Graph Algorithms

Tanjina Helaly

Main Topics

- Graph representation
- Graph traversal
  - Breadth-first search
  - Depth-first search

Graphs

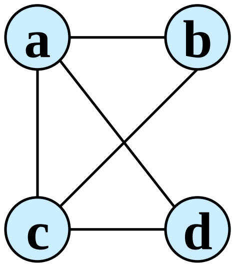
- *Graph* $G = (V, E)$
  - V = set of vertices
  - E = set of edges $\subseteq (V \times V)$
- Types of graphs
  - Undirected: edge $(u, v) = (v, u)$; for all v, $(v, v) \notin E$ (No self loops.)
  - Directed: (u, v) is edge from u to v, denoted as $u \rightarrow v$. Self loops are allowed.
  - Weighted: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

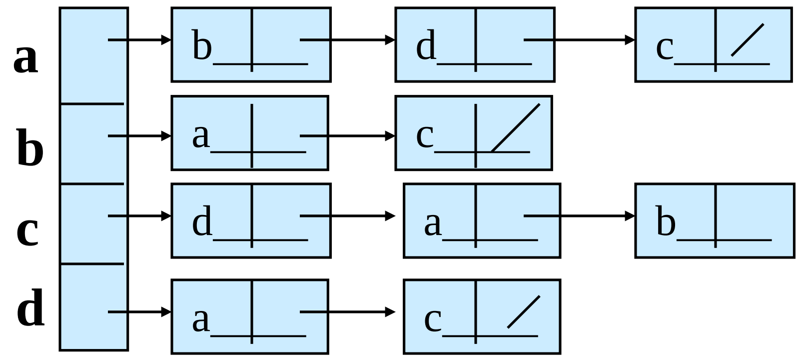
GRAPHS

- 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.
- Path
  - a **path** in a graph is a finite sequence of edges which connect a sequence of vertices.
    - Sequence of alternating vertices and edges.
    - Start from a vertex and ends to a vertex.
  - Simple path
    - All vertices and edges are distinct.
- Cycle
  - A path whose start and end vertex is same.

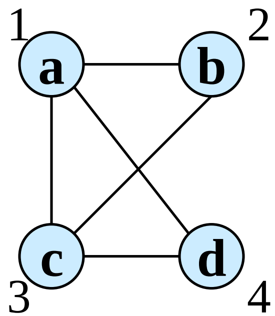
Representation of Graphs

- Two standard ways.
  - Adjacency Lists.





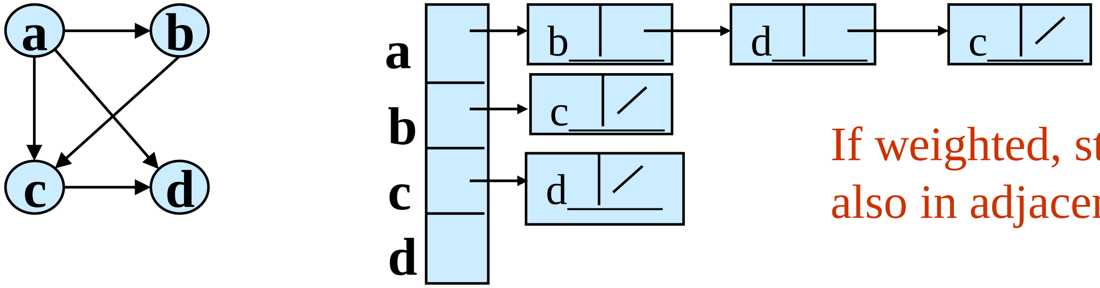
  - Adjacency Matrix.



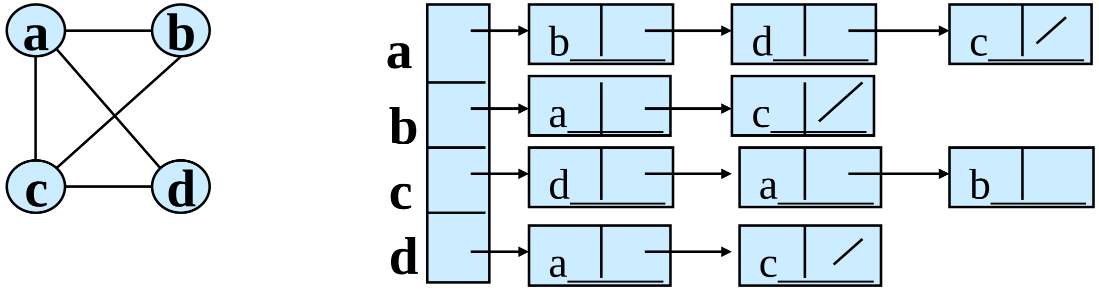
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

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*.

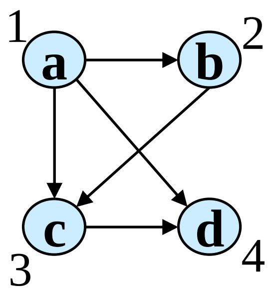
  - Total storage: $\Theta(V+E)$

Pros and Cons: adj list

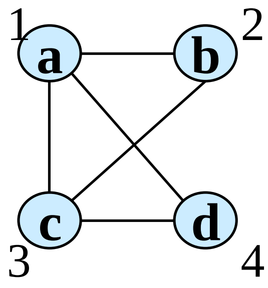
- Pros
  - Space-efficient, when a graph is sparse.
  - Can be modified to support many graph variants.
- Cons
  - Determining if an edge $(u,v) \in G$ is not efficient.
    - Have to search in u's adjacency list. $\Theta(\text{degree}(u))$ time.
    - $\Theta(V)$ in the worst case.

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



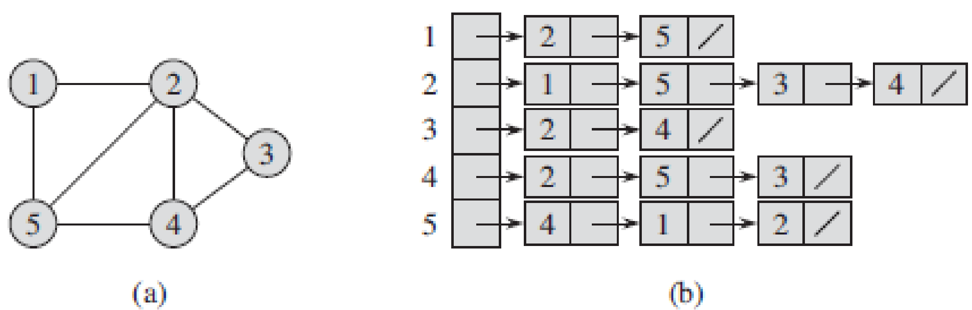
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

$A = A^{\mathrm{T}}$ 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.

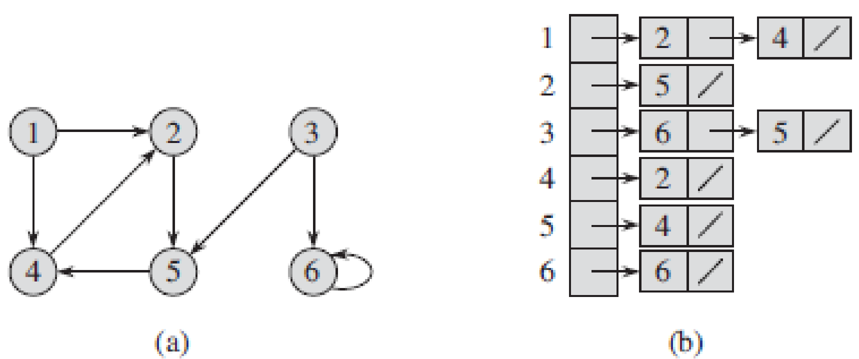
Graph Representation – List vs. Matrix



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

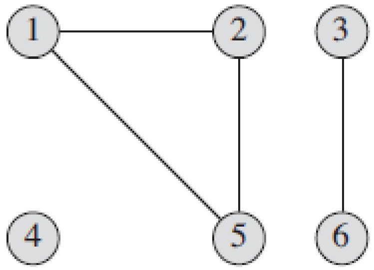
(c)

Graphs

- Reachable checking
  - If a path exists
- Cycle checking
  - a vertex is reachable from itself.

Connected Component

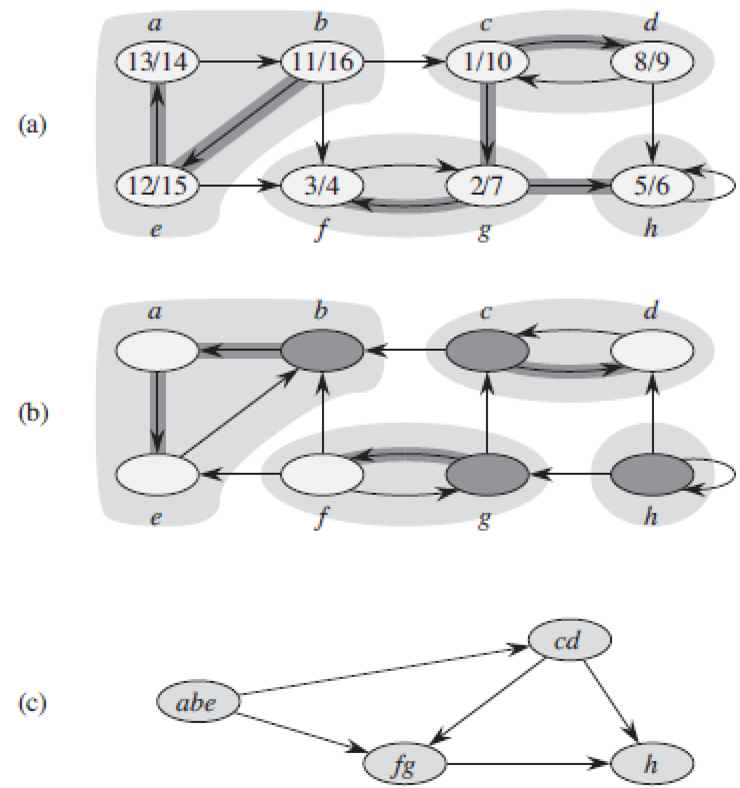
- An undirected graph is ***connected if there is a path between every pair of v***ertices.
- A **connected component** of an undirected graph is a maximal set of nodes such that each pair of nodes is connected by a path.



(b)

Strongly Connected Components

- A graph is **strongly connected** if every vertex is reachable from every other vertex.
- The ***strongly connected components of a directed graph*** *is the subgraph where every vertex is reachable from every other vertices.*



Connected Components

- Connected (Strongly or not) components form a **partition** of the set of graph vertices, meaning
  - connected components are non-empty,
  - they are pair-wise **disjoints**, and
  - the union of connected components forms the set of all vertices.

- Algorithm to find the strong connectivity of a graph takes linear time ($\Theta(V+E)$).

Connected Components

- Connected components count
  - Count of disjoint connected components
- Size of connected component
  - number of vertices in a component

- Figure
  - Fig 1: has 3 connected components with 3 different sizes
  - Fig 2: has 4 connected components with 3 different sizes

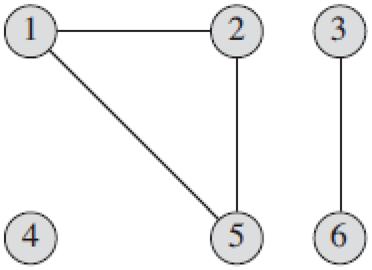


Fig 1

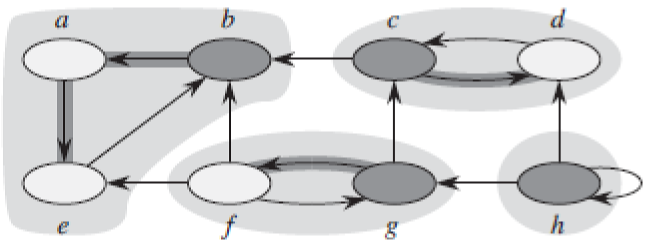


Fig 2

TREE

- **Tree**: Connected Acyclic undirected graph
- **Forest**: If an undirected graph is acyclic but possibly disconnected, it is a ***forest***
- **Spanning Tree**: is a tree which includes all of the vertices of the graph.

Appendix B Sets, Etc.

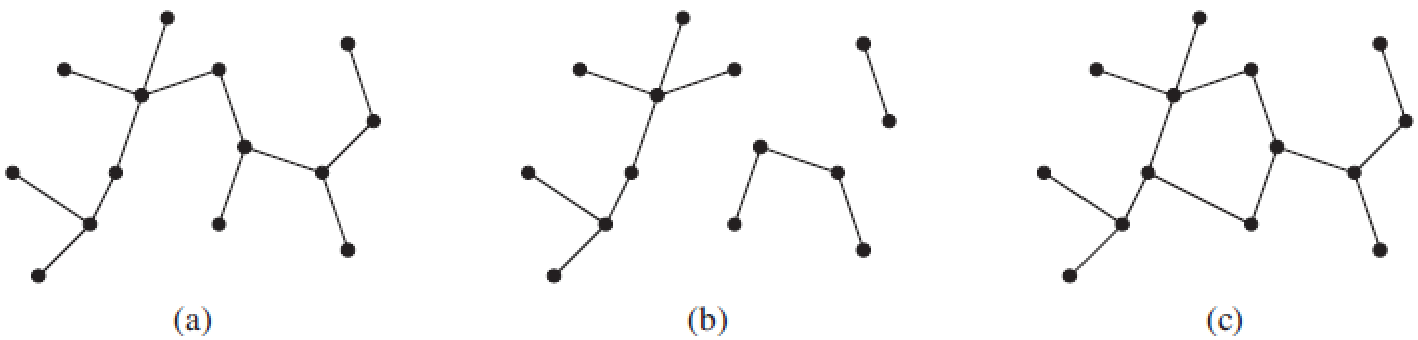


Figure B.4 (a) A free tree. (b) A forest. (c) A graph that contains a cycle and is therefore neither a tree nor a forest.

Properties of Free Trees

- Let G ={V;E} be an undirected graph. The following statements are equivalent.
  - Any two vertices in G are connected by a unique simple path.
  - G is connected, acyclic and $|E|=|V| - 1$.
  - G is connected, but if any edge is removed from E, the resulting graph is disconnected.
  - G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

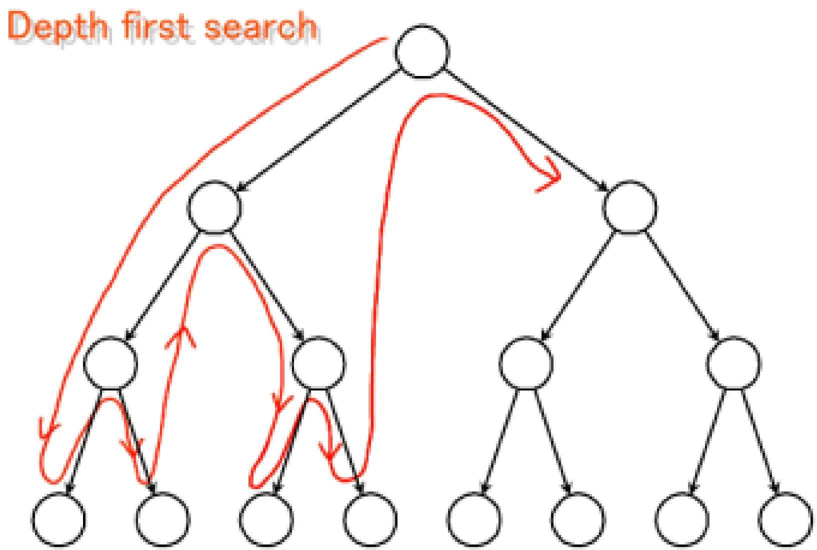
Graph-searching Algorithms

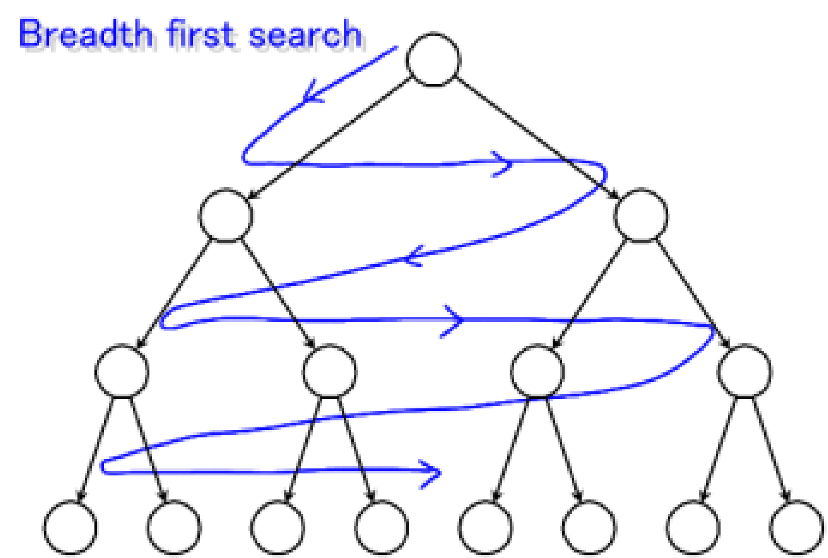
- Given: a graph G = (V, E), directed or undirected
- Goal: methodically explore every vertex and every edge
- Ultimately: build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Note: might also build a *forest if graph is not connected*
- Used to discover the structure of a graph.

Graph-searching Algorithms

- Standard graph-searching algorithms.
  - Breadth-first Search (BFS).
  - Depth-first Search (DFS).
    - "Search as deep as possible first."

Graph Traversals





- Both take time: O(V+E)

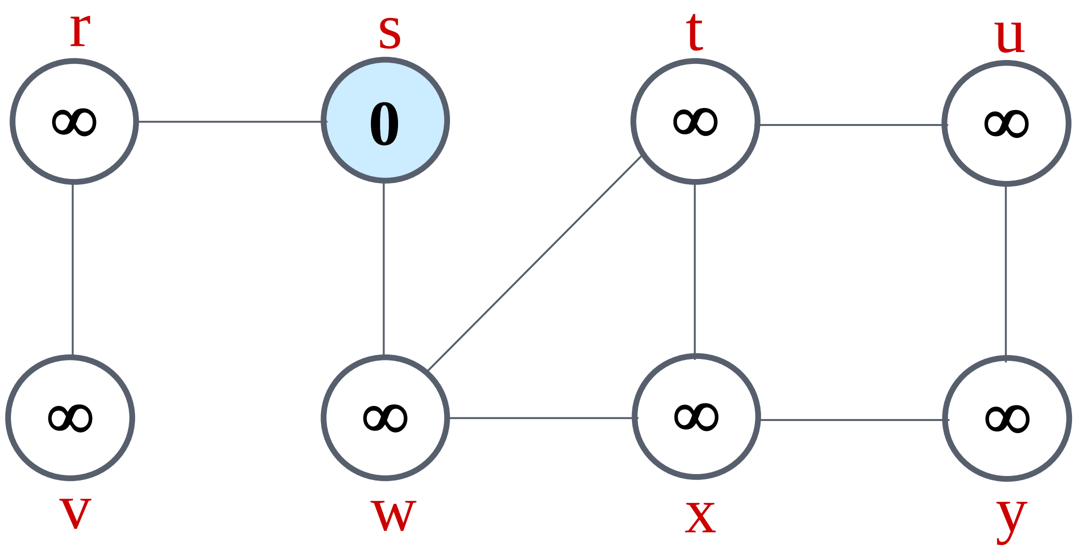
BFS(G,s)

```
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

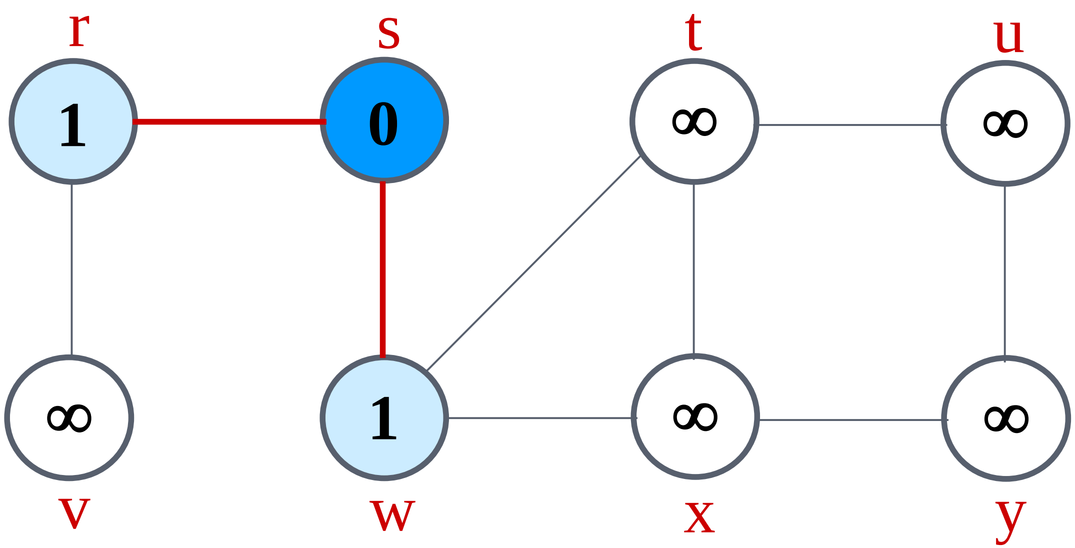
Q: a queue of discovered vertices
color[*v*]: color of v
d[*v*]: distance from s to v
π[*u*]: predecessor of v

EXAMPLE (BFS)



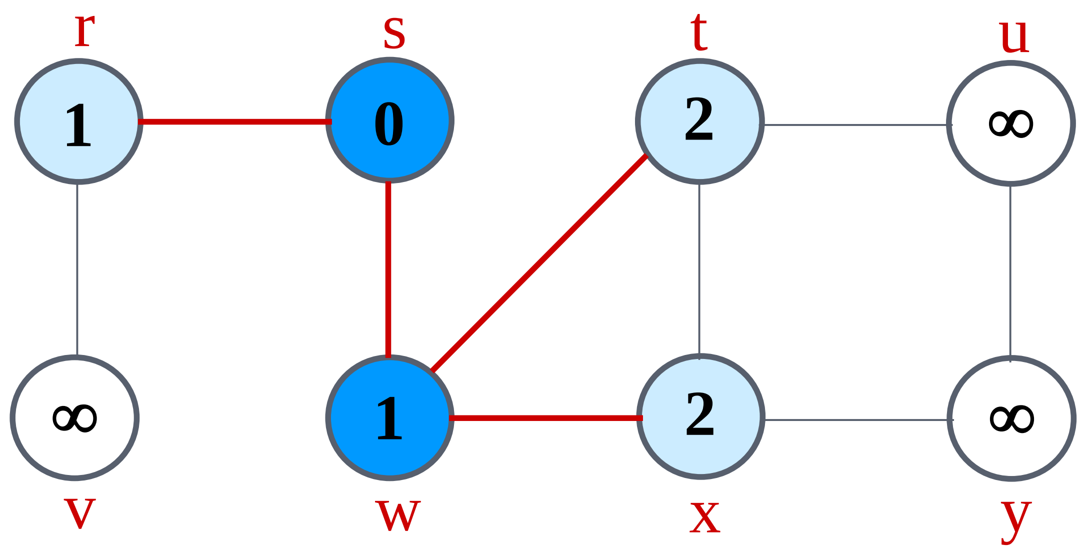
Q:	s
	0

Example (BFS)



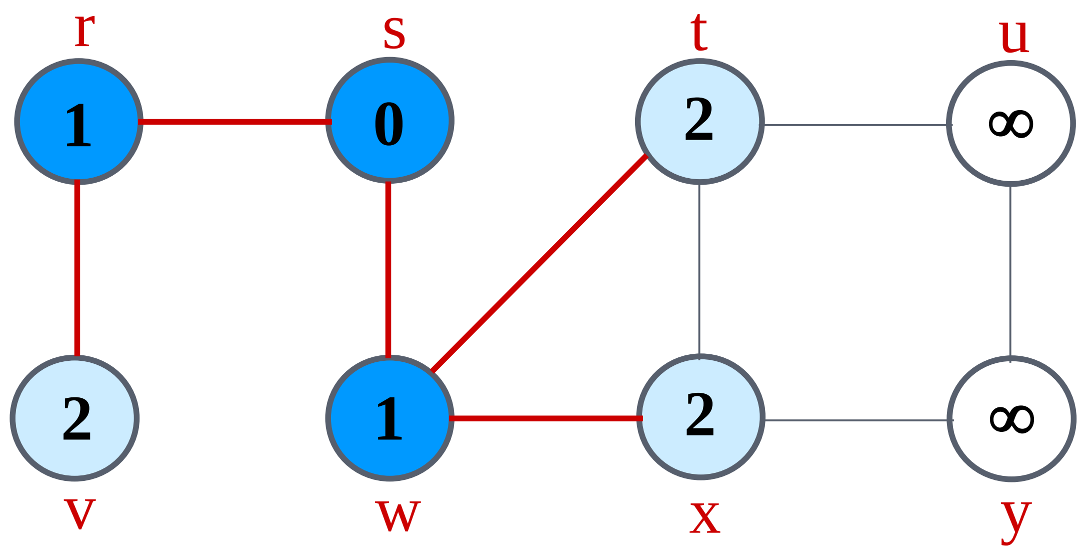
Q:	w	r
	1	1

Example (BFS)



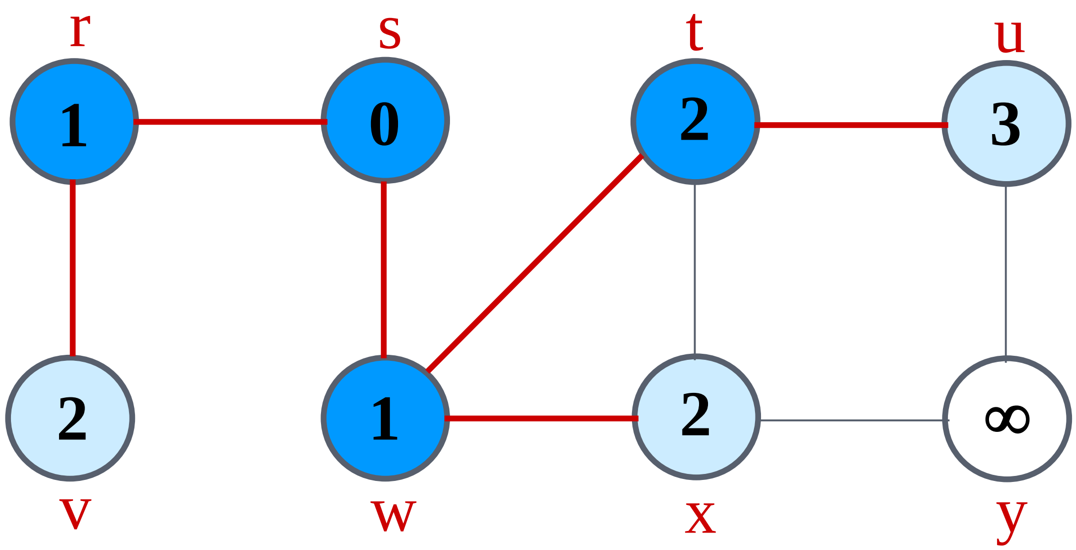
Q:	r	t	x
	1	2	2

Example (BFS)



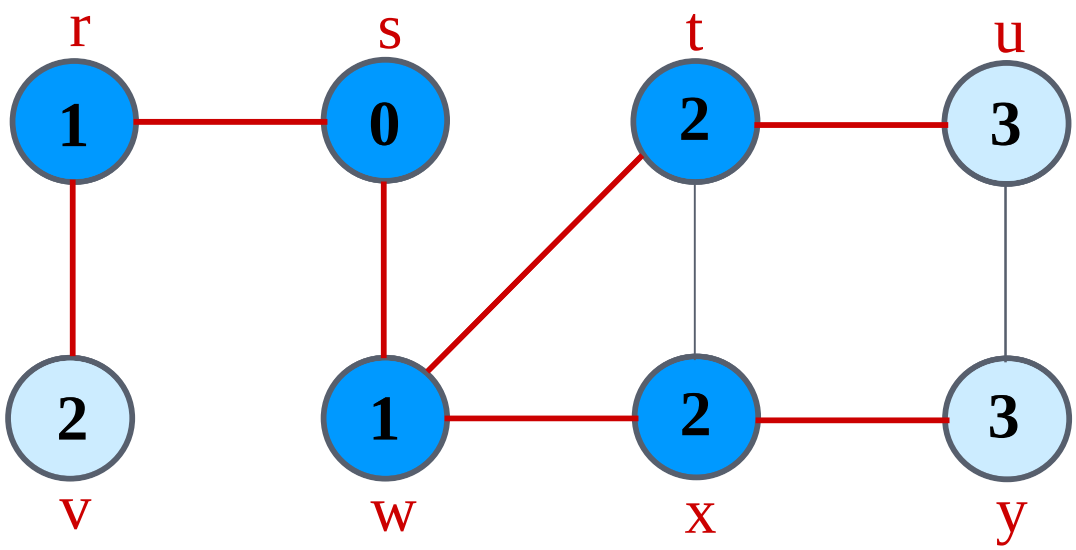
Q: t x v
2 2 2

Example (BFS)



Q: x v u
2 2 3

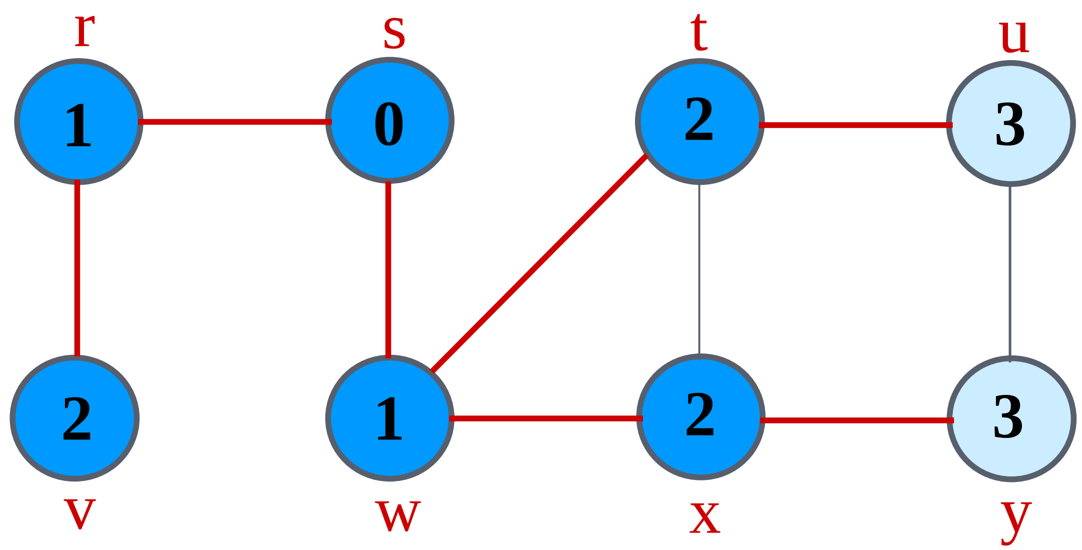
Example (BFS)



Q: v u y

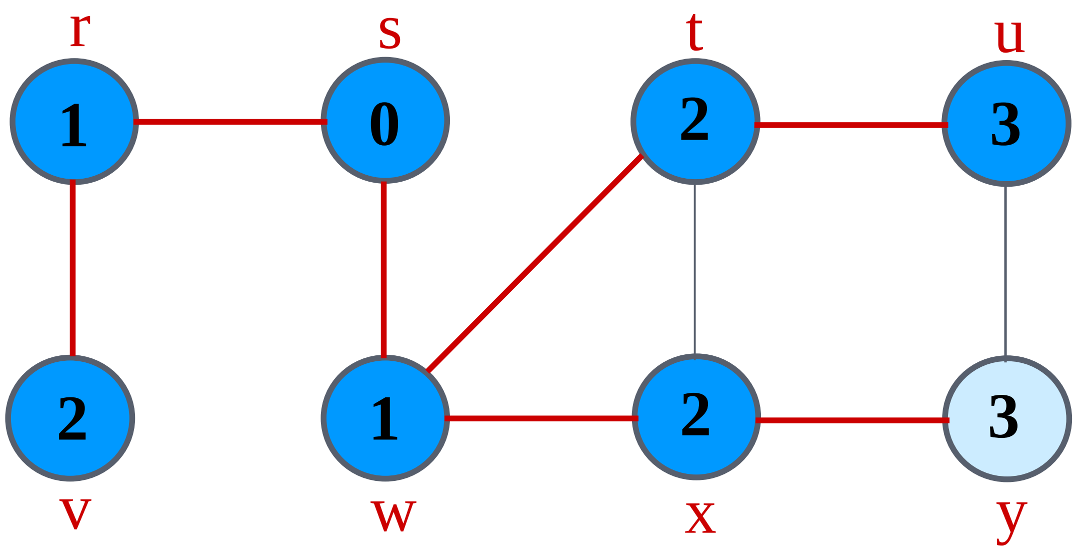
2 3 3

Example (BFS)



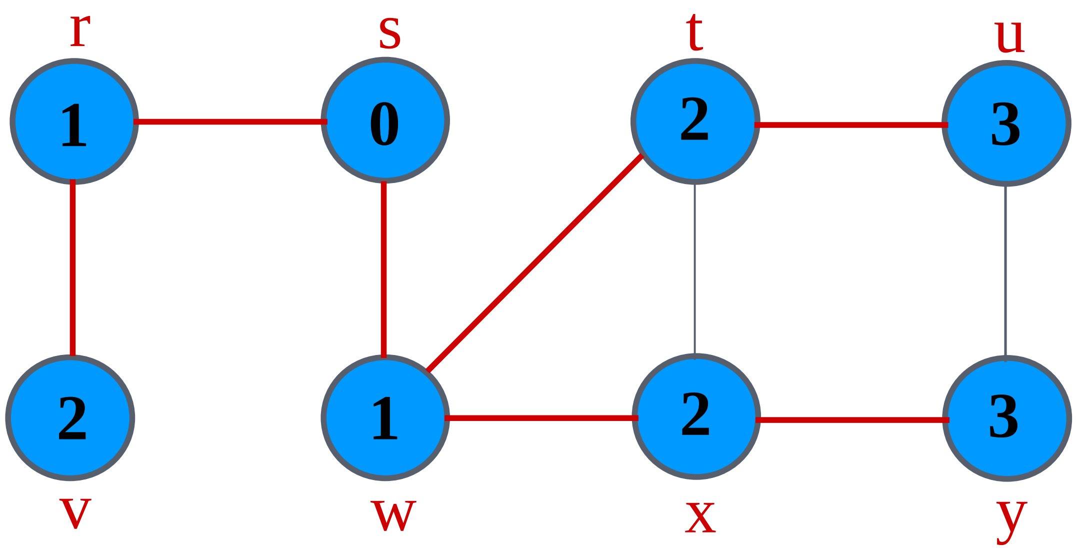
Q:	u	y
	3	3

Example (BFS)



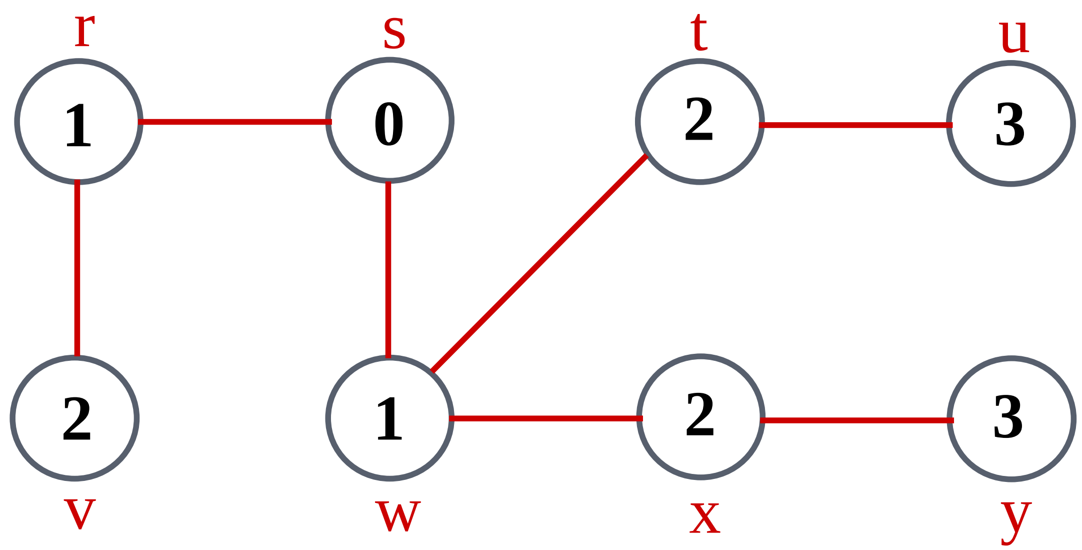
Q: y
3

Example (BFS)



Q: $\varnothing$

Example (BFS)



BF Tree

Application of BFS

- Finding Shortest Path
  - network (Computer, transportation[highway, flight])
  - GPS finding direction with shortest distance
  - Broadcasting of network node
  - Social network find people within k distance
  - **Crawlers in Search Engines**

DEPTH-FIRST SEARCH

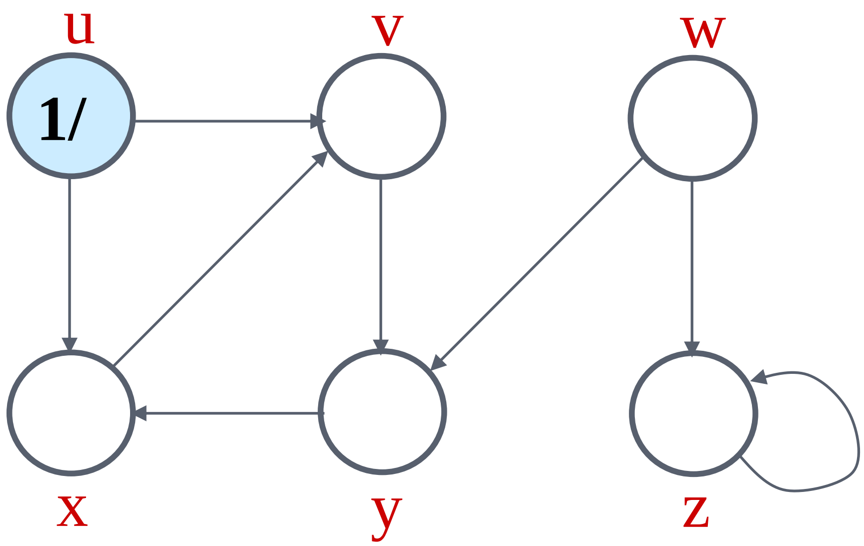
PSEUDO-CODE

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

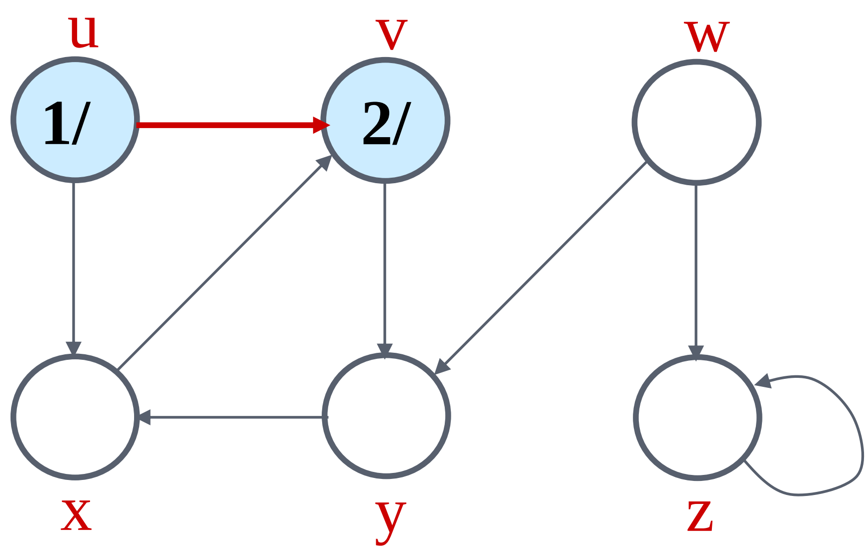
```
DFS-VISIT(G, u)
time = time + 1                  // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]            // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                  // blacken u; it is finished
time = time + 1
u.f = time
```

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

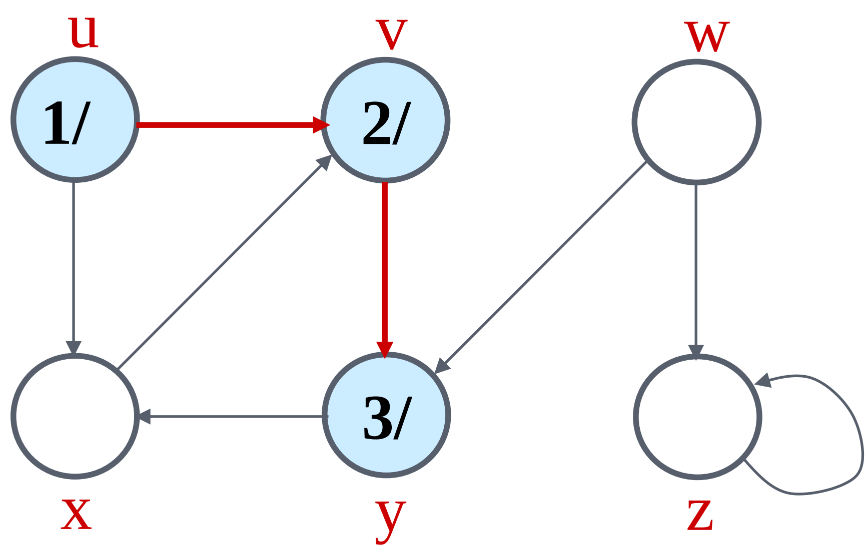
EXAMPLE (DFS)



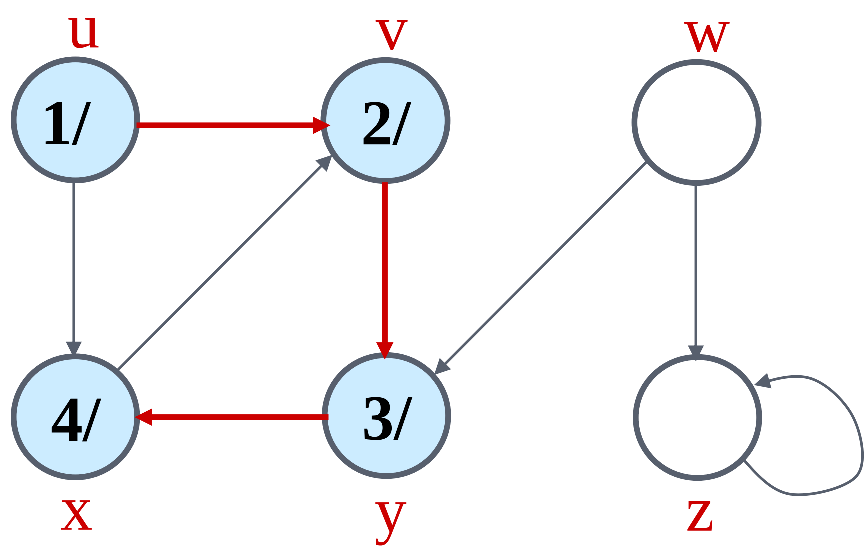
Example (DFS)



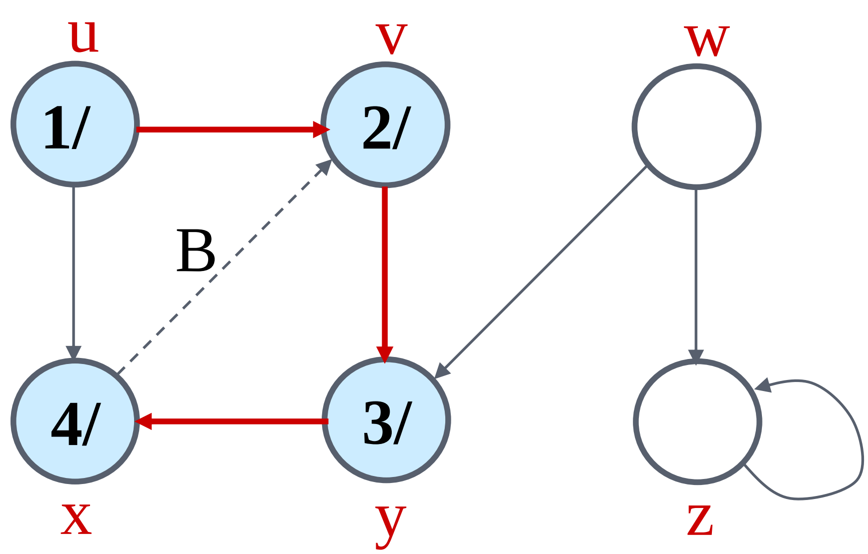
Example (DFS)



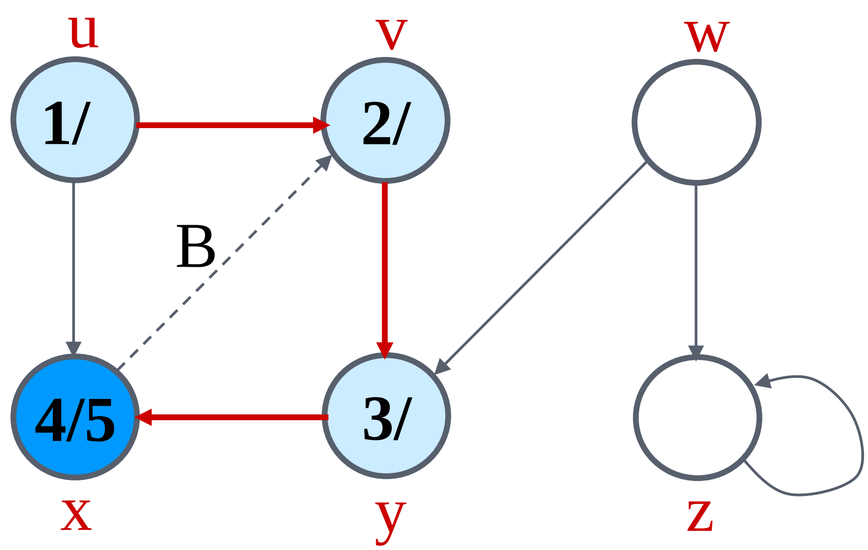
Example (DFS)



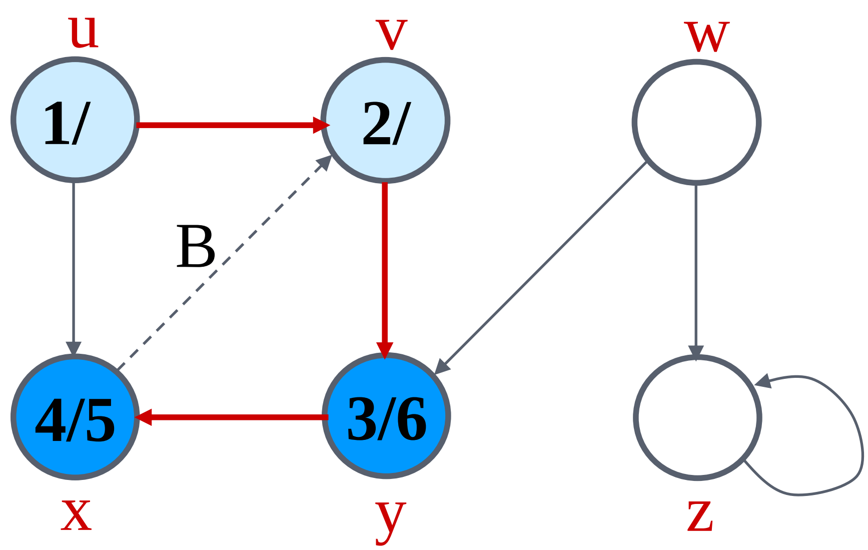
EXAMPLE (DFS)



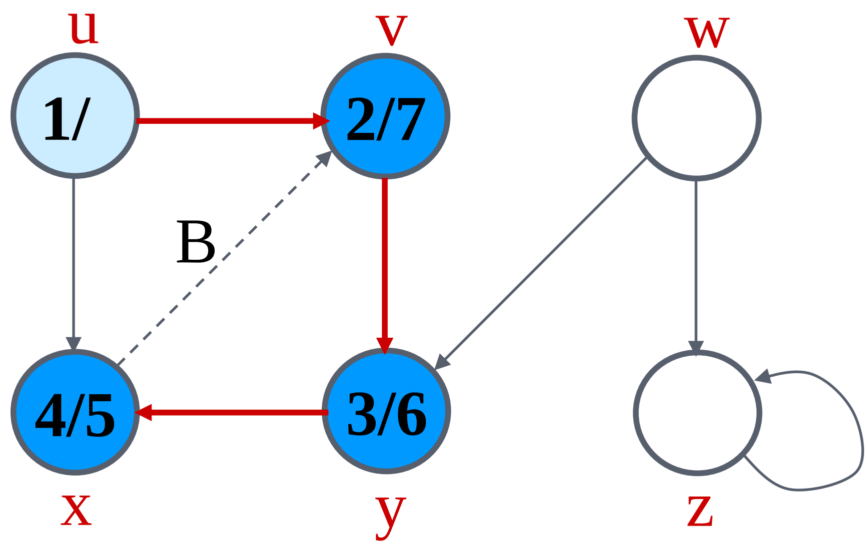
Example (DFS)



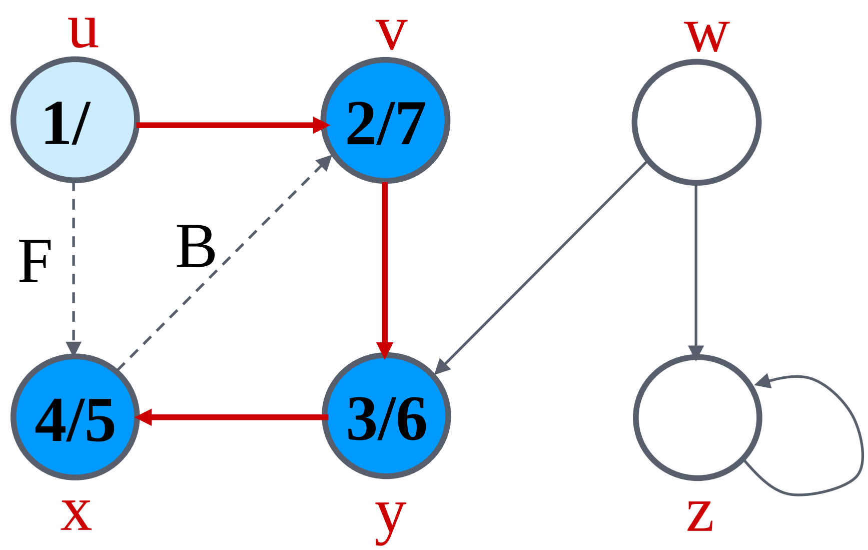
Example (DFS)



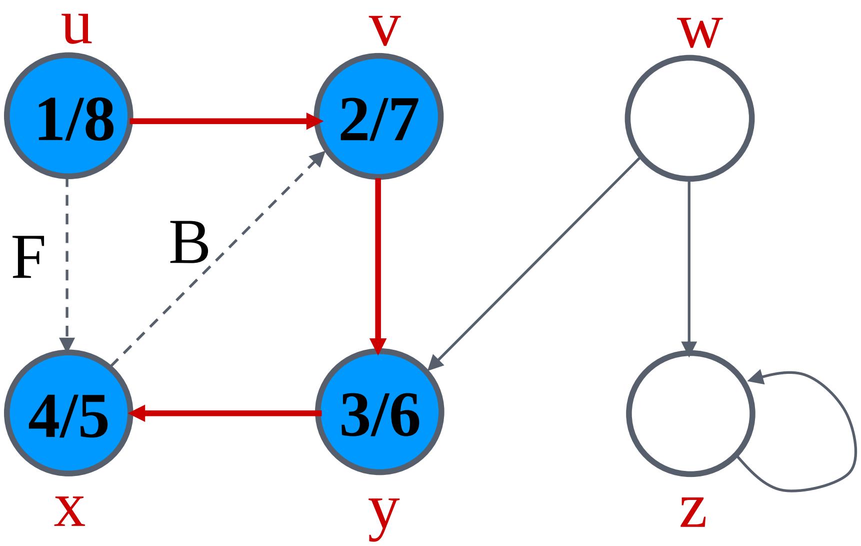
Example (DFS)



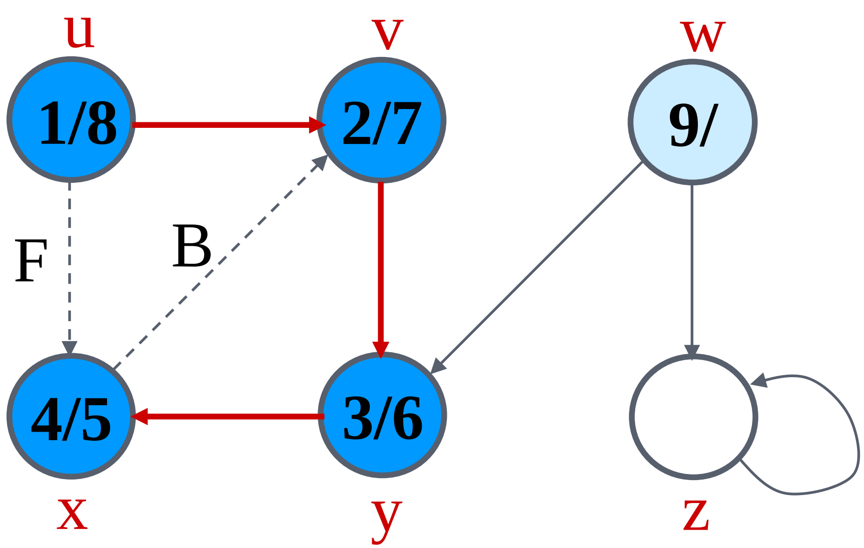
EXAMPLE (DFS)



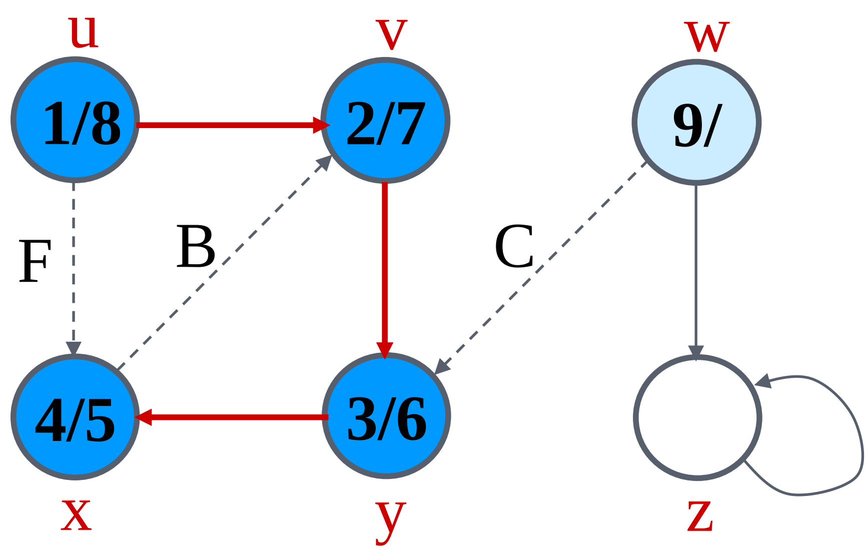
Example (DFS)



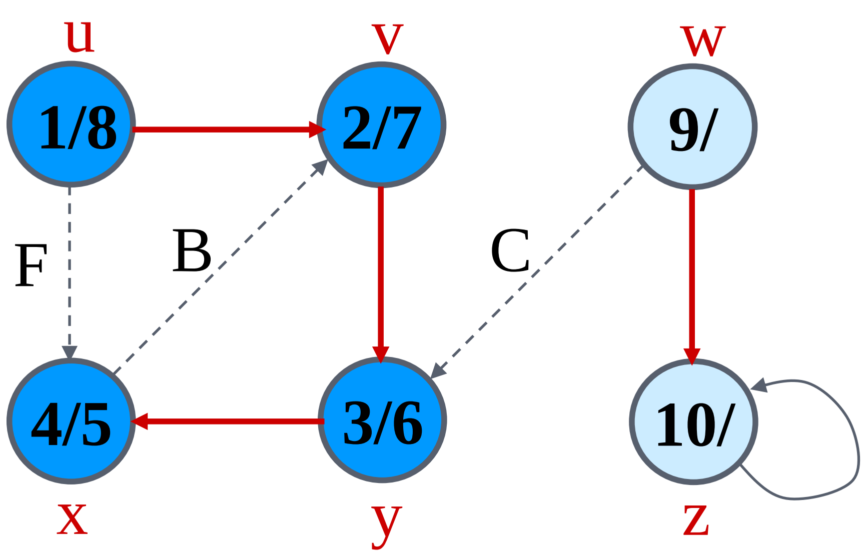
Example (DFS)



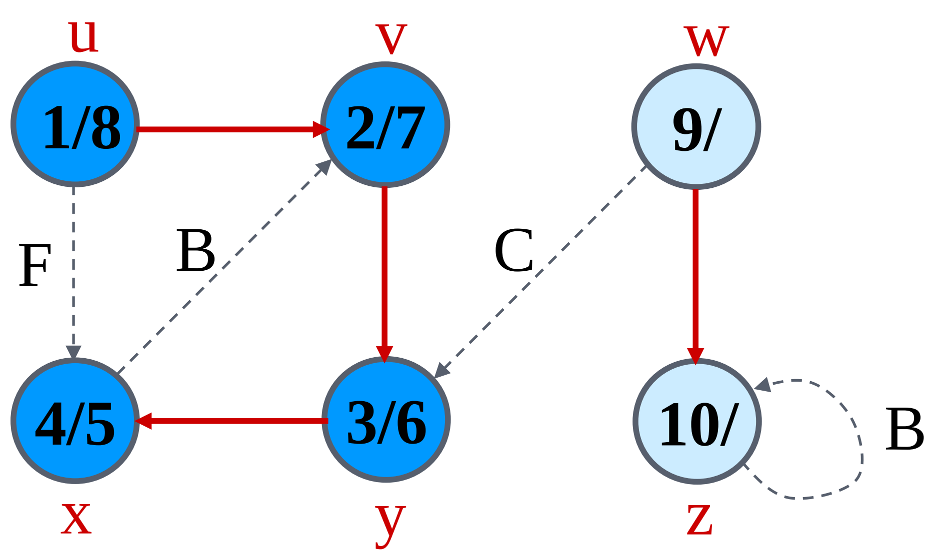
Example (DFS)



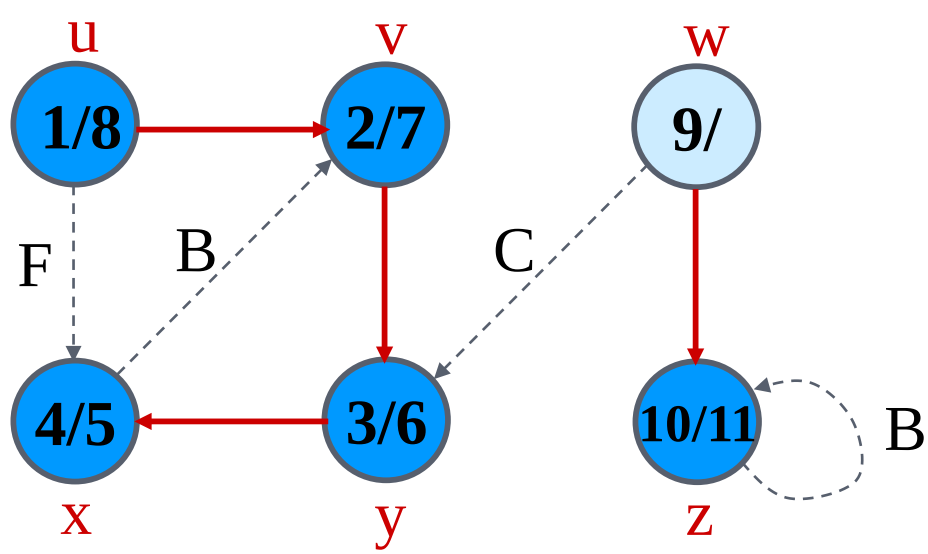
Example (DFS)



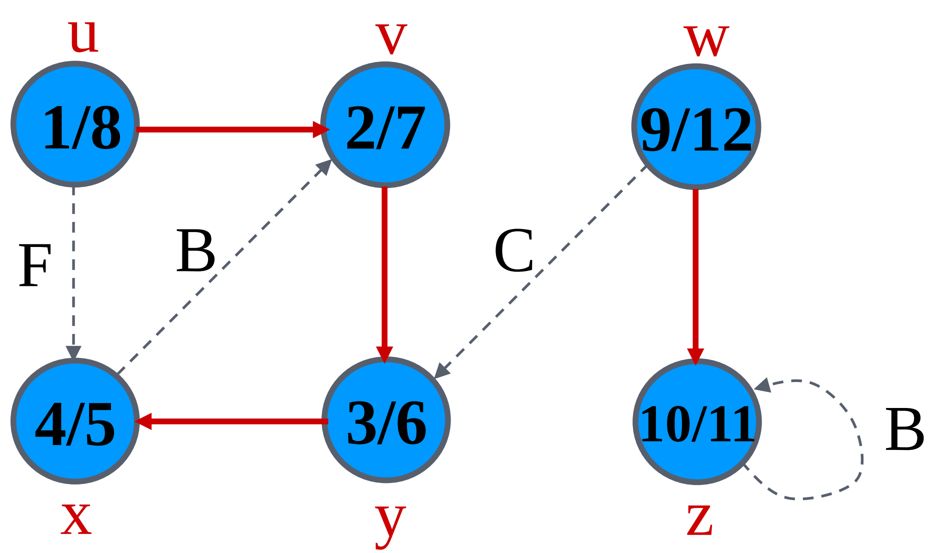
Example (DFS)



Example (DFS)



Example (DFS)



Application of DFS

- Finding path between 2 vertices.
- Finding connected components
- Topological sorting (Dependency resolution)
  - mainly used for scheduling a sequence of jobs or tasks based on their dependencies.

Reference

- Chapter 22 (22.1, 22.2, 22.3, 22.4) (Cormen)