Elementary Graph Algorithms

Lecture 11

Graphs

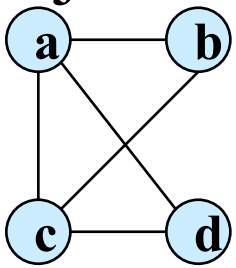
- *Graph* $G = (V, E)$
  - » V = set of vertices
  - » E = set of edges $\subseteq (V \times V)$
- Types of graphs
  - » Undirected: edge $(u, v) = (v, u)$; for all v, $(v, v) \notin E$ (No self loops.)
  - » Directed: (u, v) is edge from u to v, denoted as $u \rightarrow v$. Self loops are allowed.
  - » Weighted: each edge has an associated weight, given by a weight function $w : E \rightarrow \mathbf{R}$.
  - » Dense: $|E| \approx |V|^2$.
  - » Sparse: $|E| << |V|^2$.
- $|E| = O(|V|^2)$

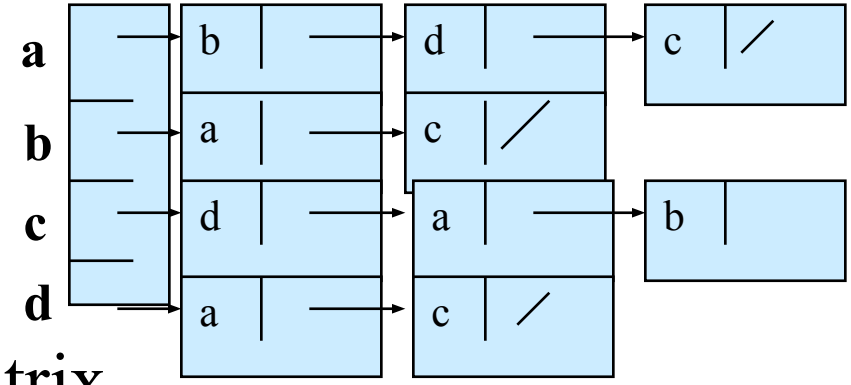
Graphs

- 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.
- If 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.

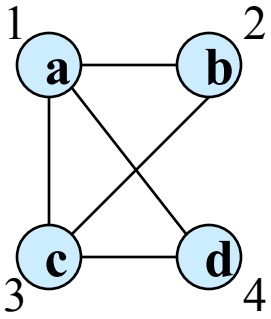
Representation of Graphs

- Two standard ways.
  - » Adjacency Lists.





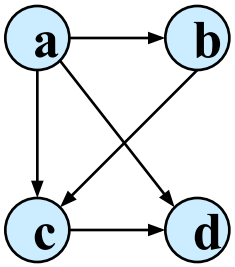
  - » Adjacency Matrix.

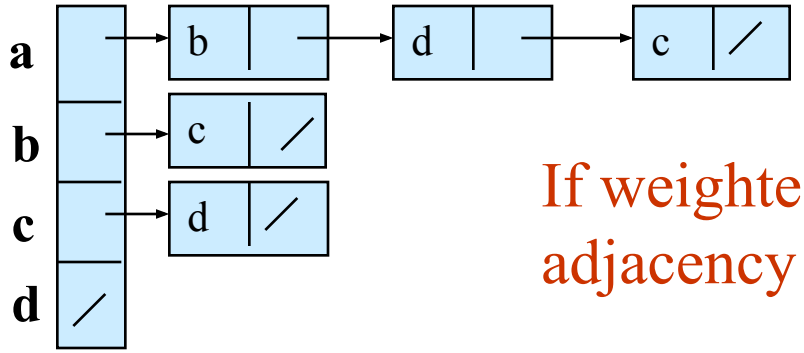


	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

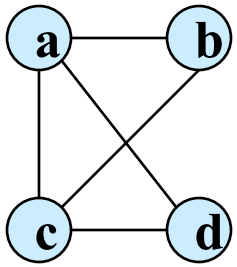
Adjacency Lists

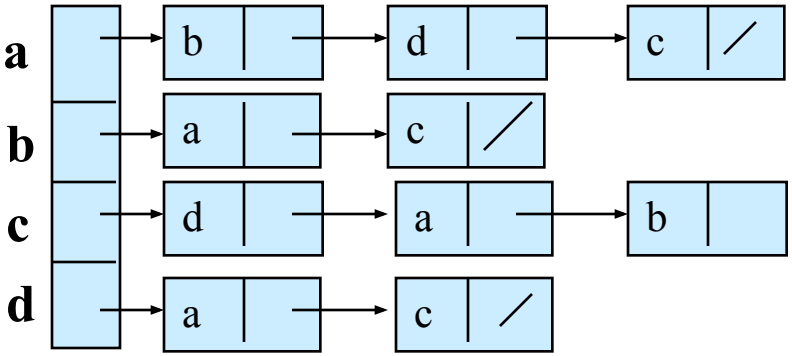
- Consists of an array Adj of $|V|$ lists.
- One list per vertex.
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to u.





If weighted, store weights also in adjacency lists.





Storage Requirement

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

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

No. of edges leaving v

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

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

No. of edges incident on v. Edge (u,v) is incident on vertices u and v.

  - » Total storage: $\Theta(V+E)$

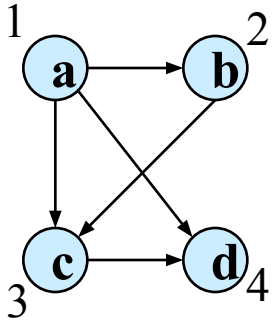
Pros and Cons: adj list

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

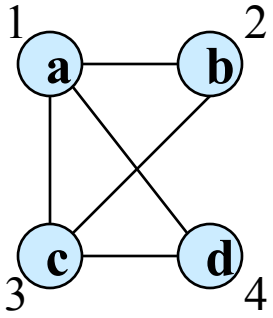
Adjacency Matrix

- $|V| \times |V|$ matrix A.
- Number vertices from 1 to $|V|$ in some arbitrary manner.
- A is then given by:

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



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



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

$A = A^{\mathrm{T}}$ for undirected graphs.

Space and Time

- **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-searching Algorithms

- 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).

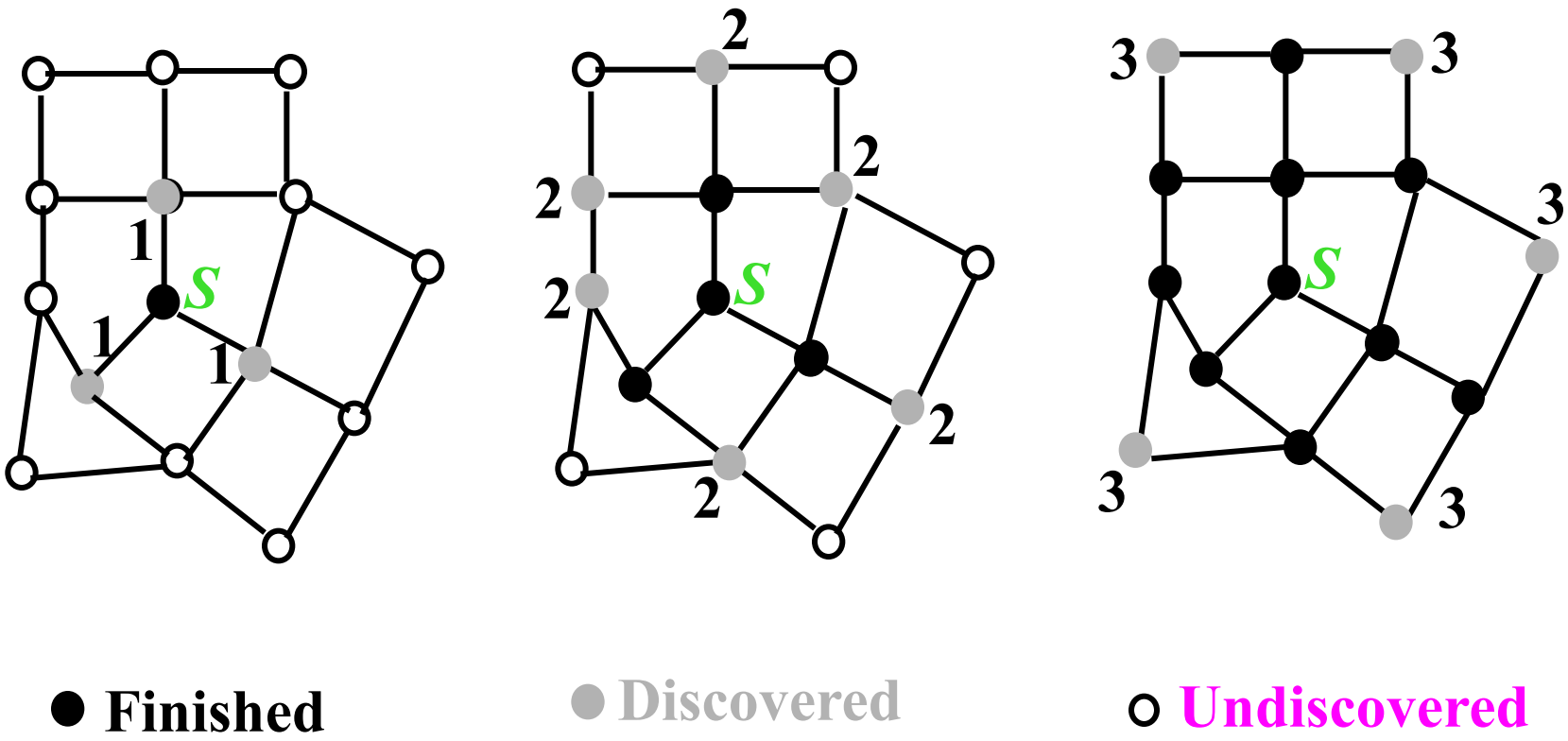
Breadth-first Search

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

Breadth-first Search

- 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 required only to reason about the algorithm. Can be implemented without colors.

BFS for Shortest Paths



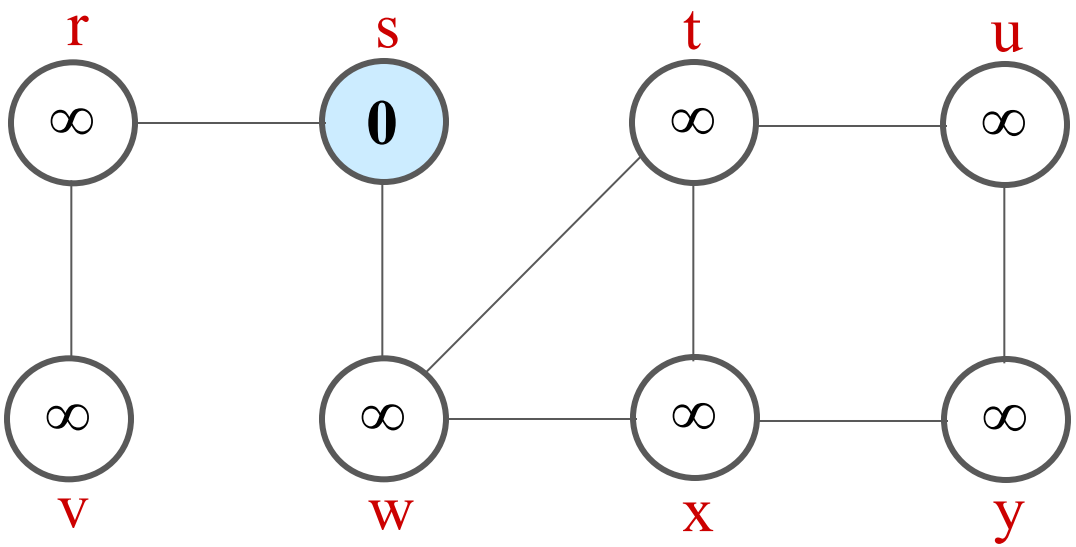
BFS(G,s)

```
1.  for each vertex u in V[G] − {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 ← Φ
9.  enqueue(Q,s)
10. while Q ≠ Φ
11.       do u ← dequeue(Q)
12.          for each v in Adj[u]
13.             do if color[v] = white
14.                then color[v] ← gray
15.                     d[v] ← d[u] + 1
16.                     π[v] ← u
17.                     enqueue(Q,v)
18.          color[u] ← black
```

white: undiscovered
gray: discovered
black: finished

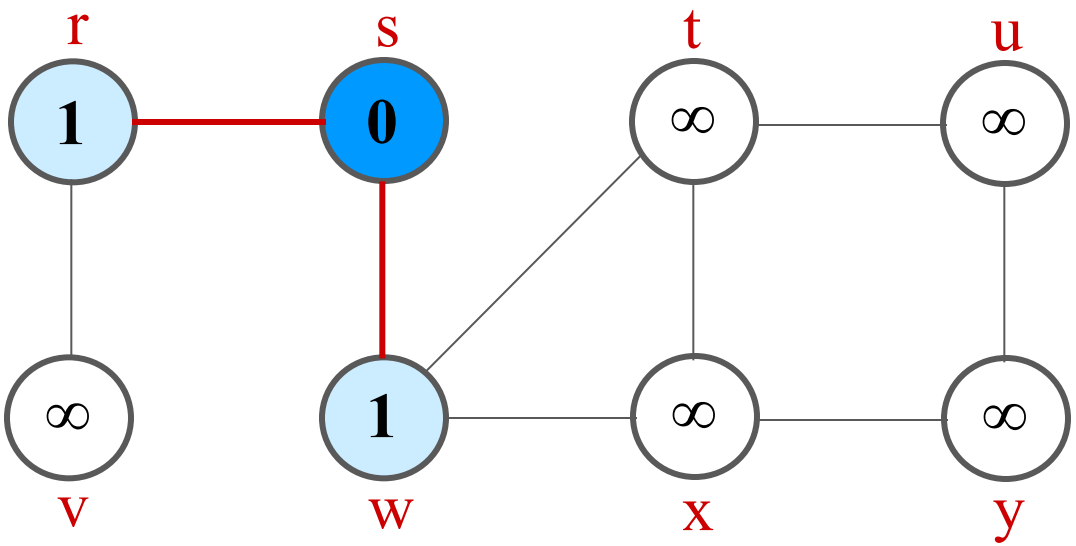
Q: a queue of discovered vertices
color[*v*]: color of v
d[*v*]: distance from s to v
π[*u*]: predecessor of v

Example (BFS)



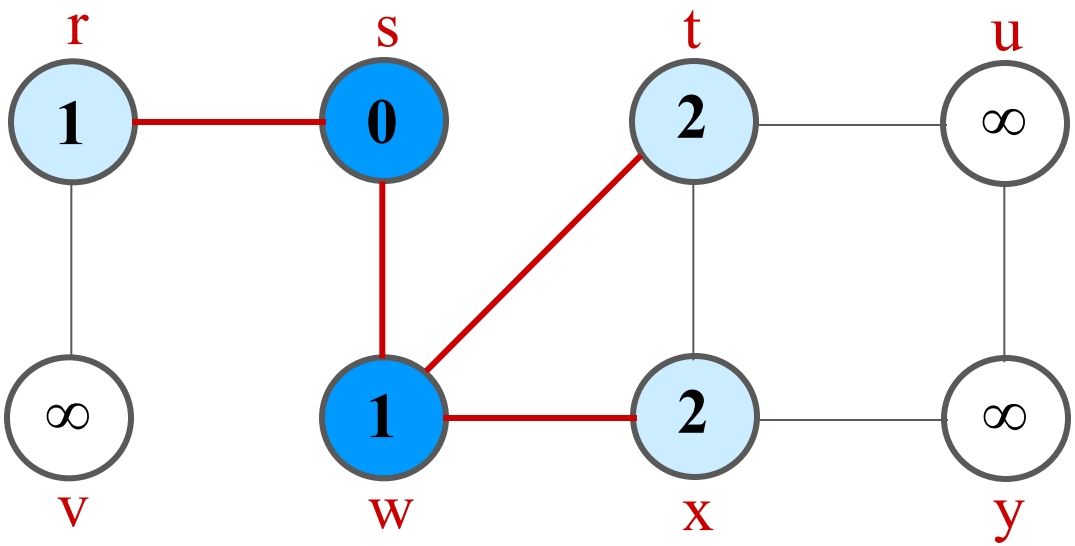
Q: s
0

Example (BFS)



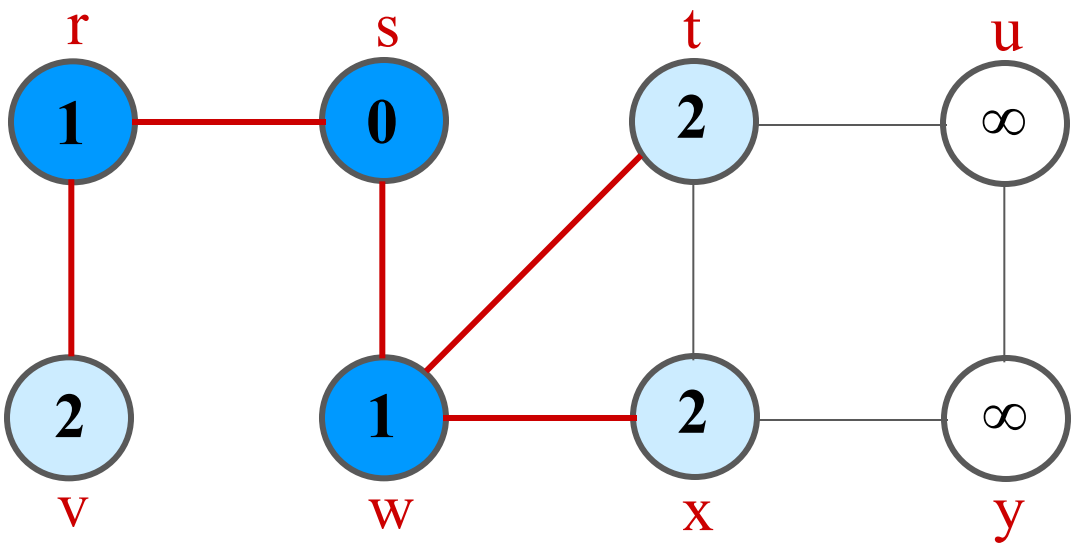
Q:	w	r
	1	1

Example (BFS)



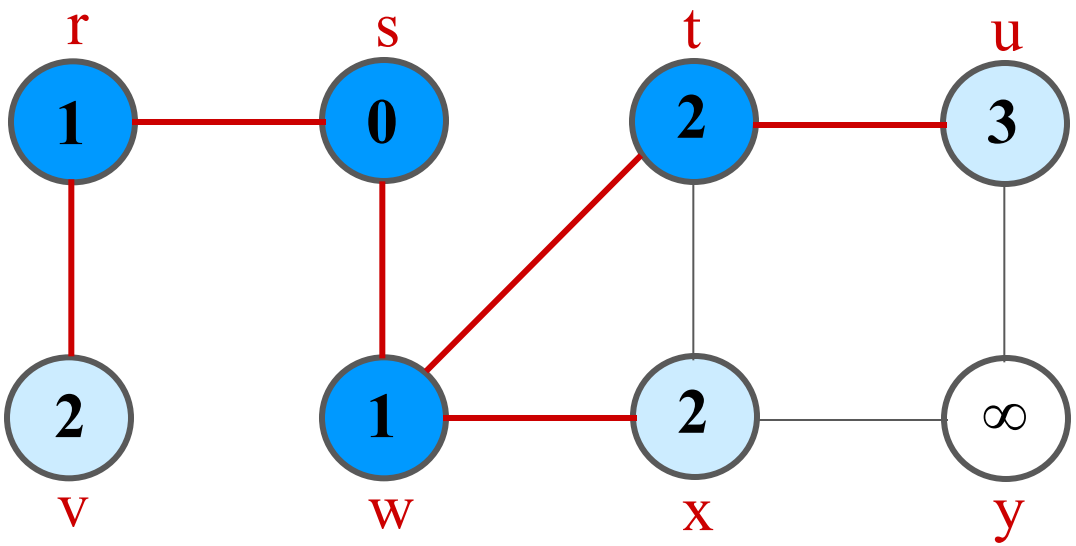
Q:	r	t	x
	1	2	2

Example (BFS)



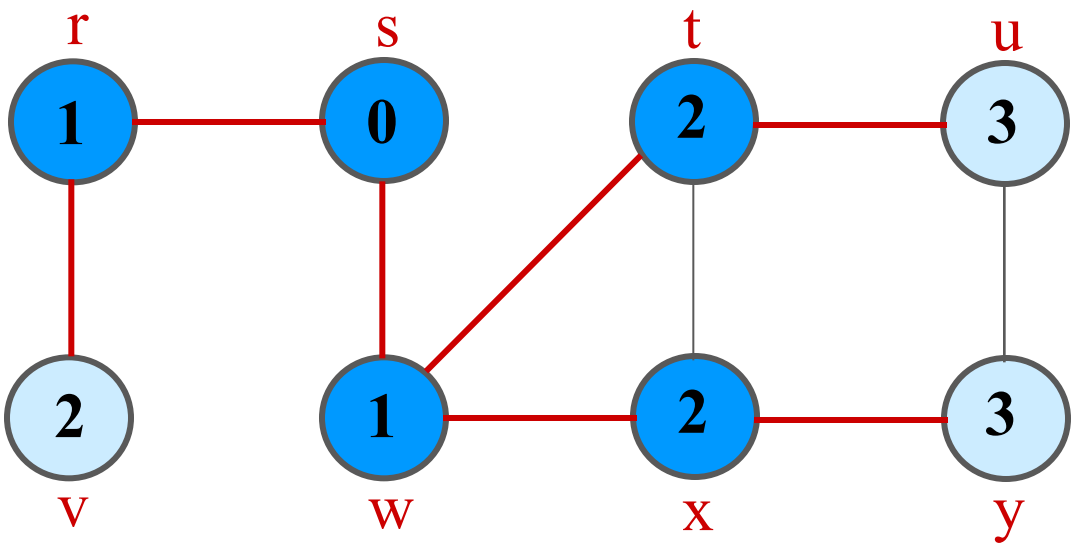
Q:	t	x	v
	2	2	2

Example (BFS)



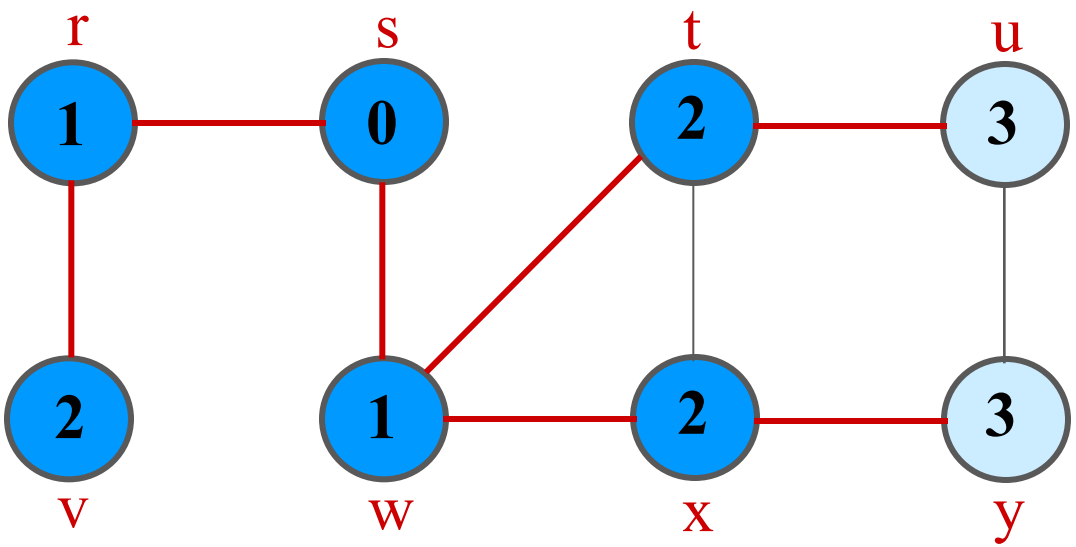
Q:	x	v	u
	2	2	3

Example (BFS)



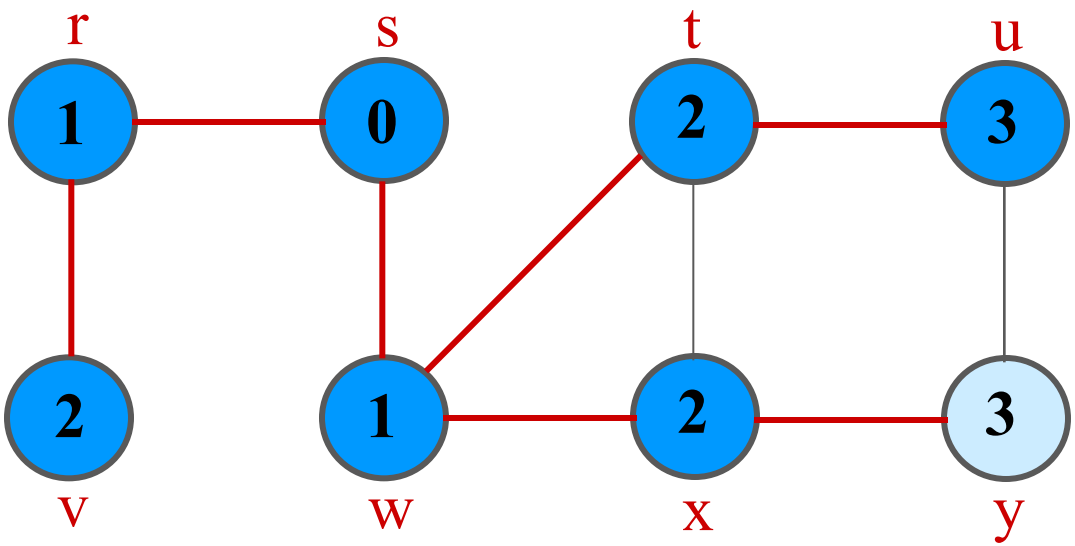
Q:	v	u	y
	2	3	3

Example (BFS)



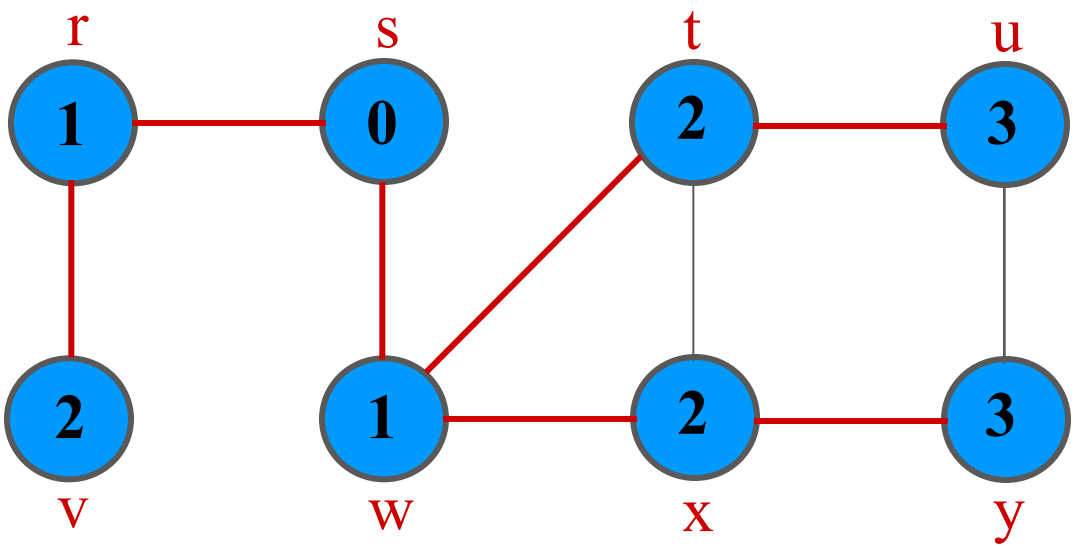
Q:	u	y
	3	3

Example (BFS)



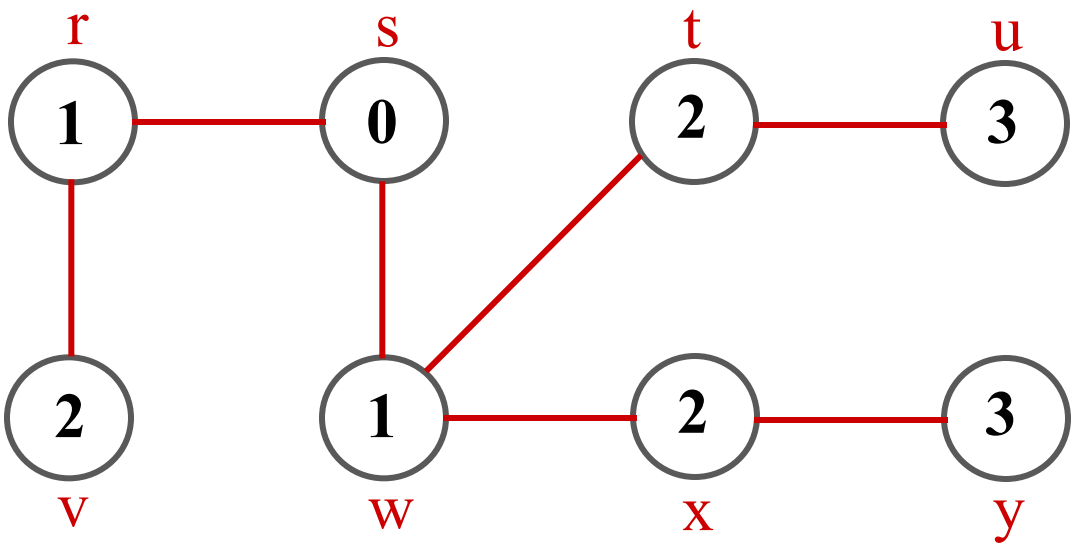
Q: y
3

Example (BFS)



Q: ∅

Example (BFS)



BF Tree

Analysis of BFS

- It takes $O(V)$ to initialize the distance and predecessor for each vertex
- Each vertex is visited at most one time, because only the first time that it is reached is its distance `null`, and so each vertex is enqueued at most one time.
- Since we examine the edges incident on a vertex only when we visit from it, each edge is examined at most twice, once for each of the vertices it's incident on.
- Thus, breadth-first search spends O(V+E) times in visiting vertices

Breadth-first Tree

- For a graph $G = (V, E)$ with source s, the **predecessor subgraph** of G is $G_\pi = (V_\pi, E_\pi)$ where
  - » $V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$
  - » $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$
- The predecessor subgraph G_π is a **breadth-first tree** if:
  - » V_π 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_π that is also a shortest path from s to v in G.
- The edges in E_π are called **tree edges**.
  $|E_\pi| = |V_\pi| - 1$.

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

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - » 2 **timestamps** on each vertex. Integers between 1 and 2|V|.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***finishing time*** (v turns from gray to black)
  - » $\pi[v]$: 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.

Pseudo-code

DFS(G)

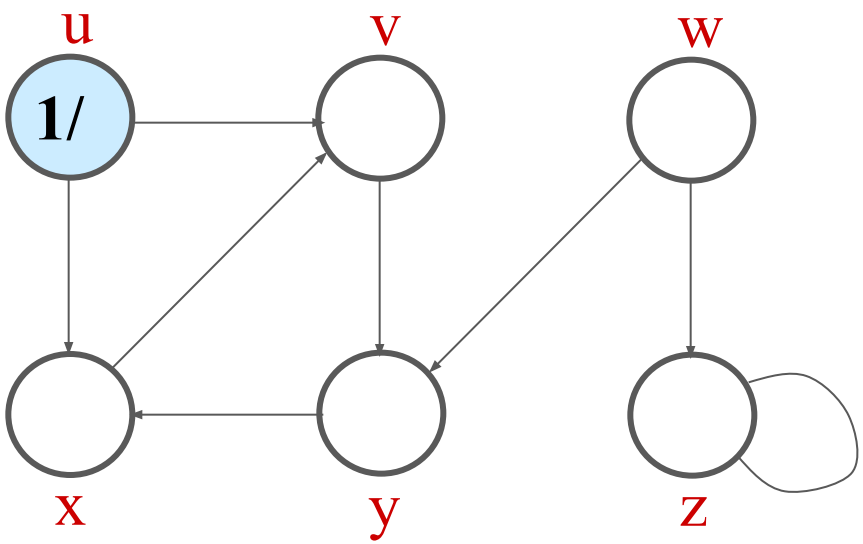
```
1. for each vertex u ∈ V[G]
2.     do color[u] ← white
3.        π[u] ← NIL
4. time ← 0
5. for each vertex u ∈ V[G]
6.     do if color[u] = white
7.        then DFS-Visit(u)
```

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

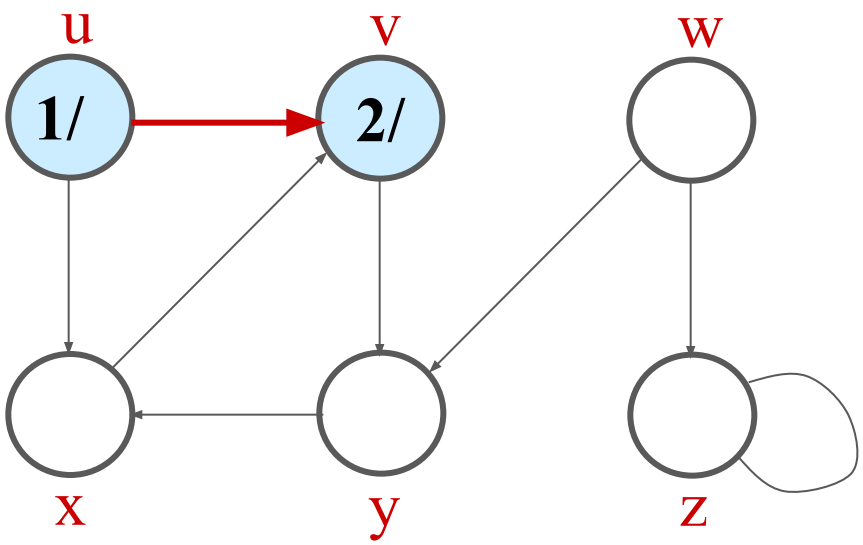
DFS-Visit(u)

```
1.  color[u] ← GRAY   ▽ White vertex u
    has been discovered
2.  time ← time + 1
3.  d[u] ← time
4.  for each v ∈ Adj[u]
5.      do if color[v] = WHITE
6.          then π[v] ← u
7.               DFS-Visit(v)
8.  color[u] ← BLACK    ▽ Blacken u;
    it is finished.
9.  f[u] ← time ← time + 1
```

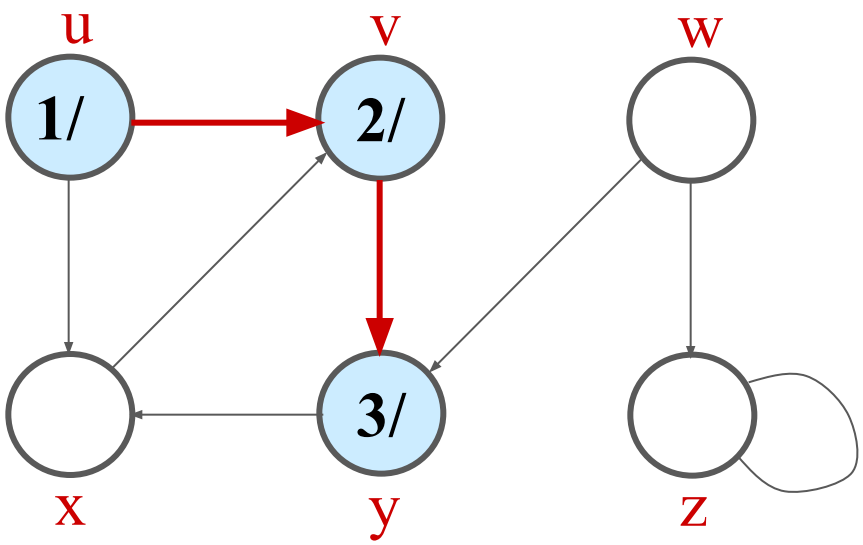
Example (DFS)



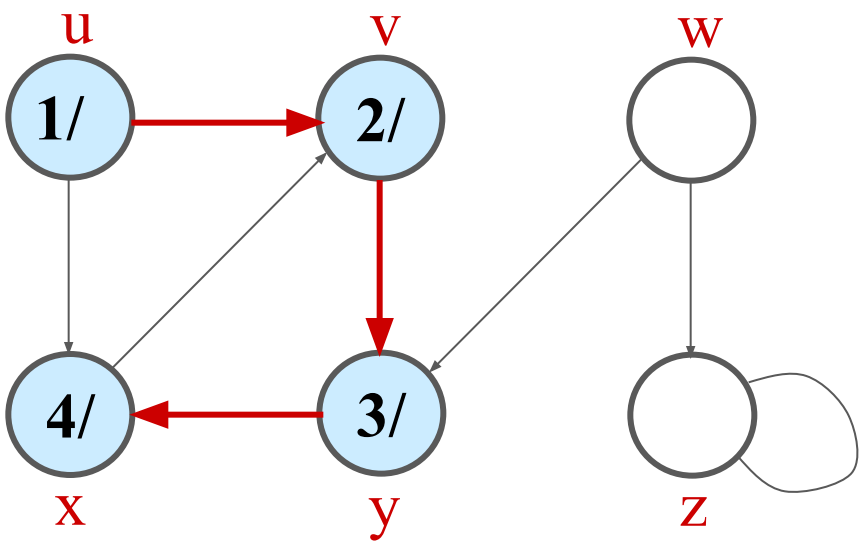
Example (DFS)



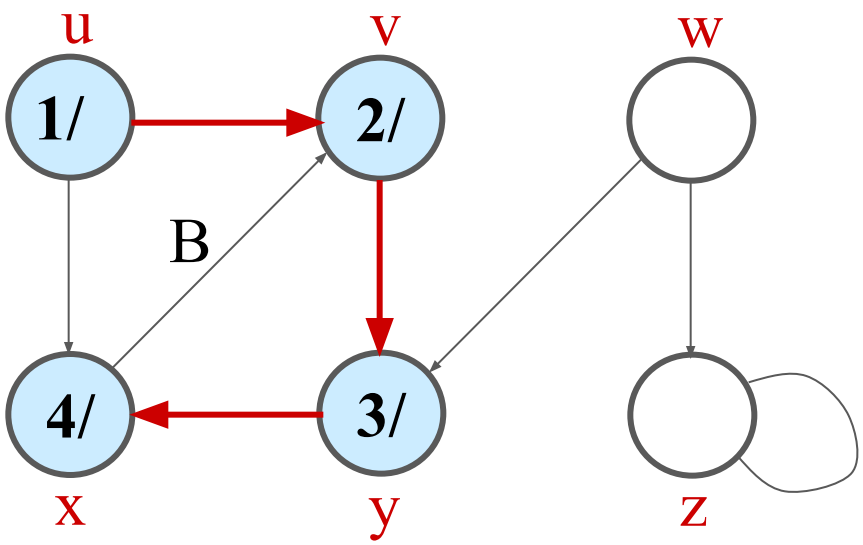
Example (DFS)



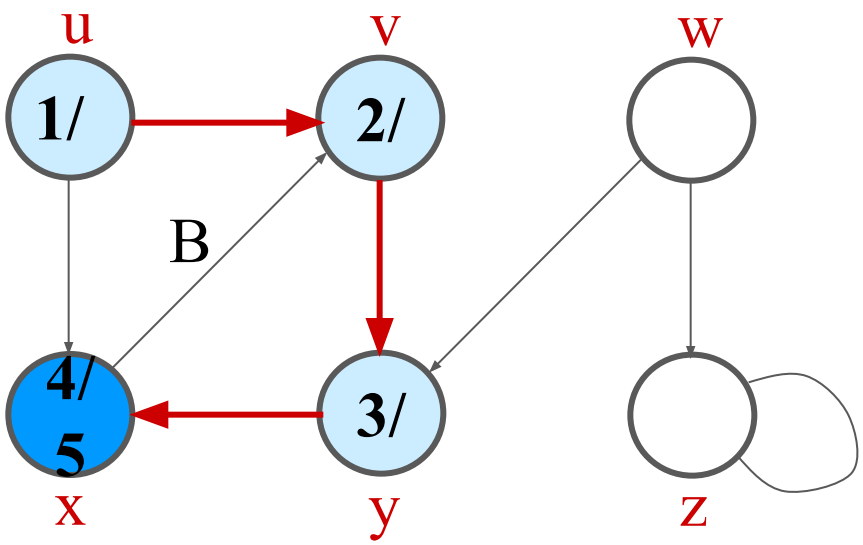
Example (DFS)



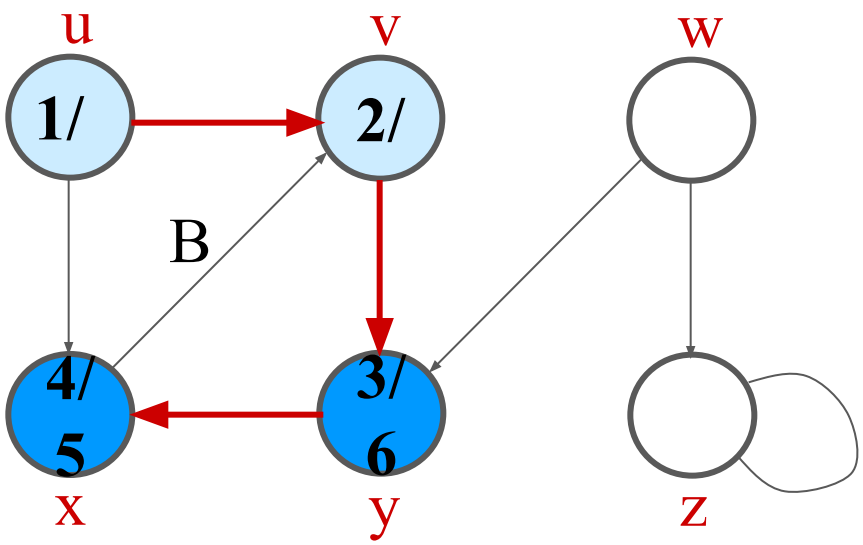
Example (DFS)



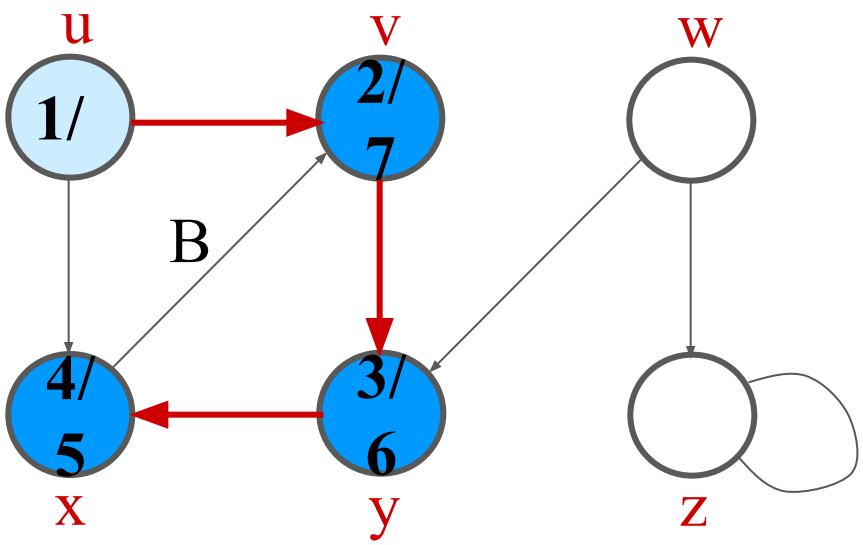
Example (DFS)



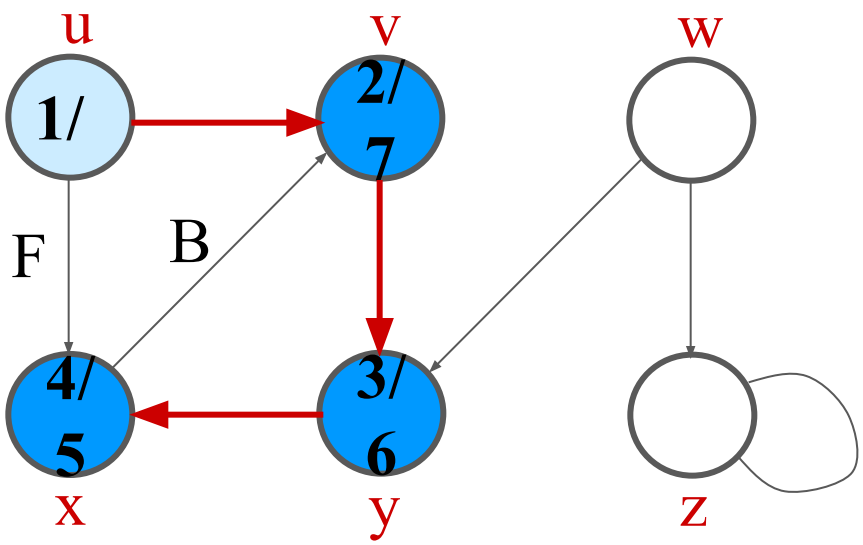
Example (DFS)



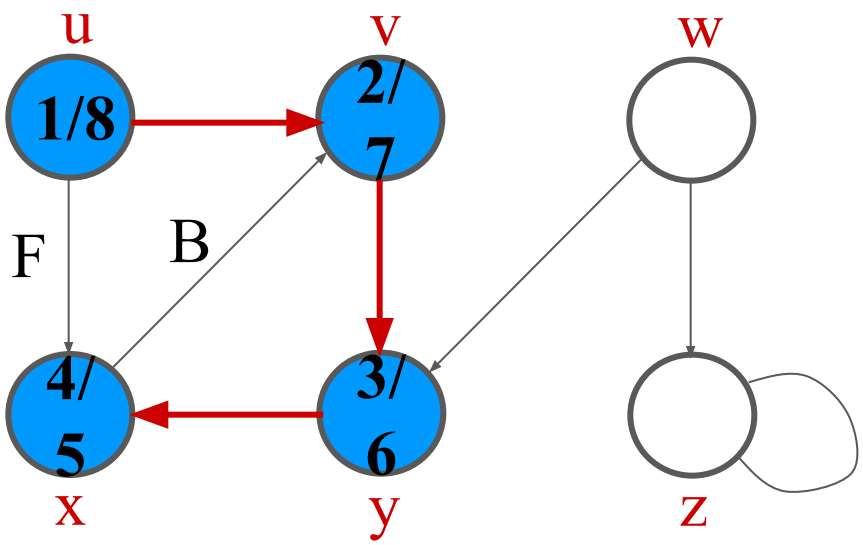
Example (DFS)



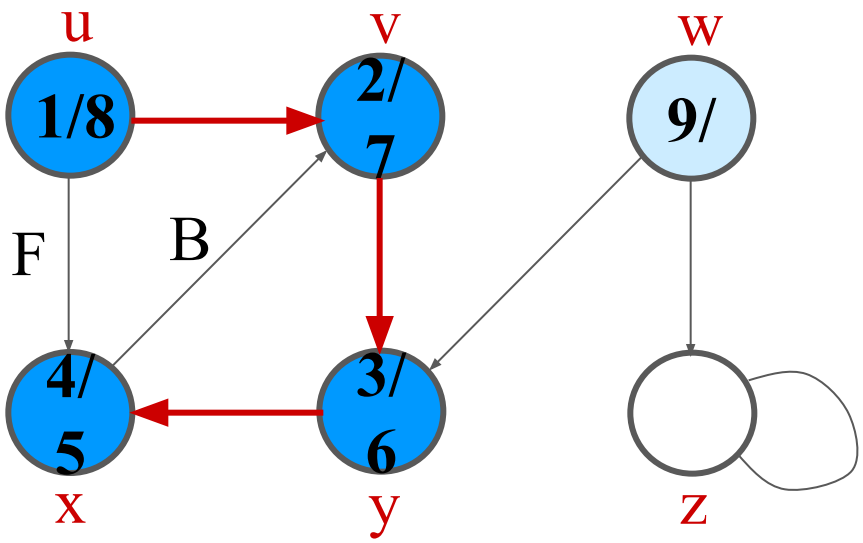
Example (DFS)



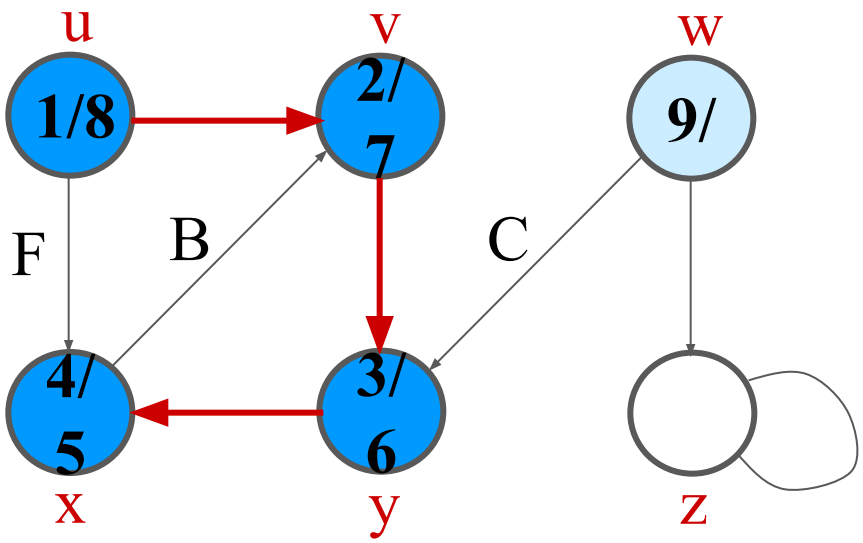
Example (DFS)



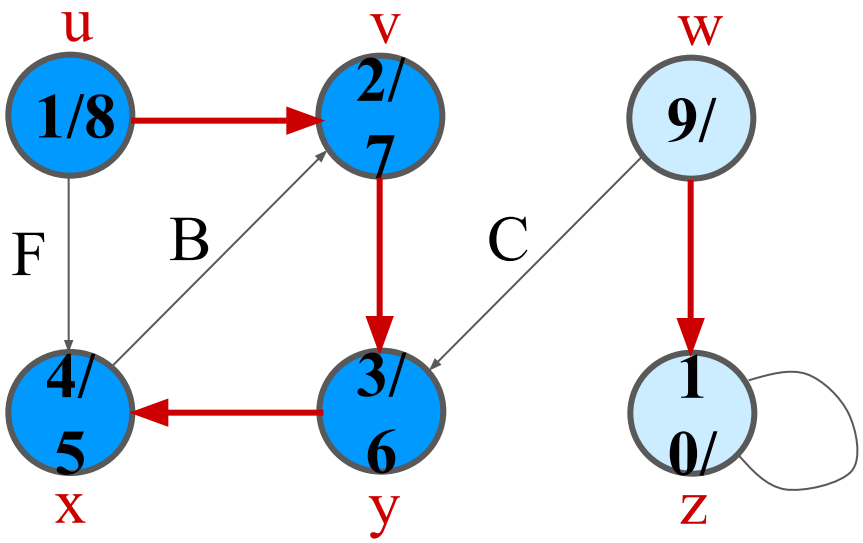
Example (DFS)



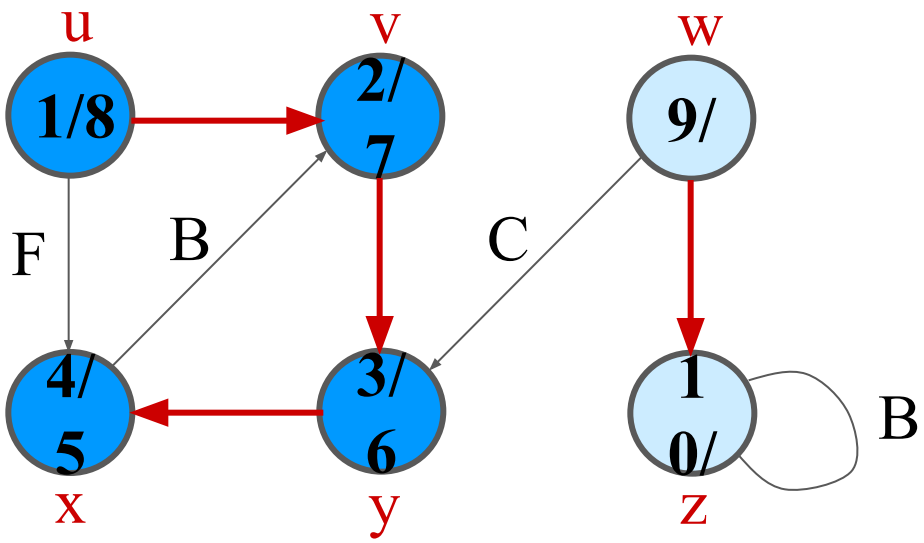
Example (DFS)



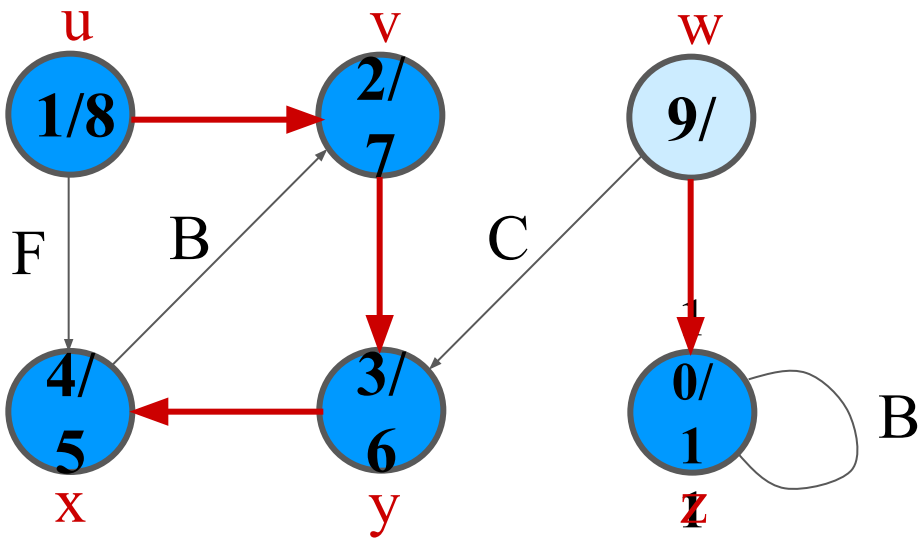
Example (DFS)



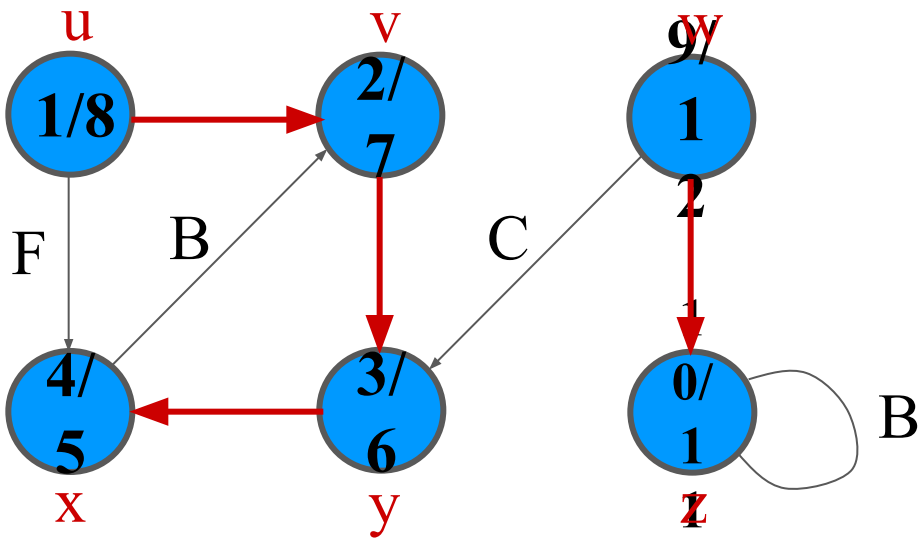
Example (DFS)



Example (DFS)



Example (DFS)



Pseudo-code

DFS(G)

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

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

DFS-Visit(u)

```
1.  color[u] ← GRAY   ▽ White vertex u
    has been discovered
2.  time ← time + 1
3.  d[u] ← time
4.  for each v ∈ Adj[u]
5.      do if color[v] = WHITE
6.          then π[v] ← u
7.               DFS-Visit(v)
8.  color[u] ← BLACK    ▽ Blacken u;
    it is finished.
9.  f[u] ← time + 1
```

Analysis of DFS

- Loops on lines 1-2 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.
- DFS-Visit is called once for each white vertex $v \in V$ when it's painted gray the first time. Lines 3-6 of DFS-Visit is executed $|\text{Adj}[v]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$
- Total running time of DFS is $\Theta(V+E)$.

Depth-First Trees

- Predecessor subgraph defined slightly different from that of BFS.
- The predecessor subgraph of DFS is $G_\pi = (V, E_\pi)$ where $E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\}$.
  - » How does it differ from that of BFS?
  - » The predecessor subgraph G_π forms a *depth-first forest* composed of several *depth-first trees*. The edges in E_π are called *tree edges*.

Definition:

Forest: An acyclic graph G that may be disconnected.

Classification of Edges

- **Tree edge:** in the depth-first forest. Found by exploring (u, v).
- **Back edge:** (u, v), where u is a descendant of v (in the depth-first tree).
- **Forward edge:** (u, v), where v is a descendant of u, but not a tree edge.
- **Cross edge:** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

Theorem:
In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

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*):
Then *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 the predecessor of *v* in *c*.

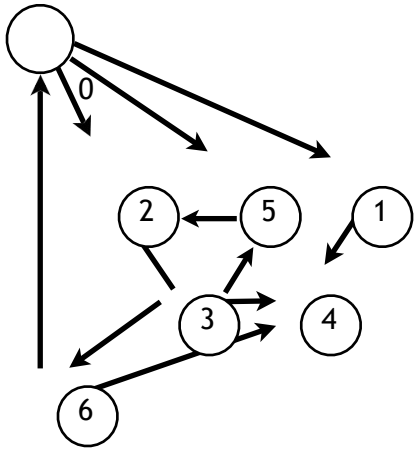
- Upon discovering *v* the whole cycle from *v* to *u* is white
- We visit all nodes reachable on this white path before DFS(*v*) returns, i.e., vertex *u* becomes a descendant of *v*
- Thus, (*u,v*) is a back edge

Thus, we can verify whether *G* is a DAG using DFS

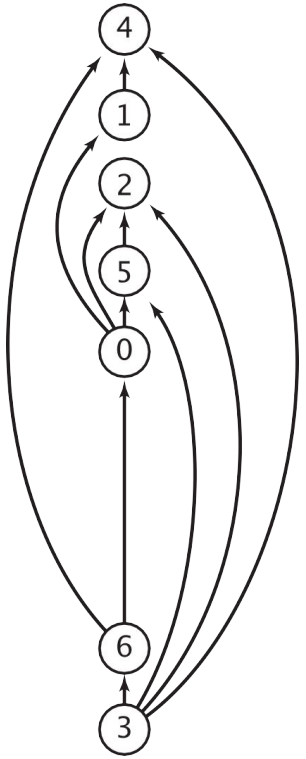
Precedence scheduling

Goal. Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

Digraph model. vertex = task; edge = precedence constraint.



precedence constraint graph



feasible schedule

Topological Sorting

- 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

Steps:

- Run depth-first search.
- Return vertices in reverse postorder

- Cycles make topological sort impossible.
- Select any node with no in-edges
  - print it
  - delete it
  - and delete all the edges leaving it
- Repeat

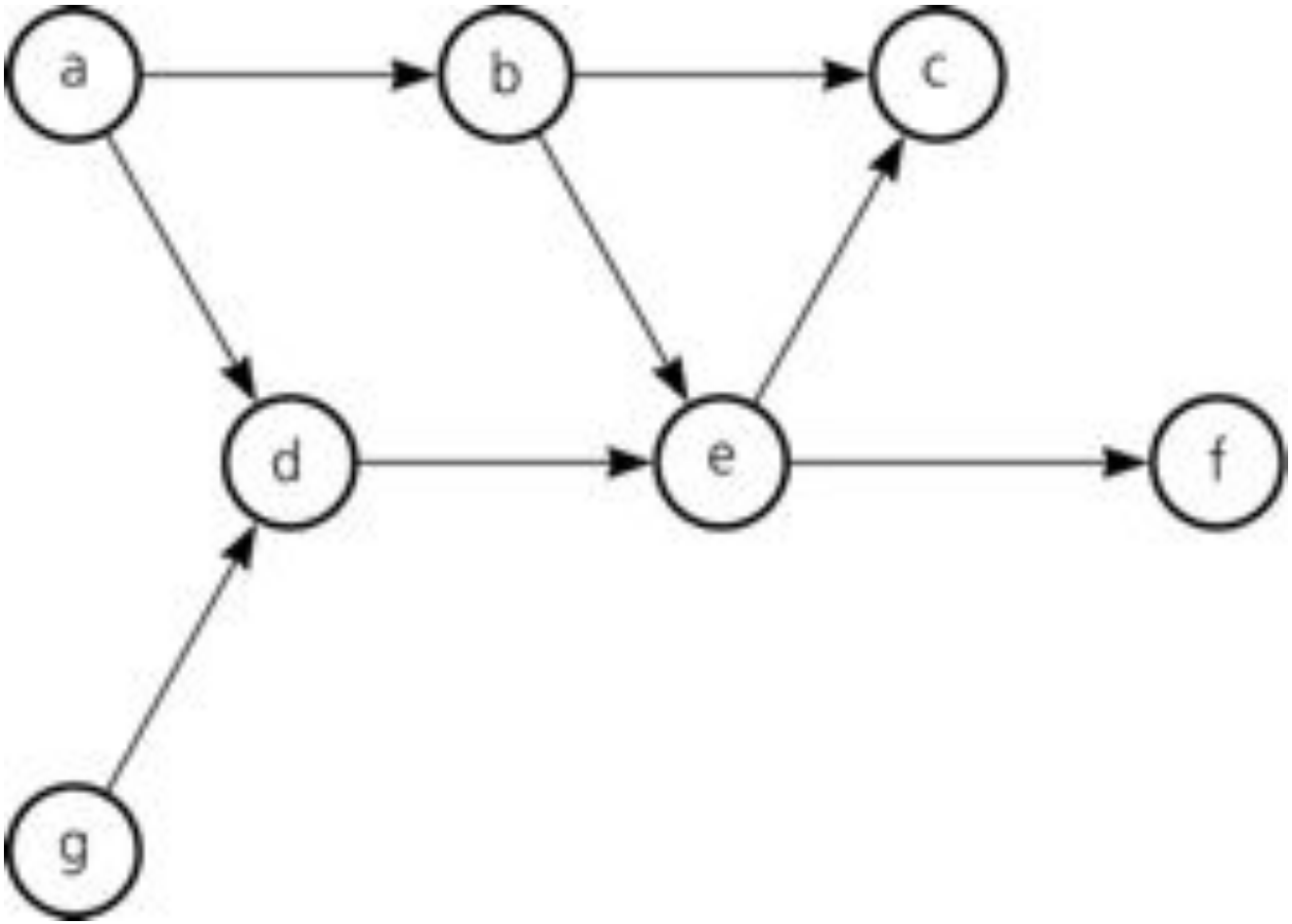
Topological Sorting

The following algorithm topologically sorts a

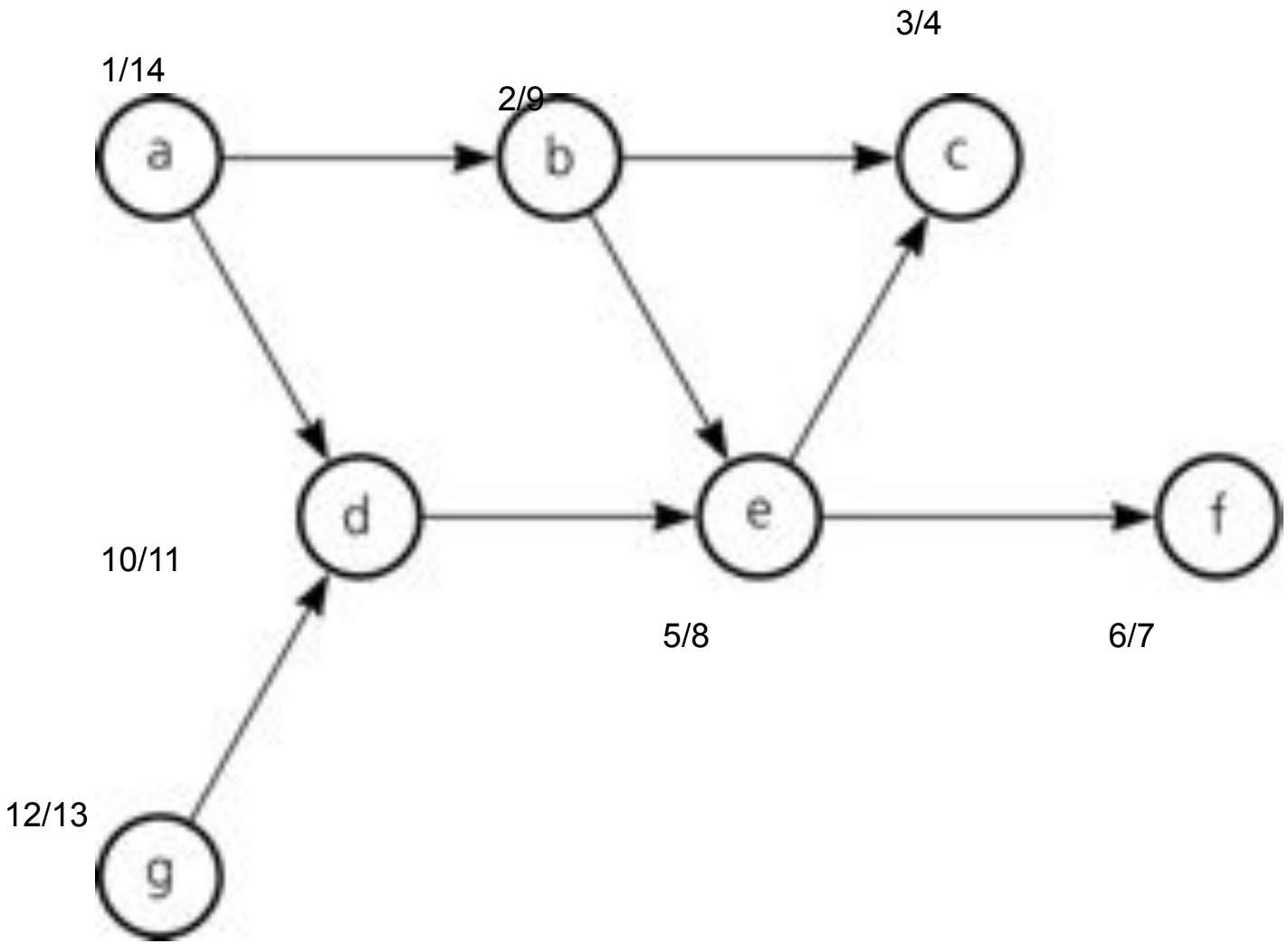
DAG The linked lists comprises a total ordering

```
TopologicalSort(G)
   Call DSF(G) to compute
      v.endtime for each vertex v
   As each vertex is finished,
   insert it at the beginning of a linked list  Return
   the linked list of vertices
```

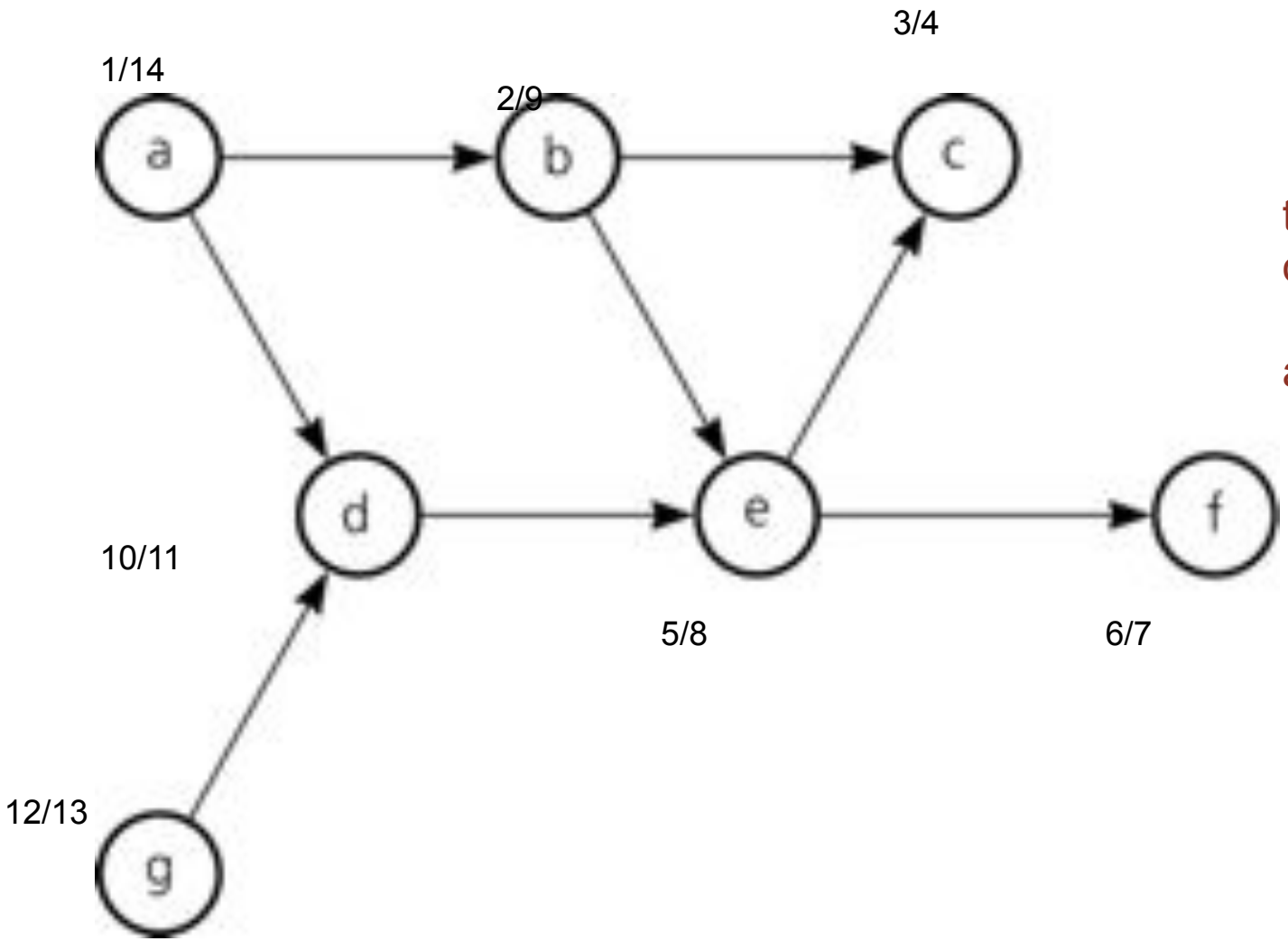
Perform Topological sorting :



Perform Topological sorting :



postorder:
c f e b d g a



topological order:

a g d b e f c

Topological sort in a DAG: correctness proof

Proposition. Reverse DFS postorder of a DAG is a topological order.

Pf. Consider any edge v→w. When dfs(v) is called:

Case 1:

dfs(w) has already been called and returned.

Thus, w was done before v.

Case 2:

dfs(w) has not yet been called.

dfs(w) will get called directly or indirectly by dfs(v) and will finish before dfs(v).

Thus, w will be done before v.

Case 3:

dfs(w) has already been called,

but has not yet returned.
Can't happen in a DAG: function call stack contains path from w to v, so v→w would complete a cycle.

Topological Sorting: Running Time

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 $O(V+E)$