Algorithms and Data Structures

11th Lecture: Graph Algorithms I

Yutaka Watanobe, Jie Huang, Yan Pei, Wenxi Chen,
S. Semba, Deepika Saxena, Yinghu Zhou, Akila Siriweera

University of Aizu

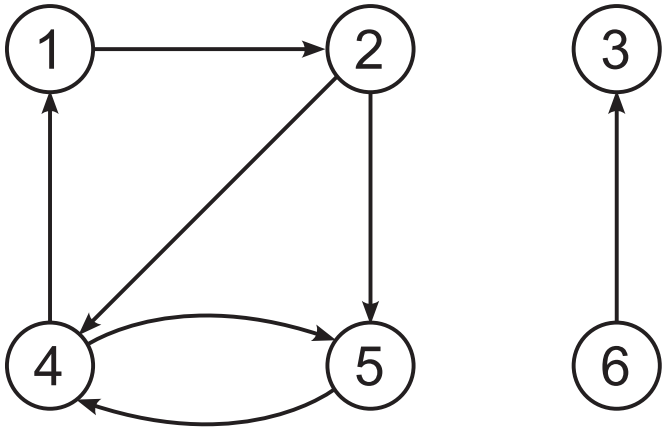
Last Updated: 2025/01/19

Outline

- Graph
- Representation of Graphs
- Depth First Search
- Breadth First Search

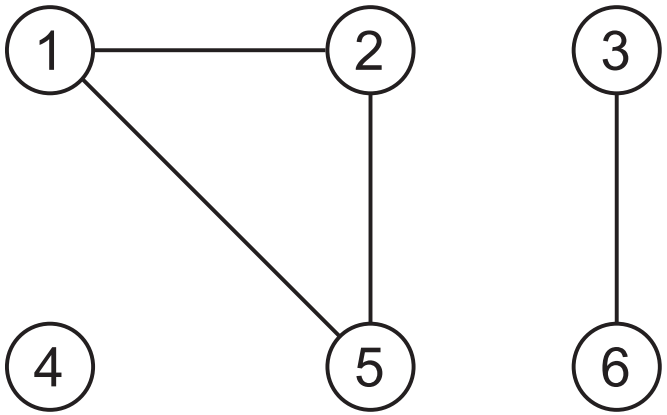
Graph: Directed Graph

- A directed graph (or digraph) G is a pair (V, E), where V is a finite set and E is a binary relation on V.
- The set V is called the vertex set of G, and its elements are called vertices (singular: vertex).
- The set E is called the edge set of G, and its elements are called edges.



Graph: Undirected Graph

- In an undirected graph $G = (V, E)$, the edge set E consists of unordered pairs of vertices, rather than ordered pairs.
- That is, an edge is a set $\{u, v\}$, where $u, v \in V$ and $u \neq v$.
- By convention, we use the notation (u, v) for an edge, and (u, v) and (v, u) are considered to be the same edge.
- In an undirected graph, self-loops are forbidden, and so every edge consists of exactly two distinct vertices.

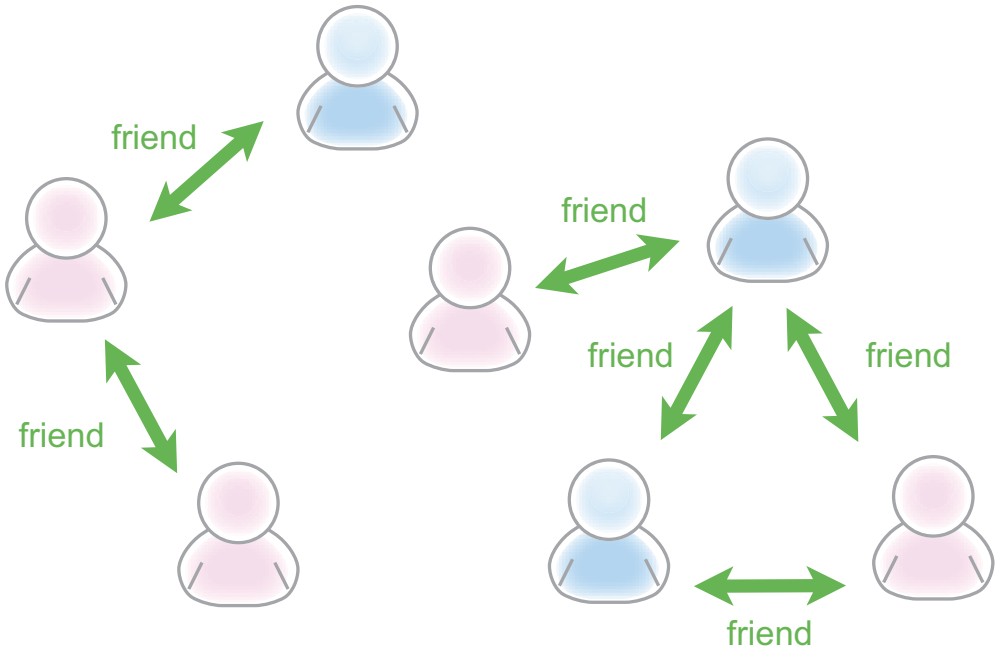


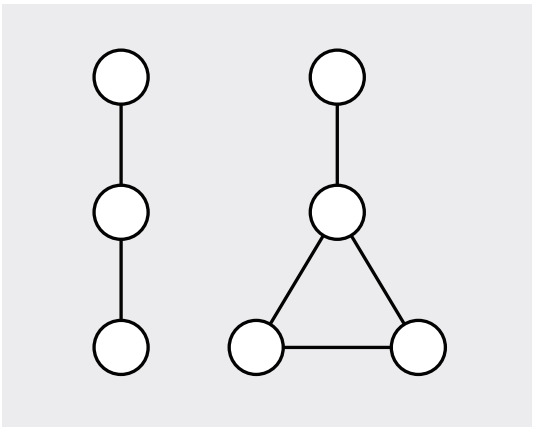
Graph: Data Structure

- Graphs are pervasive data structure in computer science, and algorithms for working with them are fundamental to the field.
- There are hundreds of interesting computational problems defined in terms of graphs.
  - Routing and analysis for the network.
  - The distance between specific places in a road map.
  - Representing relationships between people in SNS.
  - Molecules in chemistry and physics.
  - Scheduling for software development.
  - etc.

Graph: Examples (1)

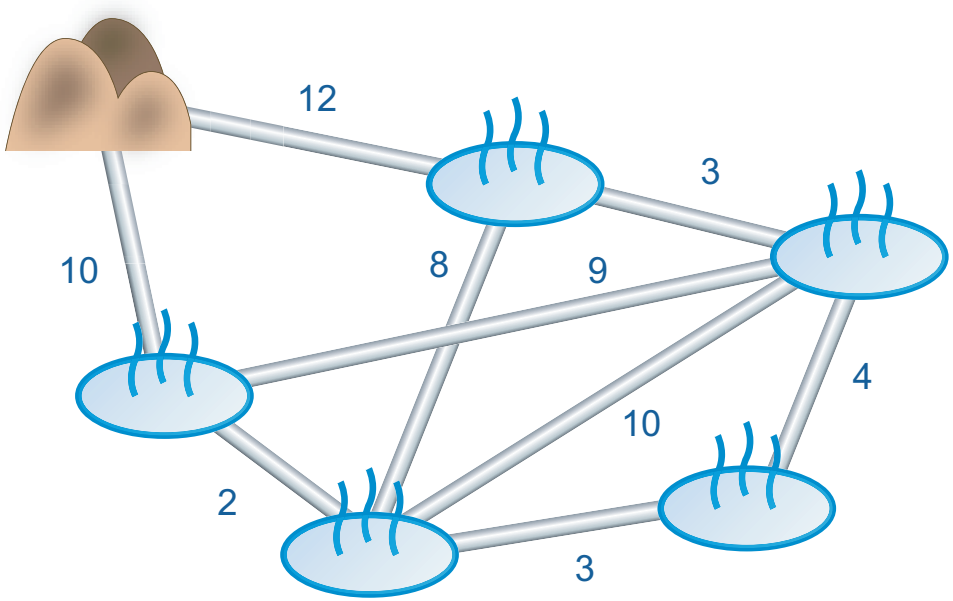
Follower relationships within a social network

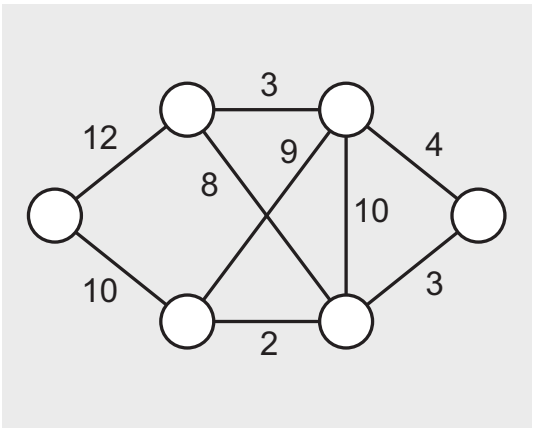




Graph: Examples (2)

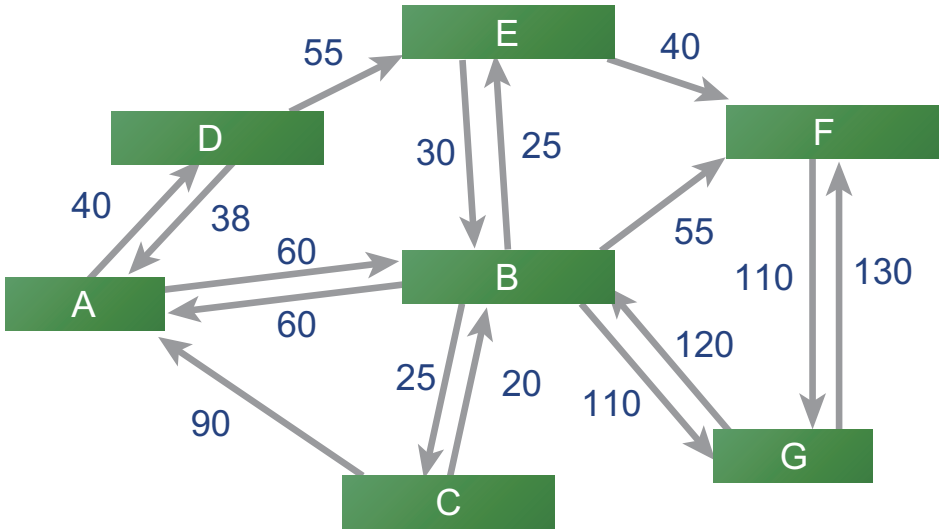
Composition of pipes connecting critical points

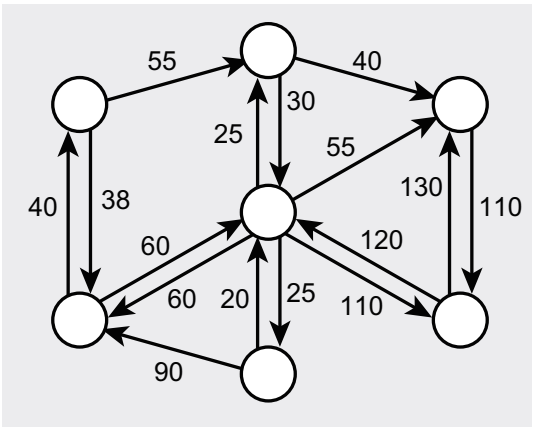




Graph: Examples (3)

Road network connecting cities





Representation of Graphs

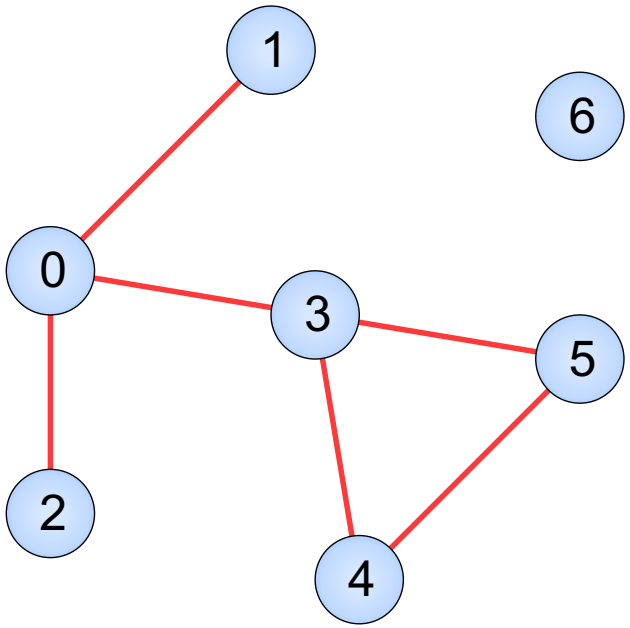
- There are two standard ways to represent a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges.
  1. Adjacency matrix representation
  2. Adjacency list representation

Adjacency Matrix Representation

- For the adjacency-matrix representation of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, ..., |V|$ in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

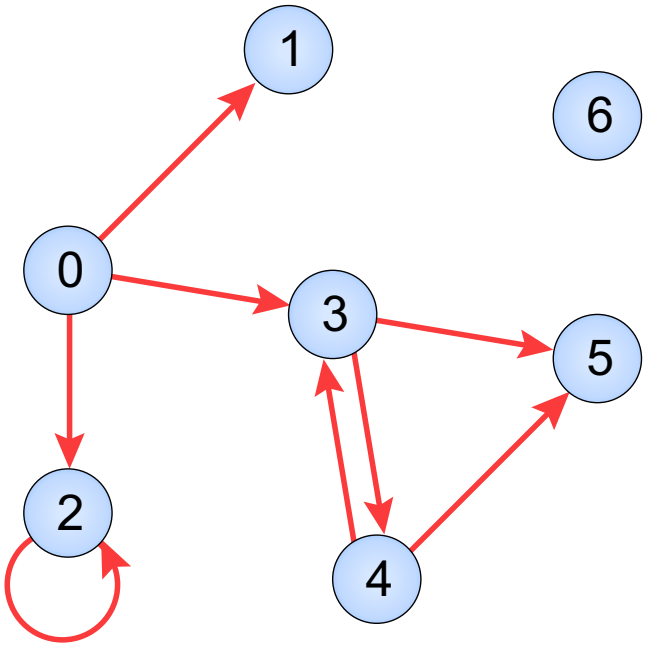
$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Adjacency Matrix Representation: Undirected Graph



i \ j	0	1	2	3	4	5	6
0	0	1	1	1	0	0	0
1	1	0	0	0	0	0	0
2	1	0	0	0	0	0	0
3	1	0	0	0	1	1	0
4	0	0	0	1	0	1	0
5	0	0	0	1	1	0	0
6	0	0	0	0	0	0	0

Adjacency Matrix Representation: Directed Graph

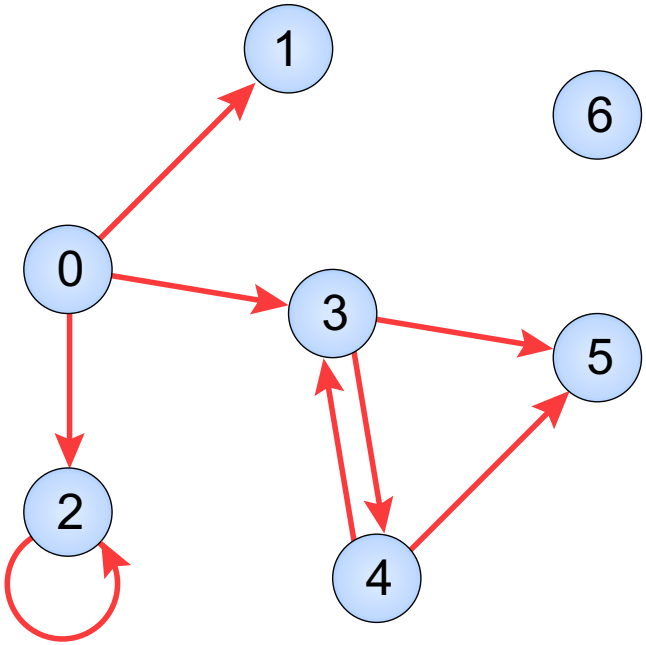


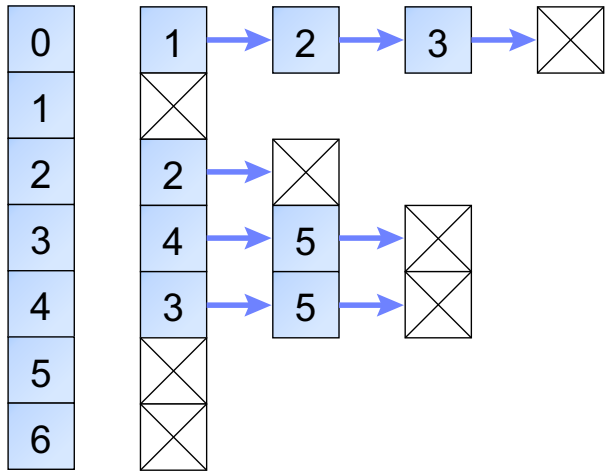
i \ j	0	1	2	3	4	5	6
0	0	1	1	1	0	0	0
1	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0
3	0	0	0	0	1	1	0
4	0	0	0	1	0	1	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0

Adjacency List Representation

- The adjacency-list representation of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V.
- For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$.
- That is, $Adj[u]$ consists of all the vertices adjacent to u in G.

Adjacency List Representation: Directed Graph





Representation of Graphs: Features

1. Adjacency matrix representation
   - The presence or absence of an edge between two points can be checked in O(1).
   - Edges between two points can be deleted in O(1).
   - Requires memory proportional to the square of the graph size.
2. Adjacency list representation
   - Requires memory only proportional to the size of the edges of the graph.
   - To check for the existence of an edge between two points, it is necessary to search the list.

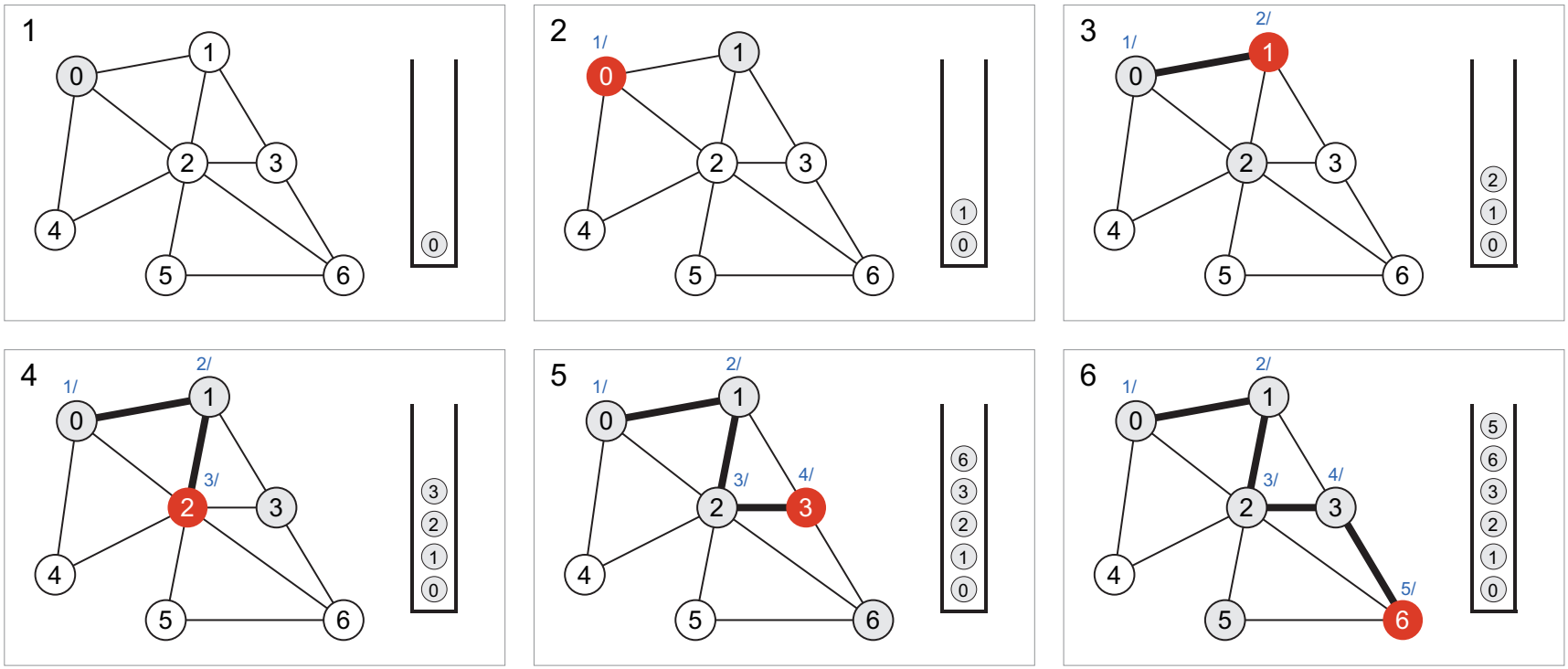
Depth First Search (1)

- The strategy followed by depth-first search (DFS) is to search ”deeper” in the graph whenever possible.
- In DFS, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it.
- When all of v’s edges have been explored, the search ”backtracks” to explore edges leaving the vertex from which v was discovered.
- This process continues until we have discovered all the vertices that are reachable from the original source vertex.
- If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source.
- This entire process is repeated until all vertices are discovered.

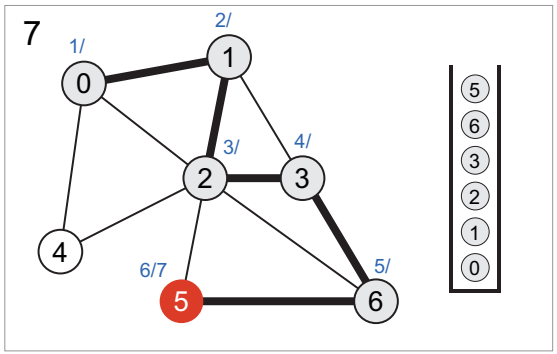
Depth First Search (2)

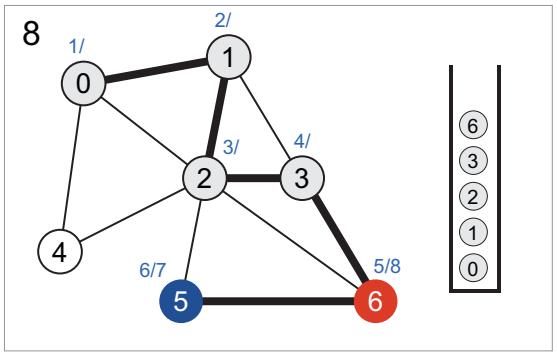
- DFS timestamps each vertex.
- Each vertex v has two timestamps:
  - $d[v]$ records when v is first discovered.
  - $f[v]$ records when the search finishes examining v's adjacency list.
- These timestamps are used in many graph algorithms and are generally helpful in reasoning about the behavior of DFS.

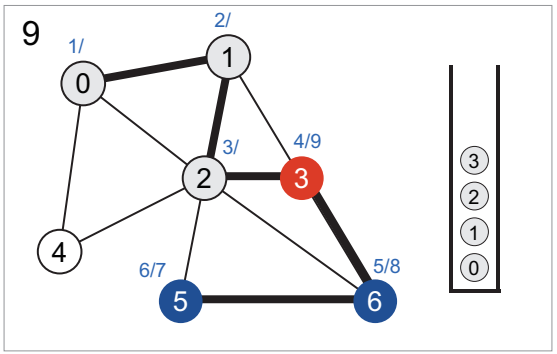
Depth First Search (3)

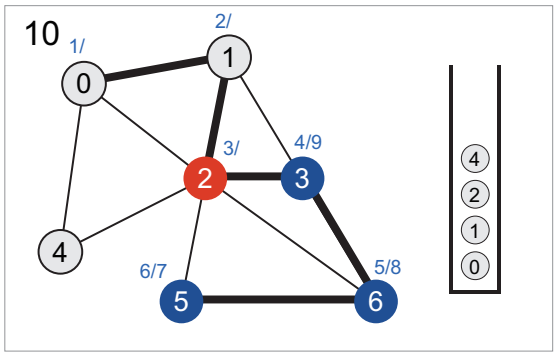


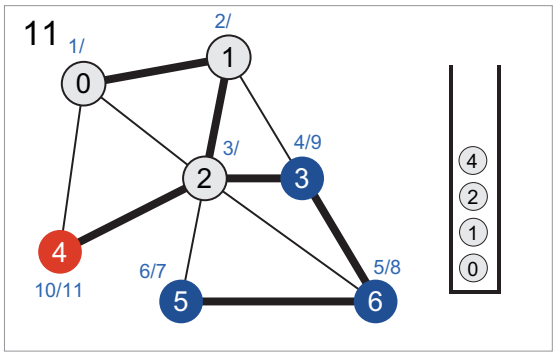
Depth First Search (4)

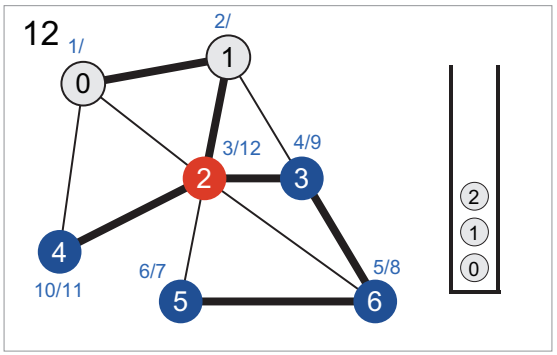




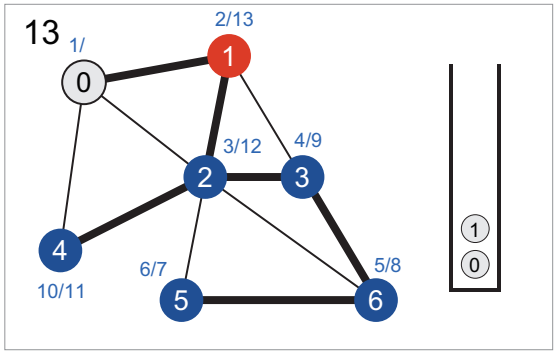


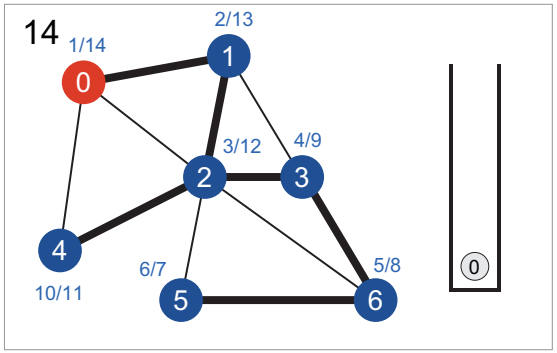


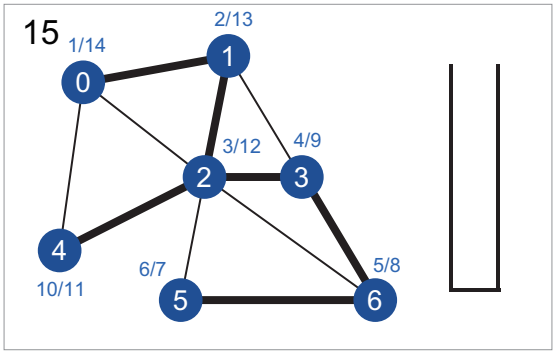




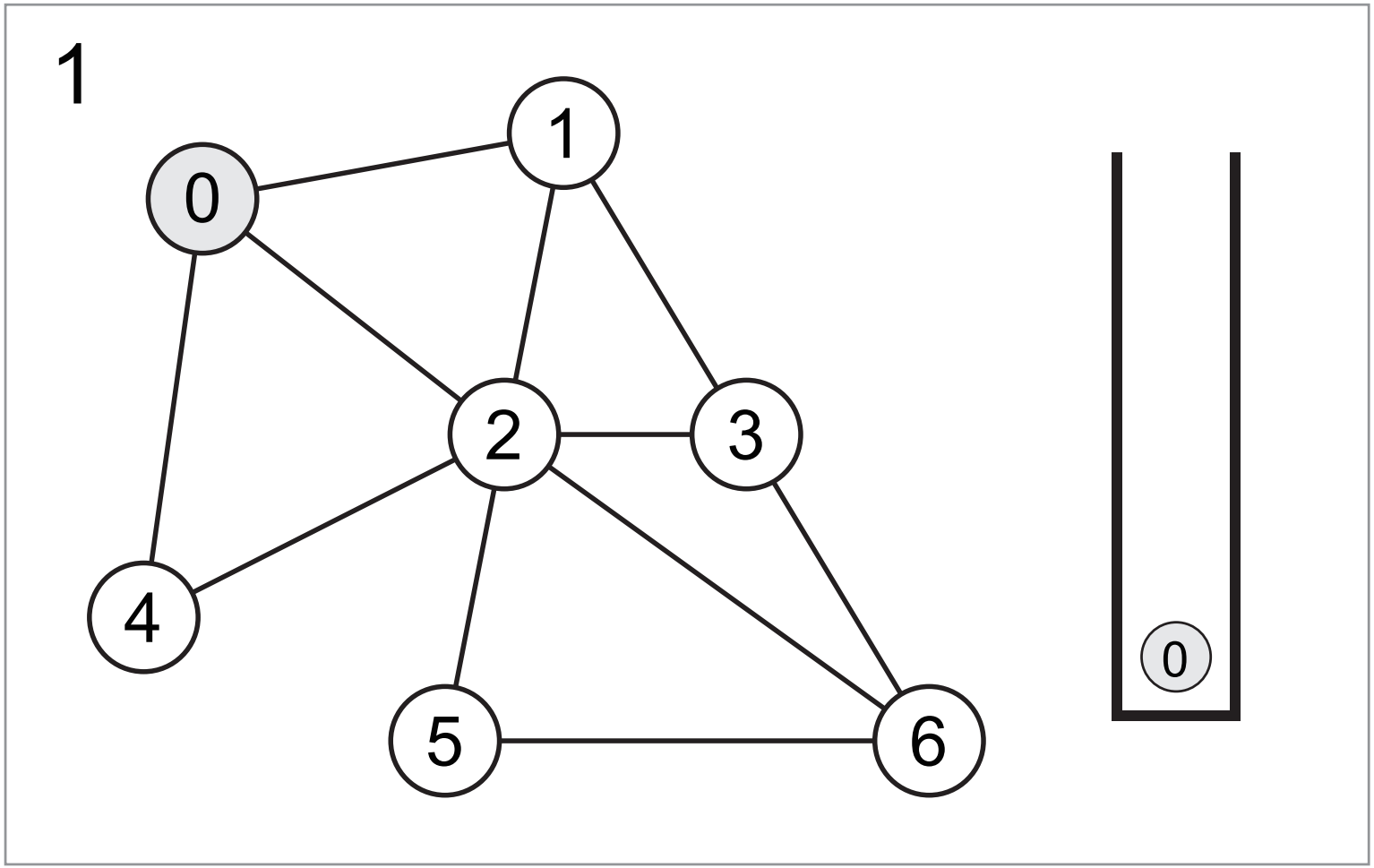
Depth First Search (5)



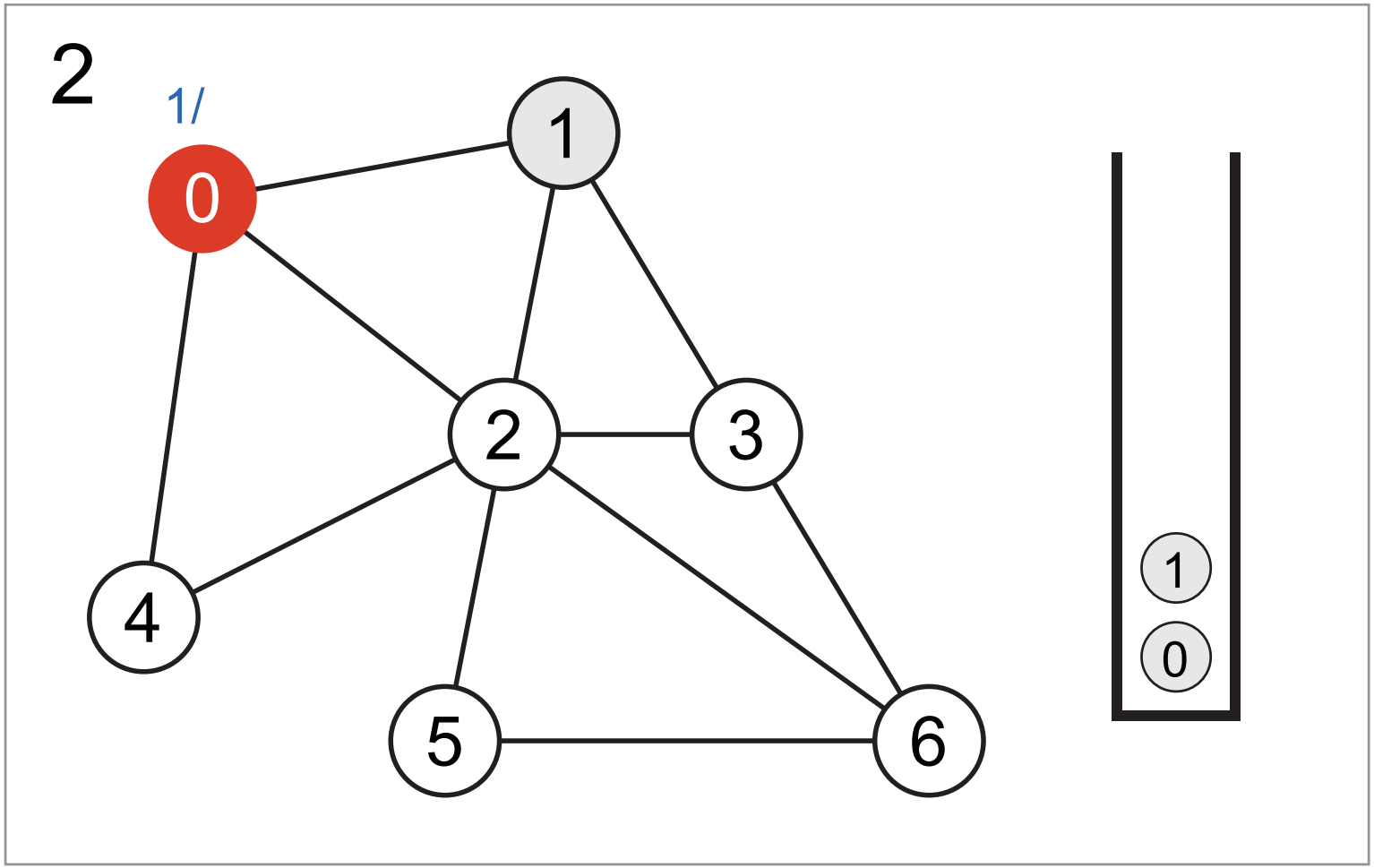




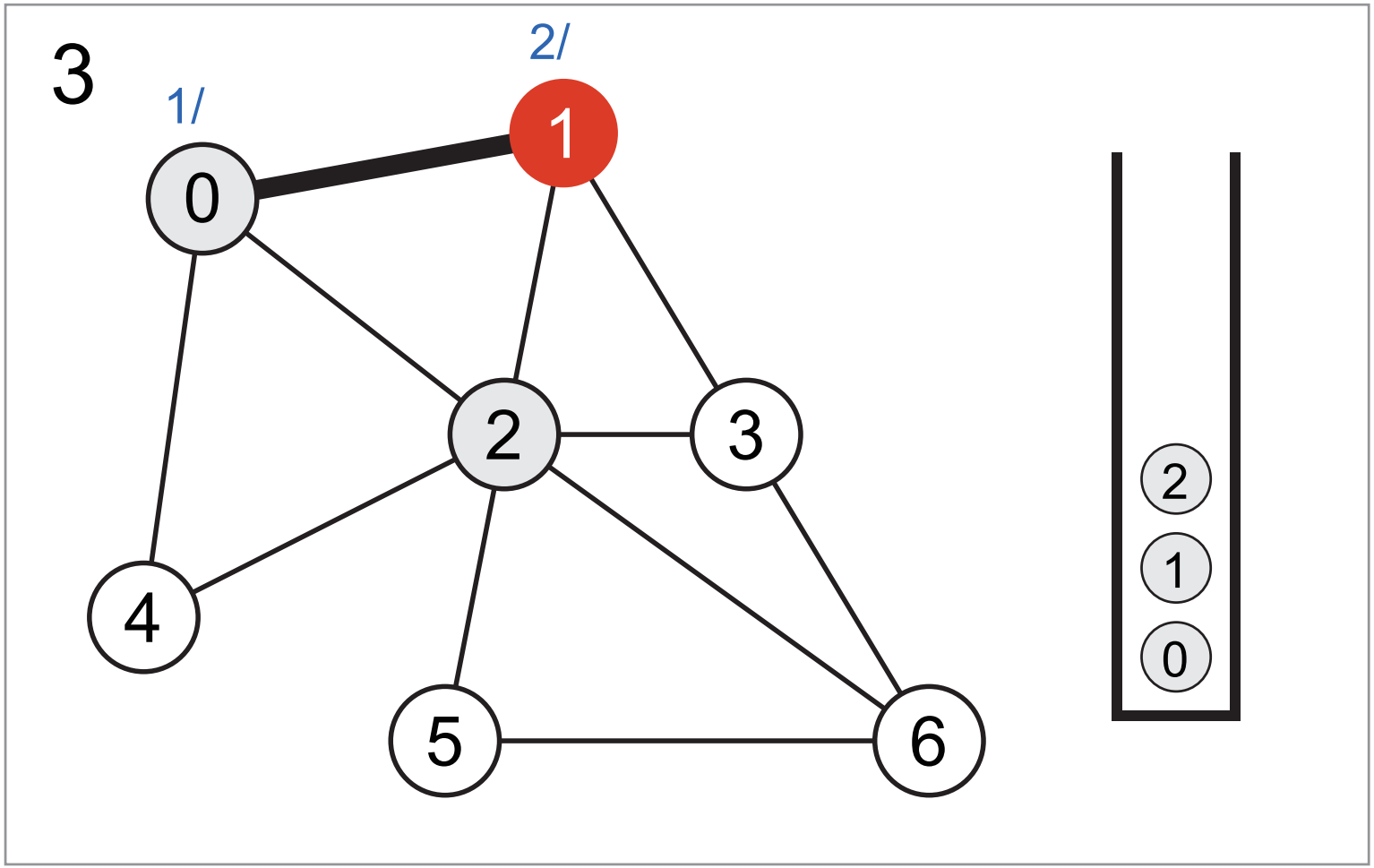
DFS Step 1



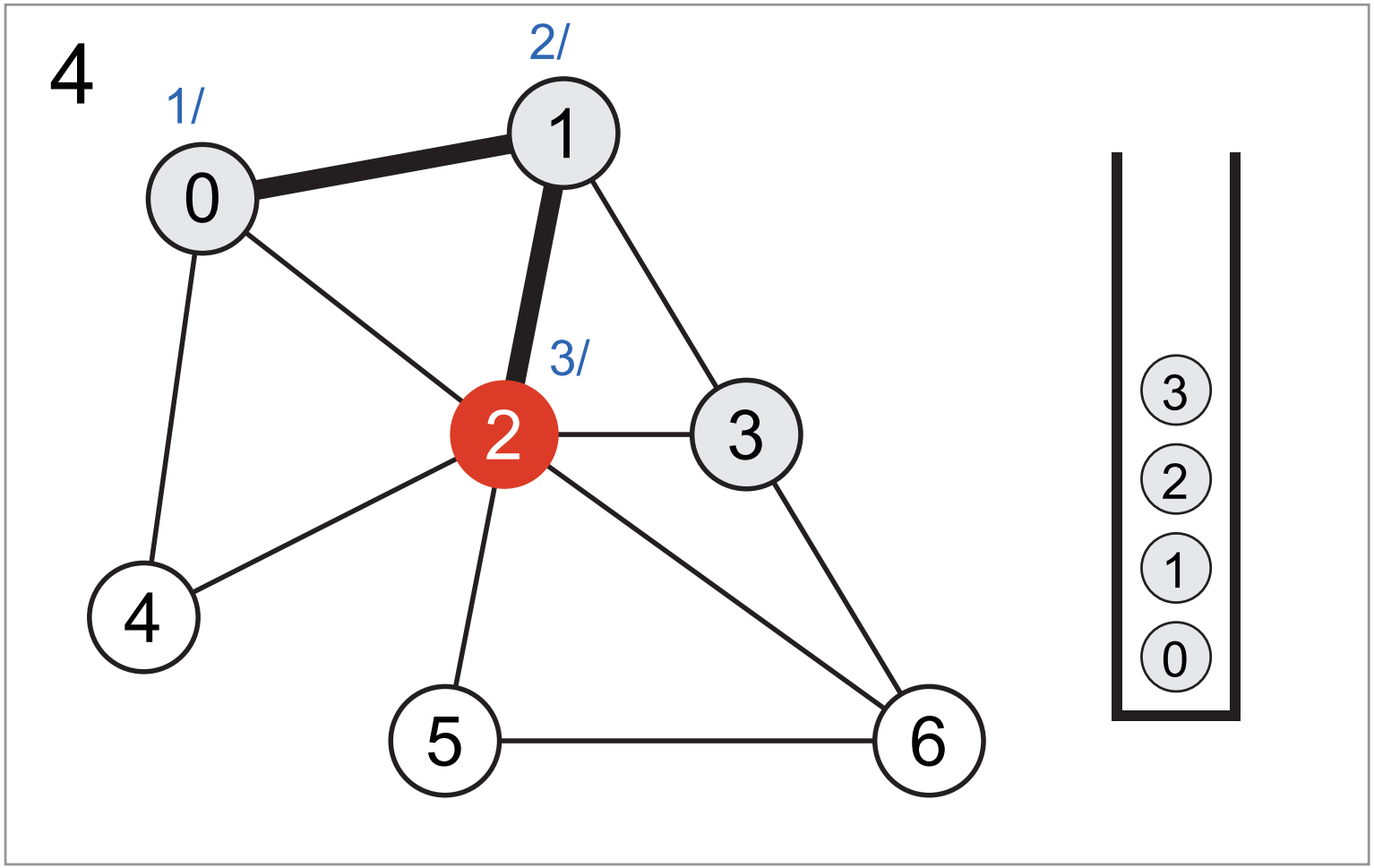
DFS Step 2



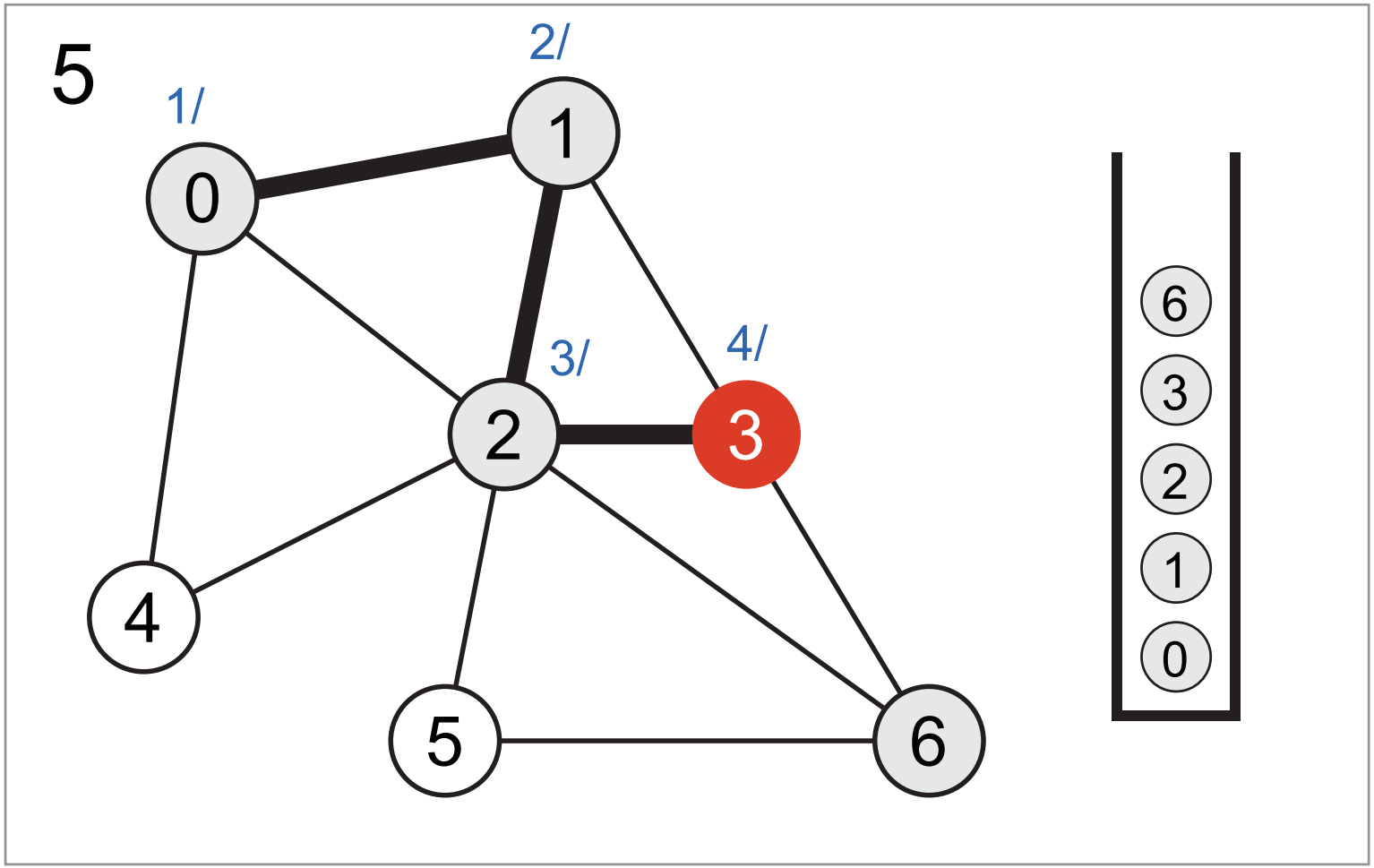
DFS Step 3



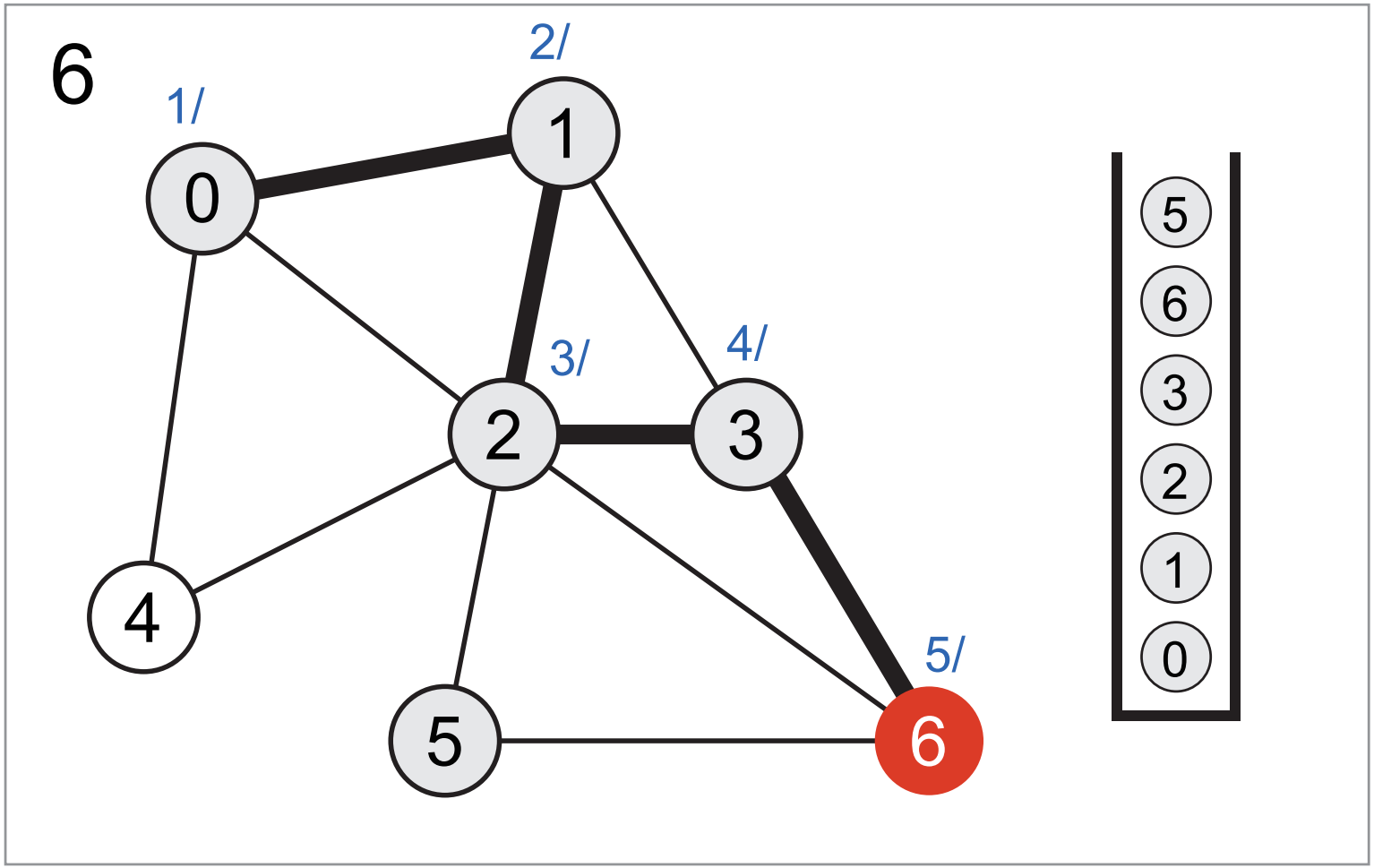
DFS Step 4



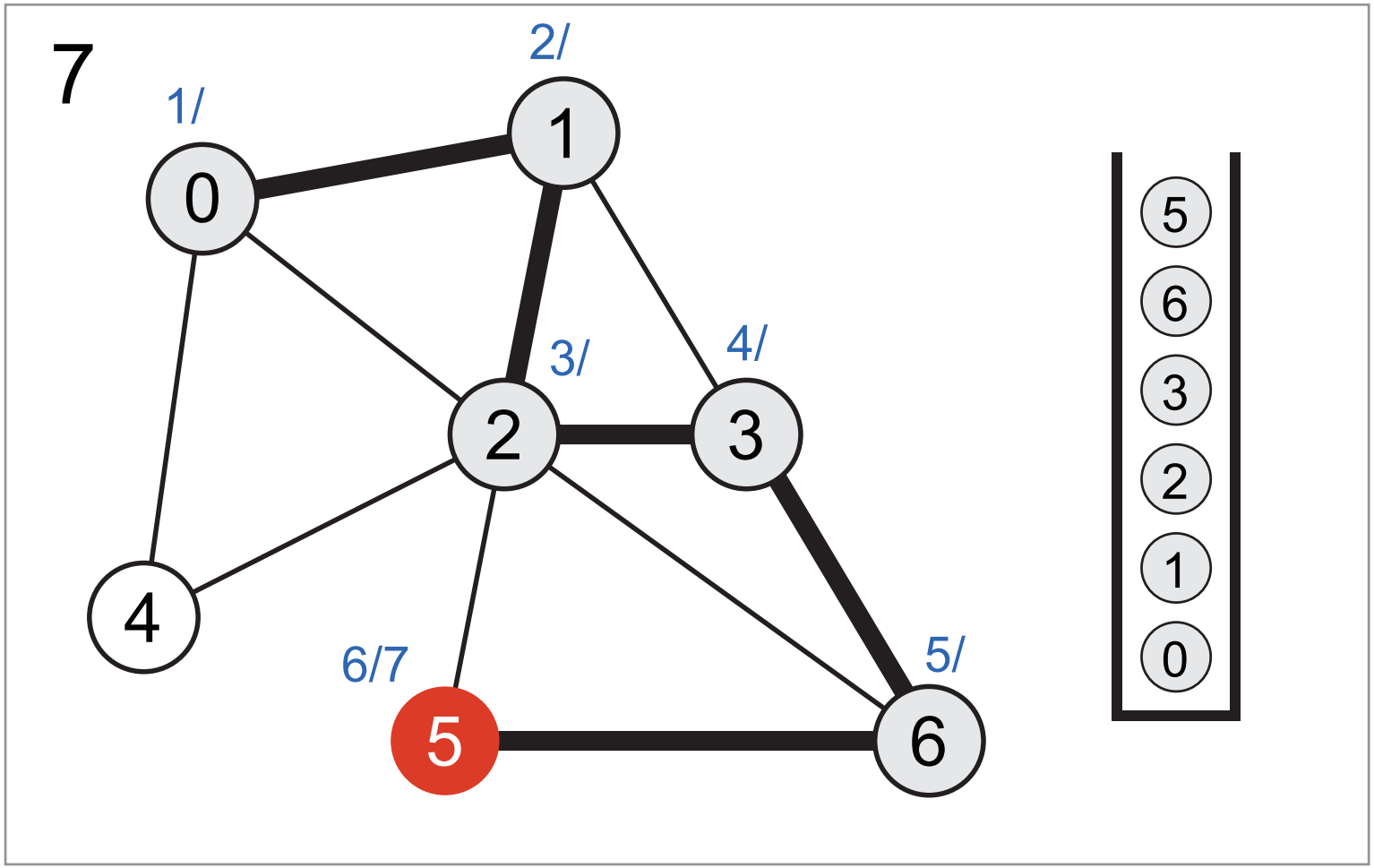
DFS Step 5



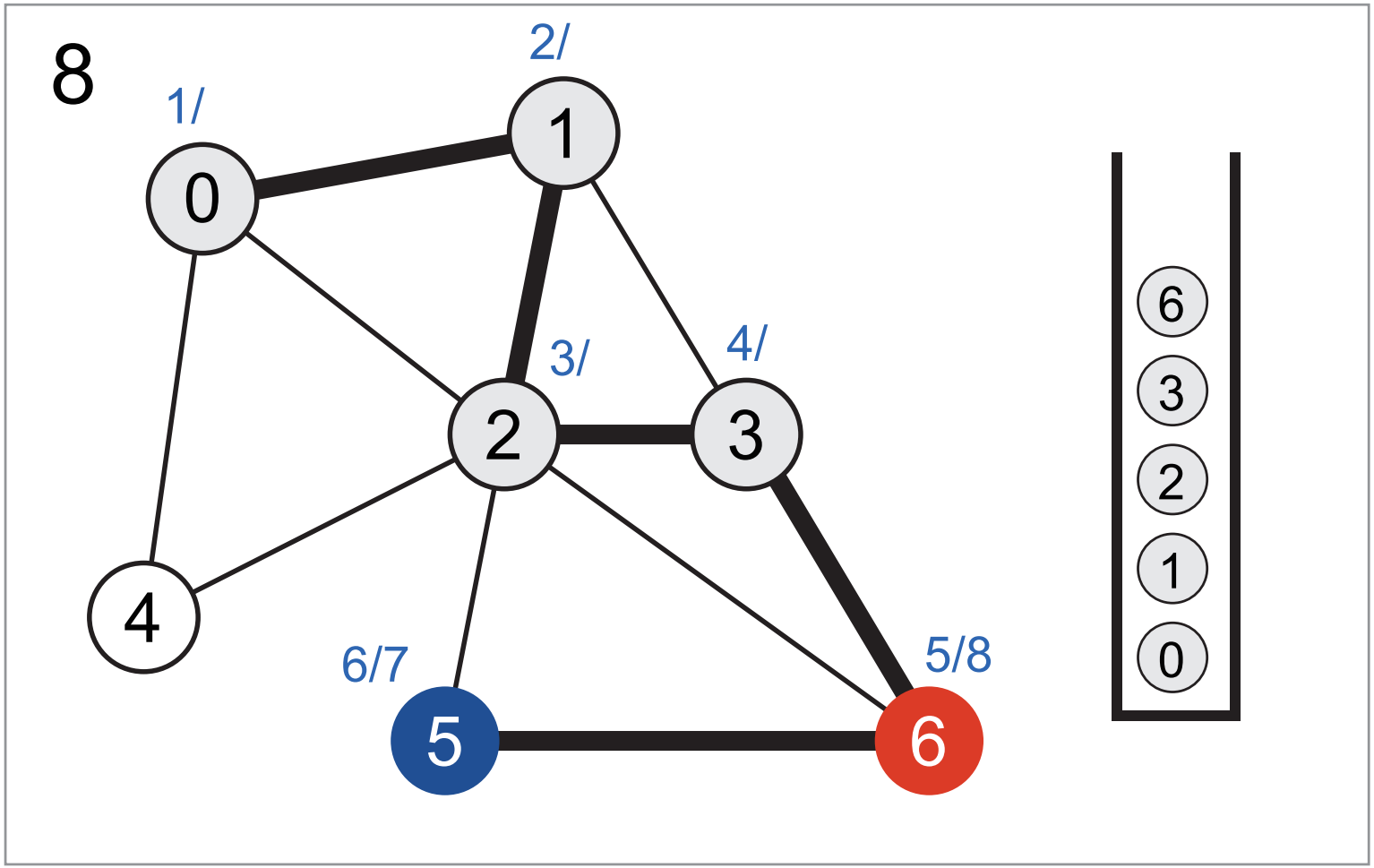
DFS Step 6



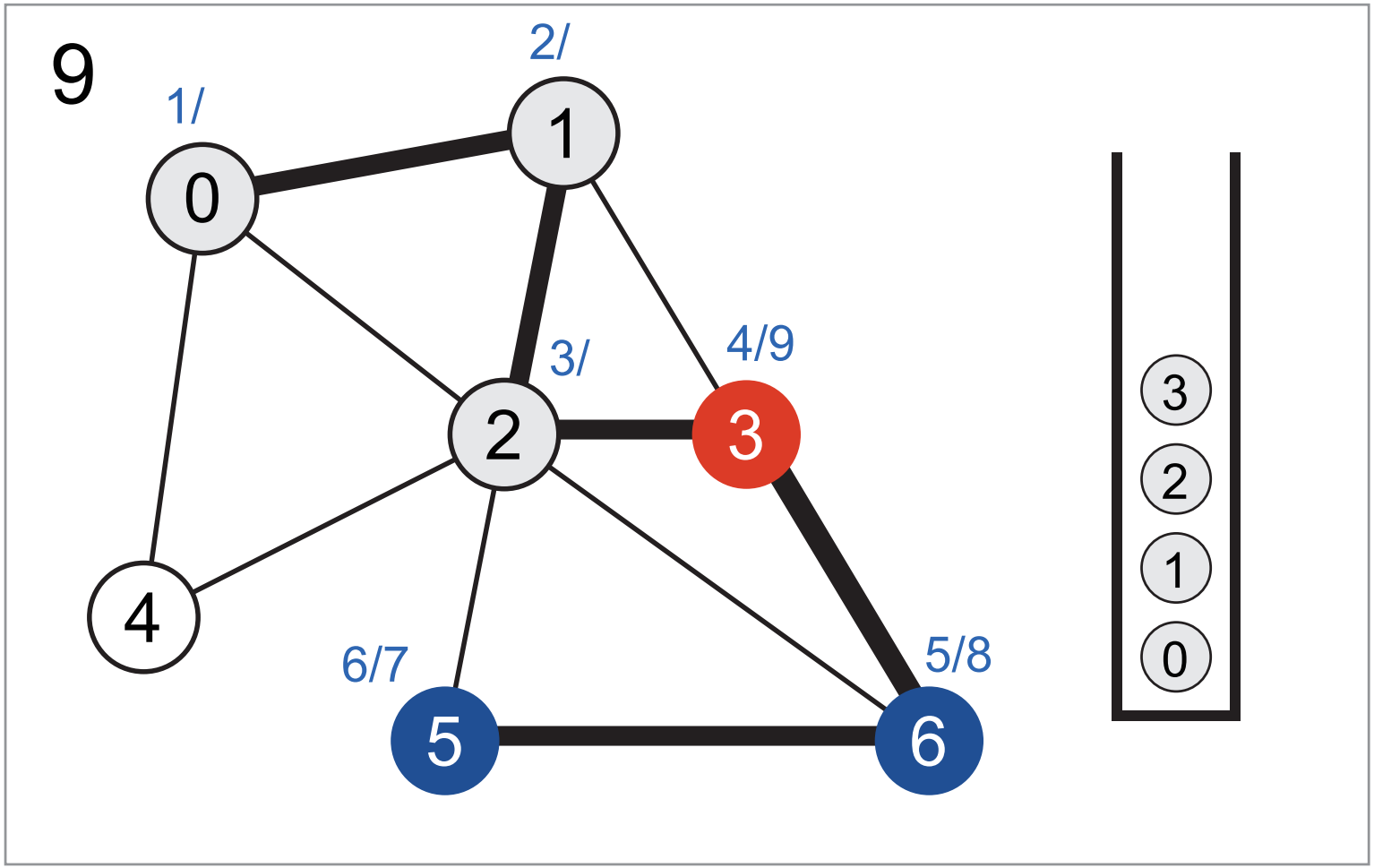
DFS Step 7



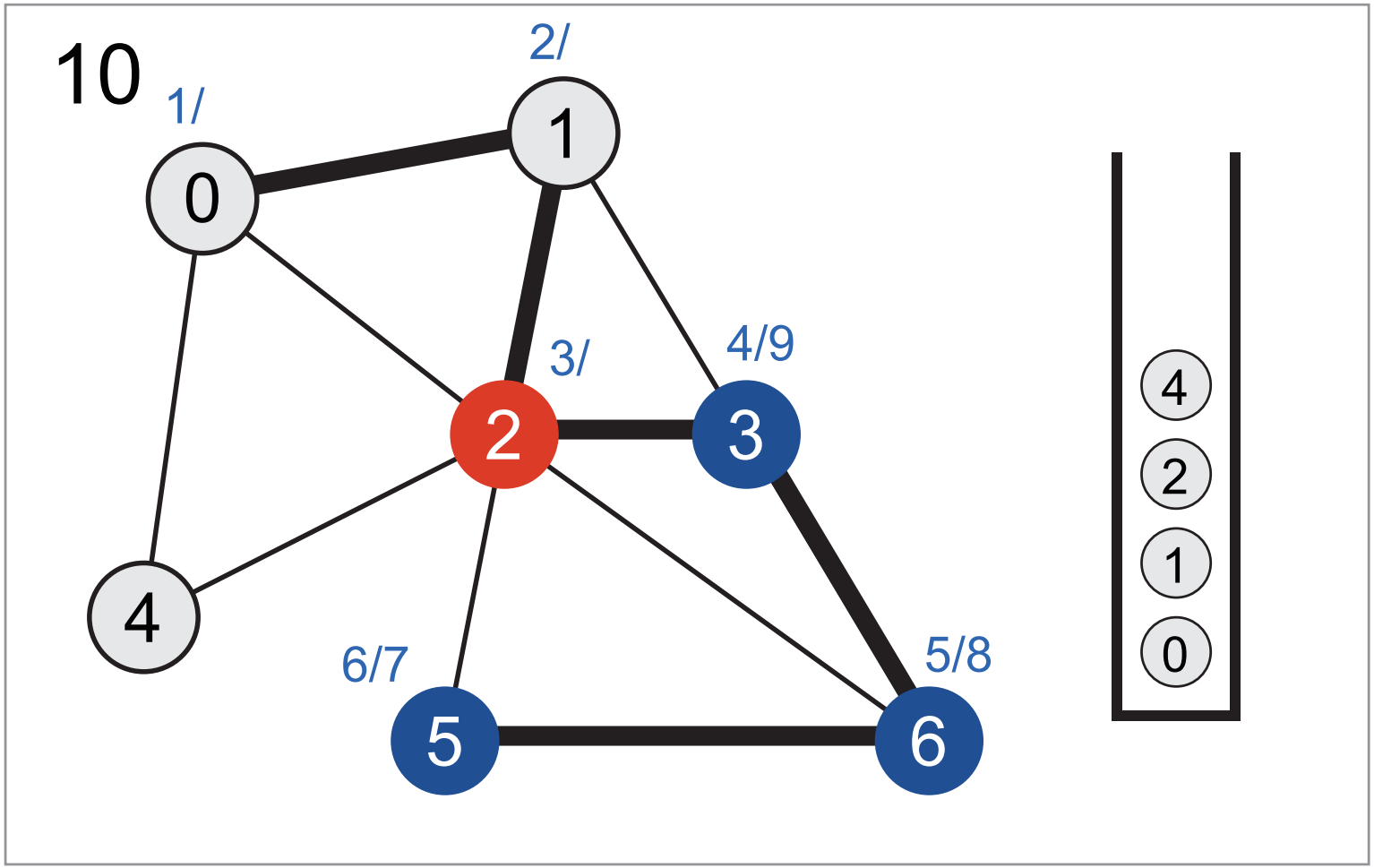
DFS Step 8



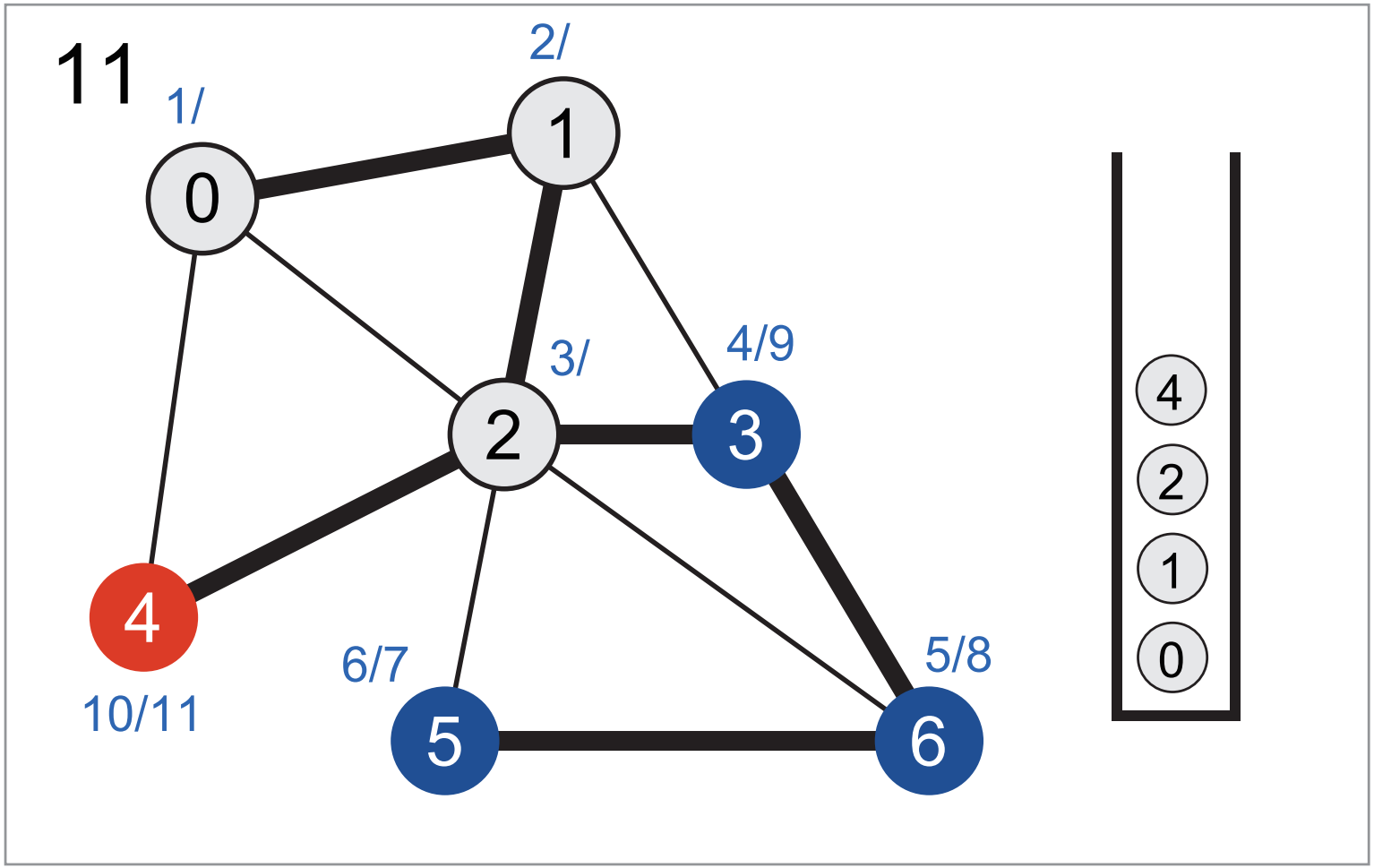
DFS Step 9



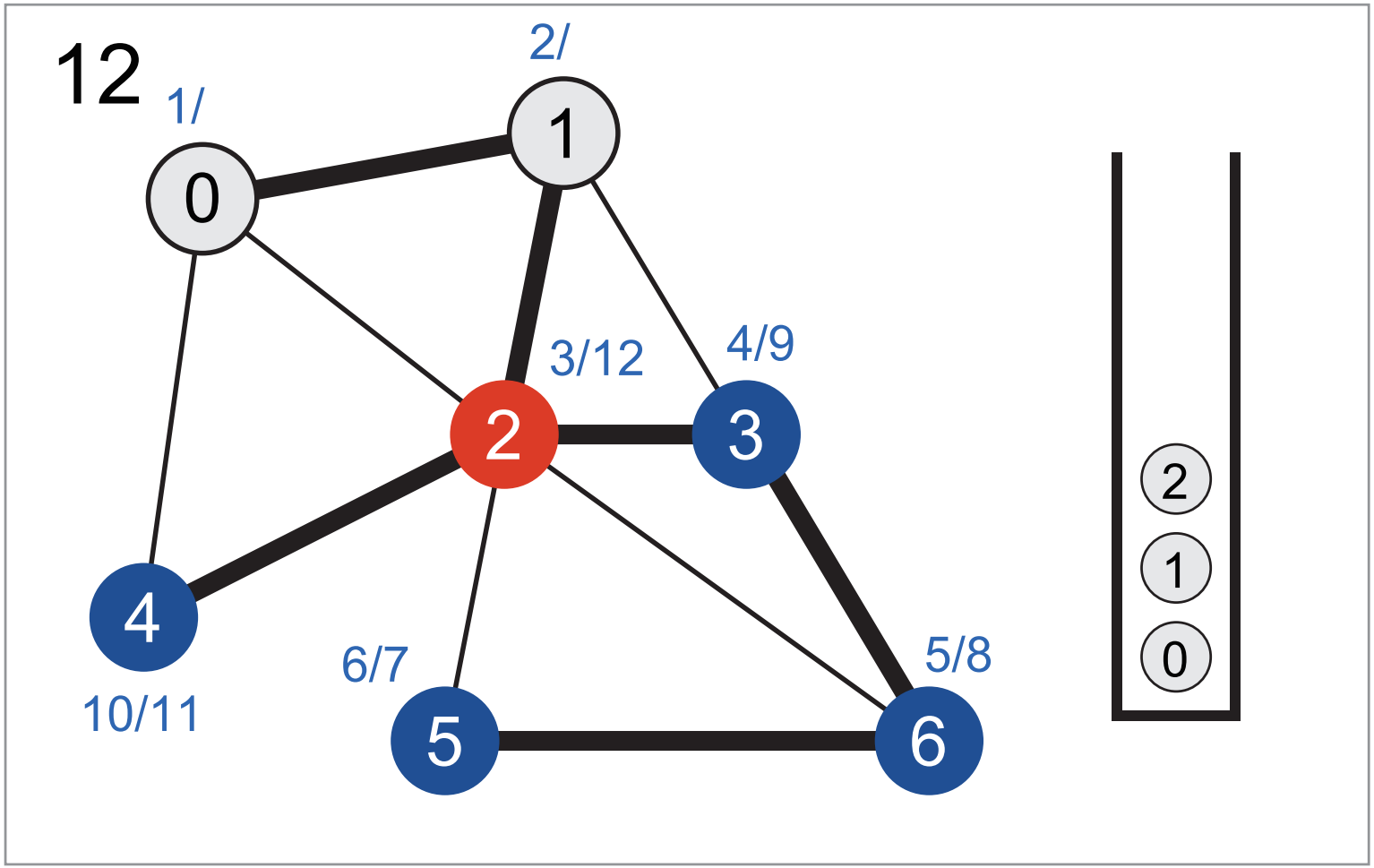
DFS Step 10



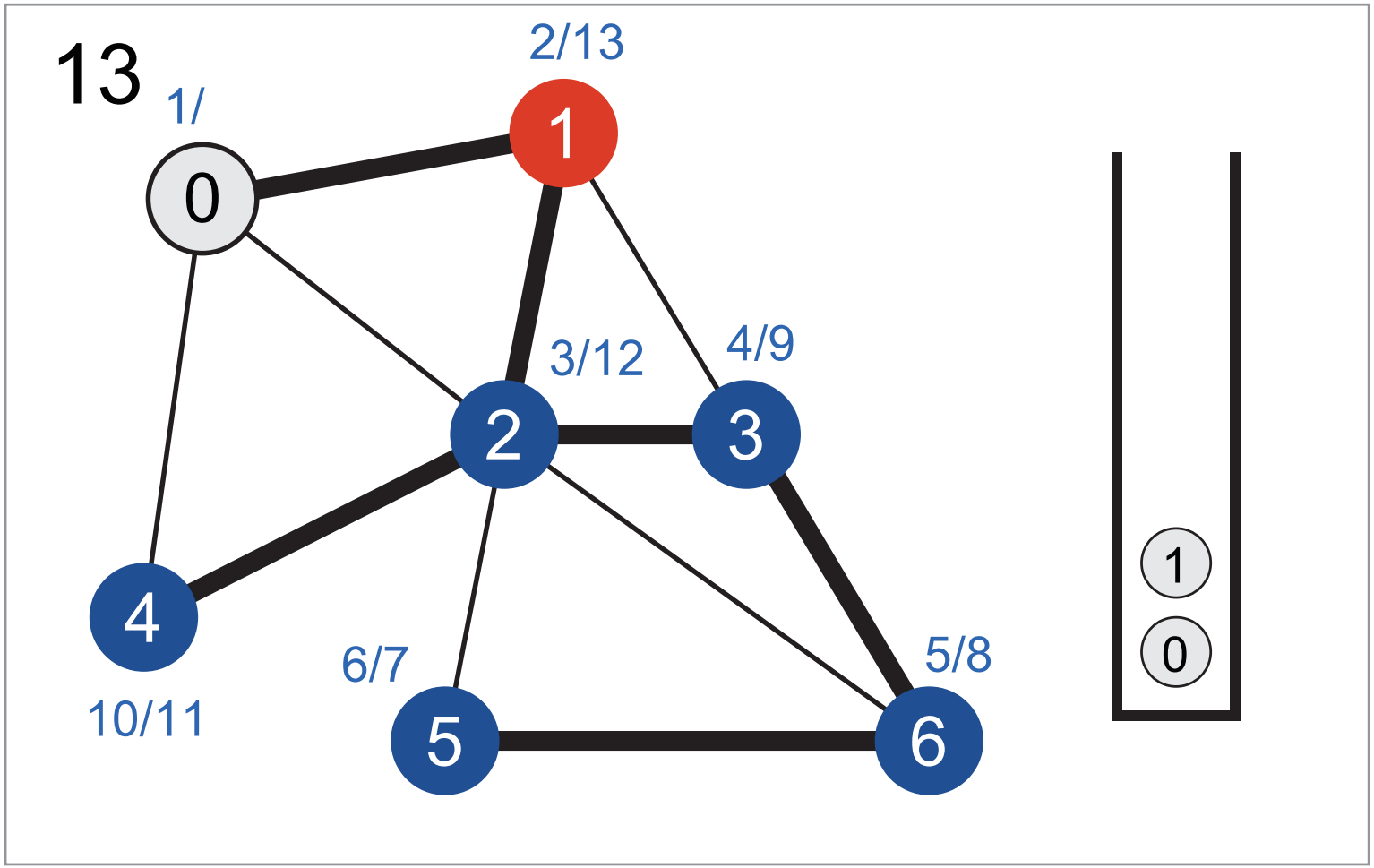
DFS Step 11



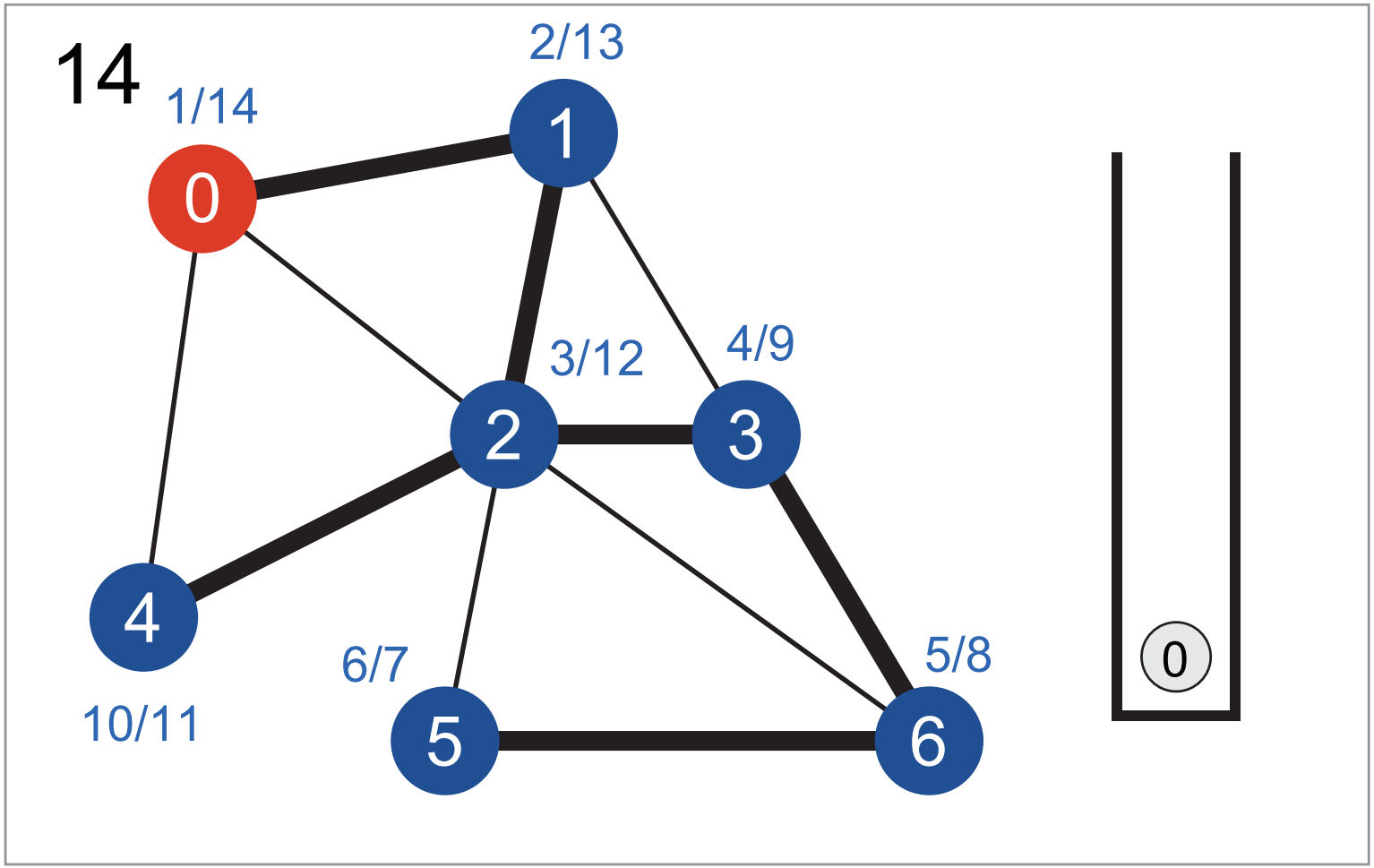
DFS Step 12



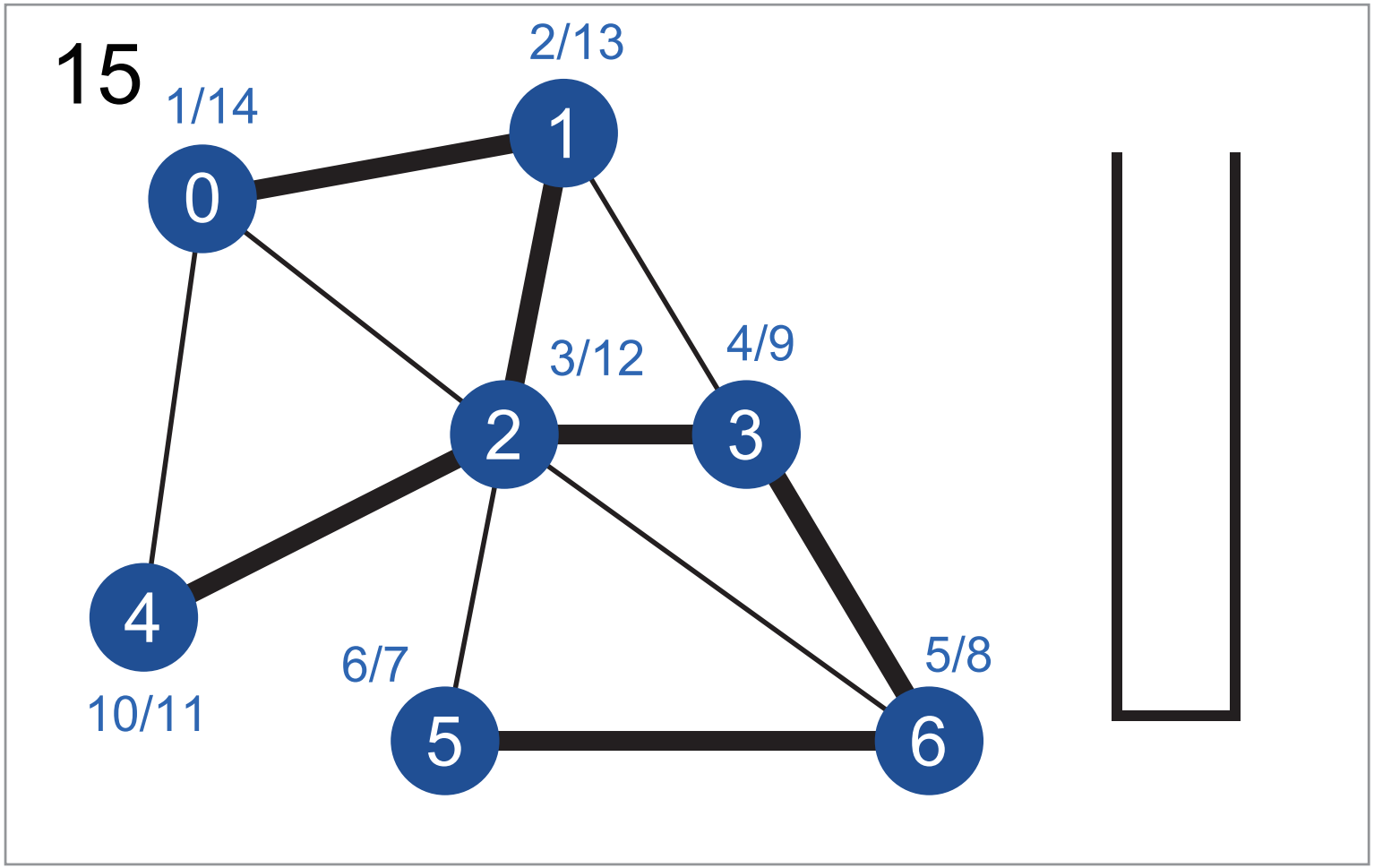
DFS Step 13



DFS Step 14



DFS Step 15



Depth First Search: Implementation with a Stack (1)

```
dfs()
    for each vertex u in V
        color[u] =  WHITE
    time = 0
    for each vertex u in V
        if color[u] == WHITE
            visit(u)
```

Depth First Search: Implementation with a Stack (2)

```
visit(r)
    S.push(r) // Stack S
    color[r] = GRAY
    d[r] = ++time
    while S is not empty
        u = S.top()
        v = next(u) // the next vertex of u
        if v != NULL
            if color[v] == WHITE
                S.push(v)
                color[v] = GRAY
                d[v] = ++time
        else
            S.pop()
            color[u] = BLACK
            f[u] = ++time
```

Depth First Search: Implementation with Recursive Functions (1)

```
dfs(G)
    for each vertex u in V
        color[u] =  WHITE
    time = 0
    for each vertex u in V
        if color[u] == WHITE
            visit(u)
```

Depth First Search: Implementation with Recursive Functions (2)

```
visit(u)
    color[u] = GRAY // White vertex u has just been discovered
    d[u] = ++time
    for each v in Adj[u] // Explore edge (u,v)
        if color[v] == WHITE
            visit(v)
    color[u] = BLACK // Blacken u; it is finished
    f[u] = ++time
```

Complexity of DFS

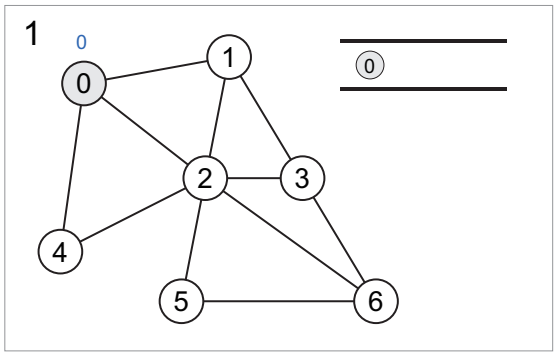
The procedure `visit` is called exactly once for each vertex $v \in V$, since `visit` is invoked only on white vertices and the first thing it does is paint the vertex gray.

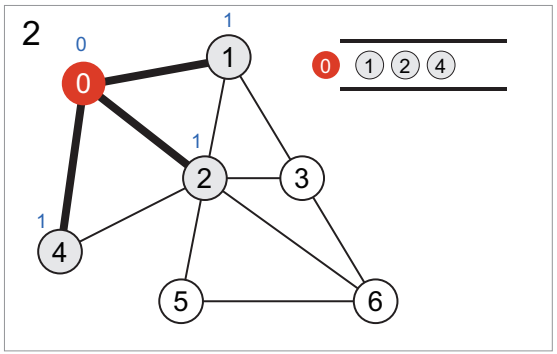
- Implementation based on the adjacency matrix:
  During an execution of `visit(u)` we need to parse a row of the matrix in $O(|V|)$. The running time of DFS is terefore $O(|V|^2)$.
- Implementation based on the adjacency lists:
  During an execution of `visit(u)`, we can explore an edge `adj[u]` times. So, the total cost of executing the exploration is $O(|E|)$. The running time of DFS is therefore $O(|V| + |E|)$.

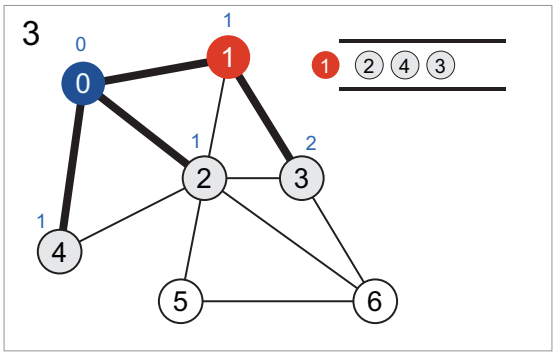
Breadth First Search (1)

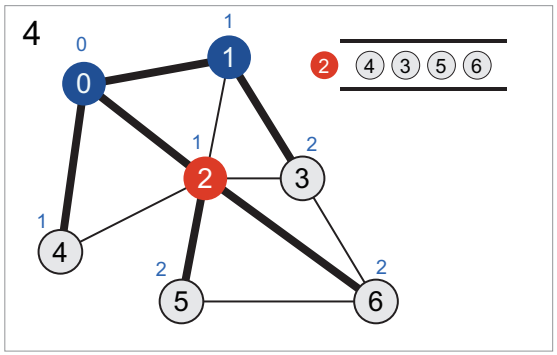
- Given a graph $G = (V, E)$ and a distinguished source vertex s, breadth-first search (BFS) systematically explores the edges of G to "discover" every vertex that is reachable from s.
- It computes the distance (smallest number of edges) from s to each reachable vertex.
- BFS discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.
- The distance from the source s to vertex u computed by BFS is stored in $d[u]$.
- The BFS uses a first-in, first-out queue Q to manage the set of vertices.

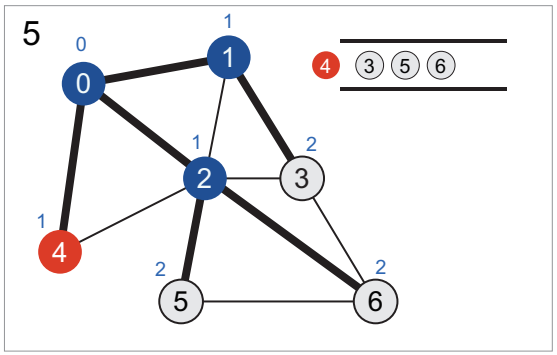
Breadth First Search (2)

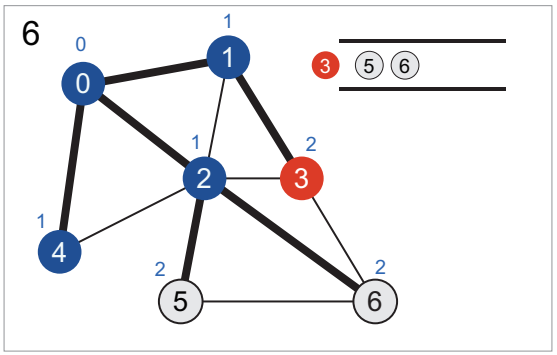




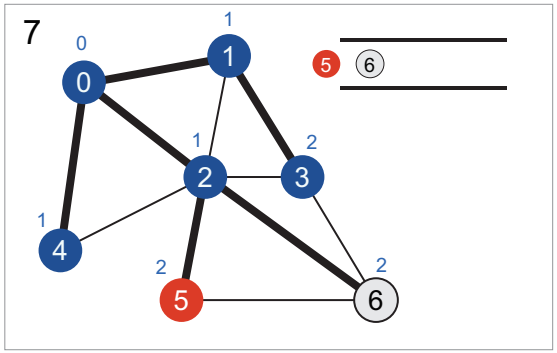


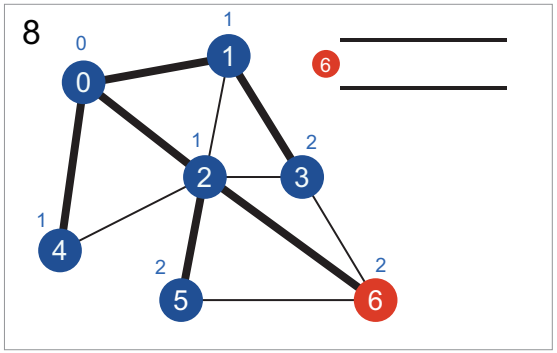


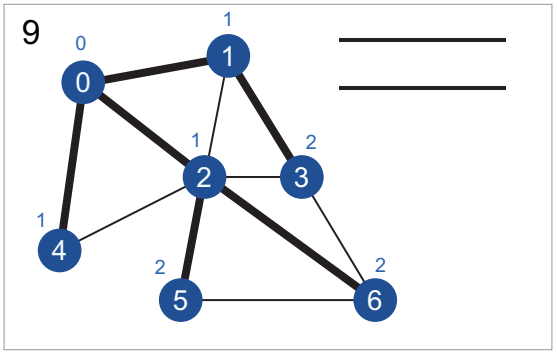




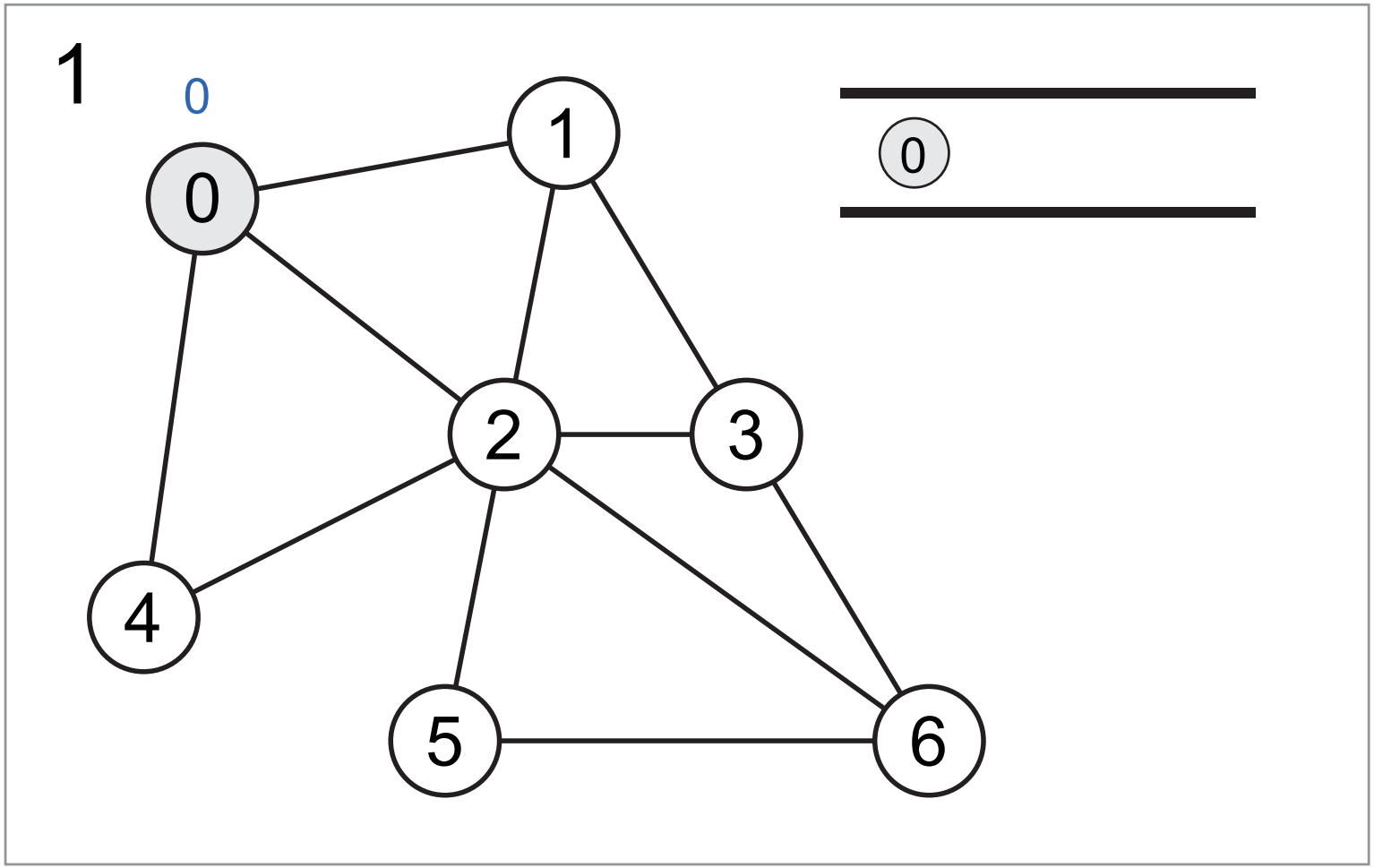
Breadth First Search (3)



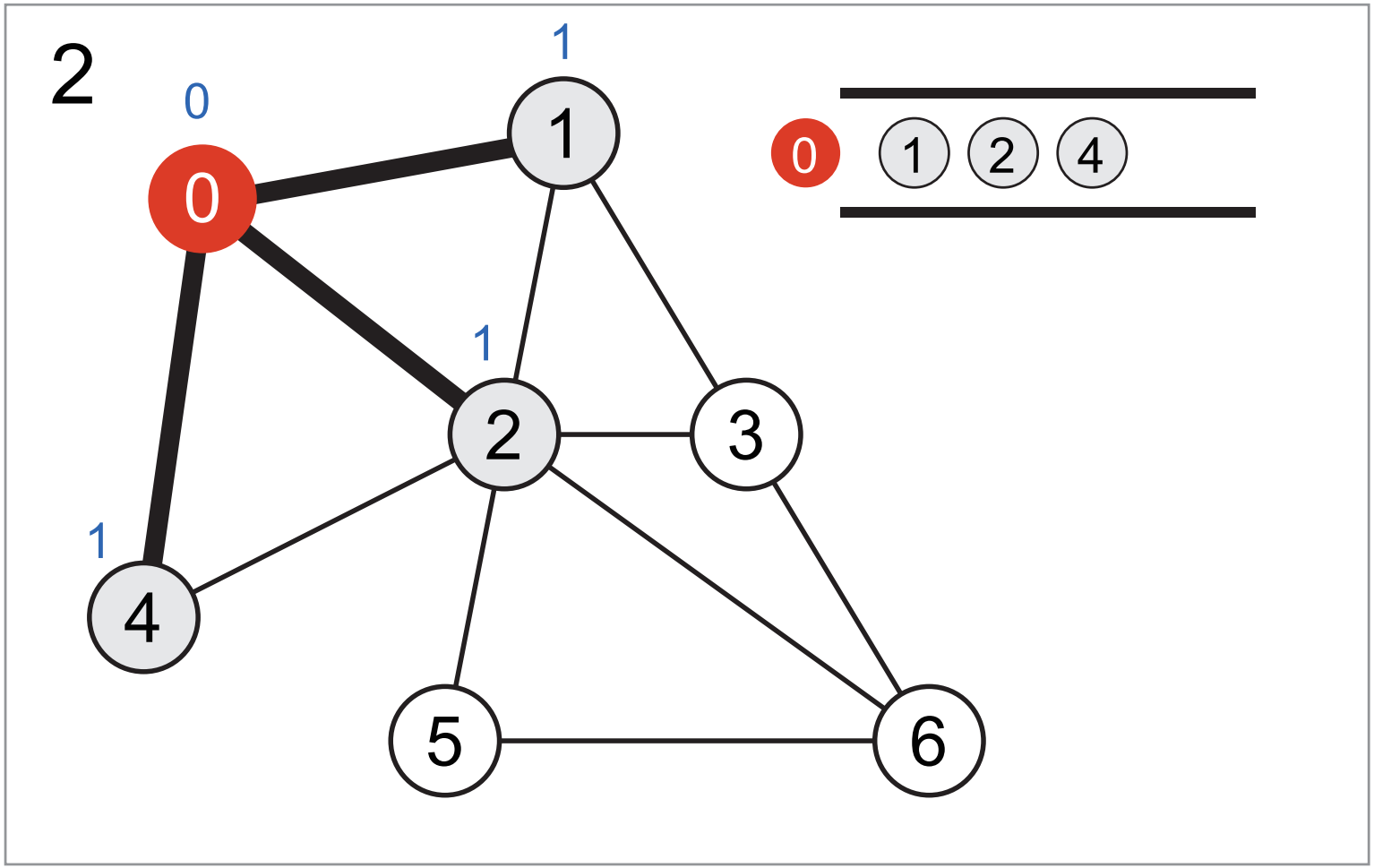




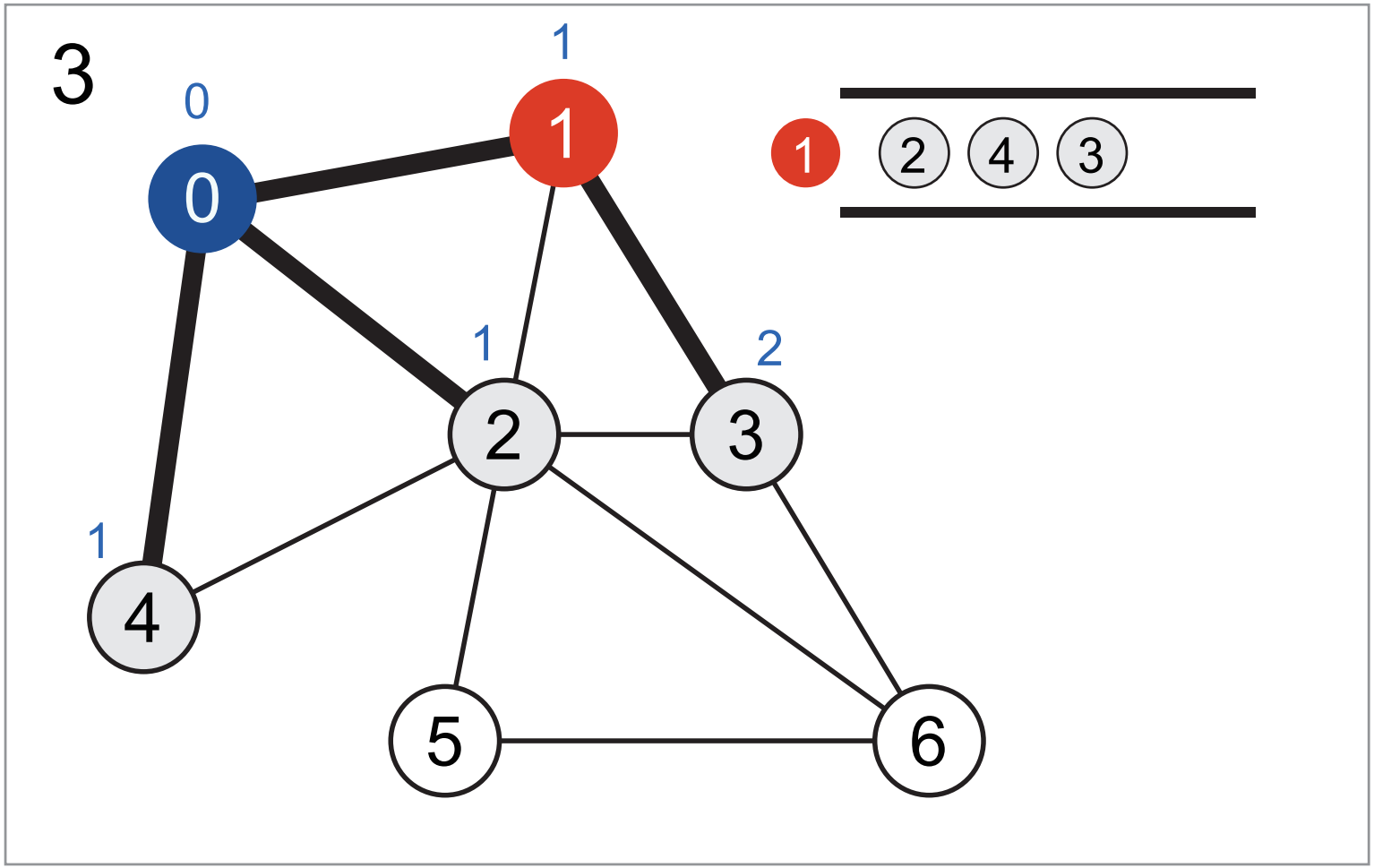
BFS Step 1



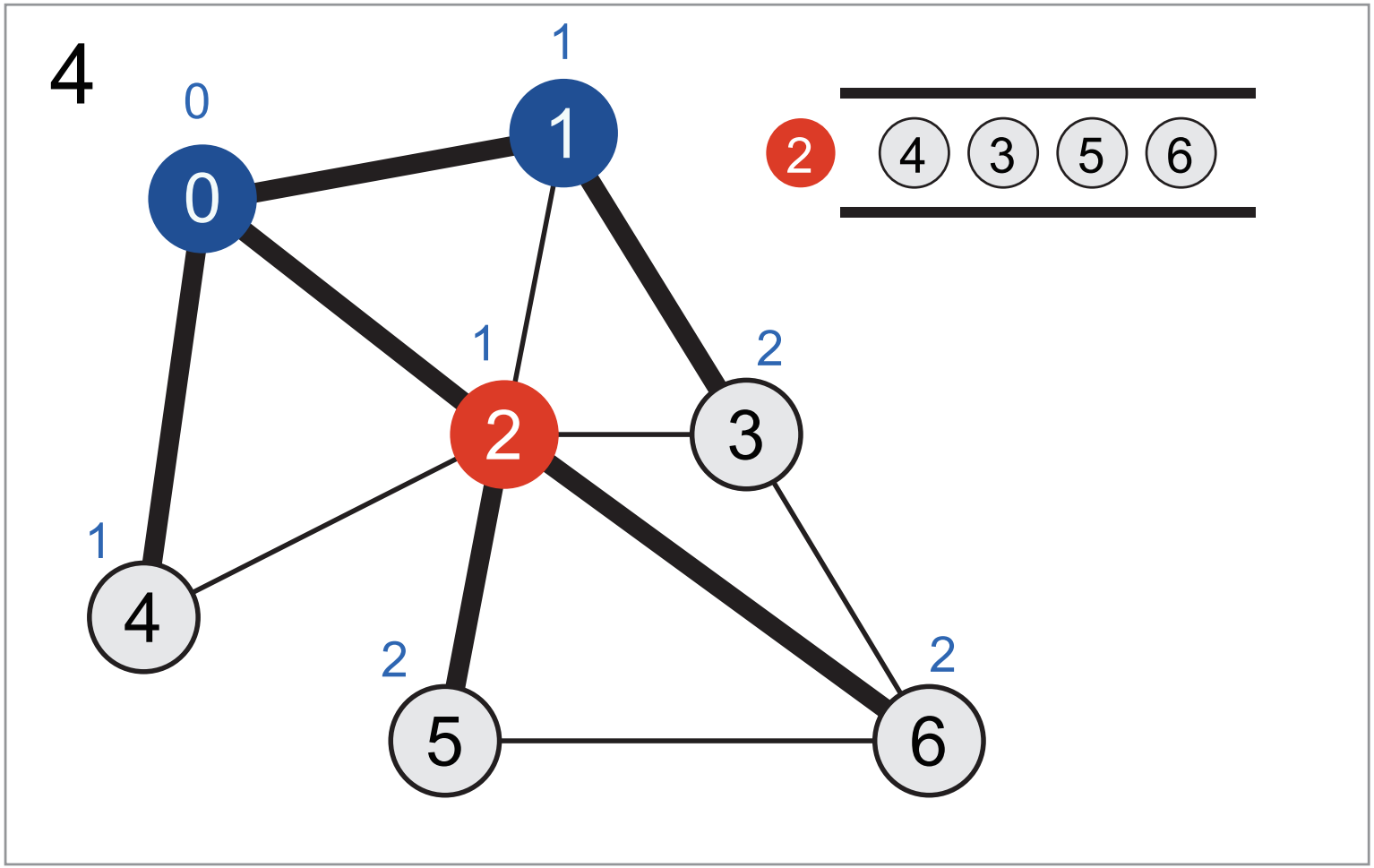
BFS Step 2



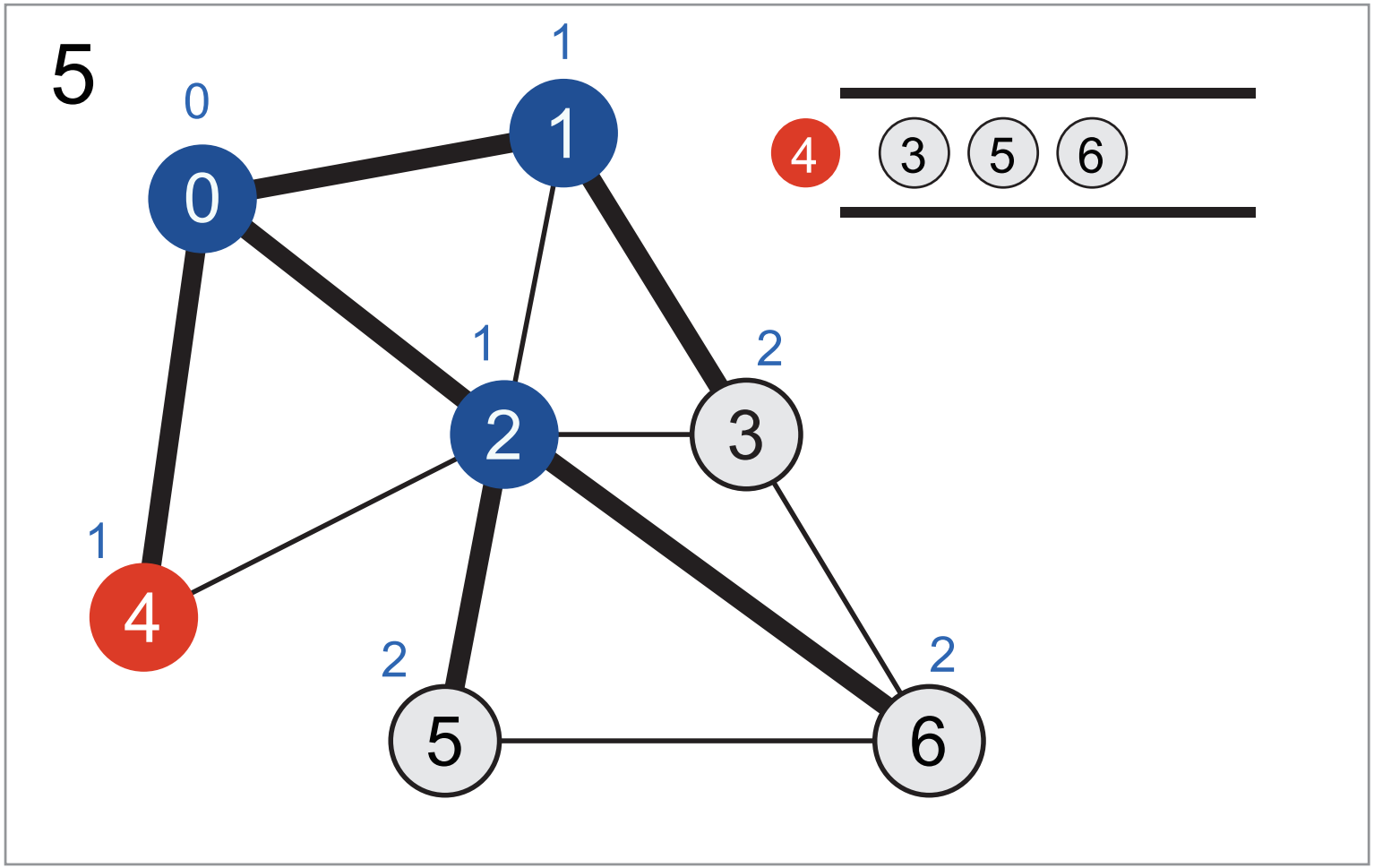
BFS Step 3



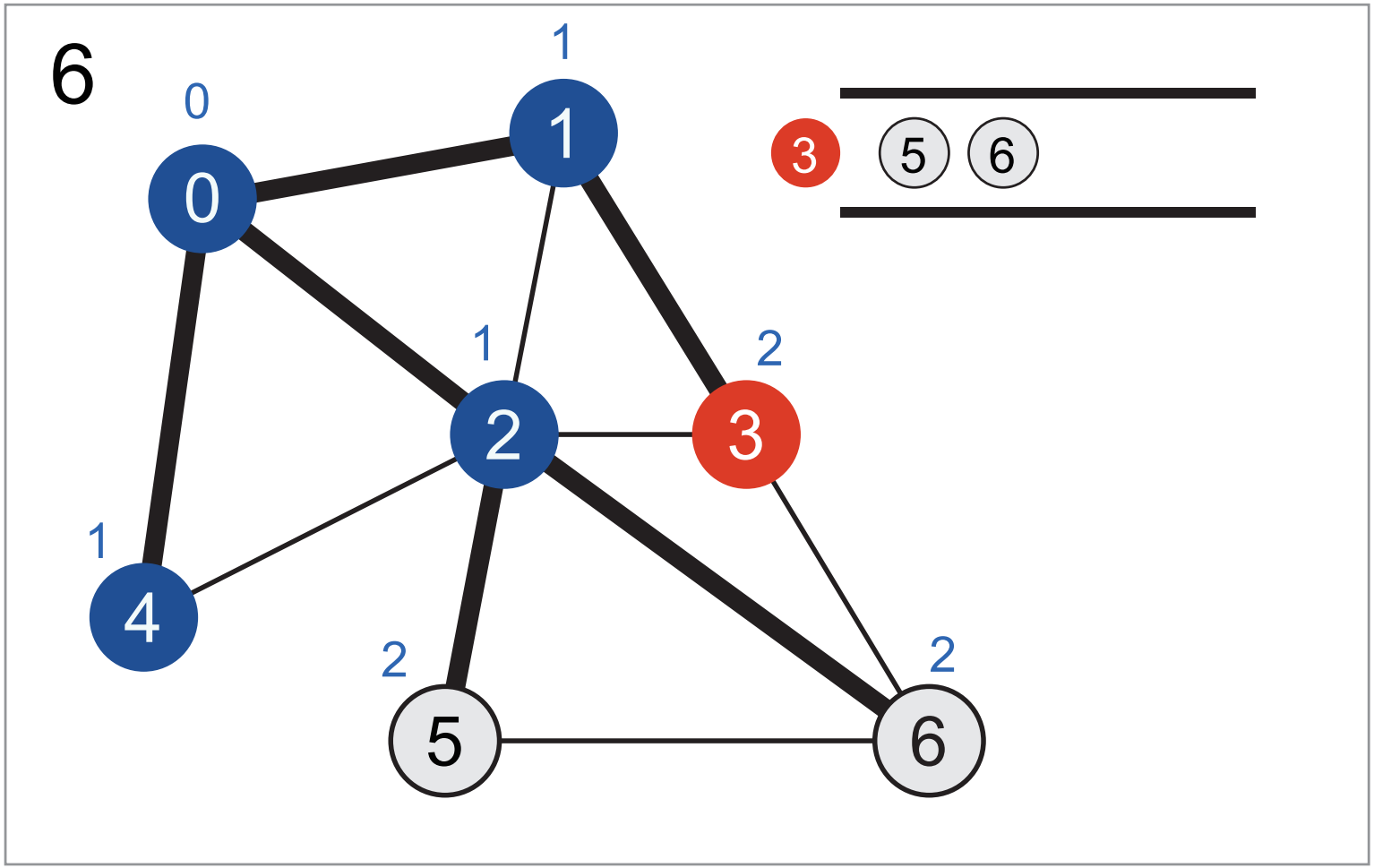
BFS Step 4



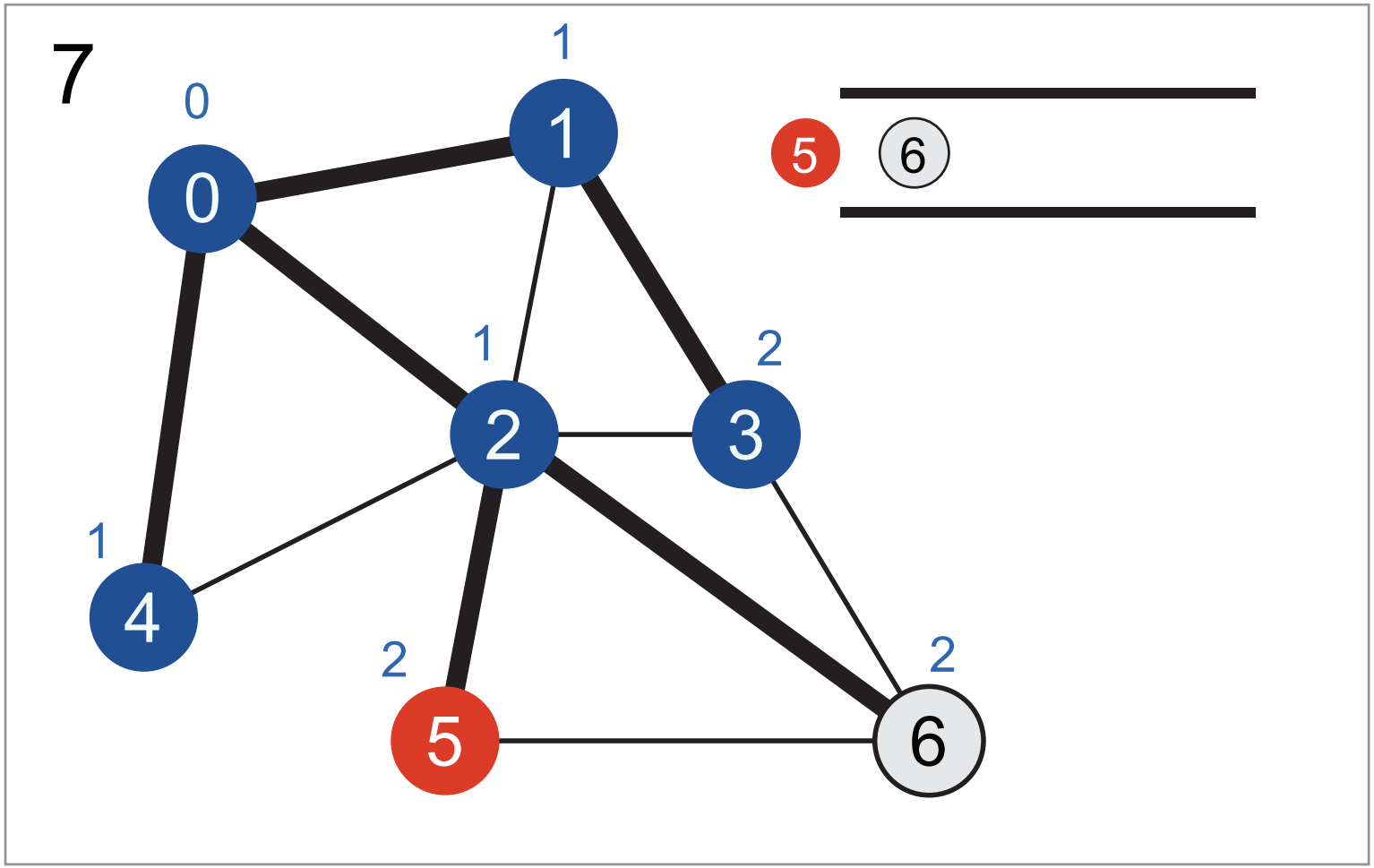
BFS Step 5



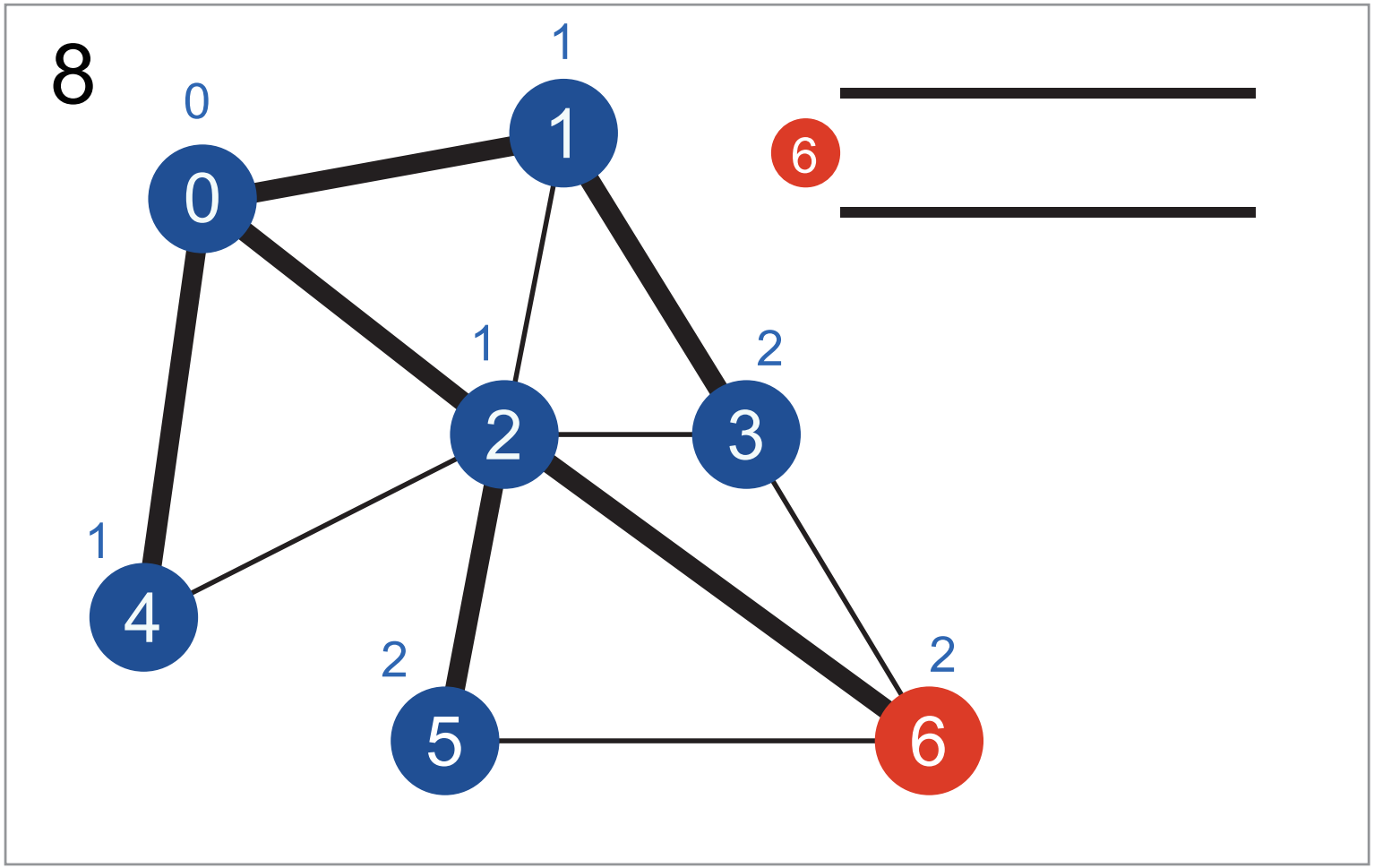
BFS Step 6



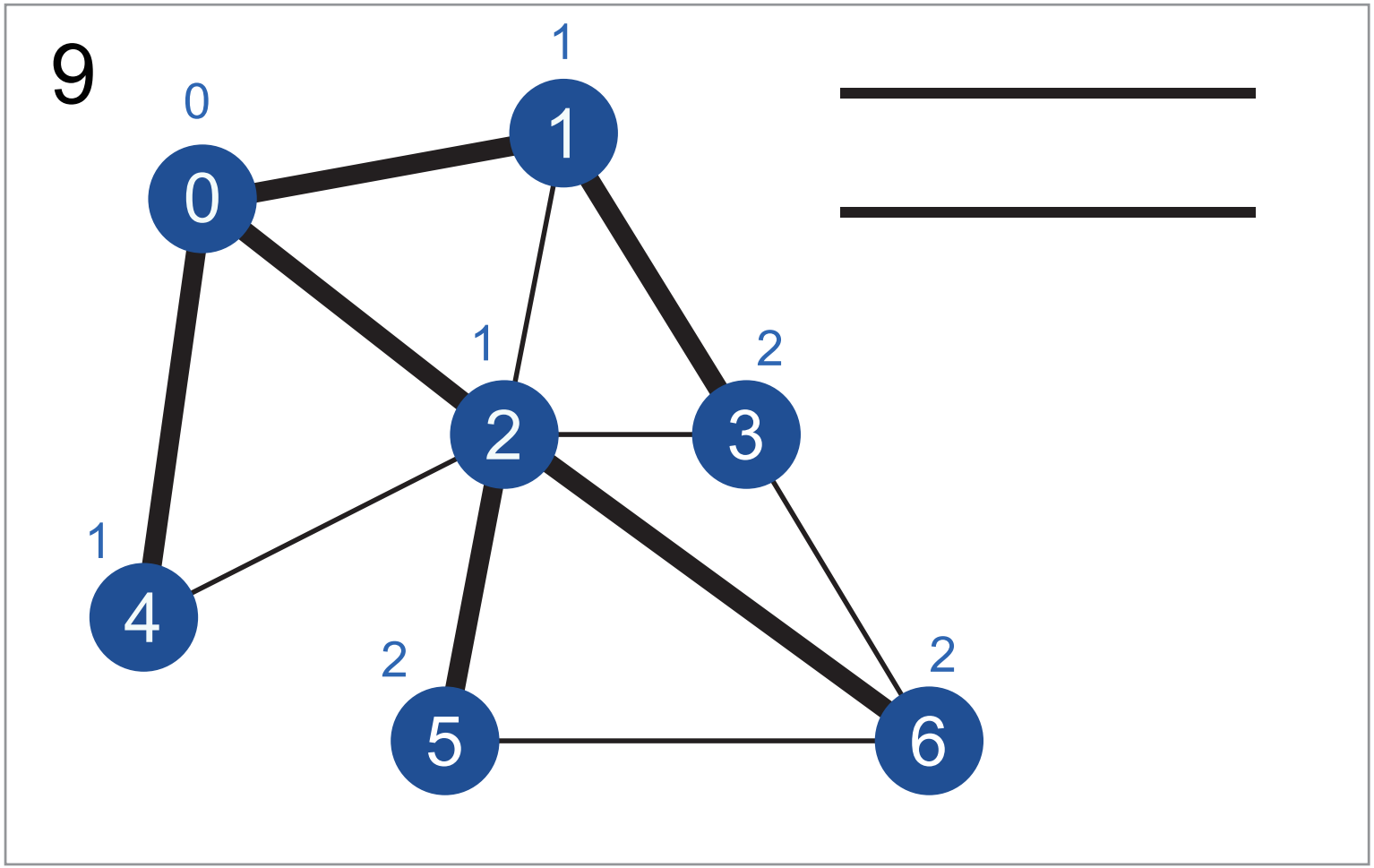
BFS Step 7



BFS Step 8



BFS Step 9



Breadth First Search: Implementation (1)

```
bfs(G,s)
    for each vertex u in V - {s}
        color[u] = WHITE
        d[u] = INF
    color[s] = GRAY
    d[s] = 0
    Q = empty
    Q.enqueue(s)
```

Breadth First Search: Implementation (2)

```
while Q != empty
    u = Q.dequeue()
    for each v in Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            Q.enqueue(v)
    color[u] = BLACK
```

Complexity of BFS

After initialization, no vertex is ever whitened, and each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, so the total time devoted to queue operations is $O(|V|)$.

- Implementation based on the adjacency matrix:
  A row of the matrix is scanned when a vertex is dequeued, so all elements of the matrix are scanned. Thus the running time of BFS is $O(|V|^2)$.
- Implementation based on the adjacency lists:
  The adjacency list of each vertex is scanned only when the vertex is dequeued. So, the total time spent in scanning adjacency lists is $O(|E|)$. The overhead for initialization is $O(|V|)$, and thus total running time of BFS is $O(|V| + |E|)$.

Reference

1 Introduction to Algorithms (third edition), Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press, 2012.