Graph Algorithms

Graphs

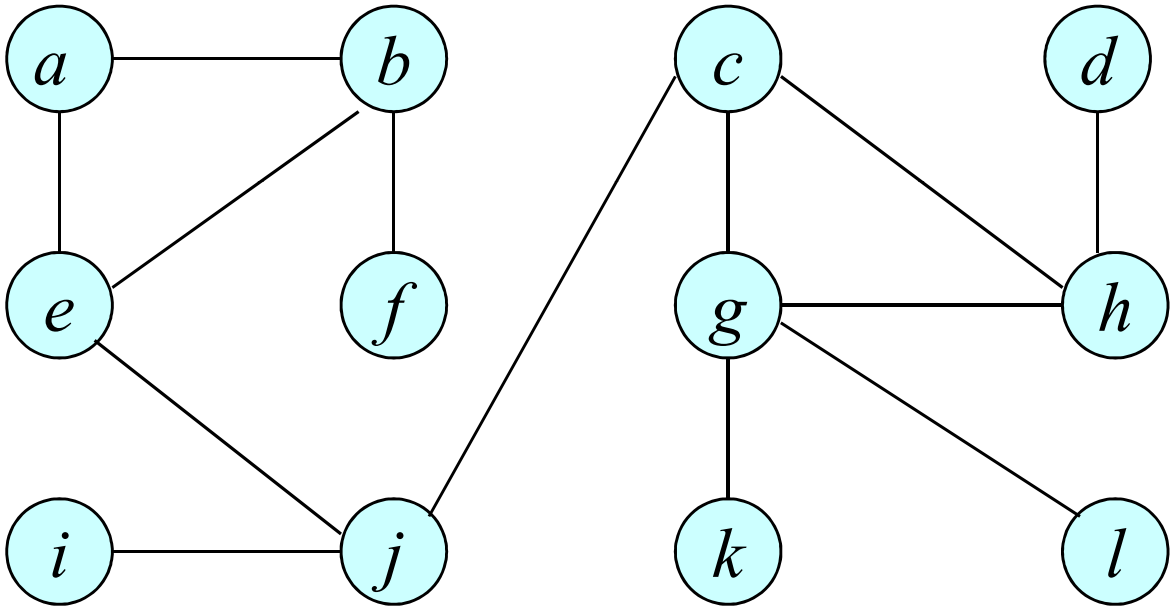
A graph **G = (V, E)** consists of a set of vertices **V**, and a set of edges **E**, such that each edge in **E** is a connection between a pair of vertices in **V**.

The number of vertices is written |**V**| called ***Order***, and the number edges is written |**E**| called ***Size***.

- A graph G = (V, E)
  - V = set of vertices
  - E = set of edges = subset of V × V (unordered pair of vertices)
  - Thus $|E| = O(|V|^2)$

Graphs

A *graph* G is a set $V(G)$ of *vertices* (*nodes*) and a set $E(G)$ of *edges* which are pairs of vertices.

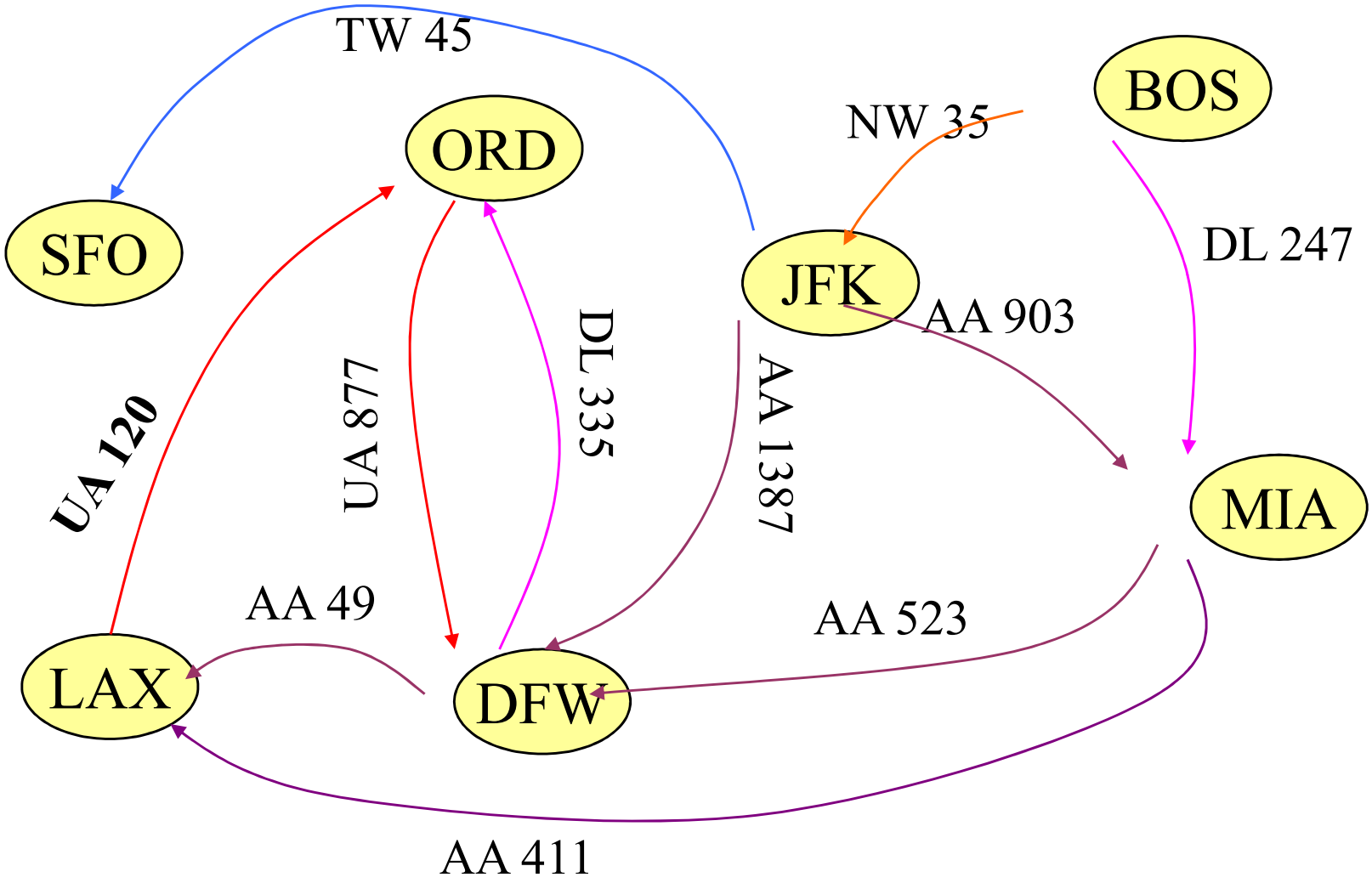


$V = \{ a, b, c, d, e, f, g, h, i, j, k, l \}$

$E = \{ (a, b), (a, e), (b, e), (b, f), (c, j), (c, g), (c, h), (d, h), (e, j), (g, k), (g, l), (g, h), (i, j) \}$

A Directed Graph

In a *directed graph* (*digraph*), edges are *ordered* pairs.



Directed Graph (Con't)

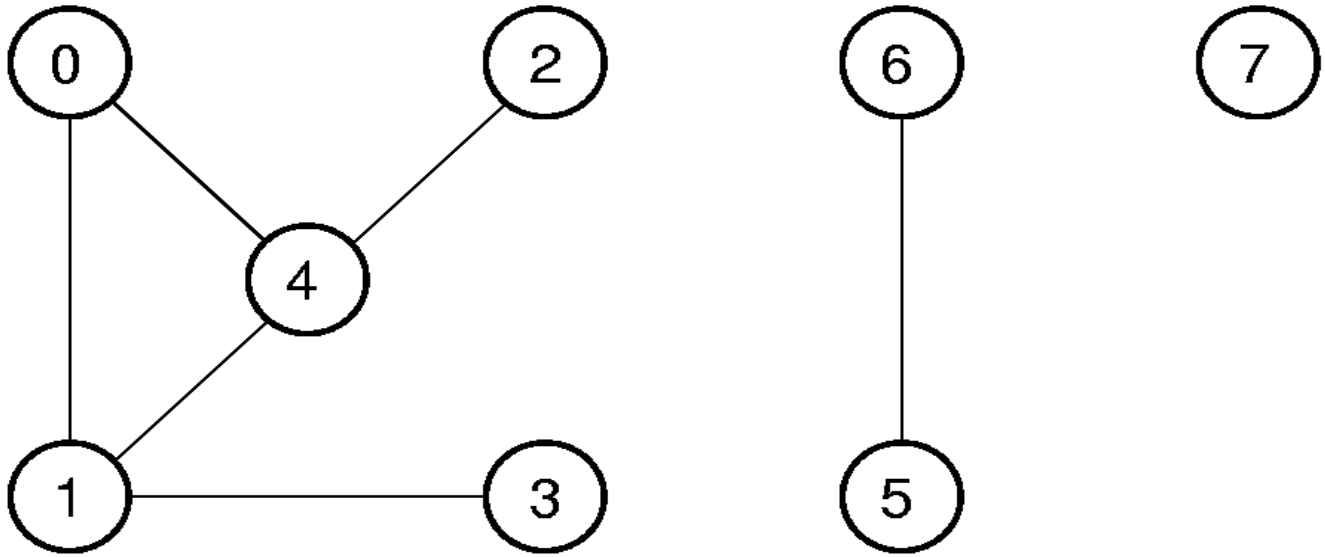
- In a *directed* graph:
  - Edge (u,v) goes from vertex u to vertex v, notated u→v
  - Consist of finite set V of vertices and a set A of ordered pair of distinct vertices called arcs.

- In a Mixed graph there will be at least one edge and at least one arc.

Undirected Graph

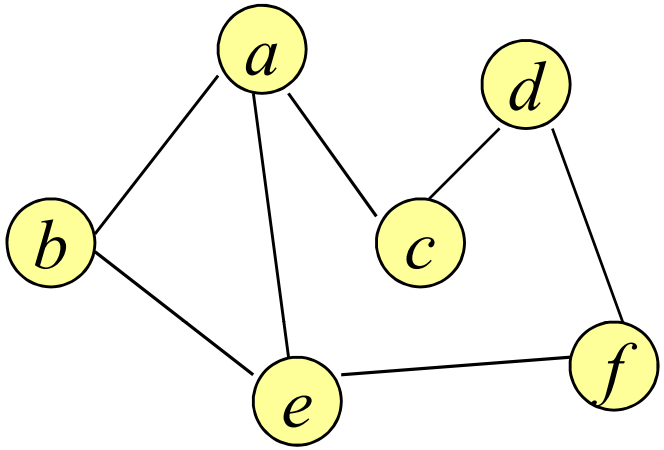
- In an *undirected graph:*
  - Edge (u,v) = edge (v,u)
  - No self-loops

- An undirected graph is connected if there is at least one path from any vertex to any other.

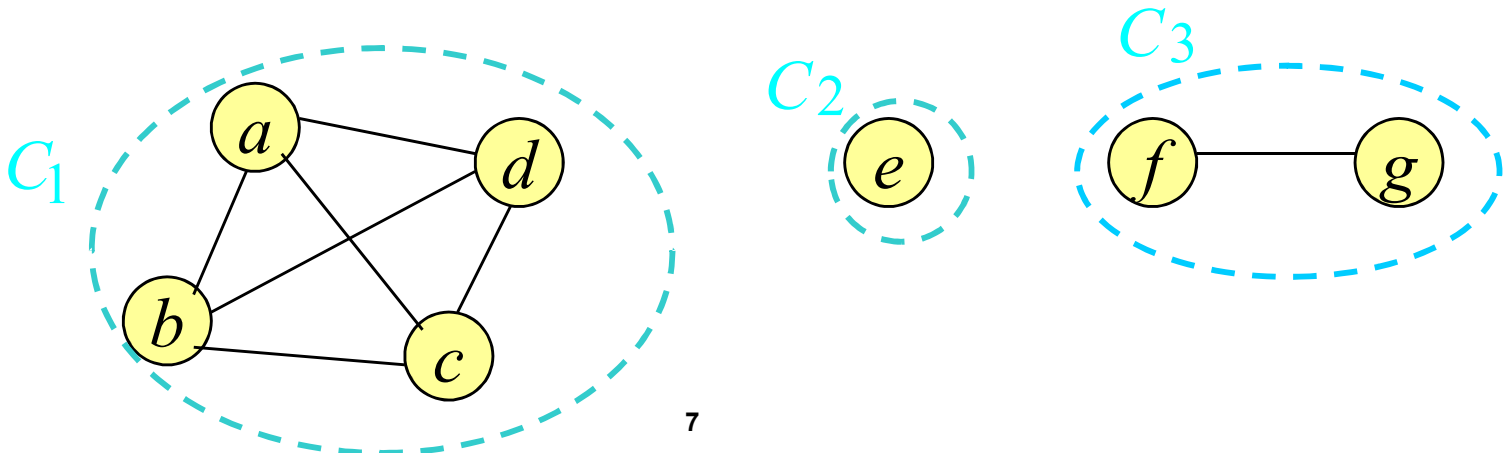


Connected Graph

G is *connected* if there is a path between every pair of vertices.



If *G* is not connected, the maximal connected subgraphs are the *connected components* of *G*.



Connected Graph

- A *connected graph* has a path from every vertex to every other

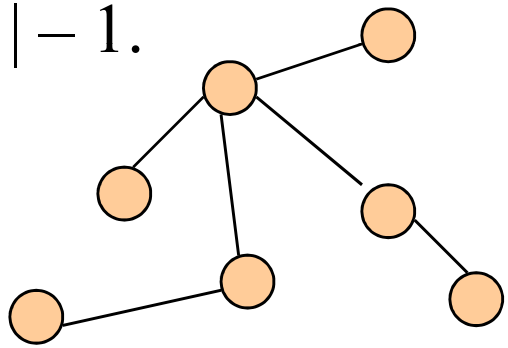
Complete Graph

- **A graph G is said to be complete if every node u in G is adjacent to every other node v in G.**
- **A complete graph with n nodes have n(n-1)/2 edges**

Property of Connectivity

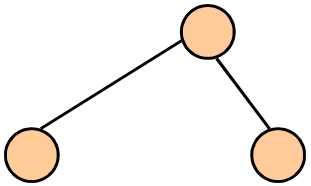
Let G = (V, E) be an undirected graph.

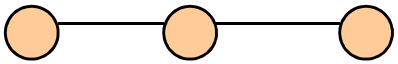
- ☆ If G is connected, then $|E| \geq |V| - 1$.
- ☆ If G is a tree, then $|E| = |V| - 1$.



- ☆ If G is a forest, then $|E| \leq |V| - 1$.

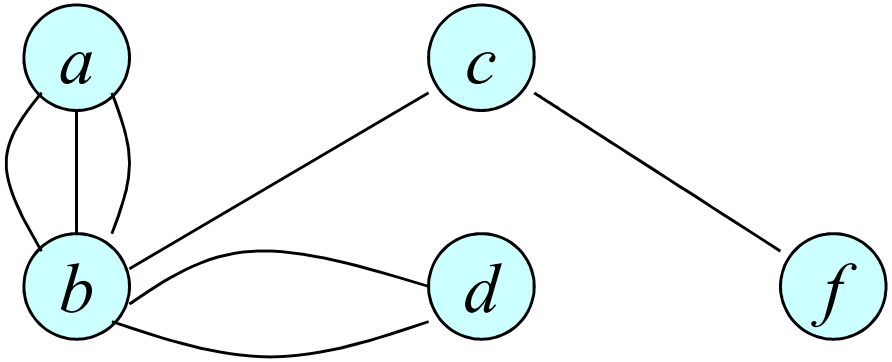
An acyclical undirected graph is forest



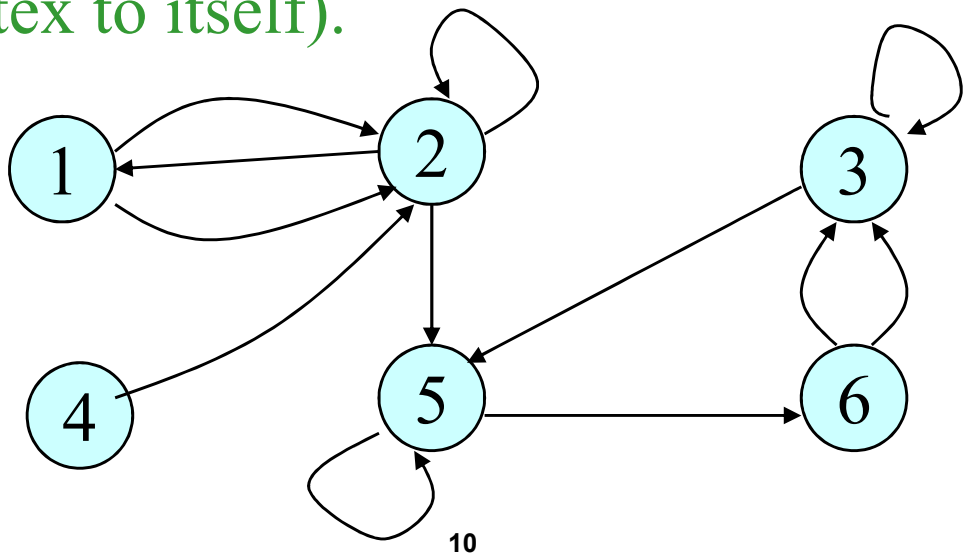


More General Graphs

A *multipgraph* allows multiple edges between two vertices.



A *pseudograph* is a multigraph that allows *self-loops* (edges from a vertex to itself).

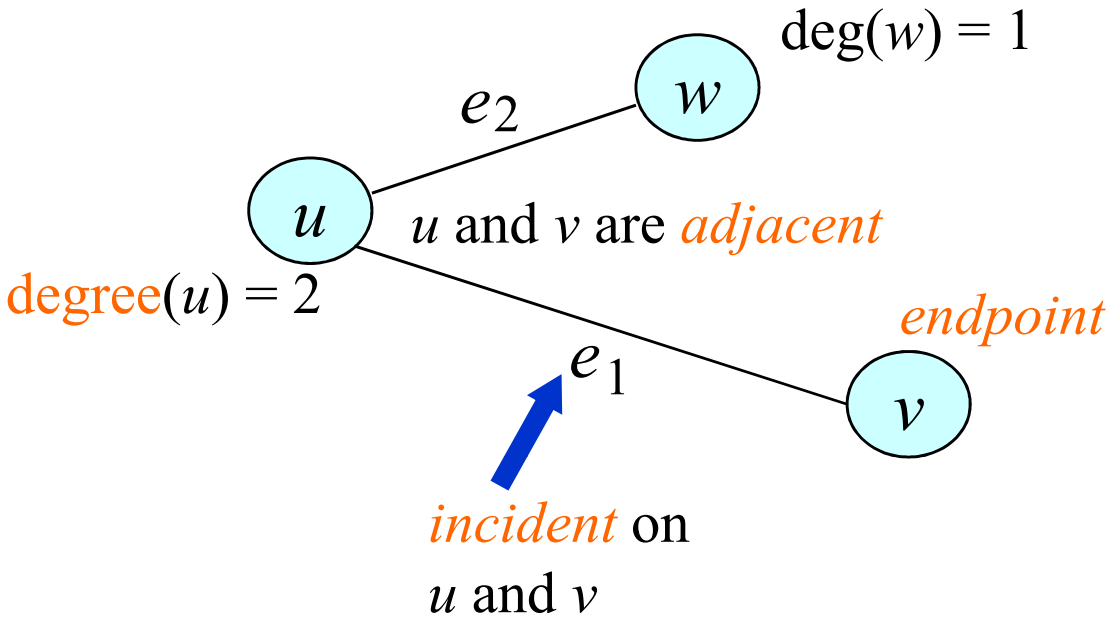


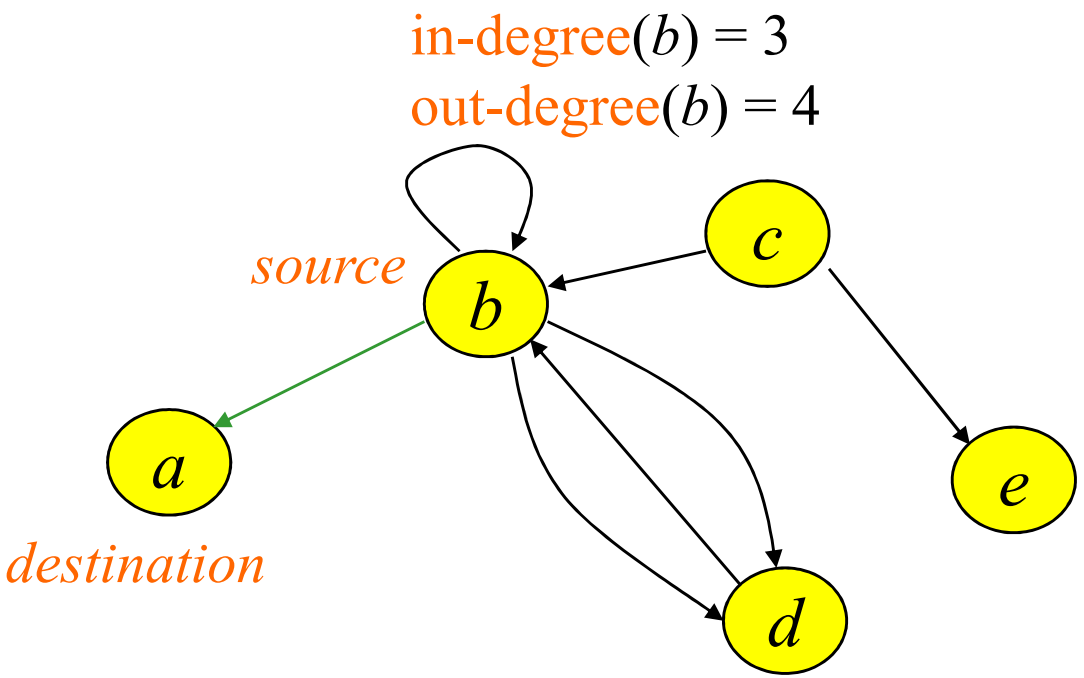
Simple Graph

A graph is simple when

- it is non-directed
- there is at most one edge between two vertices
- there is no loop of length one..

Edges & Degrees



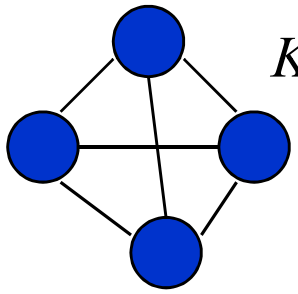


Handshaking Theorem

Let $G = (V, E)$ be an undirected simple graph.

$$\sum_{v \in V} \deg(v) = 2 \mid E \mid$$

$$\mid E \mid \leq \mid V \mid \cdot (\mid V \mid - 1) / 2$$



K_4 has $\binom{4}{2} = 6$ edges

If G is directed:

$$\sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} \text{outdeg}(v) = \mid E \mid$$

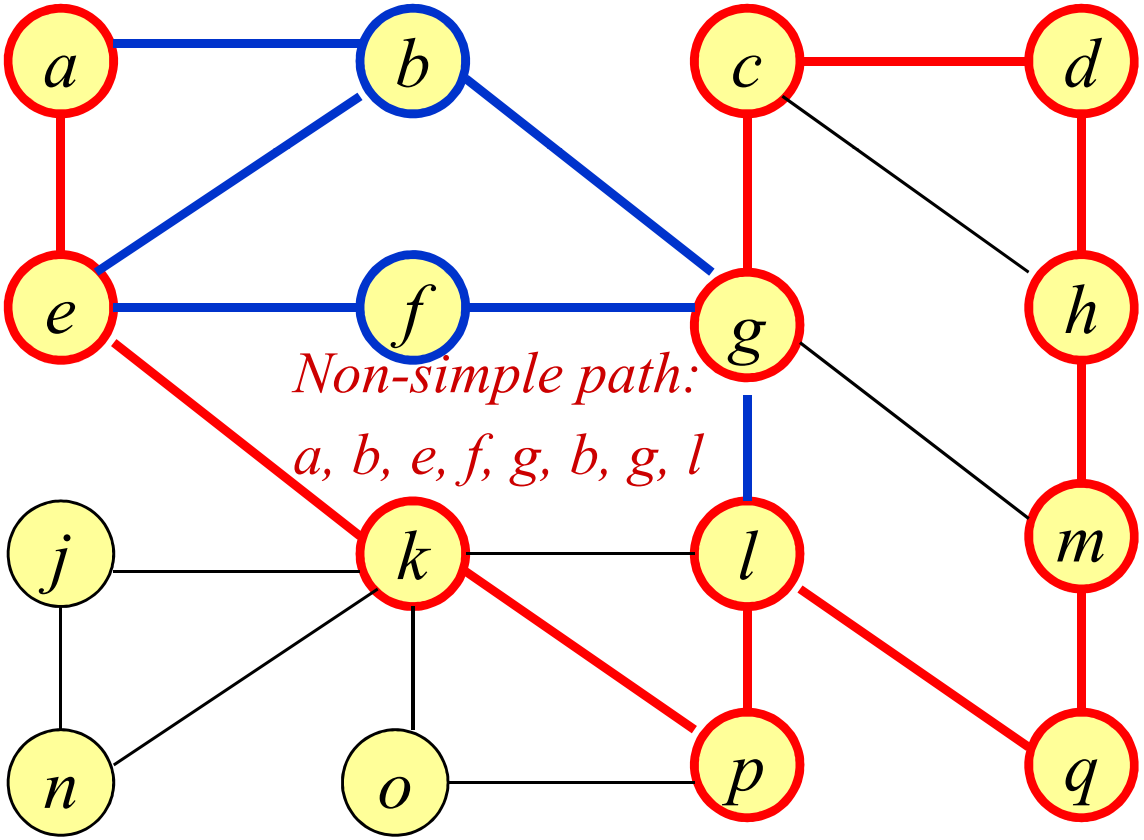
$$\mid E \mid \leq \mid V \mid \cdot (\mid V \mid - 1)$$

Path

A *path* of length k is a sequence $v_0, v_1, \ldots, v_k$ of vertices such that (v_i, v_{i+1}) for $i = 0, 1, \ldots, k-1$ is an edge of G.

b, c, d not a path

Simple path:
a, e, k, p, l, q
m, h, d, c, g
(no repeated vertices)

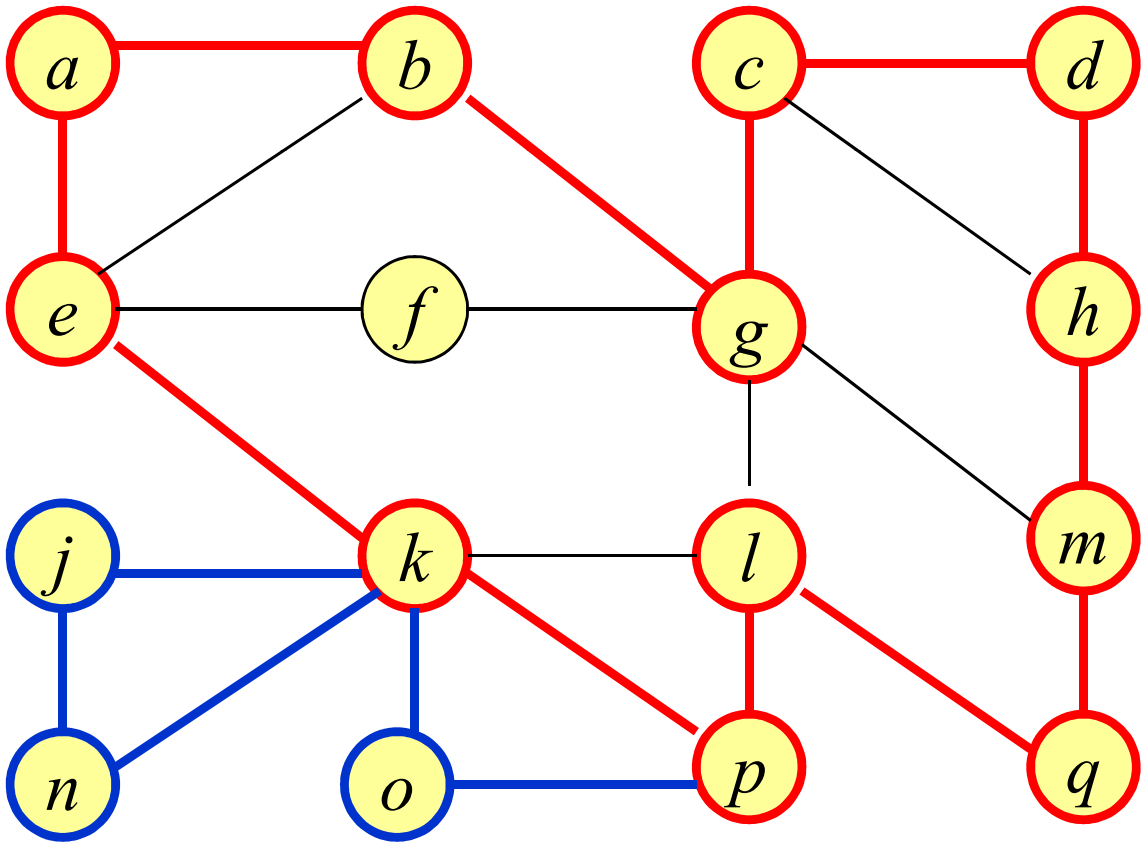


Non-simple path:
a, b, e, f, g, b, g, l

Cycle

A *cycle* is a path that starts and ends at the same vertex.

A *simple cycle* has no repeated vertices.

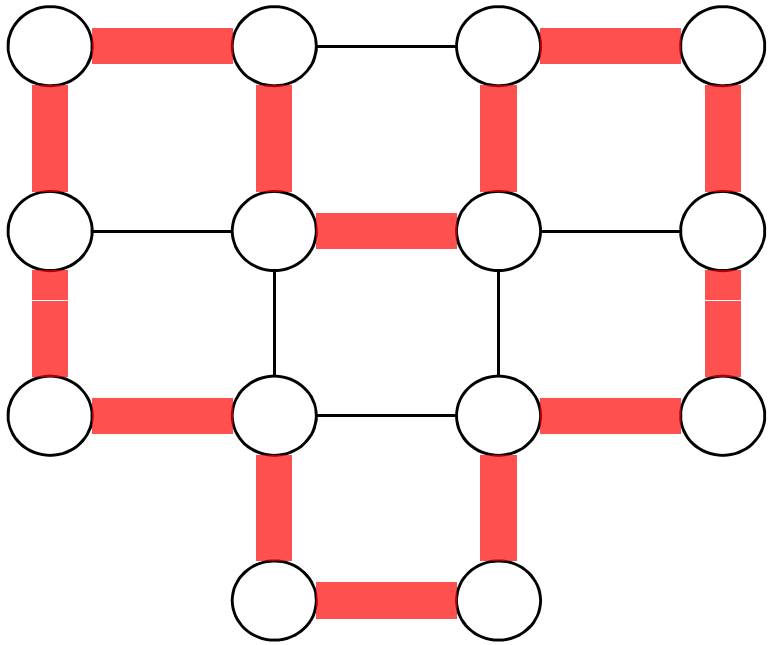


k, j, n, k, p, o,k is not simple.

Hamiltonian Cycle (HC)

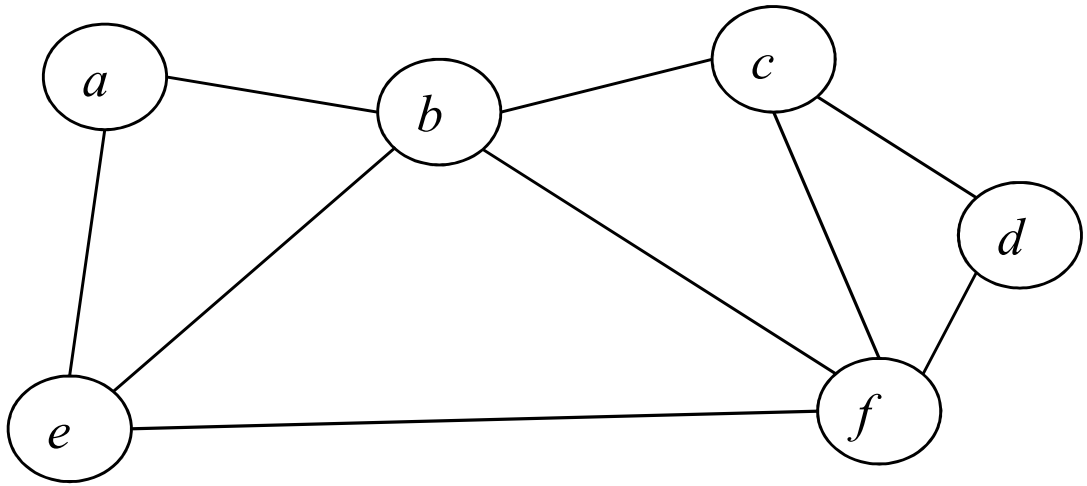
A Hamiltonian cycle (or Hamiltonian circuit) is a simple cycle In an undirected graph which visits each vertex exactly once and also returns to the starting vertex.

.

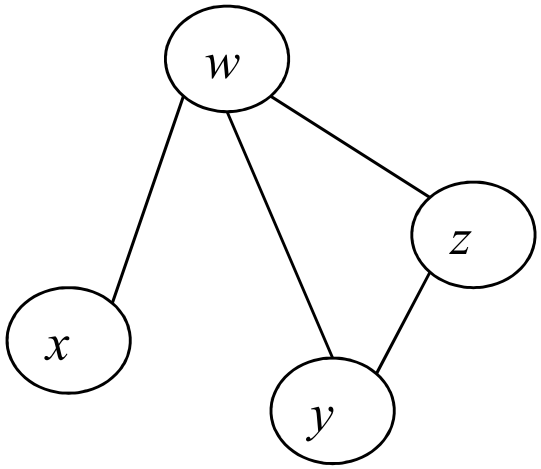


Hamiltonian Cycle

- Hamiltonian cycle: simple cycle containing all vertices



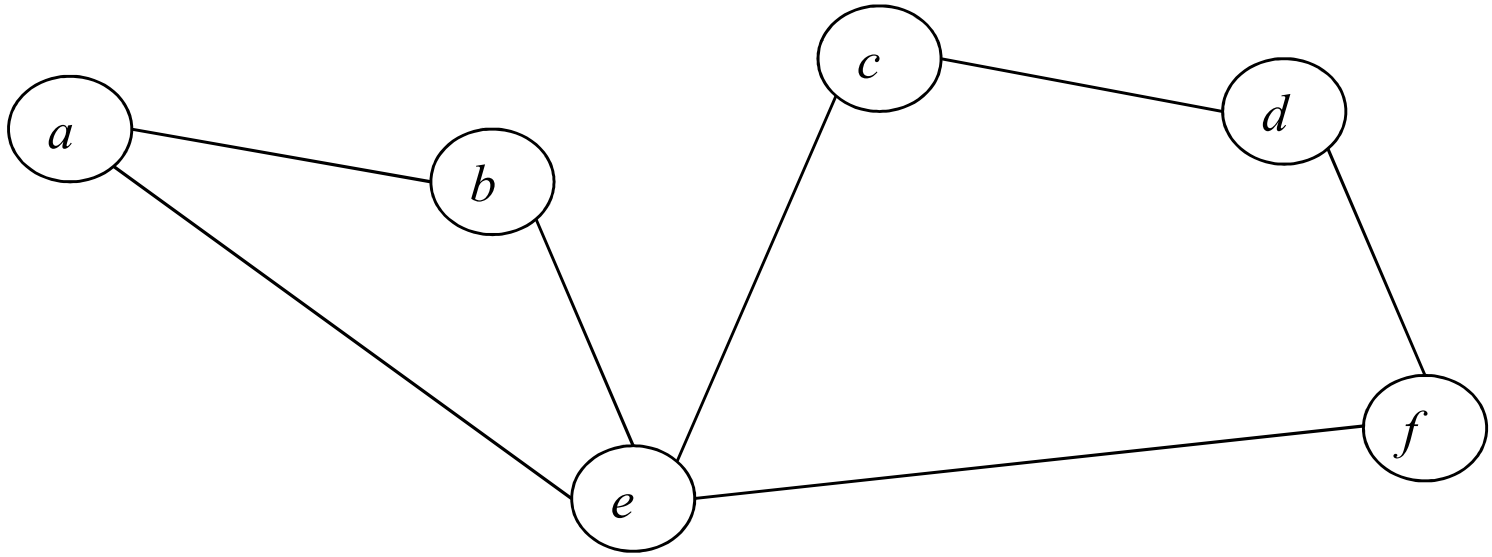
hamiltonian cycle:
a,b,c,d,f,e,(then back to a)



no hamiltonian cycle

Eulerian Tour

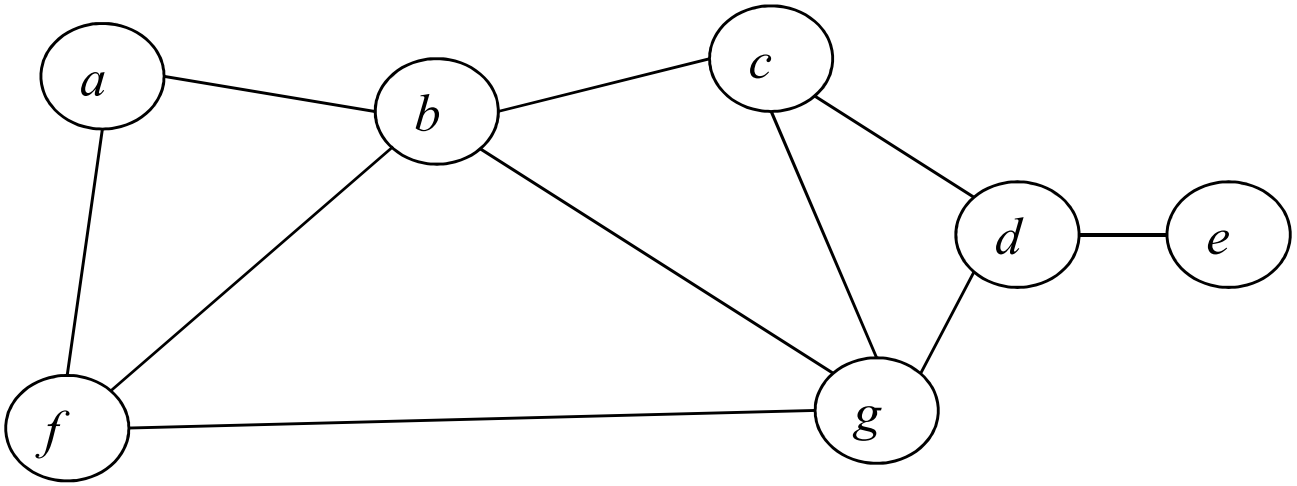
*An Euler tour, **Eulerian cycle**, or **Eulerian circuit** of a connected directed/undirected graph is a cycle that traverses each edge of G exactly once, although it may visits a vertex more than once.*



Euler tour: c,d,f,e,a,b,e,c

Euler Tour (2)

- Some graphs do not have euler tours
  - specifically those with vertices whose degree (# of neighbors) is odd

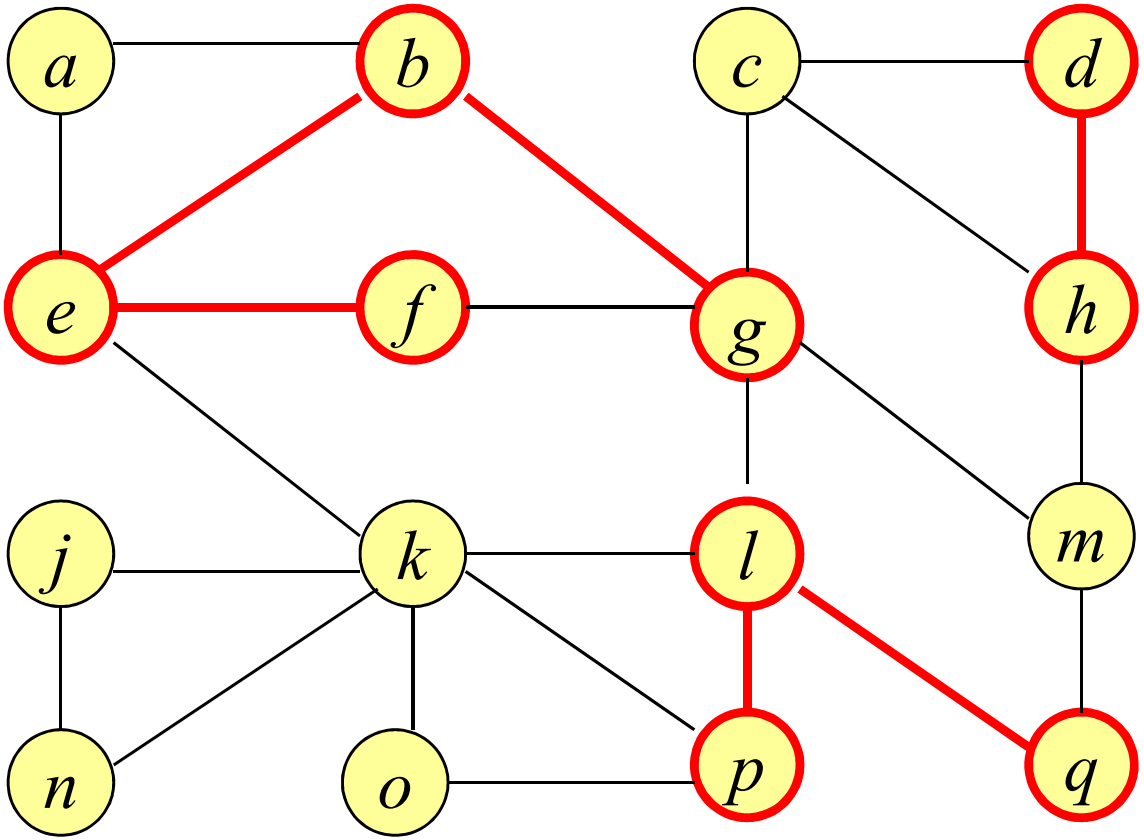


f,c,d and e have odd degree

Subgraph

A *subgraph* H of G

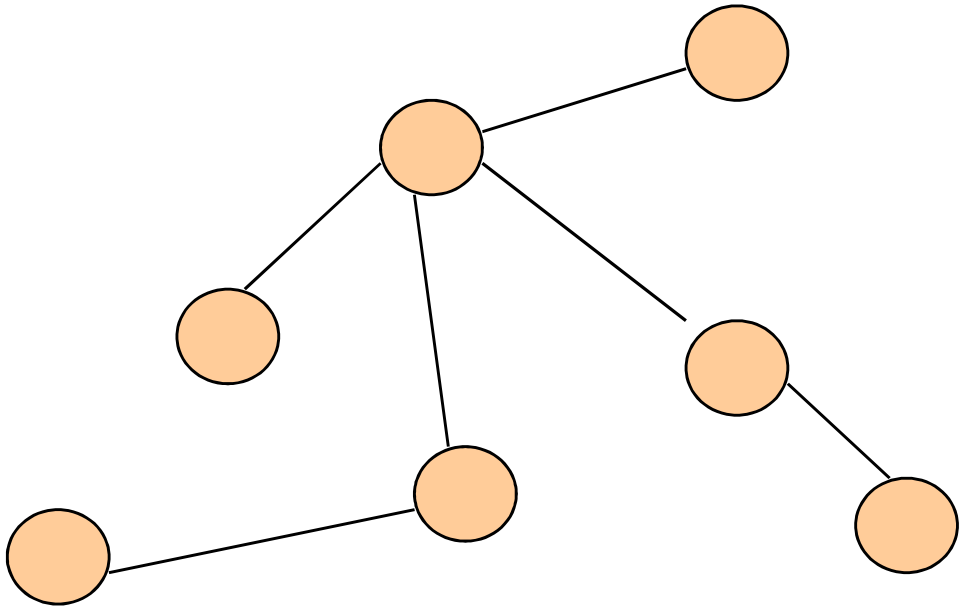
- is a graph;
- its edges and vertices are subsets of those of G.



$V(H) = \{b, d, e, f, g, h, l, p, q\}$ $E(H) = \{(b, e), (b, g), (e, f), (d, h), (l, p), (l, q)\}$

Tree Graph/Tree

- A connected graph T without any cycles is called a tree graph or free tree or a tree
- There is a unique simple path p between any two nodes U and V in T



Free Tree and Forest

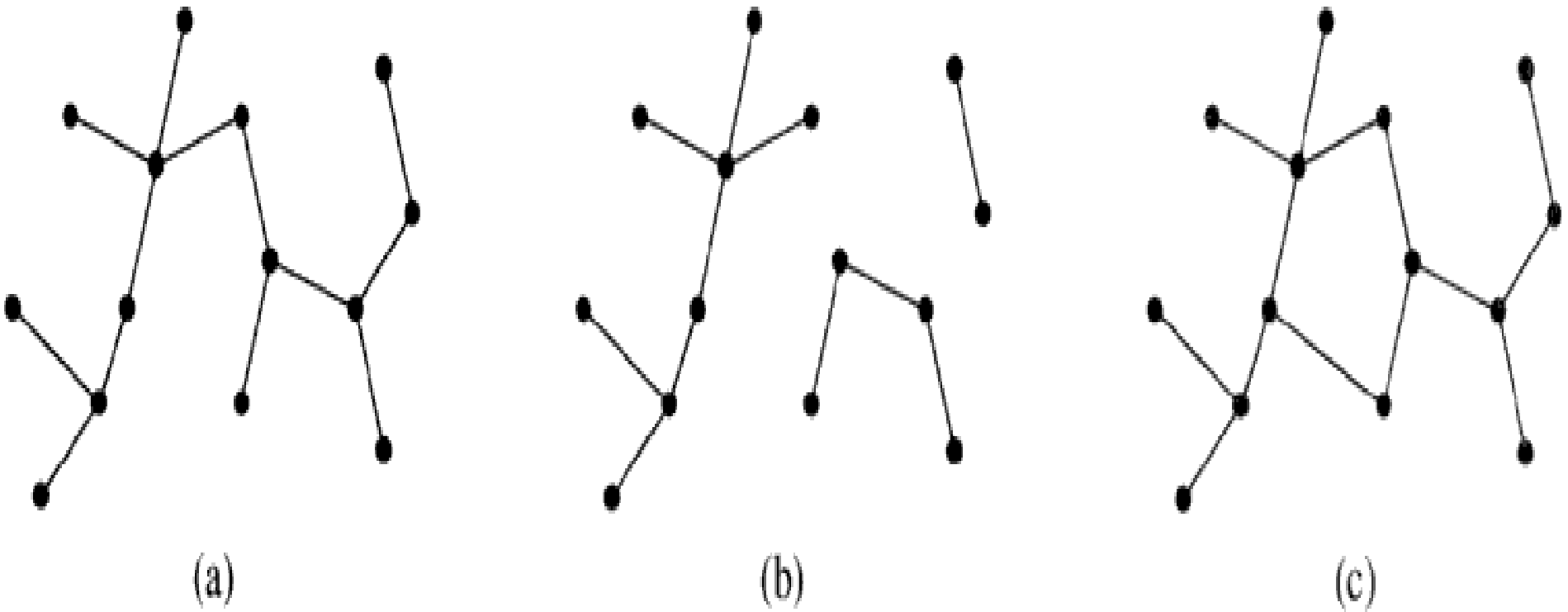


Figure B.4 (a) A free tree. (b) A forest. (c) A graph that contains a cycle and is therefore neither a tree nor a forest.

Graphs

- We will typically express running times in terms of |E| and |V| (often dropping the |'s)
  - If $|E| \approx |V|^2$ the graph is *dense*
  - If $|E| \approx |V|$ the graph is *sparse*
- If you know you are dealing with dense or sparse graphs, different data structures(List, Matrix) may make sense

Representing Graphs

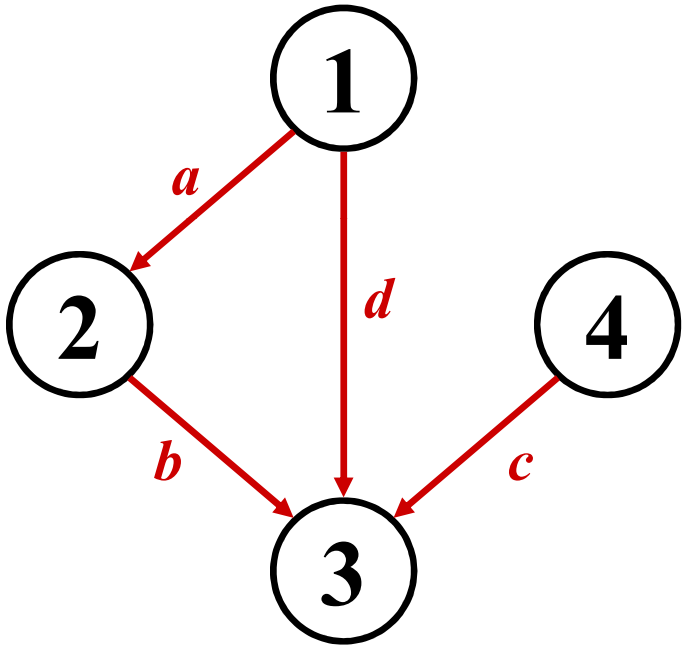
Two techniques are available:

1. **Adjacency matrix**
2. **Adjacency List**

- Assume V = {1, 2, …, n}
- An *adjacency matrix* represents the graph as a n x n matrix A:
  - A[i, j] = 1 if edge (i, j) $\in$ E (or weight of edge)
    = 0 if edge (i, j) $\notin$ E

Graphs: Adjacency Matrix

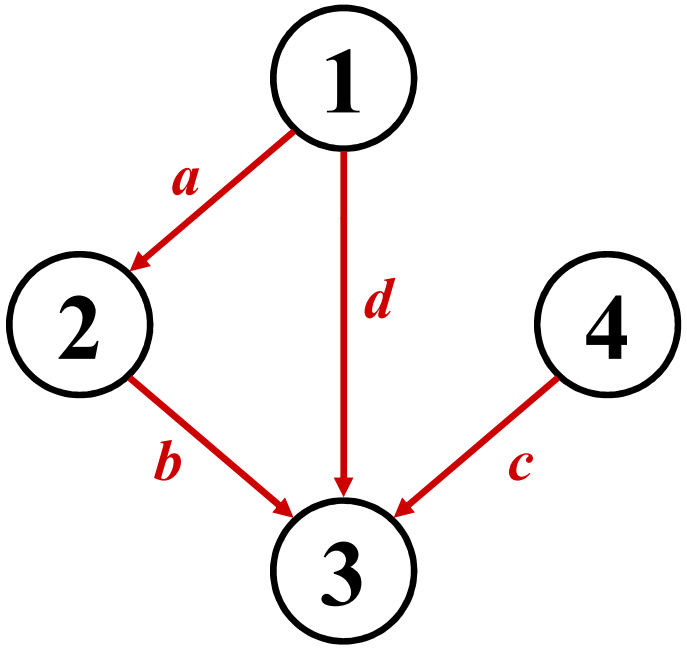
- Example:



A	1	2	3	4
1				
2				
3			??	
4				

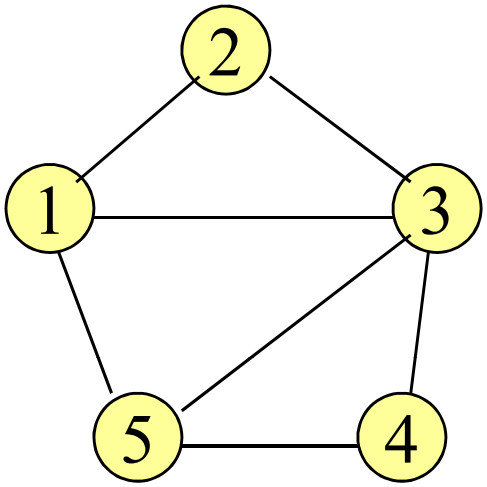
Graphs: Adjacency Matrix

- Example:



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

Graphs: Adjacency Matrix



$$A = (a_{ij})$$

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

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Space: $\Theta(|V|^2)$.

Preferred when the graph is small or dense.

Graphs: Adjacency Matrix

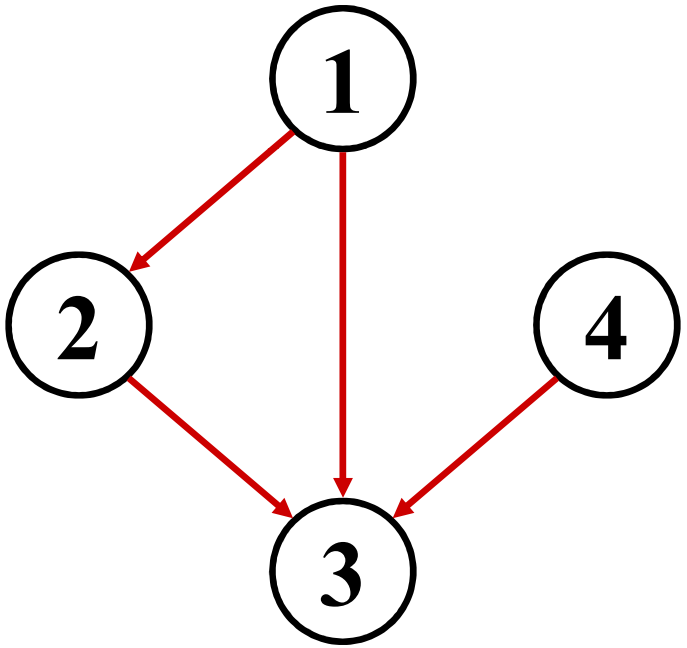
- *How much storage does the adjacency matrix require?*
- A: $O(V^2)$

Graphs: Adjacency Matrix

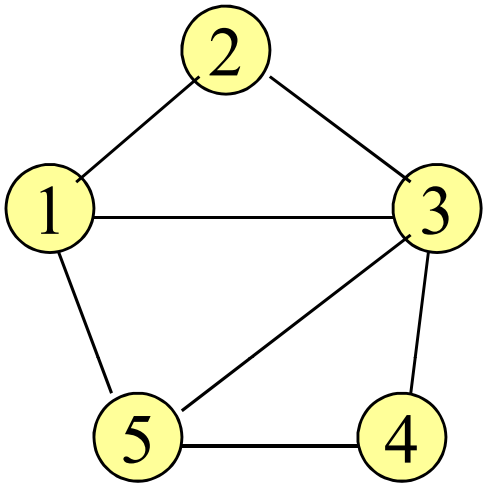
- The adjacency matrix is a dense representation
  - Usually too much storage for large graphs
  - But can be very efficient for small graphs
- Most large interesting graphs are sparse
  - E.g., planar graphs, in which no edges cross, have $|E| = O(|V|)$ by Euler's formula
  - For this reason the *adjacency list* is often a more appropriate representation

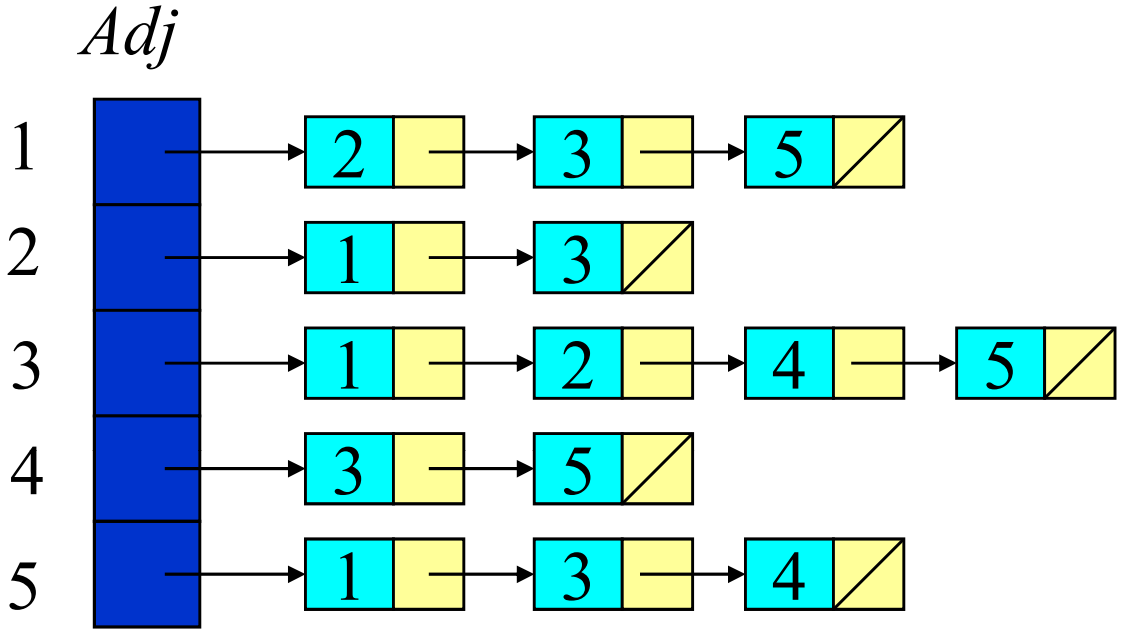
Graphs: Adjacency List

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to v
- Example:
  - Adj[1] = {2,3}
  - Adj[2] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}
- Variation: can also keep a list of edges coming *into* vertex



Graphs: Adjacency List



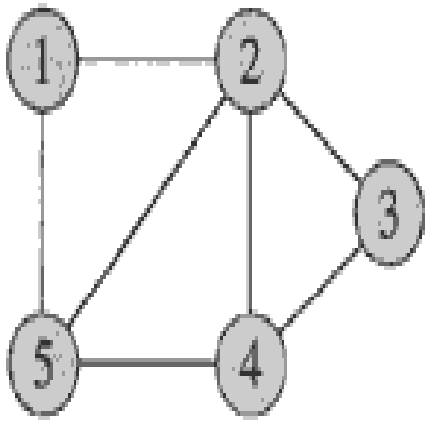


If G is directed, the total length of all the adjacency lists is $|E|$.

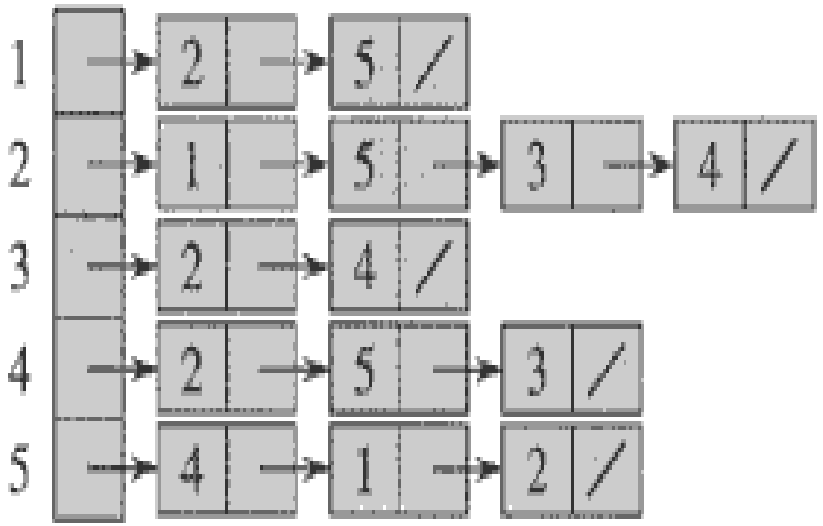
If G is undirected, the total length is $2|E|$.

Space requirement: $\Theta(|V| + |E|)$. Preferable representation.

Adjacency List and Matrix of undirected Graph



(a)



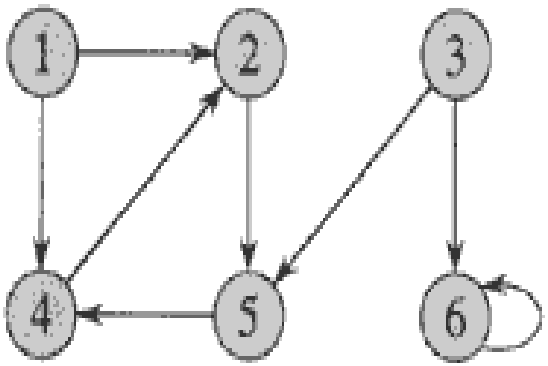
(b)

	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

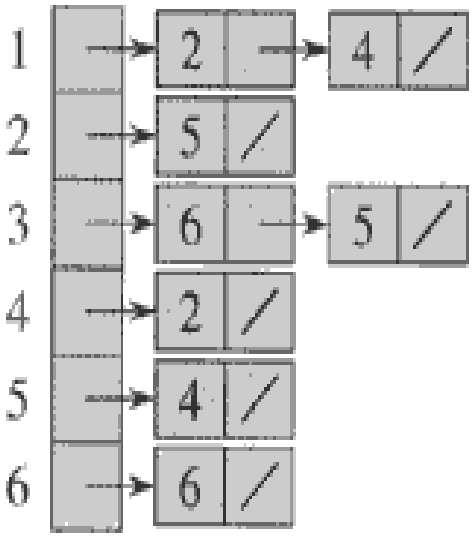
(c)

Figure 22.1 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.

Adjacency List and Matrix of Directed Graph



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Figure 22.2 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.

Graphs: Adjacency List

- How much storage is required?
  - The *degree* of a vertex v = # incident edges
    - Directed graphs have in-degree, out-degree
  - For directed graphs, # of items in adjacency lists is
    $$\Sigma \text{ out-degree}(v) = |E|$$
    takes $\Theta(V + E)$ storage (*Why?*)
  - For undirected graphs, # items in adj lists is
    $$\Sigma \text{ degree}(v) = 2\,|E|$$ (*handshaking lemma*)
    also $\Theta(V + E)$ storage
- So: Adjacency lists take O(V+E) storage

Operation Time

Operation	Adjacency List	Adjacency Matrix
Scan incident edges	$\Theta(\deg(v))$	$\Theta(\|V\|)$
Scan outgoing edges	$\Theta(\text{outdeg}(v))$	$\Theta(\|V\|)$
Scan incoming edges	$\Theta(\text{indeg}(v))$	$\Theta(\|V\|)$
Test adjacency of u and v	$\Theta(\min(\deg(u), \deg(v))$	$O(1)$
Space	$\Theta(\|V\|+\|E\|)$	$\Theta(\|V\|^2)$

Graph Searching/ Traversals

- Given: a graph G = (V, E), directed or undirected
- Goal: explore every vertex and every edge
- Some applications require visiting every vertex in the graph exactly once
- Ultimately: build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Note: might also build a *forest* if graph is not connected

Graph Searching/ Traversals

Two methods

1- Breath-first Search

- Use a queue as an auxiliary structure to hold nodes for future processing

2- Depth-first Search

- Use stack to handle traversing

Breadth-First Search

"Explore" a graph, turning it into a tree

- **One vertex at a time**
- **Pick a *source vertex* to be the root**
- **Find ("discover") its children, then their children, etc.**

Shortest Path

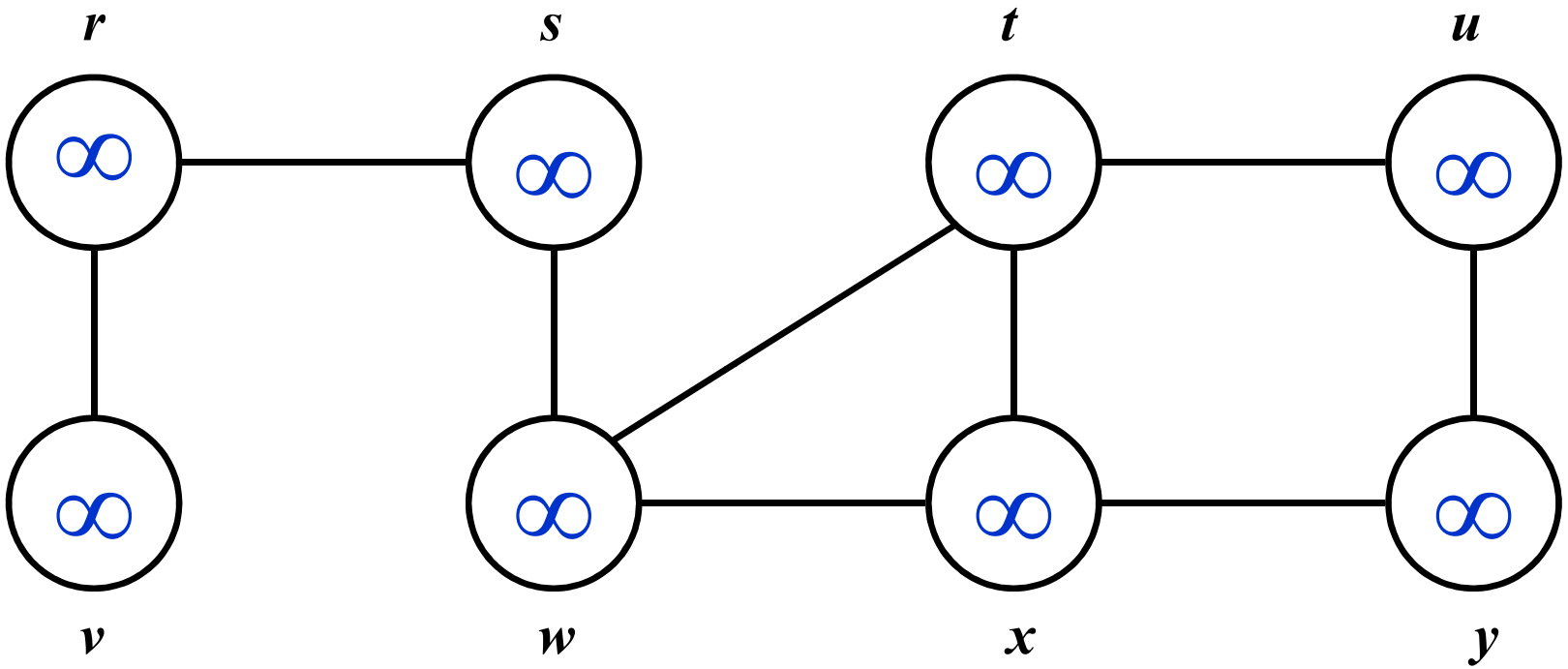
- **It computes the distance (smallest number of edges) from s (source vertex) to each reachable vertices.**
- **For every vertex v reachable from s the path in the breath-first tree from s to v corresponds to a "shortest path" from s to v in G, that is the path containing the smallest number of edges.**

Discovery (discovers all vertices at distance k from s before discovering any vertices at distance k +1)

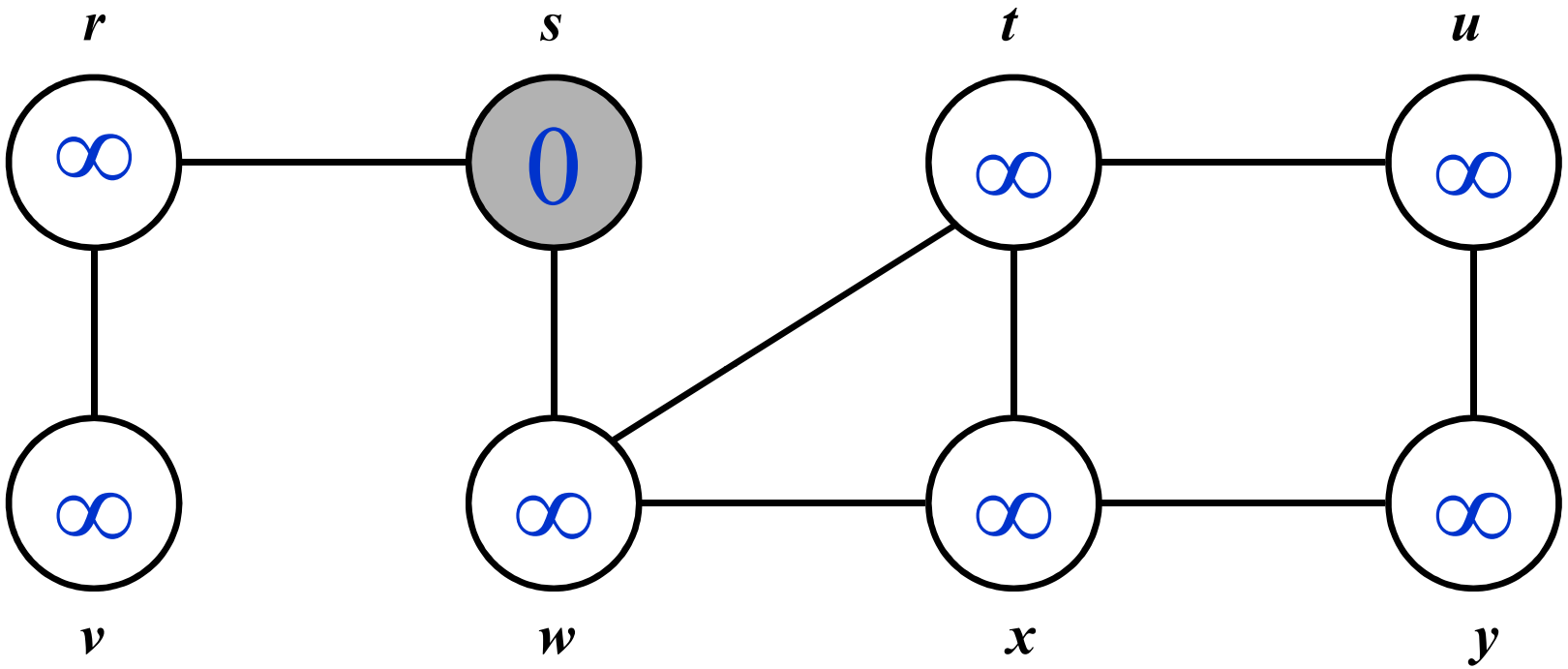
Breadth-First Search

- To keep track of progress, breath-first search colors each vertex white, gray, or black.
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

Breadth-First Search: Example

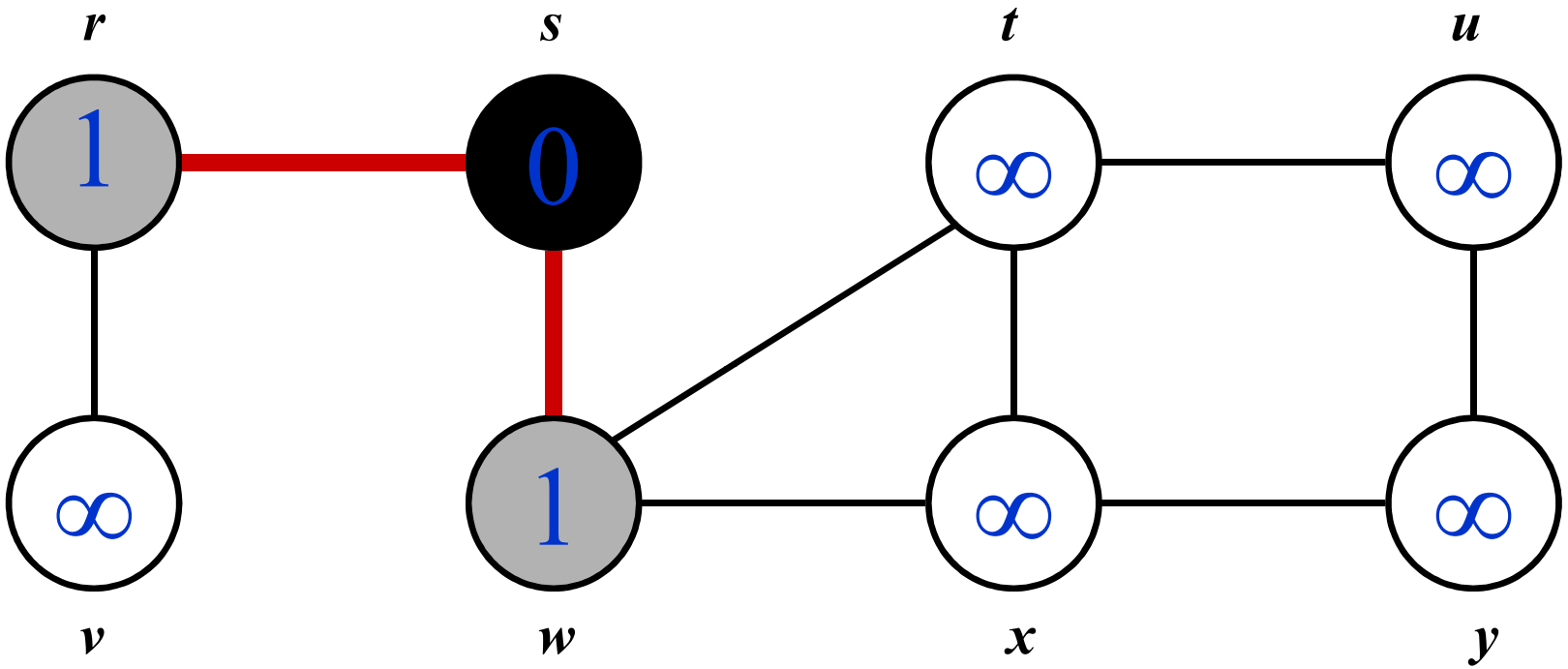


Breadth-First Search: Example



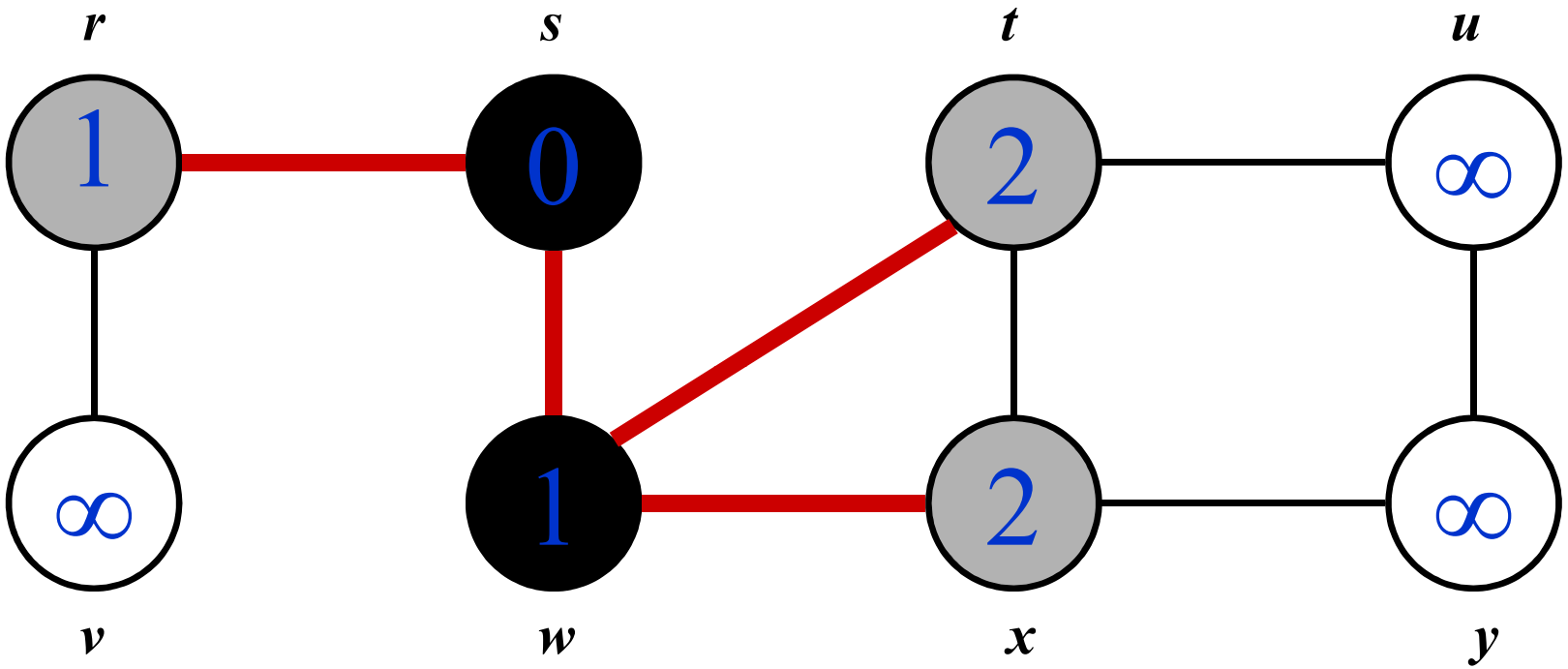
Q: | s |

Breadth-First Search: Example



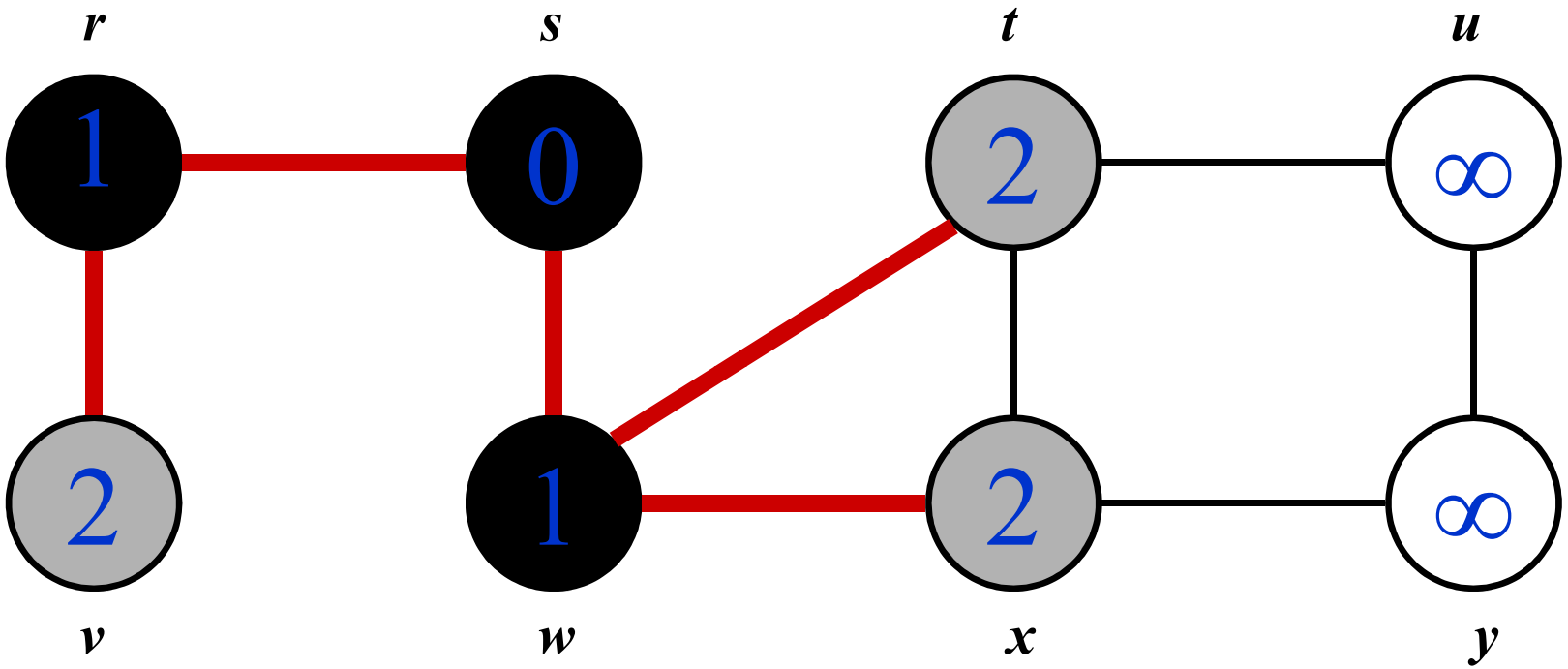
Q: | w | r |

Breadth-First Search: Example



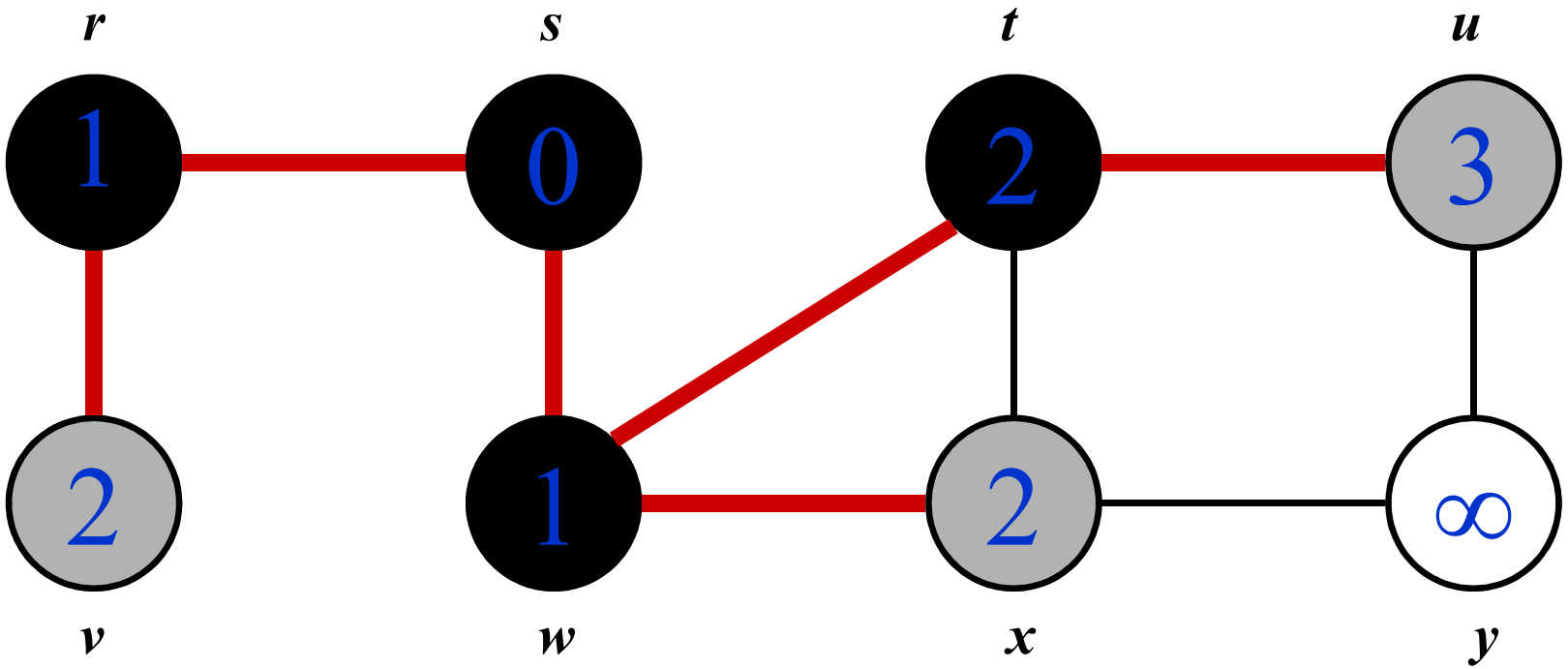
Q: | *r* | *t* | *x* |

Breadth-First Search: Example



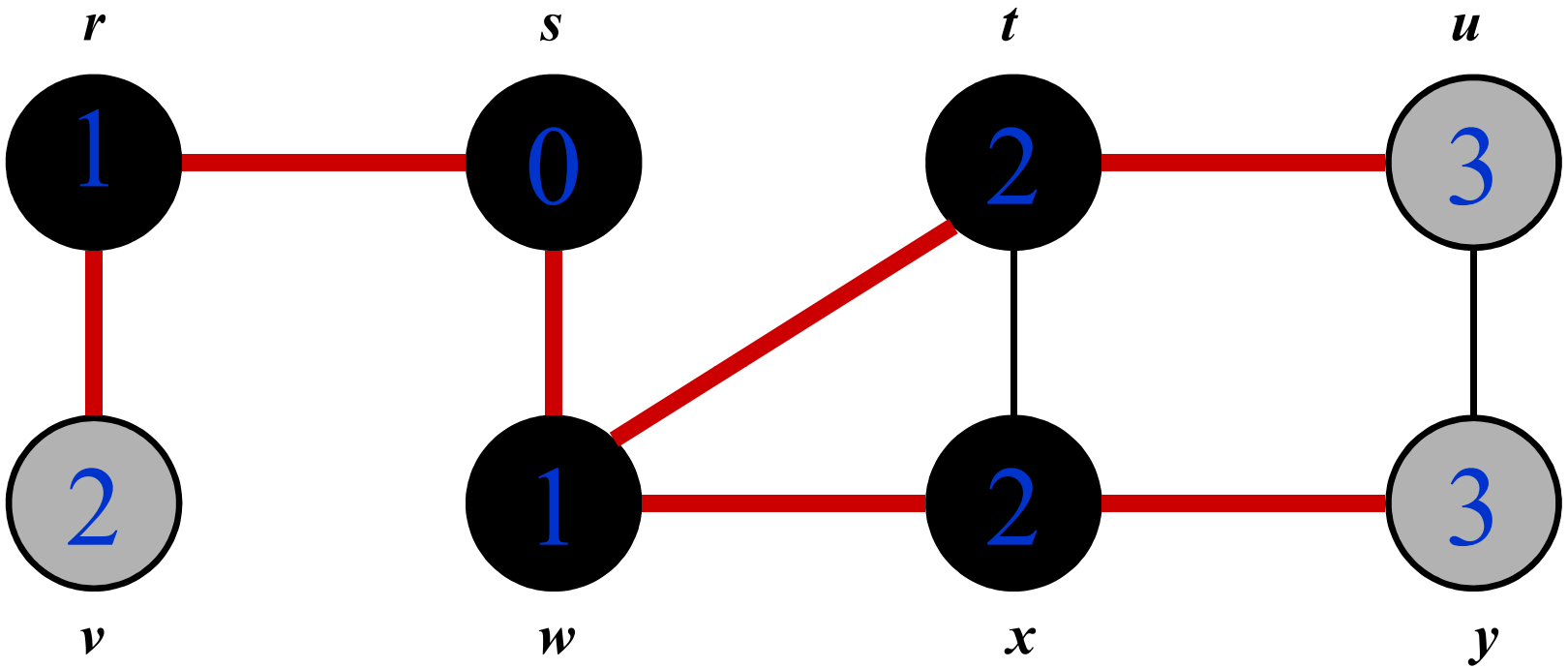
Q: | t | x | v |

Breadth-First Search: Example



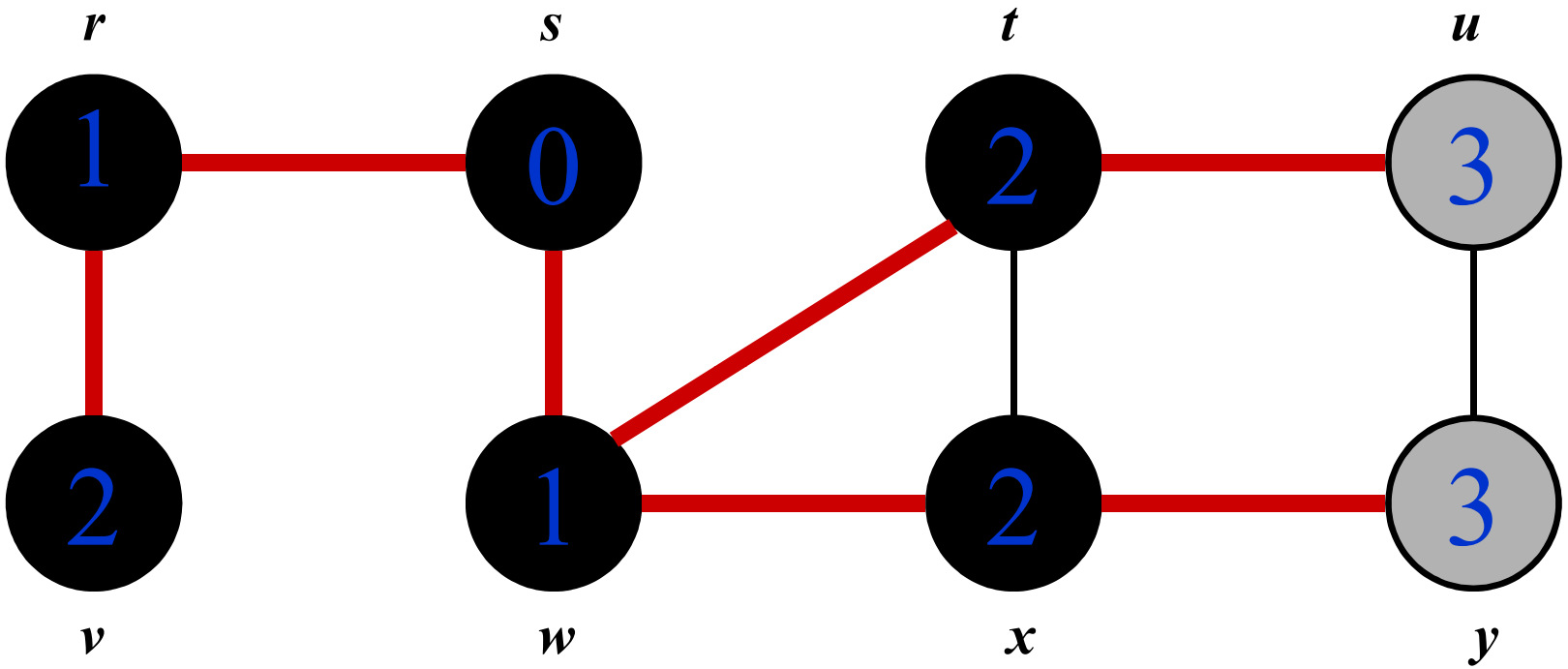
Q: | x | v | u |

Breadth-First Search: Example



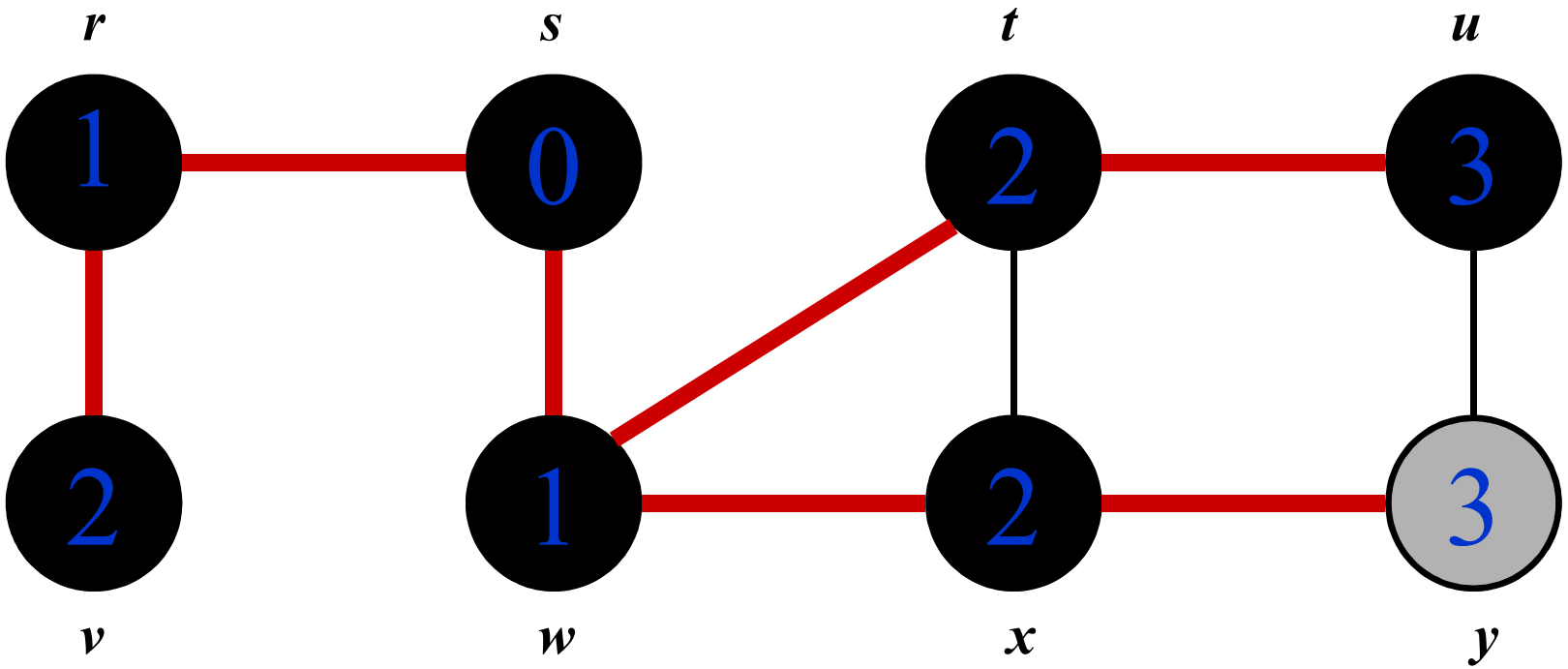
Q: | *v* | *u* | *y* |

Breadth-First Search: Example



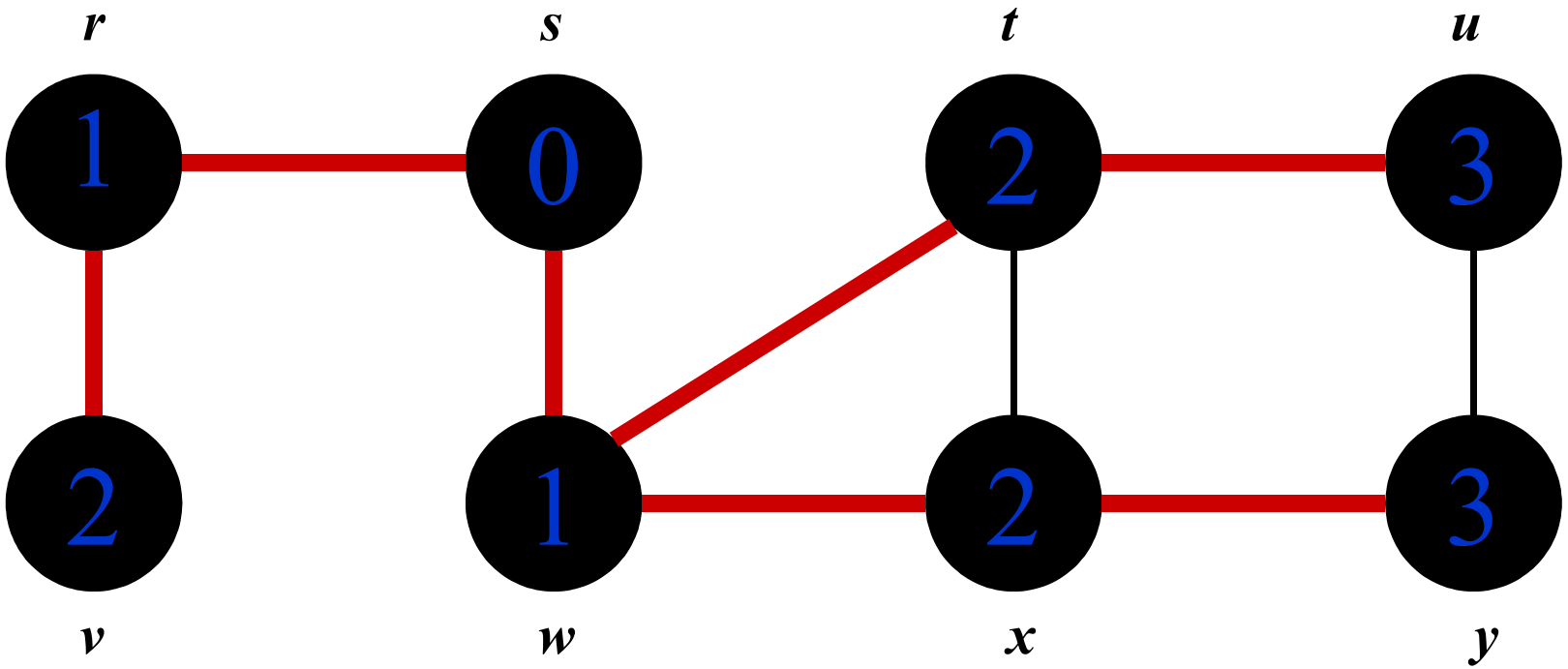
Q: | *u* | *y* |

Breadth-First Search: Example



Q: | y |

Breadth-First Search: Example



Q: ø

Breadth-First Search (from Book)

```
BFS(G, s)
 1   for each vertex u ∈ V[G] − {s}
 2        do color[u] ← WHITE
 3           d[u] ← ∞        // set distance (d) of every node to 0
 4           π[u] ← NIL      //Set parent of every vertex to nil
 5   color[s] ← GRAY
 6   d[s] ← 0                // Set distance of source s to 0
 7   π[s] ← NIL              //set predecessor/parent of the source to be nil
 8   Q ← ∅
 9   ENQUEUE(Q, s)
10   while Q ≠ ∅
11        do u ← DEQUEUE(Q)  //consider each vertex
12           for each v ∈ Adj[u]   //v in Adj of u
13                do if color[v] = WHITE
14                     then color[v] ← GRAY
15                          d[v] ← d[u] + 1
16                          π[v] ← u
17                          ENQUEUE(Q, v)
18           color[u] ← BLACK
```

Breadth-First Search

(slightly different form book)

```
BFS(G, s) {
    initialize vertices;
    Q = {s};          // Q is a queue (duh); initialize to s
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

What does `v->d` *represent?*

What does `v->p` *represent?*

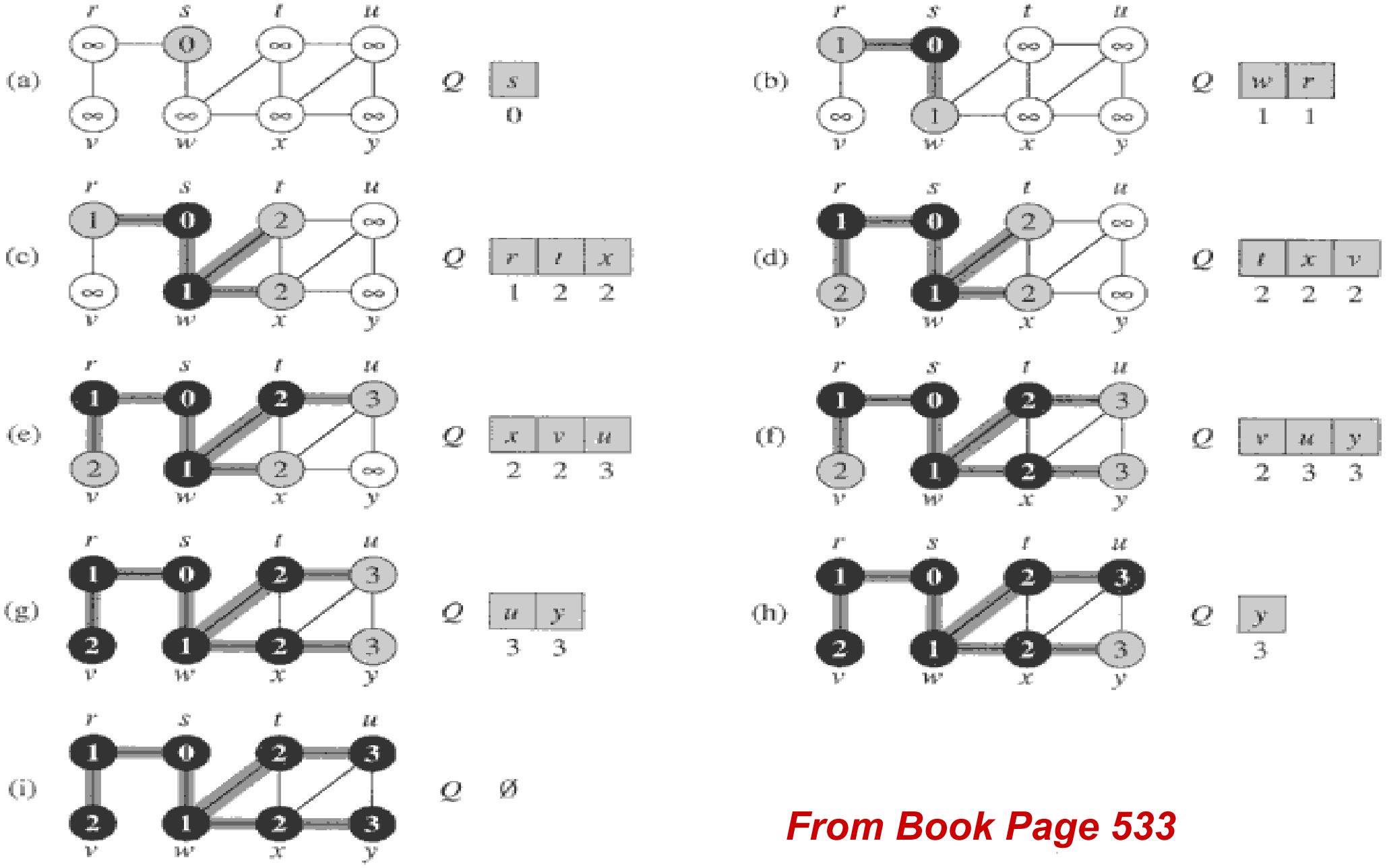


Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue.

Analysis BFS

- The operation of enqueuing and dequeuing take O(1) time. (i.e each vertex is enqueued at most once and thus dequeued at most once) so the total time devoted to queue operation is O(V).
- Adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once.
- Sum of length of the lengths of all the adjacency list is O(E), the total time spent in scanning adjacency lists is O(E).
- The overhead for initialization is O(V)
- **So the total running time of BFS is O(V+E)**

BFS: The Code Again

```
BFS(G, s) {
    initialize vertices;
    Q = {s};
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

Touch every vertex: O(V)

u = every vertex, but only once (Why?)

So v = every vertex that appears in some other vert's adjacency list

What will be the running time?

Total running time: O(V+E)

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
  - Proof given in the book (p. 472-5)
- 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

Print Shortest Path

The following procedure prints out the vertices on a shortest path from s to v assuming that BFS has already been run to compute the shortest-path tree

```
PRINT-PATH(G, s, v)
1  if v = s
2     then print s
3     else if π[v] = NIL
4             then print "no path from" s "to" v "exists"
5             else PRINT-PATH(G, s, π[v])
6                  print v
```

This procedure runs in time linear in the number of vertices in the path printed, since each recursive call is for a path one vertex shorter.

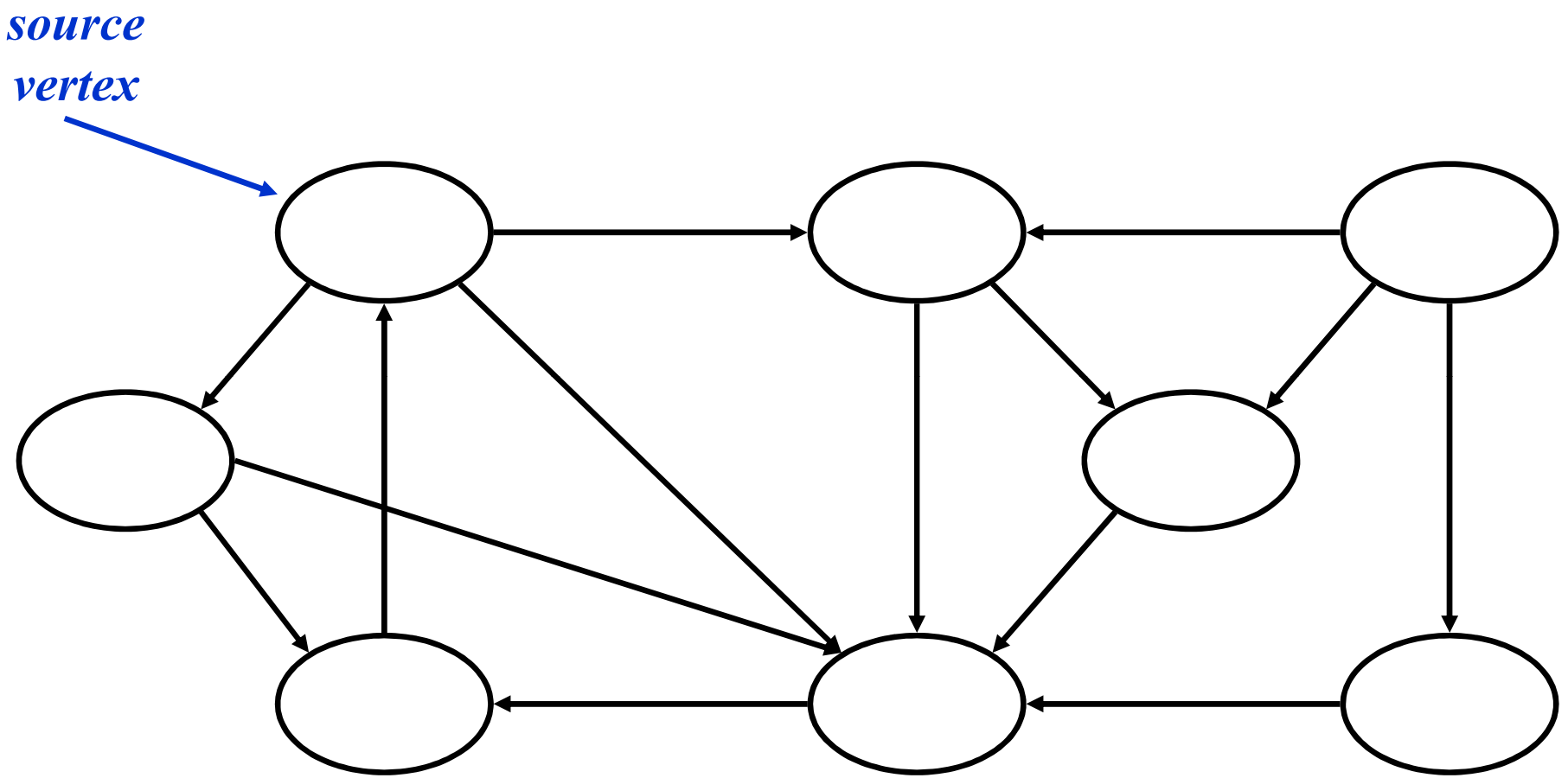
Depth-First Search

- *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
  - The process continues until we have discovered all the vertices that are reachable from the original source vertex.

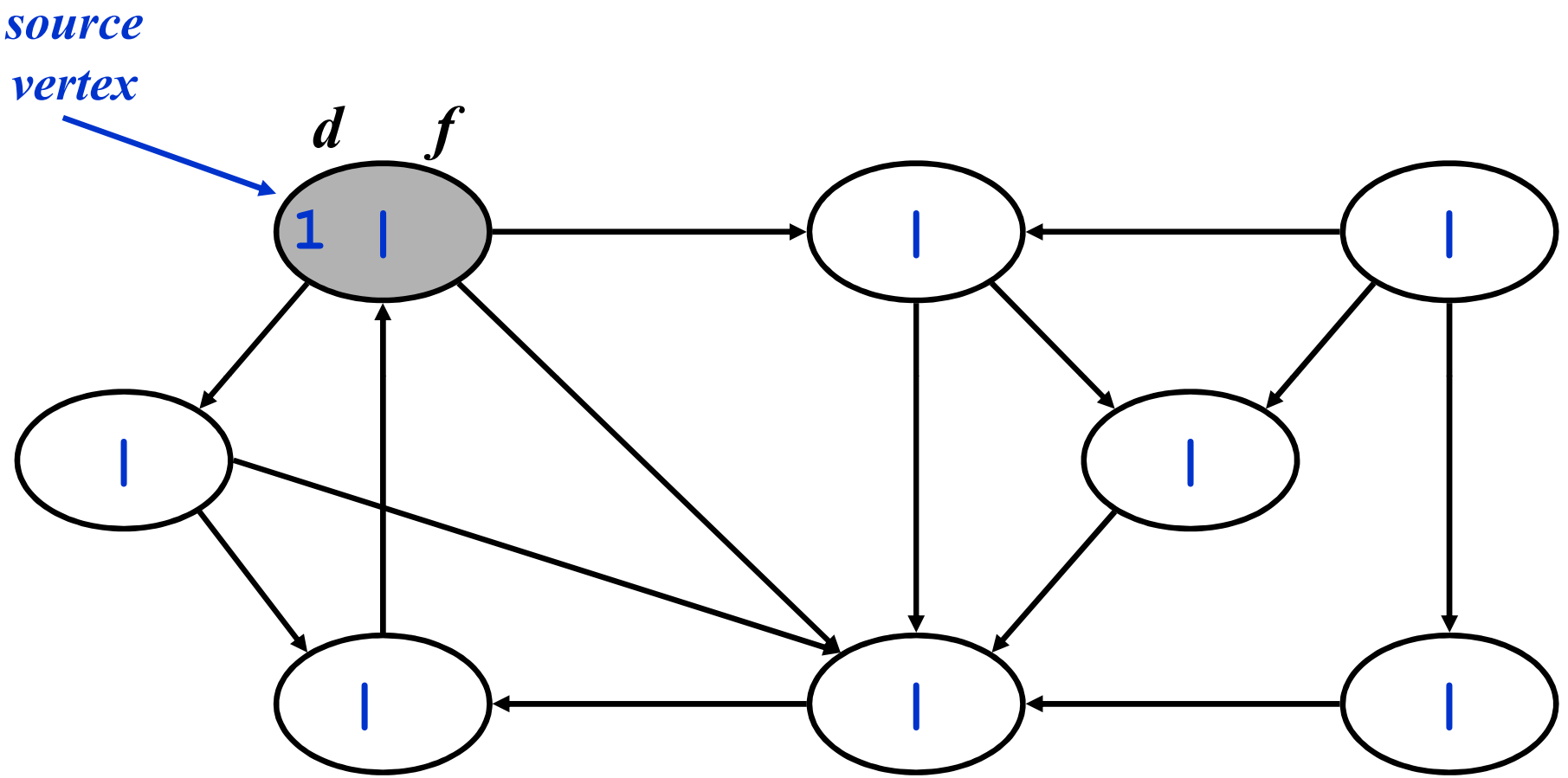
Depth-First Search

- If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from the source.
- The entire process is repeated until all the vertices are discovered.
- Vertices initially colored white
- Then colored gray when discovered
- Then black when finished

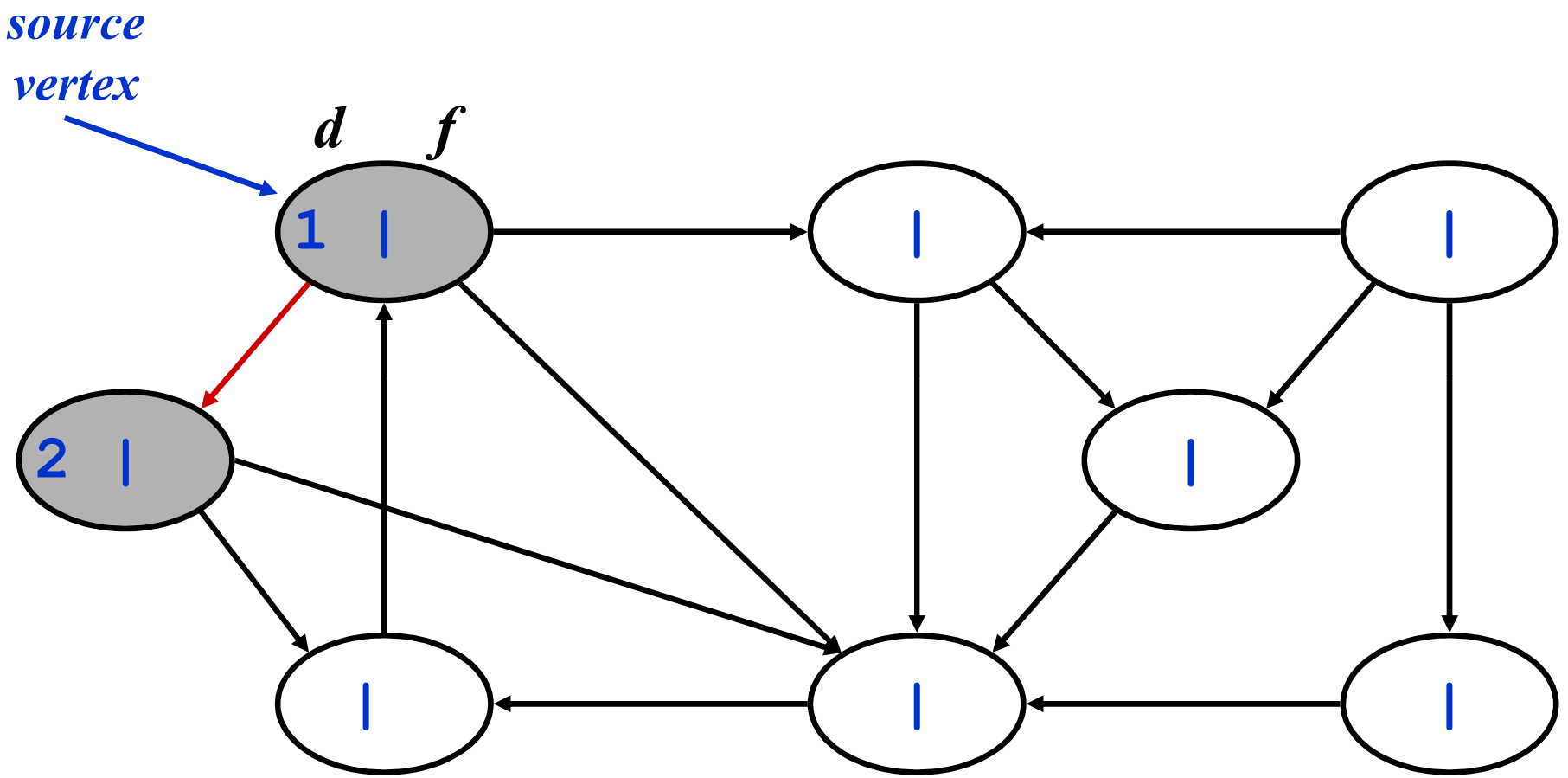
DFS Example



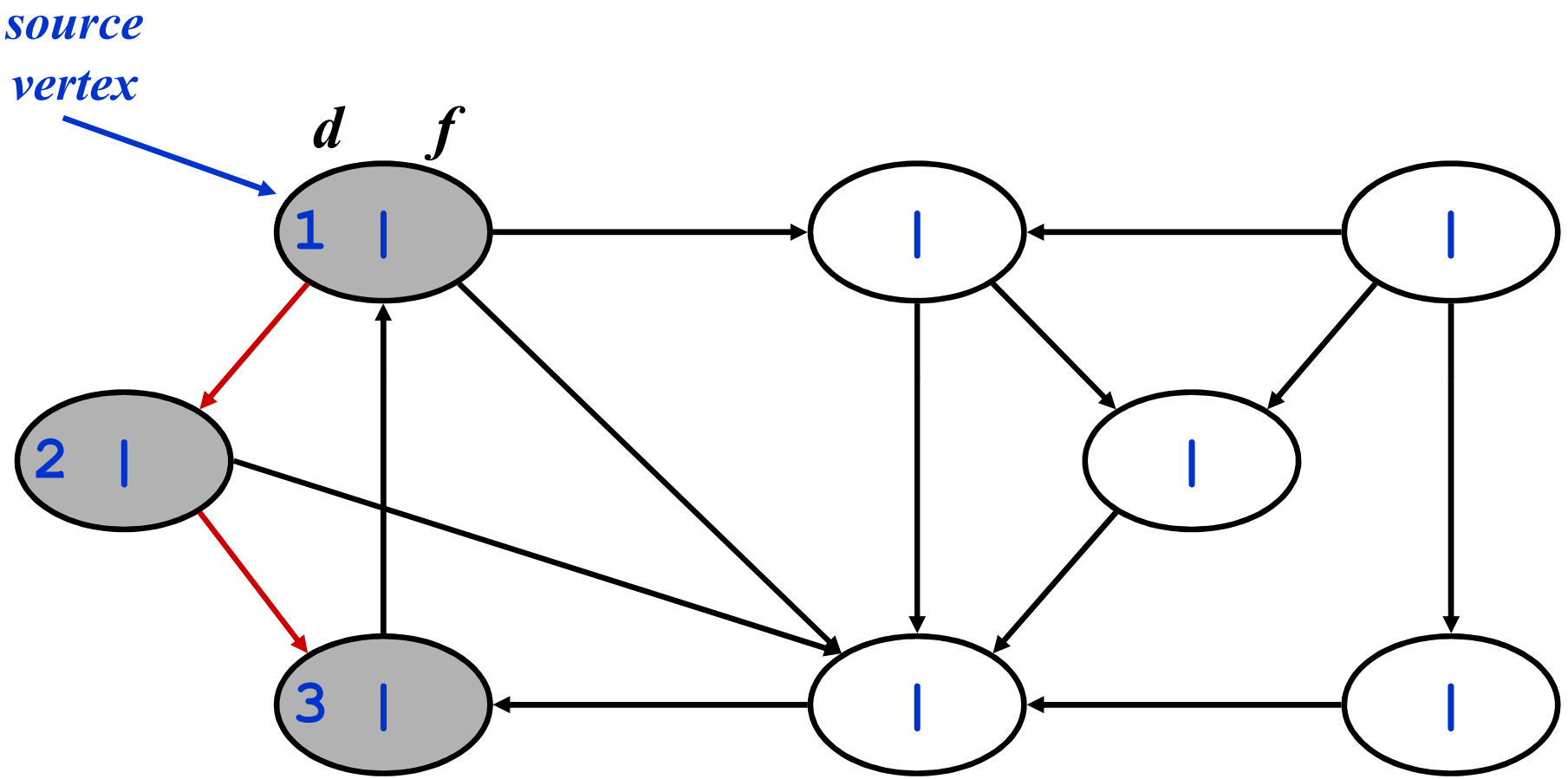
DFS Example



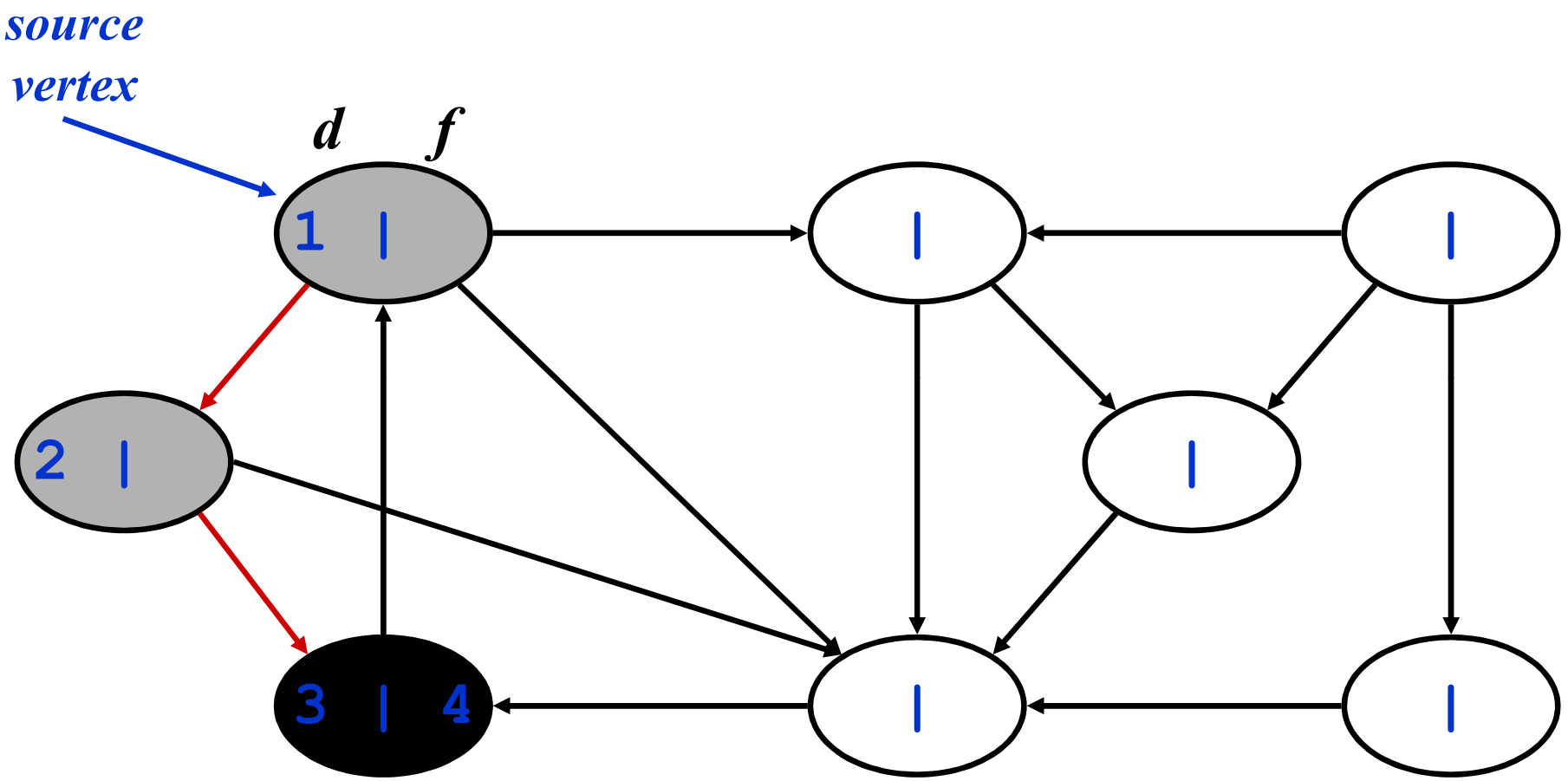
DFS Example



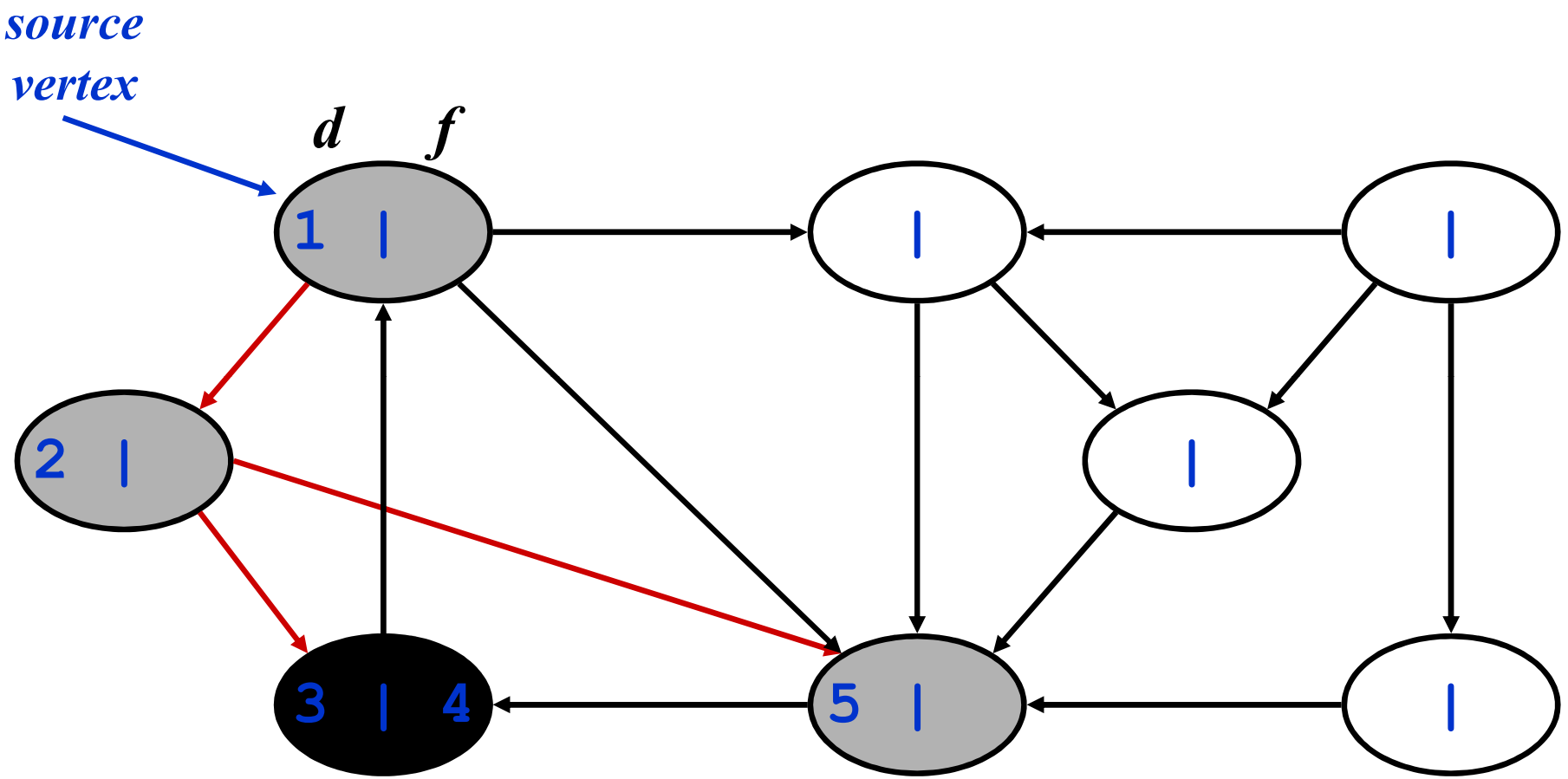
DFS Example



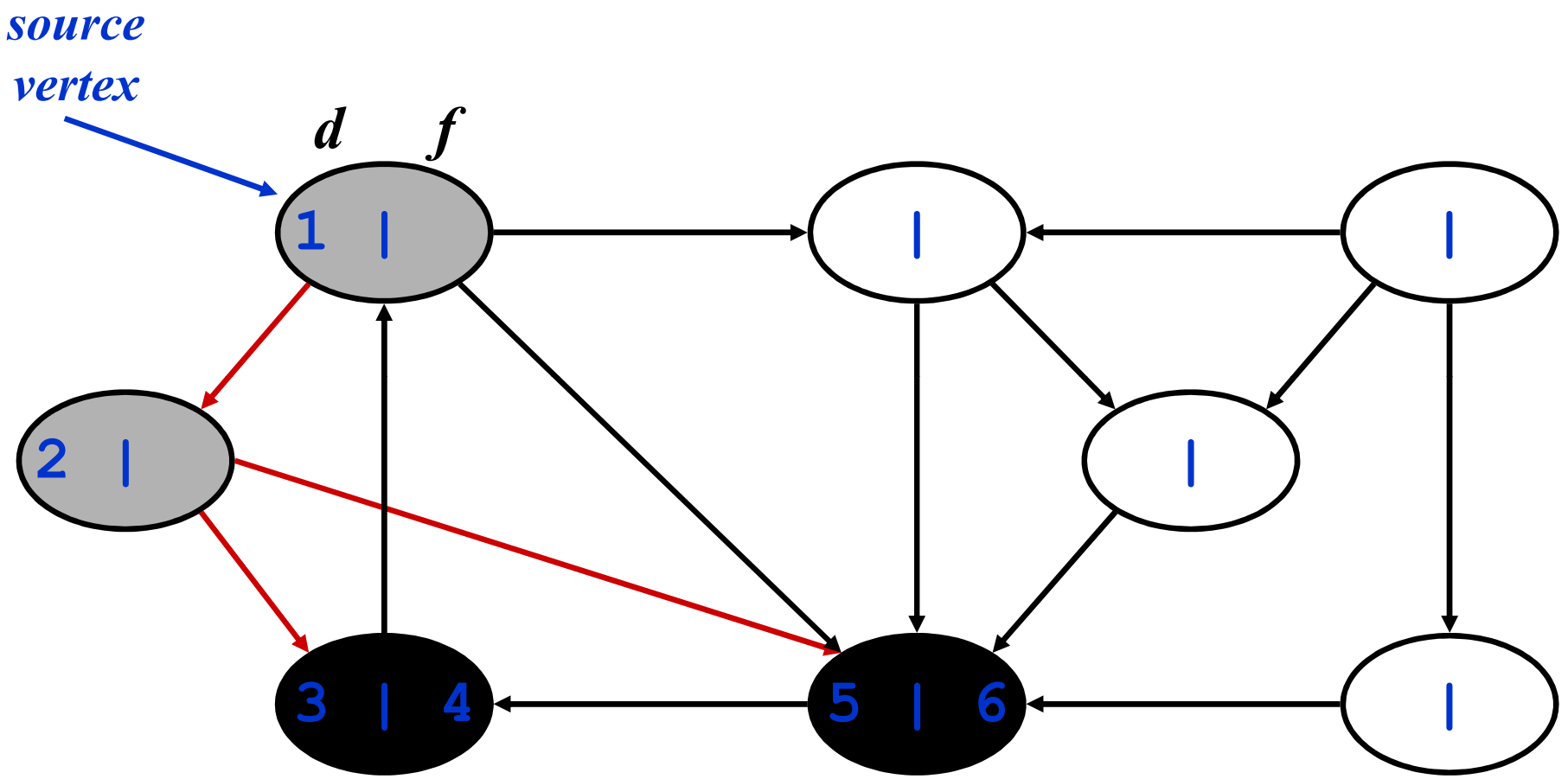
DFS Example



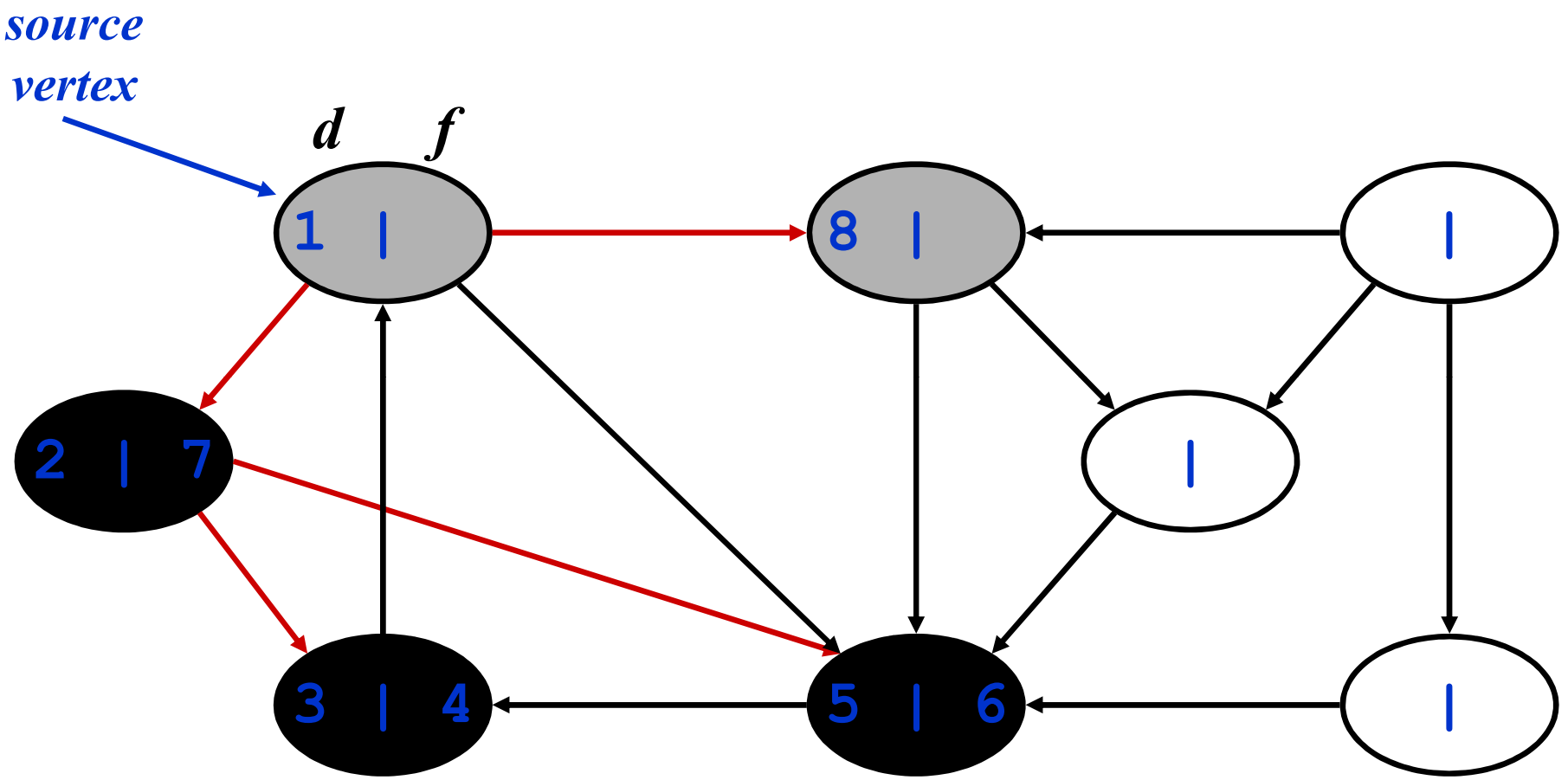
DFS Example



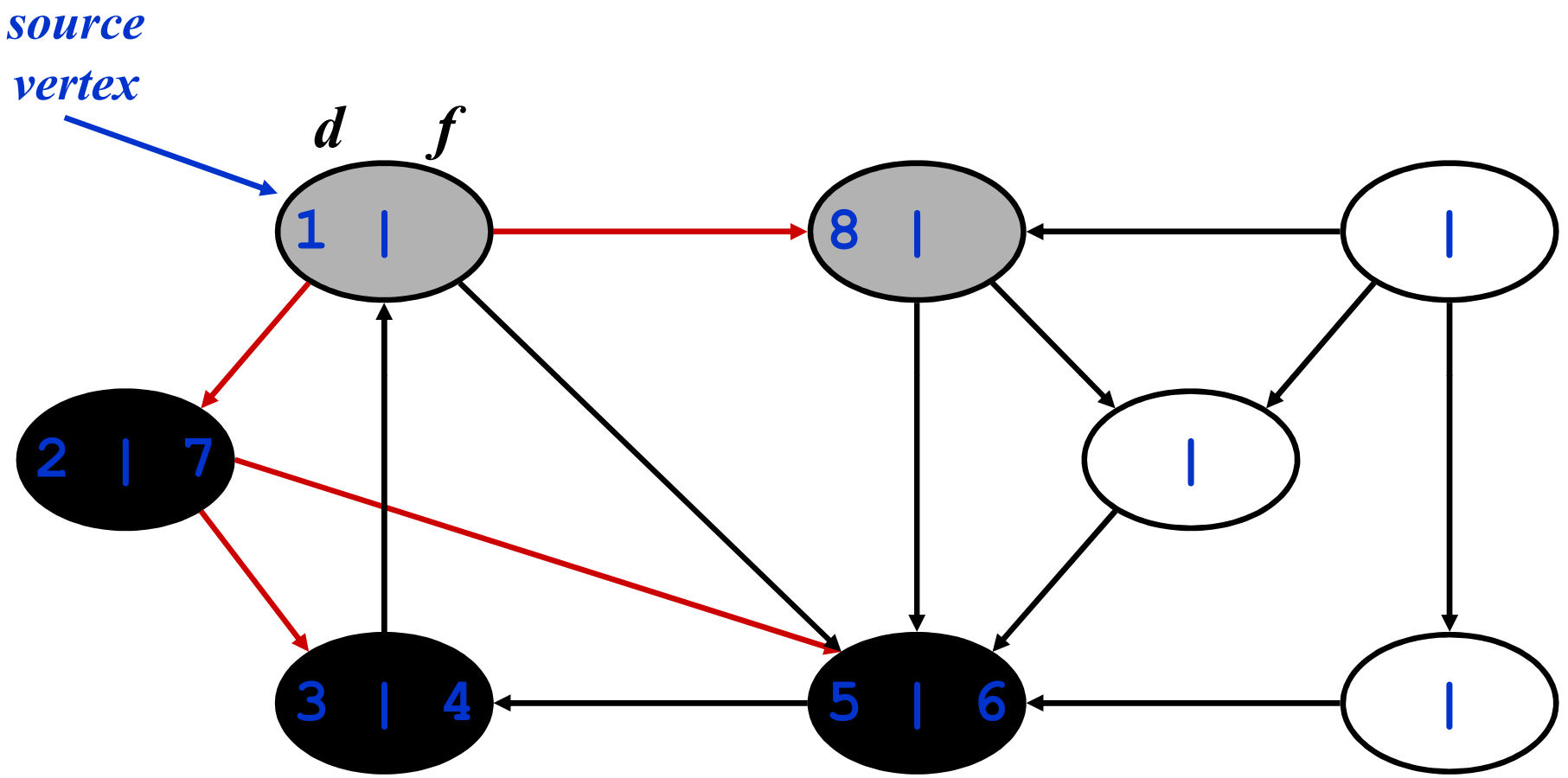
DFS Example



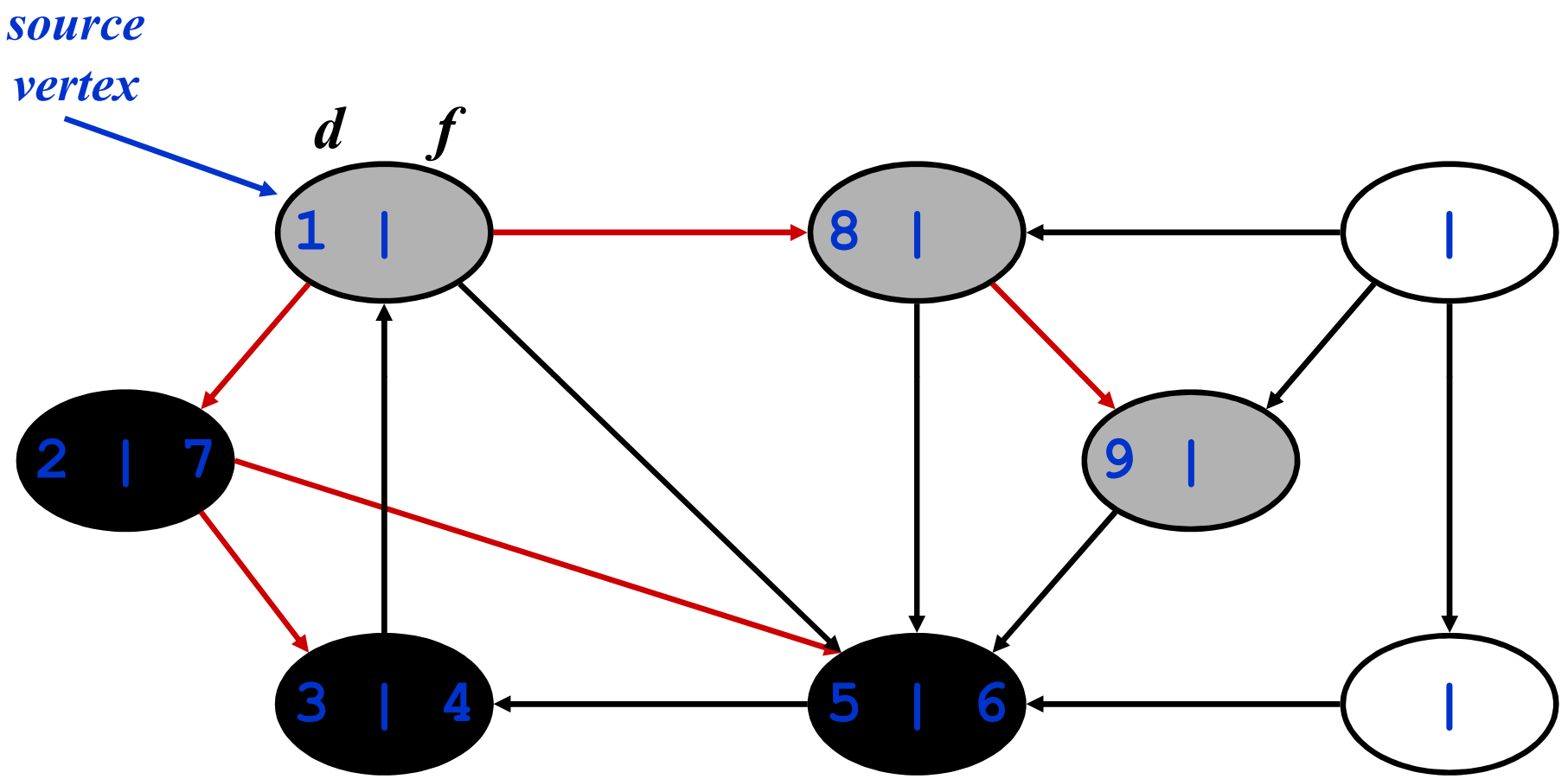
DFS Example



DFS Example

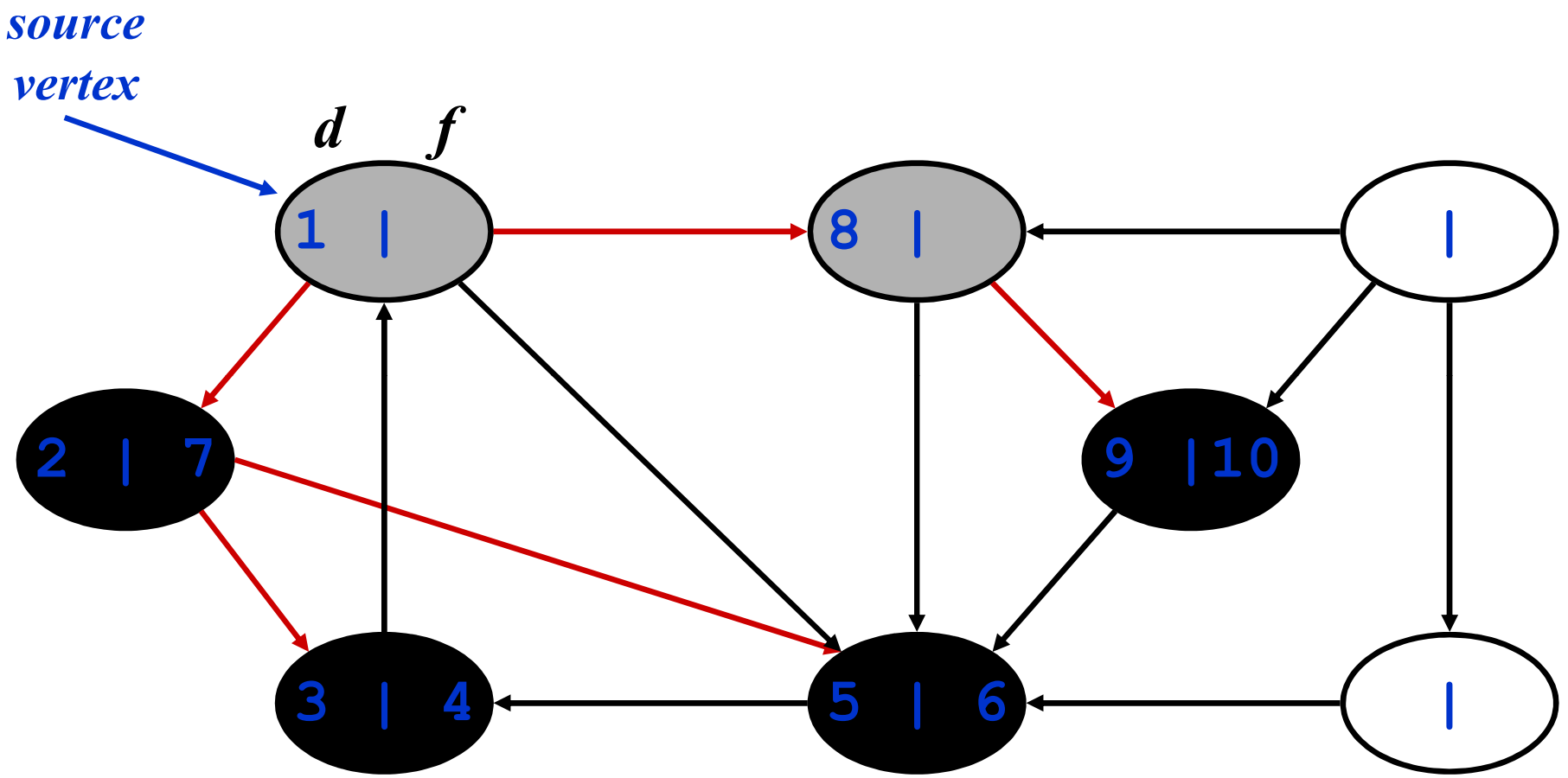


DFS Example

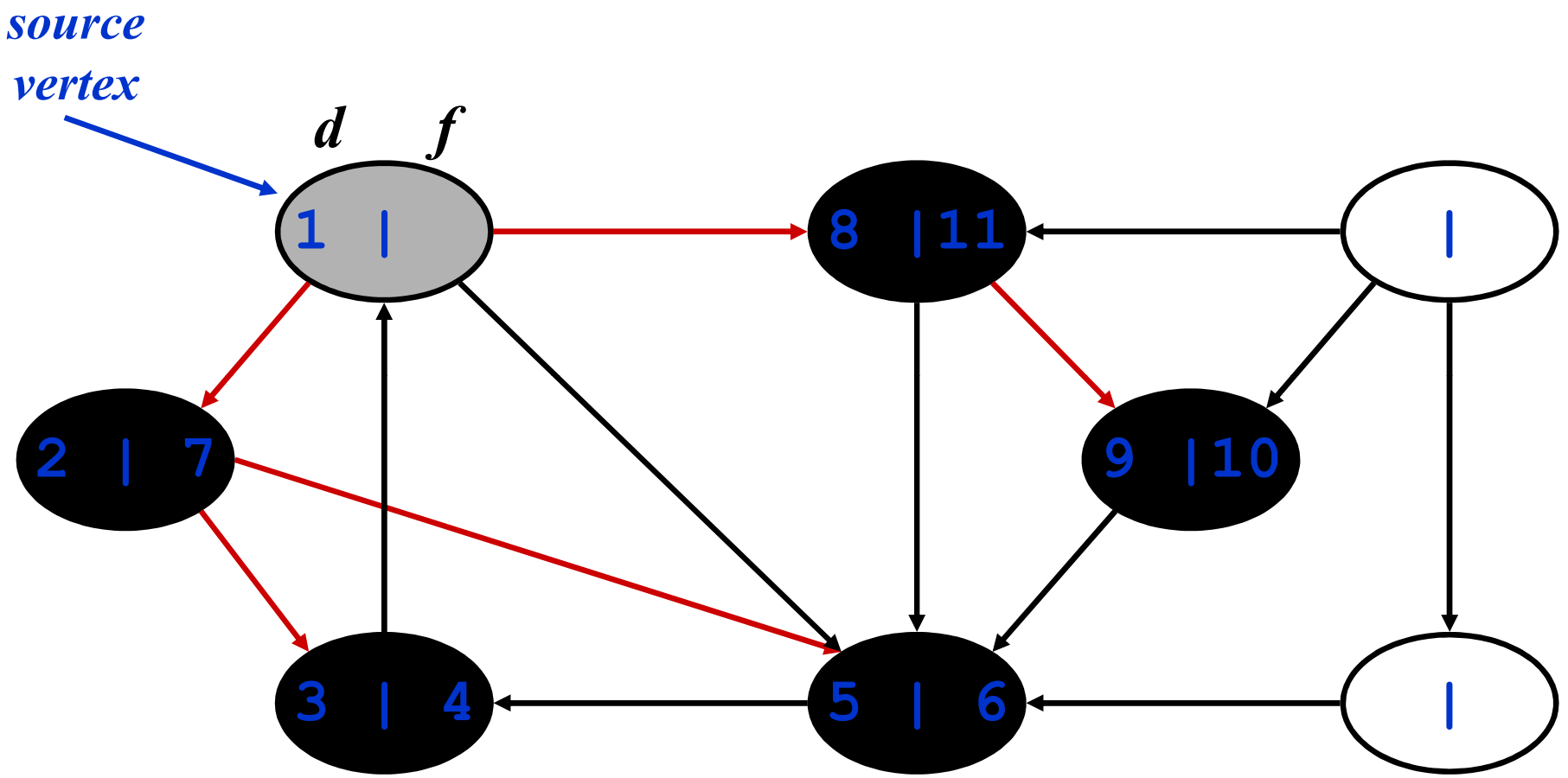


What is the structure of the grey vertices?
What do they represent?

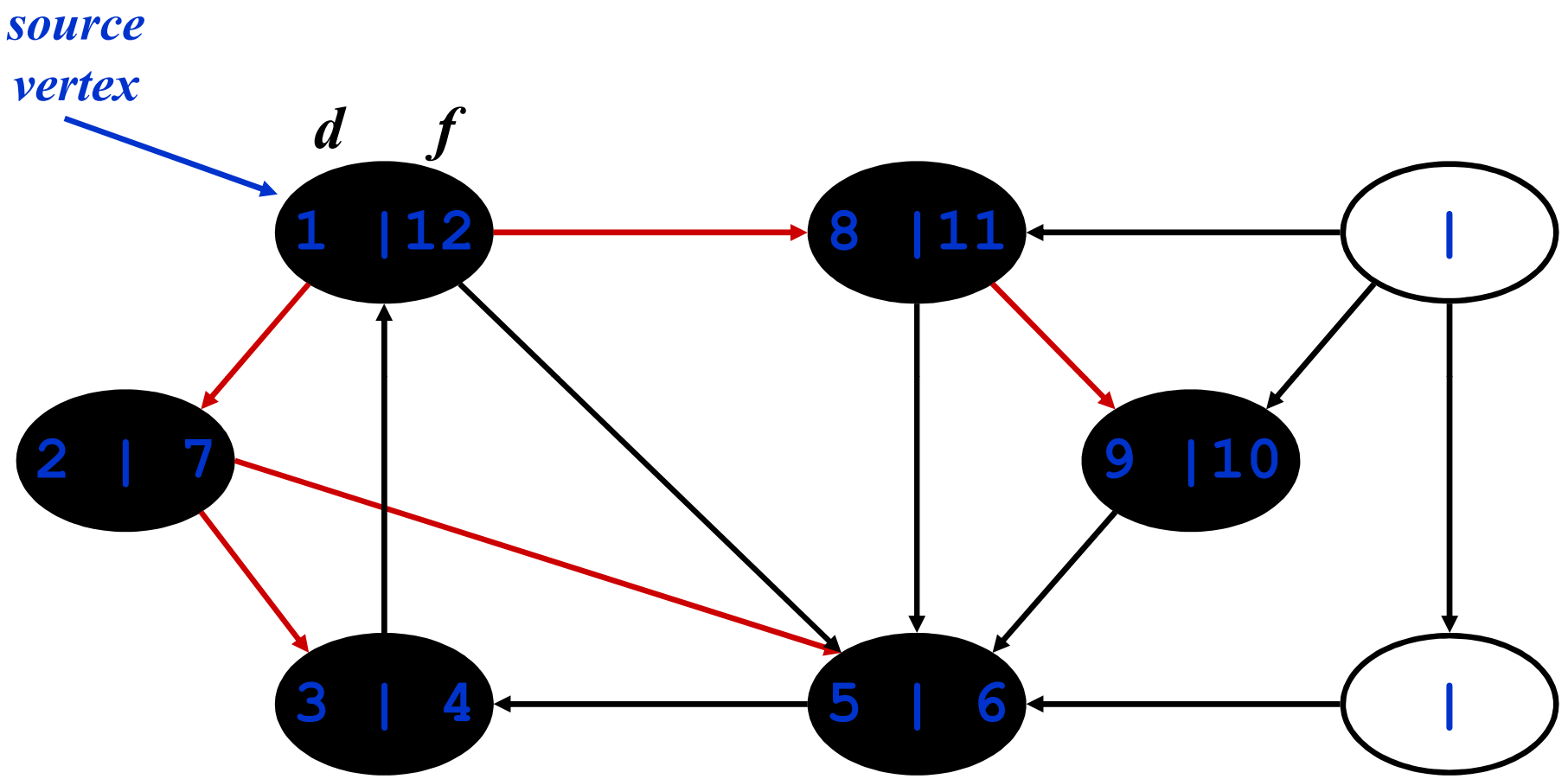
DFS Example



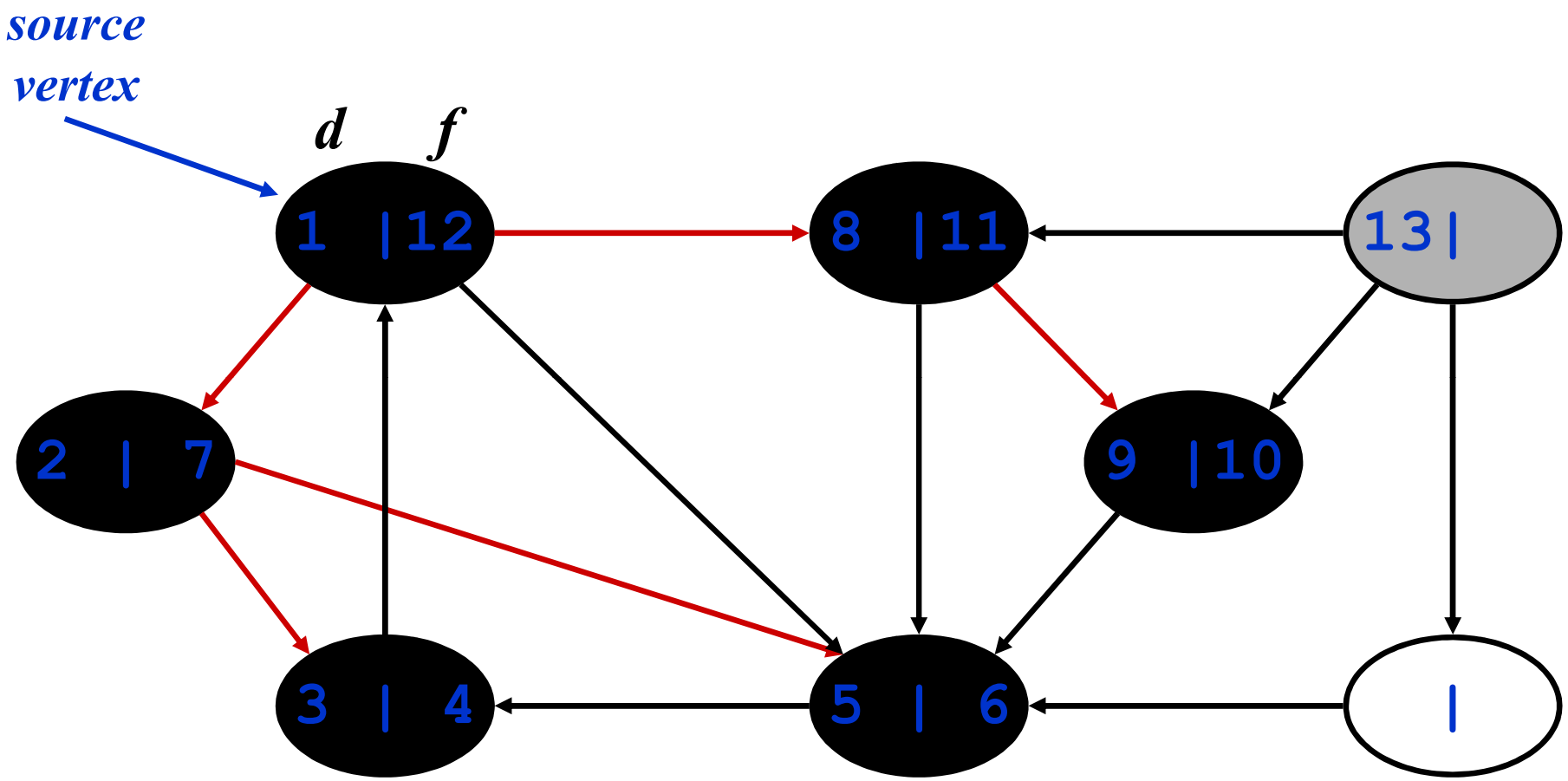
DFS Example



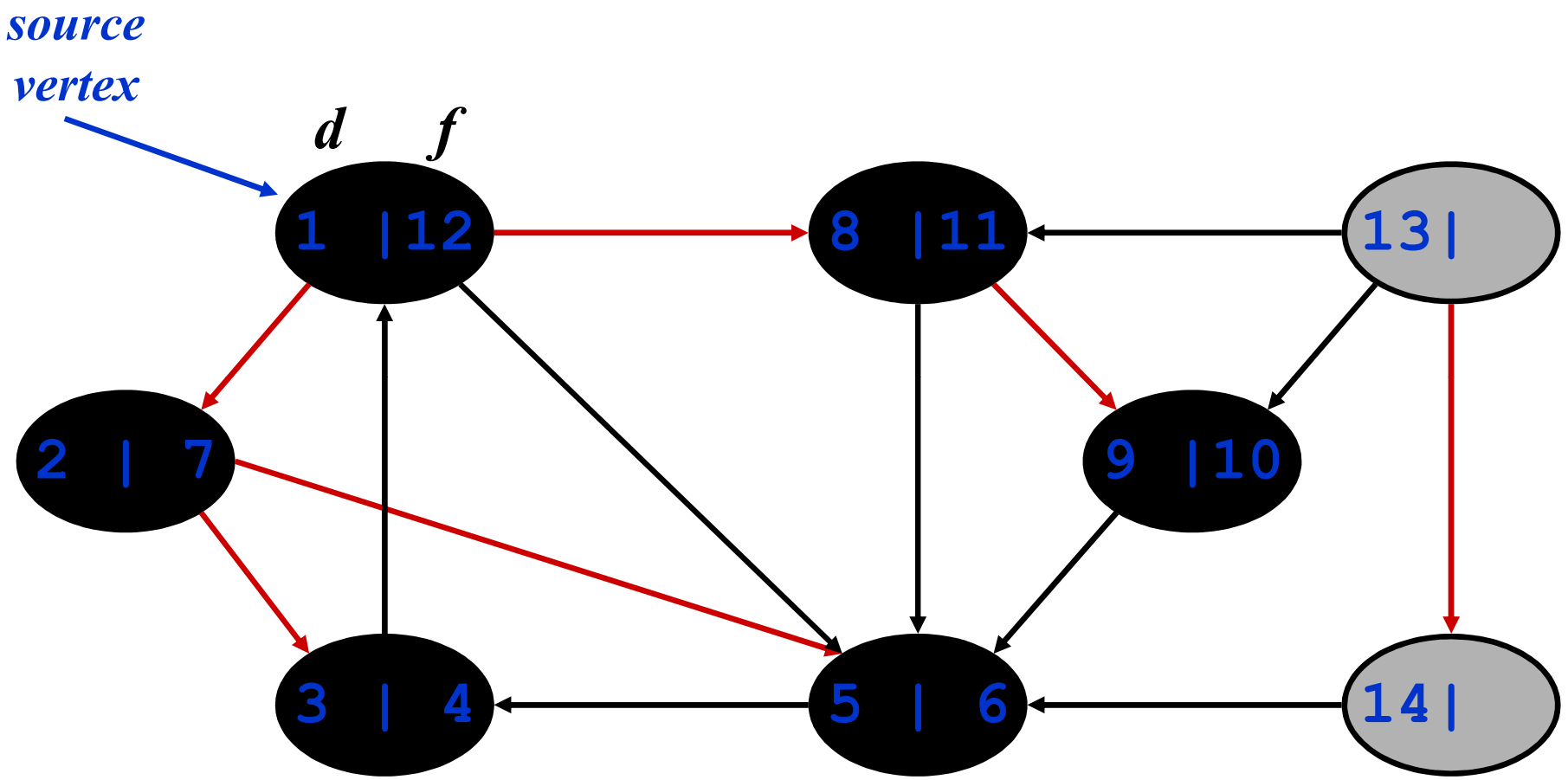
DFS Example



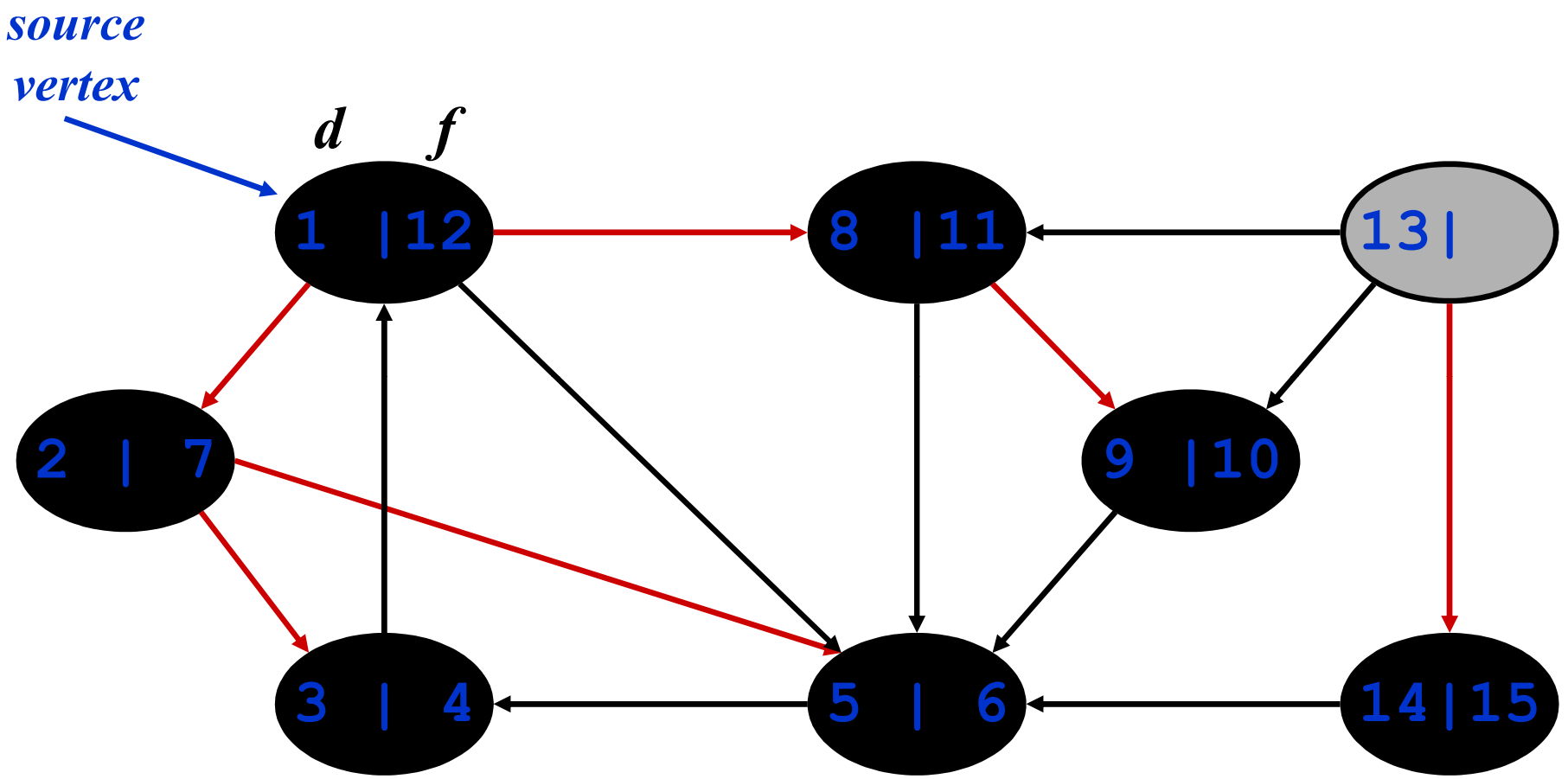
DFS Example



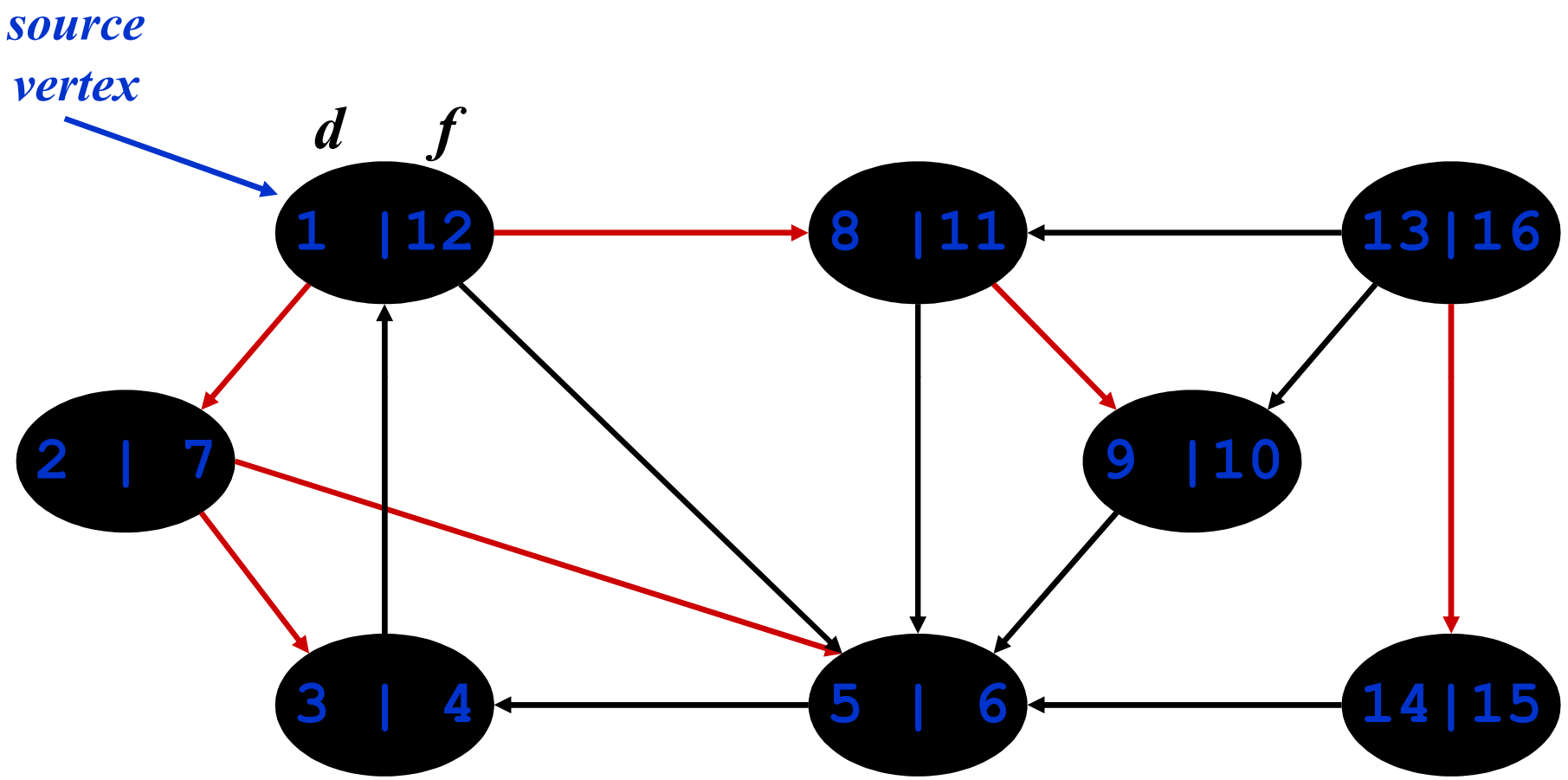
DFS Example



DFS Example



DFS Example



Vertices are timestamped by discovery time/finishing time

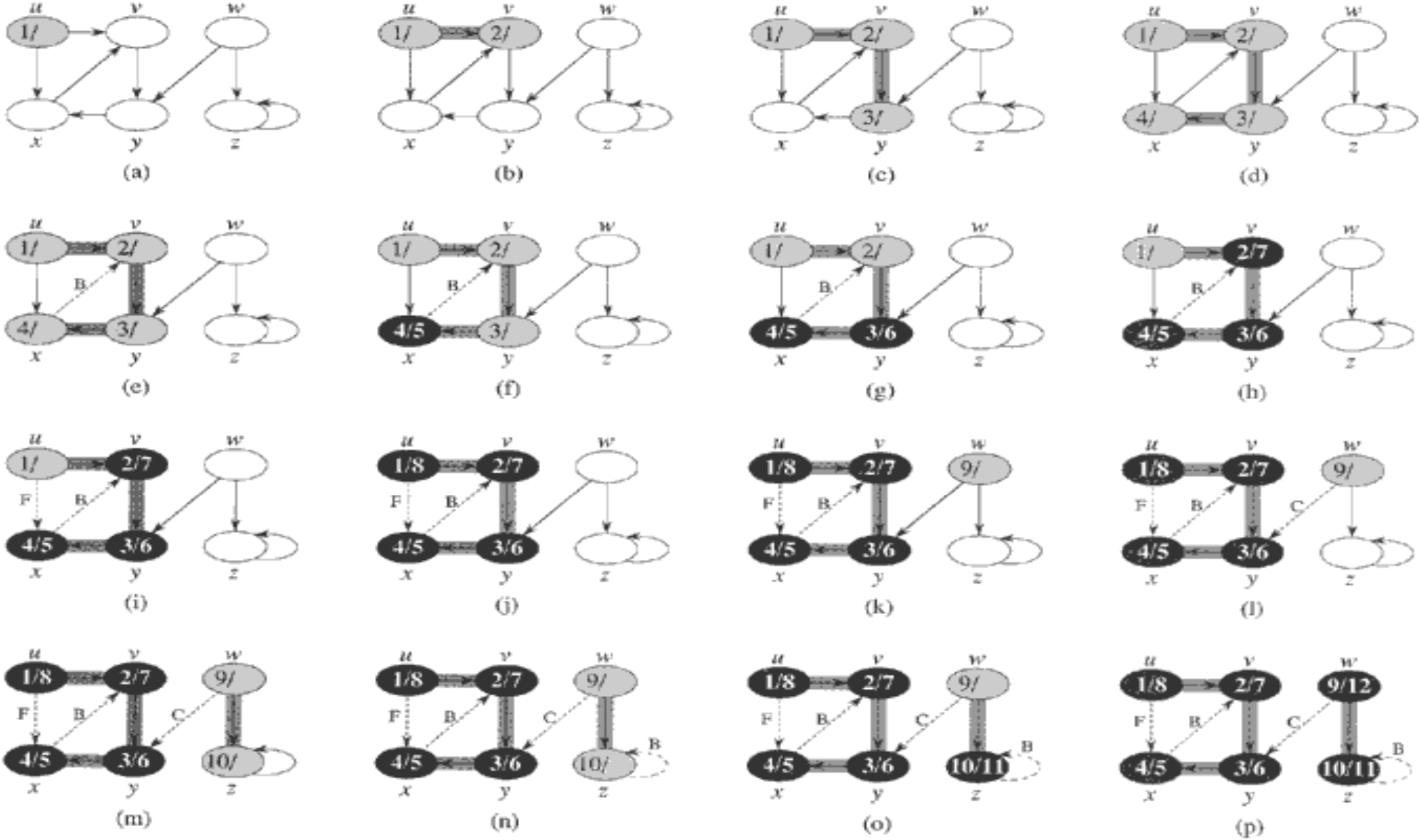


Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time.

Depth-First Search: The Code

```
DFS(G)

    for each vertex u ∈ V[G]

        color[u] = WHITE;
        Π[u]= NIl

    time = 0;
    for each vertex u ∈ V[G]

        if (color[u] == WHITE)
            DFS_Visit(u);
```

```
DFS_Visit(u)

    color[u] = GREY;
    time = time+1;
    d[u] = time;
    for each v ∈ Adj[u]

        if (color[v] == WHITE)
            Π[v]= u
            DFS_Visit(v);

    color[u] = BLACK;
    time = time+1;
    f[u] = time;
```

Depth-First Search: The Code

```
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)
```

Depth-First Search (Analysis)

- The loops on line 1-3 and lines 5-7 of DFS take time $O(V)$, exclusive DFS call.

- The procedure DFS-Visit is called exactly once for each vertex $v \in V$.

Depth-First Search: The Code

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

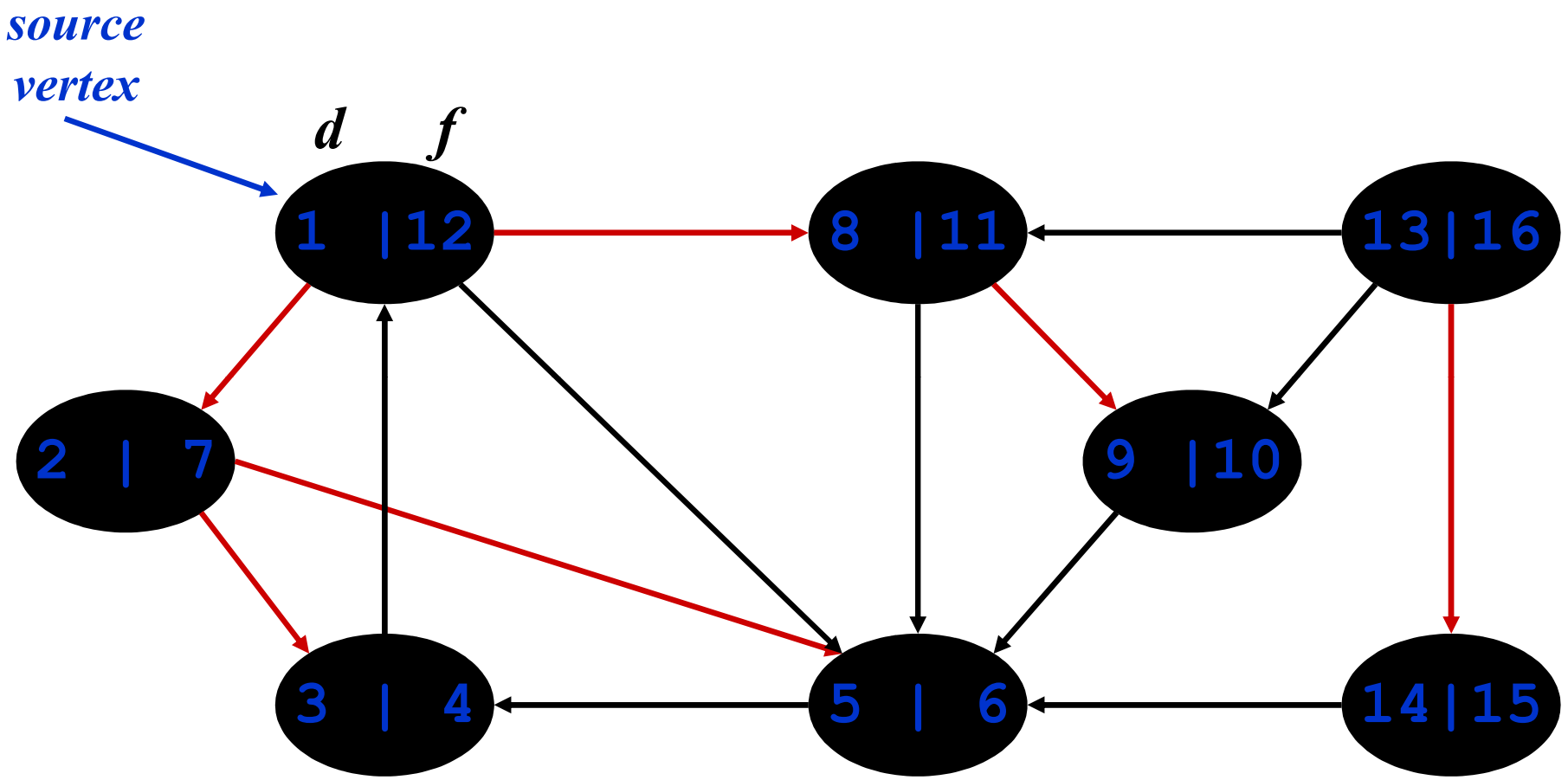
Depth-First Search (Analysis)

- Since DFS-Visit is invoked only on white vertices and the first thing it does is paint the vertex gray.
- During an execution of DFS-Visit(v), the loop on lines 4-7 is executed $|Adj[v]|$ times.
- The total cost of executing lines 4-7 of DFS-Visit is $O(E)$. The running time of DFS is $O(V+E)$.
  - Each loop in DFS_Visit can be attributed to an edge in the graph
  - Runs once/edge if directed graph, twice if undirected
  - Thus loop will run in O(E) time, algorithm O(V+E)

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
    - *Can tree edges form cycles? Why or why not?*

DFS Example

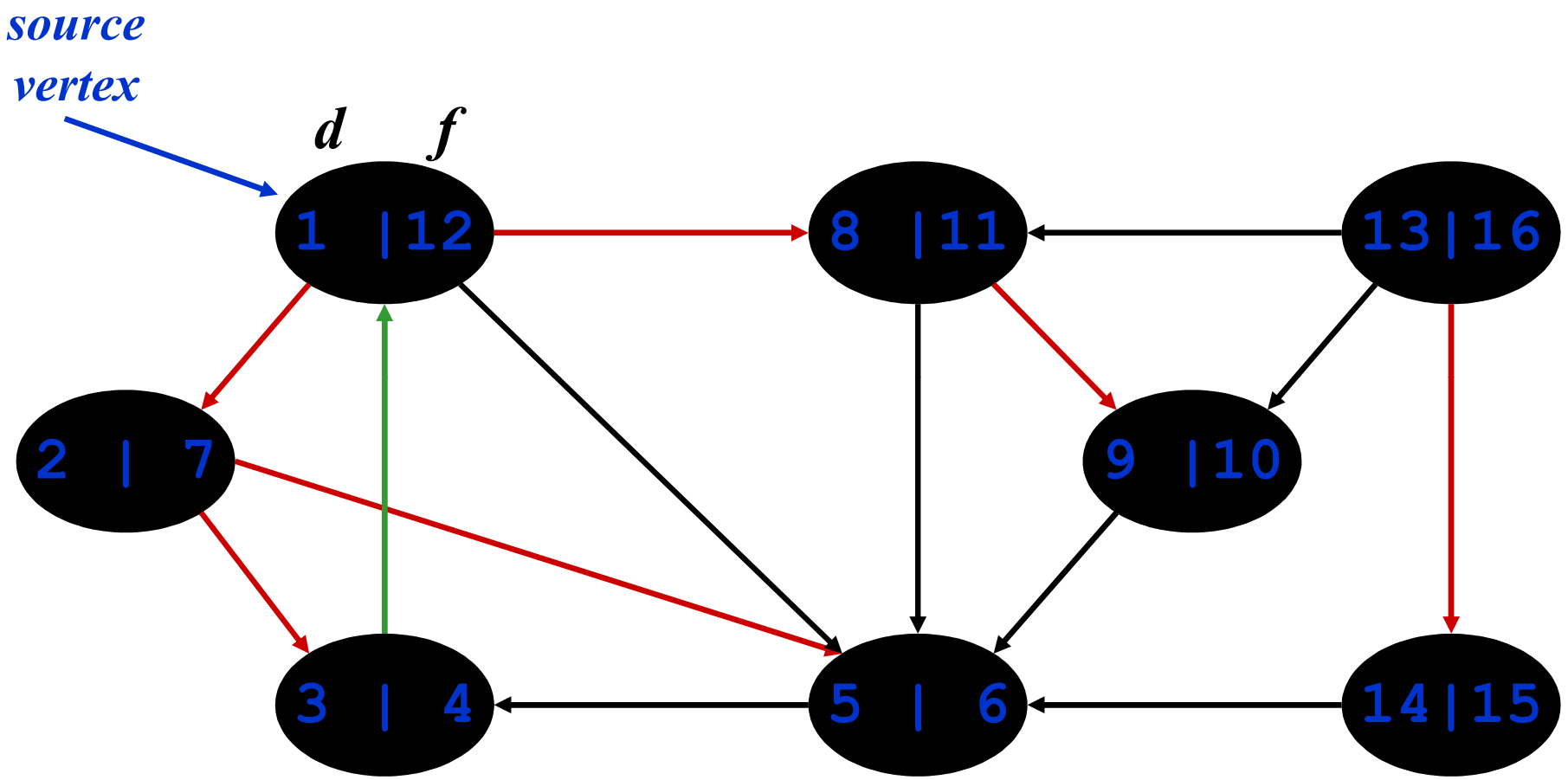


Tree edges

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
    - Encounter a grey vertex (grey to grey)

DFS Example

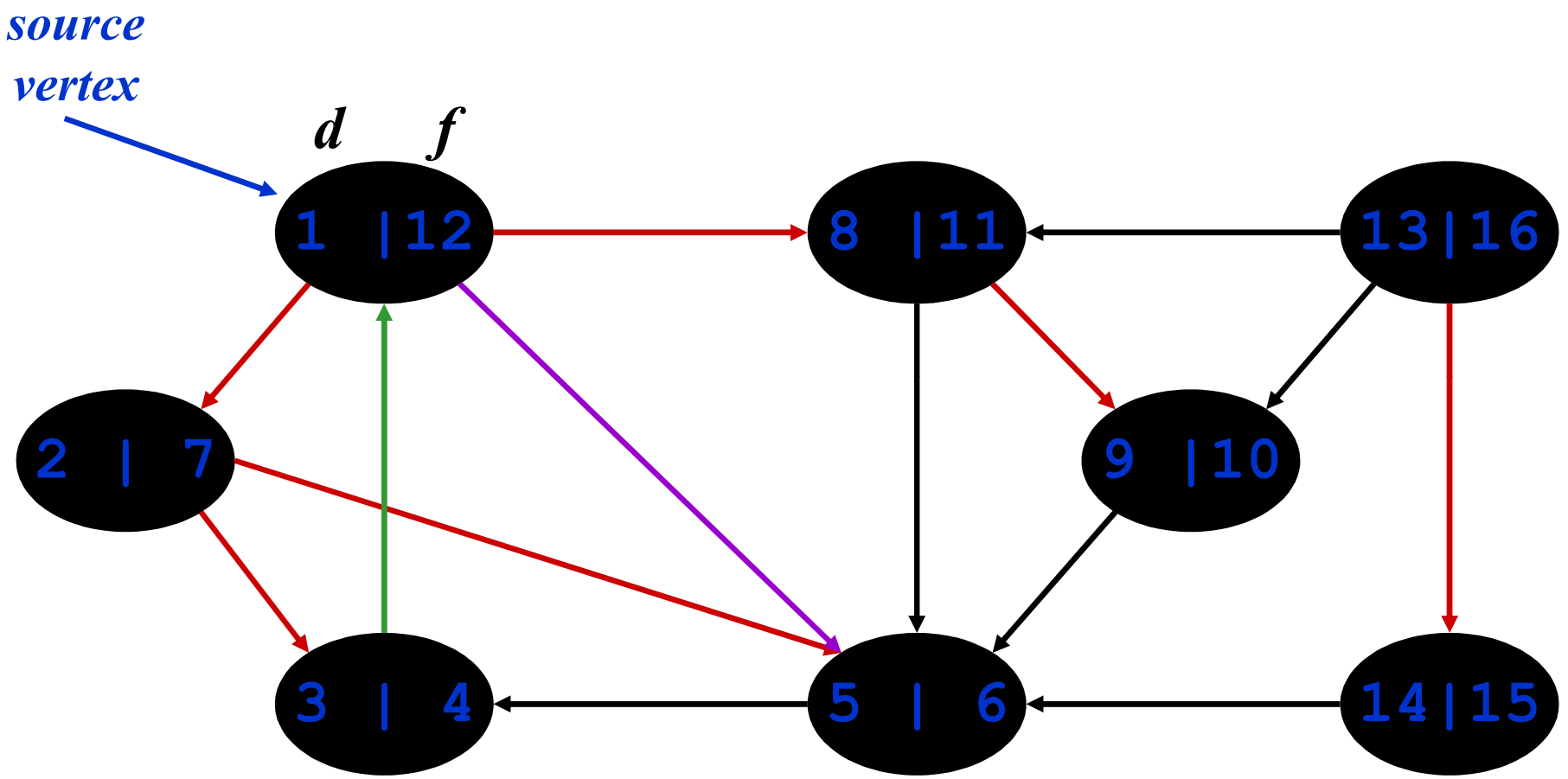


Tree edges *Back edges*

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
  - *Forward edge*: from ancestor to descendent
    - Not a tree edge, though
    - From grey node to black node

DFS Example

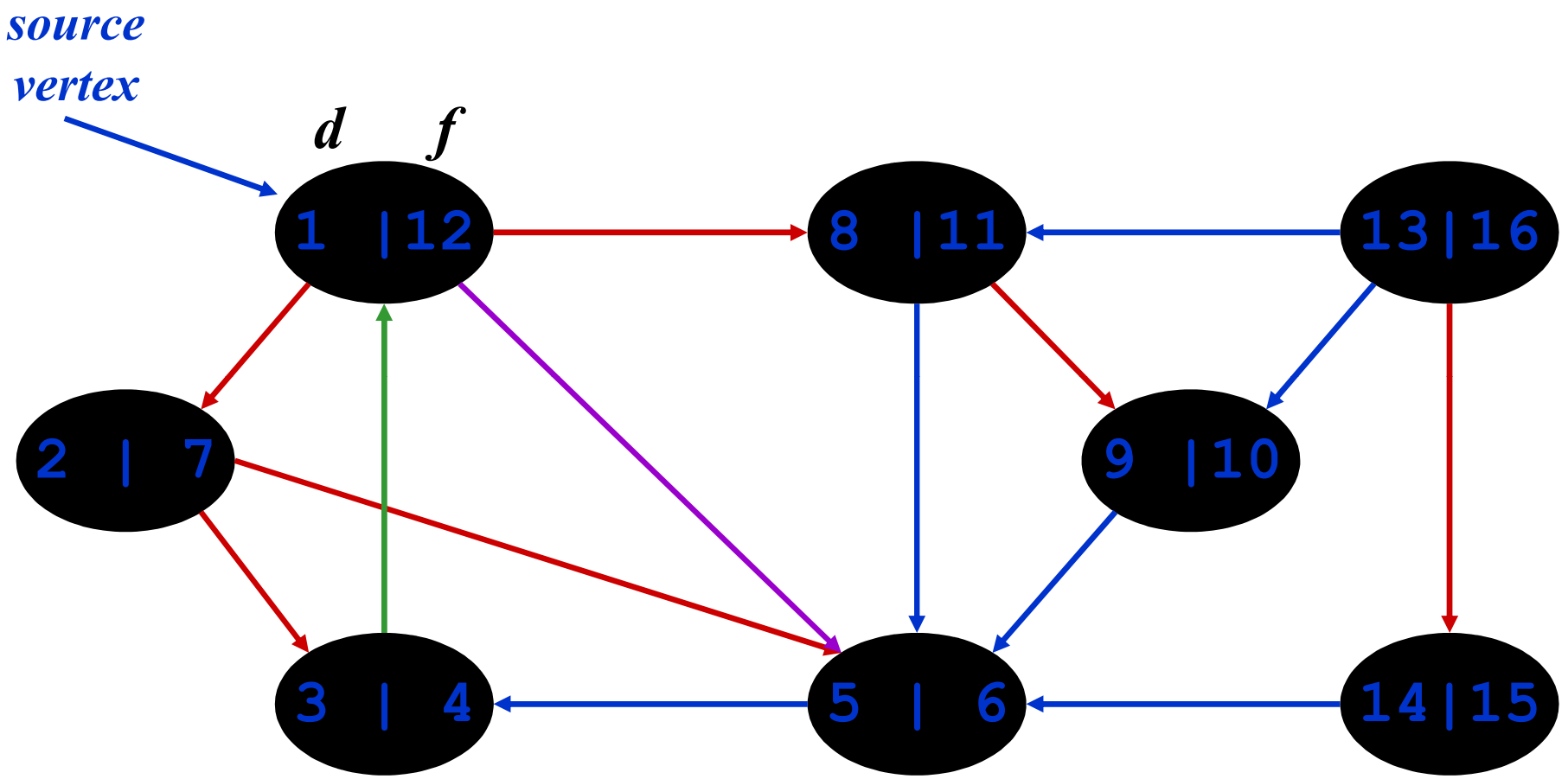


Tree edges *Back edges* *Forward edges*

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
  - *Forward edge*: from ancestor to descendent
  - *Cross edge*: between a tree or subtrees
    - From a grey node to a black node

DFS Example



Tree edges *Back edges* *Forward edges* *Cross 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
  - *Forward edge*: from ancestor to descendent
  - *Cross edge*: between a tree or subtrees
- Note: tree & back edges are important; most algorithms don't distinguish forward & cross

Properties of DFS

- Structure of DFS shows forest of trees
- DFS exactly mirrors the structure of recursive calls of DFS-Visit.
- Discovery and finishing time have parenthesis structure (i.e. discovery of vertex u with a left parenthesis "(u" and represent its finishing time by a right parenthesis "u)")

Properties of depth First search

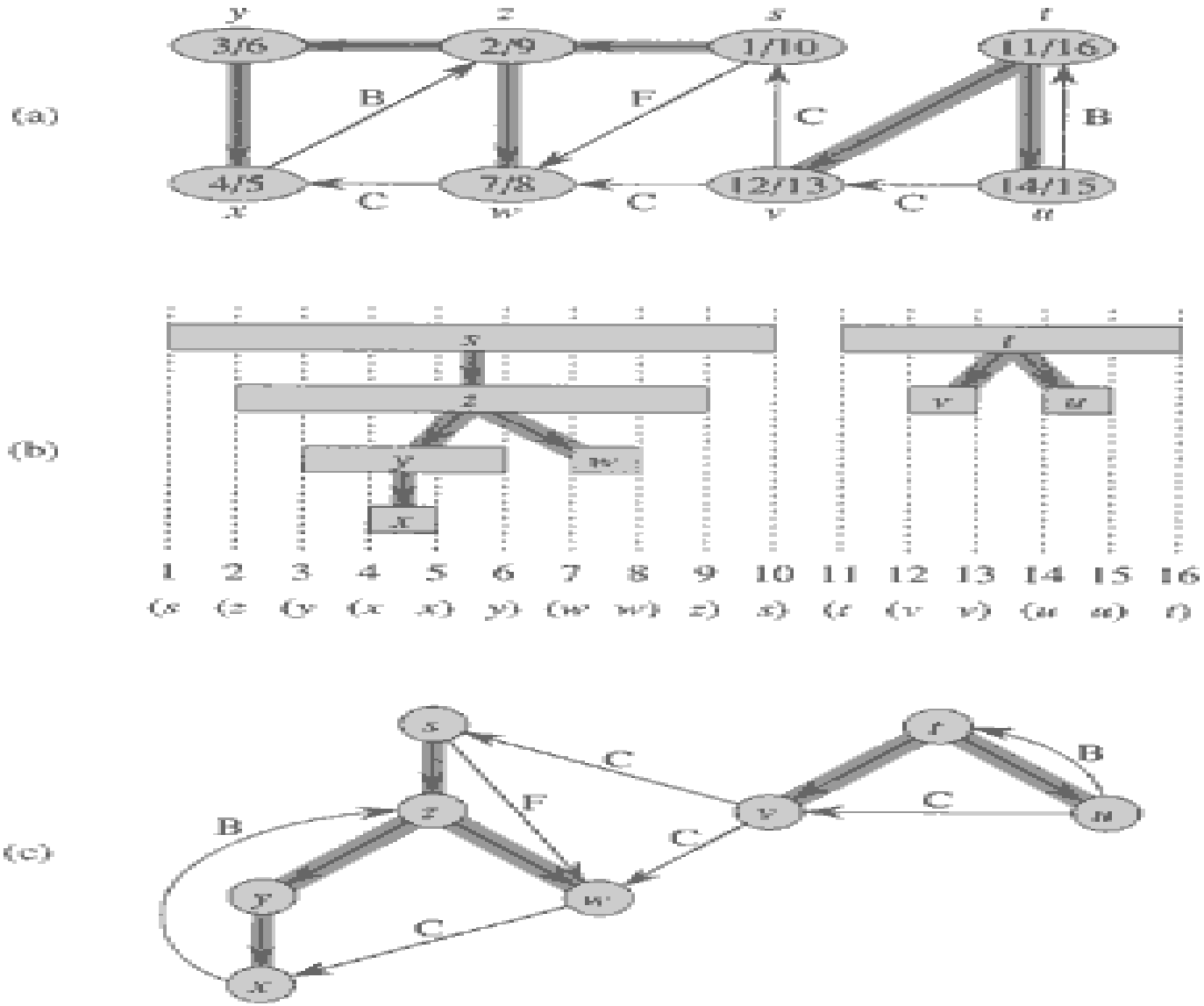
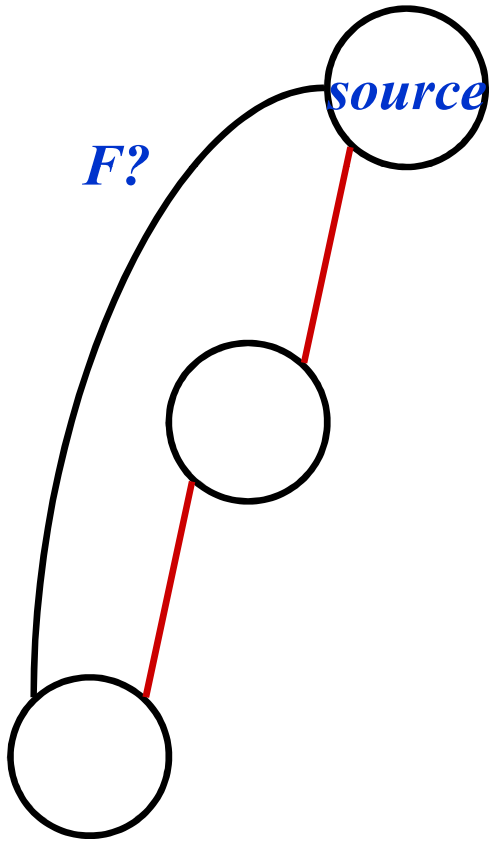


Figure 22.5 Properties of depth-first search. (a) The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 22.4. (b) Intervals for the discovery time and finishing time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finishing times of the corresponding vertex. Tree edges are shown. If two intervals overlap, then one is nested within the other, and the vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the larger. (c) The graph of part (a) redrawn with all tree and forward edges going down within a depth-first tree and all back edges going up from a descendant to an ancestor.

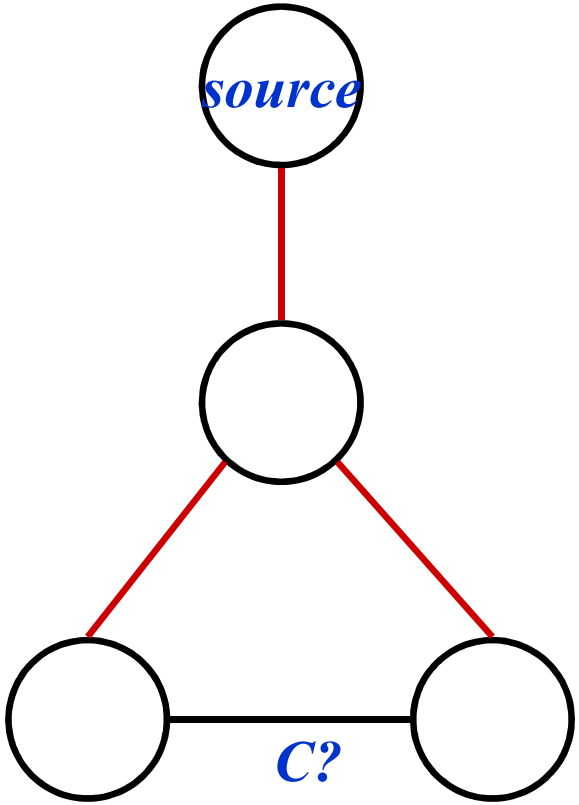
DFS: Kinds Of Edges

- Thm 23.9: If G is undirected, a DFS produces only tree and back edges
- Proof by contradiction:
  - Assume there's a forward edge
    - But F? edge must actually be a back edge (*why?*)



DFS: Kinds Of Edges

- Thm 23.9: If G is undirected, a DFS produces only tree and back edges
- Proof by contradiction:
  - Assume there's a cross edge
    - But C? edge cannot be cross:
    - must be explored from one of the vertices it connects, becoming a tree vertex, before other vertex is explored
    - So in fact the picture is wrong…both lower tree edges cannot in fact be tree edges



DFS And Graph Cycles

- Thm: An undirected graph is *acyclic* if 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 (*Why?*)
    - 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

DFS And Cycles

- *How would you modify the code to detect cycles?*

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

DFS And Cycles

- *What will be the running time?*

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

DFS And Cycles

- *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