COT 6405 Introduction to Theory of Algorithms

Topic 14. Graph Algorithms

Elementary Graph Algorithms

- How to represent a graph?
  - Adjacency lists
  - Adjacency matrix
- How to search a graph?
  - Breadth-first search
  - Depth-first search

Graph Variations

- Variations:
  - A connected graph has a path from every vertex to every other
  - In an undirected *graph:*
    - edge (u,v) = edge (v,u)
    - No self-loops
  - In a directed graph:
    - Edge (u,v) goes from vertex u to vertex v, notated $u \rightarrow v$

Graph Variations

- More variations:
  - A weighted graph associates weights with either the edges or the vertices
    - E.g., a road map: edges 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)

Graph G = (V, E)

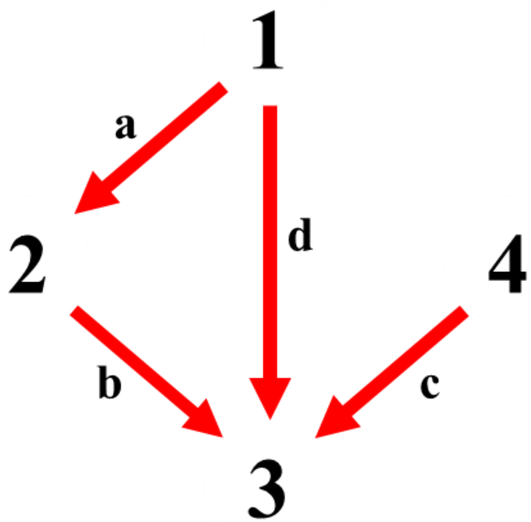
- A graph G = (V, E)
  - V = set of vertices, E = set of edges
- We will typically express running times in terms of $|E|$ and $|V|$ (often dropping the $|\,|$'s)
  - If $|E| \approx |V|^2$, the graph is dense
  - If $|E| \approx |V|$, the graph is sparse
- If you know you are dealing with dense or sparse graphs, we different data structures
  - Dense graph → adjacency matrix
  - Sparse graph → adjacency lists

22.1 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

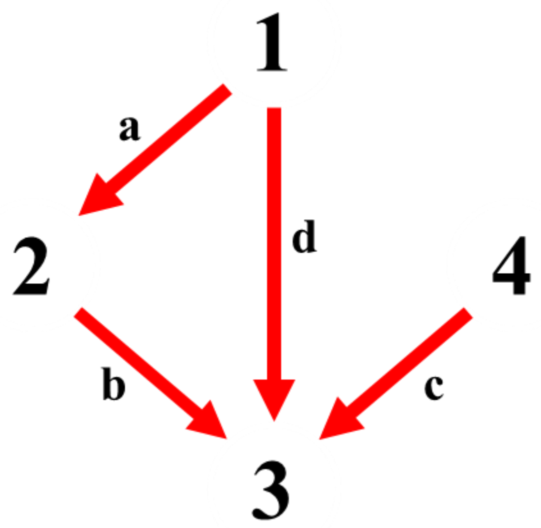
- Example:



A	1	2	3	4
1				
2				
3			??	
4				

Graphs: Adjacency Matrix

- Example:



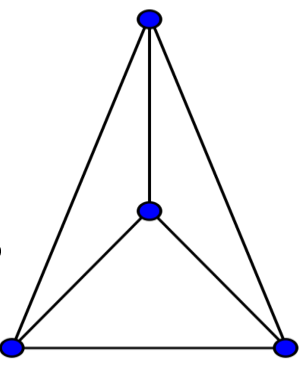
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

Graphs: Adjacency Matrix

- How much storage does the adjacency matrix require?
- A: $O(V^2)$
- What is the minimum amount of storage needed by an adjacency matrix representation of an undirected graph with 4 vertices?
- A: 6 bits
  - Undirected graph $\rightarrow$ matrix is symmetric
  - No self-loops $\rightarrow$ don't need diagonal

Graphs: Adjacency Matrix

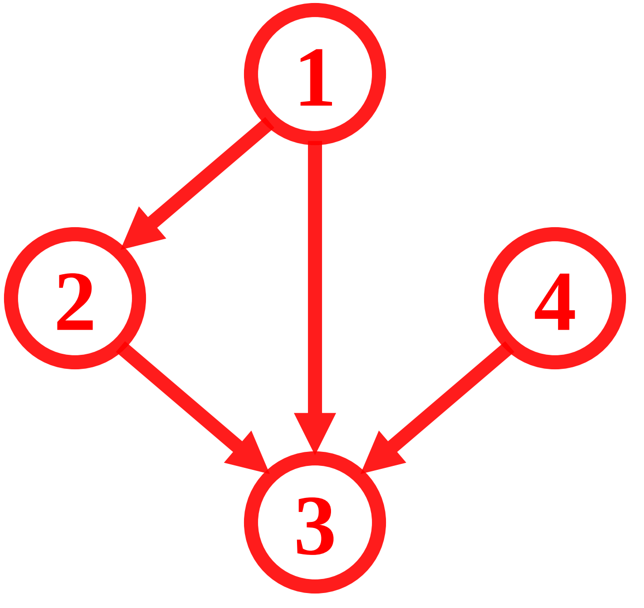
- The adjacency matrix is a dense representation
  - Usually too much storage for large graphs
  - But efficient for small graphs



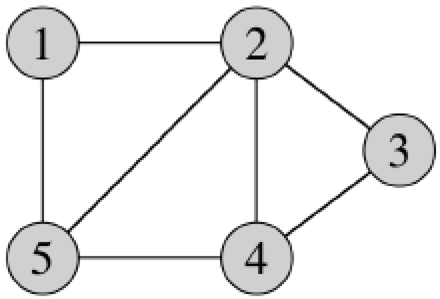
- Most large interesting graphs are sparse
  - E.g., planar graphs, in which no edges cross, have |E| = O(|V|) by Euler’s formula
  - For this reason the adjacency list is often a more appropriate representation

Graphs: Adjacency List

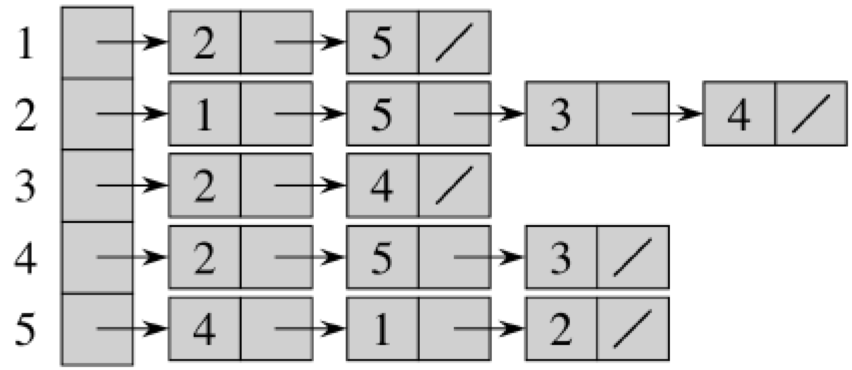
- For each vertex $v \in V$, store a list of vertices adjacent to v
- The same example:
  - Adj[1] = {2, 3}
  - Adj[2] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}



- Undirected

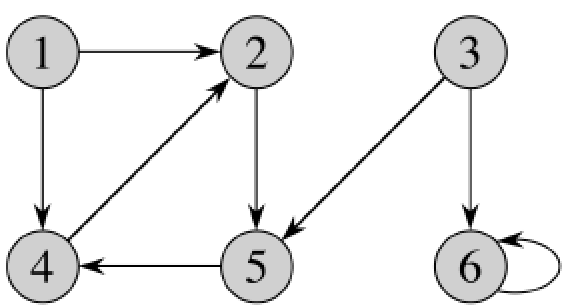


(a)

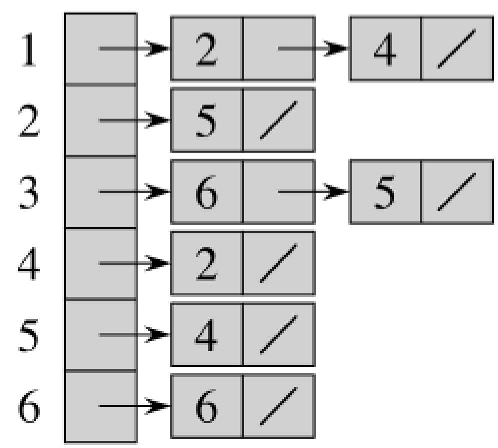


(b)

- Directed Graph



(a)



(b)

Graphs: Adjacency List

- How much storage is required?
  - The degree of a vertex *v* = # incident edges
    - Two edges are called incident, if they share a vertex
    - Directed graphs have in-degree, out-degree
  - For directed graphs, # of items in adjacency lists is
    Σ out-degree(*v*) = |E|
    takes Θ(V + E) storage
  - For undirected graphs, # items in adjacency lists is
    Σ degree(v) = 2 |E|
    also Θ(V + E) storage
- So: Adjacency lists take O(V+E) storage

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: may build a forest if a graph is not connected

Breadth-First Search (BFS)

- "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

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

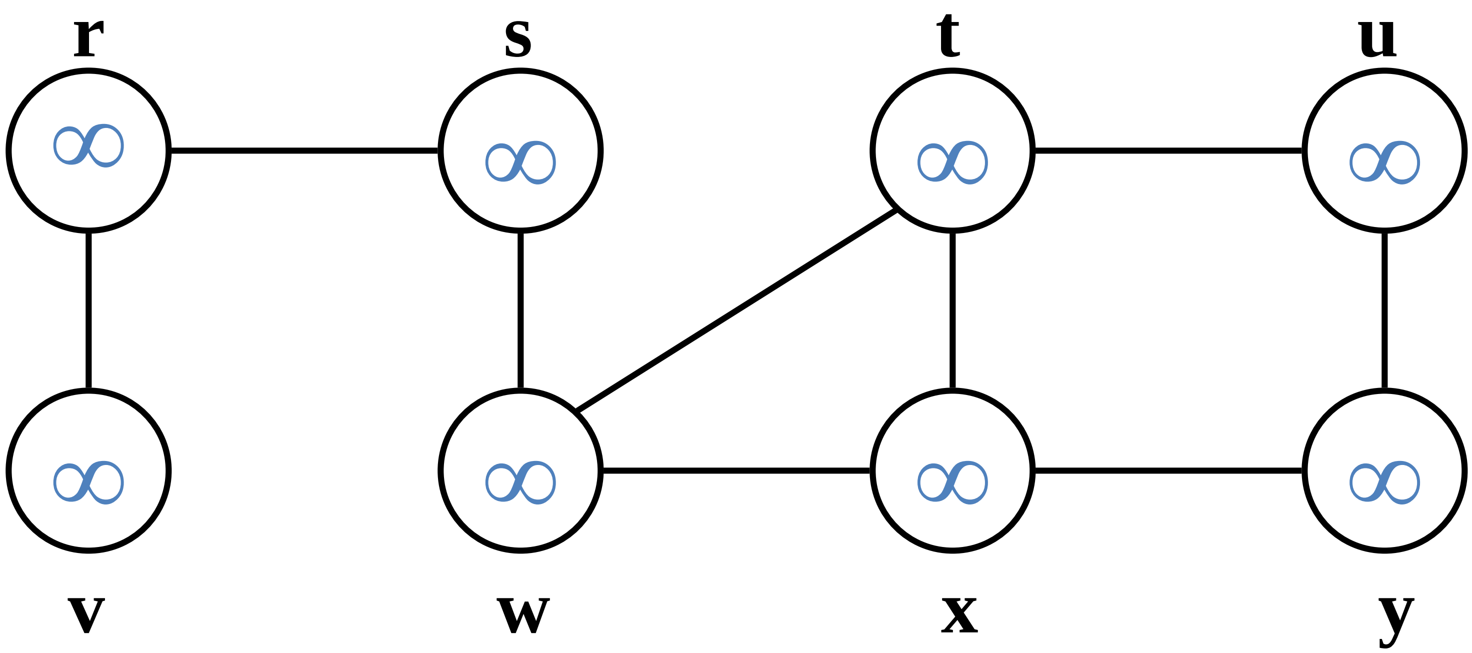
Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    Q = {s};              // Q is a queue; 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;
    }
}
```

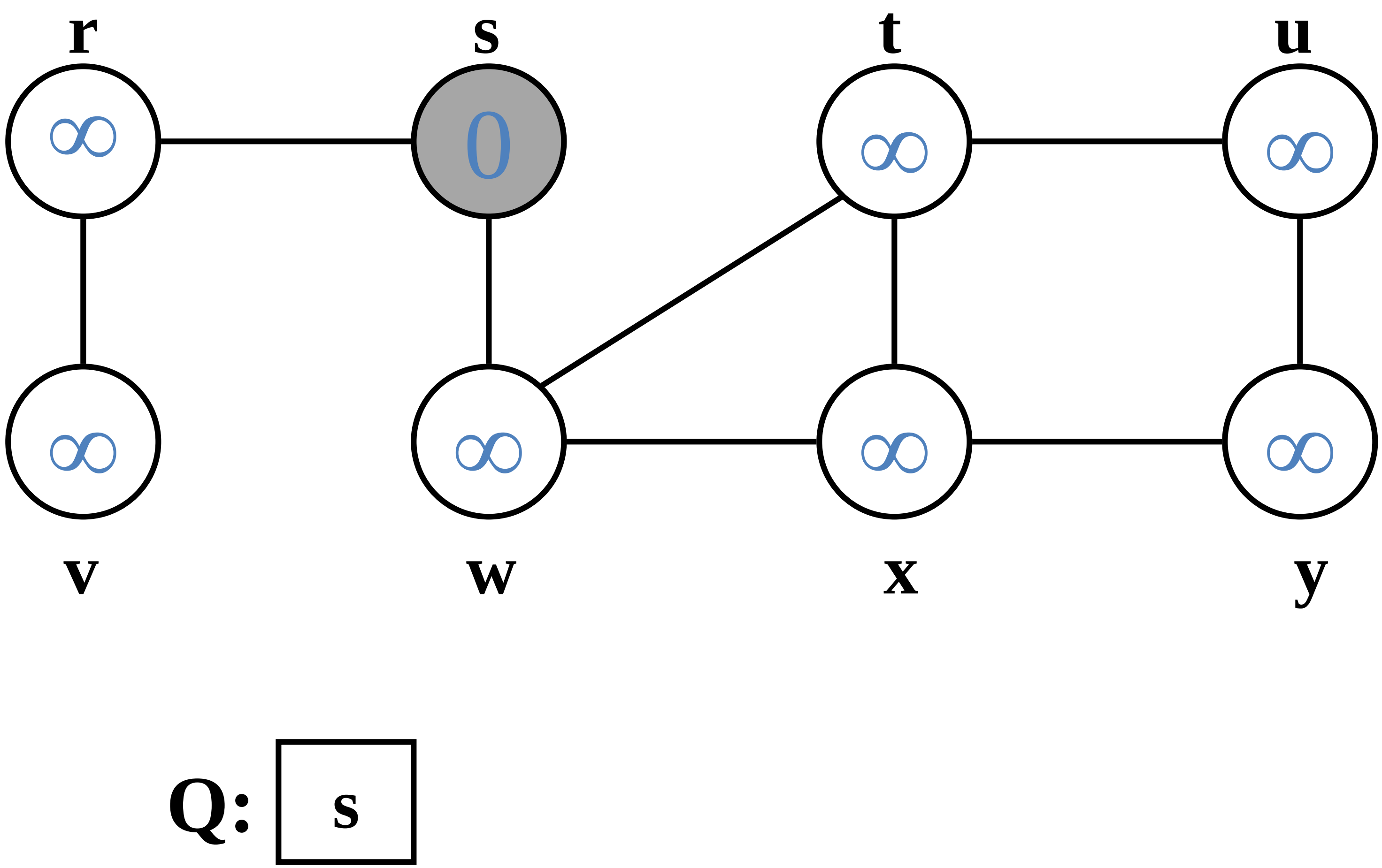
What does `v.d` represent?

What does `v.p` represent?

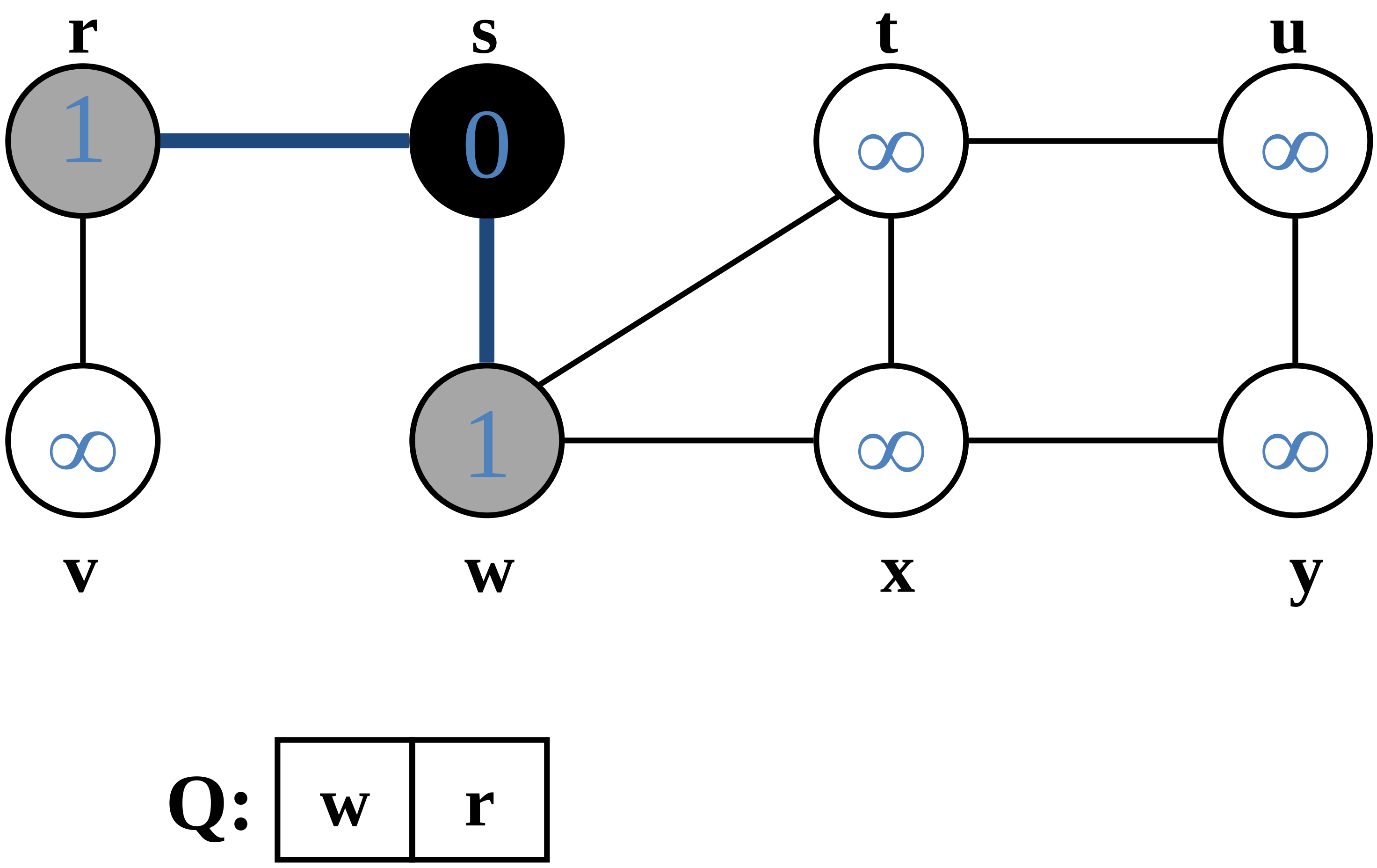
BFS: Initialization all nodes WHITE



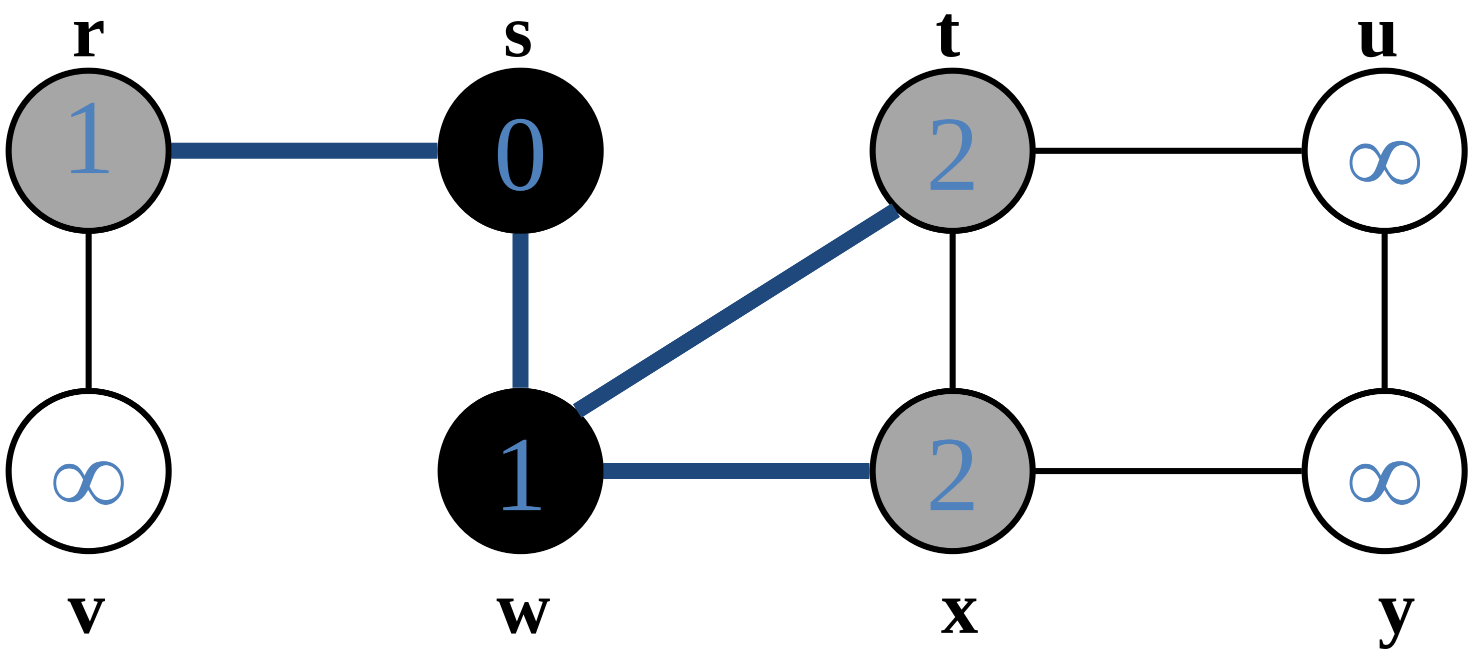
Breadth-First Search: enqueue s



dequeue s; s is done; enqueue w and r

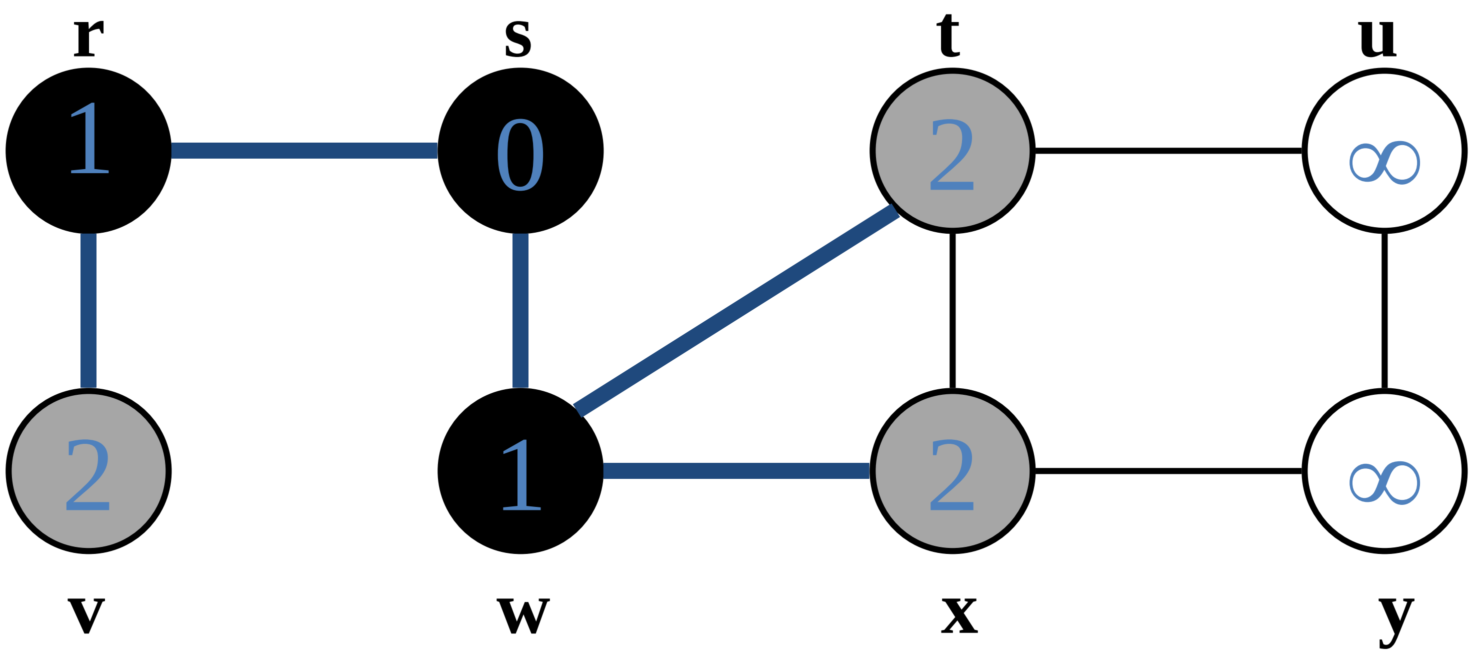


dequeue w, enqueue t and x



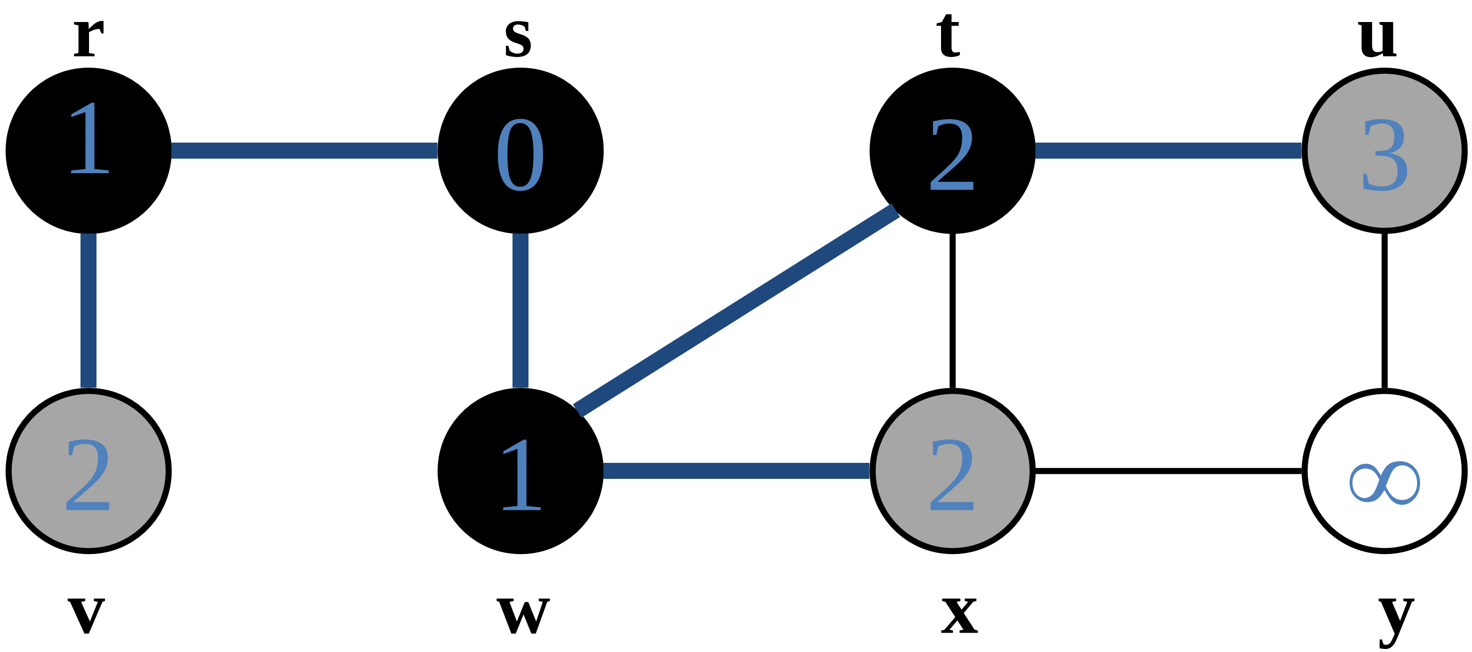
Q: | r | t | x |

dequeue r, enqueue v



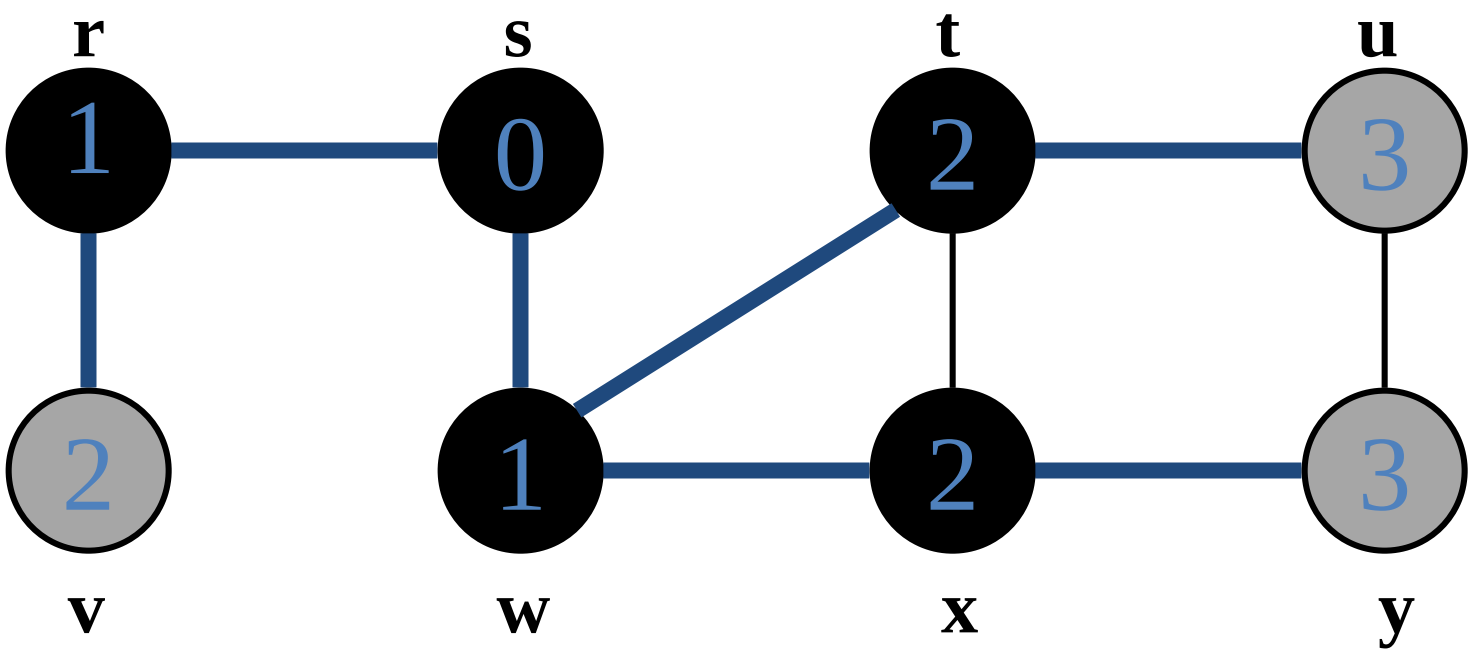
Q: | t | x | v |

dequeue t, enqueue u



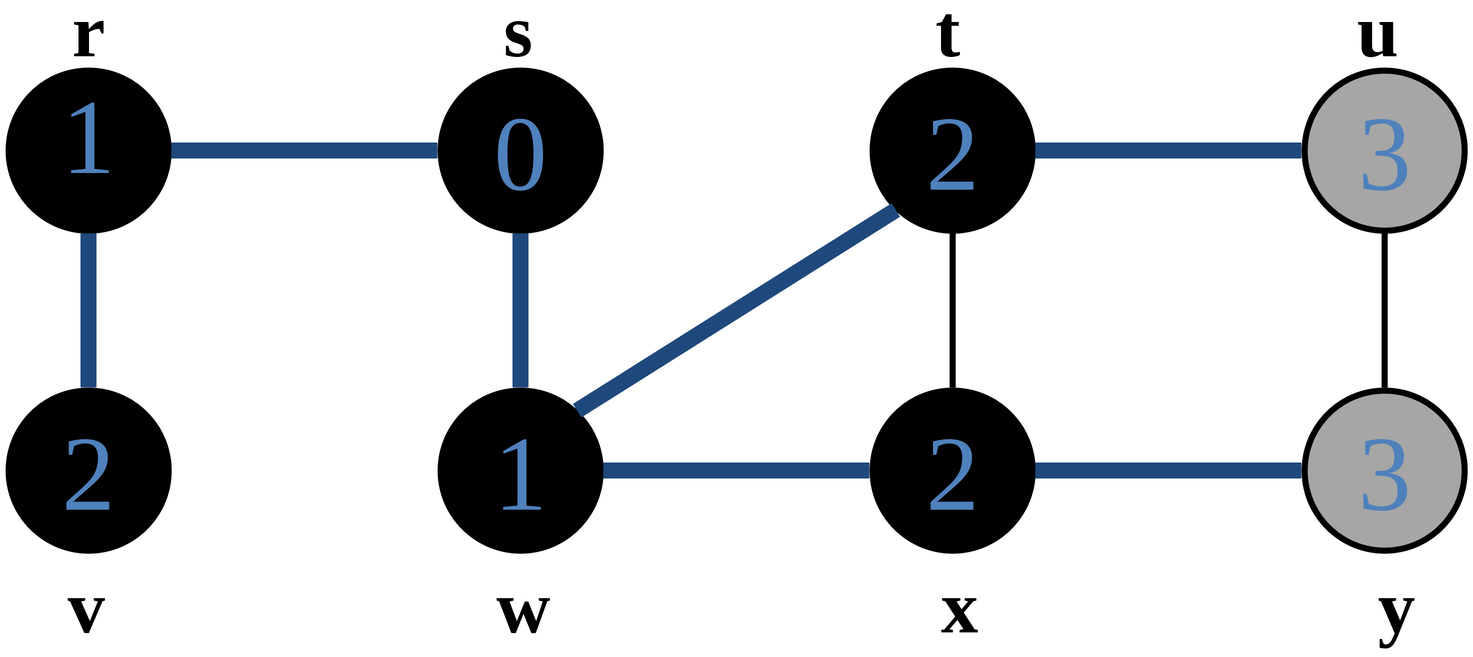
Q:	x	v	u

dequeue x, no enqueue



Q: | v | u | y |

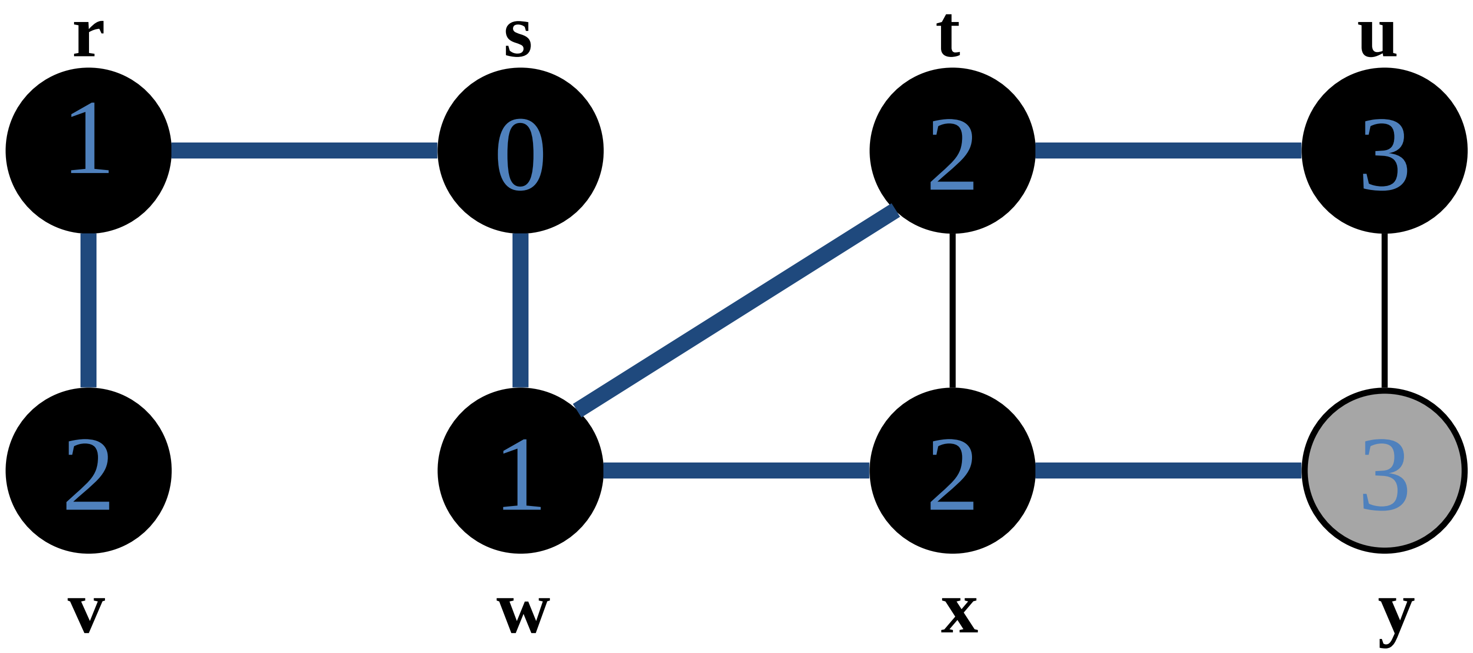
dequeue v, no enqueue



Q:

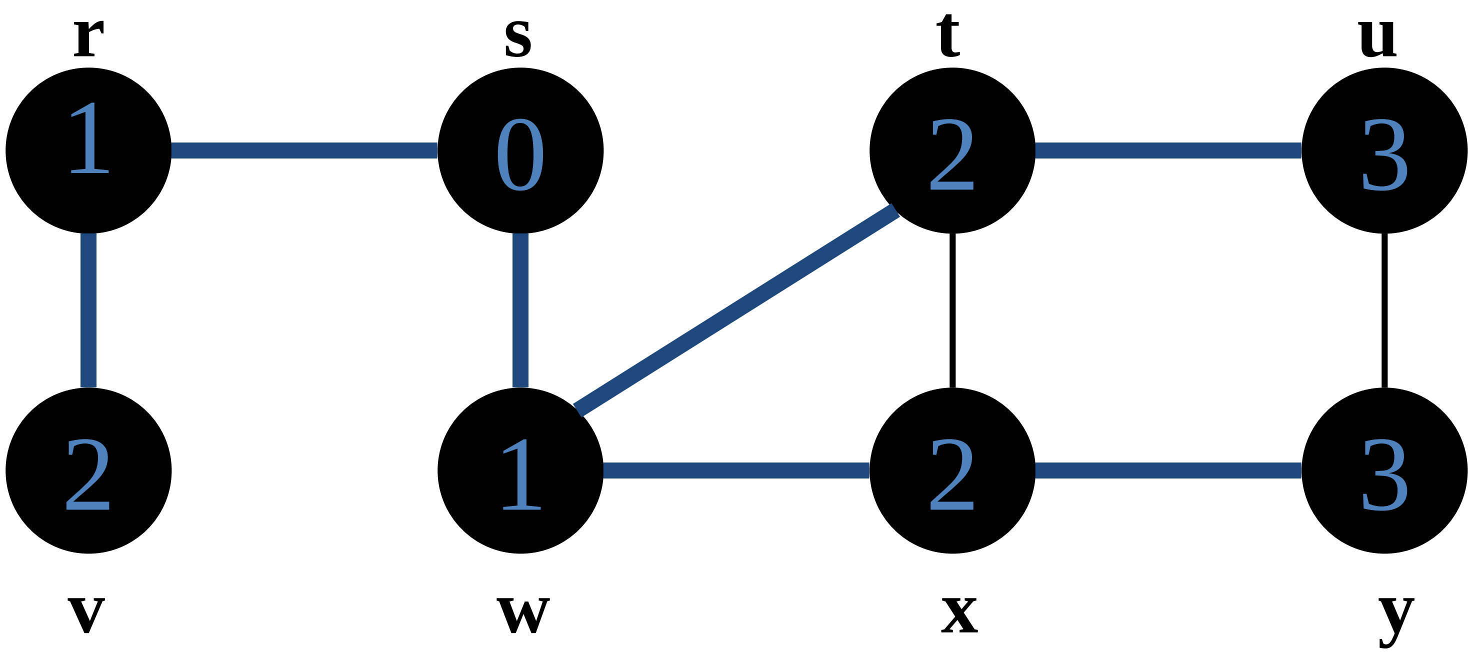
u	y

dequeue u, no enqueue



Q: y

dequeue y, no enqueue



Q: Ø

BFS: The Code Again

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

What will be the running time?

Time analysis

- The total running time of BFS is $O(V + E)$
- Proof:
  - Each vertex is dequeued at most once. Thus, total time devoted to queue operations is $O(V)$.
  - For each vertex, the corresponding adjacency list is scanned at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$.
  - Thus, the total running time is O(V+E)

BFS: The Code Again

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

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

Total space used: O(V)

Breadth-First Search: Properties

- BFS calculates the shortest-path distance to the source node
  - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or ∞ if v not reachable from s
- BFS builds breadth-first tree, in which paths to root represent shortest paths in G
  - Thus, we can use BFS to calculate a shortest path from one vertex to another in O(V+E) time

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
    - Timestamp to help us remember who is “new”
  - When all of v’s edges have been explored, backtrack to the vertex from which v was discovered

Depth-First Search: The Code

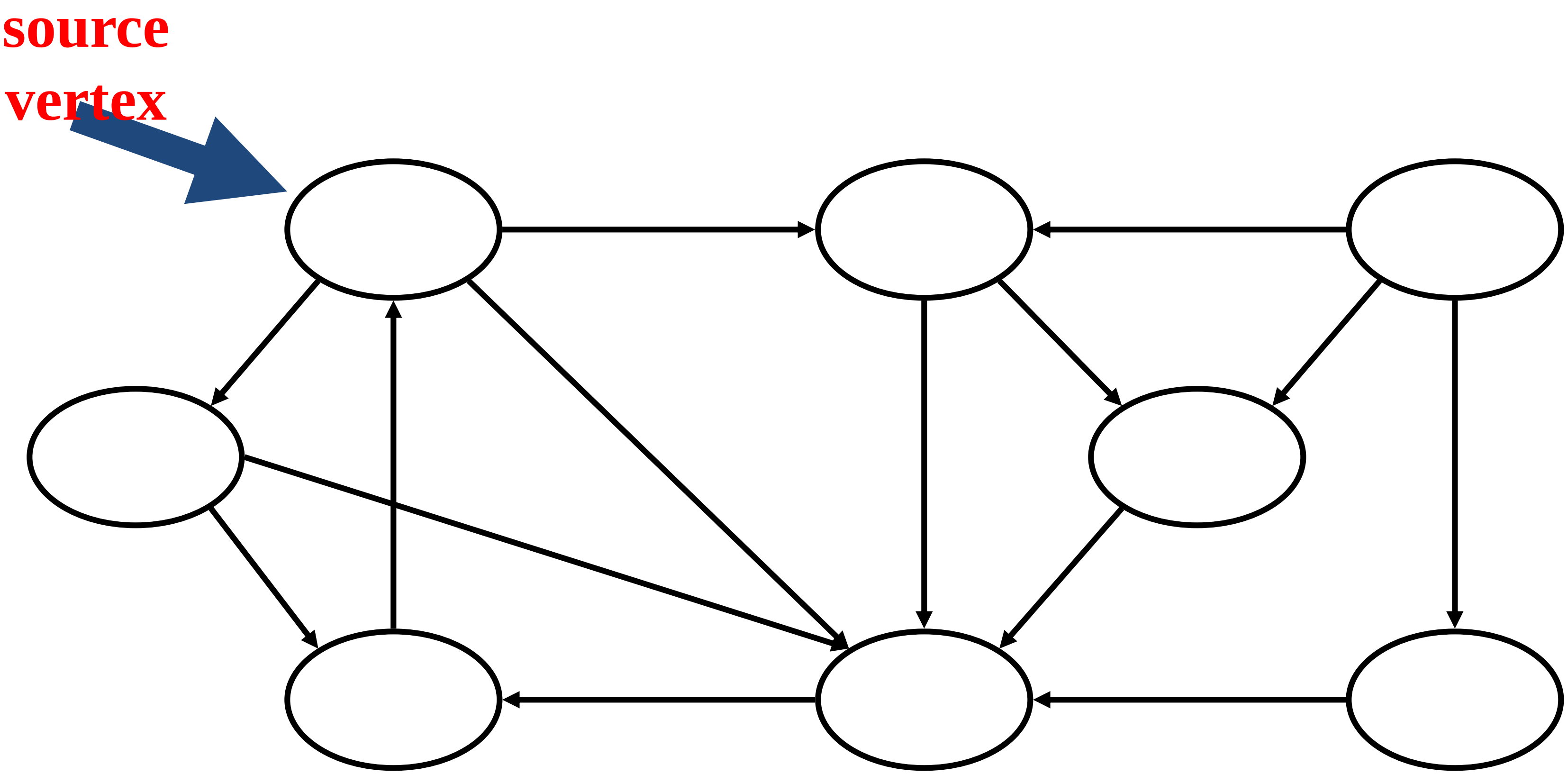
```
DFS(G)
{
  for each vertex u ∈ G.V
  {
      u.color = WHITE
      u.π  = NIL
  }
  time = 0
  for each vertex u ∈ G.V
  {
    if (u.color == WHITE)
       DFS_Visit(G, u)
  }
}
```

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

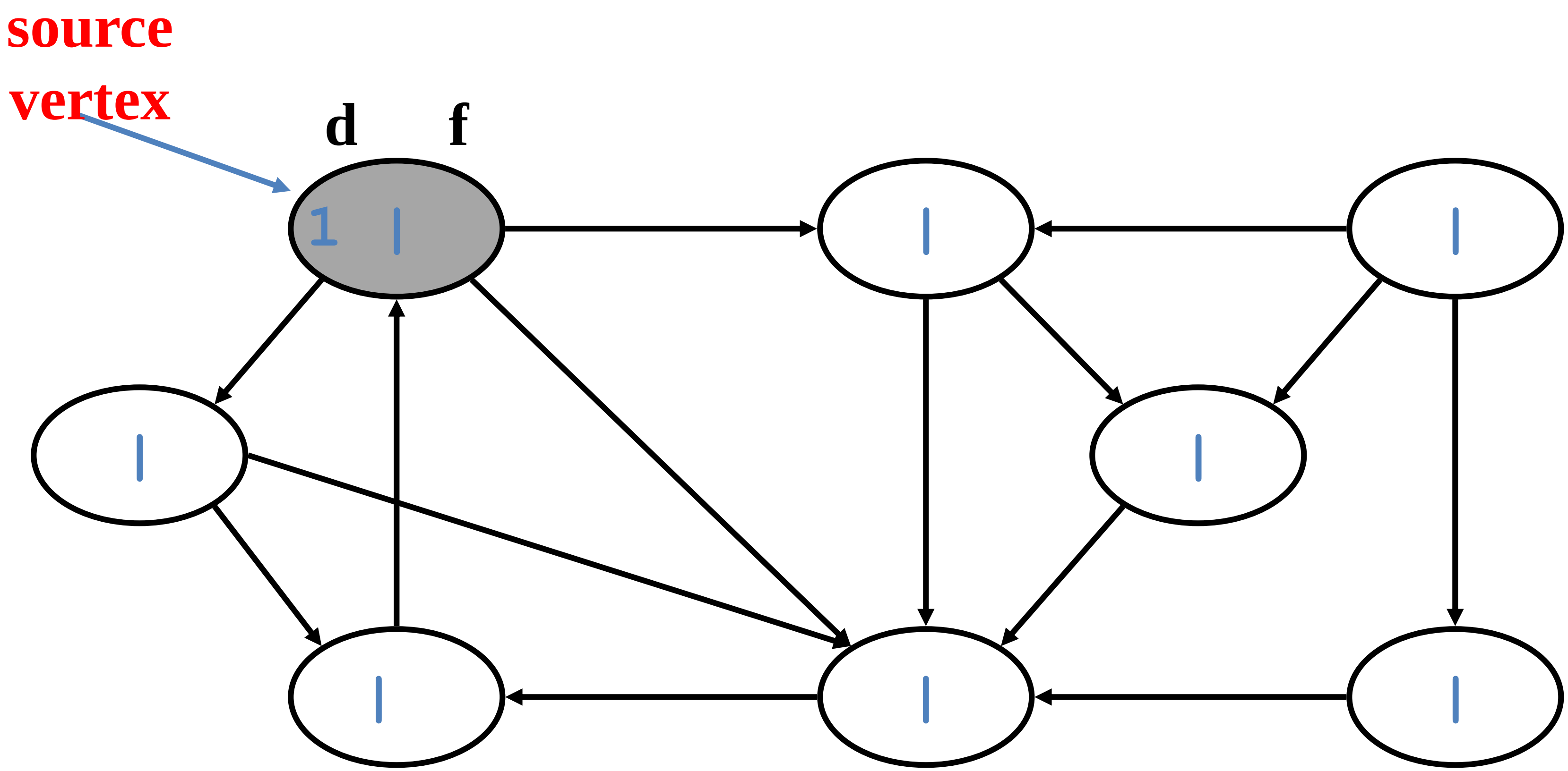
Variables

- u.π stores the predecessor of vertex u
- The first timestamp *u.d* records when *u* is first discovered (and grayed)
- The second timestamp *u.f* records when the search finishes examining *u*'s adjacency list (and blackens *v*).
- These timestamps are used in many graph algorithms and are generally helpful in reasoning about the behavior of depth-first search

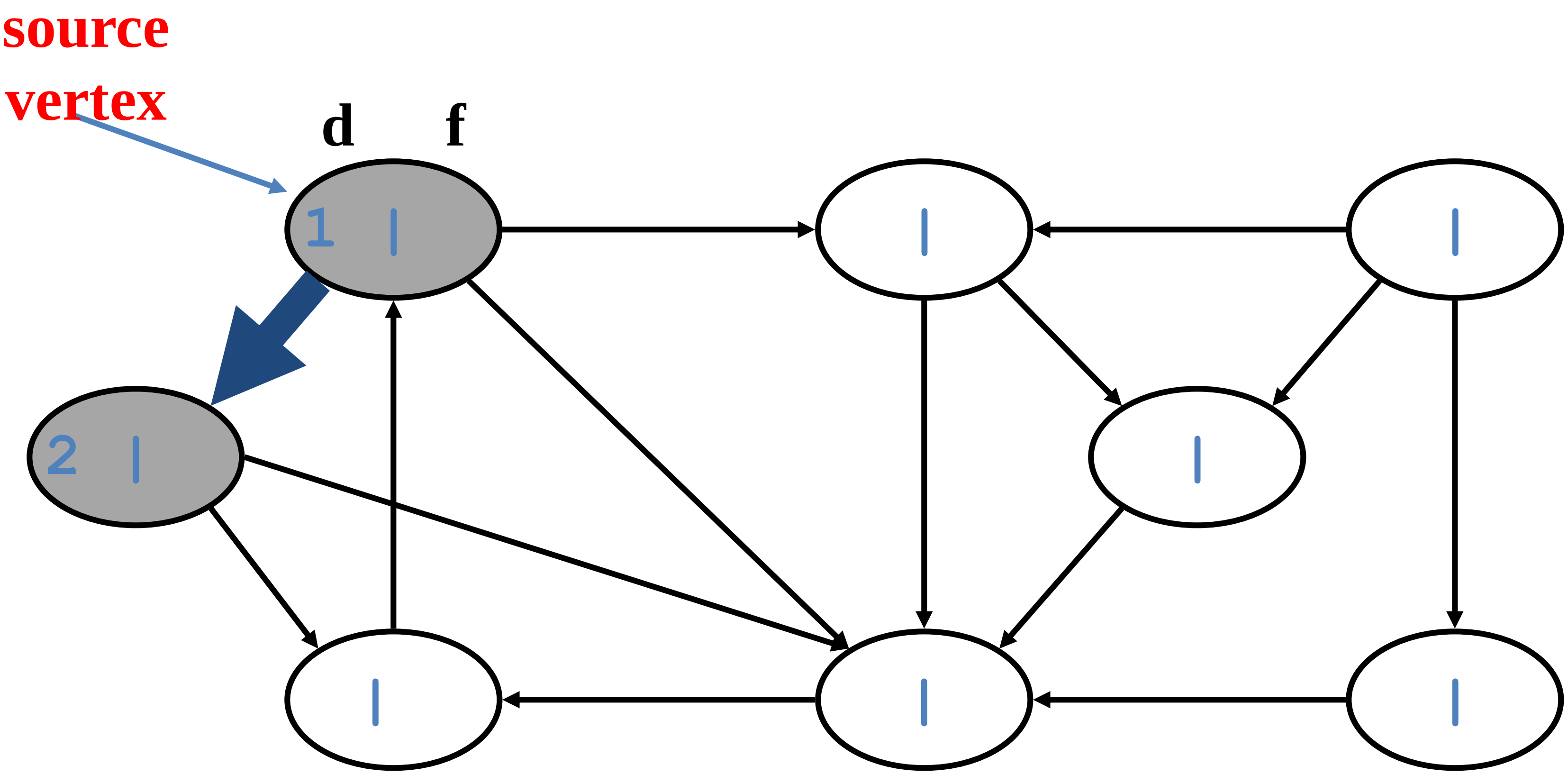
DFS Example: time = 0



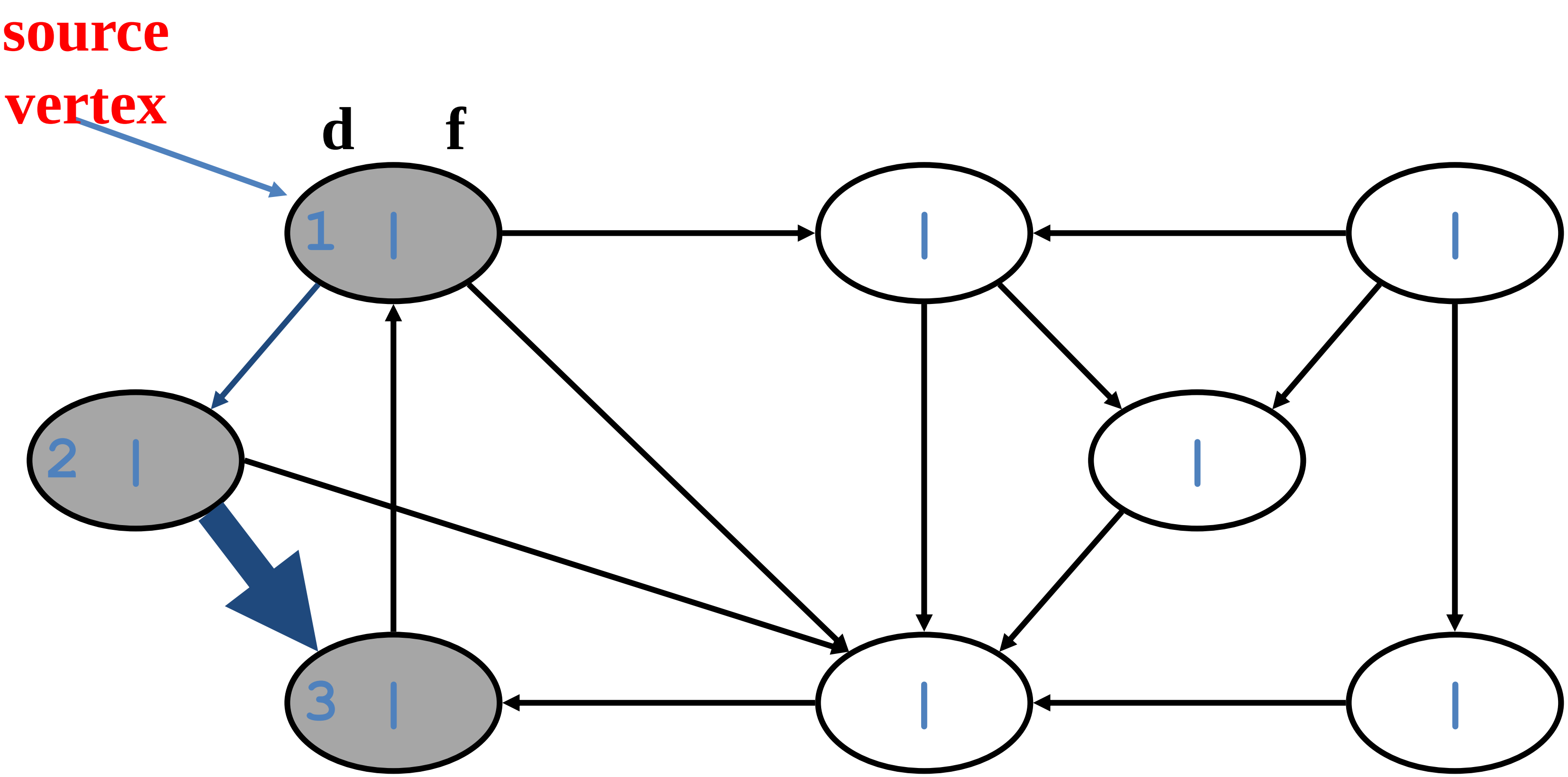
DFS Example: time = 1



DFS Example: time = 2

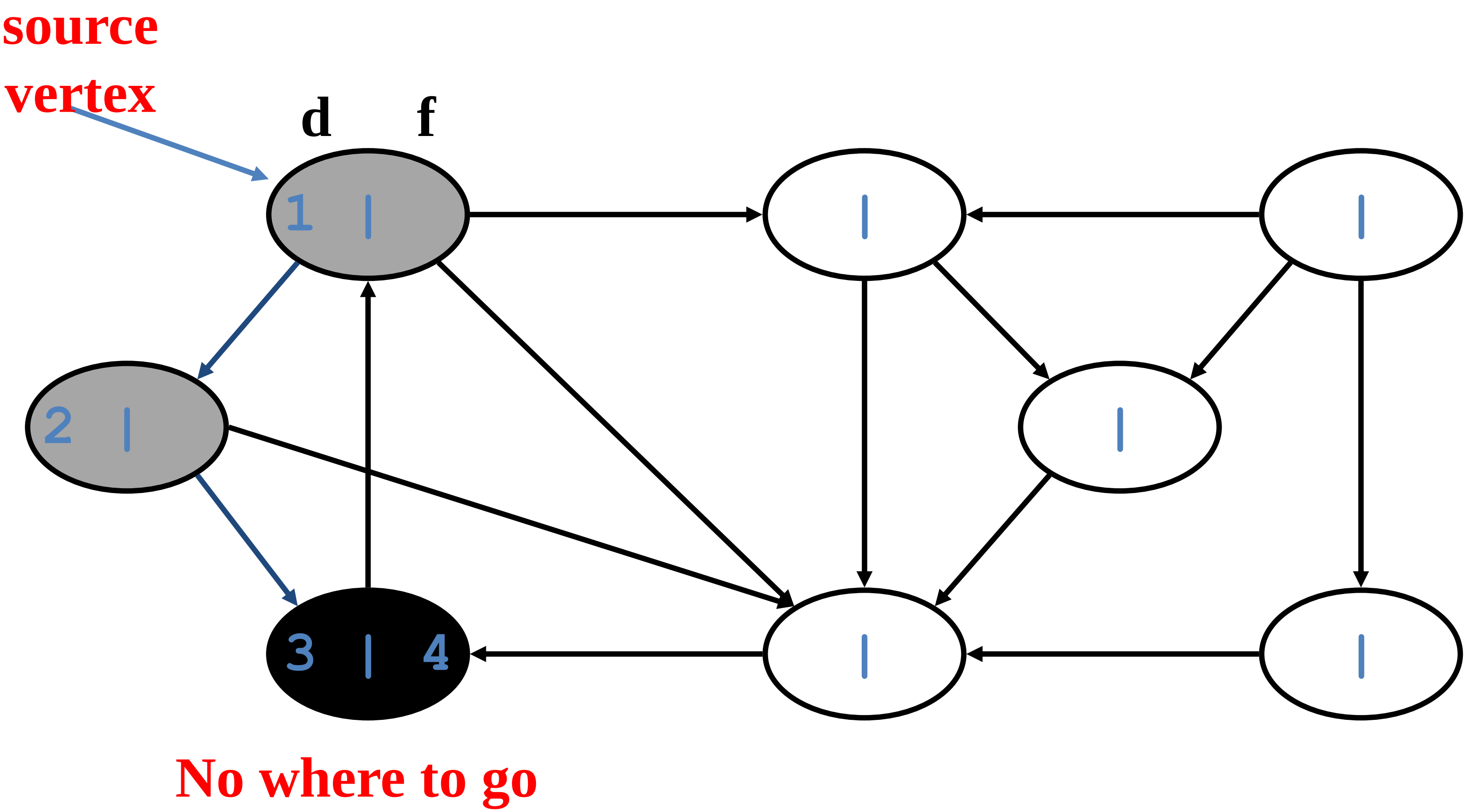


DFS Example: time = 3



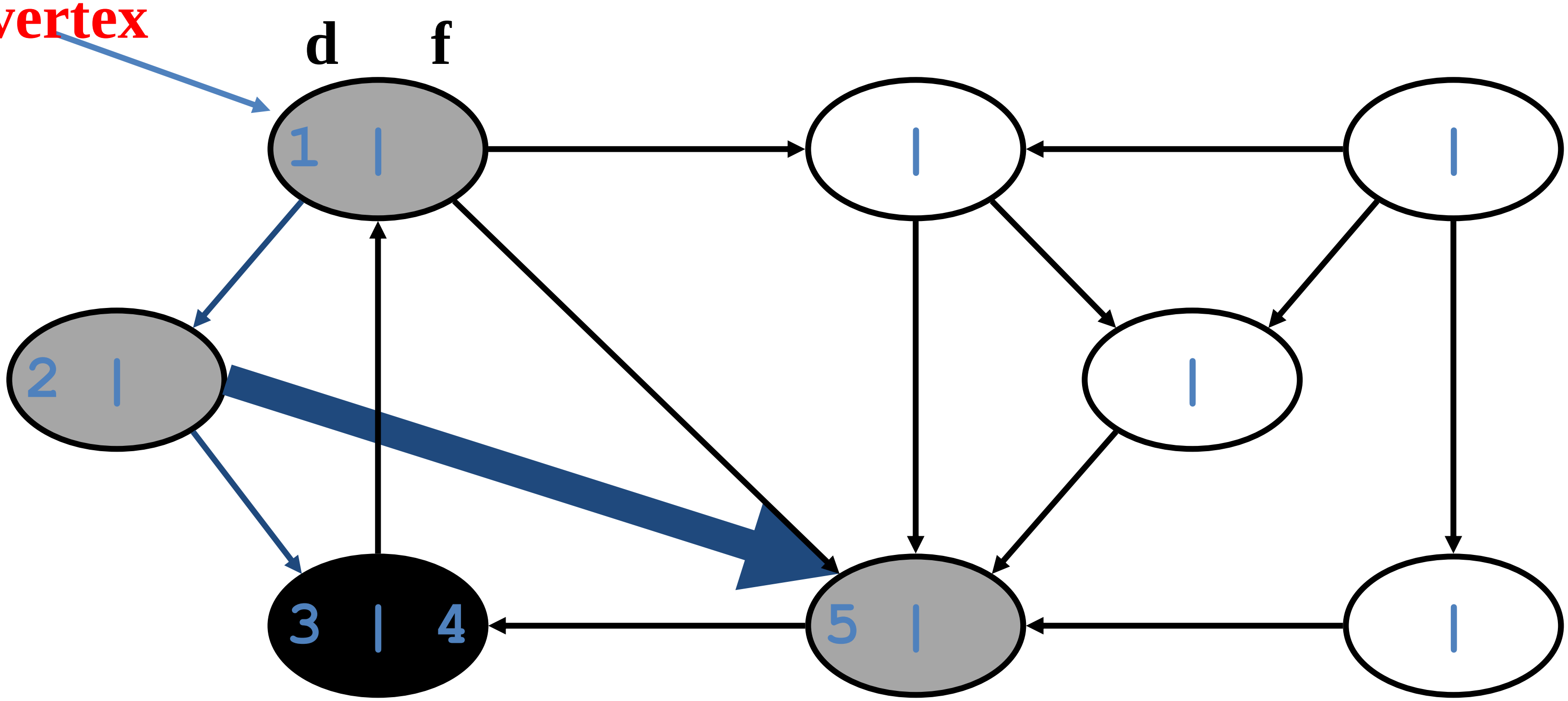
GREEDY: Always to go with white nodes if possible

DFS Example: time = 4



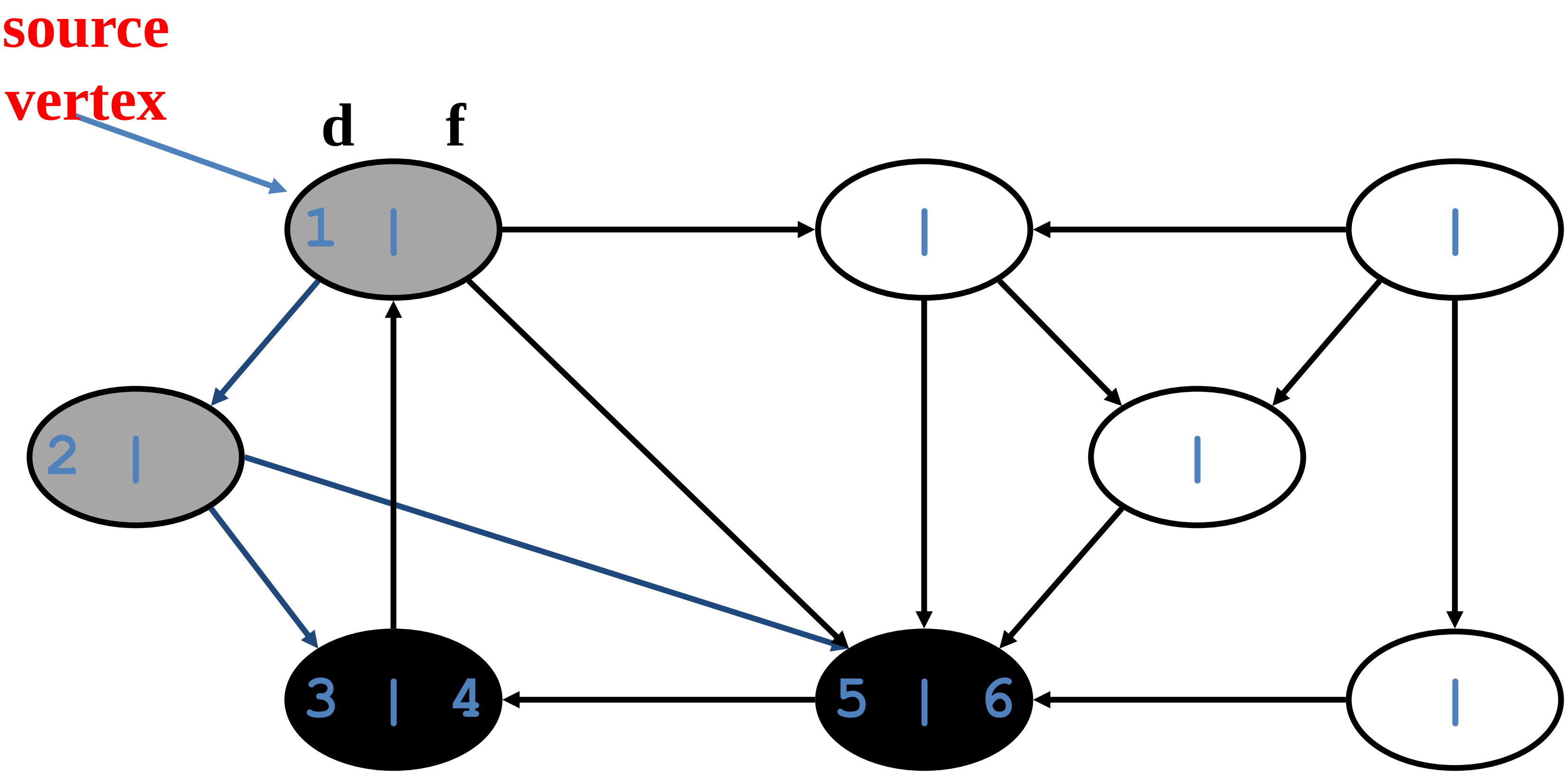
DFS Example: time = 5

source vertex



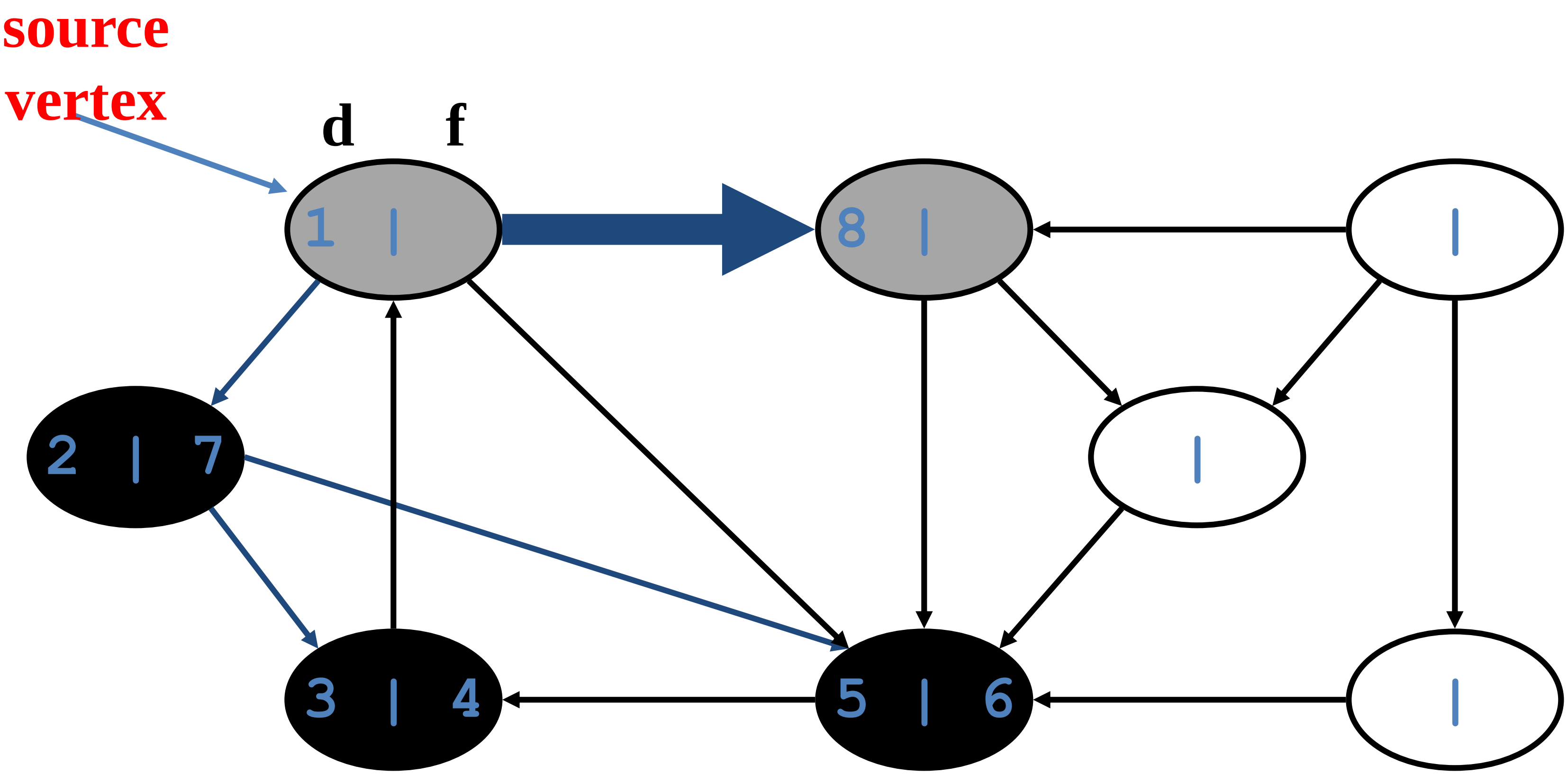
GREEDY: Always to go with white nodes if possible
Based on timestamp, 2 is the newest at this moment

DFS Example: time = 6

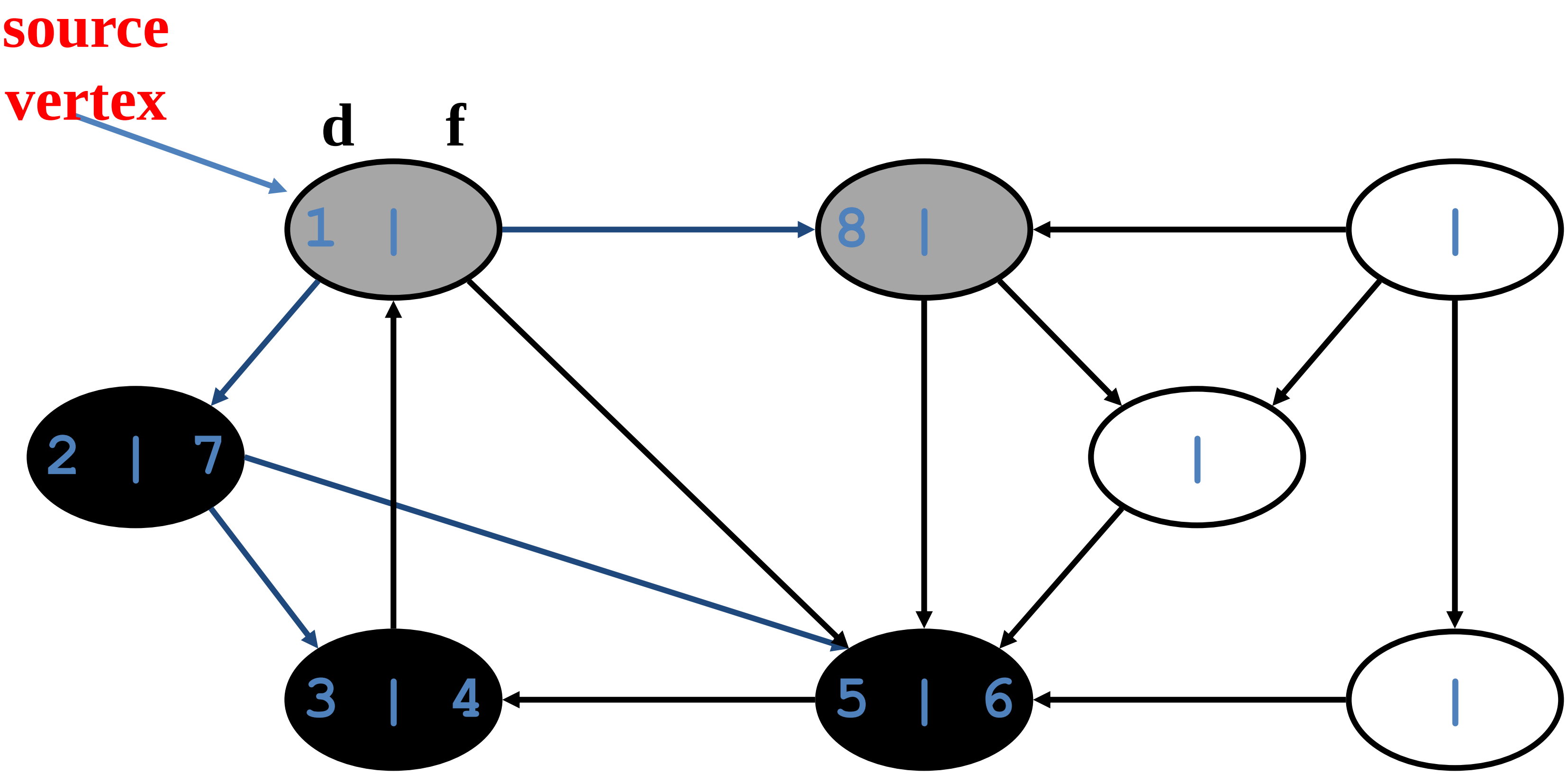


No where to go

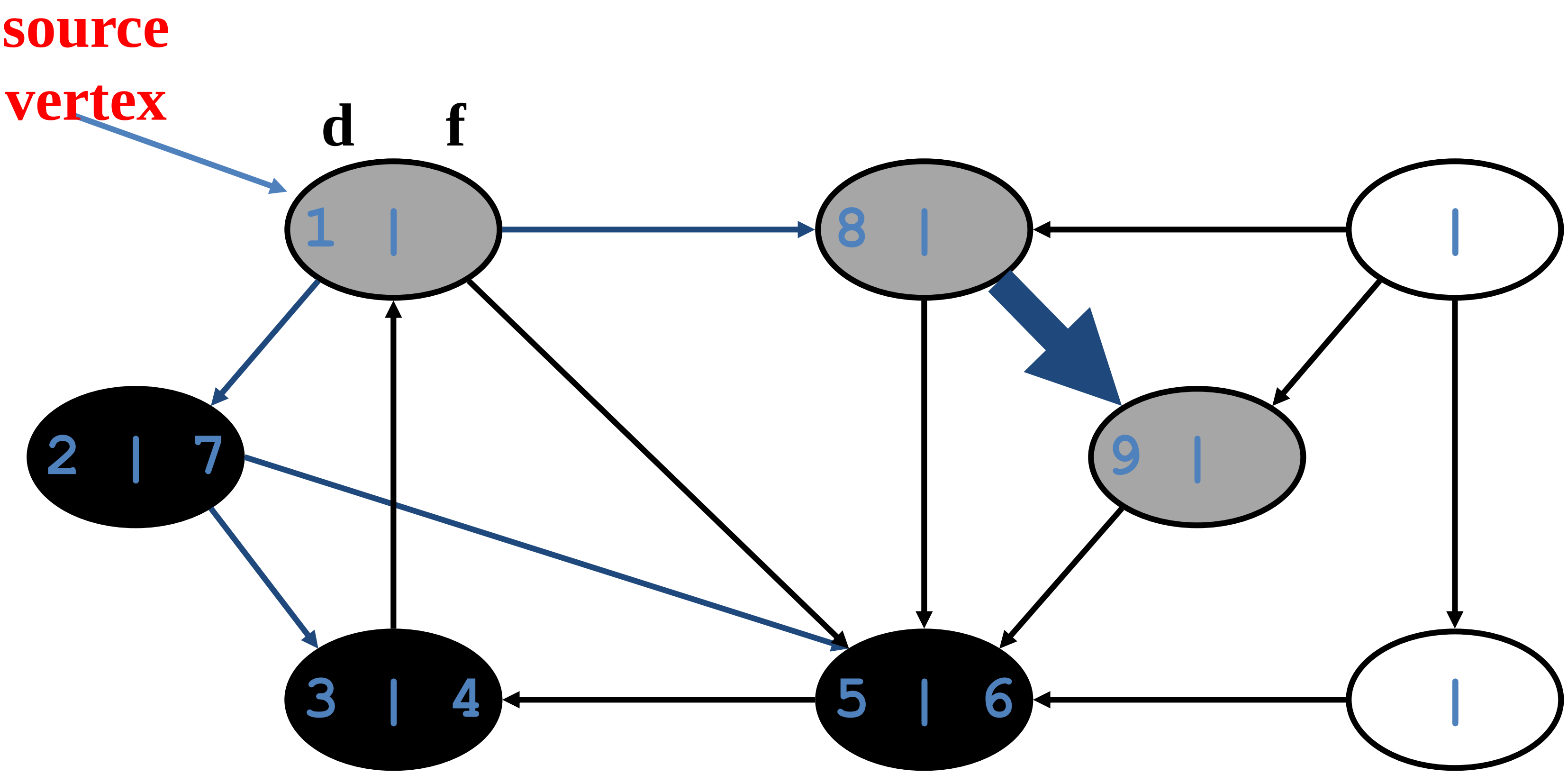
DFS Example: time = 7 and 8



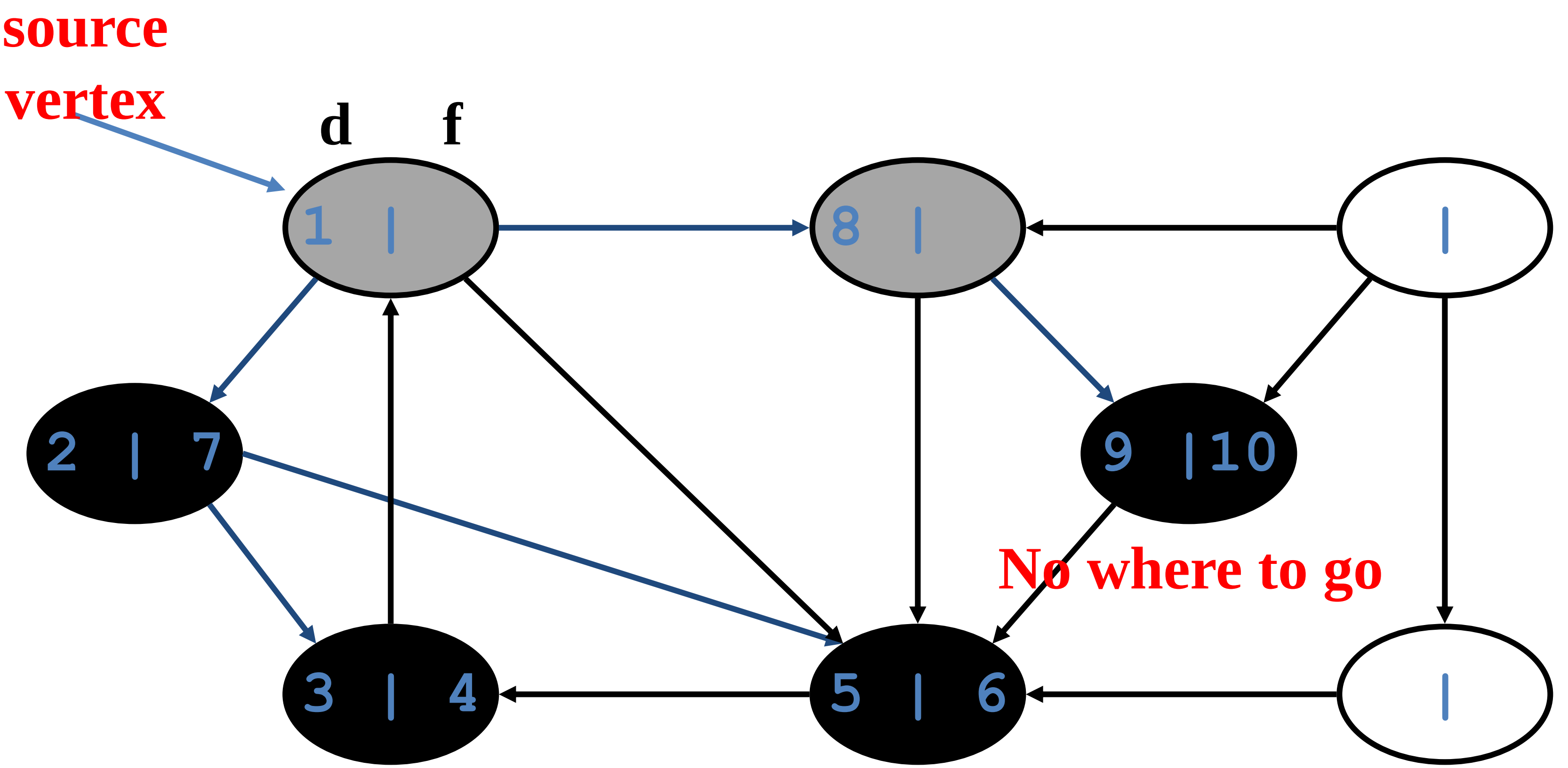
DFS Example



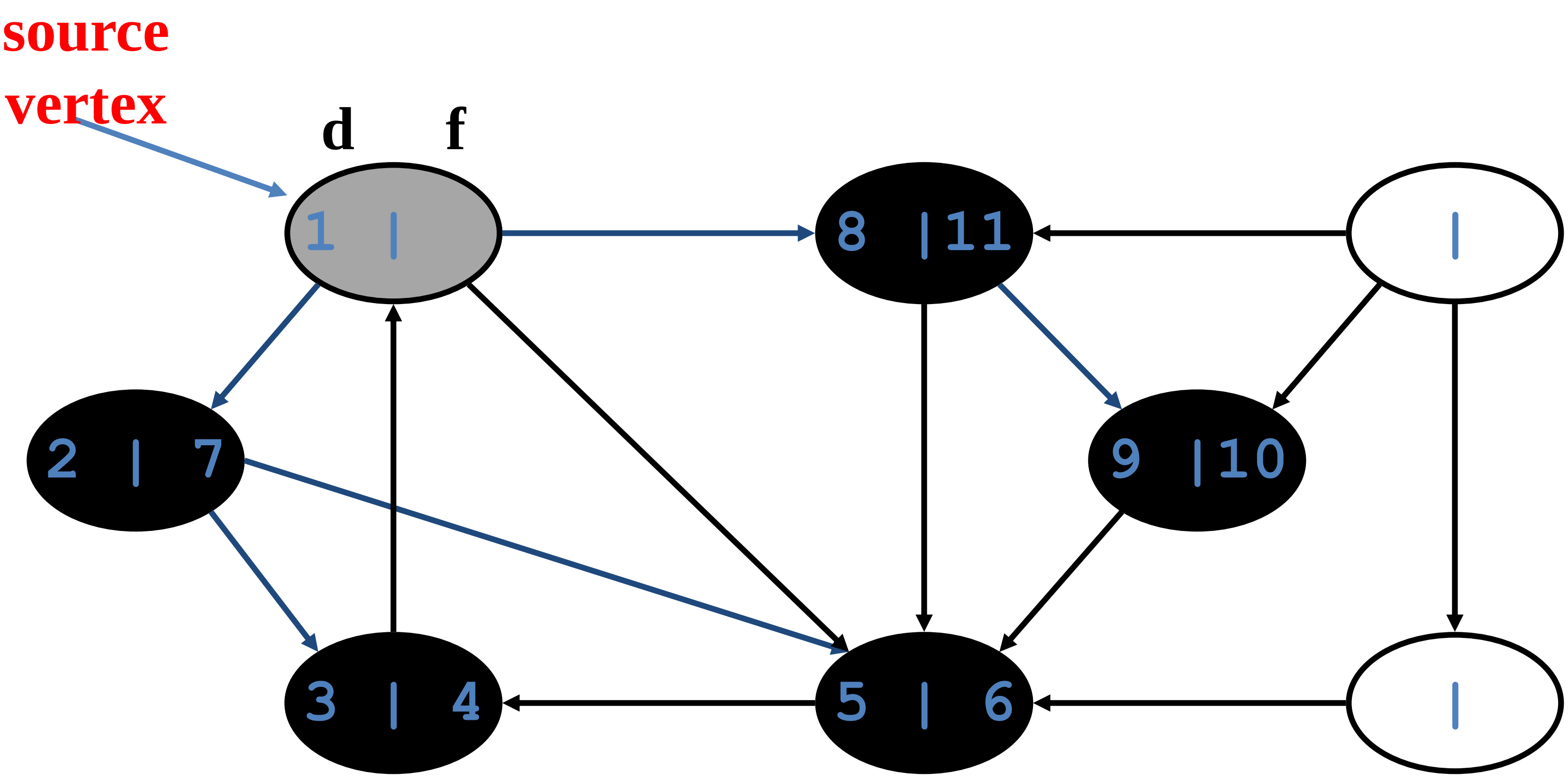
DFS Example: time = 9



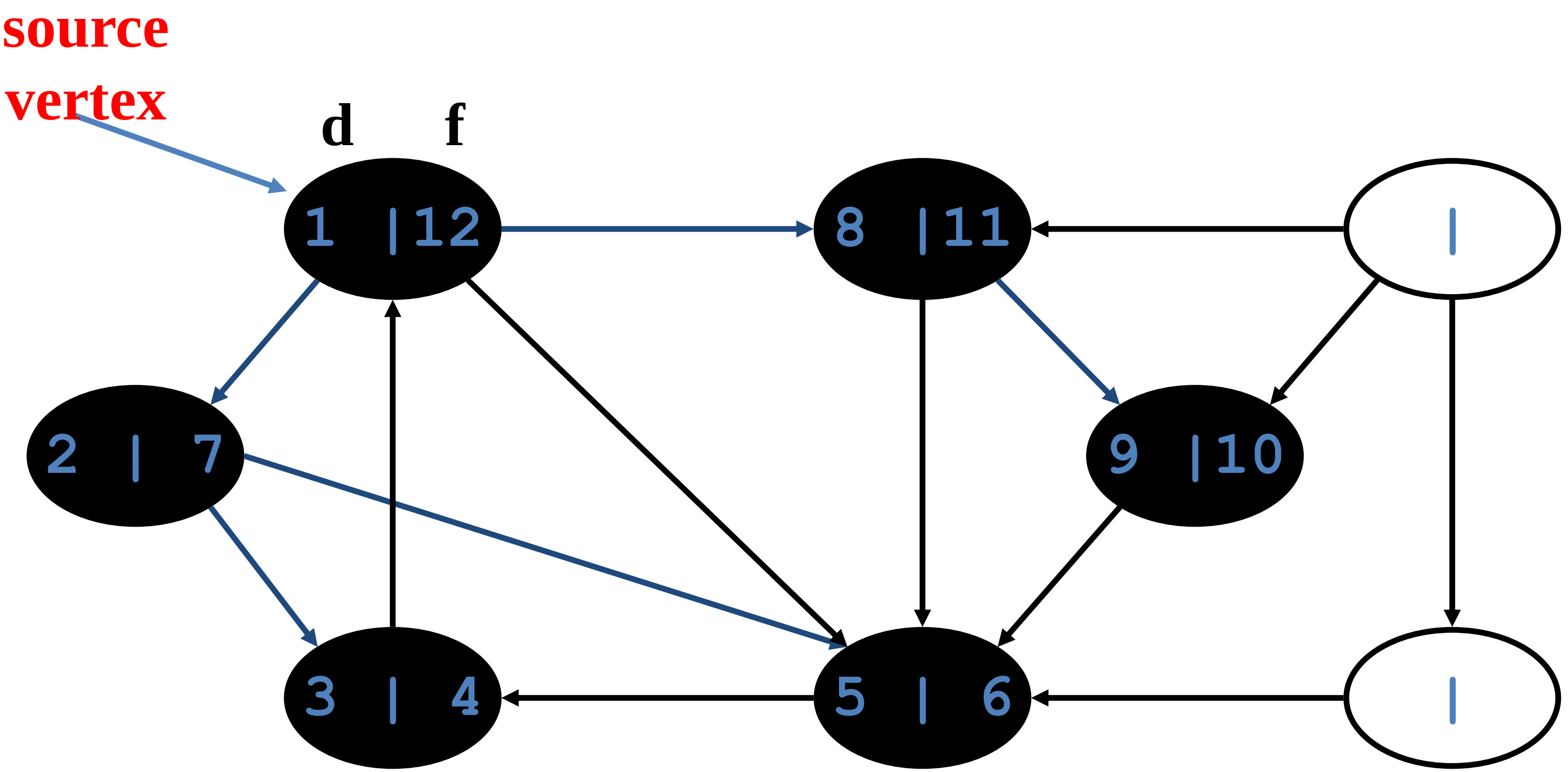
DFS Example: time = 10



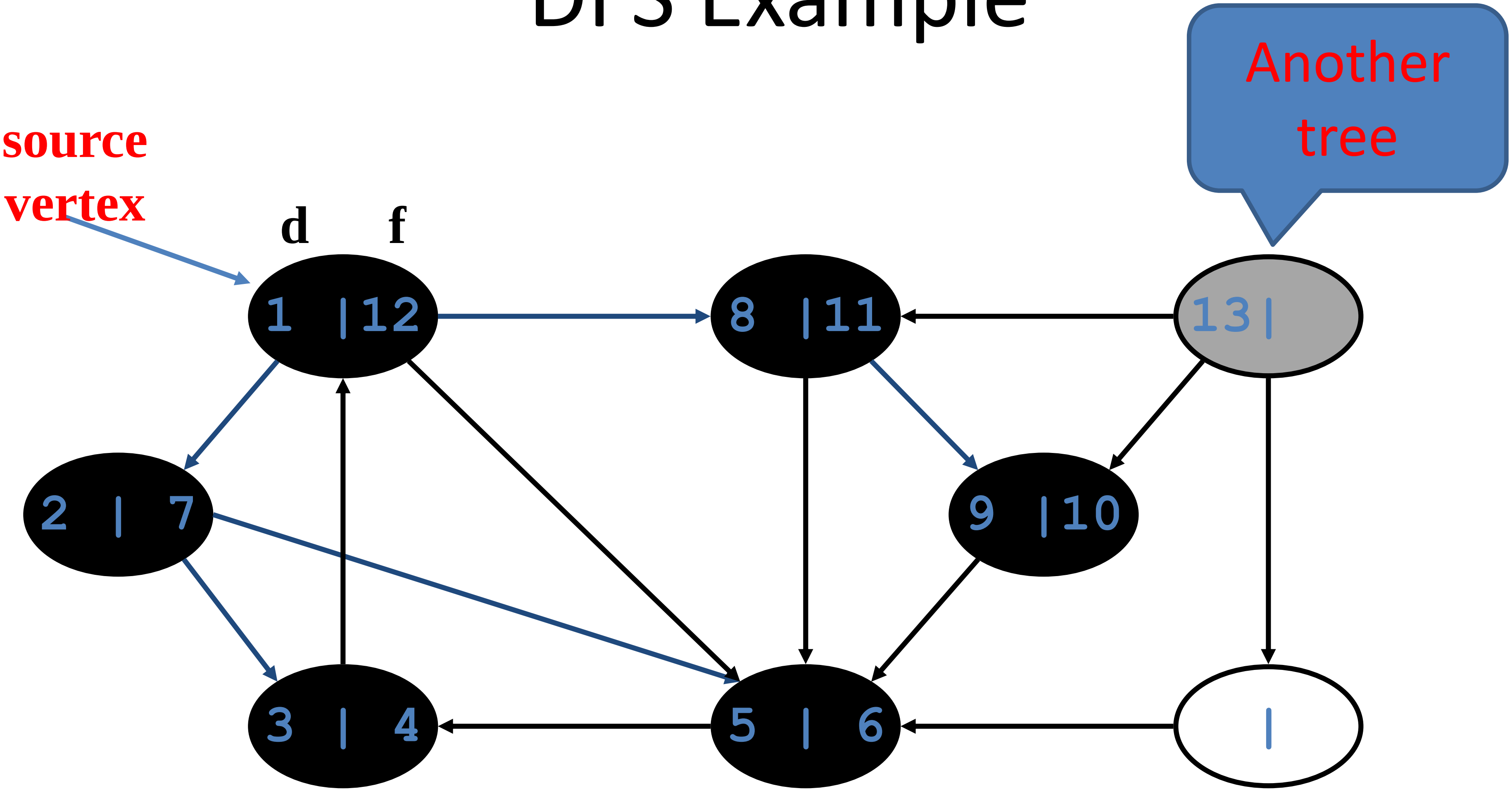
DFS Example: time = 11



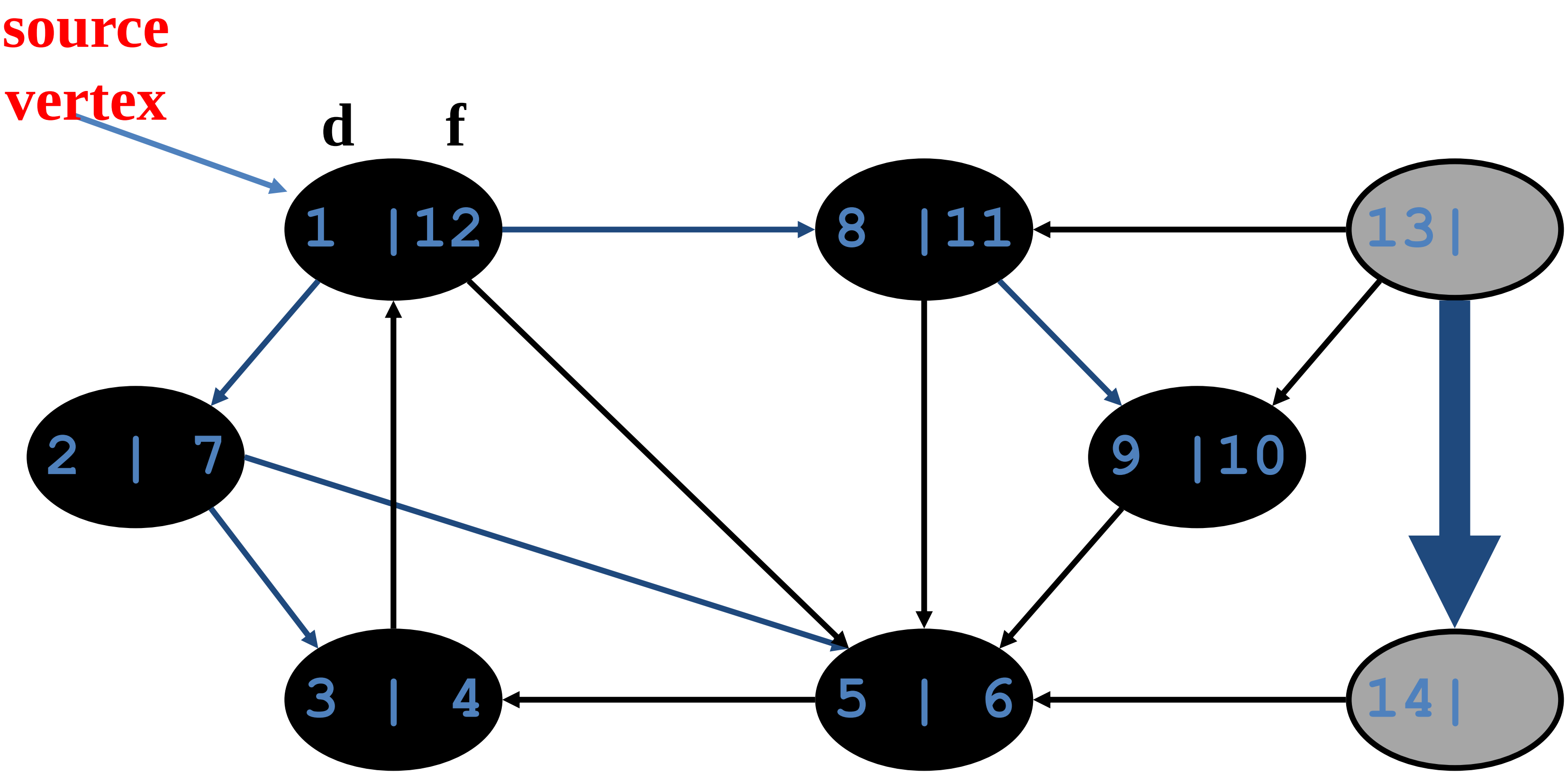
DFS Example: time = 12



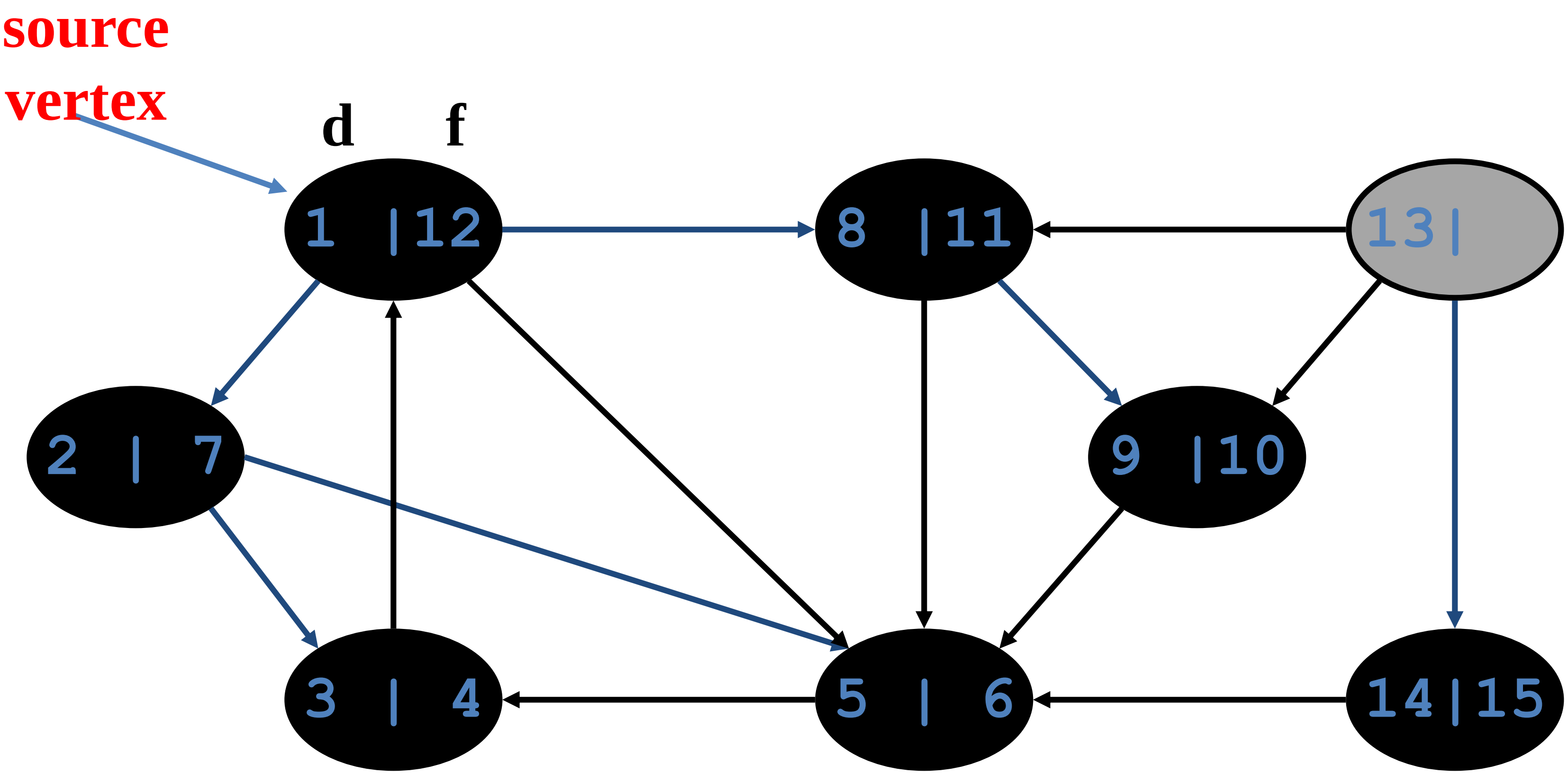
DFS Example



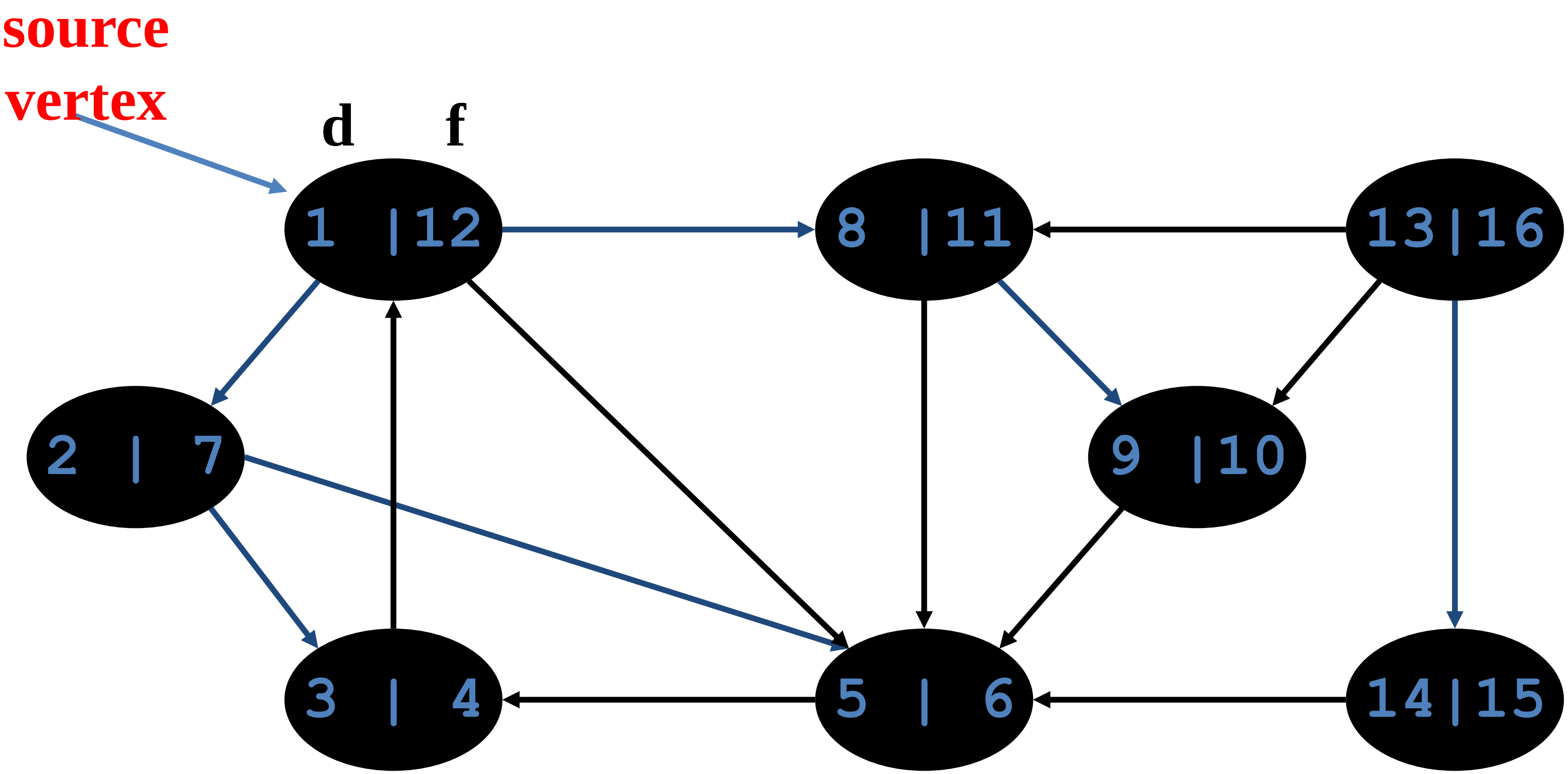
DFS Example



DFS Example



DFS Example



Depth-First Search: running time

- Running time: O($|V|^2$) because call DFS_Visit on each vertex, and the loop over Adj[] can run as many as |V| times.
- BUT, there is actually a tighter bound.

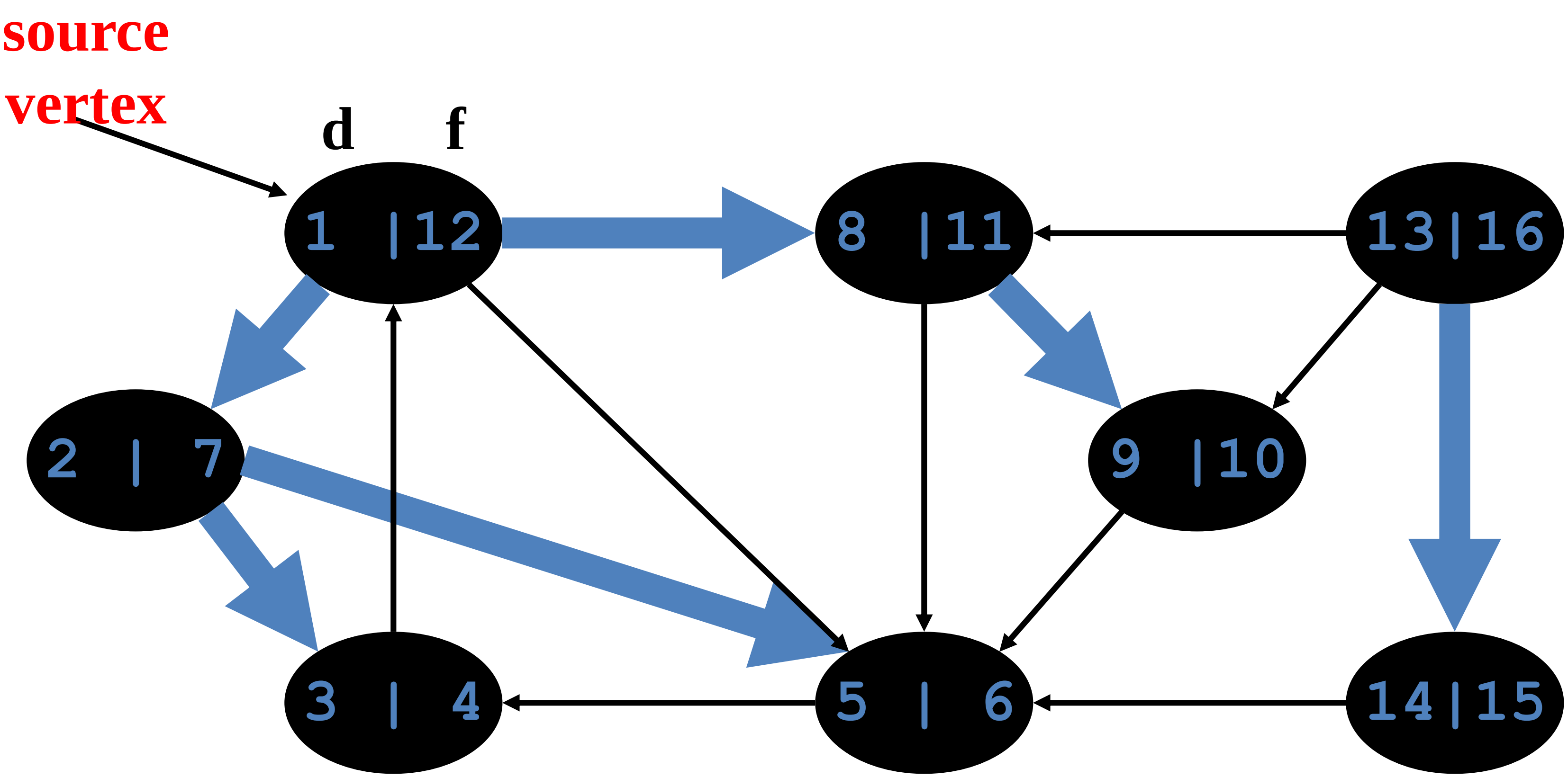
DFS: running time (cont’d)

- How many times will DFS_Visit() actually be called?
  - The loops on lines 1–3 and lines 5–7 of DFS take time Θ(V), exclusive of the time to execute the calls to DFS-VISIT.
  - DFS-VISIT is called exactly once for each vertex v
  - During an execution of DFS-VISIT*(v)*, the loop on lines 4–7 is executed $|Adj[v]|$ times.
  - $\sum_{v \in V} |Adj[v]| = \Theta(E)$
  - Total running time is $\Theta(V+E)$

DFS: Different Types of edges

- DFS introduces an important distinction among edges in the original graph:
  - Tree edge: Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v)

DFS Example: Tree edges

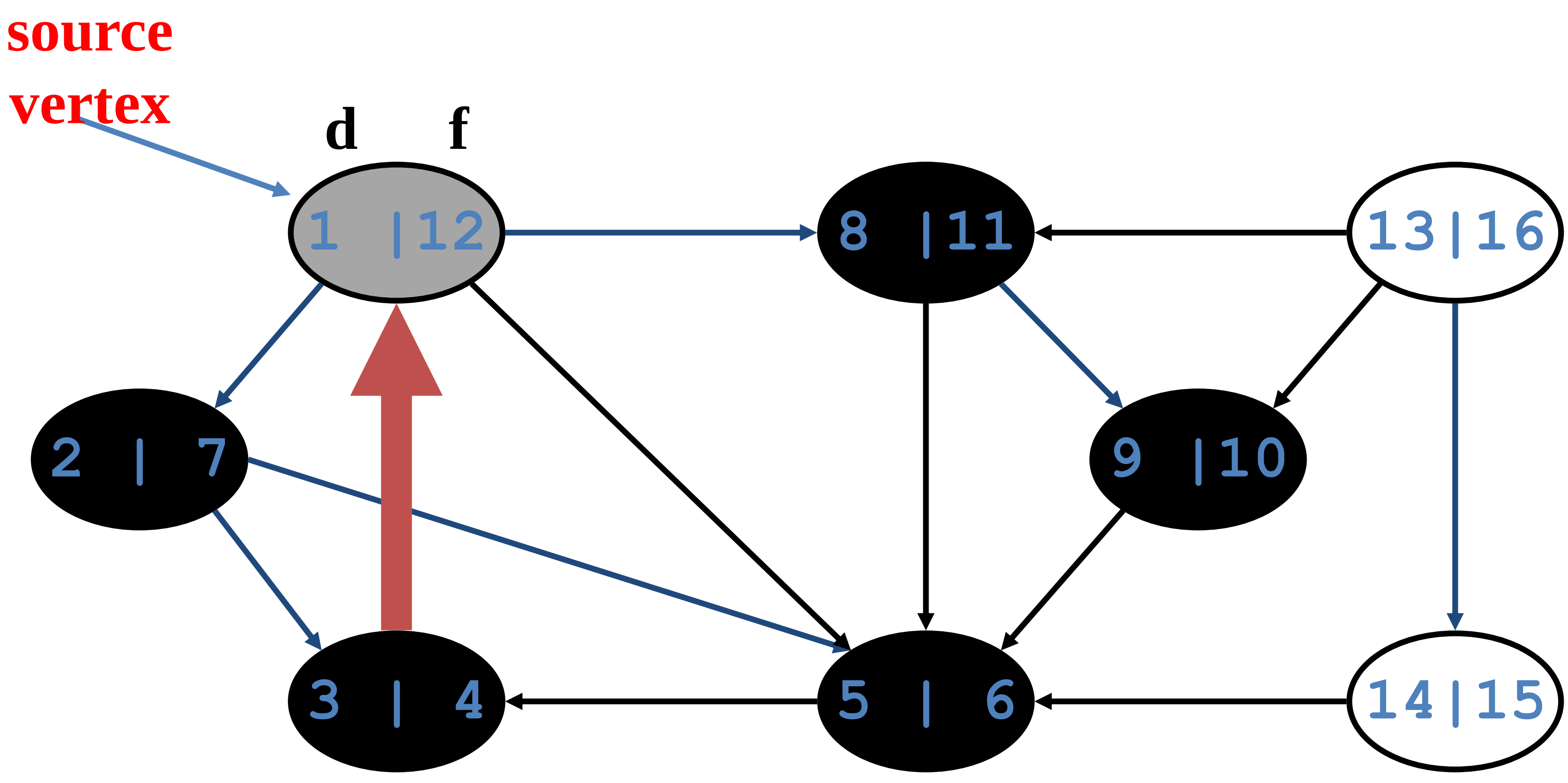


Tree edges

DFS: Different Types of edges

- DFS introduces an important distinction among edges in the original graph:
  - Tree edge: encounter new vertex
  - Back edge: from descendent to ancestor

DFS Example

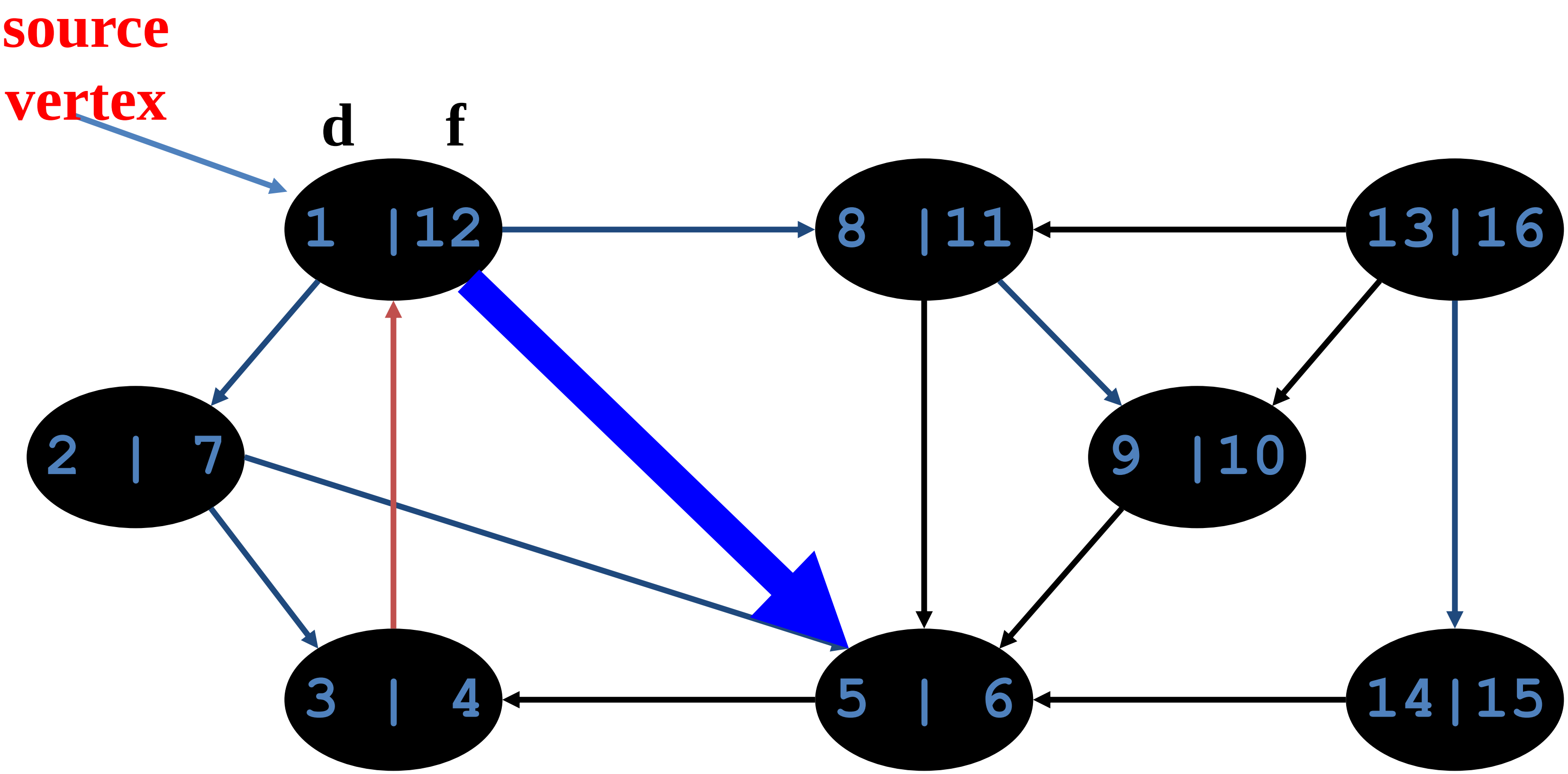


Tree edges **Back edges**

DFS: Different Types of edges

- DFS introduces an important distinction among edges in the original graph:
  - Tree edge: encounter new vertex
  - Back edge: from descendent to ancestor
  - Forward edge: from ancestor to descendent
    - Not a tree edge, though

DFS Example: **Forward edges**

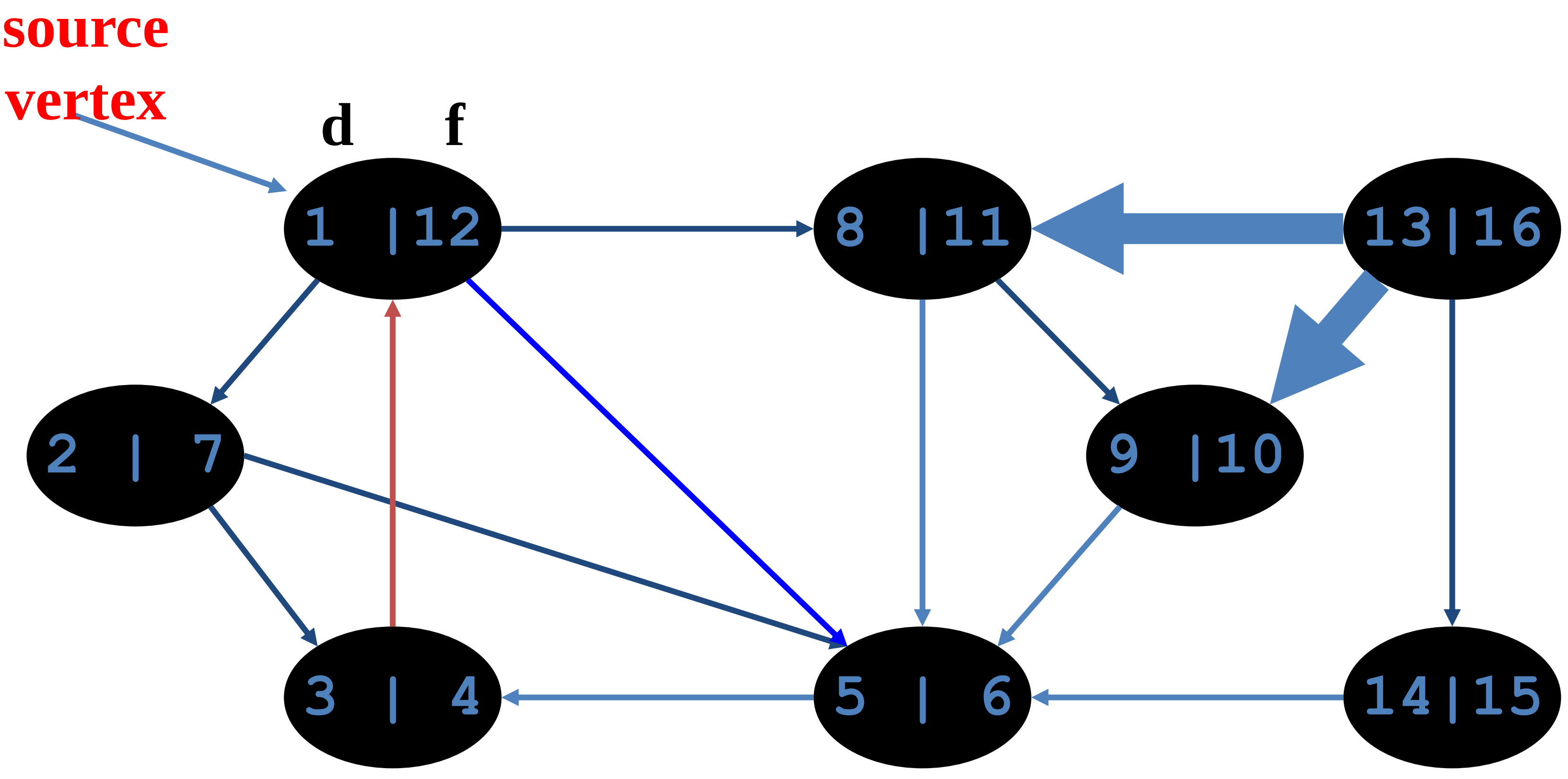


Tree edges **Back edges** **Forward edges**

DFS: Different Types of edges

- DFS introduces an important distinction among edges in the original graph:
  - Tree edge: encounter new vertex
  - Back edge: from descendent to ancestor
  - Forward edge: from ancestor to descendent
  - Cross edge: between subtrees

DFS Example



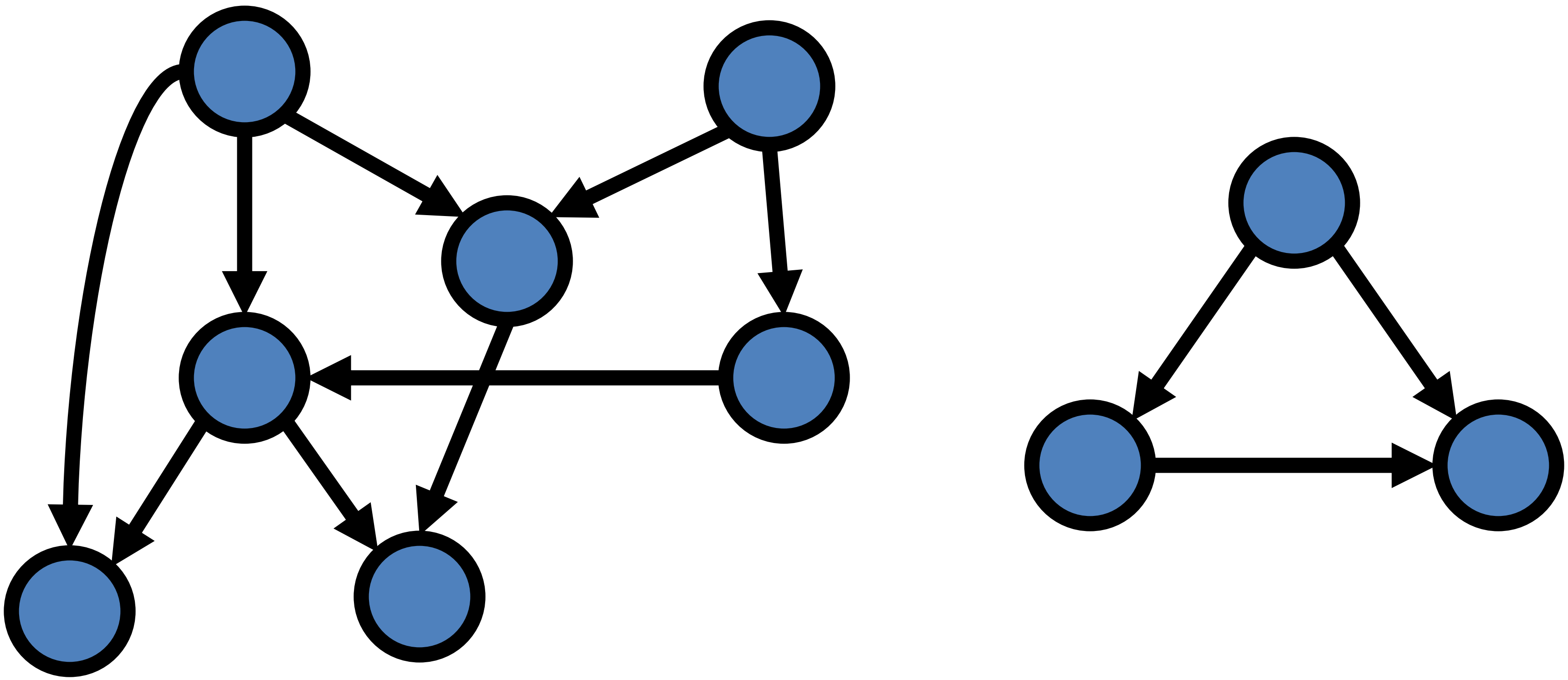
Tree edges Back edges Forward edges Cross edges

DFS: Different Types of edges

- DFS introduces an important distinction among edges in the original graph:
  - Tree edge: encounter new vertex
  - Back edge: from a descendent to an ancestor
  - Forward edge: from an ancestor to a descendent
  - Cross edge: between a tree or subtrees
- Note: tree & back edges are important
  - most algorithms don't distinguish forward & cross

Directed Acyclic Graphs

- A directed acyclic graph (DAG) is a directed graph with no directed cycles:



DFS and DAGs

- A directed graph G is acyclic i.f.f. a DFS of G yields no back edges
  - If G is acyclic: no back edges
  - If G has a cycle, there must exist a back edge
- How would you modify the DFS code to detect cycles?
  - Detect back edges
  - edge (u, v) is a back edge if and only if $d[v] < d[u] < f[u] < f[v]$
    - u is the descendent
    - v is the ancestor

Run DFS to find whether a graph has a cycle

```
DFS(G)
{
    for each vertex u ∈ G.V
    {
        u.color = WHITE
        u.π = NIL
    }
    time = 0
    for each vertex u ∈ G.V
    {
        if (u.color == WHITE)
            DFS_Visit(G, u)
    }
}
```

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

DFS and Cycles

- What will be the running time?
- A: O(V+E)
- We can actually determine if cycles exist in O(V) time:
  - In an undirected acyclic tree, $|E| \leq |V| - 1$
  - So, count the number of edges:
    - if ever see |V| distinct edges, we must have seen a back edge along the way