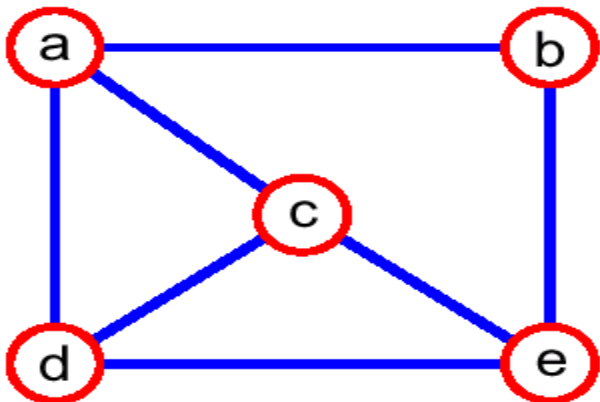


# Graphs – Definition

- A **graph**  $G = (V, E)$  is composed of:
  - $V$ : set of **vertices**
  - $E \subset V \times V$ : set of **edges** connecting the **vertices**
- An **edge**  $e = (u, v)$  is a pair of vertices
- $(u, v)$  is ordered, if  $G$  is a **directed** graph



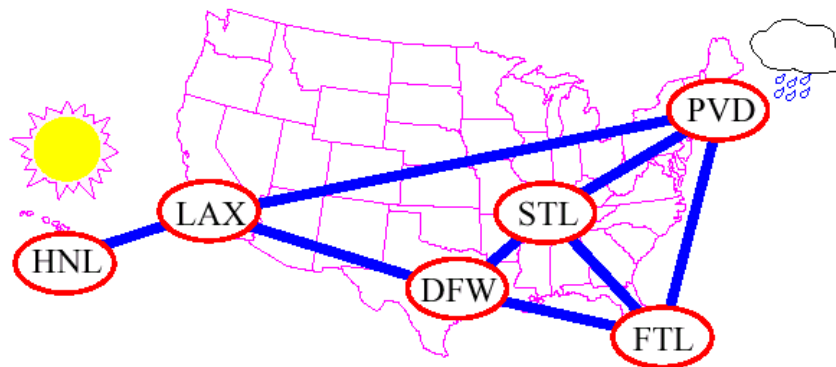
$V = \{a, b, c, d, e\}$

$E =$

$\{(a, b), (a, c), (a, d),$   
 $(b, e), (c, d), (c, e),$   
 $(d, e)\}$

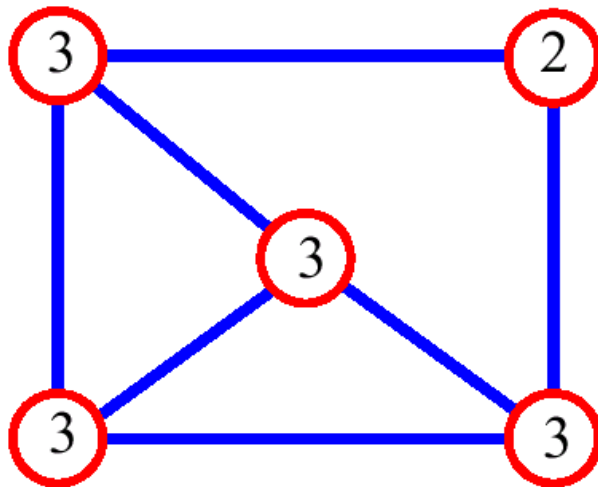
# Applications

- Electronic circuits, pipeline networks
- Transportation and communication networks
- Modeling any sort of relationships (between components, people, processes, concepts)



# Graph Terminology

- **adjacent vertices**: connected by an edge
- **degree** (of a **vertex**): # of adjacent vertices



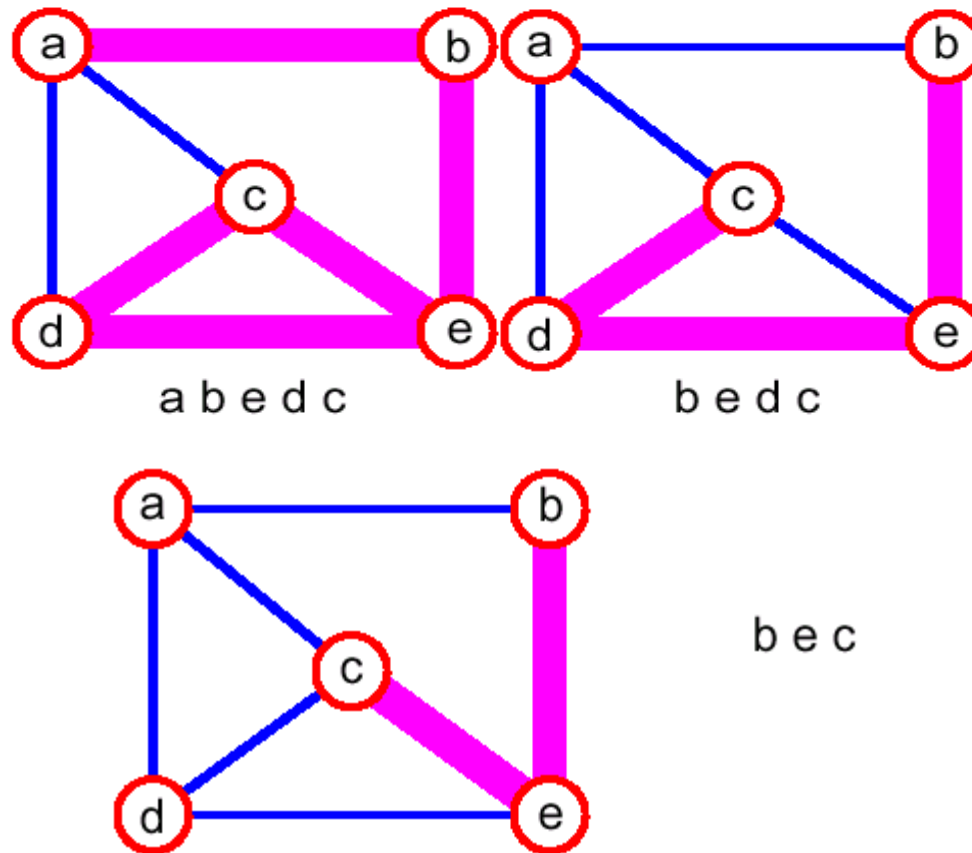
$$\sum_{v \in V} \deg(v) = 2(\# \text{ of edges})$$

Since adjacent vertices each count the adjoining edge, it will be counted twice

- **path**: sequence of vertices  $v_1, v_2, \dots, v_k$  such that consecutive vertices  $v_i$  and  $v_{i+1}$  are adjacent

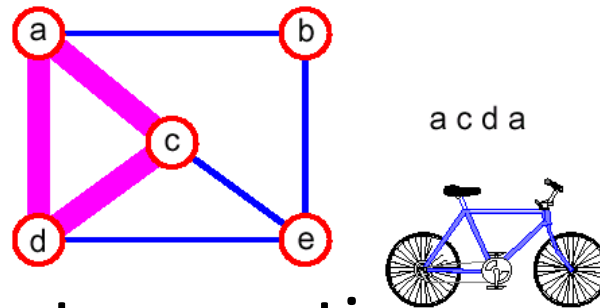
# Graph Terminology (2)

- **simple path:** no repeated vertices

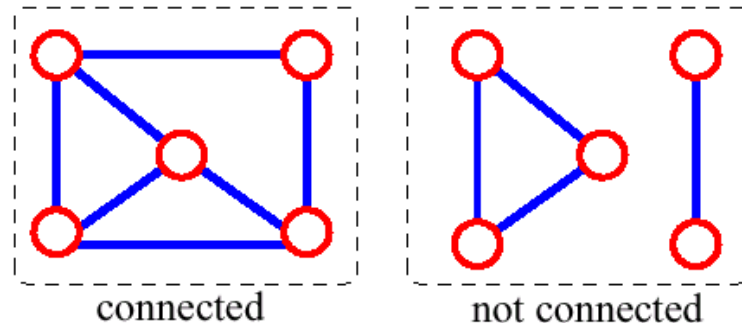


# Graph Terminology (3)

- **cycle**: simple path, except that the last vertex is the same as the first vertex

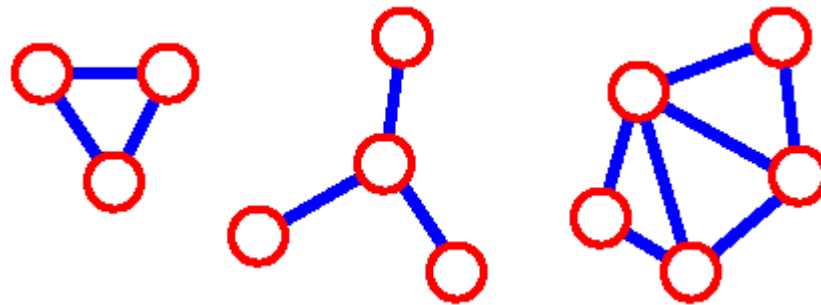


- **connected graph**: any two vertices are connected by some path



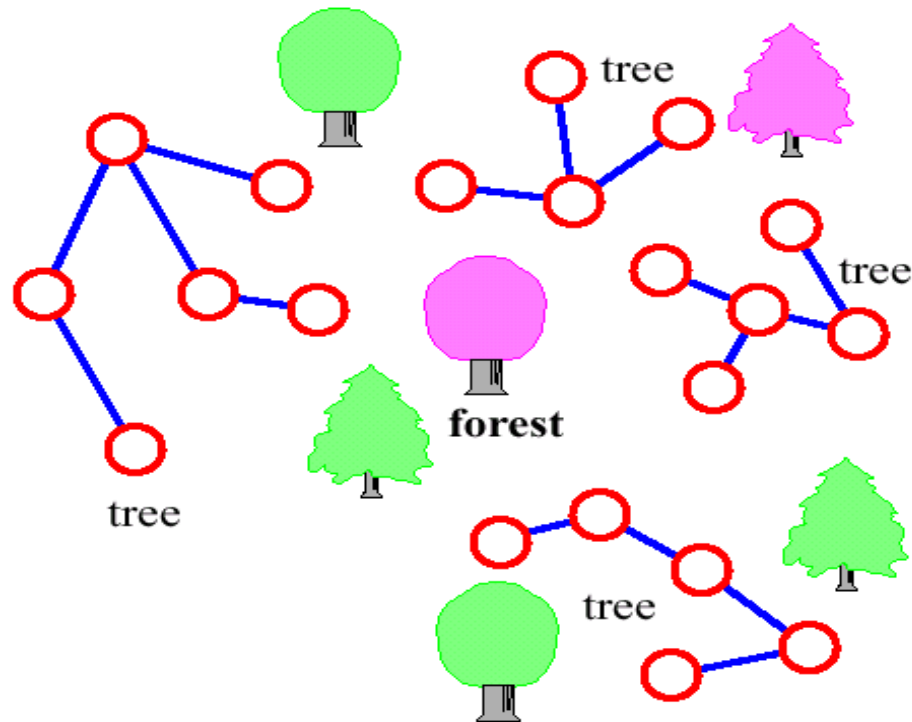
# Graph Terminology (4)

- **subgraph**: subset of vertices and edges forming a graph
- **connected component**: maximal connected subgraph. E.g., the graph below has 3 connected components



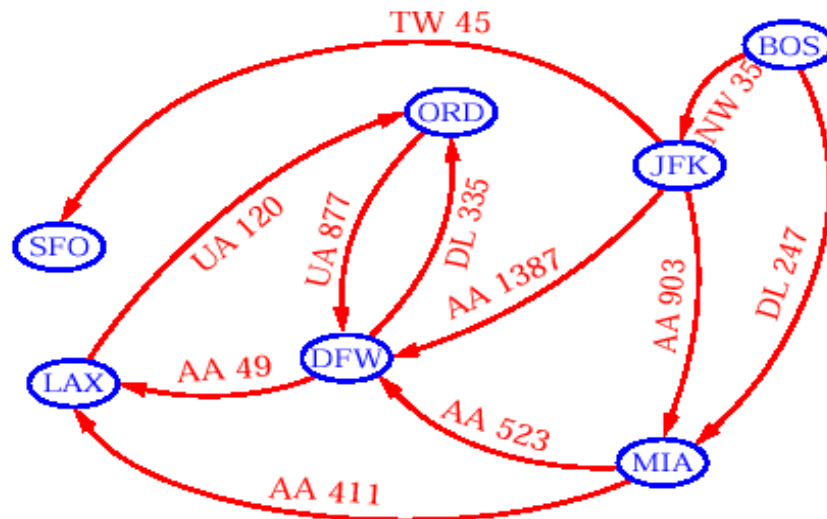
# Graph Terminology (5)

- (free) tree - connected graph without cycles
- forest - collection of trees



# Data Structures for Graphs

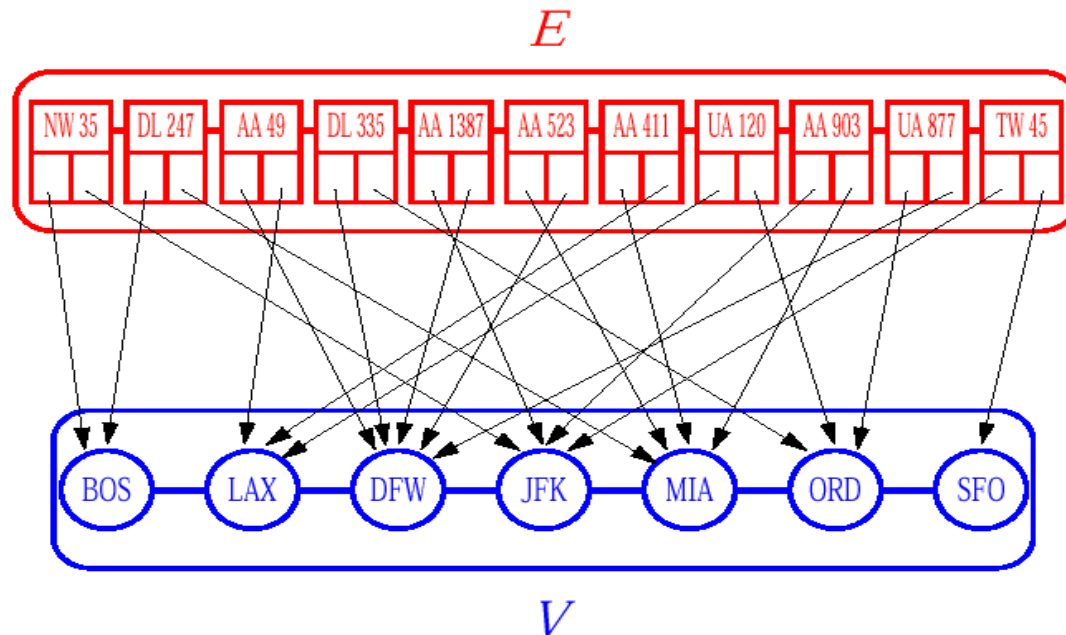
- How can we represent a graph?
  - To start with, we can store the vertices and the edges in two containers, and we store with each edge object references to its start and end vertices





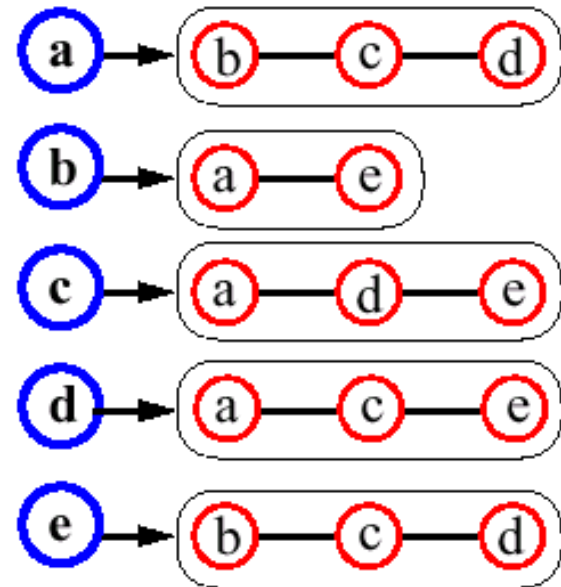
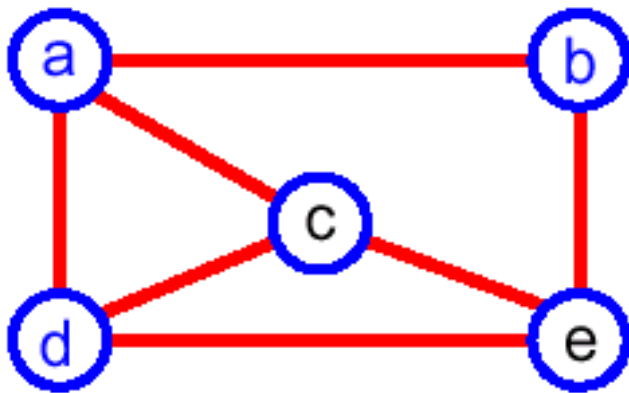
# Edge List

- The **edge list**
  - Easy to implement
  - Finding the edges incident on a given vertex is inefficient since it requires examining the entire edge sequence



# Adjacency List

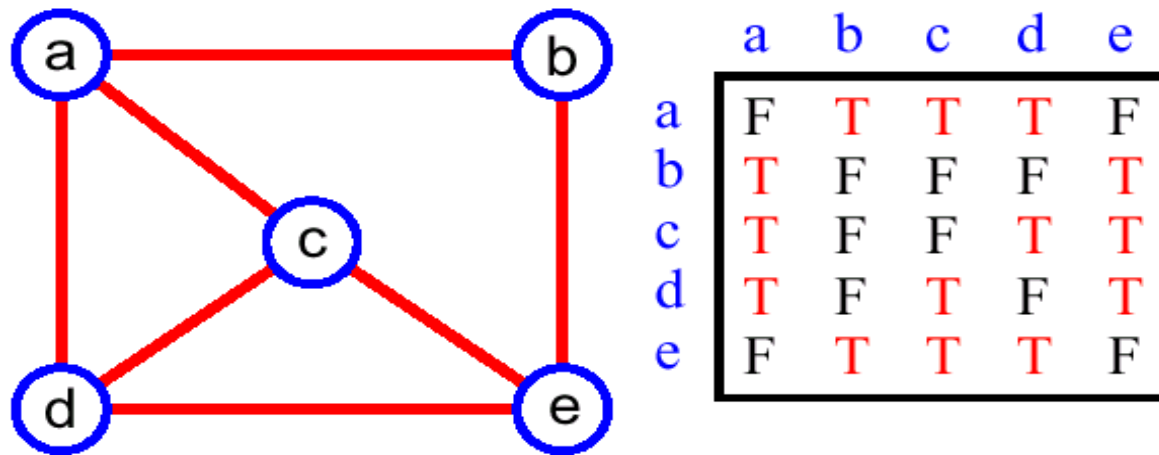
- The **Adjacency list** of a vertex  $v$ : a sequence of vertices adjacent to  $v$
- Represent the graph by the adjacency lists of all its vertices



$$\text{Space} = \Theta(n + \sum \deg(v)) = \Theta(n + m)$$

# Adjacency Matrix

- Matrix  $M$  with entries for all pairs of vertices
- $M[i,j] = \text{true}$  – there is an edge  $(i,j)$  in the graph
- $M[i,j] = \text{false}$  – there is no edge  $(i,j)$  in the graph
- Space =  $O(n^2)$



# Graph Searching Algorithms

- Systematic search of every edge and vertex of the graph
- Graph  $G = (V, E)$  is either directed or undirected
- Today's algorithms assume an adjacency list representation
- Applications
  - Compilers
  - Graphics
  - Maze-solving
  - Mapping
  - Networks: routing, searching, clustering, etc.

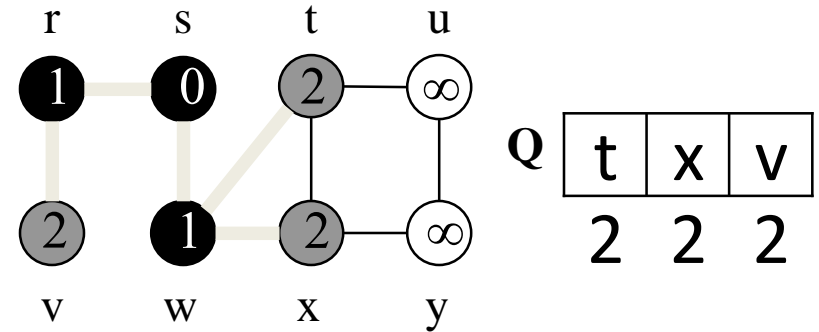
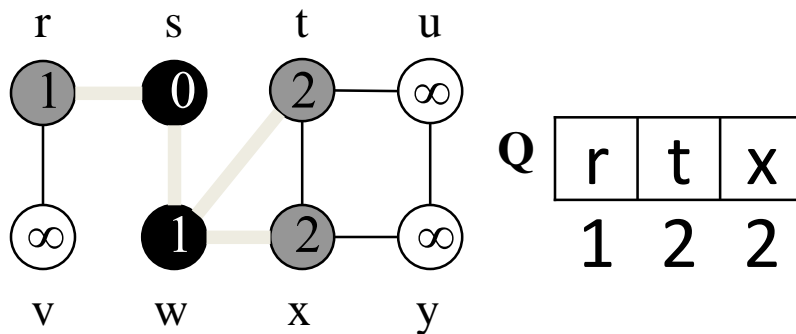
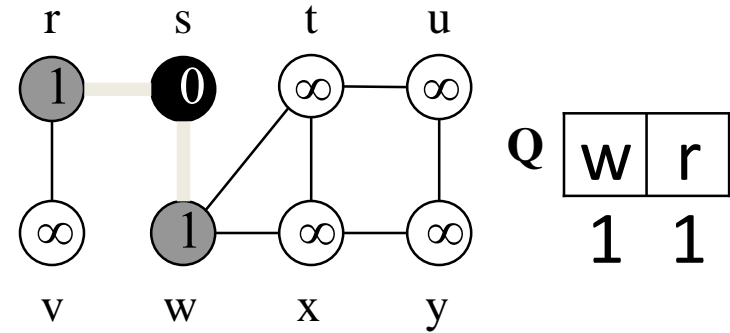
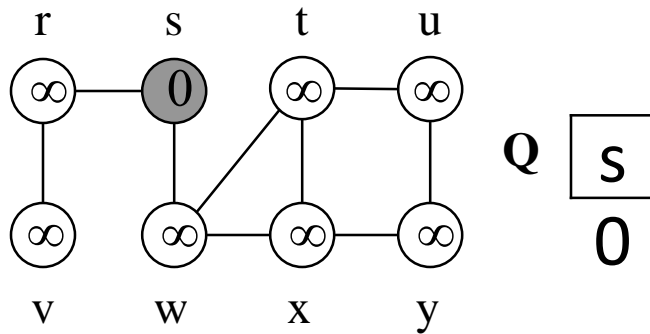
# Breadth First Search

- A **Breadth-First Search (BFS)** traverses a **connected component** of a graph, and in doing so defines a **spanning tree** with several useful properties
- BFS in an **undirected** graph  $G$  is like wandering in a labyrinth with a string.
- The starting vertex  $s$  is assigned a distance 0.
- In the first round, the string is unrolled the length of one edge, and all of the edges that are only one edge away from the anchor are visited (**discovered**), and assigned distances of 1

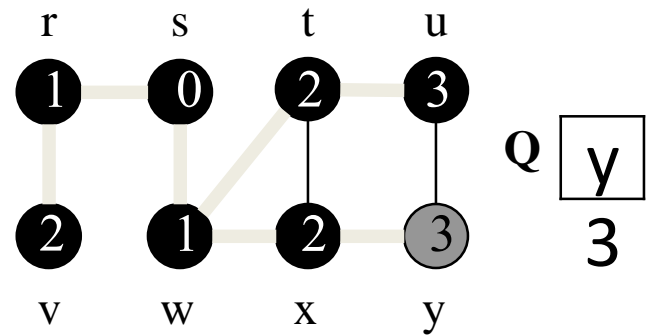
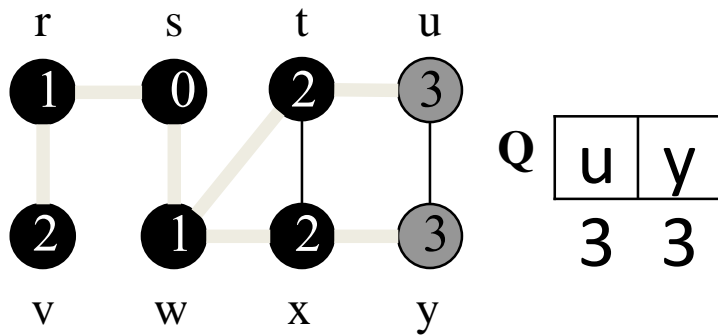
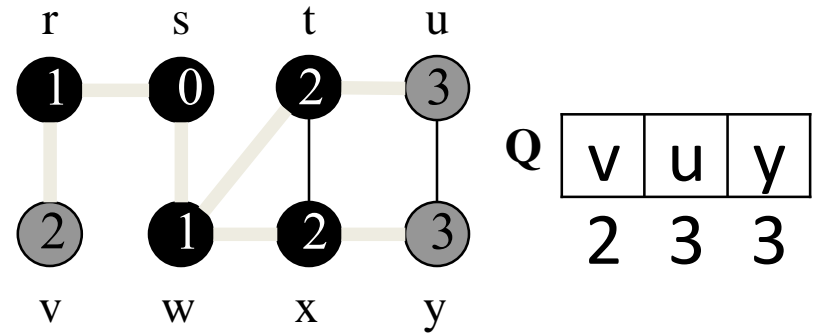
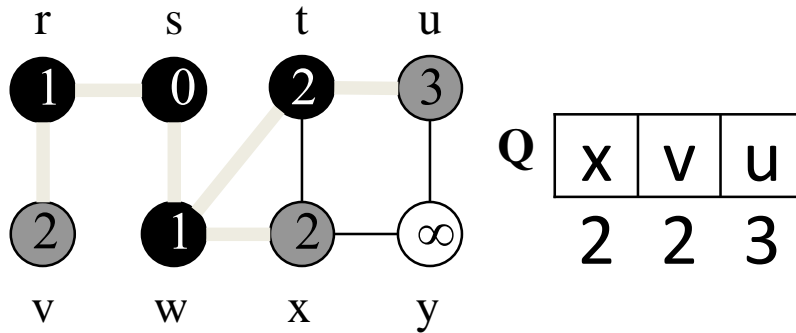
# Breadth-First Search (2)

- In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and assigned a distance of 2
- This continues until every vertex has been assigned a level
- The label of any vertex  $v$  corresponds to the length of the shortest path (in terms of edges) from  $s$  to  $v$

# BFS Example

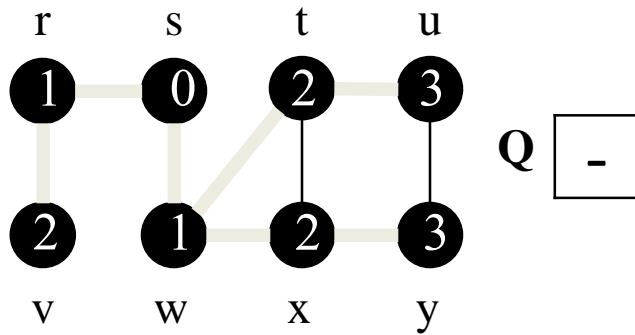


# BFS Example





# BFS Example: Result



# BFS Algorithm

**BFS** ( $G, s$ )

```
01 for each vertex  $u \in V[G] - \{s\}$ 
02      $\text{color}[u] \leftarrow \text{white}$ 
03      $d[u] \leftarrow \infty$ 
04      $\pi[u] \leftarrow \text{NIL}$ 
```

Init all  
vertices

```
05  $\text{color}[s] \leftarrow \text{gray}$ 
06  $d[s] \leftarrow 0$ 
07  $\pi[s] \leftarrow \text{NIL}$ 
08  $Q \leftarrow \{s\}$ 
```

Init BFS with  $s$

```
09 while  $Q \neq \emptyset$  do
10      $u \leftarrow \text{head}[Q]$ 
11     for each  $v \in \text{Adj}[u]$  do
12         if  $\text{color}[v] = \text{white}$  then
13              $\text{color}[v] \leftarrow \text{gray}$ 
14              $d[v] \leftarrow d[u] + 1$ 
15              $\pi[v] \leftarrow u$ 
16              $\text{Enqueue}(Q, v)$ 
```

Handle all  $u$ 's  
children before  
handling any  
children of  
children

```
17      $\text{Dequeue}(Q)$ 
18      $\text{color}[u] \leftarrow \text{black}$ 
```

# BFS Running Time

- Given a graph  $G = (V, E)$ 
  - Vertices are enqueued if their color is white
  - Assuming that en- and dequeuing takes  $O(1)$  time the total cost of this operation is  $O(V)$
  - Adjacency list of a vertex is scanned when the vertex is dequeued (and only then...)
  - The sum of the lengths of all lists is  $\Theta(E)$ . Consequently,  $O(E)$  time is spent on scanning them
  - Initializing the algorithm takes  $O(V)$
- **Total running time  $O(V+E)$**  (linear in the size of the adjacency list representation of  $G$ )

# BFS Properties

- Given a graph  $G = (V, E)$ , BFS **discovers all vertices reachable from a source vertex  $s$**
- It computes the **shortest distance** to all reachable vertices
- It computes a **breadth-first tree** that contains all such 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** in  $G$

# Breadth First Tree

- Predecessor subgraph of  $G$

$$G_{\pi} = (V_{\pi}, E_{\pi})$$

$$V_{\pi} = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

$$E_{\pi} = \{(\pi[v], v) \in E : v \in V_{\pi} - \{s\}\}$$

- $G_{\pi}$  is a breadth-first tree
  - $V_{\pi}$  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  $G$
- The edges in  $G_{\pi}$  are called tree edges

# Depth-First Search

- **A depth-first search (DFS)** in an undirected graph  $G$  is like wandering in a labyrinth with a **string** and a **can of paint**
  - We start at vertex  $s$ , tying the end of our string to the point and painting  $s$  “visited (discovered)”. Next we label  $s$  as our current vertex called  $u$
  - Now, we travel along an arbitrary edge  $(u,v)$ .
  - If edge  $(u,v)$  leads us to an already visited vertex  $v$  we return to  $u$
  - If vertex  $v$  is unvisited, we unroll our string, move to  $v$ , paint  $v$  “visited”, set  $v$  as our current vertex, and repeat the previous steps

# Depth-First Search (2)

- Eventually, we will get to a point where **all incident edges on  $u$  lead to visited vertices**
- We then **backtrack** by unrolling our string to a previously visited vertex  $v$ . Then  $v$  becomes our current vertex and we repeat the previous steps
- Then, if all incident edges on  $v$  lead to visited vertices, we backtrack as we did before. We **continue to backtrack along the path we have traveled**, finding and exploring unexplored edges, and repeating the procedure

# DFS Algorithm

- Initialize – color all vertices white
- Visit each and every white vertex using DFS-Visit
- Each call to DFS-Visit( $u$ ) roots a new tree of the depth-first forest at vertex  $u$
- A vertex is **white** if it is undiscovered
- A vertex is **gray** if it has been discovered but not all of its edges have been discovered
- A vertex is **black** after all of its adjacent vertices have been discovered (the adj. list was examined completely)



# DFS Algorithm (2)

DFS( $G$ )

```
1 for each vertex  $u \in V[G]$ 
2   do  $color[u] \leftarrow \text{WHITE}$ 
3  $time \leftarrow 0$ 
```

Init all  
vertices

```
4 for each vertex  $u \in V[G]$ 
5   do if  $color[u] = \text{WHITE}$ 
6     then DFS-VISIT( $u$ )
```

DFS-VISIT( $u$ )

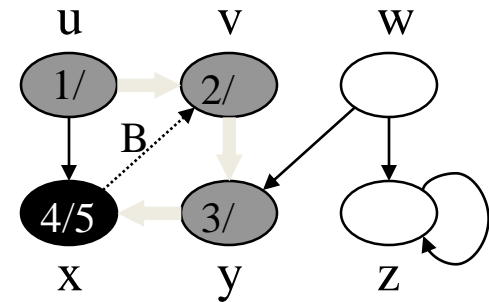
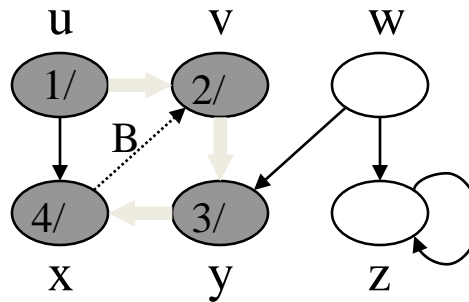
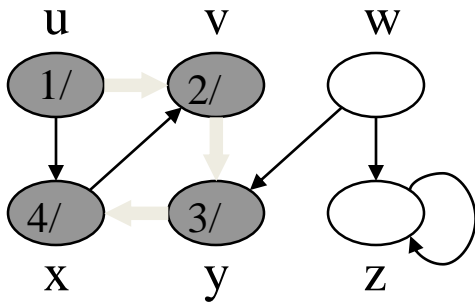
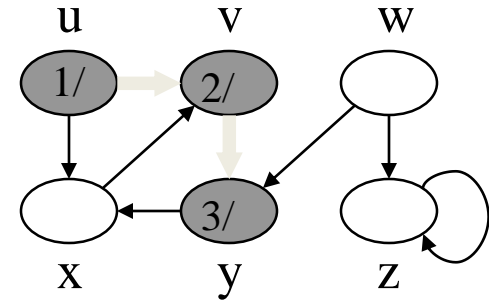
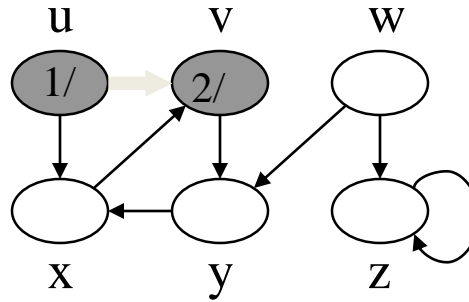
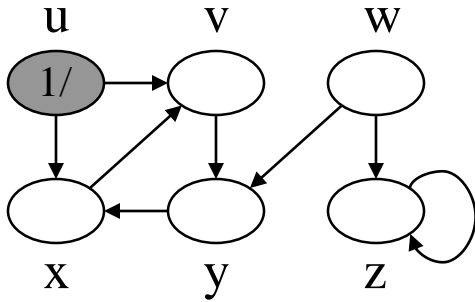
```
1  $color[u] \leftarrow \text{GRAY}$            ▷ White vertex  $u$  discovered.
2  $d[u] \leftarrow time$              ▷ Mark with discovery time.
3  $time \leftarrow time + 1$          ▷ Tick global time.
```

```
4 for each  $v \in Adj[u]$            ▷ Explore all edges  $(u, v)$ .
5   do if  $color[v] = \text{WHITE}$ 
6     then DFS-VISIT( $v$ )
```

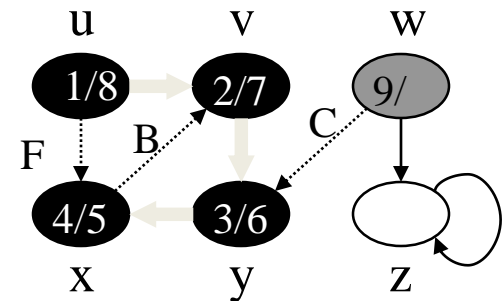
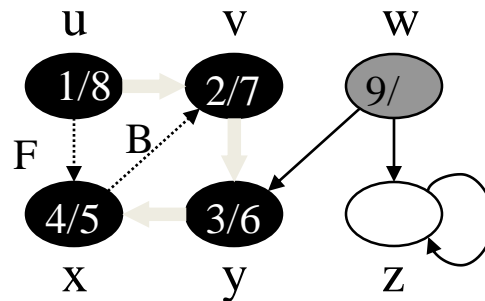
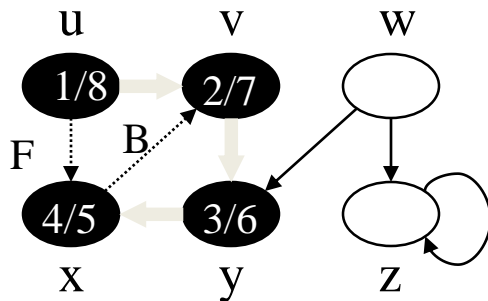
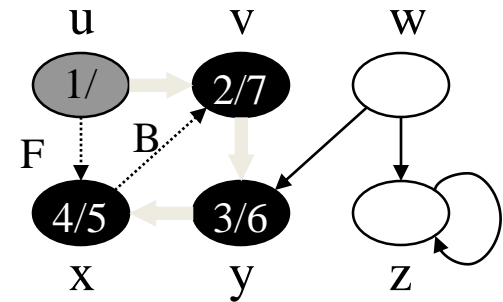
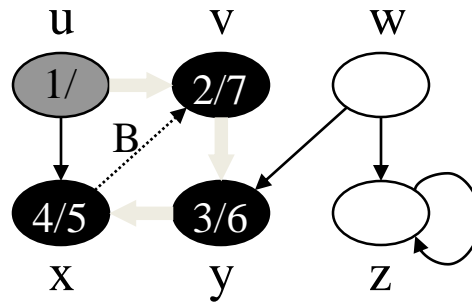
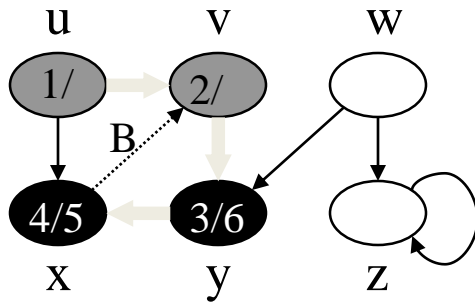
Visit all children  
recursively

```
7  $color[u] \leftarrow \text{BLACK}$        ▷ Blacken  $u$ ; it is finished.
8  $f[u] \leftarrow time$              ▷ Mark with finishing time.
9  $time \leftarrow time + 1$          ▷ Tick global time.
```

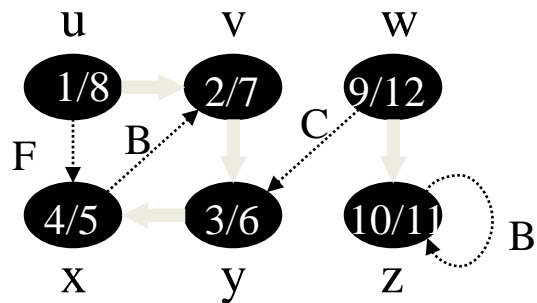
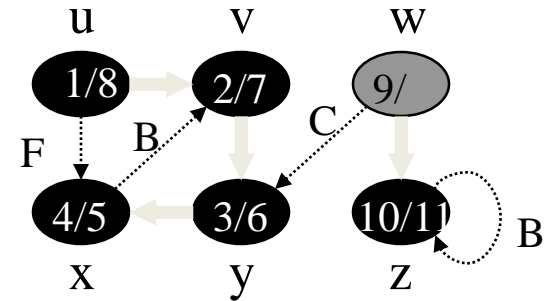
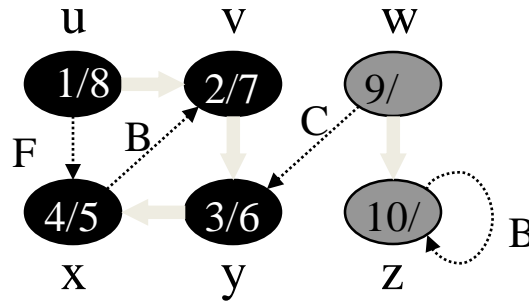
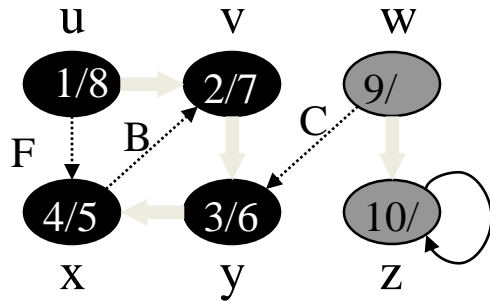
# DFS Example



# DFS Example (2)



# DFS Example (3)













# DFS Algorithm (3)

- When DFS returns, every vertex  $u$  is assigned
  - a discovery time  $d[u]$ , and a finishing time  $f[u]$
- Running time
  - the loops in DFS take time  $\Theta(V)$  each, excluding the time to execute DFS-Visit
  - DFS-Visit is called once for every vertex
    - its only invoked on white vertices, and
    - paints the vertex gray immediately
  - for each DFS-visit a loop iterates over all  $Adj[v]$
  - the total cost for DFS-Visit is  $\Theta(E)$ 

$\sum_{v \in V} |Adj[v]| = \Theta(E)$
  - **the running time of DFS is  $\Theta(V+E)$**

# Predecessor Subgraph

- Define slightly different from BFS

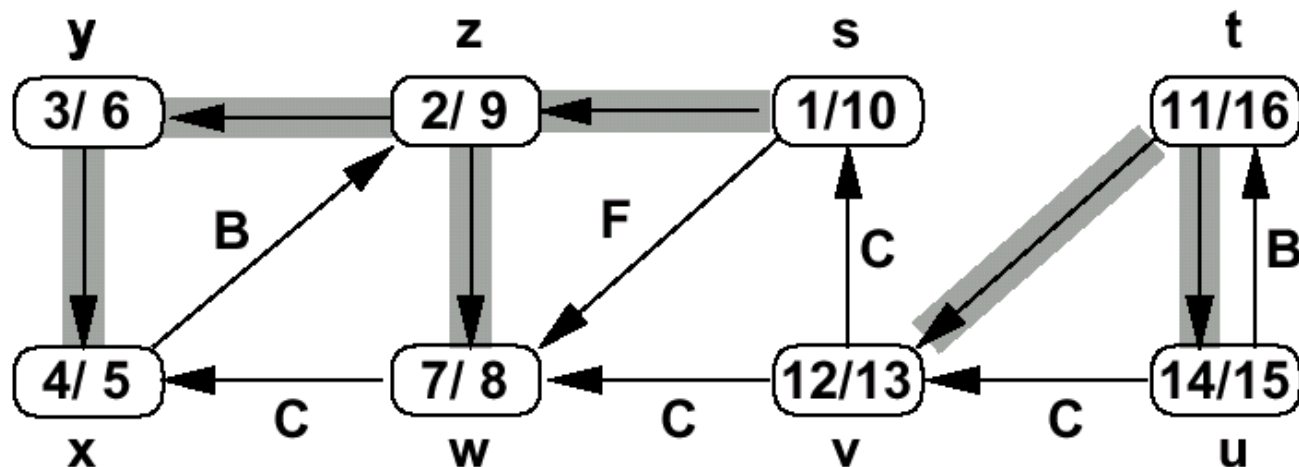
$$G_{\pi} = (V, E_{\pi})$$

$$E_{\pi} = \{(\pi[v], v) \in E : v \in V \text{ and } \pi[v] \neq \text{NIL}\}$$

- The PD subgraph of a depth-first search forms a **depth-first forest** composed of several depth-first trees
- The edges in  $G_{\pi}$  are called tree edges

# DFS Timestamping

- The DFS algorithm maintains a monotonically increasing global clock
  - discovery time  $d[u]$  and finishing time  $f[u]$
- For every vertex  $u$ , the inequality  $d[u] < f[u]$  must hold



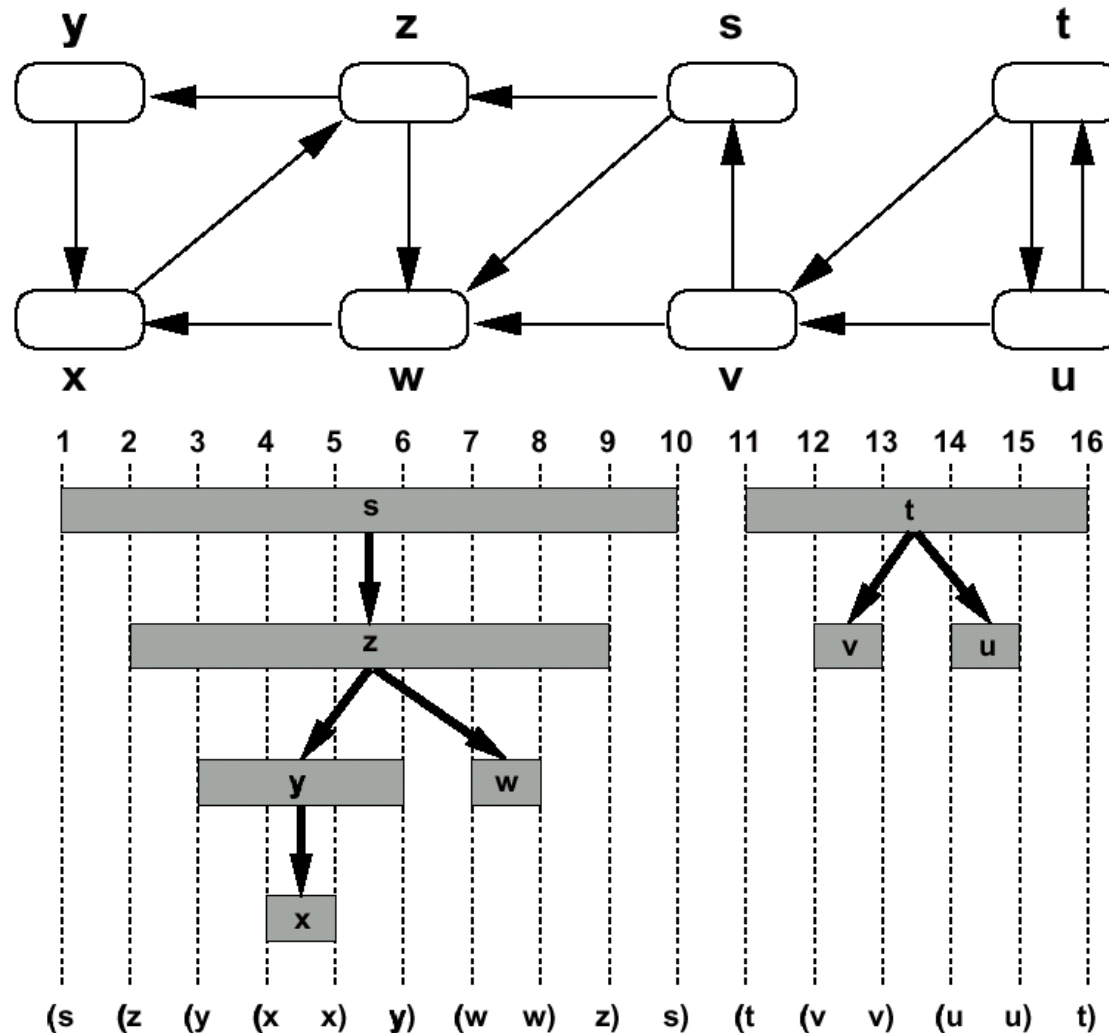
# DFS Timestamping

- Vertex  $u$  is
  - white before time  $d[u]$
  - gray between time  $d[u]$  and time  $f[u]$ , and
  - black thereafter
- Notice the structure throughout the algorithm.
  - gray vertices form a linear chain
  - corresponds to a stack of vertices that have not been exhaustively explored (DFS-Visit started but not yet finished)

# DFS Parenthesis Theorem

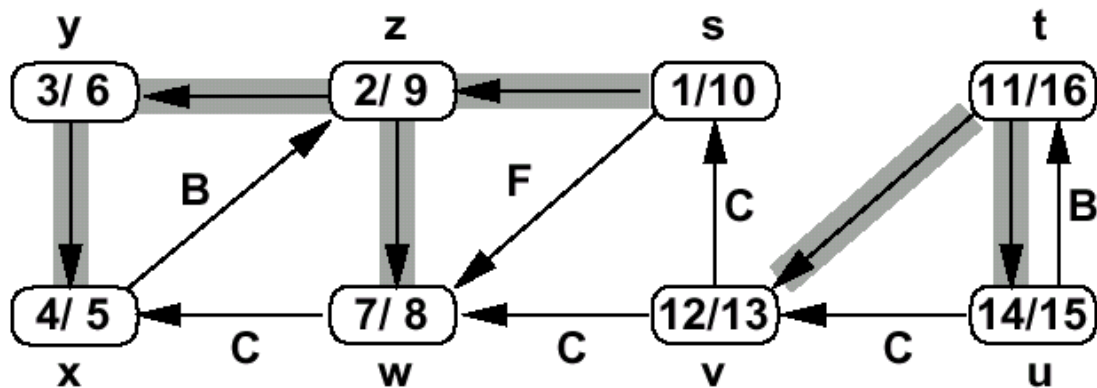
- Discovery and finish times have parenthesis structure
  - represent discovery of  $u$  with left parenthesis " $(u$ "
  - represent finishing of  $u$  with right parenthesis " $u)$ "
  - history of discoveries and finishings makes a well-formed expression (parenthesis are properly nested)
- Intuition for proof: any two intervals are either disjoint or enclosed
  - Overlapping intervals would mean finishing ancestor, before finishing descendant or starting descendant without starting ancestor

# DFS Parenthesis Theorem (2)



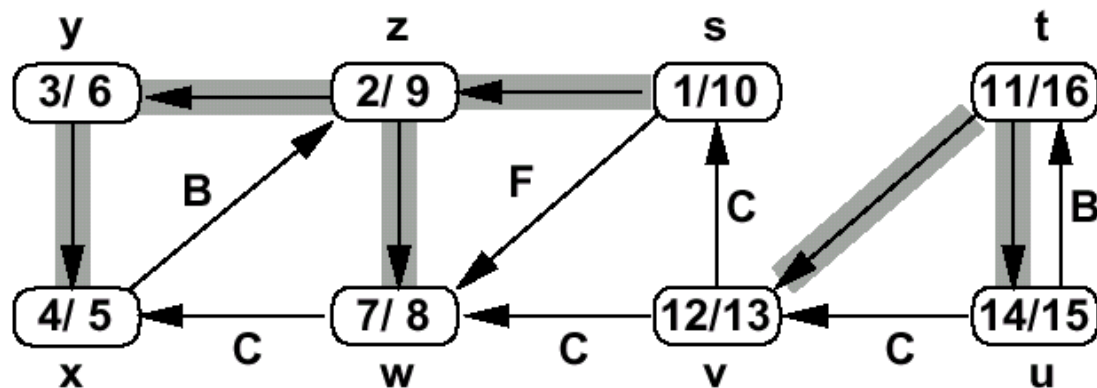
# DFS Edge Classification

- Tree edge (gray to white)
  - encounter new vertices (white)
- Back edge (gray to gray)
  - from descendant to ancestor



# DFS Edge Classification (2)

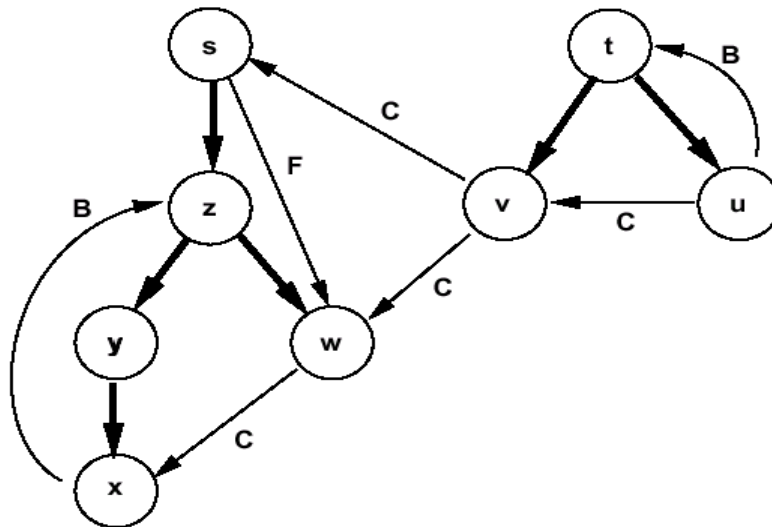
- Forward edge (gray to black)
  - from ancestor to descendant
- Cross edge (gray to black)
  - remainder – between trees or subtrees





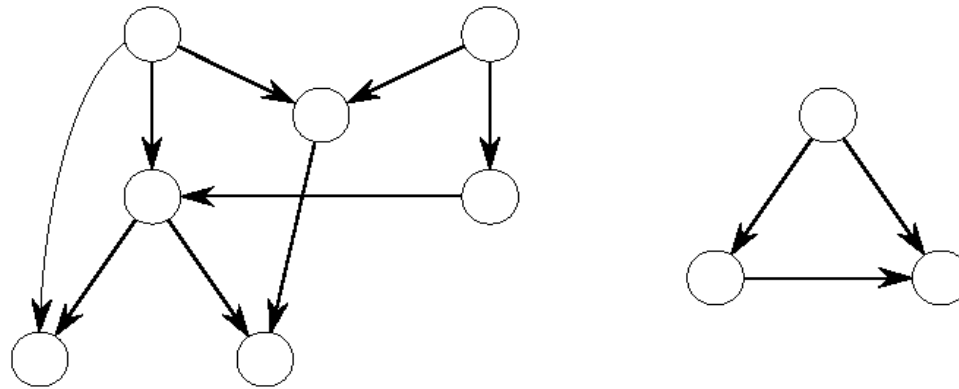
# DFS Edge Classification (3)

- Tree and back edges are important
- Most algorithms do not distinguish between forward and cross edges



# Directed Acyclic Graphs

- A DAG is a directed graph with no cycles



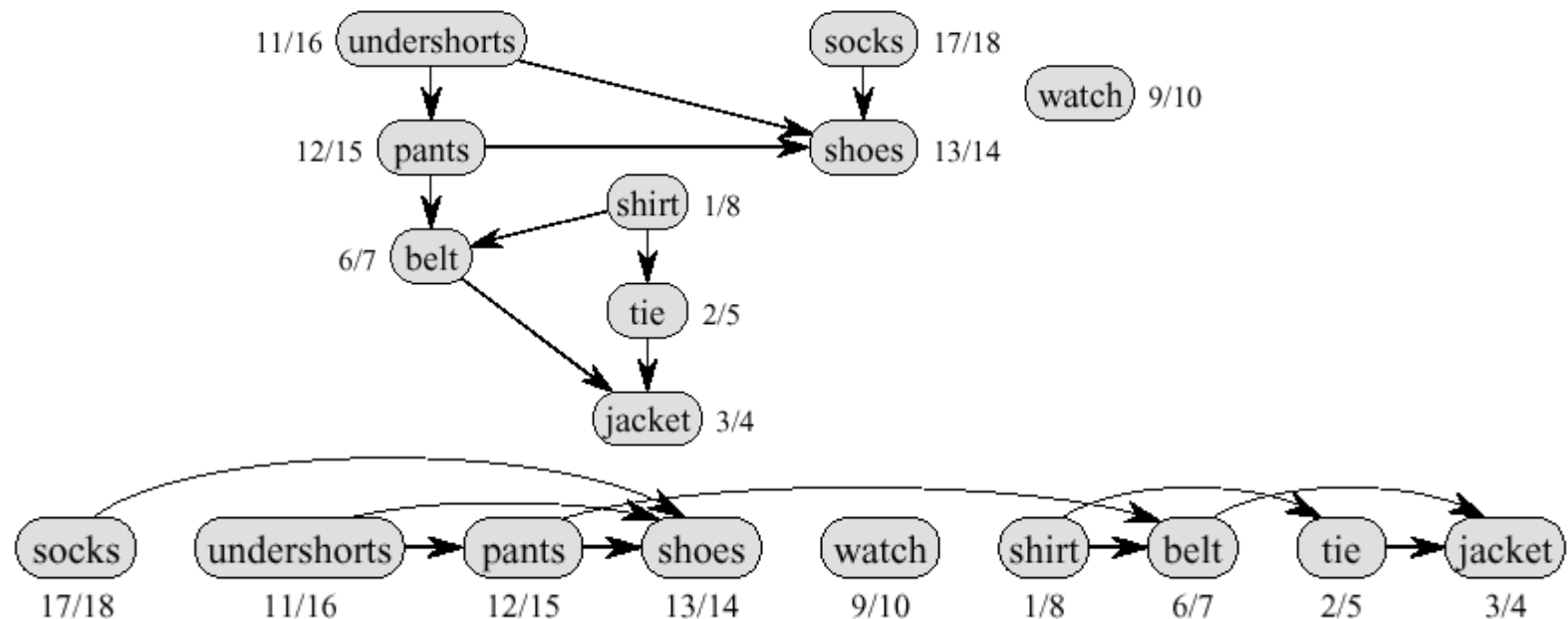
- Often used to indicate precedences among events, i.e., event  $a$  must happen before  $b$
- An example would be a parallel code execution
- Total order can be introduced using **Topological Sorting**

# DAG Theorem

- A directed graph  $G$  is acyclic if and only if a DFS of  $G$  yields no back edges. Proof:
  - **suppose there is a back edge  $(u,v)$** ;  $v$  is an ancestor of  $u$  in DFS forest. Thus, there is a path from  $v$  to  $u$  in  $G$  and  $(u,v)$  completes the cycle
  - **suppose there is a cycle  $c$** ; let  $v$  be the first vertex in  $c$  to be discovered and  $u$  is a predecessor of  $v$  in  $c$ .
    - Upon discovering  $v$  the whole cycle from  $v$  to  $u$  is white
    - We must visit all nodes reachable on this white path before return DFS-Visit( $v$ ), i.e., vertex  $u$  becomes a descendant of  $v$
    - Thus,  $(u,v)$  is a back edge
- Thus, we can verify a DAG using DFS!

# Topological Sort Example

- Precedence relations: an edge from  $x$  to  $y$  means one must be done with  $x$  before one can do  $y$
- Intuition: can schedule task only when all of its subtasks have been scheduled



# Topological Sort

- Sorting of a directed acyclic graph (DAG)
- A topological sort of a DAG is a linear ordering of all its vertices such that for any edge  $(u,v)$  in the DAG,  $u$  appears before  $v$  in the ordering
- The following algorithm topologically sorts a DAG

## **Topological-Sort(G)**

- 1) call 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
- The linked lists comprises a total ordering

# Topological Sort

- Running time
  - depth-first search:  $O(V+E)$  time
  - insert each of the  $|V|$  vertices to the front of the linked list:  $O(1)$  per insertion
- Thus the total running time is  $O(V+E)$

# Topological Sort Correctness

- Claim: for a DAG, an edge  $(u, v) \in E \Rightarrow f[u] > f[v]$
- When  $(u, v)$  explored,  $u$  is gray. We can distinguish three cases
  - $v = \text{gray}$ 
    - $\Rightarrow (u, v) = \text{back edge (cycle, contradiction)}$
  - $v = \text{white}$ 
    - $\Rightarrow v$  becomes descendant of  $u$
    - $\Rightarrow v$  will be finished before  $u$
    - $\Rightarrow f[v] < f[u]$
  - $v = \text{black}$ 
    - $\Rightarrow v$  is already finished
    - $\Rightarrow f[v] < f[u]$
- The definition of topological sort is satisfied