COMP 550
Algorithm and Analysis

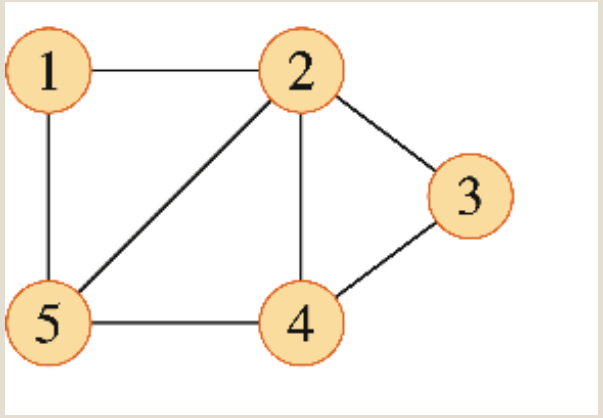
Elementary Graph Algorithms

Based on CLRS Sec. 20 and Appendix B.4

Graph: Terminology

- A Graph $G = (V, E)$
  - V = Set of vertices (or nodes)
  - E = Set of edges $\subseteq (V \times V)$
- $V = \{v_1, v_2, \ldots, v_n\}$ (vertices are typically denoted as v_i)
  - Number of vertices, $|V| = n$
- $E = \{e_1 = (v_i, v_j), \ldots, e_m = (v_k, v_\ell)\}$
  - Number of edges, $|E| = m$
  - $|E| = O(|V|^2)$

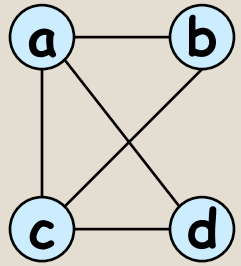
Example:



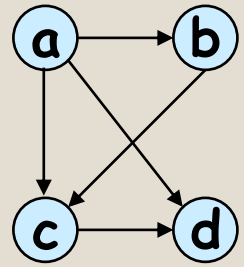
- $V = \{1,2,3,4,5\}$
  - $|V| = 5$
- $E = \{(1,2), (1,5), (2,3), (2,4), (2,5), (3,4), (4,5)\}$
  - $|E| = 7$

Graph: Terminology

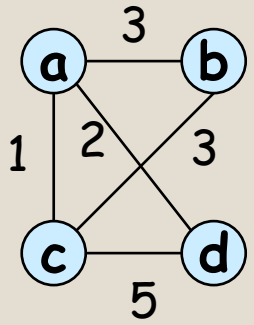
- Types of graphs
  - Undirected: edge $(u, v) = (v, u)$
    - CLRS definition forbids self loop.
  - Directed (digraph): (u, v) is edge from u to v.
    - Self loop possible. (Simple digraph has no self loop)
  - Weighted: each edge has an associated weight, given by a weight function $w : E \rightarrow \boldsymbol{R}$
  - Dense: $|E| \approx |V|^2$
  - Sparse: $|E| << |V|^2$



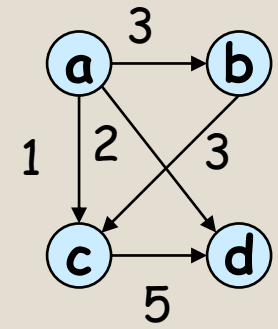
Undirected graph



Directed graph



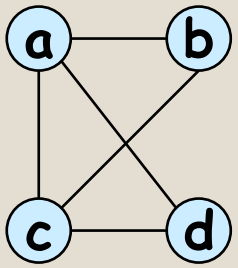
Weighted undirected graph



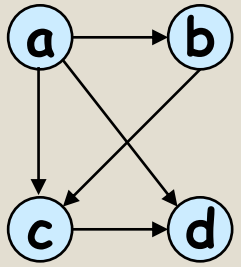
Weighted directed graph

Graph: Terminology

- Degree of a vertex $\deg(v)$: Number of edges incident to v
  - For directed graph, in-degree and out-degree of a vertex v are the number of edges to and from v.
- If $(u, v) \in E$, then vertex v is adjacent to vertex u.
- Adjacency relationship is:
  - Symmetric if G is undirected
  - Not necessarily so if G is directed



Undirected graph



Directed graph

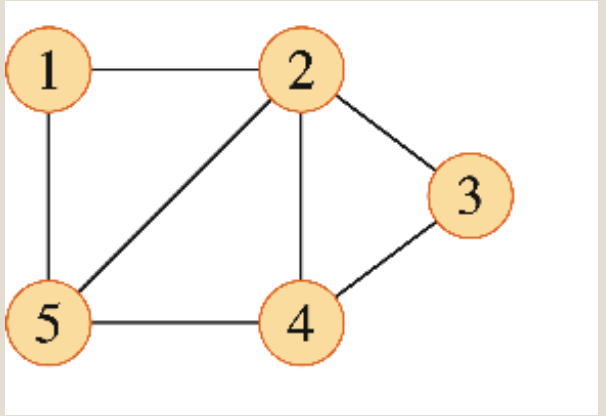
Degree of c is 3

In-Degree of c is 1

Out-degree of c is 2

Graph: Terminology

- Path:
  - A sequence of vertices $\langle v_1, v_2, \ldots, v_k \rangle$ where $\forall 1 \leq i \leq k-1,\ (v_i, v_{i+1}) \in E$
  - Length of the path: Number of edges in the path.
  - Path is simple if no vertex is repeated.
- Cycle
  - Path that ends back at starting nod
- G is connected:
  - There is a path between every pair of vertices.
  - $|E| \geq |V| - 1$.
  - Furthermore, if $|E| = |V| - 1$, then G is a tree.
- Other definitions in Appendix B (B.4 and B.5) as needed



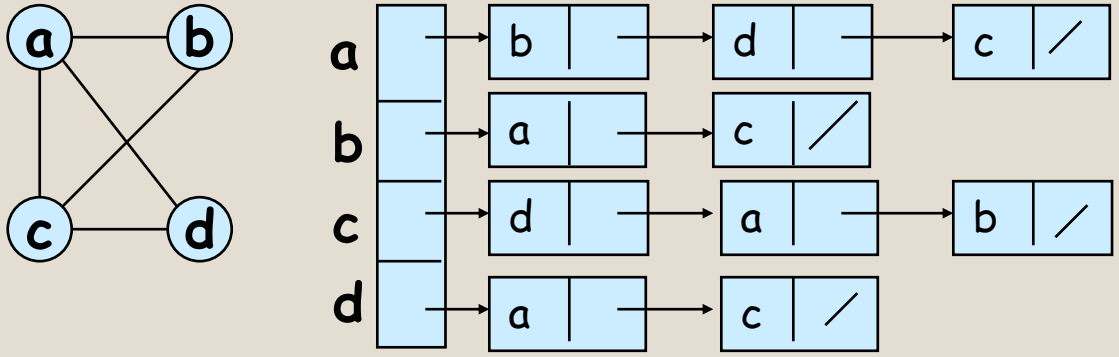
- $\langle 1,2,3 \rangle$ is a path
- $\langle 1,2,5,1 \rangle$ is a cycle

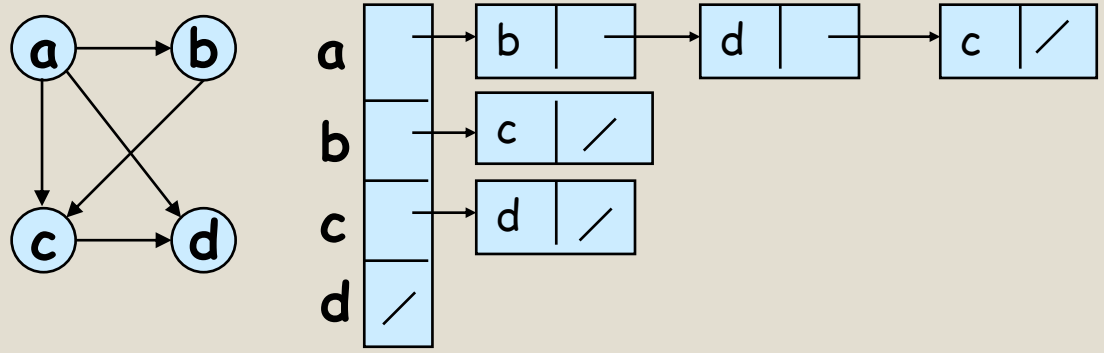
Applications

- Everywhere!
  - Road or communication network
  - Social media
  - Protein-protein interactions
  - etc.

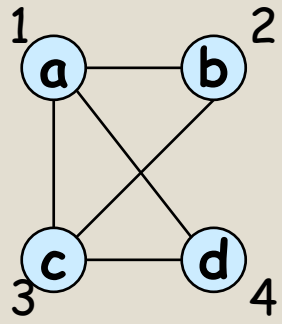
Graph Representations

- Two standard ways
  - Adjacency Lists

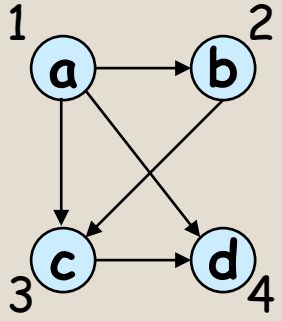




  - Adjacency Matrix



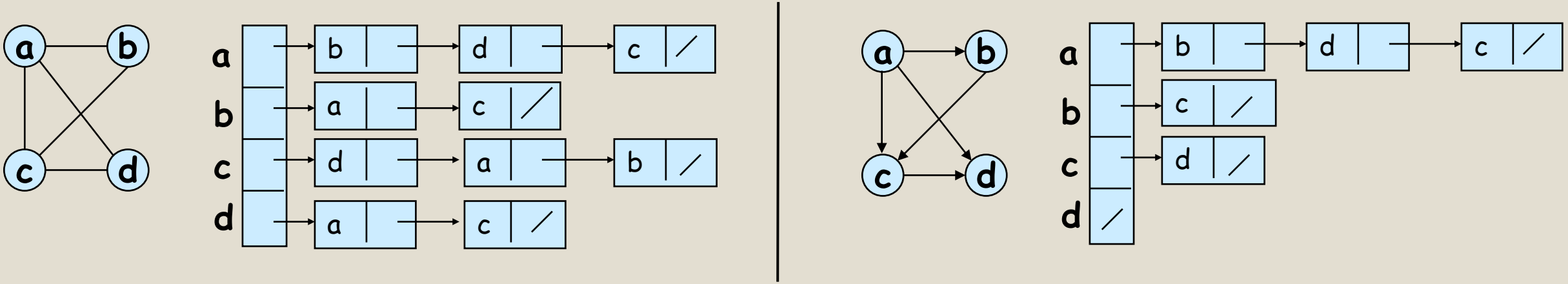
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0



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

Adjacency Lists

- Consists of an array Adj of $|V|$ lists
- One list per vertex
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to u



If weighted, store weights also in adjacency lists

Adjacency Lists

- For directed graphs:
  - Sum of lengths of all adj. lists is

$$\sum_{v \in V} \text{out-degree}(v) = |E|$$

  - Total storage: $\Theta(V + E)$
- For undirected graphs:
  - Sum of lengths of all adj. lists is

$$\sum_{v \in V} \text{degree}(v) = 2|E|$$

  - Total storage: $\Theta(V + E)$

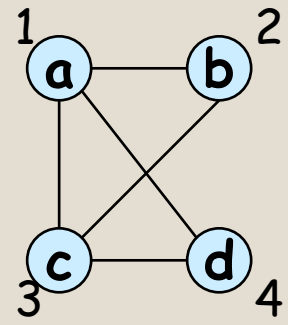
Adjacency Lists

- Pros
  - Space-efficient, when a graph is sparse.
  - Can be modified to support many graph variants.
- Cons
  - Determining if an edge $(u, v) \in E$ is not efficient.
    - Have to search in u's adjacency list in $\Theta(degree(u))$ time.
    - $\Theta(V)$ in the worst case.

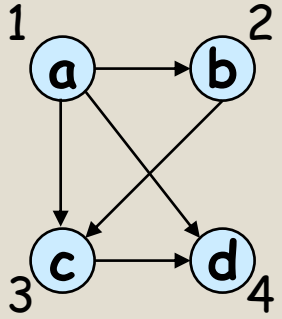
Adjacency Matrix

- $|V| \times |V|$ matrix A
- Number vertices from 1 to $|V|$ in some arbitrary manner

$$A[i, j] = a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$



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



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

Adjacency Matrix

- **Space:** $\Theta(V^2)$
  - Not memory efficient for large graphs
- **Time:** to list all vertices adjacent to u: $\Theta(V)$
- **Time:** to determine if $(u, v) \in E$: $\Theta(1)$
- Can store weights instead of bits for weighted graph.

Graph Search

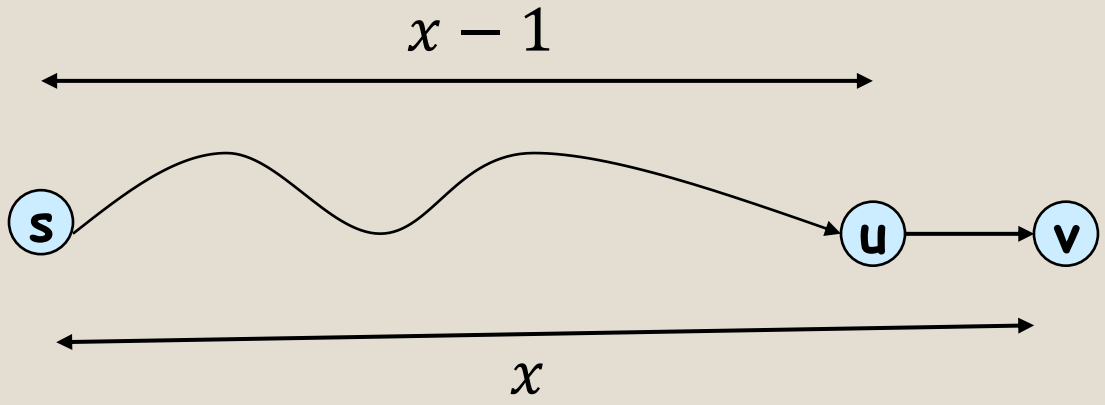
- Searching a graph:
  - Systematically follow the edges of a graph to visit the vertices of the graph
- Used to discover the structure of a graph
- Standard graph-searching algorithms
  - Breadth-first Search (BFS)
  - Depth-first Search (DFS)

Breadth-First Search (BFS)

- Given a graph $G = (V, E)$ and a source vertex s, want to discover vertices reachable from s and their shortest path distance from s
- Input: Graph $G = (V, E)$, either directed or undirected, and a source vertex $s \in V$
- Output:
  - $v.d$ = distance (smallest # of edges, or shortest path) from s to v, for all $v \in V$. $v.d = \infty$ if v is not reachable from s.
  - $v.\pi = u$ such that (u, v) is last edge on shortest path $s \rightsquigarrow v$
    - u is v's predecessor.
  - Builds breadth-first tree with root s that contains all reachable vertices

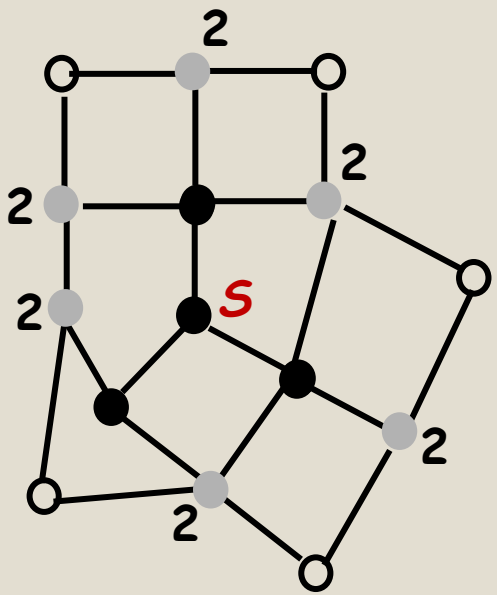
Breadth-First Search (BFS)

- We want our search algorithm to produce shortest distance from s to v for all v.
- Idea:
  - Notation: Shortest-path distance from s to v is $\delta(s, v)$
  - If $\delta(s, v) = x \geq 1$, then there is a vertex u with $\delta(s, u) = x - 1$
  - We want to discover v via u



Breadth-First Search (BFS)

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - A vertex is "discovered" the first time it is encountered during the search.
  - A vertex is "finished" if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Gray – Discovered but not finished.
  - Black – Finished.
  - Colors are basically different numbers/characters
  - Colors are also not required



Finished

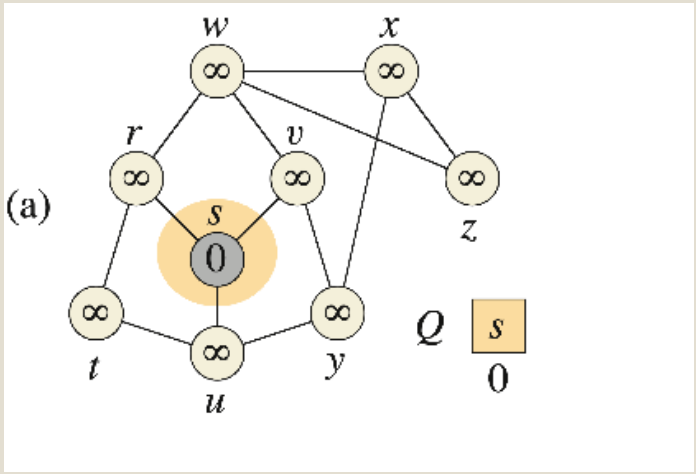
Discovered

Undiscovered

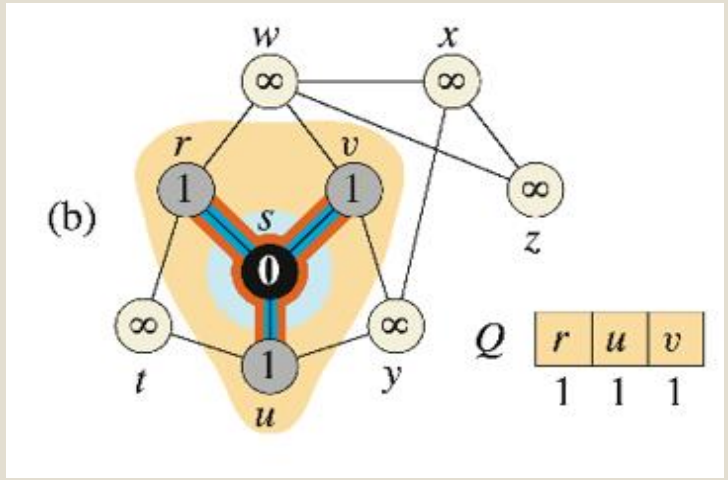
Breadth-First Search (BFS)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each vertex v in G.Adj[u]    // search the neighbors of u
13            if v.color == WHITE         // is v being discovered now?
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)          // v is now on the frontier
18       u.color = BLACK                  // u is now behind the frontier
```

Q holds discovered by unfinished vertices (gray vertices).



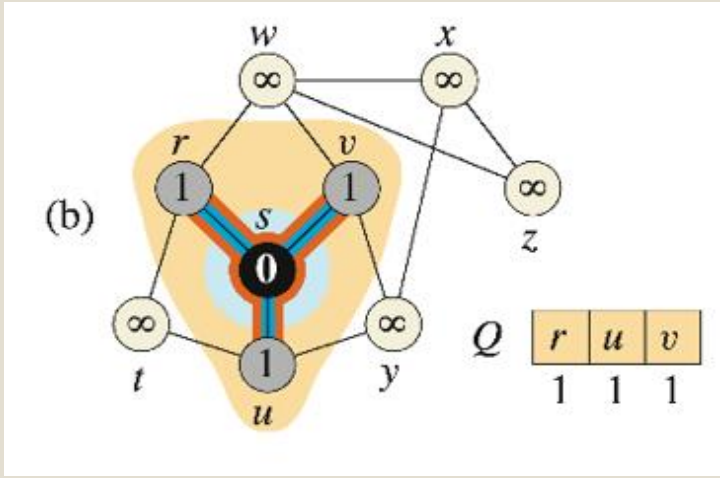
i) Dequeue ***s***, ii) Enqueue ***s***'s undiscovered neighbors $\{r, v, u\}$, make then gray, and update their d and π, iii) Make ***s*** black



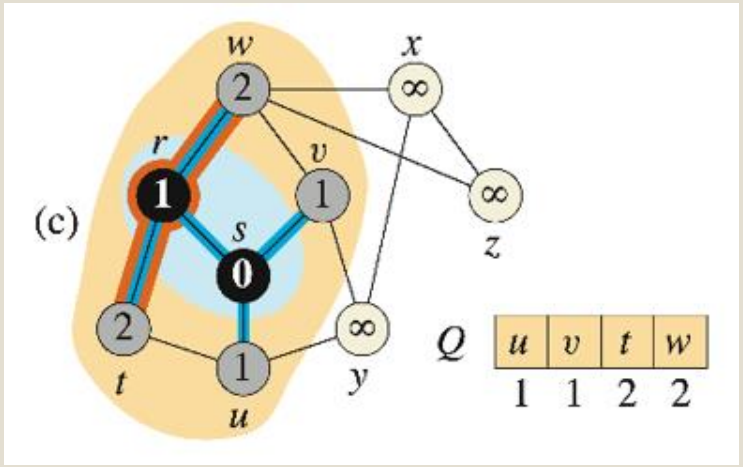
Breadth-First Search (BFS)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each vertex v in G.Adj[u]    // search the neighbors of u
13            if v.color == WHITE         // is v being discovered now?
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)          // v is now on the frontier
18       u.color = BLACK                  // u is now behind the frontier
```

Q holds discovered by unfinished vertices (gray vertices).



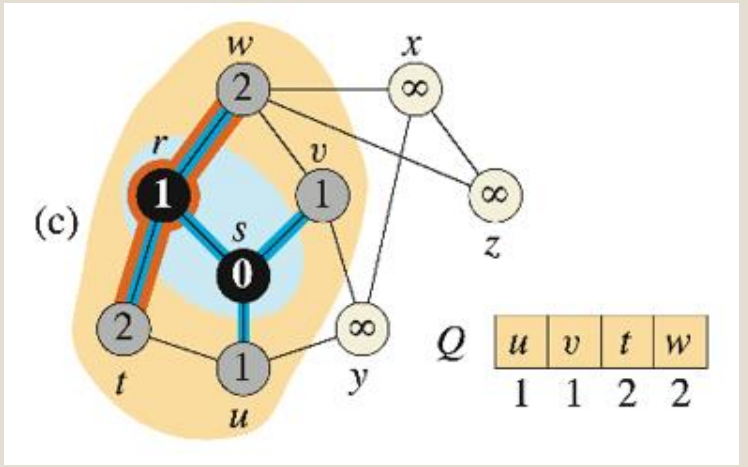
i) Dequeue ***r***, ii) Enqueue ***r***'s undiscovered neighbors $\{t, w\}$, make then gray, and update their d and π, iii) Make ***r*** black



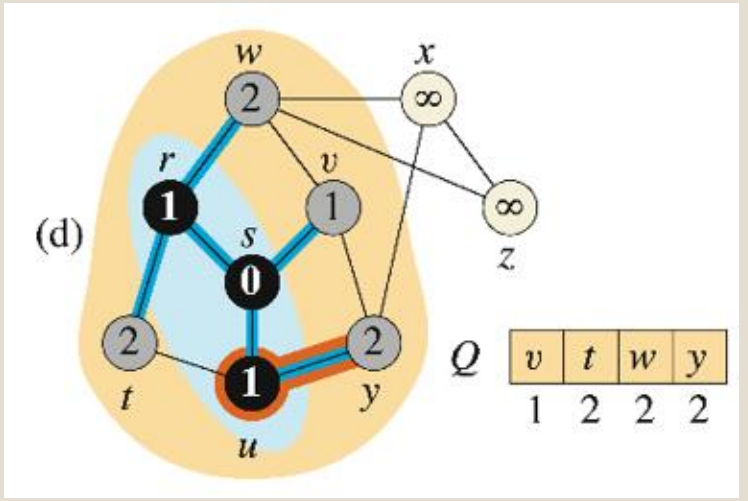
Breadth-First Search (BFS)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each vertex v in G.Adj[u]    // search the neighbors of u
13            if v.color == WHITE         // is v being discovered now?
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)          // v is now on the frontier
18       u.color = BLACK                  // u is now behind the frontier
```

Q holds discovered by unfinished vertices (gray vertices).



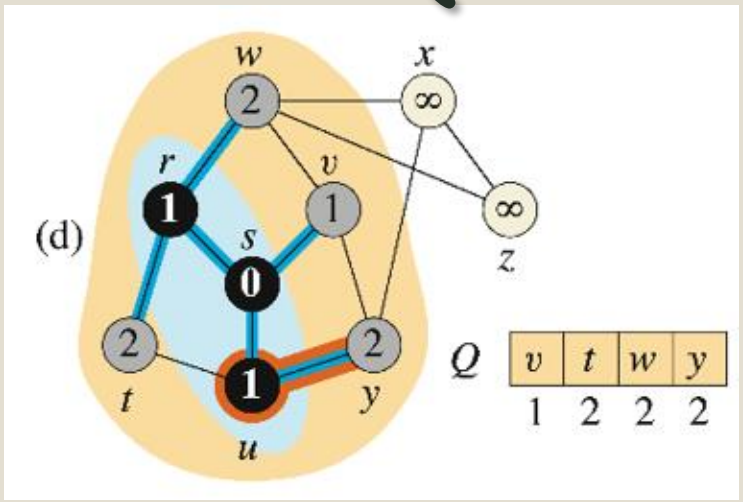
i) Dequeue ***u***, ii) Enqueue ***u***'s undiscovered neighbors $\{y\}$, make then gray, and update their d and π, iii) Make ***u*** black



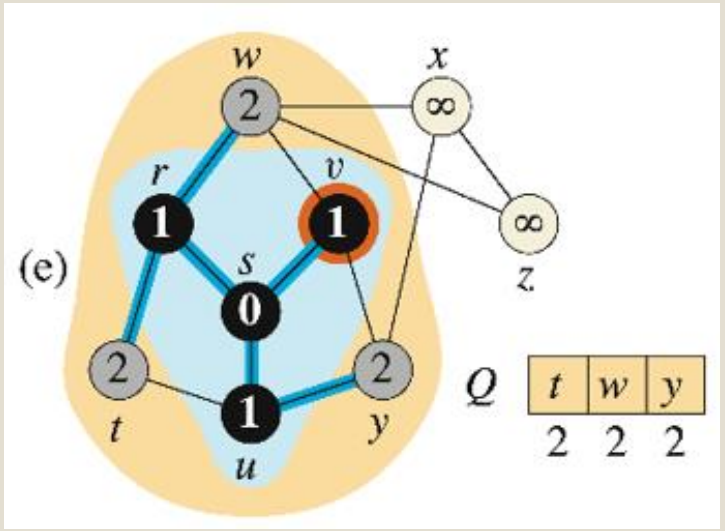
Breadth-First Search (BFS)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each vertex v in G.Adj[u]    // search the neighbors of u
13            if v.color == WHITE         // is v being discovered now?
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)          // v is now on the frontier
18       u.color = BLACK                  // u is now behind the frontier
```

Q holds discovered by unfinished vertices (gray vertices).



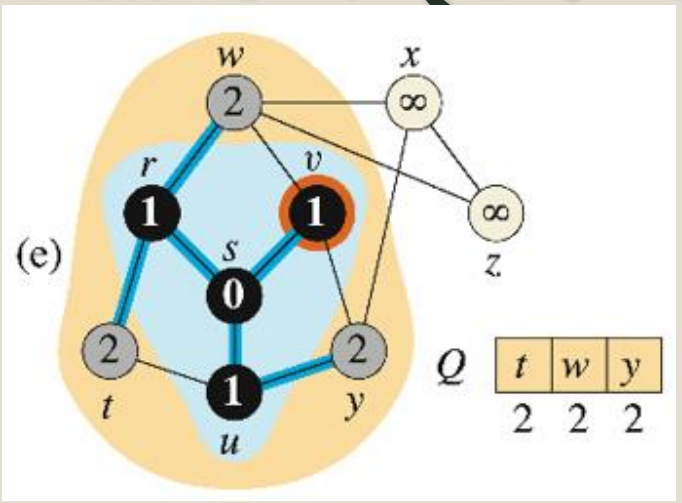
i) Dequeue v, ii) Enqueue v's undiscovered neighbors (None), make then gray, and update their d and π, iii) Make v black



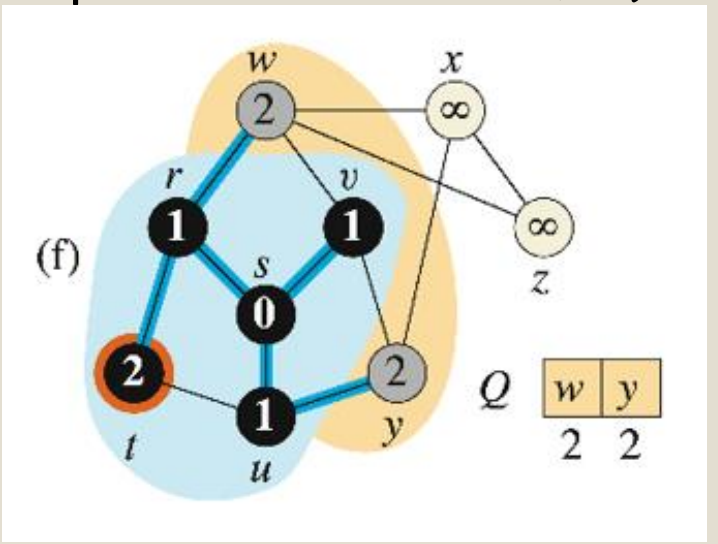
Breadth-First Search (BFS)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each vertex v in G.Adj[u]    // search the neighbors of u
13            if v.color == WHITE         // is v being discovered now?
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)          // v is now on the frontier
18       u.color = BLACK                  // u is now behind the frontier
```

Q holds discovered by unfinished vertices (gray vertices).



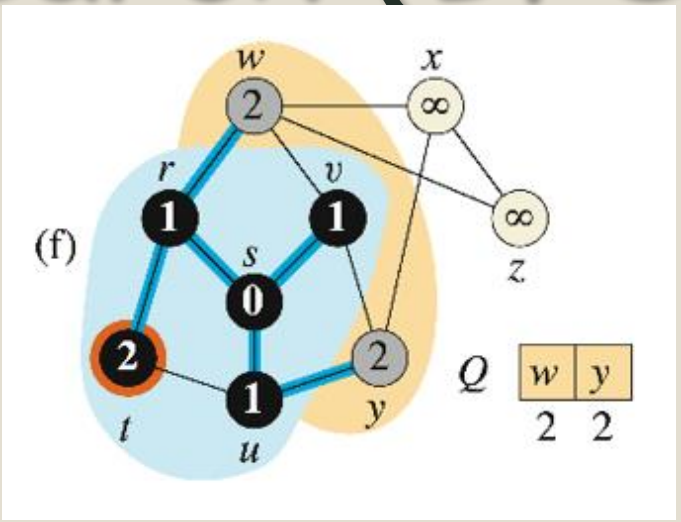
i) Dequeue ***t***, ii) Enqueue ***t***'s undiscovered neighbors (None), make then gray, and update their d and π, iii) Make ***t*** black



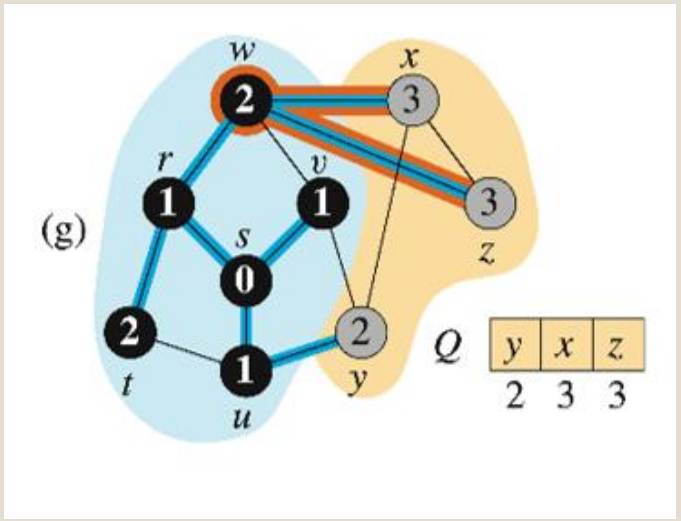
Breadth-First Search (BFS)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each vertex v in G.Adj[u]    // search the neighbors of u
13            if v.color == WHITE         // is v being discovered now?
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)          // v is now on the frontier
18       u.color = BLACK                  // u is now behind the frontier
```

Q holds discovered by unfinished vertices (gray vertices).



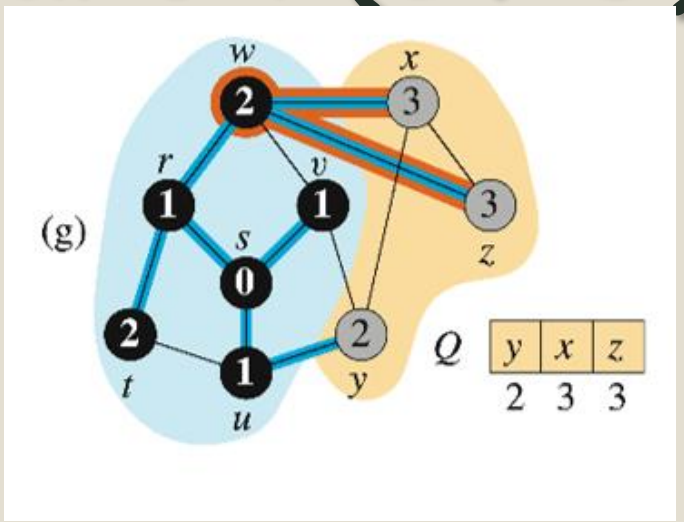
i) Dequeue ***w***, ii) Enqueue ***w***'s undiscovered neighbors $\{x, z\}$, make then gray, and update their d and π, iii) Make ***w*** black



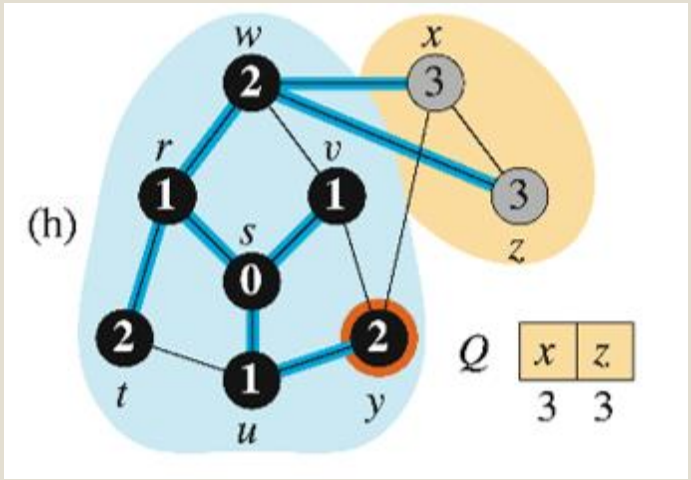
Breadth-First Search (BFS)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each vertex v in G.Adj[u]    // search the neighbors of u
13            if v.color == WHITE         // is v being discovered now?
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)          // v is now on the frontier
18       u.color = BLACK                  // u is now behind the frontier
```

Q holds discovered by unfinished vertices (gray vertices).



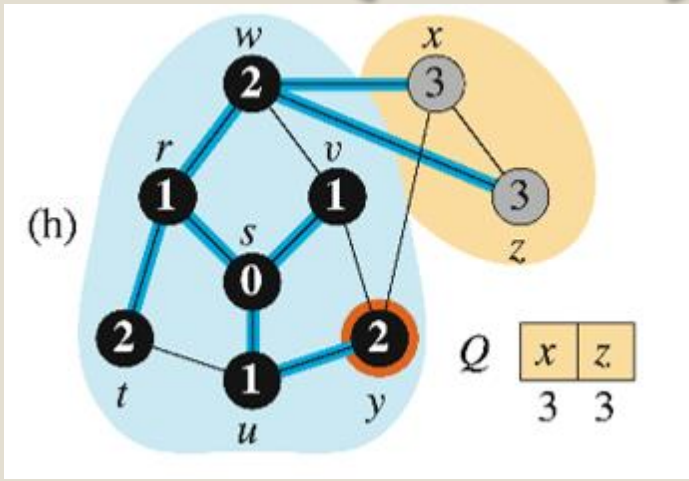
i) Dequeue ***y***, ii) Enqueue ***y***'s undiscovered neighbors (None), make then gray, and update their d and π, iii) Make ***y*** black



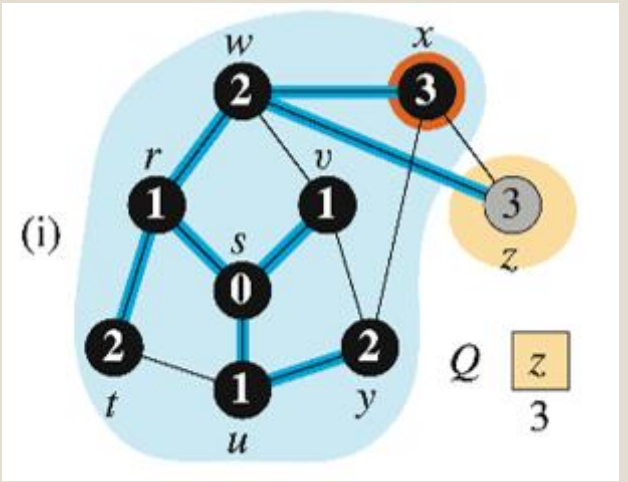
Breadth-First Search (BFS)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each vertex v in G.Adj[u]    // search the neighbors of u
13            if v.color == WHITE         // is v being discovered now?
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)          // v is now on the frontier
18       u.color = BLACK                  // u is now behind the frontier
```

Q holds discovered by unfinished vertices (gray vertices).



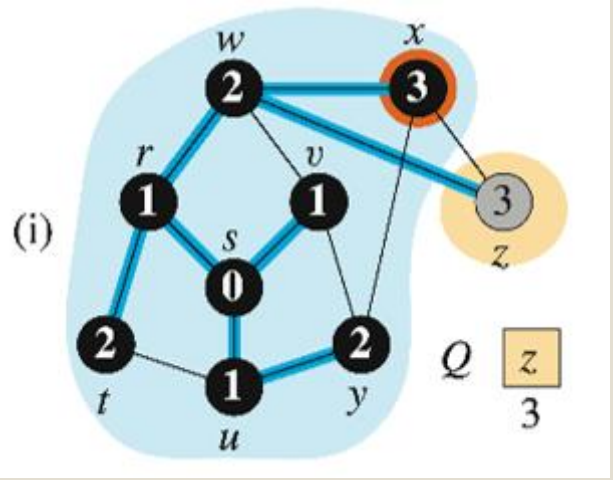
i) Dequeue ***x***, ii) Enqueue ***x***'s undiscovered neighbors (None), make then gray, and update their d and π, iii) Make ***x*** black



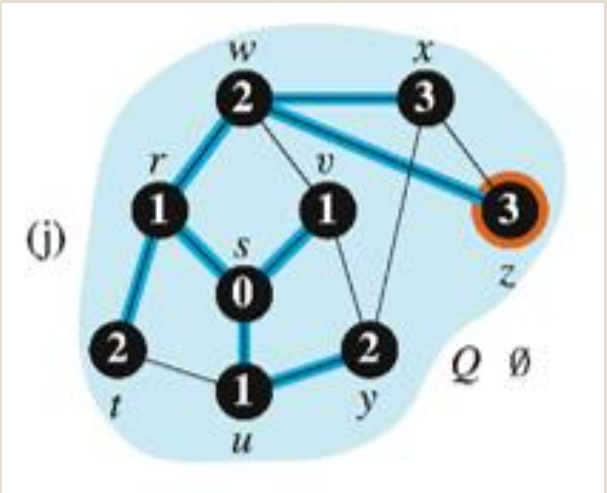
Breadth-First Search (BFS)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each vertex v in G.Adj[u]    // search the neighbors of u
13          if v.color == WHITE          // is v being discovered now?
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)            // v is now on the frontier
18      u.color = BLACK                  // u is now behind the frontier
```

Q holds discovered by unfinished vertices (gray vertices).



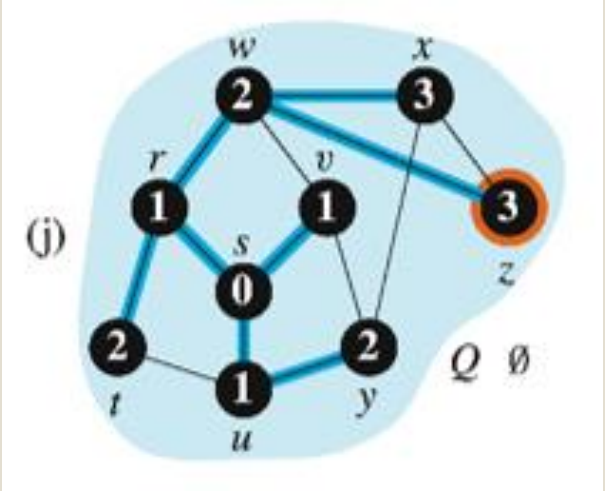
i) Dequeue **z**, ii) Enqueue **z**'s undiscovered neighbors (None), make then gray, and update their d and π, iii) Make **z** black



Breadth-First Search (BFS)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each vertex v in G.Adj[u]    // search the neighbors of u
13            if v.color == WHITE         // is v being discovered now?
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)          // v is now on the frontier
18       u.color = BLACK                  // u is now behind the frontier
```

Q holds discovered by unfinished vertices (gray vertices).



BFS tree formed by blue edges

BFS Time Complexity

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each vertex v in G.Adj[u]    // search the neighbors of u
13          if v.color == WHITE          // is v being discovered now?
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)            // v is now on the frontier
18      u.color = BLACK                  // u is now behind the frontier
```

- Initialization (lines 1-4): $\Theta(V)$
- Lines 5-9: $\Theta(1)$
- Aggregate analysis for lines 10-18
  - Each vertex is enqueued and dequeued at most once
  - Line 11, 14-16 take $\Theta(V)$ time
  - Adjacency list of each vertex is scanned at most once.
  - Line 12-13 take $\Theta(E)$ time
- With adjacency list, running time $\Theta(\mathrm{V} + \mathrm{E})$
- With adjacency matrix, running time $\Theta(V^2)$

Depth-First Search (DFS)

- Explore edges out of the most recently discovered vertex *v*.
- When all edges of *v* have been explored, backtrack to explore other edges leaving the vertex from which *v* was discovered (its *predecessor*).
- "Search as deep as possible first."
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

Depth-First Search (DFS)

- Input: $G = (V, E)$, directed or undirected. No source vertex!
- Output:
  - 2 timestamps on each vertex. Integers between 1 and $2|V|$
    - $v.d$ = discovery time (v turns from white to gray)
    - $v.f$ = finishing time (v turns from gray to black)
  - $v.\pi$: predecessor of $v = u$, such that v was discovered during the scan of u's adjacency list
- Uses the same coloring scheme for vertices as BFS

Depth-First Search (DFS)

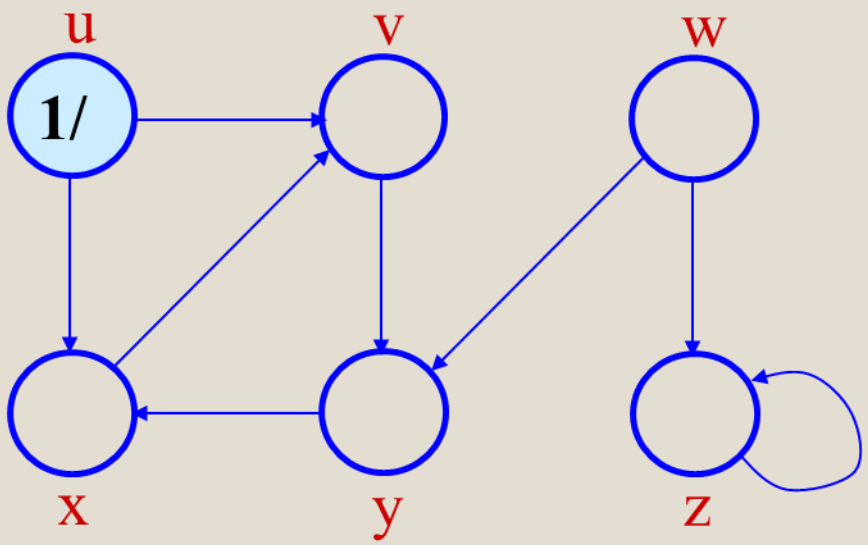
```
DFS(G)
1. for each vertex u ∈ G.V
2.       u.color = white
3.       u.π = NIL
4.  time = 0
5.  for each vertex u ∈ G.V
6.       if u.color == white
7.            DFS-Visit(G, u)
```

Uses a global timestamp ***time***.

```
DFS-Visit(G, u)
1.      u.color = GRAY     //white vertex u has been discovered
2.      time = time + 1
3.      u.d = time
4.      for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.           if v.color = WHITE
6.                v.π = u
7.                DFS-Visit(G, v)
8.       u.color = BLACK     // Blacken u;  it is finished
9.       u.f = time
10.     time = time + 1
```

Depth-First Search (DFS)

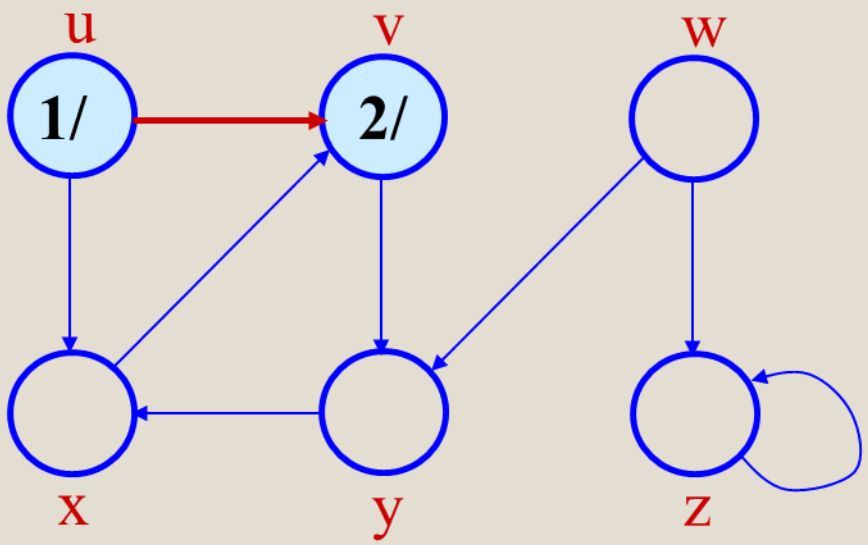
```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



(Courtesy of Prof. Jim Anderson)

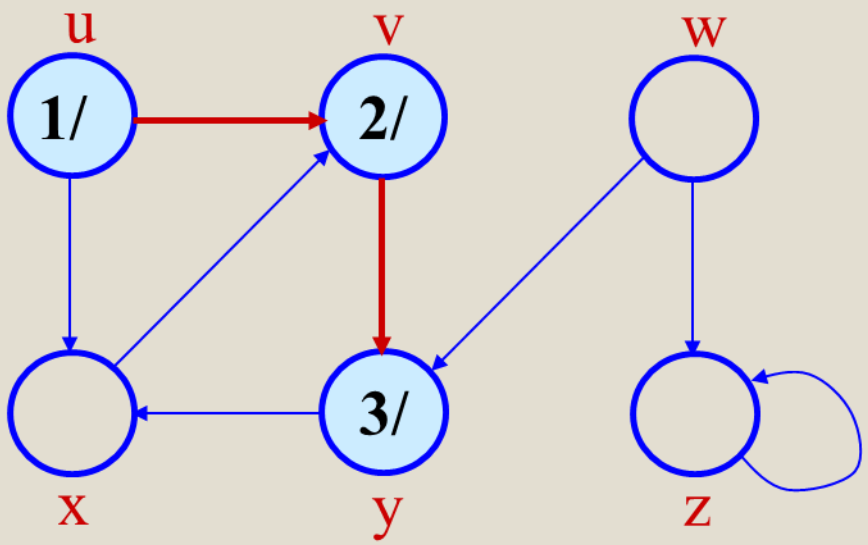
Depth-First Search (DFS)

```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



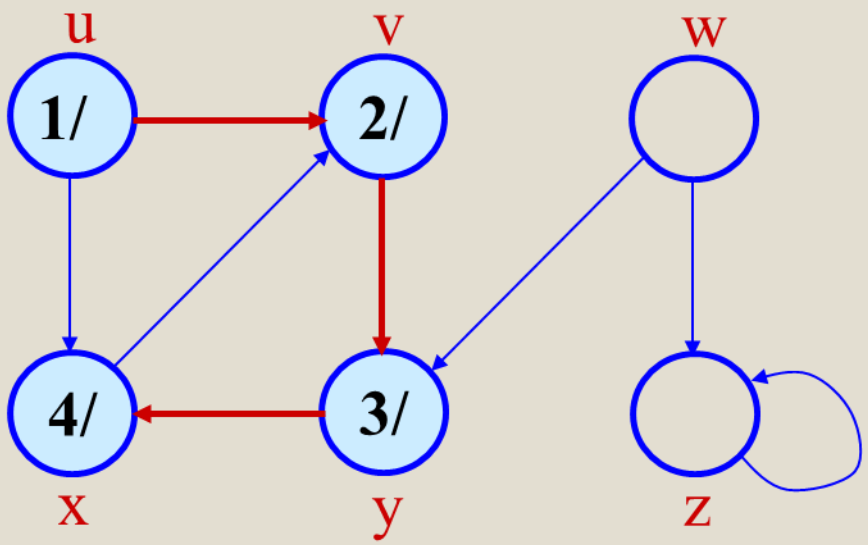
Depth-First Search (DFS)

```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



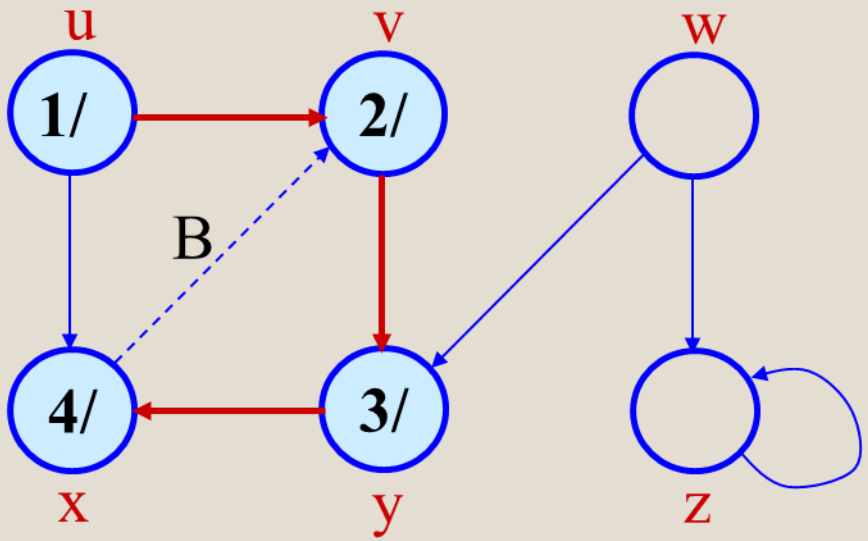
Depth-First Search (DFS)

```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



Depth-First Search (DFS)

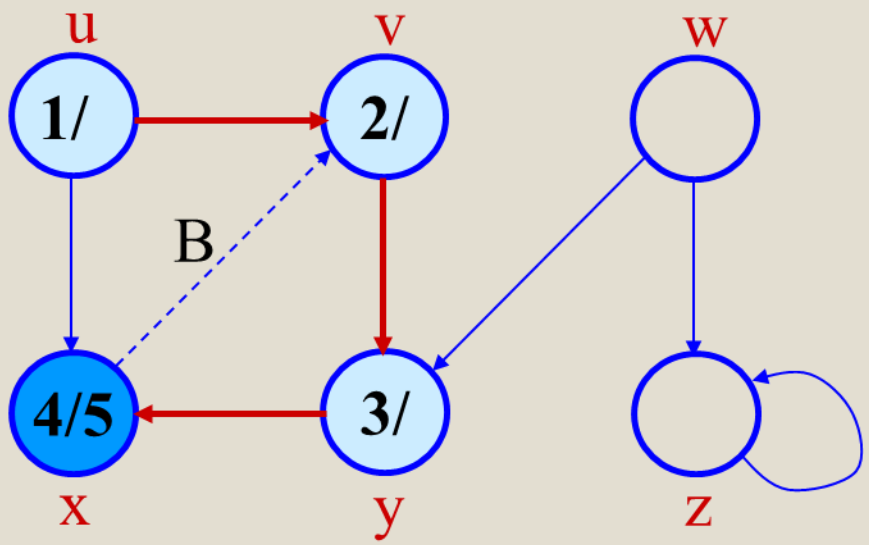
```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



Back Edge: (x, v) is a back edge if it is explored when both x and v are grey. (x is discovered by a path from v, i.e., **v is an ancestor of x**)

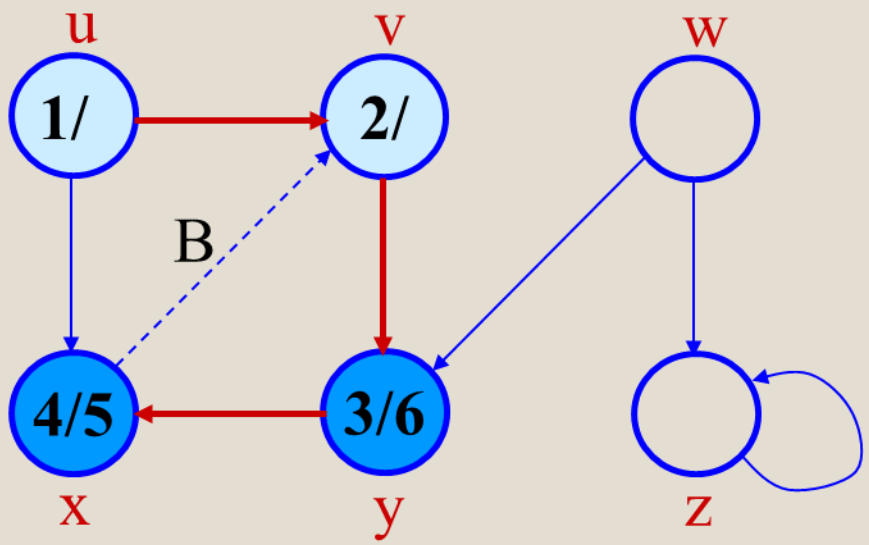
Depth-First Search (DFS)

```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



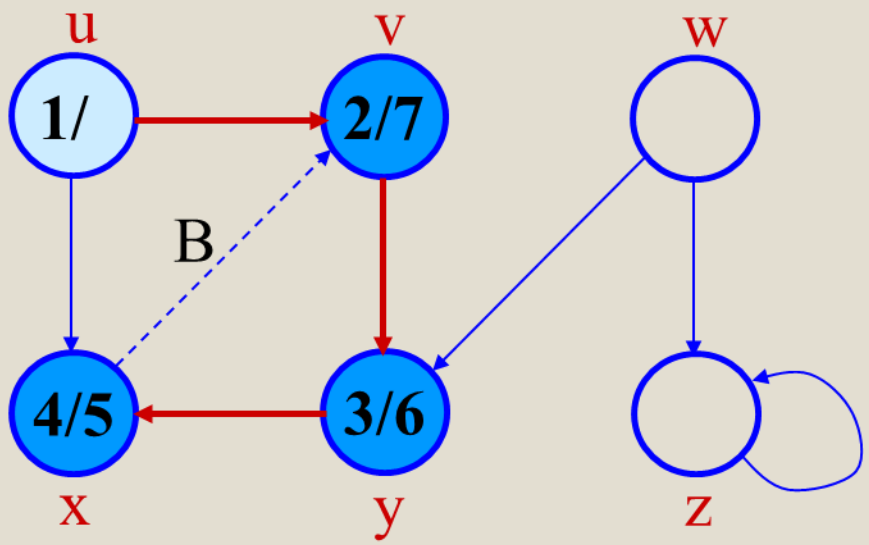
Depth-First Search (DFS)

```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



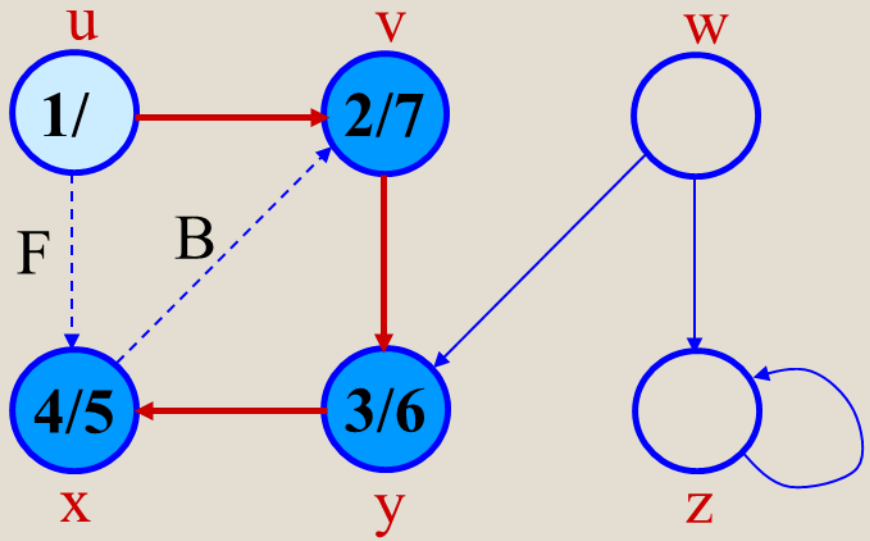
Depth-First Search (DFS)

```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



Depth-First Search (DFS)

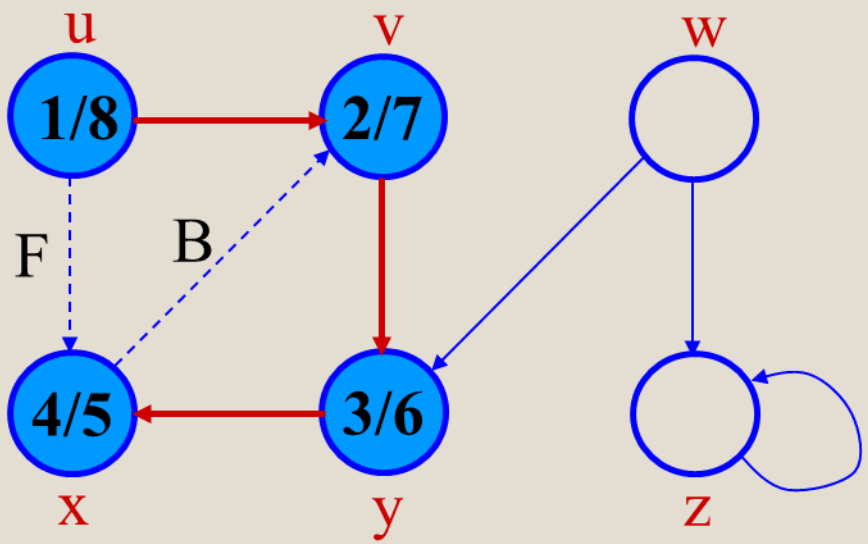
```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



Forward Edge: (u, x) is a forward edge if x is discovered by a ">1"-length path from u. (x **is a descendant of** u)

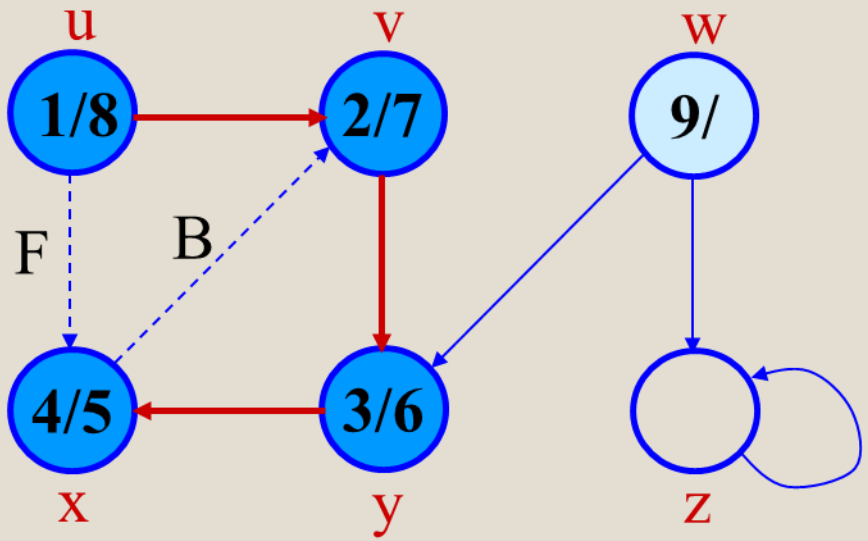
Depth-First Search (DFS)

```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



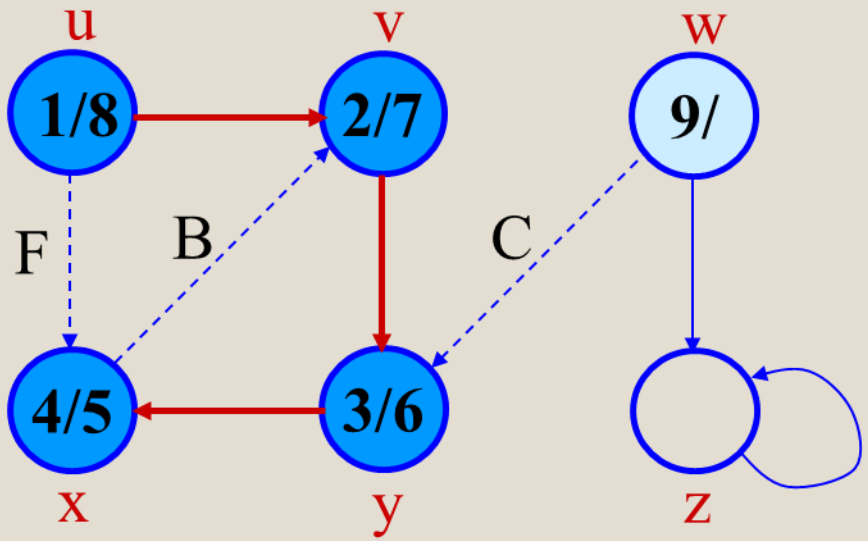
Depth-First Search (DFS)

```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



Depth-First Search (DFS)

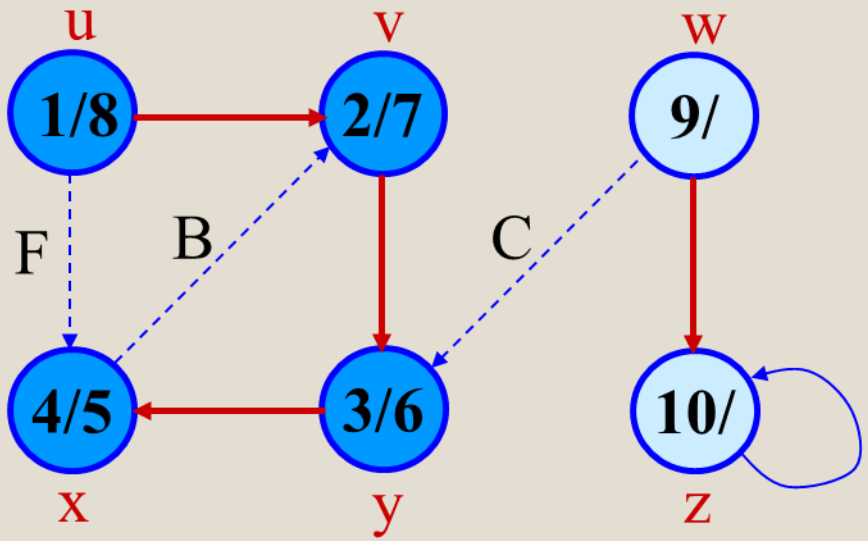
```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



Cross Edge: (w, y) is a back edge if no ancestor descendant relationships between them.

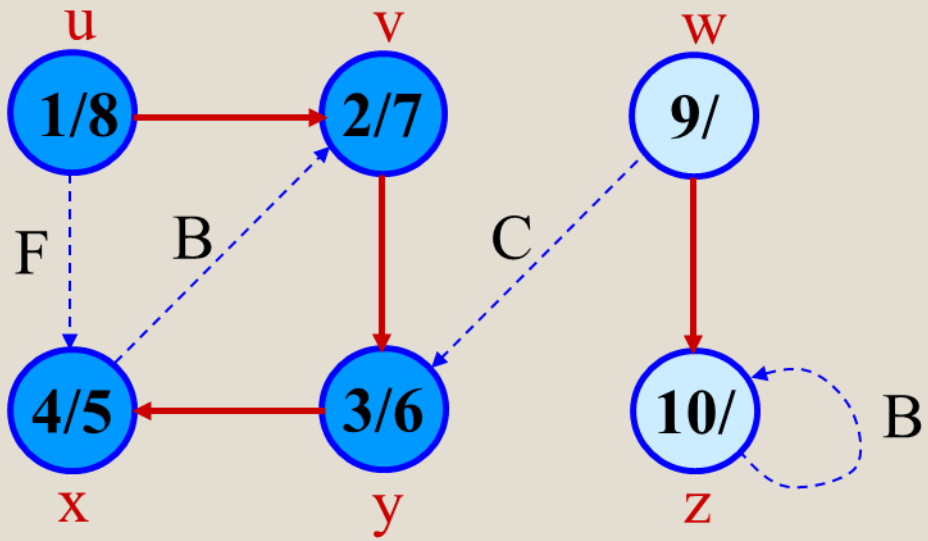
Depth-First Search (DFS)

```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



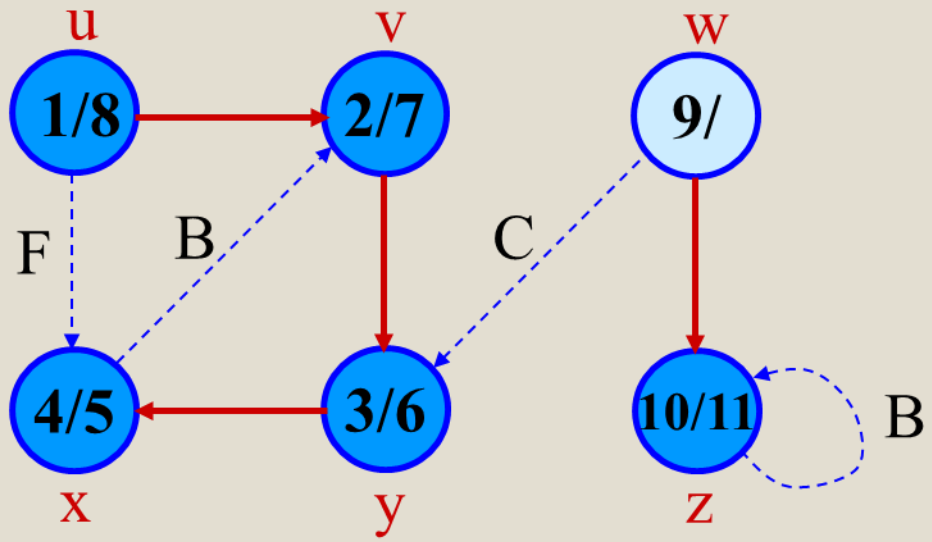
Depth-First Search (DFS)

```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



Depth-First Search (DFS)

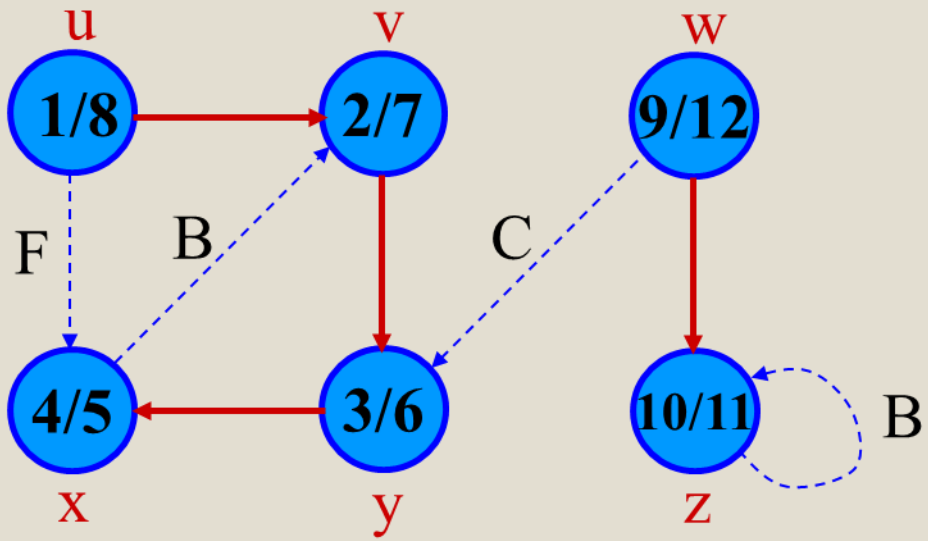
```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



Self loops are considered as back edge

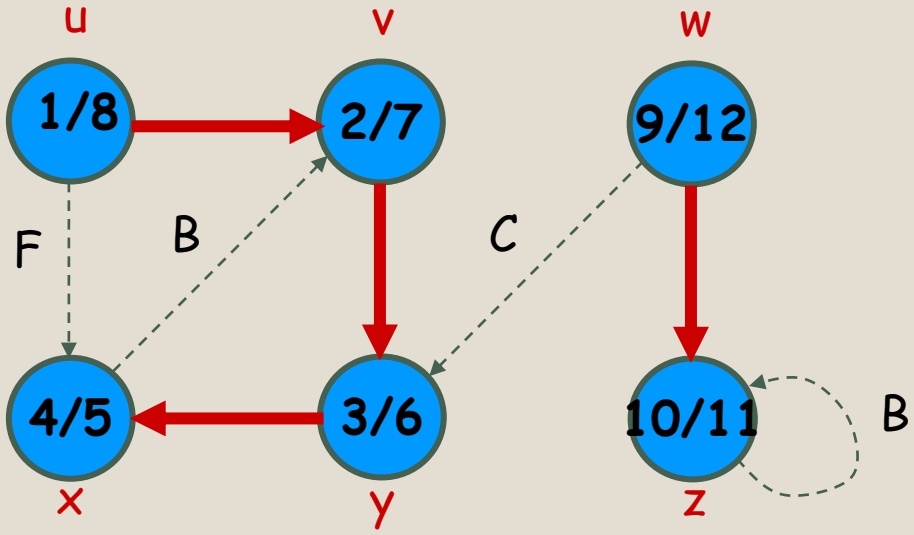
Depth-First Search (DFS)

```
DFS-Visit(G, u)
1.    u.color = GRAY    //white vertex u has been discovered
2.    time = time + 1
3.    u.d = time
4.    for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.        if v.color = WHITE
6.            v.π = u
7.            DFS-Visit(G, v)
8.    u.color = BLACK     // Blacken u;  it is finished
9.    u.f = time
10.   time = time + 1
```



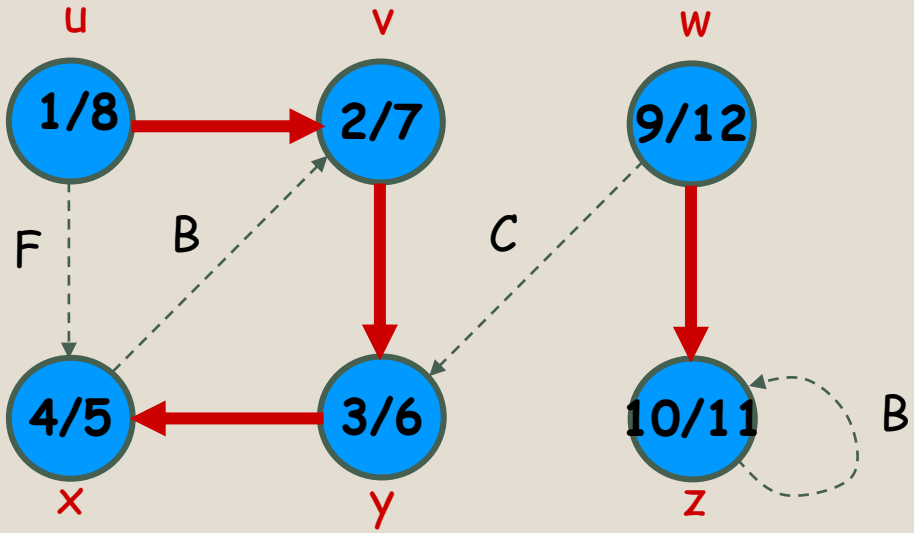
Depth-First Forest

- DFS creates a forest (subgraph induced by the red edges in the example)
  - Forest: An acyclic graph G that may be disconnected. (G has multiple trees)
- Tree edge: Edges in the forest
  - Edge that discovers new vertices
- Recall:
  - Back edge: edge from a descendant to an ancestor
  - Forward edge: edge from ancestor to a proper descendant



Depth-First Forest

- Recall:
  - Back edge: edge from a descendant to an ancestor
  - Forward edge: edge from ancestor to a **proper descendant**
  - Cross edge: no ancestor descendant relationship



- Vocabulary:
  - End-points of tree edges are predecessors and successors (e.g., v is processor of y)
  - u is an ancestor of v is we can go to v from u using tree edges only
  - u is a proper ancestor of v if we can go to v from u using at least two tree edges only

Depth-First Search (DFS)

```
DFS(G)
1. for each vertex u ∈ G.V
2.     u.color = white
3.     u.π = NIL
4. time = 0
5. for each vertex u ∈ G.V
6.     if u.color == white
7.         DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
1.  u.color = GRAY    //white vertex u has been discovered
2.  time = time + 1
3.  u.d = time
4.  for each v ∈ G.Adj[u]   // explore each edge (u, v)
5.      if v.color = WHITE
6.          v.π = u
7.          DFS-Visit(G, v)
8.  u.color = BLACK    // Blacken u;  it is finished
9.  u.f = time
10. time = time + 1
```

- DFS(G) lines 1-3 & 5-6 take $\Theta(V)$ time, line 4 takes $\Theta(1)$ time
- DFS-Visit is called once for each white vertex $u \in V$ when it's painted gray the first time. Lines 3-6 of DFS-Visit is executed $|Adj[u]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |Adj[u]| = \Theta(E)$
- Total running time of DFS is $\Theta(V + E)$

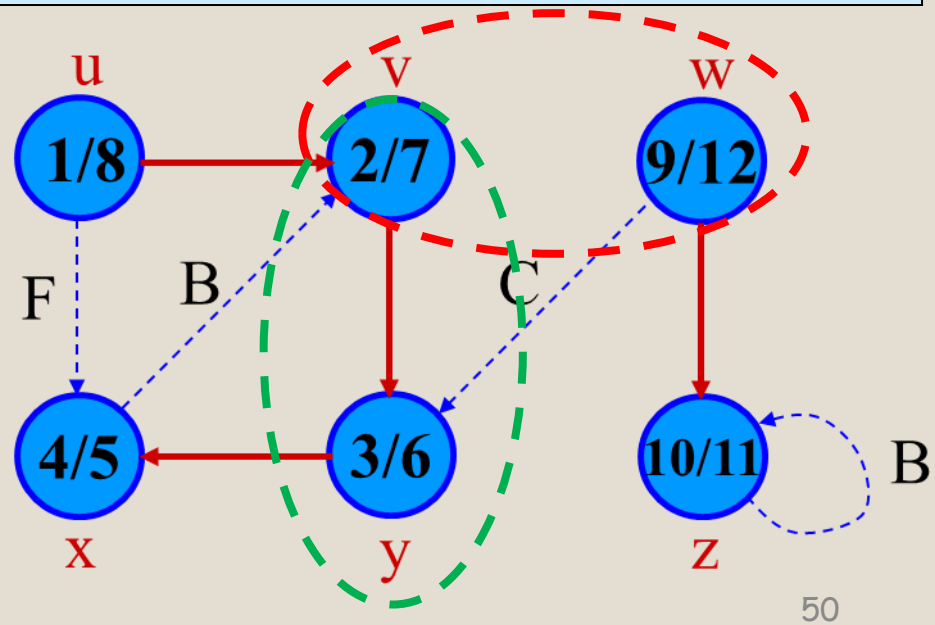
Parentheses Theorem

Theorem 20.7

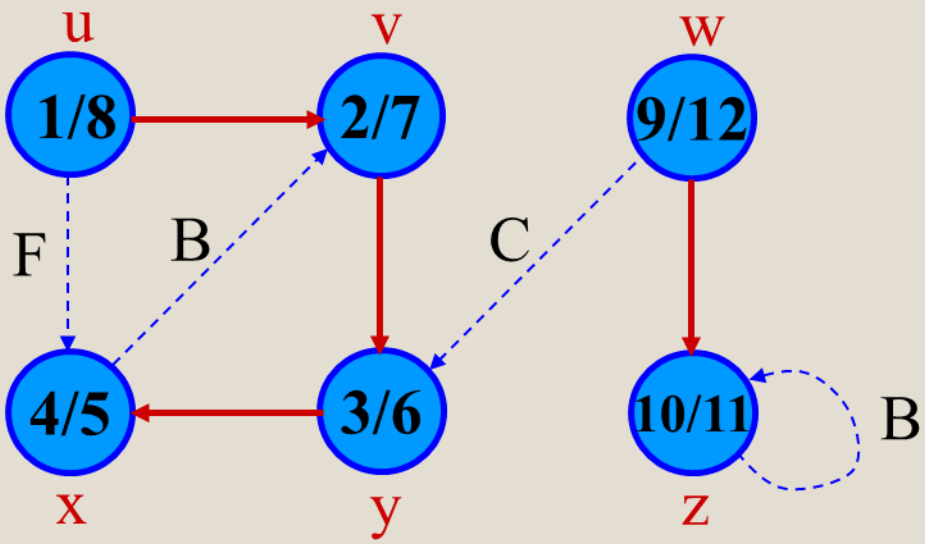
For all u, v, exactly one of the following holds:

1. The intervals $[u.d,\ u.f]$ and $[v.d,\ v.f]$ are entirely disjoint and neither u nor v is a descendant of the other.
2. $[v.d,\ v.f]$ is contained within $[u.d, u, f]$ and v is a descendant of u.
3. $[u.d,\ u.f]$ is contained within $[v.d,\ v.f]$ and u is a descendant of v.

- $u.d < v.d < u.f < v.f$ is impossible
- Like parentheses:
  - OK: ( ) [ ] ([ ]) [( )]
  - Not OK: ([ )] [( ])



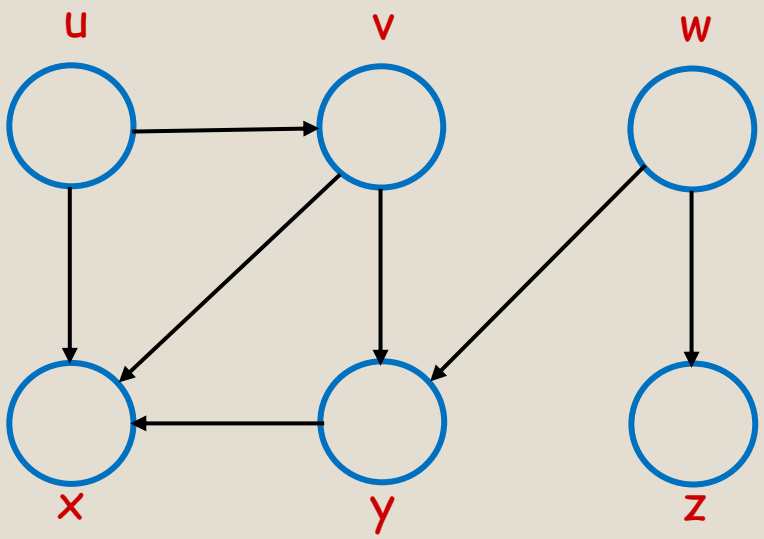
Parentheses Theorem

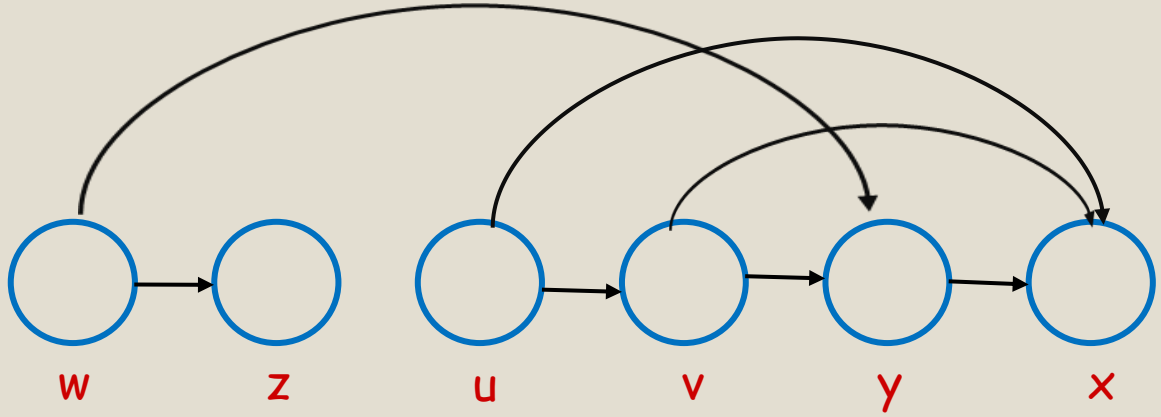


(u (v (y (x x) y) v u) (w (z z) w)

Topological Sorting

- Directed Acyclic Graph (DAG): directed graph with no cycle
  - Can be used to model dependency relationship
- Topological Sort: Ordering of vertices of a DAG so that for any edge (u, v), u appears before v in the ordering



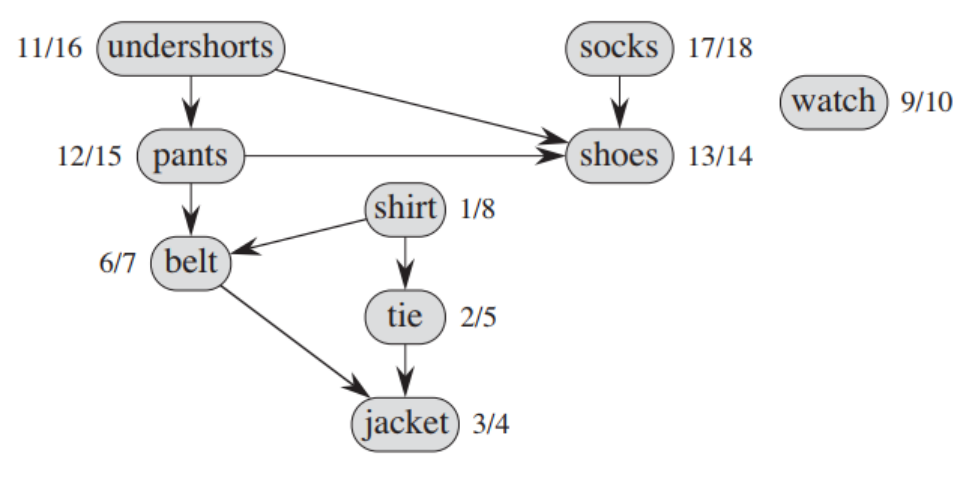


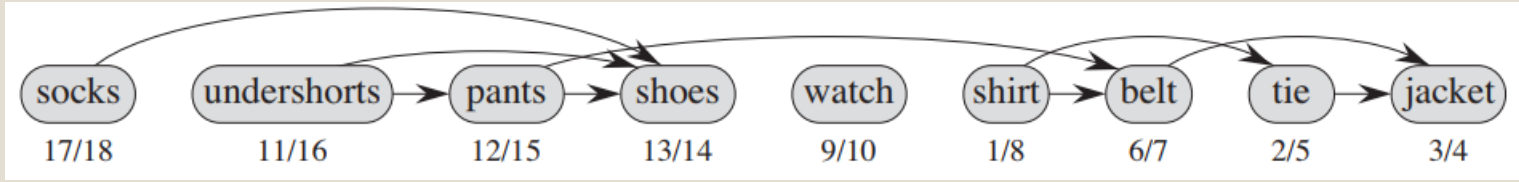
Topological Sorting

- Topological Sort: Ordering of vertices of a DAG so that for any edge (u, v), u appears before v in the ordering

TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finish 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





Thank You!