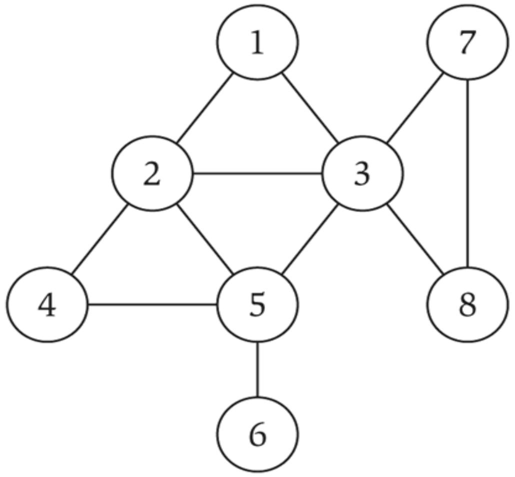
Graph Algorithms

Introduction to graph theory

Graph – mathematical object consisting of a set of:

- Denoted by $G = (V, E)$.
- V = **vertices** (nodes, points). $V(G)$ and V_G
- E = **edges** (links, arcs) between pairs of vertices. Also denoted by $E(G)$ and E_G ; $E \subseteq V \times V$
- **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) }

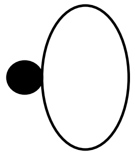
n = 8

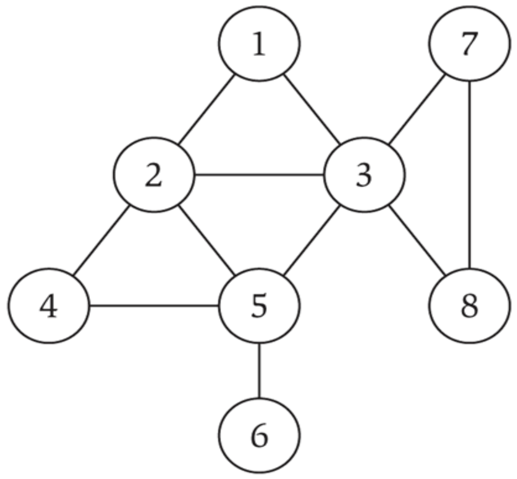
m = 11

Introduction to graph theory

For graph *G(V,E)*:

- If edge e=*(u,v)* ∈ *E(G)*, we say that *u* and *v* are **adjacent** or **neighbors**
- *u* and *v* are **incident** with *e*
- *u* and *v* are **end-vertices** of *e*
- An edge where the two end vertices are the same is called a **loop**, or a **self-loop**





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) }

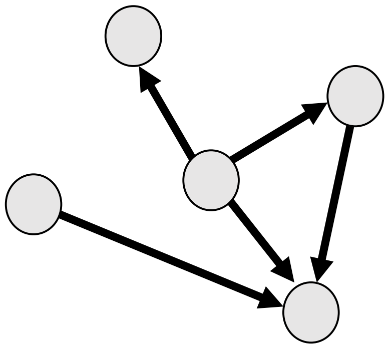
n = 8

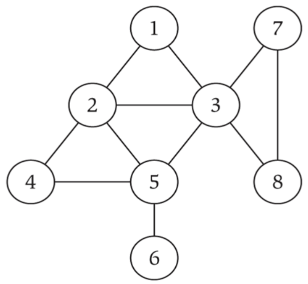
m = 11

Directed graph (digraph)

- Directed edge - ordered pair of vertices $(\boldsymbol{u},\boldsymbol{v})$

- A graph with directed edges is called a directed graph or digraph



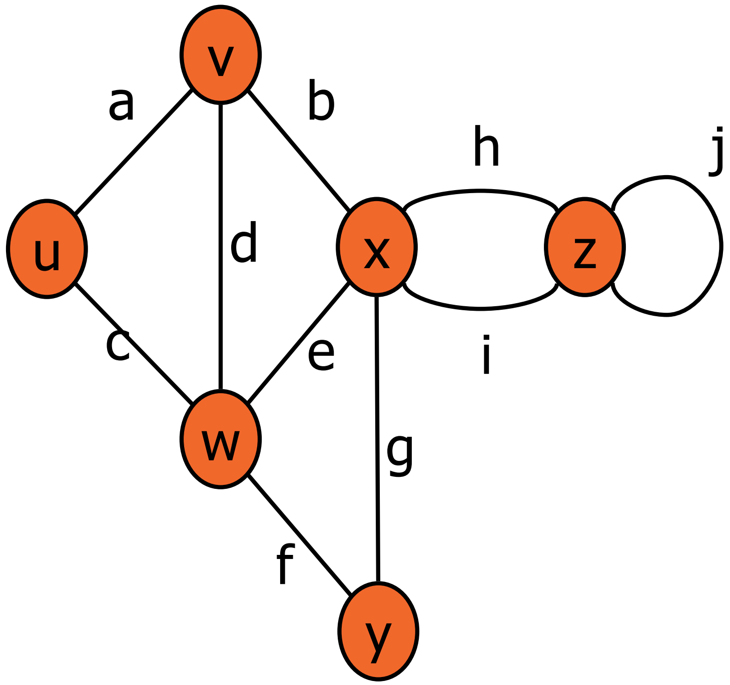


- Undirected edge- unordered pair of vertices $(\boldsymbol{u},\boldsymbol{v})$

- A graph with undirected edges is an undirected graph or simply a graph

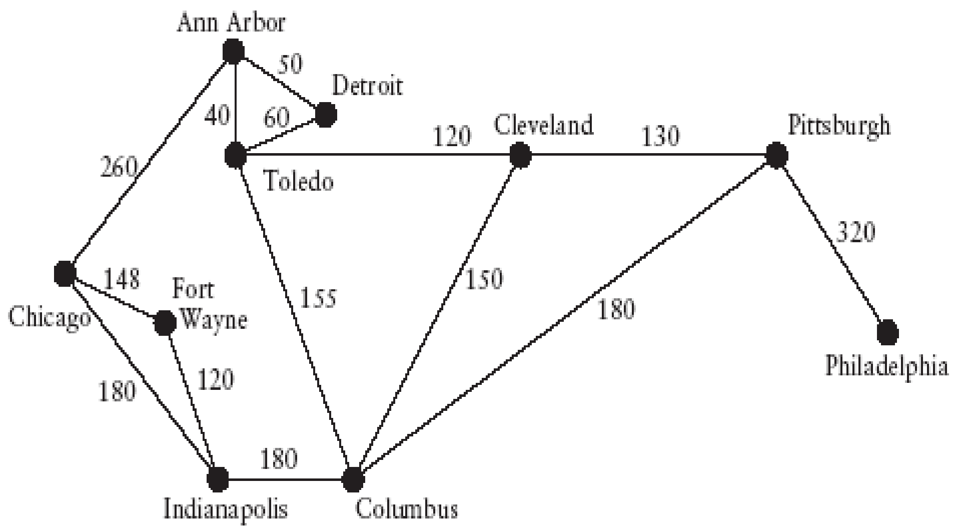
Terminology

- End vertices (or endpoints) of an edge
  - u and v are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on v
- Adjacent vertices
  - u and v are adjacent
- Degree of a vertex
  - x has degree 5
- Self-loop
  - j is a self-loop



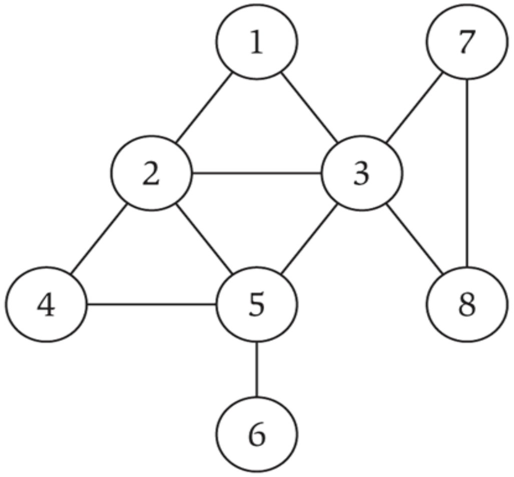
Weighted Graphs

- The edges in a graph may have values associated with them known as their weights
- A graph with weighted edges is known as a weighted graph

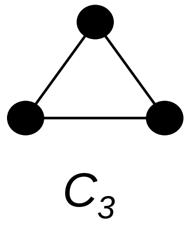


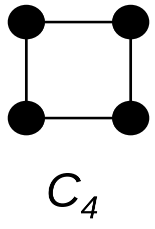
Terminology

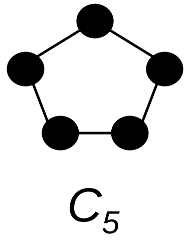
- A **walk** in an undirected graph G = (V, E) is a sequence P of vertices v_1, v_2, …, v_{k-1}, v_k with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E.
- A walk is a **path** if all vertices are distinct.
- A **cycle** is a path in which the first and final vertices are the same
- Cycles can be denoted by C_k, where k is the number of vertices in the cycle



(1, 3, 7, 8, 3, 5) is a walk
(6, 5, 3, 2) is a path
(1, 2, 3, 1) is a cycle

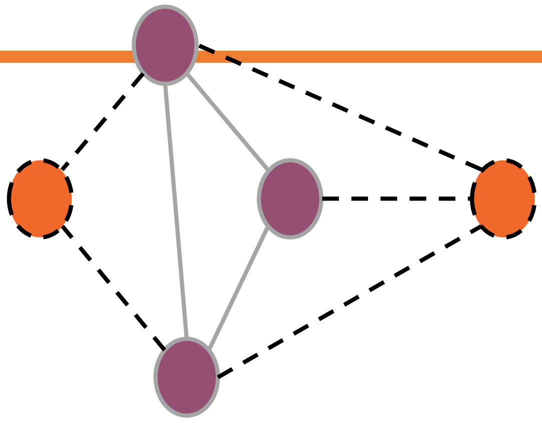


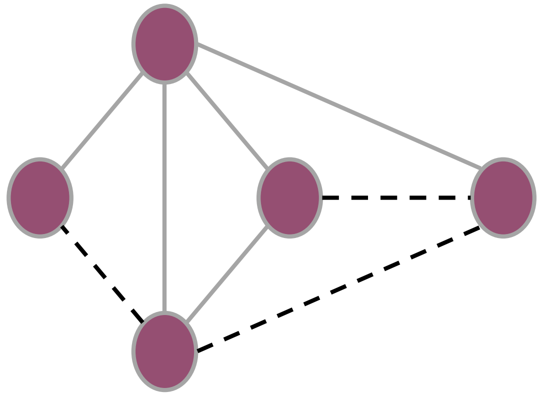




Subgraphs

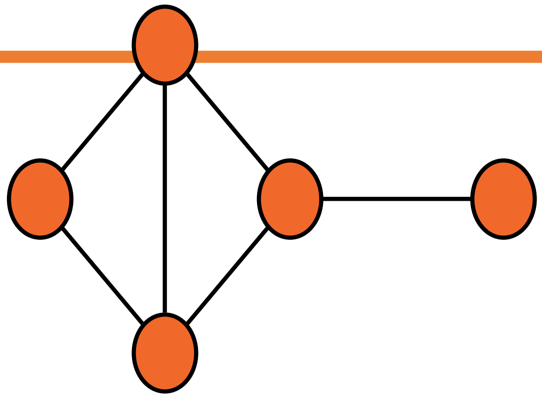
- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G

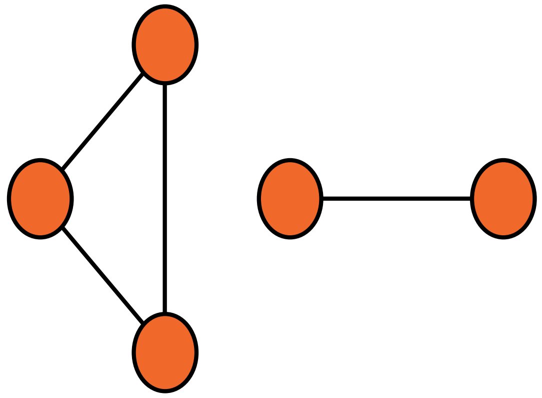

Subgraph


Spanning subgraph

Connectivity

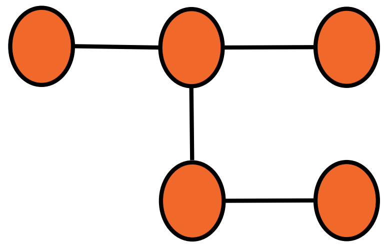
- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G


Connected graph


Non connected graph with two connected components

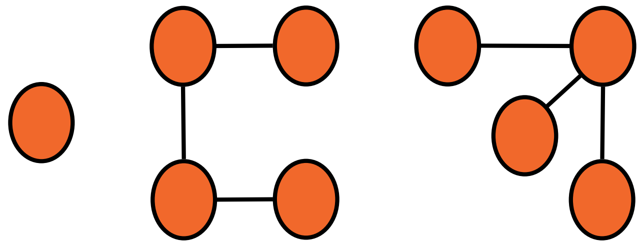
Trees and Forests

- A tree is an undirected graph T such that
  - T is connected
  - T has no cycles



Tree

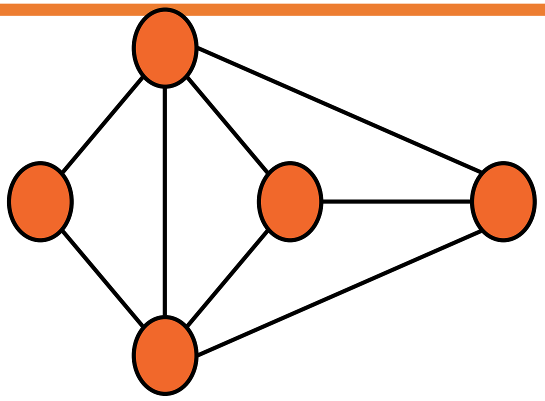
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



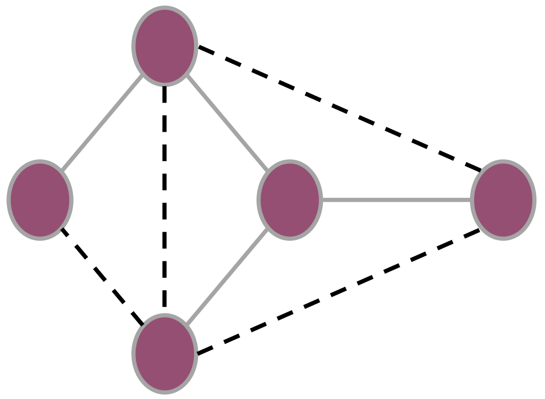
Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree



Graph



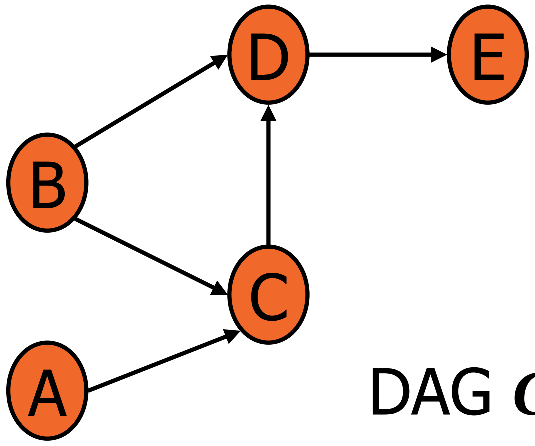
Spanning tree

Directed graphs

- A directed walk is a sequence of directed edges, where the head of each edge is the tail of the next; a directed path is a directed walk without repeated vertices.
- Vertex v is reachable from vertex u in a directed graph G if and only if G contains a directed walk (and therefore a directed path) from u to v.
- A directed graph is strongly connected if every vertex is reachable from every other vertex.

DAG

- A directed acyclic graph (DAG) is a digraph that has no directed cycles



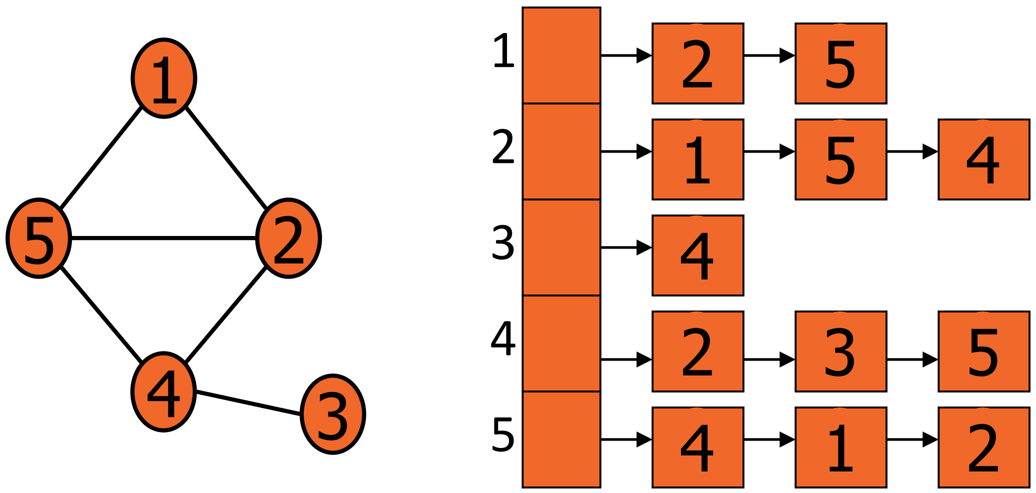
DAG G

Representation of Graphs

Two standard ways:

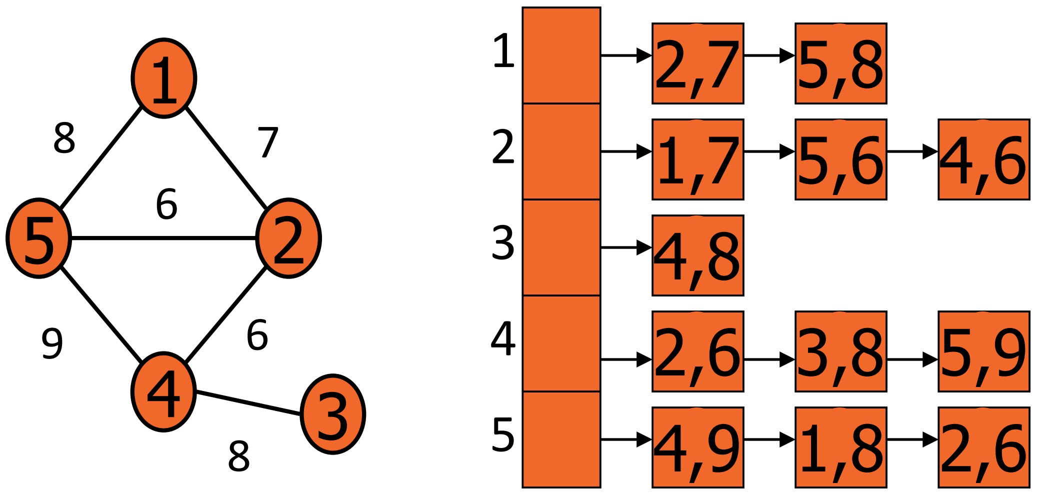
- Adjacency List
  - preferred for sparse graphs (|E| is much less than |V|^2)
  - Unless otherwise specified we will assume this representation
- Adjacency Matrix
  - Preferred for dense graphs

Adjacency List



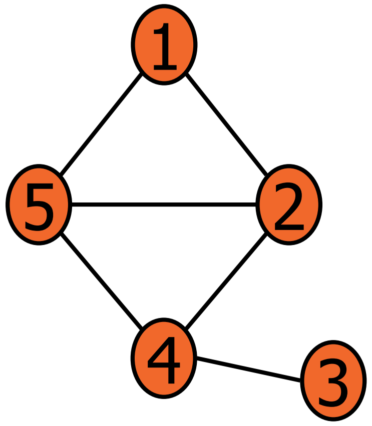
- An array Adj of |V| lists, one per vertex
- For each vertex u in V,
    - Adj[u] contains all vertices v such that there is an edge (u,v) in E (i.e. all the vertices adjacent to u)
- Space required Θ(|V|+|E|) (Following CLRS, we will use V for |V| and E for |E|) thus Θ(V+E)

Adjacency List (weights)



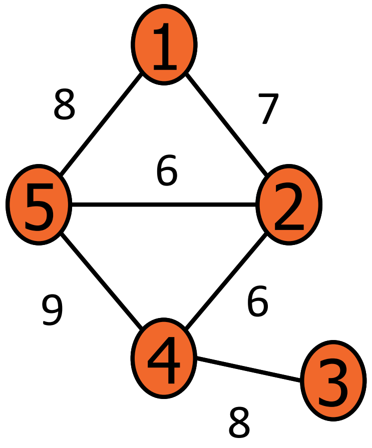
- An array Adj of |V| lists, one per vertex
- For each vertex u in V,
  - Adj[u] contains all vertices v such that there is an edge (u,v) in E (i.e. all the vertices adjacent to u)
- Space required Θ(|V|+|E|) (Following CLRS, we will use V for |V| and E for |E|) thus Θ(V+E)

Adjacency Matrix



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

Adjacency Matrix (weights)



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 7 & 0 & 0 & 8 \\ 2 & 7 & 0 & 0 & 6 & 6 \\ 3 & 0 & 0 & 0 & 8 & 0 \\ 4 & 0 & 6 & 8 & 0 & 9 \\ 5 & 8 & 6 & 0 & 9 & 0 \end{array}$$

Comparisons

	Standard Adj list (linked list)	Adj Matrix
Space	$\Theta(V+E)$	$\Theta(V^2)$
Test if $(u,v)\in E$		
Test if $u\text{->}v\in E$		
List v’s (out)neighbors		
List all edges		
Insert edge (u,v)		
Delete edge (u,v)		

Comparisons

	Standard Adj list (linked list)	Adj Matrix
Space	$\Theta(V+E)$	$\Theta(V^2)$
Test if $(u,v) \in E$	$O(\min\{deg(u),deg(v)\}) = O(V)$	$O(1)$
Test if $u\text{->}v \in E$	$O(deg(u)) = O(V)$	$O(1)$
List v's (out)neighbors	$\Theta(deg(v)) = O(V)$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge (u,v)	$O(1)$	$O(1)$
Delete edge (u,v)	$O(deg(u)+deg(v)) = O(V)$	$O(1)$

Graph Traversals

- For solving most problems on graphs
  - Need to systematically visit all the vertices and edges of a graph
- Two major traversals
  - Breadth-First Search (BFS)
  - Depth-First Search(DFS)

BFS

- Starts at some source vertex s
- Discover every vertex that is reachable from s
- Also produces a BFS tree with root s and including all reachable vertices

- Discovers vertices in increasing order of distance from s
    - Distance between v and s is the minimum number of edges on a path from s to v

- i.e. discovers vertices in a series of layers

BFS : vertex colors stored in color[]

- Initially all undiscovered: white
- When first discovered: gray
  - They represent the frontier of vertices between discovered and undiscovered
  - Frontier vertices stored in a queue
  - Visits vertices across the entire breadth of this frontier
- When processed: black

Review: Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.

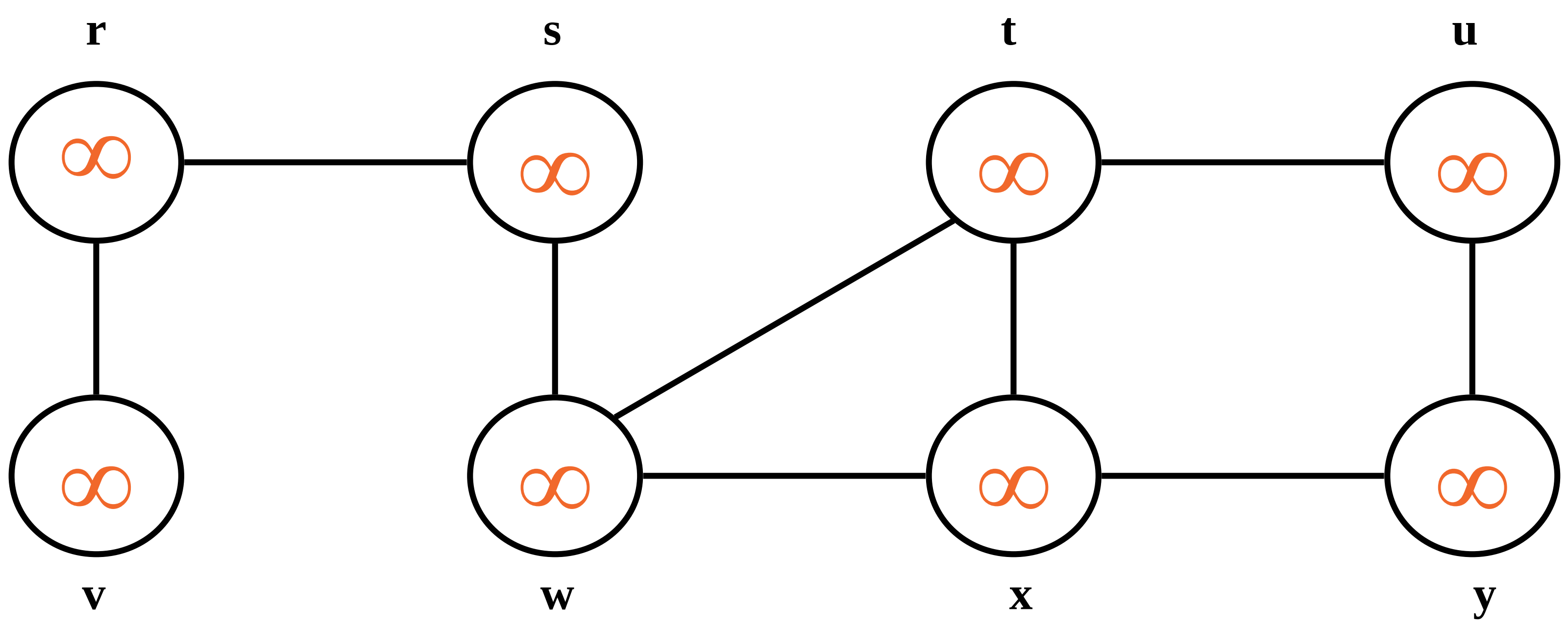
Breadth-First Search

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

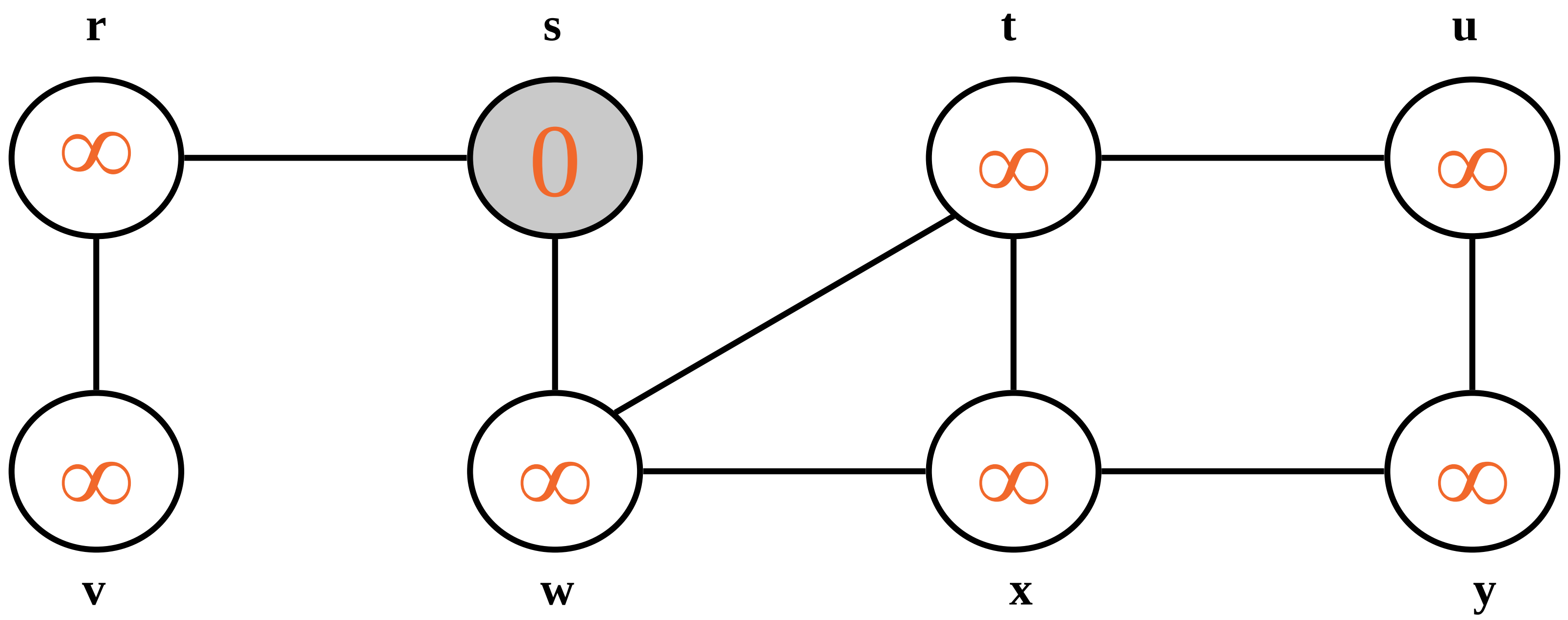
Review: Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    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.p = u;
                ENQUEUE(Q, v);
        }
        u.color = BLACK;
    }
}
```

Breadth-First Search: Example

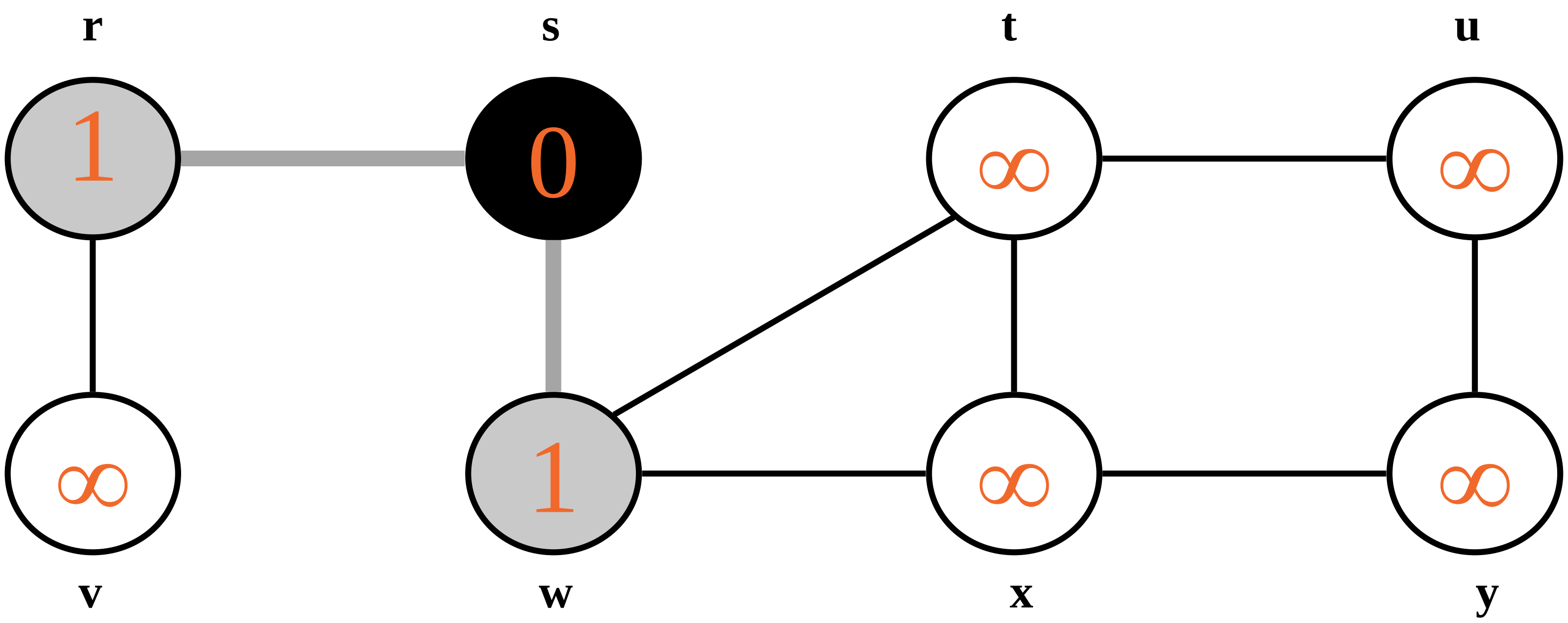


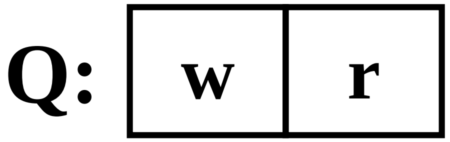
Breadth-First Search: Example



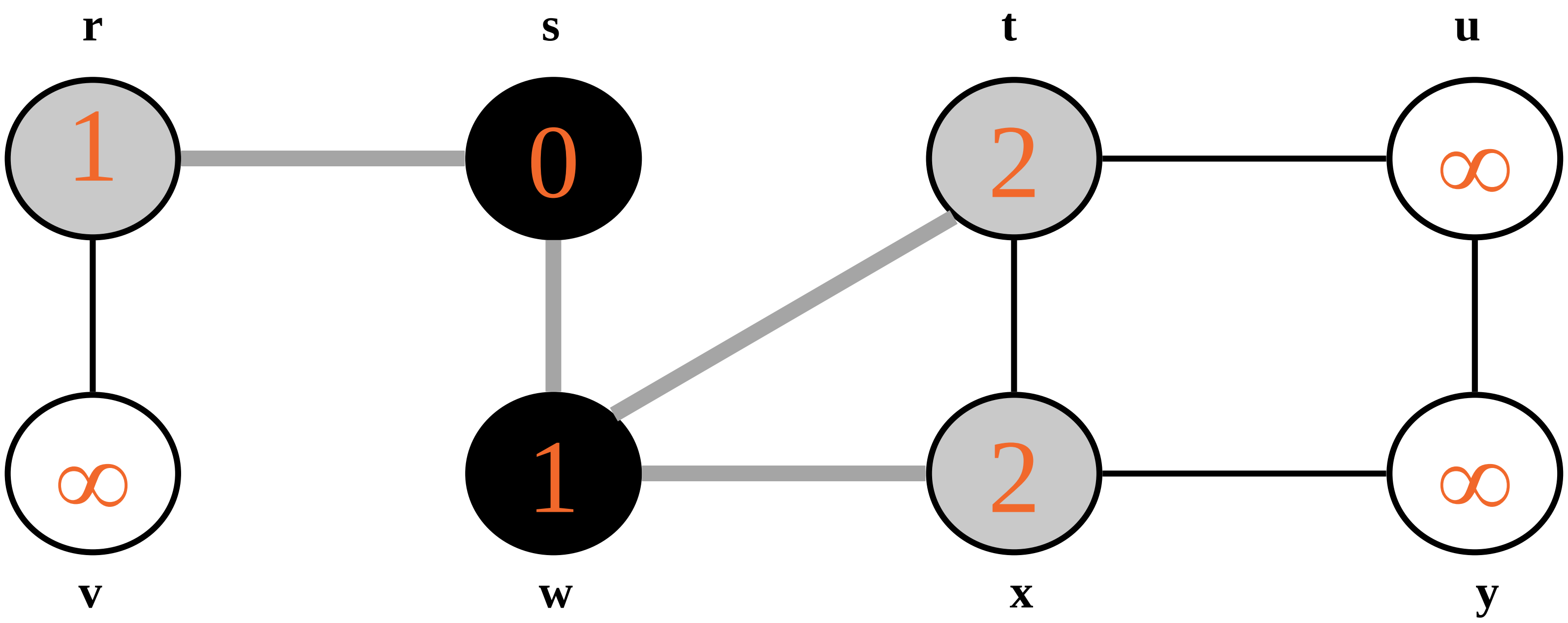
Q: $\boxed{s}$

Breadth-First Search: Example



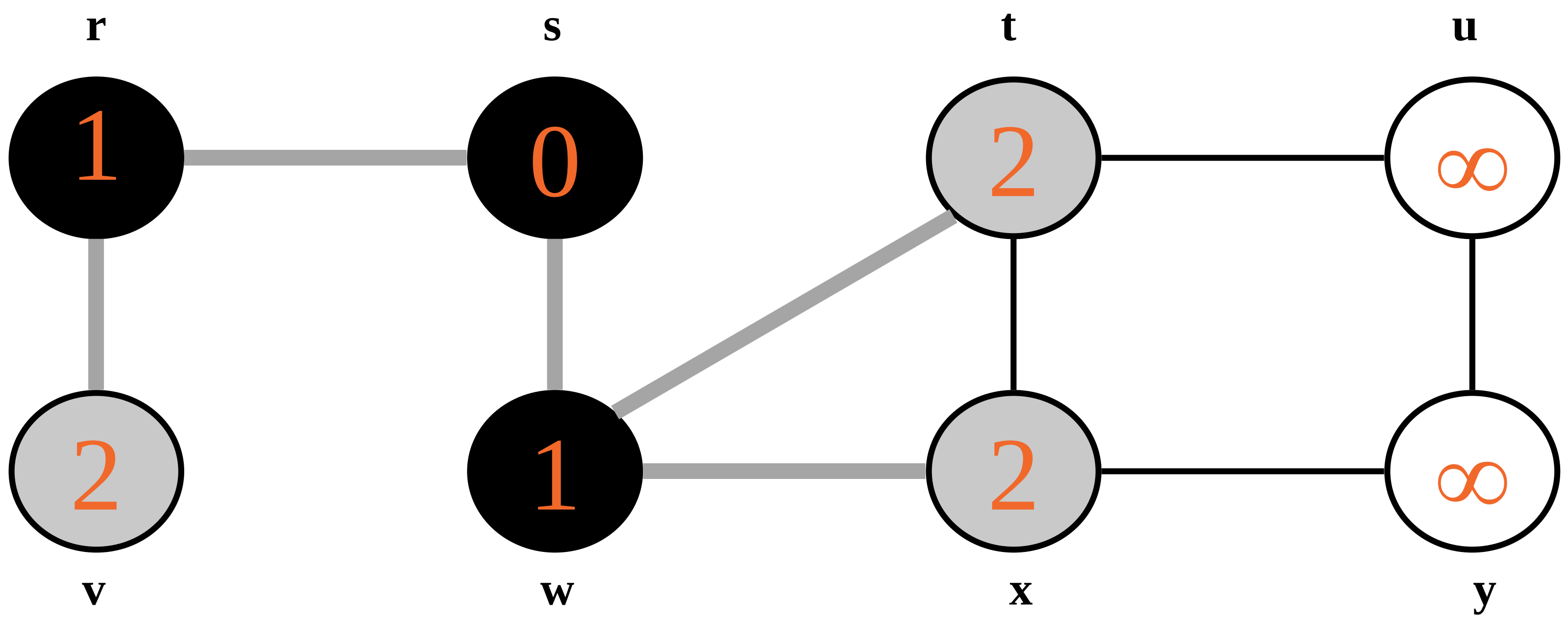


Breadth-First Search: Example



Q:	r	t	x

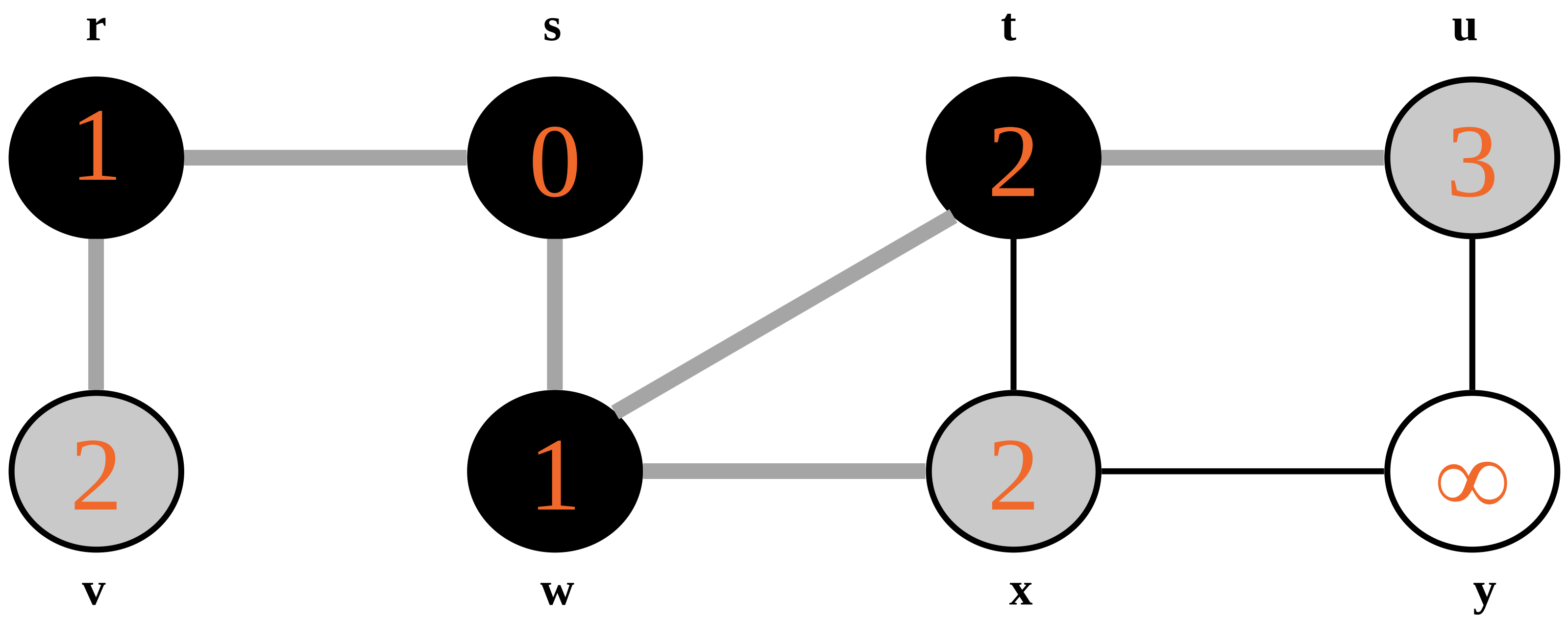
Breadth-First Search: Example



Q:

t	x	v

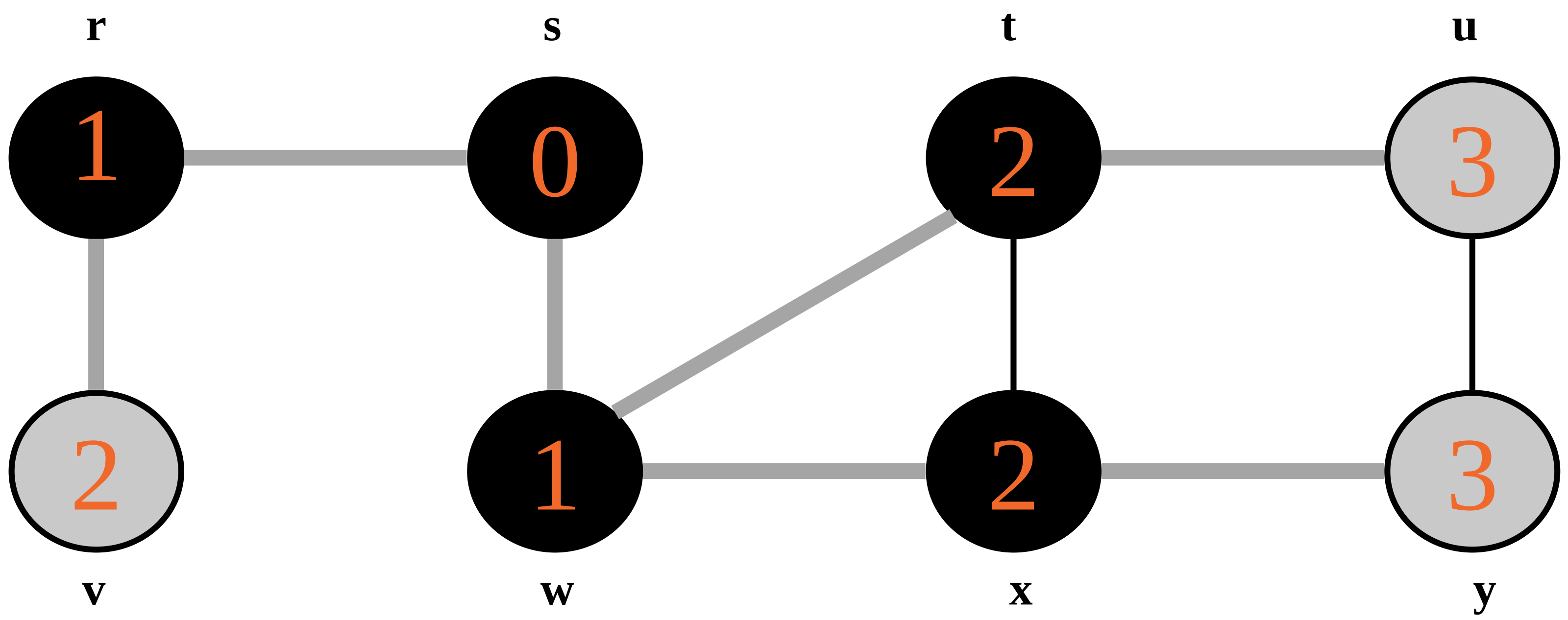
Breadth-First Search: Example



Q:

x	v	u

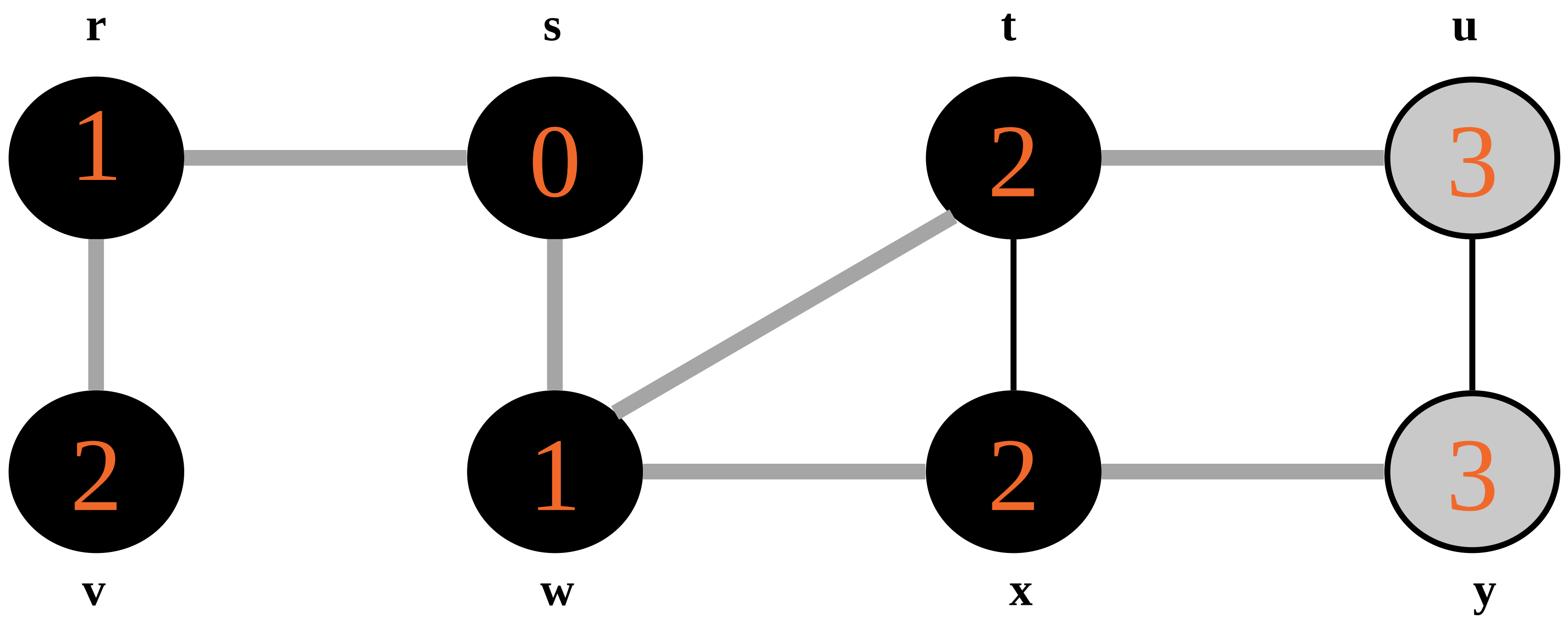
Breadth-First Search: Example

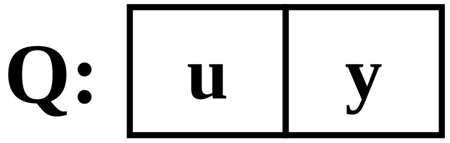


Q:

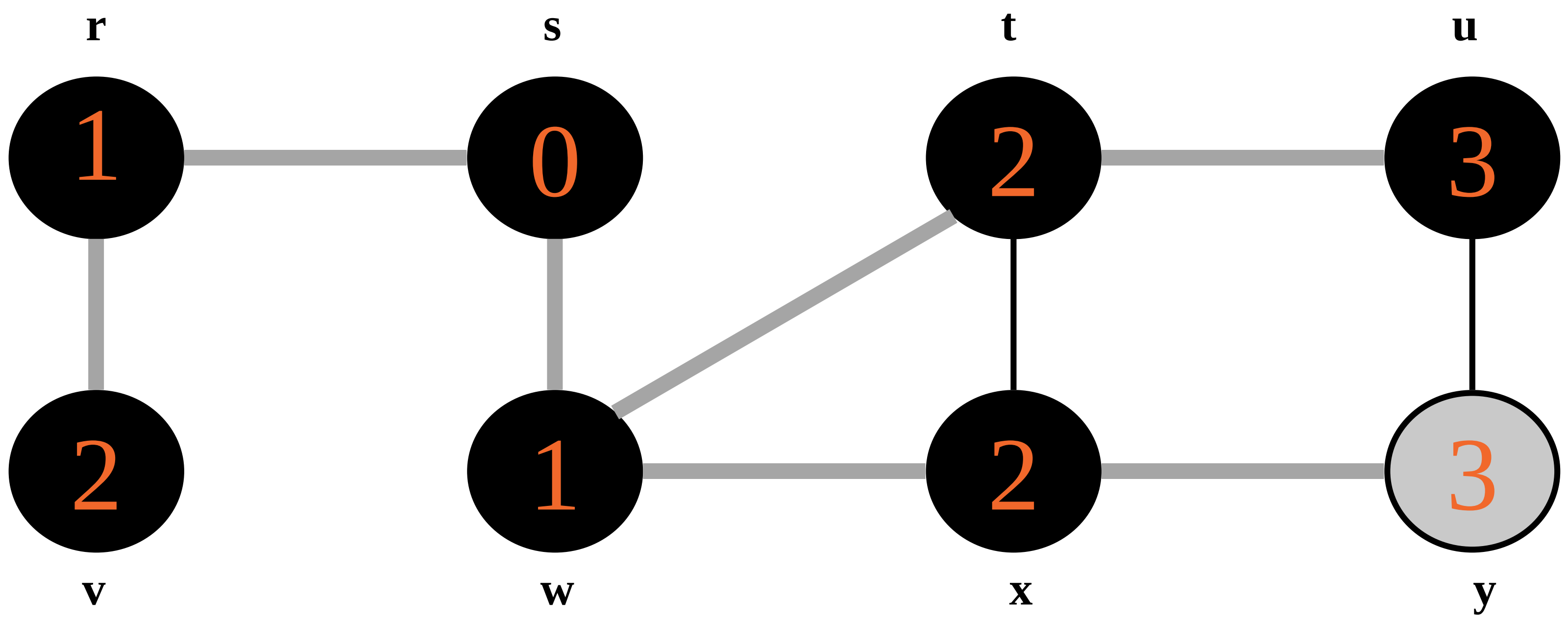
v	u	y

Breadth-First Search: Example



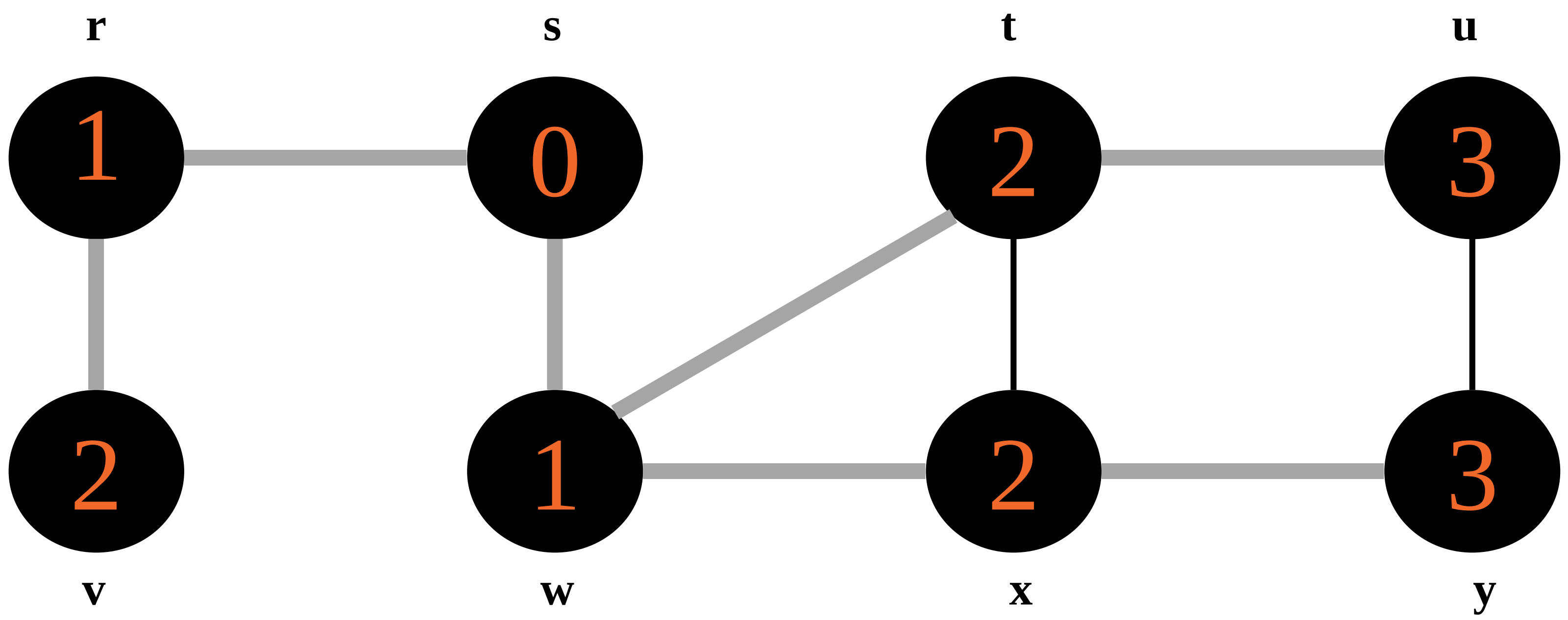


Breadth-First Search: Example



Q: $\boxed{y}$

Breadth-First Search: Example



Q: Ø

BFS: The Code Again

```
BFS(G, s) {
    initialize vertices;
    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.p = 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

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 vertex
  - Shortest-path distance $\delta(s,v)$ = 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

Analysis

- Each vertex is enqueued once and dequeued once : O(V)

- Each adjacency list is traversed once:

$$\sum_{u \in V} \deg(u) = O(E)$$

- Total: O(V+E)

BFS and shortest paths

Theorem: Let G=(V,E) be a directed or undirected graph, and suppose BFS is run on G starting from vertex s. During its execution BFS discovers every vertex v in V that is reachable from s. Let δ(s,v) denote the number of edges on the shortest path form s to v. Upon termination of BFS, d[v] = δ(s,v) for all v in V.

Print vertices on shortest path from s to v.

```
PrintPath(G,s,v) {
    if v == s
        print s;
    elseif v.p == nil
        print "No Path";
    else PrintPath(G, s, v.p)
        print v;
}
```

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
- recursive

Time stamps, color[u] and pred[u] as before

We store two time stamps:

- d[u] or u.d: the time vertex u is first discovered (discovery time)
- f[u] or u.f: the time we finish processing vertex u (finish time)

color[u] or u.color

- Undiscovered: white
- Discovered but not finished processing: gray
- Finished: black

pred[u] or u.π

- Pointer to the vertex that first discovered u

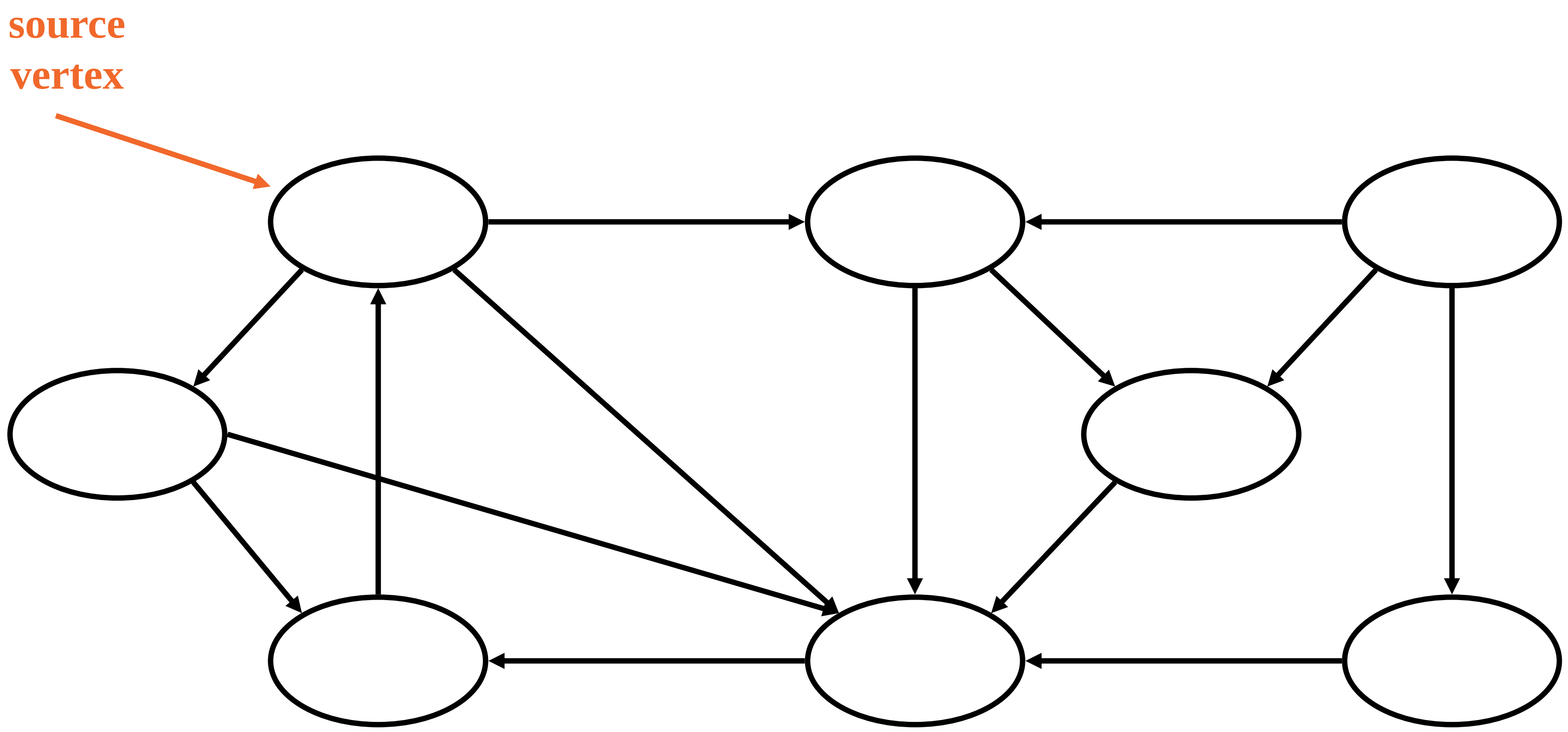
Depth-First Search: The Code

```
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(G,u);
    }
}
```

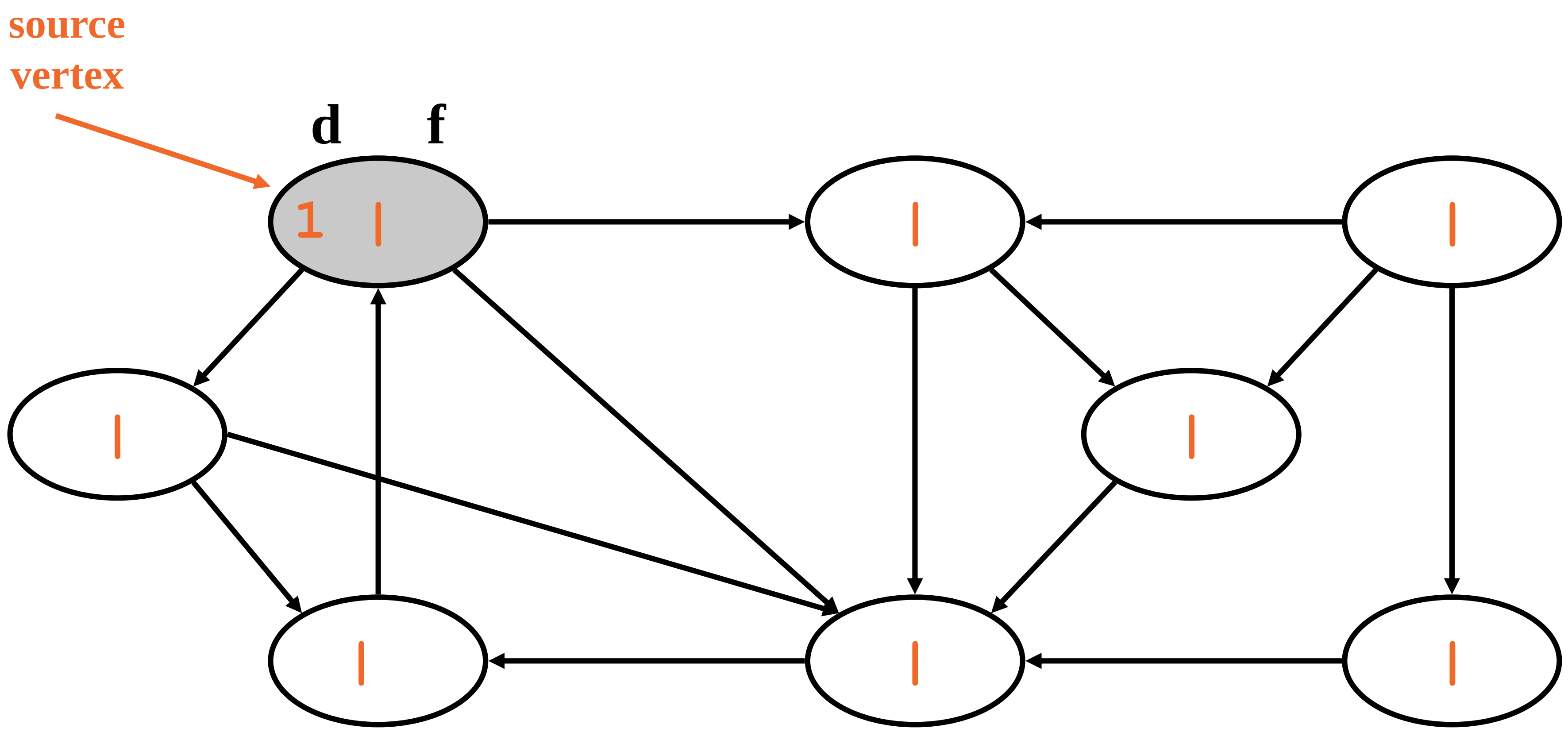
```
DFS_Visit(G, u)
{
    u.color = GREY;
    time = time+1;
    u.d = time;
    for each v ∈ G.Adj[u]
    {
        if (v.color == WHITE)
            DFS_Visit(G,v);
    }
    u.color = BLACK;
    time = time+1;
    u.f = time;
}
```

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

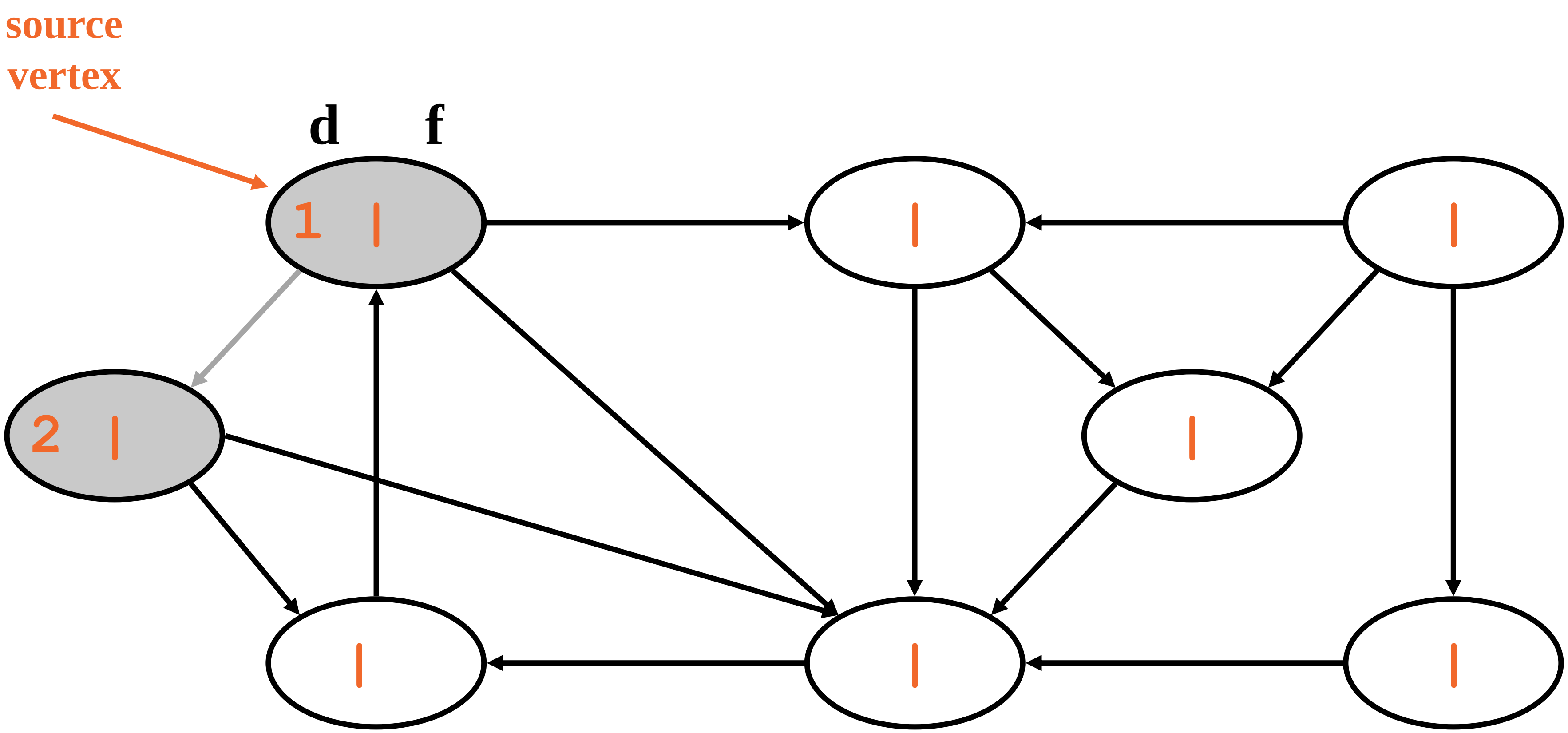
DFS Example



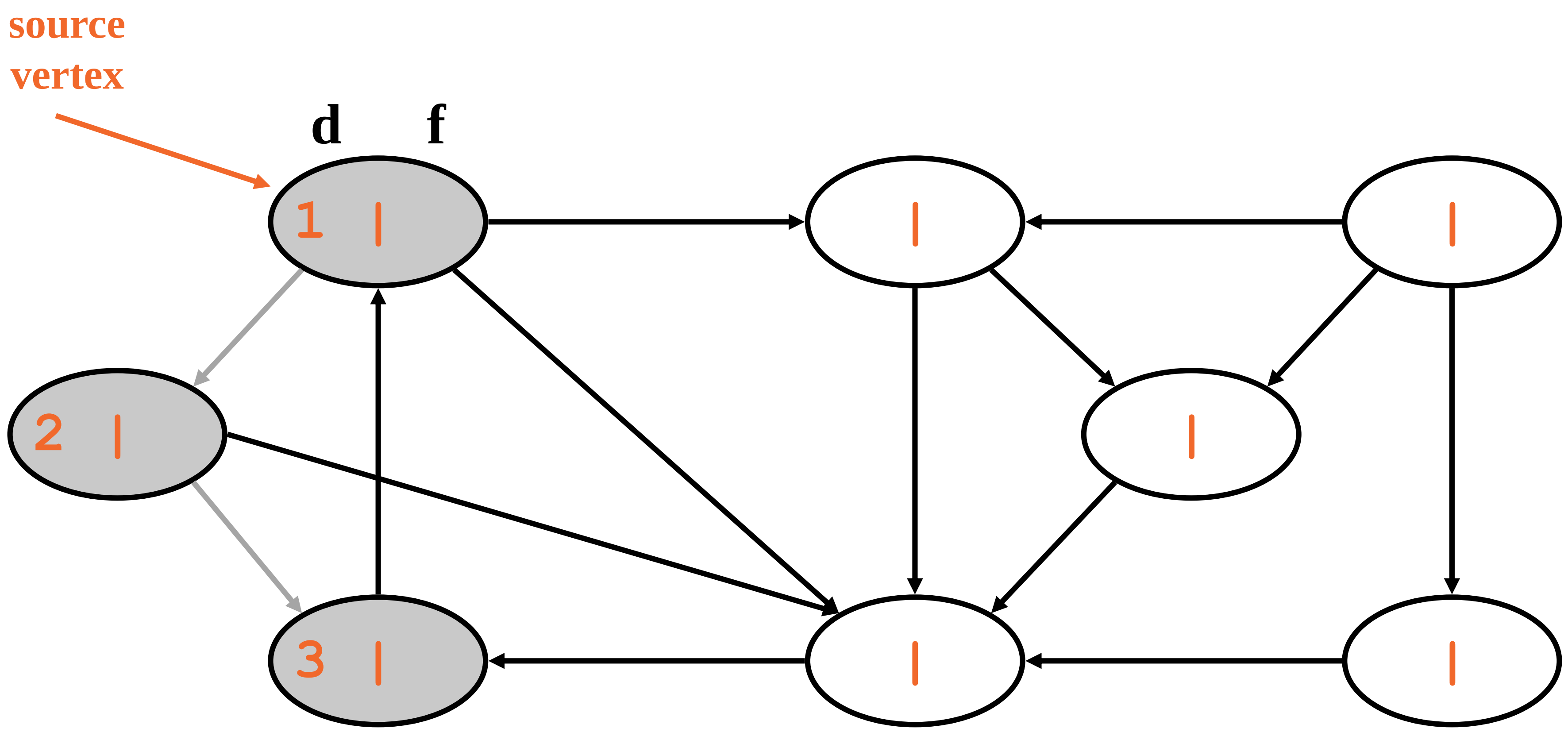
DFS Example



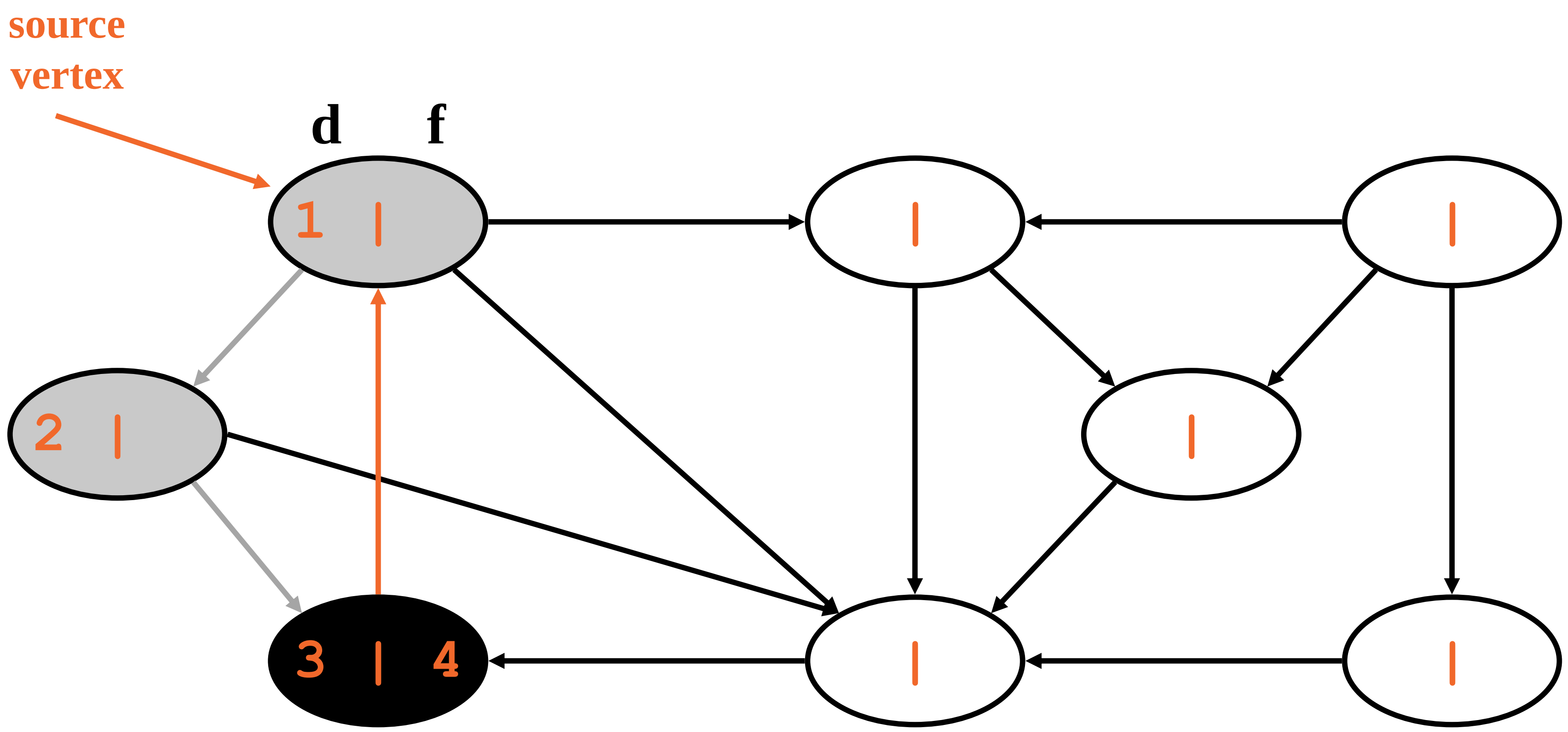
DFS Example



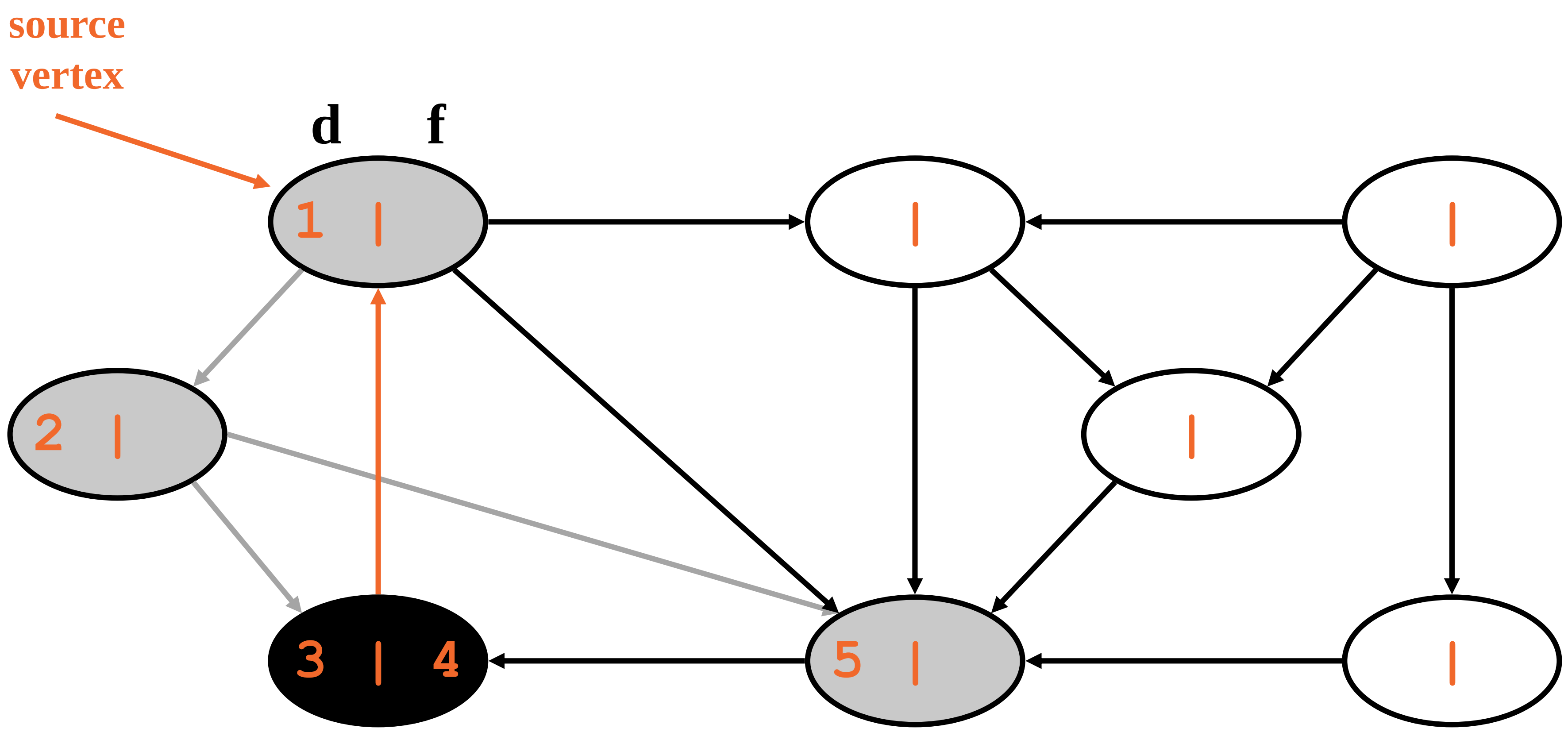
DFS Example



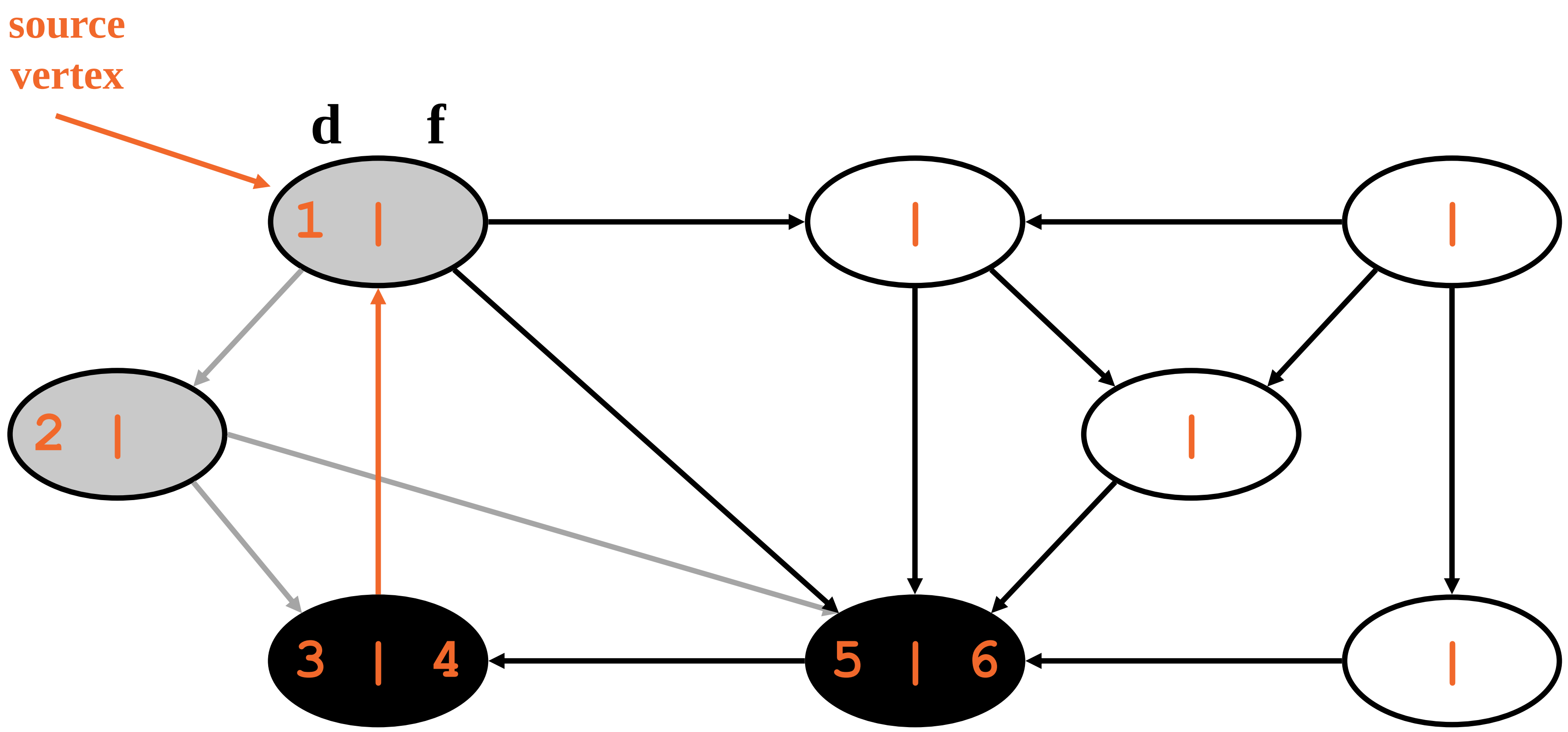
DFS Example



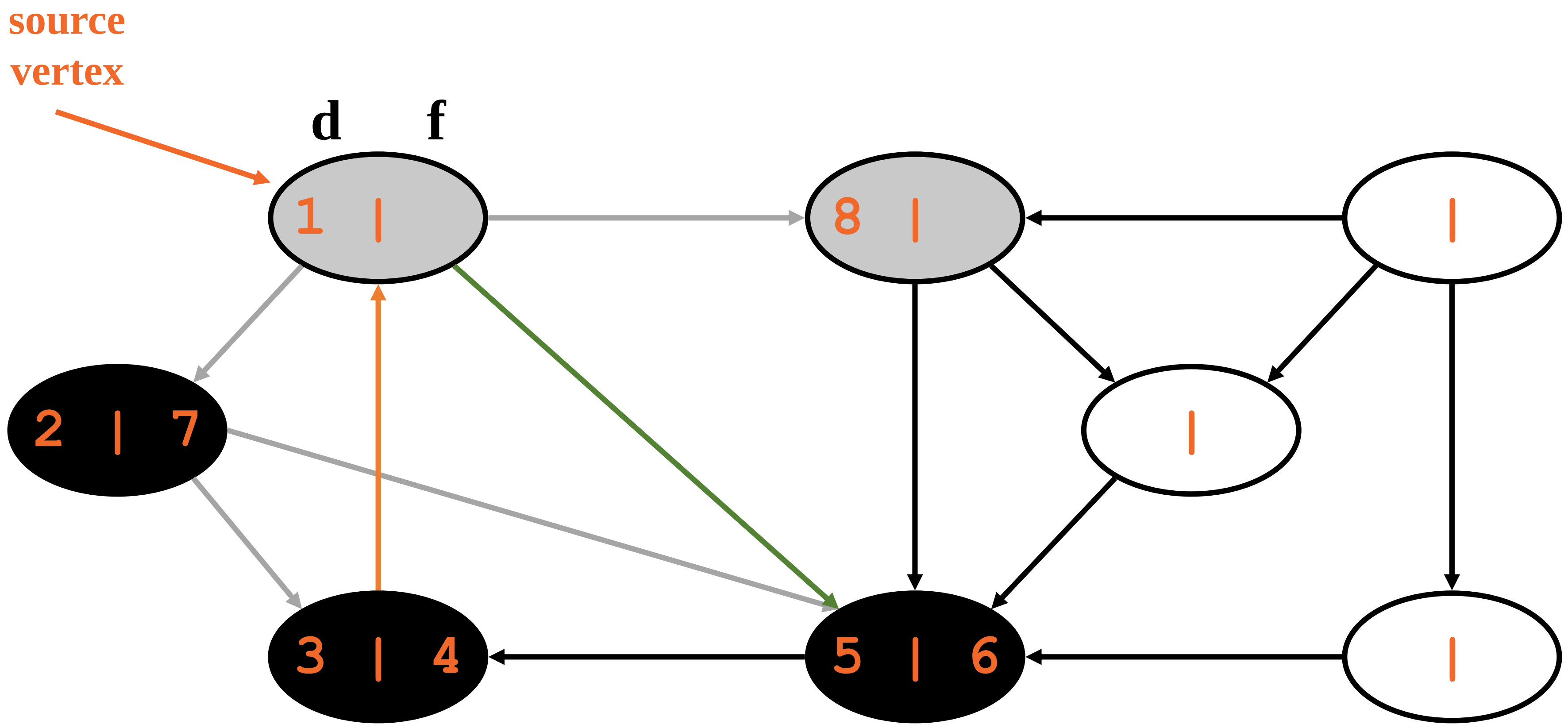
DFS Example



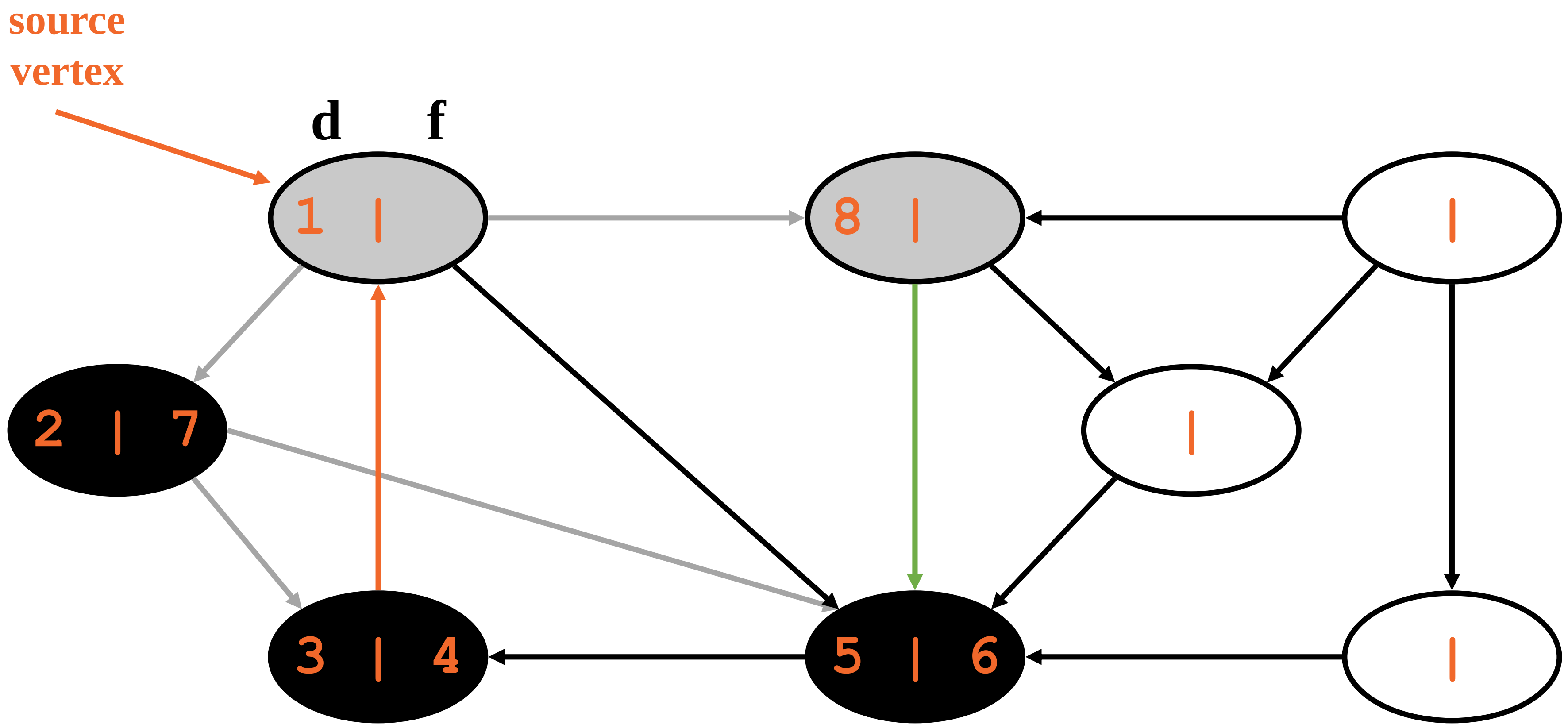
DFS Example



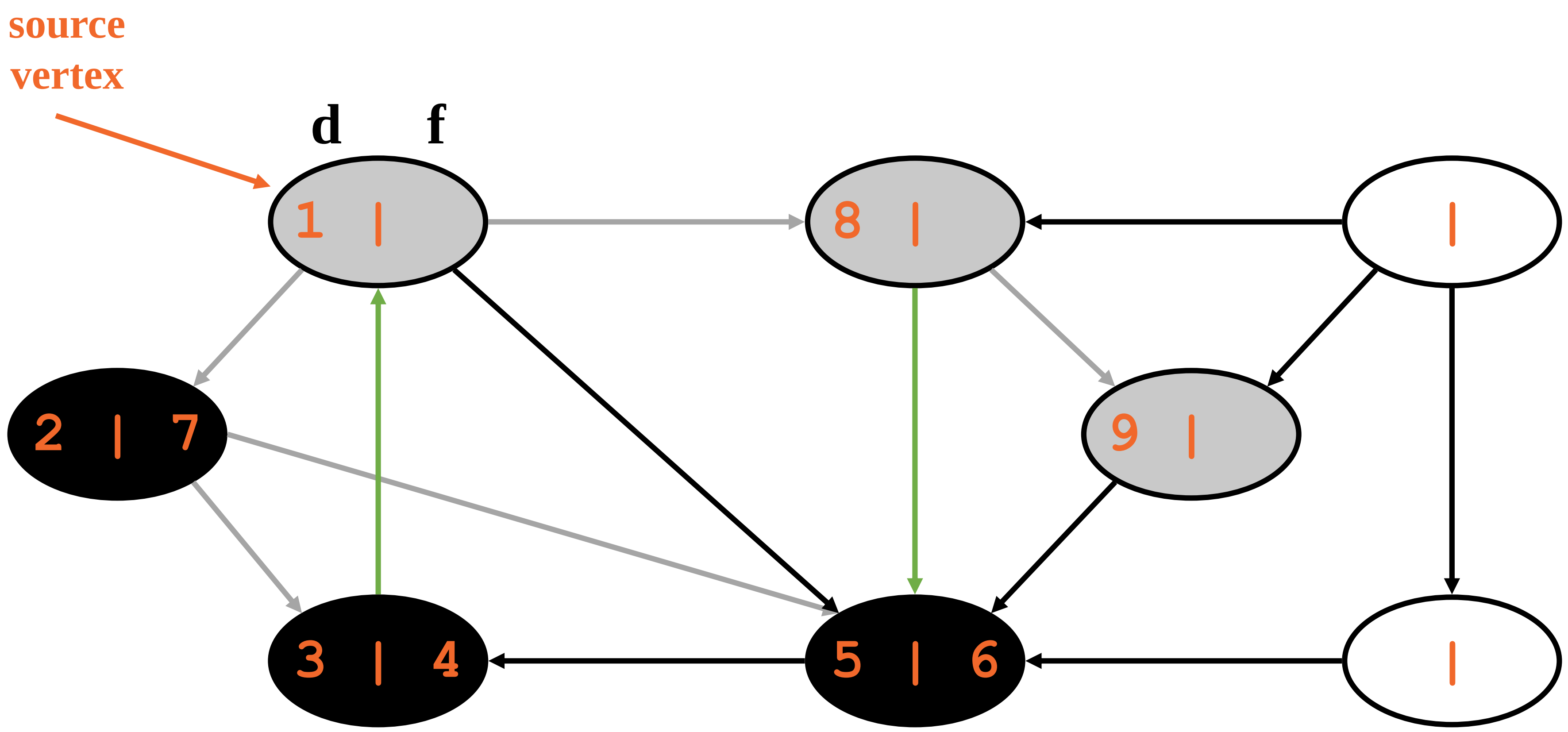
DFS Example



DFS Example

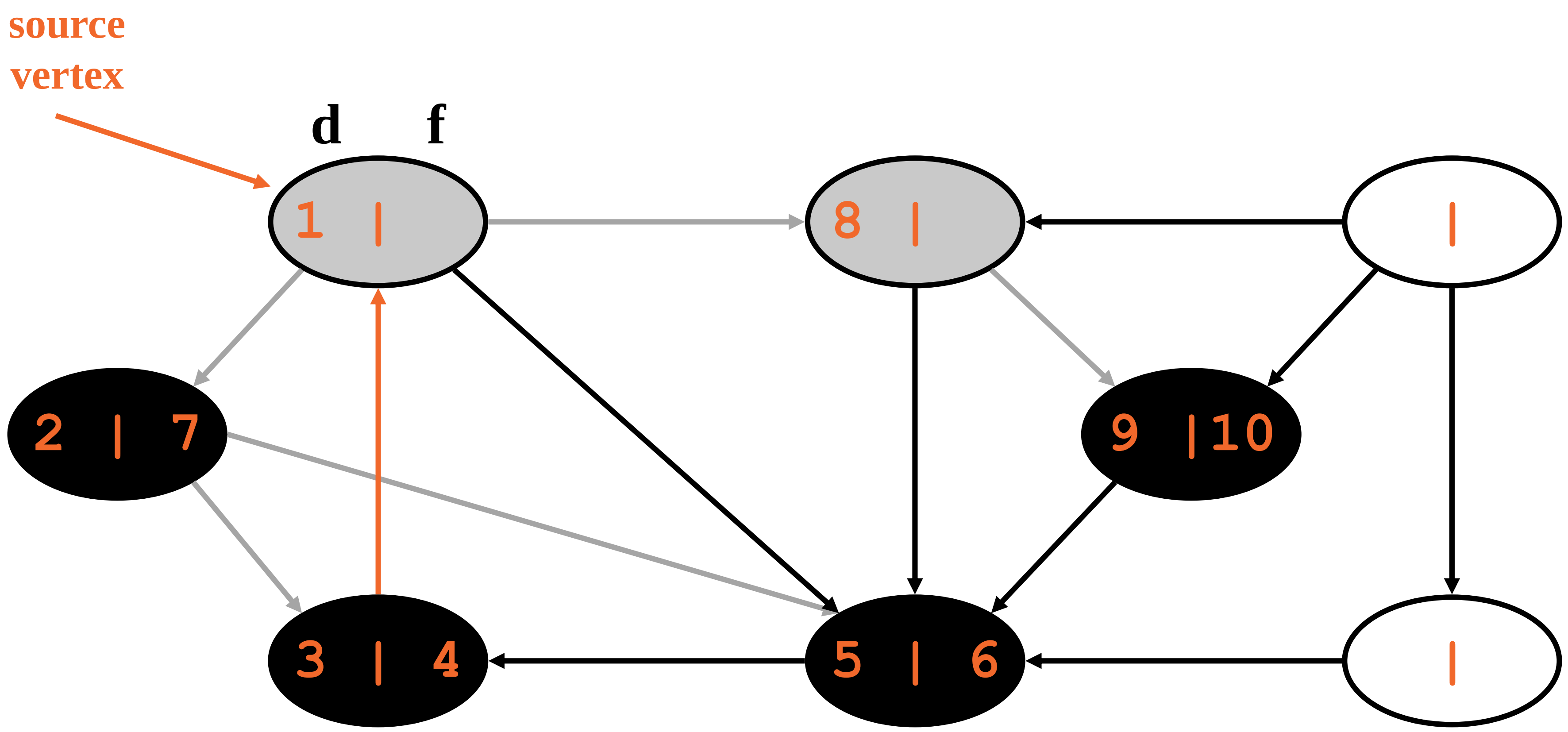


DFS Example

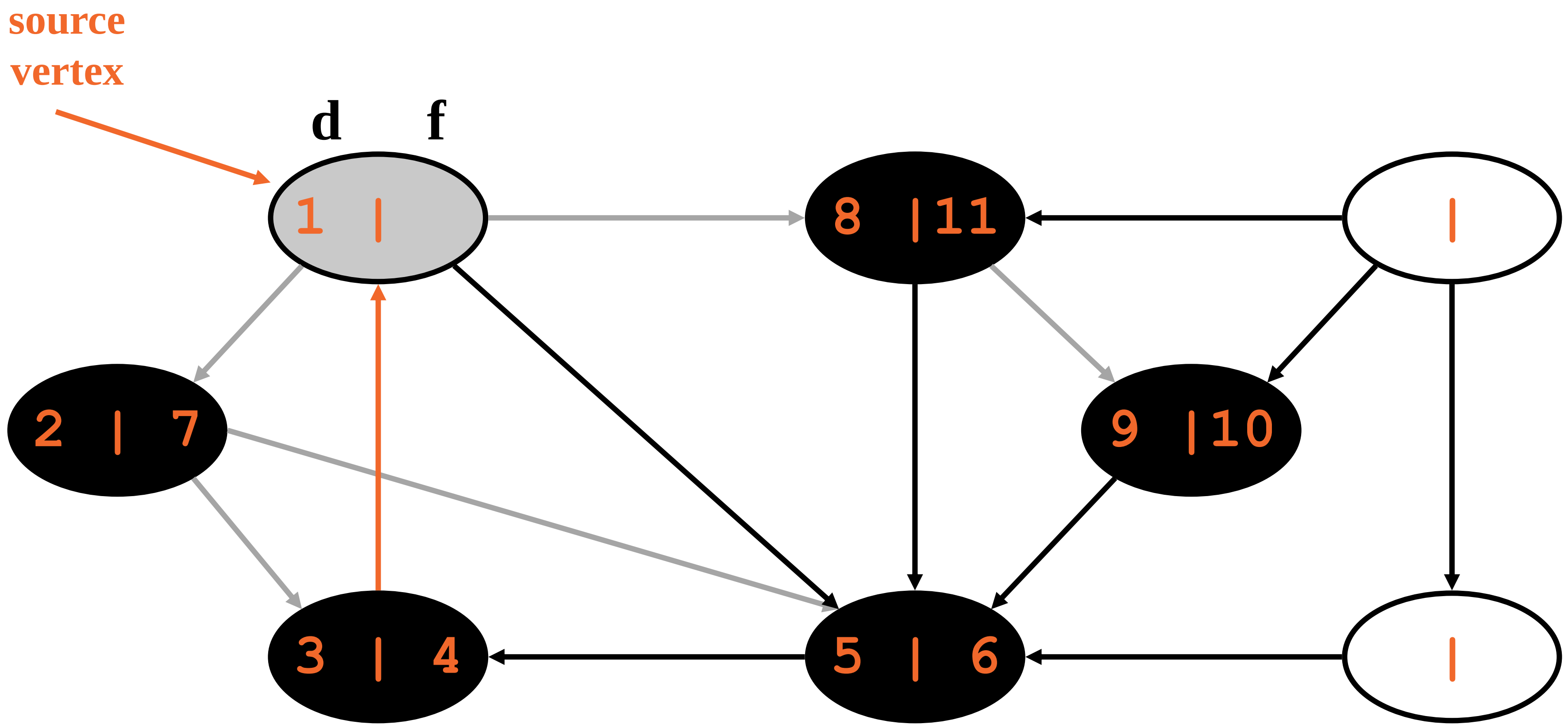


What is the structure of the grey vertices?
What do they represent?

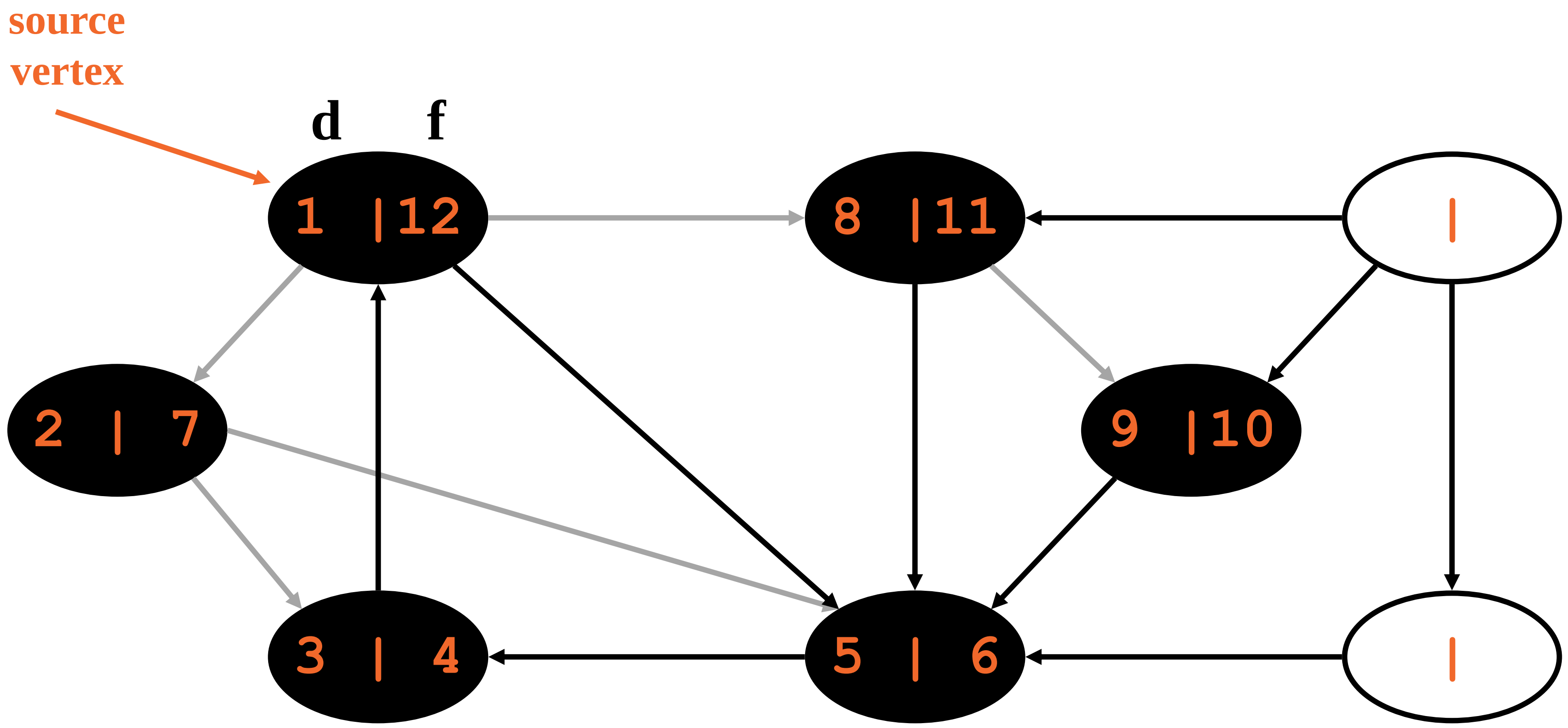
DFS Example



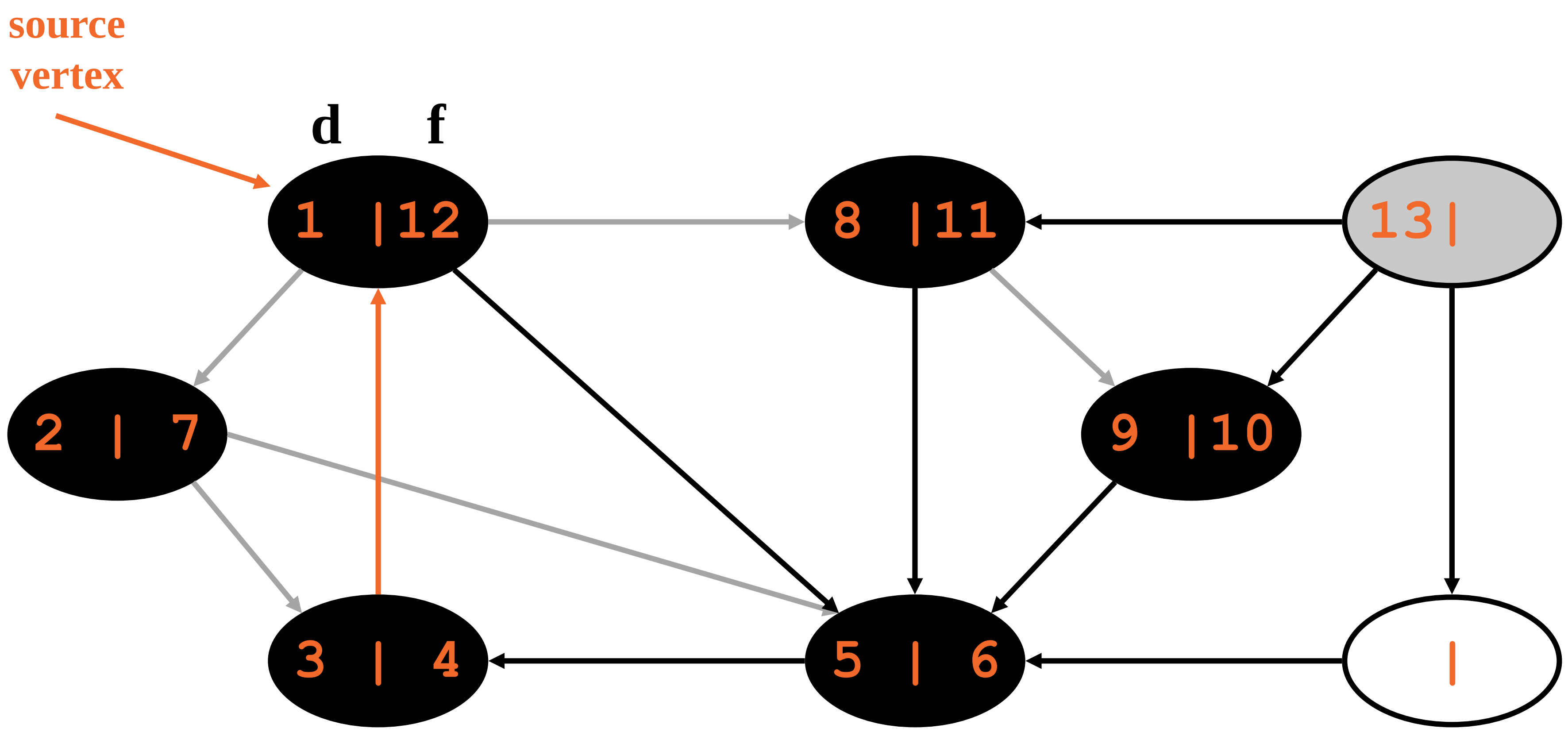
DFS Example



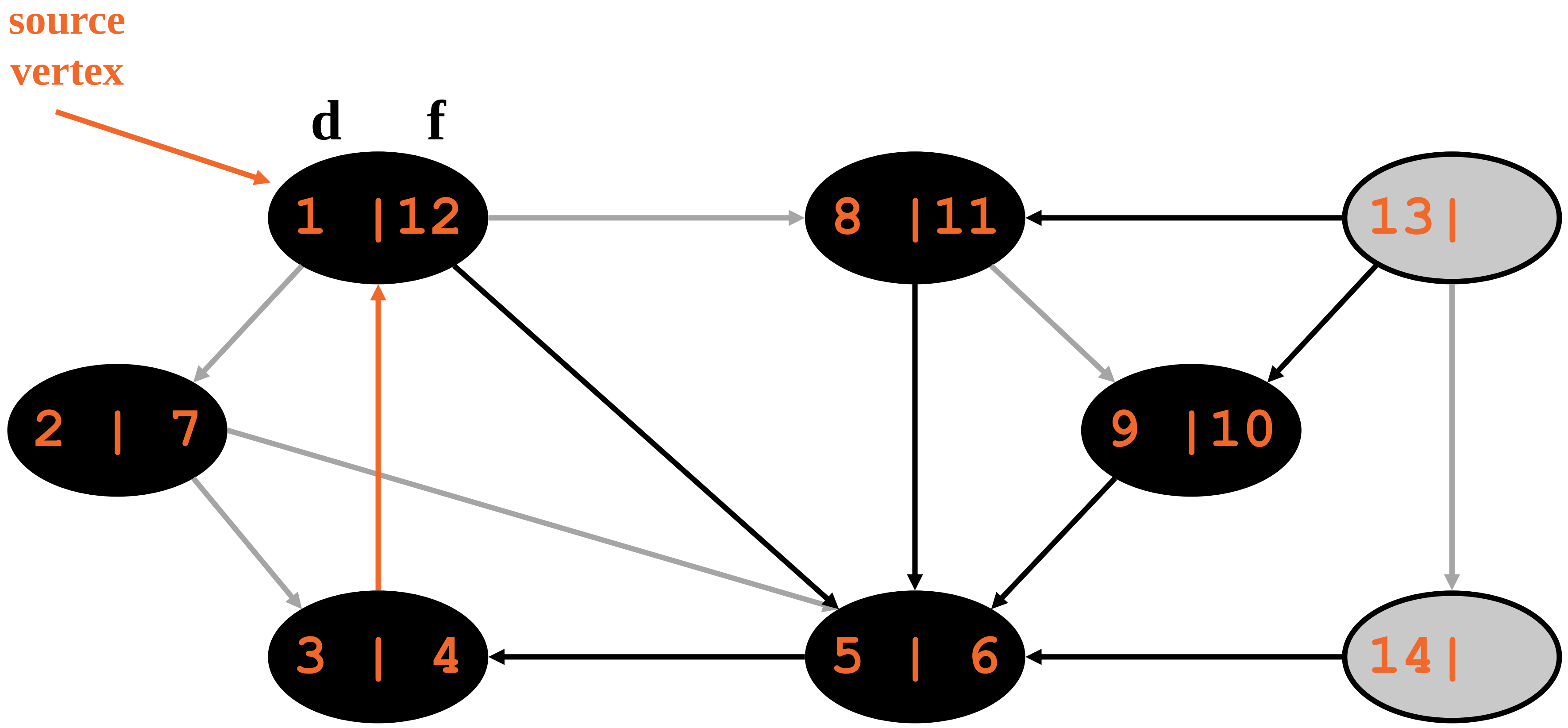
DFS Example



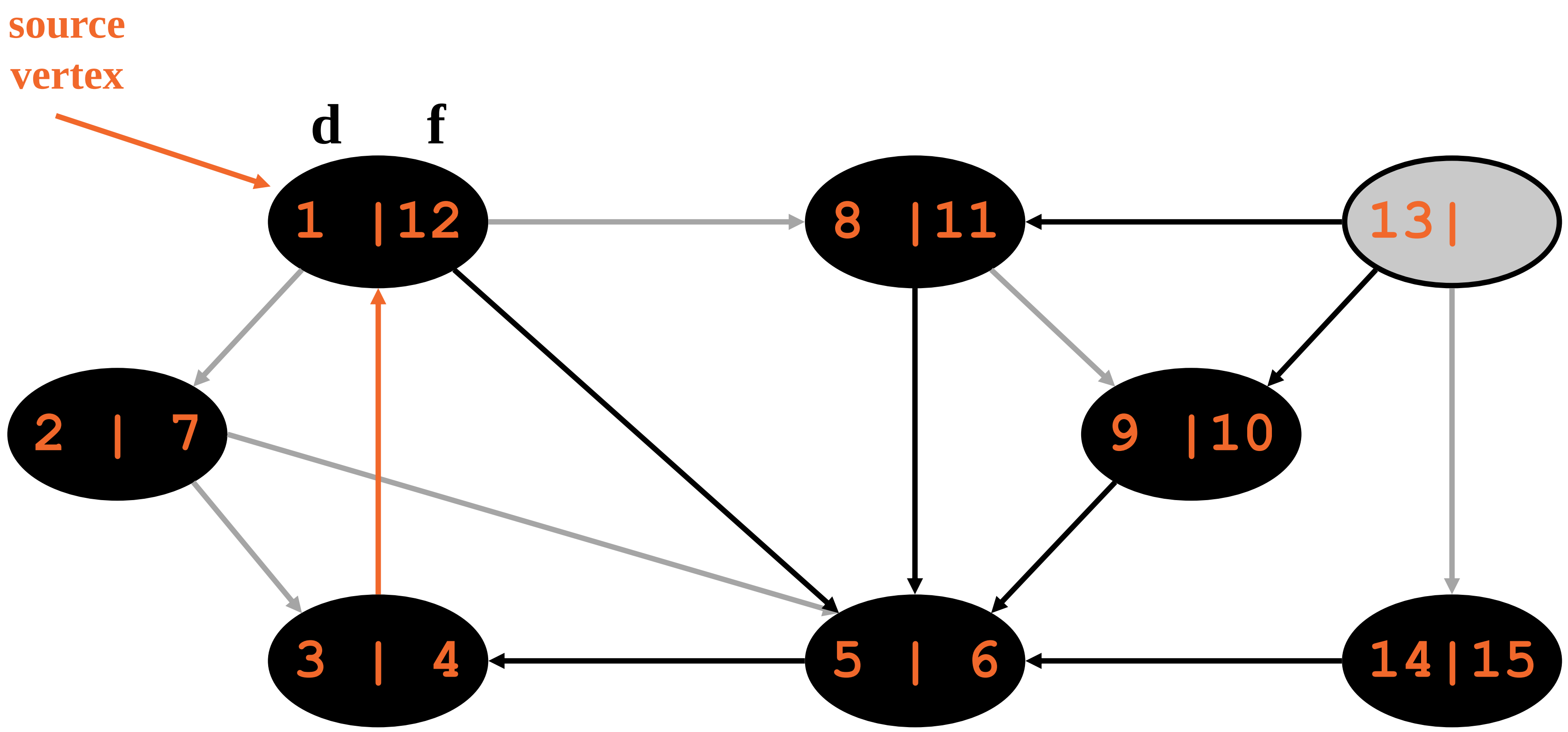
DFS Example



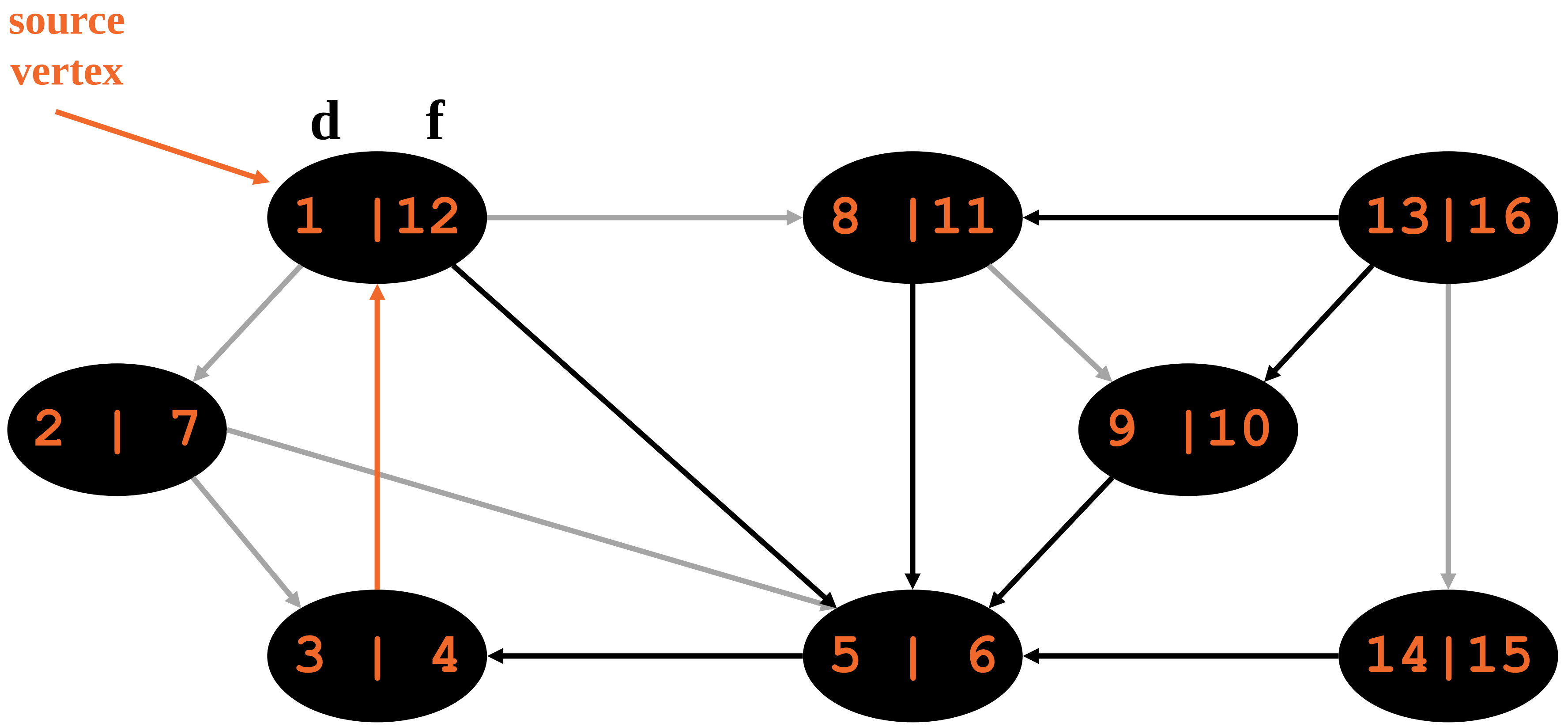
DFS Example



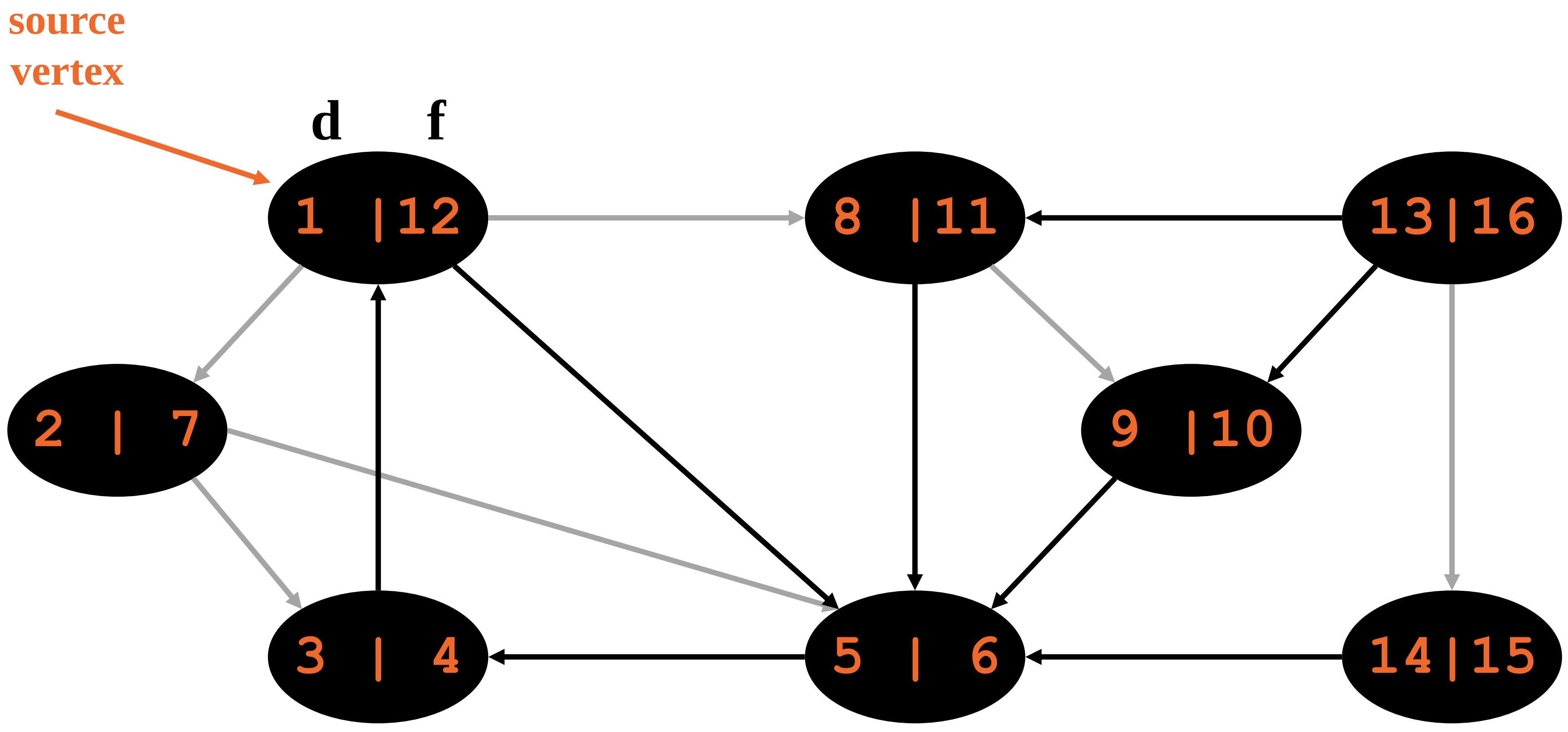
DFS Example



DFS Example

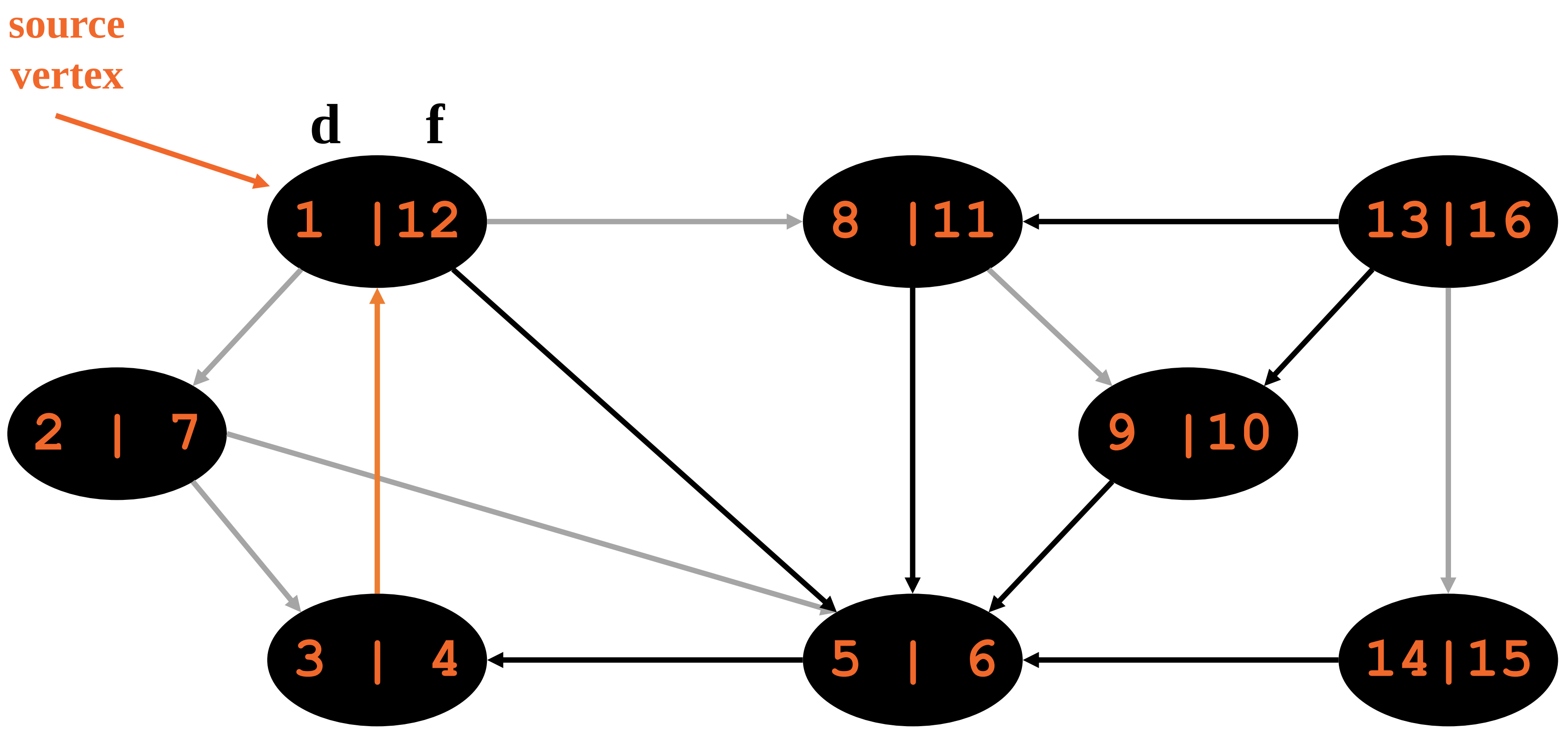


DFS Example



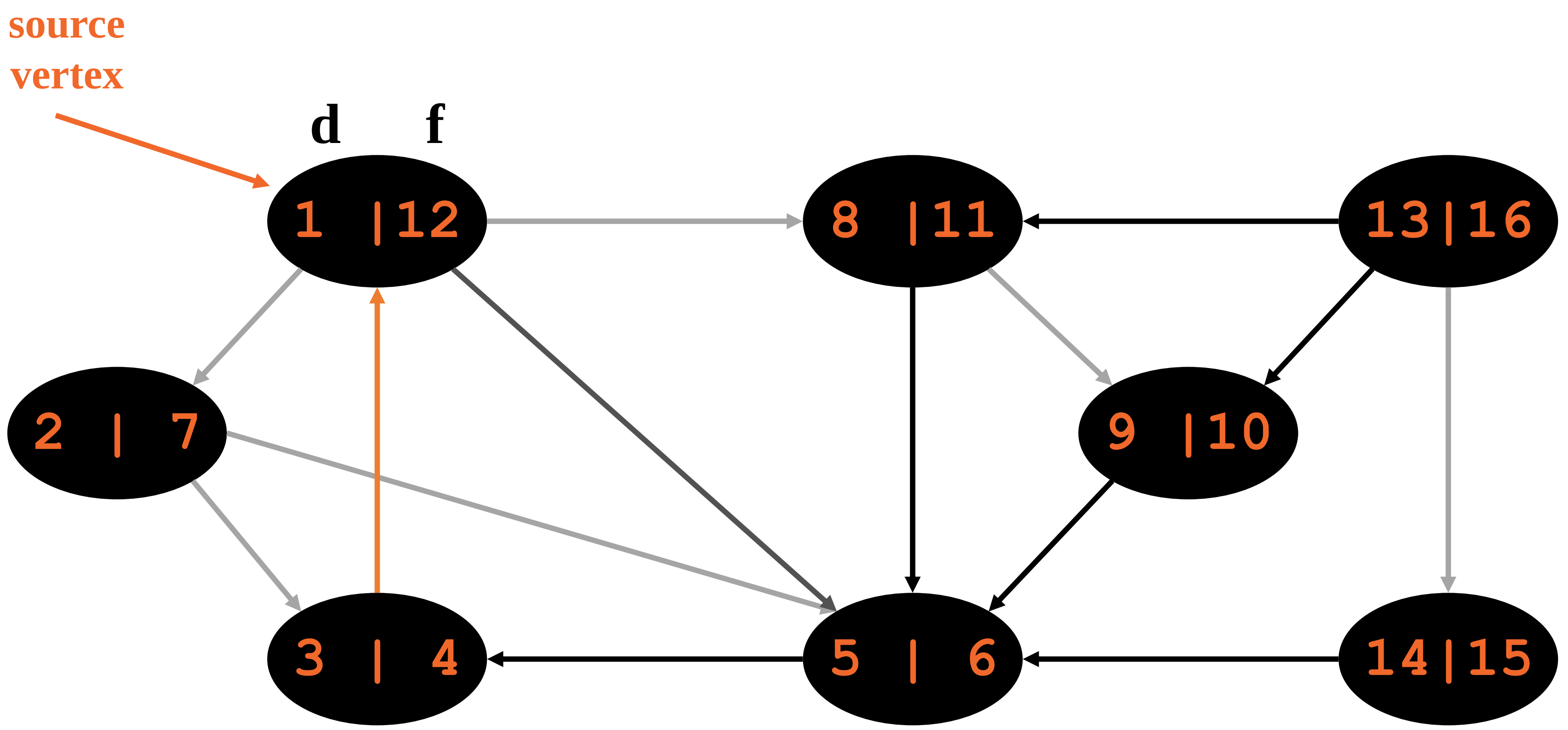
Tree edges

DFS Example



Tree edges Back edges

DFS Example

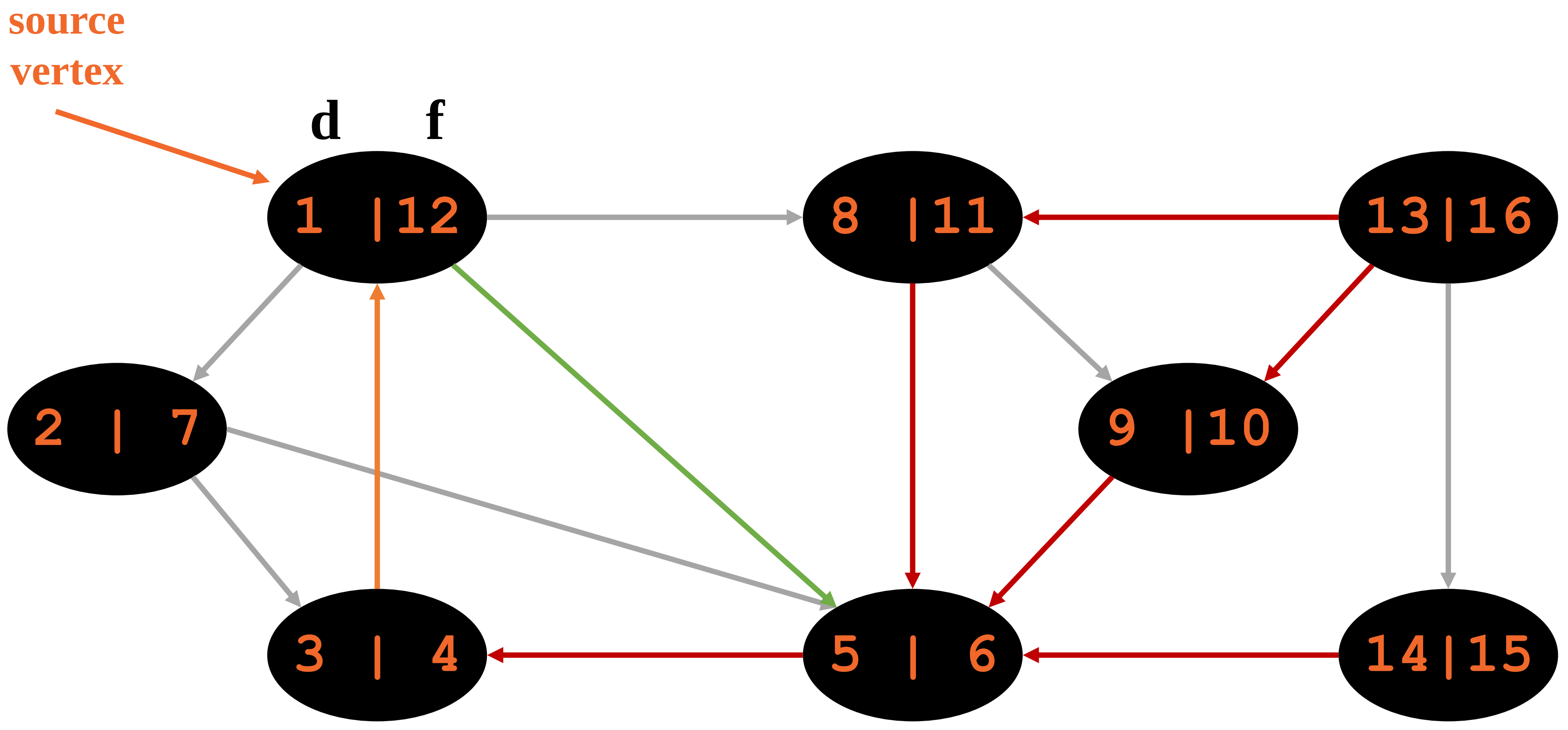


Tree edges Back edges Forward edges

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
- Note: tree & back edges are important; most algorithms don’t distinguish forward & cross

DFS Example



Tree edges Back edges Forward edges Cross edges

DFS And Graph Cycles

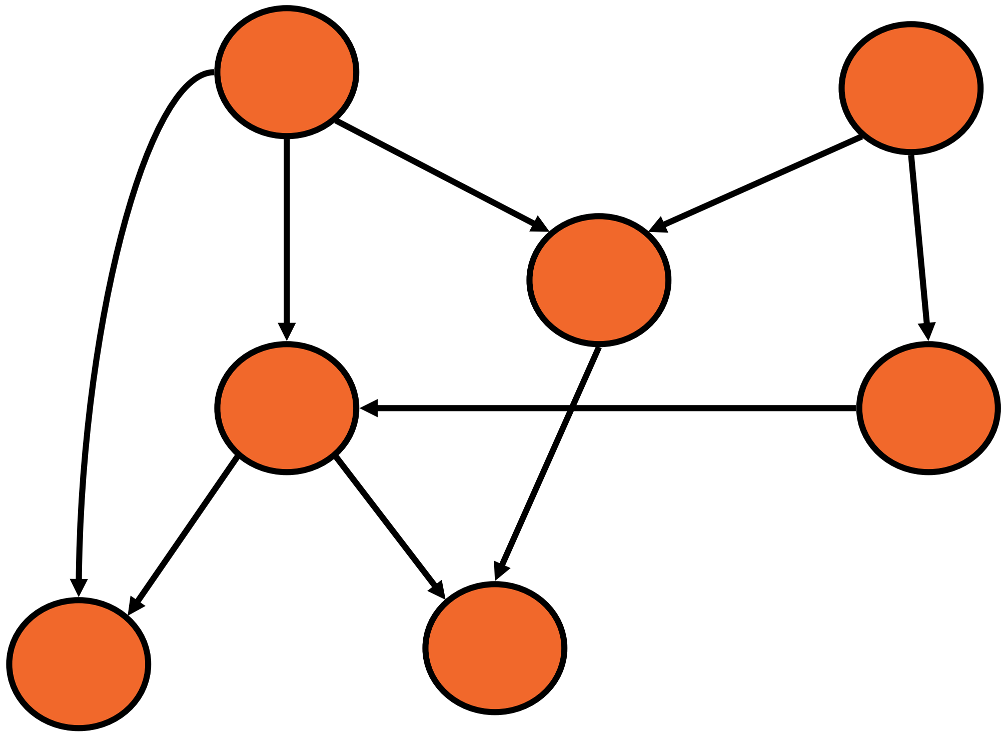
- Thm: An undirected graph is *acyclic* iff a DFS yields no back edges
  - If acyclic, no back edges (because a back edge implies a cycle
  - If no back edges, acyclic
    - No back edges implies only tree edges Only tree edges implies we have a tree or a forest
    - Which by definition is acyclic
- Thus, can run DFS to find whether a graph has a cycle

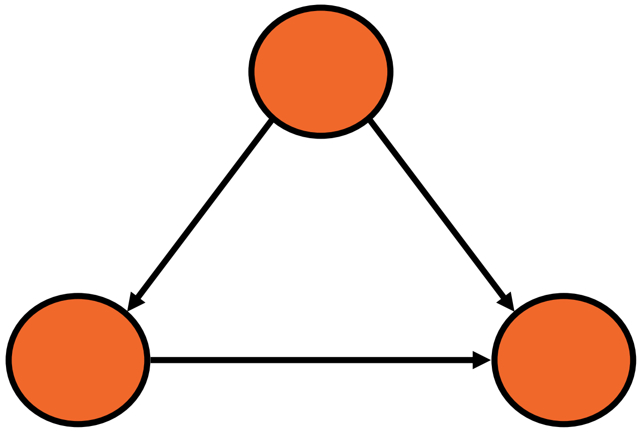
DFS And Cycles

- $\Theta(V+E)$
- We can actually determine if cycles exist in $\Theta(V)$ time:
  - In an undirected acyclic forest, $|E| \leq |V| - 1$
  - So count the edges: if ever see $|V|$ distinct edges, must have seen a back edge along the way

Directed Acyclic Graphs

- A *directed acyclic graph* or *DAG* is a directed graph with no directed cycles:
- directed graph G is acyclic iff a DFS of G yields no back edges:



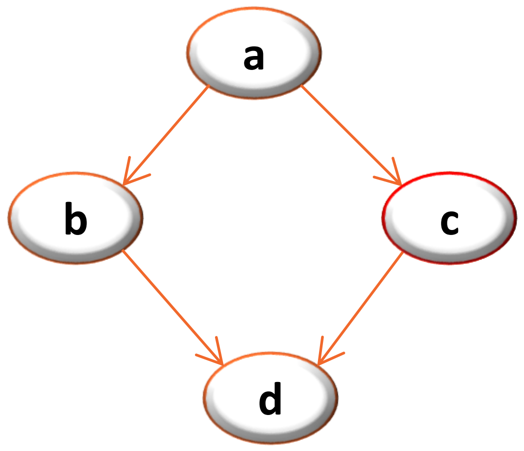


Topological Sort

- *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:

    Topological sort is a, b, c, d

    or   a, c, b, d



Topological Sort Algorithm

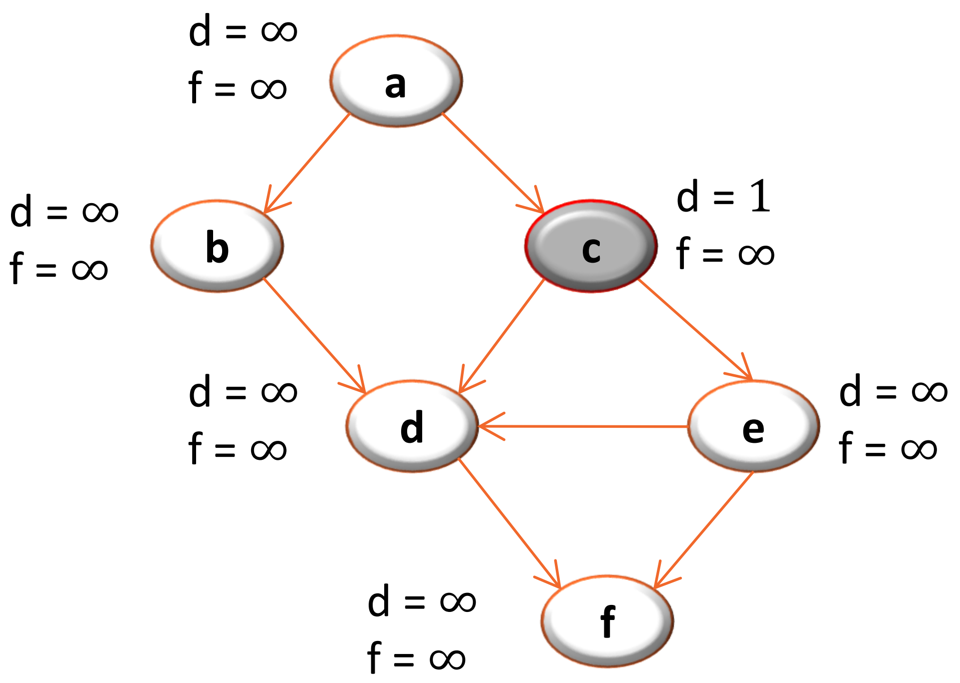
```
Topological-Sort()
{
    Run DFS
    When a vertex is finished, output it
    Vertices are output in reverse topological
      order
}
```

- Time: $\Theta(V+E)$

Topological Example

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 2



Let's say we start the DFS from the vertex **c**

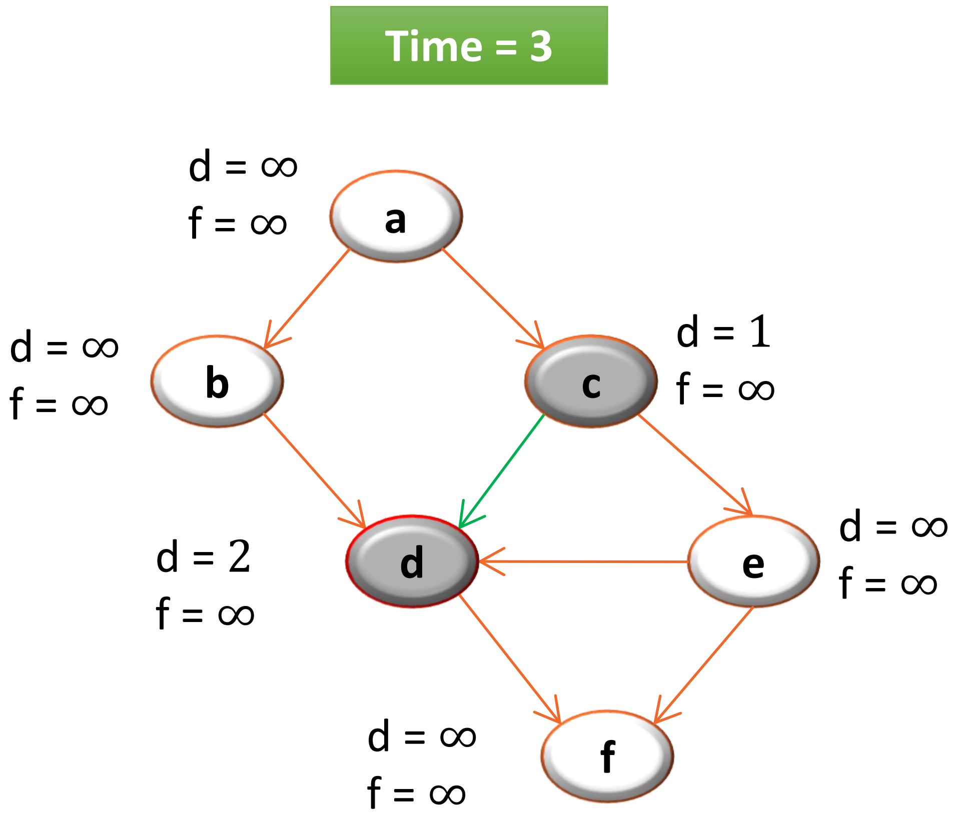
Next we discover the vertex **d**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**



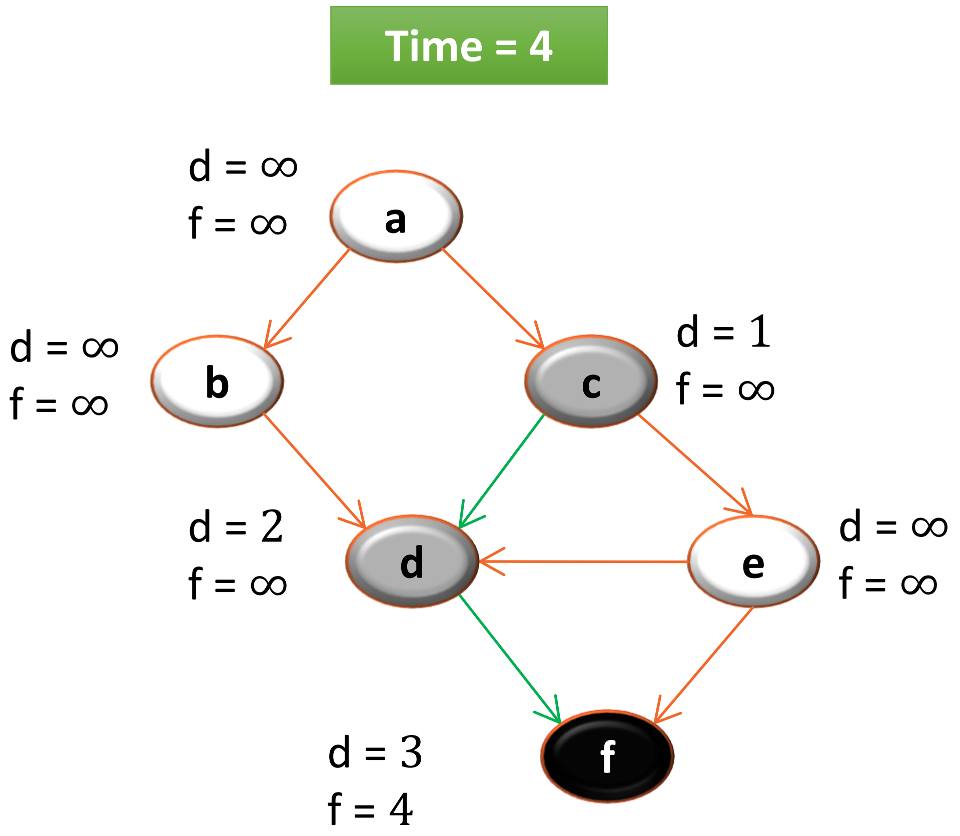
Topological sort

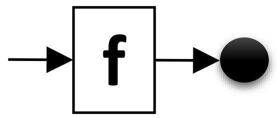
1) Call DFS(**G**) to compute the finishing times **f**[**v**]

2) as each vertex is finished, insert it onto the **front** of a linked list

Next we discover the vertex **f**

f is done, move back to **d**

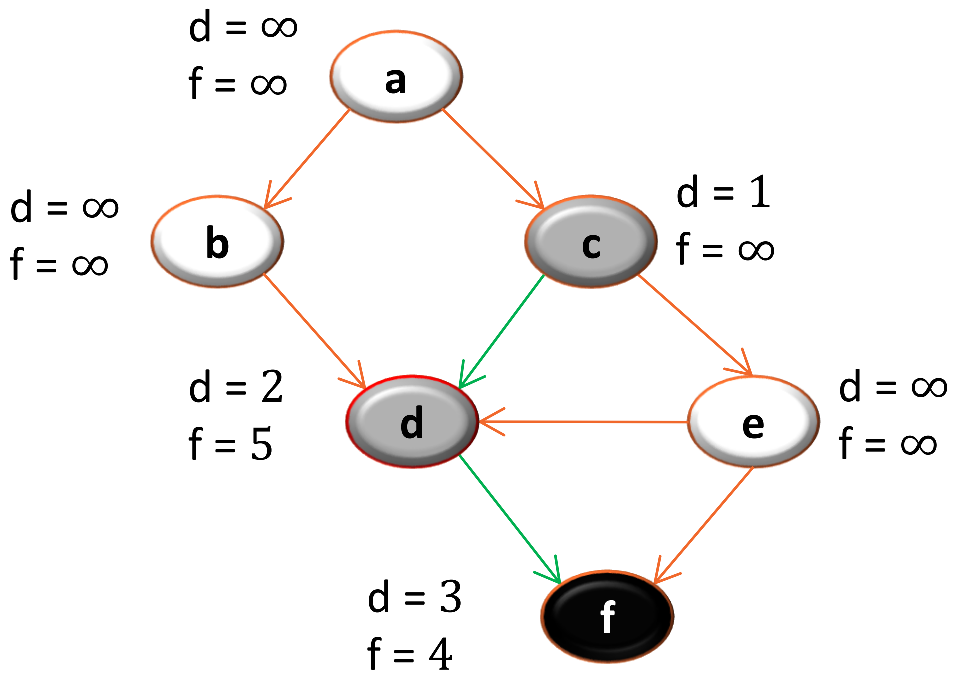


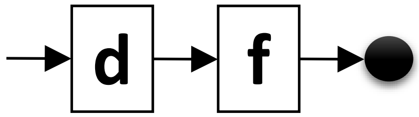


Topological sort

1) Call DFS(**G**) to compute the finishing times **f[v]**

Time = 5





Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Next we discover the vertex **f**

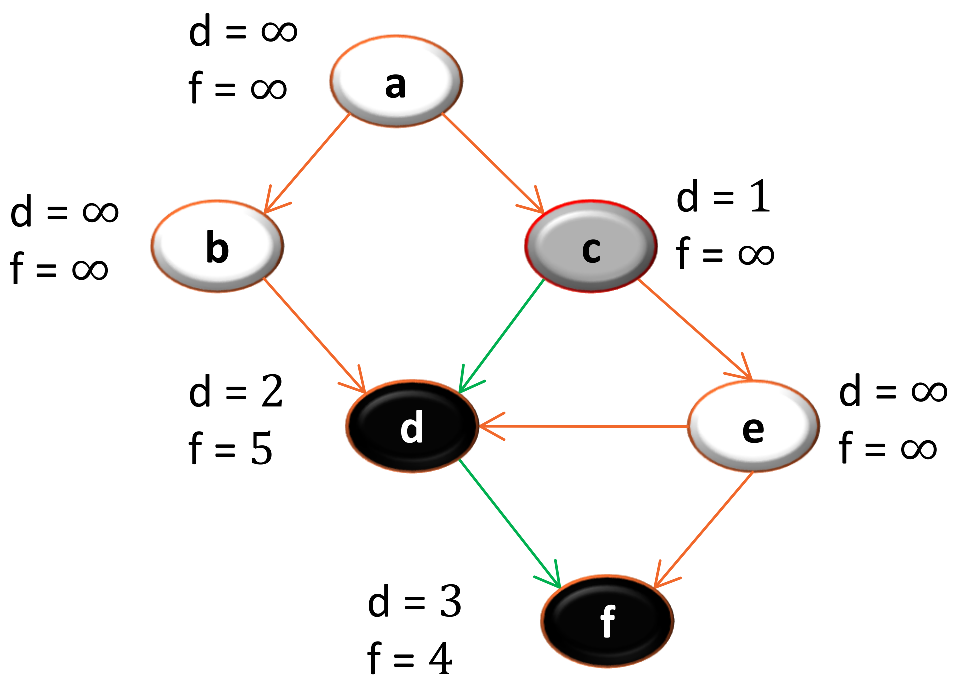
f is done, move back to **d**

d is done, move back to **c**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 6



Let's say we start the DFS from the vertex **c**

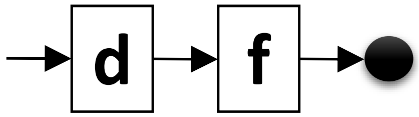
Next we discover the vertex **d**

Next we discover the vertex **f**

f is done, move back to **d**

d is done, move back to **c**

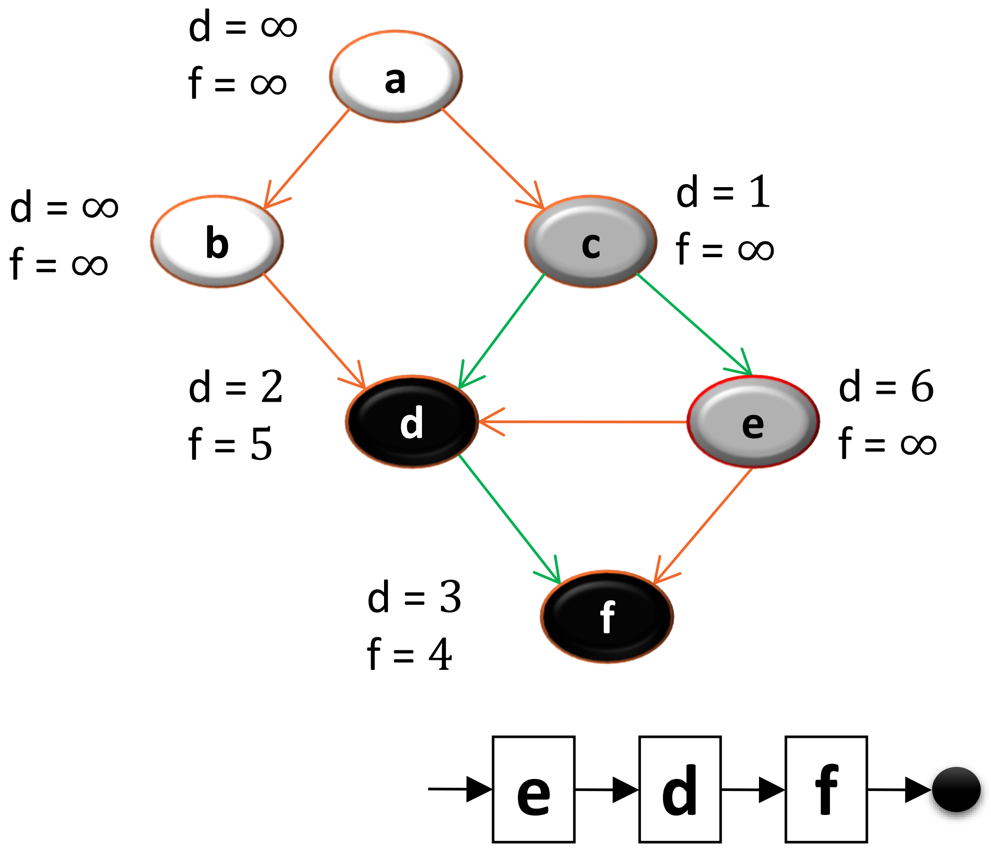
Next we discover the vertex **e**



Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 7



Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Both edges from **e** are **cross edges**

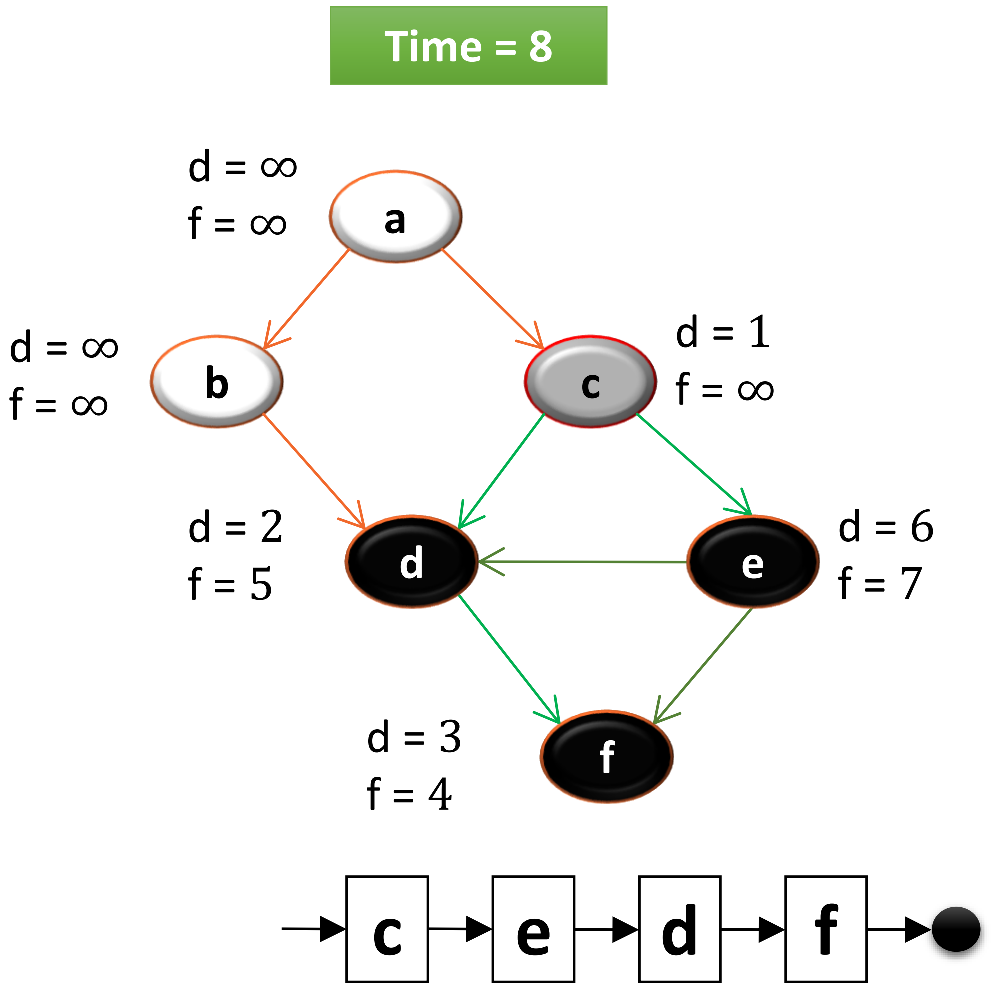
d is done, move back to **c**

Next we discover the vertex **e**

e is done, move back to **c**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f[v]**



Let's say we start the DFS from the vertex **c**

Just a note: If there was (**c**,**f**) edge in the graph, it would be classified as a **forward edge** (in this particular DFS run)

d is done, move back to **c**

Next we discover the vertex **e**

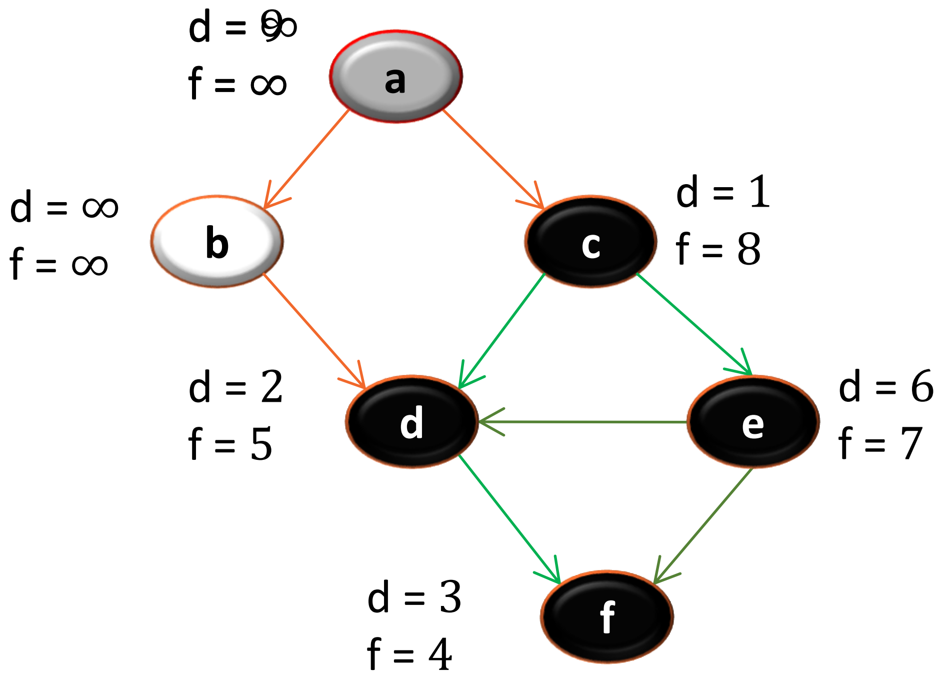
e is done, move back to **c**

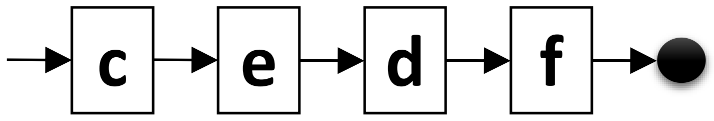
c is done as well

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 10





Let’s now call DFS visit from the vertex **a**

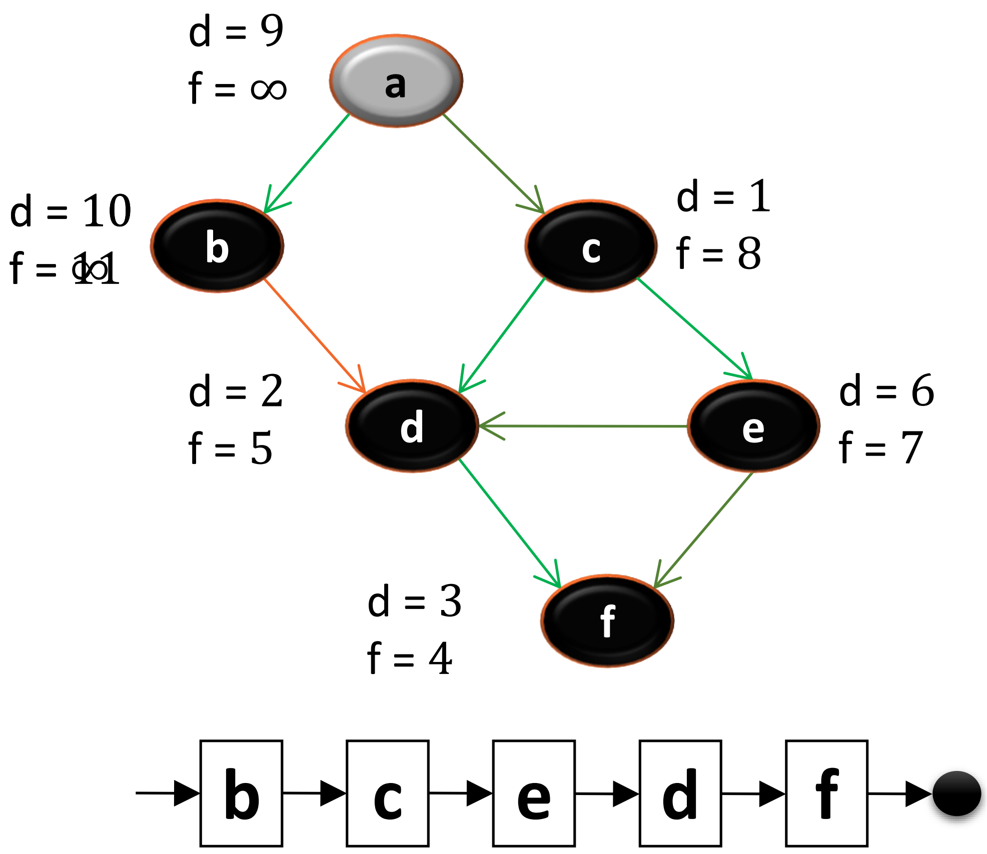
Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 11



Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

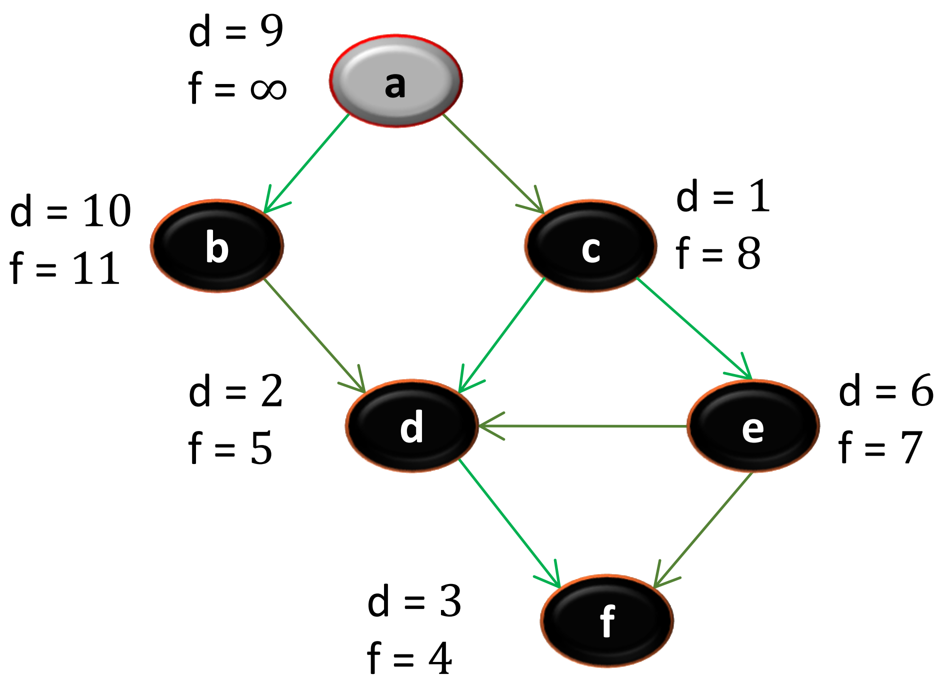
Next we discover the vertex **b**

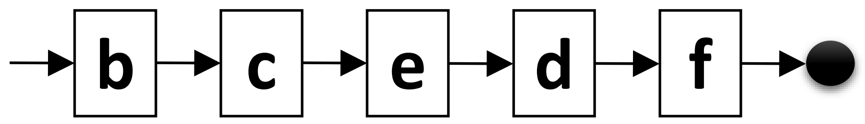
b is done as (**b**,**d**) is a cross edge => now move back to **c**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 12





Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well

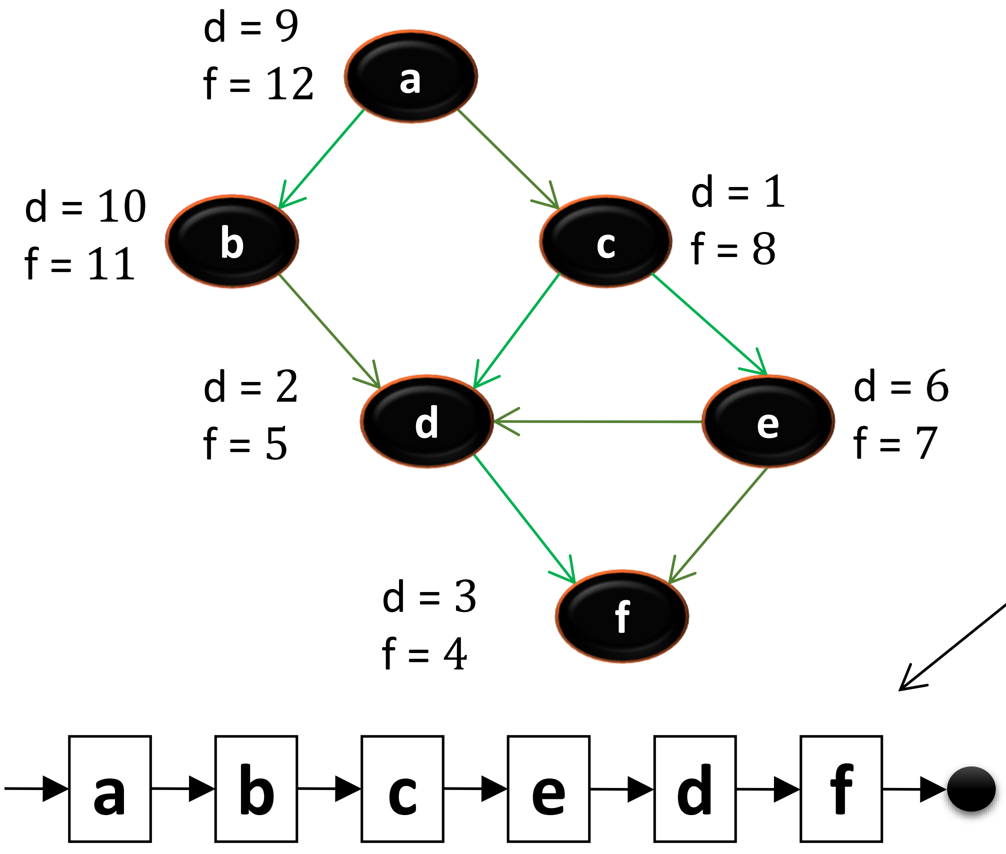
Topological sort

1) Call DFS(**G**) to compute the finishing times **f[v]**

WE HAVE THE RESULT!

3) return the linked list of vertices

Time = 13



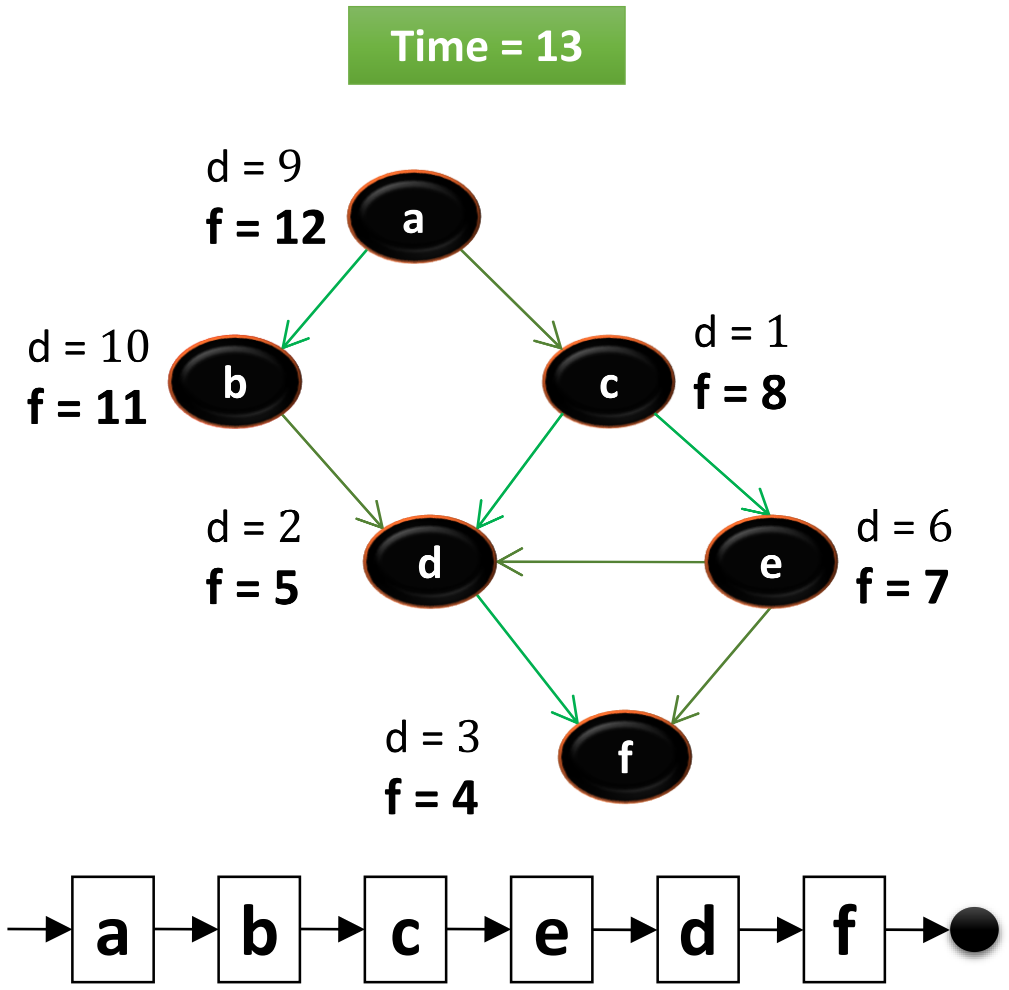
(**a**,**c**) is a cross edge

Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well

Topological sort



The linked list is sorted in **decreasing** order of finishing times **f**[]

Try yourself with different vertex order for DFS visit

Note: If you redraw the graph so that all vertices are in a line ordered by a valid topological sort, then all edges point „**from left to right**“

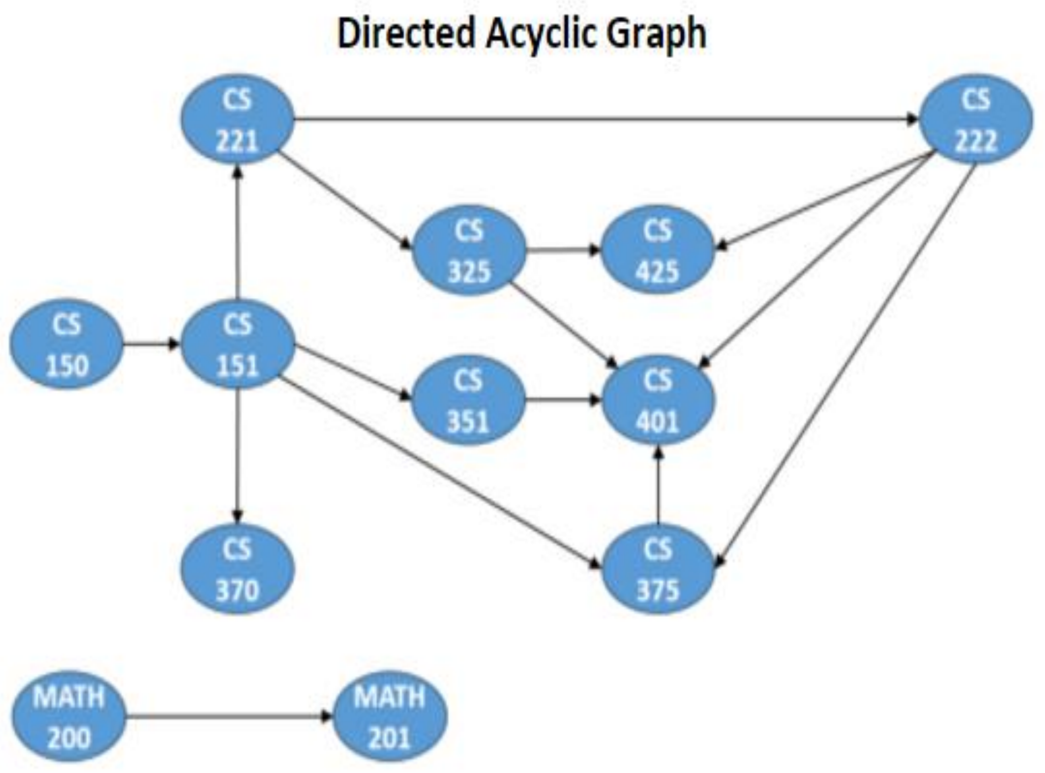
College Courses

Below is a list of courses and prerequisites for a factious CS degree. A DAG can represent the precedence among the courses. A topological sort gives an order to take classes.

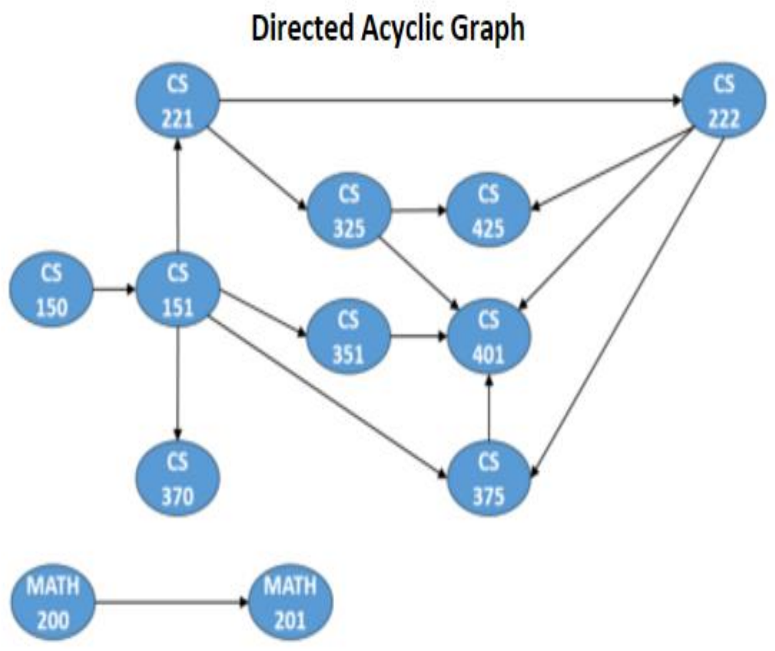
Course	Prerequisite
CS 150	None
CS 151	CS 150
CS 221	CS 151
CS 222	CS 221
CS 325	CS 221
CS 351	CS 151
CS 370	CS 151
CS 375	CS 151, CS 222
CS 401	CS 375, CS 351, CS 325, CS 222
CS 425	CS 325, CS 222
MATH 200	None
MATH 201	MATH 200

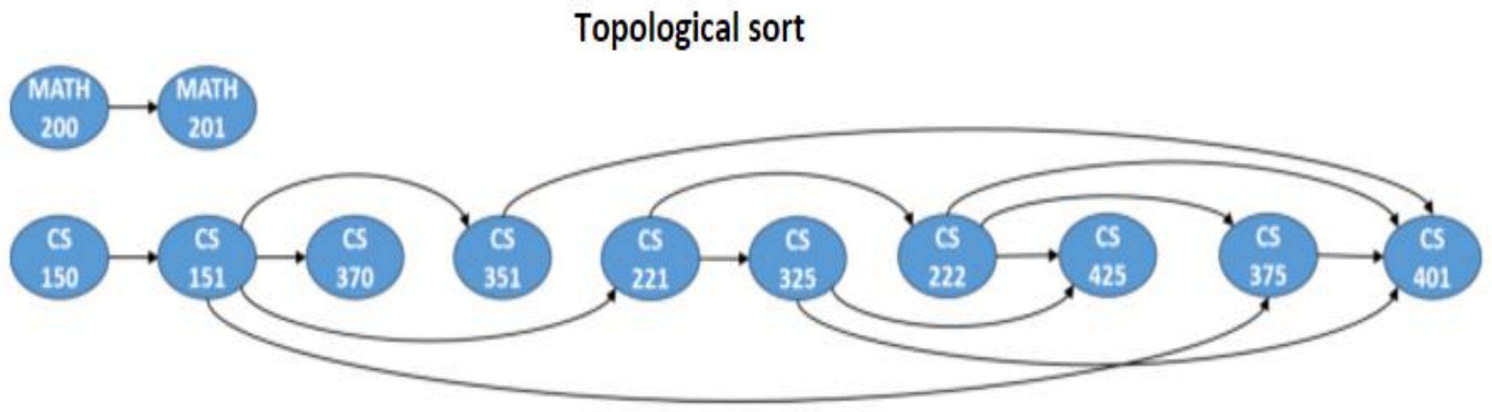
College Courses - DAG

Course	Prerequisite
CS 150	None
CS 151	CS 150
CS 221	CS 151
CS 222	CS 221
CS 325	CS 221
CS 351	CS 151
CS 370	CS 151
CS 375	CS 151, CS 222
CS 401	CS 375, CS 351, CS 325, CS 222
CS 425	CS 325, CS 222
MATH 200	None
MATH 201	MATH 200

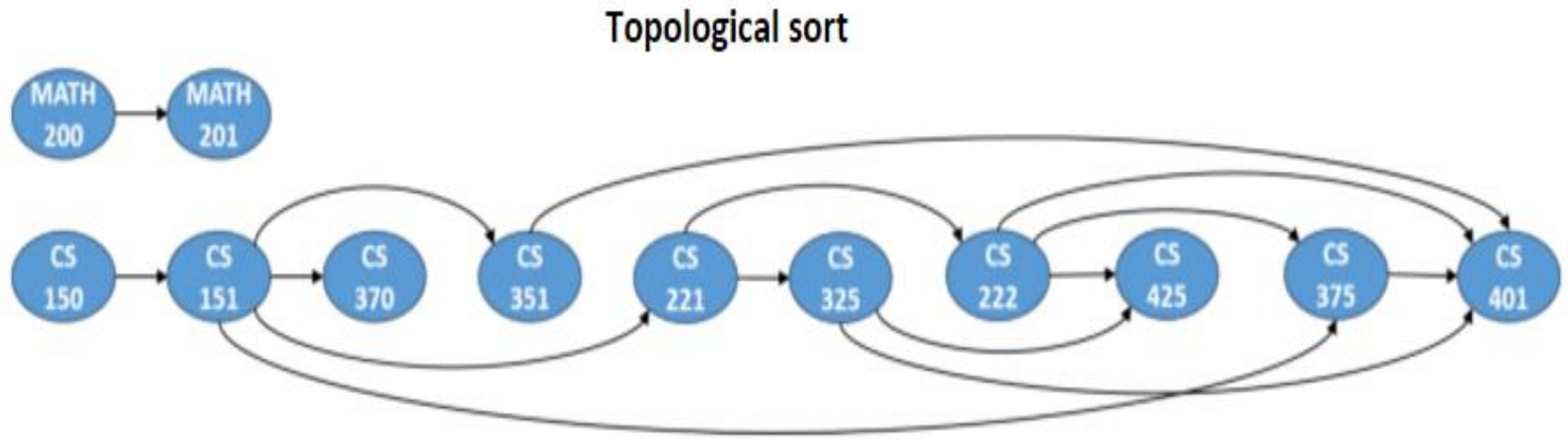


College Courses – Topological Ordering

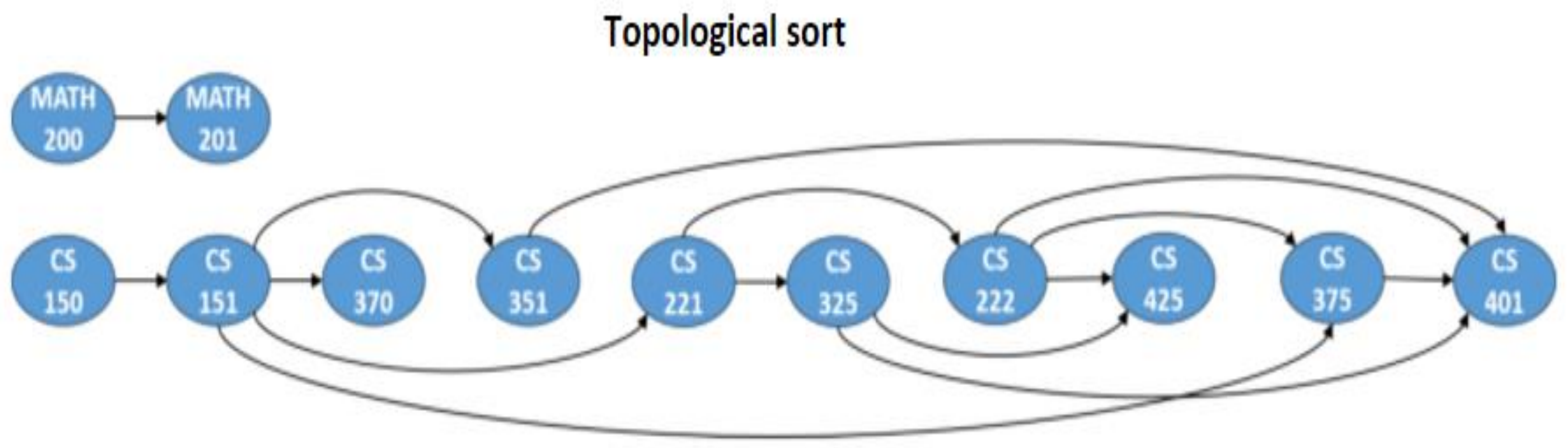




College Courses – Longest Path



College Courses – Longest Path



The length of the longest path is 5 which corresponds to the courses

CS150, CS151, CS 221, CS222, CS375, CS401.

To find the Longest Path in a DAG

1) Topologically sort the graph

2) Initialize the distances associated with vertices in the graph to 0. That is d(v)=0 for all v in G.

3) Start with the first vertex v in the topological sort.

For each u in Adj[v] set d(u) = max { d(u), d(v)+1 }

4) Repeat step 4 with the next vertex in the topological sorted order until all vertices have been examined.

O(E+V)

Hamiltonian Path in a DAG

A Hamiltonian path in a graph G=(V,E) is a simple path that includes every vertex in V. Design an algorithm to determine if a directed acyclic graph (DAG) G has a Hamiltonian path.

Example

Hamiltonian Path in a DAG

Idea

Compute a topological sort and check if there is an edge between each consecutive pair of vertices in the topological order. If each consecutive pair of vertices are connected, then every vertex in the DAG is connected, which indicates a Hamiltonian path exists. Running time for the topological sort is O(V+E), running time for the next step is O(V) so total running time is O(V+E).

Hamiltonian Path in a DAG

PSEUDOCODE:

```
HAS_HAM_PATH(G)
    1. Call DFS(G) to computer finishing time v.f for each vertex v.
    2. As each vertex is finished, insert it into the front of the list
    3. Iterate each through the lists of vertices in the list
    4. If any pairs of consecutive vertices are not connected RETURN FALSE
    5. After all pairs of vertices are examined RETURN TRUE
```