COMP 251

Algorithms & Data Structures (Winter 2021)

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.

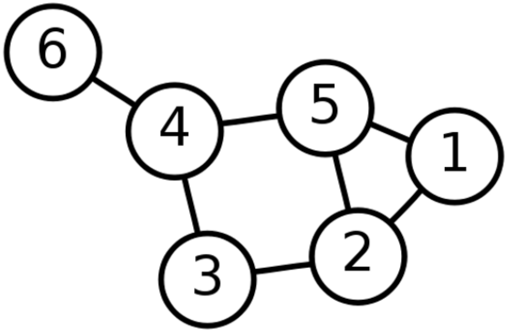
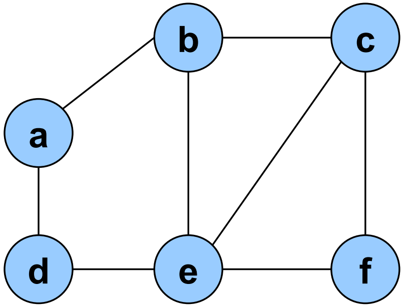
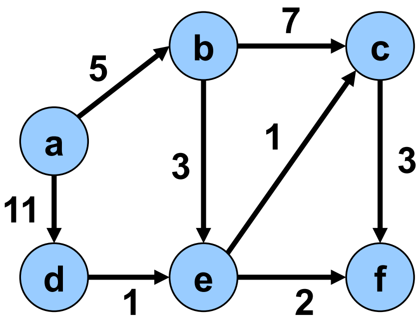


Figure from Wikipedia

Graphs - Definition



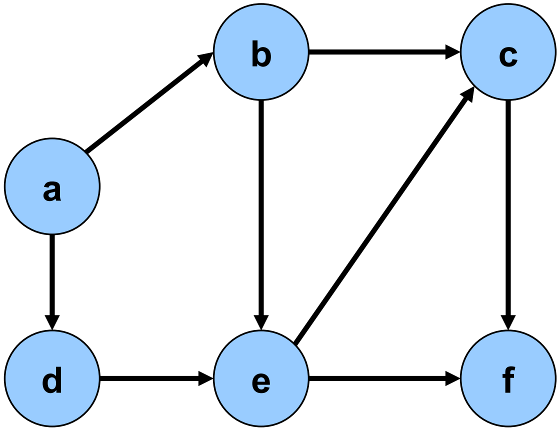
- *Graph* $G = (V, E)$
  - V = set of vertices
  - E = set of edges $\subseteq (V \times V)$
- Types of graphs
  - Undirected: edge $(u, v) = (v, u)$; for all v, $(v, v) \notin E$ (No self loops.)
  - Directed: (u, v) is edge from u to v, denoted as $u \rightarrow v$. Self loops are allowed.
  - Weighted: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



Graphs - Properties

- If $(u, v) \in E$, then vertex v is adjacent to vertex u.
- Adjacency relationship is:
  - Symmetric if G is undirected.
  - 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 $|E| = |V| - 1$, then G is a tree.

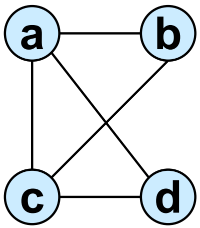
Graphs - Vocabulary

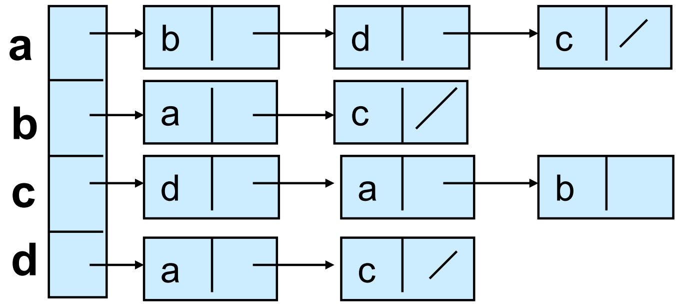


- Ingoing edges of u: { (v,u) ∈ E } (e.g. in(e) = { (b,e), (d,e) })
- Outgoing edges of u: { (u,v) ∈ E } (e.g. out(d) = { (d,e) })
- In-degree(u): | in(u) | // Number of predecessors
- Out-degree(u): | out(u) | //Number of successors

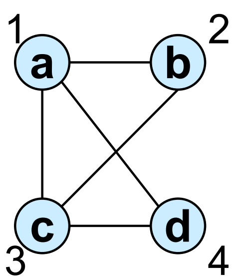
Graphs - Representation

- Two standard ways.
  - Adjacency Lists.





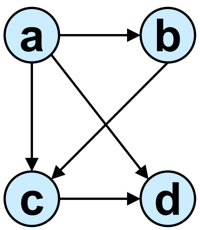
  - Adjacency Matrix.

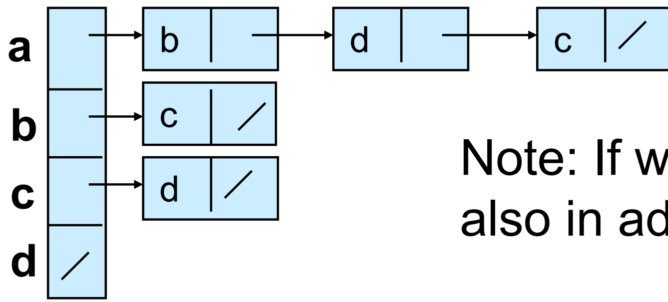


	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

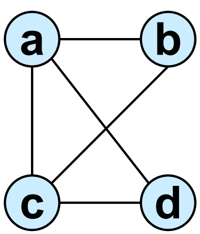
Graphs – Representation – Adjacency List

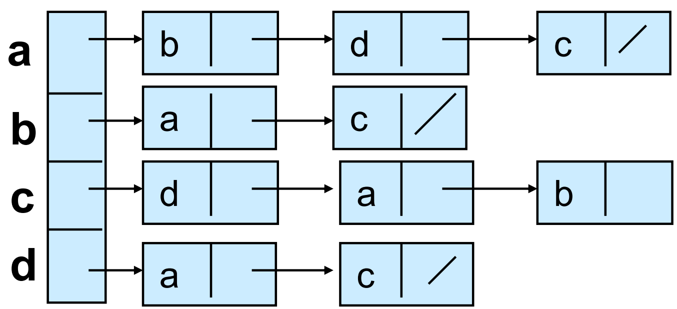
- Consists of an array *Adj* of $|V|$ lists.
- One list per vertex.
- For $u \in V$, *Adj*[u] consists of all vertices adjacent to u.





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: $\Theta(V+E)$

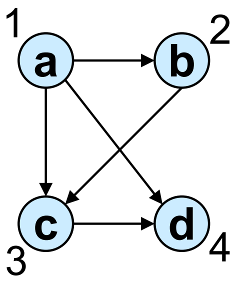
Adjacency List – Pros and Cons

- 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.

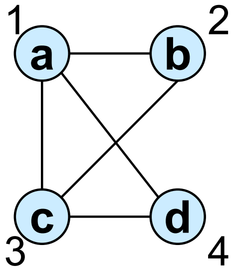
Graphs – Representation – Adjacency Matrix

- $|V| \times |V|$ 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



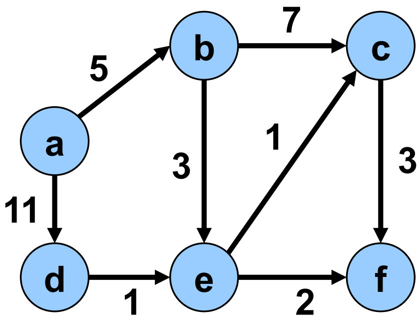
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

$A = A^{T}$ 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.
  - Breadth-first Search (BFS).
  - Depth-first Search (DFS).

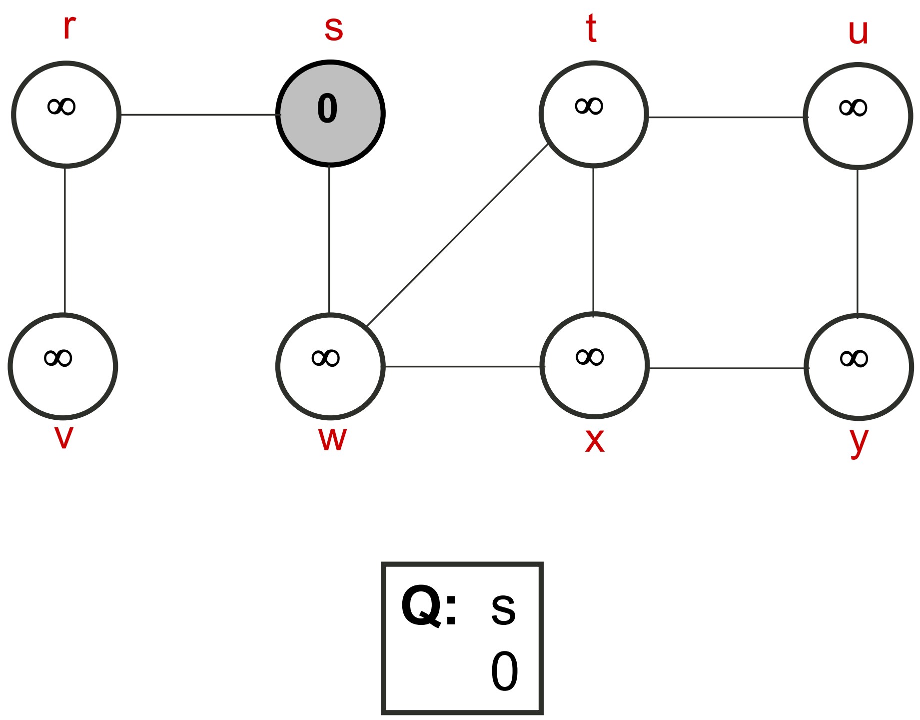
Graphs– Breadth-First Search

- 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.
  - Gray – Discovered but not finished.
  - Black – Finished.
    - Colors are required only to reason about the algorithm. Can be implemented without colors.

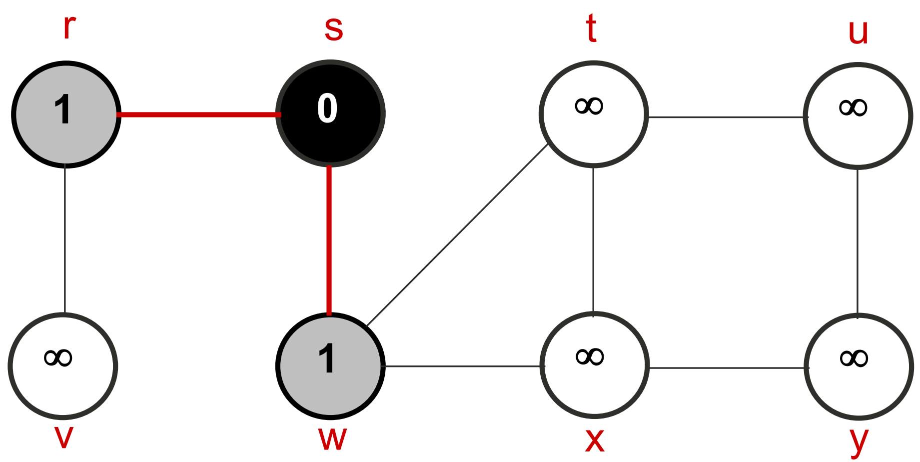
Graphs– Breadth-First Search

- **Input:** Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.
- **Output:**
  - $d[v]$ = 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 $s \leadsto v$.
    - u is v' s predecessor.
  - Builds breadth-first tree with root s that contains all reachable vertices.

Breadth-First Search - Example

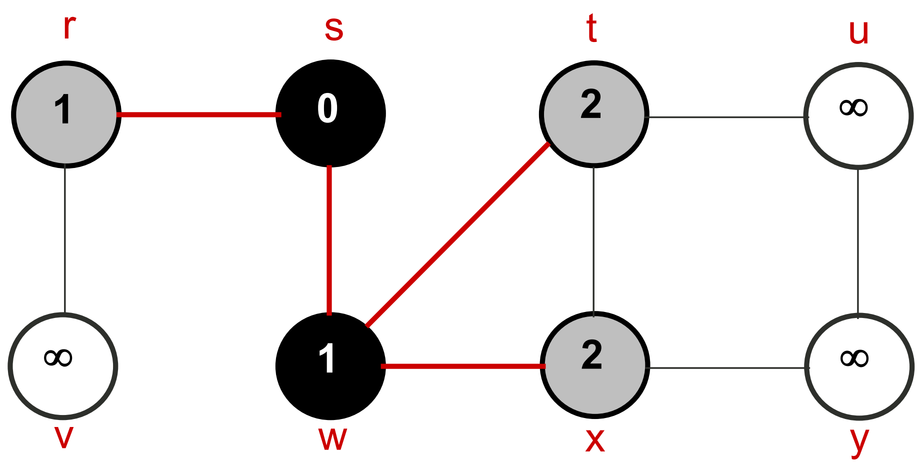


Breadth-First Search - Example



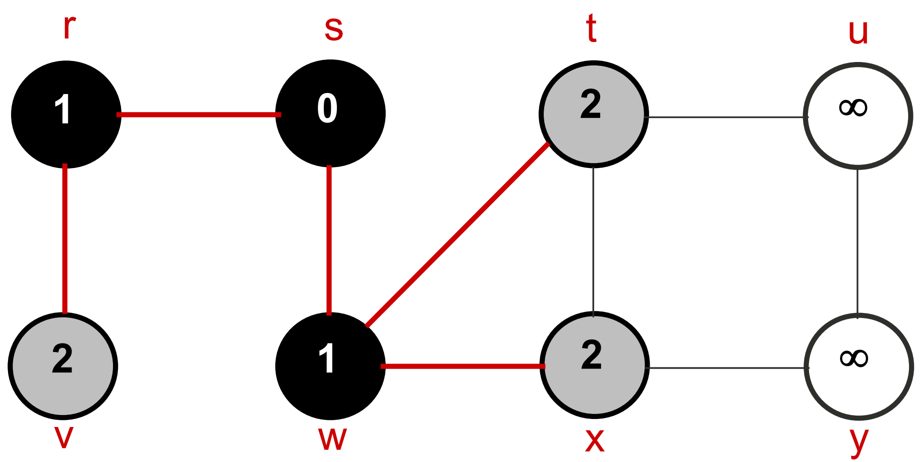
Q: w r
1 1

Breadth-First Search - Example



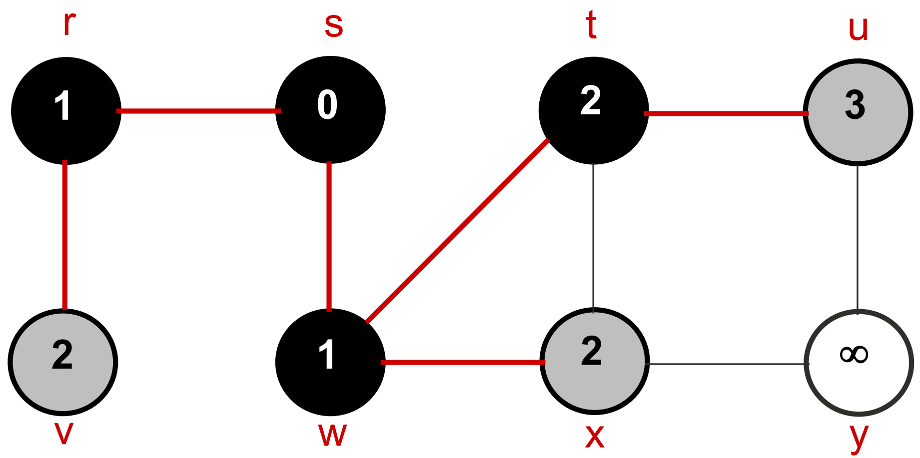
Q:	r	t	x
	1	2	2

Breadth-First Search - Example



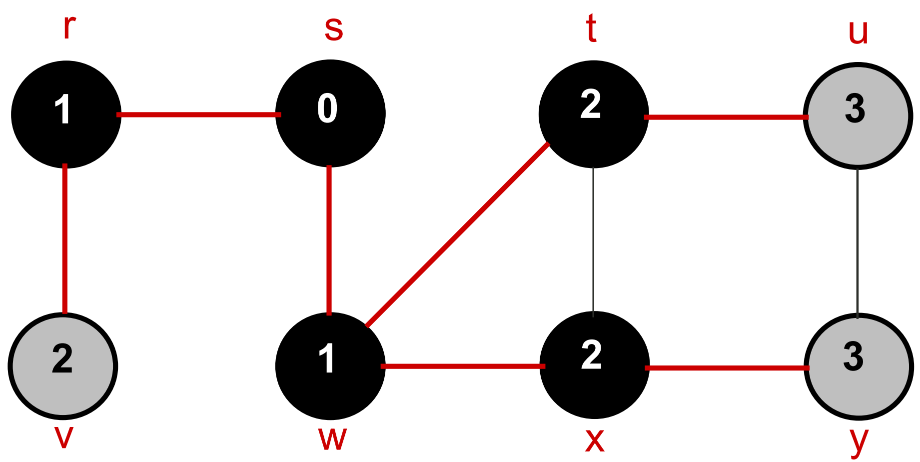
Q: t x v
2 2 2

Breadth-First Search - Example



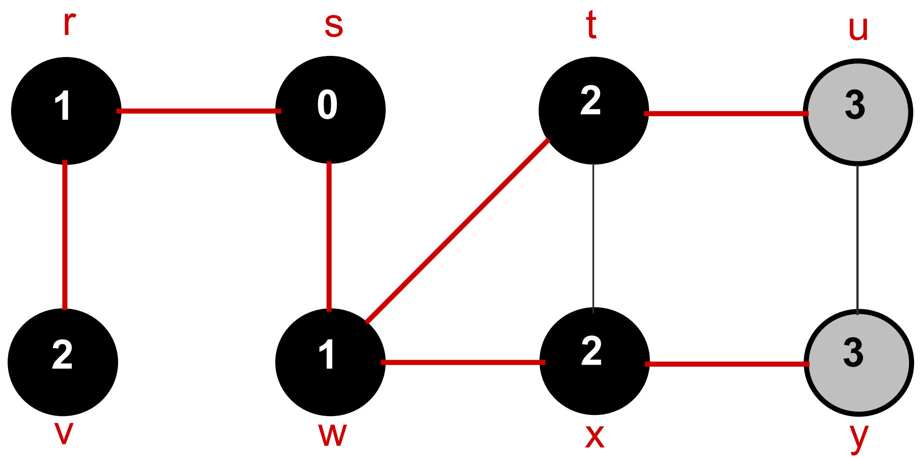
Q:	x	v	u
	2	2	3

Breadth-First Search - Example



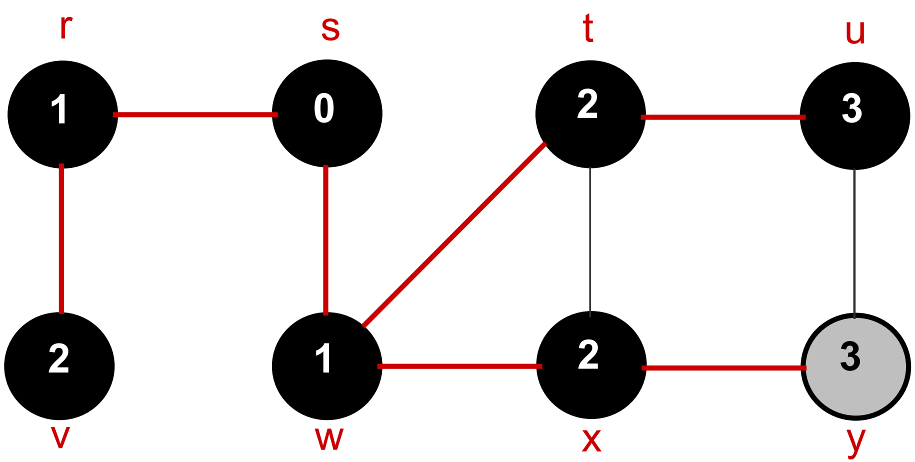
Q:	v	u	y
	2	3	3

Breadth-First Search - Example



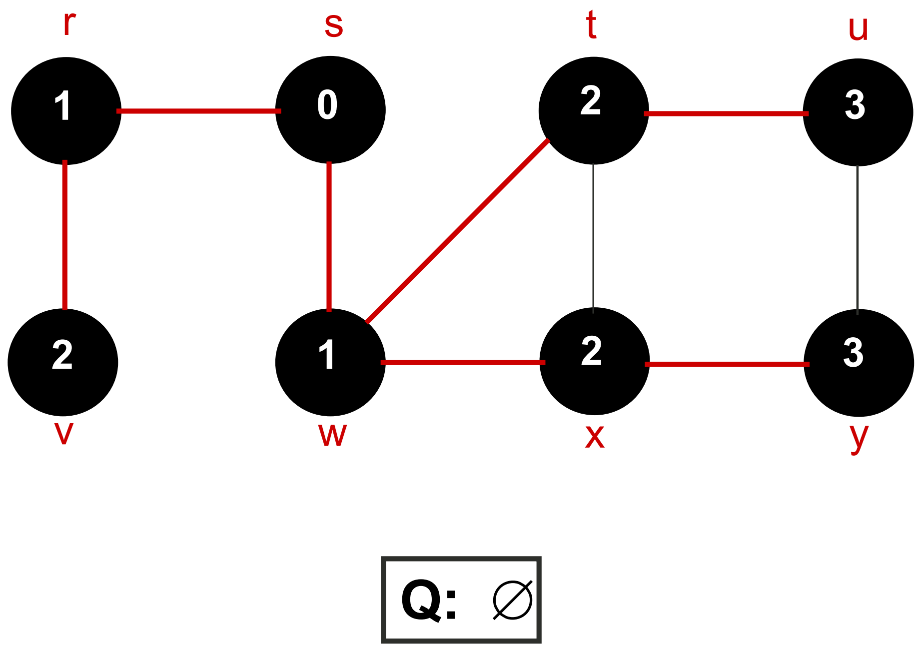
Q: u y
3 3

Breadth-First Search - Example

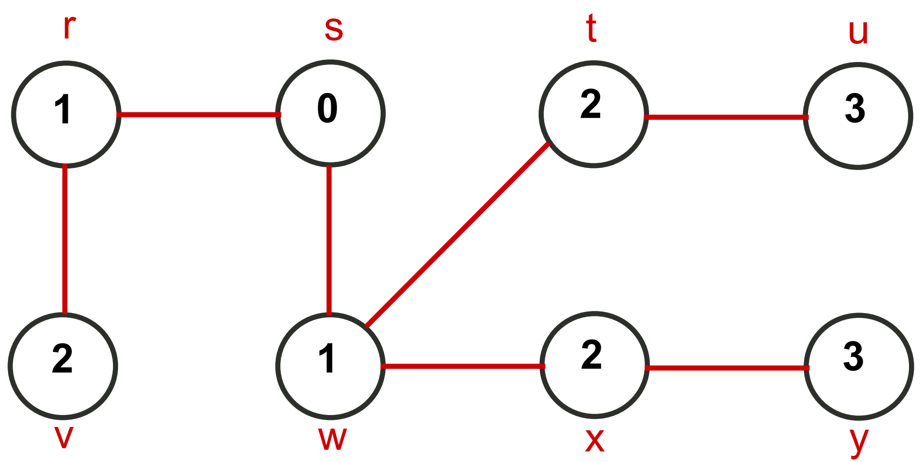


Q:	y
	3

Breadth-First Search - Example



Breadth-First Search - Example



BF Tree

Breadth-First Search - Analysis

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = 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 $\Theta(E)$.

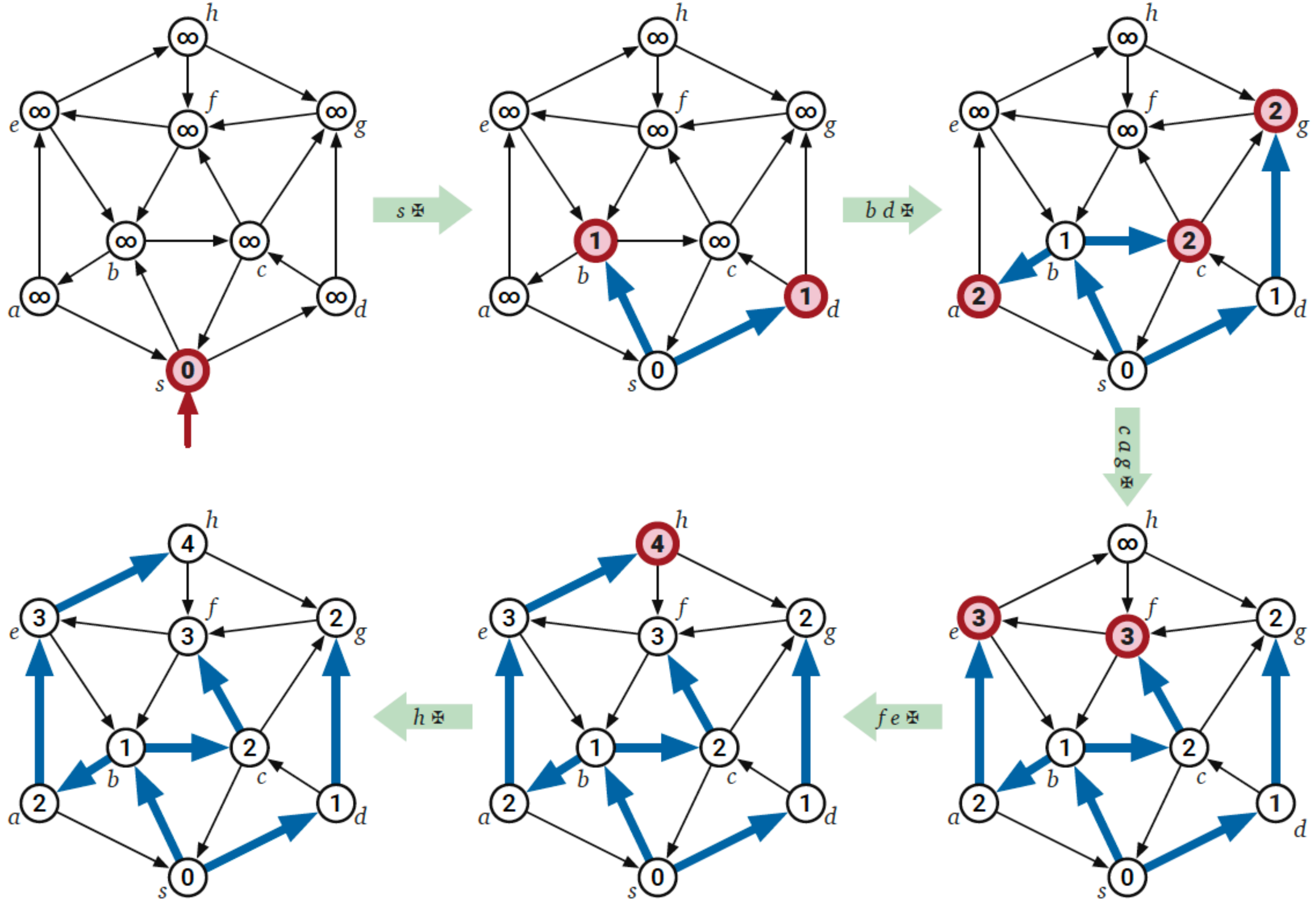
Breadth-First Search - Analysis

- 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 ⇒ 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



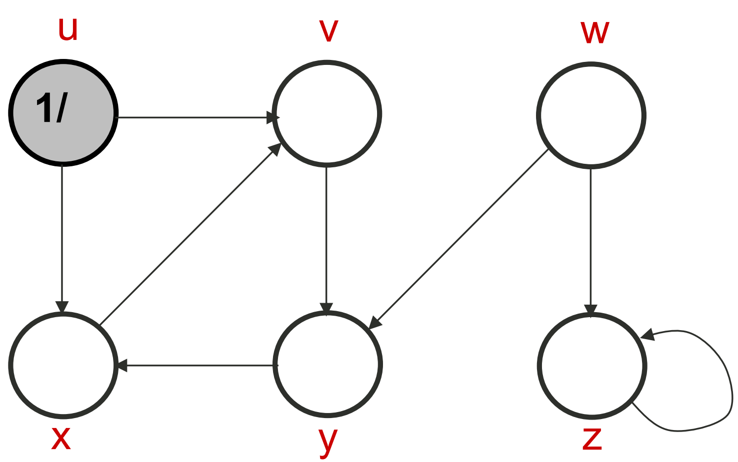
Depth-First Search

- 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.

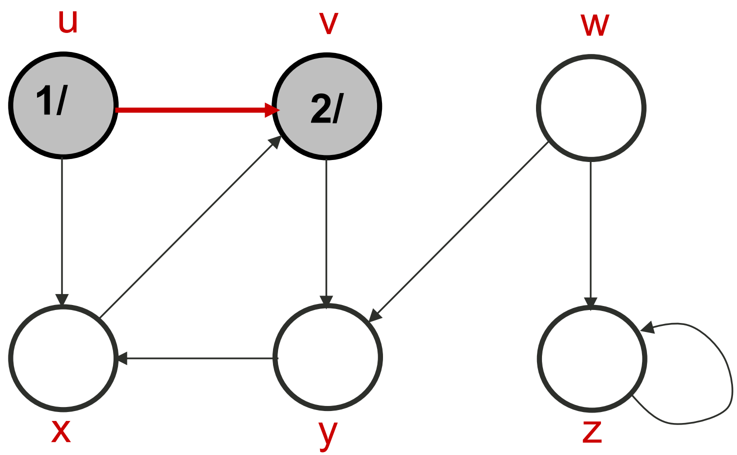
Depth-First Search

- **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.

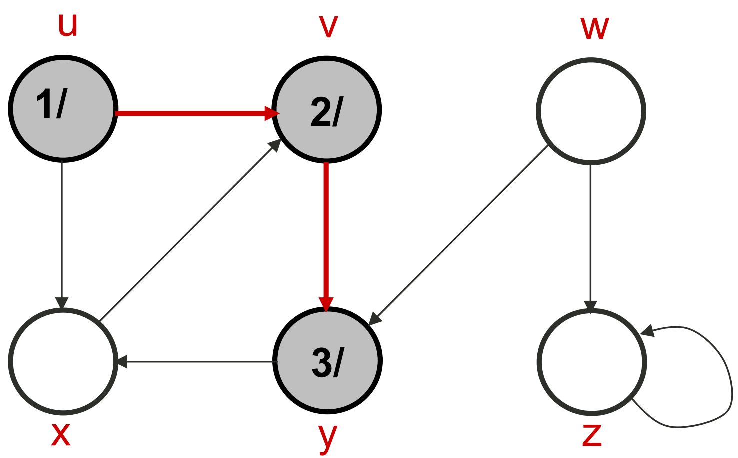
Depth-First Search - Example



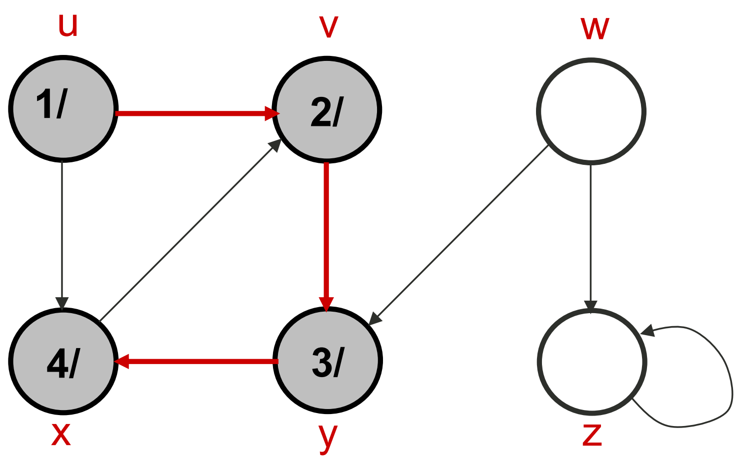
Depth-First Search - Example



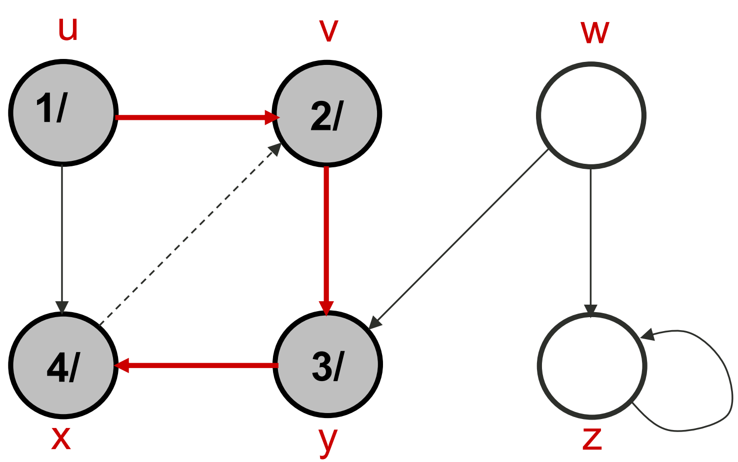
Depth-First Search - Example



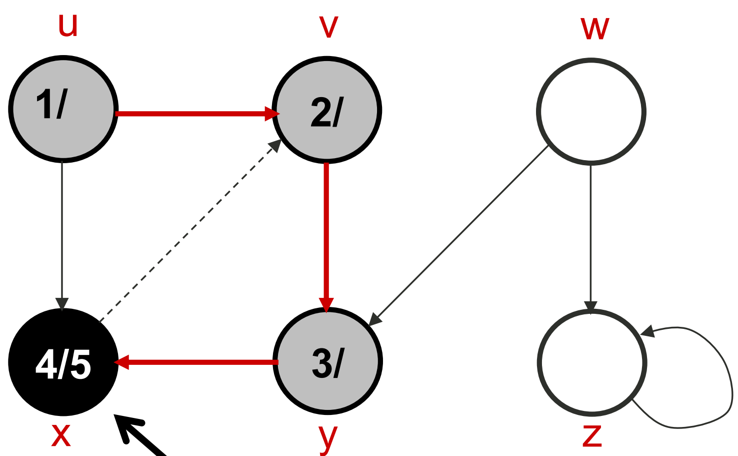
Depth-First Search - Example



Depth-First Search - Example



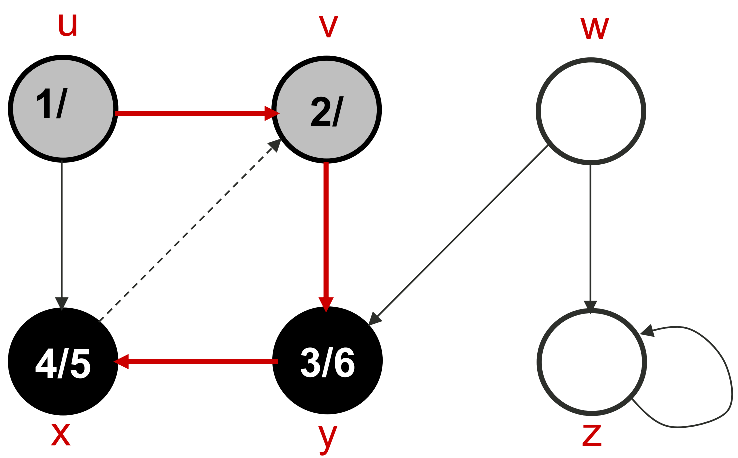
Depth-First Search - Example



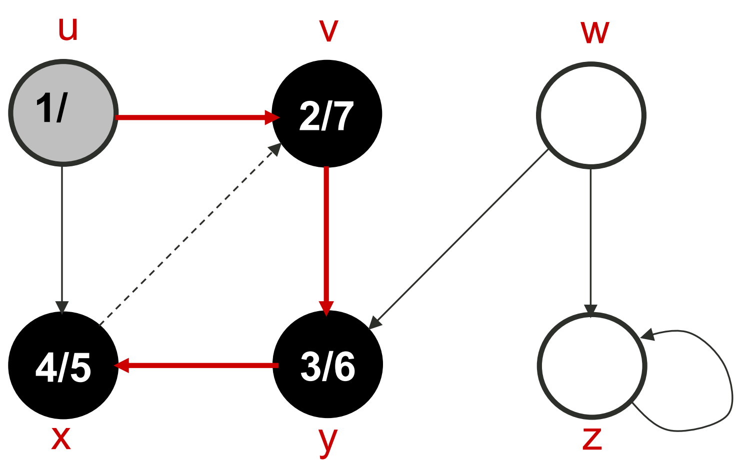
Starting time
d(x)

Finishing time
f(x)

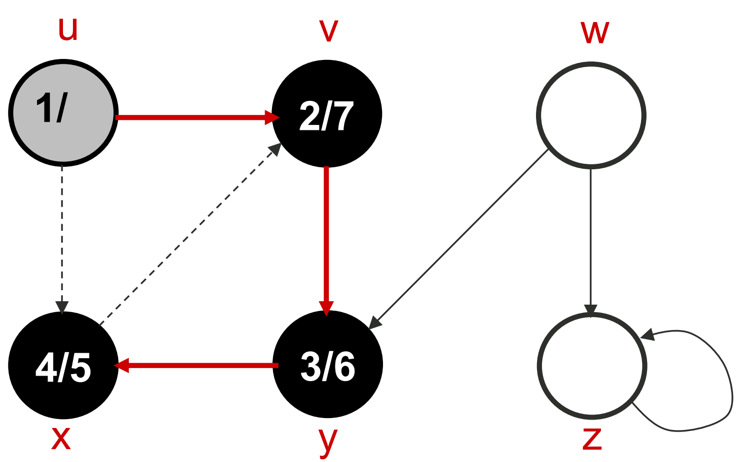
Depth-First Search - Example



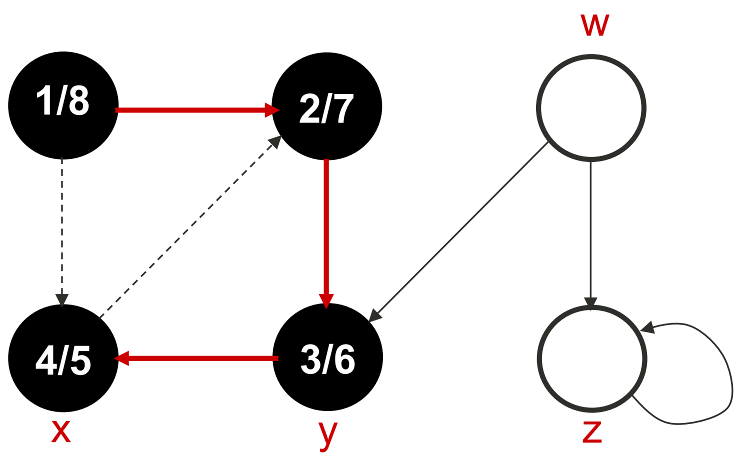
Depth-First Search - Example



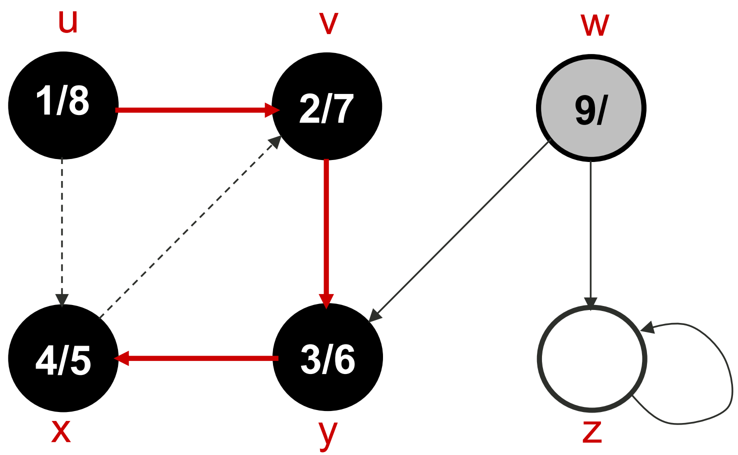
Depth-First Search - Example



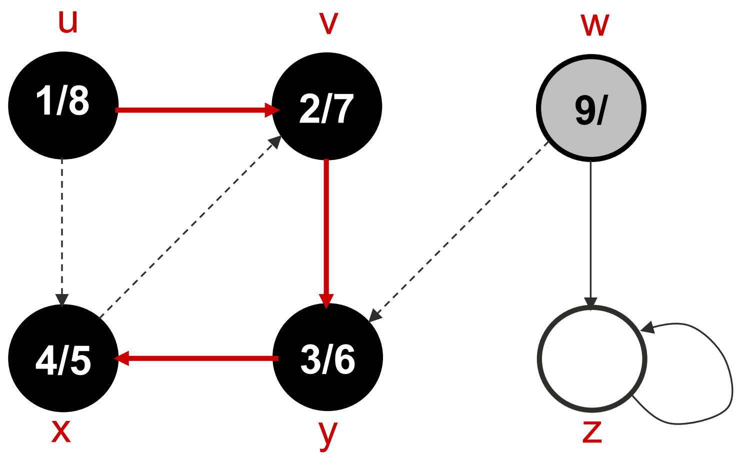
Depth-First Search - Example



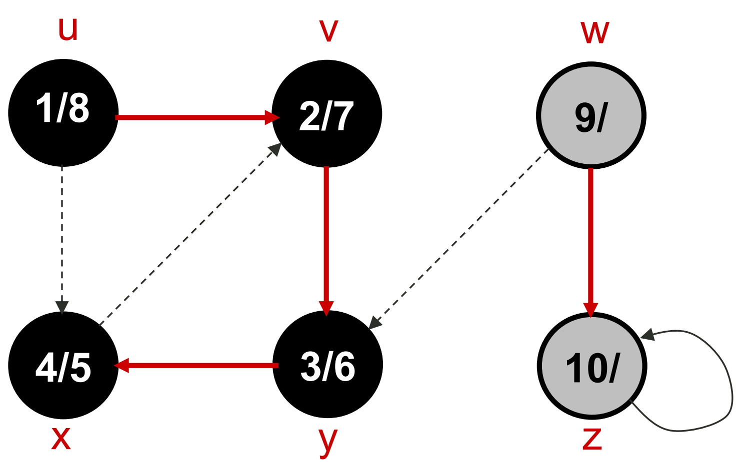
Depth-First Search - Example



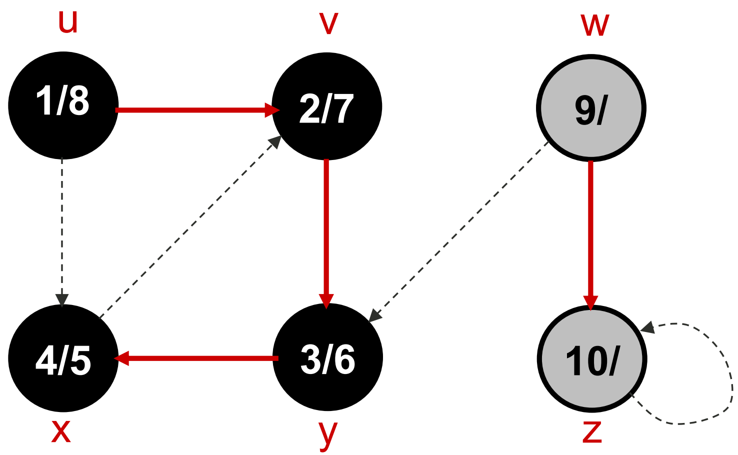
Depth-First Search - Example



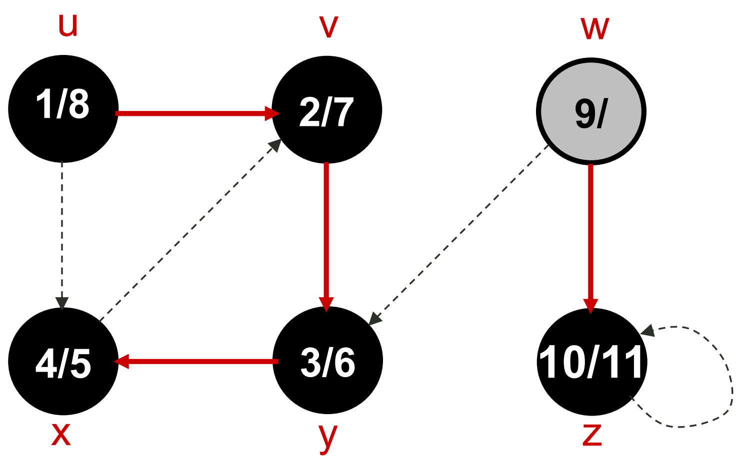
Depth-First Search - Example



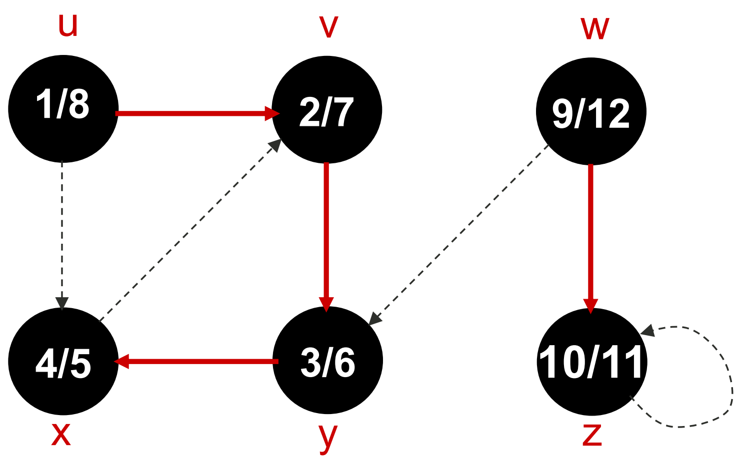
Depth-First Search - Example



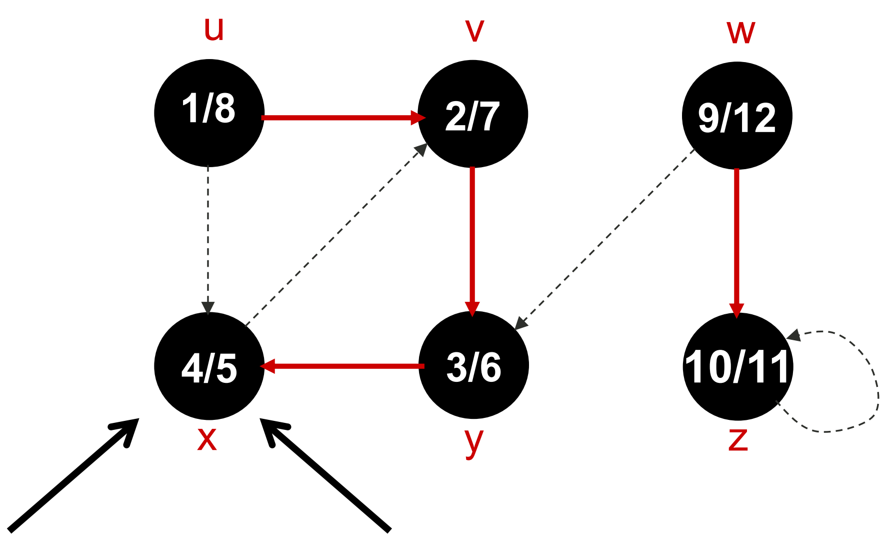
Depth-First Search - Example



Depth-First Search - Example



Depth-First Search - Example



Starting time
d(x)

Finishing time
f(x)

Depth-First Search

```
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
```

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 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 Search - Analysis

```
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 ← Pop
        if v is unmarked
            mark v
            for each edge vw
                Push(w)
```

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 (Shortest Path)
Bag = Queue : Breadth-First (Topological Sort)
Bag = Priority Queue : Best-First (Minimum Spanning Tree)

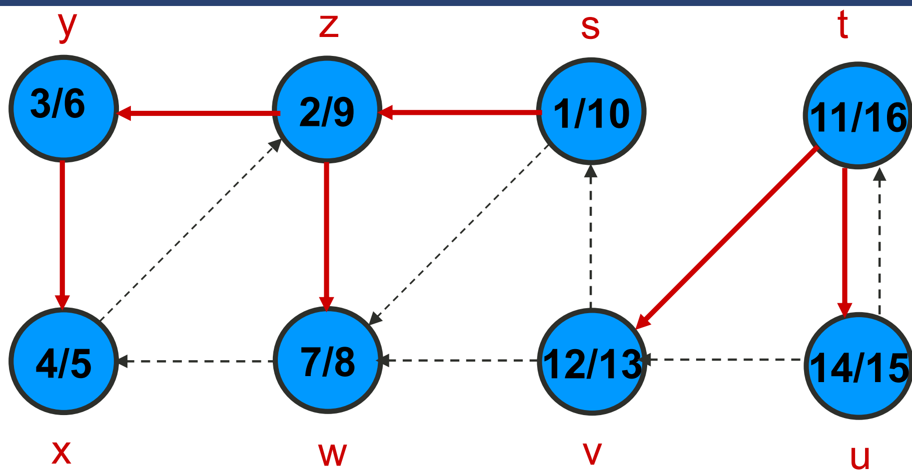
Depth-First Search - Application

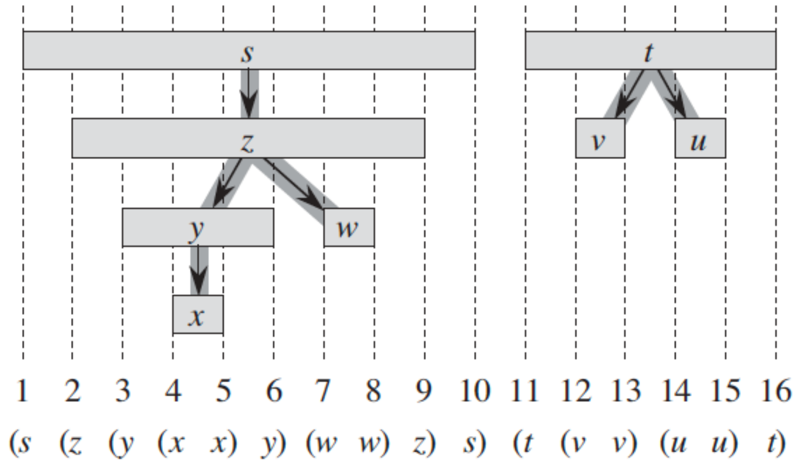
- Depth-first search is often a subroutine in another algorithm.
  - Depth-first search yields valuable information about the structure of a graph.
  - The most basic property of depth-first search is that the predecessor subgraph G does indeed form a forest of trees.
  - Another important property of depth-first search is that discovery and finishing times have **parenthesis structure**.
    - If we represent the discovery of vertex u with a left parenthesis “(u” and represent its finishing by a right parenthesis “u)”, 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





Outline

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