Graph Algorithms

Graph

- A graph $G = (V, E)$
  - V = set of vertices
  - E = set of edges = subset of $V \times V$
  - Thus $|E| = O(|V|^2)$

Graph

- Variations:
  - A *connected graph* has a path from every vertex to each 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 $u \rightarrow v$
    - Self loops are allowed.

Graph

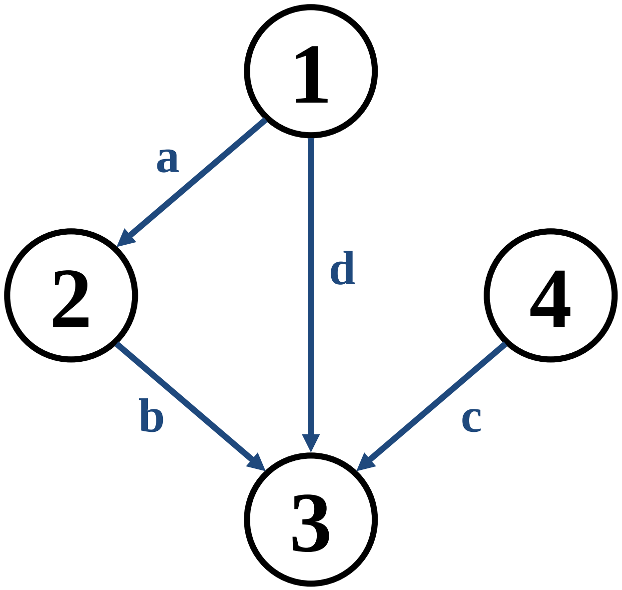
- A *weighted graph* associates weights with either the edges or the vertices
  - E.g., a road map: edges might be weighted w/ distance
- A *multigraph* allows multiple edges between the same vertices
  - E.g., the call graph in a program (a function can get called from multiple points in another function)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

Representing Graphs

- 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 $(i, j) \notin E$

Adjacency Matrix

- **Space:** $\Theta(V^2)$.
  - Not memory efficient for large graphs.
- **Time:** to list all vertices adjacent to u: $\Theta(V)$.
- **Time:** to determine if $(u, v) \in E$: $\Theta(1)$.



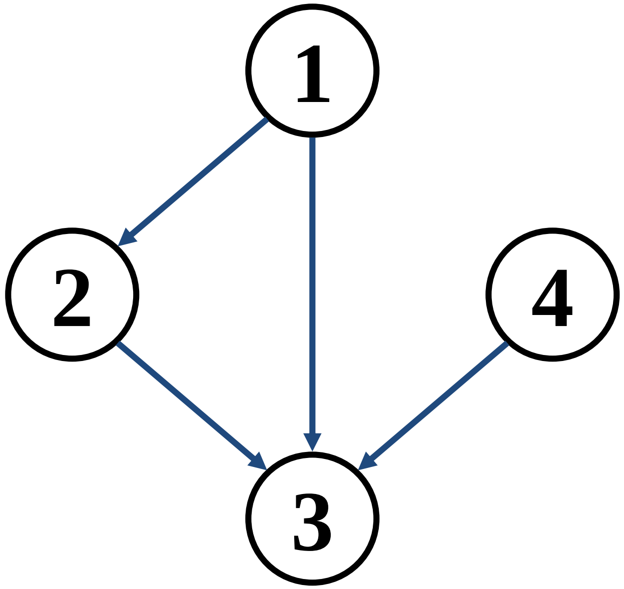
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

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
  - For this reason the *adjacency list* is often a more appropriate representation

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



Adjacency list

- For directed graphs:
  - Sum of lengths of all adj. lists is

    $$\sum_{v \in V} \text{out-degree}(v) = |E|$$

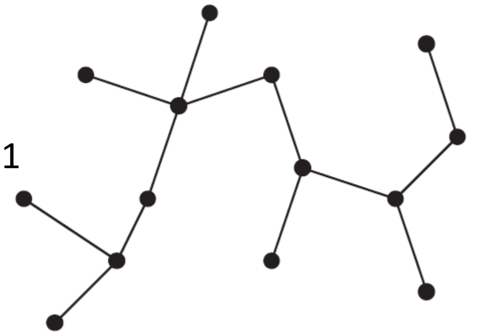
    No. of edges leaving v
  - Total storage: $\Theta(V+E)$
- For undirected graphs:
  - Sum of lengths of all adj. lists is

    $$\sum_{v \in V} \text{degree}(v) = 2|E|$$

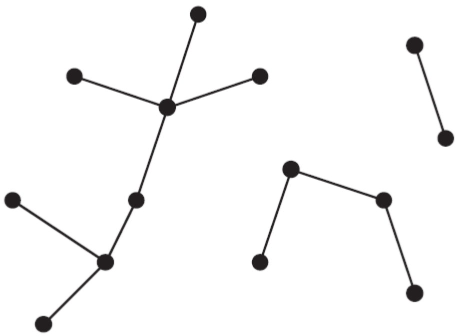
    No. of edges incident on v. Edge (u,v) is incident on vertices u and v.
  - Total storage: $\Theta(V+E)$

Graph Definitions

- Path
  - Sequence of nodes $n_1, n_2, \ldots n_k$
  - Edge exists between each pair of nodes n_i , n_{i+1}
- Cycle
  - Path that ends back at starting node
- Acyclic graph
  - No cycles in graph
- Tree
  - Undirected, acyclic and connected graph.
- Forest
  - Undirected, acyclic but possibly disconnected graph.



A tree



A forest

Breadth-First Search

- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find (“discover”) its children, then their children, etc.
- **Input:** Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.
- **Output:**
  - $d[v] =$ distance (smallest # of edges, or shortest path) from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  - $\pi[v] = u$ such that (u, v) is last edge on shortest path $s \rightarrow v$.
    - u is v’s predecessor.
  - Builds breadth-first tree with root s that contains all reachable vertices.

BFS(G,s)

```
for each vertex u in V[G] – {s}
      color[u] ← white
      d[u] ← ∝
      π[u] ← nil
color[s] ← gray
d[s] ← 0
π[s] ← nil
Q ← Φ
enqueue(Q,s)
while Q ≠ Φ
      u ← dequeue(Q)
      for each v in Adj[u]
            if color[v] = white
                  color[v] ← gray
                  d[v] ← d[u] + 1
                  π[v] ← u
                  enqueue(Q,v)
      color[u] ← black
```

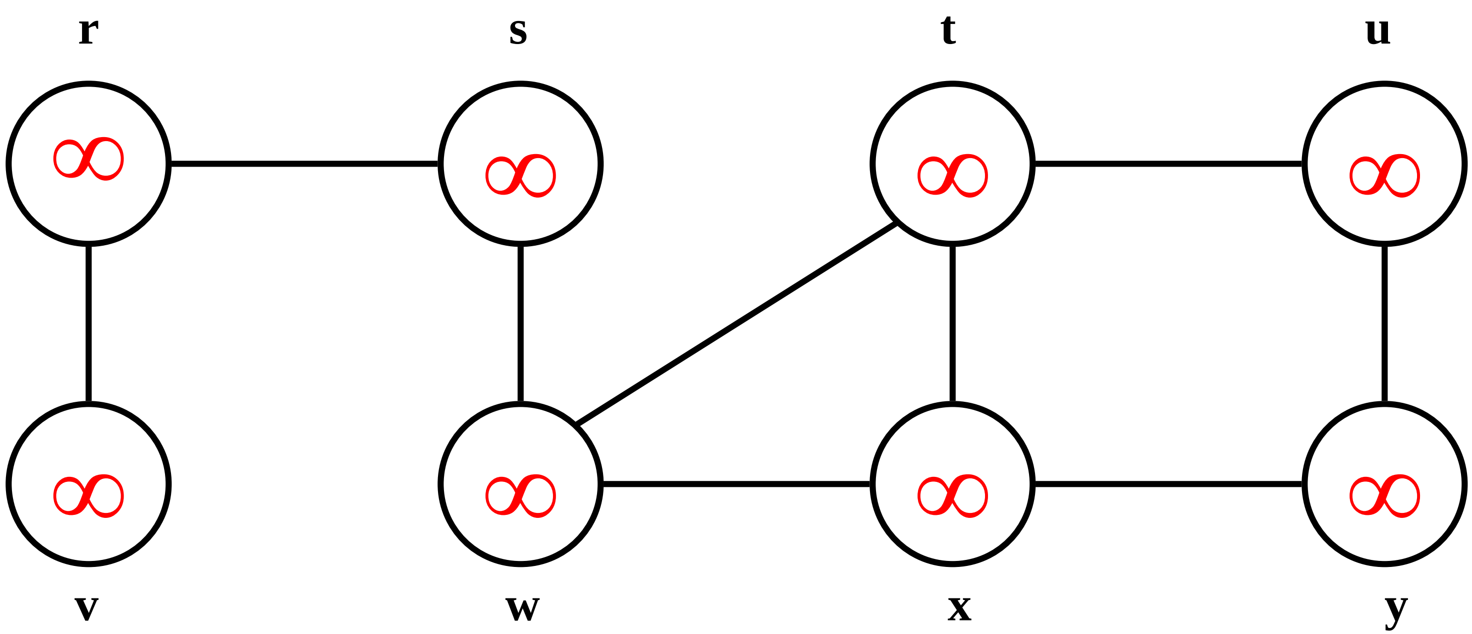
initialization

access source *s*

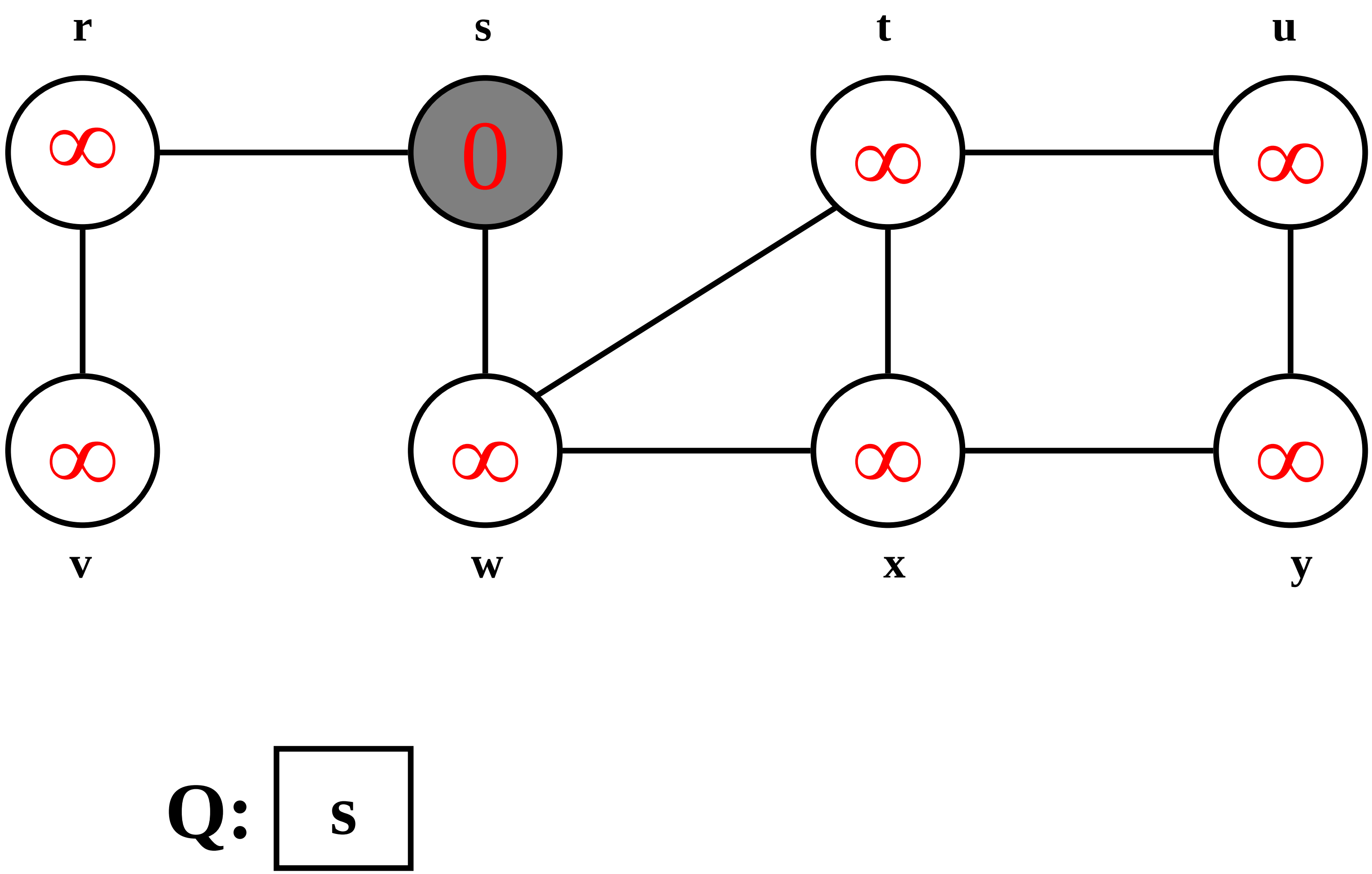
white: undiscovered
gray: discovered
black: finished

Q: a queue of discovered vertices
color[*v*]: color of v
d[*v*]: distance from s to v
π[*u*]: predecessor of v

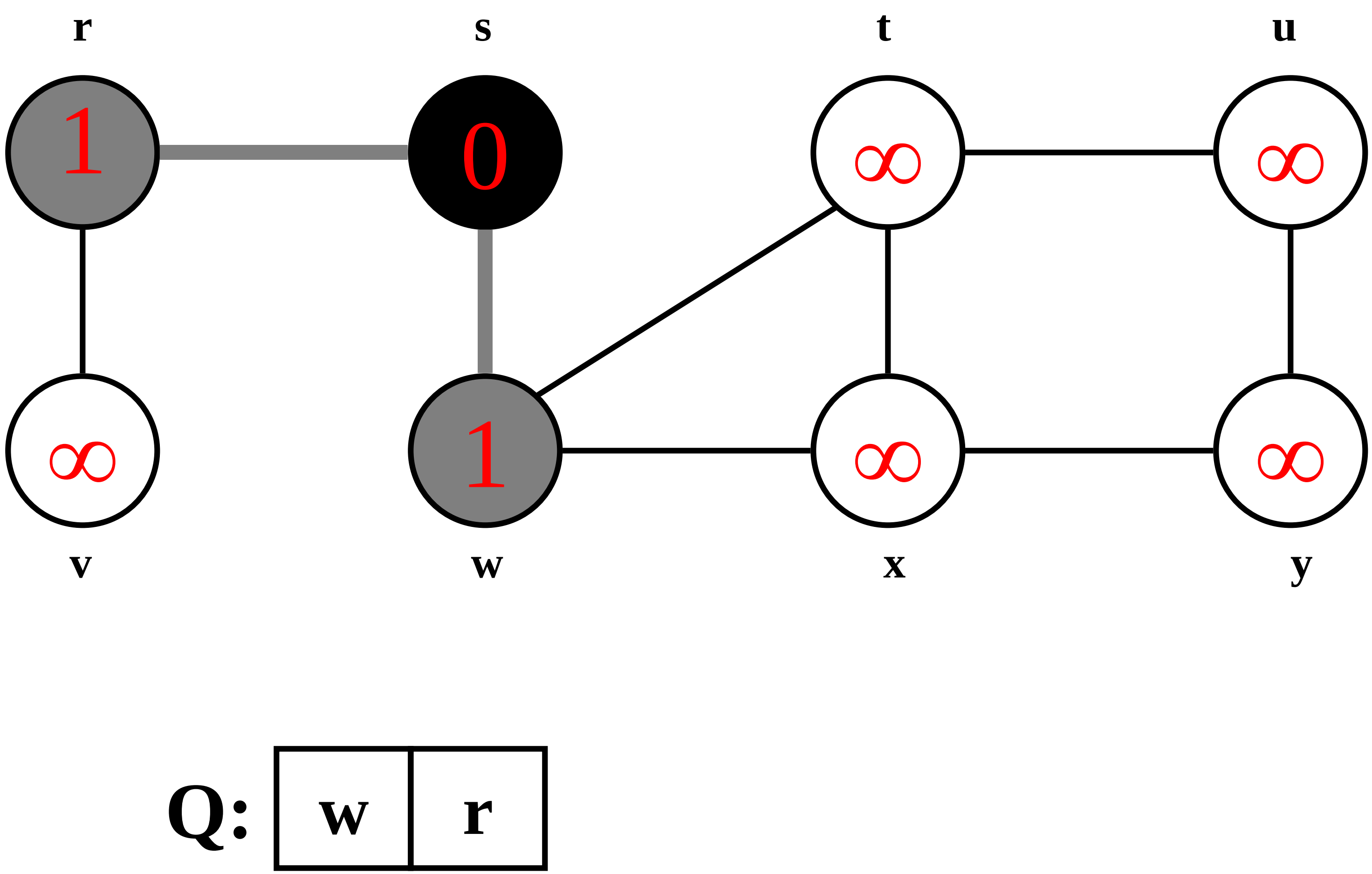
Breadth-First Search: Example



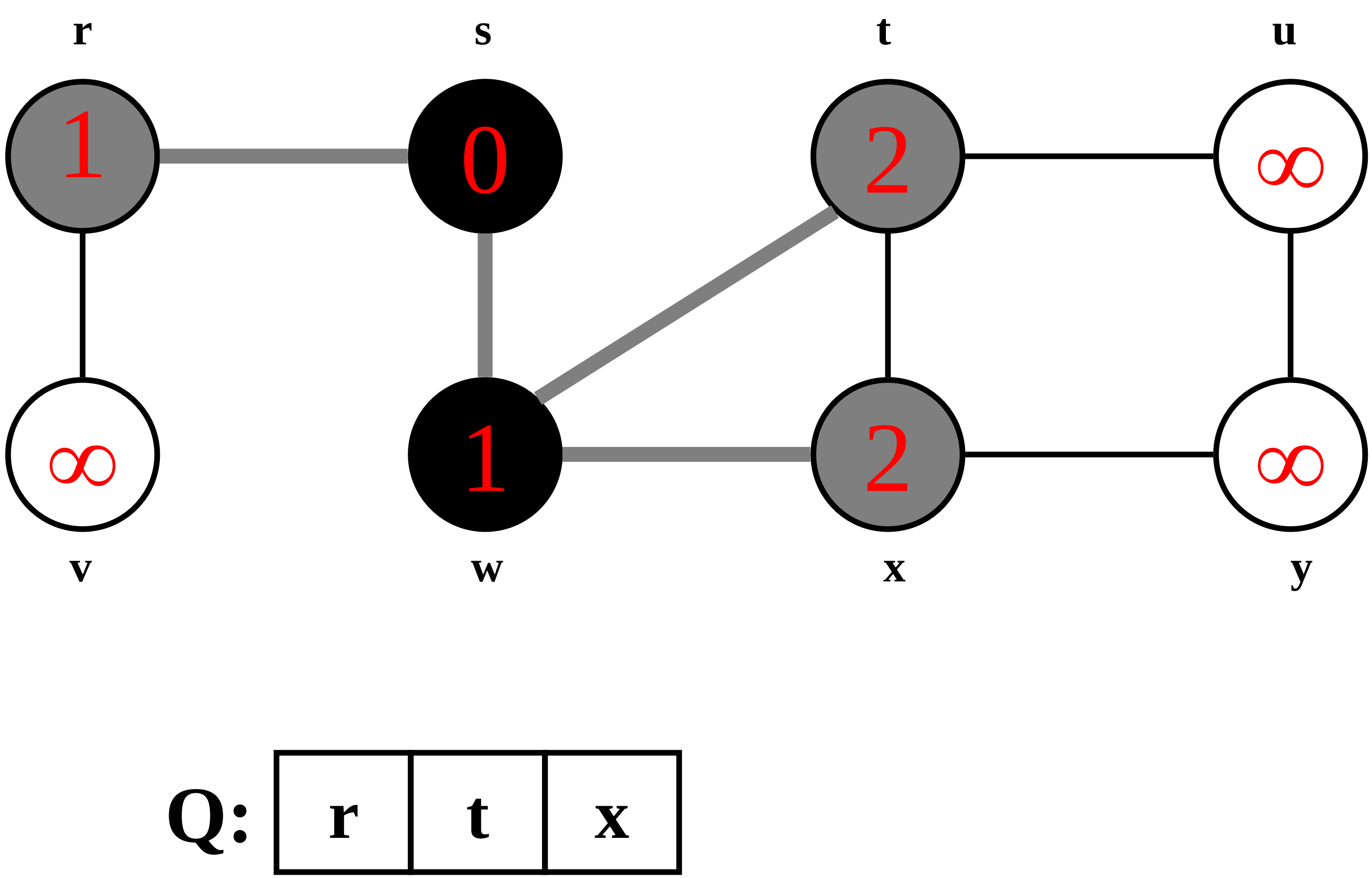
Breadth-First Search: Example



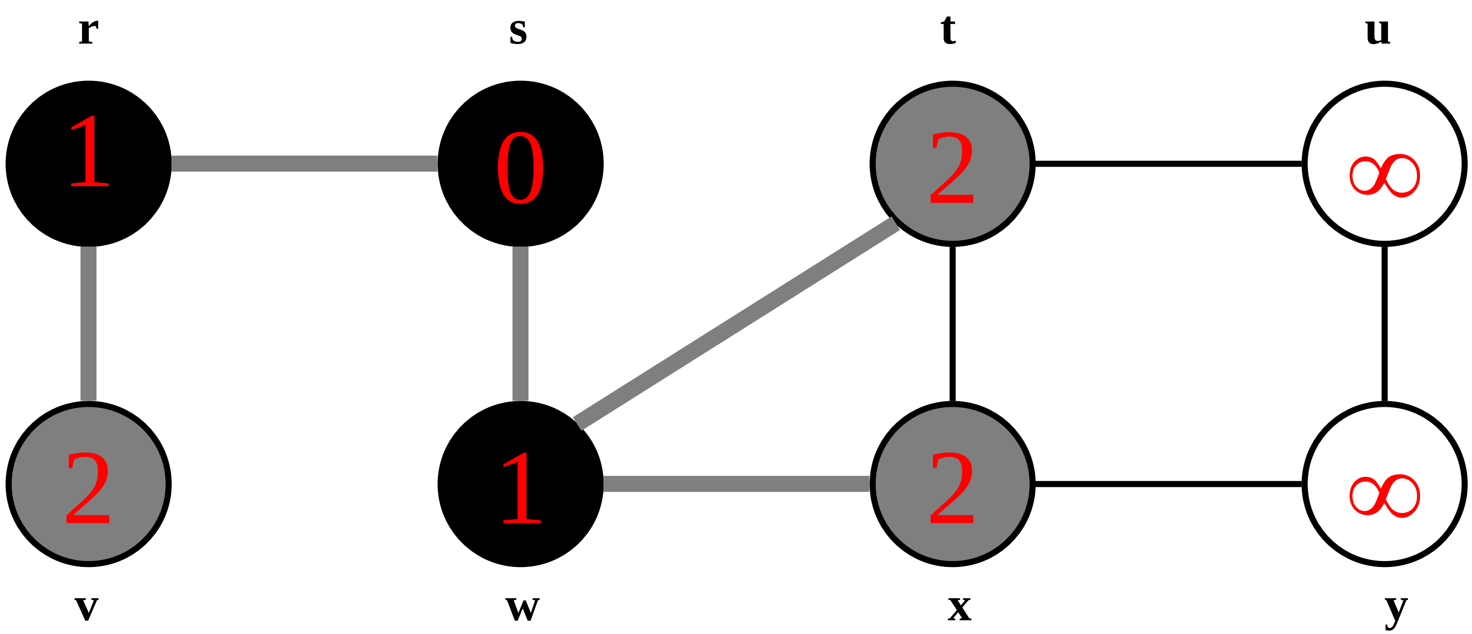
Breadth-First Search: Example



Breadth-First Search: Example

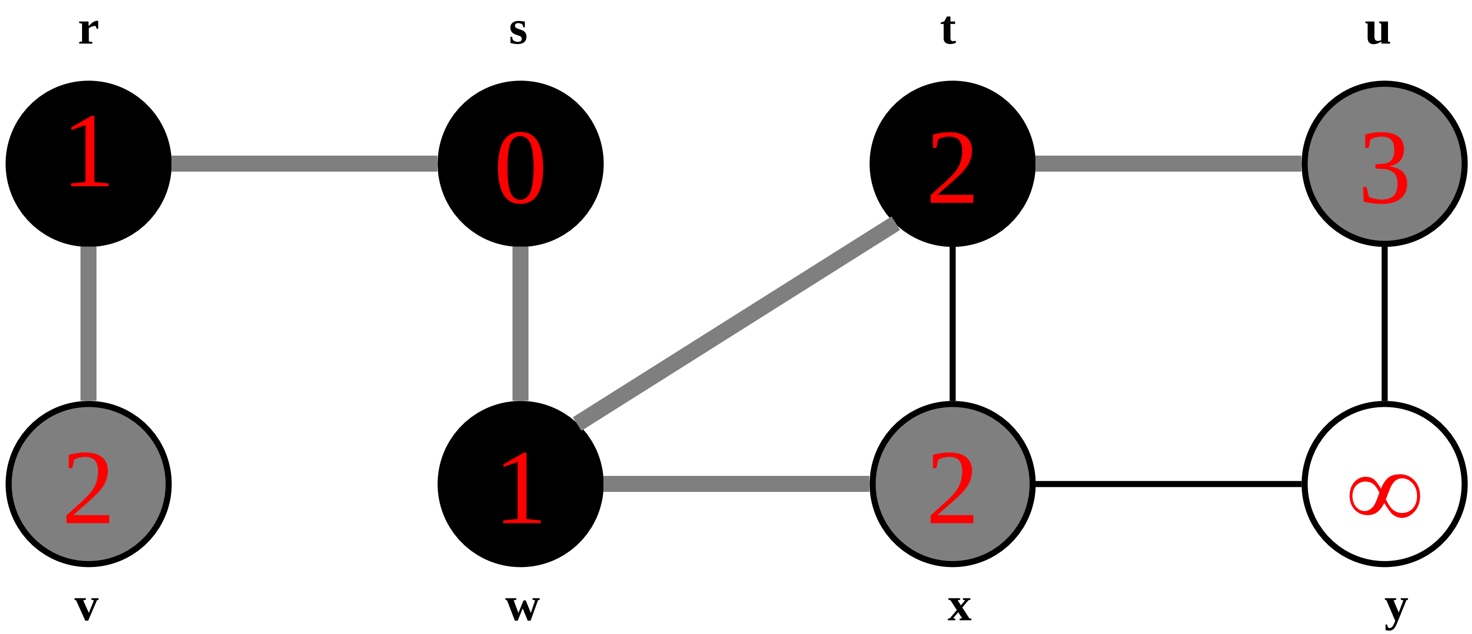


Breadth-First Search: Example



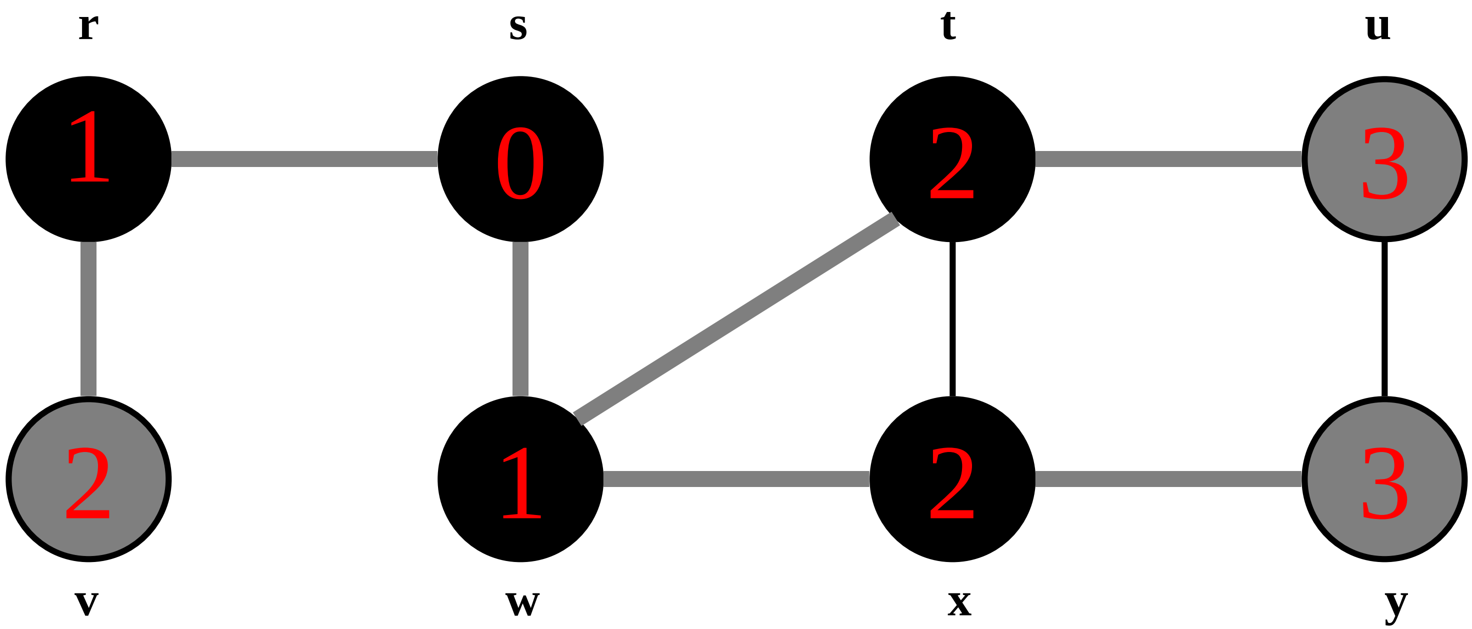
Q: | t | x | v |

Breadth-First Search: Example



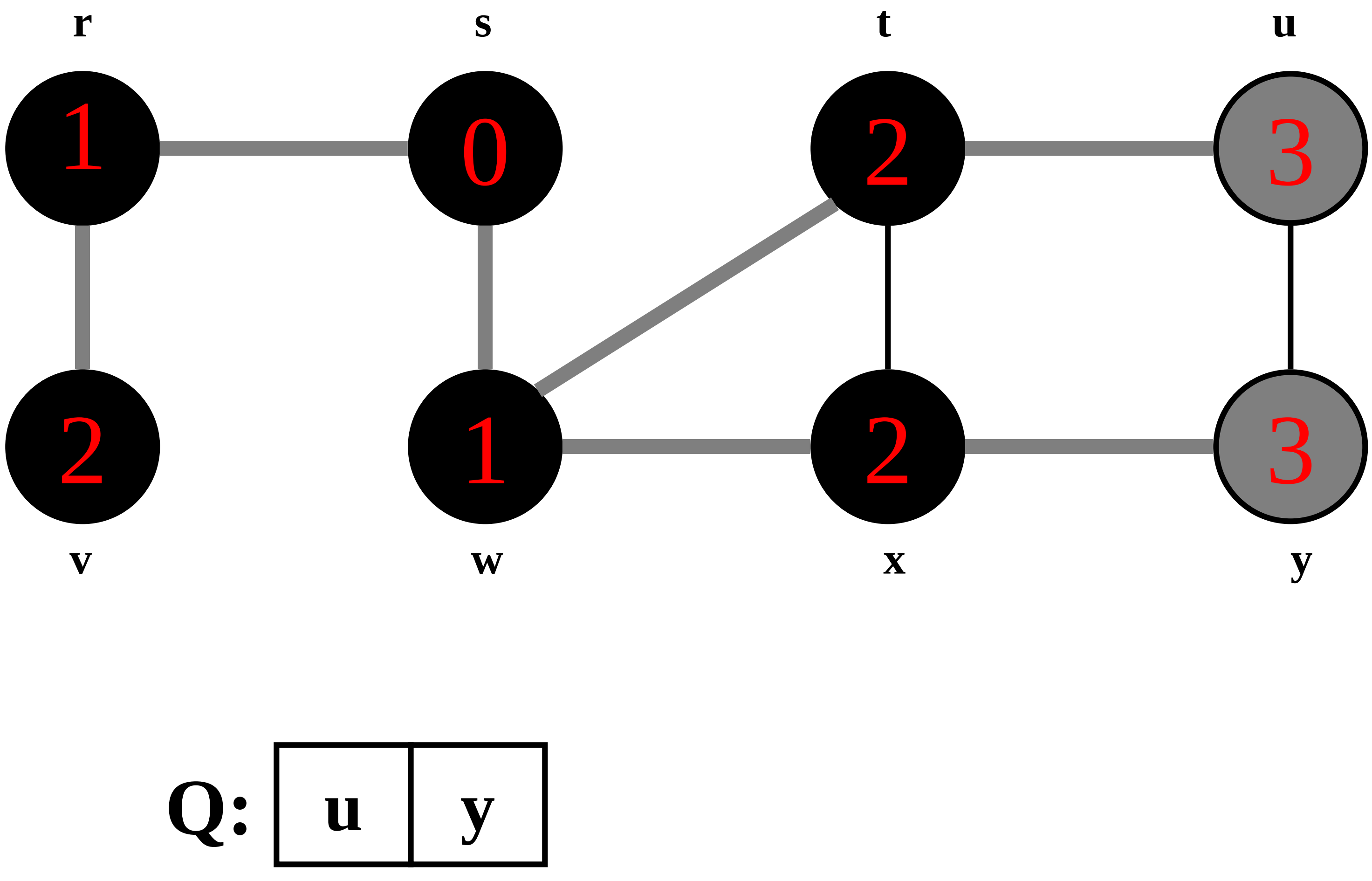
Q: | x | v | u |

Breadth-First Search: Example

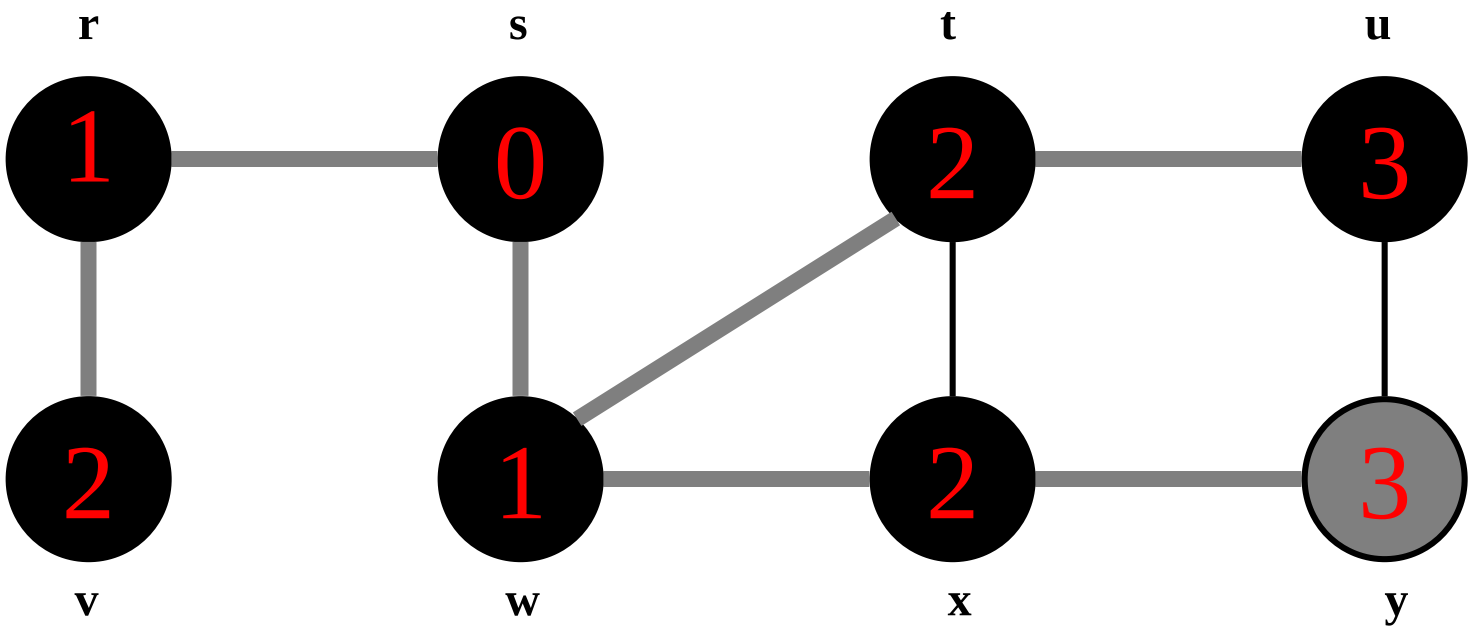


Q: | v | u | y |

Breadth-First Search: Example

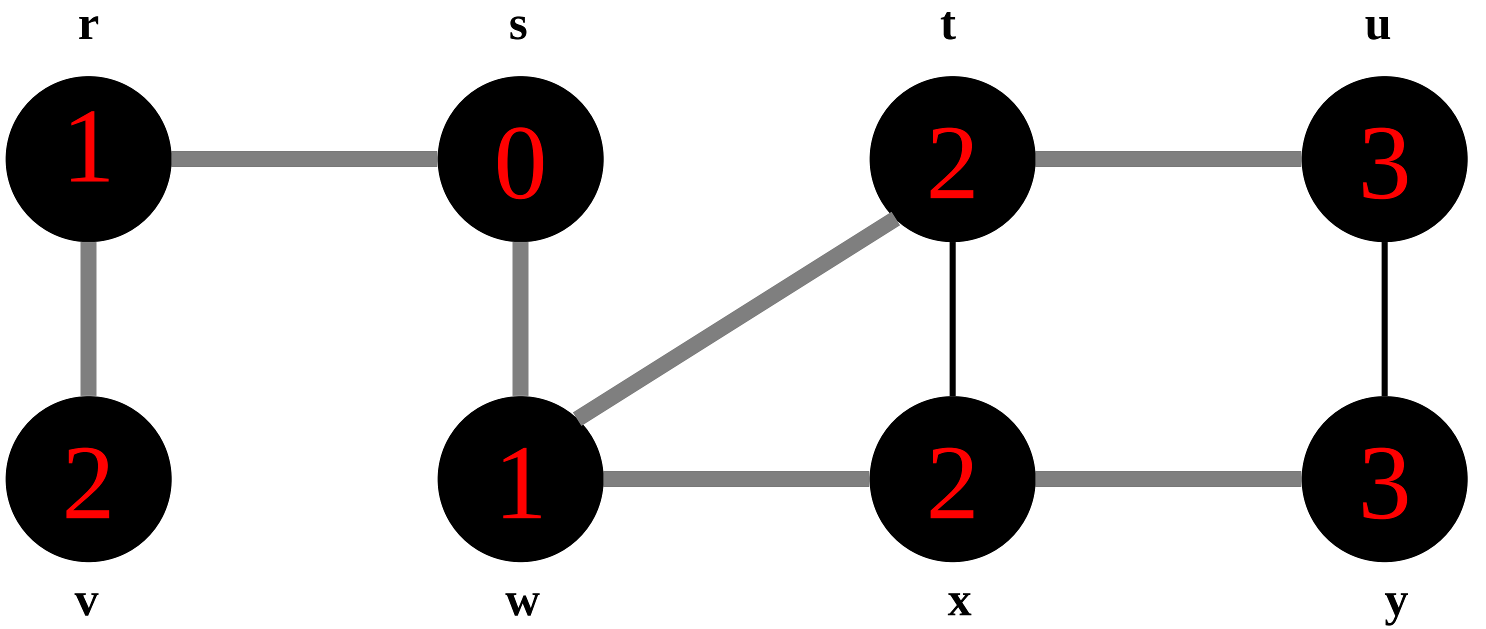


Breadth-First Search: Example



Q: y

Breadth-First Search: Example



Q: Ø

Analysis of BFS

- Initialization takes $O(|V|)$.

- Traversal Loop
  - After initialization, each vertex is enqueued and dequeued at most once, and each operation takes $O(1)$. So, total time for queuing is $O(|V|)$.

  - The adjacency list of each vertex is scanned at most once. The total time spent in scanning adjacency lists is $O(|E|)$.

- Summing up over all vertices => total running time of BFS is $O(|V| + |E|)$

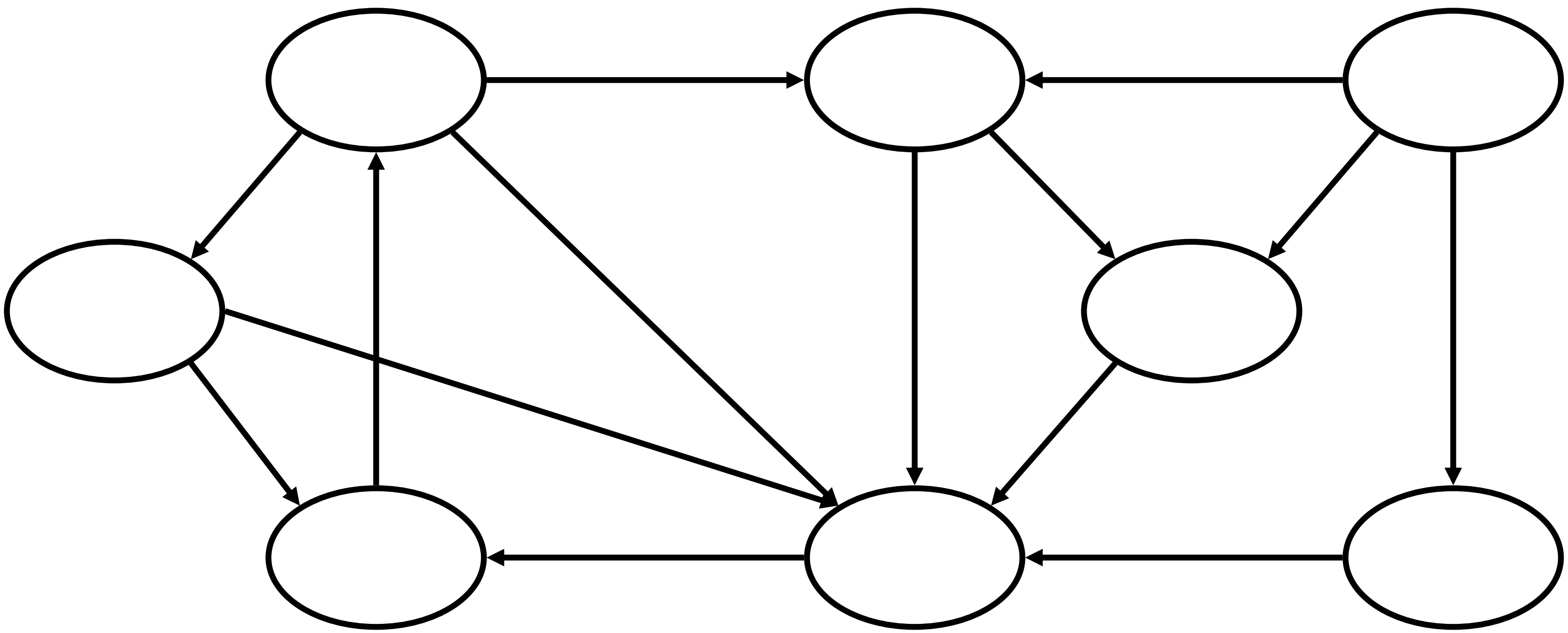
Depth-first Search (DFS)

- Explore edges out of the most recently discovered vertex v.
- When all edges of v have been explored, backtrack to explore other edges leaving the vertex from which v was discovered (its *predecessor*).
- "Search as deep as possible first."

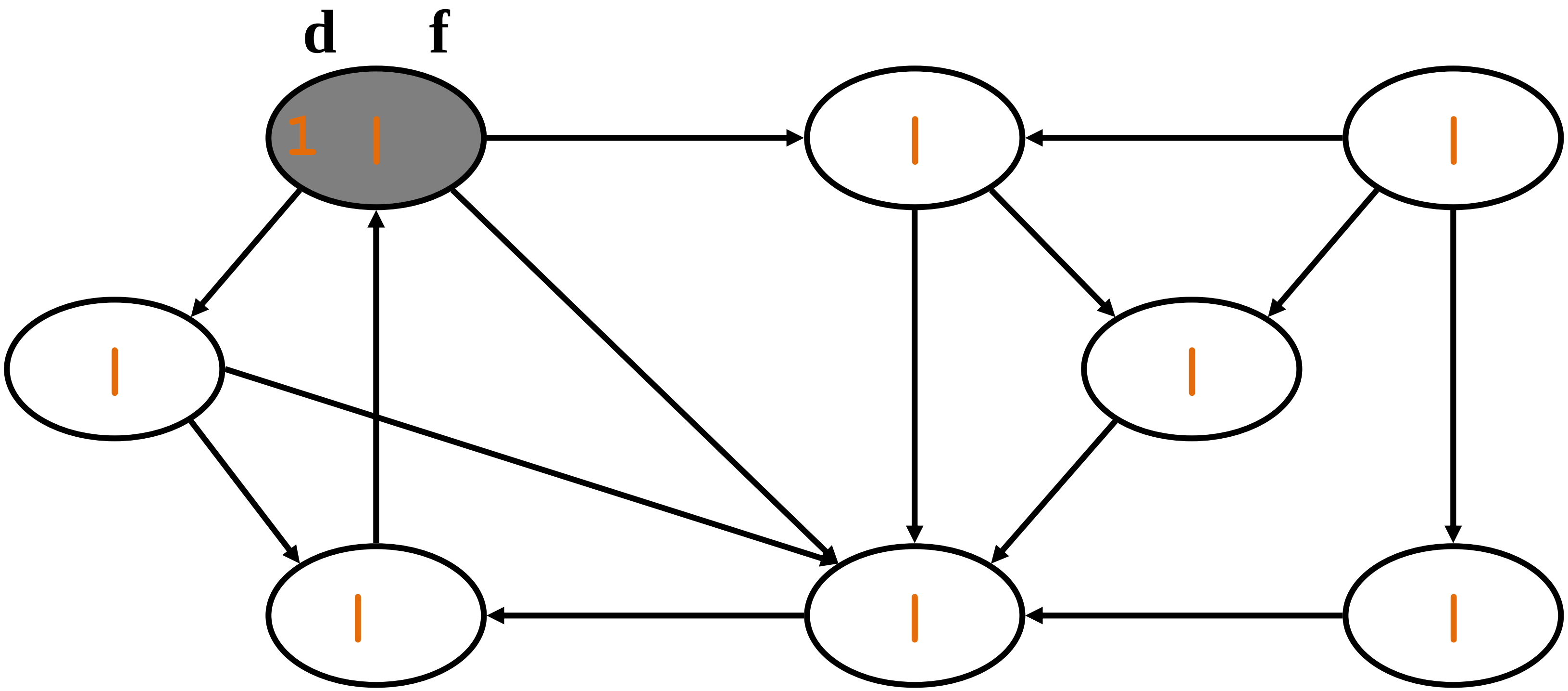
Depth-first Search

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - 2 **timestamps** on each vertex.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***finishing time*** (v turns from gray to black)
  - $\pi[v]$: predecessor of $v = u$, such that v was discovered during the scan of u's adjacency list.
  - Depth-first forest

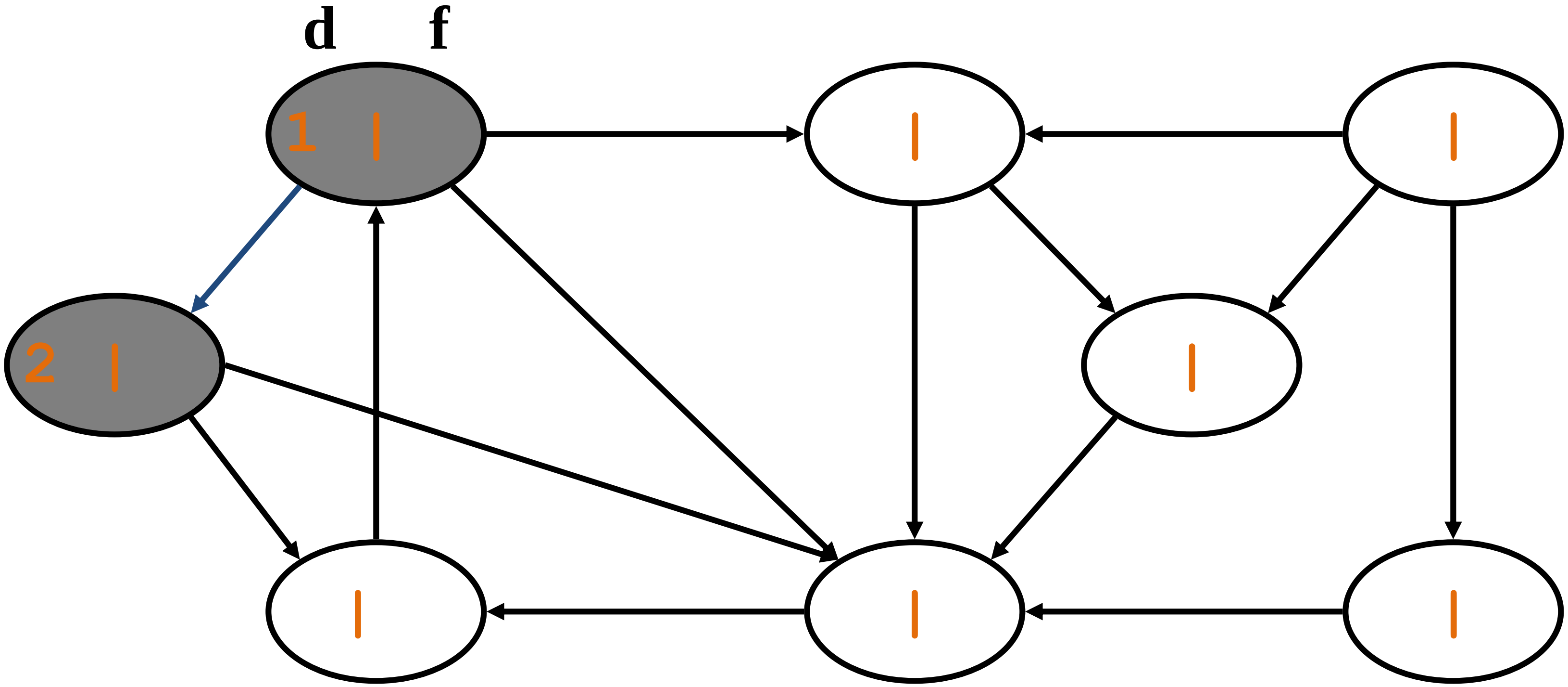
DFS Example



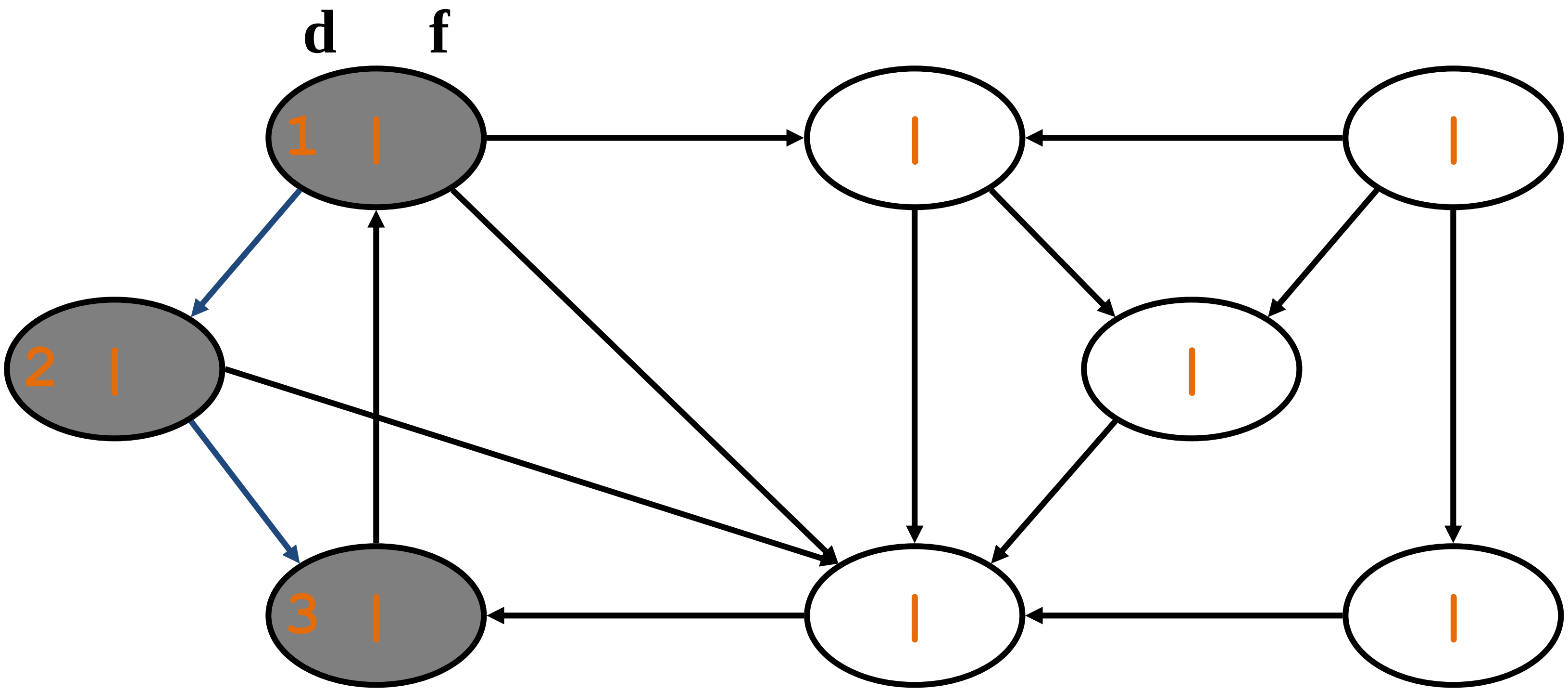
DFS Example



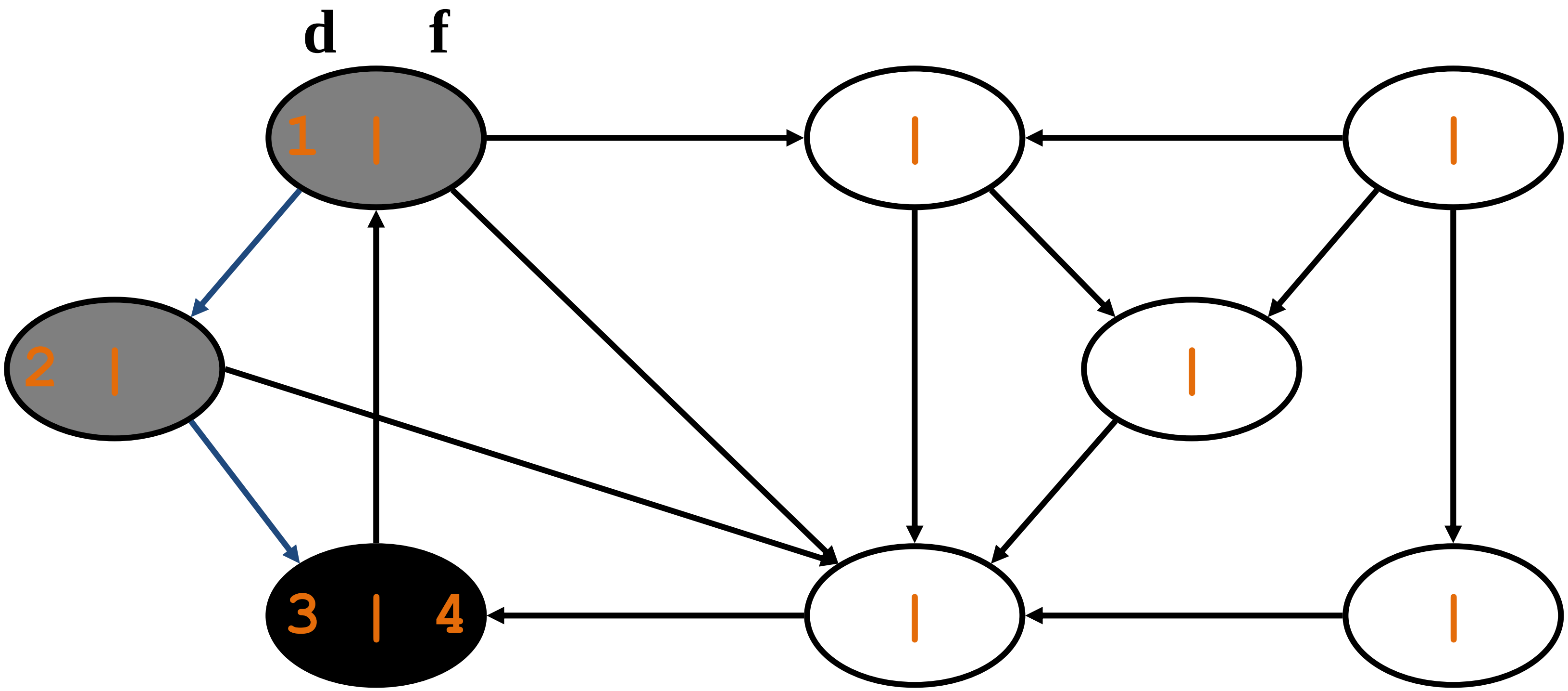
DFS Example



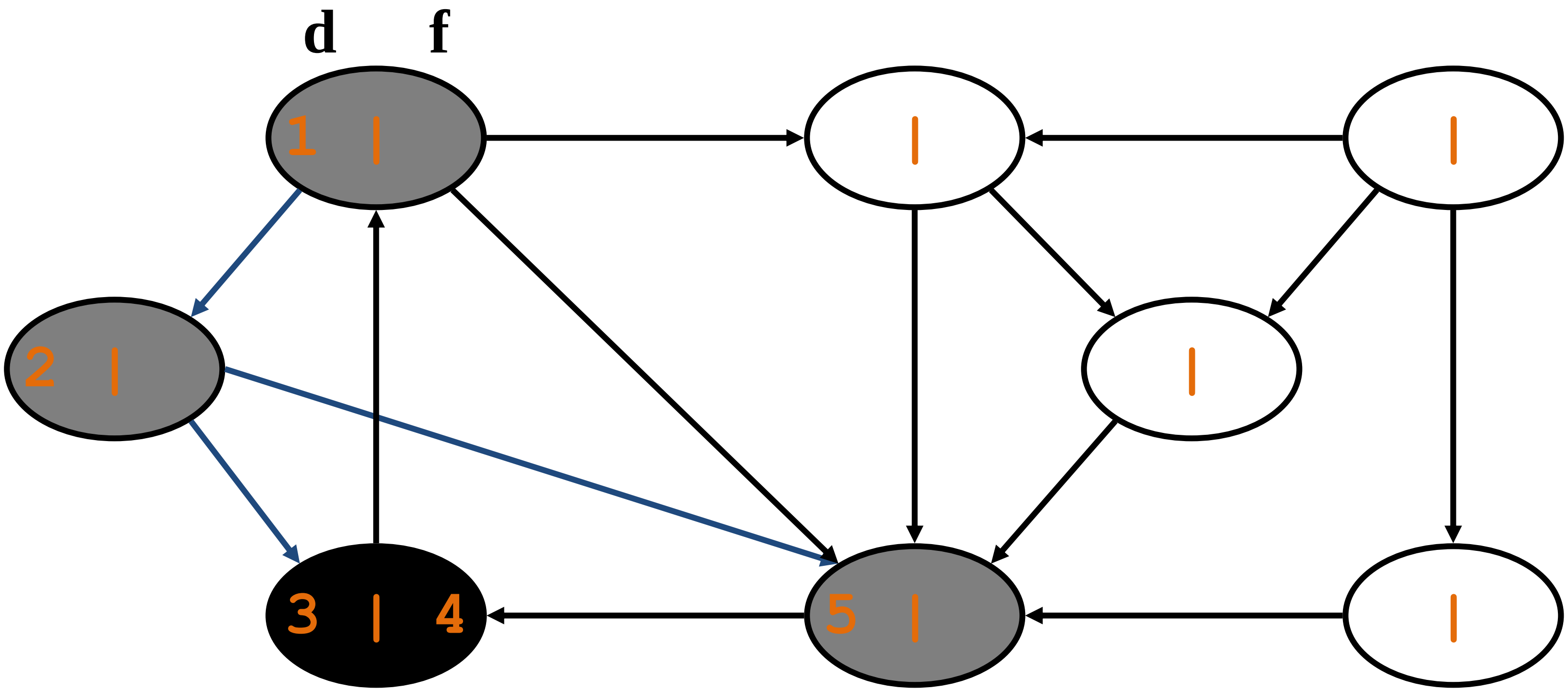
DFS Example



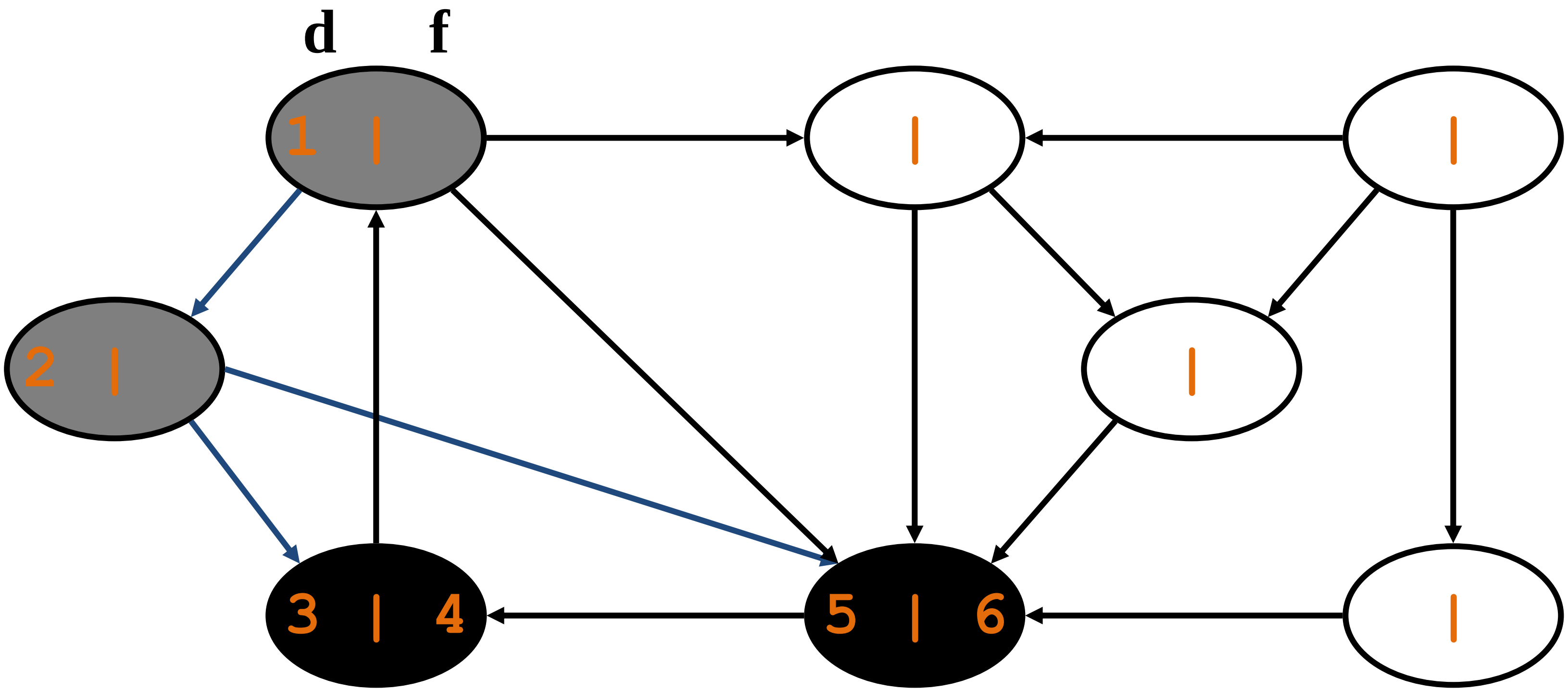
DFS Example



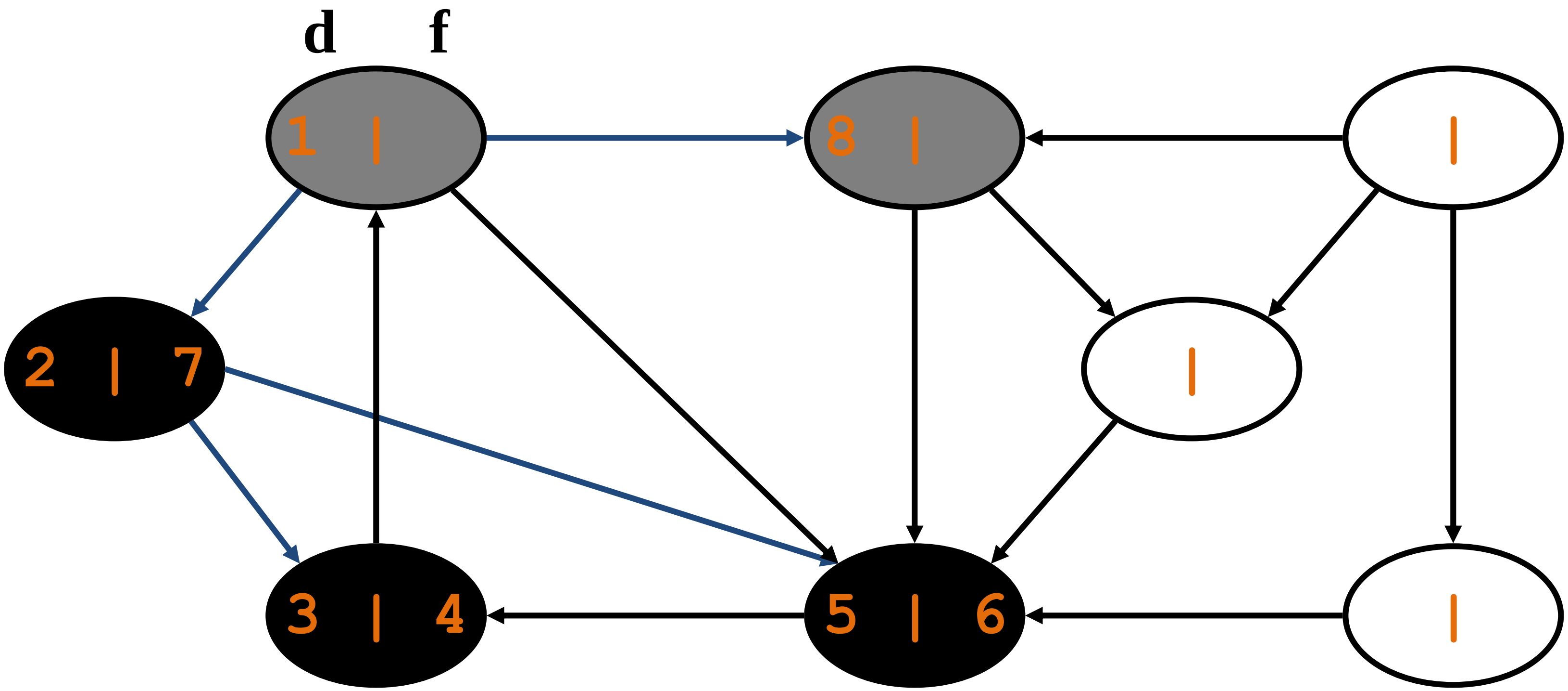
DFS Example



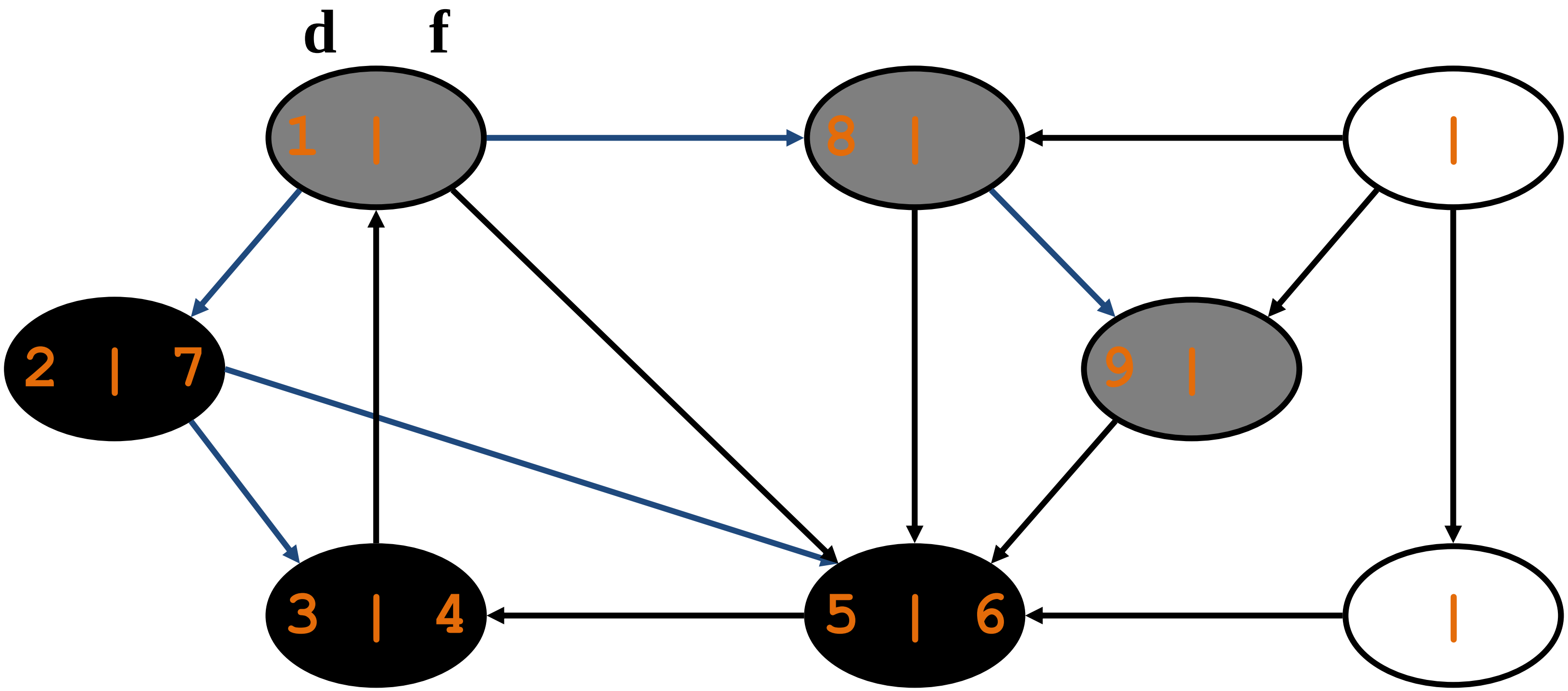
DFS Example



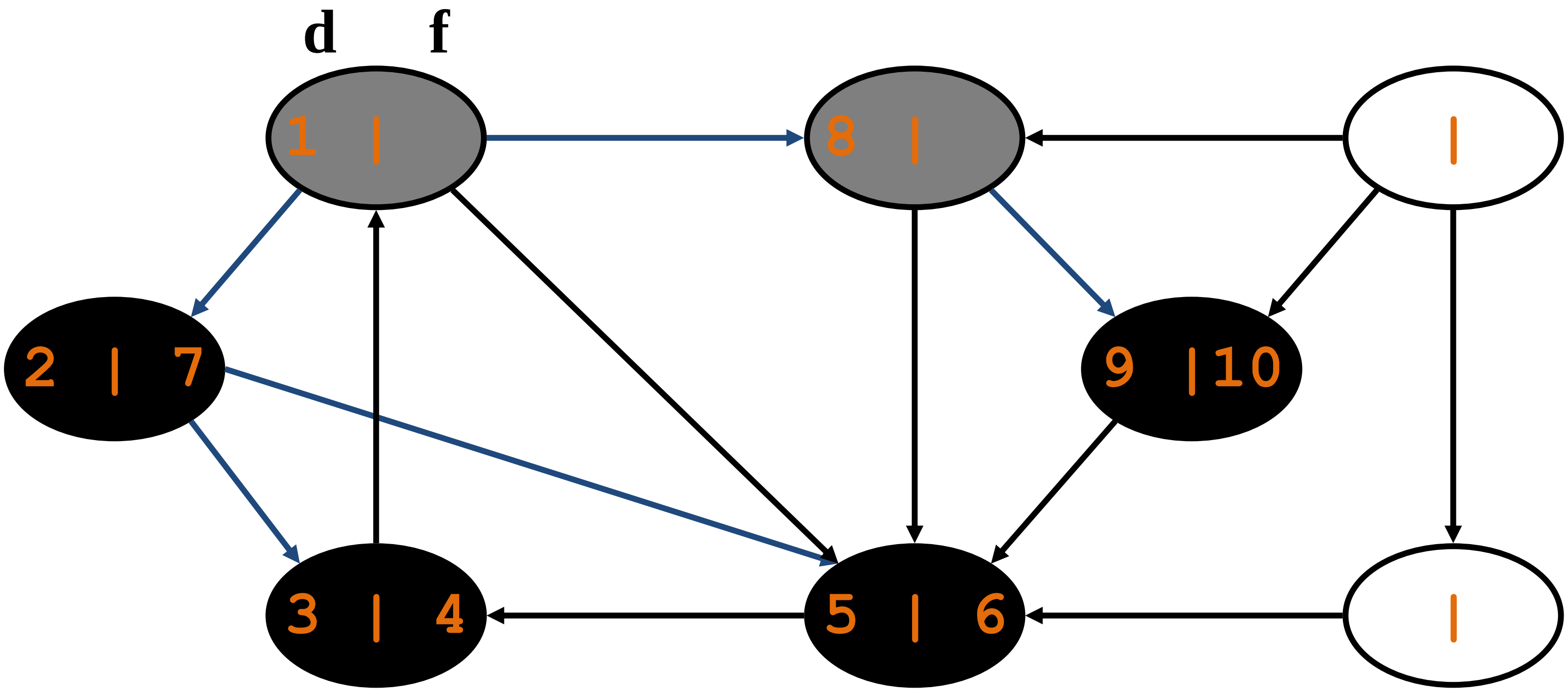
DFS Example



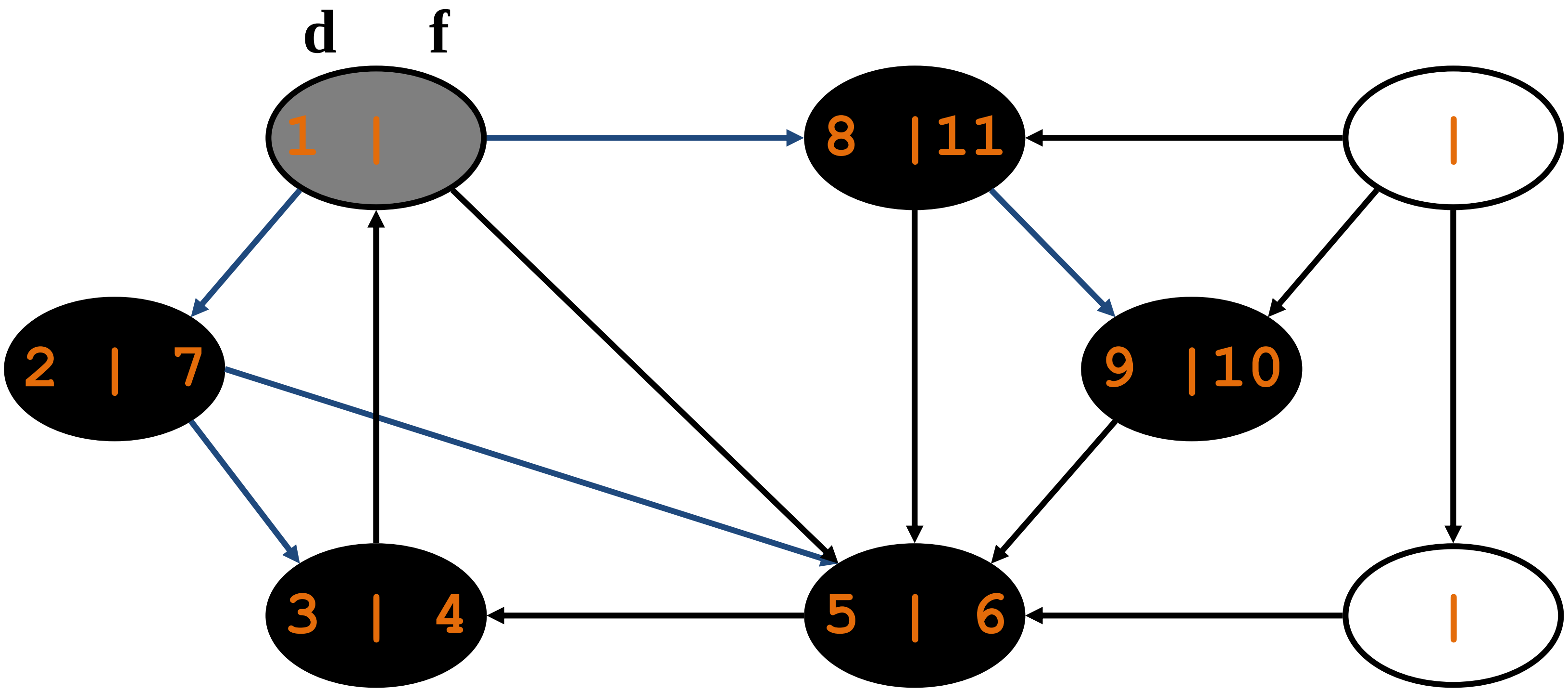
DFS Example



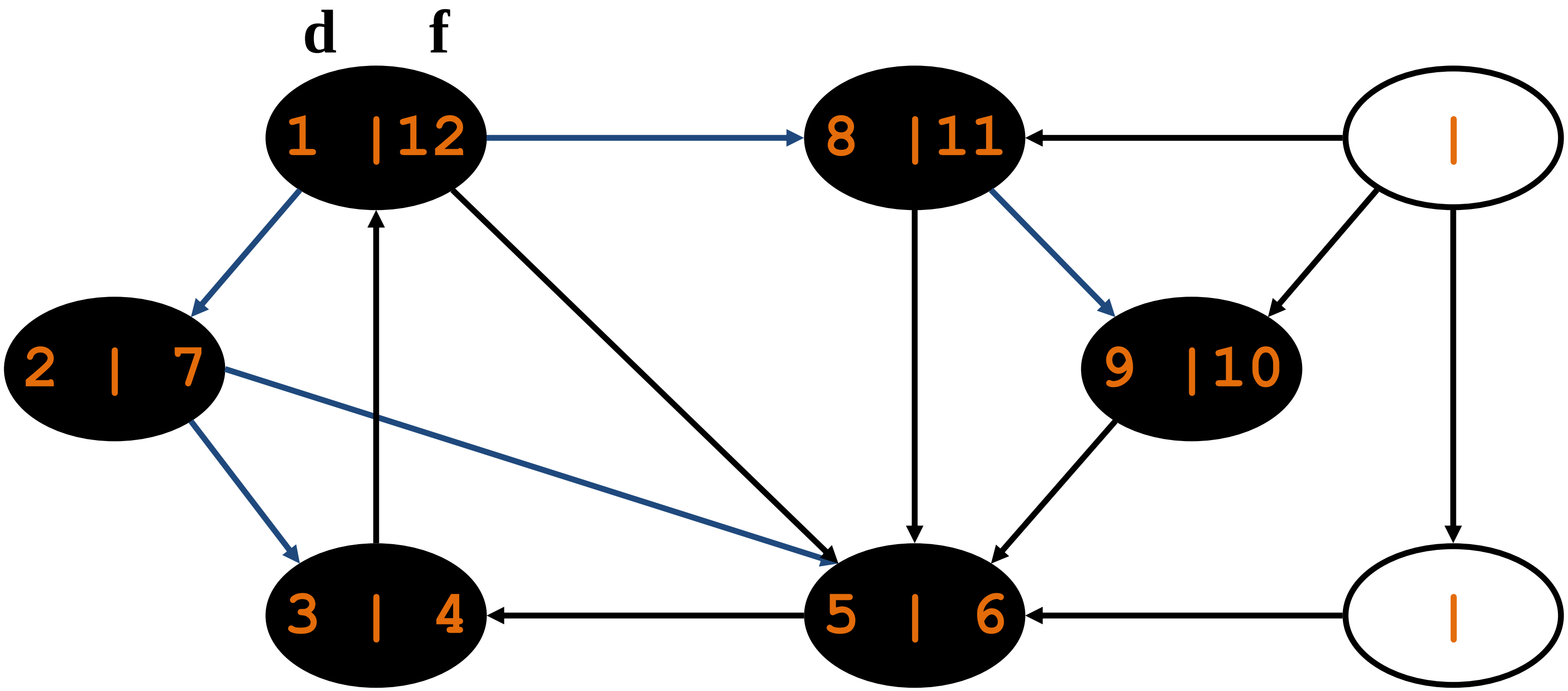
DFS Example



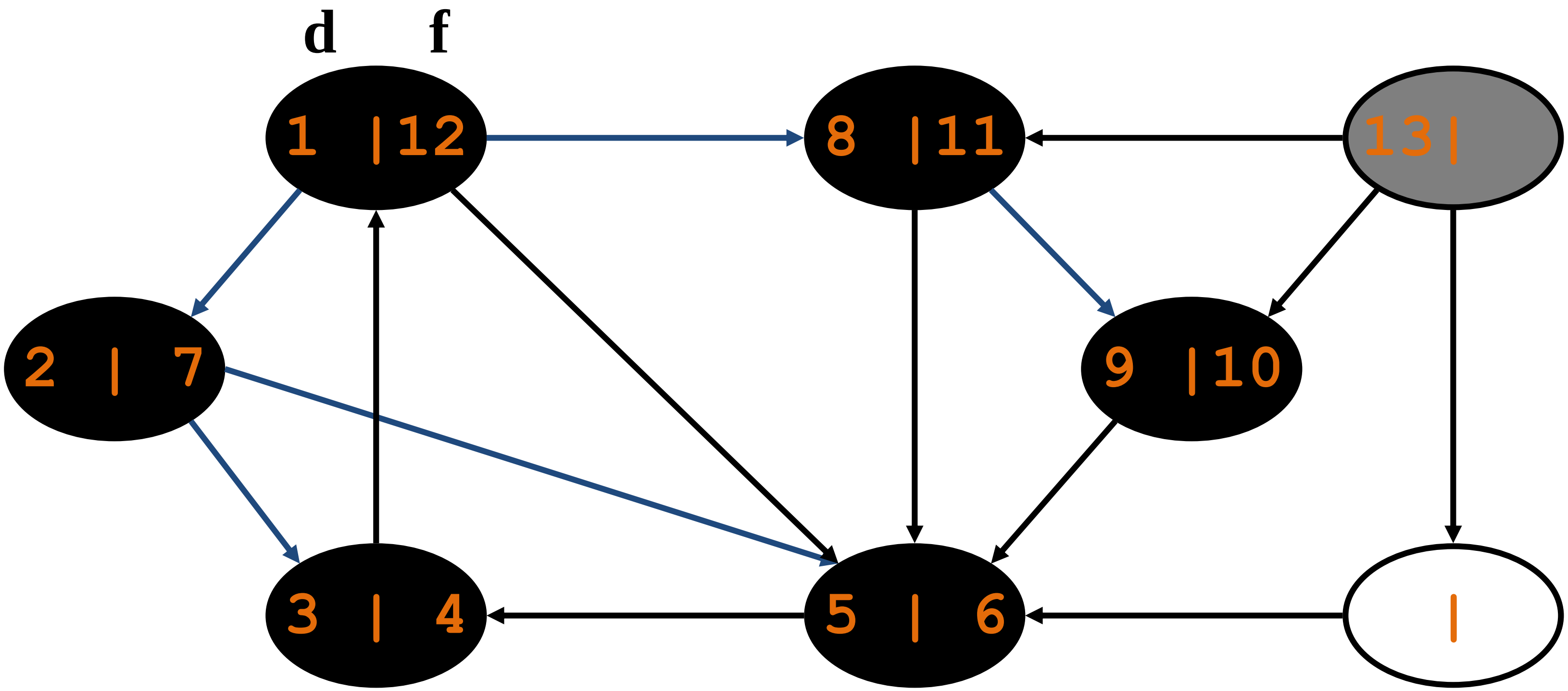
DFS Example



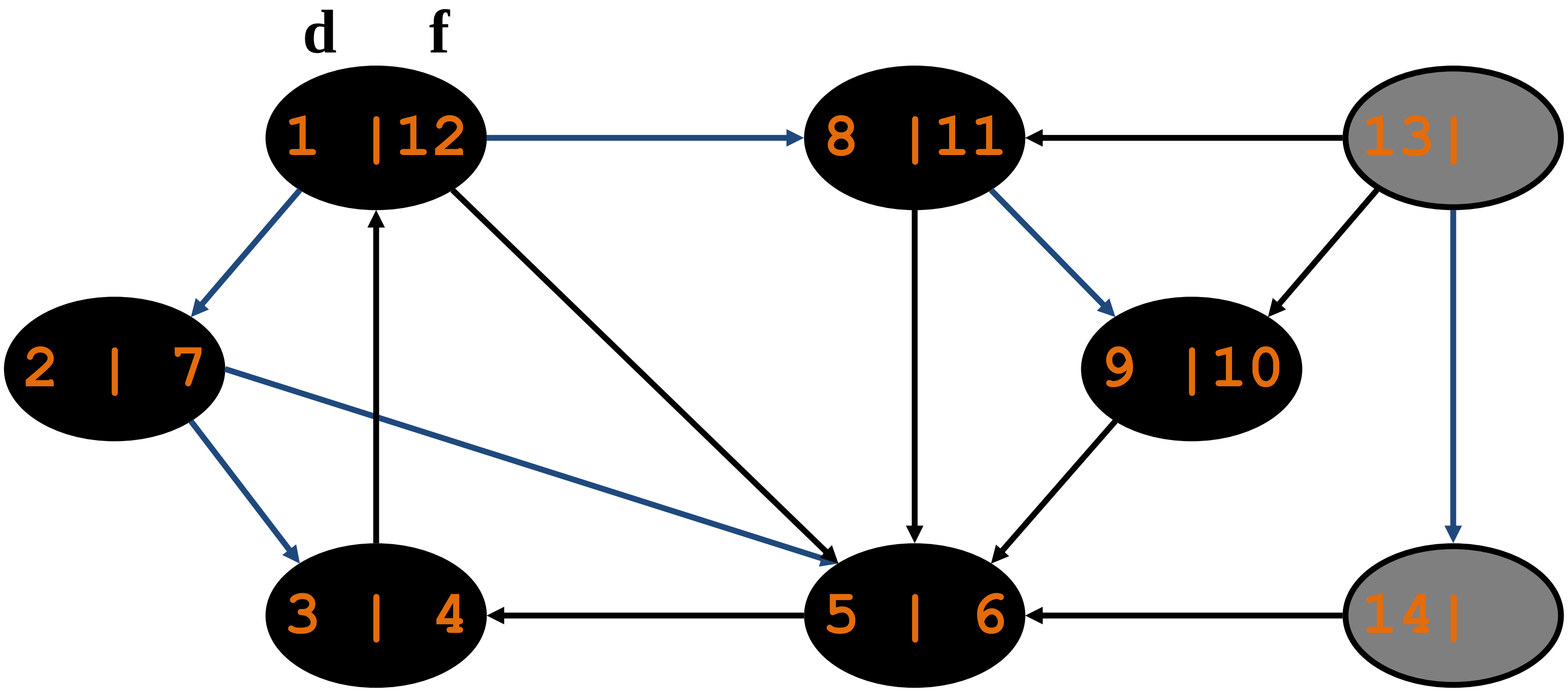
DFS Example



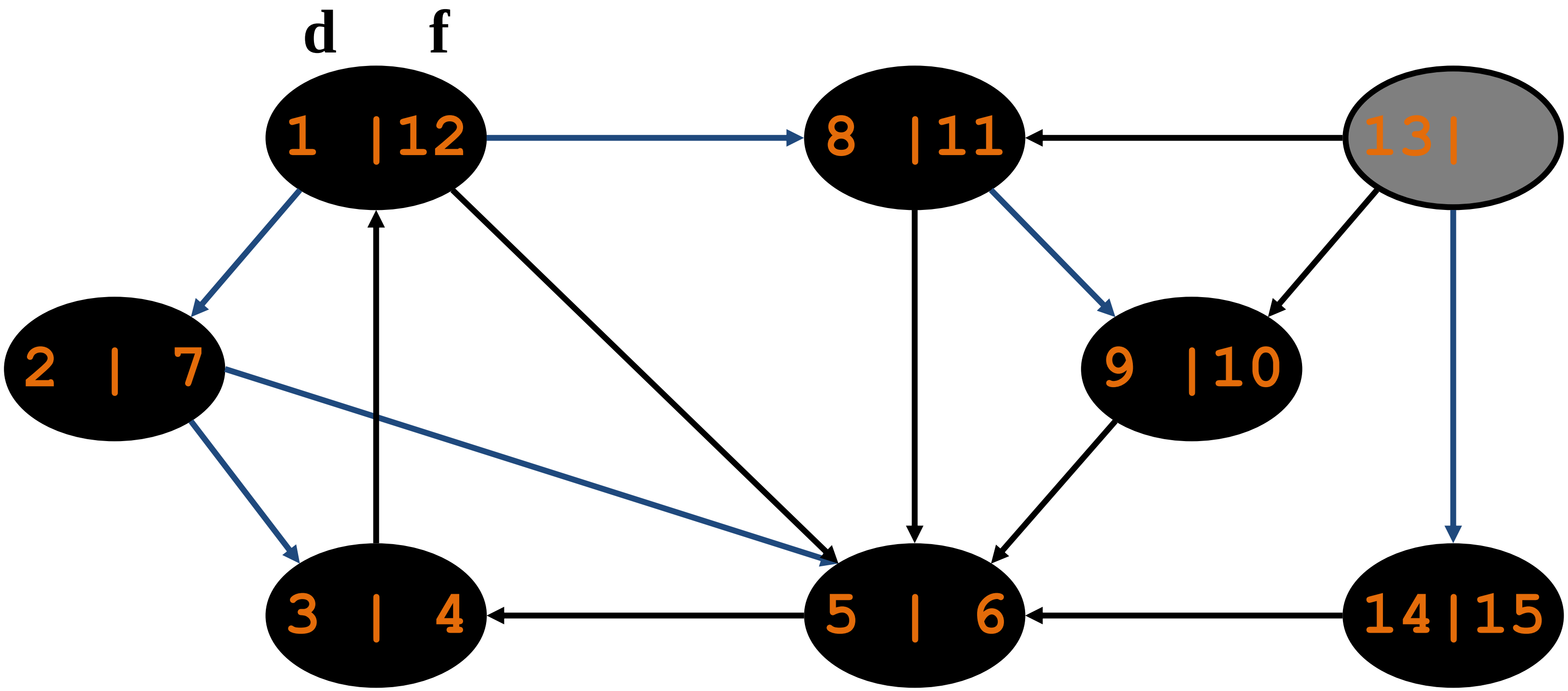
DFS Example



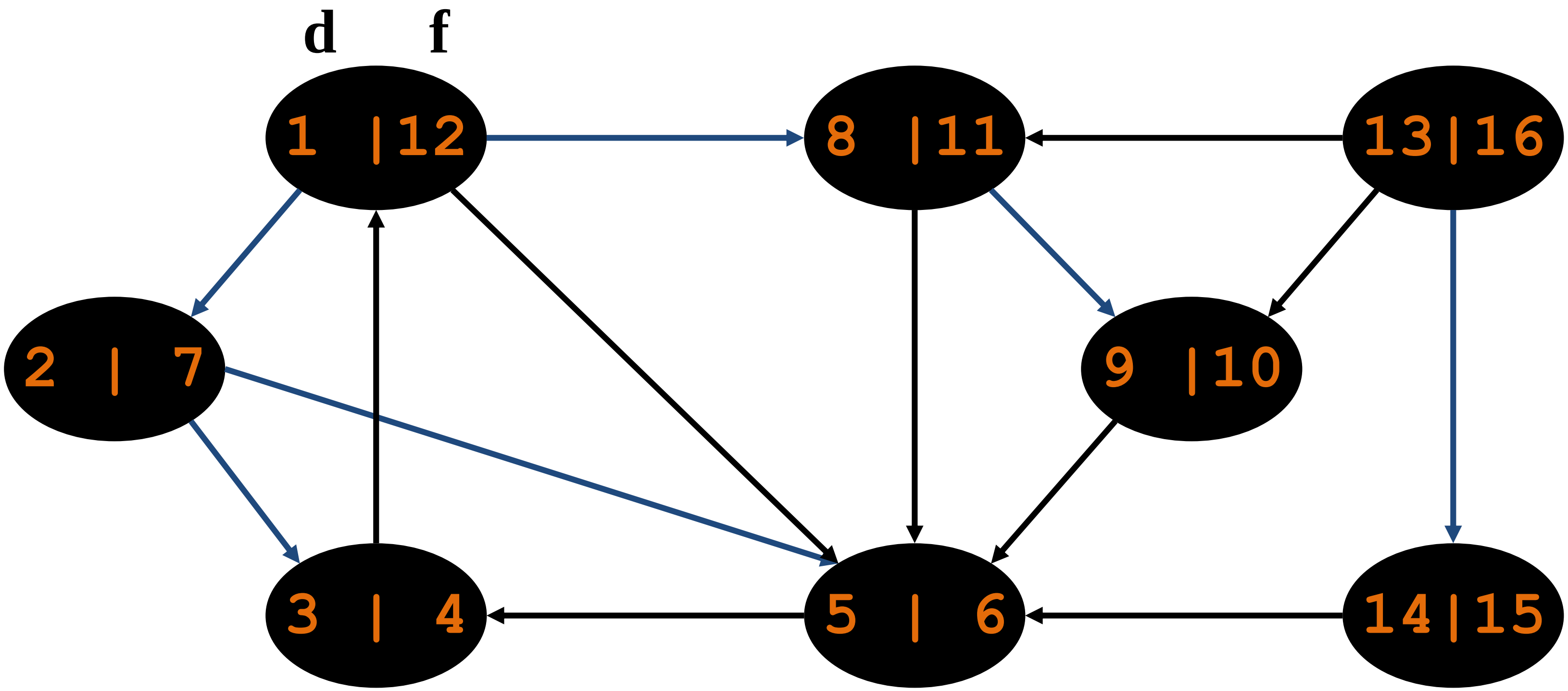
DFS Example



DFS Example



DFS Example



Pseudocode

DFS(*G*)

1. **for** each vertex $u \in V[G]$
2. **do** *color*$[u] \leftarrow$ white
3. $\pi[u] \leftarrow$ NIL
4. *time* $\leftarrow 0$
5. **for** each vertex $u \in V[G]$
6. **do if** *color*$[u]$ = white
7. **then** DFS-Visit(u)

Uses a global timestamp ***time***.

DFS-Visit(*u*)

1. *color*$[u] \leftarrow$ GRAY // White vertex u has been discovered
2. *time* $\leftarrow$ *time* + 1
3. $d[u] \leftarrow$ *time*
4. **for** each $v \in Adj[u]$
5. **do if** *color*$[v]$ = WHITE
6. **then** $\pi[v] \leftarrow u$
7. DFS-Visit(v)
8. *color*$[u] \leftarrow$ BLACK // Blacken u; it is finished.
9. *time* $\leftarrow$ *time* + 1
10. $f[u] \leftarrow$ *time*

Analysis of DFS

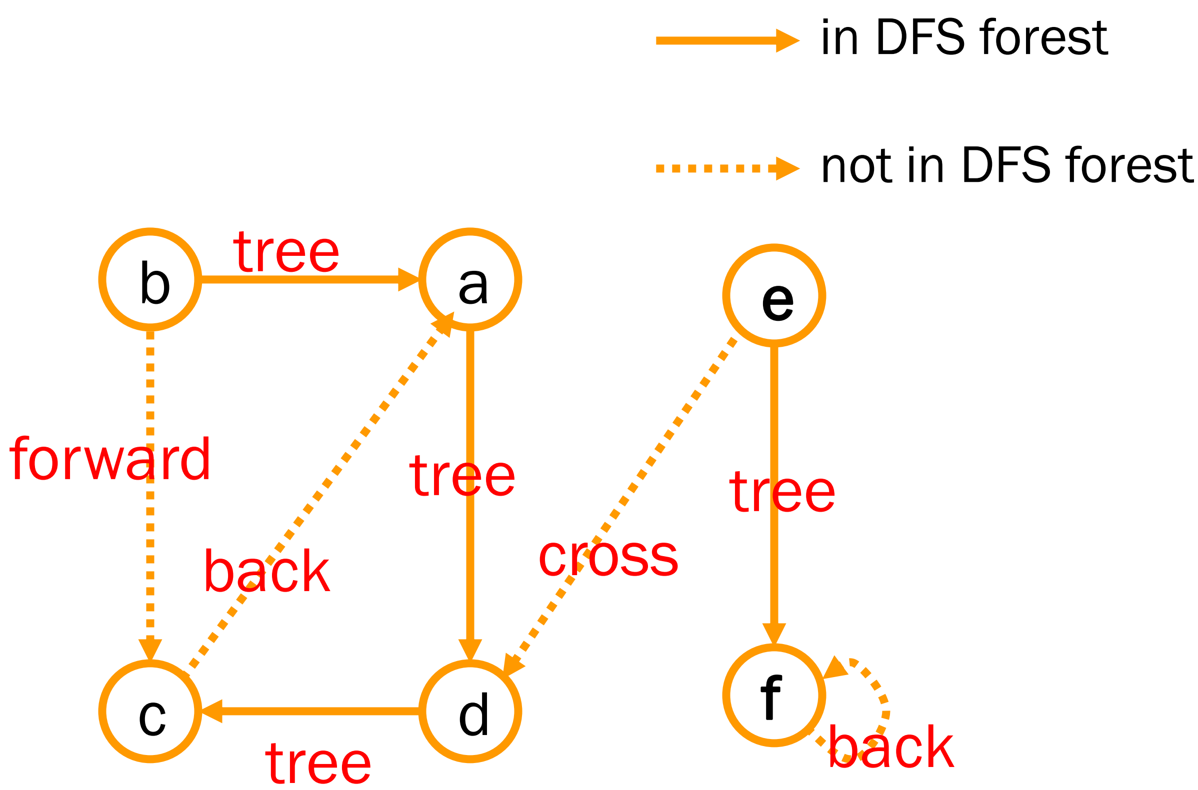
- Loops on lines 1-3 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.

- DFS-Visit is called once for each white vertex $v \in V$ when it's painted gray the first time. Lines 4-7 of DFS-Visit is executed $|\text{Adj}[v]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$

- Total running time of DFS is $\Theta(|V| + |E|)$.

DFS: Kinds of edges

- Consider a directed graph G = (V, E). After a DFS of graph G we can put each edge into one of four classes:
  - Tree edge
  - Back edge
  - Forward edge
  - Cross edge



Classifying edges of a Graph

- (u, v) is:
  - Tree edge – if v is white
  - Back edge – if v is gray
  - Forward or cross – if v is black
- (u, v) is:
  - Forward edge – if v is black and d[u] < d[v] (v was discovered after u)
  - Cross edge – if v is black and d[u] > d[v] (u was discovered after v)

DFS: Kinds of edges

DFS-Visit(*u*)

```
color[u] ← GRAY
time ← time + 1
d[u] ← time
for each vertex v adjacent to u
   do if color[v] ← BLACK
        then if d[u] < d[v]
               then Classify (u, v) as a forward edge
               else Classify (u, v) as a cross edge
        if color[v] ← GRAY
               then Classify (u, v) as a back edge
        if color[v] ← WHITE
               then π[v] ← u
                    Classify (u, v) as a tree edge
                    DFS-Visit(v)
 color[u] ← BLACK
 time ← time + 1
 f[u] ← time
```

Some Applications of BFS and DFS

- BFS
  - To find the shortest path from a vertex *s to a vertex v in* an unweighted graph
  - To find the length of such a path
  - Find the bipartiteness of a graph.

- DFS
  - To find a path from a vertex *s to a vertex v.*
  - To find the length of such a path.
  - To find out if a graph contains cycles

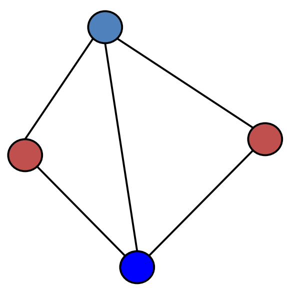
Application of BFS: Bipartite Graph

- Graph $G = (V,E)$ is **bipartite** iff V can be partitioned into two sets of nodes A and B such that each edge has one end in A and the other end in B

Alternatively:

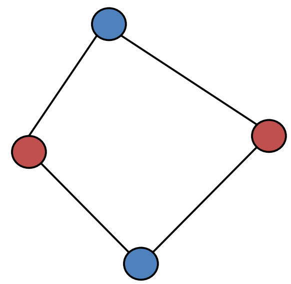
- Graph $G = (V,E)$ is bipartite iff all its cycles have even length
- Graph $G = (V,E)$ is bipartite iff nodes can be coloured using two colours

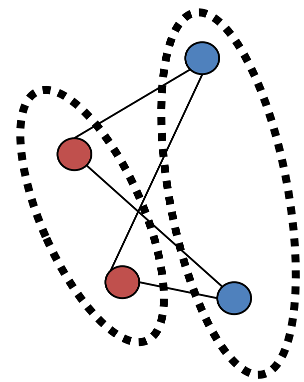
Application of BFS: Bipartite Graph



non bipartite

bipartite:





Question: given a graph G, how to test if the graph is bipartite?

Application of BFS: Bipartite Graph

```
For each vertex u in V[G] − {s}
  do color[u] ← WHITE
       d[u] ← ∞
       partition[u] ← 0
color[s] ← GRAY
partition[s] ← 1
d[s] ← 0
Q ← [s]
while Queue 'Q' is non-empty
     do u  ← head [Q]
         for each v in Adj[u] do
             if partition [u] = partition [v] then
                          return 0
             else if color[v] ← WHITE then
                            color[v] ← gray
                            d[v] = d[u] + 1
                            partition[v] ← 3 − partition[u]
                            ENQUEUE (Q, v)
         DEQUEUE (Q)
          Color[u] ← BLACK
Return 1
```

Application of DFS: Detecting Cycle for Directed Graph

```
DFS_visit(u)
    color(u) ← GRAY
    time ← time + 1
    d[u] ← time
    for each v adjacent to u  do
        if color[v] ← GRAY then
            return "cycle exists"
        else if color[v] ← WHITE then
            predecessor[v] ←  u
            DFS_visit(v)
    color[u] ← BLACK
    time ← time + 1
    f[u] ← time
```

Application of DFS: Detecting Cycle for Undirected Graph

```
DFS_visit(u)
    color(u) ← GRAY
    time ← time + 1
    d[u] ← time
    for each v adjacent to u  do
        if color[v] ← GRAY and π[u] ≠ v then
            return "cycle exists"
        else if color[v] ← WHITE then
            predecessor[v] ←  u
            DFS_visit(v)
    color[u] ← BLACK
    time ← time + 1
    f[u] ← time
```

Self-Study

- *Lemma 22.1, 22.2, Theorem 22.5, 22.10.*
- *Excercises:*
  - *22.2-2, 22.2-4, 22.3-2, 22.3-3, 22.3-9*