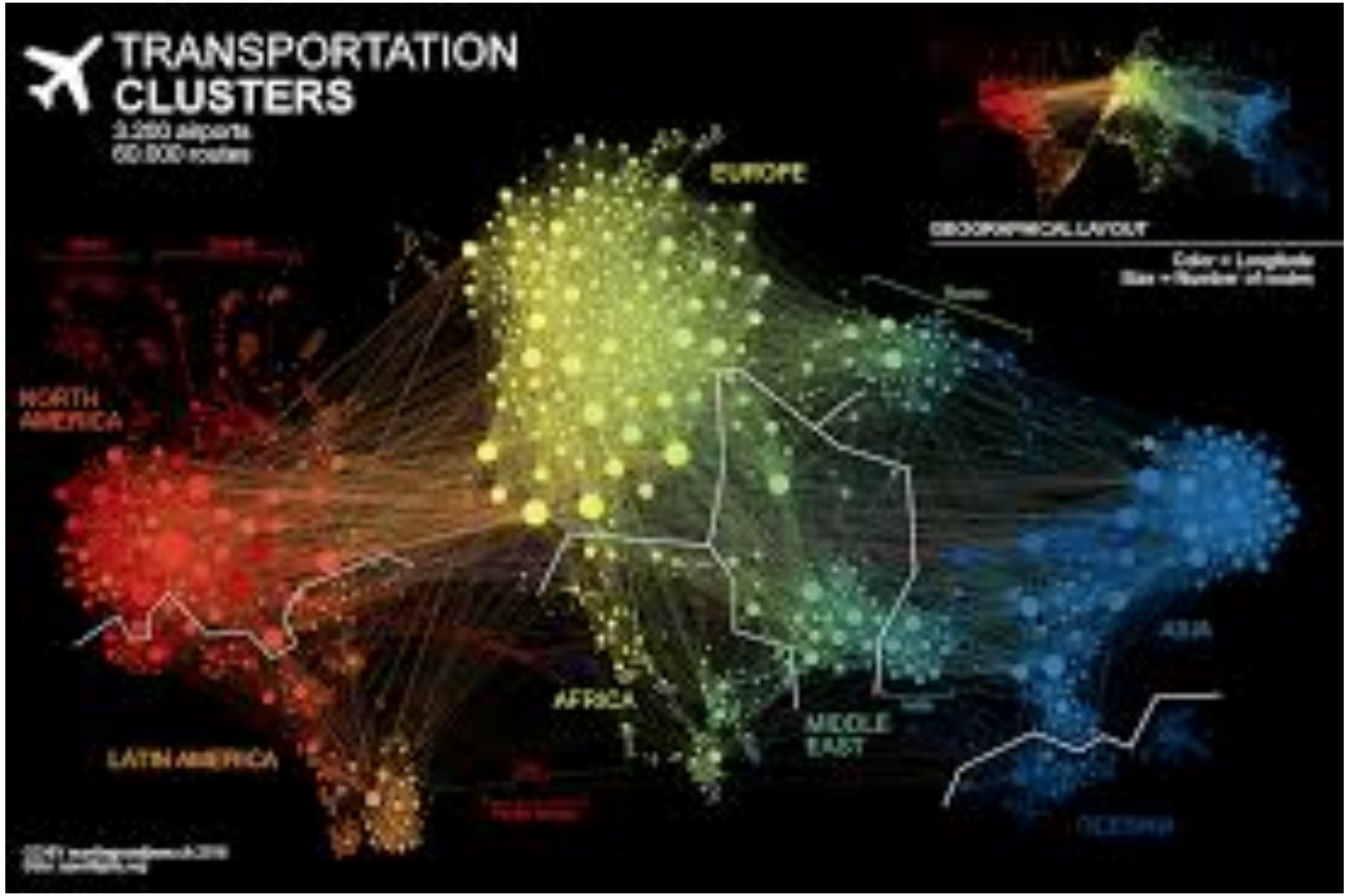
Lecture 6:

Elementary Graph Algorithms

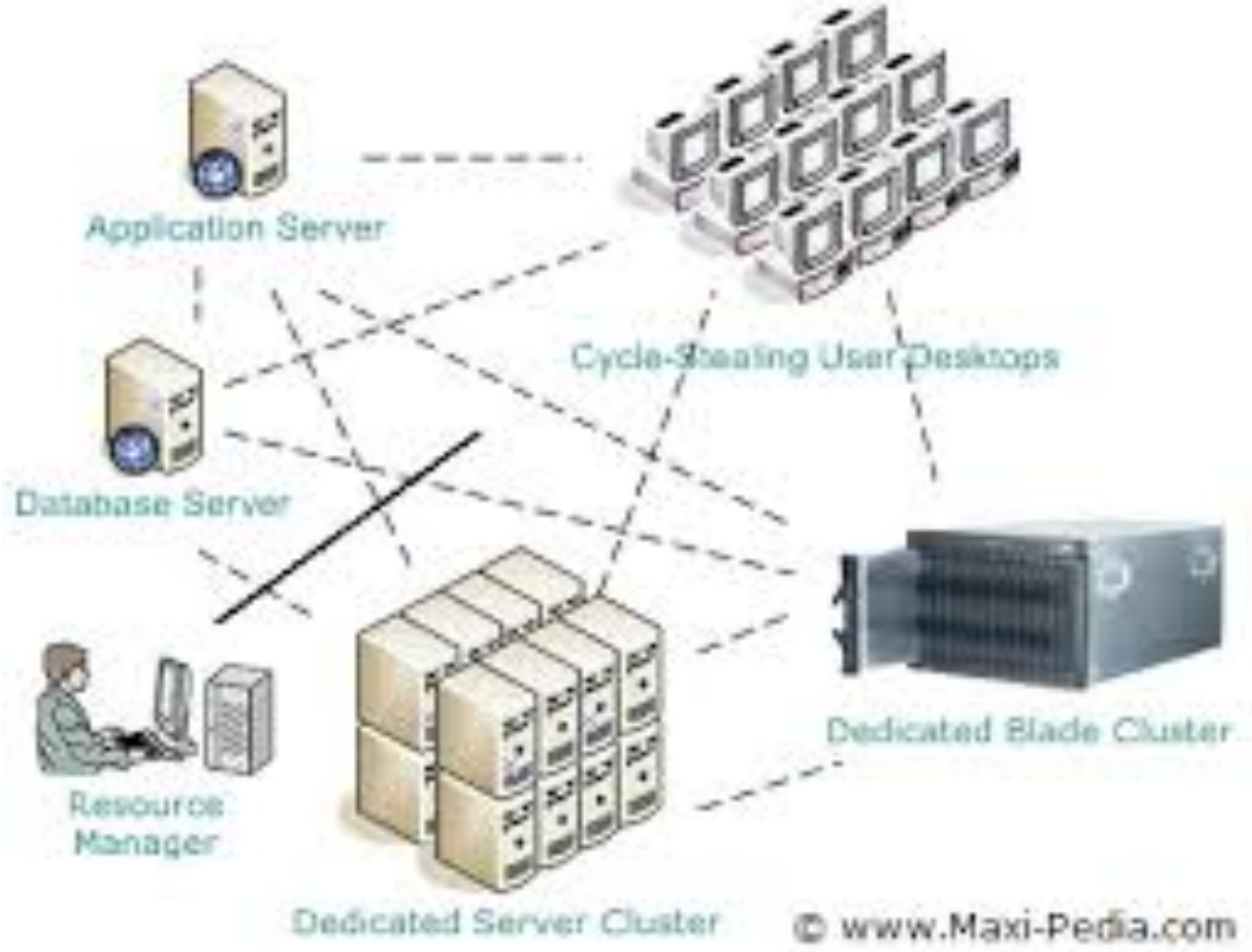
Algorimiek (INFOAL)

Graph: an Abstract Model of the World

Graph: an Abstract Model of the World

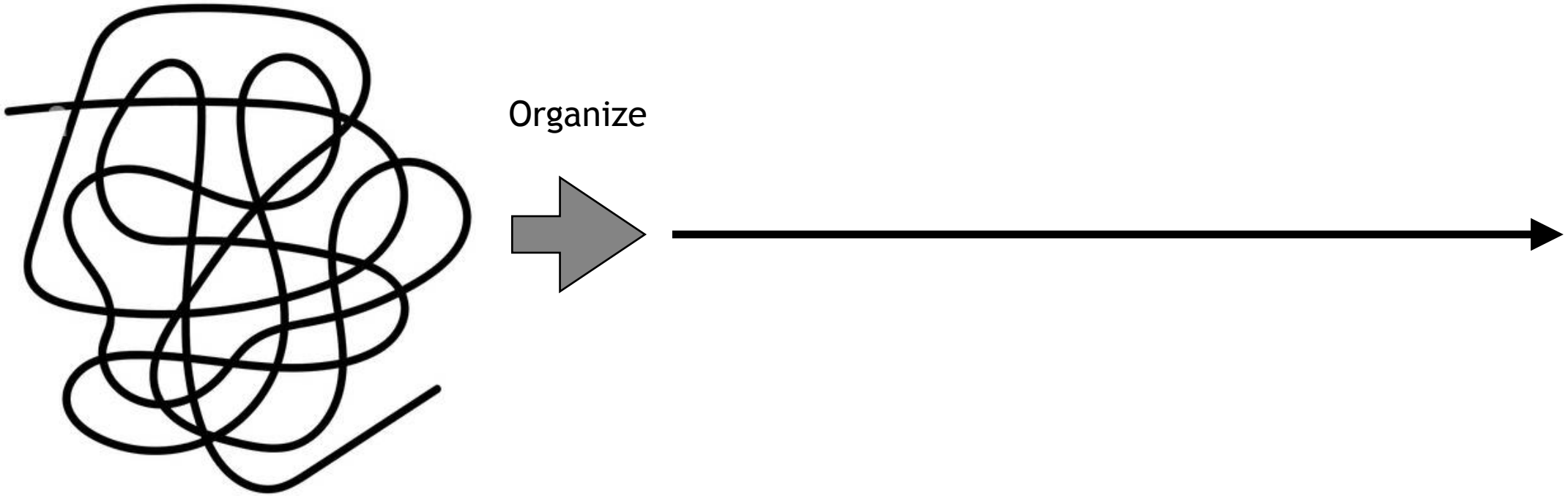


Graph: an Abstract Model of the World

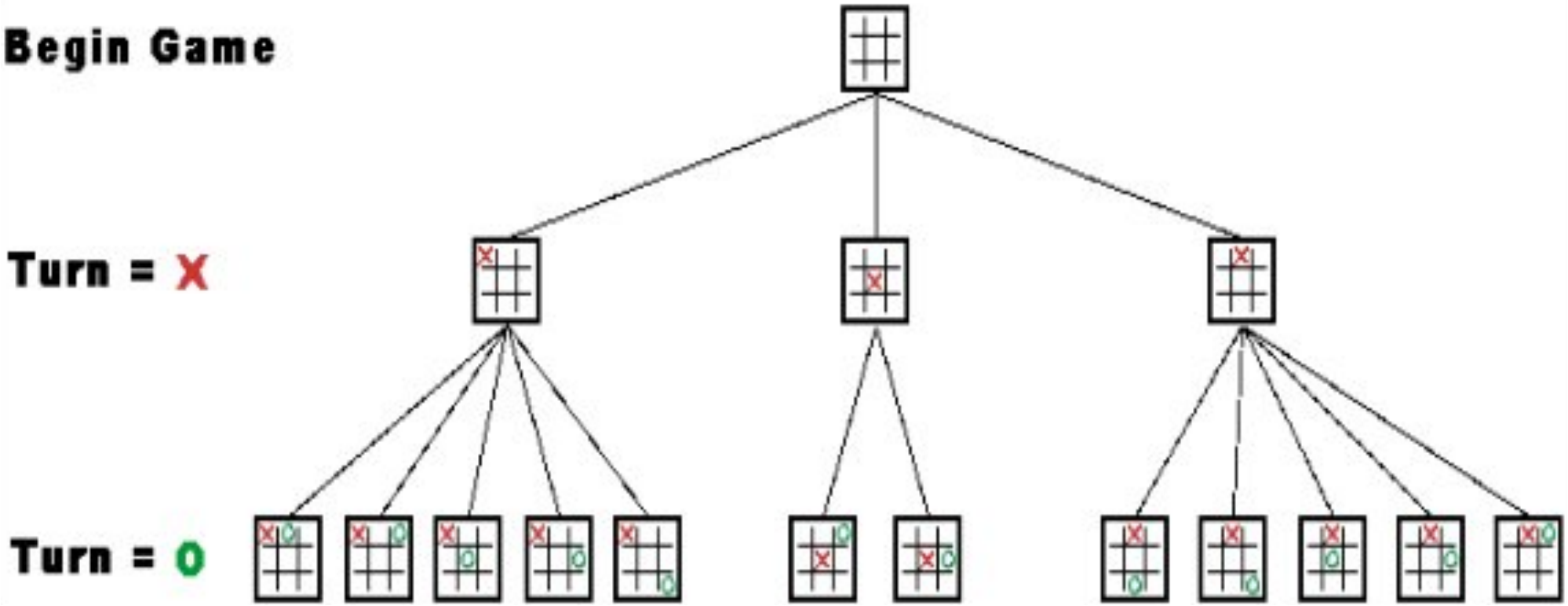


Graph: an Abstract Model of the World

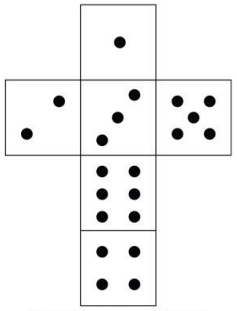
Graph: an Abstract Model of the World

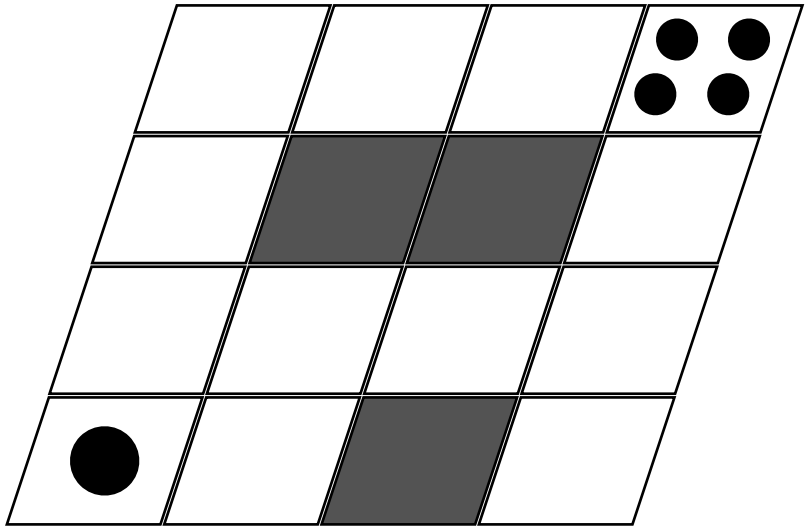


Graph: an Abstract Model of the World

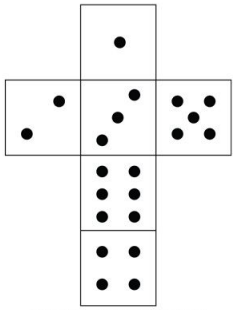


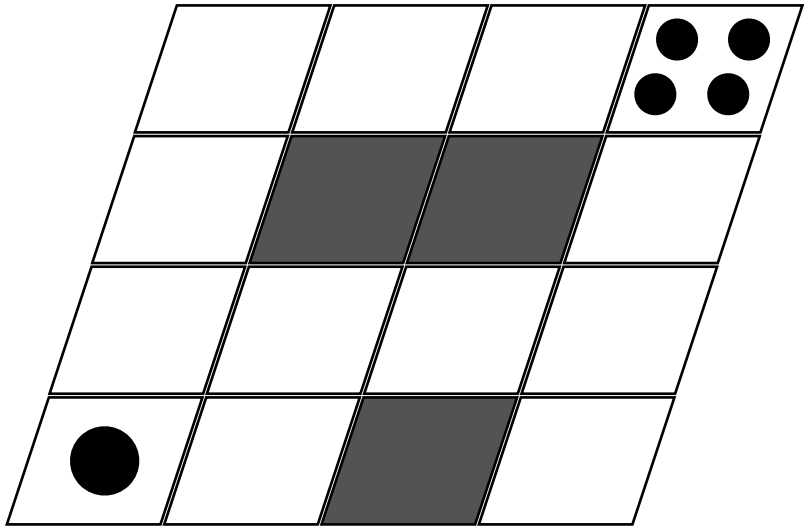
Graph: an Abstract Model of the World

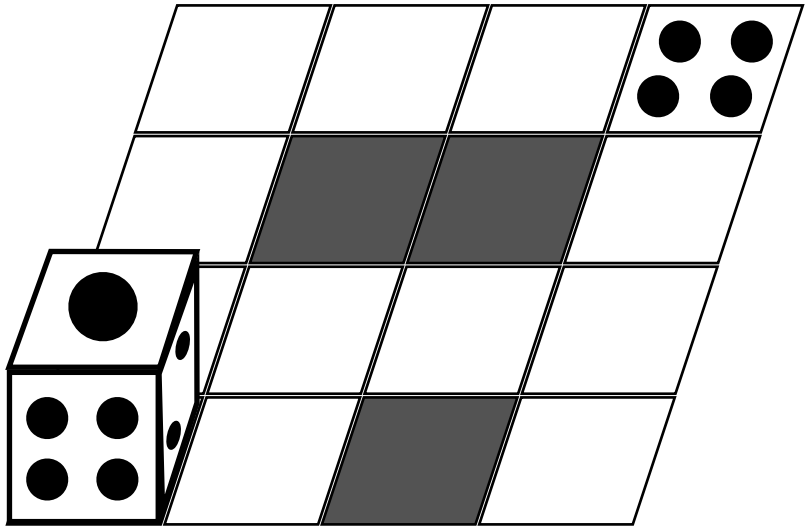




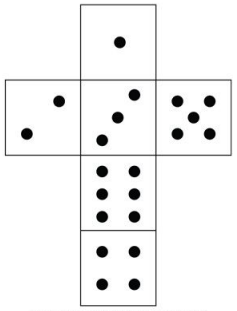
Graph: an Abstract Model of the World

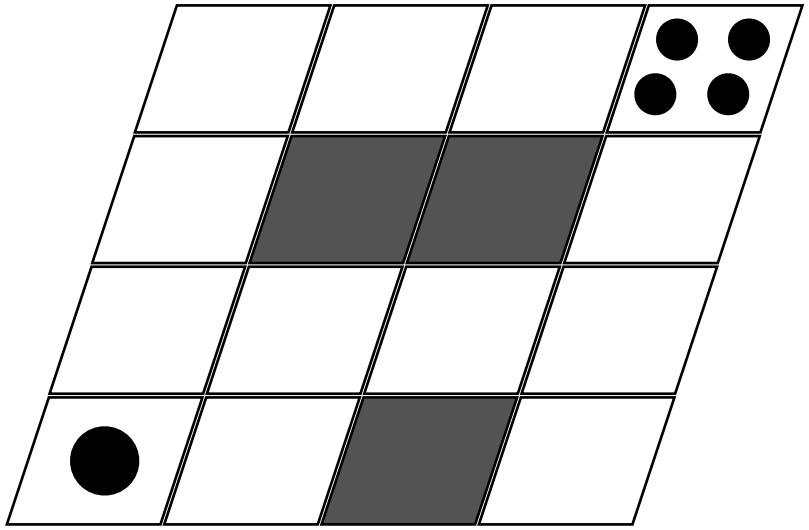


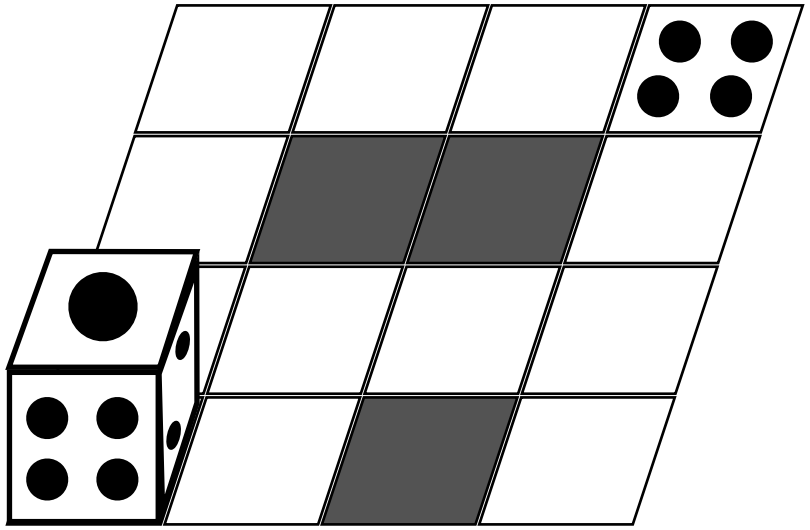


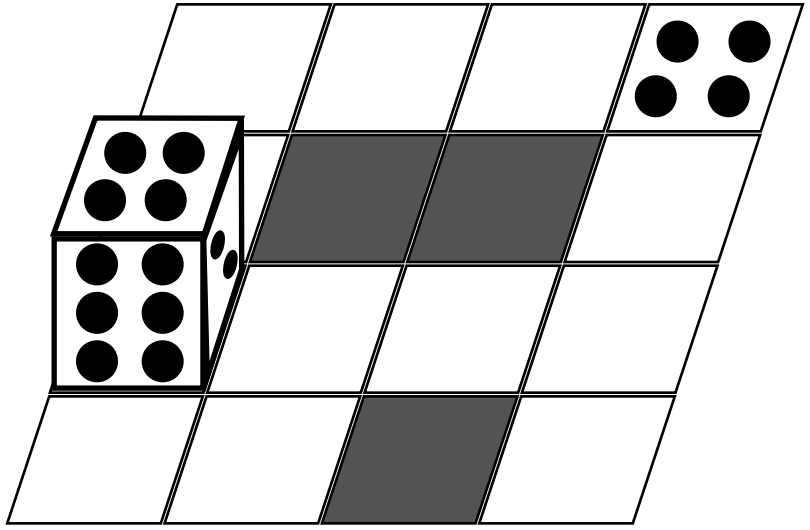


Graph: an Abstract Model of the World

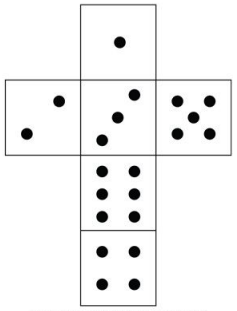


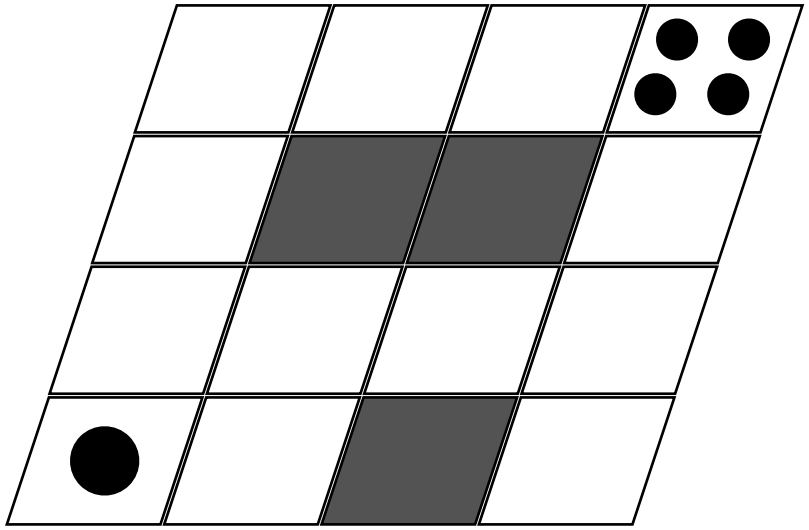


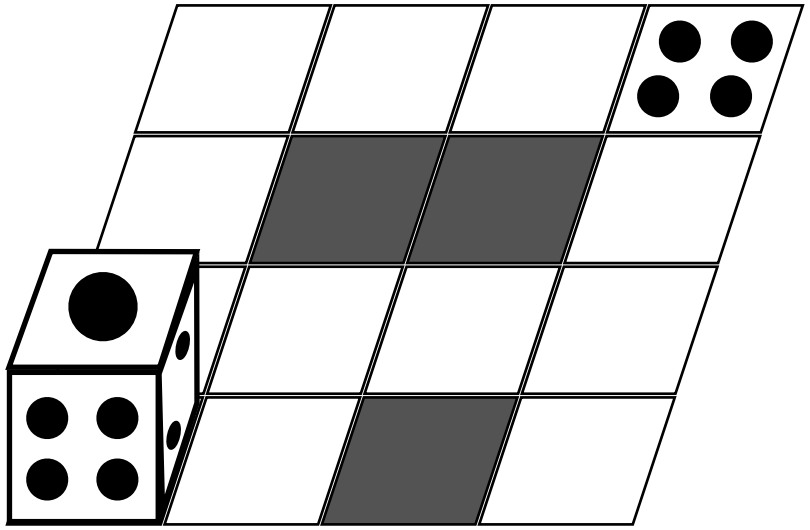


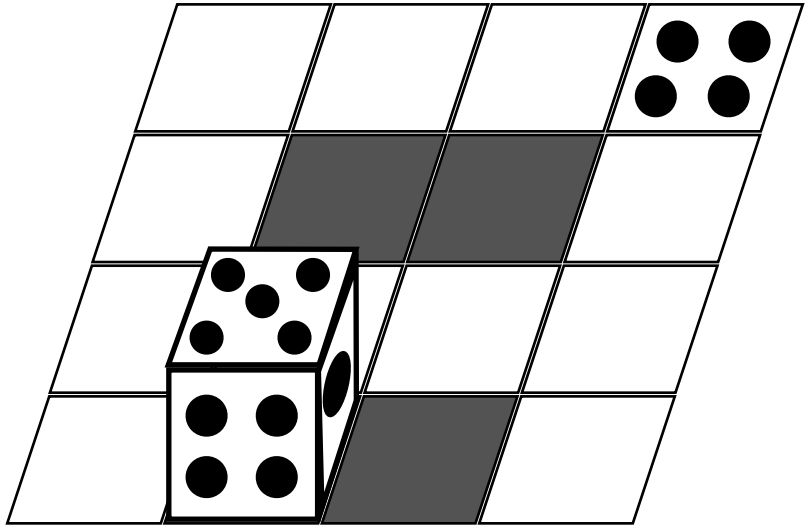


Graph: an Abstract Model of the World









Outline

- Graph representations
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
- Two applications on DFS:
  - Topological sort
  - Strongly connected components

How to Represent a Graph

- $G = (V, E)$

How to Represent a Graph

- $G = (V, E)$
  - V: set of vertices in the graph

How to Represent a Graph

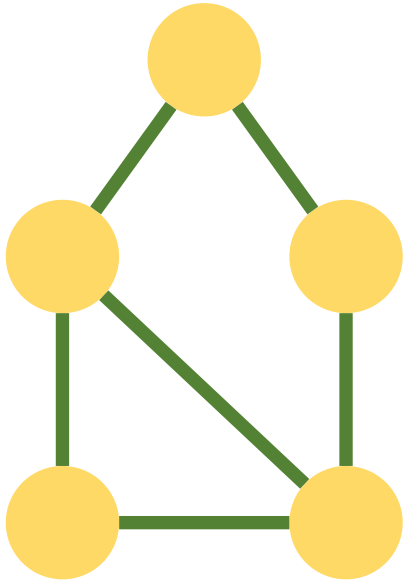
- $G = (V, E)$
  - V: set of vertices in the graph
  - E: set of edges in the graph

How to Represent a Graph

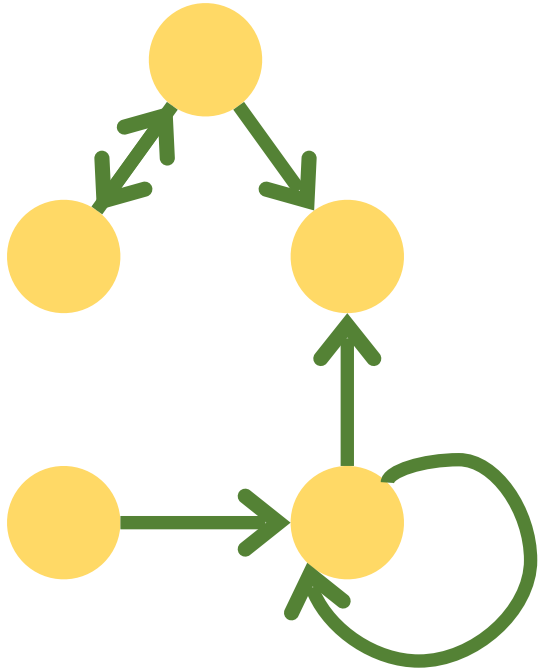
- $G = (V, E)$
  - V: set of vertices in the graph
    - $n = |V|$
  - E: set of edges in the graph
    - $m = |E|$

Undirected Graph and Directed Graph

- In directed graphs, the edges are ordered set. That is, $(u, v) \neq (v, u)$



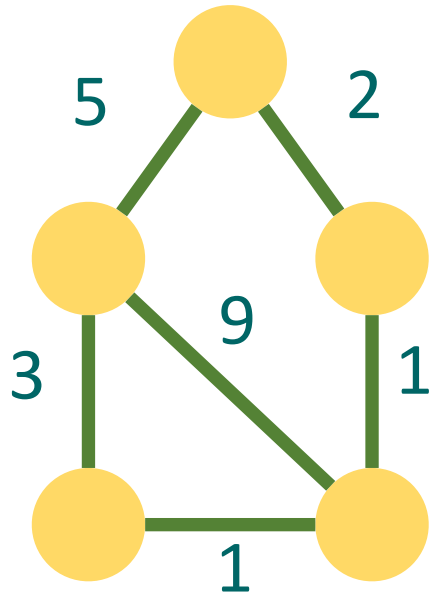
undirected graph

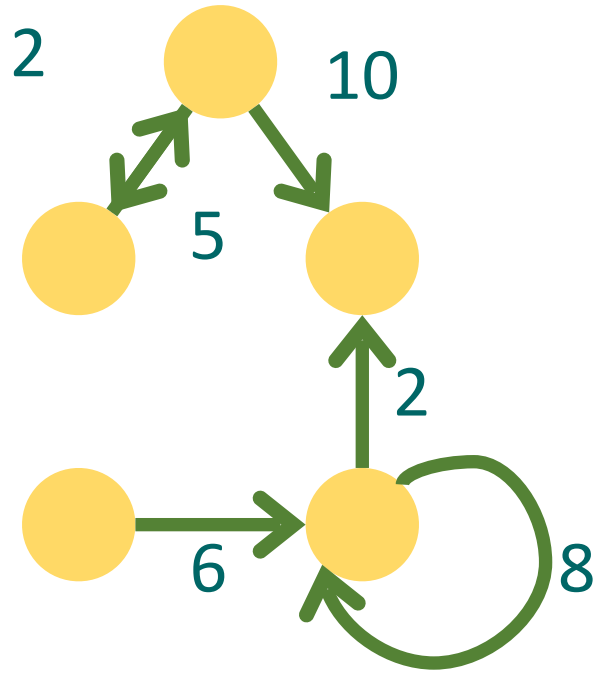


directed graph

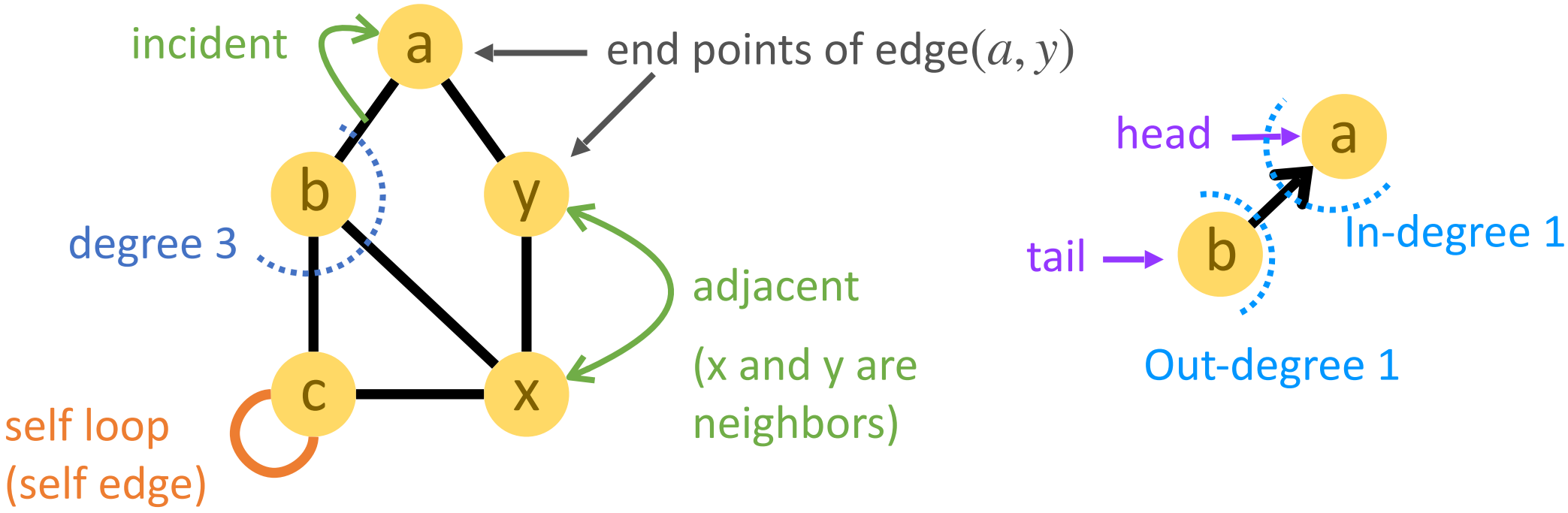
Weighted Graph

- The edges can be associated with **weights**.



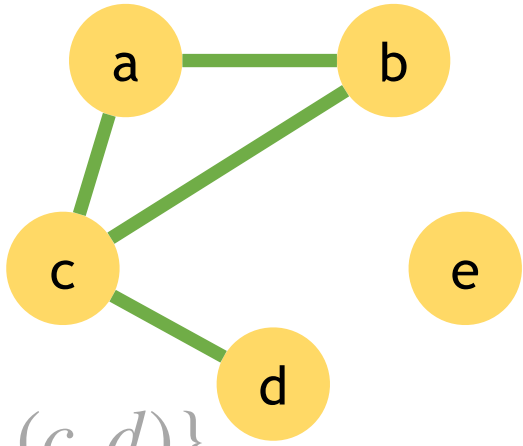


Basic Terminologies



How to Represent a Graph

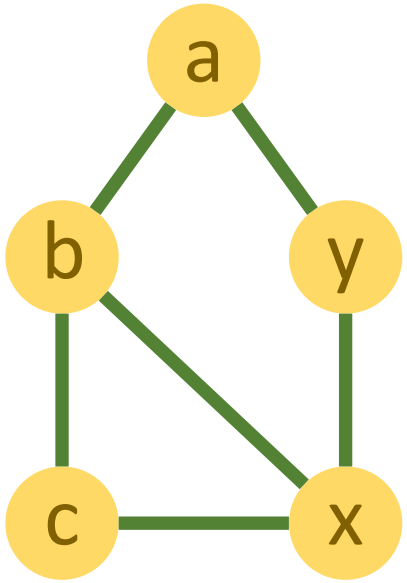
- $G = (V, E)$
  - V: set of vertices in the graph $V = \{a, b, c, d, e\}$
    - $n = |V| n = 5$
  - E: set of edges in the graph $E = \{(a, b), (a, c), (b, c), (c, d)\}$
    - $m = |E| m = 4$
- To represent the relationship between vertices and edges, we can use
  - **Adjacency list**
  - **Adjacency matrix**

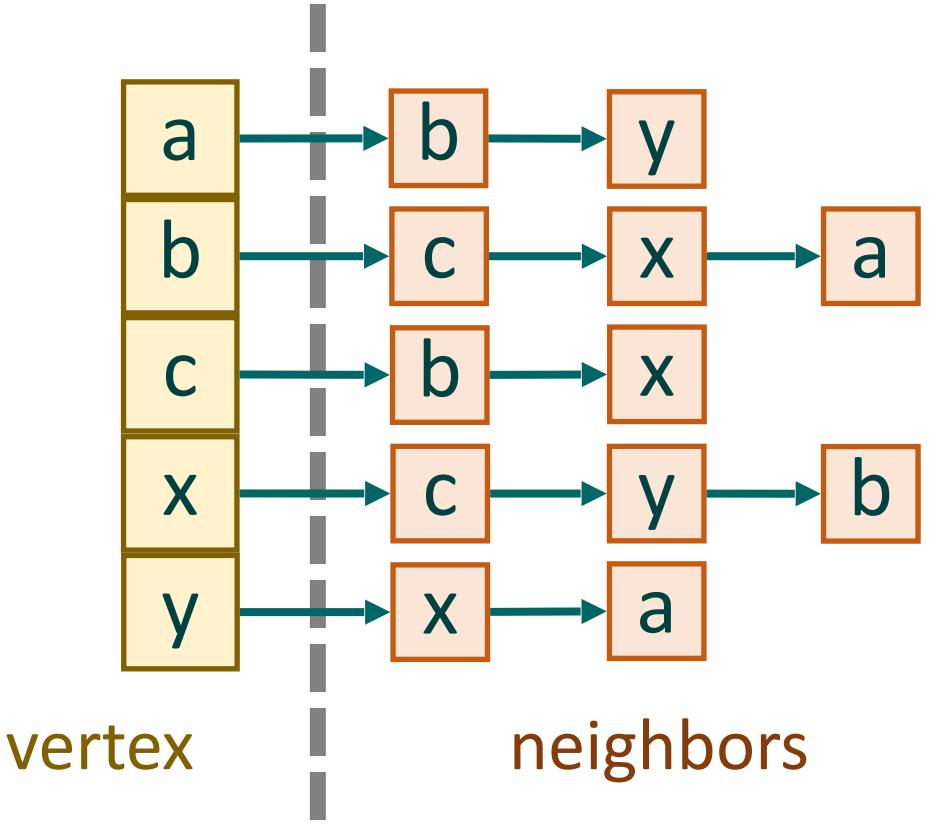


Adjacency List

Adjacency List

- For each vertex, store its neighbors in a linked list



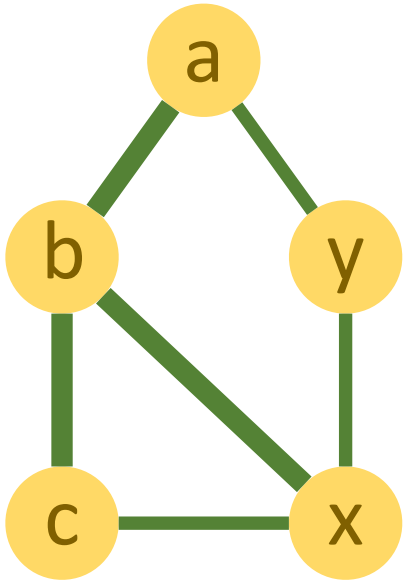


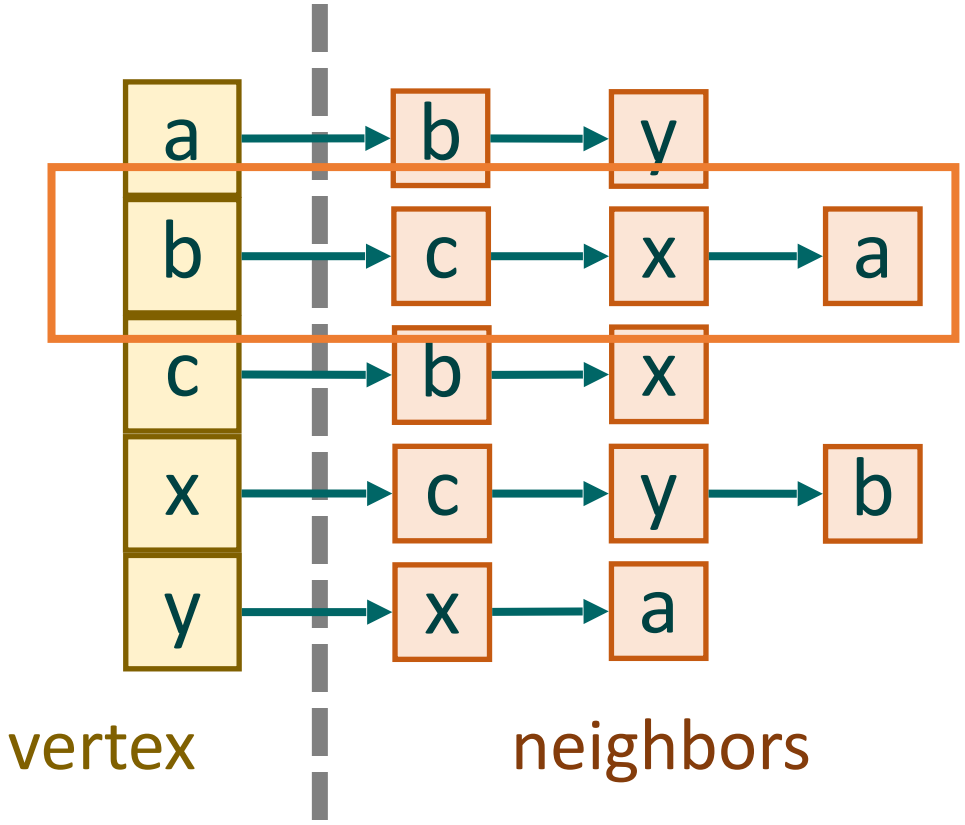
$$V = \{a, b, c, x, y\}$$

$$E = \{(a, b), (a, y), (b, c), (b, x), (c, x), (x, y)\}$$

Adjacency List

- For each vertex, store its neighbors in a linked list



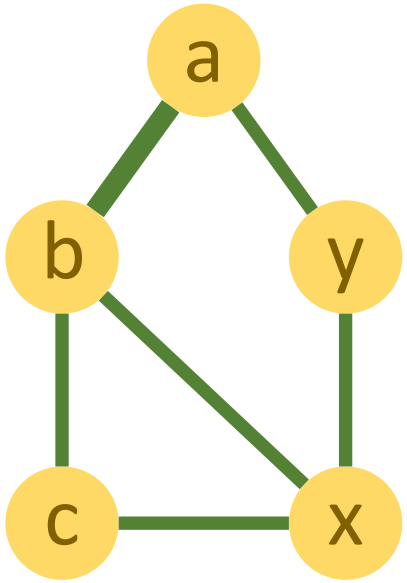


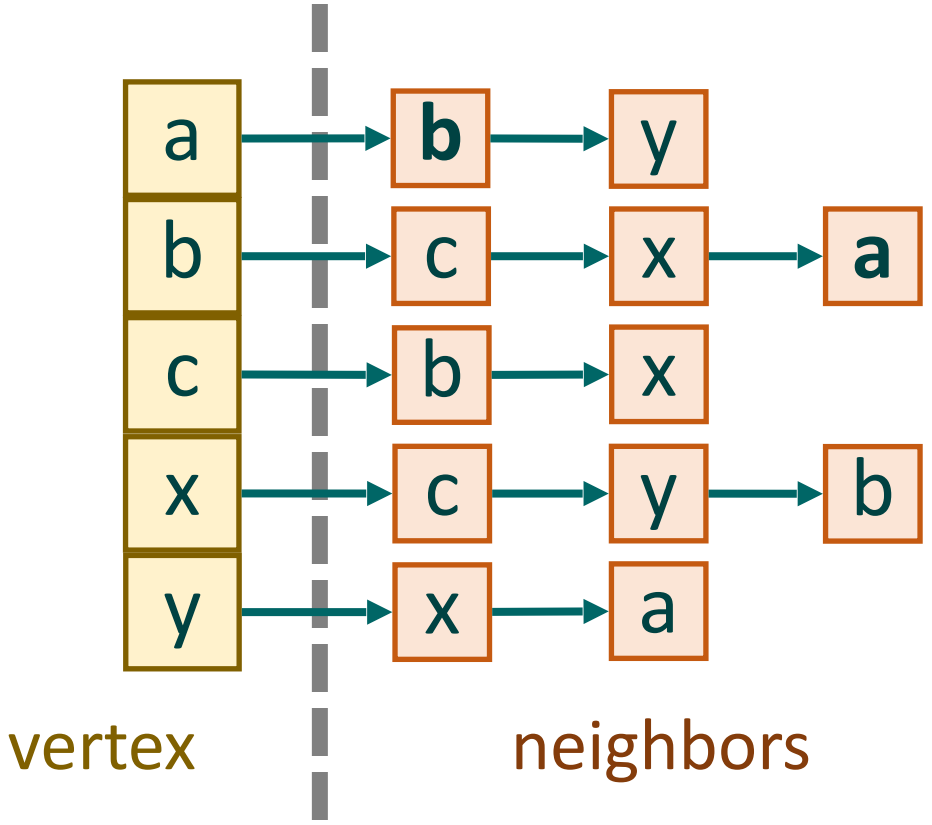
$$V = \{a, b, c, x, y\}$$

$$E = \{(a, b), (a, y), (b, c), (b, x), (c, x), (x, y)\}$$

Adjacency List

- For each vertex, store its neighbors in a linked list



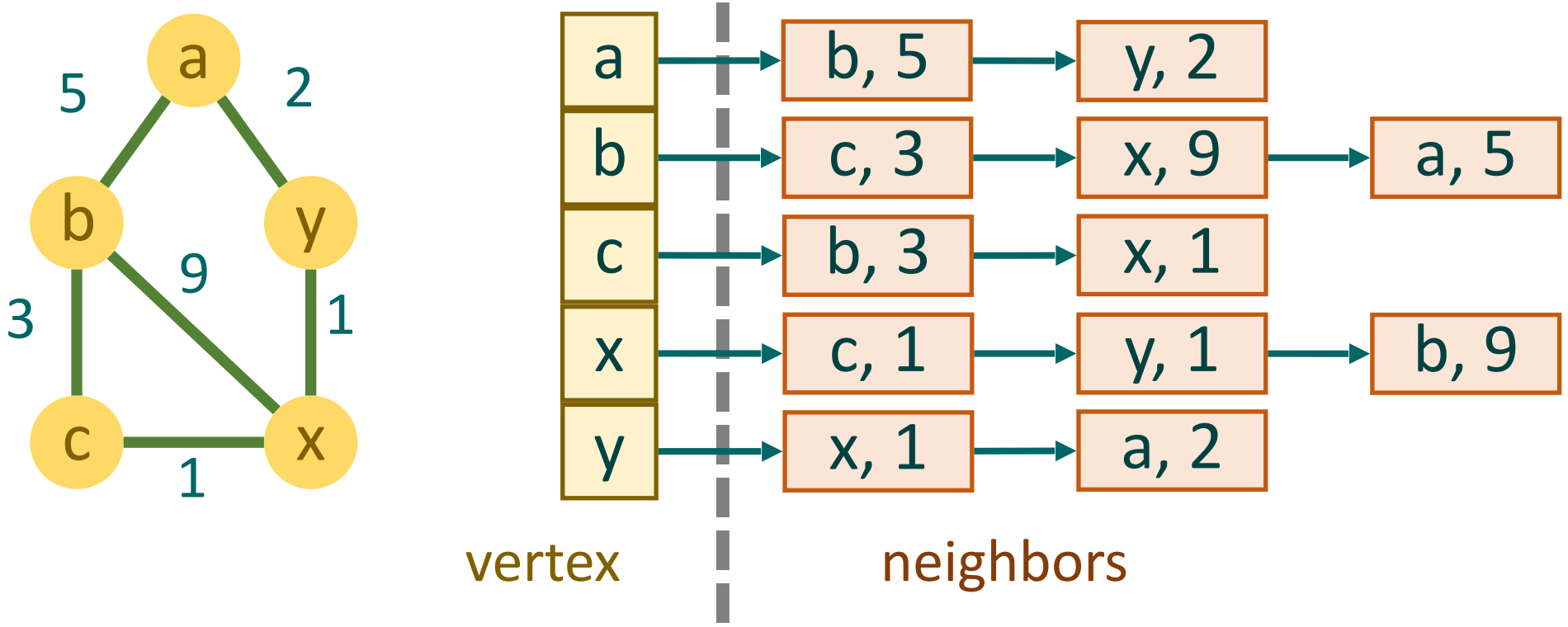


$$V = \{a, b, c, x, y\}$$

$$E = \{(a, b), (a, y), (b, c), (b, x), (c, x), (x, y)\}$$

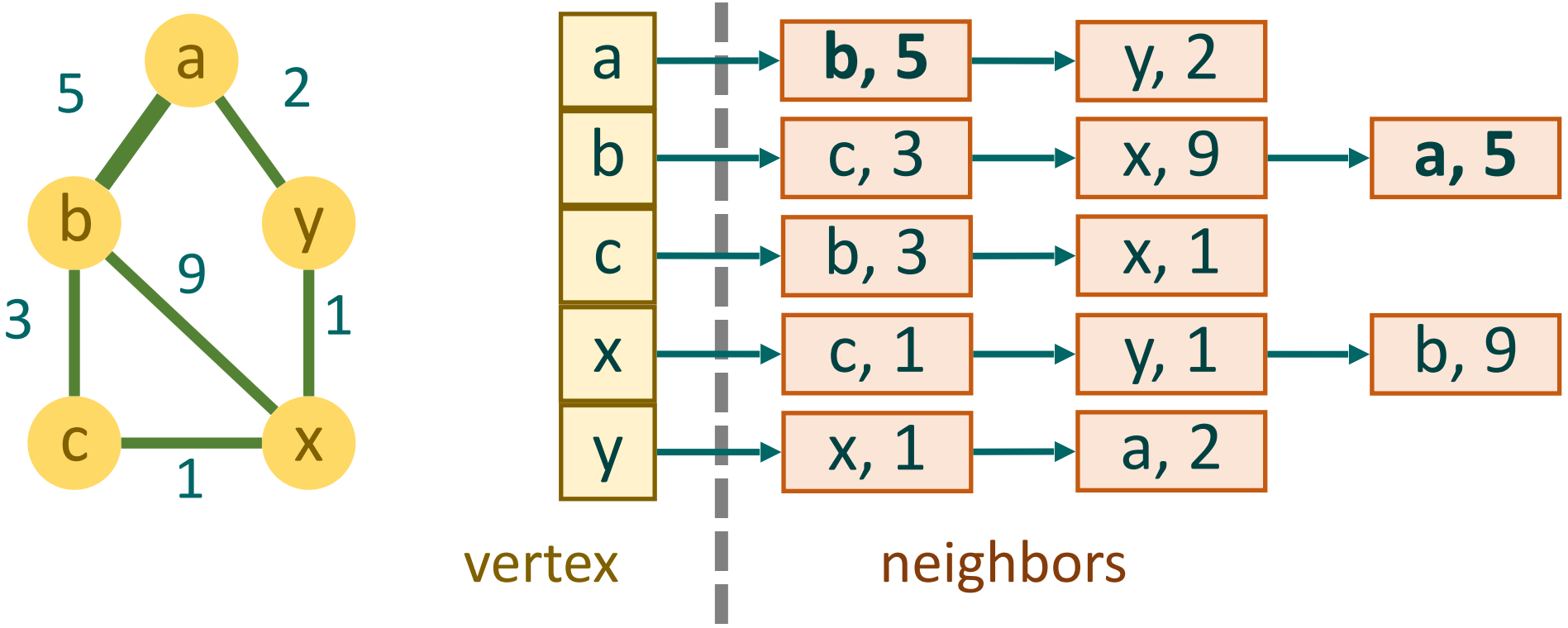
Adjacency List

- For each vertex, store its neighbors in a linked list



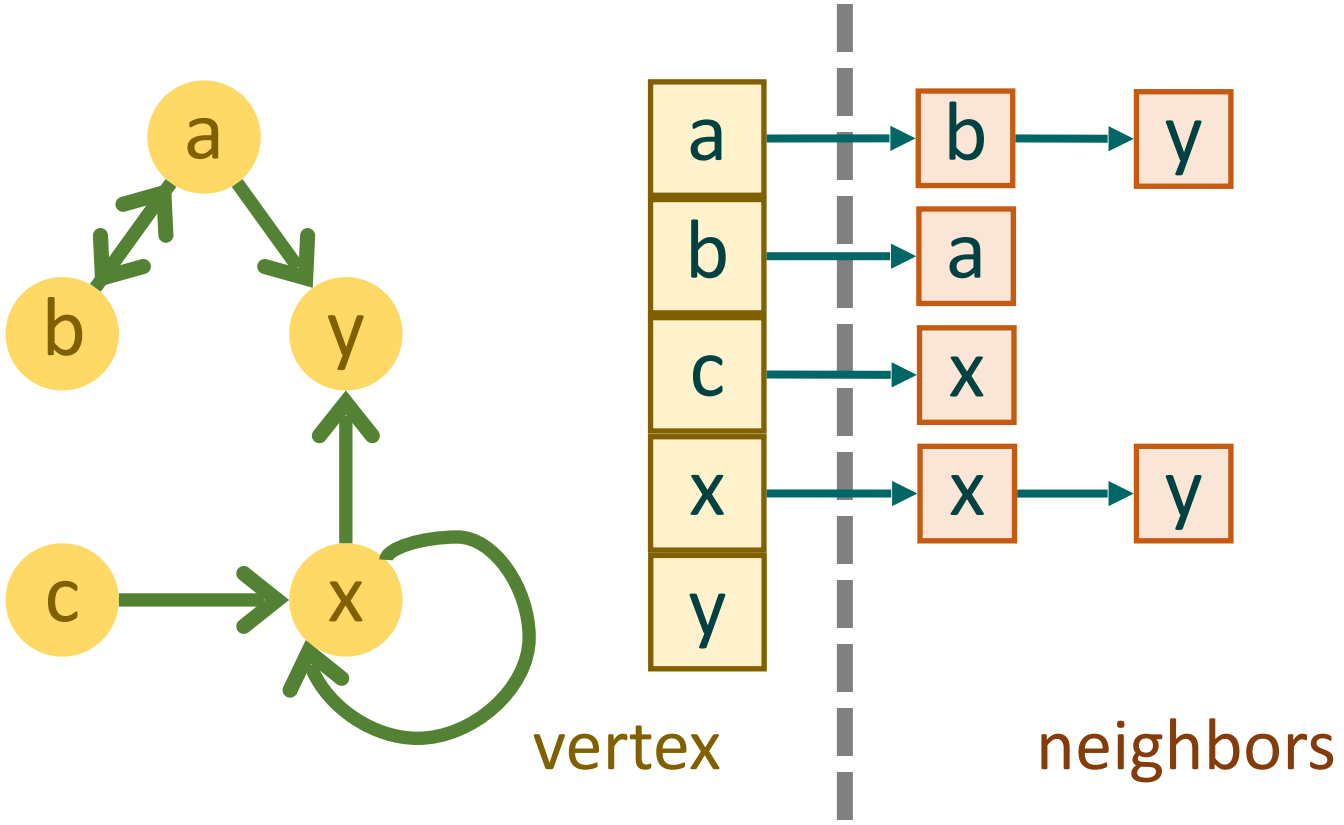
Adjacency List

- For each vertex, store its neighbors in a linked list



Adjacency List

- For each vertex, store its neighbors in a linked list

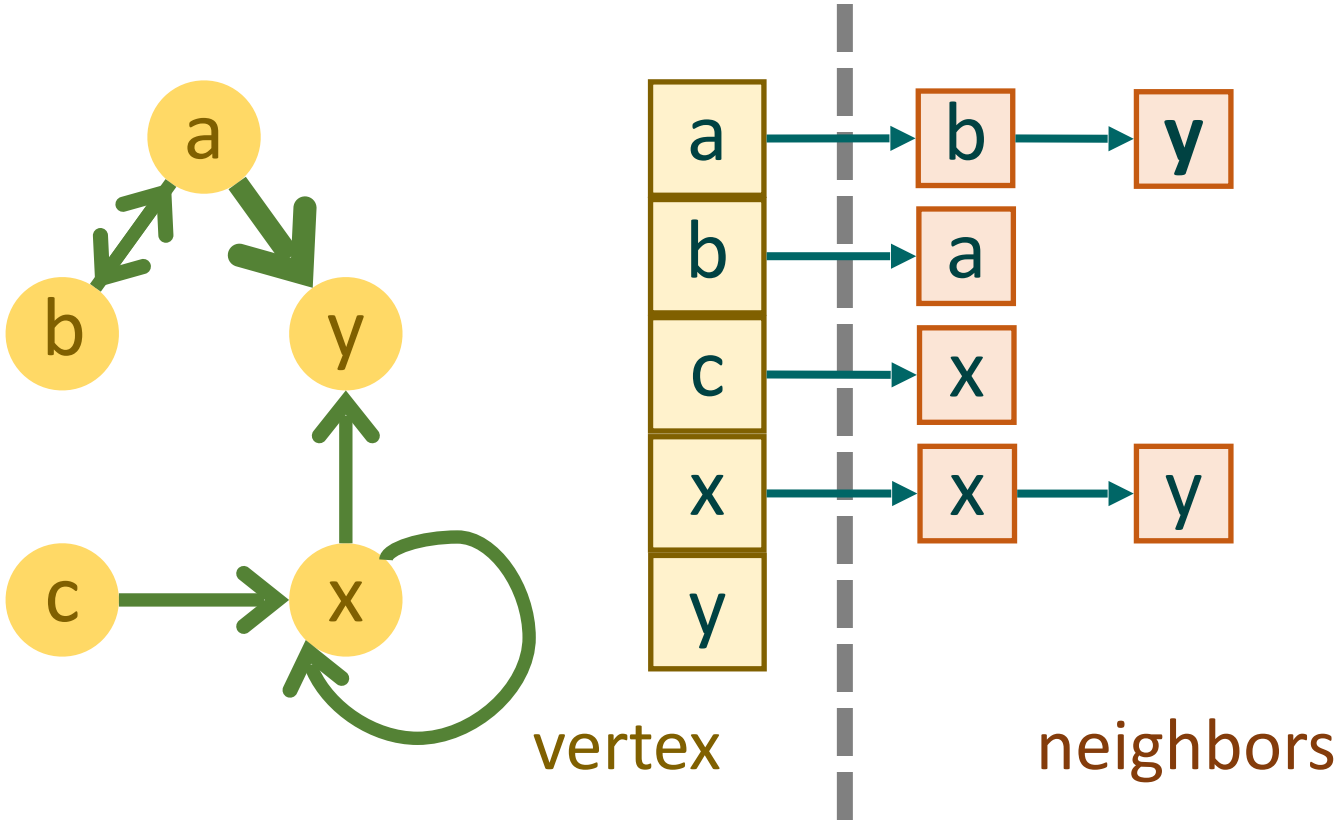


$$V = \{a, b, c, x, y\}$$

$$E = \{(a, b), (b, a), (a, y), (x, y), (x, x), (c, x)\}$$

Adjacency List

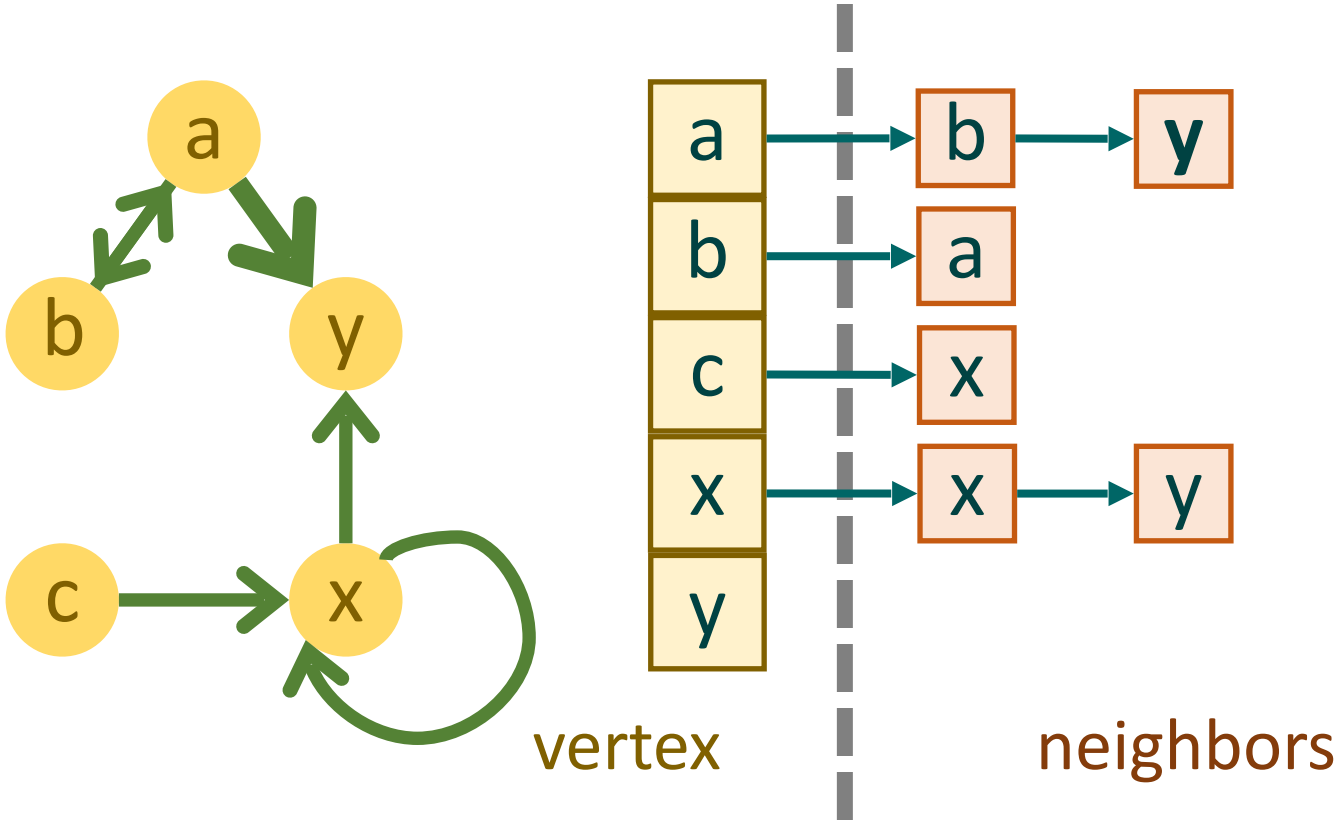
- For each vertex, store its neighbors in a linked list



- Space complexity: $O(|E|)$

Adjacency List

- For each vertex, store its neighbors in a linked list



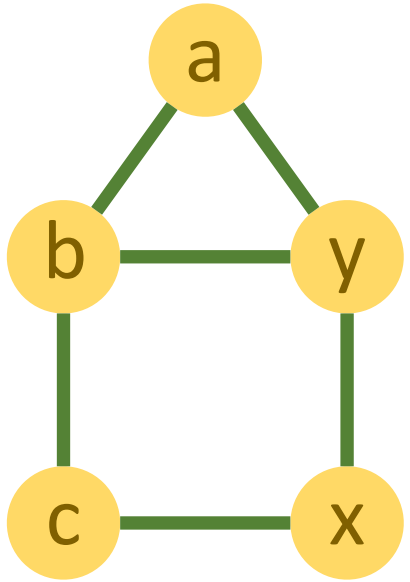
check if u and v are adjacent

- Space complexity: $O(|E|)$
- Querying complexity: $O(\text{max degree})$
  - For each vertex, store its neighbors in a linked list

Adjacency Matrix

Adjacency Matrix

- Use a $|V| \times |V|$ matrix A such that
  - $A(u, v) = 1$ iff $(u, v) \in E$
  - $A(u, v) = 0$ otherwise
- Space complexity: $O(|V|^2)$

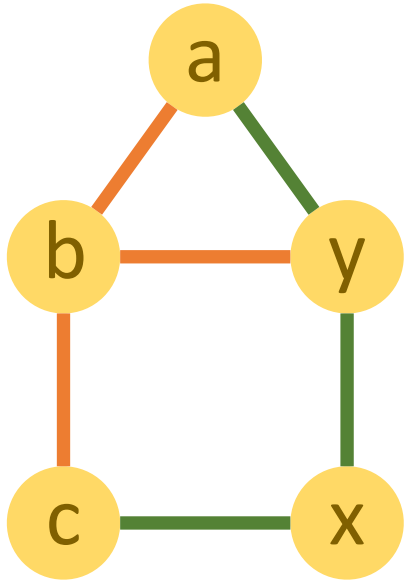


	a	b	c	x	y
a	0	1	0	0	1
b	1	0	1	0	1
c	0	1	0	1	0
x	0	0	1	0	1
y	1	1	0	1	0

Adjacency Matrix

- Use a $|V| \times |V|$ matrix A such that
  - $A(u, v) = 1$ iff $(u, v) \in E$
  - $A(u, v) = 0$ otherwise
- Space complexity: $O(|V|^2)$
- Querying complexity: $O(1)$
  - Check adjacency of vertices u and v

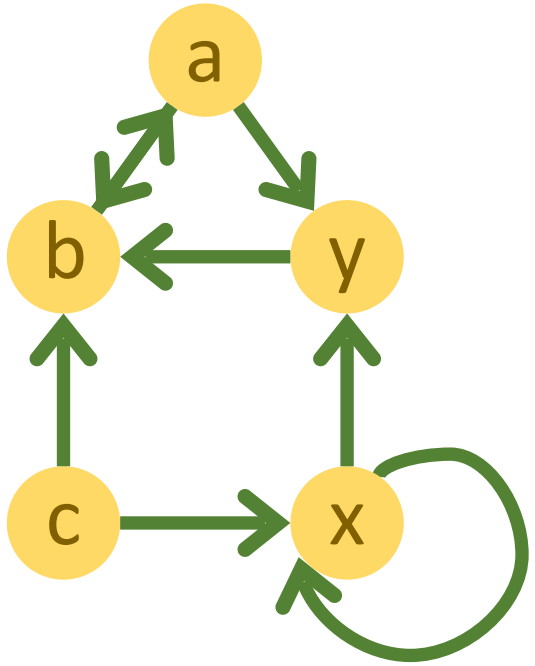
check if u and v are adjacent



	a	b	c	x	y
a	0	1	0	0	1
b	1	0	1	0	1
c	0	1	0	1	0
x	0	0	1	0	1
y	1	1	0	1	0

Adjacency Matrix

- Use a $|V| \times |V|$ matrix A such that
  - $A(u, v) = 1$ iff $(u, v) \in E$
  - $A(u, v) = 0$ otherwise
- Space complexity: $O(|V|^2)$
- Querying complexity: $O(1)$
  - Check adjacency of vertices u and v
- For directed graphs, A is not necessary symmetric



	a	b	c	x	y
a	0	1	0	0	1
b	1	0	0	0	0
c	0	1	0	1	0
x	0	0	0	1	1
y	0	1	0	0	0

Adjacency List and Adjacency Matrix

	Adjacency list	Adjacency matrix
Space	$O(\|V\|+\|E\|)$	$O(\|V\|^2)$
Add an edge		
Delete an edge		
List all neighbors of a vertex		
Check adjacency of vertices u and v		
List all edges		

Adjacency List and Adjacency Matrix

	Adjacency list	Adjacency matrix
Space	$O(\|V\|+\|E\|)$	$O(\|V\|^2)$
Add an edge	$O(1)$	$O(1)$
Delete an edge		
List all neighbors of a vertex		
Check adjacency of vertices u and v		
List all edges		

Adjacency List and Adjacency Matrix

	Adjacency list	Adjacency matrix
Space	$O(\|V\|+\|E\|)$	$O(\|V\|^2)$
Add an edge	$O(1)$	$O(1)$
Delete an edge	$O(\text{max degree}) = O(\|V\|)$	$O(1)$
List all neighbors of a vertex	$O(\#\text{ of neighbors}) = O(\|V\|)$	$O(\|V\|)$
Check adjacency of vertices u and v		
List all edges		

Adjacency List and Adjacency Matrix

	Adjacency list	Adjacency matrix
Space	$O(\|V\|+\|E\|)$	$O(\|V\|^2)$
Add an edge	$O(1)$	$O(1)$
Delete an edge	O(max degree) = $O(\|V\|)$	$O(1)$
List all neighbors of a vertex	O(# of neighbors) = $O(\|V\|)$	$O(\|V\|)$
Check adjacency of vertices u and v	O(max degree) = $O(\|V\|)$	$O(1)$
List all edges	$O(\|E\|)$	$O(\|V\|^2)$

Outline

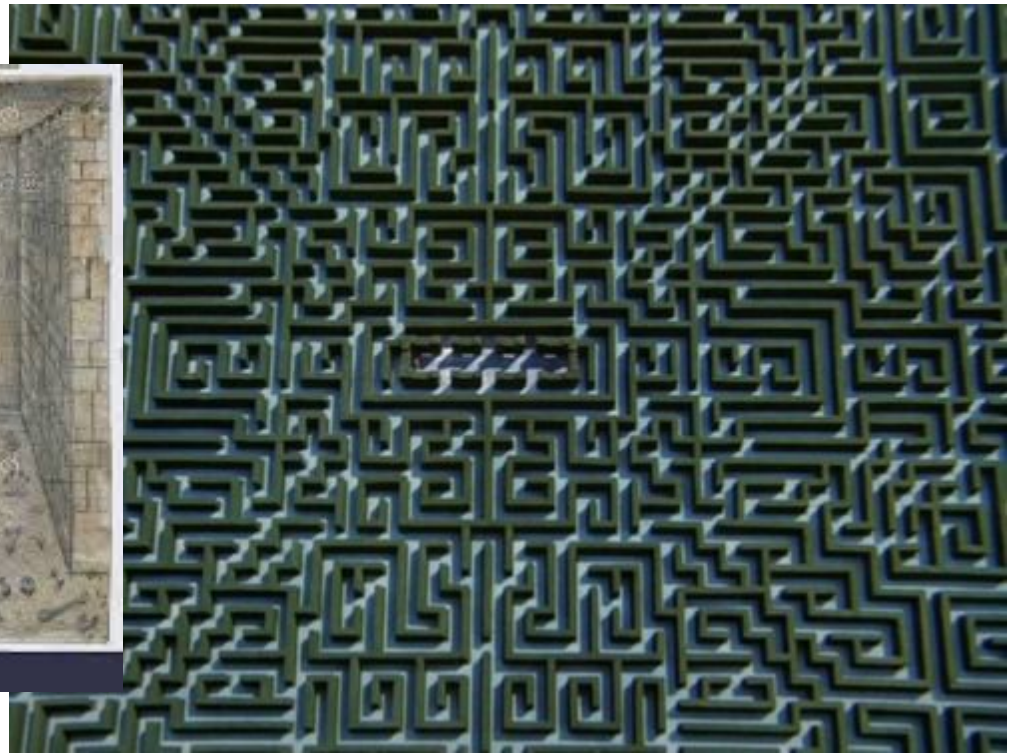
- Graph representations
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
- Two applications on DFS:
  - Topological sort
  - Strongly connected components

Outline

- Graph representations
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
    - Parenthesis theorem
    - Edge classification
    - White-path theorem
- Two applications on DFS:
  - Topological sort
  - Strongly connected components

Depth-First Search

- Recursively explore the graph starting from the current position
  - Walk as far as possible
  - Back track when there is no further vertex



Depth-First Search Initialization

```
For each u ∈ V[G]
    Mark u as not-discovered
    π[v] ← NIL
time ← 0
For each vertex u ∈ V[G]
    If u is not-discovered
        DFS-VISIT(u)
```

- Each vertex is marked as
  - White: **not-discovered** by the algorithm
  - Gray: **discovered** by the algorithm
  - Black: **finished**; all its neighbors are discovered

- For each of the vertices v, we take information:
  - $d[v]$: **discover time**, timestamp when v is first discovered
  - $f[v]$: **finish time**, timestamp when the search finishes examining v's adjacency list
    - $d[v] < f[v] \leq 2|V|$
  - $\pi[v]$: the predecessor of v when we traverse the graph

Depth-First Search

```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```

Depth-First Search

```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```

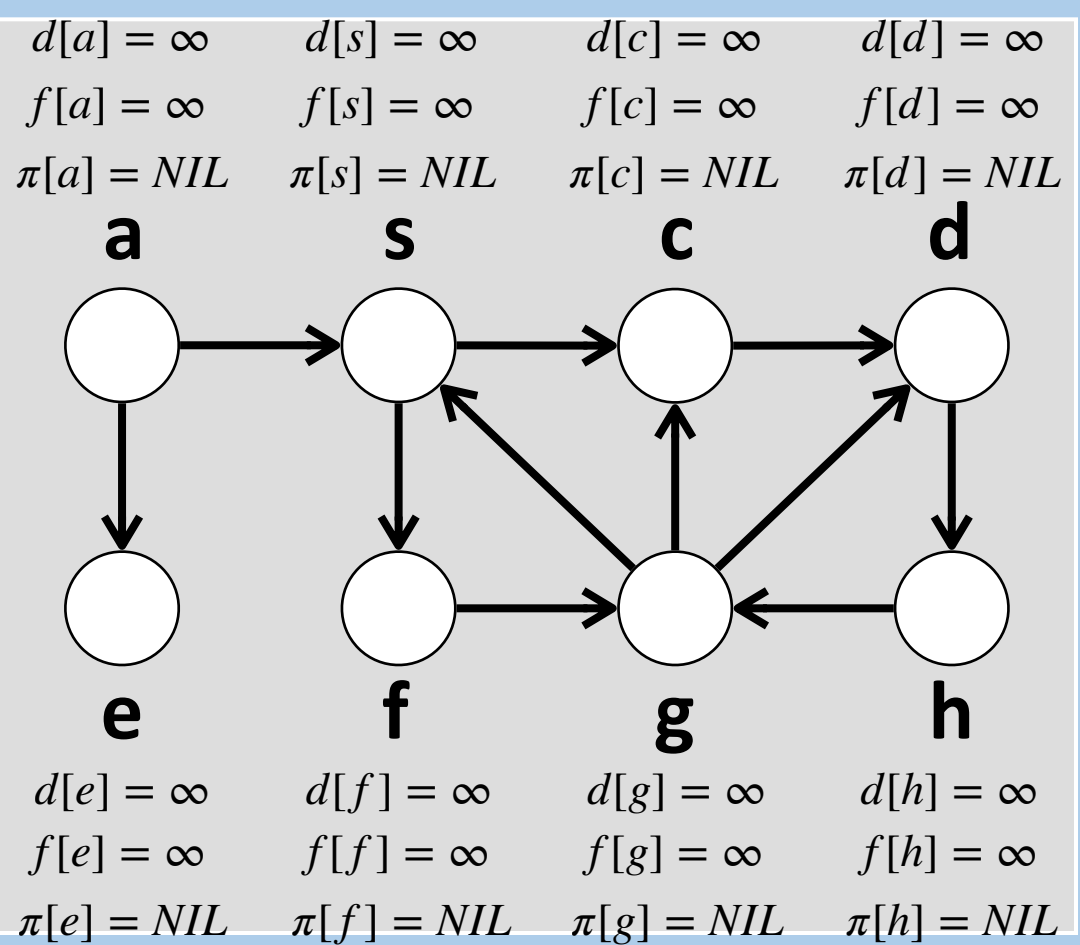
Time stamp:

At any time t, a vertex is discovered or a vertex is finished

Not-discovered
Discovered
Finished

Depth-First Search

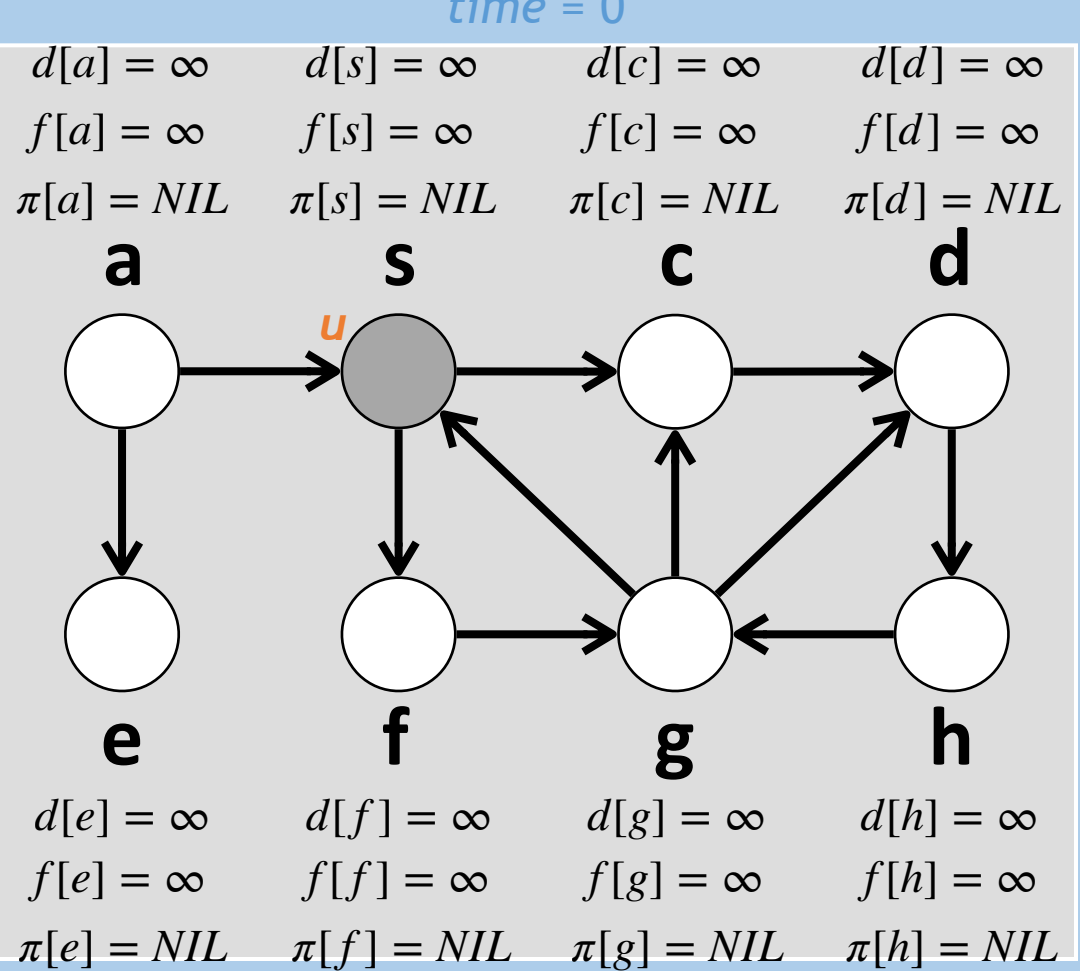
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

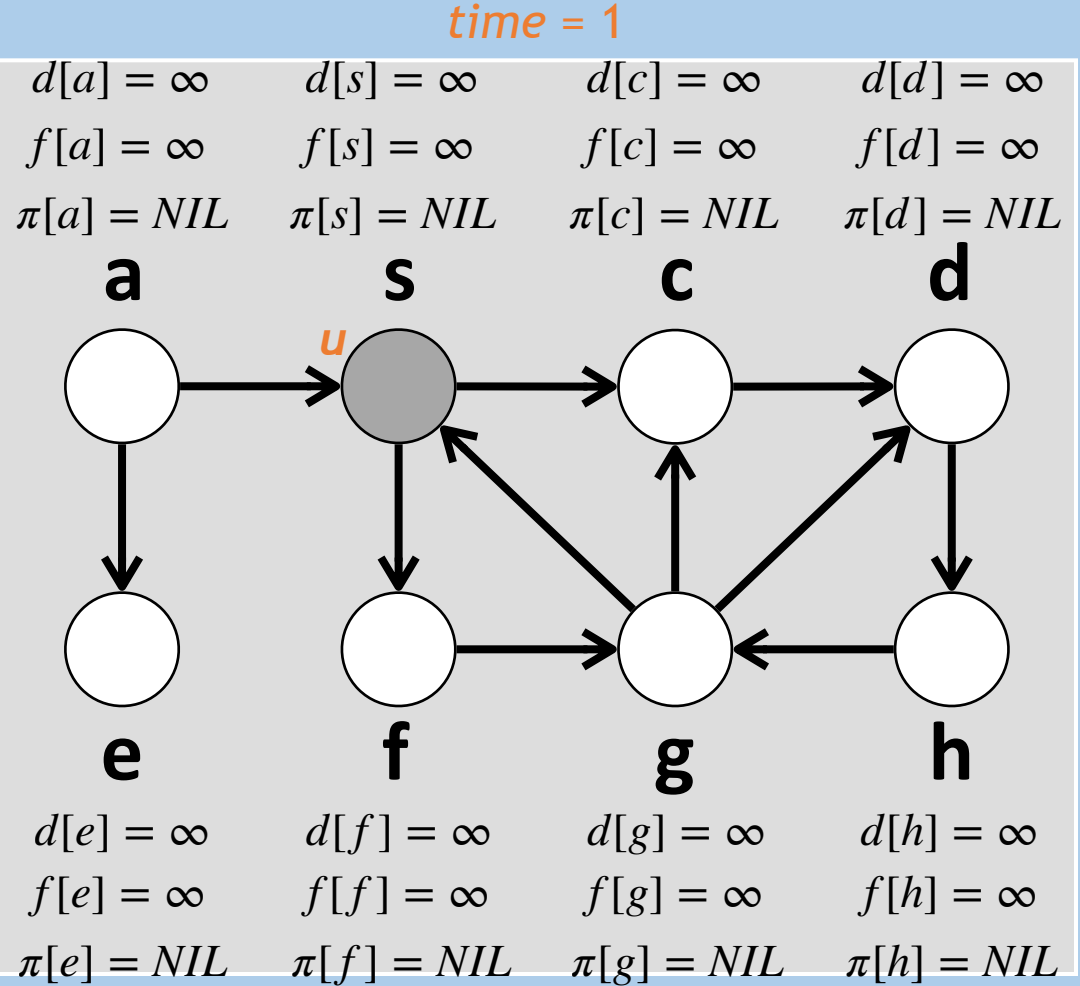
```
Procedure DFS-VISIT(u)
  Mark u as discovered
  time ← time + 1
  d[u] ← time
  For each neighbor v of u
     If v is not-discovered
        π[v] ← u
        DFS-VISIT(v)
  Mark u as finished
  time ← time + 1
  f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

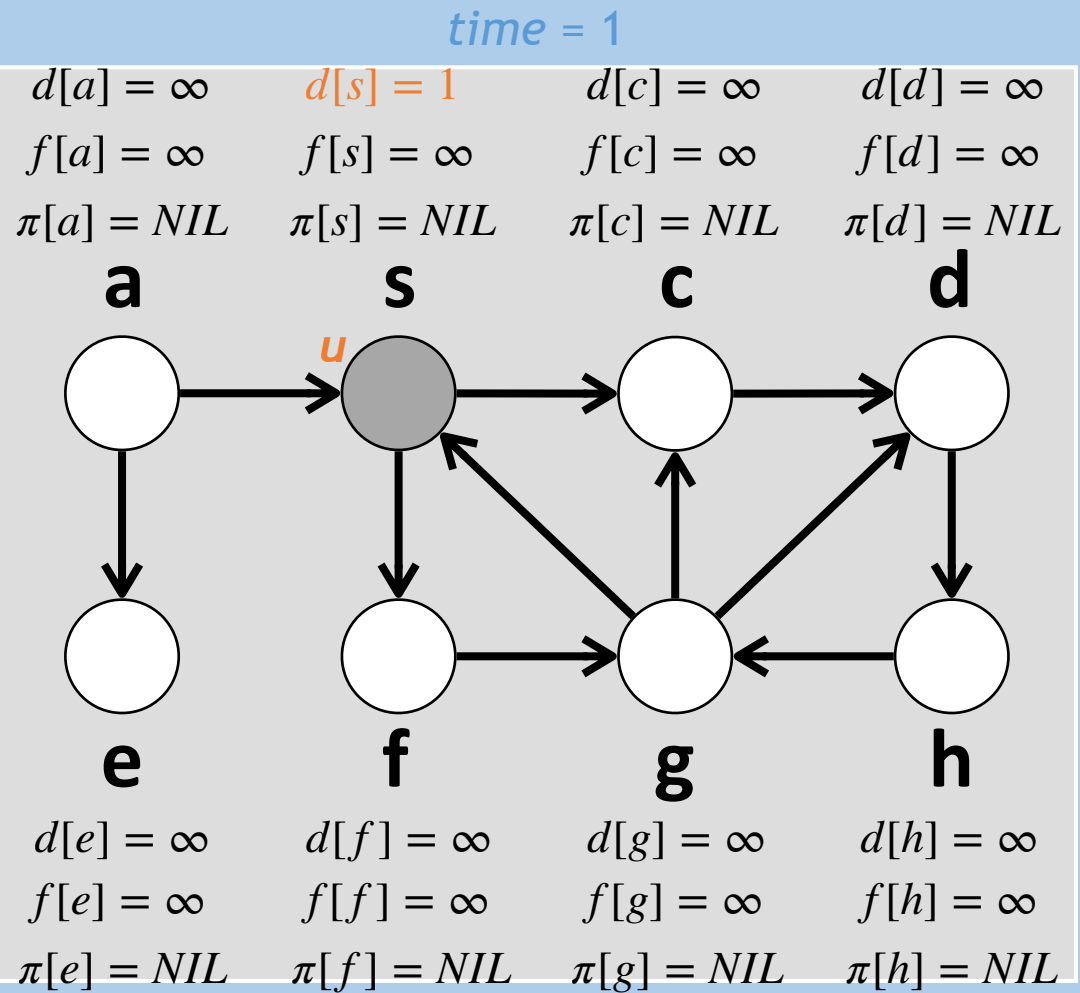
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

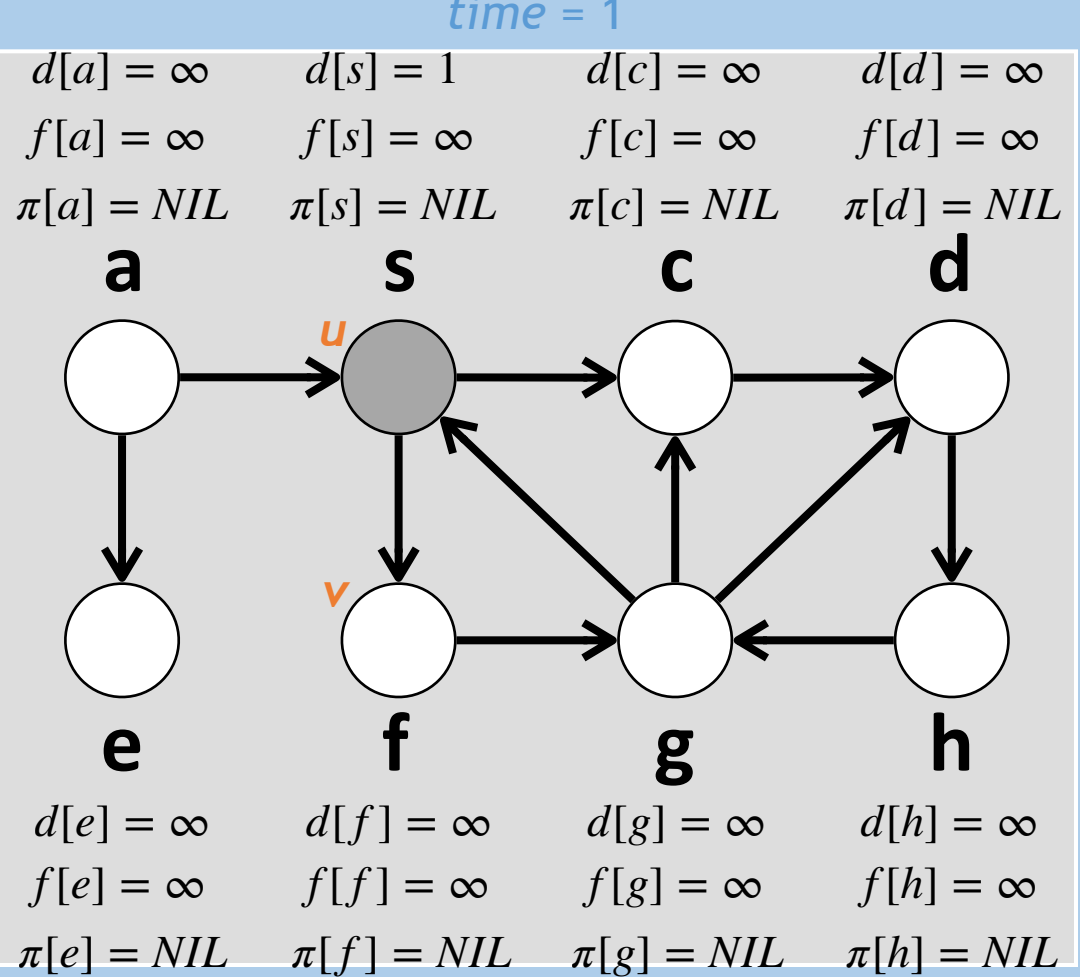
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

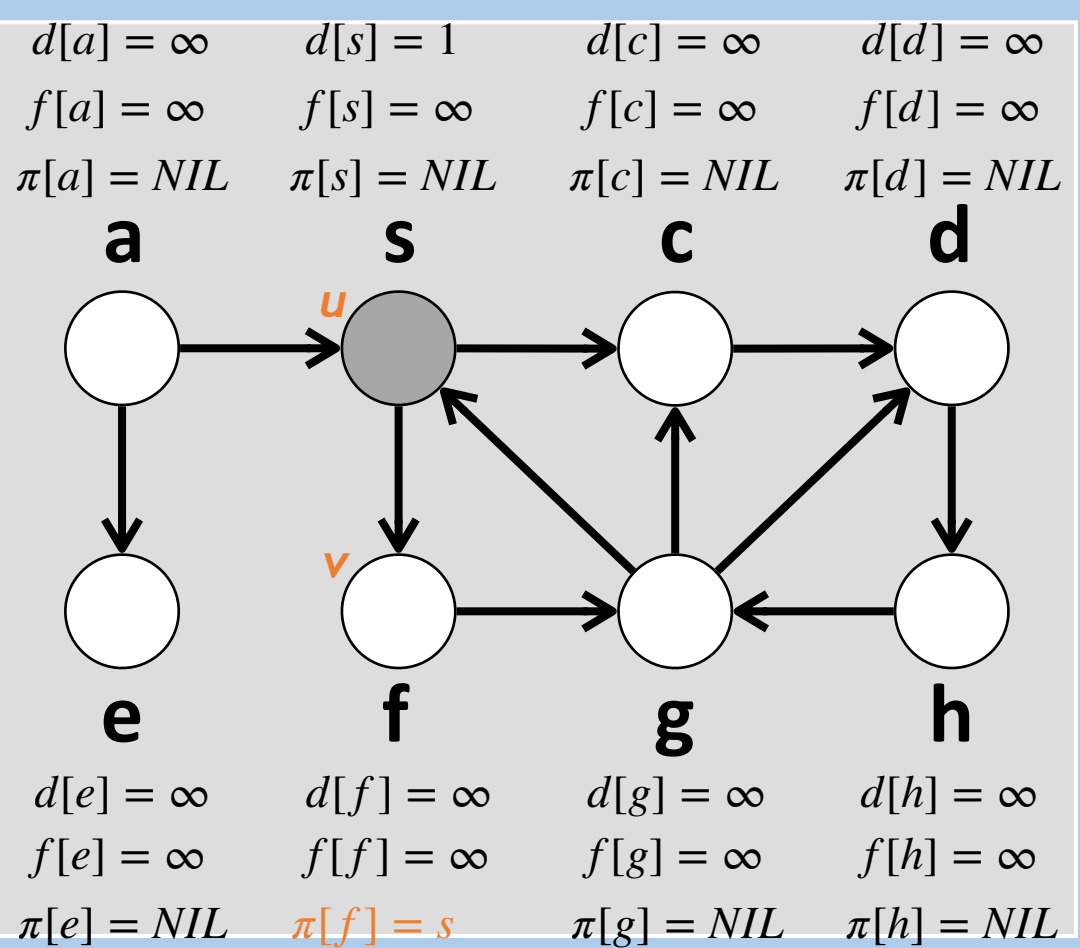
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

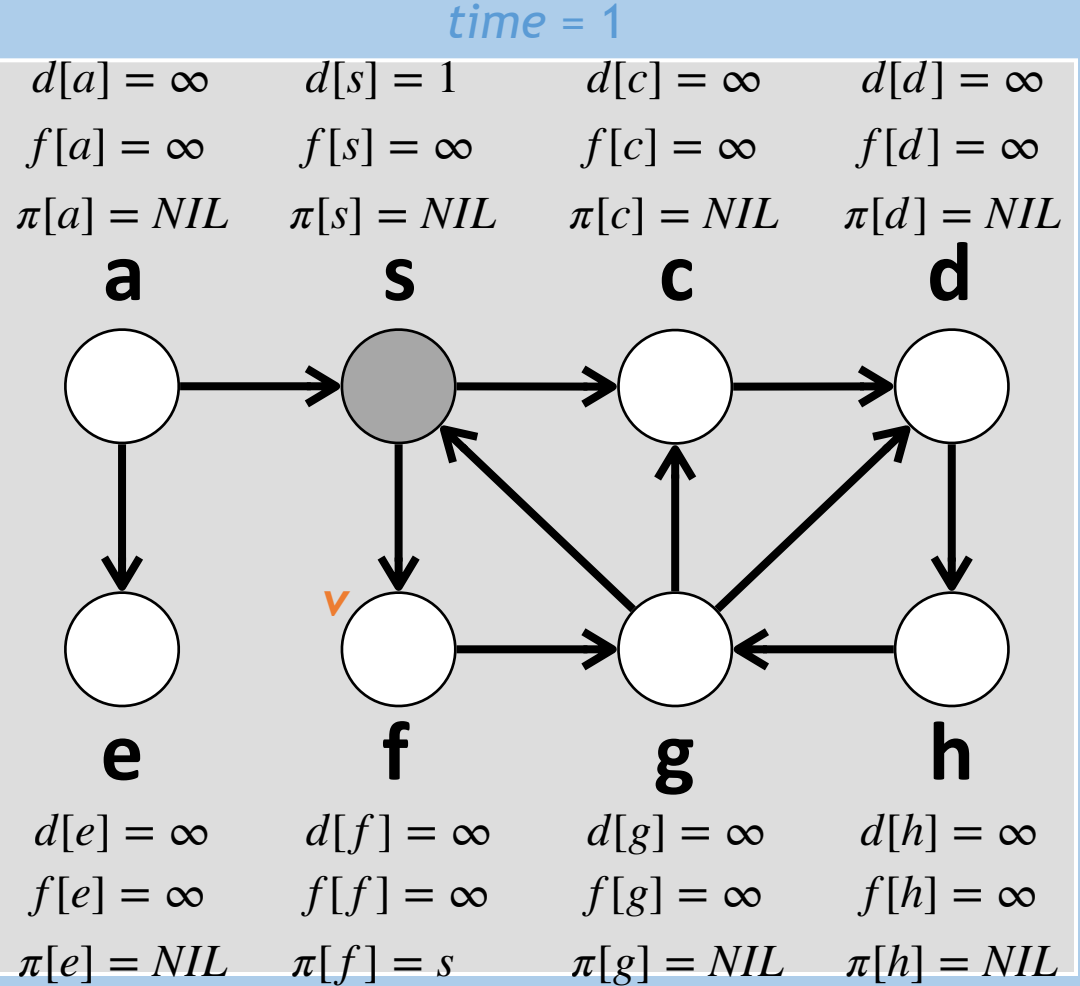
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

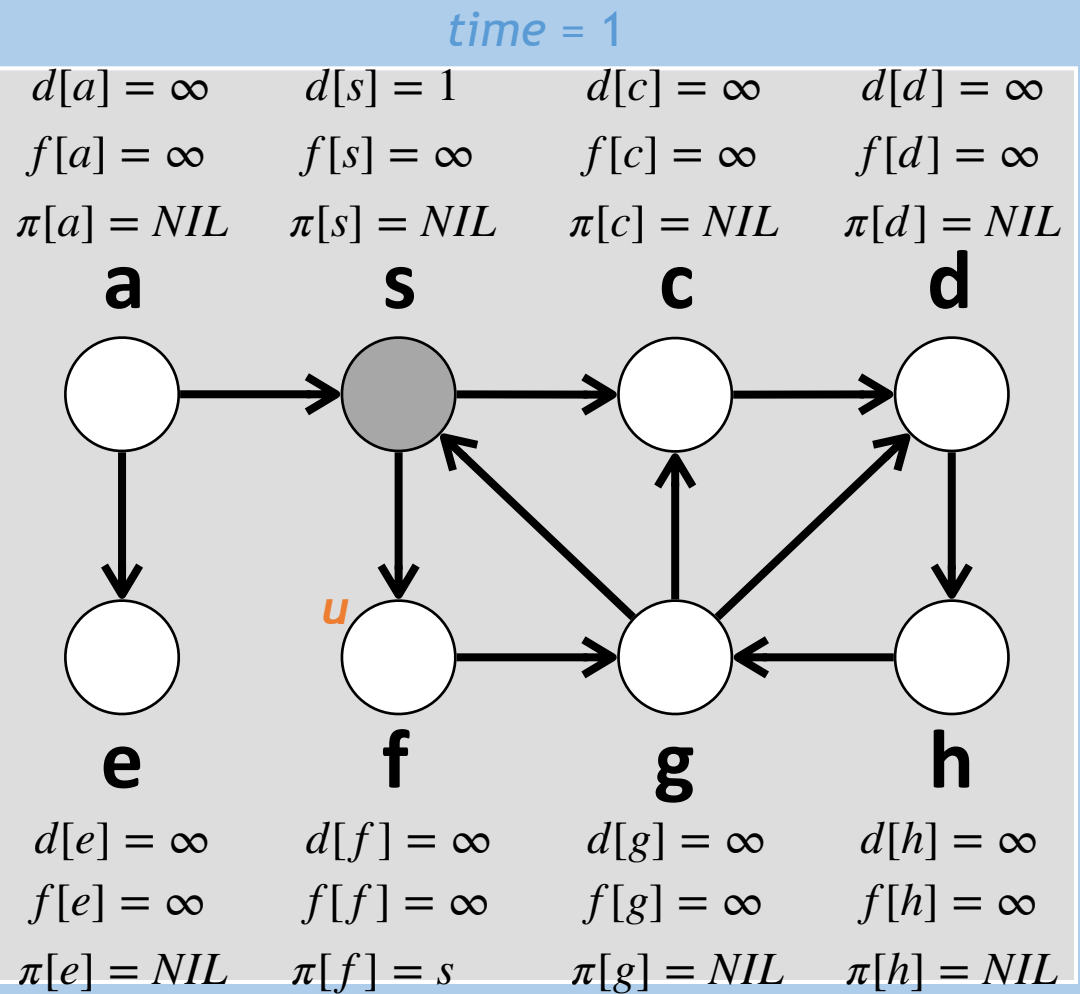
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

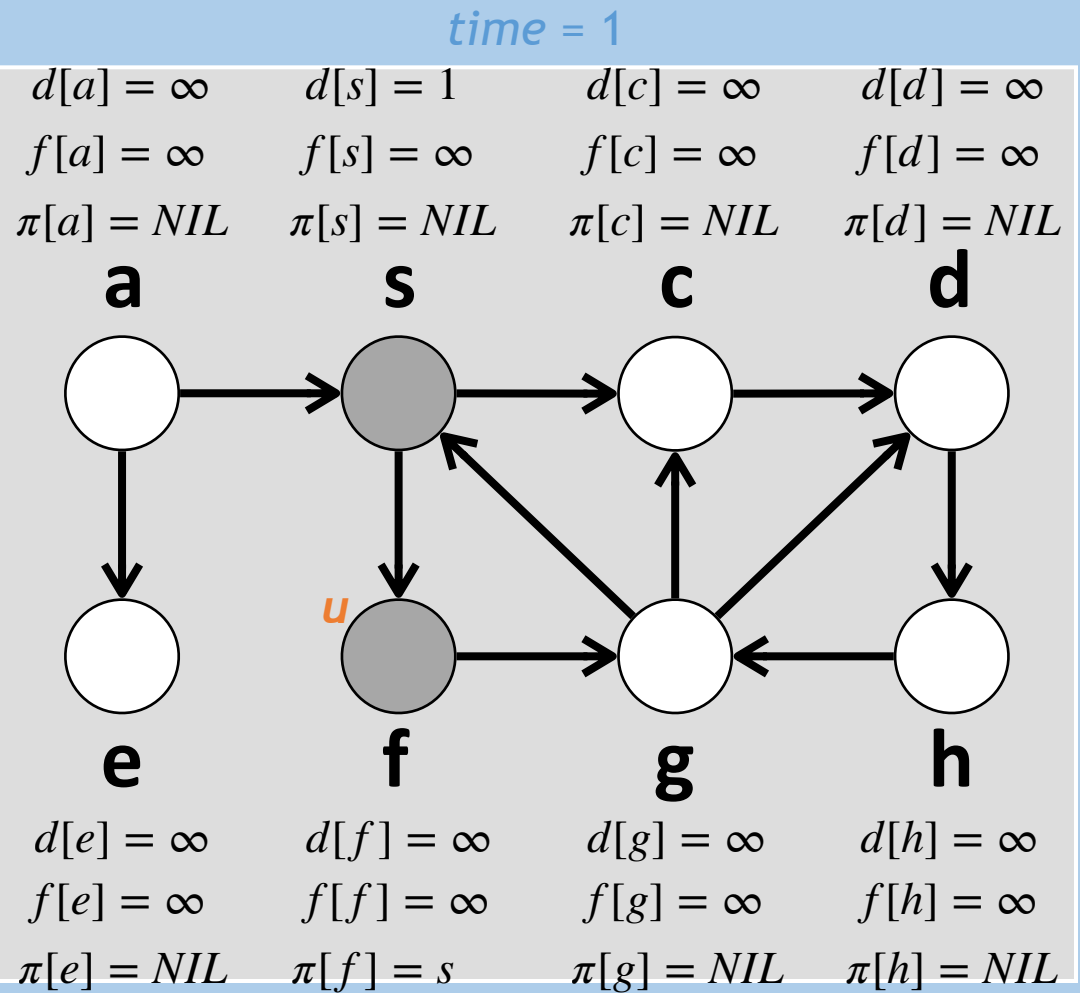
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

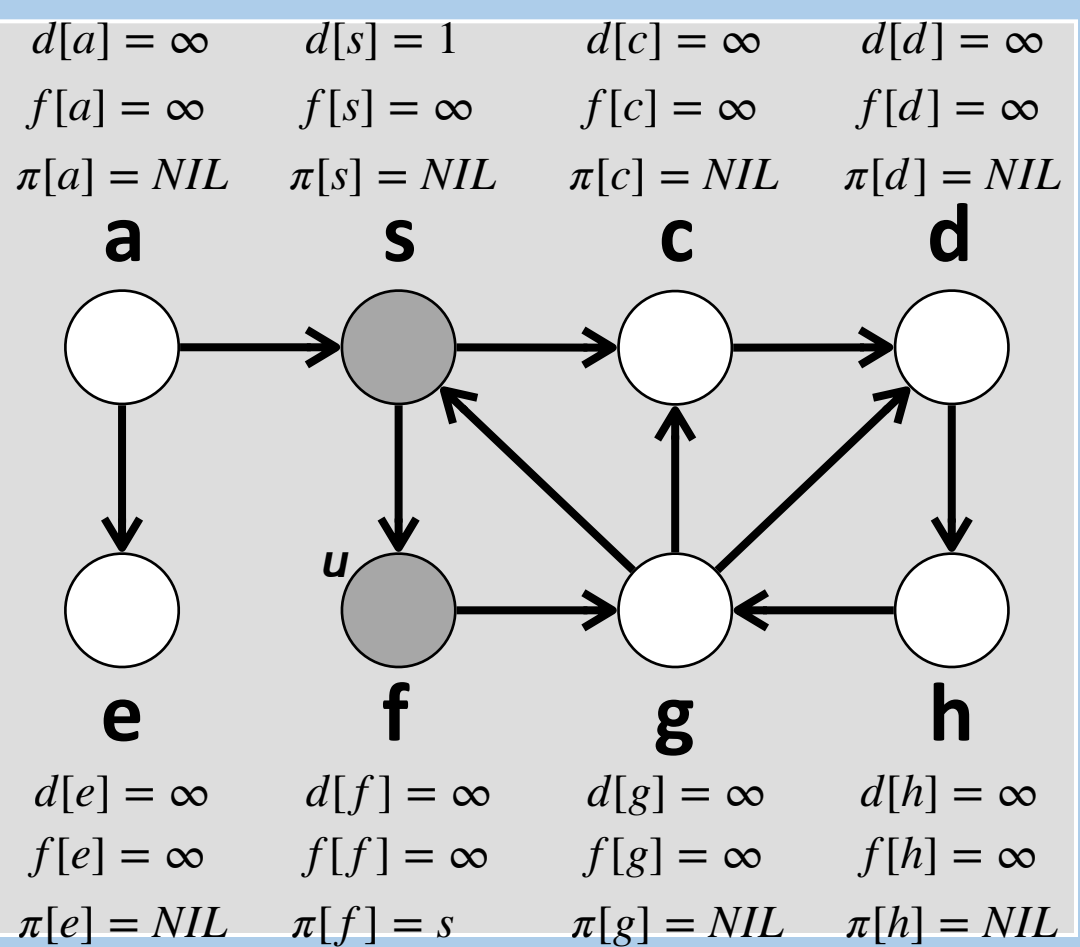
```
Procedure DFS-VISIT(u)
  Mark u as discovered
  time ← time + 1
  d[u] ← time
  For each neighbor v of u
     If v is not-discovered
        π[v] ← u
        DFS-VISIT(v)
  Mark u as finished
  time ← time + 1
  f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

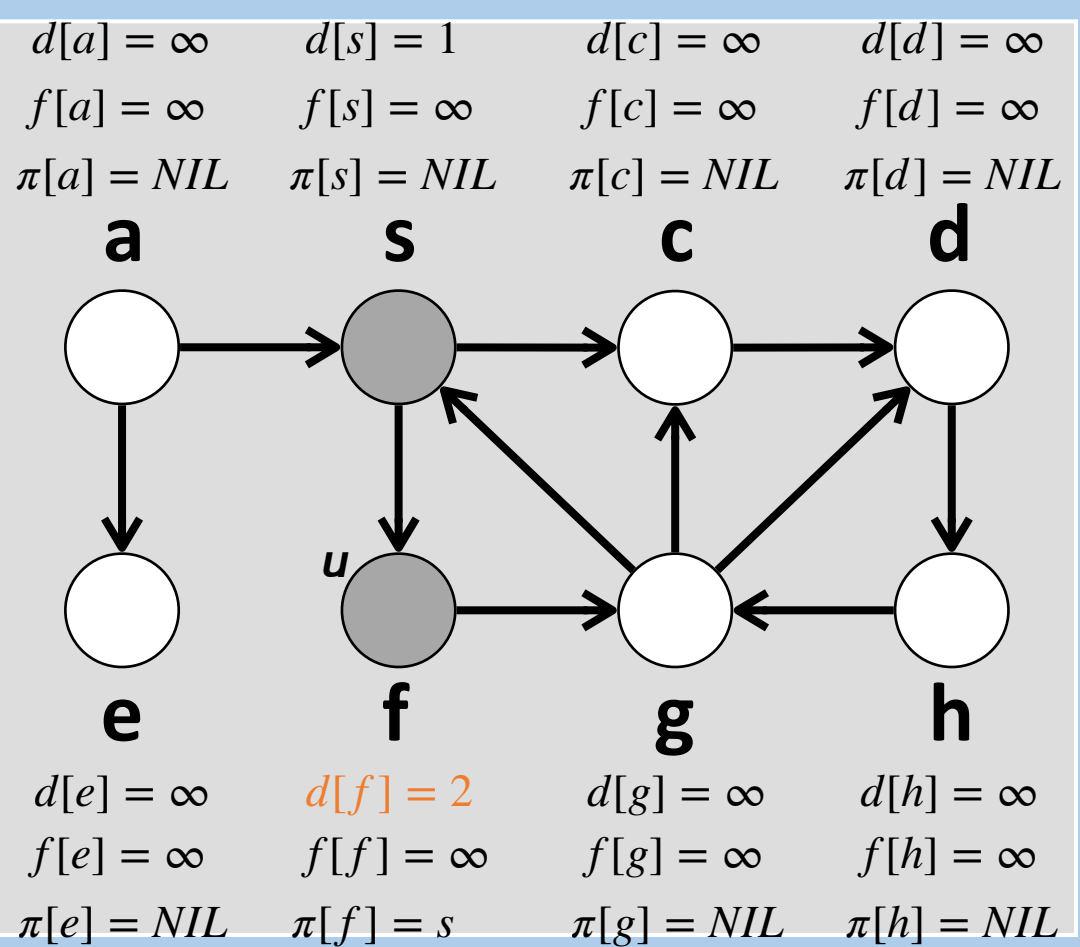
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

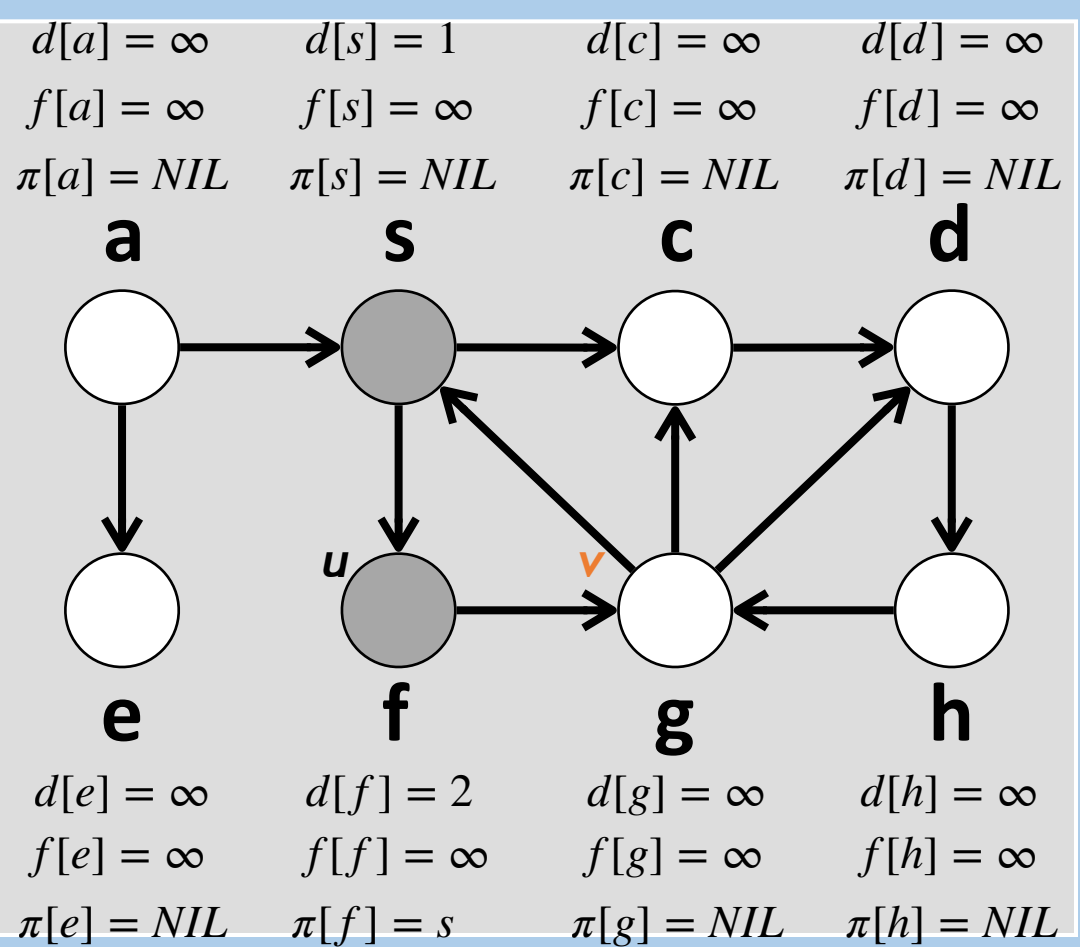
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

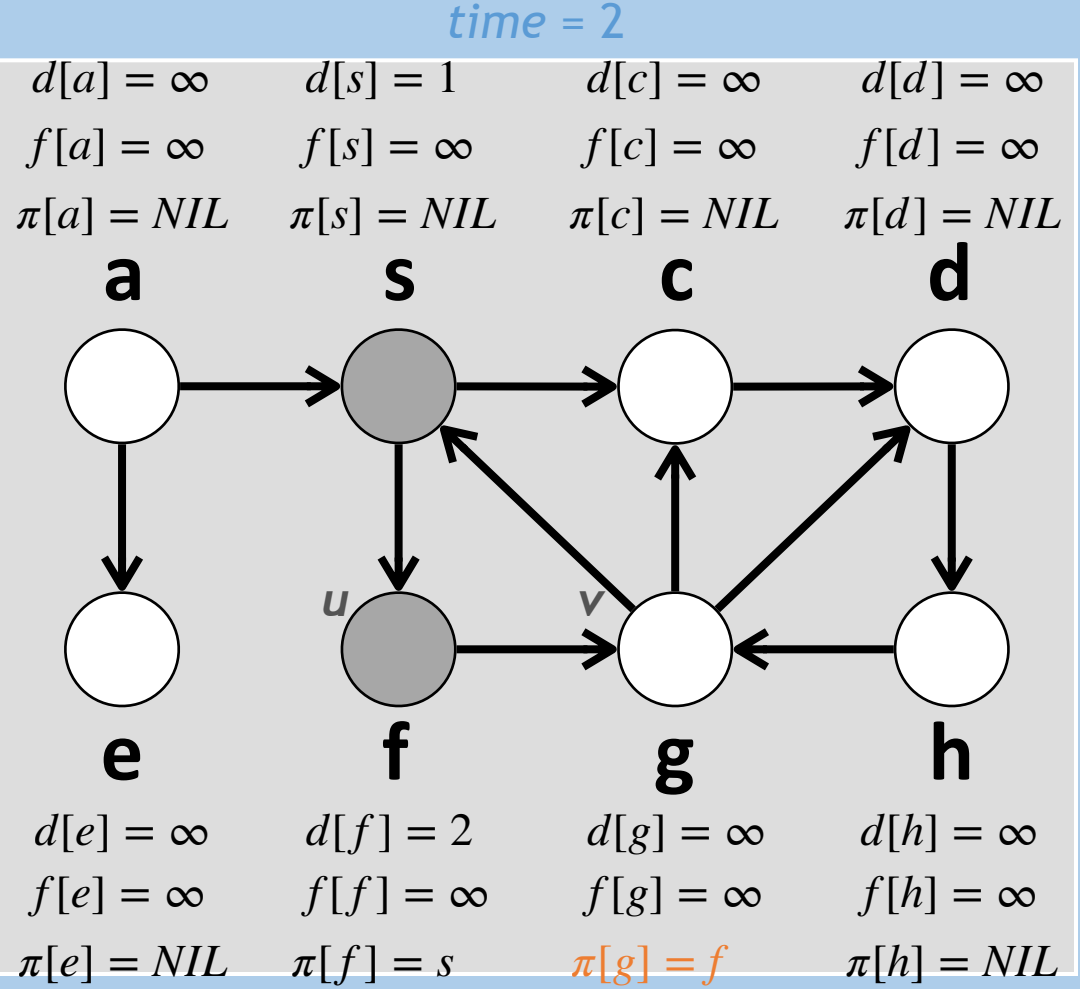
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

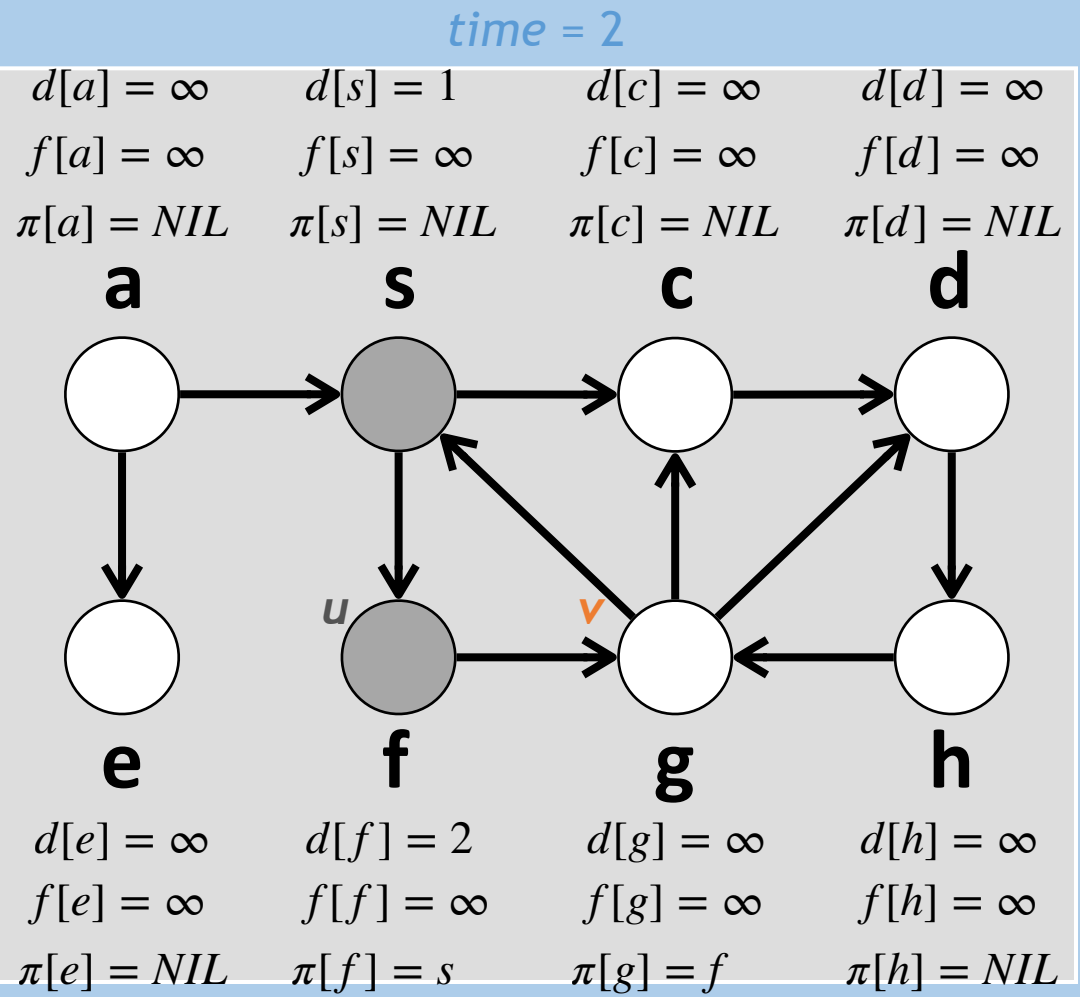
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

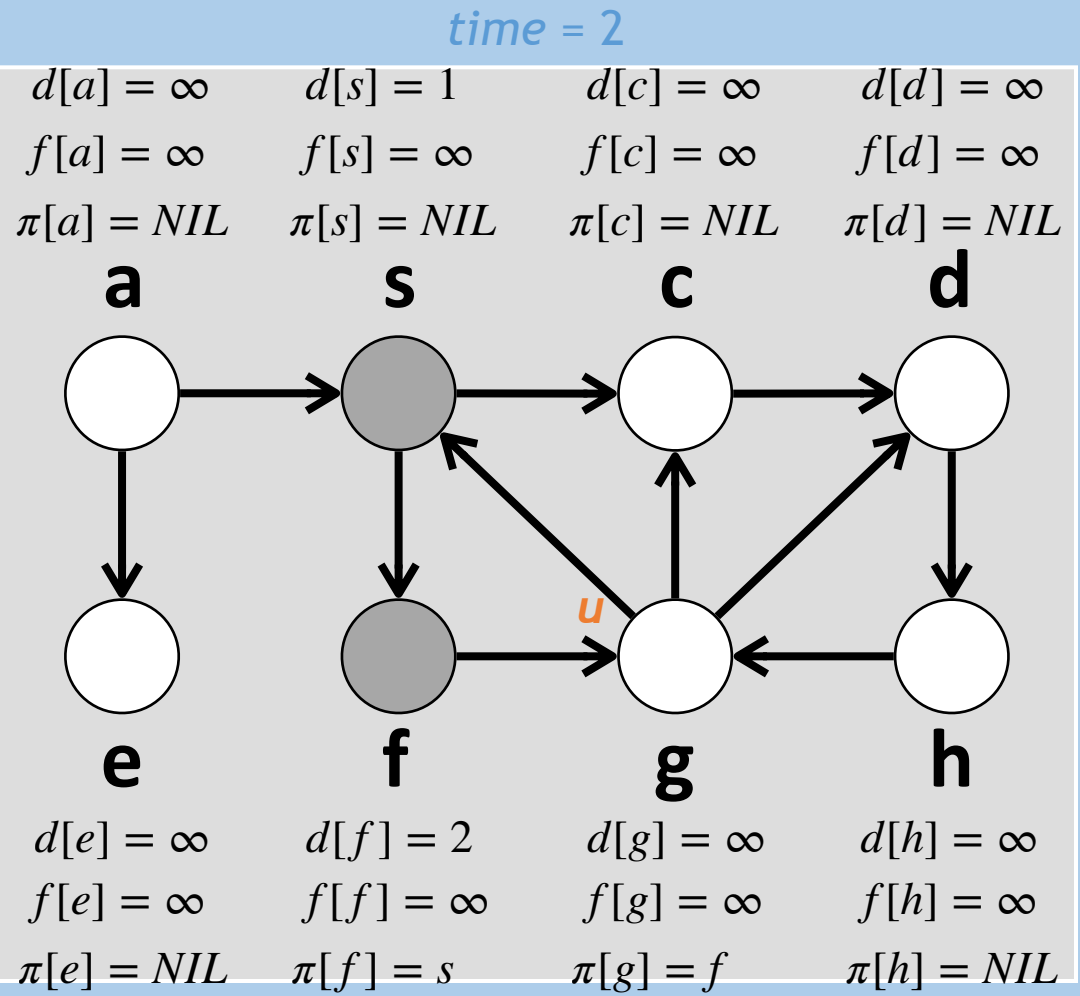
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

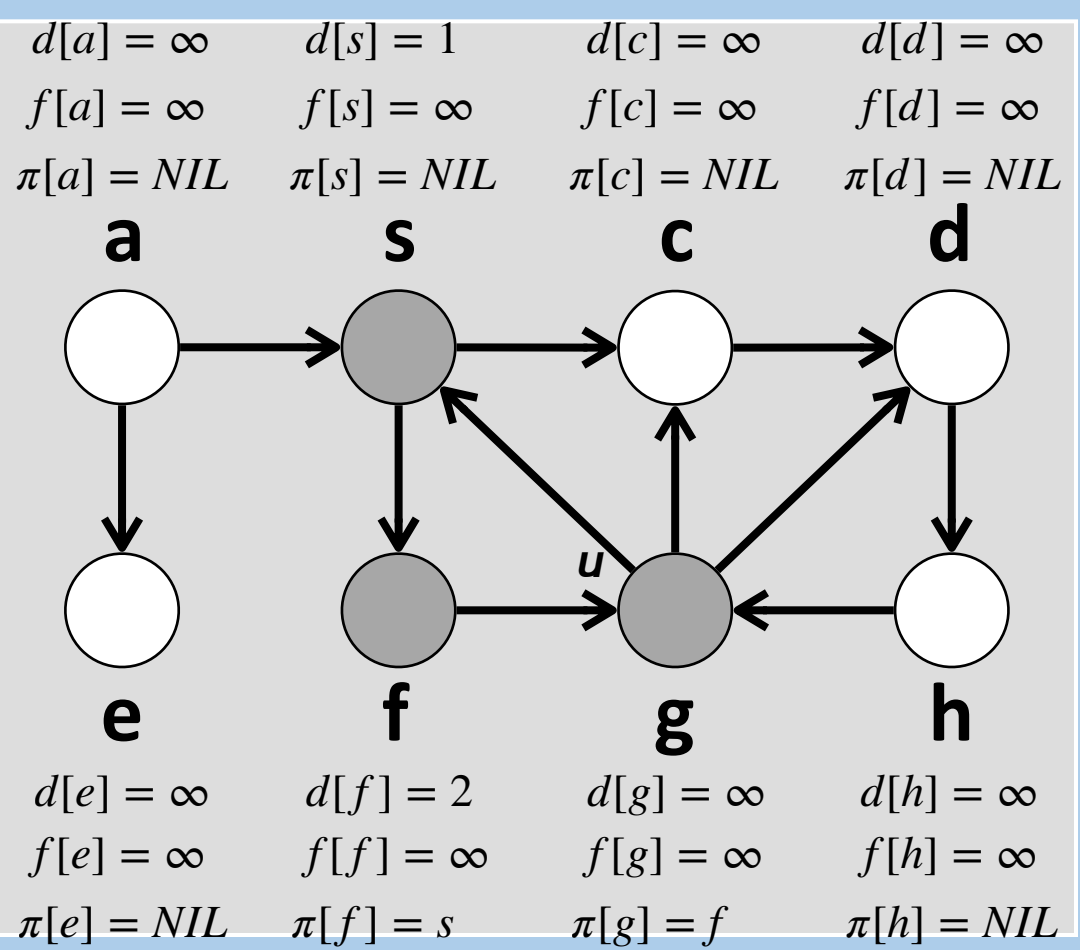
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

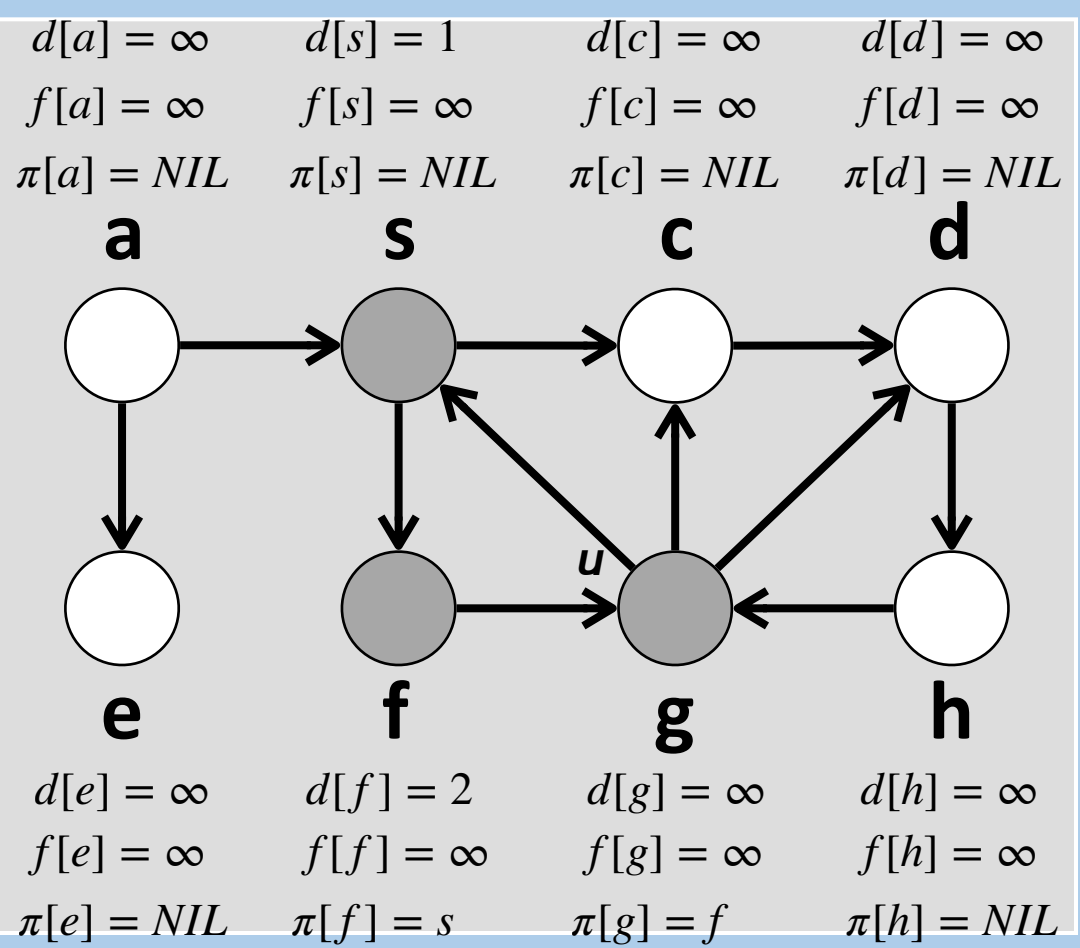
```
Procedure DFS-VISIT(u)
  Mark u as discovered
  time ← time + 1
  d[u] ← time
  For each neighbor v of u
     If v is not-discovered
        π[v] ← u
        DFS-VISIT(v)
  Mark u as finished
  time ← time + 1
  f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

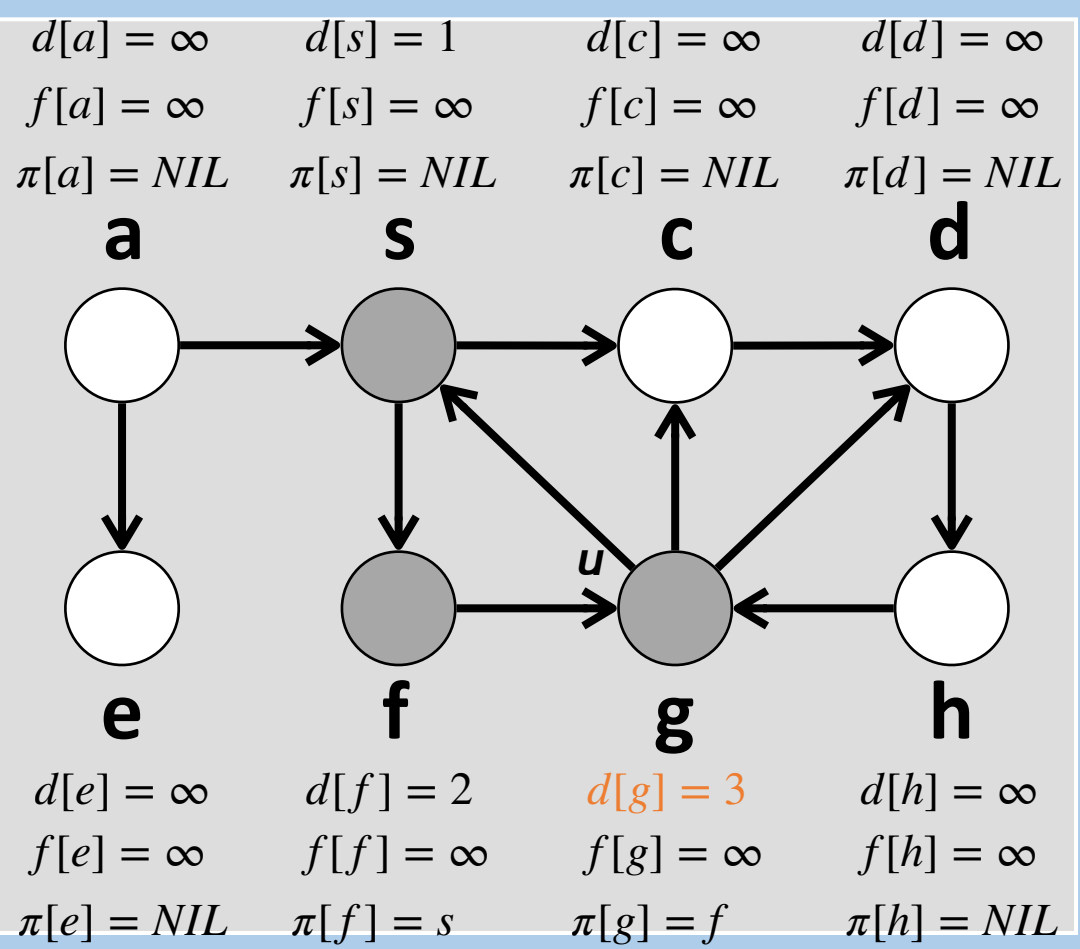
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

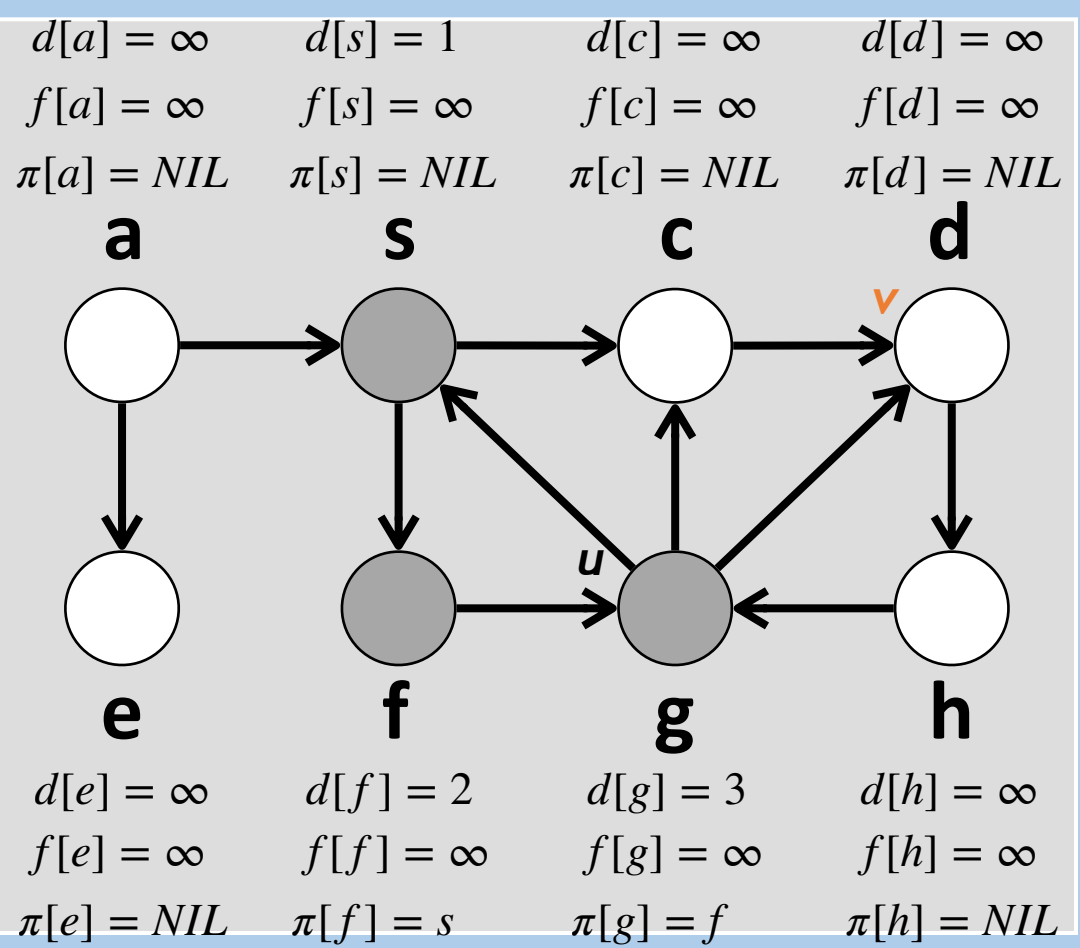
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

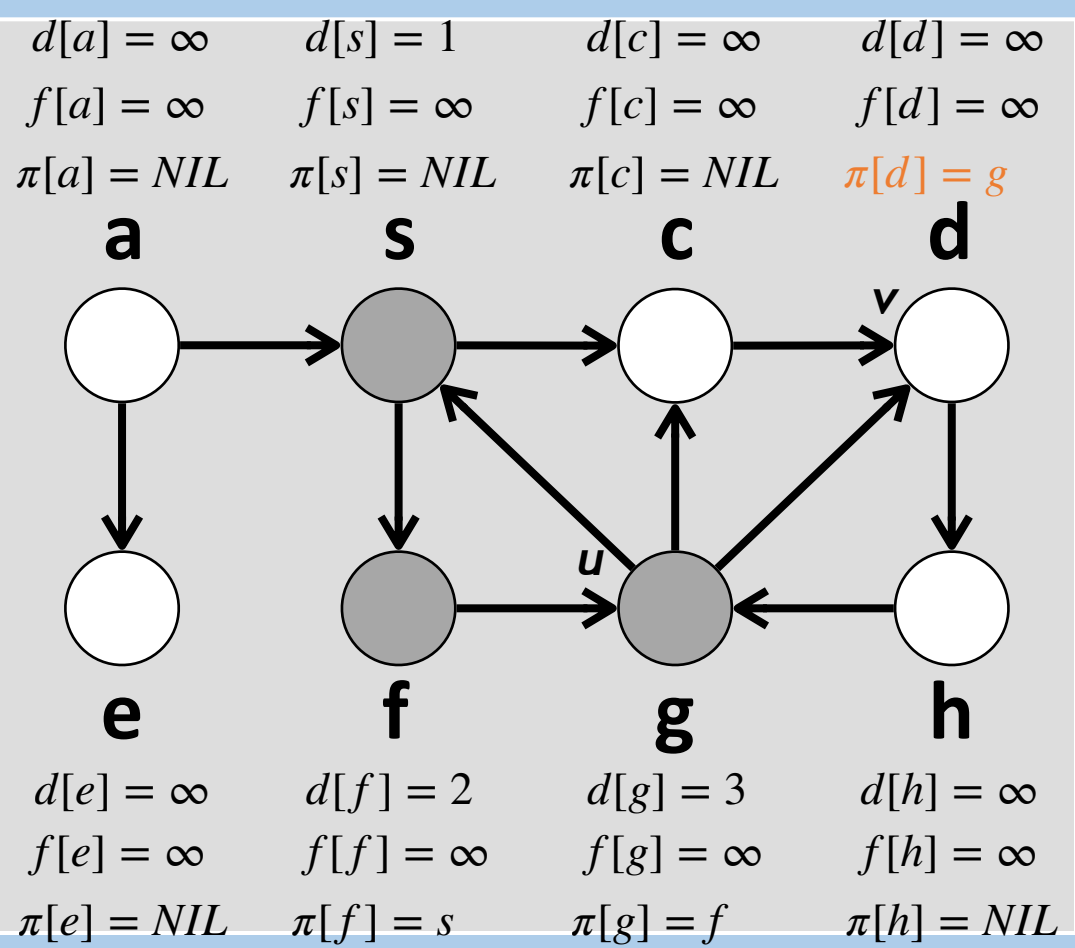
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

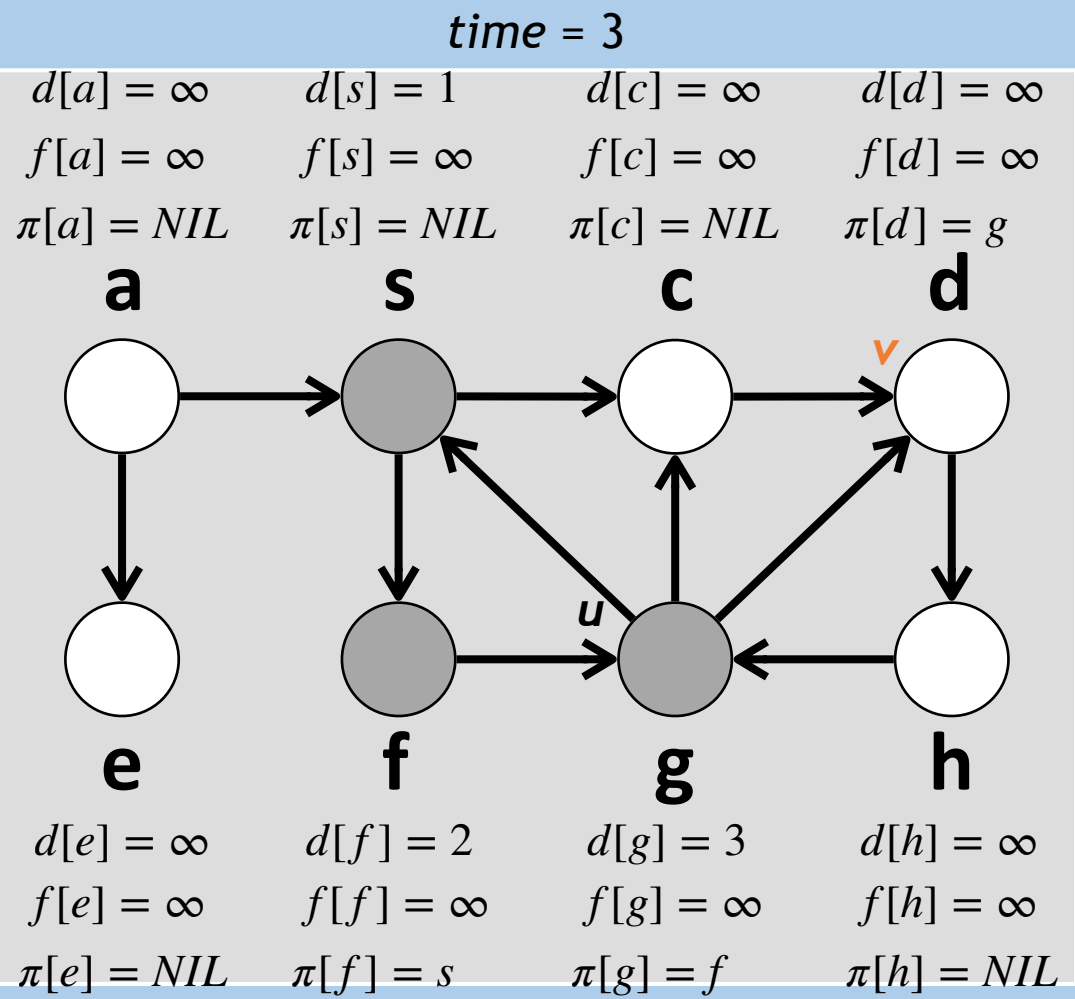
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

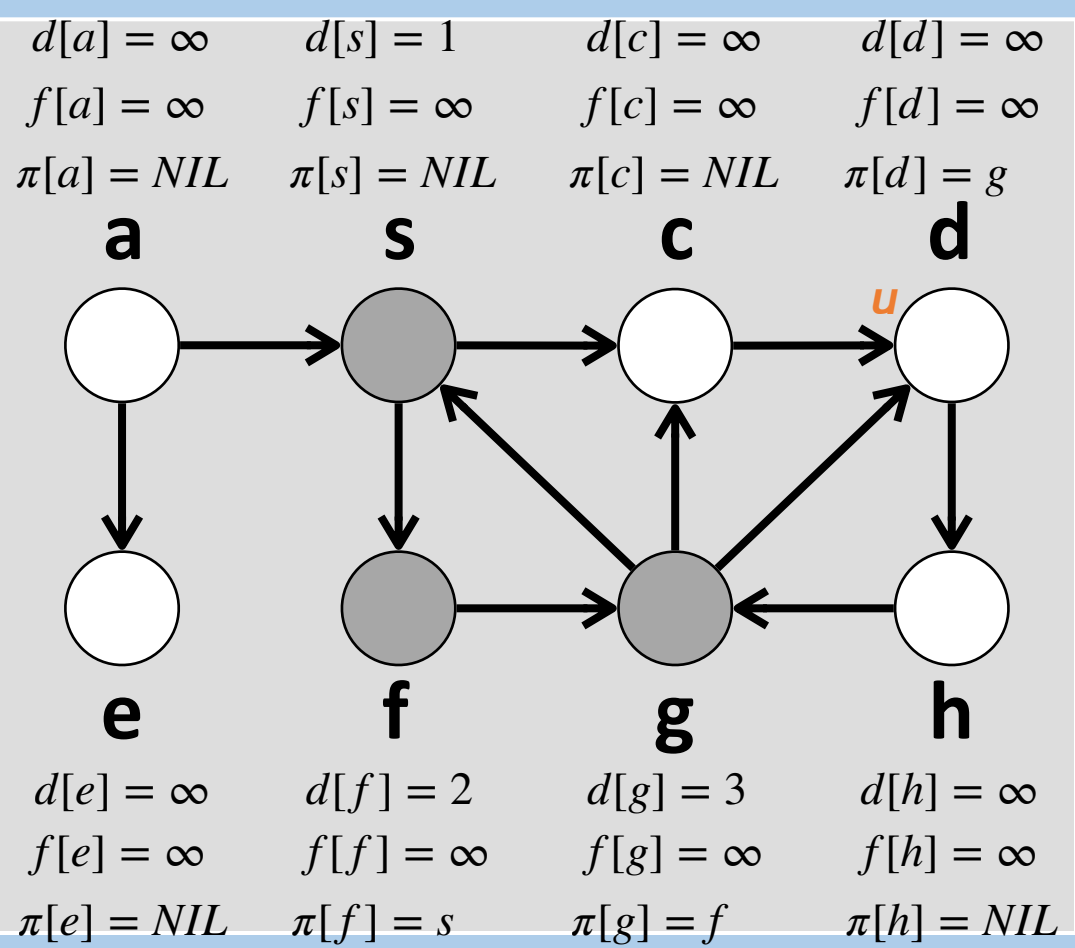
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

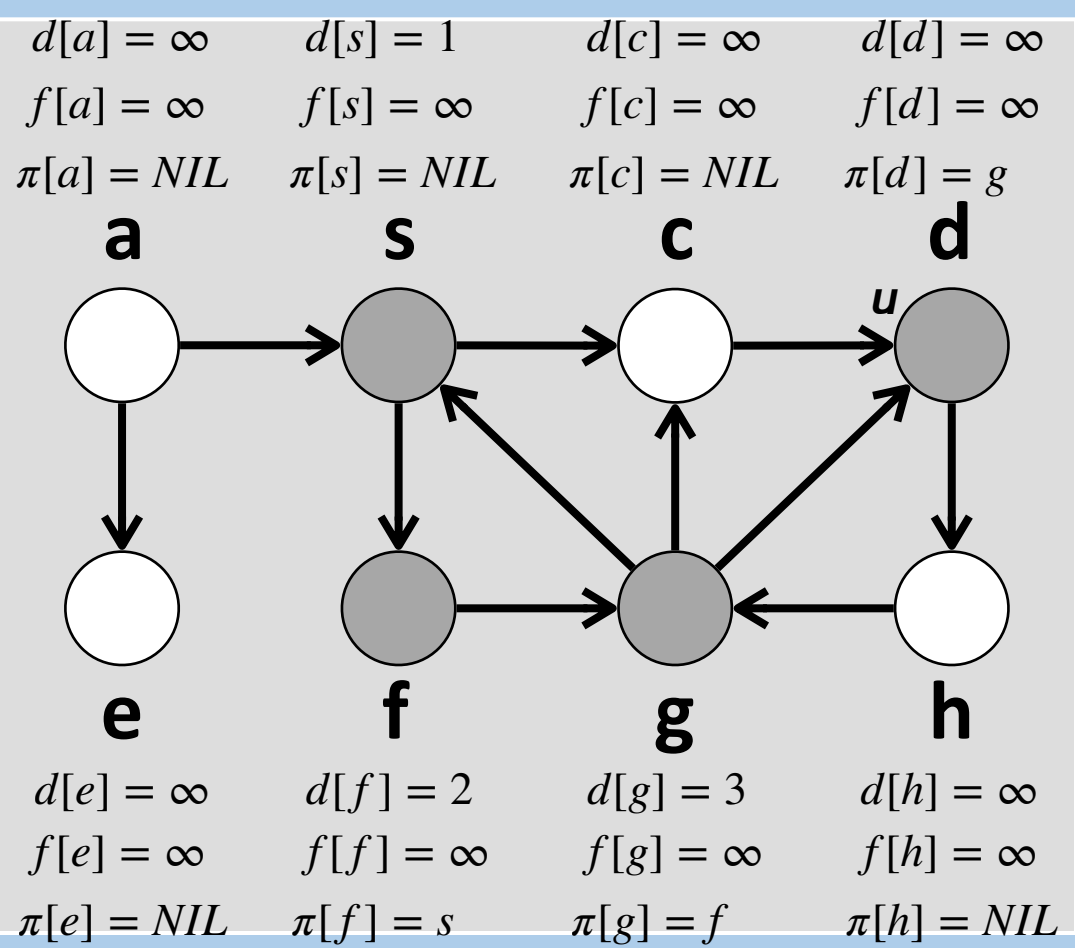
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

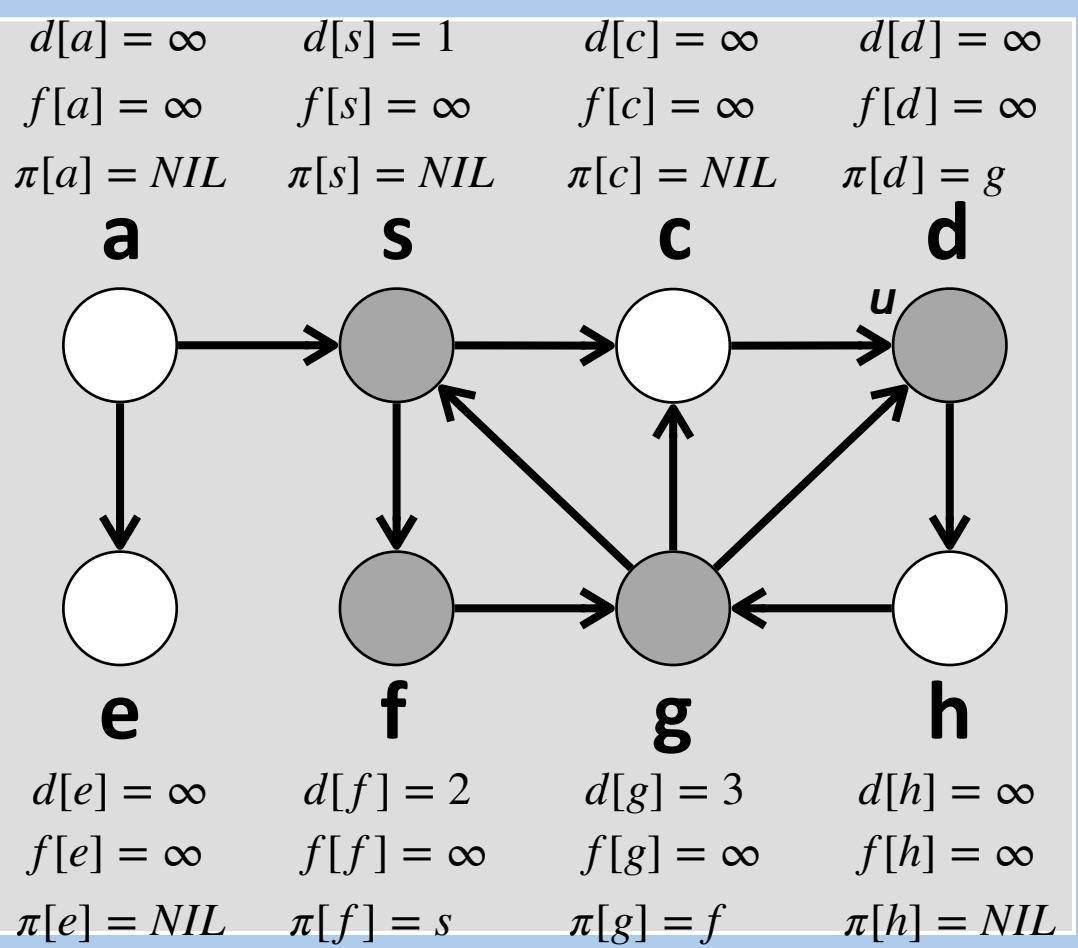
```
Procedure DFS-VISIT(u)
  Mark u as discovered
  time ← time + 1
  d[u] ← time
  For each neighbor v of u
     If v is not-discovered
        π[v] ← u
        DFS-VISIT(v)
  Mark u as finished
  time ← time + 1
  f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

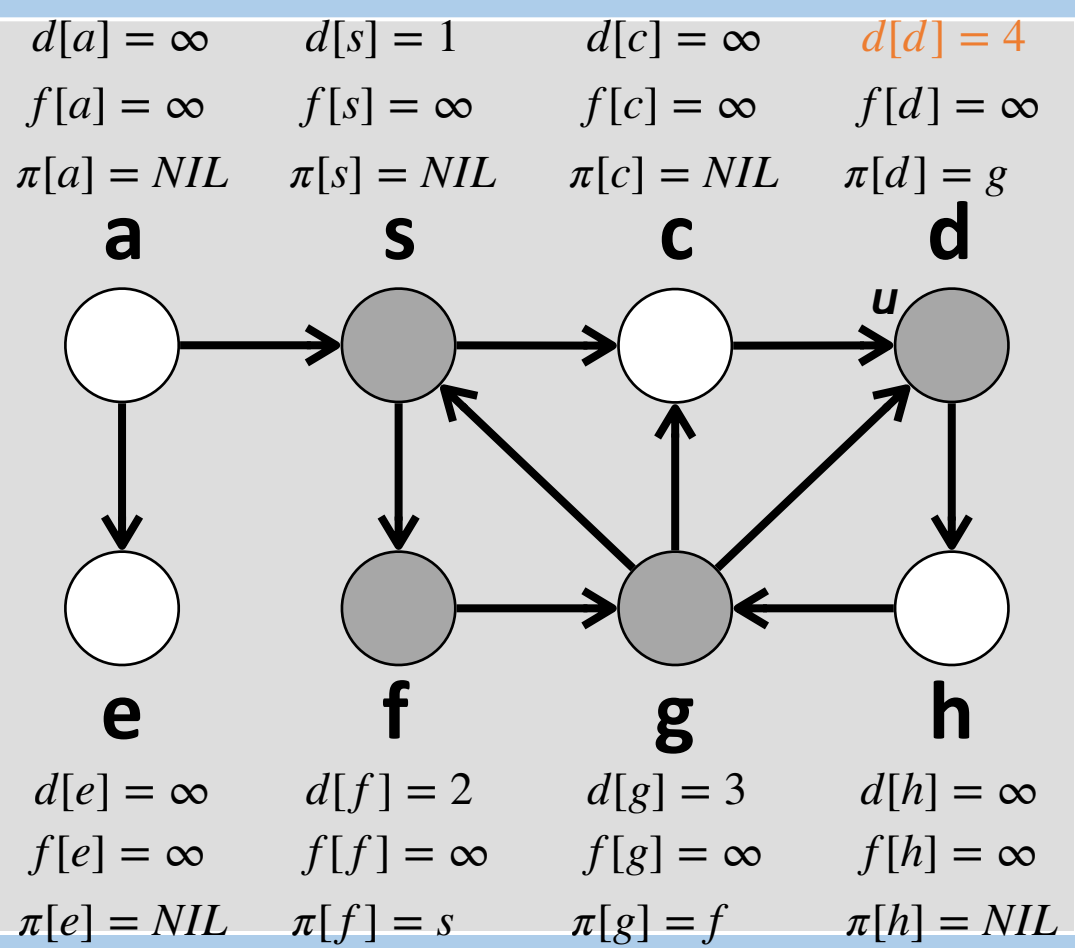
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

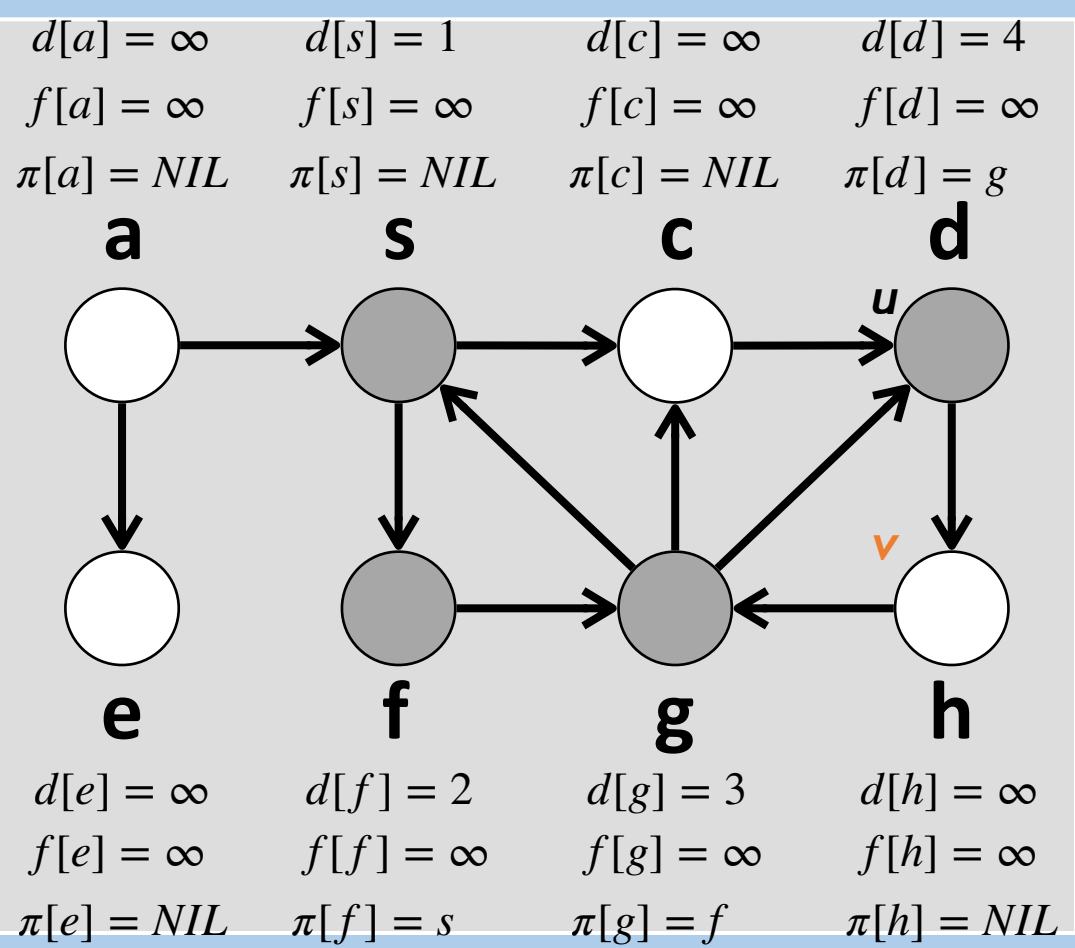
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

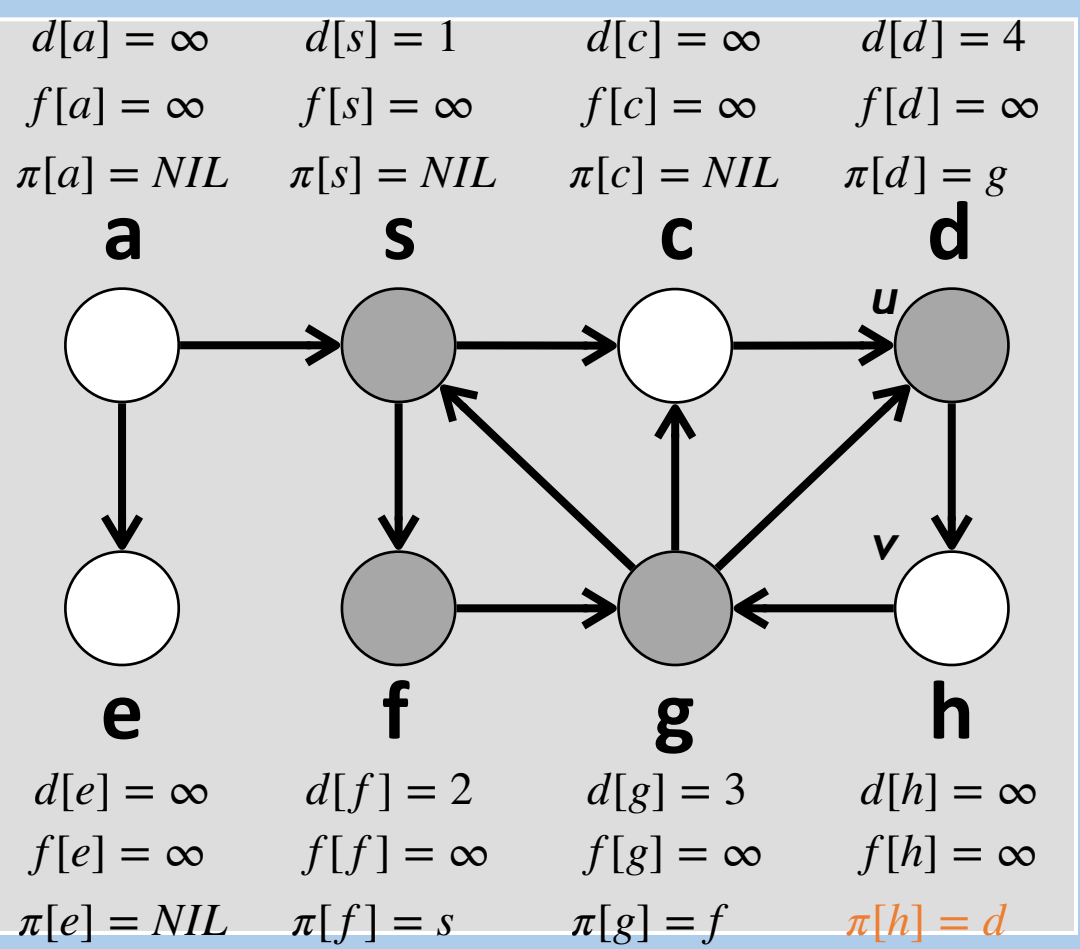
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

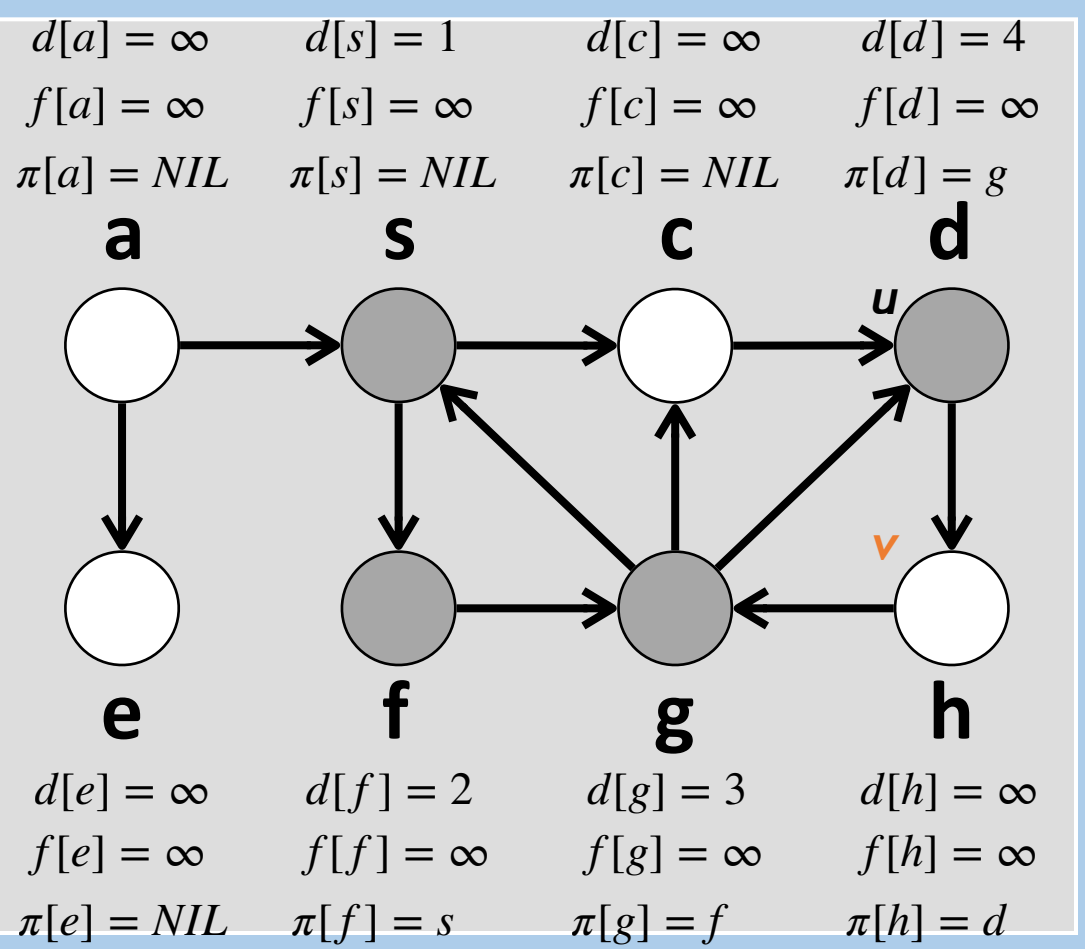
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

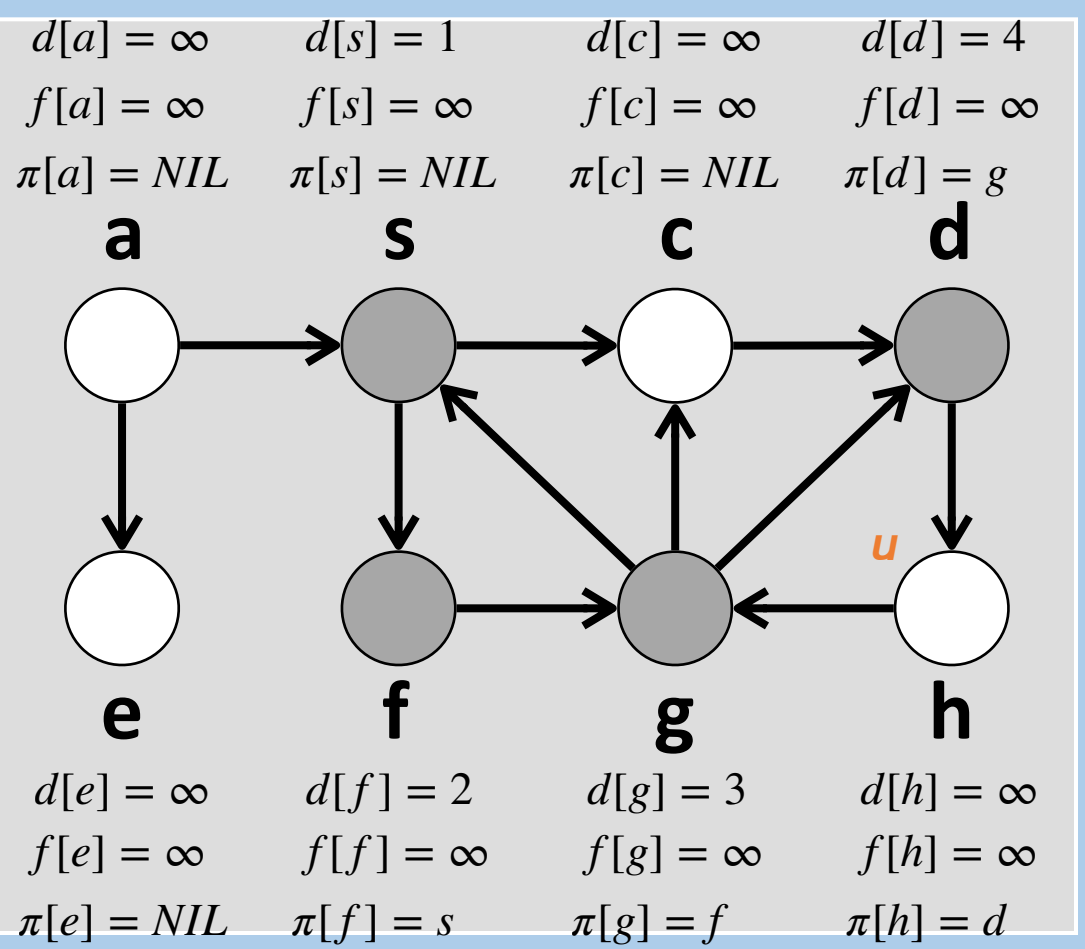
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

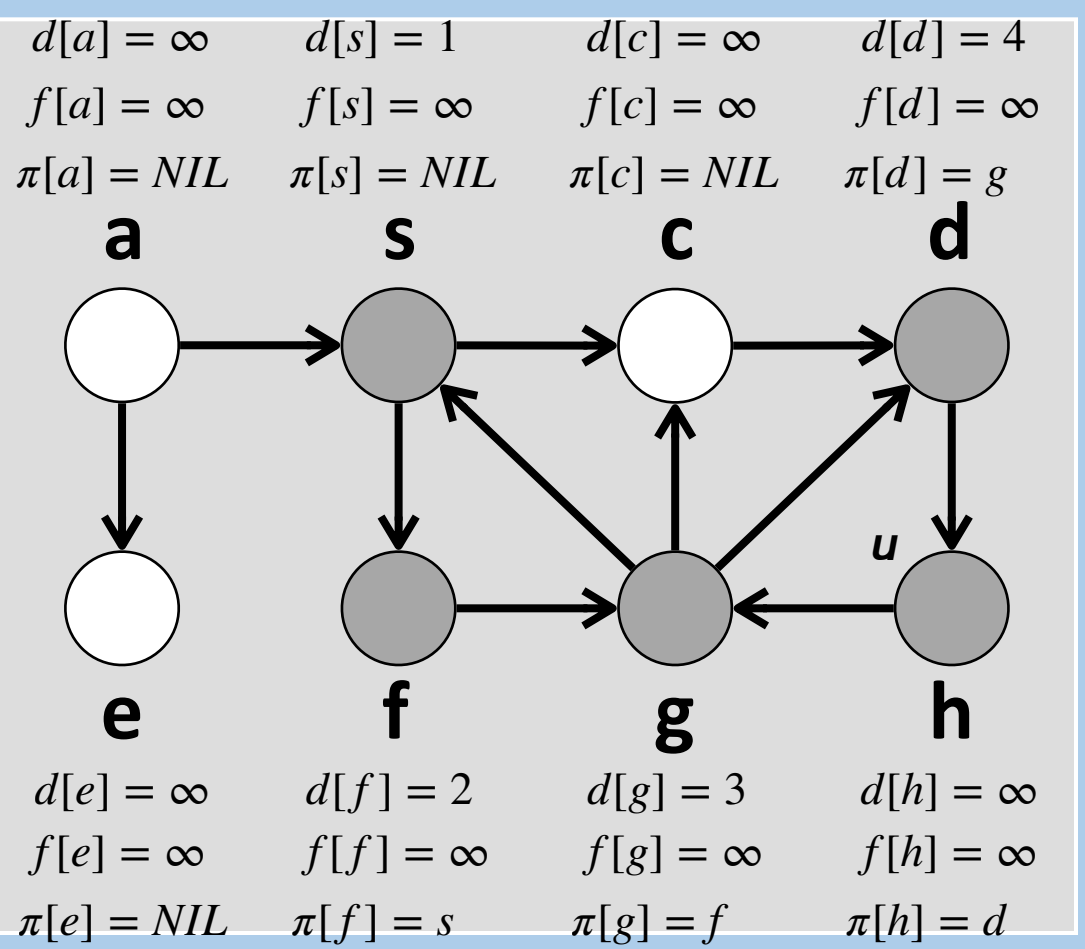
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

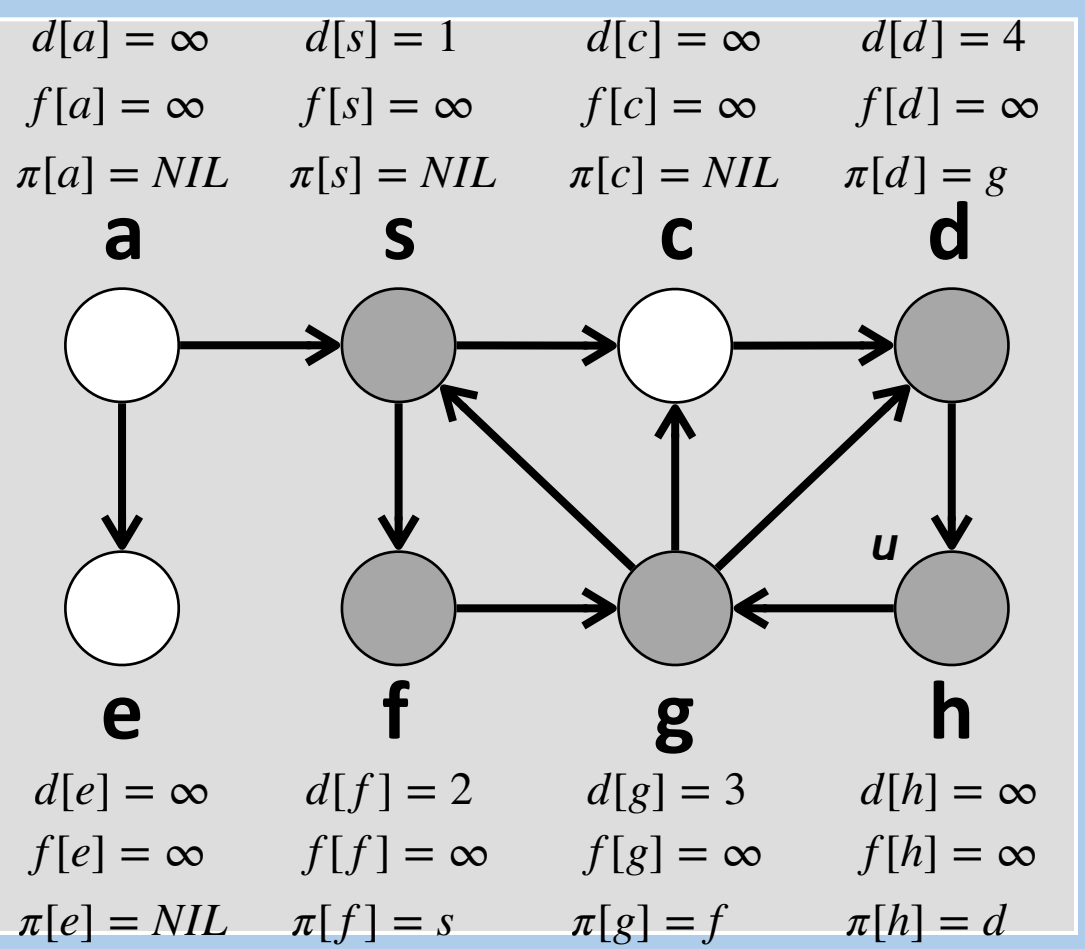
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

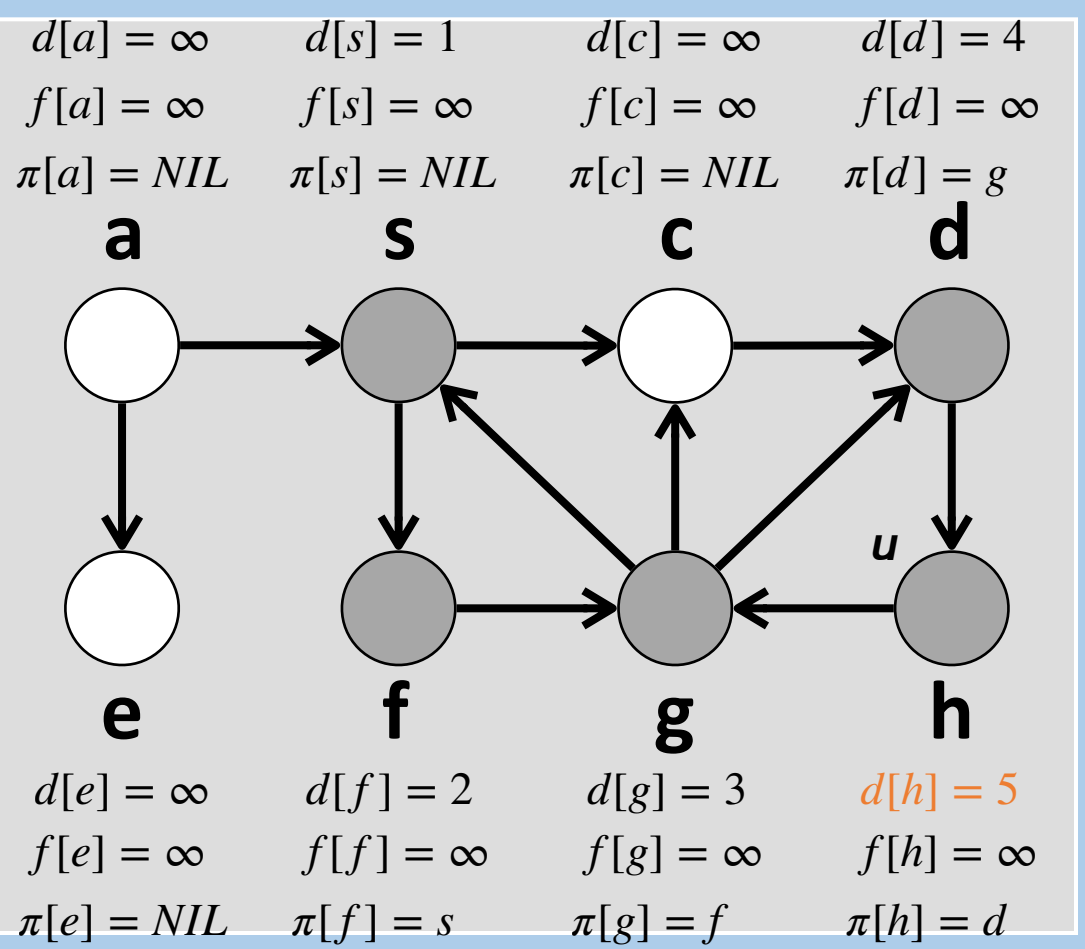
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

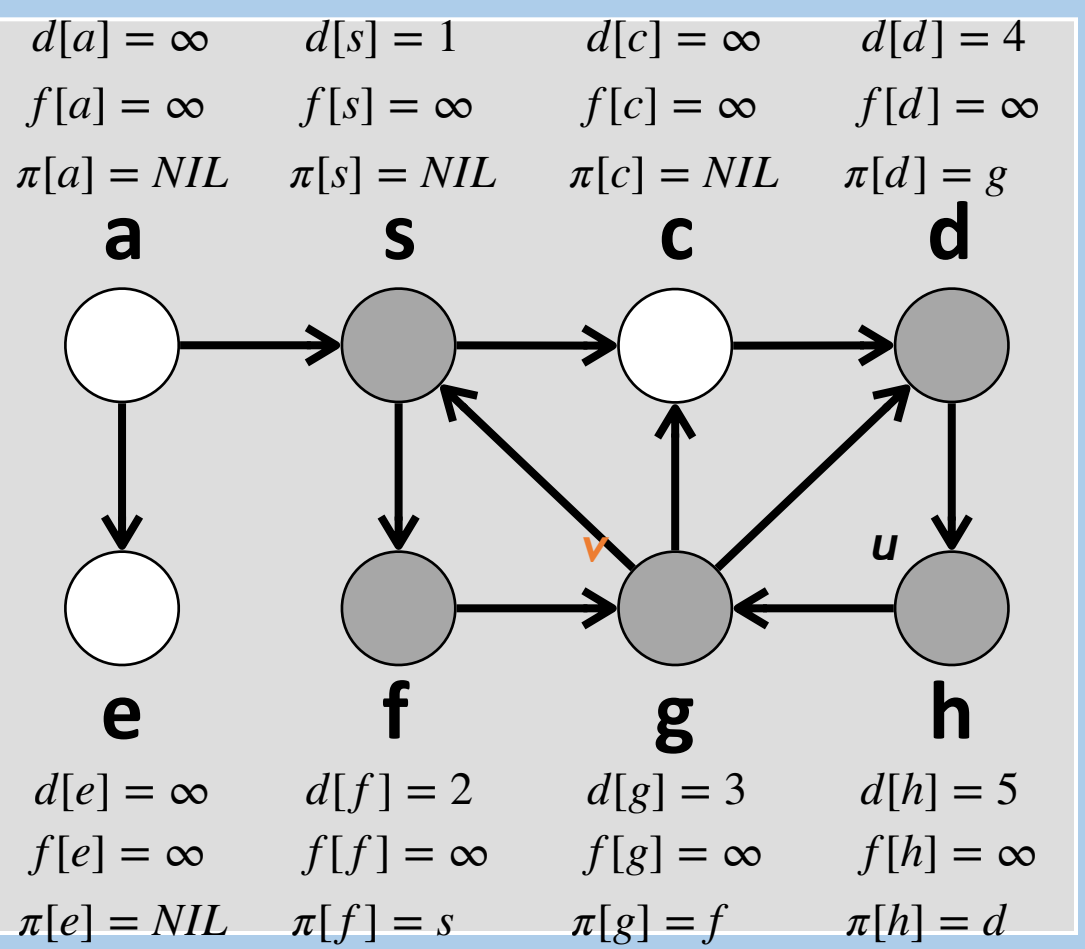
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

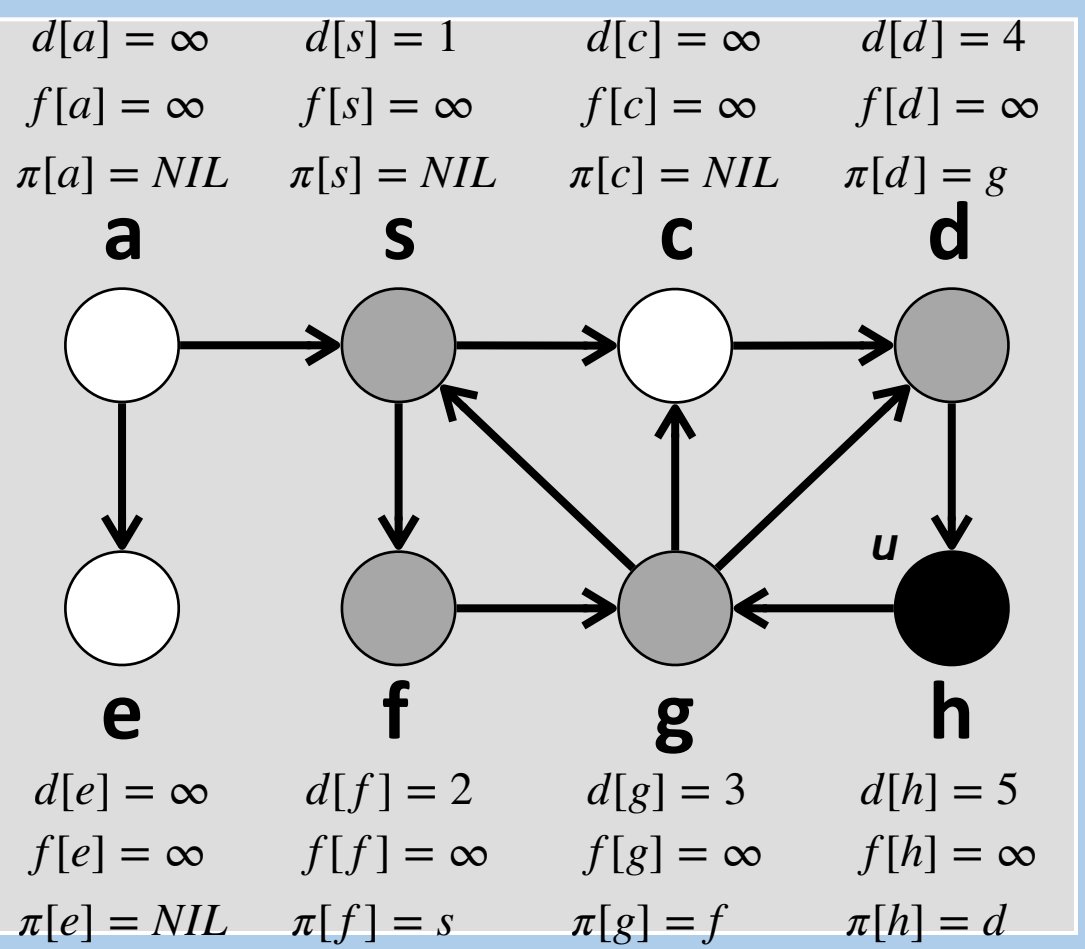
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

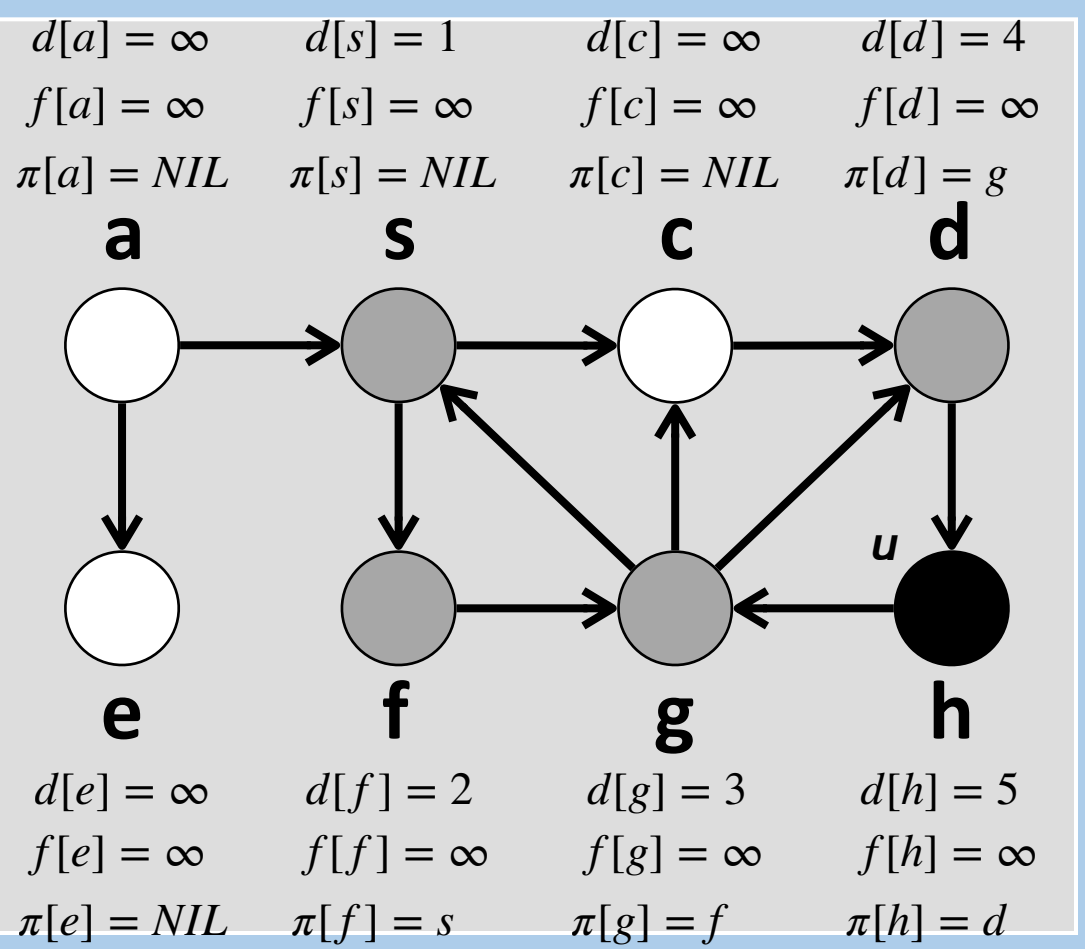
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

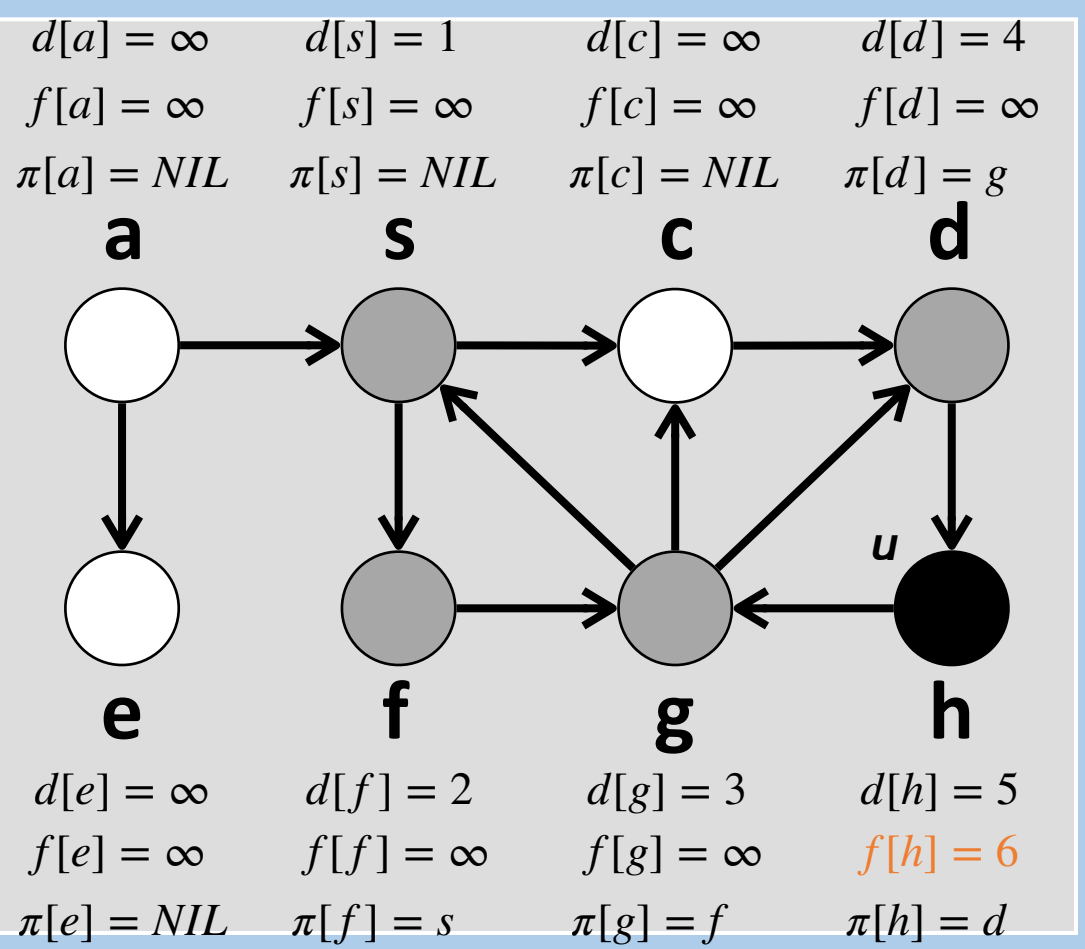
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

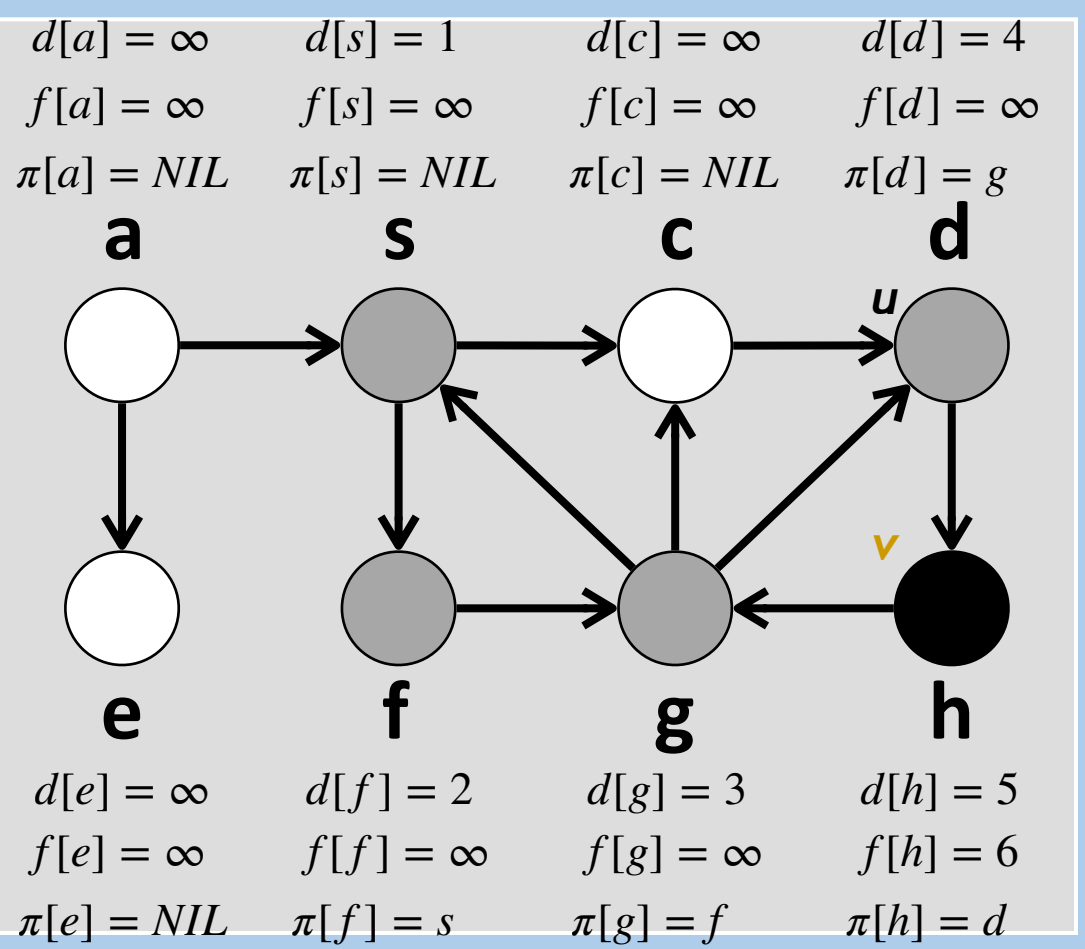
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

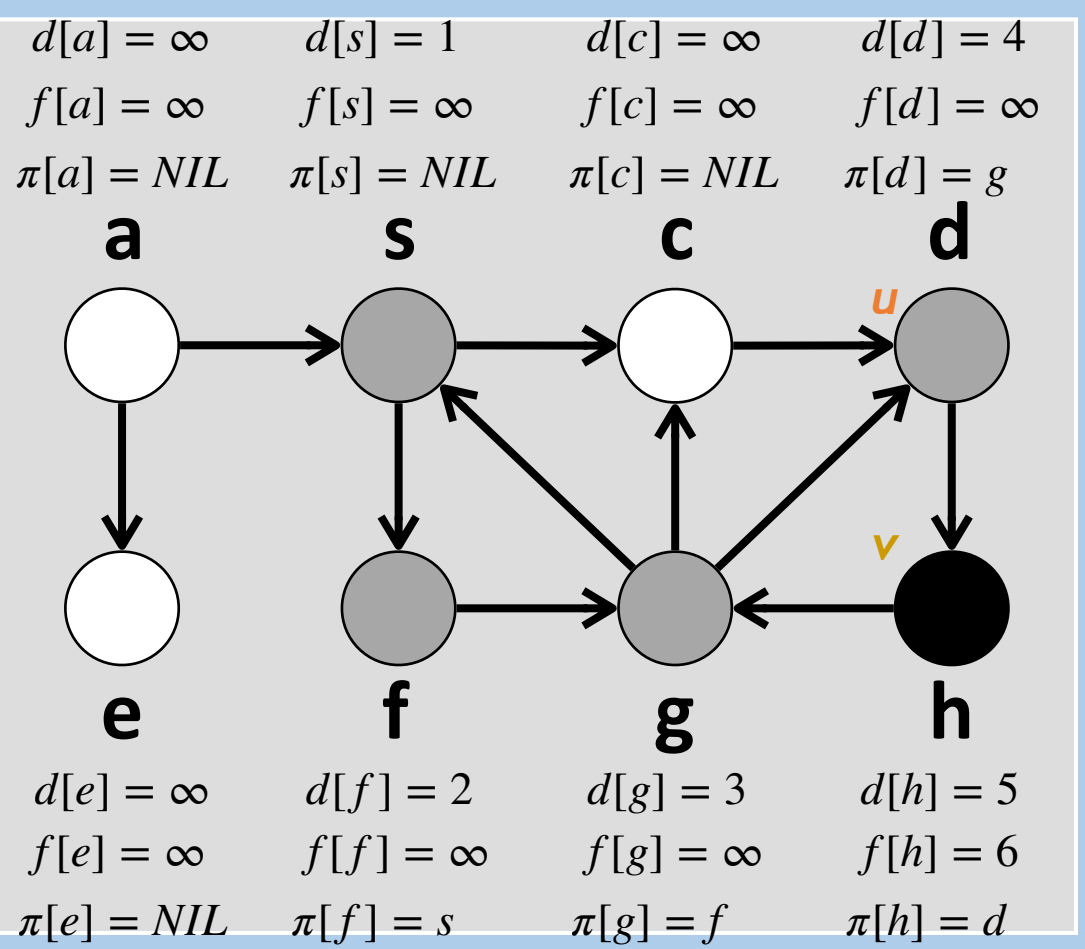
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

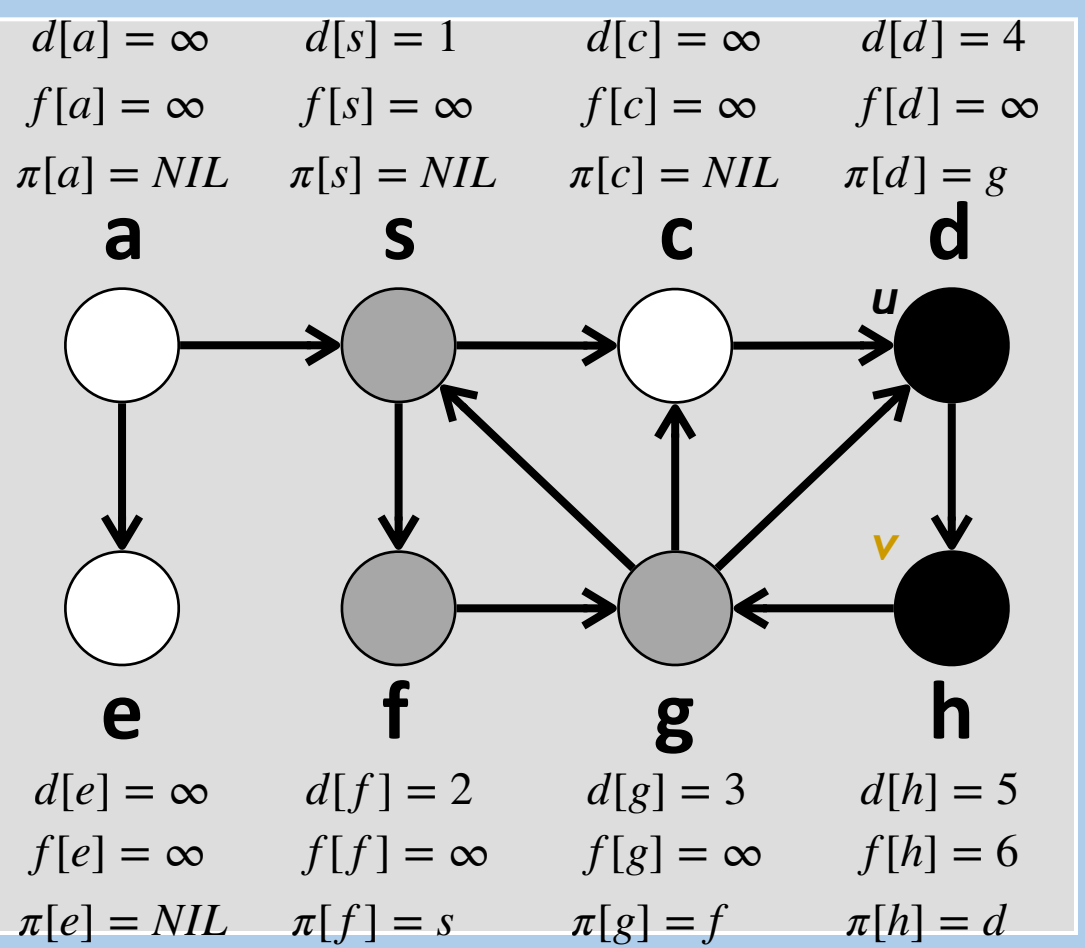
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

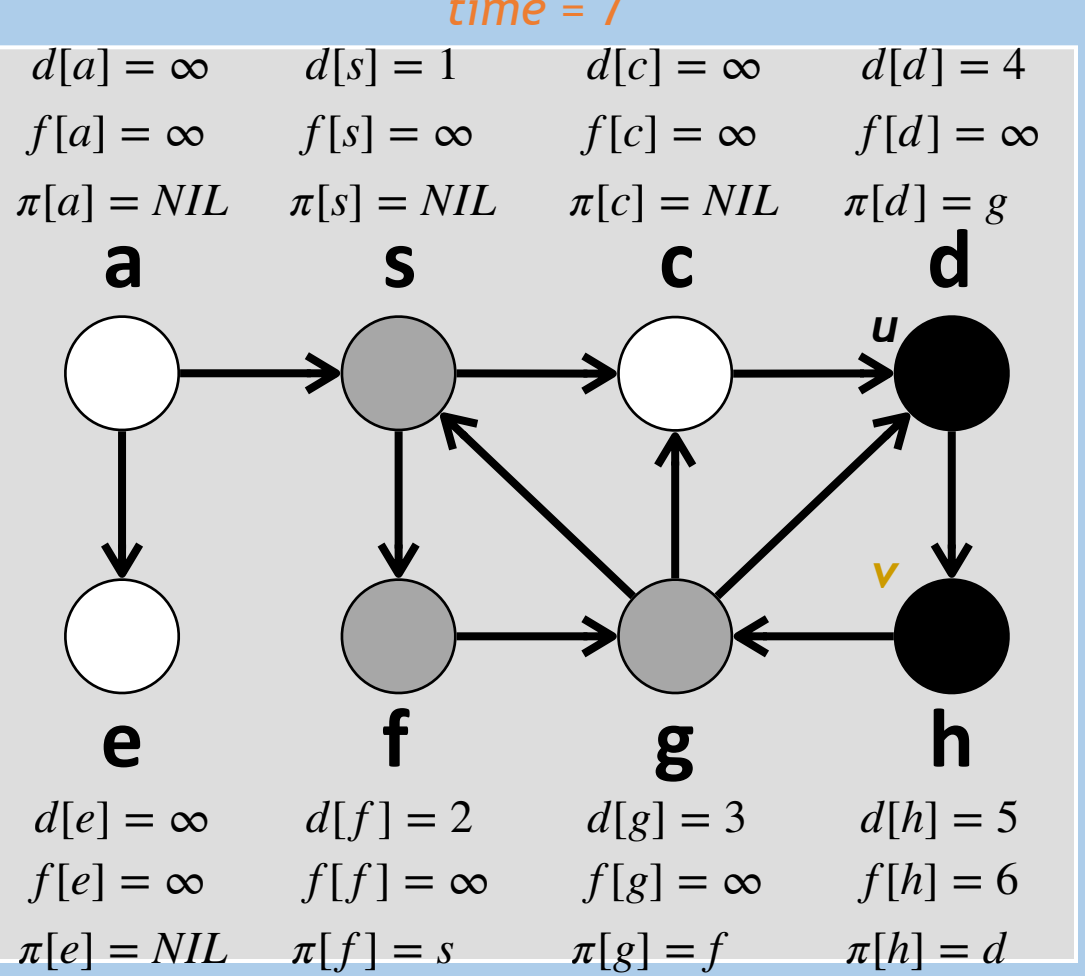
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

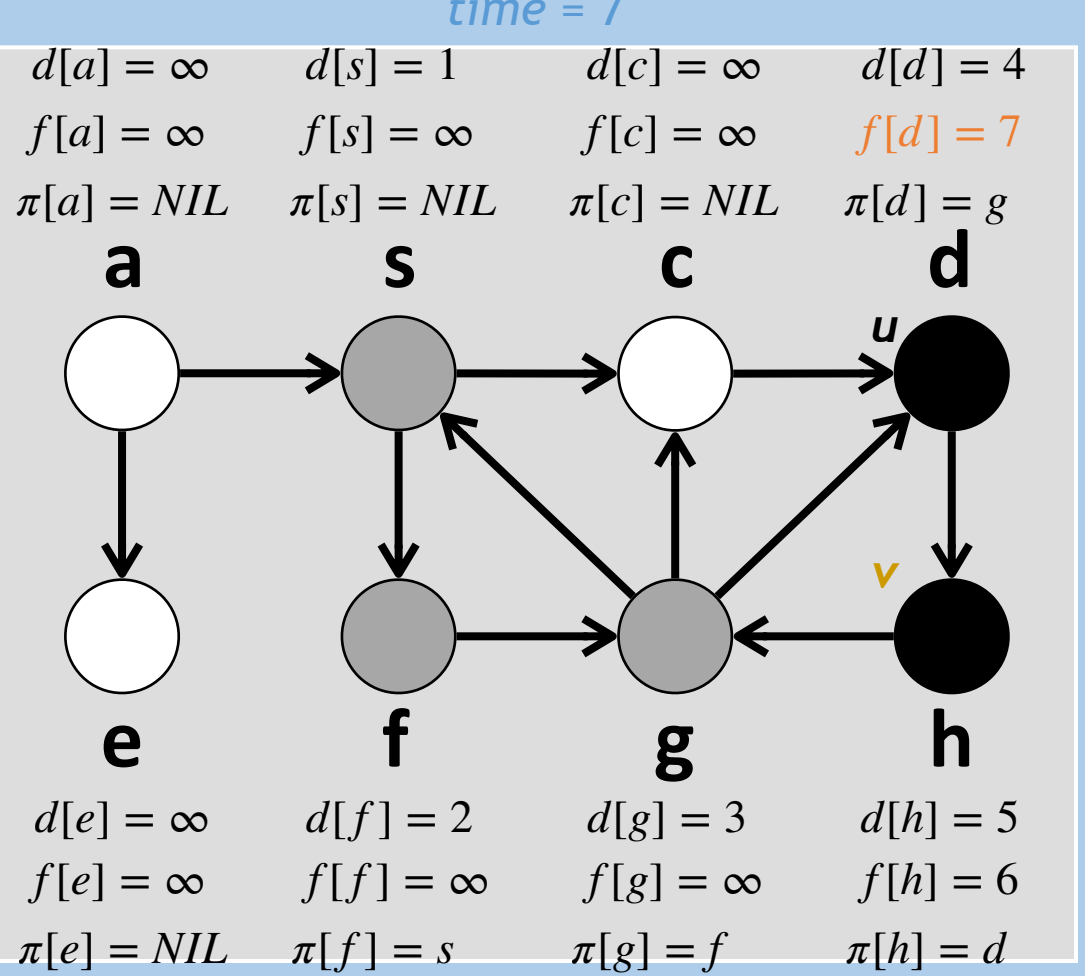
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

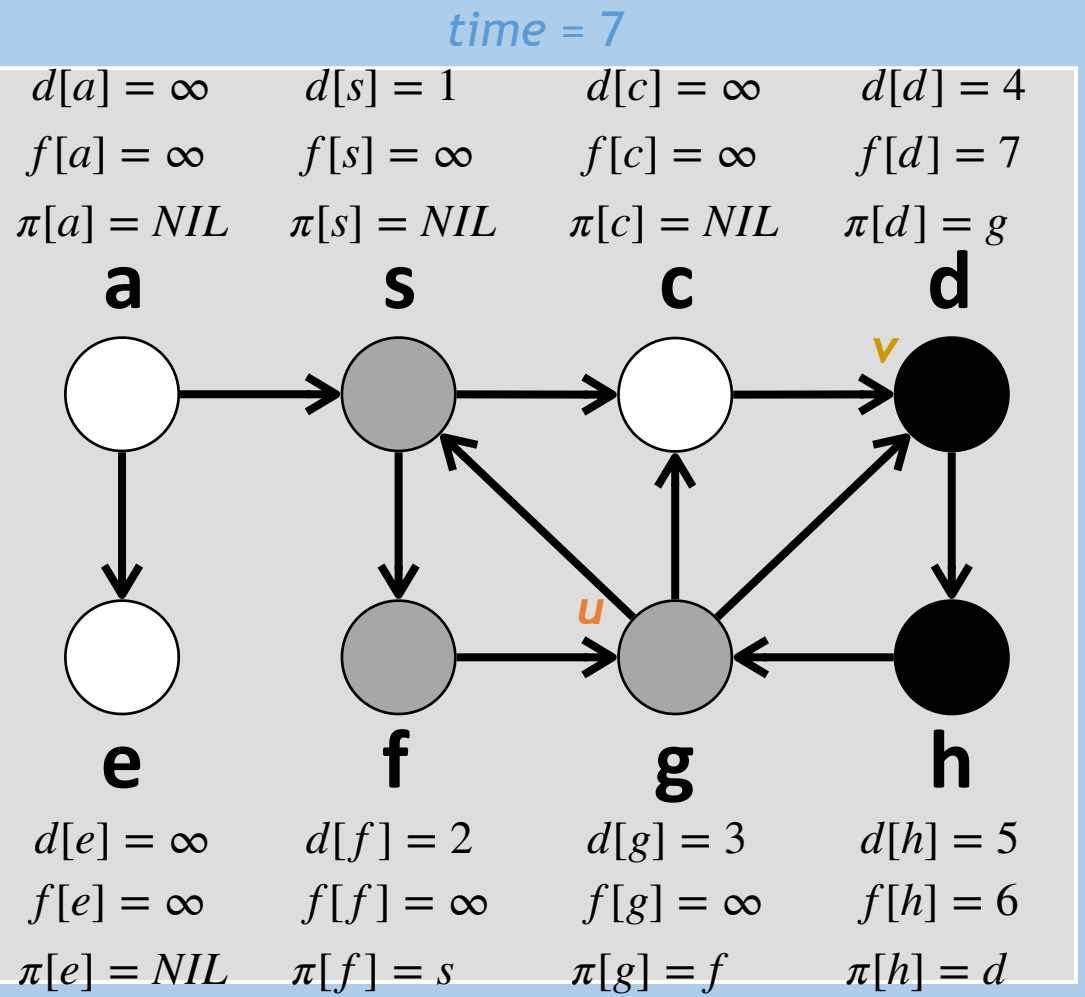
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

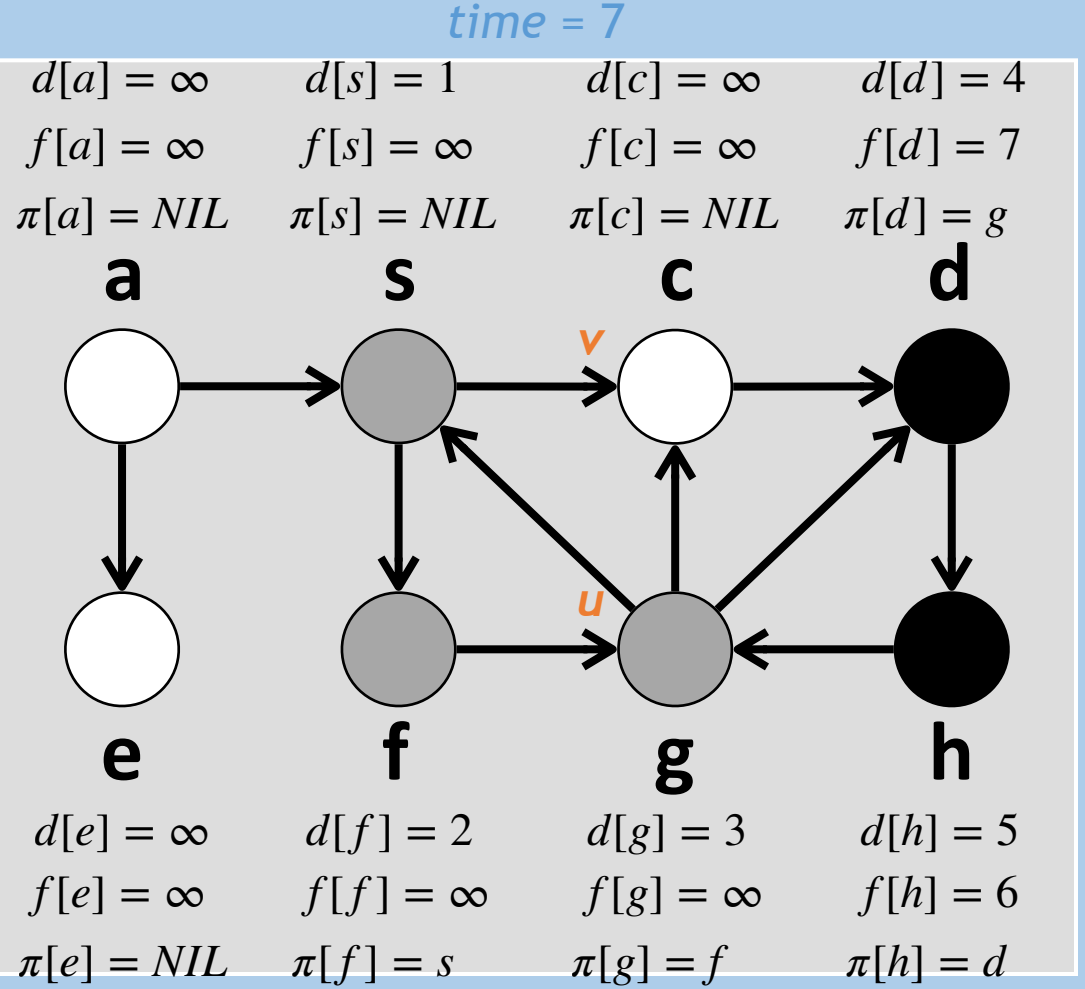
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

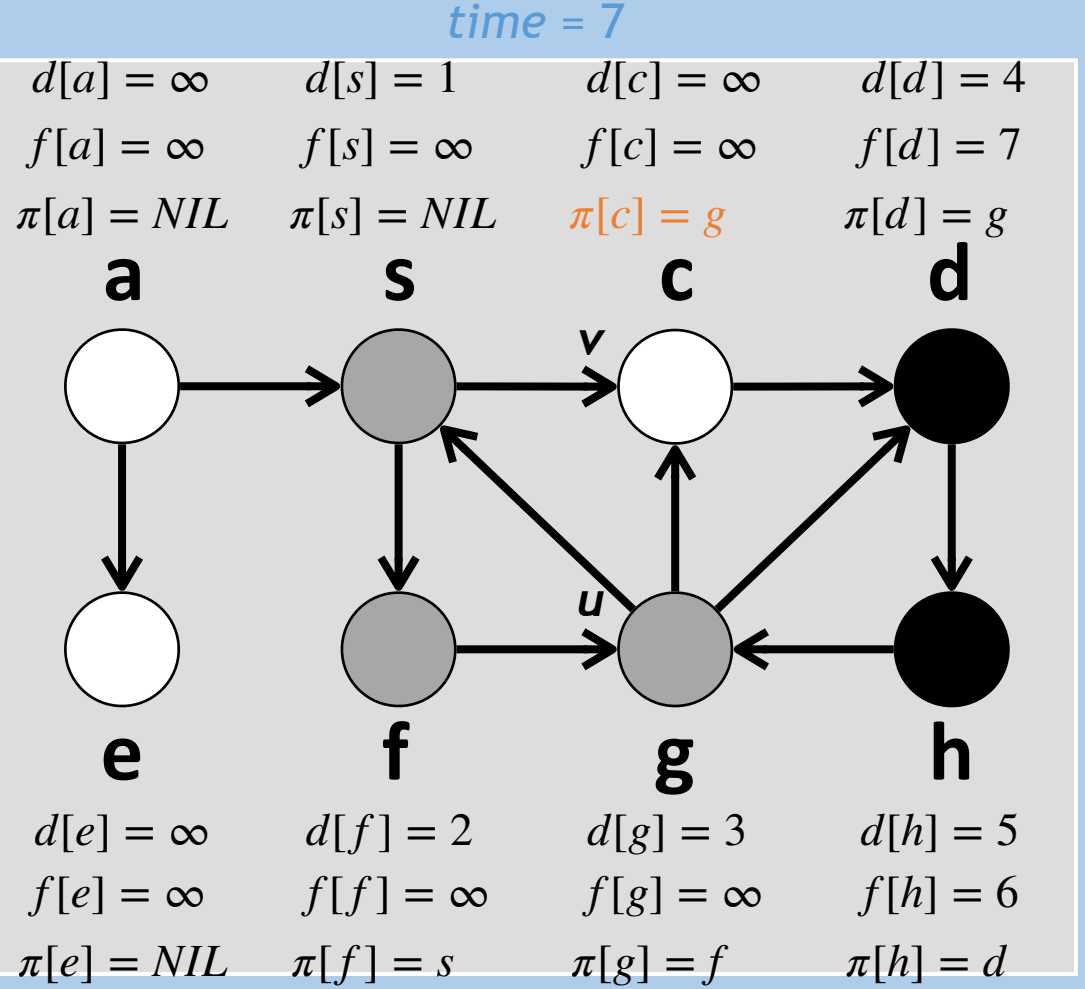
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

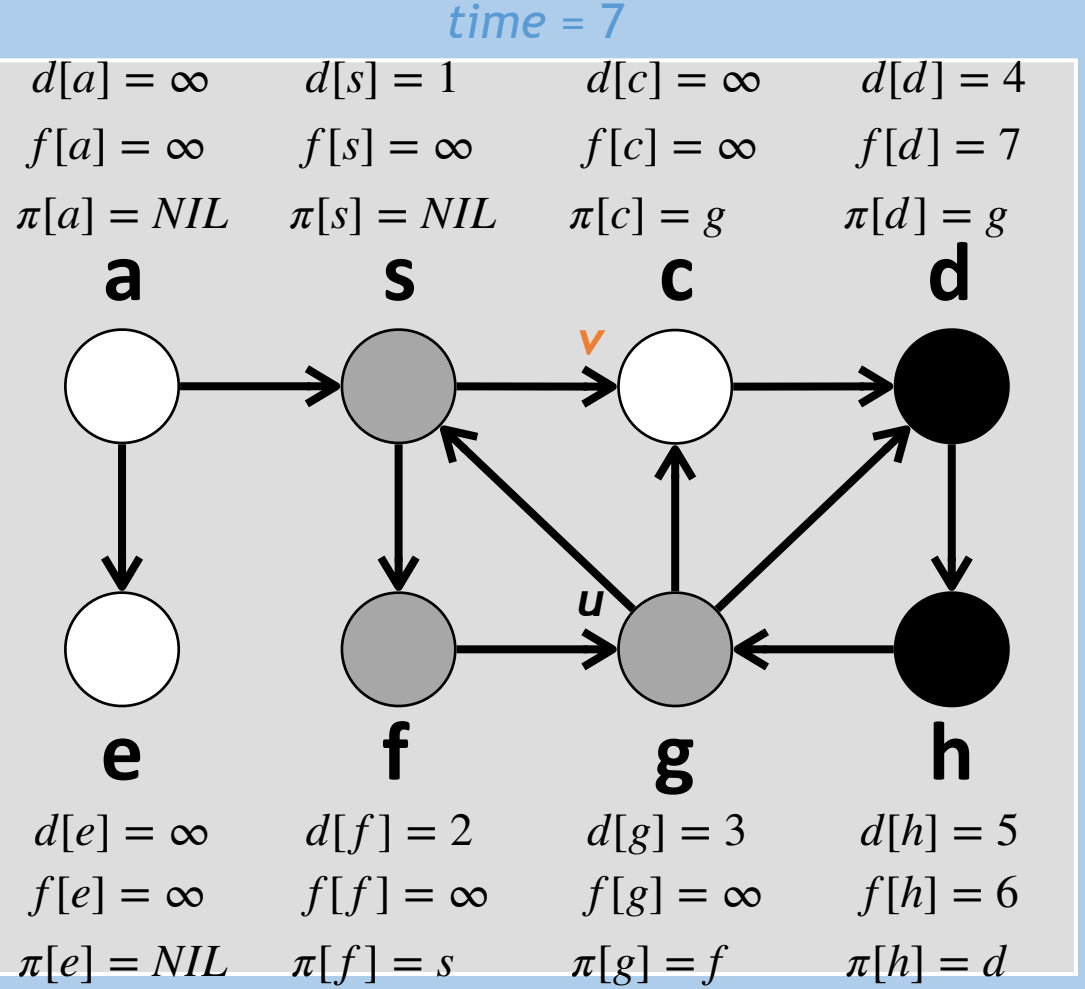
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

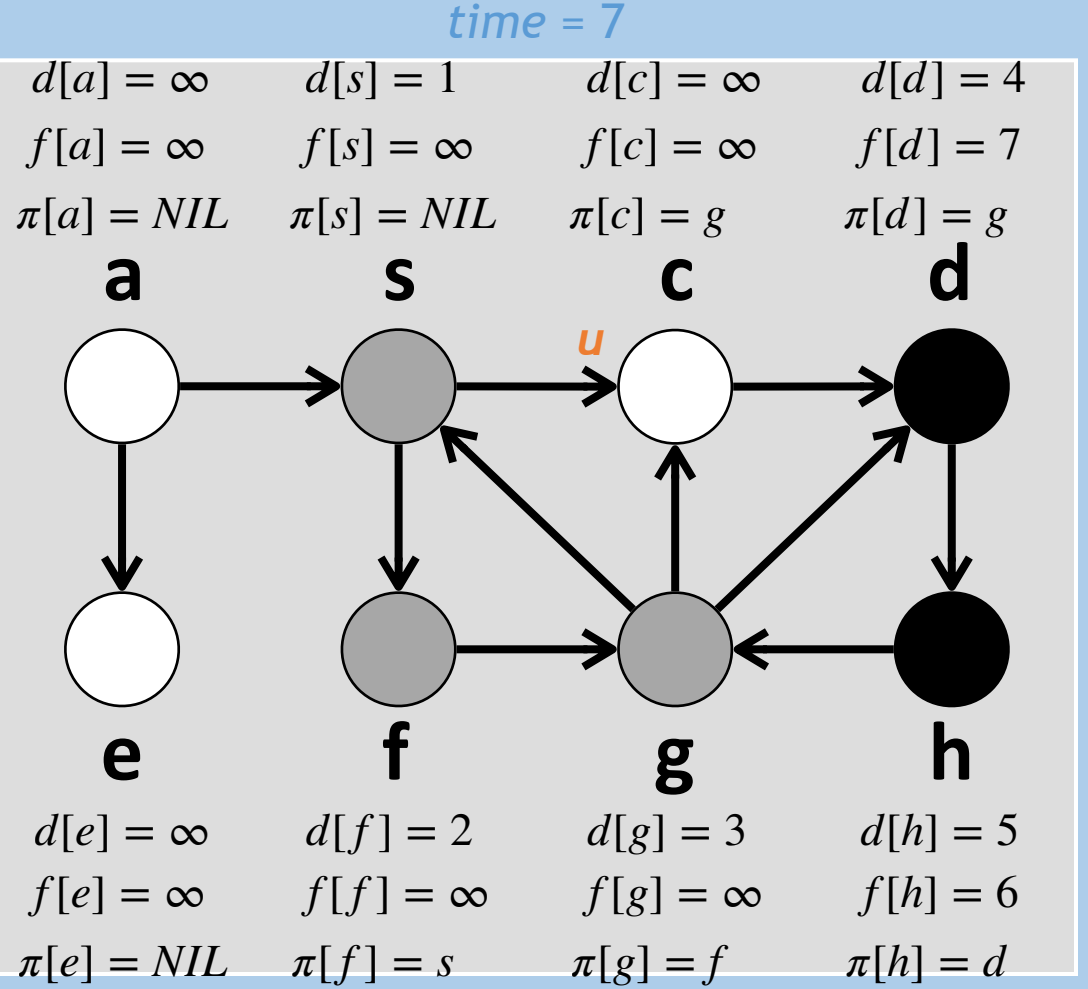
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

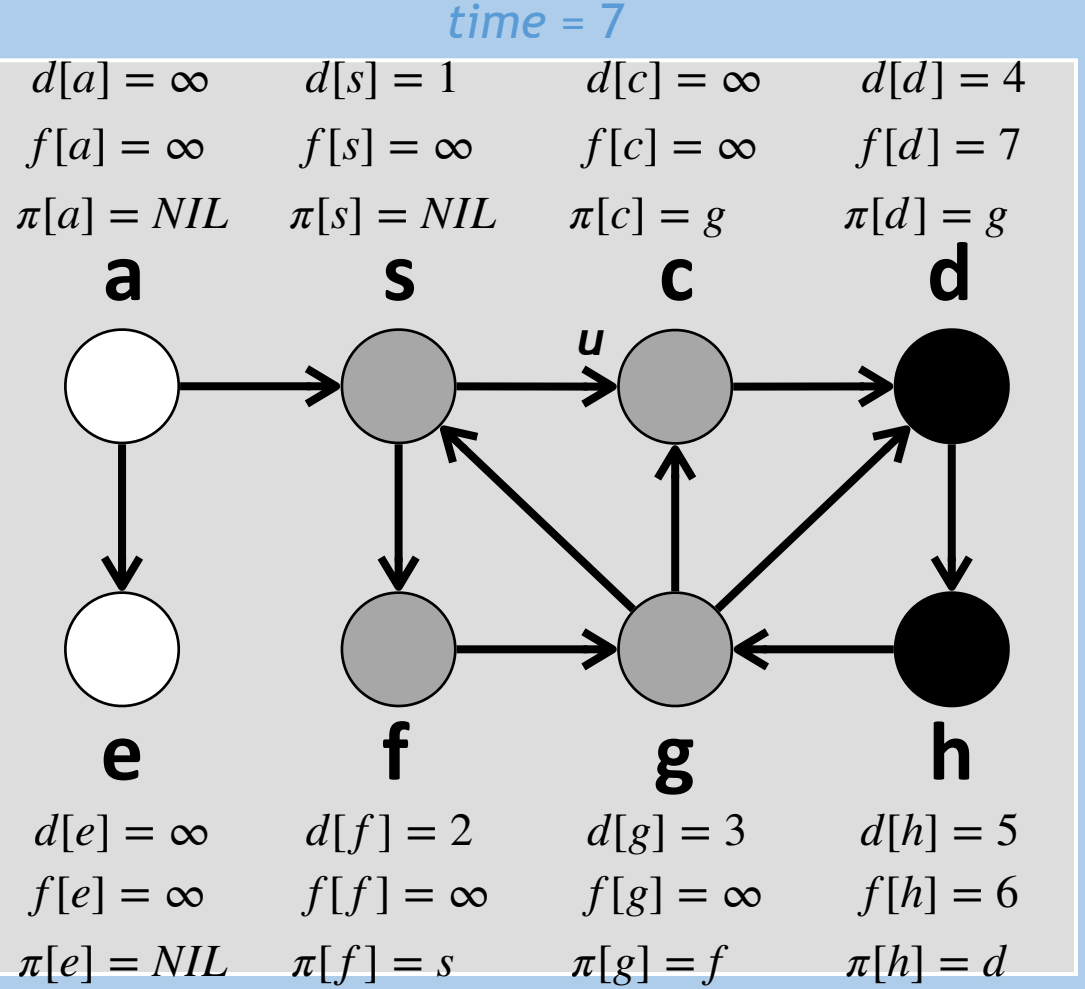
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

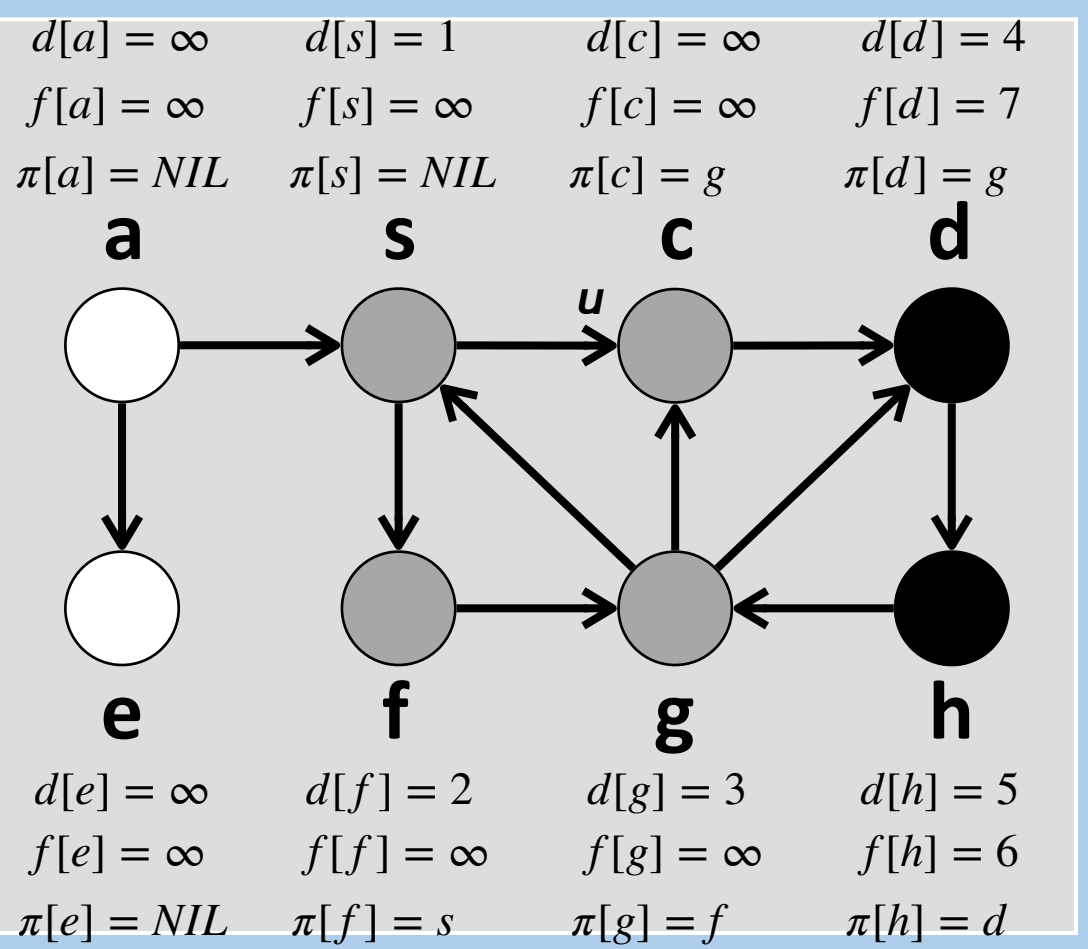
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

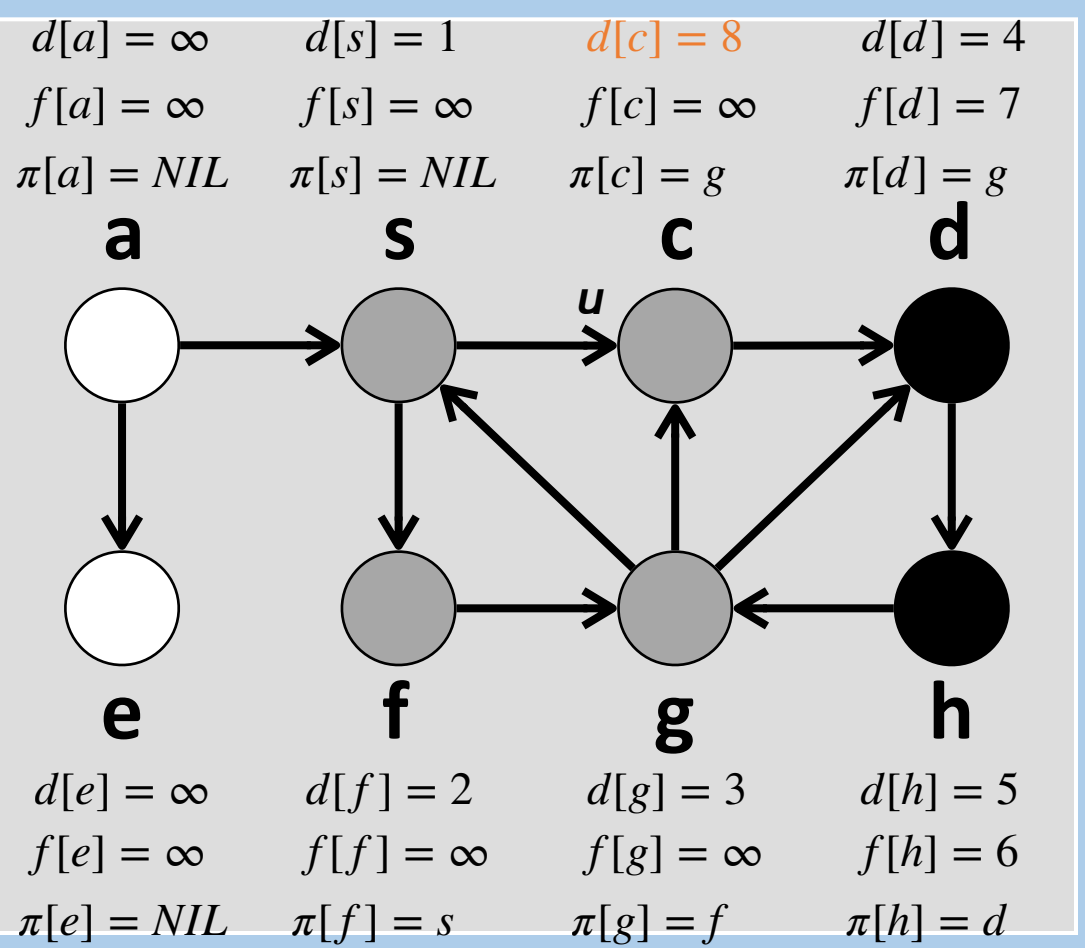
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

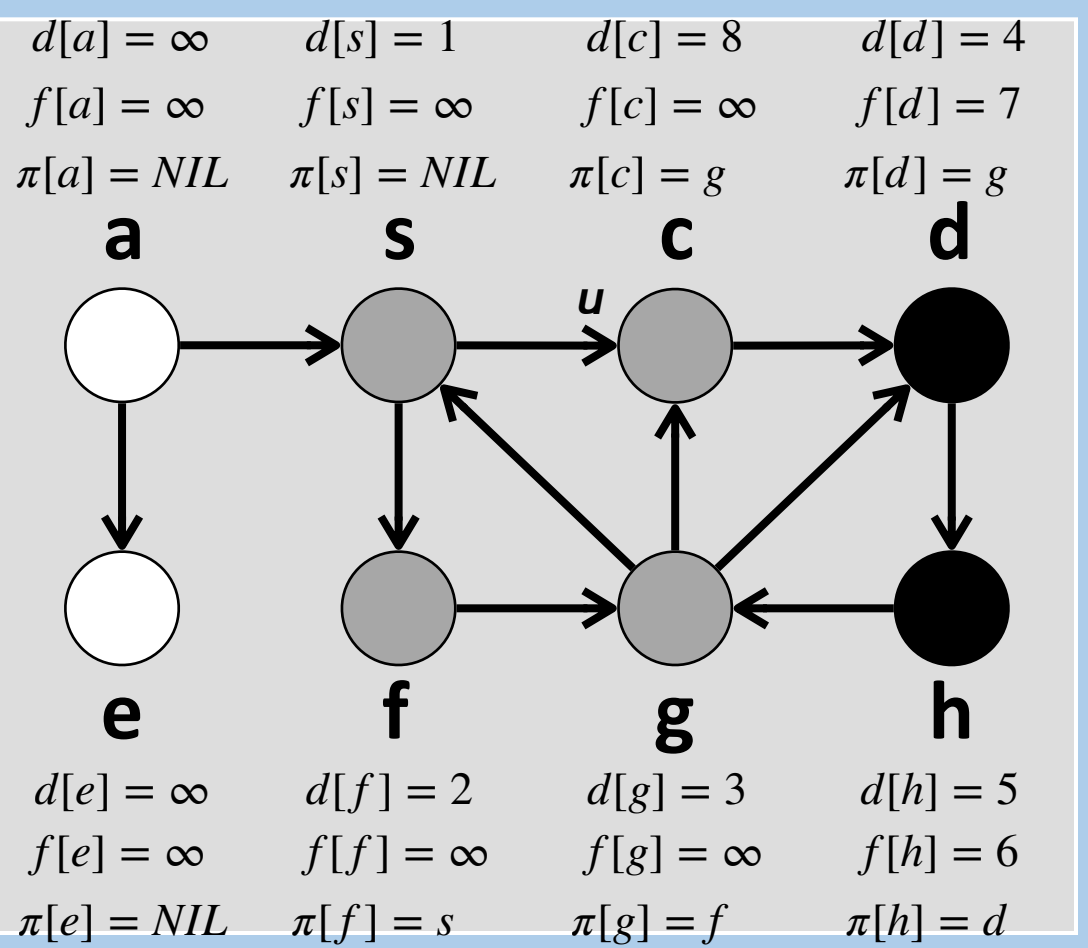
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

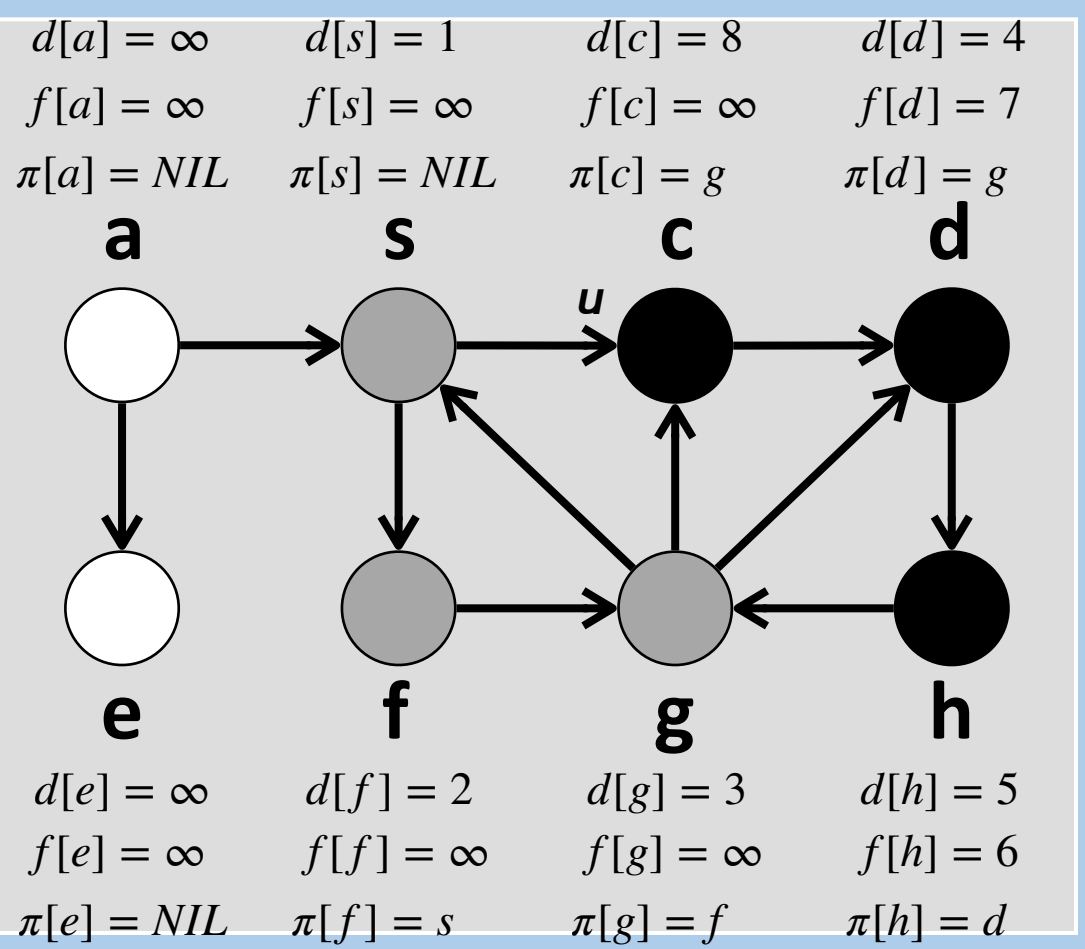
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

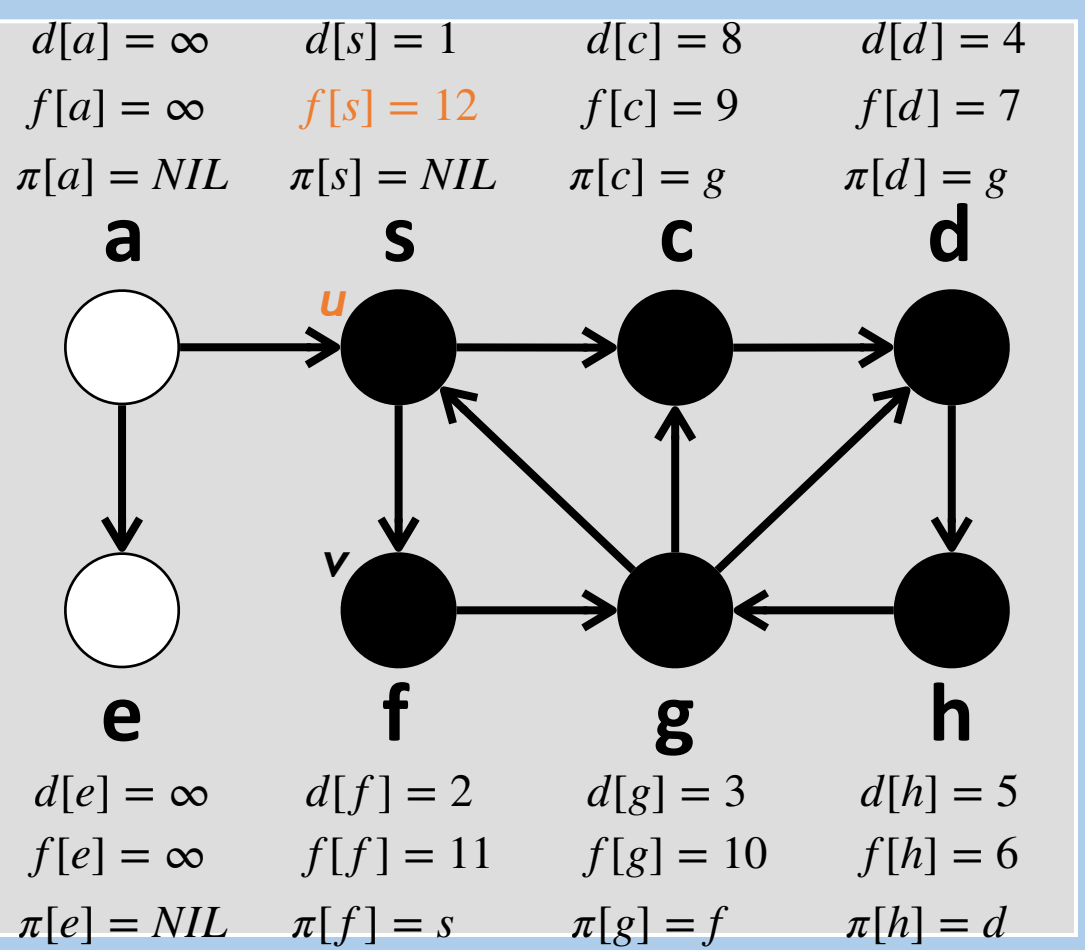
```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



Not-discovered
Discovered
Finished

Depth-First Search

```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```

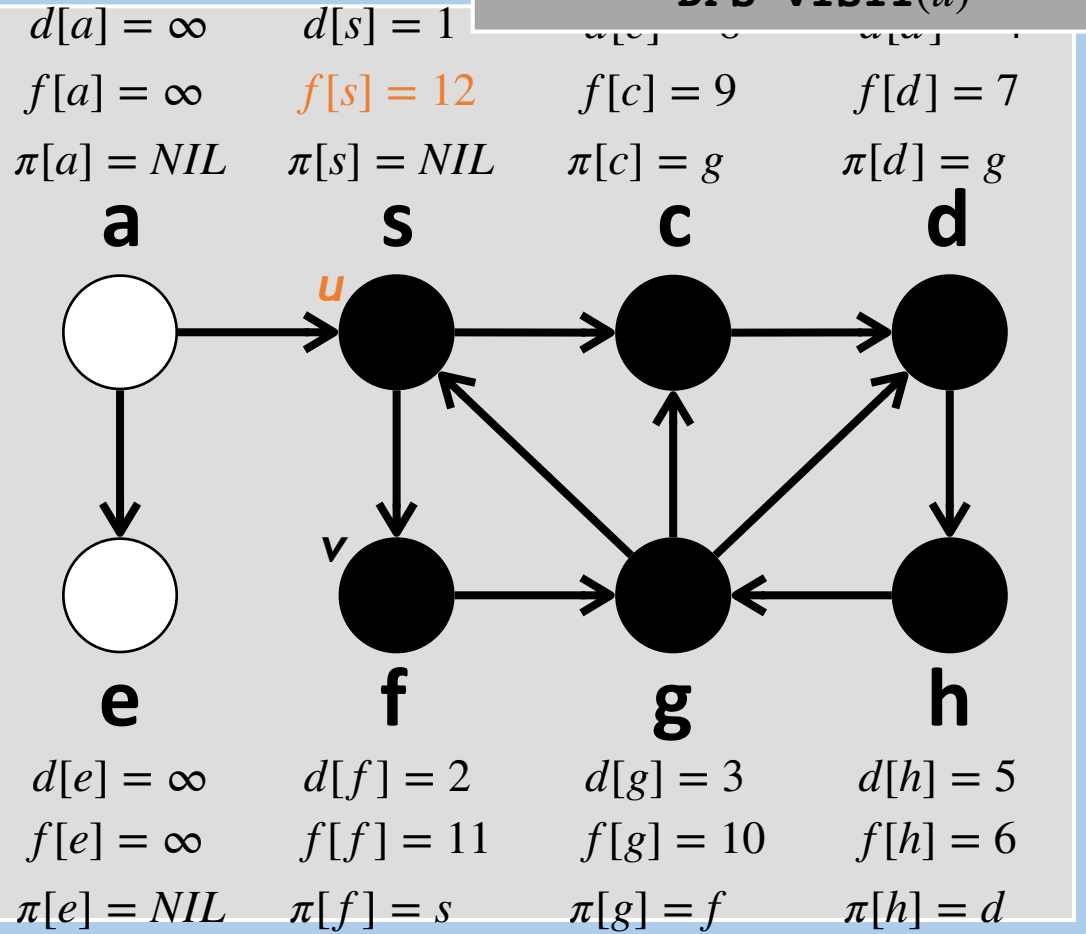


Not-discovered
Discovered
Finished

Depth-First Search

```
For each u ∈ V[G]
    Mark u as not-discovered
    π[v] ← NIL
time ← 0
For each vertex u ∈ V[G]
    If u is not-discovered
        DFS-VISIT(u)
```

```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```

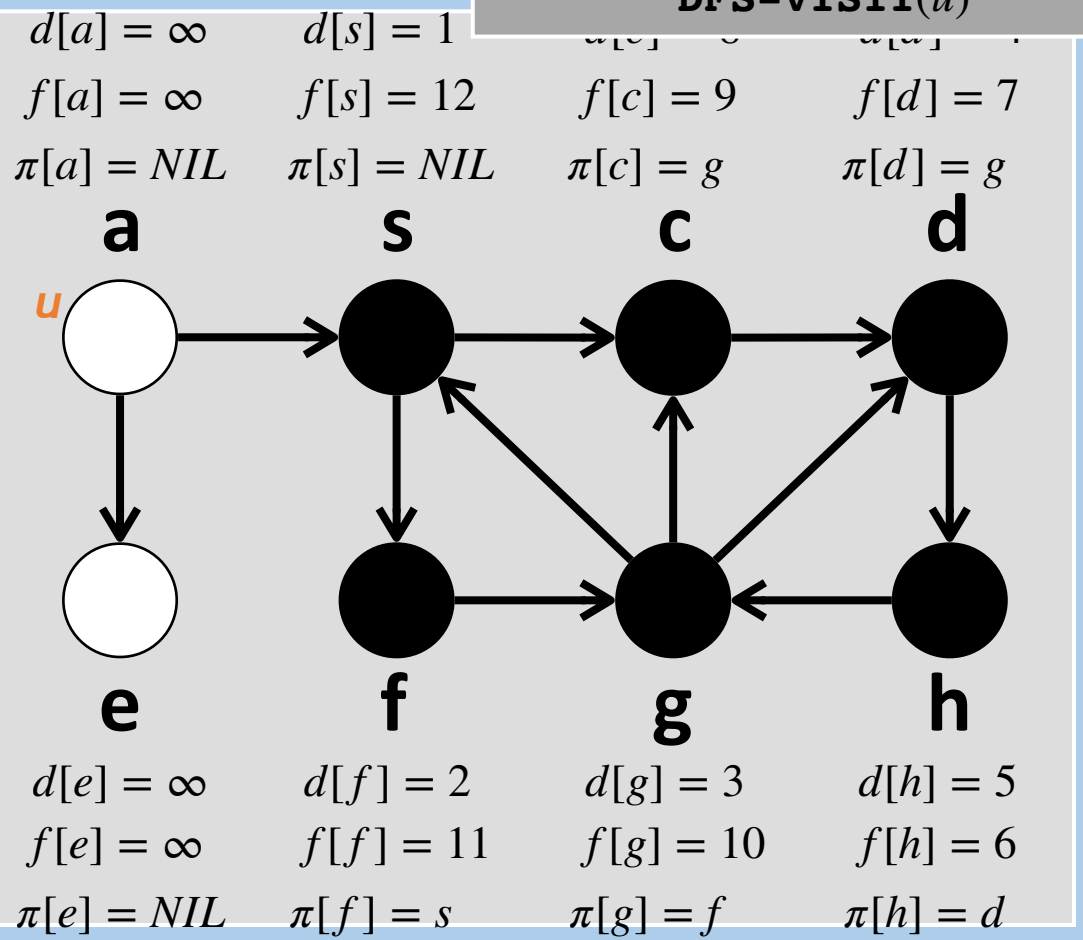


Not-discovered
Discovered
Finished

Depth-First Search

```
For each u ∈ V[G]
    Mark u as not-discovered
    π[v] ← NIL
time ← 0
For each vertex u ∈ V[G]
    If u is not-discovered
        DFS-VISIT(u)
```

```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```



DFS Performance

- Counting:

DFS Performance

- Counting:
  - Each vertex is marked to white, gray, and black once: $O(|V|)$ time

DFS Performance

- Counting:
  - Each vertex is marked to white, gray, and black once: $O(|V|)$ time
  - Each edge is considered at most twice: $O(|E|)$ time

DFS Performance

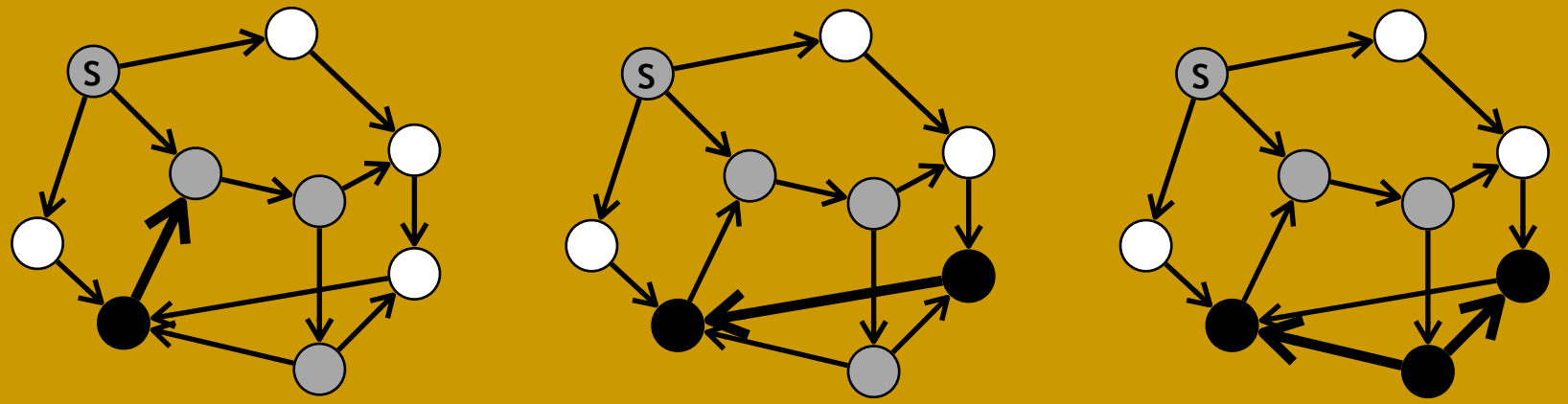
- Counting:
  - Each vertex is marked to white, gray, and black once: $O(|V|)$ time
  - Each edge is considered at most twice: $O(|E|)$ time
- Total time complexity: $O(|V| + |E|)$

Linear time on graphs:
$O(|V| + |E|)$

What's Happening

```
Procedure DFS-VISIT(u)
    Mark u as discovered
    time ← time + 1
    d[u] ← time
    For each neighbor v of u
        If v is not-discovered
            π[v] ← u
            DFS-VISIT(v)
    Mark u as finished
    time ← time + 1
    f[u] ← time
```

- Each vertex is marked as
  - White: **not-discovered** by the algorithm
  - Gray: **discovered** by the algorithm
  - Black: **finished**; all its neighbors are discovered



A vertex is marked in black if every neighbor of it is discovered (gray or black).

The stack contains all gray vertices. Once a vertex turns black, it is popped from the stack.

(Stack: from system function call)

Outline

- Graph representations
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
    - Parenthesis theorem
    - Edge classification
    - White-path theorem
- Two applications on DFS:
  - Topological sort
  - Strongly connected components

Depth-First Search Properties

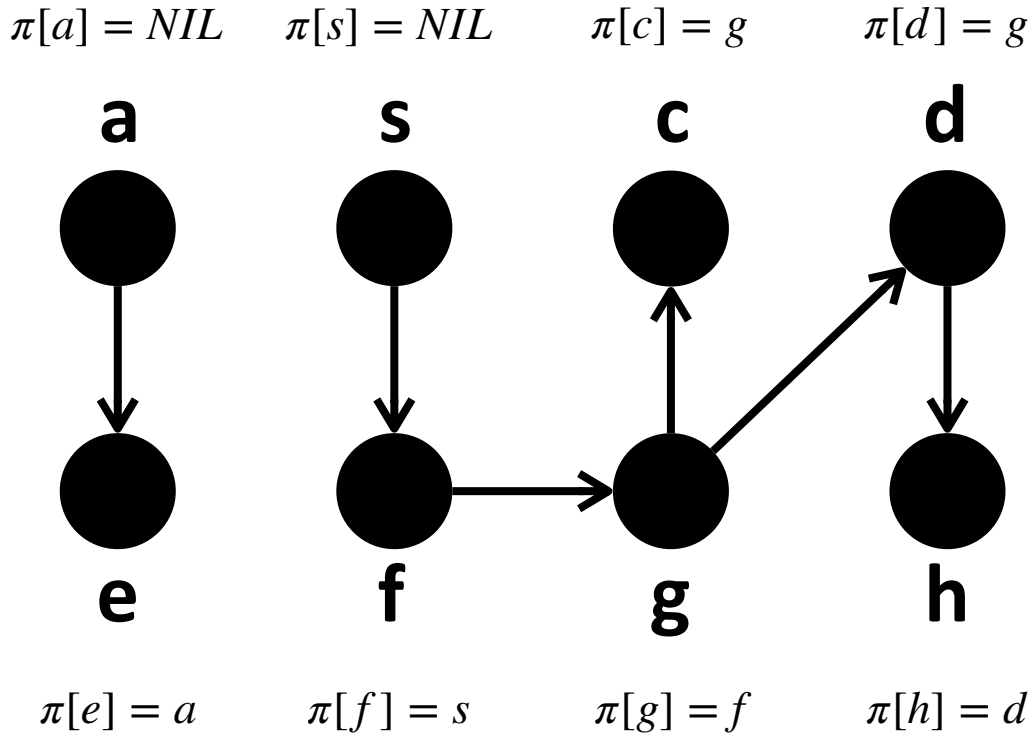
- Parenthesis theorem
- Edge classification
- White-path theorem

Depth-First Search Properties

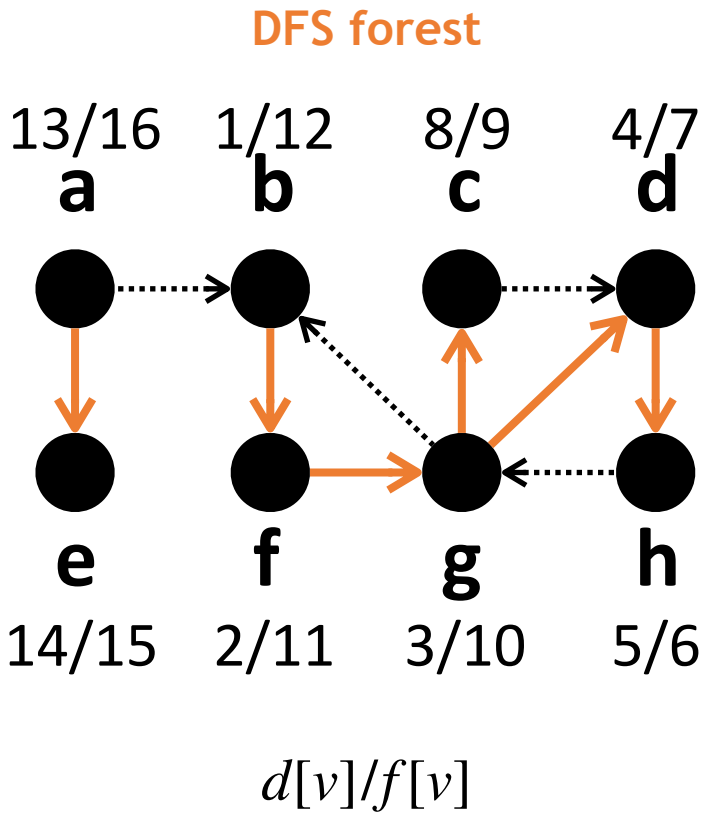
- Parenthesis theorem
- Edge classification
- White-path theorem

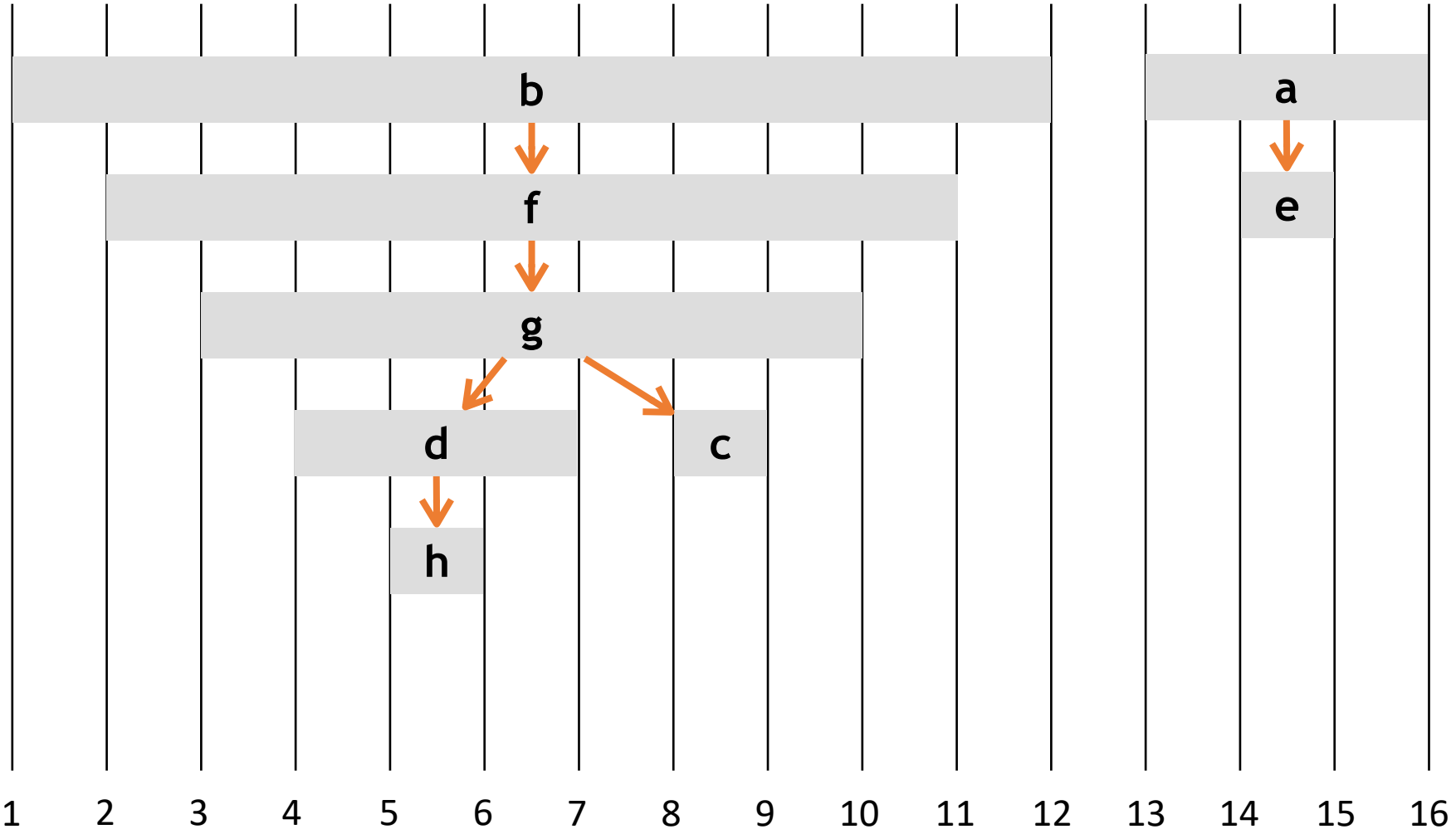
Depth-First Forest

- The predecessor subgraph of a DFS $G_\pi = (V, E_\pi)$, where $E_\pi = \{(\pi[v], v) \mid v \in V \text{ and } \pi[v] \neq \phi\}$, is a forest

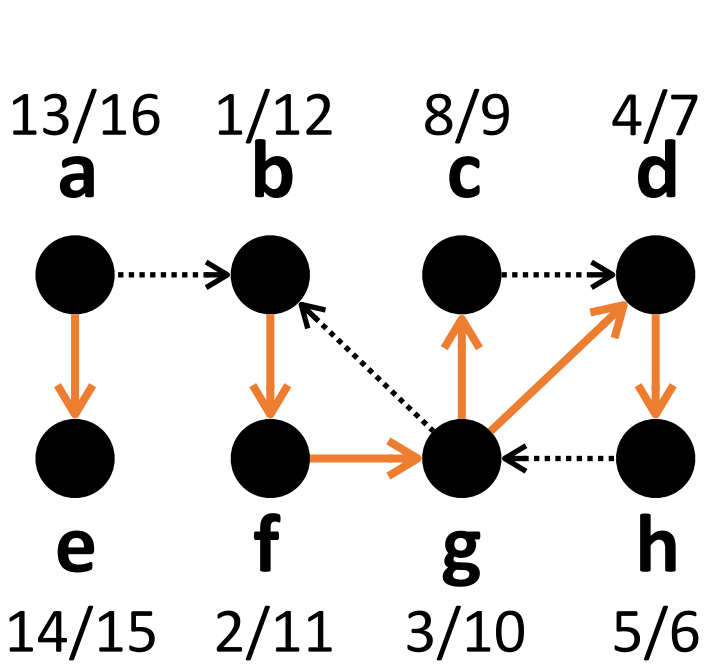


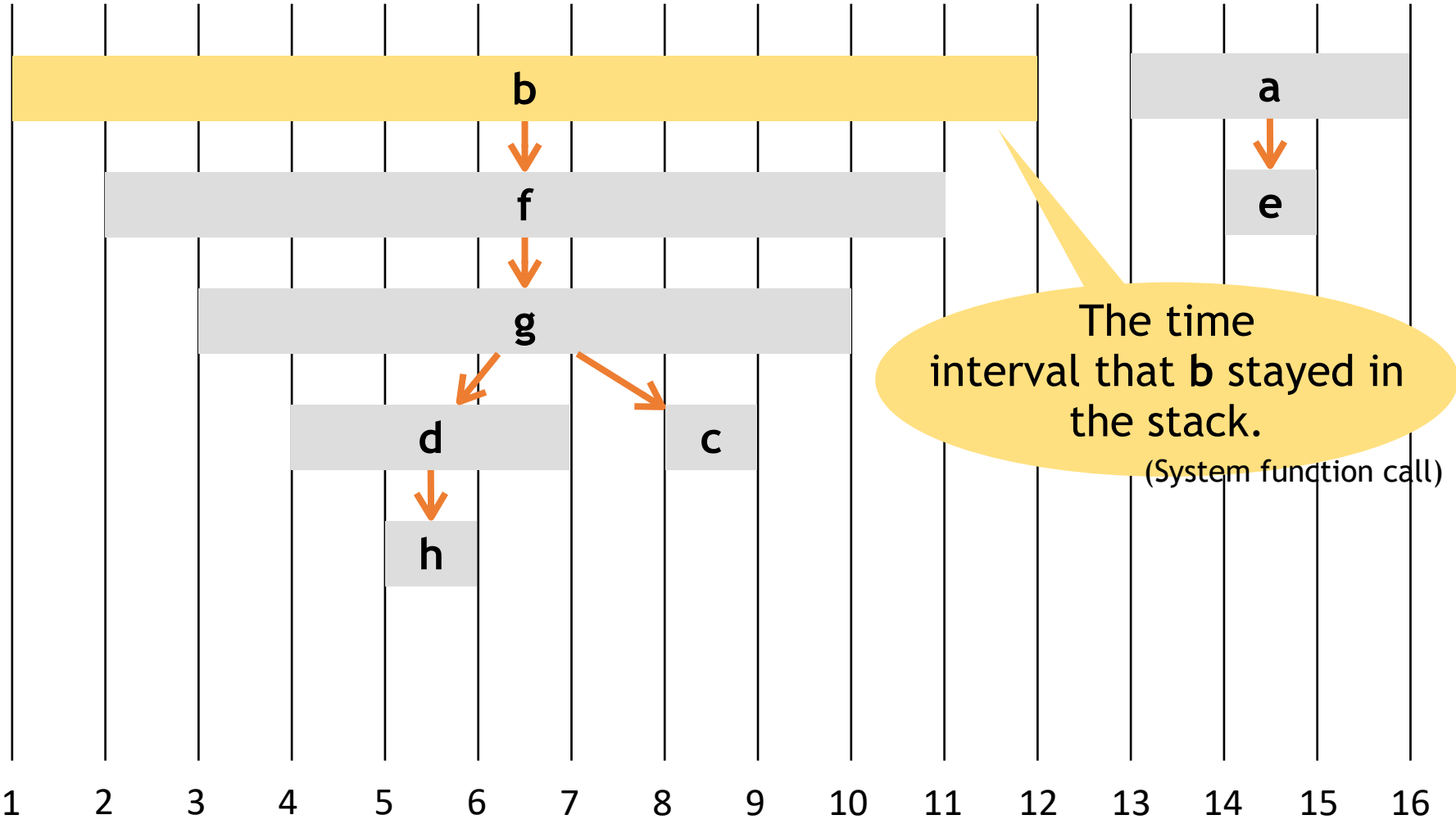
Parenthesis Structure of DFS



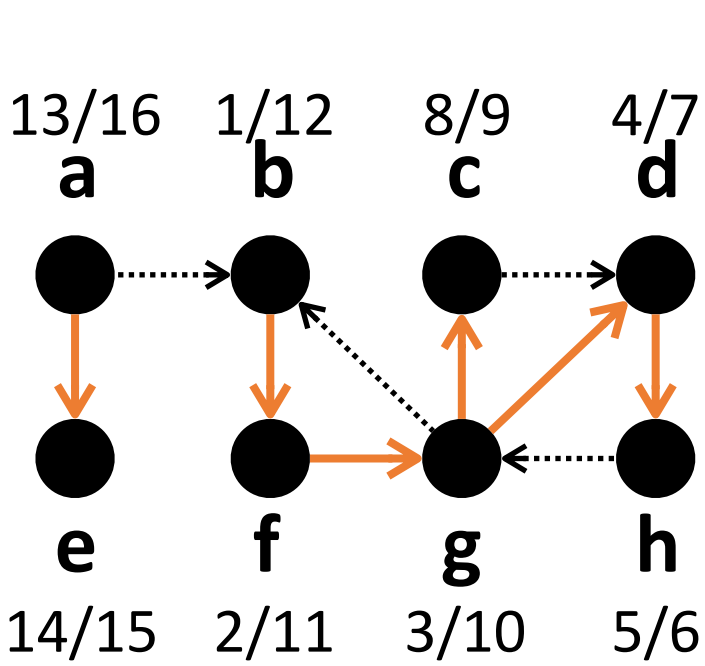


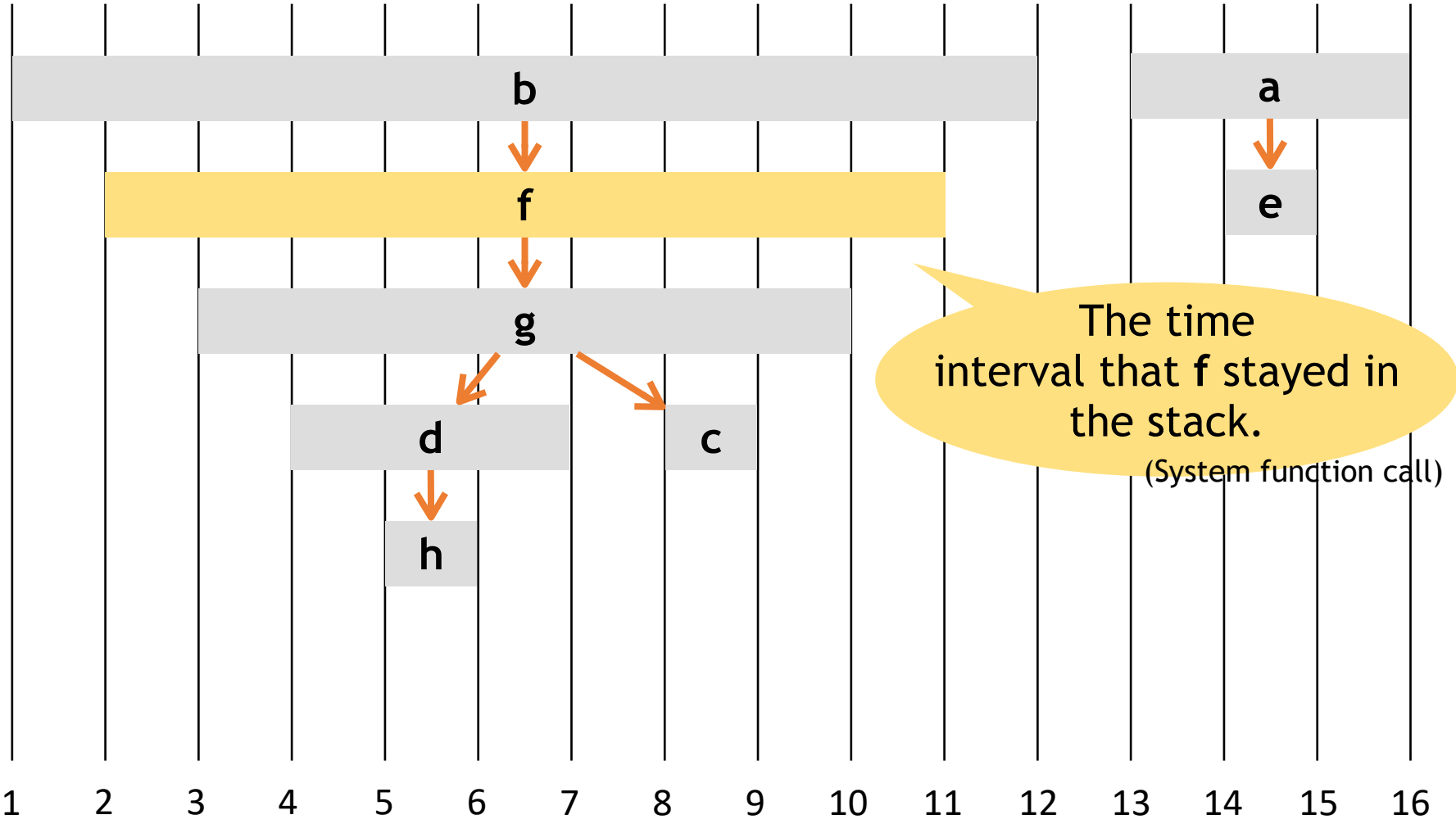
Parenthesis Structure of DFS





Parenthesis Structure of DFS





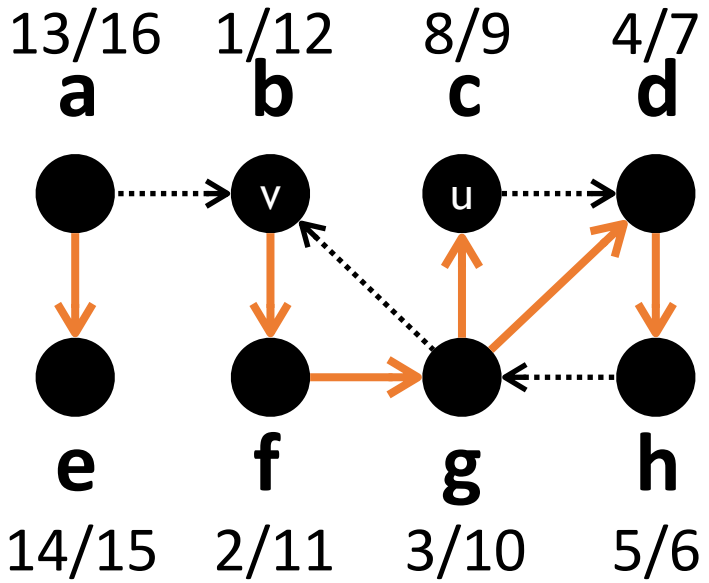
The time interval that **f** stayed in the stack.

(System function call)

Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices $u, v \in V$, there are three cases of the positions of u and v in the depth-first forest:

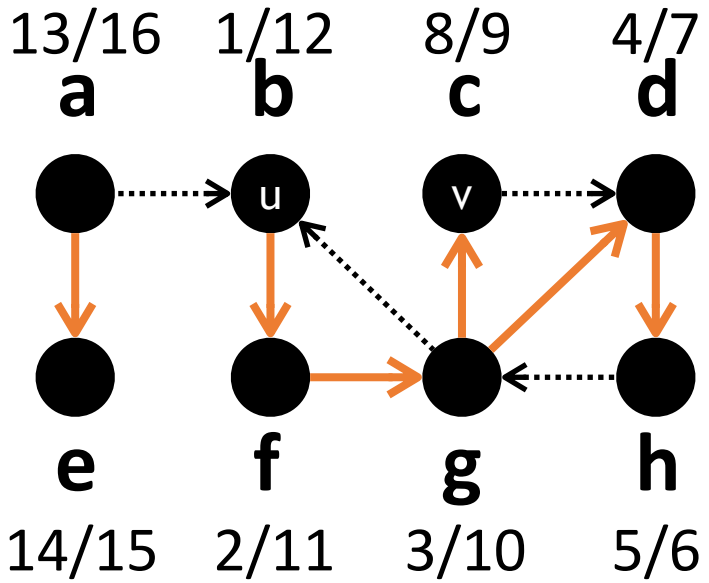
- u is a descendant of v,



Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices $u, v \in V$, there are three cases of the positions of u and v in the depth-first forest:

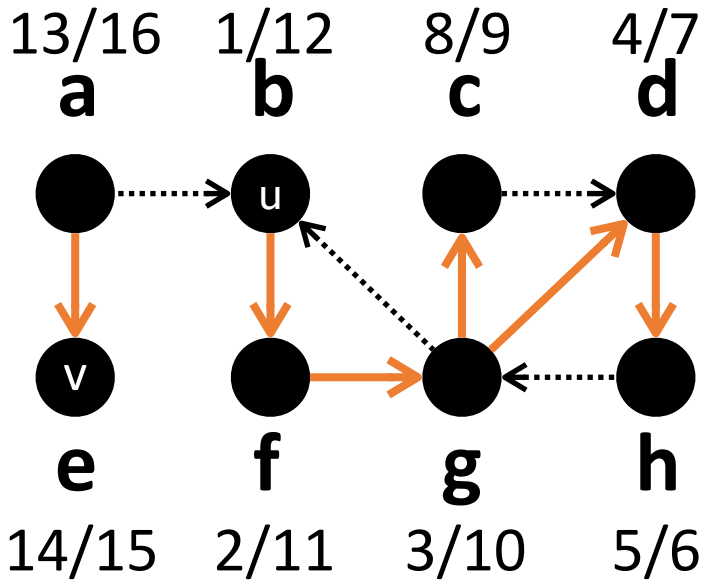
- u is a descendant of v,
- v is a descendant of u, or



Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices $u, v \in V$, there are three cases of the positions of u and v in the depth-first forest:

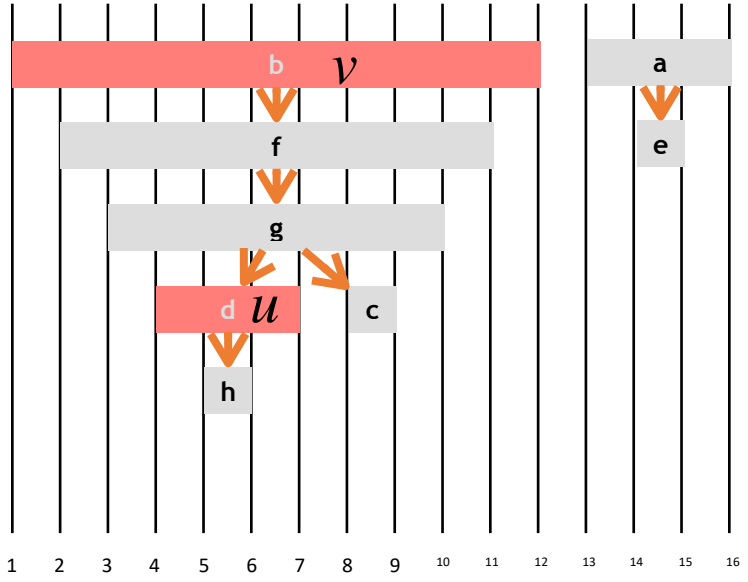
- u is a descendant of v,
- v is a descendant of u, or
- either u nor v is as descendant of the other



Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices $u, v \in V$, there are three cases of the positions of u and v in the depth-first forest:

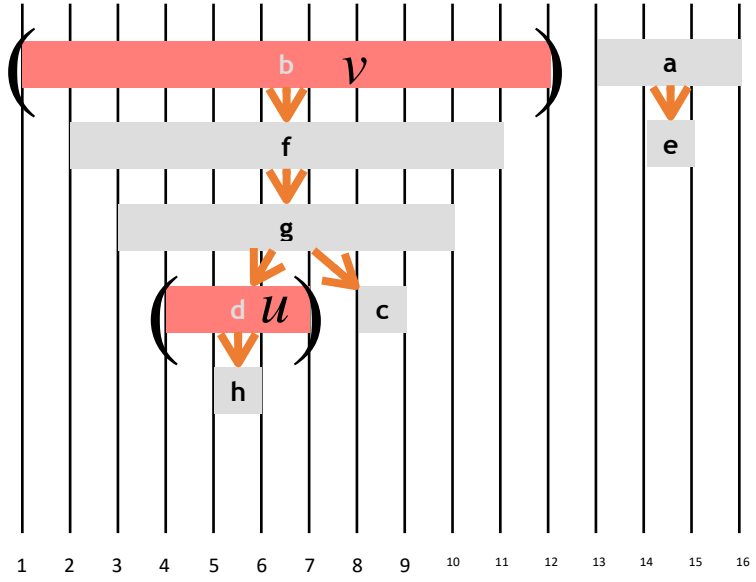
- u is a descendant of v,
  - $[d[u], f[u]]$ is contained entirely within $[d[v], f[v]]$
- v is a descendant of u, or
- either u nor v is as descendant of the other



Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices $u, v \in V$, there are three cases of the positions of u and v in the depth-first forest:

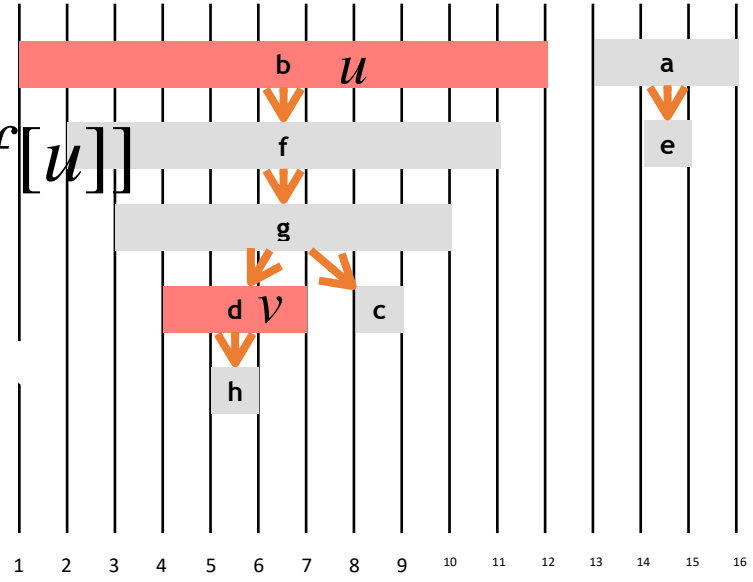
- u is a descendant of v,
  - $[d[u], f[u]]$ is contained entirely within $[d[v], f[v]]$
- v is a descendant of u, or
- either u nor v is as descendant of the other



Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices $u, v \in V$, there are three cases of the positions of u and v in the depth-first forest:

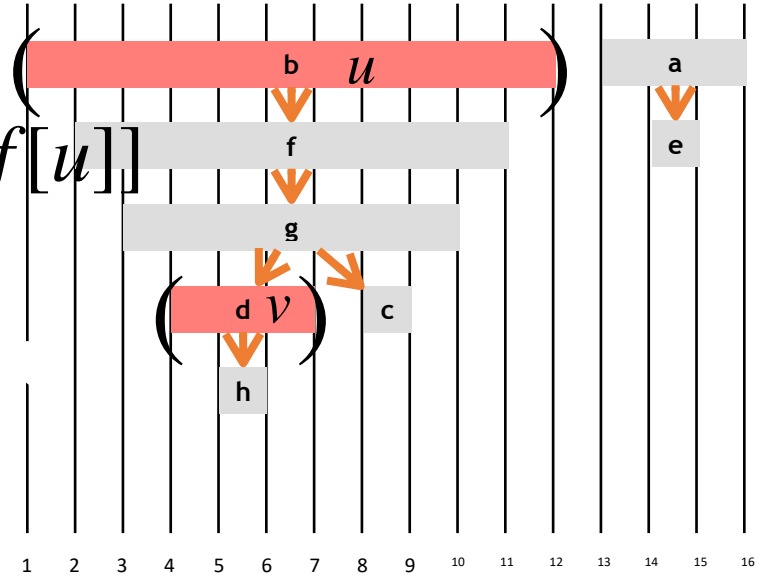
- u is a descendant of v,
  - $[d[u], f[u]]$ is contained entirely within $[d[v], f[v]]$
- v is a descendant of u, or
  - $[d[v], f[v]]$ is contained entirely within $[d[u], f[u]]$
- either u nor v is as descendant of the other



Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices $u, v \in V$, there are three cases of the positions of u and v in the depth-first forest:

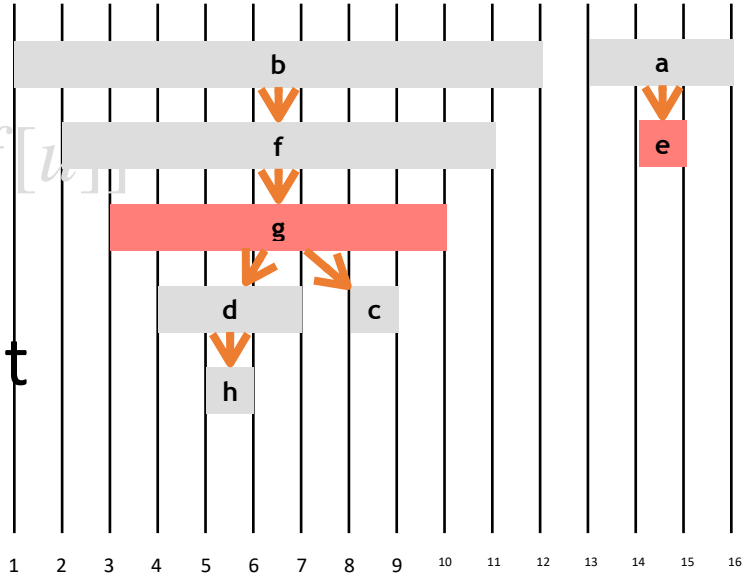
- u is a descendant of v,
  - $[d[u], f[u]]$ is contained entirely within $[d[v], f[v]]$
- v is a descendant of u, or
  - $[d[v], f[v]]$ is contained entirely within $[d[u], f[u]]$
- either u nor v is as descendant of the other



Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices $u, v \in V$, there are three cases of the positions of u and v in the depth-first forest:

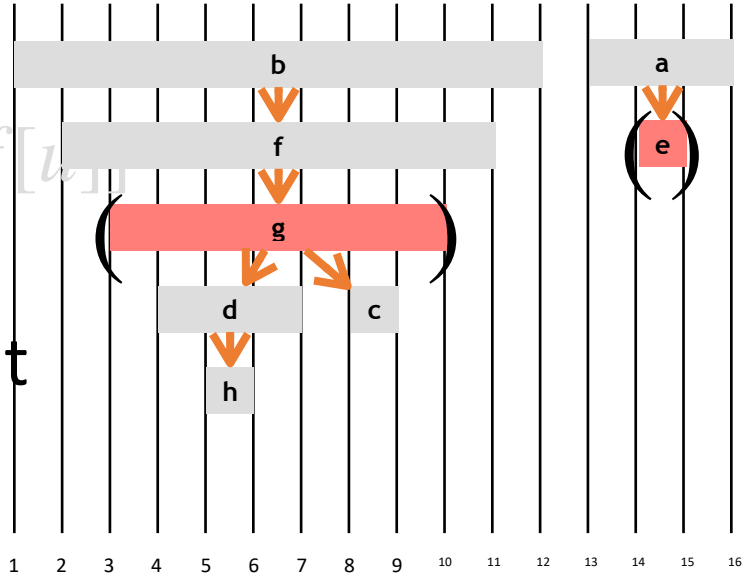
- u is a descendant of v,
  - $[d[u], f[u]]$ is contained entirely within $[d[v], f[v]]$
- v is a descendant of u, or
  - $[d[v], f[v]]$ is contained entirely within $[d[u], f[u]]$
- either u nor v is as descendant of the other
  - $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint



Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices $u, v \in V$, there are three cases of the positions of u and v in the depth-first forest:

- u is a descendant of v,
  - $[d[u], f[u]]$ is contained entirely within $[d[v], f[v]]$
- v is a descendant of u, or
  - $[d[v], f[v]]$ is contained entirely within $[d[u], f[u]]$
- either u nor v is as descendant of the other
  - $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint



Parenthesis Theorem

In any DFS of a graph $G = (V, E)$, for any two vertices $u, v \in V$,
v is a proper descendant of u in the depth-first forest if and only if

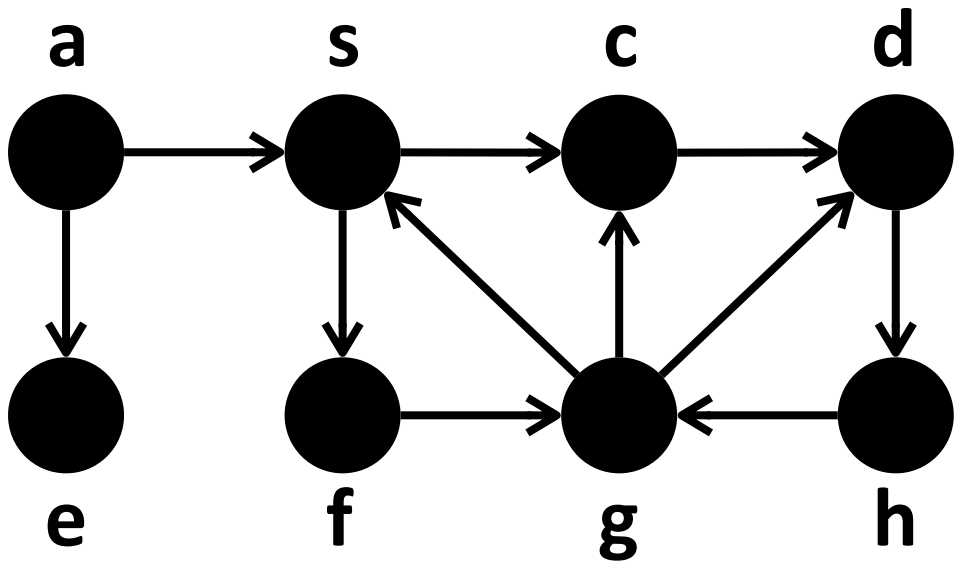
$$d[u] < d[v] < f[v] < f[u]$$

Depth-First Search Properties

- Parenthesis theorem
- Edge classification
- White-path theorem

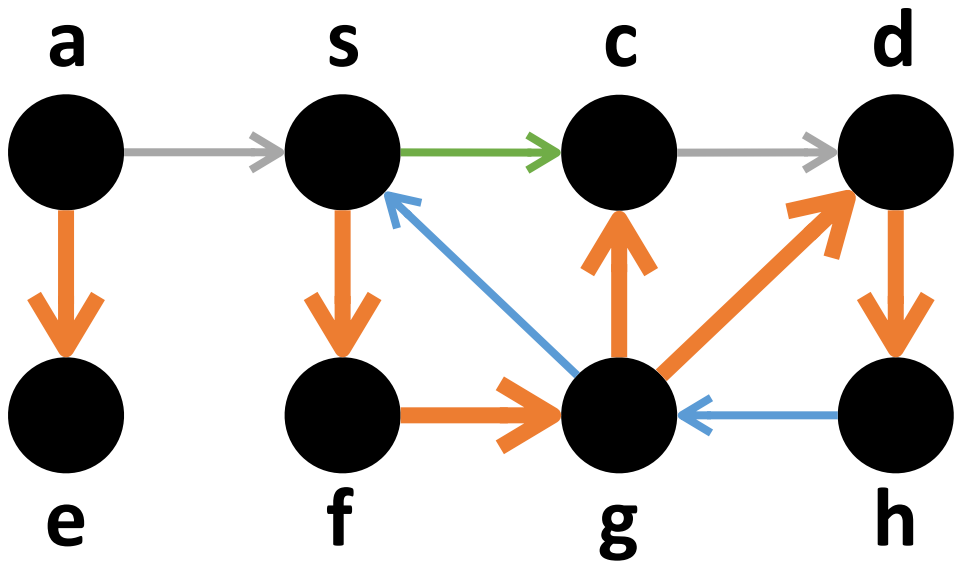
Edges Classification

- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:



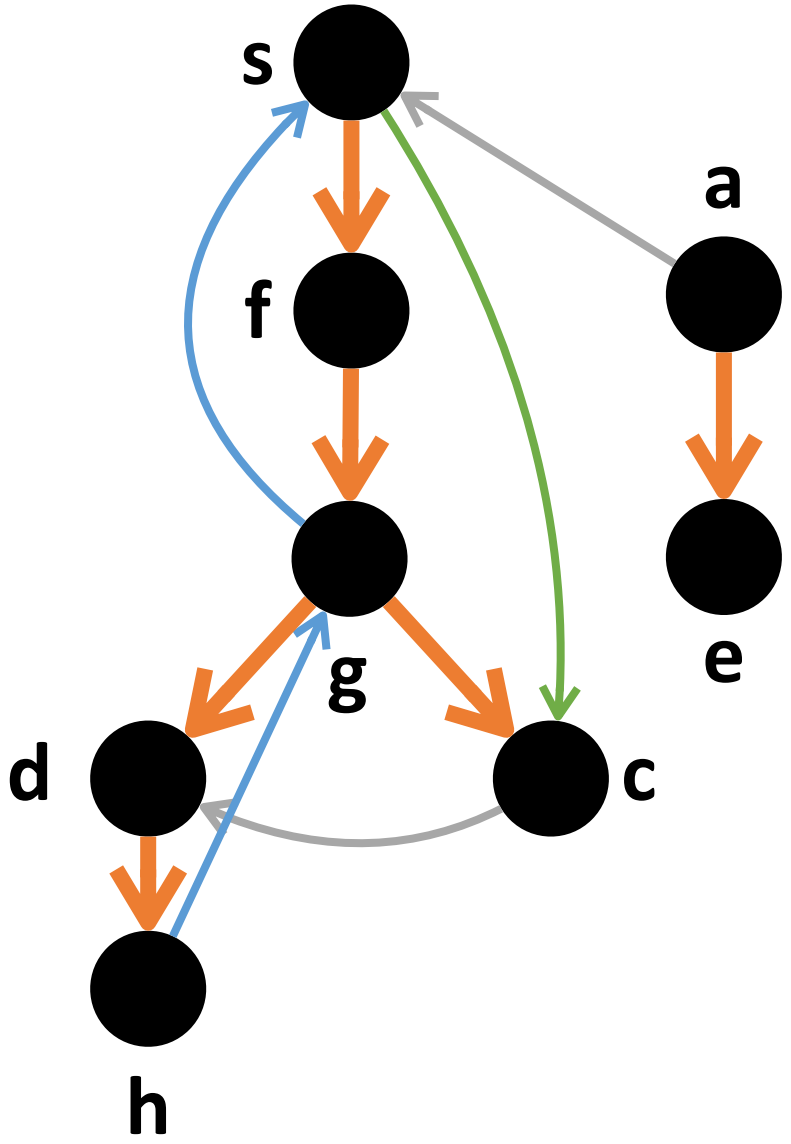
Edges Classification

- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:
  - Tree edges: the edges in E_π
  - Back edges: pointing from a descendant to an ancestor
  - Forward edges: non-tree-edges pointing from an ancestor to a descendant
  - Cross edges: all other edges



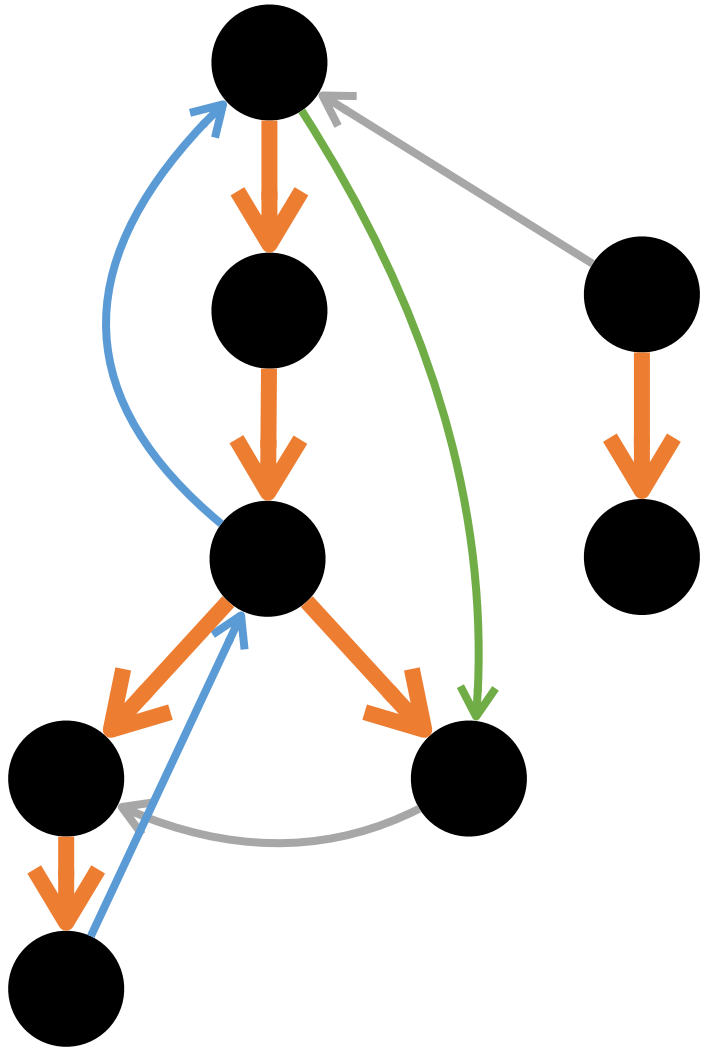
Edges Classification

- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:
  - Tree edges: the edges in E_π
  - Back edges: pointing from a descendant to an ancestor
  - Forward edges: non-tree-edges pointing from an ancestor to a descendant
  - Cross edges: all other edges



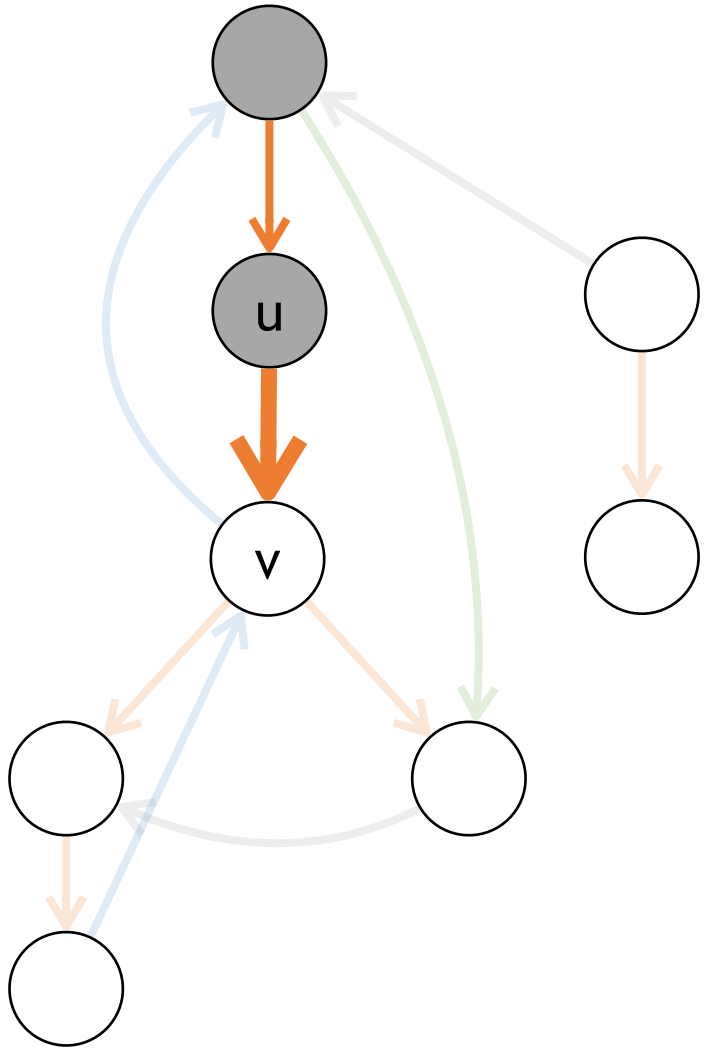
Edges Classification

- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:
  - Tree edges: the edges in E_π
    - $d[u] < d[v] < f[v] < f[u]$



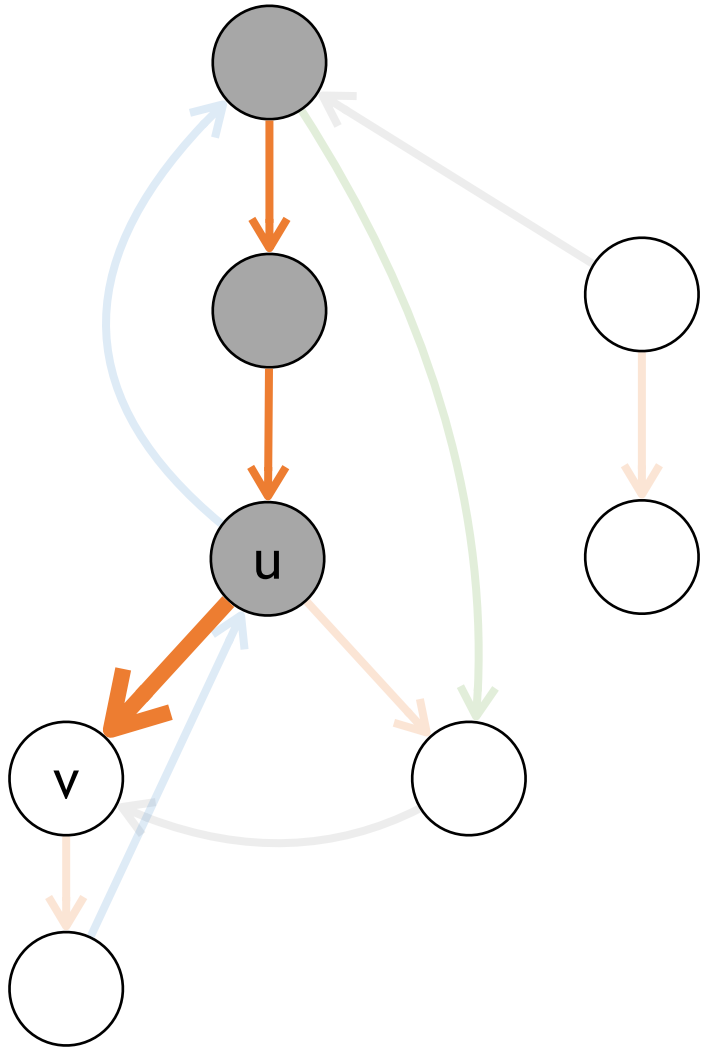
Edges Classification

- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:
  - Tree edges: the edges in E_π
    - $d[u] < d[v] < f[v] < f[u]$



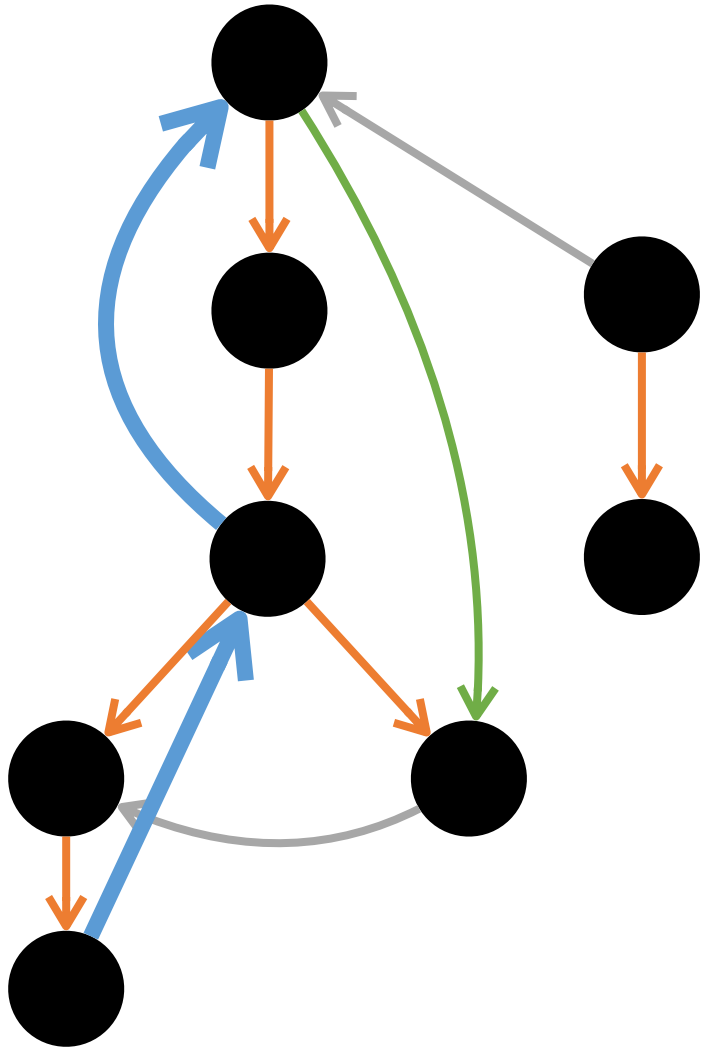
Edges Classification

- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:
  - Tree edges: the edges in E_π
    - $d[u] < d[v] < f[v] < f[u]$



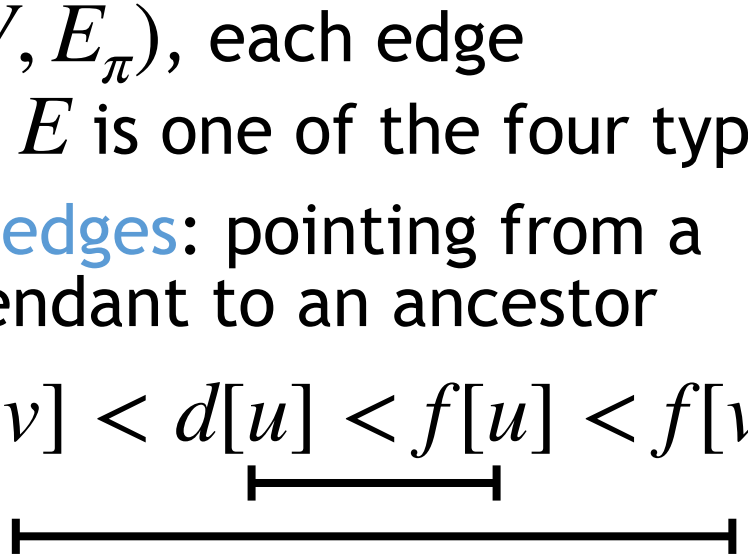
Edges Classification

- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:
  - Back edges: pointing from a descendant to an ancestor
    - $d[v] < d[u] < f[u] < f[v]$

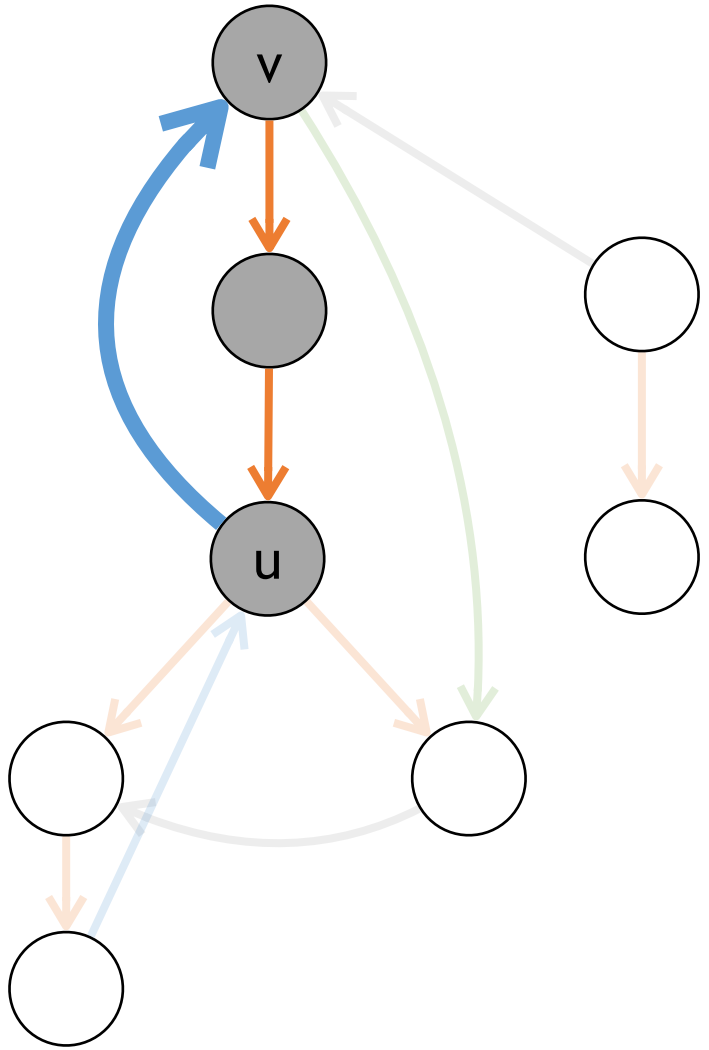


Edges Classification

- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:
  - Back edges: pointing from a descendant to an ancestor
    - $d[v] < d[u] < f[u] < f[v]$

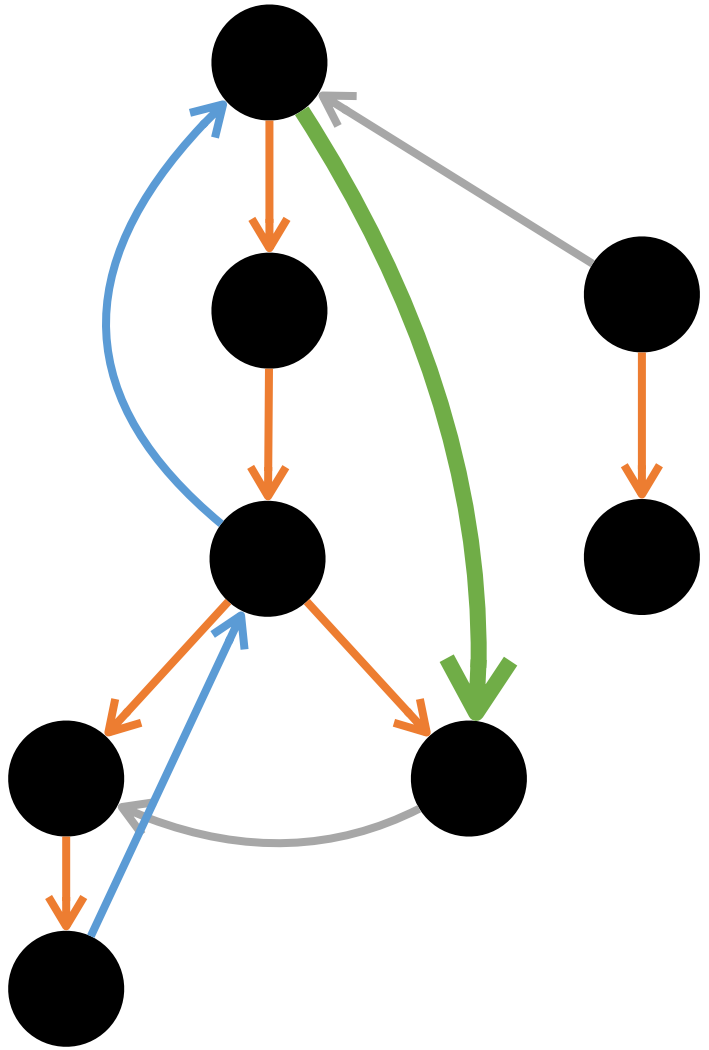


    - 



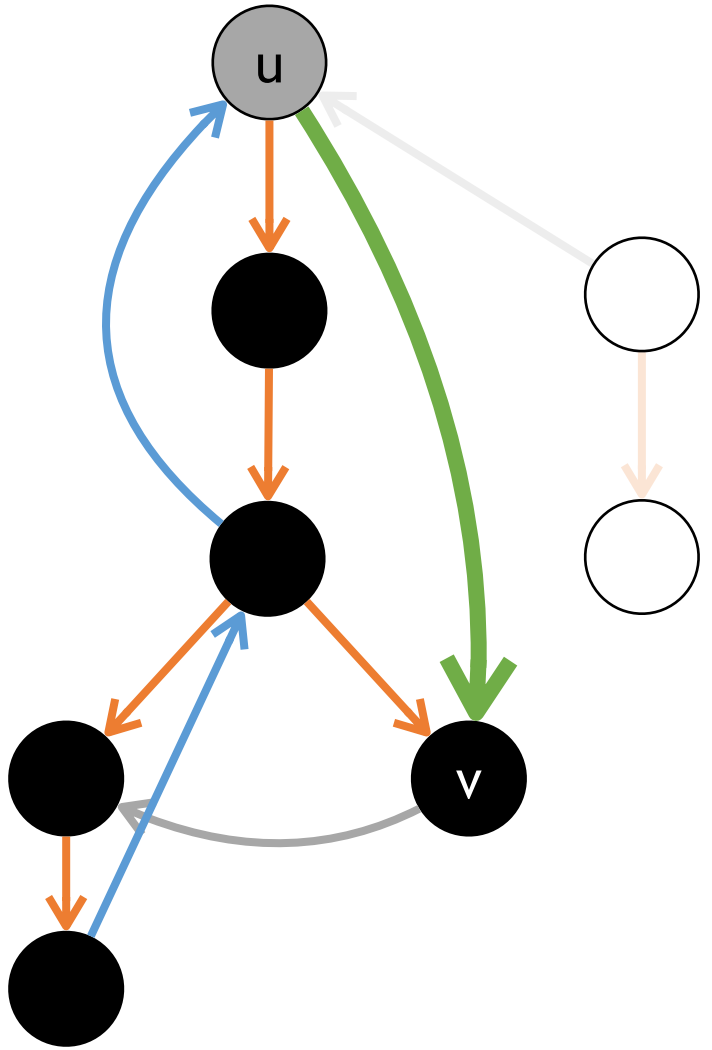
Edges Classification

- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:
  - Forward edges: non-tree-edges pointing from an ancestor to a descendant
    - $d[u] < d[v] < f[v] < f[u]$



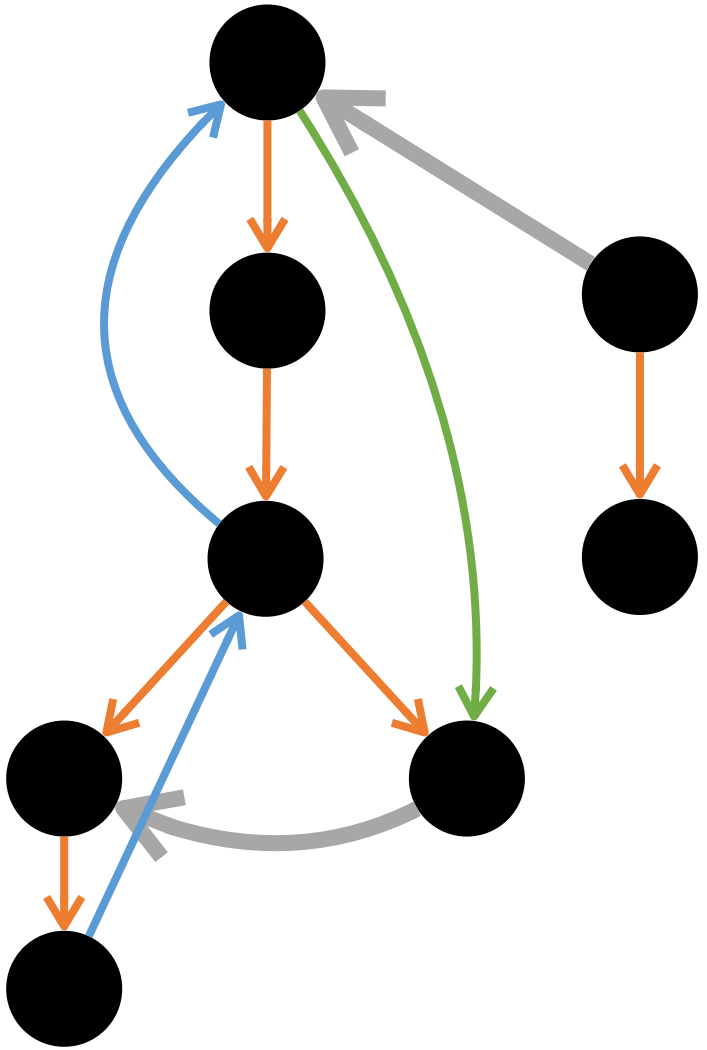
Edges Classification

- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:
  - Forward edges: non-tree-edges pointing from an ancestor to a descendant
    - $d[u] < d[v] < f[v] < f[u]$



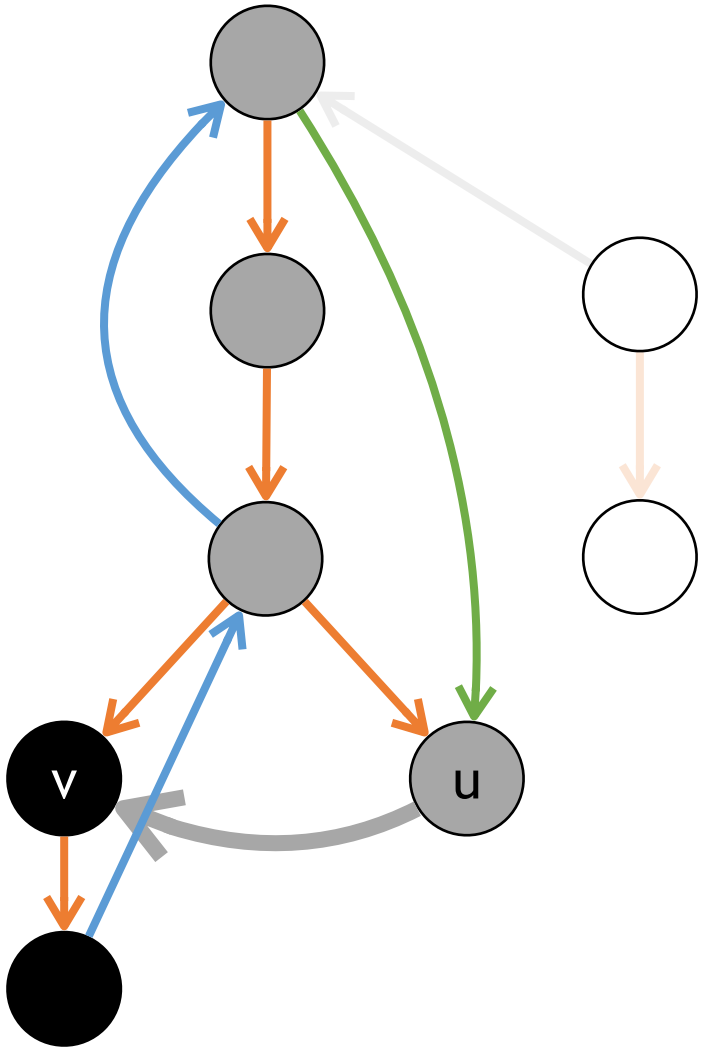
Edges Classification

- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:
  - Cross edges: all other edges
    - $d[v] < f[v] < d[u] < f[u]$



Edges Classification

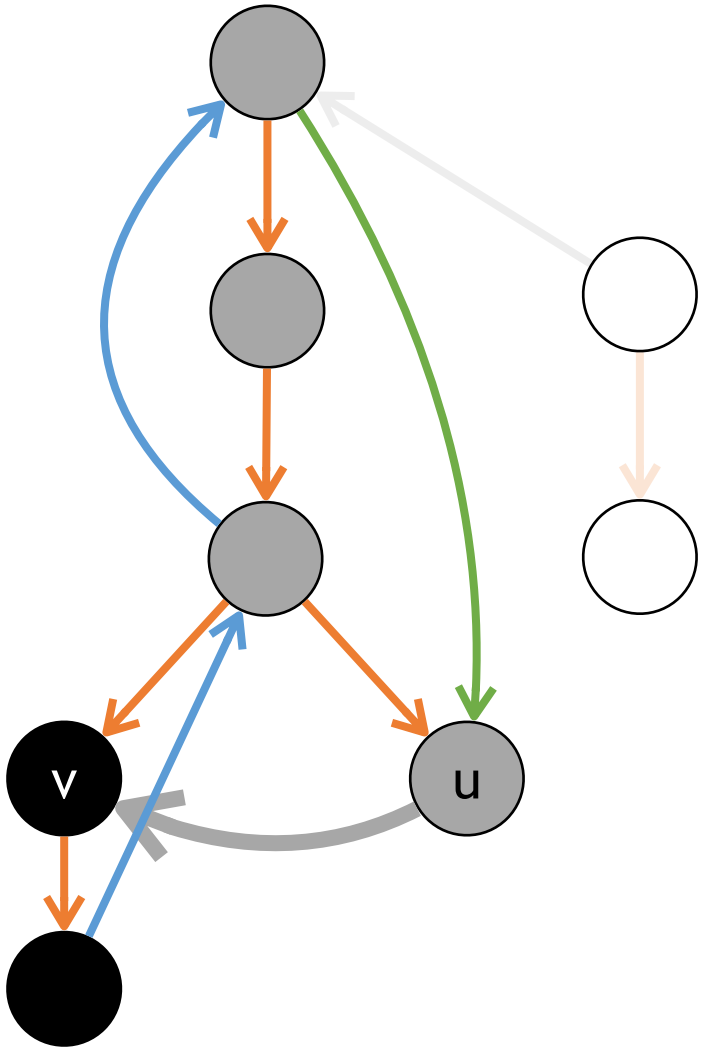
- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:
  - Cross edges: all other edges
    - $d[v] < f[v] < d[u] < f[u]$



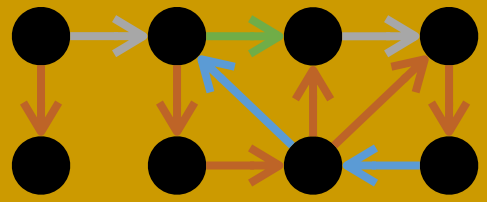
Edges Classification

- According to the Depth-first forest $G_\pi = (V, E_\pi)$, each edge $(u, v) \in E$ is one of the four types:
  - Cross edges: all other edges
    - $d[v] < f[v] < d[u] < f[u]$

    - If $d[u] < d[v]$, (u, v) would be a forward edge



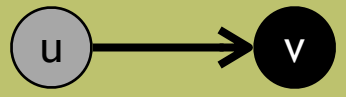
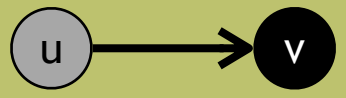
Edges Classification



Type of (u, v)	Relation between d[u], f[u], d[v], f[v]	Intervals [d[u], f[u]] and [d[v], f[v]]	Color of v at the time (u, v) is explored
Tree edge	d[u] < d[v] < f[v] < f[u]	[d[u], f[u]] [d[v], f[v]]	u → v
Back edge	d[v] < d[u] < f[u] < f[v]	[d[u], f[u]] [d[v], f[v]]	u → v
Forward edge	d[u] < d[v] < f[v] < f[u]	[d[u], f[u]] [d[v], f[v]]	u → v
Cross edge	d[v] < f[v] < d[u] < f[u]	[d[u], f[u]] [d[v], f[v]]	u → v

During the DFS, we know which class the visited edge belongs to

Nesting of descendants' Intervals

Type of (u, v)	Relation between d[u], f[u], d[v], f[v]	Intervals [d[u], f[u]] and [d[v], f[v]]	Color of v at the time (u, v) is explored
Tree edge	d[u] < d[v] < f[v] < f[u]	[d[u], f[u]] [d[v], f[v]]	
Back edge	d[v] < d[u] < f[u] < f[v]	[d[u], f[u]] [d[v], f[v]]	
Forward edge	d[u] < d[v] < f[v] < f[u]	[d[u], f[u]] [d[v], f[v]]	
Cross edge	d[v] < f[v] < d[u] < f[u]	[d[u], f[u]] [d[v], f[v]]	

Vertex v is a descendant of vertex u in the depth-first forest **if and only if** $d[u] < d[v] < f[v] < f[u]$

Depth-First Search Properties

- Parenthesis theorem
- Edge classification
- White-path theorem

White-Path Theorem

In a depth-first forest of a graph G, vertex v is a descendant of vertex u **if and only if** at the time that u is discovered by DFS, vertex v can be reached from u along a path consisting entirely of white vertices

White-Path Theorem

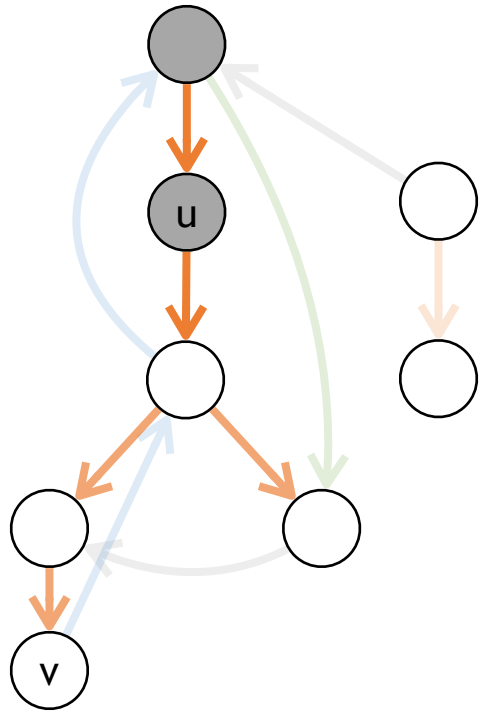
In a depth-first forest of a graph G, vertex v is a descendant of vertex u if and **only if** at the time that u is discovered by DFS, vertex v can be reached from u along a path consisting entirely of white vertices

If vertex v is a descendant of vertex u,
v can be reached from u along a white path

White-Path Theorem

In a depth-first forest of a graph G, vertex v is a descendant of vertex u if and **only if** at the time that u is discovered by DFS, vertex v can be reached from u along a path consisting entirely of white vertices

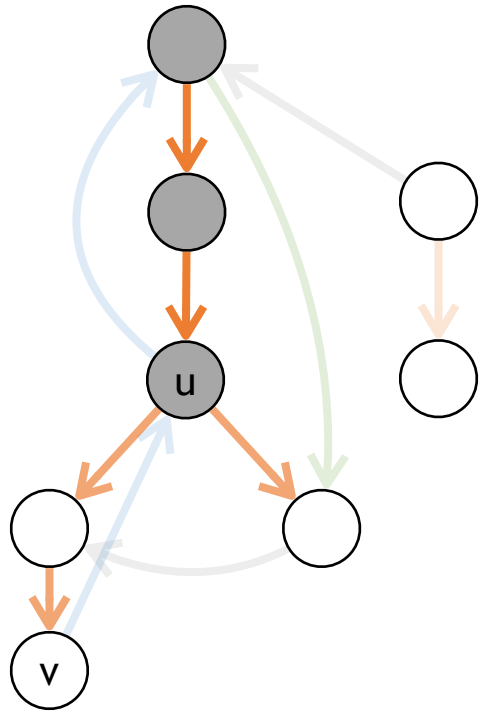
If vertex v is a descendant of vertex u,
v can be reached from u along a white path



White-Path Theorem

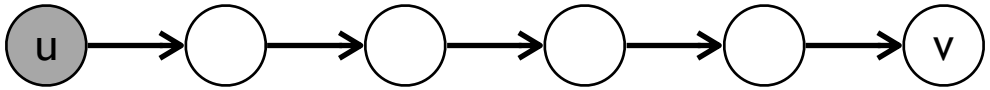
In a depth-first forest of a graph G, vertex v is a descendant of vertex u if and **only if** at the time that u is discovered by DFS, vertex v can be reached from u along a path consisting entirely of white vertices

If vertex v is a descendant of vertex u,
v can be reached from u along a white path



White-Path Theorem

In a depth-first forest of a graph G, vertex v is a descendant of vertex u **if** and only if at the time that u is discovered by DFS, vertex v can be reached from u along a path consisting entirely of white vertices

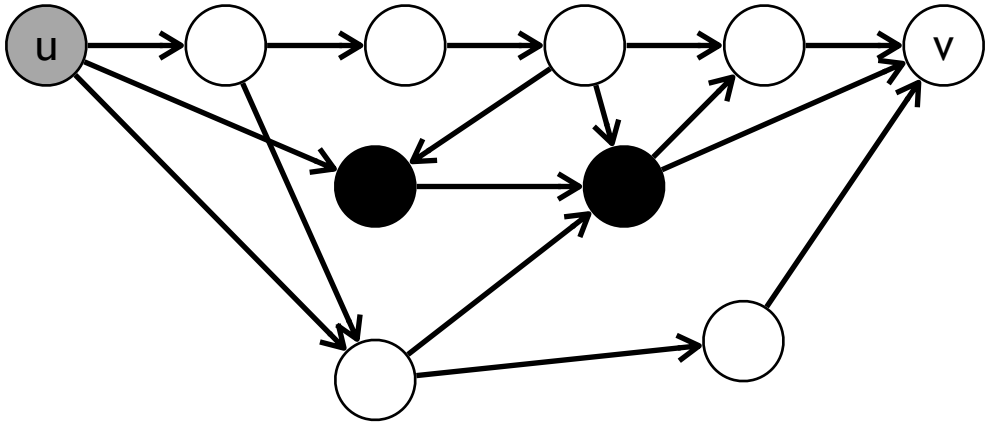


If v can be reached from u along a white path, vertex v is a descendant of vertex u

Proof: exercise

White-Path Theorem

In a depth-first forest of a graph G, vertex v is a descendant of vertex u **if** and only if at the time that u is discovered by DFS, vertex v can be reached from u along a path consisting entirely of white vertices



If v can be reached from u along a white path, vertex v is a descendant of vertex u

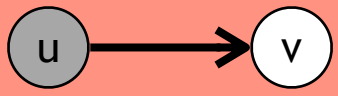
Proof: exercise

Application: Cycle Detection

Given a directed graph G, there is a directed cycle **if and only if** a DFS of G yields at least one back edge.

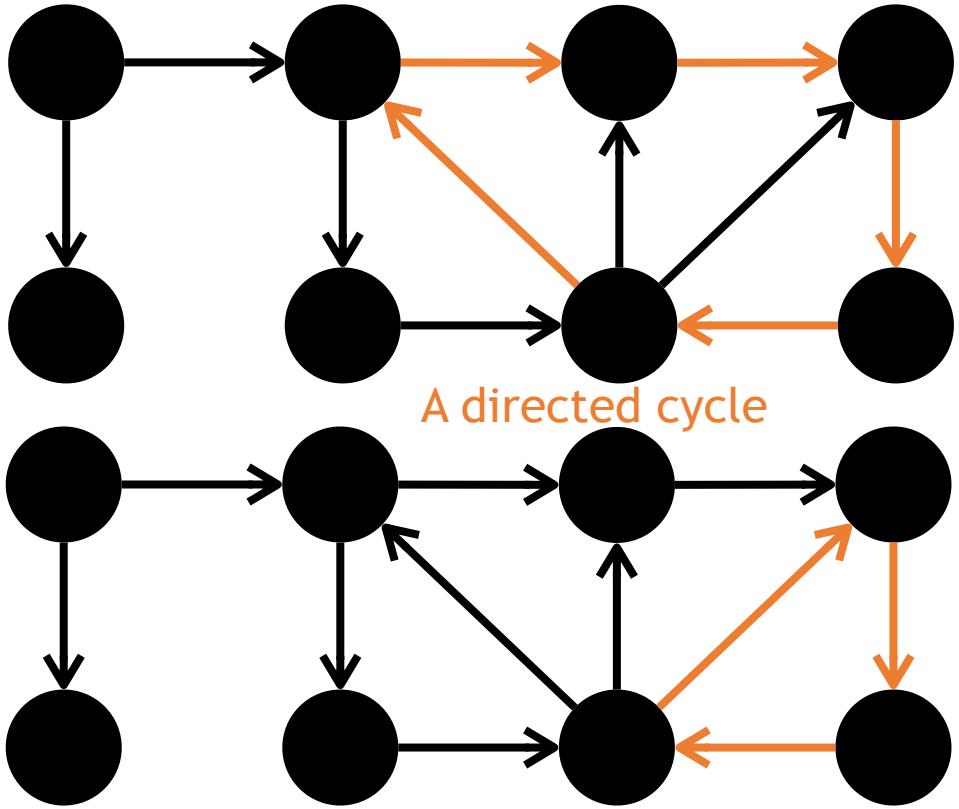
Application: Cycle Detection

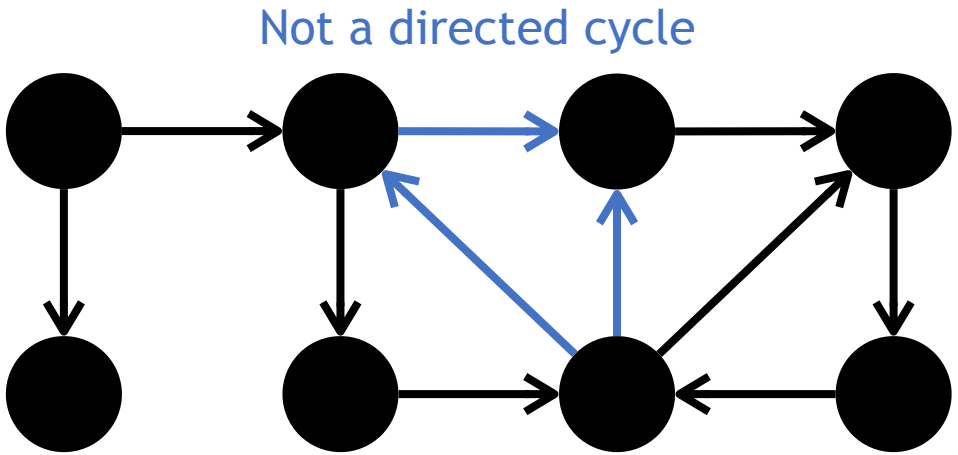
Given a directed graph G, there is a directed cycle **if and only if** a DFS of G yields at least one back edge.

Type of (u, v)	Relation between d[u], f[u], d[v], f[v]	Intervals [d[u], f[u]] and [d[v], f[v]]	Color of v at the time (u, v) is explored
Tree edge	d[u] < d[v] < f[v] < f[u]	[d[u], f[u]] [d[v], f[v]]	
Back edge	d[v] < d[u] < f[u] < f[v]	[d[u], f[u]] [d[v], f[v]]	
Forward edge	d[u] < d[v] < f[v] < f[u]	[d[u], f[u]] [d[v], f[v]]	
Cross edge	d[v] < f[v] < d[u] < f[u]	[d[u], f[u]] [d[v], f[v]]	

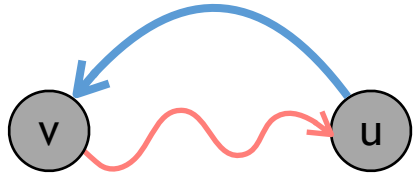
Application: Cycle Detection

Given a directed graph G, there is a directed cycle **if and only if** a DFS of G yields at least one back edge.





Application: Cycle Detection

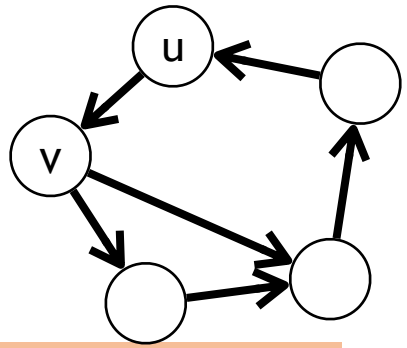


Given a directed graph G, there is a directed cycle **if** and only if a DFS of G yields at least one back edge.

<Proof Idea>

⇐: Assume there is a back edge (u, v). The Tree edges from v to u together with (u, v) form a directed cycle.

Application: Cycle Detection



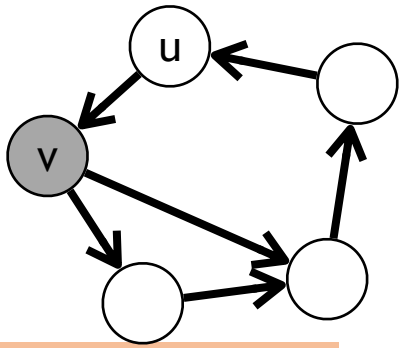
Given a directed graph G, there is a directed cycle if **and only if** a DFS of G yields at least one back edge.

<Proof Idea>

$\Leftarrow$: Assume there is a back edge (u, v). The Tree edges from v to u together with (u, v) form a directed cycle.

$\Rightarrow$: Suppose there is a directed cycle $c = [v, u_1, u_2, \cdots, u_k, u, v]$

Application: Cycle Detection



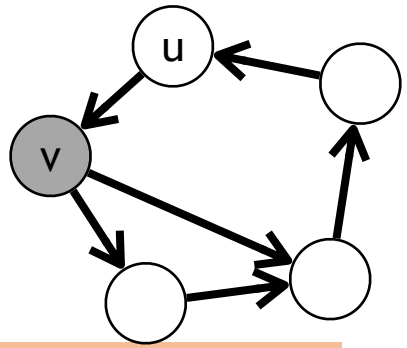
Given a directed graph G, there is a directed cycle if **and only if** a DFS of G yields at least one back edge.

<Proof Idea>

$\Leftarrow$: Assume there is a back edge (u, v). The Tree edges from v to u together with (u, v) form a directed cycle.

$\Rightarrow$: Suppose there is a directed cycle $c = [v, u_1, u_2, \cdots, u_k, u, v]$ and let v be the first vertex discovered by DFS.

Application: Cycle Detection



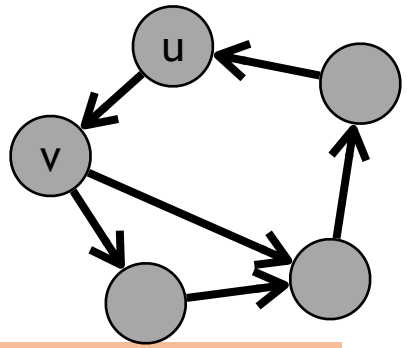
Given a directed graph G, there is a directed cycle if **and only if** a DFS of G yields at least one back edge.

<Proof Idea>

$\Leftarrow$: Assume there is a back edge (u, v). The Tree edges from v to u together with (u, v) form a directed cycle.

$\Rightarrow$: Suppose there is a directed cycle $c = [v, u_1, u_2, \cdots, u_k, u, v]$ and let v be the first vertex discovered by DFS. By White-Path theorem, u is a descendent of v in the depth-first forest.

Application: Cycle Detection



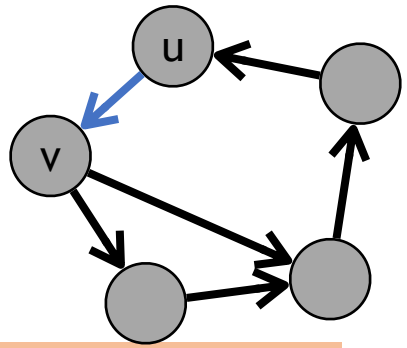
Given a directed graph G, there is a directed cycle if **and only if** a DFS of G yields at least one back edge.

<Proof Idea>

$\Leftarrow$: Assume there is a back edge (u, v). The Tree edges from v to u together with (u, v) form a directed cycle.

$\Rightarrow$: Suppose there is a directed cycle $c = [v, u_1, u_2, \cdots, u_k, u, v]$ and let v be the first vertex discovered by DFS. By White-Path theorem, u is a descendent of v in the depth-first forest. Since u and v are both gray when (u, v) is explored, (u, v) is a back edge.

Application: Cycle Detection



Given a directed graph G, there is a directed cycle if **and only if** a DFS of G yields at least one back edge.

<Proof Idea>

$\Leftarrow$: Assume there is a back edge (u, v). The Tree edges from v to u together with (u, v) form a directed cycle.

$\Rightarrow$: Suppose there is a directed cycle $c = [v, u_1, u_2, \cdots, u_k, u, v]$ and let v be the first vertex discovered by DFS. By White-Path theorem, u is a descendent of v in the depth-first forest. Since u and v are both gray when (u, v) is explored, (u, v) is a back edge.

What's Happening

- Use the edge-classification, we can detect if there is a directed cycle in the graph by checking if there is a back edge.
- The correctness proof uses the White-Path theorem.

Outline

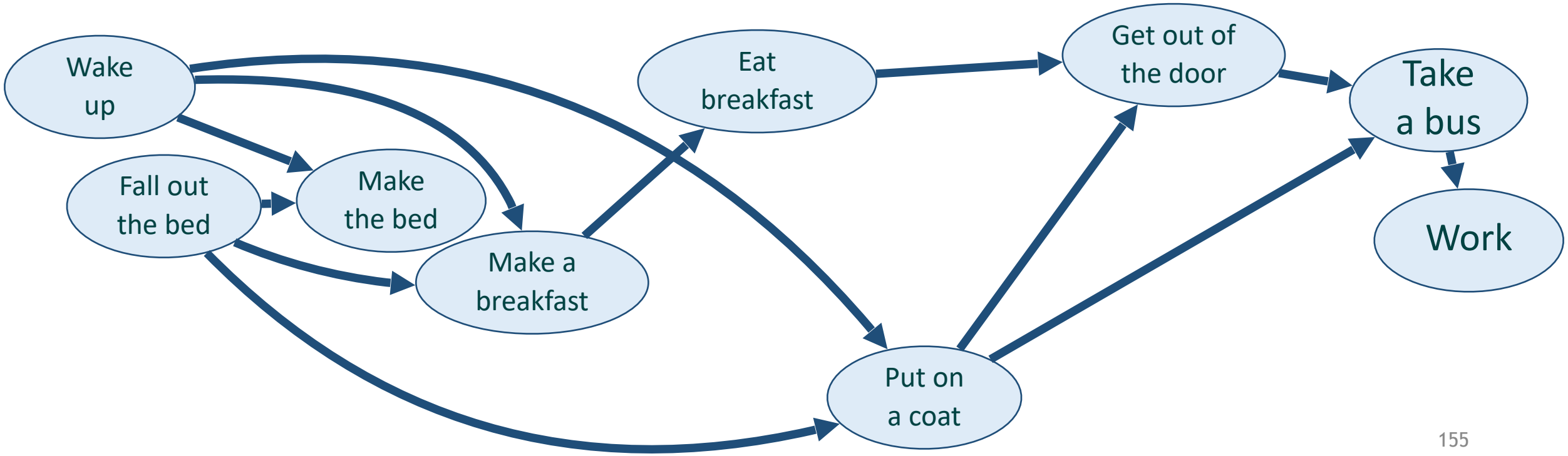
- Graph representations
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
- Two applications on DFS:
  - Topological sort
  - Strongly connected components

Directed Acyclic Graph

Definition: A **directed acyclic graph (DAG)** is a directed graph which does not have any directed cycles

Directed Acyclic Graph

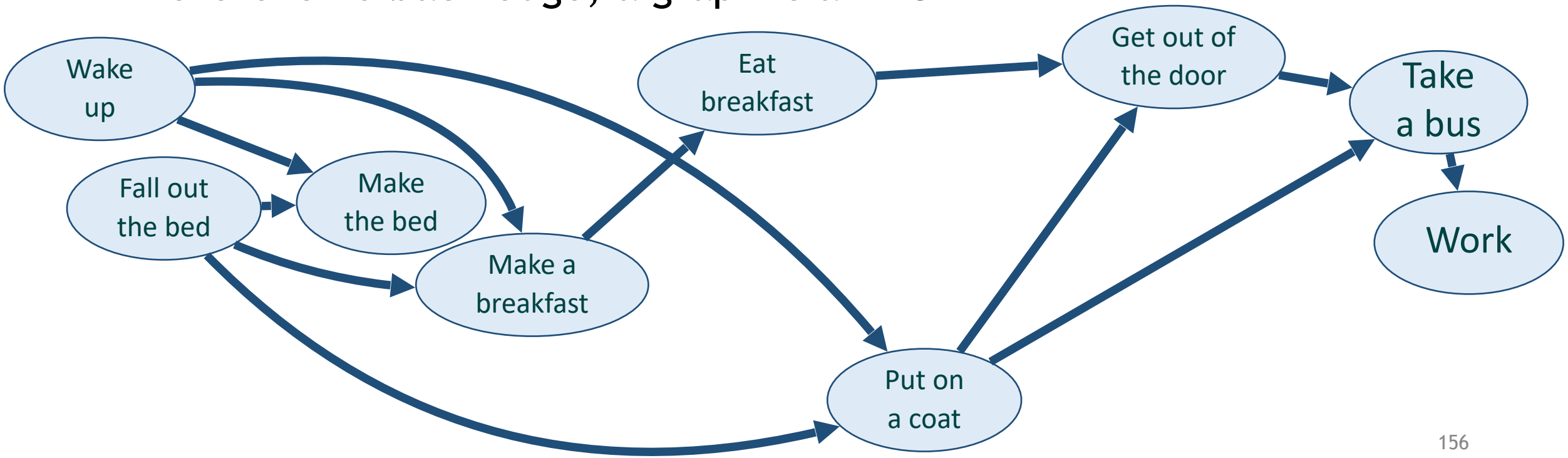
Definition: A **directed acyclic graph (DAG)** is a directed graph which does not have any directed cycles



Directed Acyclic Graph

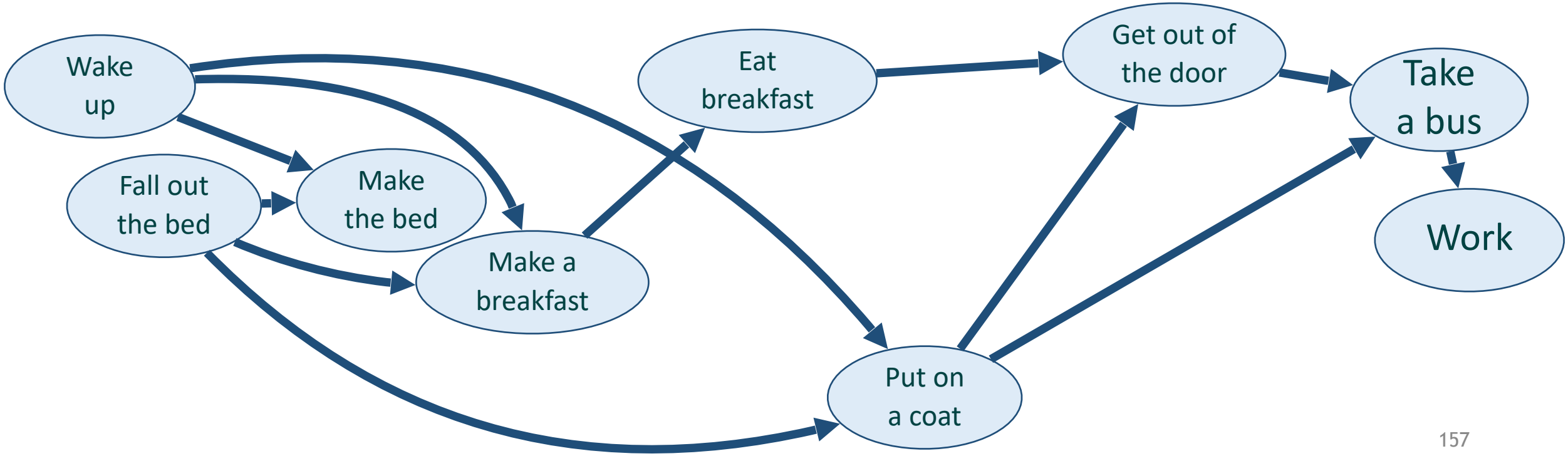
Definition: A **directed acyclic graph (DAG)** is a directed graph which does not have any directed cycles

- If there is no back edge, a graph is a DAG



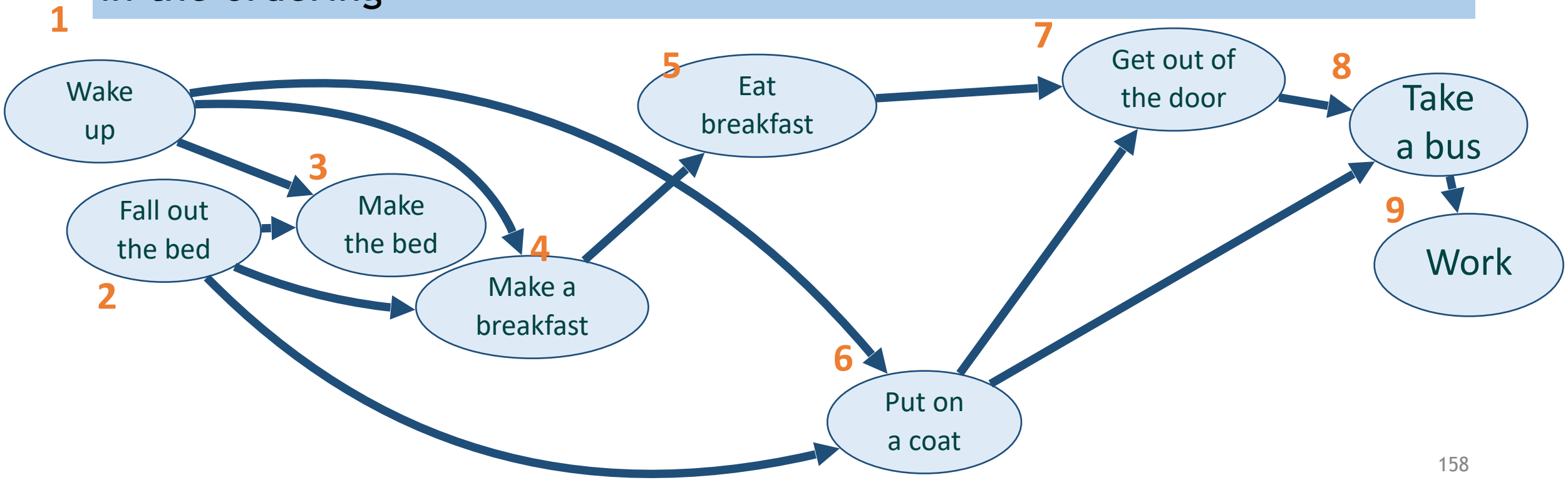
Topological Sort

Definition: A **topological sort** of a DAG $G = (V, E)$ is a linear ordering of all its vertices such that for any edge $(u, v) \in E$, u appears before v in the ordering



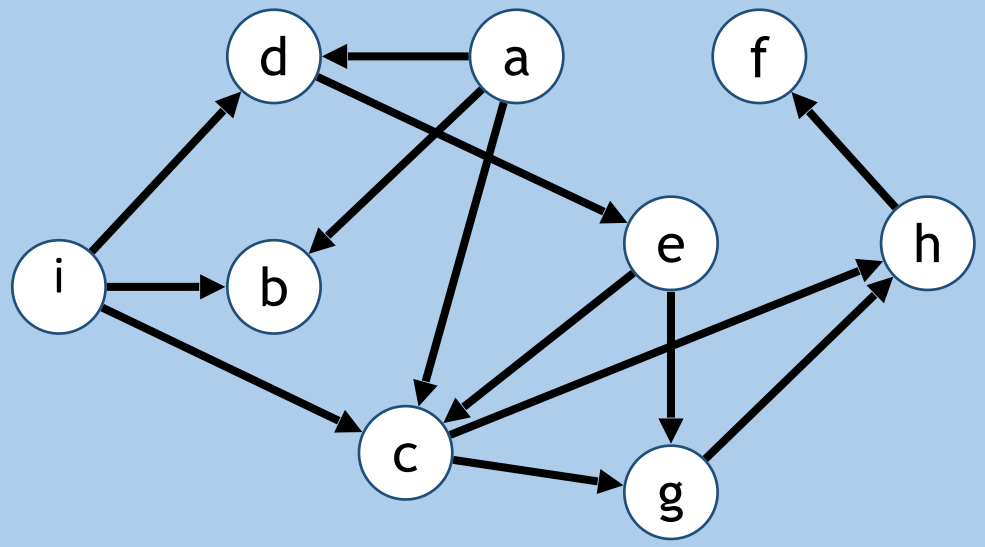
Topological Sort

Definition: A **topological sort** of a DAG $G = (V, E)$ is a linear ordering of all its vertices such that for any edge $(u, v) \in E$, u appears before v in the ordering



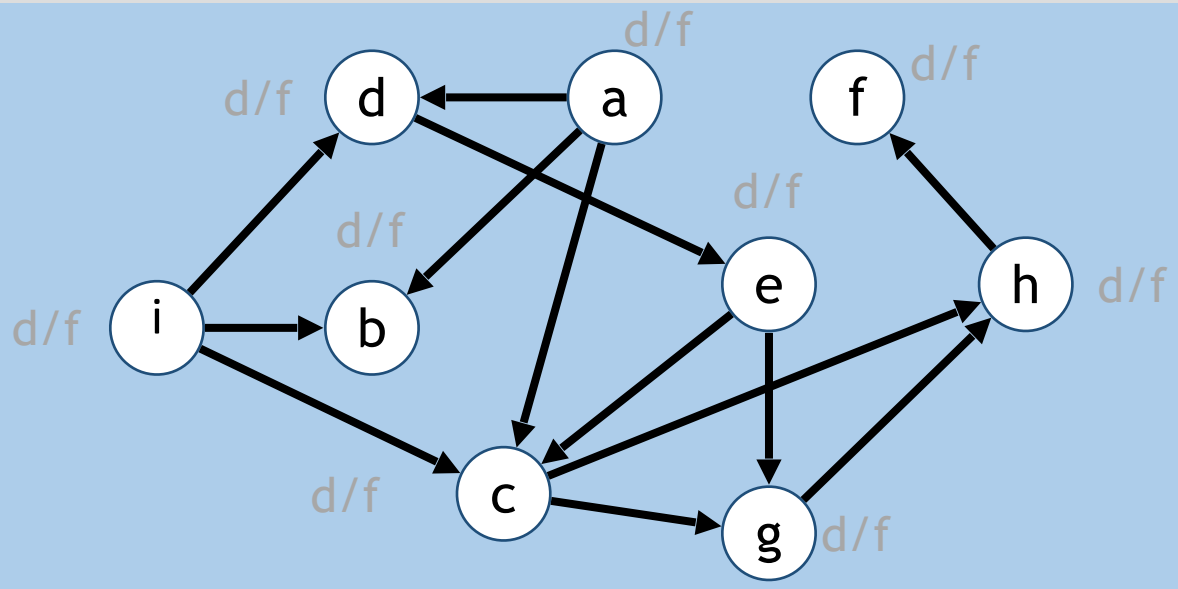
Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



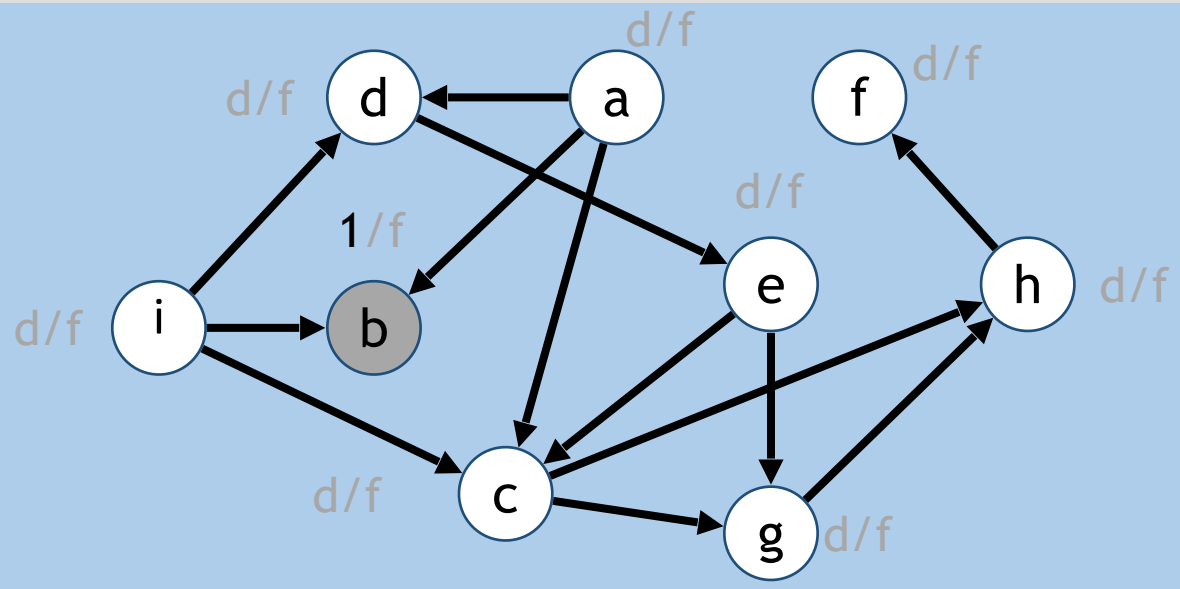
Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



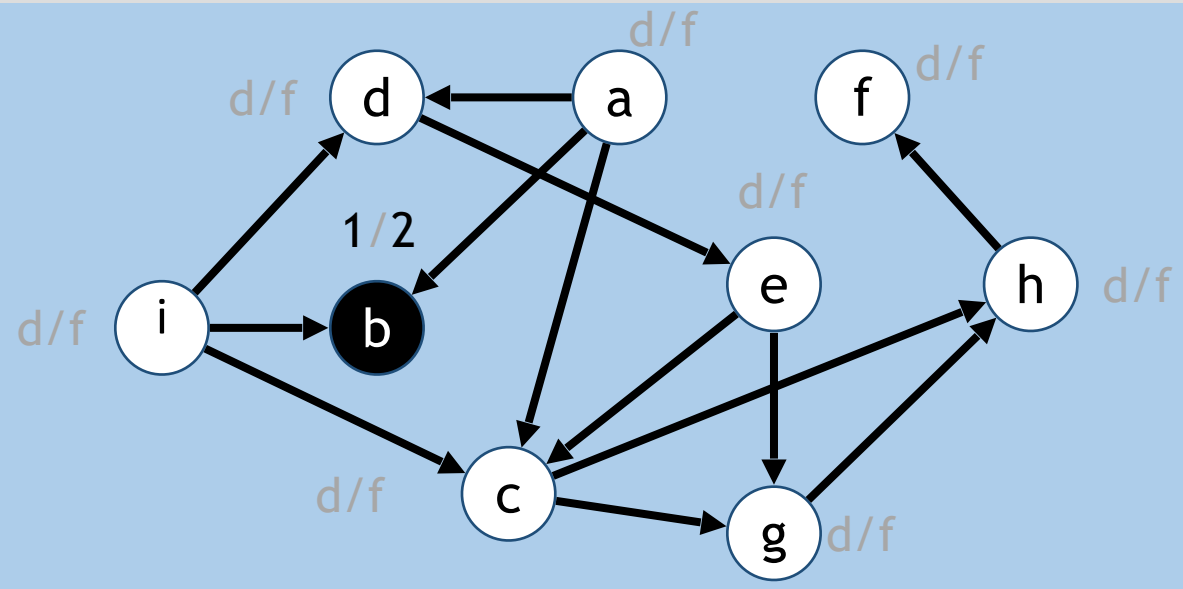
Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

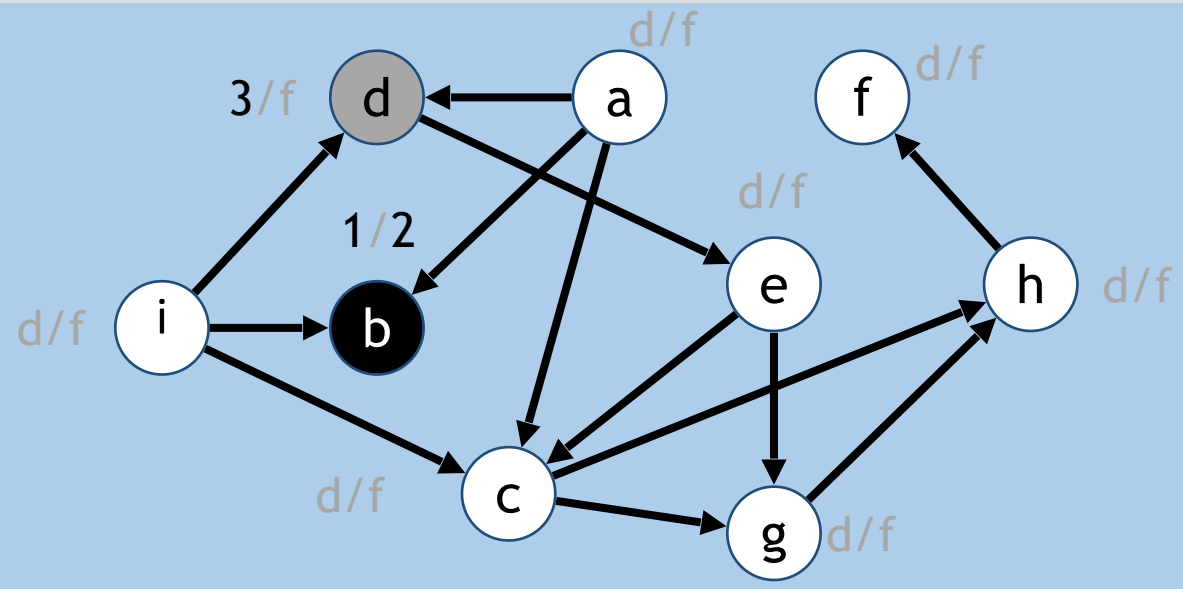


L_{sort}

b

Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

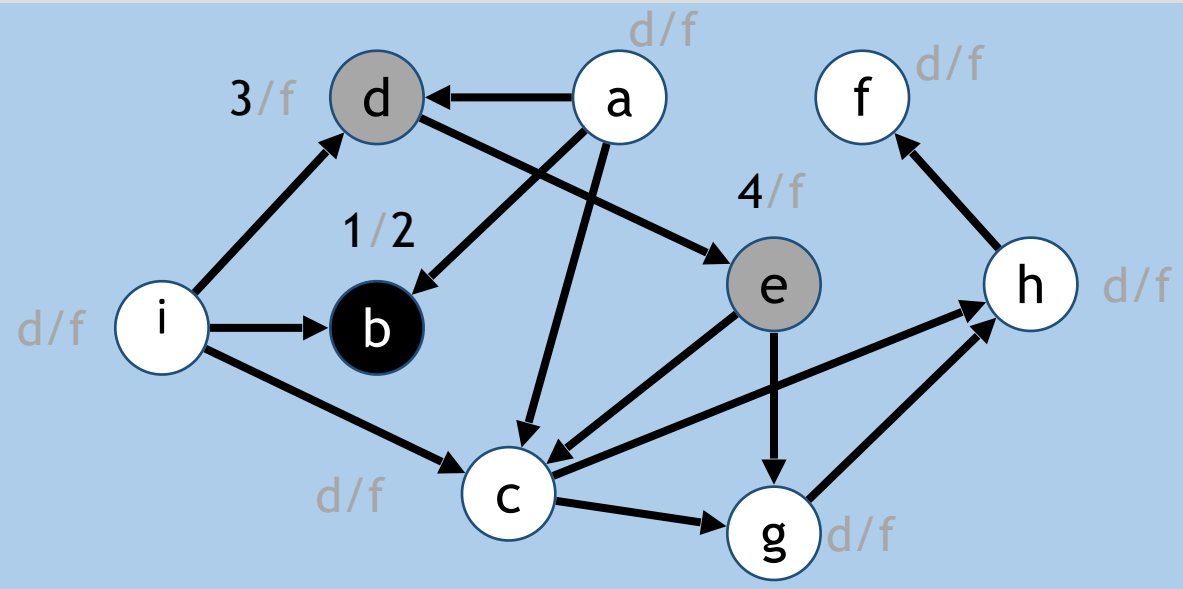


L_{sort}

b

Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

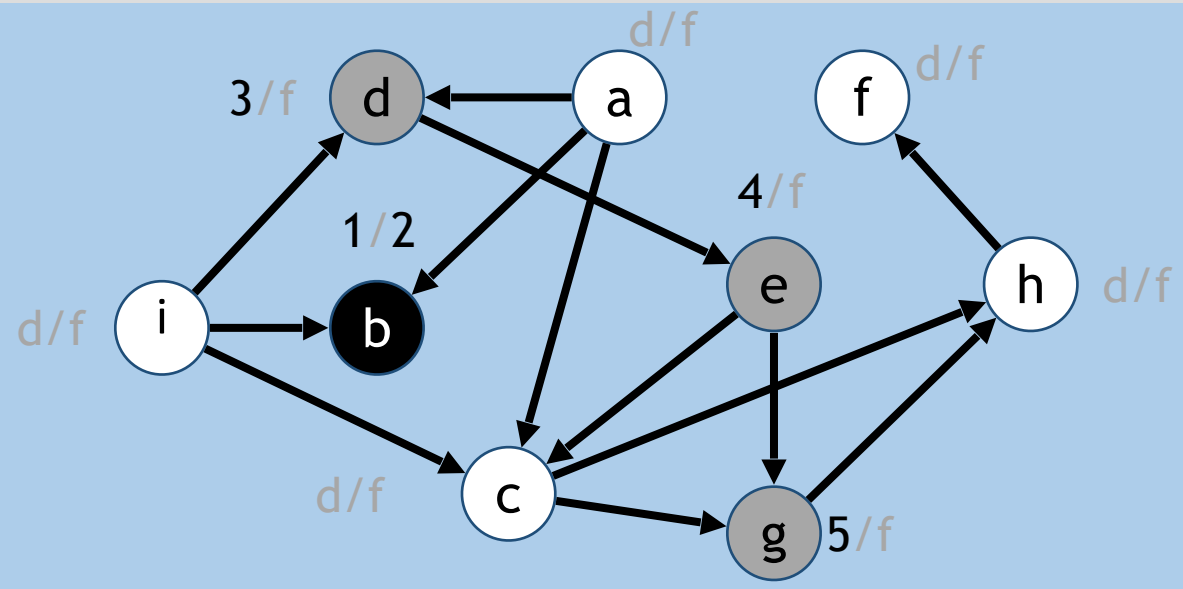


L_{sort}

b

Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

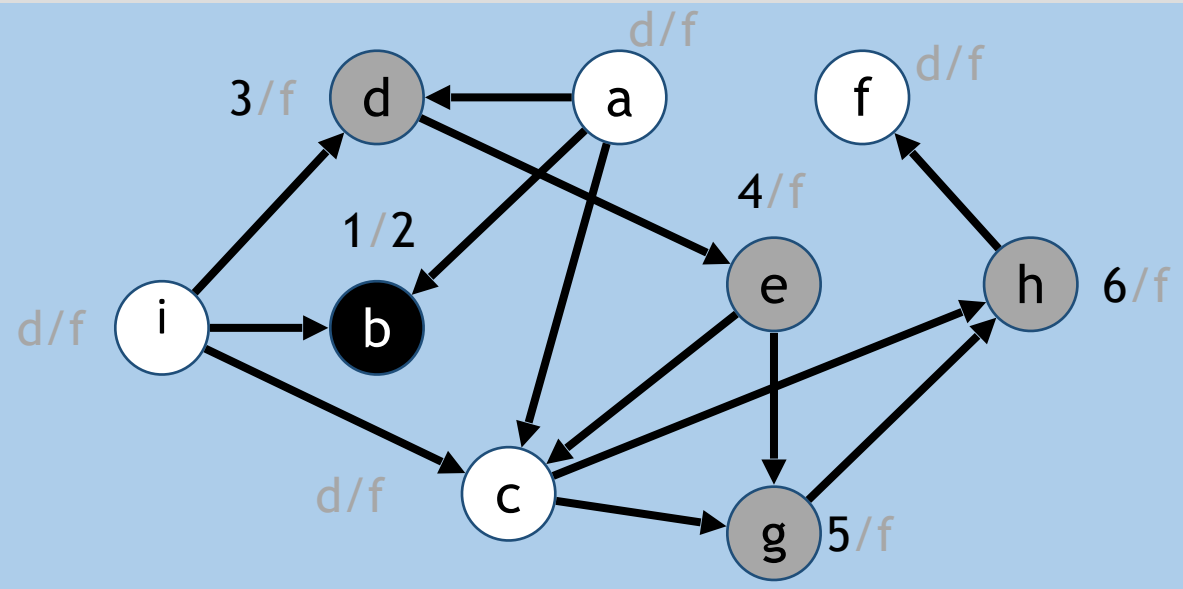


L_{sort}

b

Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

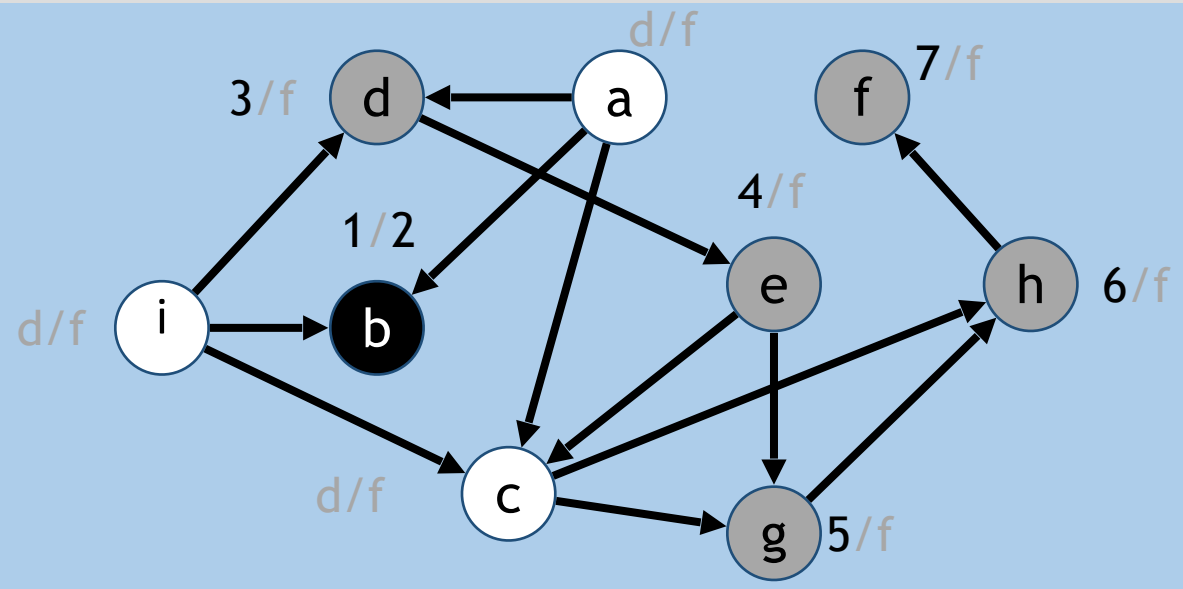


L_{sort}

b

Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

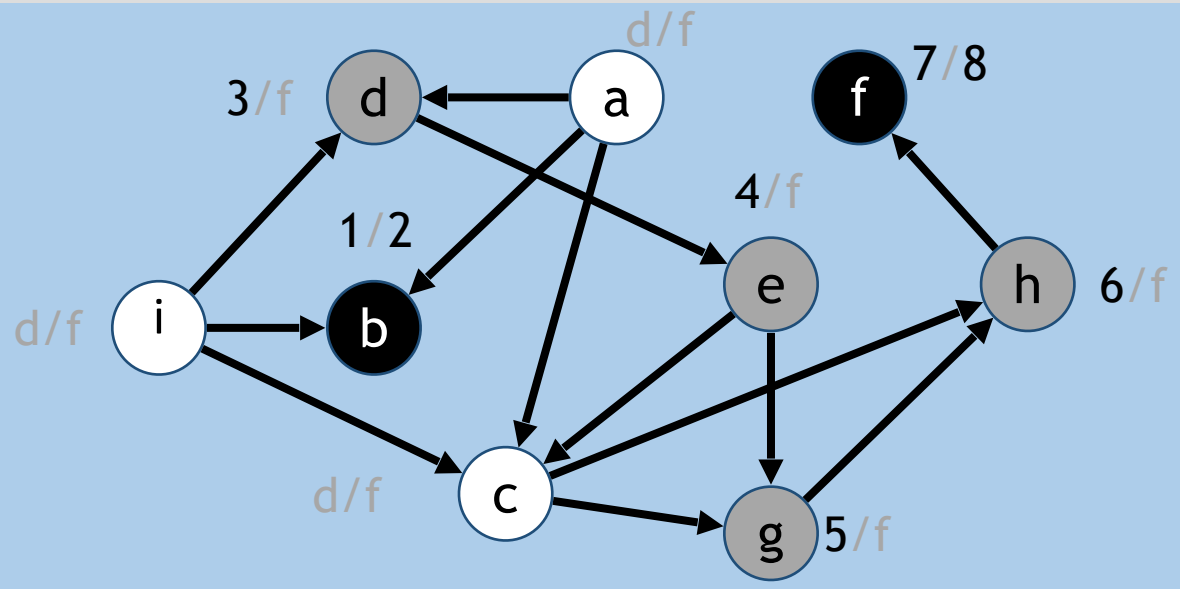


L_{sort}

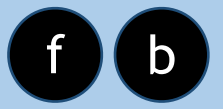
b

Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

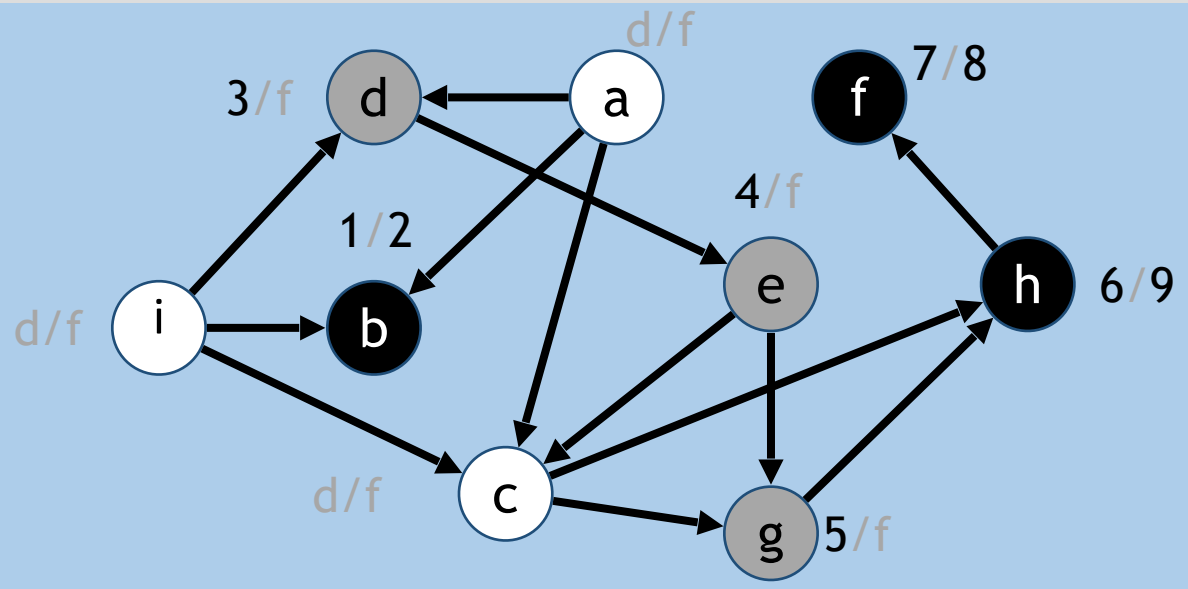


L_{sort}



Topological Sort

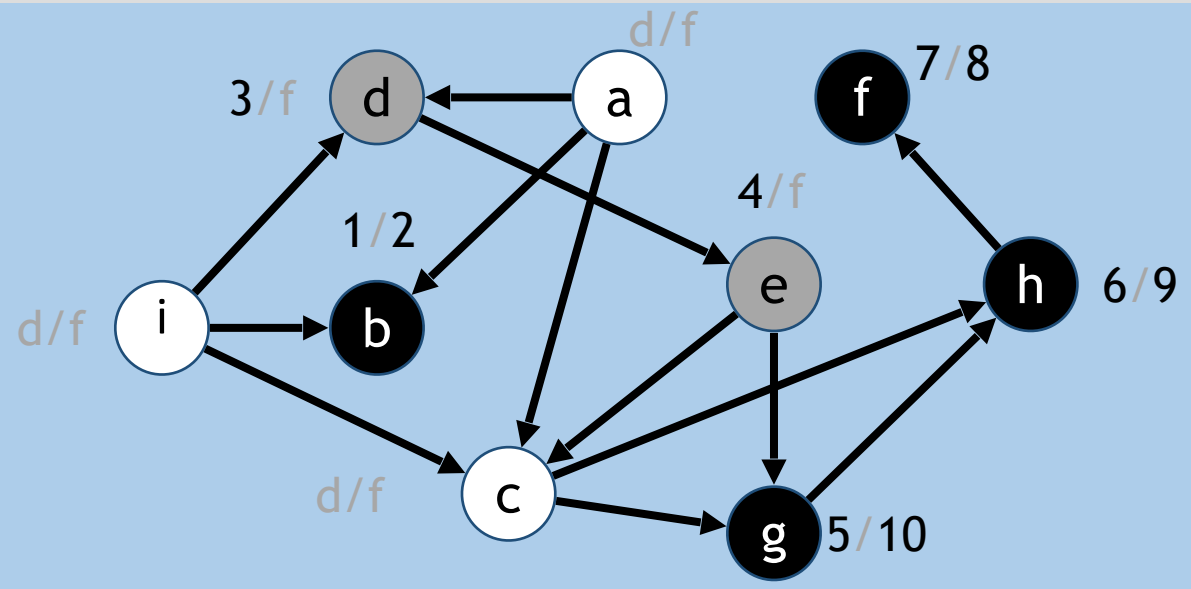
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



L_{sort}

Topological Sort

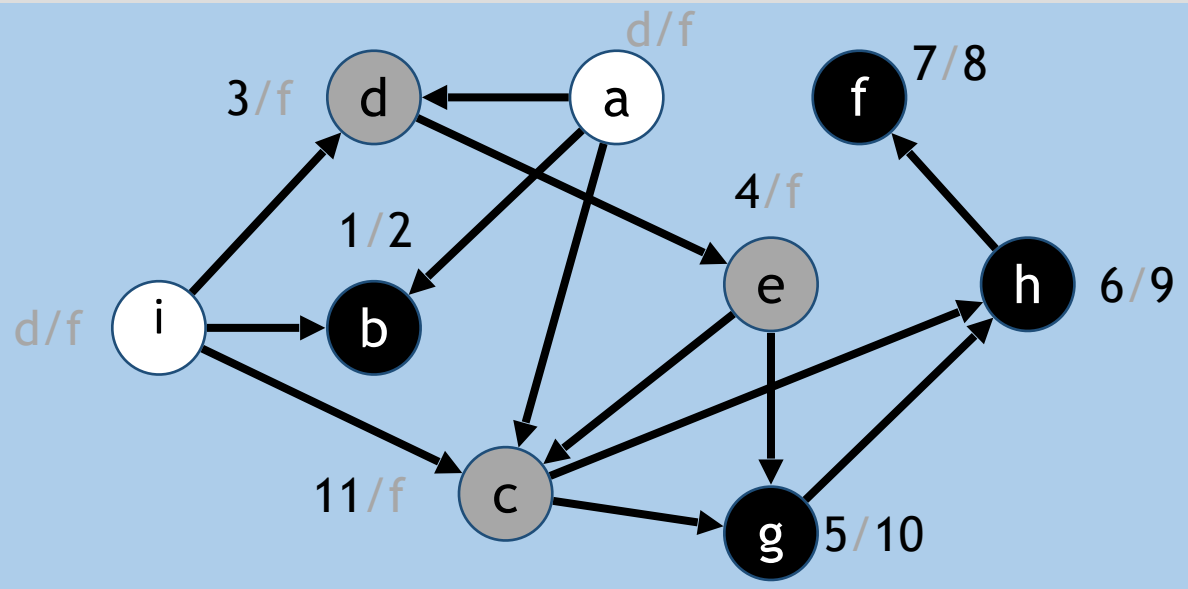
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



L_{sort}

Topological Sort

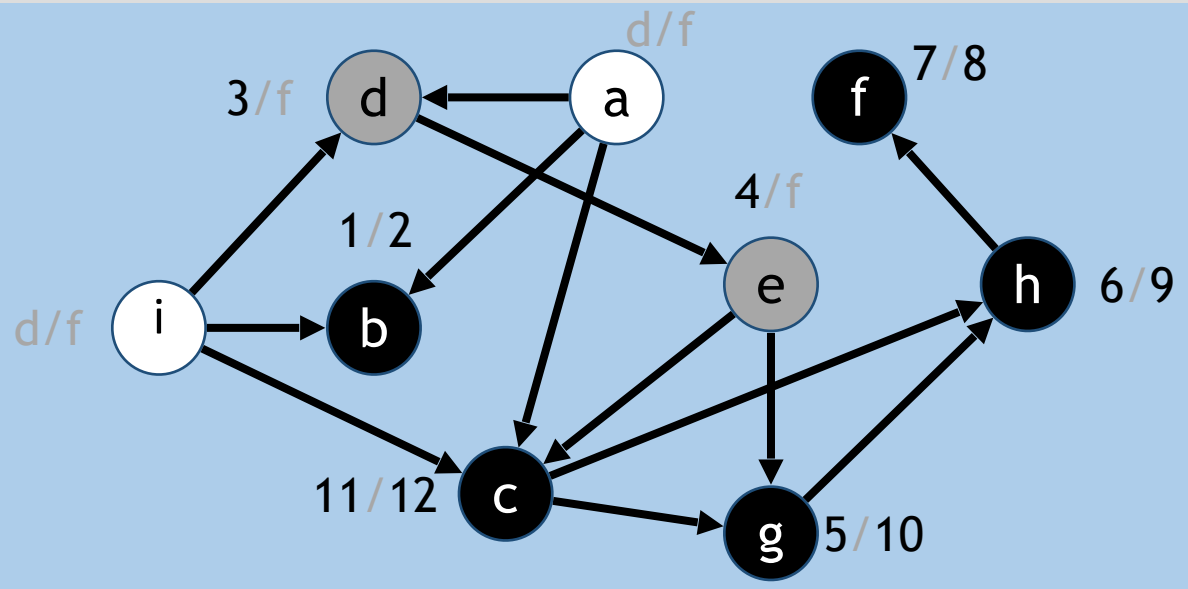
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



L_{sort}

Topological Sort

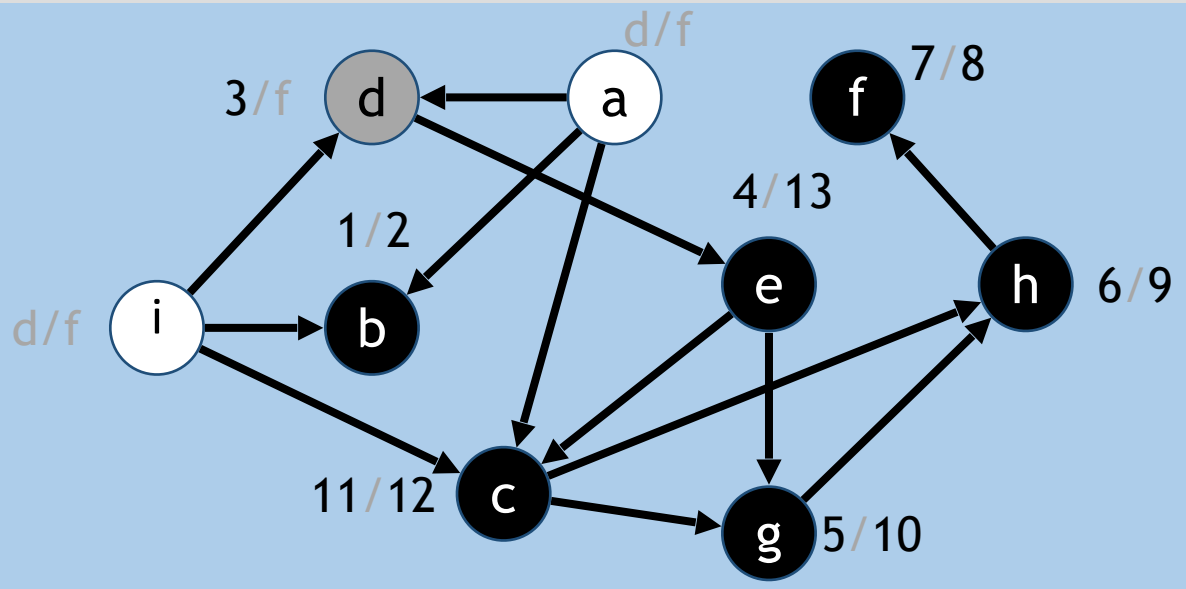
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



L_{sort}

Topological Sort

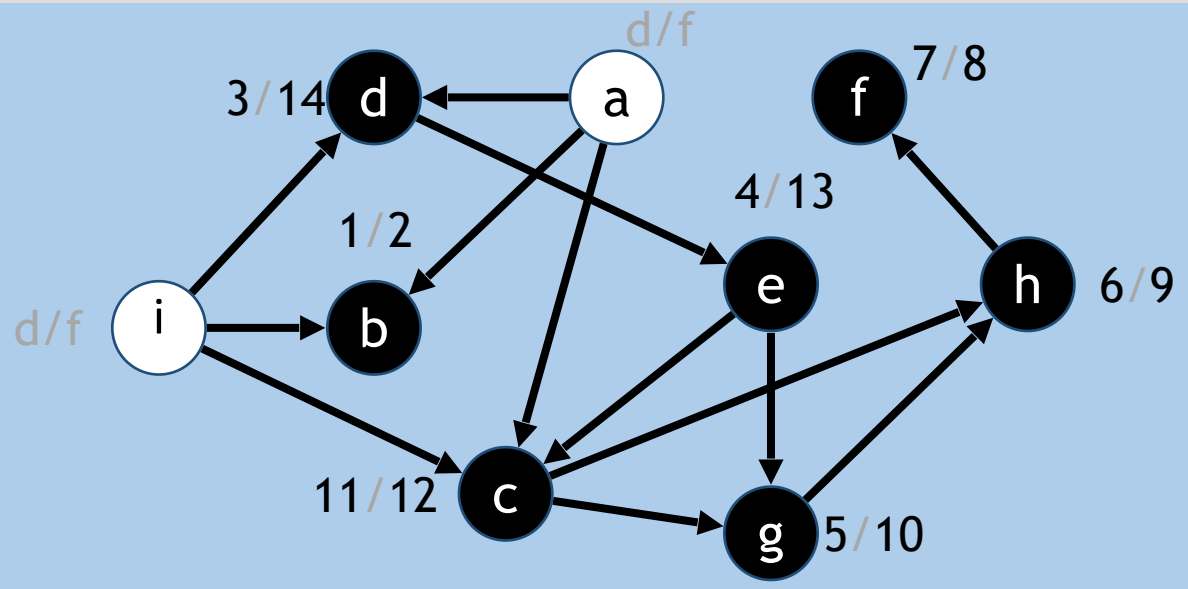
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



L_{sort}

Topological Sort

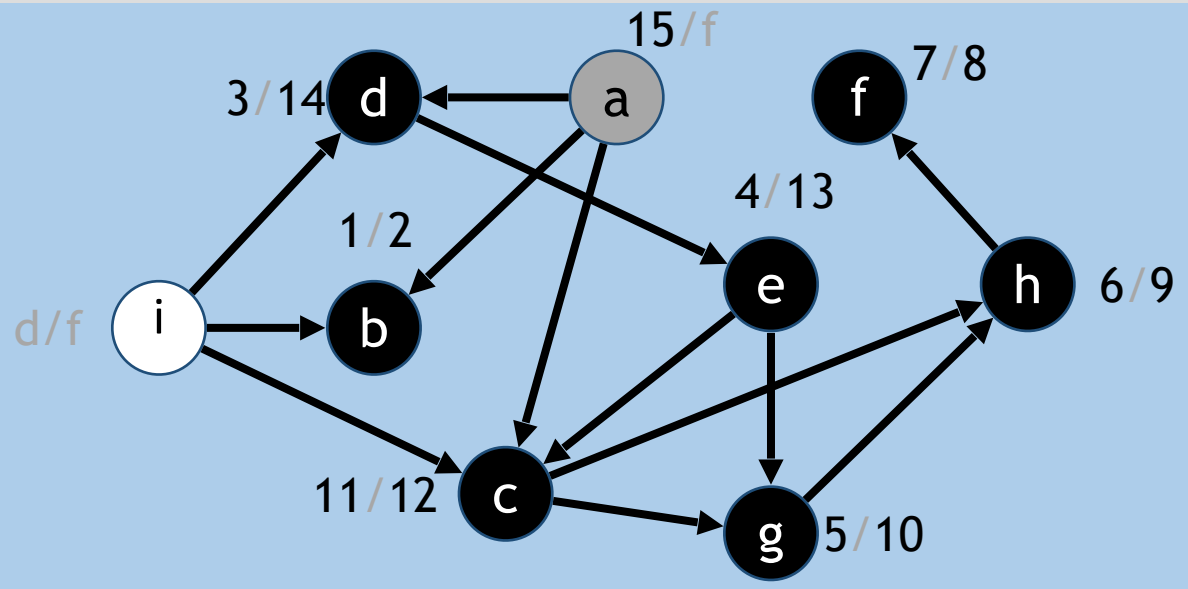
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



L_{sort}

Topological Sort

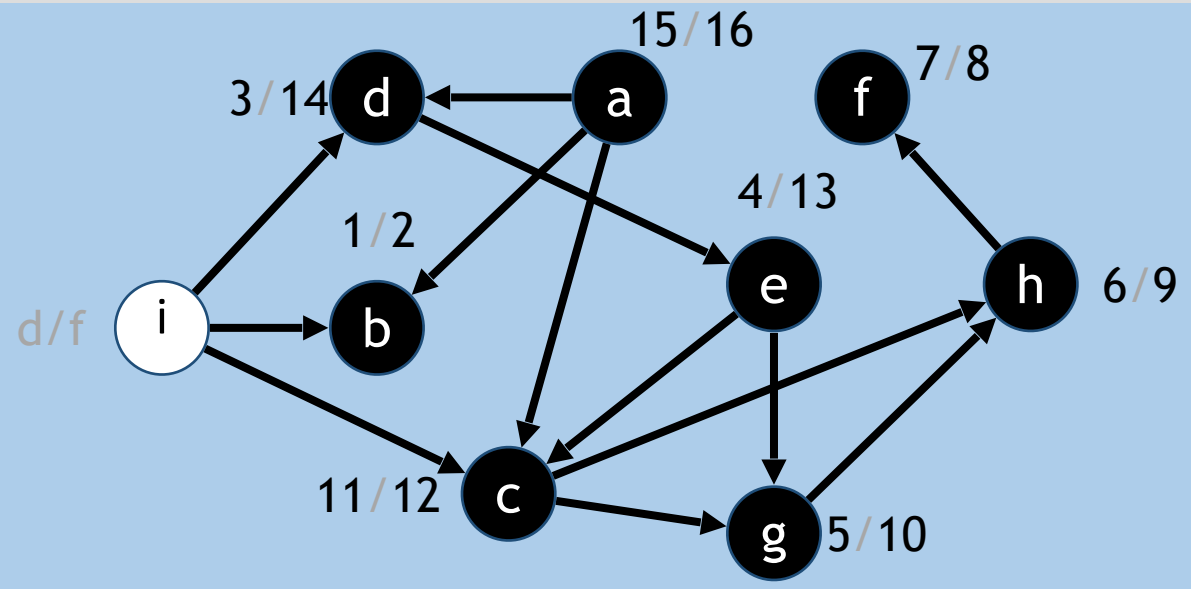
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



L_{sort}

Topological Sort

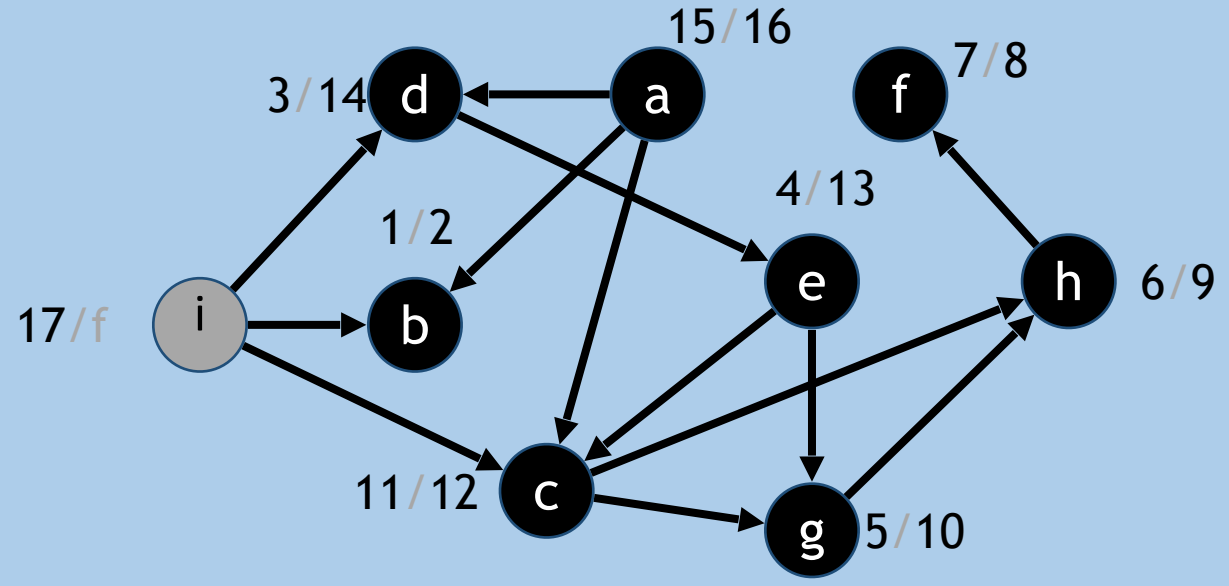
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



L_{sort}

Topological Sort

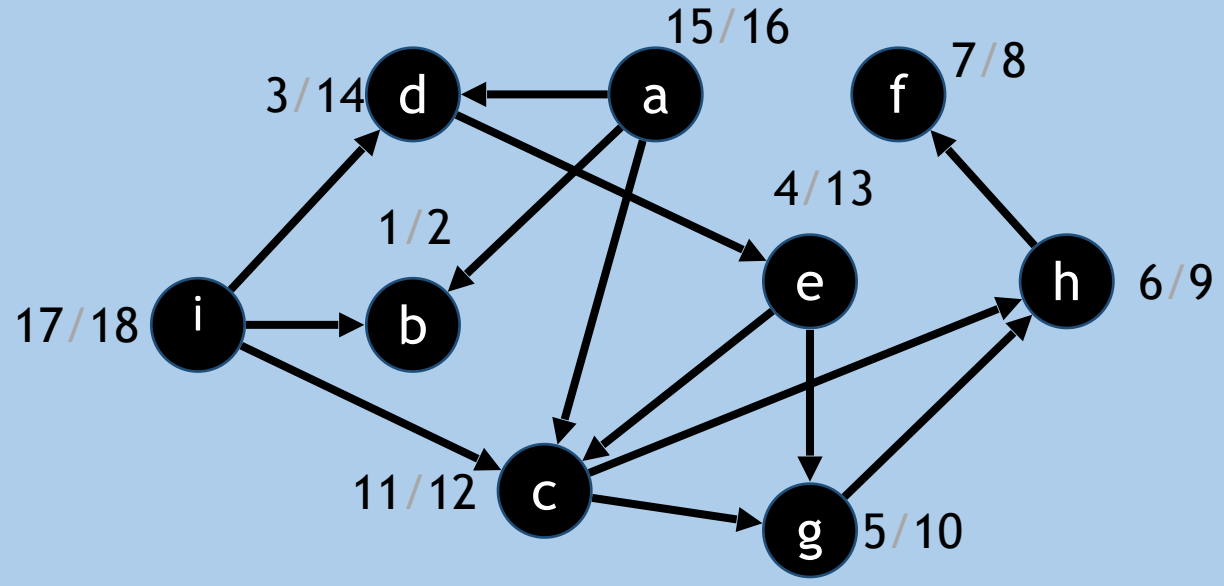
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



L_{sort}

Topological Sort

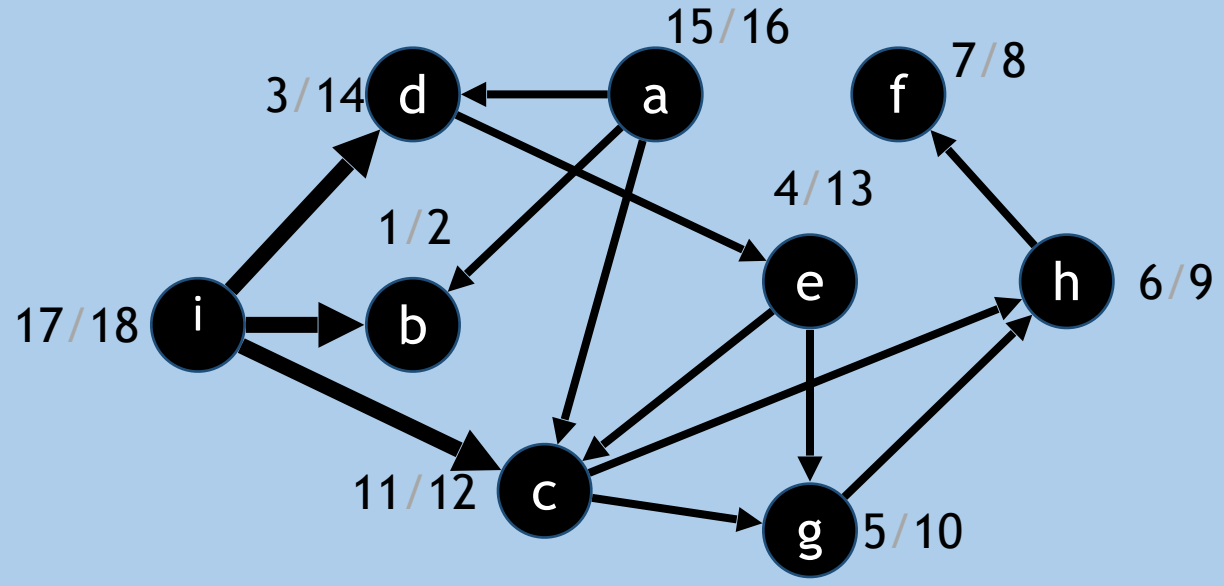
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

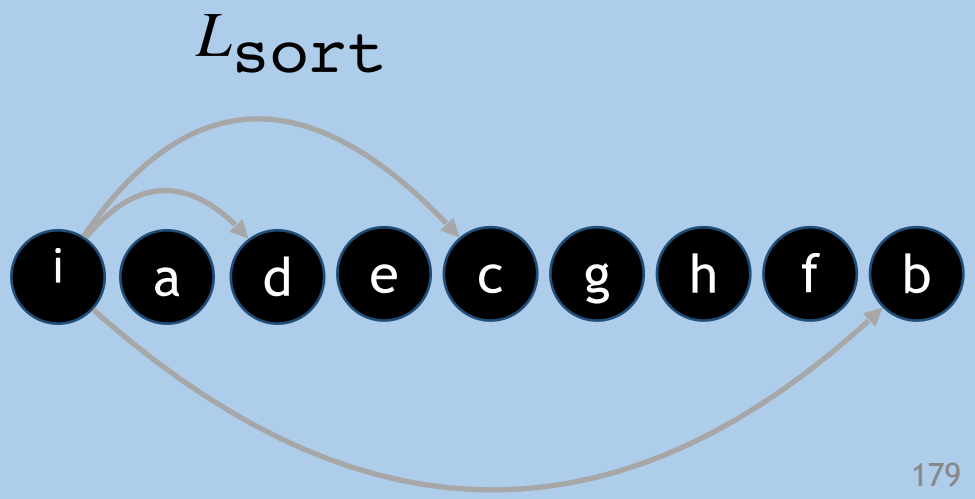


L_{sort}

Topological Sort

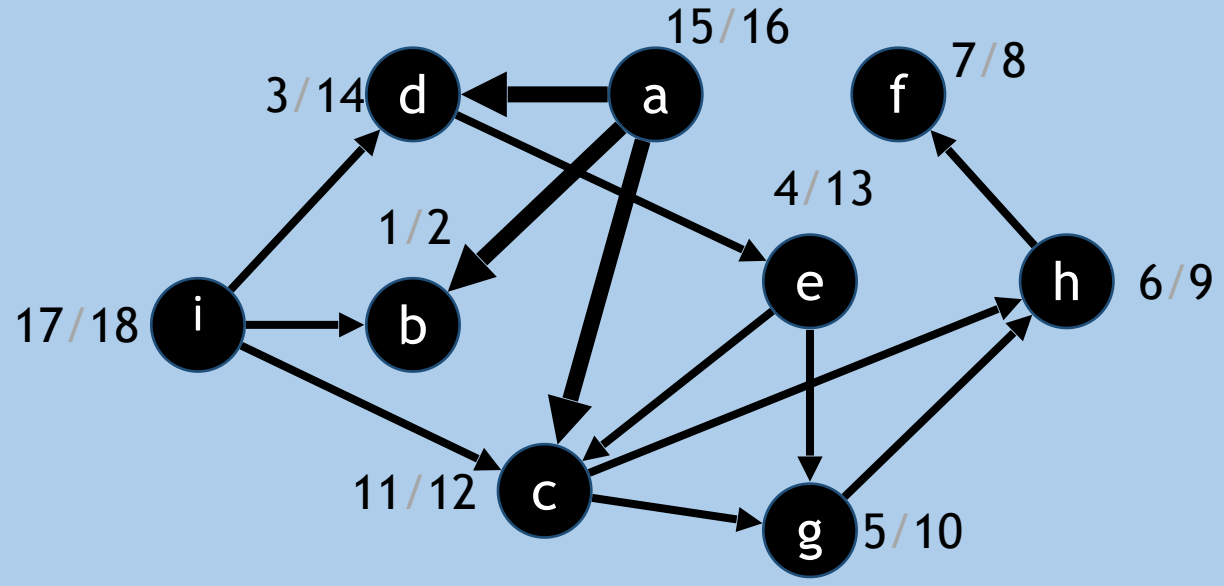
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

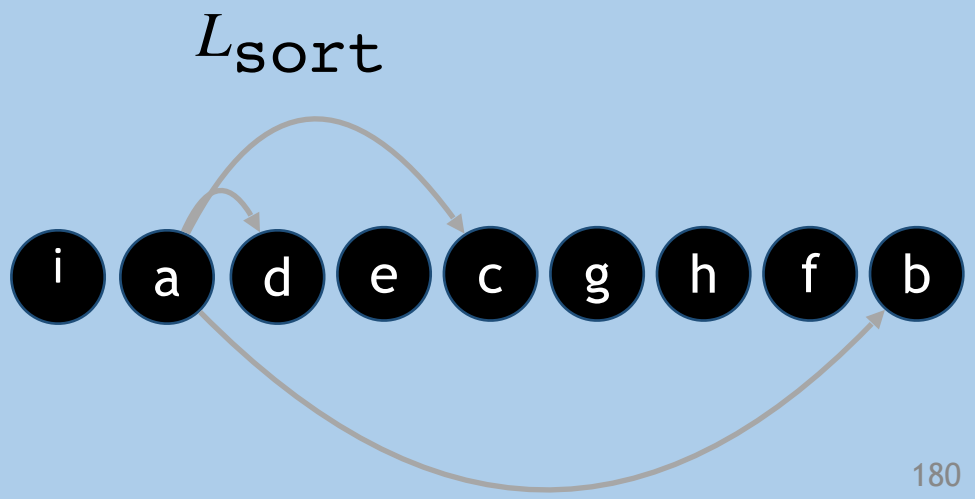




Topological Sort

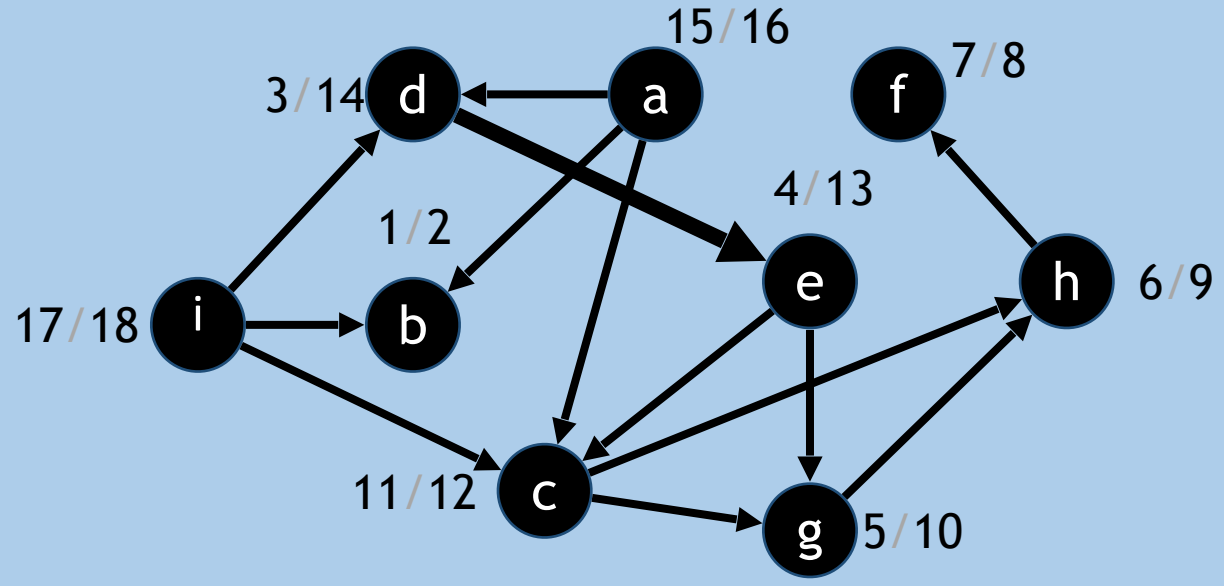
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



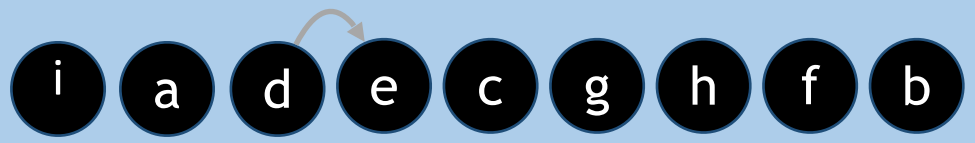


Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

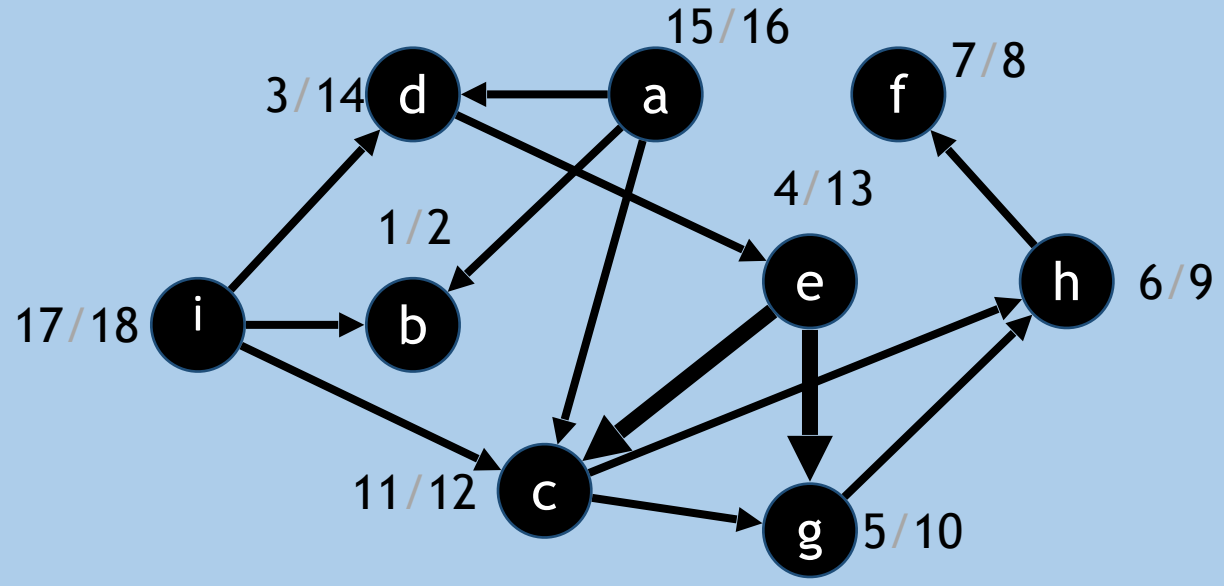


L_{sort}

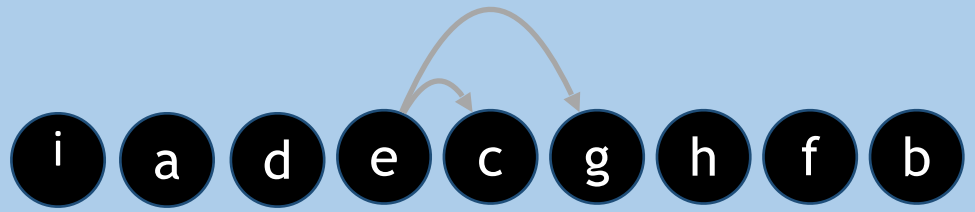


Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

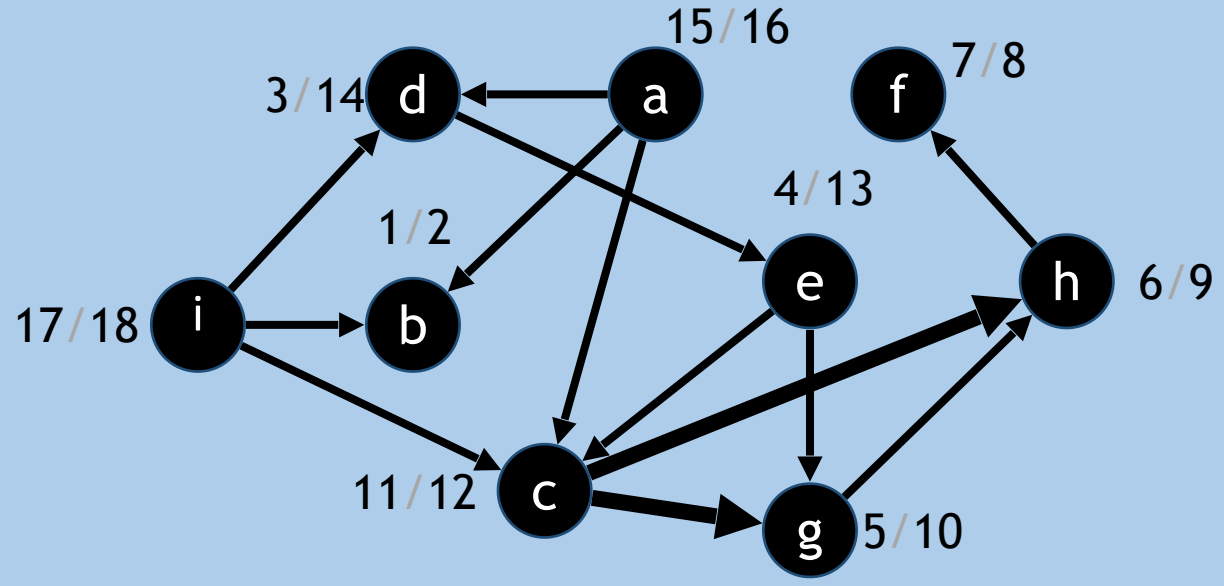


L_{sort}

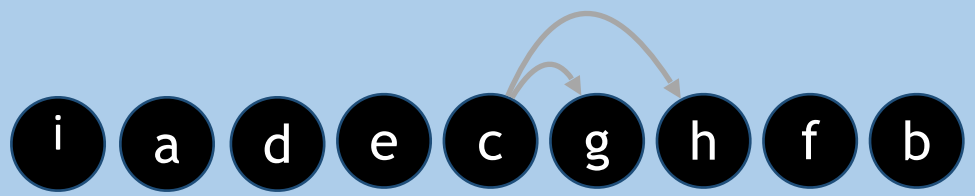


Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

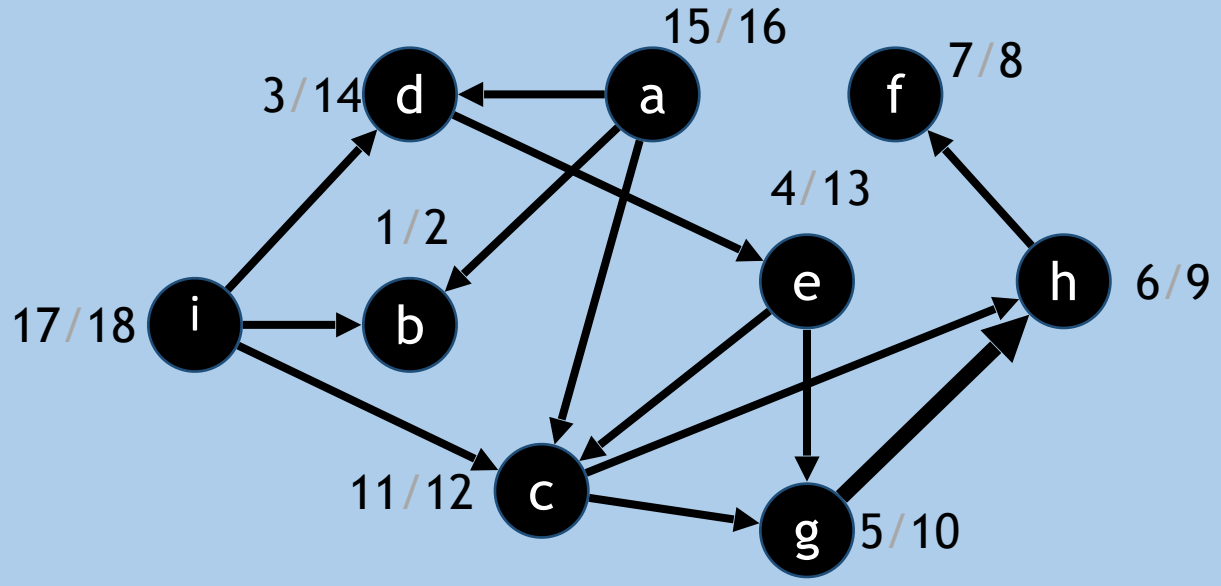


L_{sort}

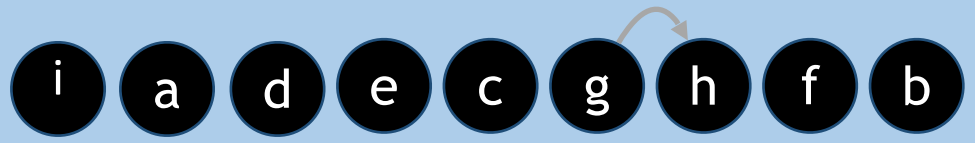


Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

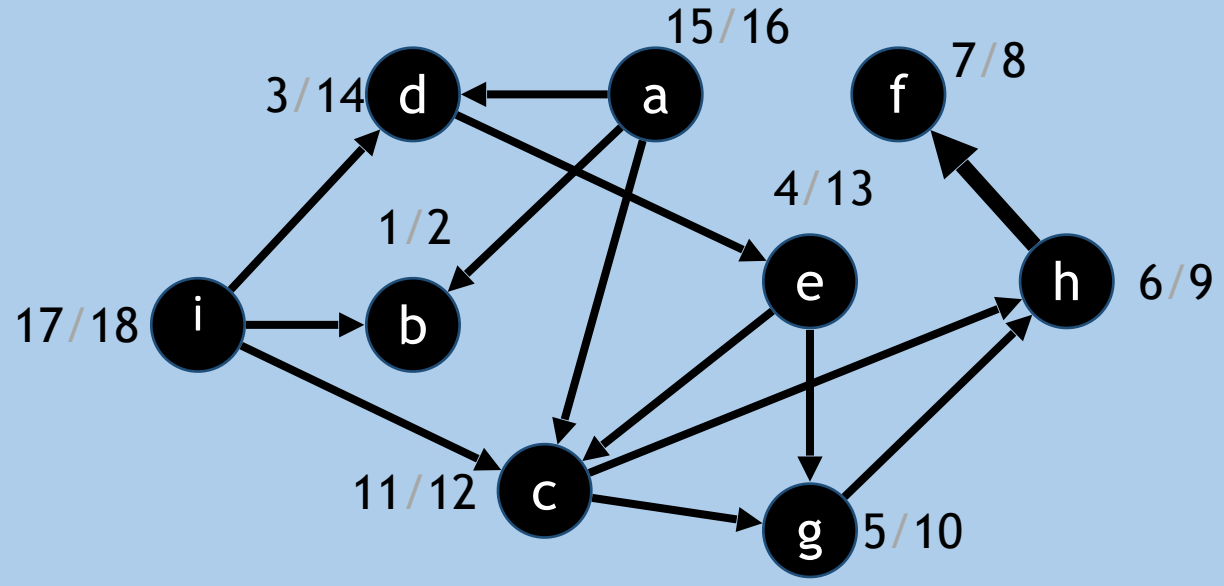


L_{sort}

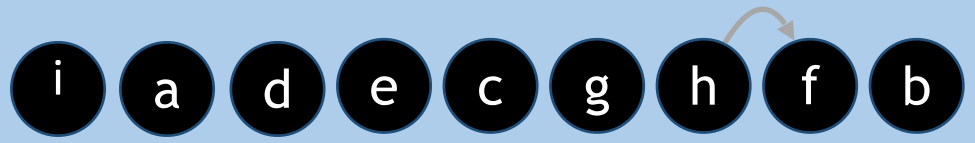


Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

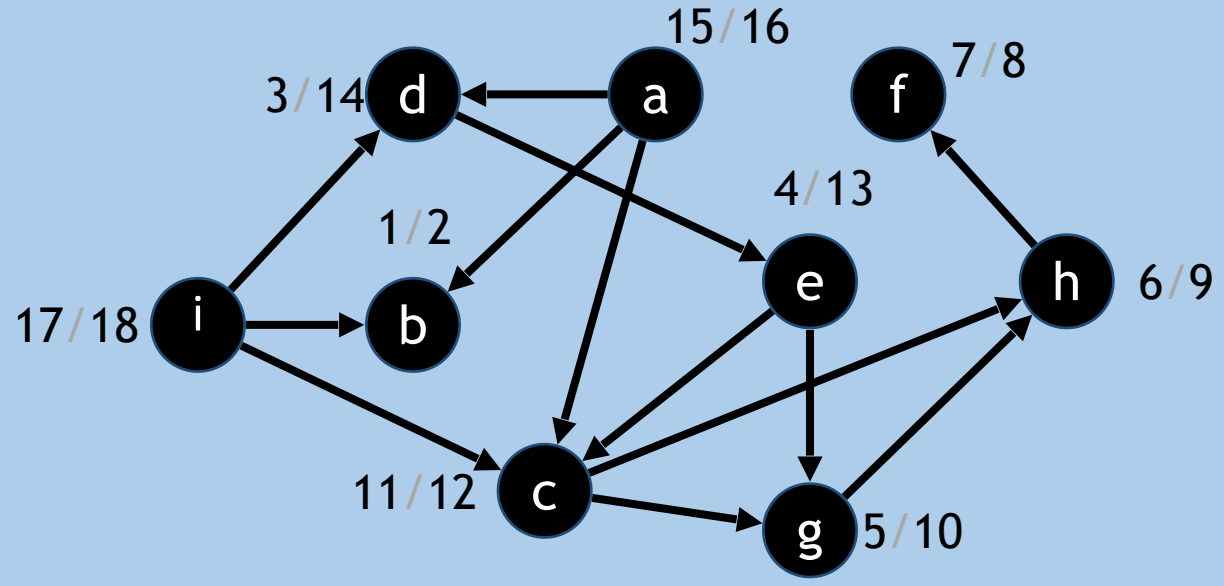


L_{sort}



Topological Sort

```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```



L_{sort}

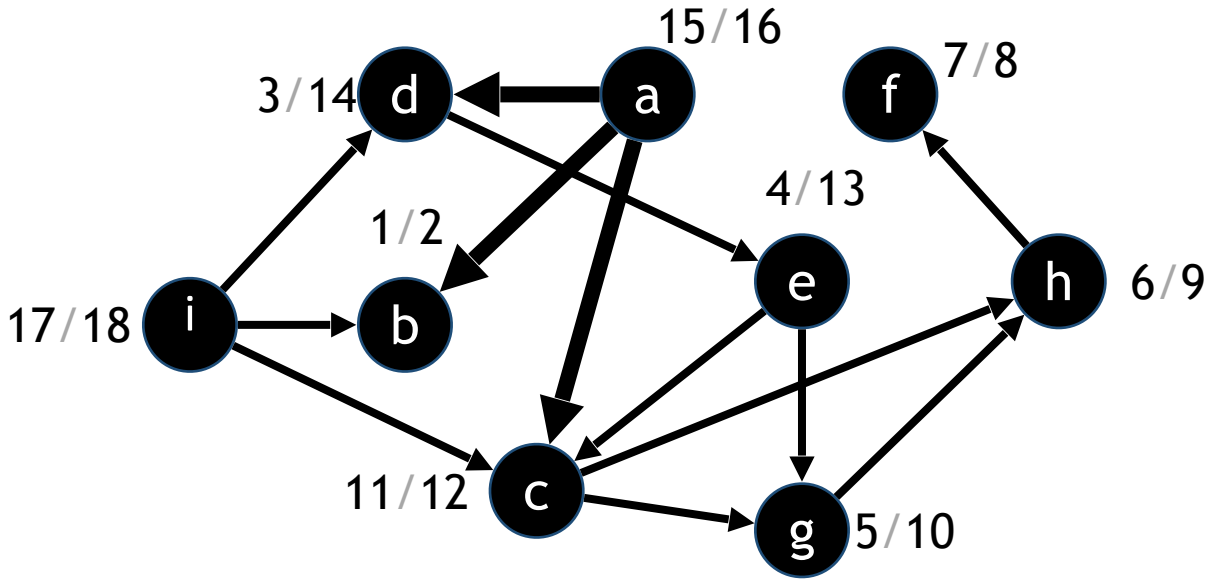
Topological Sort Performance

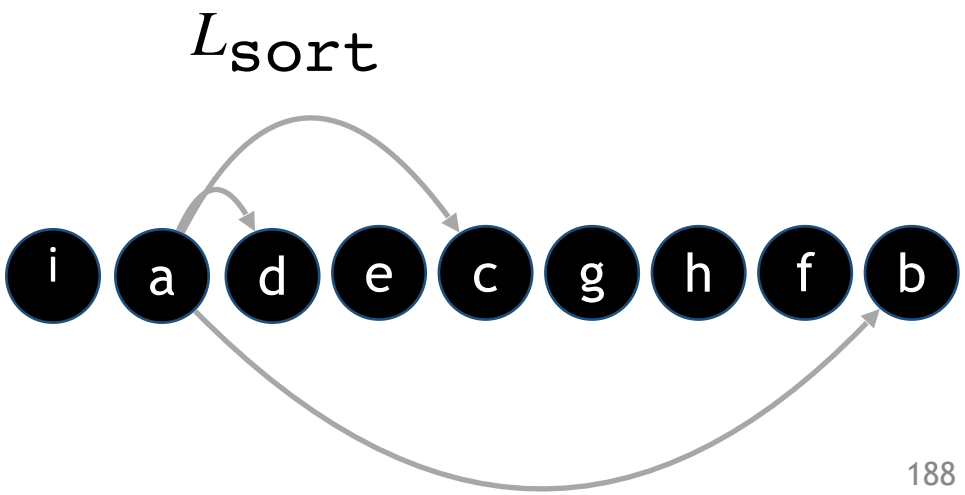
```
Procedure Topological-Sort(G)
   Linked list L_sort ← NIL
   Call DFS(G)
   As each vertex v is finished
      Put v to the front of L_sort
   Return L_sort
```

- DFS: $O(|V| + |E|)$ time
  - We put the v to L_{sort} once it is marked as finished (no need to sort the vertices by their finish time): total $O(|V|)$ time

Topological Sort Correctness

Theorem: After the procedure **Topological-Sort**, for any two distinct vertices $u, v \in V$, if there is an edge $(u, v) \in E$, in the list L_{sort}, u appears before v.

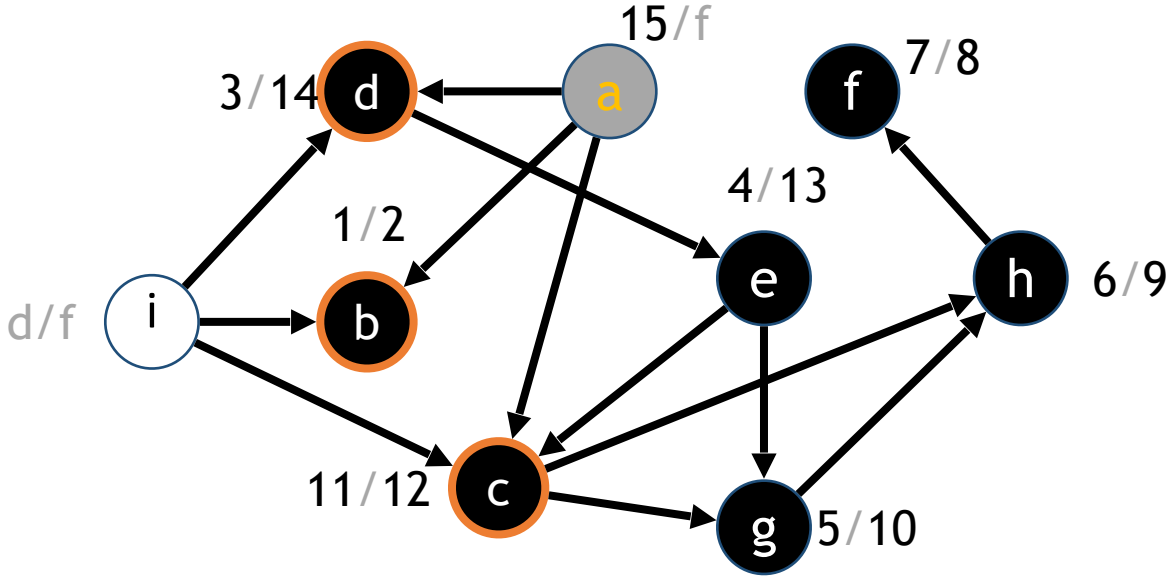




Topological Sort Correctness

Theorem: After the procedure **Topological-Sort**, for any two distinct vertices $u, v \in V$, if there is an edge $(u, v) \in E$, in the list L_{sort}, u appears before v.

<Proof Idea>: when a vertex is finished, all its children have been marked as finished $\Rightarrow$ all the children appears after it in L_{sort}



L_{sort}

Proof: exercise

What's Happening

- Given a directed acyclic graph, we can find a linear ordering of the vertices by performing a DFS and sort the vertices backwards according to their finish time.

- You don't really need a sorting. The time complexity is $O(|V| + |E|)$

Outline

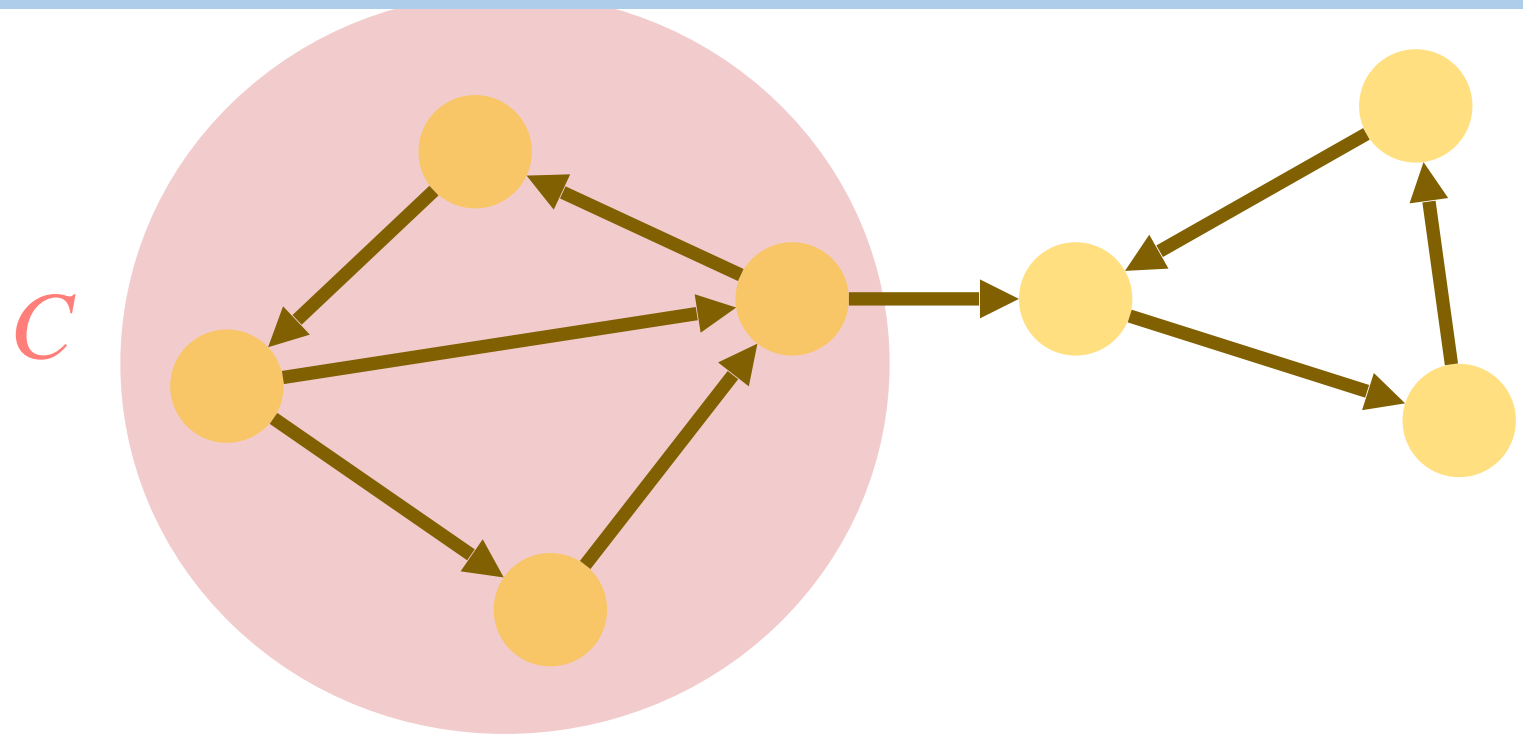
- Graph representations
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
- Two applications on DFS:
  - Topological sort
  - Strongly connected components

Strongly Connected Component

Definition: A **strongly connected component** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, there is a path from u to v and a path from v to u. That is, u and v are reachable from each other.

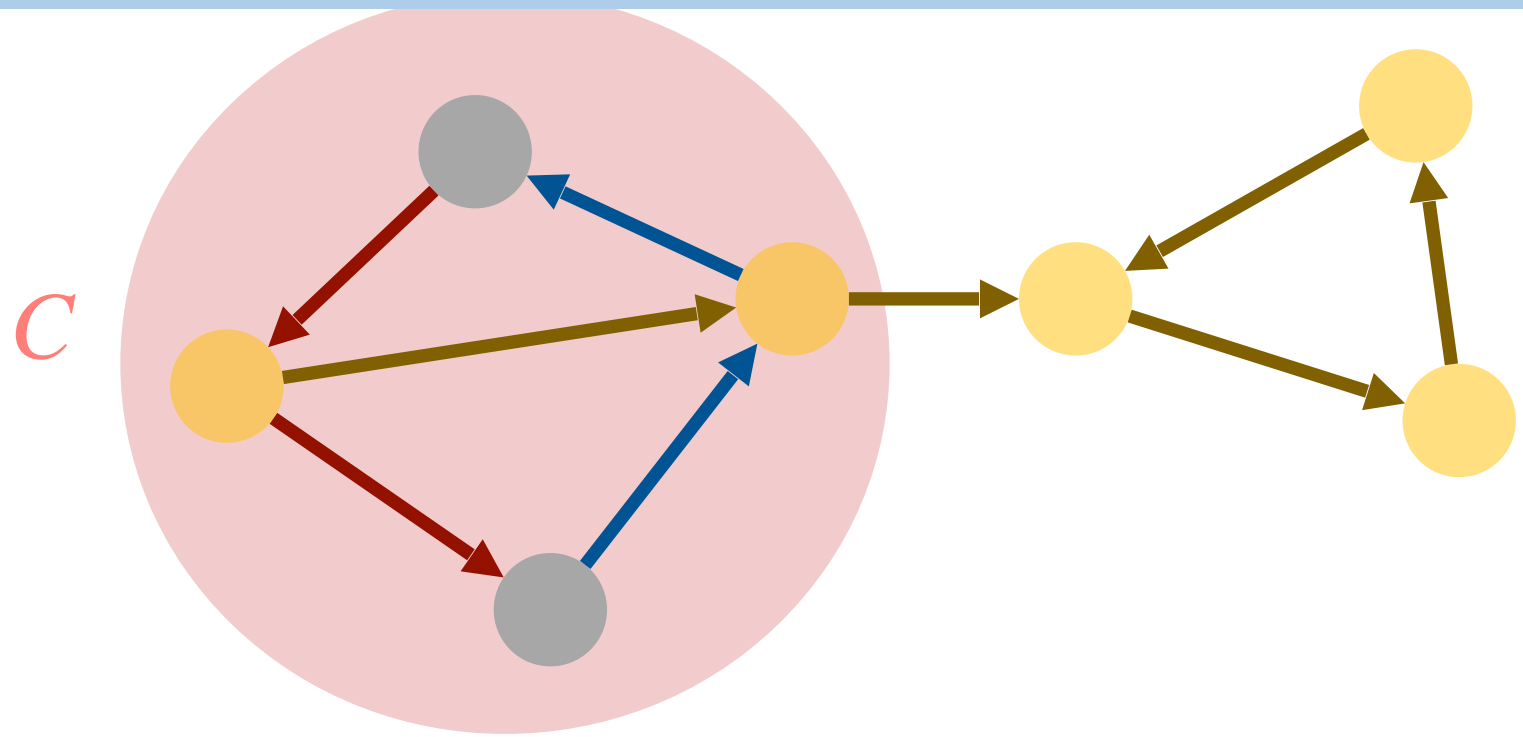
Strongly Connected Component

Definition: A **strongly connected component** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, there is a path from u to v and a path from v to u. That is, u and v are reachable from each other.



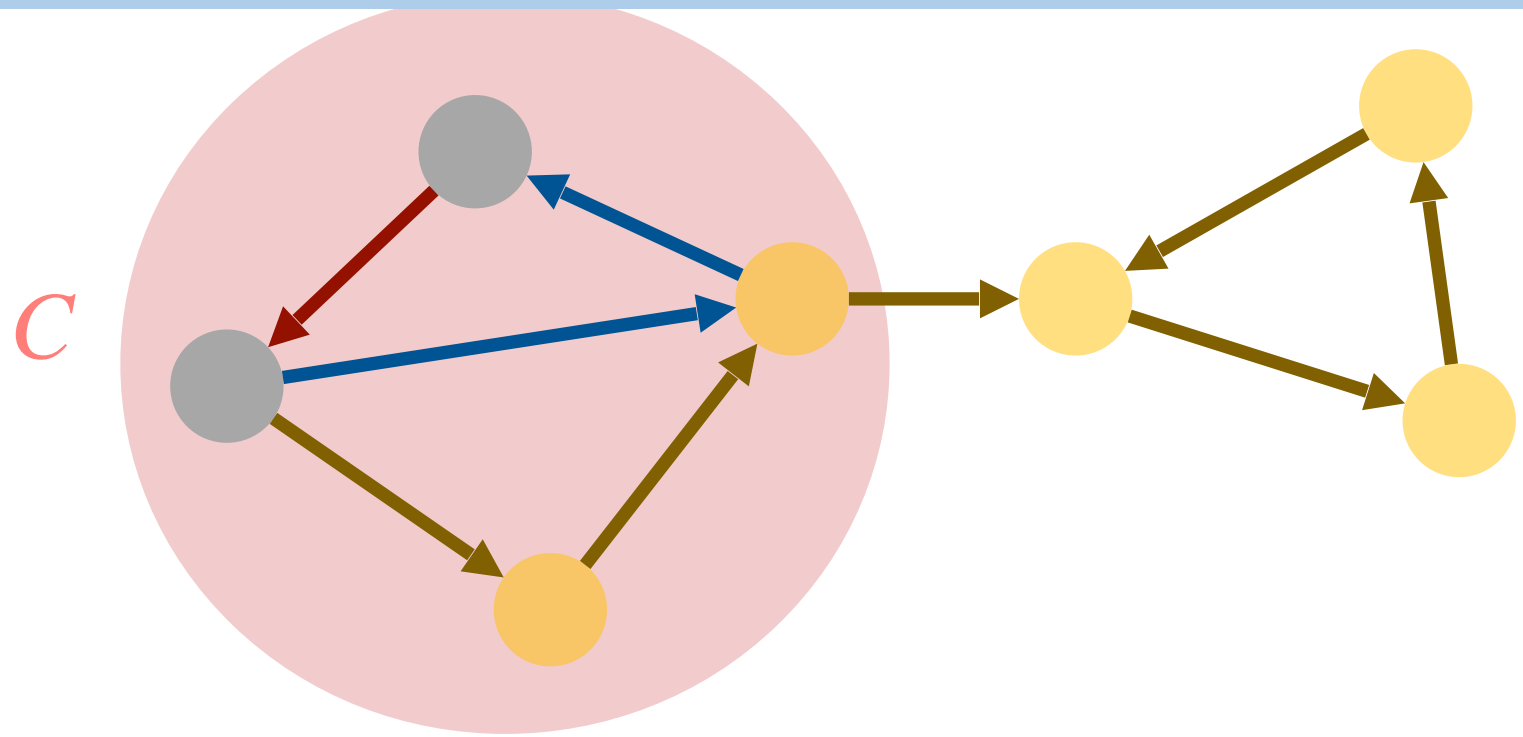
Strongly Connected Component

Definition: A **strongly connected component** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, there is a path from u to v and a path from v to u. That is, u and v are reachable from each other.



Strongly Connected Component

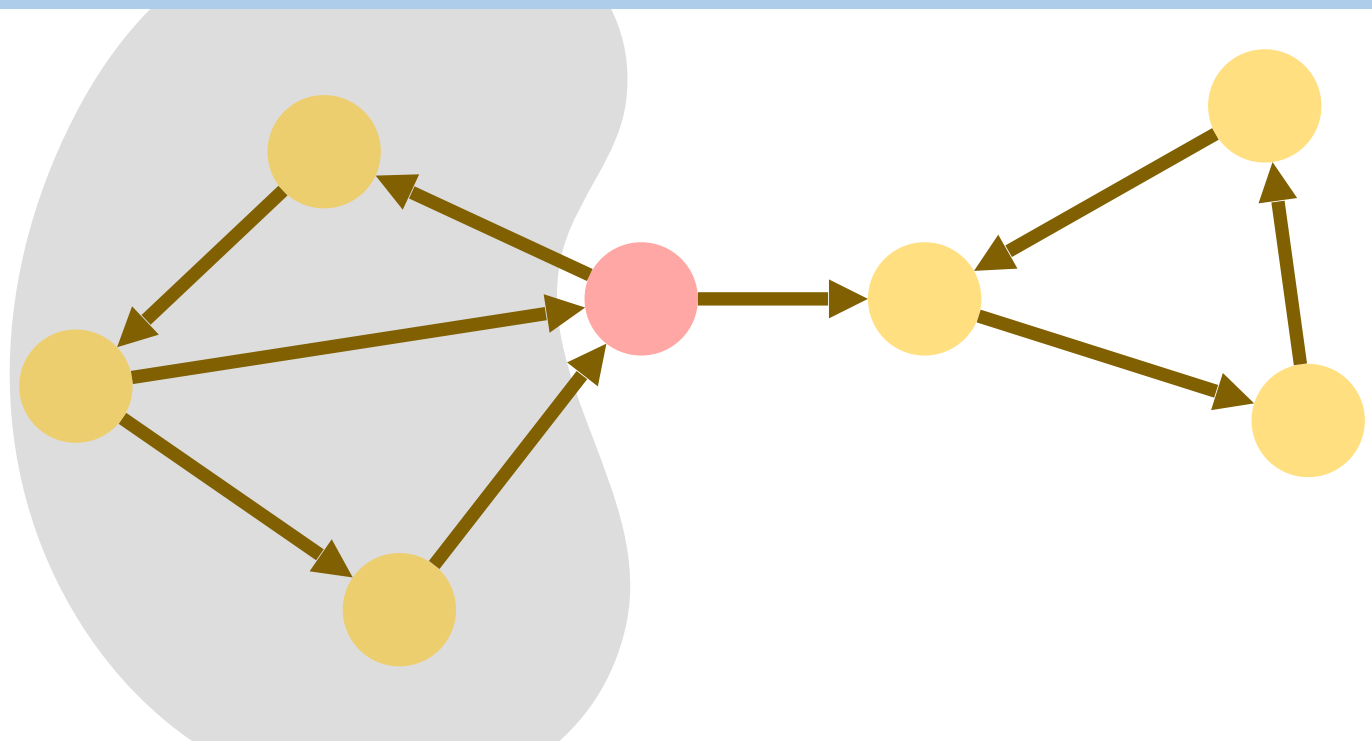
Definition: A **strongly connected component** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, there is a path from u to v and a path from v to u. That is, u and v are reachable from each other.



Strongly Connected Component

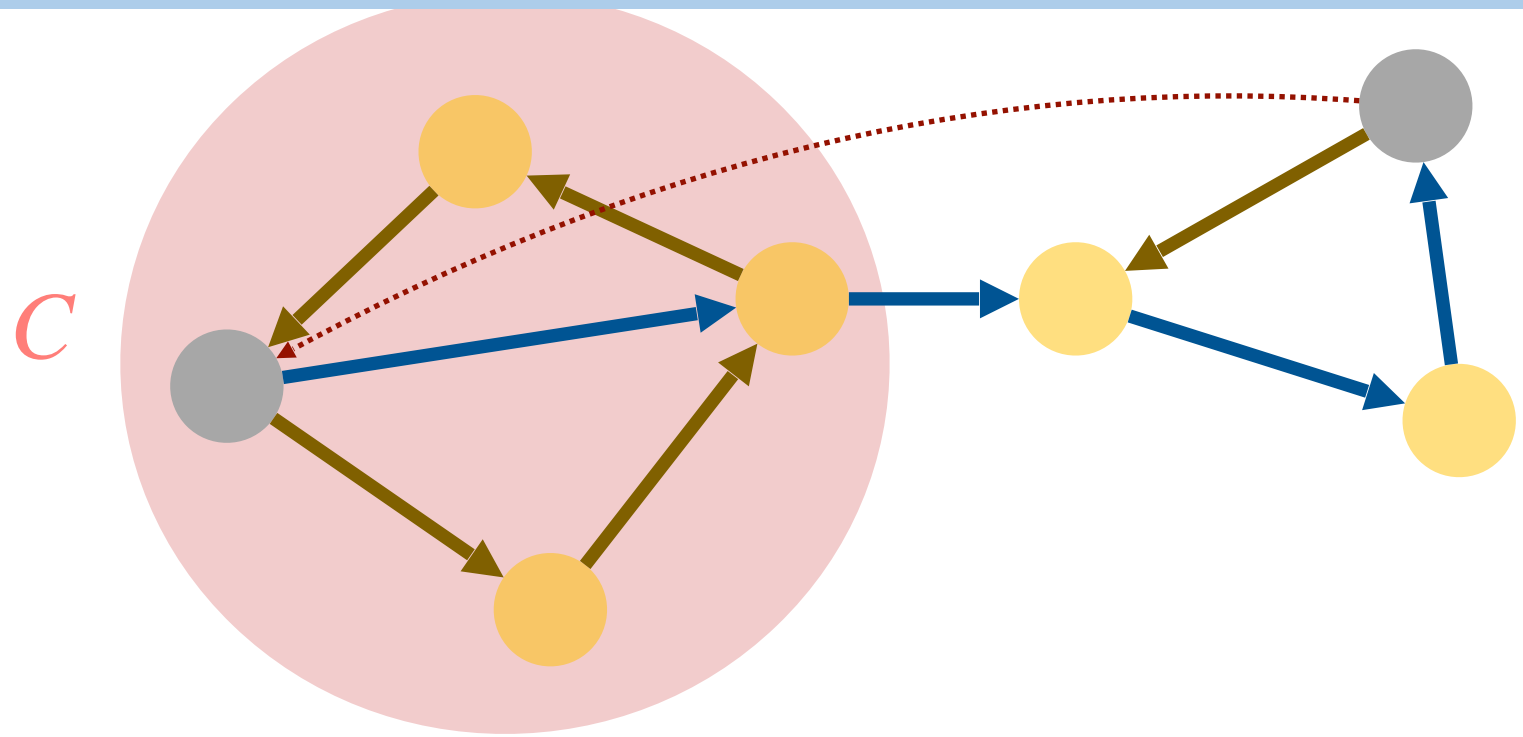
Definition: A **strongly connected component** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, there is a path from u to v and a path from v to u. That is, u and v are reachable from each other.

Not maximal



Strongly Connected Component

Definition: A **strongly connected component** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, there is a path from u to v and a path from v to u. That is, u and v are reachable from each other.



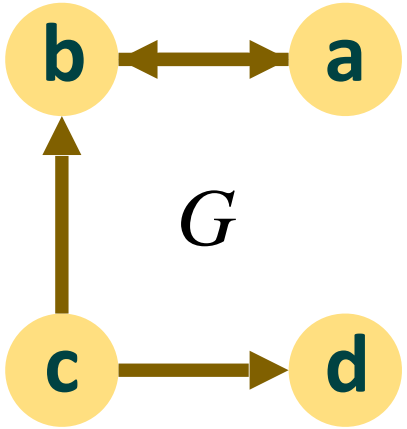
Find Strongly Connected Component

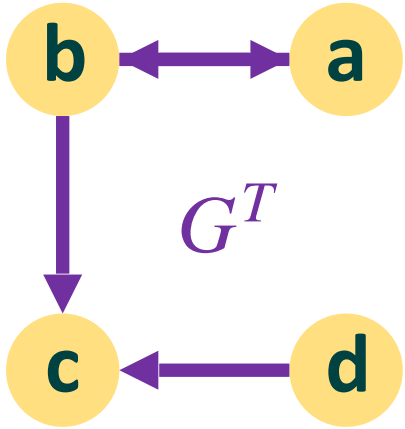
Find Strongly Connected Component

A **transpose** of a directed graph $G = (V, E)$ is a directed graph $G^T = (V, E^T)$, where $E^T = \{(u, v) \mid (v, u) \in E\}$

Find Strongly Connected Component

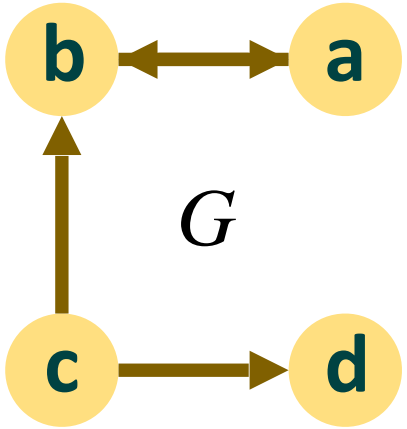
A **transpose** of a directed graph $G = (V, E)$ is a directed graph $G^T = (V, E^T)$, where $E^T = \{(u, v) \mid (v, u) \in E\}$

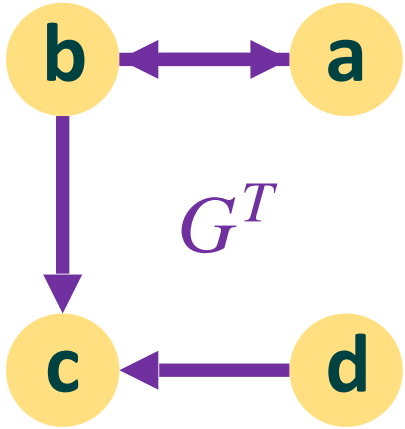




Find Strongly Connected Component

A **transpose** of a directed graph $G = (V, E)$ is a directed graph $G^T = (V, E^T)$, where $E^T = \{(u, v) \mid (v, u) \in E\}$

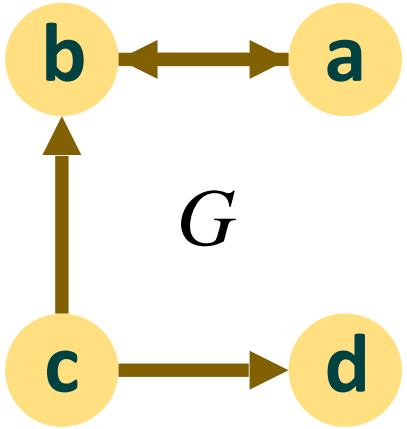


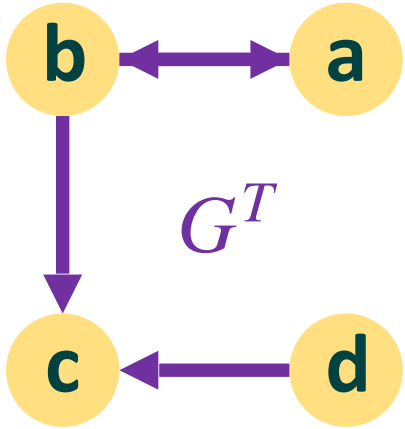


- G^T can be constructed in $O(|V| + |E|)$ time

Find Strongly Connected Component

A **transpose** of a directed graph $G = (V, E)$ is a directed graph $G^T = (V, E^T)$, where $E^T = \{(u, v) \mid (v, u) \in E\}$

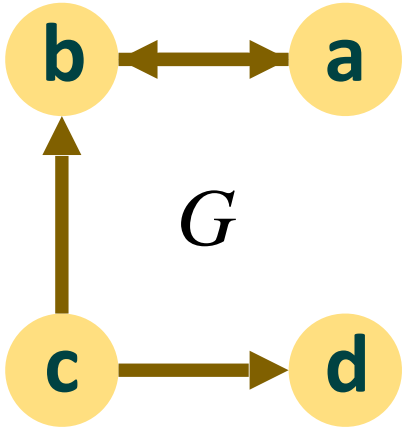


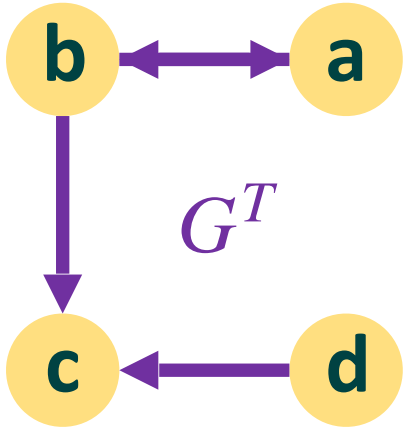


- If there is a path from u to v in G, there is a path from v to u in G^T
- For two vertices u and v in a strongly connected component, there is a path from u to v, and a path from v to u

Find Strongly Connected Component

A **transpose** of a directed graph $G = (V, E)$ is a directed graph $G^T = (V, E^T)$, where $E^T = \{(u, v) \mid (v, u) \in E\}$

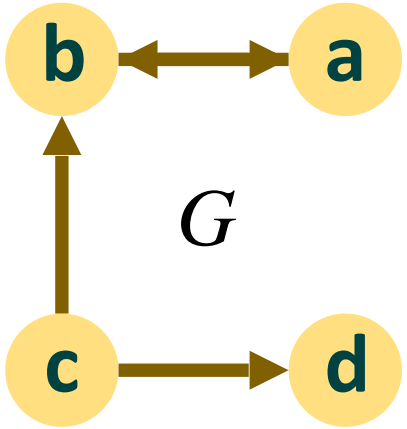


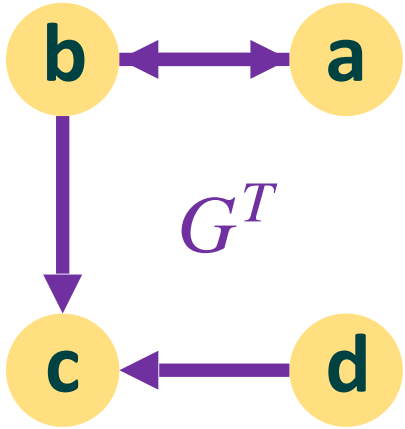


- If there is a path from u to v in G, there is a path from v to u in G^T
- For two vertices u and v in a strongly connected component, there is a path from u to v in G, and a path from u to v in G^T

Find Strongly Connected Component

A **transpose** of a directed graph $G = (V, E)$ is a directed graph $G^T = (V, E^T)$, where $E^T = \{(u, v) \mid (v, u) \in E\}$

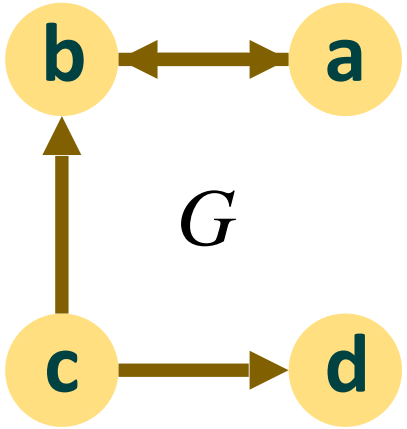


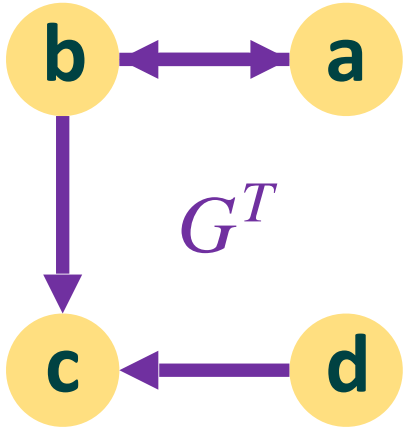


- Maybe we can run a DFS on G, and another DFS on G^T to get the reachable sets of vertices in both graphs and find their intersection?

Find Strongly Connected Component

A **transpose** of a directed graph $G = (V, E)$ is a directed graph $G^T = (V, E^T)$, where $E^T = \{(u, v) \mid (v, u) \in E\}$





- Maybe we can run a DFS on G, and another DFS on G^T to get the reachable sets of vertices in both graphs and find their intersection?
  - Finding the intersections can be tricky…

Strongly Connected Component

```
Procedure STRONGLY-CONNECTED-COMPONENTS(G)
   Call DFS(G) and compute finishing times f[u] for each u
   Compute G^T
   While (some node in G^T is not-discovered){
      u ← not-discovered node with latest f[u]
      Call DFS-VISIT(u) on G^T}
   The depth-first trees in DFS(G^T) are SCCs
```

Strongly Connected Component

```
Procedure STRONGLY-CONNECTED-COMPONENTS(G)
   Call DFS(G) and compute finishing times f[u] for each u
   Compute G^T
   While (some node in G^T is not-discovered){
      u ← not-discovered node with latest f[u]
      Call DFS-VISIT(u) on G^T}
   The depth-first trees in DFS(G^T) are SCCs
```

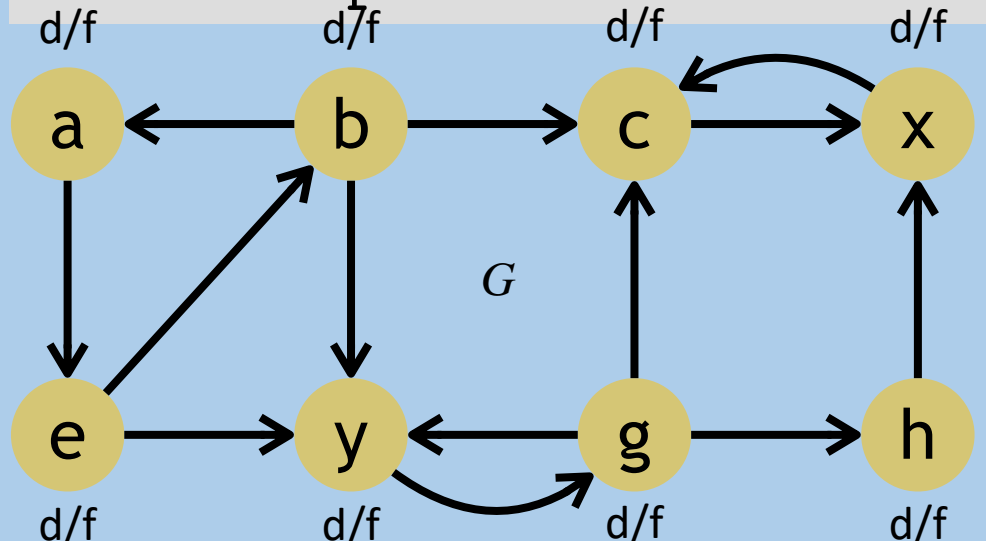
Strongly Connected Component

```
Procedure STRONGLY-CONNECTED-COMPONENTS(G)
   Call DFS(G) and compute finishing times f[u] for each u
   Compute G^T
   While (some node in G^T is not-discovered){
      u ← not-discovered node with latest f[u]
      Call DFS-VISIT(u) on G^T}
   The depth-first trees in DFS(G^T) are SCCs
```

```
Procedure DFS(G)
     For each u ∈ V[G]
          Mark u as not-discovered
          π[v] ← NIL
     time ← 0
     For each vertex u ∈ V[G]
          If u is not-discovered
               DFS-VISIT(u)
```

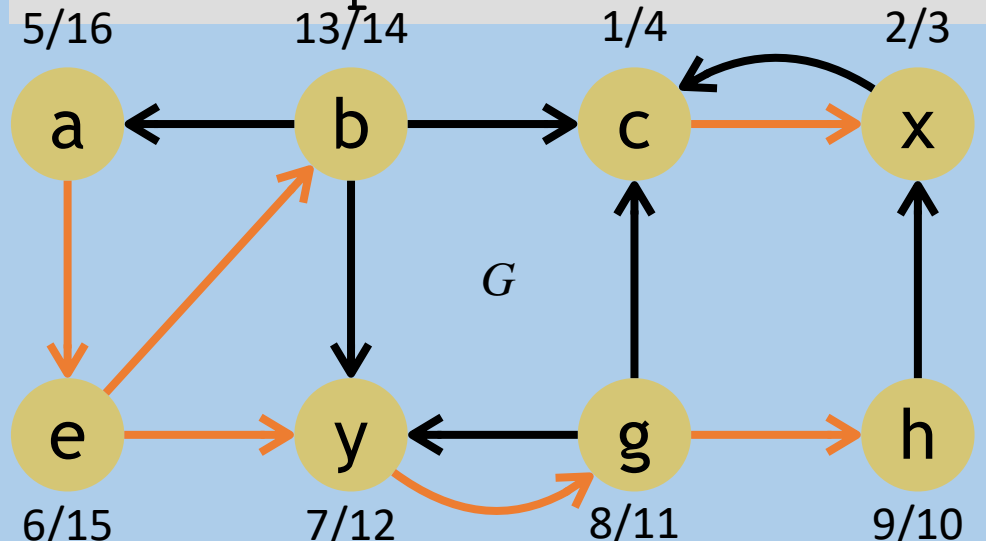
Strongly Connected Component

Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs



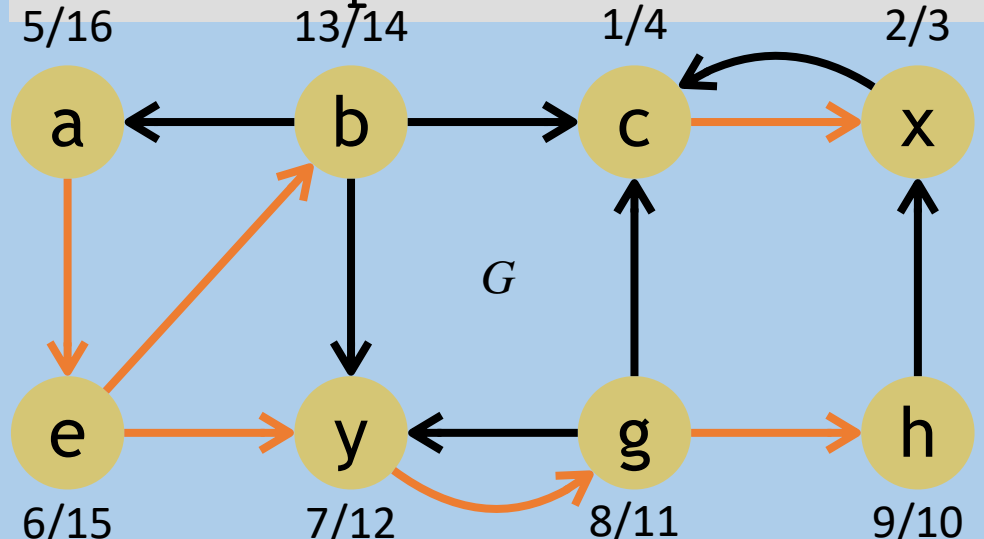
Strongly Connected Component

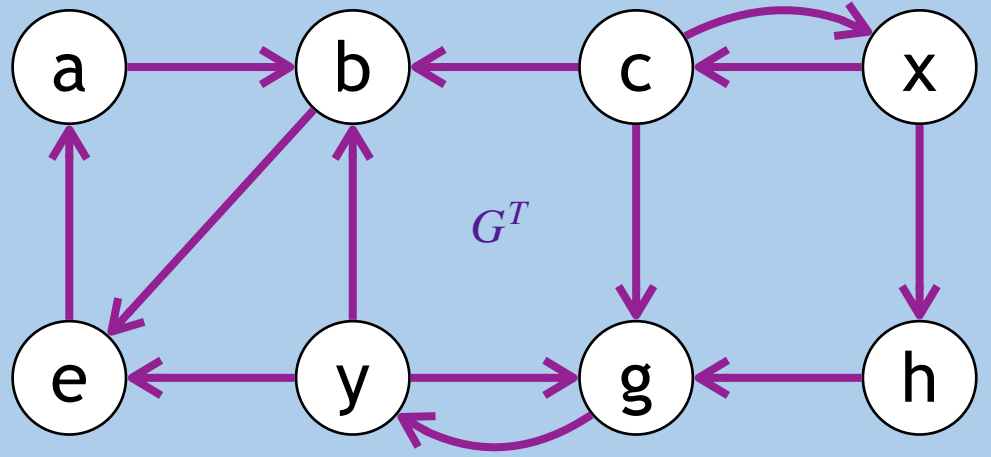
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs



Strongly Connected Component

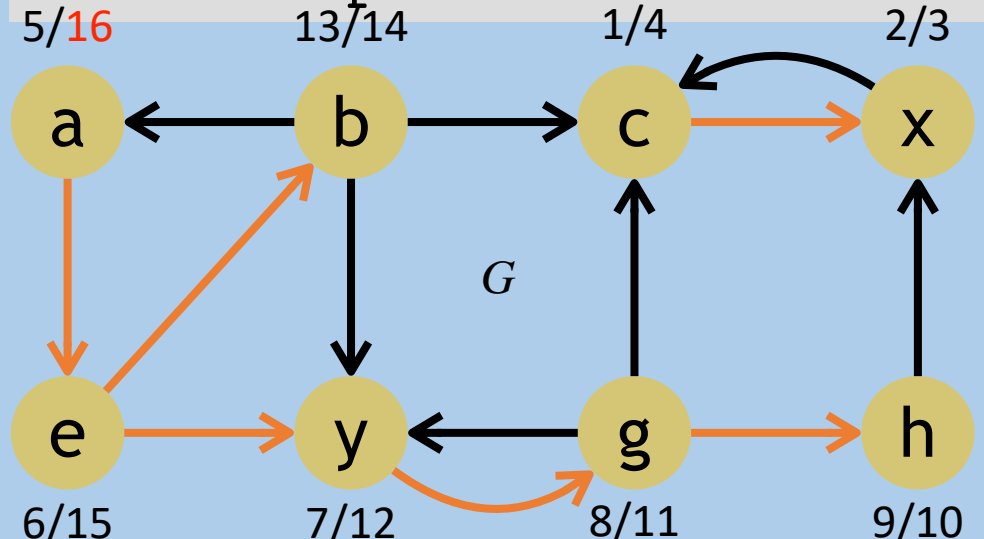
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

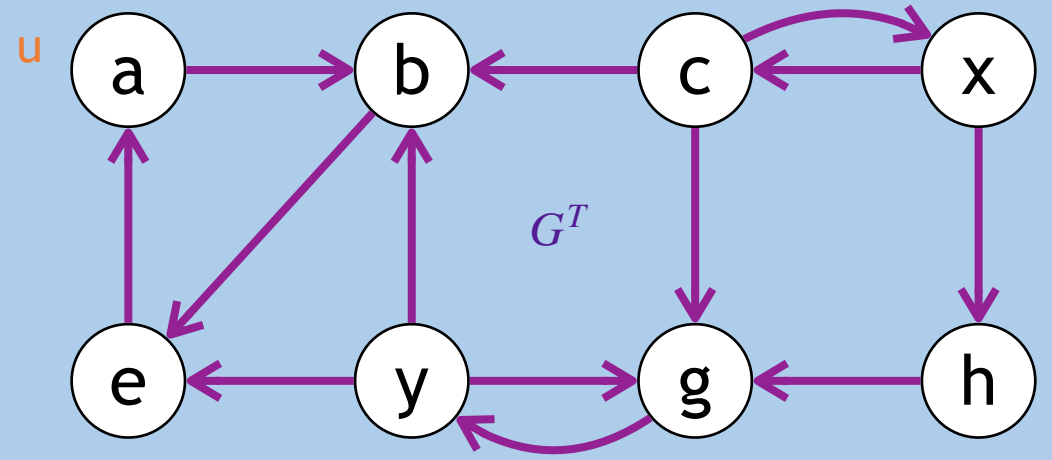




Strongly Connected Component

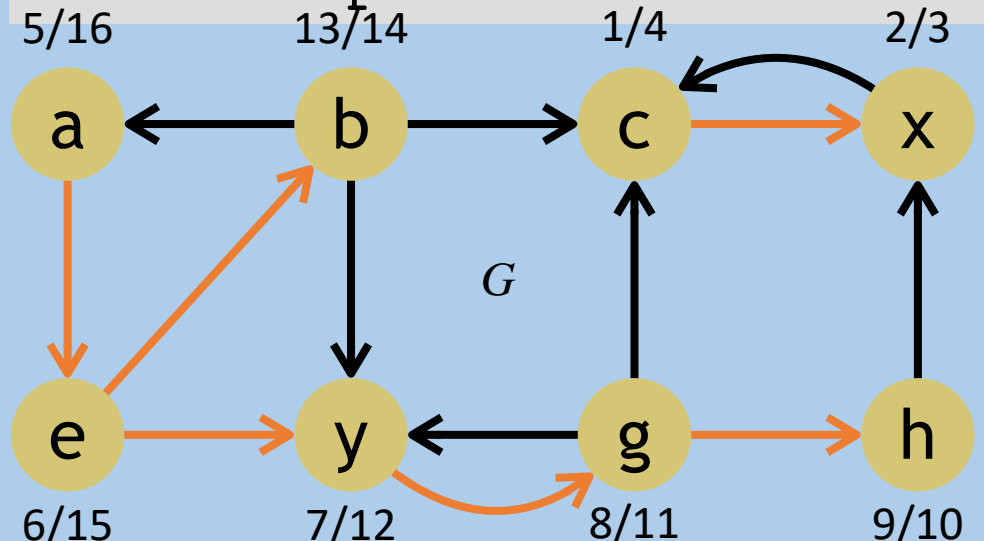
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

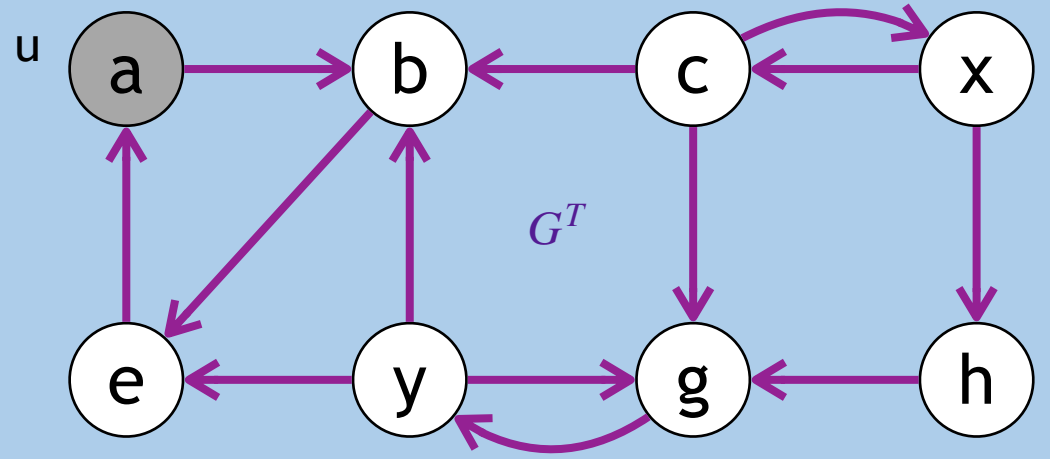




Strongly Connected Component

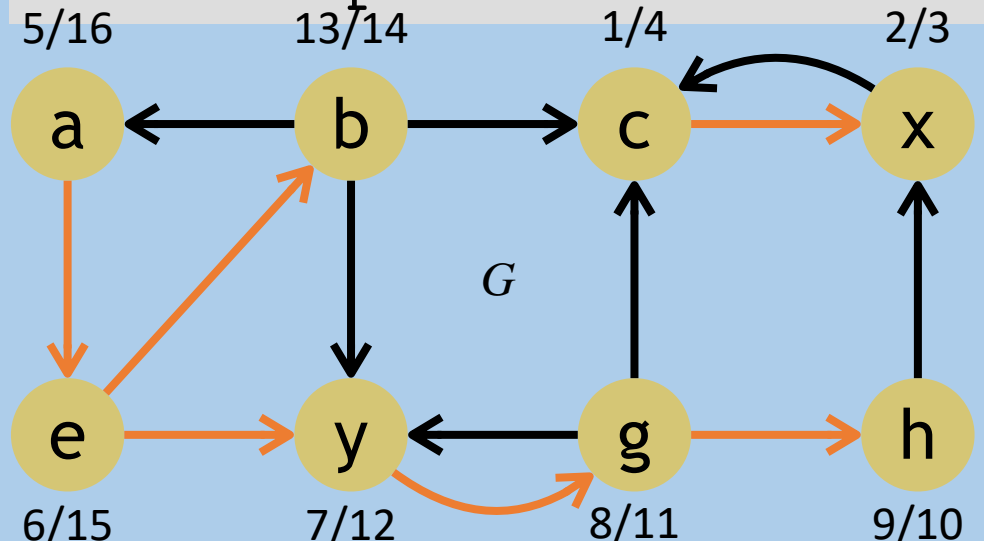
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

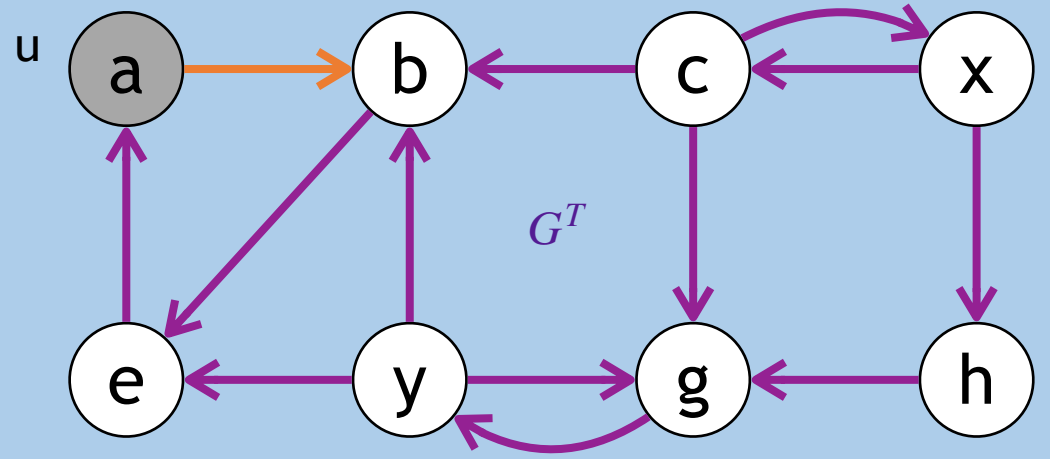




Strongly Connected Component

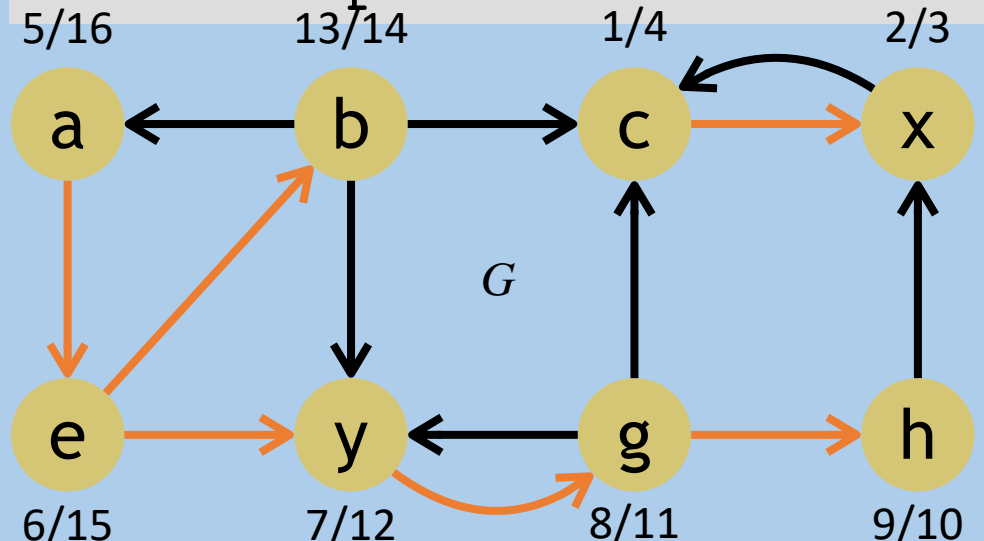
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

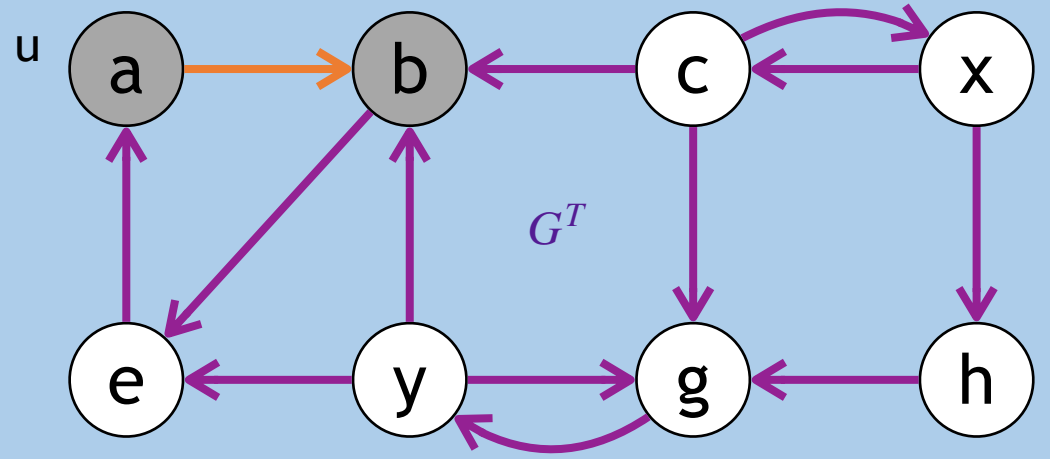




Strongly Connected Component

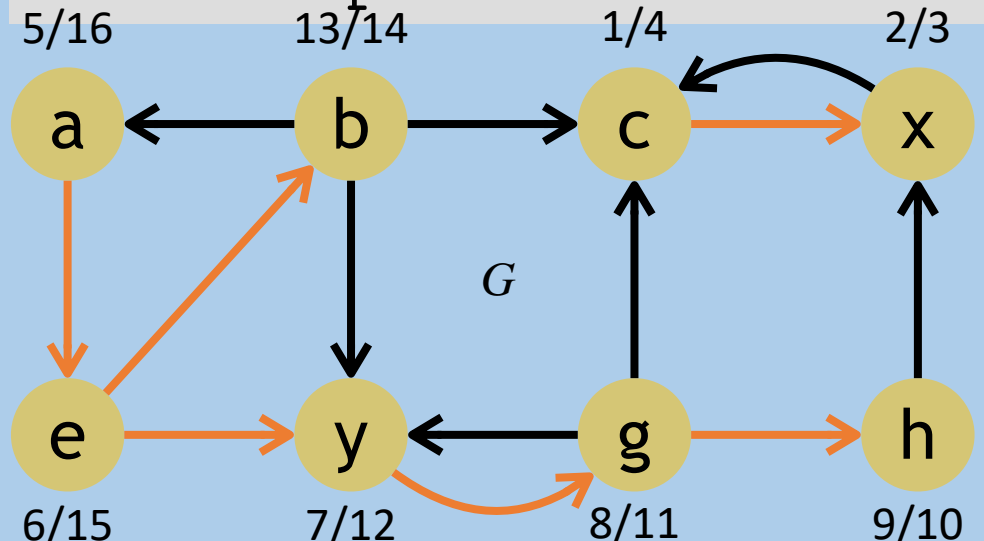
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

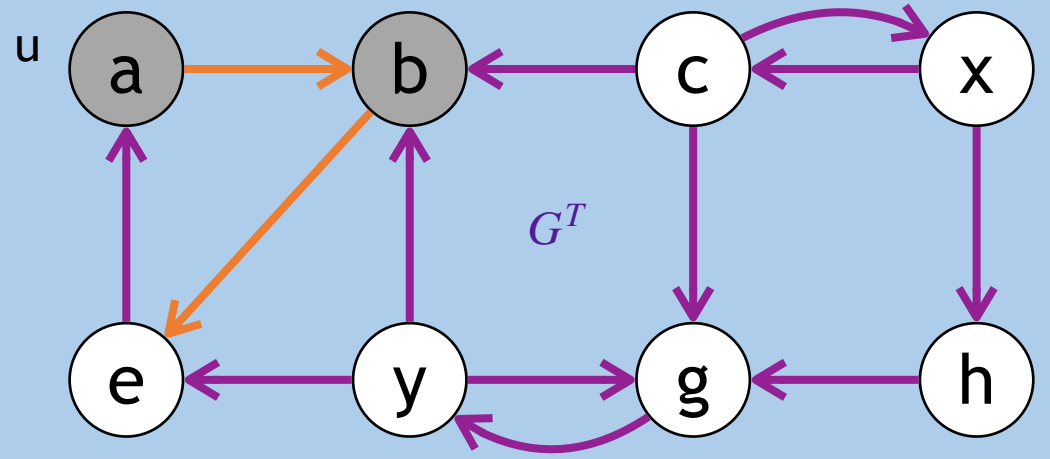




Strongly Connected Component

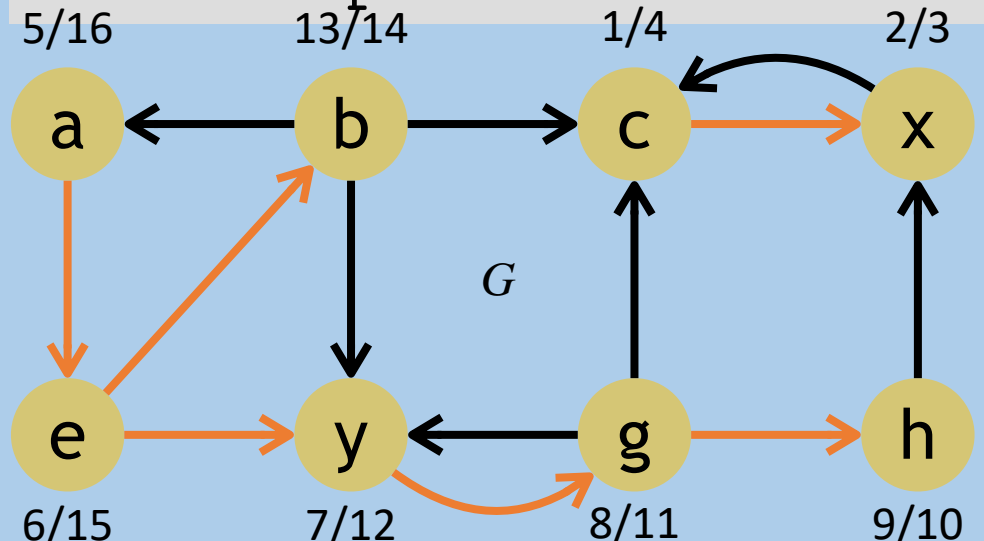
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

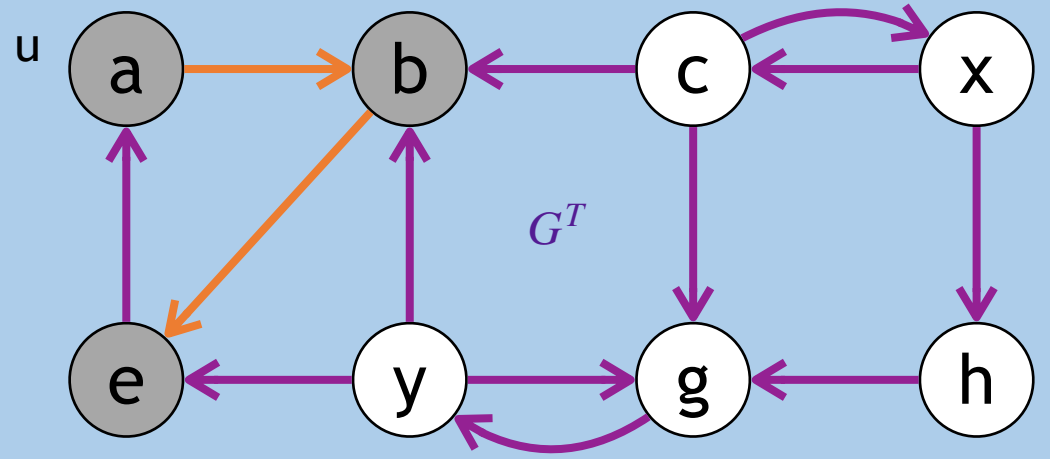




Strongly Connected Component

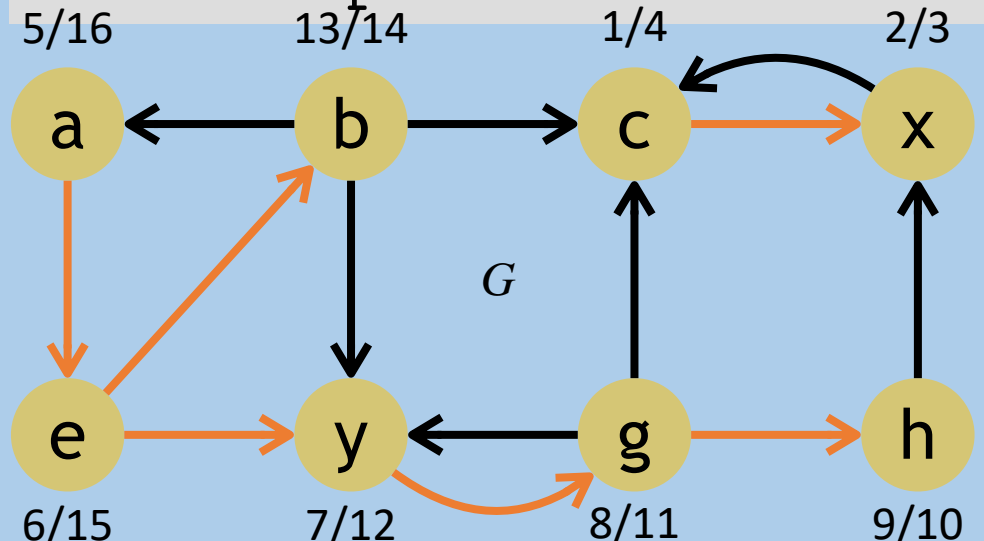
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

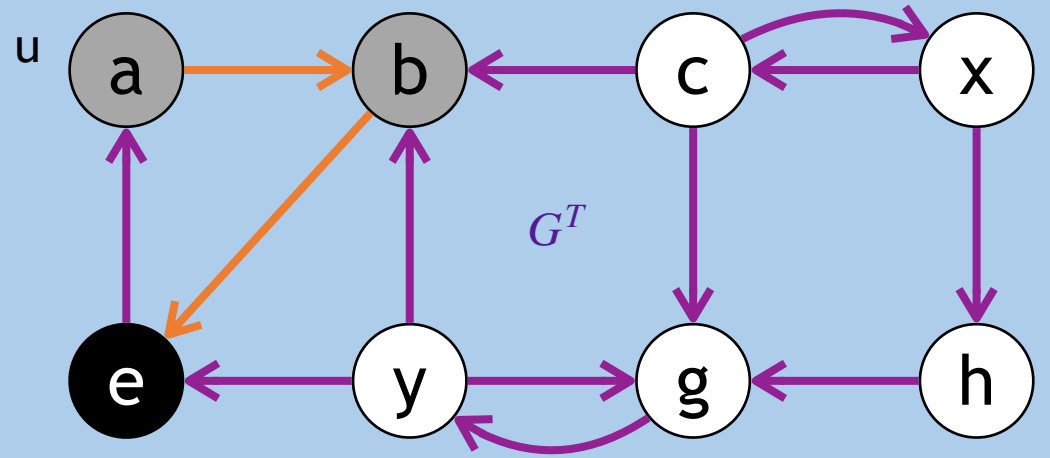




Strongly Connected Component

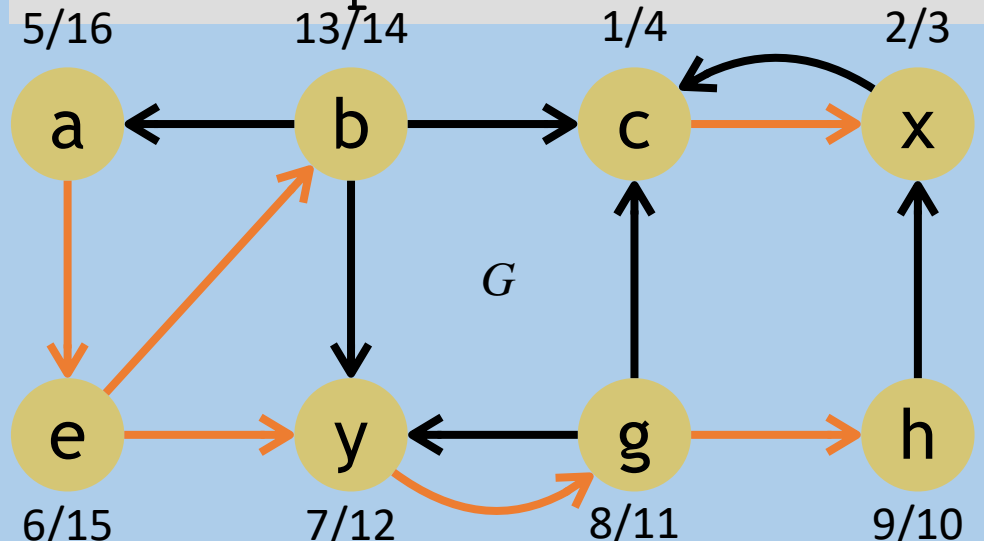
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

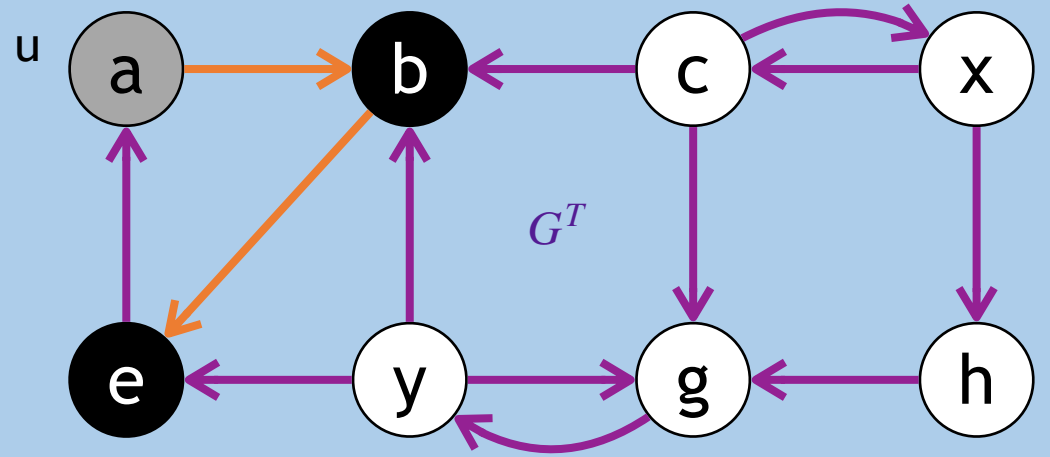




Strongly Connected Component

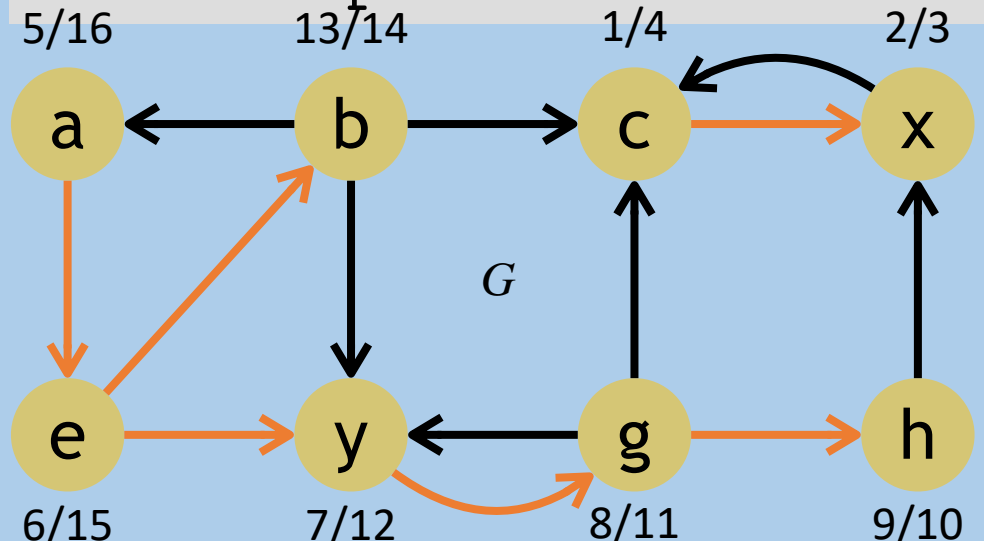
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

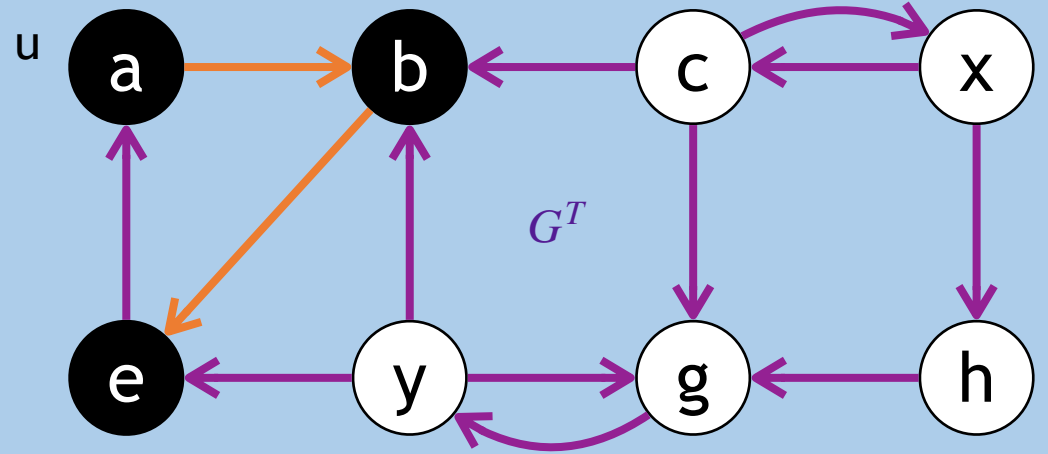




Strongly Connected Component

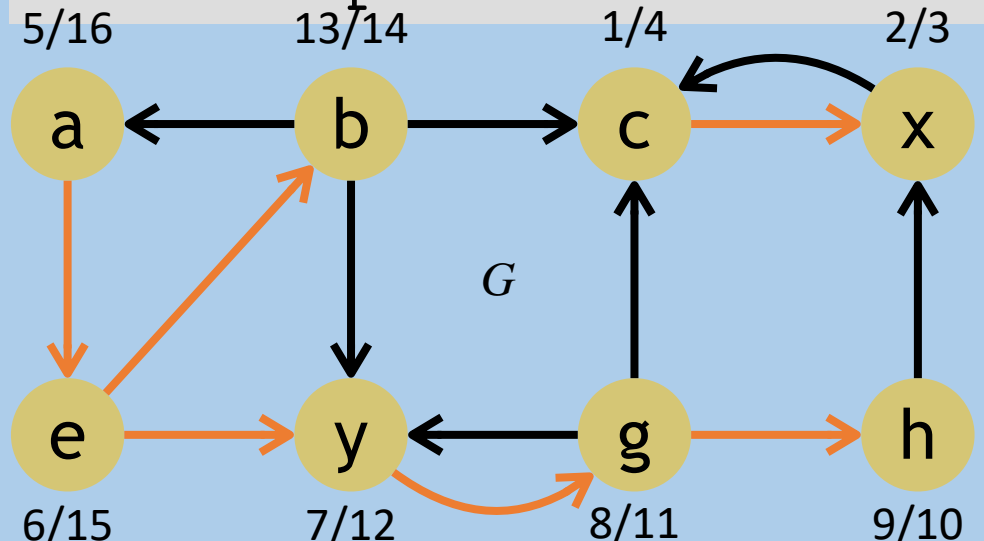
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

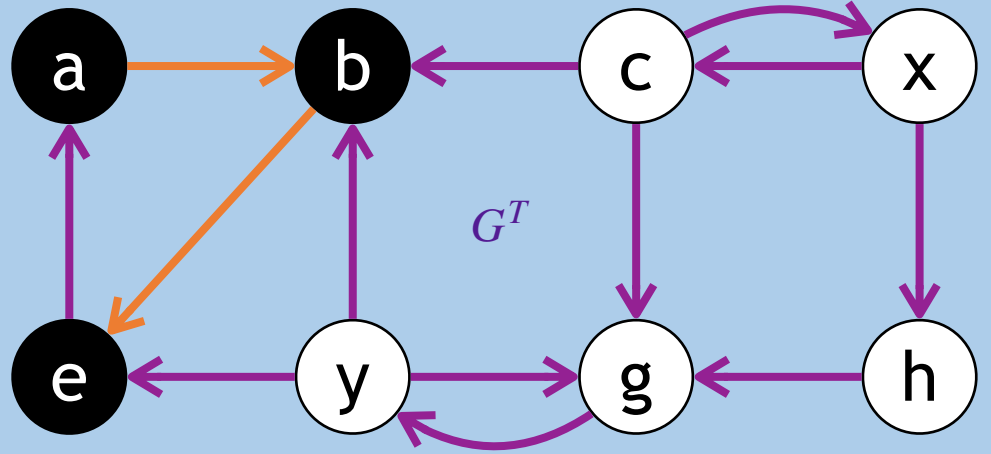




Strongly Connected Component

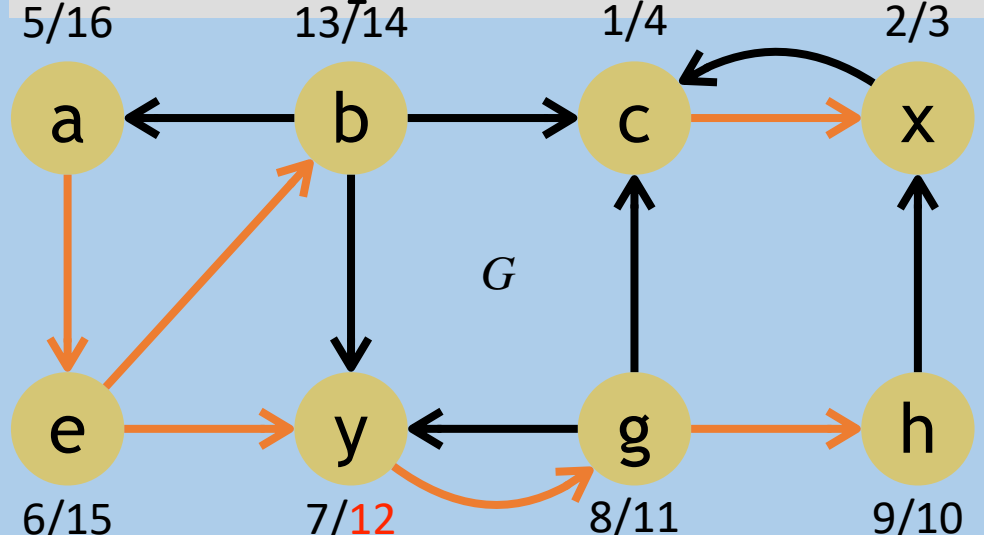
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

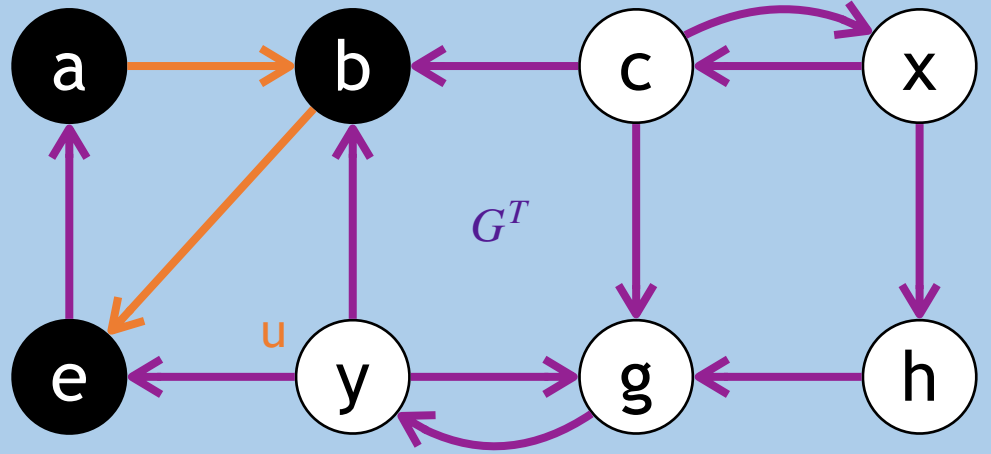




Strongly Connected Component

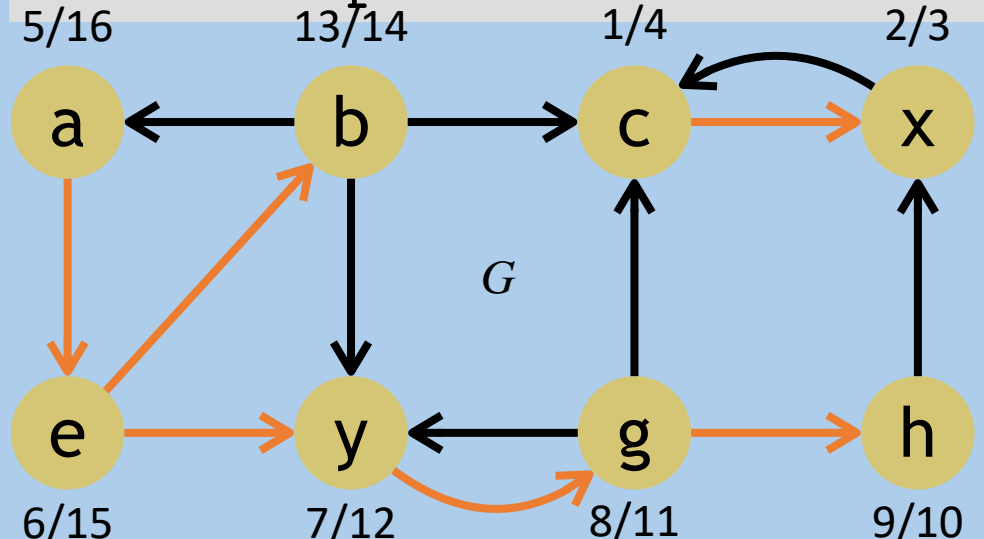
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

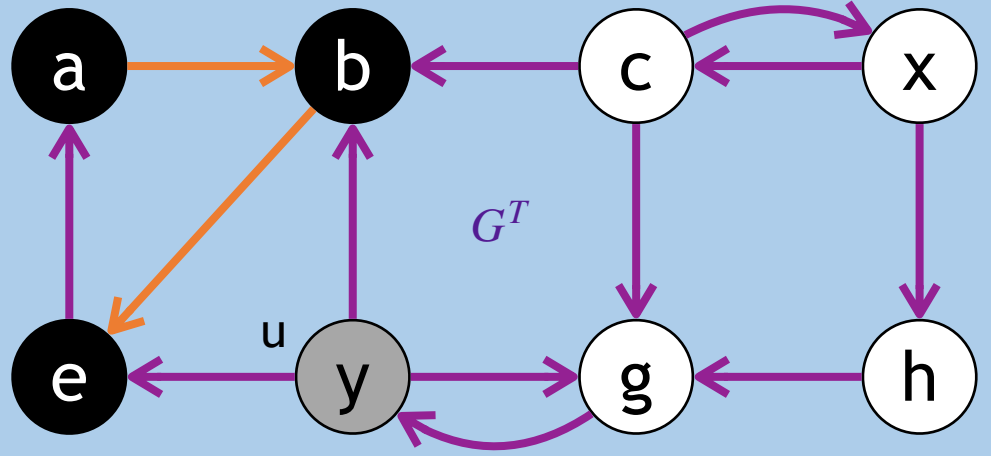




Strongly Connected Component

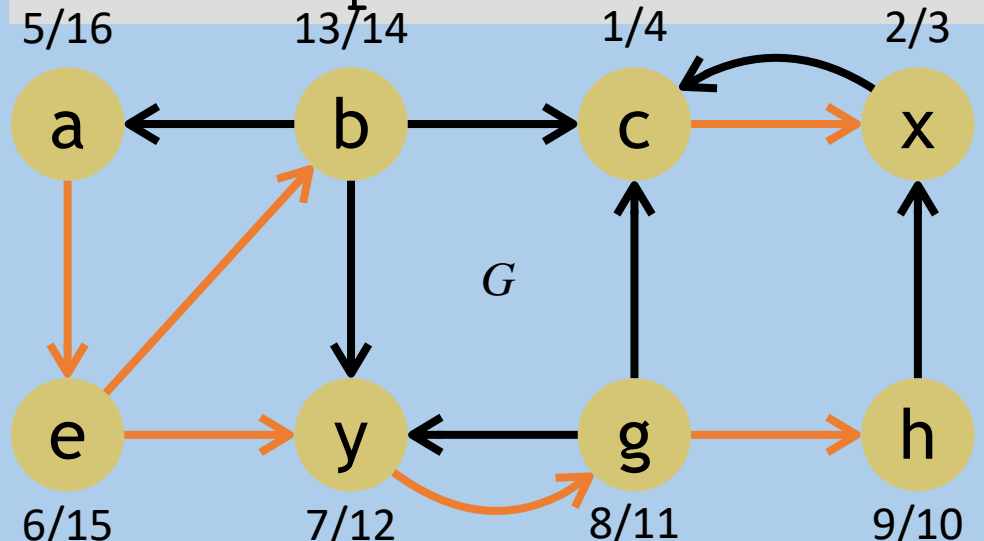
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

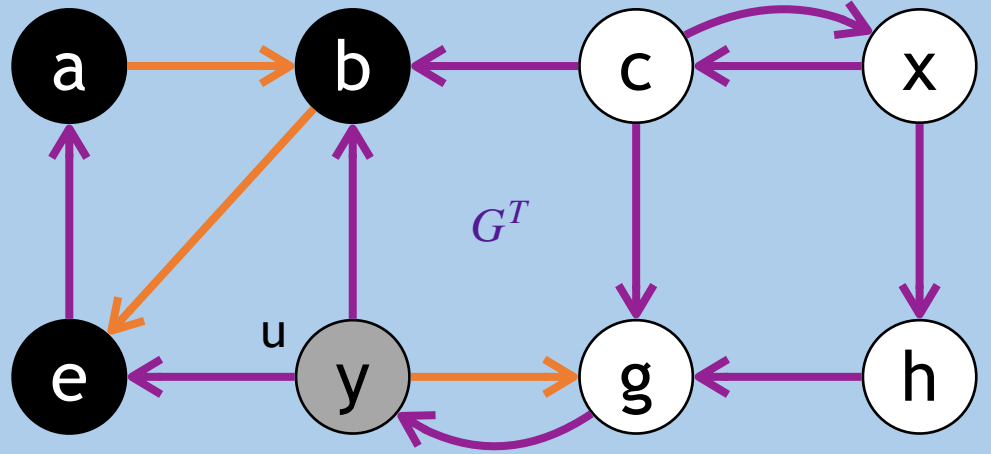




Strongly Connected Component

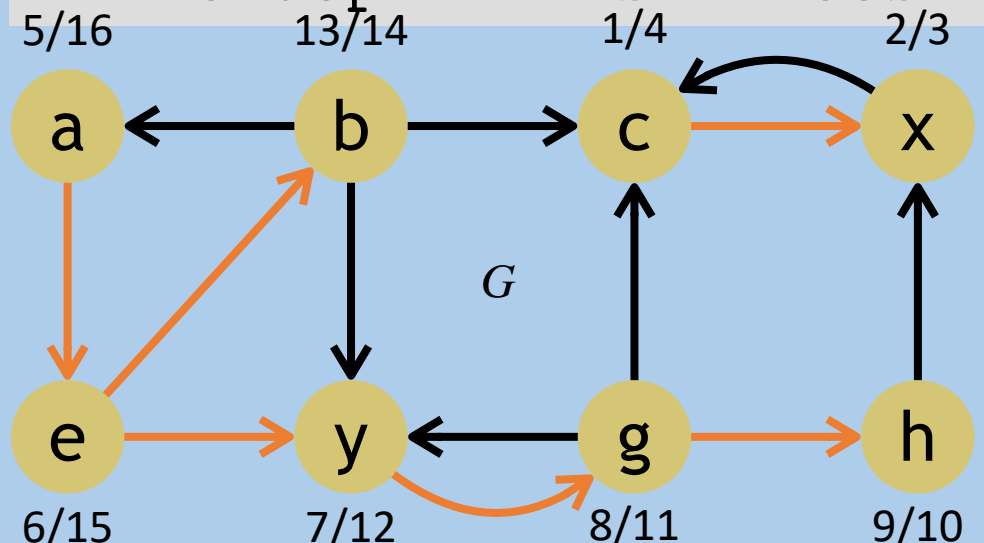
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

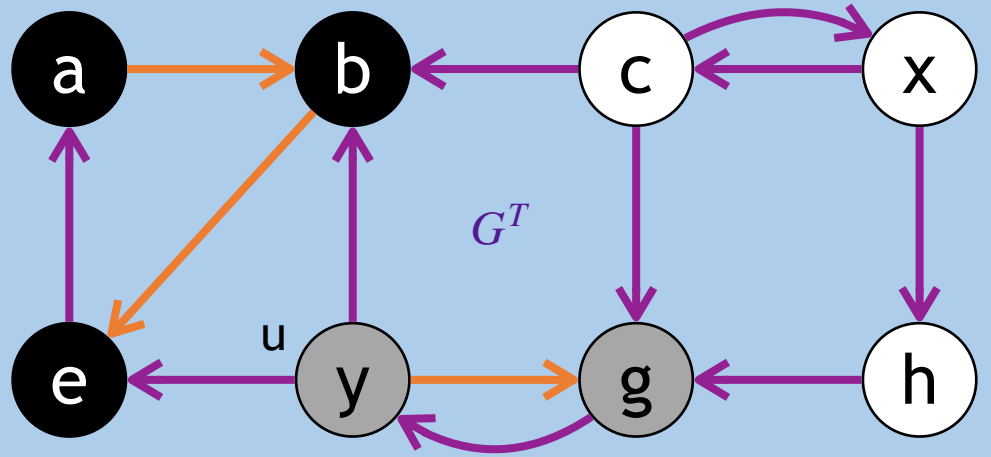




Strongly Connected Component

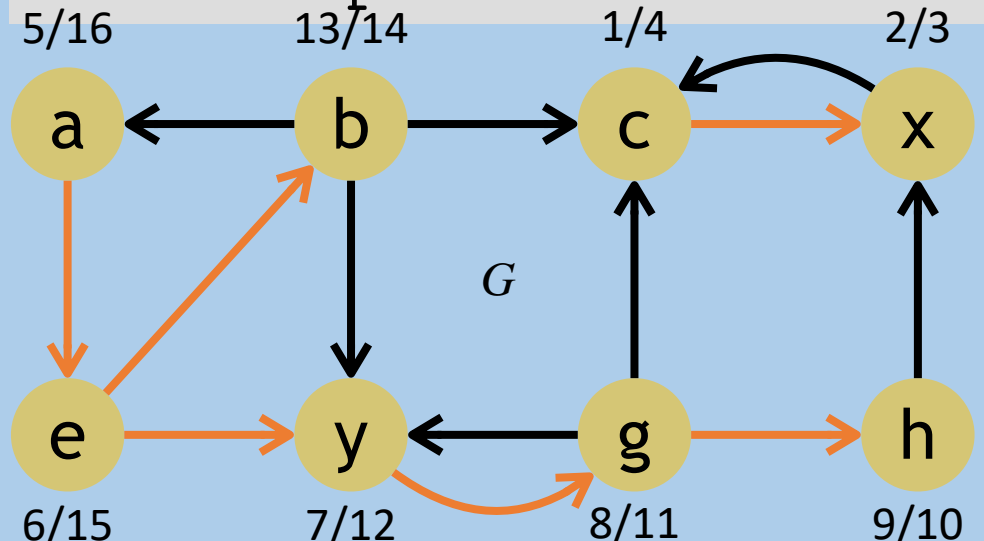
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

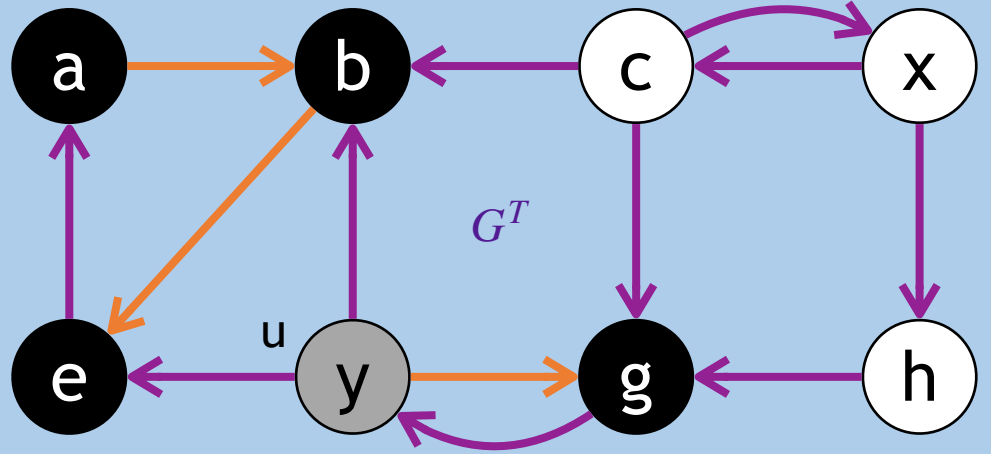




Strongly Connected Component

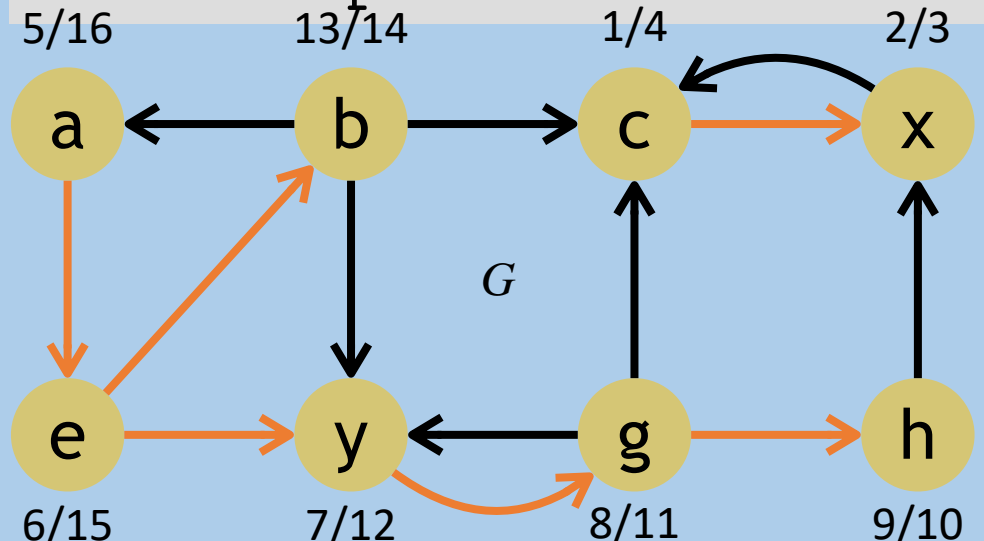
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

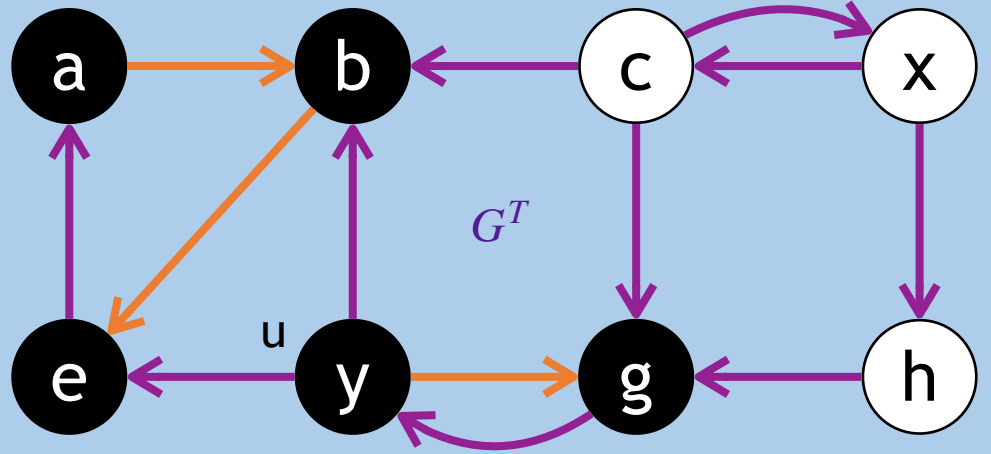




Strongly Connected Component

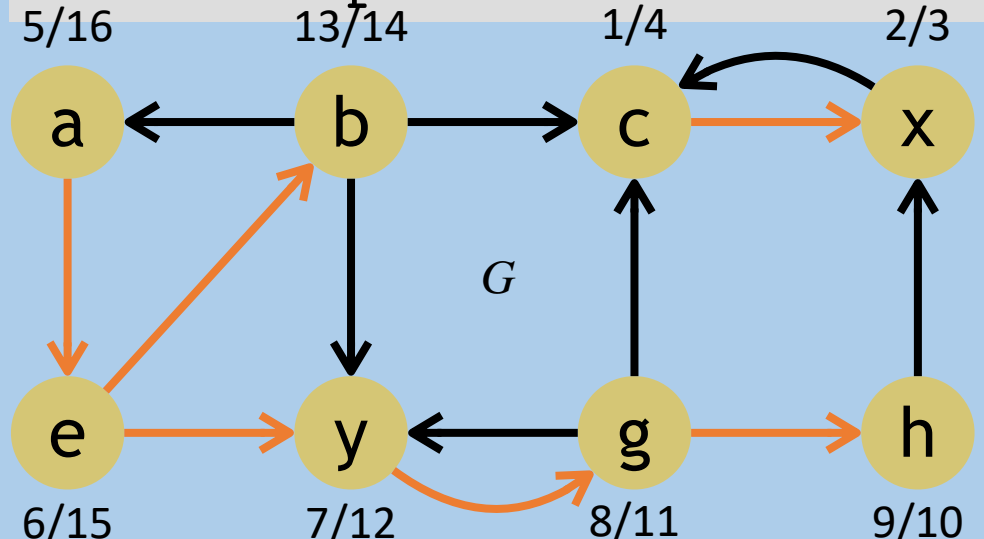
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

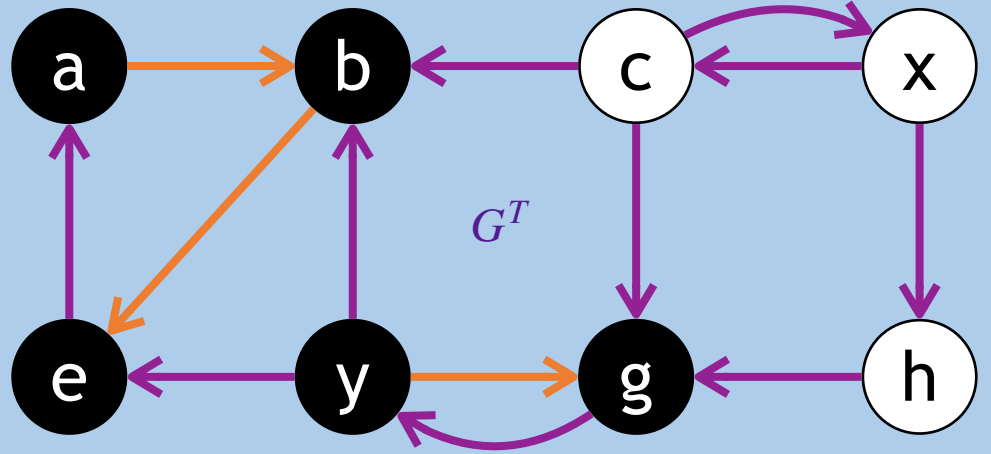




Strongly Connected Component

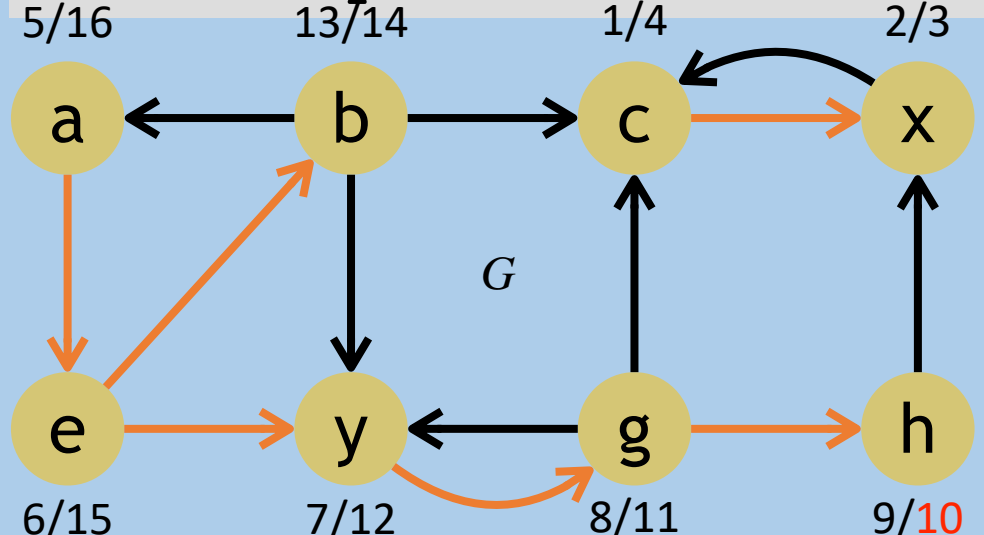
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

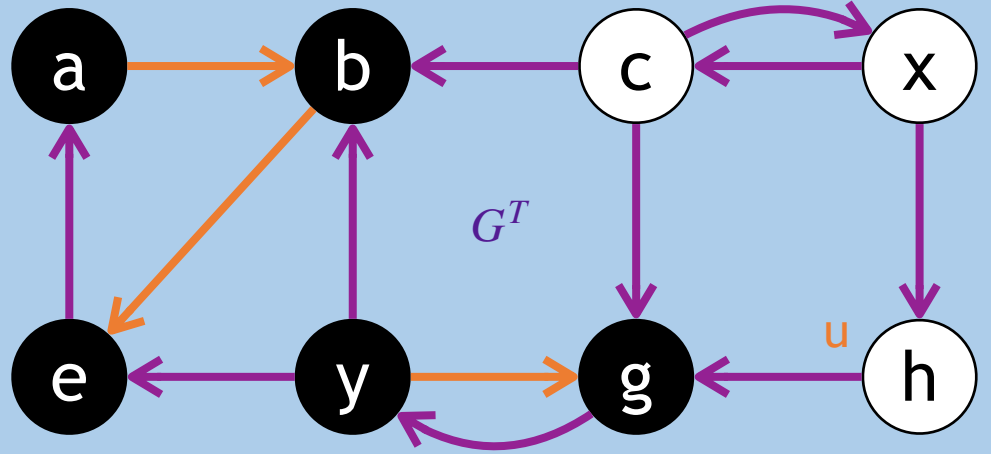




Strongly Connected Component

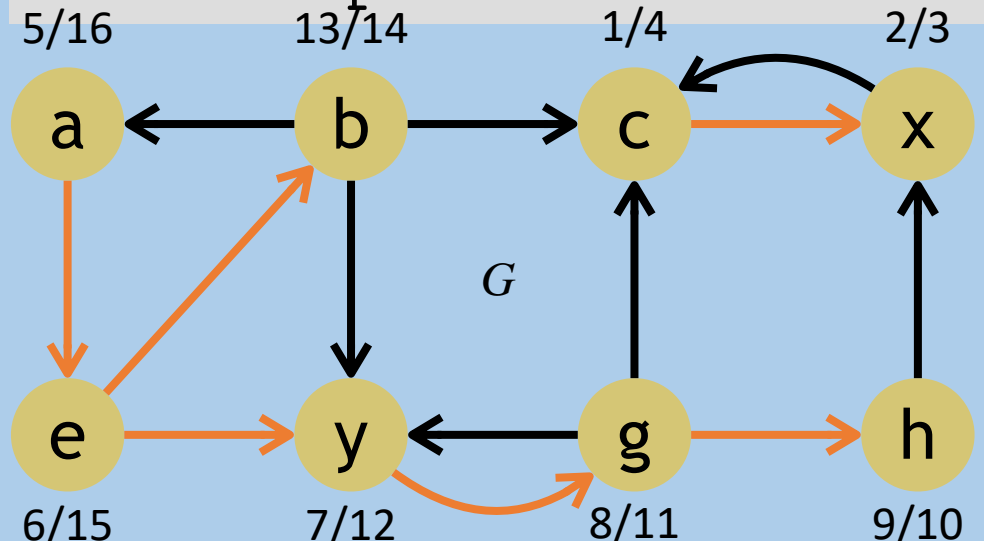
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

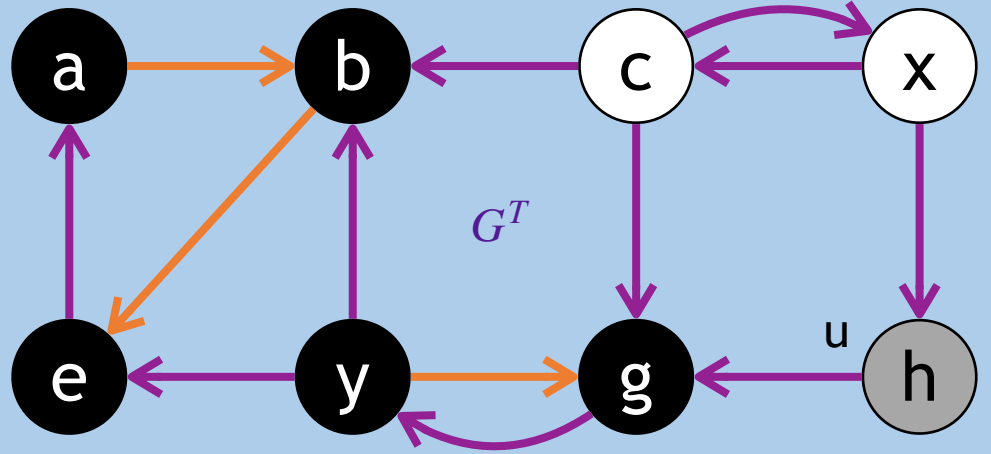




Strongly Connected Component

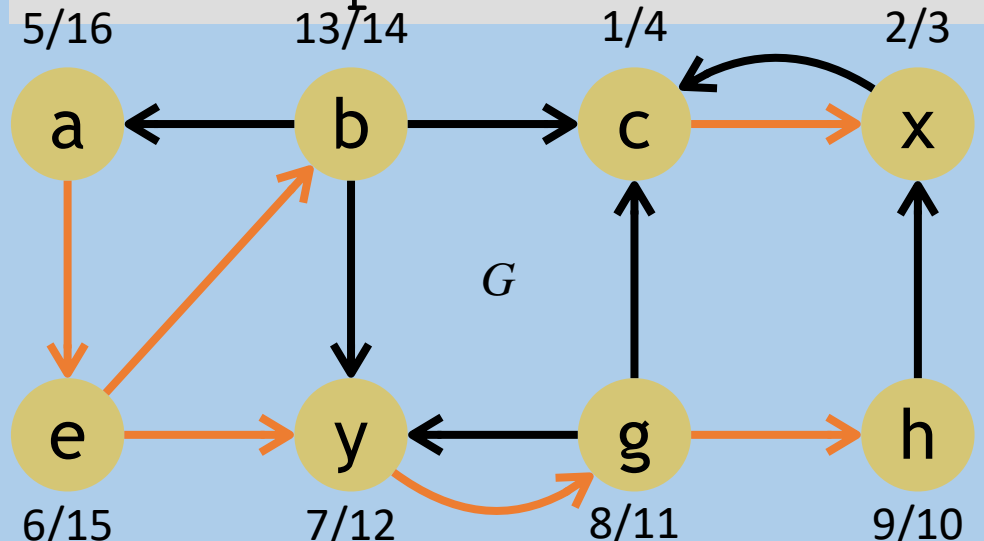
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

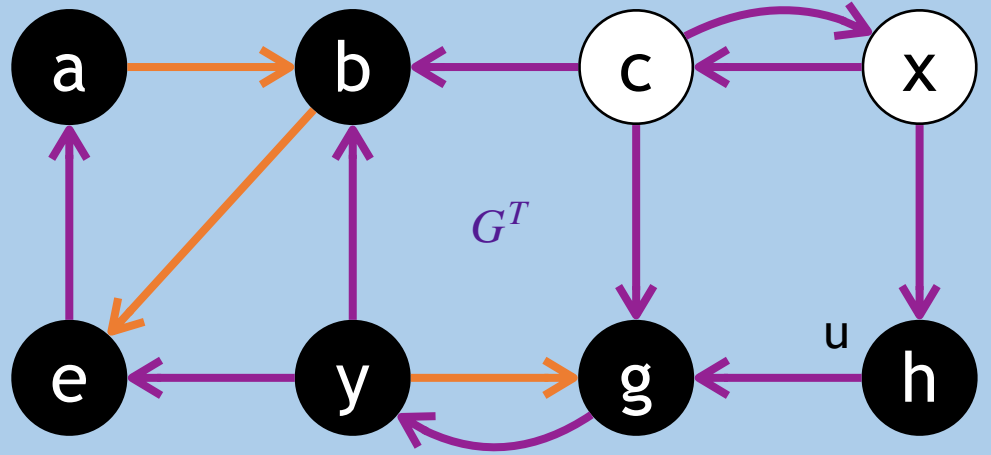




Strongly Connected Component

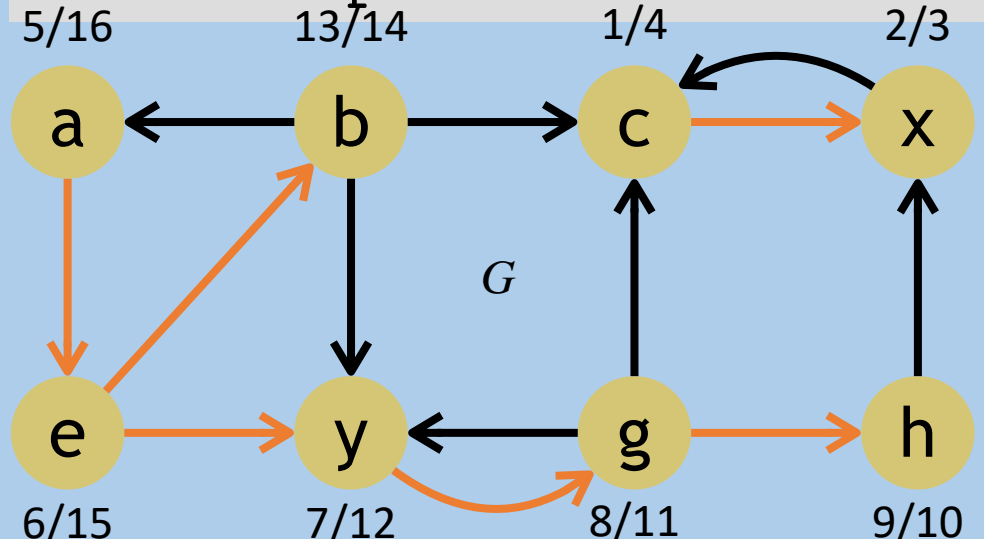
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

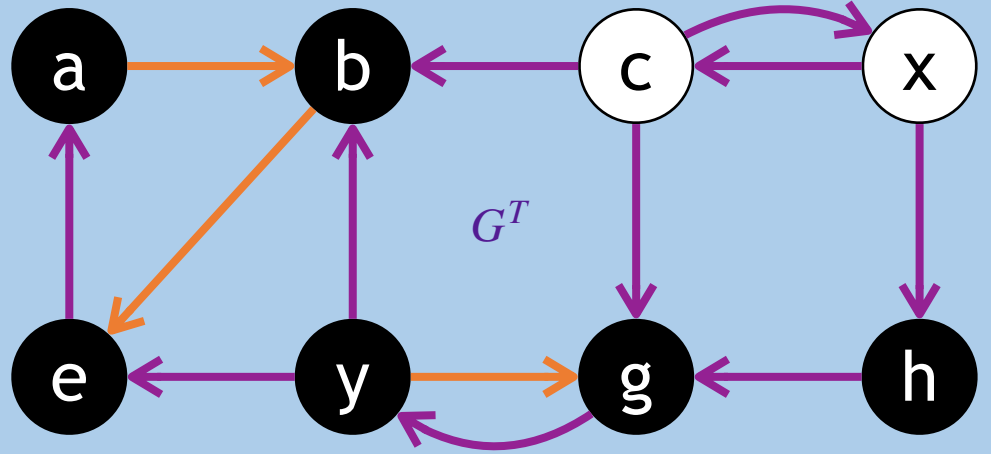




Strongly Connected Component

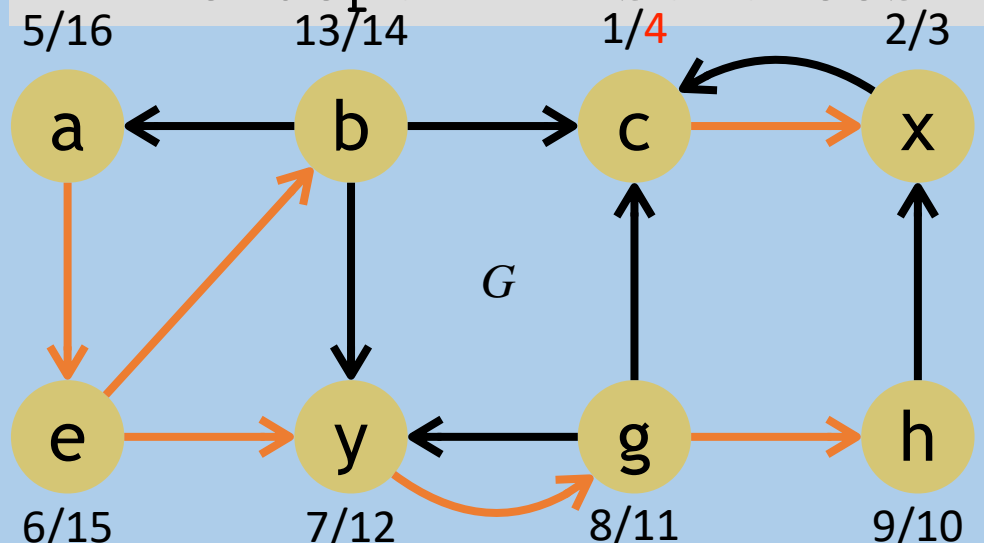
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

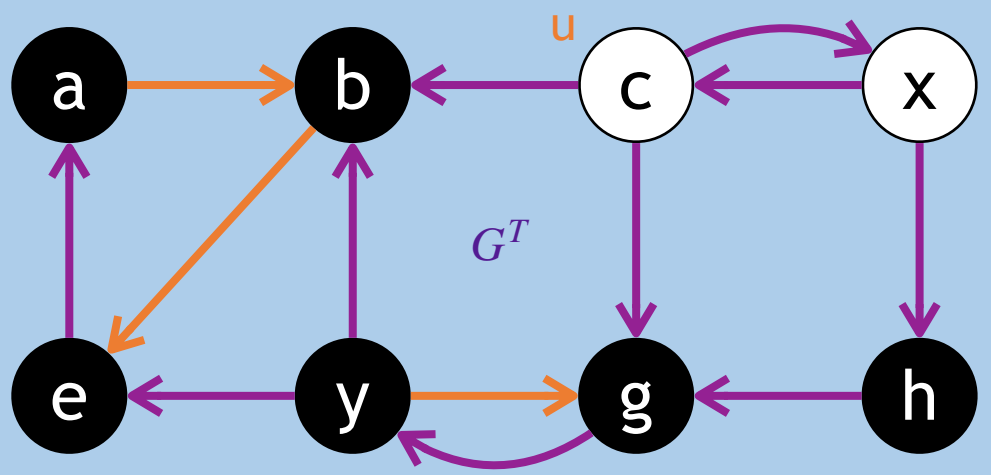




Strongly Connected Component

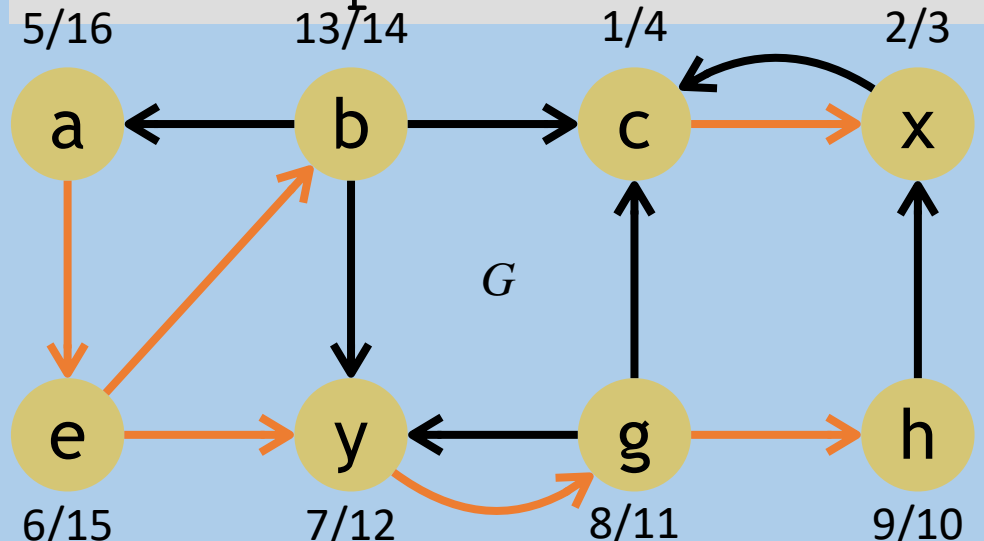
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

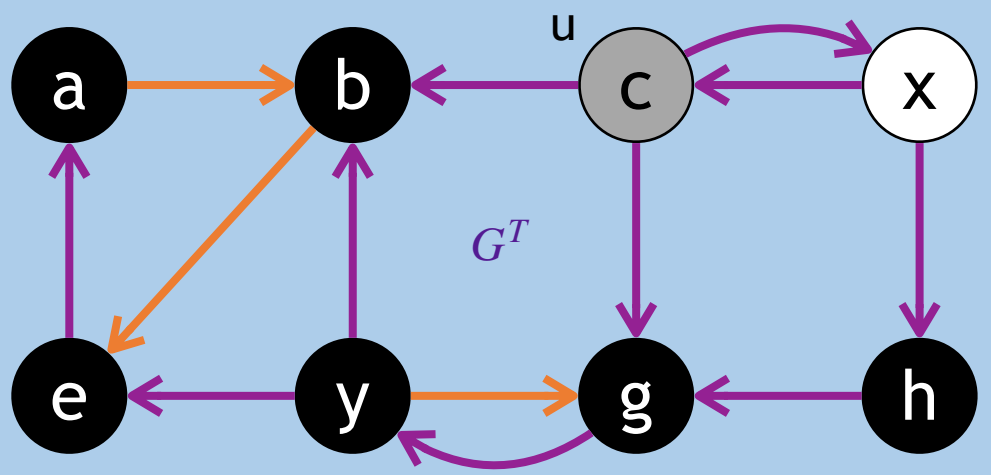




Strongly Connected Component

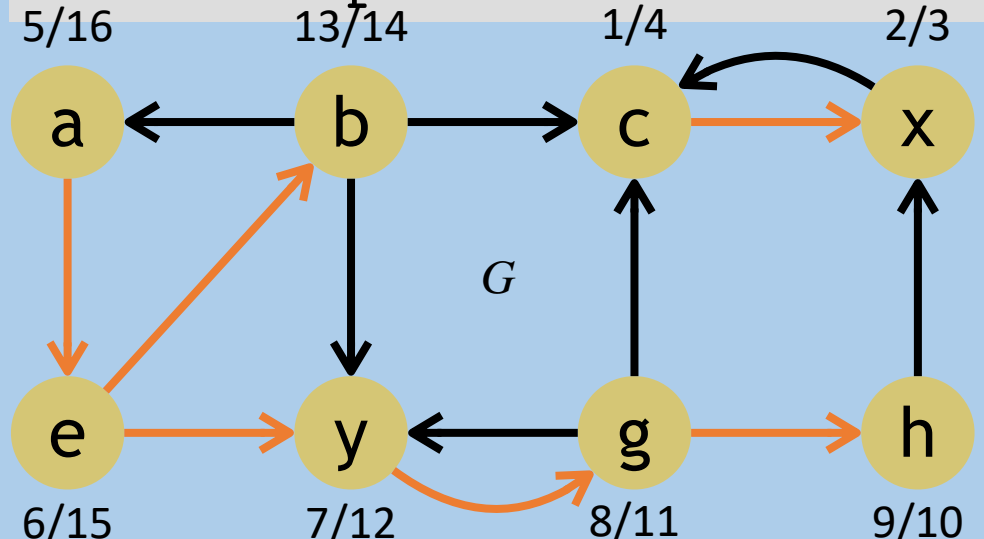
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

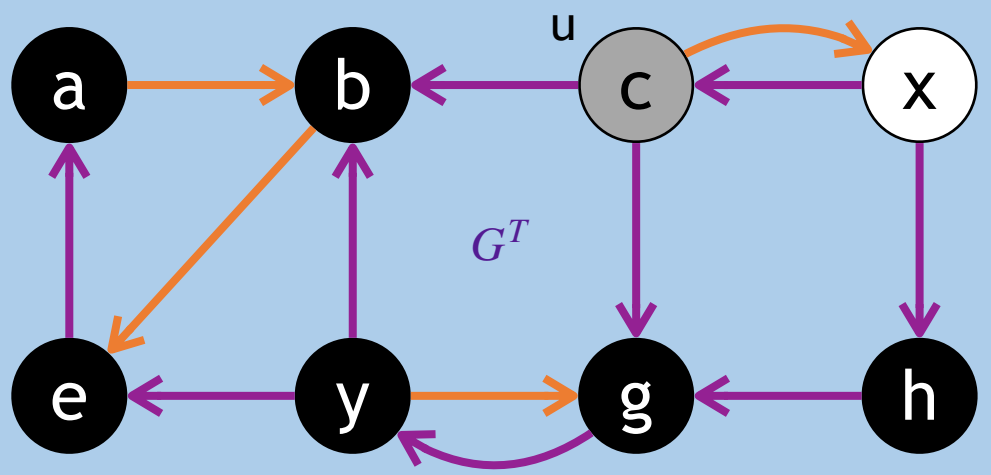




Strongly Connected Component

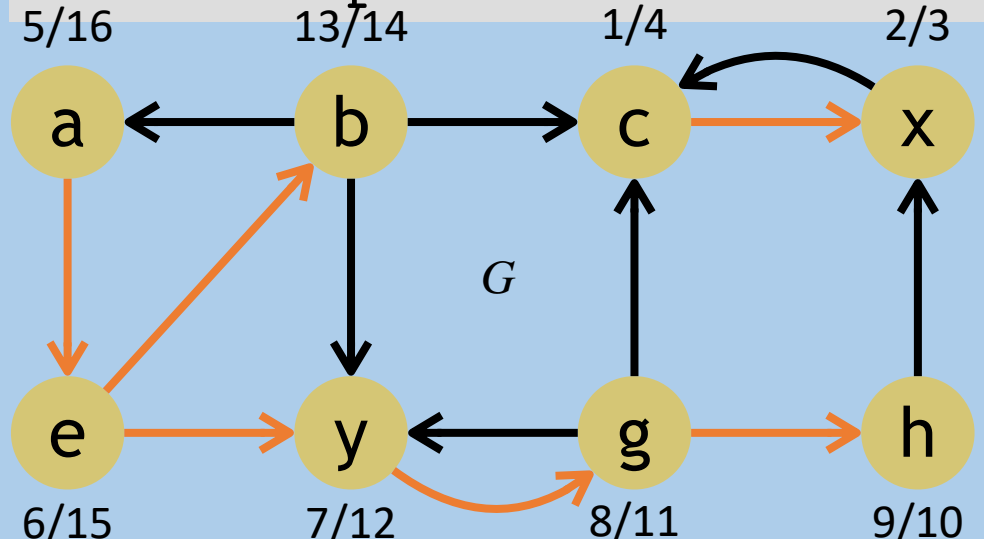
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

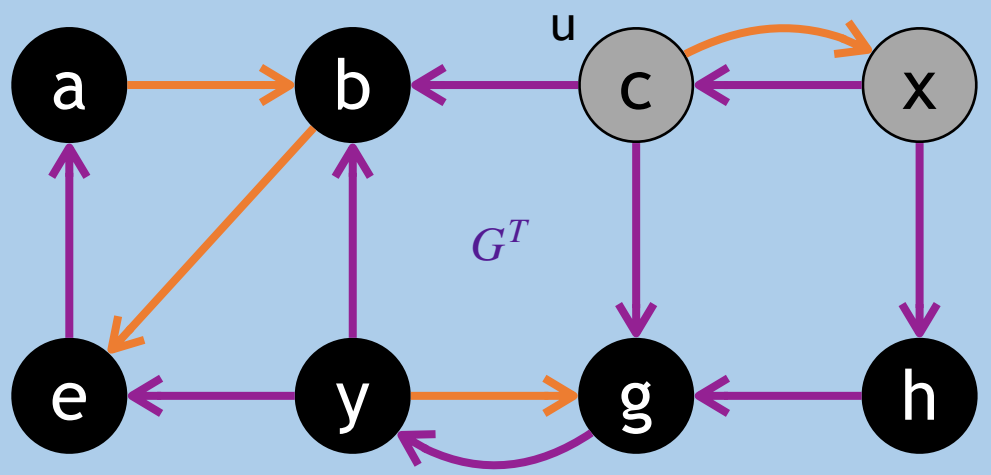




Strongly Connected Component

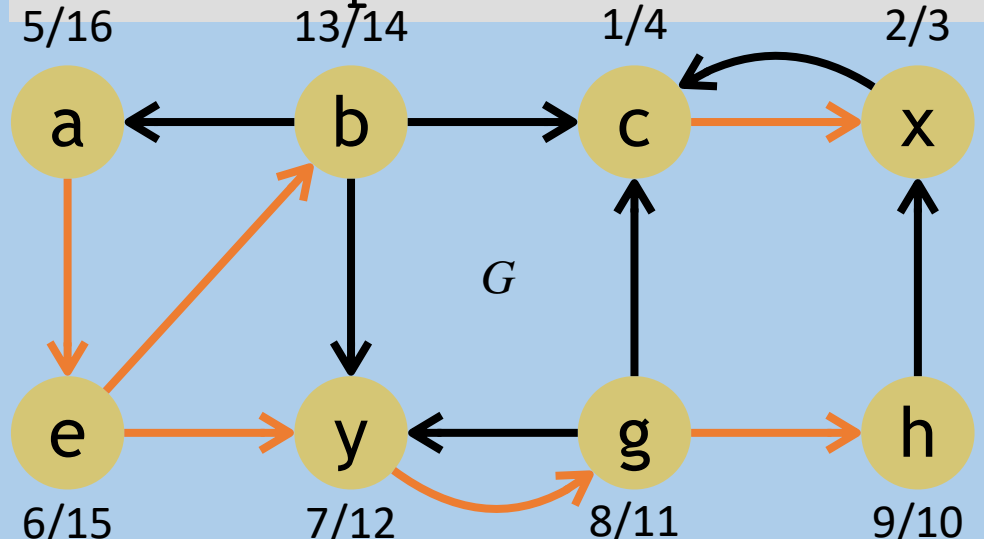
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

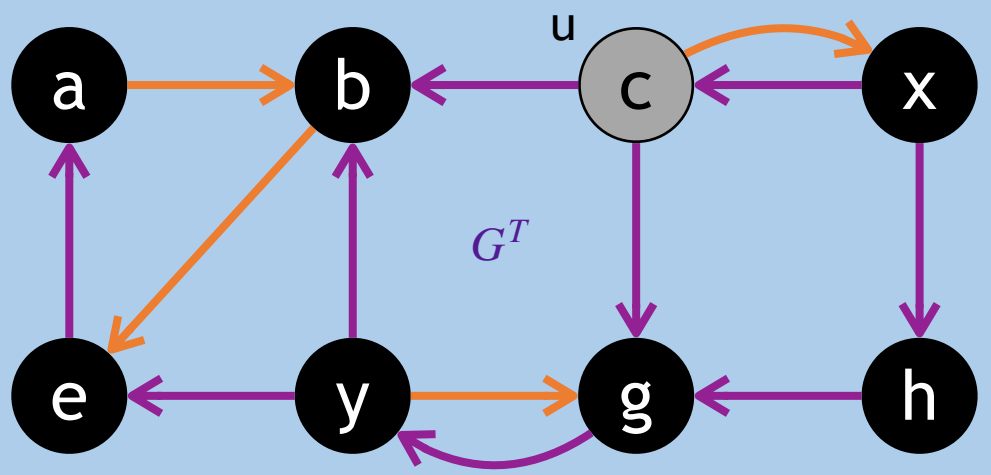




Strongly Connected Component

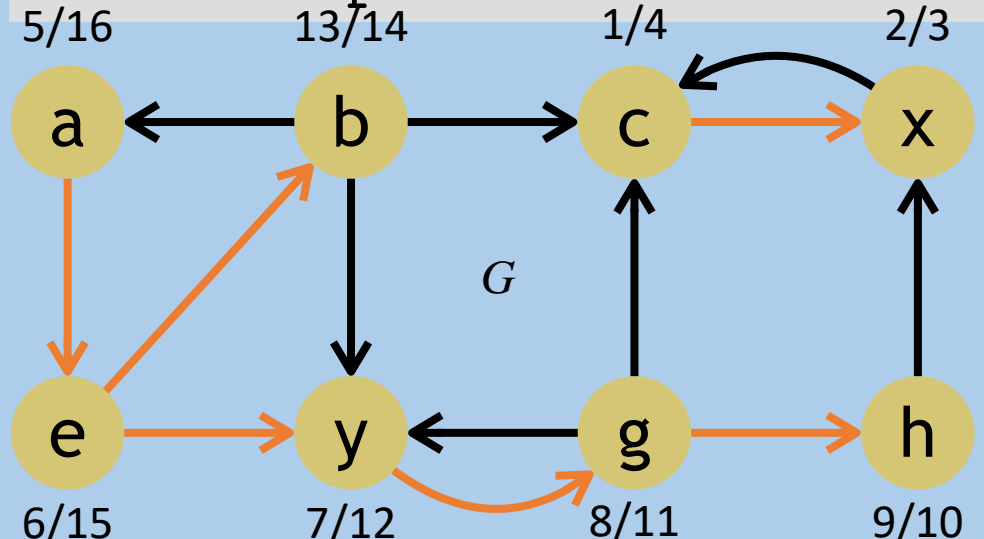
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

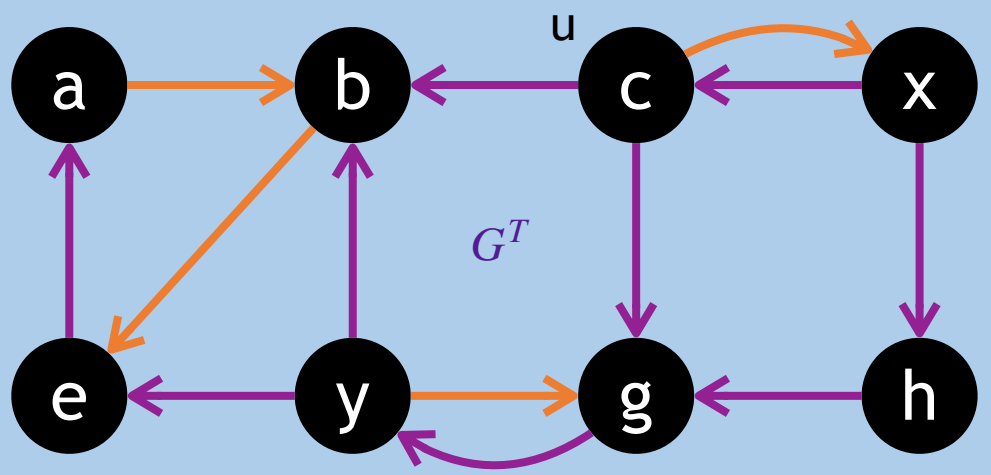




Strongly Connected Component

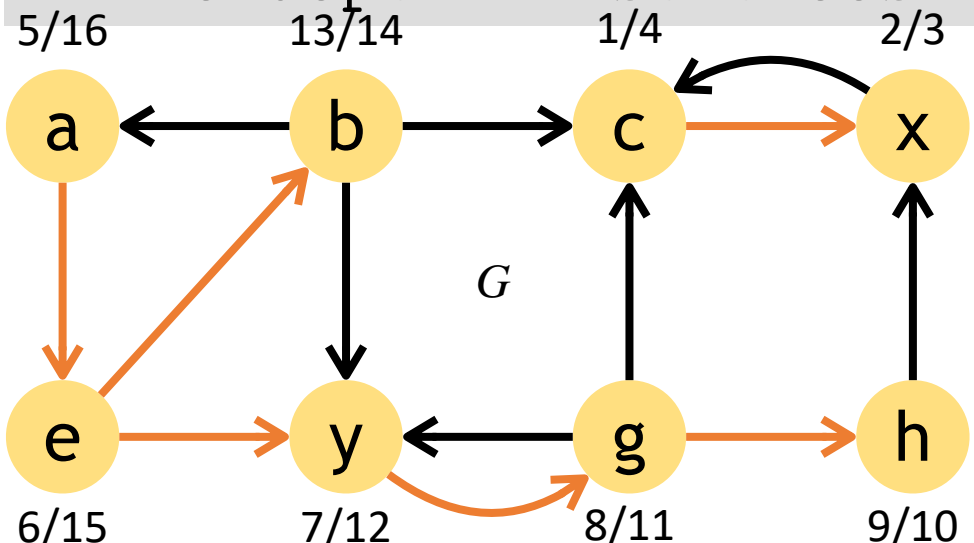
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

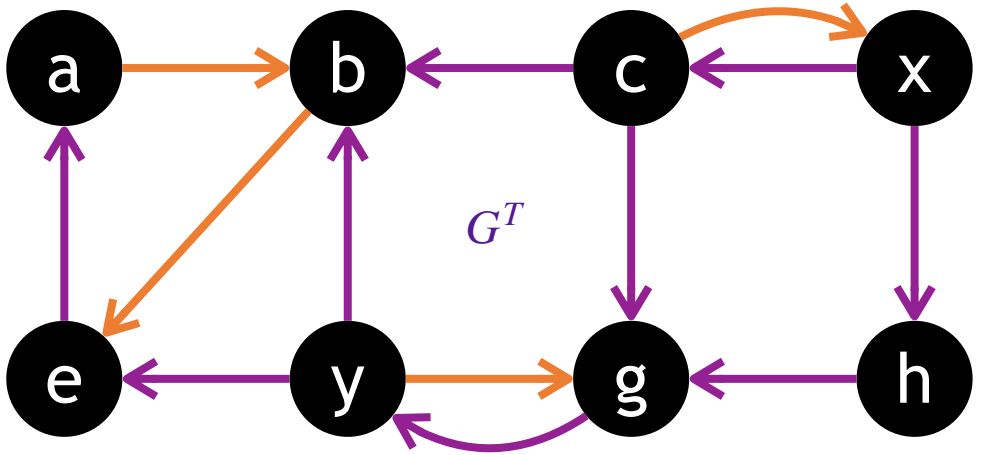




Strongly Connected Component

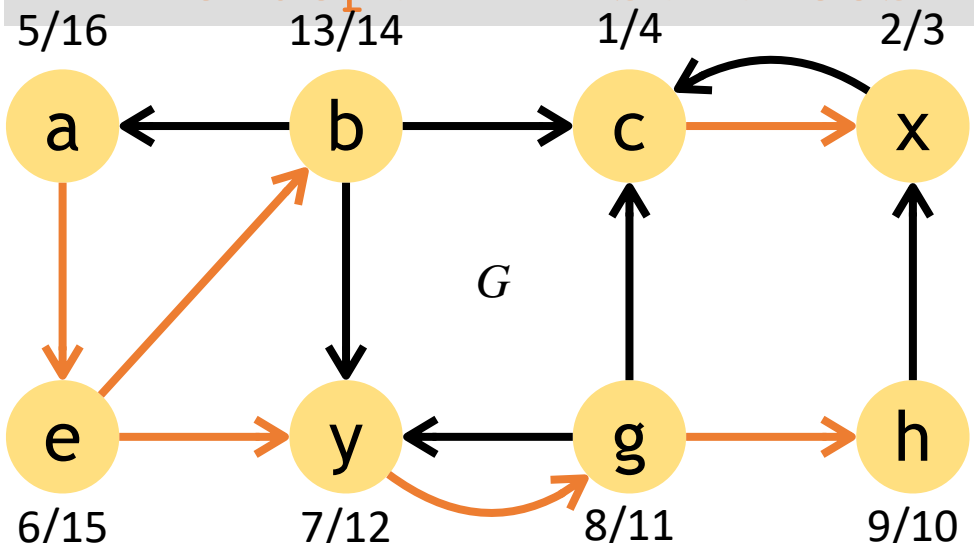
Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs

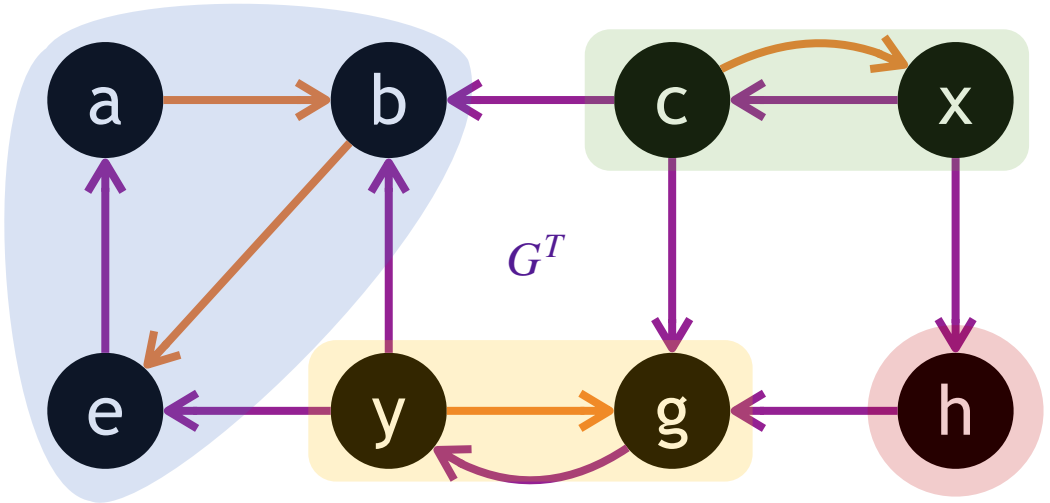




Strongly Connected Component

Procedure **STRONGLY-CONNECTED-COMPONENTS**(G)
  Call **DFS**(G) and compute finishing times $f[u]$ for each u
  Compute G^T
  While (some node in G^T is not-discovered){
    $u \leftarrow$ not-discovered node with latest $f[u]$
    Call **DFS-VISIT**(u) on G^T}
  The depth-first trees in **DFS**(G^T) are SCCs





What's Happening

- Given a directed graph, we can find the strongly connect components by performing two DFSes: the first one is on the original graph, and the second one is on the transpose graph.

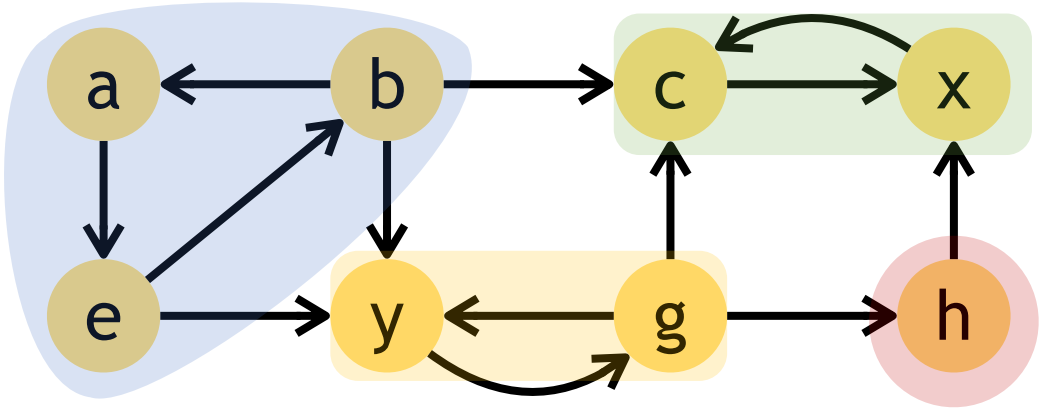
- In the second DFS on the transpose graph, it is important that when you are selecting the vertex to start a new DFS tree, you choose the one with the largest finish time in the first DFS.

Component graph

Definition: Given a graph $G = (V, E)$ and suppose G has strongly connected components $C_1, C_2, \cdots, C_k$. Its **component graph** $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$ is a directed graph such that:

- $V^{\text{SCC}} = \{v_1, v_2, \cdots, v_k\}$, where v_i corresponds to C_i for each i
- $(v_i, v_j) \in E^{\text{SCC}}$ if and only if there is a $x \in C_i$, $y \in C_j$ and $(x, y) \in E$

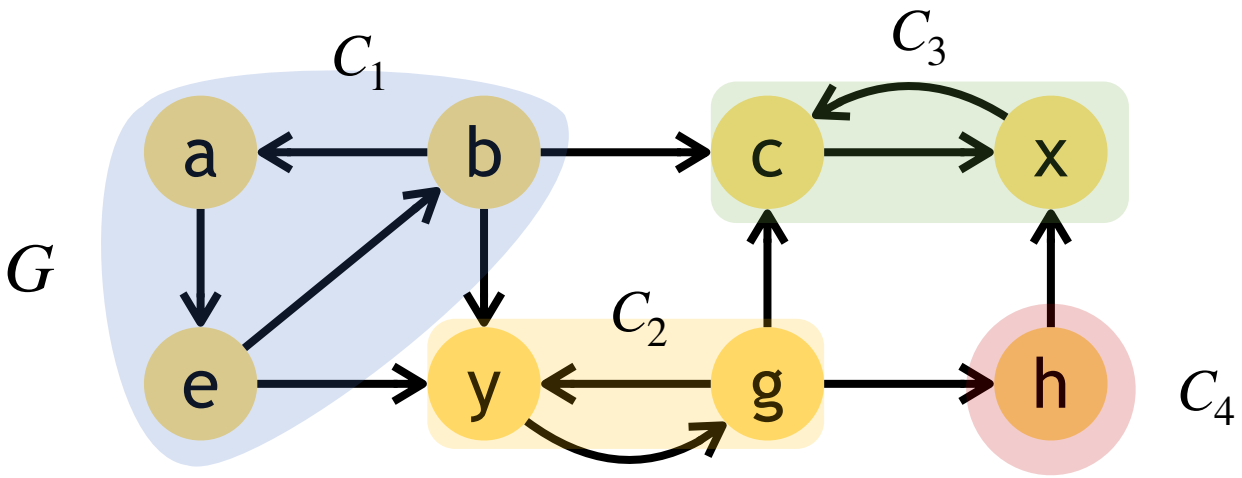
G



Component graph

Definition: Given a graph $G = (V, E)$ and suppose G has strongly connected components $C_1, C_2, \cdots, C_k$. Its **component graph** $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$ is a directed graph such that:

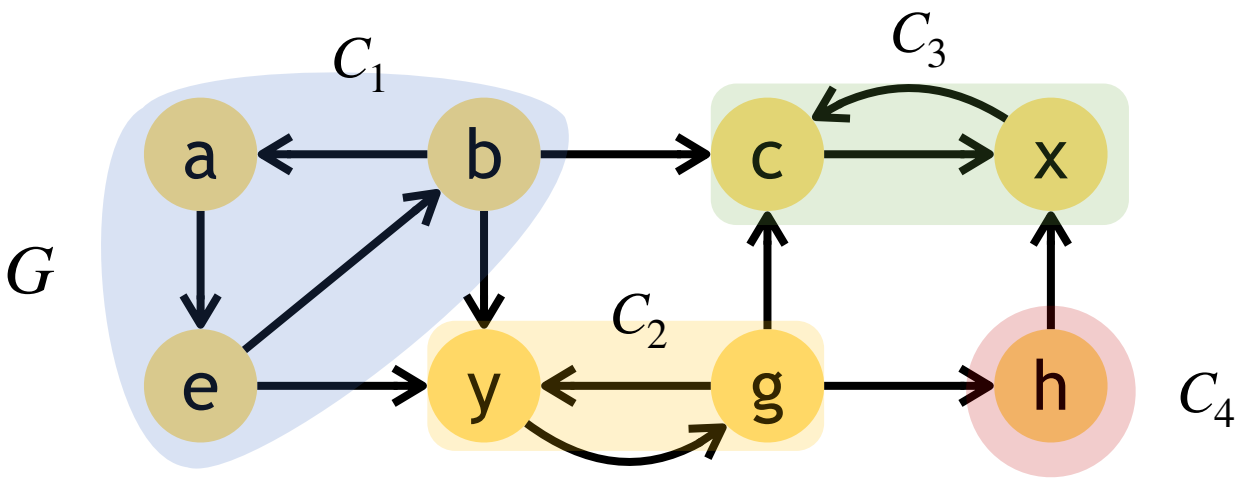
- $V^{\text{SCC}} = \{v_1, v_2, \cdots, v_k\}$, where v_i corresponds to C_i for each i
- $(v_i, v_j) \in E^{\text{SCC}}$ if and only if there is a $x \in C_i$, $y \in C_j$ and $(x, y) \in E$

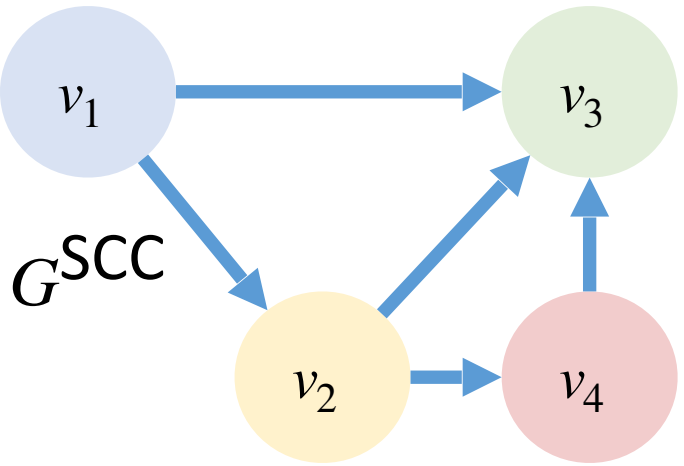


Component graph

Definition: Given a graph $G = (V, E)$ and suppose G has strongly connected components $C_1, C_2, \cdots, C_k$. Its **component graph** $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$ is a directed graph such that:

- $V^{\text{SCC}} = \{v_1, v_2, \cdots, v_k\}$, where v_i corresponds to C_i for each i
- $(v_i, v_j) \in E^{\text{SCC}}$ if and only if there is a $x \in C_i$, $y \in C_j$ and $(x, y) \in E$



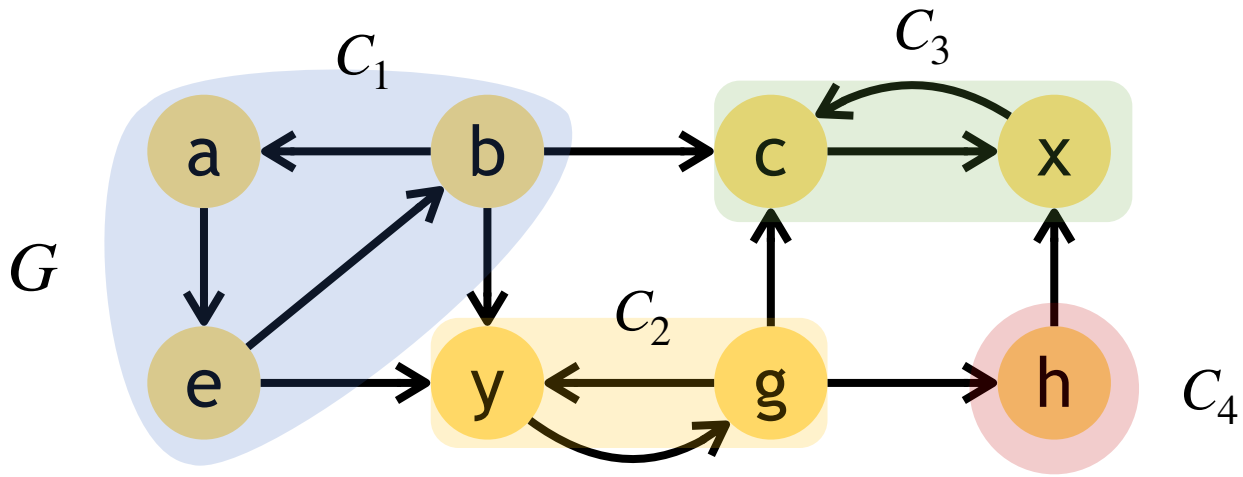


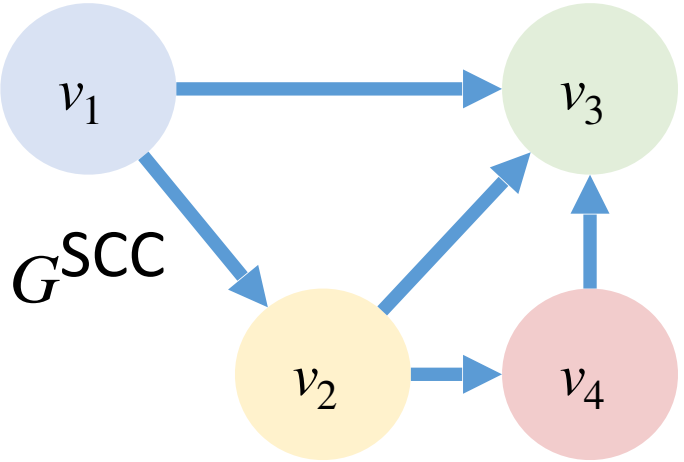
Correctness of the SCC Algorithm

The component graph G^{SCC} should be a DAG

Correctness of the SCC Algorithm

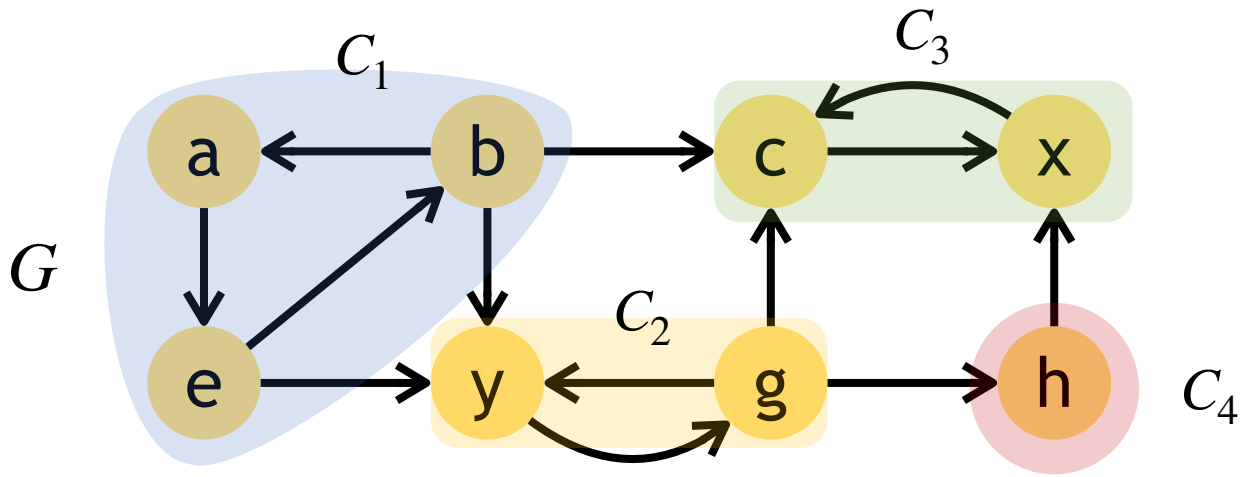
The component graph G^{SCC} should be a DAG

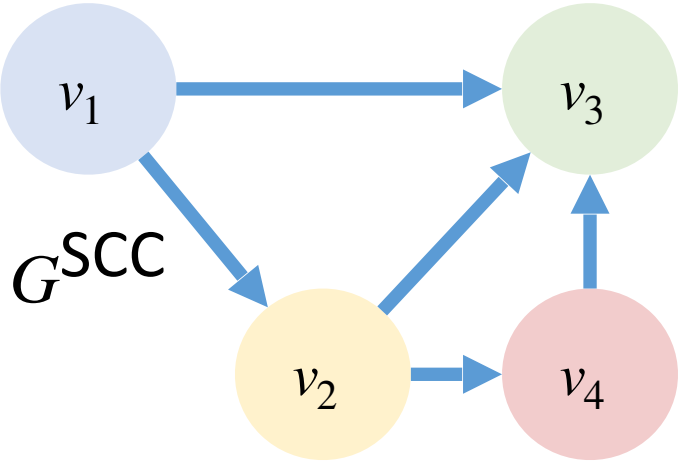




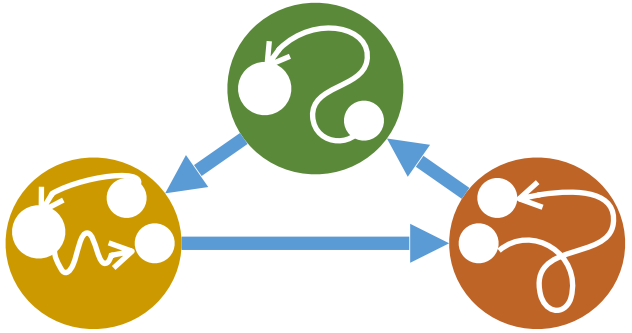
Correctness of the SCC Algorithm

The component graph G^{SCC} should be a DAG



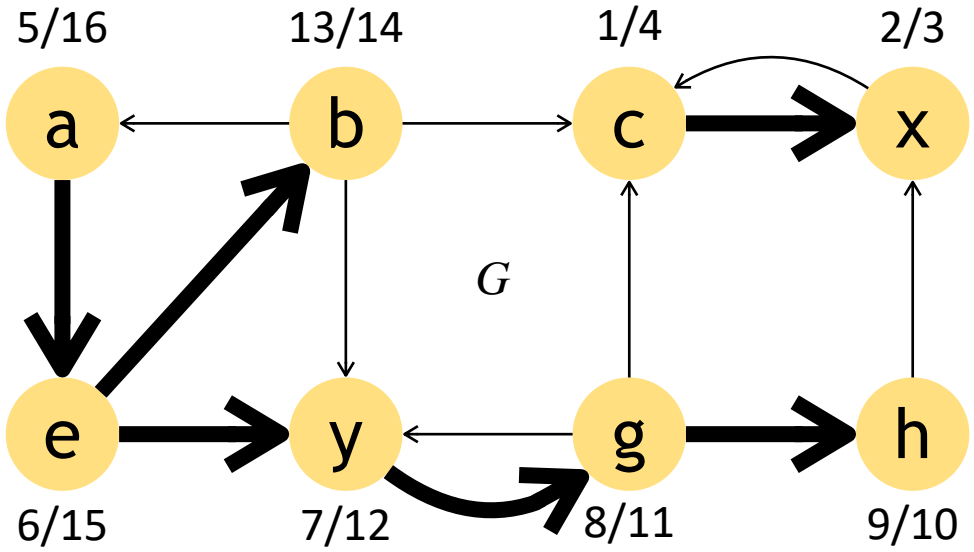


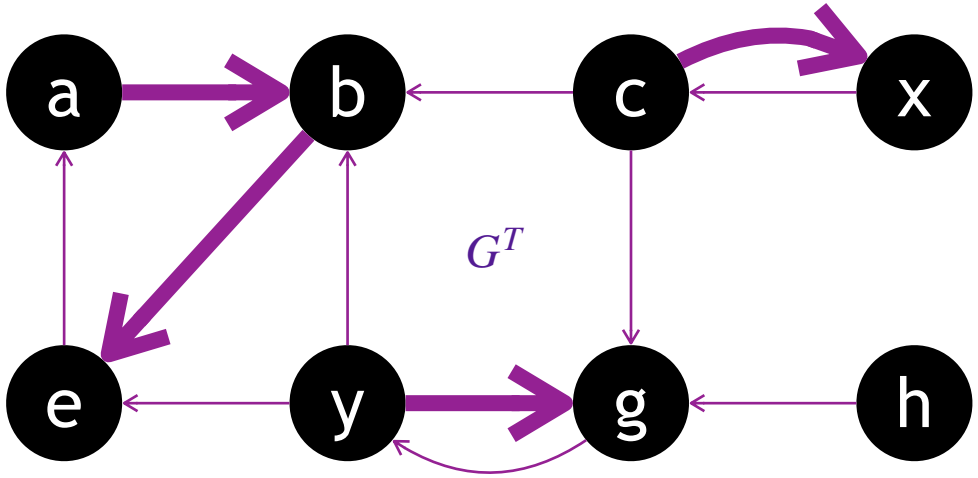
If there is a cycle in G^{SCC}, for any pair of involved components, there are mutual paths. It contradicts to the fact that the SCCs are maximal.



SCC Algorithm Correctness

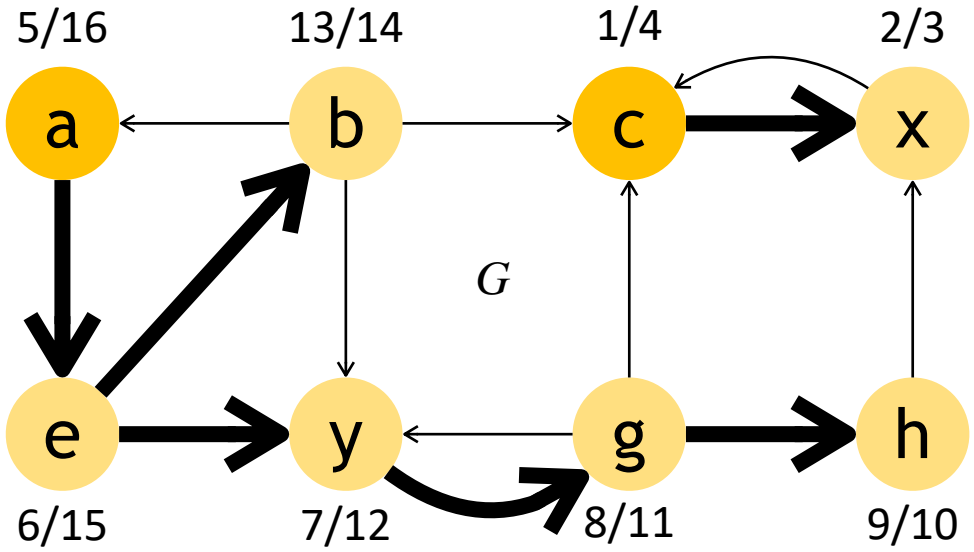
Theorem: **STRONGLY-CONNECTED-COMPONENTS**(G) correctly computes the strongly connected components of a directed graph G.

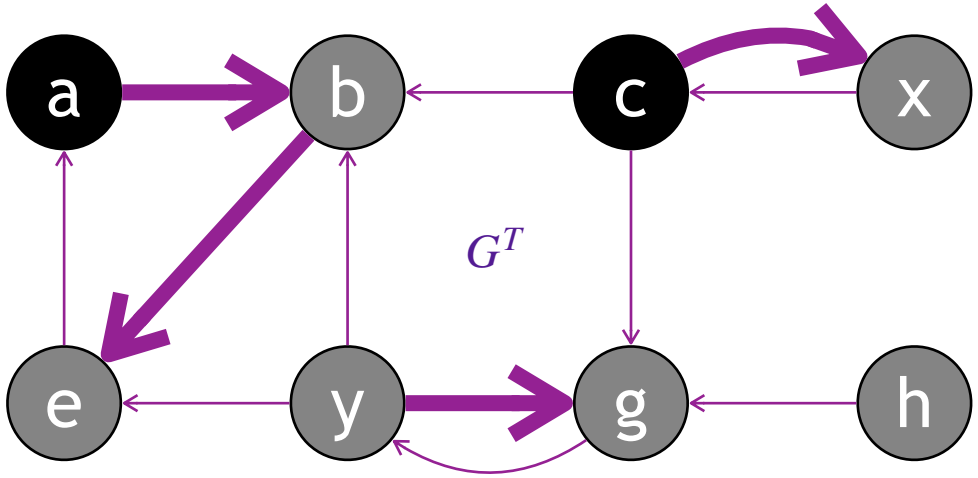




SCC Algorithm Correctness

Theorem: **STRONGLY-CONNECTED-COMPONENTS**(G) correctly computes the strongly connected components of a directed graph G.

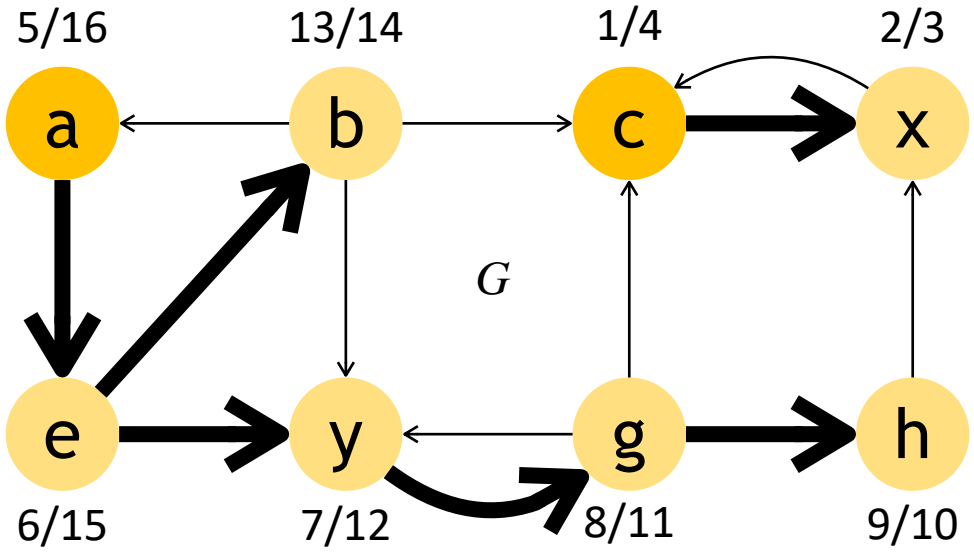


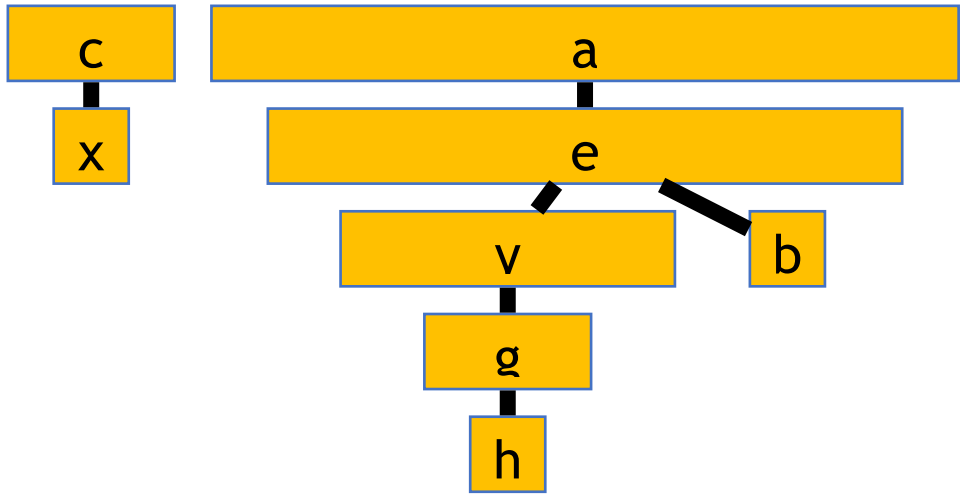


The vertex with the largest finishing time must be one of the roots of DFS trees in G

SCC Algorithm Correctness

Theorem: **STRONGLY-CONNECTED-COMPONENTS**(G) correctly computes the strongly connected components of a directed graph G.

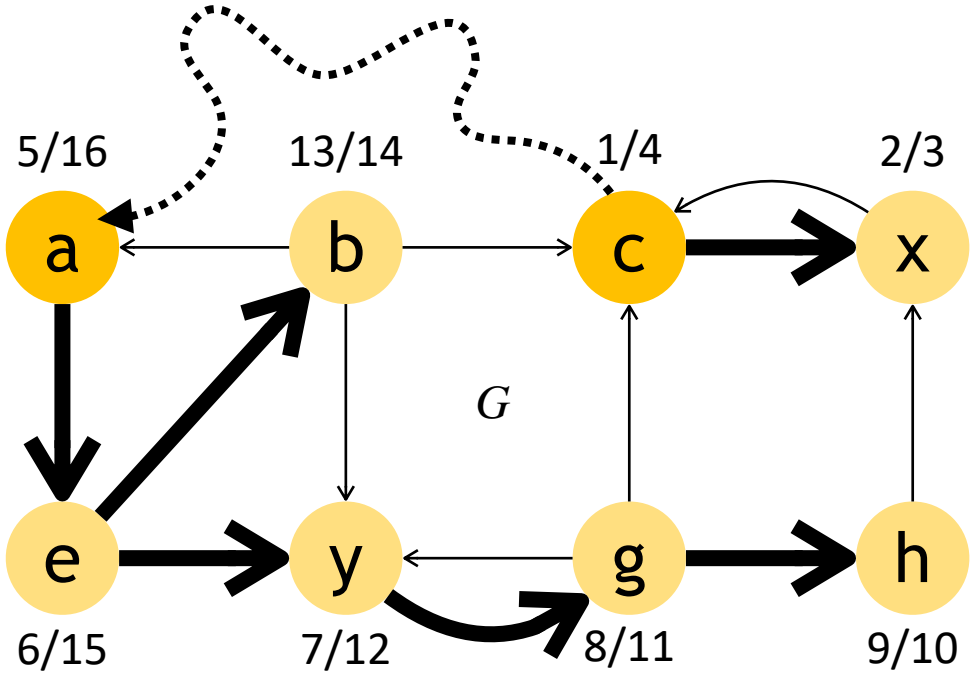




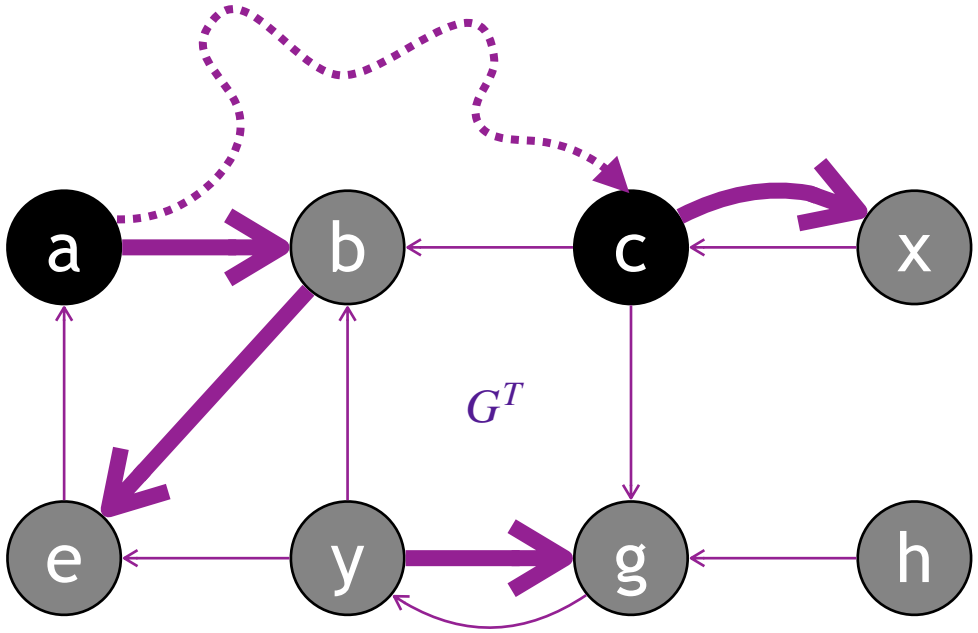
The vertex with the largest finishing time must be one of the roots of DFS trees in G

SCC Algorithm Correctness

Theorem: **STRONGLY-CONNECTED-COMPONENTS**(G) correctly computes the strongly connected components of a directed graph G.



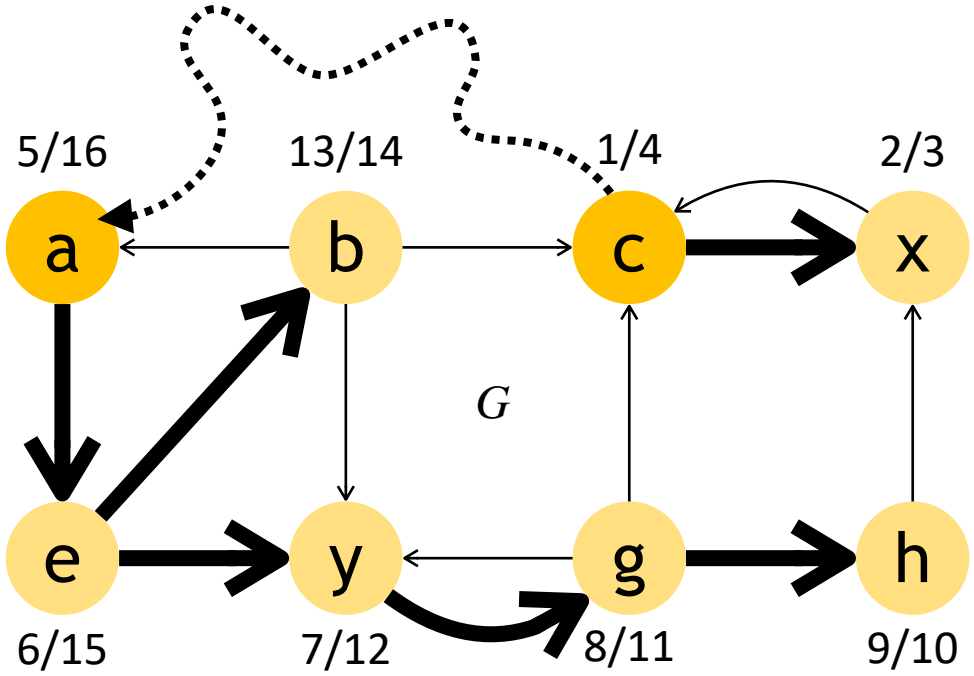
There is no path from c (root with a smaller finish time) to a (root with a larger finish time) in G



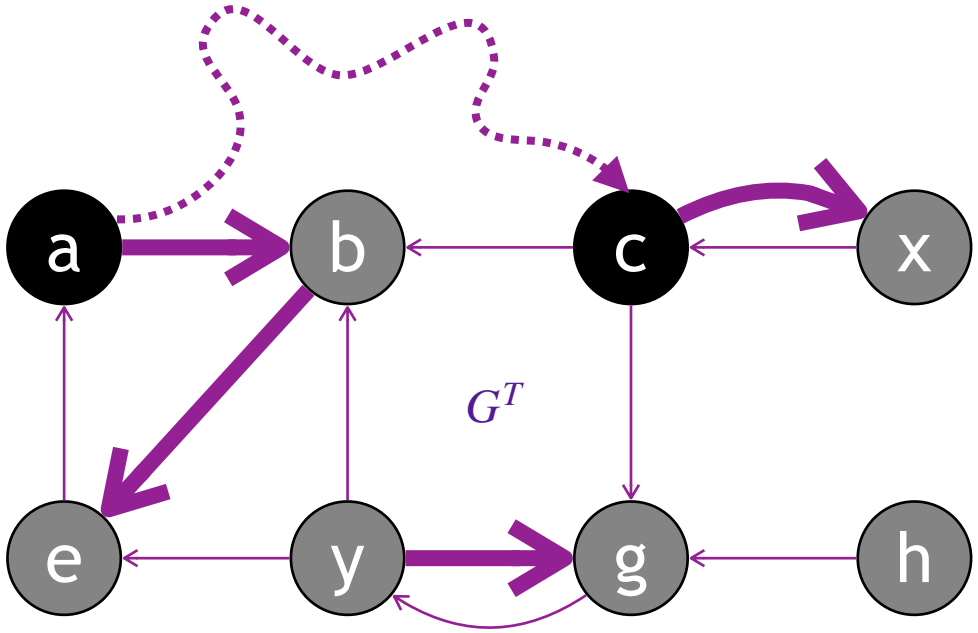
There is no path from a to c in G^T
$\Rightarrow$ DVS-VISIT(a) only reaches (some of) the vertices in the DFS tree rooted on a in G

SCC Algorithm Correctness

Theorem: **STRONGLY-CONNECTED-COMPONENTS**(G) correctly computes the strongly connected components of a directed graph G.



There is no path from c (root with a smaller finish time) to a (root with a larger finish time) in G

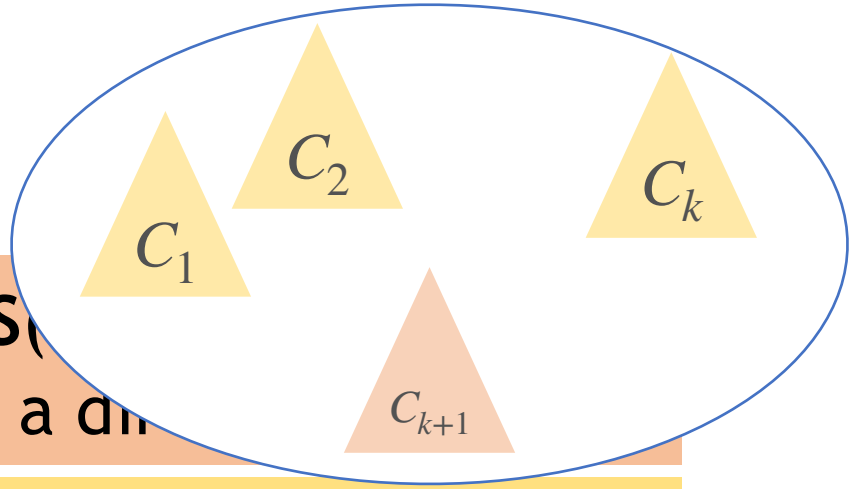


There is no path from a to c in G^T
$\Rightarrow$ DVS-VISIT(a) only reaches (some of) the vertices in the DFS tree rooted on a in G

SCC Correctness

Theorem: **STRONGLY-CONNECTED-COMPONENTS**(G) correctly computes the strongly connected components of a directed graph G.

SCC Correctness



Theorem: **STRONGLY-CONNECTED-COMPONENTS(**
computes the strongly connected components of a di

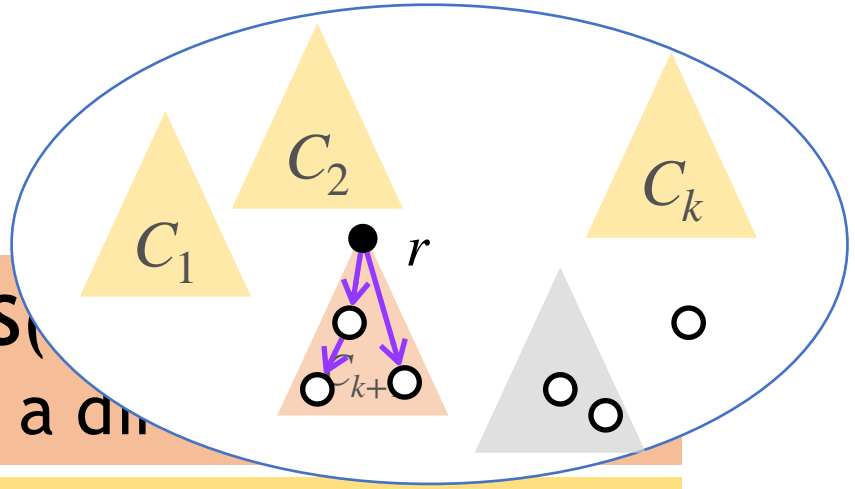
Claim: The first k trees produced by **DFS**(G^T) are SCCs

Prove by induction on k

Given the inductive hypothesis, show that the $(k+1)$-st tree, T_{k+1} is also a strongly connected component, C_{k+1}

- Every vertex $v \in C_{k+1}$ will be in T_{k+1}
- Every vertex $v \notin C_{k+1}$ will not be in T_{k+1}

SCC Correctness



Theorem: **STRONGLY-CONNECTED-COMPONENTS(**
computes the strongly connected components of a di

Claim: The first k trees produced by **DFS**(G^T) are SCCs

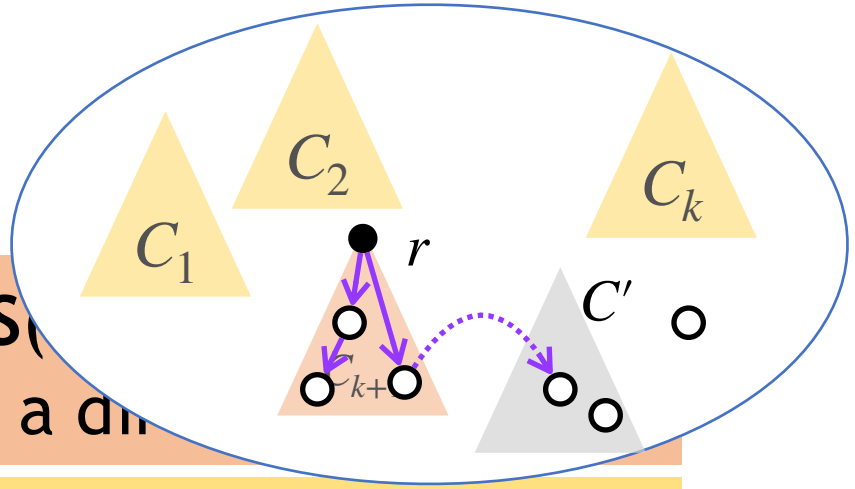
Prove by induction on k

Given the inductive hypothesis, show that the $(k+1)$-st tree, T_{k+1} is also a strongly connected component, C_{k+1}

- Every vertex $v \in C_{k+1}$ will be in T_{k+1}
- Every vertex $v \notin C_{k+1}$ will not be in T_{k+1}

By White-Path theory, let r be the first vertex in C_{k+1} visited by **DFS**(G^T), all vertices in C_{k+1} are descendants of r

SCC Correctness



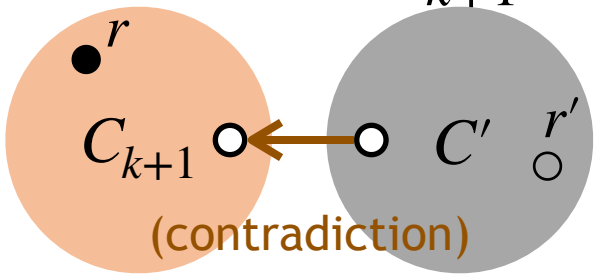
Theorem: **STRONGLY-CONNECTED-COMPONENTS(**
computes the strongly connected components of a di

Claim: The first k trees produced by **DFS**(G^T) are SCCs

Prove by induction on k

Given the inductive hypothesis, show that the $(k+1)$-st tree, T_{k+1} is also a strongly connected component, C_{k+1}

- Every vertex $v \in C_{k+1}$ will be in T_{k+1}
- Every vertex $v \notin C_{k+1}$ will not be in T_{k+1}

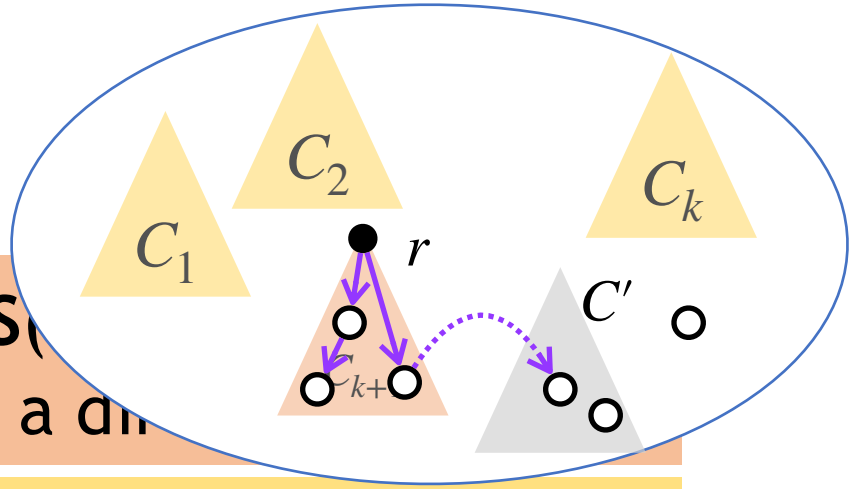


In G^T, there is no edge from C_{k+1} to SCC $C' \notin \{C_1, C_2, \cdots, C_{k+1}\}$

Otherwise, there is an edge from C' to C_{k+1} in G

- If C' is visited by DFS(G) first, there is $r' \in C'$ with $d[r'] < d[r] < f[r] < f[r']$

SCC Correctness



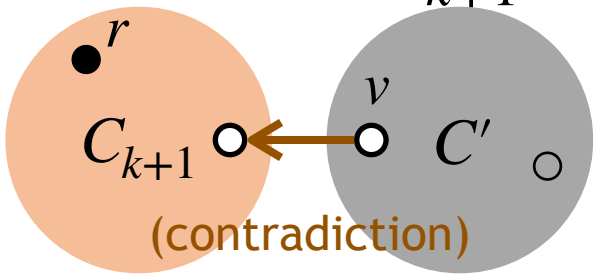
Theorem: **STRONGLY-CONNECTED-COMPONENTS(**
computes the strongly connected components of a di

Claim: The first k trees produced by **DFS**(G^T) are SCCs

Prove by induction on k

Given the inductive hypothesis, show that the $(k+1)$-st tree, T_{k+1} is also a strongly connected component, C_{k+1}

- Every vertex $v \in C_{k+1}$ will be in T_{k+1}
- Every vertex $v \notin C_{k+1}$ will not be in T_{k+1}



In G^T, there is no edge from C_{k+1} to SCC $C' \notin \{C_1, C_2, \cdots, C_{k+1}\}$

Otherwise, there is an edge from C' to C_{k+1} in G

- If C_{k+1} is visited by DFS(G) first, by the parenthesis property, $d[r] < f[r] < d[v] < f[v]$ for all $v \in C'$. (Contradiction)

Wrap Up

- Graph representation (exercise 4)
- DFS and its useful properties:
  - Parenthesis theorem
  - Classification of edges
  - White-path theorem (exercise 1)
- Applications of DFS:
  - Topological sort of DAGs (exercise 2)
  - Finding strongly connected components (exercise 3)