

# طراحی الگوریتم ها

جلسه ۹ و ۱۰ و ۱۱

ملکی مجد

# مباحث

- گراف - نحوه نمایش و خصوصیات
- الگوریتم BFS
- الگوریتم DFS
- الگوریتم Topological Sort
- الگوریتم برای پیدا کردن Strongly Connected Components (SCC)

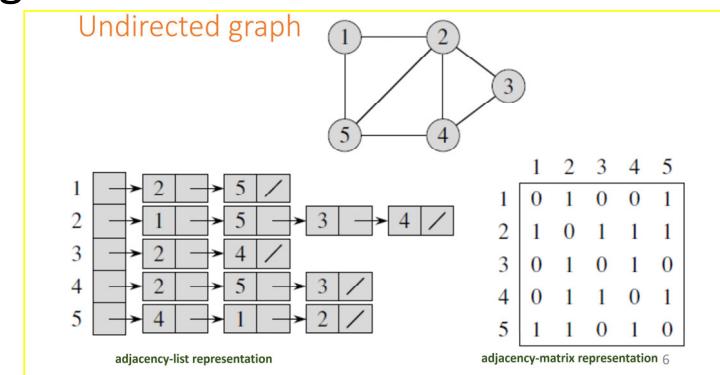
مبحث گراف از فصل ۲۲ کتاب CLRS تدریس می شود.

# Running time of a graph algorithm

- the size of the **input** of the graph
  - the number of vertices  $|V|$
  - and the number of edges  $|E|$
- Inside asymptotic notation (such as  $O$ -notation or  $\Theta$ -notation)
  - the symbol  $V$  denotes  $|V|$  and the symbol  $E$  denotes  $|E|$
- the pseudocode views vertex and edge sets as attributes of a graph
  - denote the vertex set of a graph  $G$  by  $V[G]$  and its edge set by  $E[G]$

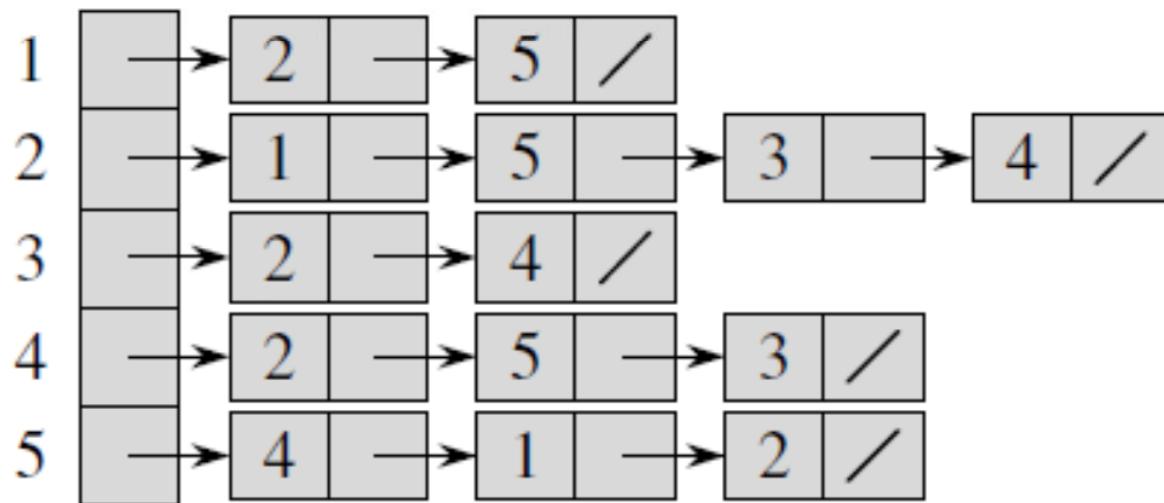
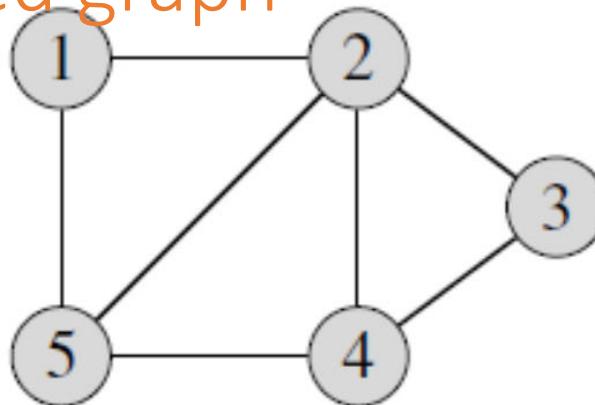
# Two representations of graphs

- as **adjacency lists**
  - provides a compact way to represent **sparse graph** ( $|E|$  is much less than  $|V|^2$ )
  - Assumed by most of the presented graph algorithms in this course
- as **adjacency matrices**
  - the graph is **dense** ( $|E|$  is close to  $|V|^2$ )
  - or when we need to be able to **tell quickly if there is an edge connecting two given vertices**
  - Rather than using one word of computer memory for each matrix entry, the adjacency matrix uses only **one bit per entry**.



Either way is applicable to both directed and undirected graphs

## Example of Undirected graph



adjacency-list representation

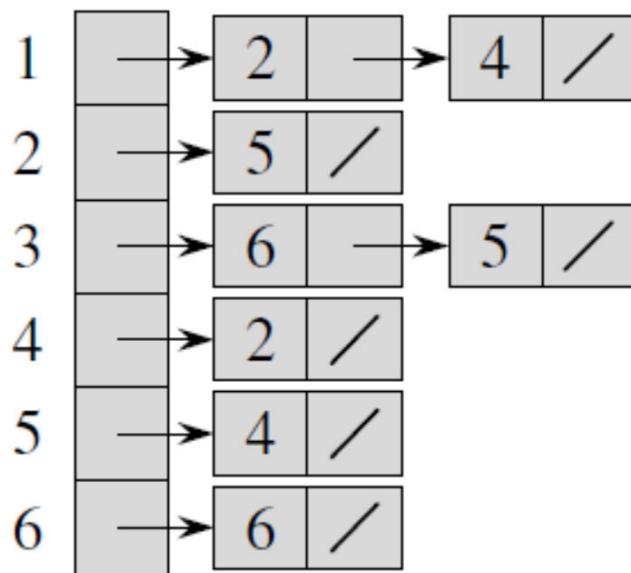
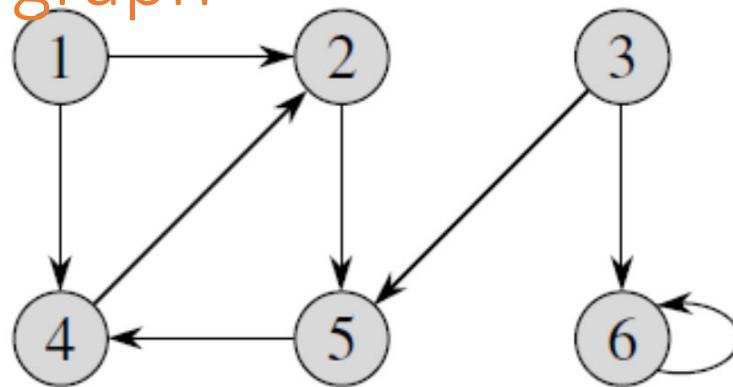
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

adjacency-matrix representation 5

# Example of undirected graph

- Two representations of an undirected graph in next slide.
  - An undirected graph  $G$  having five vertices and seven edges.
  - The **adjacency-list** representation of  $G$ .
    - consists of an array  $\text{Adj}$  of  $|V|$  lists, one for each vertex in  $V$ .
    - For each  $u \in V$ , the adjacency list  $\text{Adj}[u]$  contains all the vertices  $v$  such that there is an edge  $(u, v) \in E$ .
    - The vertices in each adjacency list are typically stored in an arbitrary order
  - The **adjacency-matrix** representation of  $G$ .

## Example of Directed graph



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

# Example of directed graph

- Two representations of a directed graph.
  - A directed graph  $G$  having six vertices and eight edges.
  - An adjacency-list representation of  $G$ .
  - The adjacency-matrix representation of  $G$ .

# Memory

- For both directed and undirected graphs, the **adjacency-list** representation has the desirable property that  
**amount of memory it requires is  $\Theta(V + E)$ .**
- sum of the lengths of all the **adjacency lists**
  - directed graph :  $|E|$
  - undirected graph :  $2|E|$
- amount of memory **adjacency-matrix** representation requires
  - $\Theta(V^2)$
  - independent of the number of edges in the graph

## *Transpose of a matrix A*

- the *transpose* of a matrix  $A = (a_{ij})$  is the matrix  $A^T = (a_{ij}^T)$   
given by  $a_{ij}^T = a_{ji}$
- adjacency matrix  $A$  of an undirected graph is its own transpose

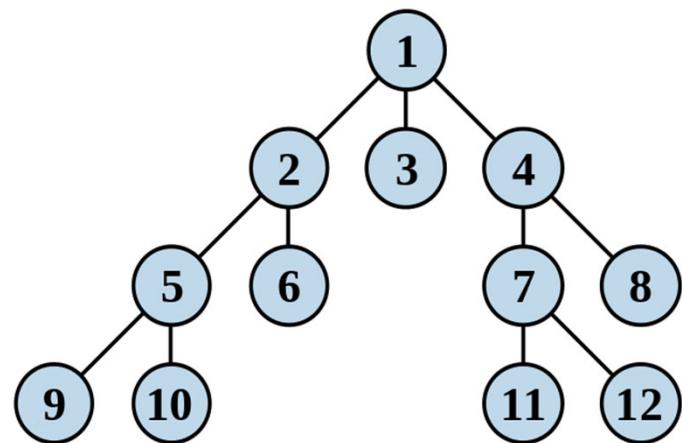
# Weighted graphs

- let  $G = (V, E)$  be a weighted graph with weight function  $w : E \rightarrow R$ .
- The weight  $w(u, v)$  of the edge  $(u, v) \in E$  is simply **stored with vertex  $v$**  in  $u$ 's adjacency list
- the weight  $w(u, v)$  of the edge  $(u, v) \in E$  is simply **stored as the entry in row  $u$  and column  $v$**  of the adjacency matrix

- زمان مورد نیاز برای پیدا کردن گره universal sink چقدر است (گراف با ماتریس مجاورت نمایش داده شده)؟
  - (گره ای با درجه خروجی  $|V| - 1$  و درجه ورودی صفر)

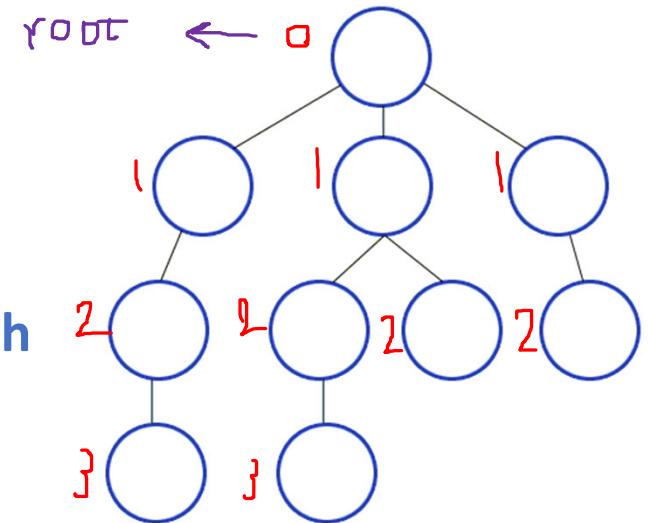
- زمان مورد نیاز برای پیدا کردن گره universal sink چقدر است (گراف با ماتریس مجاورت نمایش داده شده)؟
  - (گره ای با درجه خروجی  $|V| - 1$  و درجه ورودی صفر)
  - در زمان  $O(V)$  می شود؟

# BFS

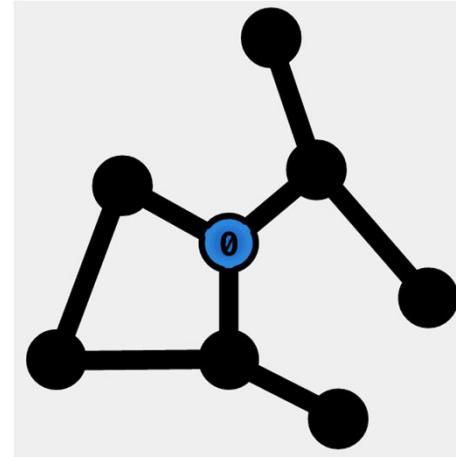


# Breadth-first search

- one of the simplest algorithms **for searching a graph**
  - archetype for many important graph algorithms
    - Prim
    - Dijkstra,
- **breadth-first search** systematically explores the edges of  $G$  to “discover” every vertex that is **reachable from  $s$**
- It computes the **distance** (smallest number of edges) from  $s$  to each reachable vertex
- produces a “**breadth-first tree**” with **root  $s$**  that contains **all reachable vertices** for any vertex  $v$  reachable from  $s$ , the path in the breadth-first tree from  $s$  to  $v$  corresponds to a “**shortest path**” from  $s$  to  $v$  in  $G$

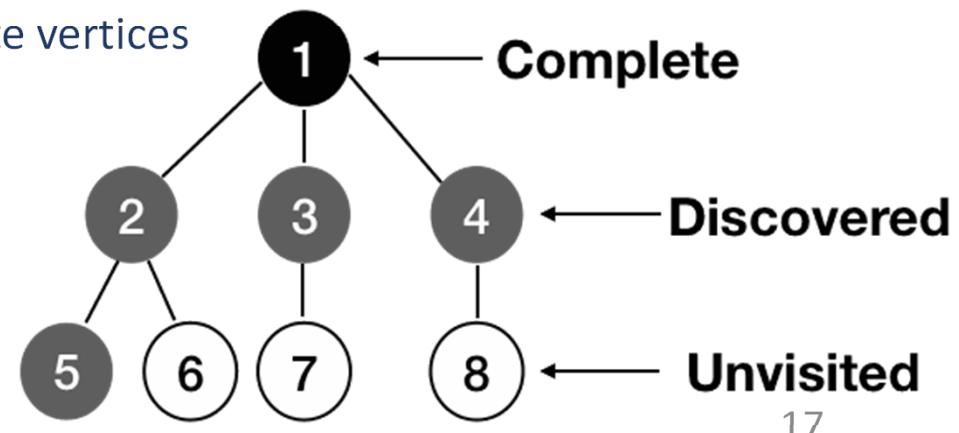


# Breadth (پھنا)



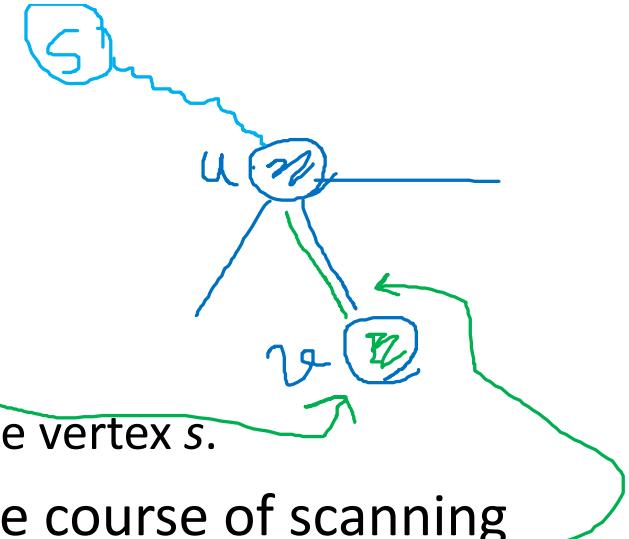
the algorithm **discovers all vertices at distance  $k$  from  $s$  before discovering any vertices at distance  $k + 1$**

- To keep track of progress, breadth-first search colors each vertex **white, gray, or black**.
- All vertices start out **white** and may later become **gray** and then **black**.
- A vertex is ***discovered*** the first time it is encountered during the search
  - at which time it becomes **nonwhite**
  - all vertices adjacent to black vertices have been discovered
  - Gray vertices may have some adjacent white vertices



# Breadth-first search

- constructs a breadth-first tree,
  - initially containing only its root, which is the source vertex  $s$ .
- Whenever a **white vertex  $v$**  is discovered in the course of scanning the adjacency list of an already discovered vertex  $u$ , the vertex  $v$  and the edge  $(u, v)$  are added to the tree.
- $u$  is the ***predecessor*** or ***parent*** of  $v$  in the breadth-first tree.
  - Since a vertex is **discovered at most once**, it has at most one parent.
  - Ancestor and descendant relationships in the breadth-first tree are defined **relative to the root  $s$**



- The breadth-first-search procedure BFS uses **adjacency lists**
- The **color** of each vertex  $u \in V$  is stored in the variable  $\text{color}[u]$
- The **predecessor** of  $u$  is stored in the variable  $\pi[u]$ .
  - If  $u$  has no predecessor (root and not discovered), then  $\pi[u] = \text{NIL}$
- The **distance** from the source  $s$  to vertex  $u$  computed by the algorithm is stored in  $d[u]$ .
- The algorithm also uses a **first-in, first-out queue**  $Q$  to manage the set of gray vertices

$\text{BFS}(G, s)$

```
1  for each vertex  $u \in V[G] - \{s\}$ 
2      do  $\text{color}[u] \leftarrow \text{WHITE}$ 
3           $d[u] \leftarrow \infty$ 
4           $\pi[u] \leftarrow \text{NIL}$ 
5       $\text{color}[s] \leftarrow \text{GRAY}$ 
6       $d[s] \leftarrow 0$ 
7       $\pi[s] \leftarrow \text{NIL}$ 
8       $Q \leftarrow \emptyset$ 
9      ENQUEUE( $Q, s$ )
10     while  $Q \neq \emptyset$ 
11         do  $u \leftarrow \text{DEQUEUE}(Q)$ 
12             for each  $v \in \text{Adj}[u]$ 
13                 do if  $\text{color}[v] = \text{WHITE}$ 
14                     then  $\text{color}[v] \leftarrow \text{GRAY}$ 
15                      $d[v] \leftarrow d[u] + 1$ 
16                      $\pi[v] \leftarrow u$ 
17                     ENQUEUE( $Q, v$ )
18              $\text{color}[u] \leftarrow \text{BLACK}$ 
```

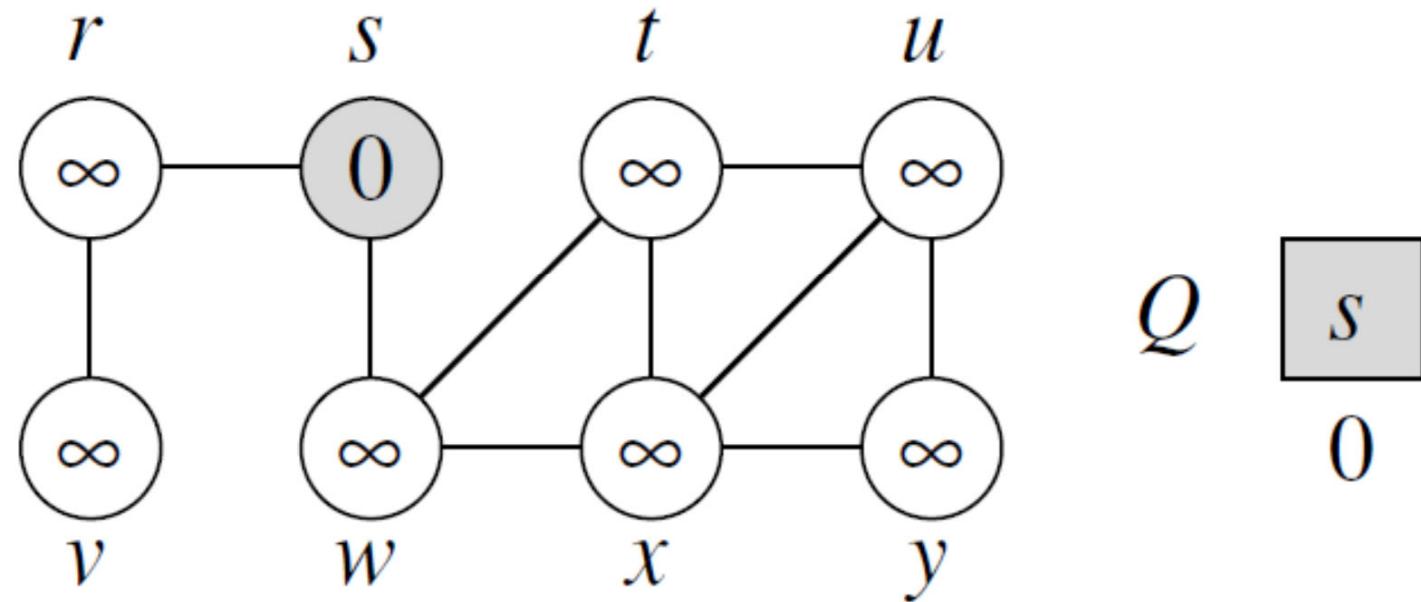
BFS( $G, s$ )

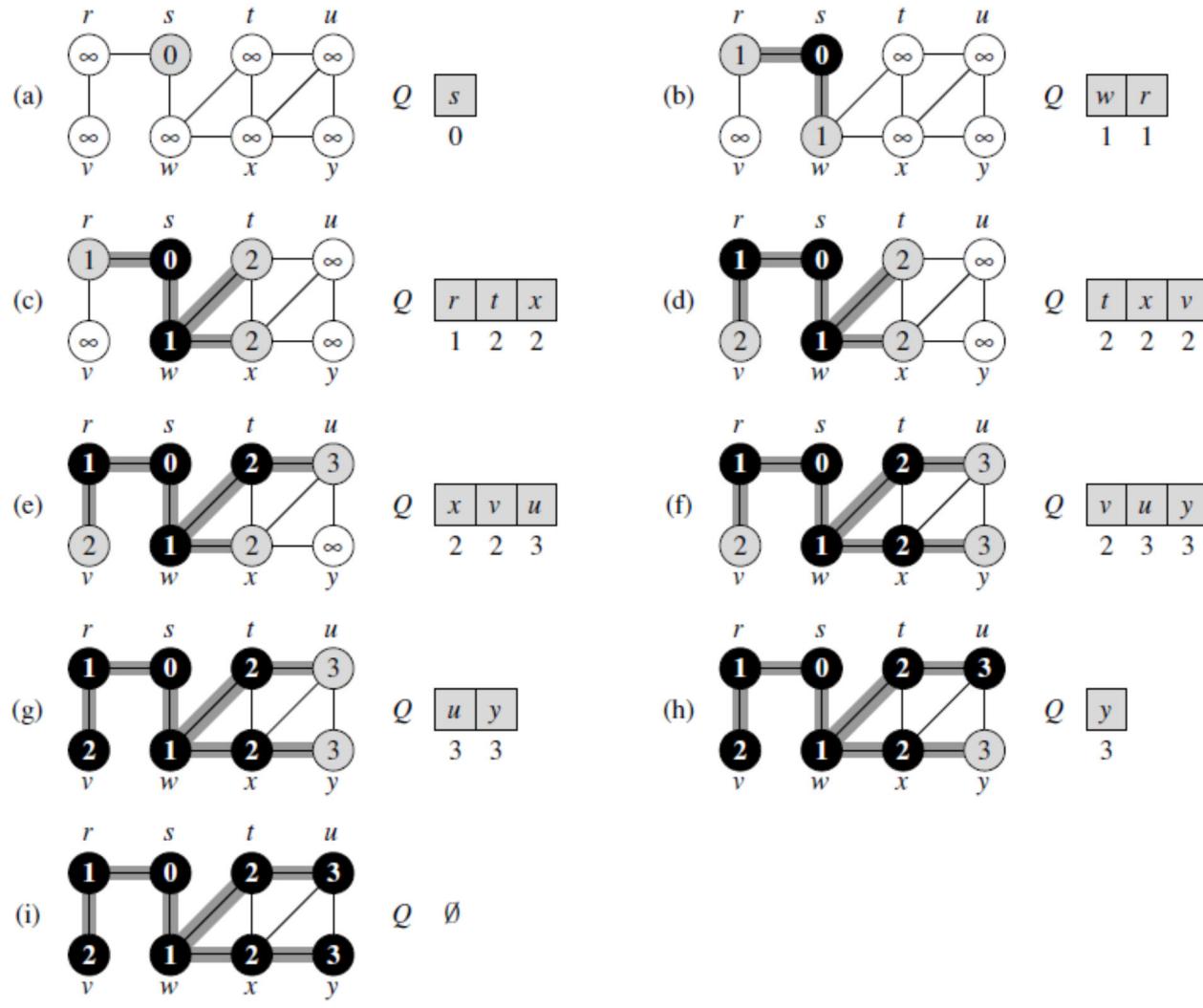
```
1  for each vertex  $u \in V[G] - \{s\}$ 
2    do  $color[u] \leftarrow \text{WHITE}$ 
3     $d[u] \leftarrow \infty$ 
4     $\pi[u] \leftarrow \text{NIL}$ 
5   $color[s] \leftarrow \text{GRAY}$ 
6   $d[s] \leftarrow 0$ 
7   $\pi[s] \leftarrow \text{NIL}$ 
8   $Q \leftarrow \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11   do  $u \leftarrow \text{DEQUEUE}(Q)$ 
12     for each  $v \in Adj[u]$ 
13       do if  $color[v] = \text{WHITE}$ 
14         then  $color[v] \leftarrow \text{GRAY}$ 
15          $d[v] \leftarrow d[u] + 1$ 
16          $\pi[v] \leftarrow u$ 
17         ENQUEUE( $Q, v$ )
18    $color[u] \leftarrow \text{BLACK}$ 
```

BFS( $G, s$ )

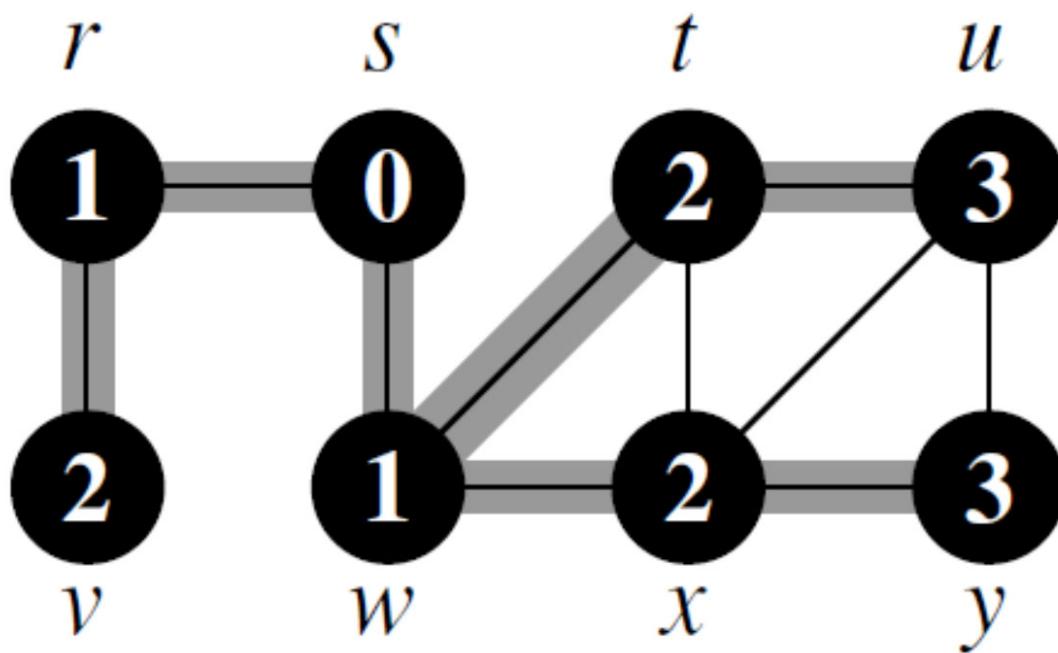
```
1  for each vertex  $u \in G.V - \{s\}$ 
2     $u.color = \text{WHITE}$ 
3     $u.d = \infty$ 
4     $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11    $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13       if  $v.color == \text{WHITE}$ 
14          $v.color = \text{GRAY}$ 
15          $v.d = u.d + 1$ 
16          $v.\pi = u$ 
17         ENQUEUE( $Q, v$ )
18    $u.color = \text{BLACK}$ 
```

Compute BFS tree?





Result:



- The results of **breadth-first search** may depend upon the **order** in which the neighbors of a given vertex are visited in line 12
- the breadth-first tree may vary, but the **distances**  $d$  computed by the algorithm **will not vary**

[www.cs.usfca.edu/~galles/visualization/BFS.html](http://www.cs.usfca.edu/~galles/visualization/BFS.html)

# Analyzing the running time of BFS

- The overhead for initialization is  $O(V)$
- Each vertex is enqueued at most once
  - and hence dequeued at most once
- The operations of enqueueing and dequeuing take  $O(1)$  time
  - the total time devoted to queue operations is  $O(V)$ .
- Each adjacency list is scanned at most once
  - sum of the lengths of all the adjacency lists is  $O(E)$ ,
- **The total running time of BFS is  $O(V+E)$** 
  - runs in time linear in the size of the adjacency-list representation

# Shortest paths

- we claimed that breadth-first search **finds the distance to each reachable vertex in a graph  $G = (V, E)$  from a given source vertex  $s \in V$ .**
- Define the **shortest-path distance**  $\delta(s, v)$  from  $s$  to  $v$  as the minimum number of edges in any path from vertex  $s$  to vertex  $v$ 
  - if there is no path from  $s$  to  $v$ , then  $\delta(s, v) = \infty$ . ( $\delta$  is pronounced delta)
- A path of length  $\delta(s, v)$  from  $s$  to  $v$  is said to be a **shortest path** from  $s$  to  $v$ .

# Breadth-first trees

- $V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$
- $E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$
- The predecessor subgraph  $G_\pi$  is a ***breadth-first tree*** if  $V_\pi$  consists of the vertices reachable from  $s$  and, for all  $v \in V_\pi$ , there is a unique simple path from  $s$  to  $v$  in  $G_\pi$  that is also a shortest path from  $s$  to  $v$  in  $G$ .

## Lemma 6

When applied to a directed or undirected graph  $G = (V, E)$ ,  
procedure BFS constructs  $\pi$  so that  
the predecessor subgraph  $G_\pi = (V_\pi, E_\pi)$  is a breadth-first tree.

$\text{PRINT-PATH}(G, s, v)$

- 1   **if**  $v = s$
- 2    **then** print  $s$
- 3    **else**   **if**  $\pi[v] = \text{NIL}$
- 4       **then** print “no path from”  $s$  “to”  $v$  “exists”
- 5       **else**  $\text{PRINT-PATH}(G, s, \pi[v])$
- 6           print  $v$

PRINT-PATH( $G, s, v$ )

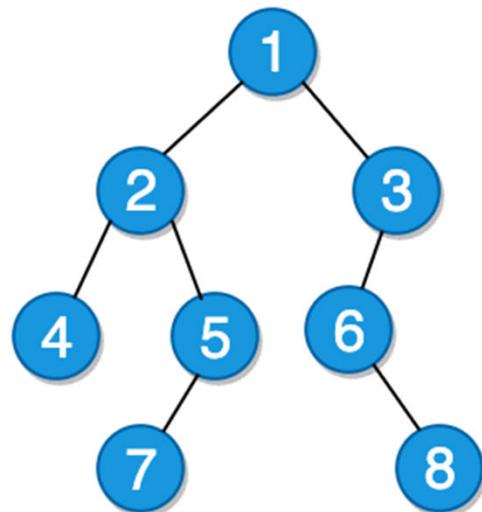
```
1  if  $v == s$ 
2    then print  $s$ 
3    else if  $\pi[v] == \text{NIL}$ 
4      then print "no path from"  $s$  "to"  $v$  "exists"
5      else PRINT-PATH( $G, s, \pi[v]$ )
6        print  $v$ 
```

PRINT-PATH( $G, s, v$ )

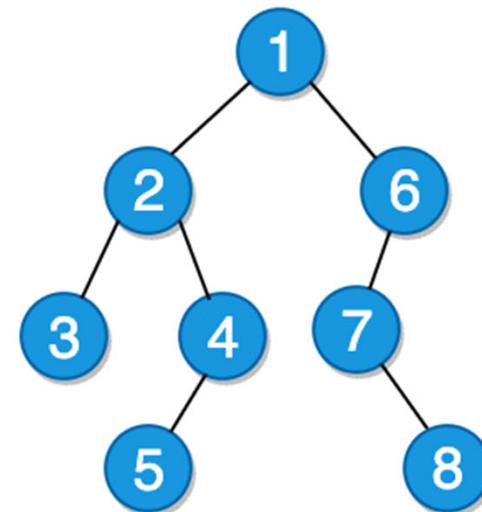
```
1  if  $v == s$ 
2    print  $s$ 
3  elseif  $v.\pi == \text{NIL}$ 
4    print "no path from"  $s$  "to"  $v$  "exists"
5  else PRINT-PATH( $G, s, v.\pi$ )
6    print  $v$ 
```

# DFS

BFS



DFS



# Depth-First Search **DFS**

- Strategy: to search **deeper** in the graph whenever possible
- Two applications of depth-first search
  - **Sorting** a directed acyclic graph (DAG)
  - Finding the **strongly connected components** of a directed graph

# Insight of DFS

In depth-first search,

- edges are explored out of the **most recently** discovered vertex  $v$  that still has unexplored edges leaving it.
- When all of  $v$ 's edges have been explored, the search **backtracks** to explore edges leaving the vertex from which  $v$  was discovered (**predecessor**)
- This process **continues** until we have discovered all the vertices that are reachable from the original source vertex.
- If any **undiscovered** vertices remain, then one of them is selected as a **new source** and the search is repeated from that source.

# From one vertex

DFS-VISIT( $u$ )

- 1  $color[u] \leftarrow \text{GRAY}$   $\triangleright$  White vertex  $u$  has just been discovered.
- 2  $time \leftarrow time + 1$
- 3  $d[u] \leftarrow time$
- 4 **for** each  $v \in Adj[u]$   $\triangleright$  Explore edge  $(u, v)$ .
- 5     **do if**  $color[v] = \text{WHITE}$
- 6         **then**  $\pi[v] \leftarrow u$
- 7             DFS-VISIT( $v$ )
- 8      $color[u] \leftarrow \text{BLACK}$   $\triangleright$  Blacken  $u$ ; it is finished.
- 9      $f[u] \leftarrow time \leftarrow time + 1$

# coloring technique

- Each vertex is **initially white**, is **grayed** when it is *discovered* in the search, and is **blackened** when it is *finished*
- This technique guarantees that each vertex ends up in exactly one depth-first tree

# timestamps

- Each vertex  $v$  has two timestamps:
  - the **first timestamp  $d[v]$**  records when  $v$  is first discovered (and grayed),
  - and the second **timestamp  $f[v]$**  records when the search finishes examining  $v$ 's adjacency list (and blackens  $v$ )
- 
- These timestamps are integers between 1 and  $2|V|$

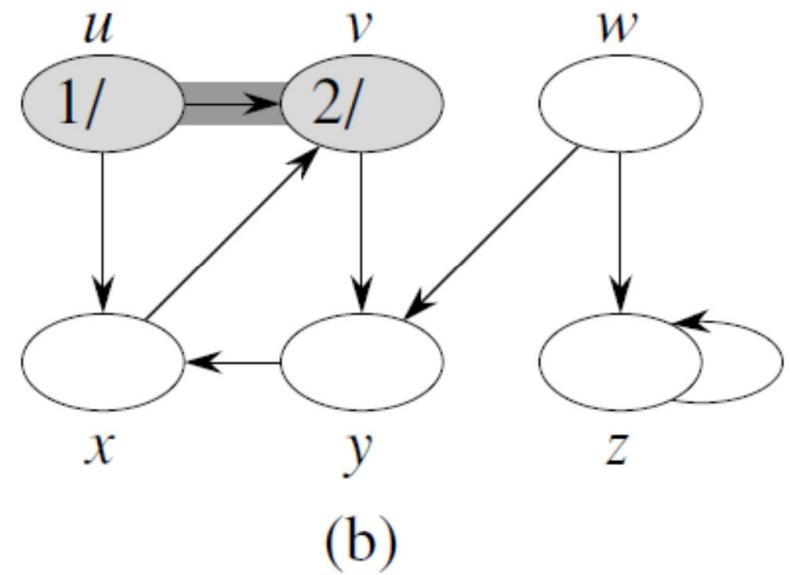
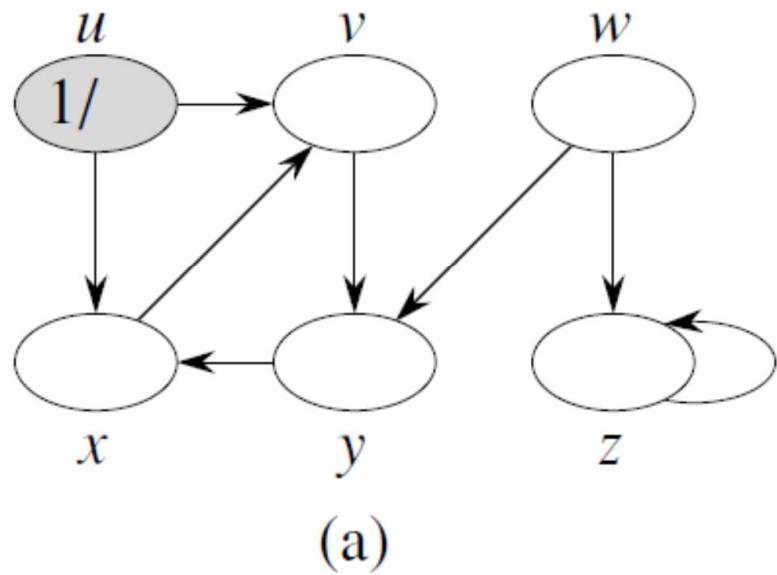
- the predecessor subgraph produced by a depth-first search may be composed of **several trees**
  - search may be repeated from multiple sources

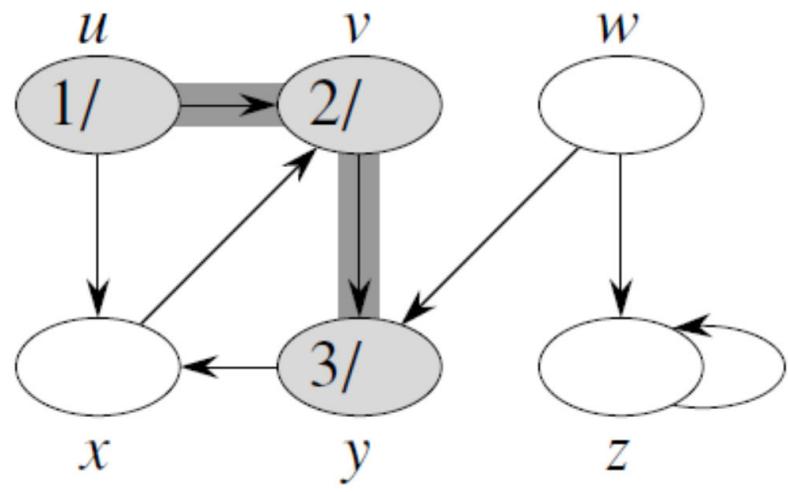
# Whole graph

$\text{DFS}(G)$

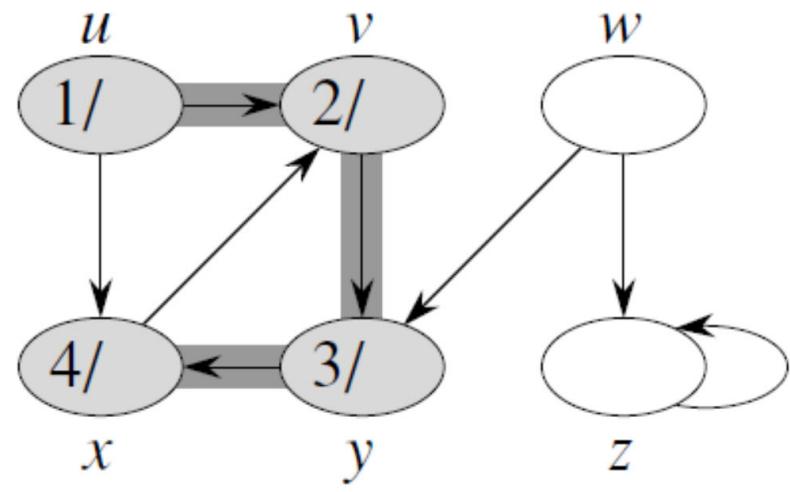
- 1   **for** each vertex  $u \in V[G]$
- 2           **do**  $\text{color}[u] \leftarrow \text{WHITE}$
- 3                $\pi[u] \leftarrow \text{NIL}$
- 4    $time \leftarrow 0$
- 5   **for** each vertex  $u \in V[G]$
- 6           **do if**  $\text{color}[u] = \text{WHITE}$
- 7               **then**  $\text{DFS-VISIT}(u)$

# Example

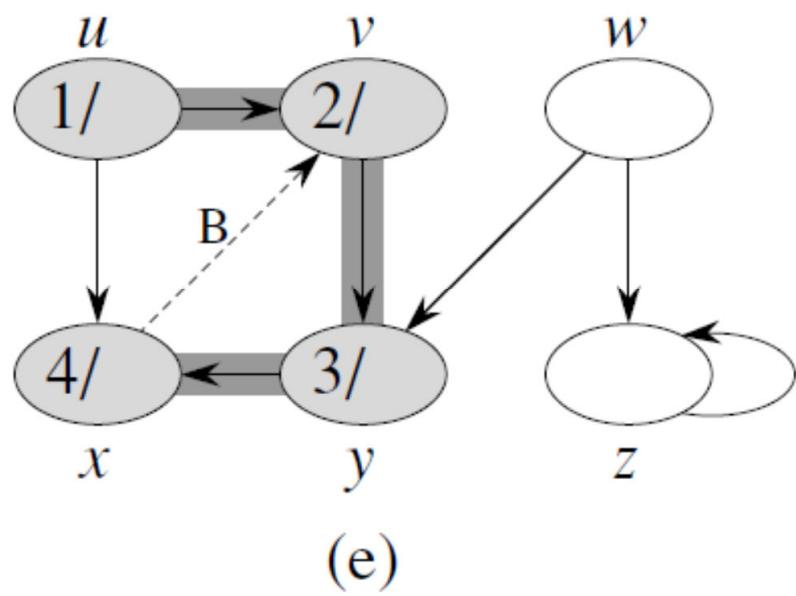




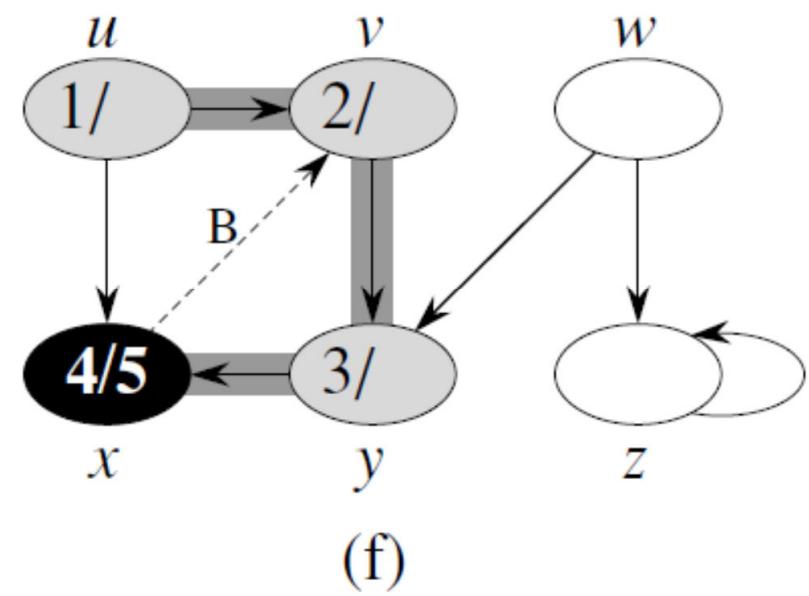
(c)



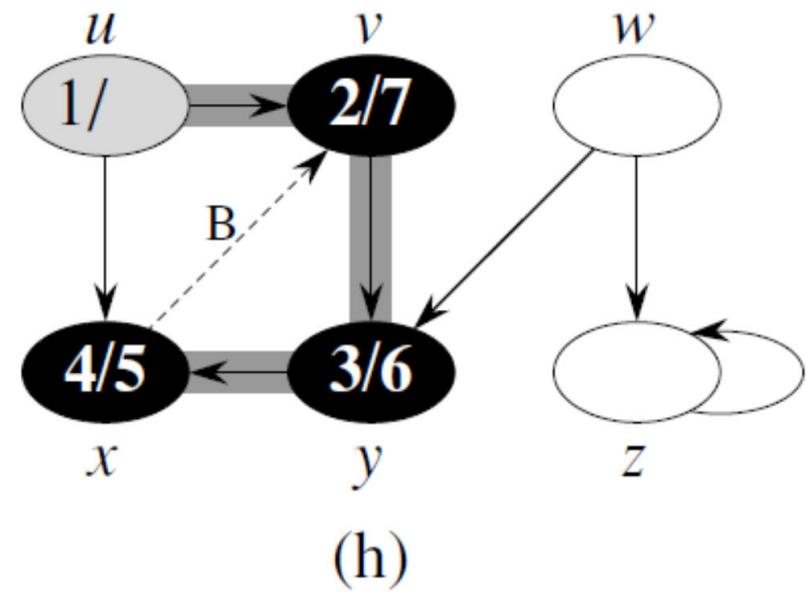
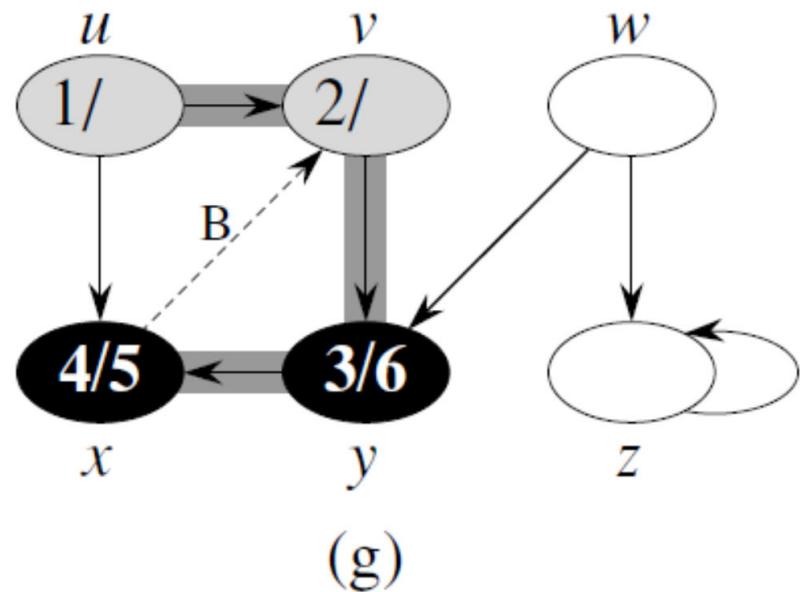
(d)

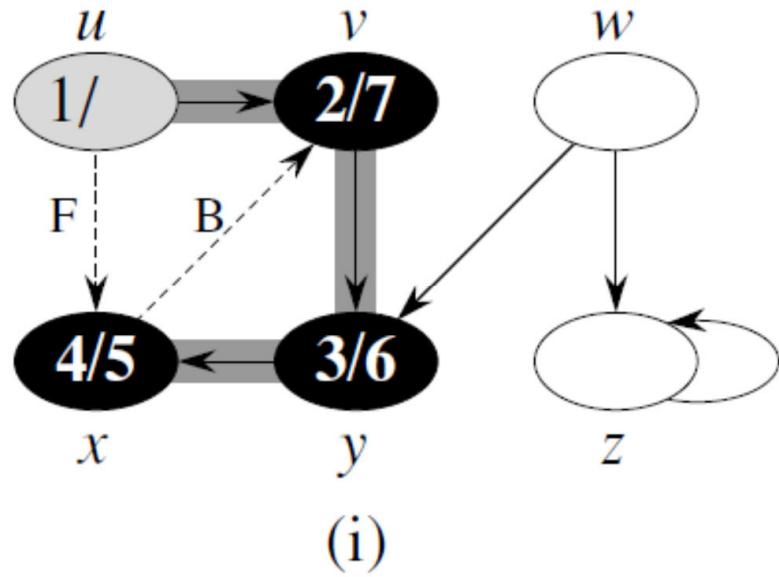


(e)

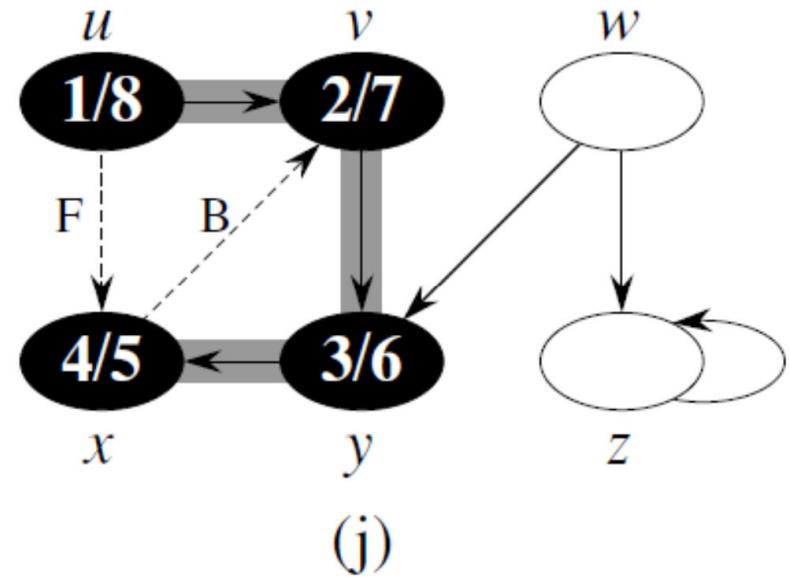


(f)

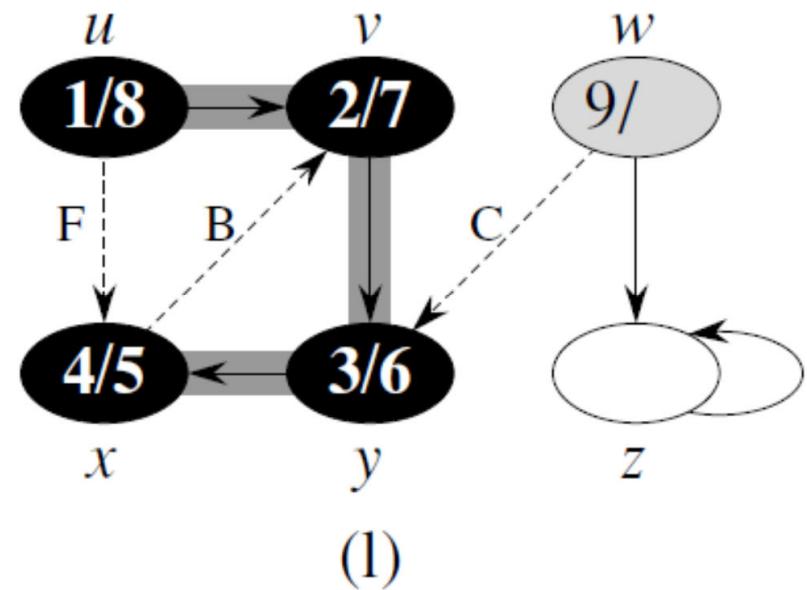
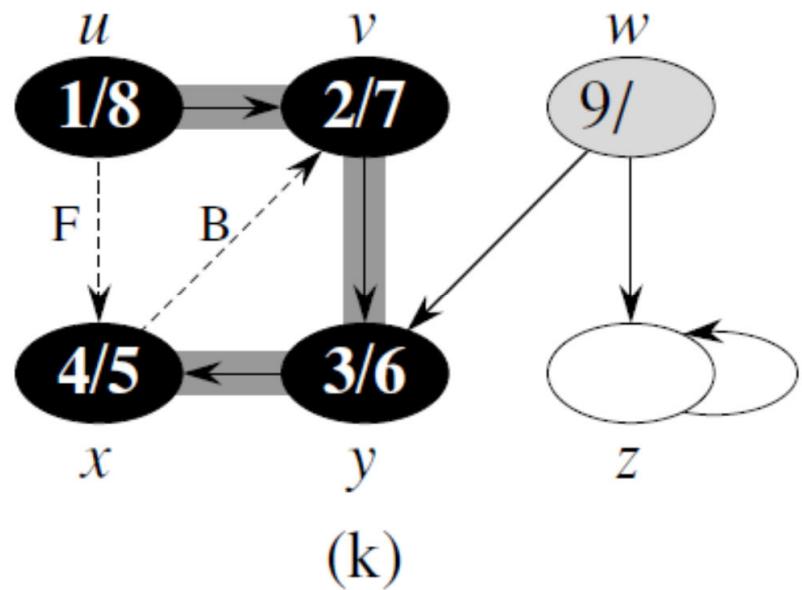


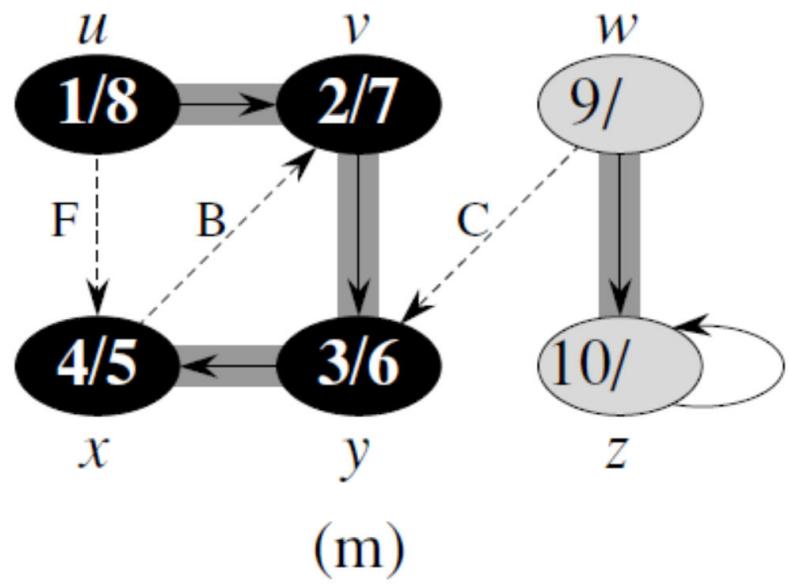


(i)

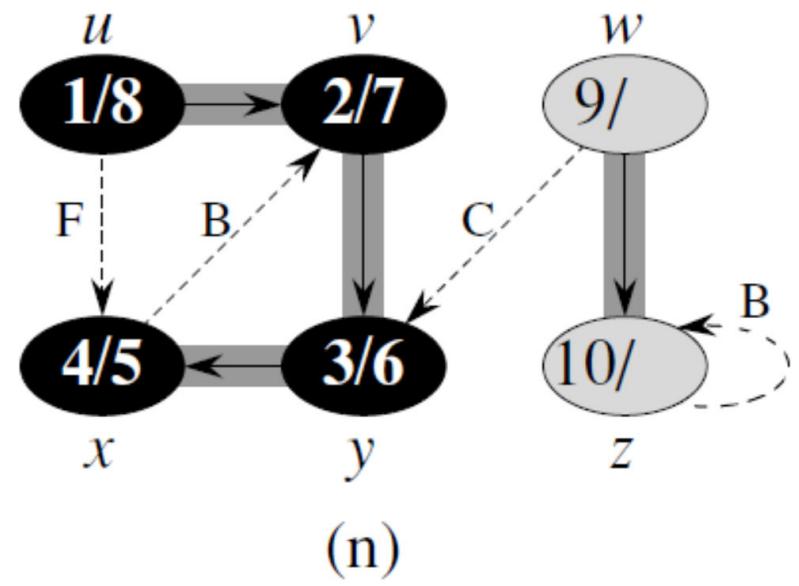


(j)

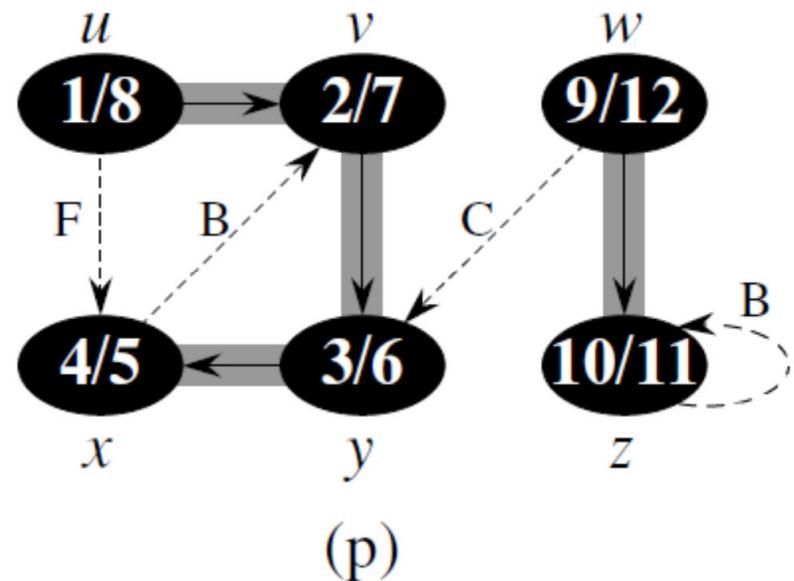
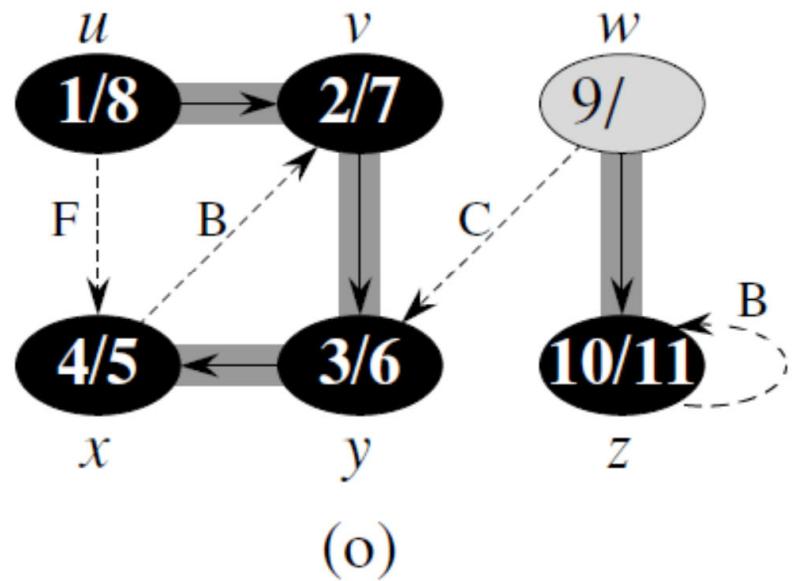




(m)



(n)



# Running time of DFS $O(V+E)$

DFS-VISIT( $u$ )

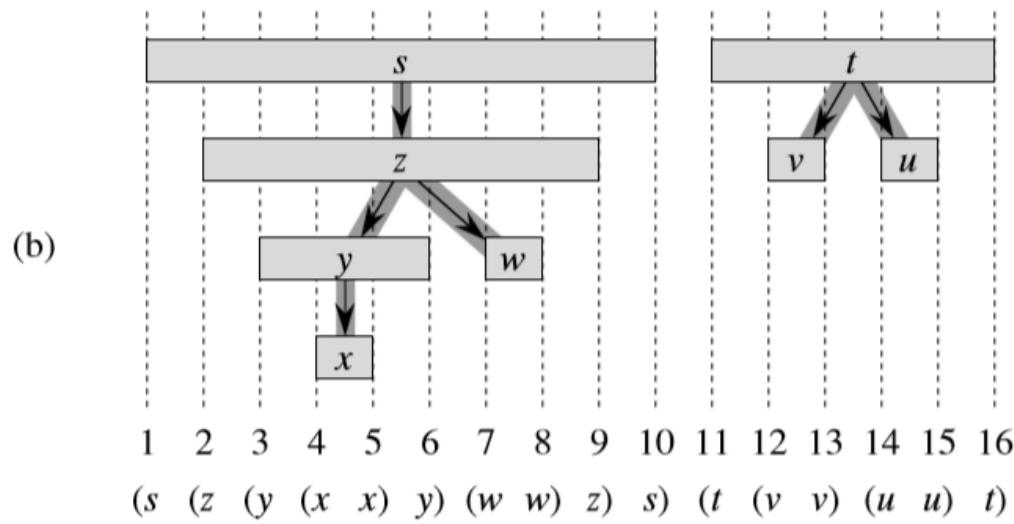
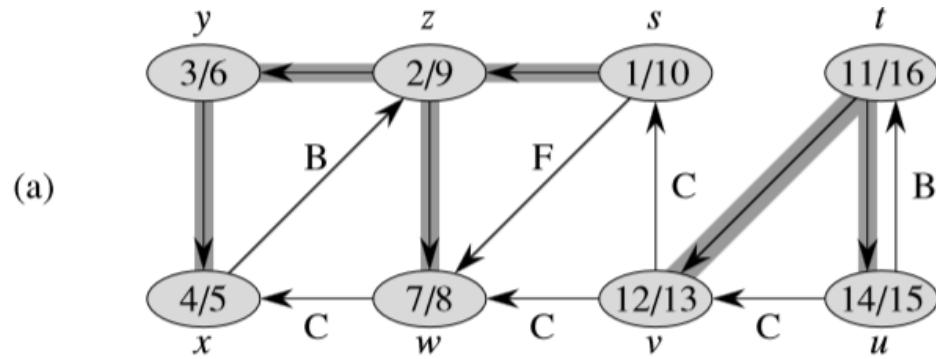
```
1  $color[u] \leftarrow \text{GRAY}$      $\triangleright$  White vertex  $u$  has just been discovered.  
2  $time \leftarrow time + 1$   
3  $d[u] \leftarrow time$   
4 for each  $v \in Adj[u]$        $\triangleright$  Explore edge  $(u, v)$ .  
5   do if  $color[v] = \text{WHITE}$   
6     then  $\pi[v] \leftarrow u$   
7     DFS-VISIT( $v$ )  
8    $color[u] \leftarrow \text{BLACK}$        $\triangleright$  Blacken  $u$ ; it is finished.  
9    $f[u] \leftarrow time \leftarrow time + 1$ 
```

DFS( $G$ )

```
1 for each vertex  $u \in V[G]$   
2   do  $color[u] \leftarrow \text{WHITE}$   
3      $\pi[u] \leftarrow \text{NIL}$   
4    $time \leftarrow 0$   
5 for each vertex  $u \in V[G]$   
6   do if  $color[u] = \text{WHITE}$   
7     then DFS-VISIT( $u$ )
```

# Properties of depth-first search

- $u = \pi[v]$  if and only if **DFS-VISIT(v)** was called during a search of u's adjacency list
- vertex v is a descendant of vertex u in the depth-first forest if and only if v is discovered during the time in which u is gray
- If we represent the discovery of vertex u with a left parenthesis "(u" and represent its finishing by a right parenthesis "u)", then the **history of discoveries and finishings** makes a **well-formed expression** in the sense that the parentheses are properly nested



## Theorem 22.7 (Parenthesis theorem)

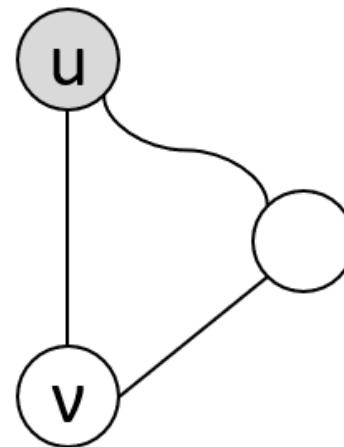
- In any depth-first search of a (directed or undirected) graph  $G = (V, E)$ , for **any two vertices  $u$  and  $v$ , exactly one of the following** three conditions holds
  - I. the intervals  $[d[u], f[u]]$  and  $[d[v], f[v]]$  are **entirely disjoint**, and neither  $u$  nor  $v$  is a descendant of the other in the depth-first forest,
  - II. the interval  $[d[u], f[u]]$  is **contained entirely within the interval**  $[d[v], f[v]]$ , and  $u$  is a descendant of  $v$  in a depth-first tree, or
  - III. the interval  $[d[v], f[v]]$  is **contained entirely within the interval**  $[d[u], f[u]]$ , and  $v$  is a descendant of  $u$  in a depth-first tree.

# Nesting of descendants' intervals

- Vertex  $v$  is a proper descendant(نسل) of vertex  $u$  in the depth-first forest for a (directed or undirected) graph  $G$  if and only if  $d[u] < d[v] < f[v] < f[u]$ .

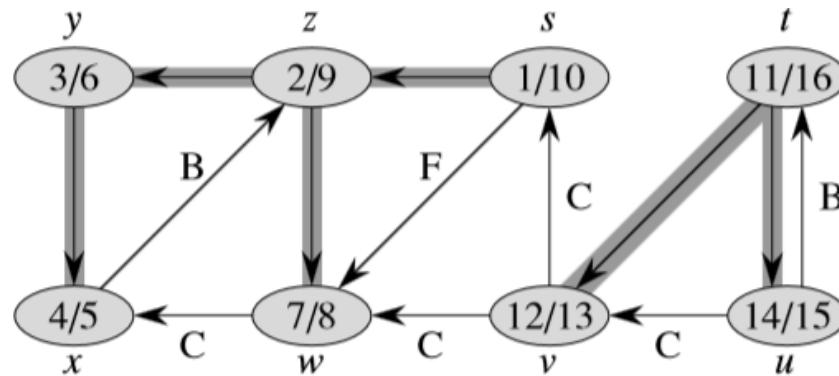
# White-path theorem

- In a depth-first forest of a (directed or undirected) graph  $G = (V, E)$ , vertex **v is a descendant of vertex u if and only if at the time  $d[u]$  that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.**

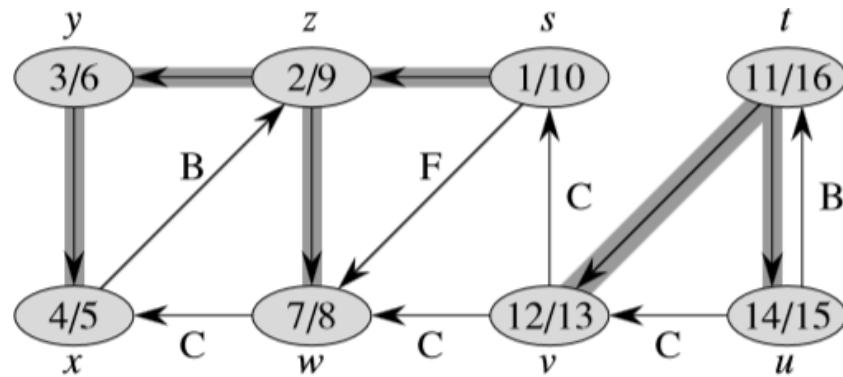


# Classification of edges

- **Tree edges** are edges in the depth-first forest  $G\pi$ .
  - Edge  $(u,v)$  is a tree edge if  $v$  was first discovered by exploring edge  $(u,v)$ .
- **Back edges** are those edges  $(u,v)$  connecting a vertex  $u$  to an ancestor  $v$  in a depth-first tree.
  - Self-loops, which may occur in directed graphs, are considered to be back edges



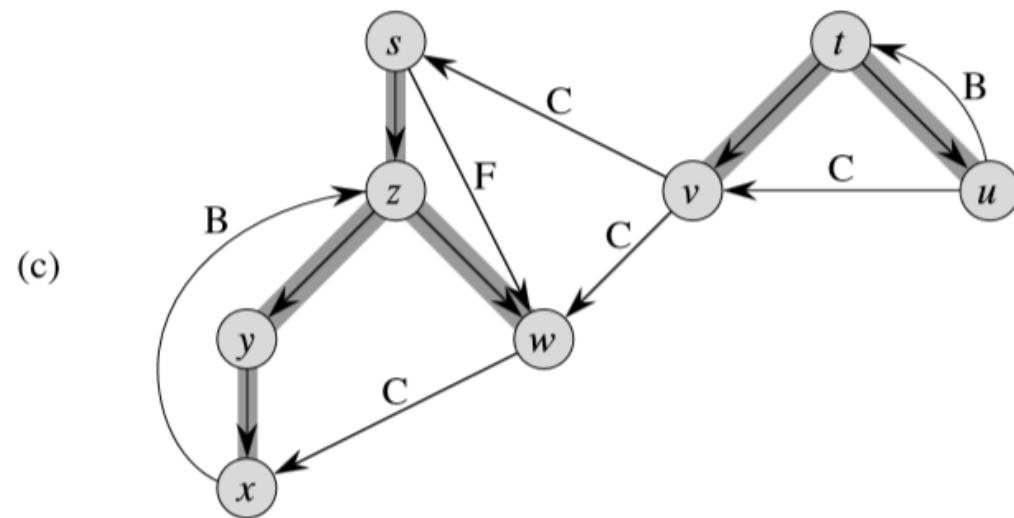
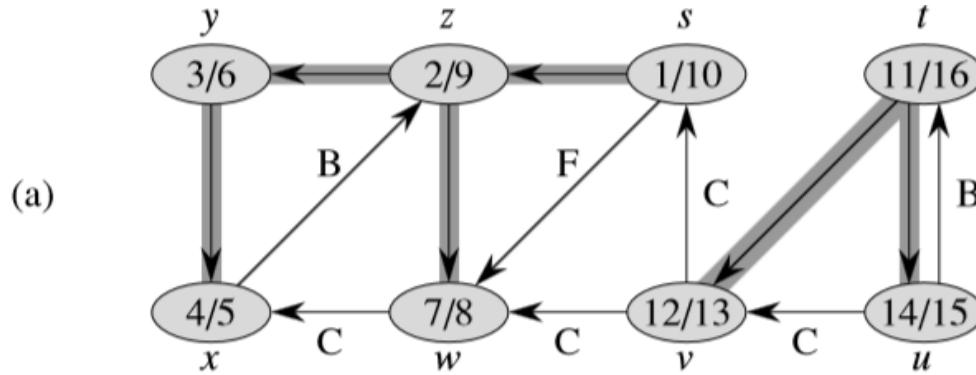
# Classification of edges



- **Forward edges** are those nontree edges  $(u,v)$  connecting a vertex  $u$  to a descendant  $v$  in a depth-first tree
- **Cross edges** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees

# DAG

a **directed graph is acyclic** if and only if a depth-first search yields **no “back” edges**



- The DFS algorithm can be modified to classify edges as it encounters them
- The key idea is that each edge  $(u,v)$  can be classified by the color of the vertex  $v$  that is reached when the edge is first explored (except that forward and cross edges are not distinguished):
  - 1) **WHITE** indicates a tree edge
  - 2) **GRAY** indicates a back edge, and
  - 3) **BLACK** indicates a forward or cross edge.
    - it can be shown that such an edge  $(u,v)$  is a **forward edge** if  $d[u] < d[v]$  and a **cross edge** if  $d[u] > d[v]$ .

# Theorem

- In a depth-first search of an **undirected** graph  $G$ , every edge of  $G$  is either **a tree edge or a back edge**.

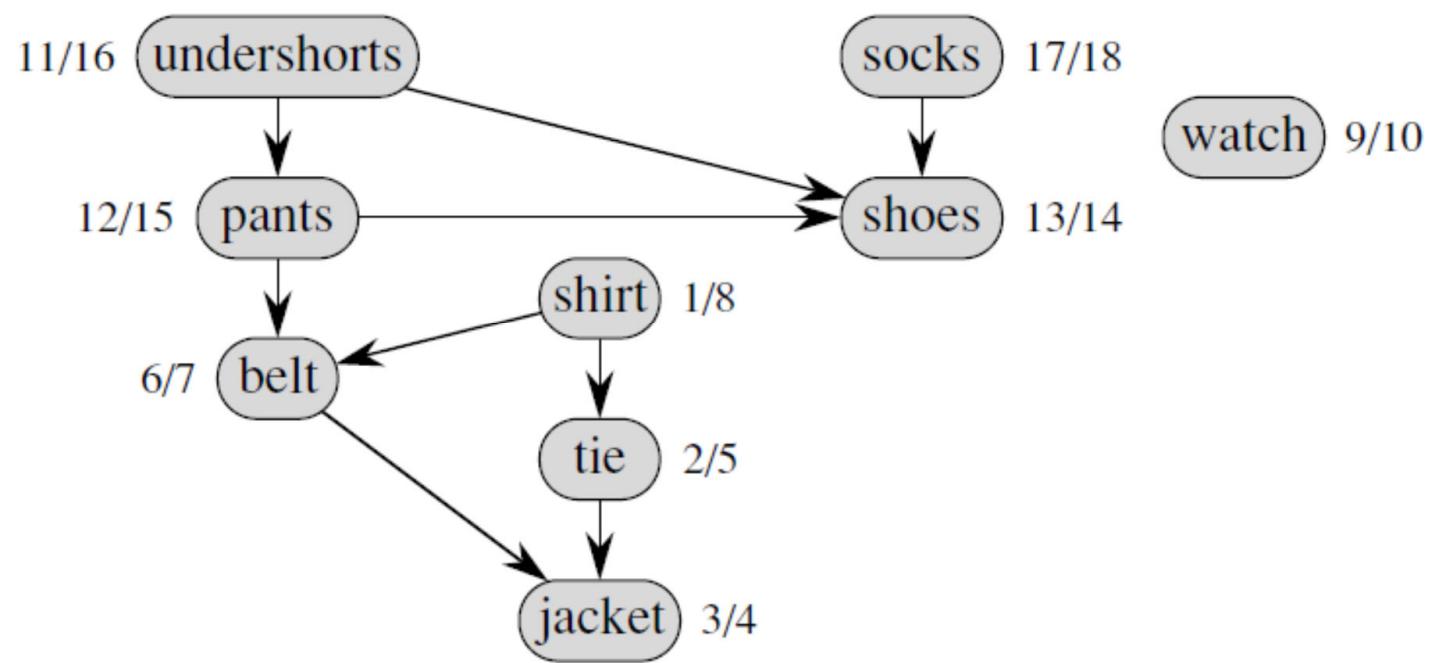
# Topological sort

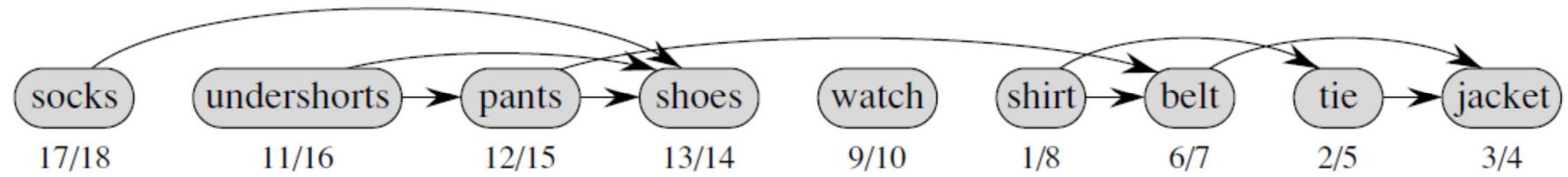
## DAG : Directed Acyclic Graph

- Directed acyclic graphs are used in many applications to indicate precedence's among events

# Topological sort

- A ***topological sort*** of a dag  $G = (V, E)$  is a linear ordering of all its vertices such that if  $G$  contains an edge  $(u, v)$ , then  **$u$  appears before  $v$**  in the ordering.
  - ordering of its vertices along a horizontal line so that all directed edges go from left to right
- If the graph is not acyclic, then no linear ordering is possible.





## TOPOLOGICAL-SORT( $G$ )

- 1 call  $\text{DFS}(G)$  to compute finishing times  $f[v]$  for each vertex  $v$
- 2 as each vertex is finished, insert it onto the front of a linked list
- 3 **return** the linked list of vertices

- We can perform a topological sort in time  $\Theta(V + E)$ , since depth-first search takes  $\Theta(V + E)$  time and it takes  $O(1)$  time to insert each of the  $|V|$  vertices onto the front of the linked list

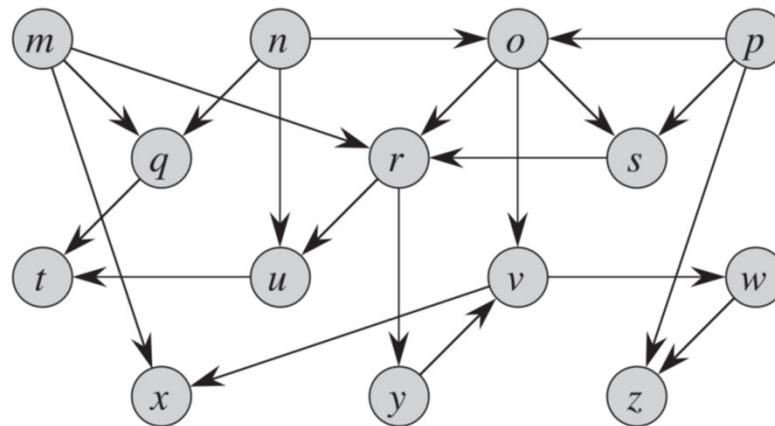
## Lemma 22.11

- A directed graph  $G$  is acyclic if and only if a depth-first search of  $G$  yields **no back edges**

## Theorem 22.12

- $\text{TOPOLOGICAL-SORT}(G)$  produces a topological sort of a directed acyclic graph  $G$ .

Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the below DAG :



**SCC**  
**strongly connected components**

# Decomposing a directed graph

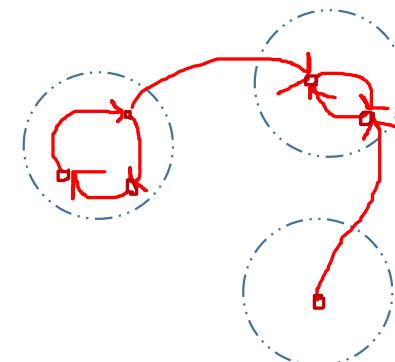
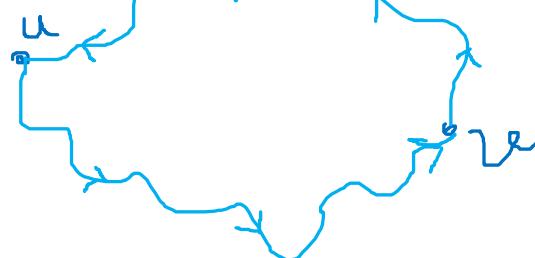
- a classic **application of depth-first search**:
  - decomposing a directed graph into its **strongly connected components**.
- Do this decomposition using **two depth-first searches (DFS)**

Many algorithms that work with directed graphs *begin with such a decomposition*. After decomposition, the algorithm is run separately on each strongly connected component. The solutions are then combined according to the structure of connections between components

# Strongly connected components

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  $C$ , we have **both  $u \rightsquigarrow v$  and  $v \rightsquigarrow u$** ; that is, **vertices  $u$  and  $v$  are reachable from each other**

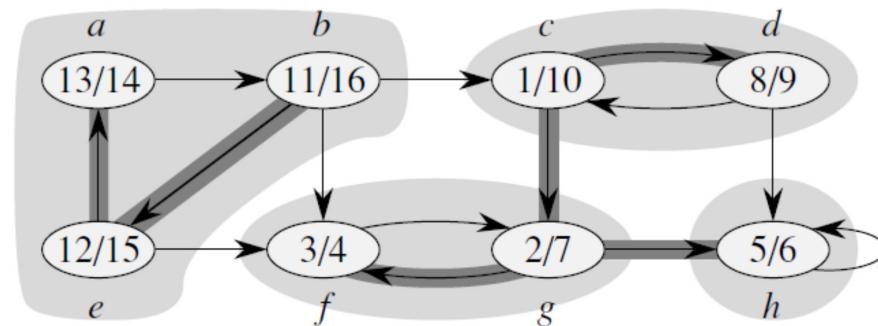


# idea

- The algorithm uses the transpose of  $G$ ,
- $G^T = (V, E^T)$ , where  $E^T = \{(u, v) : (v, u) \in E\}$ .
- The time to create  $G^T$  is  $O(V + E)$ .
- $G$  and  $G^T$  have the **same** strongly connected components

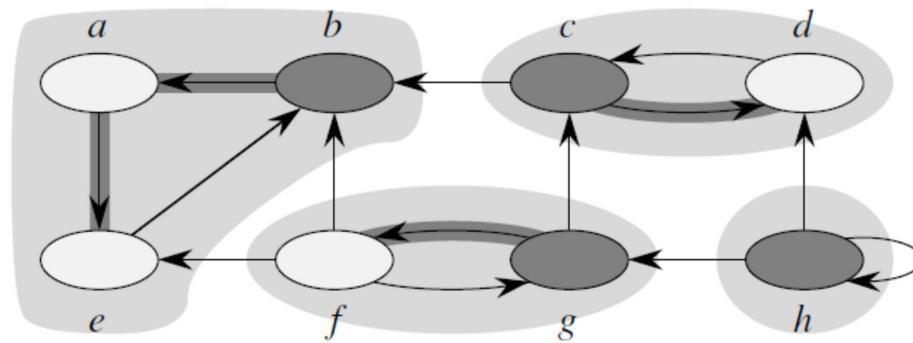
## STRONGLY-CONNECTED-COMPONENTS( $G$ )

- 1) call  $\text{DFS}(G)$  to compute finishing times  $f[u]$  for each vertex  $u$
- 2) compute  $G^T$
- 3) call  $\text{DFS}(G^T)$ , but in the main loop of  $\text{DFS}$ , consider the vertices in order of decreasing  $f[u]$  (as computed in line 1)
- 4) output the vertices of each tree in the depth-first forest formed in line 3 as a separate SCC (strongly connected component)



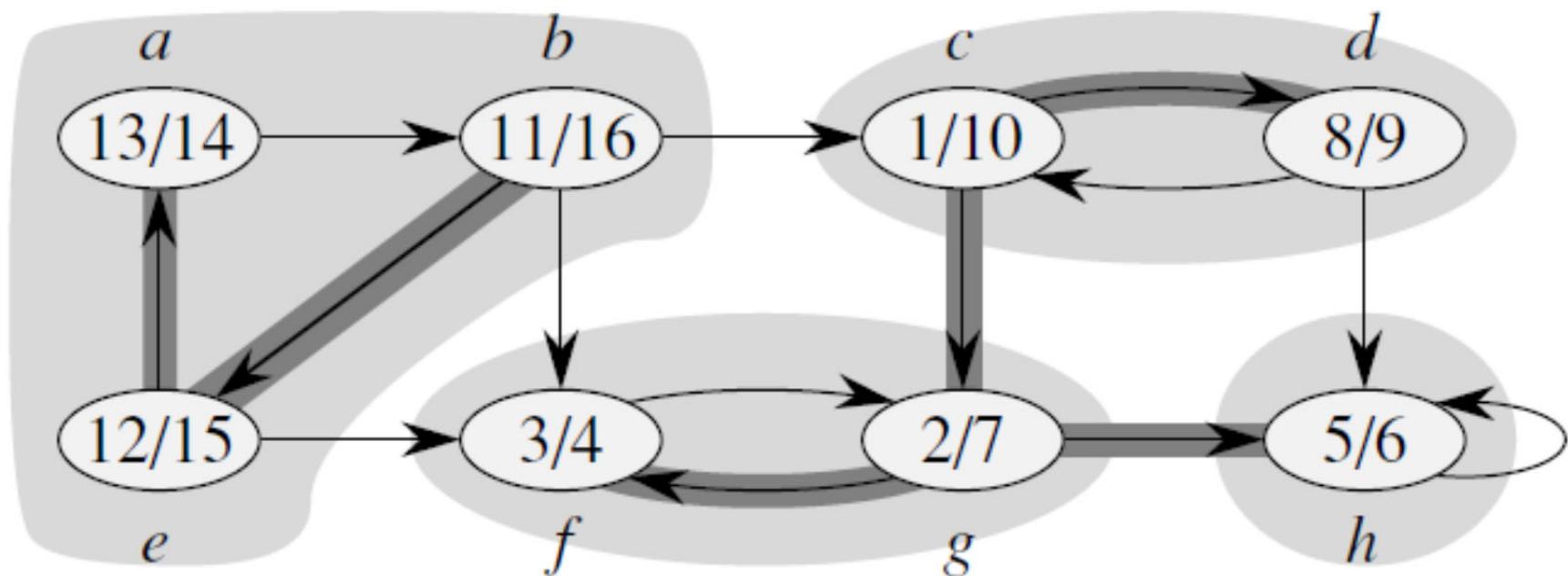
## STRONGLY-CONNECTED-COMPONENTS( $G$ )

- 1) call  $DFS(G)$  to compute finishing times  $f[u]$  for each vertex  $u$
- 2) compute  $G^T$
- 3) call  $DFS(G^T)$ , but in the main loop of  $DFS$ , consider the vertices in order of decreasing  $f[u]$  (as computed in line 1)
- 4) output the vertices of each tree in the depth-first forest formed in line 3 as a separate SCC (strongly connected component)



# strongly connected components

+labeled with its discovery and finishing times

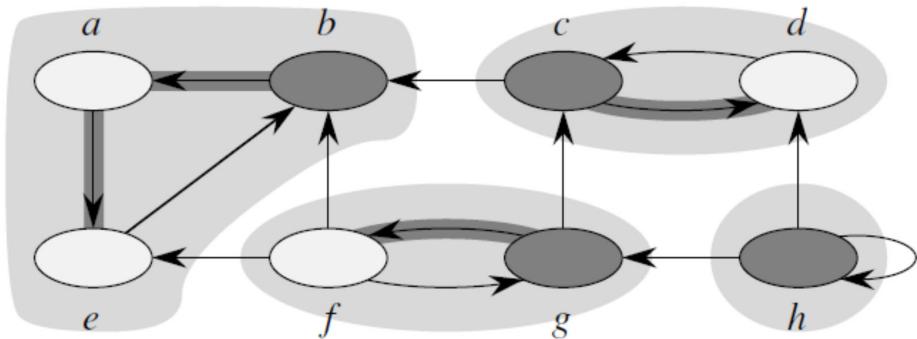


*component graph*  $G^{SCC} = (V^{SCC}, E^{SCC})$

- Suppose that  $G$  has strongly connected components  $C_1, C_2, \dots, C_k$
- The vertex set  $V^{SCC}$  is  $\{v_1, v_2, \dots, v_k\}$ , and it contains a vertex  $v_i$  for each strongly connected component  $C_i$  of  $G$
- There is an edge  $(v_i, v_j) \in E^{SCC}$  if  $G$  contains a directed edge  $(x, y)$  for some  $x \in C_i$  and some  $y \in C_j$ 
  - contracting all edges whose incident vertices are within the same strongly connected component of  $G$

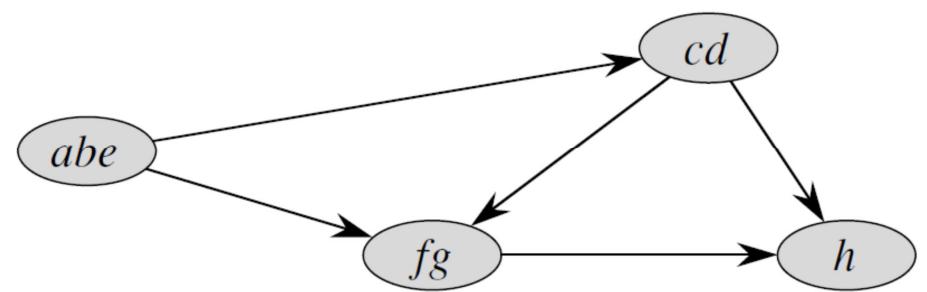
the component graph is a **dag**

## Transpose of graph G



## The acyclic component graph $G^{SCC}$

It is a dag



How can the number of strongly connected components of a graph change if a new edge is added?

A directed graph  $G = (V, E)$  is *semiconnected* :

if, for all pairs of vertices  $u, v \in V$ , we have  $u \rightsquigarrow v$  or  $v \rightsquigarrow u$ .

Give an efficient algorithm to determine whether  $G$  is semiconnected.  
Analyze its running time.

A directed graph  $G = (V, E)$  is *semiconnected* :

if, for all pairs of vertices  $u, v \in V$ , we have  $u \rightsquigarrow v$  or  $v \rightsquigarrow u$ .

Give an efficient algorithm to determine whether  $G$  is semiconnected.

Analyze its running time.

+SCC

+Sort