COMP319 Algorithms 1
Lecture 13
Graph Search

Instructor: Gil-Jin Jang

School of Electronics Engineering

Kyungpook National University

Textbook Chapters 23 and 24

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Table of Contents

Graph definition

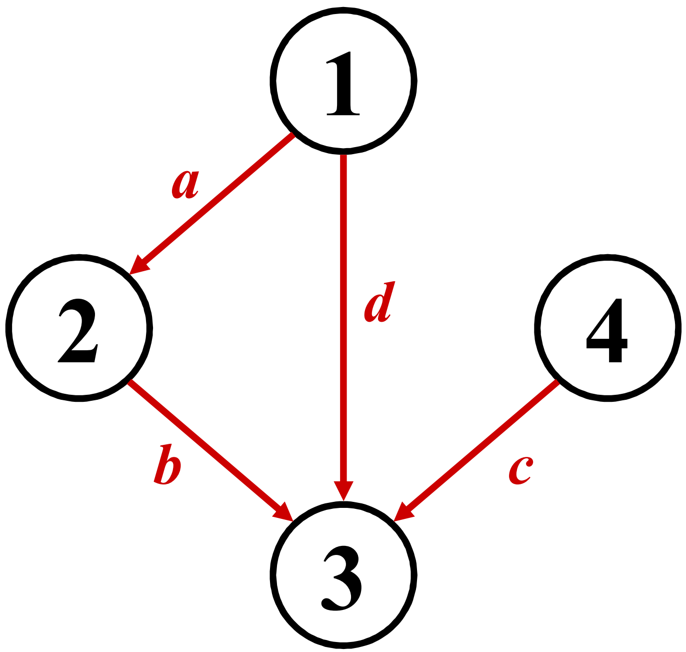
Implementation of graphs

GRAPH ALGORITHMS

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Graphs

- A graph G = (V, E)
  - V = set of vertices (nodes)
  - E = set of edges = subset of $V \times V$ (Cartesian product of two vertices)
  - 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 $u \rightarrow v$

Graph Variations

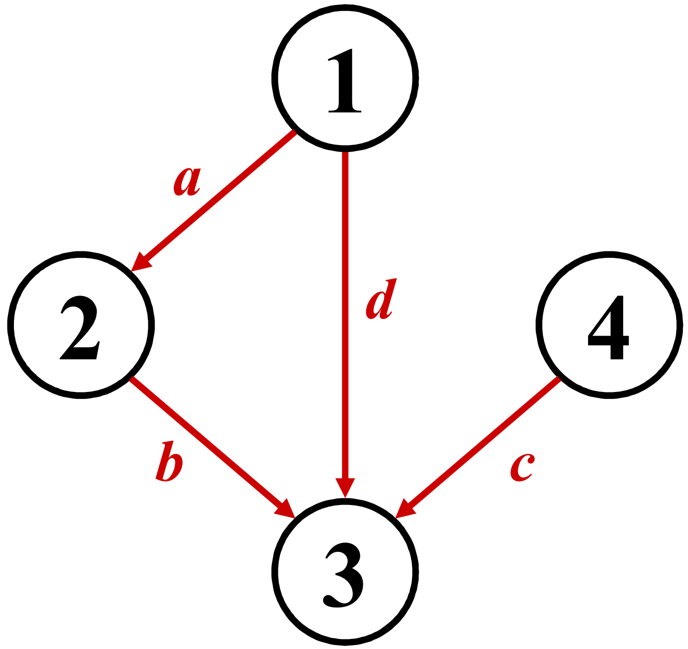
- More variations:
  - A *weighted graph* associates weights with either the edges or the vertices
    - E.g., a road map: edges might be weighted w/ distance
  - A *multigraph* allows multiple edges between the same vertices
    - E.g., the call graph in a program (a function can get called from multiple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

Representing 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

- 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



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

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

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 $\rightarrow$ matrix is symmetric
  - No self-loops $\rightarrow$ don't need diagonal

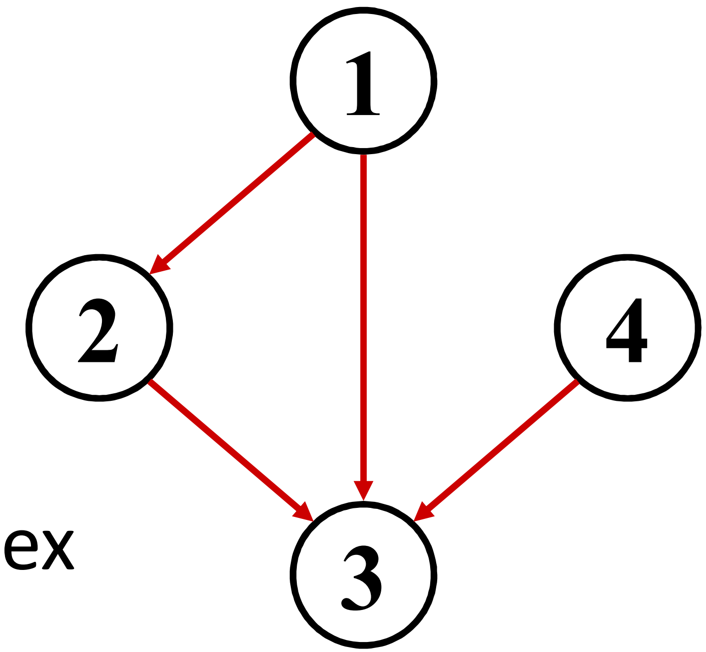
Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

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
    Σ out-degree(*v*) = |E|
    takes Θ(V + E) storage (*Why?*)
  - For undirected graphs, # items in adj lists is
    Σ degree(v) = 2 |E| (*handshaking lemma*)
    also Θ(V + E) storage
- So: Adjacency lists take O(V+E) storage

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Graph Searching

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

- Goal:
  - Find methods (algorithms) that explore every vertex and every edge

- Ultimately: build a **<u>TREE</u>** 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

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

BREADTH-FIRST SEARCH

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.

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

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

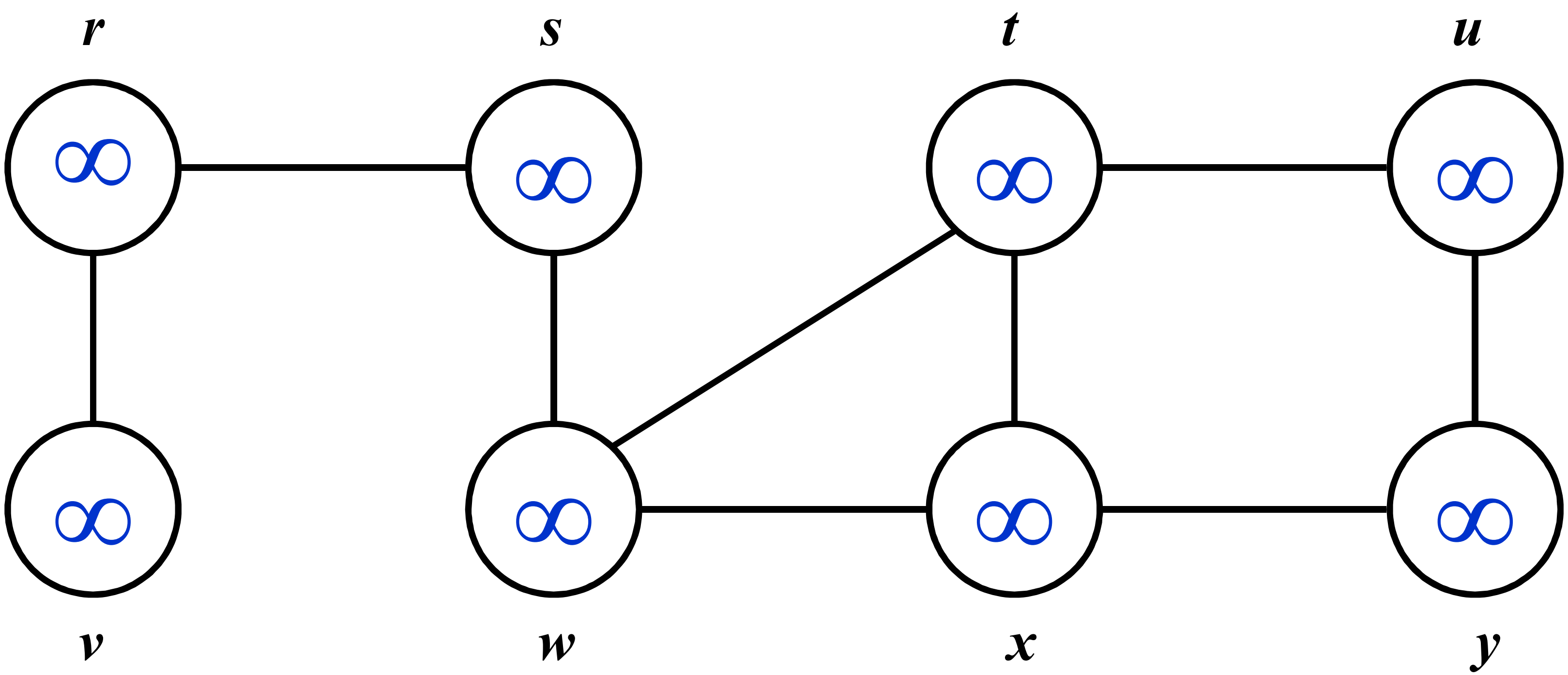
Breadth-First Search

```
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?*

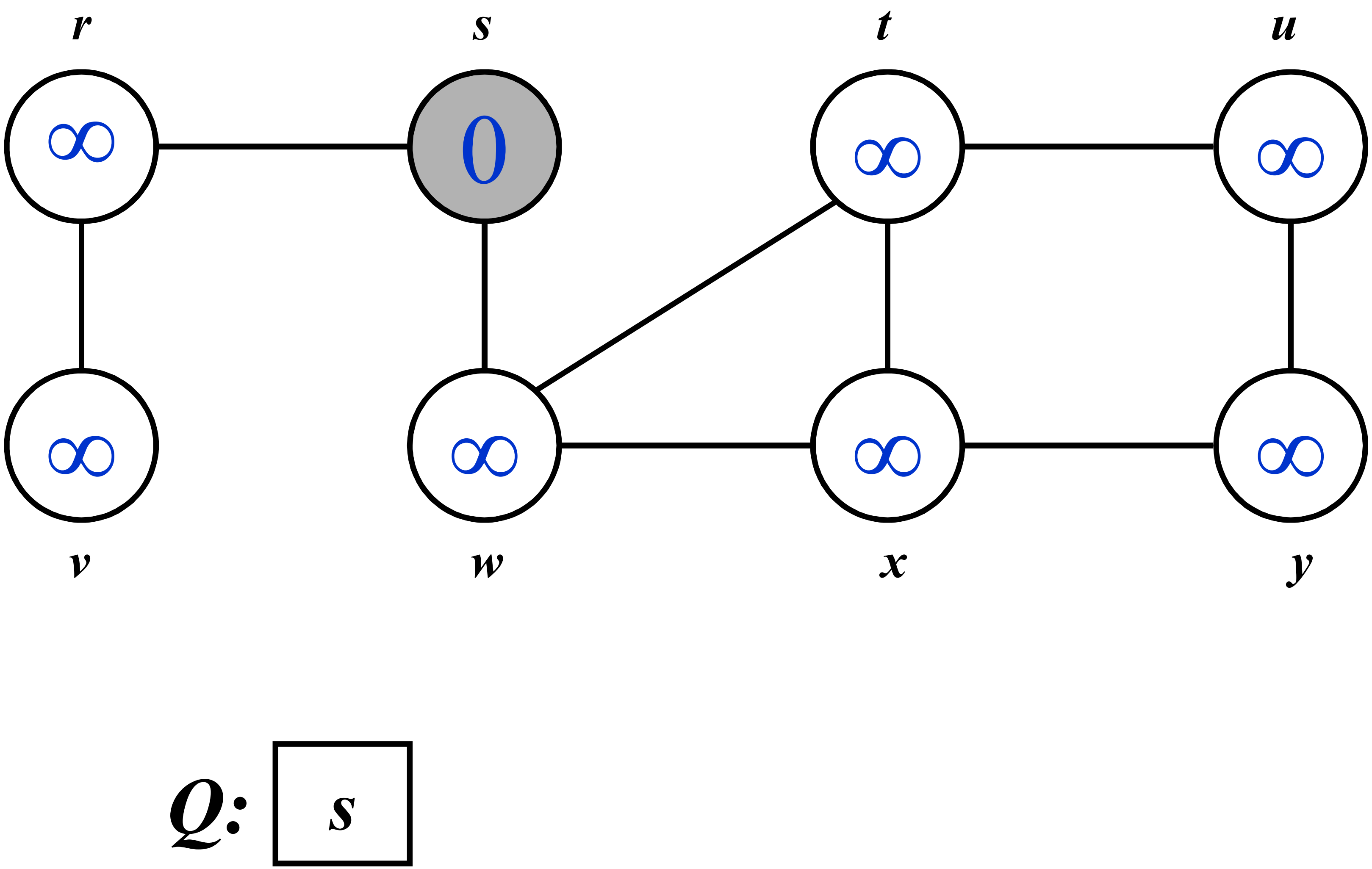
What does **v->p** *represent?*

Breadth-First Search: Example



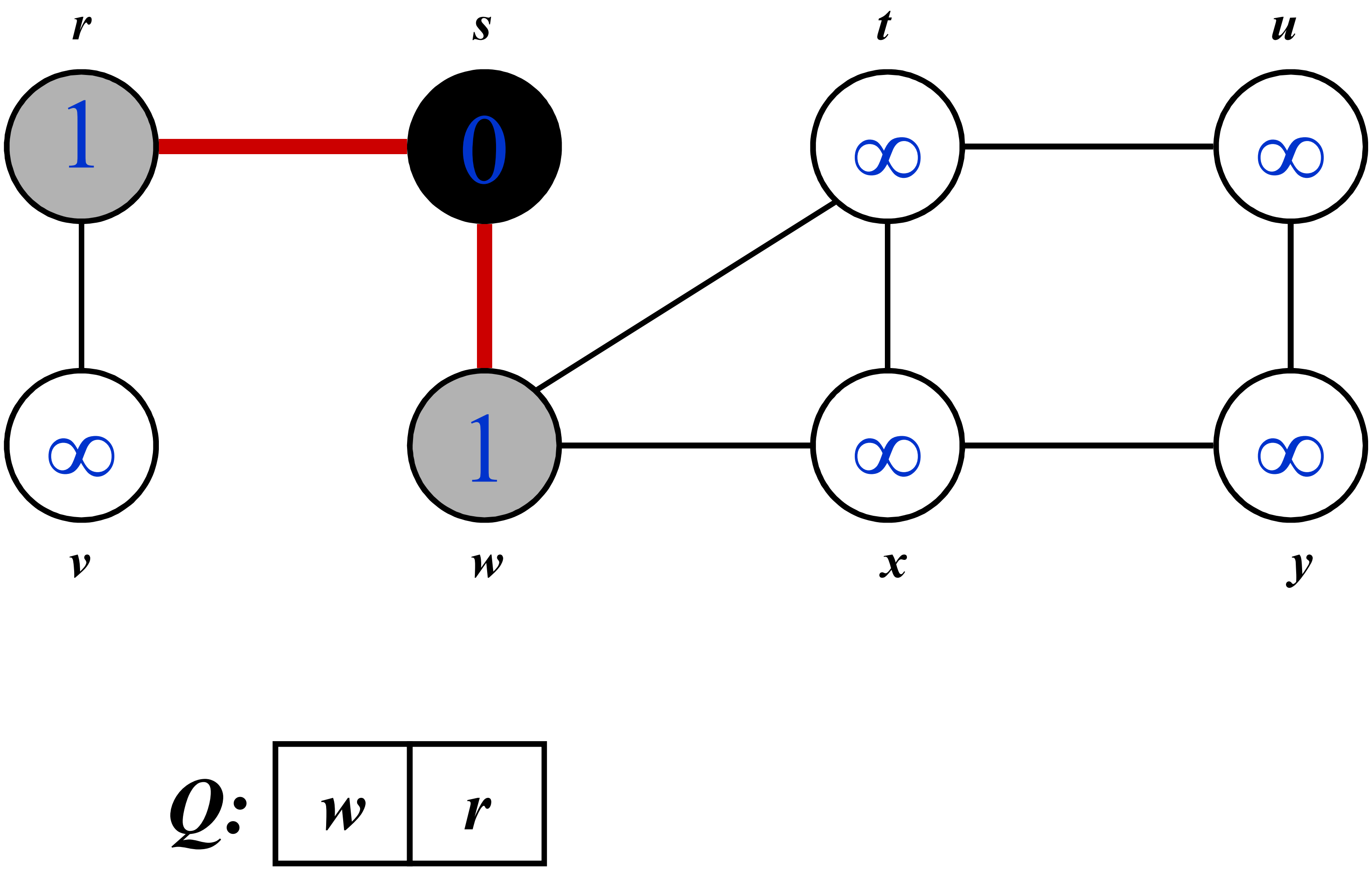
Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Breadth-First Search: Example

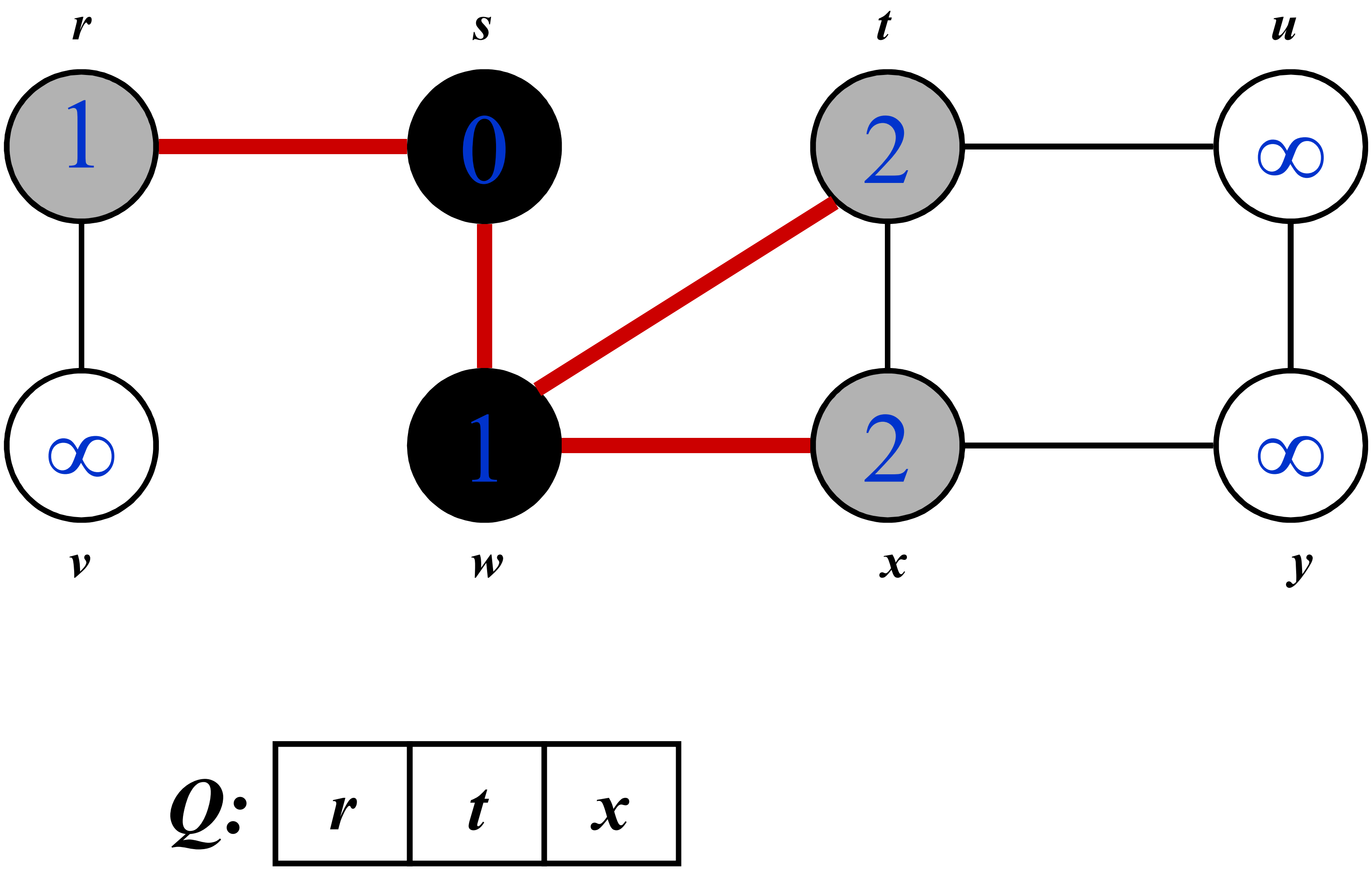


Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Breadth-First Search: Example

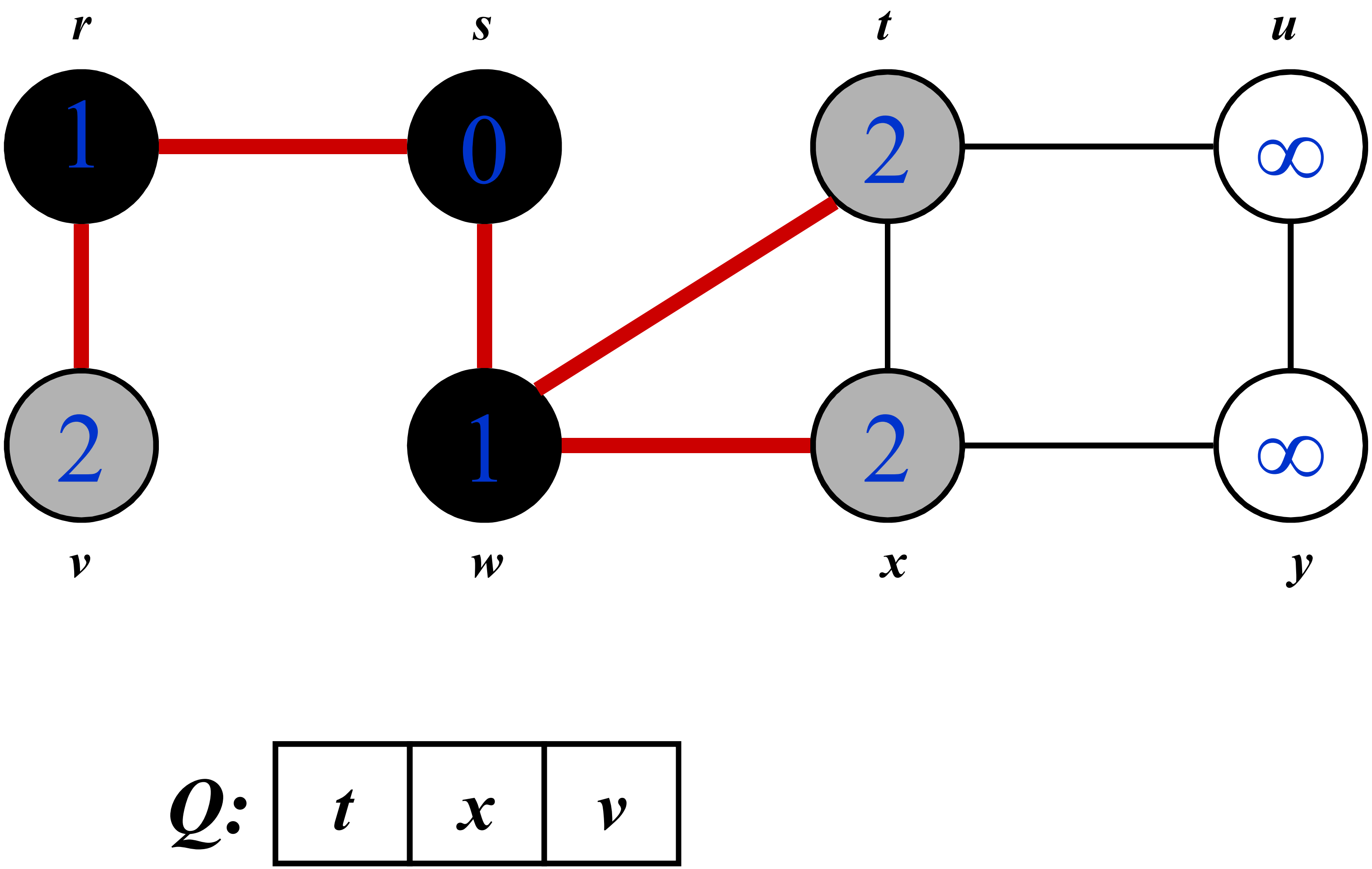


Breadth-First Search: Example

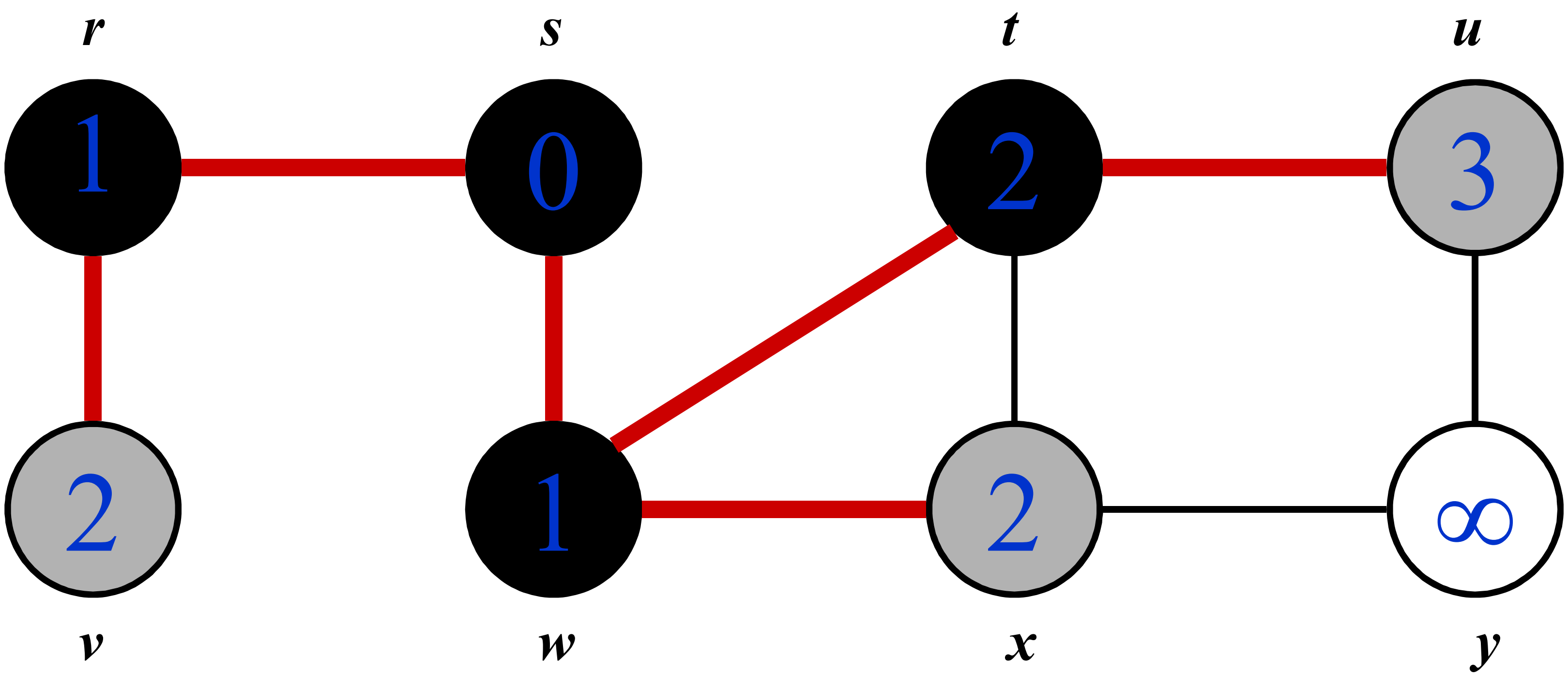


Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Breadth-First Search: Example



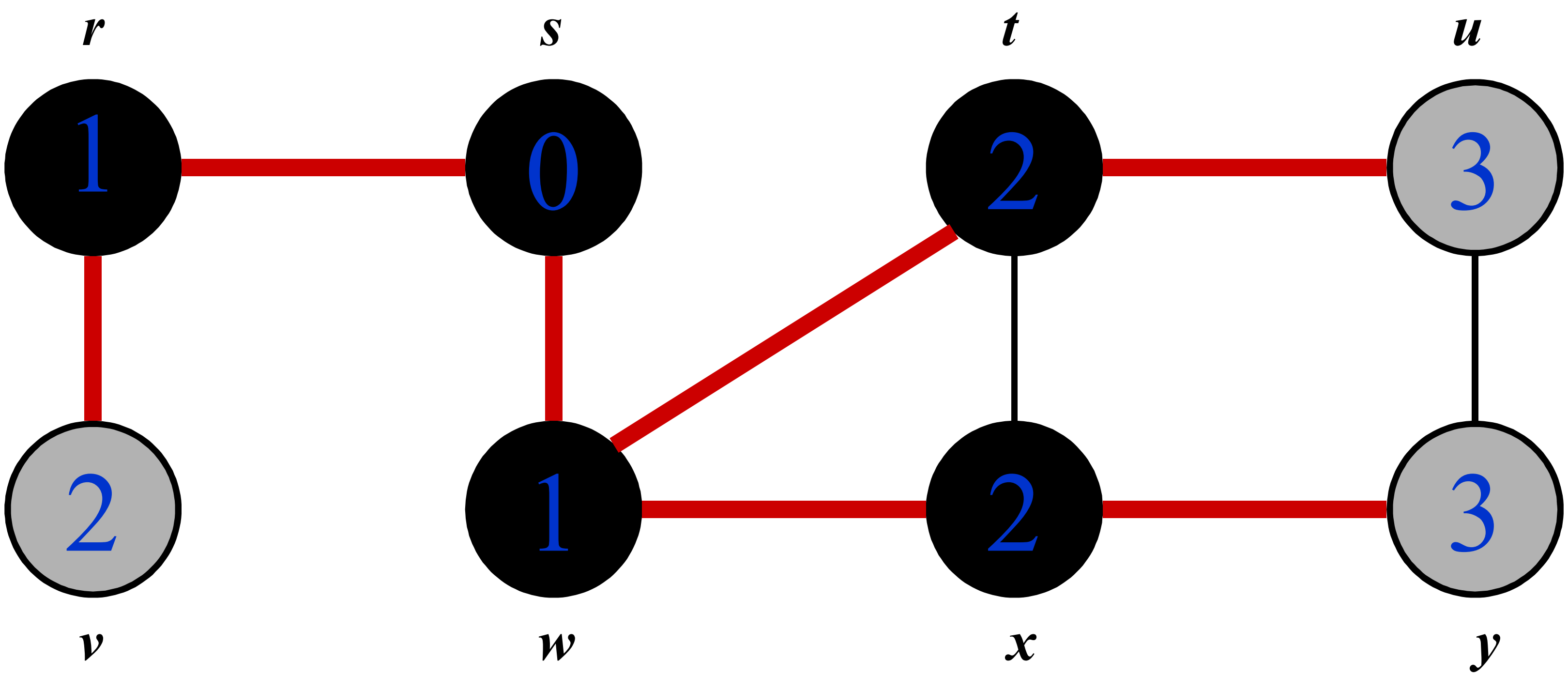
Breadth-First Search: Example



Q:	x	v	u

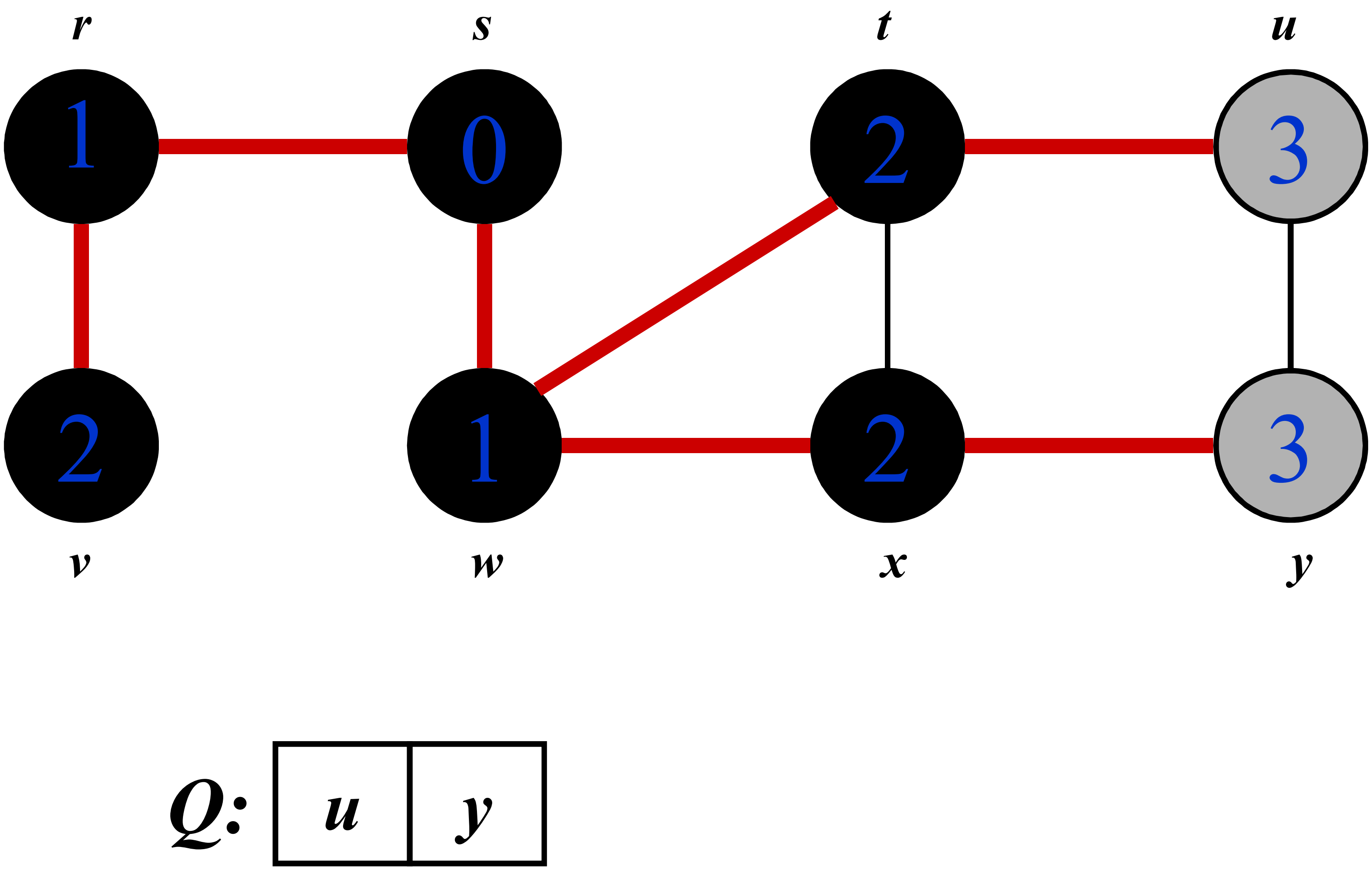
Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Breadth-First Search: Example

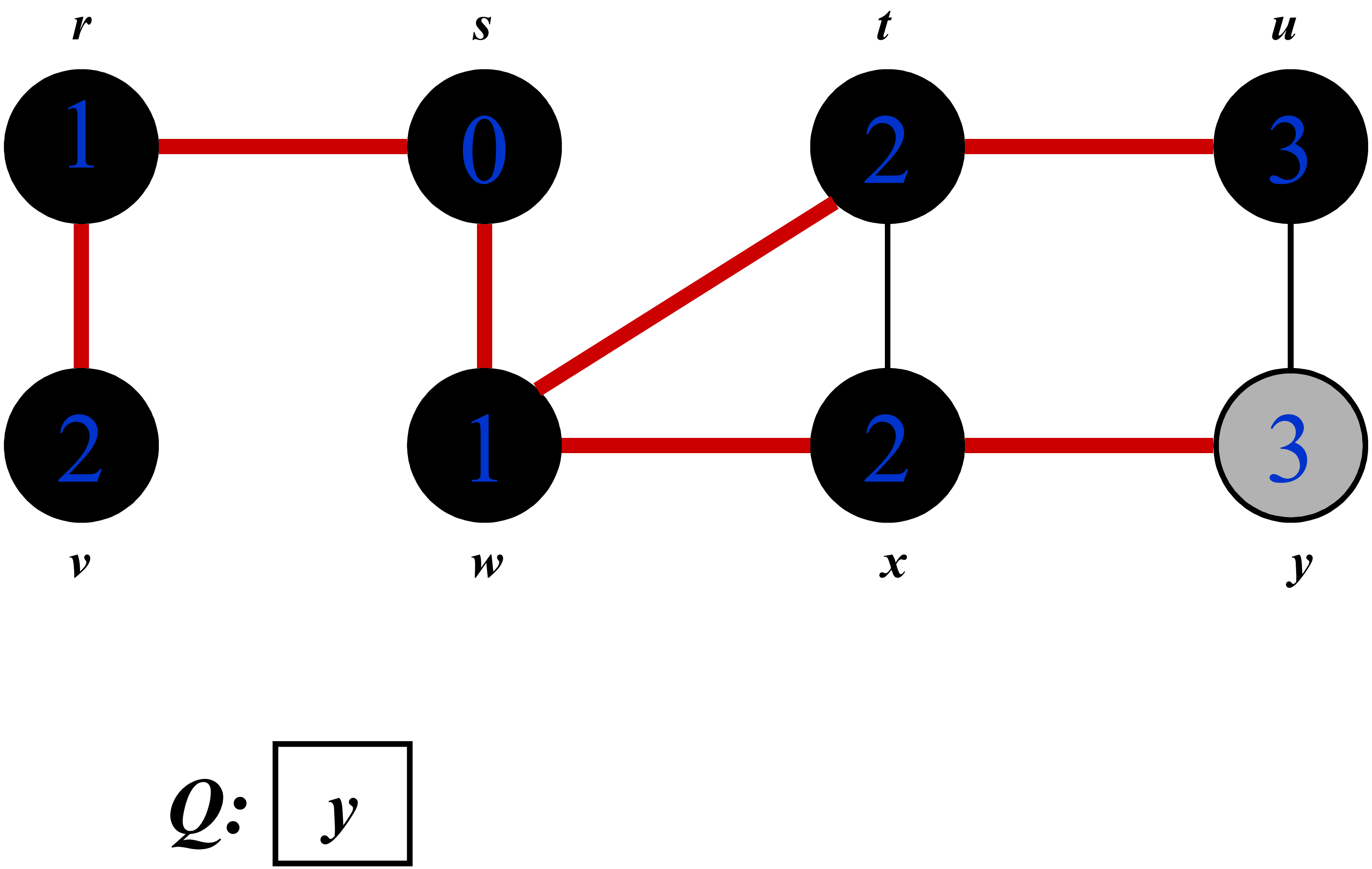


Q: | v | u | y |

Breadth-First Search: Example

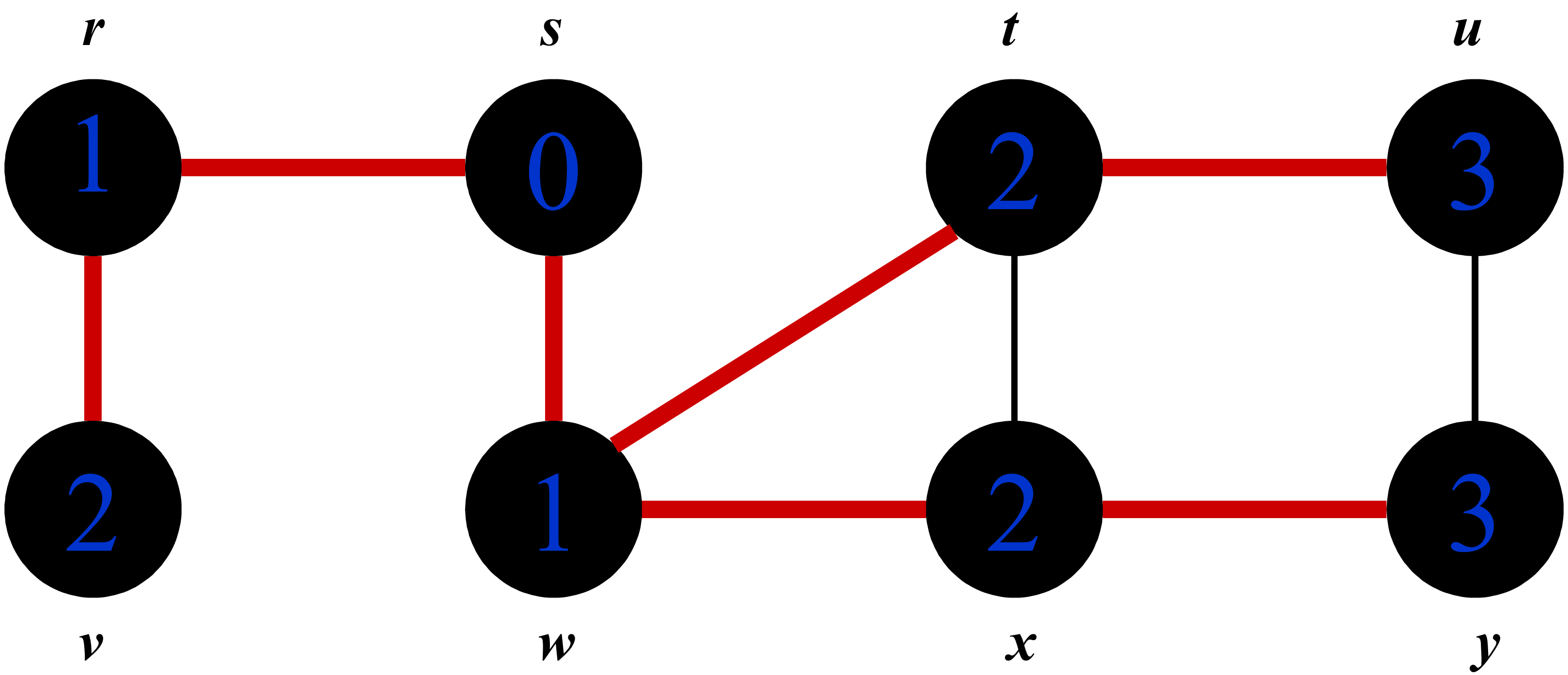


Breadth-First Search: Example



Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Breadth-First Search: Example



Q: Ø

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

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)

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

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;
    }
}
```

What will be the storage cost in addition to storing the tree?

Total space used:

O(max(degree(v))) = O(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

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

DEPTH-FIRST SEARCH

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

- Vertices initially colored white
- Then colored gray when discovered
- Then black when finished

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Depth-First Search: The 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);
    }
}
```

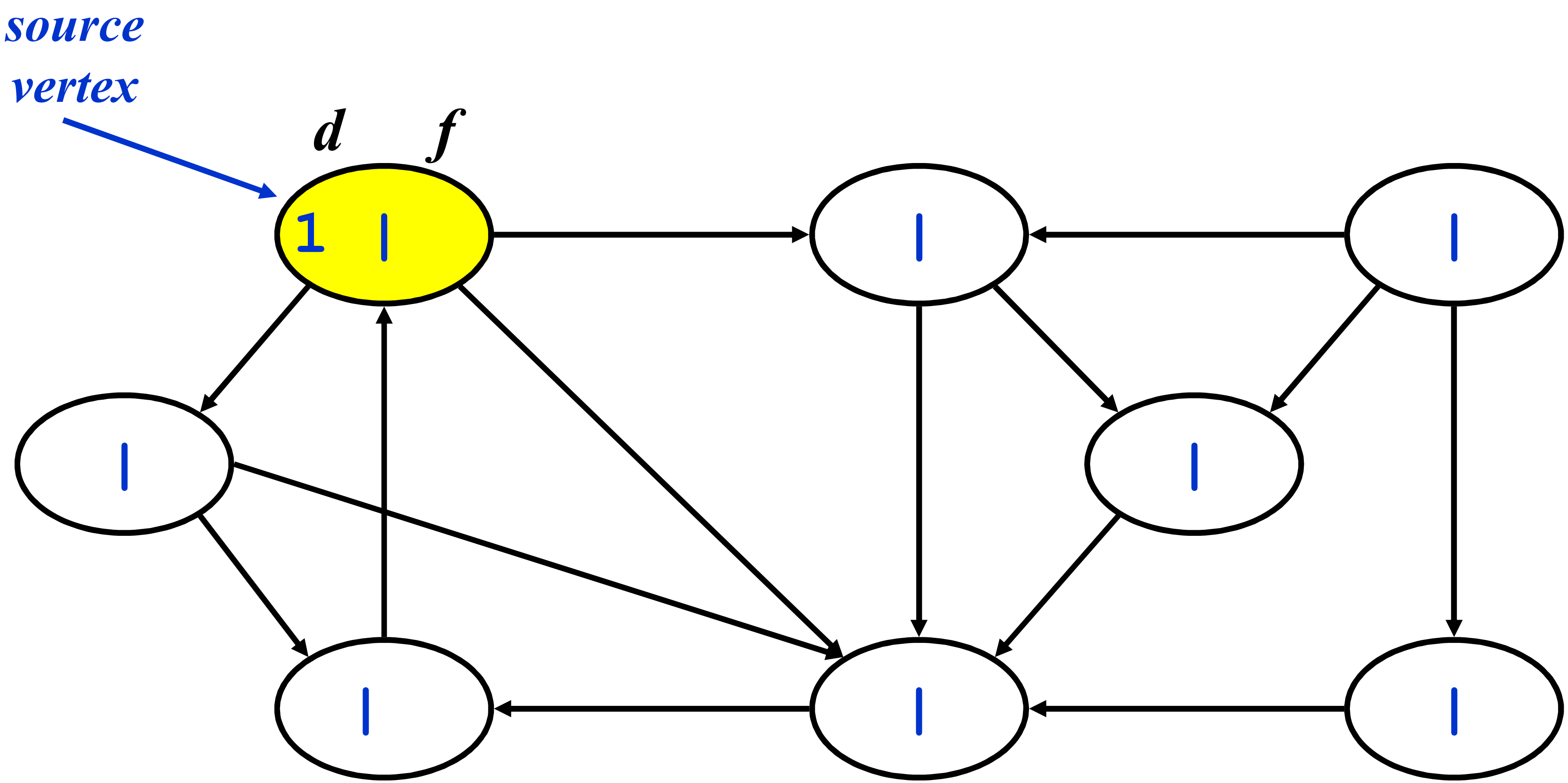
```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

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

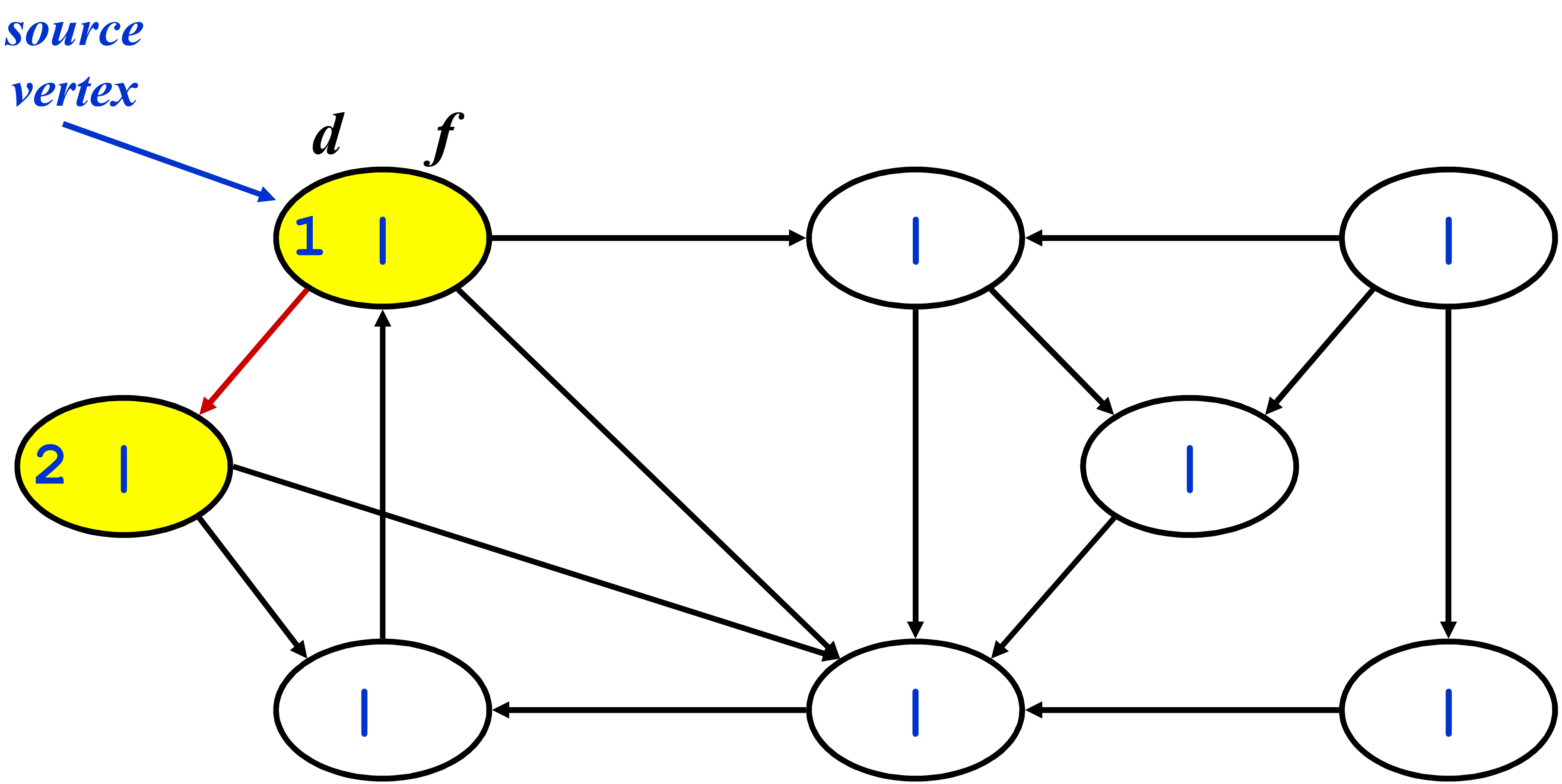
What does* `u->f` *represent?

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

DFS Example

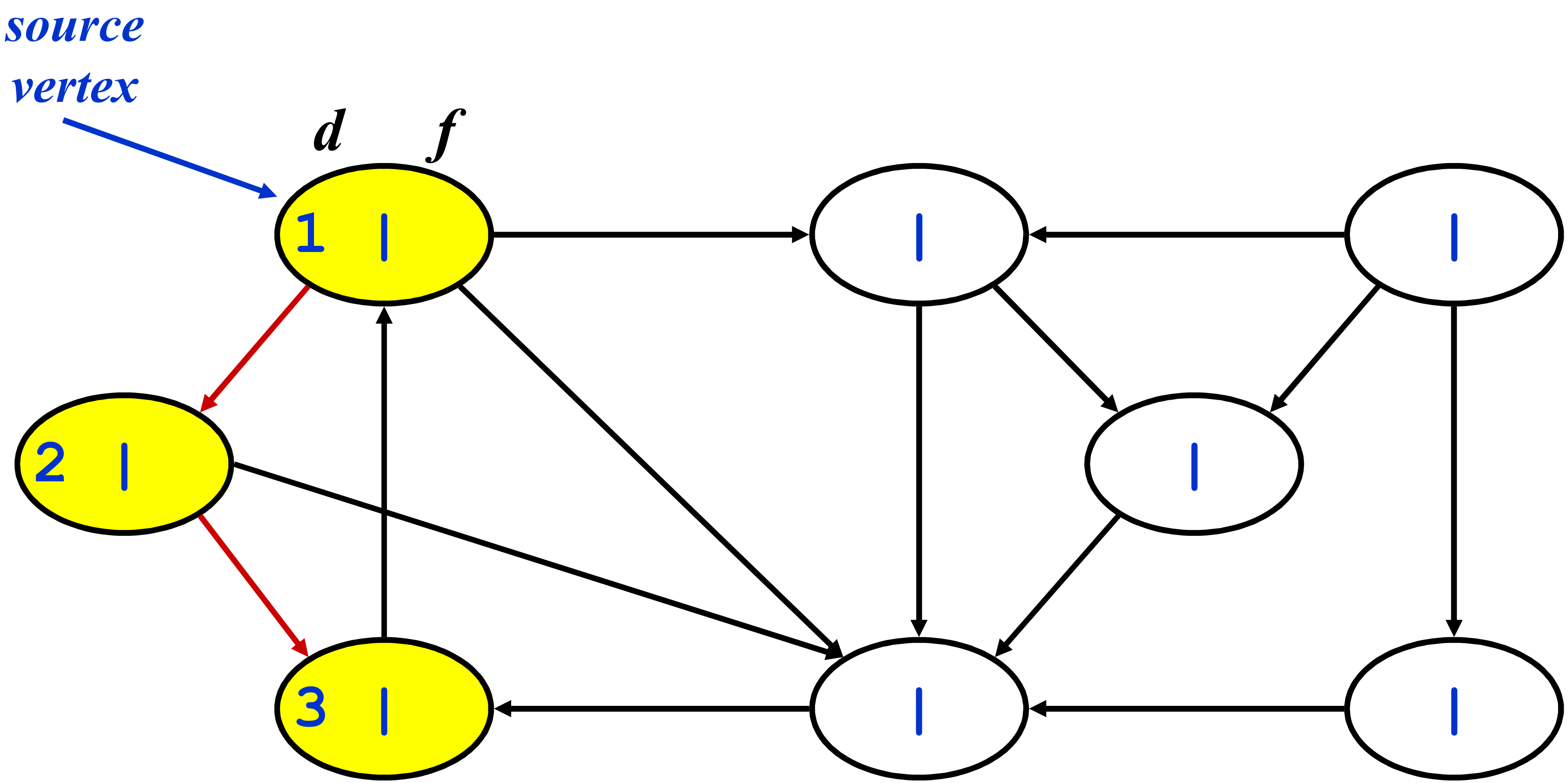


DFS Example



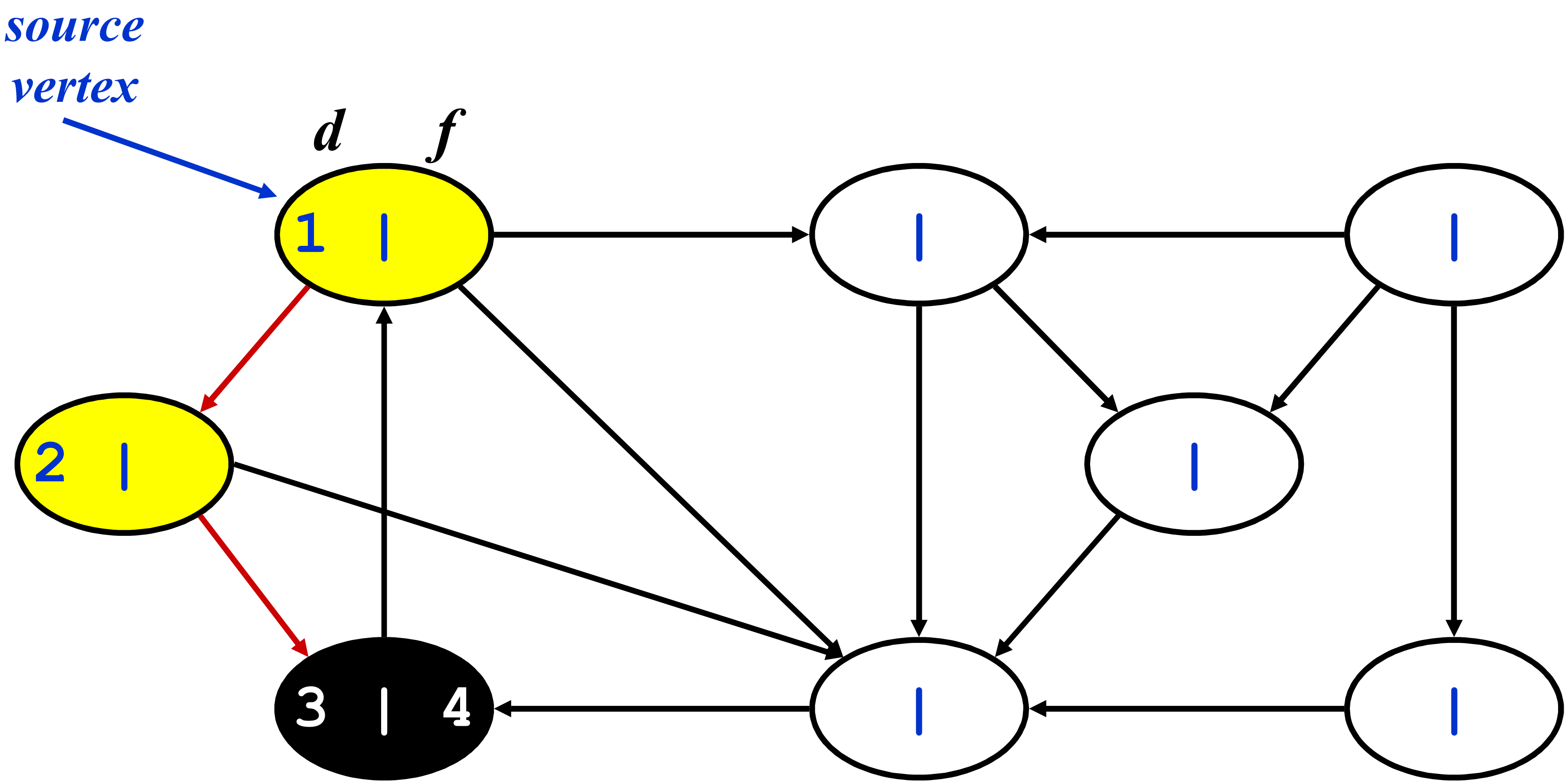
Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

DFS Example

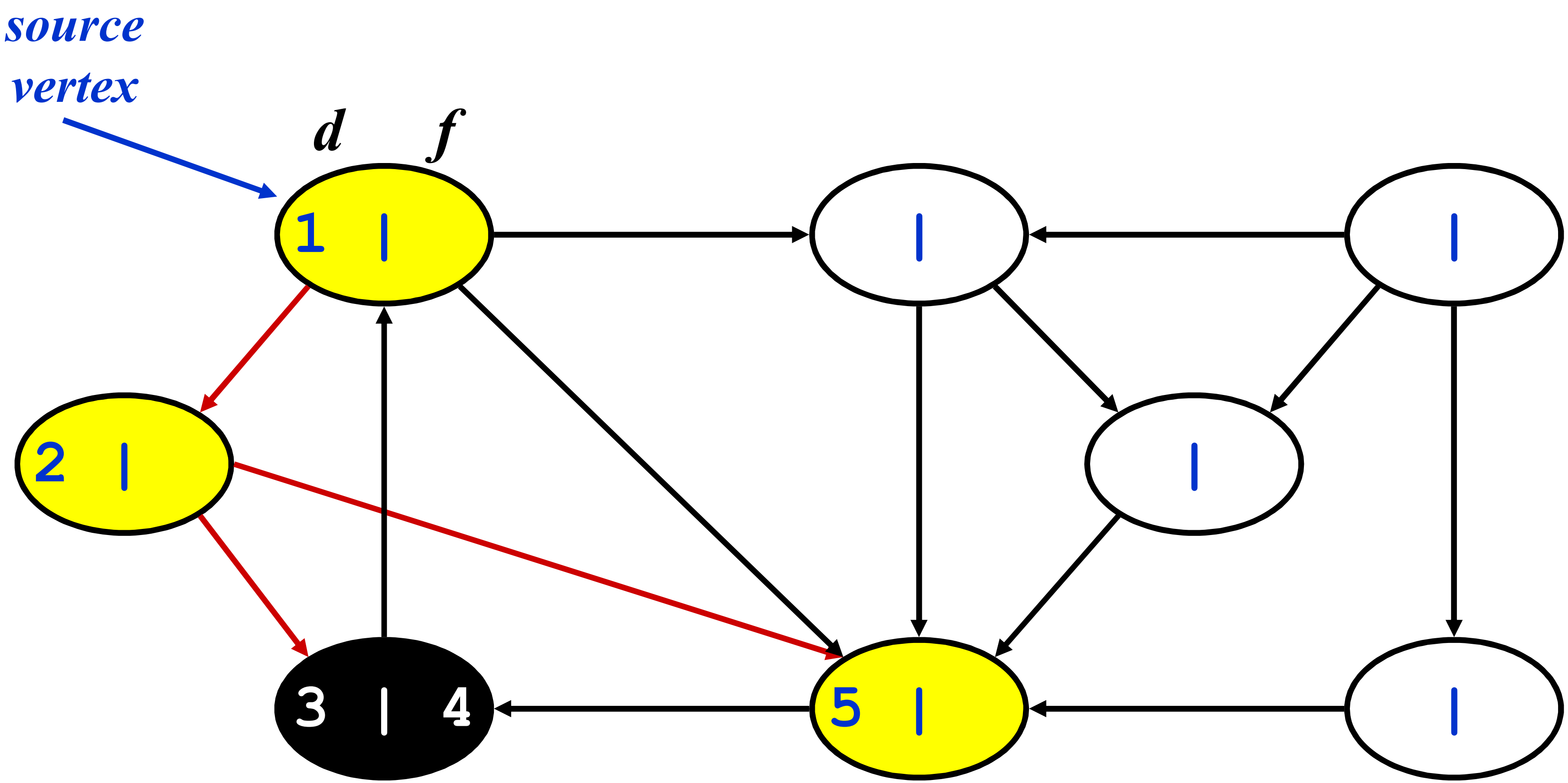


Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

DFS Example

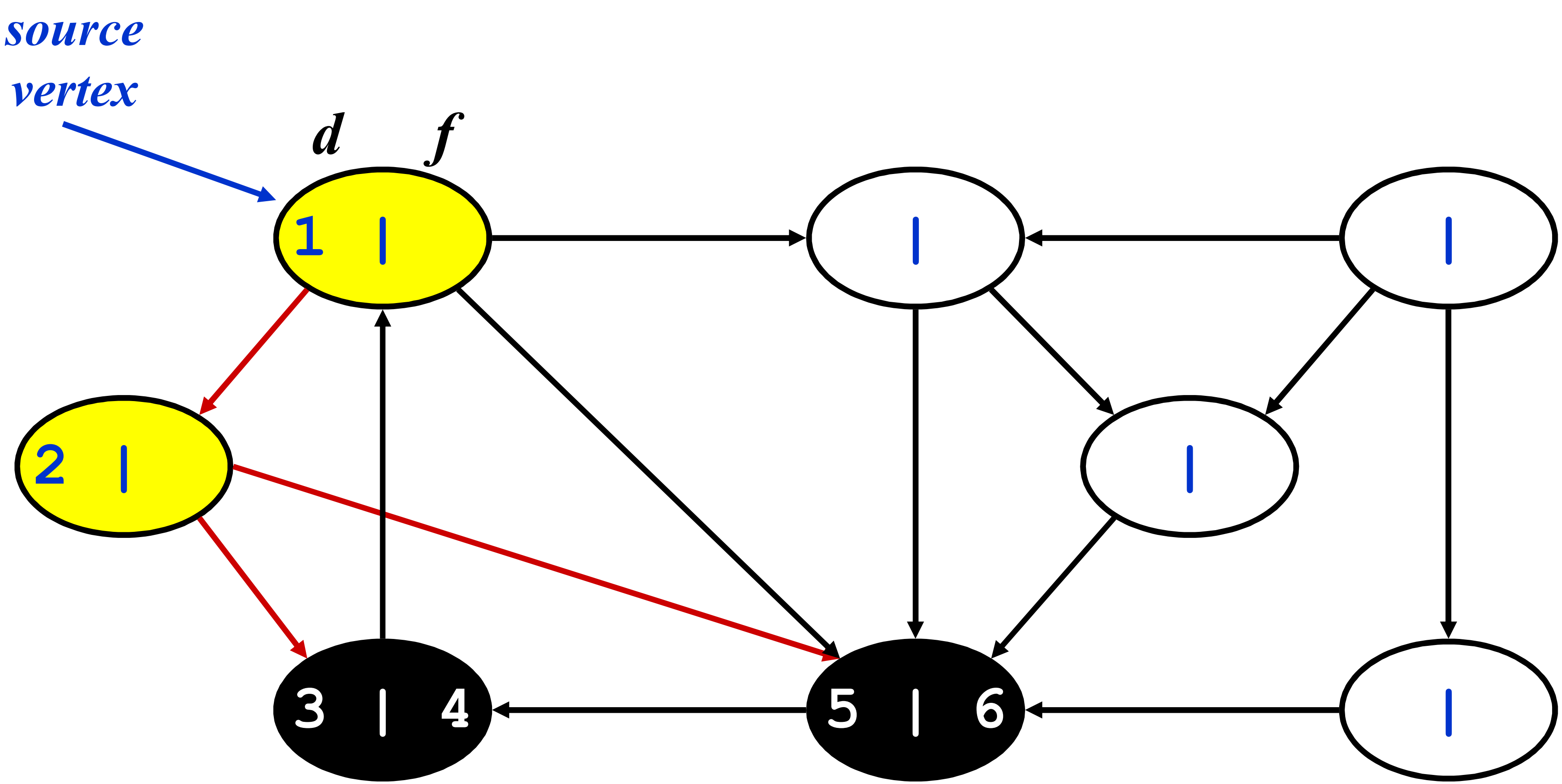


DFS Example



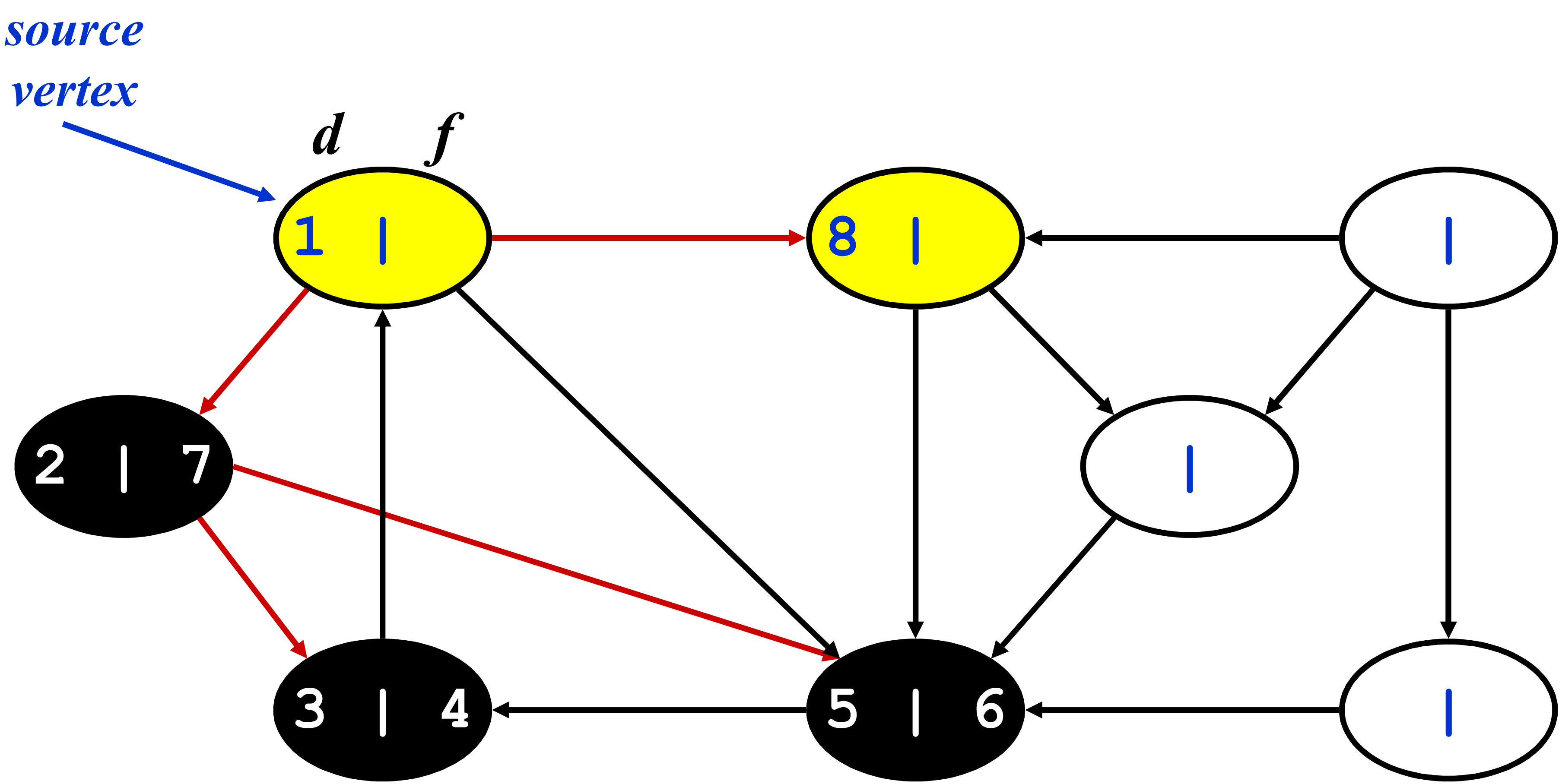
Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

DFS Example

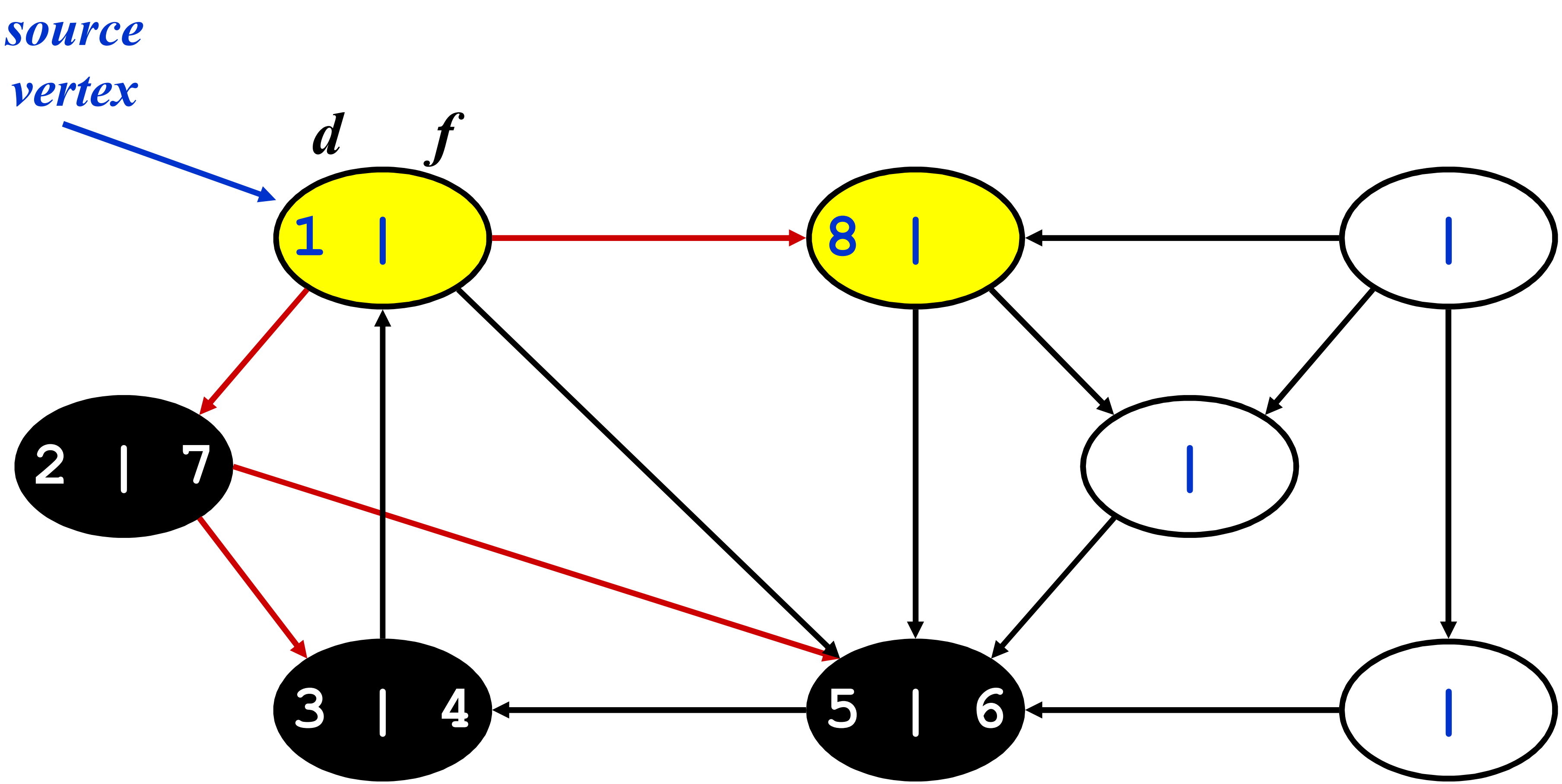


Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

DFS Example

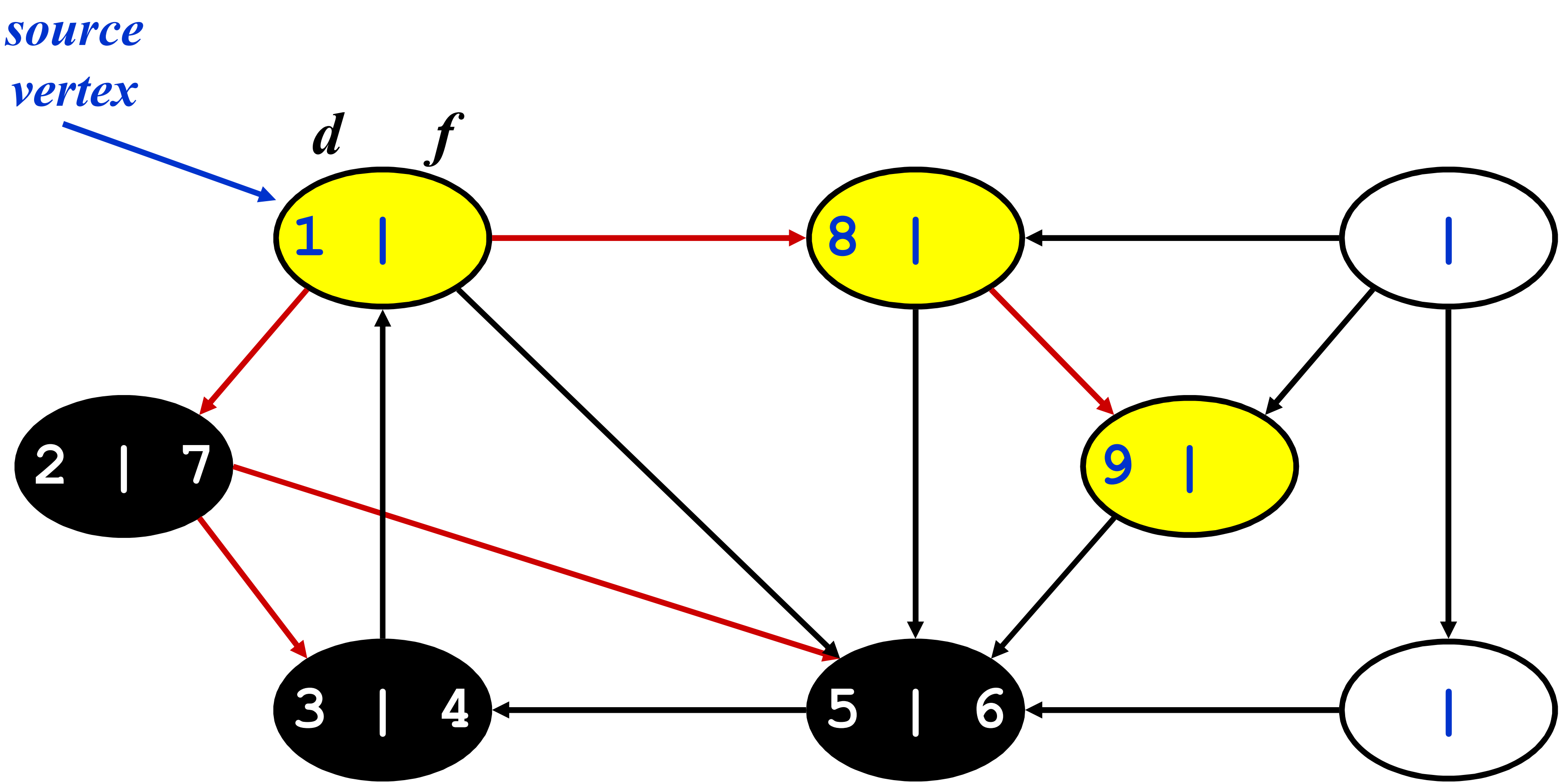


DFS Example



Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

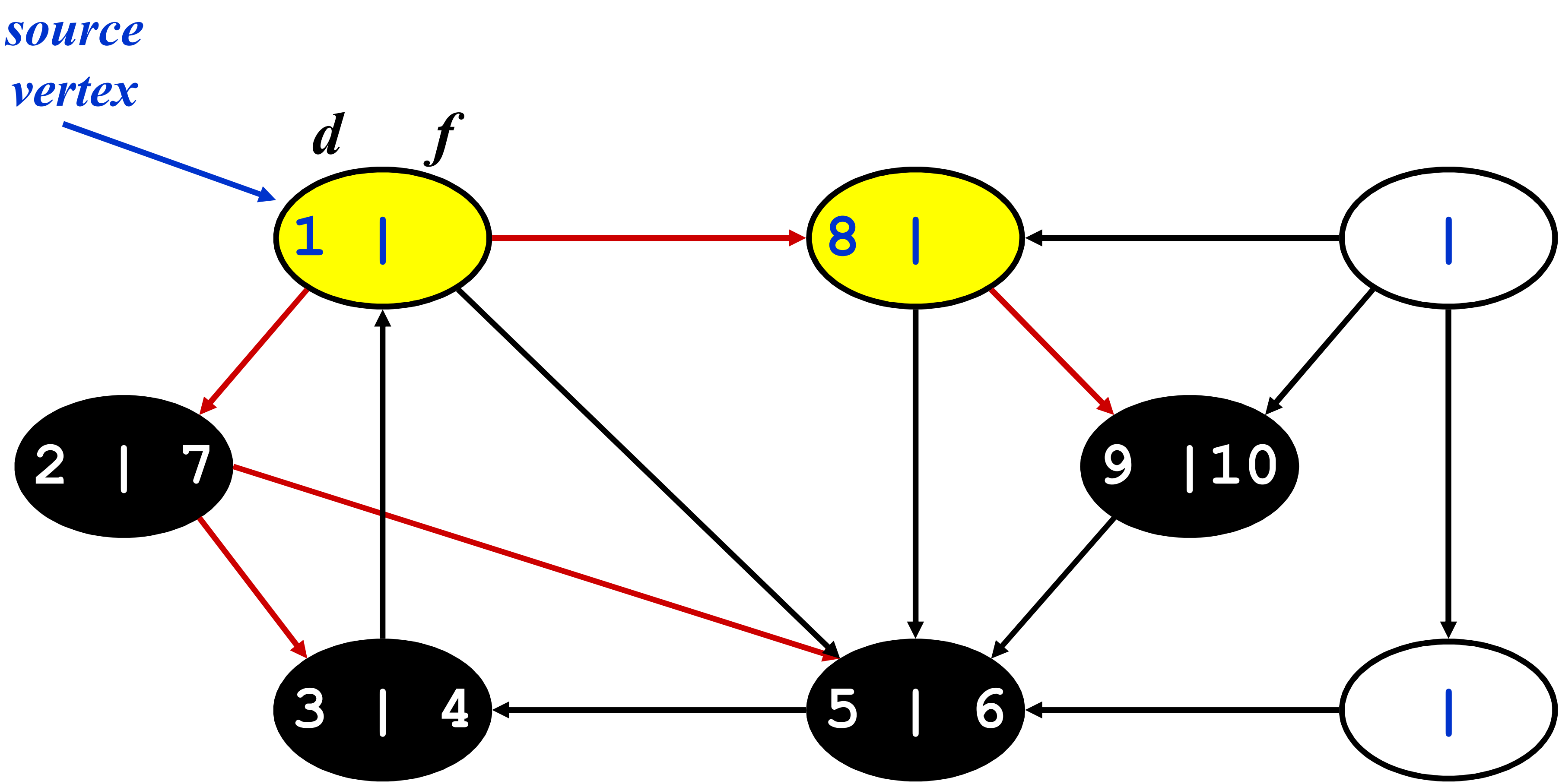
DFS Example



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

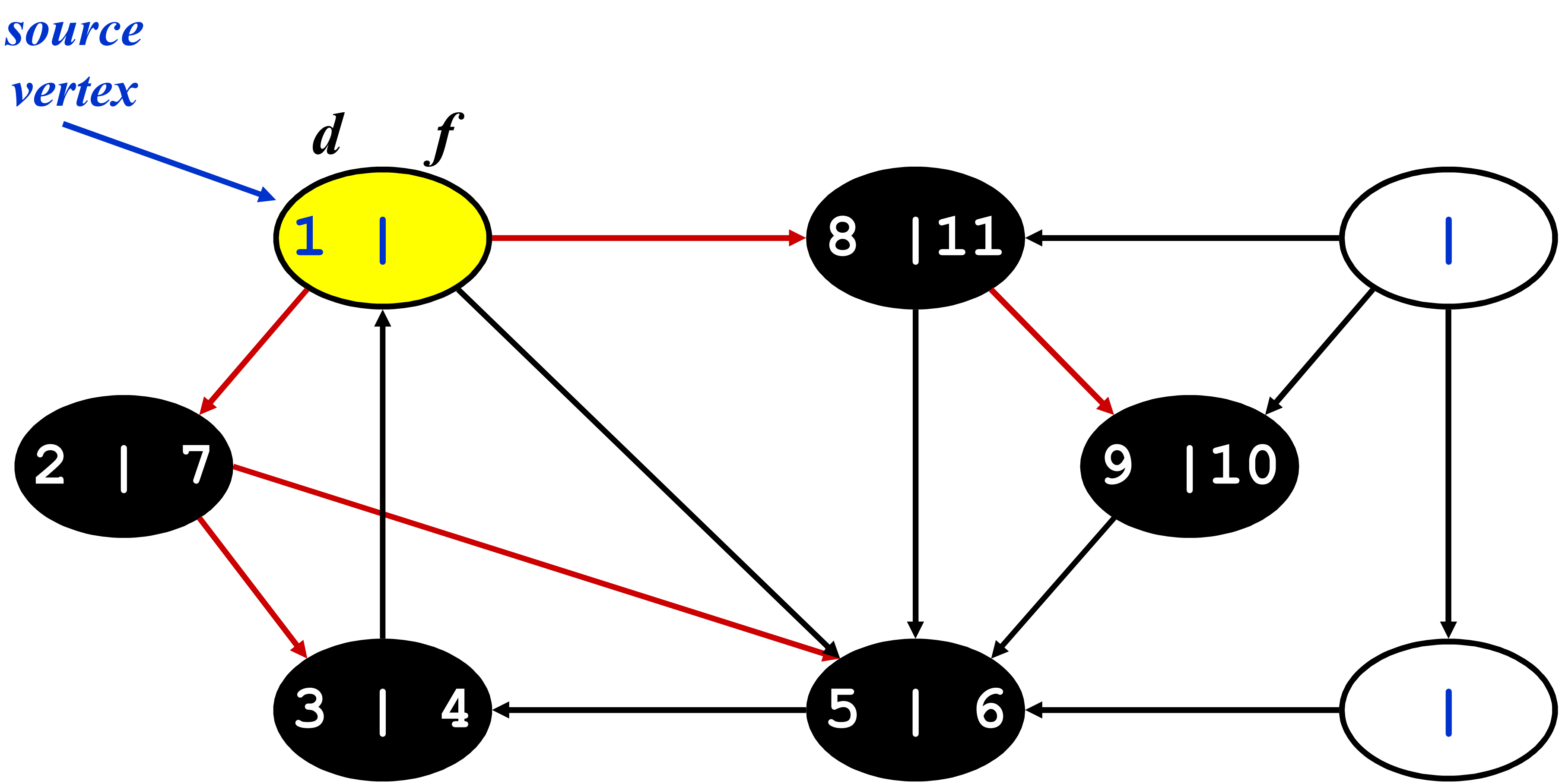
Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

DFS Example

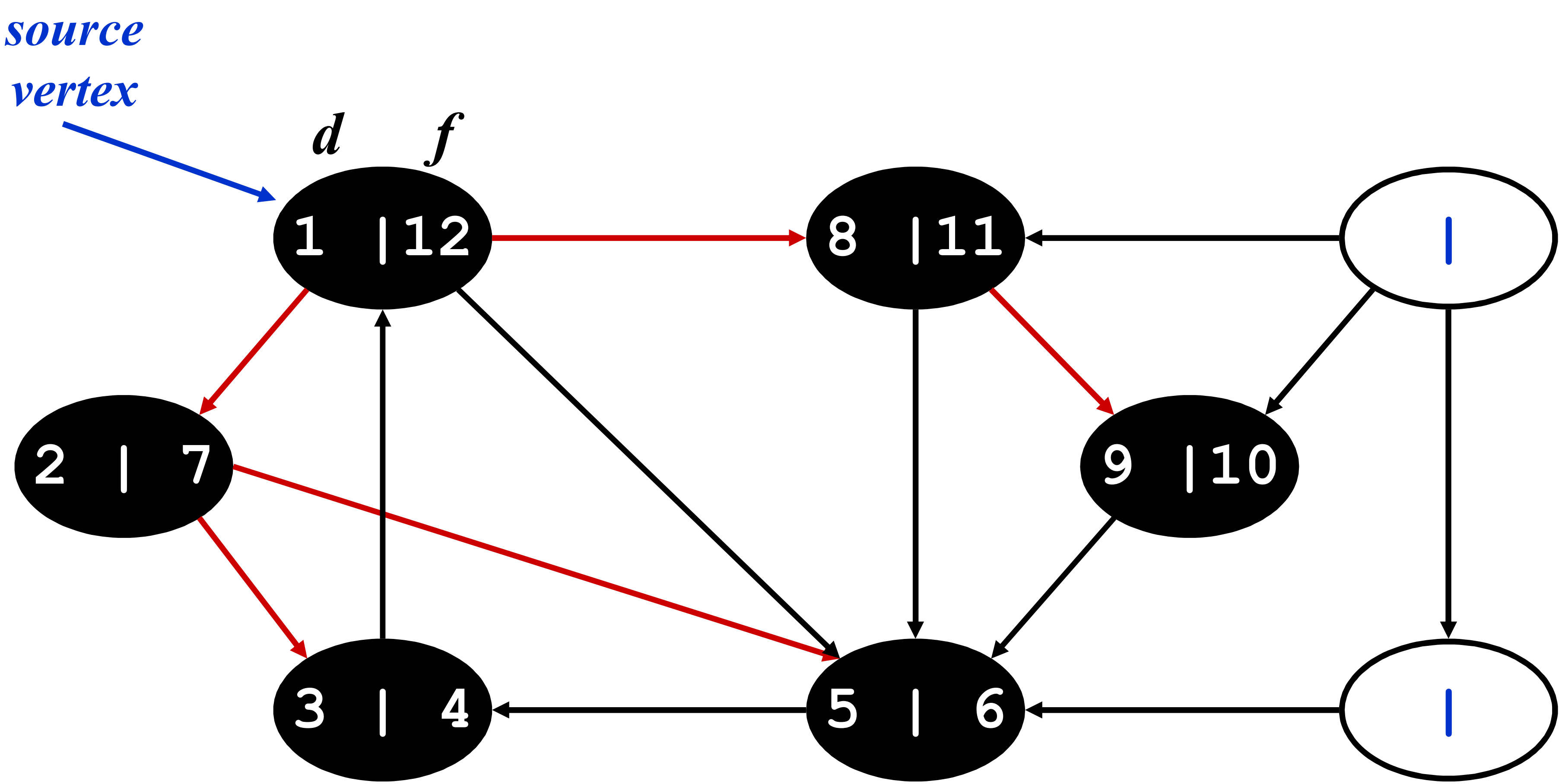


Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

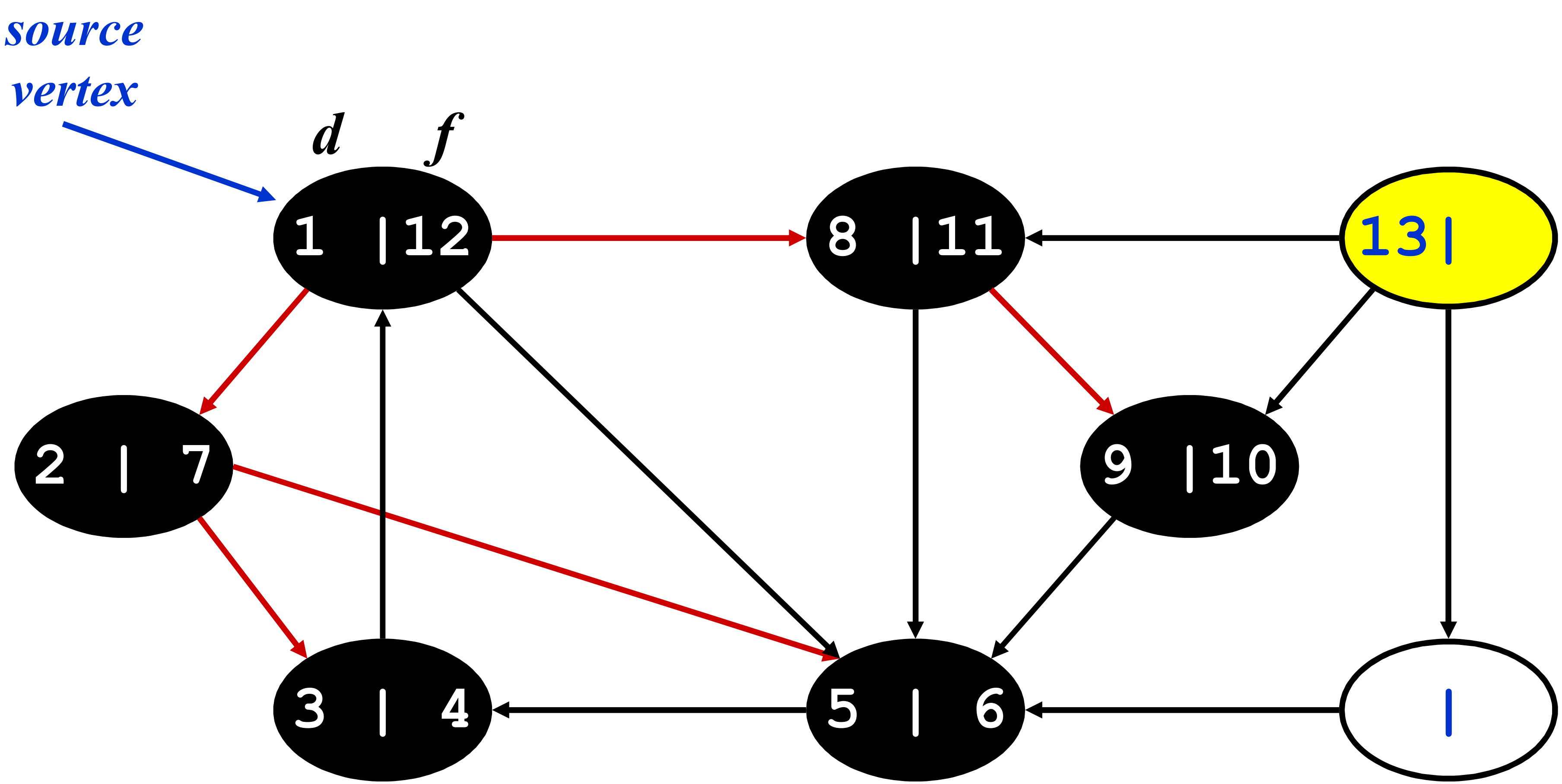
DFS Example



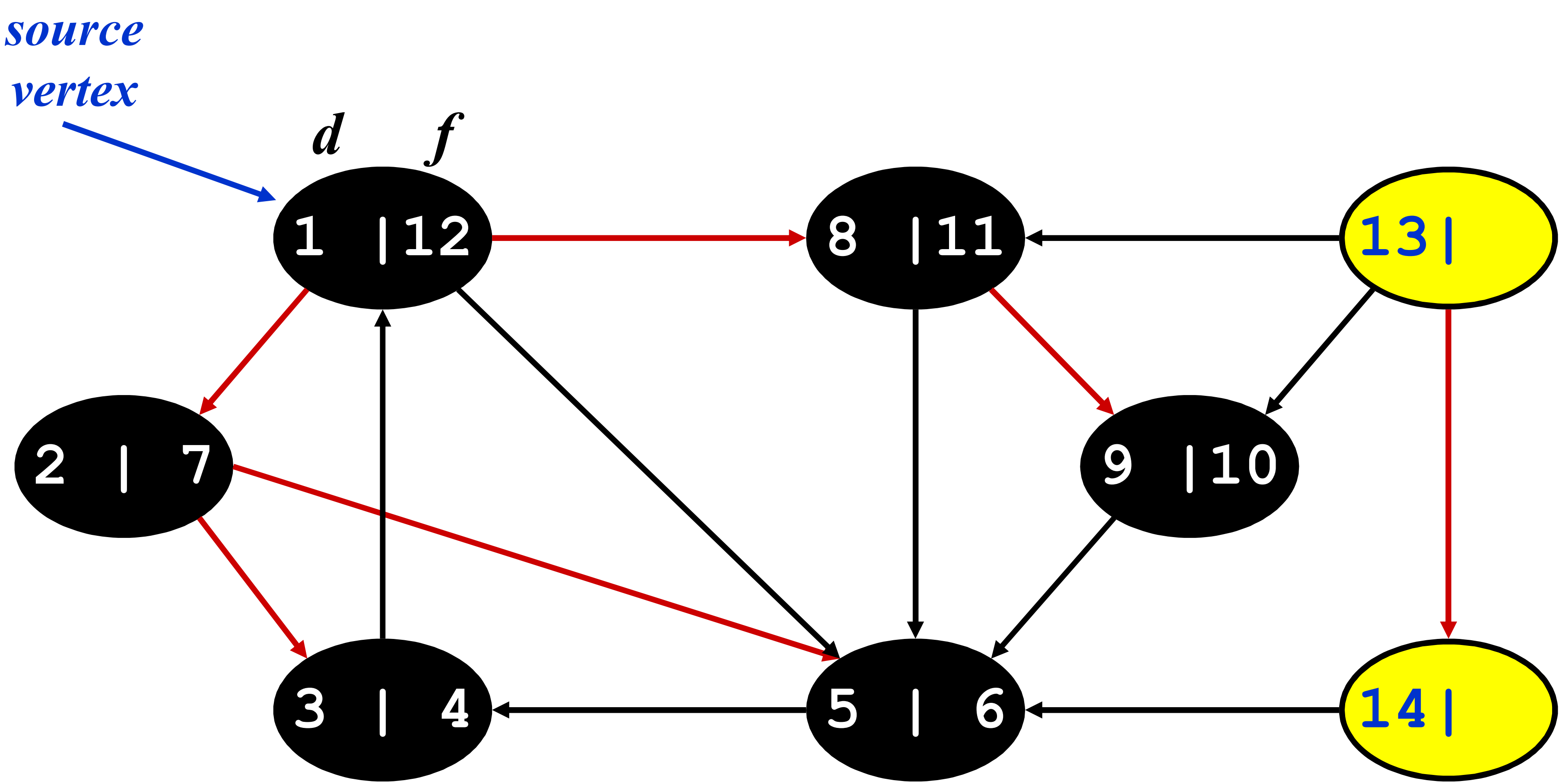
DFS Example



DFS Example

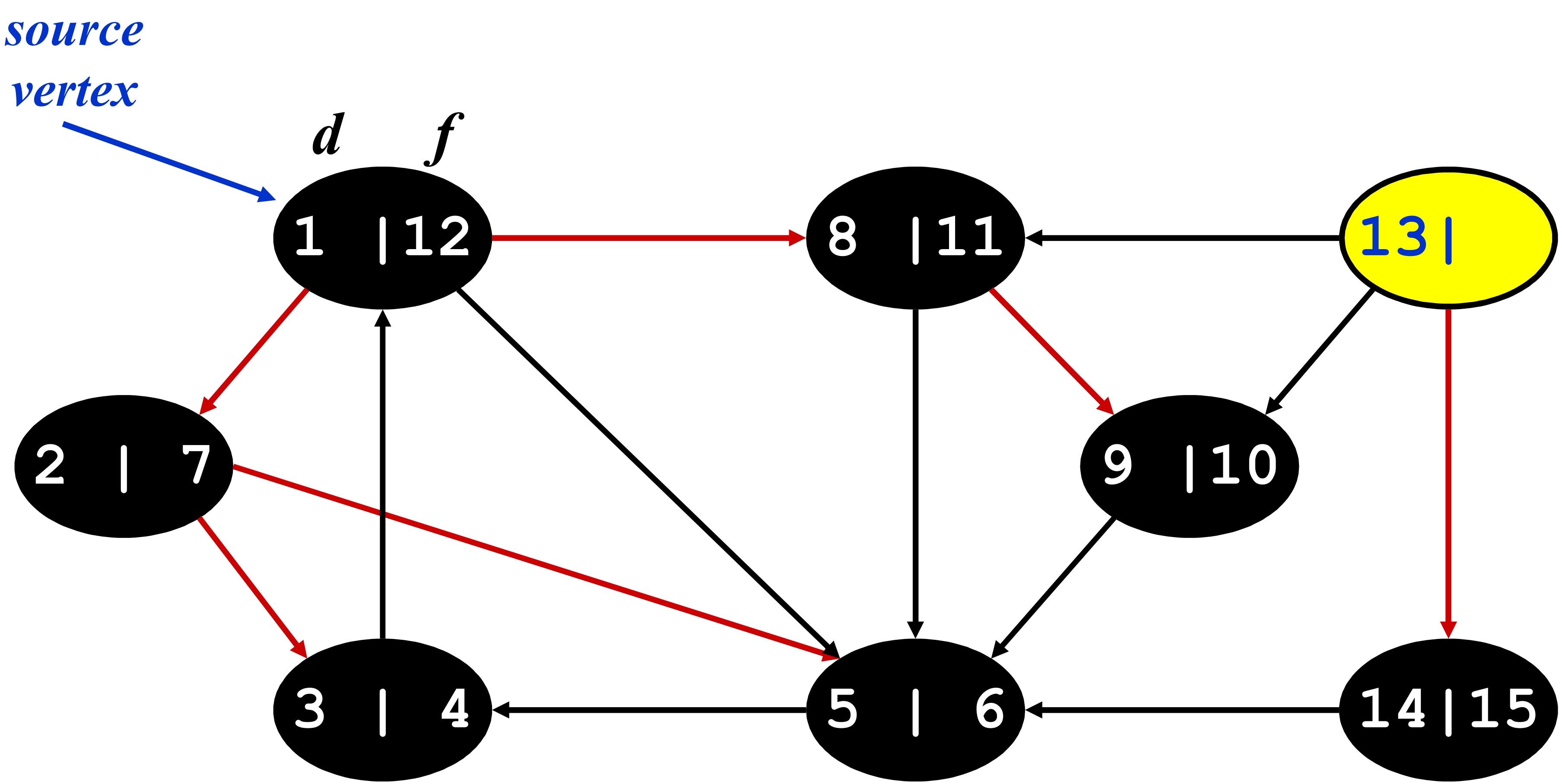


DFS Example



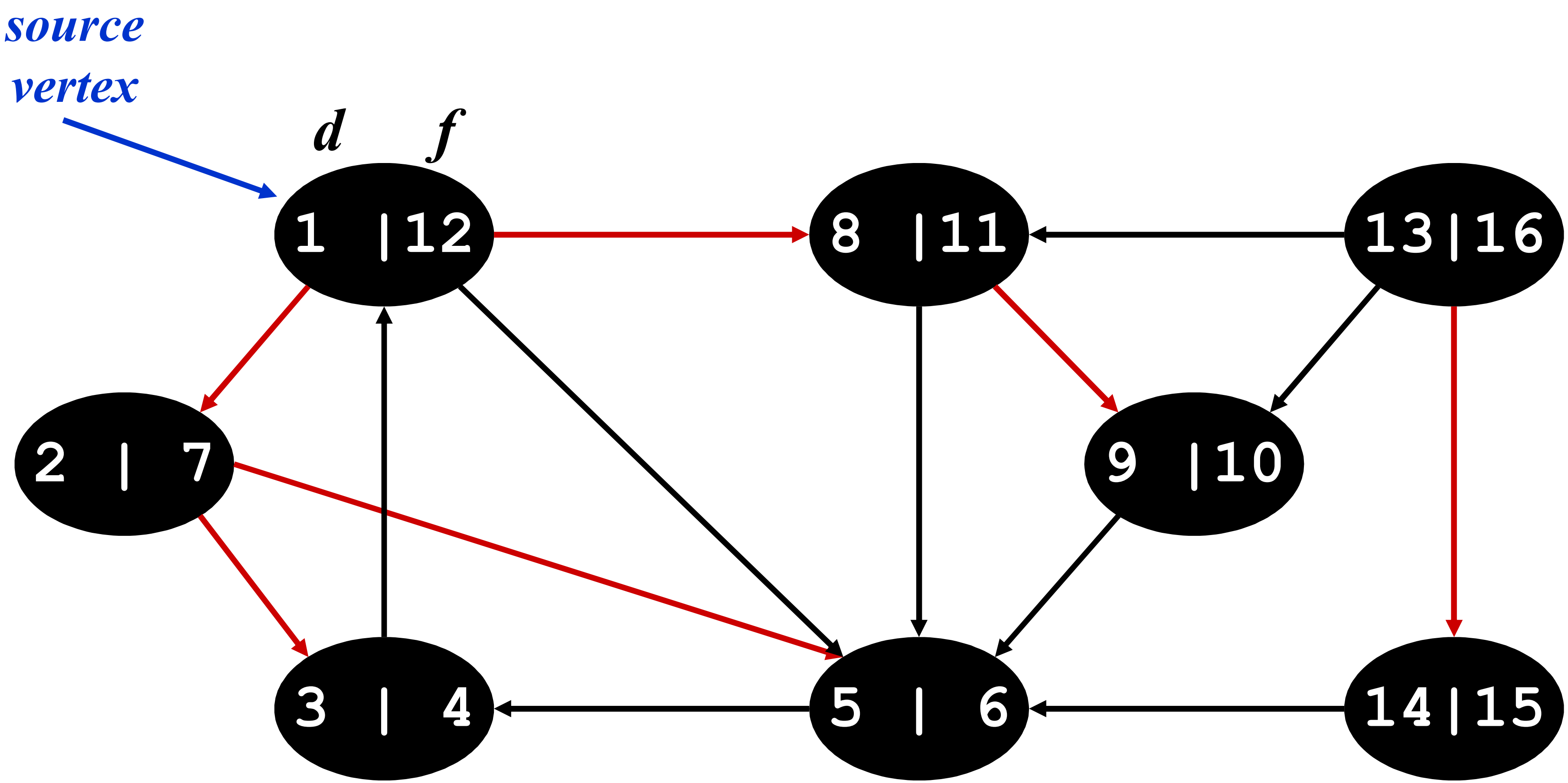
Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

DFS Example



Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

DFS Example



Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Depth-First Search: The 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 = 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;
}
```

Will all vertices eventually be colored black?

What will be the running time?

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Depth-First Search: The 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 = 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;
}
```

Running time: $O(n^2)$ because call DFS_Visit on each vertex, and the loop over Adj[] can run as many as |V| times

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Depth-First Search: The 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 = 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;
}
```

BUT, there is actually a tighter bound.

How many times will DFS_Visit() actually be called?

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Depth-First Search: The 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 = 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;
}
```

So, running time of DFS = O(V+E)

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

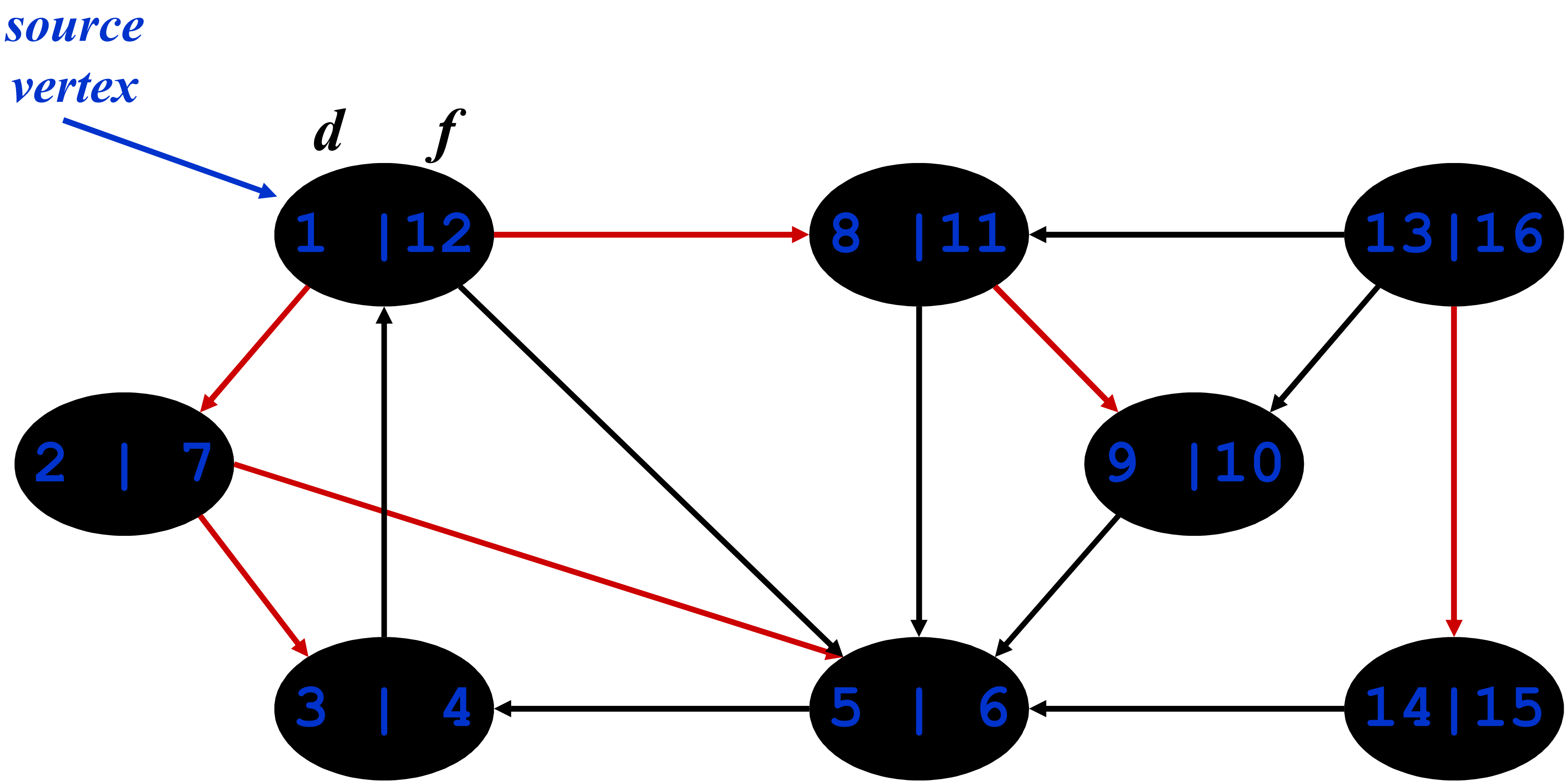
Depth-First Sort Analysis

- This running time argument is an informal example of *amortized analysis*
  - "Charge" the exploration of edge to the edge:
    - Each loop 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)
      - Considered linear for graph, b/c adj list requires O(V+E) storage
  - Important to be comfortable with this kind of reasoning and analysis

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

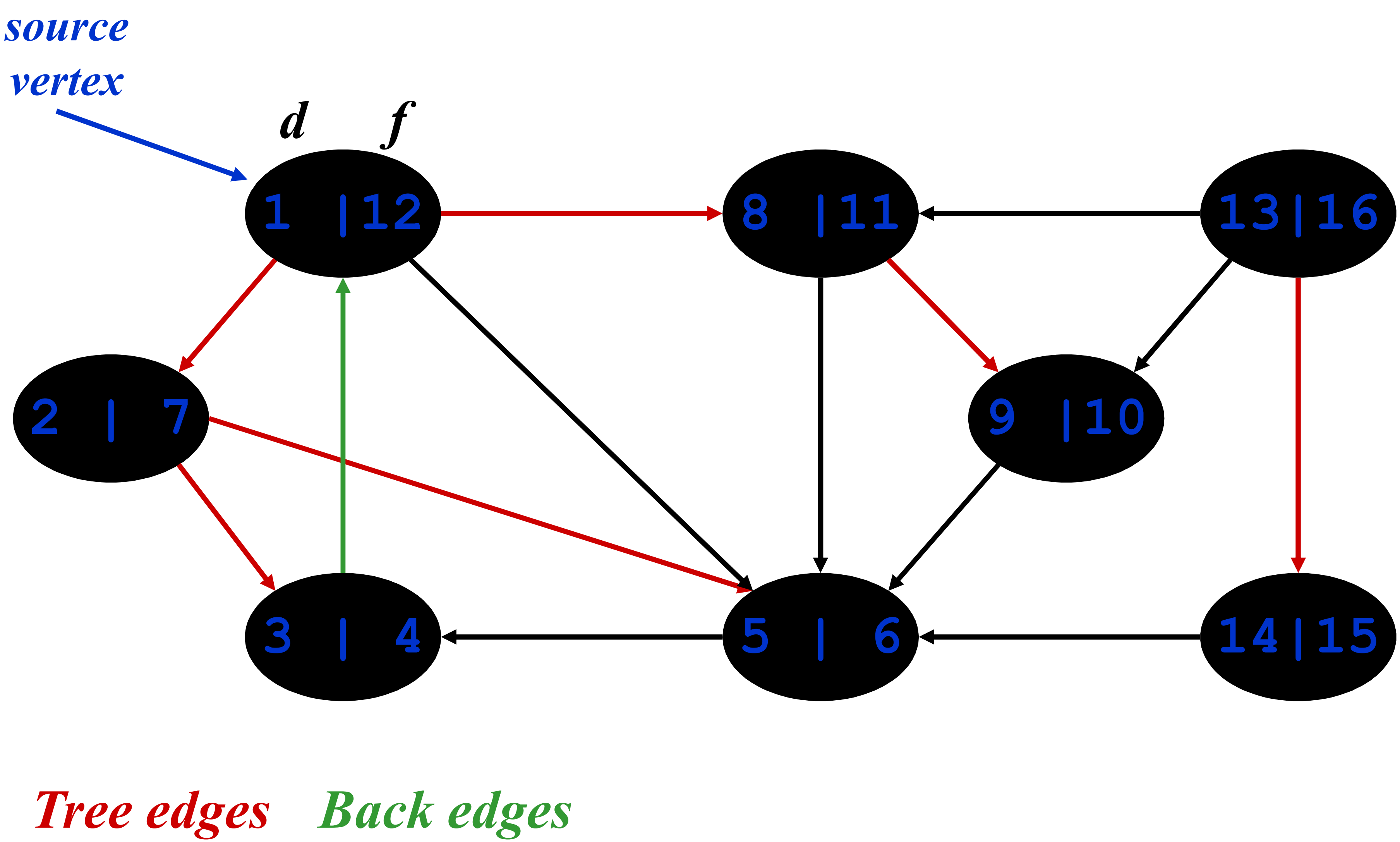
Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

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)

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

DFS Example

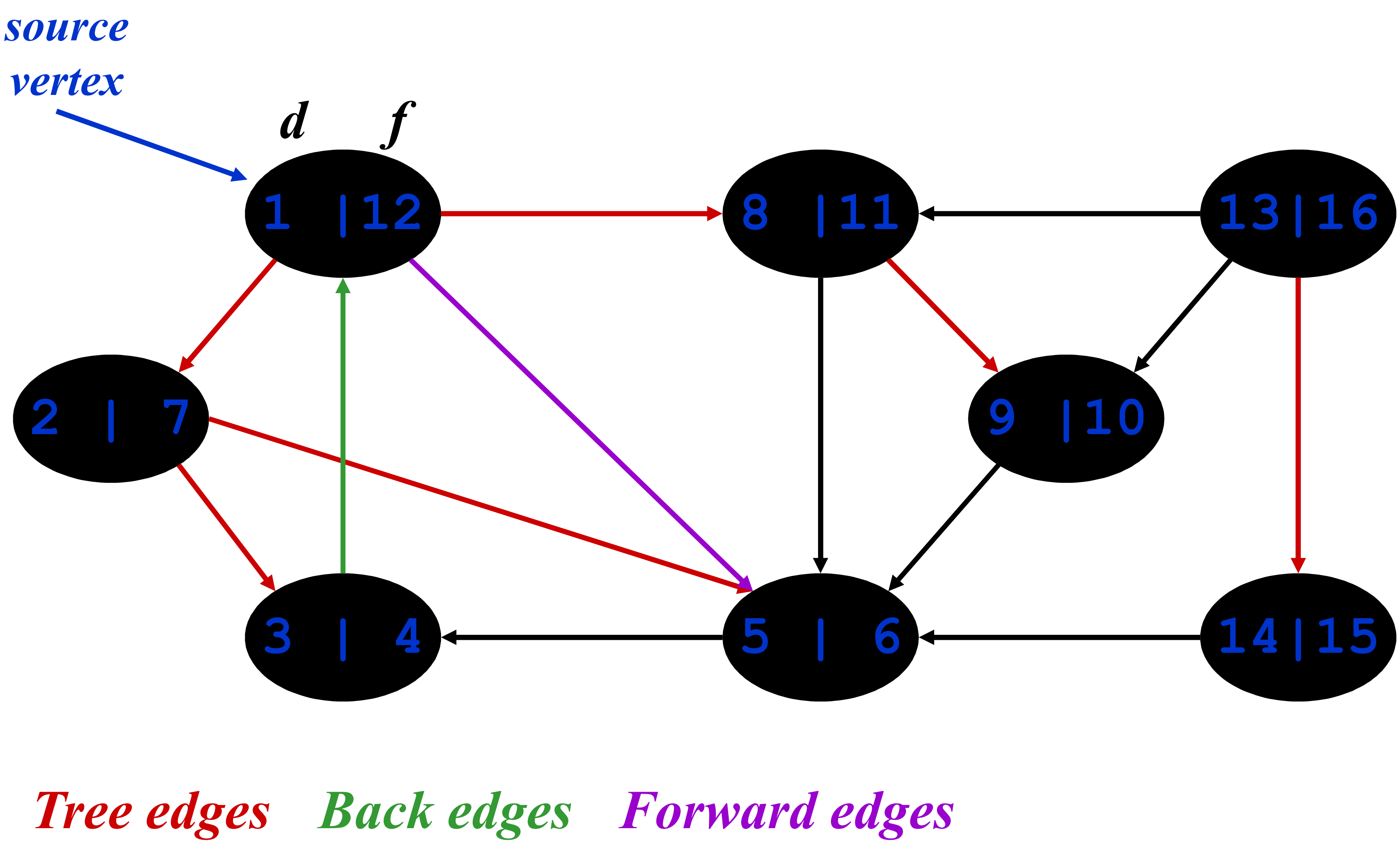


Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

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

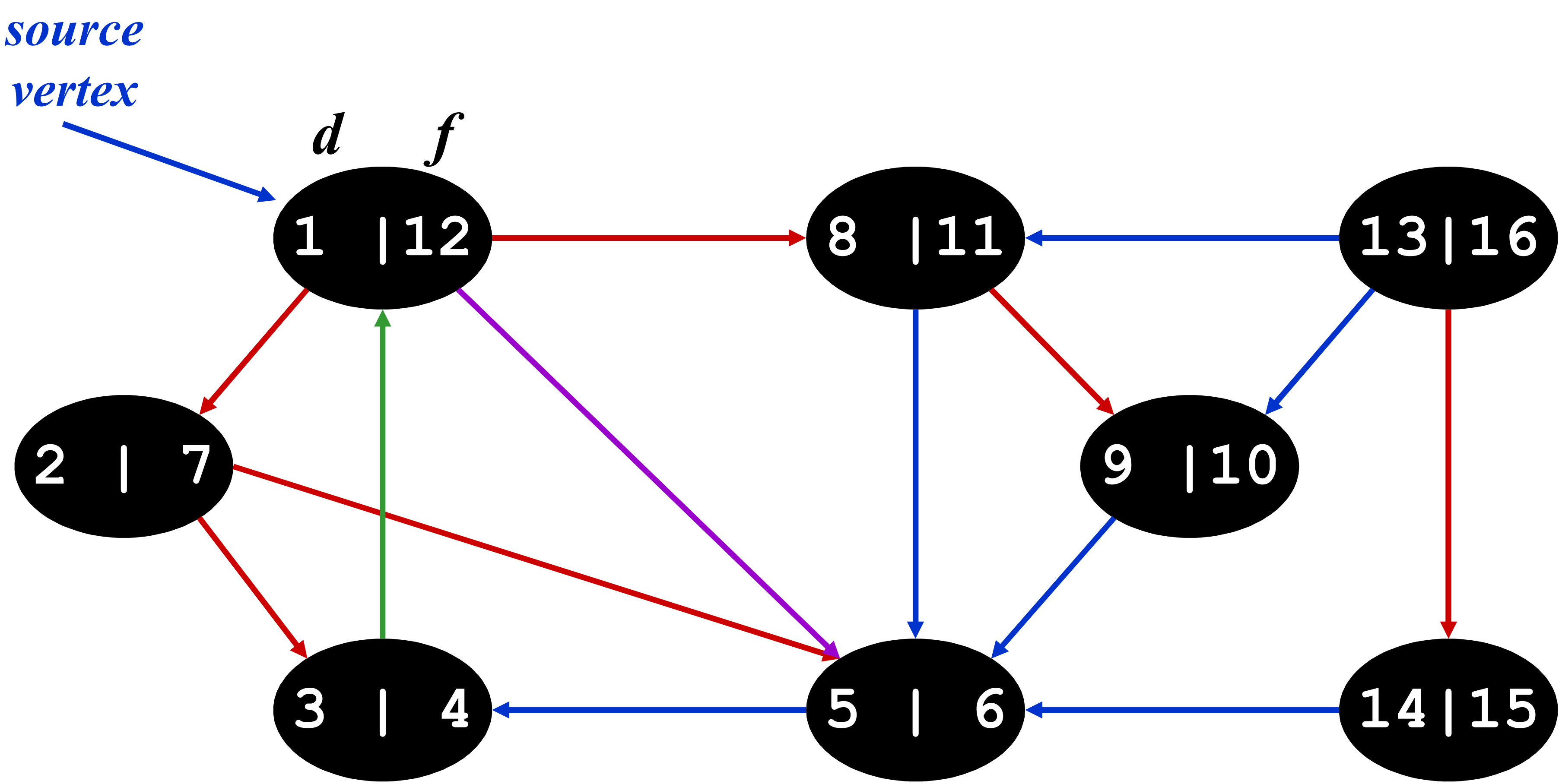


Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

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*

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

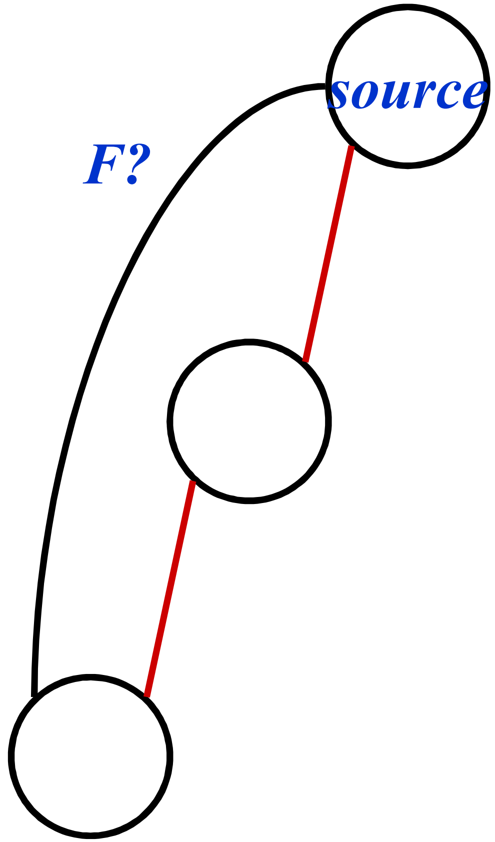
DFS: Kinds of edges

- 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

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

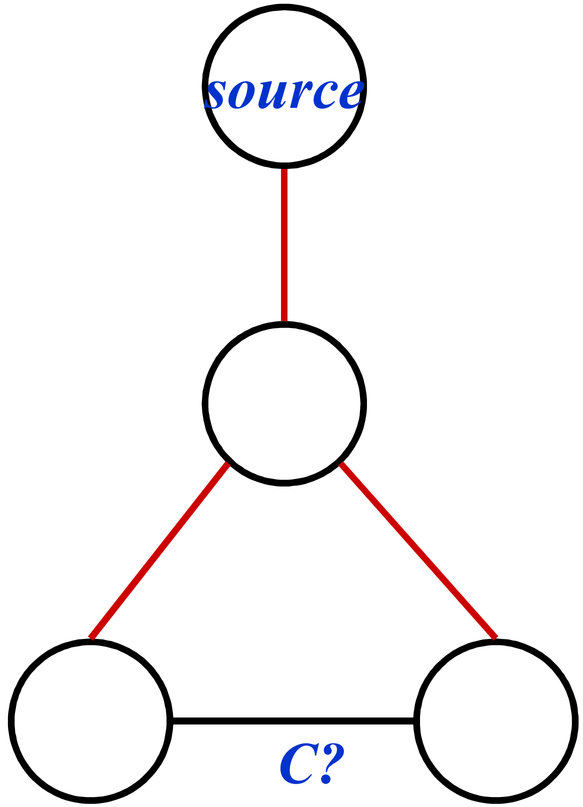
DFS: Kinds Of Edges

- 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* iff a DFS yields no back edges
  - If acyclic, no back edges (because a back edge implies a cycle
  - If no back edges, acyclic
    - No back edges implies only tree edges (*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

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

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;
}
```

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

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

DFS And Cycles

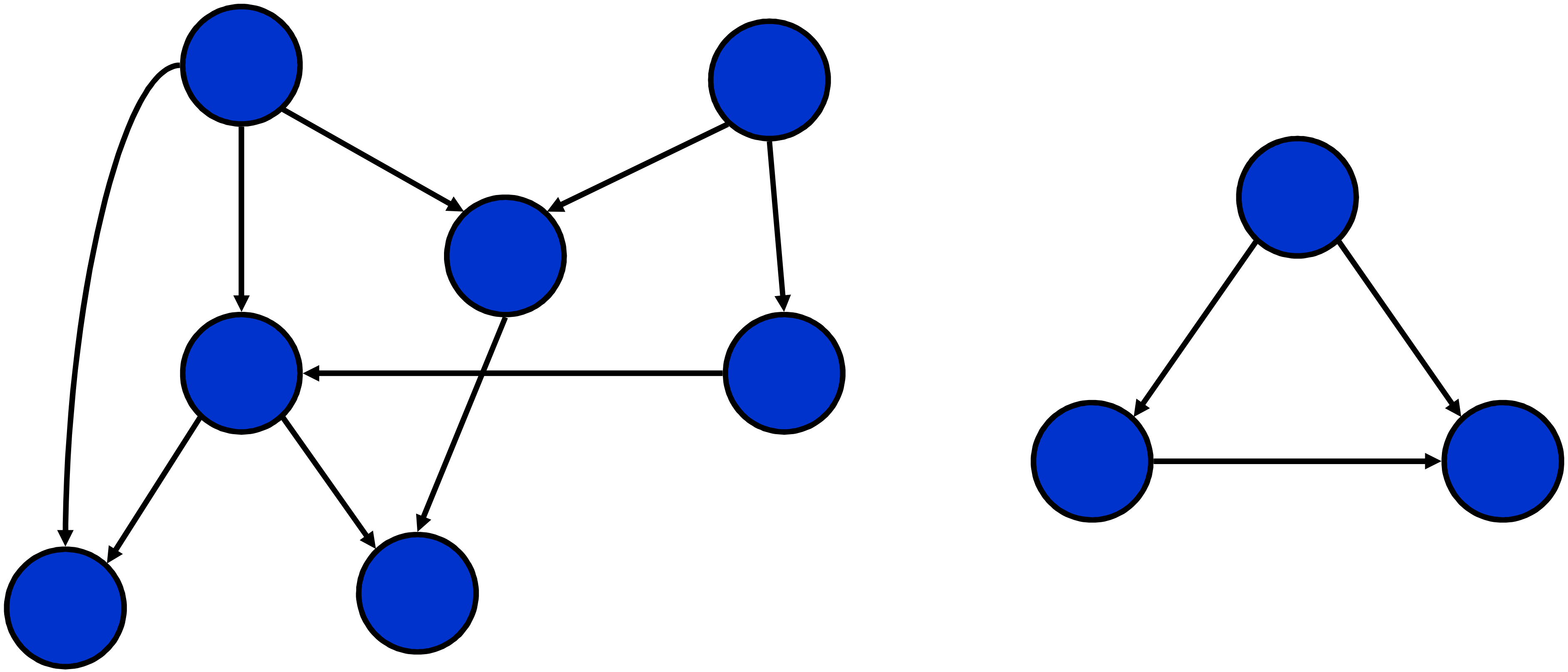
- Running time: O(V+E)
- We can actually determine if cycles exist in O(V) time:
  - In an undirected acyclic forest, $|E| \leq |V| - 1$
  - So count the edges: if ever see $|V|$ distinct edges, must have seen a back edge along the way
  - *Why not just test if $|E| < |V|$ and answer the question in constant time?*

DIRECTED ACYCLIC GRAPHS AND TOPOLOGICAL SORTING

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Directed Acyclic Graphs

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



Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

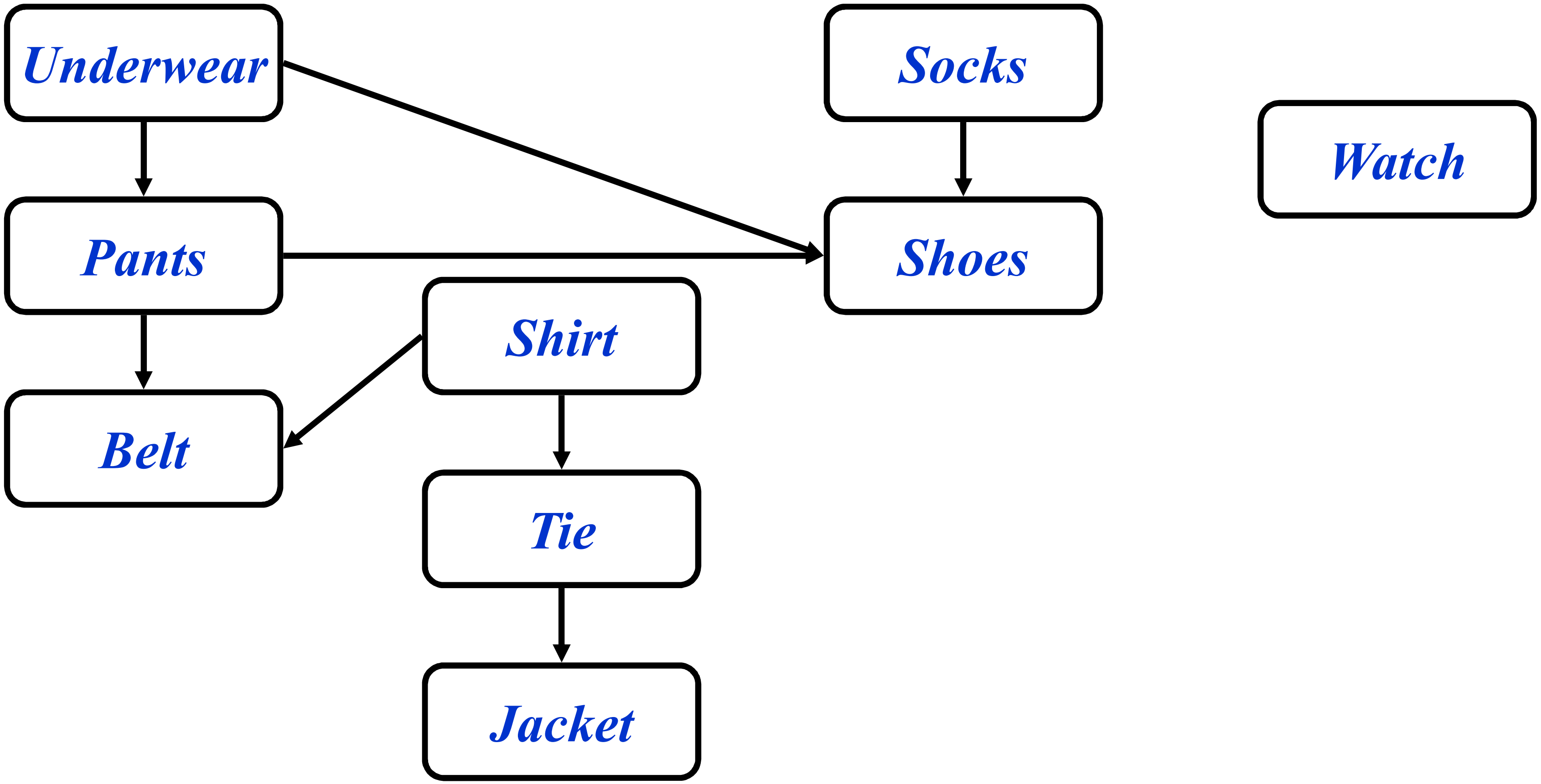
DFS and DAGs

- Argue that a directed graph G is acyclic iff a DFS of G yields no back edges:
  - Forward: if G is acyclic, will be no back edges
    - Trivial: a back edge implies a cycle
  - Backward: if no back edges, G is acyclic
    - Argue contrapositive: G has a cycle $\Rightarrow \exists$ 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→v is yellow→yellow, thus (*u*, *v*) is a back edge

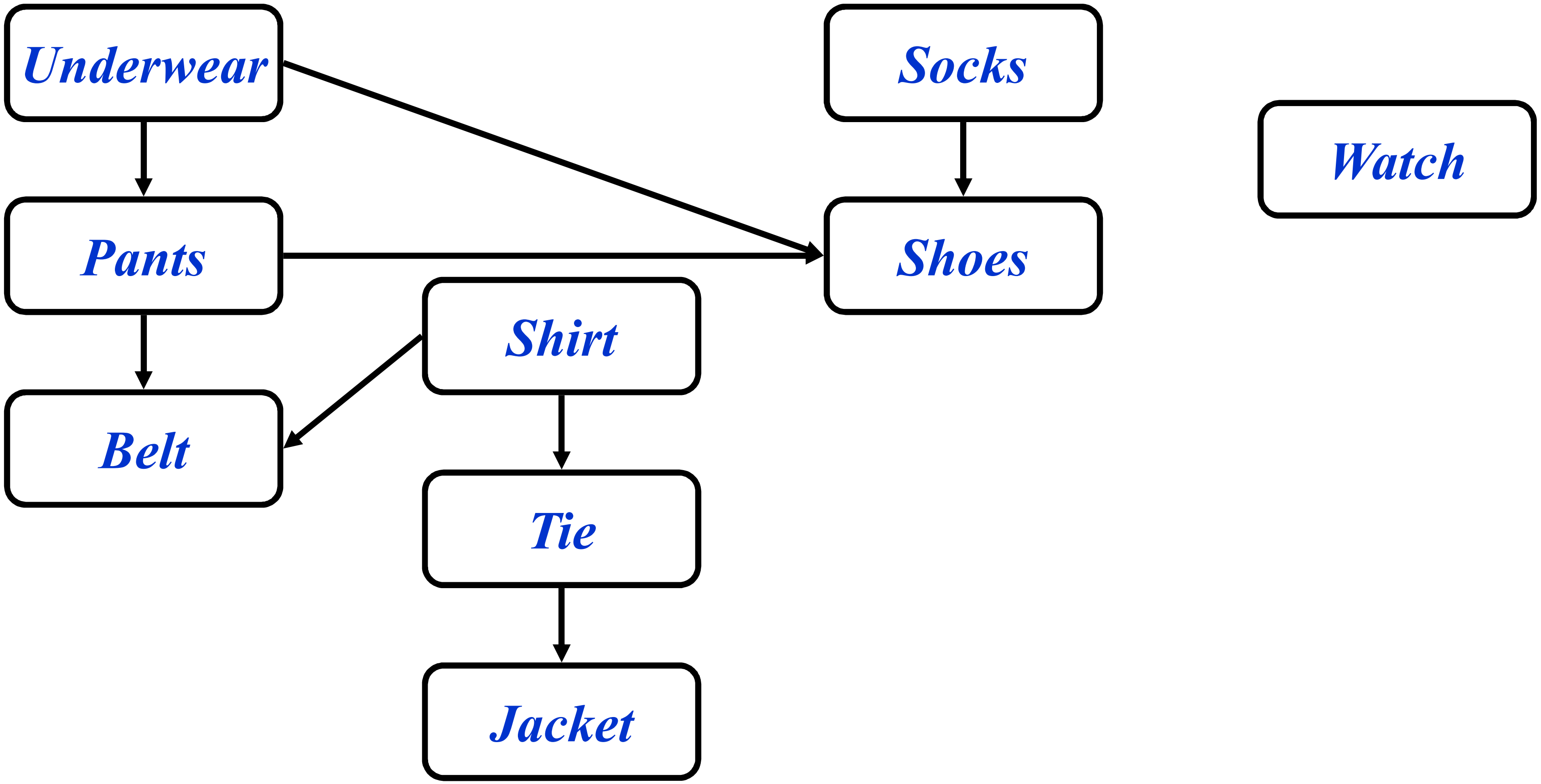
Topological Sort

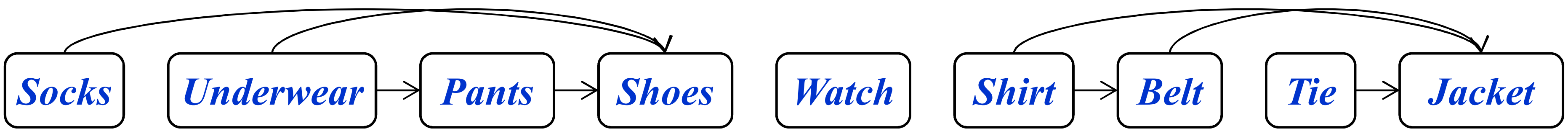
- *Topological sort* of a DAG:
  - Linear ordering of all vertices in graph G such that vertex *u* comes before vertex *v* if edge $(u, v) \in G$
- Real-world example: getting dressed

Getting Dressed



Getting Dressed





Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Topological Sort Algorithm

```
Topological-Sort()
{
    Run DFS
    When a vertex is finished, output it
    Vertices are output in reverse
      topological order
}
```

- Time: O(V+E)
- Correctness: Want to prove that

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

Correctness of Topological Sort

- Claim: $(u,v) \in G \Rightarrow u \rightarrow f > v \rightarrow f$
  - When (u,v) is explored, u is yellow
    - v = yellow $\Rightarrow$ (u,v) is back edge. Contradiction (*Why?*)
    - v = white $\Rightarrow$ v becomes descendent of $u \Rightarrow v \rightarrow f < u \rightarrow f$ (since must finish v before backtracking and finishing u)
    - v = black $\Rightarrow$ v already finished $\Rightarrow v \rightarrow f < u \rightarrow f$

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC

Shortest path search

NEXT TOPICS

Next topic: Finding Shortest Paths

END OF LECTURE 13

Slide credit: 홍석원, 명지대학교; 김한준, 서울시립대학교; J. Lillis, UIC