

# COMP 251

Algorithms & Data Structures (Winter 2022)

## Graphs - Introduction

---

School of Computer Science  
McGill University

Slides of (Comp321 ,2021), Langer (2014), Kleinberg & Tardos, 2005 & Cormen et al., 2009, Jaehyun Park' slides CS 97SI, Top-coder tutorials, T-414-AFLV Course, Programming Challenges books, slides from D. Plaisted (UNC) and Comp251-Fall McGill.

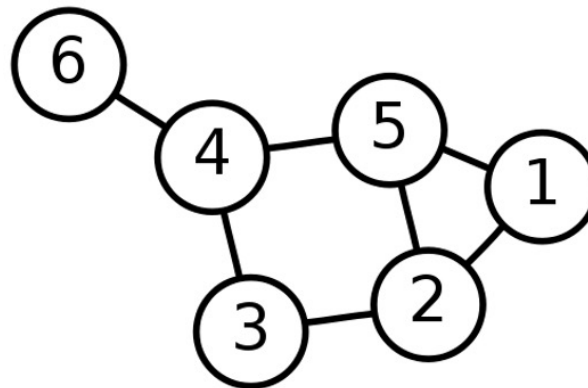
# Outline

- Graphs.
  - Introduction.
    - BFS.
    - DFS.
  - Strong Connected Components / Topological Sort.
  - Network Flow 1.
  - Network Flow 2.
  - Shortest Path.
  - Minimum Spanning Trees.
  - Bipartite Graphs.

# Graphs - Definition

- An abstract way of representing connectivity using nodes (also called vertices) and edges.
- $m$  edges connect some pairs of nodes.
  - Edges can be either one-directional (directed) or bidirectional.
- Nodes and edges can have some auxiliary information.

*like cities connected by highways*

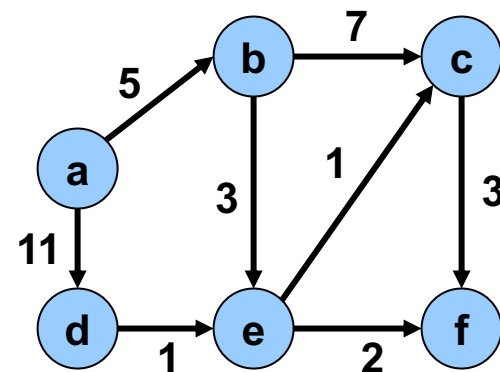
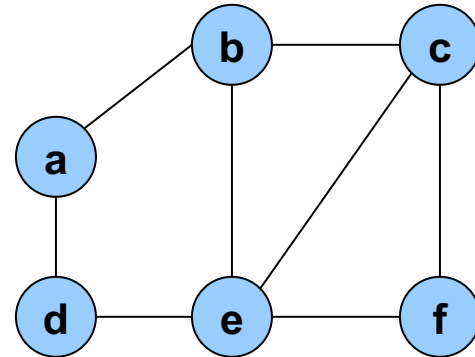


*model networks with graphs*

Figure from Wikipedia

# Graphs - Definition

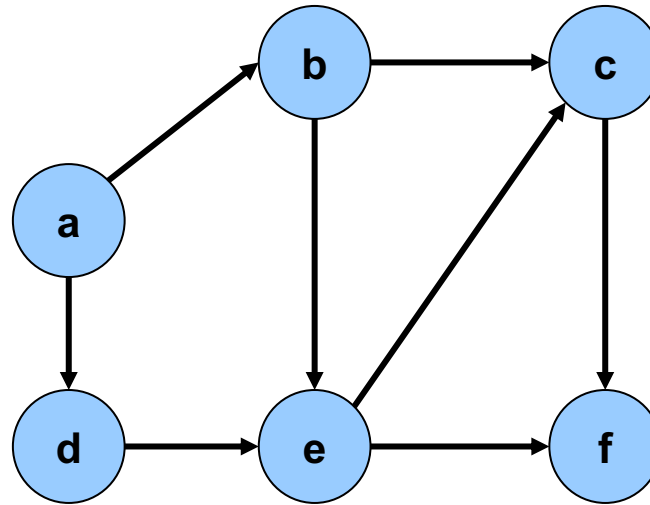
- **Graph**  $G = (V, E)$ 
  - $V$  = set of vertices
  - $E$  = set of edges  $\subseteq (V \times V)$
- **Types of graphs**
  - Undirected: edge  $(u, v) = (v, u)$ ; for all  $v$ ,  $(v, v) \notin E$  (No self loops.)
  - Directed:  $(u, v)$  is edge from  $u$  to  $v$ , denoted as  $u \rightarrow v$ . 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:  $|E| \ll |V|^2$ .
- $|E| = O(|V|^2)$



# Graphs - Properties

- If  $(u, v) \in E$ , then vertex  $v$  is adjacent to vertex  $u$ .
- Adjacency relationship is:
  - Symmetric if  $G$  is undirected.  $(u,v) \sim (v,u)$
  - Not necessarily so if  $G$  is directed.
- If  $G$  is (strongly) connected:
  - There is a path between every pair of vertices.
  - $|E| \geq |V| - 1$ .
  - Furthermore, if  $|E| \equiv |V| - 1$ , then  $G$  is a tree.

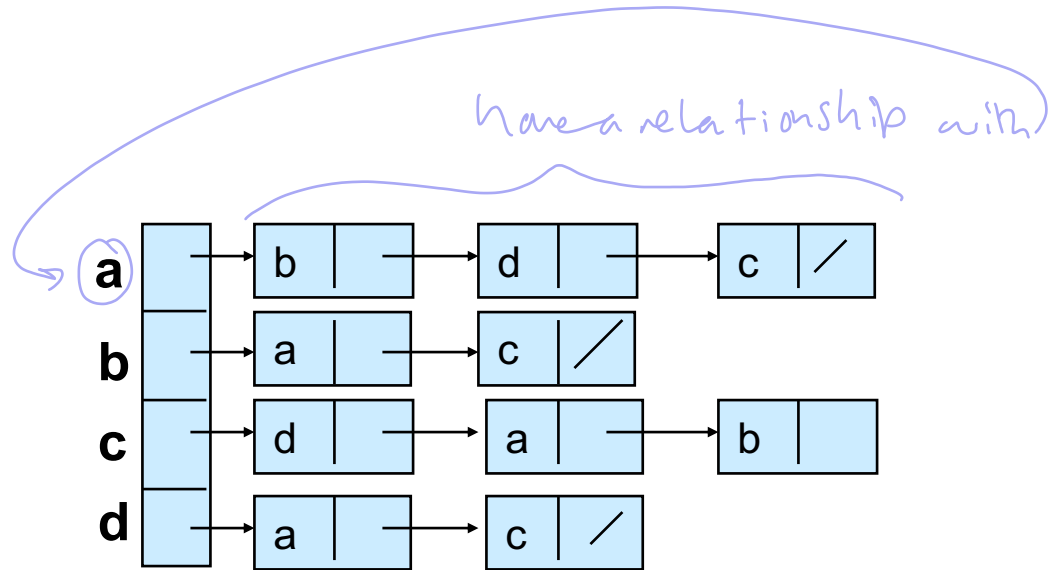
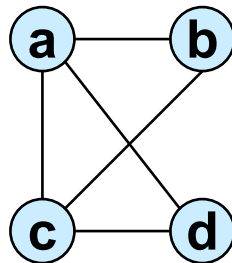
# Graphs - Vocabulary



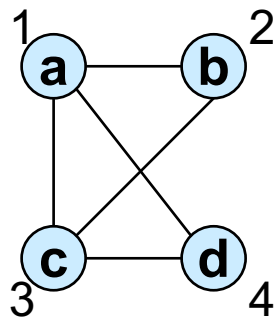
- Ingoing edges of  $u$ :  $\{ (v,u) \in E \}$  (e.g.  $\text{in}(e) = \{ (b,e), (d,e) \}$  )
- Outgoing edges of  $u$ :  $\{ (u,v) \in E \}$  (e.g.  $\text{out}(d) = \{ (d,e) \}$  )
- In-degree( $u$ ):  $|\text{in}(u)|$  // Number of predecessors
- Out-degree( $u$ ):  $|\text{out}(u)|$  // Number of successors

# Graphs - Representation

- Two standard ways.
  - Adjacency Lists.



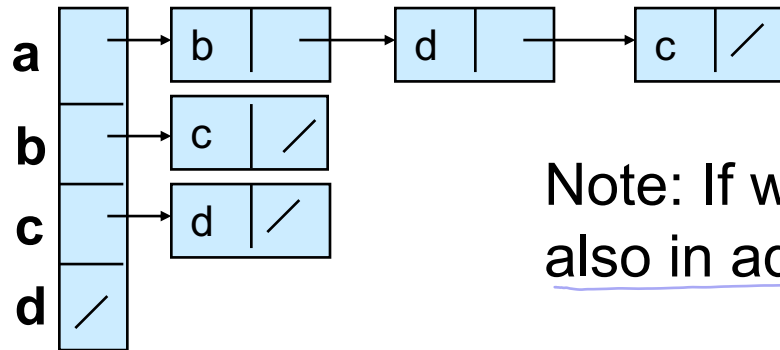
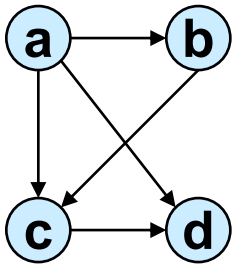
- Adjacency Matrix.



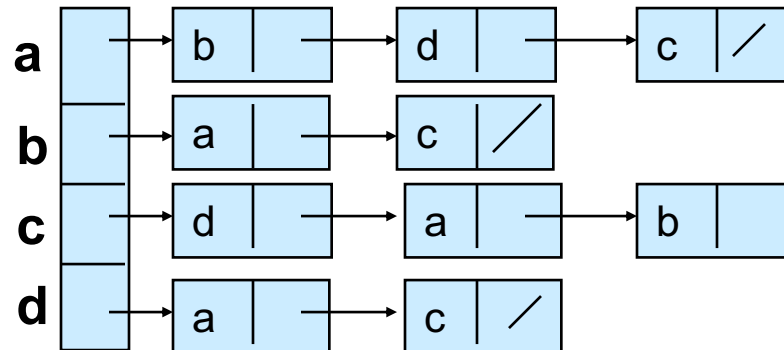
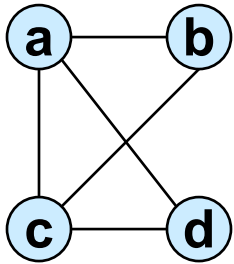
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

# Graphs – Representation – Adjacency List

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



Note: If weighted, store weights  
also in adjacency lists.





# Adjacency List – 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:  $\Theta(V+E)$

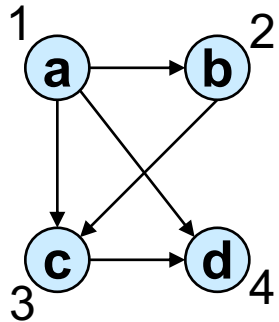
# Adjacency List – Pros and Cons

- 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.  $\Theta(\text{degree}(u))$  time.
    - $\Theta(V)$  in the worst case.

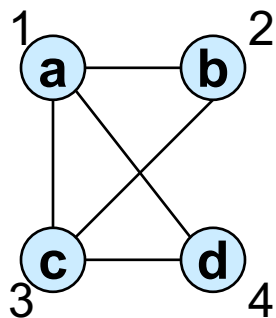
# Graphs – Representation – Adjacency Matrix

- $|V| \times |V|$  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



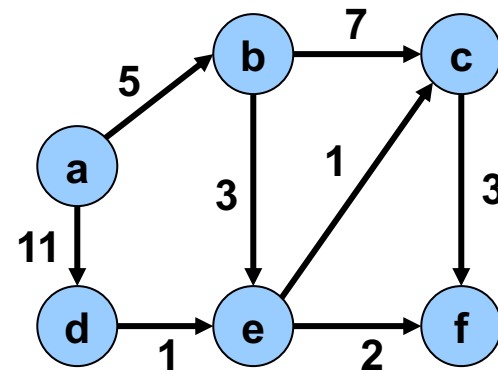
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

$A = A^T$  for undirected graphs.

# Adjacency Matrix – Storage-Time Requirement

- **Space:**  $\Theta(V^2)$ .
  - Not memory efficient for large sparse 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.

	a	b	c	d	e	f
a	0	5	0	11	0	0
b	0	0	7	0	3	0
c	0	0	0	0	0	3
d	0	0	0	0	1	0
e	0	0	1	0	0	2
f	0	0	0	0	0	0



# Graphs— Traversal-Searching

- Is there a path from  $s$  to  $v$  in  $G$ ?
- 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.
  - ① • Breadth-first Search (BFS).
  - ② • Depth-first Search (DFS).

# Graphs—Breadth-First Search

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - The algorithm discovers all vertices at distance  $k$  from  $s$  before discovering any vertices at distance  $k + 1$ .
  - 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.
    - It uses a first-in, first-out queue  $Q$  to manage the progress.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Grey – Discovered but not finished.
  - Black – Finished.

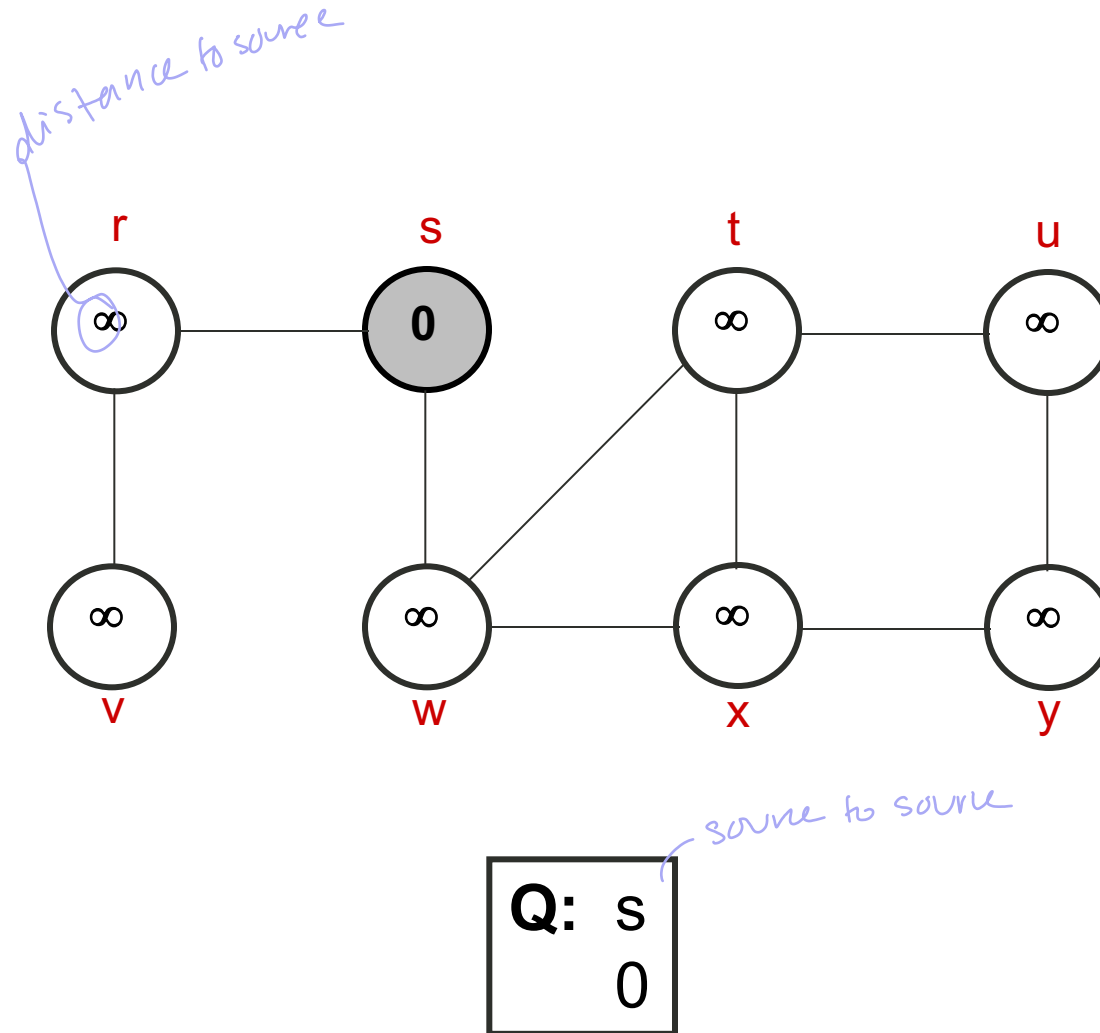
} 3 stages

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

# Graphs— Breadth-First Search

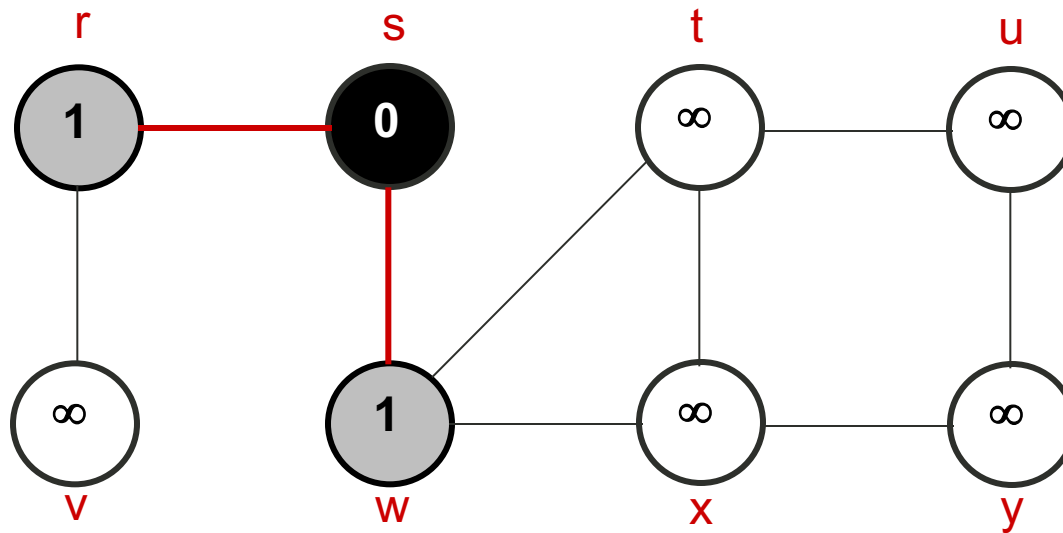
- **Input:** Graph  $G = (V, E)$ , either directed or undirected, and **source vertex**  $s \in V$ .
- **Output:** *array of distance between source node and each node*
  - $d[v]$  = distance (smallest # of edges, or shortest path) from  $s$  to  $v$ , for all  $v \in V$ .  $d[v] = \infty$  if  $v$  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. *gives order for each node*
  - Builds breadth-first tree with root  $s$  that contains all reachable vertices.

# Breadth-First Search - Example





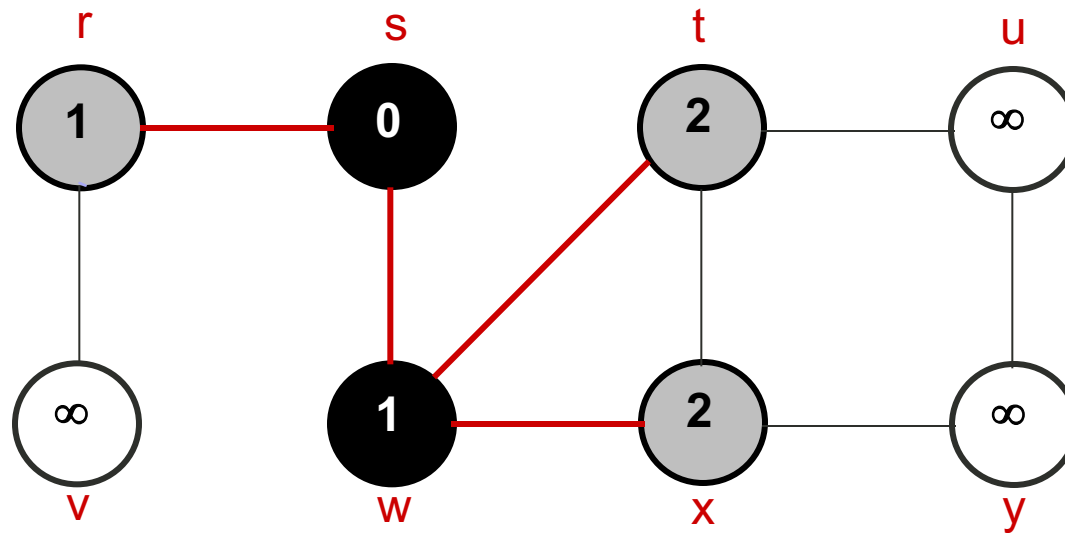
# Breadth-First Search - Example



Q:	w	r
	1	1

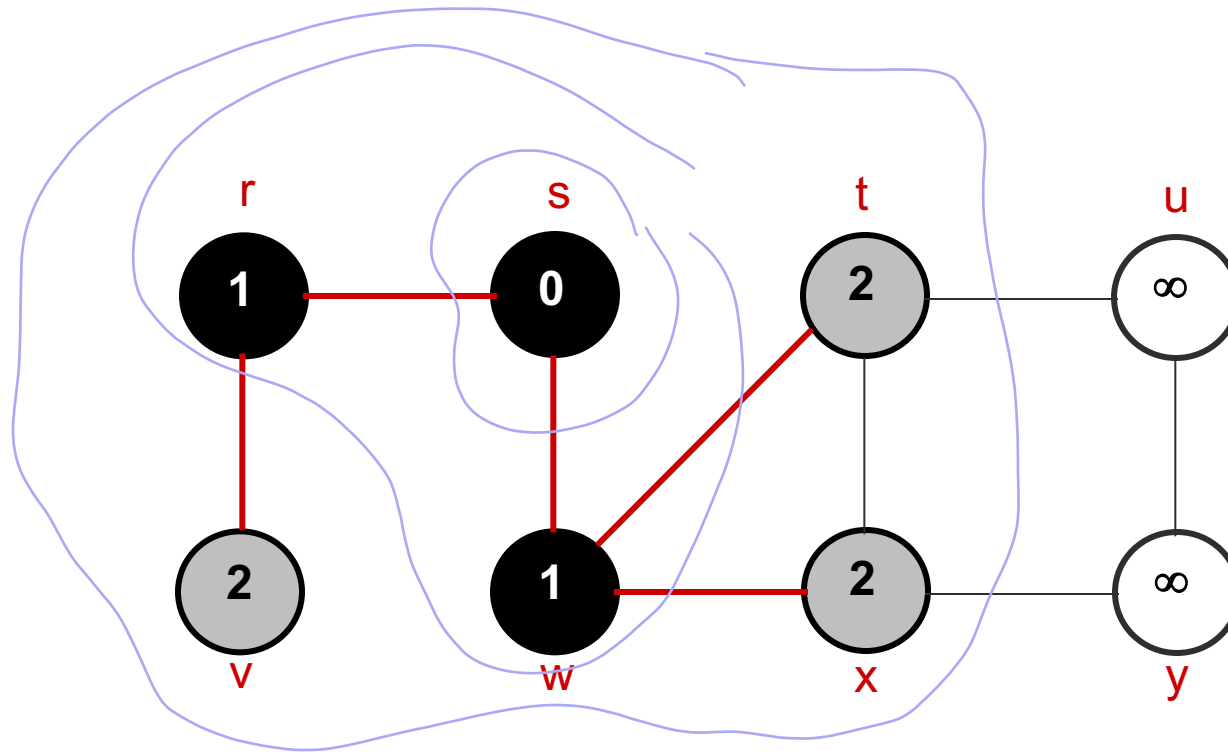
*dist 1 from s*

# Breadth-First Search - Example



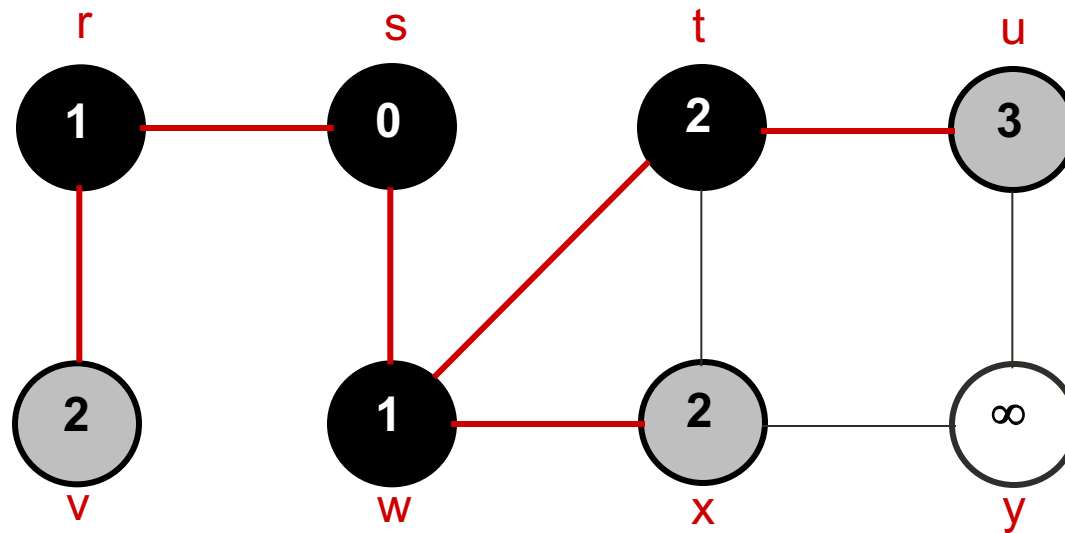
Q: r t x  
1 2 2

# Breadth-First Search - Example



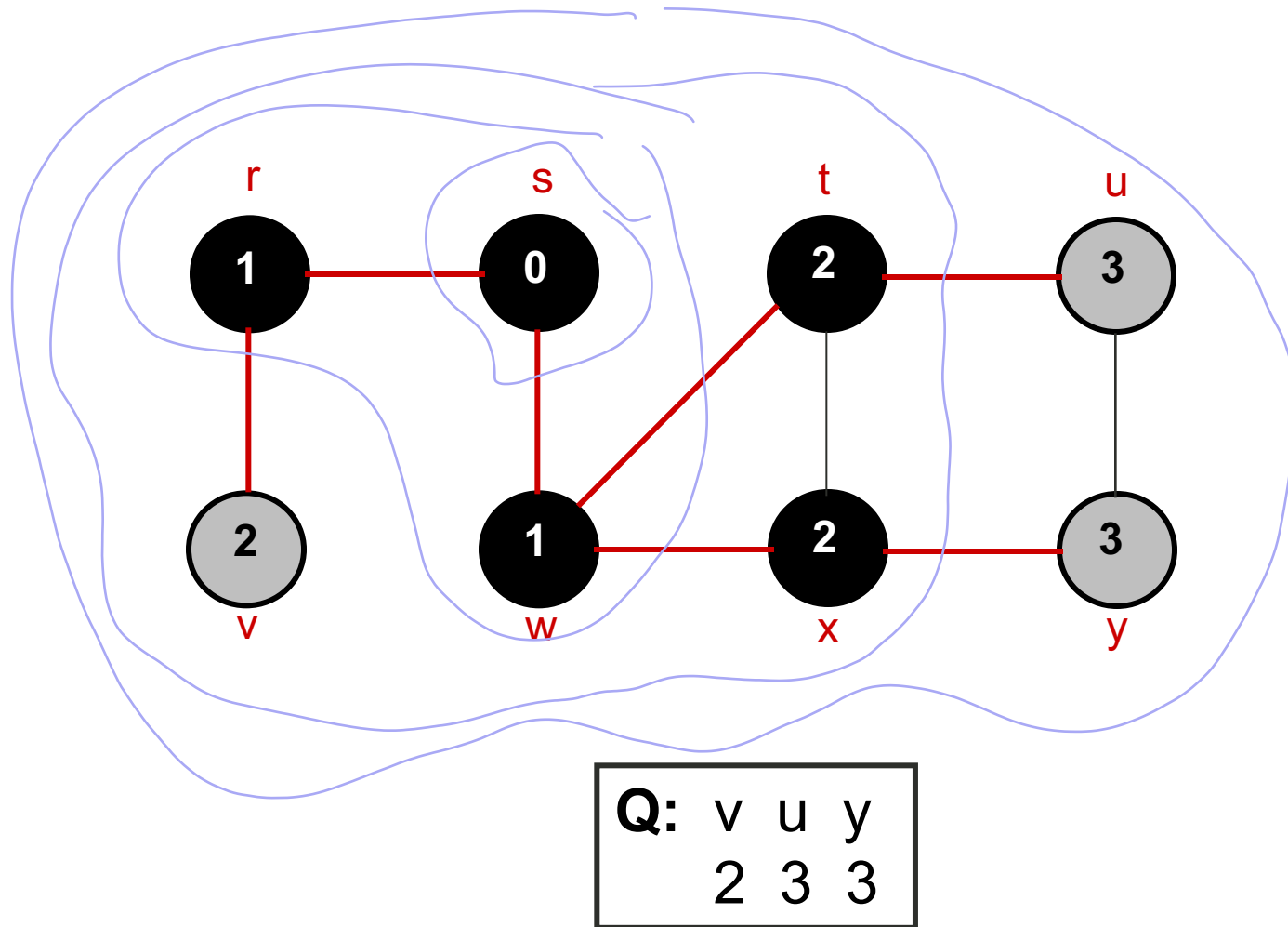
Q:	t	x	v
	2	2	2

# Breadth-First Search - Example

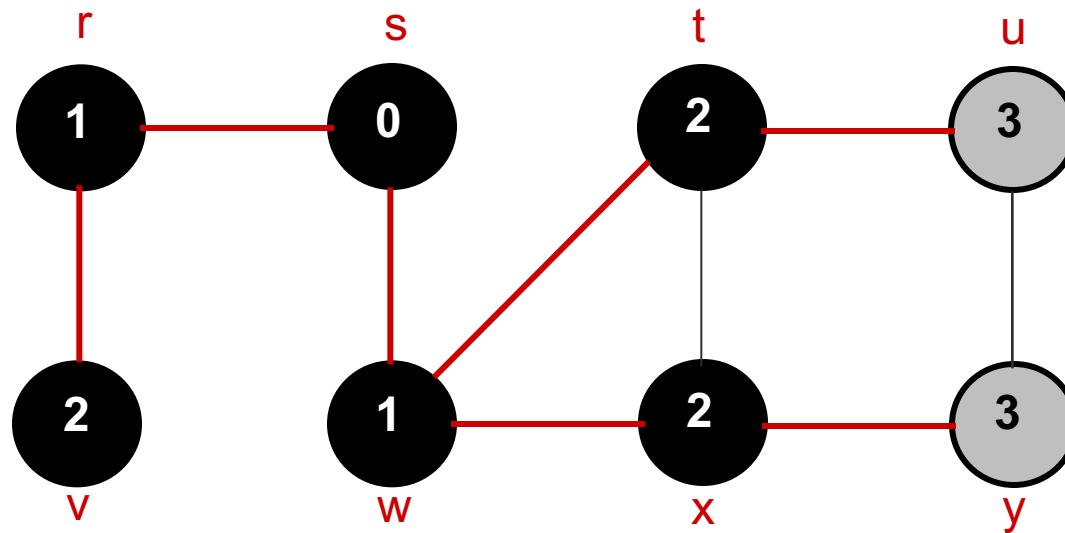


Q:	x	v	u
	2	2	3

# Breadth-First Search - Example

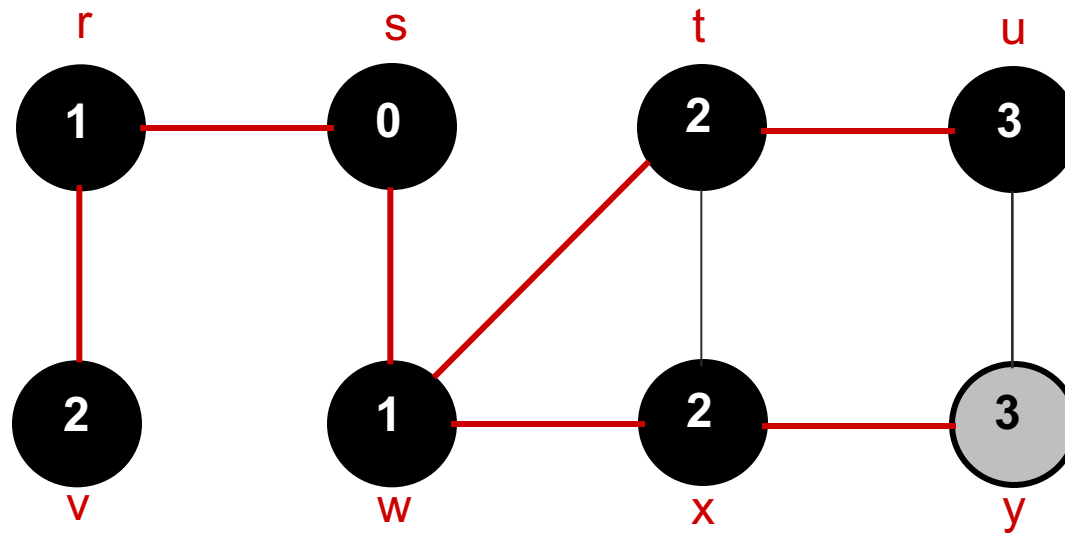


# Breadth-First Search - Example



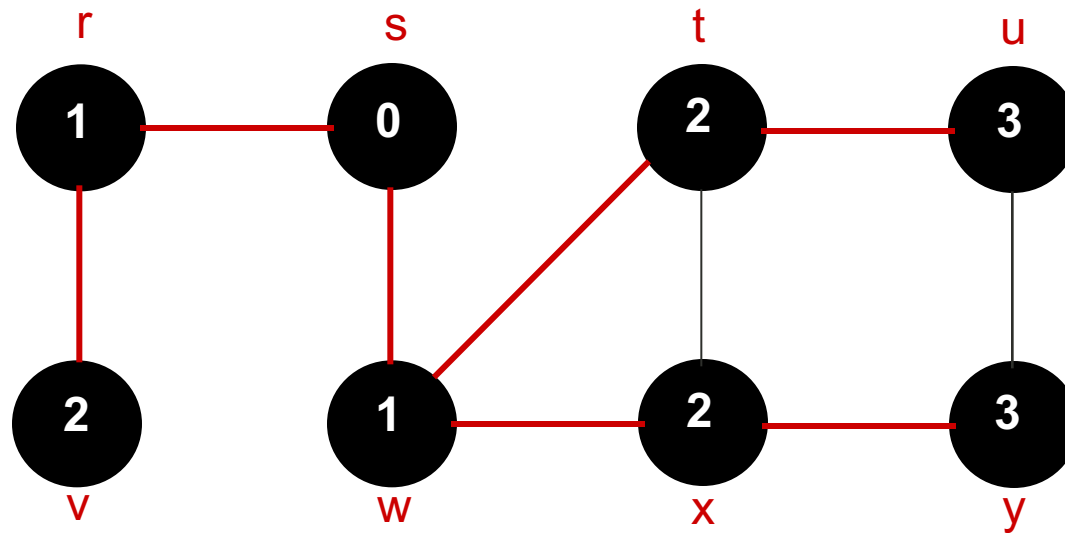
Q:	u	y
	3	3

# Breadth-First Search - Example



Q: y  
3

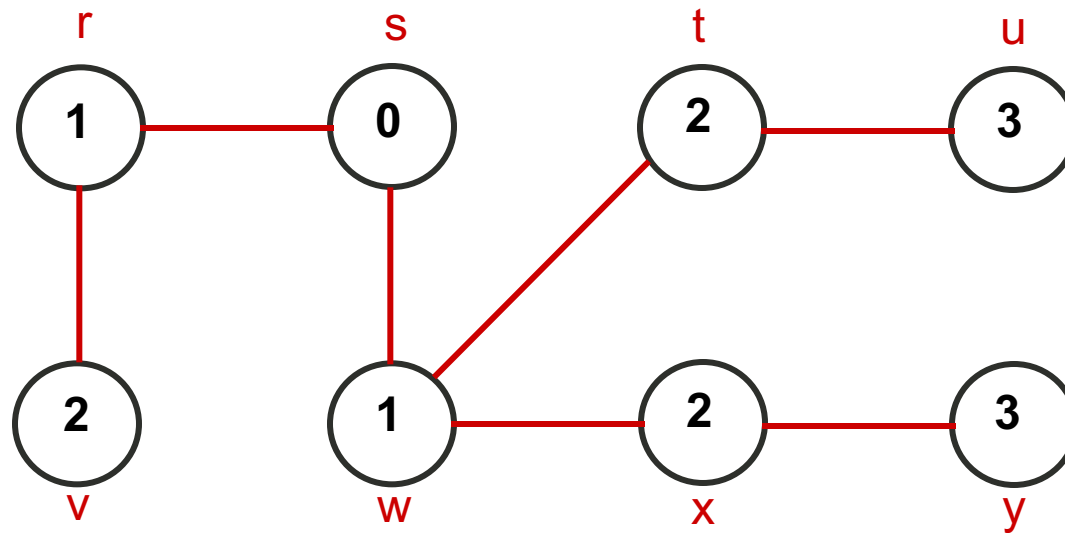
# Breadth-First Search - Example



Q:  $\emptyset$



# Breadth-First Search - Example



**BF Tree**

# Breadth-First Search - Analysis

BFS( $G, s$ )

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```

Initialization  $O(V)$

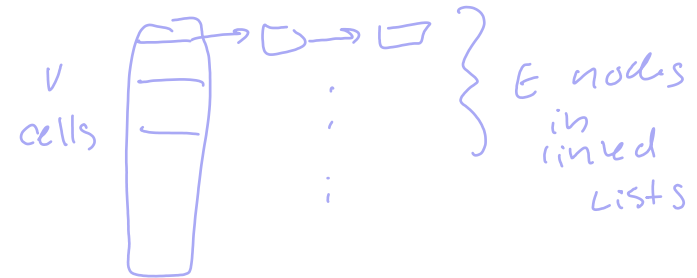
Traversal Loop:

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  $\Theta(E)$ .

# Breadth-First Search - Analysis

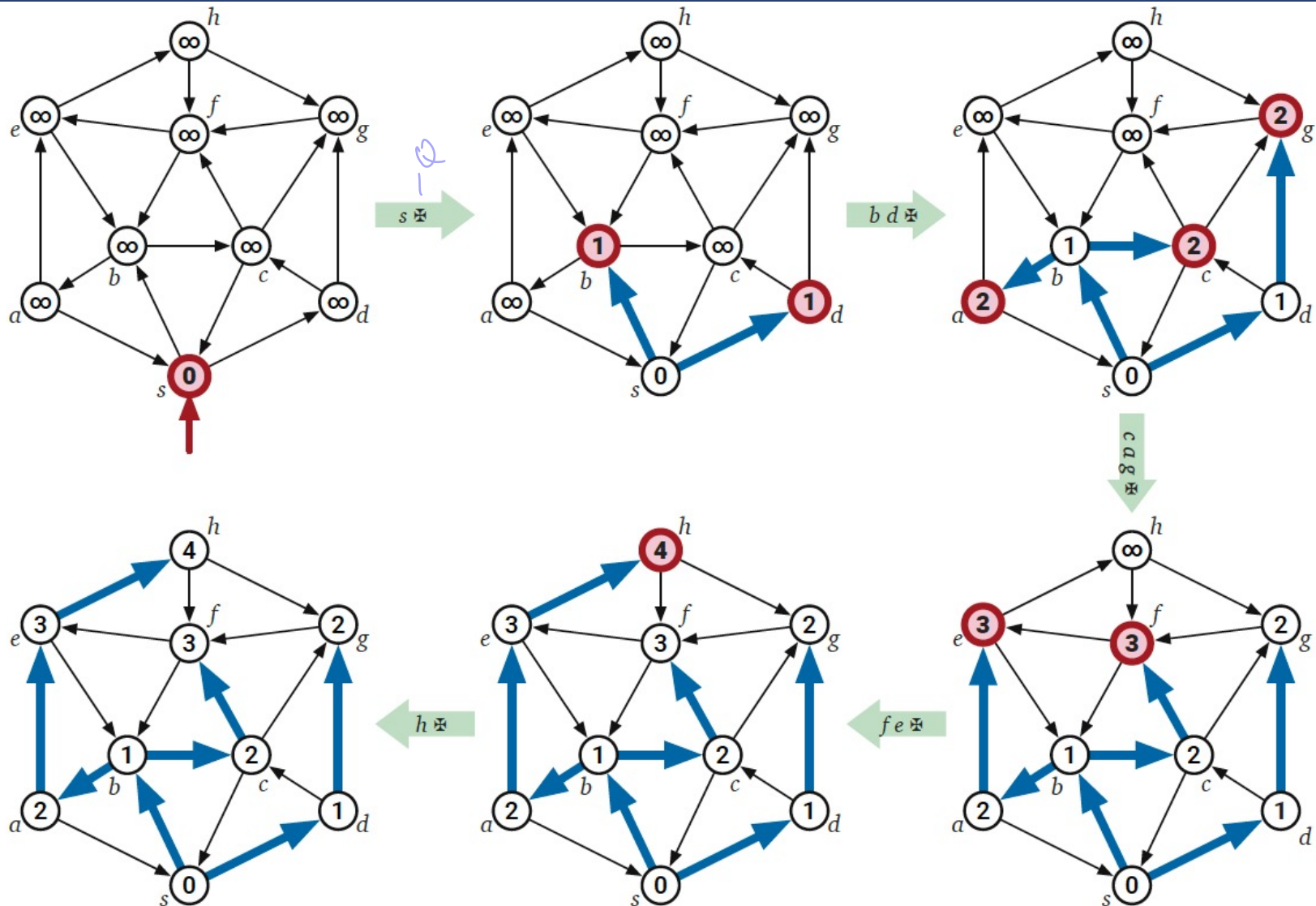
- Initialization takes  $O(V)$ .
- Traversal Loop
  - After initialization, each vertex is enqueued and dequeued at most once, and each operation takes  $O(1)$ . So, total time for queuing is  $O(V)$ .
  - The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency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



# Breadth-First Search - Application

- Suppose we are given an unweighted directed graph  $G = (V, E)$  with two special vertices, and we want to find the shortest path from a source vertex  $s$  to a target vertex  $t$ .
  - Special case of shortest path.
    - All edges have weight 1, and the length of a path is just the number of edges.

# Breadth-First Search - Application



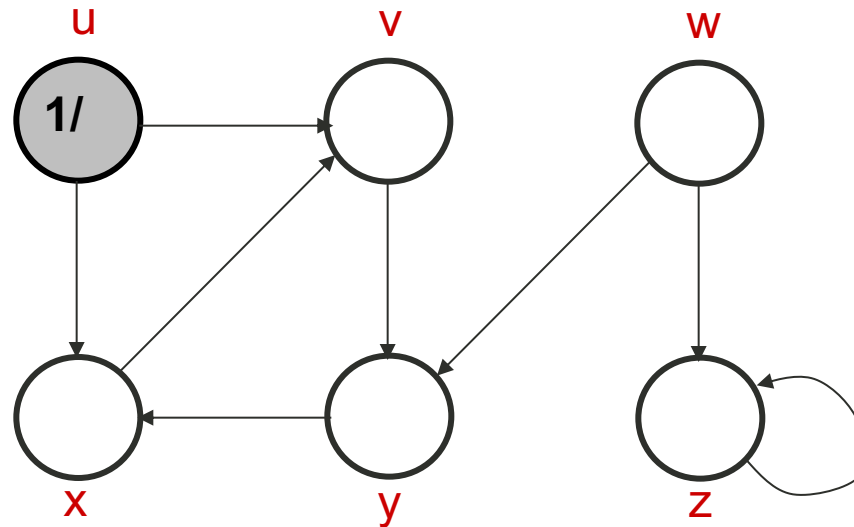
# Depth-First Search

- 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

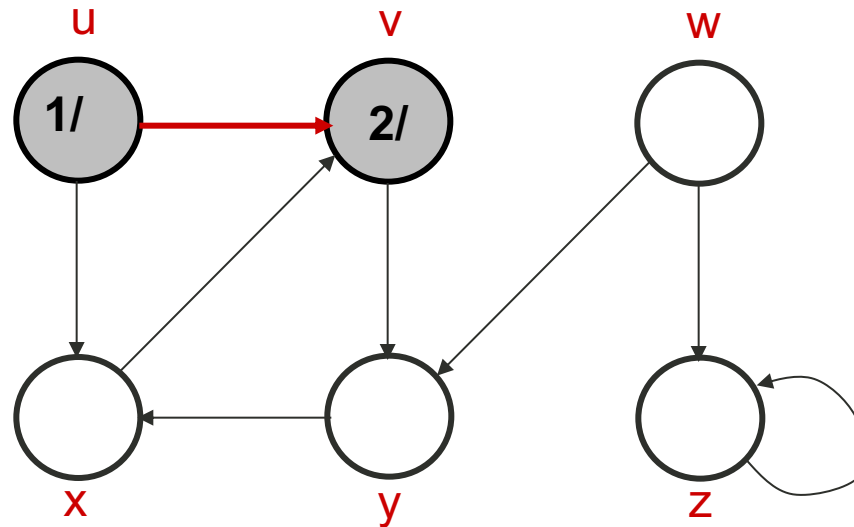
- **Input:**  $G = (V, E)$ , directed or undirected. No source vertex given.
- **Output:**
  - 2 **timestamps** on each vertex. Integers between 1 and  $2|V|$ .
    - $d[v] =$ **discovery time** ( $v$  turns from white to gray)
    - $f[v] =$ **finishing time** ( $v$  turns from gray to black)
  - $\pi[v]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

# Depth-First Search - Example

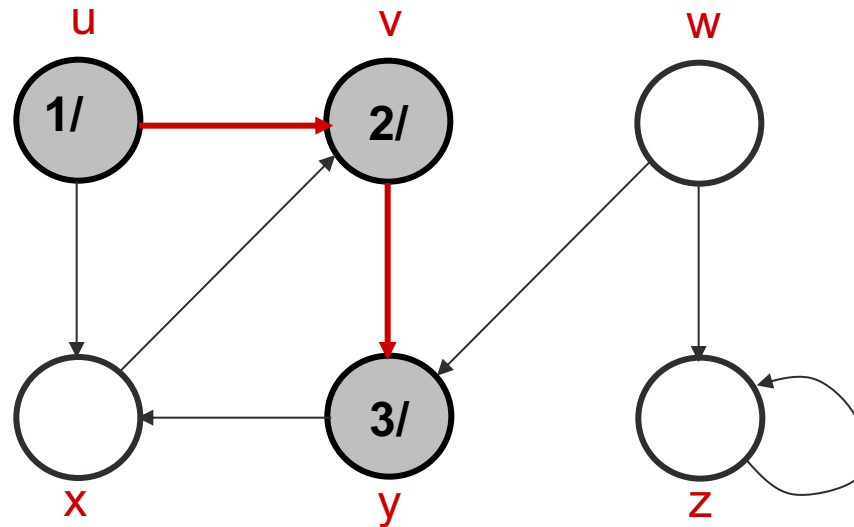




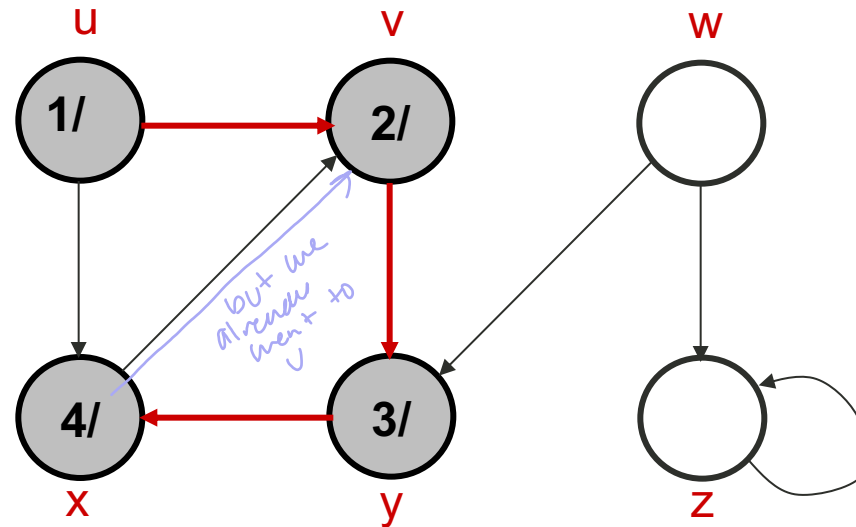
# Depth-First Search - Example



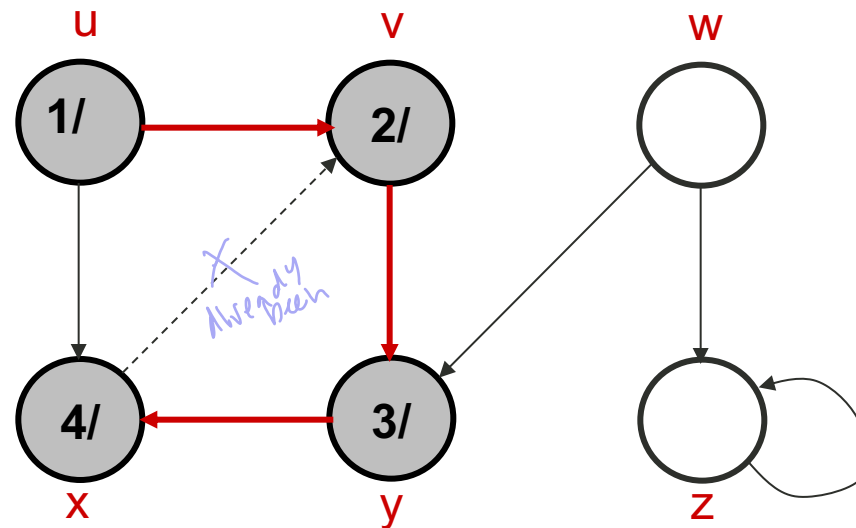
# Depth-First Search - Example



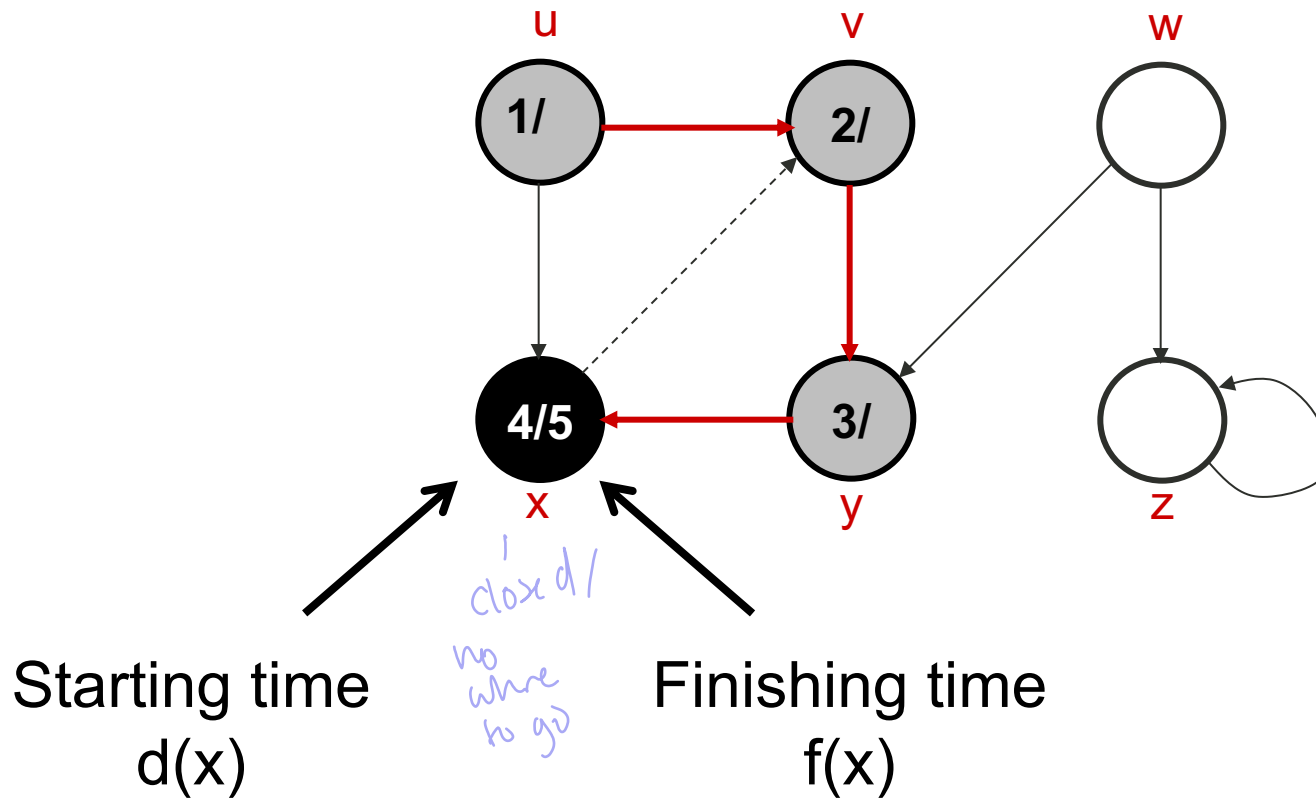
# Depth-First Search - Example



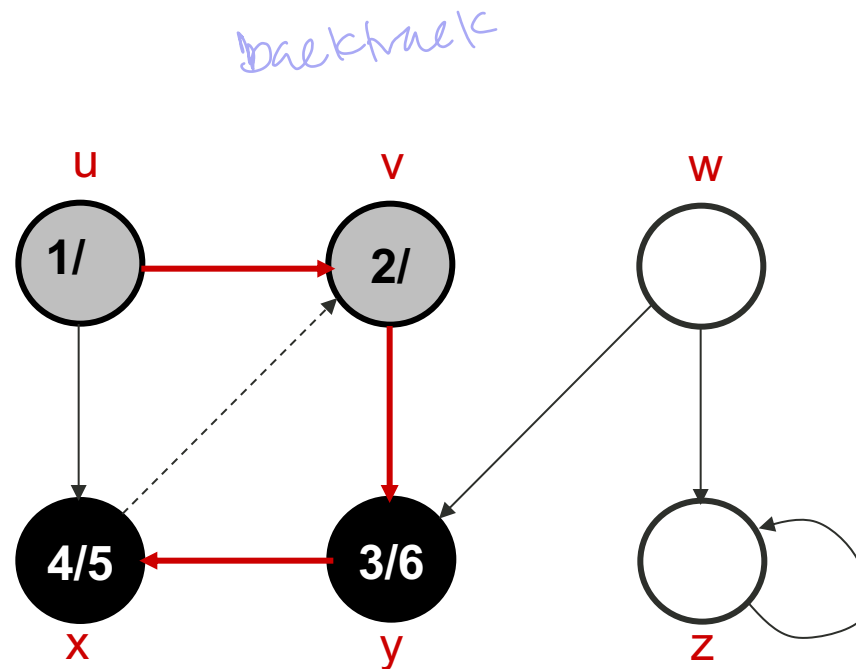
# Depth-First Search - Example



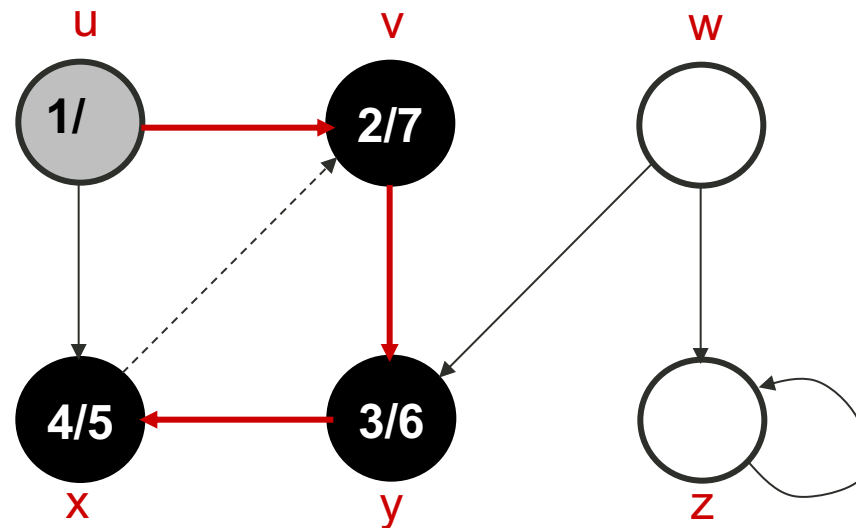
# Depth-First Search - Example



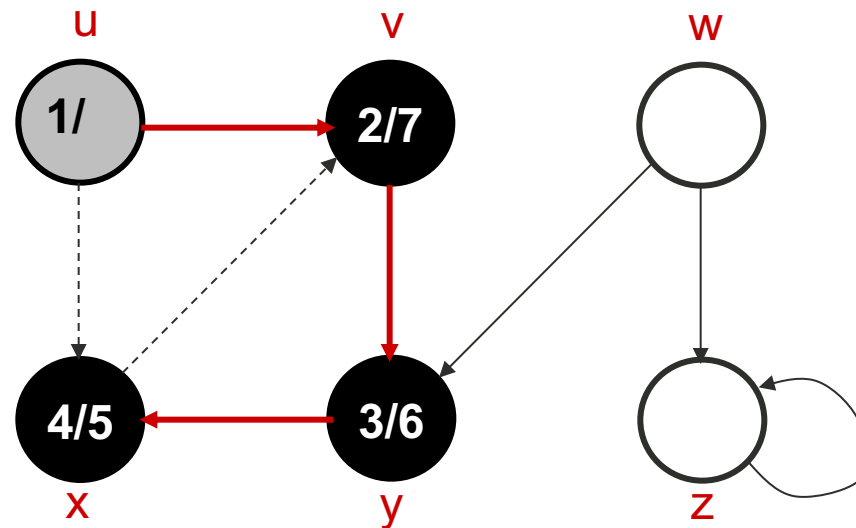
# Depth-First Search - Example



# Depth-First Search - Example

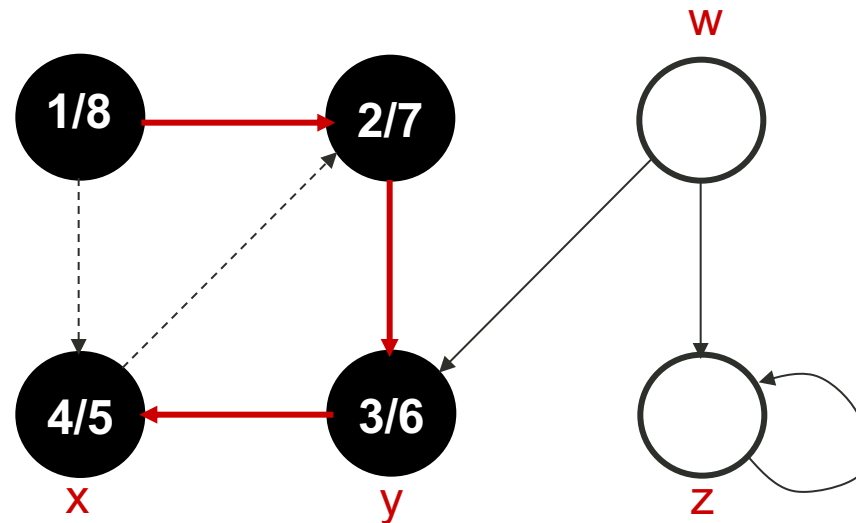


# Depth-First Search - Example

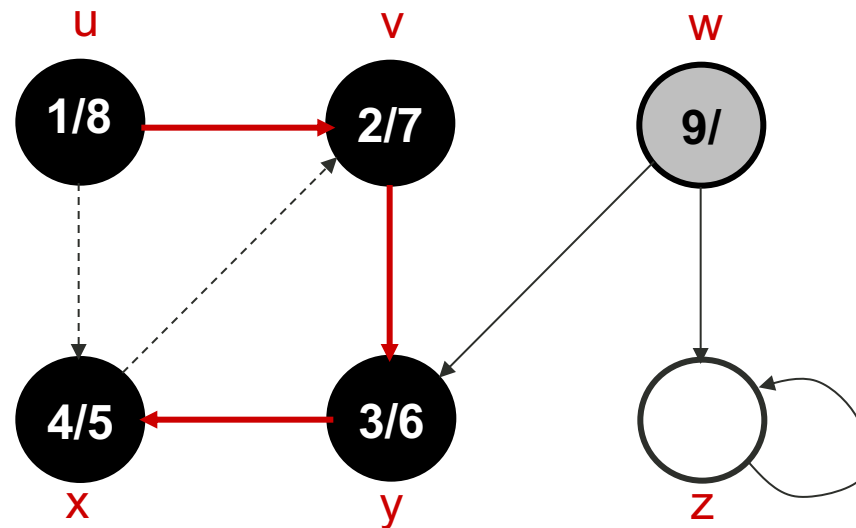




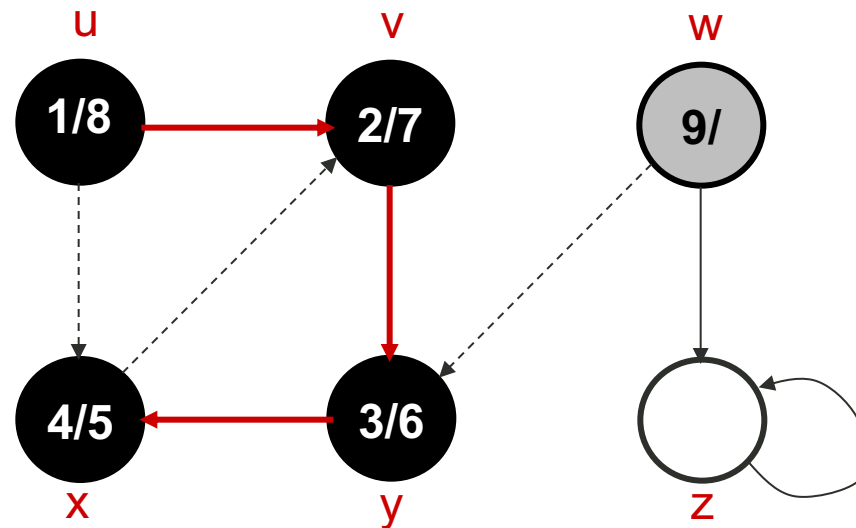
# Depth-First Search - Example



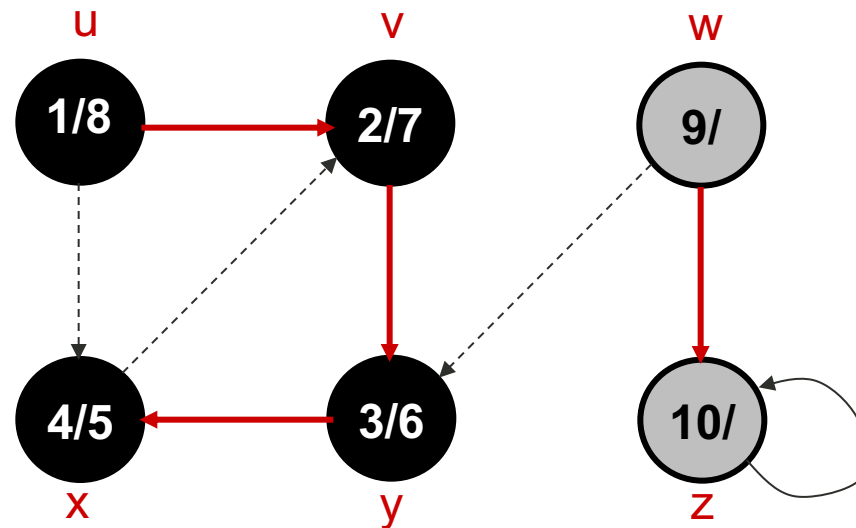
# Depth-First Search - Example



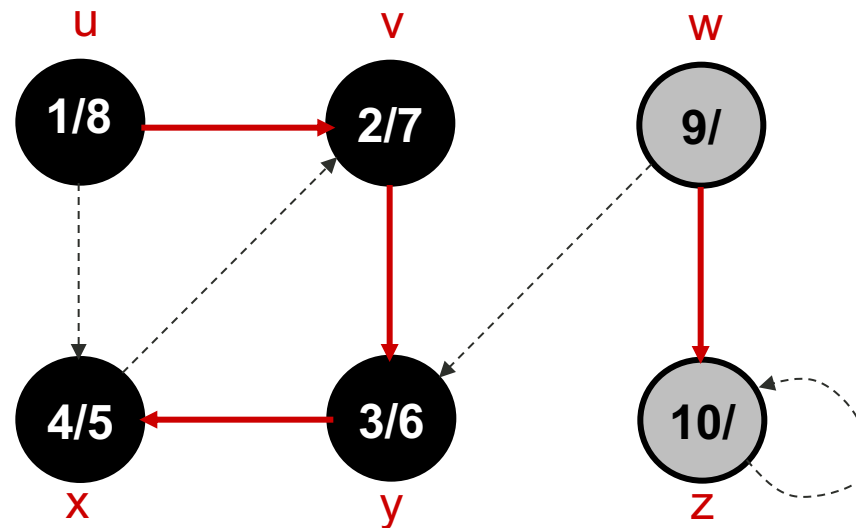
# Depth-First Search - Example



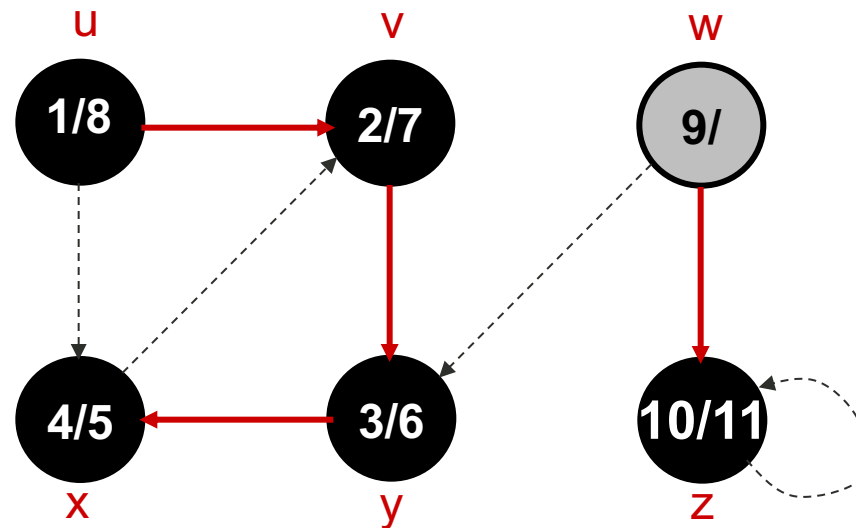
# Depth-First Search - Example



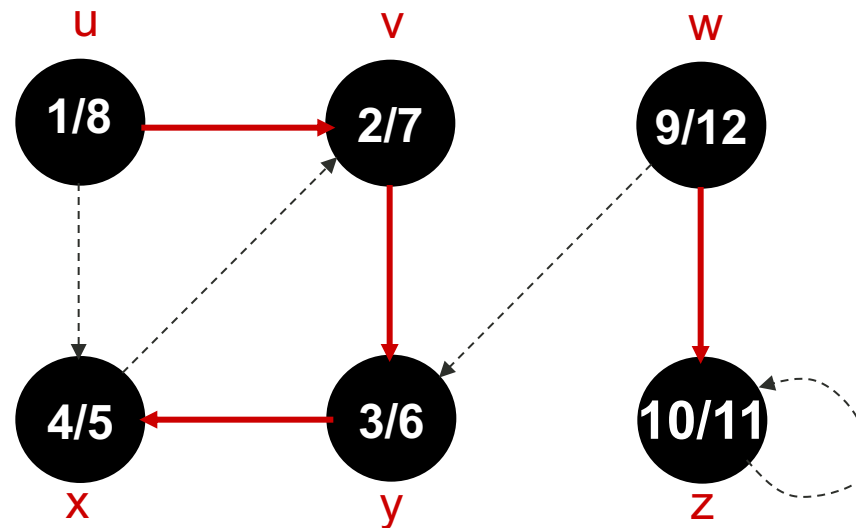
# Depth-First Search - Example



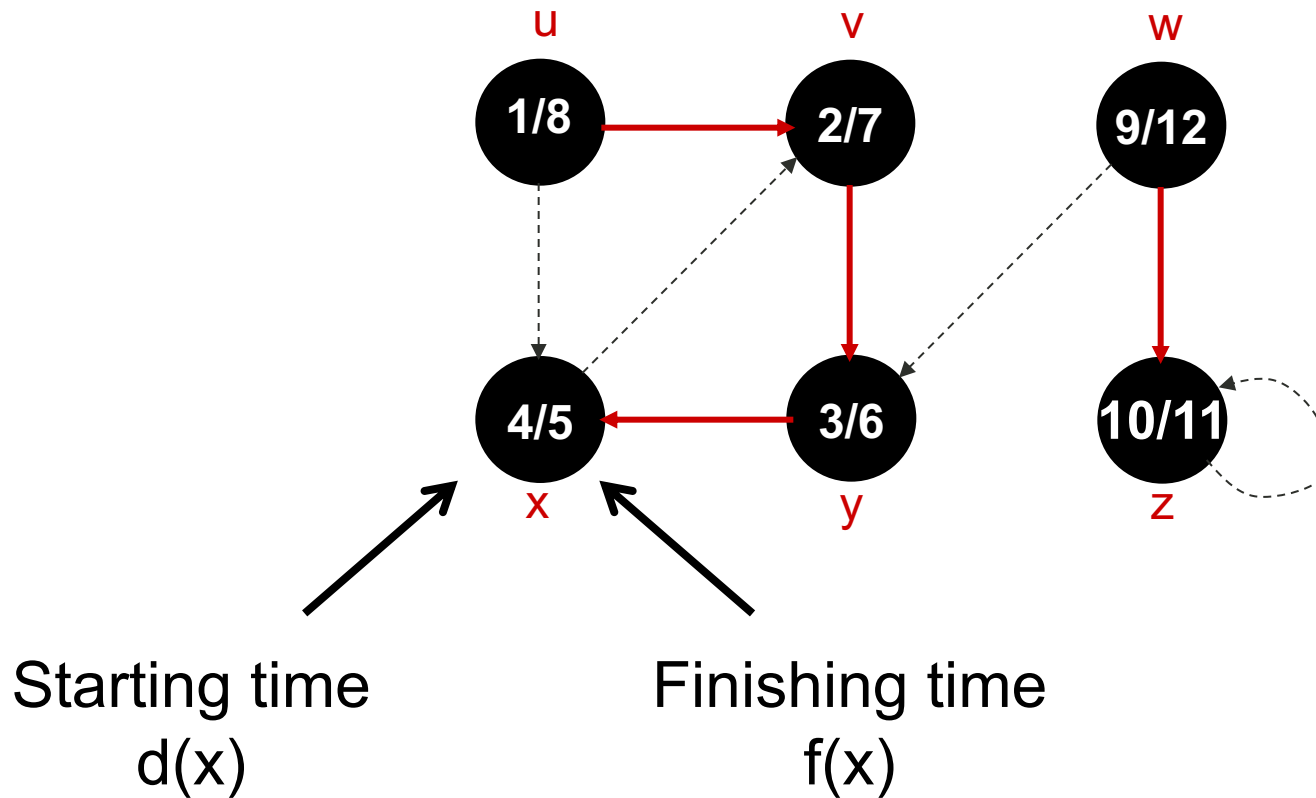
# Depth-First Search - Example



# Depth-First Search - Example



# Depth-First Search - Example





# Depth-First Search

## DFS(G)

1. **for** each vertex  $u \in V[G]$
2.     **do**  $color[u] \leftarrow \text{white}$
3.      $\pi[u] \leftarrow \text{NIL}$
4.  $time \leftarrow 0$
5. **for** each vertex  $u \in V[G]$
6.     **do if**  $color[u] = \text{white}$
7.         **then** DFS-Visit( $u$ )

Uses a global timestamp **time**.

## DFS-Visit( $u$ )

1.  $color[u] \leftarrow \text{GRAY}$  # White vertex  $u$  has been discovered
2.  $time \leftarrow time + 1$
3.  $d[u] \leftarrow time$
4. **for** each  $v \in Adj[u]$
5.     **do if**  $color[v] = \text{WHITE}$
6.         **then**  $\pi[v] \leftarrow u$
7.         DFS-Visit( $v$ )
8.  $color[u] \leftarrow \text{BLACK}$  # Blacken  $u$ ; it is finished.
9.  $f[u] \leftarrow time \leftarrow time + 1$

# Depth-First Search - Analysis

- 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 4-7 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 Search - Analysis

*use stack*

RECURSIVEDFS( $v$ ):

if  $v$  is unmarked

mark  $v$

for each edge  $vw$

RECURSIVEDFS( $w$ )

ITERATIVEDFS( $s$ ):

PUSH( $s$ )

while the stack is not empty

$v \leftarrow \text{POP}$

if  $v$  is unmarked

mark  $v$

for each edge  $vw$

PUSH( $w$ )

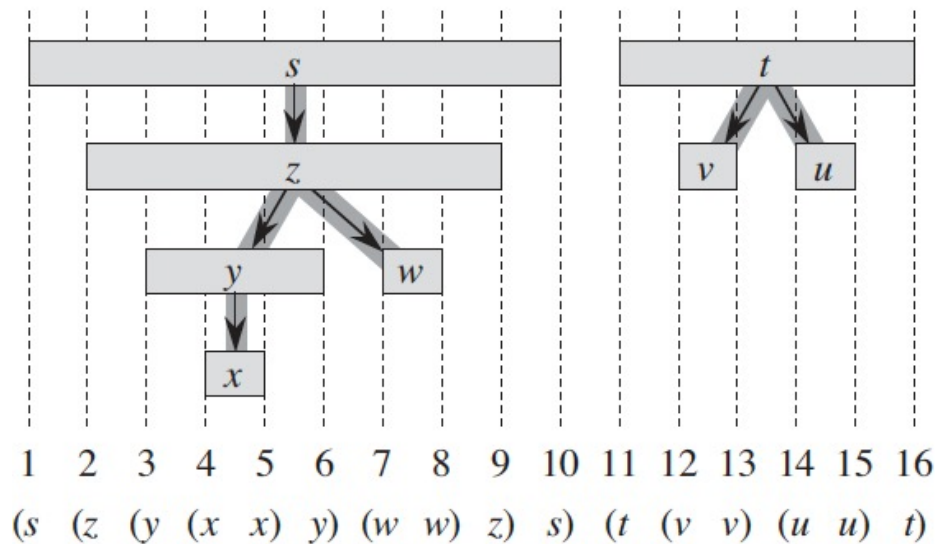
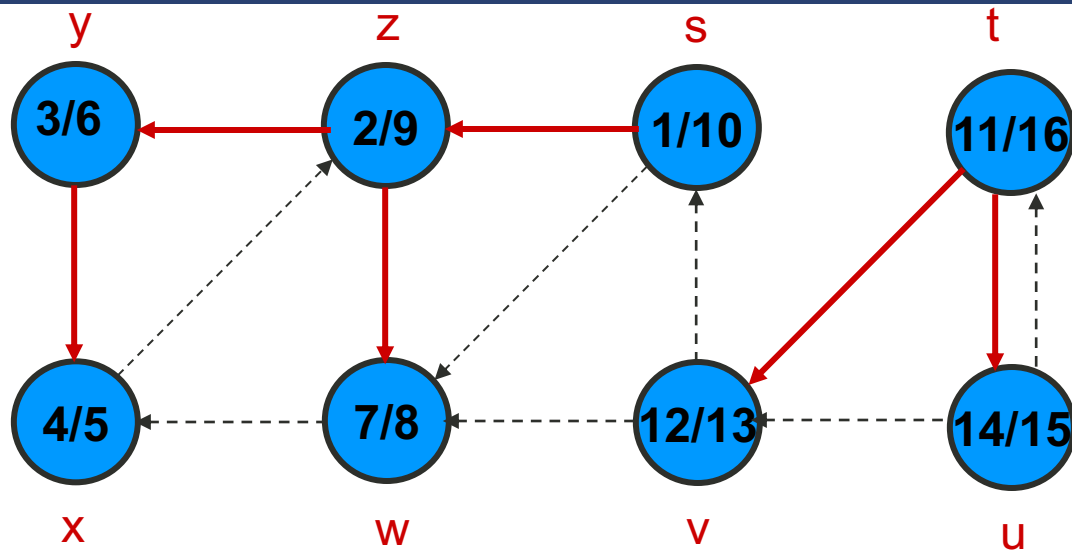
# Depth-First Search - Application

- Depth-first search is often a subroutine in another algorithm.
  - Depth-first search yields valuable information about the structure of a graph.
  - The most basic property of depth-first search is that the predecessor subgraph  $G$  does indeed form a forest of trees.
  - Another important property of depth-first search is that discovery and finishing times have parenthesis structure.
    - If we represent the discovery of vertex  $u$  with a left parenthesis “(” and represent its finishing by a right parenthesis “)”, then the history of discoveries and finishings makes a well-formed expression in the sense that the parentheses are properly nested.

OK: ( { } ) [ ]  
1 2 3 4 5 6

Not OK: ( { ) }  
1 2 3 4

# Parenthesis Property



*info about  
overlapping*

# Whatever First Search

BFS and DFS differ essentially only in that one uses a queue and the other uses a stack.

WHATEVERFIRSTSEARCH( $s$ ):

put  $s$  into the bag

while the bag is not empty

take  $v$  from the bag

if  $v$  is unmarked

mark  $v$

for each edge  $vw$

put  $w$  into the bag

Bag = Stack : Depth-First (Topological Sort)

Bag = Queue : Breadth-First (Shortest Path)

Bag = Priority Queue : Best-First (Minimum Spanning Tree)

# Outline

- Graphs.
  - Introduction.
  - Strong Connected Components / Topological Sort.
  - Network Flow 1.
  - Network Flow 2.
  - Shortest Path.
  - Minimum Spanning Trees.
  - Bipartite Graphs.

