Graph Algorithms

Graphs

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

Graph Variations

- Variations:
  - A *connected graph* has a path from every vertex to every other
  - In an *undirected graph:*
    - Edge (u,v) = edge (v,u)
    - No self-loops
  - In a *directed* graph:
    - Edge (u,v) goes from vertex u to vertex v, notated u→v

Graph Variations

- More 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 vertices
    - E.g., the call graph in a program (a function can get called from multiple points in another function)

Graphs

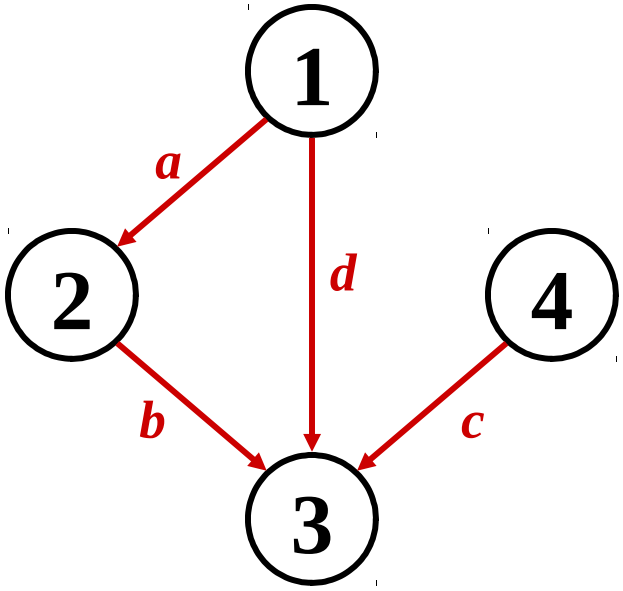
- We will typically express running times in terms of |E| and |V| (often dropping the |'s)
  - If $|E| \approx |V|^2$ the graph is *dense*
  - If $|E| \approx |V|$ the graph is *sparse*
- If you know you are dealing with dense or sparse graphs, different data structures may make sense

Representing Graphs

- 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

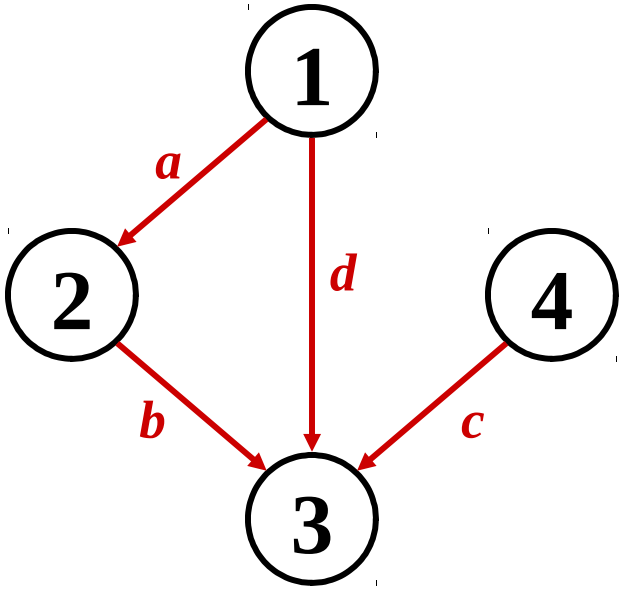
- Example:



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

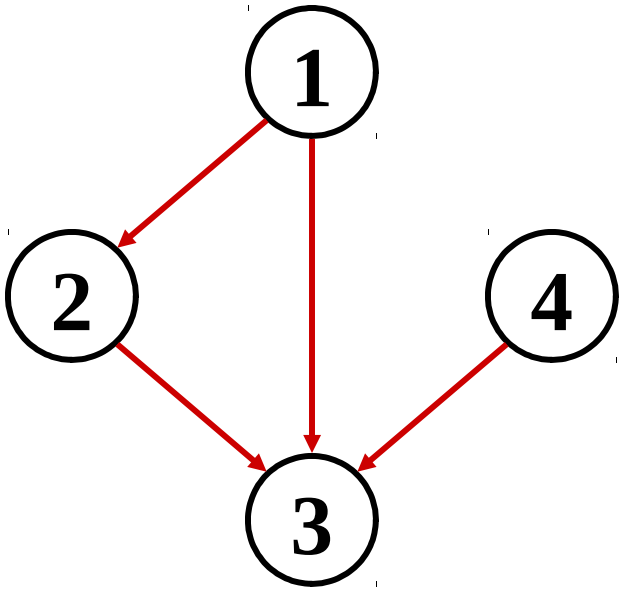
- *How much storage does the adjacency matrix require?*
- A: $O(V^2)$
- *What is the minimum amount of storage needed by an adjacency matrix representation of an undirected graph with 4 vertices?*
- A: 6 bits
  - Undirected graph → matrix is symmetric
  - No self-loops → don't need diagonal

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 respresentation

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

- 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
    $$\Sigma \text{ degree}(v) = 2\,|E|$$
    also $\Theta(V + E)$ storage
- So: Adjacency lists take O(V+E) storage

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
  - Note: might also build a *forest* if graph is not connected

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

- Again will associate vertex “colors” to guide the algorithm
  - 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

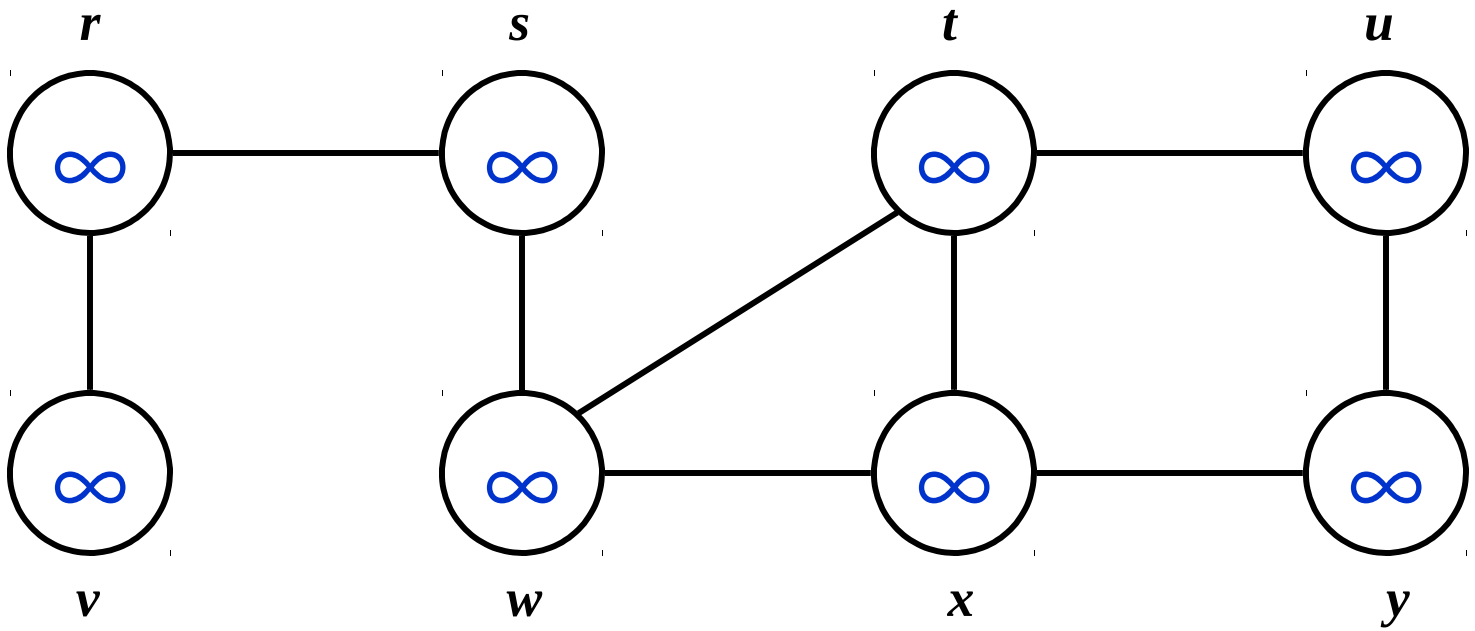
Breadth-First Search

```
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;
    }
}
```

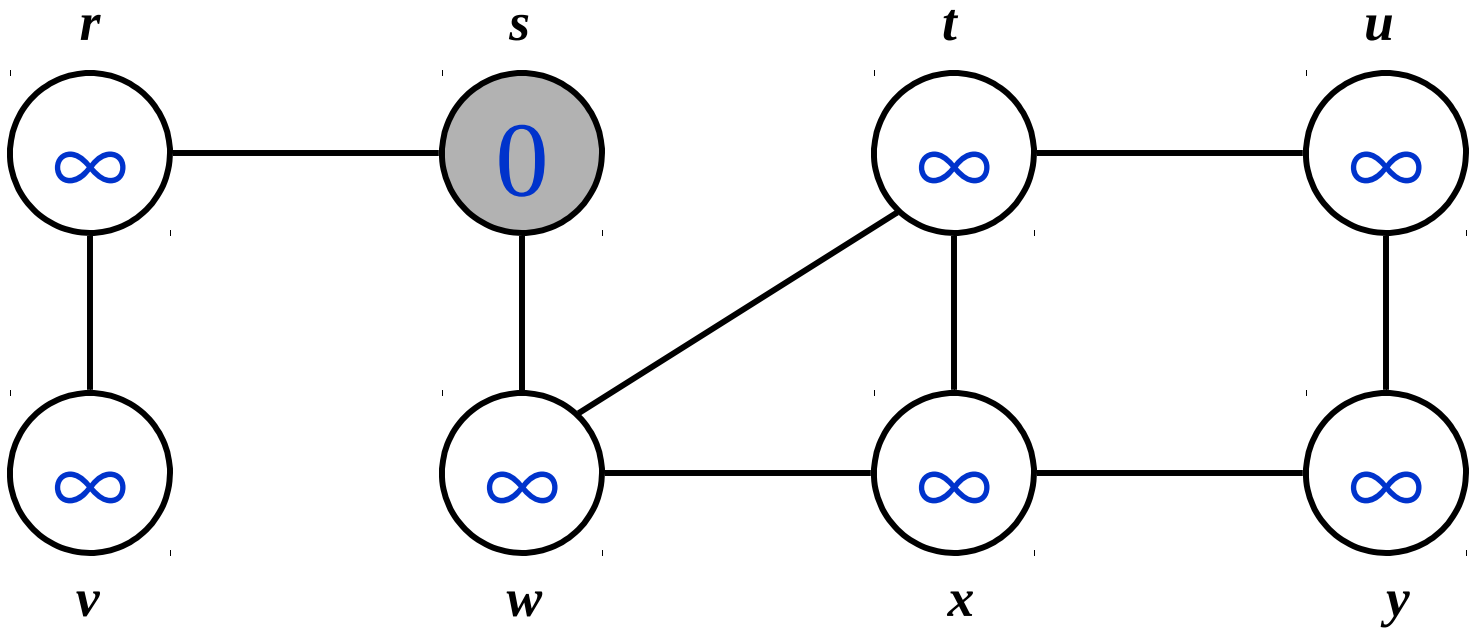
What does `v->d` *represent?*

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

Breadth-First Search: Example

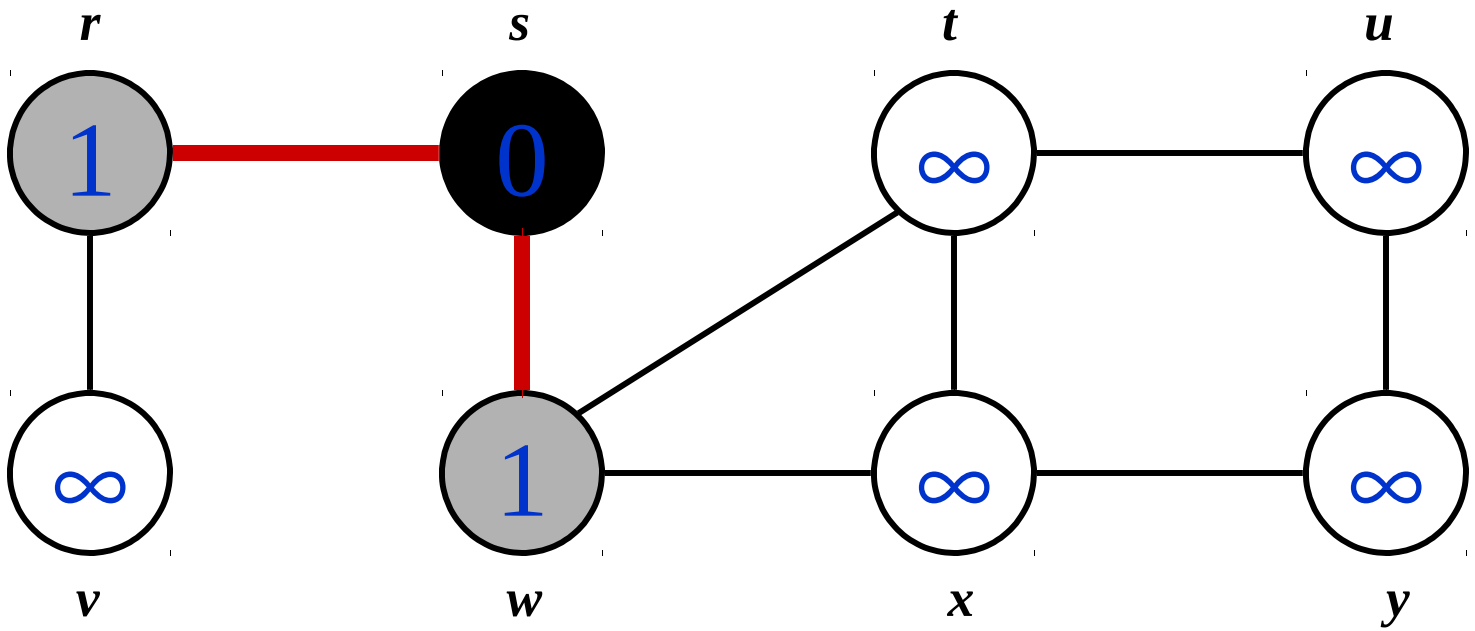


Breadth-First Search: Example



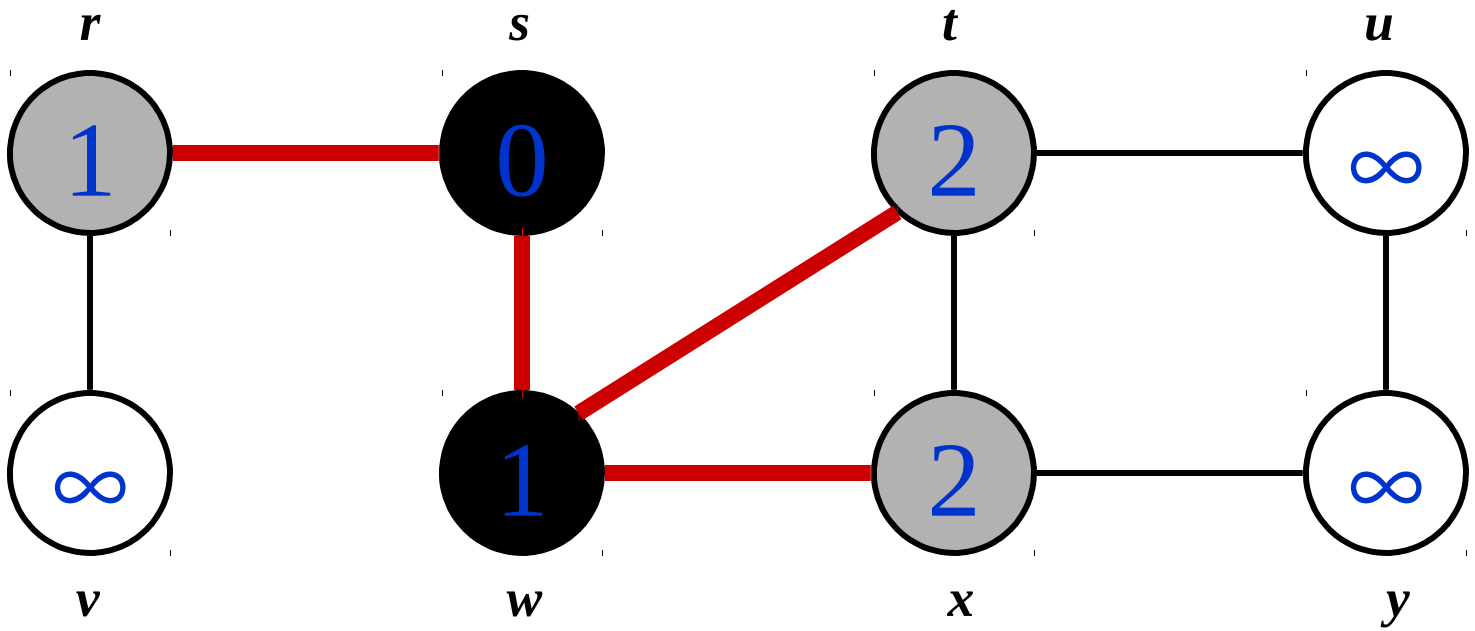
Q: | *s* |

Breadth-First Search: Example



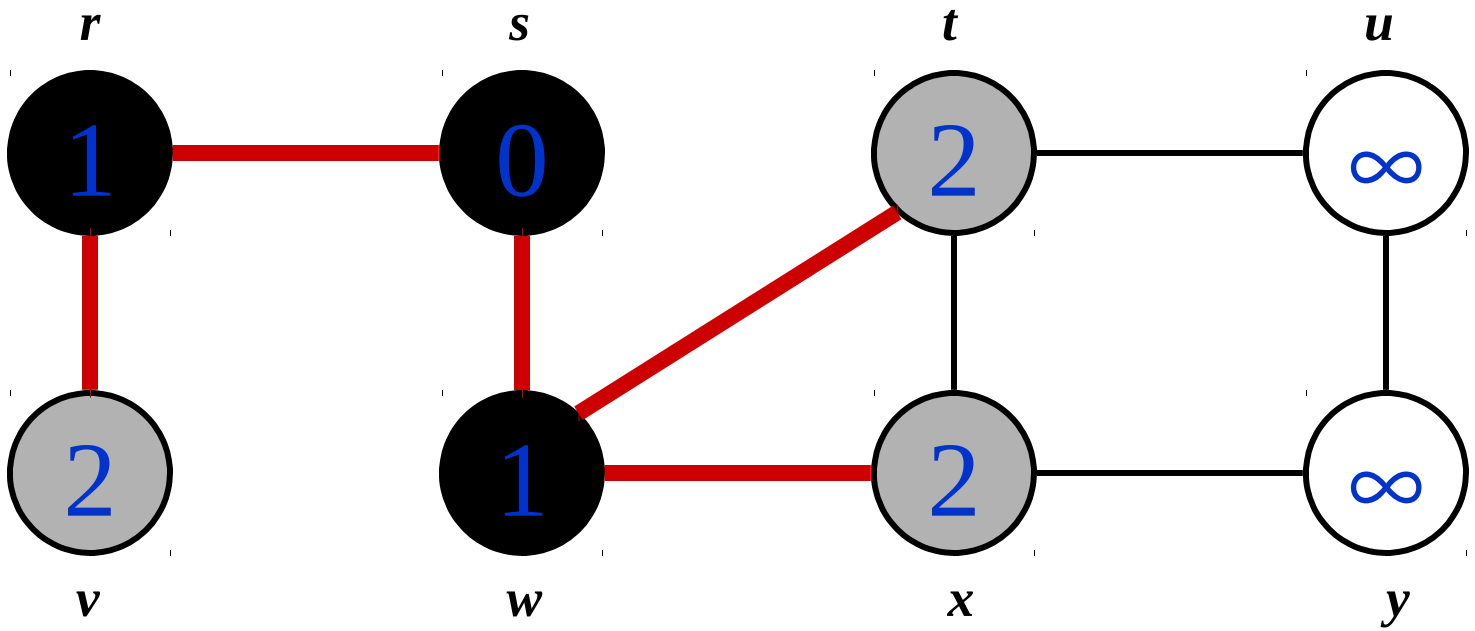
Q: | *w* | *r* |

Breadth-First Search: Example



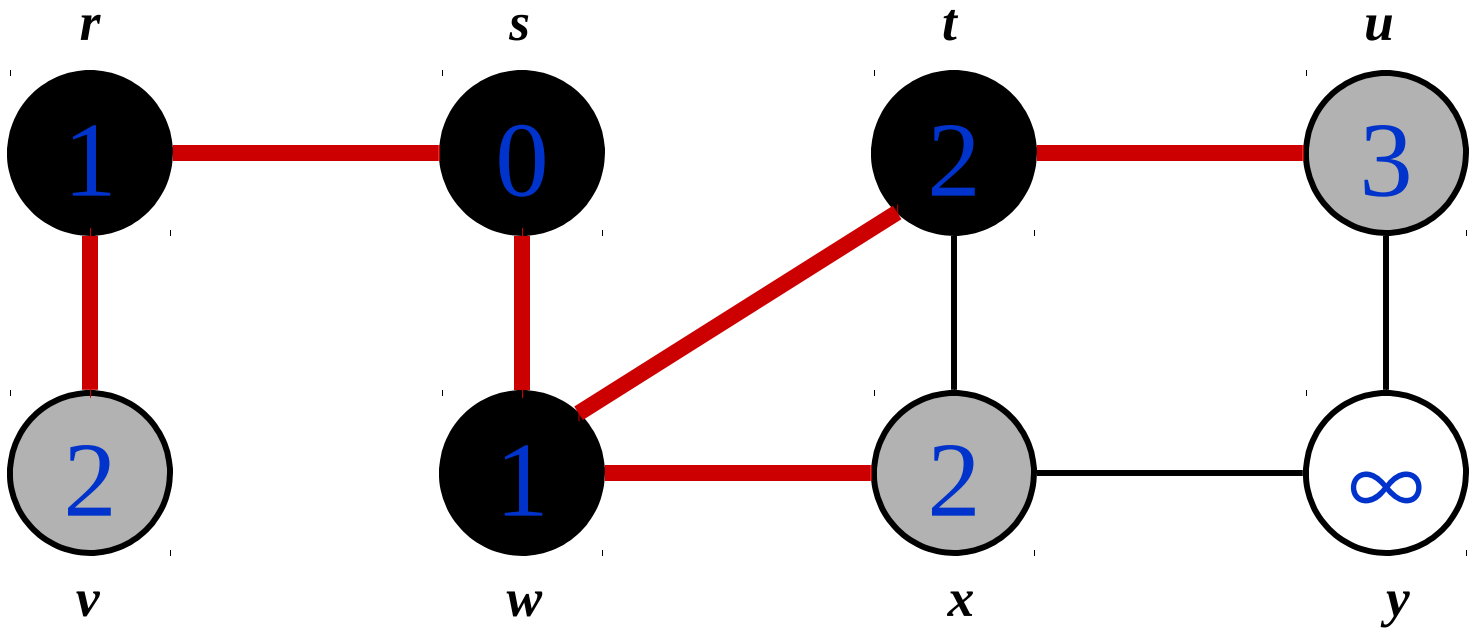
Q: | *r* | *t* | *x* |

Breadth-First Search: Example



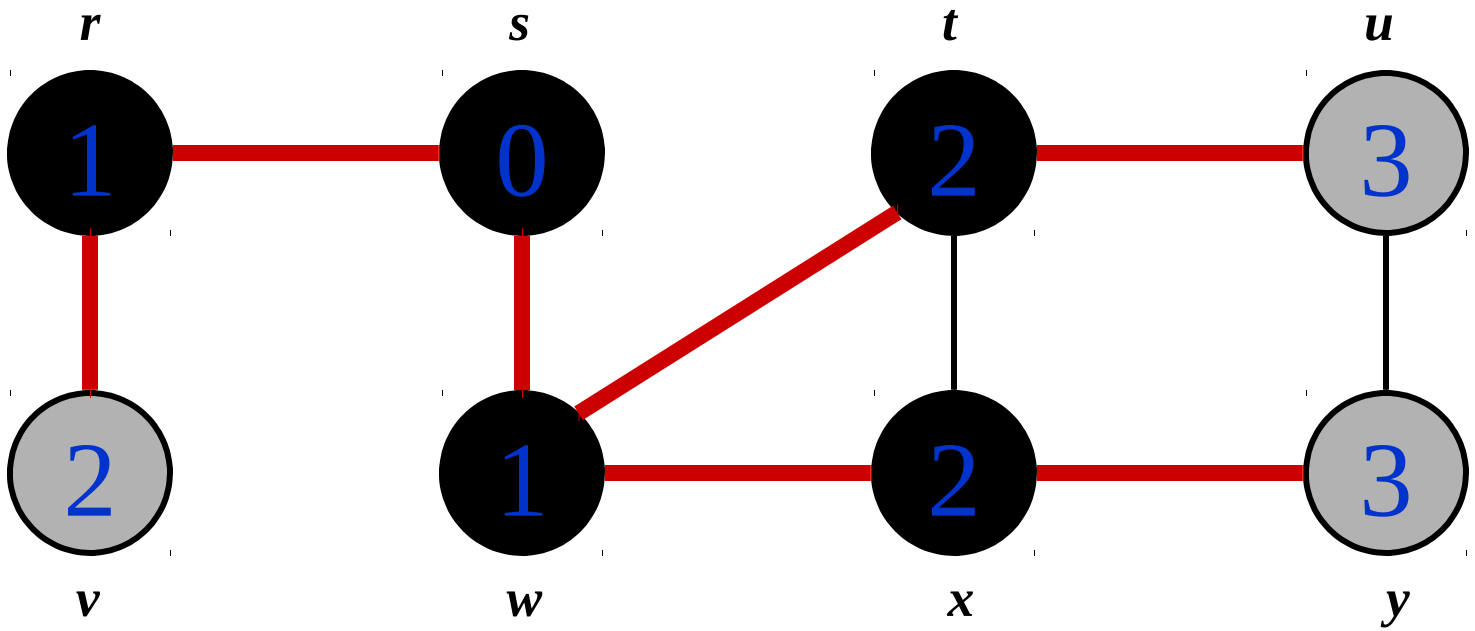
Q: | *t* | *x* | *v* |

Breadth-First Search: Example



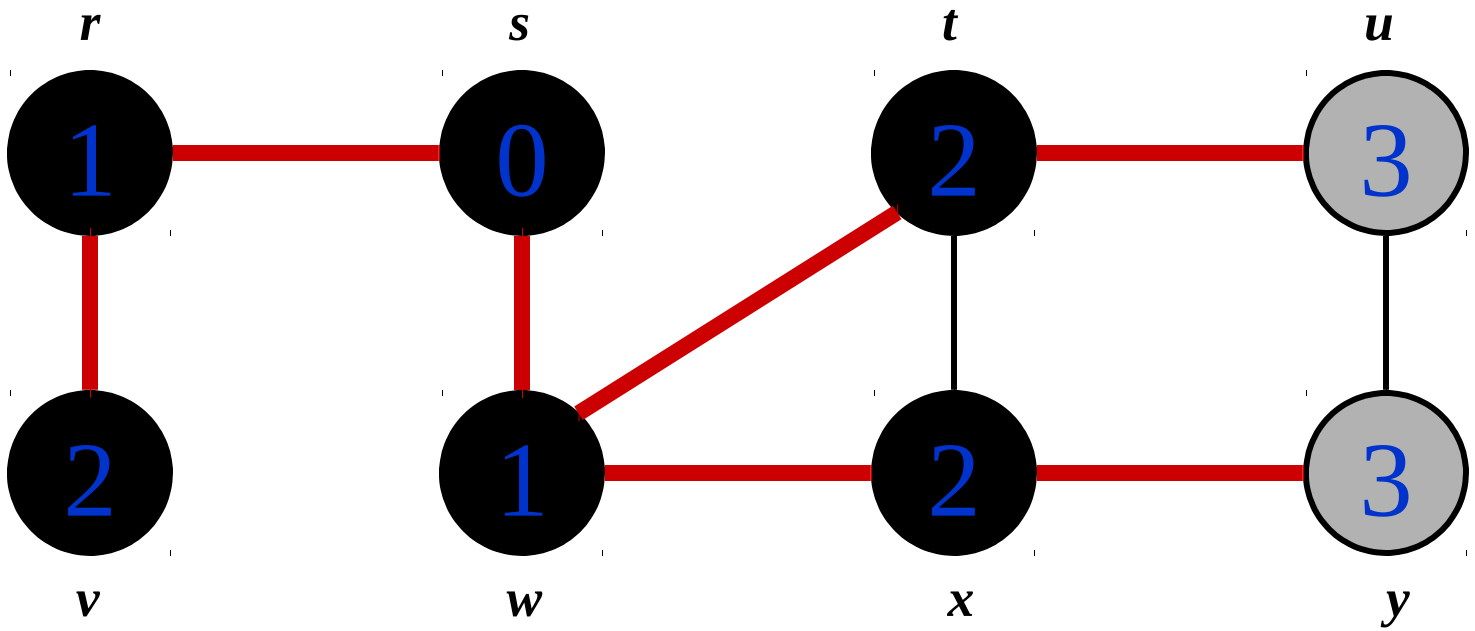
Q: | *x* | *v* | *u* |

Breadth-First Search: Example



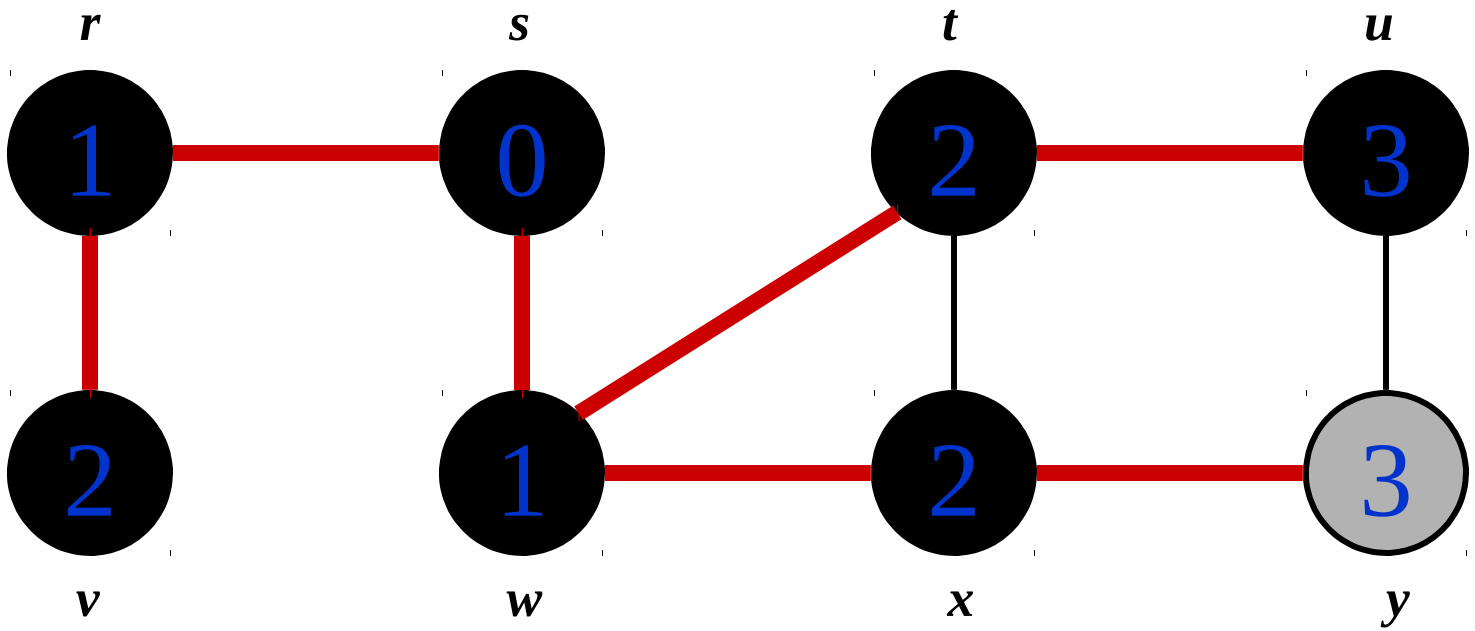
Q: | *v* | *u* | *y* |

Breadth-First Search: Example



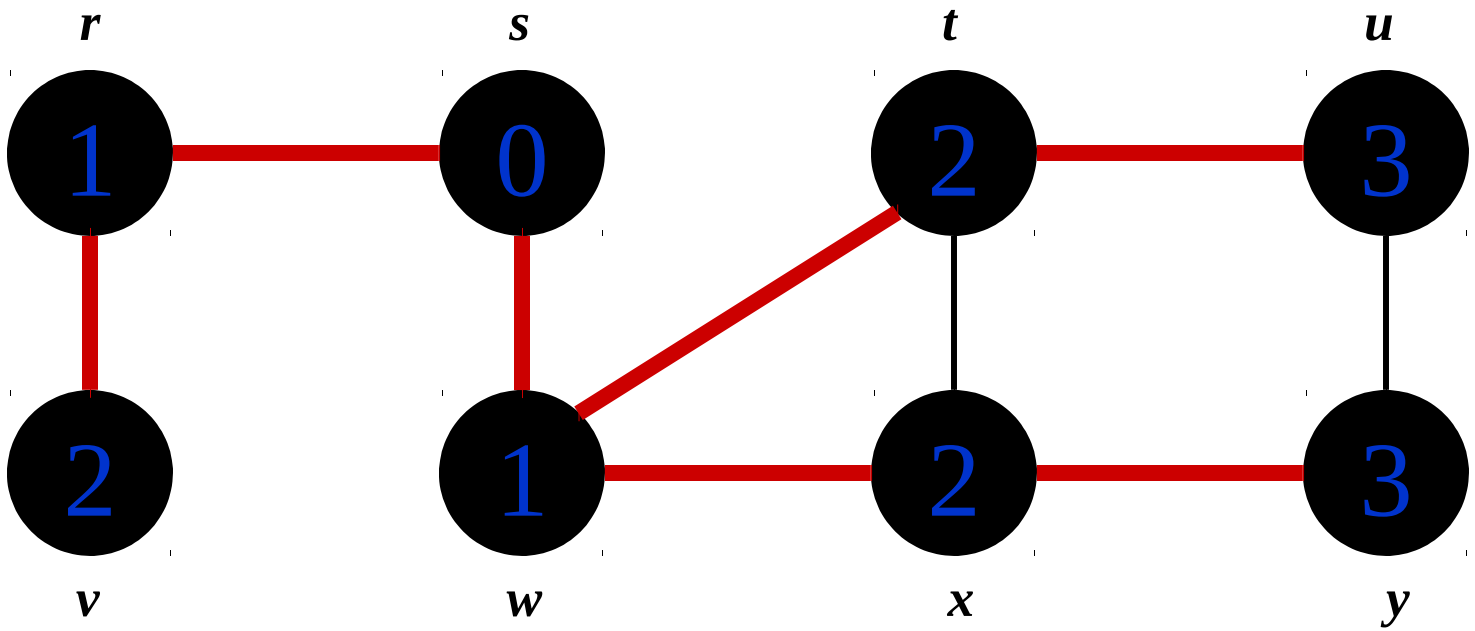
Q: | u | y |

Breadth-First Search: Example



Q: y

Breadth-First Search: Example



Q: Ø

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

Review: Depth-First Search

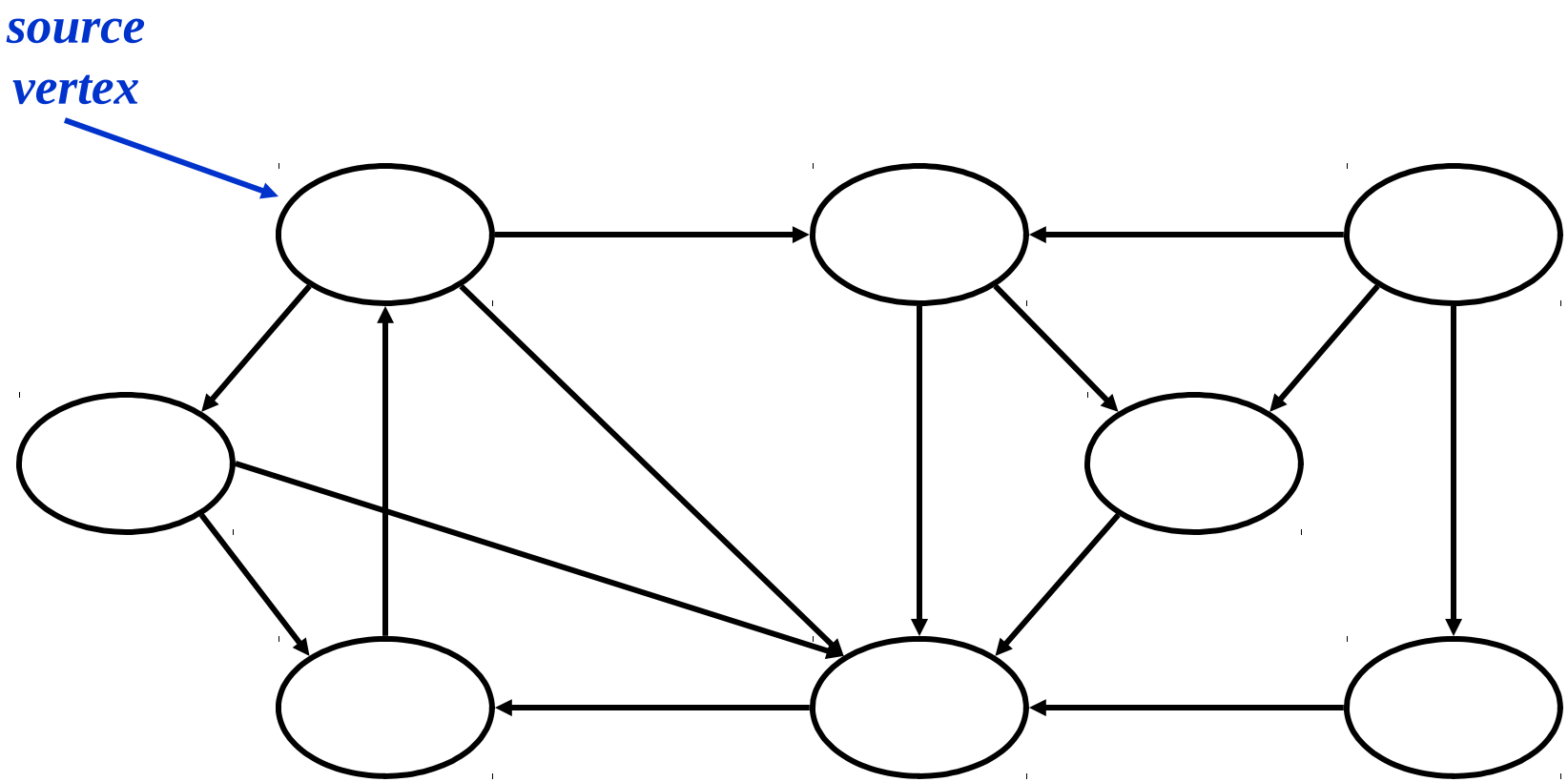
- *Depth-first search* is another strategy for exploring a graph
  - Explore “deeper” in the graph whenever possible
  - Edges are explored out of the most recently discovered vertex *v* that still has unexplored edges
  - When all of *v*’s edges have been explored, backtrack to the vertex from which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

Review: DFS Code

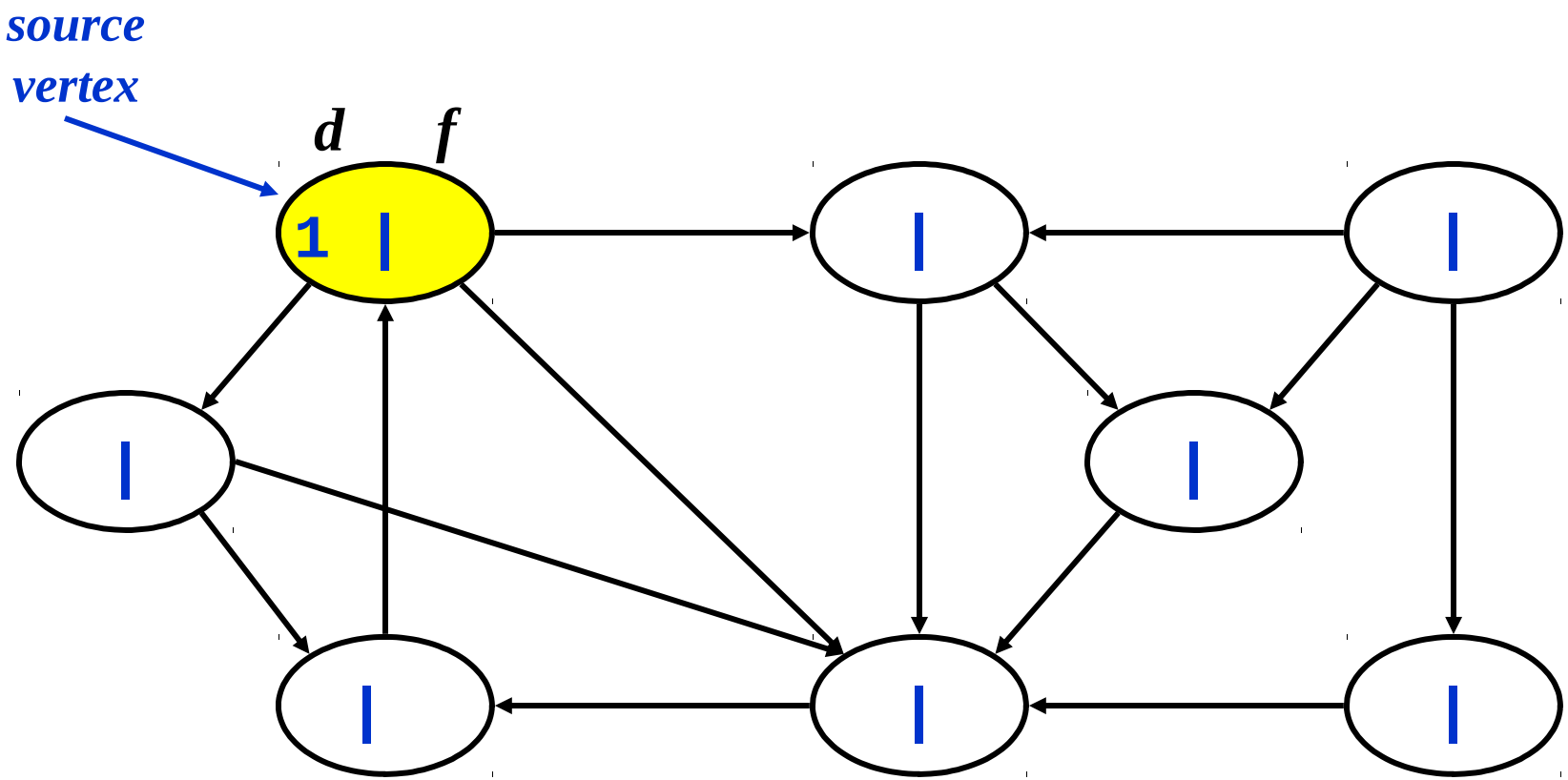
```
DFS(G)
{
   for each vertex u ∈ G->V
   {
      u->color = WHITE;
   }
   time = 0;
   for each vertex u ∈ G->V
   {
      if (u->color == WHITE)
         DFS_Visit(u);
   }
}
```

```
DFS_Visit(u)
{
   u->color = YELLOW;
   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;
}
```

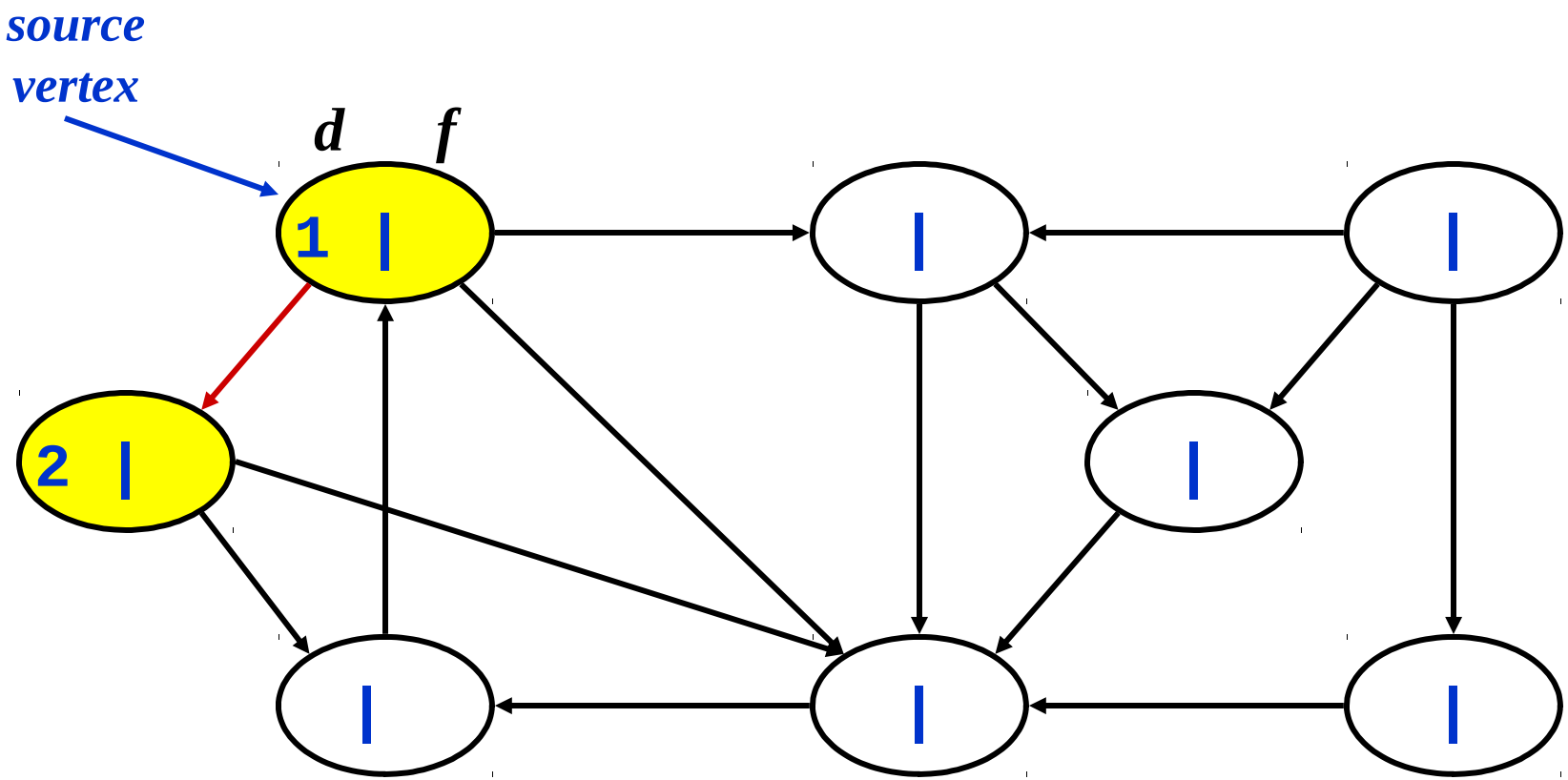
DFS Example



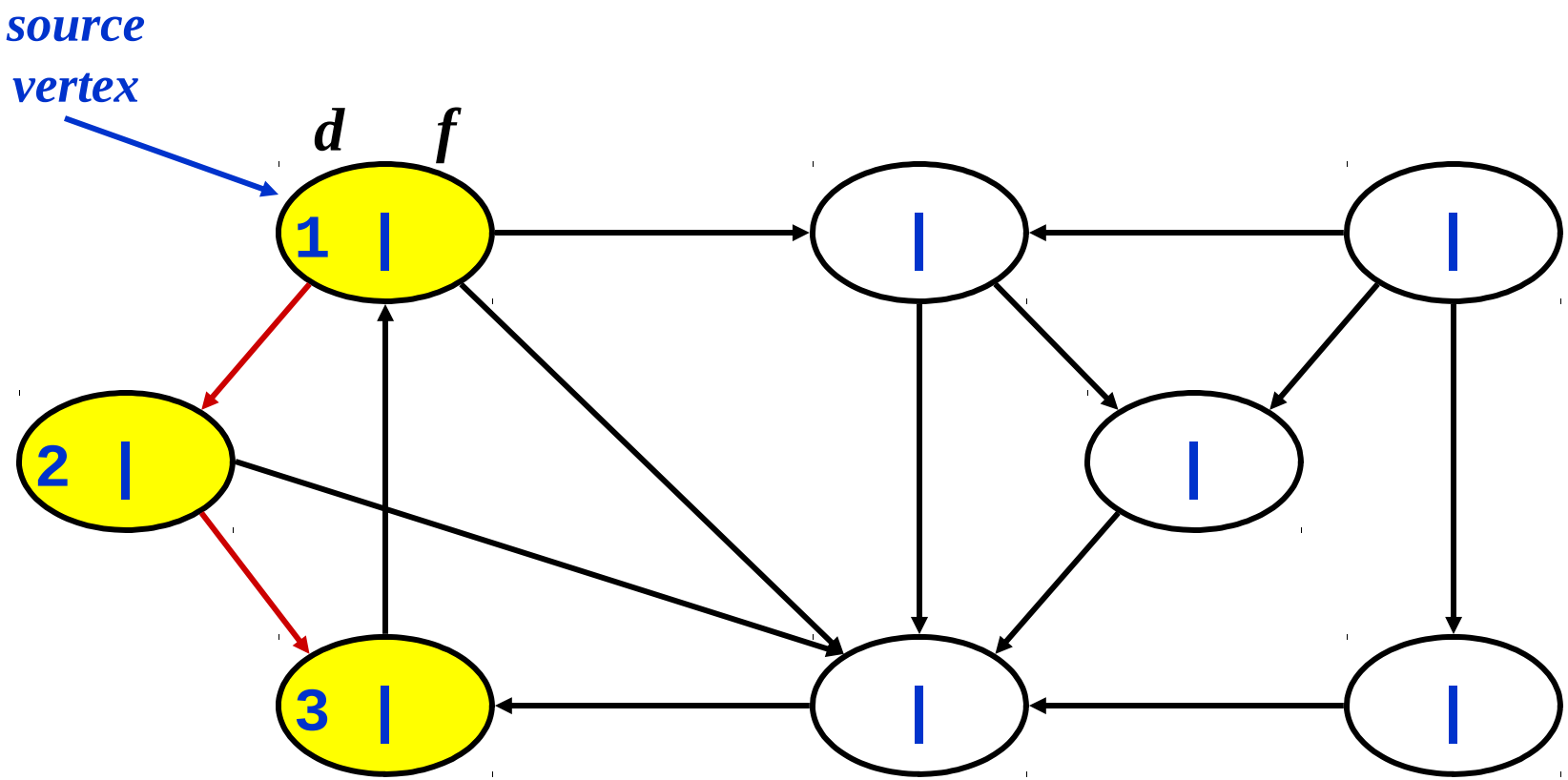
DFS Example



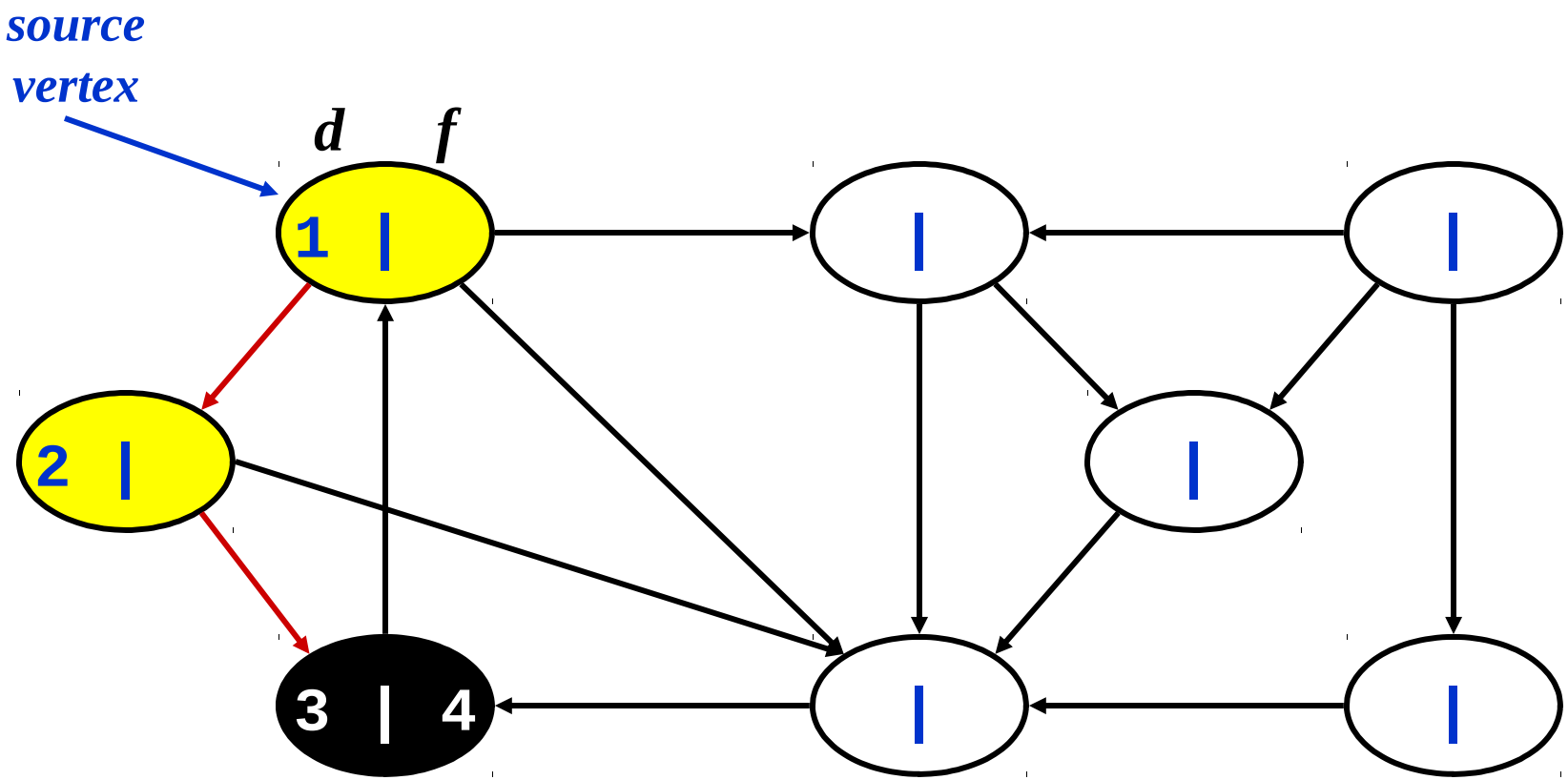
DFS Example



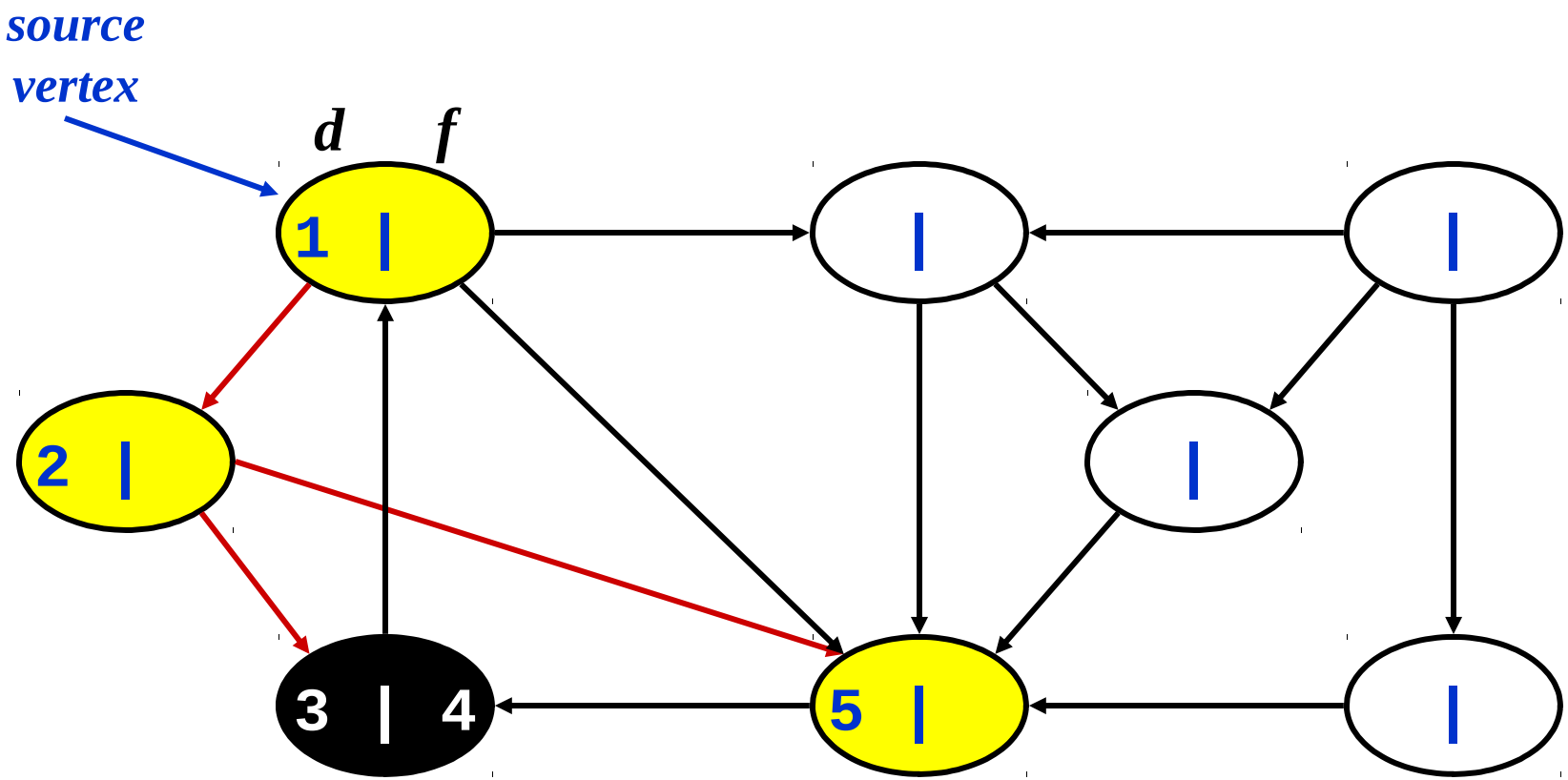
DFS Example



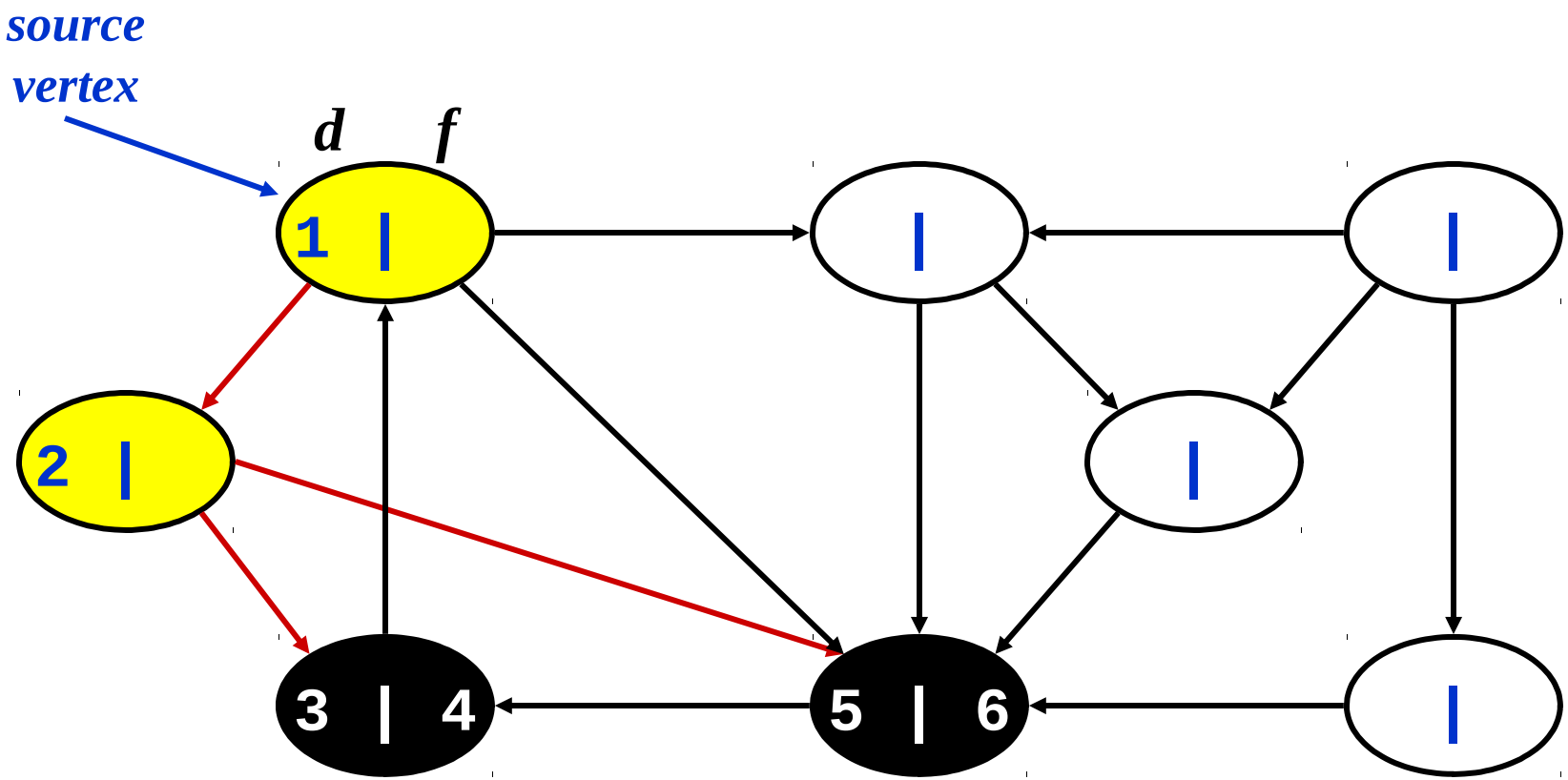
DFS Example



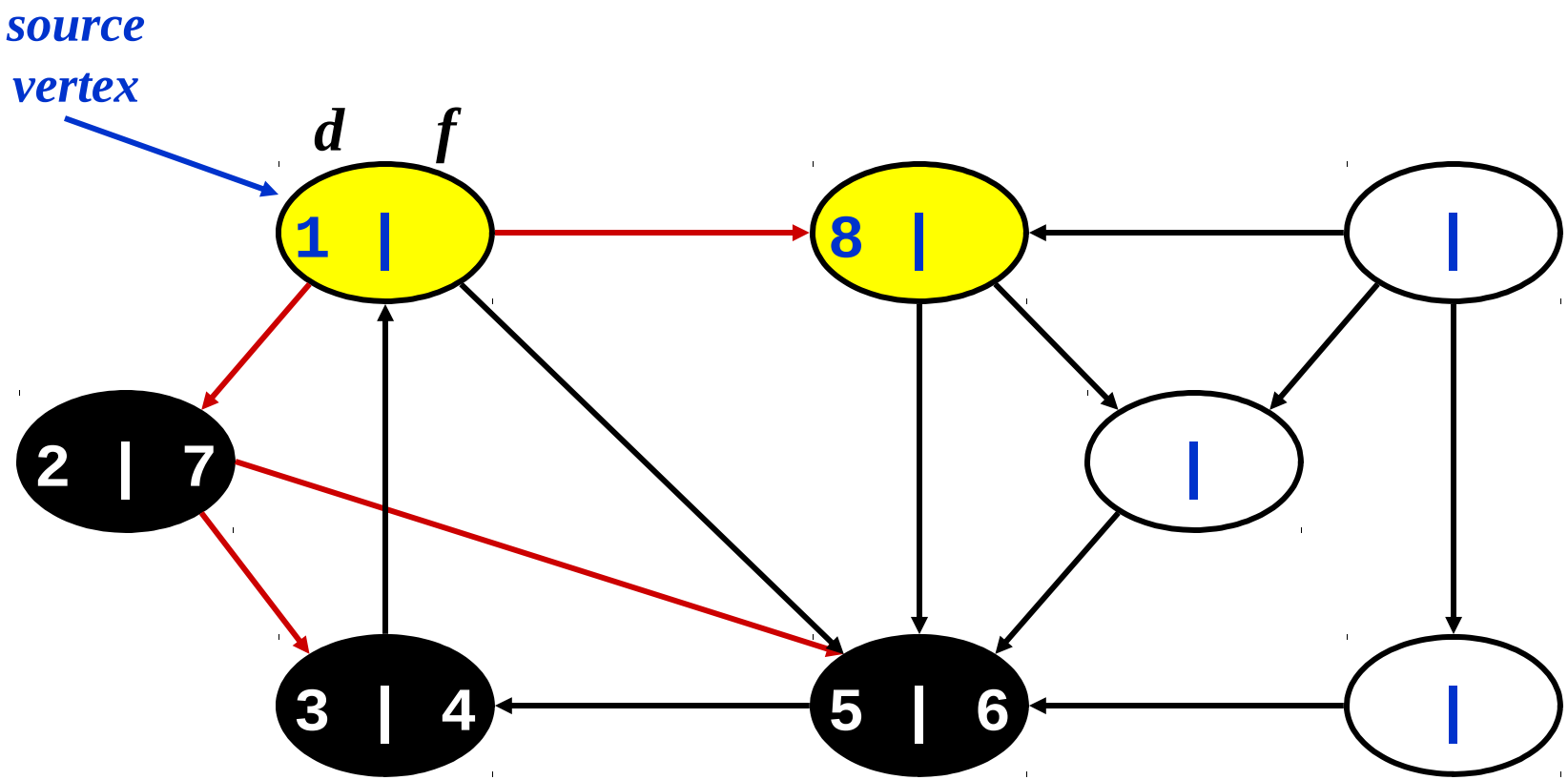
DFS Example



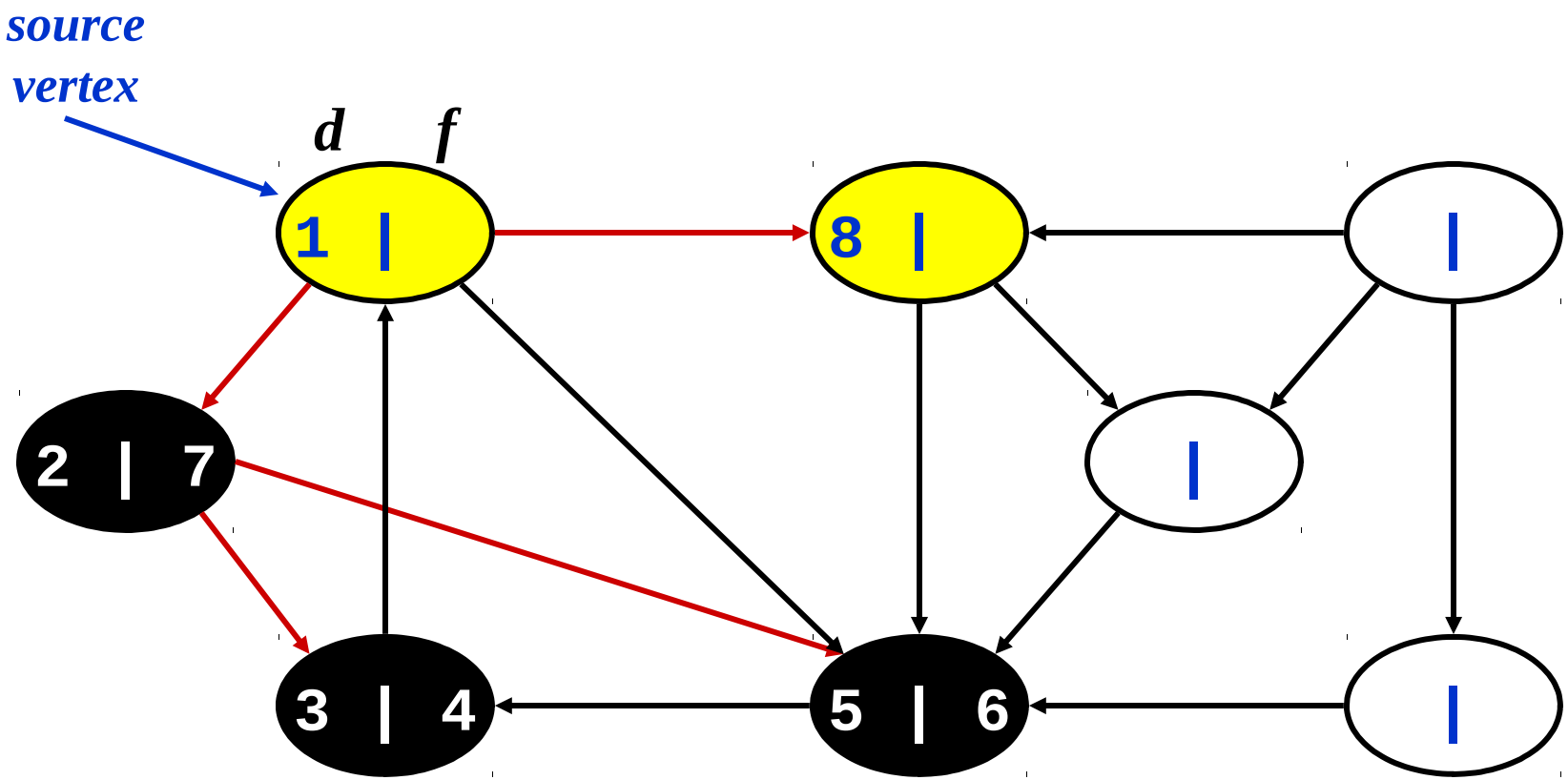
DFS Example



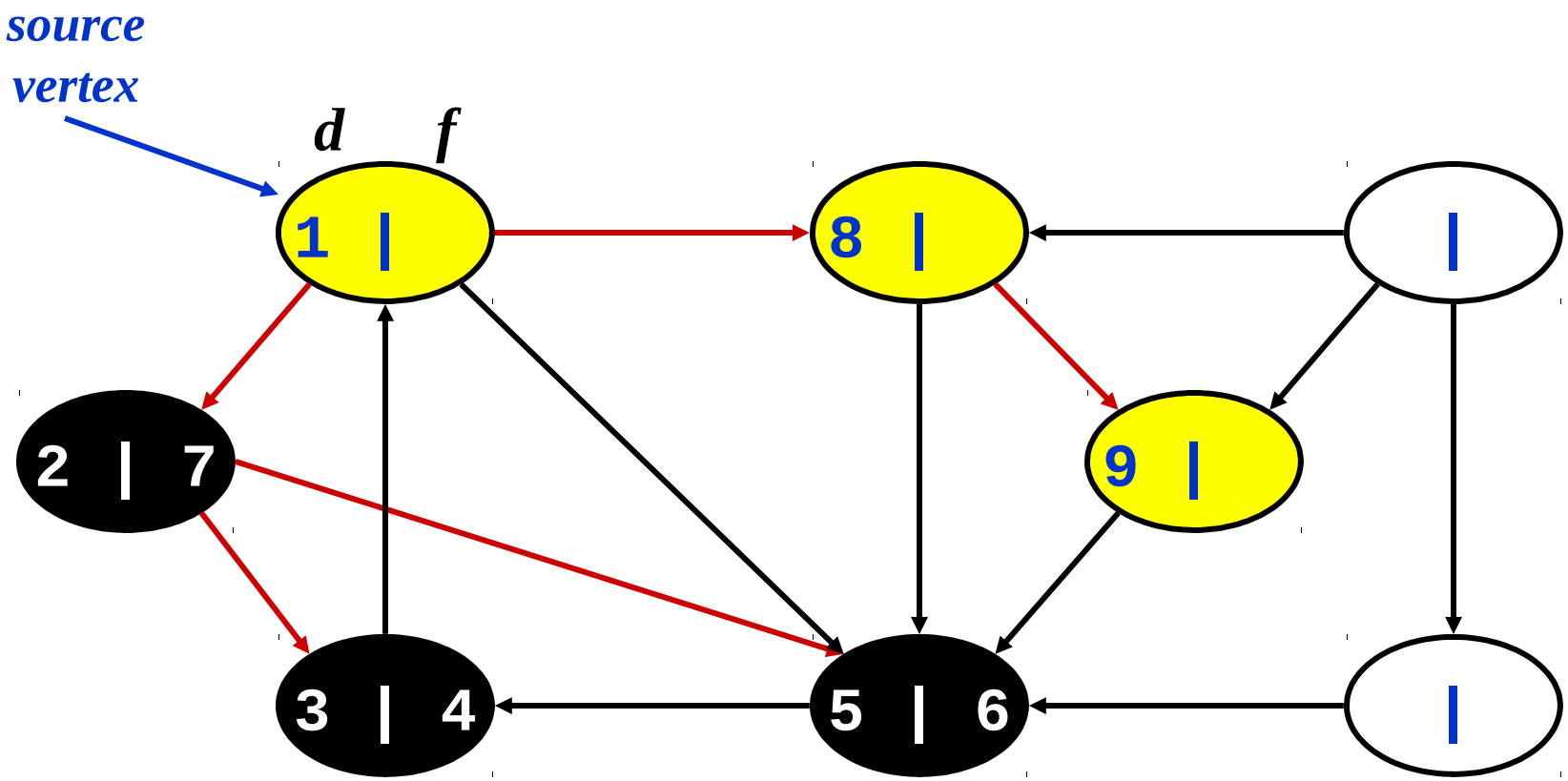
DFS Example



DFS Example

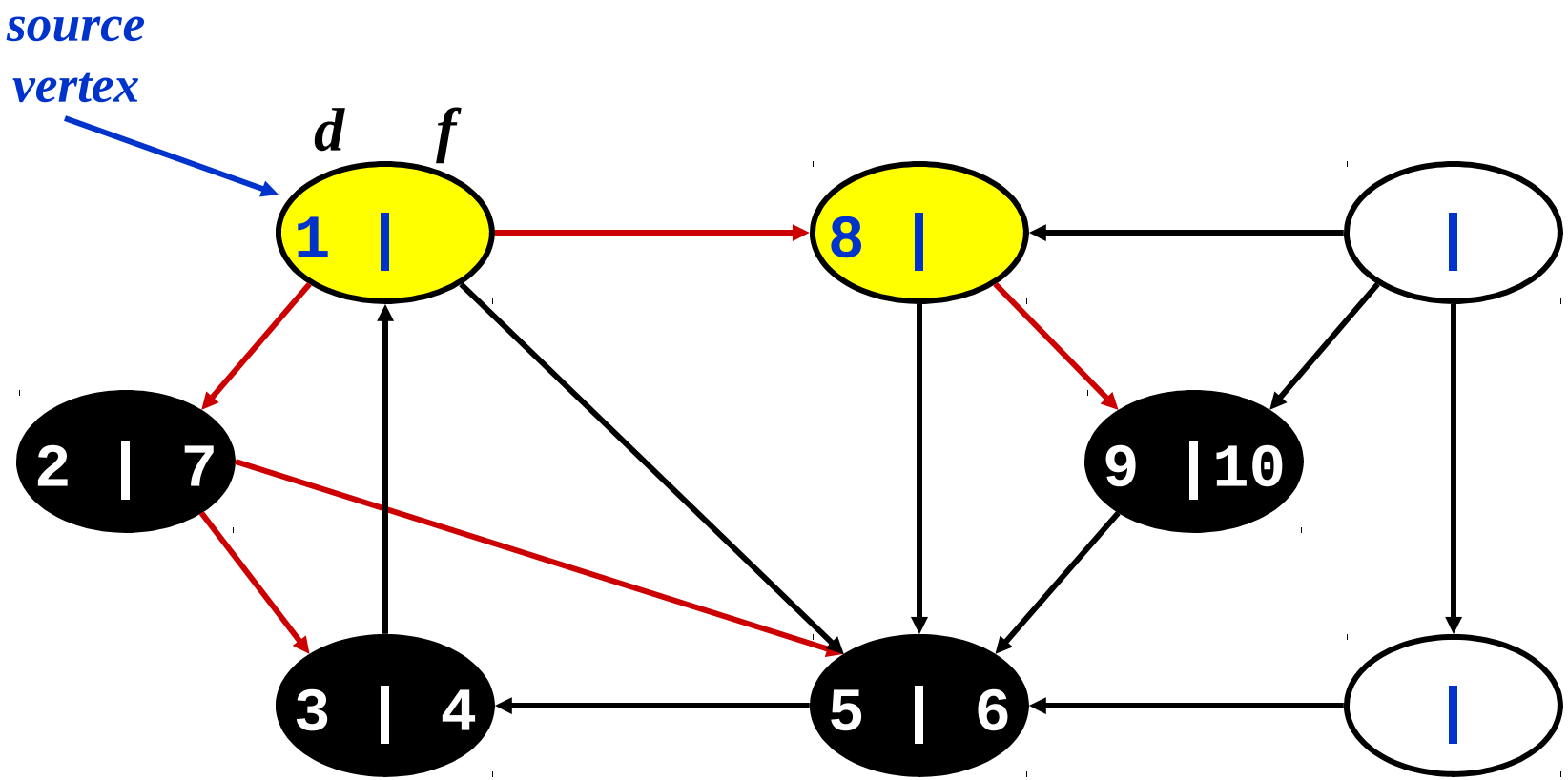


DFS Example

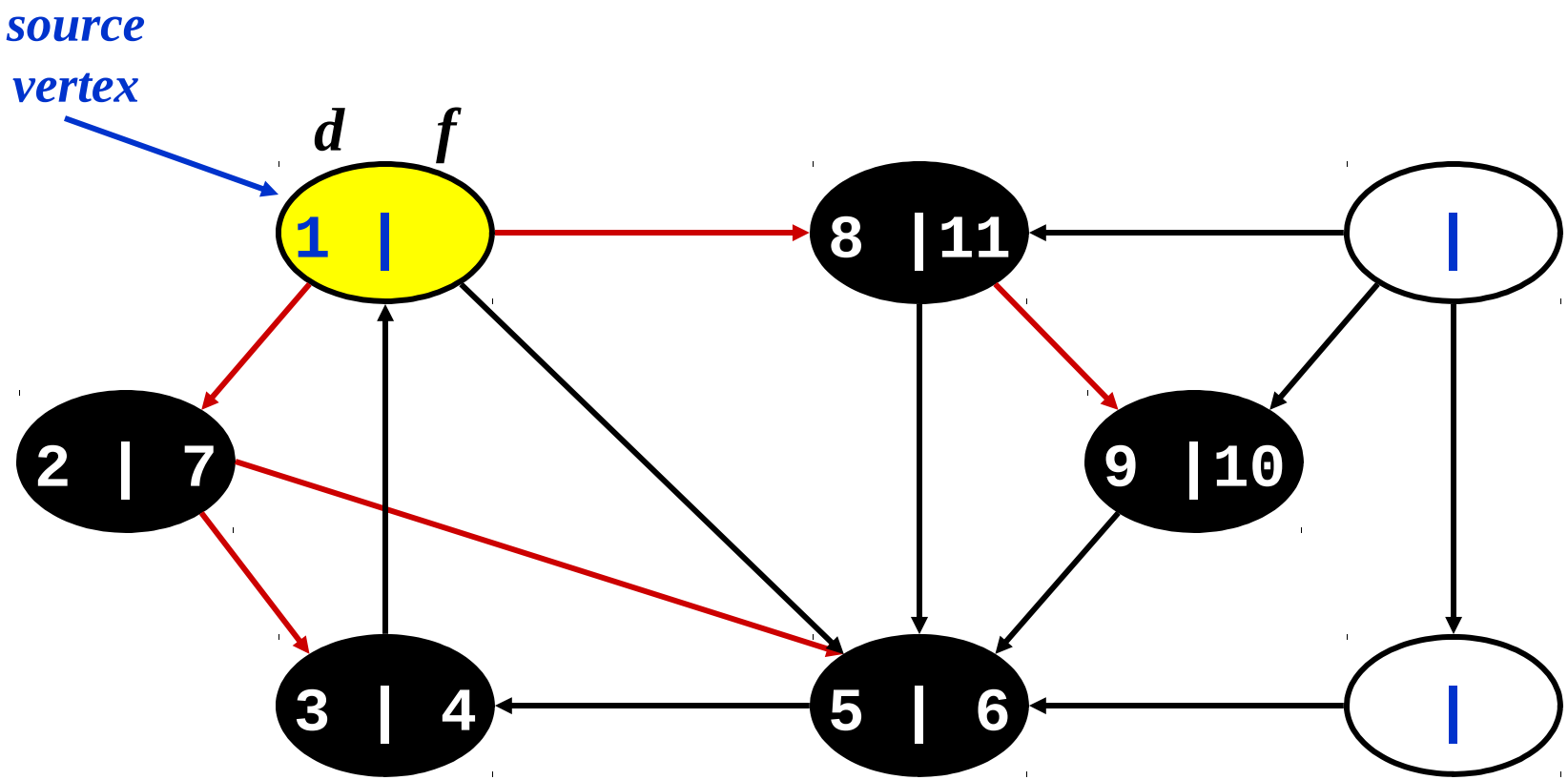


What is the structure of the yellow vertices?
What do they represent?

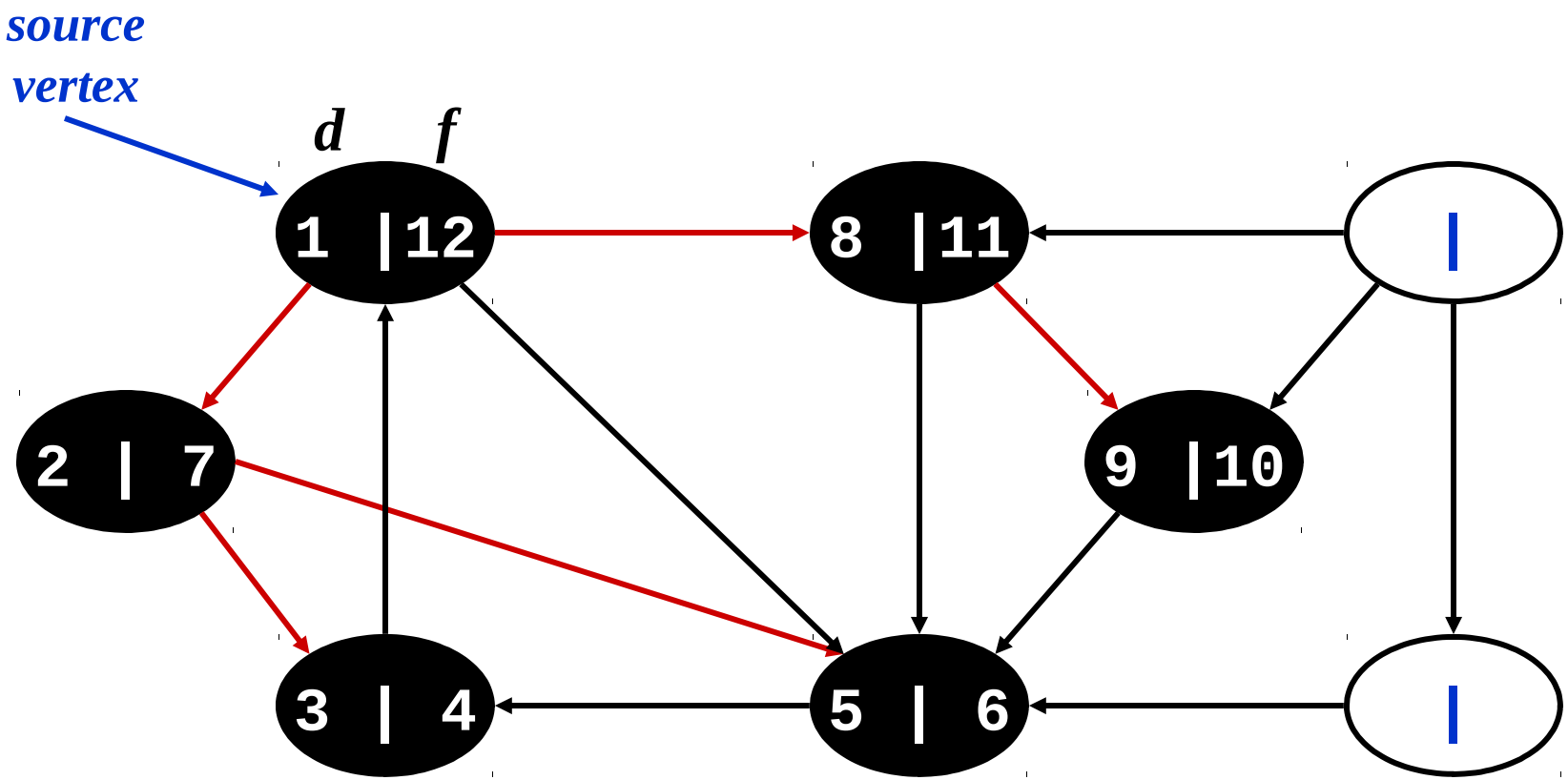
DFS Example



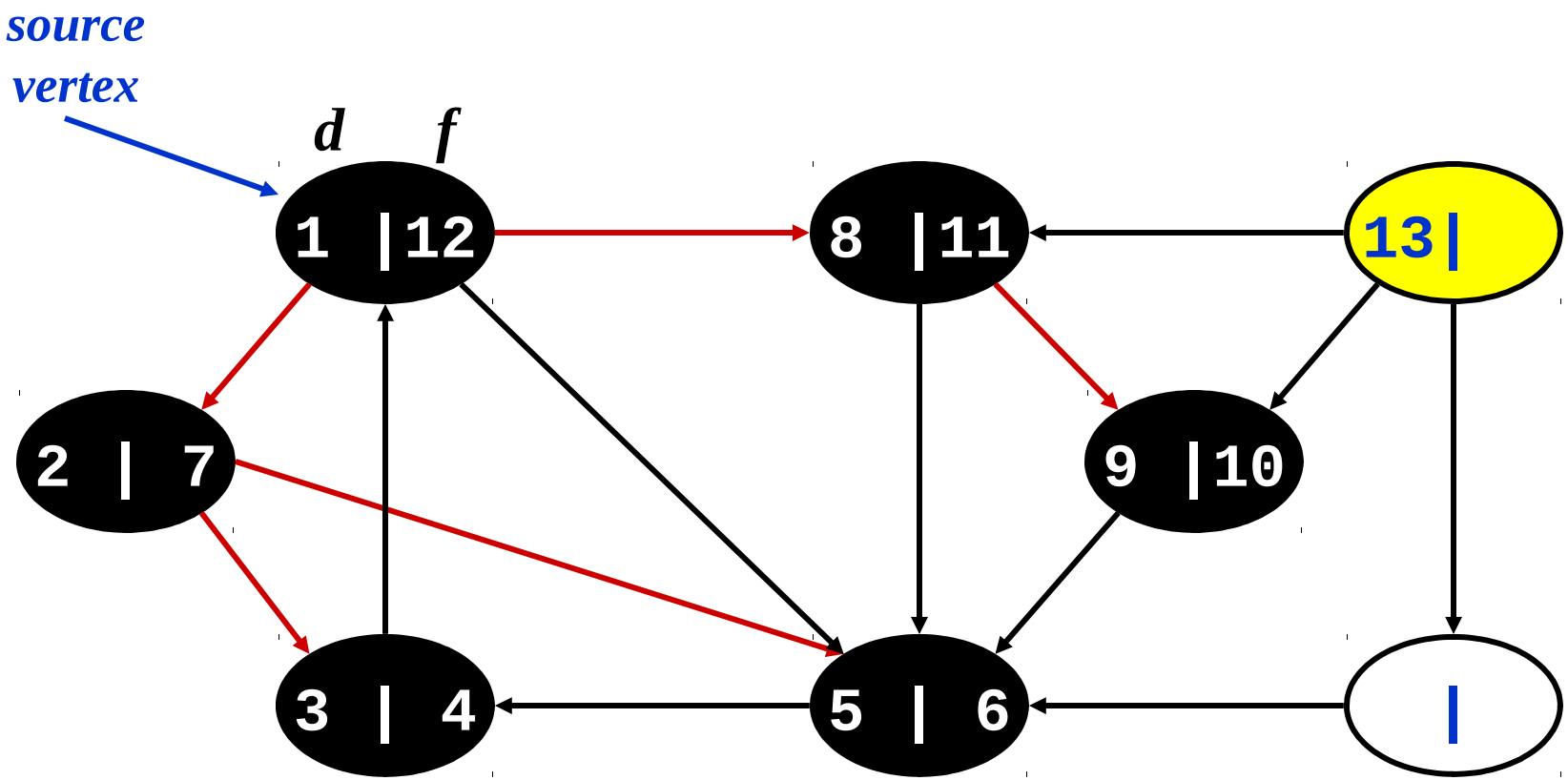
DFS Example



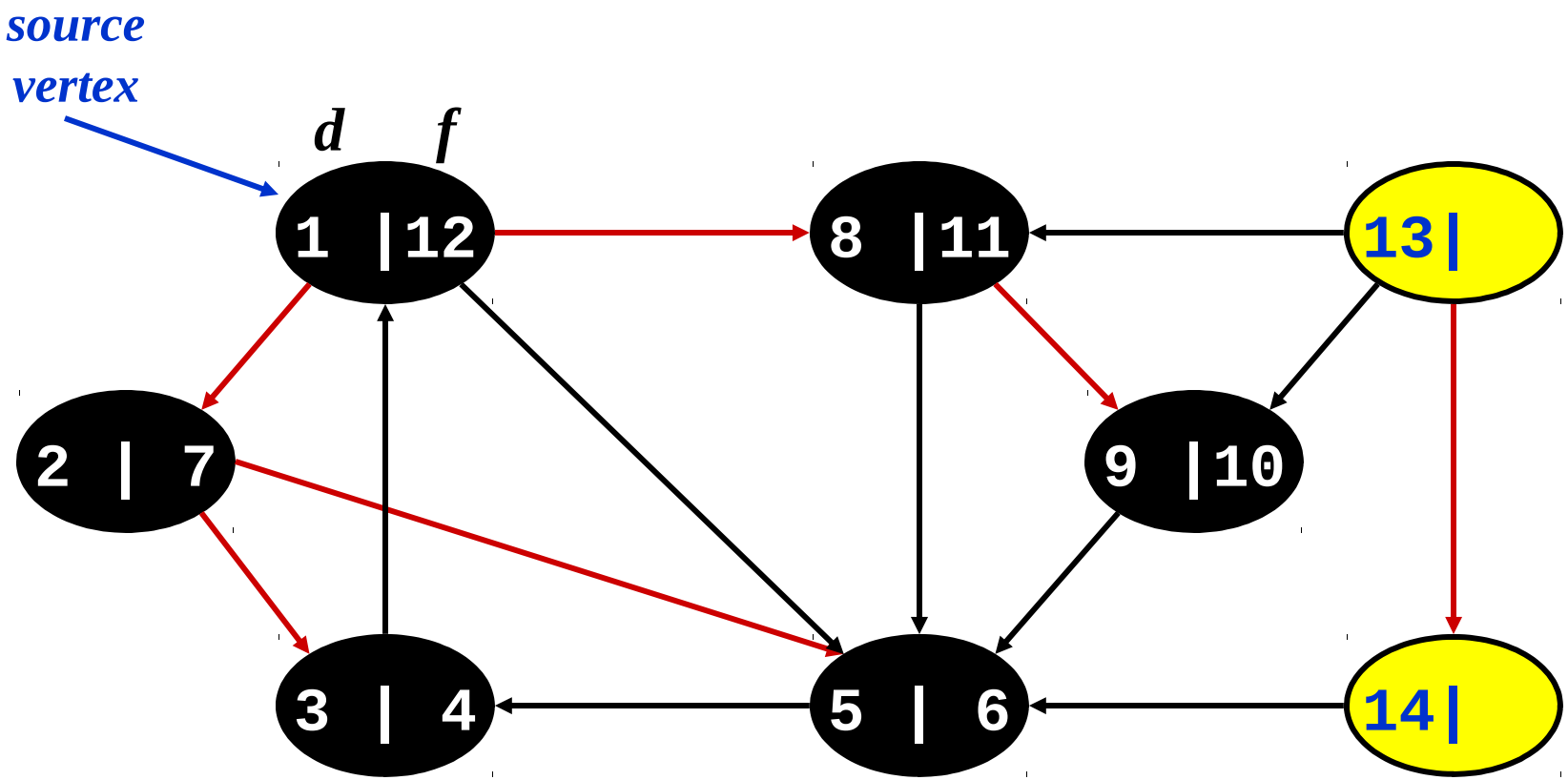
DFS Example



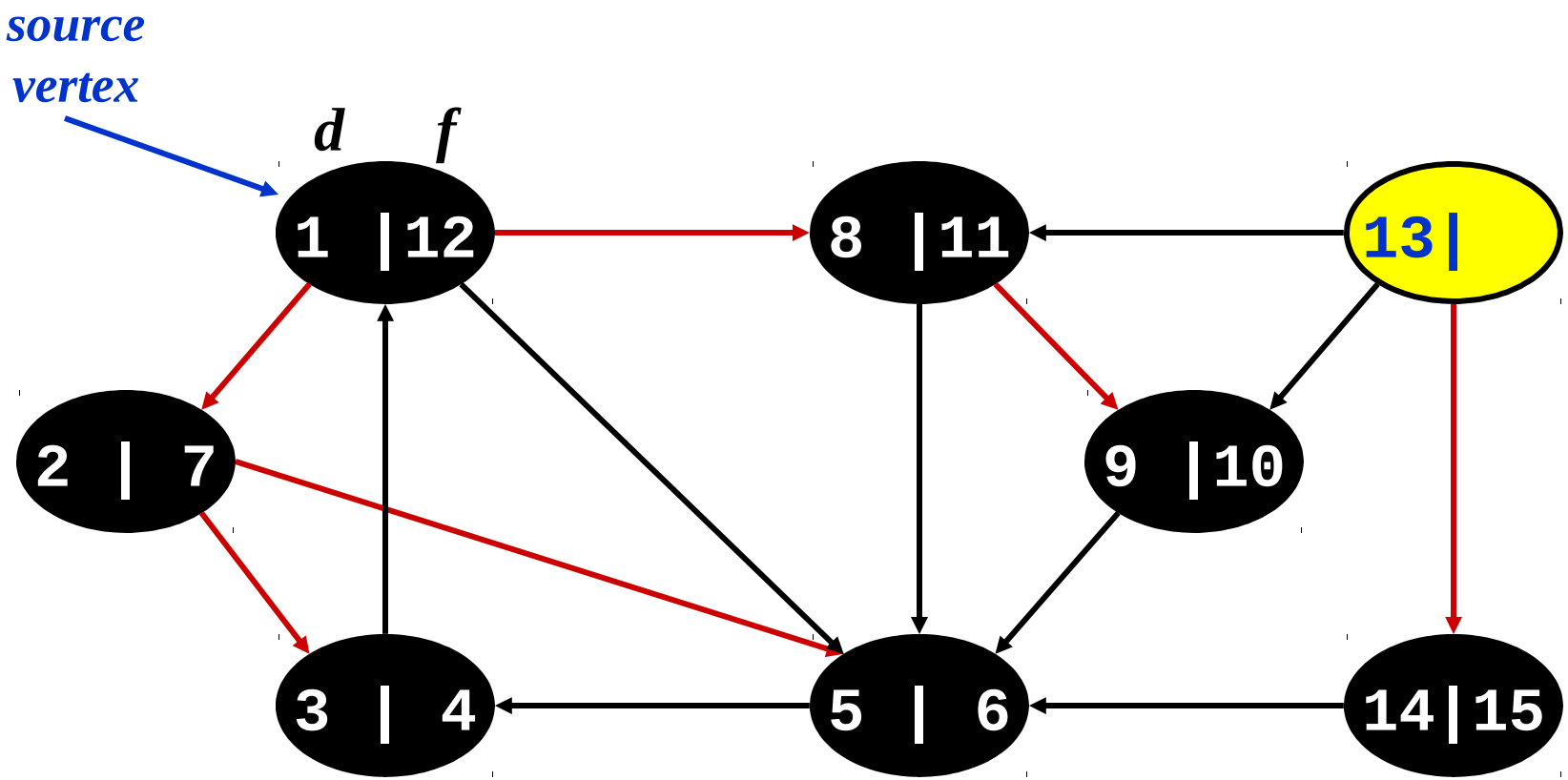
DFS Example



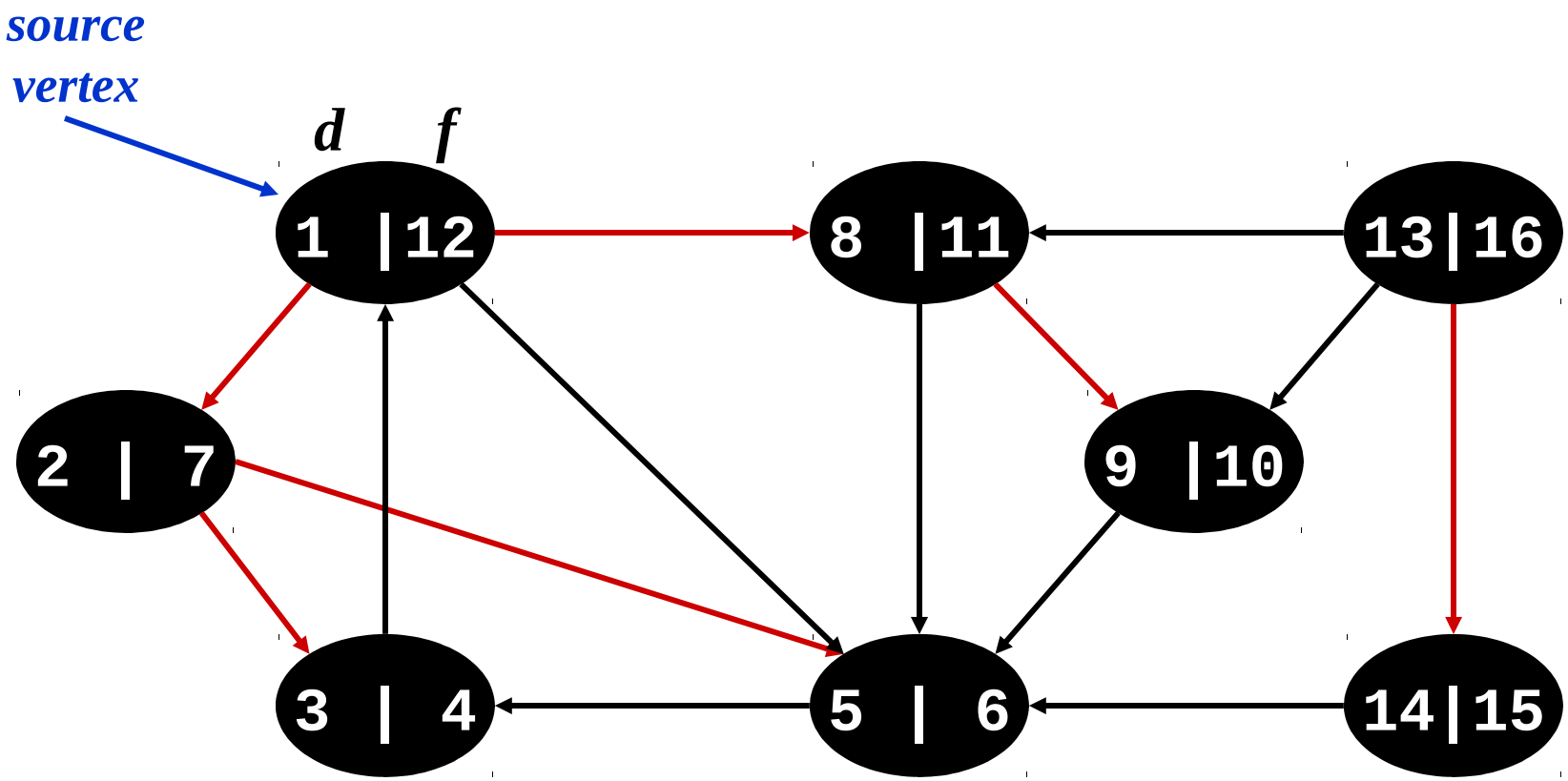
DFS Example



DFS Example



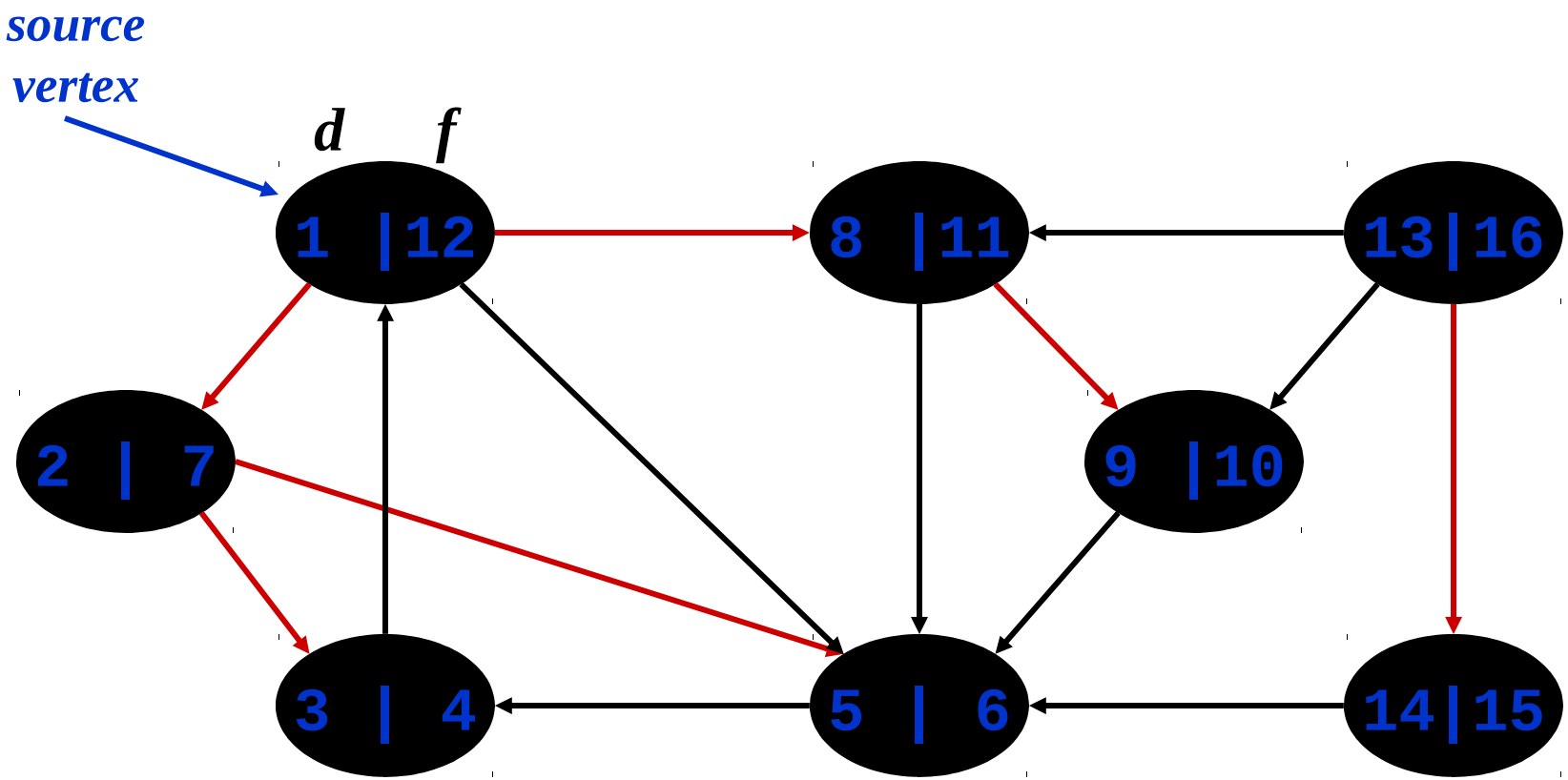
DFS Example



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

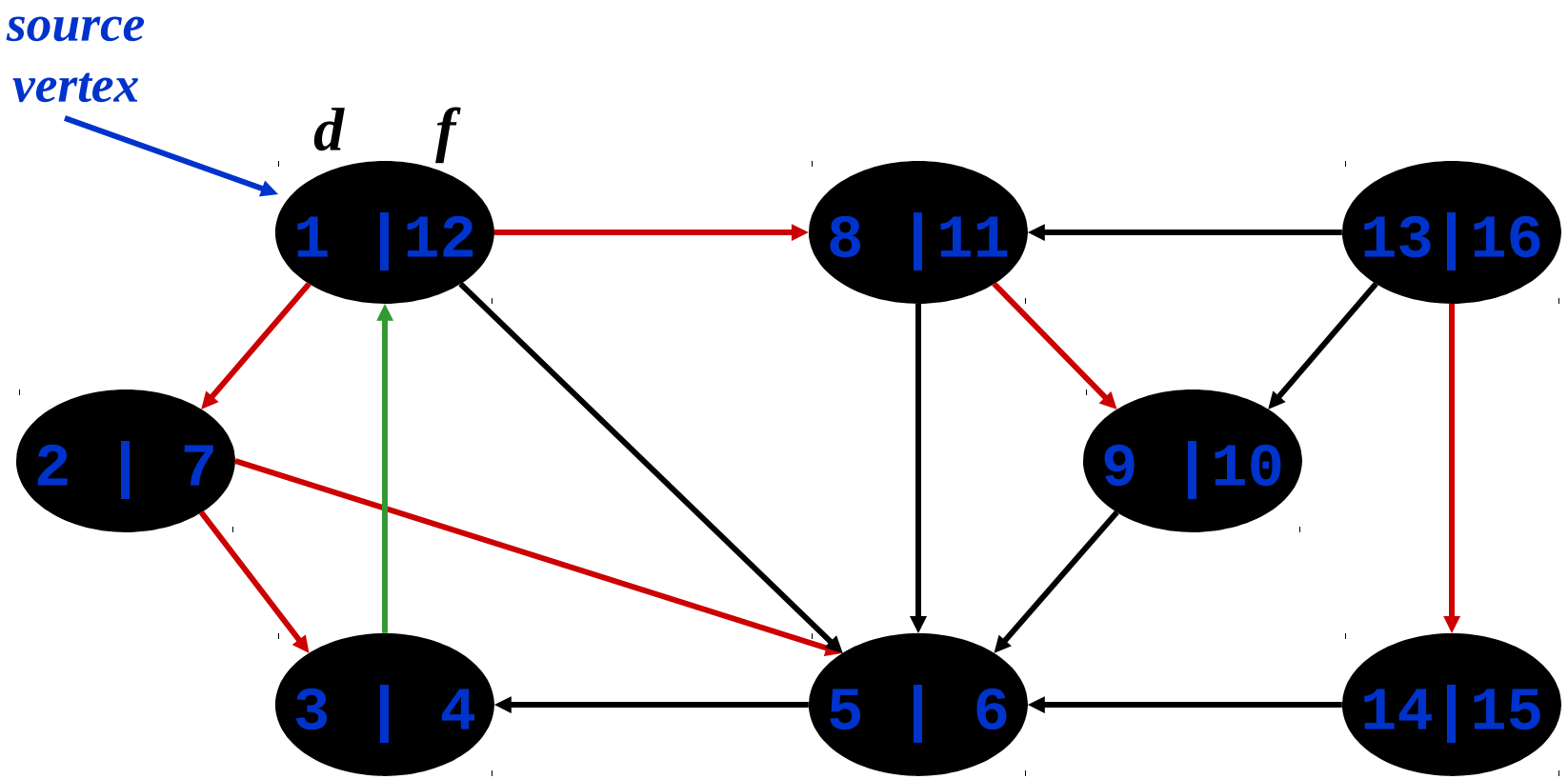


Tree edges

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 yellow vertex (yellow to yellow)

DFS Example

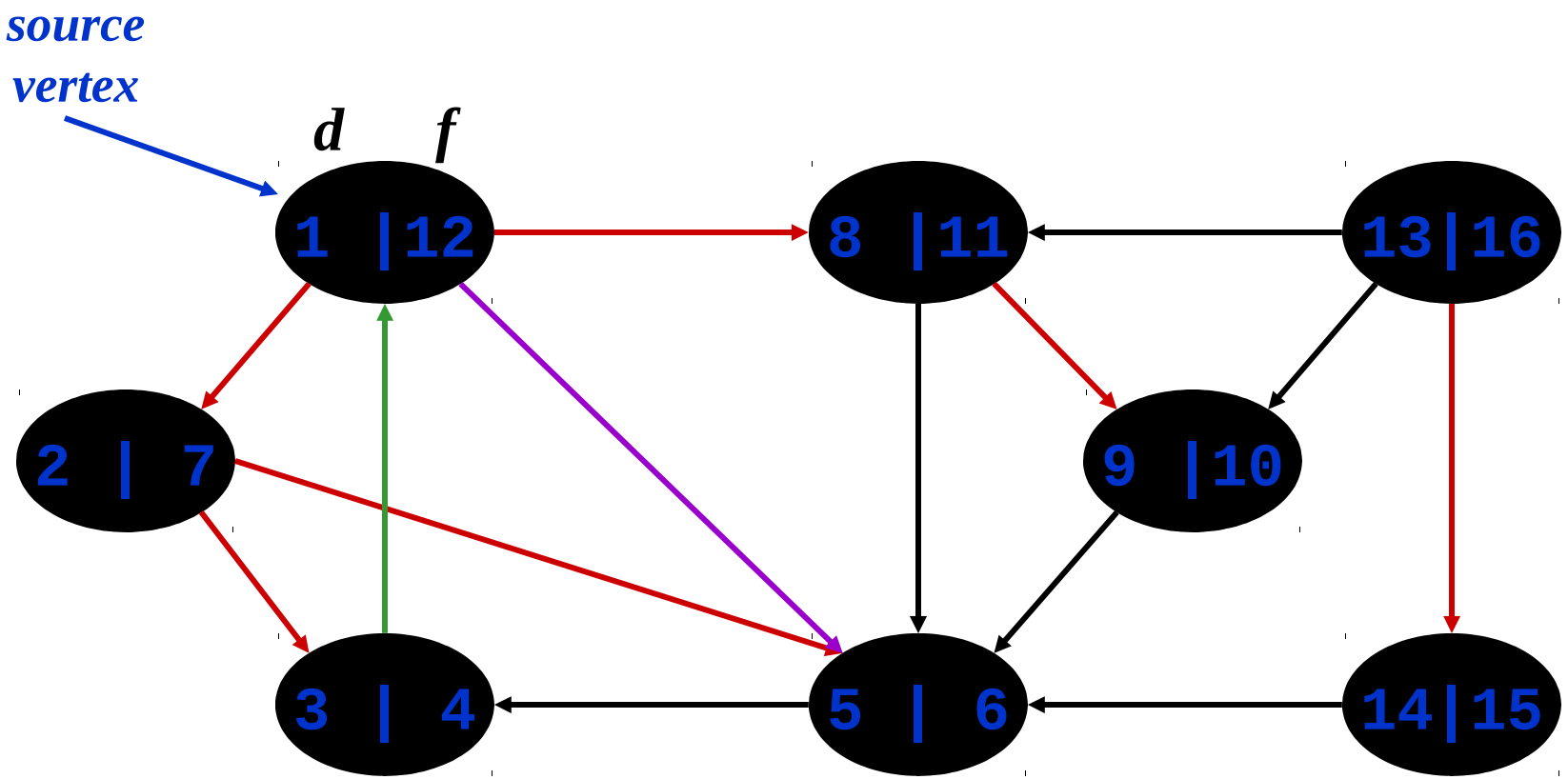


Tree edges *Back edges*

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 yellow node to black node

DFS Example

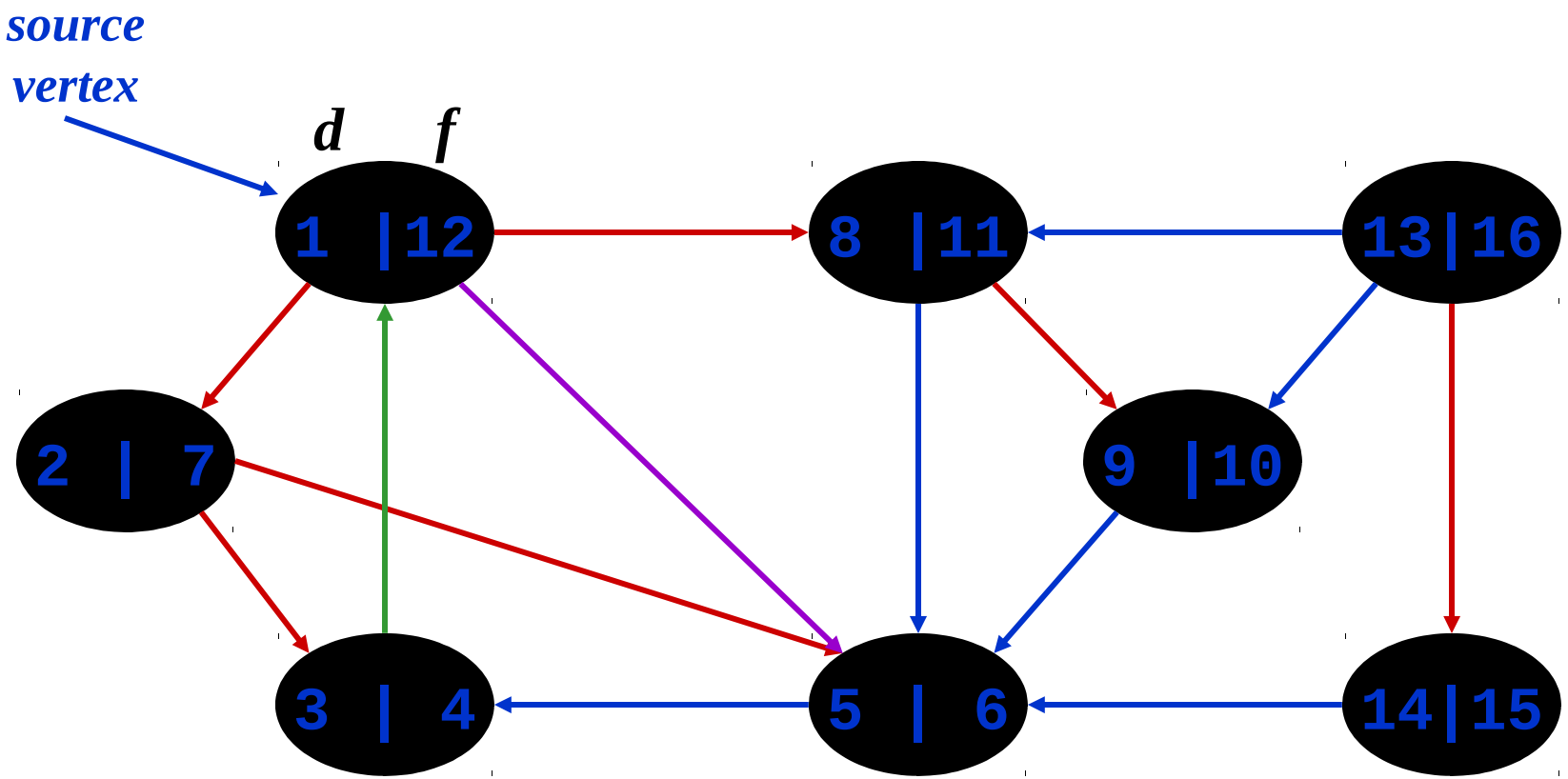


Tree edges *Back edges* *Forward edges*

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 yellow 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

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);
   }
}
```

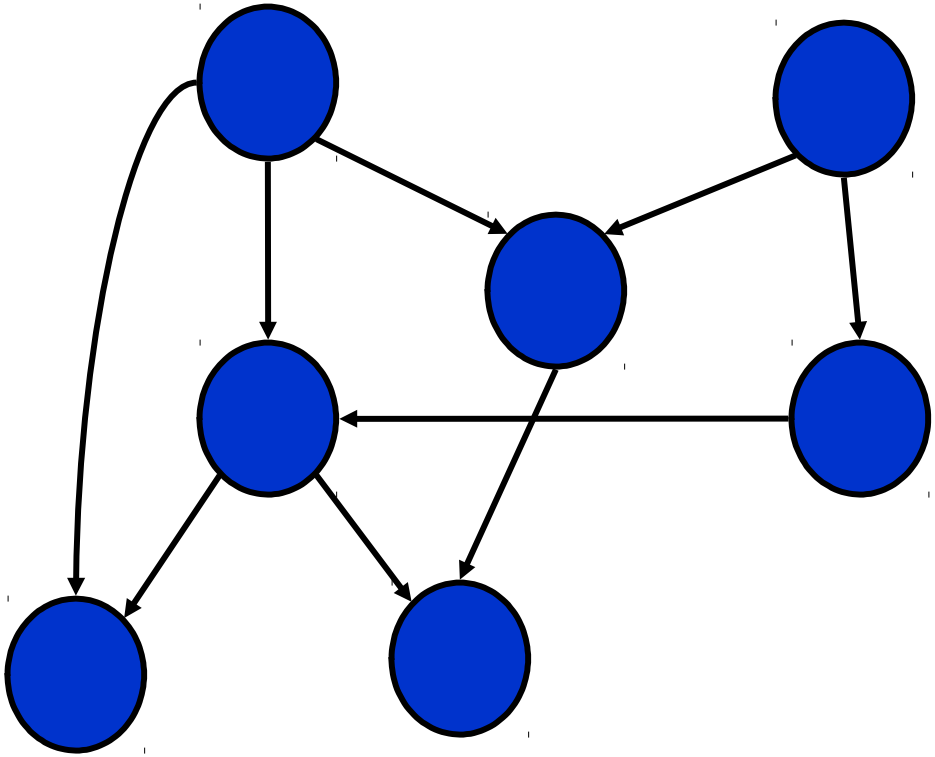
```
DFS_Visit(u)
{
   u->color = GREY;
   time = time+1;
   u->d = time;
   for each v ∈ u->Adj[]
   {
      if (v->color == WHITE)
         DFS_Visit(v);
   }
   u->color = BLACK;
   time = time+1;
   u->f = time;
}
```

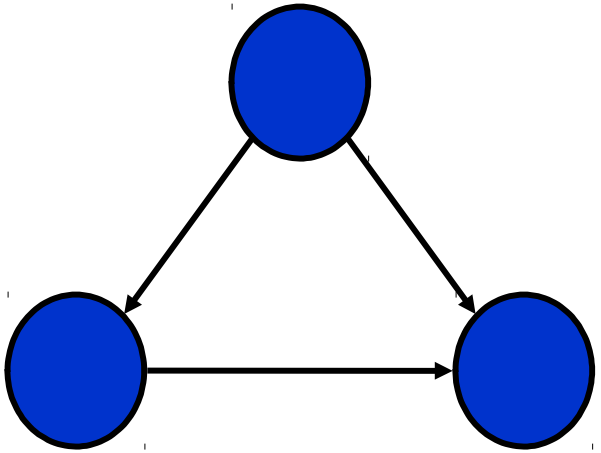
DFS And Cycles

- *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

Directed Acyclic Graph

- A *directed acyclic graph* or *DAG* is a directed graph with no directed cycles:





DFS and DAG

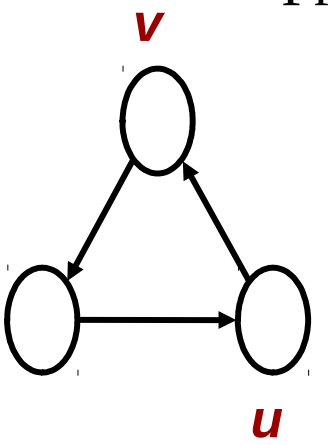
- Theorem: a directed graph G is acyclic if and only if a DFS of G yields no back edges:

  => if G is acyclic, will be no back edges

  - Trivial: a back edge implies a cycle

  <= if no back edges, G is acyclic

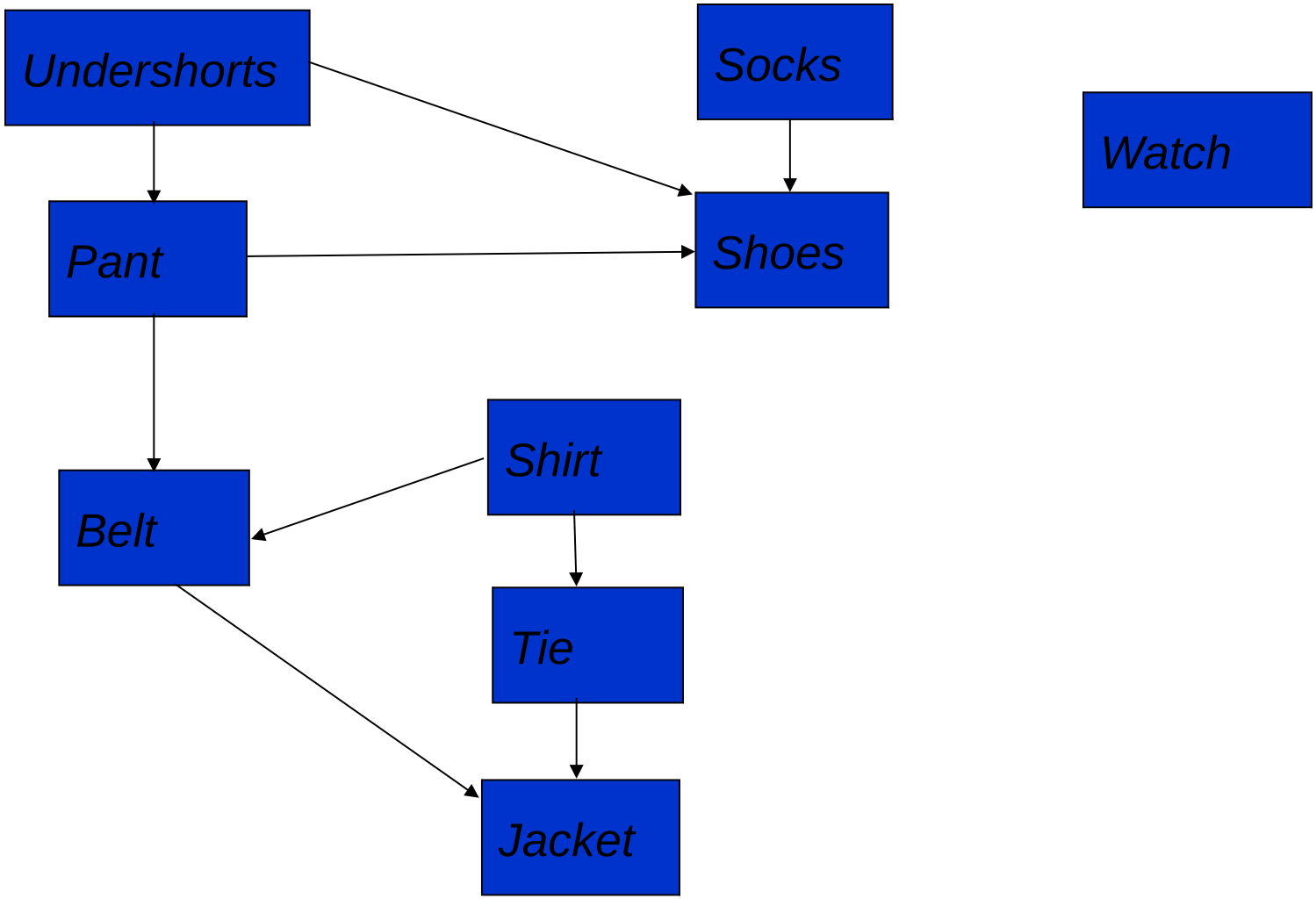
  - Proof by contradiction: G has a cycle ⇒ ∃ a back edge
    - Let *v* be the vertex on the cycle first discovered, and *u* be the predecessor of *v* on the cycle
    - When *v* discovered, whole cycle is white
    - Must visit everything reachable from *v* before returning from DFS-Visit()
    - So path from u (gray)→v (gray), thus (*u*, *v*) is a back edge



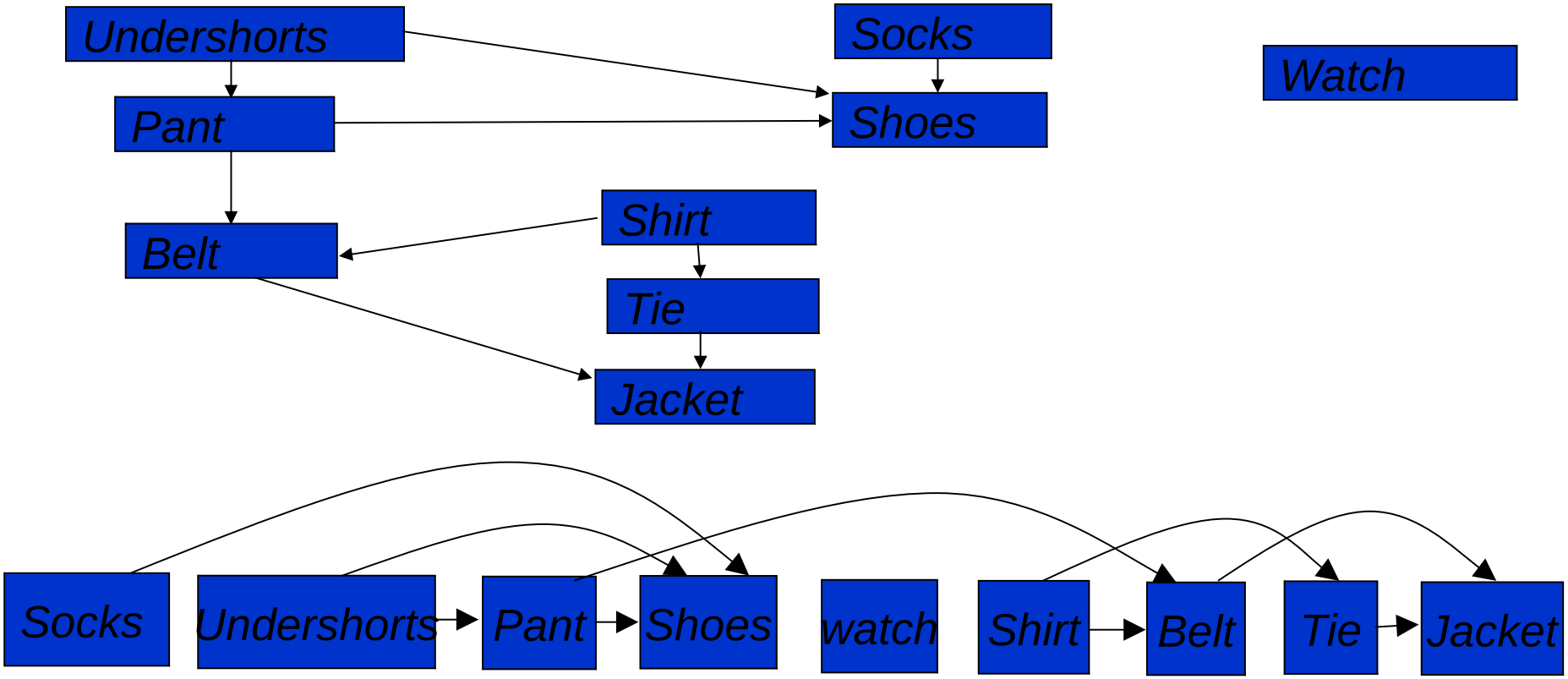
Topological Sort

- *Topological sort* of a DAG:
  - Linear ordering of all vertices in graph G such that vertex *u* comes before vertex *v* if edge $(u, v) \in$ G
- Real-world application:
  - ✓ Scheduling a dependent graph,
  - ✓ Find a feasible course plan for university studies

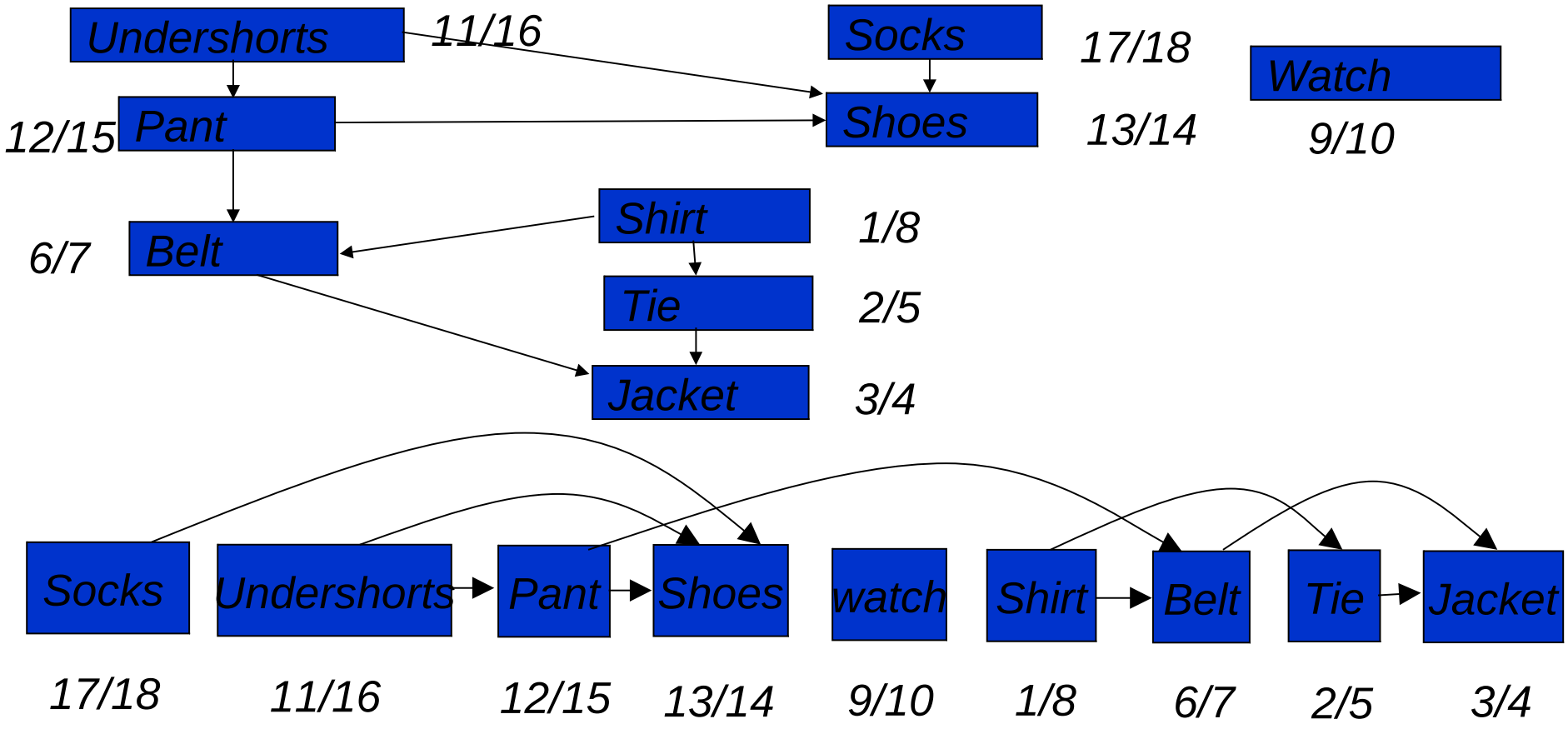
Example



Example (Cont.)



Algorithm


Undershorts
11/16
Socks
17/18
Watch
12/15
Pant
Shoes
13/14
9/10
Shirt
1/8
6/7
Belt
Tie
2/5
Jacket
3/4
Socks
Undershorts
Pant
Shoes
watch
Shirt
Belt
Tie
Jacket
17/18
11/16
12/15
13/14
9/10
1/8
6/7
2/5
3/4

Algorithm

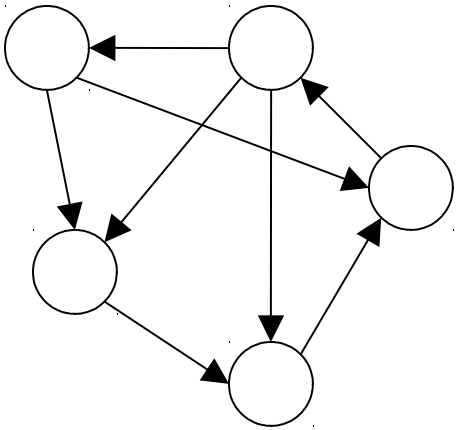
```
Topological-Sort()
{
  1. Call DFS to compute finish time
     f[v] for each vertex
  2. As each vertex is finished, insert
     it onto the front of a linked list
  3. Return the linked list of vertices
}
```

- Time: O(V+E)

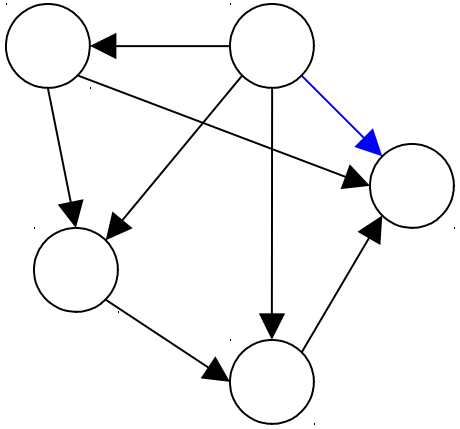
Strongly Connected Components

- Every pair of vertices are reachable from each other
- Graph *G* is ***strongly connected*** if, for every *u* and *v* in *V*, there is some path from *u* to *v* and some path from *v* to *u*.

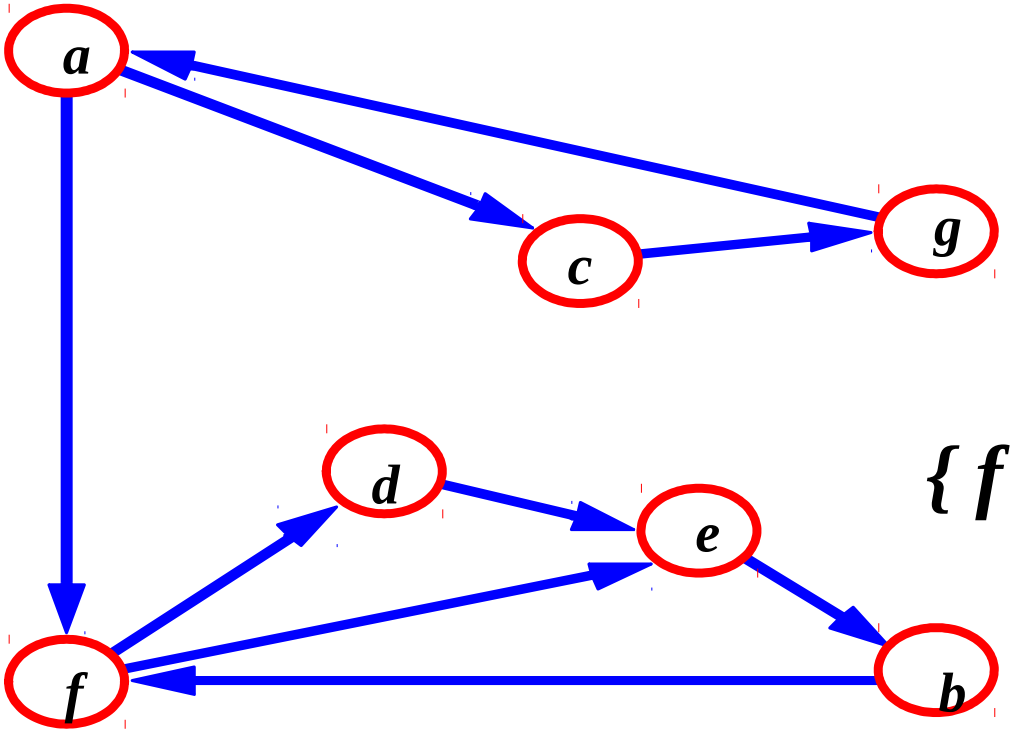
Strongly Connected



Not Strongly Connected



Example



$\{a, c, g\}$

$\{f, d, e, b\}$

Finding Strongly Connected Components

- Input: A directed graph G = (V,E)
- Output: a partition of V into disjoint sets so that each set defines a strongly connected component of G

Algorithm

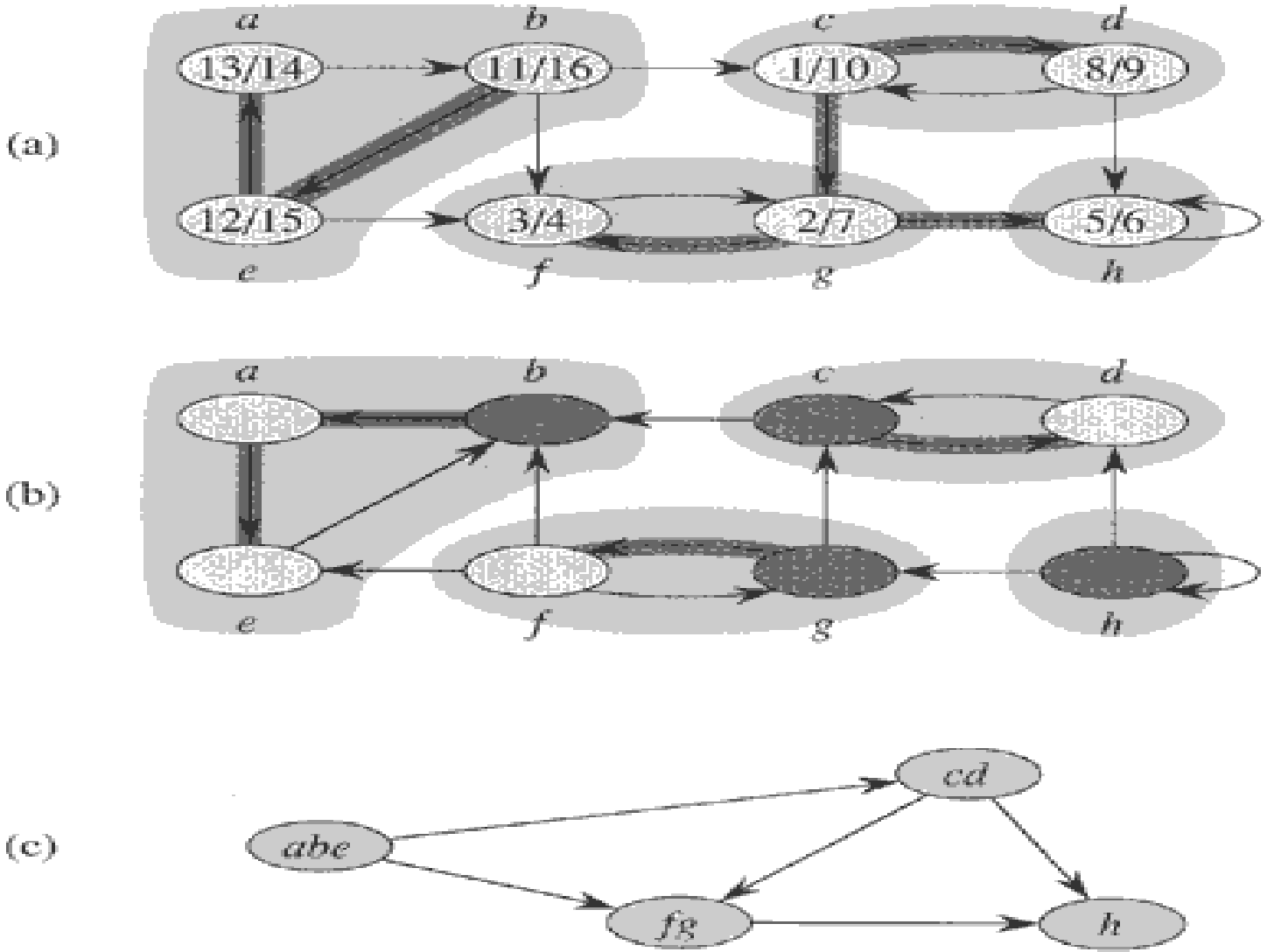
Strongly-Connected-Components(G)

1. call DFS(G) to compute finishing times f[u] for each vertex u.
   Cost: O(E+V)
2. compute G^T Cost: O(E+V)
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing f[u] Cost: O(E+V)
4. output the vertices of each tree in the depth-first forest of step 3 as a separate strongly connected component.

The graph G^T is the transpose of G, which is visualized by reversing the arrows on the digraph.

- Cost: O(E+V)

Example


(a)
a
b
c
d
13/14
11/16
1/10
8/9
12/15
3/4
2/7
5/6
e
f
g
h
(b)
a
b
c
d
e
f
g
h
(c)
cd
abe
fg
h