

EL9343

Data Structure and Algorithm

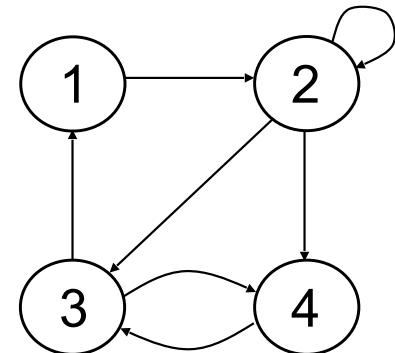
Lecture 8: Graph Basics

Instructor: Yong Liu

Graphs

- ▶ **Definition** = a set of **vertices** (nodes) with **edges** (links) between them.

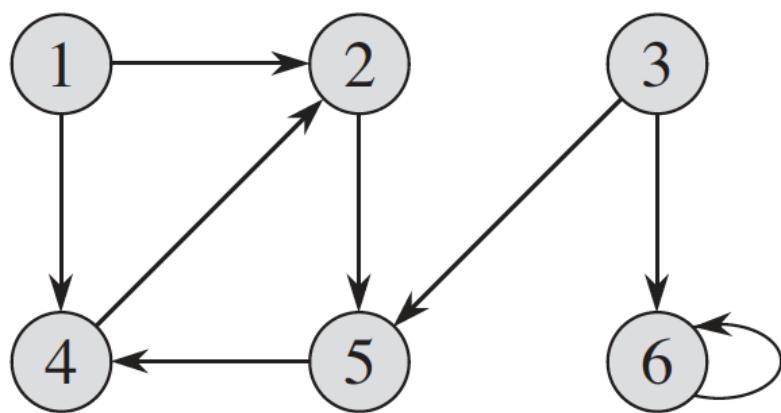
- ▶ $G = (V, E)$ - graph
- ▶ V = set of vertices
- ▶ E = set of edges = subset of $V \times V$
- ▶ Thus $|E| = O(|V|^2)$



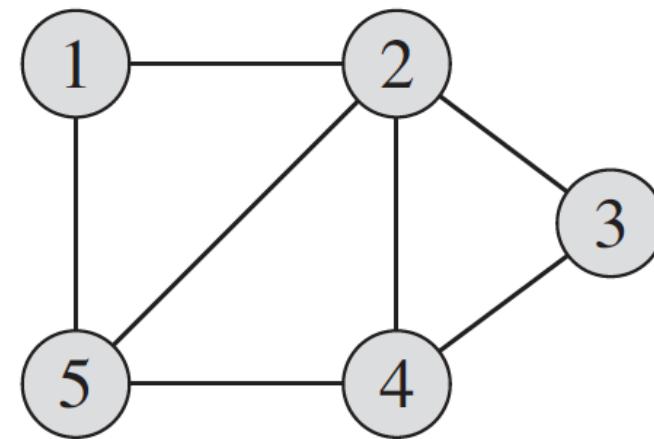
Directed

Directed Graphs (Digraph) VS. Undirected Graph

Directed Graphs (digraphs)
(ordered pairs of vertices)



Undirected Graphs
(unordered pairs of vertices)



Node 4:

in-degree of v: # of edges entering v: 2

out-degree of v: # of edges leaving v: 1

v is adjacent to u if there is an edge (u,v)
from u to v. it's directed.

Node 5:
degree of v: # of edges incident on v
= 3

v is adjacent to u and u is adjacent to v if there is an edge between v and u

*from u to v
from v to u > No direction.*

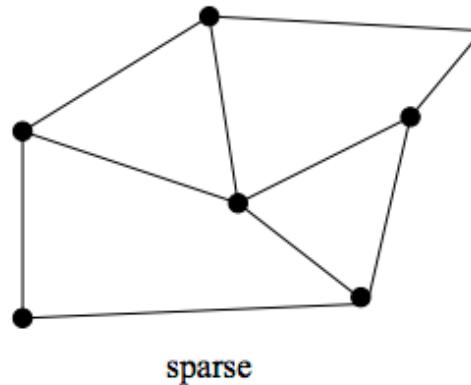
Weighted & Multiple

More Graph variations

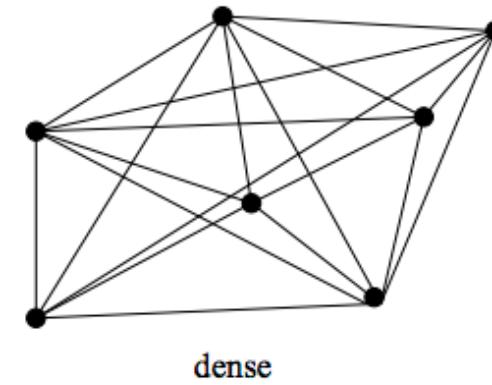
- ▶ A *weighted graph* associates weights with either the edges or the vertices
 - ▶ e.g., a road map: edges might be weighted w/ distance
- ▶ A *multigraph* allows multiple edges between the same pair of vertices
 - ▶ e.g., the call graph in a program (a function can get called from multiple points in another function)
relation about functions

Sparse VS. Dense Graphs

- ▶ We will typically express running times in terms of $|E|$ and $|V|$ $|E| = O(|V|^2)$
 - ▶ If $|E| \approx |V|^2$ the graph is *dense*
 - ▶ have a quadratic number of edges
 - ▶ If $|E| \approx |V|$ the graph is *sparse*
 - ▶ linear in size, only a small fraction of the possible number of vertex pairs actually have edges defined between them



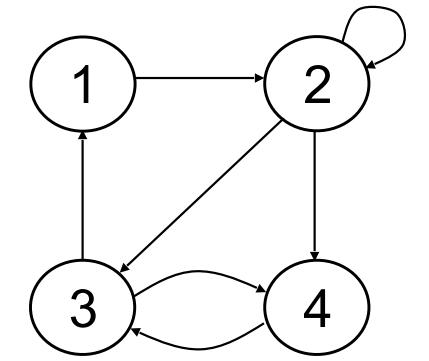
sparse



dense

Terminology

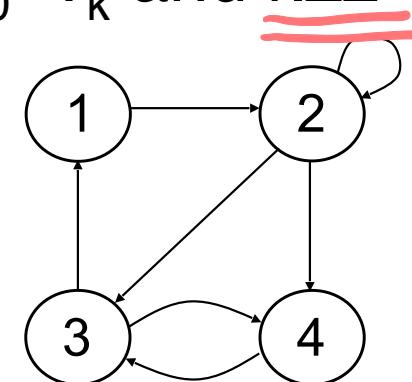
- ▶ Complete graph
 - ▶ A graph with an edge between each pair of vertices
- ▶ Subgraph
 - ▶ A graph (V', E') such that $V' \subseteq V$ and $E' \subseteq E$
- ▶ Path from v to w $\langle (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k) \rangle$
 - ▶ A sequence of vertices $\langle v_0, v_1, \dots, v_k \rangle$ such that $v_0 = v$ and $v_k = w$. *Directed* *not unique*
- ▶ Length of a path
 - ▶ Number of edges along the path



path from v_1 to v_4
 $\langle v_1, v_2, v_4 \rangle$

Terminology (cont'd)

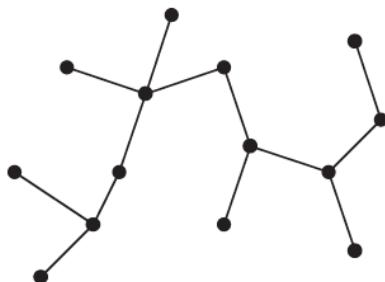
- ▶ w is **reachable** from v
 - ▶ If there is a path from v to w
- ▶ **Simple** path
 - ▶ All the vertices in the path are distinct
- ▶ Cycles $\langle (1,2), (2,3), (3,1) \rangle$
 - ▶ A path $\langle v_0, v_1, \dots, v_k \rangle$ forms a cycle if $v_0 = v_k$ and $k \geq 2$
- ▶ **Acyclic graph**
 - ▶ A graph without any cycles



cycle from v_1 to v_1
 $\langle v_1, v_2, v_3, v_1 \rangle$

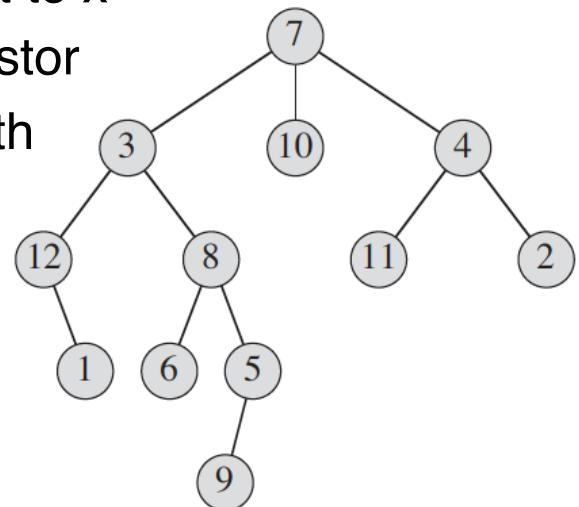
Special case: Tree is a special case of graph

- ▶ **(Free) Tree**: connected, acyclic, undirected graph
- ▶ **Forest**: acyclic, undirected graph, possibly disconnected



- ▶ **Rooted Tree**: a free tree with special **root** node

- **Ancestor** of node x: any node on the path from root to x
 - **Descendant** of node x: any node with x as its ancestor
 - **Parent** of node x: node immediately before x on path from root
 - **Child** of node x: any node with x as its parent
 - **Siblings** of node x: nodes sharing parent with x
 - **Leaf/external node**: without child
- ▶ 8
- **Internal node**: with at least one child

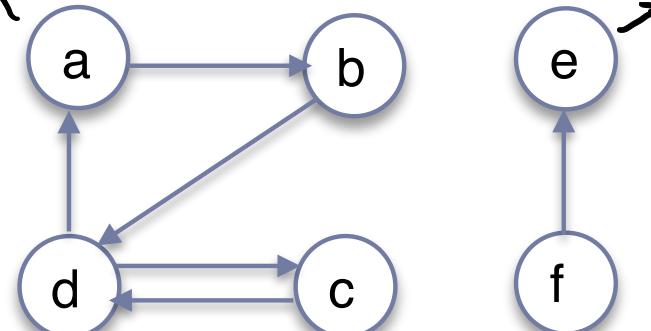


Strongly connected VS. Connected

Directed Graphs

Strongly connected: every two vertices are reachable from each other

Strongly connected components: all possible strongly connected subgraphs

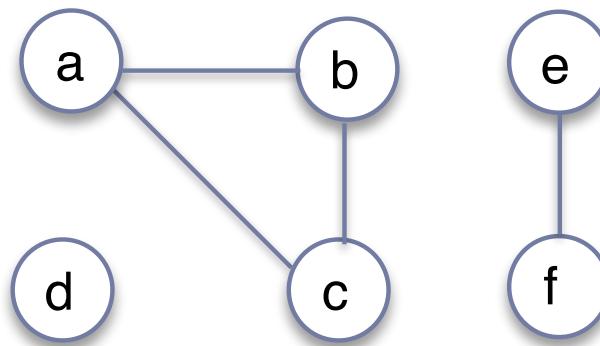


strongly connected components:
 $\{a,b,c,d\}, \{e\}, \{f\}$

Undirected Graphs

connected: every pair of vertices are connected by a path

connected components: all possible connected subgraphs



connected components:
 $\{a,b,c\}, \{d\}, \{e,f\}$

Representing Graphs

▶ Adjacency matrix representation of $G = (V, E)$

- ▶ Assume vertices are numbered $1, 2, \dots |V|$
- ▶ The representation consists of a matrix $A_{|V| \times |V|}$:

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

memory required: $\Theta(|V|^2)$

Adjacency matrix: advantage: find edge between two vertices quickly $O(1)$

Disadvantage: Can't find all adjacent to u quickly $O(|V|)$

Adjacency List: advantage: find all adjacent to u quickly $O(\deg(u))$

Disadvantage: find edge between two vertices quickly $O(n)$ slower

memory required:
 $\Theta(V+E)$

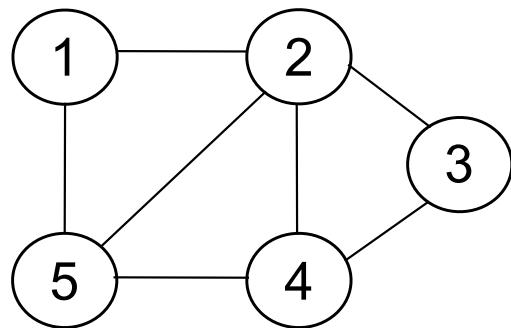
$S O(\deg(u)) : \text{if } (u, v) \in E$

$O(\deg(u)) : \text{all vertices adjacent to } u$

Graphs: Adjacency Matrix

▶ Example

undirected: symmetric



Undirected graph

	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

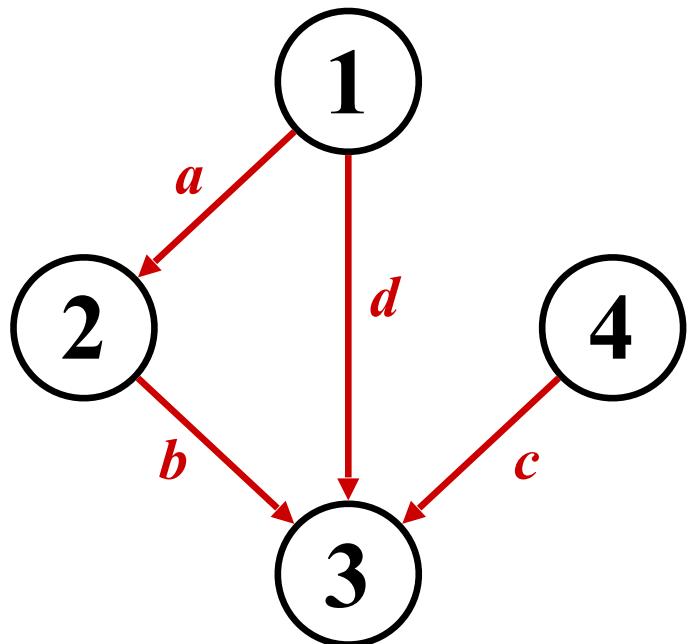
For undirected graphs, matrix A is symmetric:

$$a_{ij} = a_{ji}$$

$$A = A^T$$

Graphs: Adjacency Matrix

- ▶ Another Example



directed graph

directed graph: Not Symmetric

		1	2	3	4
		A	to	B	
1	0	1	1	0	
2	0	0	1	0	
3	0	0	0	0	
4	0	0	1	0	

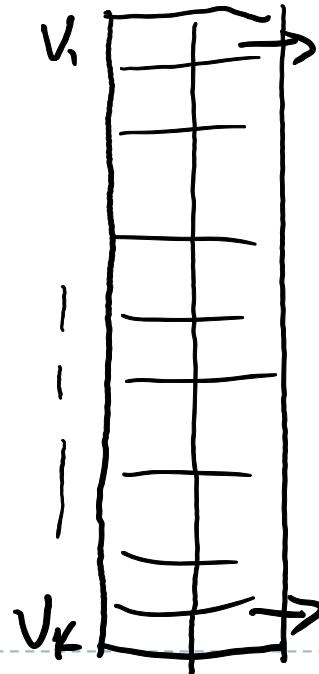
Properties of Adjacency Matrix Representation

- ▶ Memory required
 - ▶ $\Theta(V^2)$, independent on the number of edges in G
- ▶ Preferred when
 - ▶ The graph is **dense**: $|E|$ is close to $|V|^2$
 - ▶ We need to quickly determine if there is an edge between two vertices
- ▶ Time to determine if $(u, v) \in E$:
 - ▶ $\Theta(1)$
- ▶ Disadvantage
 - ▶ No quick way to determine the vertices adjacent to a vertex
- ▶ Time to list all vertices adjacent to u:
 - ▶ $\Theta(V)$

$A(u, _)$

Graph Adjacency List

- ▶ **Adjacency list representation of $G = (V, E)$**
 - ▶ An array of $|V|$ lists, one for each vertex in V
 - ▶ Each list $\text{Adj}[u]$ contains all the vertices v that are adjacent to u (i.e., there is an edge from u to v)
 - ▶ Can be used for both directed and undirected graphs



number of list:

Graph Adjacency List

undirected: degree

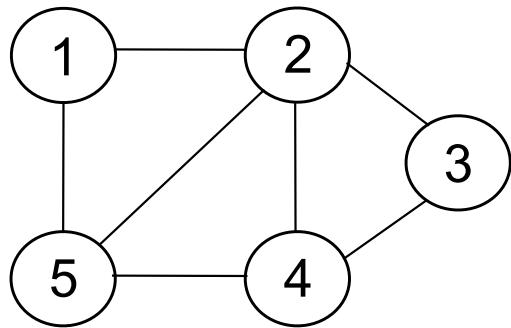
directed: out degree.

total length of lists:

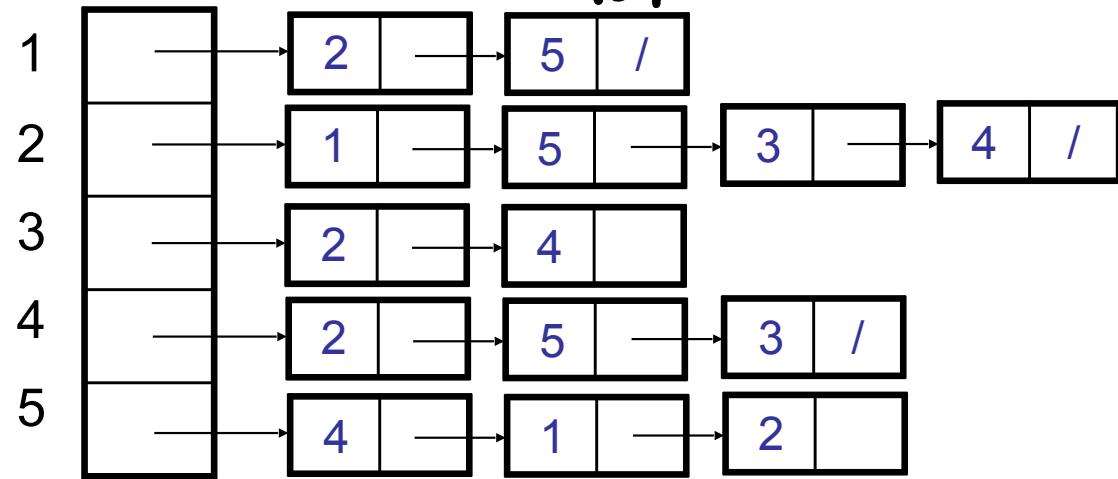
undirected = $|E|$

directed = $|E|$

Example



Undirected graph

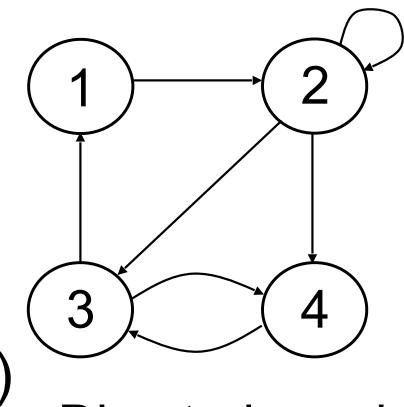


Properties of Adjacency-List Representation

- ▶ Sum of “lengths” of all adjacency lists

- ▶ Directed graph: $|E|$

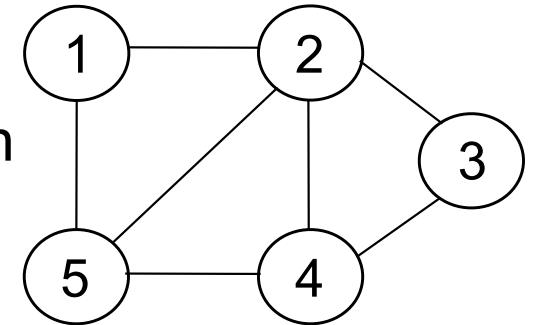
- ▶ edge (u, v) appears only once (i.e., in the list of u)



Directed graph

- ▶ Undirected graph: $2|E|$

- ▶ edge (u, v) appears twice (i.e., in the lists of both u and v)

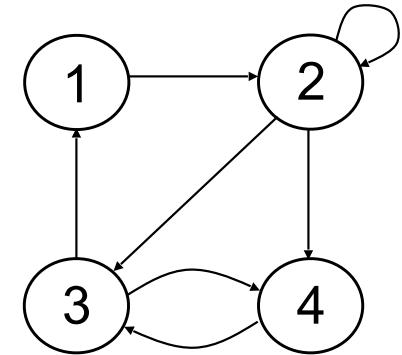


Undirected graph

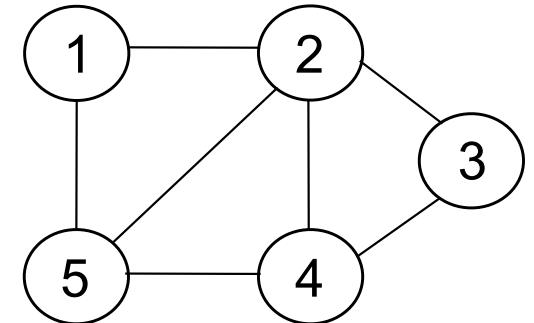
Properties of Adjacency-List Representation

- ▶ Memory required
 - ▶ $\Theta(V + E)$
- ▶ Preferred when
 - ▶ The graph is **sparse**: $|E| \ll |V|^2$
 - ▶ We need to quickly determine the nodes adjacent to a given node.
- ▶ Disadvantage
 - ▶ No quick way to determine whether there is an edge between node u and v
- ▶ Time to determine if $(u, v) \in E$:
 - ▶ $O(\text{degree}(u))$
- ▶ Time to list all vertices adjacent to u:
 - ▶ $\Theta(\text{degree}(u))$

$O(1)$ }
 $O(|V|)$ }
if use
adjacency matrix.



Directed graph



Undirected graph

Graph Search

- ▶ Given: a graph $G = (V, E)$, directed or undirected
- ▶ In general, given a vertex s , we want to locate some vertex t .
 - ▶ Find a path in G
 - ▶ We want to visit all vertices in a "local" organized manner

Breadth-First Search (BFS)

- ▶ “Explore” a graph, turning it into a tree
 - ▶ start with a *source vertex*, explore all other vertices *reachable* from the source, one vertex at a time
 - ▶ expand frontier of explored vertices across the *breadth* of the frontier
 - ▶ compute the distance (smallest number of edges) from source to each reachable vertex
- ▶ Builds a breadth-first tree over the graph
 - ▶ source is the root, cover all reachable vertices
 - ▶ find (“discover”) its children, then their children, etc.
 - ▶ discover vertices at distance k from source before discovering vertices at distance $k+1$
 - ▶ the path from source to a vertex in breadth-first tree is the shortest path in the original graph

Breadth-First Search (BFS)

- ▶ Associate vertex “colors” to guide the algorithm
 - ▶ White vertices have not been discovered
 - ▶ All vertices start out white
 - ▶ Gray vertices are discovered but not fully explored
 - ▶ They may have some adjacent white vertices: *have white neighbors*.
 - ▶ Black vertices are discovered and fully explored
 - ▶ adjacent vertices of a black vertices are either black or gray
- ▶ Explore vertices by scanning adjacency list of gray vertices

Breadth-First Search (BFS)

```
BFS(G, s) {
    for each u in V {                                // initialization
        color[u] = white
        d[u]      = infinity // distance
        pred[u]   = null // shortest path from s to u // predecessor
    }
    color[s] = gray                                // initialize source s
    d[s] = 0
    Q = {s} // BFS currently.                      // put s in the queue FIFO
    while (Q is nonempty) {
        u = Q.Dequeue()                            // u is the next to visit
        for each v in Adj[u] {
            if (color[v] == white) {
                color[v] = gray
                d[v]      = d[u]+1 // ...
                pred[v]   = u
                Q.Enqueue(v) // ...
            }
        }
        color[u] = black                           // we are done with u
    }
}
```

Starting point ↗

*works for both
Directed & Undirected.* ↗

// distance

// shortest path from s to u // predecessor

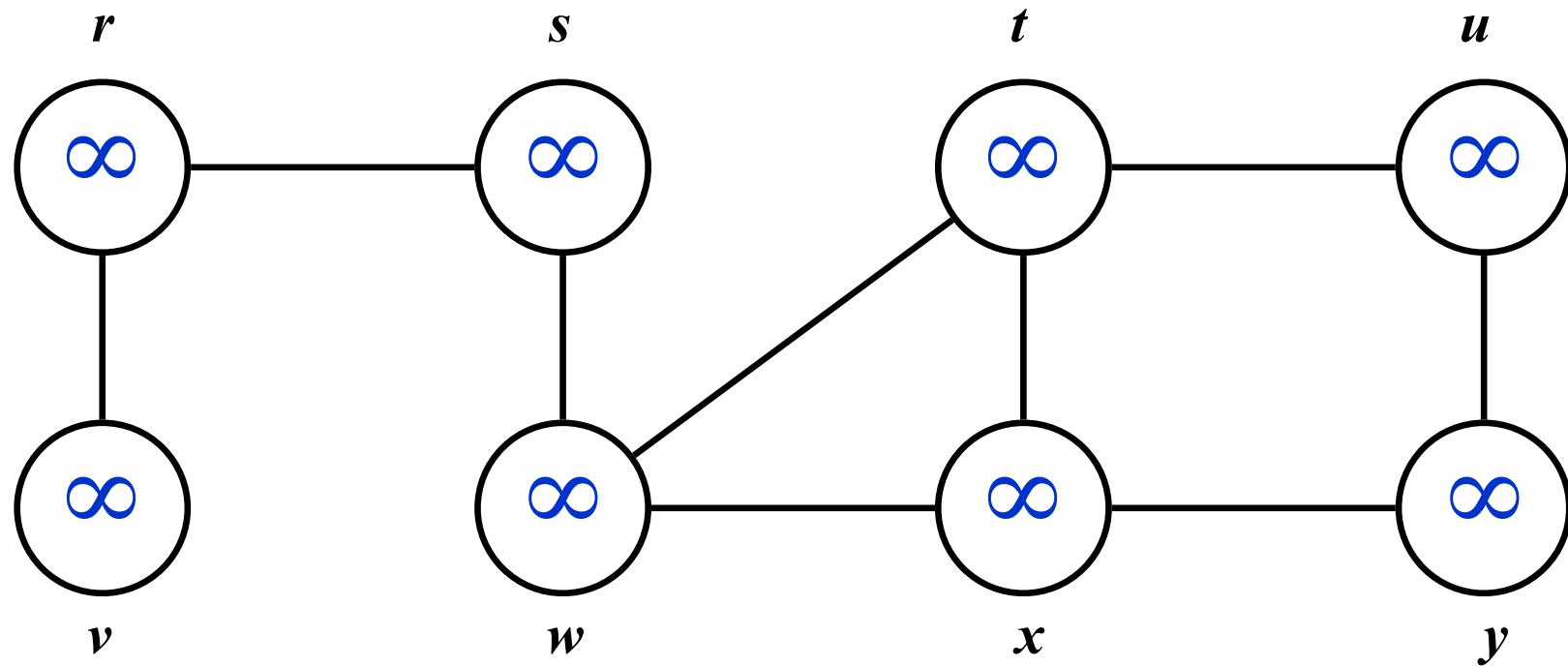
// ...mark it discovered

// ...set its distance

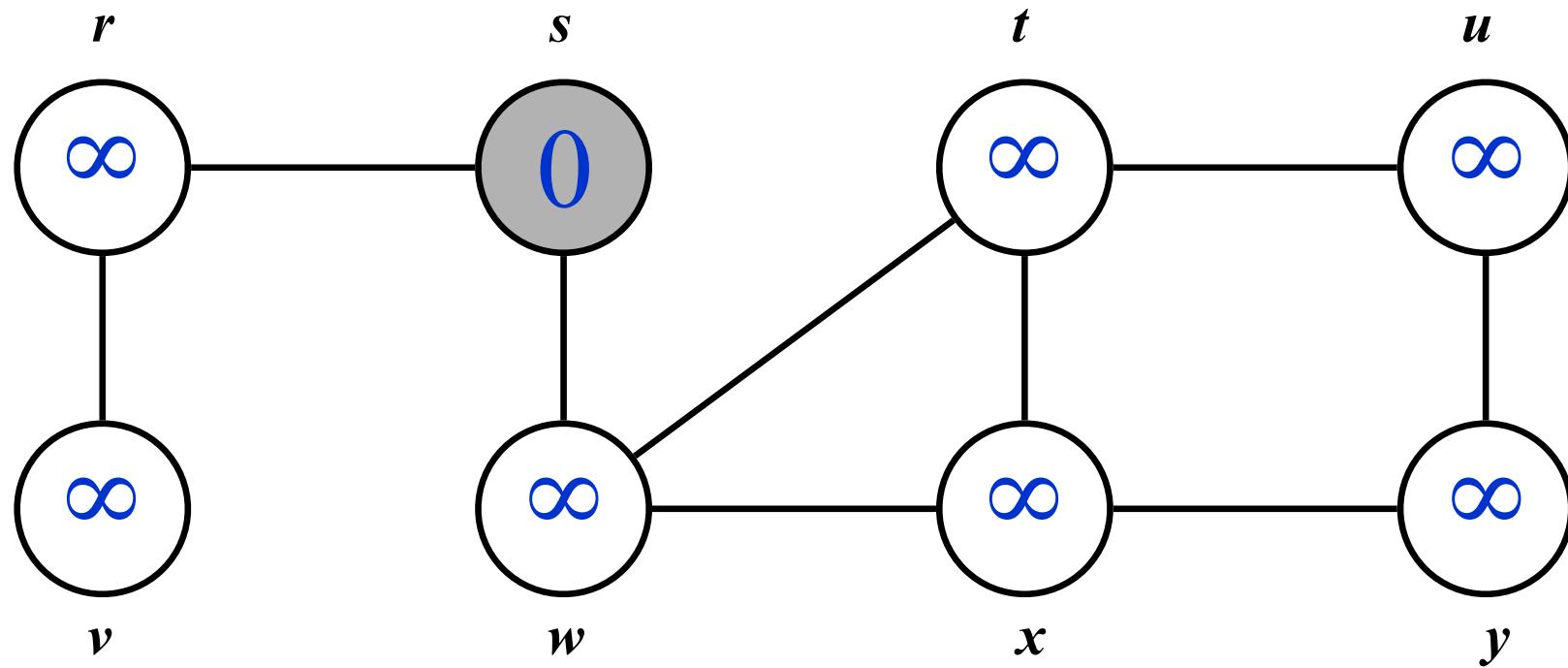
// ...and its predecessor

// ...put it in the queue

Breadth-First Search (BFS)-Example

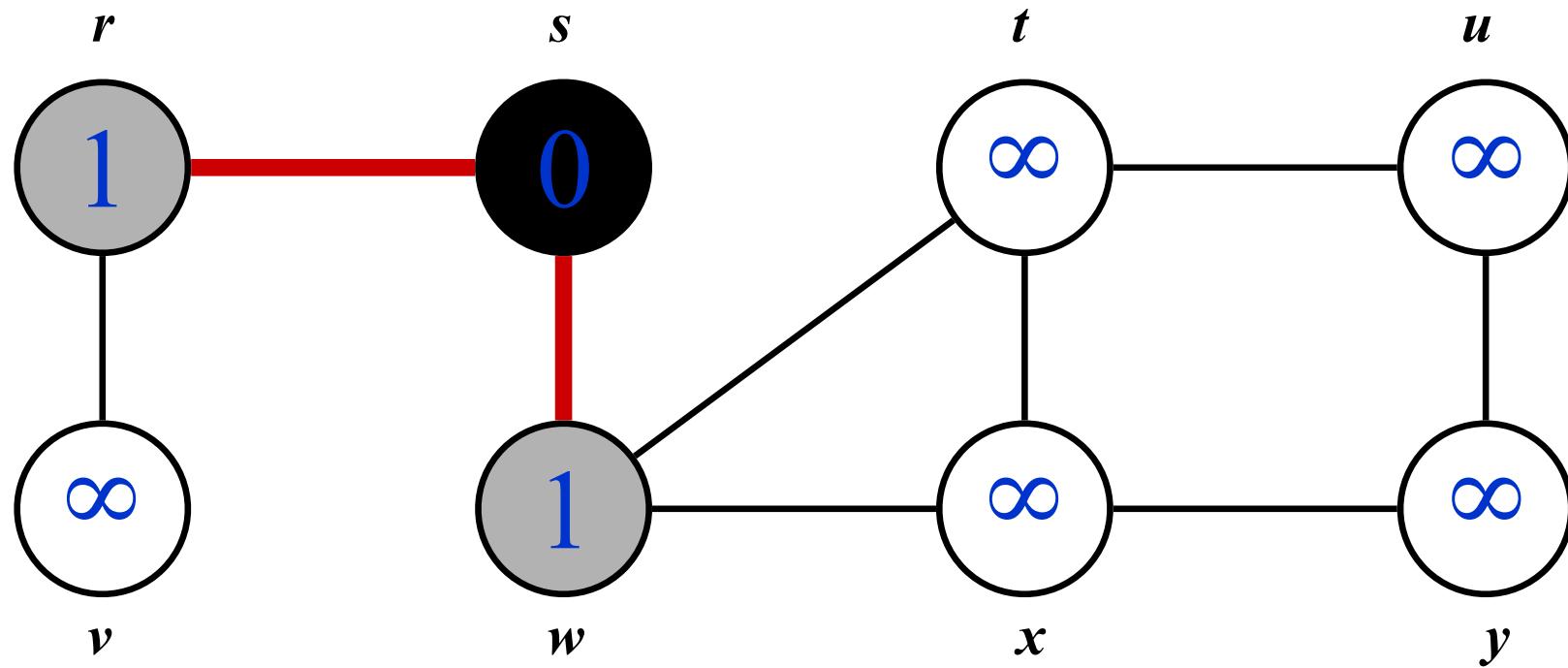


Breadth-First Search (BFS)-Example



$Q:$ s

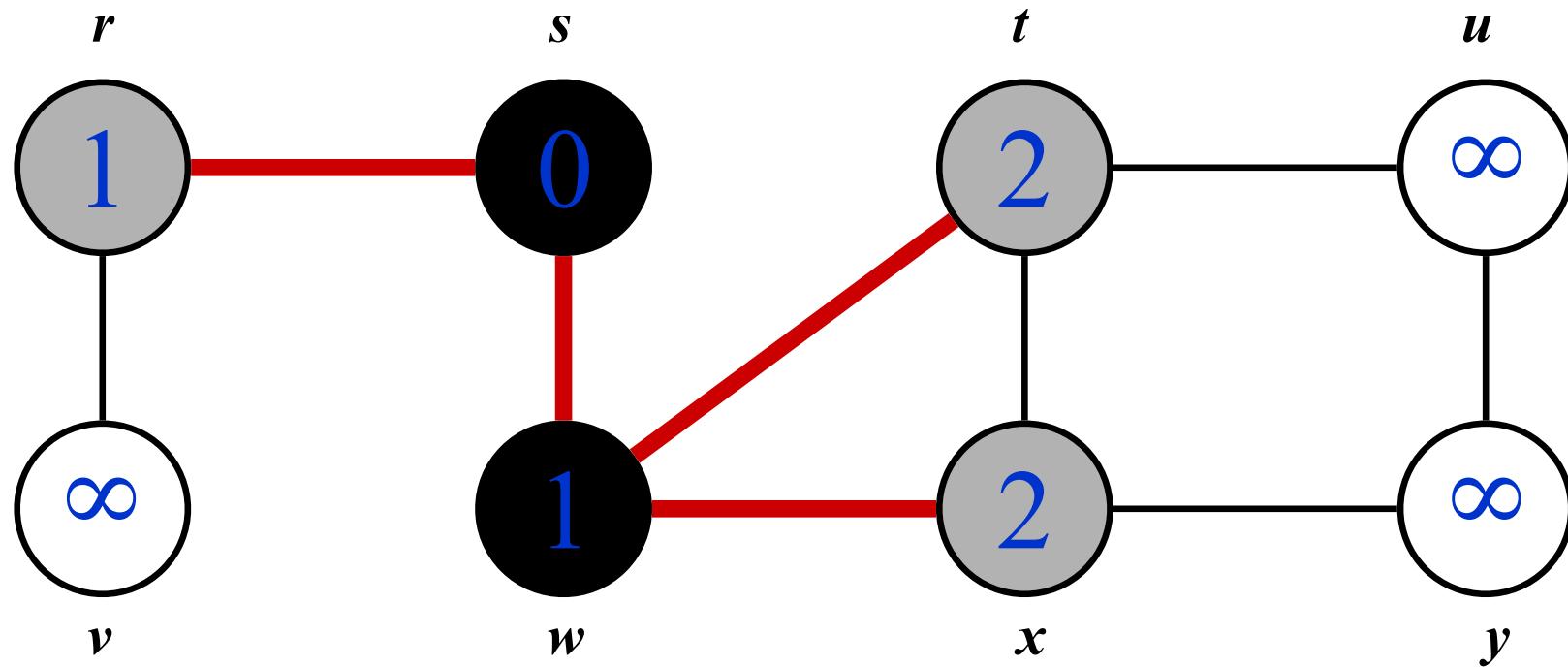
Breadth-First Search (BFS)-Example



$Q:$

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

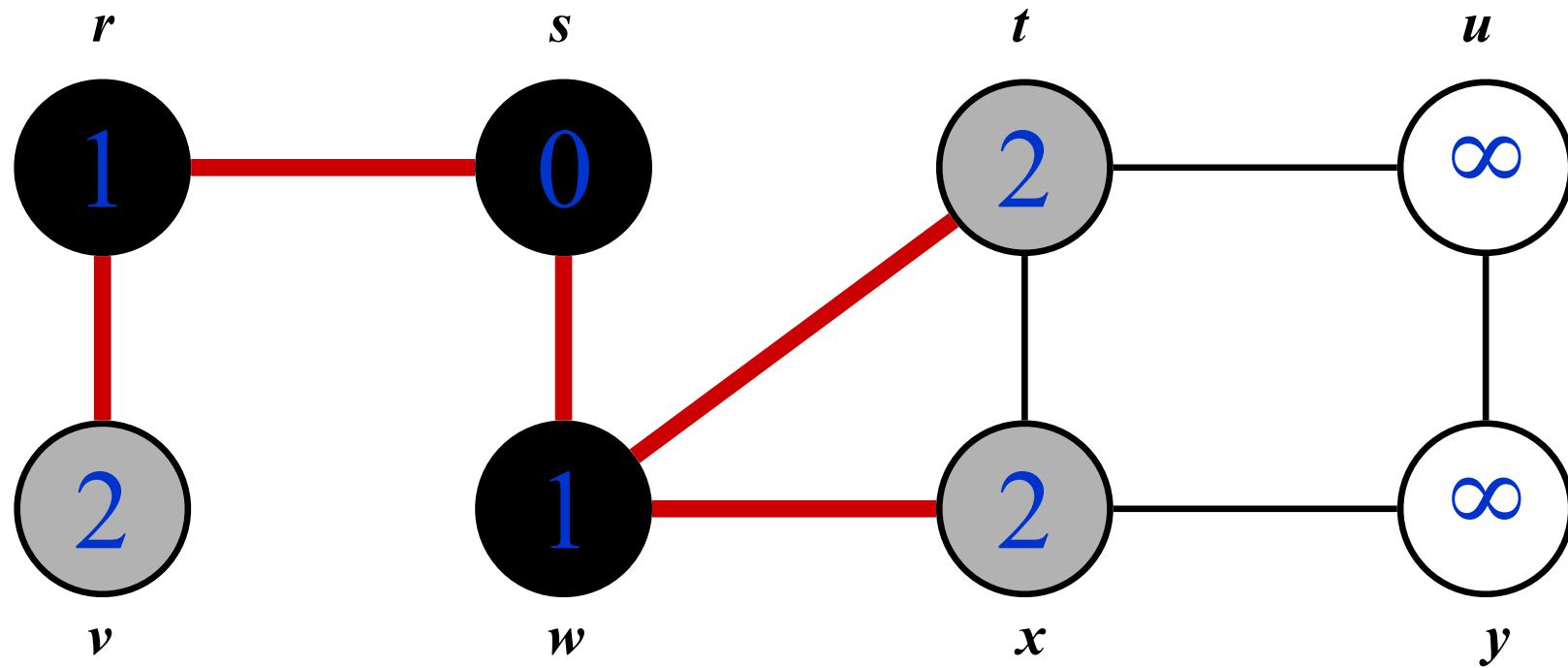
Breadth-First Search (BFS)-Example



$Q:$

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

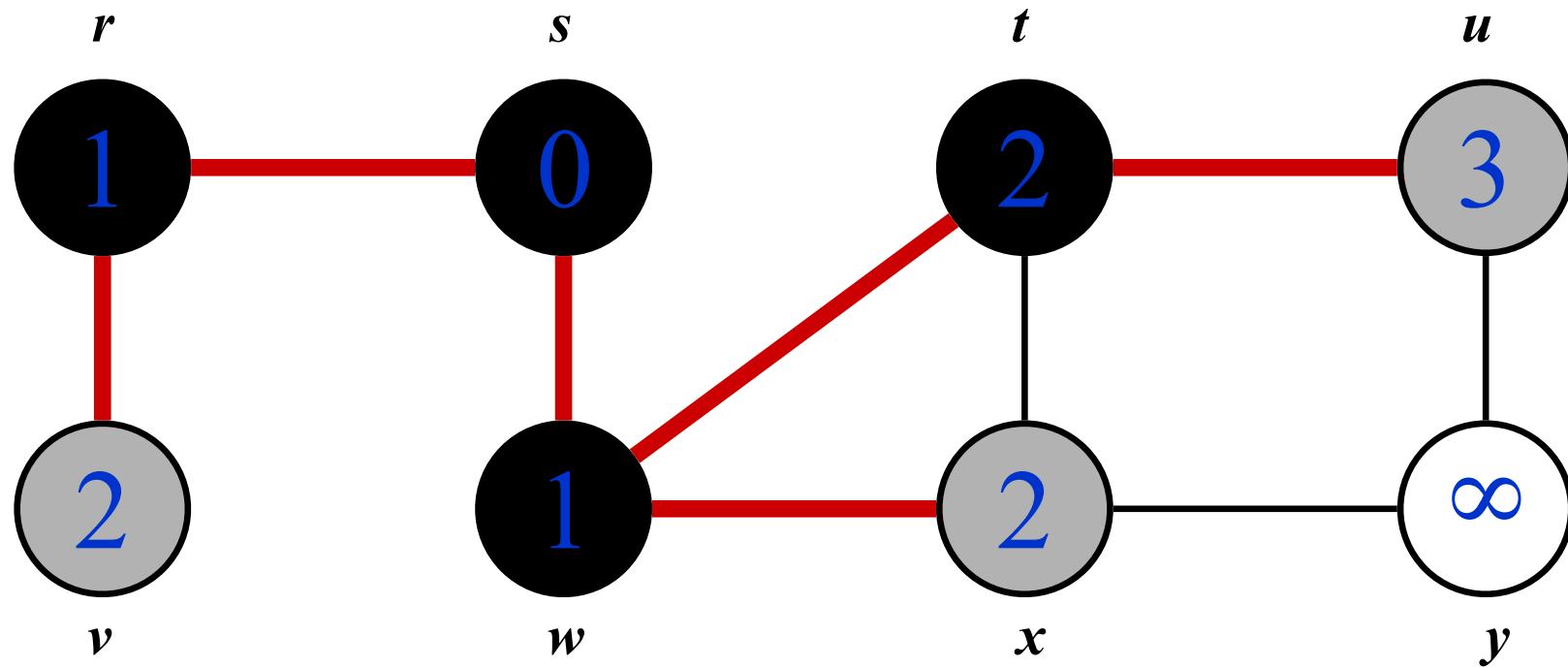
Breadth-First Search (BFS)-Example



$Q:$

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

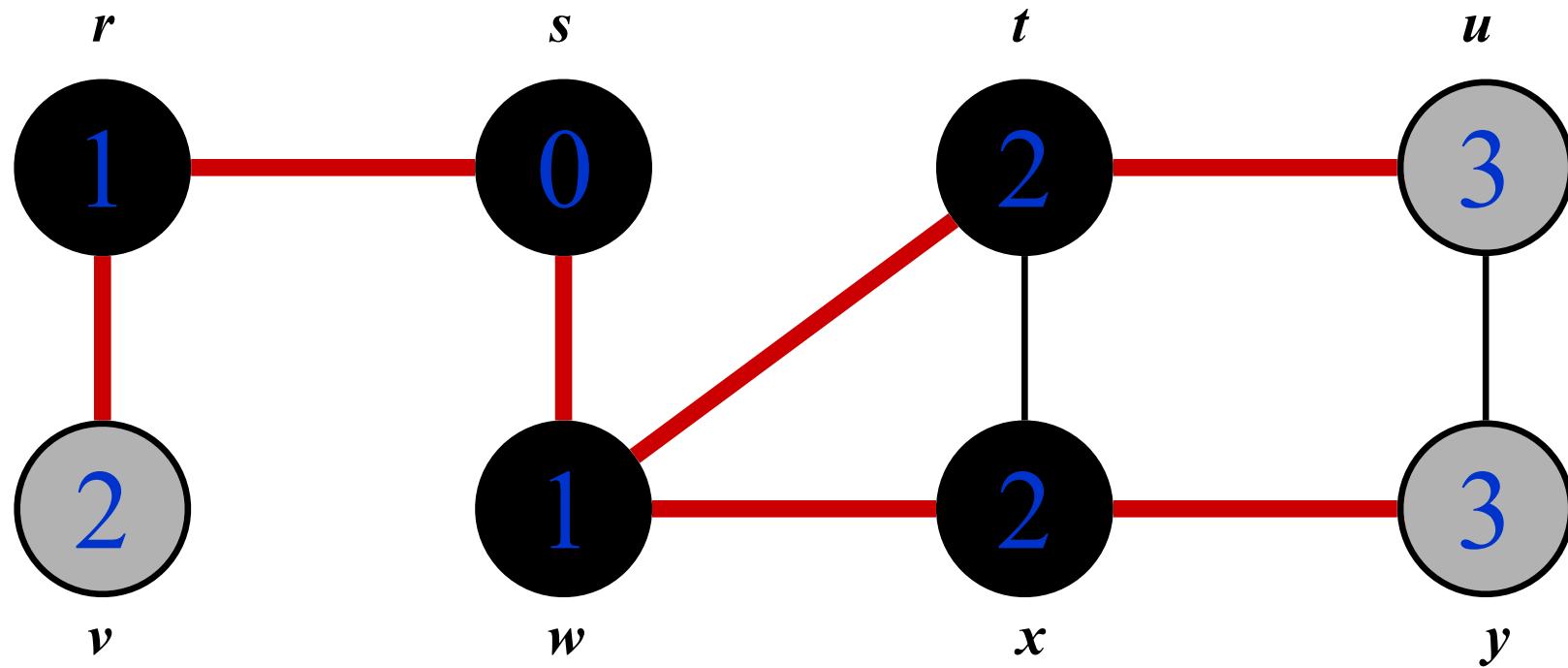
Breadth-First Search (BFS)-Example



$Q:$

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

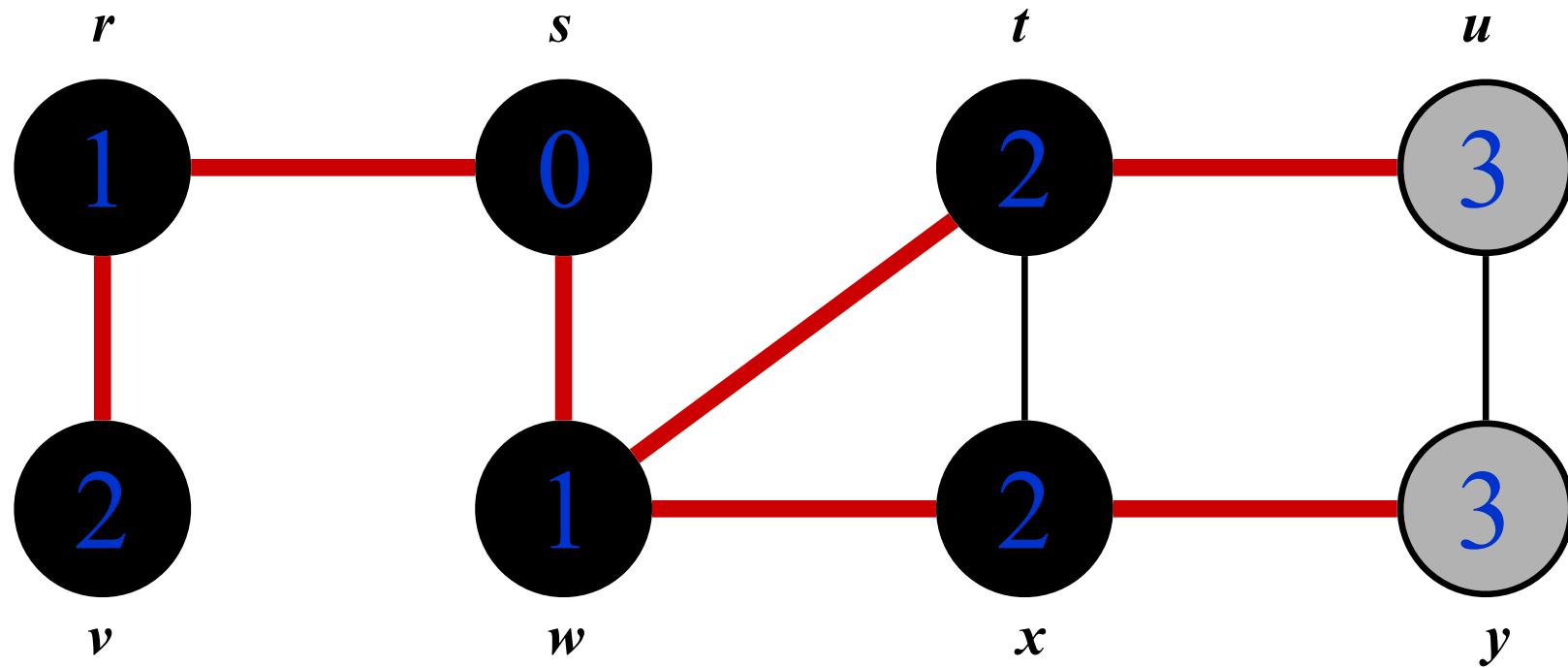
Breadth-First Search (BFS)-Example



$Q:$

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

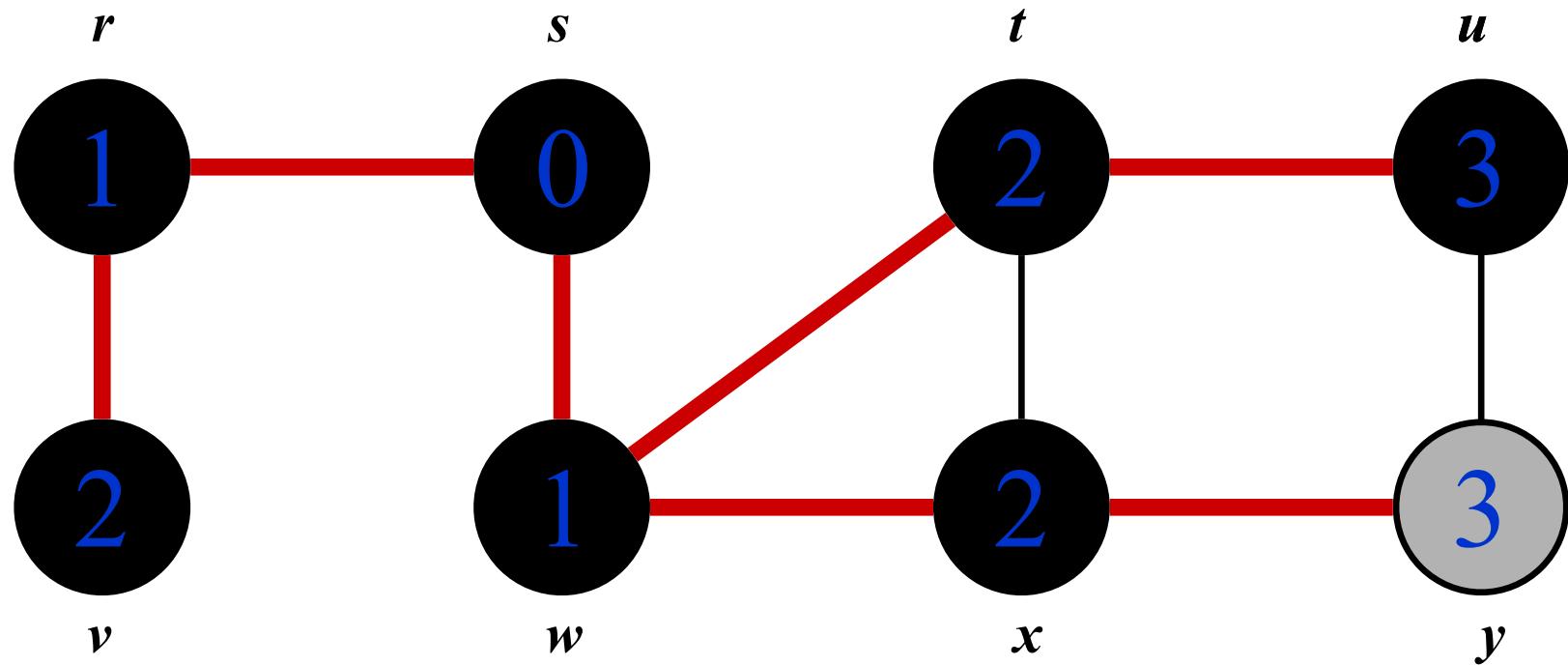
Breadth-First Search (BFS)-Example



$Q:$

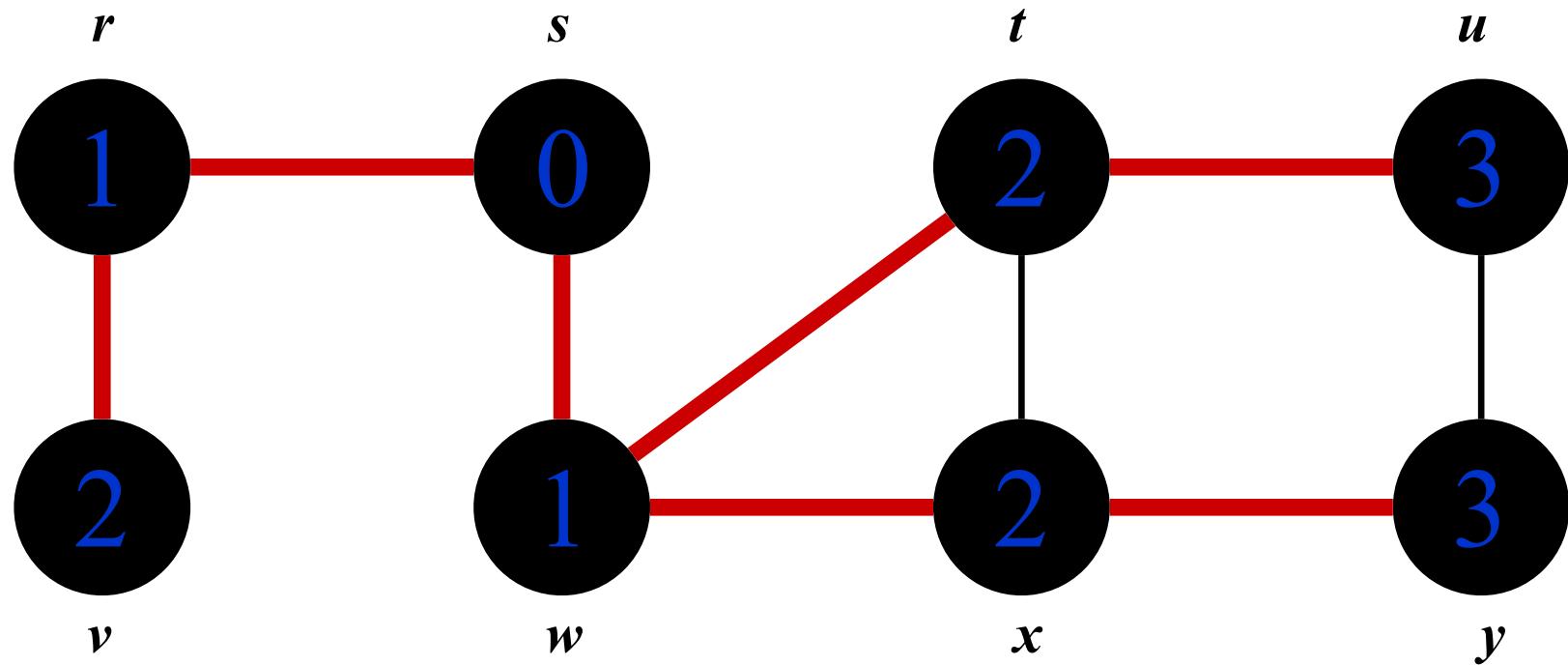
u	y
-----	-----

Breadth-First Search (BFS)-Example



$Q:$ y

Breadth-First Search (BFS)-Example



$Q:$ \emptyset

BFS Properties

- ▶ BFS calculates a *shortest-path distance* from the source node to all other nodes
 - ▶ Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or ∞ if v not reachable from s
⇒ $d(v) = \delta(s,v)$, see proof in the book
- ▶ BFS builds a *breadth-first tree*
 - ▶ s is the root, $\text{pred}(v)$ is the predecessor/parent of v in breadth-first tree (relative to s)
 - ▶ path from s to v in tree is a shortest path from s to v 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

Again will associate vertex “colors” to guide the algorithm

- ▶ Vertices initially colored white
- ▶ Then colored gray when discovered, not finished
- ▶ Then black when finished

Depth-First Search (DFS)

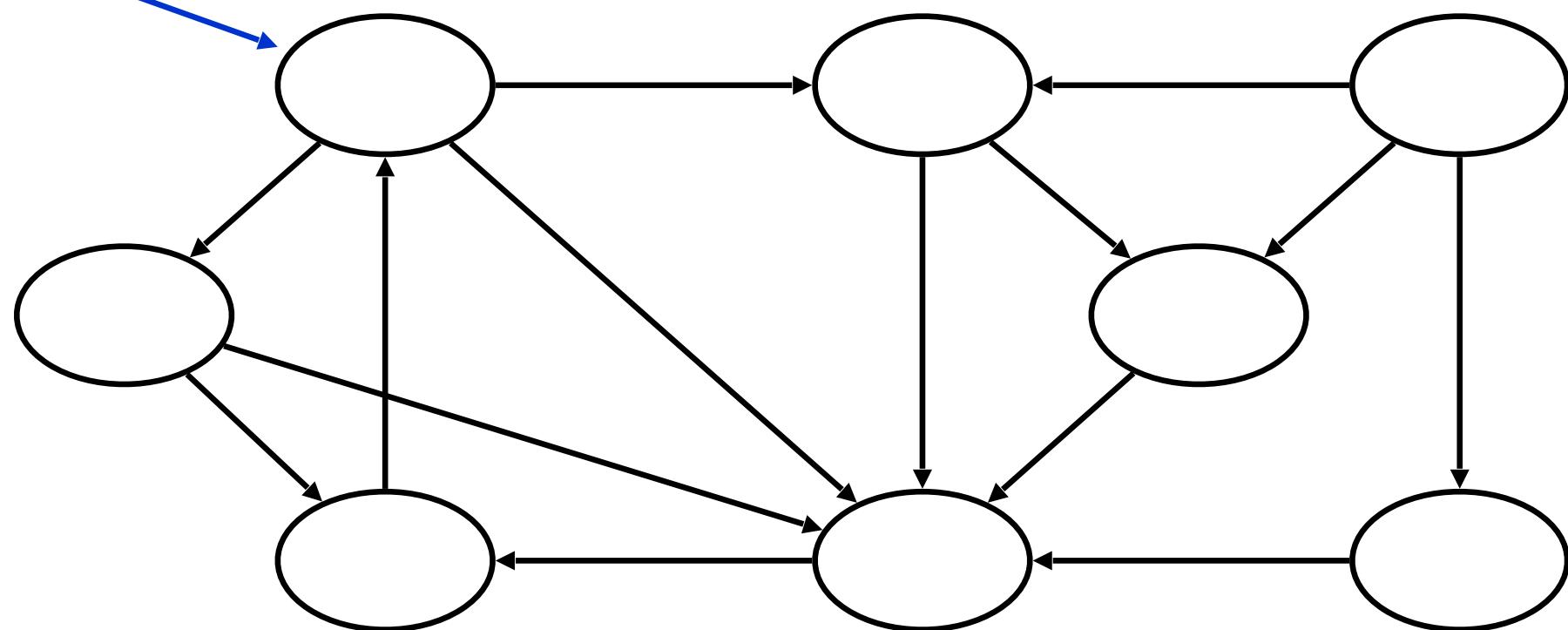
```
DFS(G) {                                // main program
    for each u in V {                   // initialization
        color[u] = white;
        pred[u] = null;
    }
    time = 0;
    for each u in V
        if (color[u] == white)          // found an undiscovered vertex
            DFSVisit(u);               // start a new search here
}

DFSVisit(u) {                           // start a search at u
    color[u] = gray;                   // mark u visited
    d[u] = ++time; update time as well.
    for each v in Adj(u) do
        if (color[v] == white) {       // if neighbor v undiscovered
            pred[v] = u;              // ...set predecessor pointer
            DFSVisit(v);             // ...visit v
        }
    color[u] = black;                 // we're done with u
    f[u] = ++time;
}
```

DFS Example

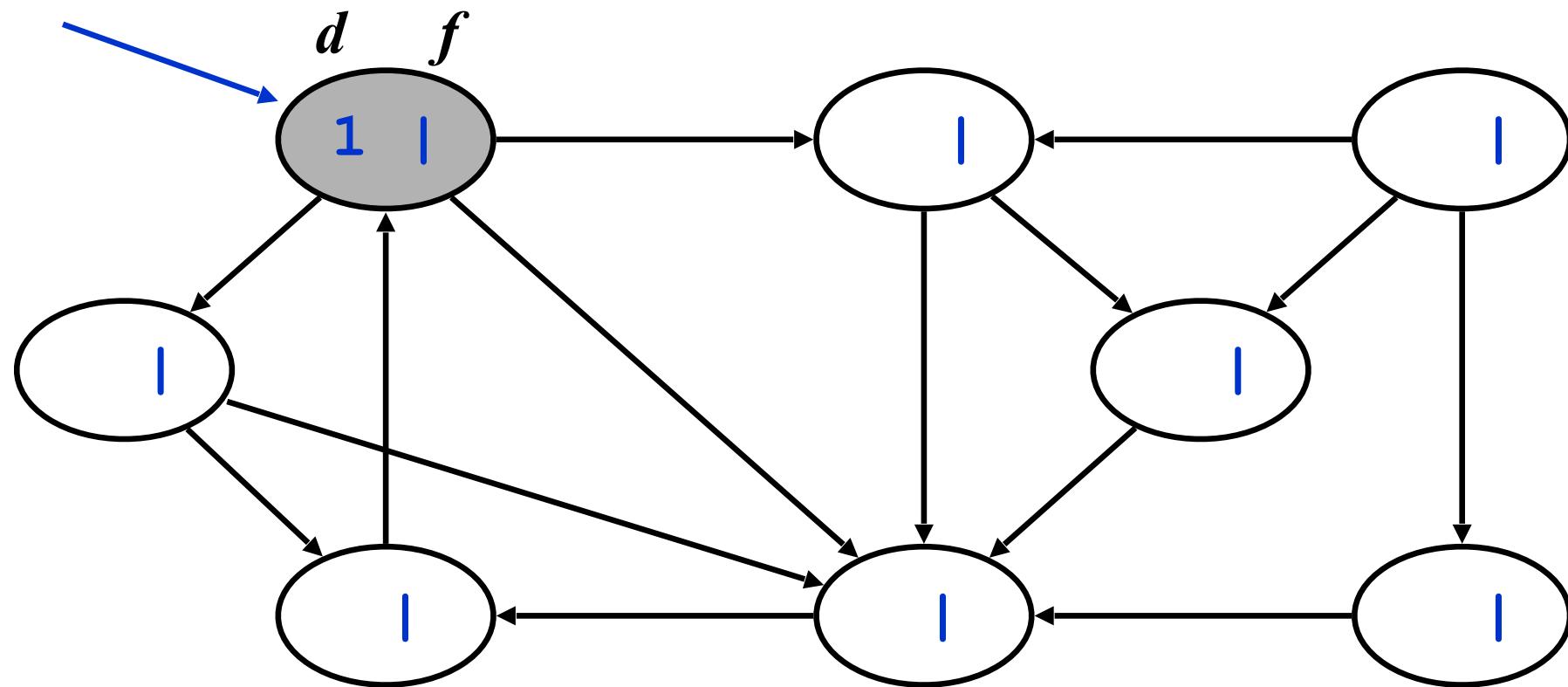
source

vertex



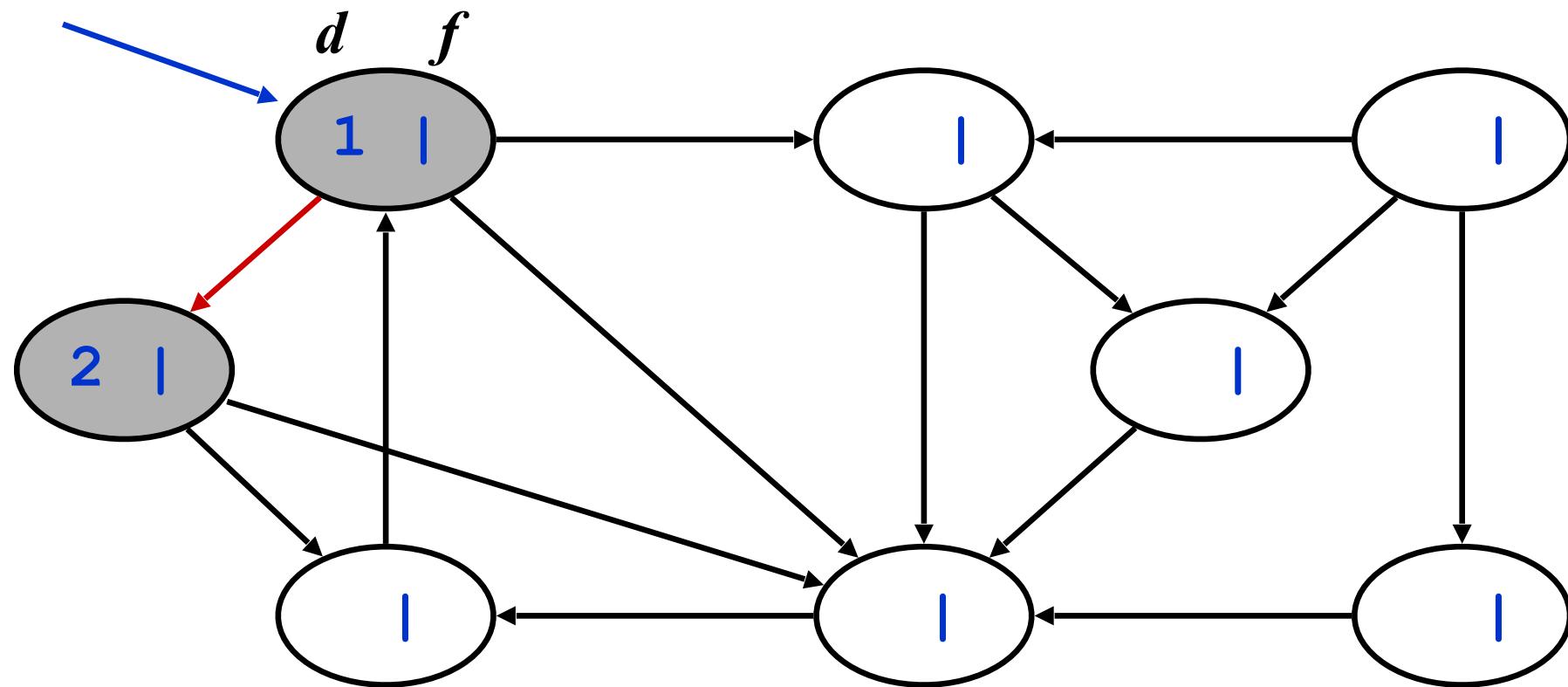
DFS Example

*source
vertex*



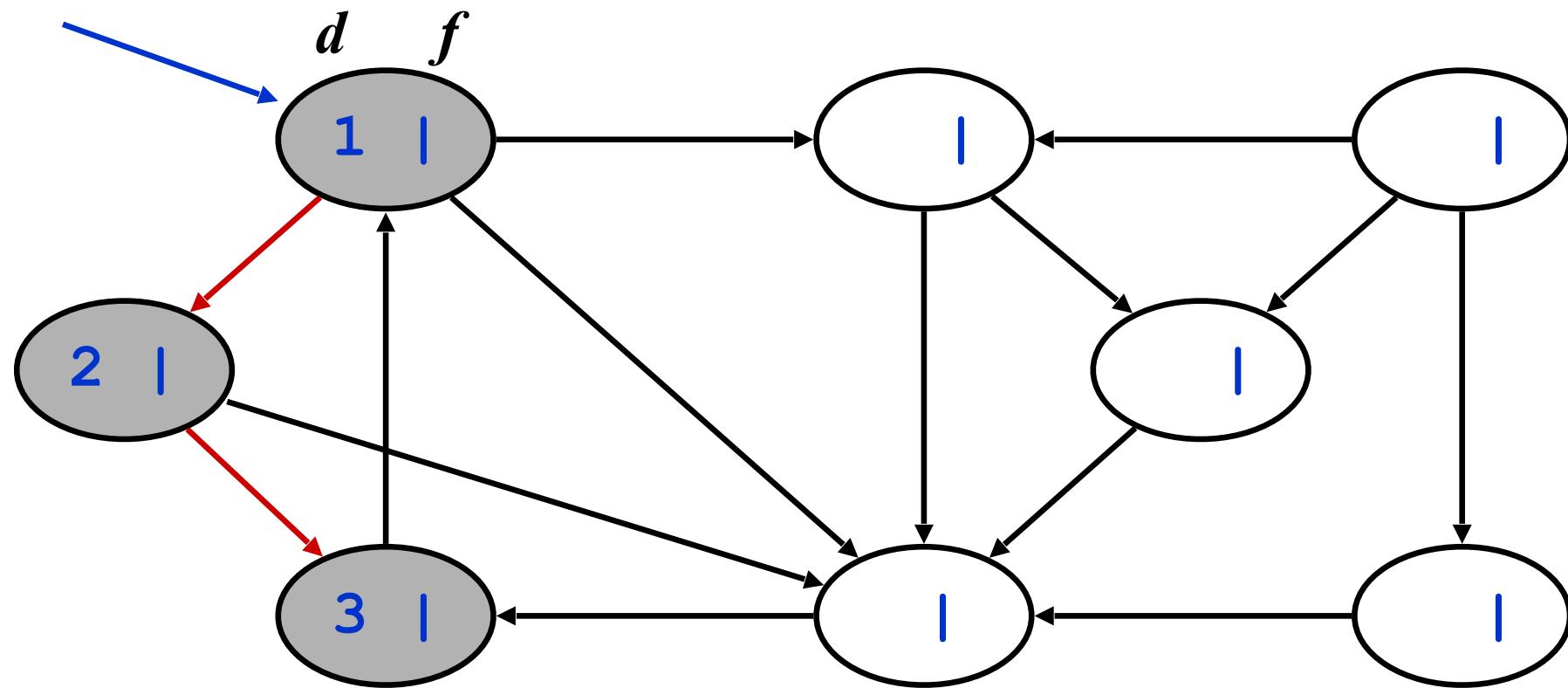
DFS Example

*source
vertex*



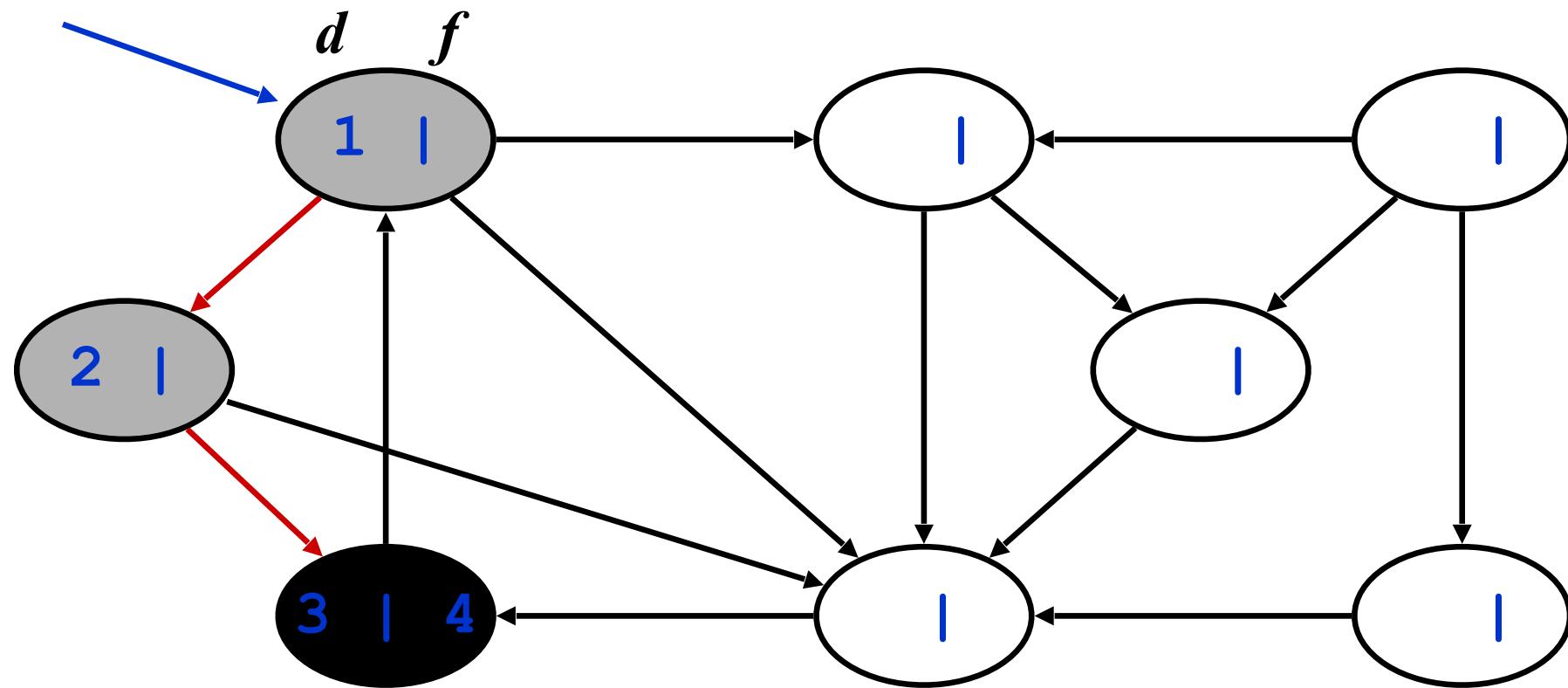
DFS Example

*source
vertex*



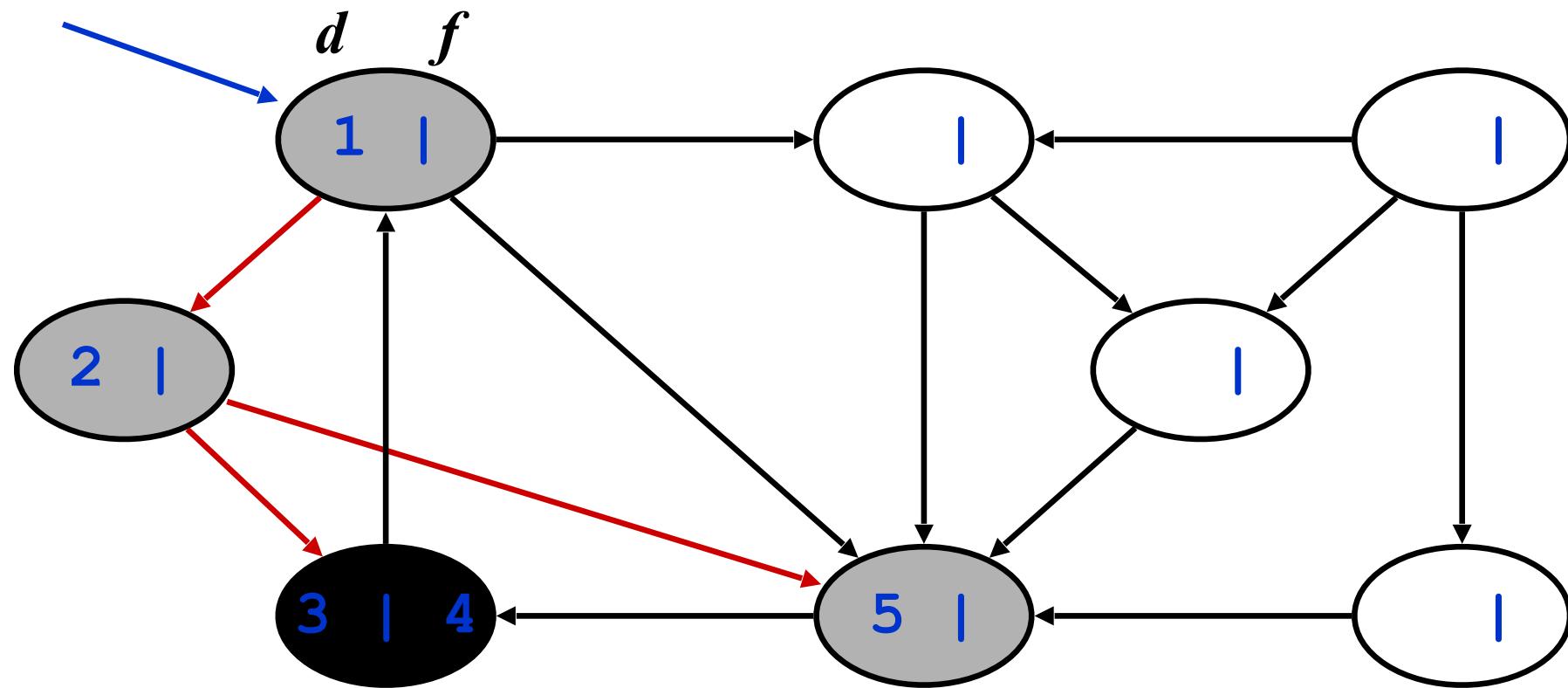
DFS Example

*source
vertex*



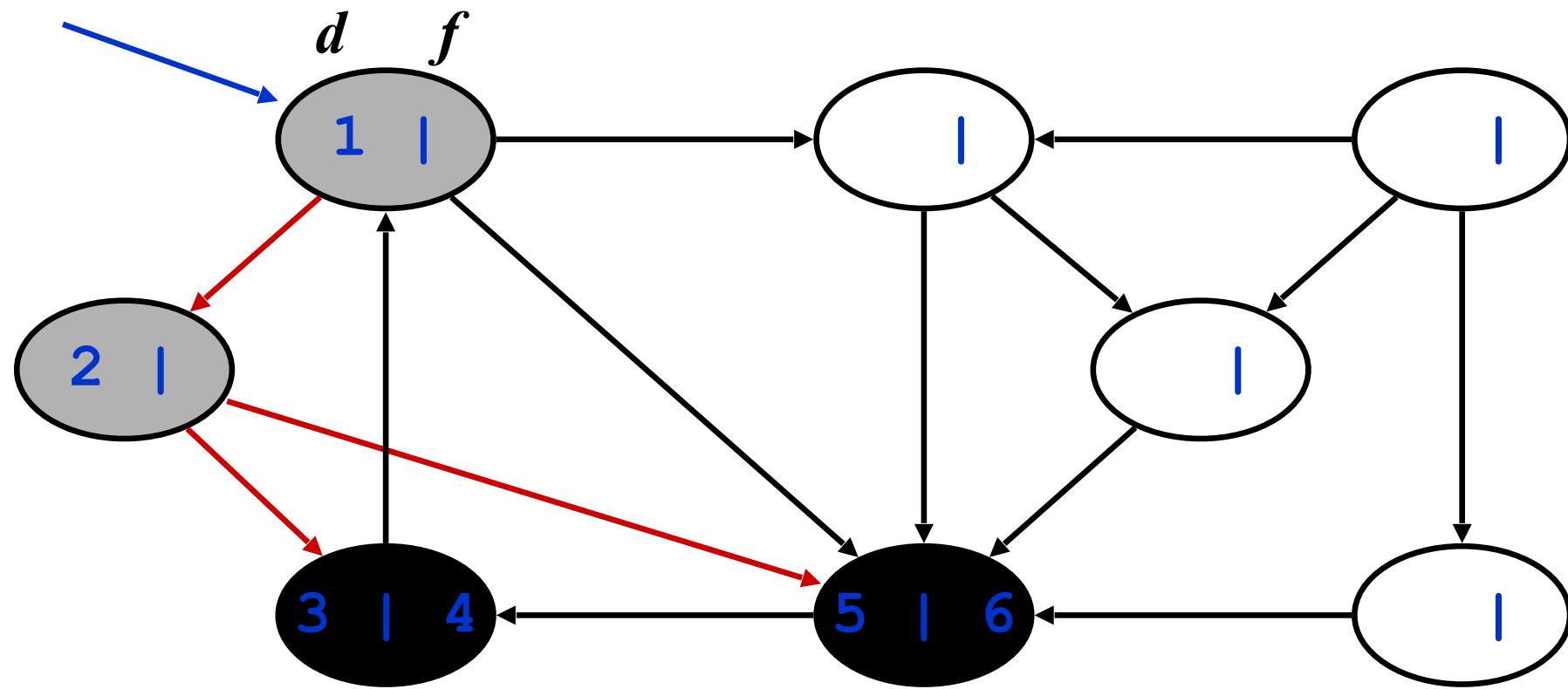
DFS Example

*source
vertex*



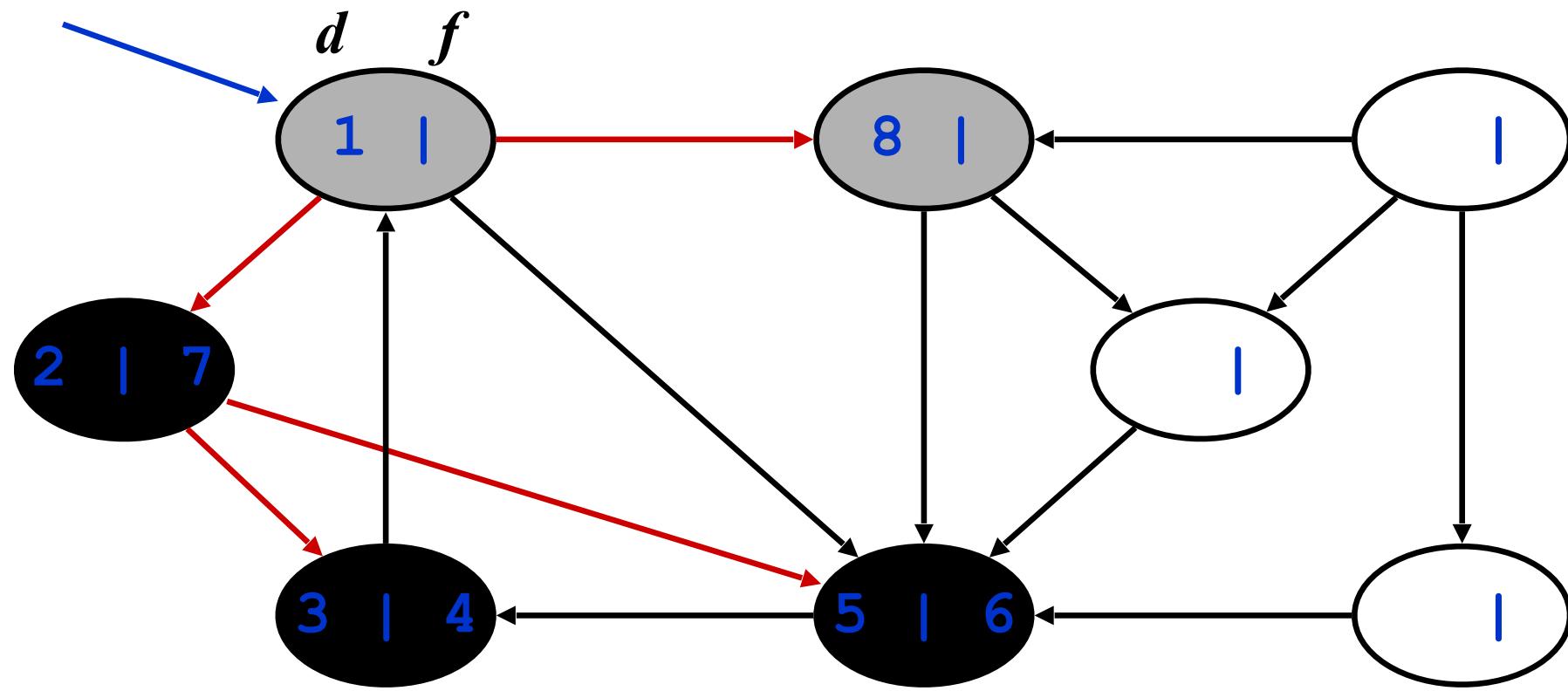
DFS Example

*source
vertex*



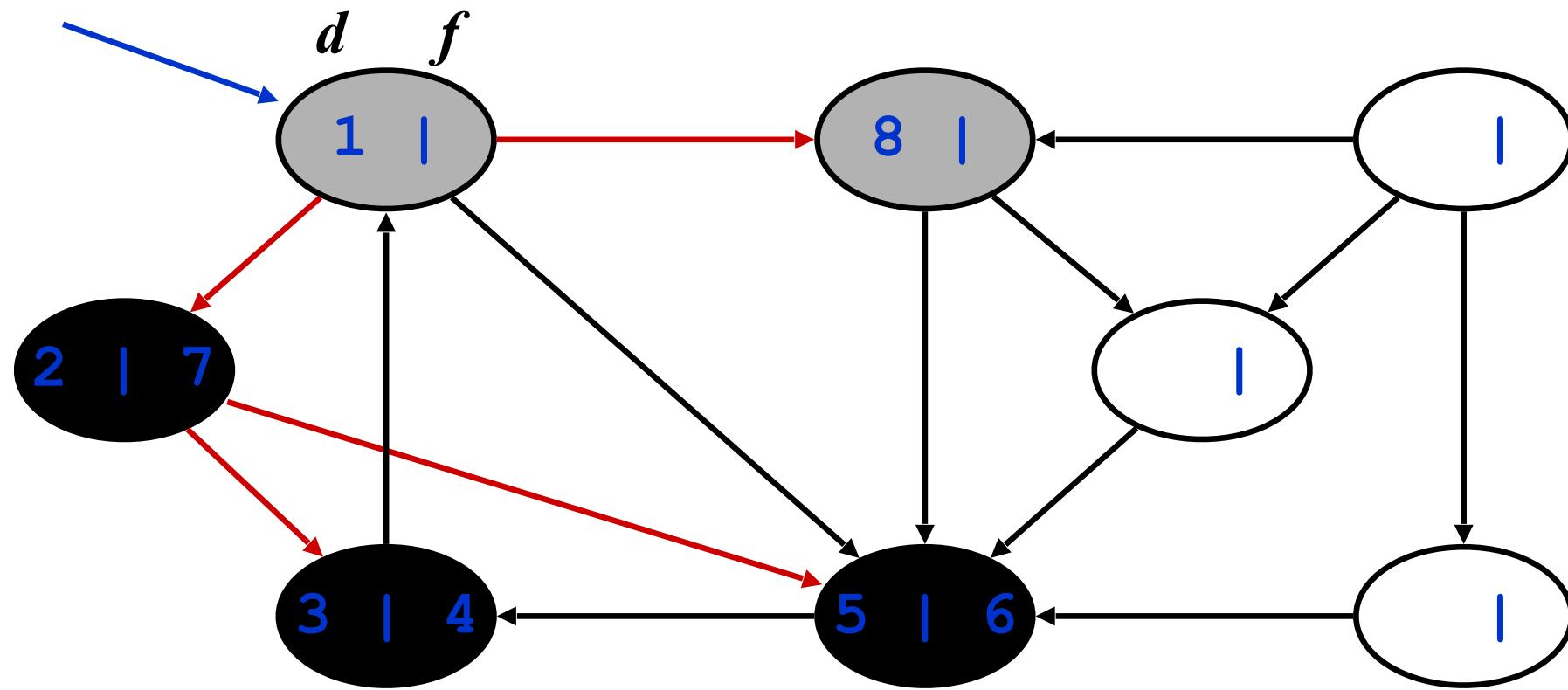
DFS Example

*source
vertex*



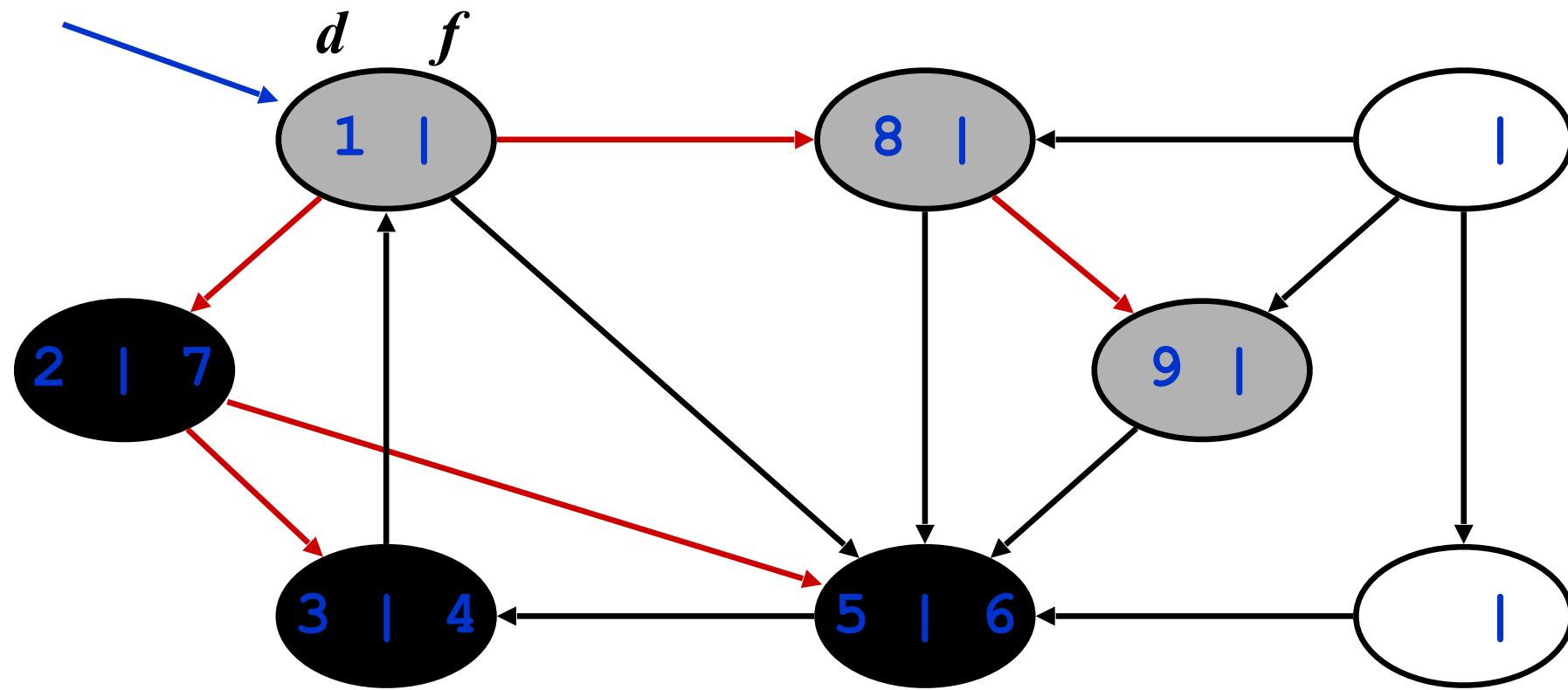
DFS Example

*source
vertex*



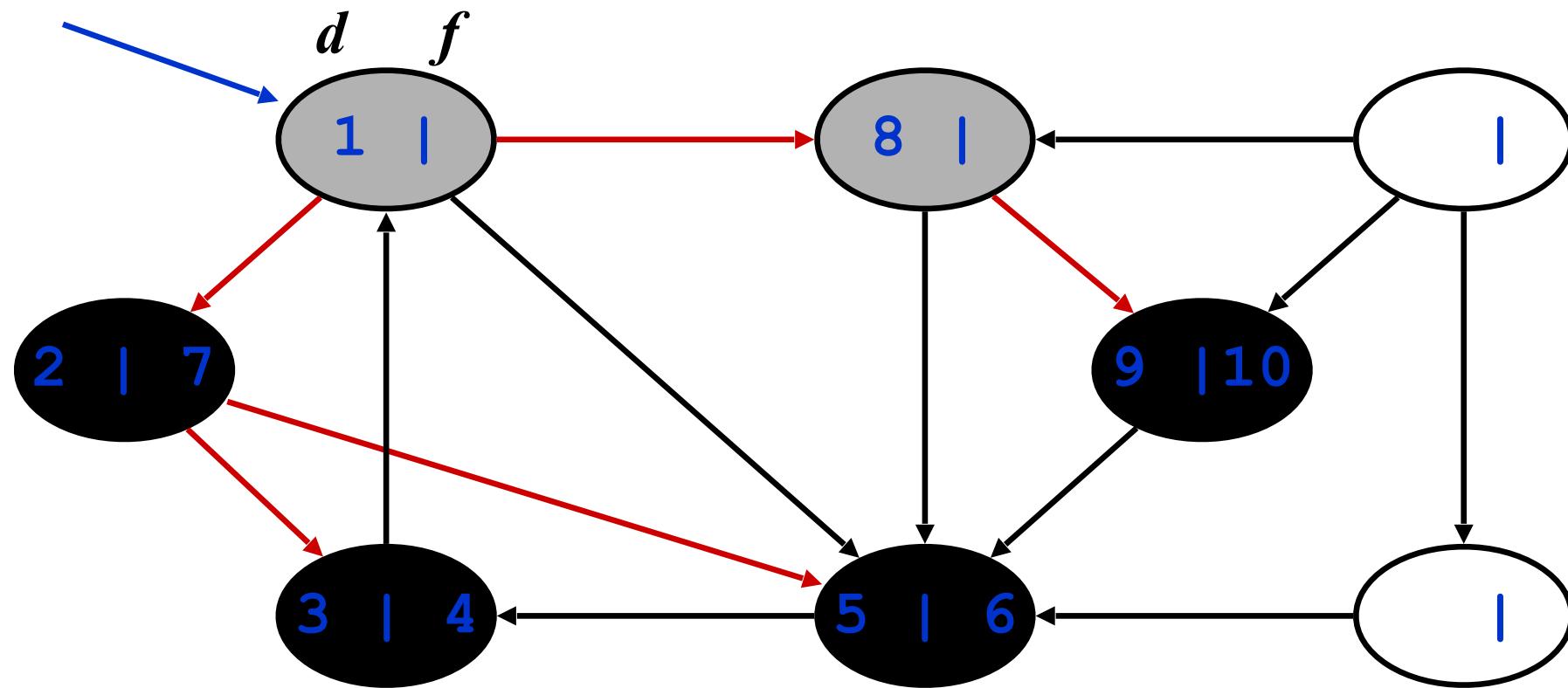
DFS Example

*source
vertex*



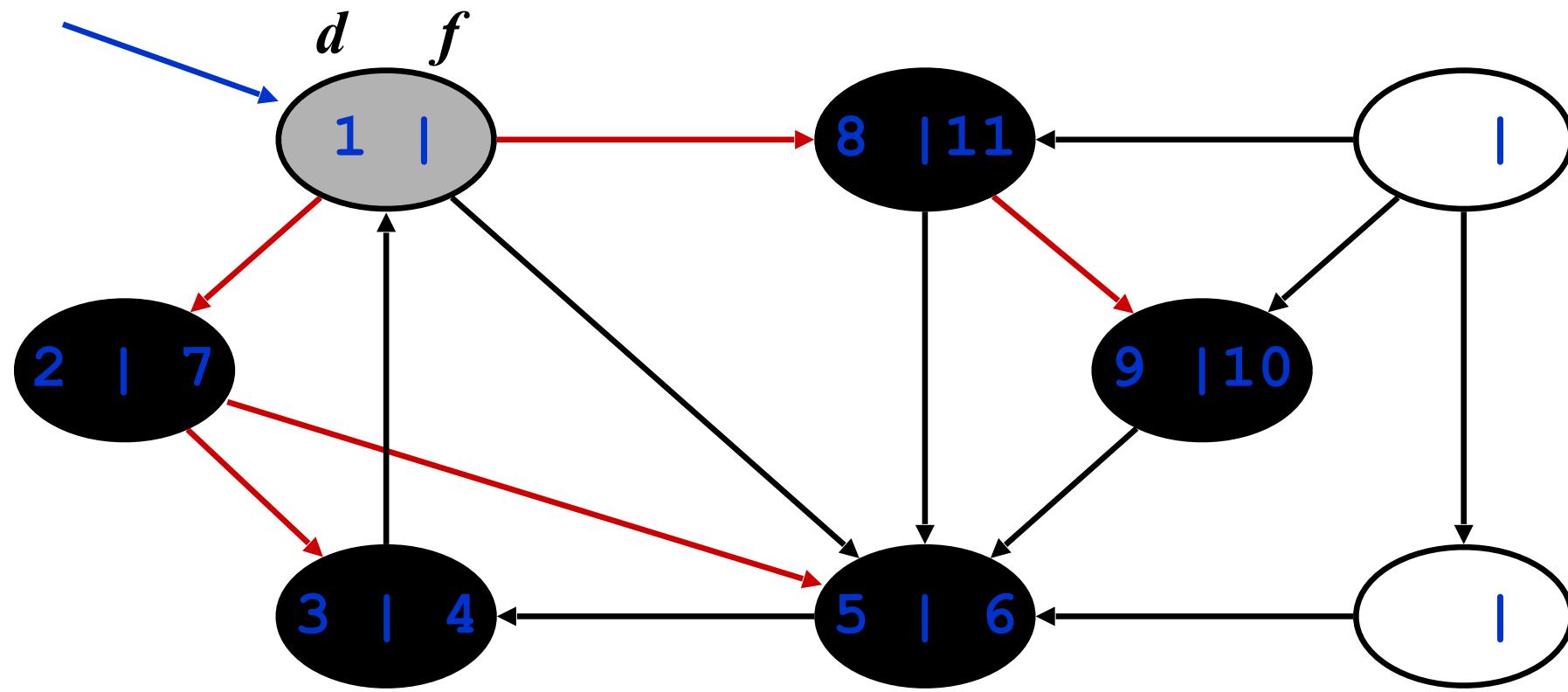
DFS Example

*source
vertex*



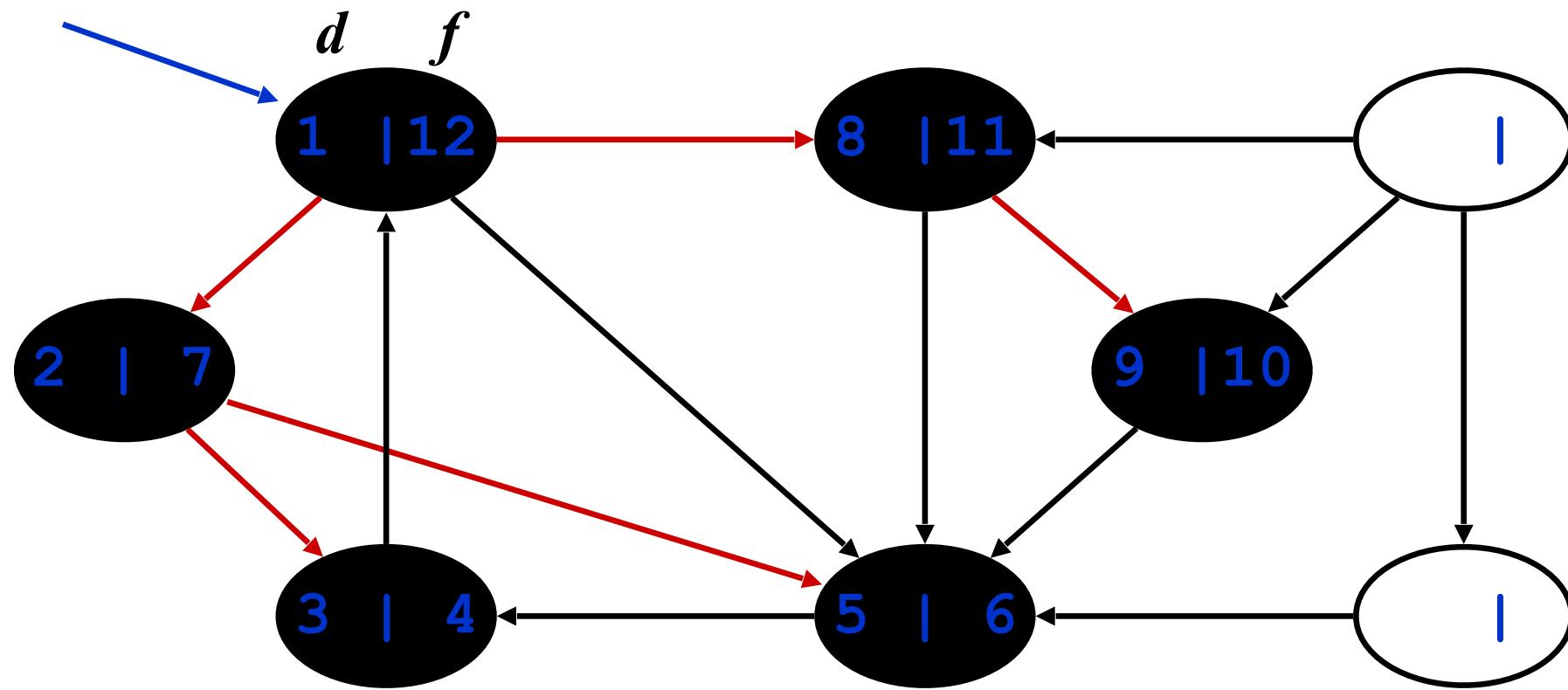
DFS Example

*source
vertex*



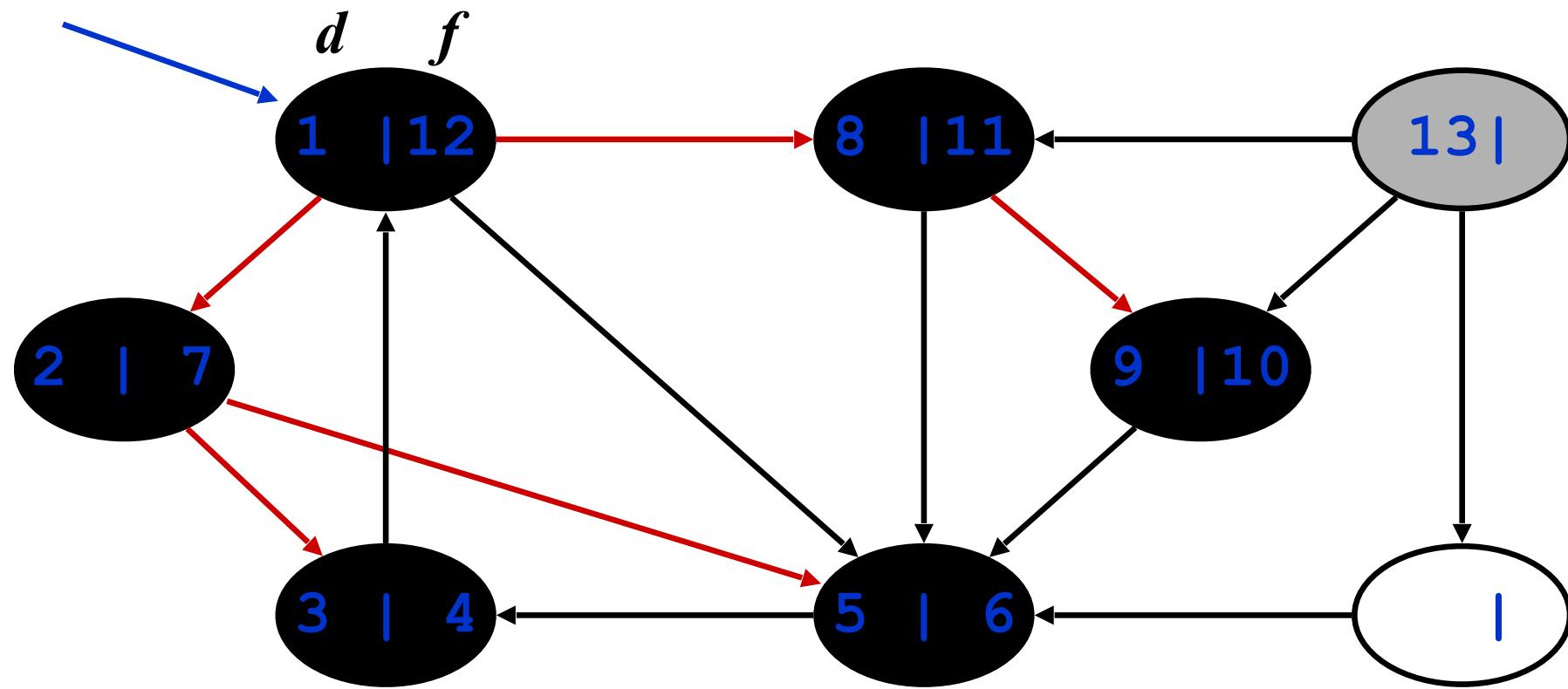
DFS Example

*source
vertex*



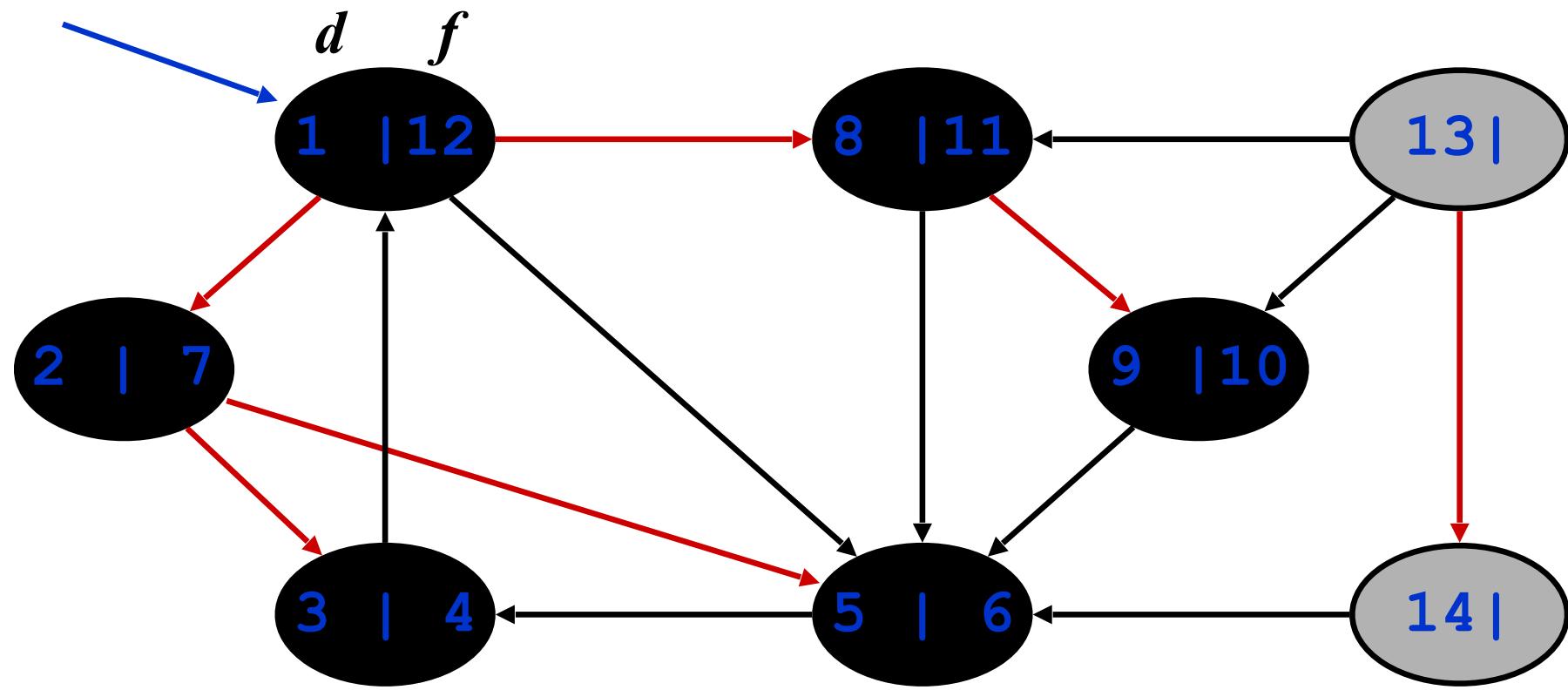
DFS Example

*source
vertex*



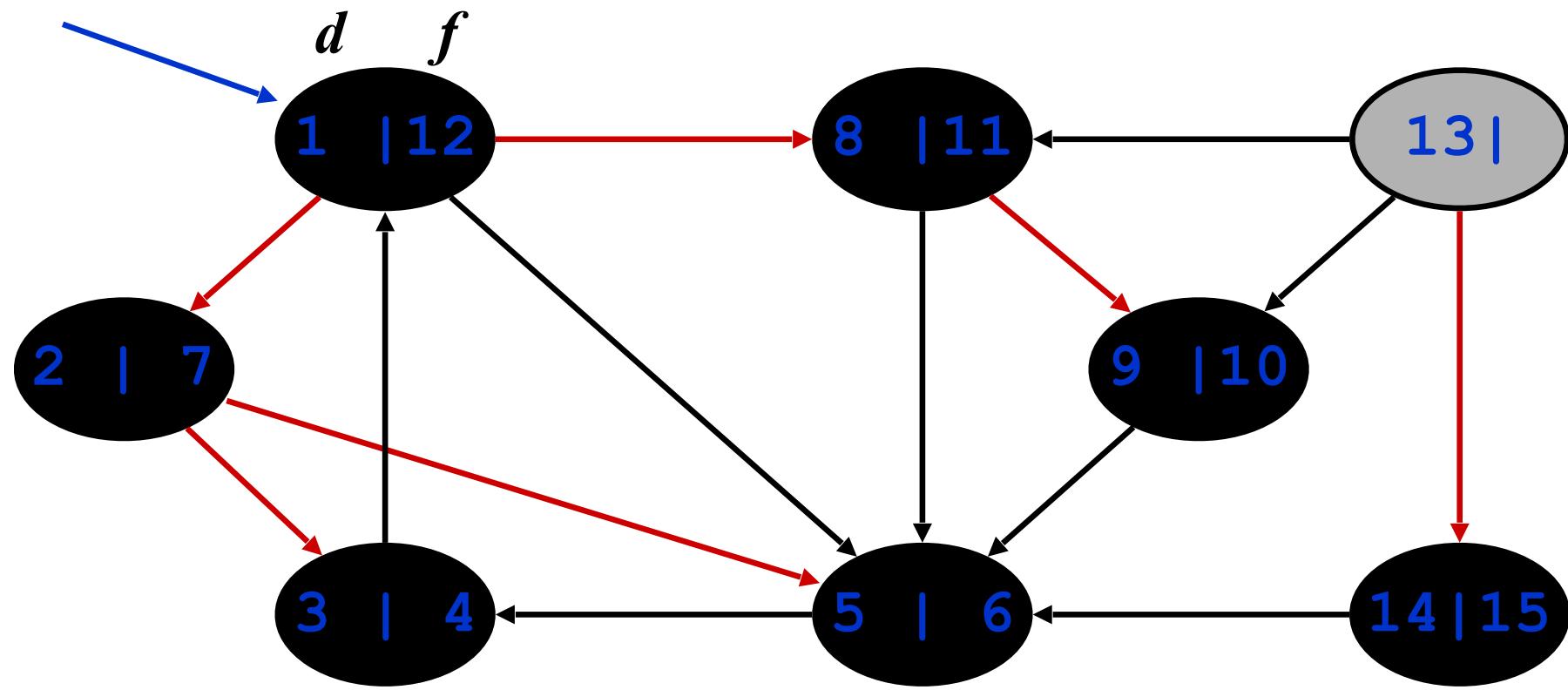
DFS Example

*source
vertex*



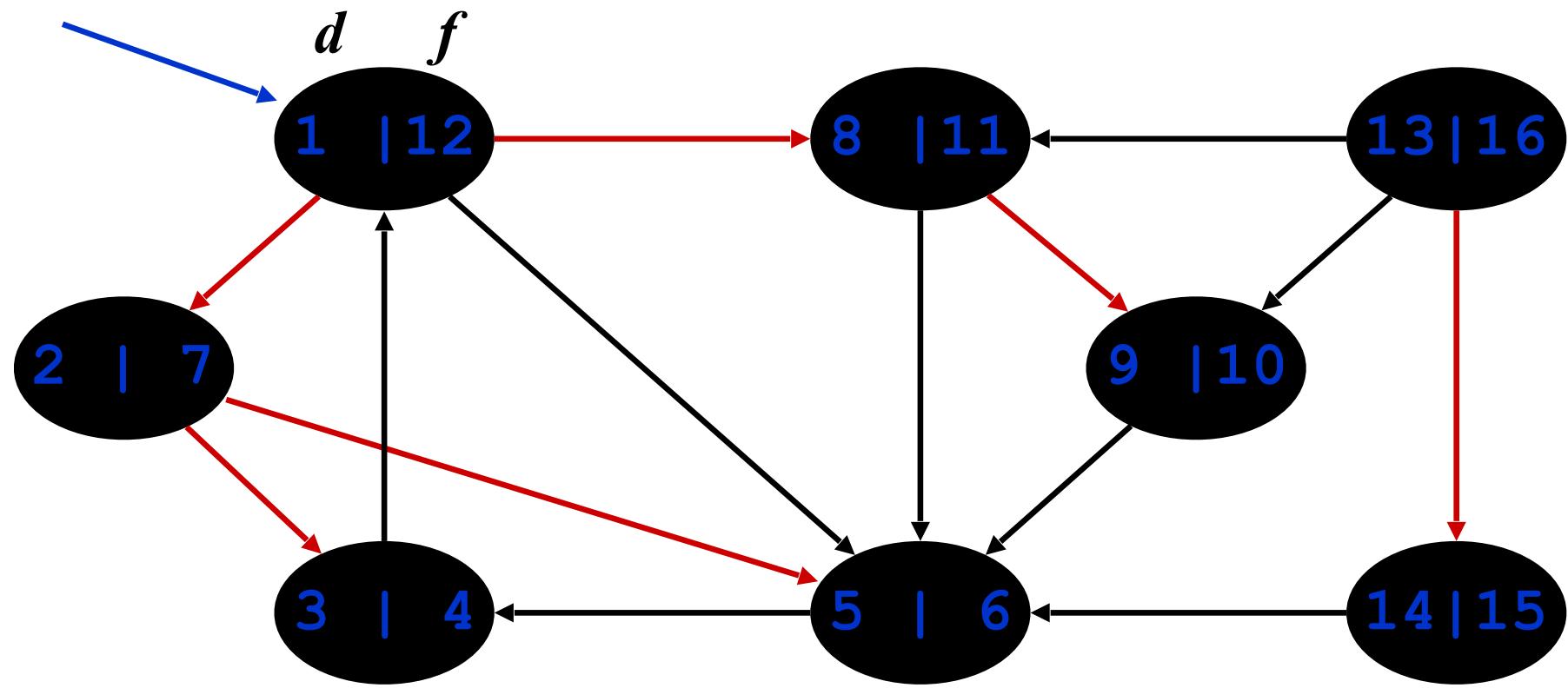
DFS Example

*source
vertex*



DFS Example

*source
vertex*



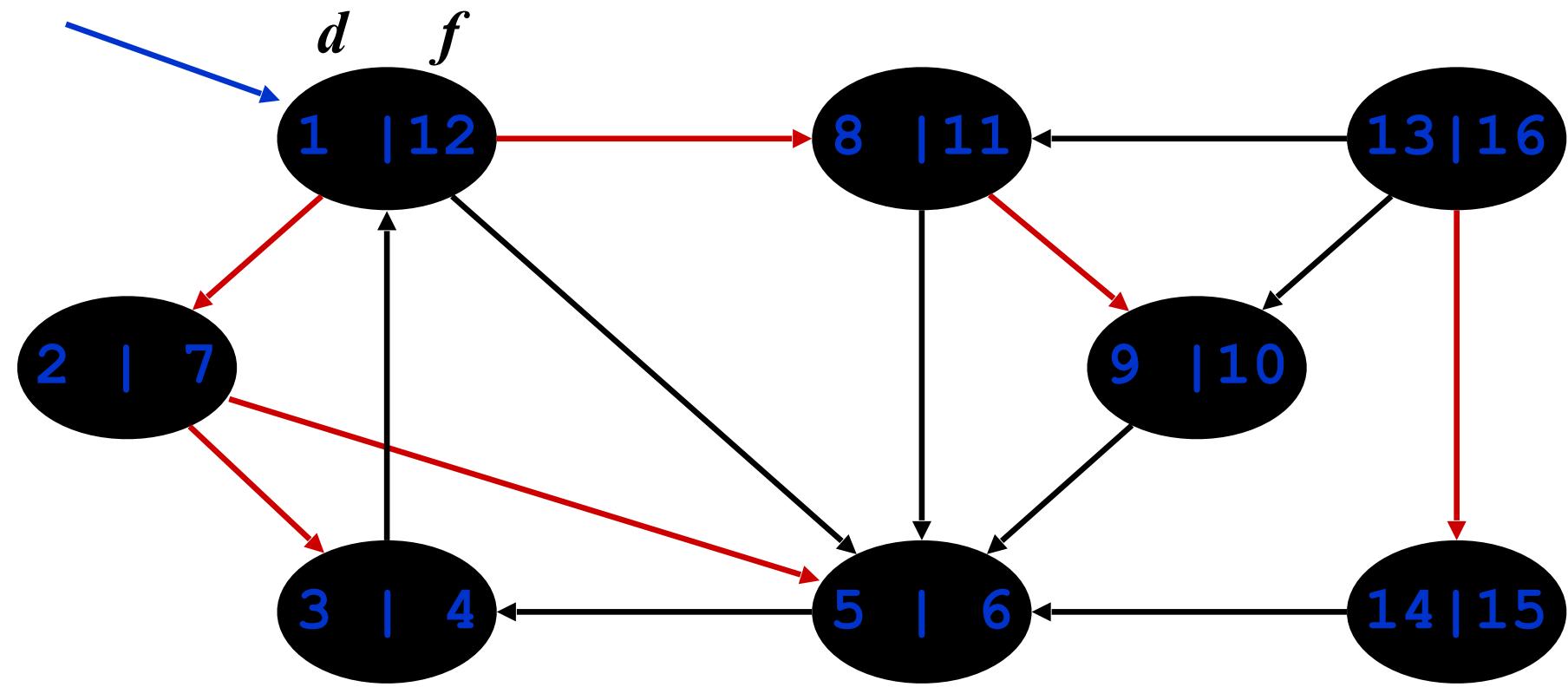
DFS: Kinds of Edges

- ▶ DFS introduces an important distinction among edges in the original graph:
 - ▶ *Tree edge*: encounter new (white) vertex
 - ▶ The tree edges form a spanning forest, called depth-first forest consisting of depth-first trees
 - $\text{pred}(v)$ is the parent of v in its depth-first tree

```
DFSVisit(u) {  
    color[u] = gray;  
    d[u] = ++time;  
    for each v in Adj(u) do  
        if (color[v] == white) {  
            pred[v] = u;  
            DFSVisit(v);  
        }  
    color[u] = black;  
    f[u] = ++time;  
}
```

DFS Example

*source
vertex*

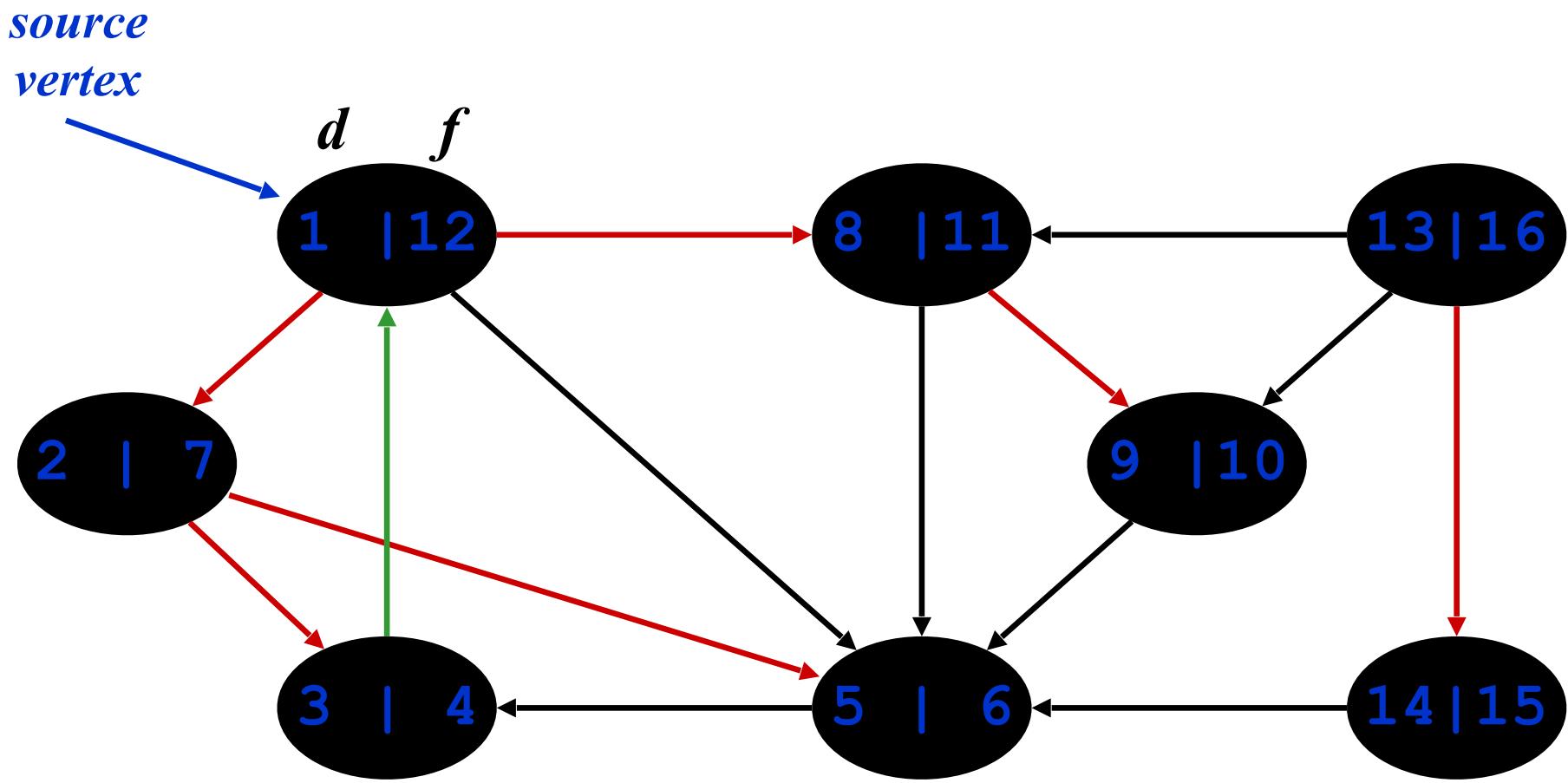


Tree edges

DFS: Kinds of Edges

- ▶ DFS introduces an important distinction among edges in the original graph:
 - ▶ *Tree edge*: encounter new (white) vertex
 - ▶ *Back edge*: from descendent to ancestor (w.r. depth-first tree)
 - ▶ Encounter a grey vertex (grey to grey)

DFS Example



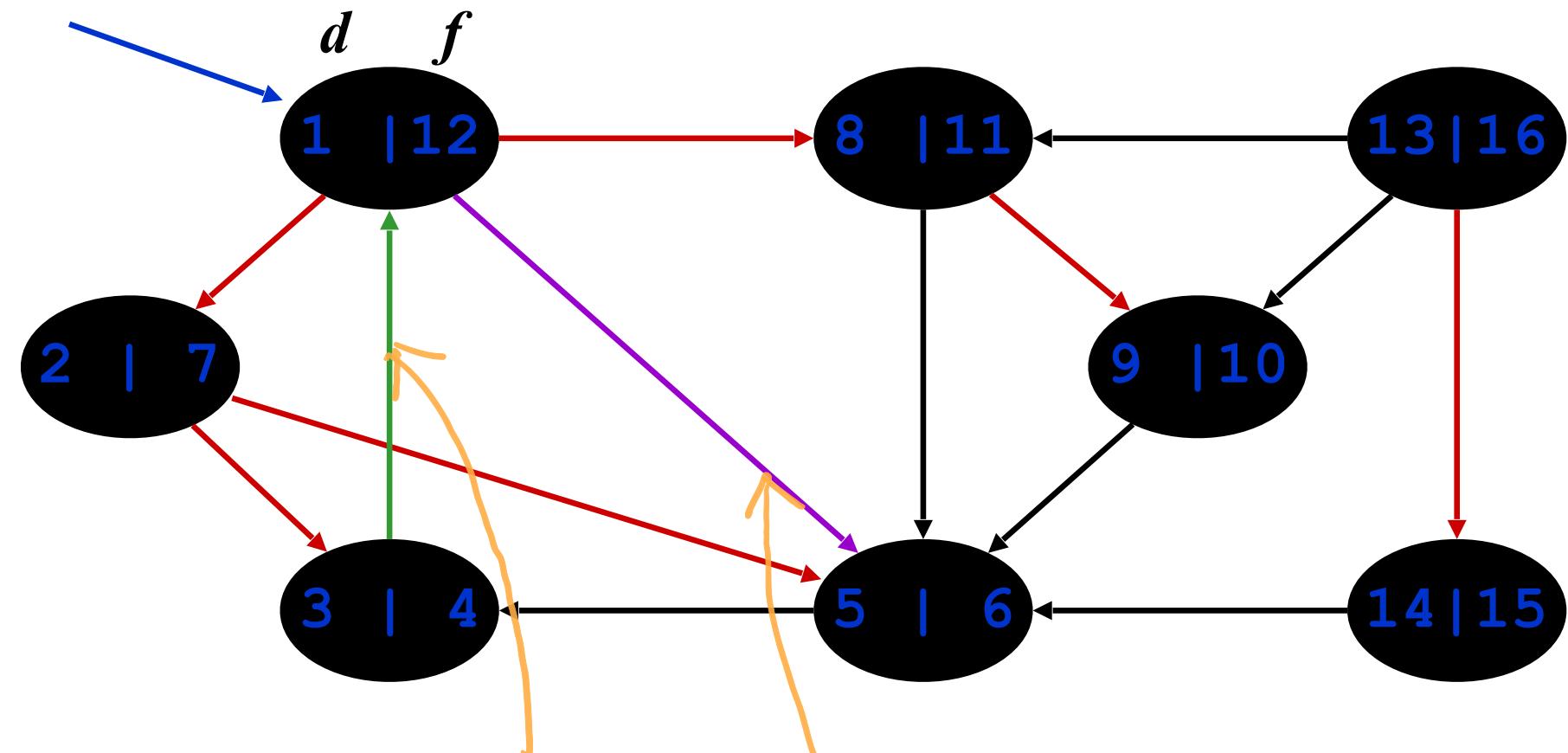
Tree edges Back edges

DFS: Kinds of Edges

- ▶ DFS introduces an important distinction among edges in the original graph:
 - ▶ *Tree edge*: encounter new (white) vertex
 - ▶ *Back edge*: from descendent to ancestor (w.r. depth-first tree)
 - ▶ *Forward edge*: from ancestor to descendent (w.r. depth-first tree)
 - ▶ not a tree edge, though
 - ▶ from grey node to black node

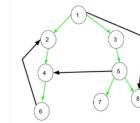
DFS Example

*source
vertex*



Tree edges Back edges Forward edges color!!

DFS: Kinds of Edges



Attention reader! Don't stop learning now. Get hold of all the important DSA concepts with the [DSA Self Paced Course](#) at a student-friendly price and become industry ready. To complete your preparation from learning a language to DS Algo and many more, please refer [Complete Interview Preparation Course](#).

In case you wish to attend live classes with experts, please refer [DSA Live Classes for Working Professionals](#) and [Competitive Programming Live for Students](#).

Tree Edge: It is an edge which is present in the tree obtained after applying DFS on the graph. All the Green edges are tree edges.

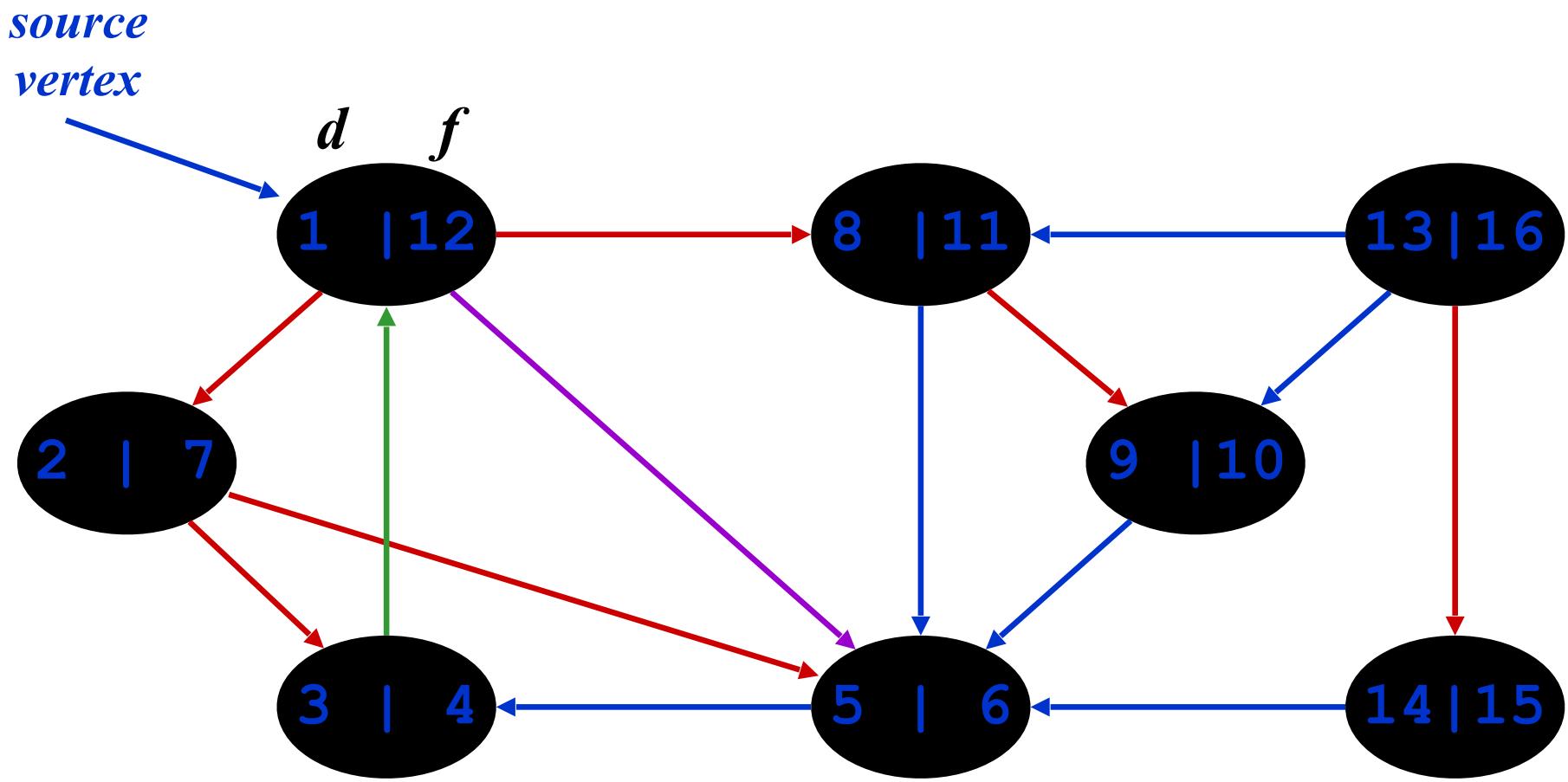
Forward Edge: It is an edge (u, v) such that v is descendant but not part of the DFS tree. Edge from 1 to 3 is a forward edge.

Back edge: It is an edge (u, v) such that v is ancestor of node u but not part of DFS tree. Edge from 4 to 2 is a back edge. [Presence of back edge indicates a cycle in directed graphs](#)

Cross Edge: It is an edge which connects two nodes such that they do not have any ancestor and a descendant relationship between them. Edge from node 5 to 4 is cross edge.

- ▶ DFS introduces an important distinction among edge (u,v) in the original graph:
 - ▶ **Tree edge:** encounter new (white) vertex, edge from parent to child in depth-first tree
 - ▶ v is white color when (u,v) is first explored
 - ▶ **Back edge:** from descendant to ancestor in depth-first tree
 - ▶ v is gray color when (u,v) is first explored
 - ▶ **Forward edge:** from ancestor to descendant in depth-first tree *(not directed child)*.
 - ▶ v is black color when (u,v) is first explored
 - ▶ **Cross edge:** between two nodes w/o ancestor-descendant relation in a depth-first tree *or two nodes in two different depth-first trees*
 - ▶ v is black color when (u,v) is first explored
- ▶ Note: tree & back edges are important; most algorithms don't distinguish forward & cross

DFS Example



Tree edges Back edges Forward edges Cross edges

Parenthesis Theorem ①

Theorem: In any DFS of directed or undirected graph, for any two vertices u and v , one of the following three conditions holds:

1. intervals $[d(u), f(u)]$ and $[d(v), f(v)]$ are disjoint;
2. $[d(u), f(u)]$ entirely inside $[d(v), f(v)]$, i.e. $d(u) < d(v) < f(v) < f(u)$;
3. $[d(v), f(v)]$ entirely inside $[d(u), f(u)]$, i.e. $d(v) < d(u) < f(u) < f(v)$.

case 4. $d(v) < d(u) < f(v) < f(u)$ → not exist.

~~Proof:~~ Without loss of generality (w.l.o.g), assume u is discovered first, $d(u) < d(v)$. There are two subclasses:

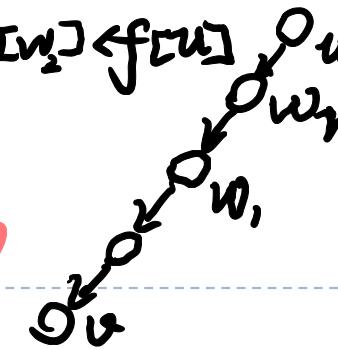
- a. $d(v) < f(u)$: v was discovered while u was still gray $\Rightarrow v$ is a descendant of u in DFS tree, search will return to u after all outgoing edge of v are explored, so that $f(v) < f(u)$, this is case 2 in the theorem.
- b. $d(v) > f(u)$: v was discovered after u was fully explored, case 1 in theorem.

Corollary: Nesting of Descendant's Intervals

v is a proper descendant of u in DFS forest if and only if $d(u) < d(v) < f(v) < f(u)$

$$d(u) < d[w_2] < f[w_2] < f[u], \text{ so } u, d[u]$$

White-path Theorem ②



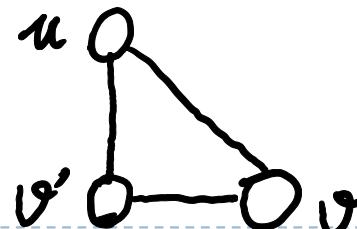
Theorem: Vertex v is a descendant of u in a DFS tree **if and only if** at time $d(u)$ that u was discovered, vertex v can be reached from u along a path consisting entirely of white vertices.

~~Proof: => (only if), let w be any vertex on the path between u and v in DFS tree, w is a descendant of u , then according to Parenthesis theorem, $d[w] > d[u]$, i.e., w was white at time $d[u]$. The path between u and v in DFS tree corresponds to a white path from u to v in the original graph.~~

~~<= (if) let p be the white path at time $d(u)$, w.l.o.g., let w_1 be the node which is the closest to u on p but not a descendant of u in DFS, let w_2 be the predecessor of w_1 on p , then w_2 is a descendant of u in DFS, according to Nesting Corollary, $d(u) < d(w_2) < f(w_2) < f(u)$, since (w_2, w_1) is an edge in G , w_1 will be discovered before w_2 is finished, so $d(u) < d(w_1) < f(w_2) < f(u)$, according to Parenthesis theorem, $d(u) < d(w_1) < f(w_1) < f(u)$, w_1 is a descendant of u in DFS according to Nesting Corollary.~~

DFS in Undirected Graph

- ▶ Theorem: In a depth-first search of an undirected graph G , every edge of G is either a tree edge or a back edge.
- ▶ Proof: for any (u,v) , w.l.o.g, assume $d(u) < d(v)$, then $d(v) < f(v) < f(u)$. (white path theorem)
 - 1) if the first time (u,v) is processed, it is from u's adjacency list, then v must not have been discovered (v is white), then (u,v) is a tree edge.
 - 2) if the first time (u,v) is processed, it is from v's adjacency list, then u is still gray, then (u,v) is a back edge.



DFS & Graph Cycle: undirected

- ▶ 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

DFS & Graph Cycle: undirected

- ▶ *What will be the running time?*
- ▶ A: $O(V+E)$
- ▶ We can actually determine if cycles exist in $O(V)$ time:
 - ▶ In an undirected acyclic forest, $|E| \leq |V| - 1$
 - ▶ So count the edges: if ever see $|V|$ distinct edges, must have seen a back edge along the way

DFS & Graph Cycle: directed

- ▶ Theorem: A directed graph is *acyclic* iff a DFS yields no back edges
- ▶ Proof: (sketch, details in book)
=> DFS produces a back edge (u,v) , v is an ancestor of u in depth-first tree, then in G there is a path from v to u , then back edge (u,v) completes a cycle
 \leq suppose G has a cycle c , let v be the first vertex to be discovered by DFS, u is the predecessor of v in cycle c , at time $d(v)$, all vertices of c are white, form a white path from v to u , then u becomes a descendant of v in depth-first tree, (white path theorem), (u,v) is a back edge.