CS146: Data Structures and Algorithms
Lecture 14

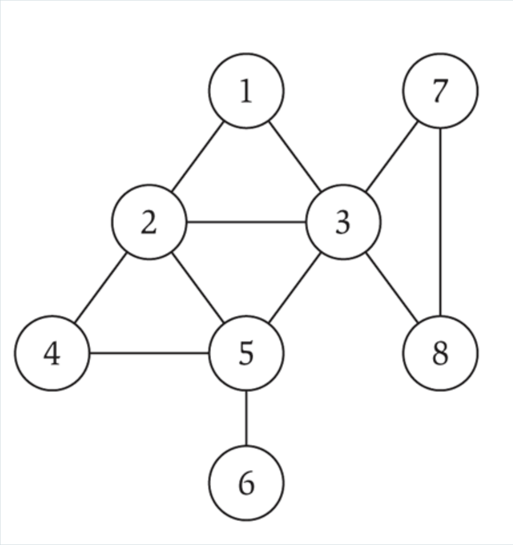
GRAPHS

INSTRUCTOR: KATERINA POTIKA
CS SJSU

Basic Definitions and Applications

Undirected Graphs

- Undirected graph. G = (V, E)
  - V = nodes.
  - E = edges between pairs of nodes.
  - Captures pairwise relationship between objects.
  - Graph size parameters: n = |V|, m = |E|.



V = { 1, 2, 3, 4, 5, 6, 7, 8 }

E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }

n = 8

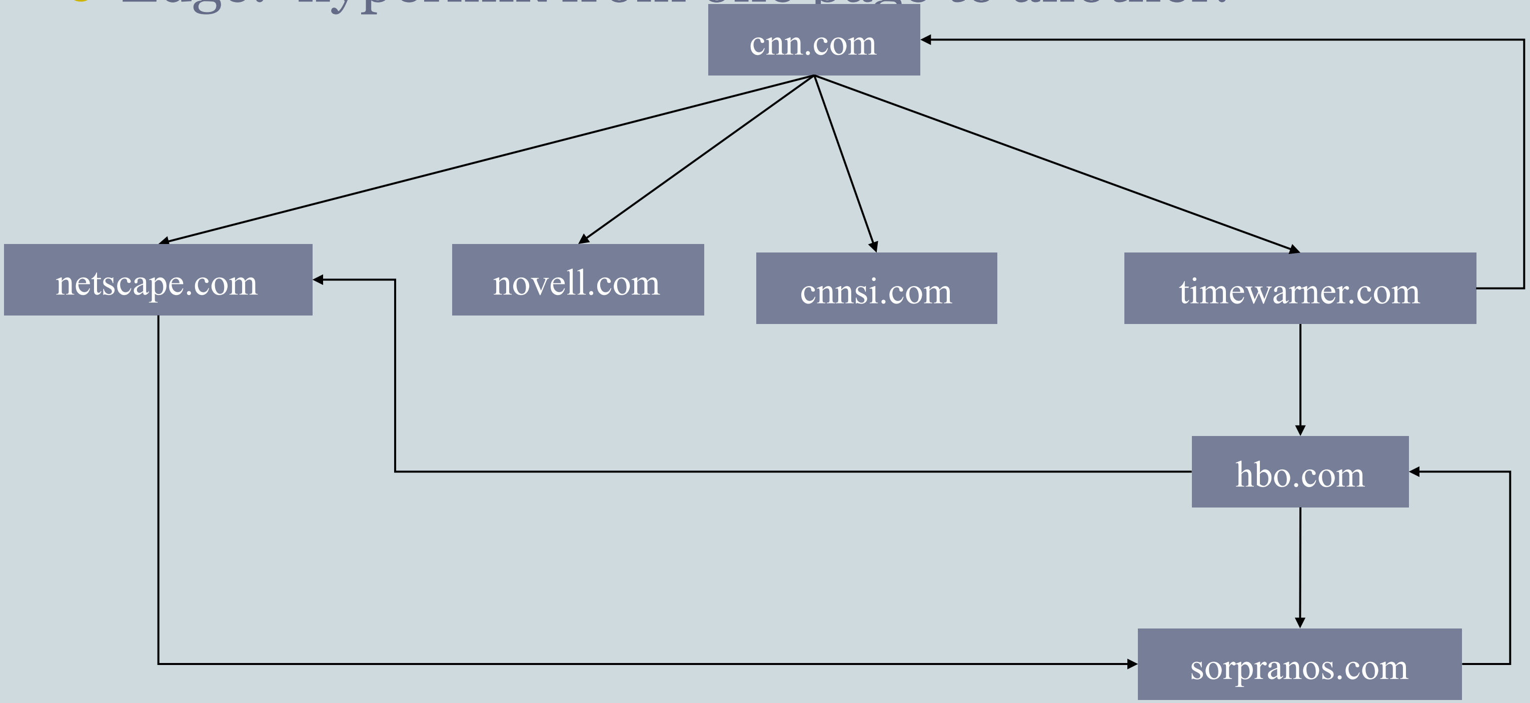
m = 11

Some Graph Applications

Graph	*Nodes*	*Edges*
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

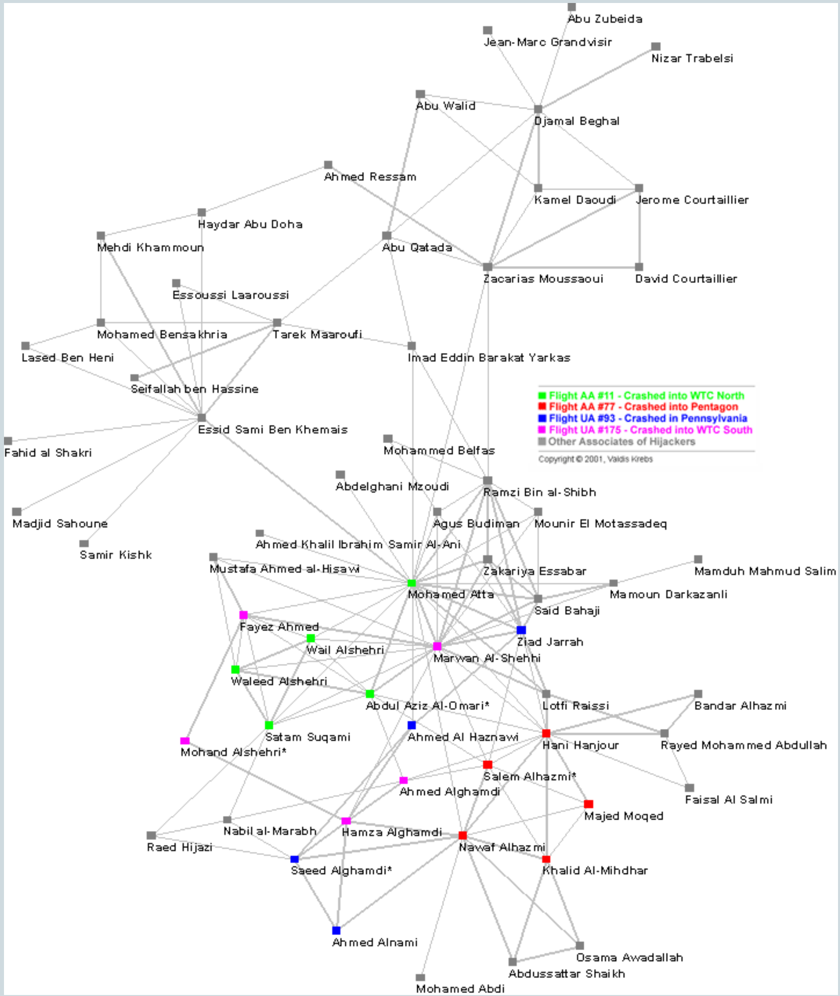
World Wide Web

- Web graph.
  - Node: web page.
  - Edge: hyperlink from one page to another.



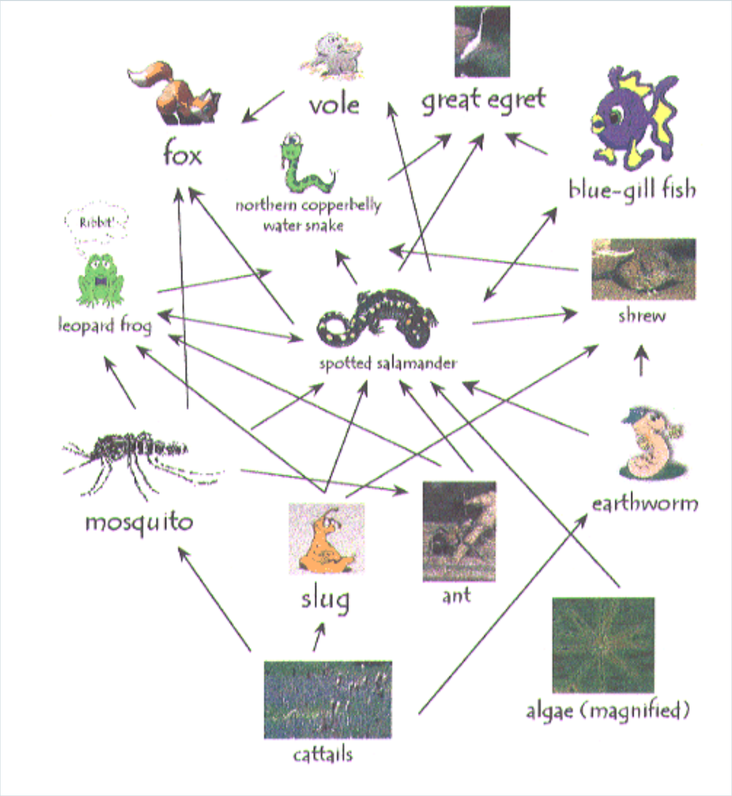
9-11 Terrorist Network

- Social network graph.
  - Node: people.
  - Edge: relationship between two people.



Ecological Food Web

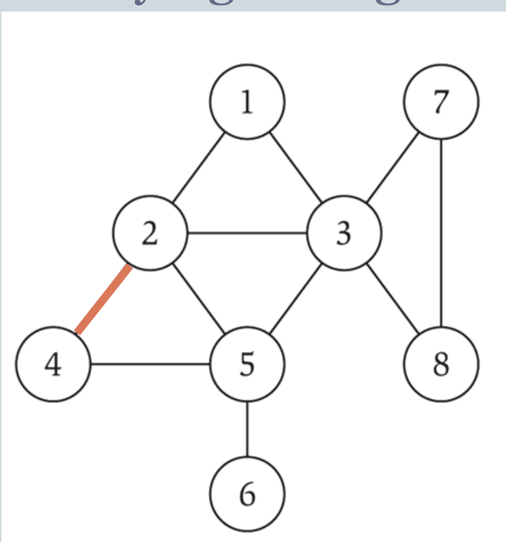
- Food web graph.
  - Node = species.
  - Edge = from prey to predator.



Reference: http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

Graph Representation: Adjacency Matrix

- Adjacency matrix. n*n matrix with $A_{uv} = 1$ if (u, v) is an edge.
  - Two representations of each edge.
  - Space proportional to n^2.
  - Checking if (u, v) is an edge takes $\Theta(1)$ time.
  - Identifying all edges takes $\Theta(n^2)$ time.



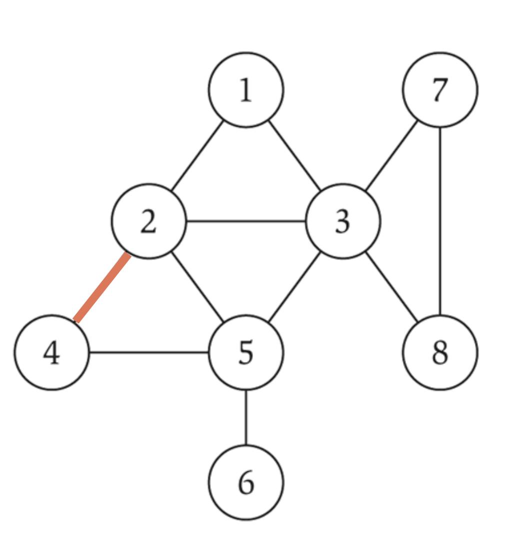
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

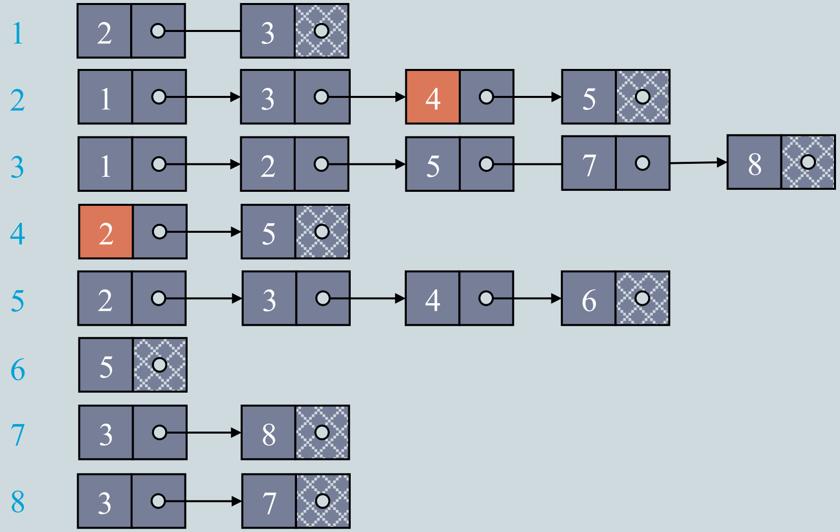
Adjacent: Two vertices connected by an edge are adjacent.

Graph Representation: Adjacency List

- Adjacency list. Node indexed array of lists.
  - Two representations of each edge.
  - Space proportional to m + n.
  - Checking if (u, v) is an edge takes O(deg(u)) time.
  - Identifying all edges takes Θ(m + n) time.

degree = number of neighbors of u



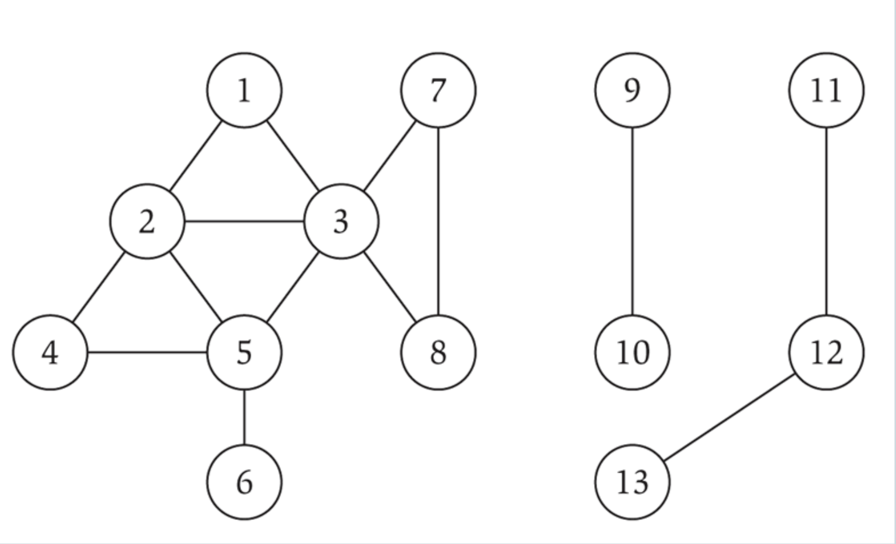


Paths and Connectivity

- Def. A path in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E.

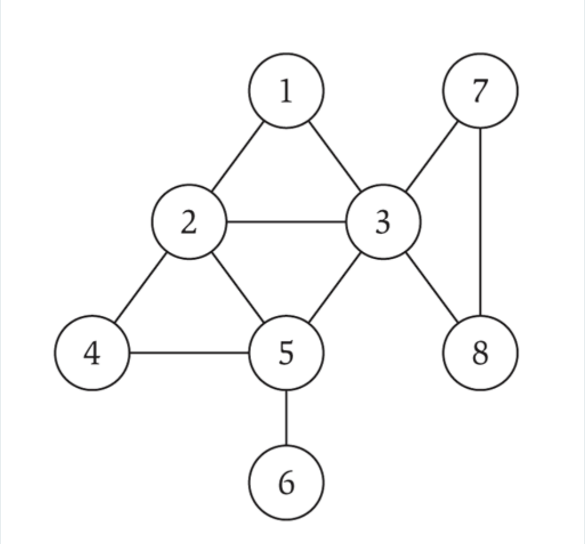
- Def. A path is simple if all nodes are distinct.

- Def. An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v. Otherwise, it is disconnected.



Cycles

- Def. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first k-1 nodes are all distinct.

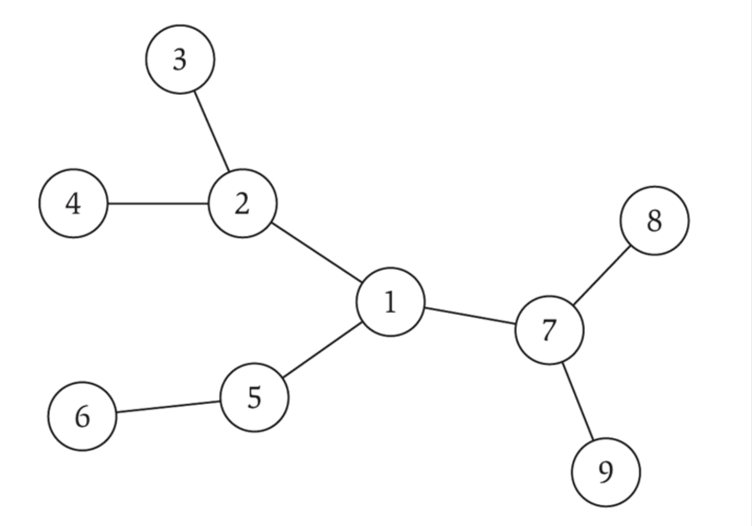


cycle C = 1-2-4-5-3-1

Trees

- Def. An undirected graph is a tree if it is connected and does not contain a cycle.

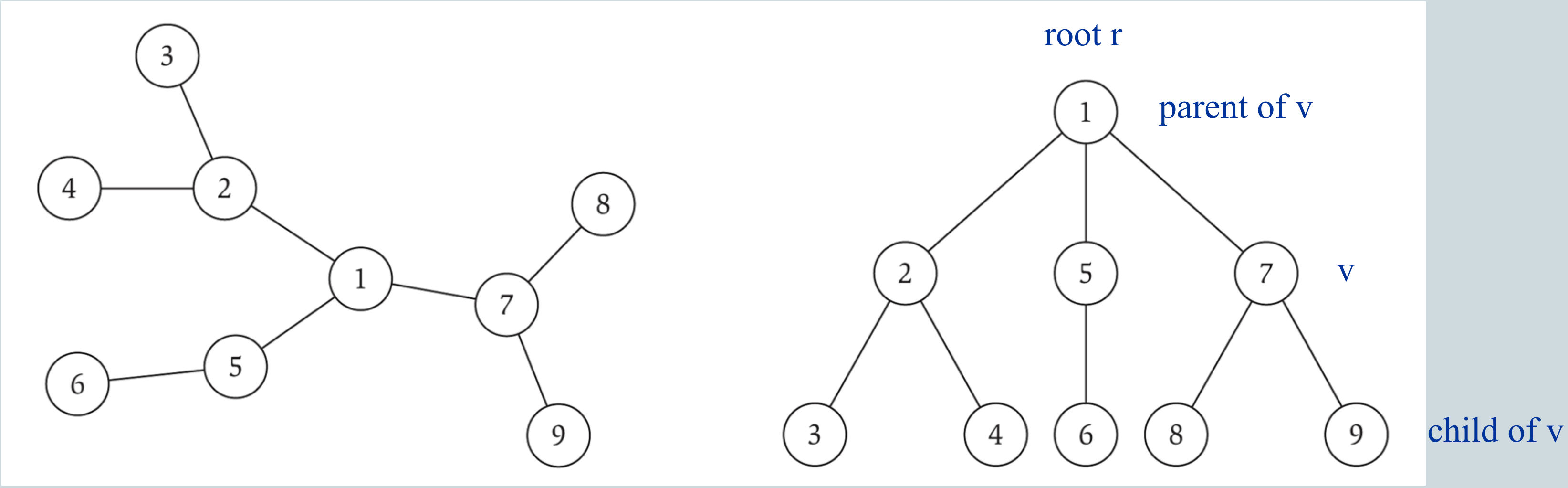
- Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.
  - G is connected.
  - G does not contain a cycle.
  - G has n-1 edges.



Rooted Trees

- Rooted tree. Given a tree T, choose a root node r and orient each edge away from r.

- Importance. Models hierarchical structure.

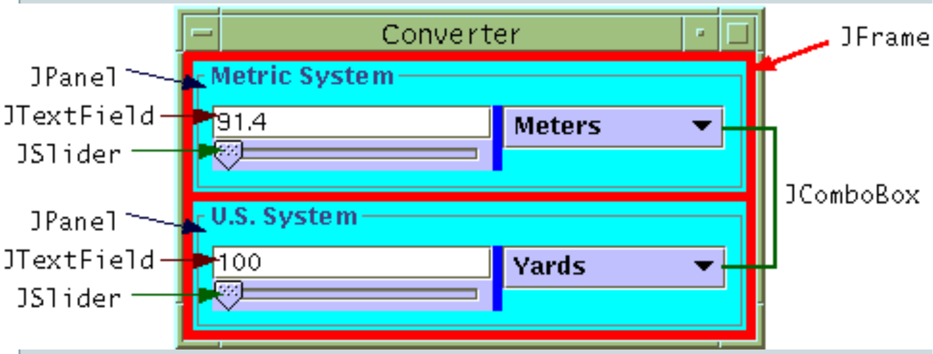


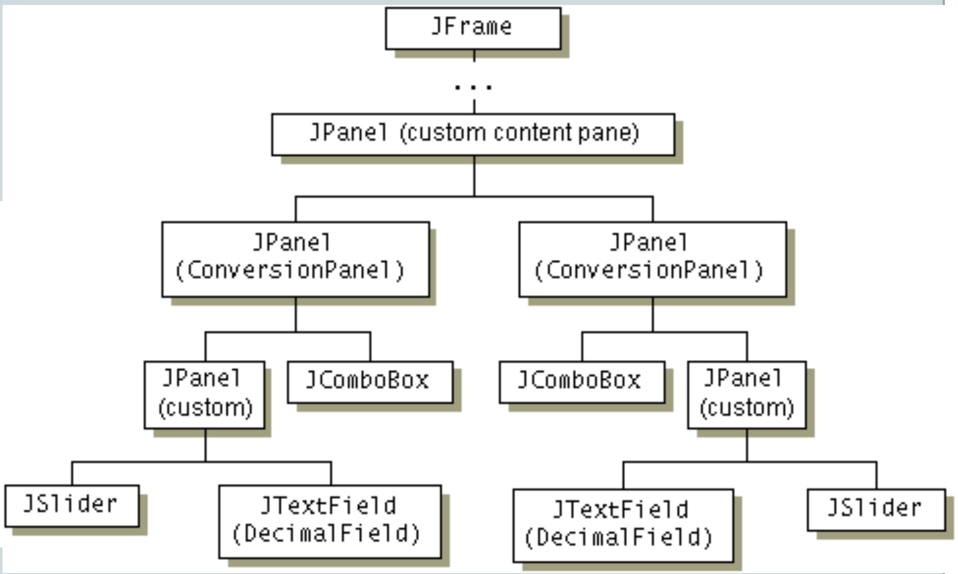
a tree

the same tree, rooted at 1

GUI Containment Hierarchy

- GUI containment hierarchy. Describe organization of GUI widgets.



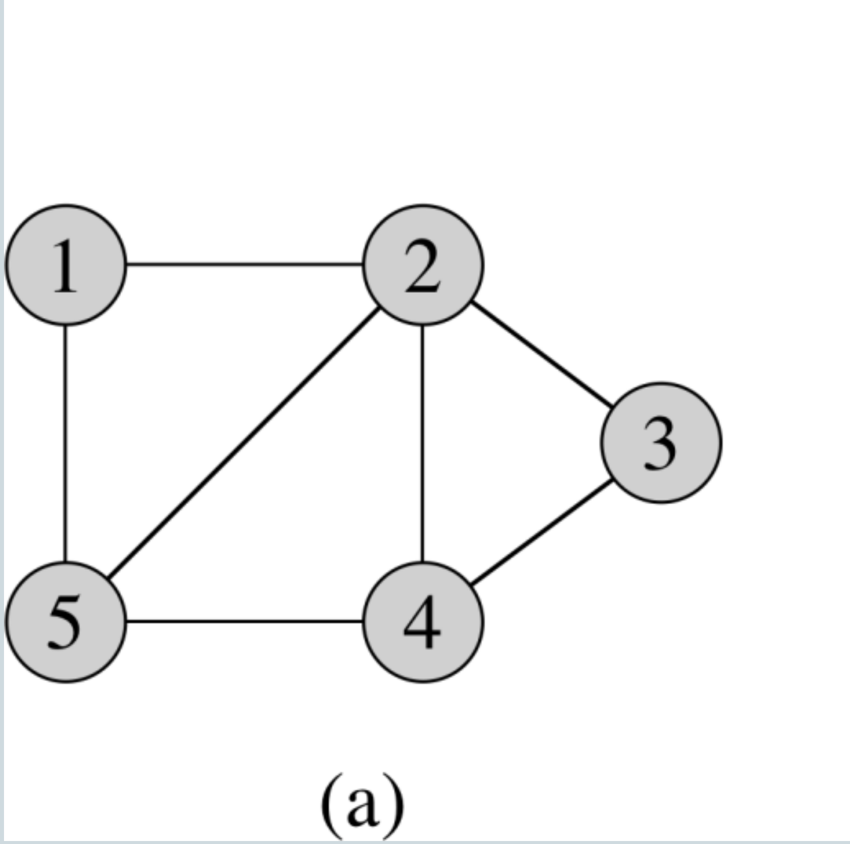
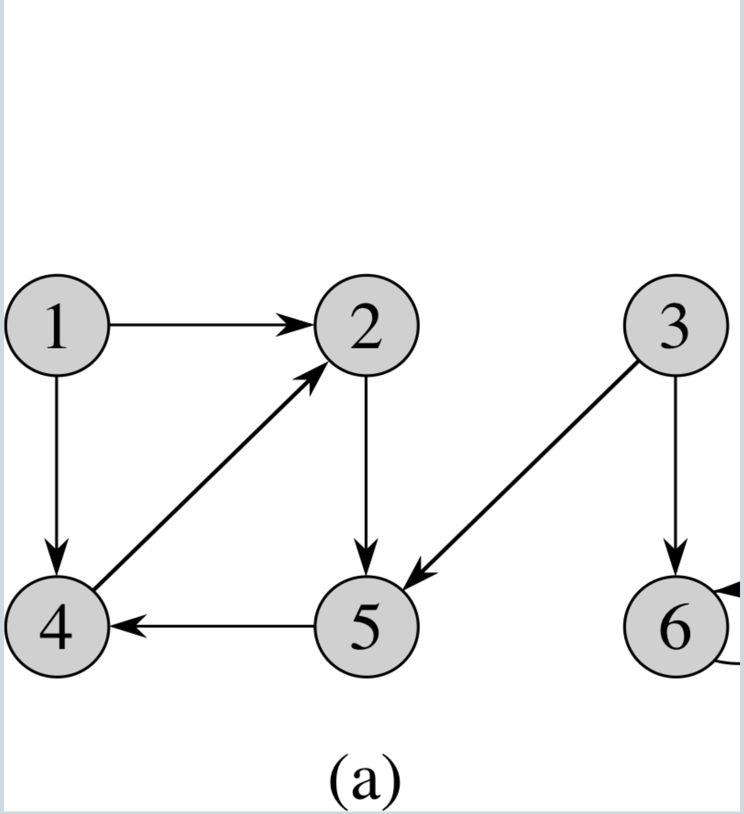


Reference: http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html

Graph Variations

- More variations:
  - Def. A *weighted graph* associates weights with either the edges or the vertices
    - E.g., a road map: edges might be weighted w/ distance
  - Def. A *simple graph* doesn't allow multiple edges or self-loops.
  - Def. 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)

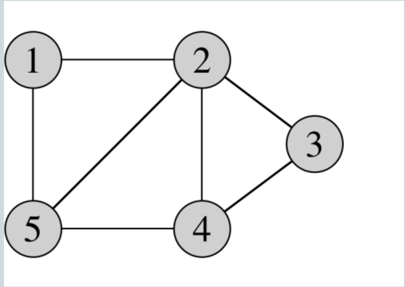
Undirected vs Directed Graph

Undirected	**Directed**
	

Degree of a vertex

Undirected Graphs

- *deg(v)*: The number of edges incident on it (loop at vertex is counted twice).

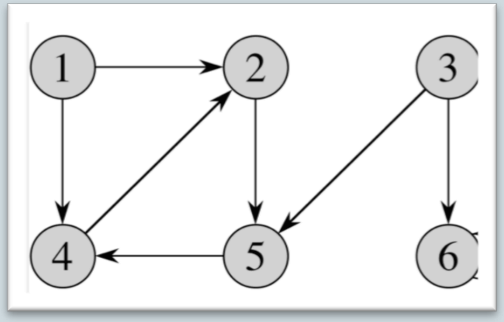


Ex $\deg(v_2)=4$

Directed graphs

(u,v): u is adjacent to v, v is adjacent from u

- $\deg^-(v)$: The in-degree of v, the number of edges entering it
- $\deg^+(v)$: The out-degree of v, the number of edges leaving it



$\deg^-(v_2)=2$

$\deg^+(v_2)=1$

Edges and Degrees: Storage of Adjacency List representation

- The *degree* of a vertex v = # incident edges
  - Directed graphs have **in-degree, out-degree**
- For directed graphs, # of items in adjacency lists is

$$\sum_{v \in V} \deg^{+}(v) = |E| \text{ (proof?)}$$

  takes $\Theta(V + E)$ storage
- For undirected graphs, # items in adj lists is

$$\sum_{v \in V} \deg(v) = 2\,|E| \quad (\textit{handshaking lemma})$$

  also $\Theta(V + E)$ storage

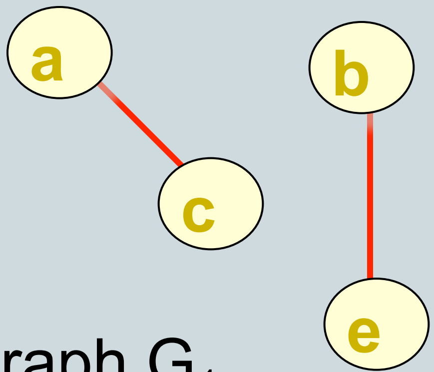
Incident: The edge that connects two vertices is incident on both of them.

Terminology

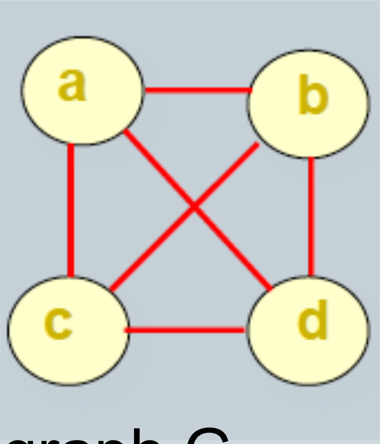
Def. A *forest* is a graph that is a collection of trees. Ex. G_1

Def. A *subgraph* of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G.

Def. A *complete* graph is an undirected graph with every pair of vertices adjacent K_2 (graph G_2)



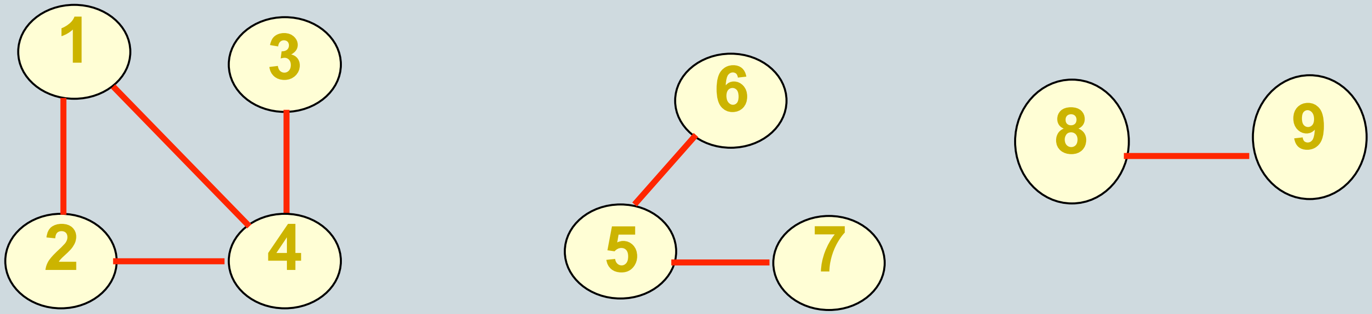
graph G_1



graph G_2

Connected Components

- The connected components of an undirected graph are the equivalence classes of vertices under the ``is reachable from'' relation.



A graph with three connected components: *{1,2,3,4}*, *{5,6,7}*, and *{8,9}*.

Input size of Graphs

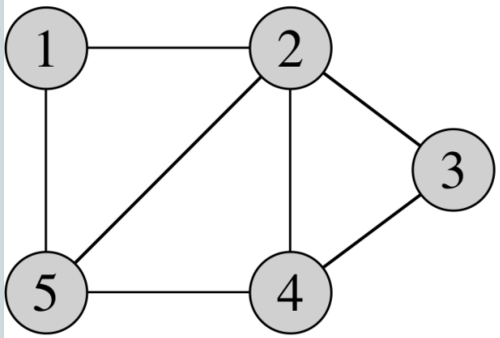
- We will typically express running times in terms of |E| and |V|
  - If $|E| \approx |V|^2$ the graph is *dense*
  - If $|E| \approx |V|$ the graph is *sparse*

- If you know you have dense or sparse graphs, different data structures (representations) may make sense

- 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 ($V - E + F = 2$)*
  - For this reason the ***adjacency list*** is often a more appropriate representation

- *In the case of the cube, $V = 8$, $E = 12$ and $F = 6$. So, $V - E + F = 8 - 12 + 6 = 14 - 12 = 2$

Review: representation of graphs
Adjacency Lists and Matrix



(a)

(b) (c)

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

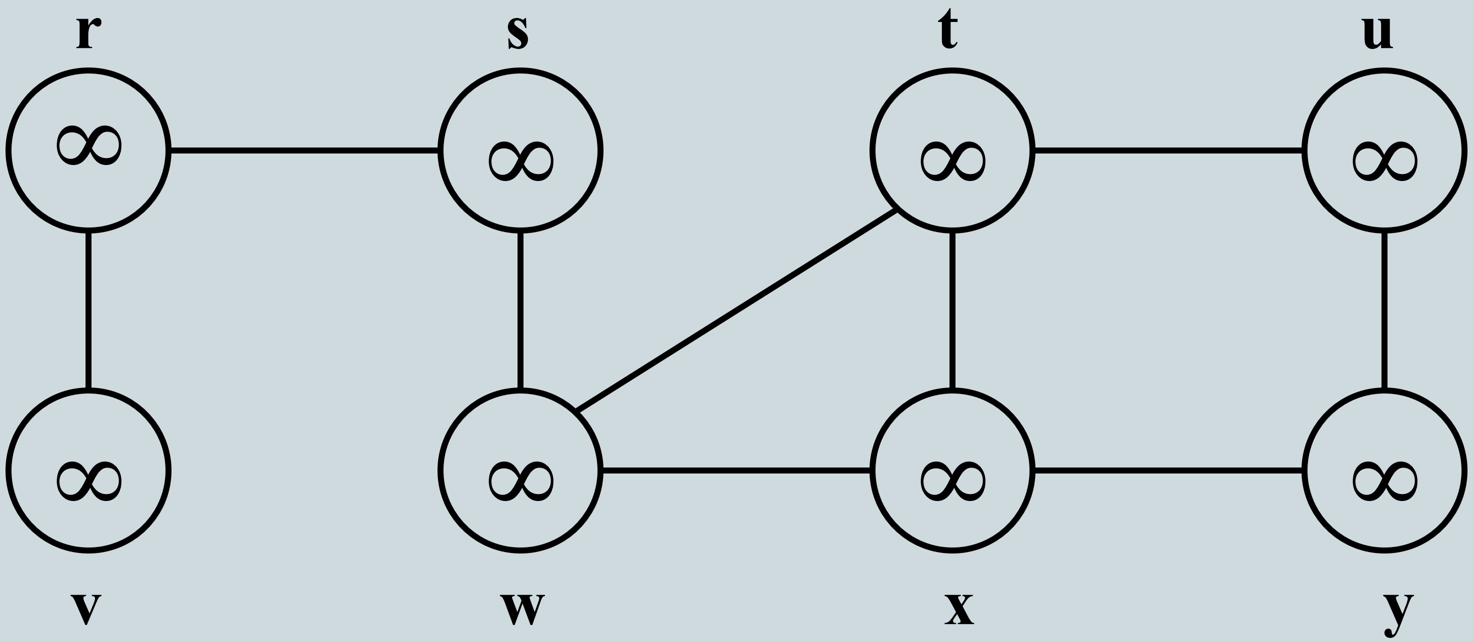
- Again will associate vertex “colors” to guide the algorithm
  - *White* vertices have not been discovered
    - All vertices start out white
  - *Grey* vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - *Black* vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- KEY IDEA: Explore vertices by scanning adjacency list of grey vertices

Breadth-First Search

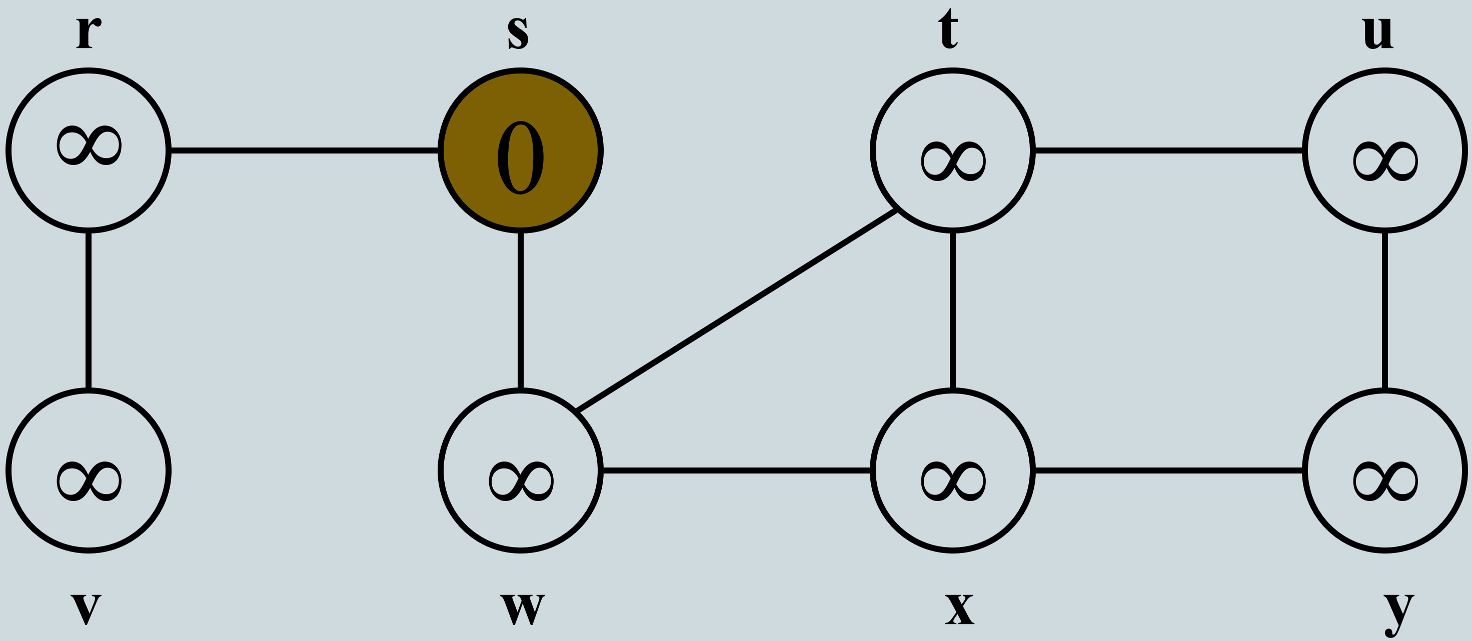

```
BFS(G, s) {
   initialize vertices; //each v will be v.color=WHITE
                                            //v.d= ∞ and
  v.π=NIL
    ENQUEUE(Q,s);          // Q is a queue; initialize to s
    while (Q not empty) {
        u = DEQUEUE(Q);
        for each v ∈ G.adj[u] {
            if (v.color == WHITE)
                v.color = GREY;
                v.d = u.d + 1;
                v.π = u;
                Enqueue(Q, v);
        }
        u.color = BLACK;
    }
}
```

What does `v.d` represent?
What does `v.π` represent?

Breadth-First Search: Example

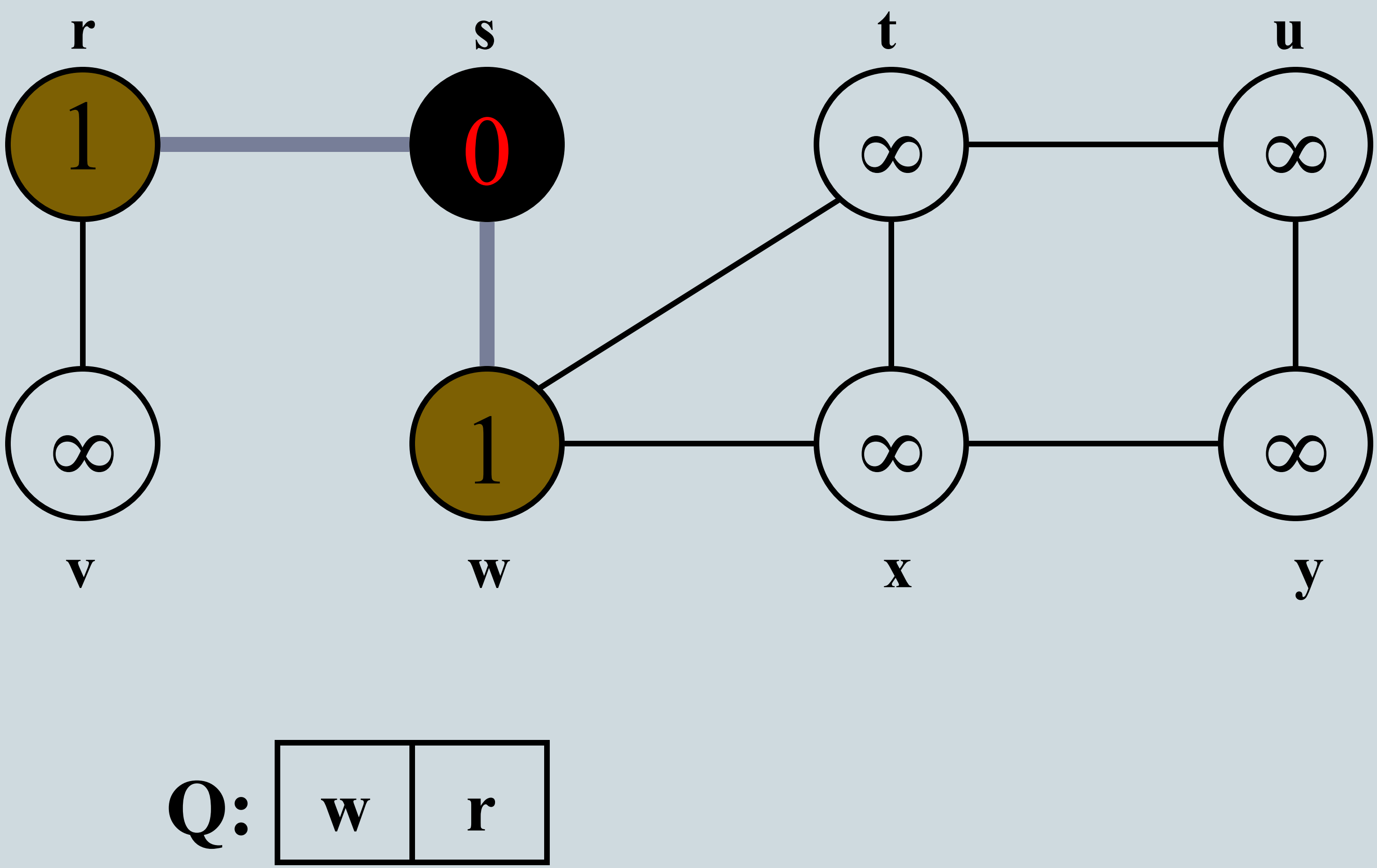


Breadth-First Search: Example

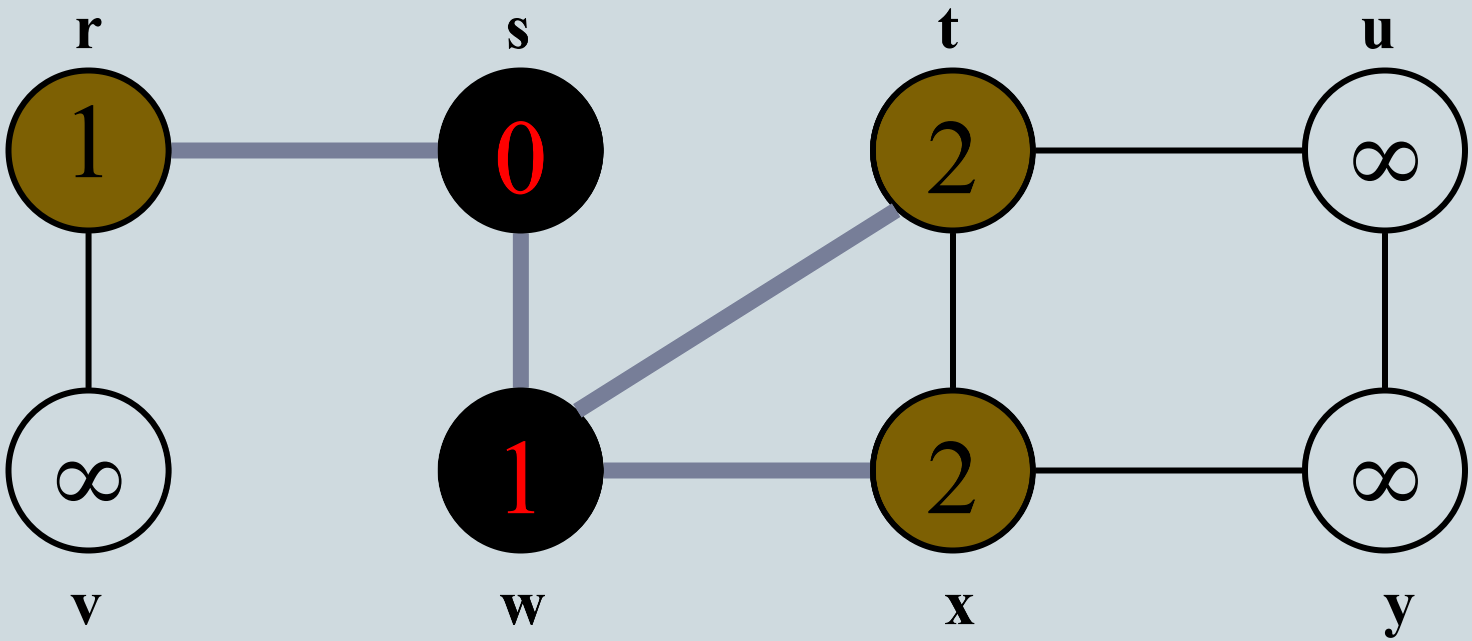


Q: s

Breadth-First Search: Example



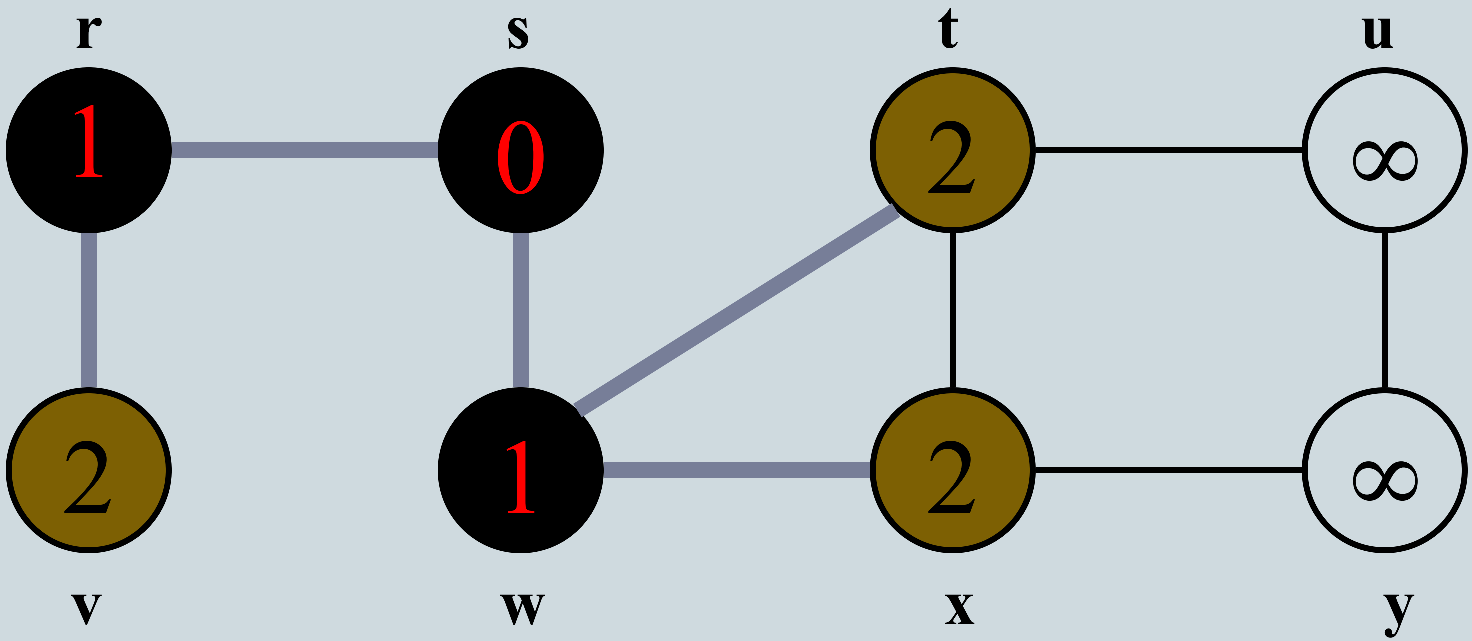
Breadth-First Search: Example



Q:

r	t	x

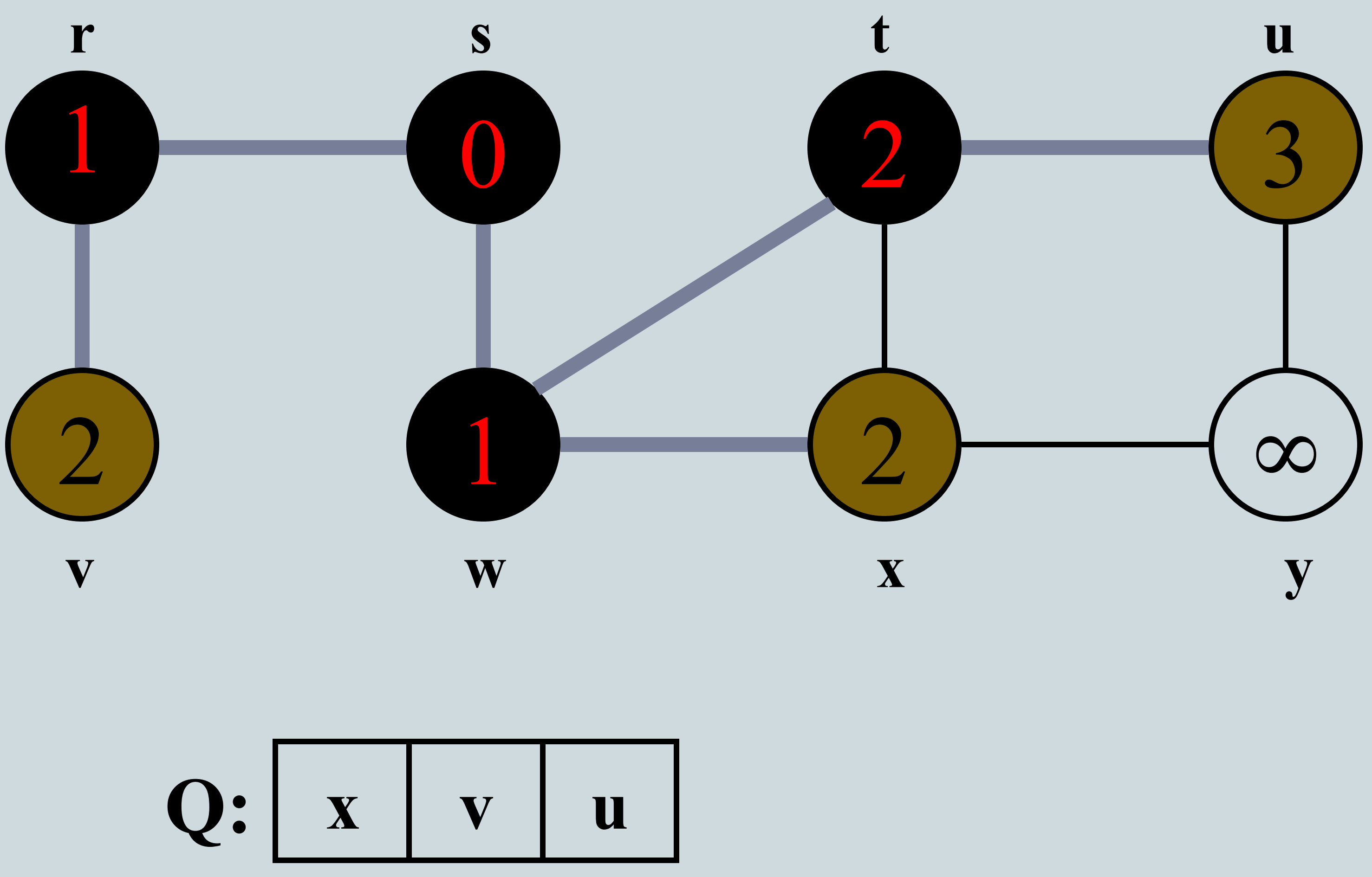
Breadth-First Search: Example



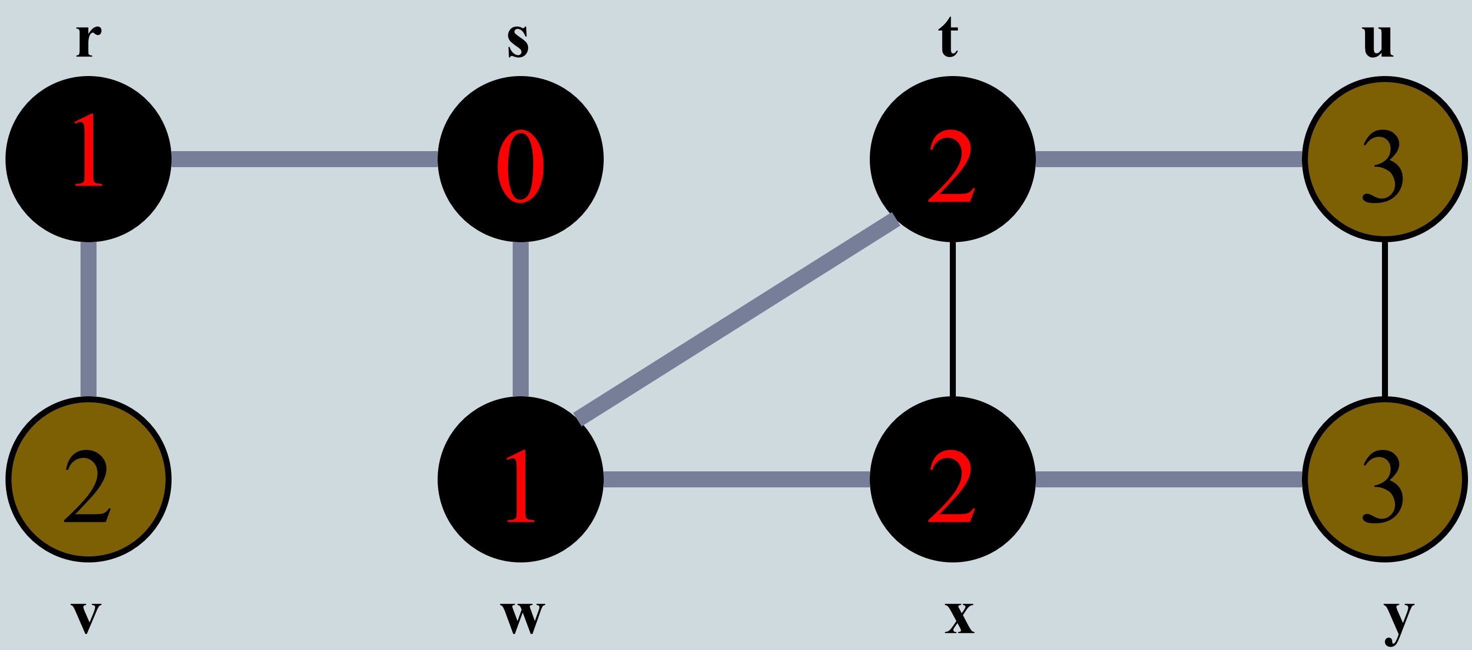
Q:

t	x	v

Breadth-First Search: Example



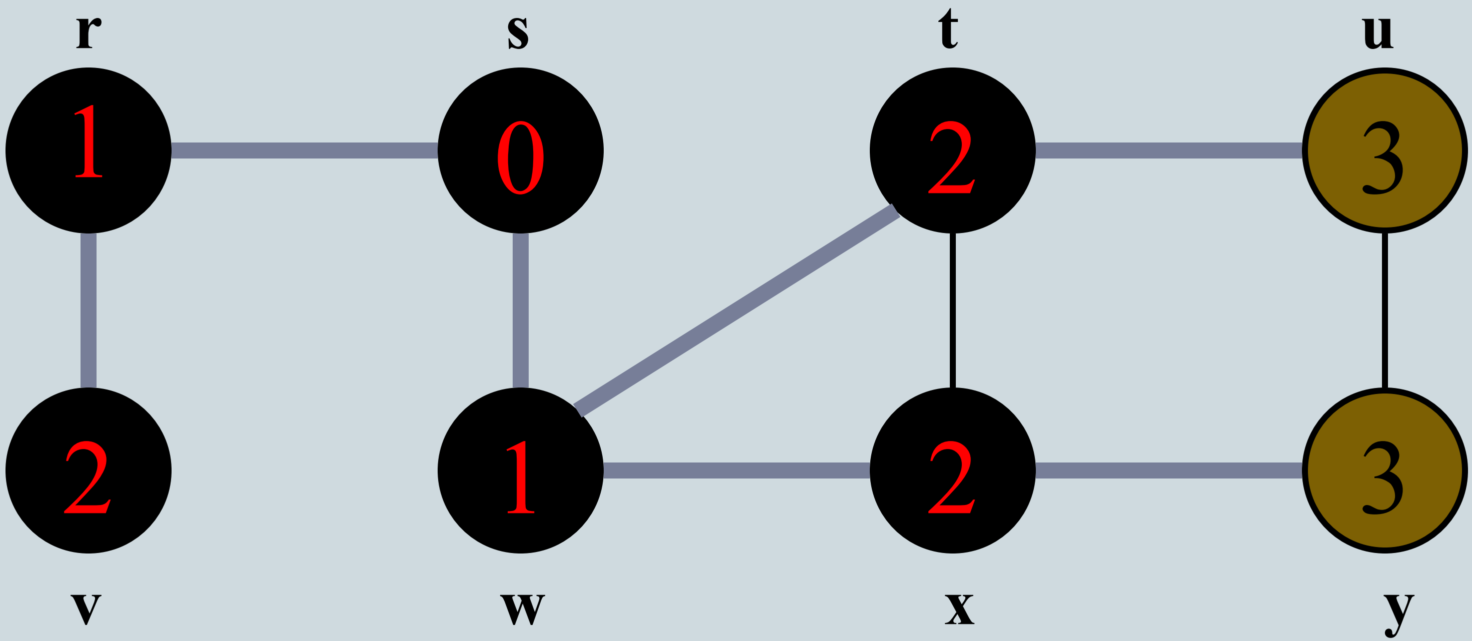
Breadth-First Search: Example



Q:

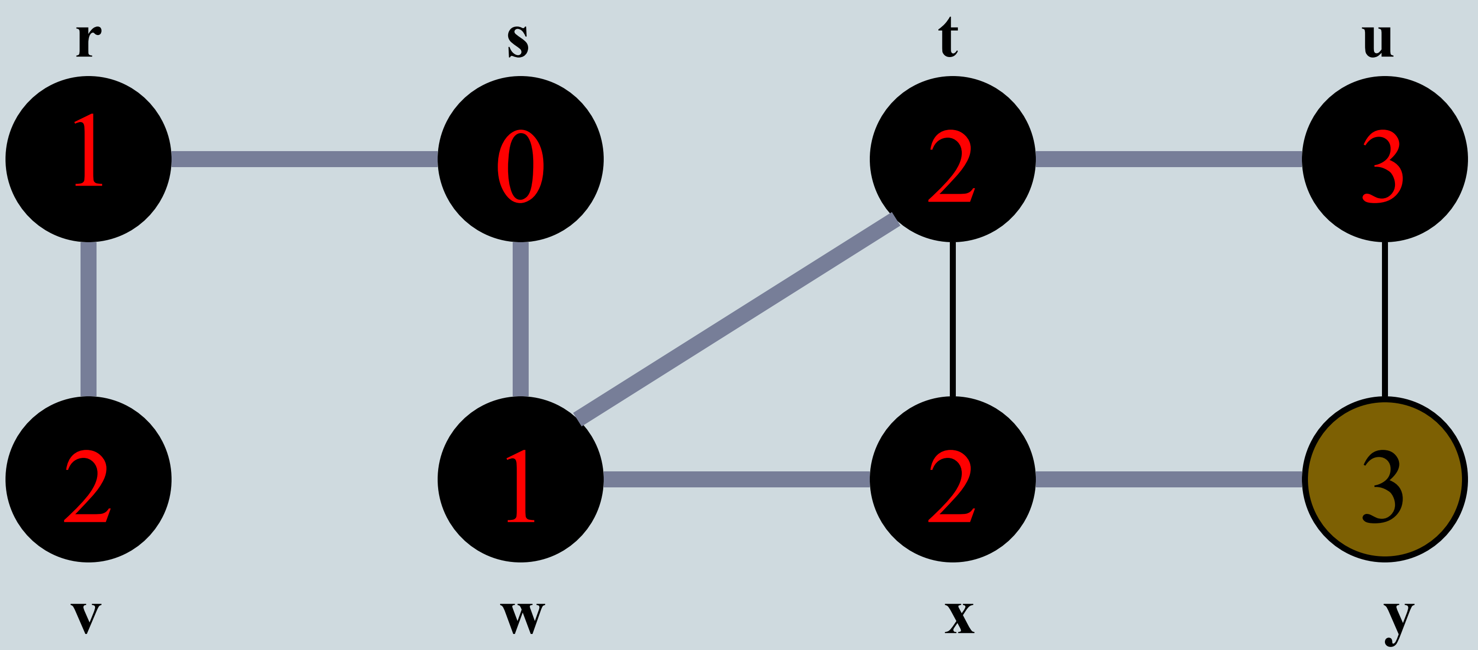
v	u	y

Breadth-First Search: Example



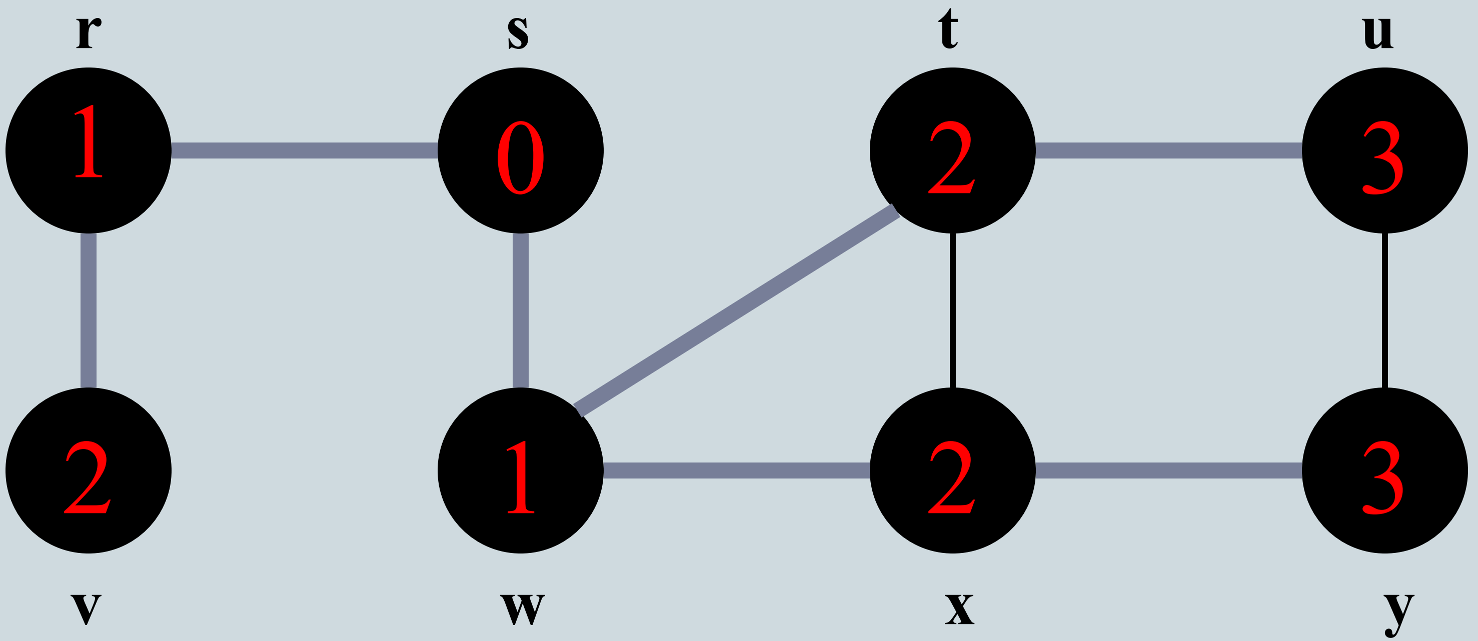
Q: | u | y |

Breadth-First Search: Example



Q: y

Breadth-First Search: Example



Q: Ø

BFS: The Algorithm


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

Touch every vertex: O(V)

u = every vertex, but only once

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

O(V+E) running time

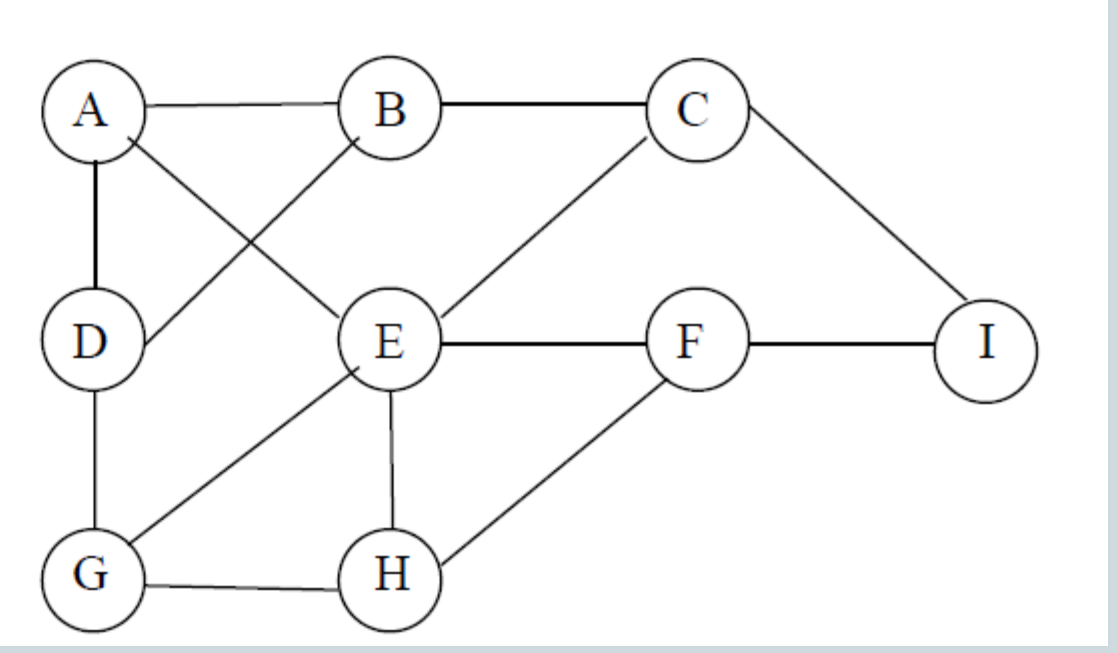
Total space used:

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

Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* to the source node
  - Shortest-path distance d(s,v) = minimum number of edges from s to v, or ∞ if v not reachable from s
- 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

example

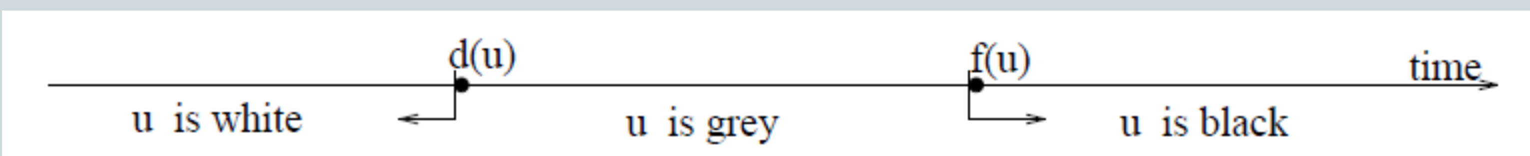


Depth-First Search

- *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

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



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);
    }
}
```

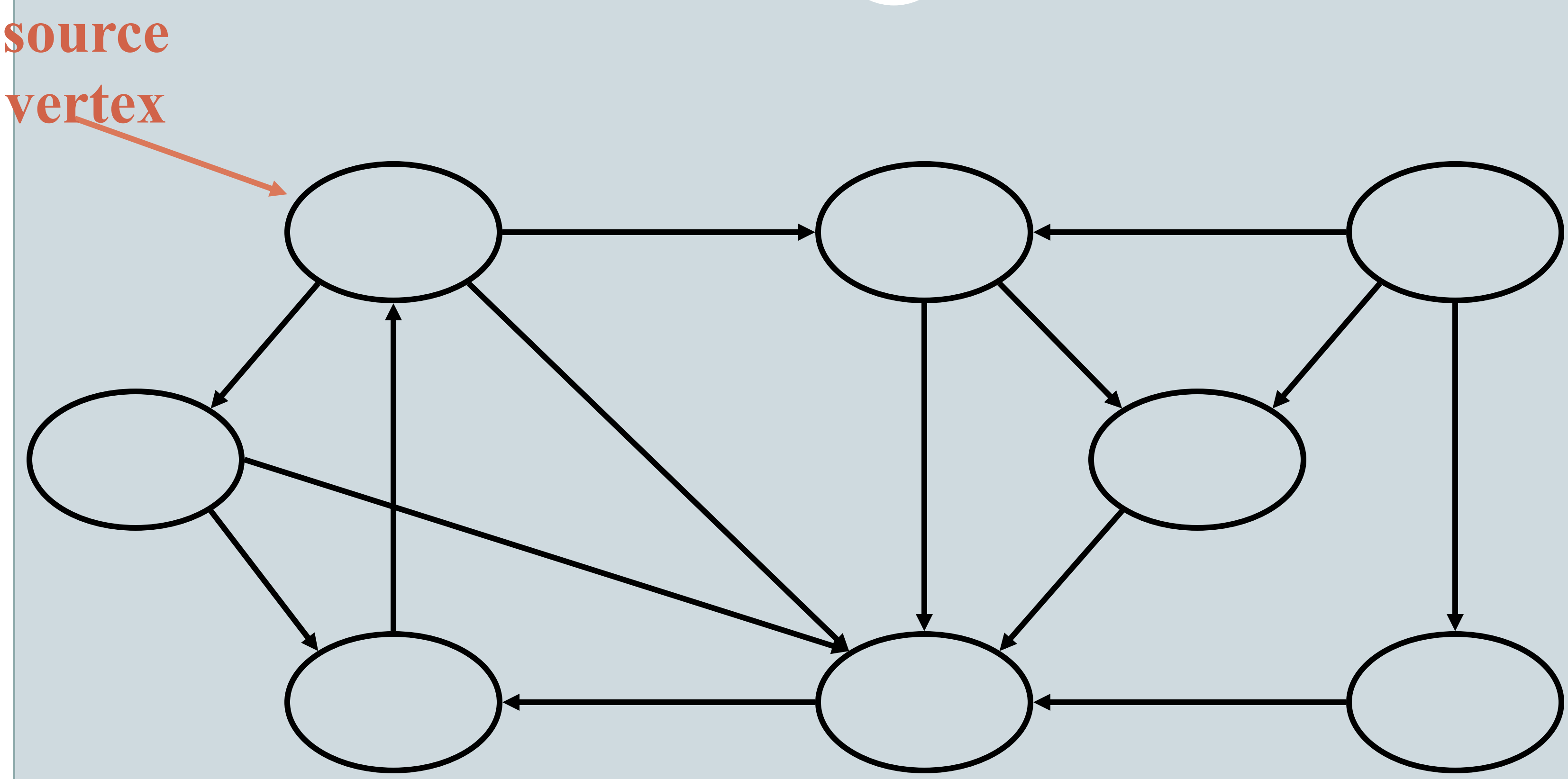
```
DFS_Visit(u)
{
    u.color = GREY;
    time = time+1;
    u.d = time;
    for each v ∈ G.Adj[u]
    {
        if (v.color == WHITE)
            DFS_Visit(v);
    }
    u.color = BLACK;
    time = time+1;
    u.f = time;
}
```

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

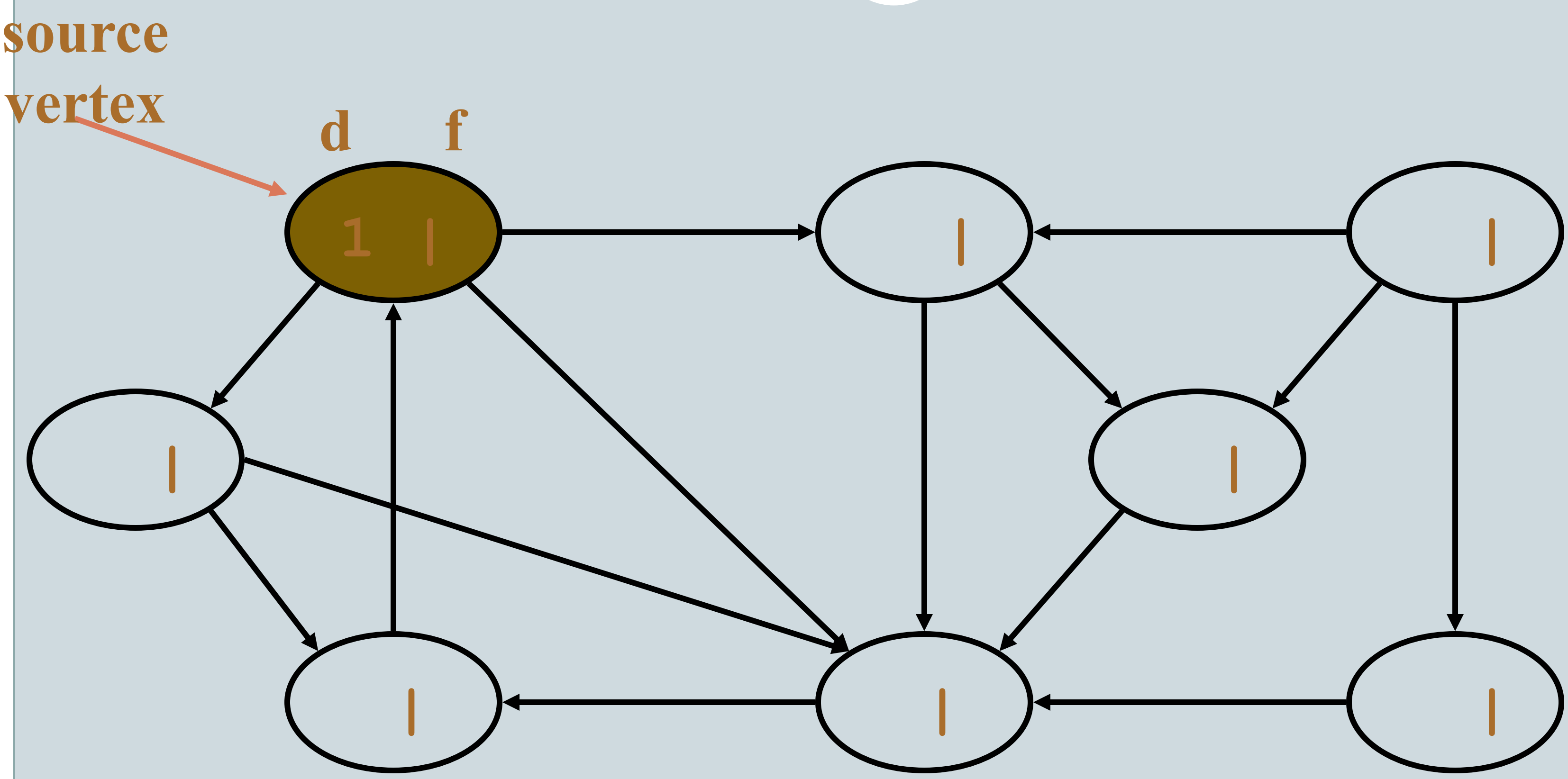
Depth-First Sort Analysis

- This running time argument is an informal example of *aggregate 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

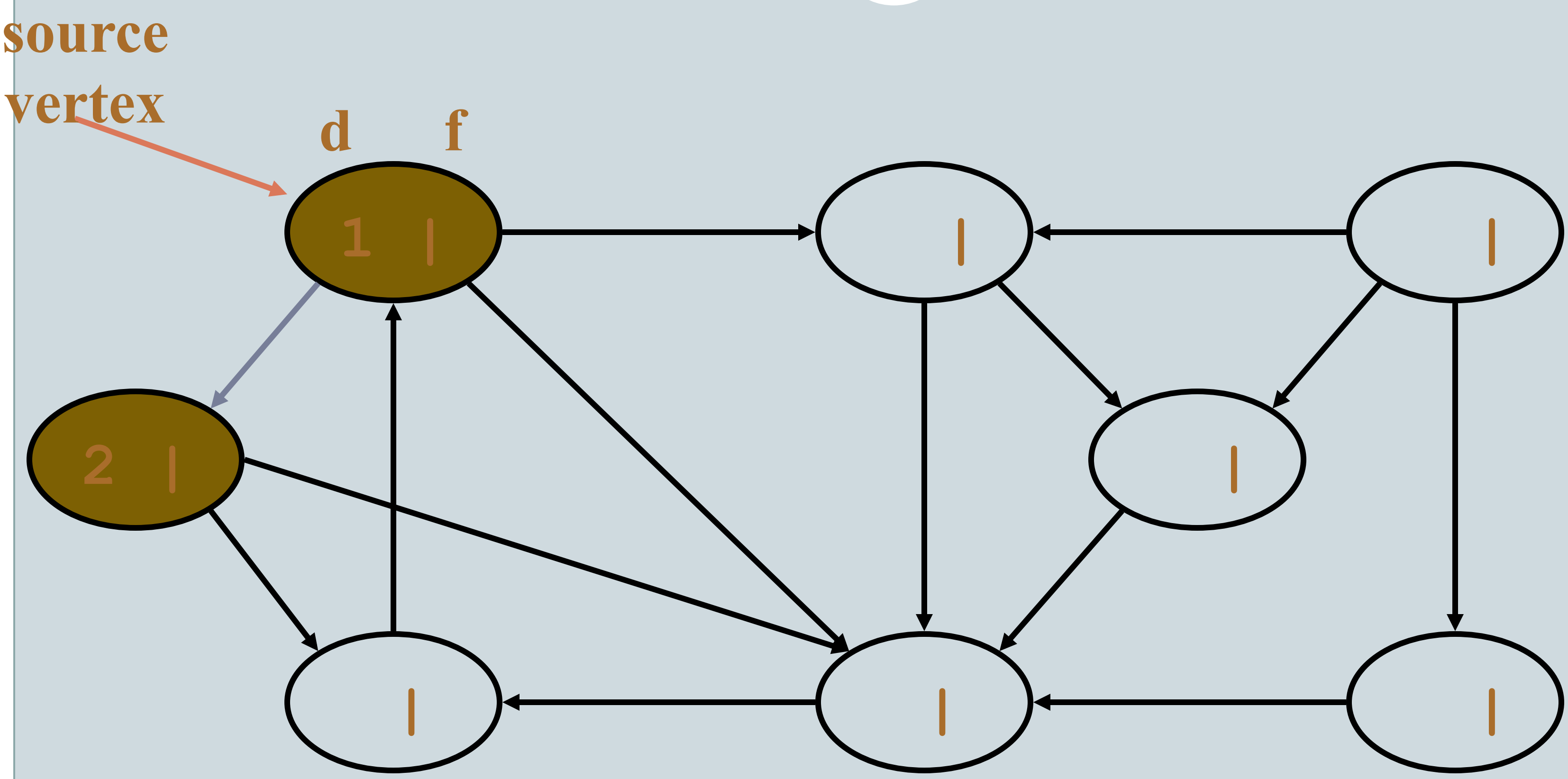
DFS Example



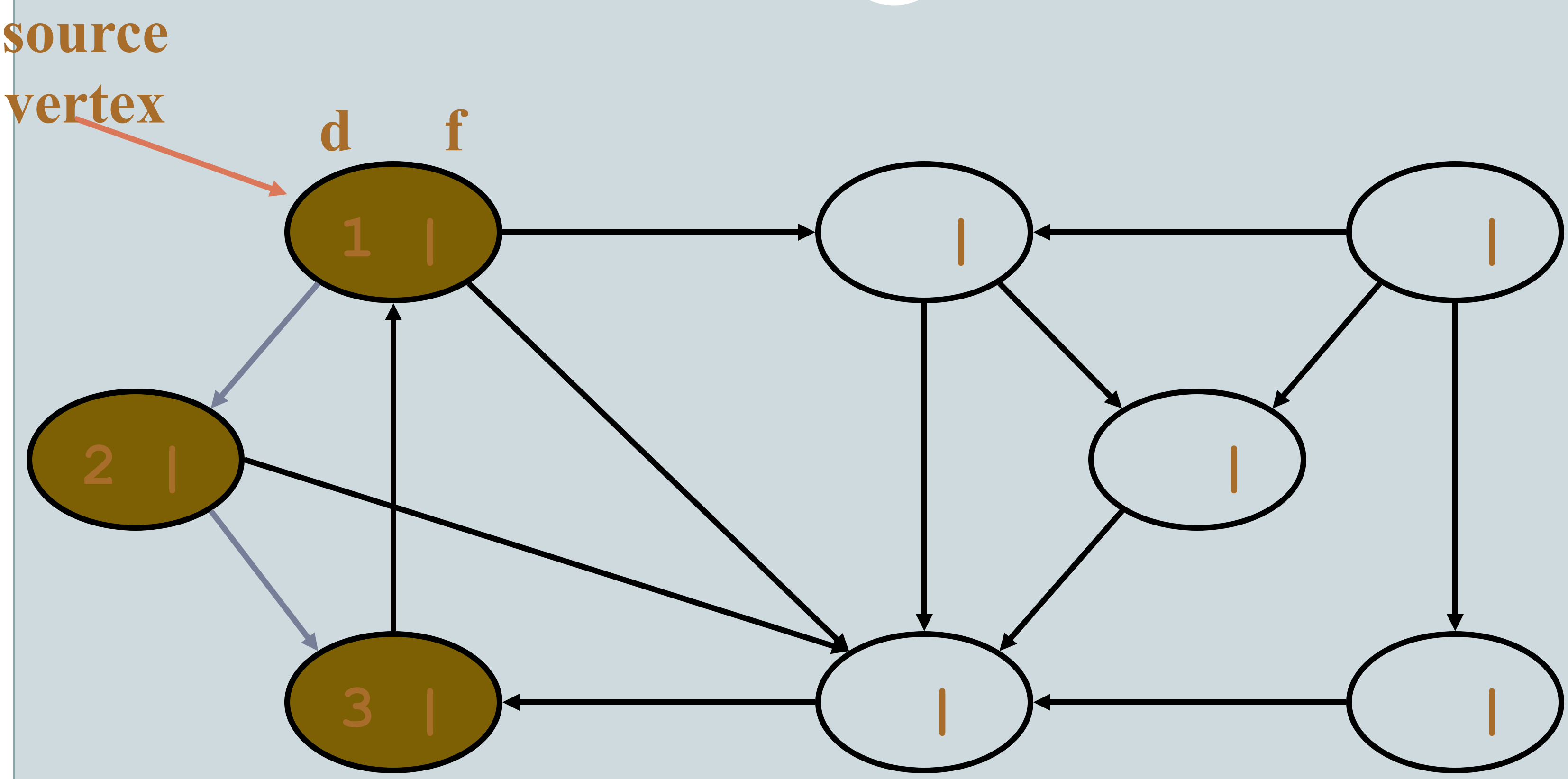
DFS Example



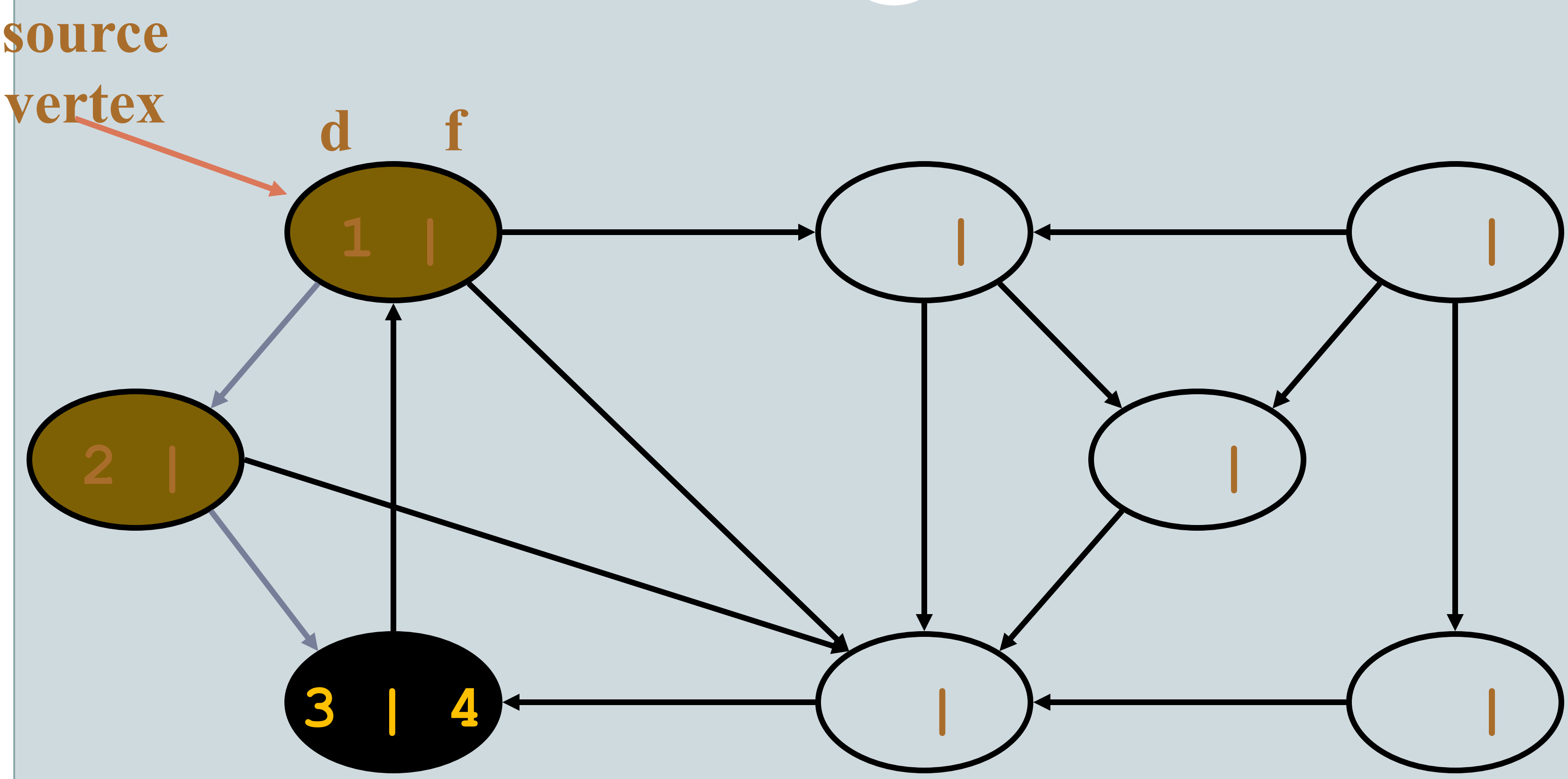
DFS Example



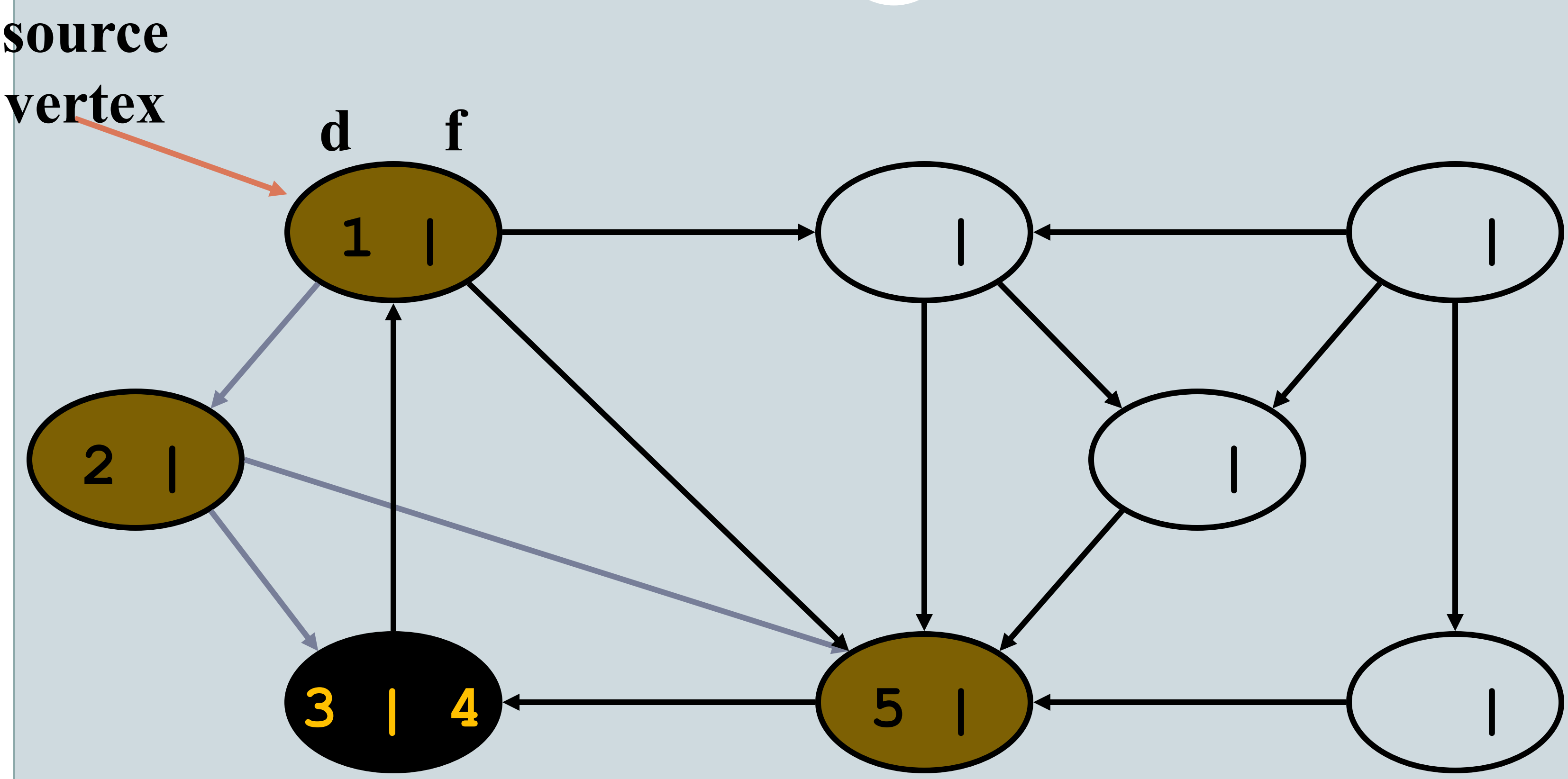
DFS Example



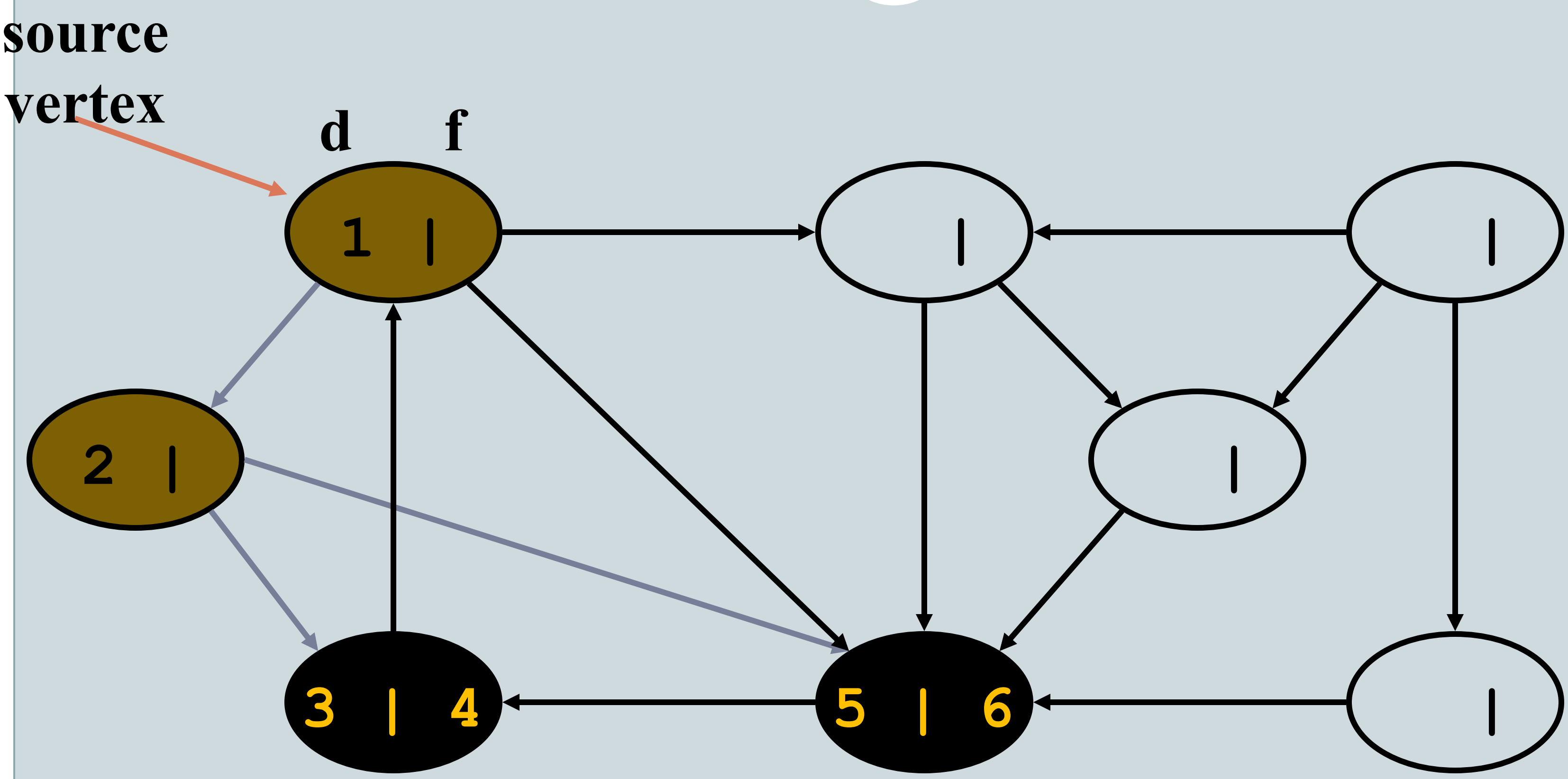
DFS Example



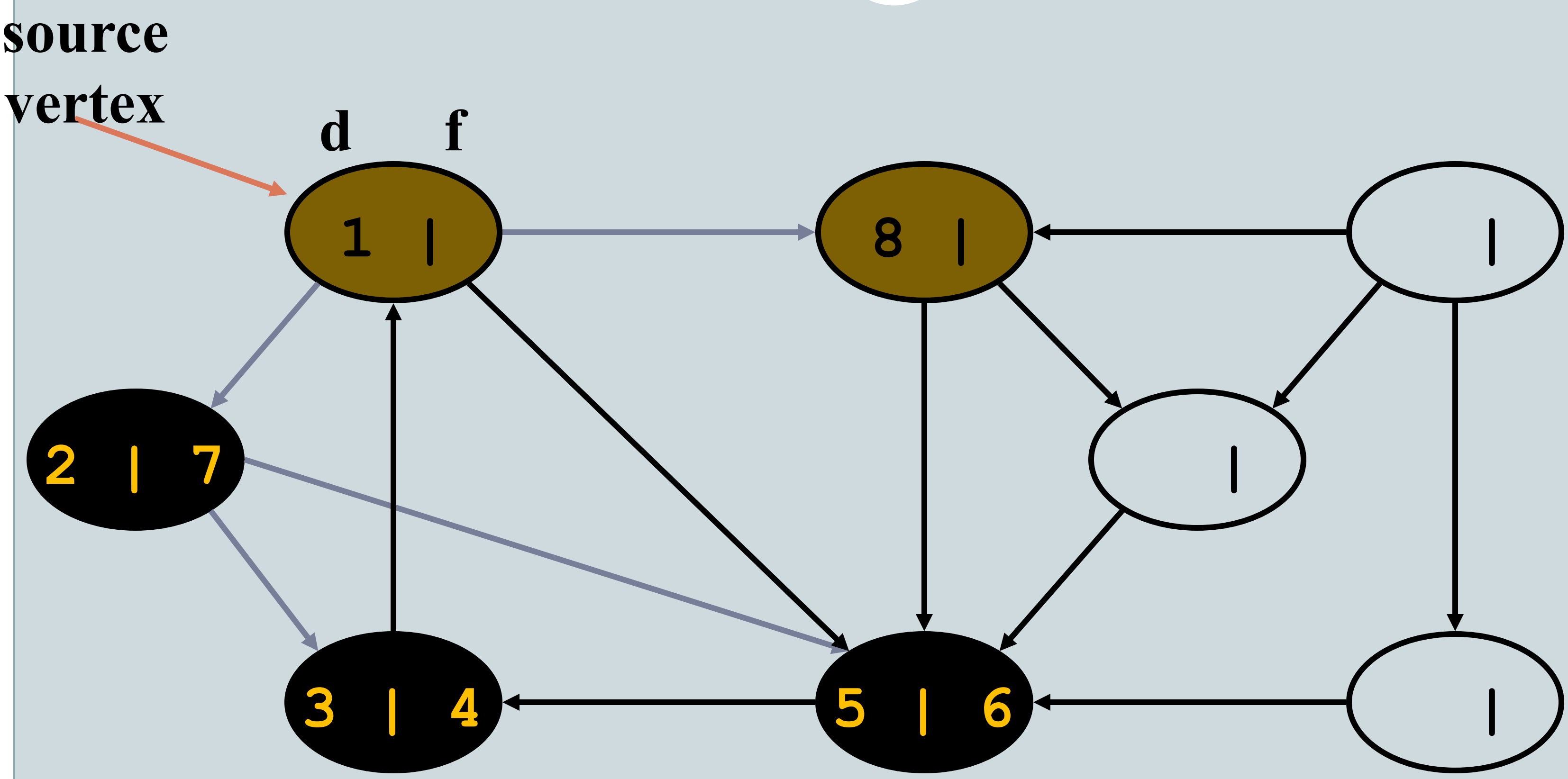
DFS Example



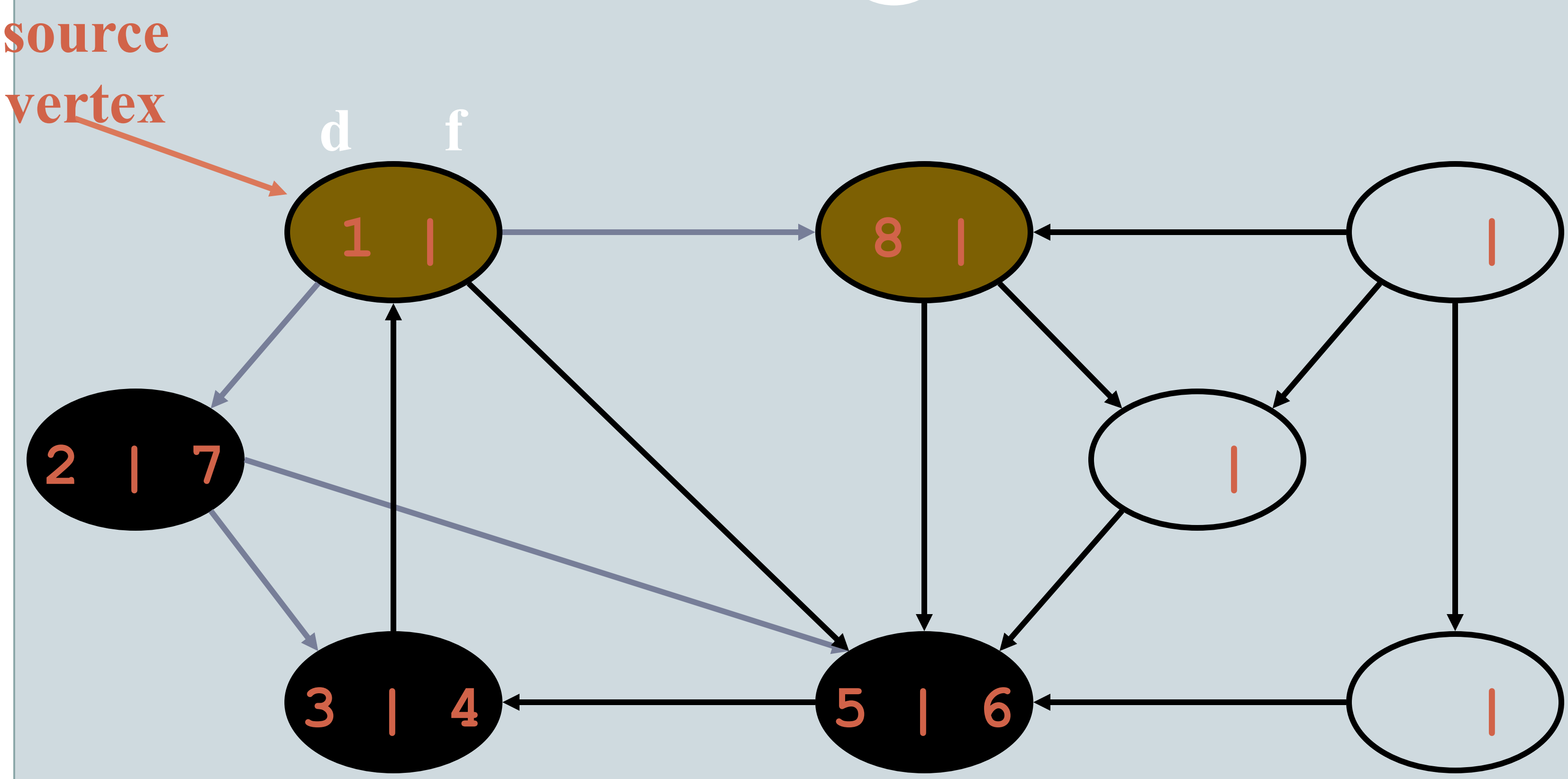
DFS Example



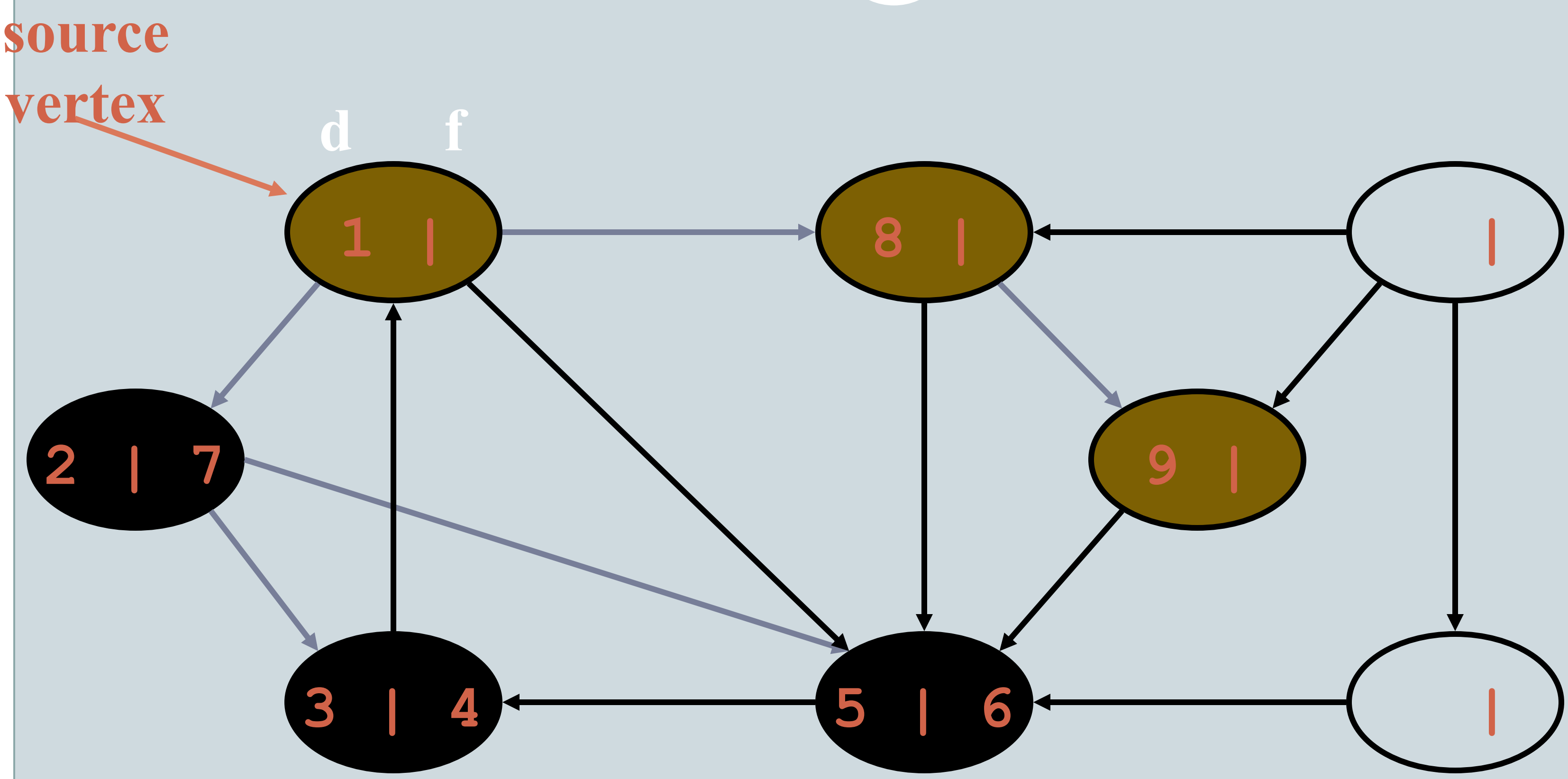
DFS Example



DFS Example

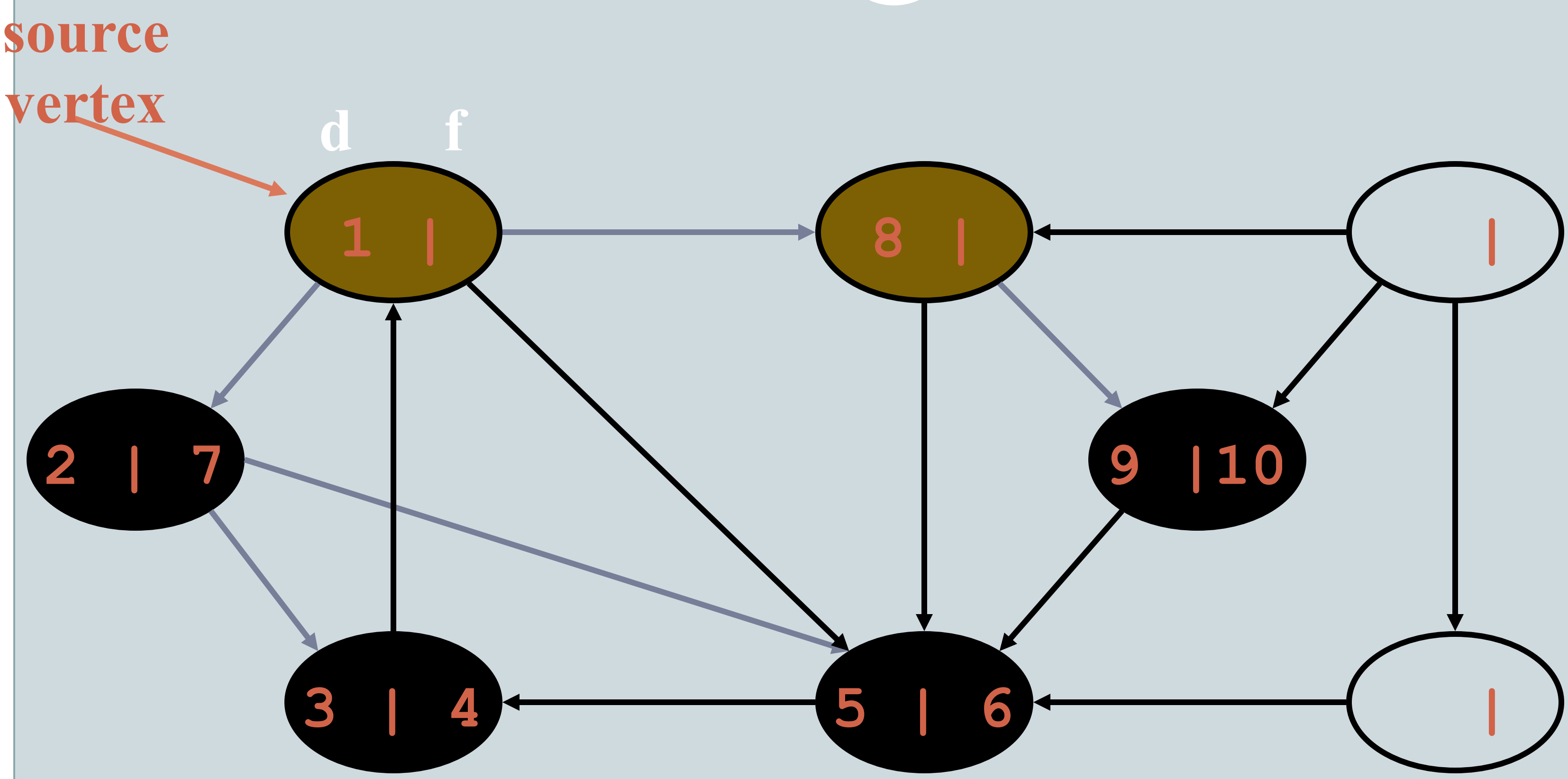


DFS Example

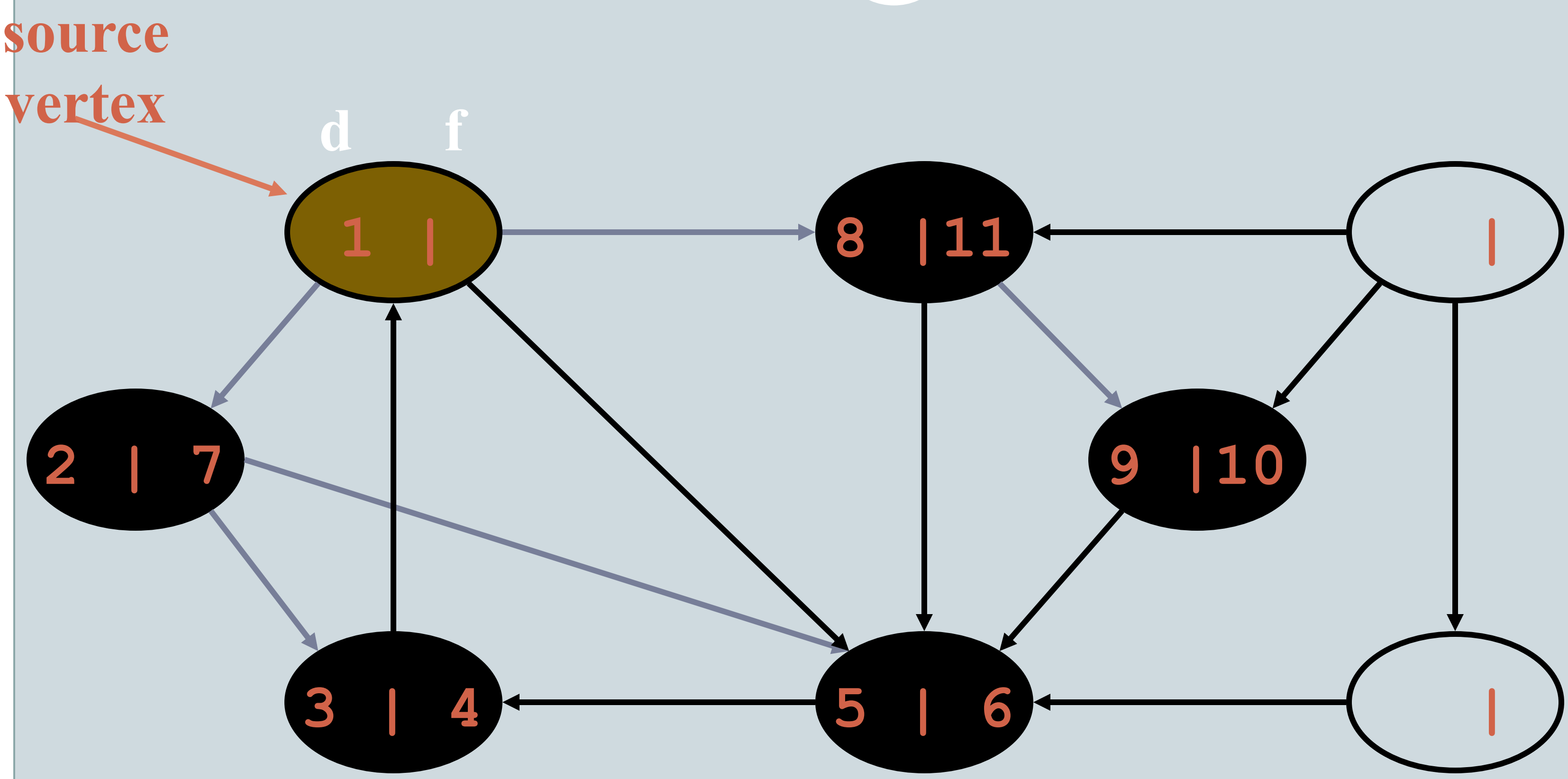


What is the structure of the grey vertices? What do they represent?

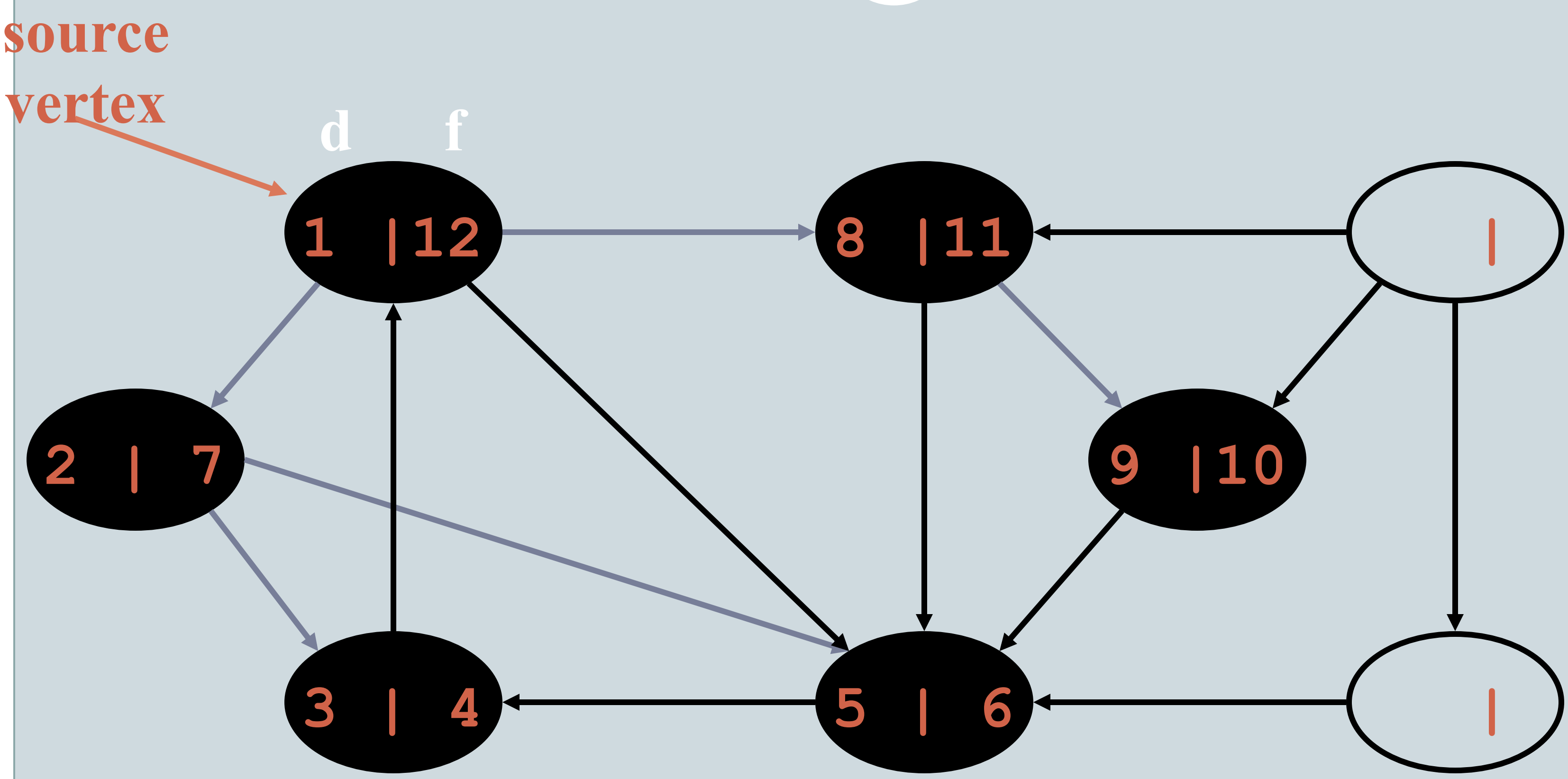
DFS Example



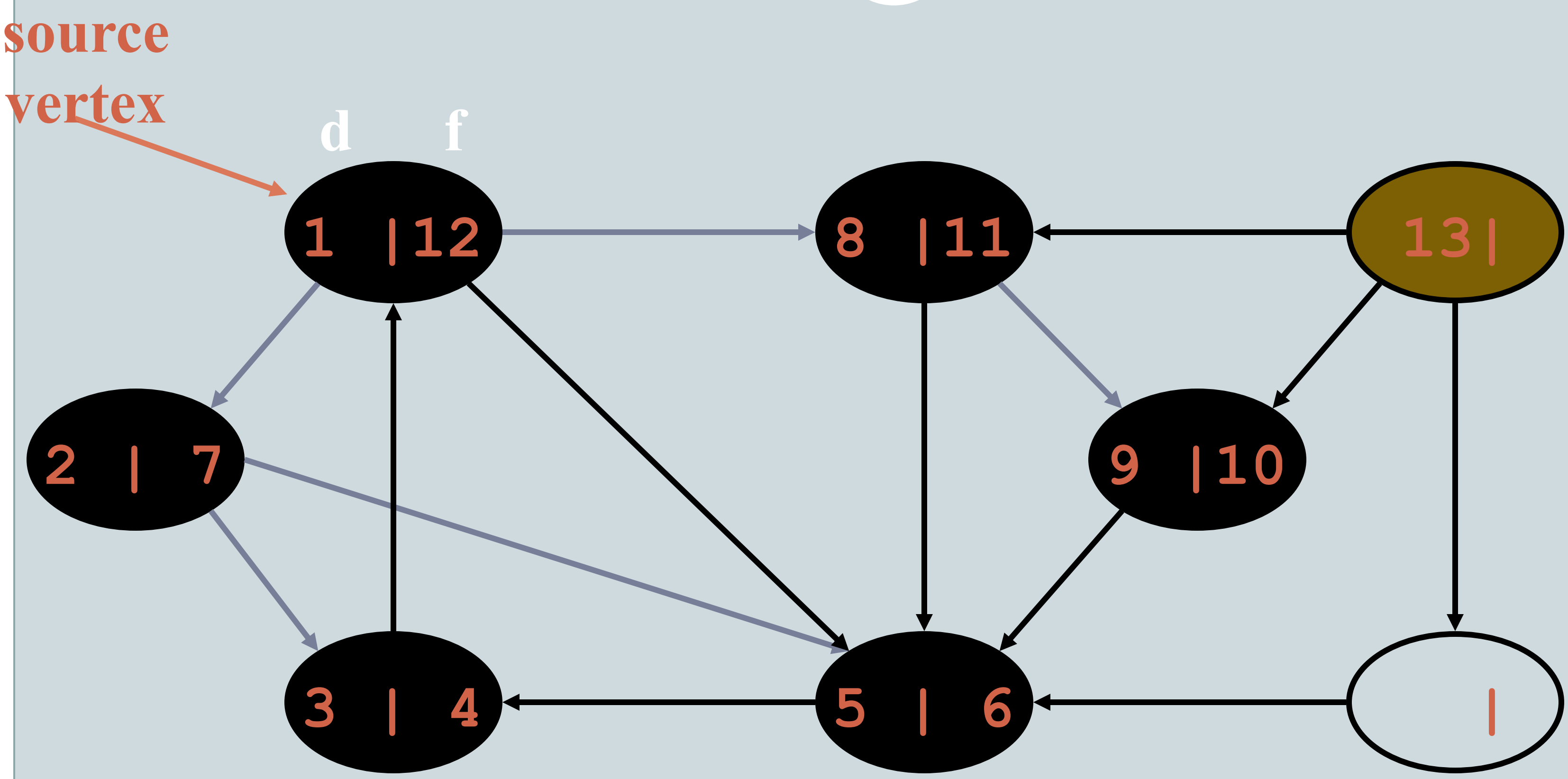
DFS Example



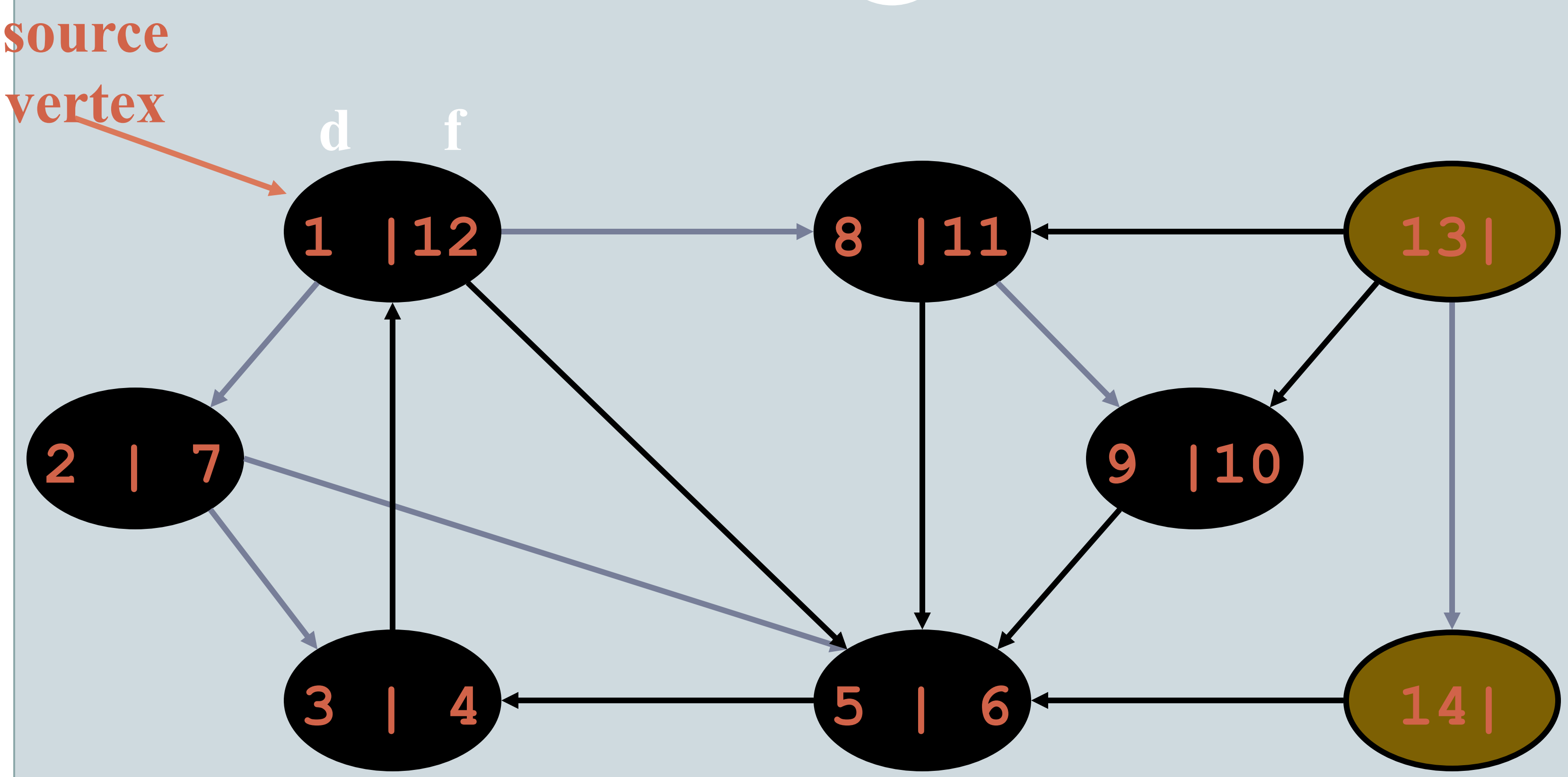
DFS Example



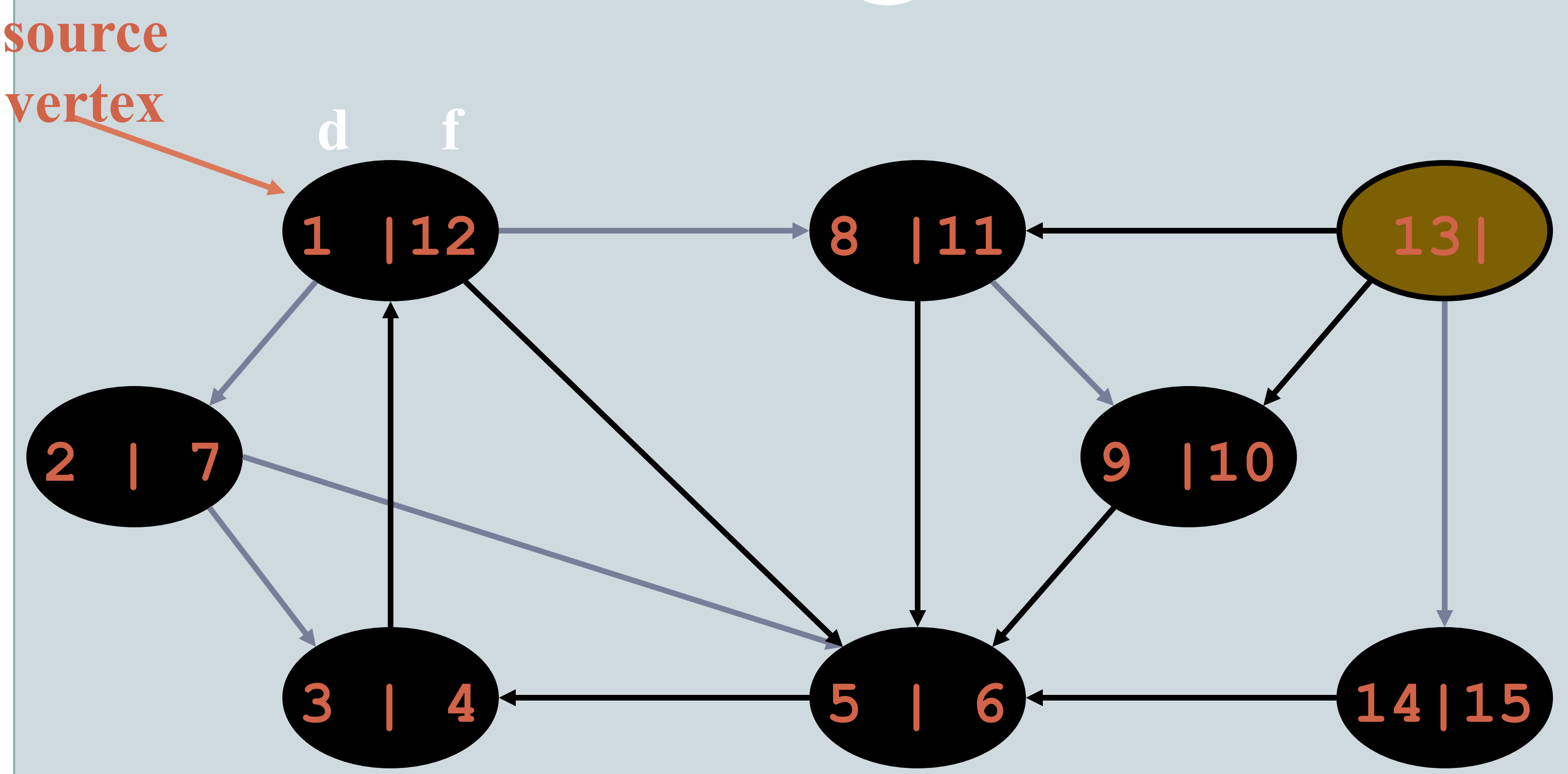
DFS Example



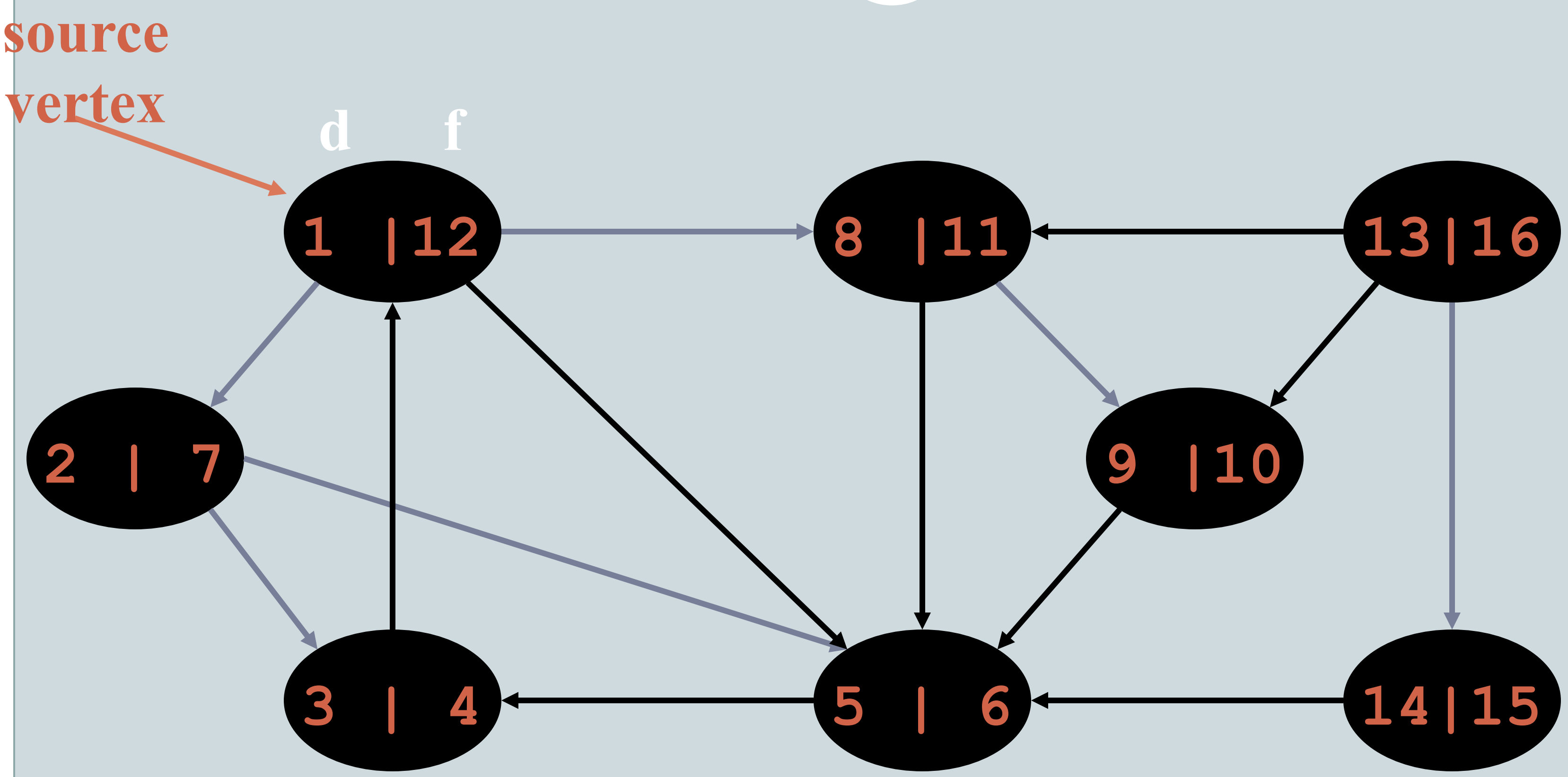
DFS Example



DFS Example



DFS Example

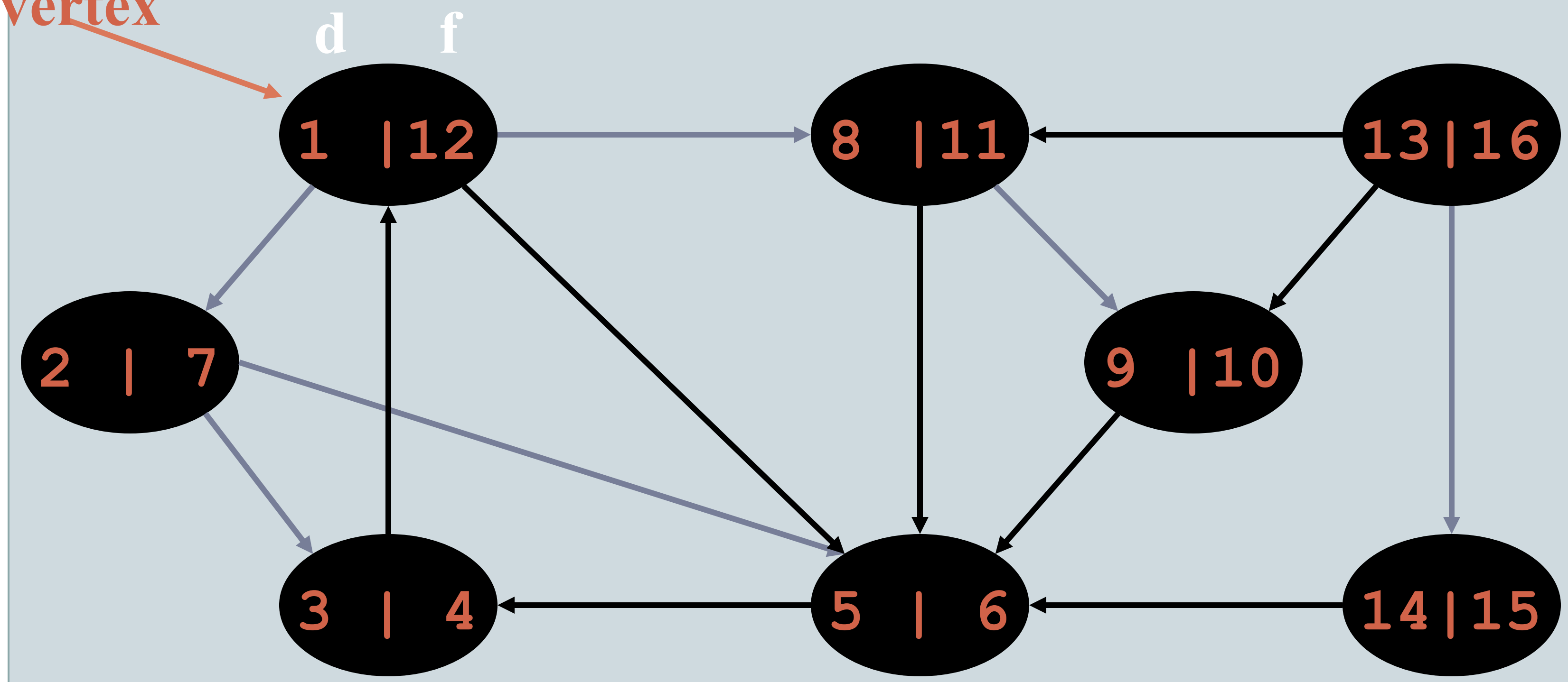


DFS: Kinds of edges

- 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

source vertex



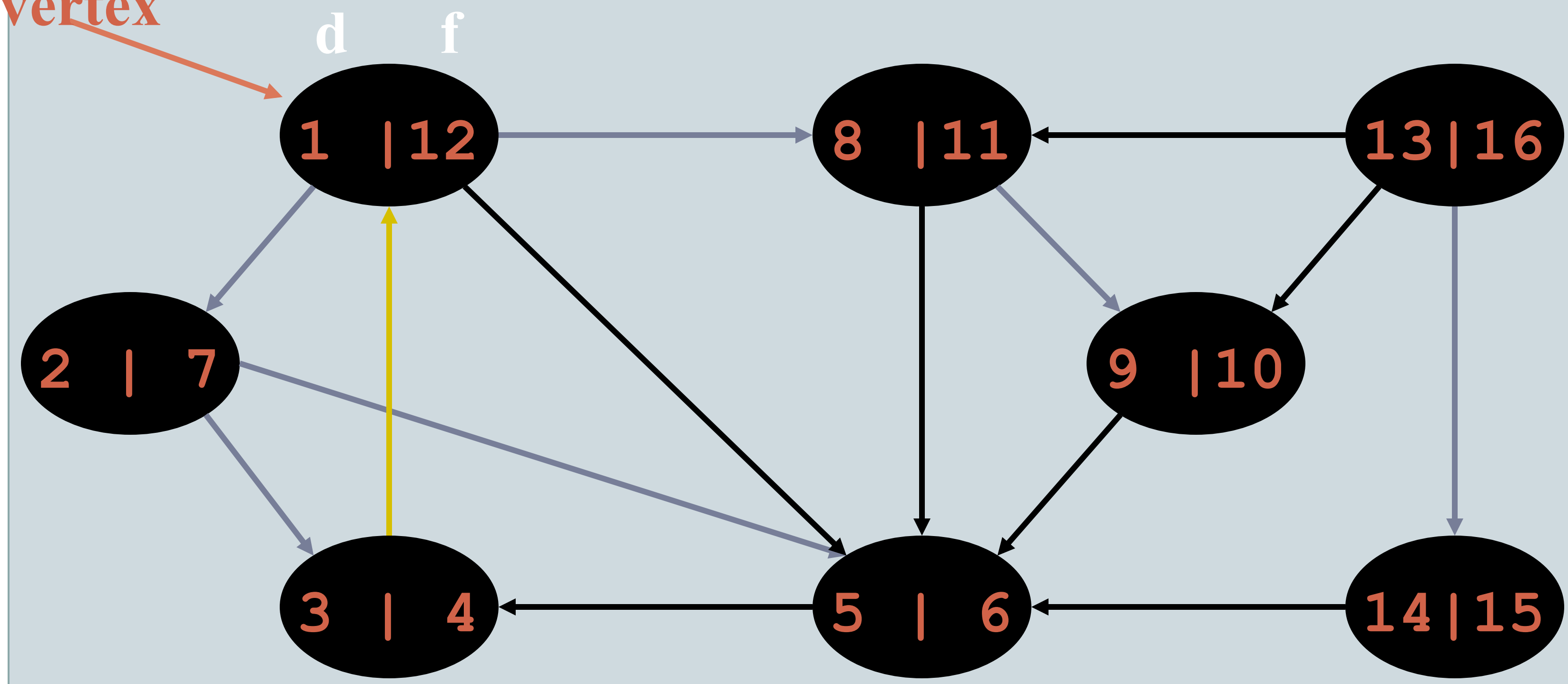
Tree edges

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
    - Encounter a grey vertex (grey to grey)

DFS Example

source vertex

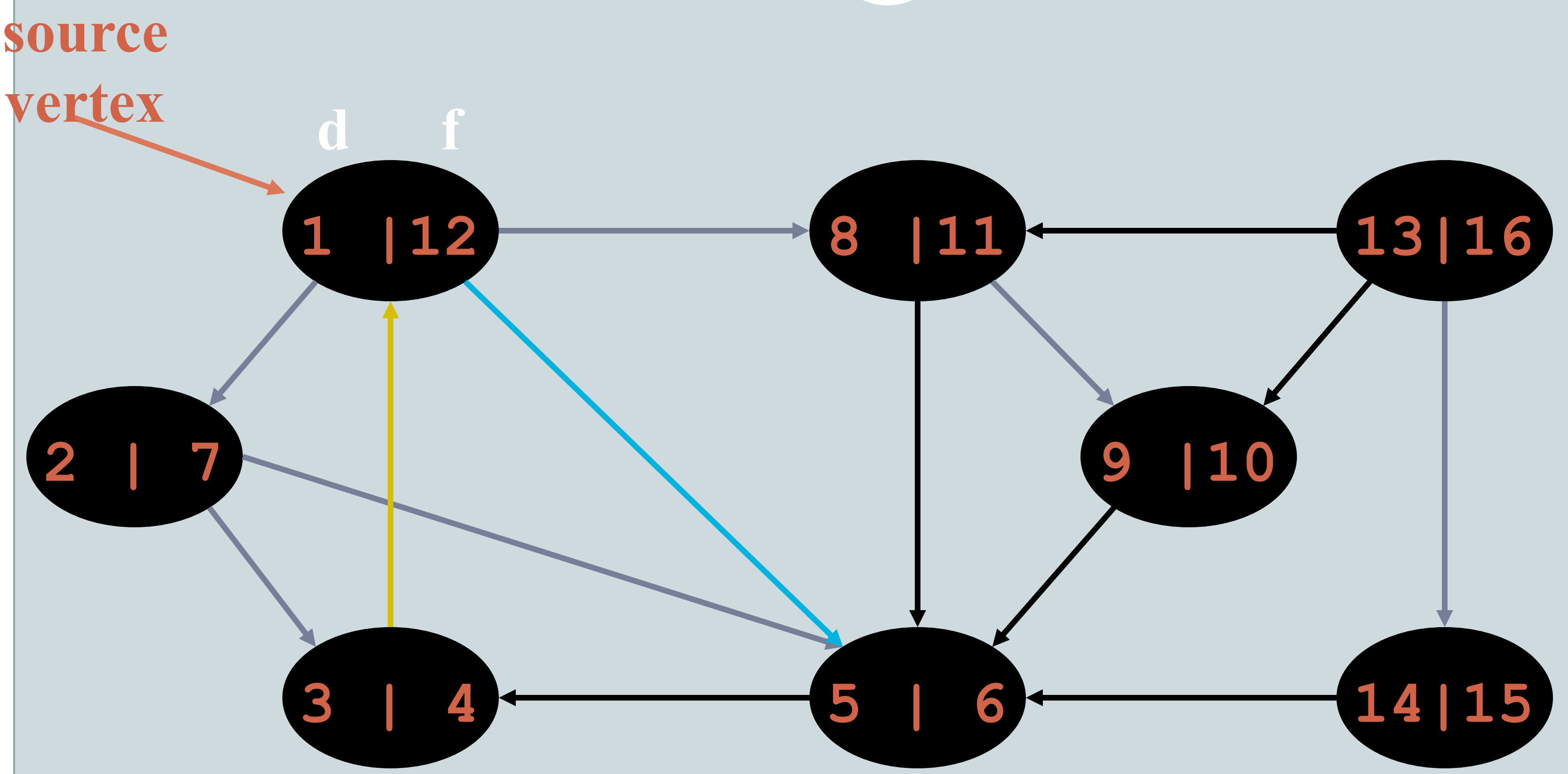


Tree edges Back edges

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
    - Not a tree edge, though
    - From grey node to black node

DFS Example

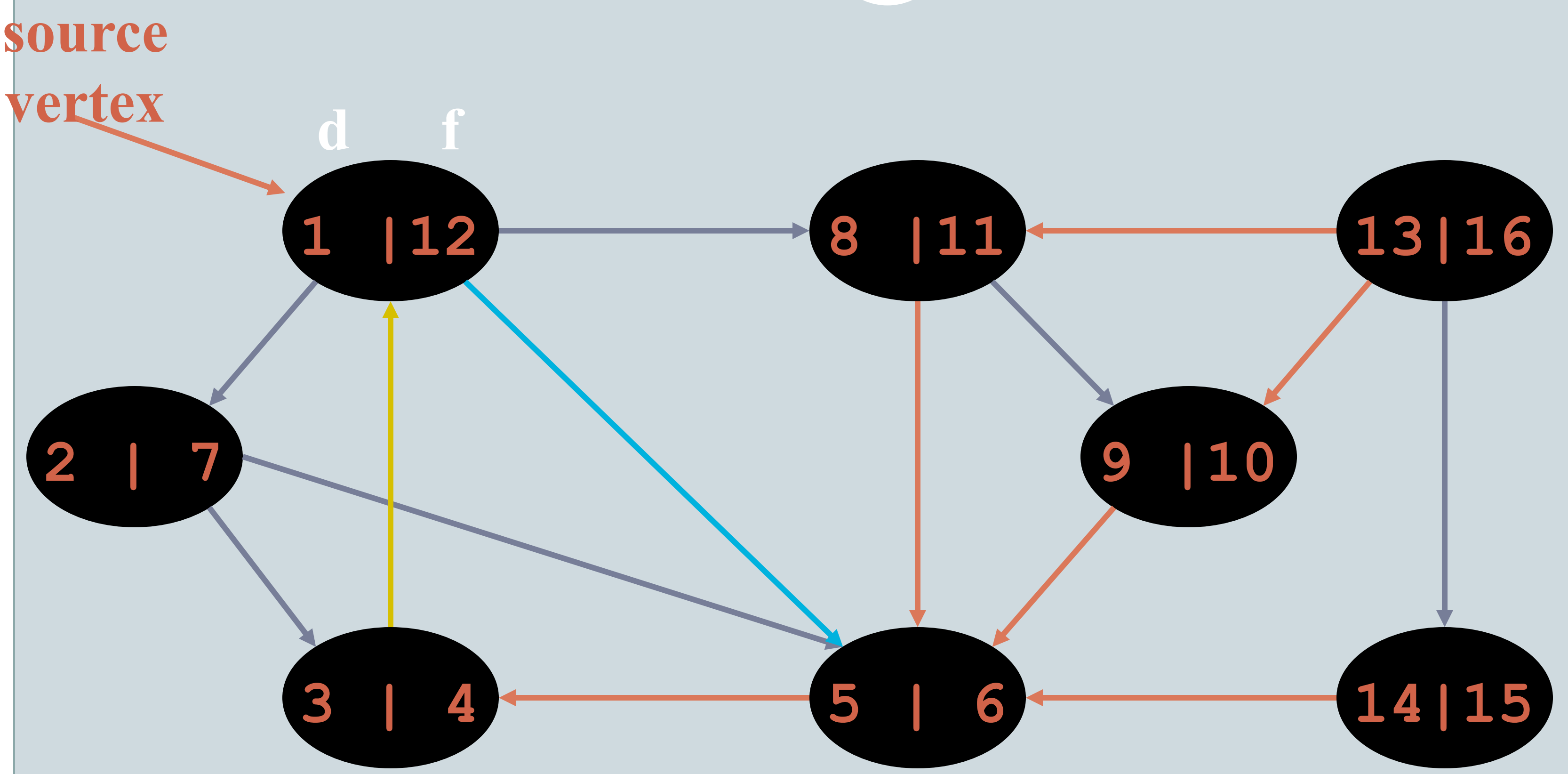


Tree edges Back edges Forward edges

DFS: Kinds of edges

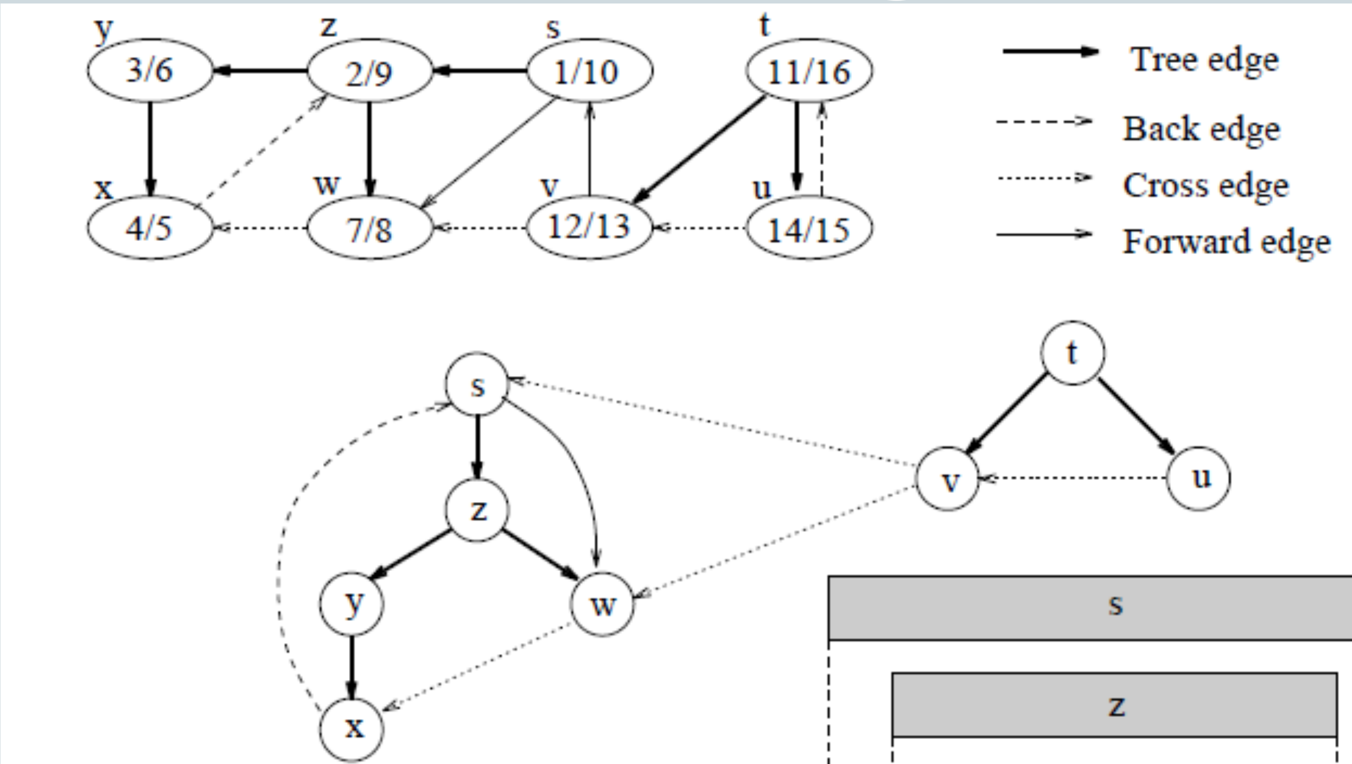
- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
  - *Cross edge*: between a tree or subtrees
    - From a grey node to a black node

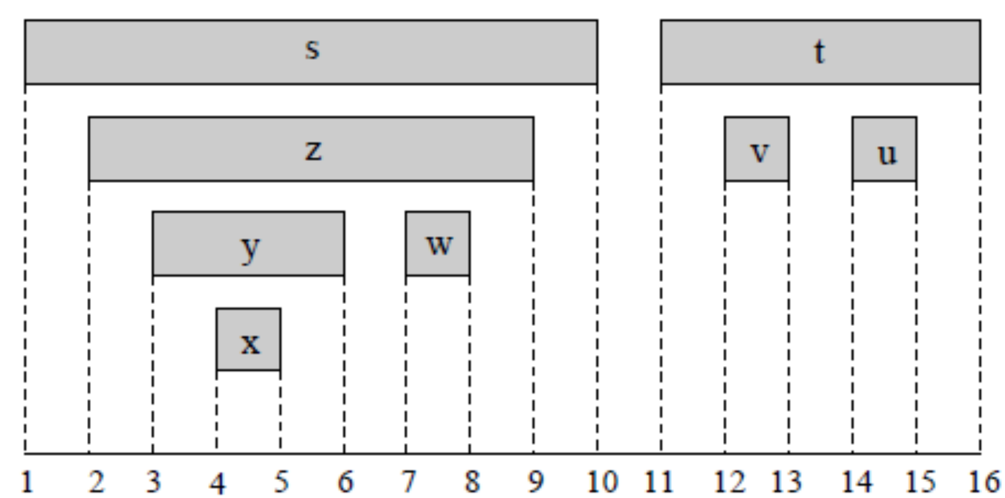
DFS Example



Tree edges   Back edges   Forward edges   Cross edges

Parenthesis structure



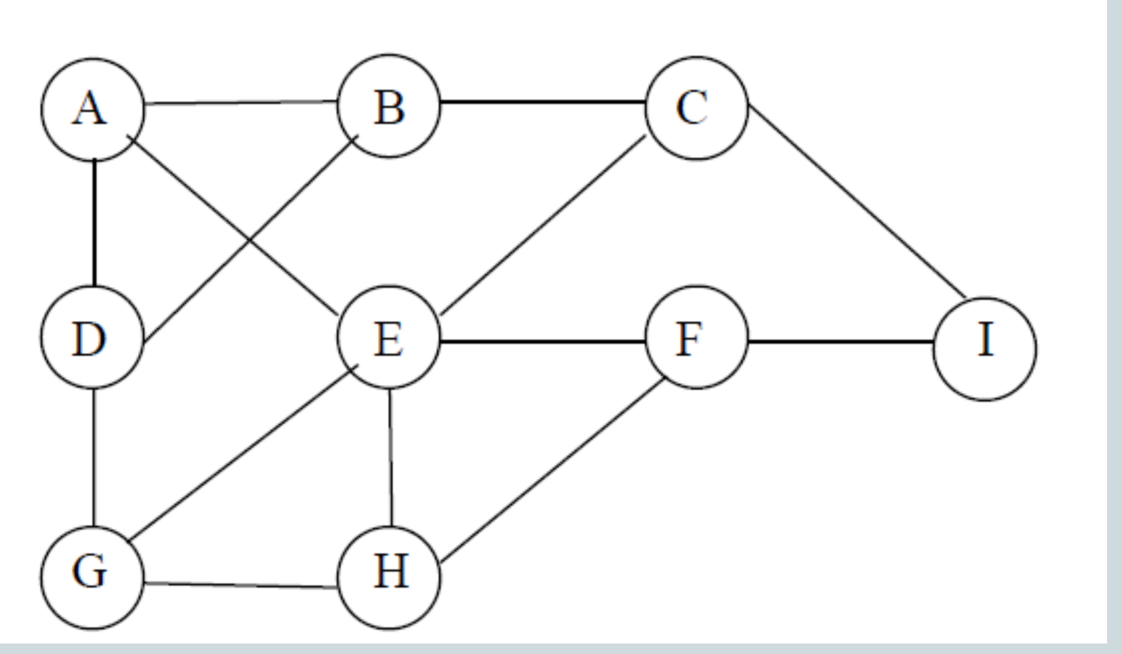


The discovery and finishing times have a parenthesis

Algorithms Related to BFS & DFS

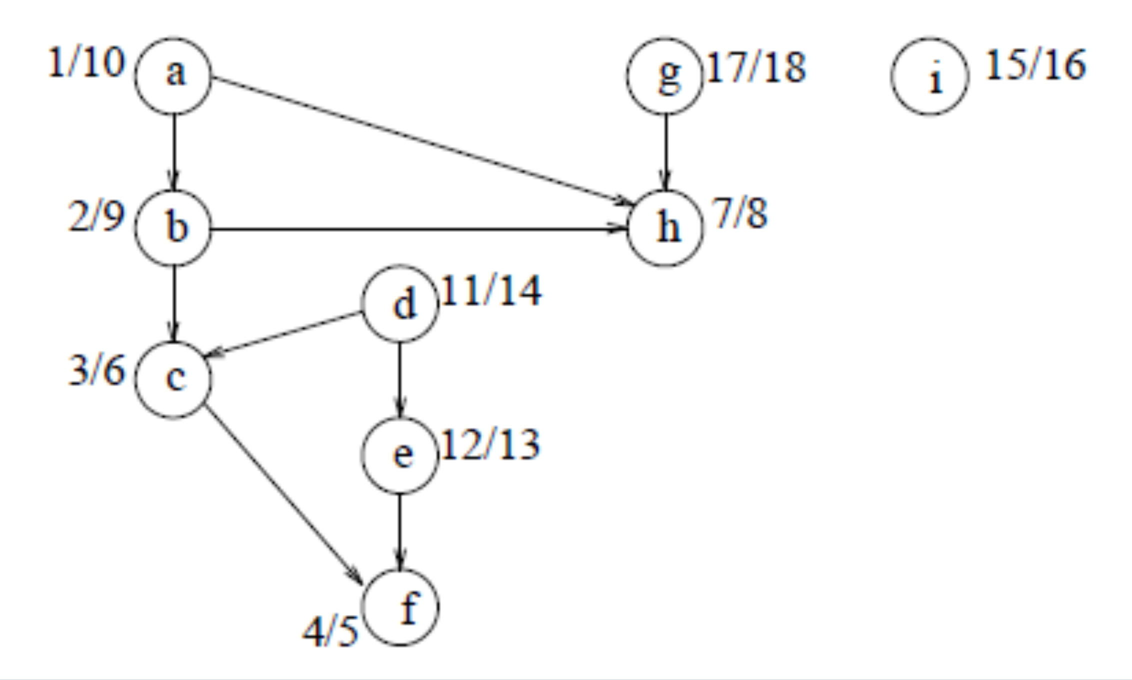
- We can solve all these problems:
- How could we test whether an undirected graph G is **connected**?
- How could we compute the **connected components** of G?
- How could we compute a **cycle** in G or report that it has no cycle?
- How could we compute a **path between any two vertices**, or report that no such path exists?
- How could we compute for every vertex v of G, the **minimum number of edges of any path between s and v**?

Example – run DFS



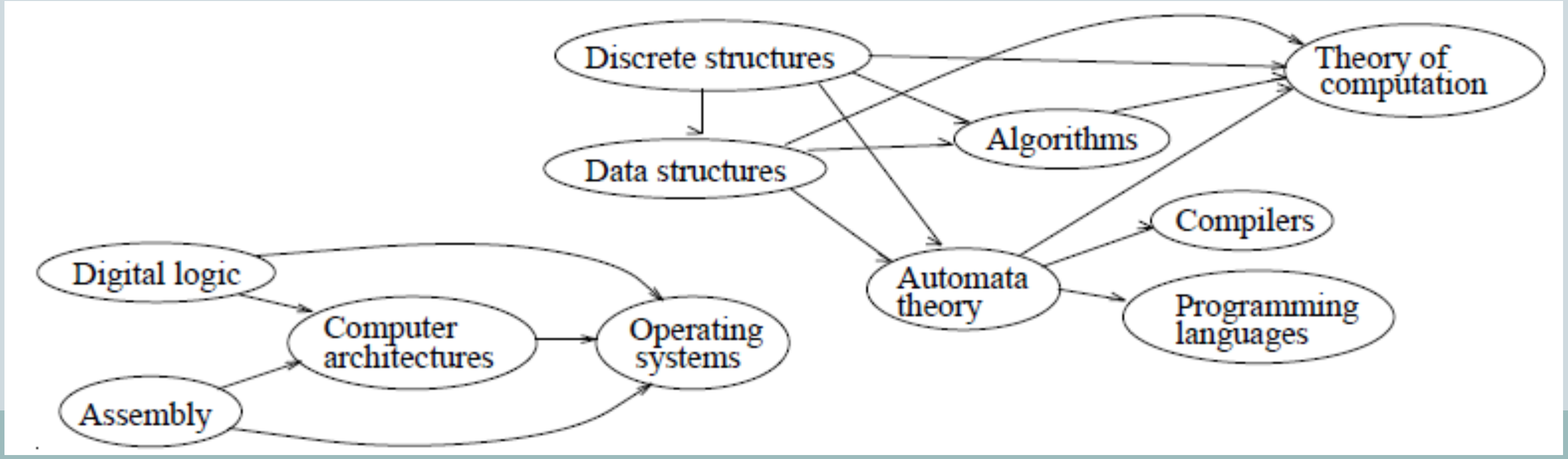
DAG

- *Directed Acyclic Graph: no directed cycle exist*

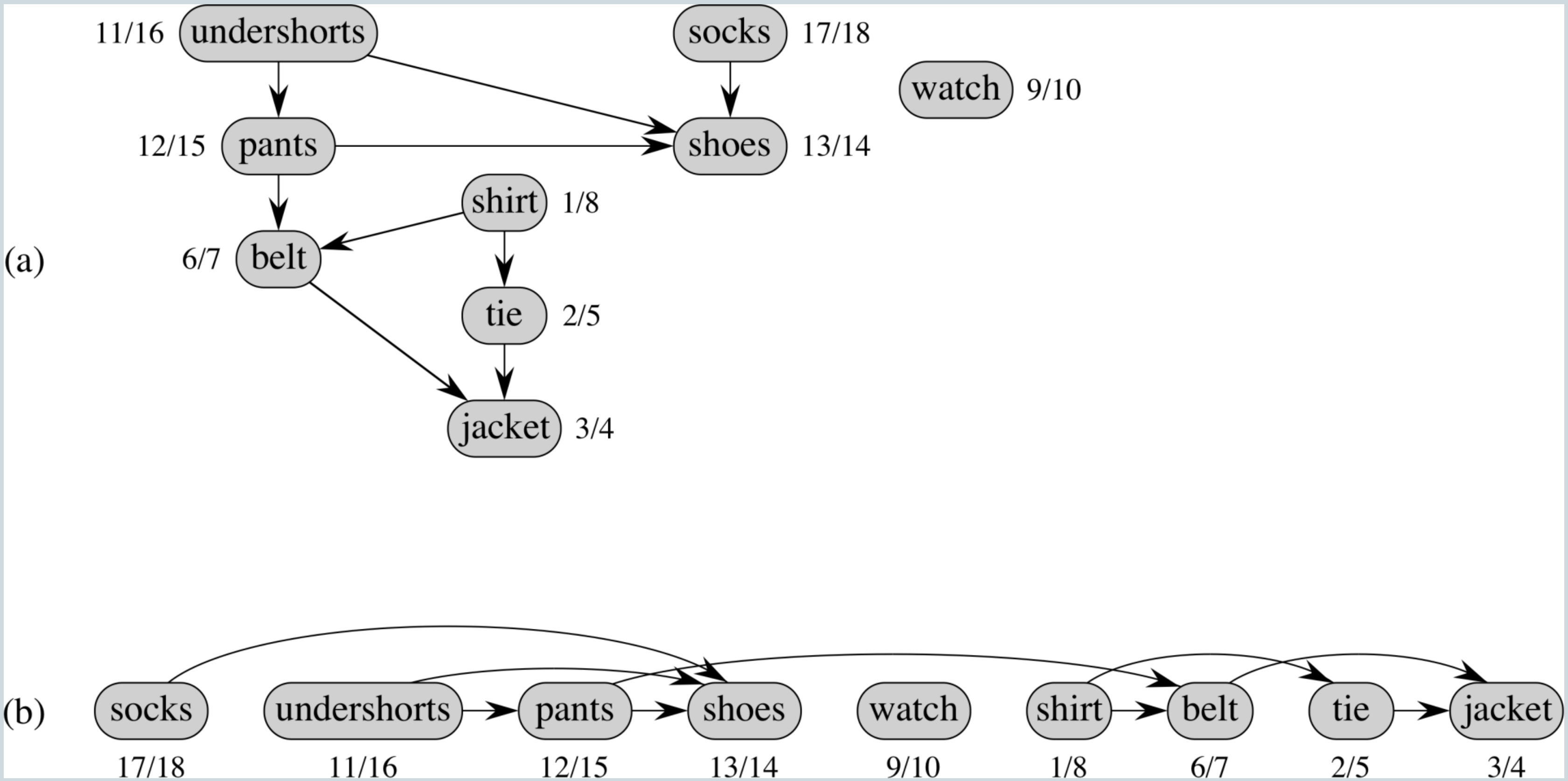


Topological sort

- A topological sort of a DAG G = (V,E) is a linear ordering of all of its vertices such that if G contains an edge (u, v) then u appears before v in the ordering.
- If the graph is not acyclic, then no linear ordering is possible.
- **Application**: Denotes precedencies among events.
- **Example**: Course prerequisites.



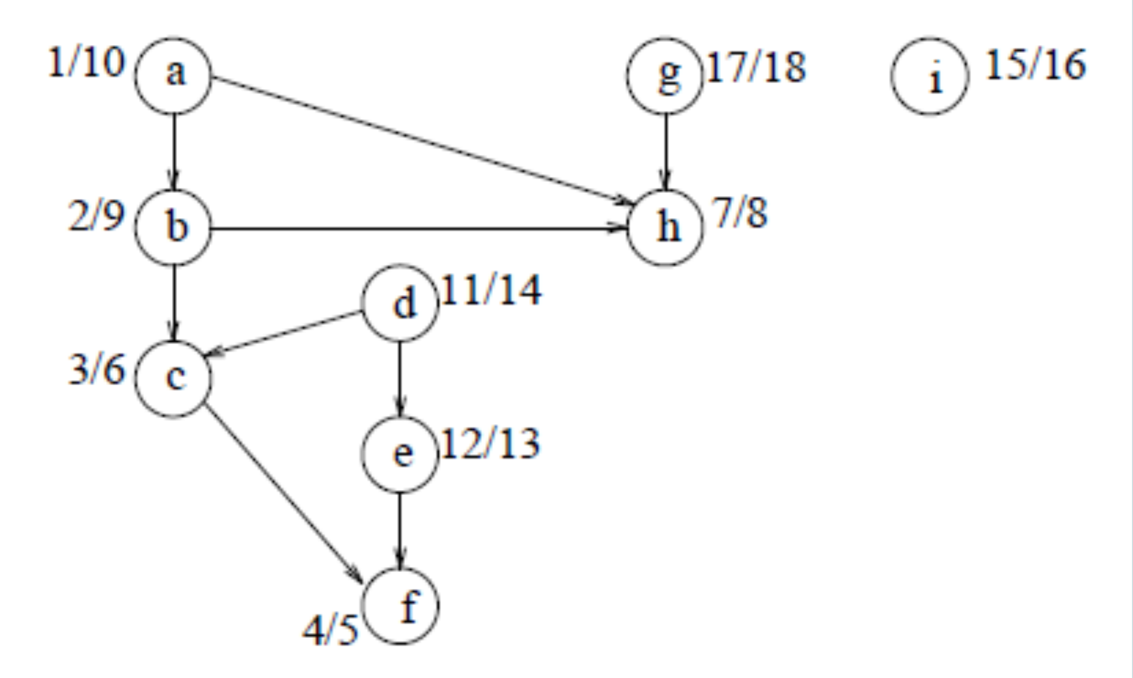
Another Example of Topological Sort

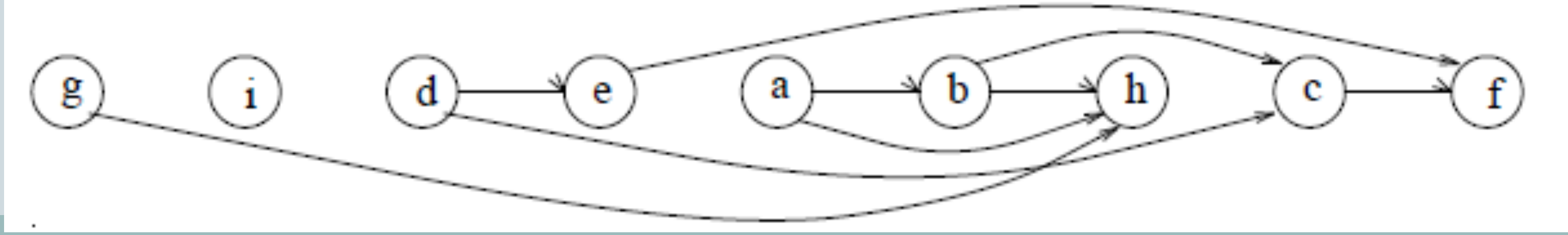


Algorithm of topological sort

- Topological_Sort(G)
- 1. Call DFS(G) to compute the finishing times f[u] for each vertex u.
- 2. As each vertex is finished, insert it at the front of a linked list.
- 3. Return the linked list of the vertices.
- **Time analysis:** O(V + E)

Example





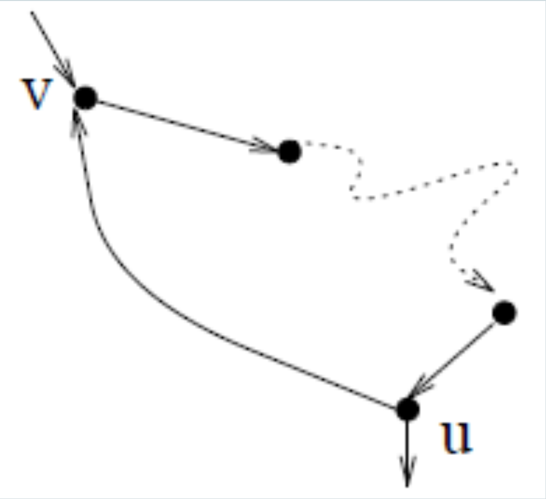
Acyclic and back edges

- **Lemma**: A directed graph G is acyclic *if and only if* a depth first search of G yields no back edges.

- Proof follows:

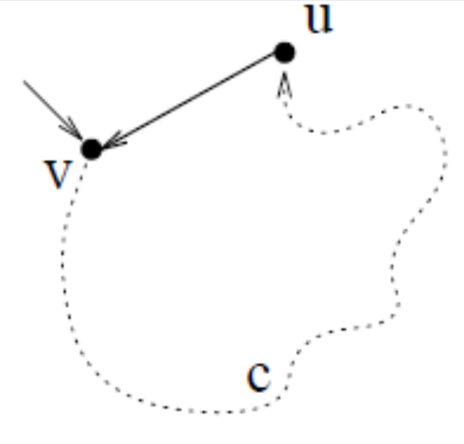
Proof of =>

- G is acyclic => G has no back edges
  - Suppose there is a back edge (u, v). Then, v is an ancestor of u in the depth first forest. Thus, there is a path from v to u in G, That path together with (u, v) form a cycle.



Proof of <=

- G has no back edges => G is acyclic
  - Assume that G contains a cycle c. We will show that a depth first search yields a back edge. Let v be the first edge discovered in c. By appropriately arranging the vertices of c in the adjacency lists, we can make the DFS reach u and thus, u becomes a descendant of v. Then, (u, v) becomes a back edge.



Theorem for topological sort

- **Theorem**: Topological_Sort(G) produces a topological sort of a DAG *G*.
- Proof:
- It suffices to show that for any pair of vertices u, v ε V , if there is an edge in G from u to v , then f[v] < f[u].
- Consider any edge (u, v) explored by DFS(G). When (u, v) is explored, v cannot be grey. (v would be an ancestor of u, and thus, (u, v) a back edge). Thus, v is either white or black.
  - If v is white, it becomes a descendant of u. => f[v] < f[u].
  - If v is black, then v is finished. => f[v] < f[u].