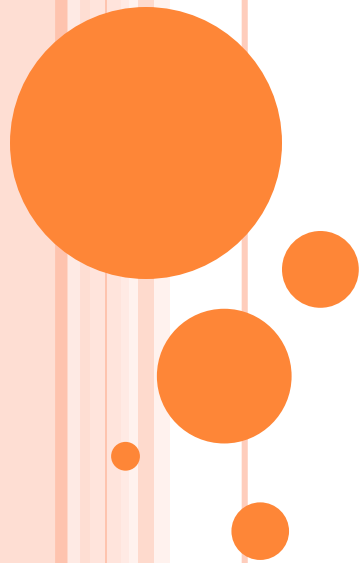
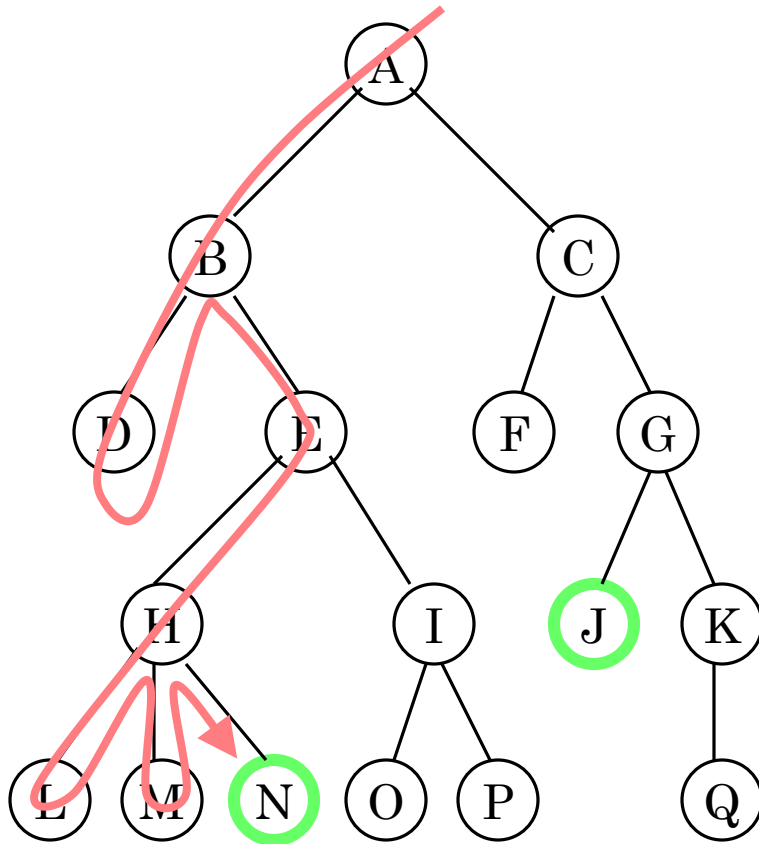


# GRAPH TRAVERSALS ALGORITHMS

**Design and Analysis of Algorithms**



# DEPTH-FIRST SEARCHING

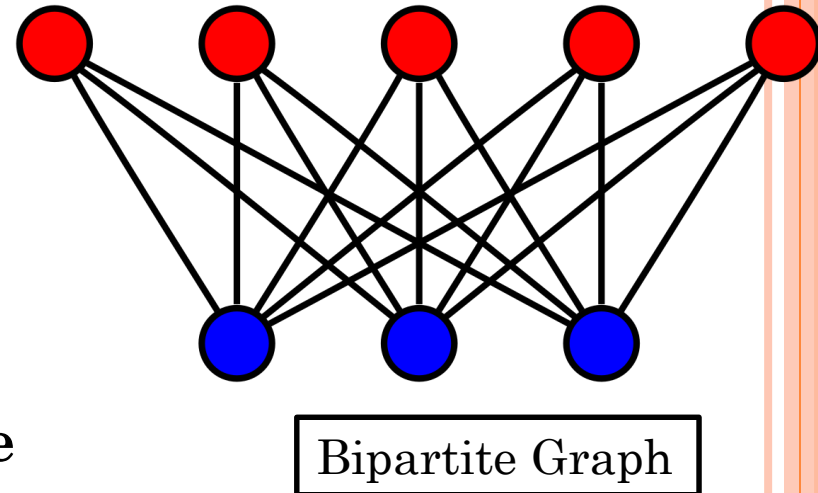


- A **depth-first** search (**DFS**) explores a path all the way to a leaf before **backtracking** and exploring another path
- For example, after searching **A**, then **B**, then **D**, the search backtracks and tries another path from **B**
- Node are explored in the order **A B D E H L M N I O P C F G J K Q**
- **N** will be found before **J**

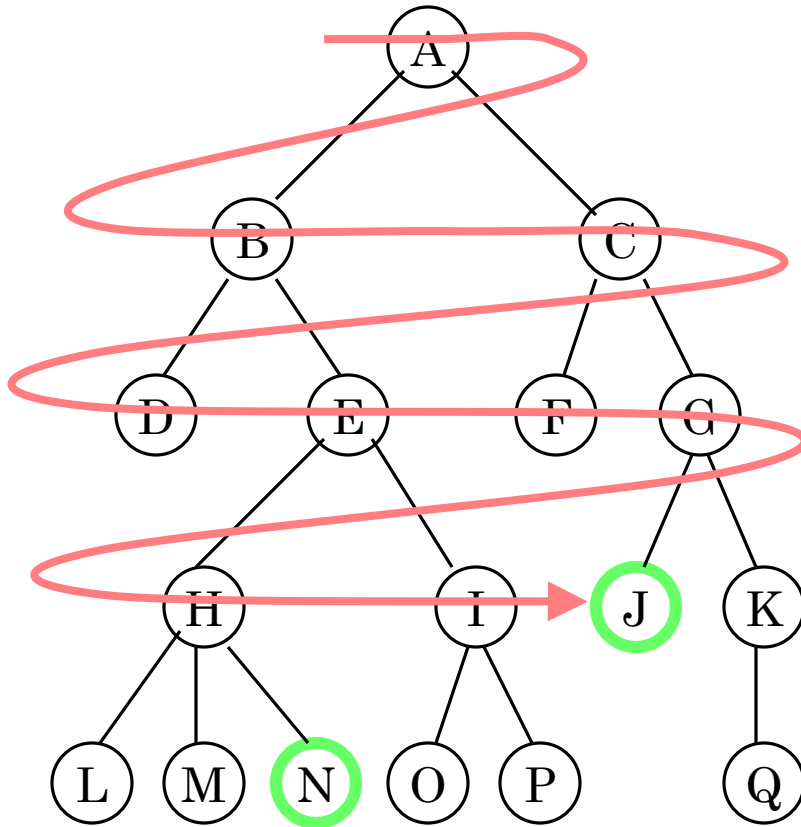
# DEPTH FIRST SEARCH

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph.

- Detecting cycle in a graph
- Path Finding
- Topological Sorting
- To test if a graph is bipartite
- Finding Strongly Connected Components of a graph
- Solving puzzles with only one solution



# BREADTH-FIRST SEARCHING



- A **breadth-first** search (**BFS**) explores nodes nearest the root before exploring nodes further away
- For example, after searching **A**, then **B**, then **C**, the search proceeds with **D**, **E**, **F**, **G**
- Node are explored in the order **A B C D E F G H I J K L M N O P Q**
- **J** will be found before **N**

# BREADTH-FIRST SEARCH

- **Breadth-first search** is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms.
- Given a graph  $G = (V, E)$  and a distinguished **source vertex  $s$ , 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.
- It also produces a “breadth-first tree” with root  $s$  that contains all reachable vertices.
- For any vertex reachable from  $s$ , the simple path in the breadth-first tree from  $s$  to  $v$  corresponds to a “shortest path” from  $s$  to  $v$  in  $G$ , that is, a path containing the smallest number of edges.
- The algorithm works on both directed and undirected graphs.

# BREADTH-FIRST SEARCHING

- **Shortest Path and Minimum Spanning Tree for unweighted graph**
- **Peer to Peer Networks**
- **Crawlers in Search Engines**
- **Social Networking Websites**
- **GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
- **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
- **In Garbage Collection**
- **Cycle detection in undirected graph:**

# BREADTH-FIRST SEARCHING

- To test if a graph is Bipartite
- Path Finding
- Finding all nodes within one connected component.

# *BREADTH-FIRST SEARCH*

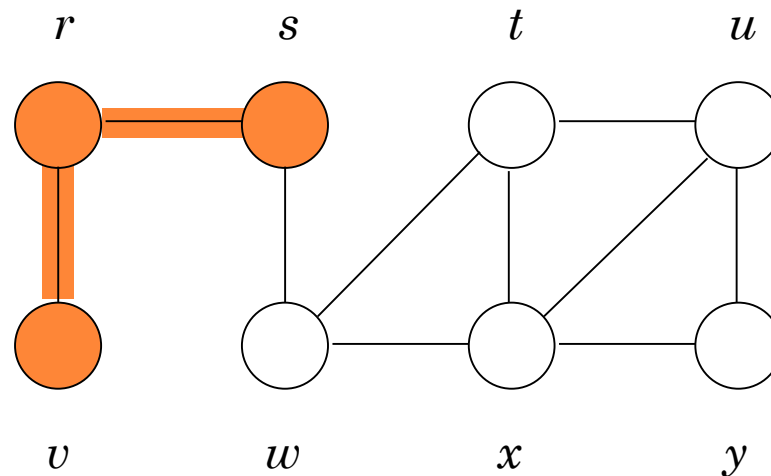
- Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
- That is, the algorithm **discovers** all vertices at distance **k from s before discovering** any vertices at distance **k + 1**.



# BREADTH-FIRST SEARCH

## Distance

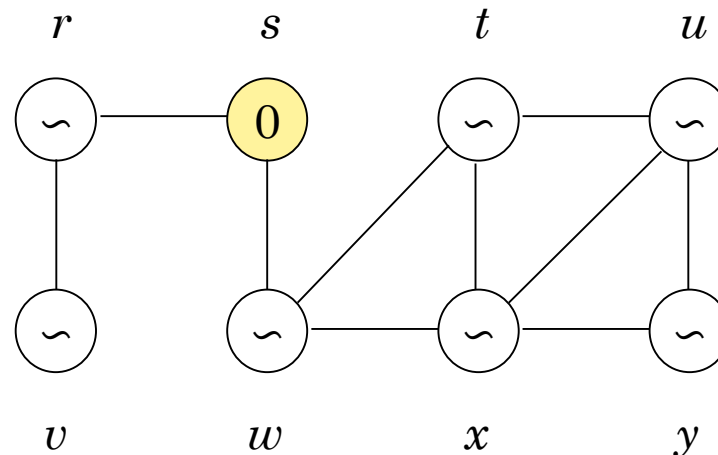
- Distance from  $u$  to  $v$ 
  - The number of edges in the shortest path from  $u$  to  $v$ .
  - The distance from  **$s$  to  $v$  is 2**.



# BREADTH-FIRST SEARCH

## ○ Breadth-first search

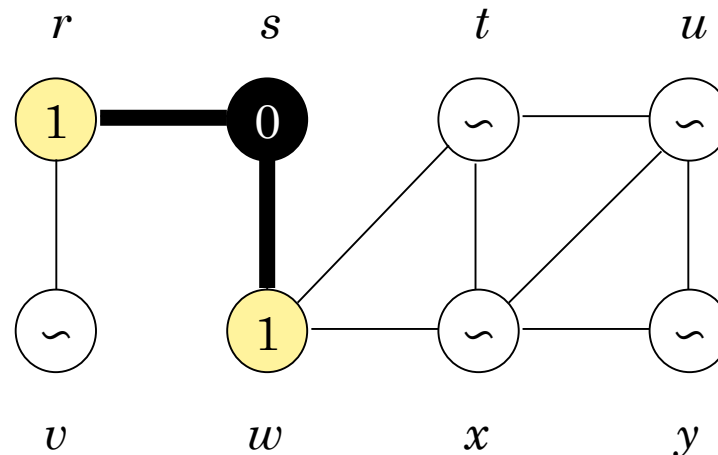
- Given a graph  $G = (V, E)$  and a **source** vertex  $s$ , it explores the edges of  $G$  to "discover" every reachable vertex from  $s$ .
- **It discovers vertices in the increasing order of distance from the source.** It first discovers all vertices at distance 1, then 2, and etc.



# BREADTH-FIRST SEARCH

## ○ Breadth-first search

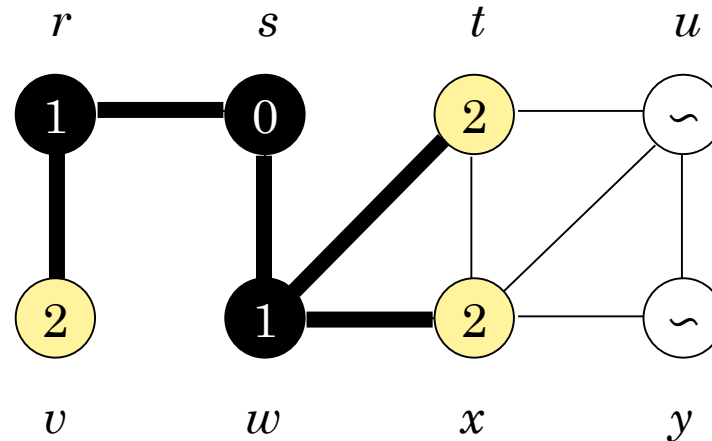
- Given a graph  $G = (V, E)$  and a **source** vertex  $s$ , it explores the edges of  $G$  to "discover" every reachable vertex from  $s$ .
- It discovers vertices in the increasing order of distance from the source. It first discovers all vertices at distance 1, then 2, and etc.



# BREADTH-FIRST SEARCH

## ○ Breadth-first search

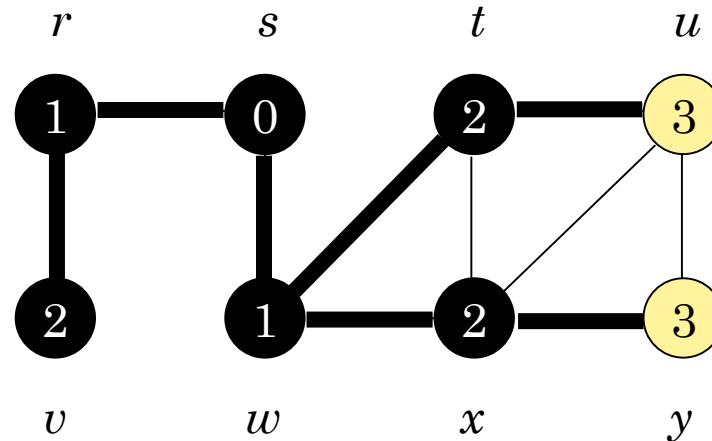
- Given a graph  $G = (V, E)$  and a **source** vertex  $s$ , it explores the edges of  $G$  to "discover" every reachable vertex from  $s$ .
- It discovers vertices in the increasing order of distance from the source. It first discovers all vertices at distance 1, then 2, and etc.



# BREADTH-FIRST SEARCH

## ○ Breadth-first search

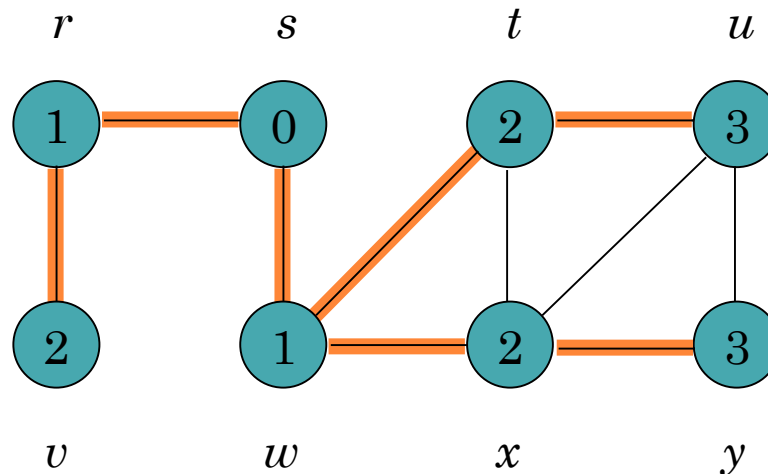
- Given a graph  $G = (V, E)$  and a **source** vertex  $s$ , it explores the edges of  $G$  to "discover" every reachable vertex from  $s$ .
- It discovers vertices in the increasing order of distance from the source. It first discovers all vertices at distance 1, then 2, and etc.



# BREADTH-FIRST SEARCH

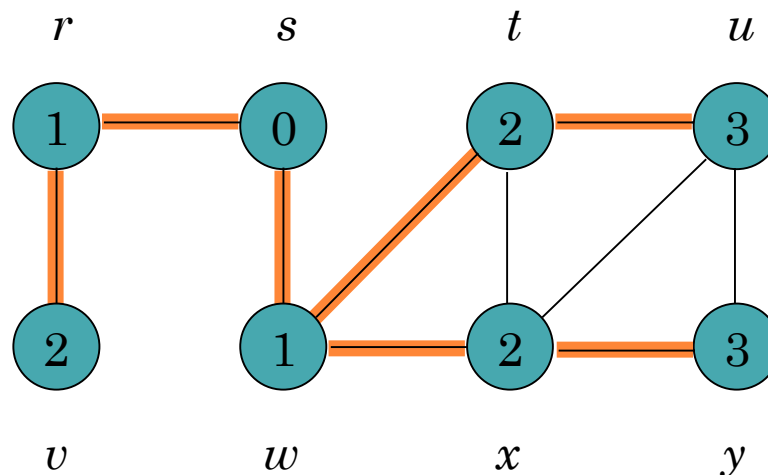
## ○ Breadth-first search

- It also computes
  - the distance of vertices from the source:  $d[u] = 3$
  - the predecessor of vertices:  $\pi[u] = t$



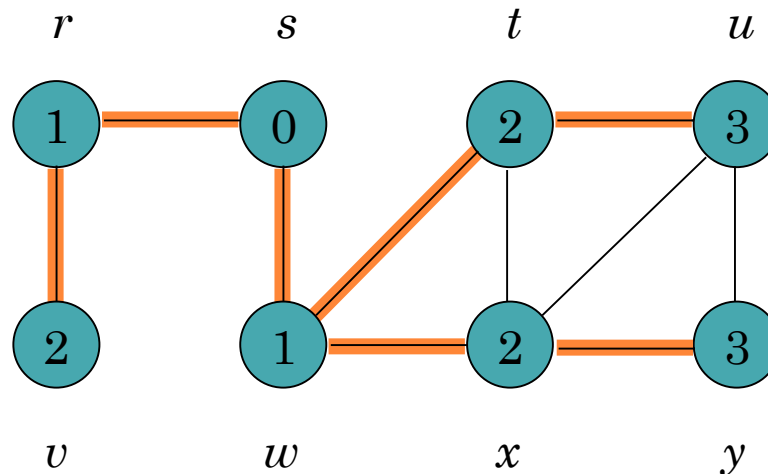
# BREADTH-FIRST SEARCH

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



# BREADTH-FIRST SEARCH

- The predecessor subgraph  $G_\pi$  is a ***breadth-first tree***.
  - since it is connected and  $|E_\pi| = |V_\pi| - 1$ .
  - The edges in  $E_\pi$  are called ***tree edges***.



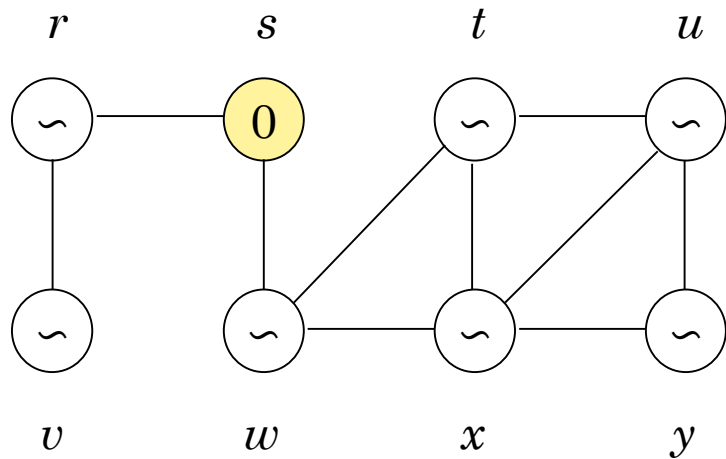


# BREADTH-FIRST SEARCH

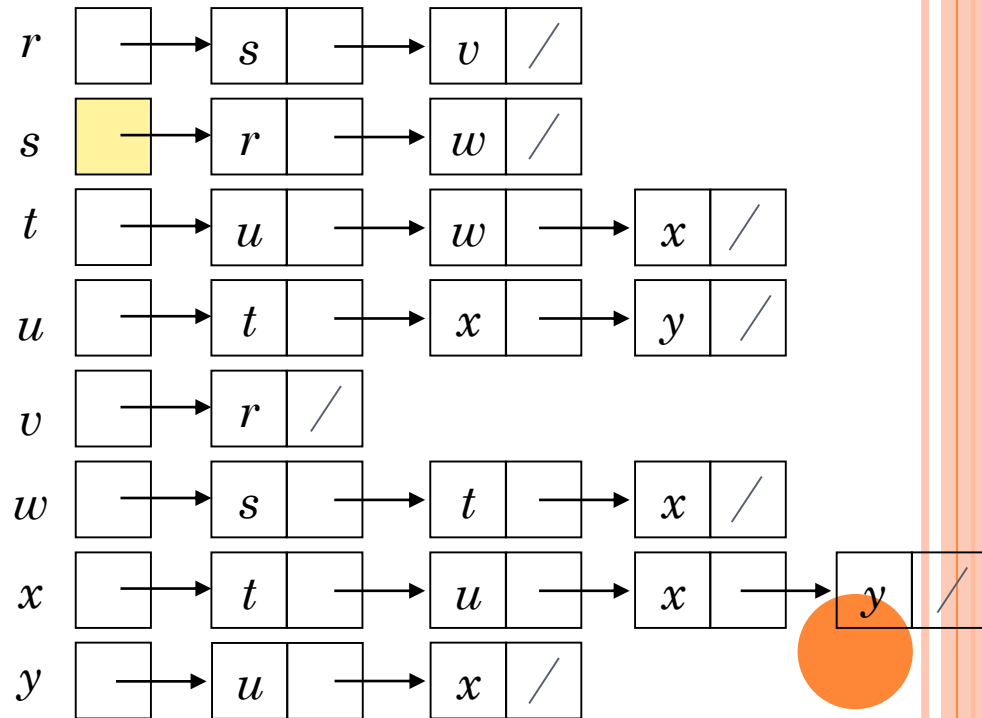
```
BFS( $G, s$ )
1 for each vertex  $u \in V[G] - \{s\}$ 
2   do  $color[u] \leftarrow \text{WHITE}$ 
3      $d[u] \leftarrow \infty$ 
4      $\pi[u] \leftarrow \text{NIL}$ 
5  $color[s] \leftarrow \text{GRAY}$ 
6  $d[s] \leftarrow 0$ 
7  $\pi[s] \leftarrow \text{NIL}$ 
8  $Q \leftarrow \emptyset$ 
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  $color[u] \leftarrow \text{BLACK}$ 
```



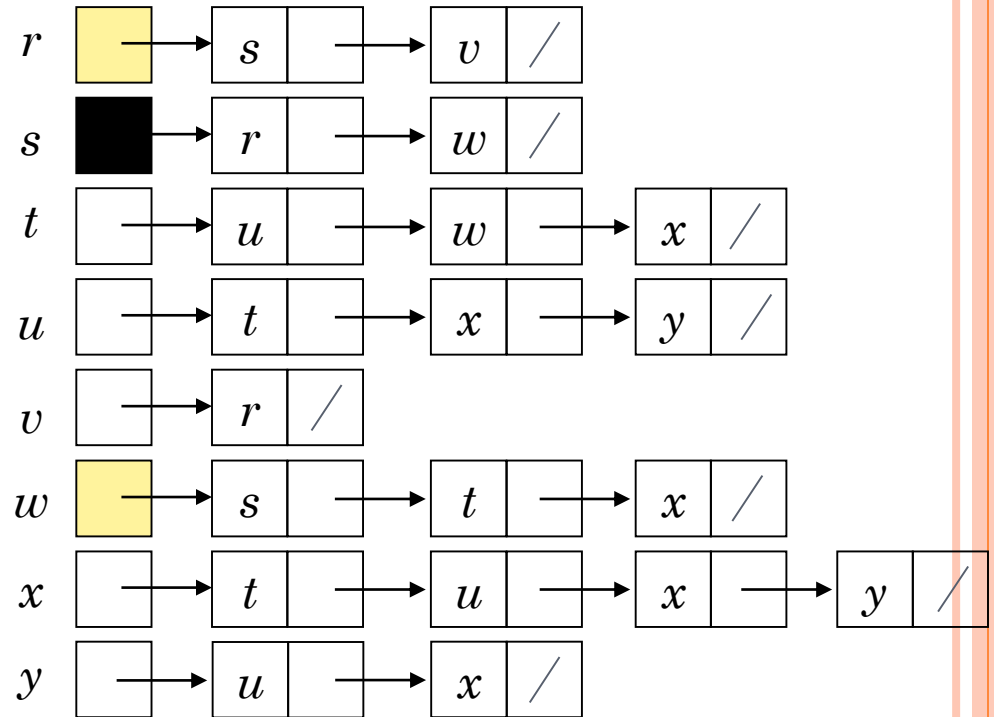
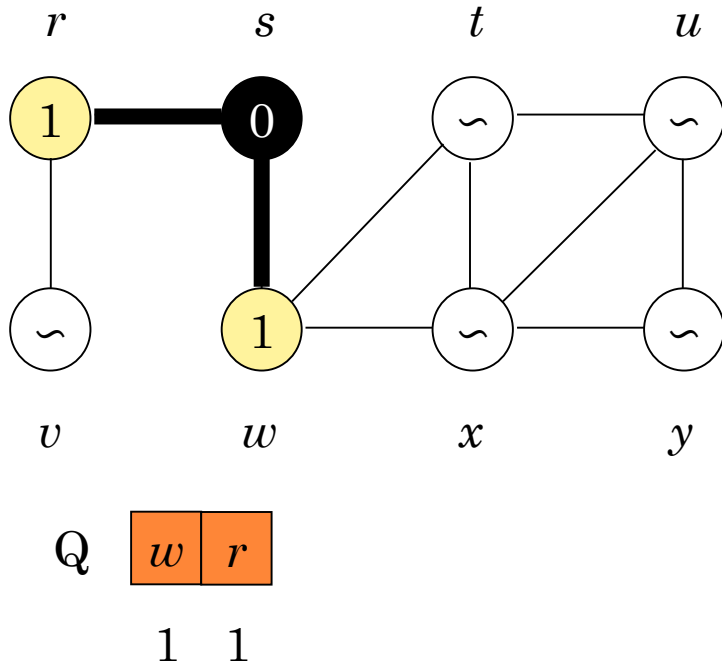
# BREADTH-FIRST SEARCH



Q s  
0



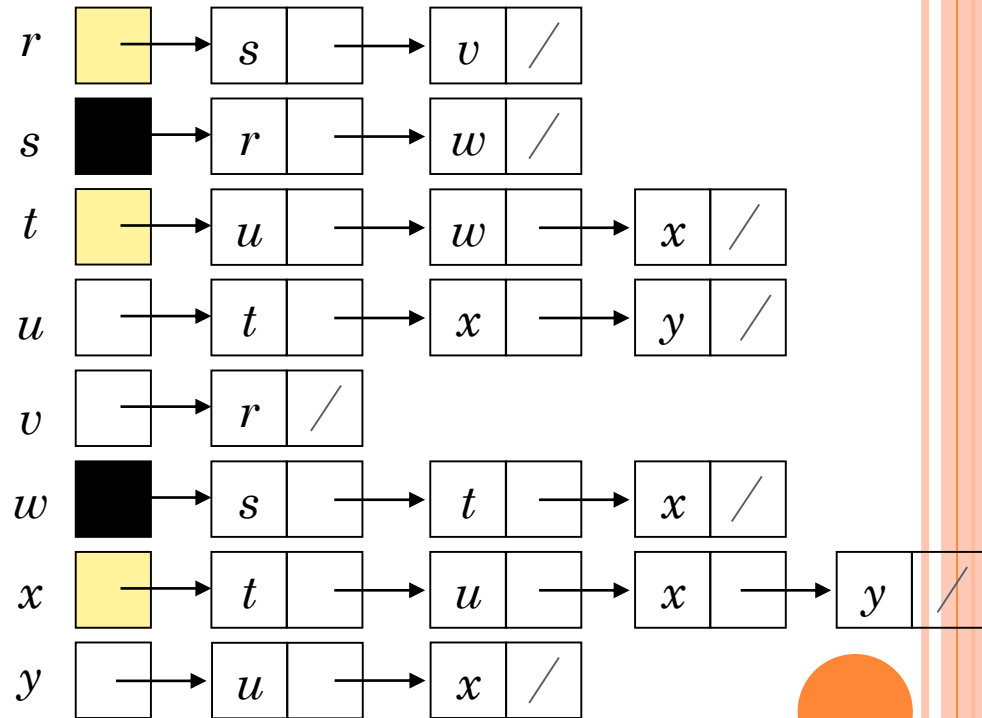
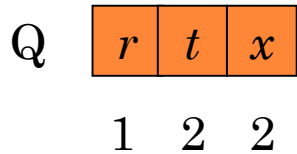
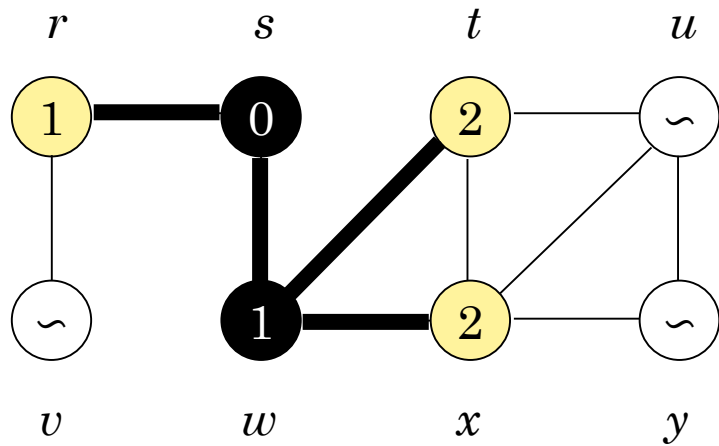
# BREADTH-FIRST SEARCH



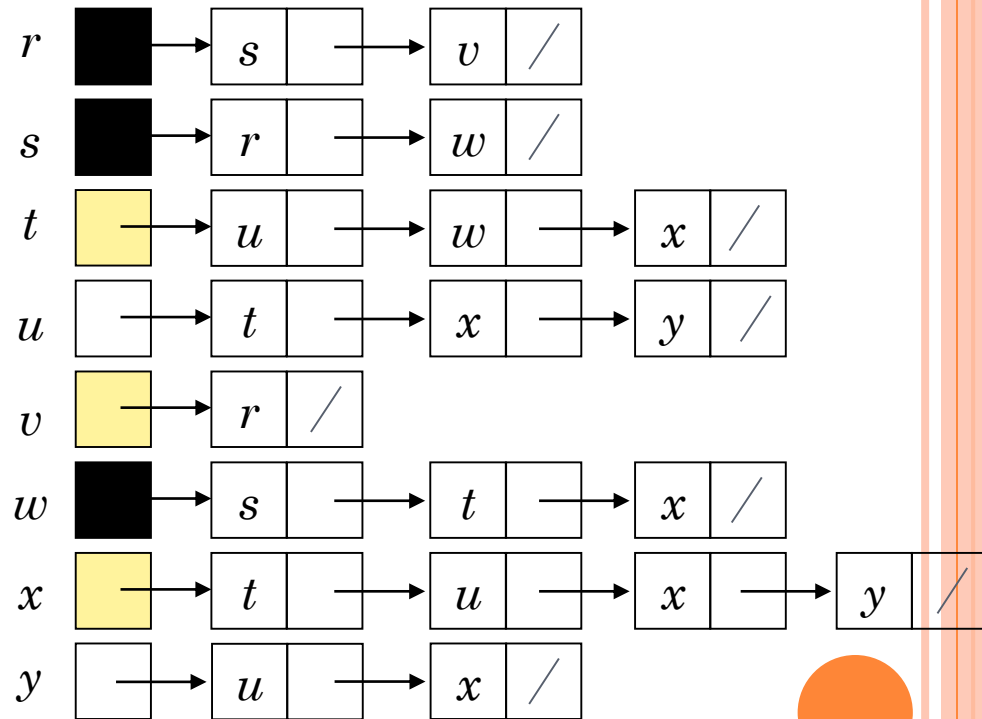
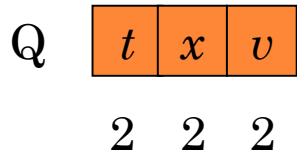
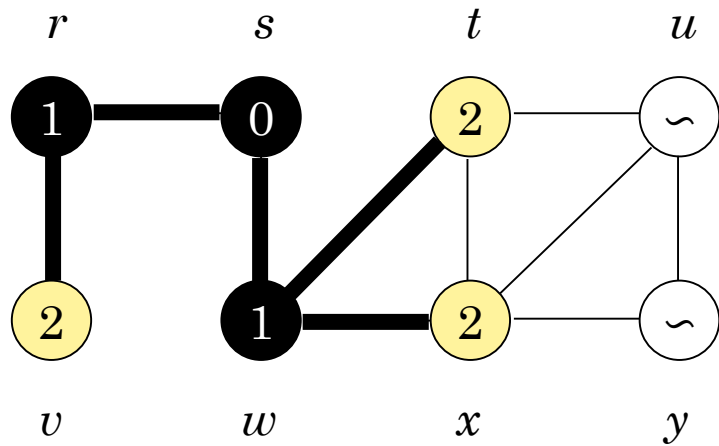
- white: not discovered (not entered the  $Q$ )
- gray: discovered (in the  $Q$ )
- black: finished (out of the  $Q$ )



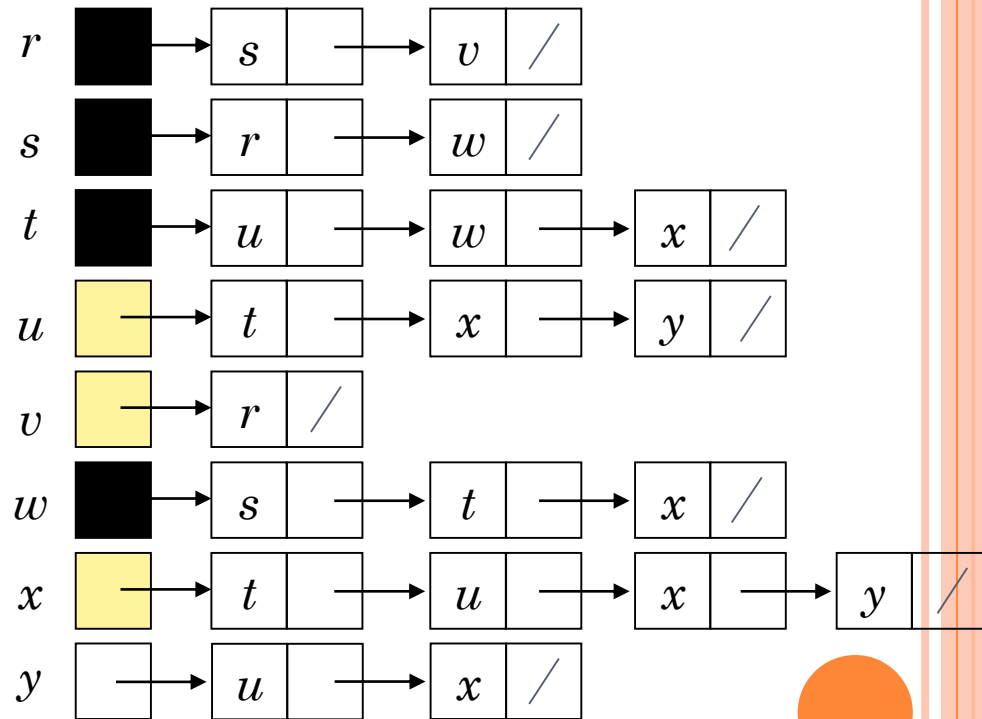
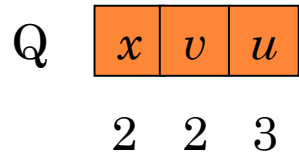
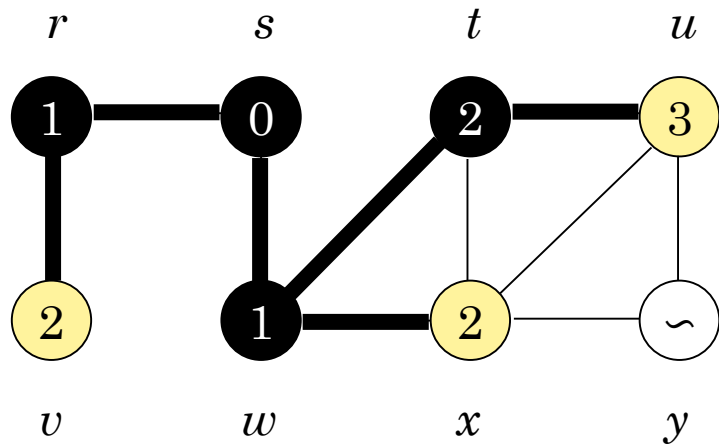
# BREADTH-FIRST SEARCH



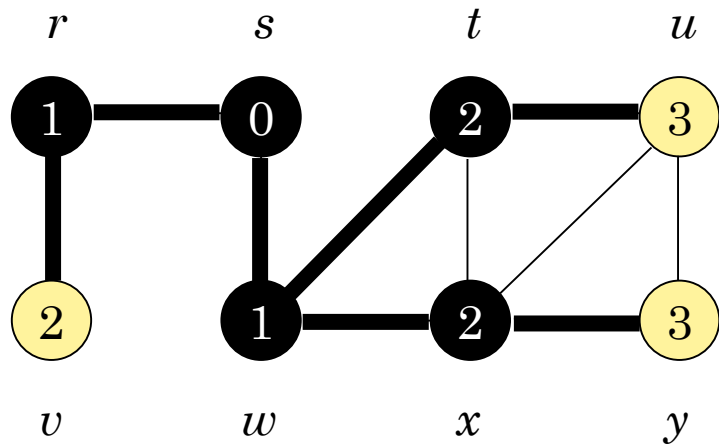
# BREADTH-FIRST SEARCH



# BREADTH-FIRST SEARCH

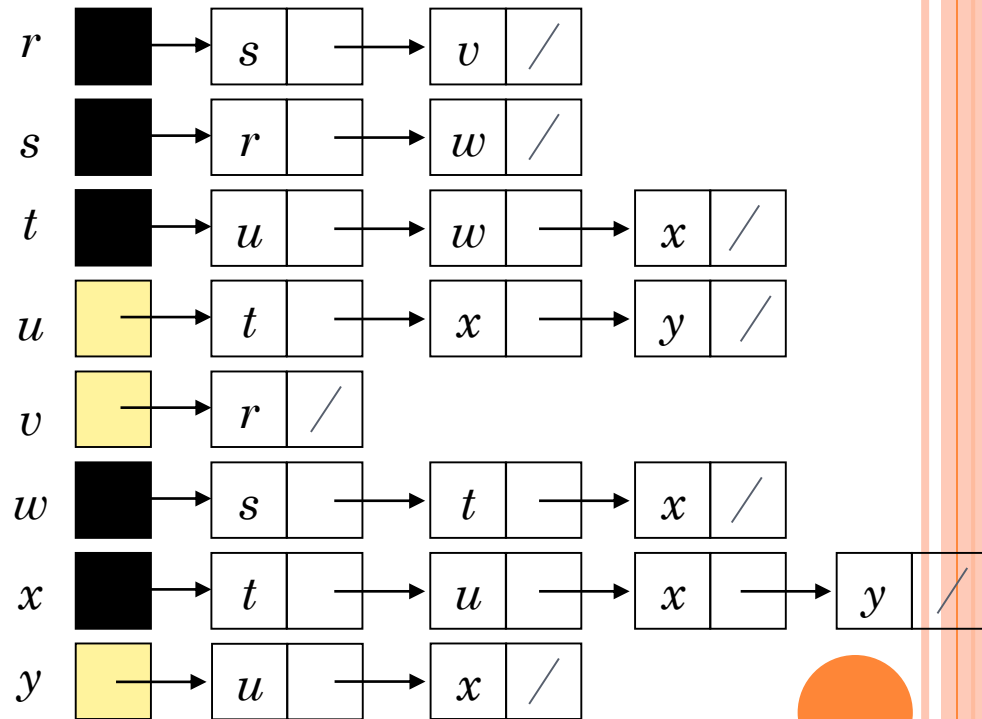


# BREADTH-FIRST SEARCH

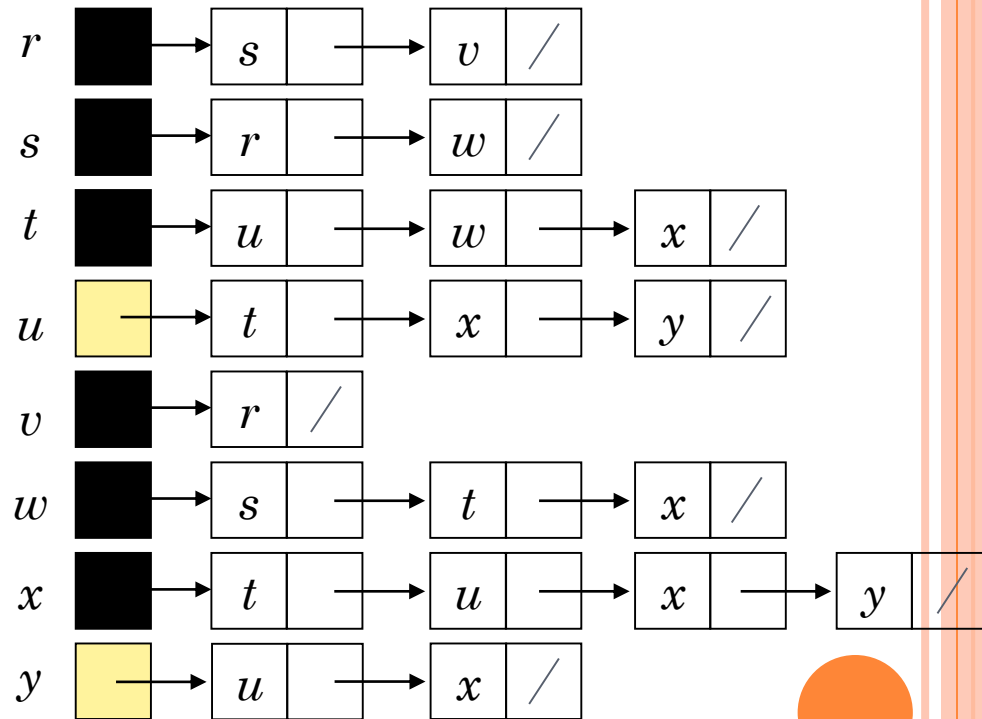
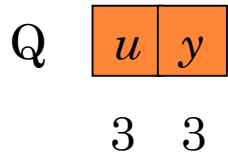
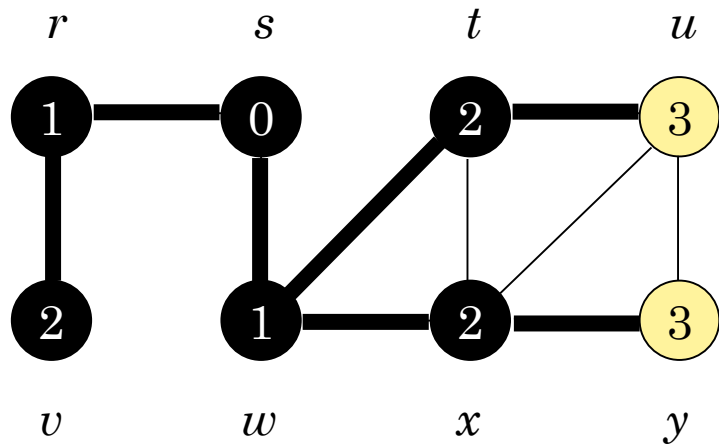


Q

$v$	$u$	$y$
2	3	3

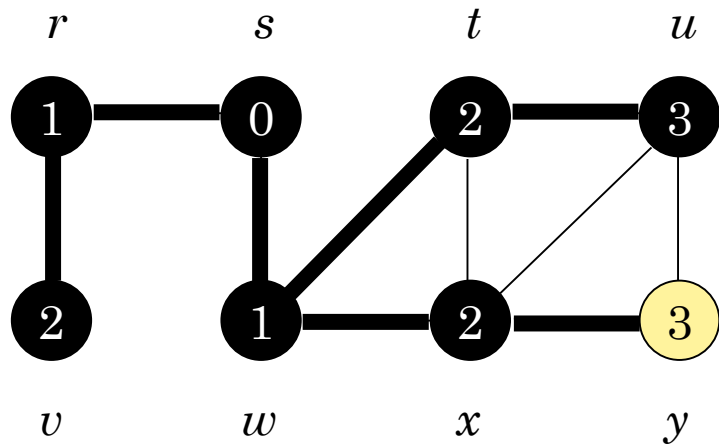


# BREADTH-FIRST SEARCH

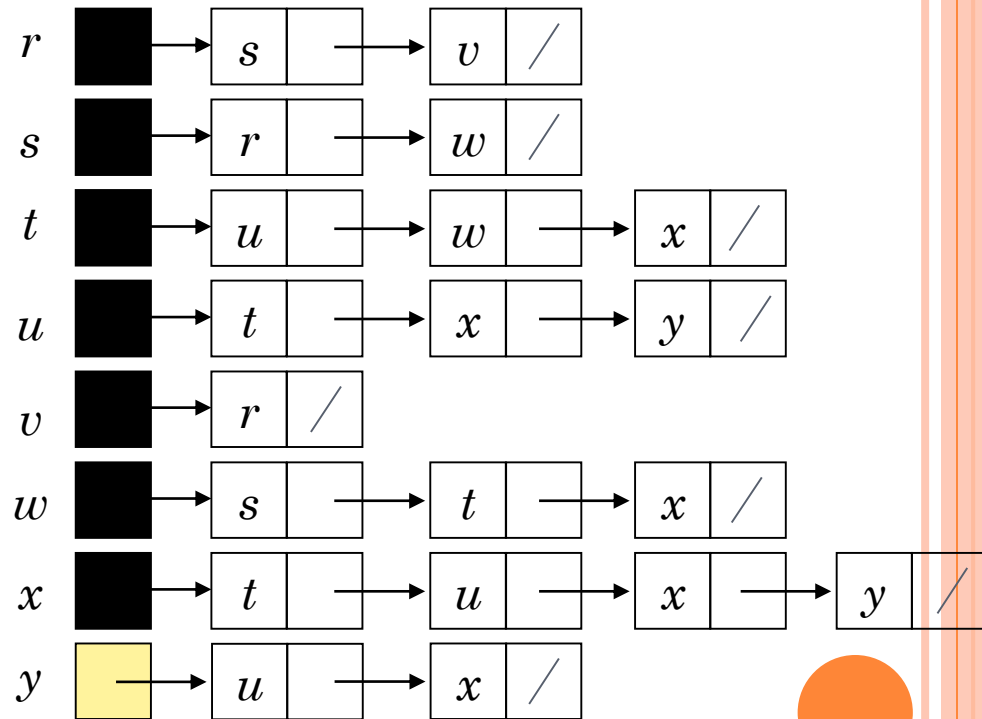




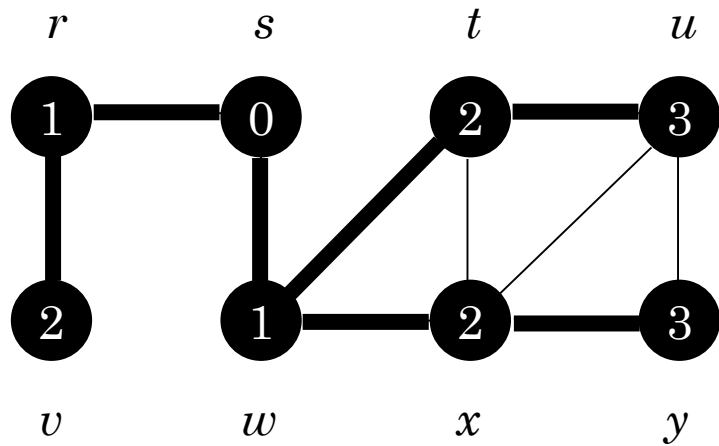
# BREADTH-FIRST SEARCH



Q y  
3

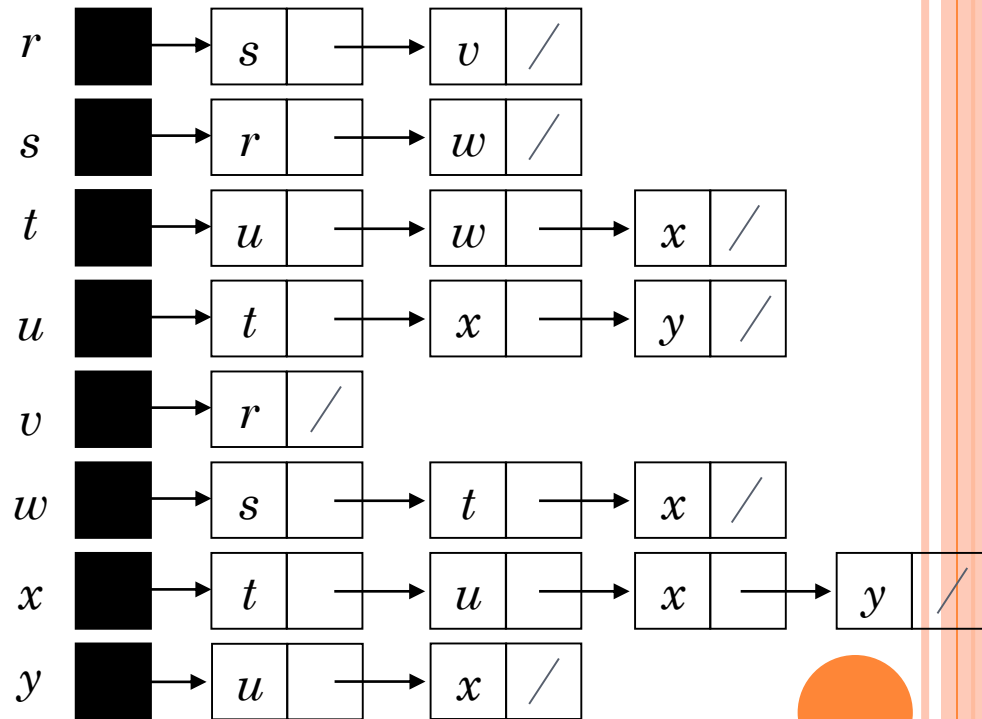


# BREADTH-FIRST SEARCH



$Q$   $\emptyset$

3



# BREADTH-FIRST SEARCH

```
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  $color[u] \leftarrow \text{BLACK}$ 
```

$O(V)$

$O(E)$

$O(V + E)$



# ANALYSIS

- **Running time**
  - **Initialization:**  $\Theta(V)$
  - **Exploring the graph:**  $O(E)$ 
    - An enqueue operation for an edge exploration.
    - #deque operations = #enqueue operations
    - An edge is explored at most once.
  - **Overall:**  $O(V + E)$



# ANALYSIS- INITIALIZATION

- After **initialization**, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once.
- The operations of enqueueing and dequeueing take  $O(1)$  time, and so the total time devoted to queue operations is  **$O(V)$** .

# ANALYSIS- EXPLORING THE GRAPH

- Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once.
- Since the sum of the lengths of all the adjacency lists is  $\Theta(E)$ , the total time spent in scanning adjacency lists is  $O(E)$
- The overhead for initialization is  $O(V)$ , and thus the total running time of the BFS procedure is  $O(V + E)$ .
- Thus, breadth-first search runs in **time linear** in the size of the adjacency-list representation of  $G$ .

# CONTENT

- *Breadth-first search*
- **Depth-first search**



# DEPTH FIRST SEARCH ALGORITHM

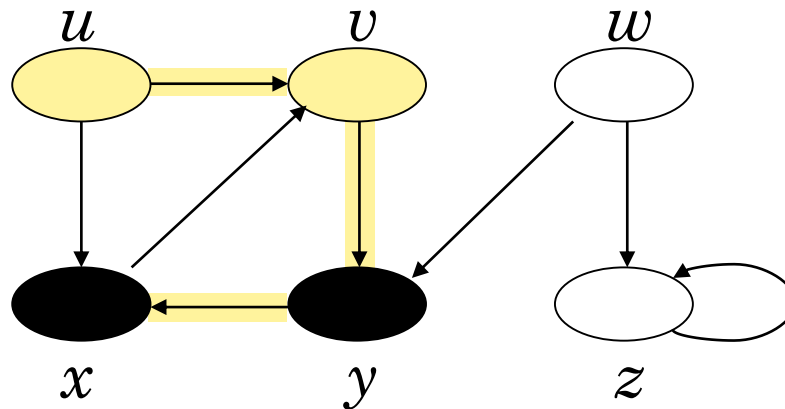
- Depth-first search explores edges out of the most recently discovered vertex that still has unexplored edges leaving it.
- Once all of its edges have been explored, the search “backtracks” to explore edges leaving the vertex from which it was discovered.
- This process continues until we have discovered all the vertices that are reachable from the original source vertex.
- If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source.
- The algorithm repeats this entire process until it has discovered every vertex.



# DEPTH-FIRST SEARCH

## ○ *Colors of vertices*

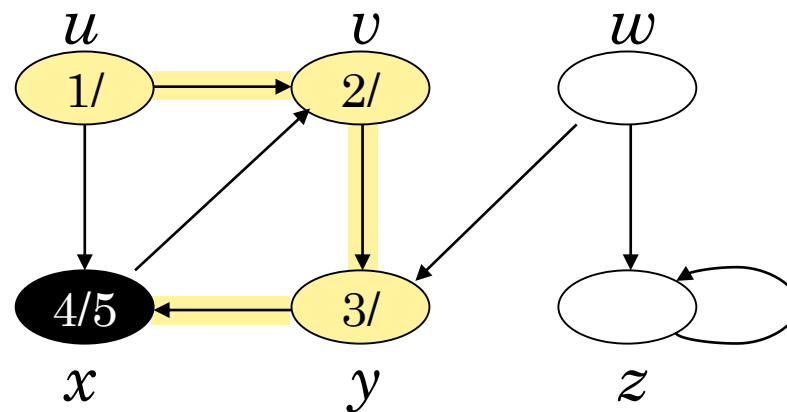
- Each vertex is initially *white*. (not discovered)
- The vertex is *grayed* when it is *discovered*.
- The vertex is *blackened* when it is *finished*, that is, when its adjacency list has been examined completely.



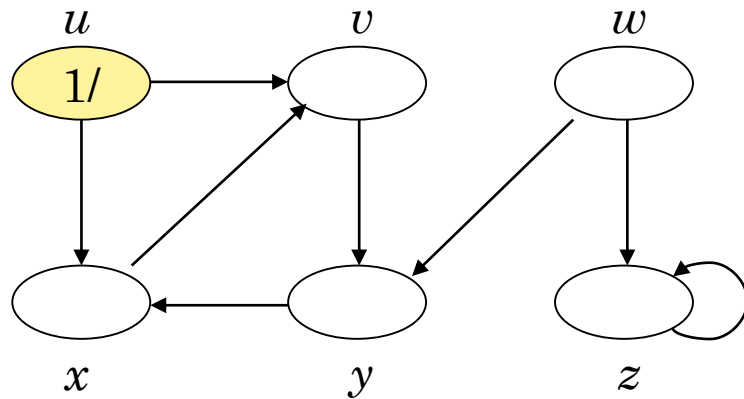
# DEPTH-FIRST SEARCH

## Timestamps

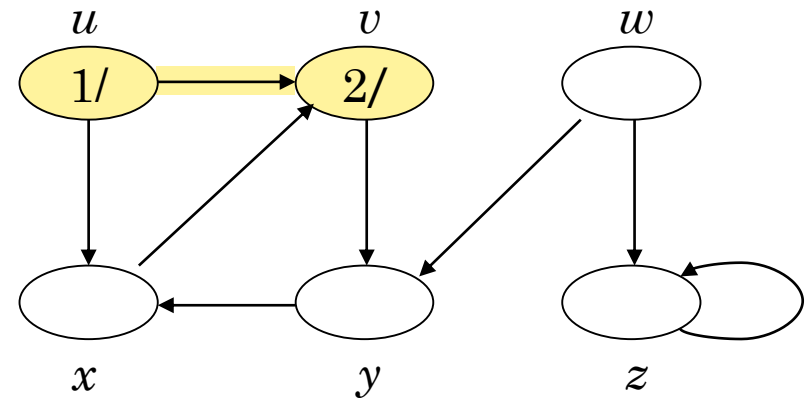
- Each vertex  $v$  has two timestamps.
  - $d[v]$ : **discovery time** (when  $v$  is grayed)
  - $f[v]$ : **finishing time** (when  $v$  is blacken)



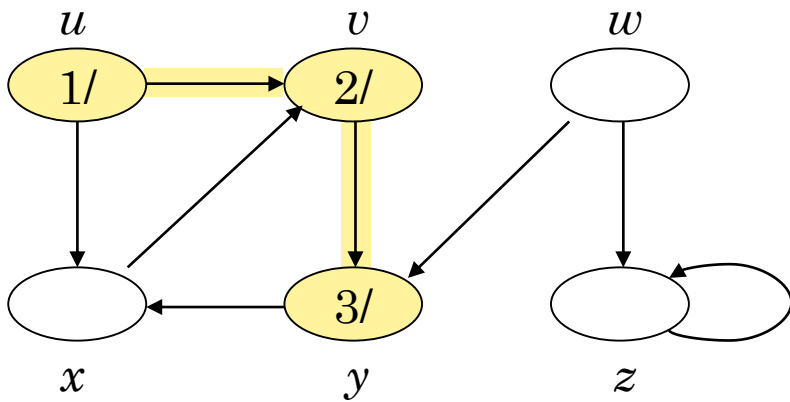
# DEPTH-FIRST SEARCH



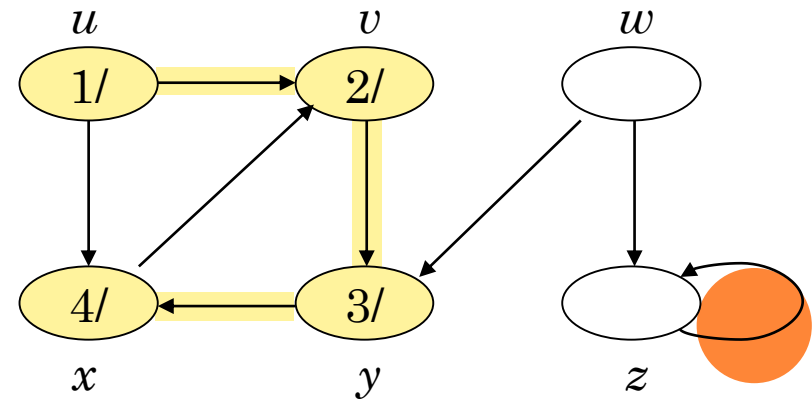
(a)



(b)

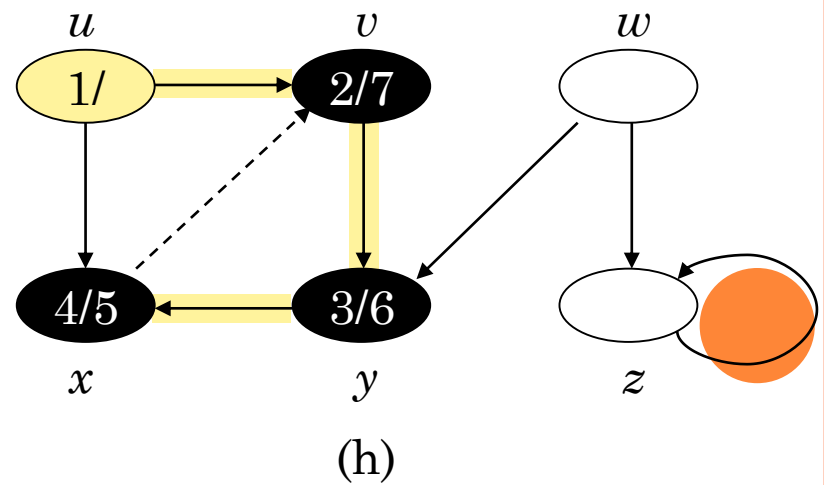
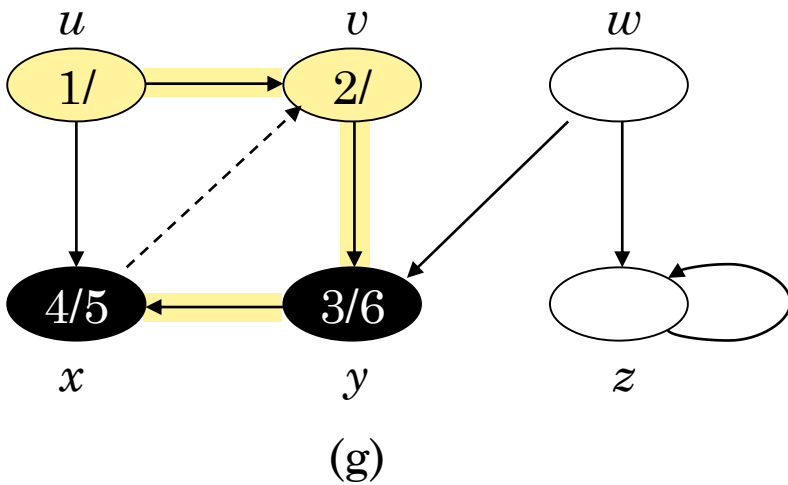
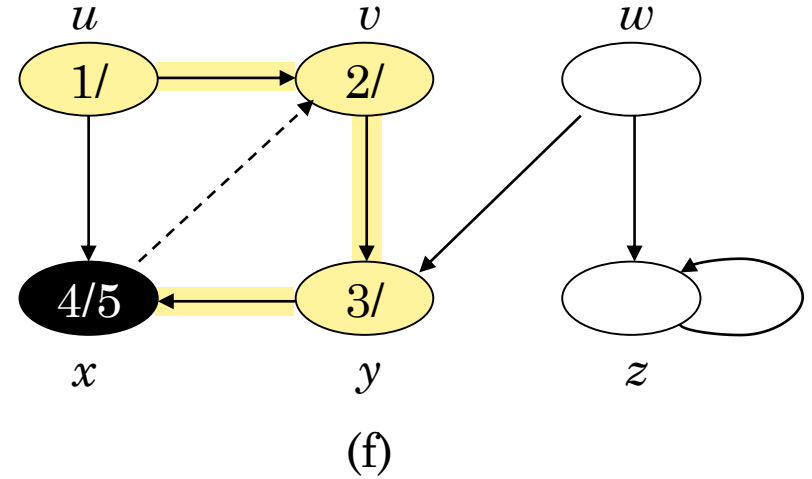
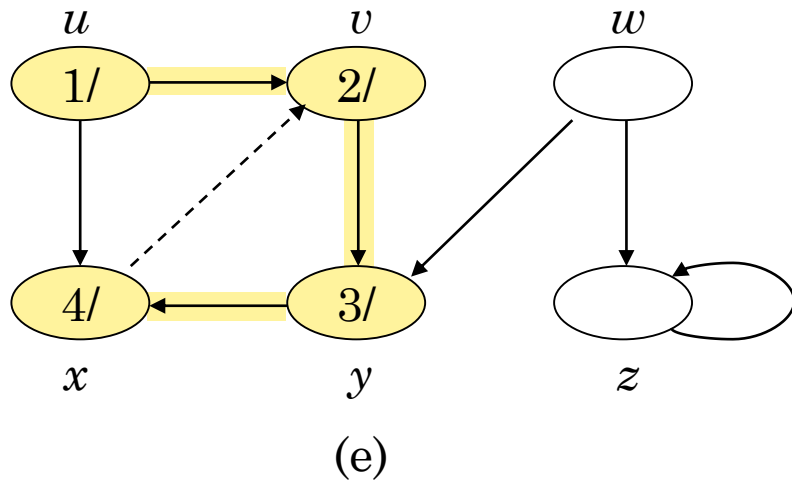


(c)

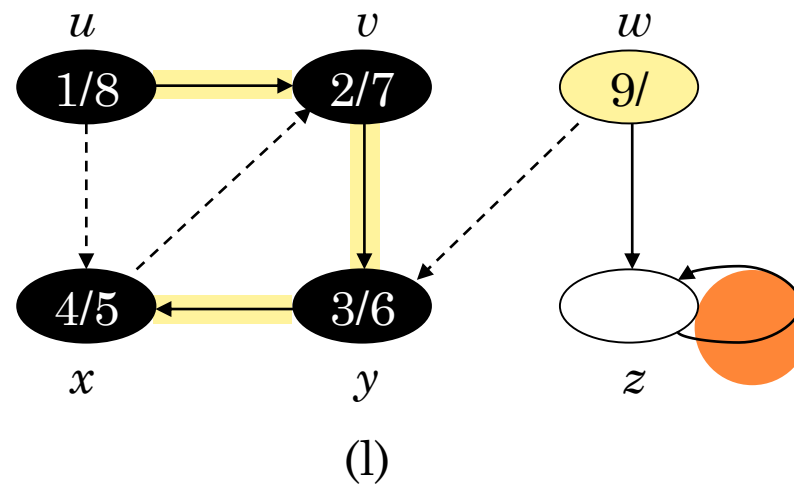
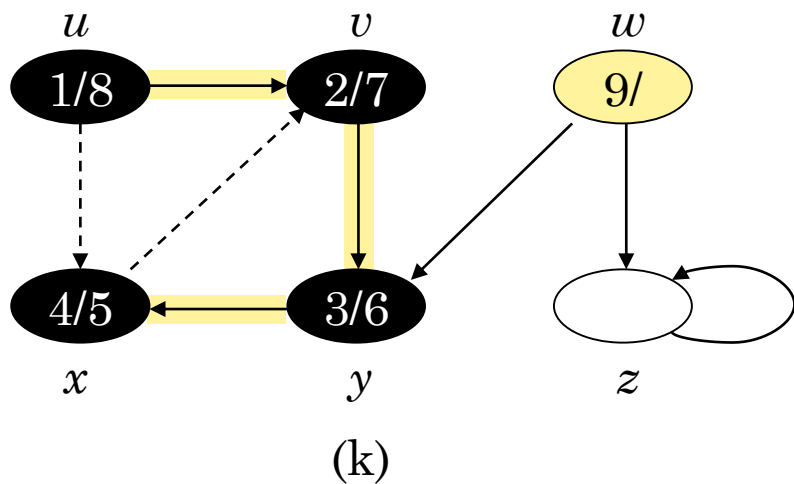
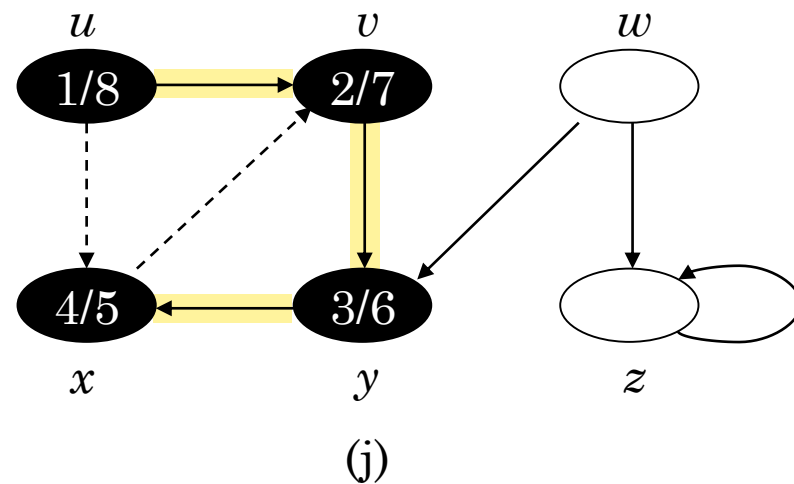
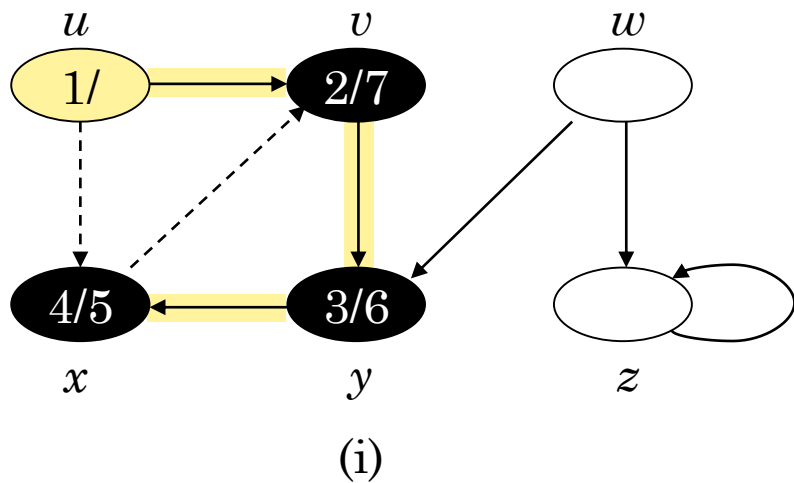


(d)

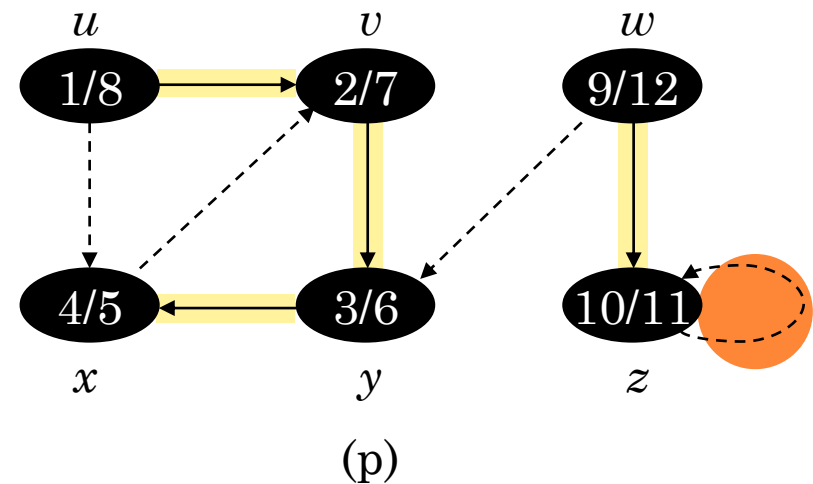
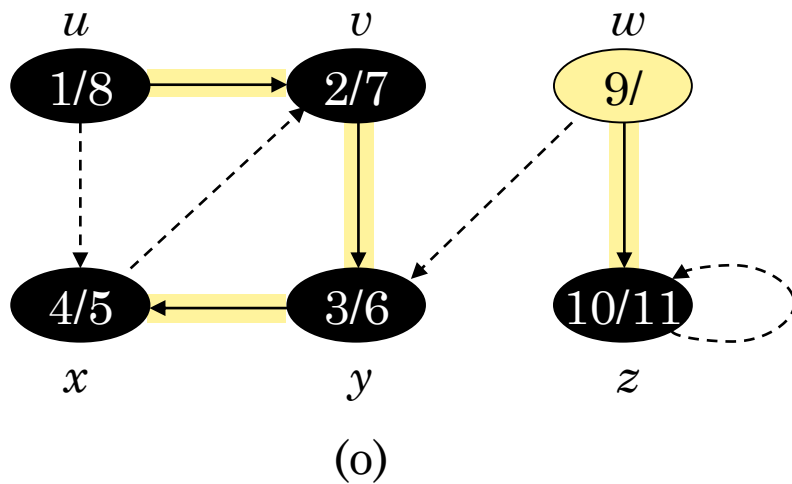
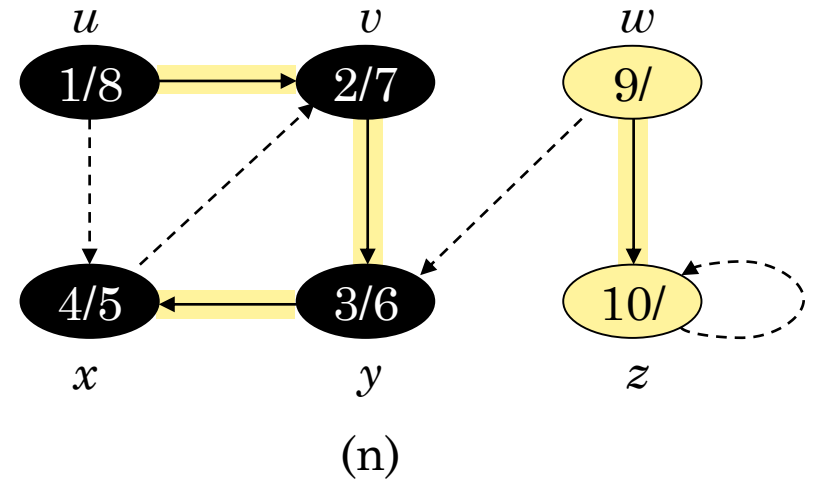
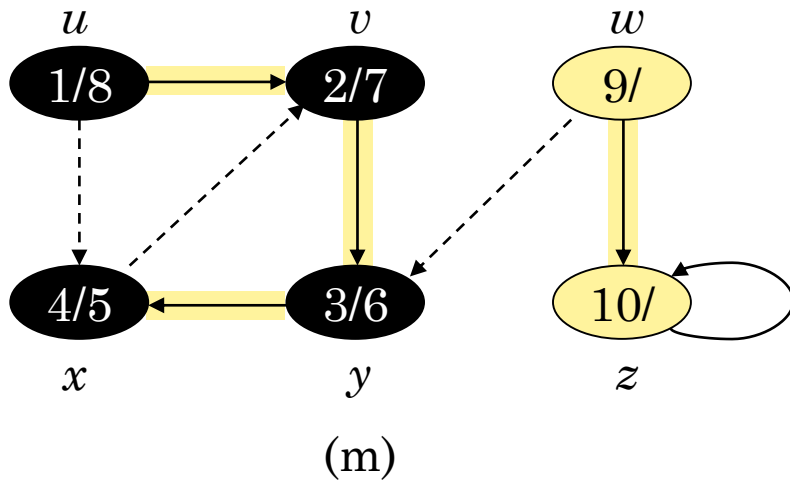
# DEPTH-FIRST SEARCH



# DEPTH-FIRST SEARCH

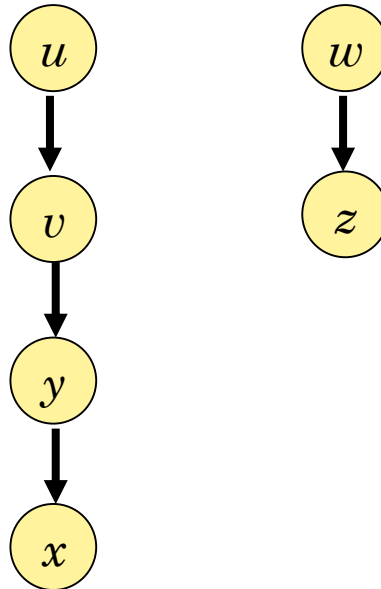


# DEPTH-FIRST SEARCH



# DEPTH-FIRST SEARCH

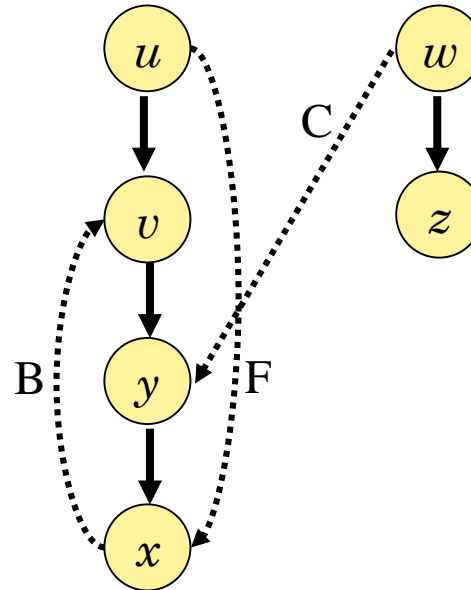
- The *predecessor subgraph* is a *depth-first forest*.



# DEPTH-FIRST SEARCH

## • Classification of edges

- *Tree edges*
- *Back edges*
- *Forward edges*
- *Cross edges*





# DEPTH-FIRST SEARCH

- ***Tree edges***: Edges in the depth-first forest.
- ***Back edges***: Those edges  $(u, v)$  connecting a vertex  $u$  to an ancestor  $v$  in a depth-first tree. Self-loops are considered to be back edges.
- ***Forward edges***: Those edges  $(u, v)$  connecting a vertex  $u$  to a descendant  $v$  in a depth-first tree.
- ***Cross edges***: 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.



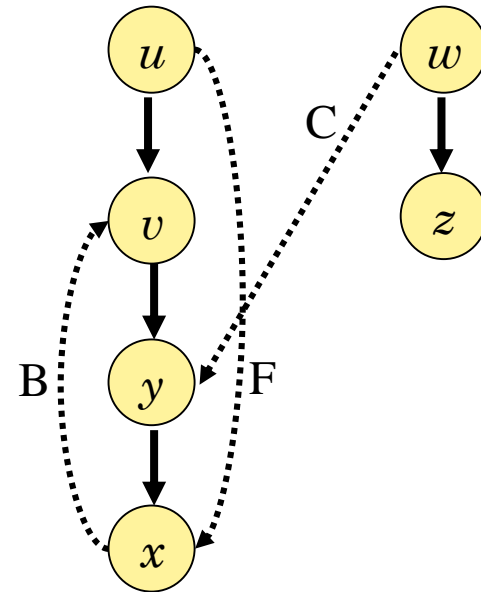
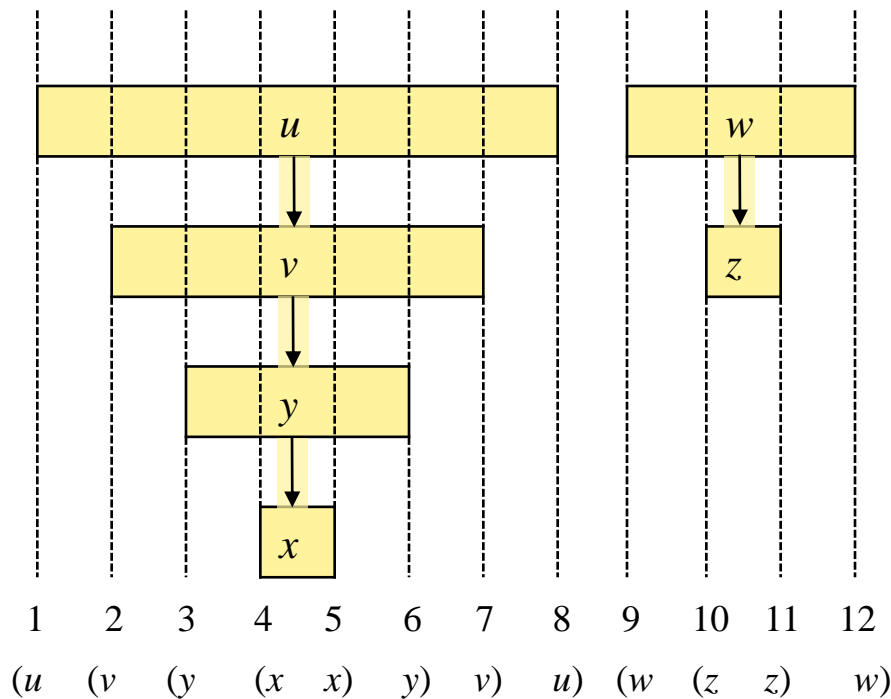
# DEPTH-FIRST SEARCH

- Classification by the DFS algorithm
  - Each edge  $(u, v)$  can be classified by the color of the vertex  $v$  that is reached when the edge is first explored:
    - WHITE indicates a tree edge,
    - GRAY indicates a back edge, and
    - BLACK indicates a forward or cross edge.

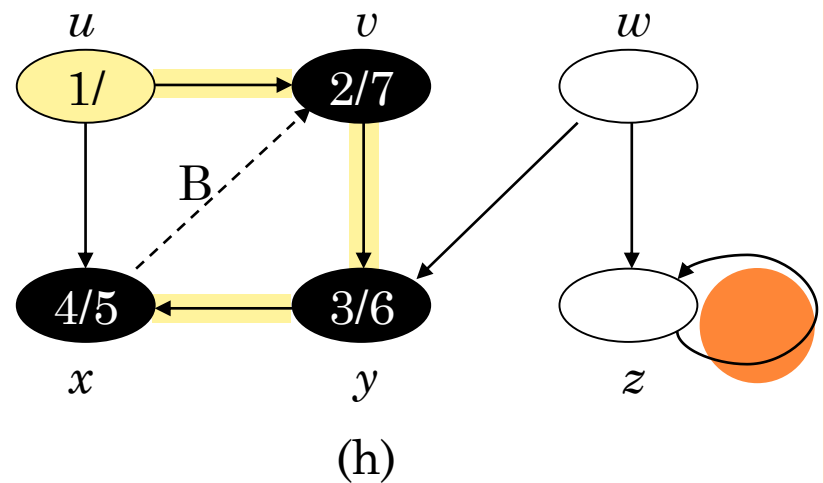
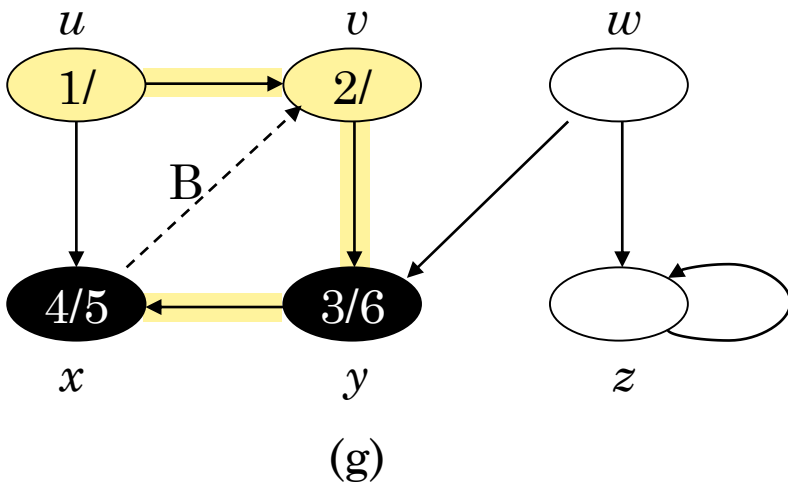
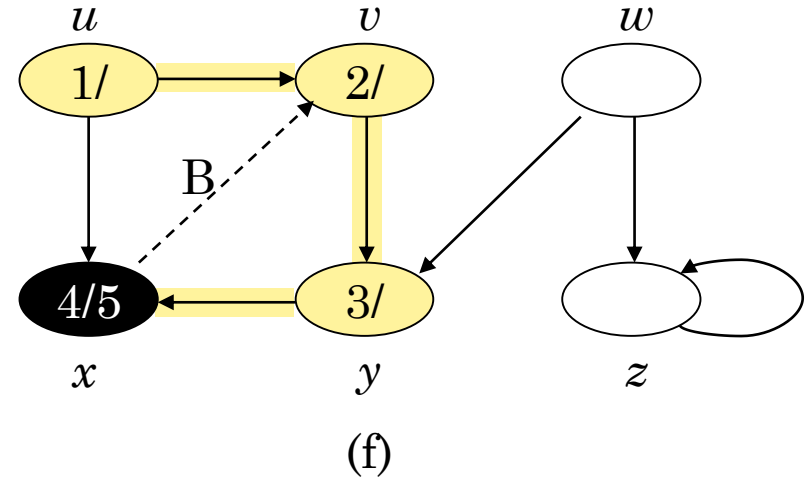
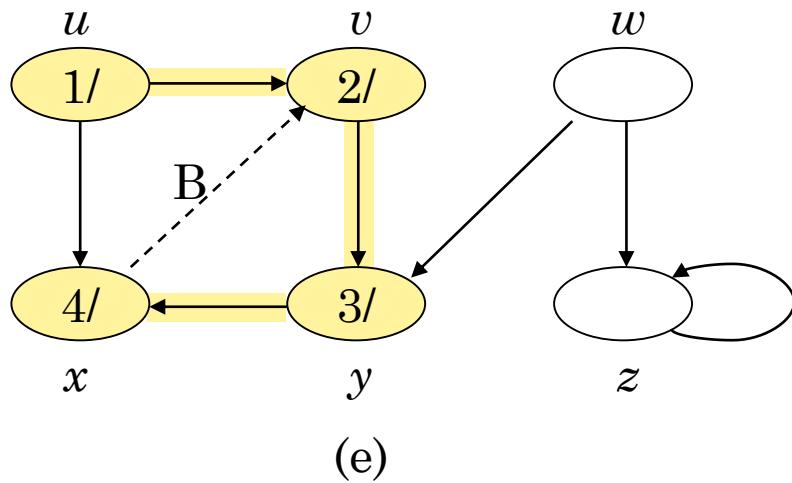


# DEPTH-FIRST SEARCH (*PARENTHESIS STRUCTURE*)

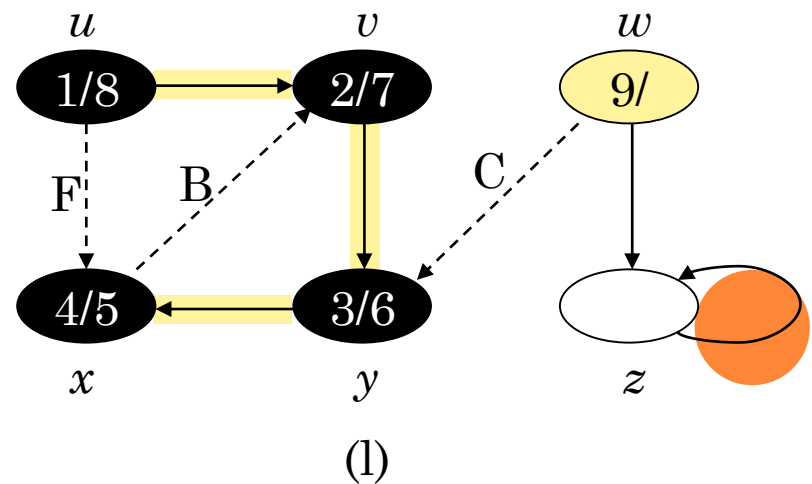
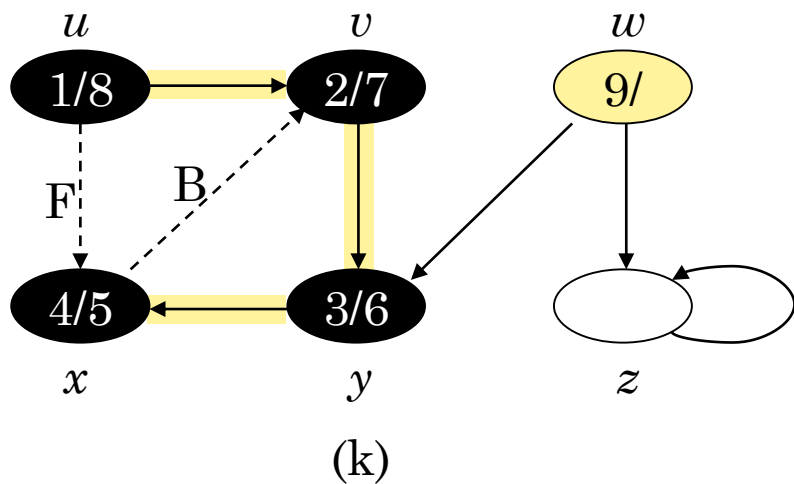
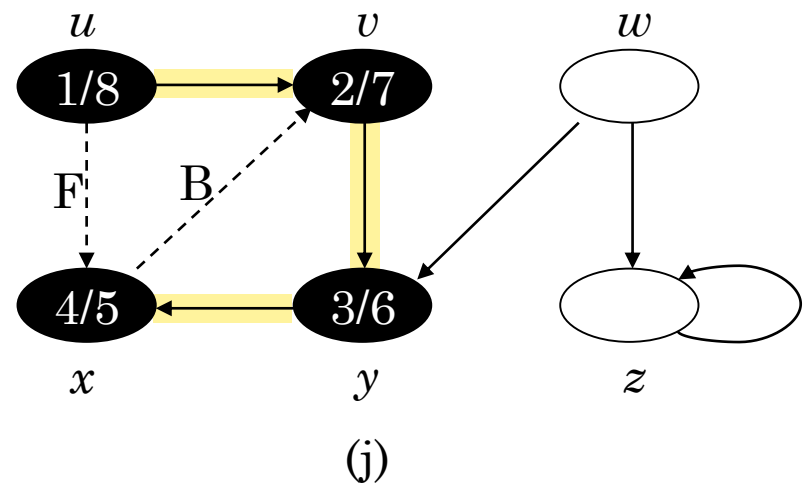
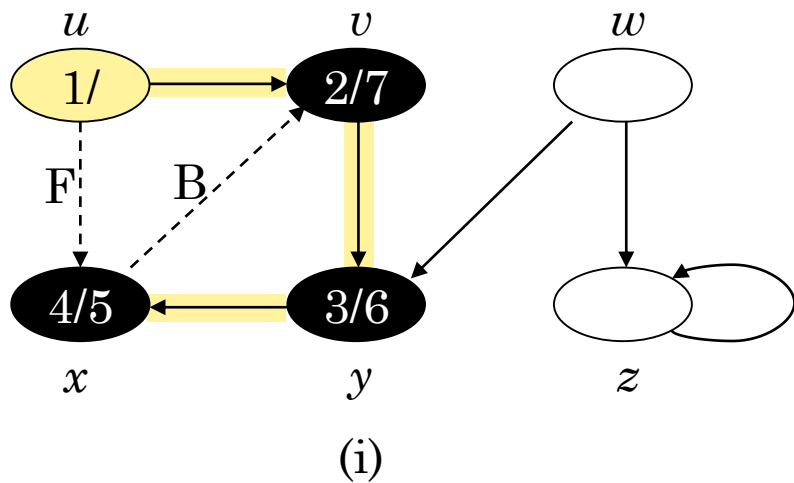
If we represent the discovery of vertex  $u$  with a left parenthesis “(u” and represent its finishing by a right parenthesis “u)”, then the history of discoveries and finishing makes a well-formed expression in the sense that the parentheses are properly nested.



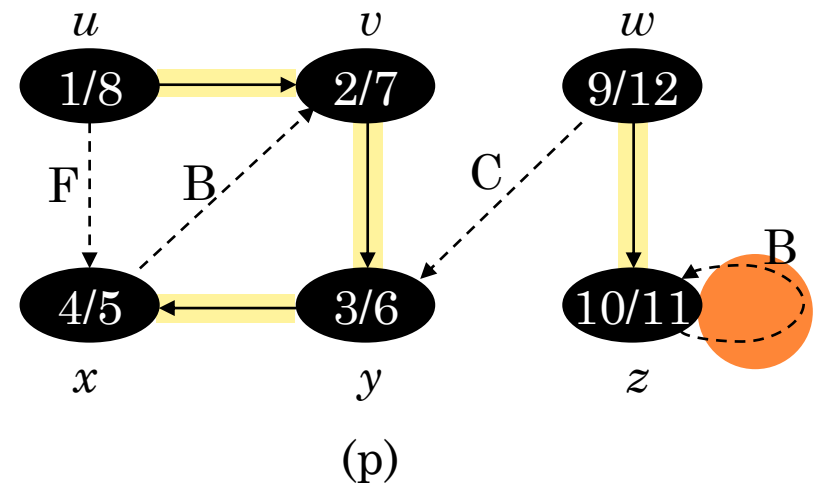
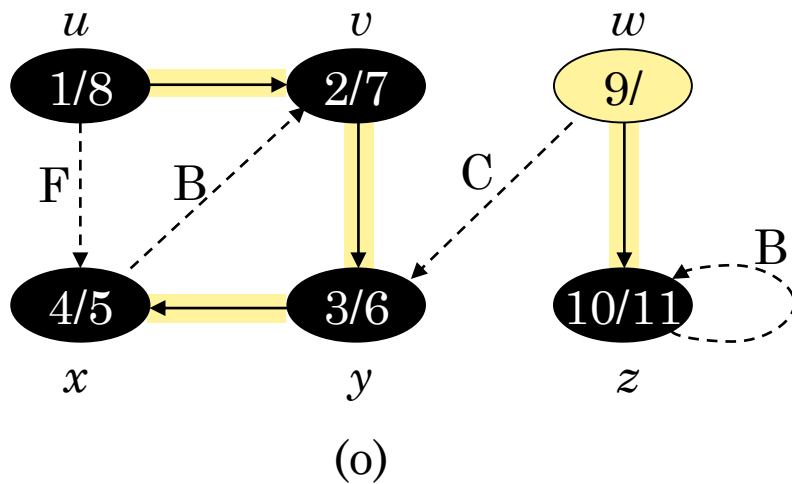
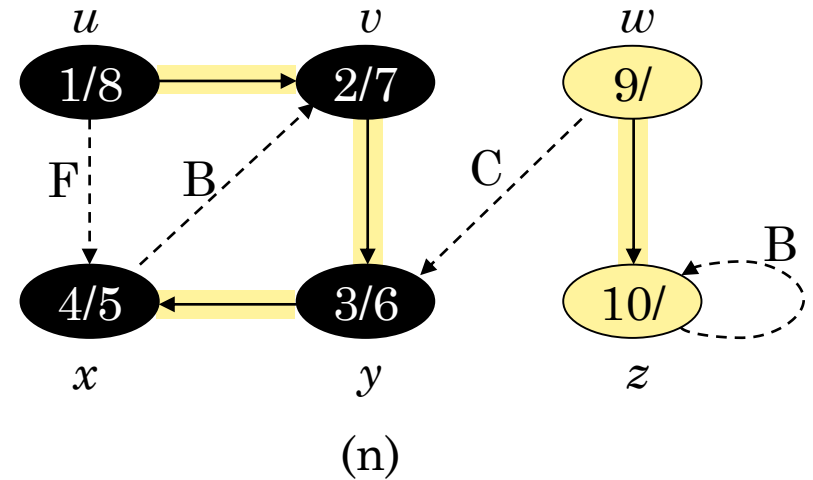
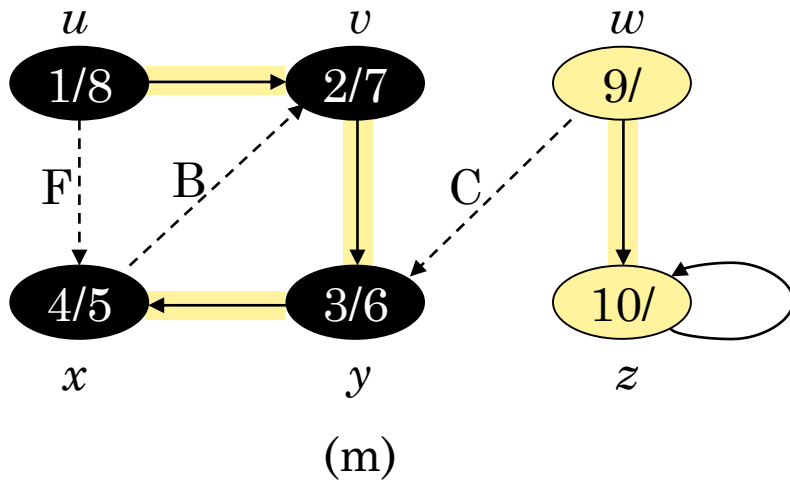
# DEPTH-FIRST SEARCH



# DEPTH-FIRST SEARCH



# DEPTH-FIRST SEARCH



# DEPTH-FIRST SEARCH

DFS(*G*)

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



$\Theta(V)$

DFS-VISIT(*G*, *u*)

```
1   $time = time + 1$            // white vertex  $u$  has just been discovered
2   $u.d = time$ 
3   $u.color = \text{GRAY}$ 
4  for each  $v \in G.Adj[u]$       // explore edge  $(u, v)$ 
5      if  $v.color == \text{WHITE}$ 
6           $v.\pi = u$ 
7          DFS-VISIT( $G, v$ )
8   $u.color = \text{BLACK}$          // blacken  $u$ ; it is finished
9   $time = time + 1$ 
10  $u.f = time$ 
```



$\Theta(E)$

$\Theta(V+E)$

# ANALYSIS

- The loops on lines 1–3 and lines 5–7 of DFS take time  $\Theta(V)$ , exclusive of the time to execute the calls to DFS-VISIT.
- The procedure DFS-VISIT is called exactly once for each vertex  $u \in V$ , since the vertex  $u$  on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex  $u$  gray.
- During an execution of DFS-VISIT  $(G, v)$ , the loop on lines 4–7 executes  $|Adj[v]|$  times. Since  $\sum_{v \in V} |Adj[v]| = \Theta(E)$ ,
- the total cost of executing lines 4–7 of DFS-VISIT is  $\Theta(E)$ .
- The running time of DFS is therefore  $\Theta(V + E)$

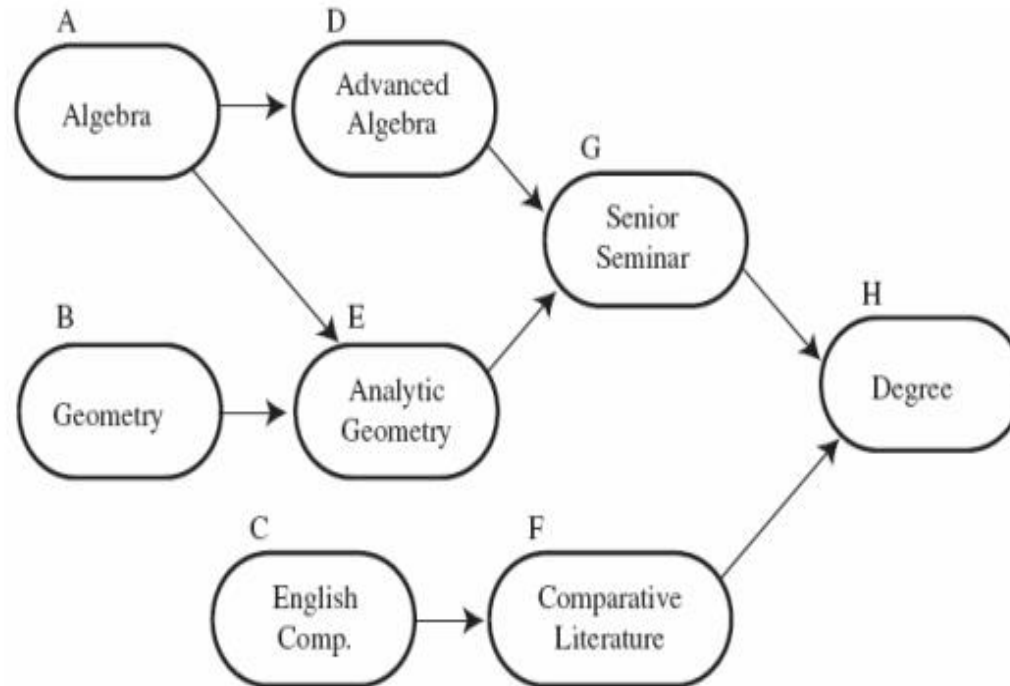




# APPLICATIONS

49

# TOPOLOGICAL SORTING

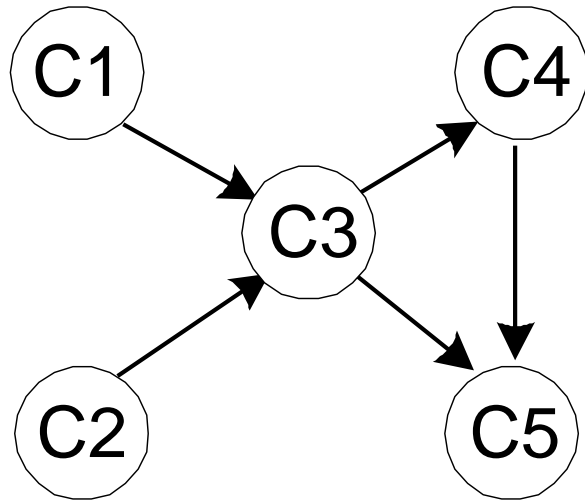


Course prerequisites

Topologically sorted order: C F B A E D G H

# TOPOLOGICAL SORTING

- Problem: find a total order consistent with a partial order
- Example:



Five courses has the prerequisite relation shown in the left. Find the right order to take all of them sequentially

Note: problem is solvable iff graph is dag



# TOPOLOGICAL SORTING ALGORITHMS

## DFS-based algorithm:

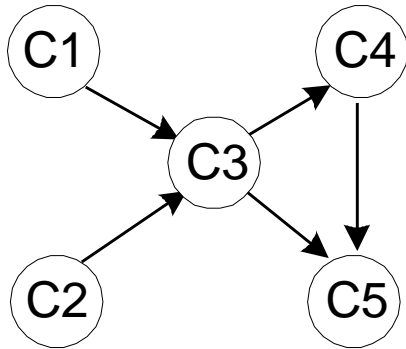
- DFS traversal: note the order with which the vertices are popped off stack (dead end)
- Reverse order solves topological sorting
- Back edges encountered? → NOT a DAG!

## Source removal algorithm

- Repeatedly identify and remove a *source* vertex, ie, a vertex that has no incoming edges
- Both  $\Theta(V+E)$  using adjacency linked lists



# AN EXAMPLE (TOPOLOGICAL SORT)



(a)

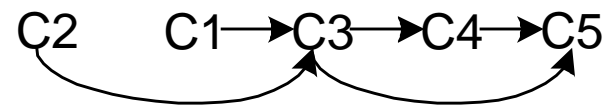
$C5_1$   
 $C4_2$   
 $C3_3$   
 $C1_4$   $C2_5$

(b)

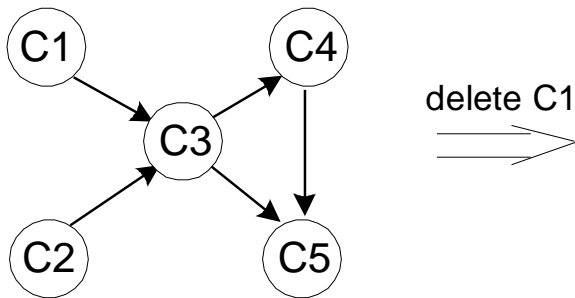
The popping-off order:

$C5, C4, C3, C1, C2$

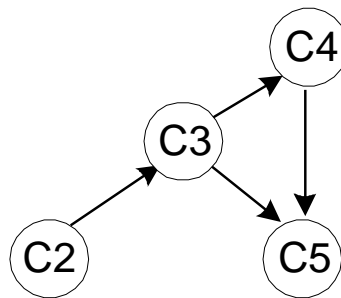
The topologically sorted list:



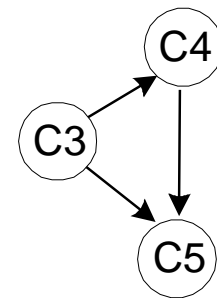
(c)



delete C1  
 $\Rightarrow$



delete C2  
 $\Rightarrow$



delete C3  
 $\Rightarrow$



delete C4  
 $\Rightarrow$



delete C5  
 $\Rightarrow$



# TOPOLOGICAL SORT

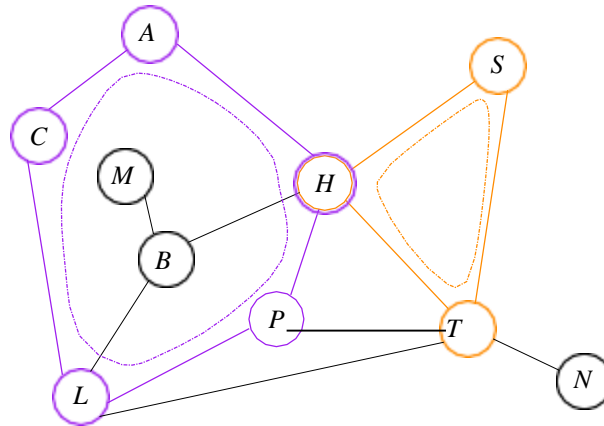
## TOPOLOGICAL-SORT( $G$ )

- 1 call DFS( $G$ ) to compute finishing times  $v.f$  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

# Cycle

A path  $(v_0, v_1, v_2, \dots, v_k)$  forms a **cycle** if  $v_0 = v_k$  and all edges on the path are distinct

- A cycle is **simple** if  $v_1, v_2, \dots, v_k$  are distinct
- A graph with no cycles is **acyclic**



- $(T, S, H, T)$  is a simple cycle
- $(A, C, L, P, H, A)$  is a simple cycle

# An Application of DFS: Cycle Finding

## Question

Given an undirected graph  $G$ , how to determine whether or not  $G$  contains a cycle?

## Lemma

*$G$  is acyclic if and only if a DFS of  $G$  yields no back edges.*

## Proof.

$\Rightarrow$ : Suppose that there is a back edge  $(u, v)$ . Then, vertex  $v$  is an ancestor (excluding predecessor) of  $u$  in the DFS trees. There is thus a path from  $v$  to  $u$  in  $G$ , and the back edge  $(u, v)$  completes a cycle.

$\Leftarrow$ : If there is no back edge, then since an edge in an undirected graph is either a tree edge or a back edge, there are only tree edges, implying that the graph is a forest, and hence is acyclic. □



# Cycle Finding

## Cycle(G)

```
foreach  $u$  in  $V$  do
    color[ $u$ ] = WHITE;
    pred[ $u$ ] = NULL;
end
foreach  $u$  in  $V$  do
    if color[ $u$ ] = WHITE then
        Visit( $u$ );
    end
end
output "No Cycle";
```

## Visit( $u$ )

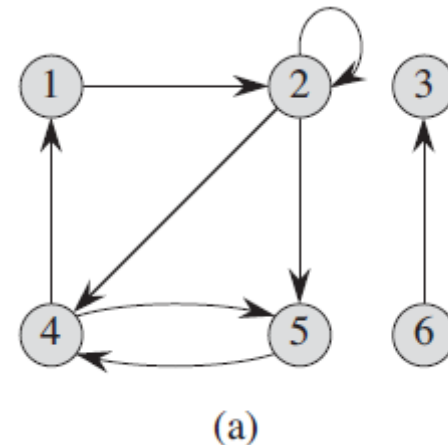
```
color[ $u$ ] = GRAY;
foreach  $v$  in Adj( $u$ ) do
    //consider ( $u, v$ )
    if color[ $v$ ] = WHITE then
        //  $v$  unvisited
        pred[ $v$ ] =  $u$ ;
        Visit( $v$ ); //visit  $v$ 
    else if  $v \neq \text{pred}[u]$  then
        //back edge detected
        output "Cycle found!";
        exit; //terminate
    end
end
color[ $u$ ] = BLACK;
```

Running time:  $O(V)$

- only traverse tree edges, until the first back edge is found
- at most  $V - 1$  tree edges

# STRONGLY CONNECTED COMPONENT

- A directed graph is ***strongly connected*** if every two vertices are reachable from each other.
- The ***strongly connected components*** of a directed graph are the equivalence classes of vertices under the “are mutually reachable” relation.
- A directed graph is strongly connected if it has only one strongly connected component.
- The graph in Figure B.2(a) has three strongly connected components:
  - {1; 2; 4; 5}, {3}, and {6}
  - All pairs of vertices in {1; 2; 4; 5} are mutually reachable.
  - The vertices {3; 6} do not form a strongly connected component, since vertex 6 cannot be reached from vertex 3.



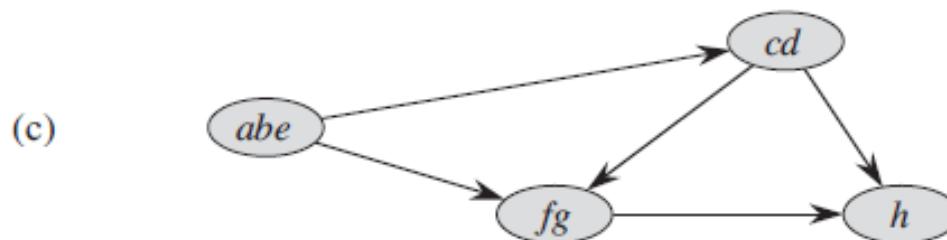
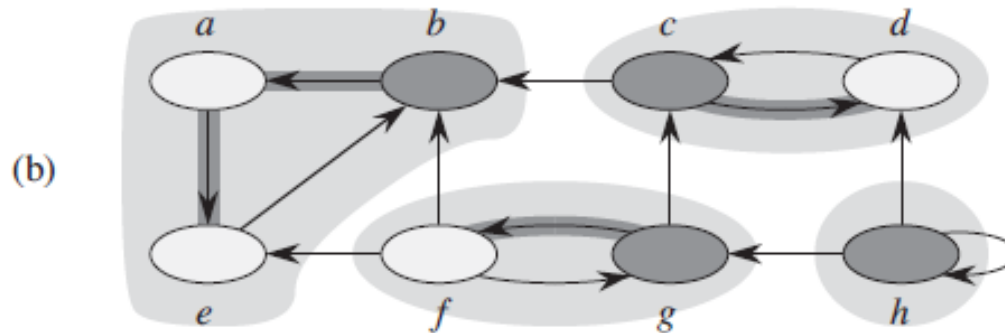
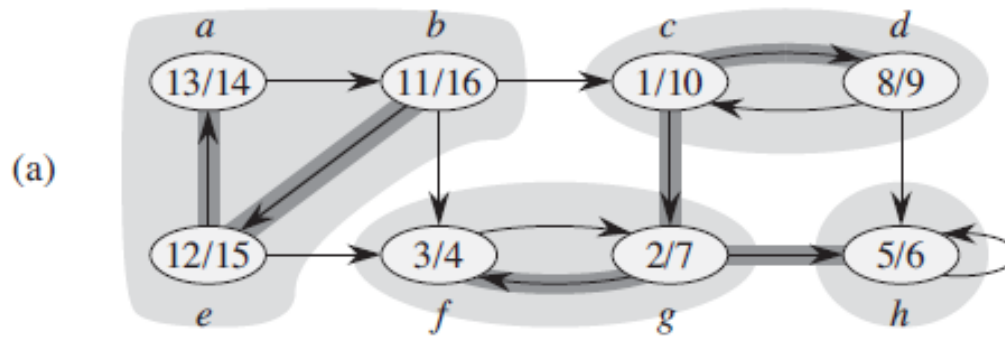
# STRONGLY CONNECTED COMPONENT

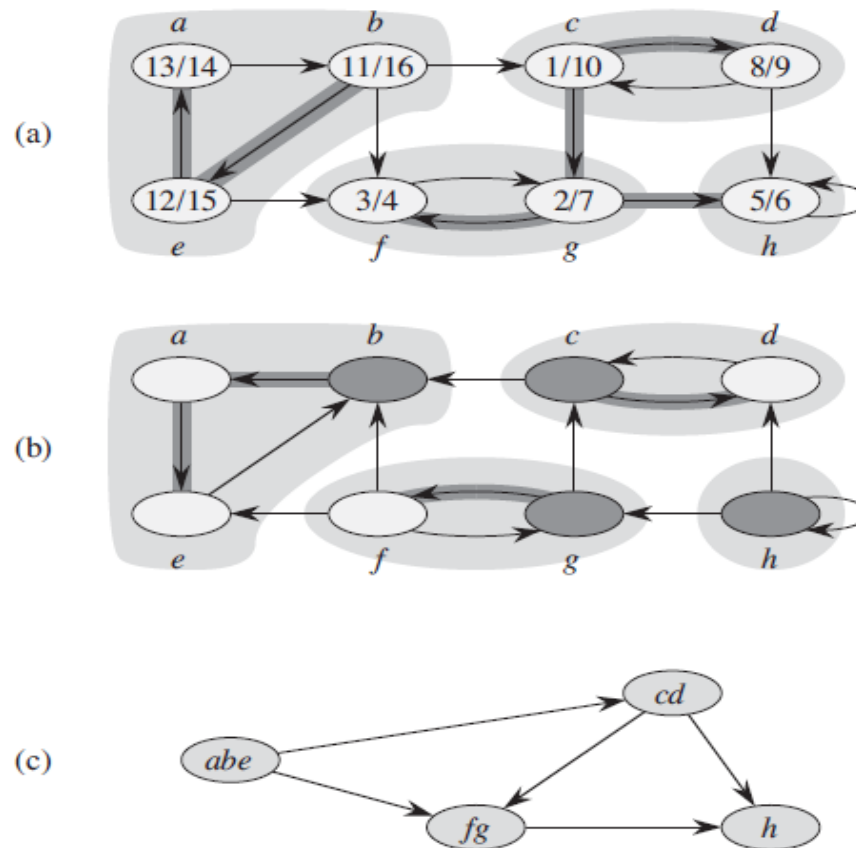
The following linear-time (i.e.,  $\Theta(V+E)$ ) time algorithm computes the strongly connected components of a directed graph  $G = \{V, E\}$  using two depth-first searches, one on  $G$  and one on  $G^T$

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

- 1 call DFS( $G$ ) to compute finishing times  $u.f$  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  $u.f$  (as computed in line 1)
- 4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

# STRONGLY CONNECTED COMPONENT





**Figure 22.9** (a) A directed graph  $G$ . Each shaded region is a strongly connected component of  $G$ . Each vertex is labeled with its discovery and finishing times in a depth-first search, and tree edges are shaded. (b) The graph  $G^T$ , the transpose of  $G$ , with the depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS shown and tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices  $b, c, g$ , and  $h$ , which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of  $G^T$ . (c) The acyclic component graph  $G^{SCC}$  obtained by contracting all edges within each strongly connected component of  $G$  so that only a single vertex remains in each component.

# REFERENCE

## ○ Introduction to Algorithms

- Chapter # 22
- Thomas H. Cormen