CMP302: Algorithms

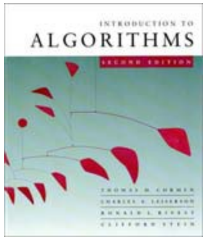
Lecture 14: Breadth First Search and Depth First Search

Mohamed Alaa El-Dien Aly
Computer Engineering Department
Cairo University
Fall 2013

Agenda

- Breadth First Search
- Depth First Search

Acknowledgment
A lot of slides adapted from the slides of David Luebke, Erik Demaine, and Charles Leiserson.



Graphs (review)

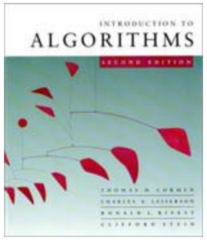
Definition. A ***directed graph (digraph)*** $G = (V, E)$ is an ordered pair consisting of

- a set V of ***vertices*** (singular: ***vertex***),
- a set $E \subseteq V \times V$ of ***edges***.

In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices.

In either case, we have $|E| = O(V^2)$. Moreover, if G is connected, then $|E| \geq |V| - 1$, which implies that $\lg|E| = \Theta(\lg V)$.

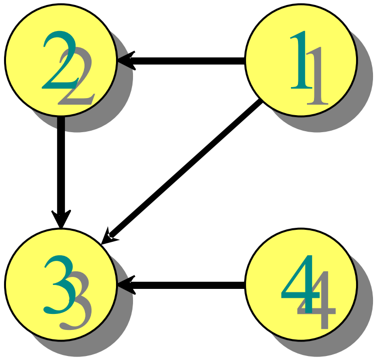
(Review CLRS, Appendix B.)



Adjacency-matrix representation

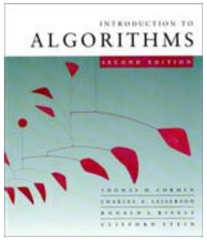
The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1\,.\,.\,n, 1\,.\,.\,n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$



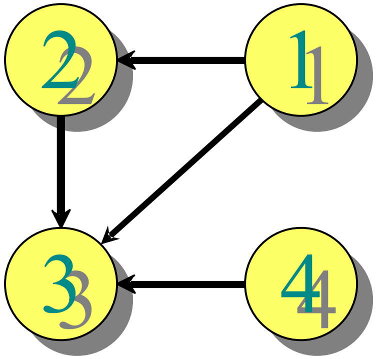
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

$\Theta(V^2)$ storage $\Rightarrow$ ***dense*** representation.

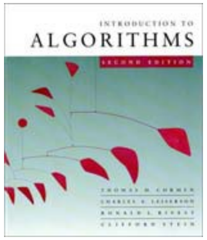


Adjacency-list representation

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.

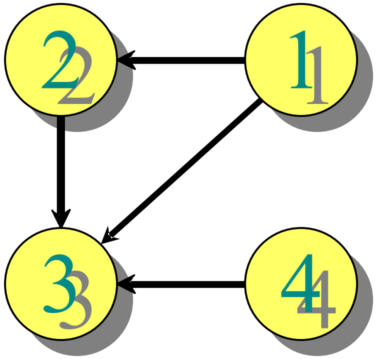


$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$



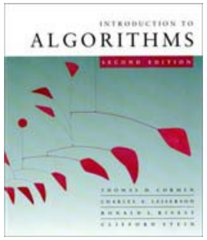
Adjacency-list representation

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



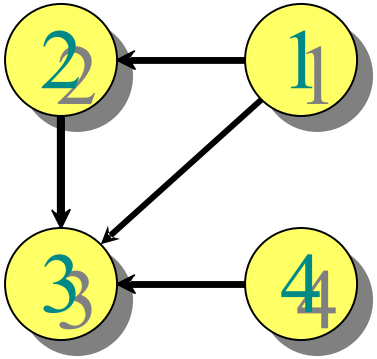
$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.



Adjacency-list representation

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

Handshaking Lemma: $\sum_{v \in V} |Adj[v]| = 2\,|E|$ for undirected graphs
i.e. adjacency lists use $\Theta(V + E)$ storage → a sparse representation for either type of graph

Graph Searching

Given: a graph $G = (V, E)$, directed or undirected

Goal: methodically explore every vertex and every edge *or* find a path from a *start* vertex to a *desired* vertex

Ultimately: build a tree on the graph

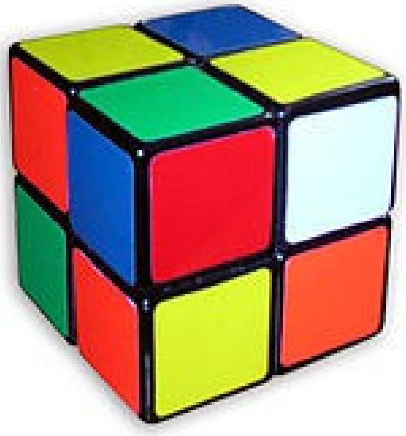
- Pick a vertex as the root
- Choose certain edges to produce a tree
- Note: might also build a *forest* if graph is not connected

Pocket Cube

2x2x2 Rubik's cube

Goal. Starting from a given configuration, find the steps to reach the goal configuration i.e. solve the cube.

Solution. Represent each *state* as a vertex in a configuration graph, and *search* the graph to solve the problem.



http://en.wikipedia.org/wiki/Pocket_Cube

http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/Jerry_Bryan__God's_Algorithm_for_the_2x2x2_Pocket_Cube.html

Breadth-First Search (BFS)

"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

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

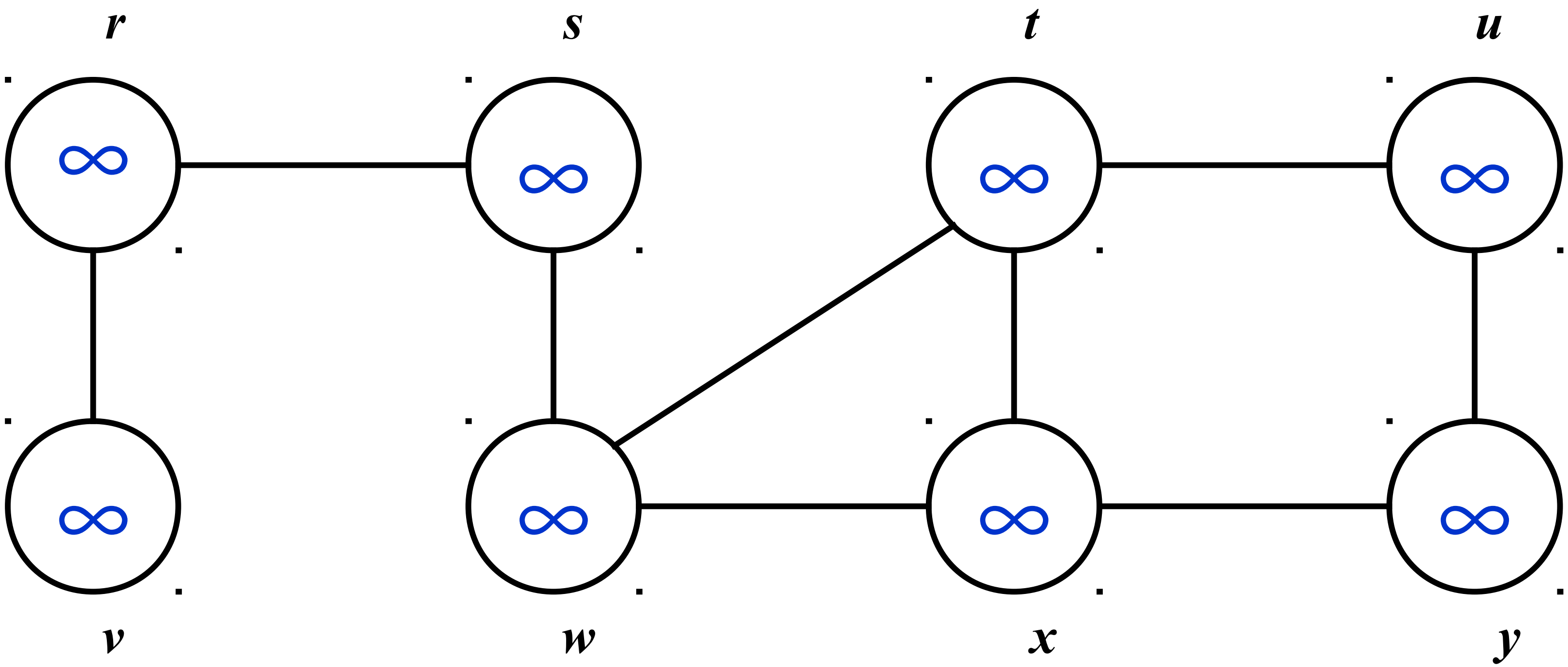
Breadth-First Search

```
BFS(G, s):
    for each v ∈ V:
        v.d = ∞
        v.p = NIL
        v.color = WHITE

    s.d = 0
    s.color = GREY
    Q = {s}                 // Initialize to s

    while (Q not empty):
        u = Dequeue(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
```

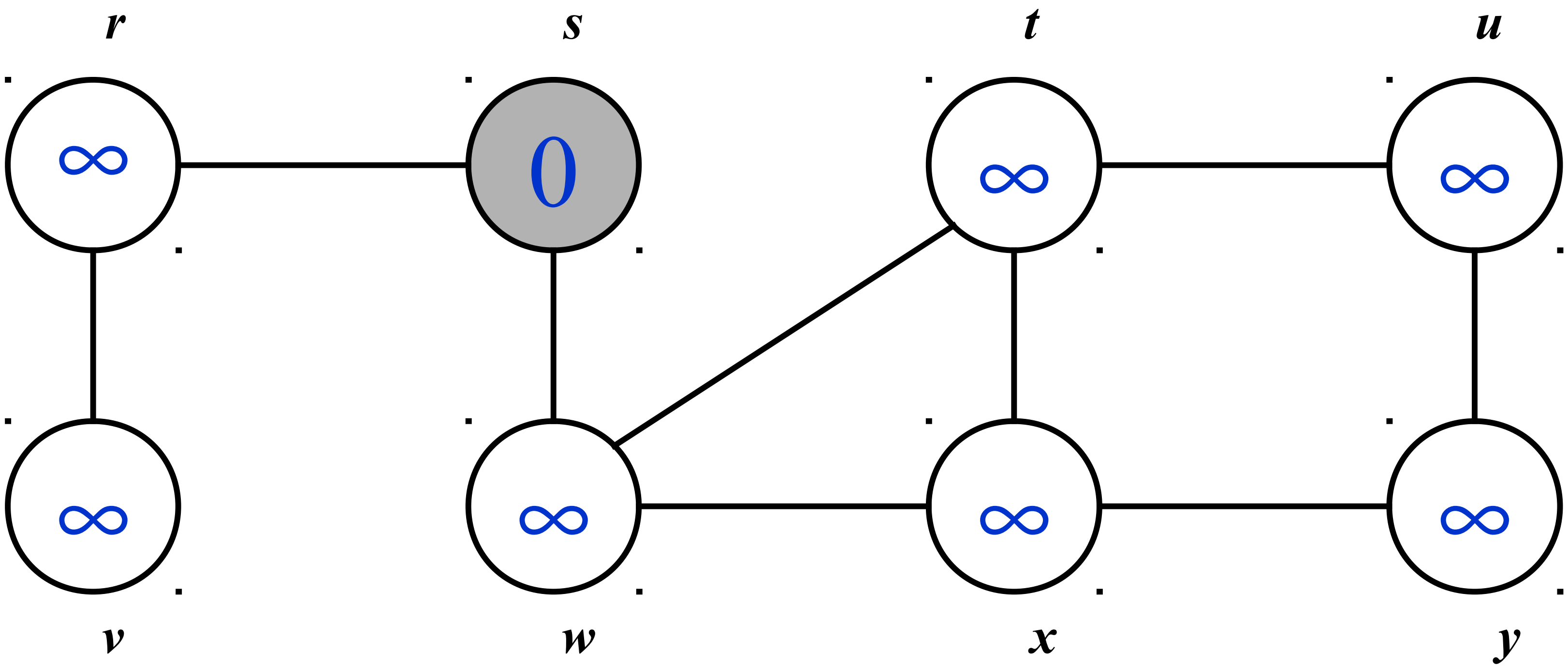
What does* `v.d` *represent?

What does* `v.p` *represent?

Breadth-First Search: Example

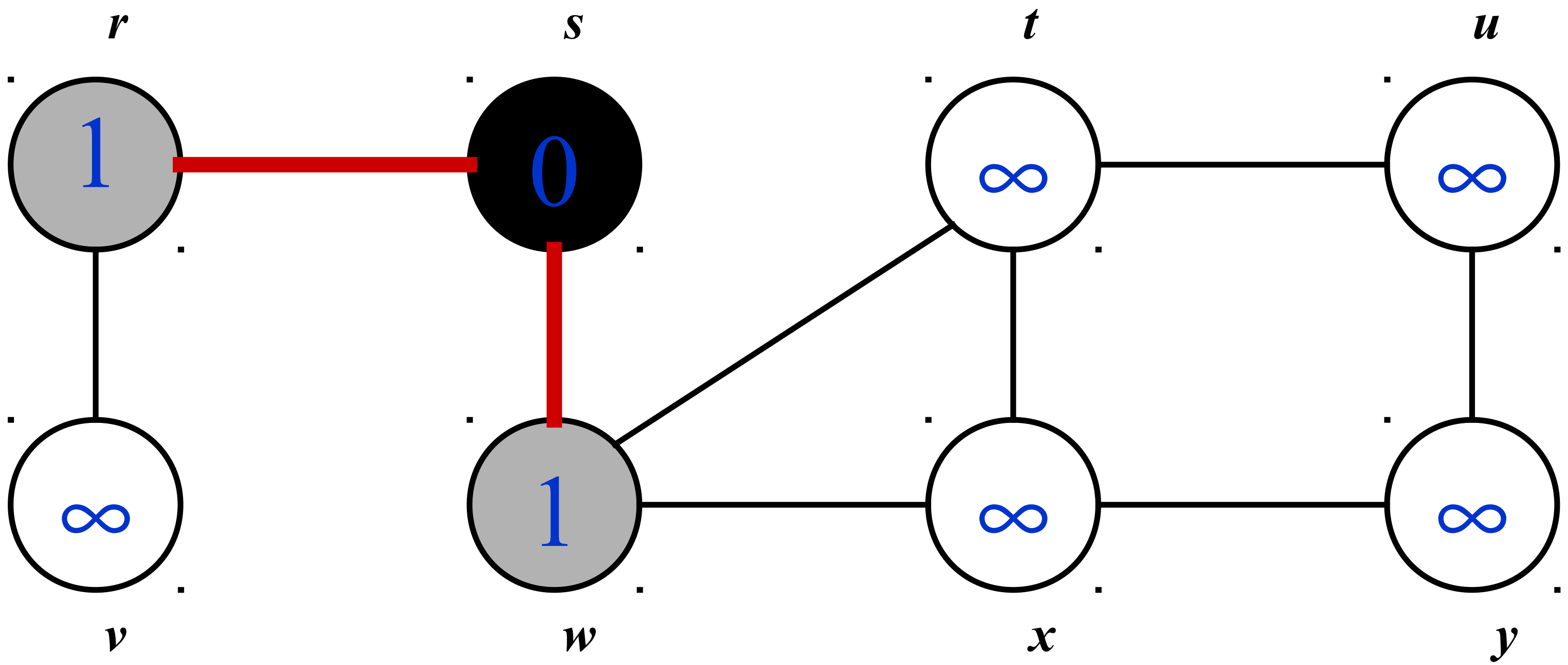


Breadth-First Search: Example



Q: s

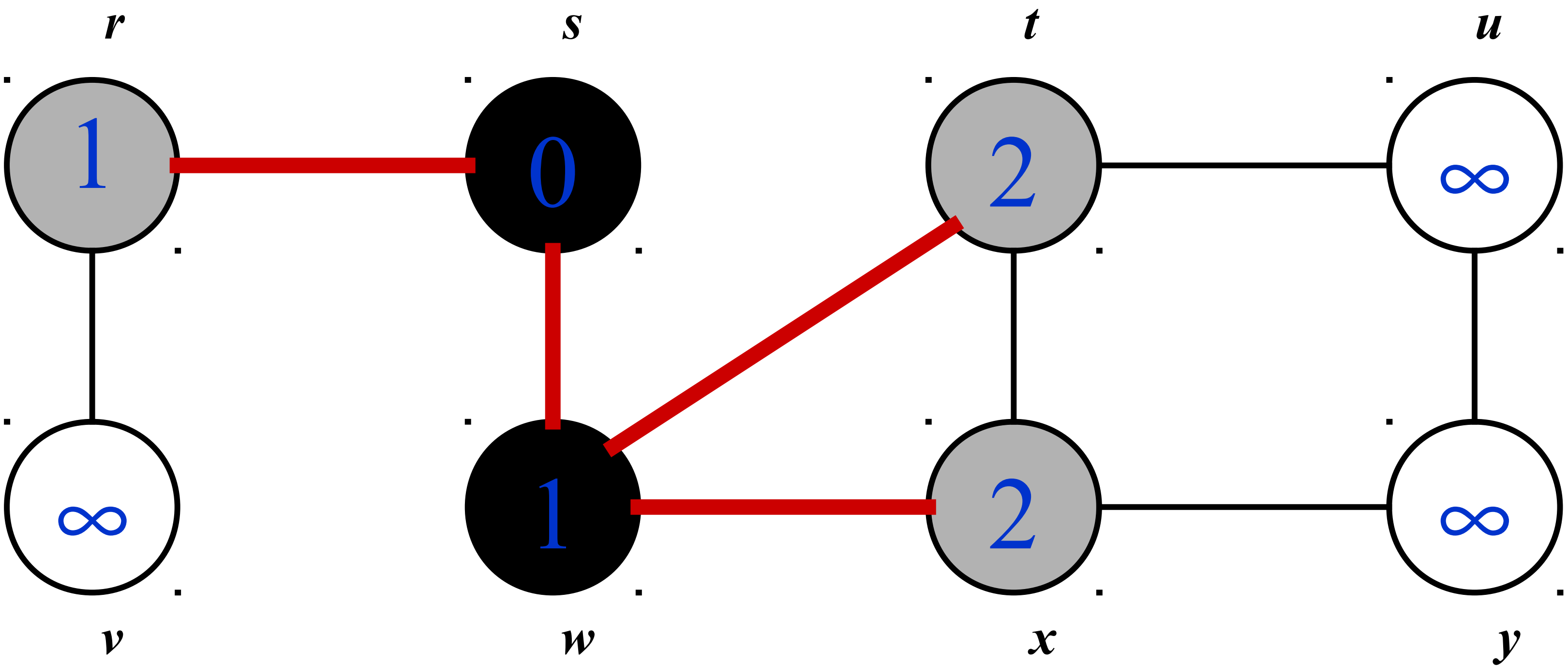
Breadth-First Search: Example



Q:

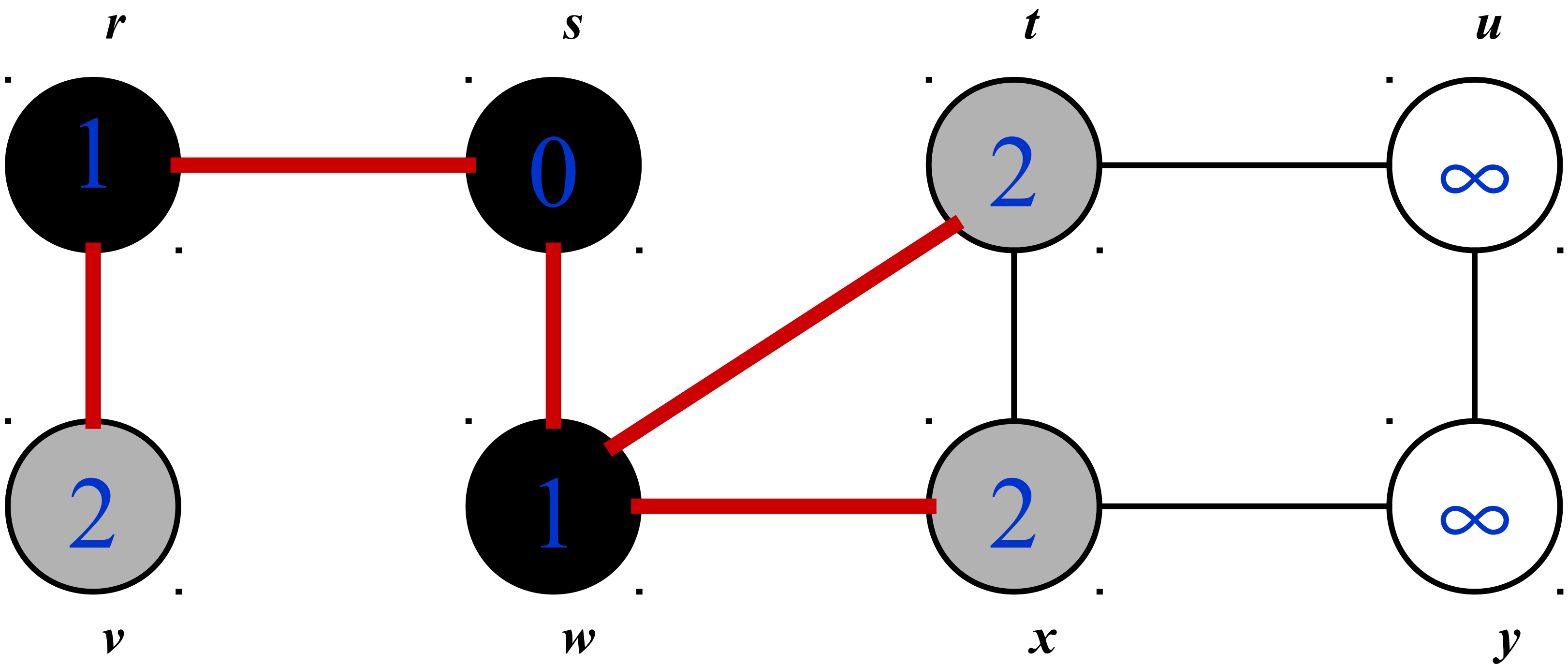
w	*r*

Breadth-First Search: Example



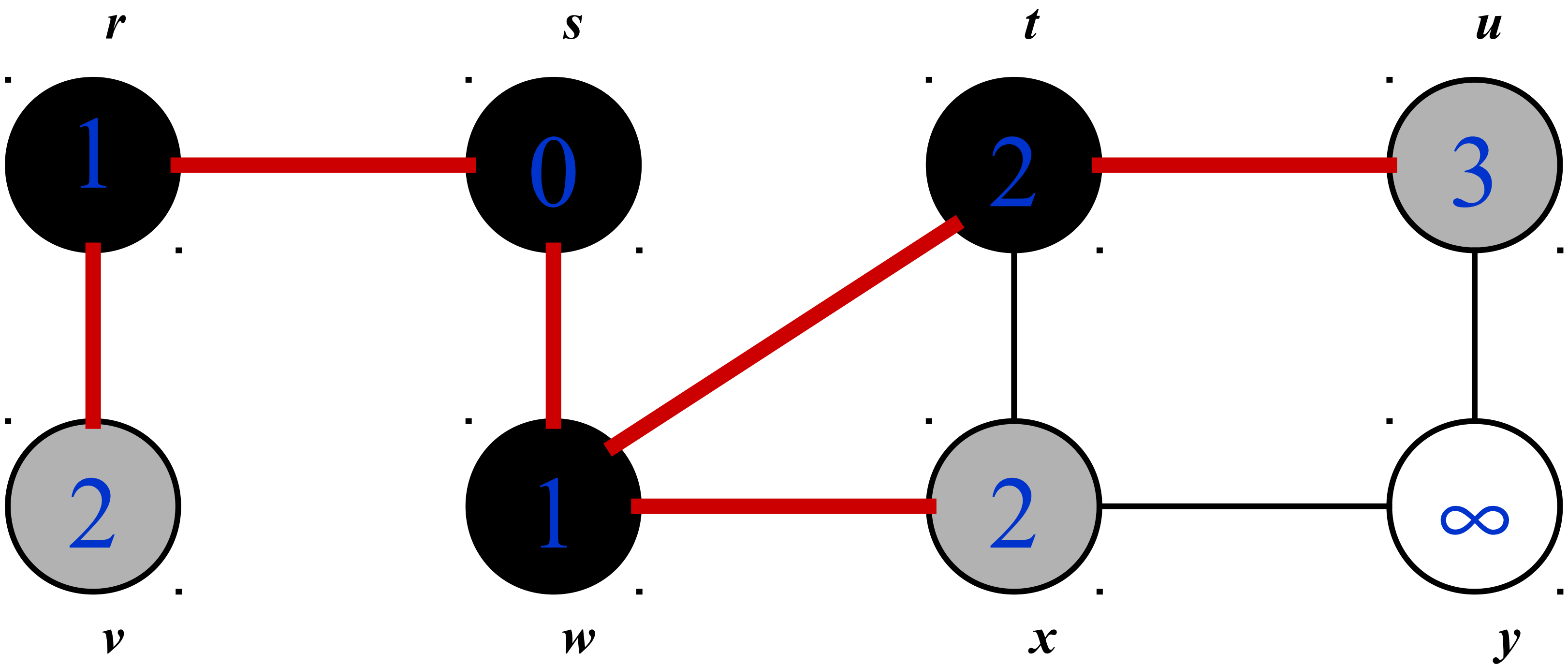
Q:			
	r	*t*	*x*

Breadth-First Search: Example



Q:	t	x	v

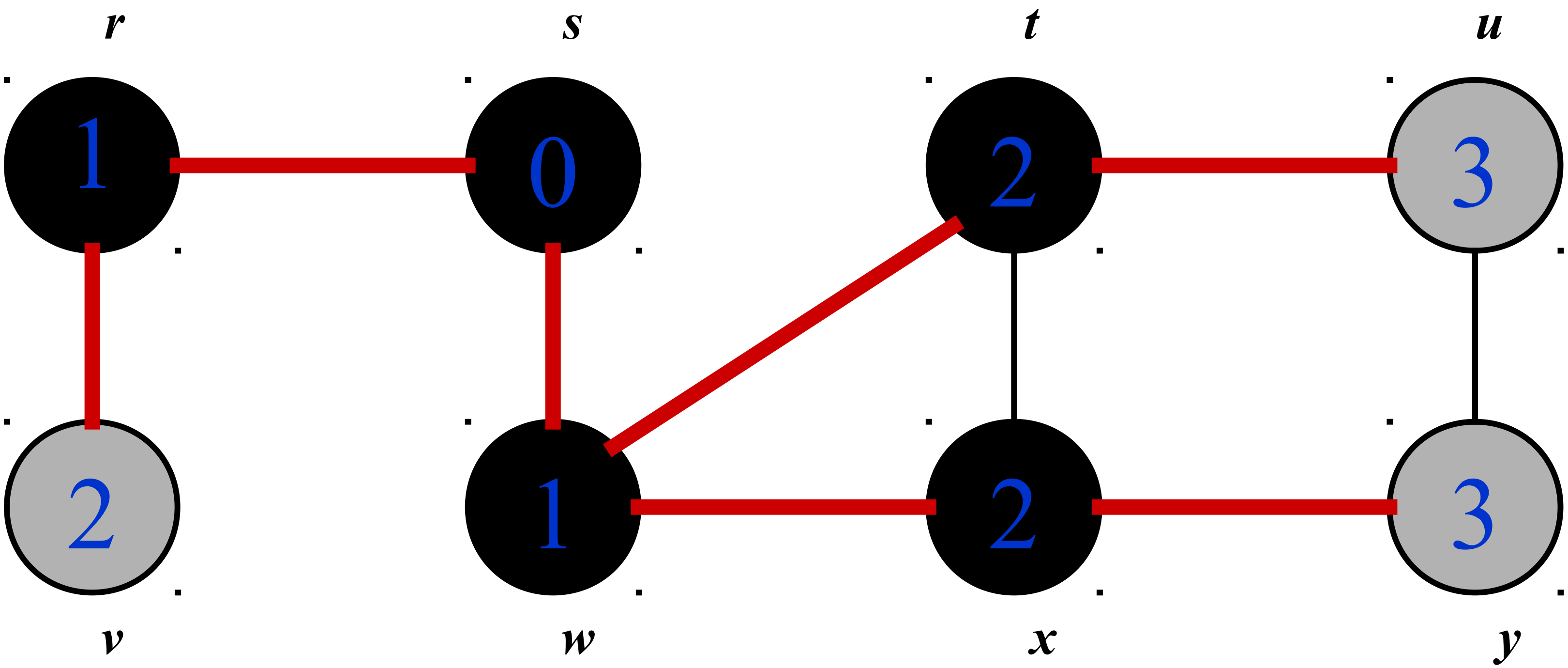
Breadth-First Search: Example



Q:

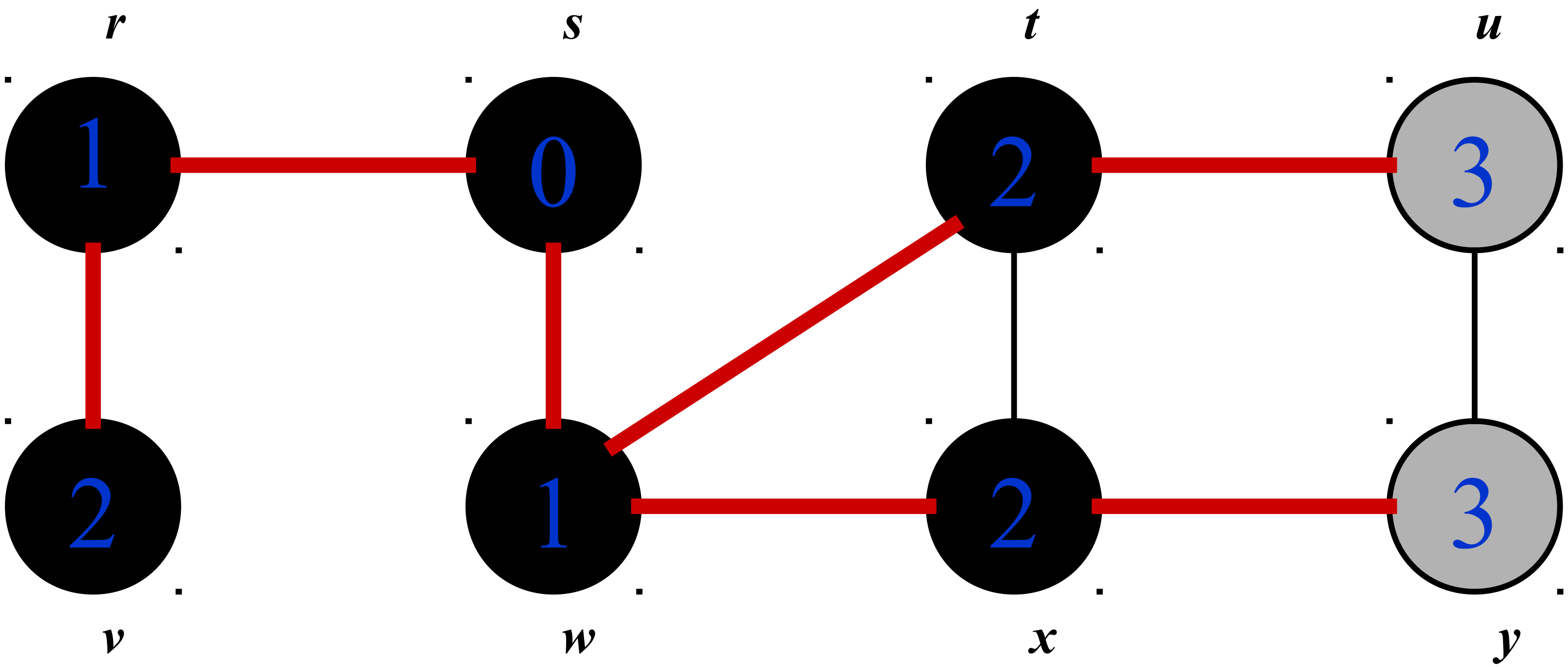
x	*v*	*u*

Breadth-First Search: Example



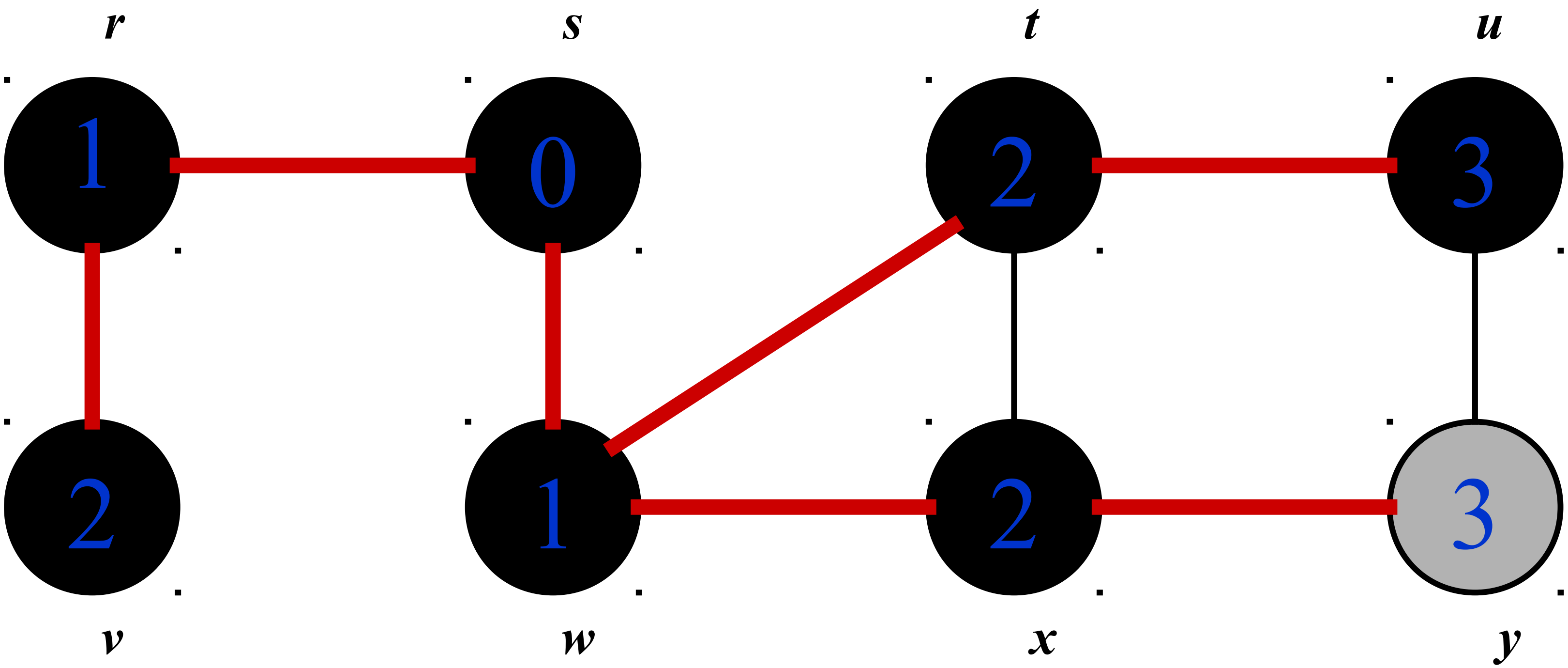
Q:	*v*	*u*	*y*

Breadth-First Search: Example



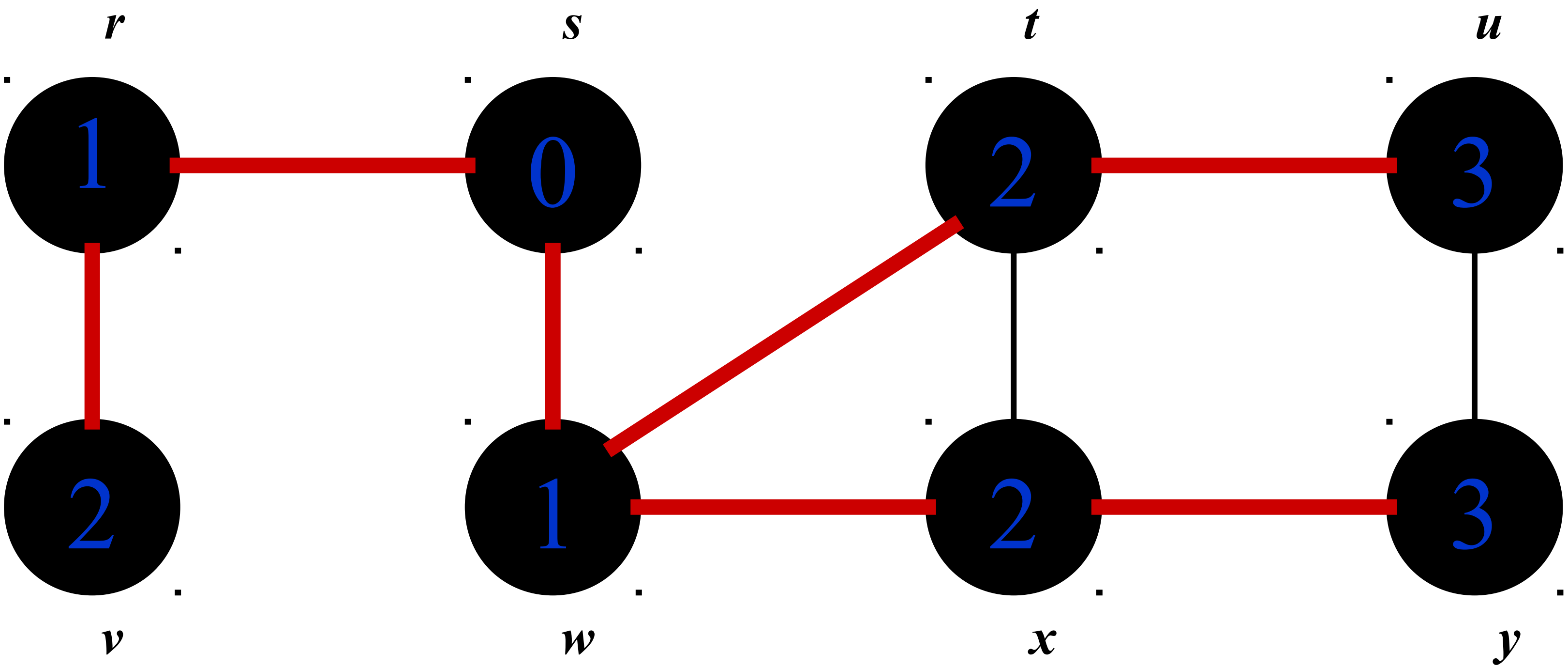
Q: *u* *y*

Breadth-First Search: Example



Q: *y*

Breadth-First Search: Example



Q: **Ø**

BFS: The Code Again

```
BFS(G, s):
    for each v ∈ V:
        v.d = ∞
        v.p = NIL
        v.color = WHITE

    s.d = 0
    s.color = GREY
    Q = {s}                 // Initialize to s

    while (Q not empty):
        u = Dequeue(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?

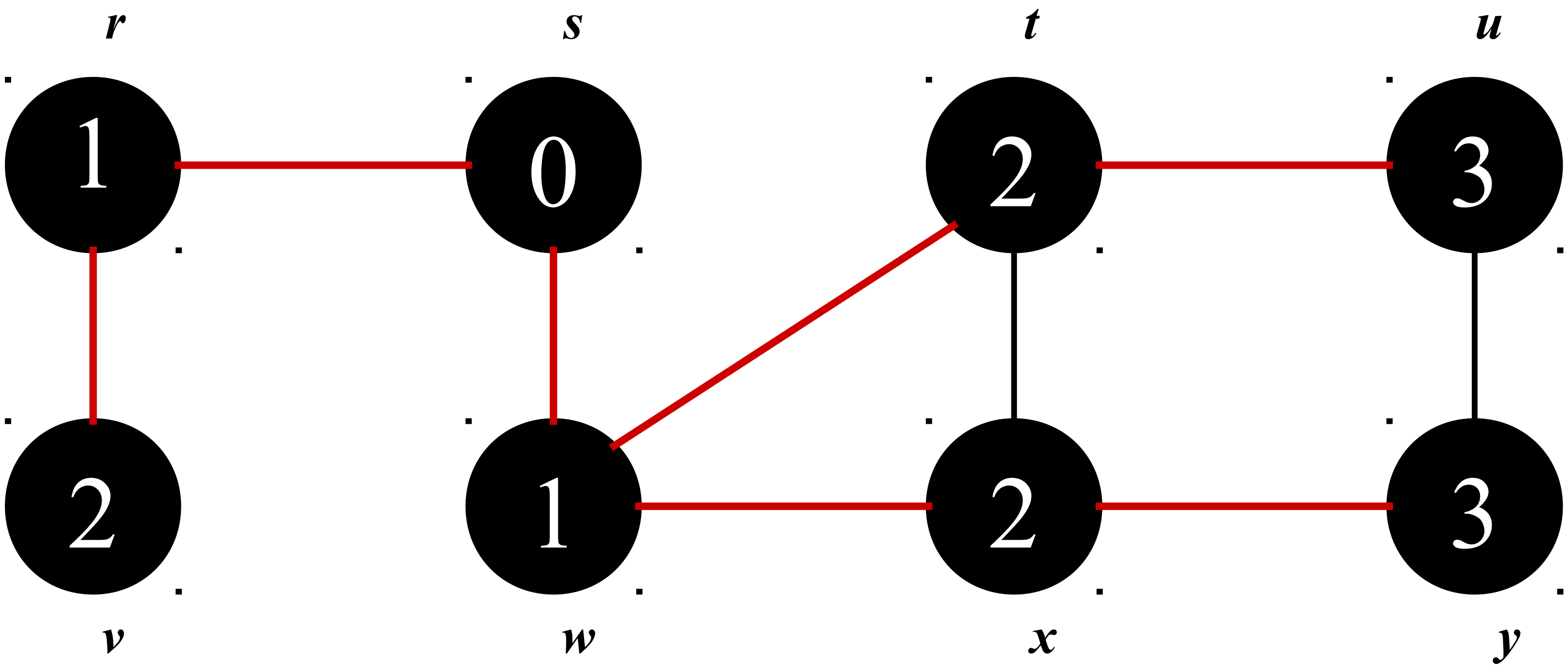
Total running time: $O(V+E)$

Breadth-First Search: Properties

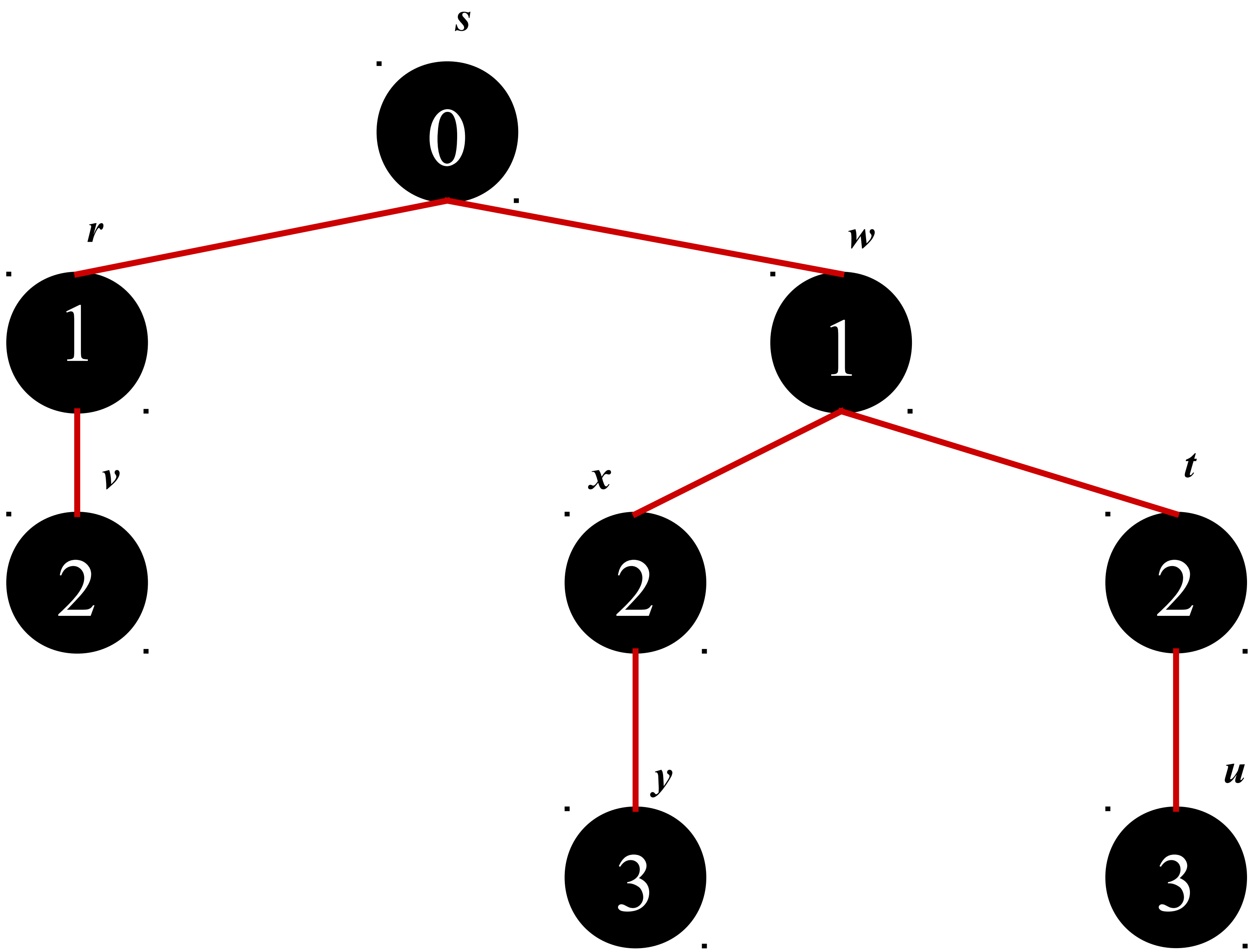
- BFS calculates the *shortest-path distance* to the source node
  - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or ∞ if v not reachable from s
  - Proof. **CLRS** Ch. 22.2
  - Will generalize later for *weighted* graphs

- 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

Breadth-First Tree



Breadth-First Tree



Depth-First Search (DFS)

- *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
- Use *colors* for exploring the graph
  - Vertices initially colored *white*
  - Then colored *gray* when discovered
  - Then *black* when finished

Depth-First Search: The Code

```
DFS(G)
   for each vertex u ∈ 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)
         v.p = u
         DFS_Visit(v)
   u.color = BLACK
   time = time+1
   u.f = time
```

What is *u.d* ?
It records the *discovery* of vertex *u*

What is *u.f* ?
It records the *finish* of processing vertex *u*

What is *u.p* ?
It records the *parent* of vertex *u*

Depth-First Search: The Code

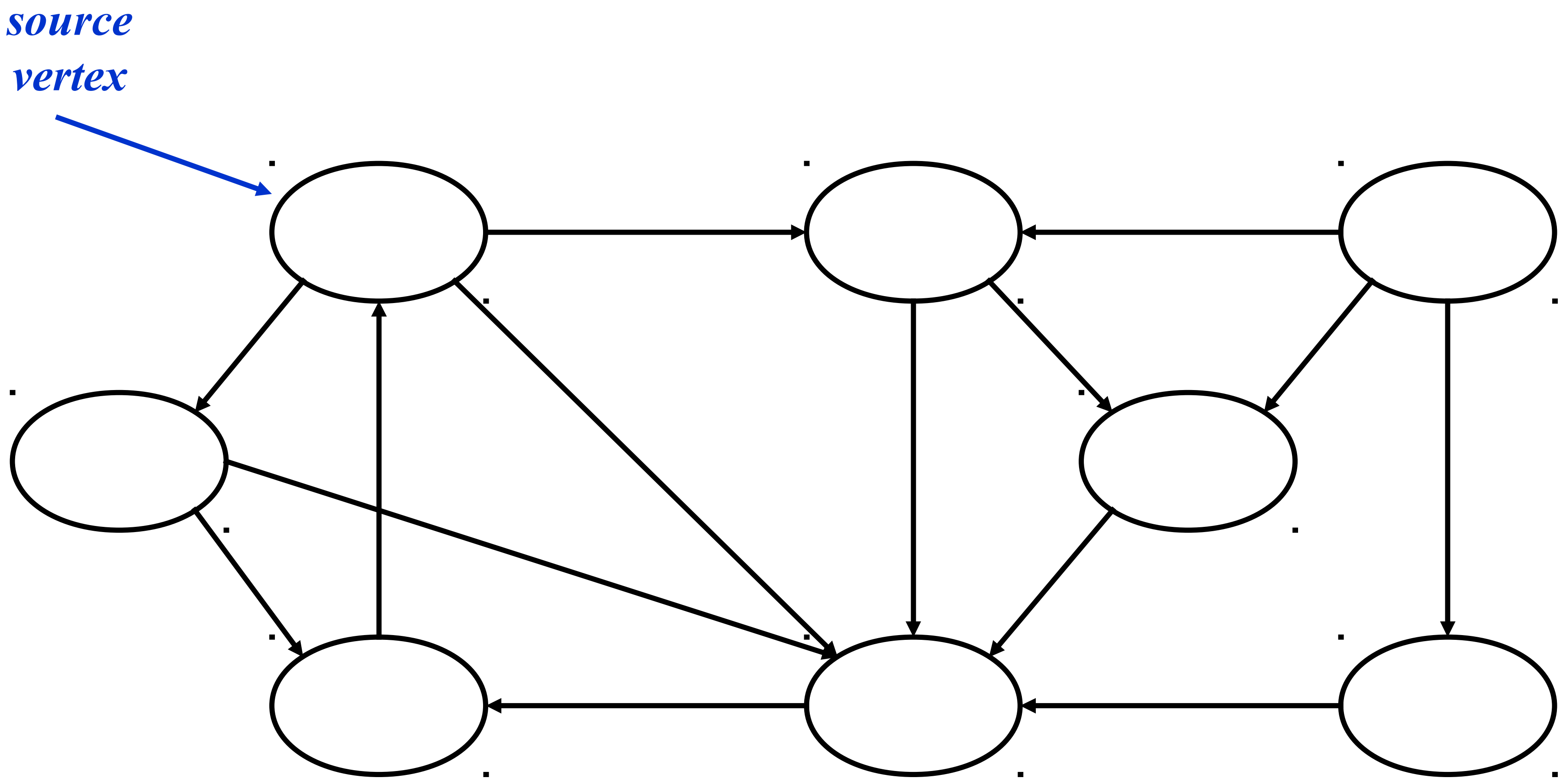
```
DFS(G)
   for each vertex u ∈ V
      u.color = WHITE
   time = 0
   for each vertex u ∈ G.V
      if (u.color == WHITE)
         DFS-Visit(u)
```

```
DFS-Visit(u)
   u.color = GREY
   time = time+1
   u.d = time
   for each v ∈ u.Adj[]
      if (v.color == WHITE)
         v.p = u
         DFS_Visit(v)
   u.color = BLACK
   time = time+1
   u.f = time
```

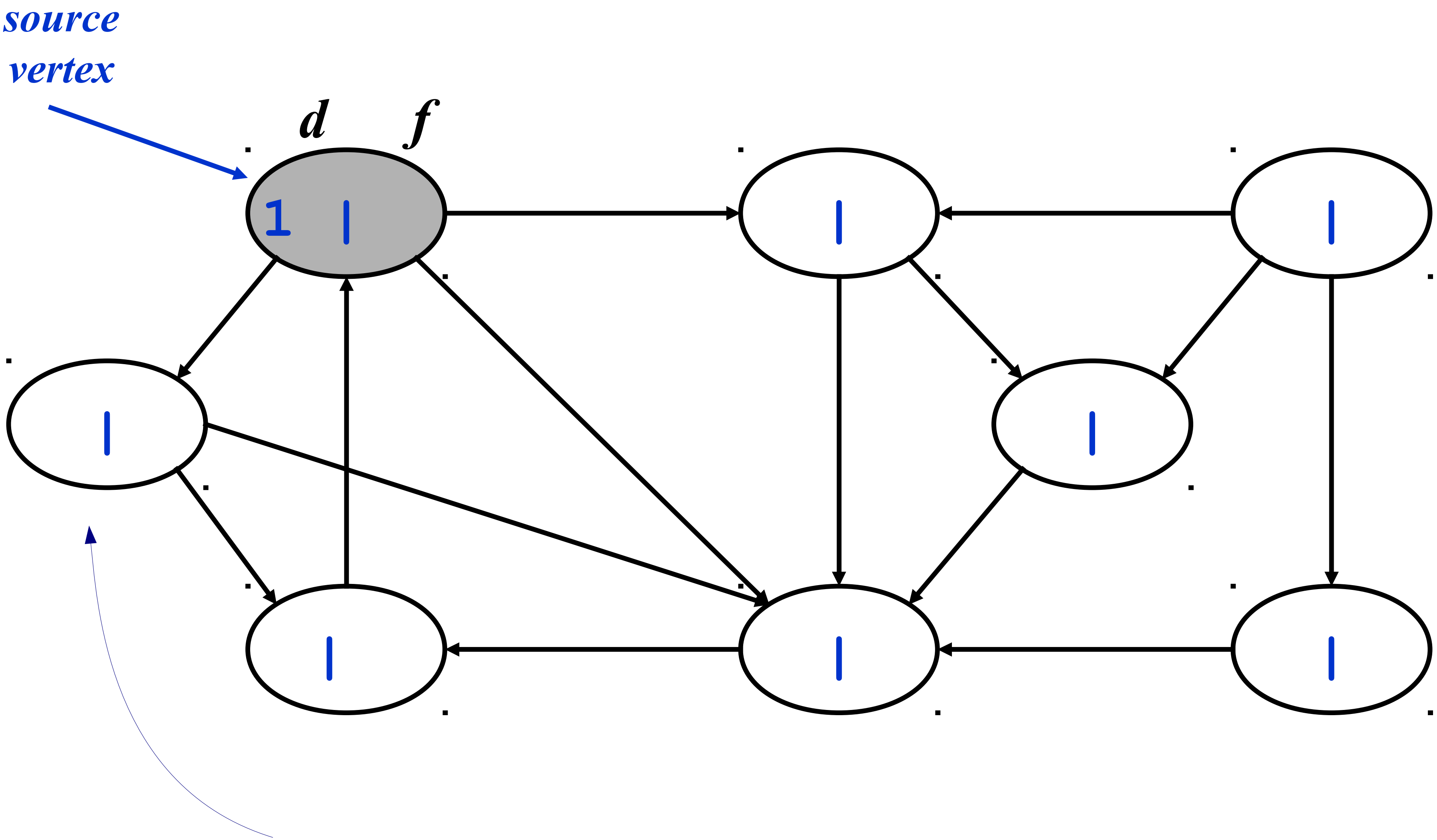
Will all vertices be colored BLACK eventually?
Yes!

What if G is *not* connected?
DFS forest!

DFS Example

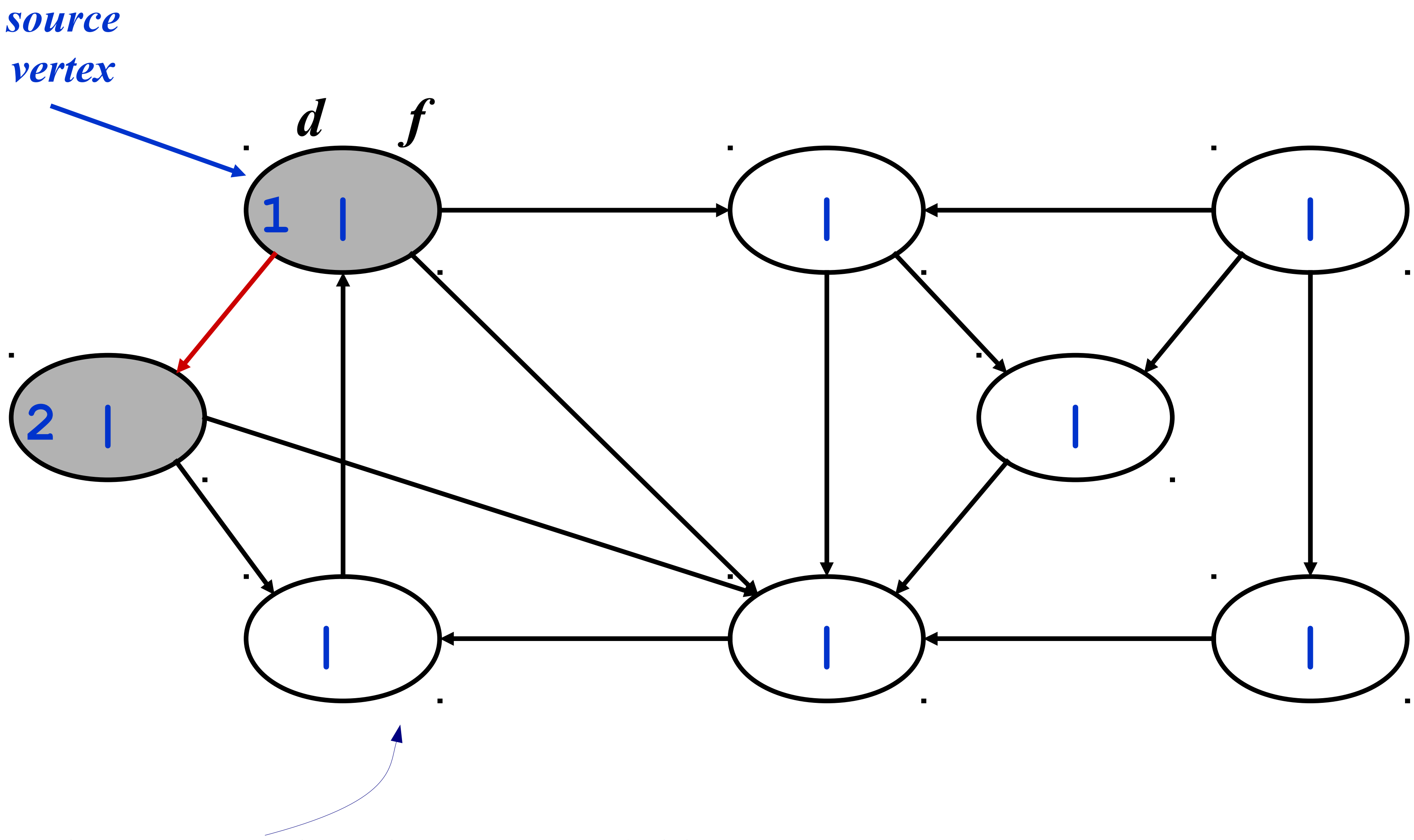


DFS Example



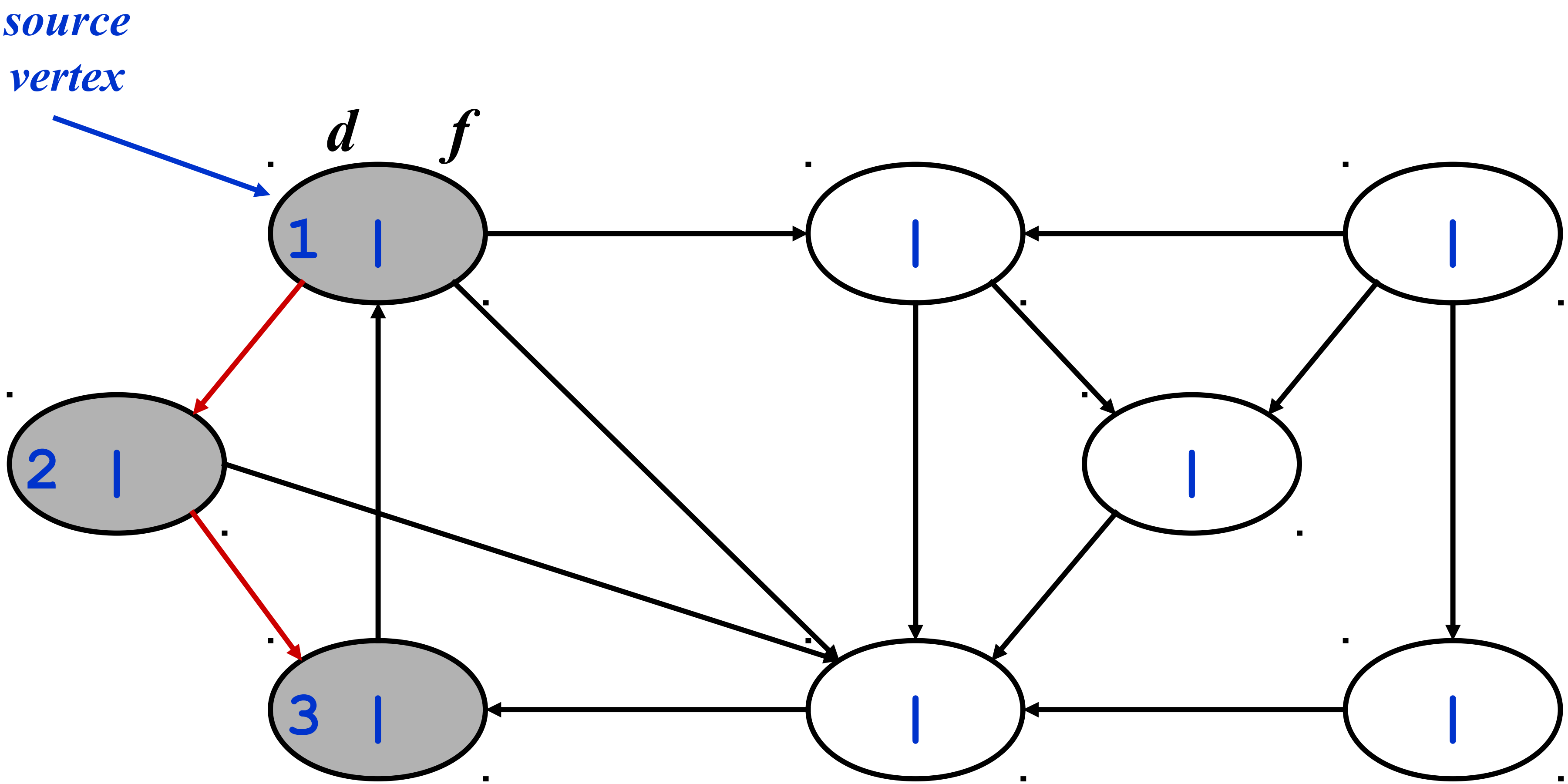
Mark as *grey* and explore *white* neighbors

DFS Example



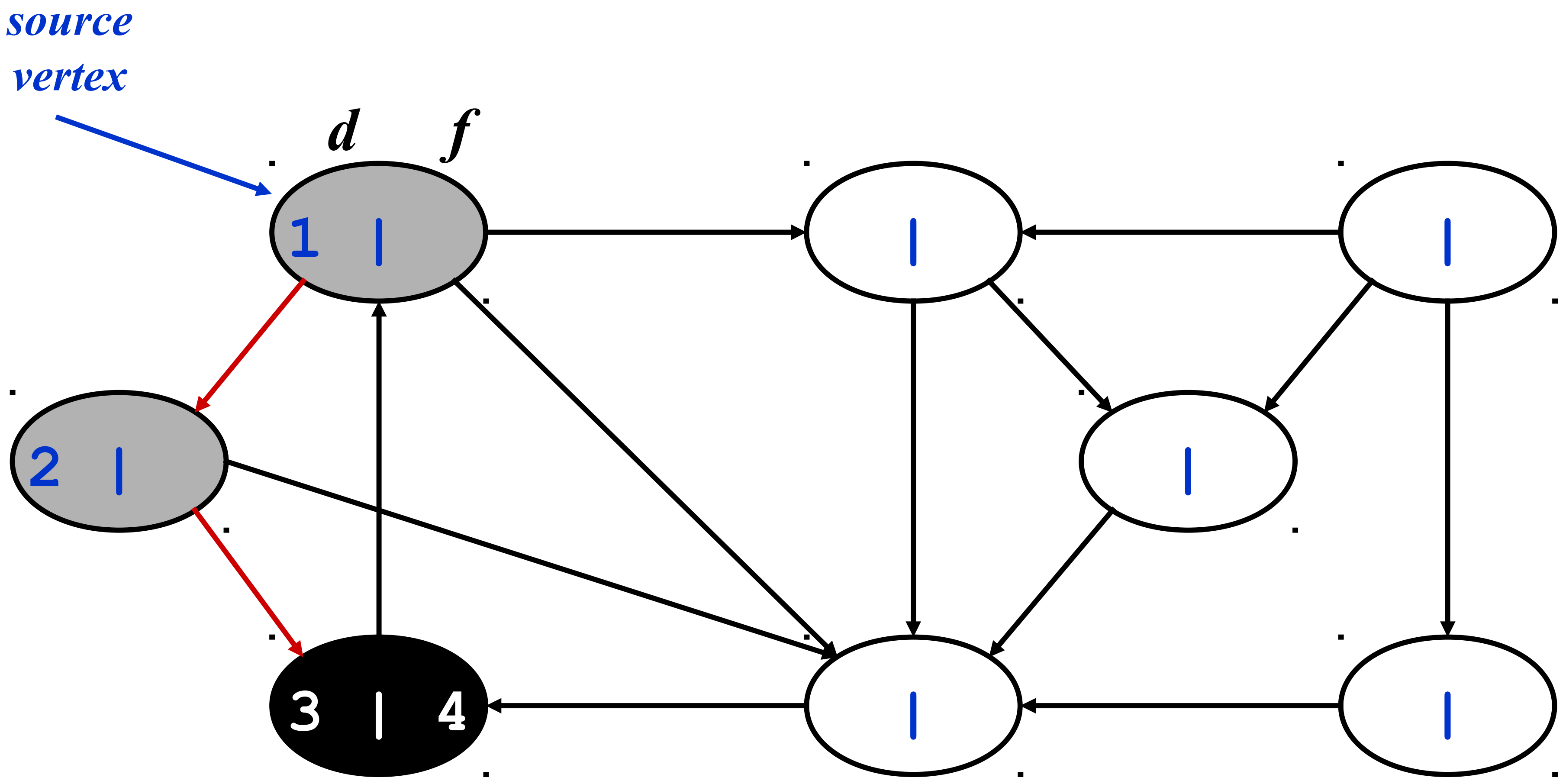
Mark as *grey* and explore *white* neighbors

DFS Example



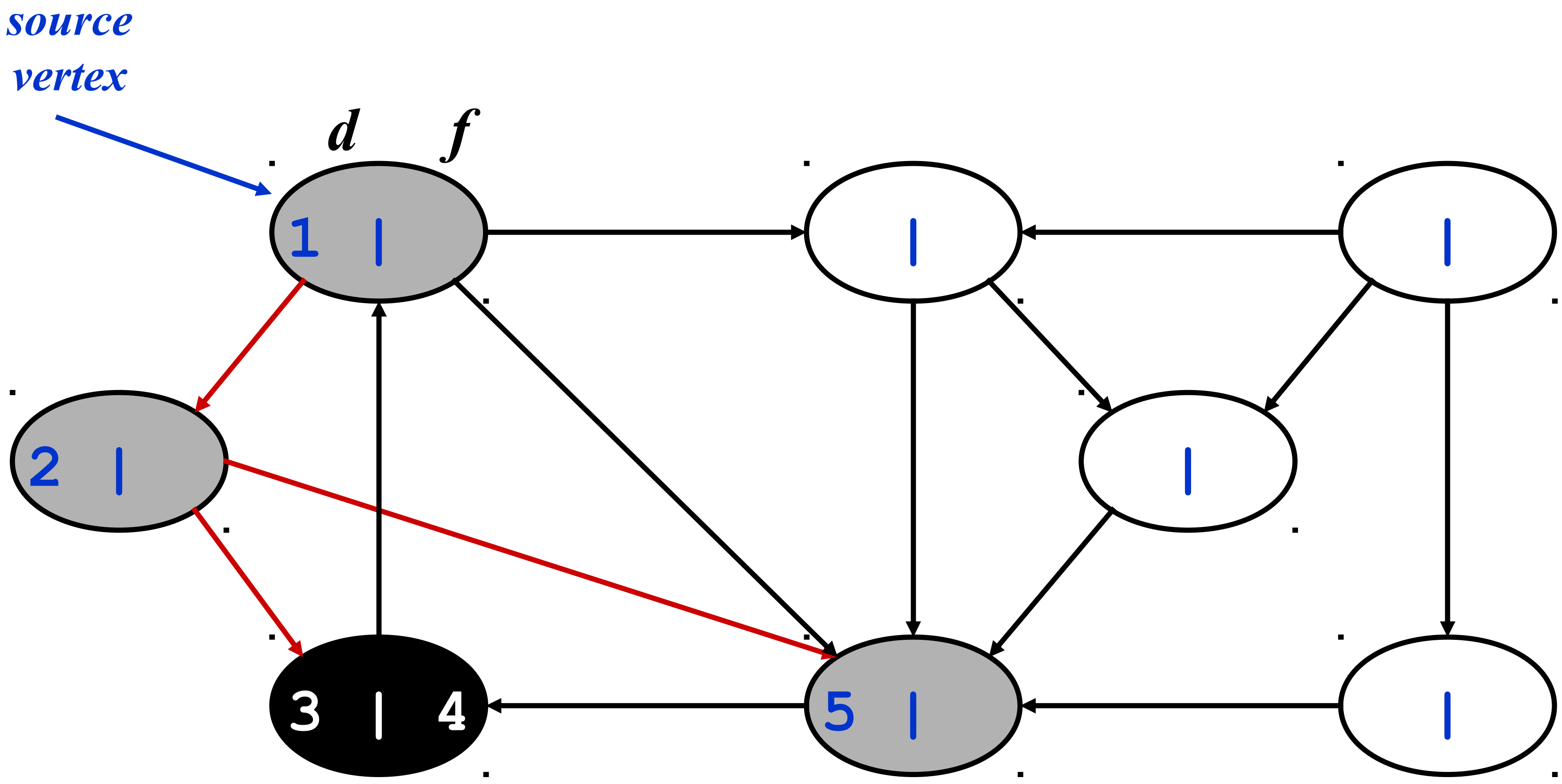
No more *white* neighbors

DFS Example

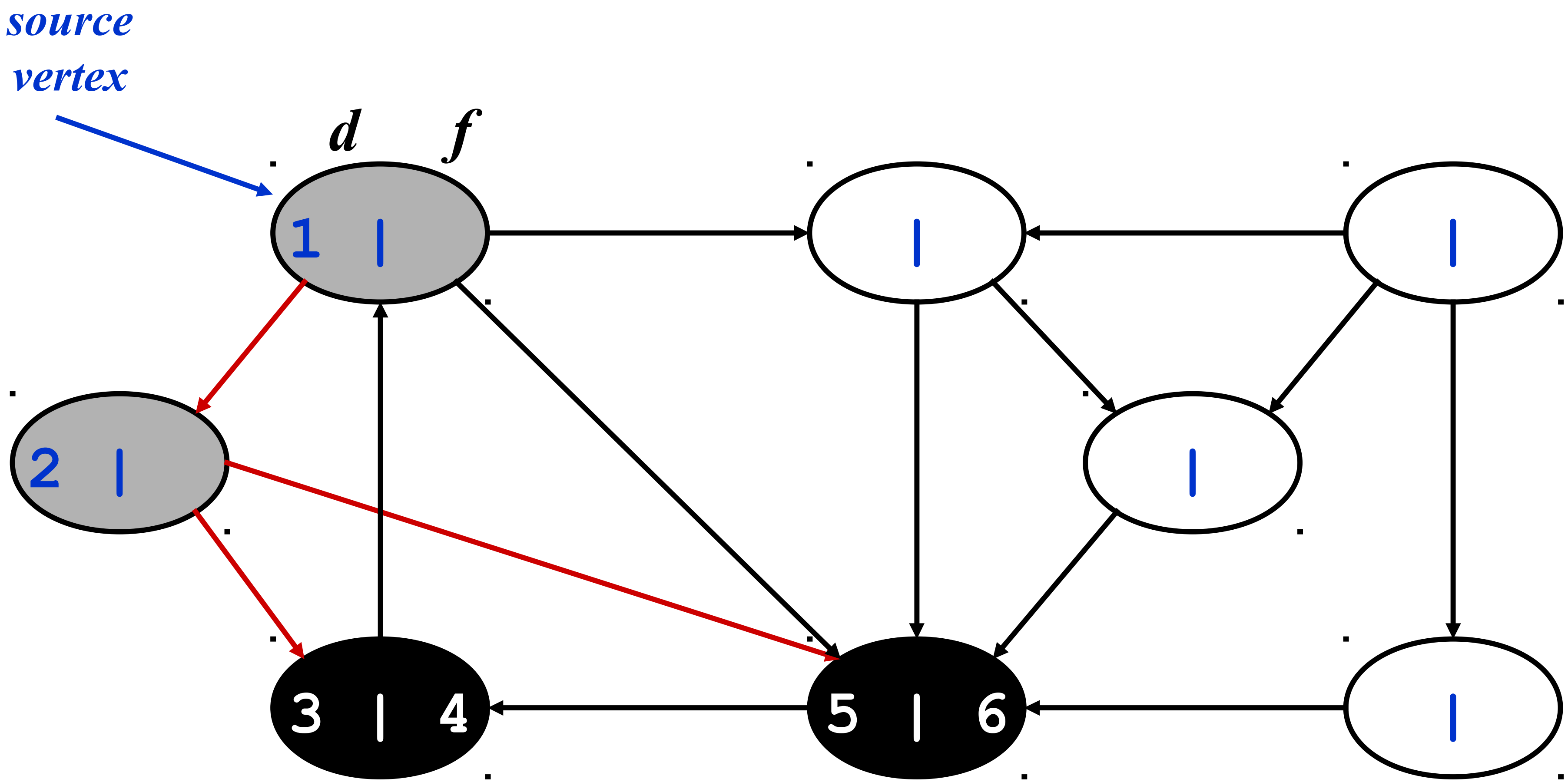


Mark as *black* and backtrack

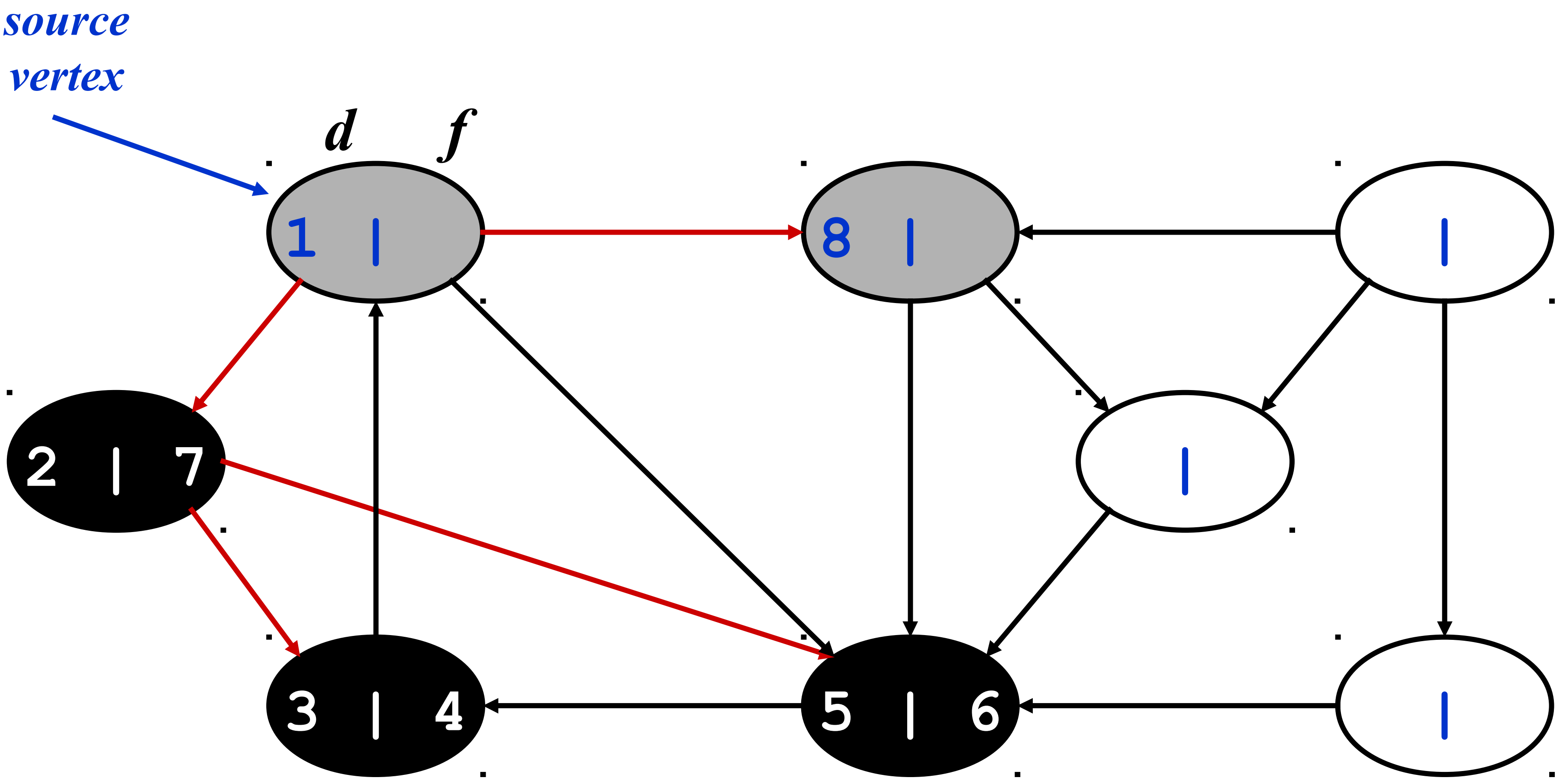
DFS Example



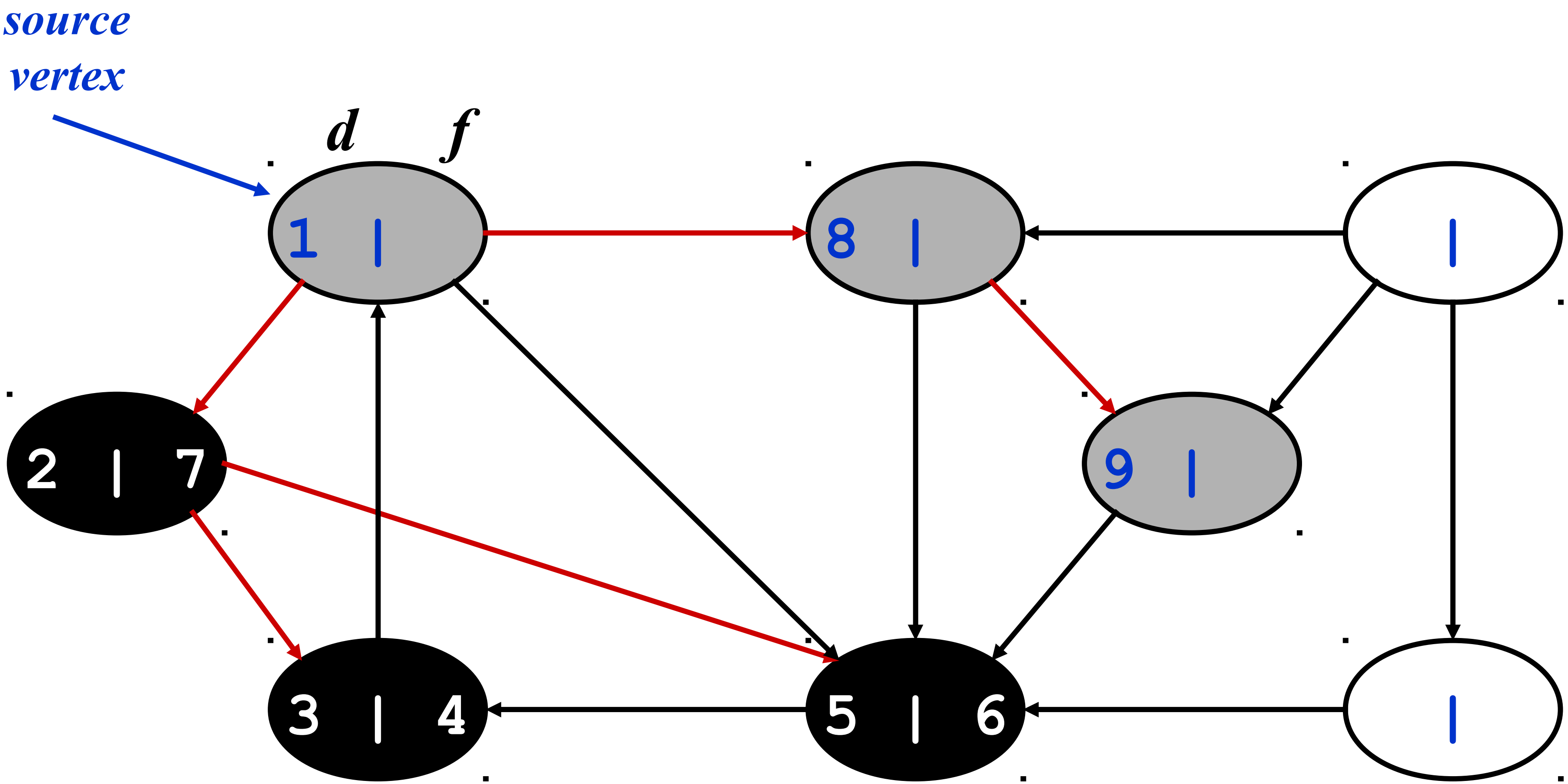
DFS Example



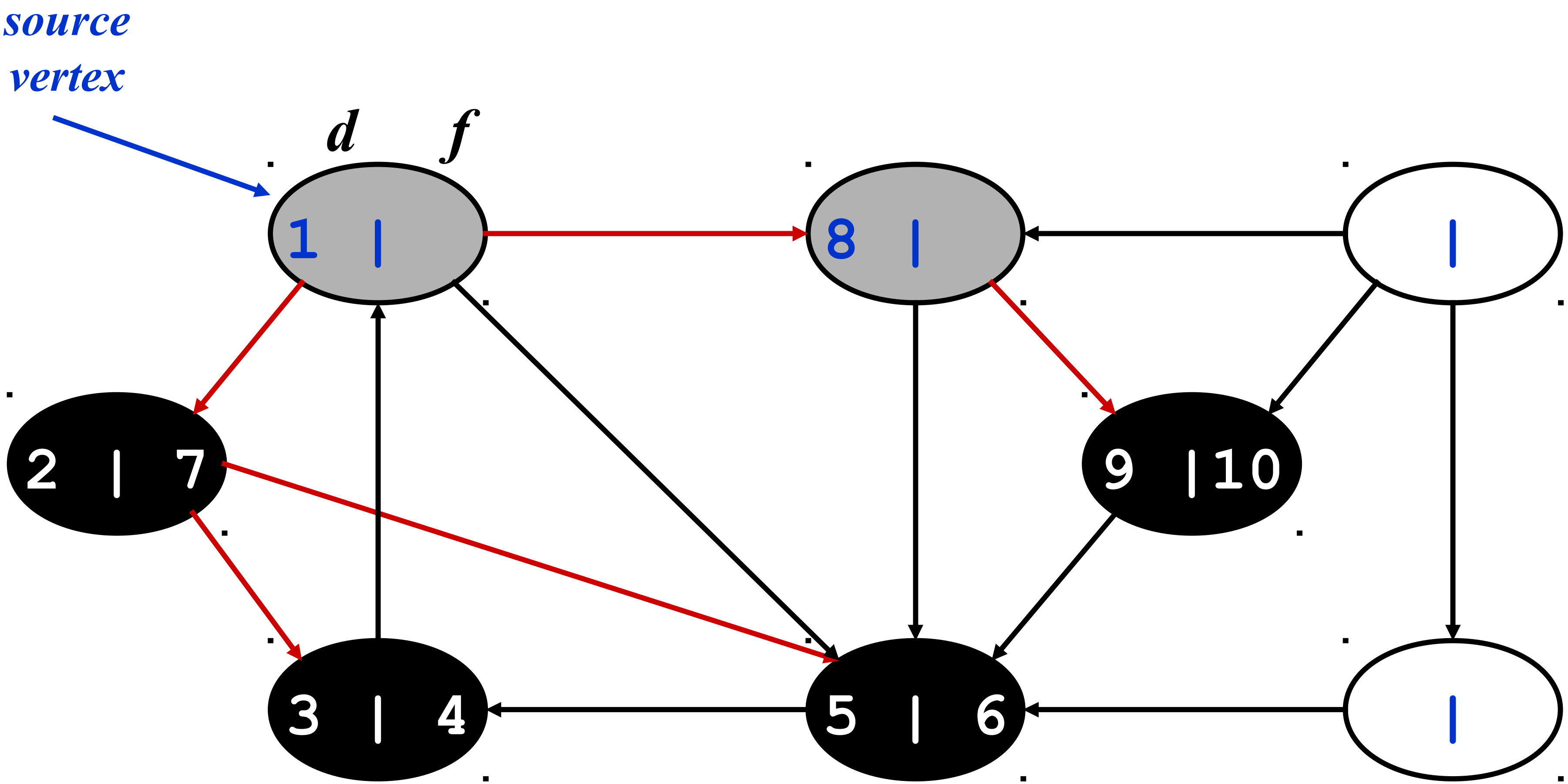
DFS Example



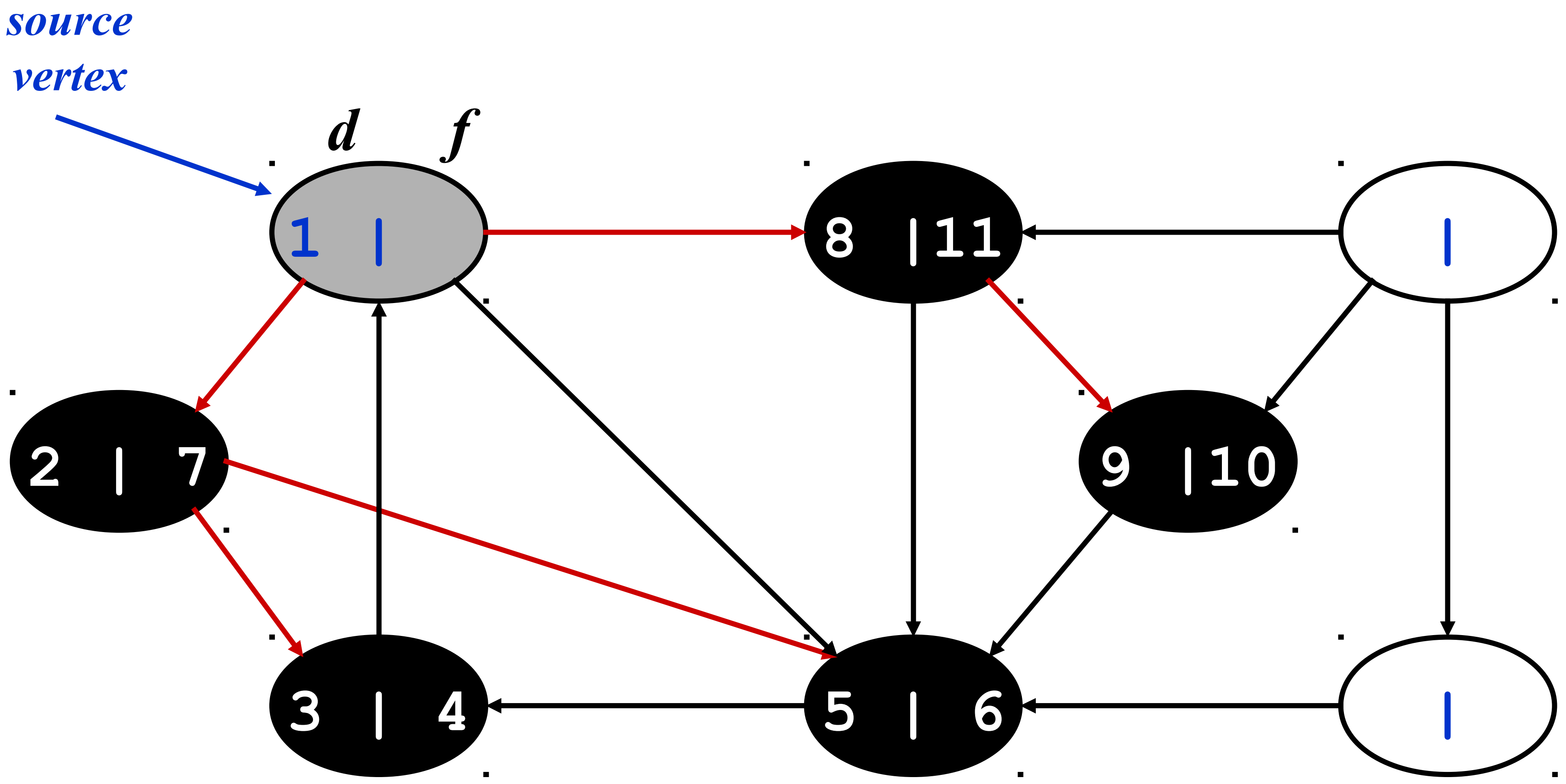
DFS Example



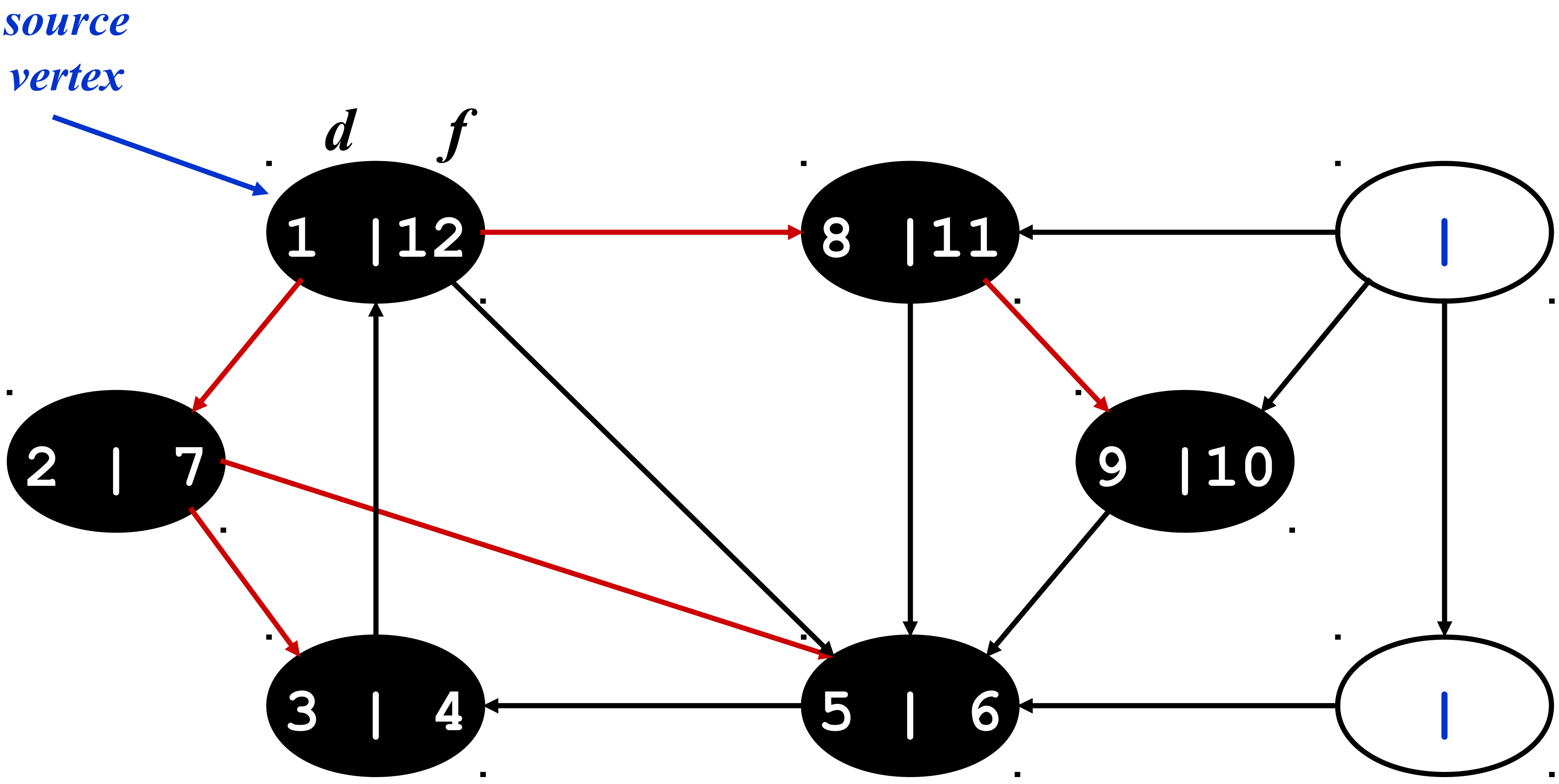
DFS Example



DFS Example

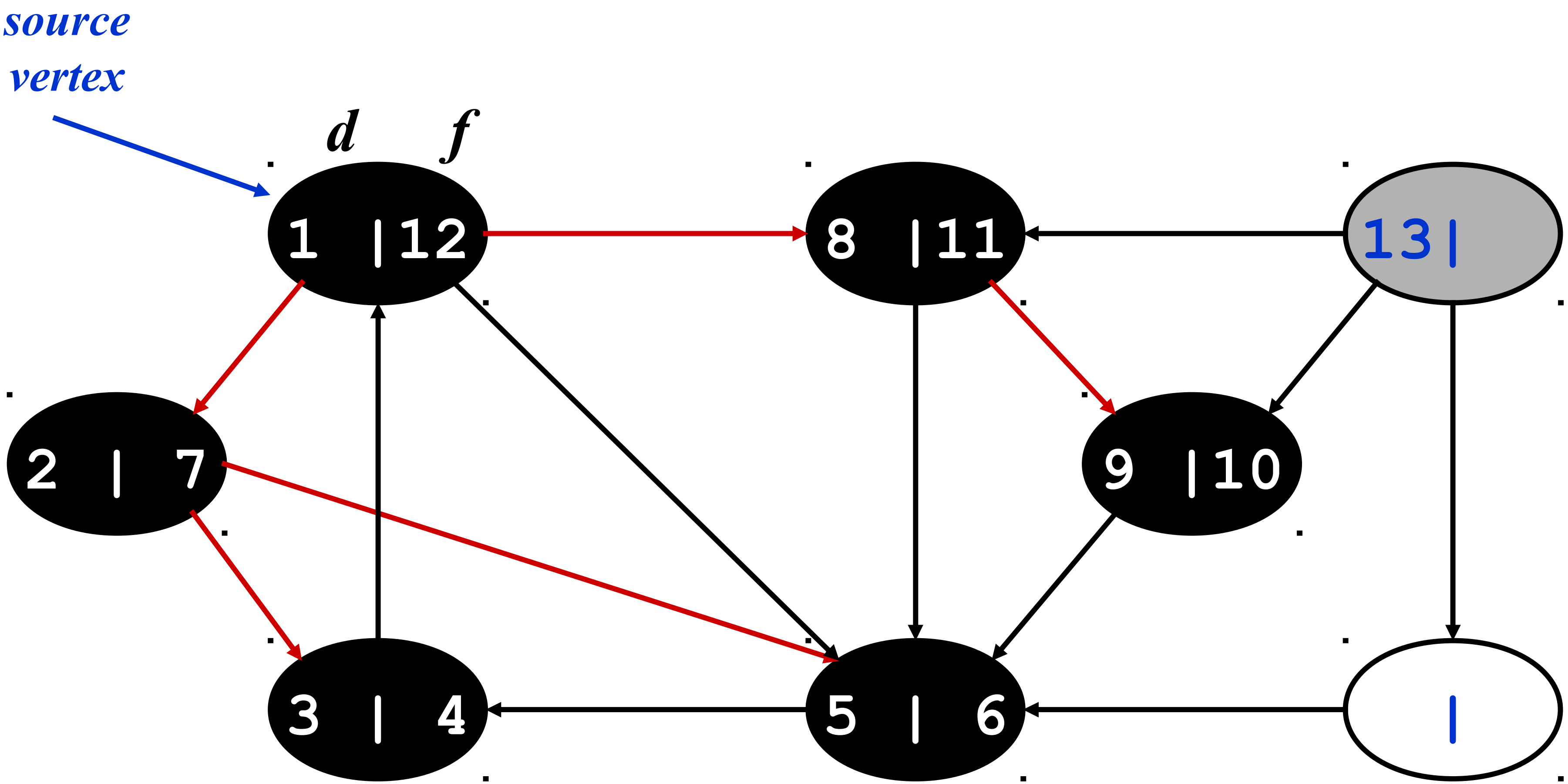


DFS Example

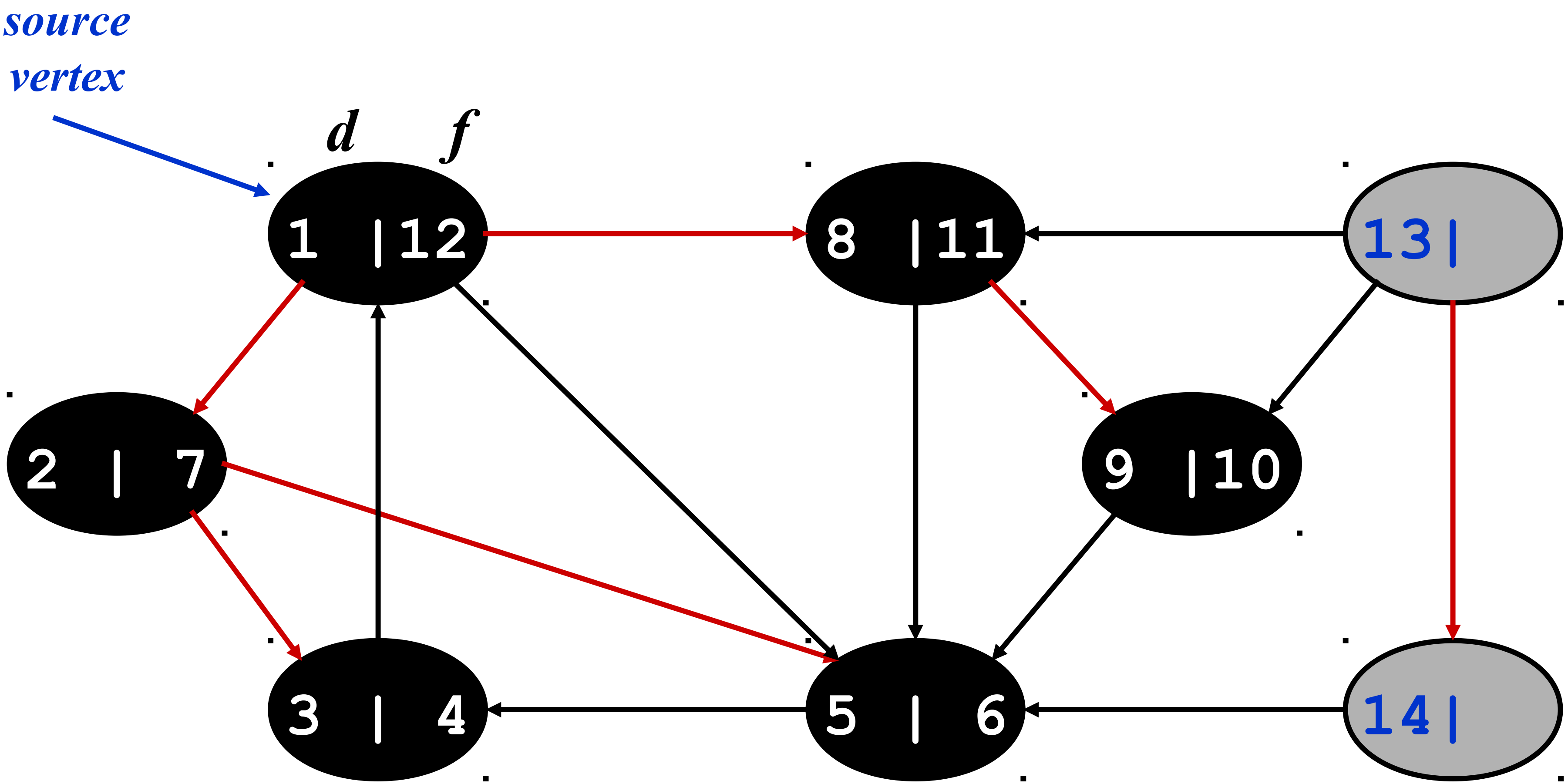


Nothing more to explore from the *source vertex*, go to another *component*

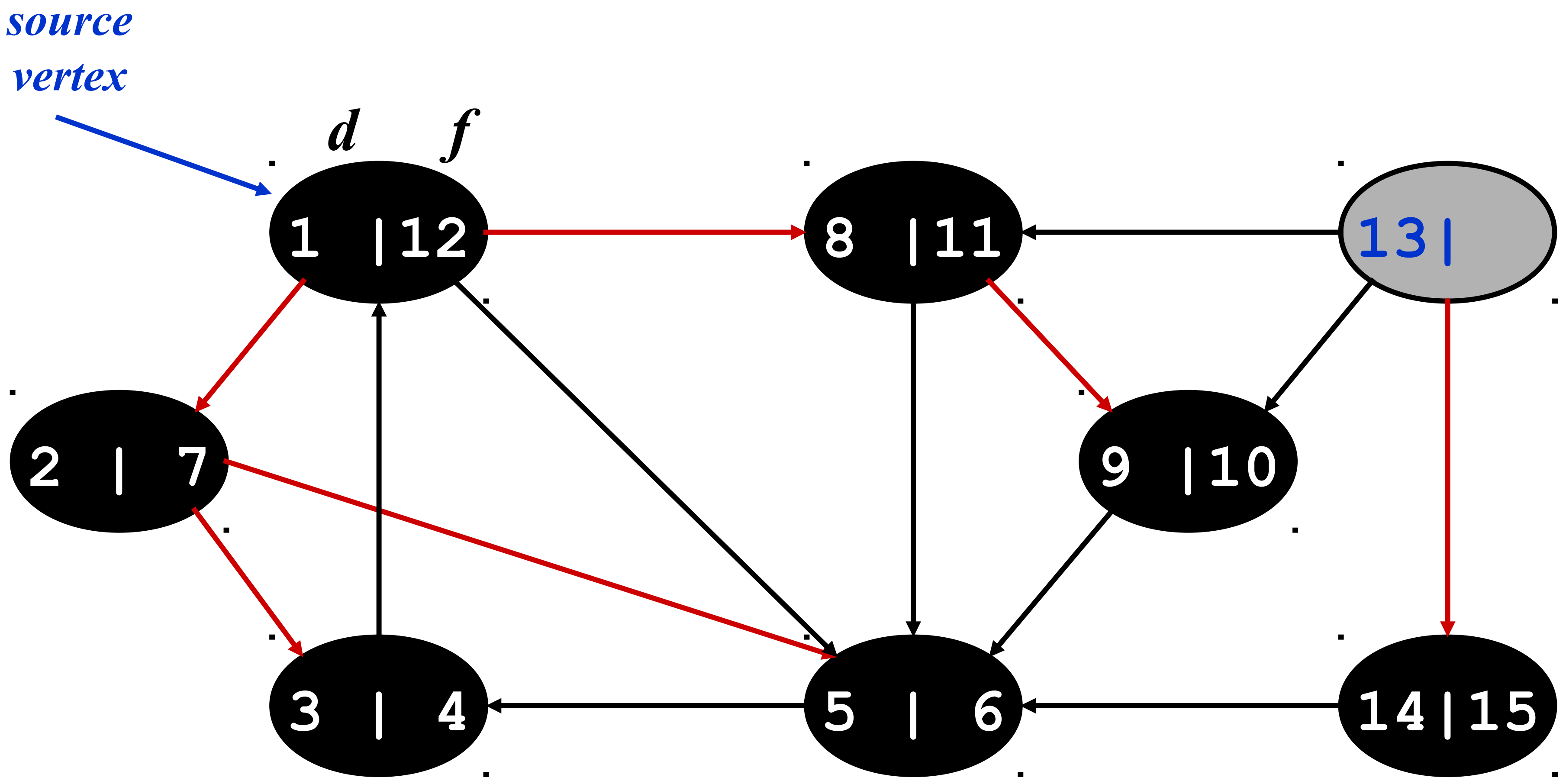
DFS Example



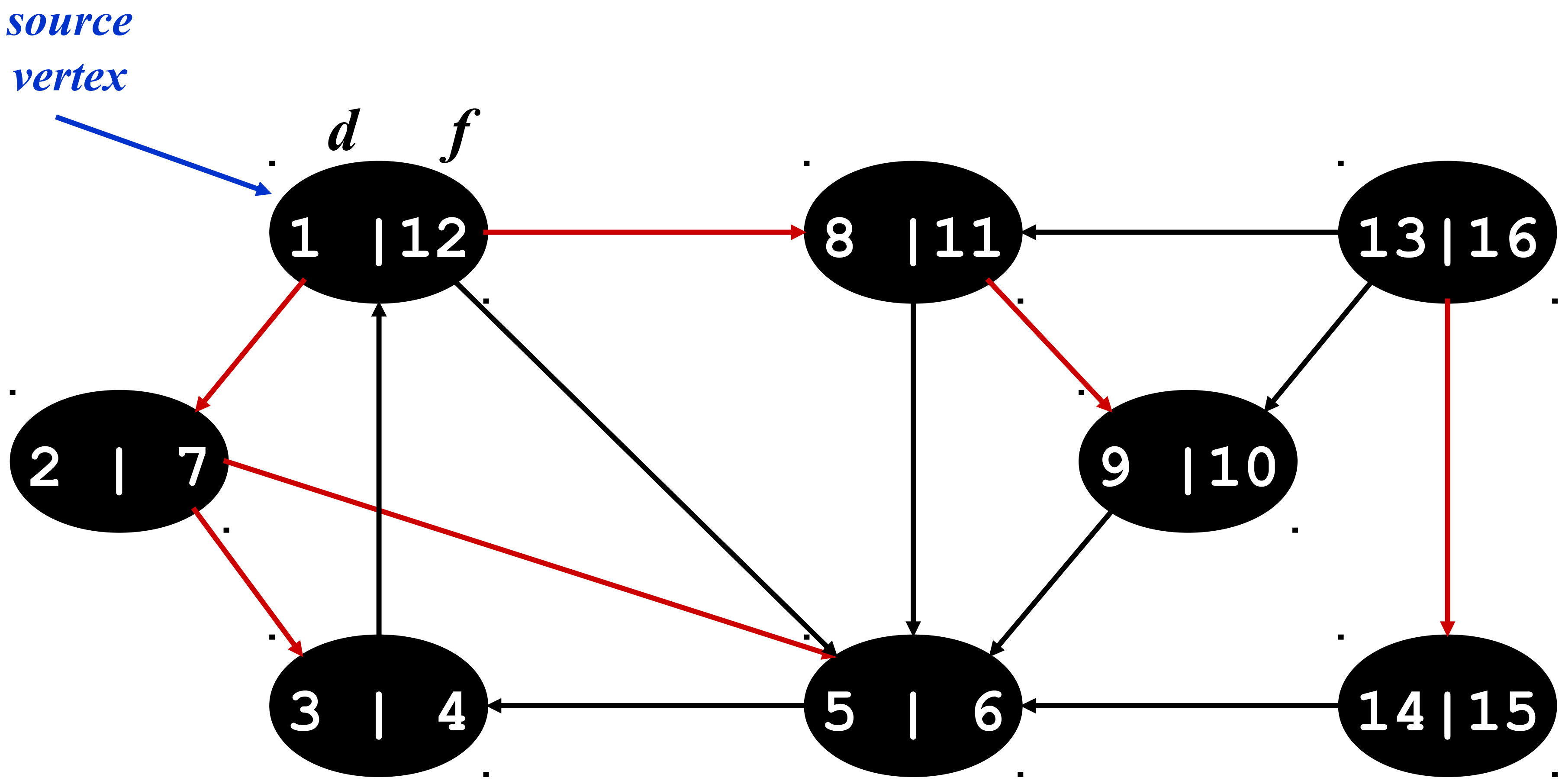
DFS Example



DFS Example



DFS Example



DFS Analysis

```
DFS(G)
    for each vertex u ∈ 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)
            v.p = u
            DFS-Visit(v)
    u.color = BLACK
    time = time+1
    u.f = time
```

What is the running time?

$\Theta(V)$

$\Theta(|u.Adj|)$

How many times is DFS-Visit called for every vertex? Exactly *once*! *Why*?

How many times total is this loop executed?

$$\sum_{v\in V} |v.Adj| = \Theta(E)$$

Total time = $\Theta(E+V)$

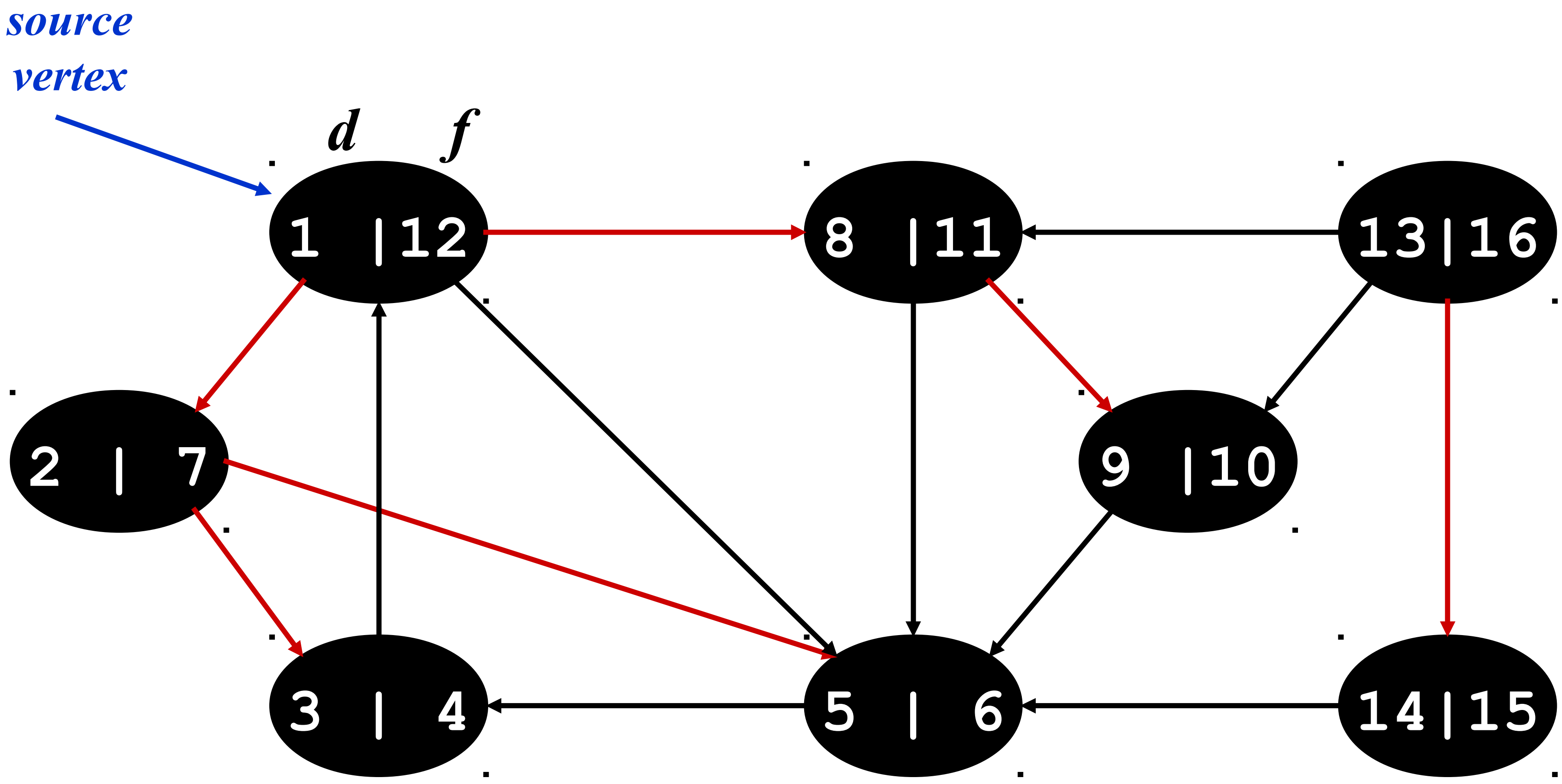
Depth-First Sort Analysis

- This running time argument is an informal example of *amortized analysis*
  - “Charge” the exploration of edge to the edge:
    - Each loop iteration in DFS-Visit can be attributed to an edge in the graph
    - Runs once per edge if directed graph, twice if undirected
    - Thus loop will run in $O(E)$ time, algorithm $O(V+E)$
    - Considered linear for graph, because adjacency list requires $O(V+E)$ storage

DFS: Kinds of edges

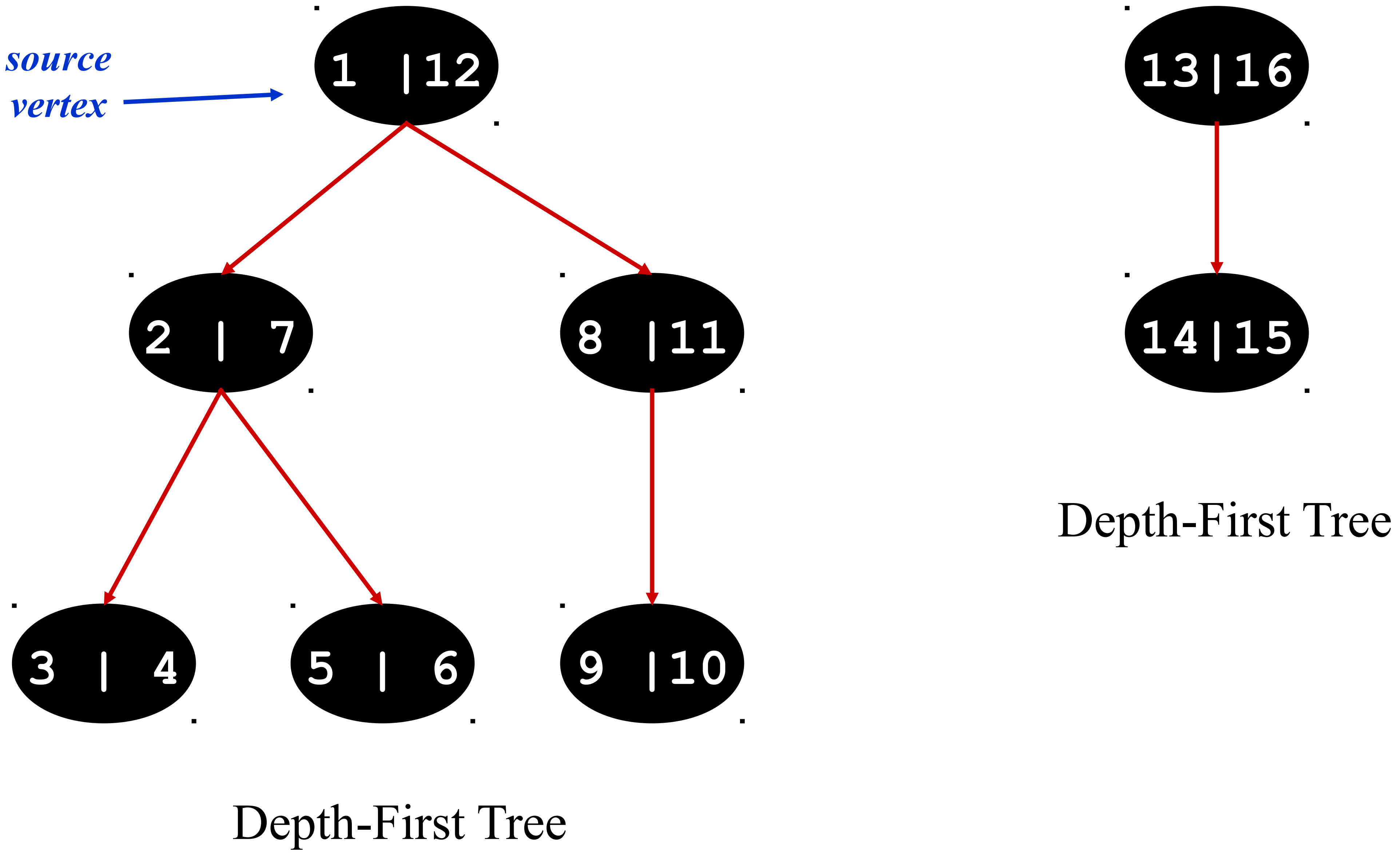
- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
    - The tree edges form a spanning forest
    - Can they form a *cycle*?

DFS Example



Tree edges

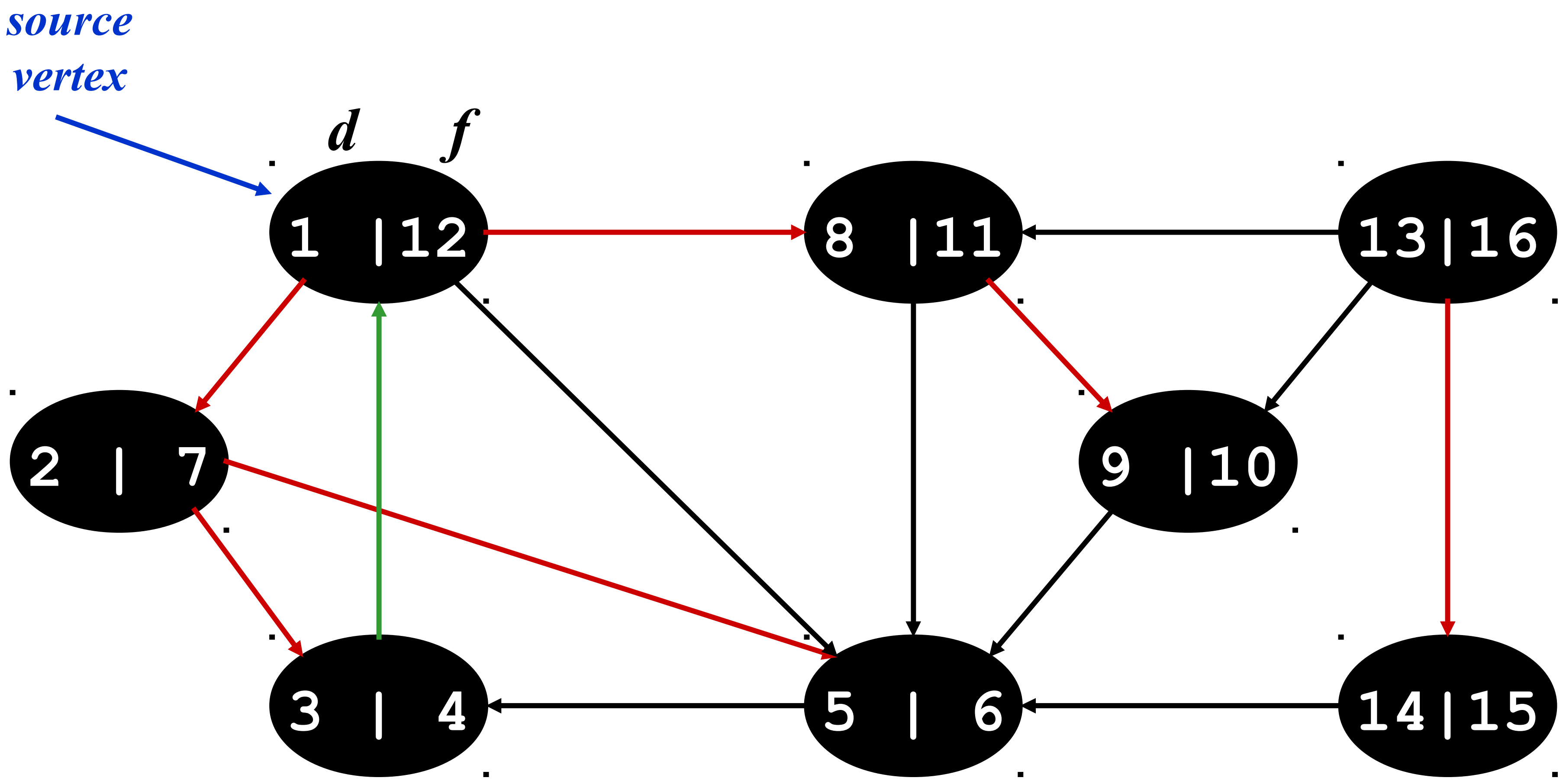
Depth-First Forest



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 in DFT
    - Encounter a *grey* vertex (*grey* to *grey*)

DFS Example

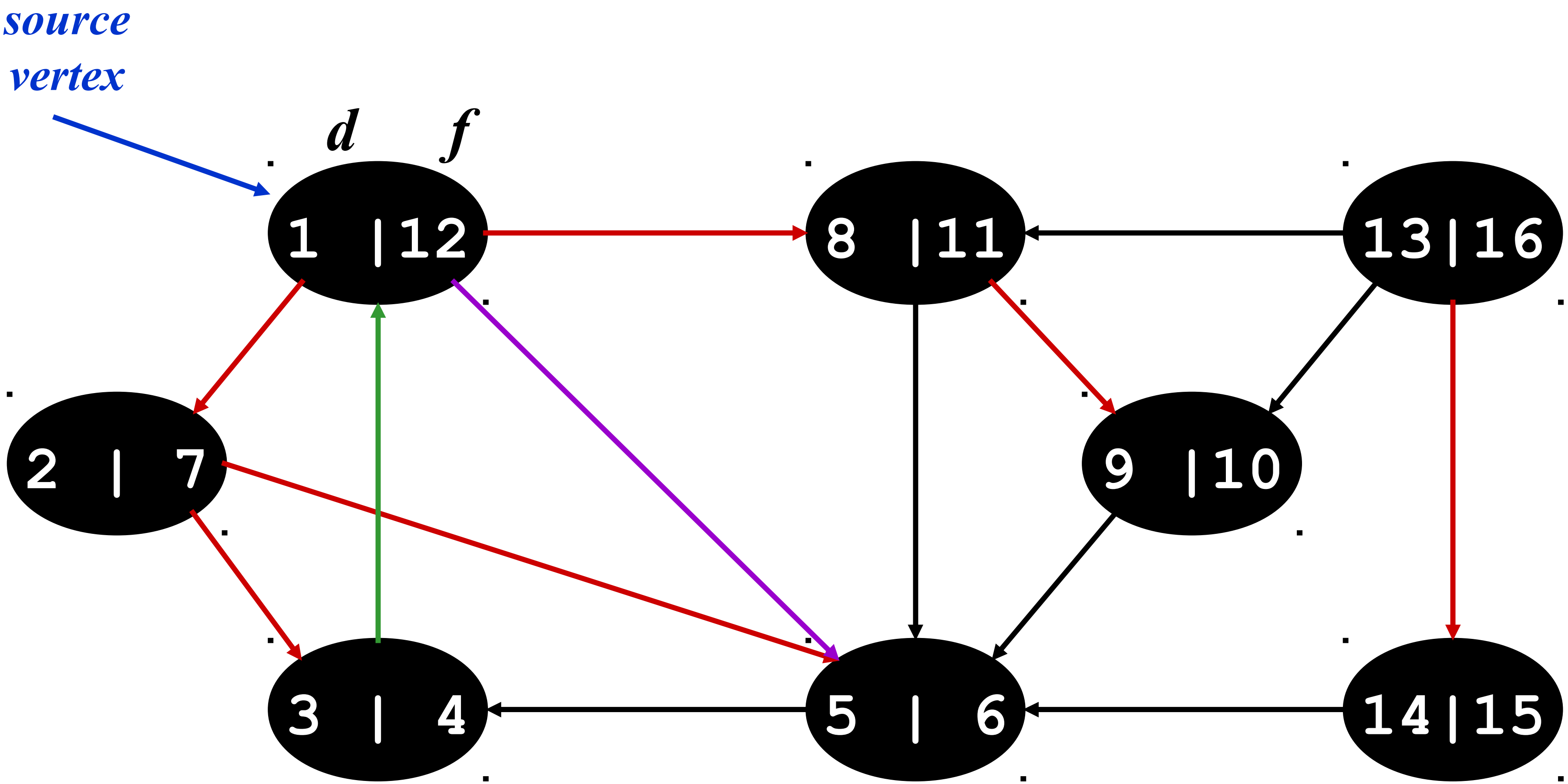


Tree edges *Back edges*

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 in DFT
  - *Forward edge*: from ancestor to descendent in DFT
    - Not a tree edge, though
    - Encounters a *black* node (from *grey* to *black*)

DFS Example

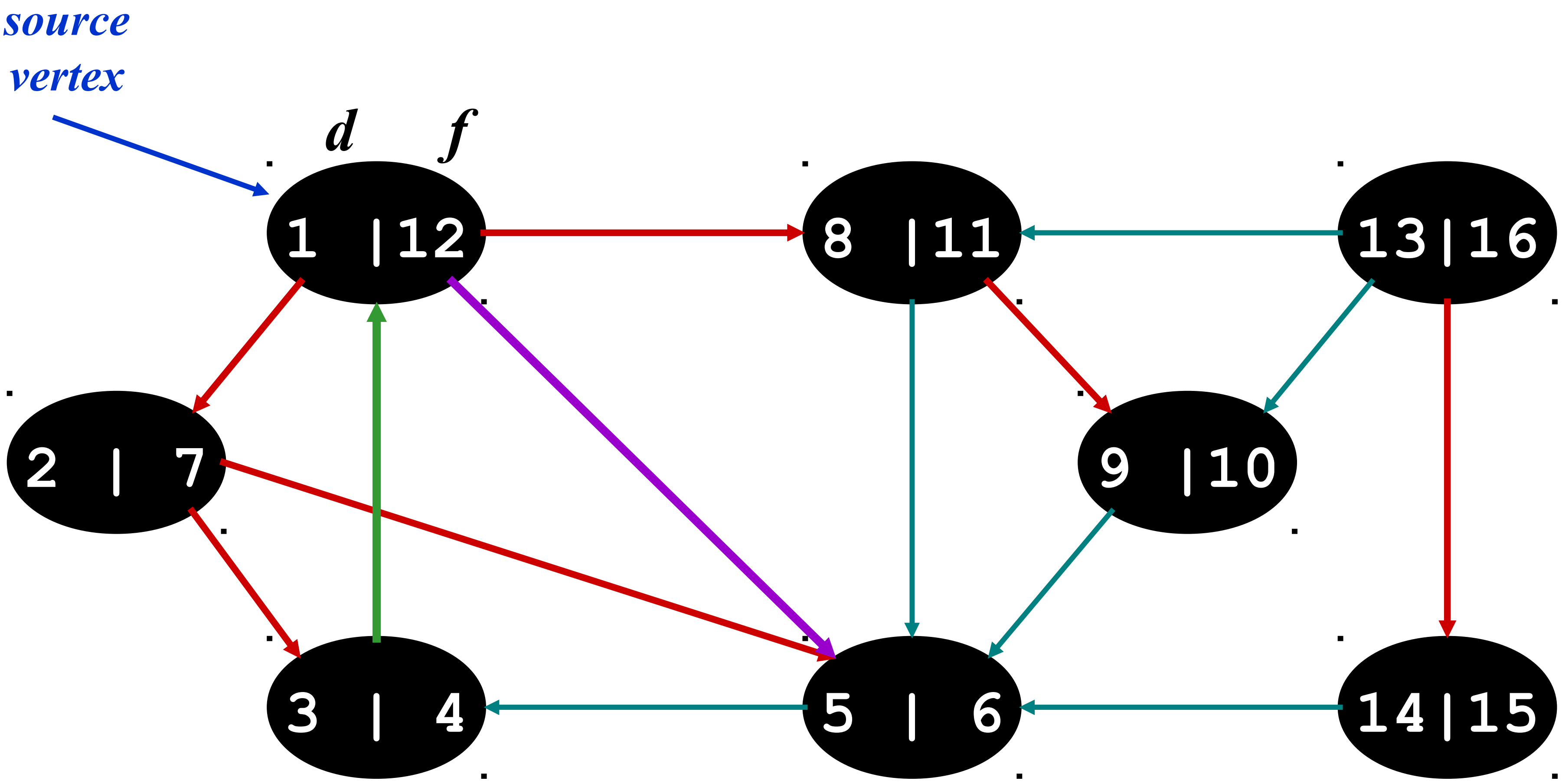


Tree edges *Back edges* *Forward edges*

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 in DFT
  - *Forward edge*: from ancestor to descendent in DFT
  - *Cross edge*: between nodes in a tree or subtrees
    - From a *grey* node to a *black* node
    - nodes not ancestors of each other

DFS Example



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

DFS: Kinds of edges

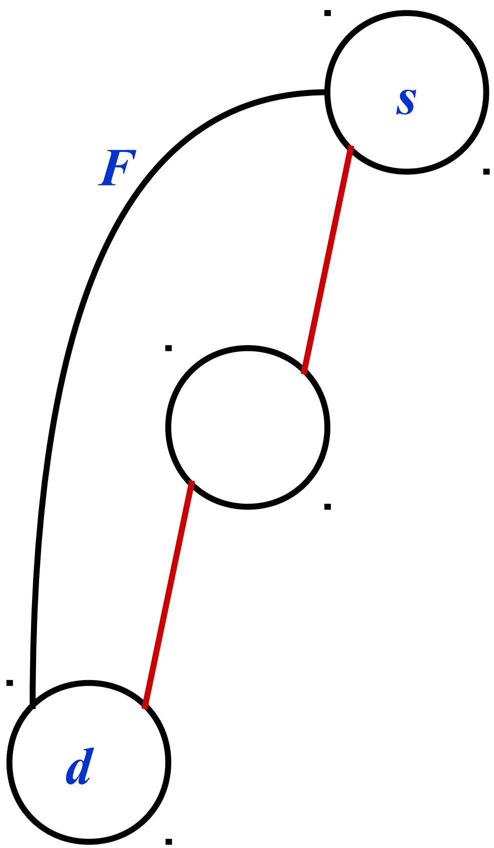
- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (*white*) vertex
  - *Back edge*: from descendent to ancestor in DFT (encounters *grey* edge)
  - *Forward edge*: from ancestor to descendent in DFT (encounters *black* edge)
  - *Cross edge*: between nodes in a tree or subtrees (encounters *black* edge)

DFS: Kinds Of Edges

Theorem. If G is undirected, a DFS produces only *tree* and *back* edges

Proof. Contradiction:

- Assume there's a forward edge
- But *F* edge must actually be a back edge (*why?*)
- It has to be discovered from *d* (goes to a *grey* vertex)

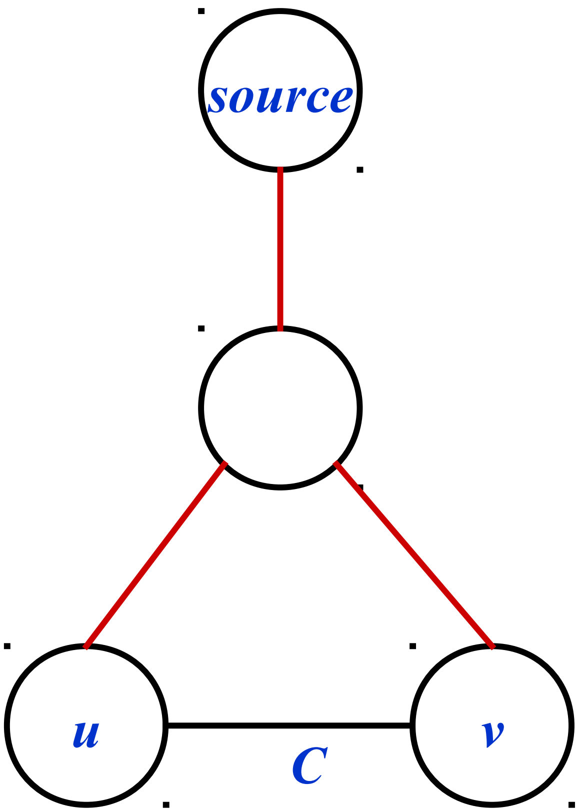


DFS: Kinds Of Edges

Theorem. If G is undirected, a DFS produces only *tree* and *back* edges

Proof. Contradiction:

- Assume there's a cross edge
- But *C* edge cannot be cross (*why?*)
- Must be explored from either *u* or *v*, 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

Theorem: An undirected graph is *acyclic* iff a DFS yields no back edges

Proof.

- 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

```
DFS(G)
    for each vertex u ∈ 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)
            v.p = u
            DFS-Visit(v)
    u.color = BLACK
    time = time+1
    u.f = time
```

How would you modify the code to detect cycles?

What's the running time?

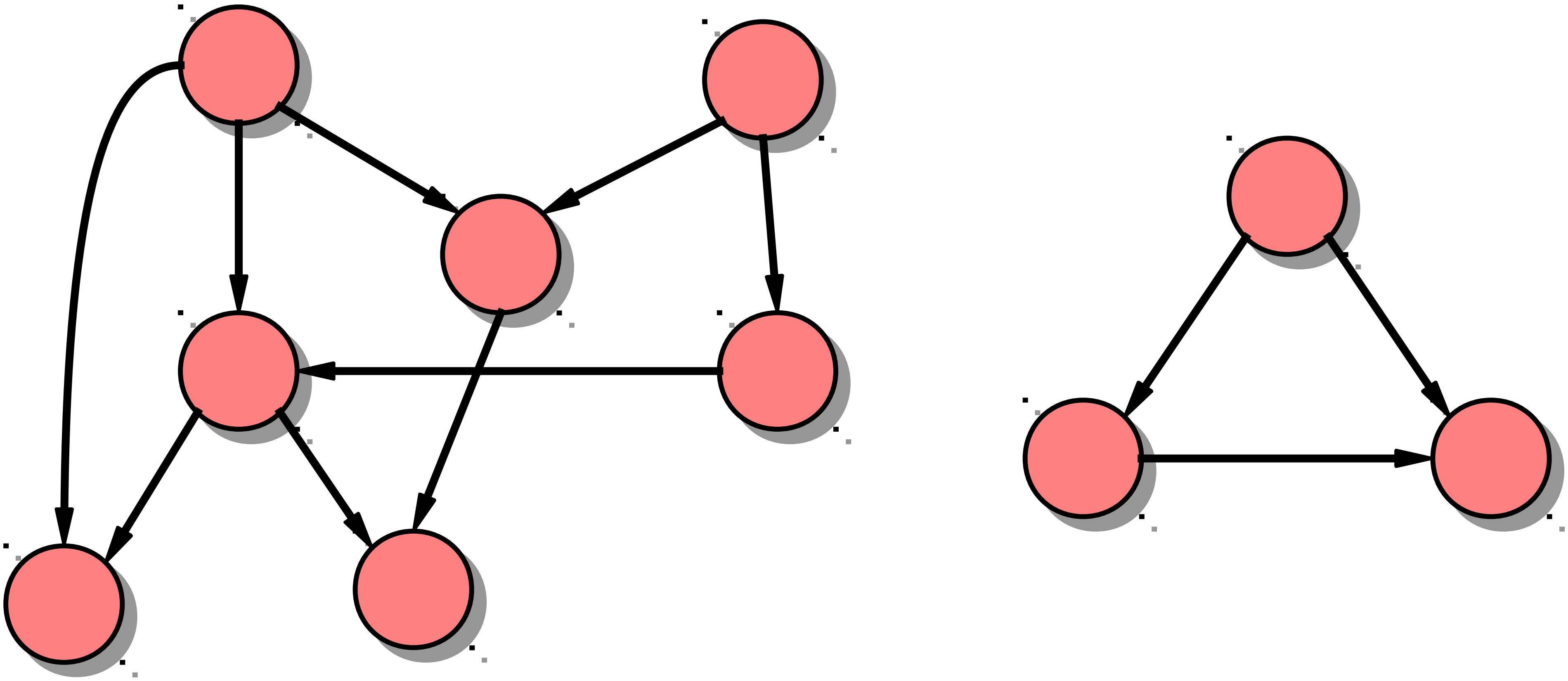
$\Theta(V+E)$

Depth First Search Applications

- Topological Sort
- Connected Components
- …

Directed Acyclic Graphs

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



DFS and DAGs

Theorem. A directed graph G is *acyclic* iff a DFS of G yields no *back* edges

Proof.

⇒ Suppose G is *acyclic* and there is a *back* edge (*u*, *v*). This means *u* is an descendant of *v* in the DFT. Thus G contains a path from *v* to *u* and the edge (*u*,*v*) completes the cycle. Contradiction.

DFS and DAGs

Theorem. A directed graph G is *acyclic* iff a DFS of G yields no *back* edges

Proof.

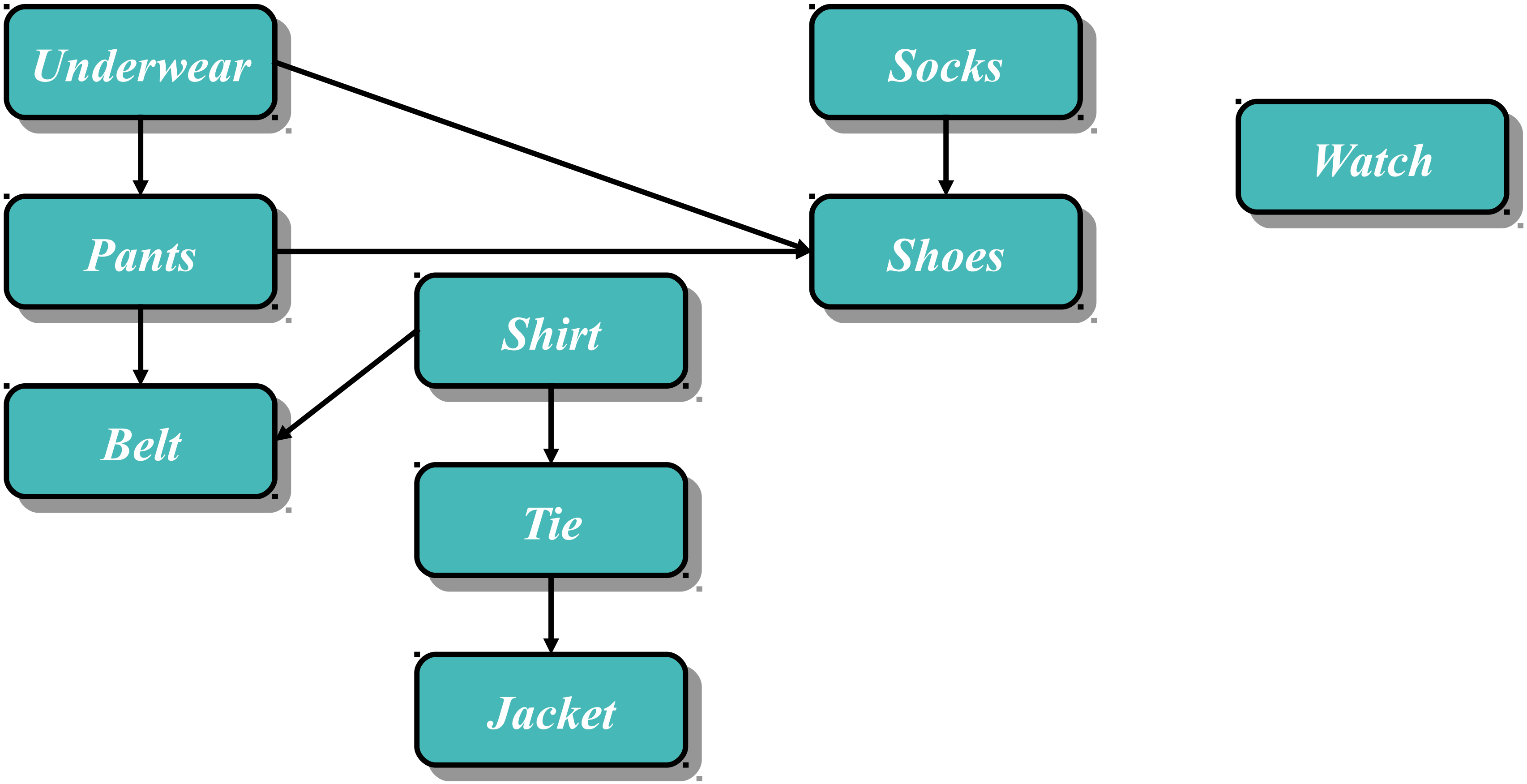
⇐ Contrapositive. Suppose G has a cycle *c*. Let *v* be the first vertex to be discovered in *c*, and let *u* be its ancestor *c*. At time *v*.*d*, there is a path of *white* vertices from *v* to *u* (on the cycle). Since DFS-Visit(*v*) does not return until all vertices reachable from *v* are visited, the edge (*u*,*v*) will be a *back* edge as *u* will be *grey*. So (*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 example: getting dressed

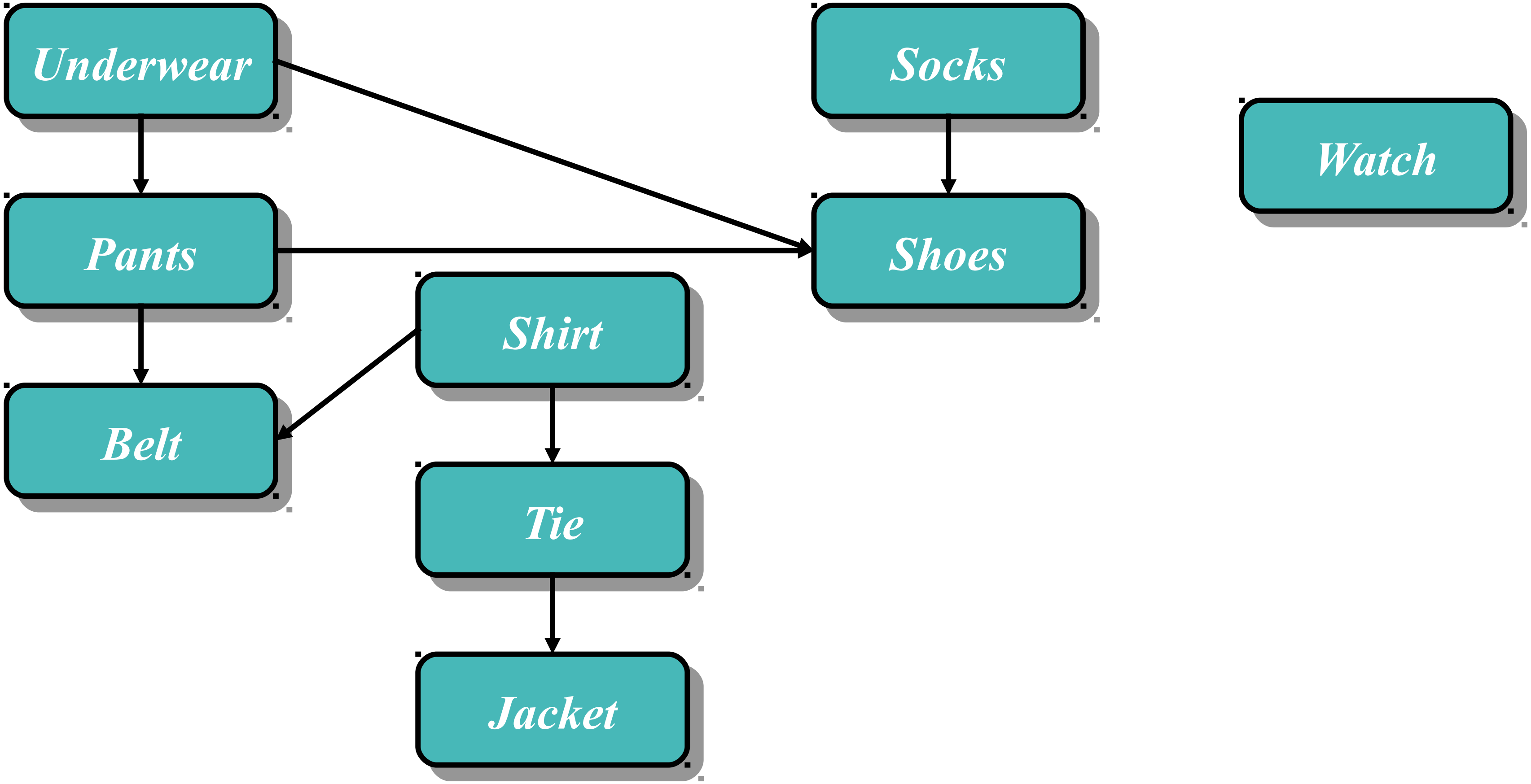
Getting Dressed



Clothes items are ordered such that an edge (u,v) implies that item u should be worn *before* item v

In what order should Mr Tidy get dressed obeying these rules?

Getting Dressed



Topological Sort

Topological Sort Algorithm

```
Topological-Sort(G)
    Run DFS(G)
    When a vertex is finished, insert to front of
        a linked list
    return linked list of vertices
```

Running time: $\Theta(E+V)$. *Why?*

Correctness of Topological Sort

Claim. $(u,v) \in G \Rightarrow u.f > v.f$

Proof.

When (u,v) is explored, u is *grey*

- $v = grey \Rightarrow (u,v)$ is *back* edge. Contradiction (*Why?*)
- $v = white \Rightarrow v$ becomes descendent of $u \Rightarrow v.f < u.f$
  (since must finish v before backtracking and finishing u)
- $v = black \Rightarrow v$ already finished $\Rightarrow v.f < u.f$

Summary

- Breadth-First Search
  - Explores the graph by discovering nodes across the breadth
  - Finds shortest paths from one vertex (and all reachable vertices)
  - Produces the Breadth-First Tree
  - Runs in time $\Theta(V+E)$
- Depth-First Search
  - Explores the graph by diving deeper into its depth
  - Produces the Depth-First Forest
  - Runs in time $\Theta(V+E)$

Recap

- Breadth First Search
- Depth First Search