CME 2001
Data Structures and Algorithms

Zerrin Işık
zerrin@cs.deu.edu.tr

Elementary Graph Algorithms

Graphs

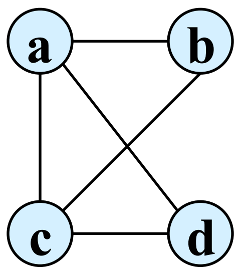
- *Graph* $G = (V, E)$
  - $V =$ set of vertices
  - $E =$ set of edges $\subseteq (V \times V)$
- Types of graphs
  - Undirected: edge $(u, v) = (v, u)$; for all v, $(v, v) \notin E$ (No self loops.)
  - Directed: (u, v) is edge from u to v, denoted as $u \rightarrow v$. Self loops are allowed.
  -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

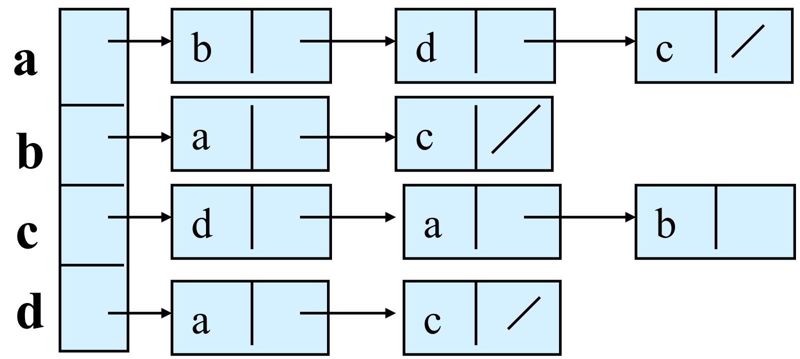
Graphs

- 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.

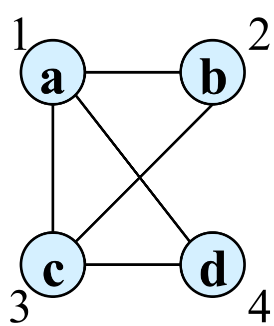
Representation of Graphs

- Two standard ways.
  - Adjacency Lists.





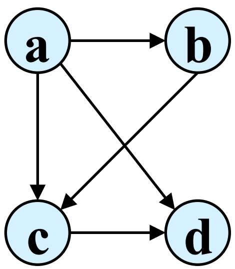
  - Adjacency Matrix.

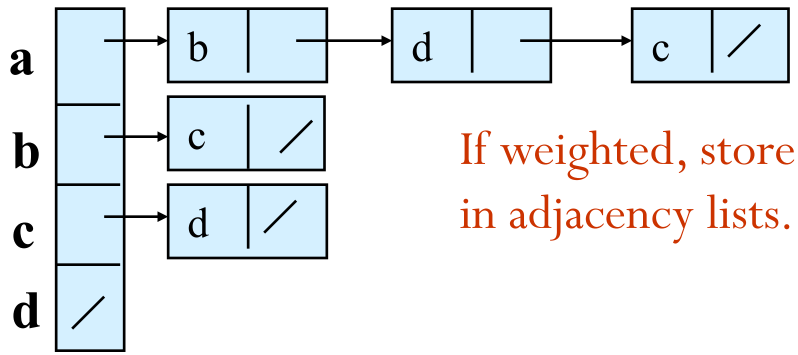


	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

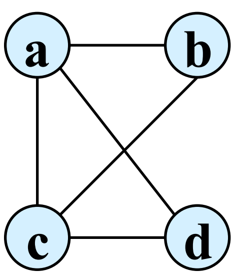
Adjacency Lists

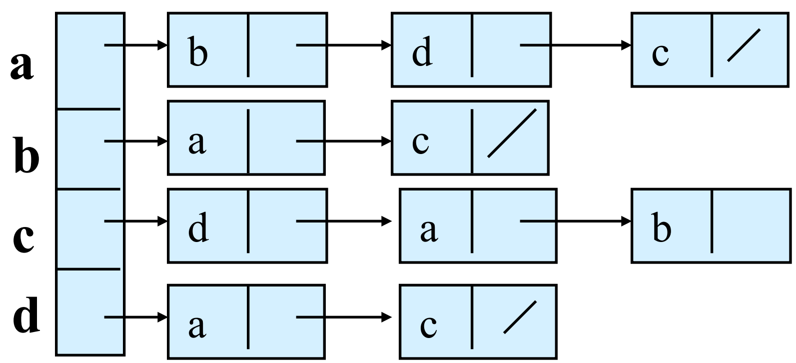
- Consists of an array *Adj* of $|V|$ lists.
- One list per vertex.
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to u.





If weighted, store weights in adjacency lists.





Storage Requirement

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

$$\sum_{v \in V} \text{out-degree}(v) = |E|$$

# of edges leaving v.

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

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

# of edges incident on v.
Edge (u,v) is incident on vertices u and v.

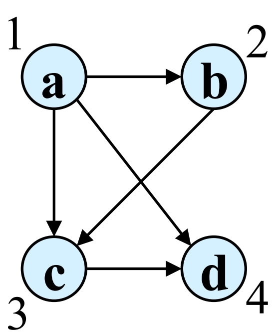
  - Total storage: $\Theta(V+E)$

Pros and Cons: adjacency list

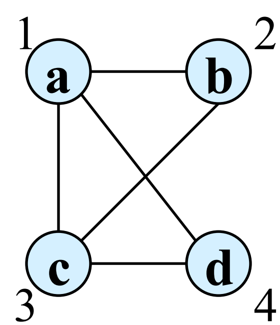
- Pros
  - Space-efficient, when a graph is sparse.
  - Can be modified to support many graph variants.
- Cons
  - Determining if an edge $(u,v) \in G$ is not efficient.
  - Have to search in u's adjacency list. $\Theta(\text{degree}(u))$ time.
  - $\Theta(V)$ in the worst case.

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



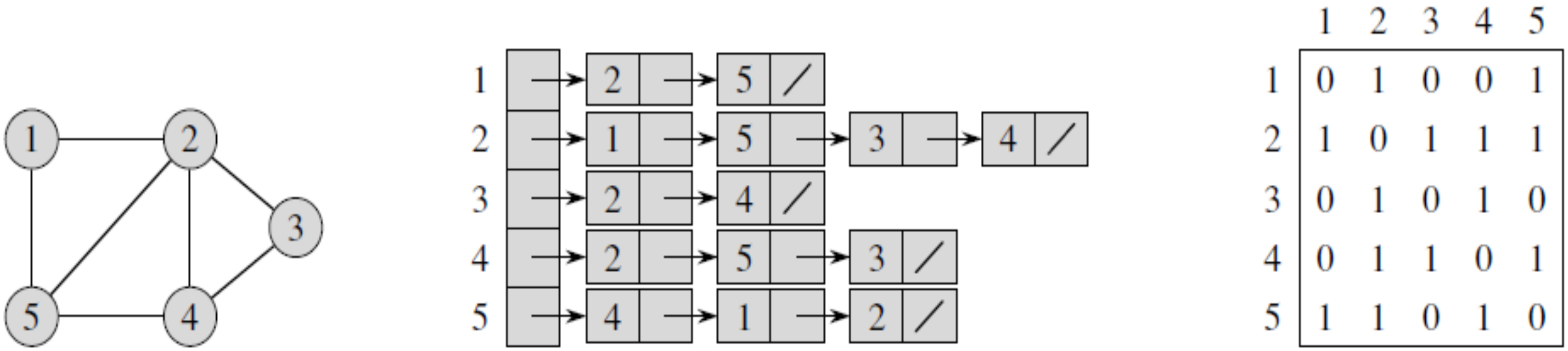
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

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

Undirected Graph



(a) (b) (c)

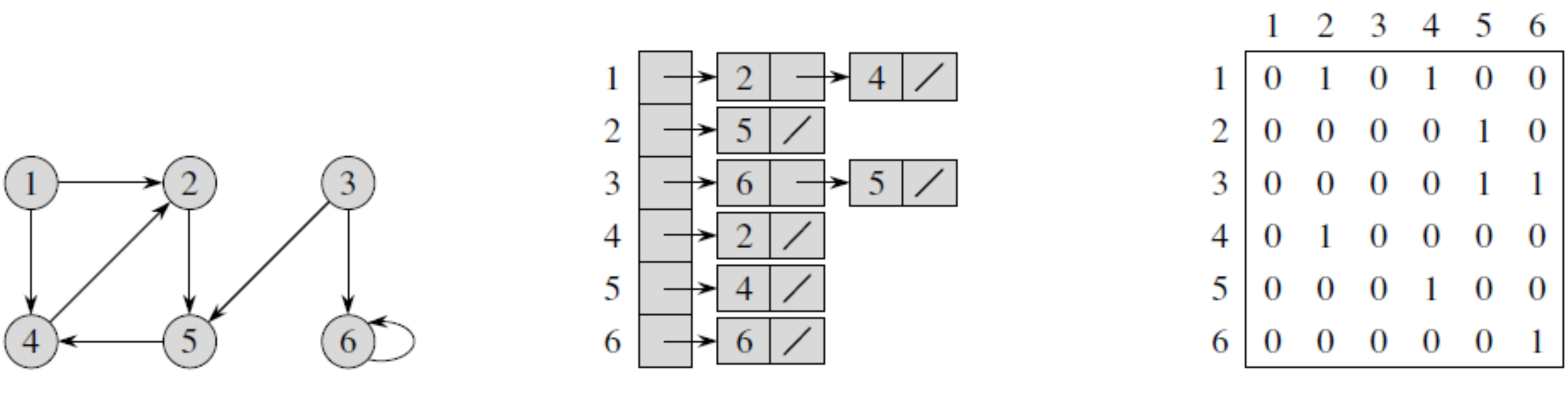
Two representations of an undirected graph.

(a) An undirected graph G *having five* vertices and seven edges.

(b) An adjacency-list representation of G.

(c) The adjacency-matrix representation of G.

Directed Graph



(a) (b) (c)

Two representations of a directed graph.

(a) A directed graph G *having six vertices* and eight edges.

(b) An adjacency-list representation of G.

(c) The adjacency-matrix representation of G.

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 $s \leadsto v$.
    - 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 $1 \leq i \leq k-1$.

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 found during the search.
  - A vertex is finished if all vertices adjacent to it have been discovered.

Pseudo-Code for Breadth-First Search

- Choose a starting vertex
- Search all adjacent vertices
- Return to each adjacent vertex in turn and visit all of its adjacent vertices

breadth-first-search

```
mark starting vertex as visited; put on queue
while the queue is not empty
  dequeue the next node
  for all unvisited vertices adjacent to this one
    • mark vertex as visited
    • add vertex to queue
```

```
BFS(G, s)                              // G is the graph and s is the starting node
1  for each vertex u ∈ V [G] - {s}
2       do color[u] ← WHITE
3          d[u] ← ∞
4          π[u] ← NIL
5  color[s] ← GRAY
6  d[s] ← 0
7  π[s] ← NIL
8  Q ← Ø
9  ENQUEUE(Q, s)
10  while Q ≠ Ø                          // Lines 10-18 iterates as long as there are gray vertices.
11      do u ← DEQUEUE(Q)
12         for each v ∈ Adj[u]
13             do if color[v] = WHITE             // discover the undiscovered adjacent vertices
14                 then color[v] ← GRAY          // enqueued whenever painted gray
15                      d[v] ← d[u] + 1
16                      π[v] ← u
17                      ENQUEUE(Q, v)
18         color[u] ← BLACK          // painted black whenever dequeued
```

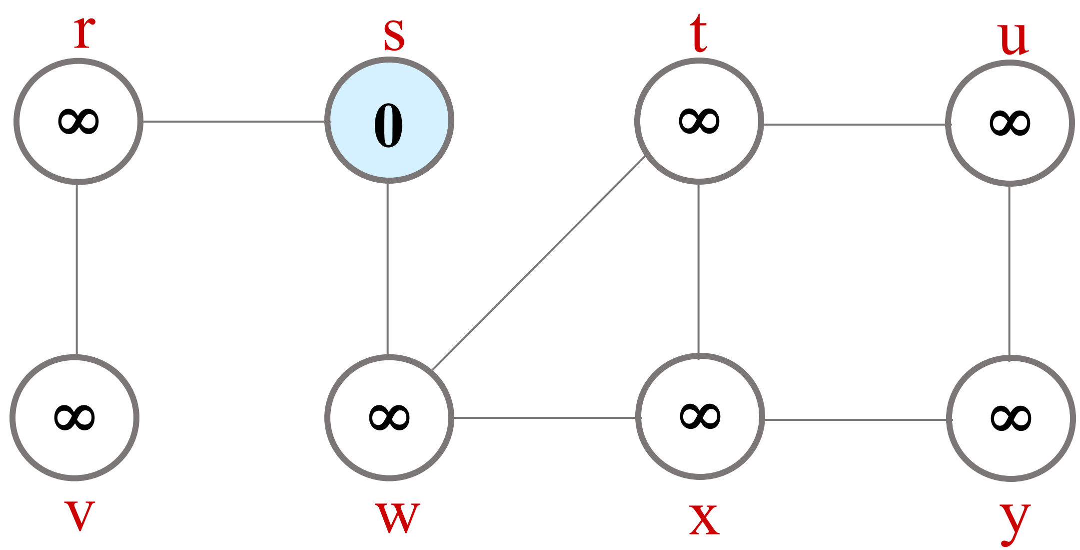
Lines 1–4 paint every vertex white,
Set *d[u]* to be infinity for each vertex *u*,
Set the parent of every vertex to be *NIL*.

Line 5 paints the source vertex *s* ***gray***

Line 6 initializes *d[s]* to 0,
Line 7 sets the predecessor of the source to be *NIL*.
Lines 8–9 initialize *Q* to the queue containing just the vertex *s*.

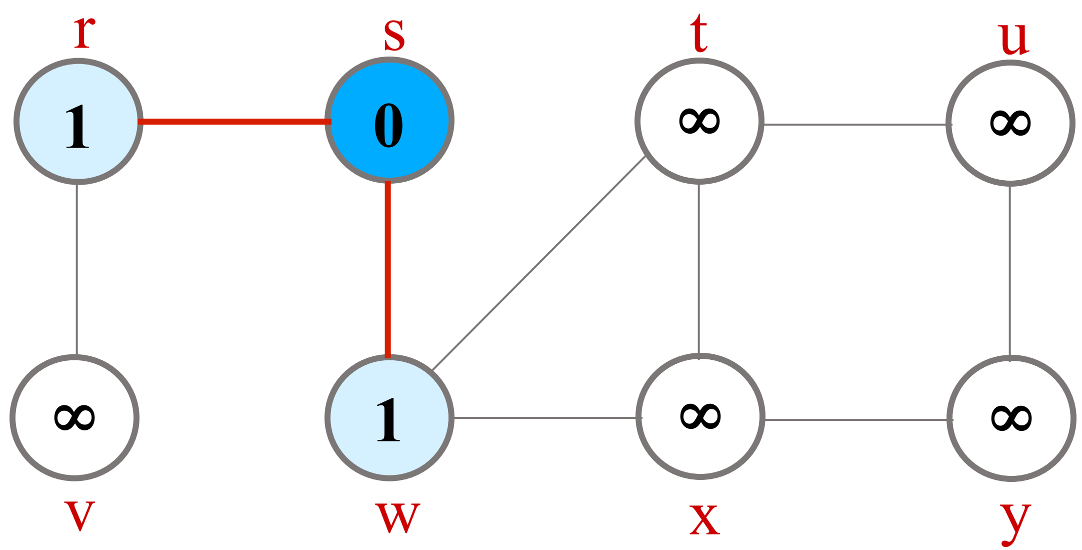
Q: a queue of discovered v
color[*v*]: color of v
d[*v*]: distance from s to v
π[*u*]: predecessor of v
white: undiscovered
gray: discovered
black: finished

Example (BFS)



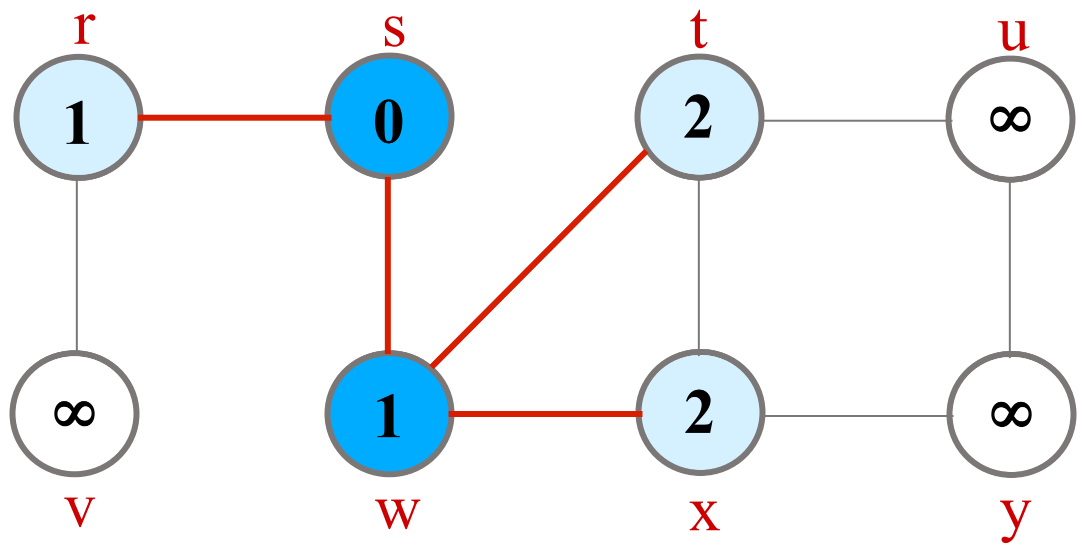
Q:	s
	0

Example (BFS)



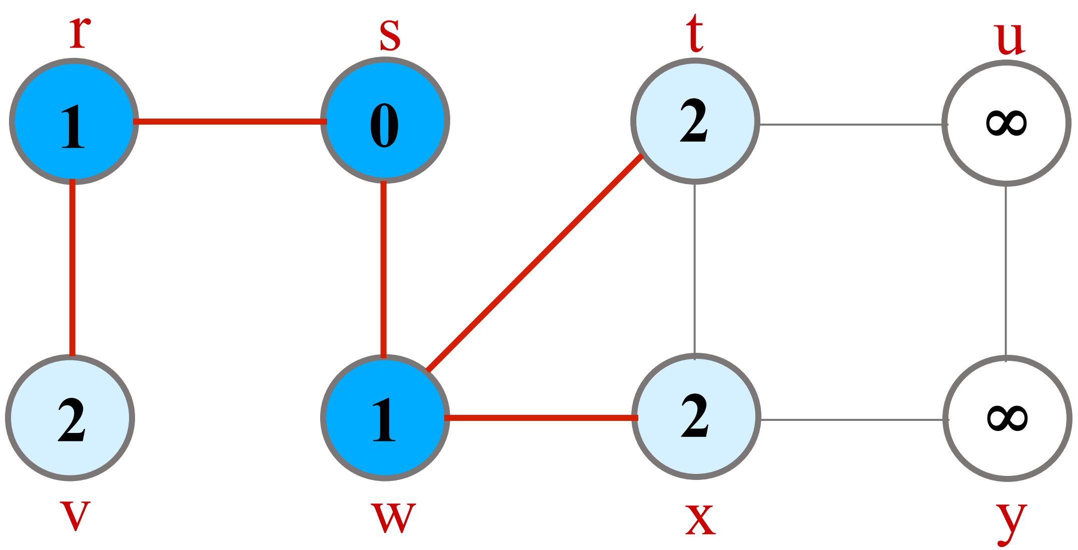
Q: w r
1 1

Example (BFS)



Q:	r	t	x
	1	2	2

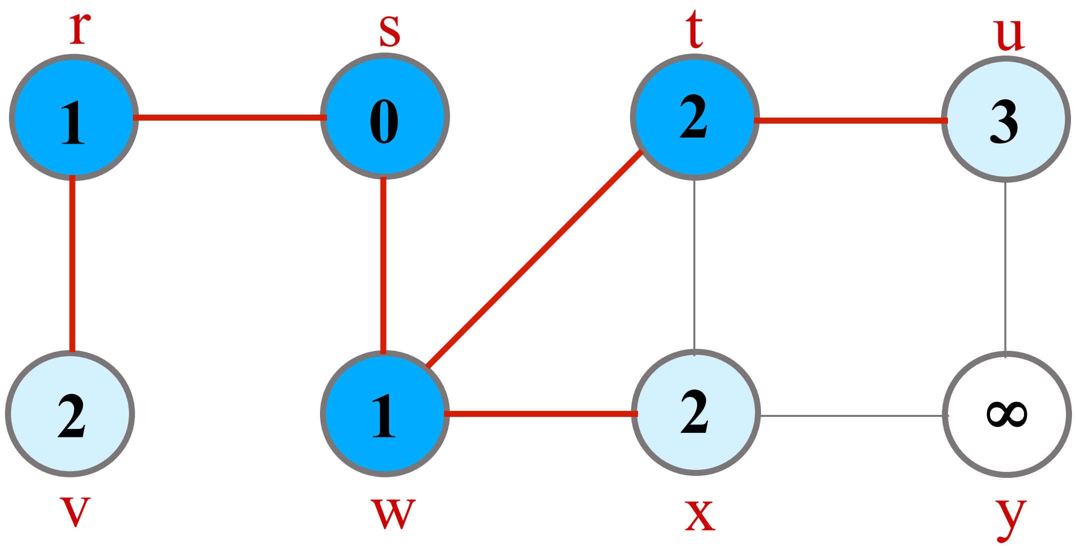
Example (BFS)



Q: t x v

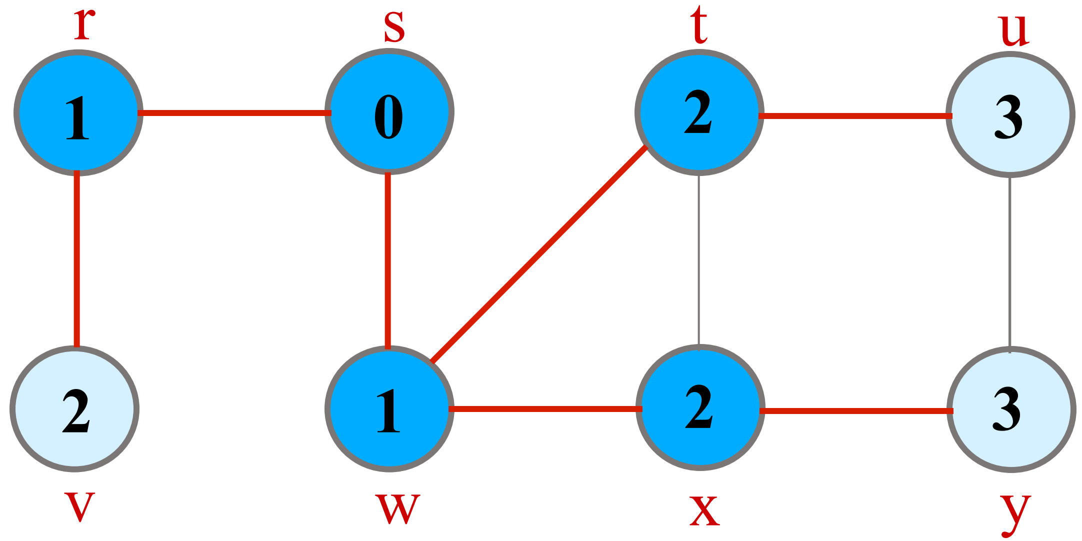
2 2 2

Example (BFS)



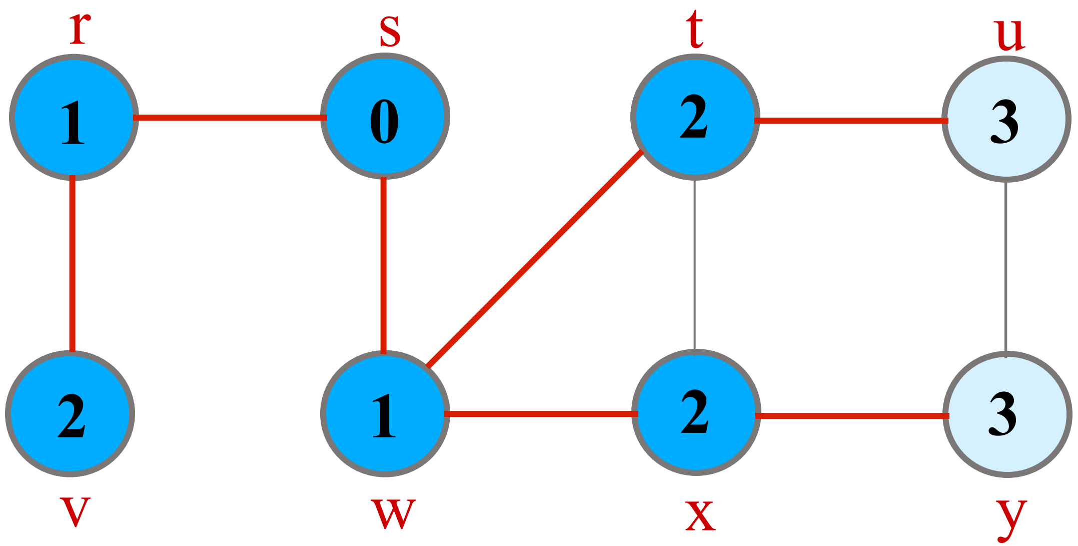
Q:	x	v	u
	2	2	3

Example (BFS)



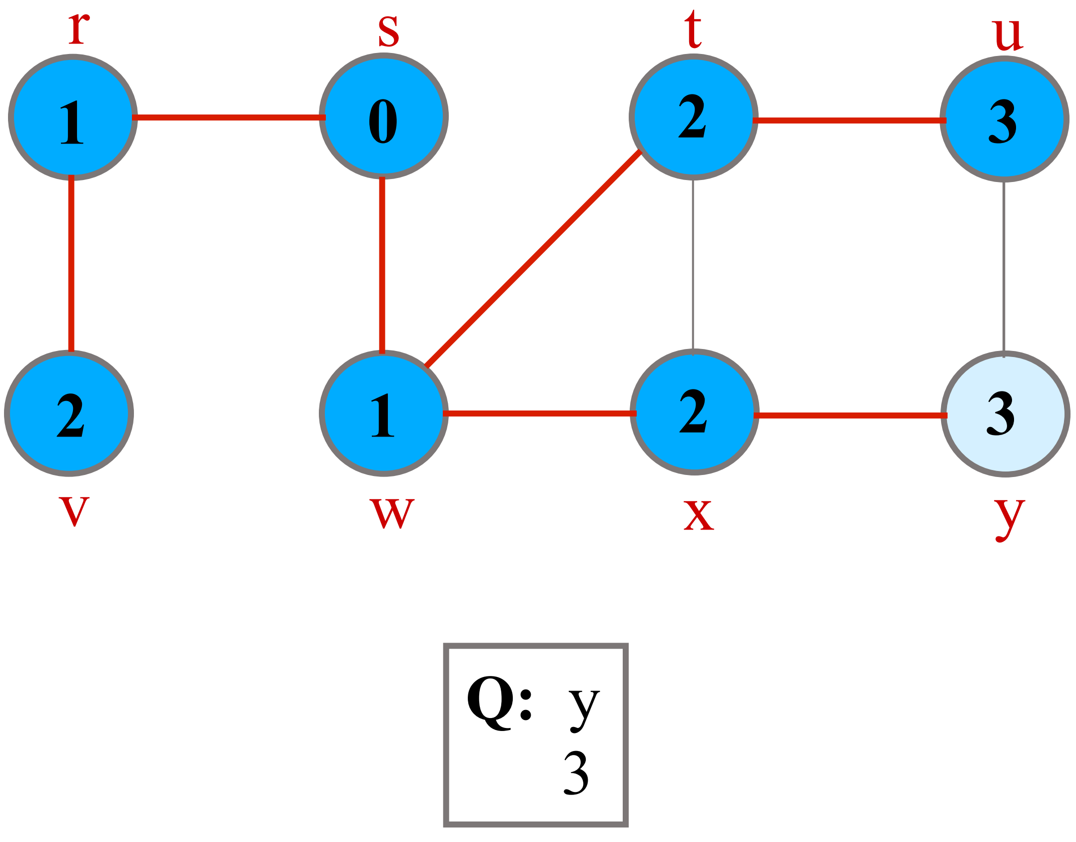
Q:	v	u	y
	2	3	3

Example (BFS)

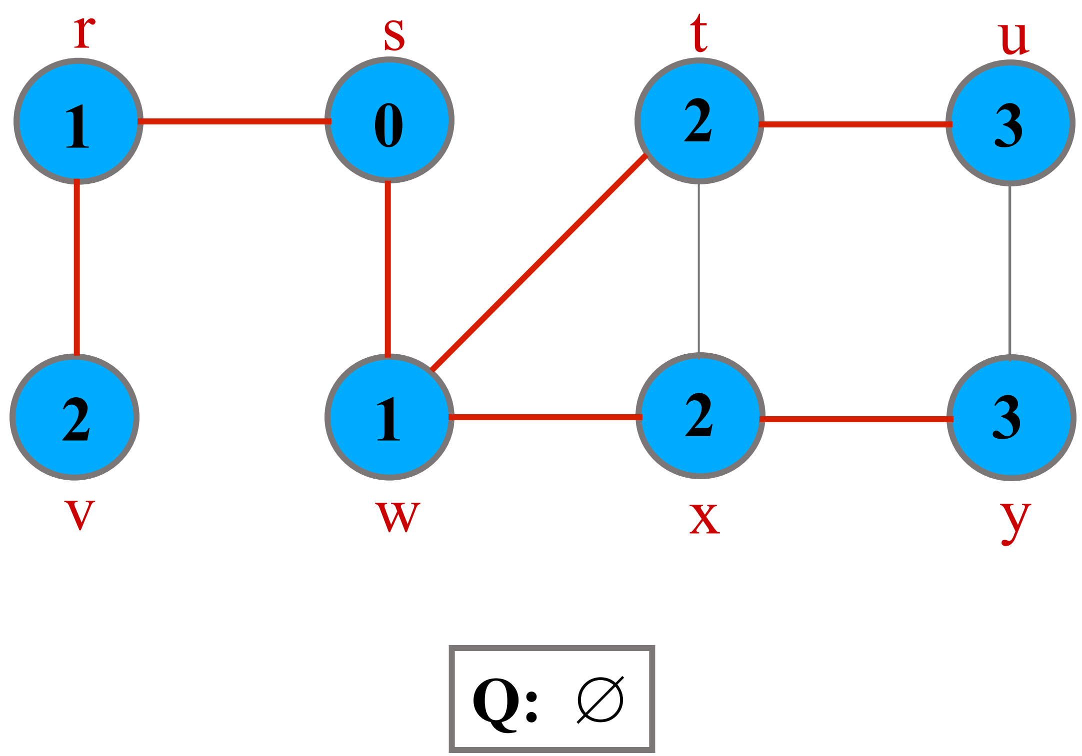


Q: u y
3 3

Example (BFS)

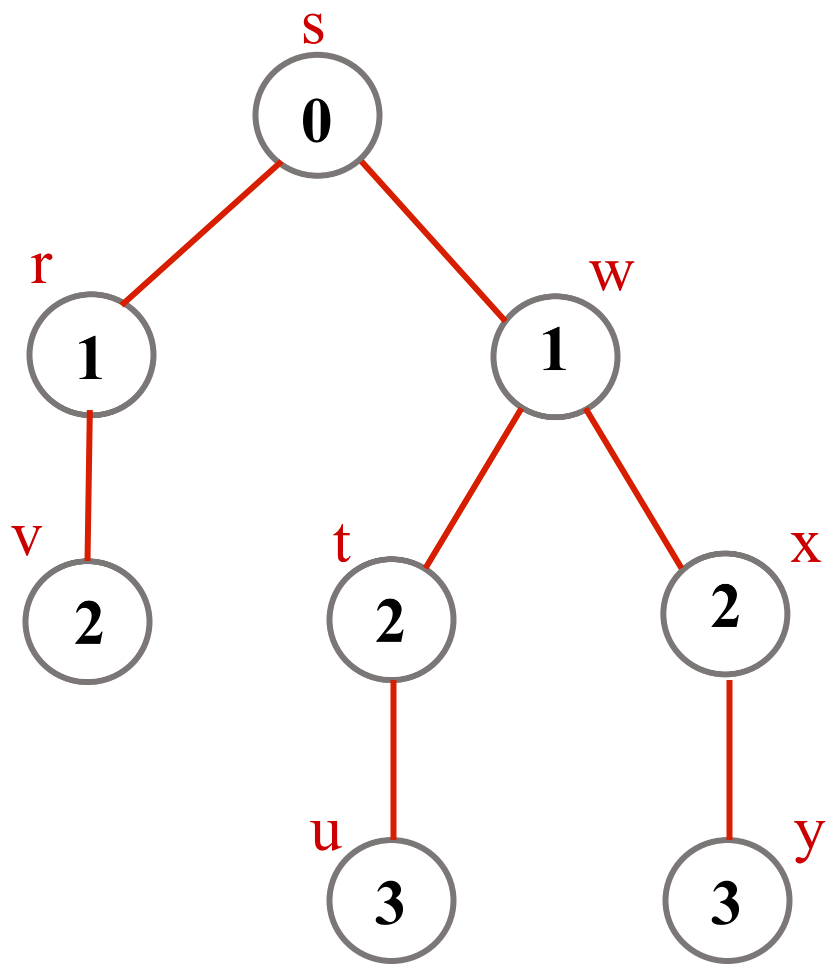


Example (BFS)



Example (BFS)

BF Tree



Analysis of BFS

```
BFS(G, s)
for each vertex u ∈ V [G] - {s}
    do color[u] ← WHITE
       d[u] ← ∞
       π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← Ø
ENQUEUE(Q, s)
while Q ≠ Ø
    do u ← DEQUEUE(Q)
       for each v ∈ 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
```

Lines 1–4: $O(V)$

Lines 9–18: $O(V+E)$

Each vertex is enqueued and dequeued at most once, and each operation takes $O(1)$. So, total time for queuing is $O(V)$.

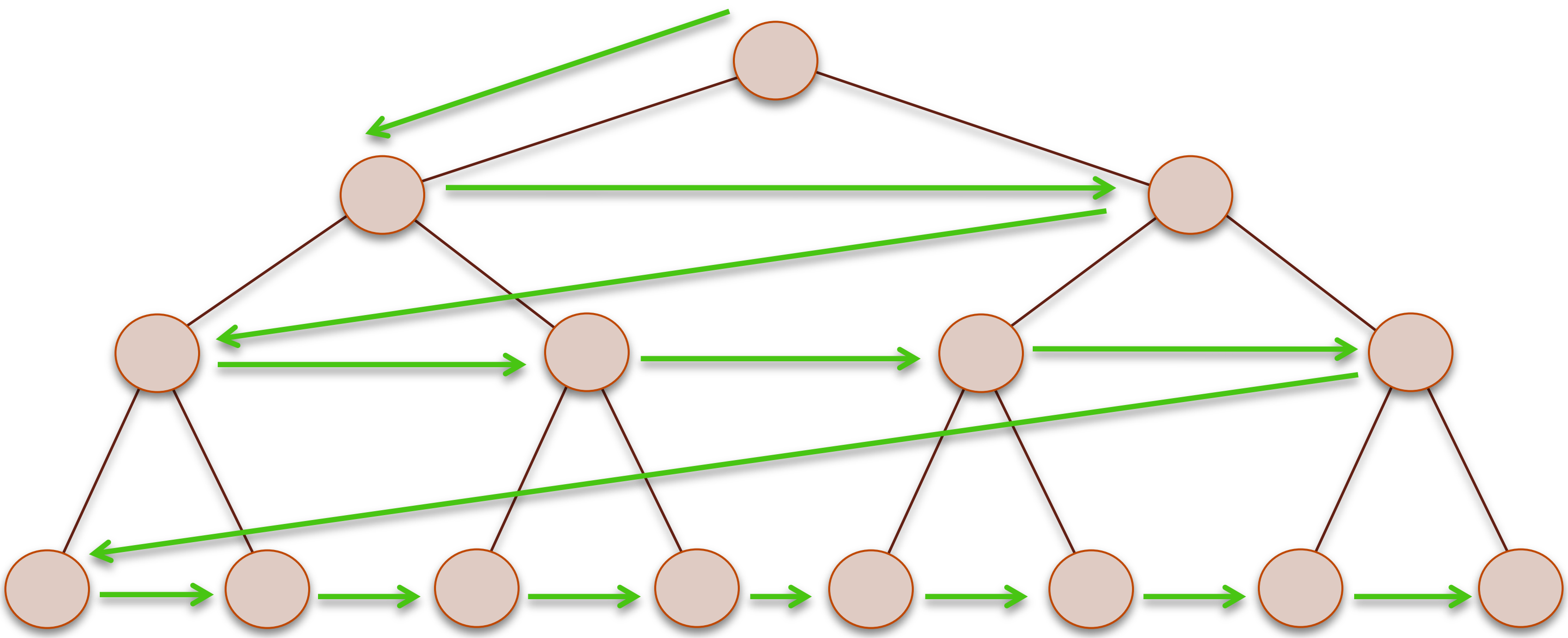
The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency lists is $O(E)$.

⇒ Total time complexity : $\boldsymbol{O(V+E)}$

⇒ Space complexity:

- ⇒Linear ($\Theta(V+E)$) in the size of adjacency list representation.
- ⇒Quadratic ($\Theta(V^2)$) in the size of adjacency matrix representation.

Breadth-First Search (BFS)



Applications of BFS

- Shortest paths in graphs which have equal edge weights.
- To compute maximum flow in Ford-Fulkerson method.
- Certain pattern (e.g., triangular) matching in a large graph.
- To test if a graph has the bipartite property or not
- …

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

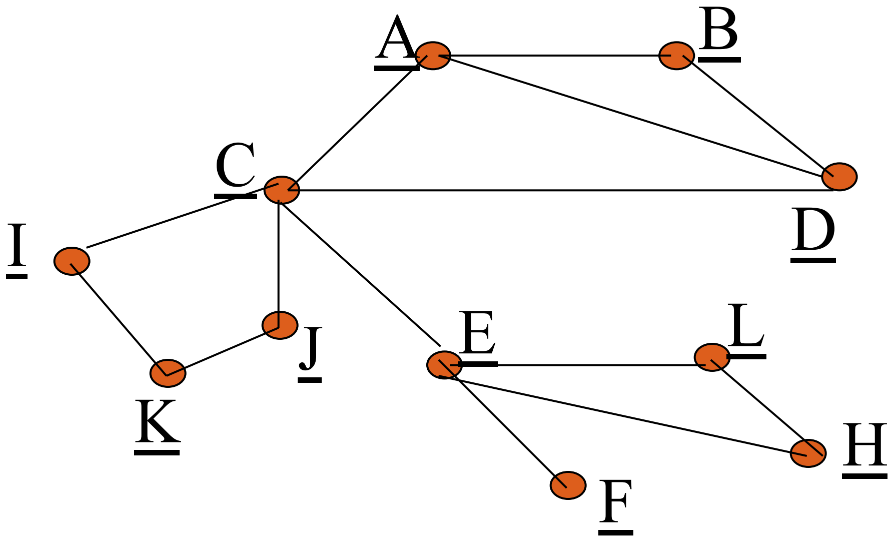
- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - 2 **timestamps** on each vertex. Integers between 1 and $2|V|$.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***finishing time*** (v turns from gray to black)
  - $\pi[v]$: predecessor of v is u, such that v was discovered during the scan of u's adjacency list.
- Uses the same coloring scheme for vertices as BFS.

Depth-First Search

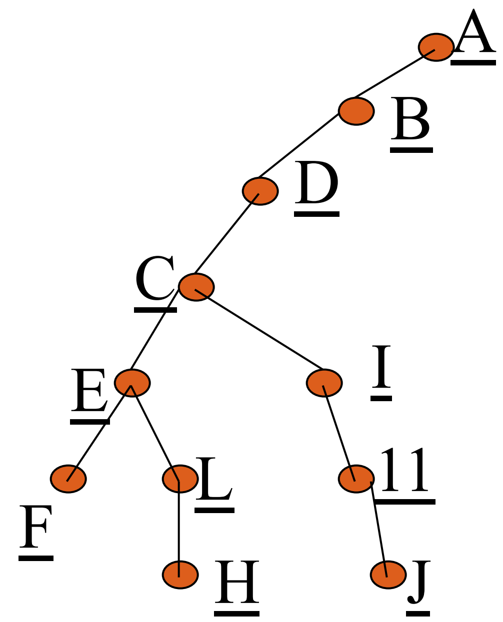
DFS follows the following rules:

1. Select an unvisited node ***x***, visit it, and treat as the current node;
2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
3. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
4. Repeat steps 3 and 4 until no more nodes can be visited.
5. If there are still unvisited nodes, repeat from step 1.

Illustration of DFS



Graph G



DFS Tree

DFS(G)

```
1.  for each vertex u ∈ V [G]
2.      do color[u] ← white
3.          π[u] ← NIL
4.  time ← 0
5.  for each vertex u ∈ V [G]
6.      do if color[u] = white
7.          then DFS-Visit(u)
```

Lines 1–3 paint all vertices white and initialize their π fields to *NIL*. Line 4 resets the global time counter. Lines 5–7 check each vertex in ***V*** in turn and, when a white vertex is found, visit it using *DFS-VISIT*. Every time DFS-VISIT(u) is called in line 7, vertex ***u*** becomes the root of a new tree in the depth-first forest. When DFS returns, every vertex ***u*** has been assigned a discovery time ***d[u]*** and a finishing time ***f [u].***

DFS-Visit(u)

```
1.  color[u] ← GRAY   // White vertex u has been
                         discovered
2.  time ← time + 1
3.  d[u] ← time
4.  for each v ∈ Adj[u]
5.      do if color[v] = WHITE
6.          then π[v] ← u
7.               DFS-Visit(v)
8.  color[u] ← BLACK     // it is finished.
9.  f[u] ← (time ← time + 1)
```

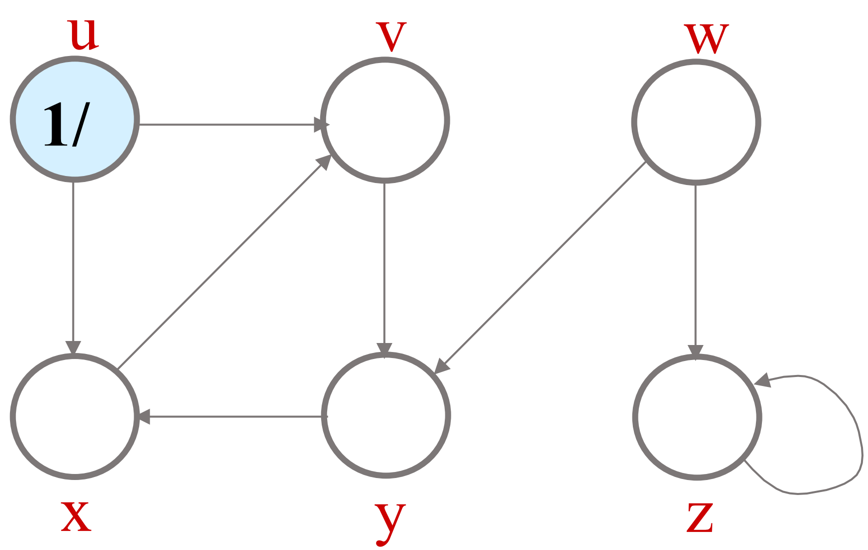
Line 1 paints ***u gray***, line 2 increments the global variable ***time***, and line 3 records the new value of time as the discovery time ***d[u]***. Lines 4–7 examine each vertex ***v*** adjacent to ***u*** and recursively visit ***v*** if it is white. As each vertex ***v*** $\in$ ***Adj[u]*** is considered in line 4, we say that edge ***(u, v)*** is ***explored*** by the depth-first search. Finally, after every edge leaving ***u*** has been explored, lines 8–9 paint ***u*** black and record the finishing time in *f [u]*.

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.

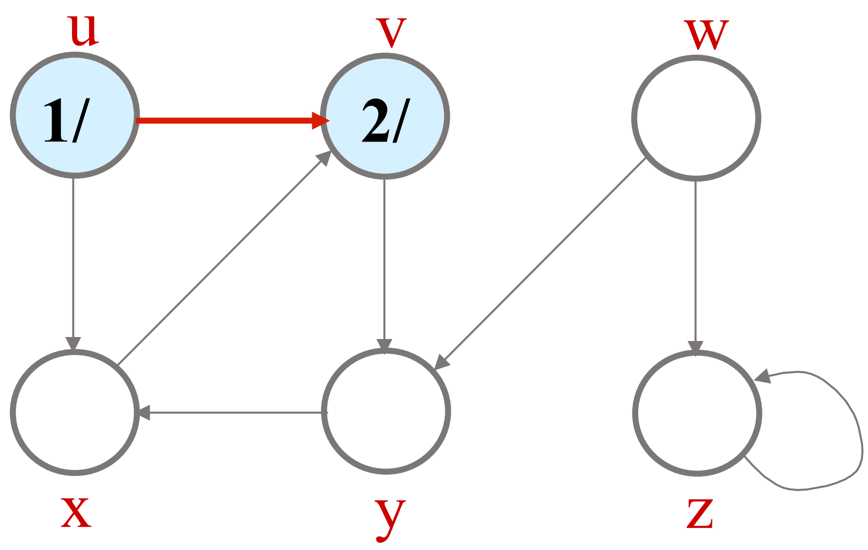
Theorem:
In DFS of an undirected graph, we get only tree and back edges.
No forward or cross edges.

Example (DFS)

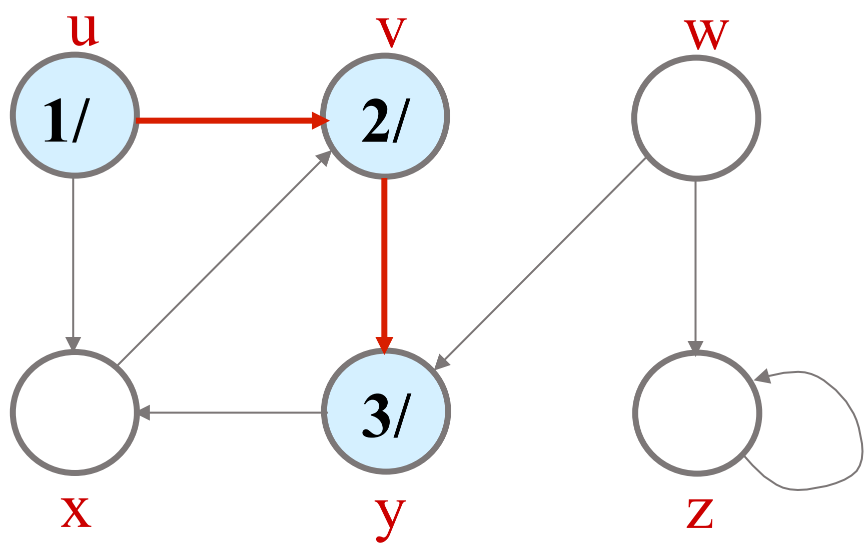


discovery t. / finishing t.

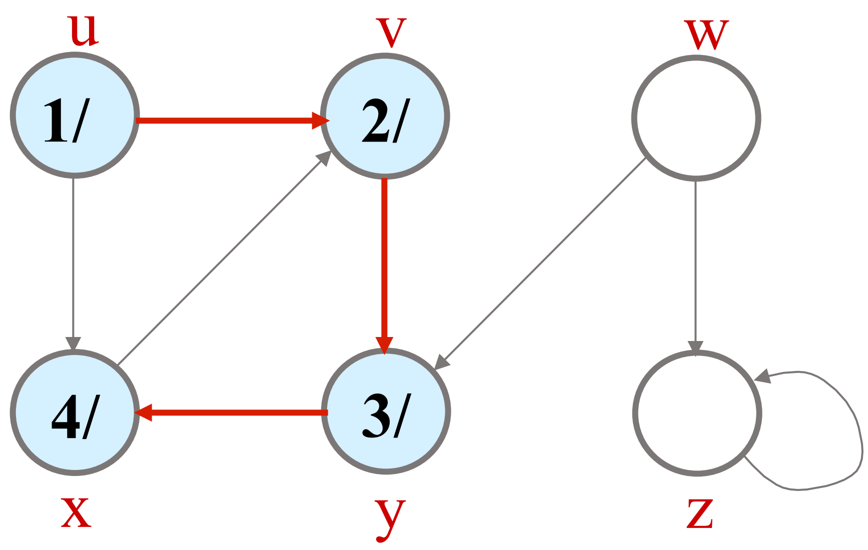
Example (DFS)



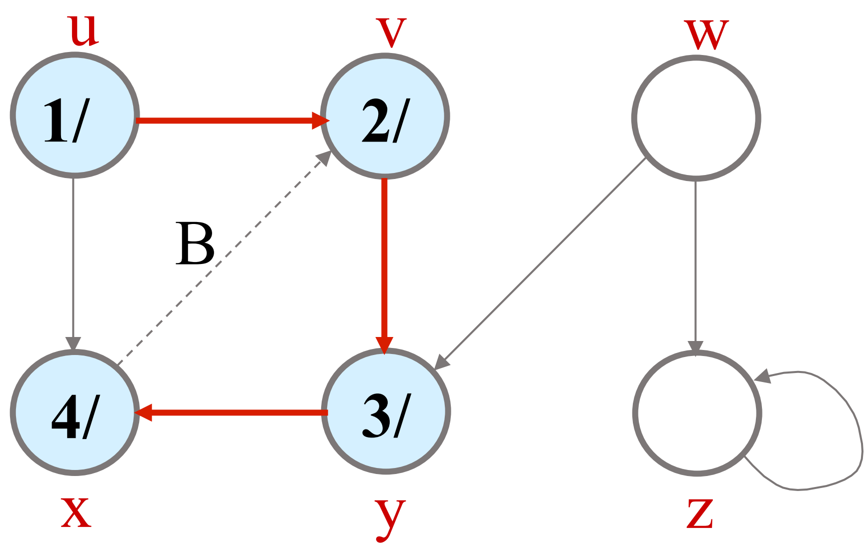
Example (DFS)



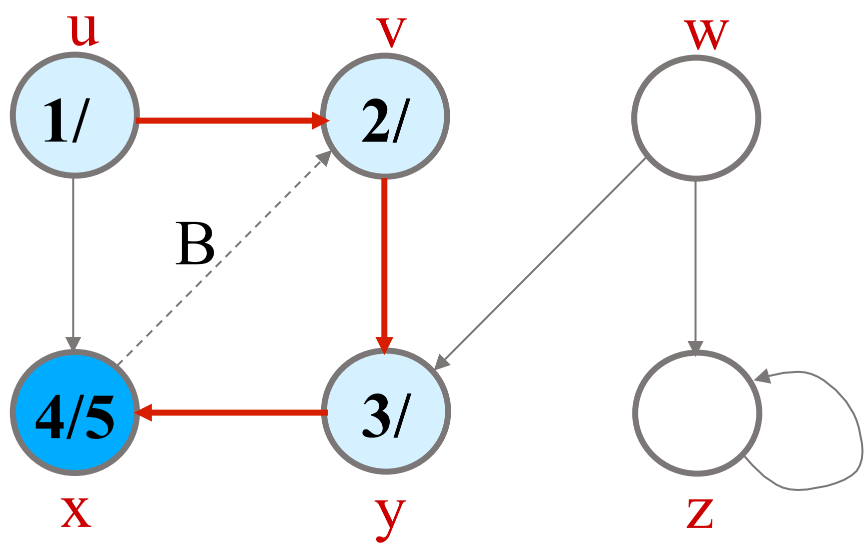
Example (DFS)



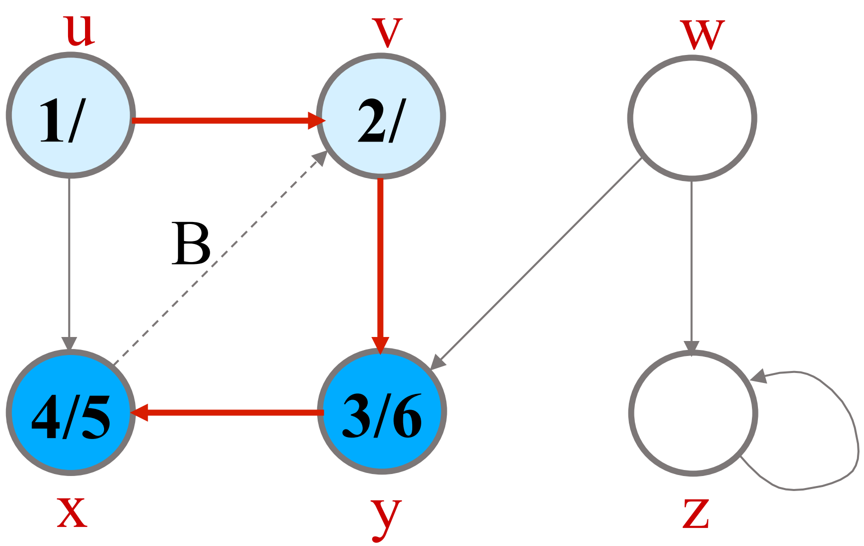
Example (DFS)



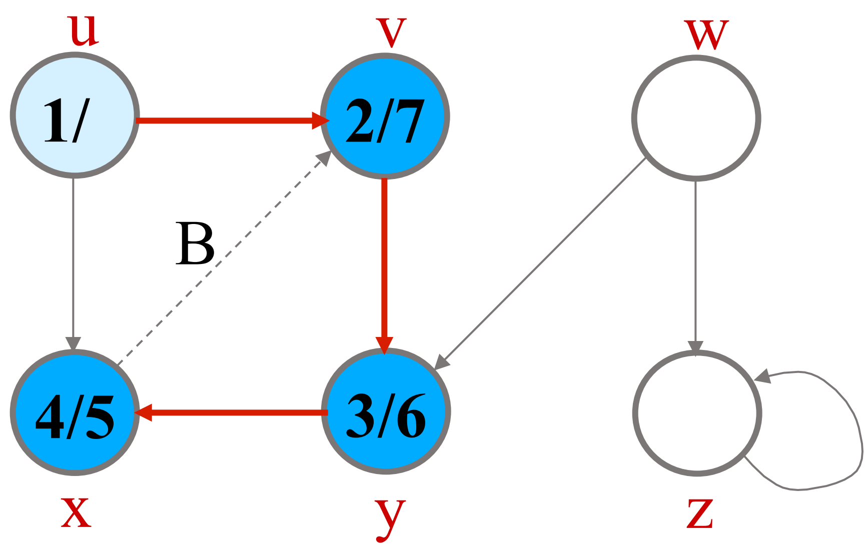
Example (DFS)



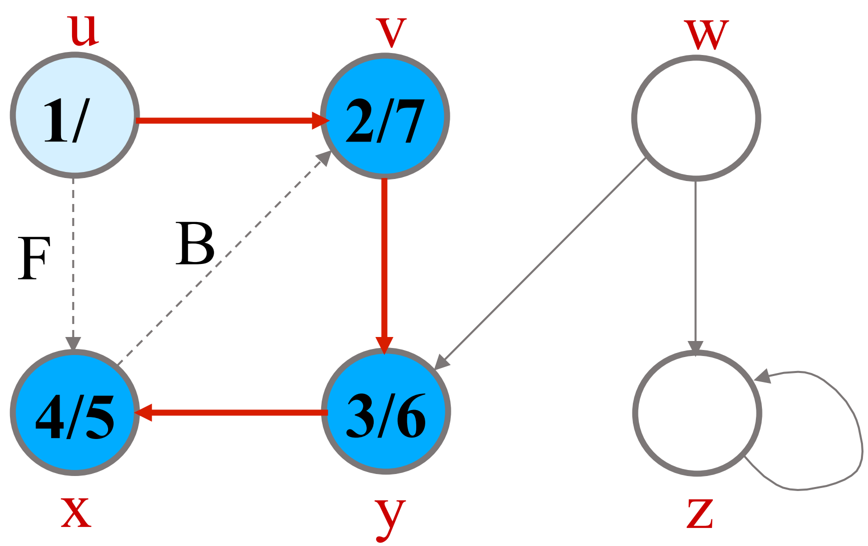
Example (DFS)



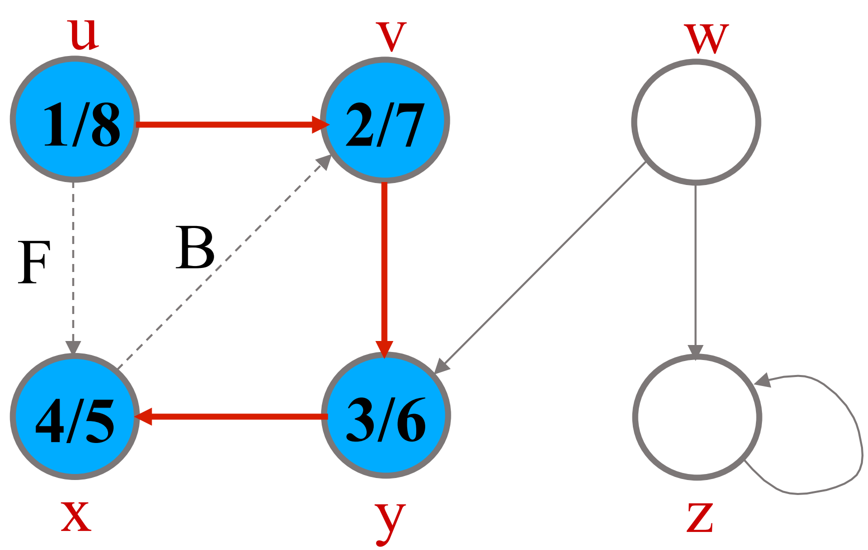
Example (DFS)



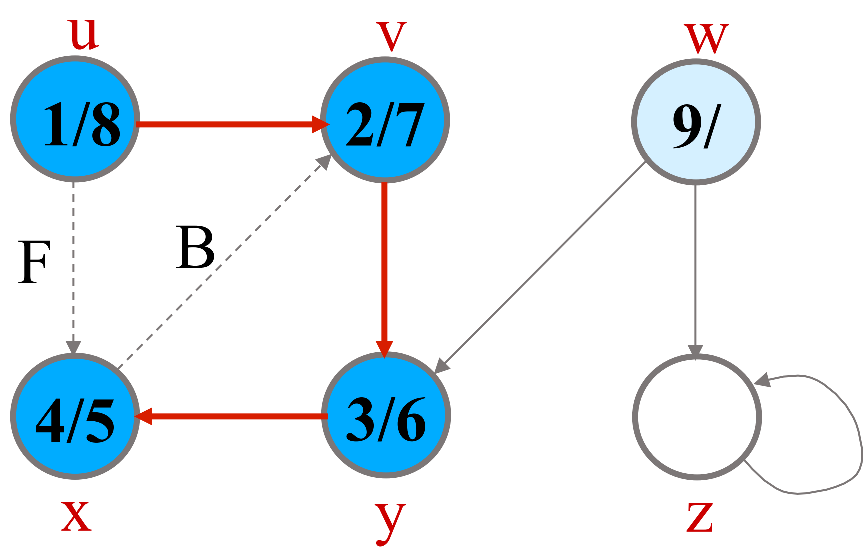
Example (DFS)



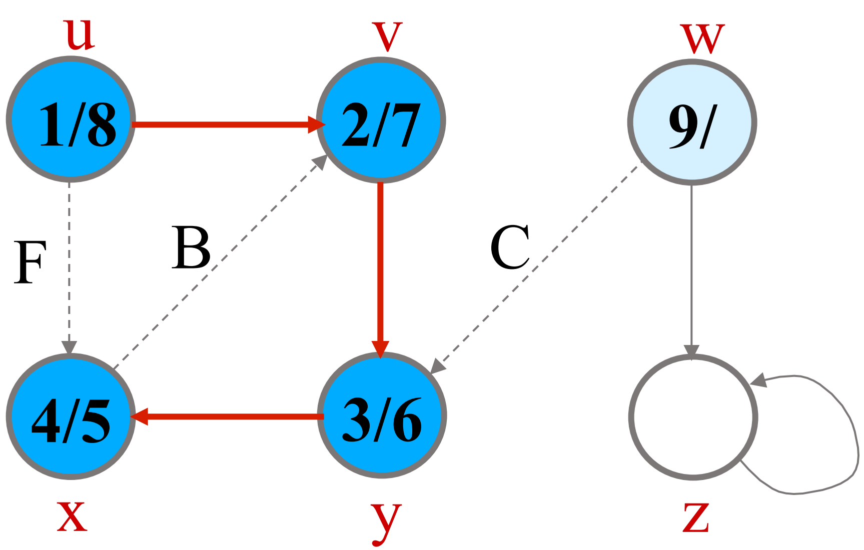
Example (DFS)



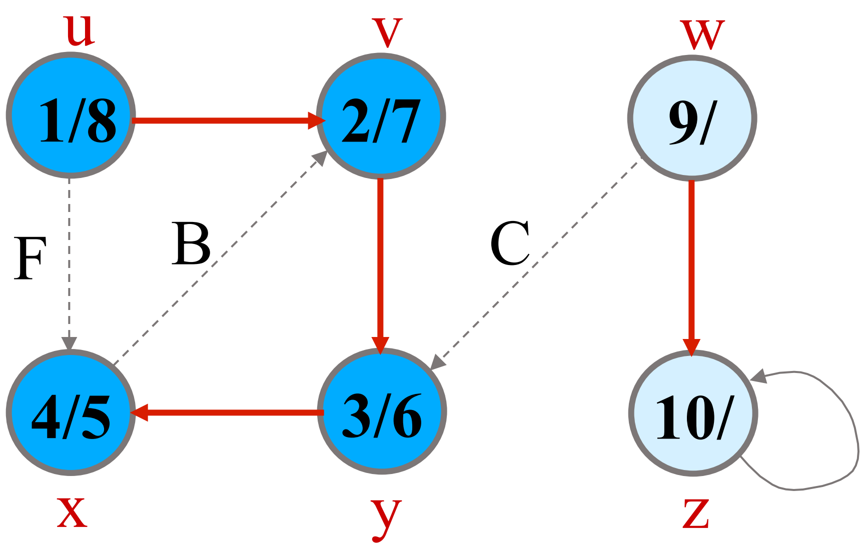
Example (DFS)



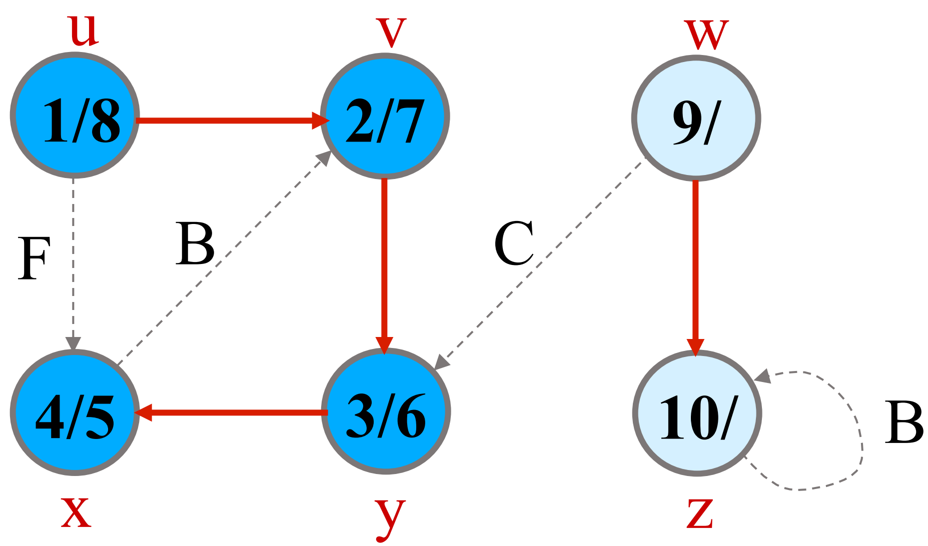
Example (DFS)



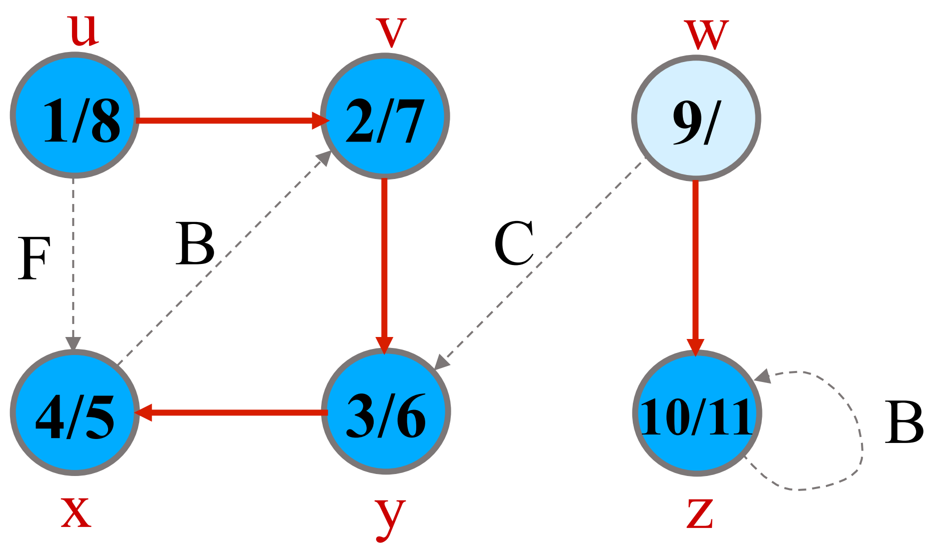
Example (DFS)



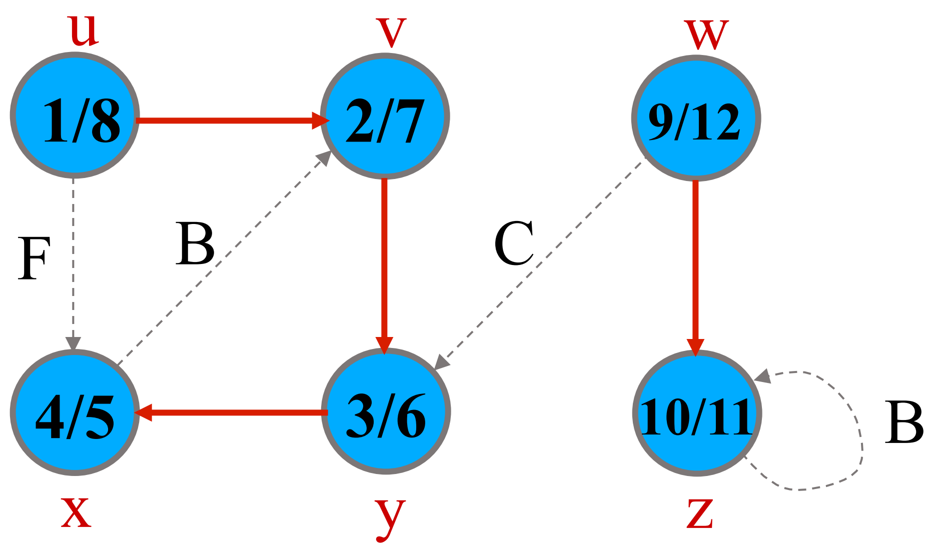
Example (DFS)



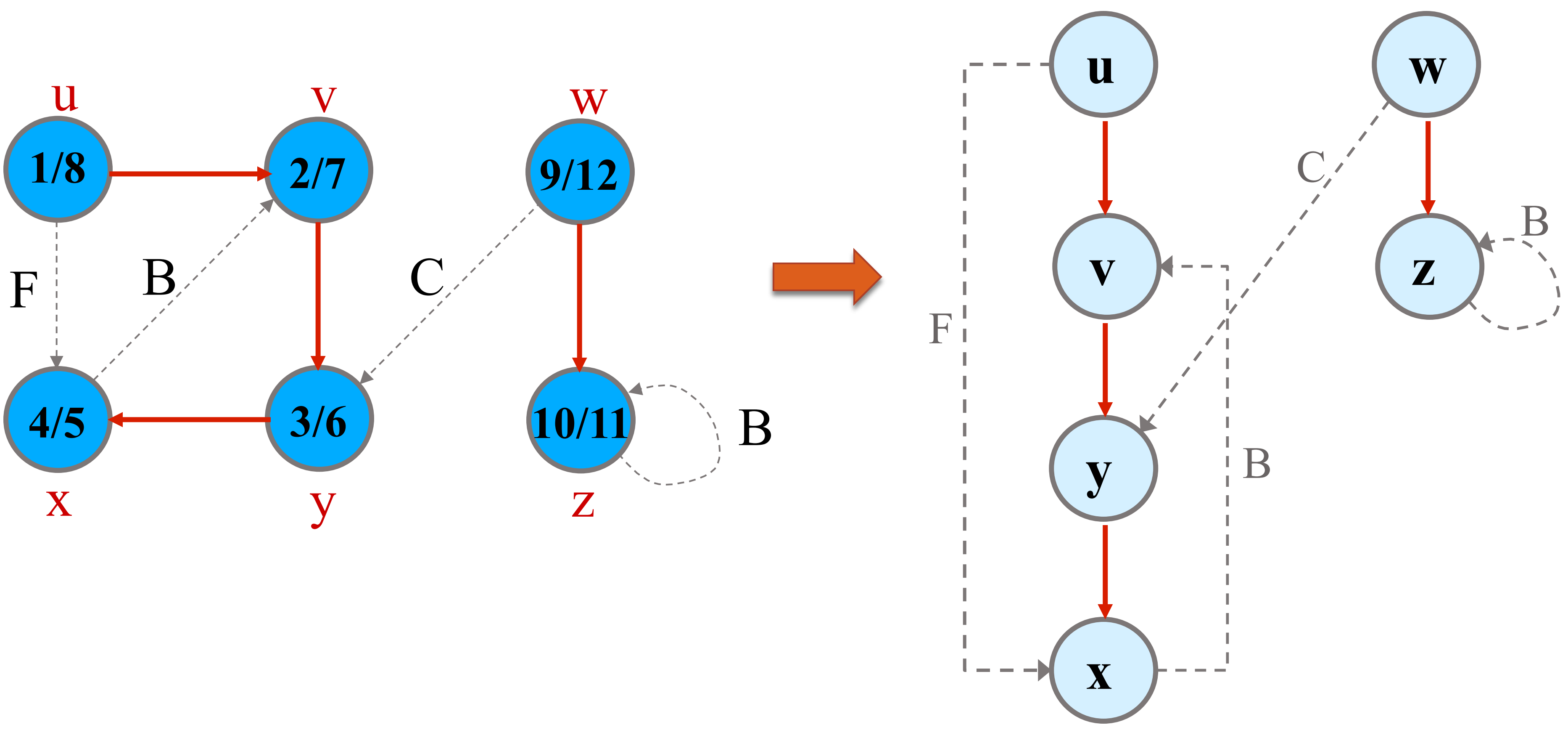
Example (DFS)



Example (DFS)



DF Tree



Analysis of DFS

DFS(G)

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

Loops on lines 1-2 and 5-7 take $\Theta(V)$ time (excluding time to execute DFS-Visit.)

DFS-Visit(u)

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

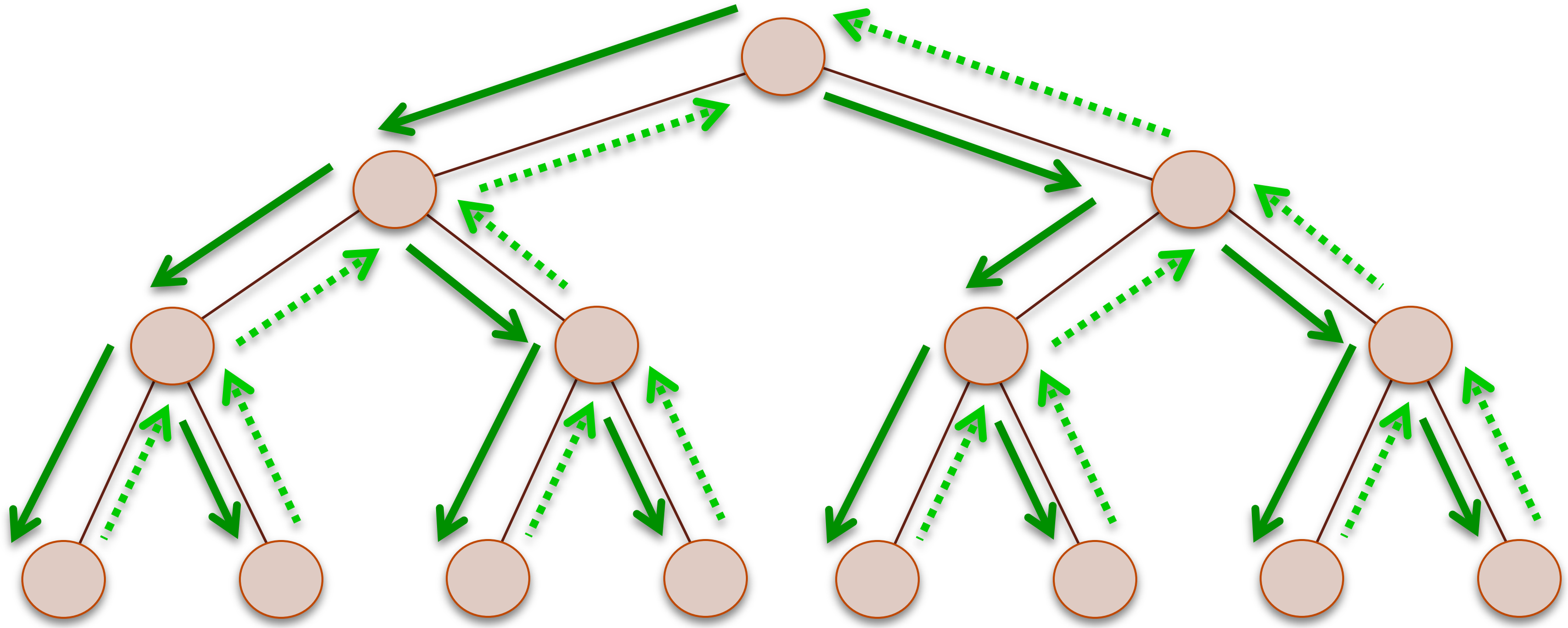
DFS-Visit is called once for each white vertex $v \in V$ when it's painted gray the first time.

Lines 3-6 are executed $|\mathrm{Adj}[v]|$ times.

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

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

Depth-First Search (DFS)



Applications of DFS

- Topological sorting of vertices
- Find connected components of a large graph
- Find bridges of a graph
- Solve one solution puzzles (e.g. maze)
- Find bi-connectivity in graphs
- …

Next Week Topics

- No Lecture – Exam week
- The week later: Minimum Spanning Trees (Chapter 23)