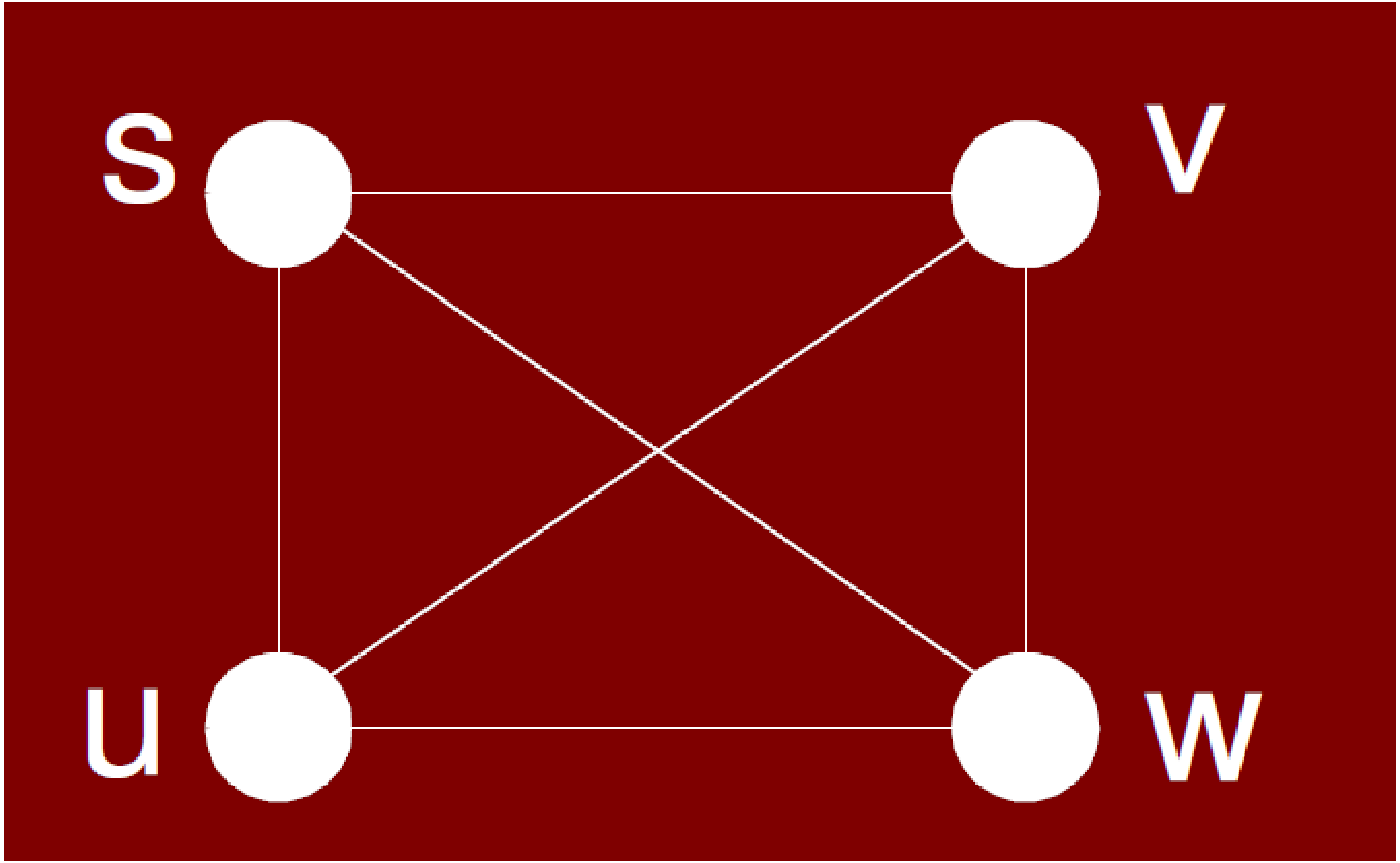
Graphs
Breadth-First Search, Generic Traversal Algorithms

(Class 29)

From Book's Page Number 554 (Chapter 20)

Breadth-First Search, Generic Traversal Algorithms

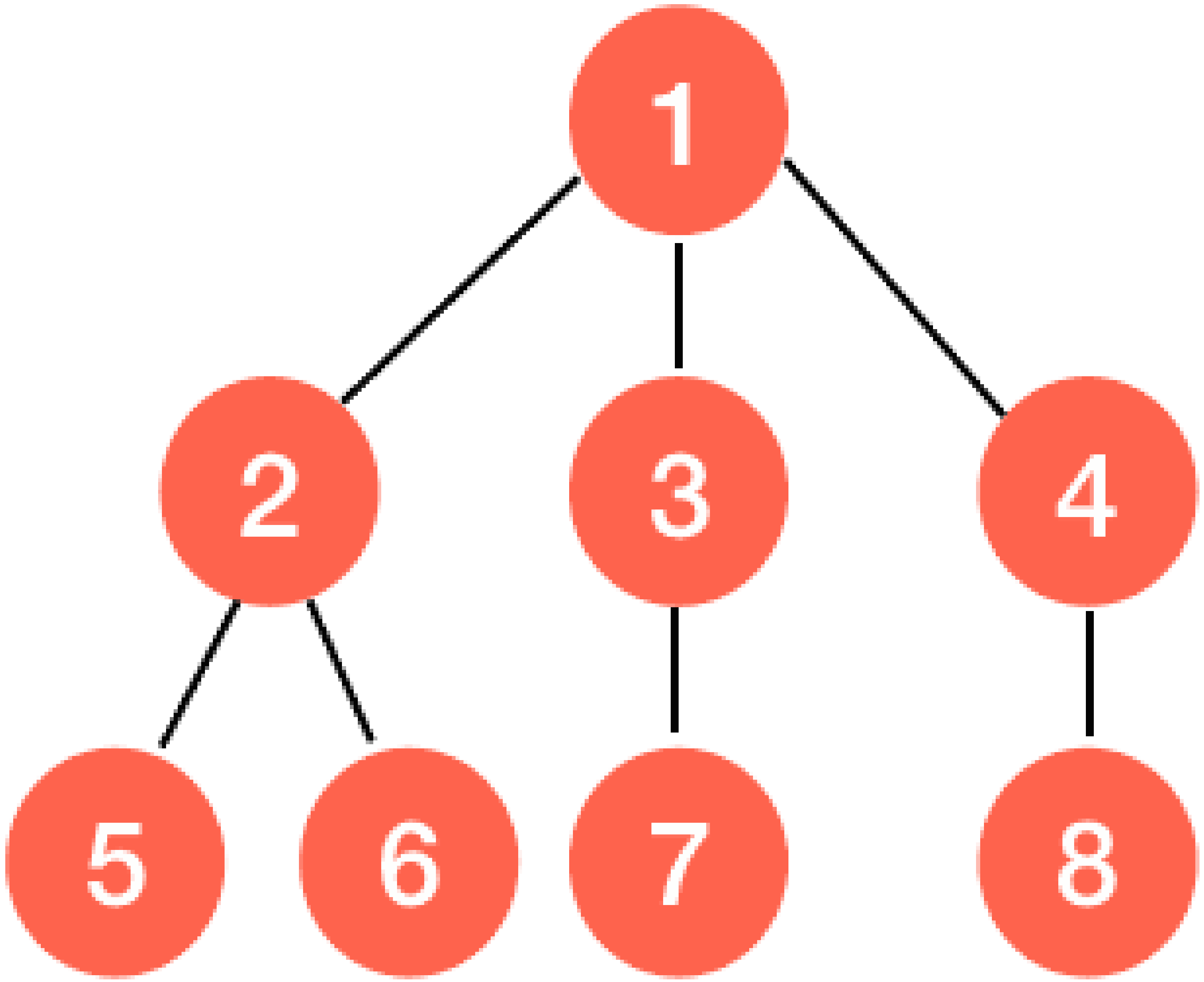
- There is a simple brute-force strategy for computing shortest paths.
- We could simply start enumerating all simple paths starting at s and keep track of the shortest path arriving at each vertex.
- However, there can be as many as $n!$ simple paths in a graph.
- To see this, consider a fully connected graph.


s
v
u
w

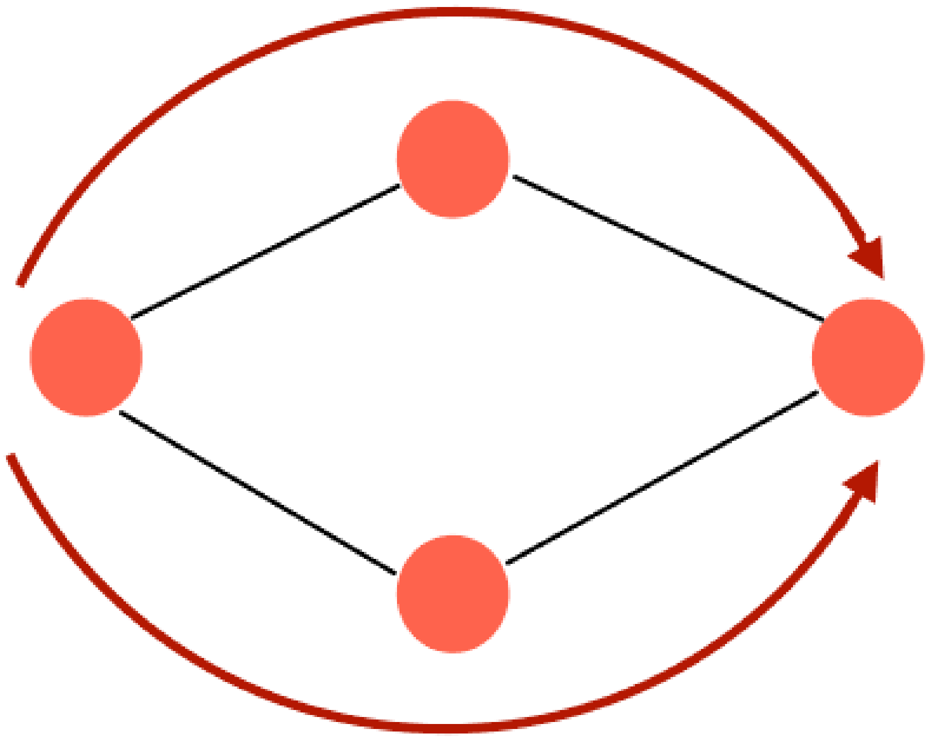
- Then, n choices for source node s, $(n-1)$ choices for destination node, $(n-2)$ for first hop (edge) in the path, $(n-3)$ for second, $(n-4)$ for third down to $(n-(n-1))$ for last leg.
- This leads to $n!$ simple paths.
- Clearly this is not feasible.

Breadth-First Search

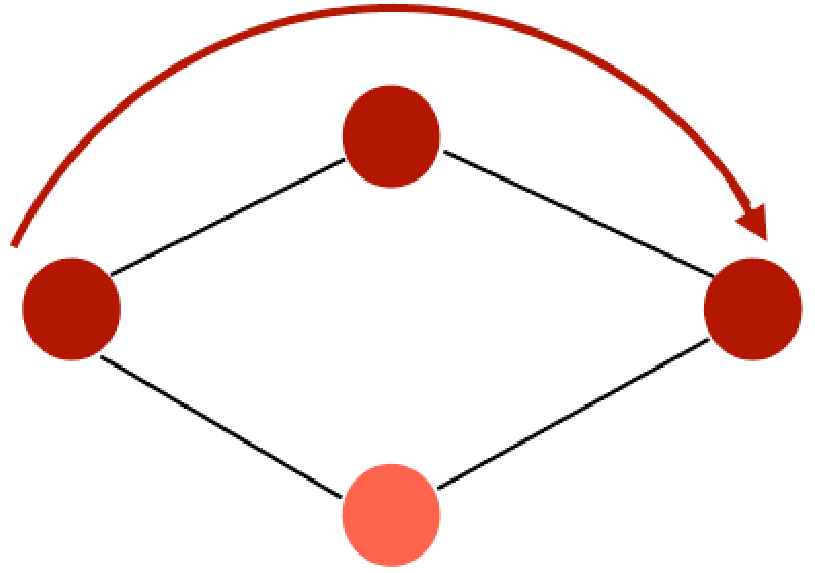
- Here is a more efficient algorithm called the *breadth-first search* (BFS).
- Breadth-first search or BFS is a searching technique for graphs in which we first visit all the nodes at the same depth first and then proceed visiting nodes at a deeper depth.


1
2
3
4
5
6
7
8

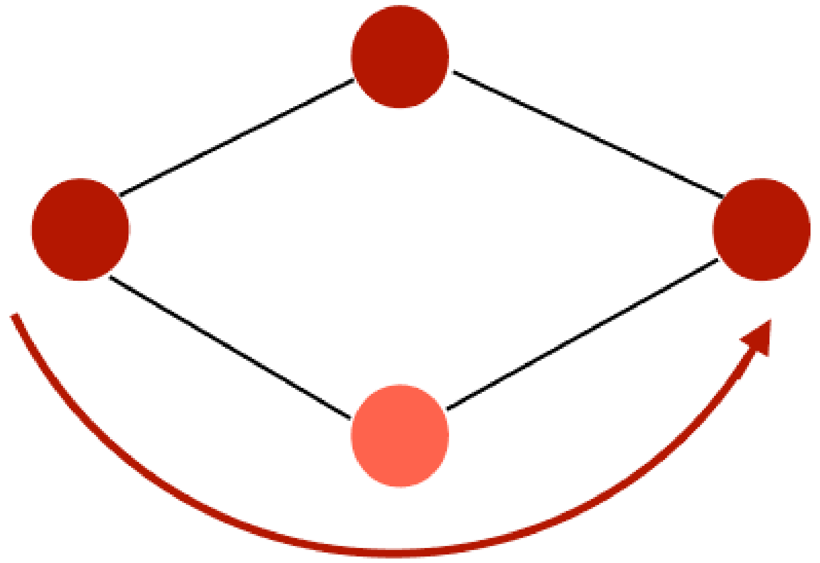
- A node can be reached from different nodes using different paths, but we need to visit each node only once.
- So, we mark each node differently into 3 categories:
    - Unvisited
    - Discovered
    - Complete



Can be reached by either paths

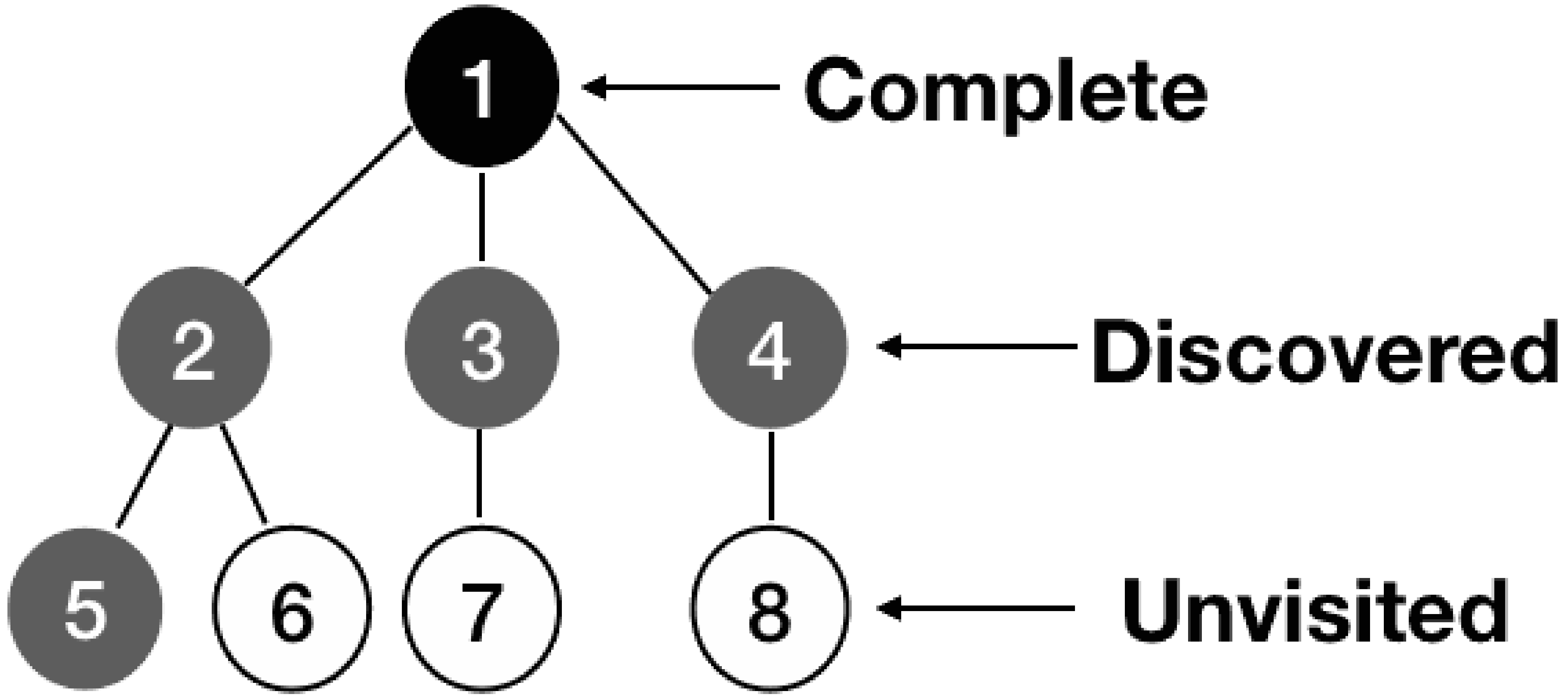


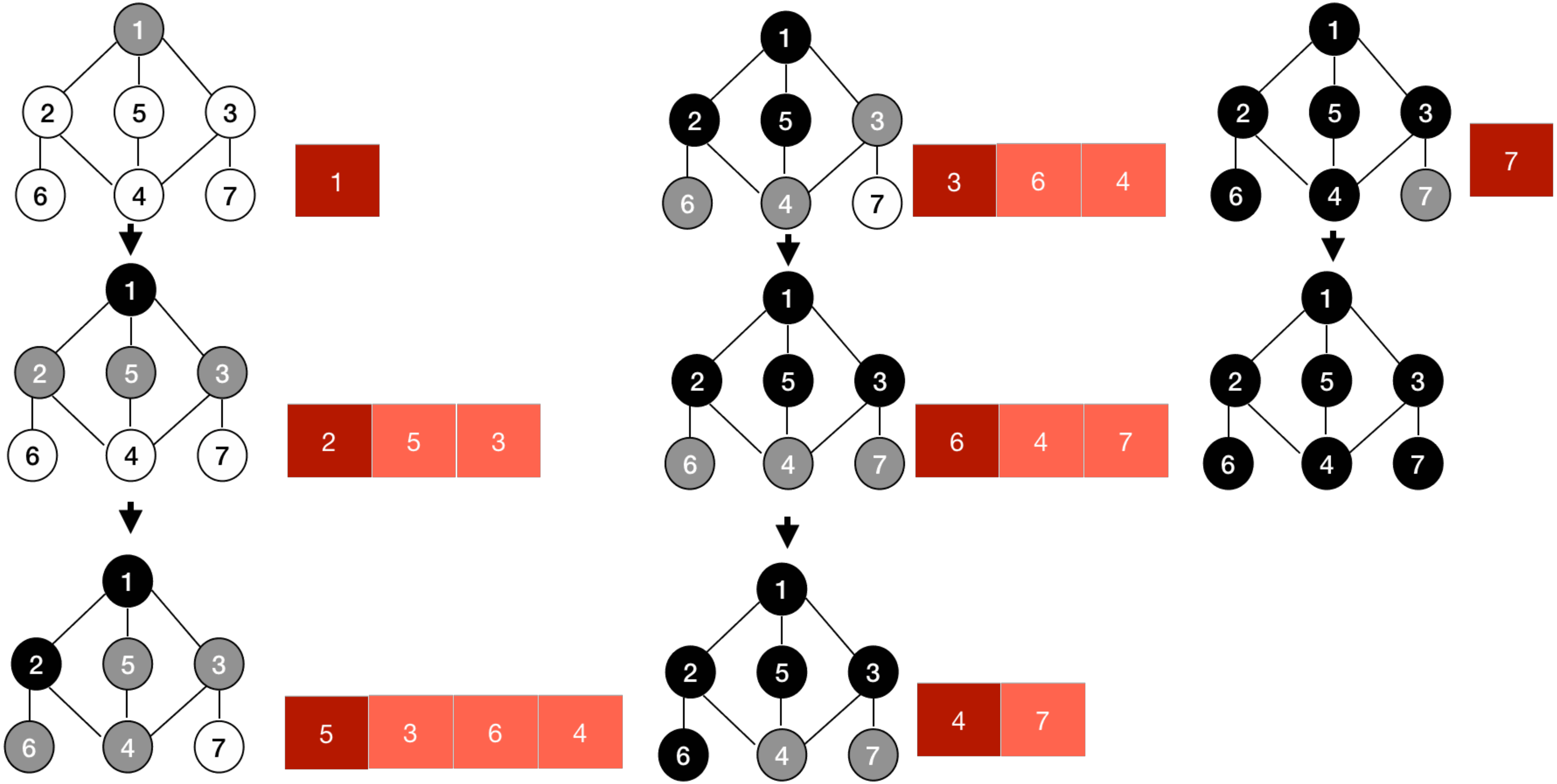
Mark discovered



Don’t visit if already discovered

- Initially, all nodes are unvisited. After visiting a node for the first time, it becomes discovered.
- A node is complete if all of its adjacent nodes have been visited.
- Thus, all the adjacent nodes of a complete node are either discovered or complete.
- Generally, three different colors i.e., white, gray and black are used to represent unvisited, discovered and complete respectively.

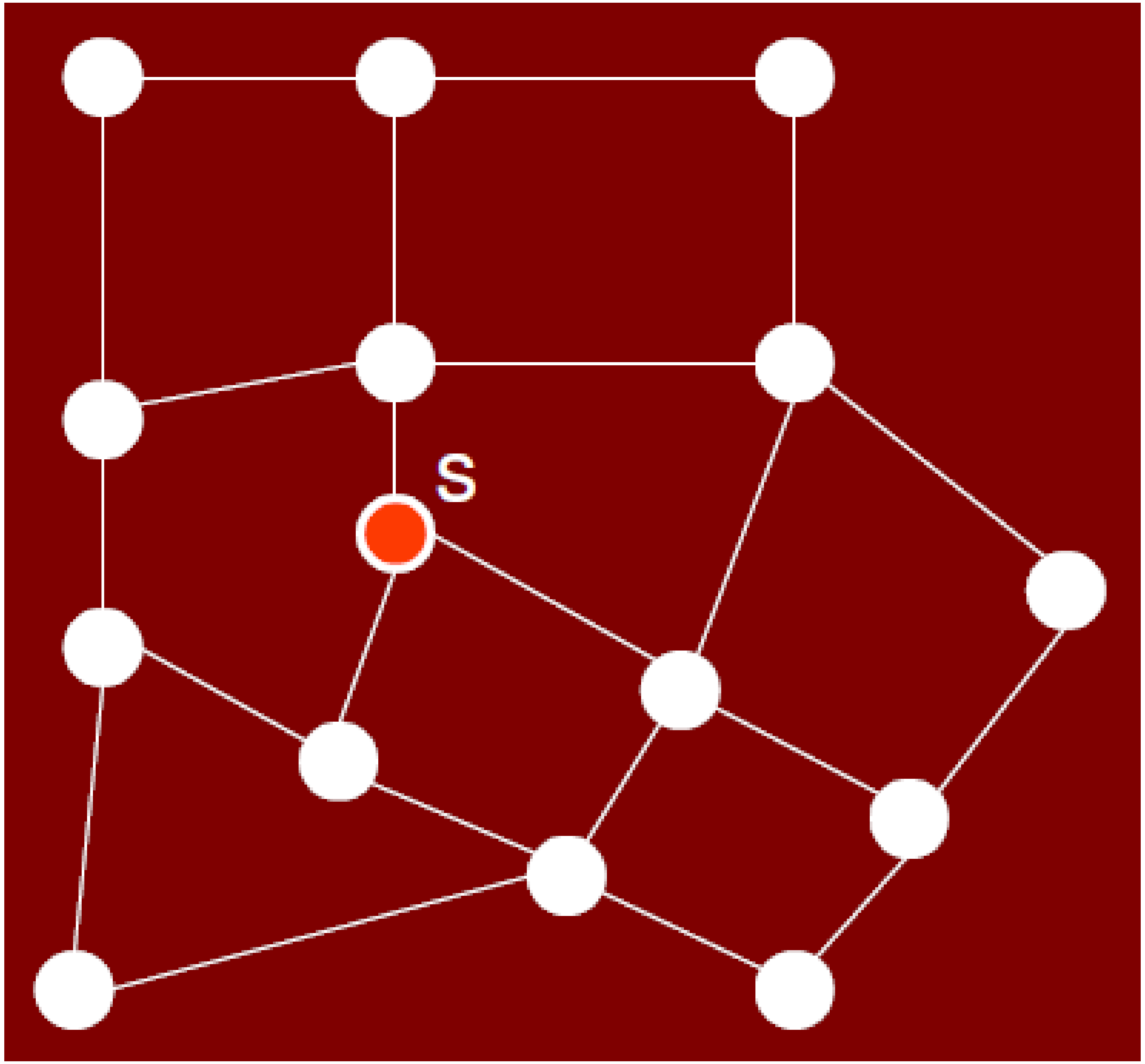

1
2
3
4
5
6
7
8
Complete
Discovered
Unvisited

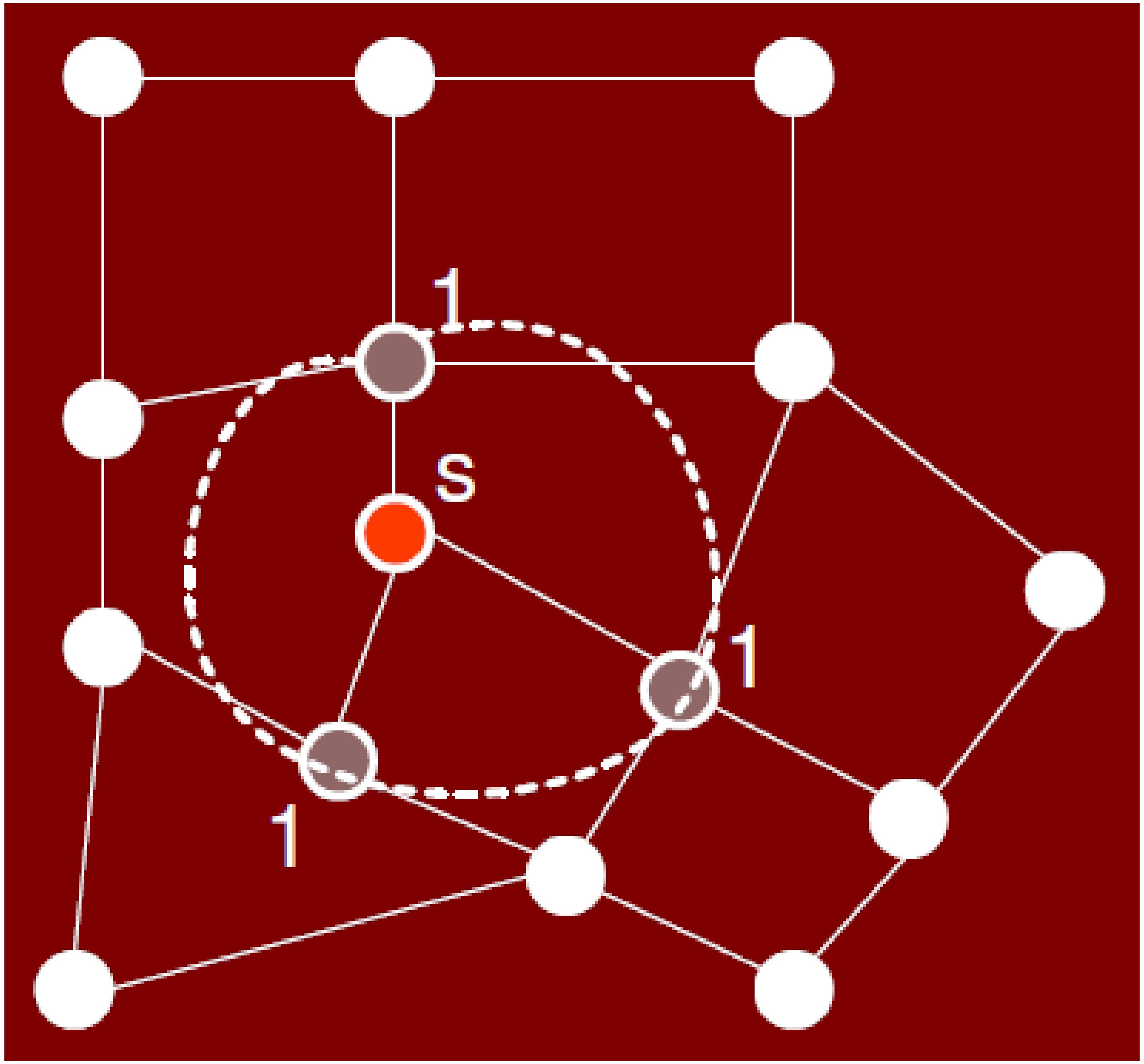

1
2 5 3
5 3 6 4
3 6 4
6 4 7
4 7
7

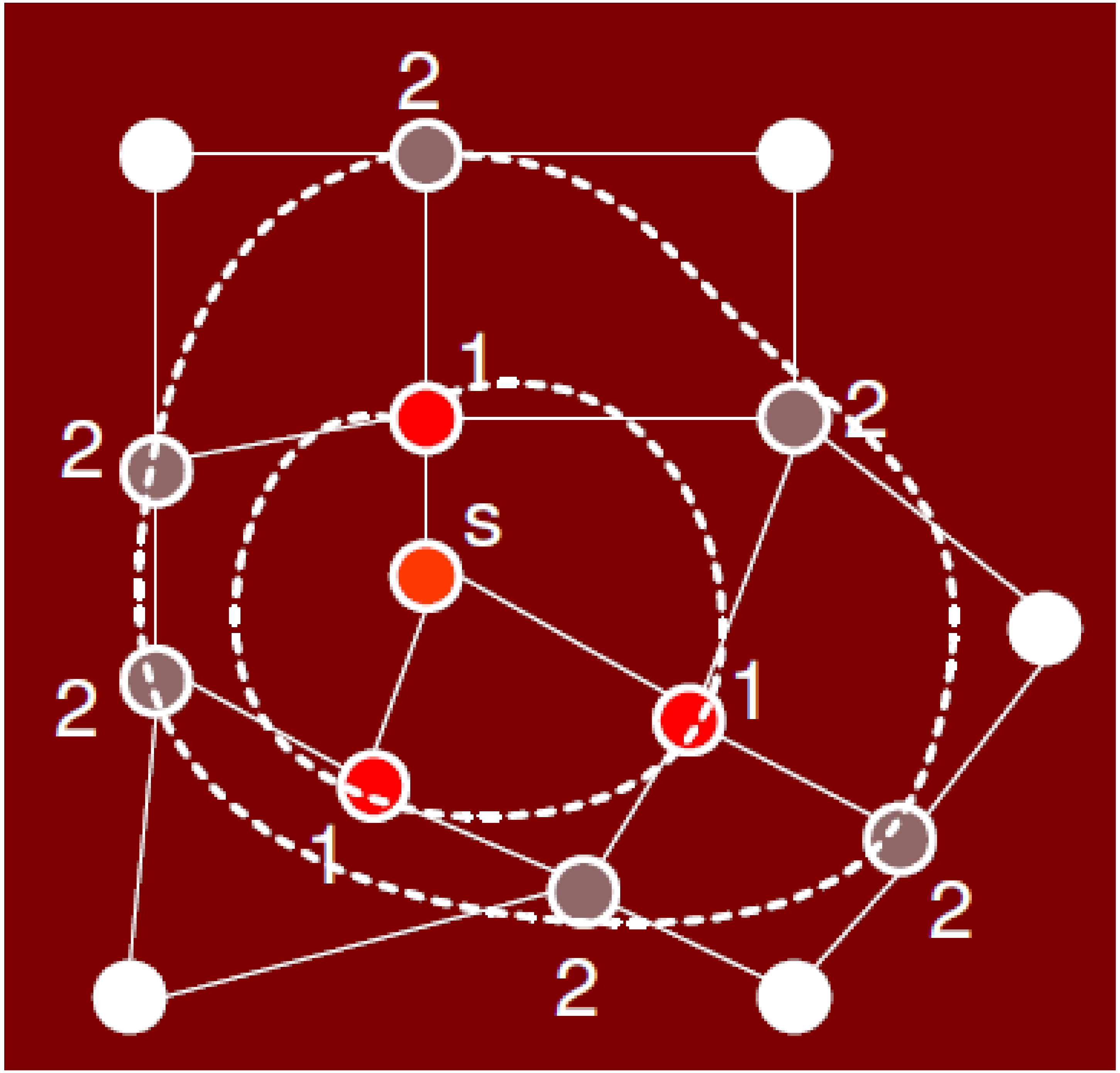
- Start with s and visit its adjacent nodes.
- Label them with distance 1.
- Now consider the neighbors of neighbors of s.
- These would be at distance 2.

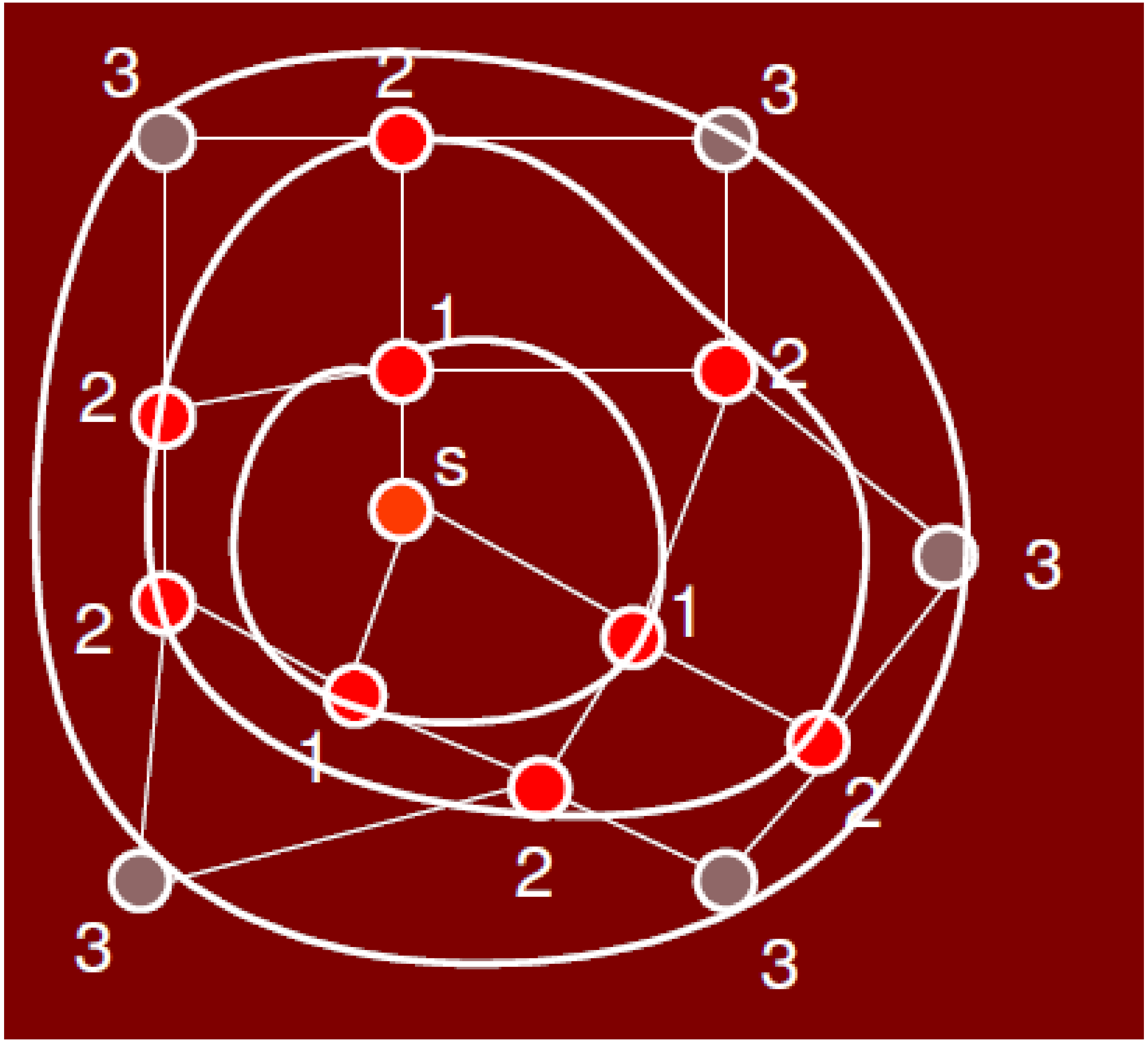
- Now consider the neighbors of neighbors of neighbors of s.
- These would be at distance 3.
- Repeat this until no more unvisited neighbors left to visit.

- The algorithm can also be visualized as a *wave front* propagating outwards from s visiting the vertices in bands at ever increasing distances from s.
- BFS uses queue in its implementation.


s


1
s
1
1


2
1
2
2
s
2
1
1
2
2

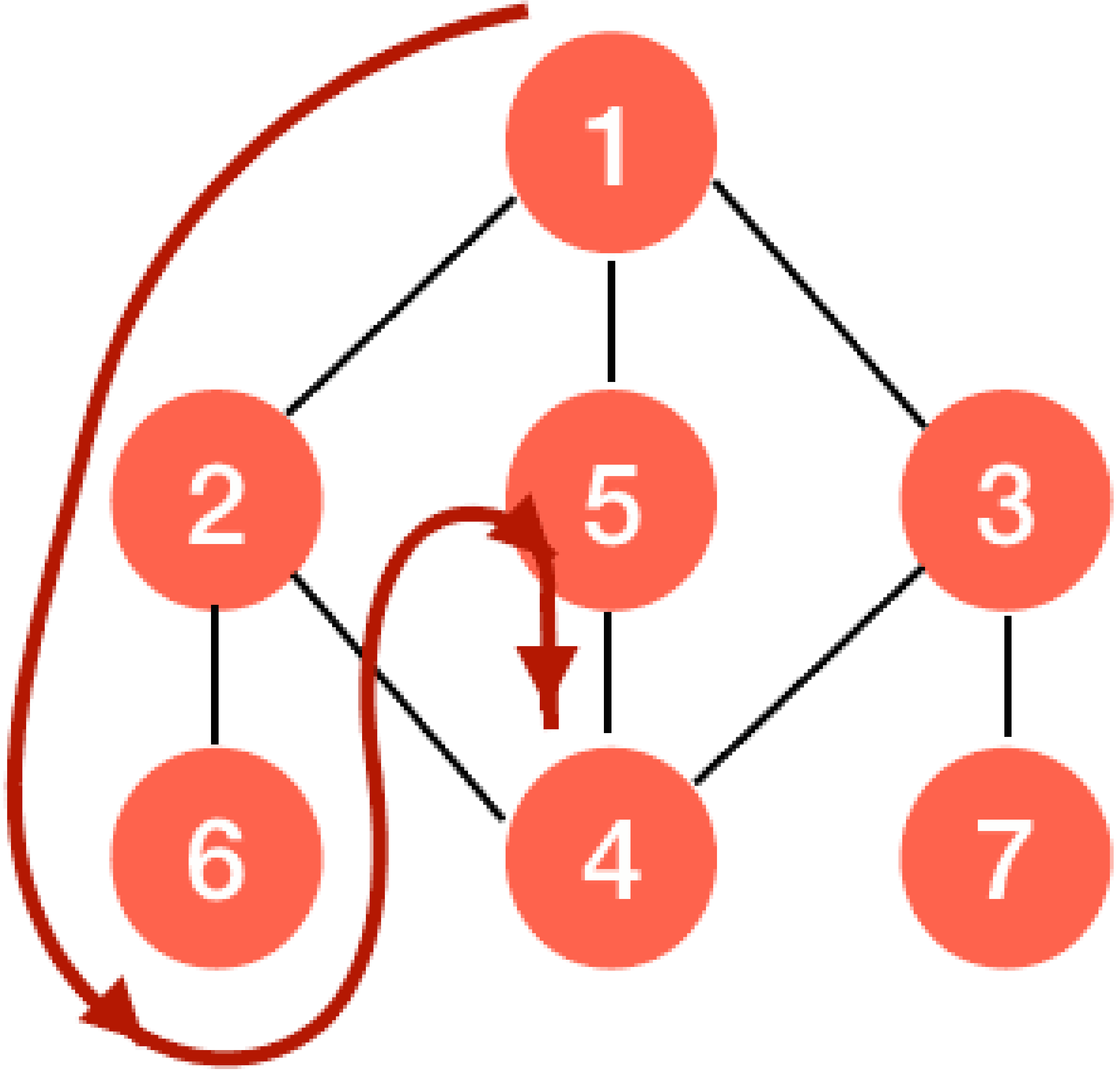


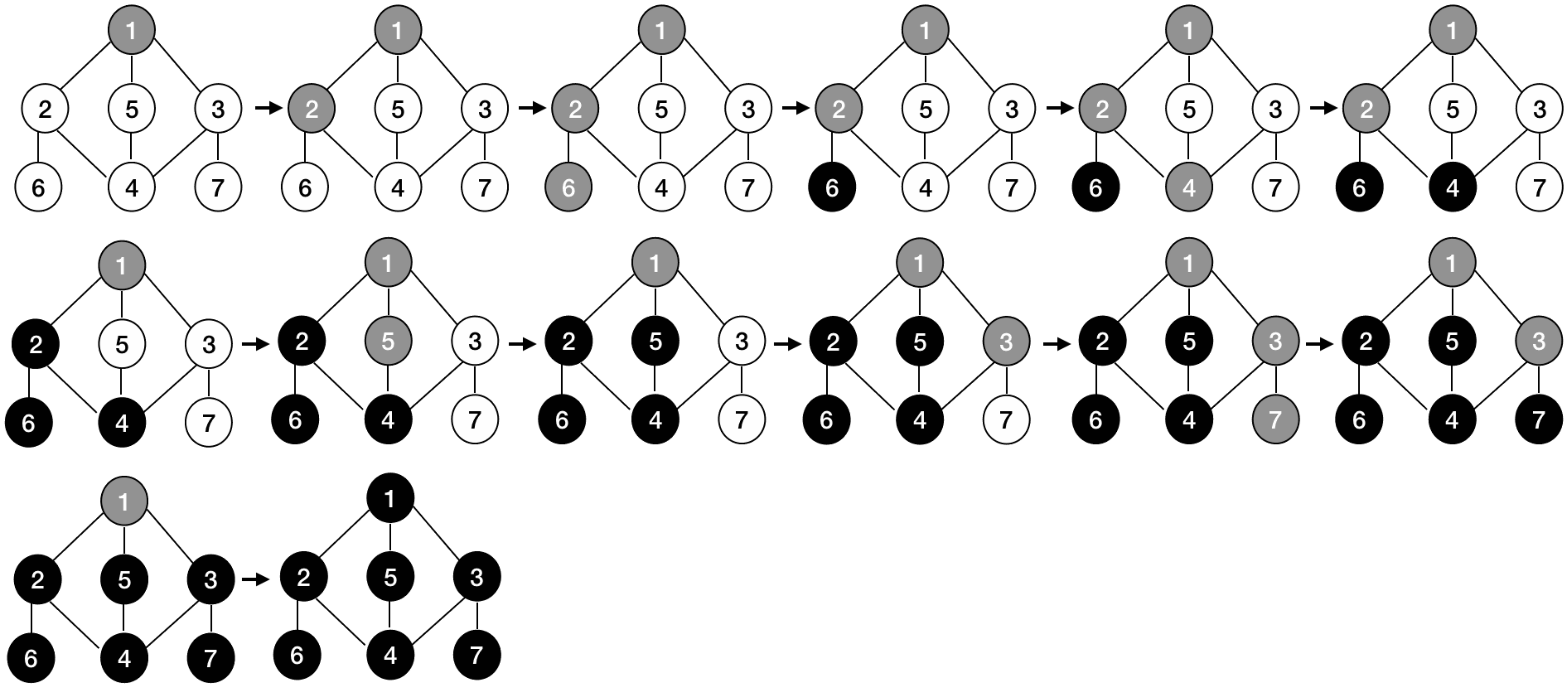
3
2
3
1
2
2
s
3
2
1
1
2
2
3
3

Depth-First Search

- Breadth-first search is one instance of a general family of *graph traversal algorithms*.
- Traversing a graph means visiting every node in the graph.
- Another traversal strategy is *depth-first search* (DFS).

- Depth-first search or DFS is also a searching technique like BFS.
- As its name suggests, it first explores the depth of the graph before the breadth i.e., it traverses along the increasing depth and upon reaching the end, it backtracks to the node from which it was started and then do the same with the sibling node.
- Similar to the BFS, we also mark the vertices white, gray and black to represent unvisited, discovered and complete respectively.


1
2
5
3
6
4
7


1
2
5
3
6
4
7

- As BFS uses a queue, the DFS is using a stack for its implementation. The recursive calls of the DFS-VISIT are stored in a stack.
- DFS procedure can be written recursively or non-recursively.
- Both versions are passed s initially.

Depth-First: Recursive Algorithm

```
RECURSIVE_DFS(v)
if (v is unmarked)
    mark v
    for each edge (v,w)   // all the adjacent vertices of v
        RECURSIVE_DFS (w)
```

Generic Graph Traversal Algorithm

- The generic graph traversal algorithm stores a set of candidate edges in some data structures we'll call a "bag".
- The only important properties of the "bag" are that we can put stuff into it and then later take stuff back out.
- Considering p as parent vertex and v in the adjacent vertex.
- Here is the generic traversal algorithm.

```
TRAVERSE(s)
1 put (∅, s) in bag
2 while bag not empty
3   take (p, v) from bag
4   if (v is unmarked )
5     mark v
6     parent (v) ← p
7     for each edge (v,w)
8       put (v,w) in bag
```

- Notice that we are keeping edges in the bag instead of vertices.
- This is because we want to remember, whenever we visit v for the first time, which previously visited vertex p.
- Put v into the bag.
- The vertex p is call the *parent* of v.

- The running time of the traversal algorithm depends on how the graph is represented and what data structure is used for the bag.
- But we can make a few general observations.
    - Since each vertex is visited at most once, the for loop in line 7 is executed at most V times.
    - Each edge is put into the bag exactly twice; once as (u, v) and once as (v, u), so line 8 is executed at most $2E$ times.
    - Finally, since we can’t take out more things out of the bag than we put in, line 3 is executed at most $2E + 1$ times.
    - Assume that the graph is represented by an adjacency list so the overhead of the for loop in line 7 is constant per edge.

- If we implement the bag by using a *stack*, we have *depth-first search* (DFS) or traversal.
- While if we implement the bag by using *queue*, we have *breadth-first search* (BFS).

Depth-First Search using Generic Traversal Algorithm

```
TRAVERSE(s)
1 push(∅, s)
2 while stack not empty
3   pop(p, v)
4     if (v is unmarked )
5       mark v
6       parent (v) ← p
7       for each edge (v,w)
8         push(v,w)
```