Lecture 23.
Elementary Graph Algorithms

Introduction to Algorithms
Sungkyunkwan University

Hyunseung Choo
choo@skku.edu

Copyright 2000-2022 Intelligent Networking Laboratory

Graphs

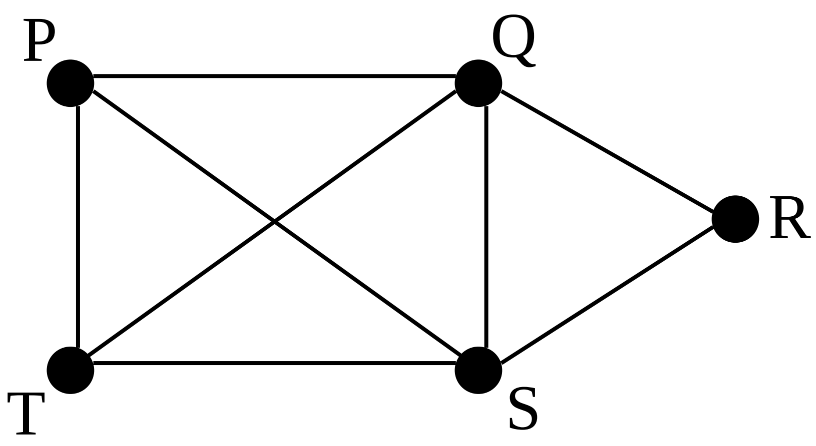
- *Graph* $G = (V, E)$
  - V is set of vertices
  - E is set (family) of edges $\subseteq (V \times V)$
    - edge or link or arc
- Types of graphs
  - Undirected: edge $(u, v) = (v, u)$
    - for all v, $(v, v) \notin E$ (No self loops)
  - Directed: (u, v) is edge from u to v, denoted as $u \rightarrow v$
    - self loops are allowed
  - Weighted: each edge has an associated weight, given by a weight function $w : E \rightarrow \mathrm{R}$
  - 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
- Other definitions in Appendix B (B.4 and B.5) as needed
  - On pages 1080-1093

Eulerian and Hamiltonian Graphs

- A graph containing paths that include every 'edge' exactly once and end at the initial vertex is called an Eulerian graph
- A graph containing paths that include every 'vertex' exactly once and end at the initial vertex is called a Hamiltonian graph



Hamiltonian but not Eulerian

Trees and Planar Graphs

- A connected graph with only one path between each pair of vertices is called a <u>tree</u>
- A tree can also be defined as a connected graph containing no cycles (figure 1)
- A graph that can be redrawn without crossings is called a <u>planar graph</u> (figures 2 and 3)

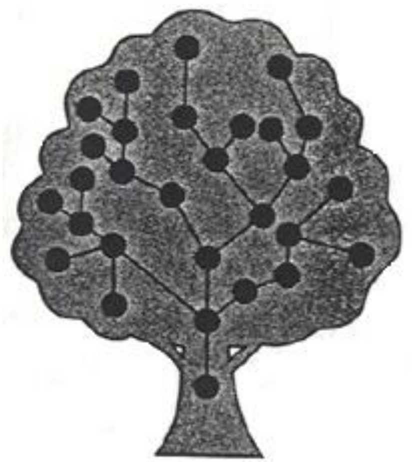


Fig. 1

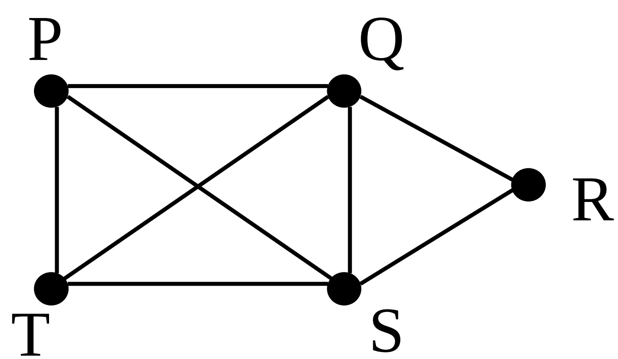


Fig. 2

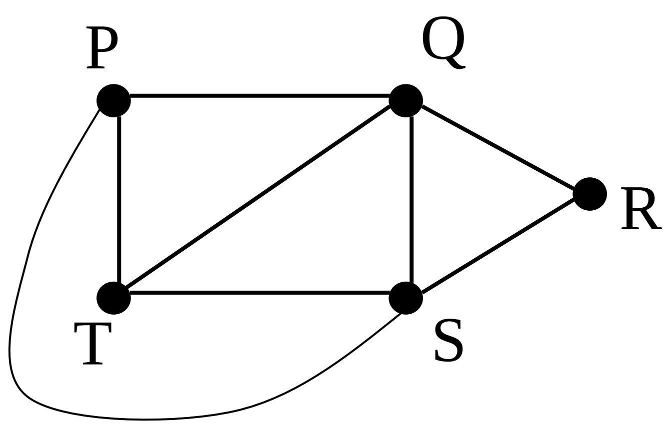


Fig. 3

Other Definitions

- Two digraphs are <u>isomorphic</u> if there is an isomorphism between their underlying graphs that preserves the ordering of the vertices in each arc
  - See figures 4 and 5 : not isomorphic
  - See figures 6 and 7 : isomorphic

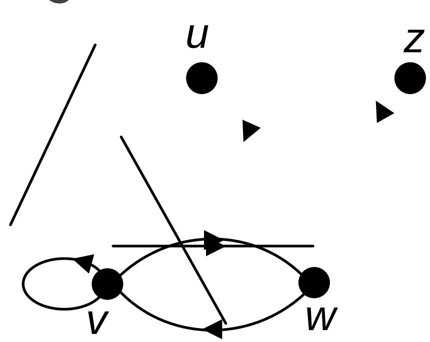


Fig. 4

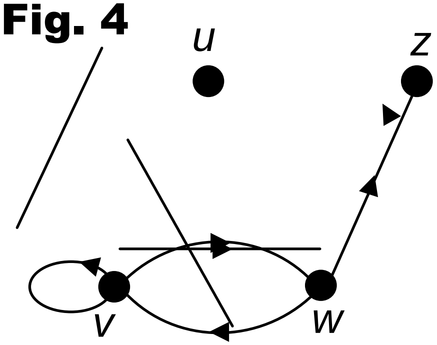


Fig. 5

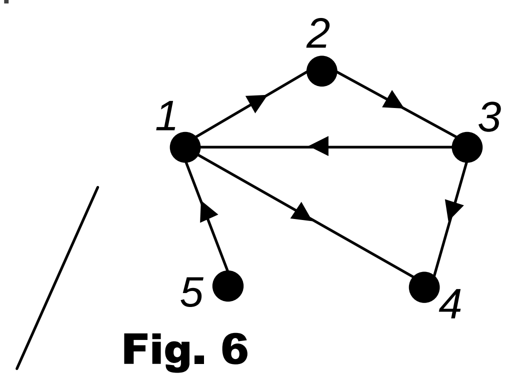


Fig. 6

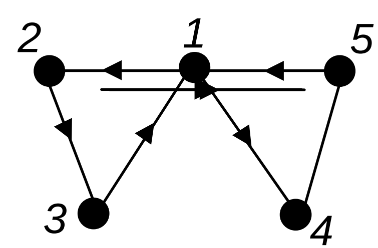
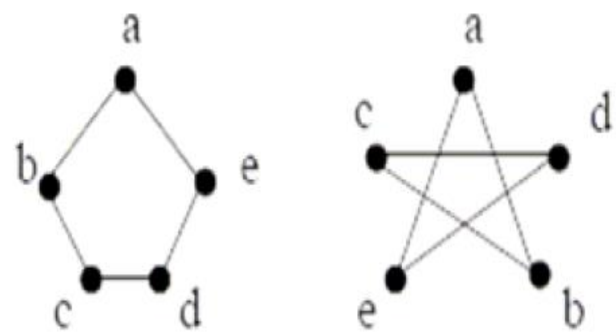


Fig. 7

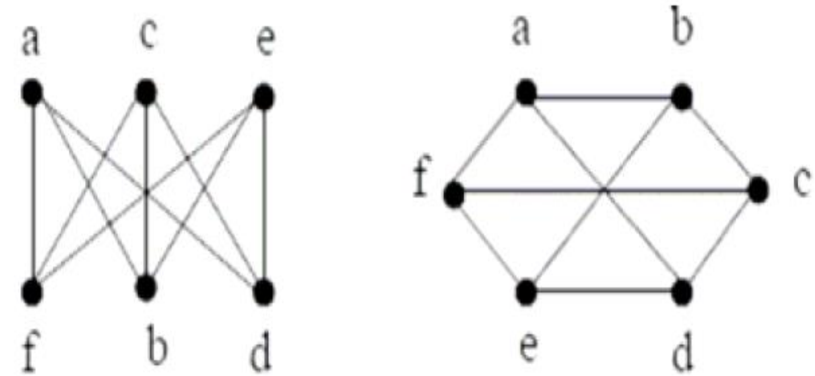
Practice Problems

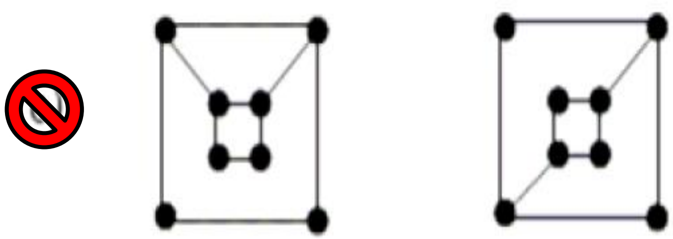
- Which of the following pairs are isomorphic graphs:

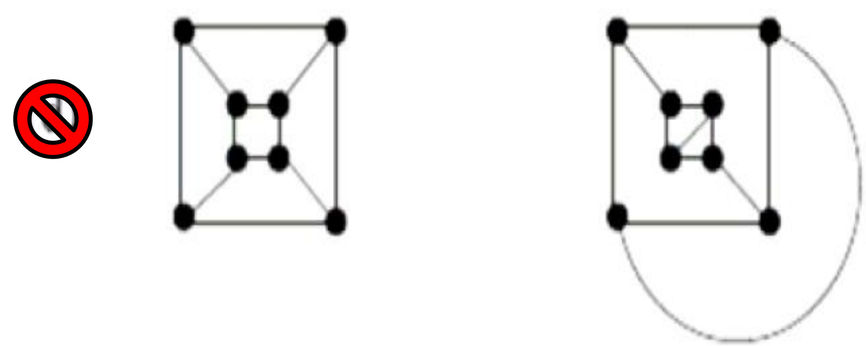
(a)



(b)







Connected and Strongly Connected

- A digraph D is <u>connected</u> if it cannot be expressed as the union of two digraphs
  - This is equivalent to saying that the underlying graph of D is a connected graph
- D is <u>strongly connected</u> if, for any two vertices v and w of D, there is a path from v to w
- Every strongly connected digraph is connected, but not all connected digraphs are strongly connected
  - See figure 4

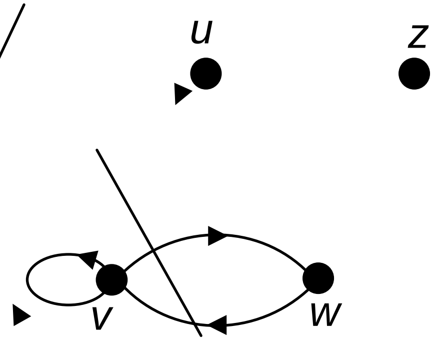


Fig. 4

Connected and Strongly Connected

- We define a graph *G* to be orientable if each edge of *G* can be directed so that the resulting digraph is strongly connected
  - See figures 8 and 9
- Any Eulerian graph is orientable

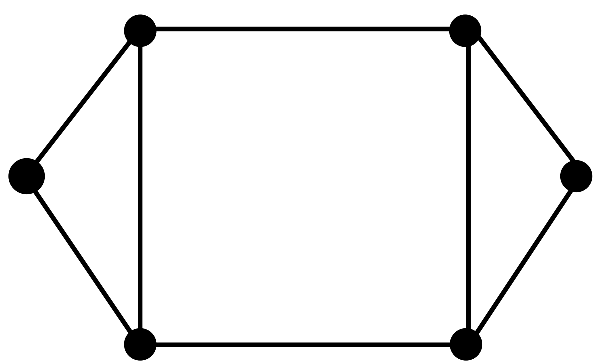


Fig. 8

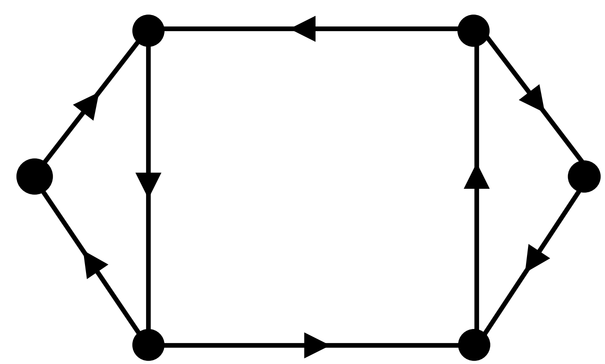
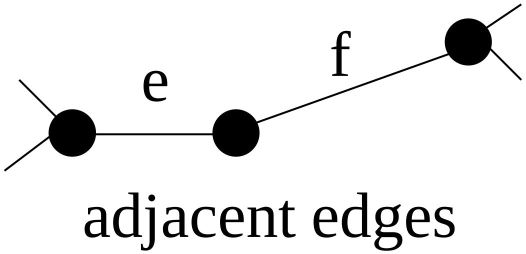


Fig. 9

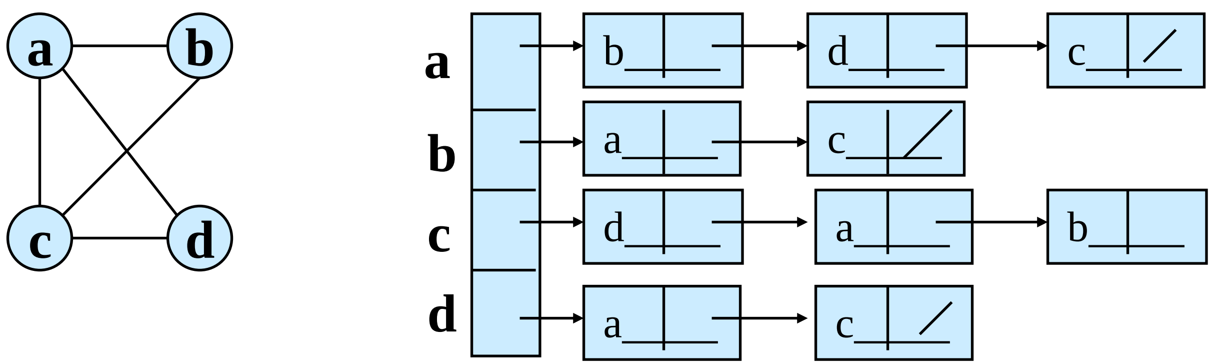
Adjacency

- Two vertices v and w are adjacent if there is an edge (v,w) joining them, and the vertices v and w are then incident with such an edge
- Two distinct edges e and f are adjacent if they have a vertex in common

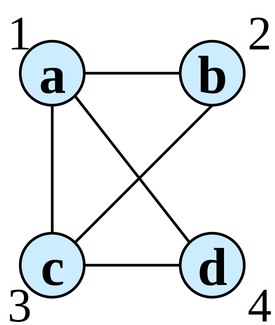


Representation of Graphs

- Two standard ways
  - Adjacency lists



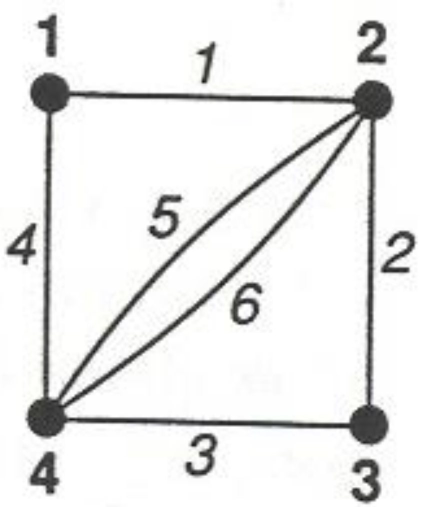
  - Adjacency matrix



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

Representation of Graphs

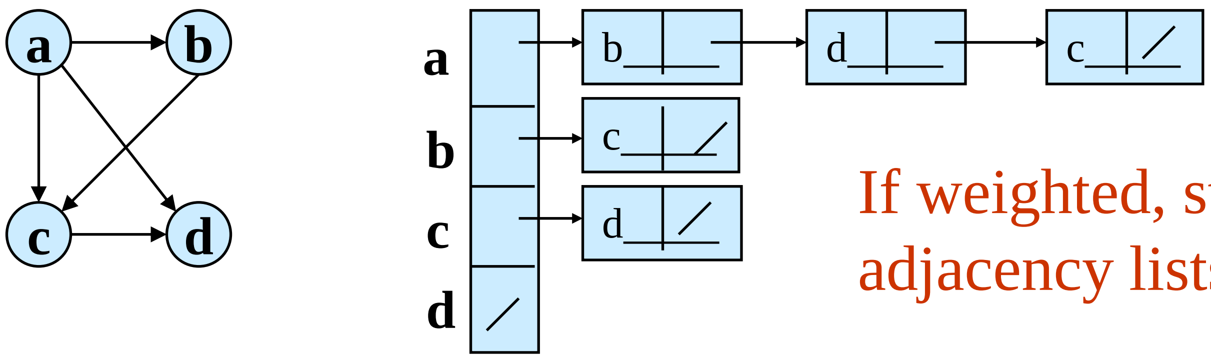
- Adjacency matrix A
- Incidence matrix M



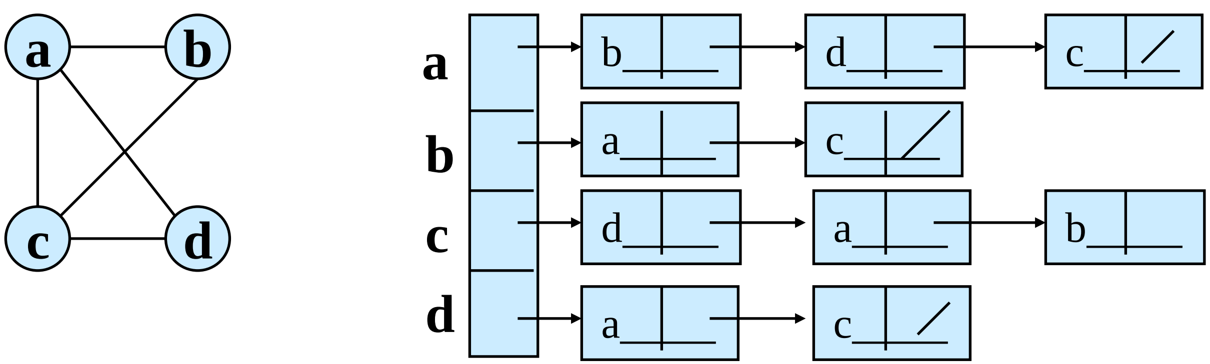
$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{pmatrix} \qquad \mathbf{M} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

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

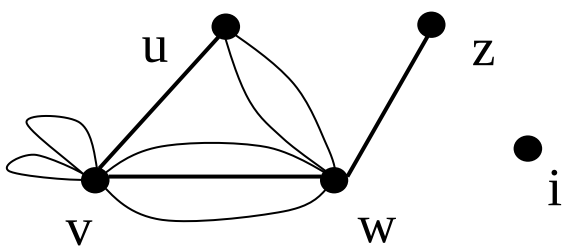


If weighted, store weights also in adjacency lists



Degree

- The degree of a vertex v is the number of edges incident with v, and is written deg(v)
- A loop at v contributes 2 to the degree of v
- A vertex of degree 0 is an isolated vertex
- A vertex of degree 1 is an end-vertex
- Remember the handshaking lemma and its corollary



Handshaking Lemma

- Handshaking Lemma
  - If several people shake hands, then the total number of hands shaken must be even
  - In any graph the sum of all the vertex degrees is an even number
    - in fact, twice the number of edges
- Corollary
  - In any graph the number of vertices of odd degree is even

Handshaking Dilemma

- The out-degree of a vertex v of G is the number of arcs of the form (v,w), and is denoted by outdeg(v)
- The in-degree of a vertex v of G is the number of arcs of the form (w,v), and is denoted by indeg(v)
- The sum of the out-degrees of all the vertices of G is equal to the sum of their in-degrees
- We call this result the <u>handshaking dilemma</u>
- A <u>source</u> of G is a vertex with in-degree 0
- A <u>sink</u> of G is a vertex with out-degree 0

Storage Requirement

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

    $$\sum_{v \in V} \mathrm{outdeg}(v) = |E|$$

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

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

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

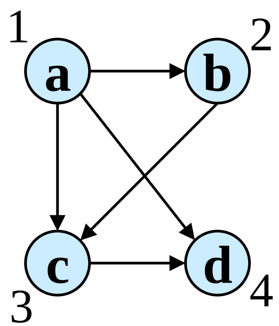
Pros and Cons: Adj List

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

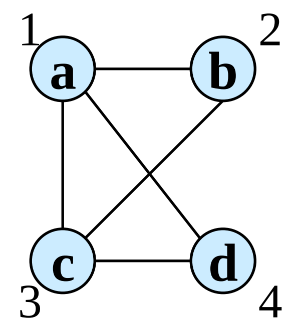
Adjacency Matrix

- $|V| \times |V|$ 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

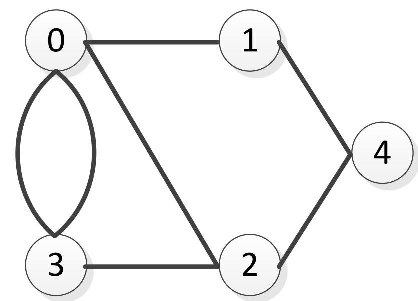
Space and Time

- 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

Practice Problems

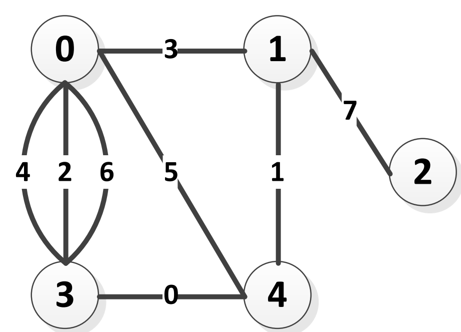
- Draw the graph whose adjacency matrix is given as follows:

$$\begin{pmatrix} 0 & 1 & 1 & 2 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 2 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$



- Draw the graph whose incidence matrix is given as follows:

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$



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)

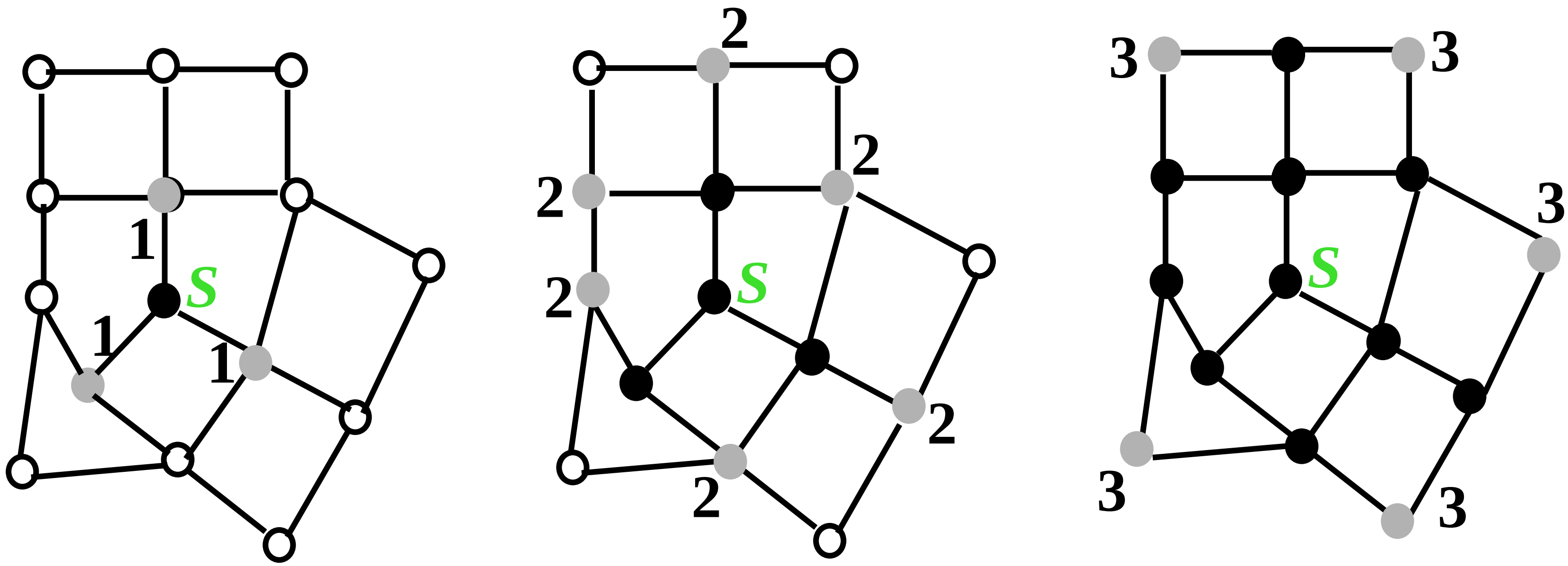
Breadth-First Search

- Input
  - Graph $G = (V, E)$, either directed or undirected, and *source vertex* $s \in V$
- Output
  - $d[v]$ = distance (smallest number 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 $s \leadsto v$
    - 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.

Breadth-First Search



● **Finished** ● **Discovered** ○ **Undiscovered**

Pseudo Code

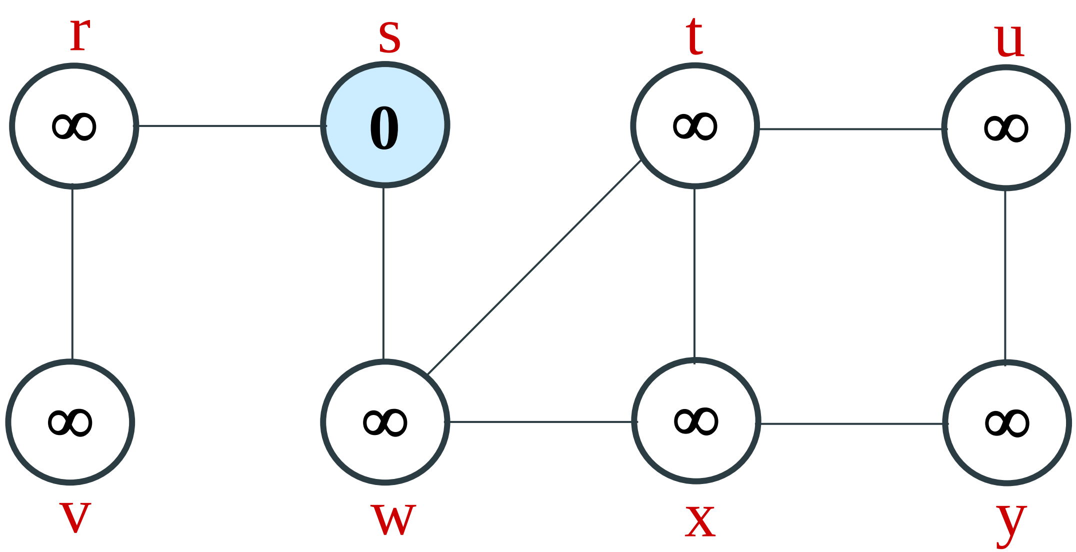
<u>BFS(G,s)</u>

```
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

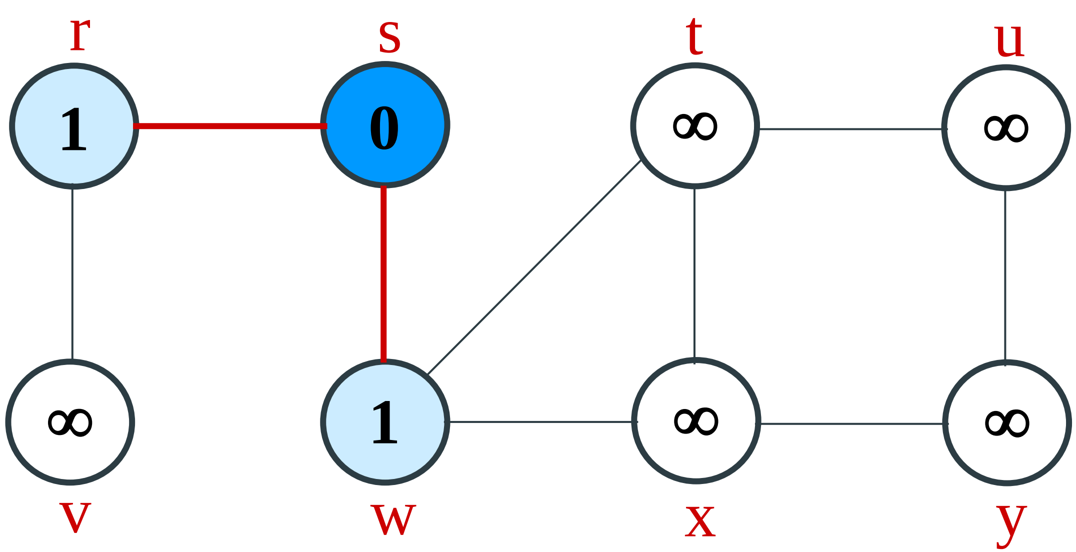
Q: a queue of discovered vertices
color[v]: color of v
d[v]: distance from s to v
π[u]: predecessor of v

BFS Example



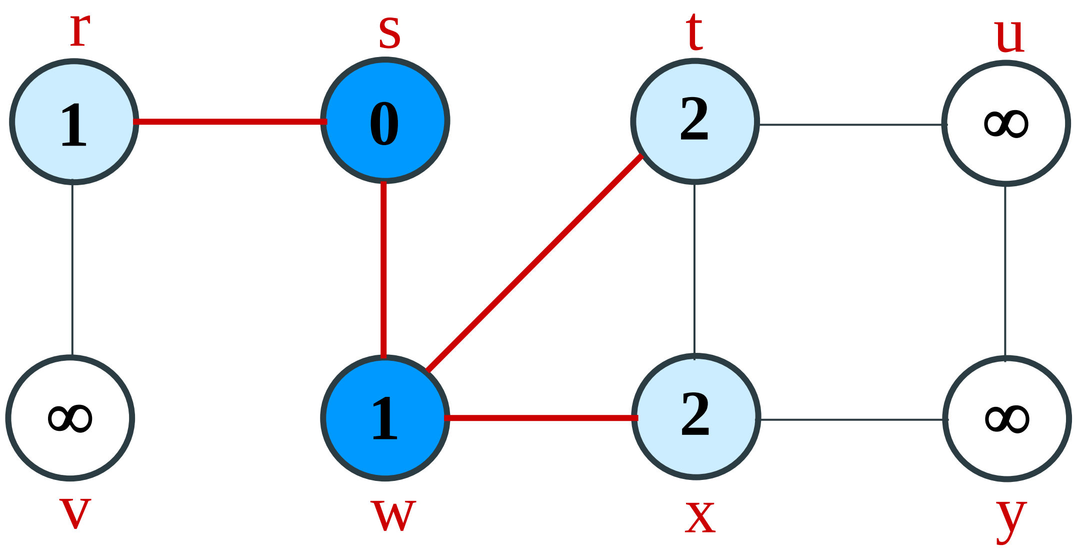
Q: s
0

BFS Example



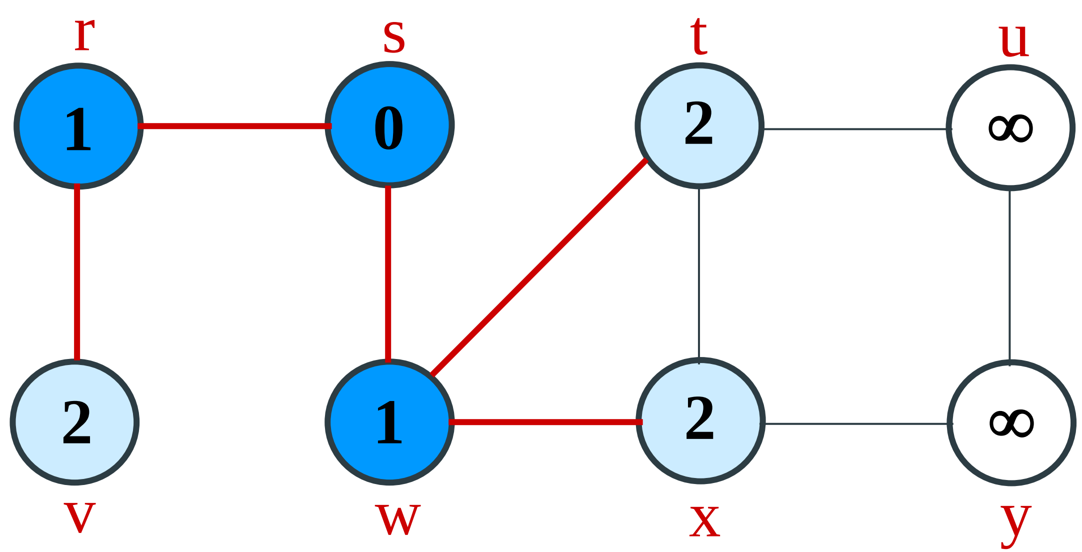
Q: w r
1 1

BFS Example



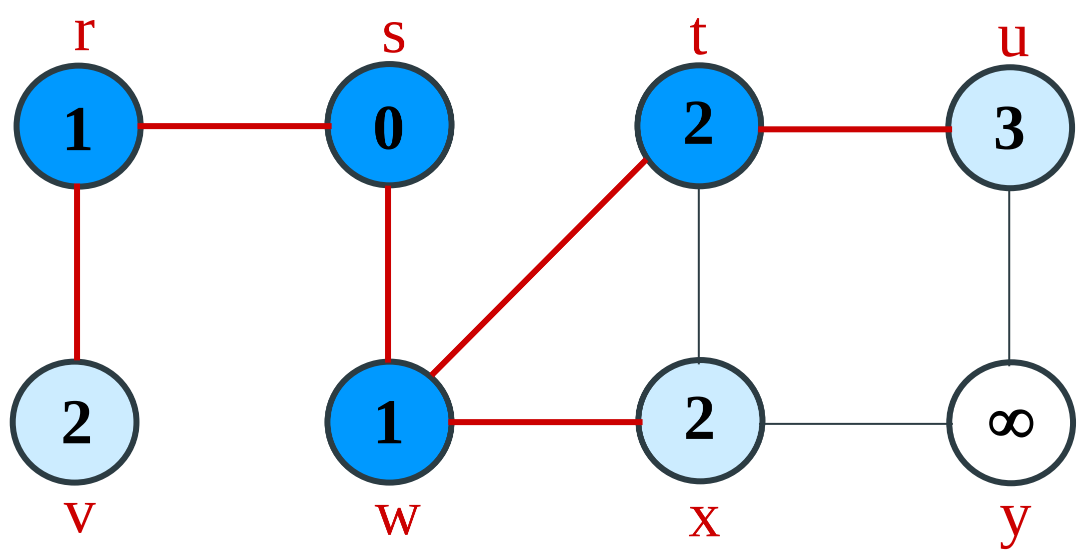
Q:	r	t	x
	1	2	2

BFS Example



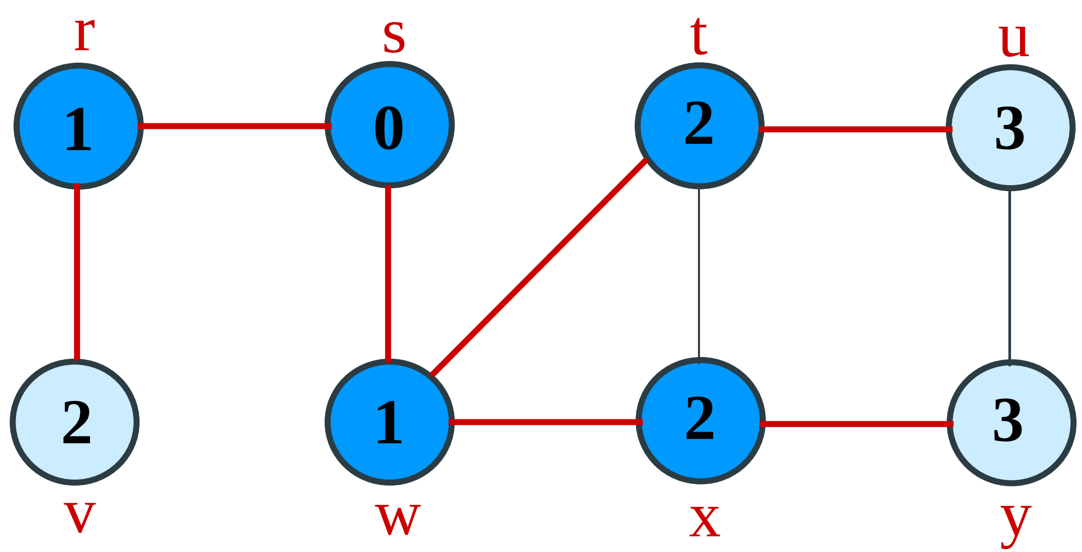
Q: t x v
2 2 2

BFS Example



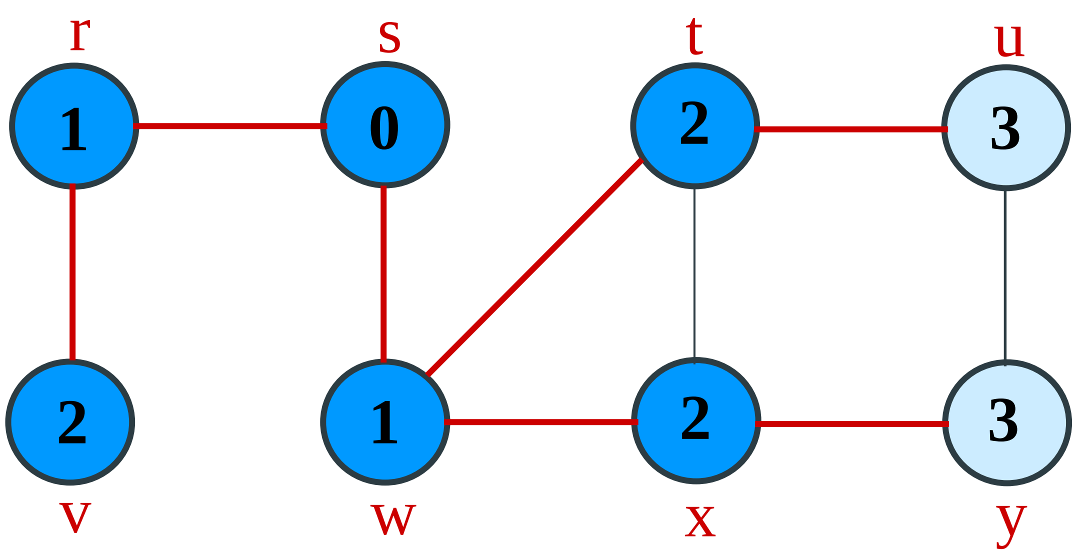
Q:	x	v	u
	2	2	3

BFS Example



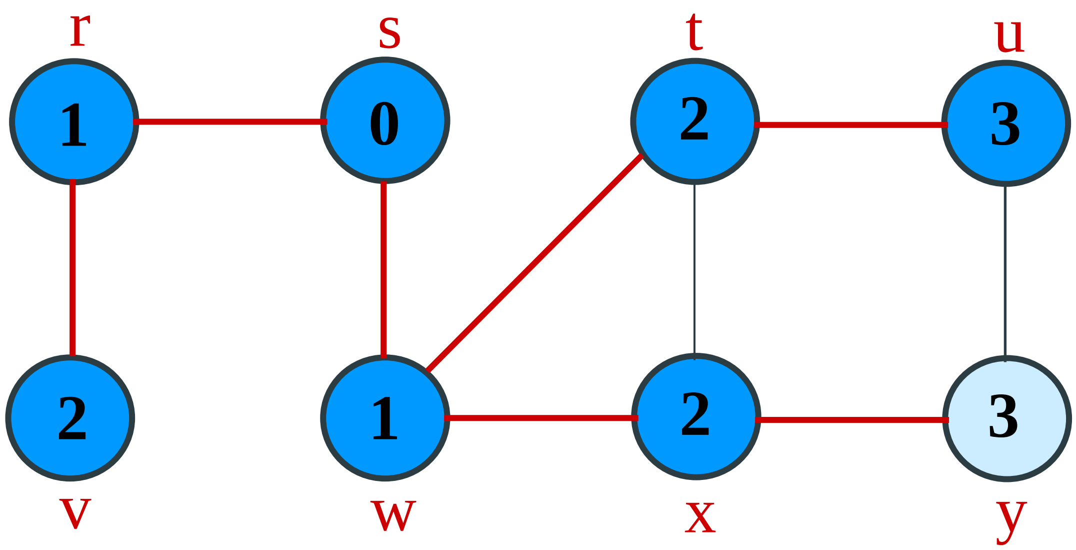
Q:	v	u	y
	2	3	3

BFS Example



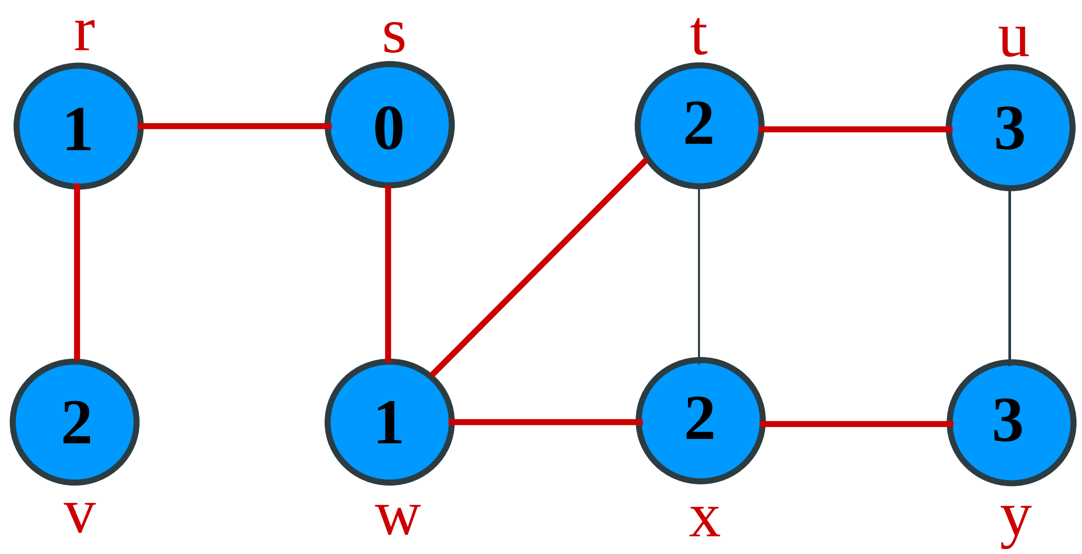
Q:	u	y
	3	3

BFS Example



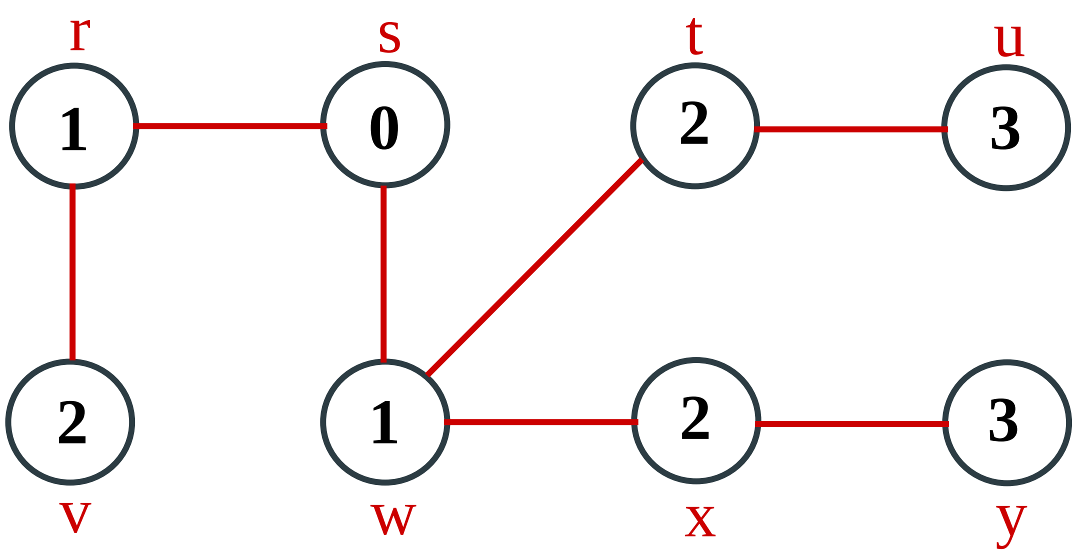
Q: y
3

BFS Example



Q: $\varnothing$

BFS Example



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 Trees

- 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

- 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
  - White – Undiscovered
  - Gray – Discovered but not finished
  - Black – Finished

Pseudo Code

DFS(*G*)

```
for each vertex u ∈ V[G]
      do color[u] ← white
            π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
      do if color[u] = white
               then DFS-Visit(u)
```

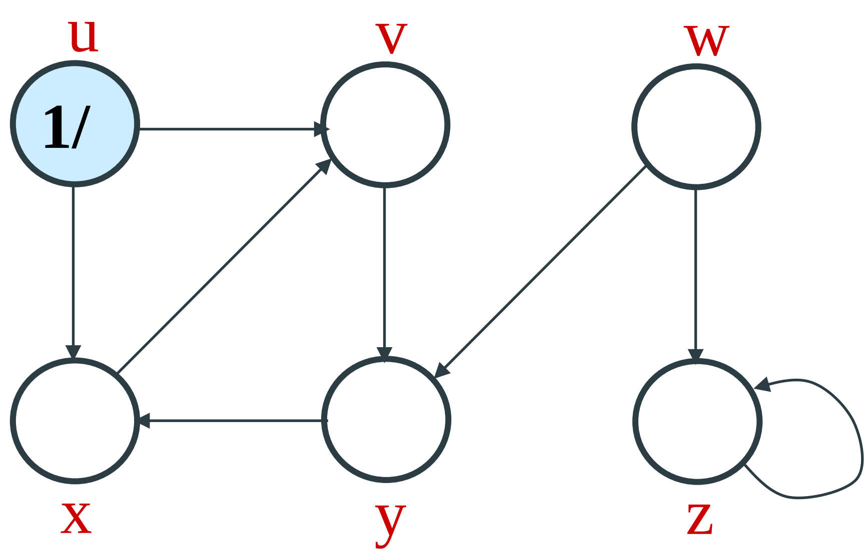
Uses a global timestamp ***time***

DFS-Visit(*u*)

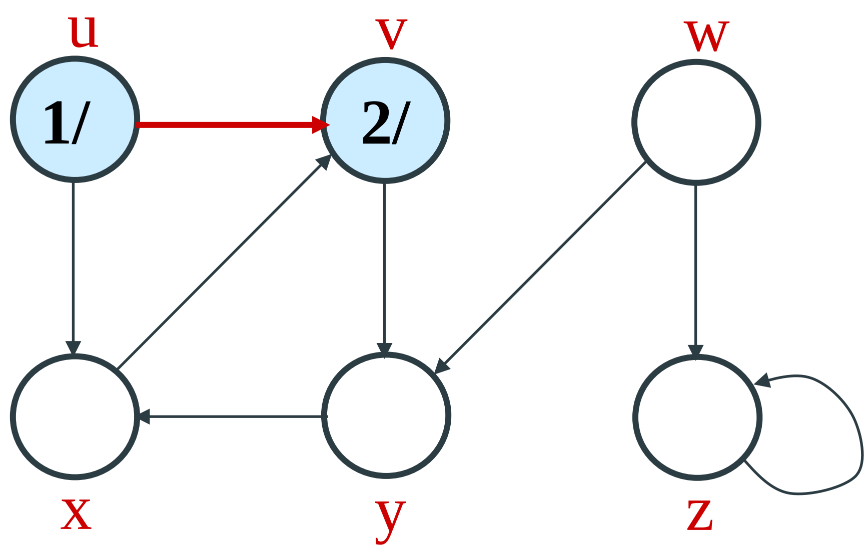
```
color[u] ← GRAY
  ∇ White vertex u has been discovered
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
      do if color[v] = WHITE
               then π[v] ← u
                    DFS-Visit(v)
color[u] ← BLACK
∇ Blacken u;  it is finished.
f[u] ← time ← time + 1
```

white: undiscovered
gray: discovered
black: finished

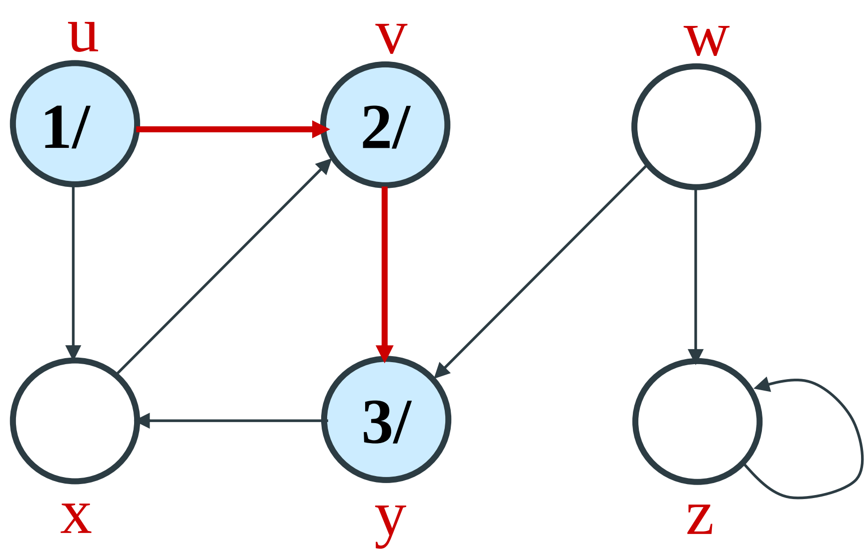
DFS Example



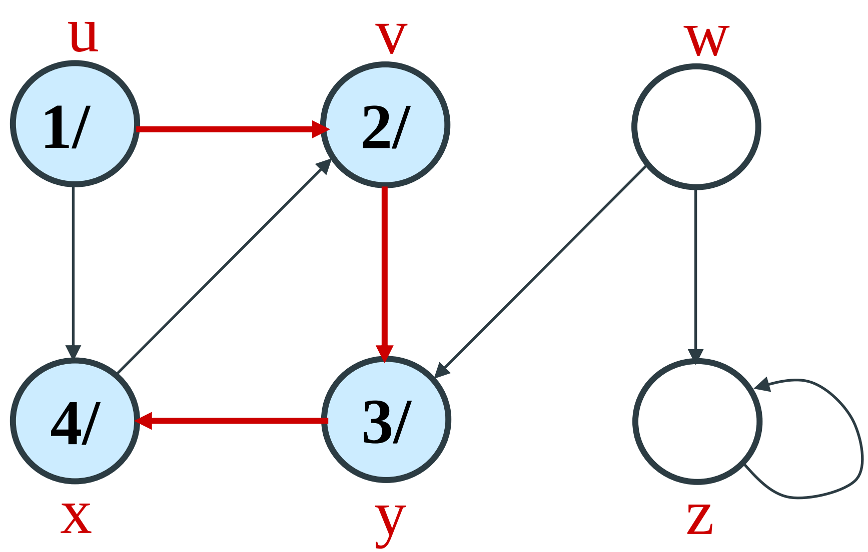
DFS Example



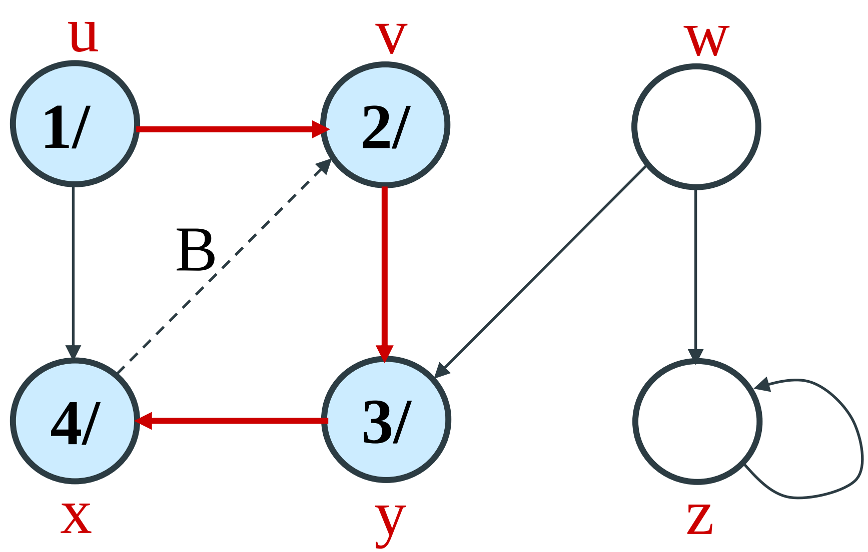
DFS Example



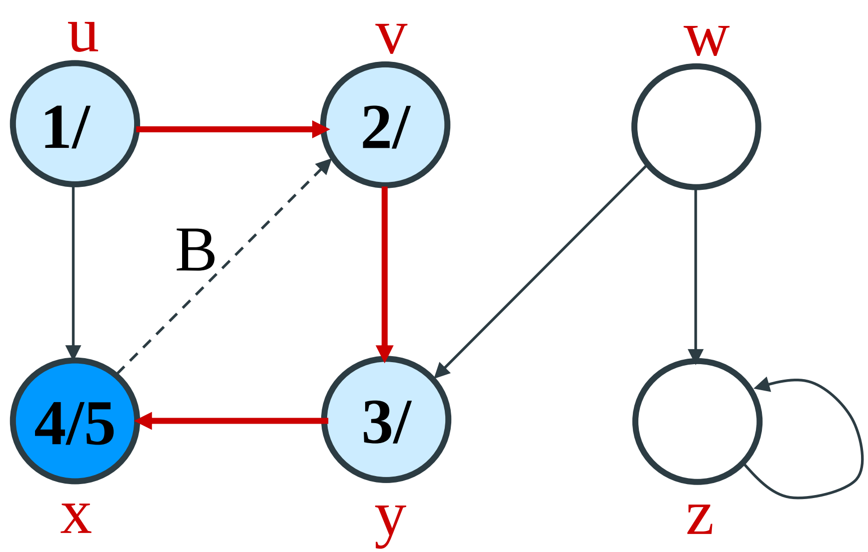
DFS Example



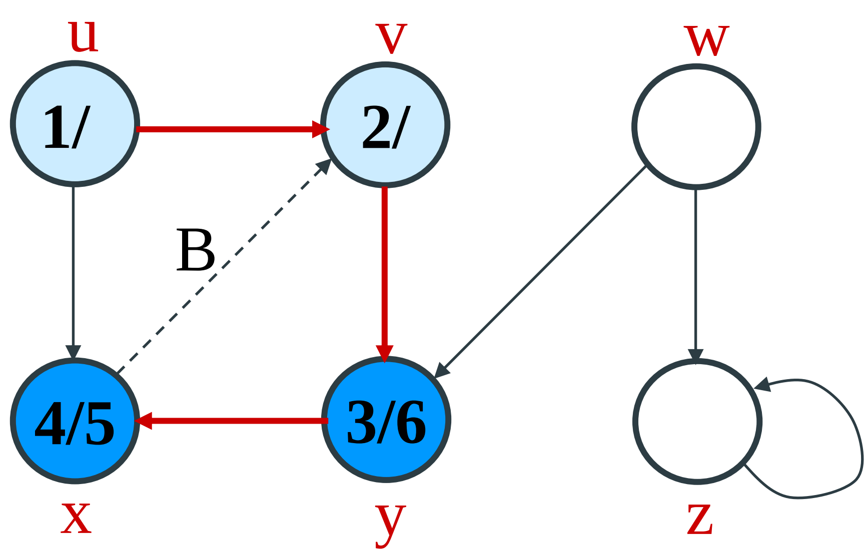
DFS Example



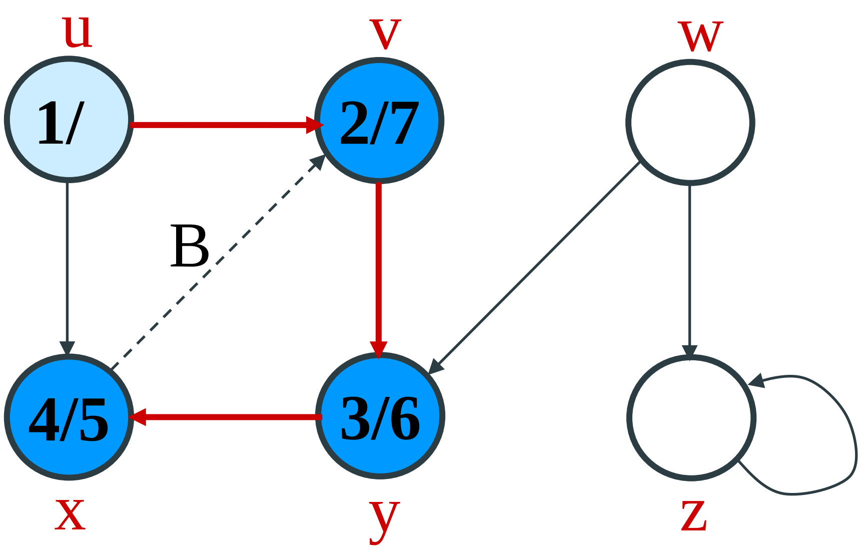
DFS Example



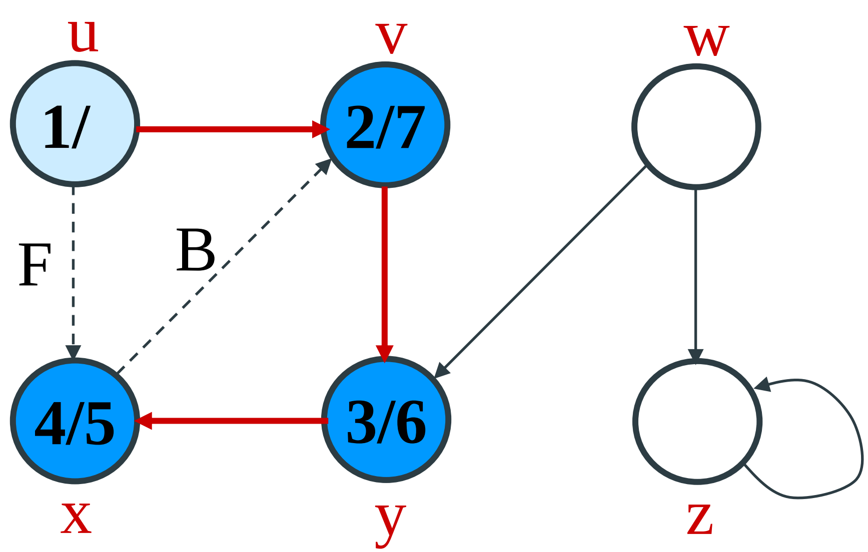
DFS Example



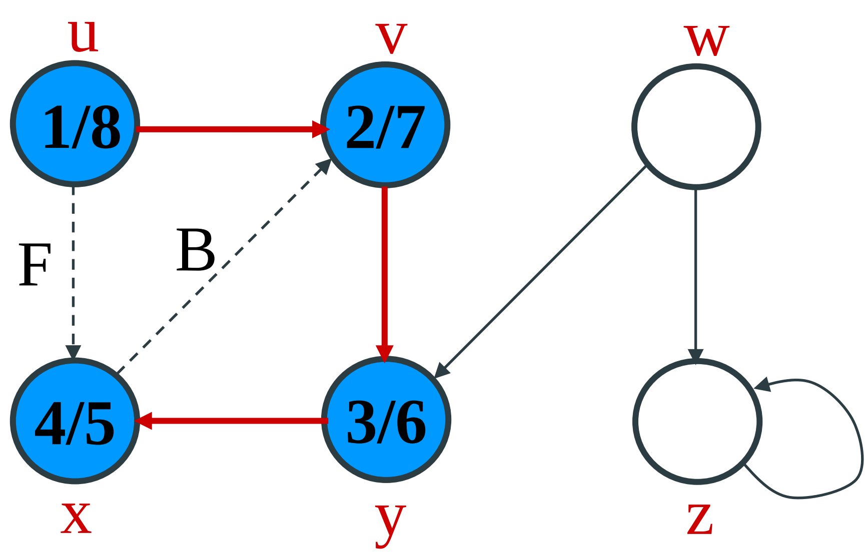
DFS Example



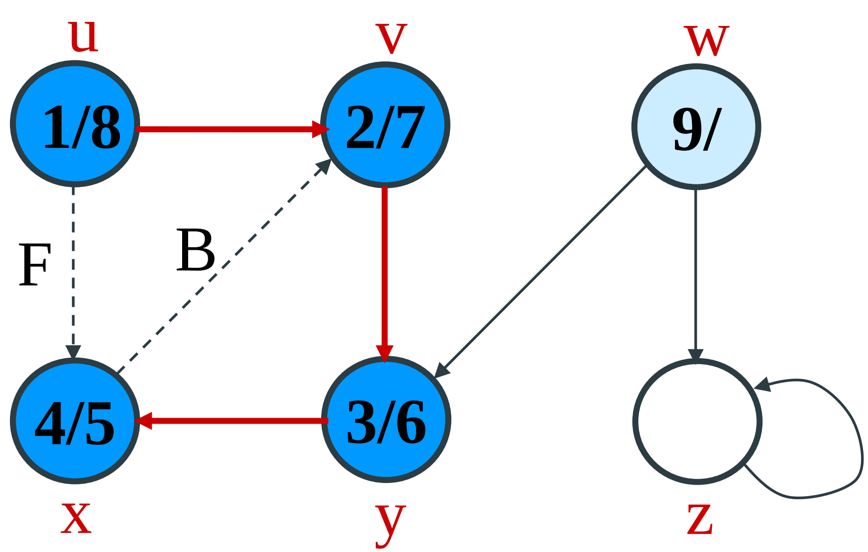
DFS Example



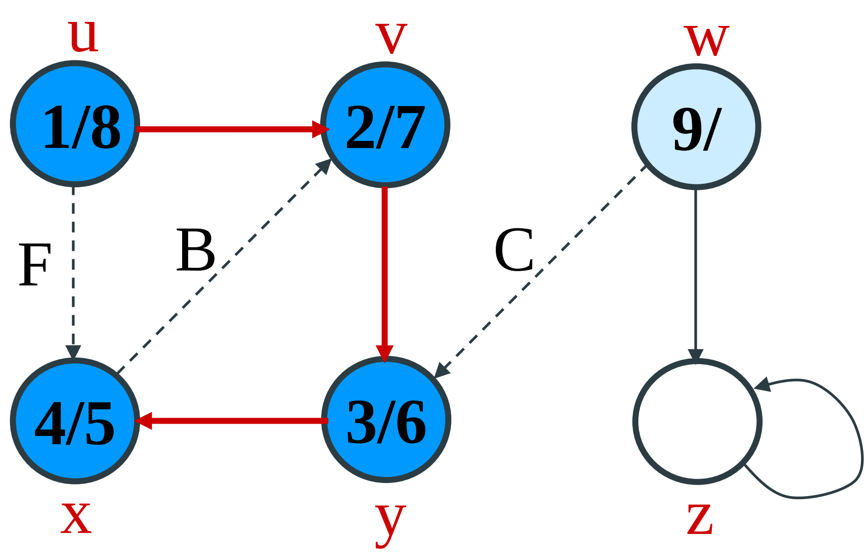
DFS Example



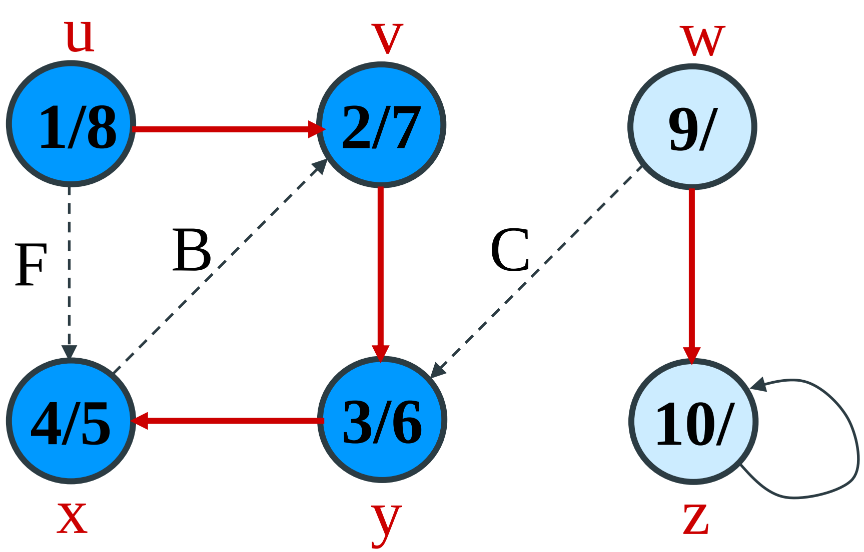
DFS Example



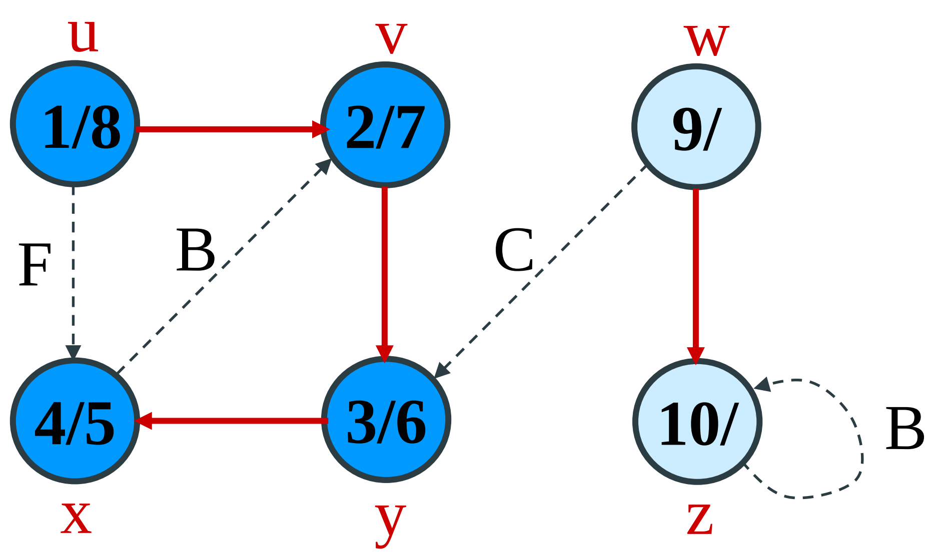
DFS Example



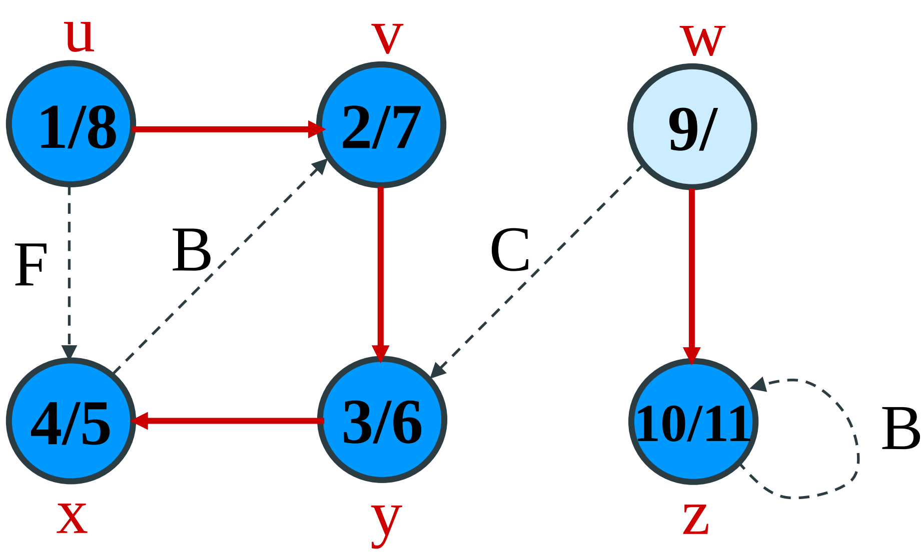
DFS Example



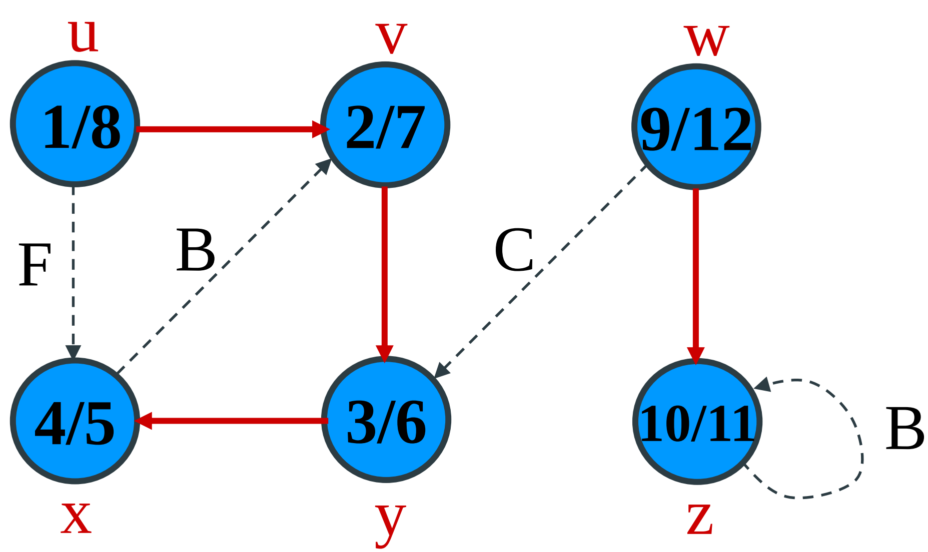
DFS Example



DFS Example



DFS Example



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 $|\mathrm{Adj}[v]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\mathrm{Adj}[v]| = \Theta(E)$

- Total running time of DFS is $\Theta(V+E)$

Depth-First Trees

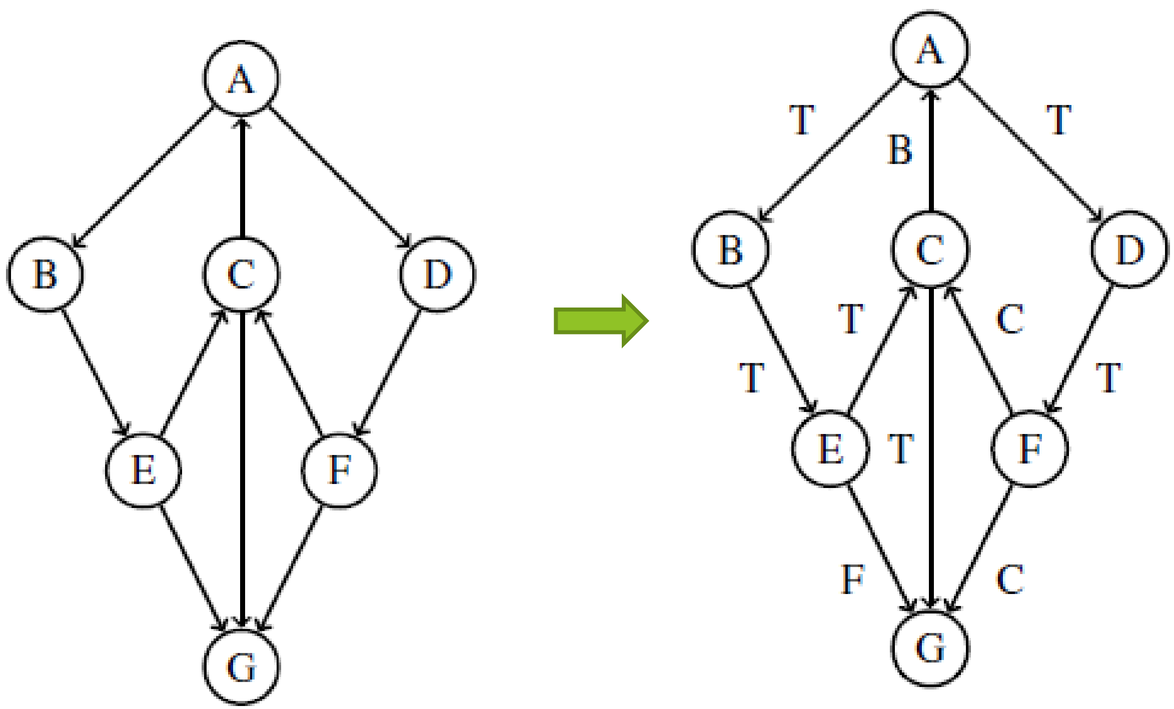
- Predecessor subgraph defined slightly different from that of BFS
- The predecessor subgraph of DFS is $G_\pi = (V, E_\pi)$ where $E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\}$.
  - How does it differ from that of BFS?
  - The predecessor subgraph G_π forms a *depth-first forest*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

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

Practice Problems

- Perform a depth-first search on the following graph starting at A. Break all ties by picking the vertices in alphabetical order (i.e A before Z).

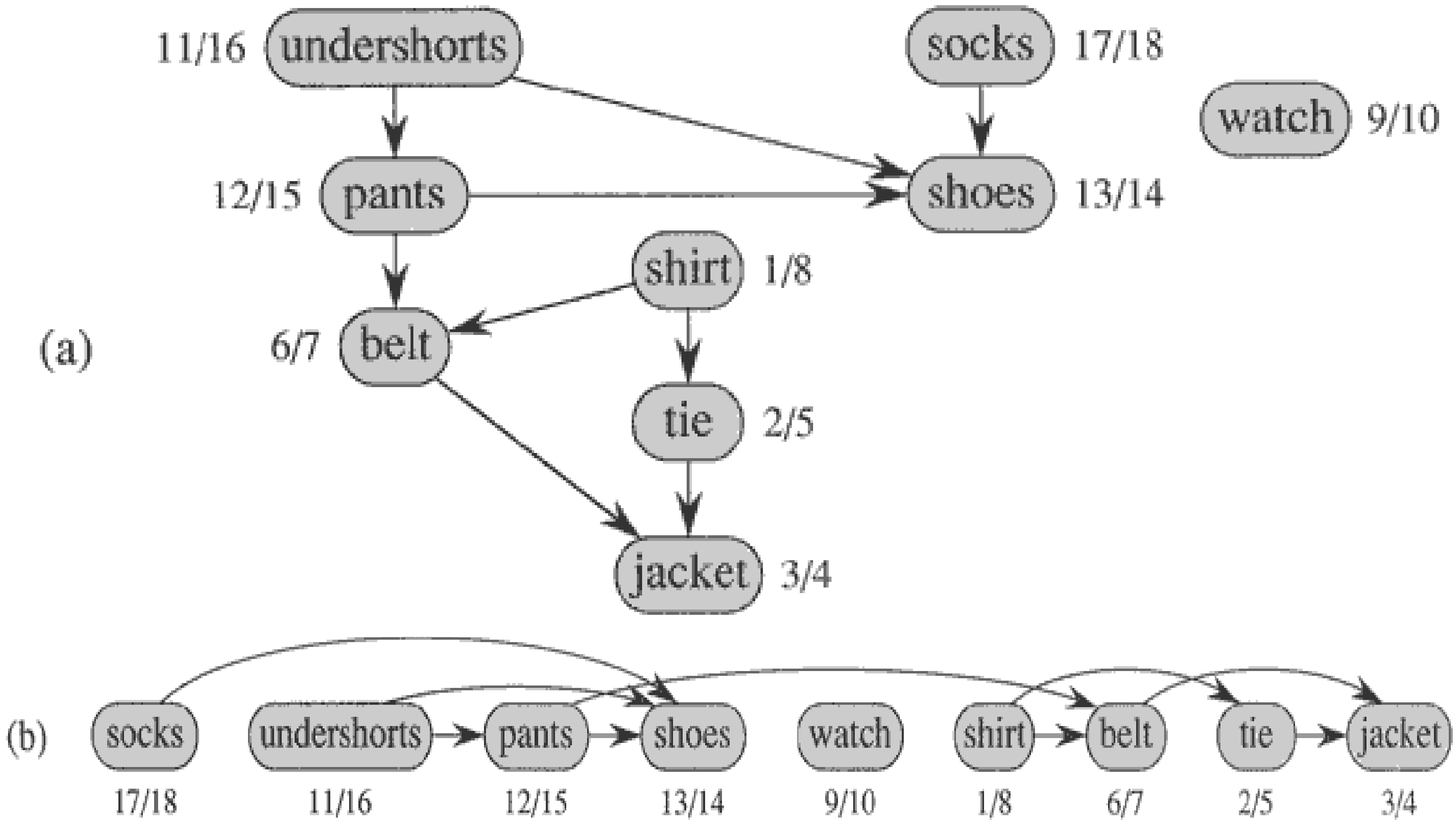


Topological Sort

- A topological sort of a Directed Acyclic Graph (DAG) is a linear order of all its vertices s.t. if G contains an edge (u, v), then u appears before v in the ordering
  - If the graph is not acyclic, then no linear ordering is possible.
  - A topological sort can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right
- TOPOLOGICAL-SORT(G) $\Theta(V+E)$

```
call DFS(G) to compute finishing times f[v] for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

Topological Sort



Thanks to contributors

Mr. Pham Van Nguyen (2022)

Dr. Thien-Binh Dang (2017 – 2022)

Prof. Hyunseung Choo (2001 – 2022)

Copyright 2000-2022 Intelligent Networking Laboratory