CSE 208: Data Structure and Algorithms II

Introduction and Graph Basics

Dr. Mohammed Eunus Ali

Professor

CSE, BUET

Slides are from Dr Tanzima Hashem, **tanzimahashem@cse.buet.ac.bd**

Graphs

- 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

- 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→v
    - Self loops are allowed.

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)

Graphs

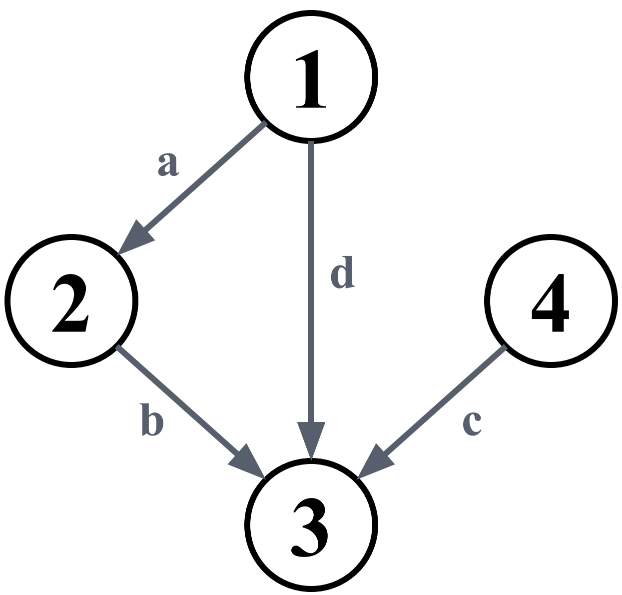
- We will typically express running times in terms of $|E|$ and $|V|$ (often dropping the |'s)
  - If $|E| \approx |V|^2$ the graph is *dense*
  - If $|E| \approx |V|$ the graph is *sparse*
- If you know you are dealing with dense or sparse graphs, different data structures may make sense

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

Graphs: Adjacency Matrix

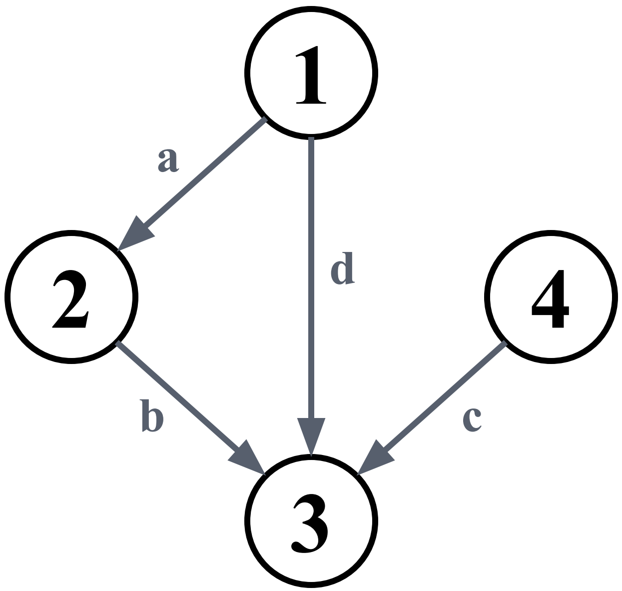
- Example:



A	1	2	3	4
1				
2				
3			??	
4				

Graphs: Adjacency Matrix

- Example:



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

Graphs: Adjacency Matrix

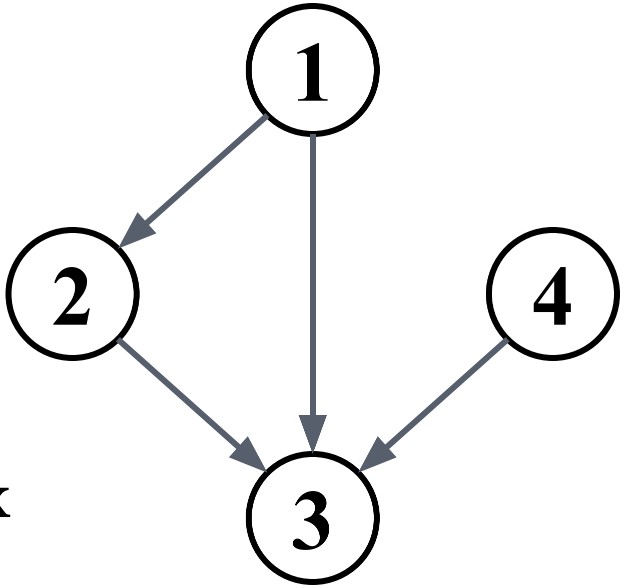
- **Space:** $\Theta(V^2)$.
  - Not memory efficient for large graphs.
- **Time:** to list all vertices adjacent to u: $\Theta(V)$.
- **Time:** to determine if $(u, v) \in E$: $\Theta(1)$.

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

- 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

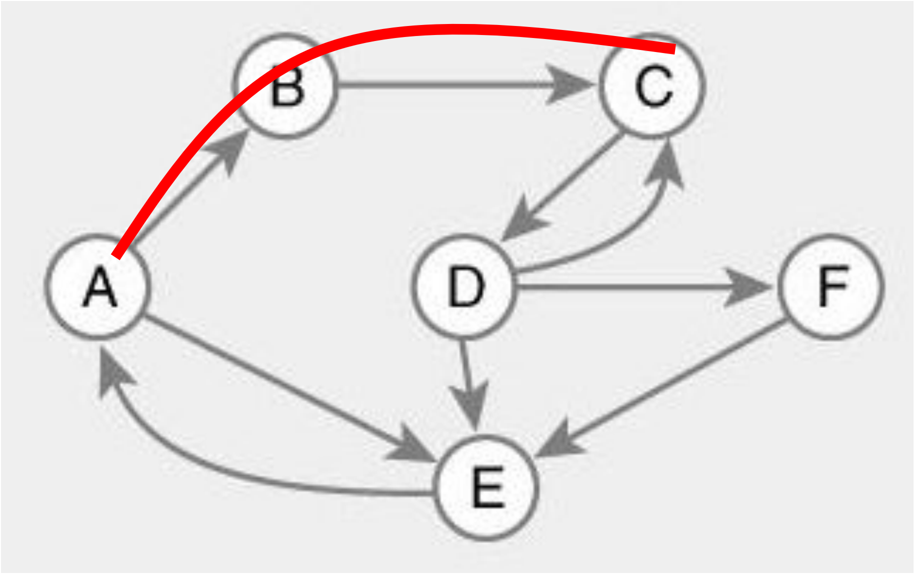
$$\sum_{v \in V} \text{degree}(v) = 2|E|$$

No. of edges incident on v. Edge (u,v) is incident on vertices u and v.

  - Total storage: $\Theta(V+E)$

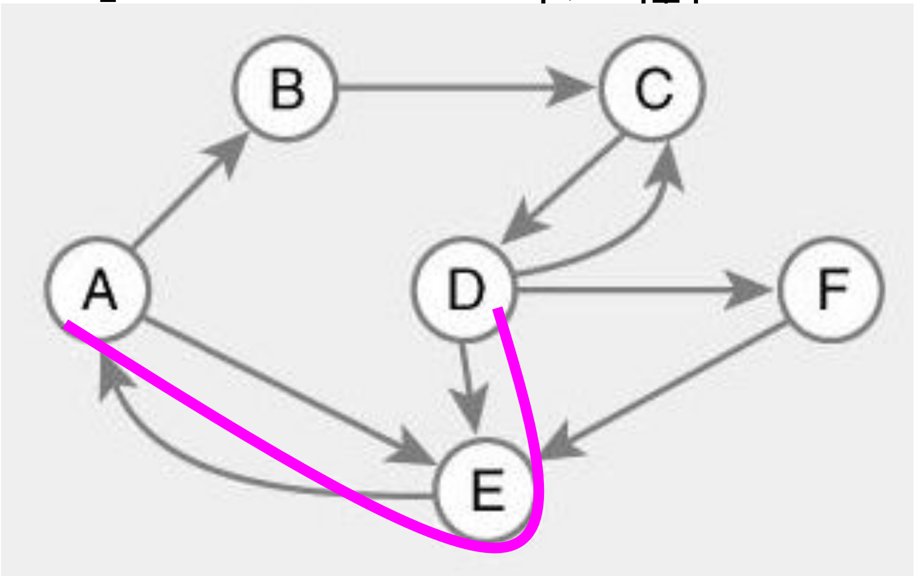
Graph Definitions

- Path
  - Sequence of nodes $n_1, n_2, \dots n_k$
  - Edge exists between each pair of nodes n_i , n_{i+1}
  - Example
    - A, B, C is a path



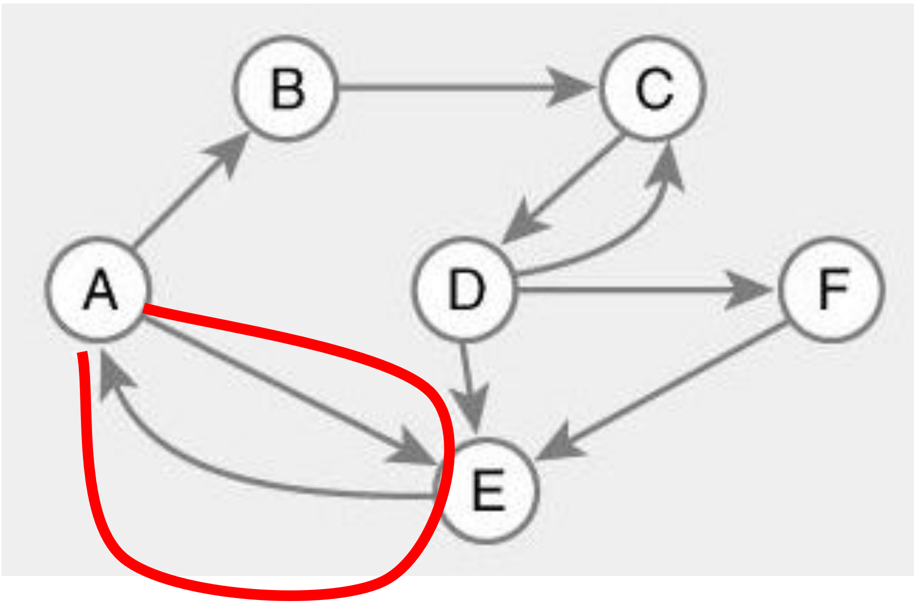
Graph Definitions

- Path
  - Sequence of nodes $n_1, n_2, \ldots n_k$
  - Edge exists between each pair of nodes n_i , n_{i+1}
  - Example
    - A, B, C is a path
    - A, E, D is not a path



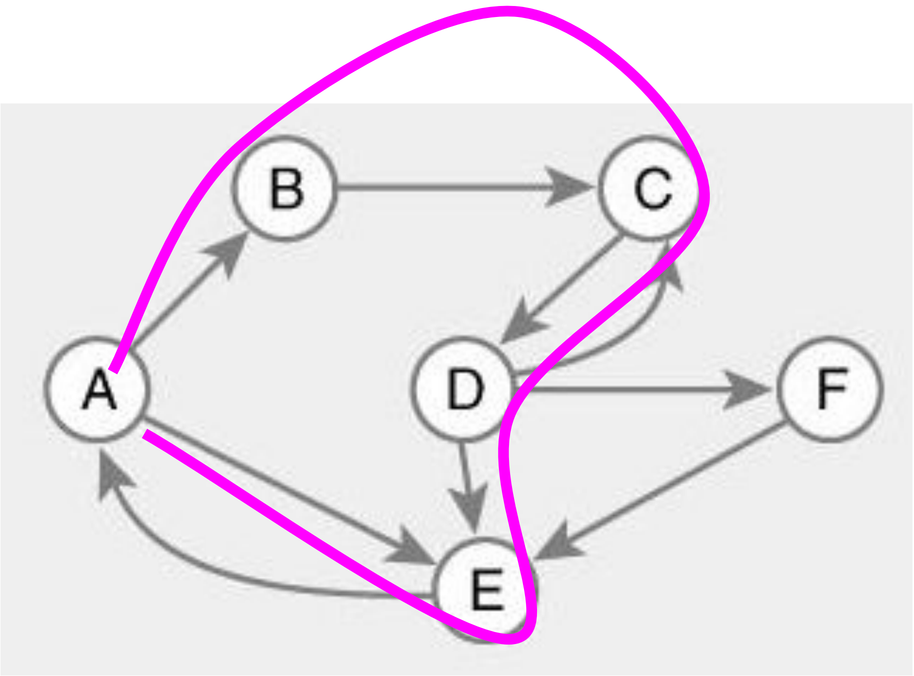
Graph Definitions

- Cycle
  - Path that ends back at starting node
  - Example
    - A, E, A



Graph Definitions

- Cycle
  - Path that ends back at starting node
  - Example
    - A, E, A
    - A, B, C, D, E, A
- Simple path
  - No cycles in path
- Acyclic graph
  - No cycles in graph



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.

BFS - Version -1

```
BFS (s,Adj)
    level ={s:0}
    parent ={s:null}
    i =0
    frontiers = [s]
    while frontiers:
        next = []
        for u in frontiers
            for v in Adj[u]
                if v is not in level
                    level[v] = i
                    paren[v] = u
                    next.append(v)
        i = i+1
        frontiers = next
```

Breadth-First Search

- **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 $s \leadsto v$.
    - u is v's predecessor.
  - Builds breadth-first tree with root s that contains all reachable vertices.

Breadth-First Search

- 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

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
```

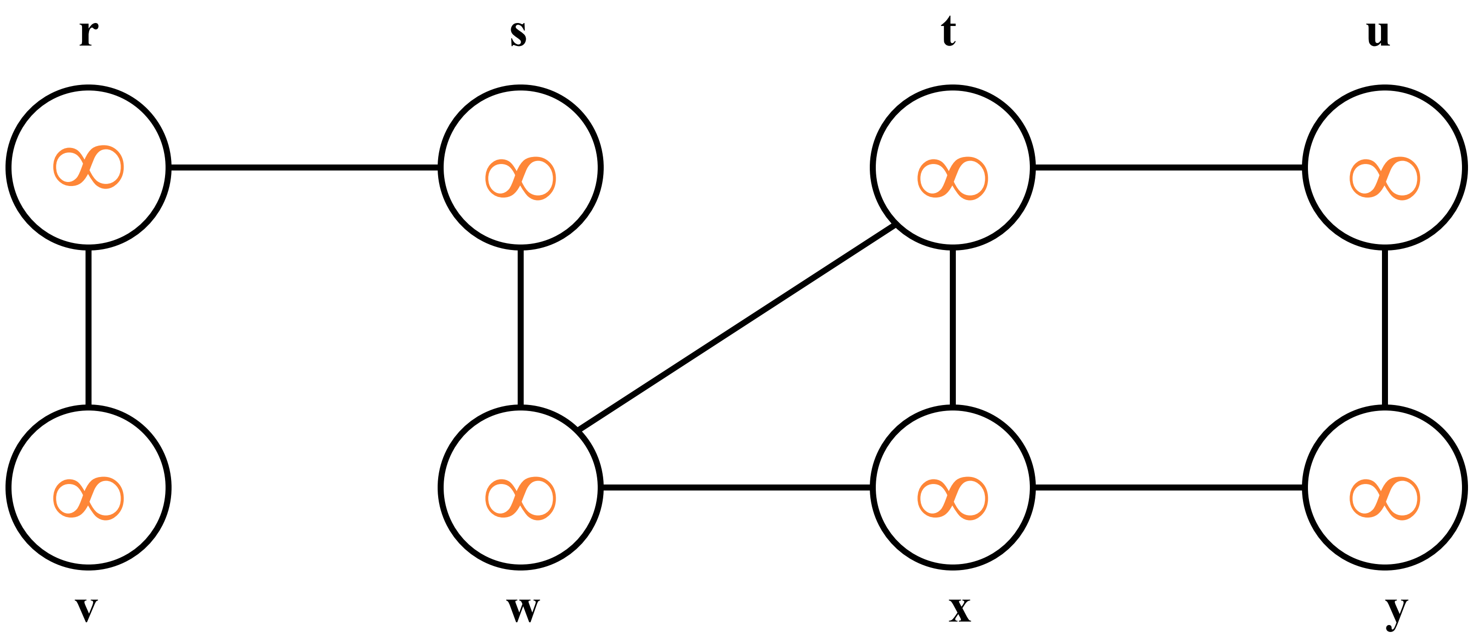
initialization

access source *s*

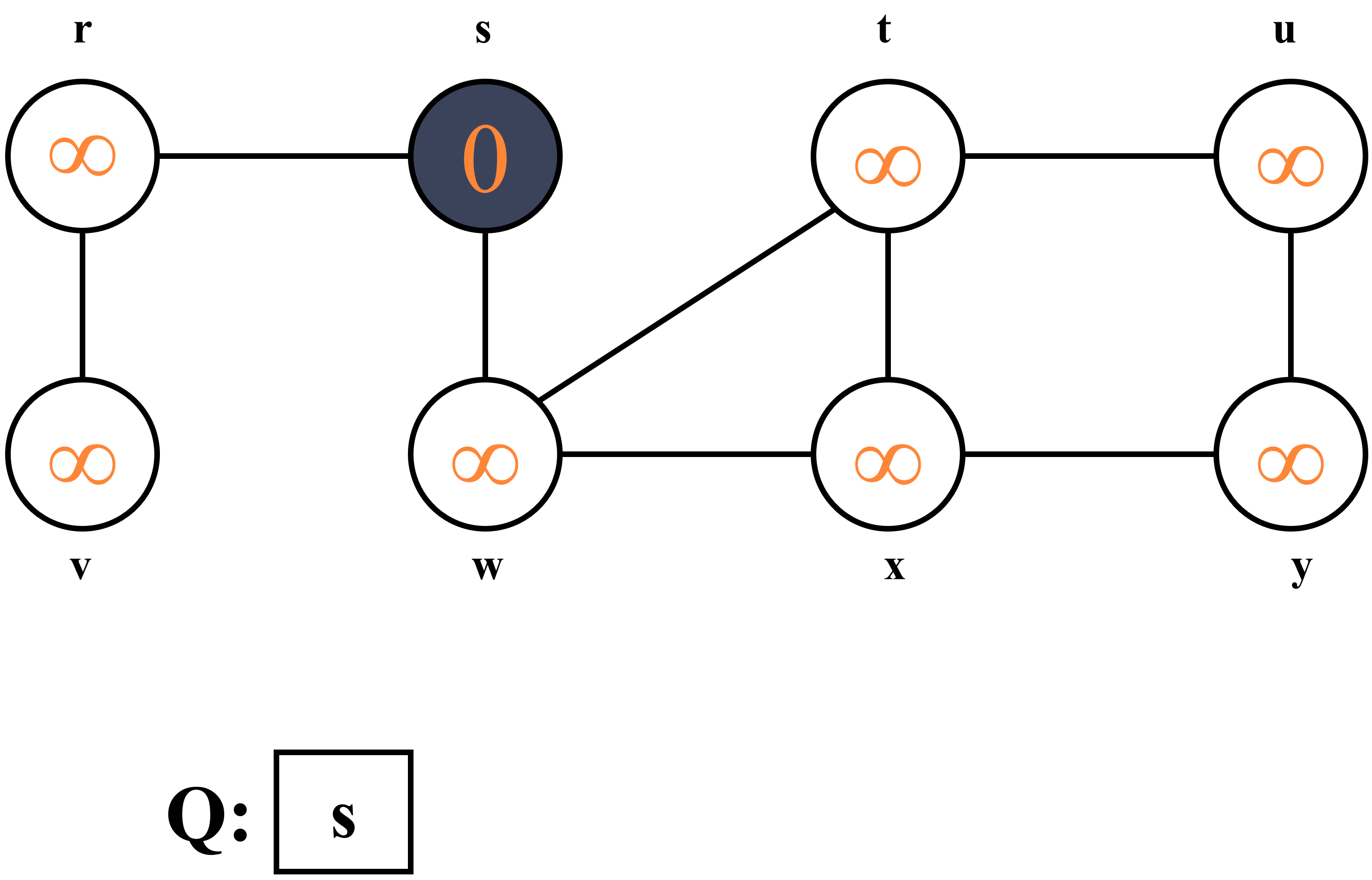
white: undiscovered
gray: discovered
black: finished

Q: a queue of discovered vertices
color[v]: color of v
d[v]: distance from s to v
$\pi[u]$: predecessor of v

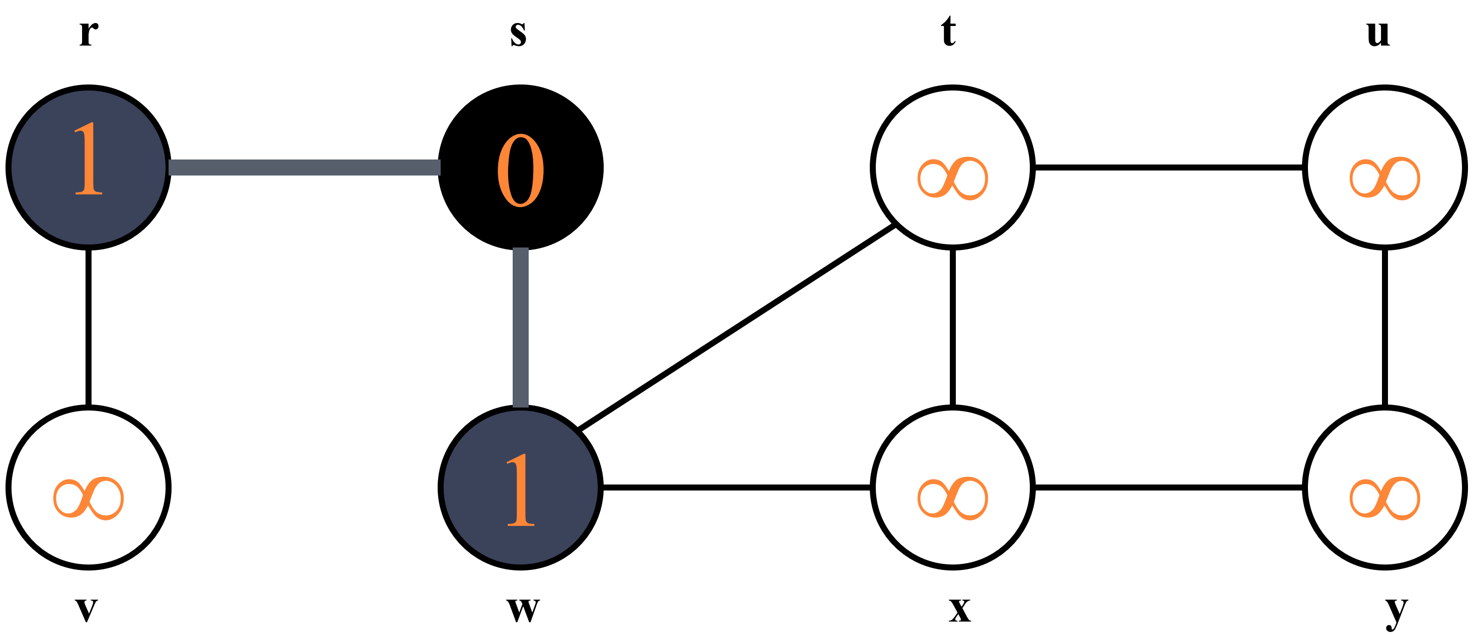
Breadth-First Search: Example



Breadth-First Search: Example

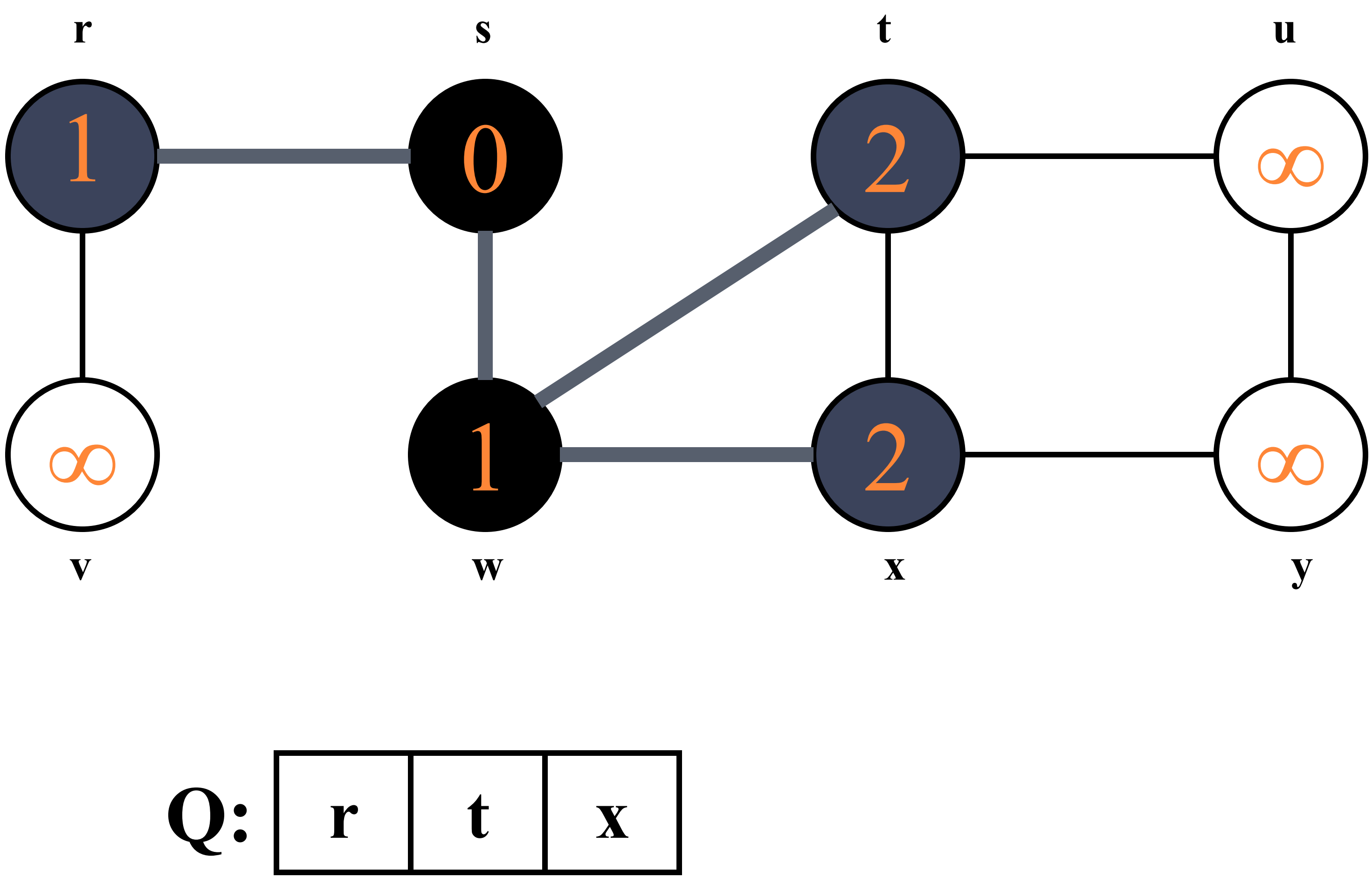


Breadth-First Search: Example

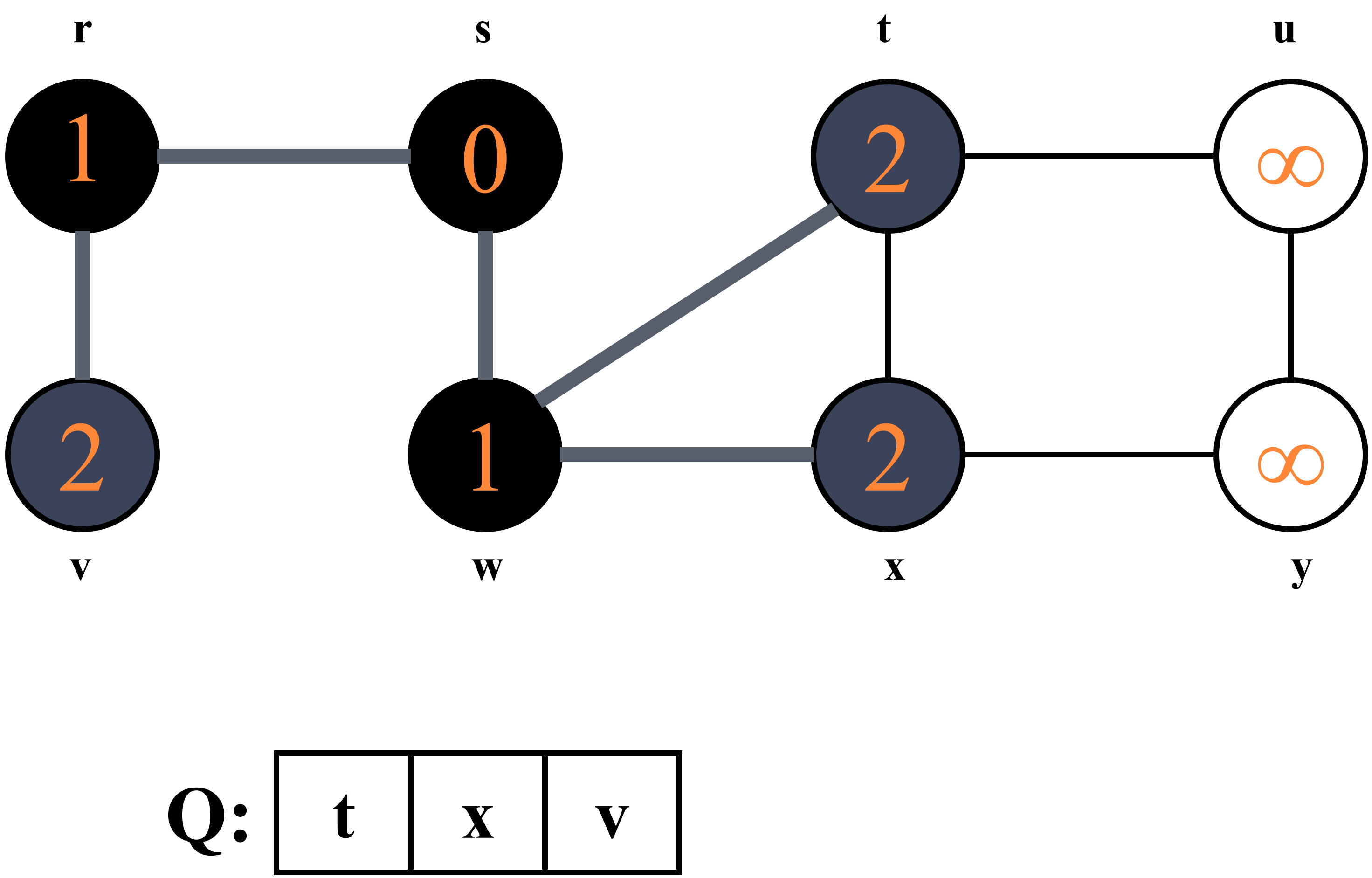


Q: | w | r |

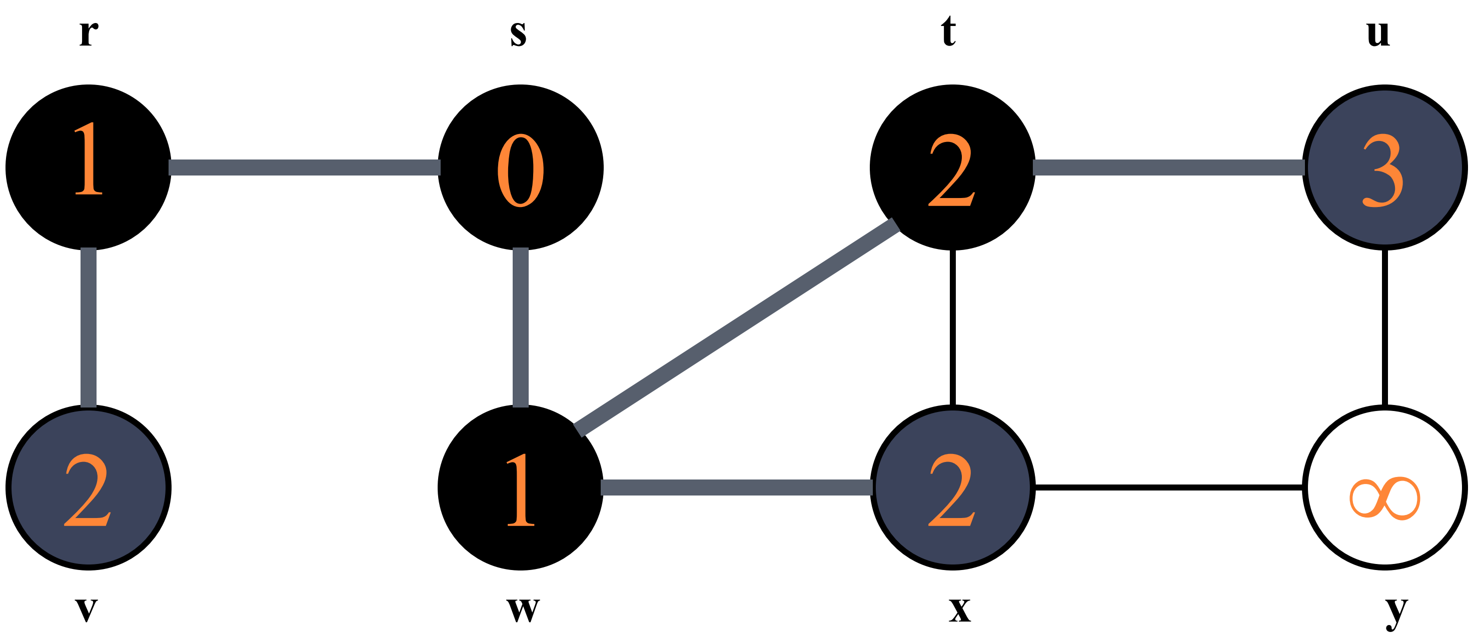
Breadth-First Search: Example



Breadth-First Search: Example



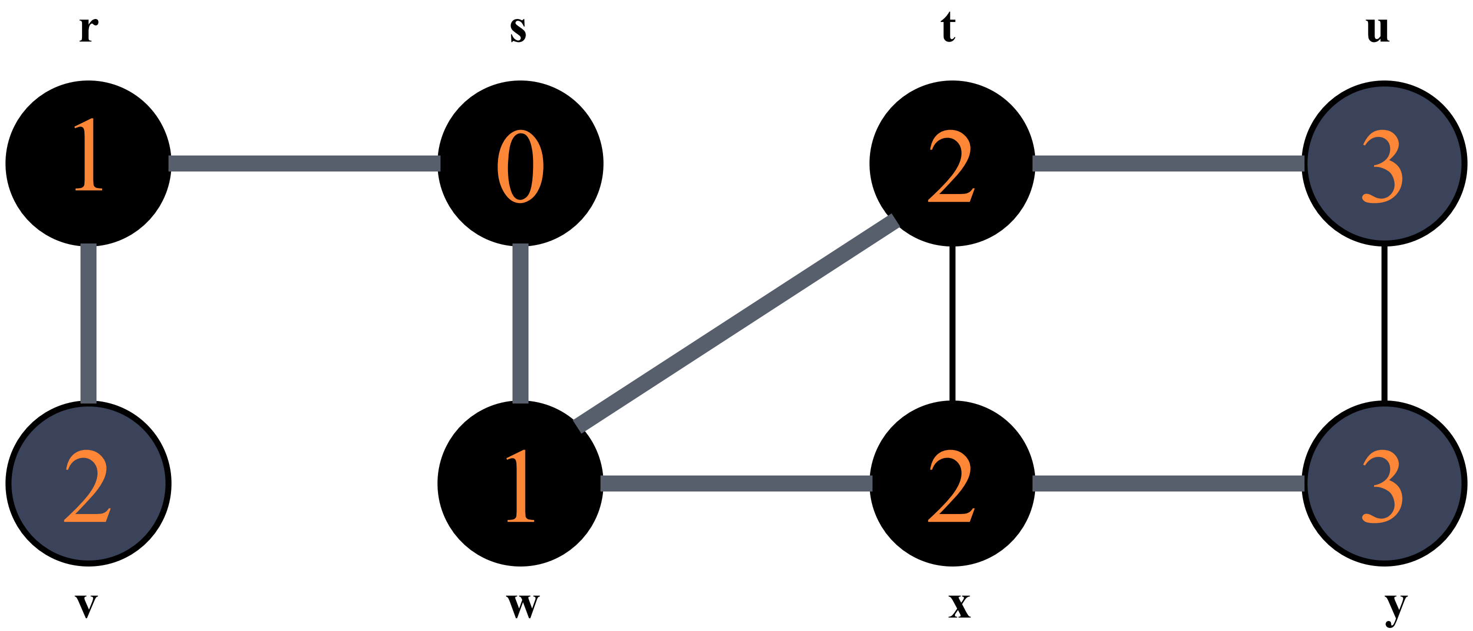
Breadth-First Search: Example



Q:

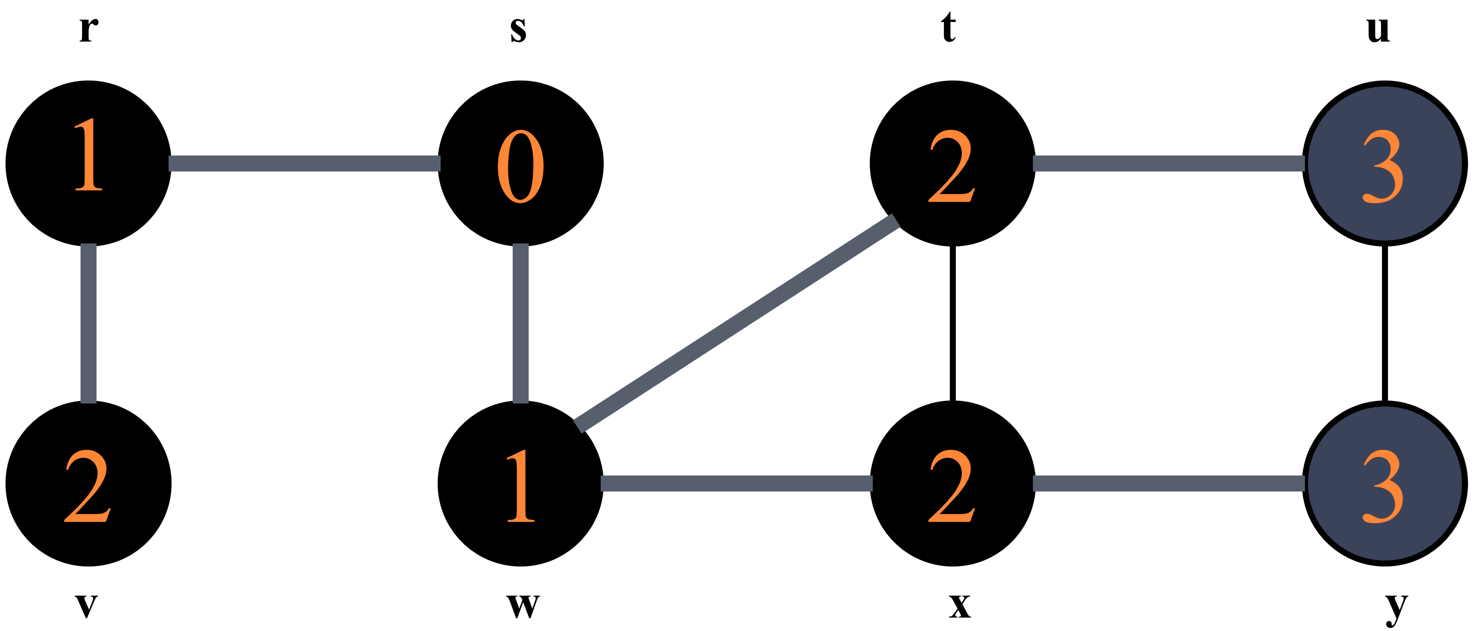
x	v	u

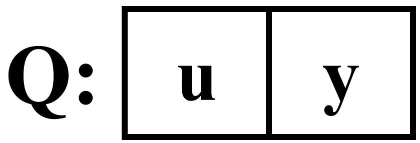
Breadth-First Search: Example



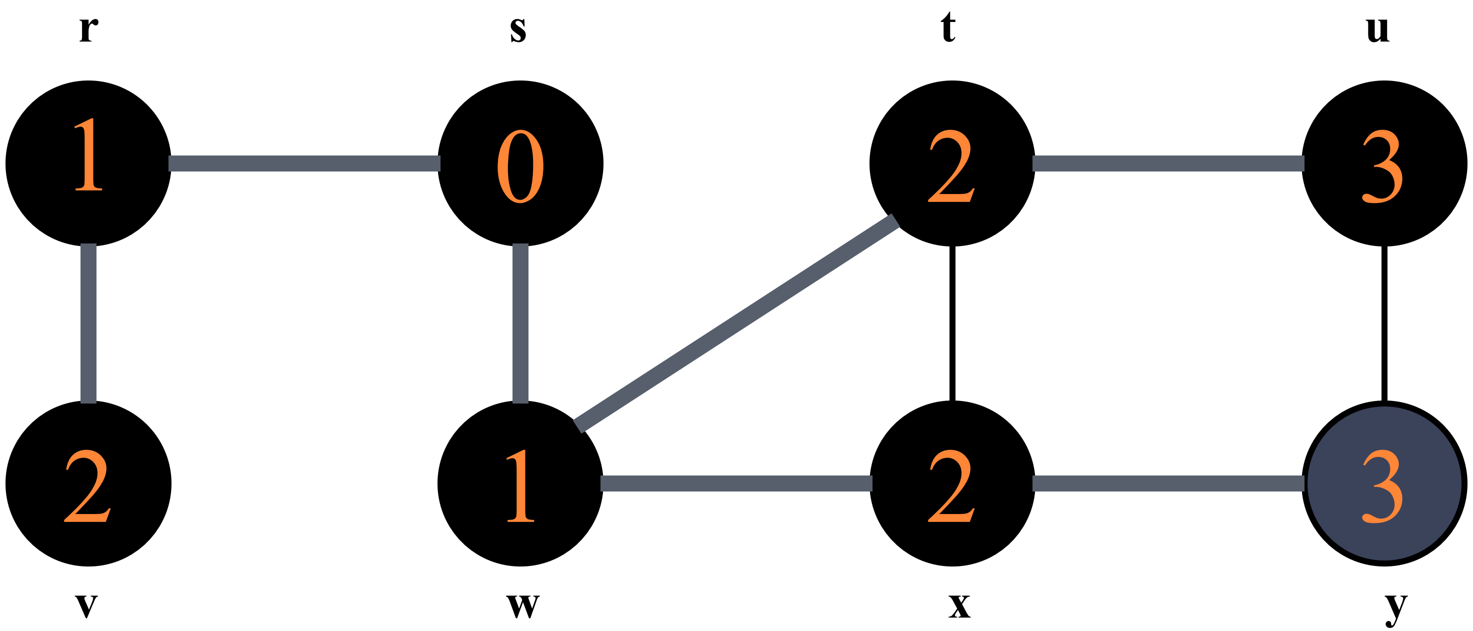
Q:	v	u	y

Breadth-First Search: Example



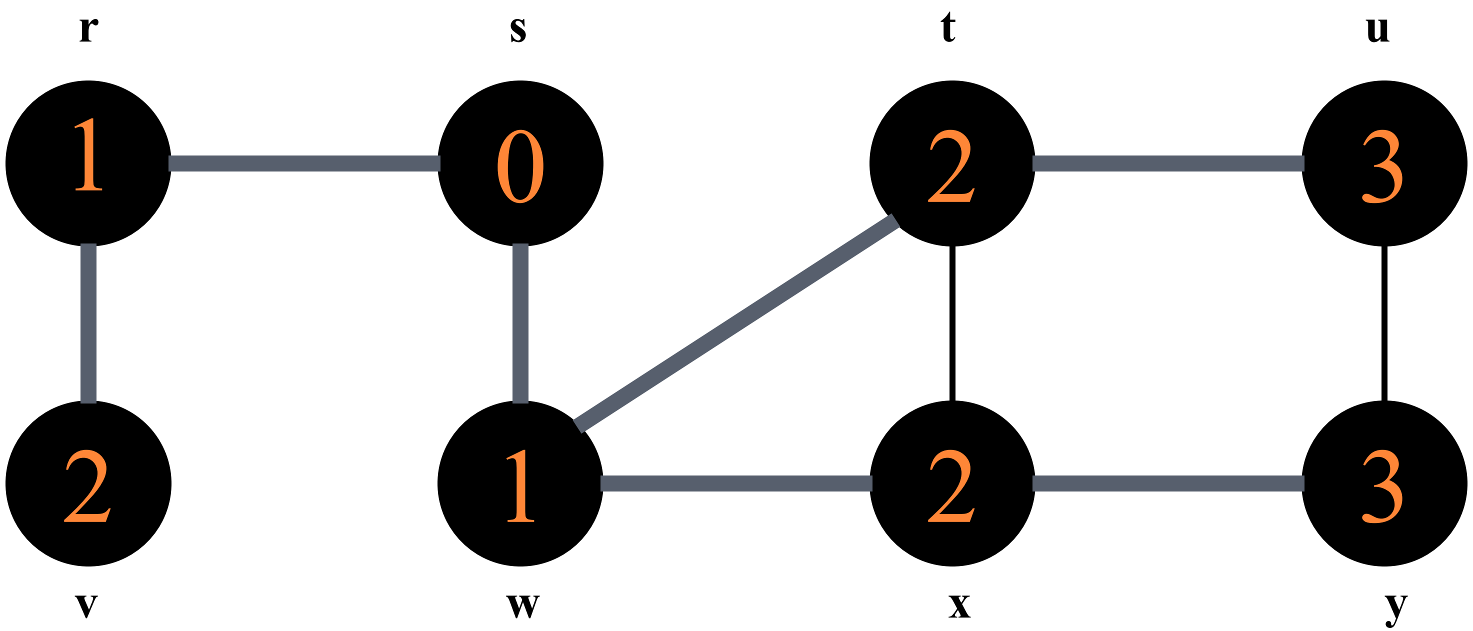


Breadth-First Search: Example



Q: $\boxed{\text{y}}$

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 $O(|V| + |E|)$

Breadth-first Tree

- For a graph $G = (V, E)$ with source s, the **predecessor subgraph** of G is $G_\pi = (V_\pi, E_\pi)$ where
  - $V_\pi = \{v \in V : \pi[v] \neq nil\}$ □ $\{s\}$
  - $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$
- The predecessor subgraph G_π is a **breadth-first tree** if:
  - V_π consists of the vertices reachable from s and
  - for all $v \in V_\pi$, there is a unique simple path from s to v in G_π that is also a shortest path from s to v in G.
- The edges in E_π are called **tree edges**. $|E_\pi| = |V_\pi| - 1$.

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."
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

DFS - 1

```
parent = {s:none}
DFS_Visit(s, Adj.s)
    for v in Adj[s]
        if v is not in parent
            parent[v] =s
            DFS_Visit(v, Adj.v)
```

```
DFS (V, Adj)
    parent ={}
    for s in V
        if s is not in parent
            parent[s] = none
            DFS_Visit(s,Adj.s)
```

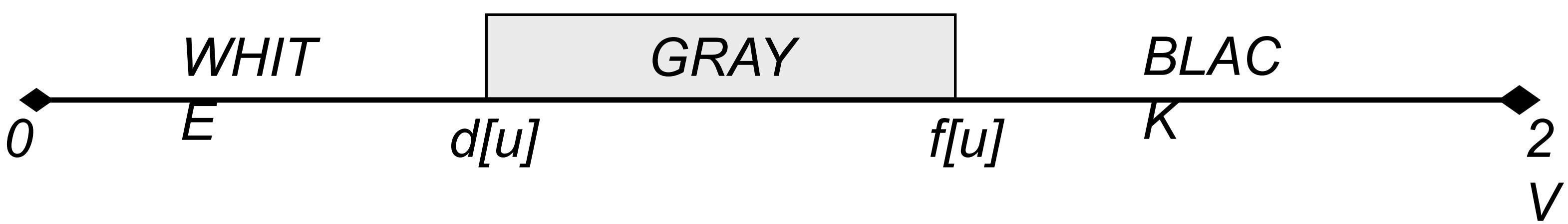
Depth-first Search

- **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

Depth-first Search

- Coloring scheme for vertices as BFS.
  - A vertex is "discovered" the first time it is encountered during the search.
  - A vertex is "finished" if it is a leaf node or all vertices adjacent to it have been finished.
  - White before discovery, gray while processing and black when finished processing

$$1 \le d[u] < f[u] \le 2|V|$$



Pseudocode

DFS(*G*)

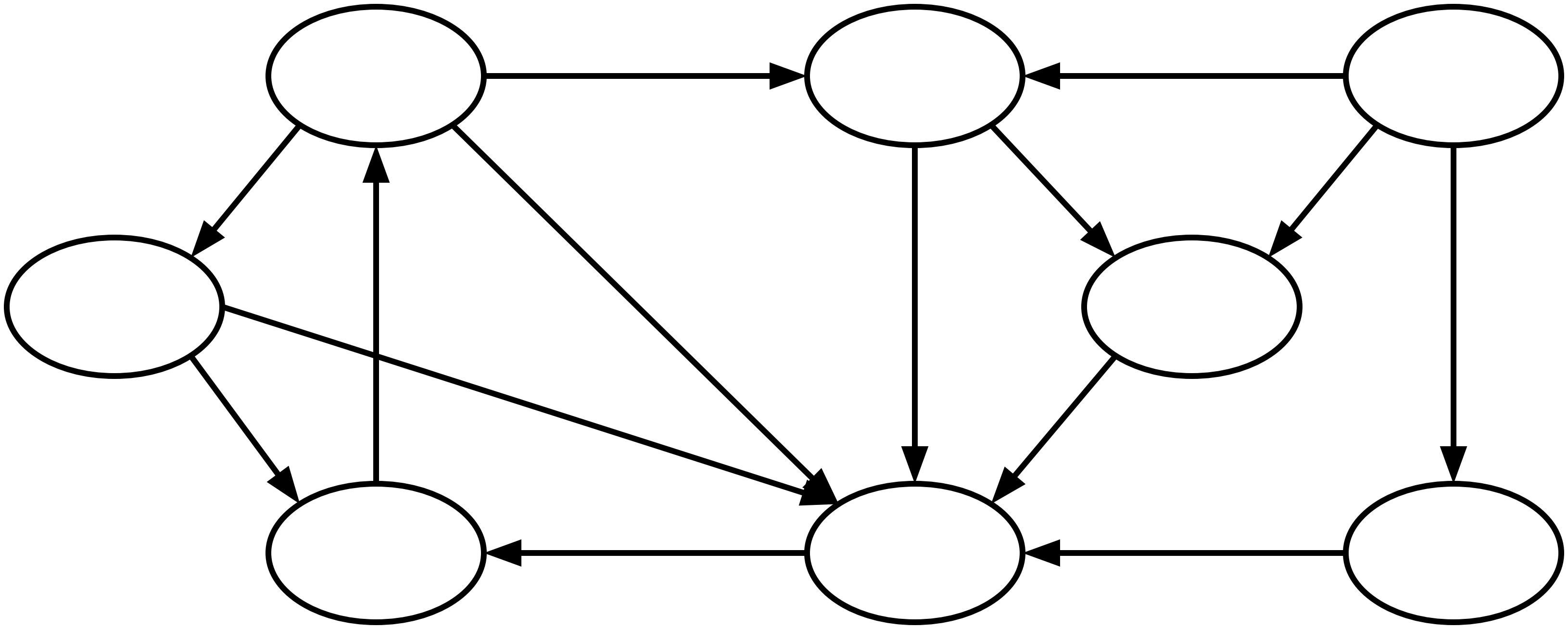
```
for each vertex u ∈ V[G]
    do color[u] ← white
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = white
         then DFS-Visit(u)
```

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

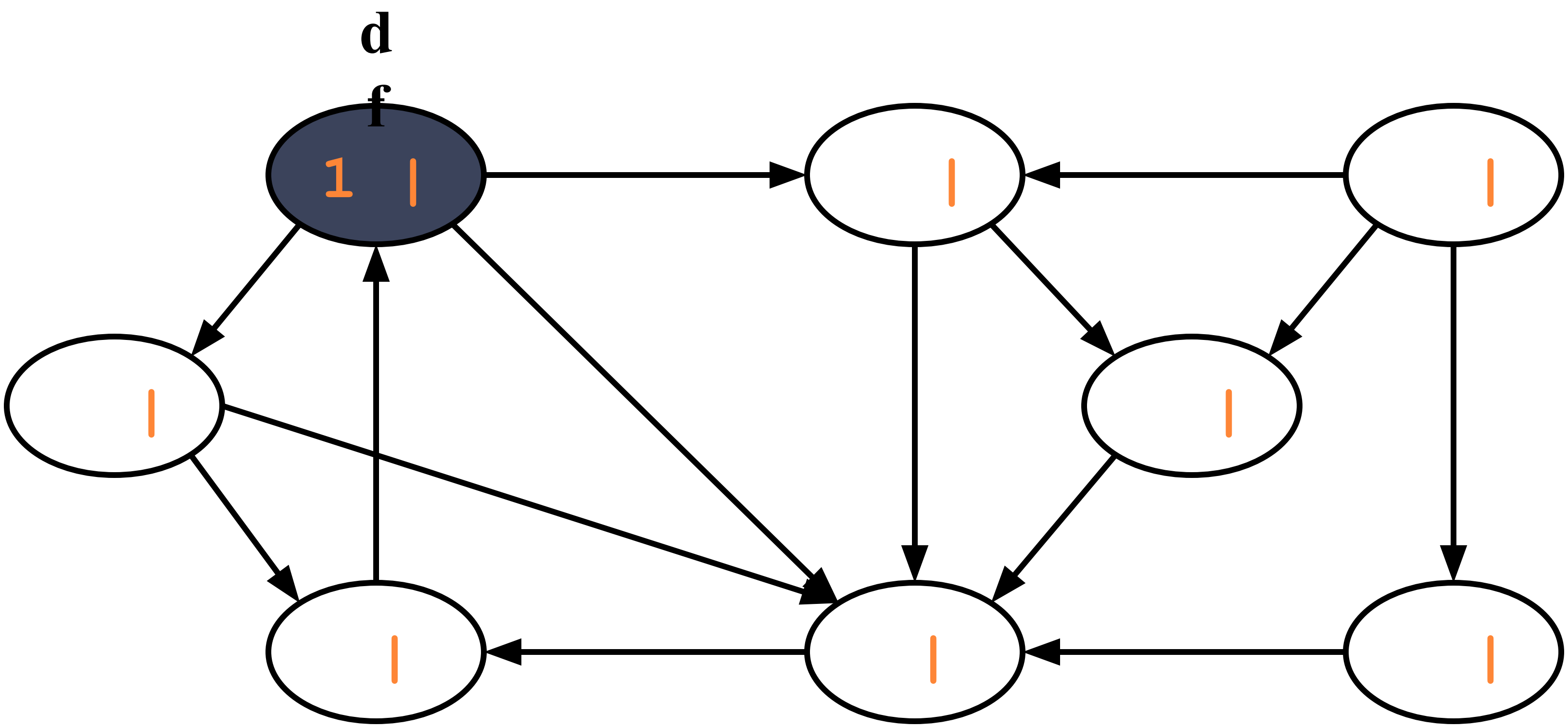
DFS-Visit(*u*)

```
color[u] ← GRAY  // White vertex u
                    has been discovered
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
         then π[v] ← u
              DFS-Visit(v)
color[u] ← BLACK    // Blacken u;
                       it is finished.
f[u] ← time ← time + 1
```

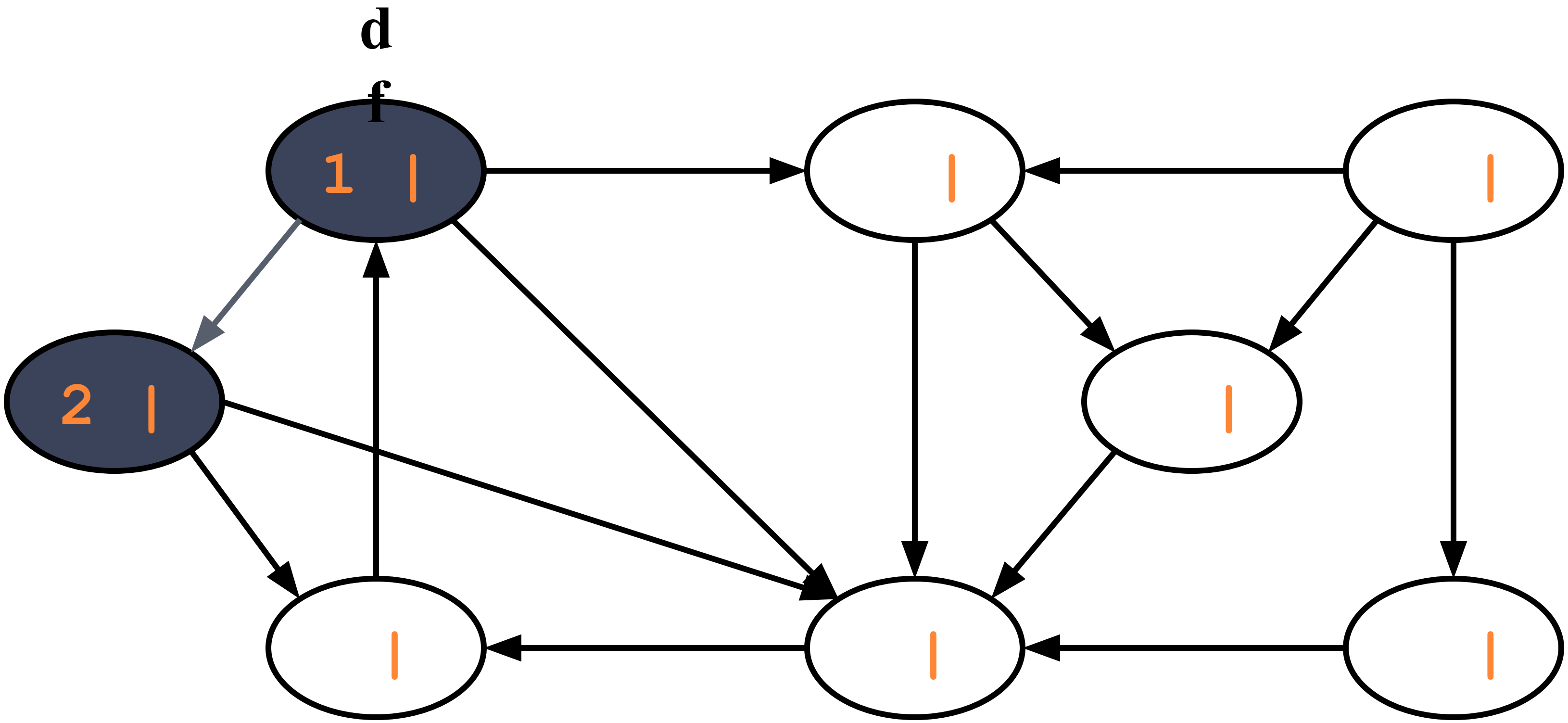
DFS Example



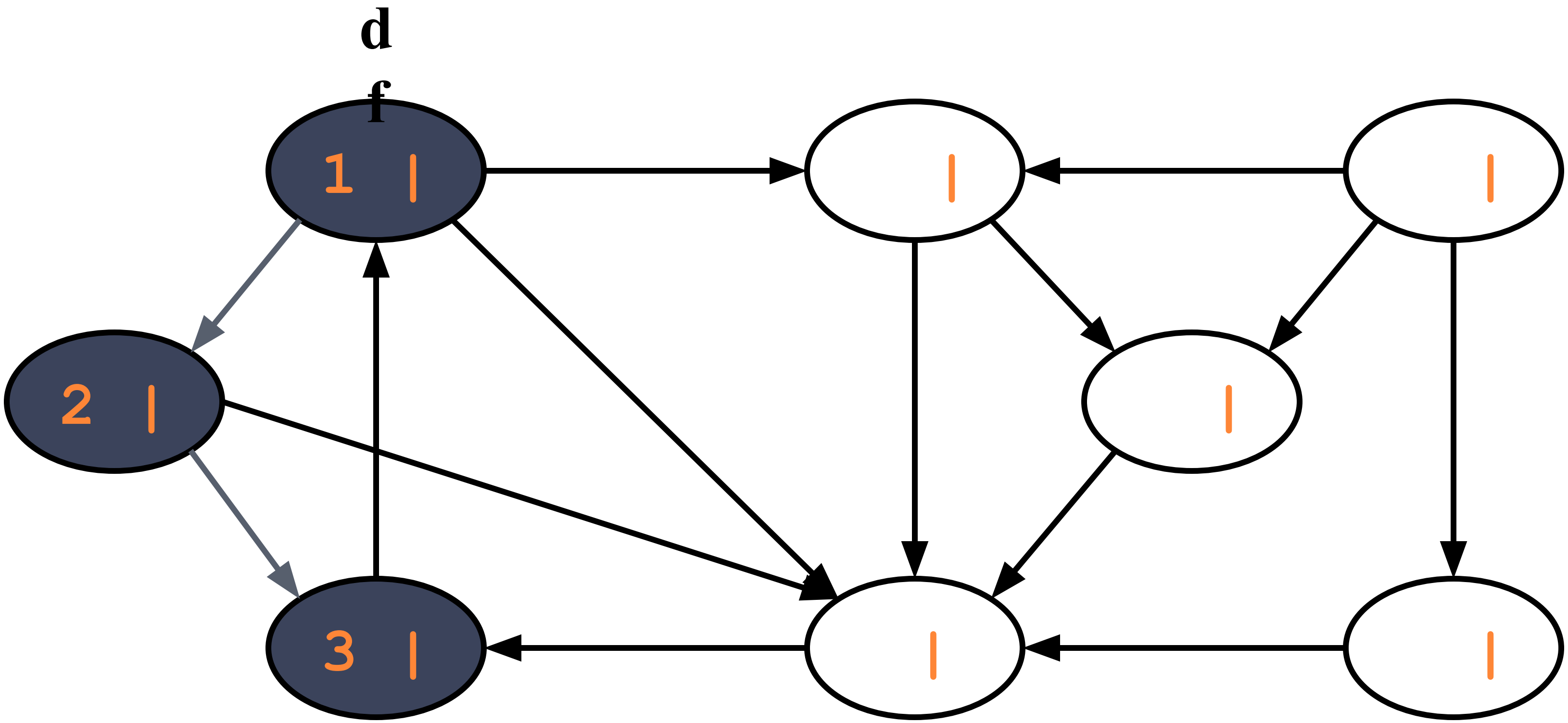
DFS Example



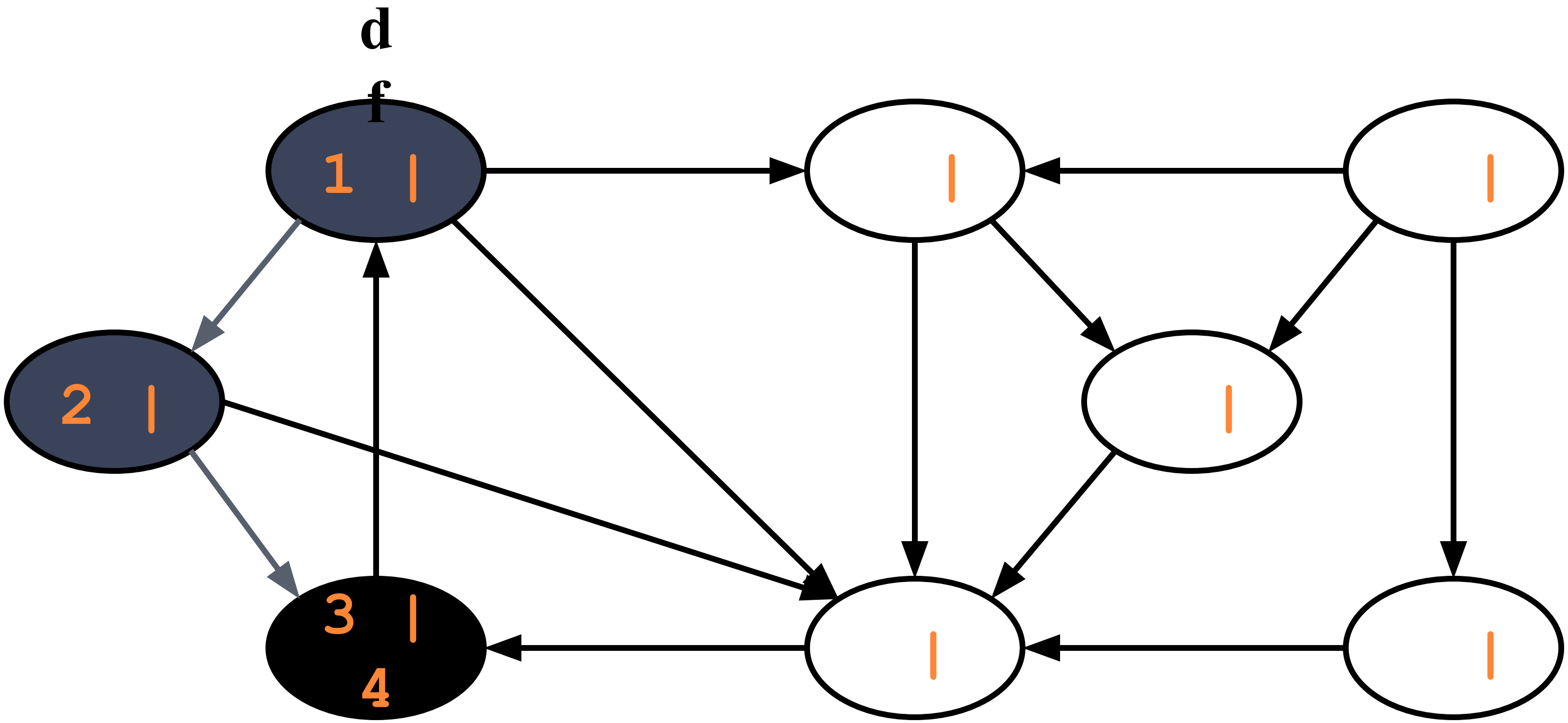
DFS Example



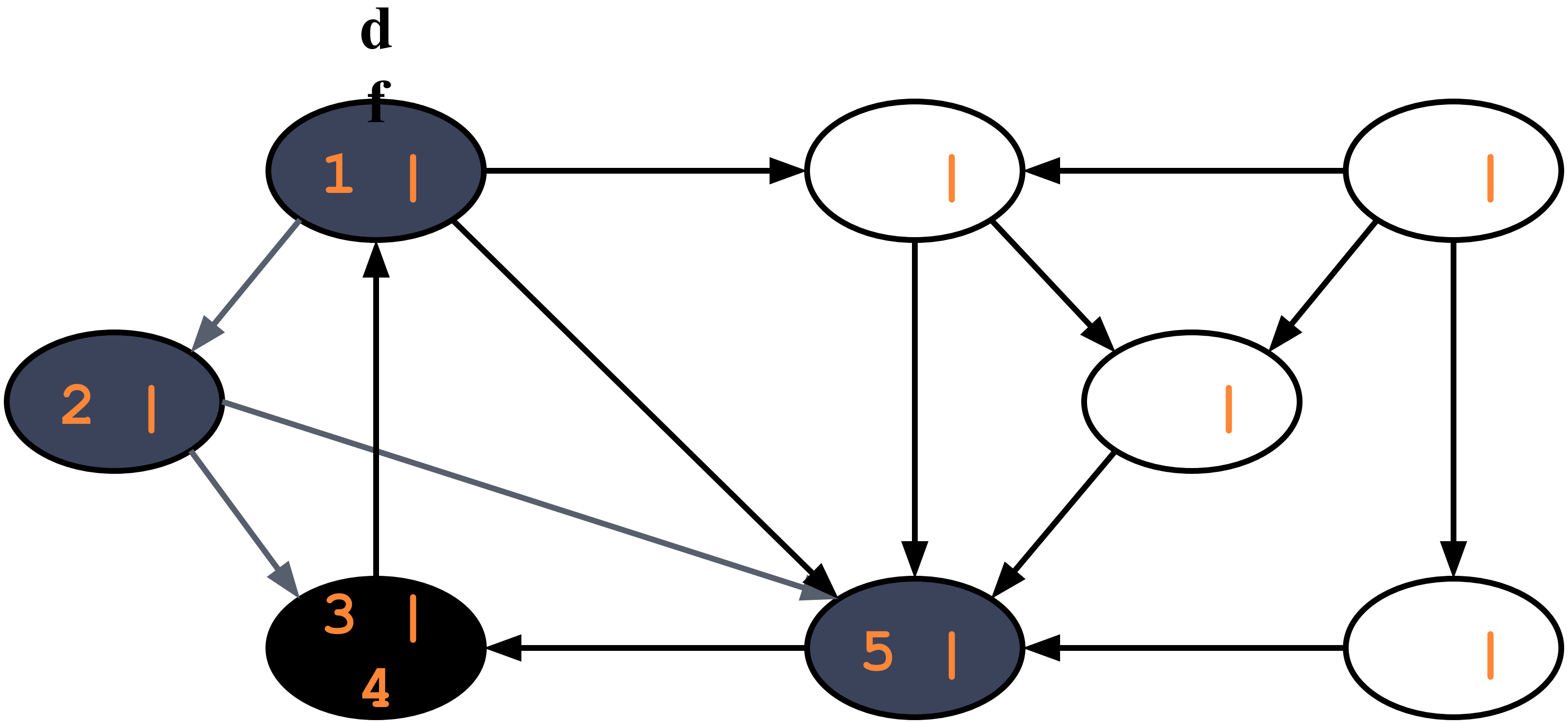
DFS Example



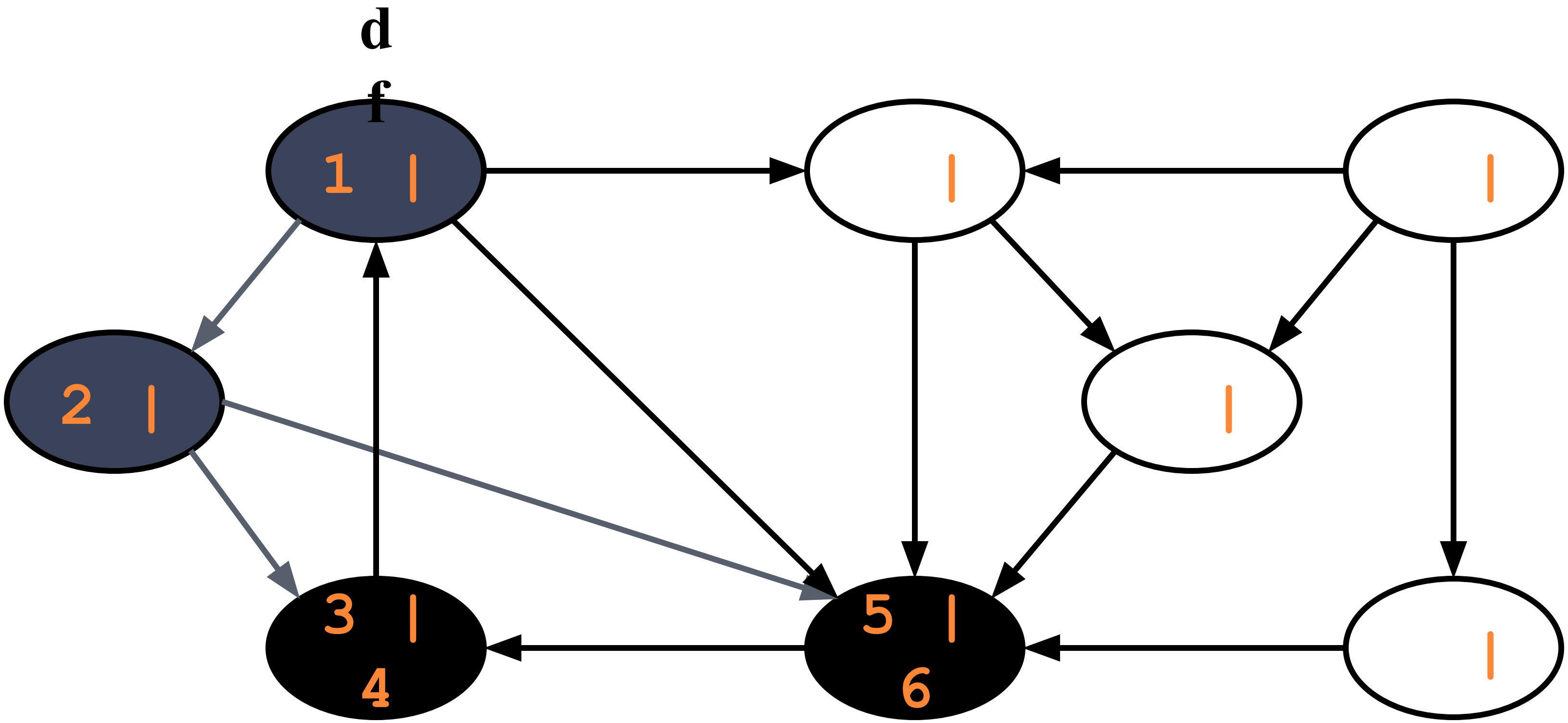
DFS Example



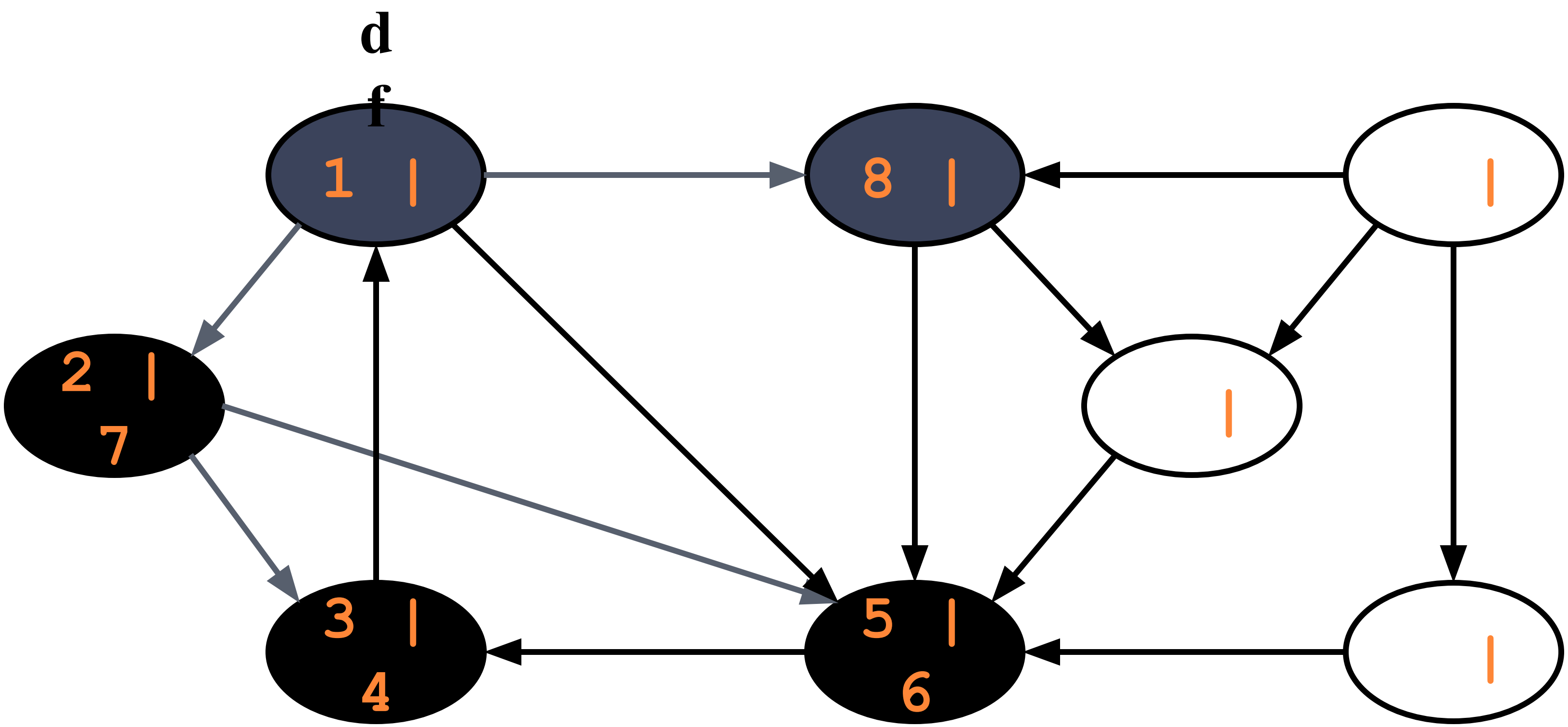
DFS Example



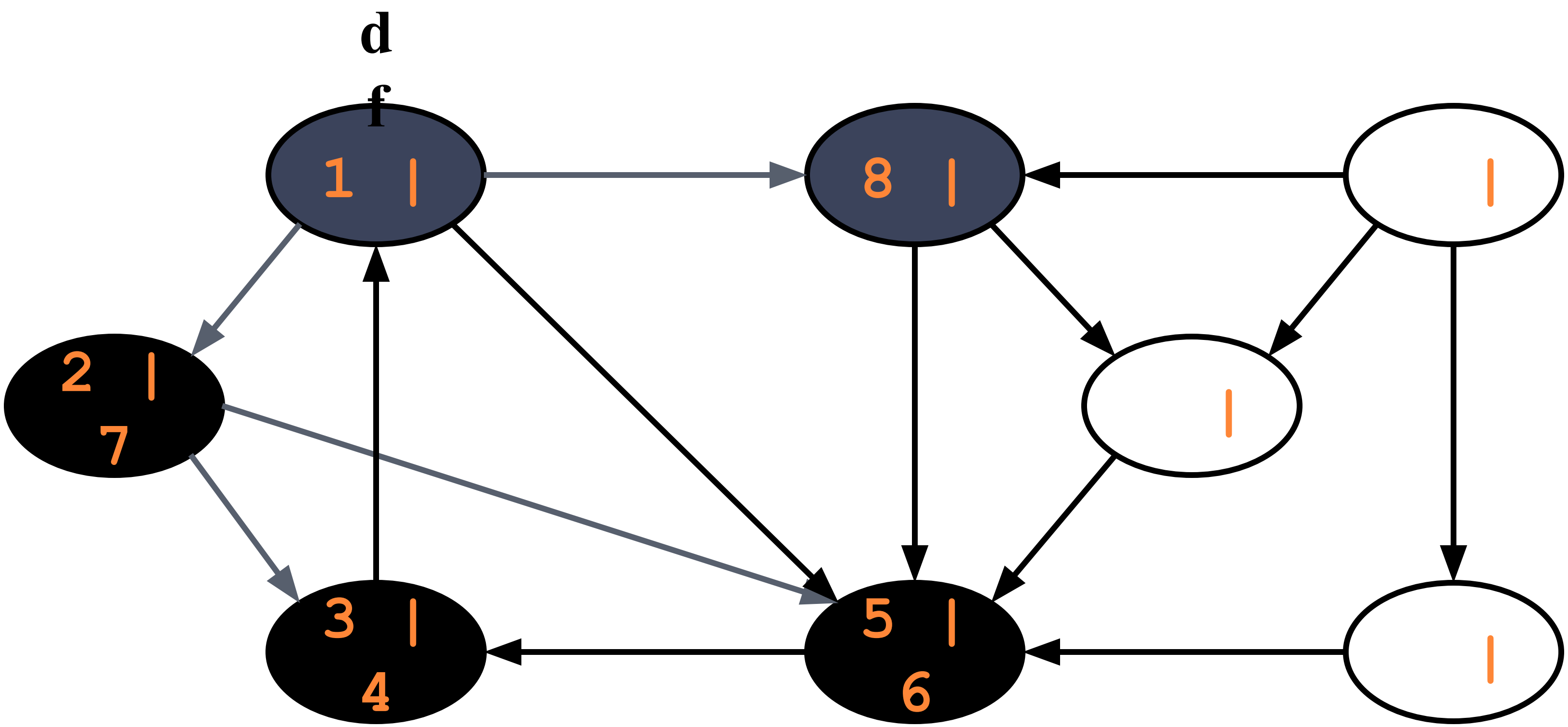
DFS Example



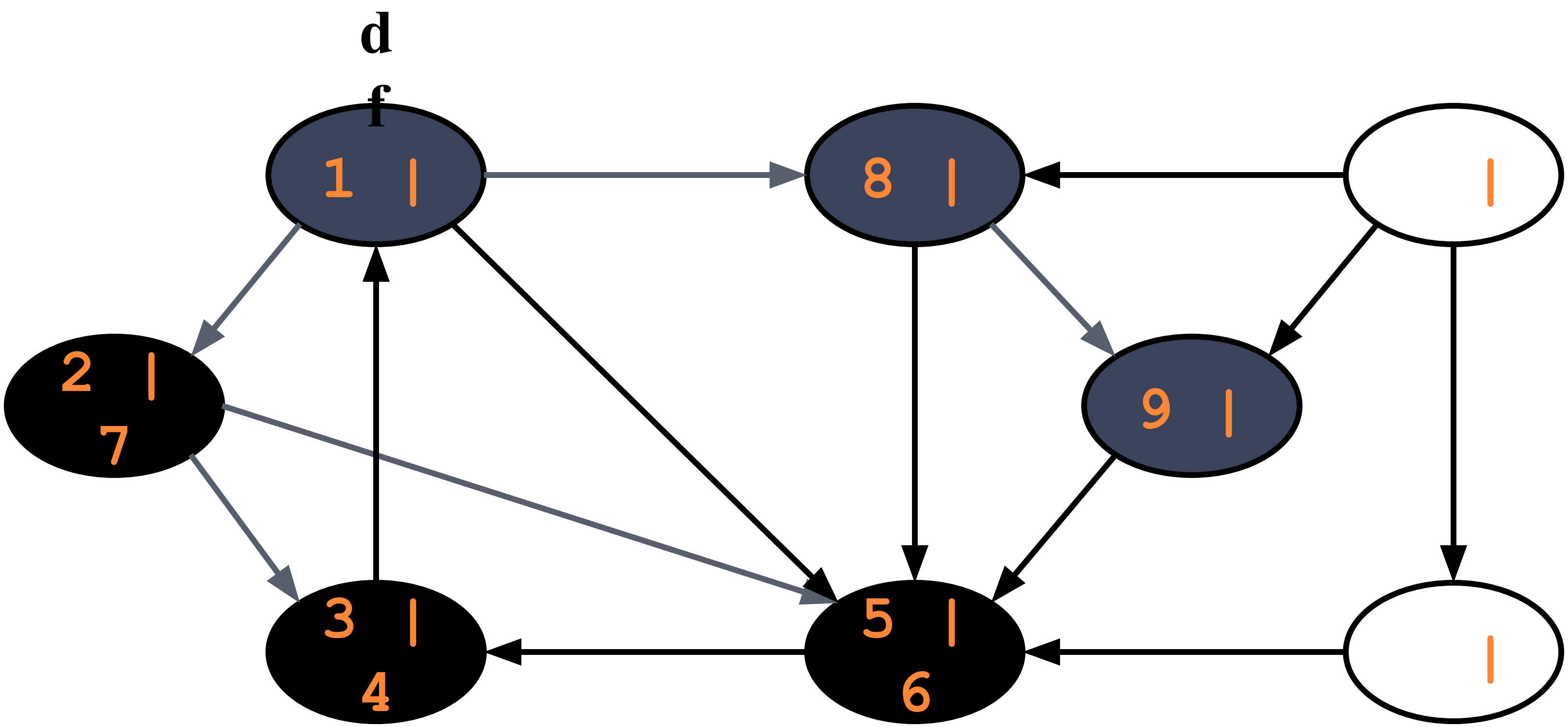
DFS Example



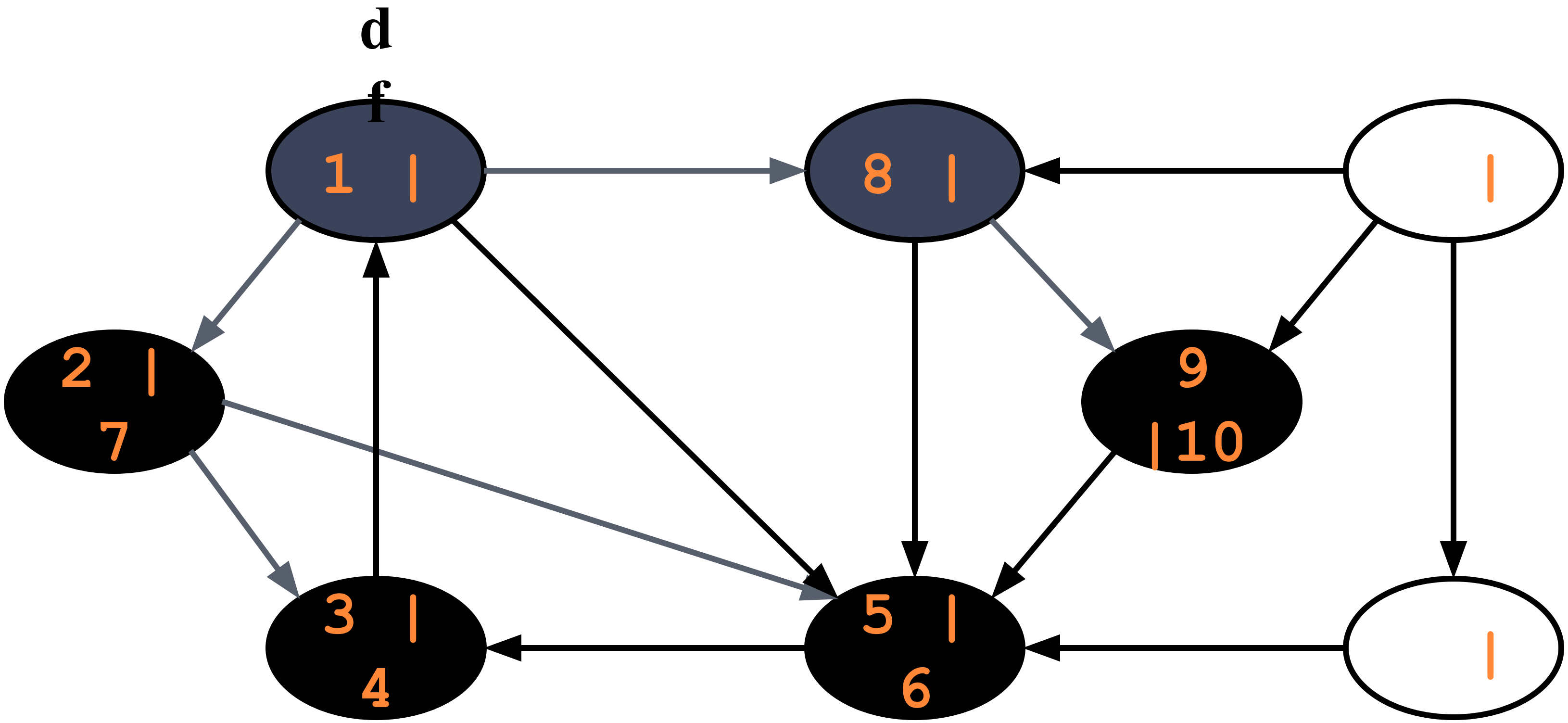
DFS Example



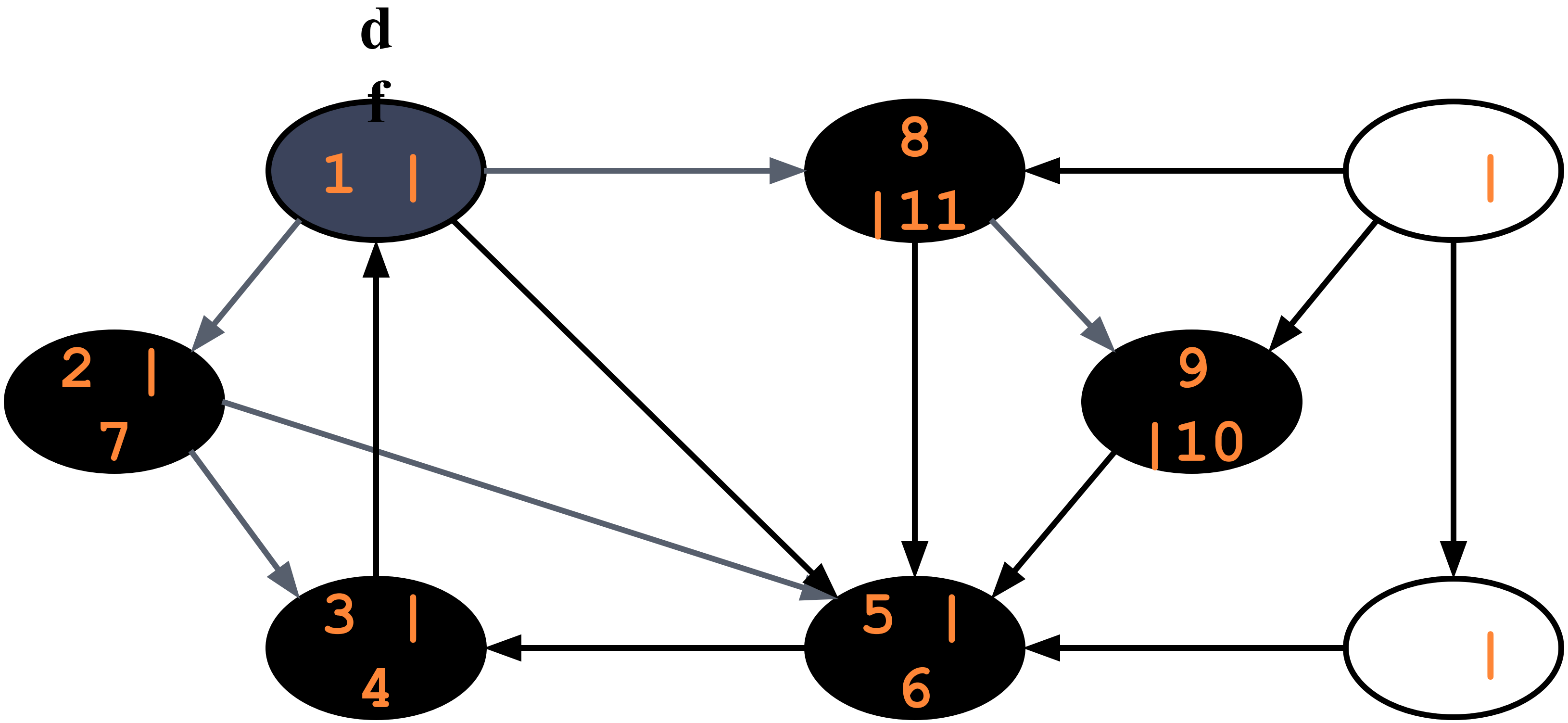
DFS Example



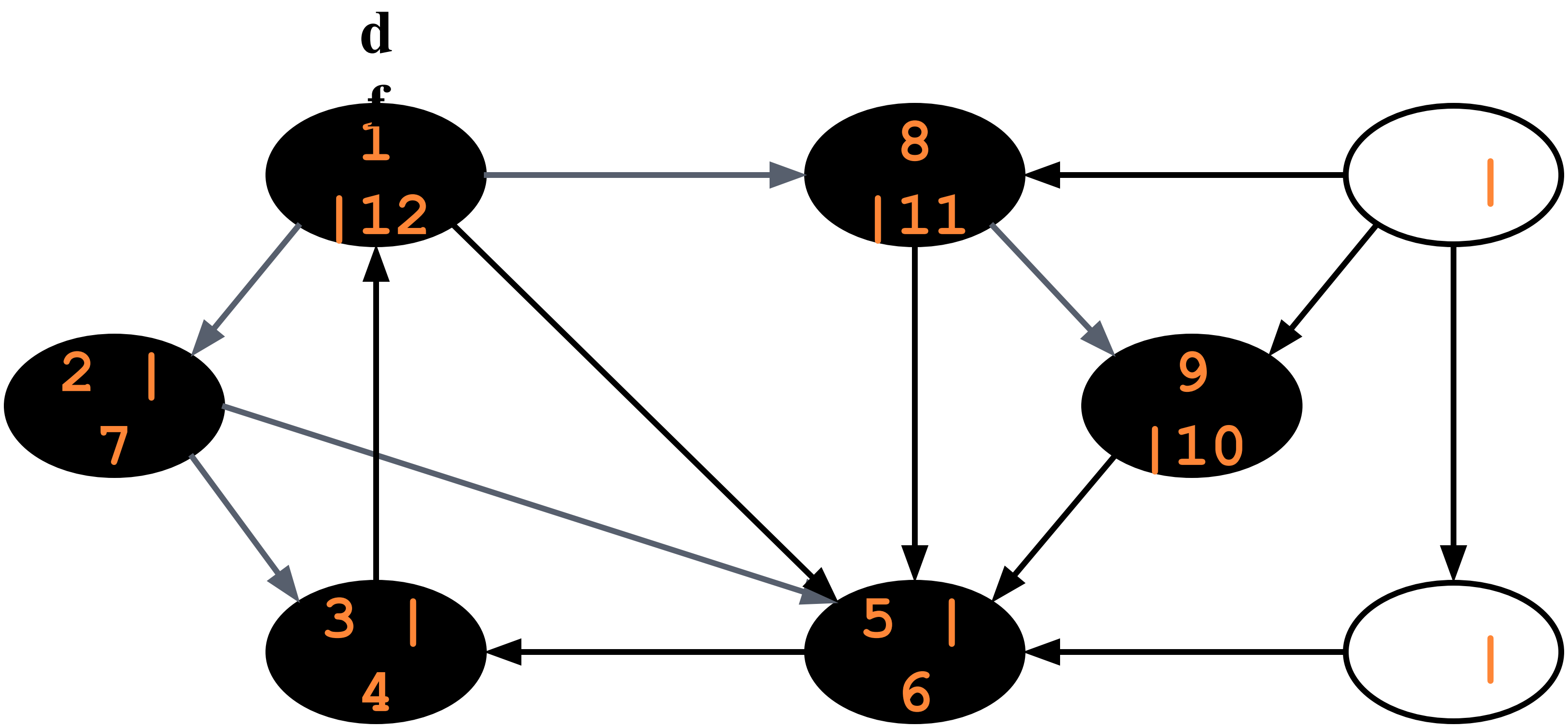
DFS Example



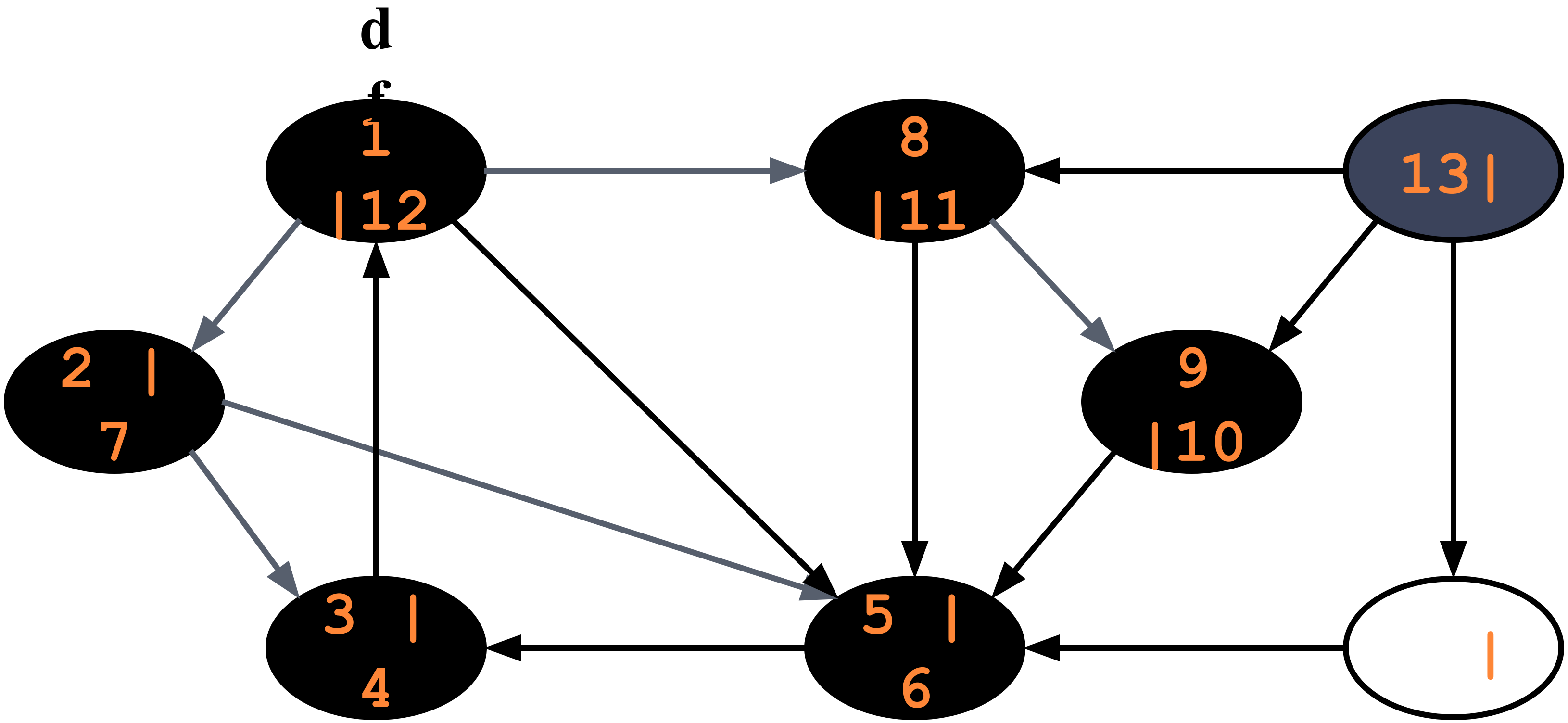
DFS Example



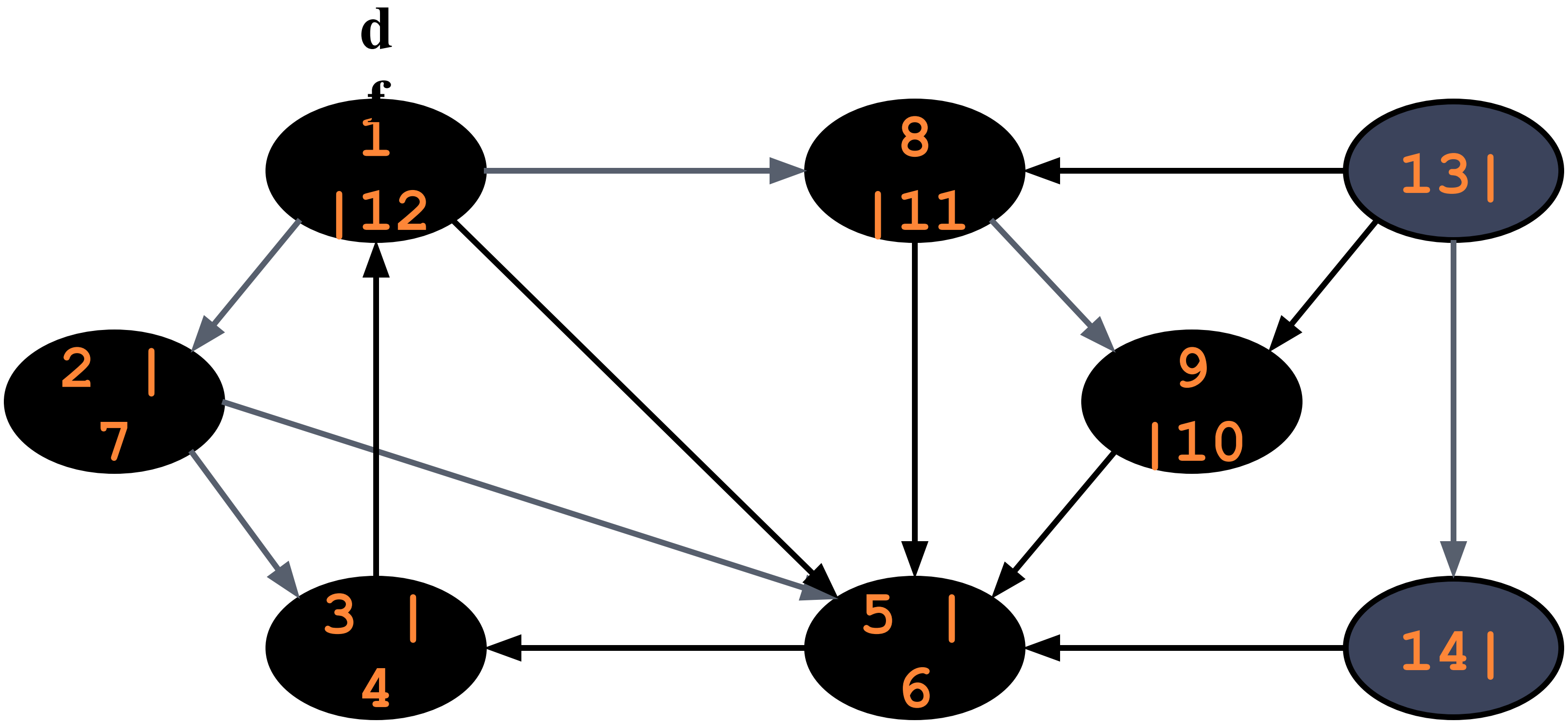
DFS Example



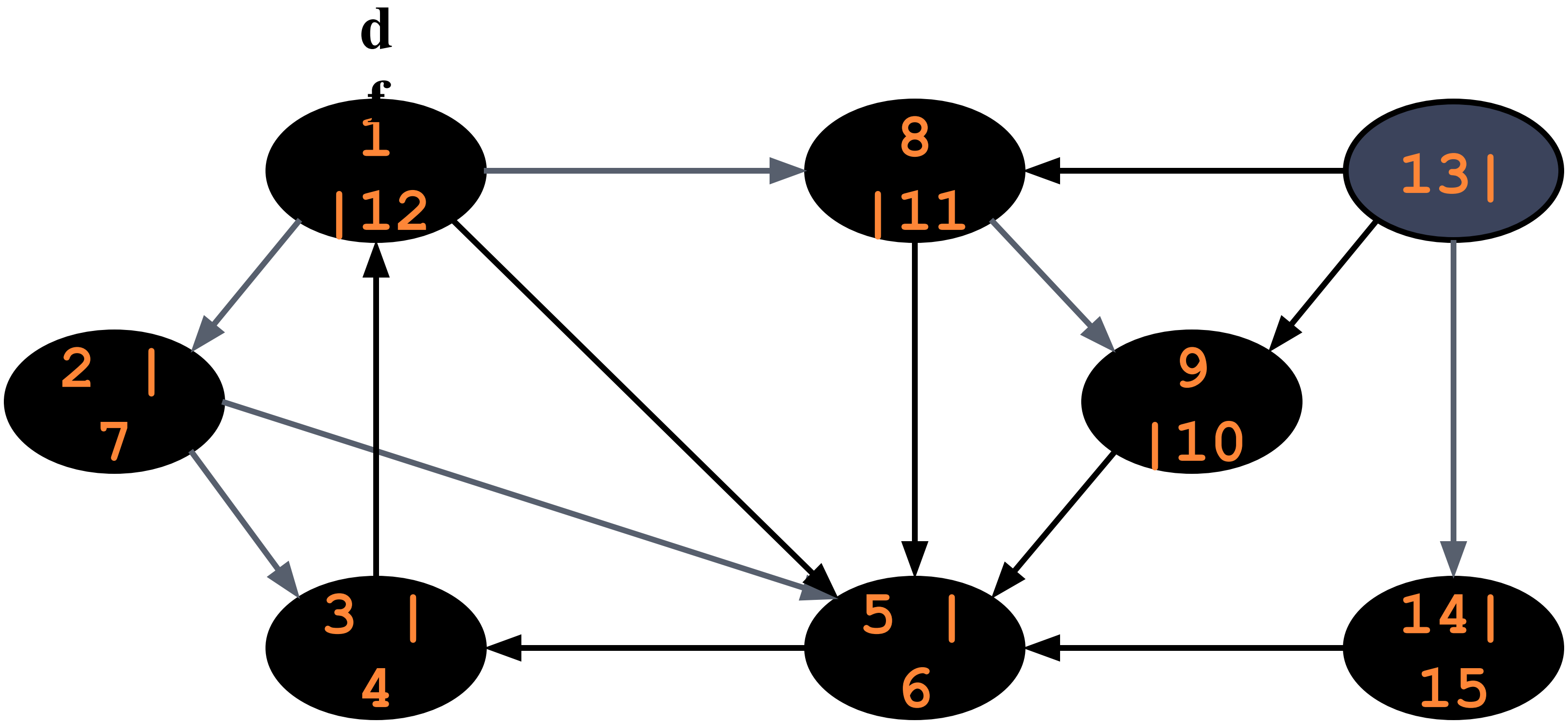
DFS Example



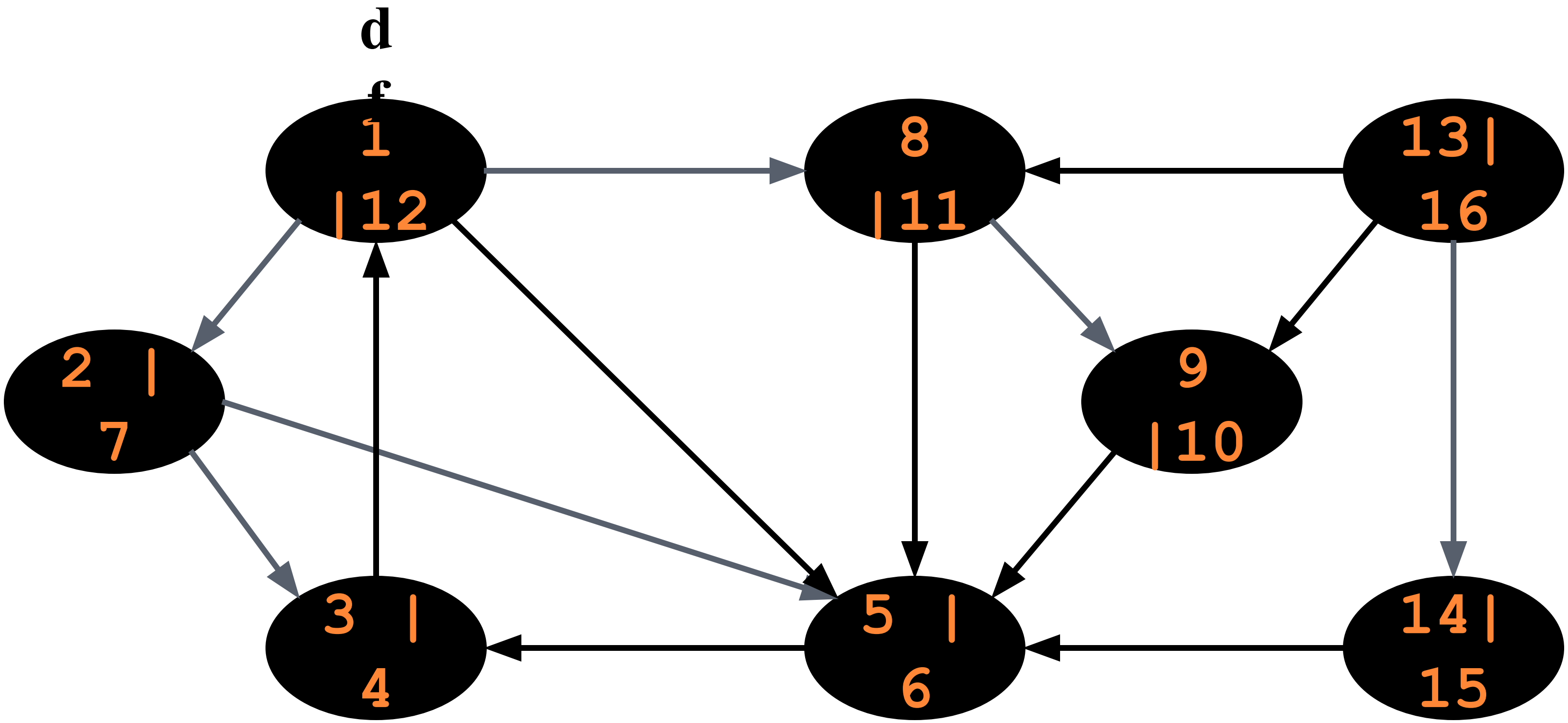
DFS Example



DFS Example



DFS Example



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 $\Theta(|V| + |E|)$.

Depth-First Trees

- Predecessor subgraph defined slightly different from that of BFS.
- The predecessor subgraph of DFS is $G_\pi = (V, E_\pi)$ where $E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq nil\}$.
  - How does it differ from that of BFS?
  - The predecessor subgraph G_π forms a *depth-first forest* composed of several *depth-first trees*. The edges in E_π are called *tree edges*.

Time-Stamp Structure in DFS

- There is also a nice structure to the time stamps, which is referred to as *Parenthesis Structure*.

Theorem 22.7

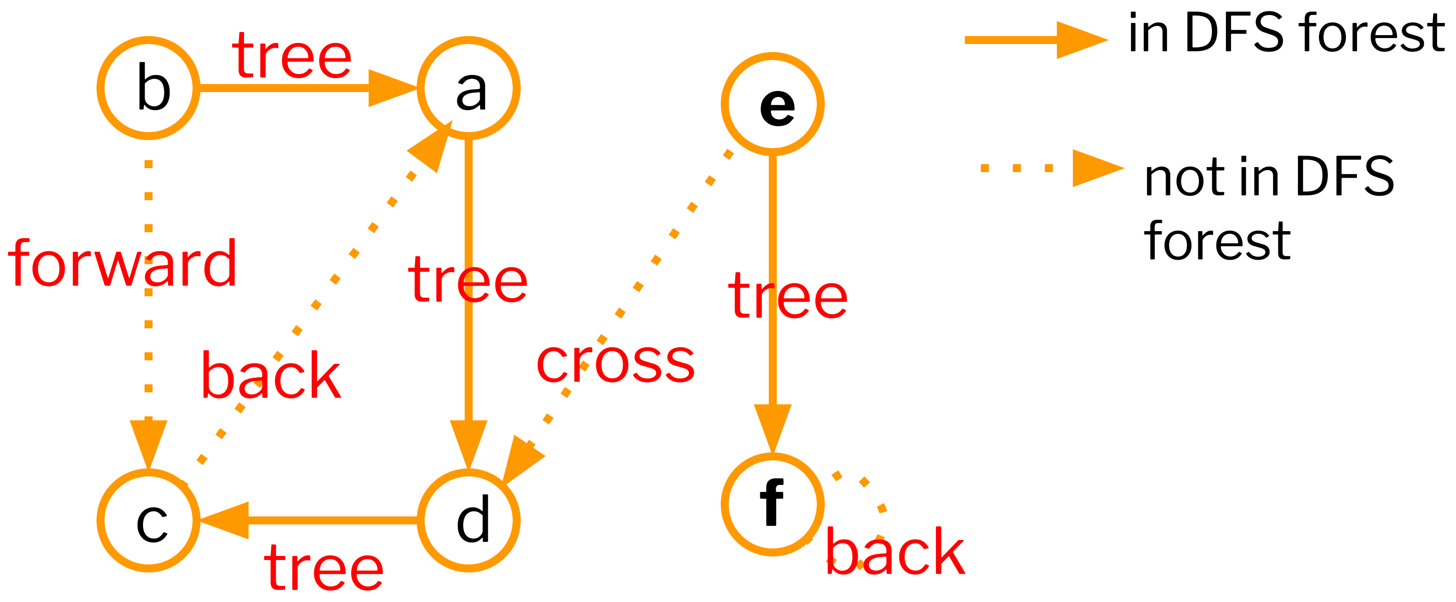
For all u, v, exactly one of the following holds:

1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither u nor v is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u.
3. $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v.

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:
  - A **tree edge** is an edge in a DFS-tree.
  - A **back edge** connects a vertex to an ancestor in a DFS-tree. Note that a self-loop is a back edge.
  - A **forward edge** is a non-tree edge that connects a vertex to a descendent in a DFS-tree.
  - A **cross edge** is any other edge in graph G. It connects vertices in two different DFS-tree or two vertices in the same DFS-tree neither of which is the ancestor of the other.

Example of Classifying Edges



Classifying edges of a digraph

- (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 discovered after v)

DFS: Kinds of edges

DFS-Visit(*u*) ▷ with edge classification. G must be a directed graph

```
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
```

DFS: Kinds of edges

- Suppose G be an undirected graph, then we have following edge classification:
  - **Tree Edge** an edge connects a vertex with its parent.
  - **Back Edge** a non-tree edge connects a vertex with an ancestor.
  - **Forward Edge** There is no forward edges because they become back edges when considered in the opposite direction.
  - **Cross Edge** There cannot be any cross edge because every edge of G must connect an ancestor with a descendant.

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

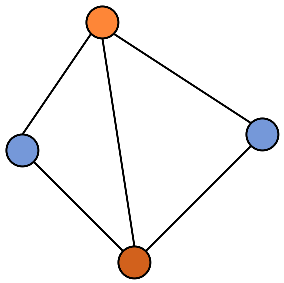
- Graph $G = (V,E)$ is **bipartite** iff V can be partitioned into two sets of nodes A and B such that each edge has one end in A and the other end in B

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

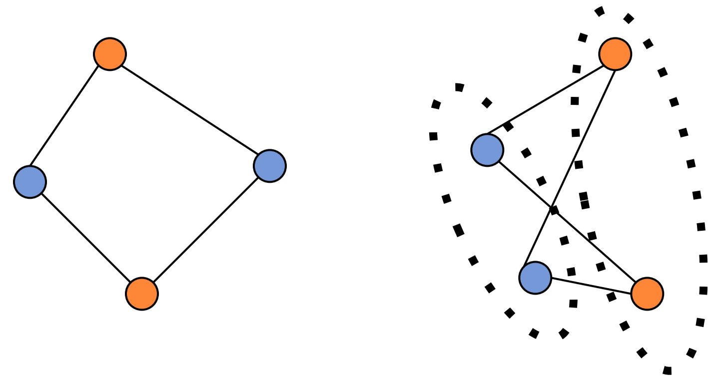
Note: graphs without cycles (trees) are bipartite

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

d[u] ← time ← time + 1

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

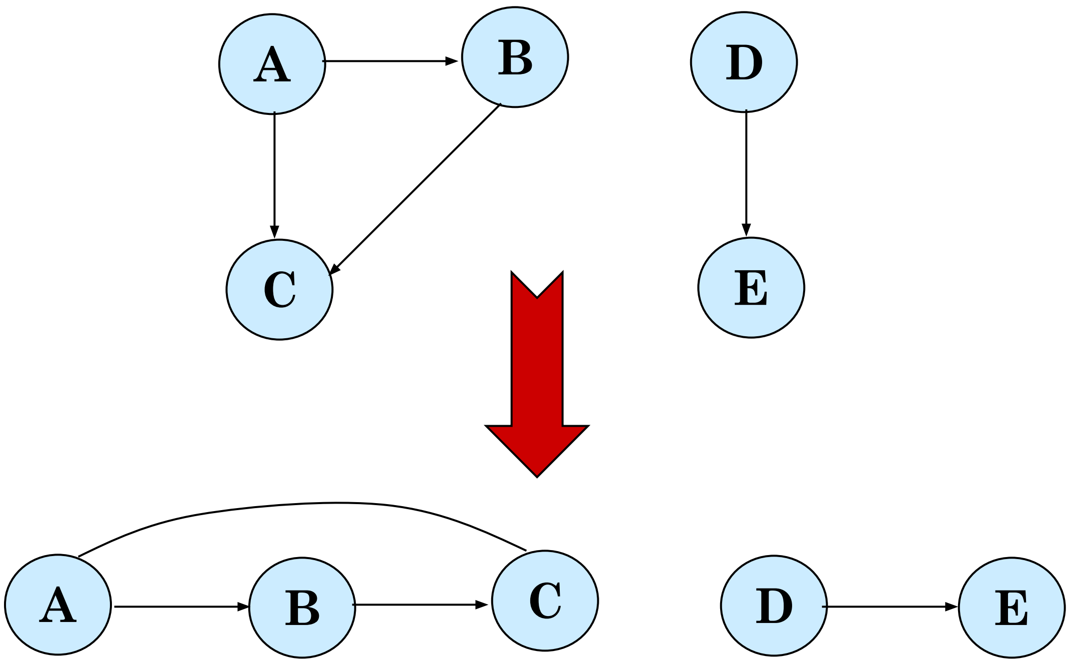
f[u] ← time ← time + 1

Application of DFS: Detecting Cycle for Undirected Graph

```
DFS_visit(u)
color(u) ← GRAY
d[u] ← time ← time + 1
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
f[u] ← time ← time + 1
```

Topological Sort

Want to “sort” a directed acyclic graph (DAG).



Think of original DAG as a **partial order**.

Want a **total order** that extends this partial order.

Topological Sort

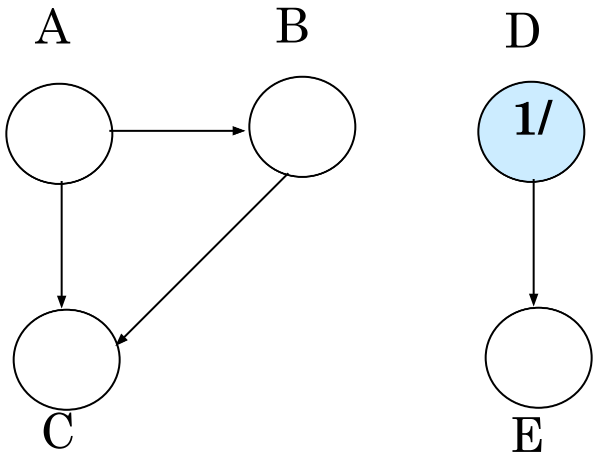
- Performed on a DAG.
- Linear ordering of the vertices of G such that if $(u, v) \in E$, then u appears somewhere before v.

Topological-Sort (G)

1. call DFS(G) to compute finishing times $f[v]$ for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. **return** the linked list of vertices

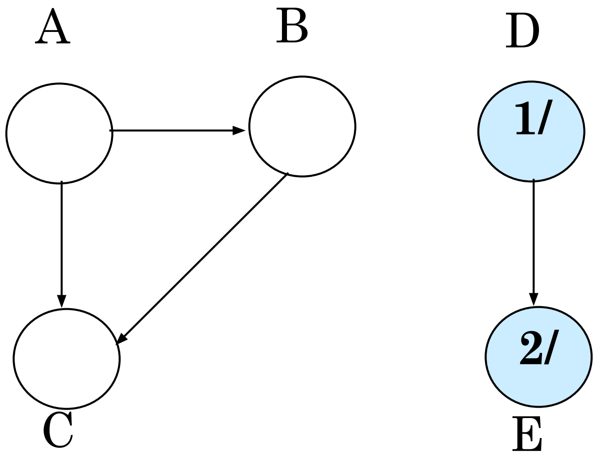
Time: $\Theta(V + E)$.

Example



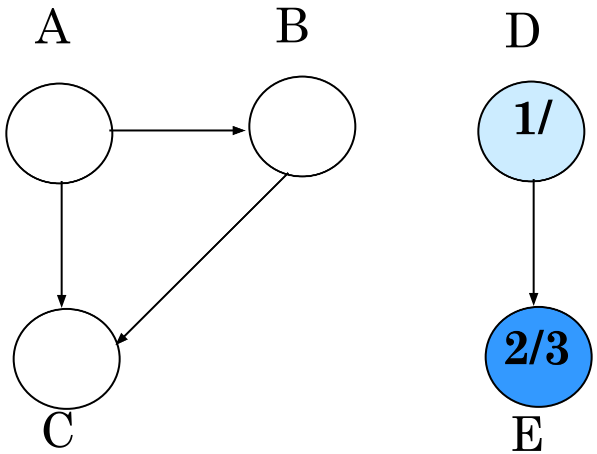
Linked List:

EXAMPLE

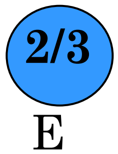


Linked List:

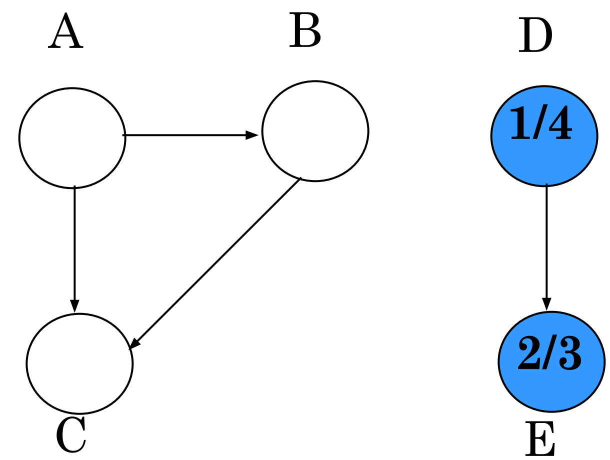
Example



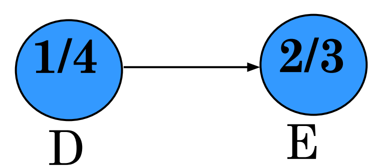
Linked List:



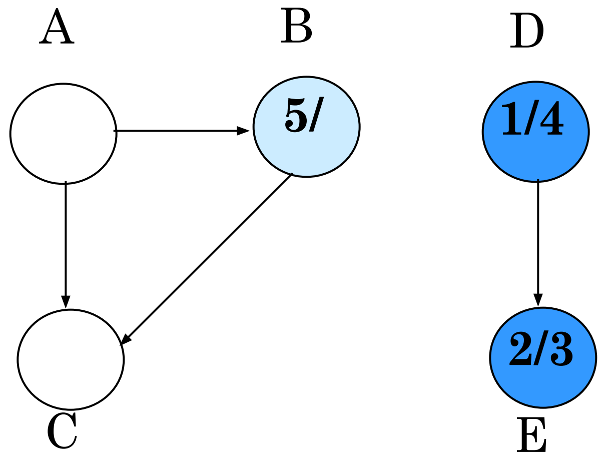
Example



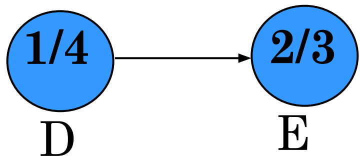
Linked List:



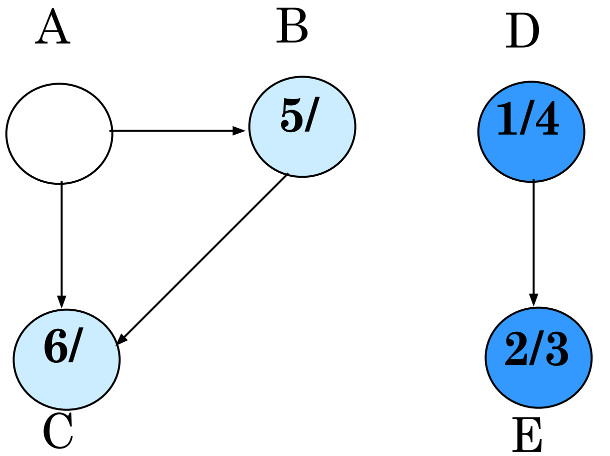
Example



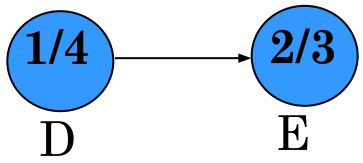
Linked List:



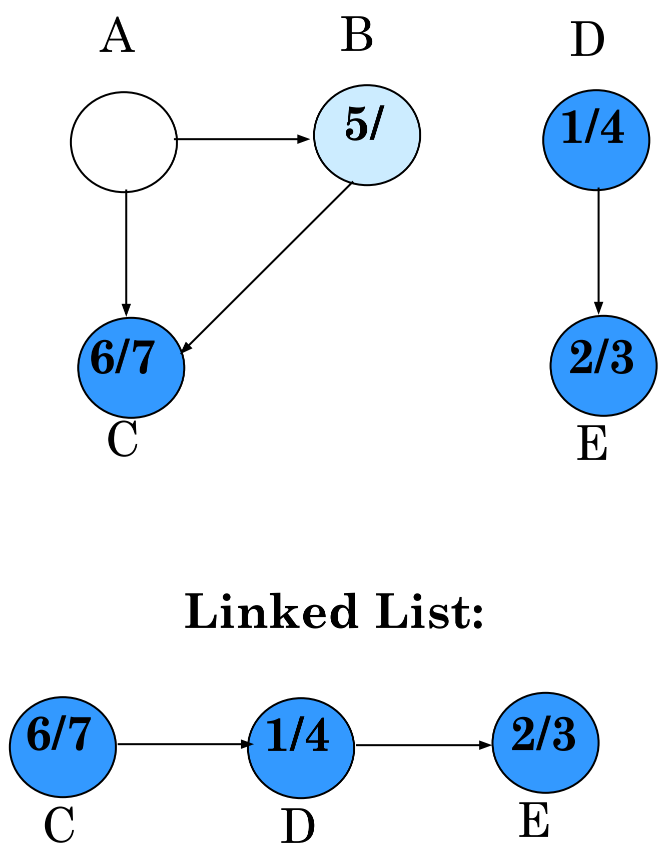
Example



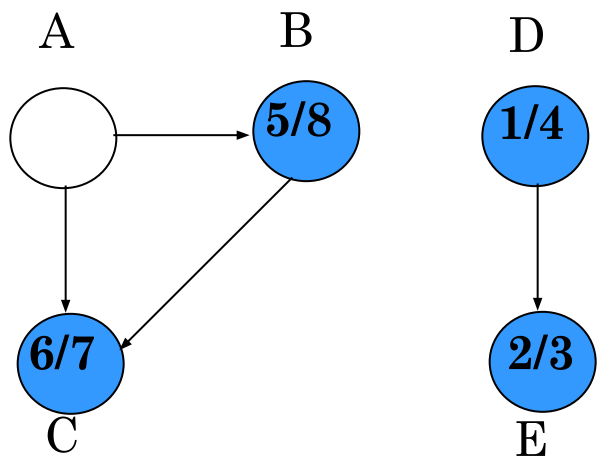
Linked List:



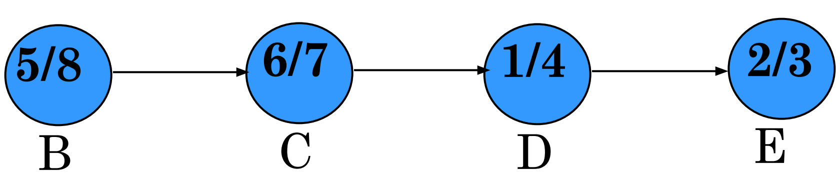
EXAMPLE



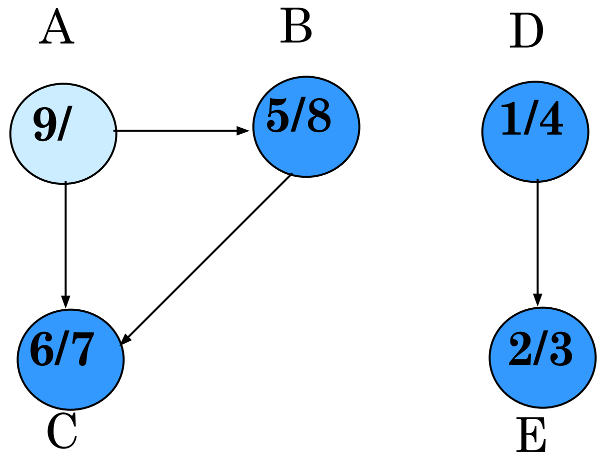
Example



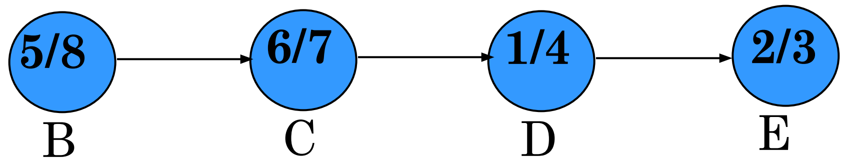
Linked List:



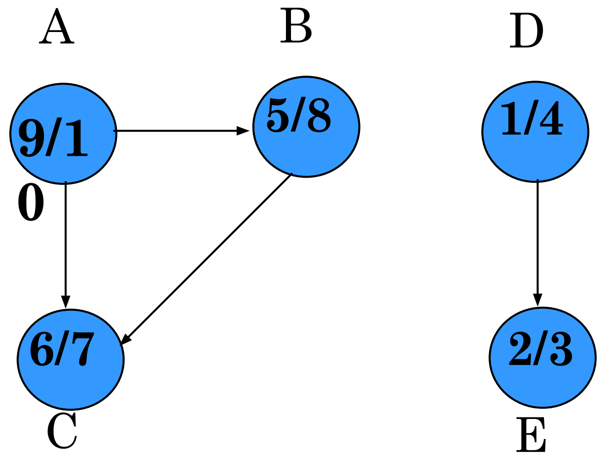
Example



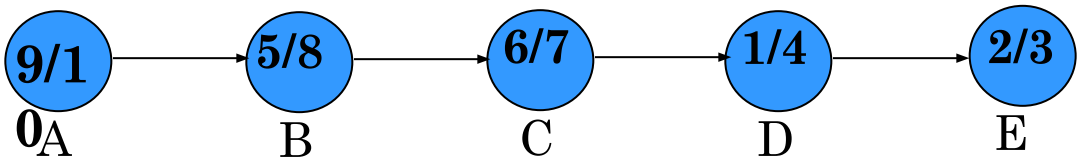
Linked List:



Example



Linked List:



The End