



Elementary Graph Algorithms

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**: $|E| \ll |V|^2$.
- $|E| = O(|V|^2)$

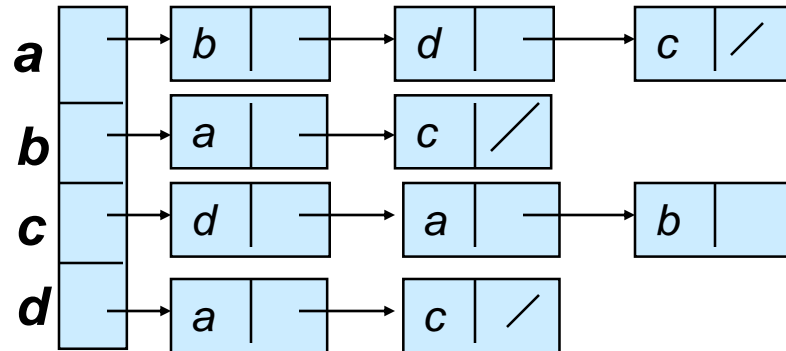
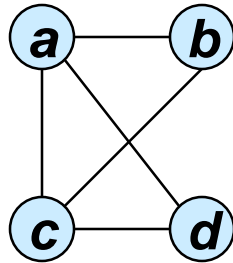
Graphs

- 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.

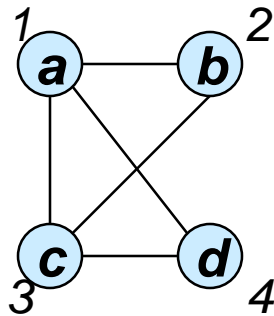
Representation of Graphs

- Two standard ways.

- Adjacency Lists.



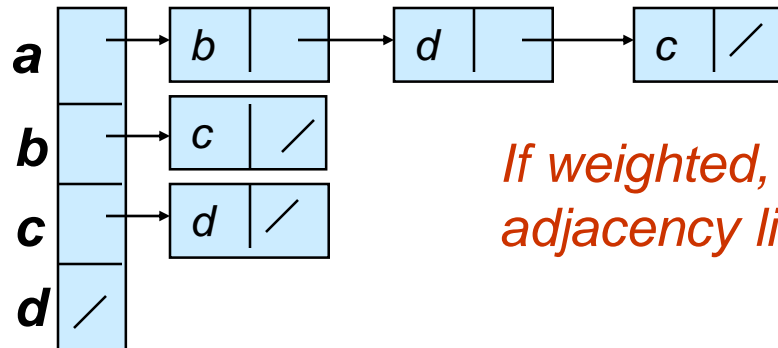
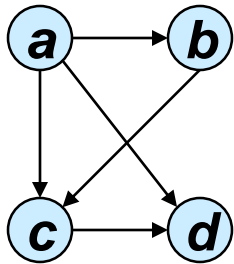
- Adjacency Matrix.



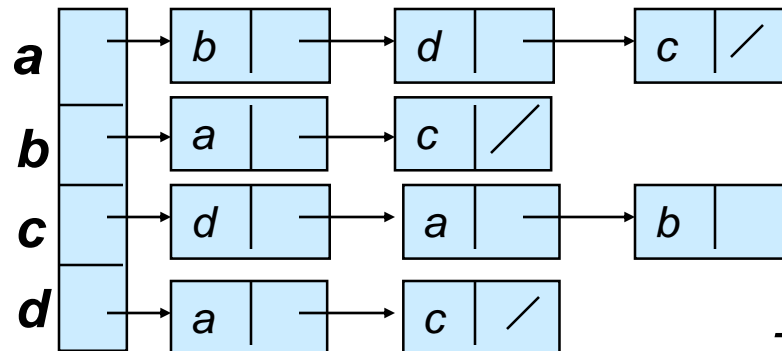
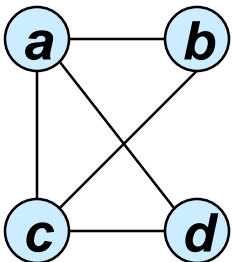
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

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.



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

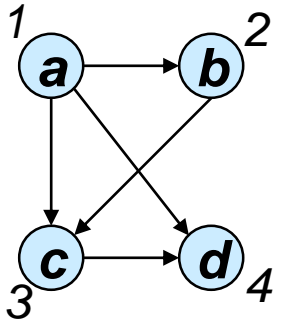
Pros and Cons: adj list

- 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.

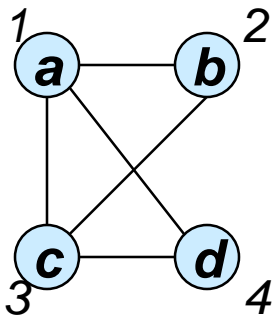
Adjacency Matrix

- $|V| \times |V|$ 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

$A = A^T$ 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.
- Advantages:
 - Simpler, preferred for graphs that are reasonably small.
 - Only one bit per entry for unweighted graphs

Graph Searching

- 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

Breadth-First Search

- “Explore” a graph, turning it into a tree
 - One vertex at a time
 - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
 - Pick a *source vertex* to be the root
 - Find (“discover”) its children, then their children, etc.

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 $s \rightsquigarrow v$.
 - 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, \dots, 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 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.
 - All vertices start out white
 - **Gray** – Discovered but not finished/fully explored.
 - Adjacent to white vertices
 - **Black** – Discovered and Finished/fully explored.
 - Colors are required only to reason about the algorithm. Can be implemented without colors.
- Explore vertices by scanning adjacency list of grey vertices

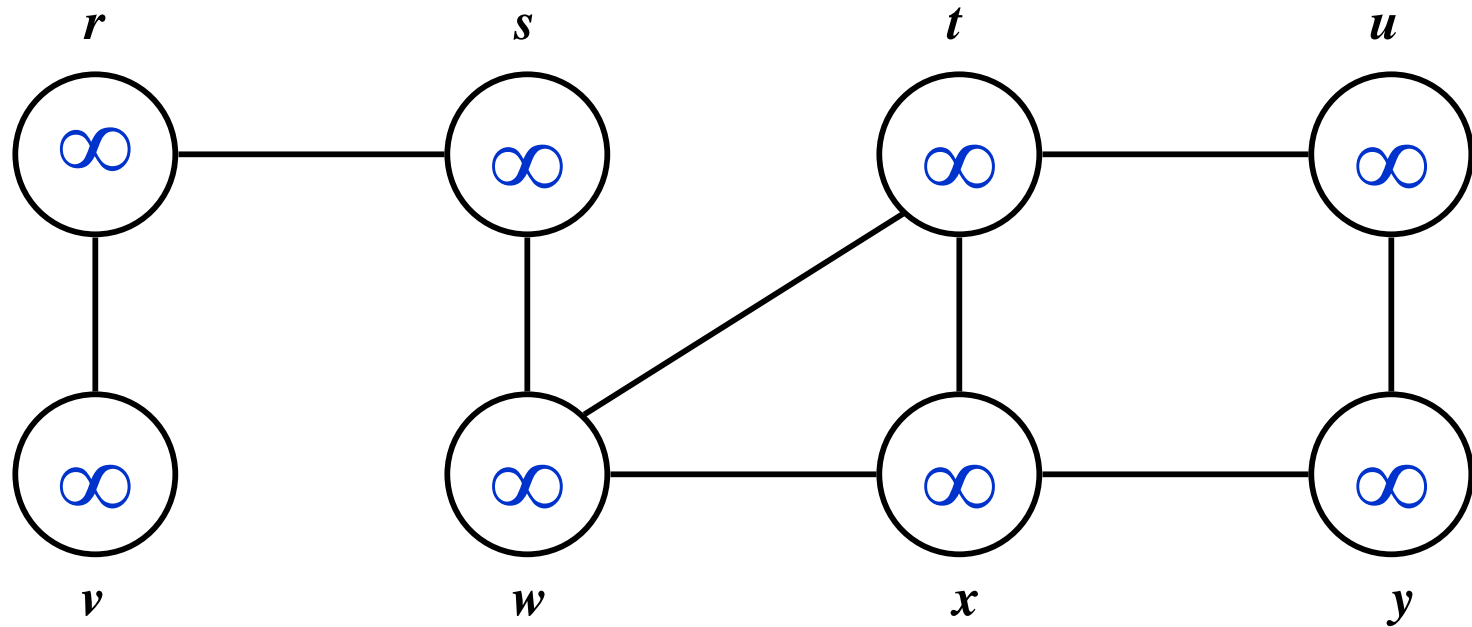
BFS(G,s)

```
1. for each vertex  $u$  in  $V[G] - \{s\}$ 
2     do  $color[u] \leftarrow \text{white}$ 
3        $d[u] \leftarrow \infty$ 
4        $\pi[u] \leftarrow \text{nil}$ 
5  $color[s] \leftarrow \text{gray}$ 
6  $d[s] \leftarrow 0$ 
7  $\pi[s] \leftarrow \text{nil}$ 
8  $Q \leftarrow \Phi$ 
9  $\text{enqueue}(Q,s)$ 
10 while  $Q \neq \Phi$ 
11     do  $u \leftarrow \text{dequeue}(Q)$ 
12       for each  $v$  in  $\text{Adj}[u]$ 
13         do if  $color[v] = \text{white}$ 
14           then  $color[v] \leftarrow \text{gray}$ 
15              $d[v] \leftarrow d[u] + 1$ 
16              $\pi[v] \leftarrow u$ 
17              $\text{enqueue}(Q,v)$ 
18        $color[u] \leftarrow \text{black}$ 
```

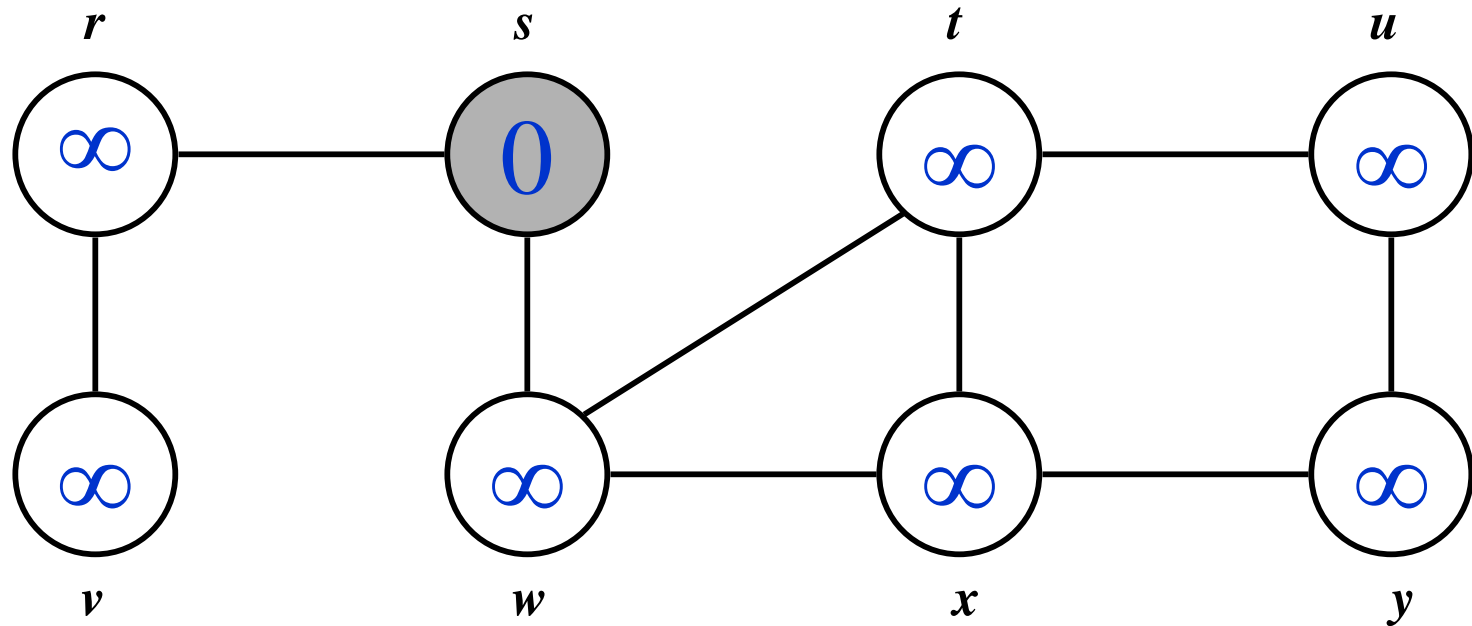
*white: undiscovered
gray: discovered
black: finished*

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

Breadth-First Search: Example

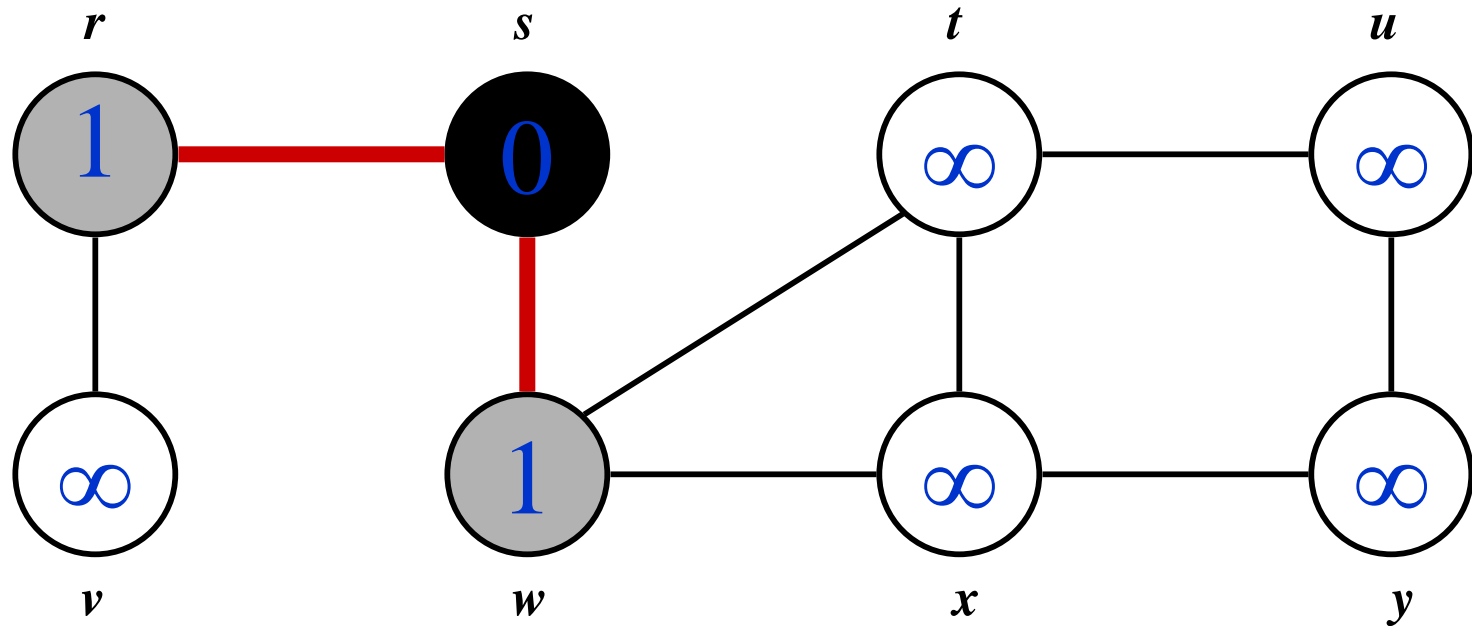


Breadth-First Search: Example



Q : s

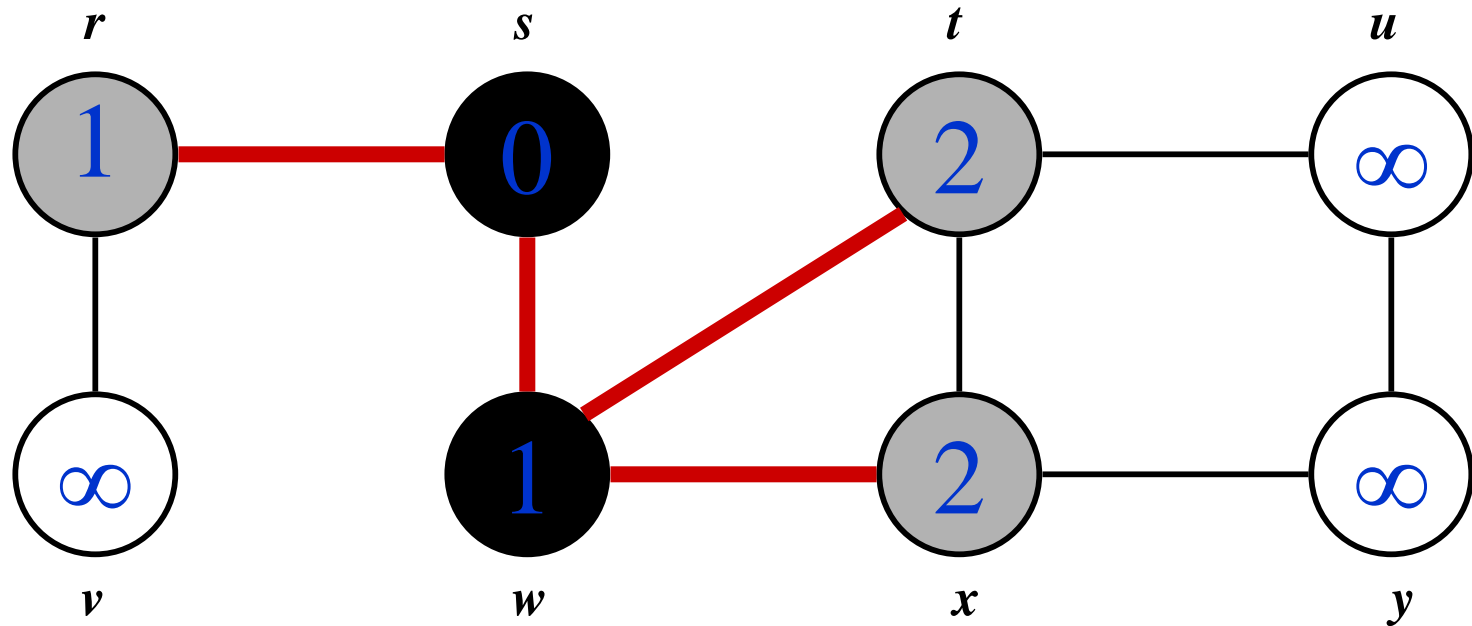
Breadth-First Search: Example



Q :

w	r
-----	-----

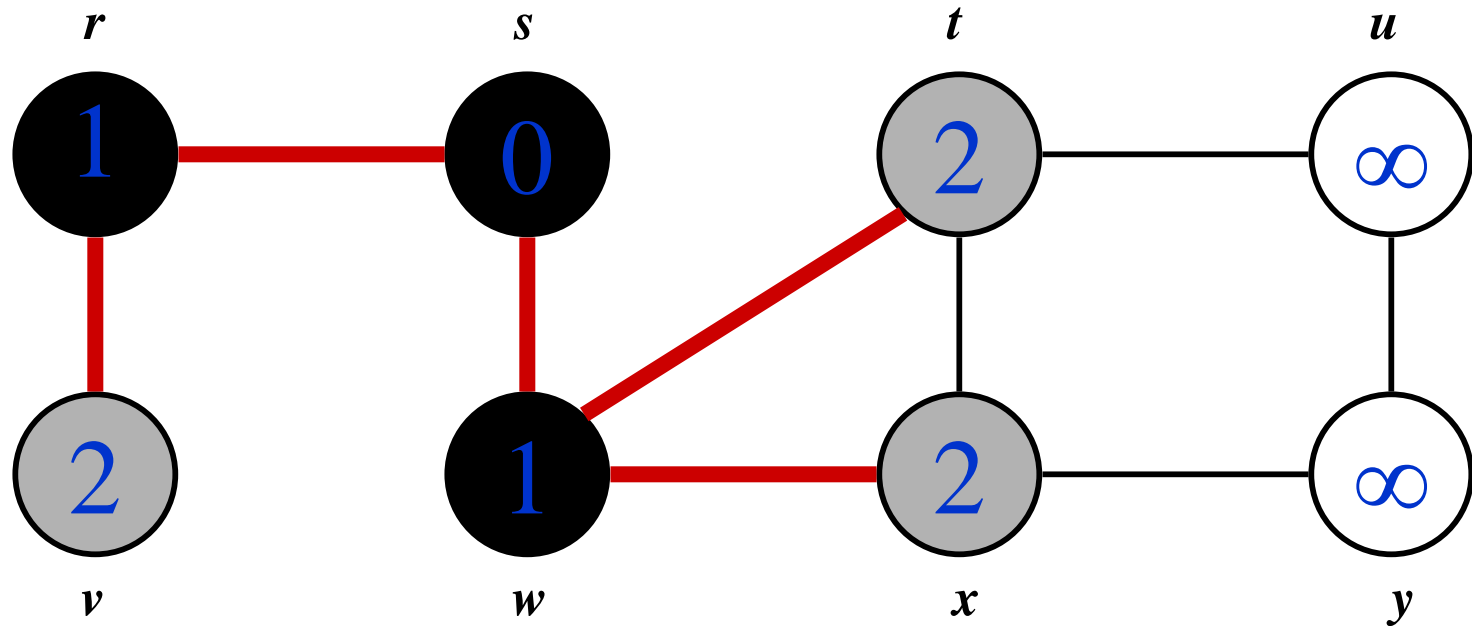
Breadth-First Search: Example



Q :

r	t	x
-----	-----	-----

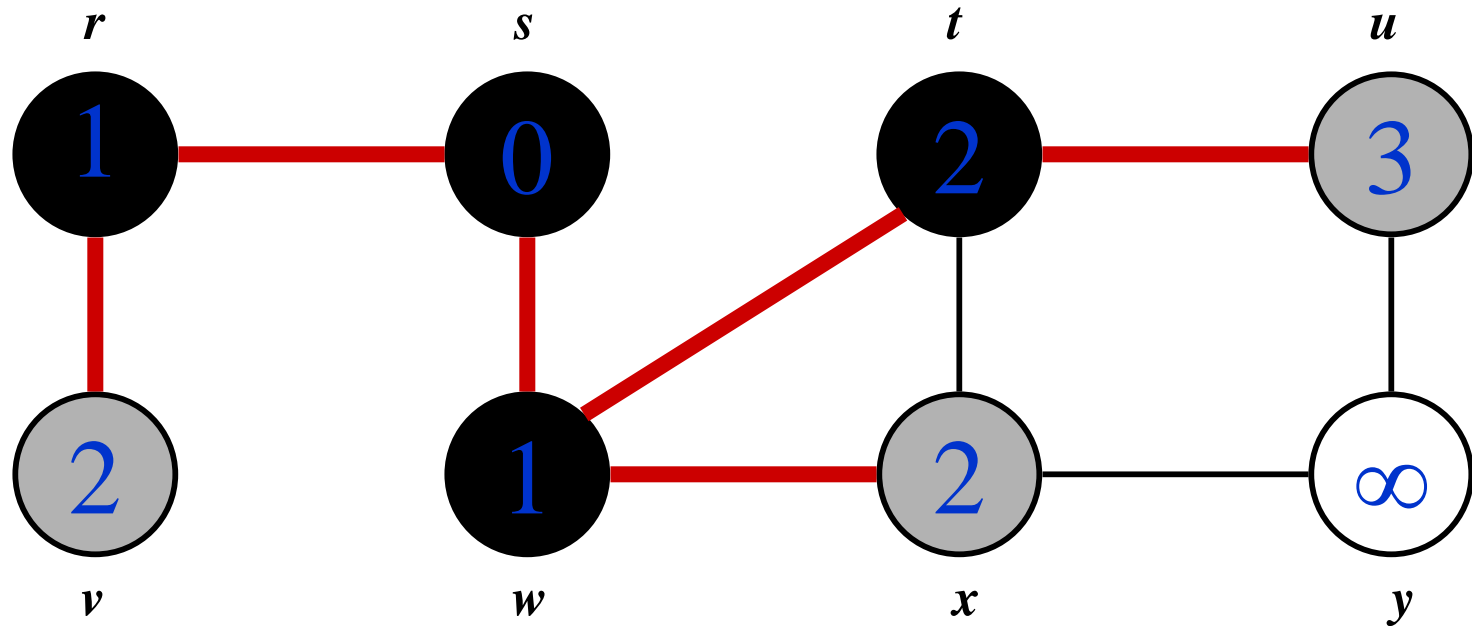
Breadth-First Search: Example



Q :

t	x	v
-----	-----	-----

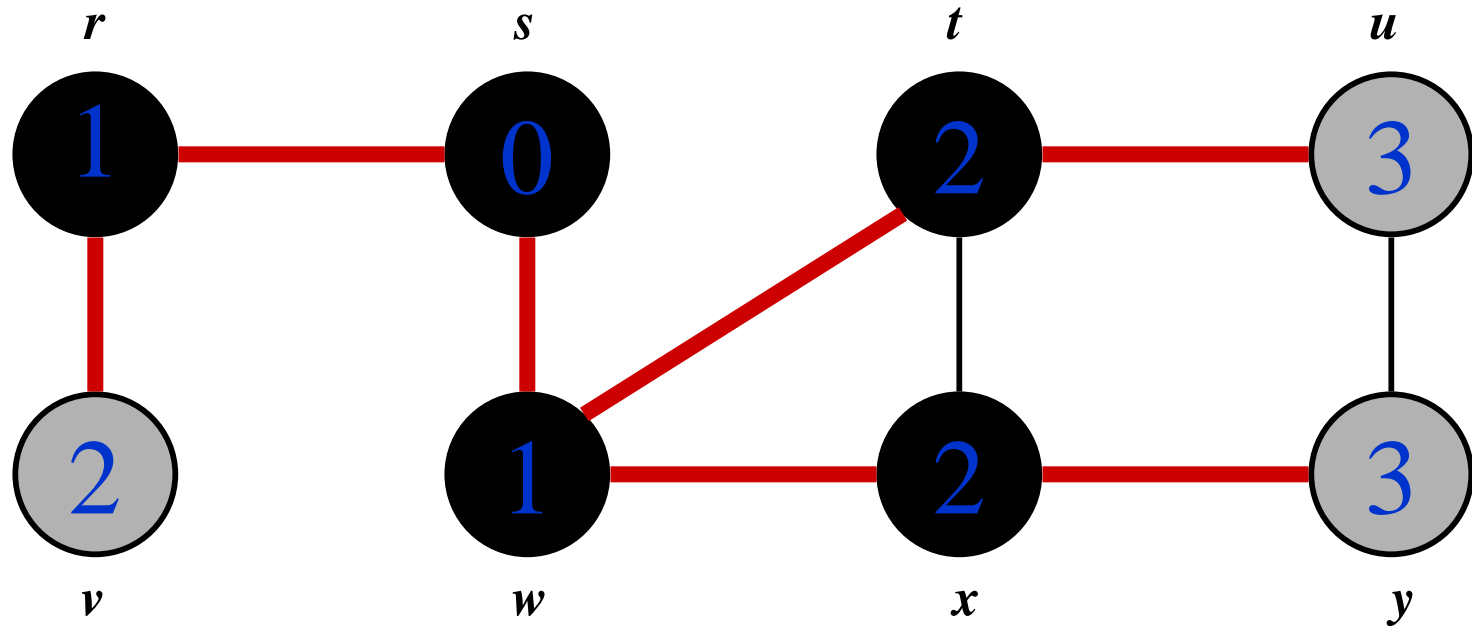
Breadth-First Search: Example



Q :

x	v	u
-----	-----	-----

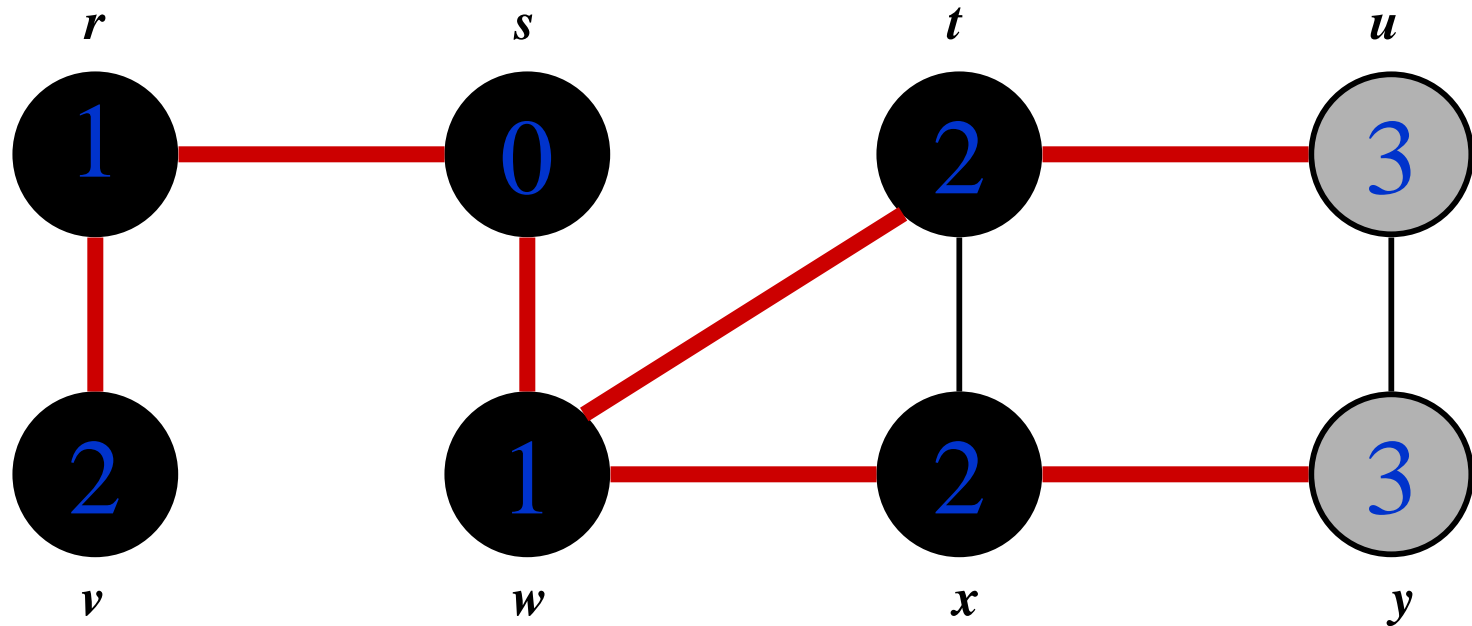
Breadth-First Search: Example



Q :

v	u	y
-----	-----	-----

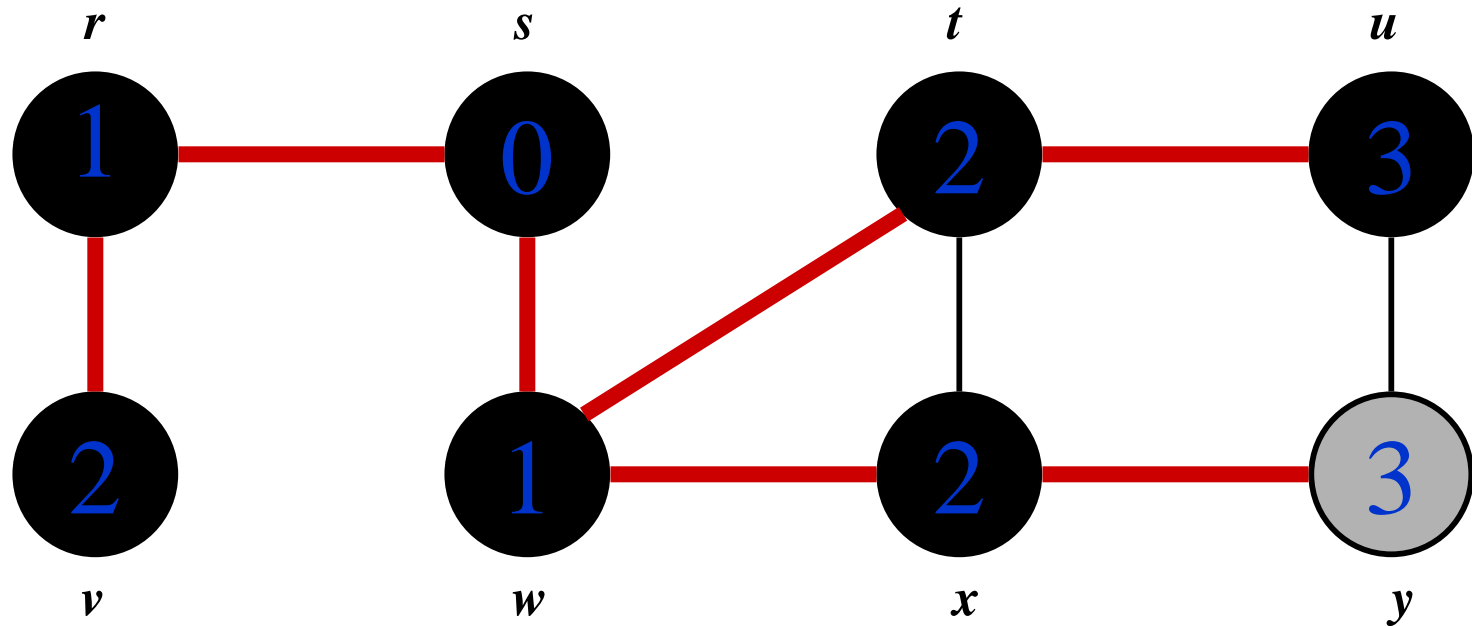
Breadth-First Search: Example



Q :

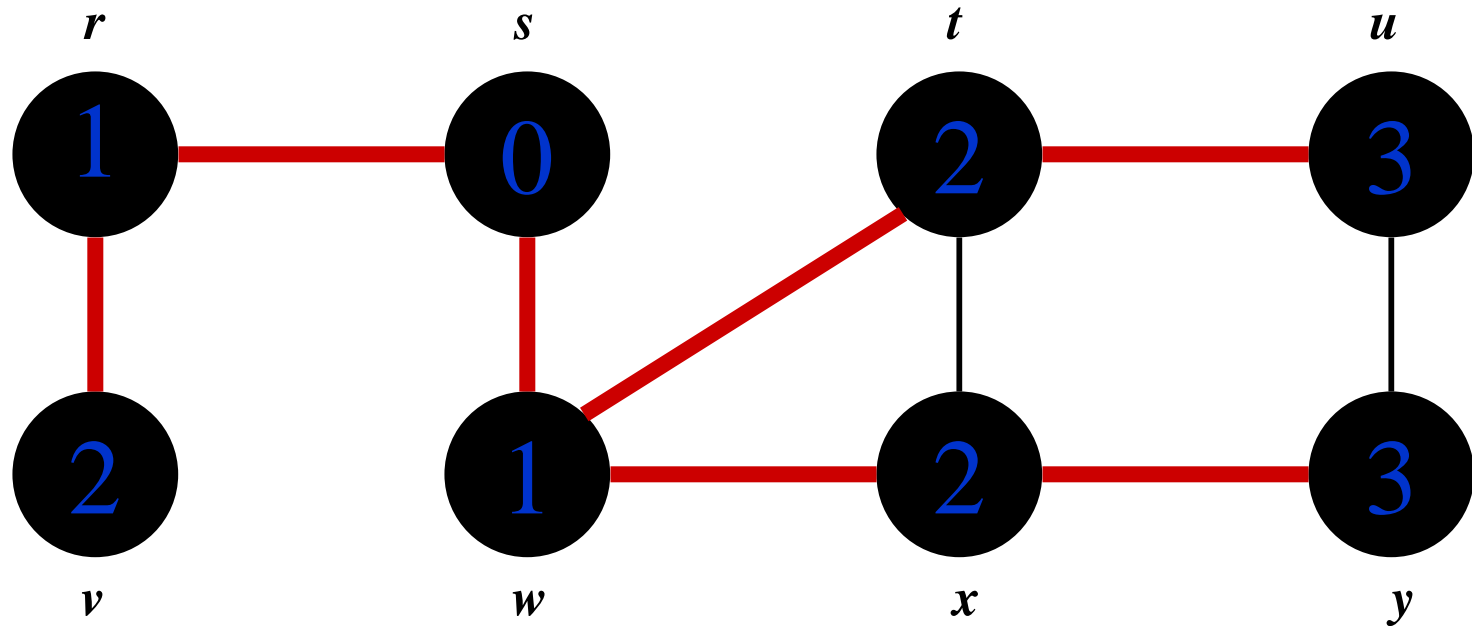
u	y
-----	-----

Breadth-First Search: Example



Q : y

Breadth-First Search: Example



$Q: \emptyset$

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 \Rightarrow total running time of BFS is $O(V+E)$, linear in the size of the adjacency list representation of graph.

Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* to the source node
 - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v , or ∞ if v not reachable from s
- BFS builds *breadth-first tree*, in which paths to root represent shortest paths in G
 - Thus can use BFS to calculate shortest path from one vertex to another in $O(V+E)$ time

Depth-First Search

- *Depth-first search* is another strategy for exploring a graph
 - Explore “deeper” in the graph whenever possible
 - Edges are explored out of the most recently discovered vertex v that still has unexplored edges
 - When all of v 's edges have been explored, backtrack to the vertex from which v was discovered

Depth-First Search

- Vertices initially colored white
- Then colored gray when discovered
- Then black when finished

Pseudo-code

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)

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. $f[u] \leftarrow time \leftarrow time + 1$

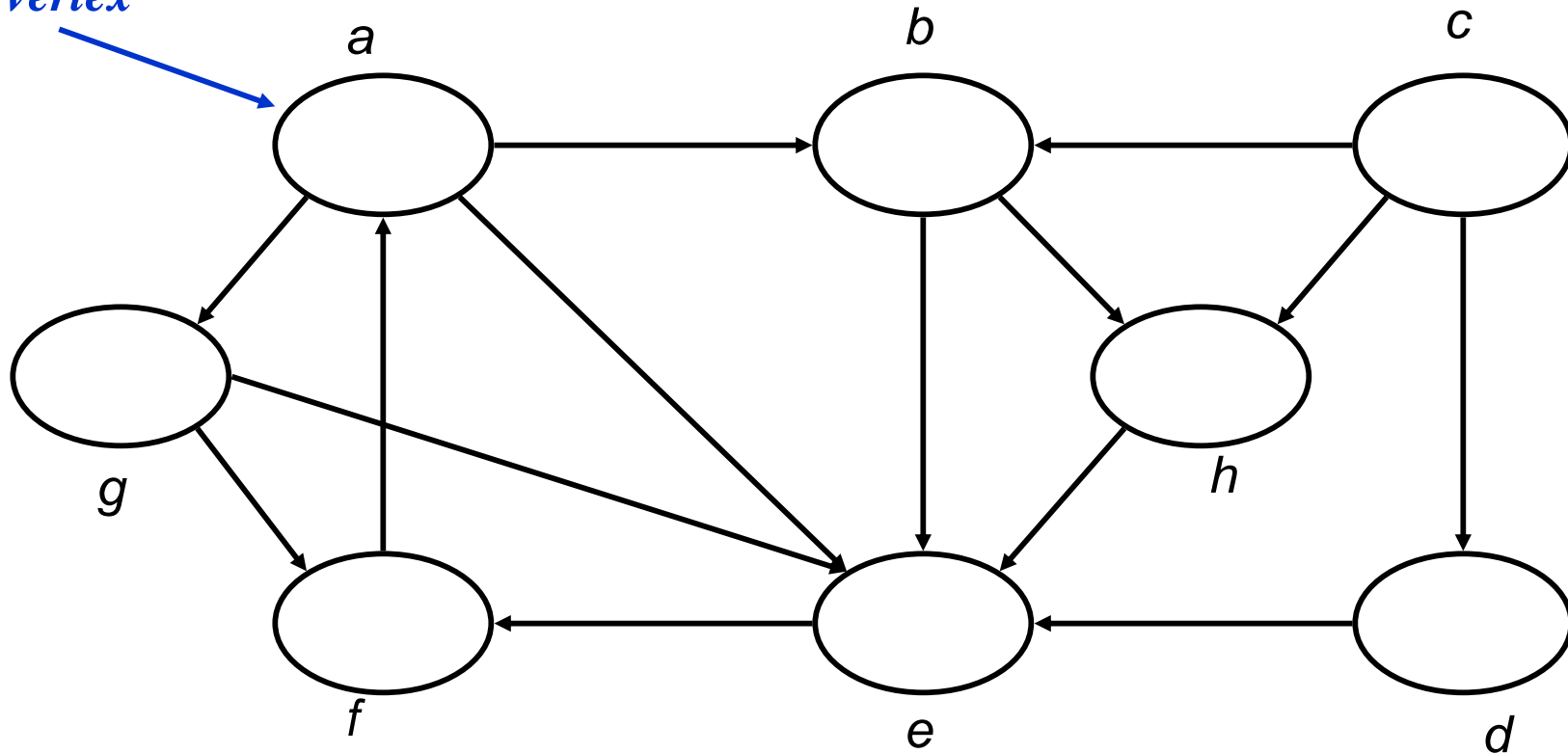
Uses a global timestamp **time**.

Depth-First Sort Analysis

- “Charge” the exploration of edge to the edge:
 - Each loop in DFS_Visit can be attributed to an edge in the graph
 - Thus loop will run in $O(E)$ time, algorithm $O(V+E)$

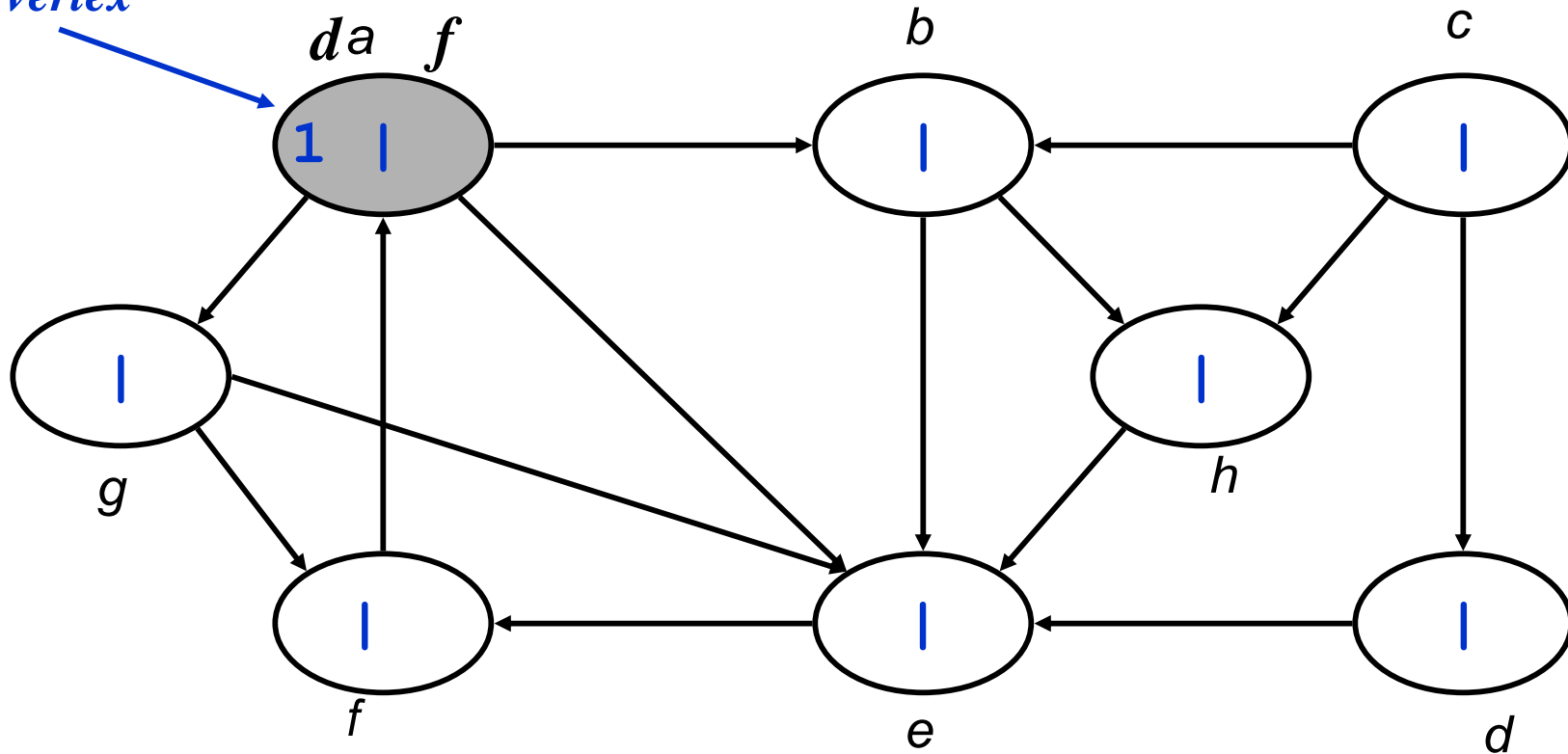
DFS Example

*source
vertex*



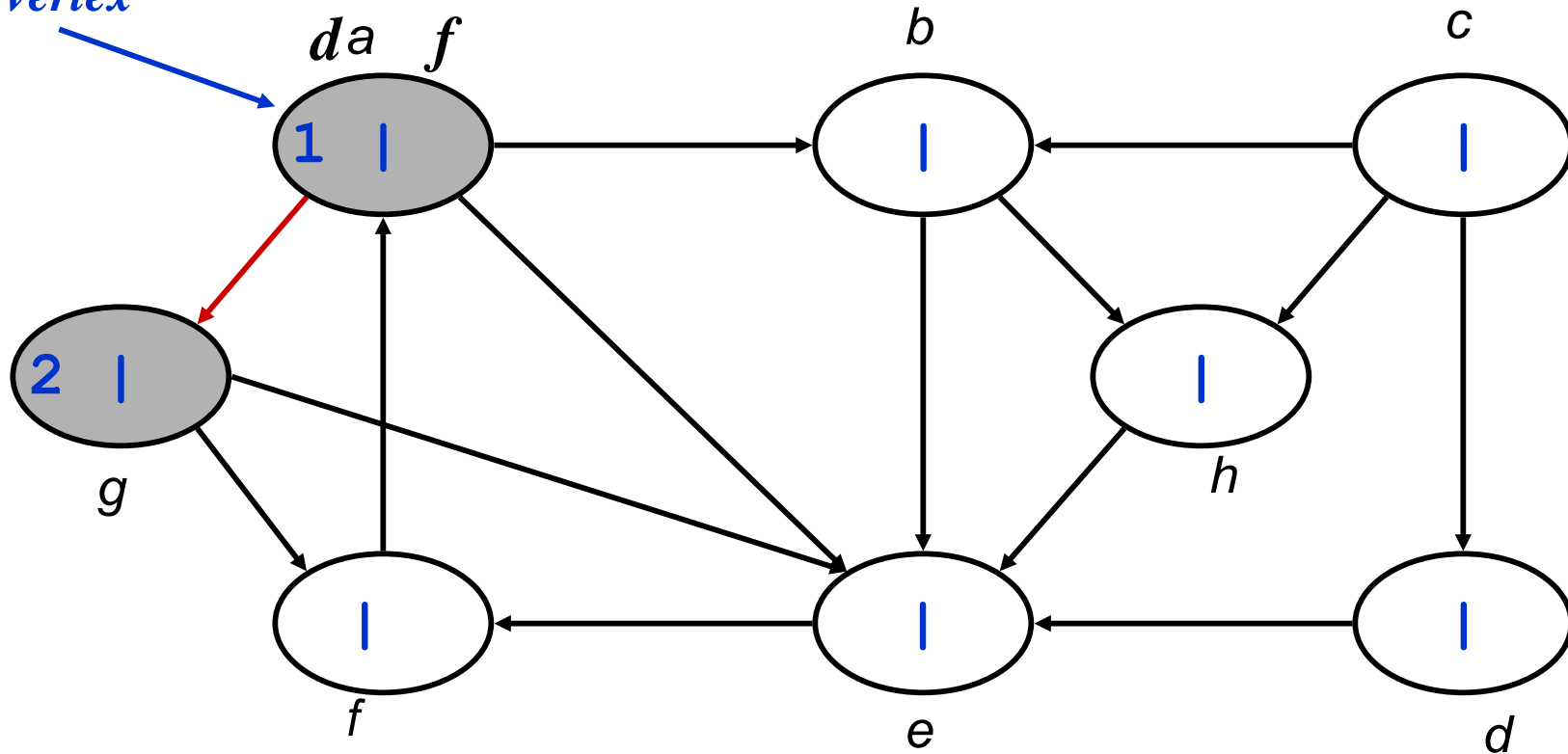
DFS Example

*source
vertex*



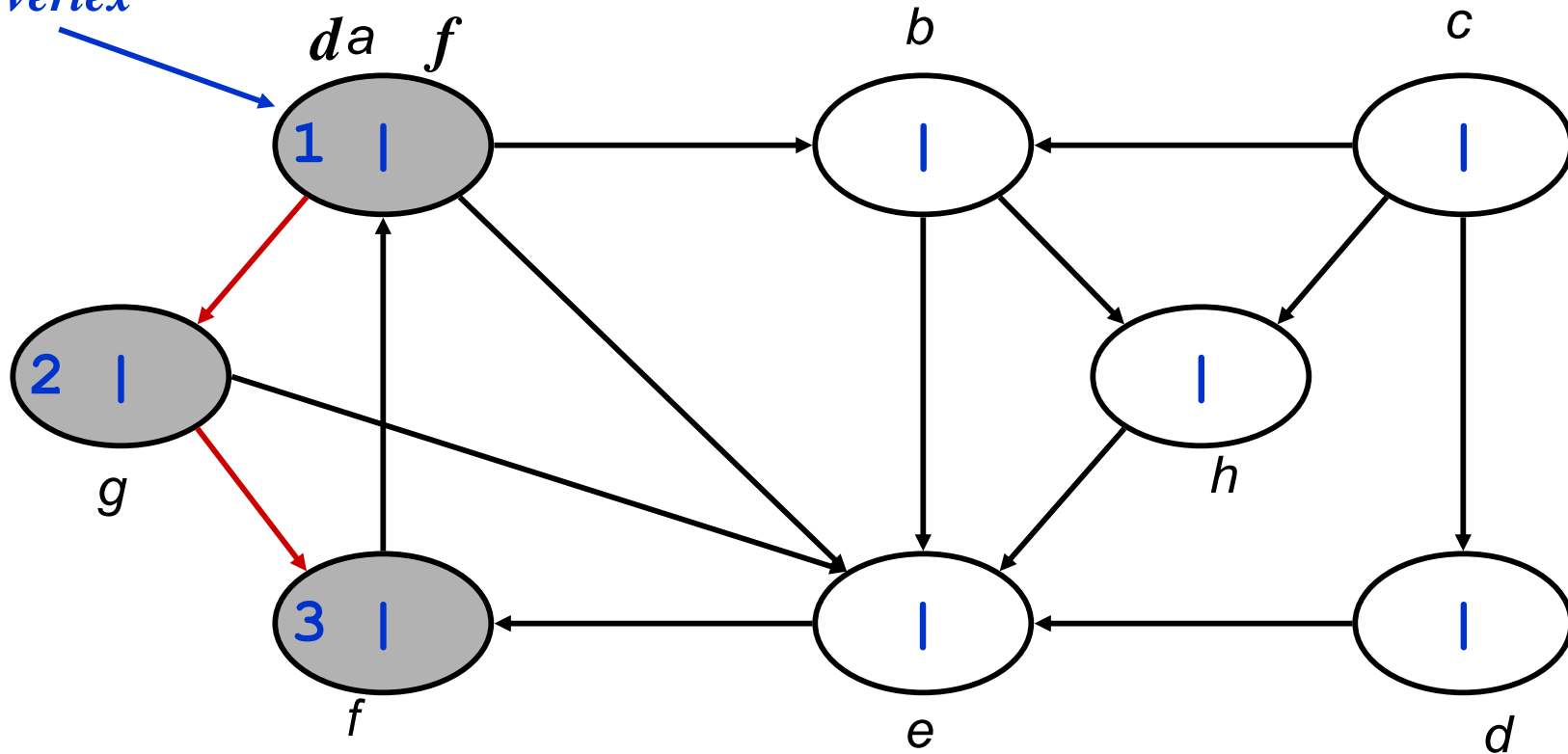
DFS Example

*source
vertex*



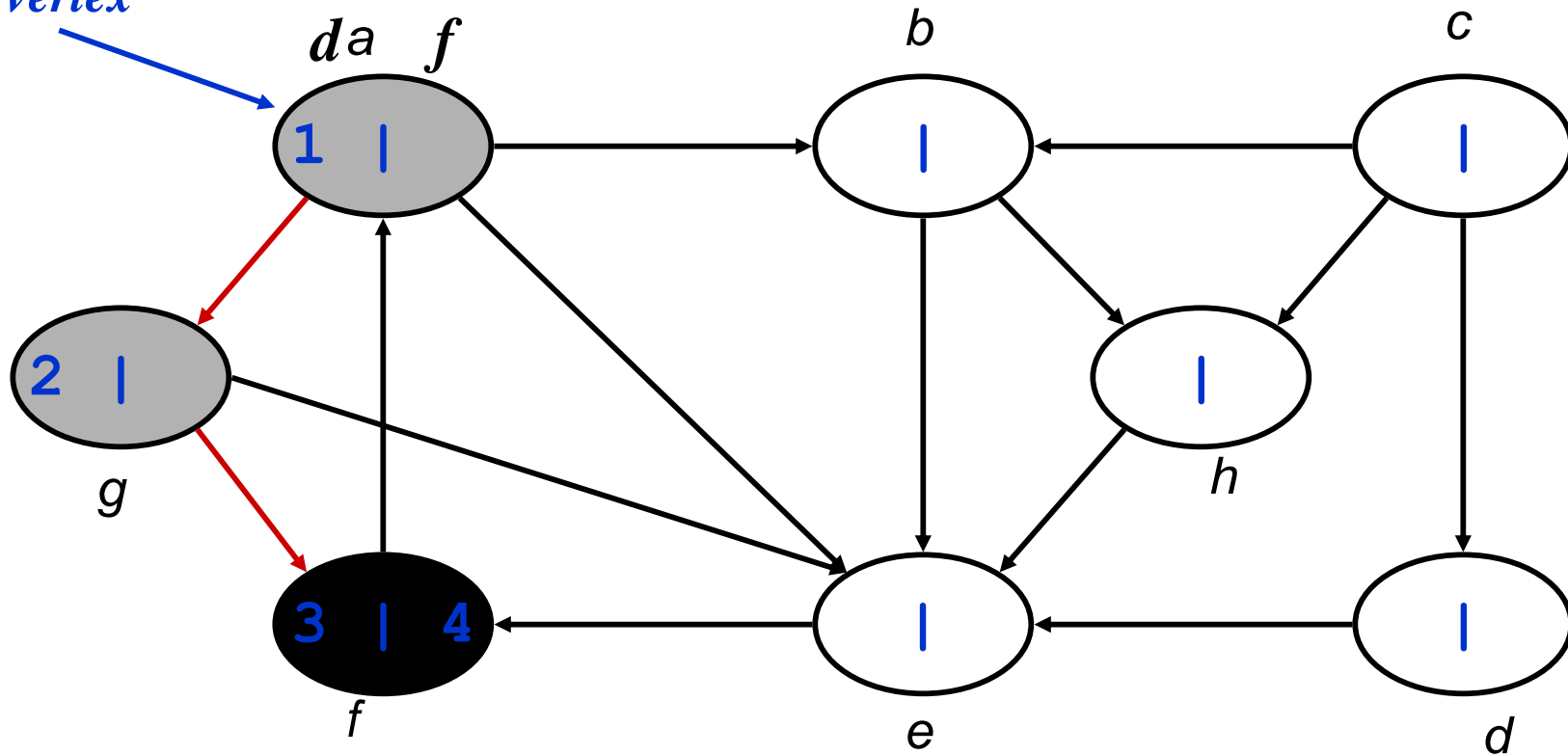
DFS Example

*source
vertex*



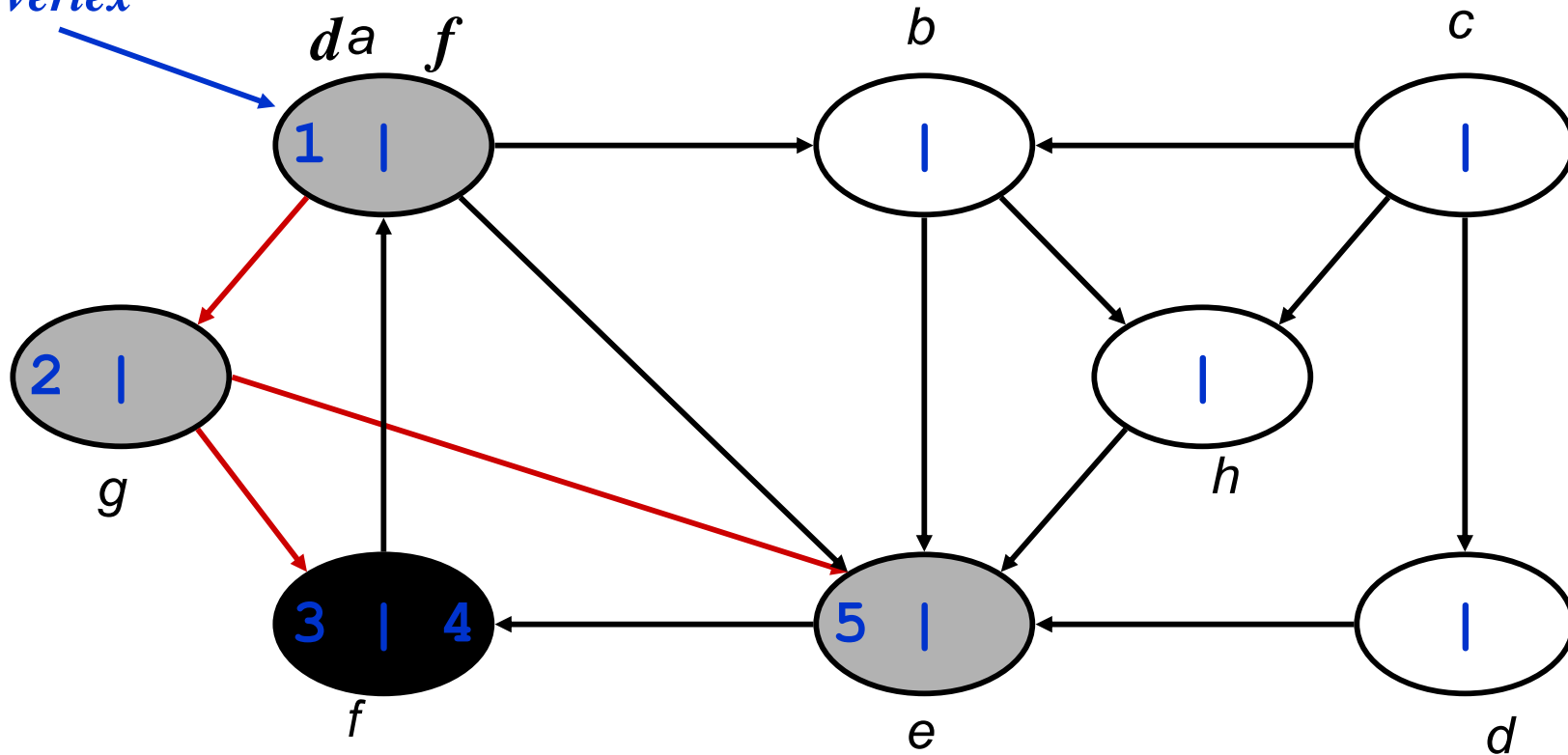
DFS Example

*source
vertex*



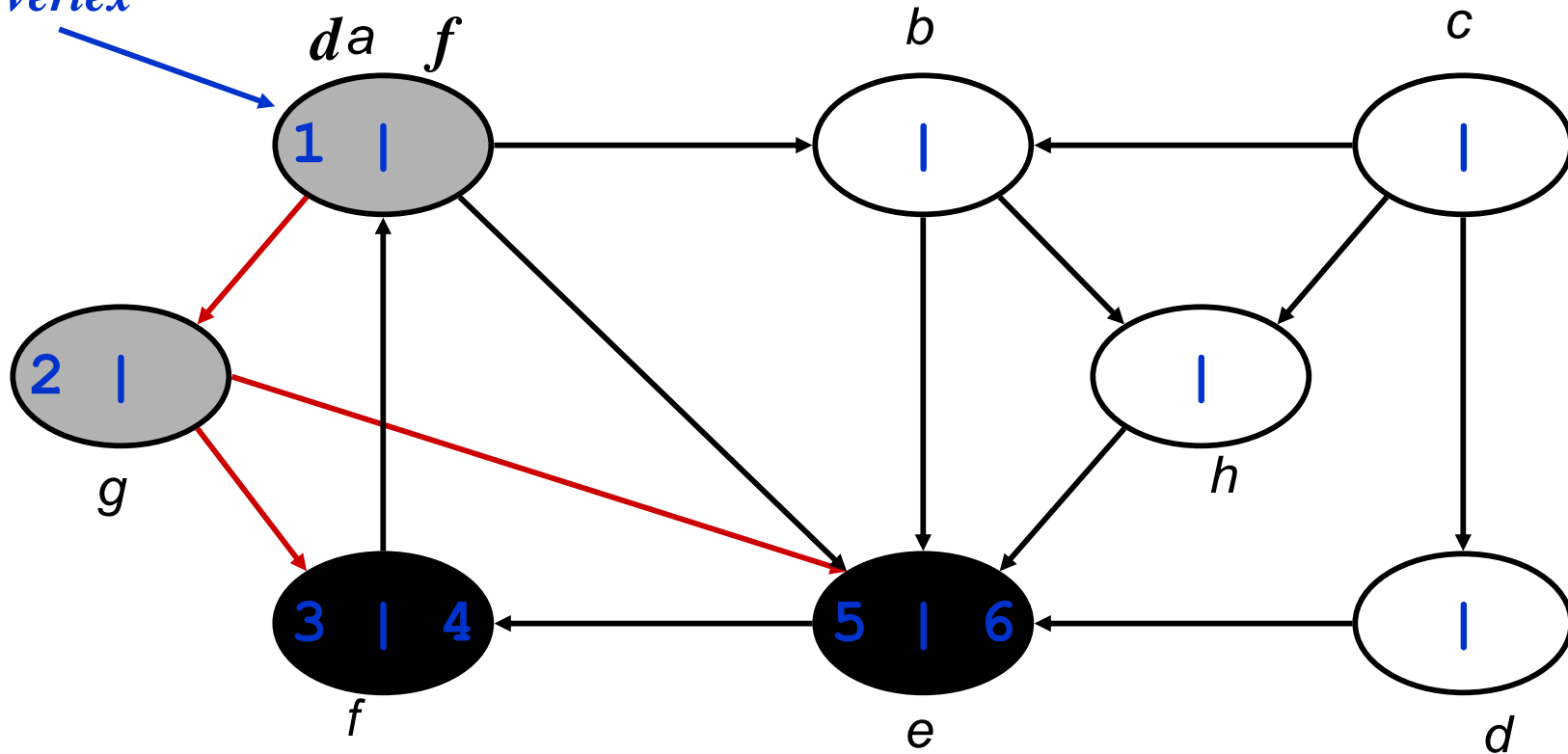
DFS Example

*source
vertex*



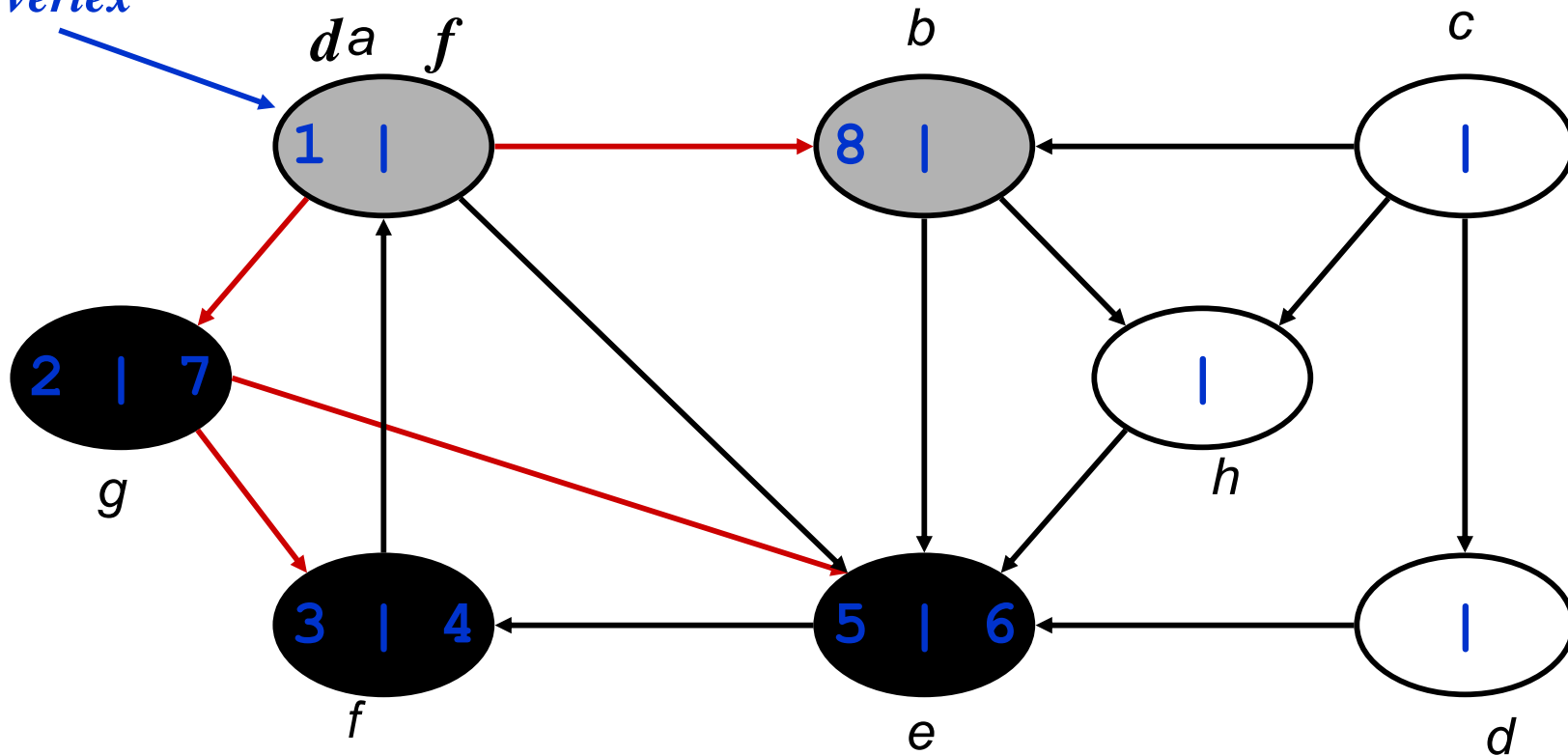
DFS Example

*source
vertex*



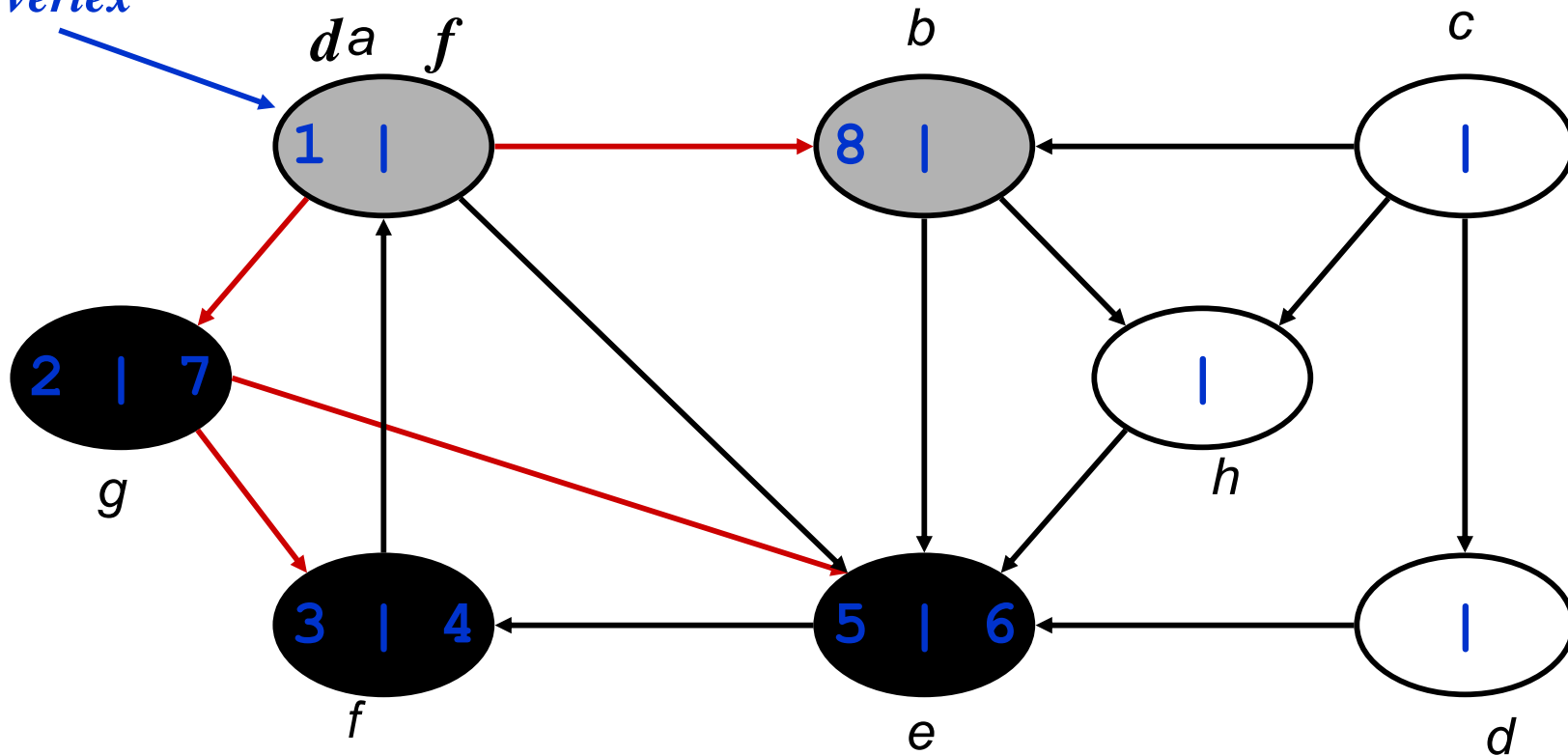
DFS Example

*source
vertex*



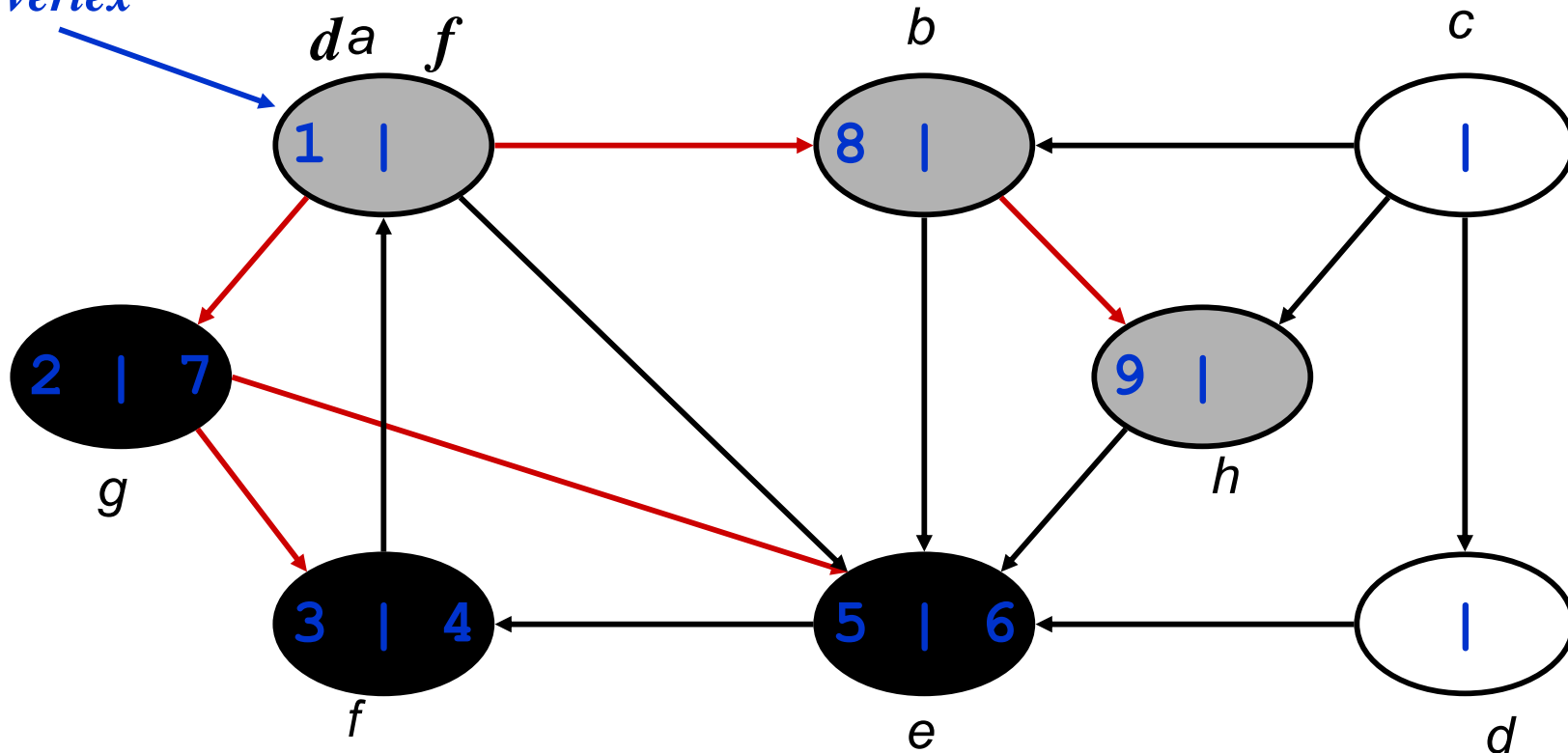
DFS Example

*source
vertex*



DFS Example

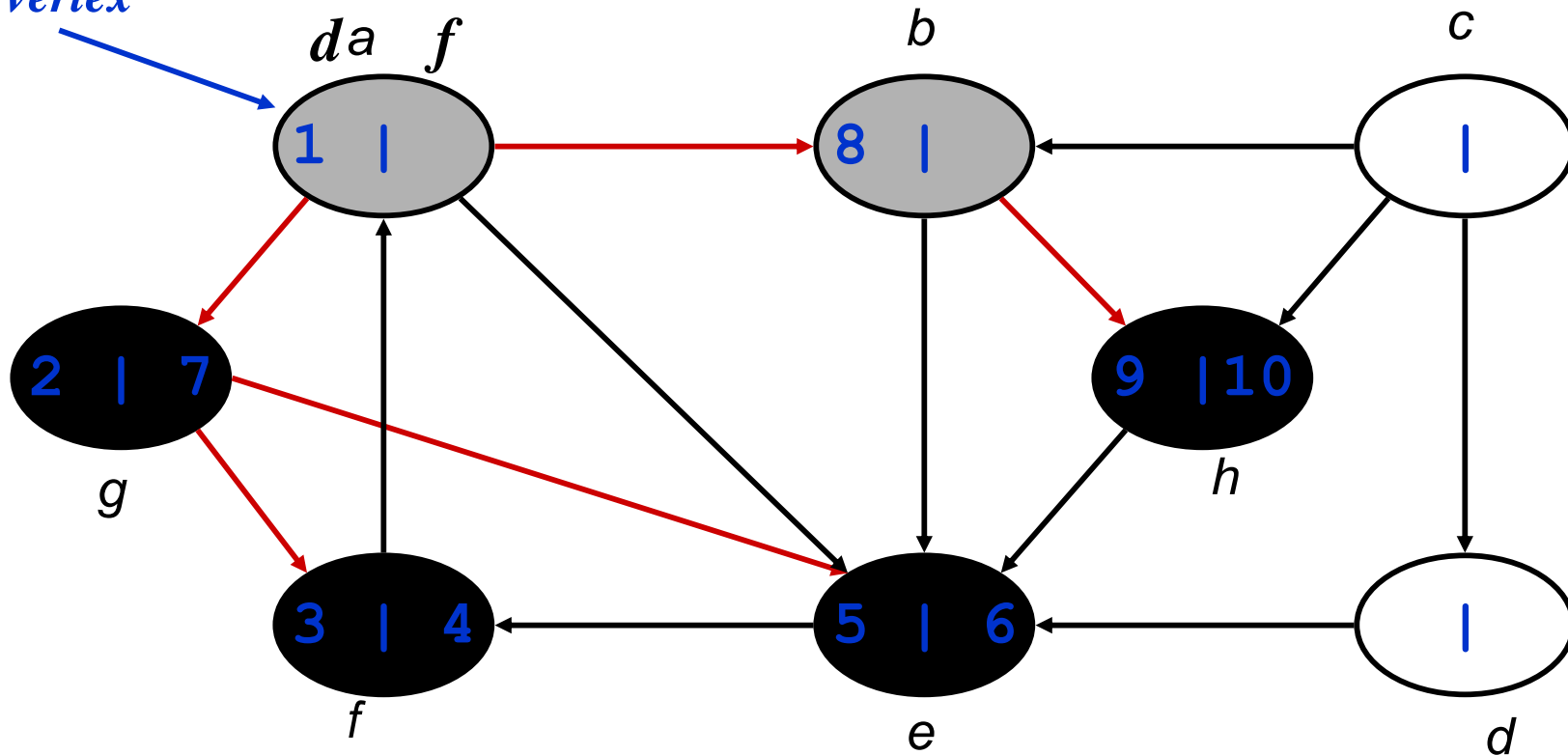
*source
vertex*



*What is the structure of the grey vertices?
What do they represent?*

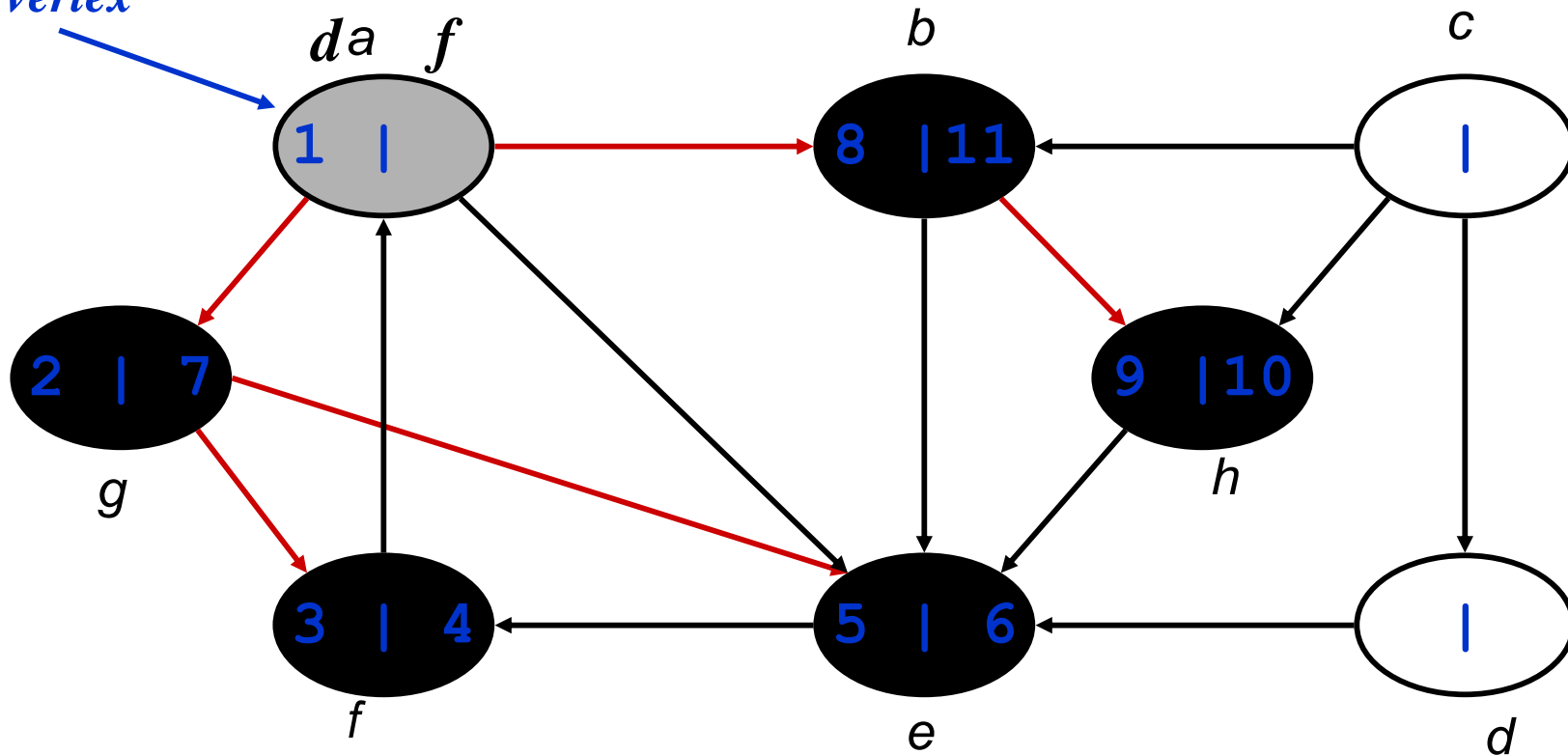
DFS Example

*source
vertex*



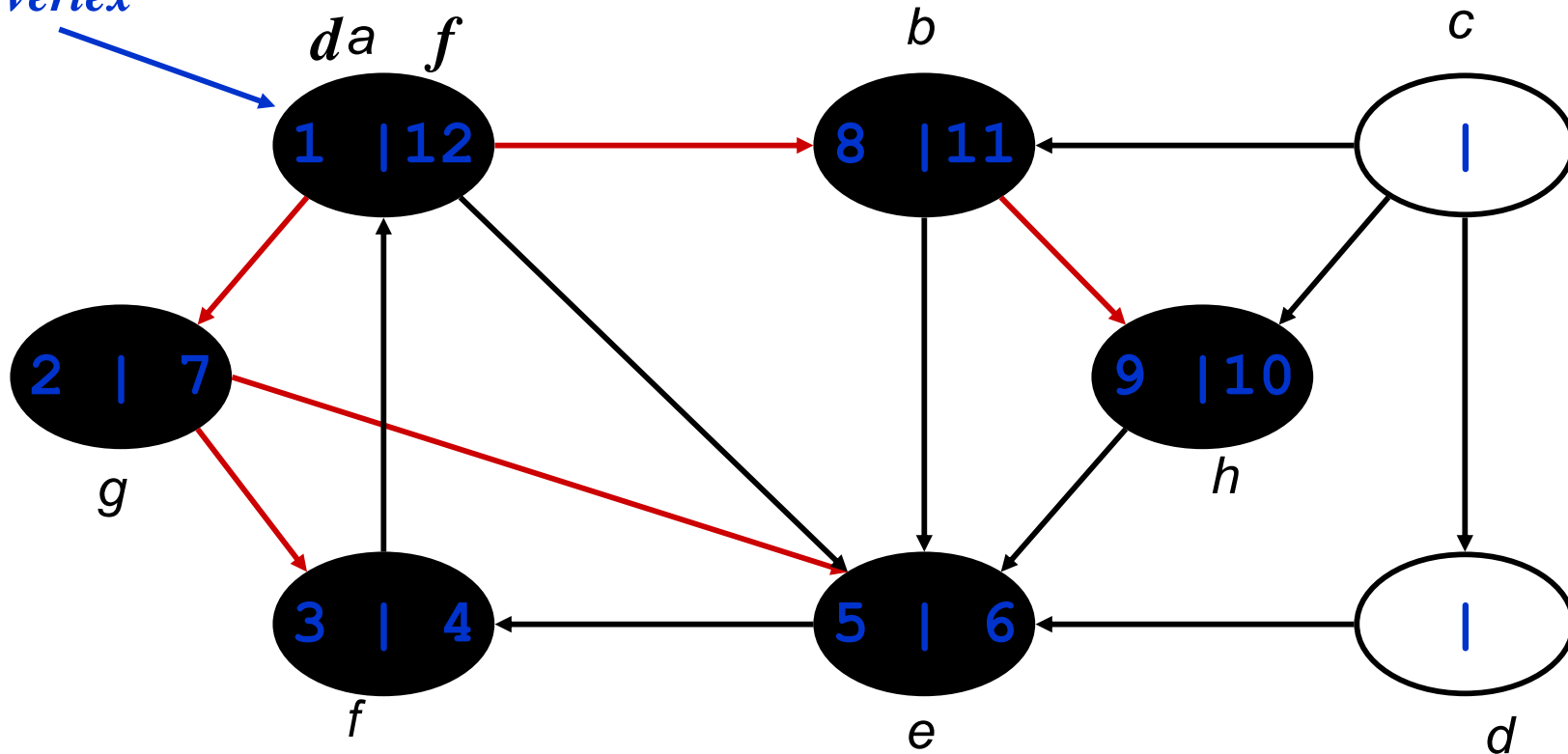
DFS Example

*source
vertex*



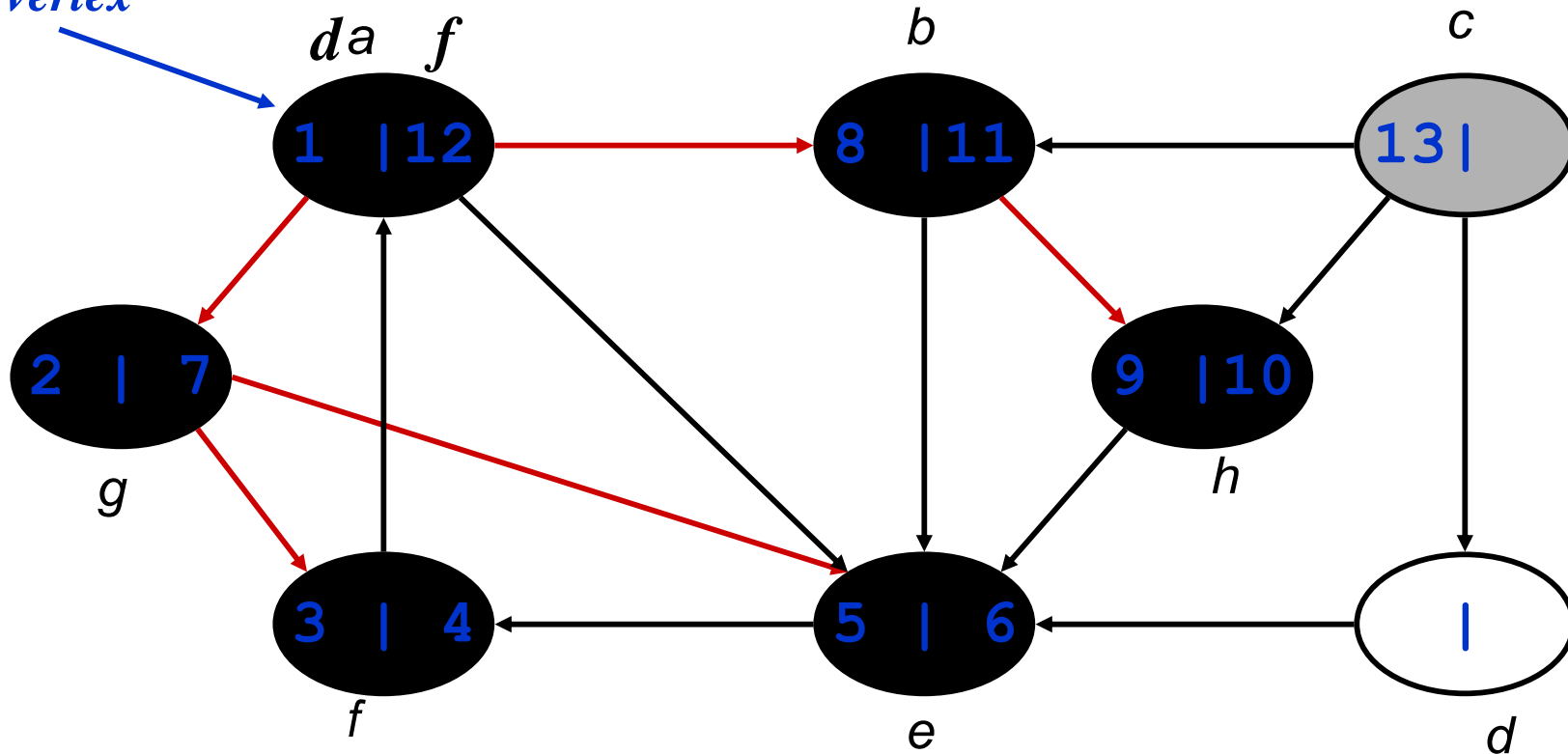
DFS Example

*source
vertex*



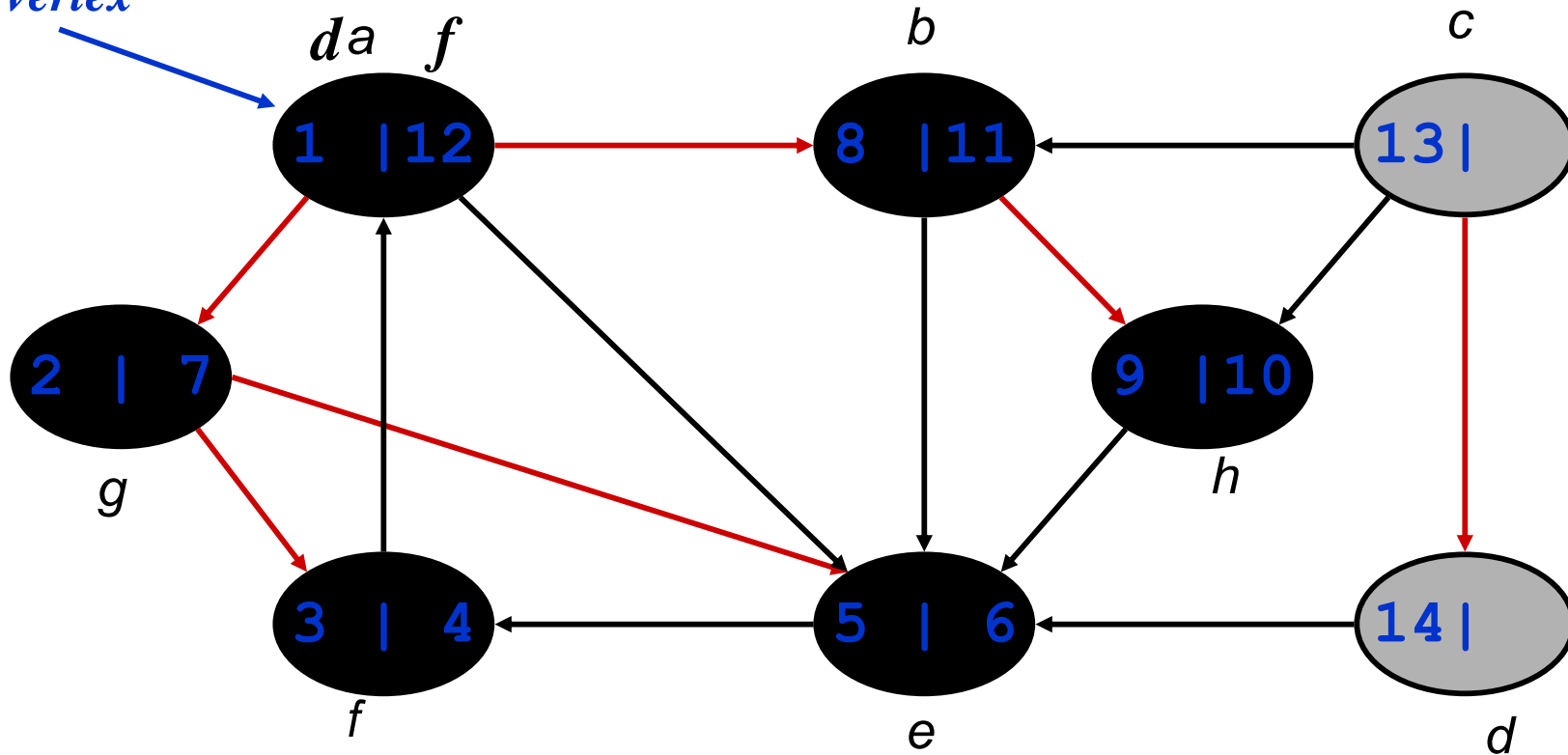
DFS Example

*source
vertex*



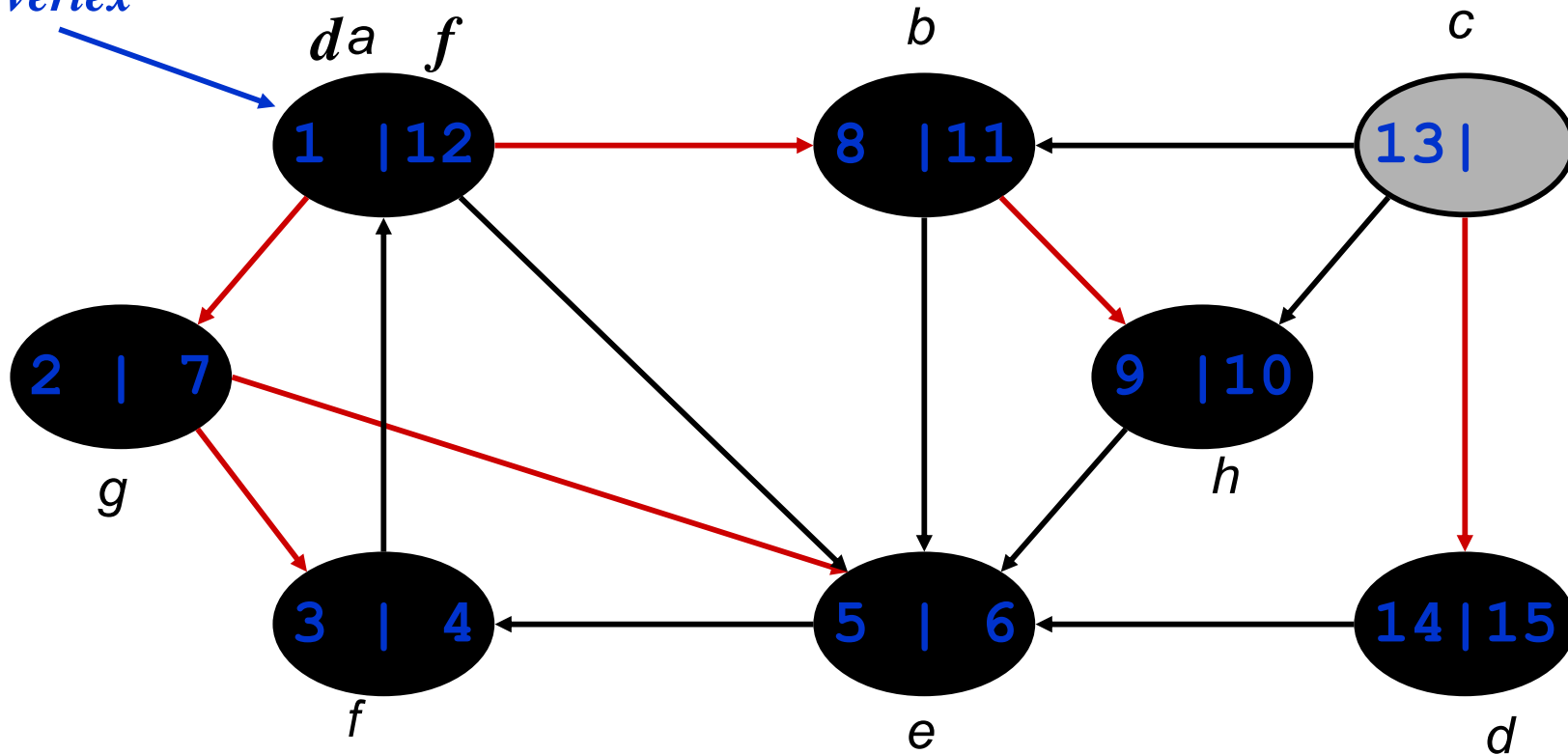
DFS Example

*source
vertex*



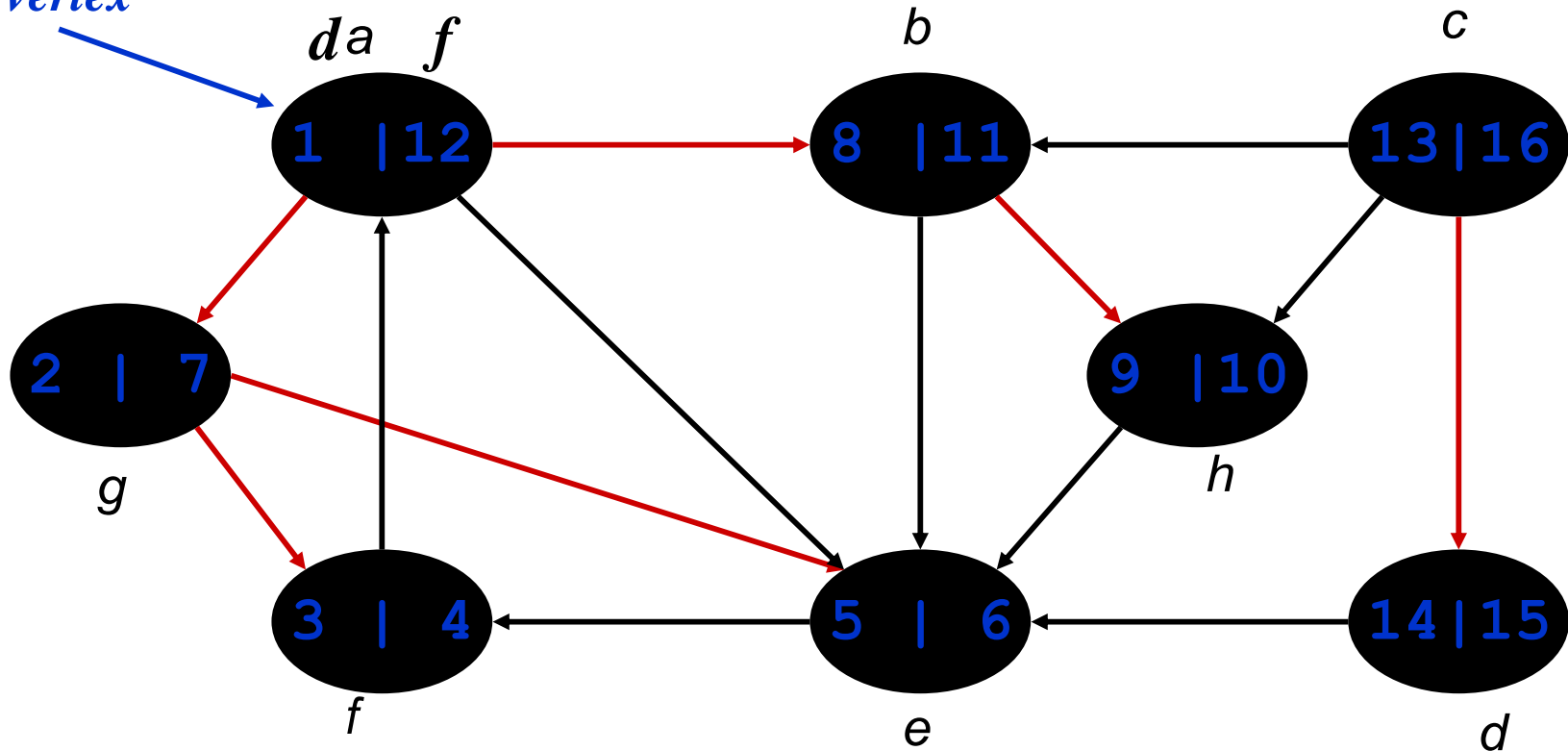
DFS Example

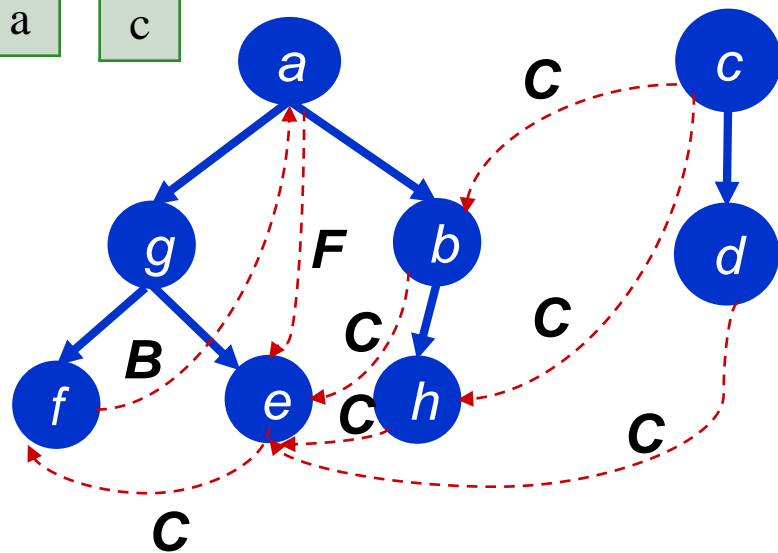
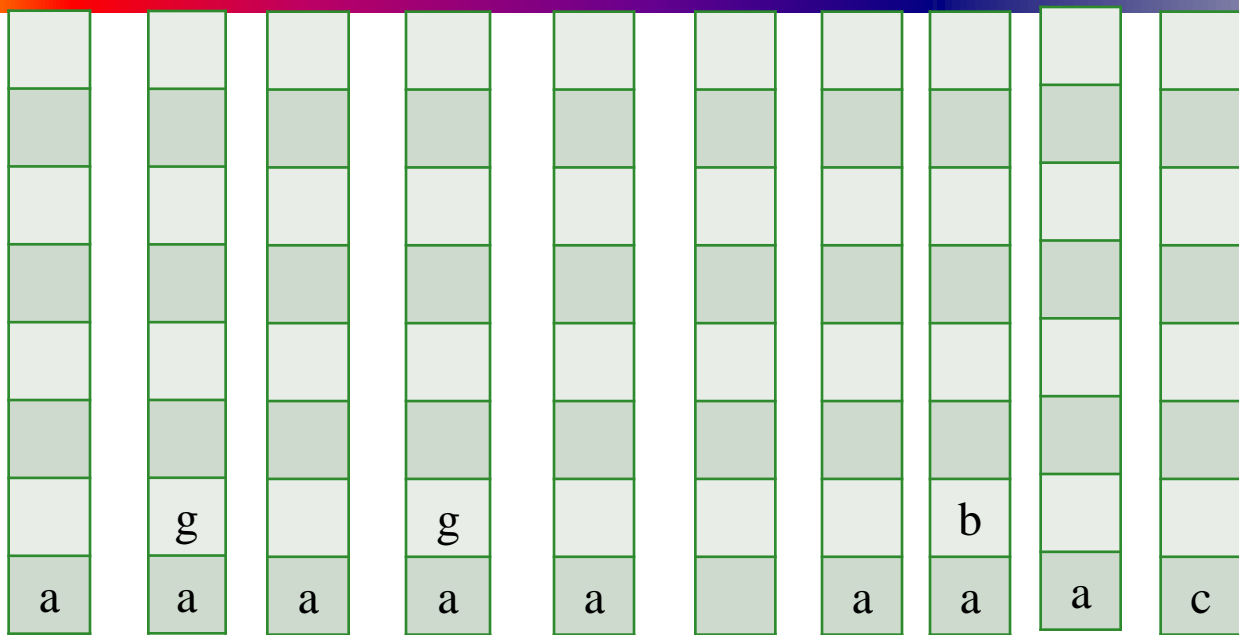
*source
vertex*



DFS Example

*source
vertex*





$a \quad g \quad f \quad e \quad b \quad h \quad c \quad d$

Pseudo-code

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)

DFS-Visit(u)

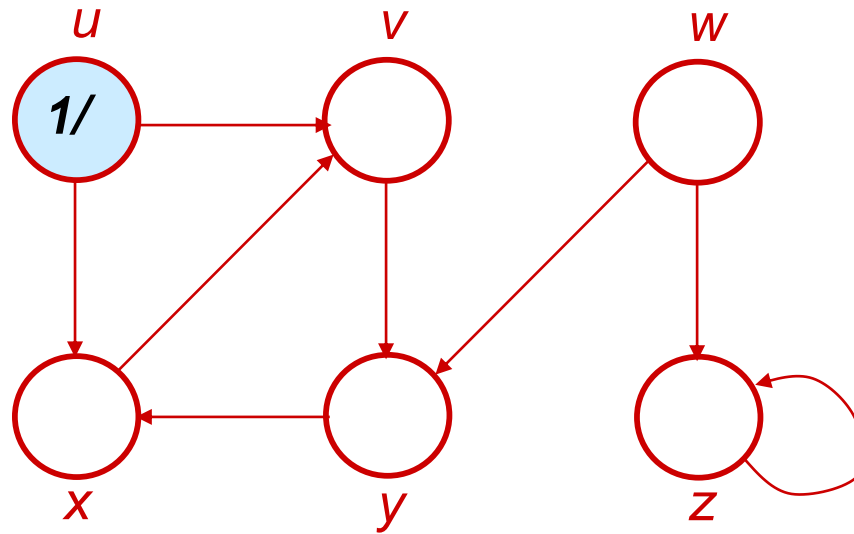
1. $color[u] \leftarrow \text{GRAY}$ ∇ White vertex u has been discovered
2. $time \leftarrow time + 1$
3. $d[u] \leftarrow time$
4. **for** each $v \in Adj[u]$
5. **do if** $color[v] = \text{WHITE}$
6. **then** $\pi[v] \leftarrow u$
7. DFS-Visit(v)
8. $color[u] \leftarrow \text{BLACK}$ ∇ Blacken u ;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

Uses a global timestamp **time**.

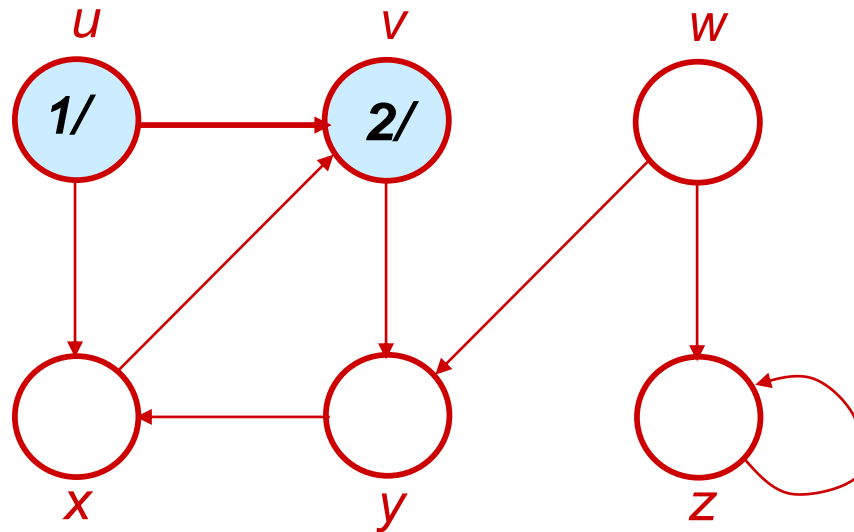
Analysis of DFS

- 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 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 $\Theta(V+E)$.

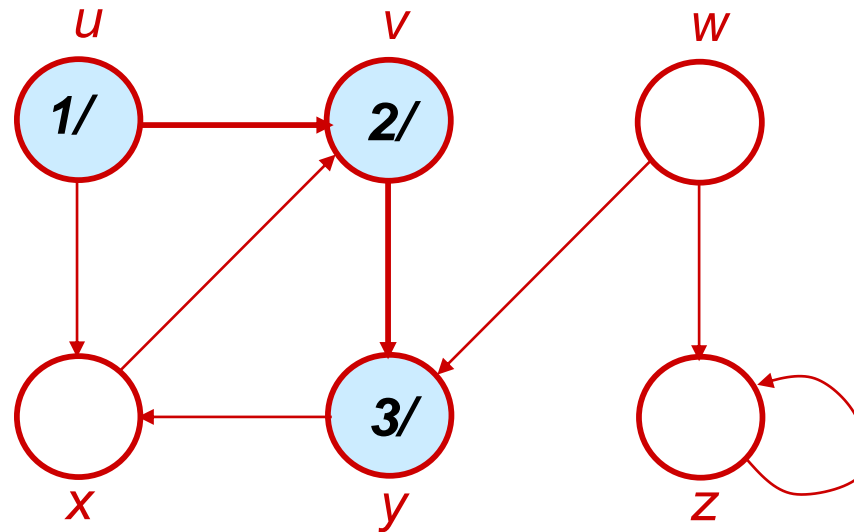
Example 2



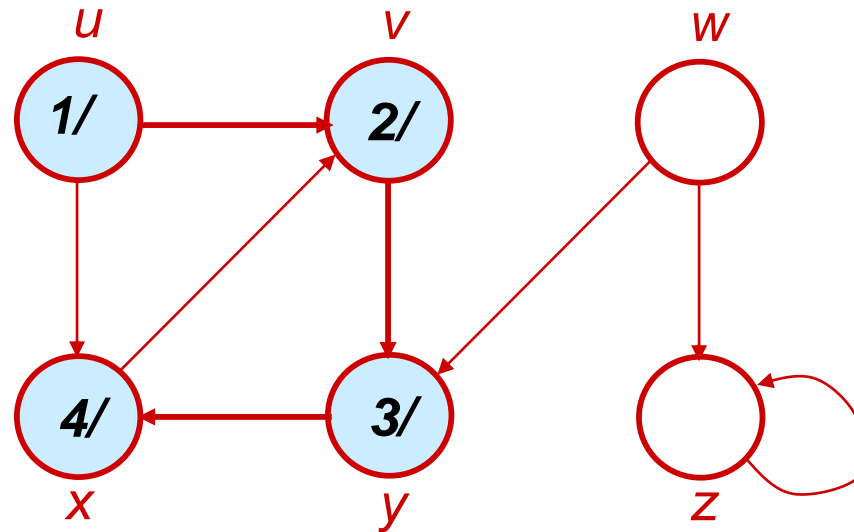
Example (DFS)



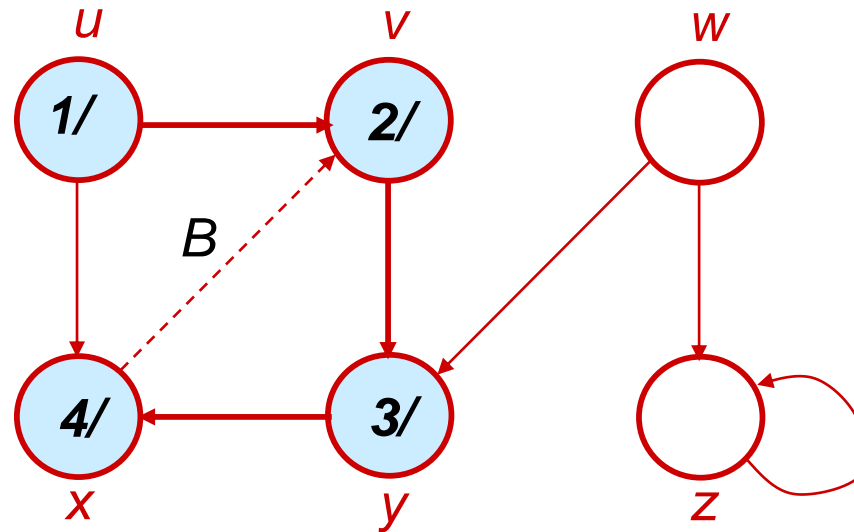
Example (DFS)



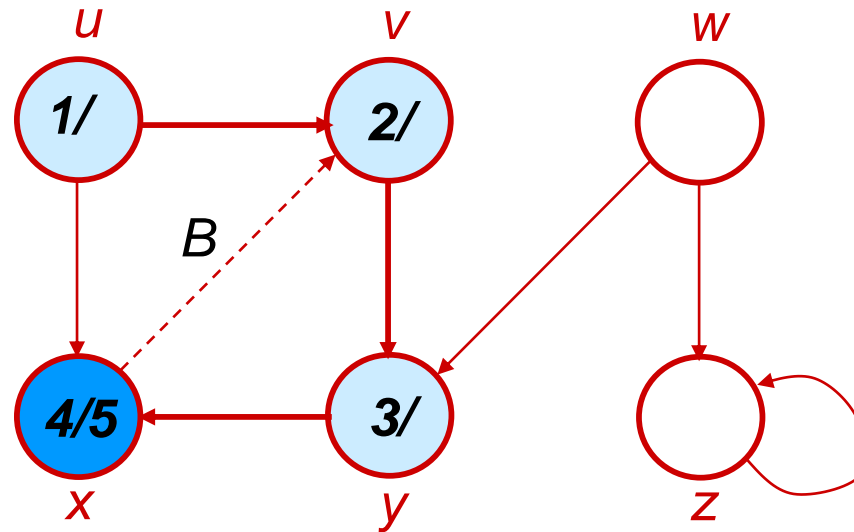
Example (DFS)



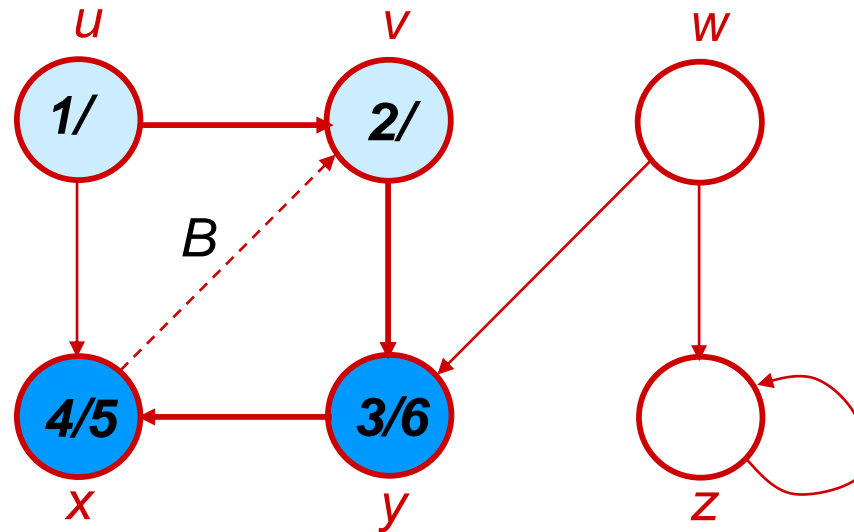
Example (DFS)



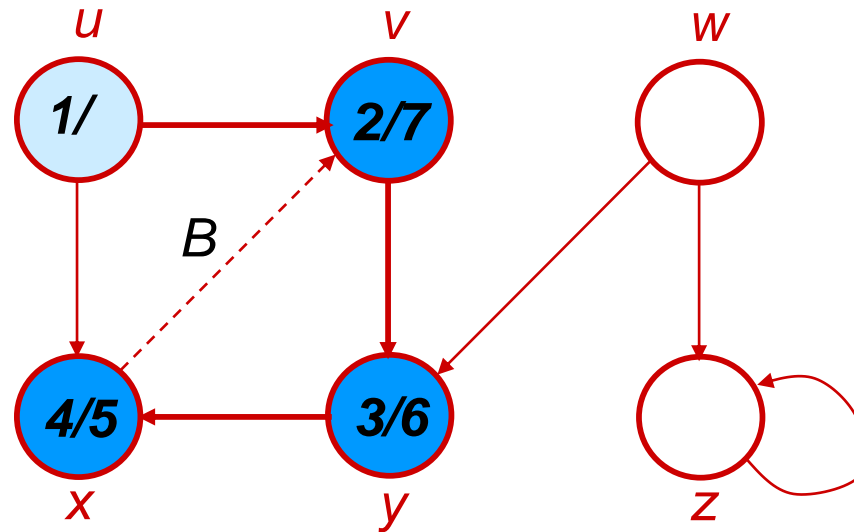
Example (DFS)



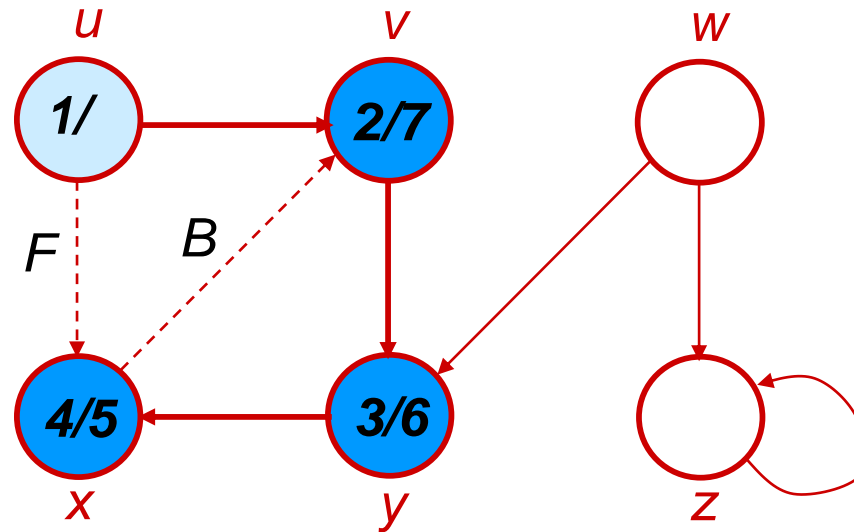
Example (DFS)



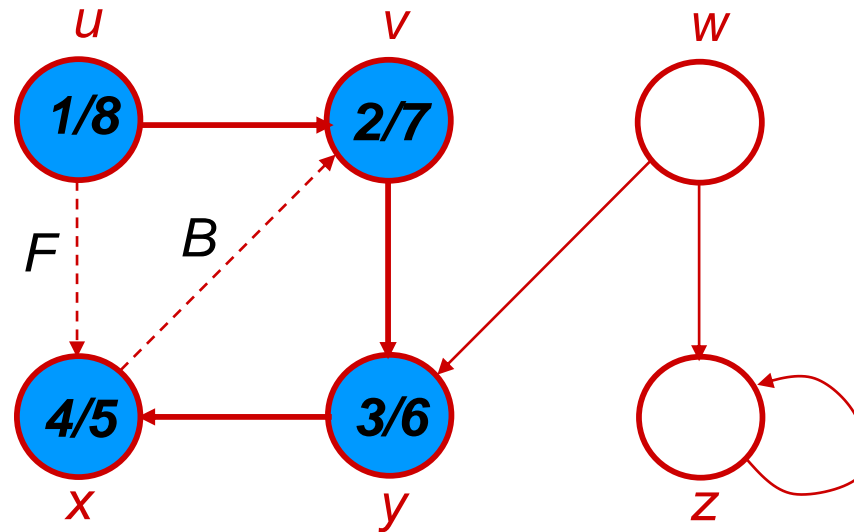
Example (DFS)



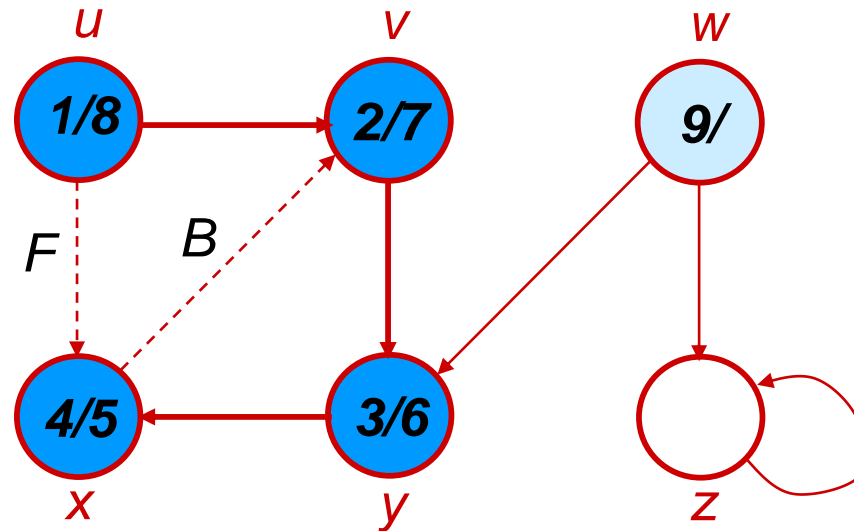
Example (DFS)



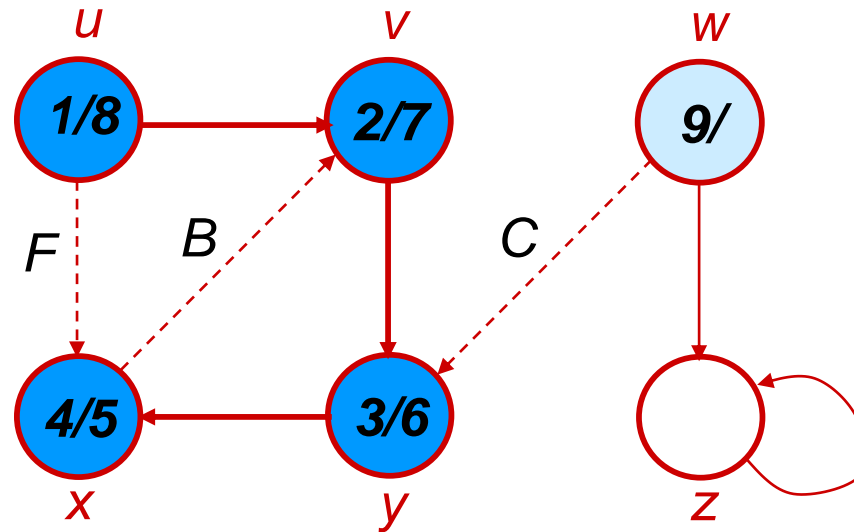
Example (DFS)



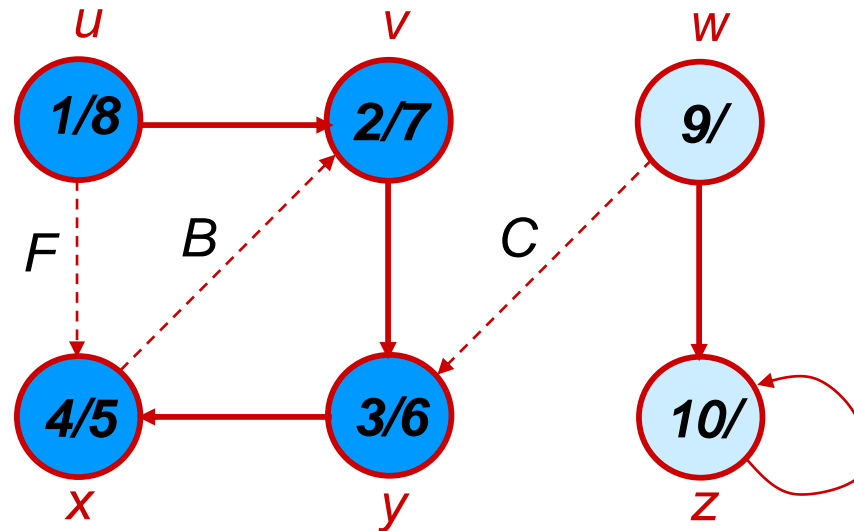
Example (DFS)



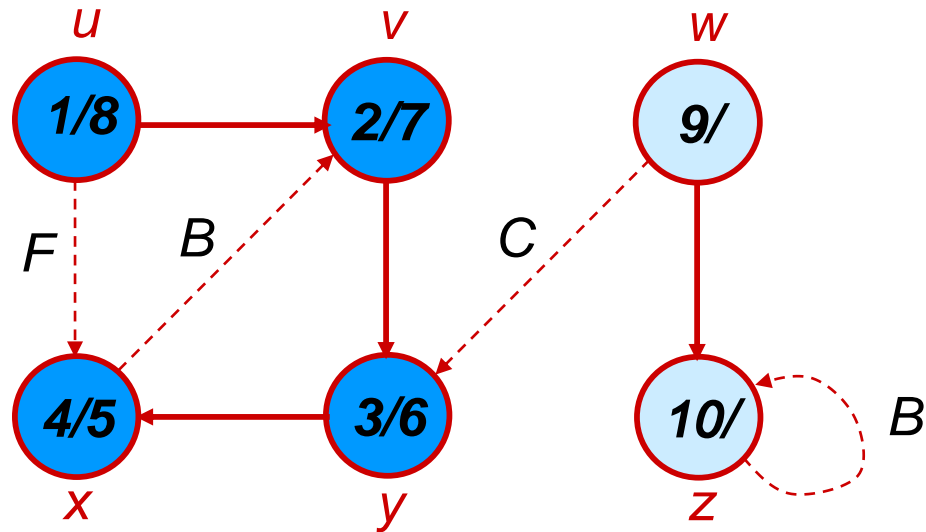
Example (DFS)



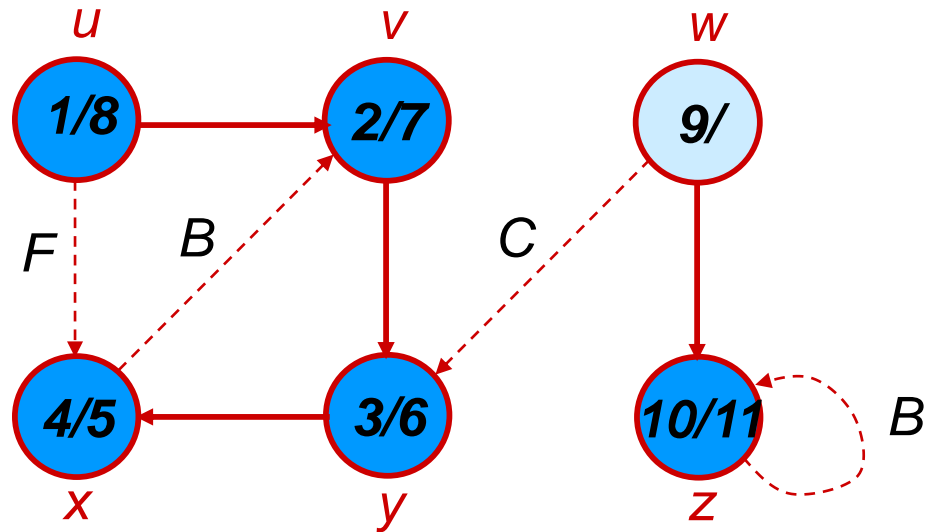
Example (DFS)



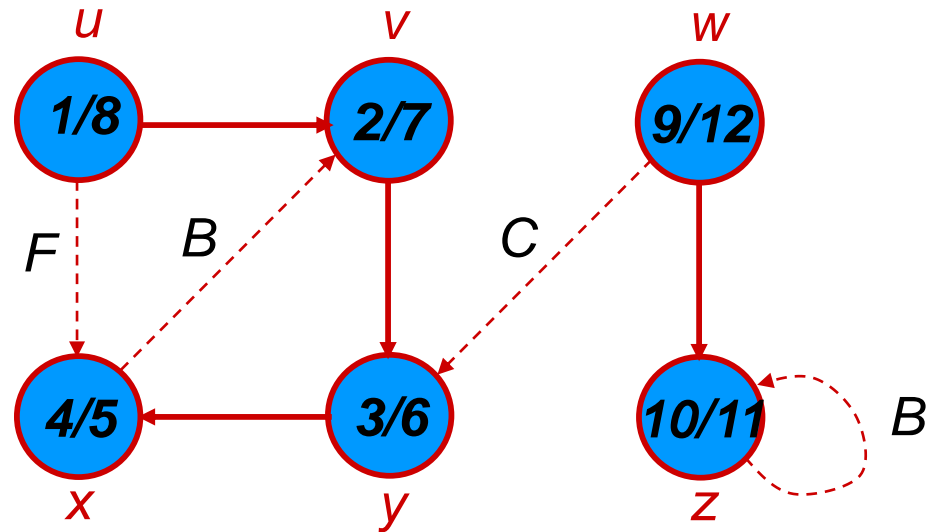
Example (DFS)



Example (DFS)



Example (DFS)



DFS-Properties

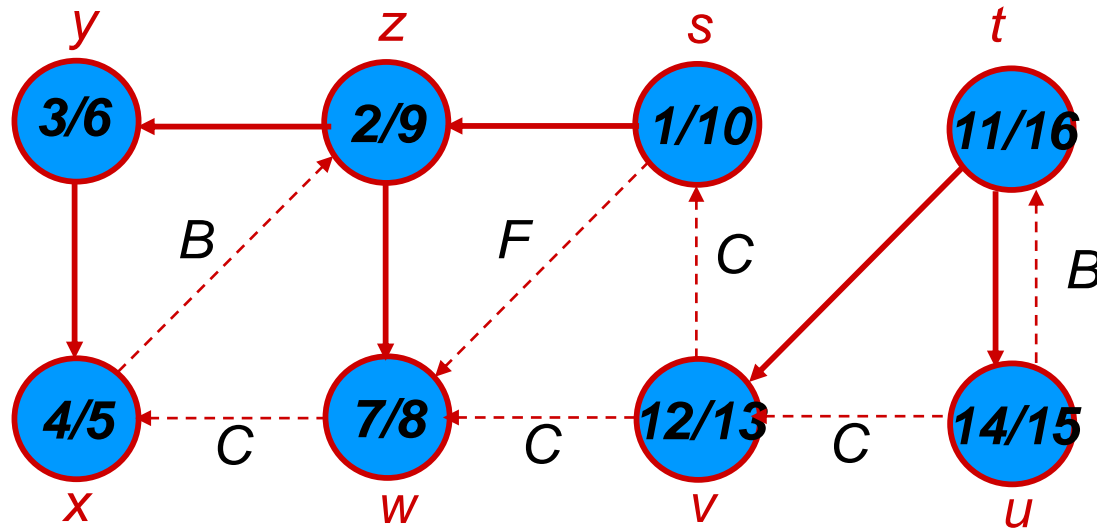
- $u=v$. π , iff DFS-Visit(G, v) was called during a search of u 's adjacency list
- V is descendent of u in DFS iff v is discovered when u is grey.
- Paranthesis structure
 - If we represent the discovery of vertex u with a left parenthesis “(u ” and represent its finishing by a right parenthesis “ u)”, then the history of discoveries and finishings makes a well-formed expression in the sense that the parentheses are properly nested.

Parenthesis Theorem

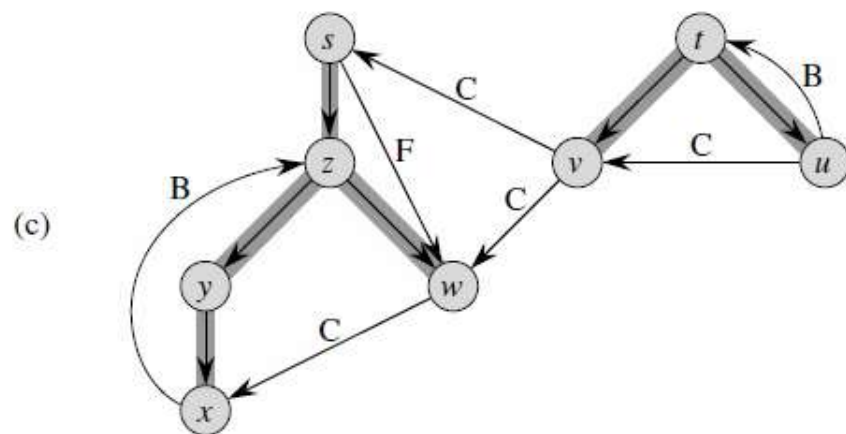
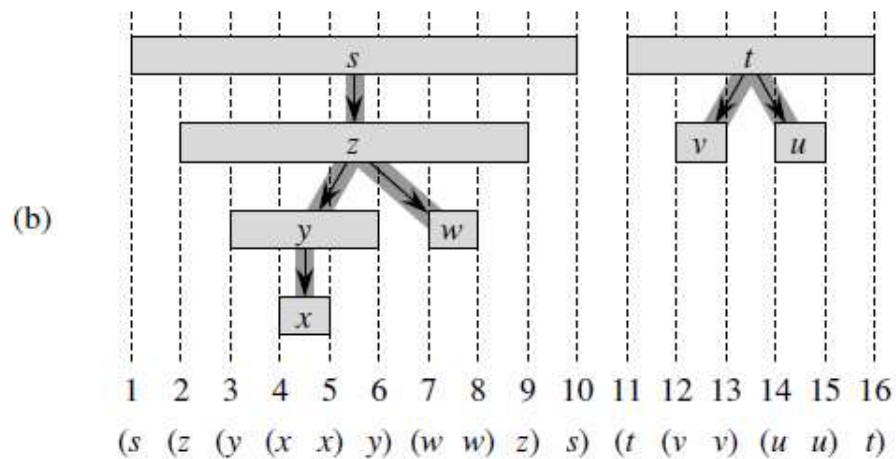
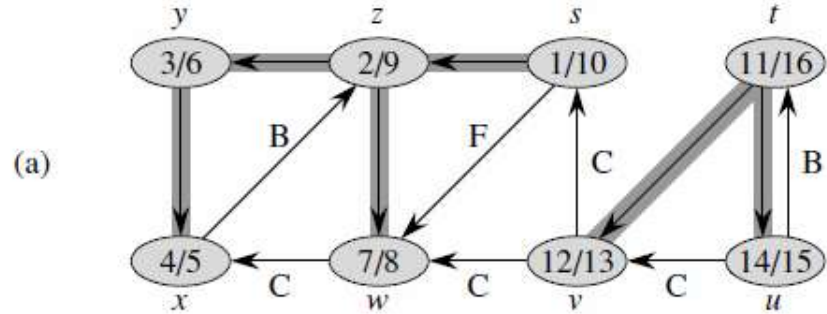
Theorem 22.7

For all u, v , exactly one of the following holds:

1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither u nor v is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u .
3. $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v .



$(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)$



Classification of Edges

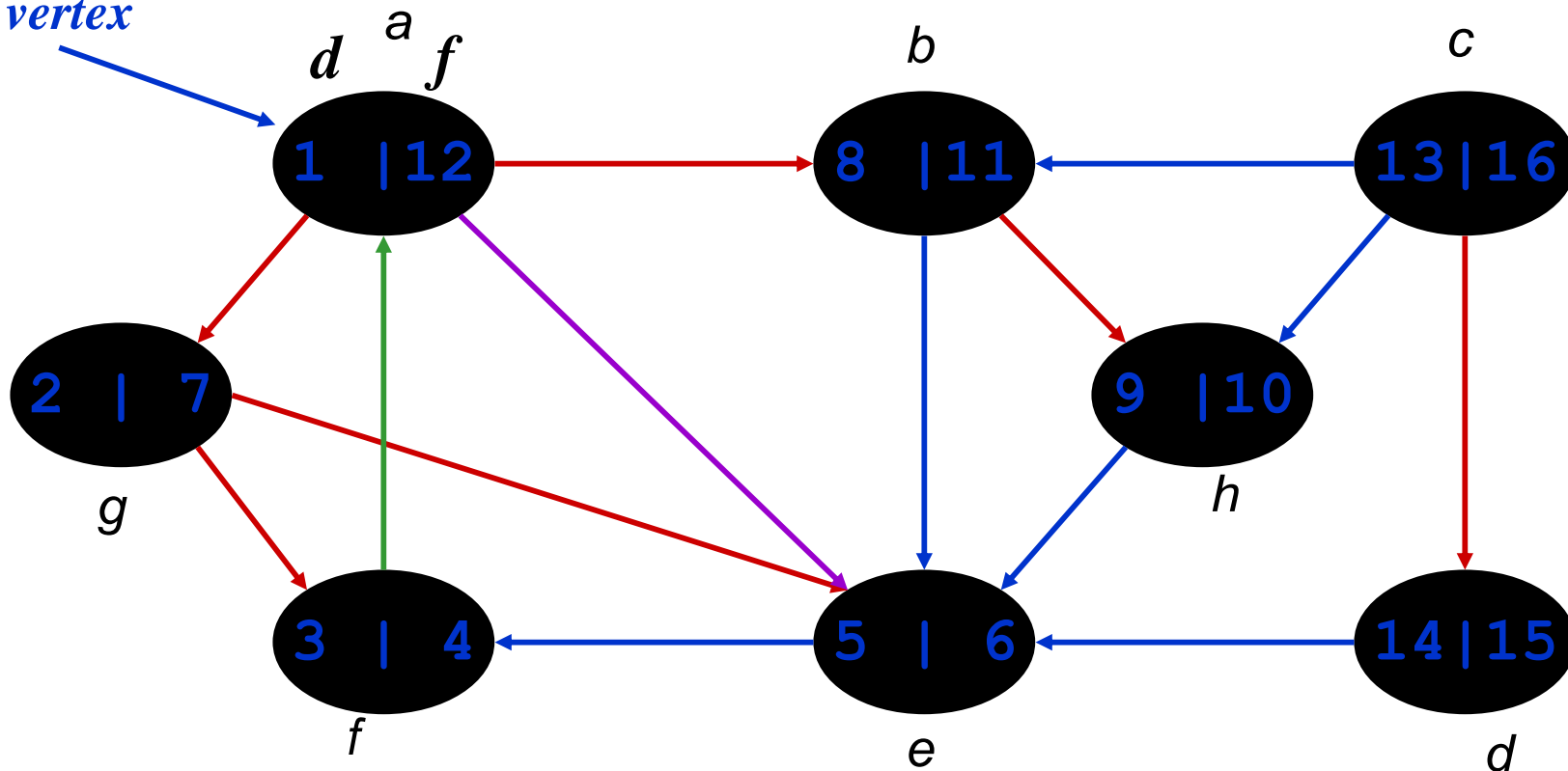
- **Tree edge:** in the depth-first forest. Found by exploring (u, v) .
- **Back edge:** (u, v) , where u is a descendant of v (in the depth-first tree).
- **Forward edge:** (u, v) , where v is a descendant of u , but not a tree edge.
- **Cross edge:** any other edge. Can go between vertices in same 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.

DFS Example

*source
vertex*



Tree edges *Back edges* *Forward edges* *Cross edges*

DFS And Graph Cycles

- Theorem: An undirected graph is *acyclic* iff a DFS yields no back edges
 - If acyclic, no back edges (because a back edge implies a cycle)
 - If no back edges, acyclic
 - No back edges implies only tree edges
 - Only tree edges implies we have a tree or a forest which by definition is acyclic
- Thus, can run DFS to find whether a graph has a cycle

Problem

