Elementary Graph Algorithms

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

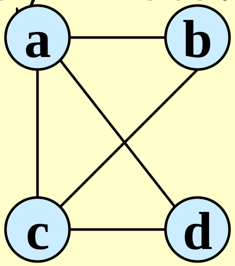
- If $(u, v) \in E$, then vertex v is adjacent to vertex u.
- Adjacency relationship is:
  - » Symmetric if G is undirected.
  - » Not necessarily so if G is directed.
- An edge (u, v) is said to be incident to u and v
- Degree of a vertex in undirected graph is the number of edges incident to it.
- In-degree of a vertex in directed graph is the number of edges entering it.
- Out-degree of a vertex in directed graph is the number of edges leaving it.

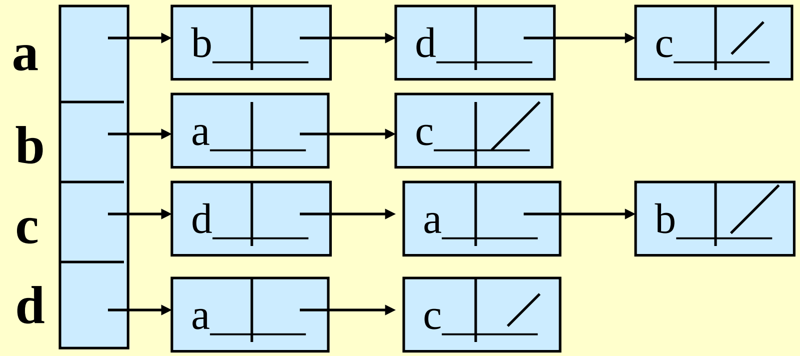
Graphs

- A path from u_0 to u_k is a sequence of vertices $<u_0, u_1, u_2, u_3, \ldots, u_k>$ such that $(u_i, u_{i+1}) \in E, \forall i = 0, 1, 2, \ldots, (k-1)$. The length of the path is k.
- If there is a path exist between u_i and u_j we say that u_j is reachable from u_i. It is a cycle if $u_i = u_j$.
- 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.

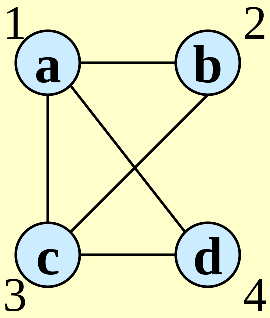
Representation of Graphs

- Two standard ways.
  - » Adjacency Lists.





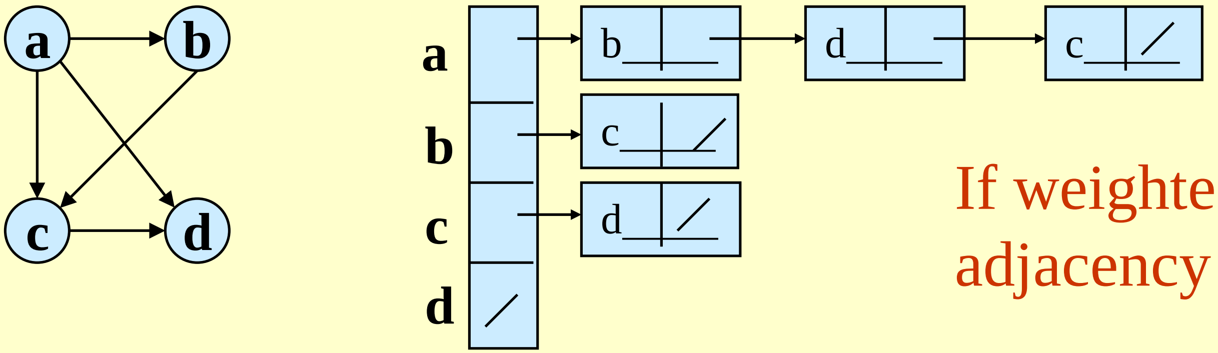
  - » Adjacency Matrix.



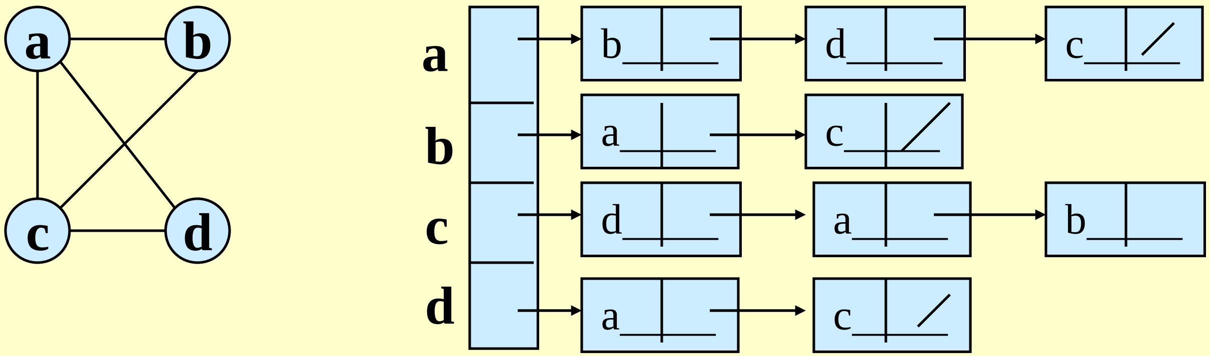
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

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: $\Theta(V+E)$

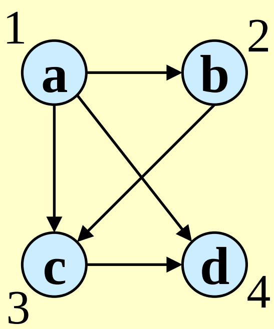
Pros and Cons: adj list

- 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.

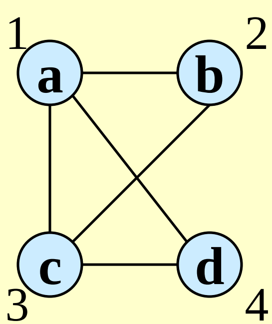
Adjacency Matrix

- $|V| \times |V|$ 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 $s \rightsquigarrow v$.
    - • u is v's predecessor.
  - » Builds breadth-first tree with root s that contains all reachable vertices.

Definitions:

Path between vertices u and v: Sequence of vertices $(v_1, v_2, \ldots, v_k)$ such that $u=v_1$ and $v=v_k$, and $(v_i,v_{i+1}) \in E$, for all $1 \le i \le k-1$.

Length of the path: Number of edges in the path.

Path is simple if no vertex is repeated.

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.

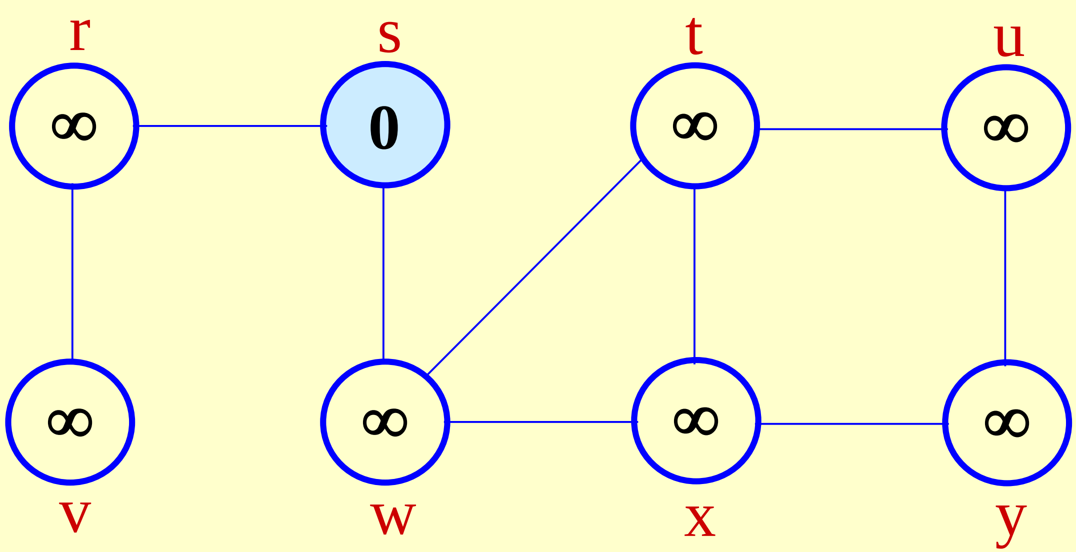
BFS(G,s)

```
for each vertex u in V[G] – {s}
        do color[u] ← white
           d[u] ← ∝
           π[u] ← nil
color[s] ← gray
d[s] ← 0
π[s] ← nil
Q ← Φ
enqueue(Q,s)
while Q ≠ Φ
        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
```

white: undiscovered
gray: discovered
black: finished

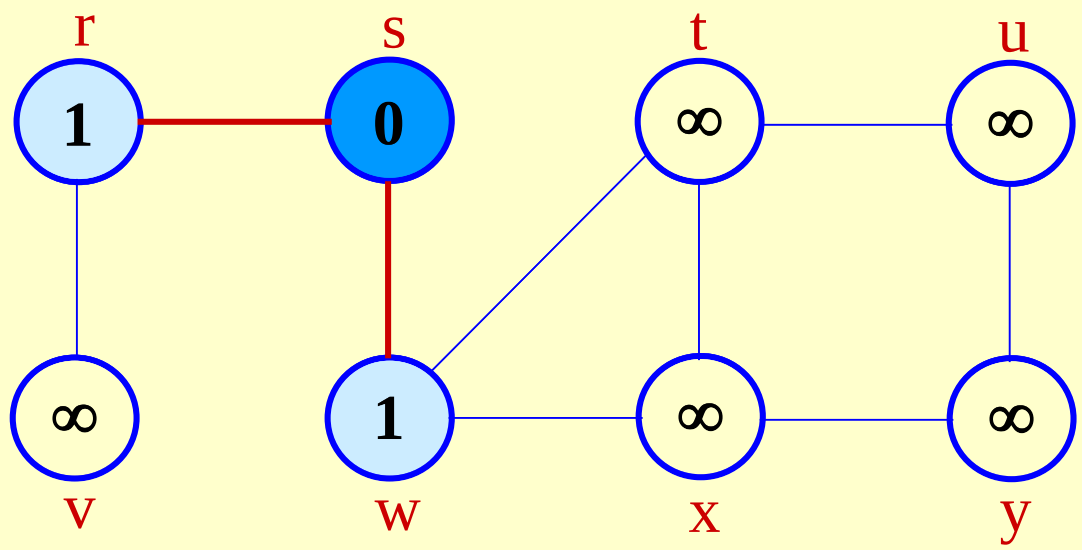
Q: a queue of discovered vertices
color[*v*]: color of v
d[*v*]: distance from s to v
π[*u*]: predecessor of v

Example (BFS)



Q: s
0

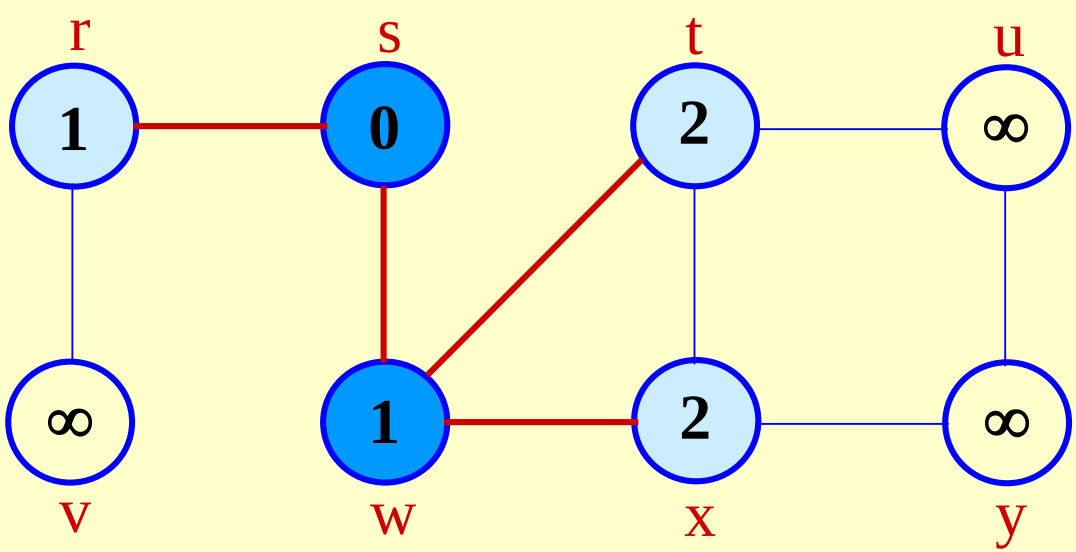
Example (BFS)



Q: w r

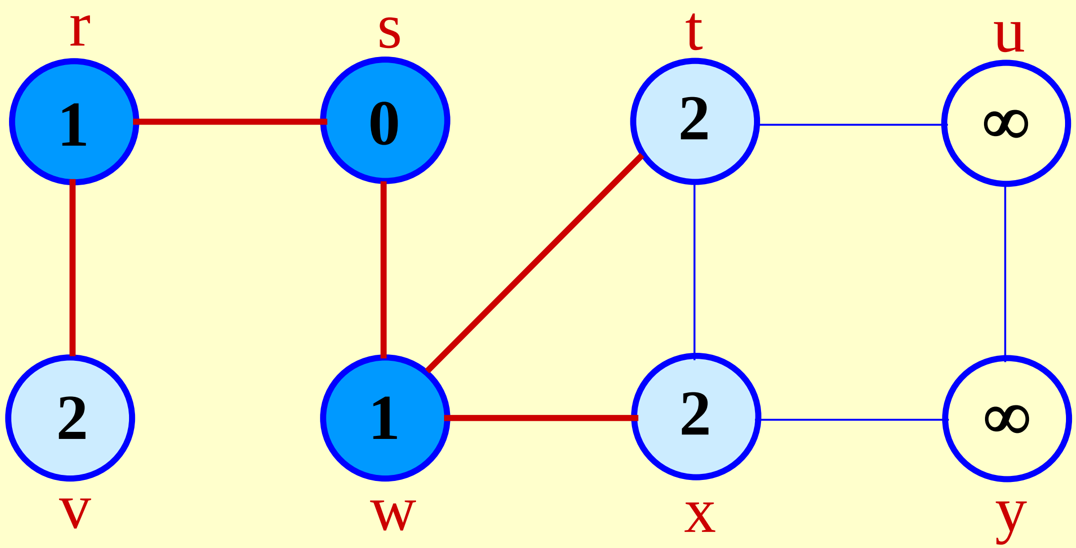
1 1

Example (BFS)



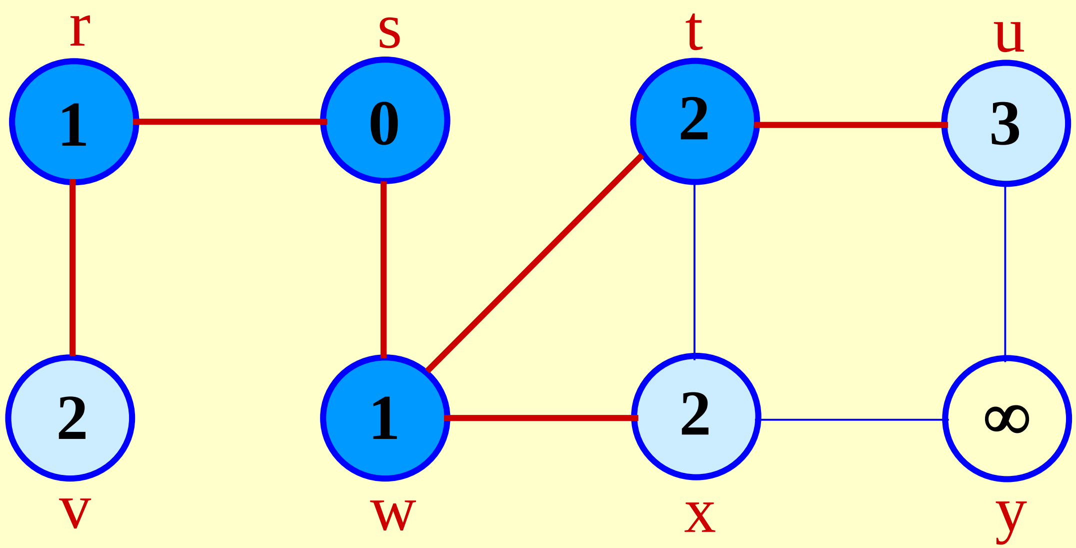
Q:	r	t	x
	1	2	2

Example (BFS)



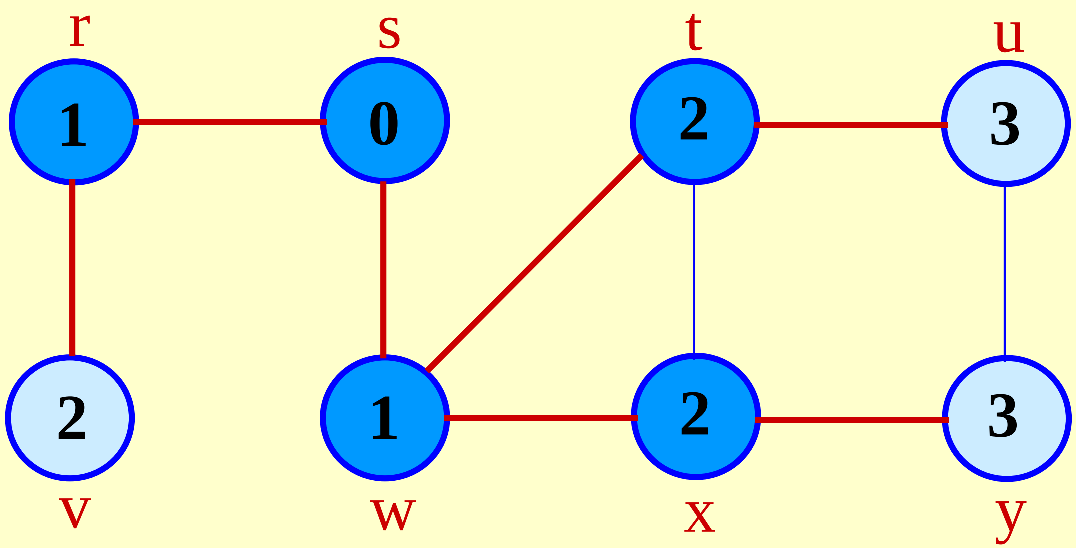
Q:	t	x	v
	2	2	2

Example (BFS)



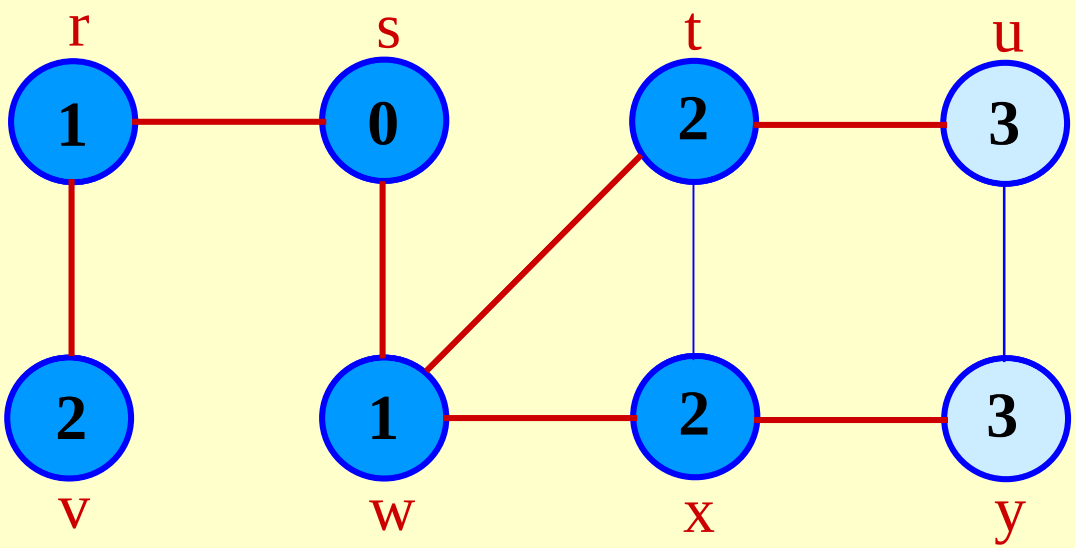
Q:	x	v	u
	2	2	3

Example (BFS)



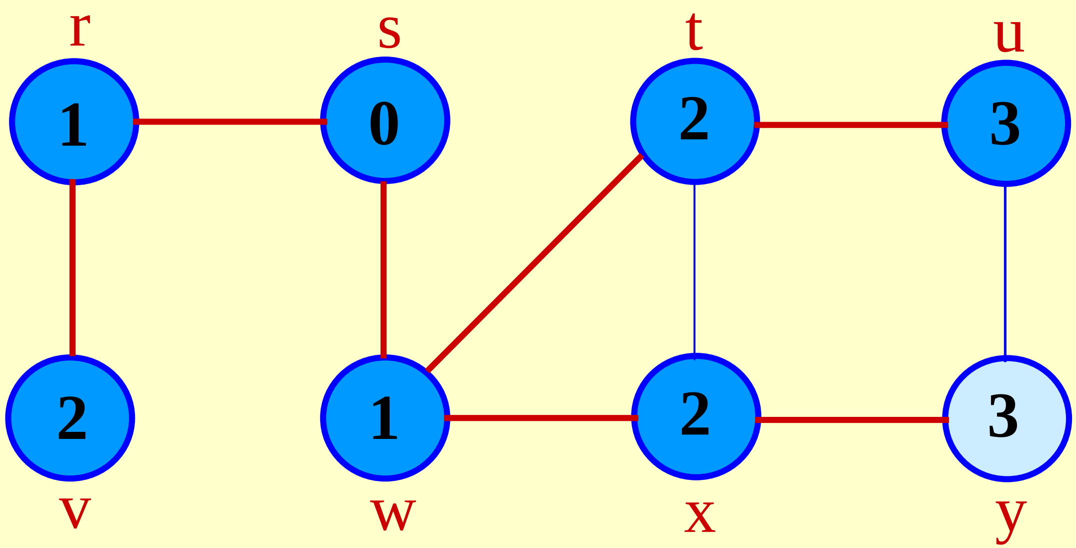
Q:	v	u	y
	2	3	3

Example (BFS)



Q: u y
3 3

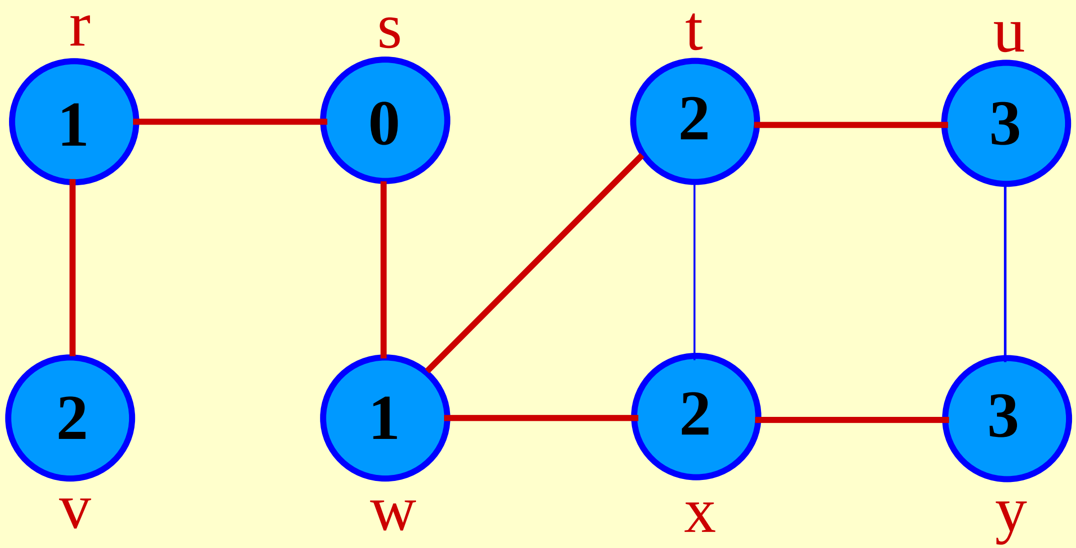
Example (BFS)



Q: y

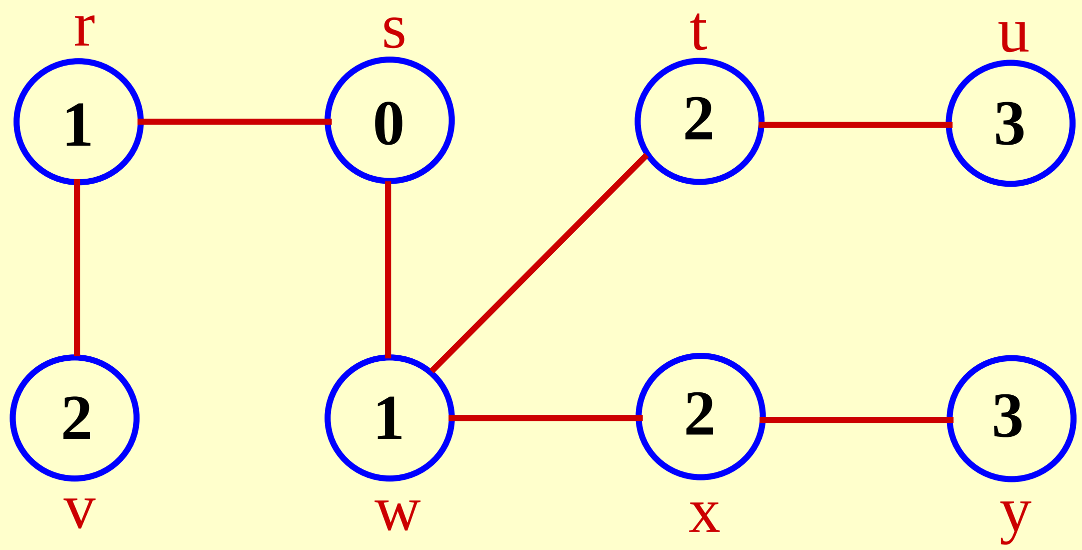
3

Example (BFS)



Q: $\varnothing$

Example (BFS)



BF Tree

Analysis of BFS

- Initialization takes $O(V)$.
- Traversal Loop
  - » After initialization, each vertex is enqueued and dequeued at most once, and each operation takes $O(1)$. So, total time for queuing is $O(V)$.
  - » The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency lists is $\Theta(E)$.
- Summing up over all vertices => total running time of BFS is $O(V+E)$, linear in the size of the adjacency list representation of graph.

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)

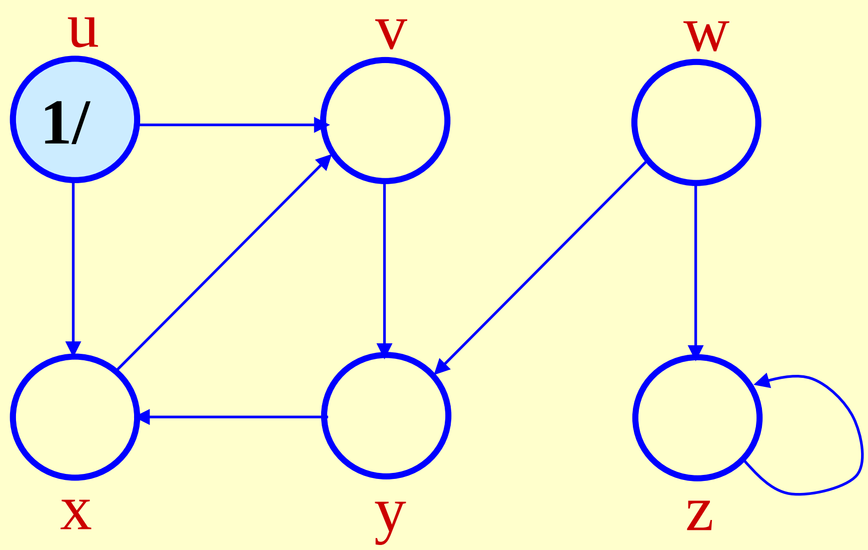
for each vertex $u \in V[G]$
  do *color*[u] $\leftarrow$ white
    $\pi[u] \leftarrow$ NIL
time $\leftarrow 0$
for each vertex $u \in V[G]$
  do if *color*[u] = white
    then DFS-Visit(u)

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

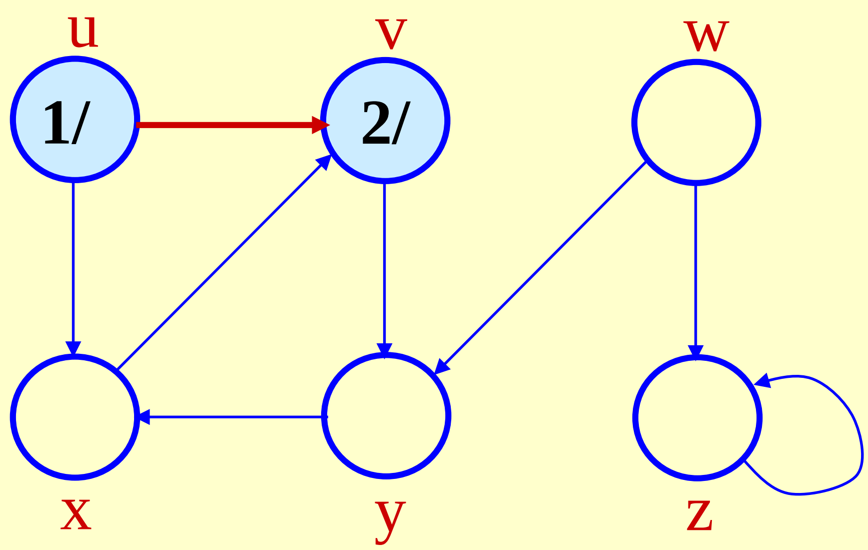
DFS-Visit(u)

color[u] $\leftarrow$ GRAY ∇ White vertex u has been discovered
time $\leftarrow$ *time* + 1
$d[u] \leftarrow$ *time*
for each $v \in Adj[u]$
  do if *color*[v] = WHITE
    then $\pi[v] \leftarrow u$
      DFS-Visit(v)
color[u] $\leftarrow$ BLACK ∇ Blacken u; it is finished.
$f[u] \leftarrow$ *time* $\leftarrow$ *time* + 1

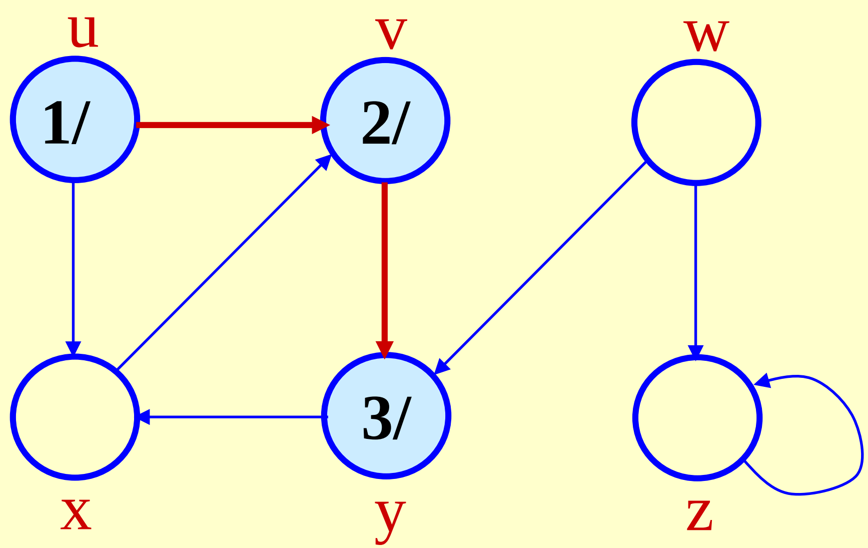
Example (DFS)



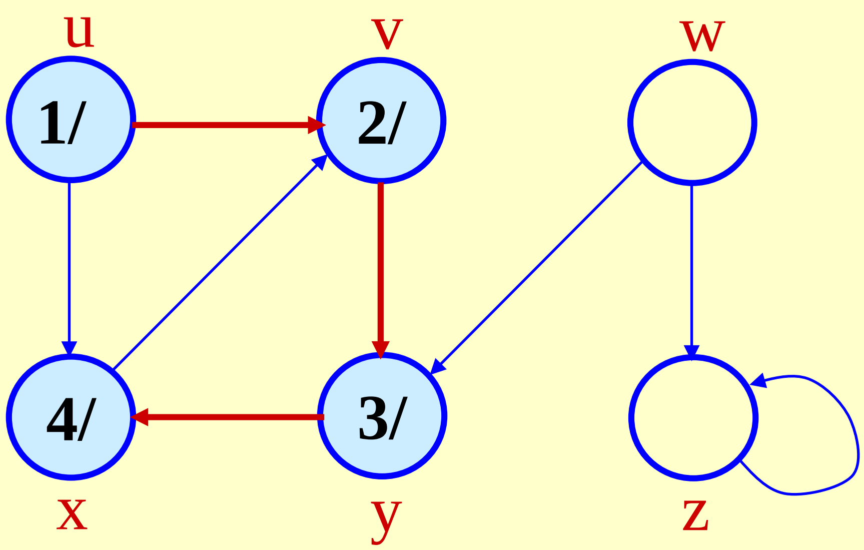
Example (DFS)



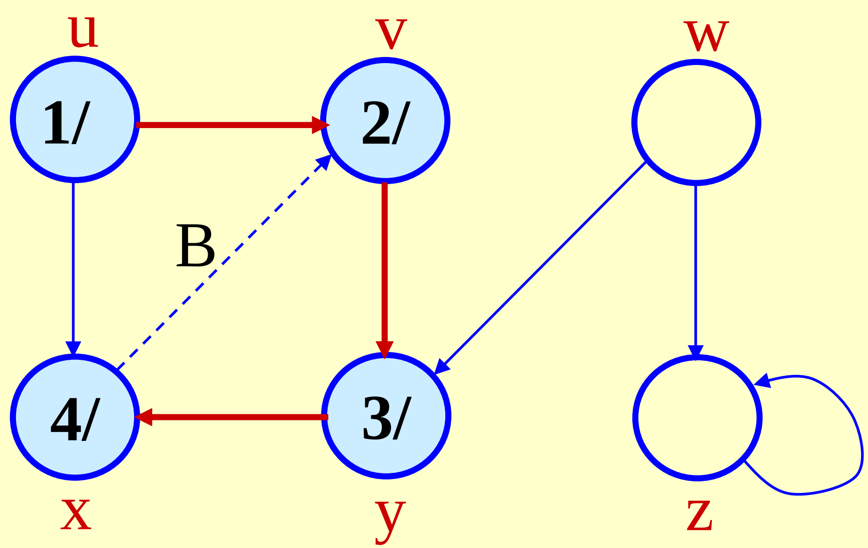
Example (DFS)



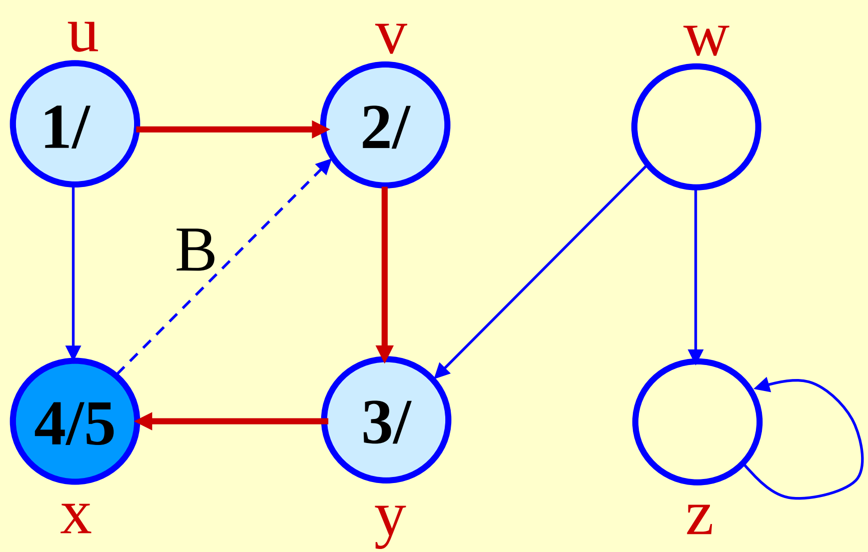
Example (DFS)



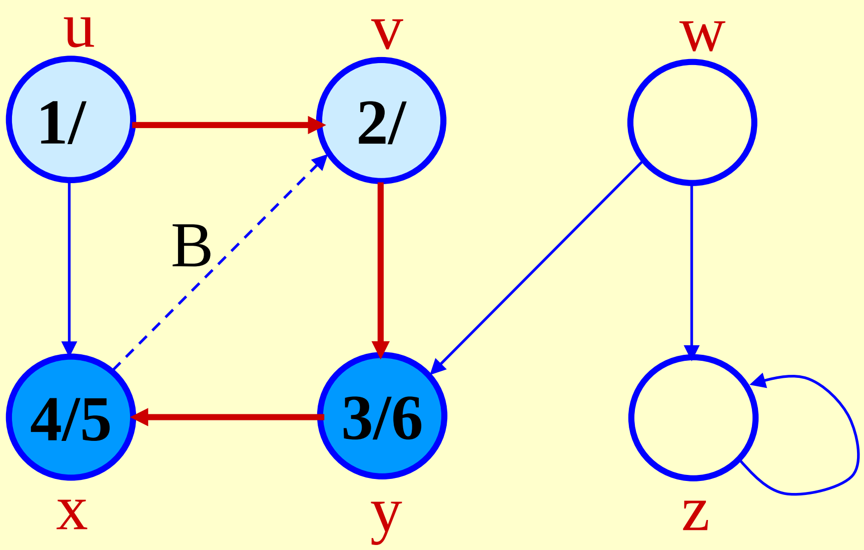
Example (DFS)



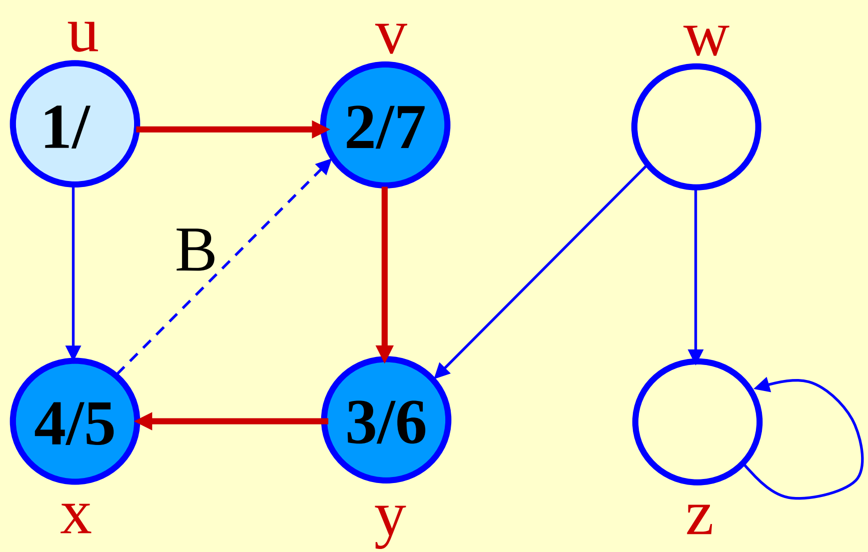
Example (DFS)



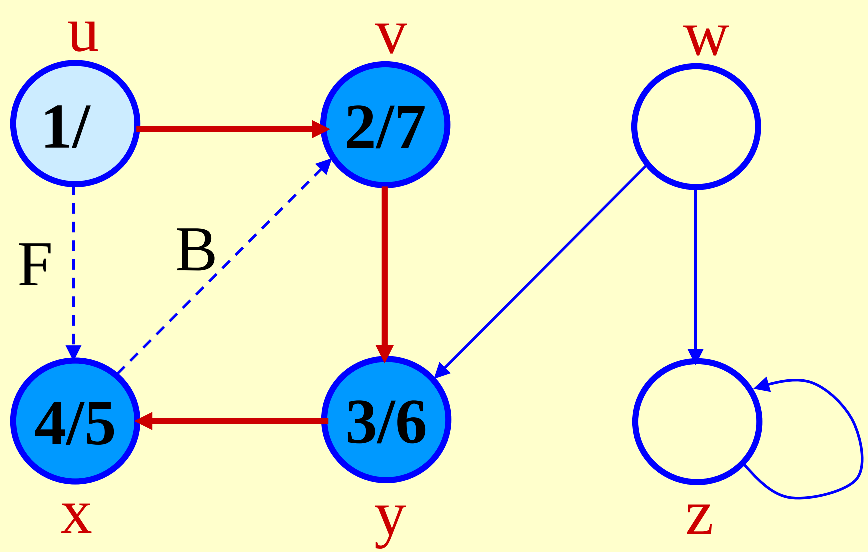
Example (DFS)



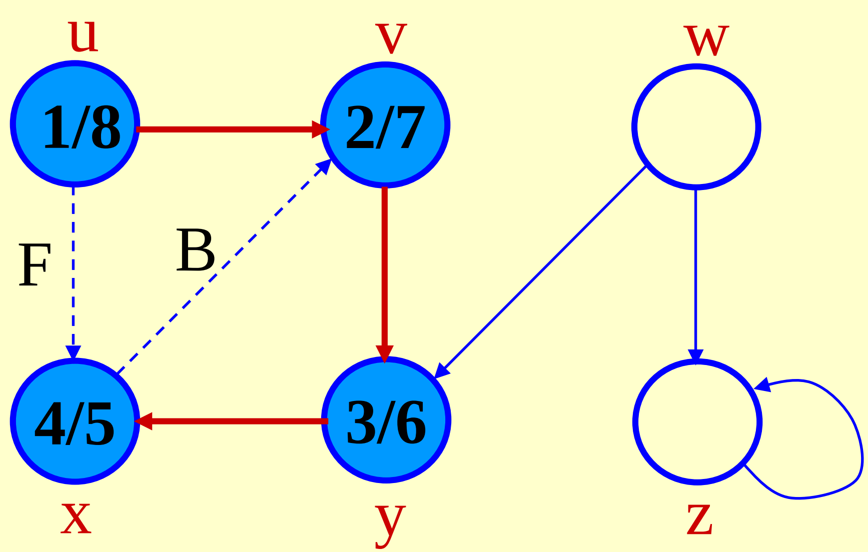
Example (DFS)



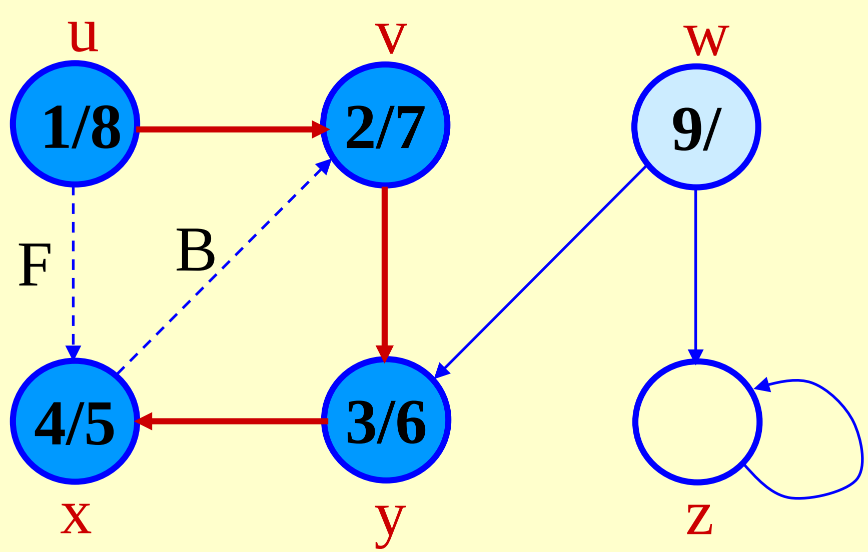
Example (DFS)



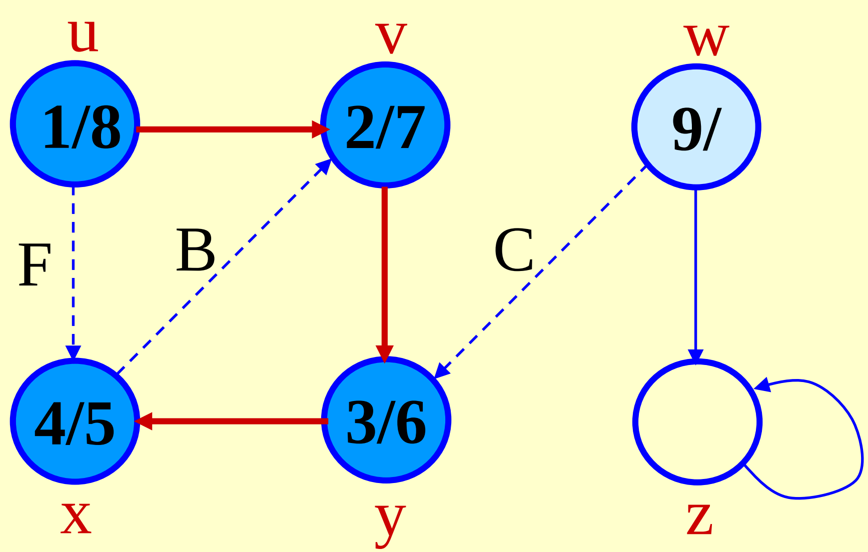
Example (DFS)



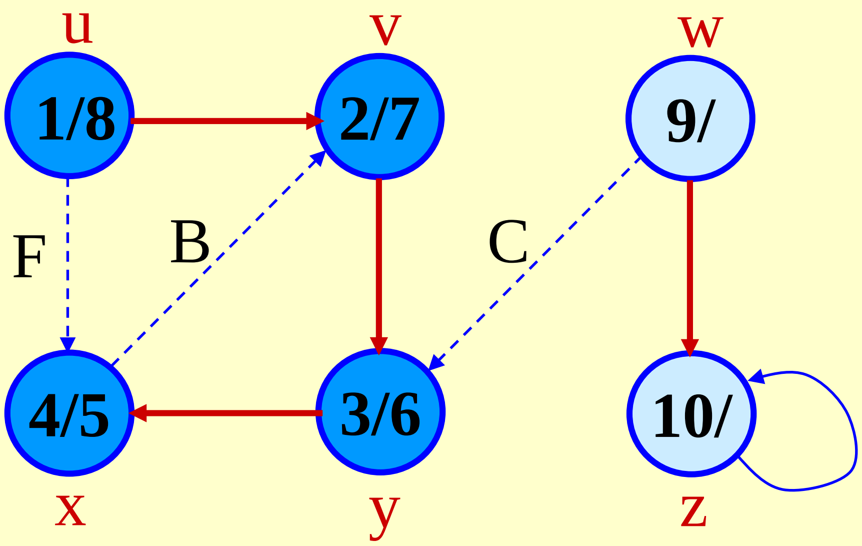
Example (DFS)



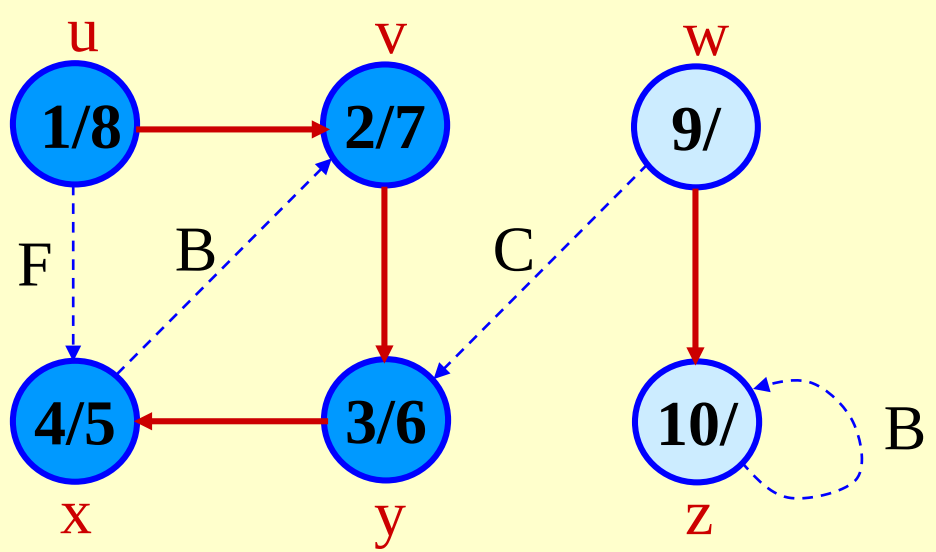
Example (DFS)



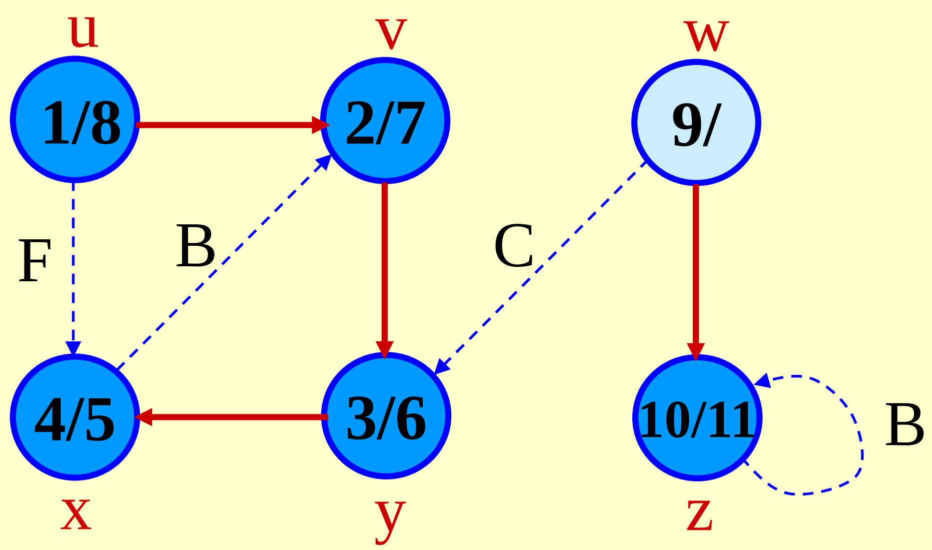
Example (DFS)



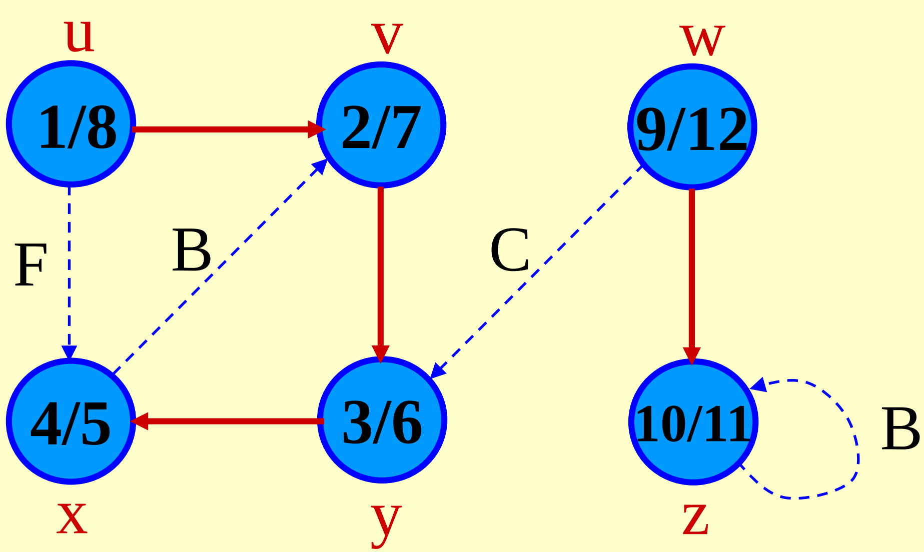
Example (DFS)



Example (DFS)



Example (DFS)



Analysis of DFS

- Loops on lines 1-3 & 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 4-6 of DFS-Visit is executed |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 $\Theta(V+E)$.

DFS-Visit(*u*)

1. *color*[*u*] ← GRA
   has been discove
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Ad*
5. **do if** *color*[
6. **then**
7.
8. *color*[*u*] ← BL
   it is finished.
9. *f*[*u*] ← *time* ←

Parenthesis Theorem

Theorem 22.7

In any depth-first search of a (directed or undirected) graph G = (V, E), for any two vertices u and v, exactly one of the following three conditions holds:

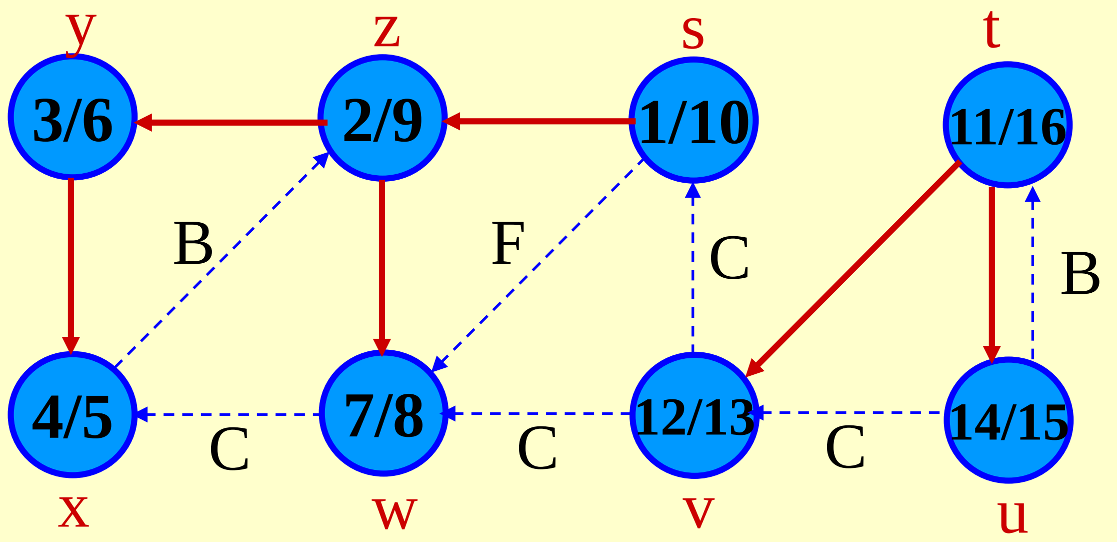
1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither u nor v is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u.
3. $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v.

- So $d[u] < d[v] < f[u] < f[v]$ *cannot* happen.
- Like parentheses:
  - OK: ( ) [ ] ([ ]) [( )]
  - Not OK: ([ )] [( ])

Corollary

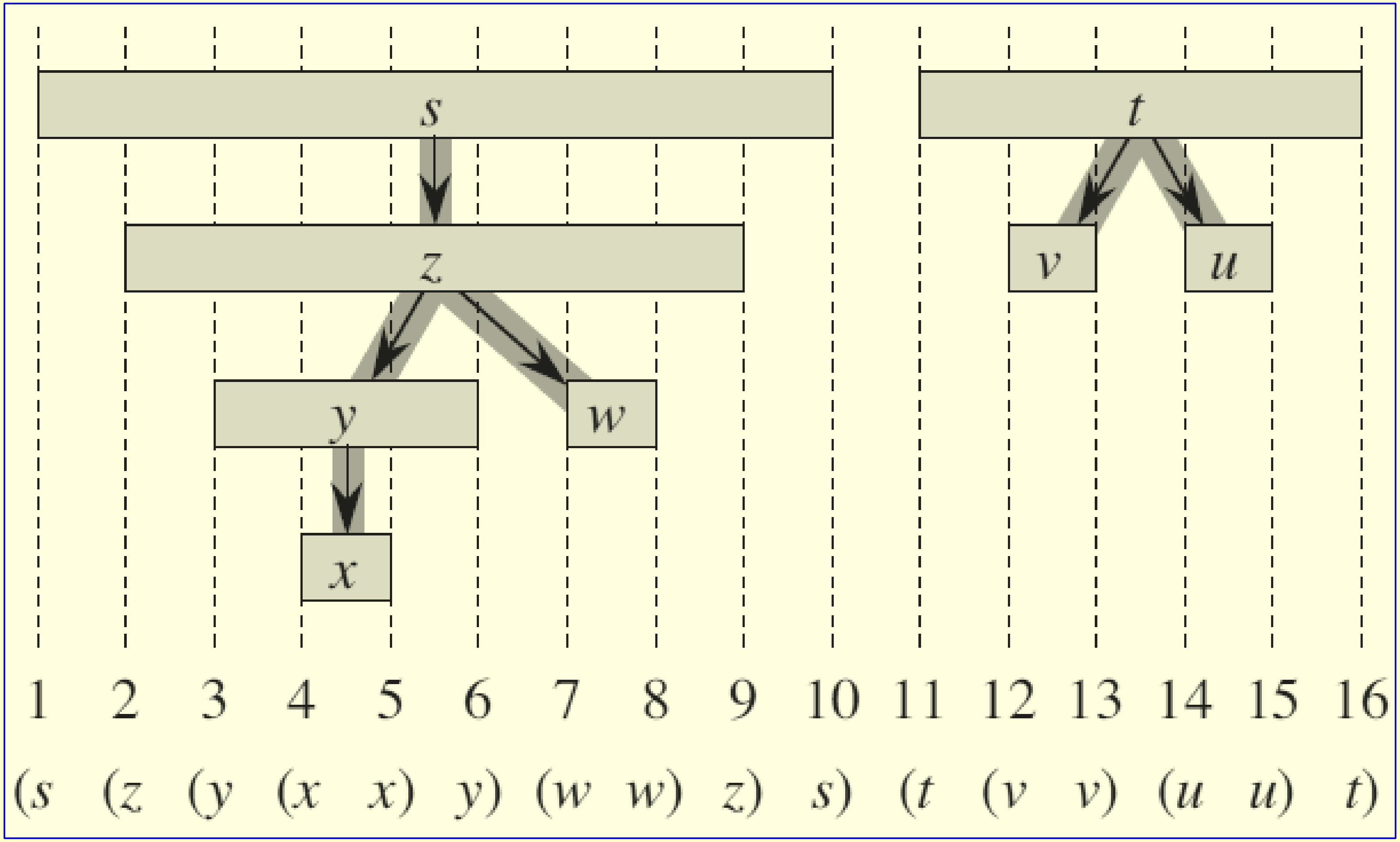
v is a proper descendant of u if and only if $d[u] < d[v] < f[v] < f[u]$.

Example (Parenthesis Theorem)



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

Example (Parenthesis Theorem)



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.

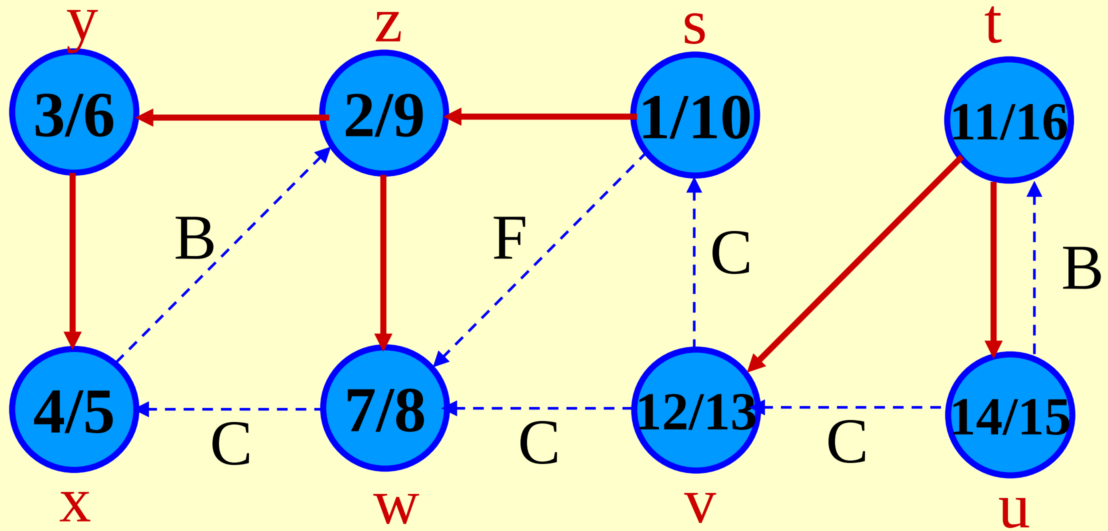
White-path Theorem

Theorem 22.9

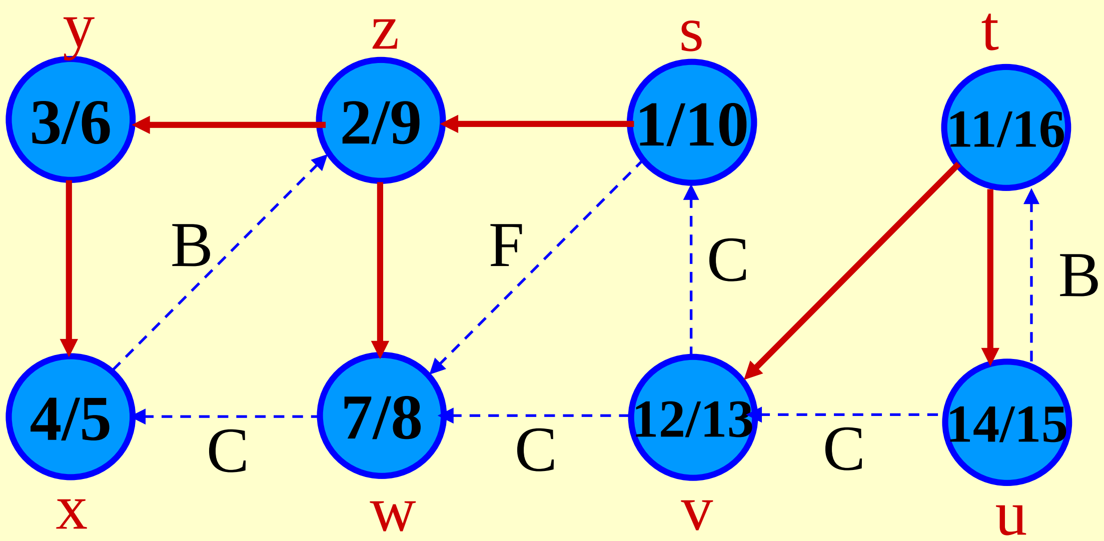
v is a descendant of u if and only if at time $d[u]$, there is a path u-v consisting of only white vertices. (Except for u, which was *just* colored gray.)

Classification of Edges

- **Tree edge:** an 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.



Classification of Edges

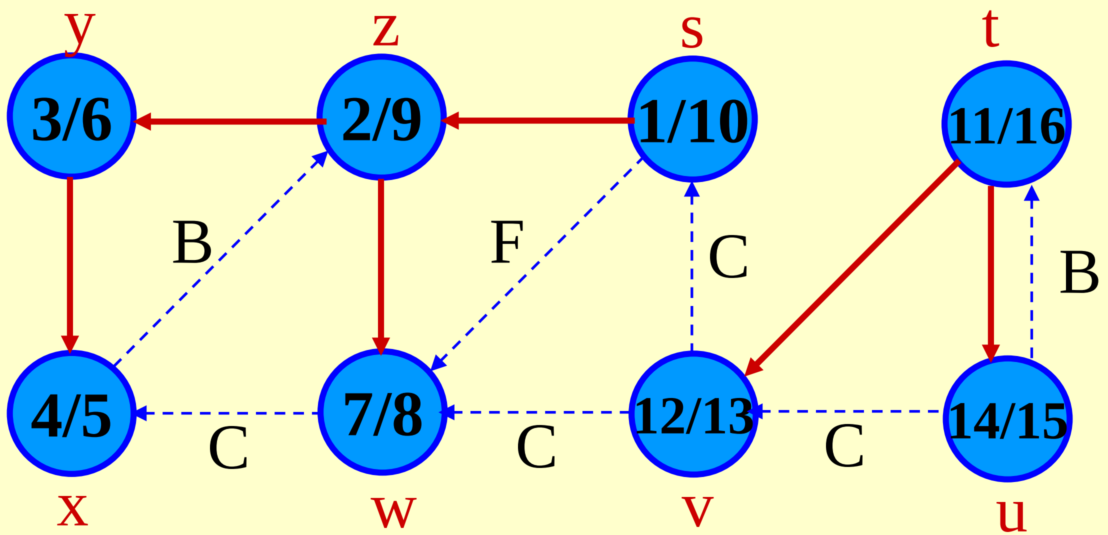


Observations:

1. In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.
2. A self loop becomes a back edge.

Identification of Edges

- Edge type for edge (*u*, *v*) can be identified when it is first explored by DFS.
- Identification is based on the **color of *v***.
  - » White – tree edge.
  - » Gray – back edge.
  - » Black – forward or cross edge.



BFS vs DFS

BFS	DFS
Uses Queue data structure	Uses Stack data structure.
Used to find single source shortest path in an unweighted graph	Path may not be the shortest.
Does not guarantee the total coverage	Guarantees the total coverage
Suitable for searching vertices which are closer to the given source	Suitable when there are solutions away from the selected source

Applications of BFS and DFS

BFS Applications

- Finding the shortest path in a graph
- Finding the MST
- Realization of P2P Networks
- Neighbour Searching
- GPS Navigation
- Broadcasting
- Route Finding

DFS applications

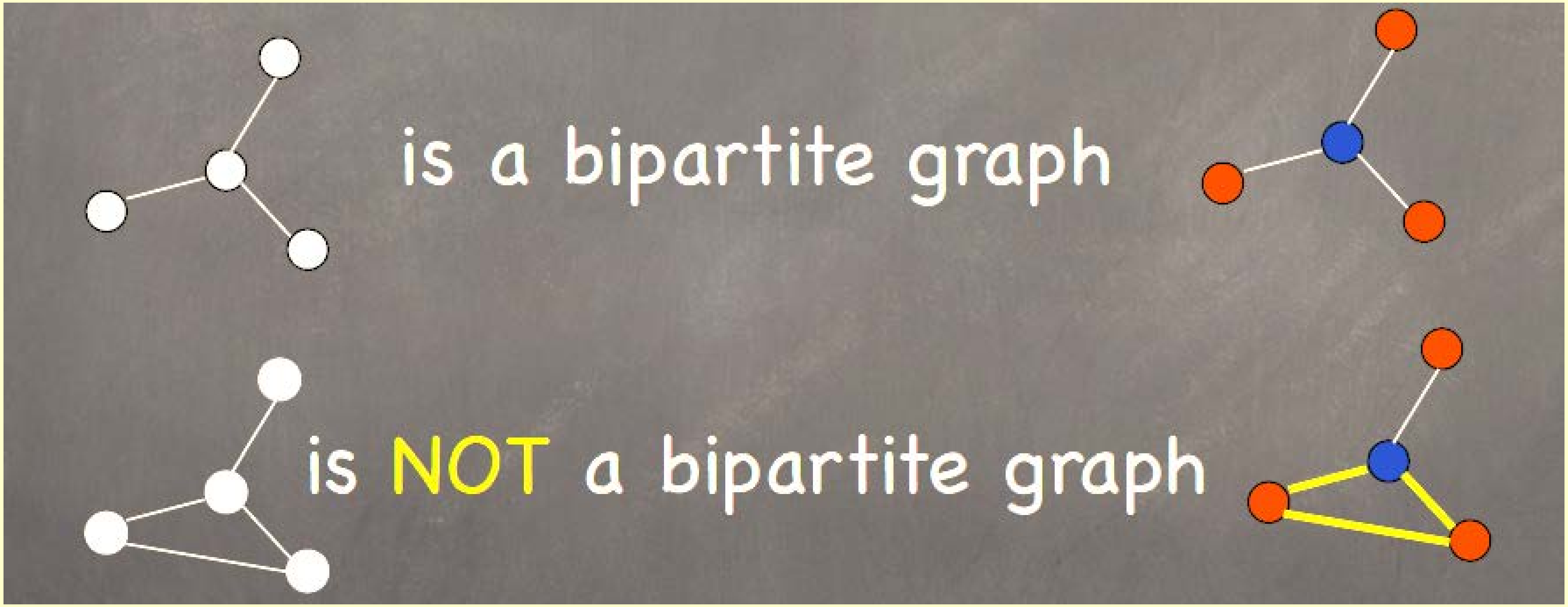
- Cycle detection in a graph.
- Path finding.
- Topological sorting
- Finding the SCC of a graph

Application of BFS

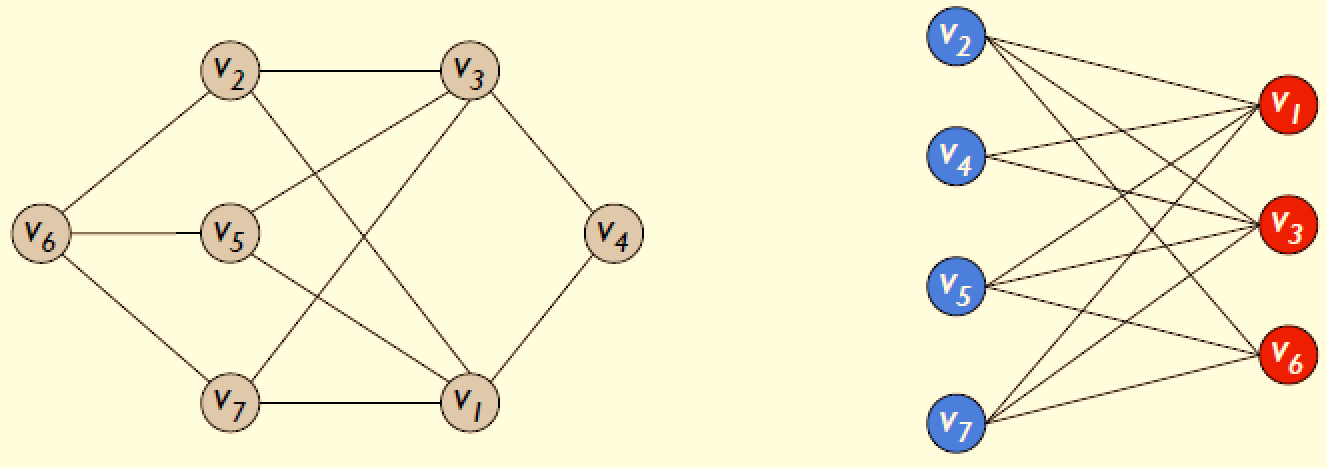
Testing the Bipartiteness of a graph

Bipartite graph

- A bipartite graph is a graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V.



Bipartite graph

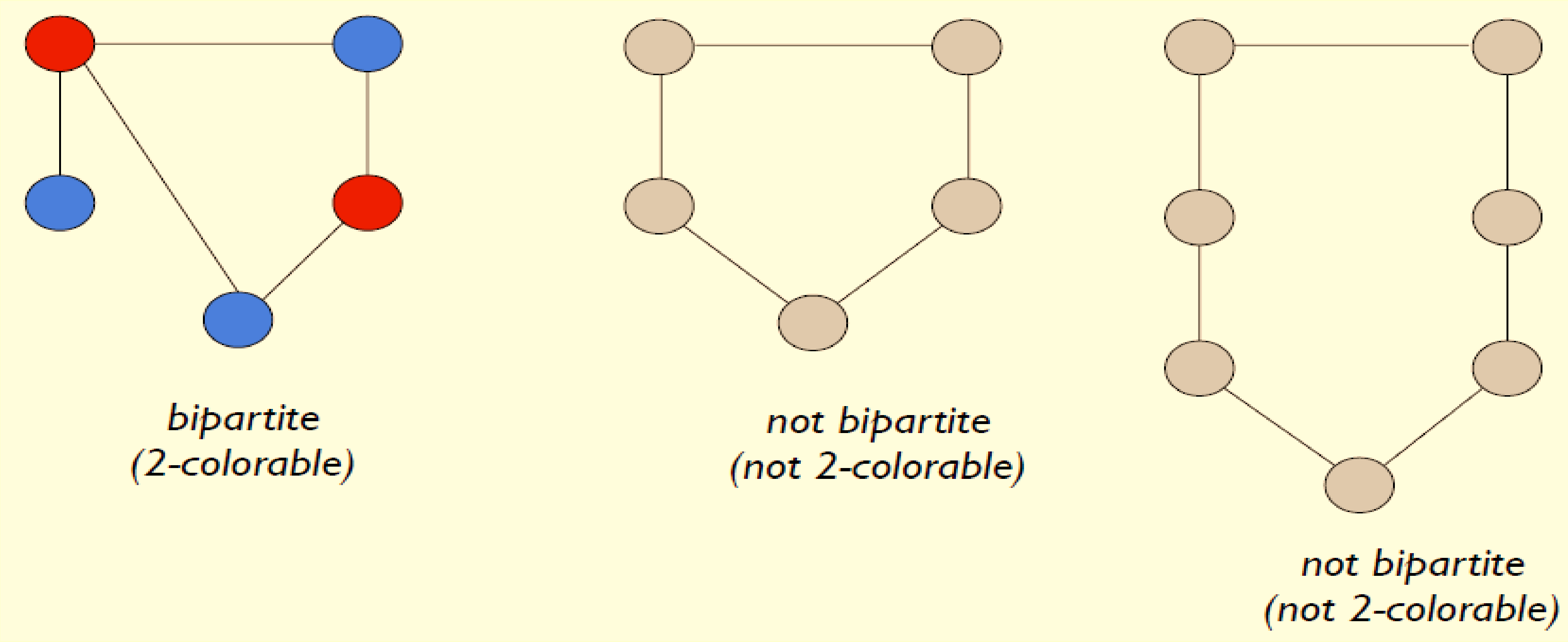


a bipartite graph G

another drawing of G

Bipartite graph

- A bipartite graph is an undirected graph G = (V, E) in which the nodes can be colored red or blue such that every edge has one red and one blue end.
- A bipartite graph is a graph that does not contain any odd-length cycles.

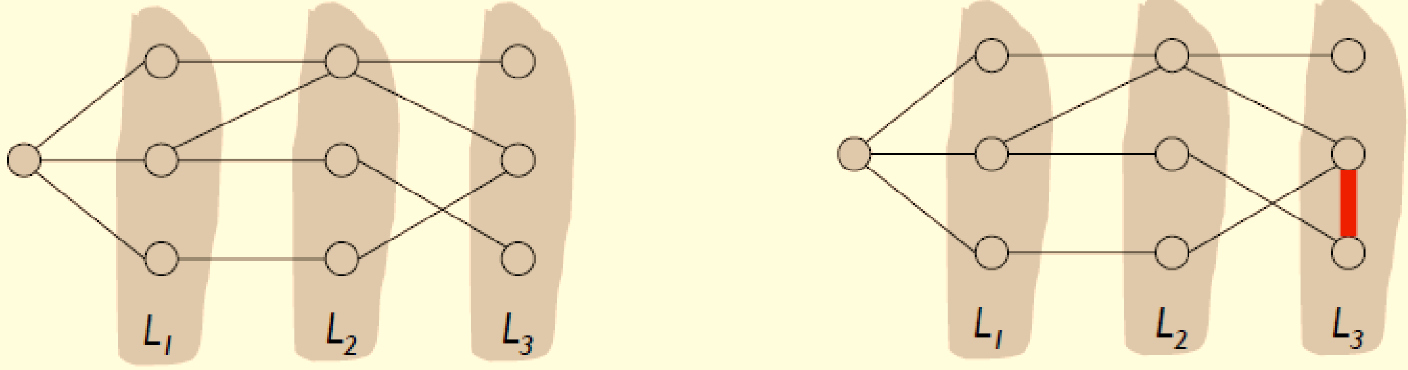


BFS and Bipartiteness

Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



Case (i) Case (ii)

BFS and Bipartiteness

Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.

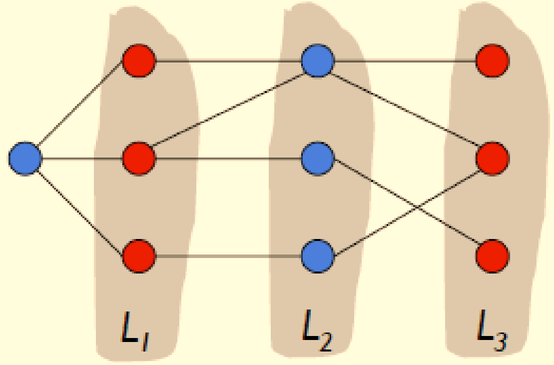
(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (i)

Suppose no edge joins two nodes in the same layer.

By previous lemma, all edges join nodes on adjacent levels.



Case (i)

Bipartition:

red = nodes on odd levels,

blue = nodes on even levels.

BFS and Bipartiteness

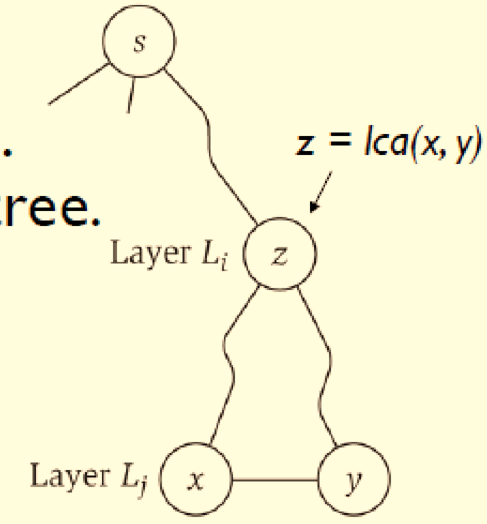
Lemma. Let G be a connected graph, and let L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (ii)

Suppose (x, y) is an edge & x, y in same level Lj.
Let z = their lowest common ancestor in BFS tree.
Let Li be level containing z.
Consider cycle that takes edge from x to y, then tree from y to z, then tree from z to x.
Its length is $\underbrace{1}_{(x,y)} + \underbrace{(j-i)}_{\text{path from } y \text{ to } z} + \underbrace{(j-i)}_{\text{path from } z \text{ to } x}$, which is odd.



Bipartiteness Testing

- How can we test if G is bipartite?
  - » Do a BFS starting from some node s.
  - » Color even levels “blue” and odd levels “red”.
  - » Check each edge to see if any edge has both endpoints the same color.

Bipartiteness Testing