Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

Lecture 23

Searching in a Graph

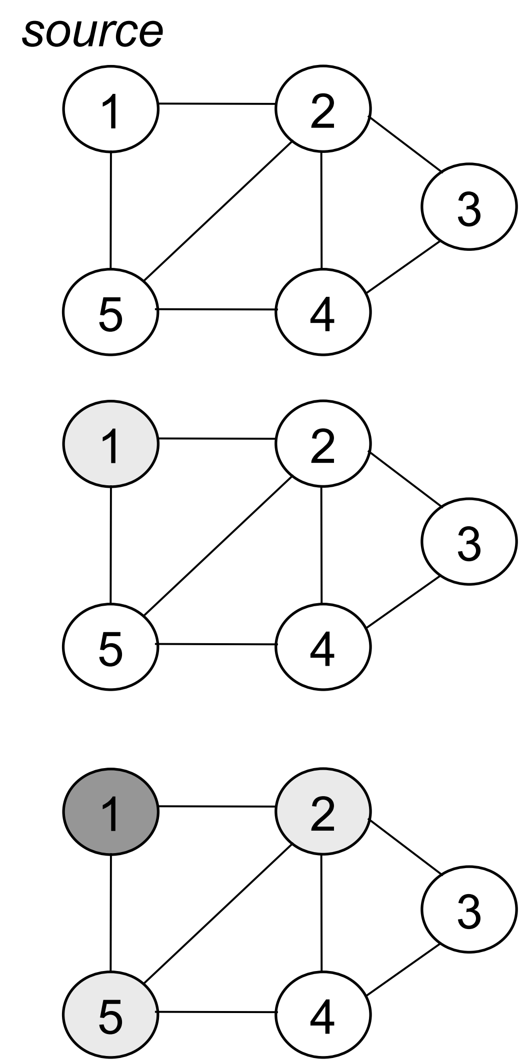
- **Graph searching** = systematically follow the edges of the graph so as to visit the vertices of the graph
- Two basic graph searching algorithms:
  - Breadth-first search
  - Depth-first search
- The difference between them is in the order in which they explore the unvisited edges of the graph
- Graph algorithms are typically elaborations of the basic graph-searching algorithms

Breadth-First Search (BFS)

- **Input**:
  - A graph $G = (V, E)$ (directed or undirected)
  - A **source** vertex s from V
- **Goal**:
  - Explore the edges of G to "discover" every vertex reachable from s, taking the ones closest to s first
- **Output**:
  - d[v] = distance (smallest # of edges) from s to v, for all v from V
  - A "breadth-first tree" rooted at s that contains all reachable vertices

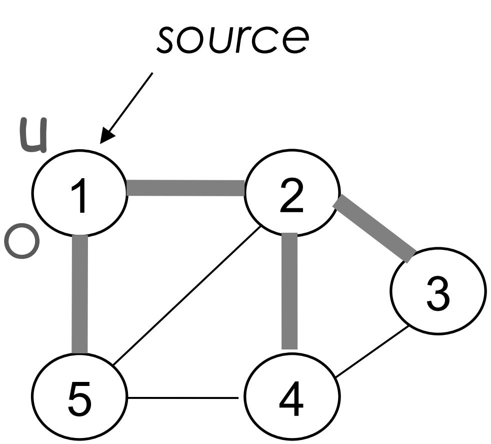
Breadth-First Search (cont.)

- Keeping track of progress:
  - Color each vertex in either **white**, **gray** or **black**
  - Initially, all vertices are **white**
  - When being discovered a vertex becomes **gray**
  - After discovering all its adjacent vertices the node becomes **black**
  - Use FIFO queue Q to maintain the set of gray vertices



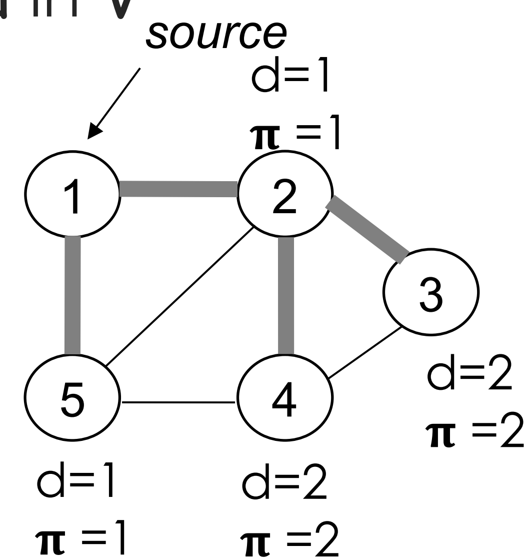
Breadth-First Tree

- BFS constructs a breadth-first tree
  - Initially contains the root (source vertex ***s***)
  - When vertex **v** is discovered while scanning the adjacency list of a vertex **u** ⇒ vertex **v** and edge **(u, v)** are added to the tree
  - **u** is the **predecessor** (**parent**) of **v** in the breadth-first tree
  - A vertex is discovered only once ⇒ it has only one parent



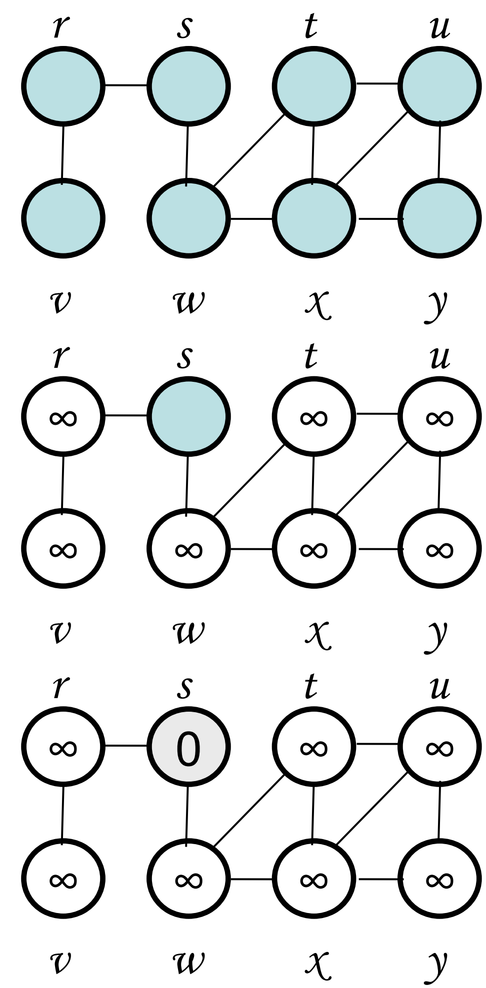
BFS Additional Data Structures

- G = (V, E) represented using adjacency lists
- **color[u]** – the color of the vertex for all **u** in **V**
- **π[u]** – predecessor of **u**
  - If **u** = **s** (root) or node **u** has not yet been discovered then [u] = π NIL
- **d[u]** – the distance from the source **s** to vertex **u**
- Use a FIFO queue Q to maintain the set of gray vertices



BFS(V, E, s)

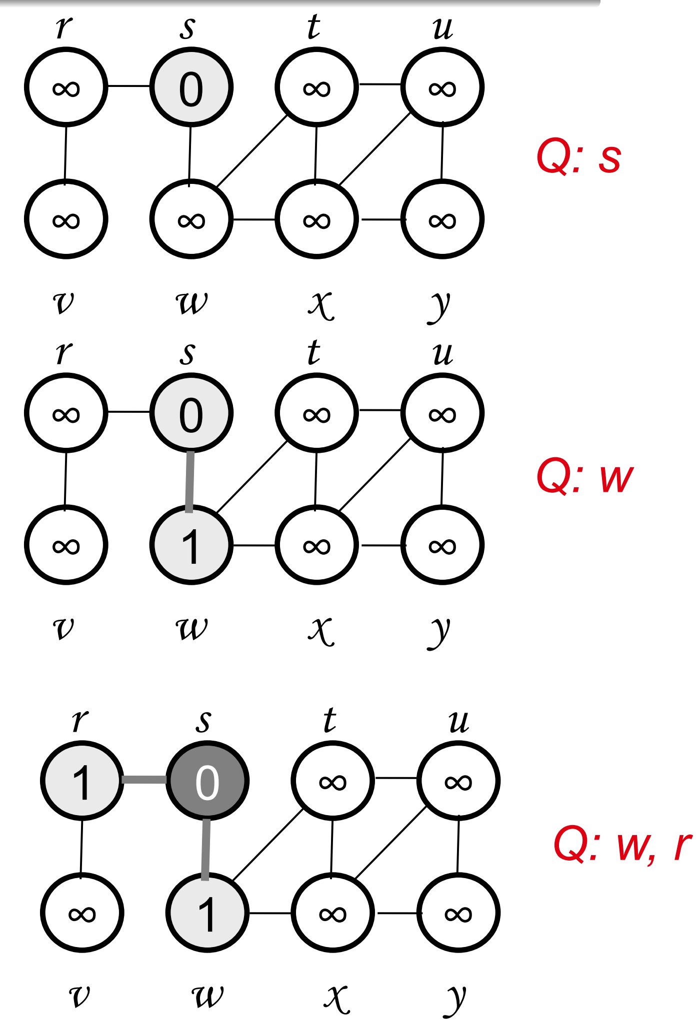
1. **for** each u in V - {s}
2. **do** color[u] = WHITE
3. d[u] ← ∞
4. π[u] = NIL
5. color[s] = GRAY
6. d[s] ← 0
7. π[s] = NIL
8. Q = empty
9. Q ← ENQUEUE(Q, s)



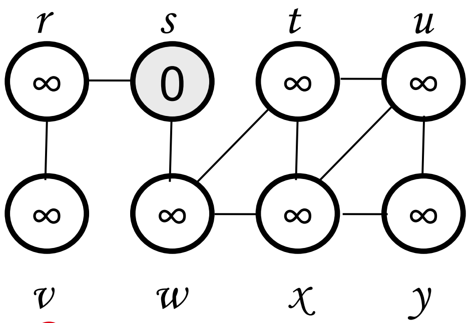
Q: *s*

BFS(V, E, s)

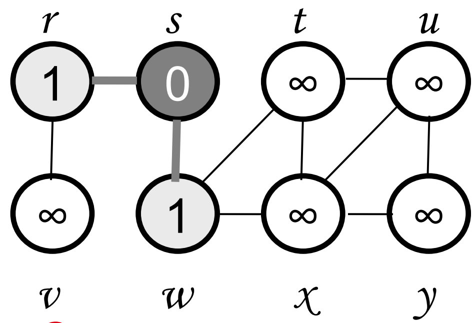
```
while Q not empty
    do u ← DEQUEUE(Q)
        for each v in Adj[u]
            do if color[v] = WHITE
                then color[v] = GRAY
                    d[v] ← d[u] + 1
                    π[v] = u
                    ENQUEUE(Q, v)
    color[u] = BLACK
```



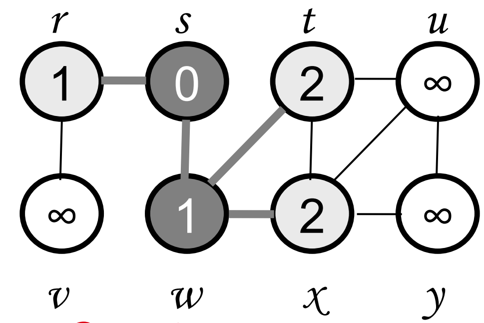
Example



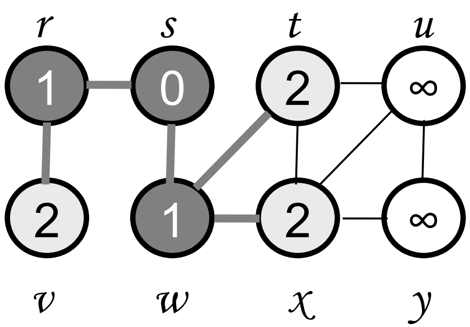
Q: *s*



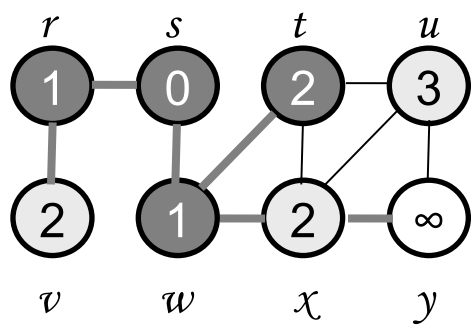
Q: *w, r*



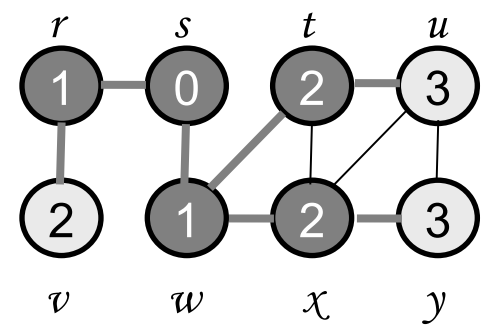
Q: *r, t, x*



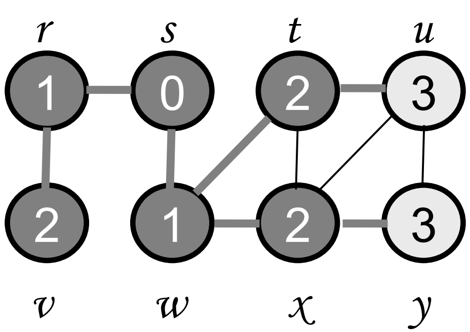
Q: *t, x, v*



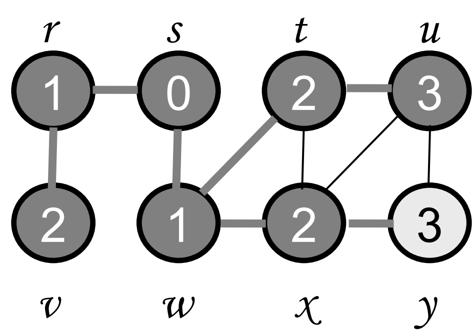
Q: *x, v, u*



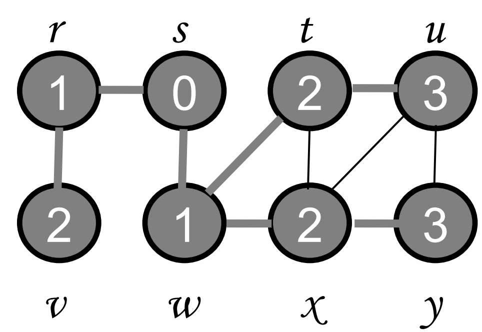
Q: *v, u, y*



Q: *u, y*



Q: *y*



Q: ∅

Analysis of BFS

for each u ∈ V - {s}
    do color[u] ← WHITE
        d[u] ← ∞
        π[u] = NIL

$O(|V|)$

color[s] ← GRAY
d[s] ← 0
π[s] = NIL
Q ← ∅
Q ← ENQUEUE(Q, s)

$\Theta(1)$

Analysis of BFS

```
while Q not empty
    do u ← DEQUEUE(Q)
        for each v in Adj[u]
            do if color[v] = WHITE
                then color[v] = GRAY
                    d[v] ← d[u] + 1
                    π[v] = u
                    ENQUEUE(Q, v)
    color[u] = BLACK
```

$\Theta(1)$ (DEQUEUE)

$\Theta(1)$ (ENQUEUE)

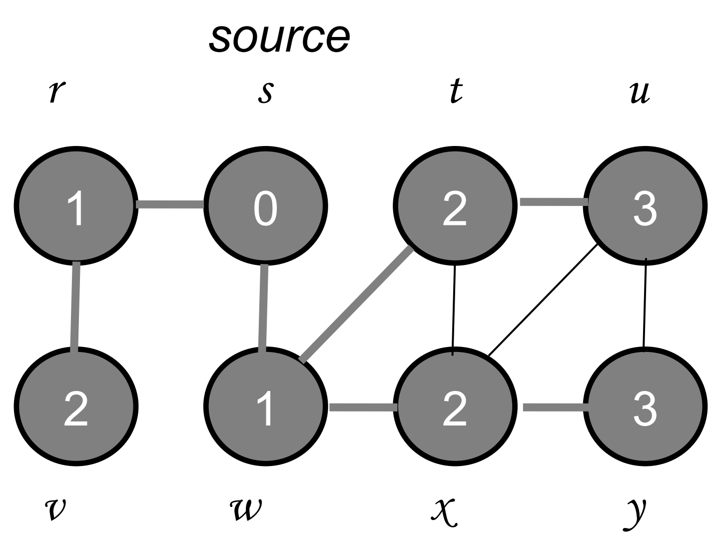
Scan Adj[u] for all vertices u in the graph

- Each vertex u is processed only once, when the vertex is dequeued
  - Sum of lengths of all adjacency lists = $\Theta(|E|)$
  - Scanning operations: $O(|E|)$

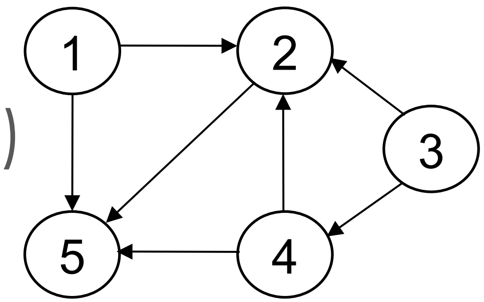
- Total running time for BFS = $O(|V| + |E|)$

Shortest Paths Property

- BFS finds the shortest-path distance from the source vertex $s \in V$ to each node in the graph
- Shortest-path distance = $\delta(s, u)$
  - Minimum number of edges in any path from s to u



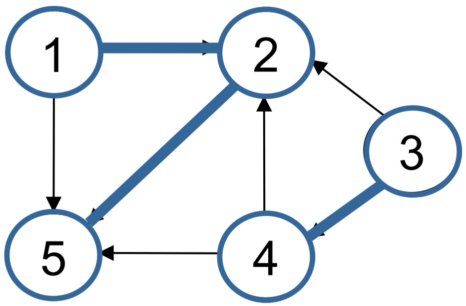
Depth-First Search



- **Input:**
  - G = (V, E) (No source vertex given!)
- **Goal**:
  - Explore the edges of G to "discover" every vertex in V starting at the most current visited node
  - Search may be repeated from multiple sources
- **Output:**
  - 2 **timestamps** on each vertex:
    - d[v] = discovery time
    - f[v] = finishing time (done with examining **v**'s adjacency list)
  - Depth-first forest

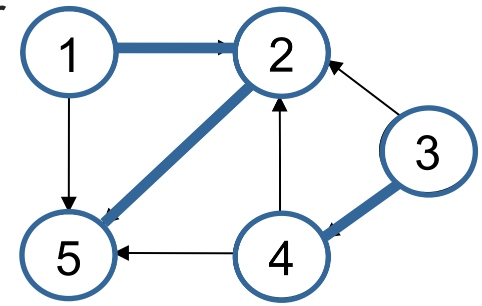
Depth-First Search

- Search "deeper" in the graph whenever possible



- Edges are explored out of the most recently discovered vertex **v** that still has unexplored edges
- After all edges of v have been explored, the search "backtracks" from the parent of v
- The process continues until all vertices reachable from the original source have been discovered
- If undiscovered vertices remain, choose one of them as a new source and repeat the search from that vertex
- DFS creates a "depth-first forest"

Depth-First Search



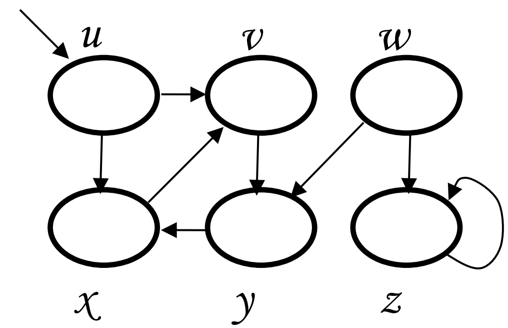
- Search "deeper" in the graph whenever possible
- Edges are explored out of the most recently discovered vertex **v** that still has unexplored edges
- After all edges of v have been explored, the search "backtracks" from the parent of v
- The process continues until all vertices reachable from the original source have been discovered
- If undiscovered vertices remain, choose one of them as a new source and repeat the search from that vertex
- DFS creates a "depth-first forest"

DFS Additional Data Structures

- Global variable: time-step
  - Incremented when nodes are discovered/finished
- **color[u]** – similar to BFS
  - White before discovery, gray while processing and black when finished processing
- **π[u]** – predecessor of **u**
- **d[u]**, **f[u]** – discovery and finish times

$$1 \le d[u] < f[u] \le 2|V|$$

DFS(V, E)

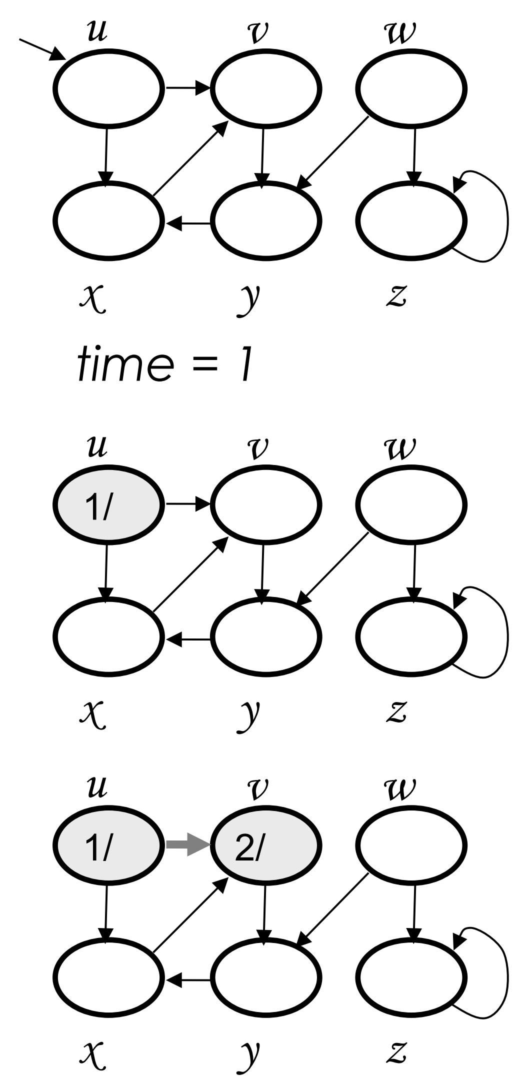



```
for each u ∈ V
    do color[u] ← WHITE
        π[u] ← NIL
time ← 0
for each u ∈ V
    do if color[u] = WHITE
        then DFS-VISIT(u)
```

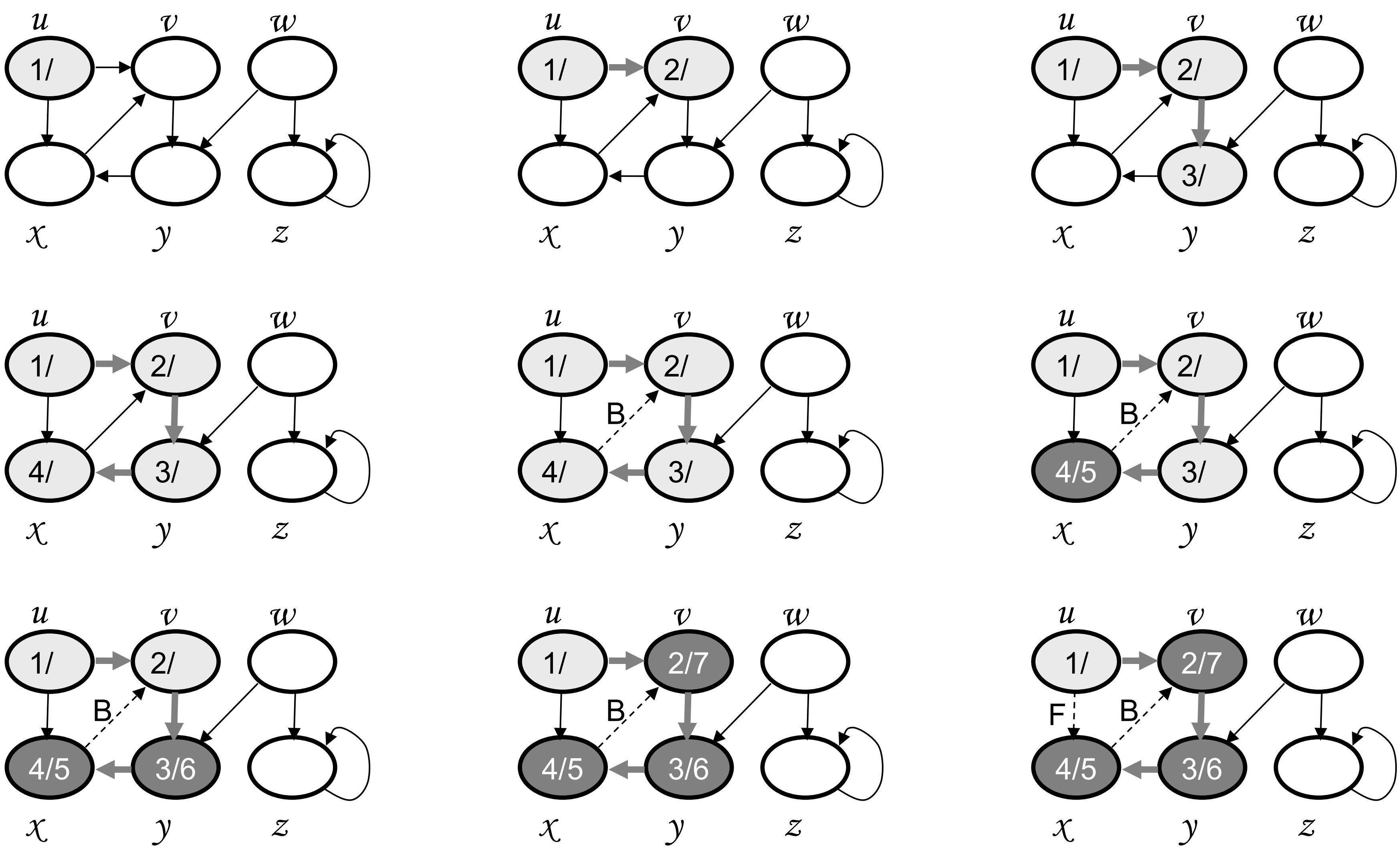
- Every time DFS-VISIT(u) is called, **u** becomes the root of a new tree in the depth-first forest

DFS-VISIT(u)

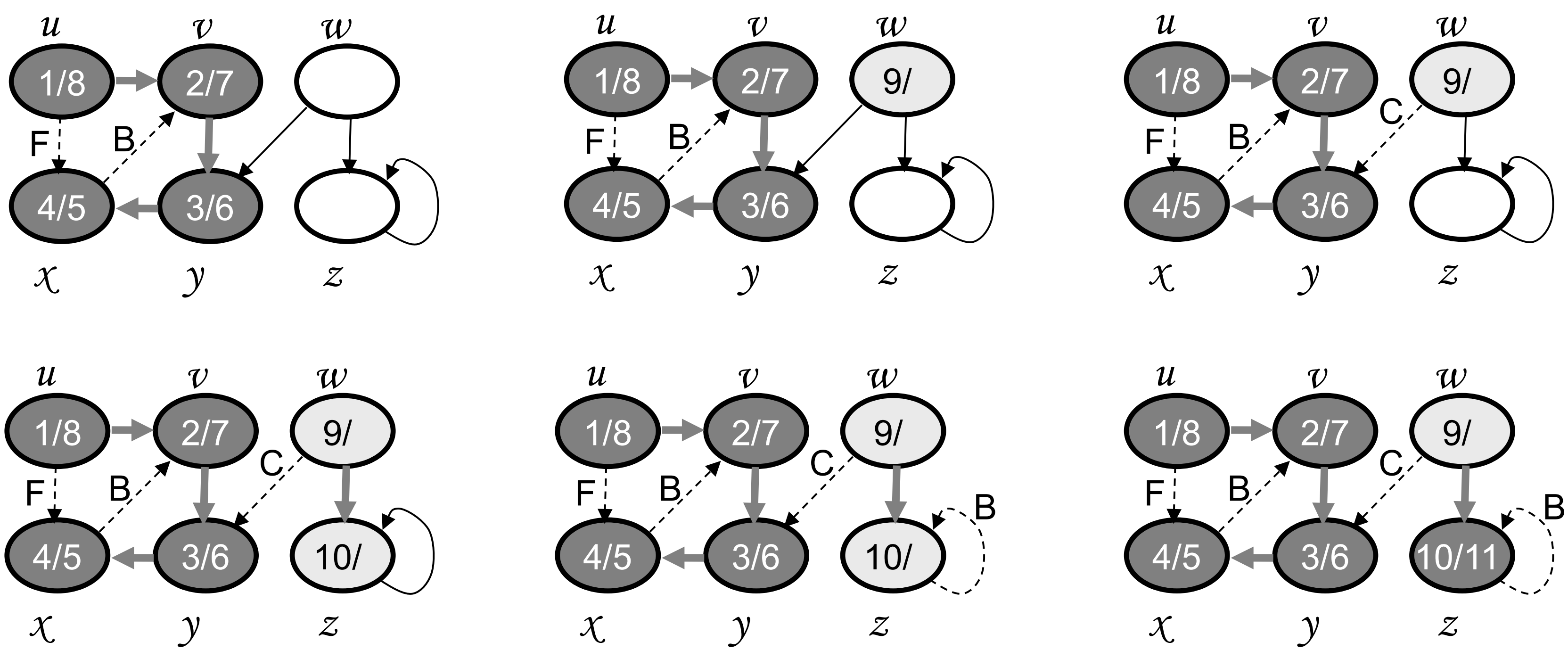
```
color[u] ← GRAY
time ← time+1
d[u] ← time
for each v ∈ Adj[u]
      do if color[v] = WHITE
            then π[v] ← u
                  DFS-VISIT(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

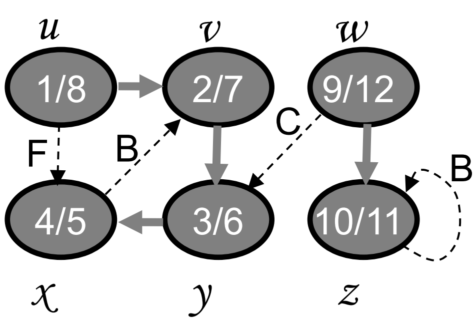


Example



Example (cont.)



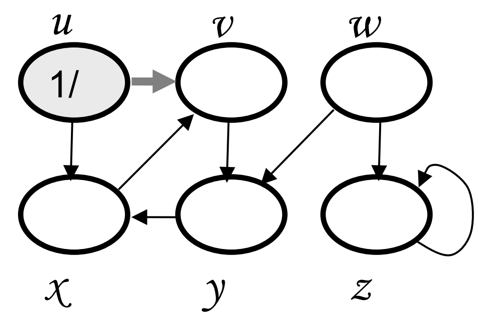


The results of DFS may depend on:

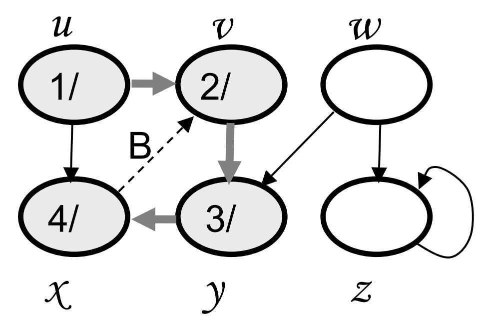
- The order in which nodes are explored in procedure DFS
- The order in which the neighbors of a vertex are visited in DFS-VISIT

Edge Classification

- **Tree edge** (reaches a WHITE vertex):
  - **(u, v)** is a tree edge if **v** was first discovered by exploring edge **(u, v)**

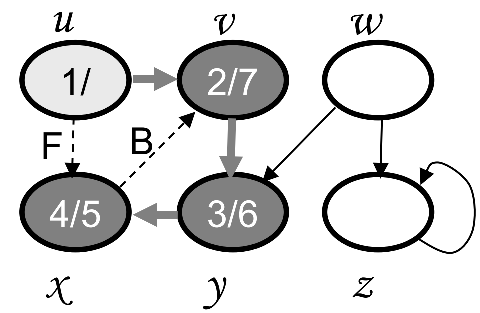


- **Back edge** (reaches a GRAY vertex):
  - **(u, v)**, connecting a vertex **u** to an ancestor **v** in a depth first tree
  - Self loops (in directed graphs) are also back edges

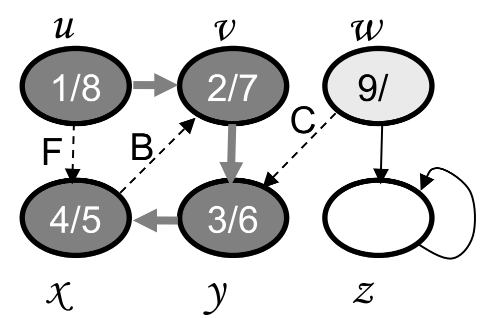


Edge Classification

- **Forward edge** (reaches a BLACK vertex & **d[u] < d[v]**):
  - Non-tree edge **(u, v)** that connects a vertex **u** to a descendant **v** in a depth first tree



- **Cross edge** (reaches a BLACK vertex & **d[u] > d[v]**):
  - Can go between vertices in same depth-first tree (as long as there is no ancestor / descendant relation) or between different depth-first trees



Analysis of DFS(V, E)

1. **for** each u ∈ V
2. **do** color[u] ← WHITE
3. π[u] ← NIL
4. time ← 0

$\Theta(|V|)$

5. **for** each u ∈ V
6. **do if** color[u] = WHITE
7. **then** DFS-VISIT(u)

$\Theta(|V|)$ – without counting the time for DFS-VISIT

Analysis of DFS-VISIT(u)

```
color[u] ← GRAY
time ← time+1
d[u] ← time
for each v ∈ Adj[u]
     do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

DFS-VISIT is called exactly once for each vertex

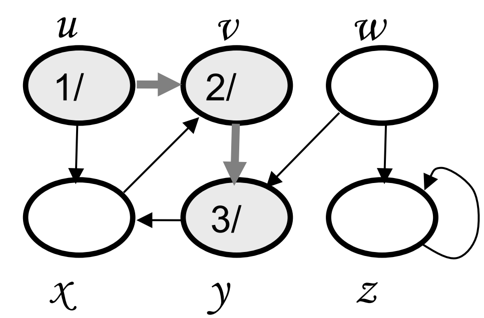
Each loop takes |Adj[u]|

Total: $\underbrace{\Sigma_{u \in V} |Adj[u]|}_{\Theta(|E|)} + \Theta(|V|) =$

$= \Theta(|V| + |E|)$

Properties of DFS

- **u** = **π[v]** ⟺ DFS-VISIT(**v**) was called during a search of **u**'s adjacency list

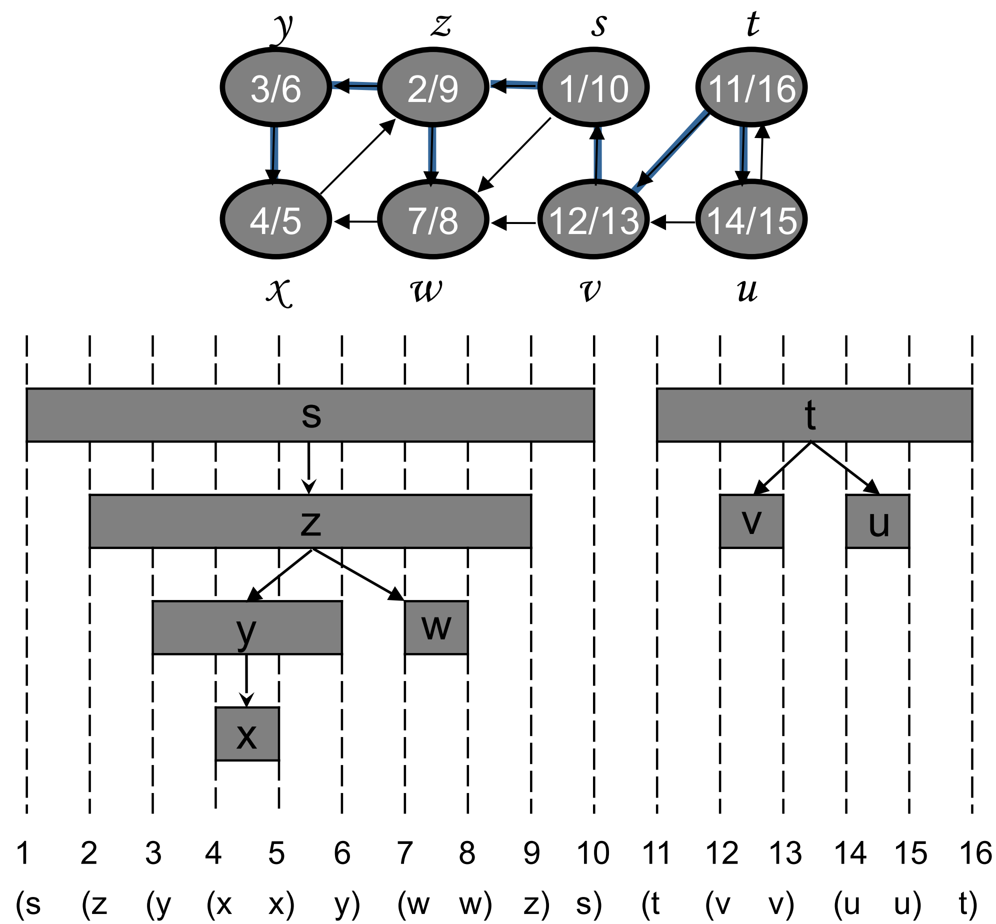


- Vertex **v** is a descendant of vertex **u** in the depth first forest ⟺ **v** is discovered during the time in which **u** is gray

Parenthesis Theorem

In any DFS of a graph G,
for all **u**, **v**, exactly one of the following holds:

1. **[d[u], f[u]]** and **[d[v], f[v]]** are disjoint, and neither of **u** and **v** is a descendant of the other
2. **[d[v], f[v]]** is entirely within **[d[u], f[u]]** and **v** is a descendant of **u**
3. **[d[u], f[u]]** is entirely within **[d[v], f[v]]** and **u** is a descendant of **v**



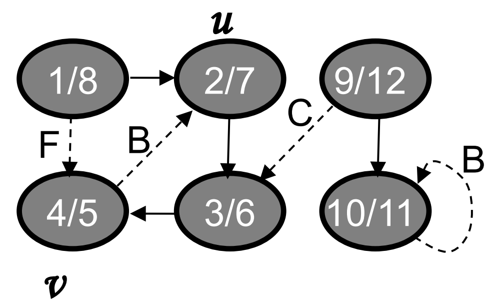
Well-formed expression: parenthesis are properly nested

Other Properties of DFS

Corollary

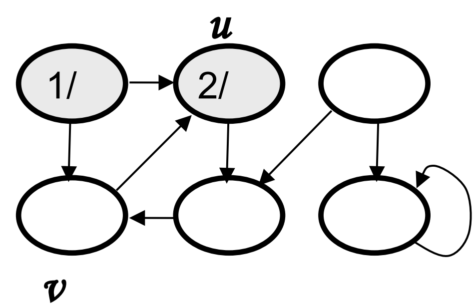
Vertex **v** is a proper descendant of **u**

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



Theorem (White-path Theorem)

In a depth-first forest of a graph G, vertex **v** is a descendant of **u** if and only if at time **d[u]**, there is a path **u** ⇨ **v** consisting of only white vertices.



Topological Sort

Topological sort of a directed acyclic graph G = (V, E): a linear order of vertices such that if there exists an edge **(u, v)**, then **u** appears before **v** in the ordering.

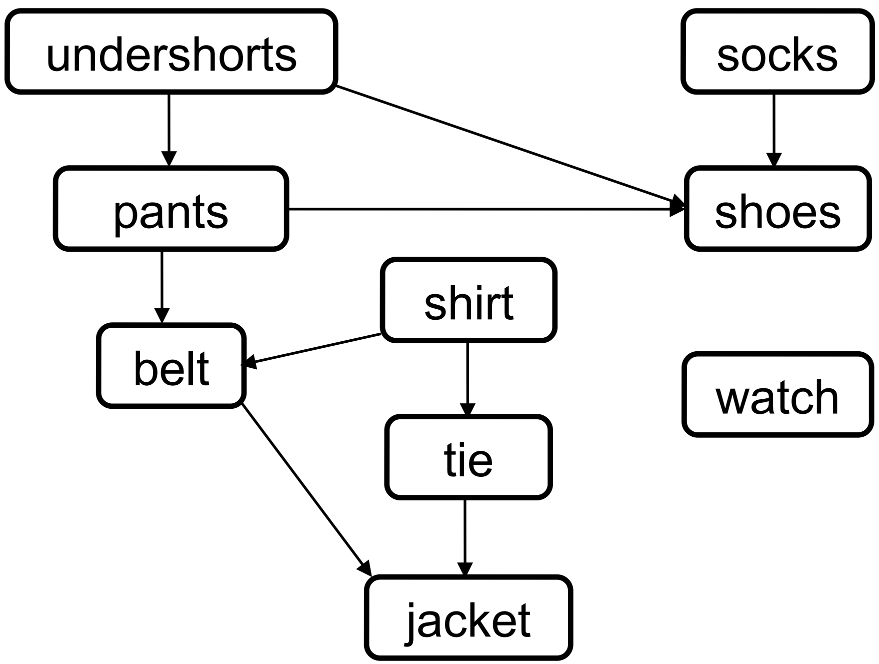
- Directed acyclic graphs (DAGs)
  - Used to represent precedence of events or processes that have a **partial order**

$\left.\begin{array}{l}\textbf{a} \text{ before } \textbf{b} \\ \textbf{b} \text{ before } \textbf{c}\end{array}\right\}$ **a** before **c**

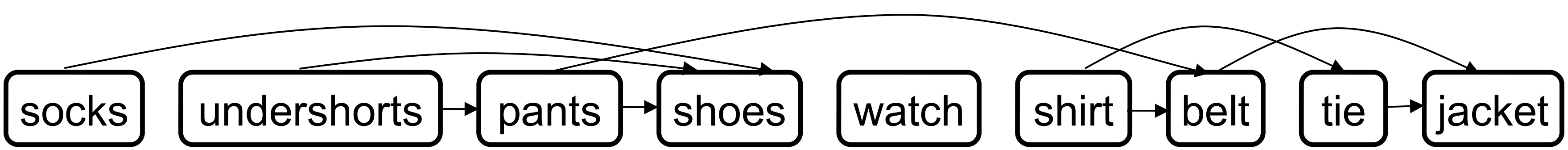
$\left.\begin{array}{l}\textbf{b} \text{ before } \textbf{c} \\ \textbf{a} \text{ before } \textbf{c}\end{array}\right\}$ What about **a** and **b**?

Topological sort helps us establish a **total order**

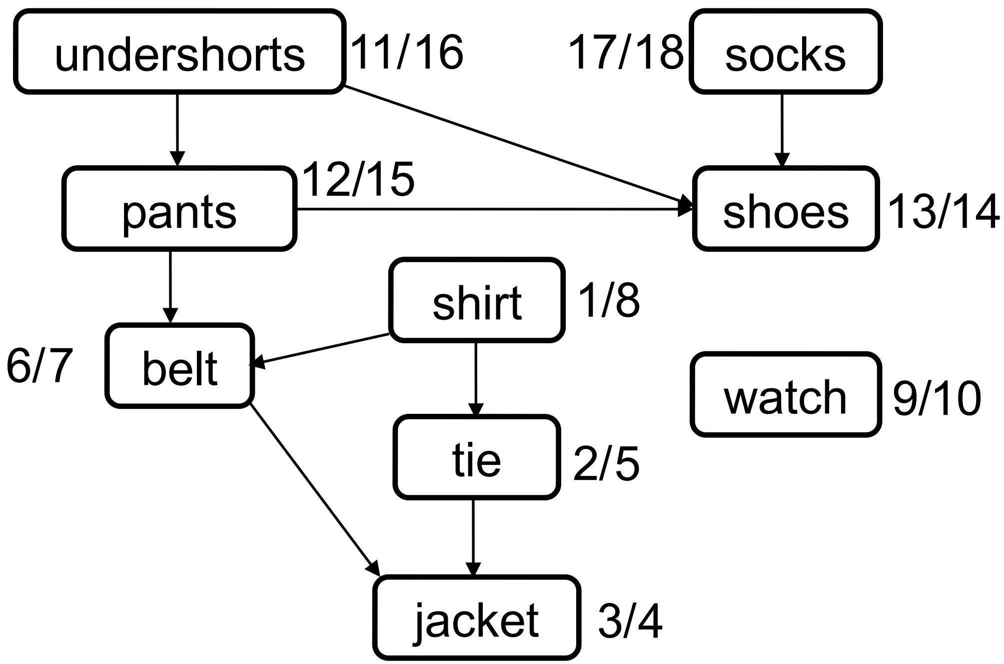
Topological Sort



Topological sort:
an ordering of vertices along a horizontal line so that all directed edges go from left to right.

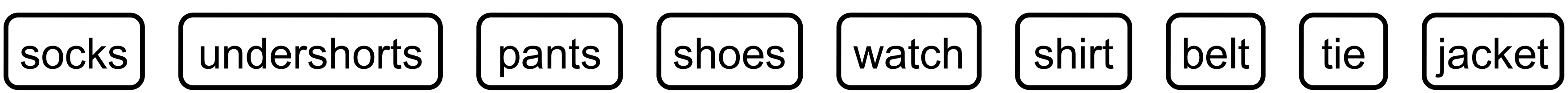


Topological Sort



TOPOLOGICAL-SORT(V, E)

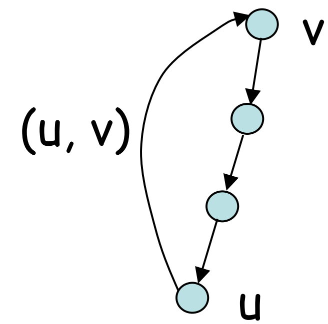
1. Call DFS(V, E) to compute finishing times **f[v]** for each vertex **v**
2. When each vertex is finished, insert it onto the front of a linked list
3. Return the linked list of vertices



Running time: $\Theta(|V| + |E|)$

Lemma

A directed graph is **acyclic** $\Longleftrightarrow$ a DFS on G yields no back edges.



Proof:

"⇒": acyclic ⇒ no back edge

- Assume **back edge** ⇒ prove **cycle**
- Assume there is a back edge **(u, v)**

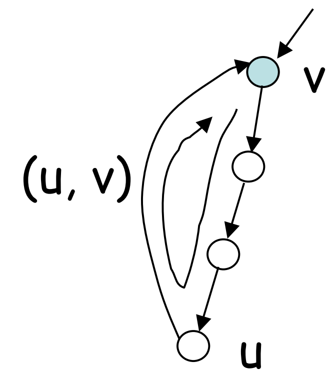
⇒ **v** is an ancestor of **u**

⇒ there is a path from **v** to **u** in G (**v** ⇨ **u**)

⇒ **v** ⇨ **u** + the back edge **(u, v)** yield a cycle

Lemma

A directed graph is **acyclic** $\Longleftrightarrow$ a DFS on G yields no back edges.



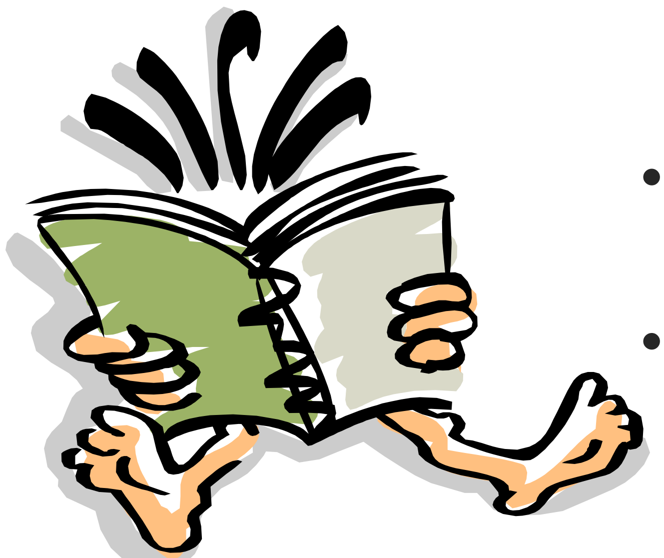
Proof:

"$\Longleftarrow$": no back edge $\Rightarrow$ acyclic

- Assume **cycle** $\Rightarrow$ prove **back edge**
- Suppose G contains cycle **c**
- Let **v** be the first vertex discovered in **c**, and **(u, v)** be the preceding edge in **c**
- At time **d[v]**, vertices of **c** form a white path **v** ⇨ **u**
- **u** is descendant of **v** in depth-first forest (by white-path theorem)

$\Rightarrow$ **(u, v)** is a back edge

Readings



- For this lecture
  - Chapter 15
- Coming next
  - Chapter 20