CS 341: ALGORITHMS

Lecture 11: graph algorithms II – finishing BFS, depth first search

Readings: see website

Trevor Brown

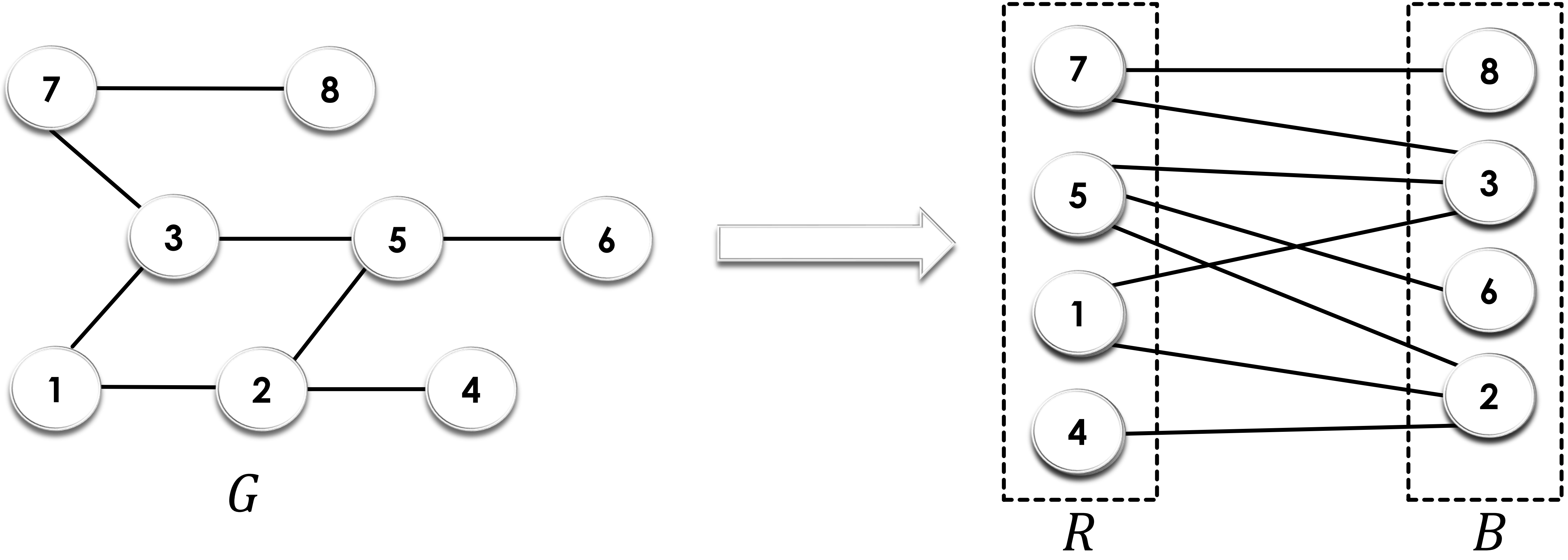
https://student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca

BFS APPLICATION:
TESTING WHETHER A GRAPH IS **BIPARTITE**

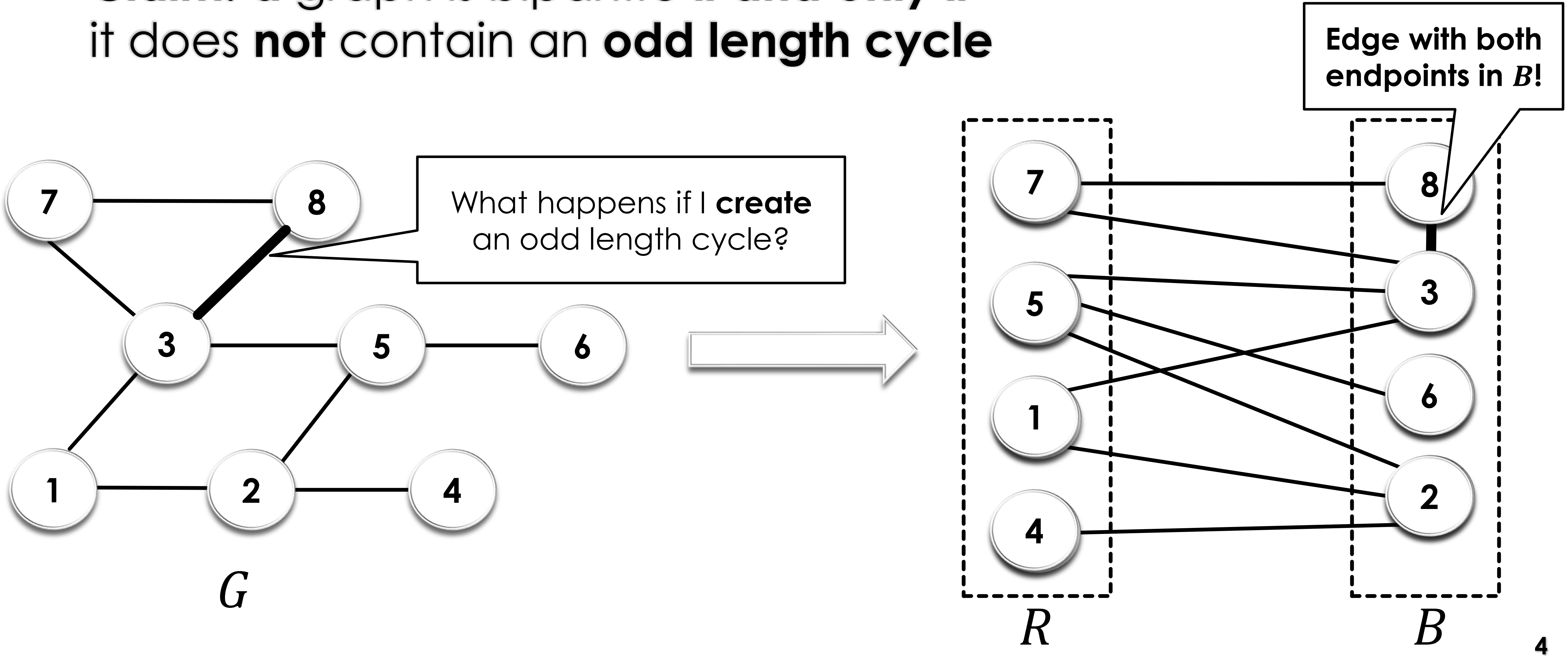
(UNDIRECTED) BIPARTITE GRAPHS AND BFS

- A graph is **bipartite** if the nodes can be **partitioned** into sets R and B such that **each edge** has one endpoint in R and one endpoint in B



CRUCIAL PROPERTY:
NO ODD CYCLES

- **Claim: a** graph is bipartite **if and only if** it does **not** contain an **odd length cycle**

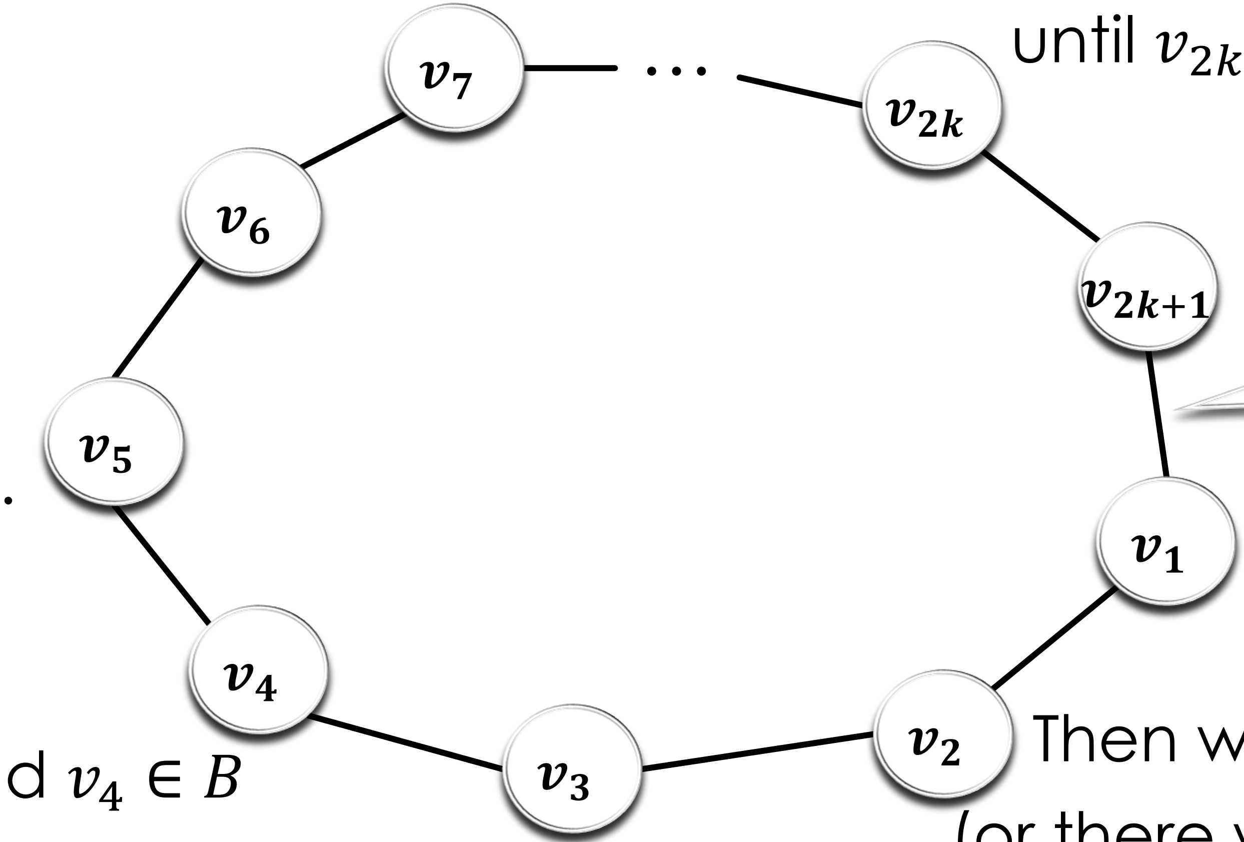

What happens if I **create** an odd length cycle?

Edge with both endpoints in B!

PROOF

PART 1: ODD CYCLE ⇒ NOT BIPARTITE

- Suppose there is an **odd** length cycle $v_1, v_2, \dots, v_{2k+1}, v_1$



until $v_{2k} \in B$

And finally $v_{2k+1} \in R$!!

Both endpoints in R! Contradiction!

And so on, alternating…

WLOG let $v_1 \in R$

And $v_4 \in B$

Then we must have $v_2 \in B$

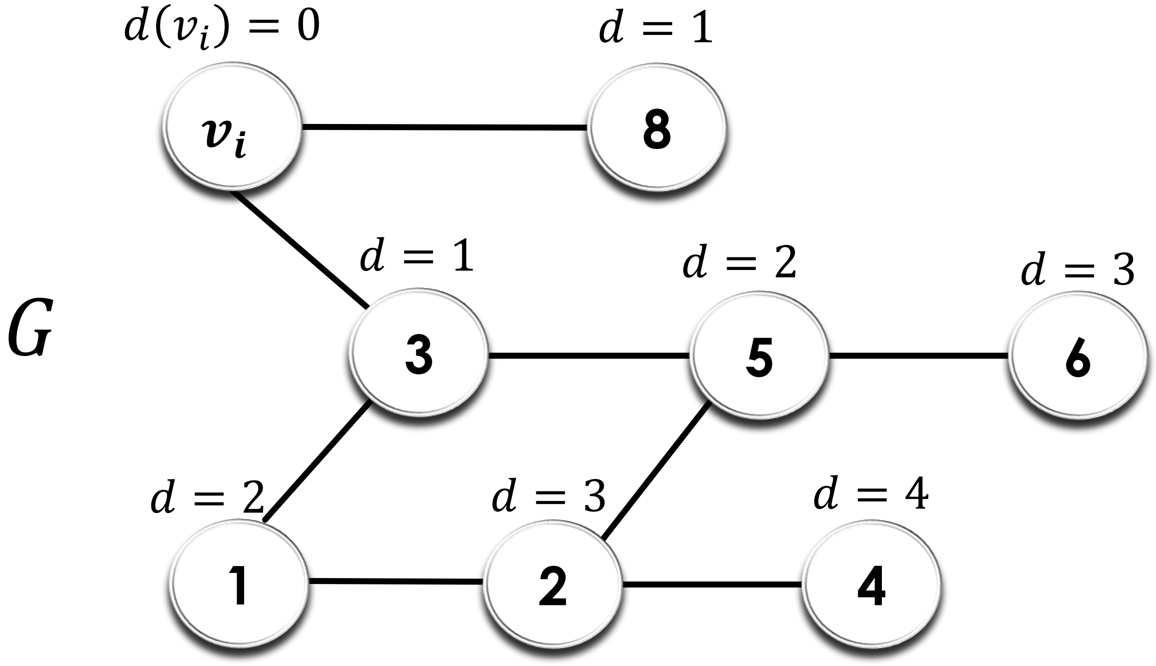
(or there will be an edge (v_1, v_2) with two endpoints in R)

And $v_3 \in R$

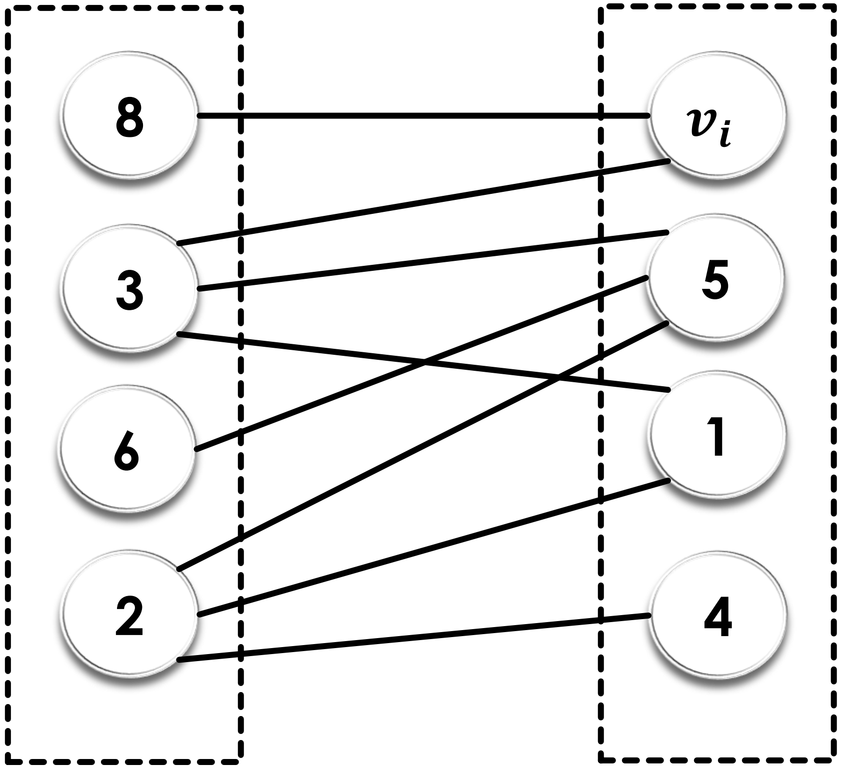
PROOF

PART 2: ALL CYCLES HAVE EVEN LENGTH ⇒ BIPARTITE

- Let v_i be any node, and $\boldsymbol{d(v)}$ be the distance **from $\boldsymbol{v_i}$ to $\boldsymbol{v}$**
- Partition nodes by even vs odd distances



WTP: no edge between red nodes
no edge between blue nodes

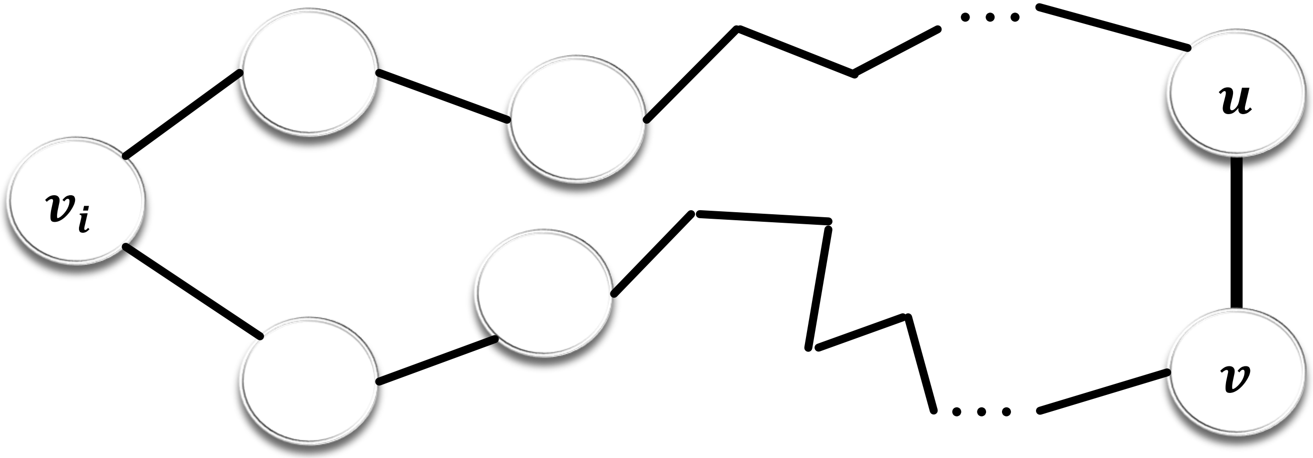


$R =$ **odd** $d(v)$ $B =$ **even** $d(v)$

BAD EDGES MEAN ODD CYCLES

- **Claim:** if there were an edge between red nodes, or between blue nodes, there would be an **odd length cycle**
- WLOG suppose for contradiction $(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{E}$ **where** $\boldsymbol{u}, \boldsymbol{v} \in \boldsymbol{R}$
- Since $u, v \in R$, distances $\boldsymbol{d(u)}$ and $\boldsymbol{d(v)}$ **from** $\boldsymbol{v_i}$ are **both odd**

Recall $\boldsymbol{d(u)}$ **= length of shortest path** $\boldsymbol{v_i \rightarrow \cdots \rightarrow u}$



$d(u) =$ **odd**

$d(v) =$ **odd**

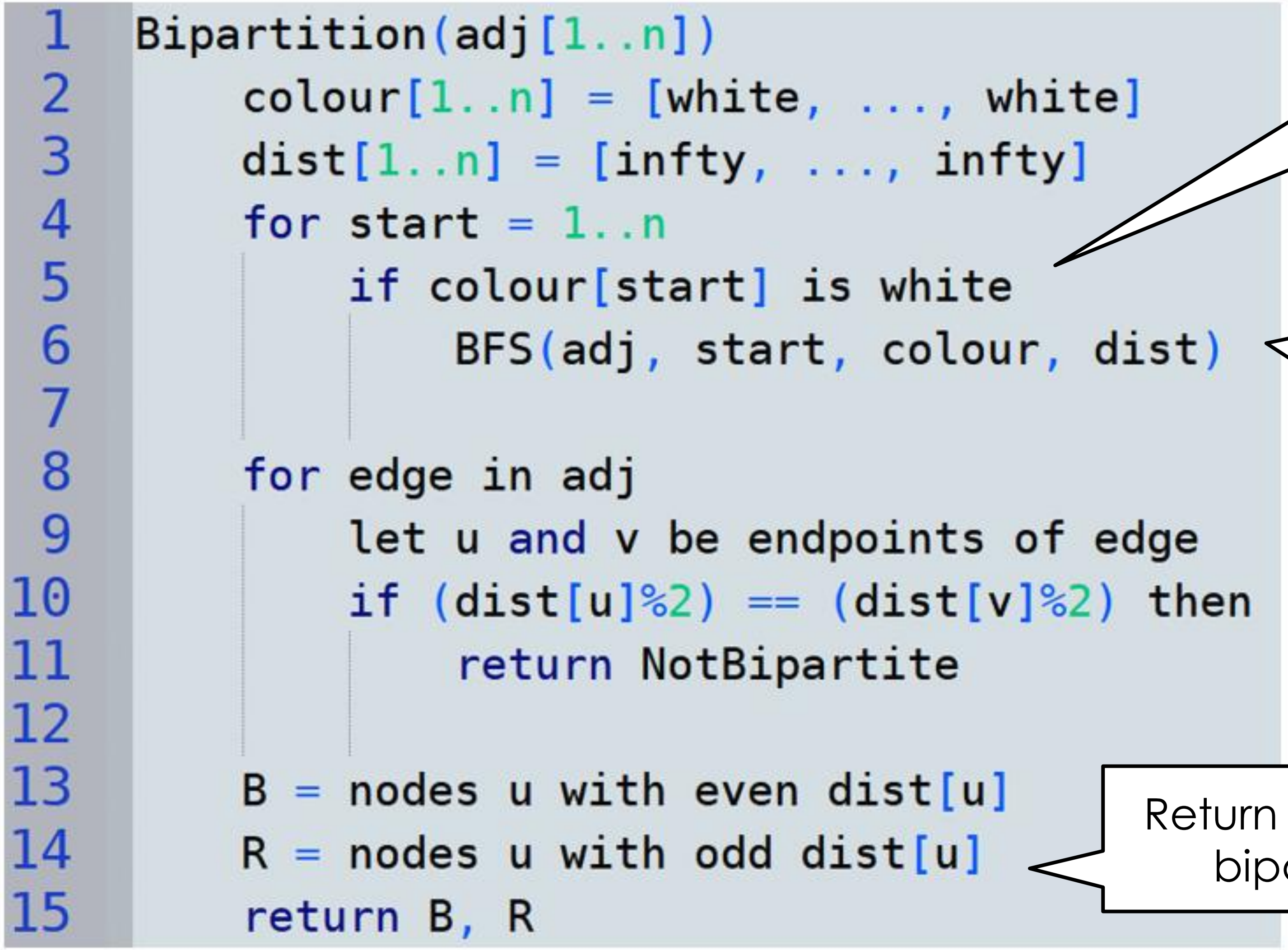
…and $\boldsymbol{d(v)}$ **the shortest path** $\boldsymbol{v_i \rightarrow \cdots \rightarrow v}$

The combined path
$\boldsymbol{v_i} \rightarrow \cdots \rightarrow u \rightarrow v \rightarrow \cdots \rightarrow \boldsymbol{v_i}$
forms a cycle

And its length is
$d(u) + 1 + d(v)$
which is odd!

So there is no edge $(\boldsymbol{u}, \boldsymbol{v})$ **where** $\boldsymbol{u}, \boldsymbol{v} \in \boldsymbol{R}$ (case $\boldsymbol{B}$ is similar)

ALGORITHM FOR TESTING BIPARTITENESS



```
Bipartition(adj[1..n])
    colour[1..n] = [white, ..., white]
    dist[1..n] = [infty, ..., infty]
    for start = 1..n
        if colour[start] is white
            BFS(adj, start, colour, dist)

    for edge in adj
        let u and v be endpoints of edge
        if (dist[u]%2) == (dist[v]%2) then
            return NotBipartite

    B = nodes u with even dist[u]
    R = nodes u with odd dist[u]
    return B, R
```

Call BFS on each component to calculate distances for each node

Modified BFS that reuses the same colour array and same dist array

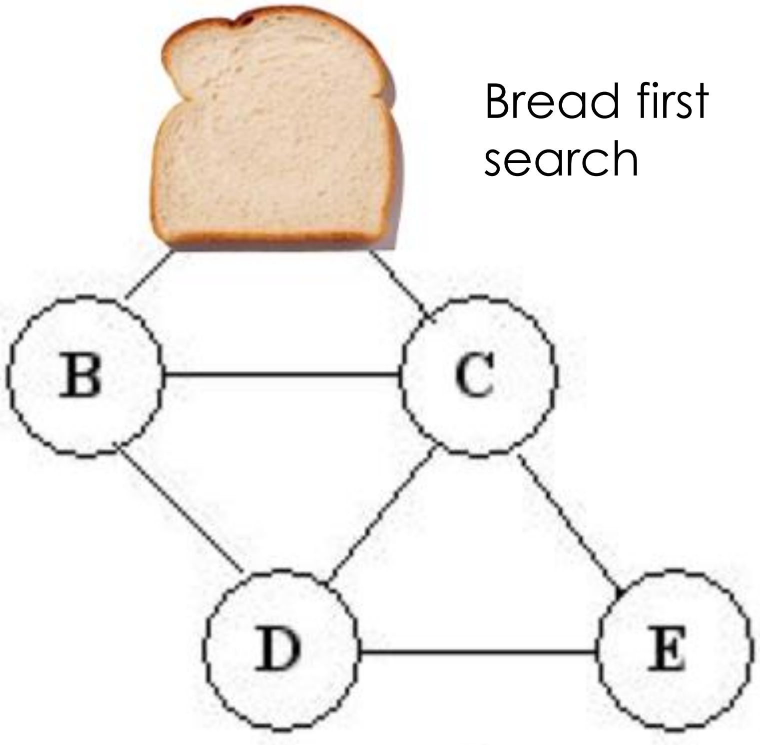
If both even or both odd

Return an actual bipartition

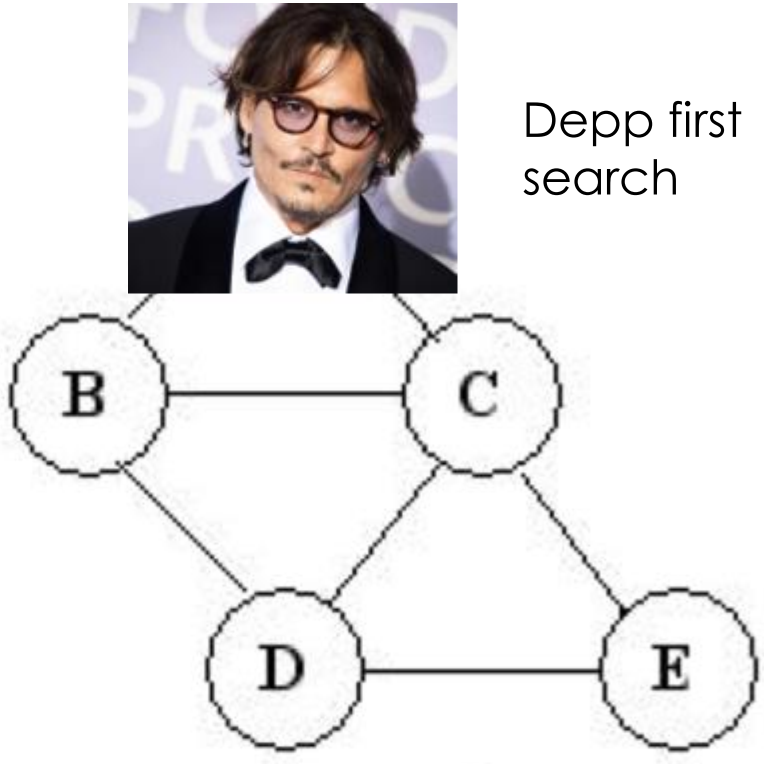
Runtime complexity?

Can be done in $O(n + m)$

DEPTH FIRST SEARCH

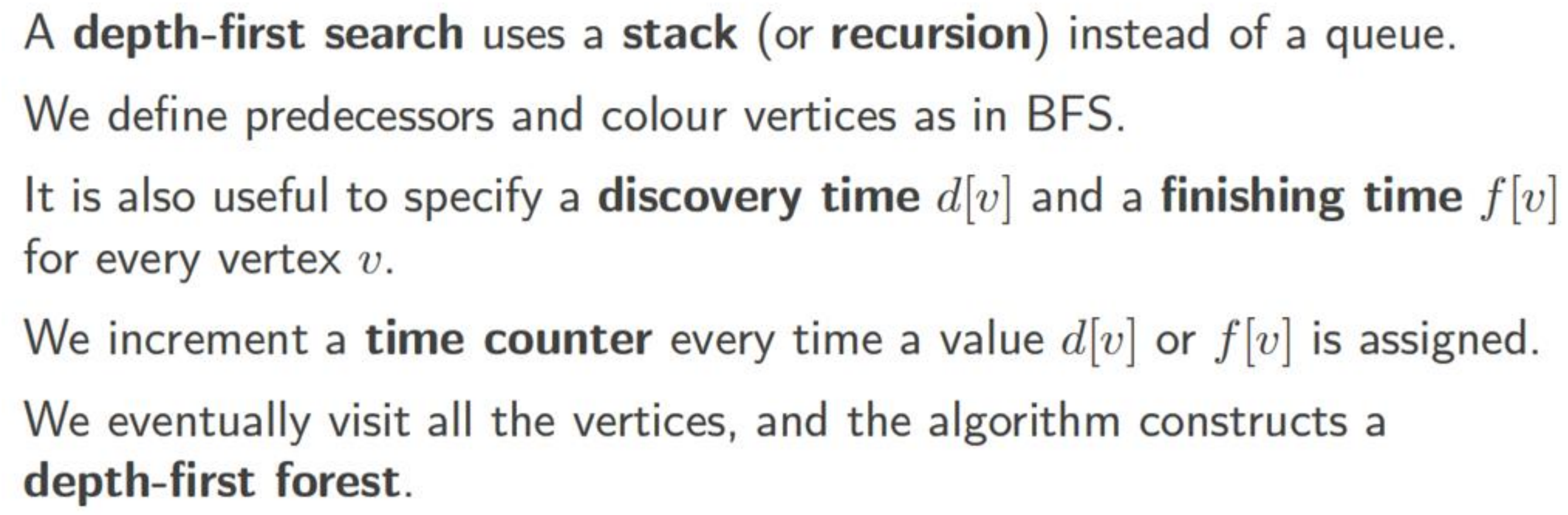


Bread first
search



Depp first
search

DEPTH-FIRST SEARCH OF A **DIRECTED** GRAPH



A **depth-first search** uses a **stack** (or **recursion**) instead of a queue.

We define predecessors and colour vertices as in BFS.

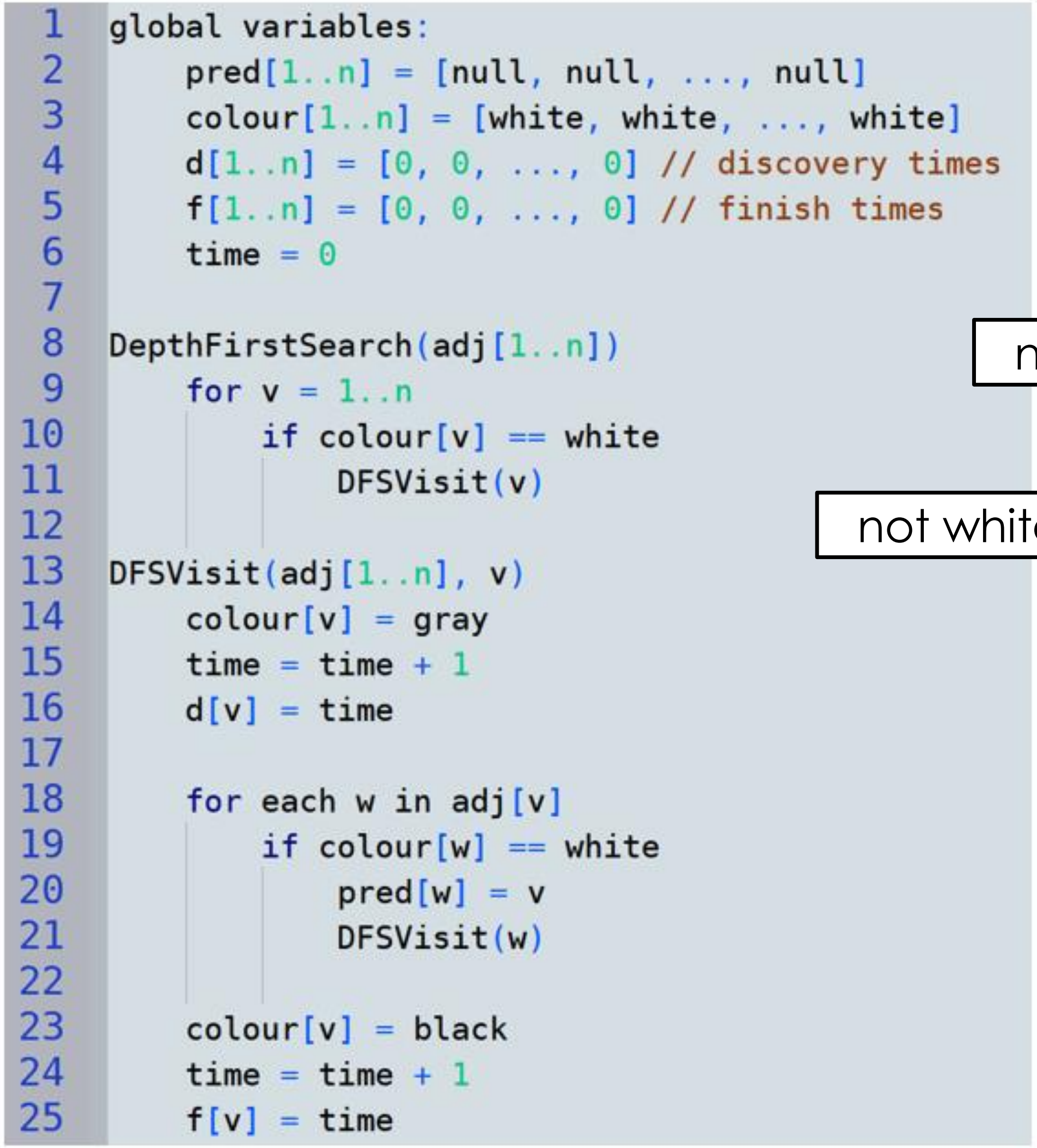
It is also useful to specify a **discovery time** $d[v]$ and a **finishing time** $f[v]$ for every vertex v.

We increment a **time counter** every time a value $d[v]$ or $f[v]$ is assigned.

We eventually visit all the vertices, and the algorithm constructs a **depth-first forest**.

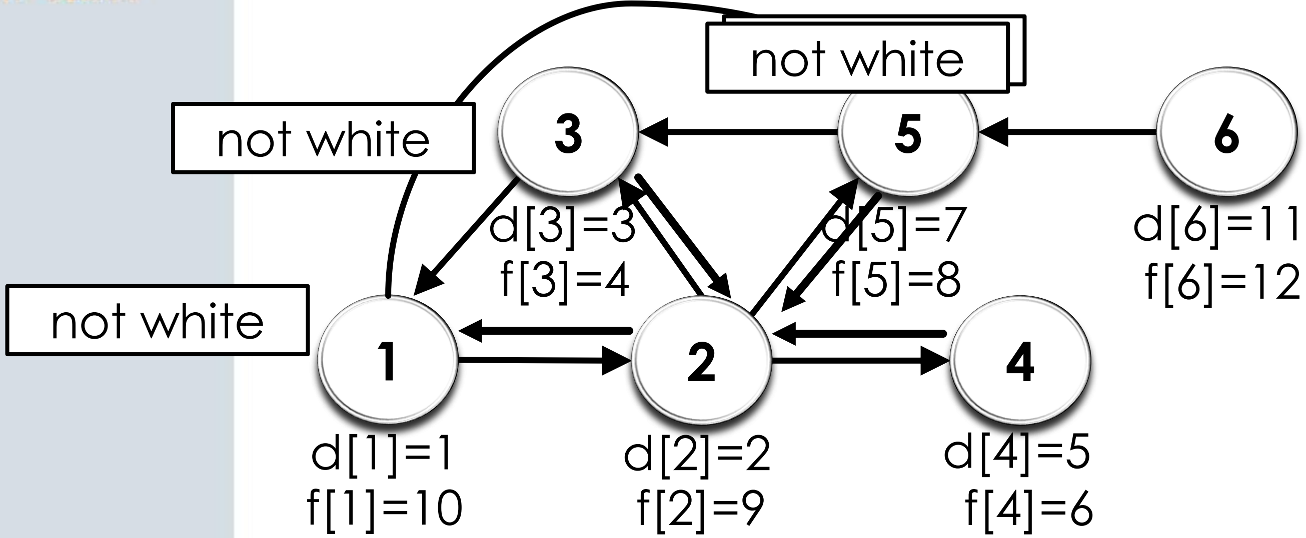
DEPTH FIRST SEARCH ALGORITHM

Example execution starting at node 1



```
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

DepthFirstSearch(adj[1..n])
    for v = 1..n
        if colour[v] == white
            DFSVisit(v)

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)

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

time = 12



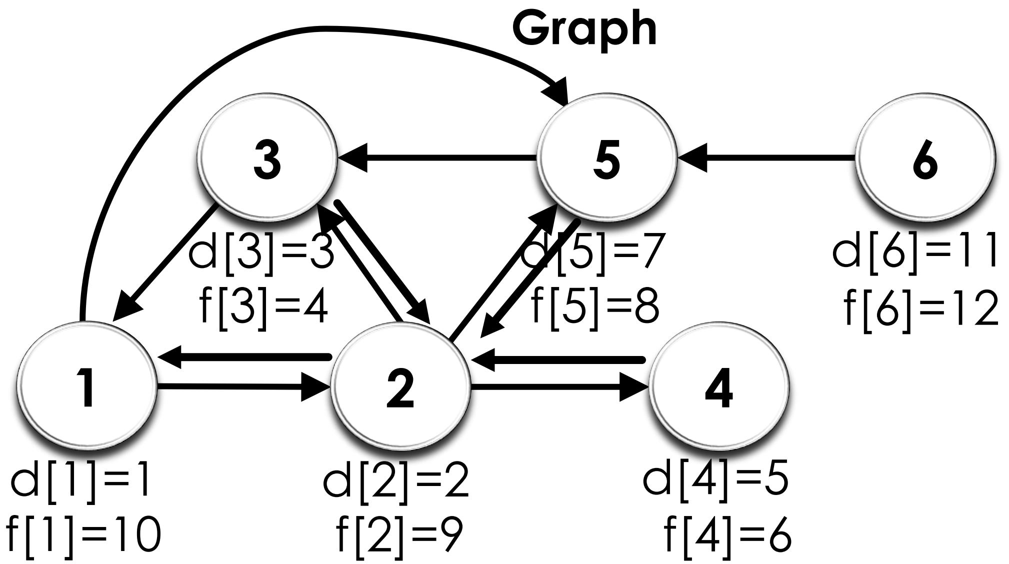
DFSVisit(1)

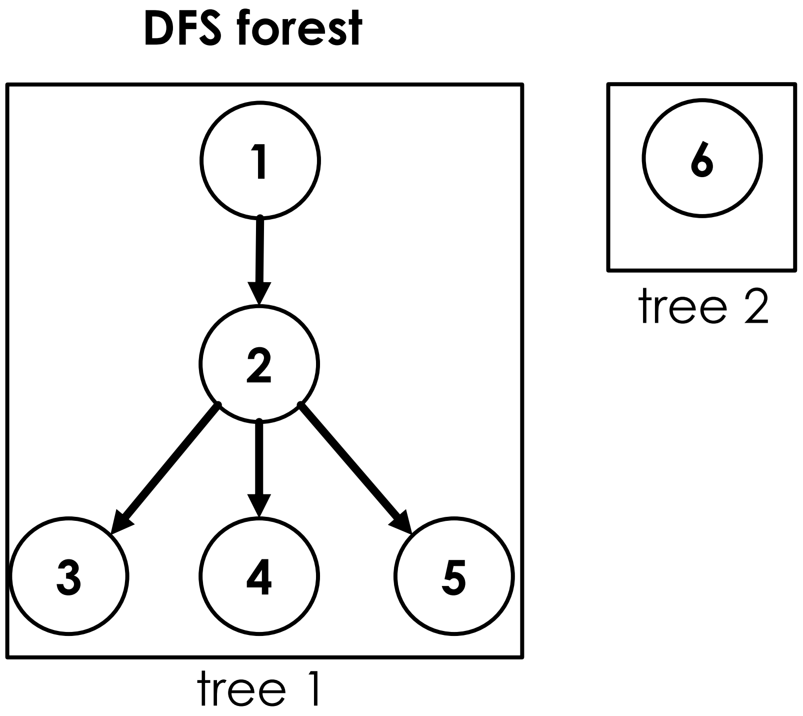
DFSVisit(6)

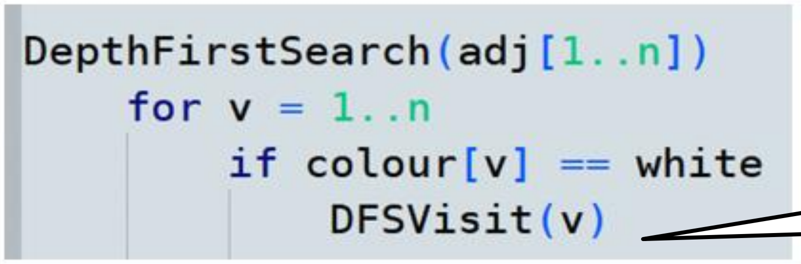
DFS TREE / FOREST

Could draw BFS forest this way also…

- As in breadth first search, $\boldsymbol{pred}$[] array induces a **forest**
- Let's match the graph's edge directions (opposite from pred)







```
DepthFirstSearch(adj[1..n])
    for v = 1..n
        if colour[v] == white
            DFSVisit(v)
```

Each top level DFSVisit call is the root of a tree

Recall: DFSVisit(1), DFSVisit(6)

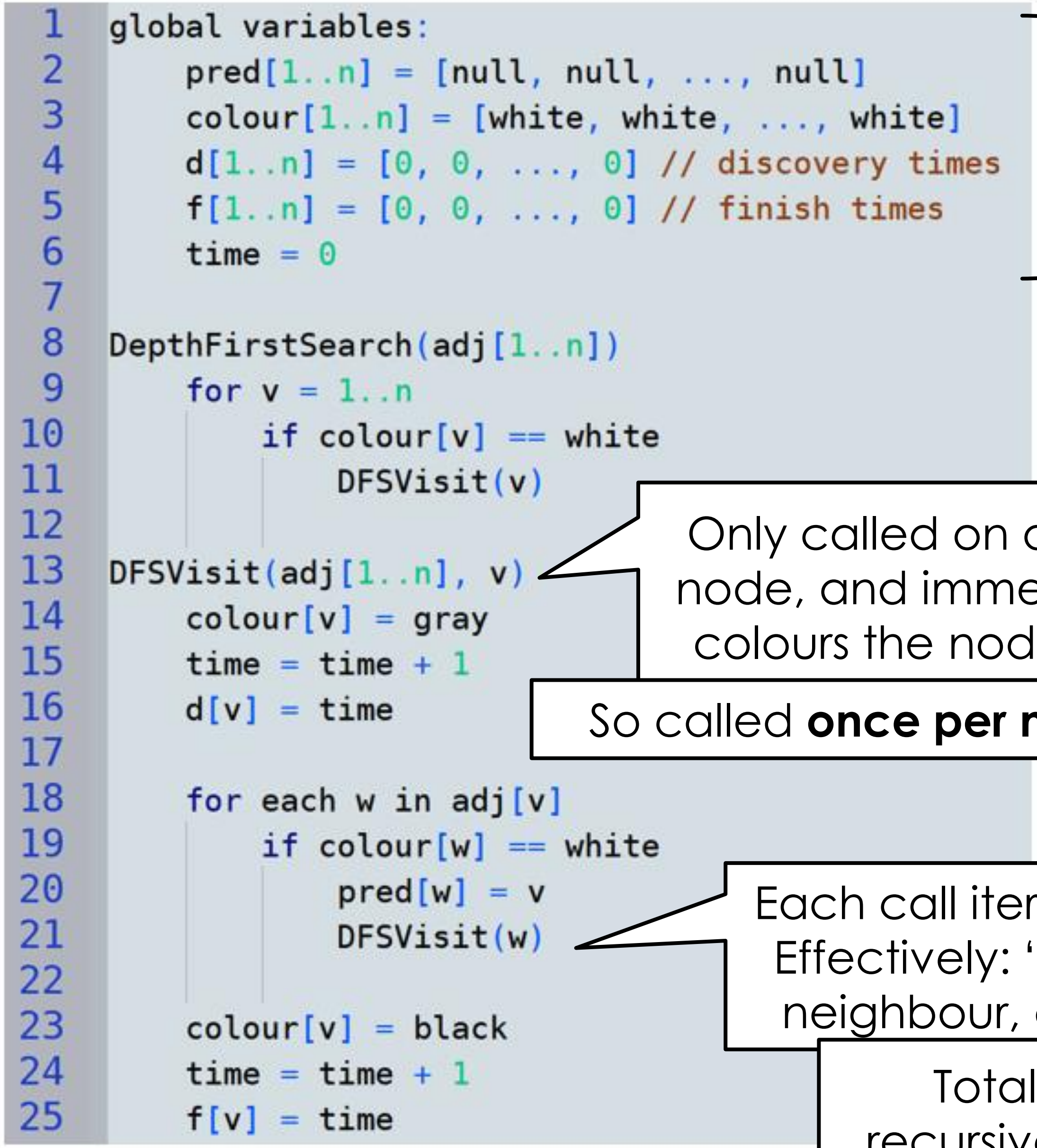
BASIC DFS PROPERTIES TO REMEMBER

- Nodes start **white**
- A node v turns **gray** when it is **discovered**, which is when the first call to $DFSVisit(v)$ happens
- **After** v is turned **gray**, we recurse on its neighbours
- After recursing on **<u>all</u> neighbours**, we turn v **black**
    - Recursive calls on neighbours end before $DFSVisit(v)$ does, so the neighbours of v turn black before v

Also gets a **discovery time** $d[v]$ at this point

Also gets a **finish time** $f[v]$ at this point

RUNTIME COMPLEXITY OF DFS **(FOR 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

DepthFirstSearch(adj[1..n])
    for v = 1..n
        if colour[v] == white
            DFSVisit(v)

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)

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

$O(n)$

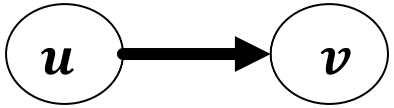
Home exercise: complexity with adjacency matrix?

Only called on a white node, and immediately colours the node gray

So called **once per node**!

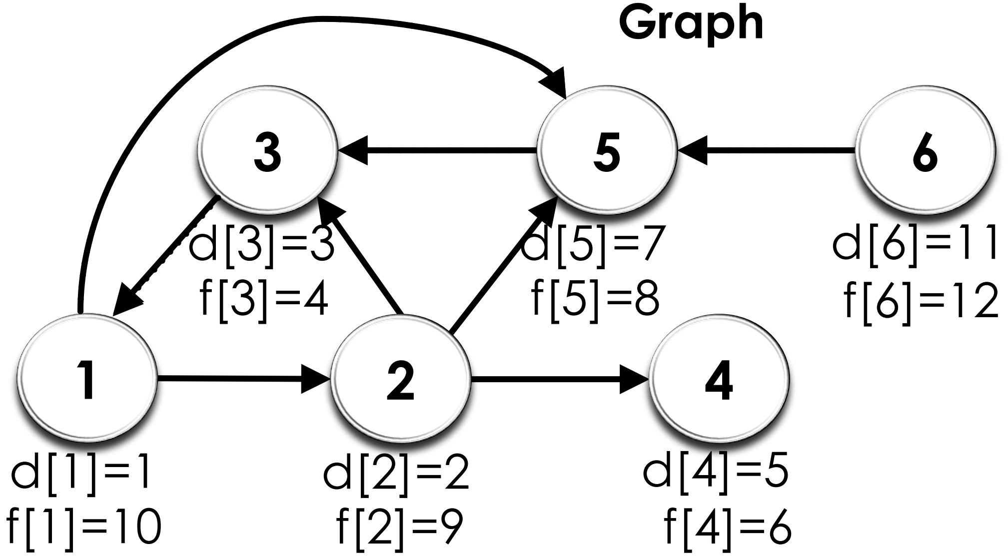
Each call iterates over the neighbours. Effectively: "for each node, for each neighbour, do O(1) work + recurse."

Total O(n+m) iterations over all recursive calls. **Total O(n+m) runtime!**

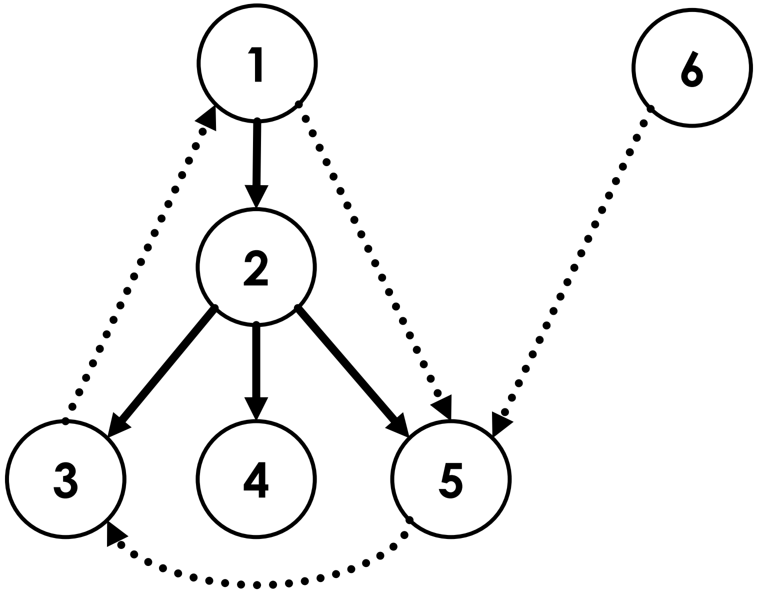
CLASSIFYING EDGE  IN DFS

- **If $\boldsymbol{pred}[v] = \boldsymbol{u}$**, then: (u, v) is a **tree edge**
- **Else** if **$\boldsymbol{v}$ is a descendent of $\boldsymbol{u}$** in the DFS forest: **forward edge**
- **Else** if **$\boldsymbol{v}$ is an ancestor of $\boldsymbol{u}$** in the DFS forest: **back edge**
- **Else**: (u, v) is a **cross edge**

Graph



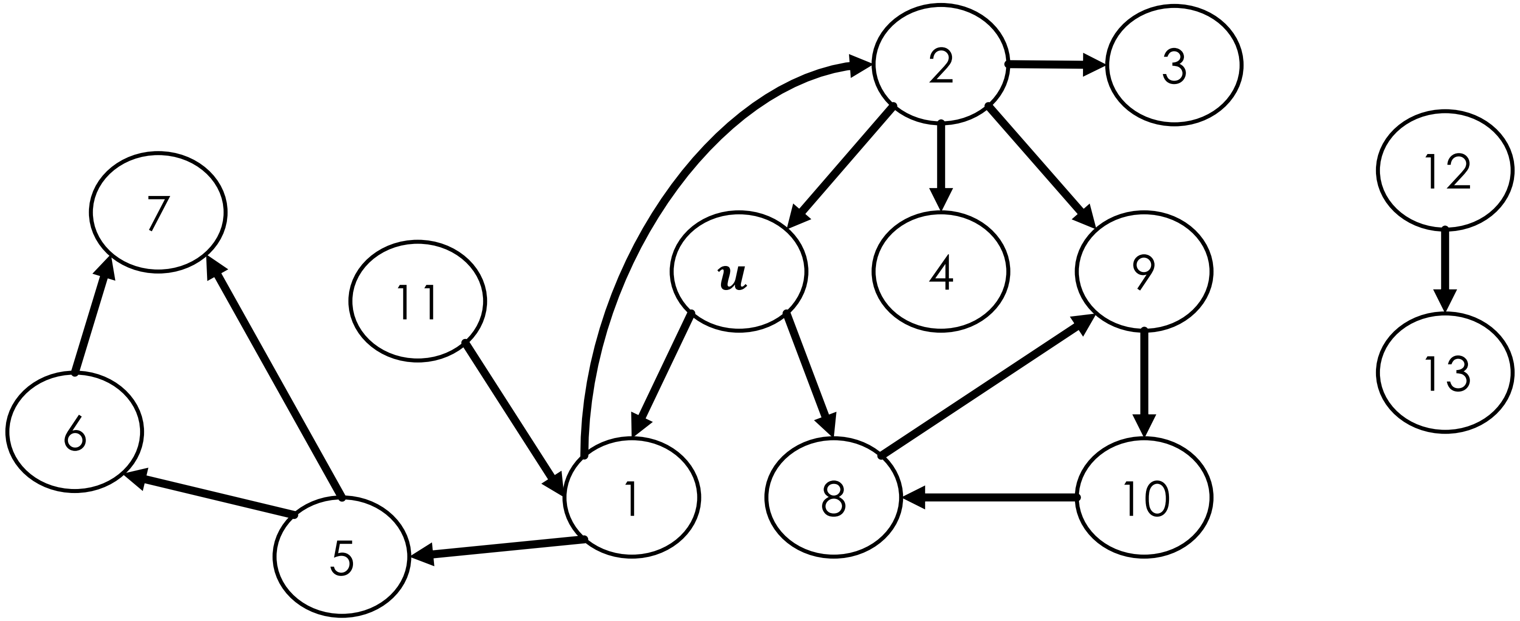
DFS forest



Can we classify edges **without** inspecting the DFS forest?
Perhaps using $\boldsymbol{d}[\ldots]$, $\boldsymbol{f}[\ldots]$, $\boldsymbol{colour}[\ldots]$?

DEFINITIONS

- **Definition:** we use $\boldsymbol{I_u}$ to denote $(d[u],f[u])$, which we call the **interval of $\boldsymbol{u}$**
- **Definition: $\boldsymbol{v}$ is white-reachable from $\boldsymbol{u}$** if there is a path from u to v containing **only white nodes** (excluding u)



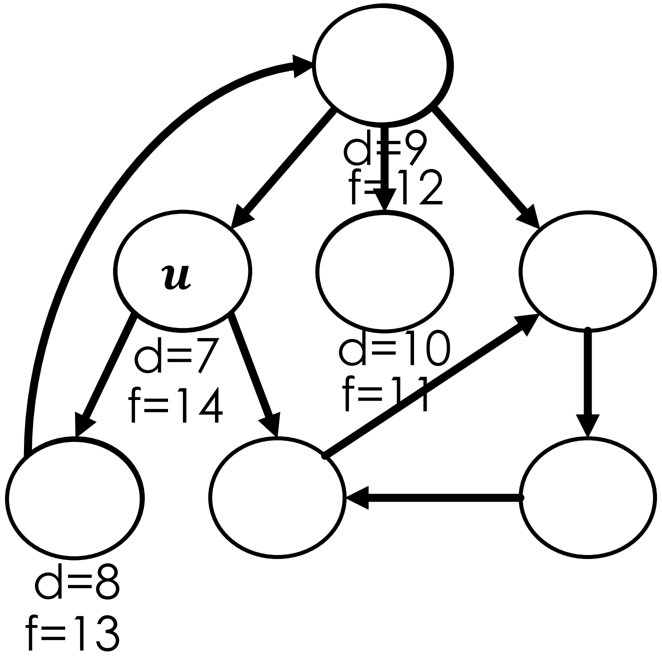
EXPLORING D[], F[] AND COLOUR[]

- **Observe: every** node v that is **white-reachable** from u when we first call $DFSVisit(u)$ becomes **gray after u** and **black before u** (so I_v is **nested inside** I_u)

Start $DFSVisit(u)$, colour u grey, and set u's discovery time

Perform $DFSVisit$ calls recursively…

Colour u black, set u's finish time and return from $DFSVisit(u)$



Consider the **tree of recursive calls** rooted at $DFSVisit(u)$.
v is discovered by a call in this tree **iff I_v is nested inside I_u**
iff v is a descendent of u in the DFS forest
iff v turns grey after u and black before u
iff v is white-reachable from u when $DFSVisit(u)$ is called

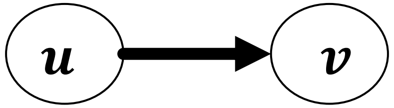
SUMMARIZING IN A THEOREM

- **Theorem:** Let $\boldsymbol{u}, \boldsymbol{v}$ be any nodes.
  The following statements are all **equivalent**.
  - (v is **white-reachable** from u when we call $DFSVisit(u)$)
  - (v turns grey after u and black before u)
  - (discovery/finish time interval I_v is **nested inside** I_u)
  - (v is discovered during $DFSVisit(u)$)
  - (v is a **descendant of** $\boldsymbol{u}$ in the DFS forest)

CLASSIFYING EDGE TYPES IN DFS

DFS inspects **every edge** in the graph.

When DFS inspects an edge $\{\boldsymbol{u}, \boldsymbol{v}\}$, the colour of v and relationship between the intervals of u and v determine the **edge type.**



edge type	colour of v	discovery/finish times
tree	Q1?	Q2?
forward	Q4?	Q3?
back	Q6?	Q5?
cross	Q8?	Q7?

v discovered **during $\boldsymbol{DFSVisit(u)}$**

but **not directly** from u (or $\{u, v\}$ would be a tree edge)

So when DFSVisit(u) inspects $\{u, v\}$, v **cannot** be white

v is already discovered!

v is a **child** of u in the DFS tree

v is a **descendent** of u

v is an **ancestor** of u

v is **not** a descendent, and **not** an ancestor

Recall: (v is discovered during $DFSVisit(u)$)

⇔ (v is **white-reachable** from u when we call $DFSVisit(u)$)

⇔ (v is a **descendant of $\boldsymbol{u}$** in the DFS forest)

⇔ (v turns grey after u and black before u)

⇔ (I_v nested inside I_u)

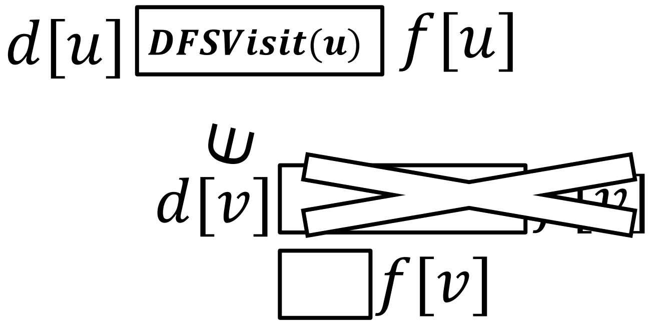
… by another recursive call that $DFSVisit(u)$ makes when it inspects a **previous edge**

That call **terminates** before $DFSVisit(u)$ inspects $\{u, v\}$

And it colors v **black!**

USEFUL FACT: PARENTHESIS THEOREM

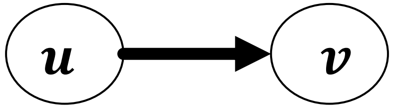
- **Theorem:** for each pair of nodes u, v the intervals of u and v are either **disjoint** or **nested**
- **Proof:** Suppose the intervals are **not disjoint**.
  - Then either $d[v] \in I_u$ or $d[u] \in I_v$
  - WLOG suppose $d[v] \in I_u$
  - Then v is discovered during $DFSVisit(u)$
  - So, v must turn gray after u and black before u
  - So $f[v] < f[u]$
  - So **the intervals are nested**. QED

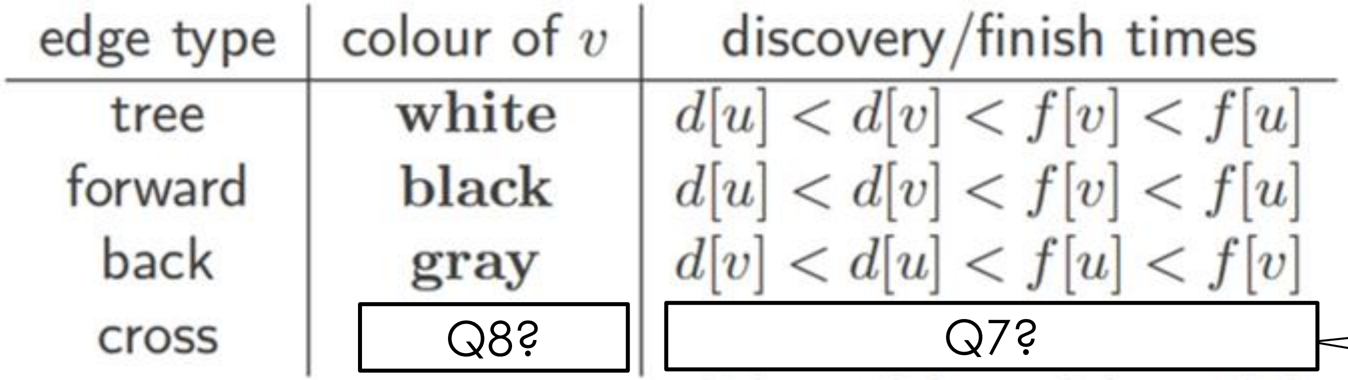


CLASSIFYING EDGE TYPES IN DFS

DFS inspects **every edge** in the graph.

When DFS inspects an edge $\{\boldsymbol{u}, \boldsymbol{v}\}$, the colour of v and relationship between the intervals of u and v determine the **edge type.**





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	Q8?	Q7?

v is **not** a descendent, and **not** an ancestor

Intervals $\boldsymbol{I_u}$ and $\boldsymbol{I_v}$ must be **disjoint**. But which is **earlier**?

If I_u were earlier, then v would be **discovered before $\boldsymbol{u}$ finishes** (because of edge $\{u, v\}$), so intervals would not be disjoint!

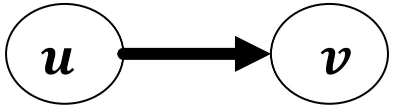
So, I_v must be earlier.

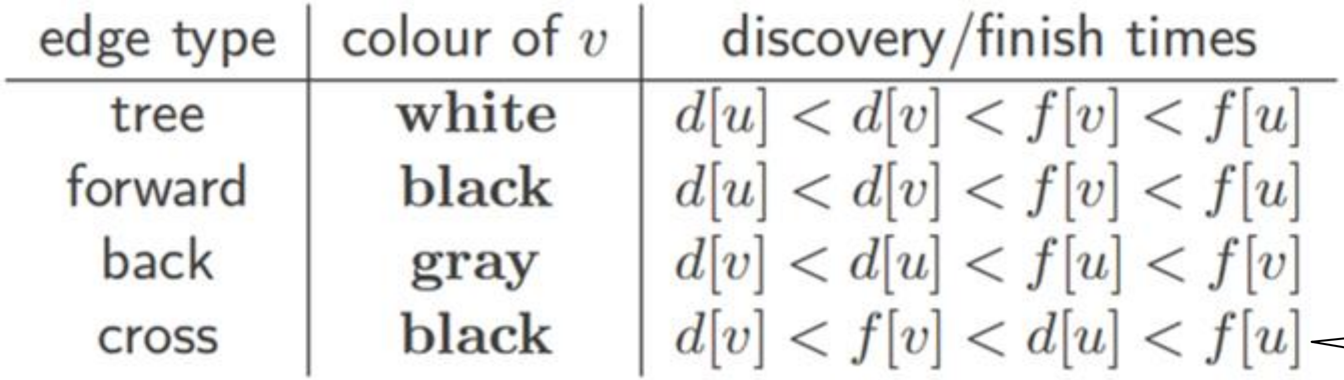
Recall: (v is discovered during $DFSVisit(u)$)
⇔ (v is **white-reachable** from u when we call $DFSVisit(u)$)
⇔ (v is a **descendant of $\boldsymbol{u}$** in the DFS forest)
⇔ (v turns grey after u and black before u)
⇔ (I_v nested inside I_u)

CLASSIFYING EDGE TYPES IN DFS

DFS inspects **every edge** in the graph.

When DFS inspects an edge $\{\boldsymbol{u}, \boldsymbol{v}\}$, the colour of v and relationship between the intervals of u and v determine the **edge type.**





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

v is **not** a descendent, and **not** an ancestor

Intervals $\boldsymbol{I_u}$ and $\boldsymbol{I_v}$ must be **disjoint**. But which is **earlier**?

If I_u were earlier, then v would be **discovered before $\boldsymbol{u}$ finishes** (because of edge $\{u, v\}$), so intervals would not be disjoint!

So, I_v must be earlier.

Recall: (v is discovered during $DFSVisit(u)$)

$\Leftrightarrow$ (v is **white-reachable** from u when we call $DFSVisit(u)$)

$\Leftrightarrow$ (v is a **descendant of $\boldsymbol{u}$** in the DFS forest)

$\Leftrightarrow$ (v turns grey after u and black before u)

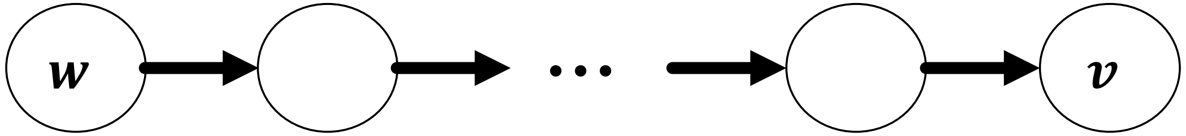
$\Leftrightarrow$ (I_v nested inside I_u)

APPLICATION OF DFS (OR BFS):
STRONG CONNECTEDNESS

Testing existence of all-to-all paths

STRONG CONNECTEDNESS

- In a directed graph,
  - v **is reachable from** w if there is a **path** from w to v



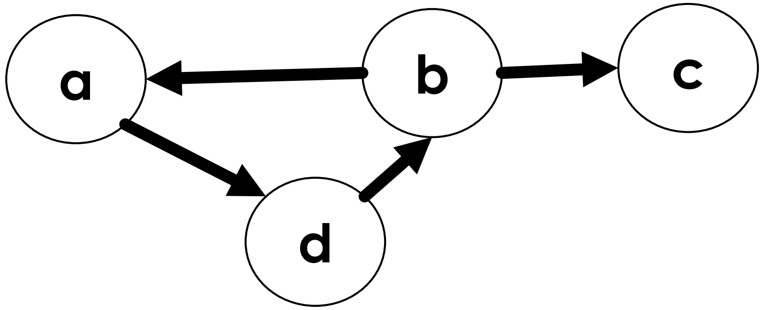
  - we denote such a path $w \rightsquigarrow v$

Compare: we use $w \rightarrow v$ to denote an **edge** from w to v

- A graph G is **strongly connected** iff every node is **reachable** from every other node
  - More formally: $\forall_{w,v} \ \exists \ w \rightsquigarrow v$

STRONG CONNECTEDNESS

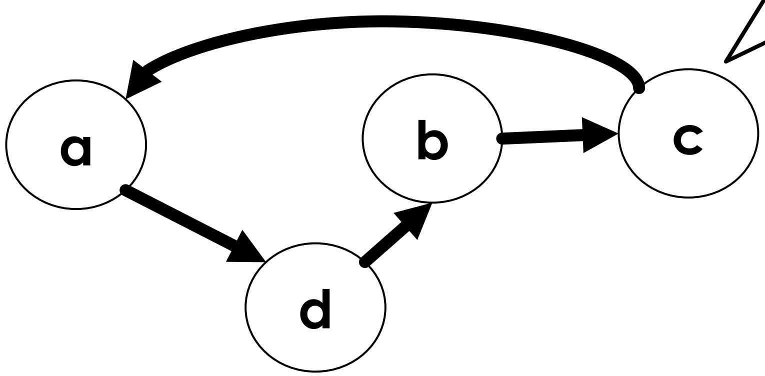
- Is this graph **strongly connected**?



No path from c to other nodes.

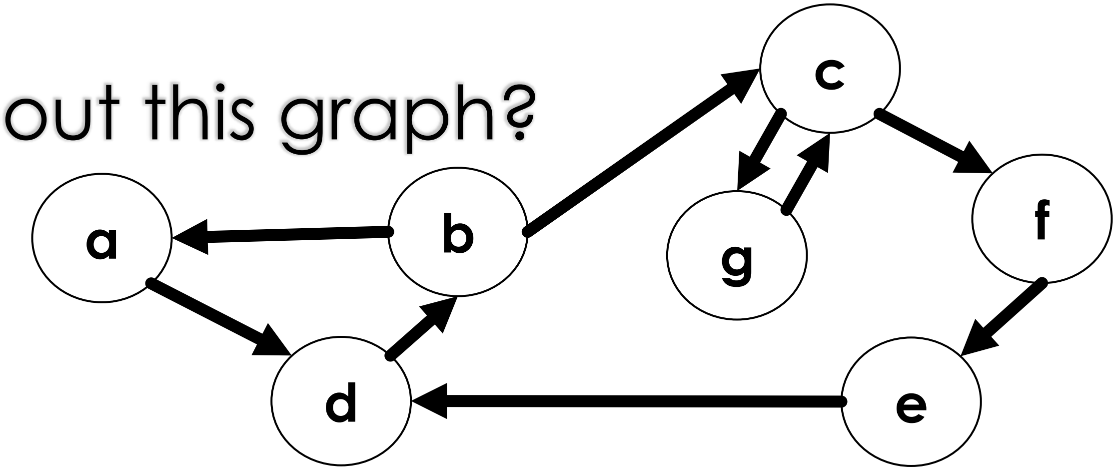
- How about this one?

Yes. One big cycle.



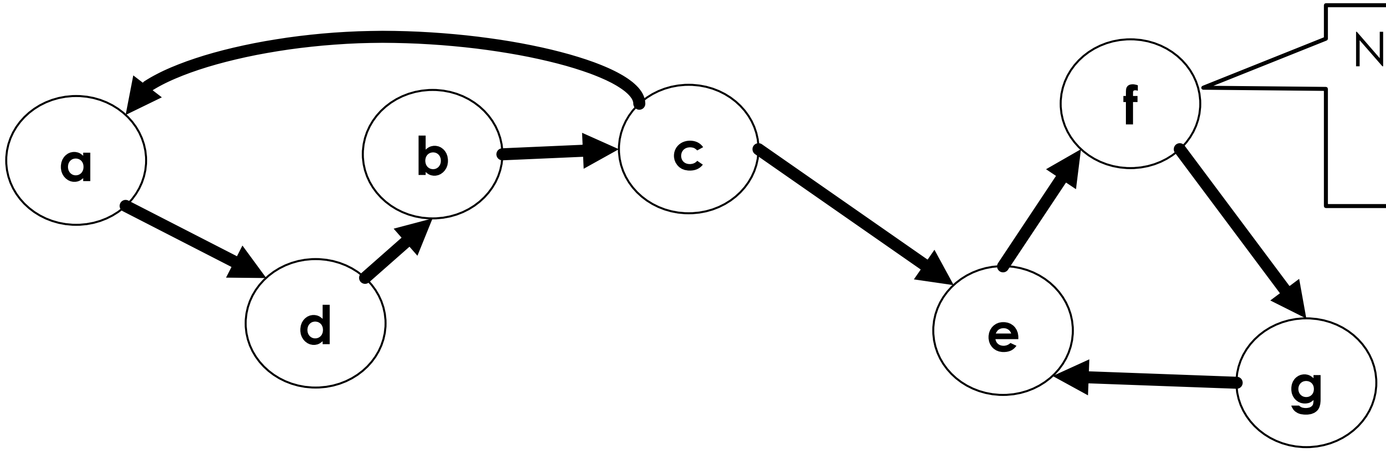
STRONG CONNECTEDNESS

- How about this graph?



Yes. Multiple **intersecting** cycles.

- How about this one?



No. Two cycles with only a one-directional path between them.

OTHER APPLICATIONS OF CHECKING STRONG CONNECTEDNESS

- You gain some **symmetry** from knowing a graph is strongly connected
- For example, you can **start a graph traversal at any node**, and know the traversal will reach **every** node
- Without strong connectedness, if you want to run a graph traversal that reaches every node in a single pass, you would have to do additional processing to determine an appropriate starting node

OTHER APPLICATIONS OF CHECKING STRONG CONNECTEDNESS

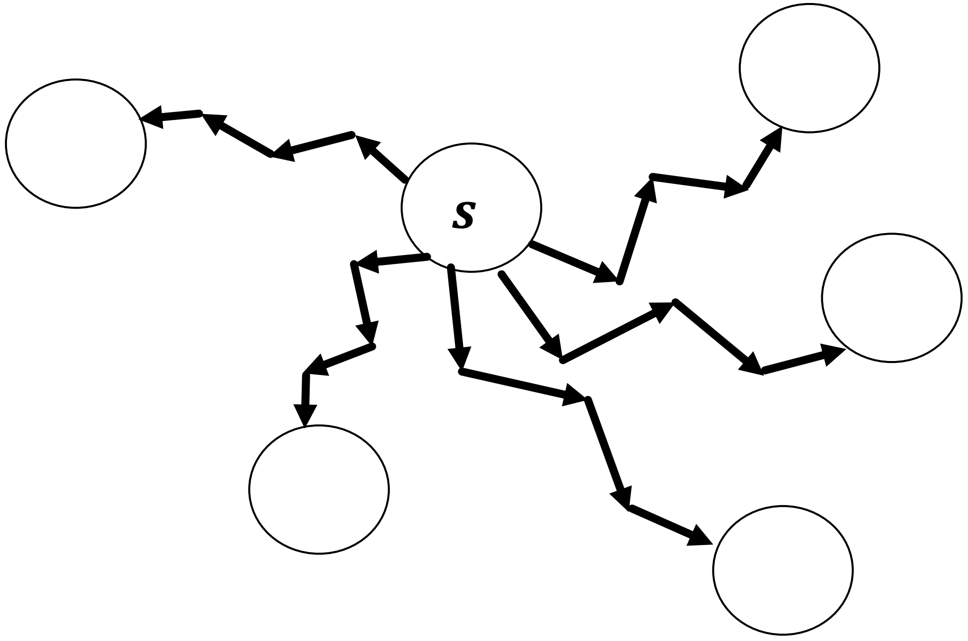
- Useful as a sanity check!
- Suppose you want to run an algorithm that **requires strong connectedness**, and you **believe** your input graph is strongly connected
- **Validate** your input by **testing** whether this is true!
- Subtle, difficult-to-detect bugs often result if such an algorithm is run only on one component of a graph
- [More concrete applications once we generalize and talk about strongly connected **components**…]

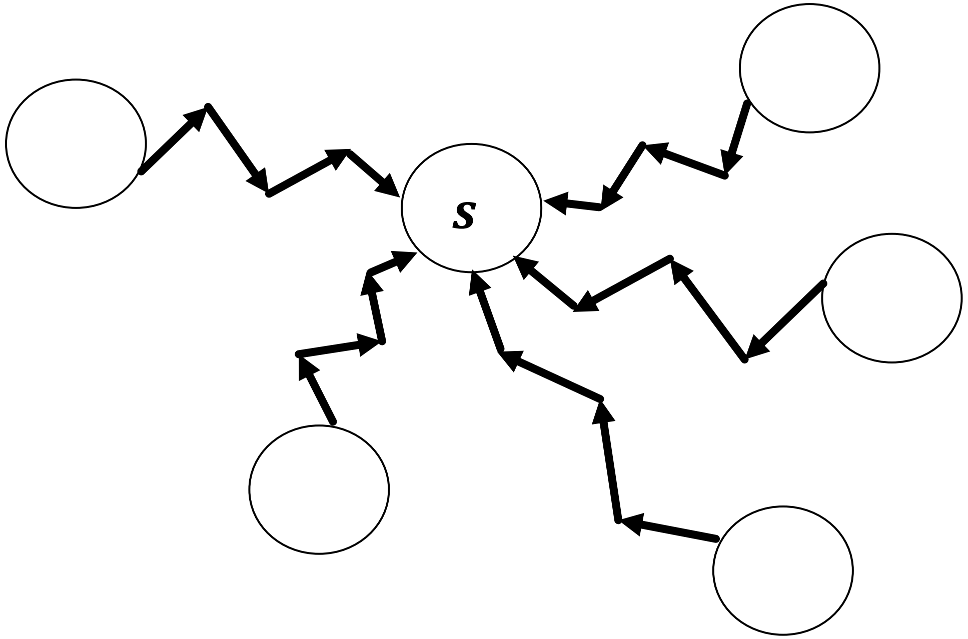
A USEFUL LEMMA

- Lemma: a graph is strongly connected
- **iff** for **any** node s,
- **all nodes are reachable** from s, and $\boldsymbol{s}$ **is reachable** from all nodes

Proof: (⇒) Suppose G is strongly connected. Then for all u, v we have $u \rightsquigarrow v$. Fix any s. Node s is reachable from all nodes, and vice versa.

(⇐) Suppose some s is reachable from all nodes and vice versa. For any u, v, we have $u \rightsquigarrow s \rightsquigarrow v$, and $v \rightsquigarrow s \rightsquigarrow u$. So G is strongly conn.





CREATING AN ALGORITHM

- How to use DFS to determine whether **every node is reachable** from a given node s?

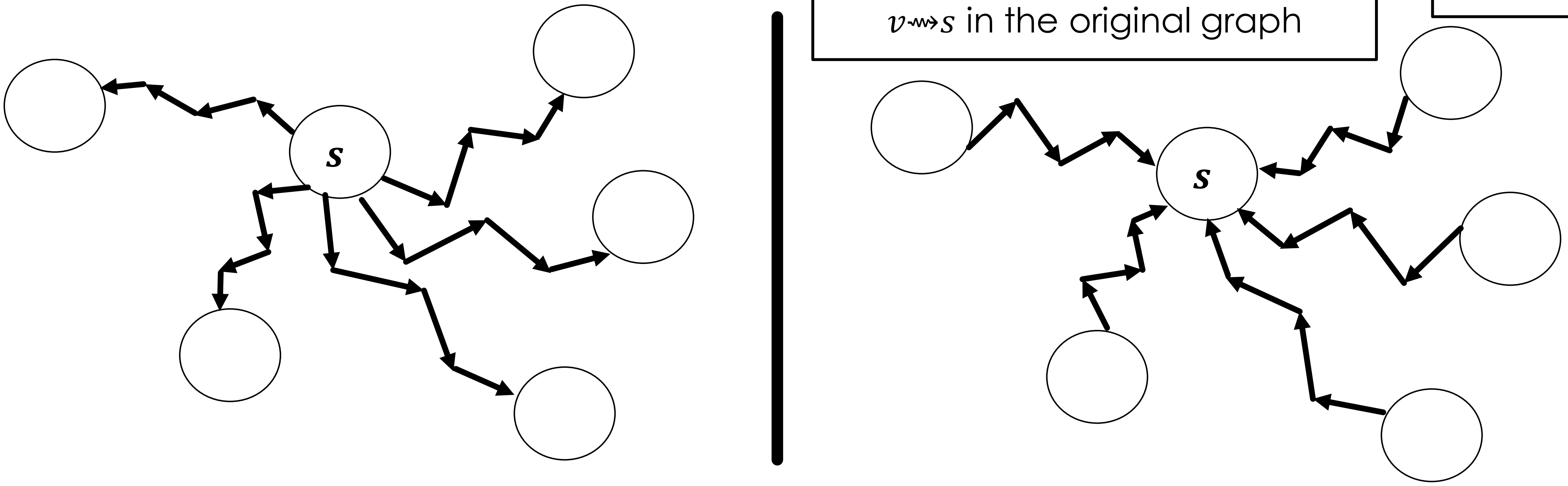
DFS from s and see if every node turns black

- How to use DFS to determine whether **s is reachable** from every node?

What if we first **reverse** the direction of every edge?

Then $s \rightsquigarrow v$ in this new graph IFF $v \rightsquigarrow s$ in the original graph

DFS from s

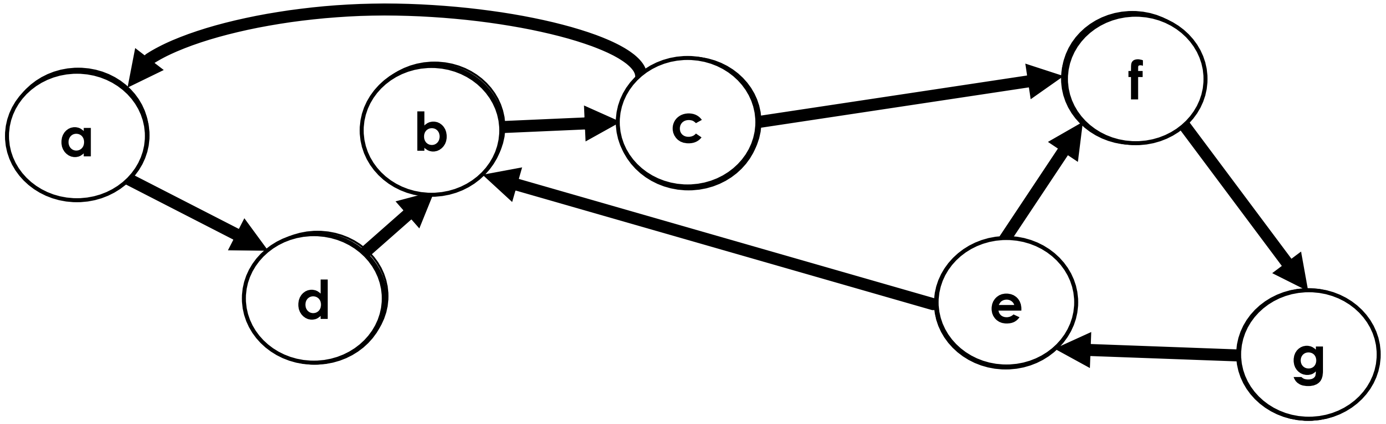


THE ALGORITHM

```
$IsStronglyConnected(G = \{V, E\})$ where $V = v_1, v_2, \dots, v_n$
    $(colour, d, f) \coloneqq DFSVisit(v_1, G)$
    for $i \coloneqq 1..n$
        if $colour[v_i] \neq black$ then return $false$
    Construct graph $H$ by **reversing** all edges in $G$
    $(colour, d, f) \coloneqq DFSVisit(v_1, \boldsymbol{H})$
    for $i \coloneqq 1..n$
        if $colour[v_i] \neq black$ then return $false$
    return $true$
```

How?

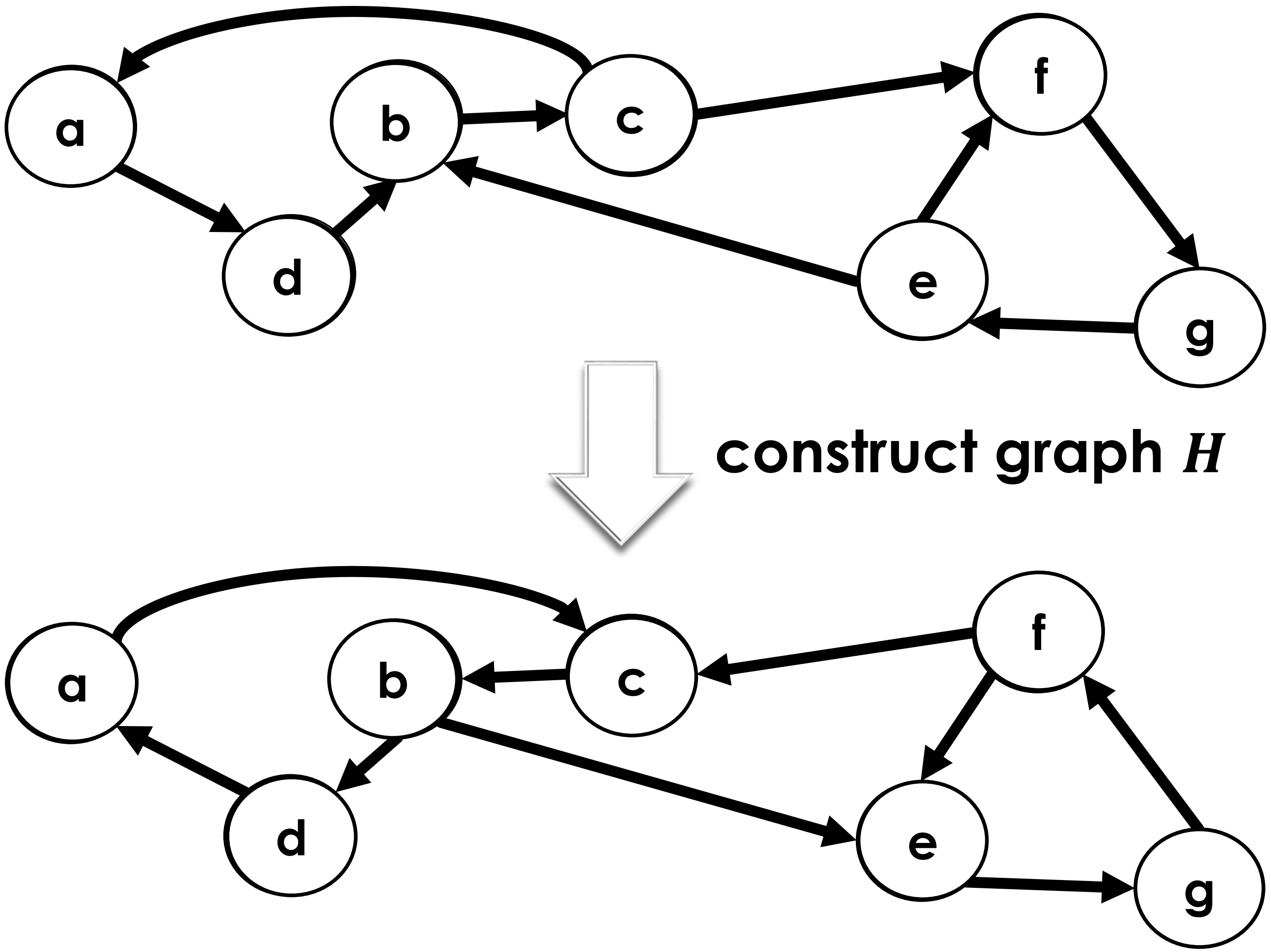
EXAMPLE EXECUTION 1



$DFSVisit(a)$ in G
(a is arbitrary)

Every node is
black. Next step!

EXAMPLE EXECUTION 1



construct graph H

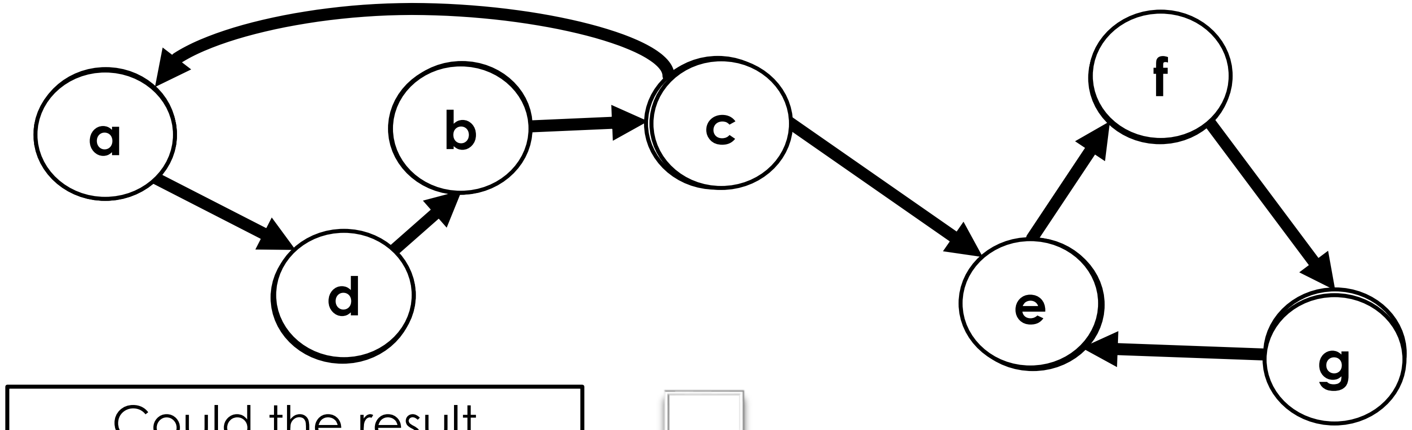
$DFSVisit(a)$ in G
(a is arbitrary)

Every node is black. Next step!

$DFSVisit(a)$ in H

Every node is black. So G is strongly connected!

EXAMPLE EXECUTION 2

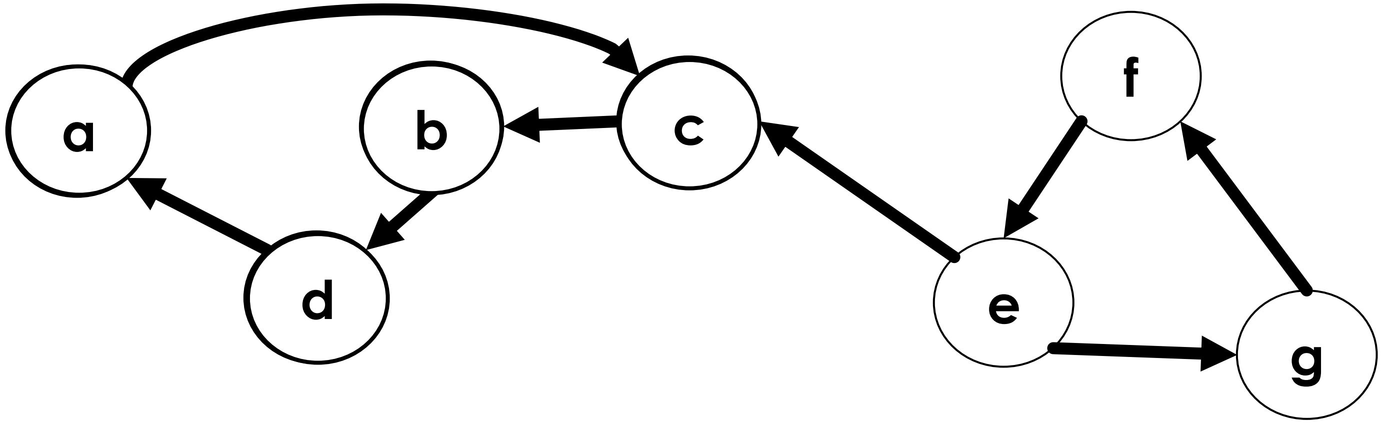


$DFSVisit(a)$ in G
(a is arbitrary)

Every node is black. Next step!

Could the result change if we **started at a different node**?

construct graph H



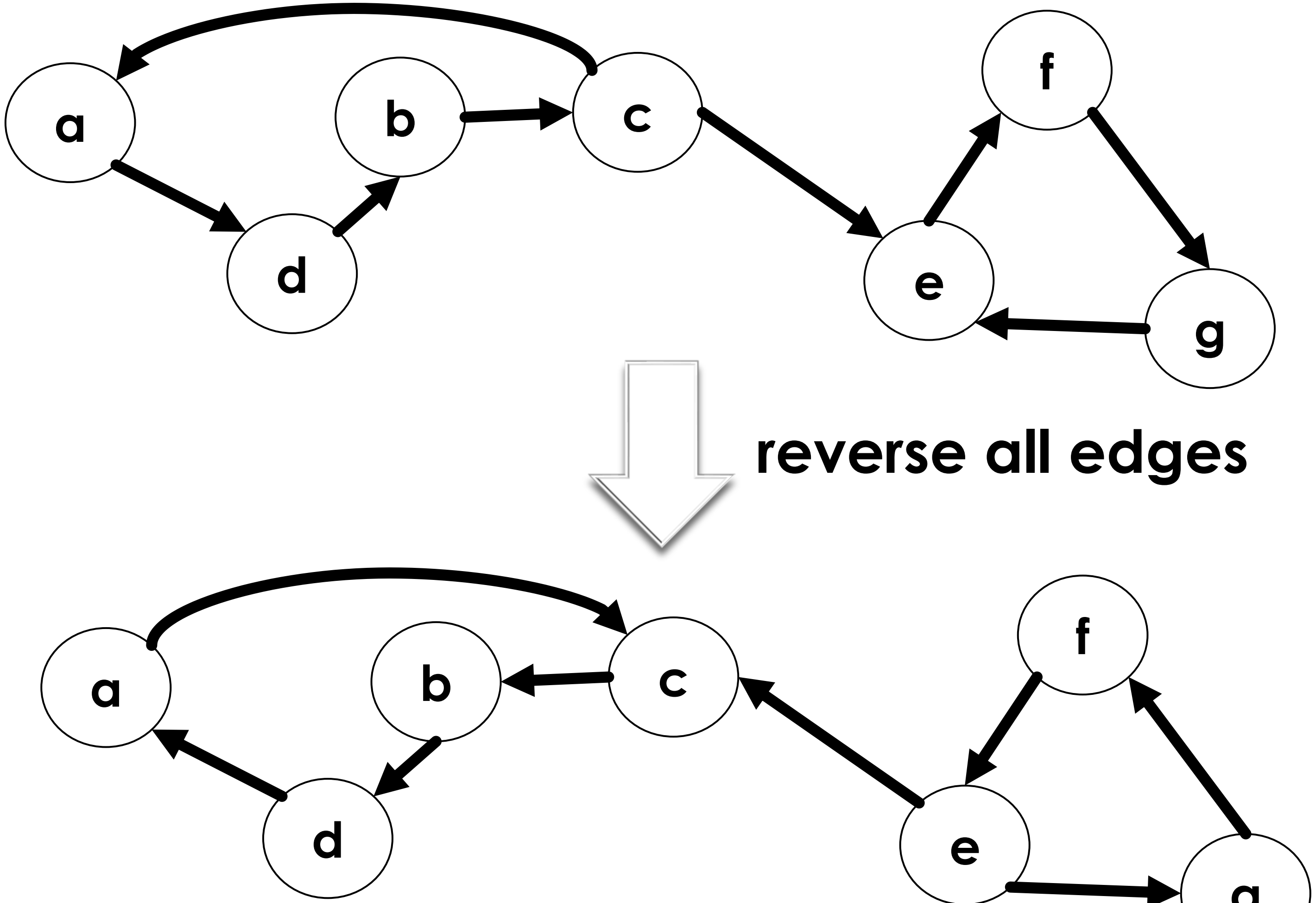
$DFSVisit(a)$ in H

Some nodes are **not black**

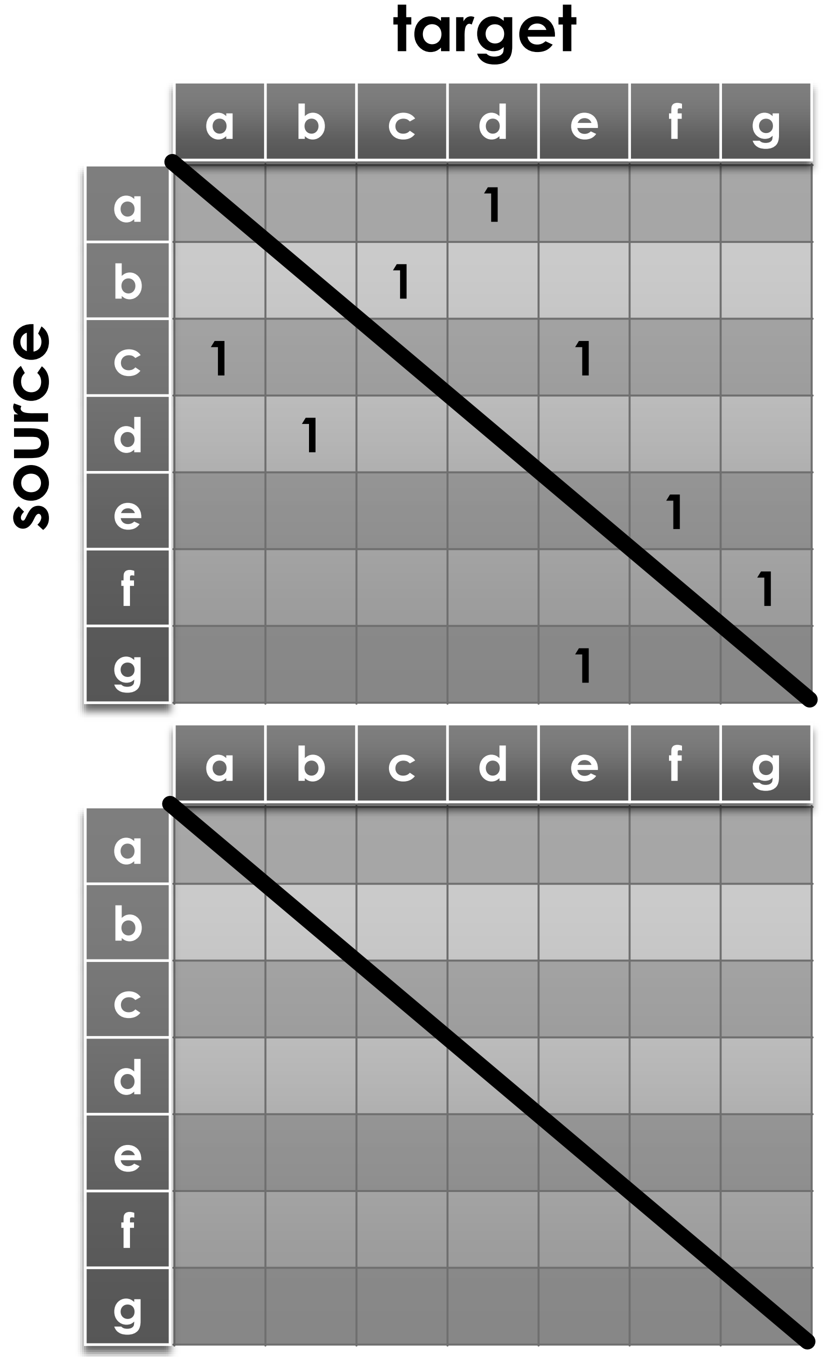
No path from those nodes to a

So G is **not strongly connected**!

REVERSING EDGES: ADJACENCY **MATRIX**



reverse all edges



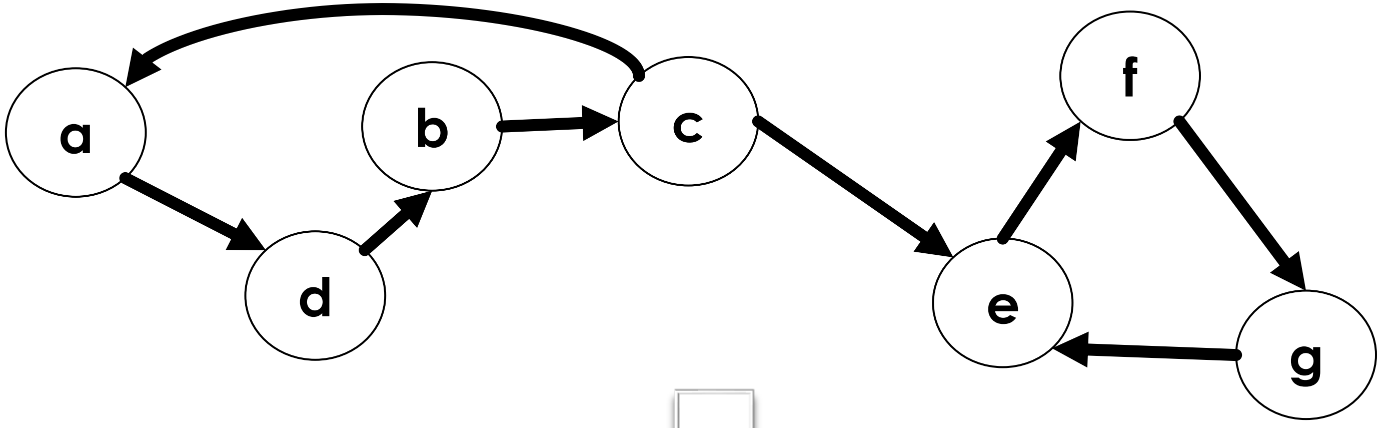
target

source

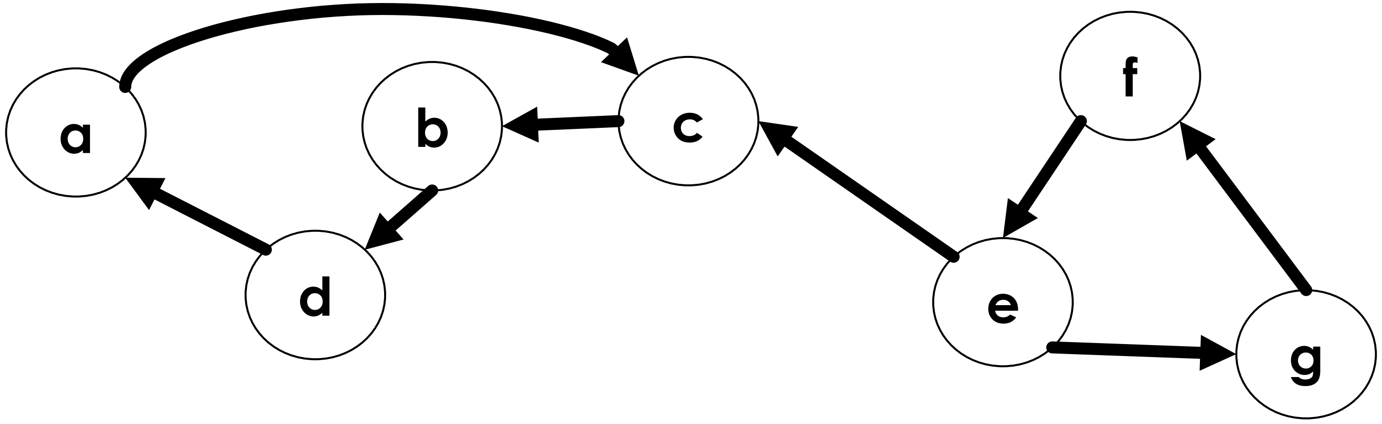
	a	b	c	d	e	f	g
a				1			
b			1				
c	1				1		
d		1					
e						1	
f							1
g					1		

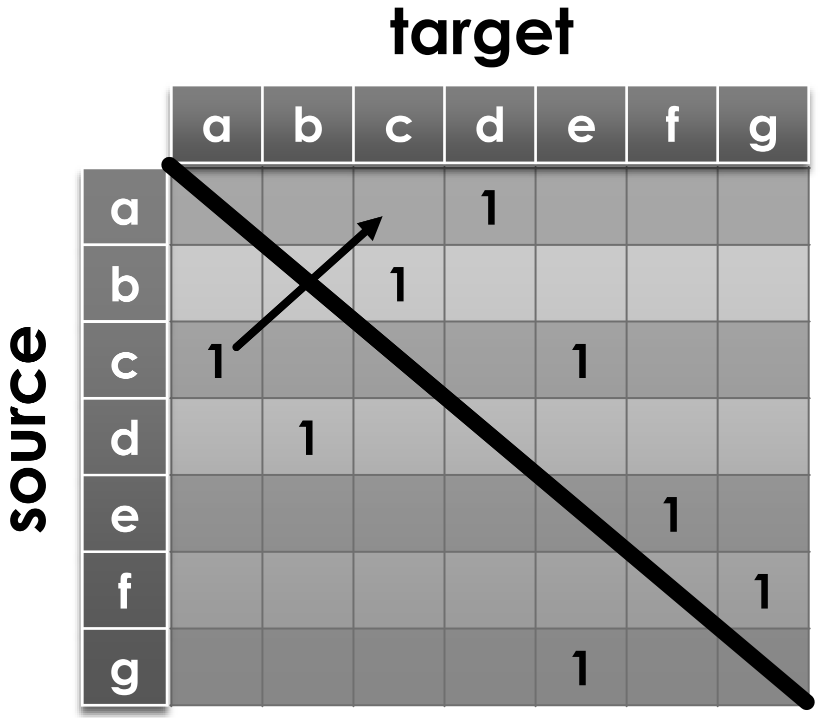
	a	b	c	d	e	f	g
a							
b							
c							
d							
e							
f							
g							

REVERSING EDGES: ADJACENCY **MATRIX**



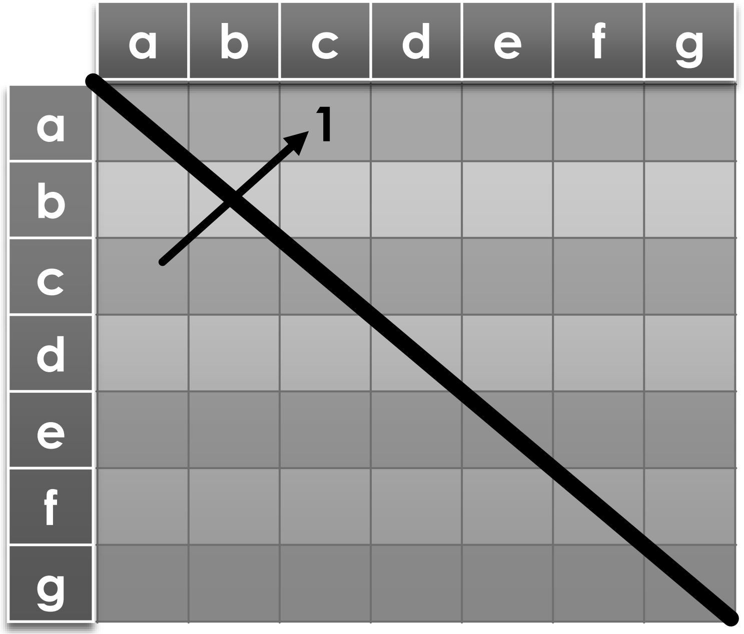
reverse all edges





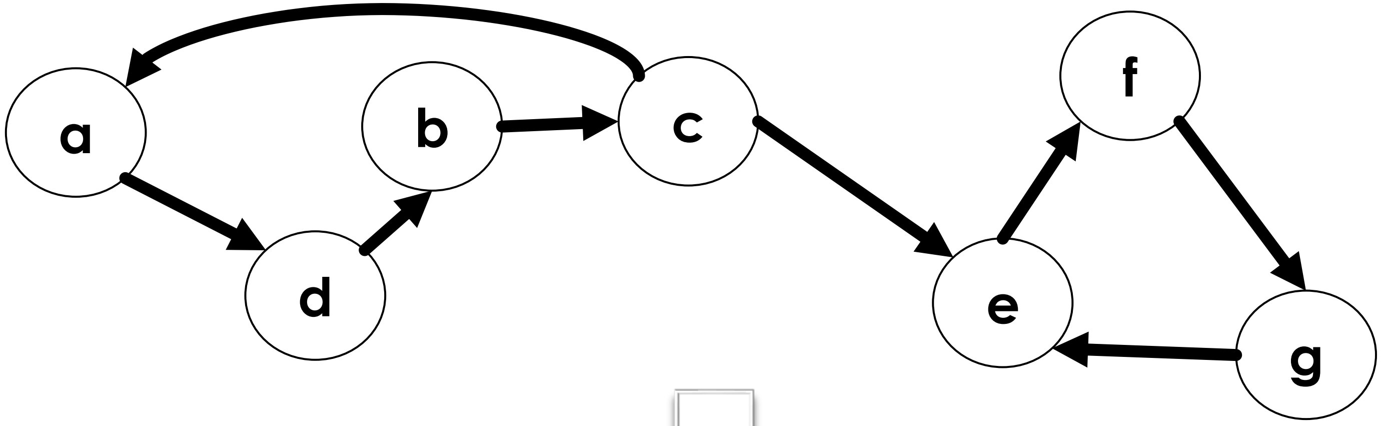
target

source	a	b	c	d	e	f	g
a				1			
b			1				
c	1				1		
d		1					
e						1	
f							1
g					1		

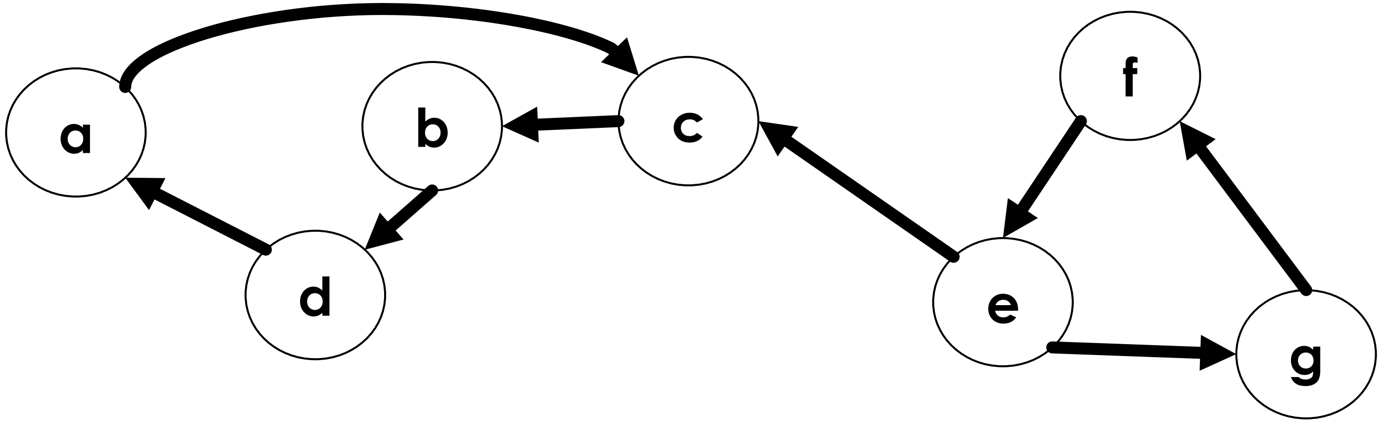


	a	b	c	d	e	f	g
a			1				
b							
c							
d							
e							
f							
g							

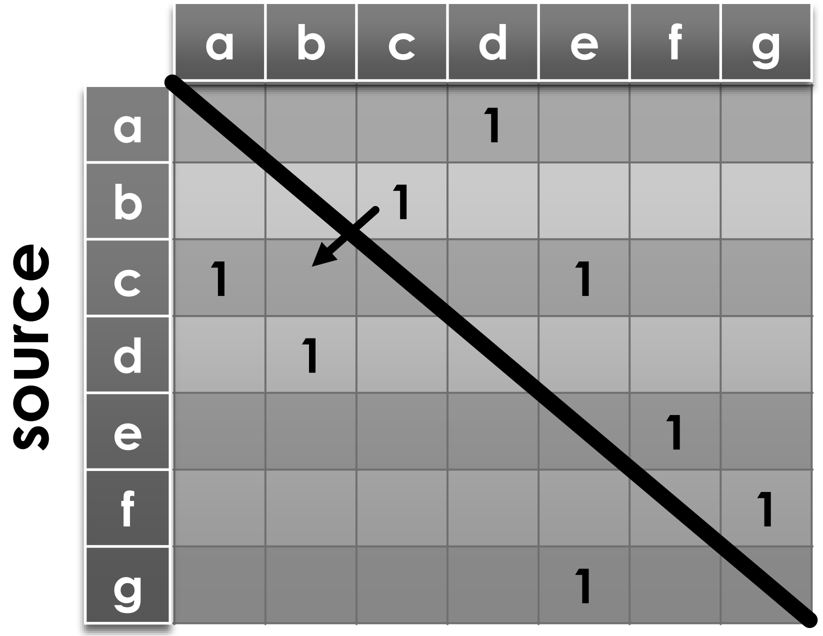
REVERSING EDGES: ADJACENCY **MATRIX**



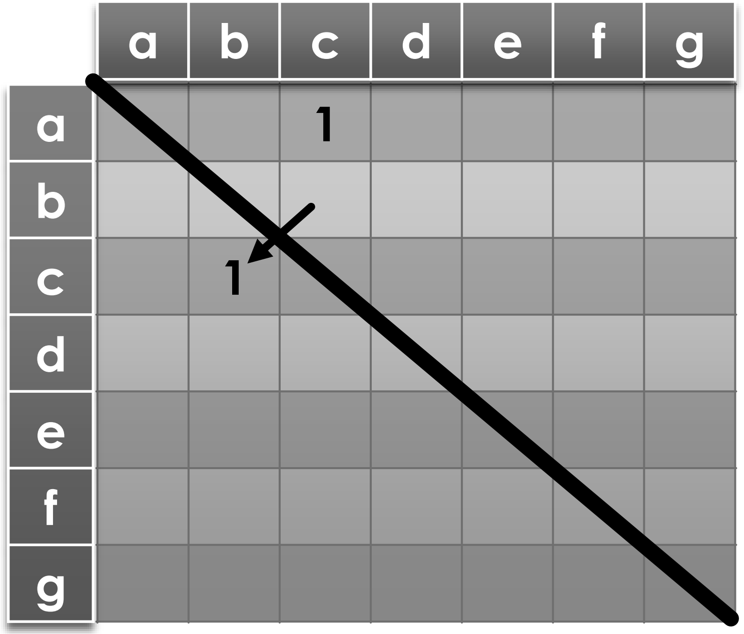
reverse all edges



target

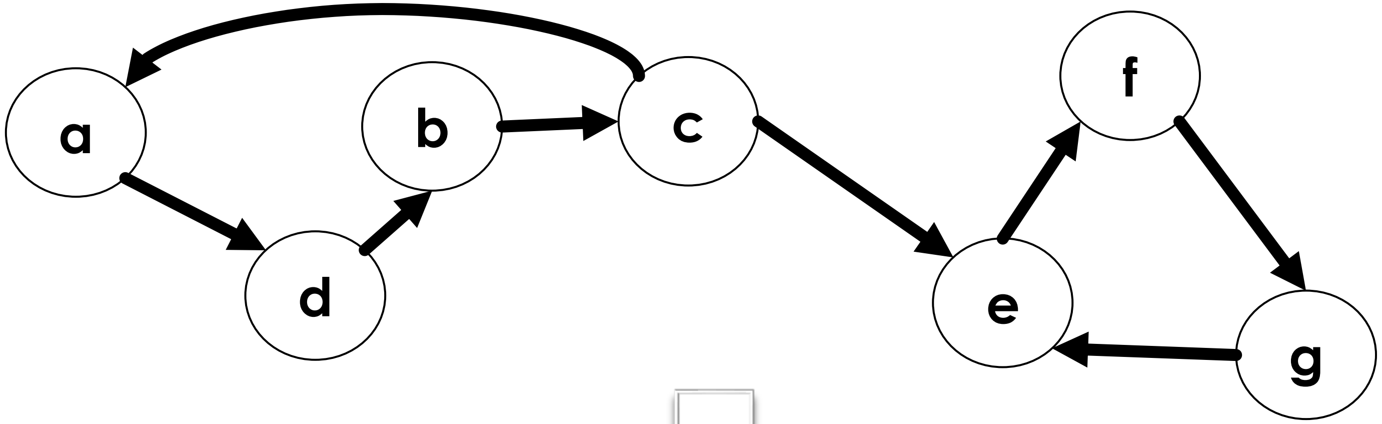


source	a	b	c	d	e	f	g
a				1			
b			1				
c	1				1		
d		1					
e						1	
f							1
g					1		

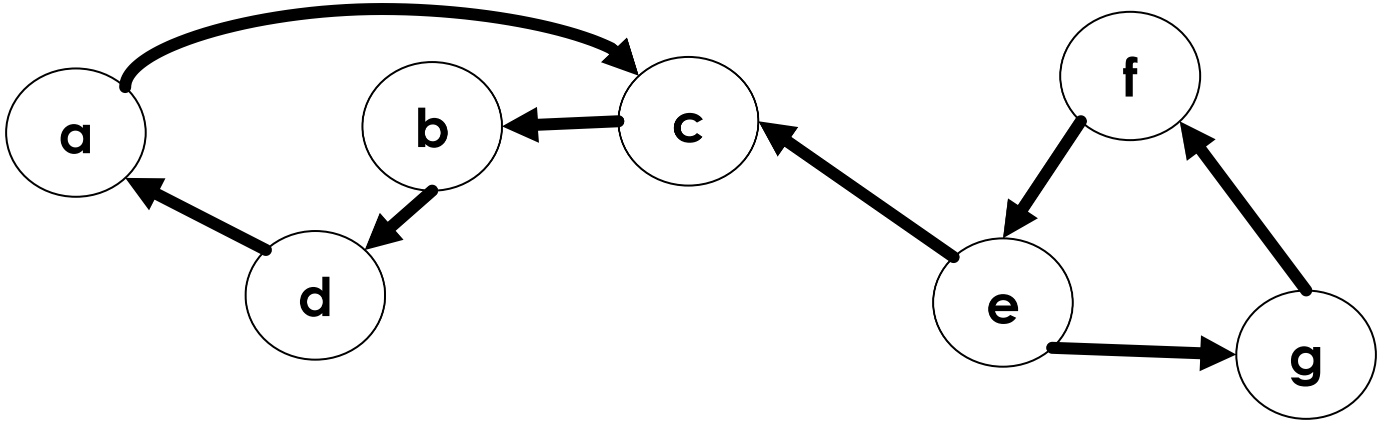


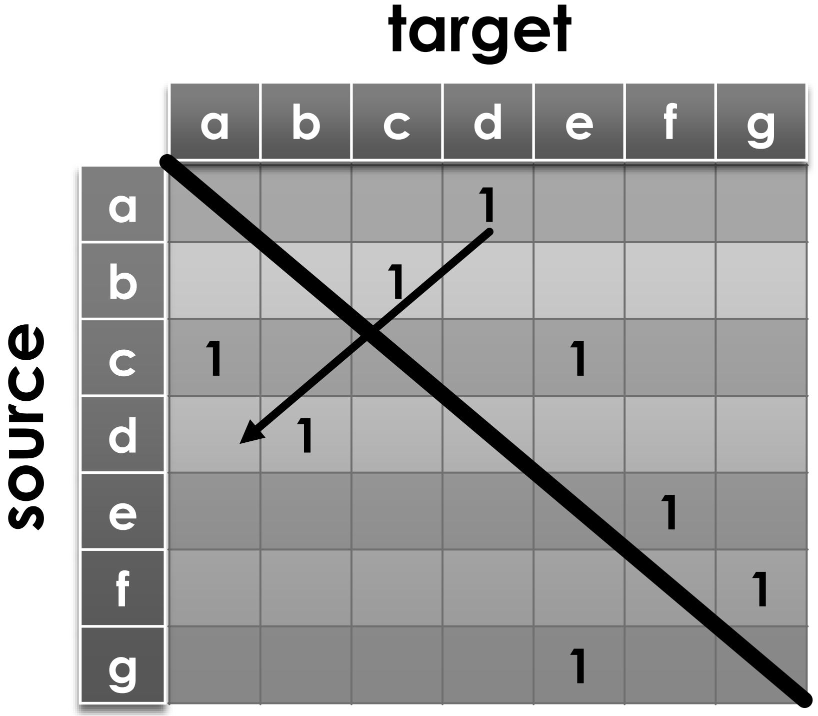
	a	b	c	d	e	f	g
a			1				
b							
c		1					
d							
e							
f							
g							

REVERSING EDGES: ADJACENCY **MATRIX**

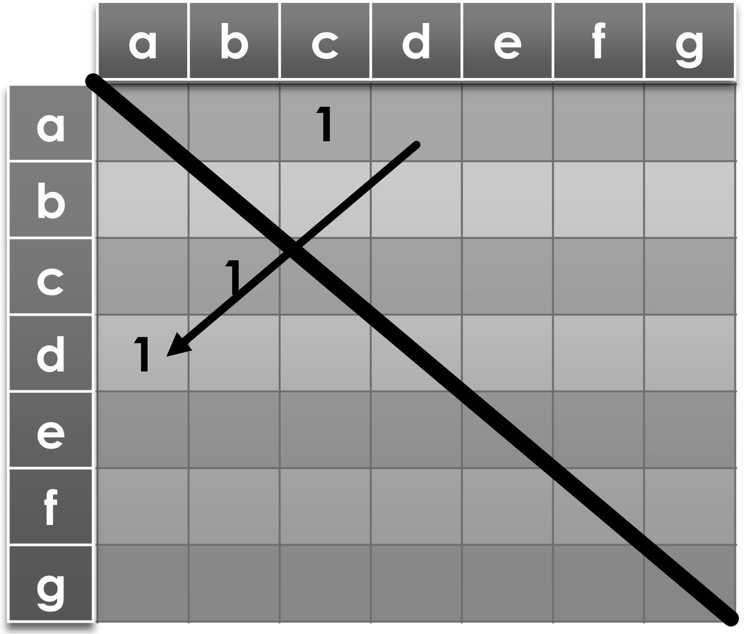


reverse all edges



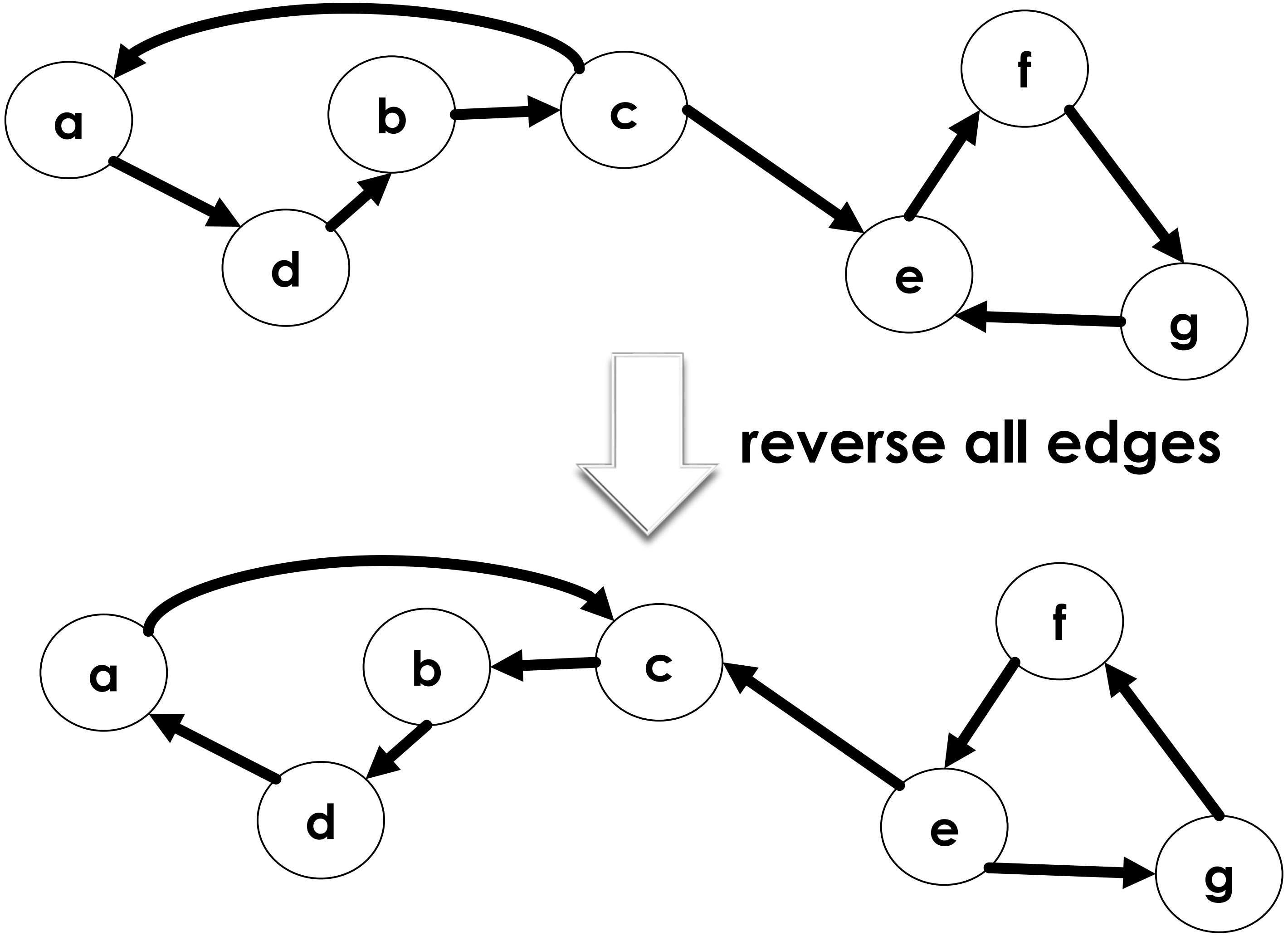


	a	b	c	d	e	f	g
a				1			
b			1				
c	1				1		
d		1					
e						1	
f							1
g					1		



	a	b	c	d	e	f	g
a			1				
b							
c		1					
d	1						
e							
f							
g							

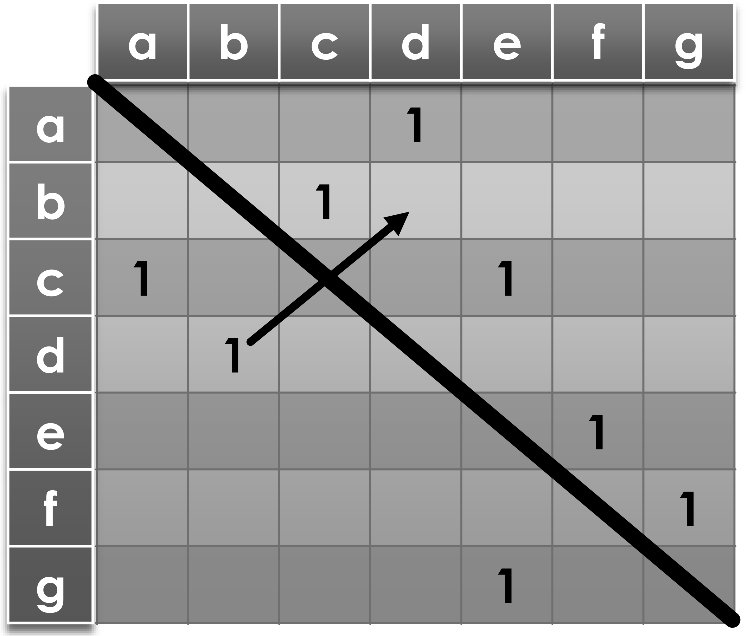
REVERSING EDGES: ADJACENCY **MATRIX**



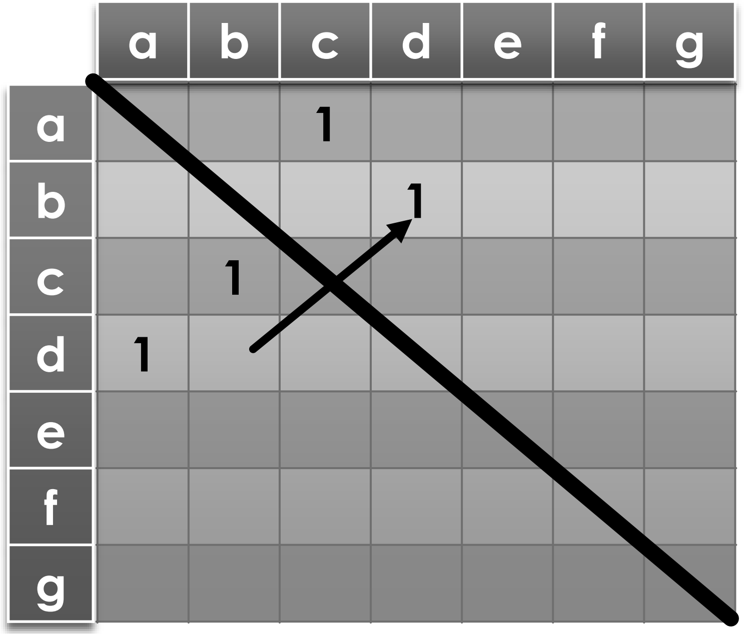
reverse all edges

target

source

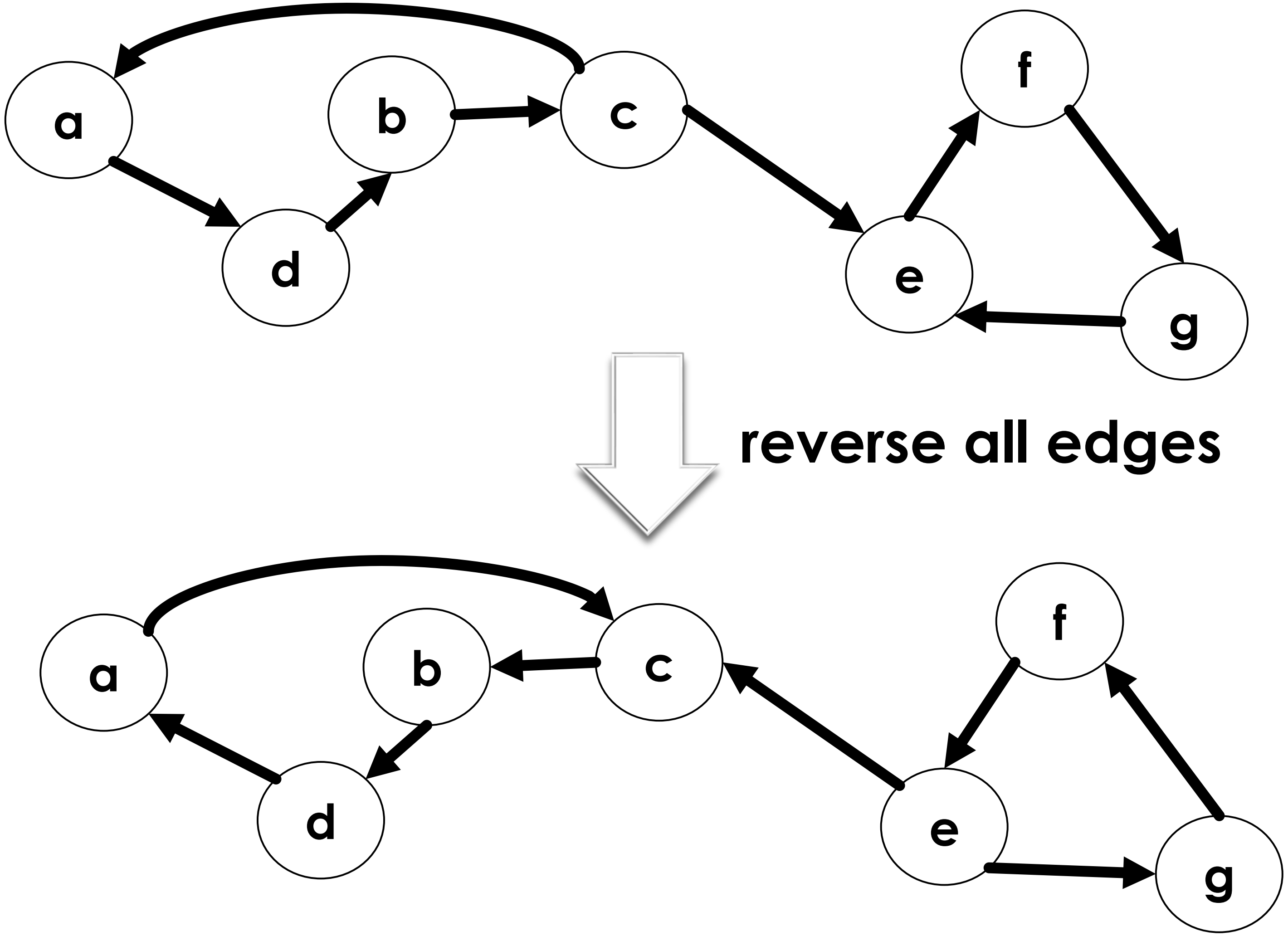


	a	b	c	d	e	f	g
a				1			
b			1				
c	1				1		
d		1					
e						1	
f							1
g					1		

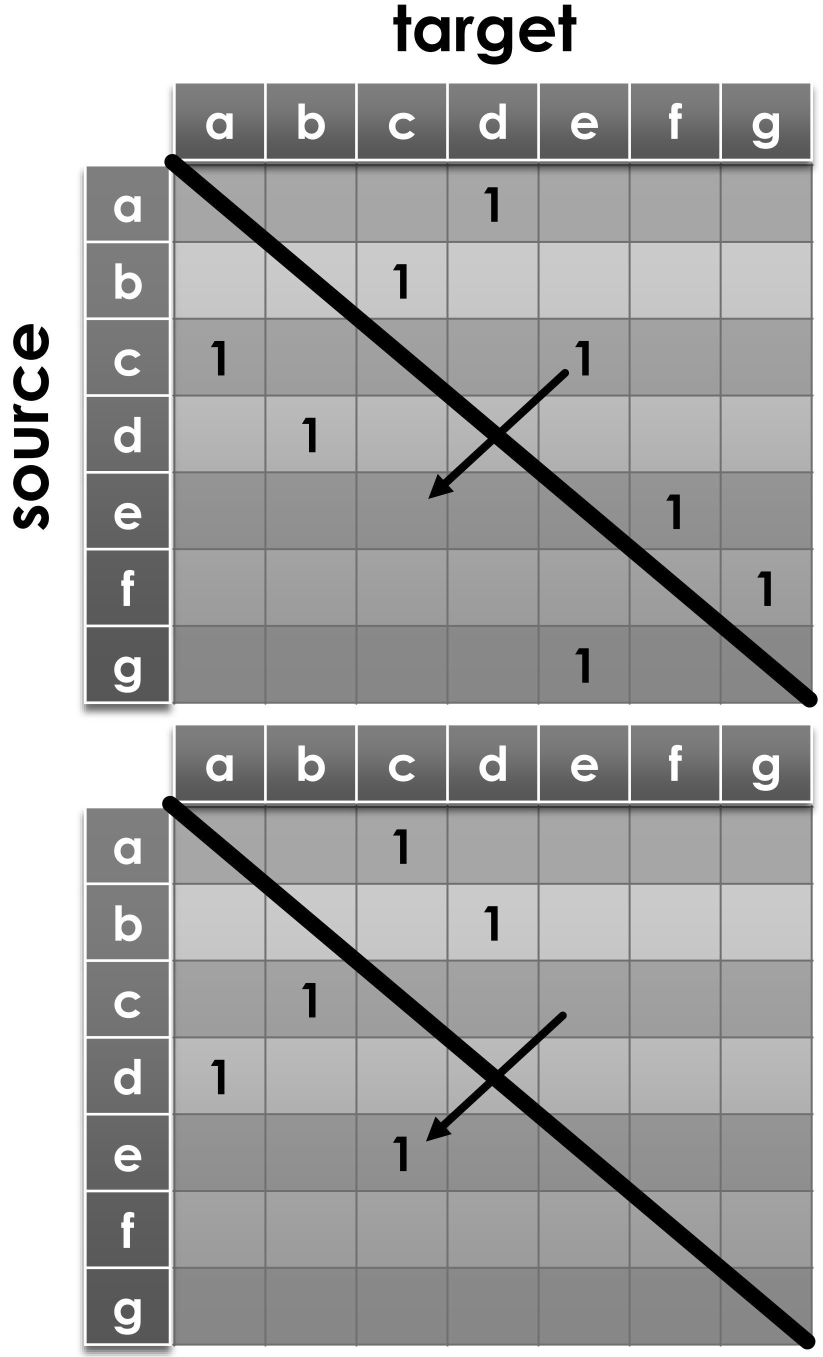


	a	b	c	d	e	f	g
a			1				
b				1			
c		1					
d	1						
e							
f							
g							

REVERSING EDGES: ADJACENCY **MATRIX**



reverse all edges



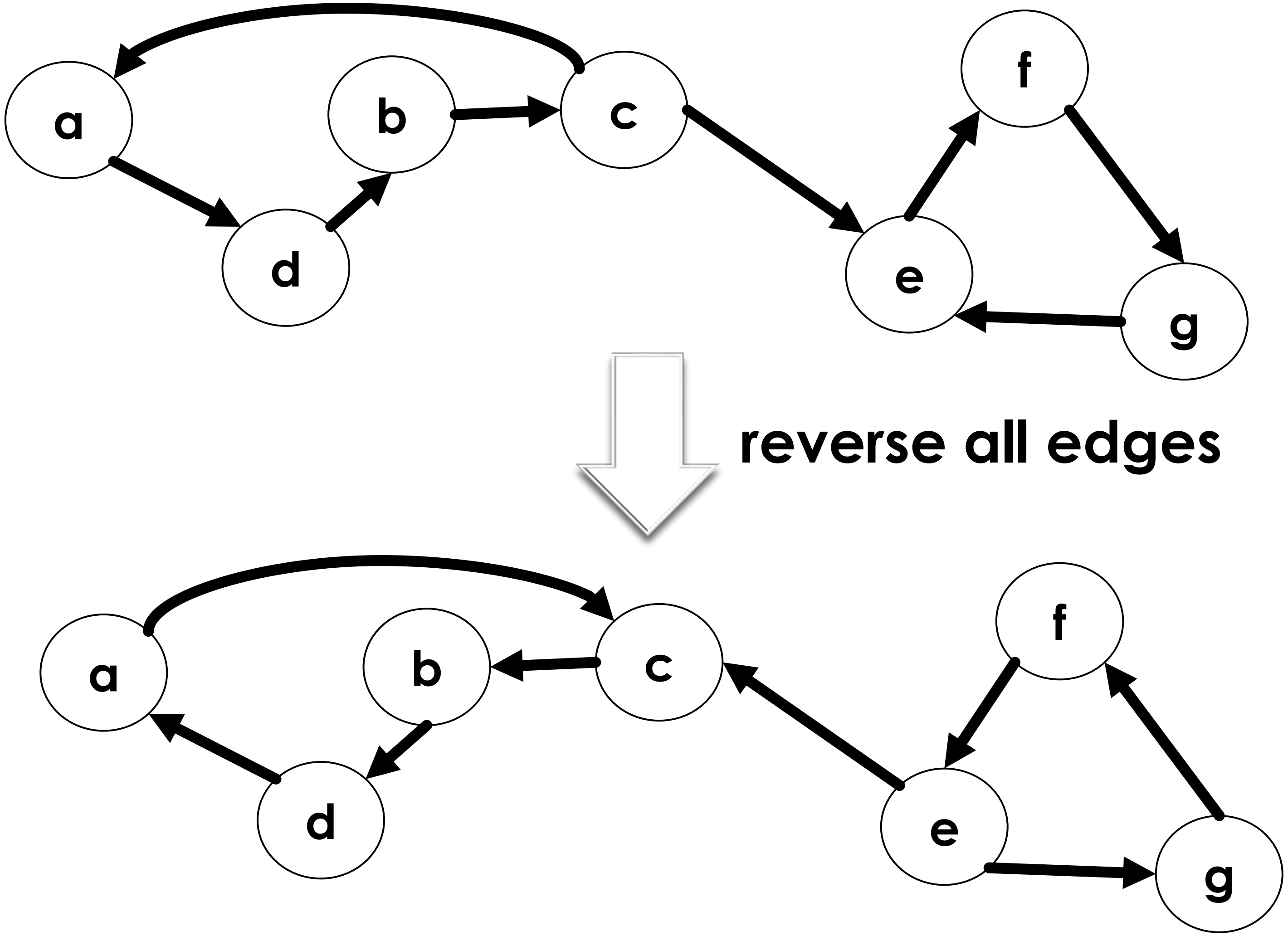
target

source

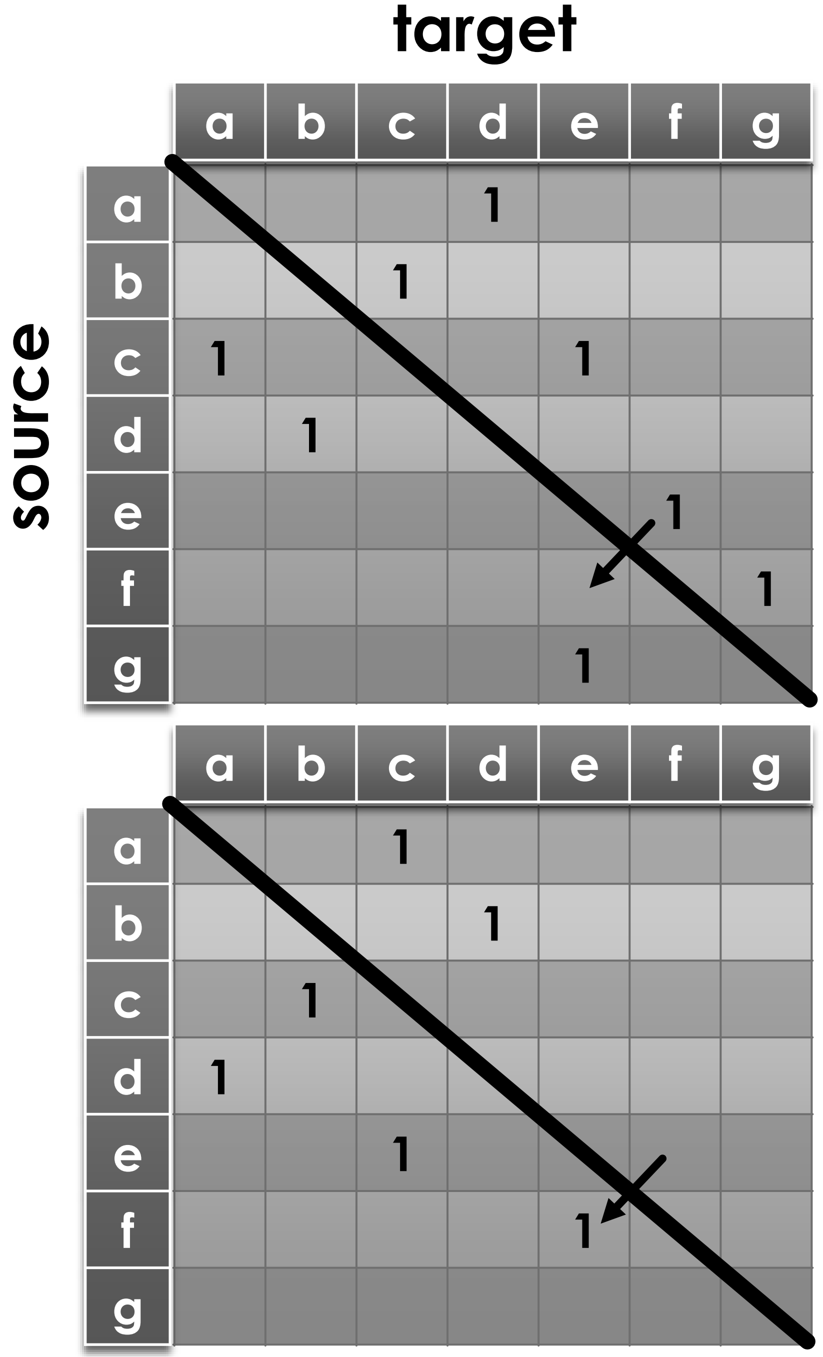
	a	b	c	d	e	f	g
a				1			
b			1				
c	1				1		
d		1					
e						1	
f							1
g					1		

	a	b	c	d	e	f	g
a			1				
b				1			
c		1					
d	1						
e			1				
f							
g							

REVERSING EDGES: ADJACENCY **MATRIX**



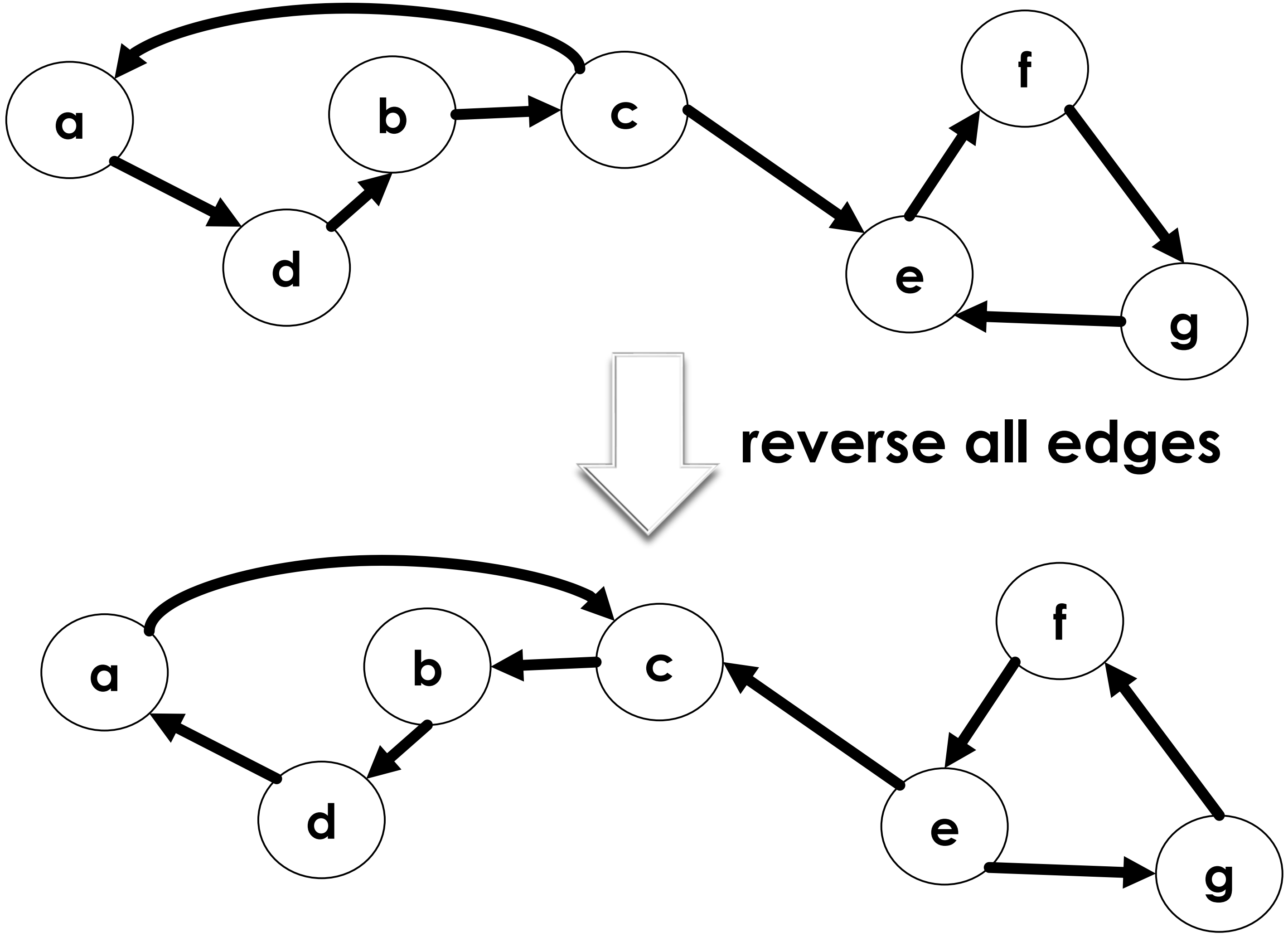
reverse all edges



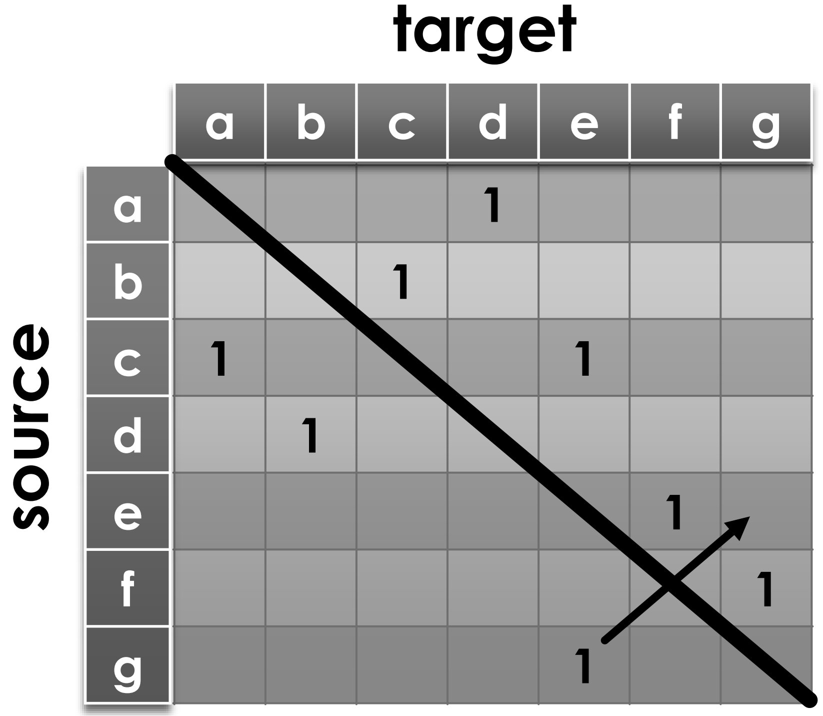
	a	b	c	d	e	f	g
a				1			
b			1				
c	1				1		
d		1					
e						1	
f							1
g					1		

	a	b	c	d	e	f	g
a			1				
b				1			
c		1					
d	1						
e			1				
f					1		
g							

REVERSING EDGES: ADJACENCY **MATRIX**

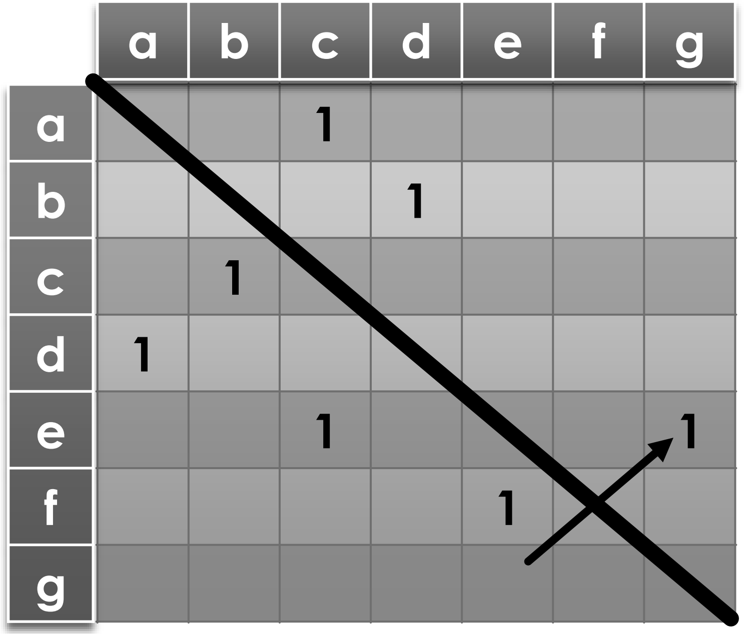


reverse all edges



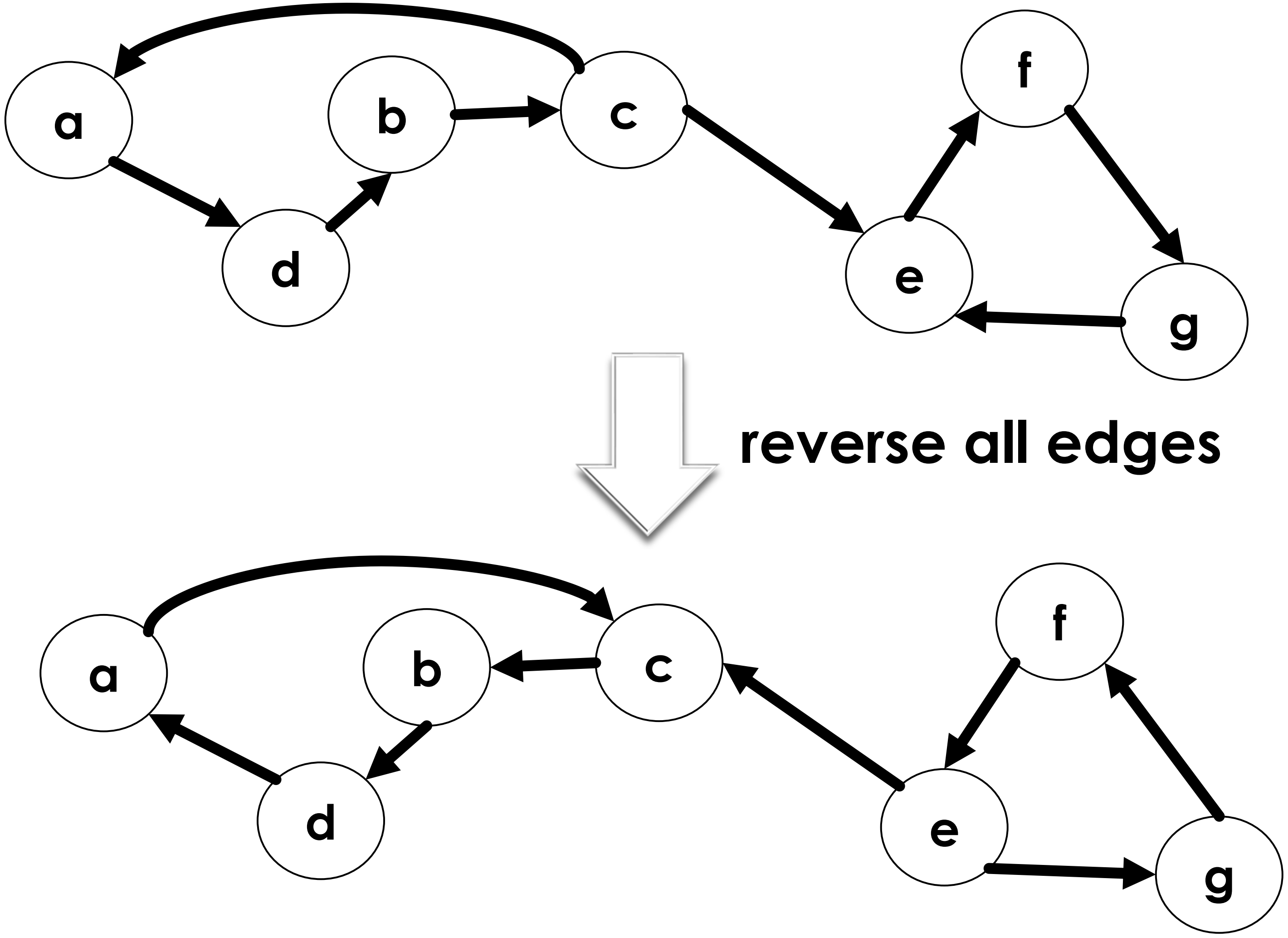
target

source	a	b	c	d	e	f	g
a				1			
b			1				
c	1				1		
d		1					
e						1	
f							1
g					1		

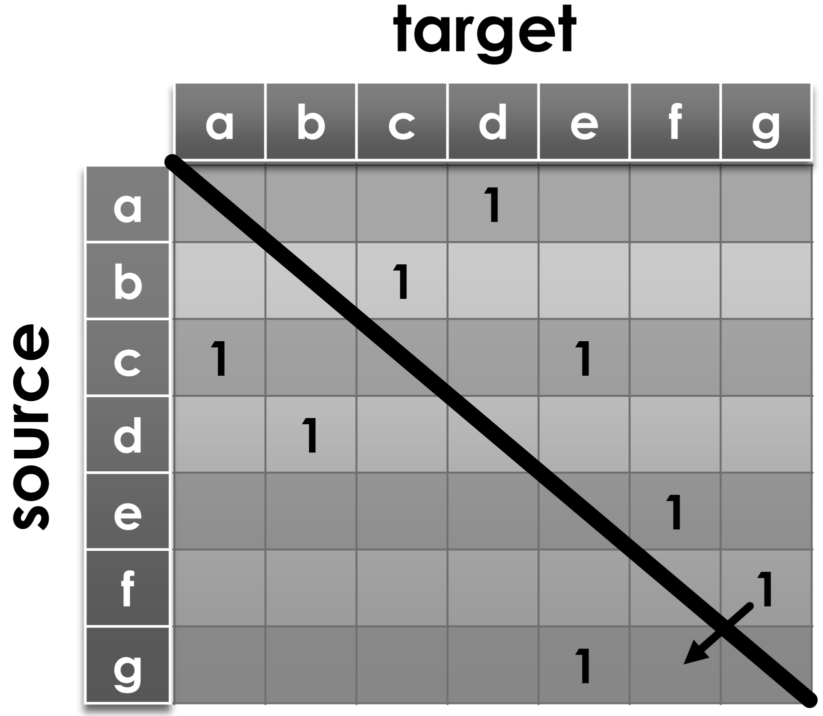


	a	b	c	d	e	f	g
a			1				
b				1			
c		1					
d	1						
e			1				1
f					1		
g							

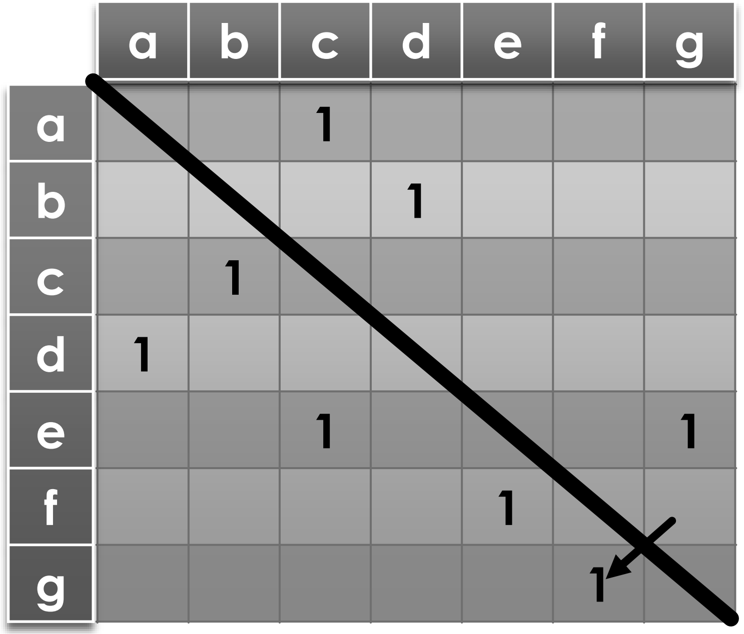
REVERSING EDGES: ADJACENCY **MATRIX**



reverse all edges

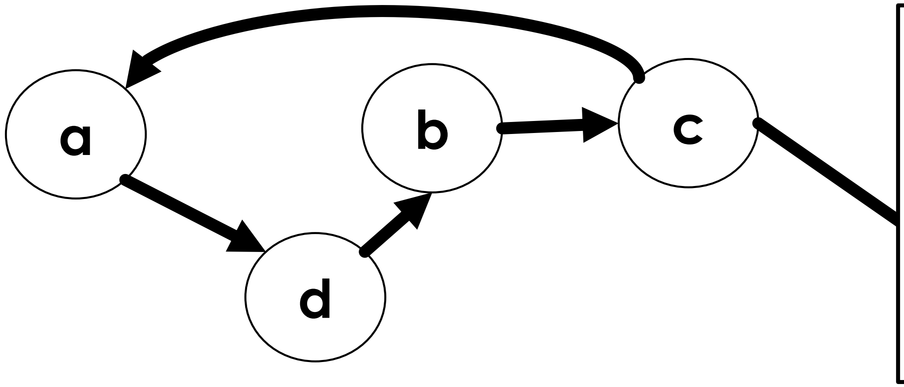


	a	b	c	d	e	f	g
a				1			
b			1				
c	1				1		
d		1					
e						1	
f							1
g					1		



	a	b	c	d	e	f	g
a			1				
b				1			
c		1					
d	1						
e			1				1
f					1		
g						1	

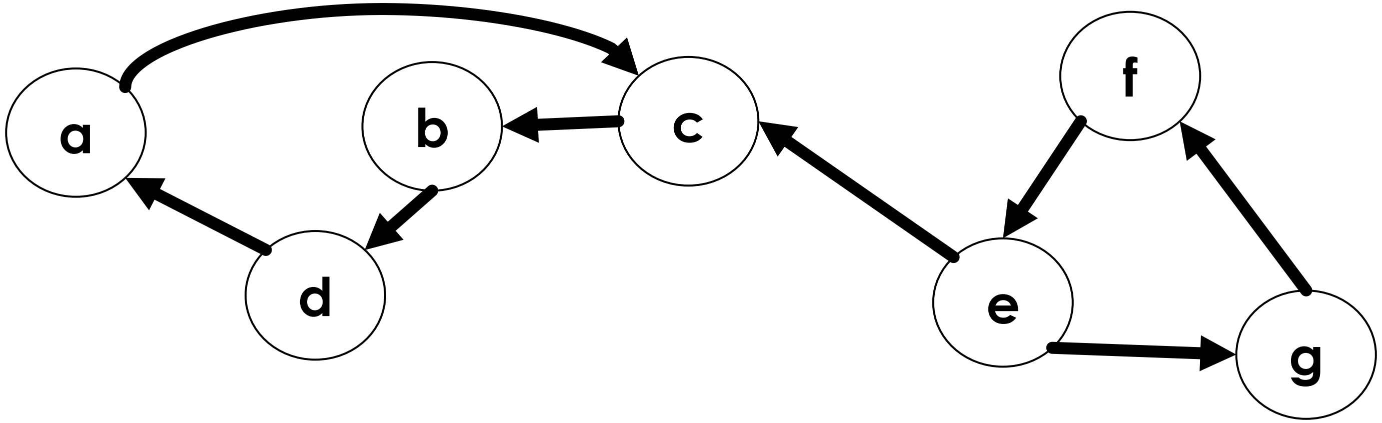
REVERSING EDGES: ADJACENCY **MATRIX**



Can do matrix transpose, **or** can just *treat rows as columns and vice versa in your code*

Complexity?

reverse all edges



target

source

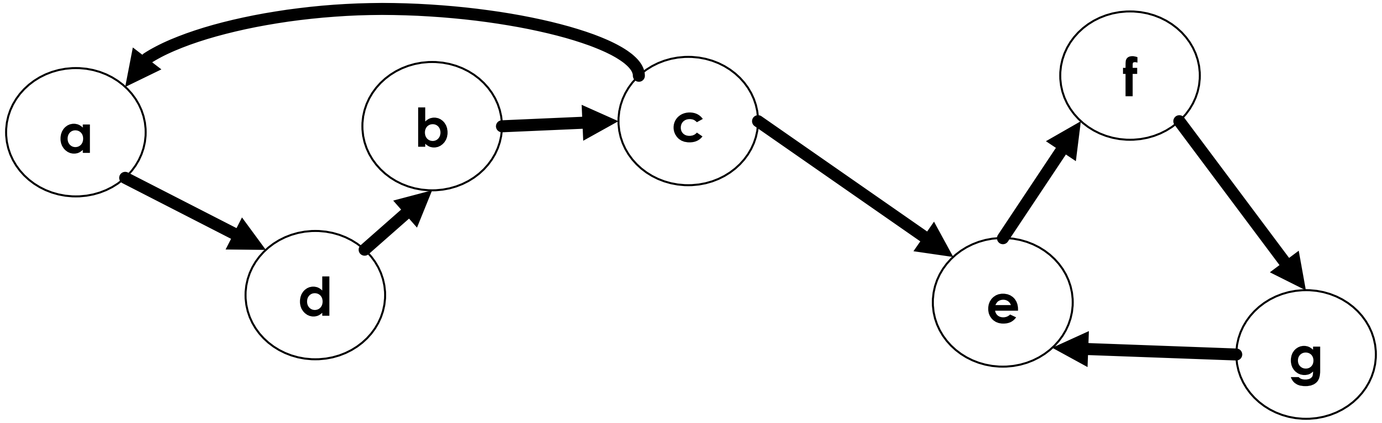
M_E

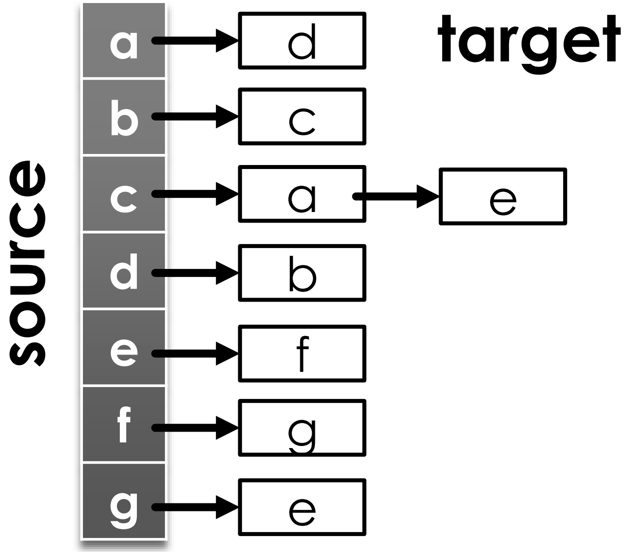
	a	b	c	d	e	f	g
a				1			
b			1				
c	1				1		
d		1					
e						1	
f							1
g					1		

$(M_E)^T$

	a	b	c	d	e	f	g
a			1				
b				1			
c		1					
d	1						
e			1				1
f					1		
g						1	

REVERSING EDGES: ADJACENCY **LISTS**



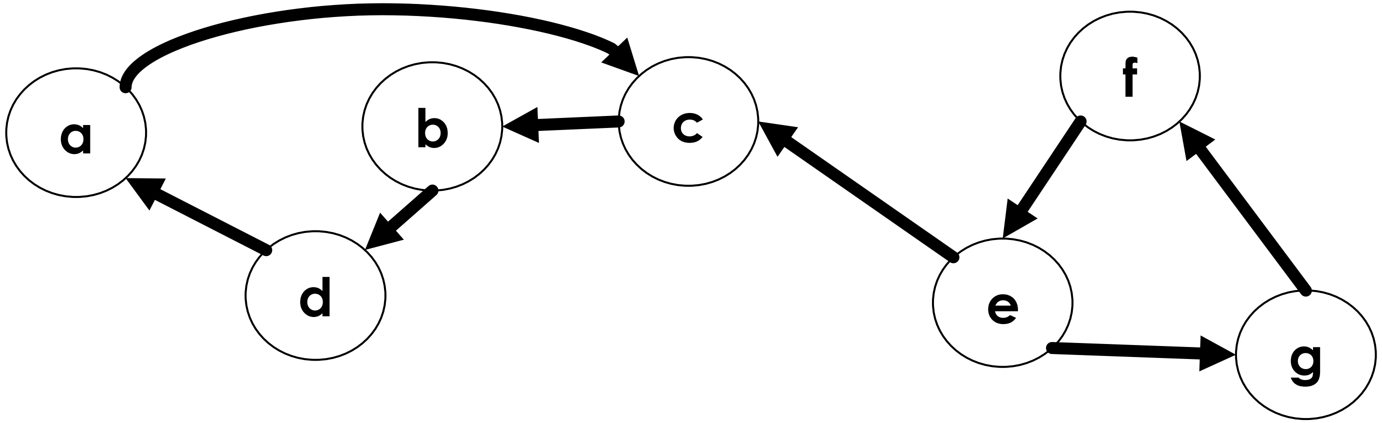


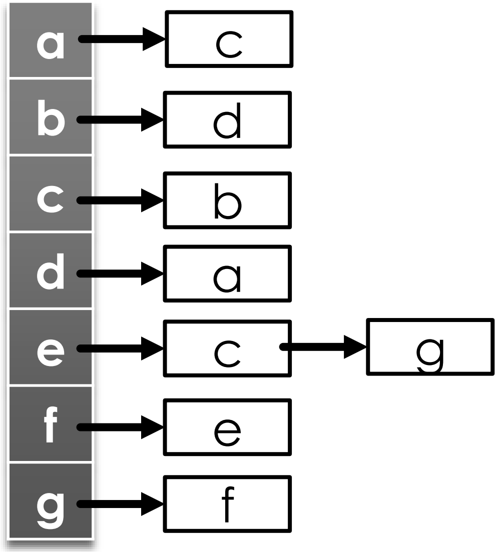
a → d
b → c
c → a → e
d → b
e → f
f → g
g → e

reverse edges

```
TransposeLists(adj[1..n])
    newAdj = new array of n lists
    for u = 1 .. n
        for v in adj[u]
            newAdj[v].insert(u)
    return newAdj
```

Complexity?





a → c
b → d
c → b
d → a
e → c → g
f → e
g → f

RUNTIME COMPLEXITY
FOR ADJACENCY LIST REPRESENTATION?

```
$IsStronglyConnected(G = \{V, E\})$ where $V = v_1, v_2, \ldots, v_n$
    $(colour, d, f) \coloneqq DFSVisit(v_1, G)$
    for $i \coloneqq 1..n$
        if $colour[v_i] \neq black$ then return $false$
    Construct graph $H$ by **reversing** all edges in $G$
    $(colour, d, f) \coloneqq DFSVisit(v_1, \boldsymbol{H})$
    for $i \coloneqq 1..n$
        if $colour[v_i] \neq black$ then return $false$
    return $true$
```

$\boxed{O(n + m)}$