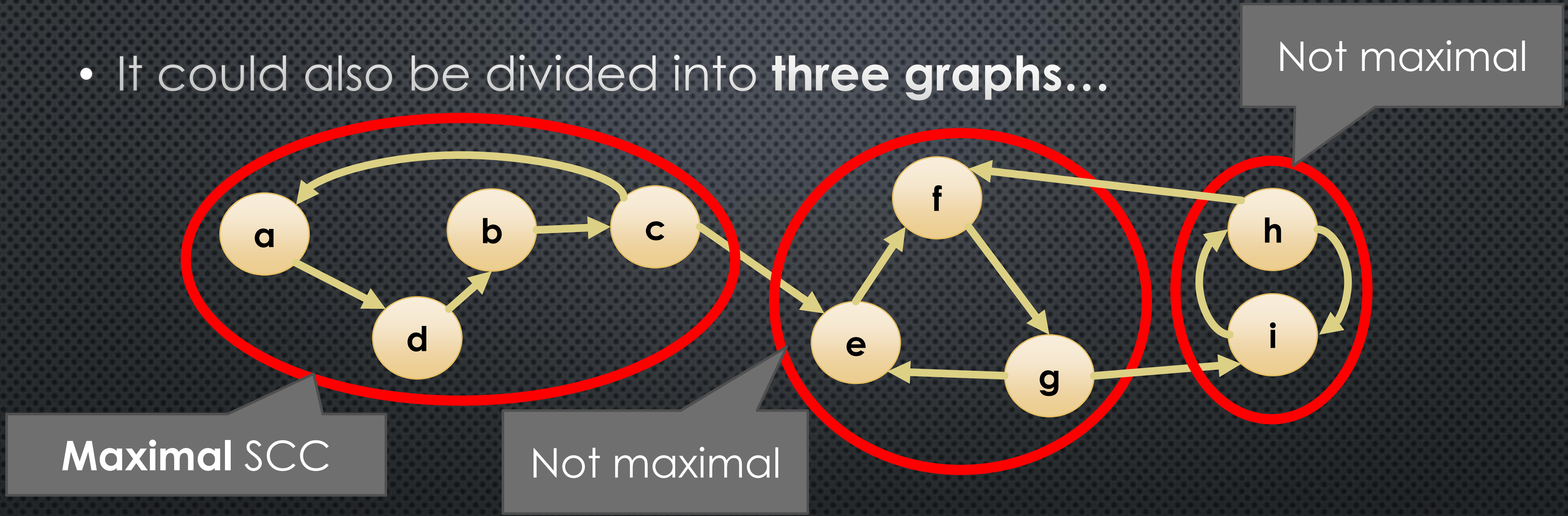
- This graph could be divided into **two graphs** that are each strongly connected



These are called **strongly connected components (SCCs)**

STRONGLY CONNECTED **COMPONENTS**

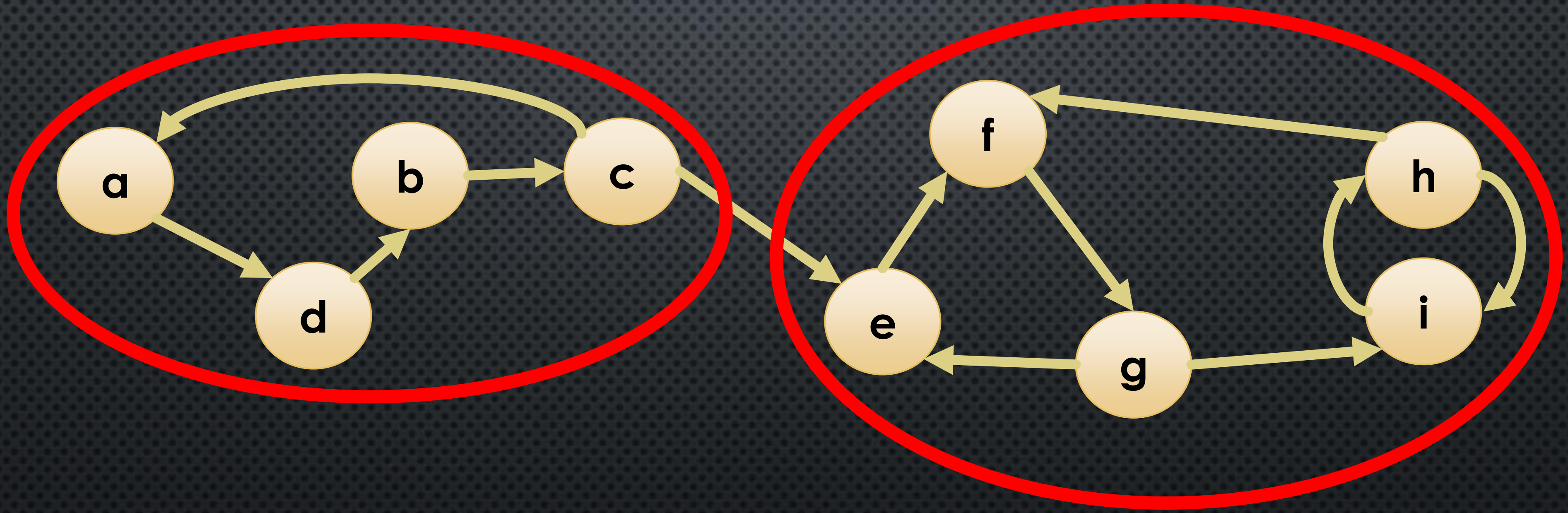
- It could also be divided into **three graphs…**



- But we want our SCCs to be **maximal** (as large as possible)

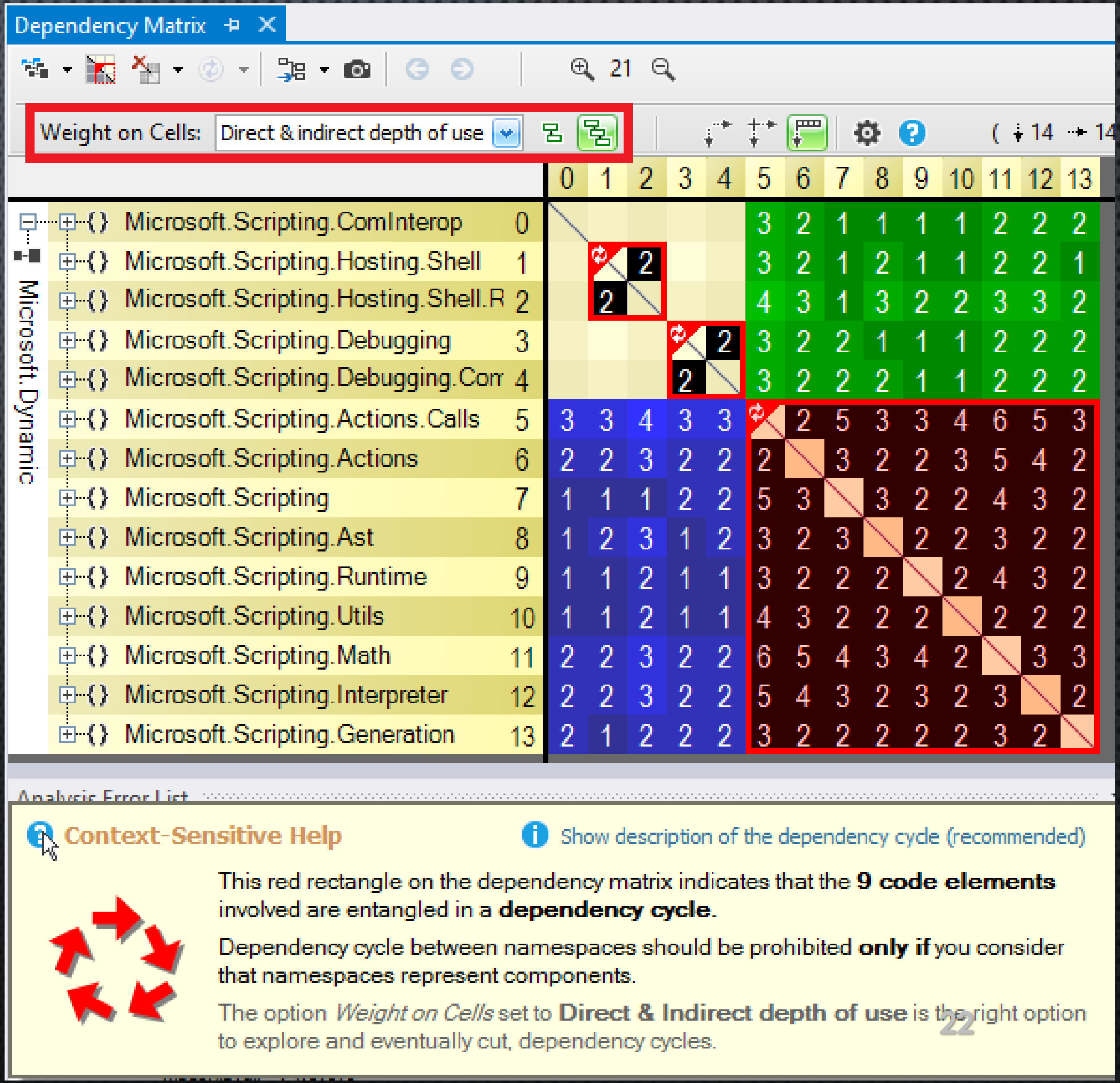
STRONGLY CONNECTED **COMPONENTS**

- So, the goal is to find **these** (maximal) SCCs:



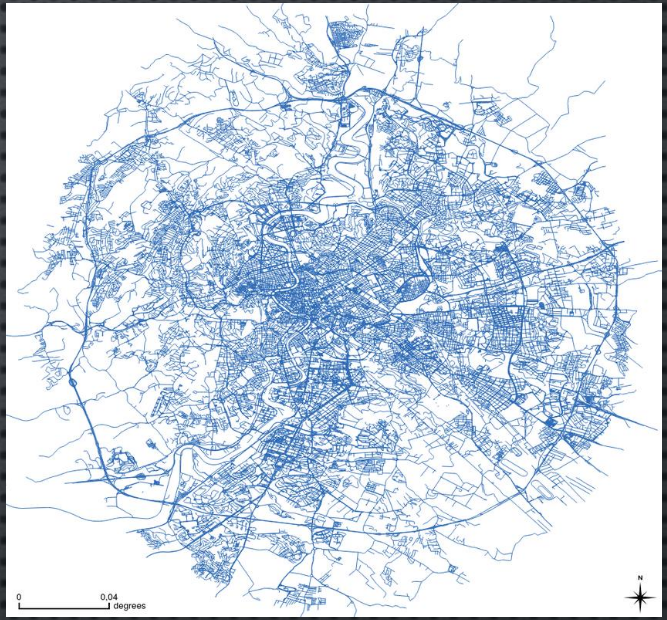
APPLICATIONS OF **SCCs** AND **COMPONENT GRAPHS**

- Finding **all cyclic** dependencies in code
  - Can find **single** cycle with an easier DFS-based algorithm
  - But it is nicer to find **all** cycles at once, so you don't have to fix one to expose another

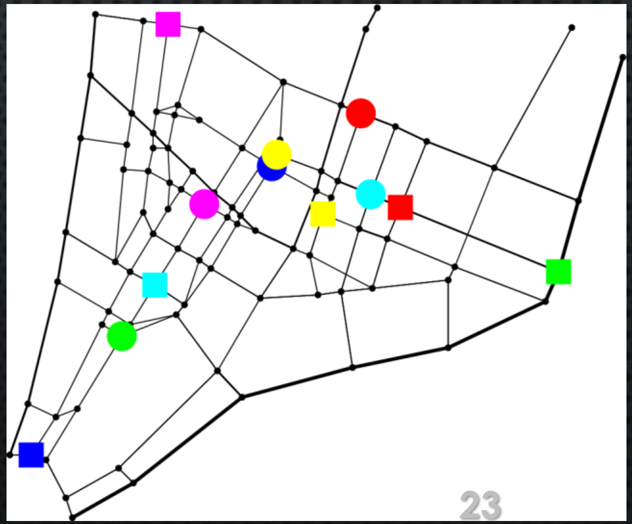


APPLICATIONS OF **SCCs** AND **COMPONENT GRAPHS**

- **Data filtering** before running other algorithms
- maps;   nodes = intersections,   edges = roads
- Don't want to run path finding algorithm on the entire **global** graph!
- Throw away everything except the (maximal) SCC containing source & target



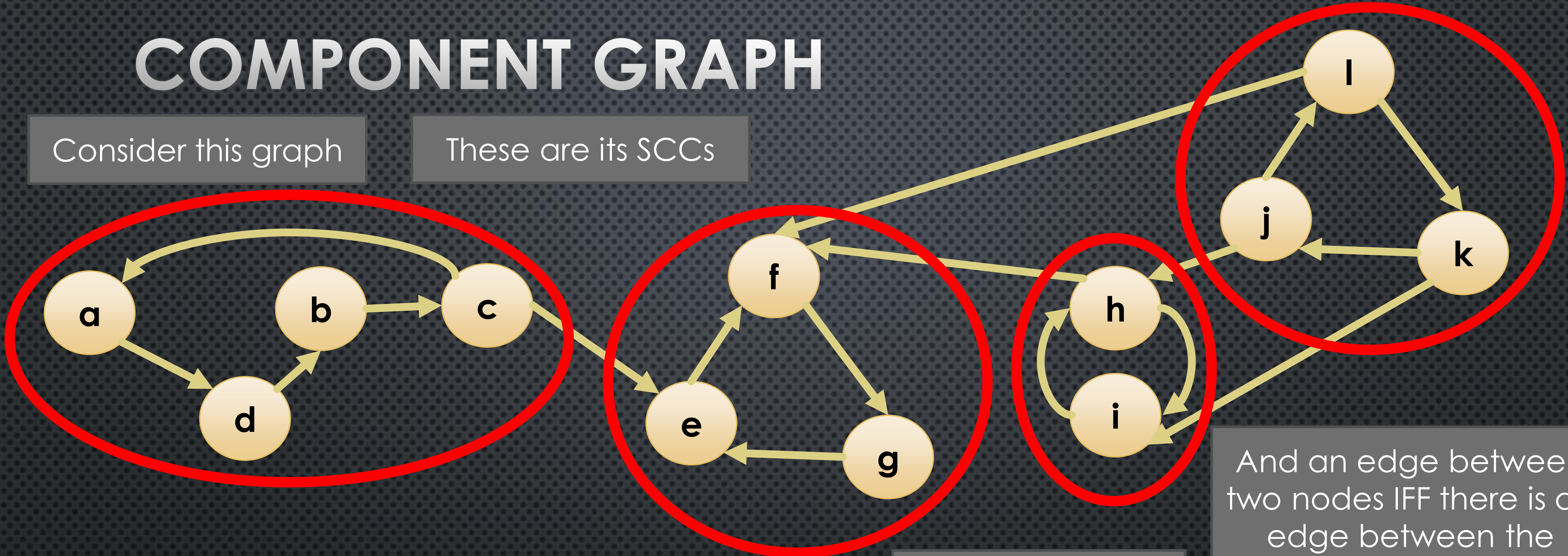
Crop & find SCCs



COMPONENT GRAPH

Consider this graph

These are its SCCs



And an edge between two nodes IFF there is an edge between the corresponding SCCs

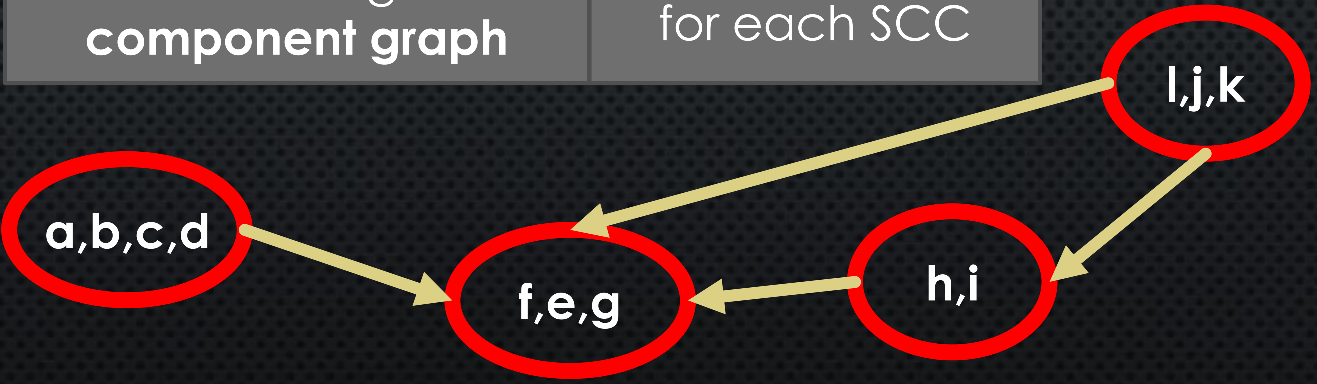
Can there be a **cycle** in the component graph?

No! If there are paths <u>both ways</u> between components, they are actually **the same SCC**

Component graph is a <u>DAG</u>!

The following is its **component graph**

It has one node for each SCC



BRAINSTORMING AN ALGORITHM

- What if we run DFS, then reverse all edges, then run DFS (like checking whether an *entire graph* is strongly connected?)

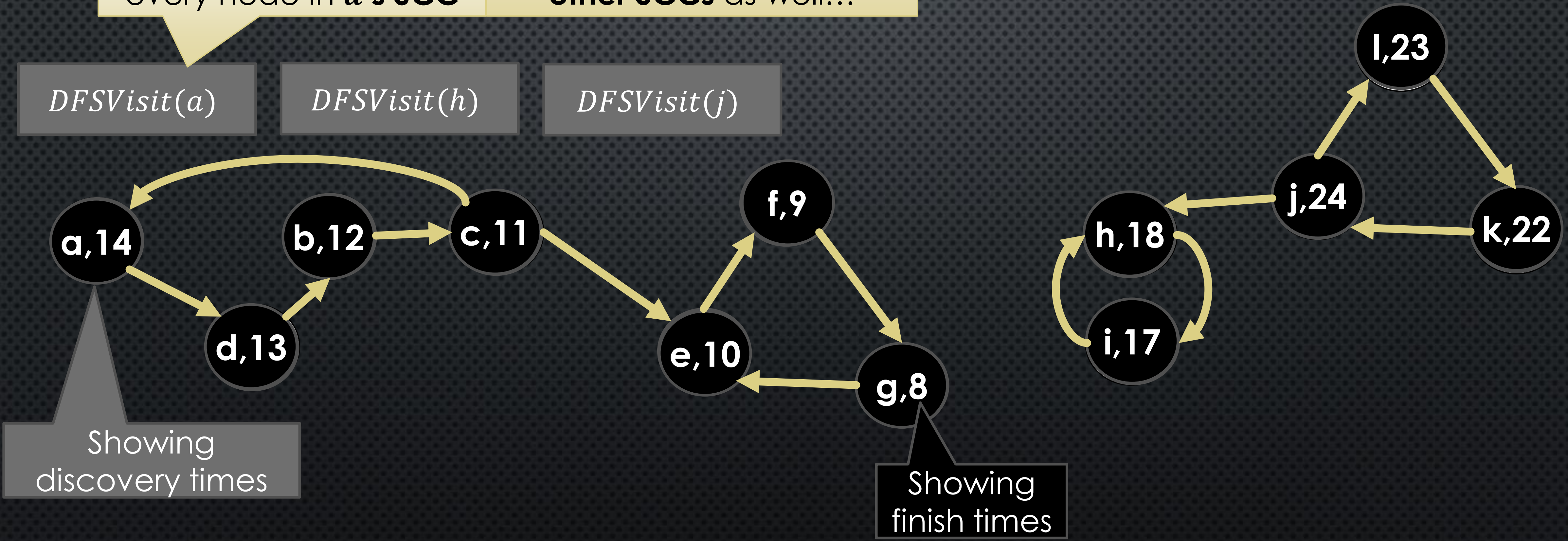
This will definitely visit every node in a's **SCC**

And in fact it might visit **other SCCs** as well…

$DFSVisit(a)$

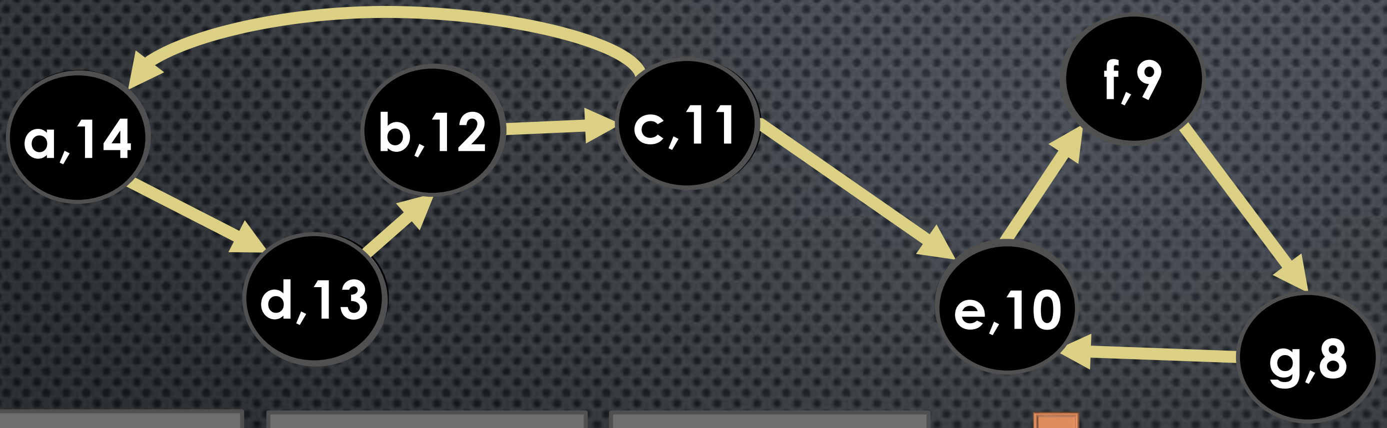
$DFSVisit(h)$

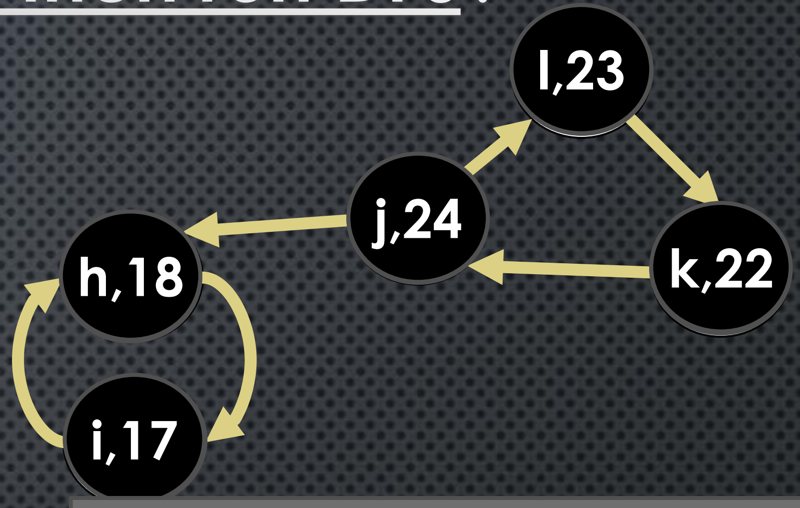
$DFSVisit(j)$



- What if we run DFS, then **reverse all edges, then run DFS**?

$DFSVisit(a)$ $DFSVisit(h)$ $DFSVisit(j)$



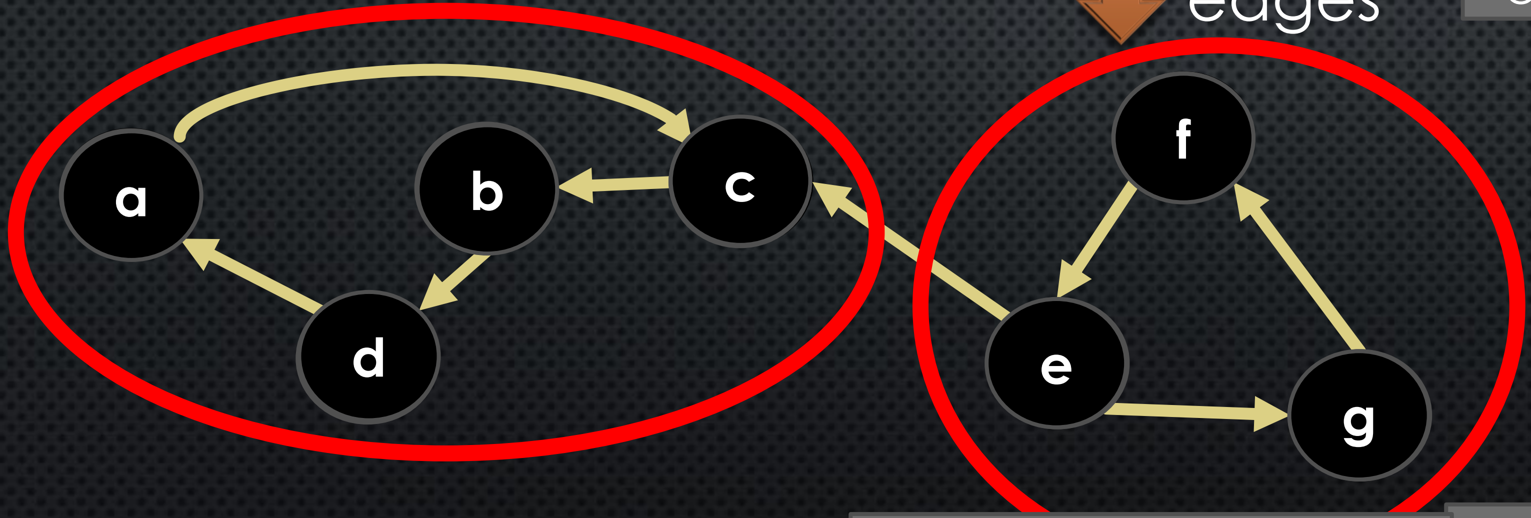


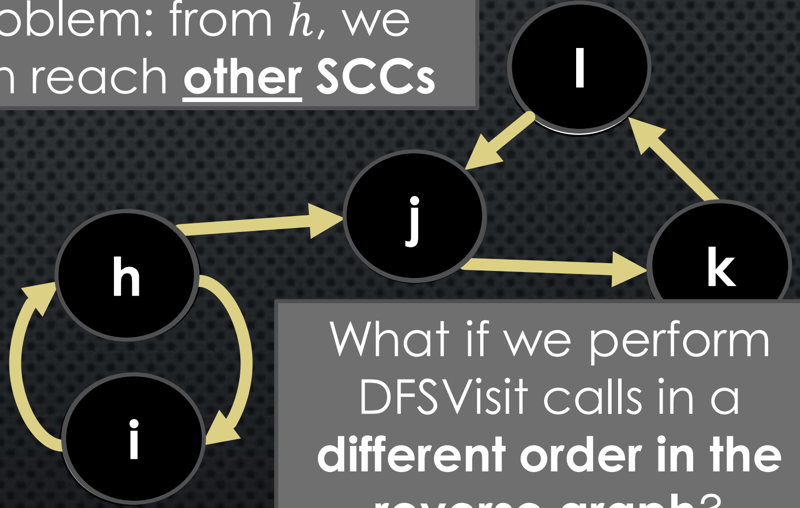
We **fail** to identify SCC { **h, i** }

$DFSVisit(a)$ $DFSVisit(e)$ $DFSVisit(h)$

reverse edges

Problem: from h, we can reach **other SCCs**





What if we perform DFSVisit calls in a **different order in the reverse graph**?

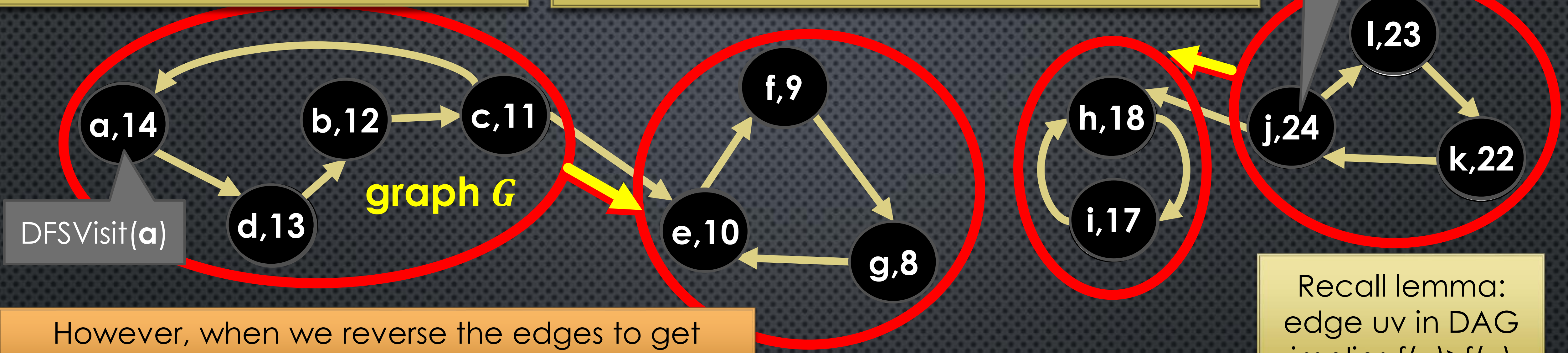
Then, each DFSVisit will visit **exactly one SCC**

(So we don't visit them again)

Other **reachable** SCCs should be **visited first**

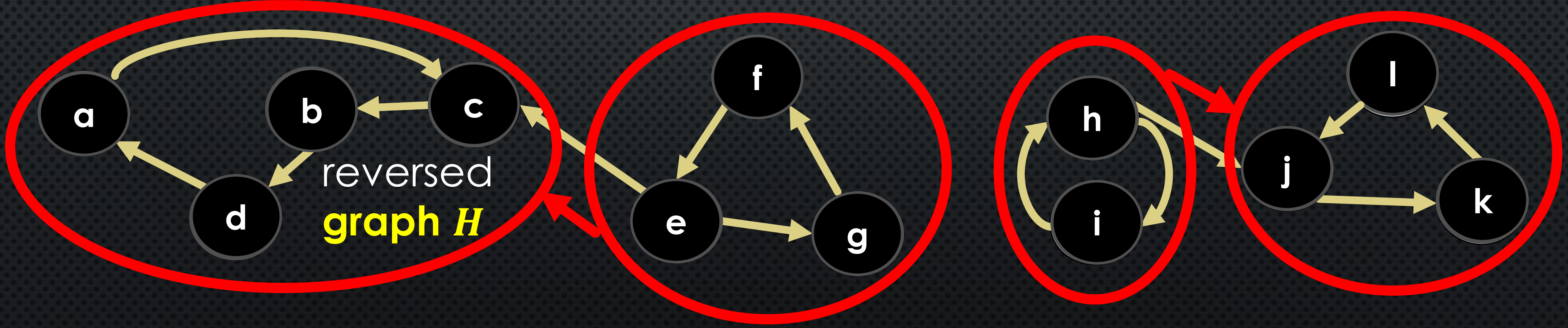
Consider **component graph C_G** of G (which we *want* to compute)

If we call DFSVisit in G from largest to smallest finish times, we can reach **other SCCs**.



However, when we reverse the edges to get graph H other SCCs can no longer be reached…

Recall lemma: edge uv in DAG implies f(u)>f(v)



SCC ALGORITHM

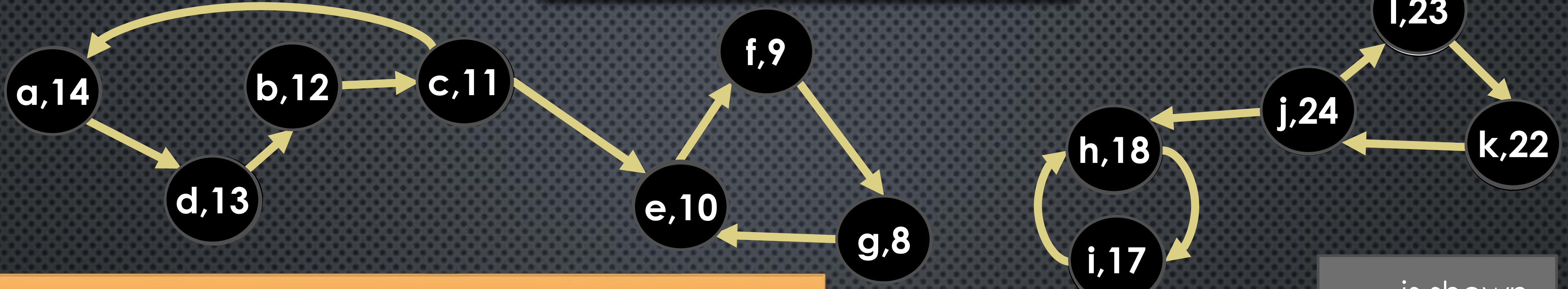
```
SCC(adj[1..n])
    DFS(adj)
    let order[1..n] = node labels sorted by
            largest to smallest finish time

    reverse all edges in adj

    colour[1..n] = [white, ..., white]
    comp[1..n] = [0, ..., 0]
    for i = 1..n
        v = order[i]
        if colour[v] == white
            scc = scc + 1
            SCCVisit(adj, v, scc, colour, comp)

    return comp
```

```
SCCVisit(adj[1..n], v, scc, colour, comp)
    colour[v] = gray
    comp[v] = scc

    for each w in adj[v]
        if colour[w] == white
            SCCVisit(w)

    colour[v] = black
```

This is called Sharir's algorithm (sometimes Kosaraju's algorithm). **This paper** first introduced it.

Running Sharir's Algorithm

Phase 1: DFS to get finish times

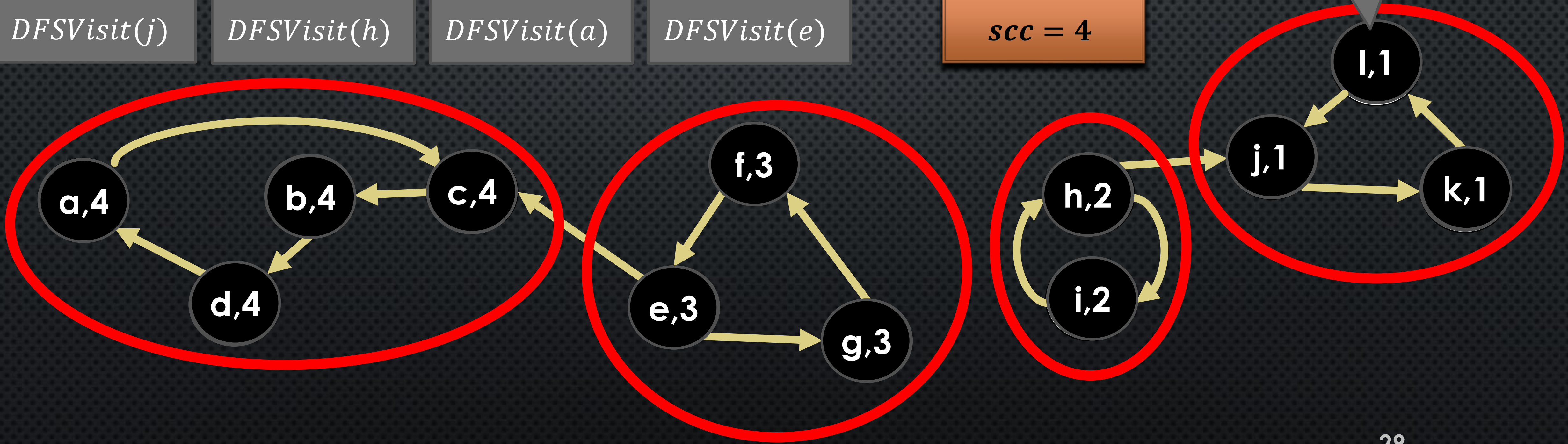


Phase 2: DFSVisit <u>reverse graph</u> by reverse finish times

$DFSVisit(j)$ $DFSVisit(h)$ $DFSVisit(a)$ $DFSVisit(e)$

$\boldsymbol{scc = 4}$

scc is shown



TIME COMPLEXITY?

```
SCC(adj[1..n])
    DFS(adj)
    let order[1..n] = node labels sorted by
            largest to smallest finish time

    reverse all edges in adj

    colour[1..n] = [white, ..., white]
    comp[1..n] = [0, ..., 0]
    for i = 1..n
        v = order[i]
        if colour[v] == white
            scc = scc + 1
            SCCVisit(adj, v, scc, colour, comp)

    return comp
```

```
SCCVisit(adj[1..n], v, scc, colour, comp)
    colour[v] = gray
    comp[v] = scc

    for each w in adj[v]
        if colour[w] == white
            SCCVisit(w)

    colour[v] = black
```

$O(n+m)$

Can be returned as part of the DFS with no added runtime

Finish times **increase** as we set them, so just use a stack...

$O(n+m)$

$O(n)$

Total of $O(n+m)$ work over all n iterations of the i loop

(each edge is inspected once, each node is visited once, constant work per visited node/inspected edge)

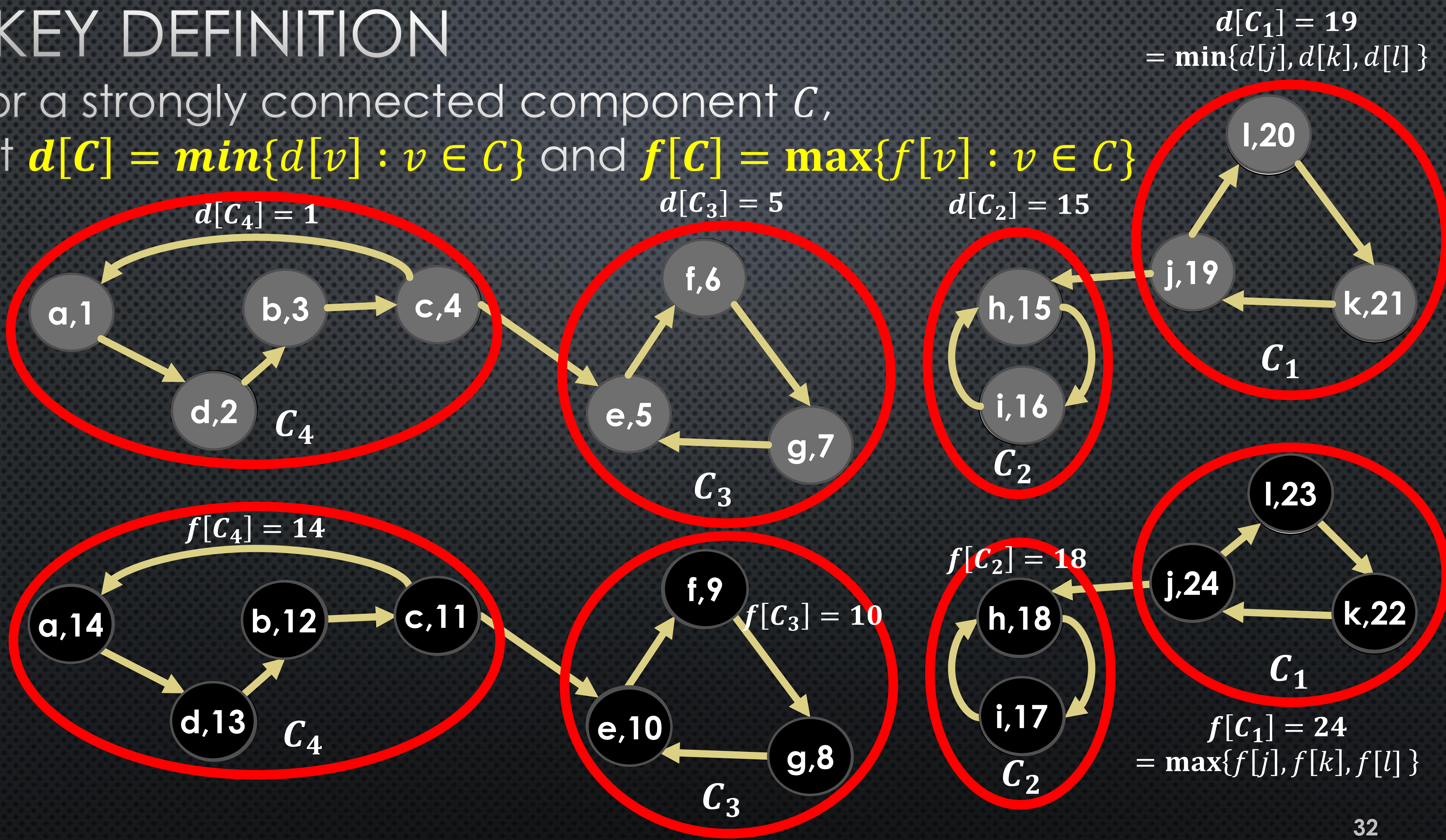
Total $O(n+m)$

CORRECTNESS

- Want to prove that each top-level call to SCCVisit explores **exactly** the nodes **in one SCC**
- Proof hinges on a key lemma that talks about the **finish times of SCCs** in the **component graph**
- To talk about finish times of **SCCs**, we need a definition…

A KEY DEFINITION

- For a strongly connected component C, let $\boldsymbol{d[C] = \min\{d[v] : v \in C\}}$ and $\boldsymbol{f[C] = \max\{f[v] : v \in C\}}$

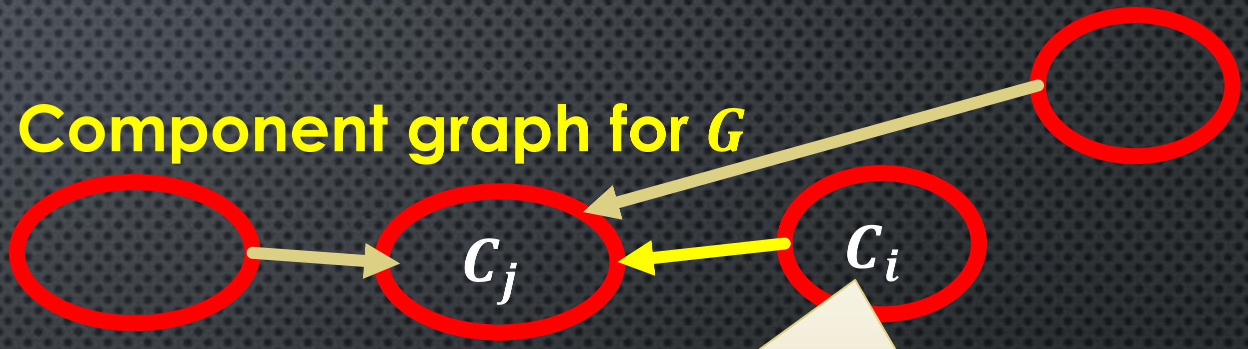


A KEY LEMMA

- **Lemma:** if C_i, C_j are SCCs and there is an edge $\boldsymbol{C_i \to C_j}$ **in** $\boldsymbol{G}$, then $\boldsymbol{f[C_i] > f[C_j]}$

$\boldsymbol{C_i}$ discovered first

Component graph for G



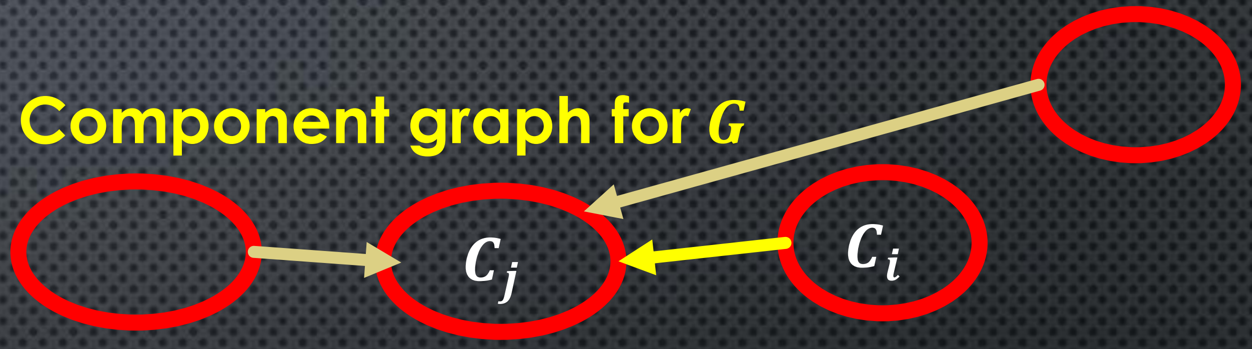
$\boldsymbol{u}$ = earliest discovered node in here

- **Proof.** Case 1 ($\boldsymbol{d[C_i] < d[C_j]}$):
  - Let u be the earliest discovered node in C_i
  - All nodes in $C_i \cup C_j$ are white-reachable from u, so they are **descendants in the DFS forest** and **finish before u**
  - So $f[C_i] = f[u] > f[C_j]$

A KEY LEMMA

- **Lemma:** if C_i, C_j are SCCs and there is an edge $\boldsymbol{C_i \to C_j}$ **in** $\boldsymbol{G}$, then $\boldsymbol{f[C_i] > f[C_j]}$

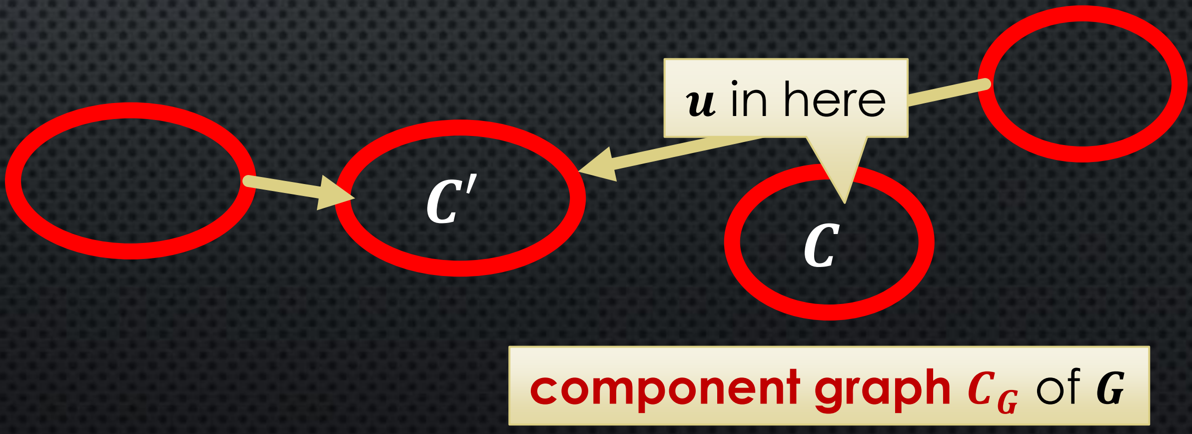
C_j discovered first



- **Proof.** **Case 2 ($d[C_j] < d[C_i]$):**
  - Since component graph is a DAG, there is **no path** $\boldsymbol{C_j \to C_i}$
  - Thus, **no nodes** in C_i are reachable from C_j
  - So we discover C_j and finish C_j **without** discovering C_i
  - Therefore $d[C_j] < f[C_j] < d[C_i] < f[C_i]$. **QED**

COMPLETING THE PROOF

- Suppose we have performed DFS to get our finish times, and we are about to perform SCCVisits on the reverse graph
- **We prove each top-level SCCVisit call visits precisely one SCC**
- Consider the first top-level SCCVisit(u)
- Let C be the SCC containing u and C' be any other SCC
- Since we call SCCVisit on nodes starting from the **largest finish time**,
  - We know $f(C) > f(C')$



COMPLETING THE PROOF

- We know $\boldsymbol{f(C) > f(C')}$
- By Lemma: if there were an edge $C' \to C$ in G, then we would have $f(C') > f(C)$
  - So there is no edge $C' \to C$ in G
  - and hence **no edge $C \to C'$ in H**
  - So, **SCCVisit(u) in H cannot visit C'**

… and sets comp[v] = scc for all nodes in the SCC

So each top-level call explores one SCC…

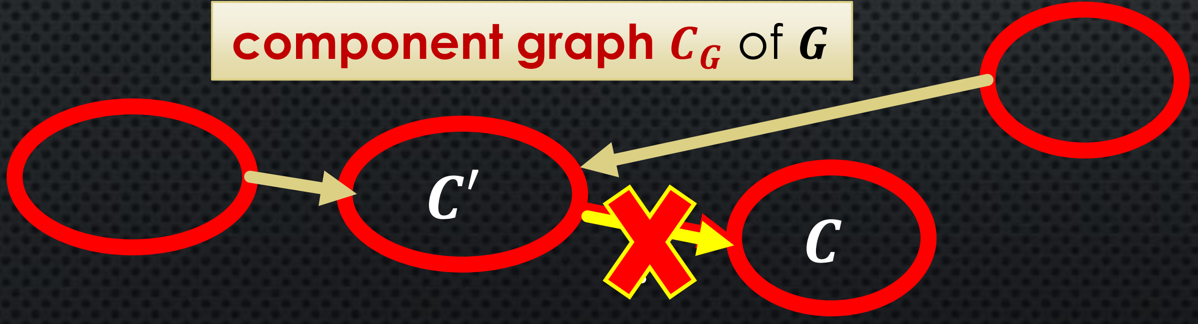
and **larger finish** time means **already explored!**

In G, edges go from larger to smaller finish times. **In H, edges go from smaller to larger.**

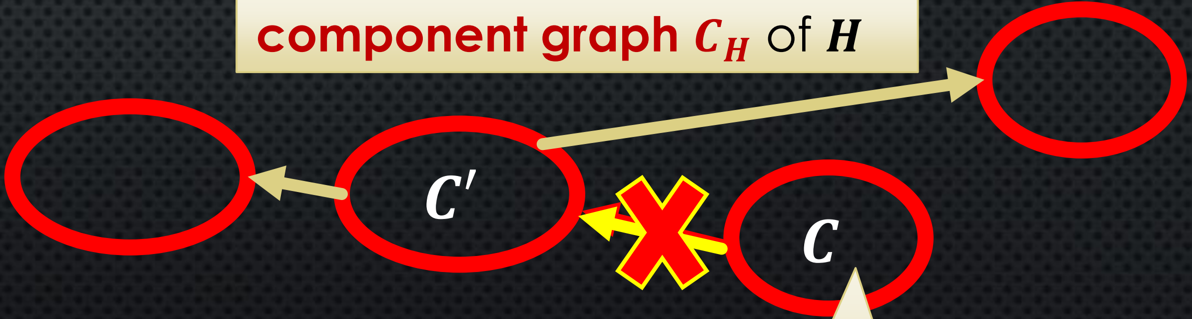
Similar argument for subsequent **top-level** calls to SCCVisit.

So SCCVisit(u) visits <u>exactly</u> the nodes in C

component graph C_G of G



component graph C_H of H



u in here

IF WE HAVE TIME

topological sort without relying on DFS

EXISTENCE OF A TOPOLOGICAL SORT ORDER

Theorem 6.6

A directed graph D has a topological sort if and only if it is a DAG.

Proof.

($\Rightarrow$): Suppose D has a directed cycle $v_1, v_2, \ldots, v_j, v_1$. Then $v_1 < v_2 < \cdots < v_j < v_1$, so a topological ordering does not exist.

($\Leftarrow$): Suppose D is a DAG. Then the algorithm below constructs a topological ordering. $\square$

```
Kahn(adj[1..n])
    indeg[1..n] = [0, ..., 0]
    for each edge (u,v) in adj
        indeg[v] = indeg[v] + 1

    order = new list
    q = new queue containing {v : indeg[v] == 0}
    for i = 1..n
        if q.empty() return null
        v = q.dequeue()
        order.append(v)

        for each w in adj[v]
            indeg[w] = indeg[w] - 1
            if indeg[w] == 0 then q.enqueue(w)

    return order
```

$indeg[v]$ = # of edges pointing **into** node v

= number of **unsatisfied constraints** on v

Nodes with $\boldsymbol{indeg}$ **0** have **no unsatisfied dependencies**

So this step is enqueuing nodes whose dependencies are already satisfied

$\boldsymbol{q}$ **always** contains nodes with no unsatisfied dependencies (indeg 0)

No such order!

Add v to the topological order

Remove v's out edges. If we have now satisfied all dependencies for some w, add w to the queue also.

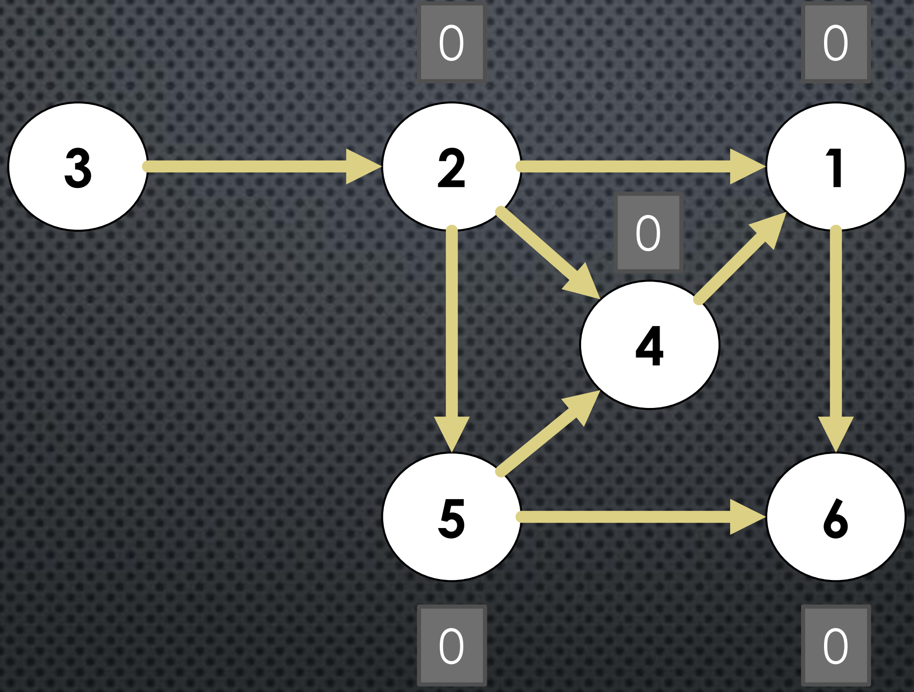
EXAMPLE (**KAHN'S** ALGORITHM)

Compute **indegree** for all vertices

For each node u
For each w in adj(u)
w.deg = w.deg+1

vertices with indeg 0
go into the queue

Until Q is empty: pop, output that element, decrement its neighbours, enqueue new indeg 0's



Queue Q

3 2 5 4 1 6

Output

```
Kahn(adj[1..n])
    indeg[1..n] = [0, ..., 0]
    for each edge (u,v) in adj
        indeg[v] = indeg[v] + 1

    order = new list
    q = new queue containing {v : indeg[v] == 0}
    for i = 1..n
        if q.empty() return null
        v = q.dequeue()
        order.append(v)

        for each w in adj[v]
            indeg[w] = indeg[w] - 1
            if indeg[w] == 0 then q.enqueue(w)

    return order
```

$O(n)$

$O(n+m)$ total work over all iterations

$O(n)$ iterations

$O(1)$ per check

$O(1)$

$O(\deg(v))$ per iteration i

$\sum_{v \in V} \deg(v) \in \boldsymbol{O(n+m)}$

total work over <u>all</u> nodes v

Running time with adjacency lists?

Total $O(n+m)$

BONUS SLIDES

SCC: HOW ABOUT A DIFFERENT ORDERING?

- Rather than doing DFS in the **reverse** graph in order of **decreasing** finish times
- Why not do DFS in the **original** graph in order of **increasing** finish times?
- Exercise: does this work?

SCC: HOW ABOUT A DIFFERENT ORDERING?

- Why not do DFS in the **original** graph in order of **increasing** finish times?

Doesn't work!

Output depends where first DFS starts…

If first DFS starts at c, then…

DFSVisit(b) would reach <u>two</u> SCCs.