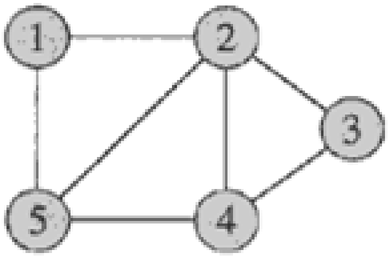
Graph Algorithms

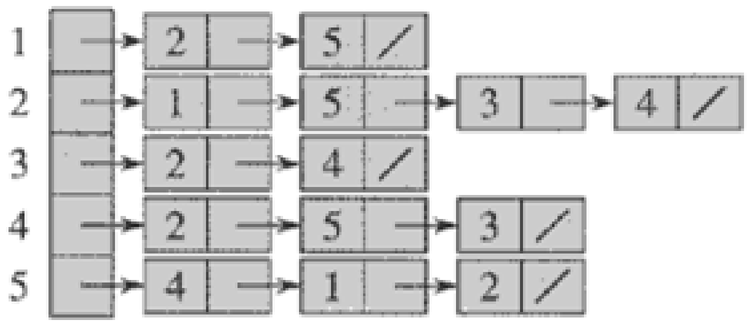
- **Representations of Graph**
- **Graph Searching**
  - **Breadth First Search**
  - **Depth First Search**
  - **Classification of Edges**
- **Directed Acyclic Graph**
- **Topological Sort**

Graphs

- A graph $G = (V, E)$
  - V = set of vertices, E = set of edges
  - *Dense* graph: $|E|$ close to $|V|^2$
  - *Sparse* graph: $|E|$ much less than $|V|^2$
  - *Undirected graph:*
    - Edge (u,v) = edge (v,u) and No self-loops
  - *Directed* graph:
    - Edge (u,v) goes from vertex u to vertex v, notated $u \rightarrow v$
  - A *weighted graph* associates weights with either the edges or the vertices

Representations: Undirected Graphs

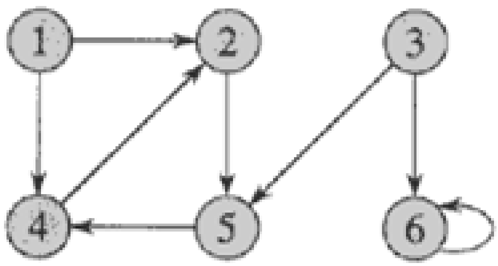


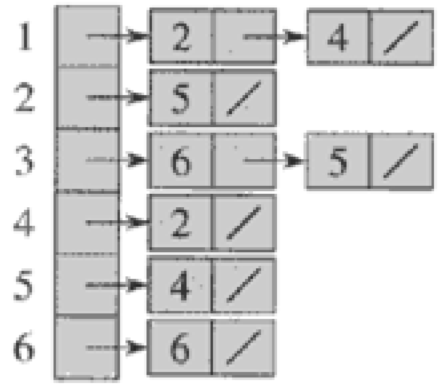


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	Adjacency List	*Adjacency Matrix*
Space complexity:	$\theta(V+E)$	$\theta(V^2)$
Time to find all neighbours of vertex u:	$\theta(\text{degree}(u))$	$\theta(V)$
Time to determine if $(u,v) \in E$:	$\theta(\text{degree}(u))$	$\theta(1)$

Representations: Directed Graphs





	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

	Adjacency List	Adjacency Matrix
Space complexity:	$\theta(V+E)$	$\theta(V^2)$
Time to find all neighbours of vertex u:	$\theta(\text{degree}(u))$	$\theta(V)$
Time to determine if $(u,v) \in E$:	$\theta(\text{degree}(u))$	$\theta(1)$

Graph Searching

- Given: a graph G = (V, E), directed or undirected
- Task: methodically explore every vertex and every edge
- build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - 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 discovered i.e. explored vertices across the *breadth* of the frontier
  - Discovers all vertices at distance k from s before discovering any vertices at distance k+1
- 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

- To keep track progress BFS associate vertex “colors” to guide the algorithm
  - White vertices→ have not been discovered
    - All vertices start out white
  - Grey vertices → discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices→ discovered and fully explored
    - They are adjacent only to black and gray vertices
- All vertices start out white and may later become gray and then black
- Explore vertices by scanning adjacency list of grey vertices

Breadth-First Search

- Completely explore the vertices in order of their distance from u
- implemented using a queue

BFS Procedure

BFS search procedure assumes that input graph G(V,E) represented using adjacency List

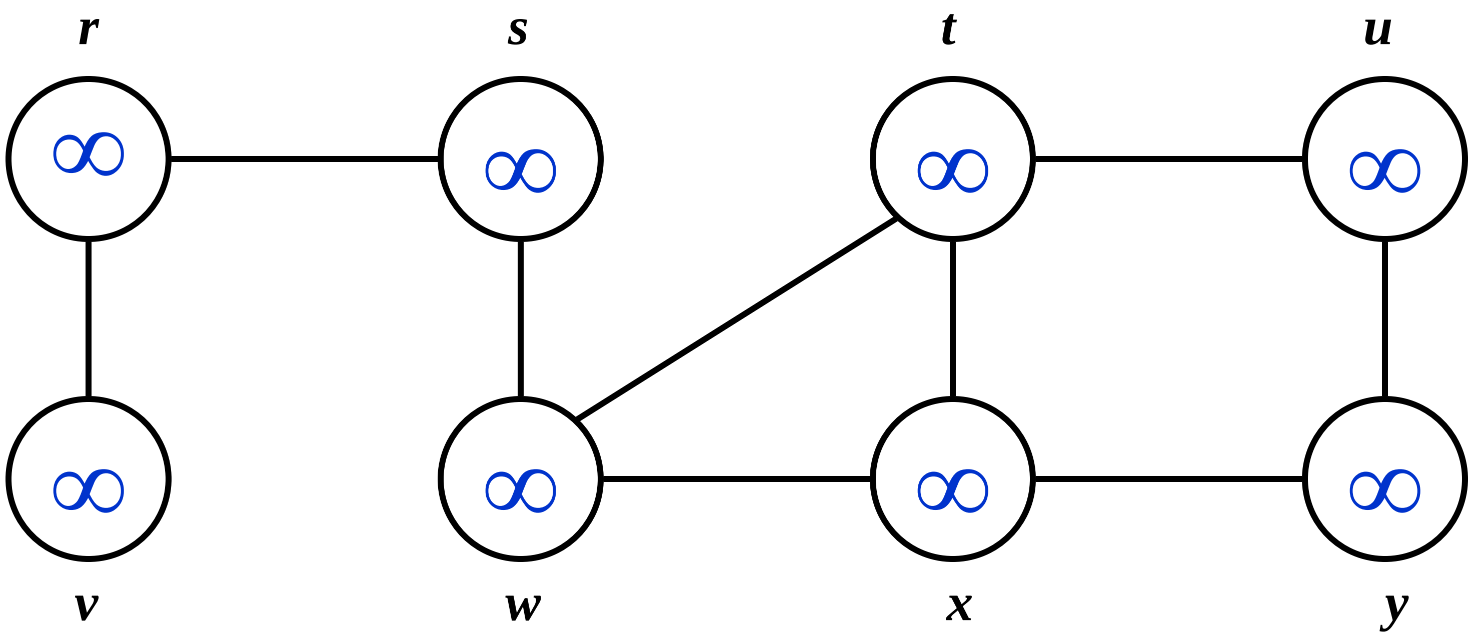
```
BFS(G, s)
for each vertex u ∈ V[G] − {s}
      do color[u] ← WHITE          # store color of each vertex u
         d[u] ← ∞                  # Store distance from s to u computed by algorithm
         π[u] ← NIL                # Store predecessor of u
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← Ø
ENQUEUE(Q, s)
while Q ≠ Ø
      do u ← DEQUEUE(Q)
         for each v ∈ Adj[u]
             do if color[v] = WHITE
                   then color[v] ← GRAY
                        d[v] ← d[u] + 1
                        π[v] ← u
                        ENQUEUE(Q, v)
         color[u] ← BLACK
```

Each vertex is enqueued at most once $\rightarrow O(V)$

Each entry in the adjacency lists is scanned at most once $\rightarrow$ O(E)

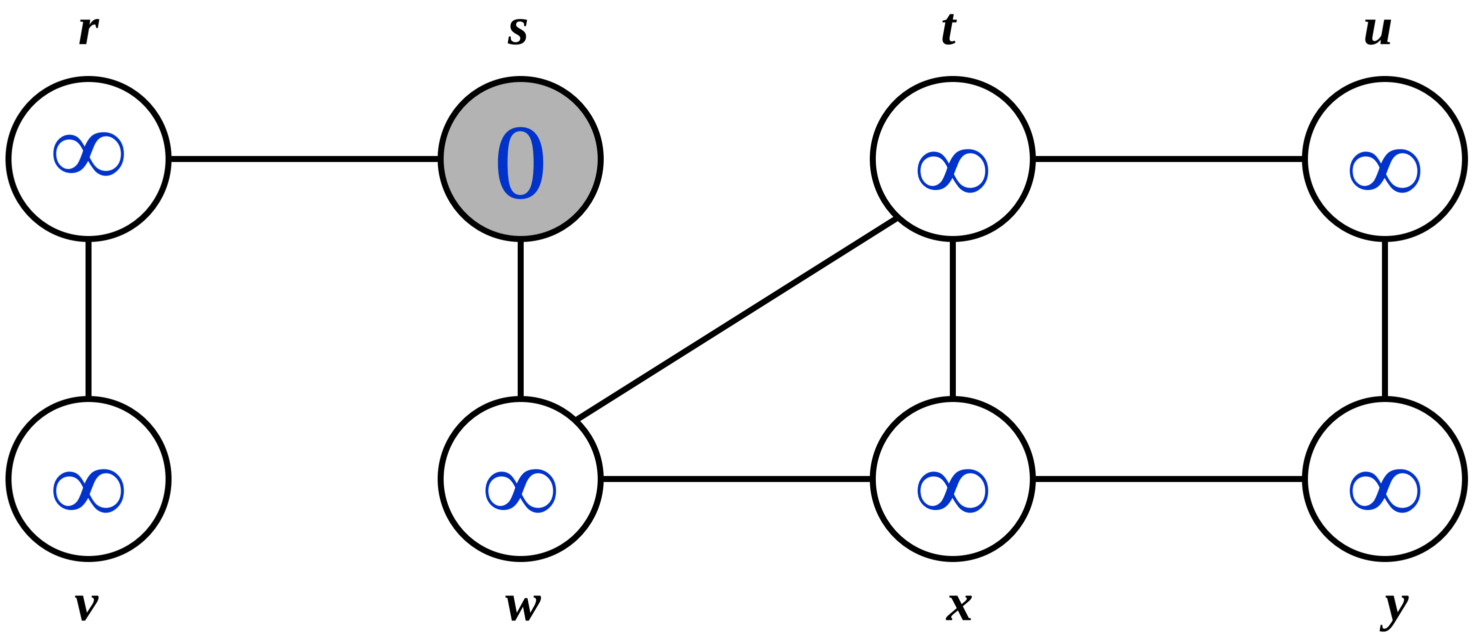
Thus run time is $O(V + E)$.

BFS: Example



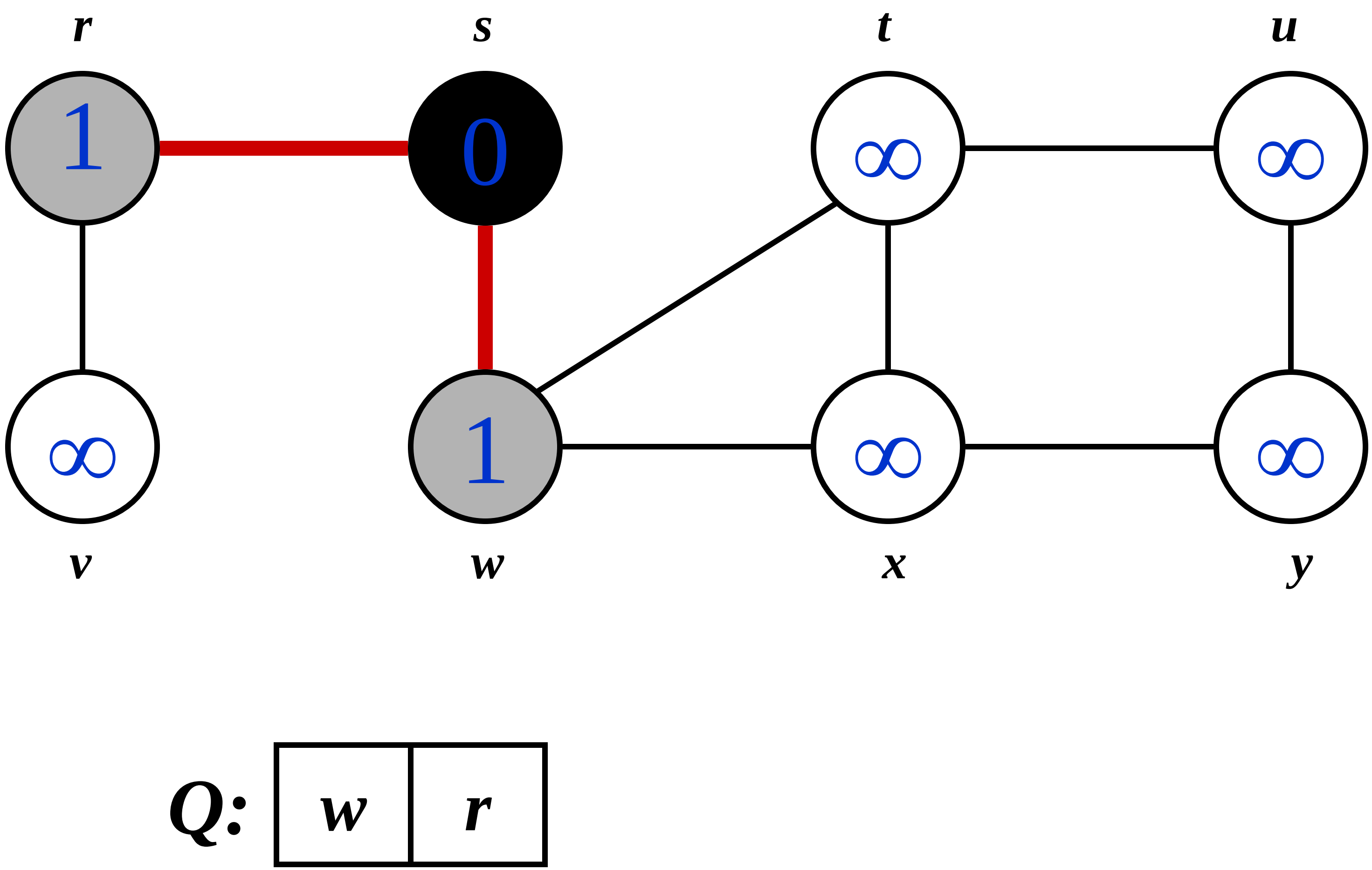
BFS search procedure assumes that input graph G(V,E) represented using adjacency List

Breadth-First Search: Example

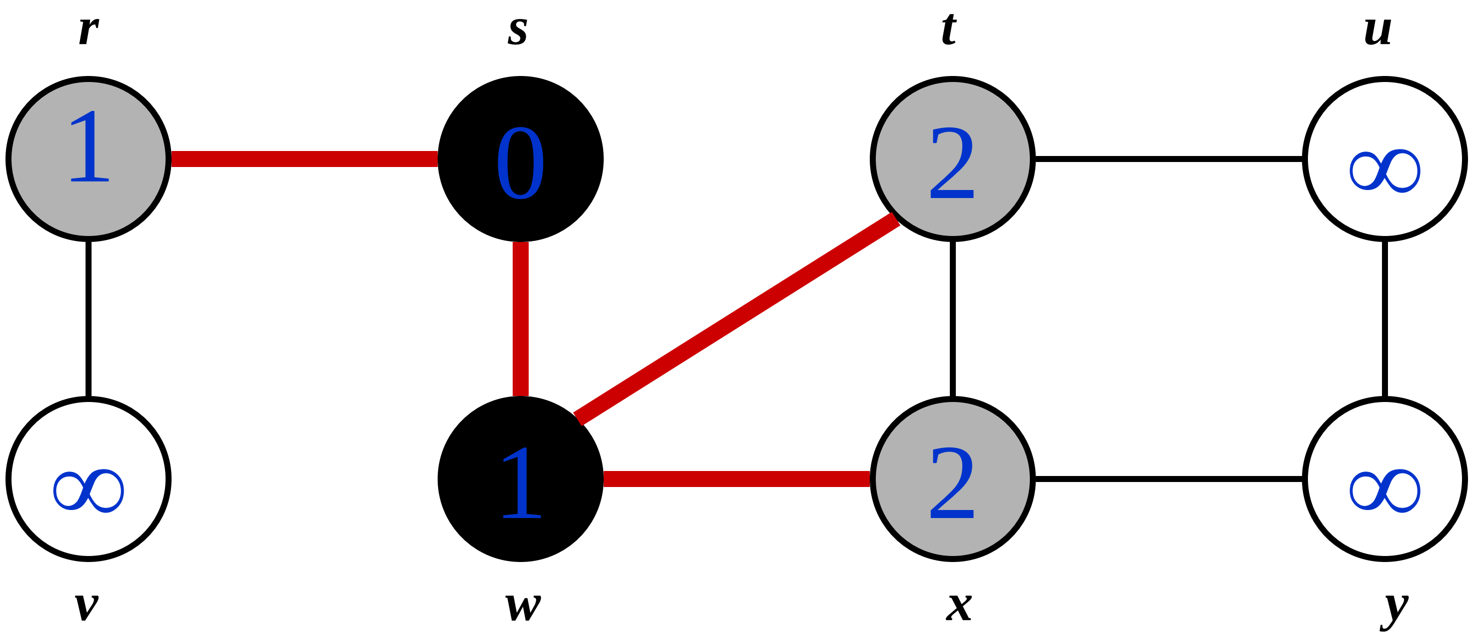


Q: *s*

Breadth-First Search: Example



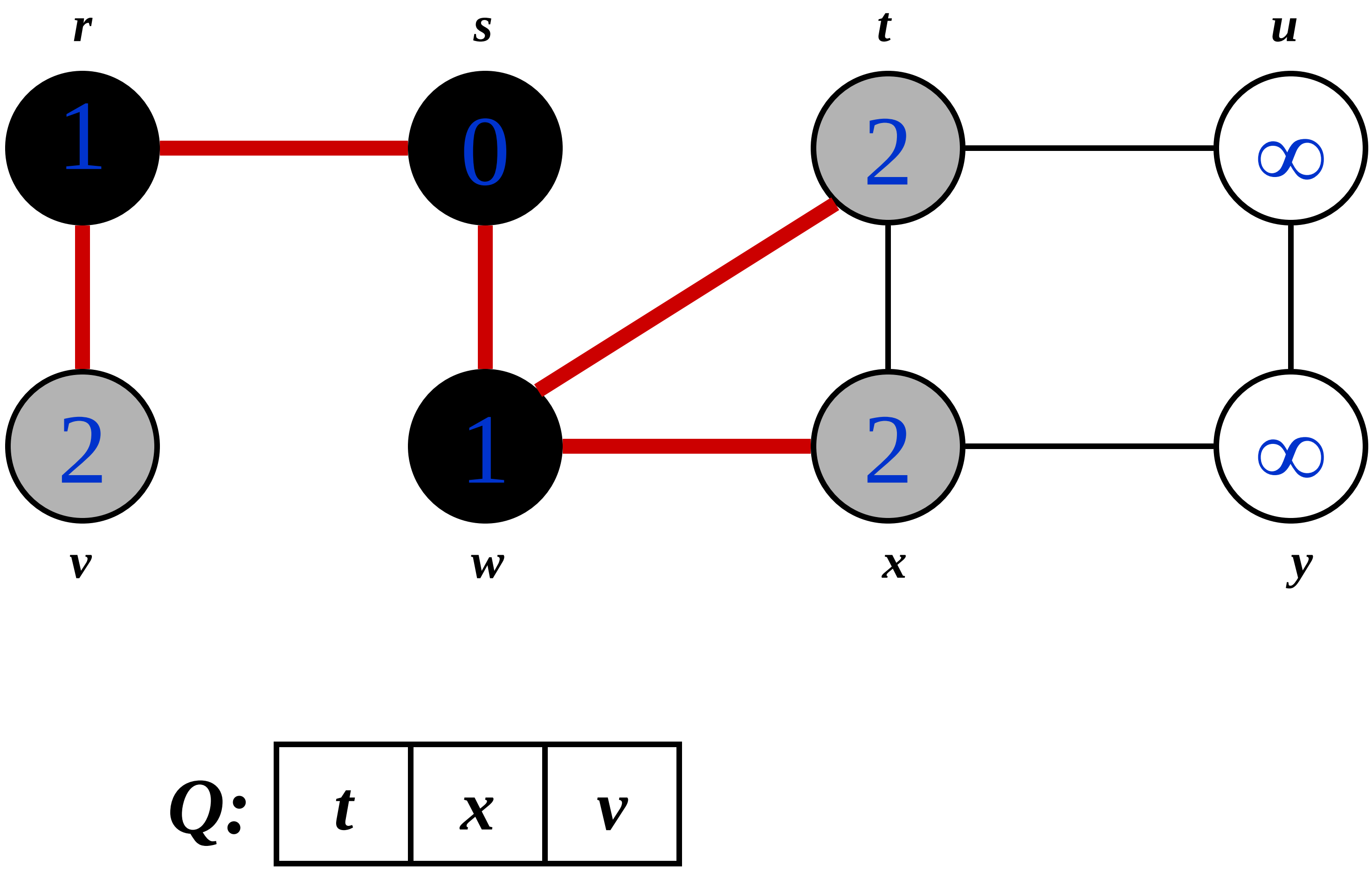
Breadth-First Search: Example



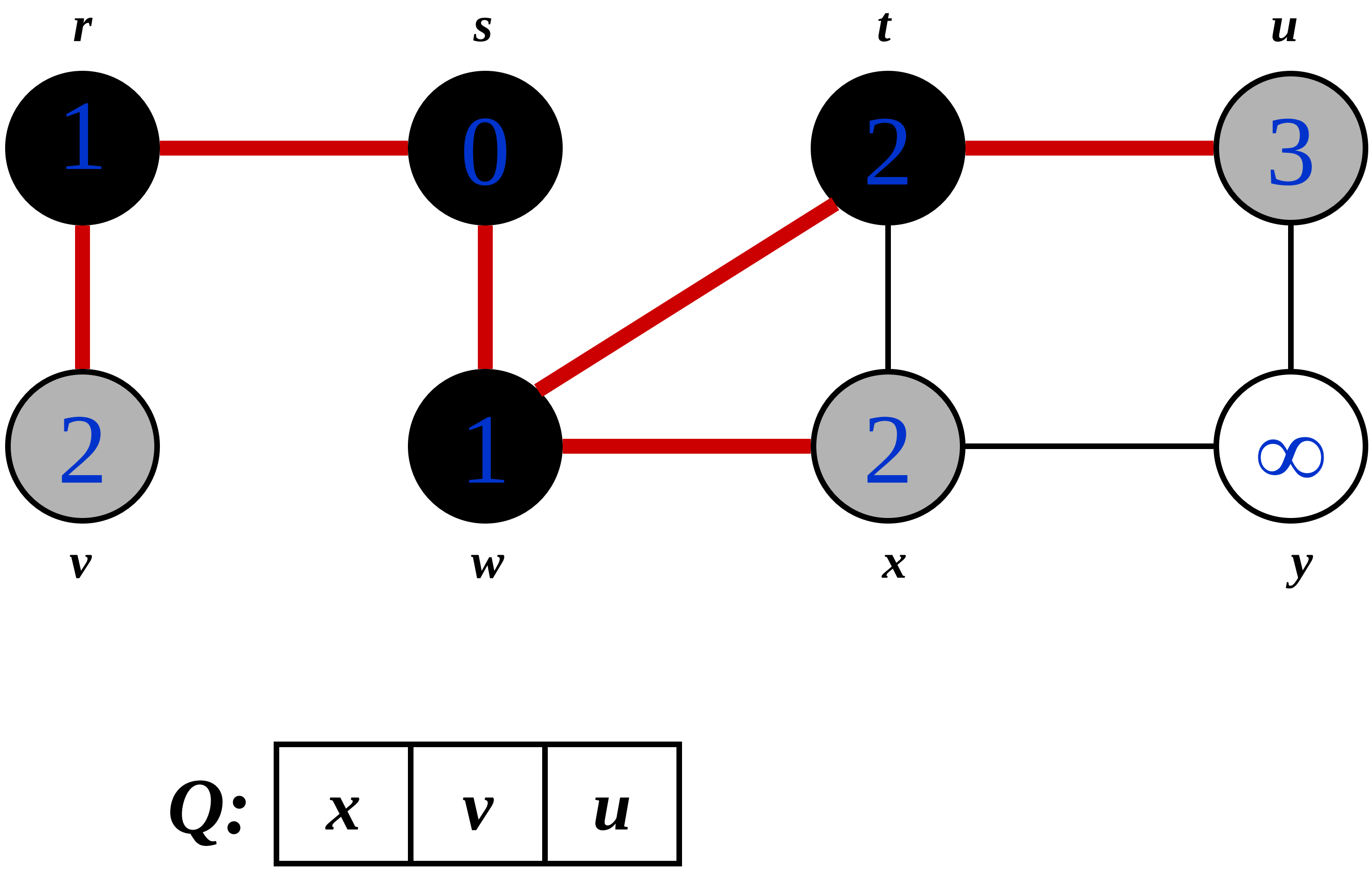
Q: | *r* | *t* | *x* |

r	*t*	*x*

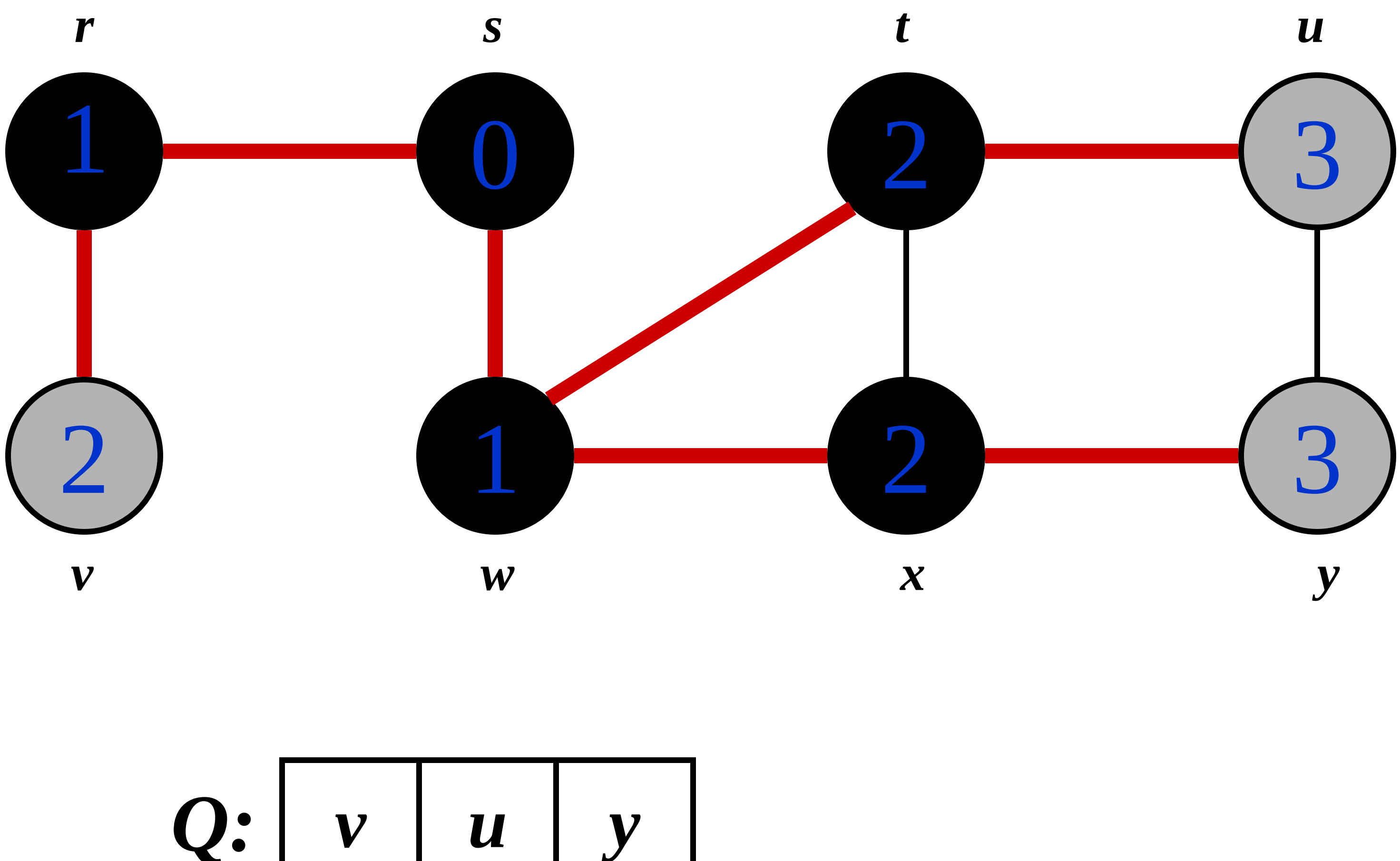
Breadth-First Search: Example



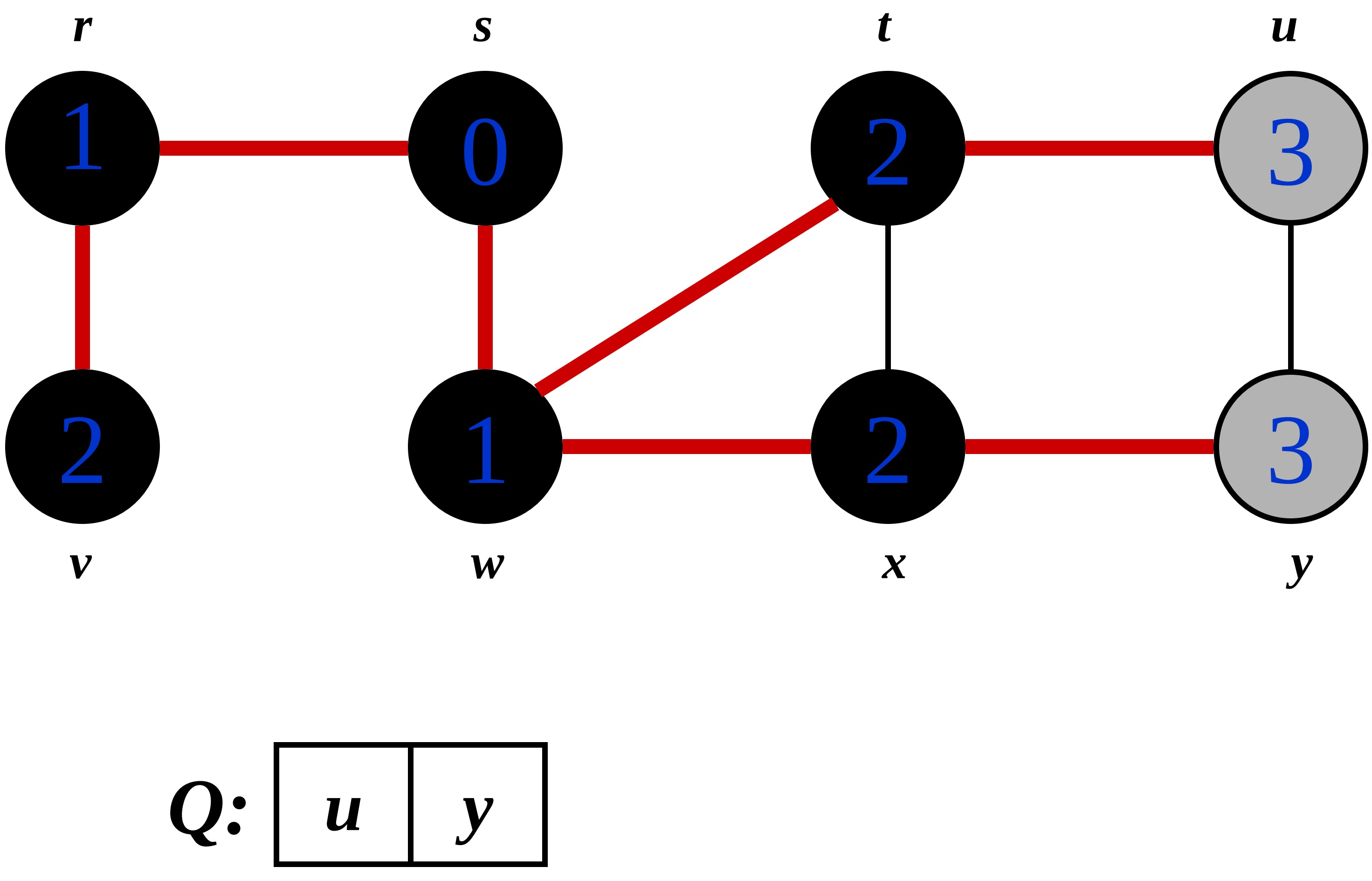
Breadth-First Search: Example



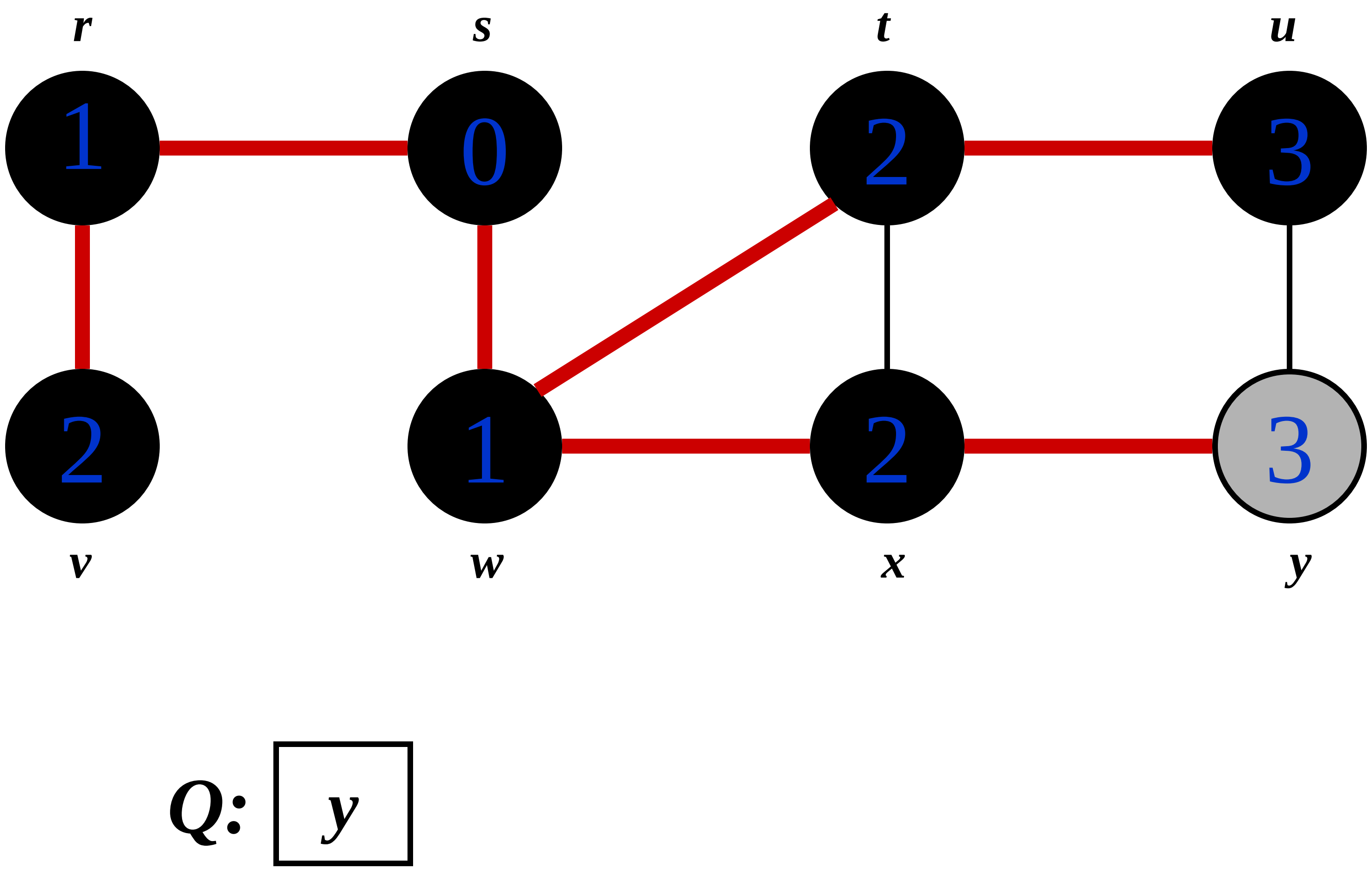
Breadth-First Search: Example



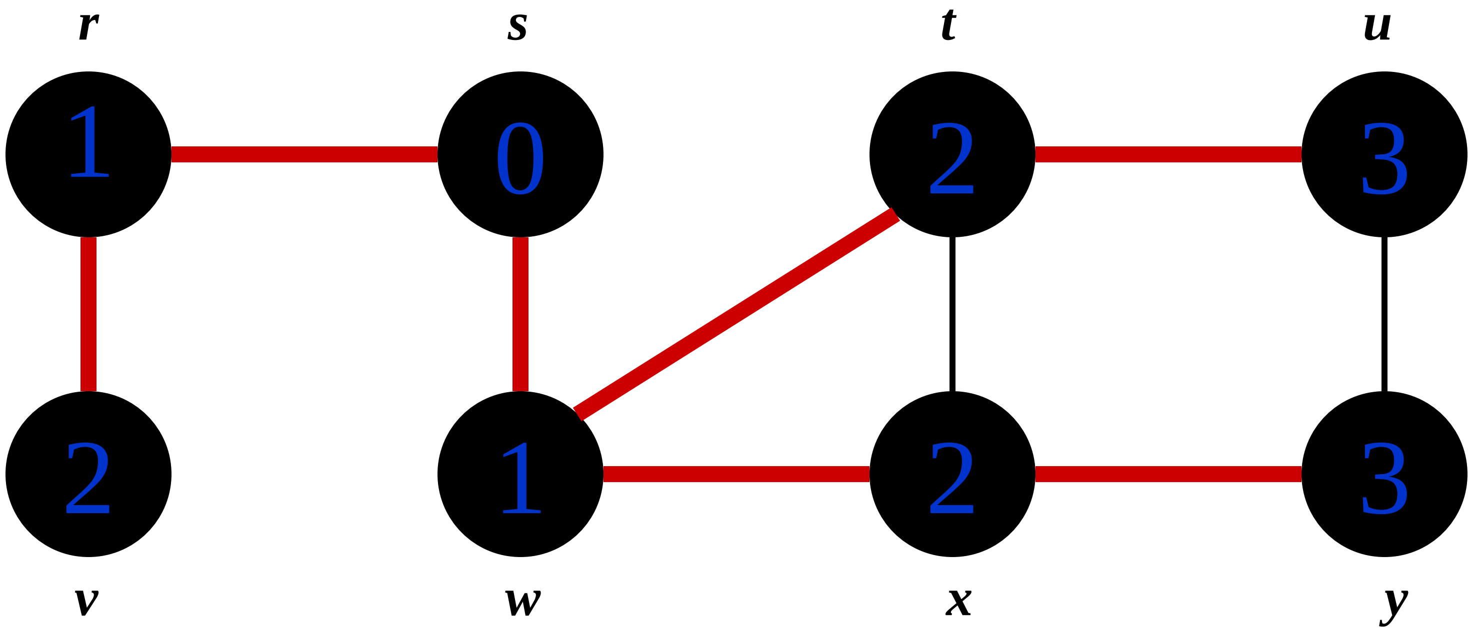
Breadth-First Search: Example



Breadth-First Search: Example



Breadth-First Search: Example



Q: $\varnothing$

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

- Like BFS it does not recover shortest paths, but can be useful for extracting other properties of graph, e.g.,
  - Topological sorts
  - Detection of cycles
  - Extraction of strongly connected components

Depth-First Search

- DFS create depth first forest
- Maintains Two timestamps on each vertex
  - *d*[v] →records when v is first discovered (and grayed)
  - *f*[v] →records when search finishes examining v's adjacency list (and blackens v)

- Explore *every* edge, starting from different vertices if necessary
- As soon as vertex discovered, explore from it.
- Keep track of progress by colouring vertices:
  - Vertices initially colored white
  - Then colored gray when discovered (but not finished still exploring it)
  - Then black when finished (found everything that reachable from it)

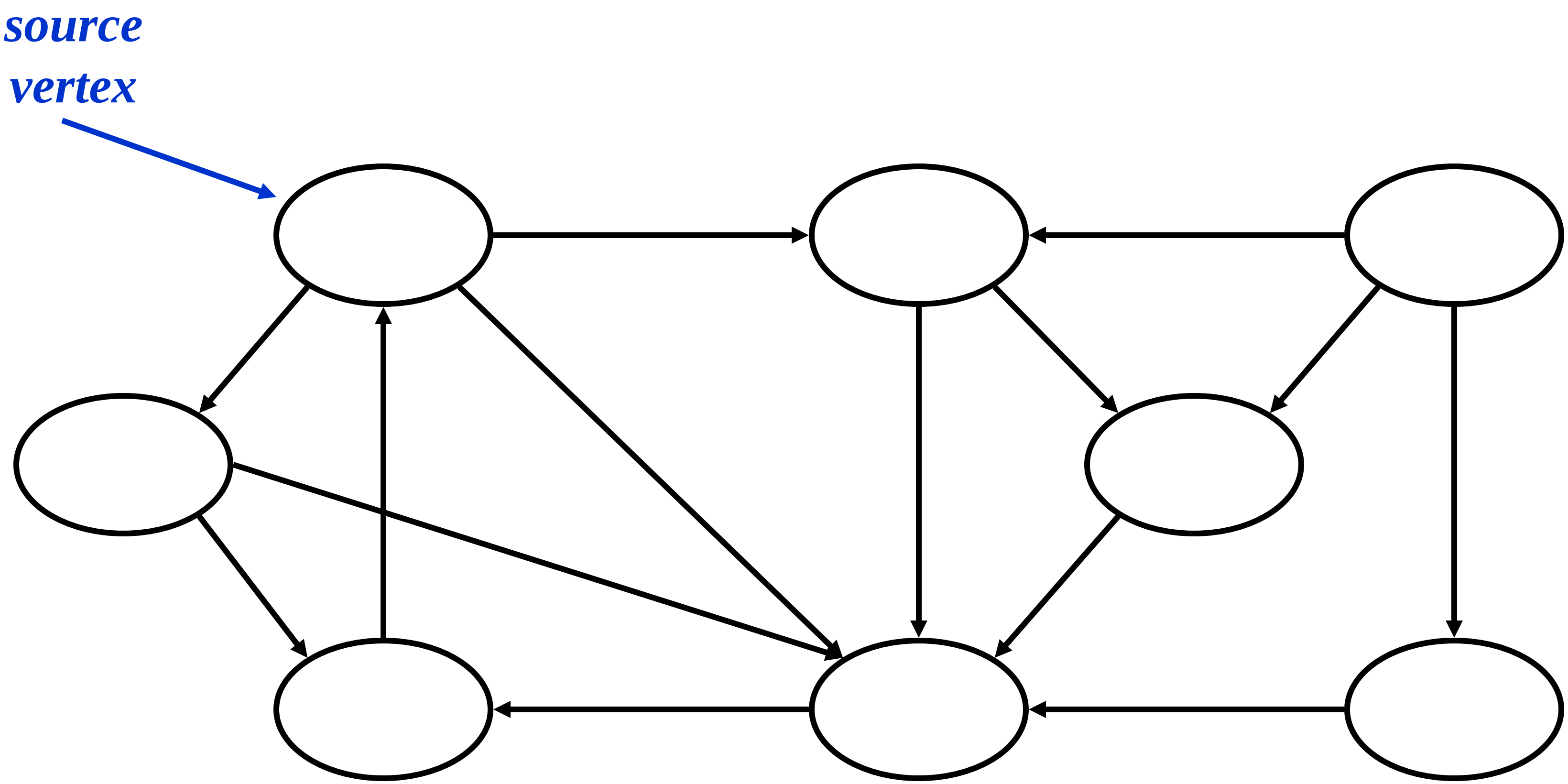
Depth-First Search Algorithm

```
DFS(G)
for each vertex u ∈ V[G]
        do color[u] ← WHITE
           π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
        do if color[u] = WHITE
              then DFS-VISIT(u)
```

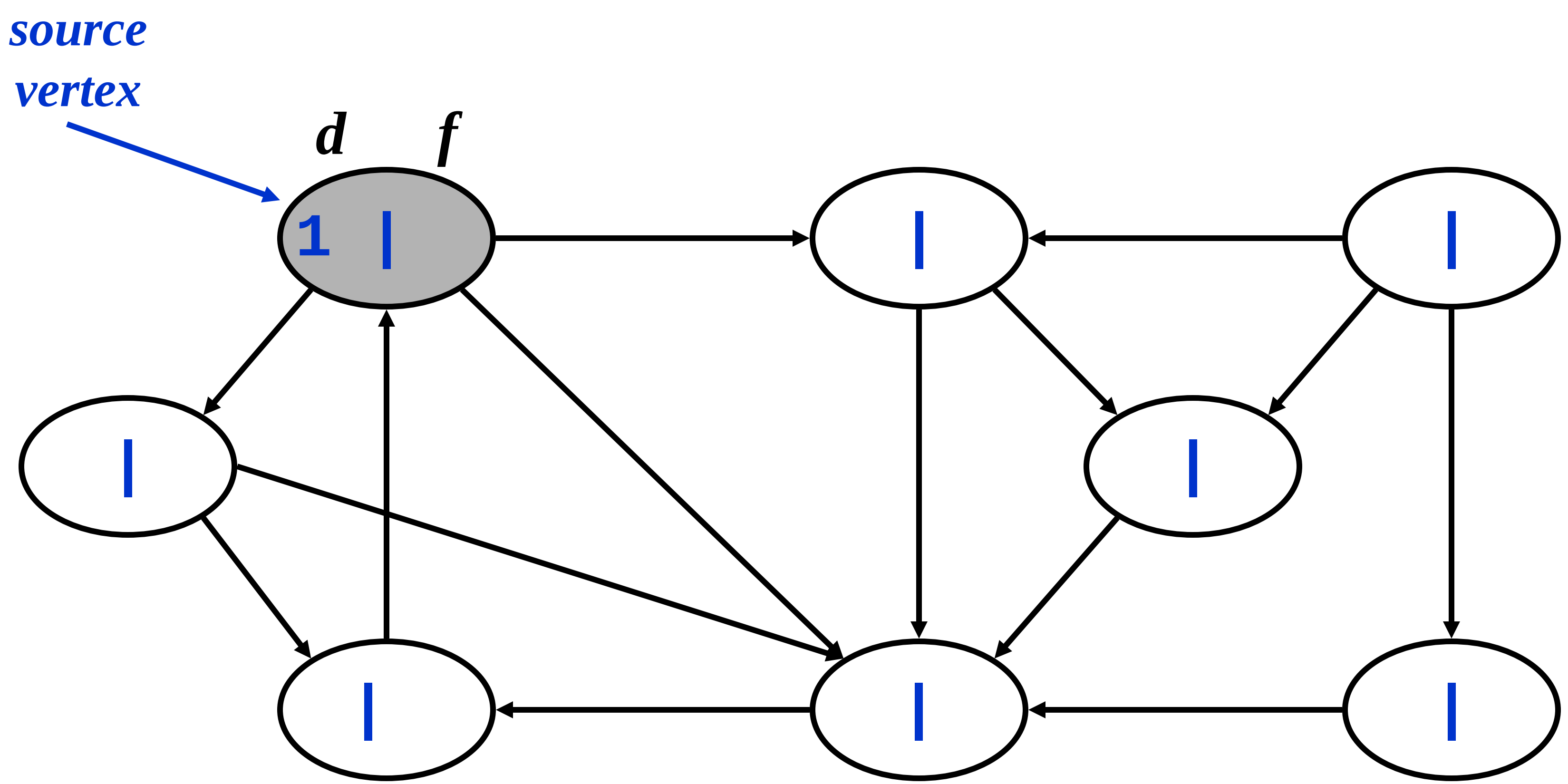
running time = O(V+E)

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

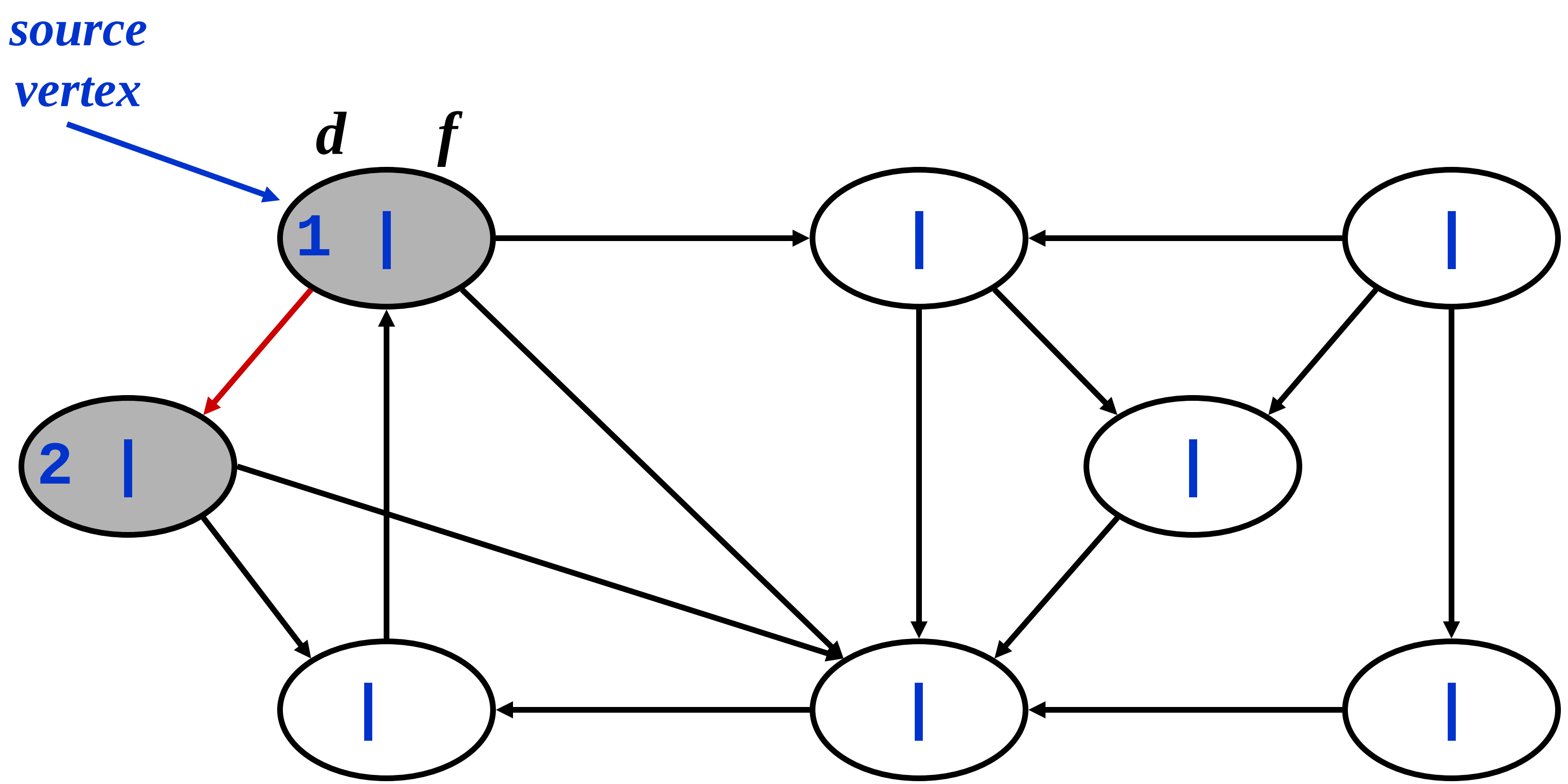
DFS Example



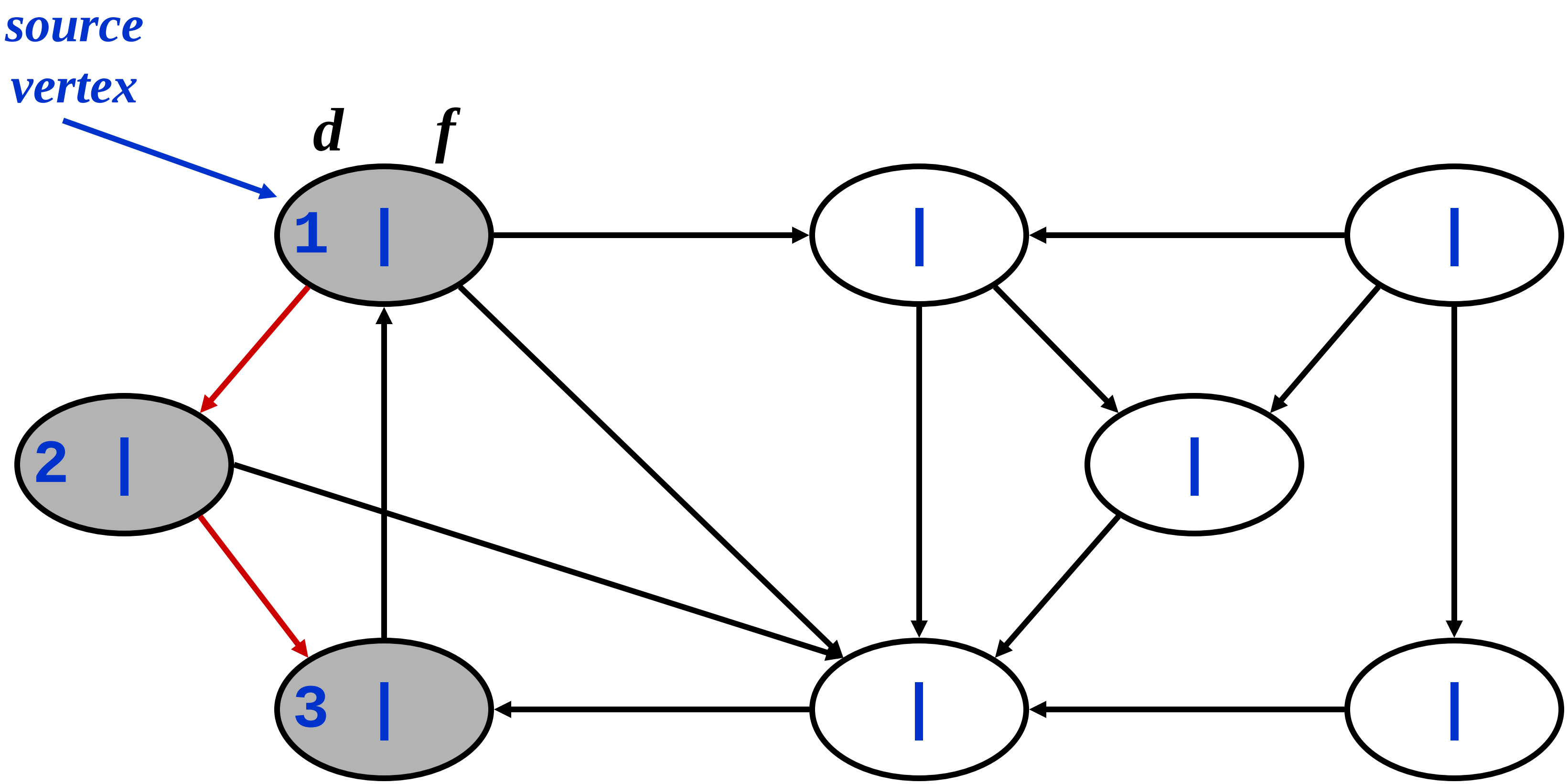
DFS Example



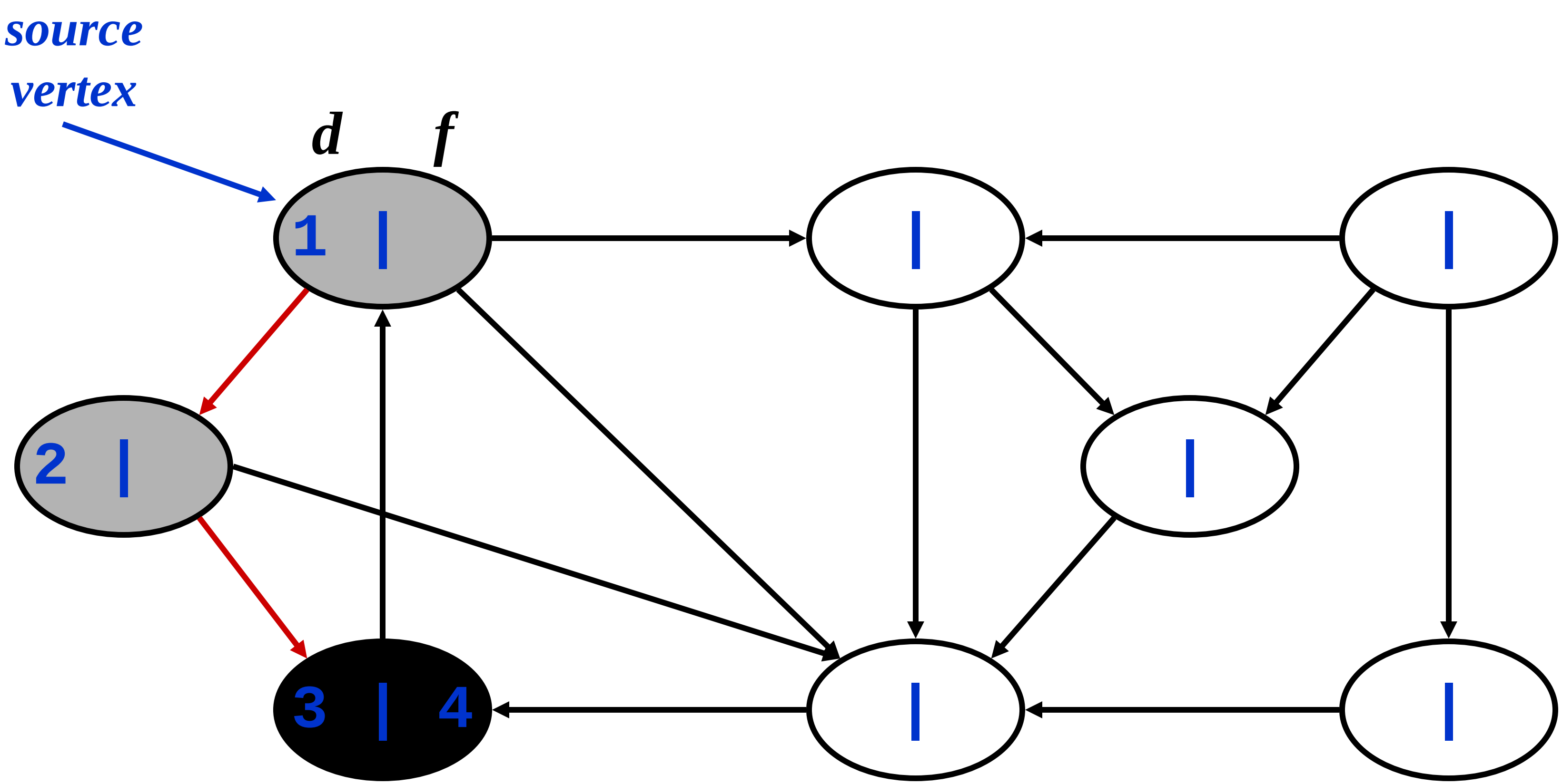
DFS Example



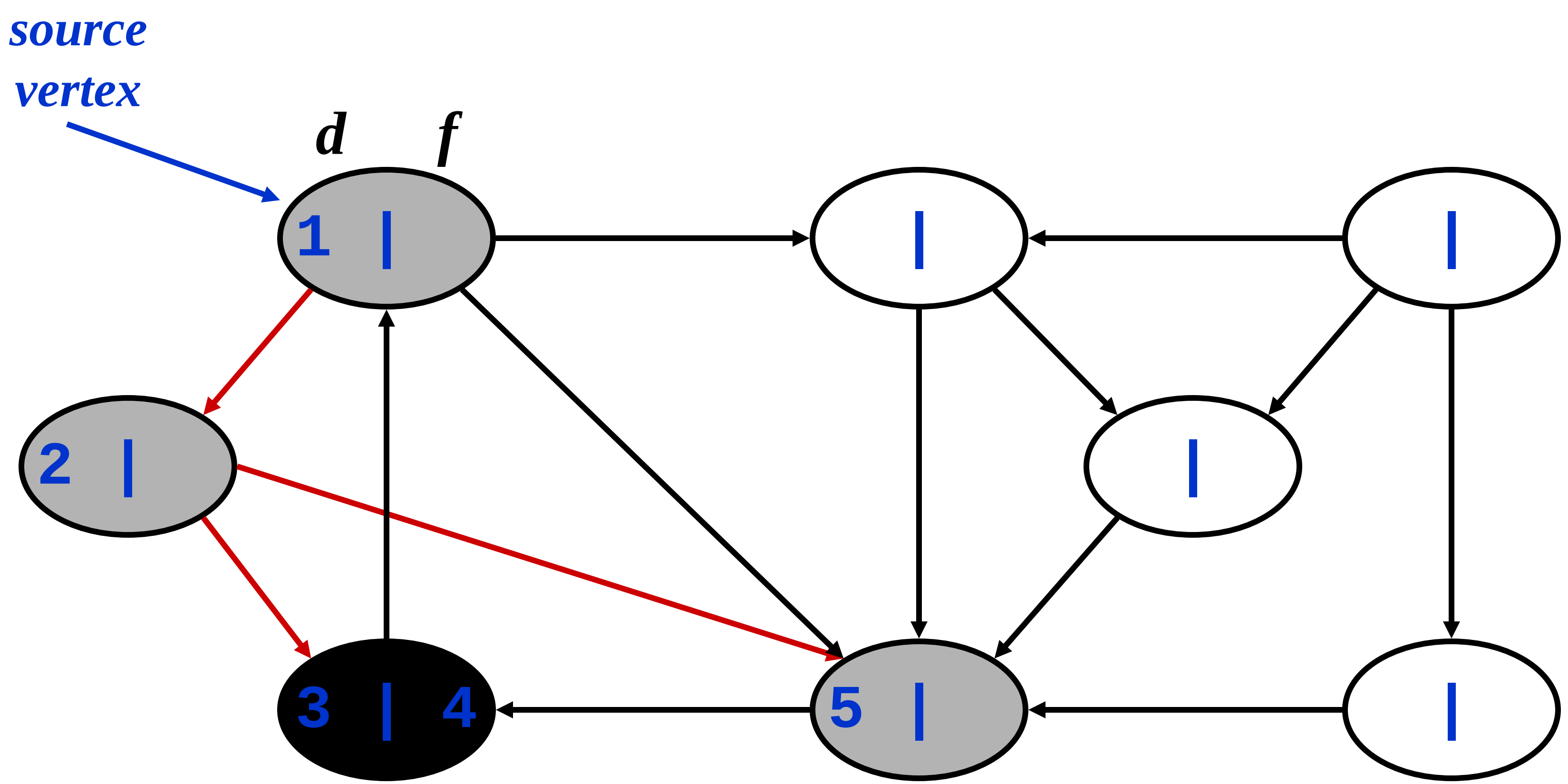
DFS Example



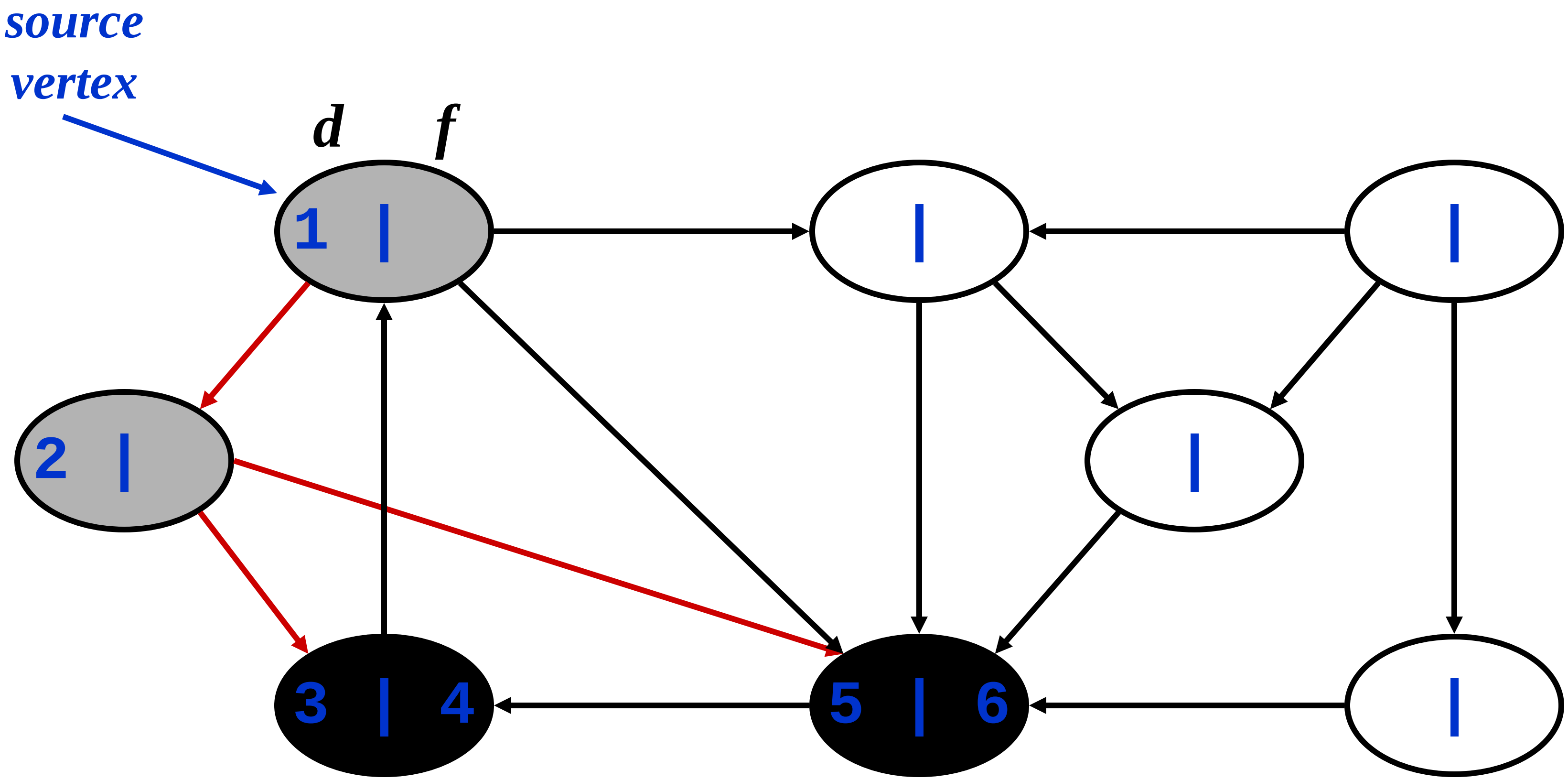
DFS Example



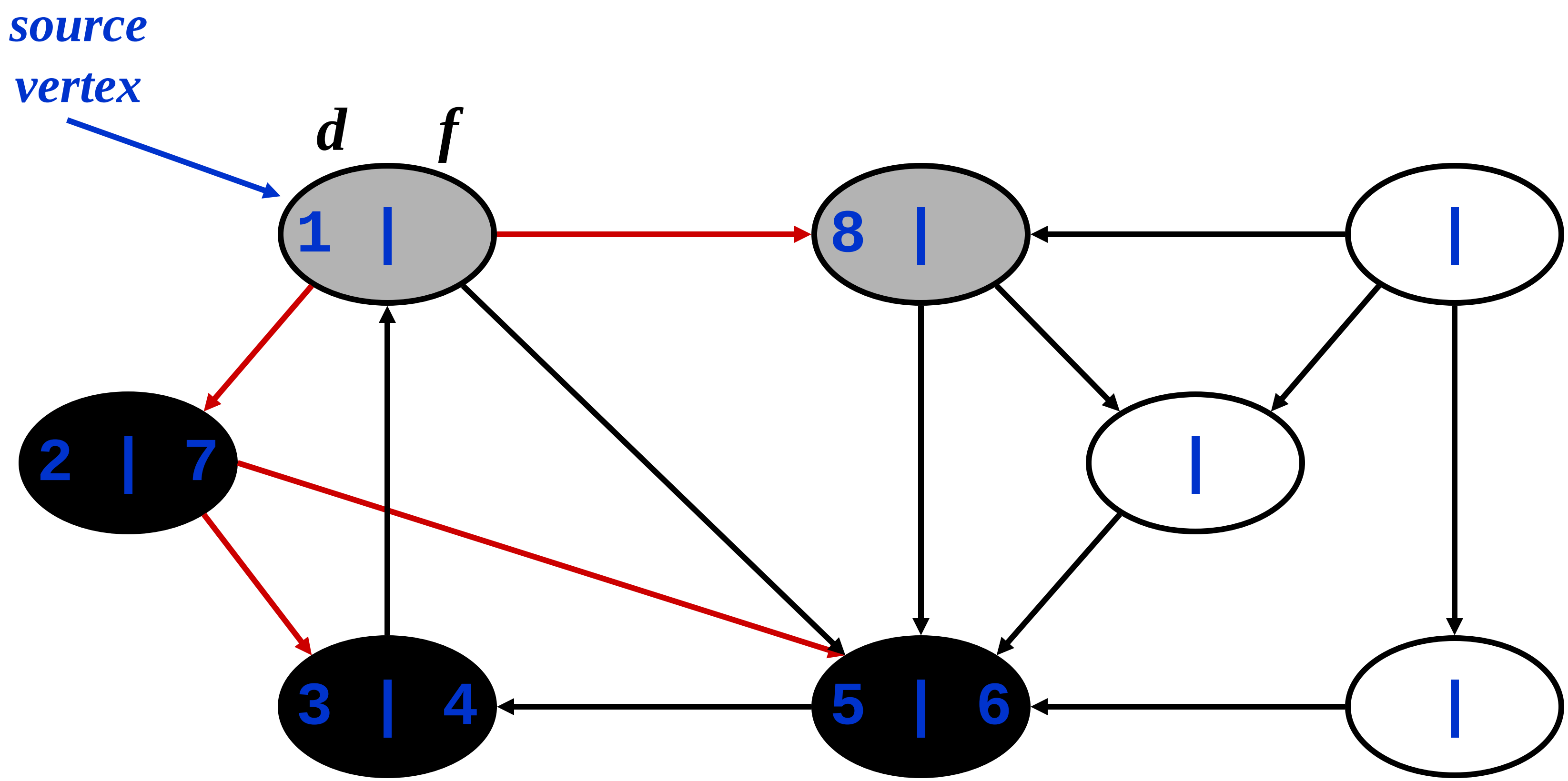
DFS Example



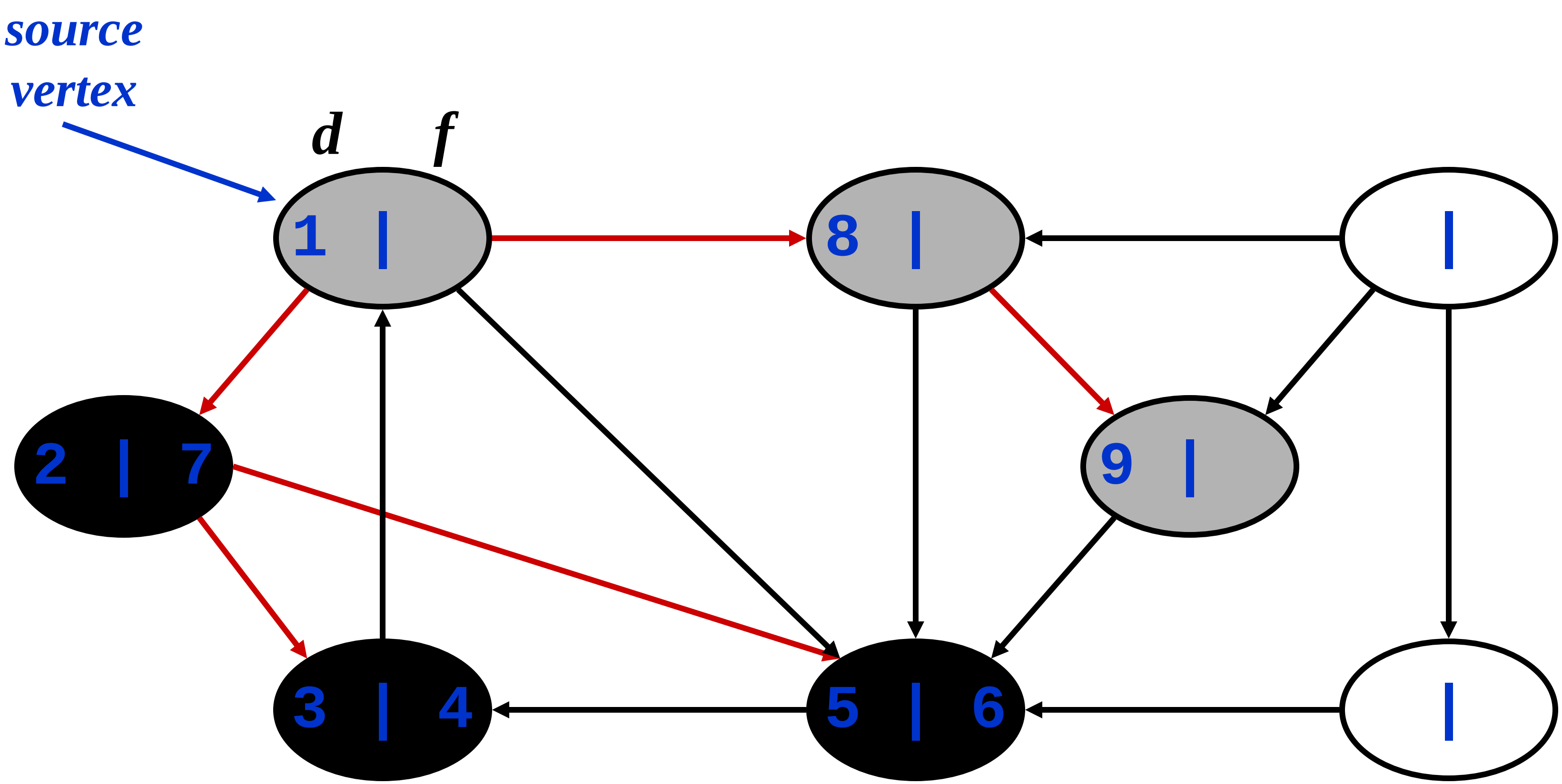
DFS Example



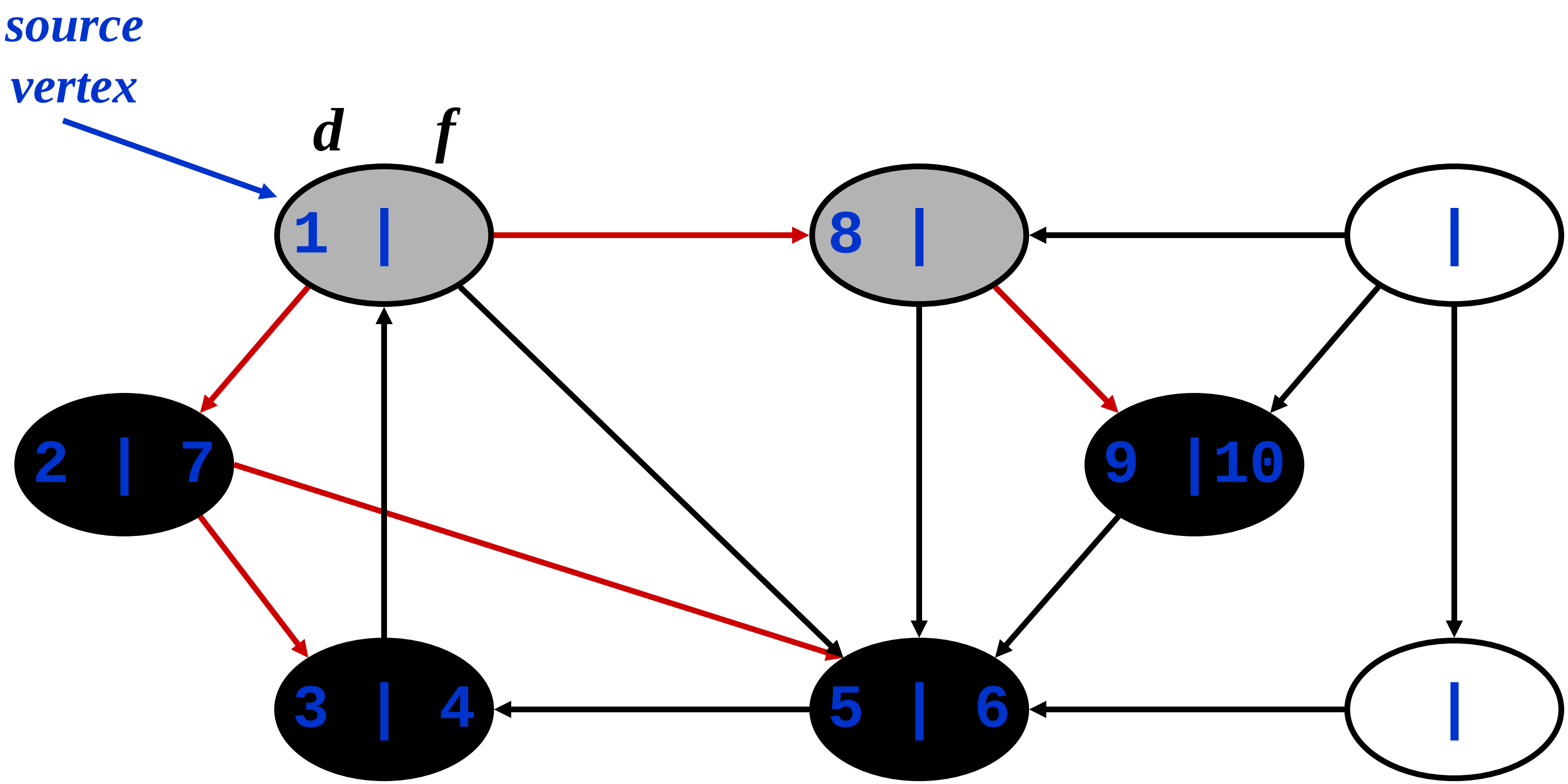
DFS Example



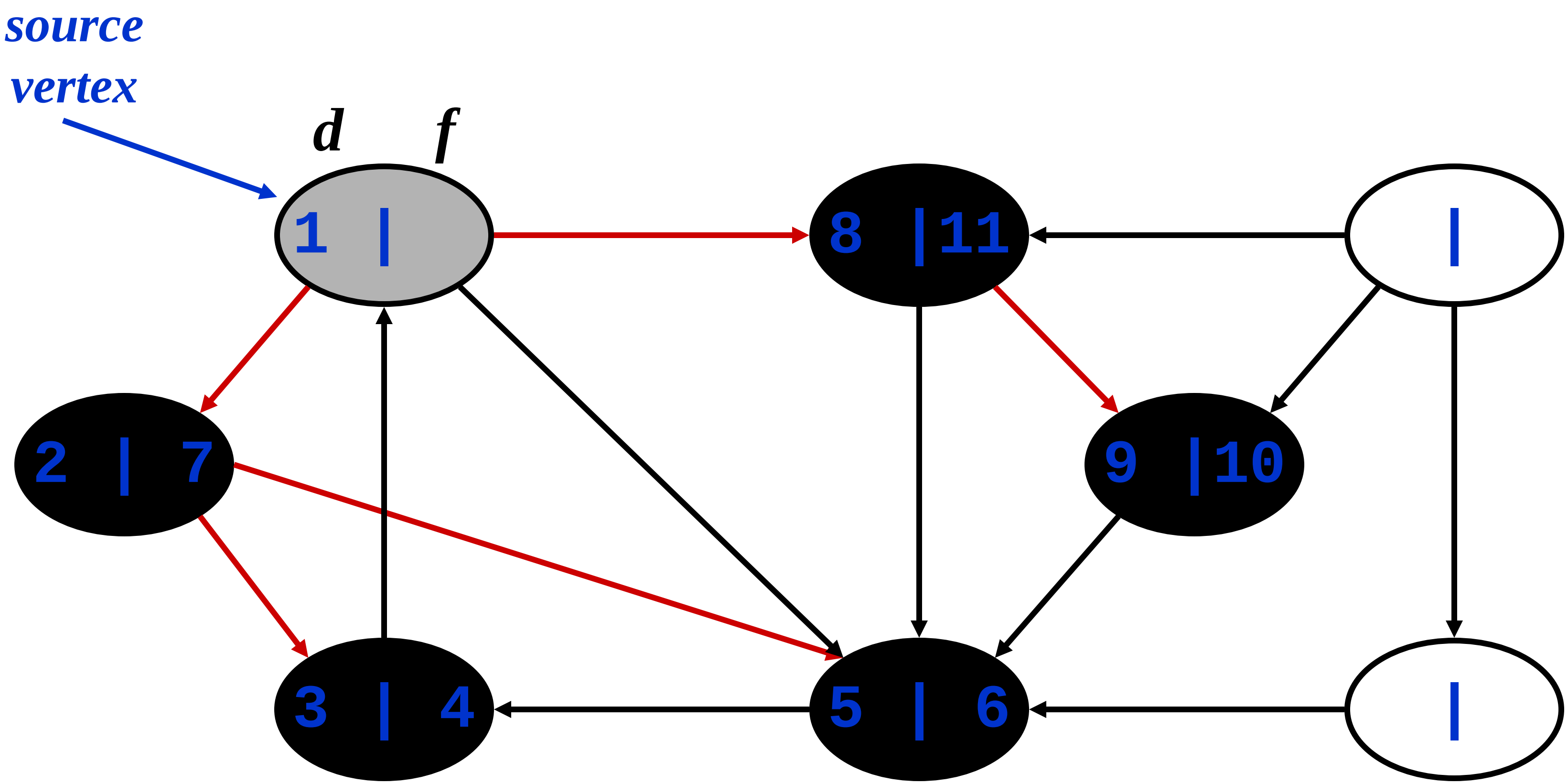
DFS Example



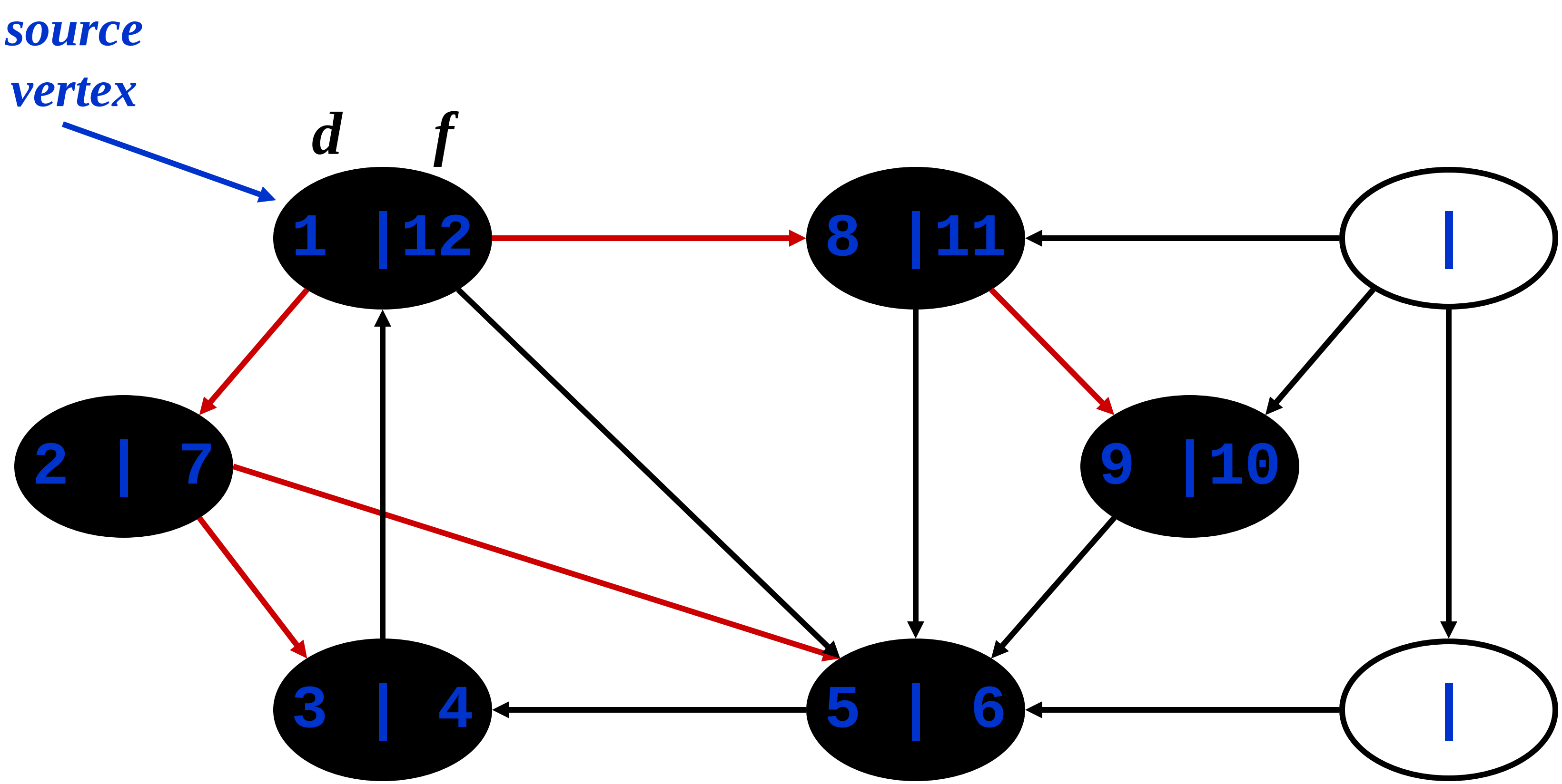
DFS Example



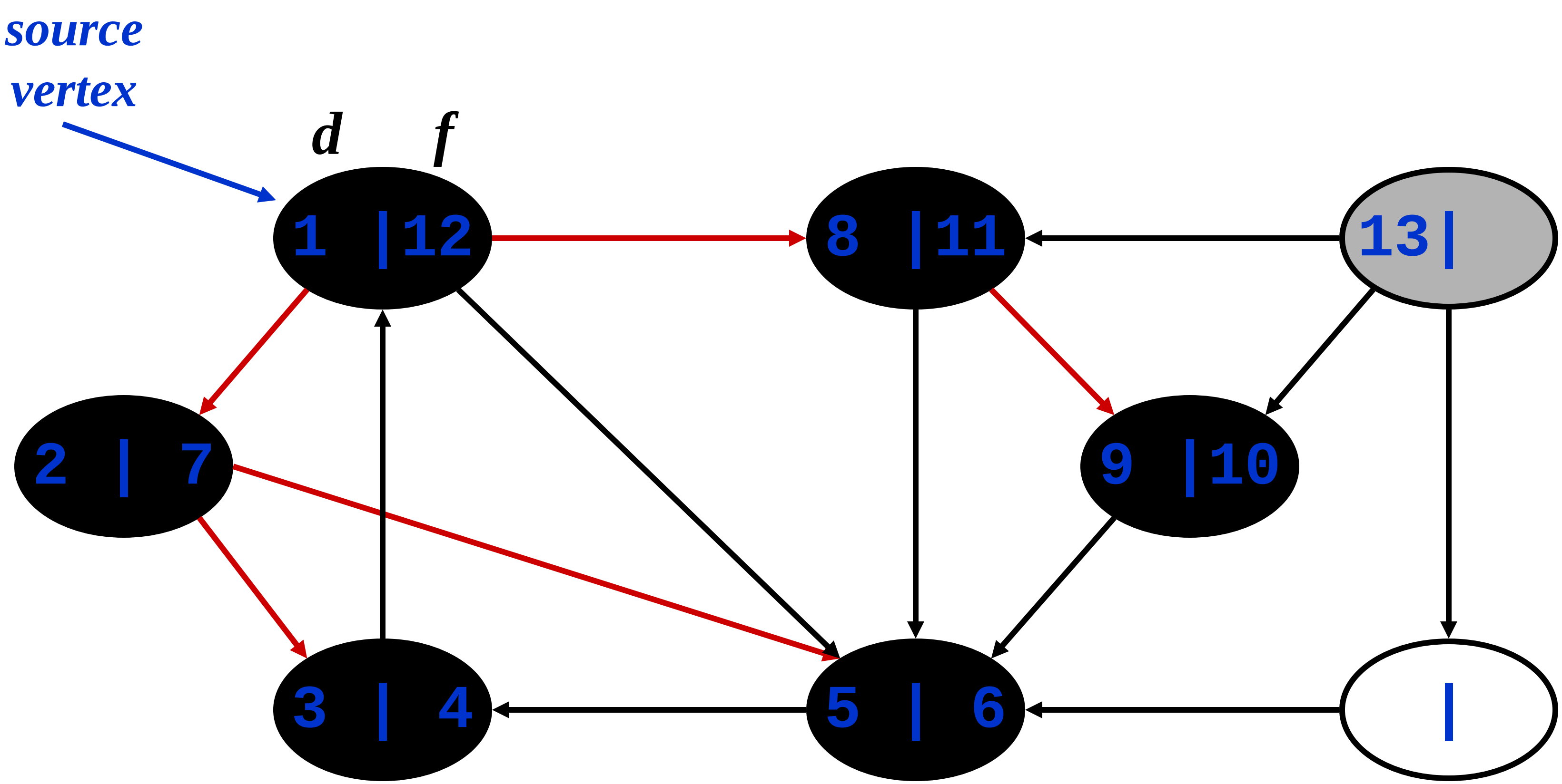
DFS Example



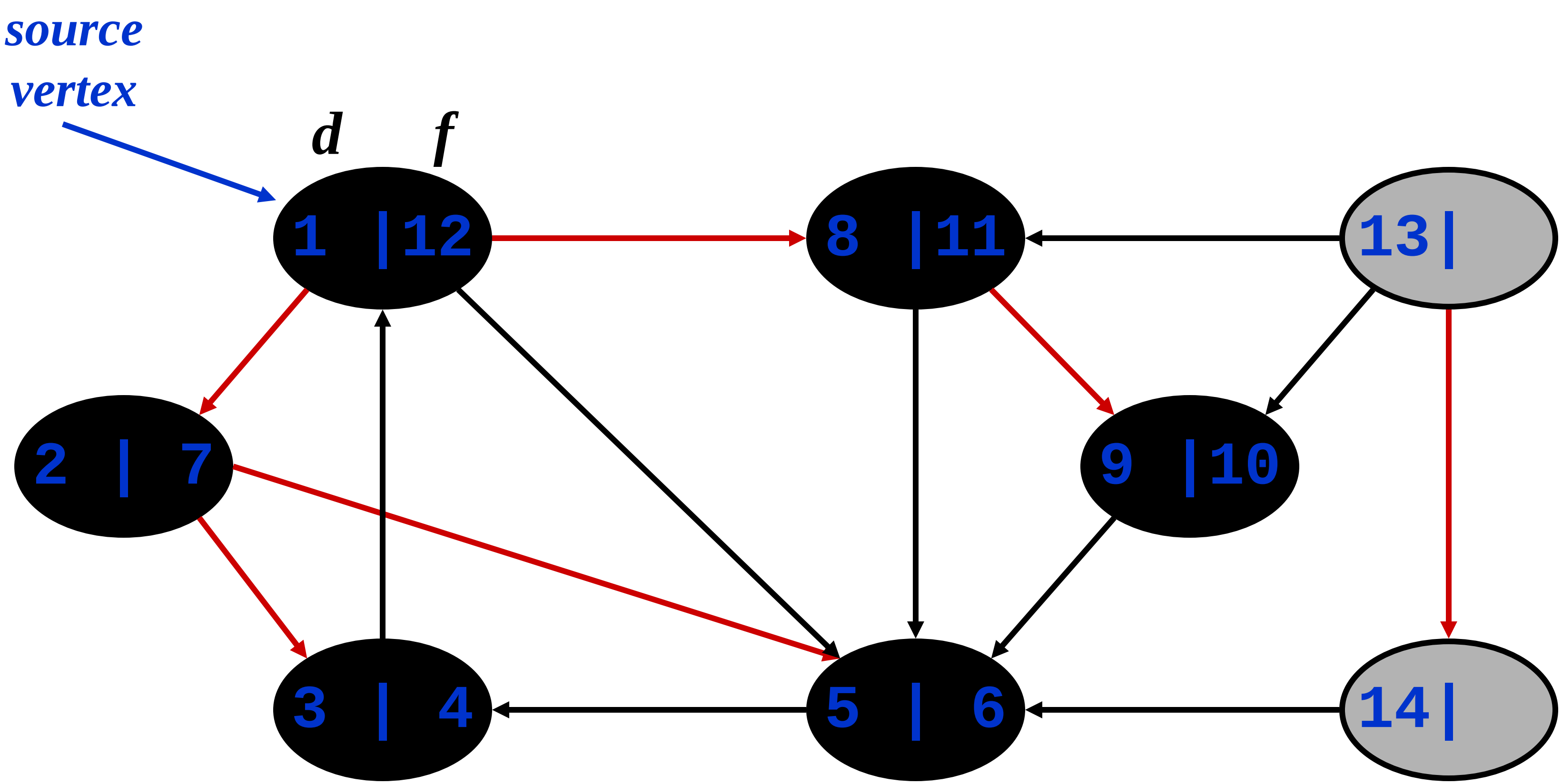
DFS Example



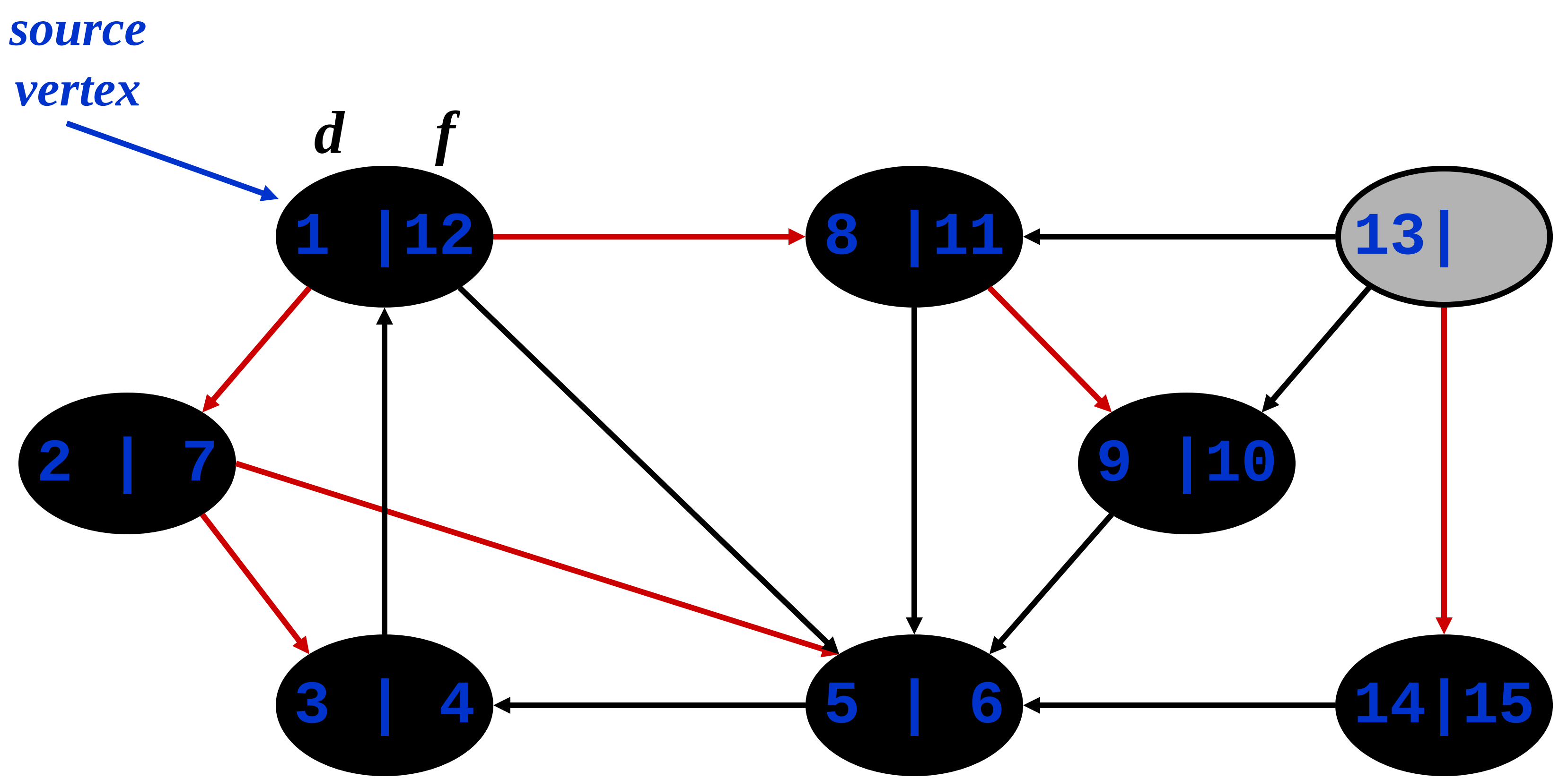
DFS Example



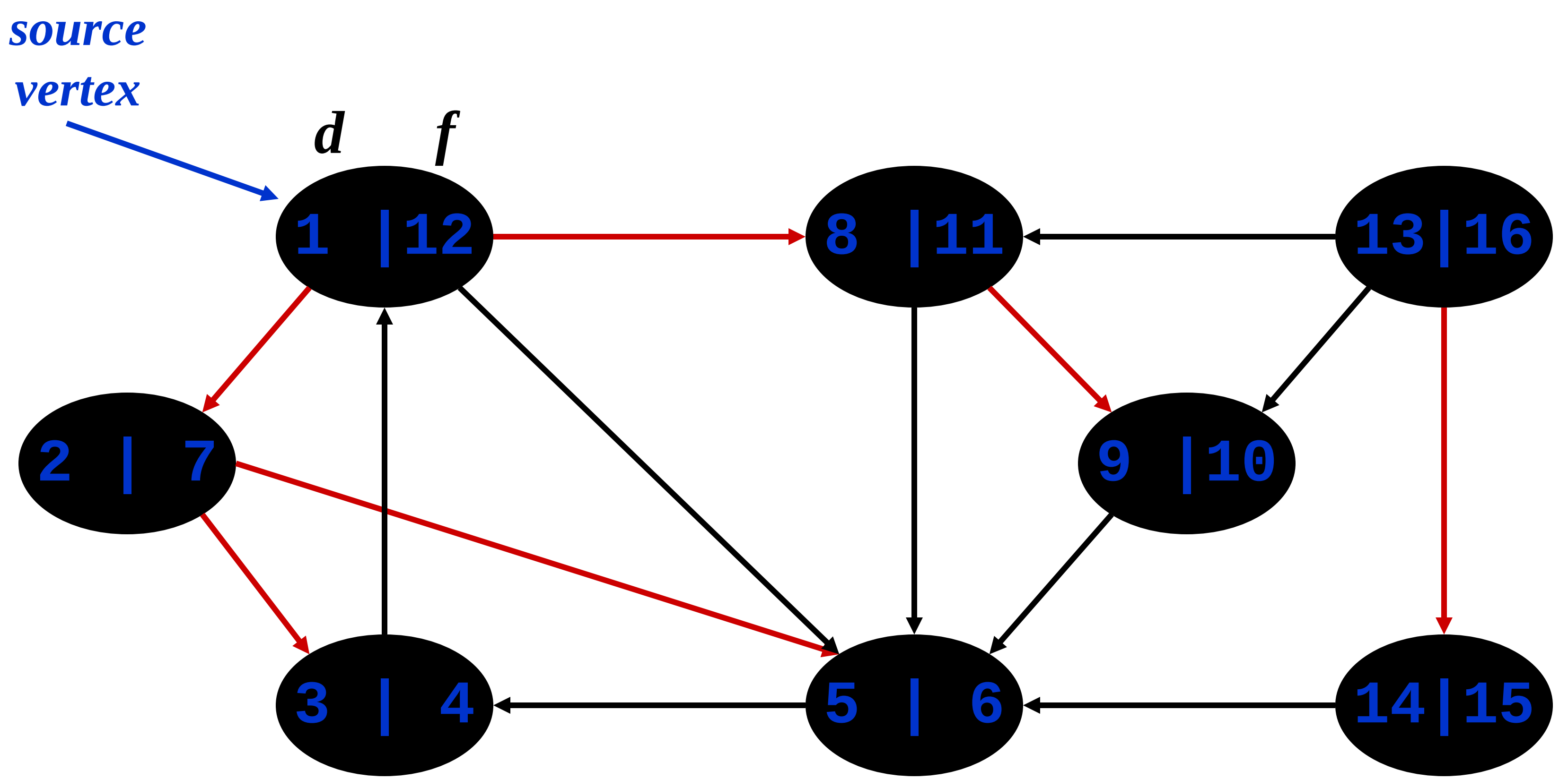
DFS Example



DFS Example



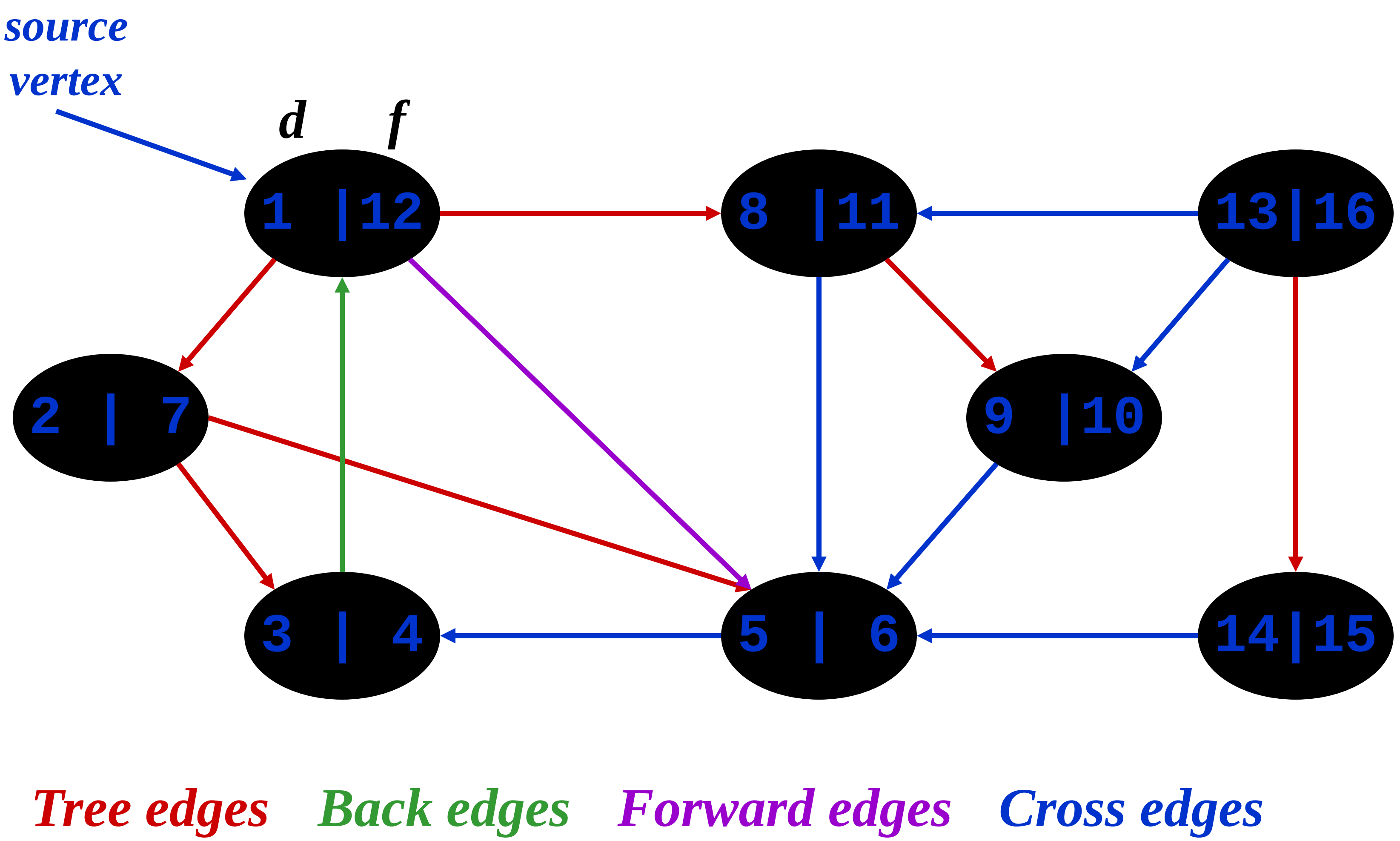
DFS Example



Classification of Edges

- Edge type for edge (u, v) can be identified when it is first explored by DFS.
- DFS algorithm can be modified to classify edges as it encounters them
- Classification is based on the **color of *v***
  - **White indicate a tree edge**: : encounter new (white) vertex
  - **Gray indicate a back edge**::from descendent to ancestor
    - Encounter a grey vertex (grey to grey)
  - **Black indicate a forward (or cross)edge**:: from ancestor to descendent
    - Not a tree edge, though
    - From grey node to black node
  - **Cross edge:** all other edges i.e. between a tree or subtrees
    - They can go between vertices in different depth-first trees

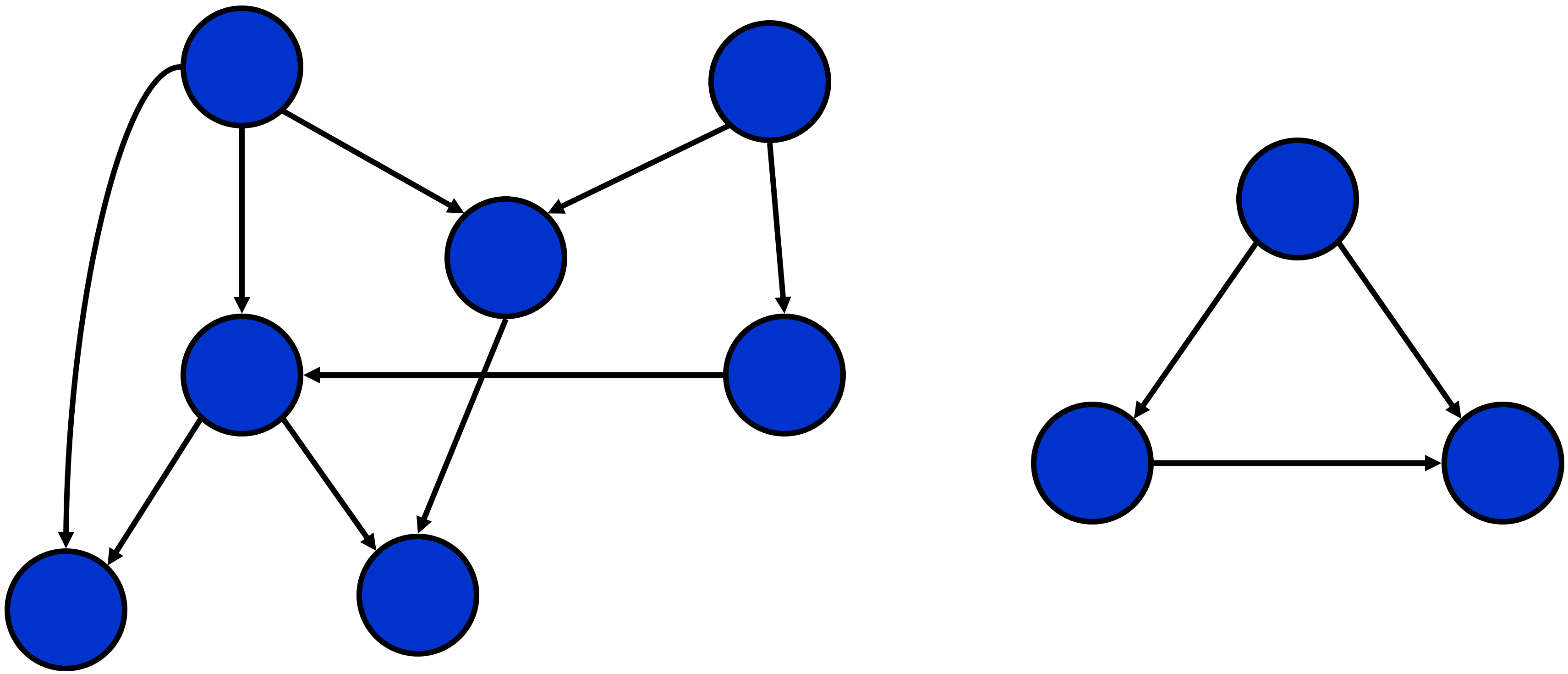
DFS Example



source vertex
d
f
1 |12
8 |11
13|16
2 | 7
9 |10
3 | 4
5 | 6
14|15
Tree edges
Back edges
Forward edges
Cross edges

Directed Acyclic Graphs

DAG is a directed graph with no directed cycles:



Used to indicate precedence among events in many applications

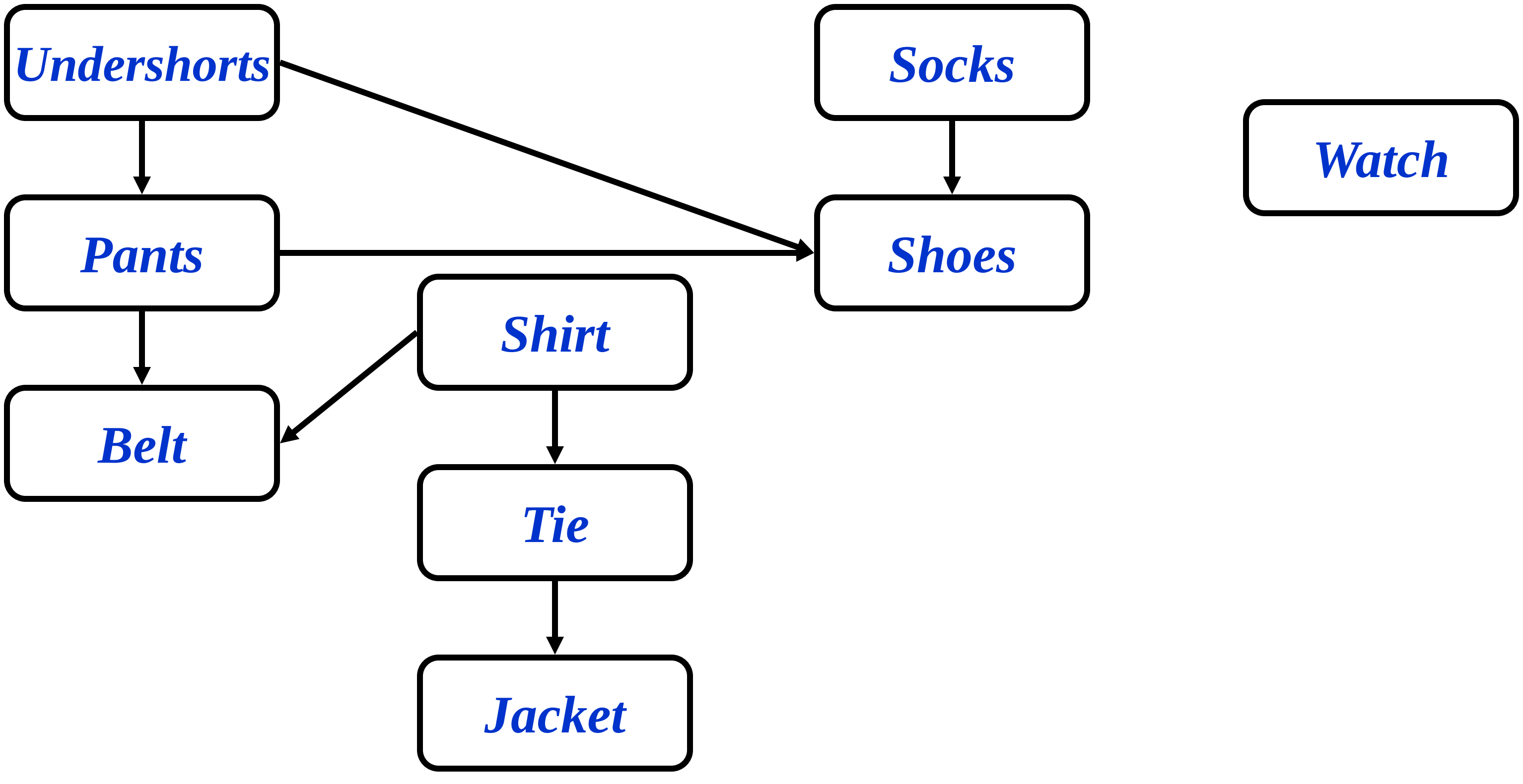
DFS and DAGs

- a directed graph G is acyclic if a DFS of G yields no back edges:
  - if G is acyclic, will be no back edges
    - But if a back edge it implies a cycle
  - if no back edges, G is acyclic

Topological Sort

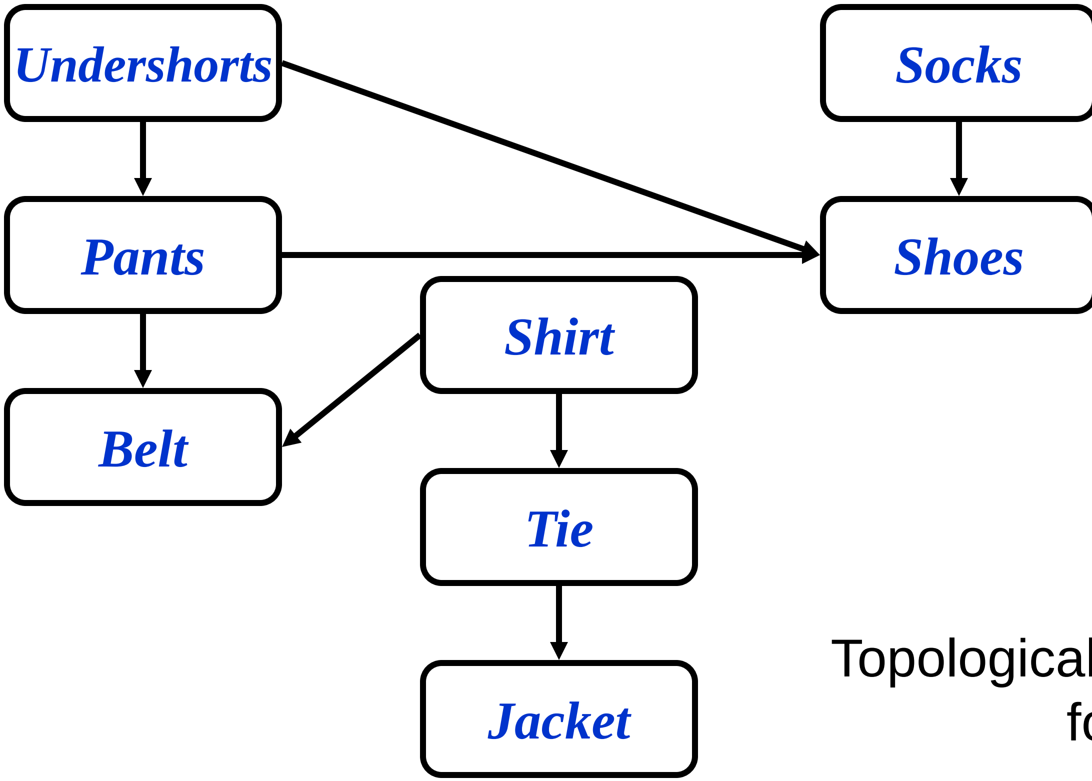
- *DFS used to perform topological sort*
- *Topological sort* of a DAG:
  - Linear ordering of all vertices in graph G such that vertex u comes before vertex v if edge $(u, v) \in G$
- Example:
  - Person getting dressed

Getting Dressed

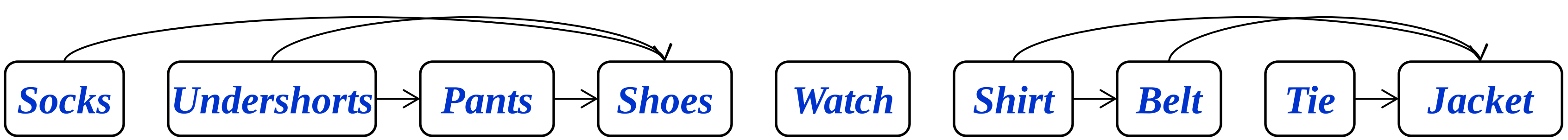


-certain garments put on before others
-other items may put on in any order
-directed edge (u,v) indicates that garment u must be put on before garment v

Getting Dressed



Topological sort of DAG gives an order for getting dressed

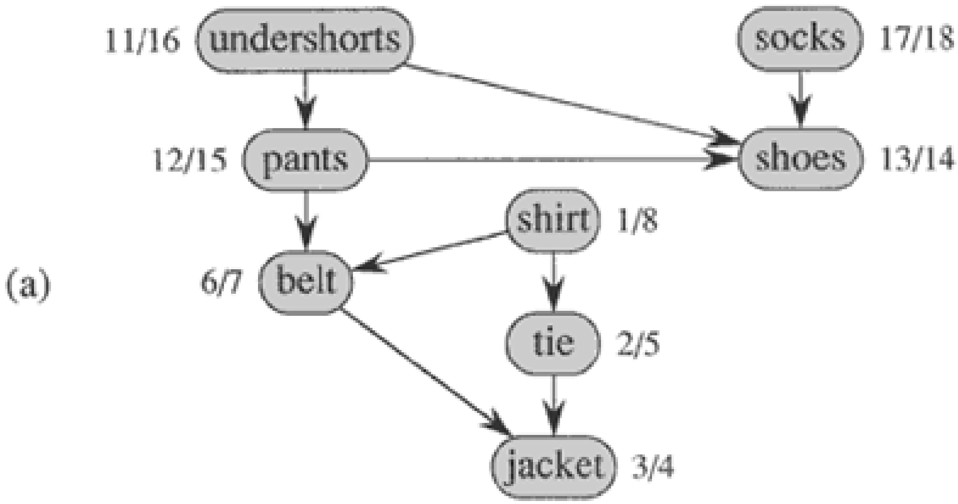


Topological Sort Algorithm

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

Running Time: O(V+E)

Topologically sorted graph



-Vertices arranged left to right
-All directed edges go from left to write