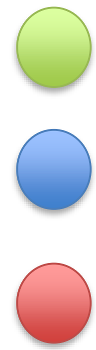
Algorithm and Data Structure Analysis (ADSA)

Depth first search / Strongly Connected Components

Breadth-first-search (BFS)

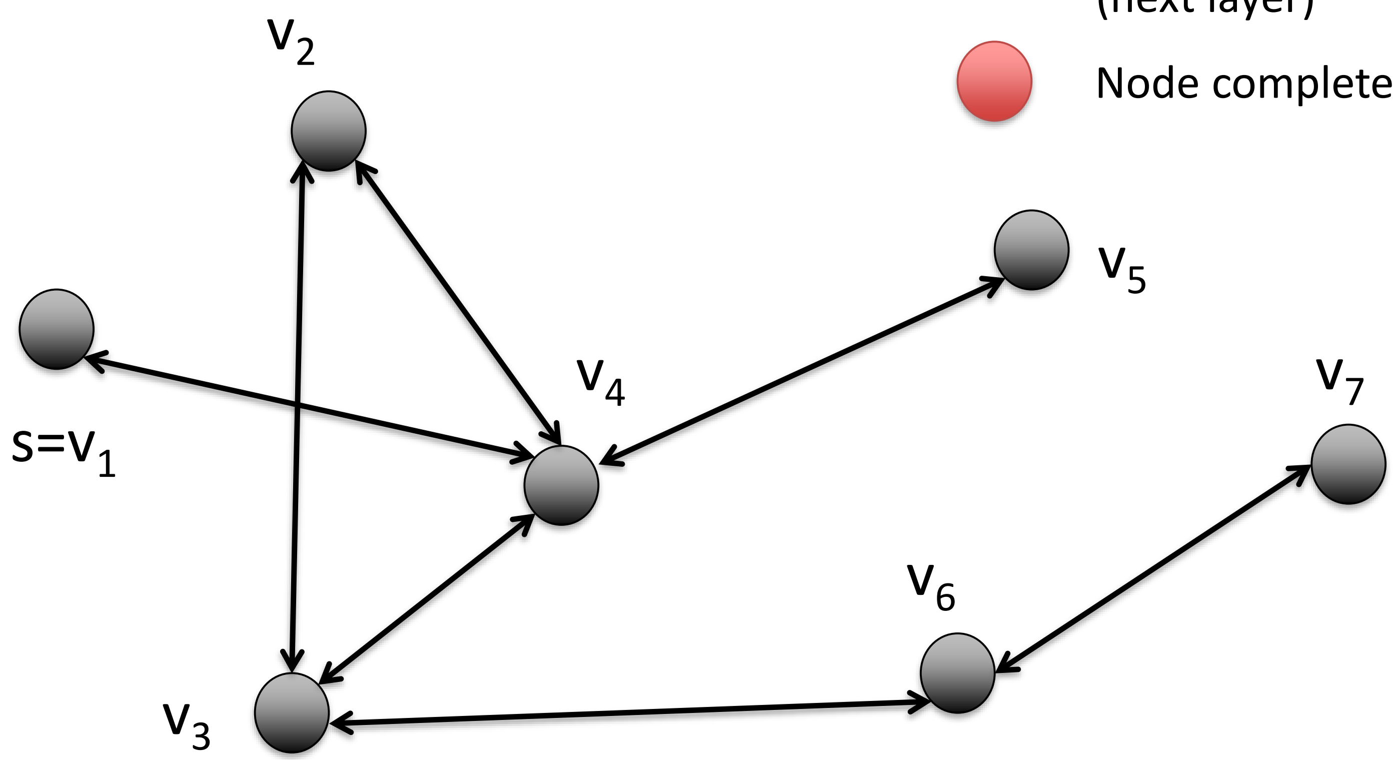
Three types of nodes



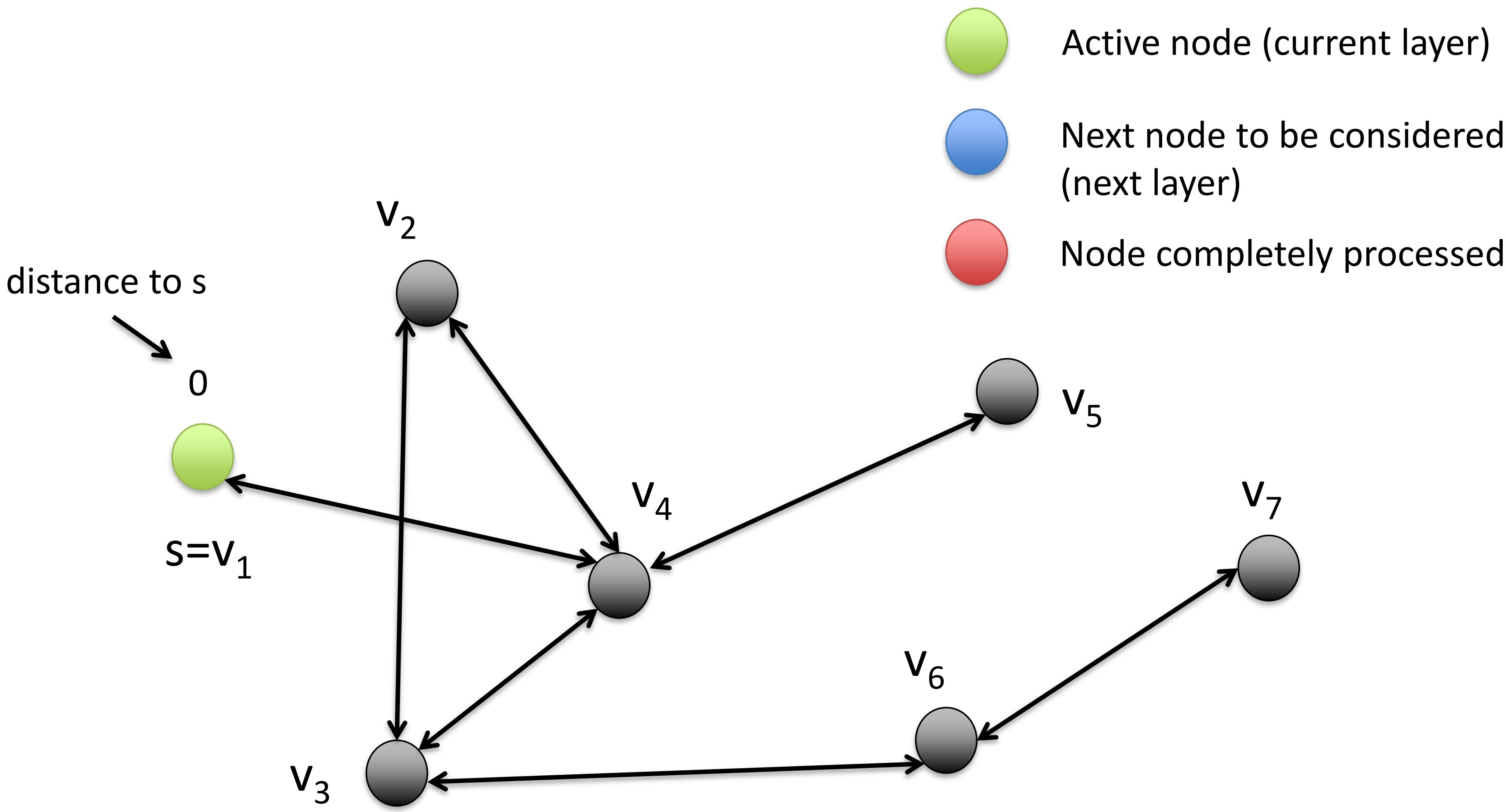
Active node (current layer)

Next node to be considered (next layer)

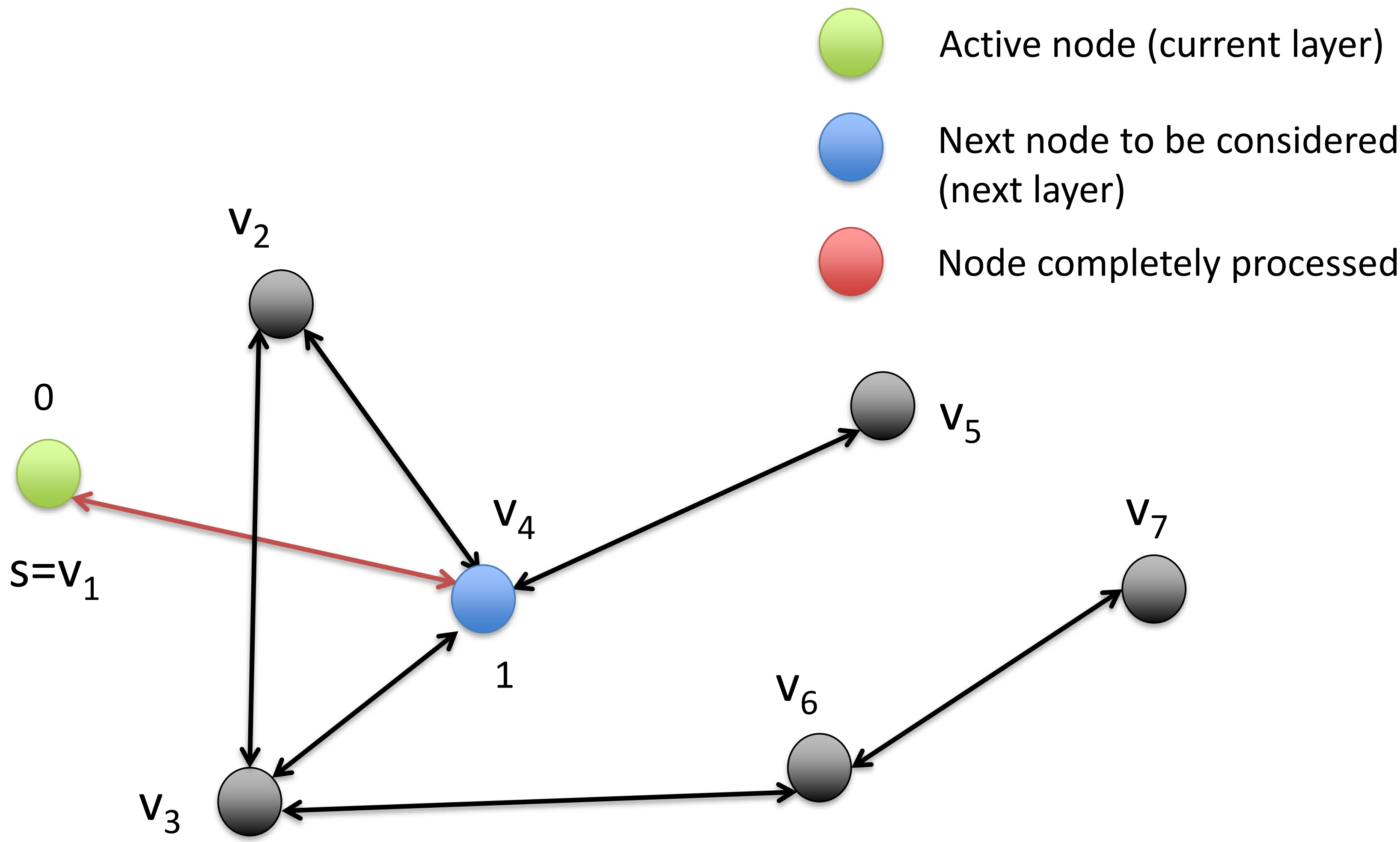
Node completely processed



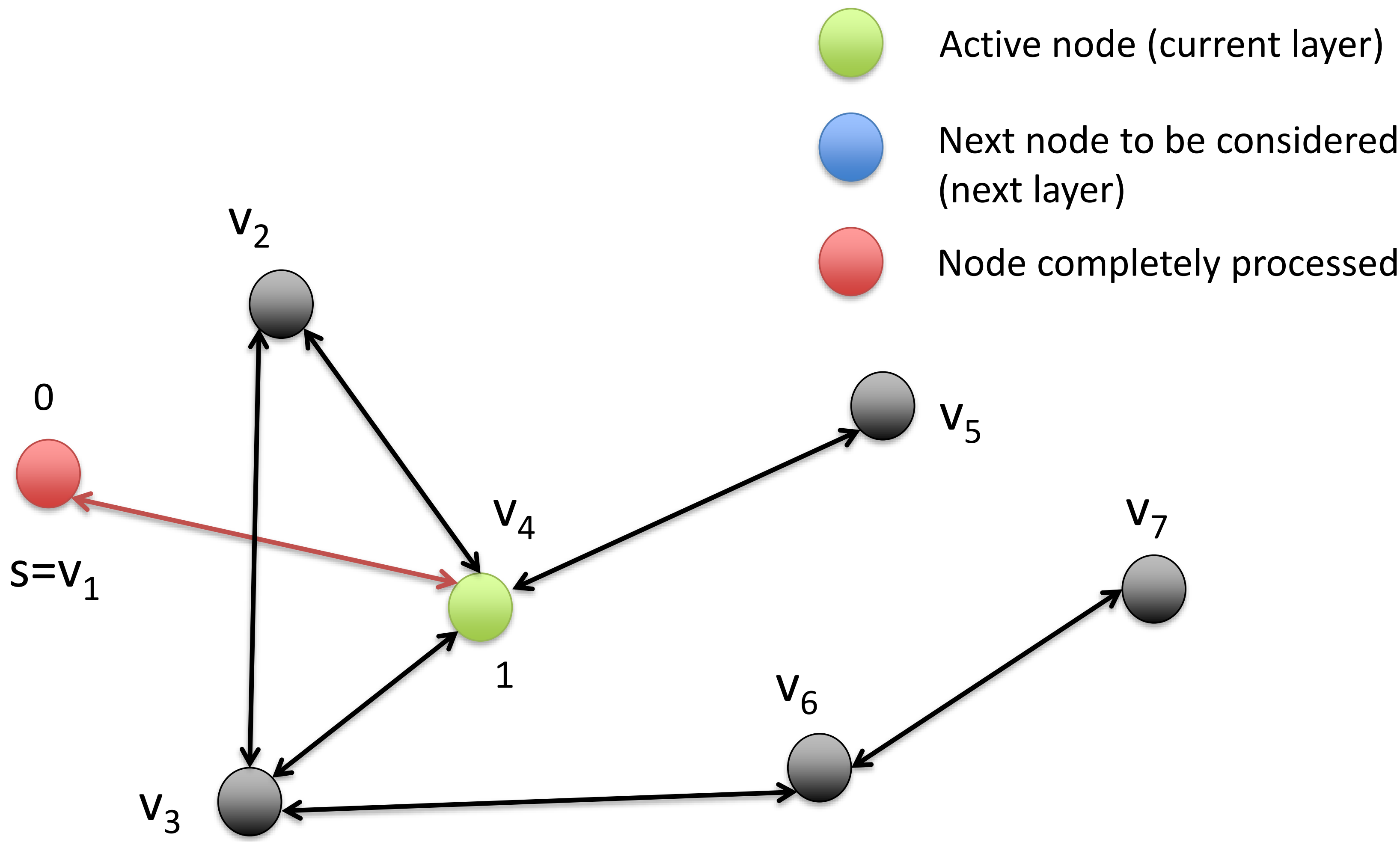
Breadth-first-search (BFS)



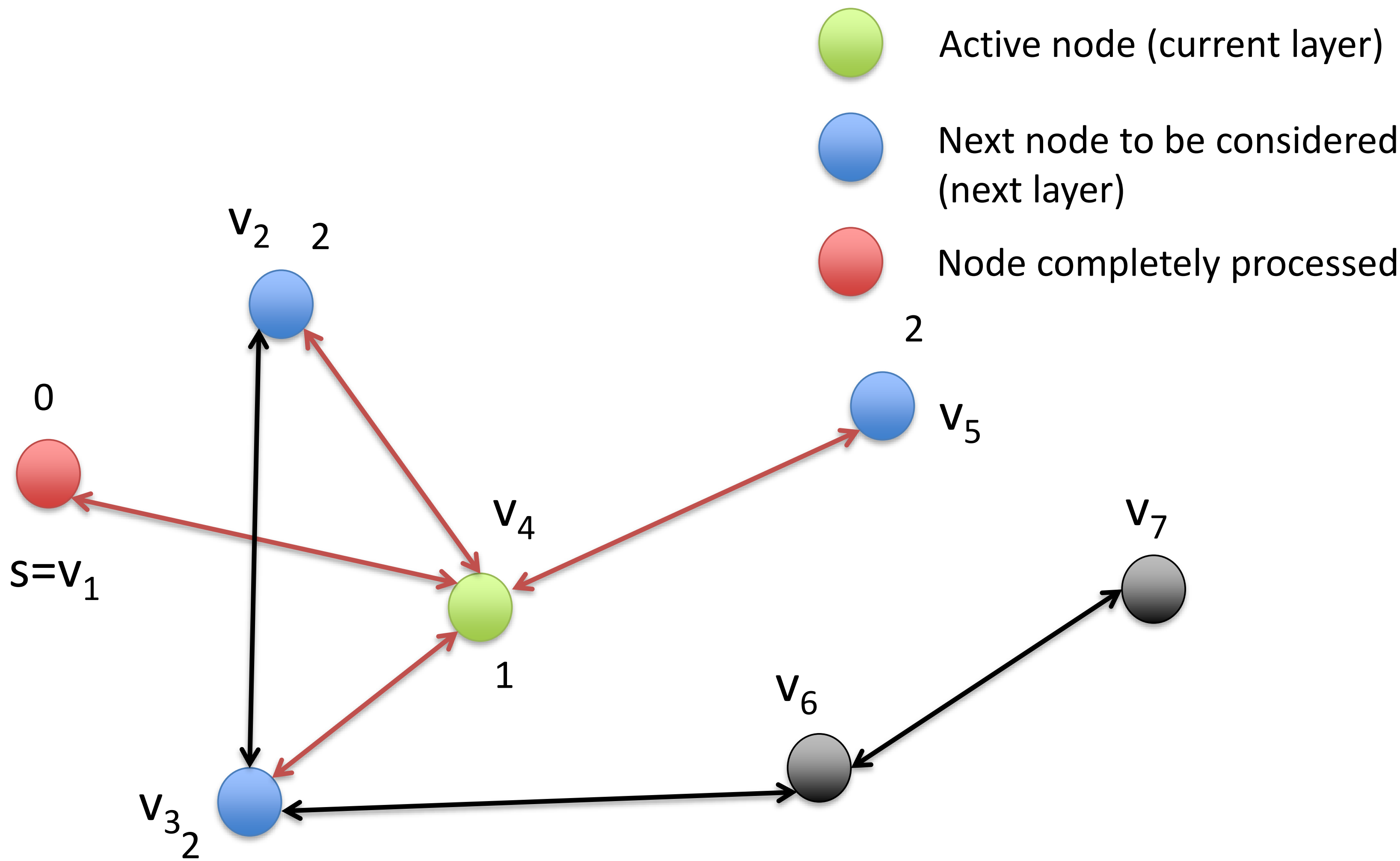
Breadth-first-search (BFS)



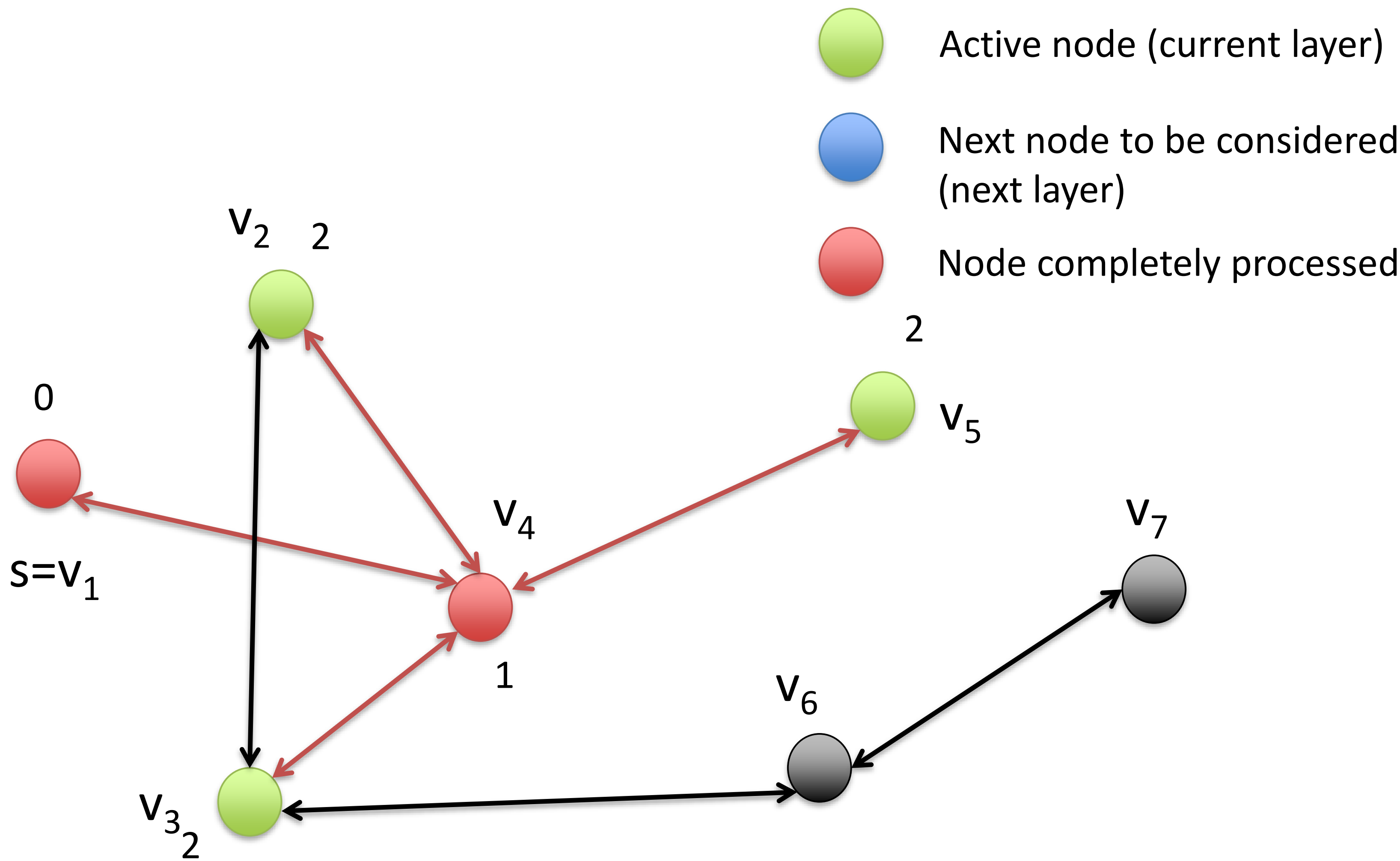
Breadth-first-search (BFS)



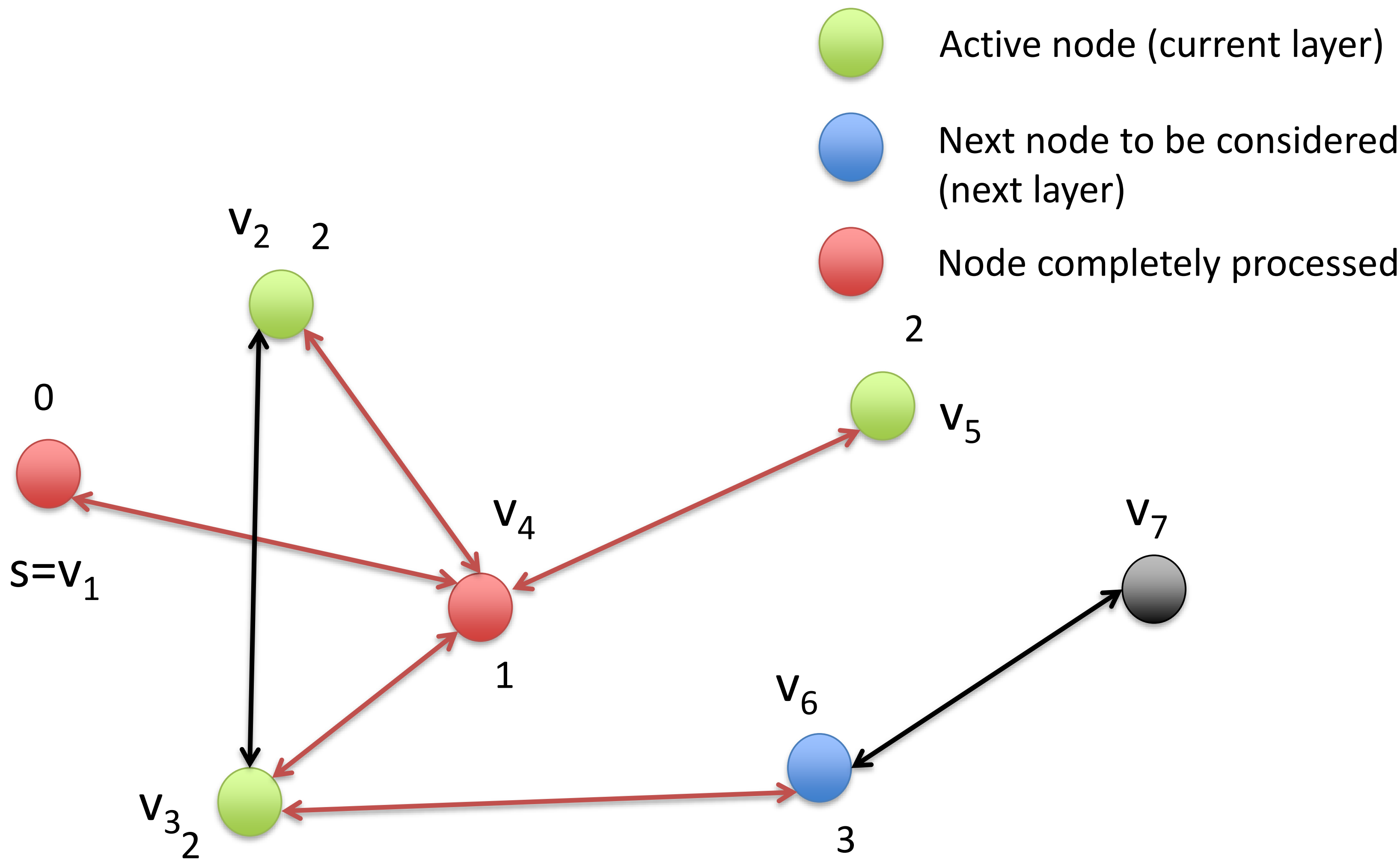
Breadth-first-search (BFS)



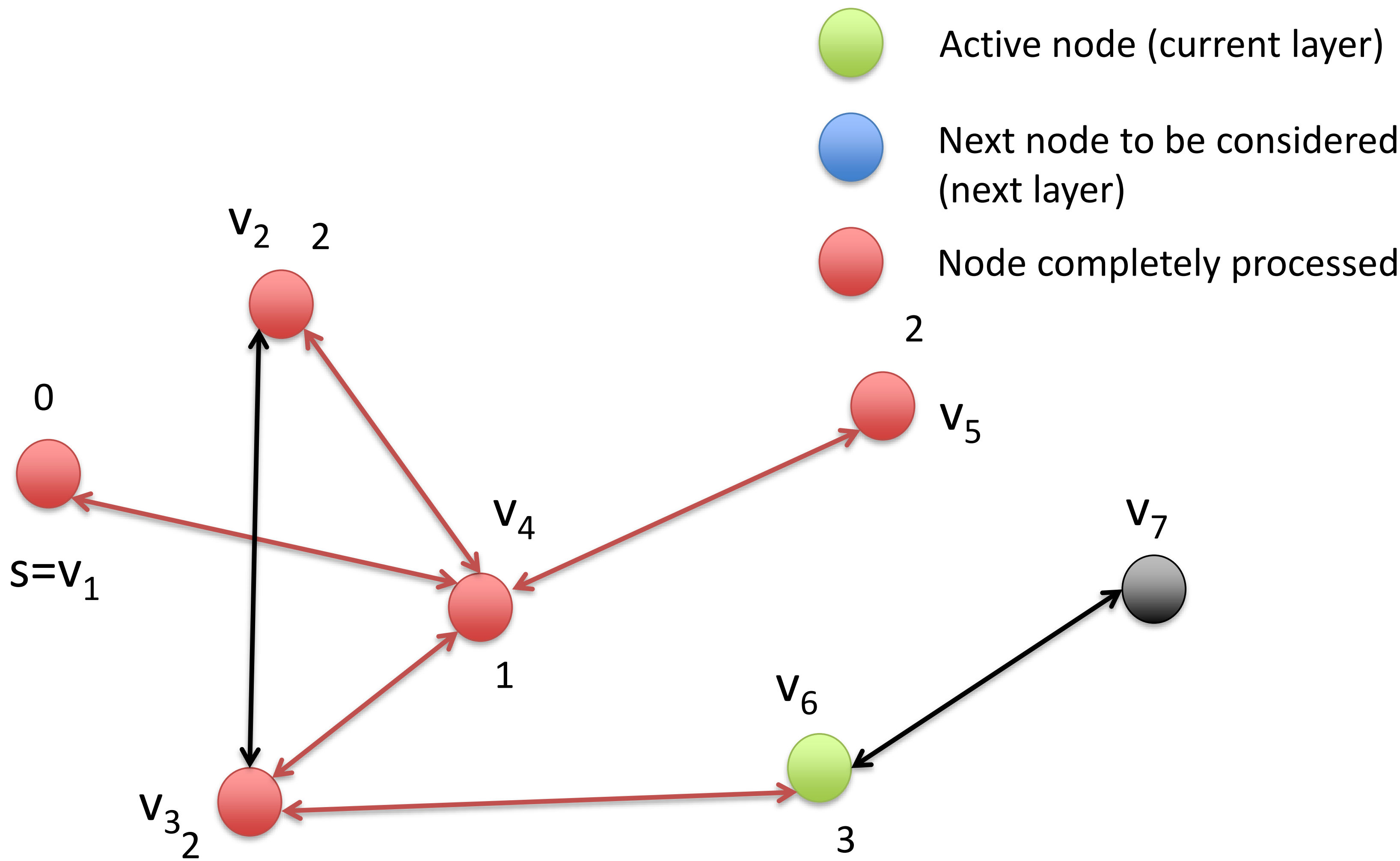
Breadth-first-search (BFS)



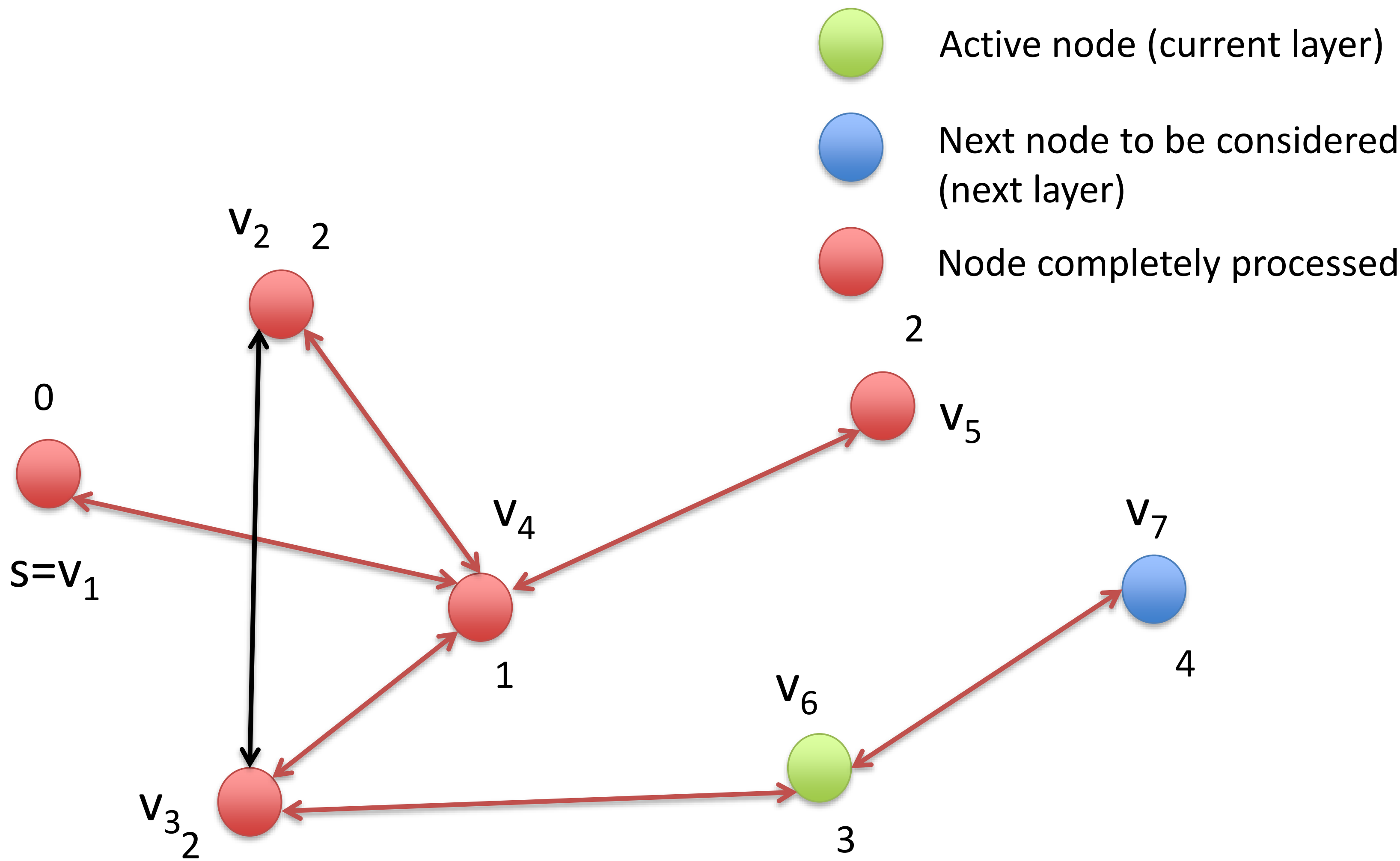
Breadth-first-search (BFS)



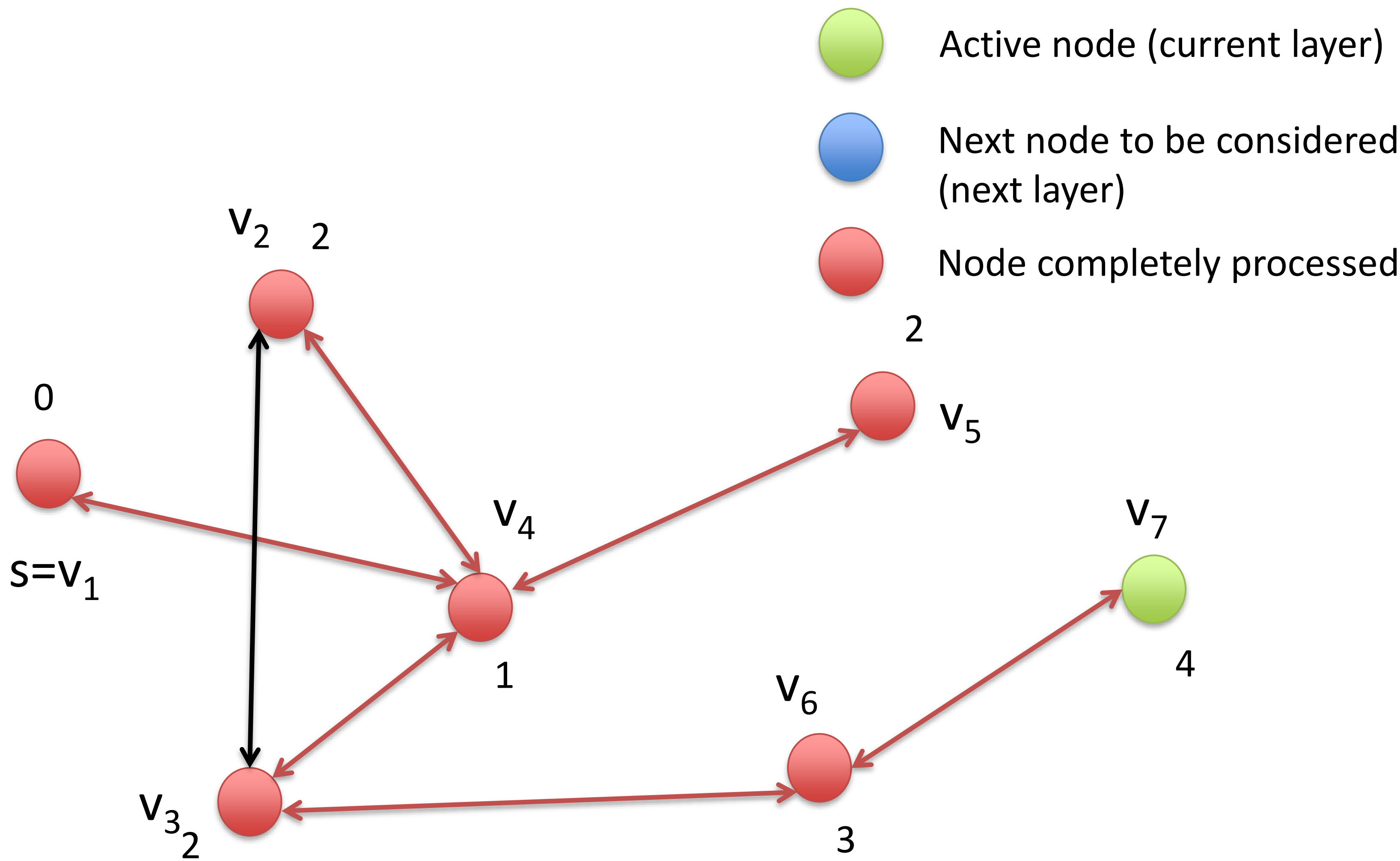
Breadth-first-search (BFS)



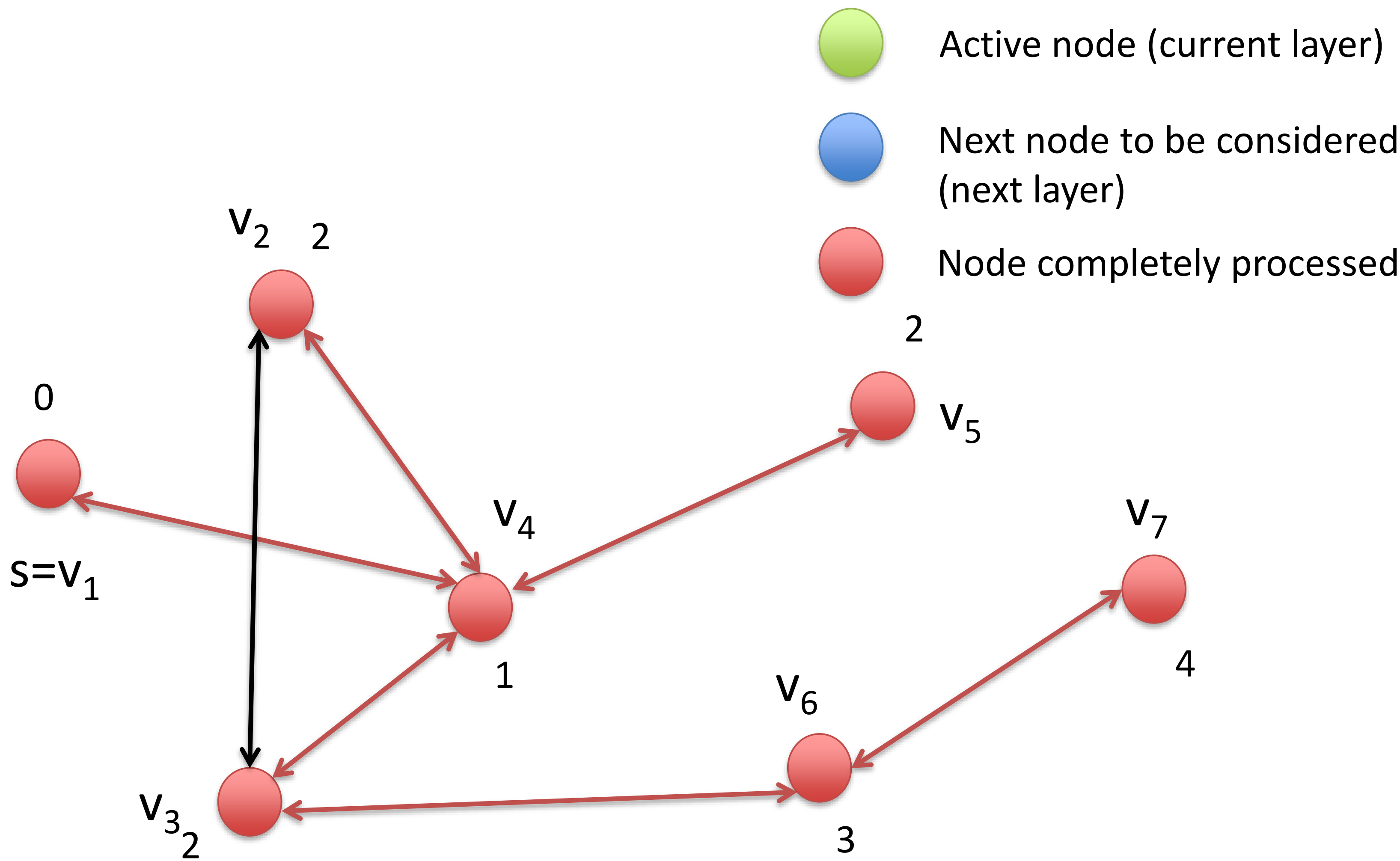
Breadth-first-search (BFS)



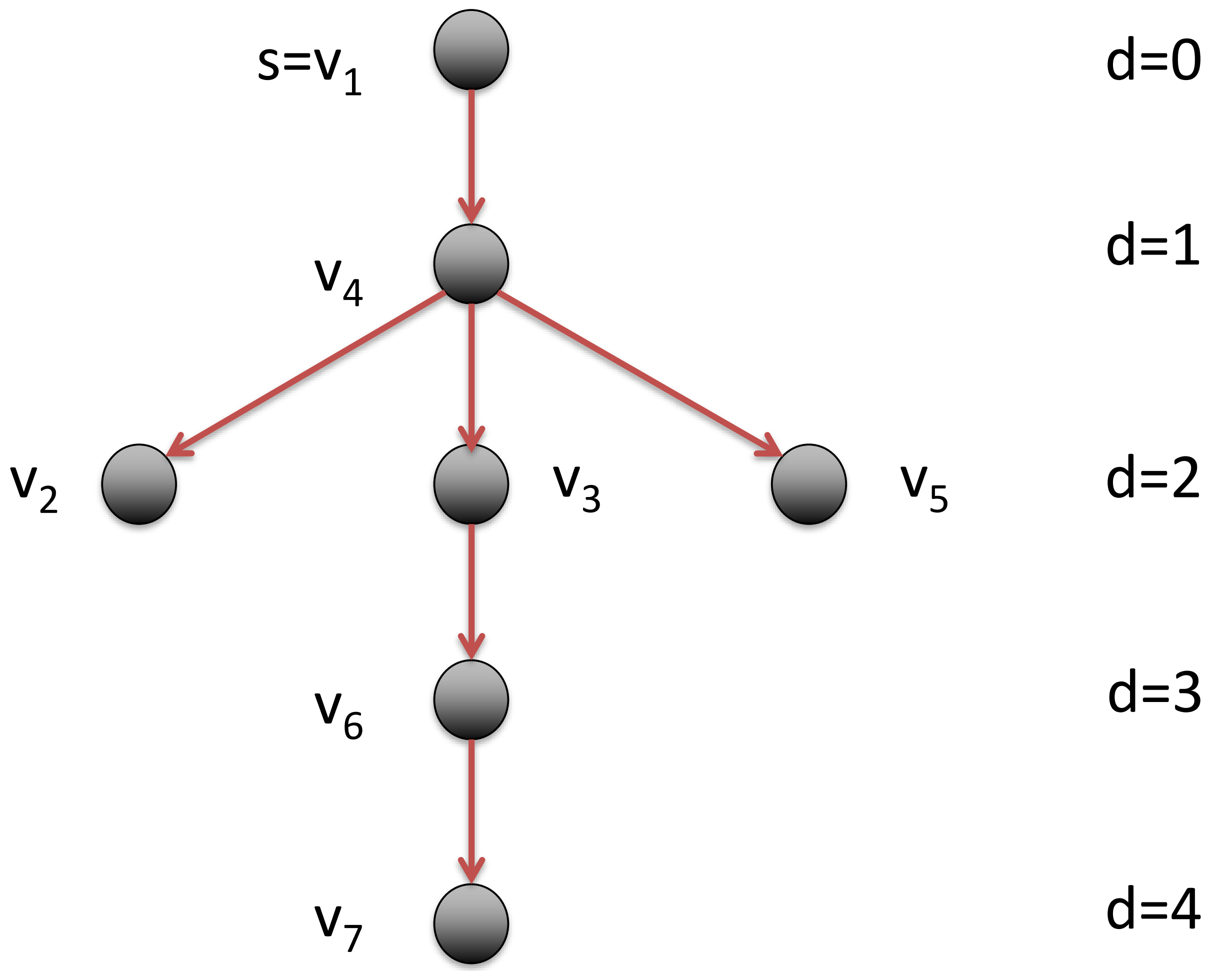
Breadth-first-search (BFS)



Breadth-first-search (BFS)



Breadth-First-Search Tree



Implementation

- Use Adjacency Array and Priority Queues Q.
- We introduce each node into the Priority Queue only once. (Time O(n))
- We only consider a node in the Priority Queues together with its edges once. (O(n+m))
- Updating the distance vector and the parent vector is done once for every node. (Time O(n))
- Total Runtime: O(n+m).

Pseudo-code Breadth-First-Search

Green nodes

Blue nodes

Green nodes become red, blue nodes become green

```
Function bfs(s : NodeId) : (NodeArray of NodeId) × (NodeArray of 0..n)
    d = ⟨∞,...,∞⟩ : NodeArray of NodeId                          // distance from root
    parent = ⟨⊥,...,⊥⟩ : NodeArray of NodeId
    d[s] := 0
    parent[s] := s                                               // self-loop signals root
    Q = ⟨s⟩ : Set of NodeId                                      // current layer of BFS tree
    Q' = ⟨⟩ : Set of NodeId                                      // next layer of BFS tree
    for ℓ := 0 to ∞ while Q ≠ ⟨⟩ do                              // explore layer by layer
        invariant Q contains all nodes with distance ℓ from s
        foreach u ∈ Q do
            foreach (u,v) ∈ E do                                 // scan edges out of u
                if parent(v) = ⊥ then                            // found an unexplored node
                    Q' := Q' ∪ {v}                               // remember for next layer
                    d[v] := ℓ + 1
                    parent(v) := u                               // update BFS tree
        (Q,Q') := (Q',⟨⟩)                                        // switch to next layer
    return (d, parent)                // the BFS tree is now {(v,w) : w ∈ V, v = parent(w)}
```

 See Mehlhorn/Sanders, Fig. 9.2

Overview

Depth-first-search

Strongly connected components

- Undirected graphs
- Directed graphs

Depth-First-Search

Given a directed graph G=(V,E).

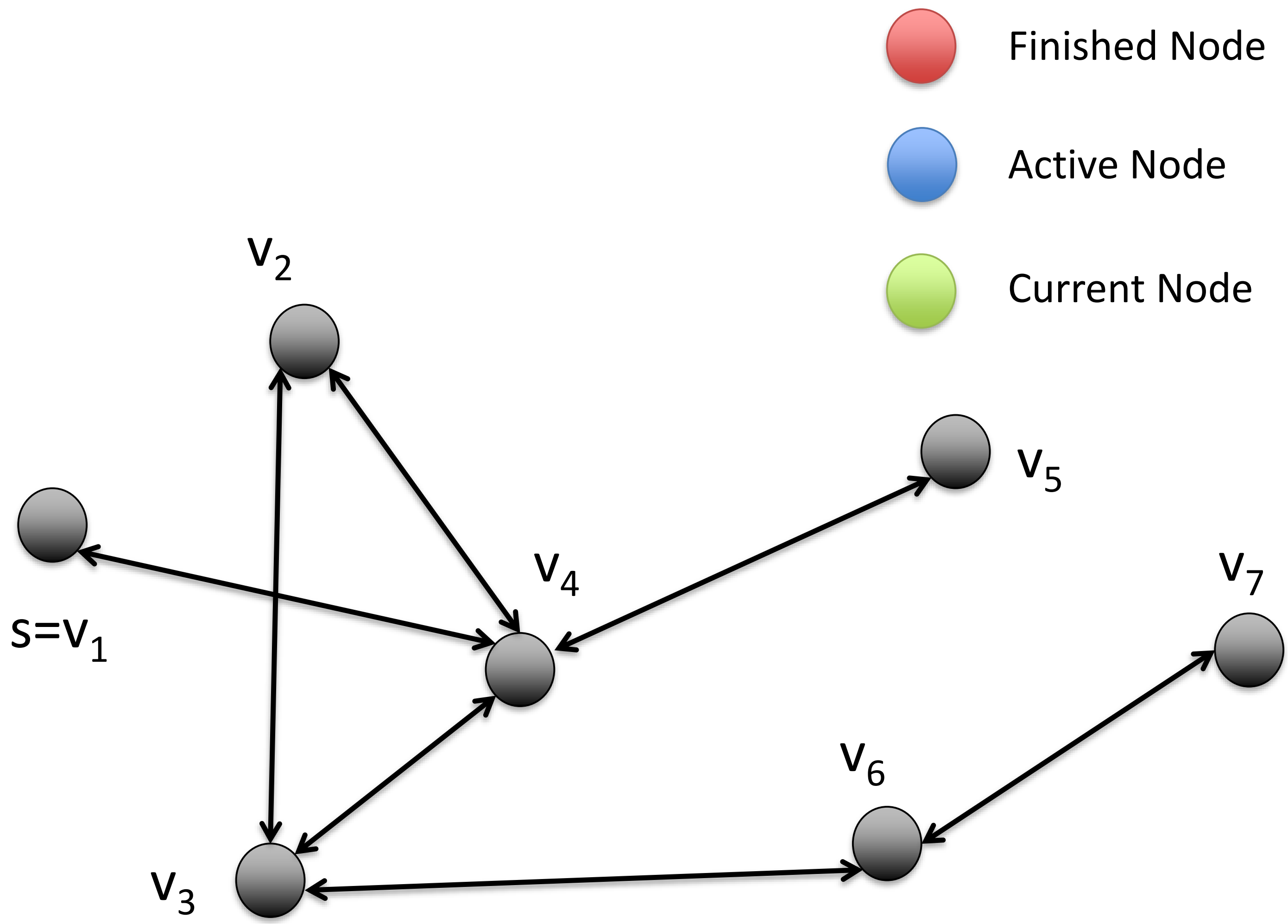
Idea for Depth-First-Search (DFS):

- Whenever you visit a vertex, explore in the next step one of its non-visited neighbours.

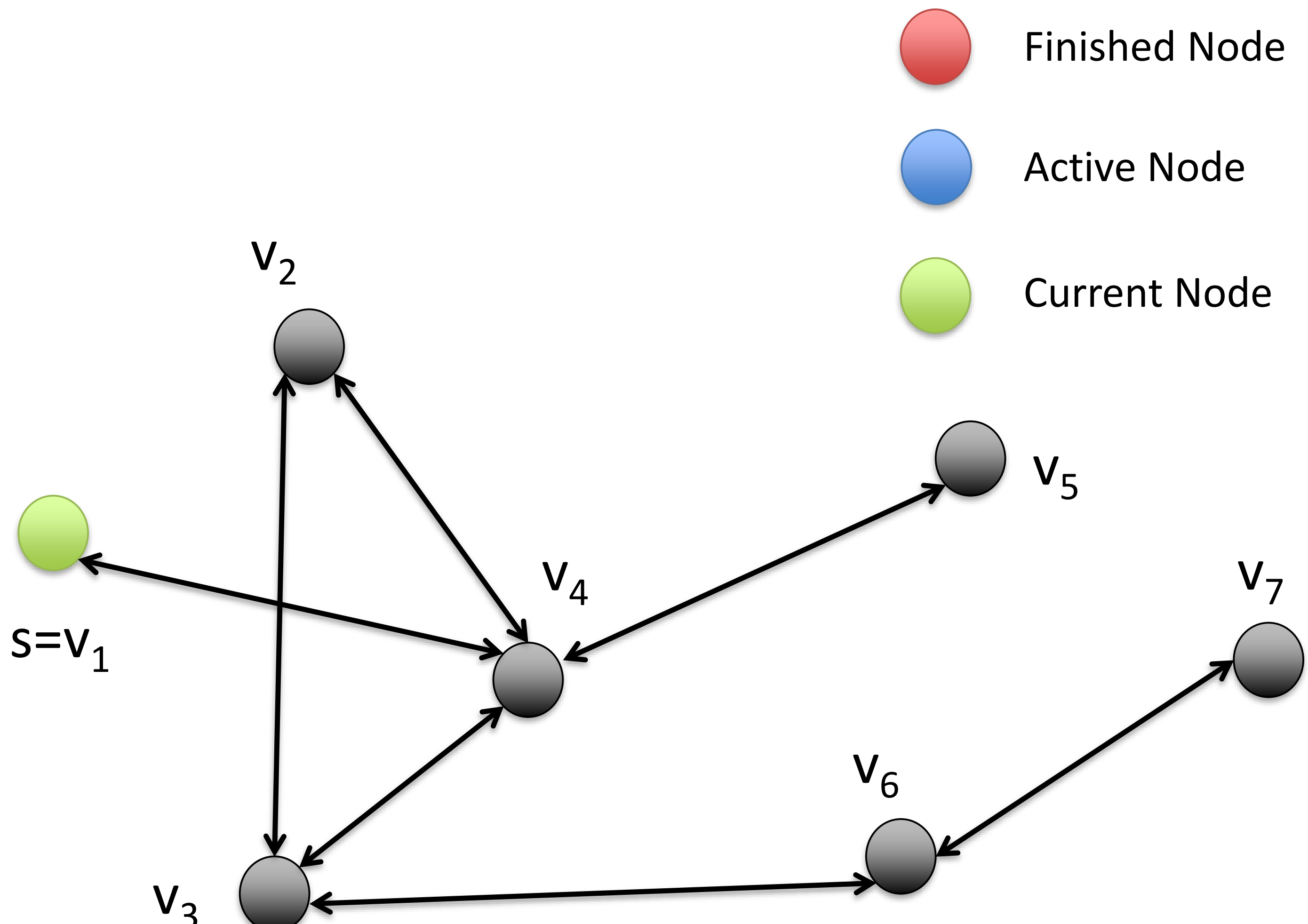
Implementation:

- When visiting a node, mark it as visited and recursively call DFS for one of its non-visited neighbors
- If there is no non-visited neighbor end recursive call.

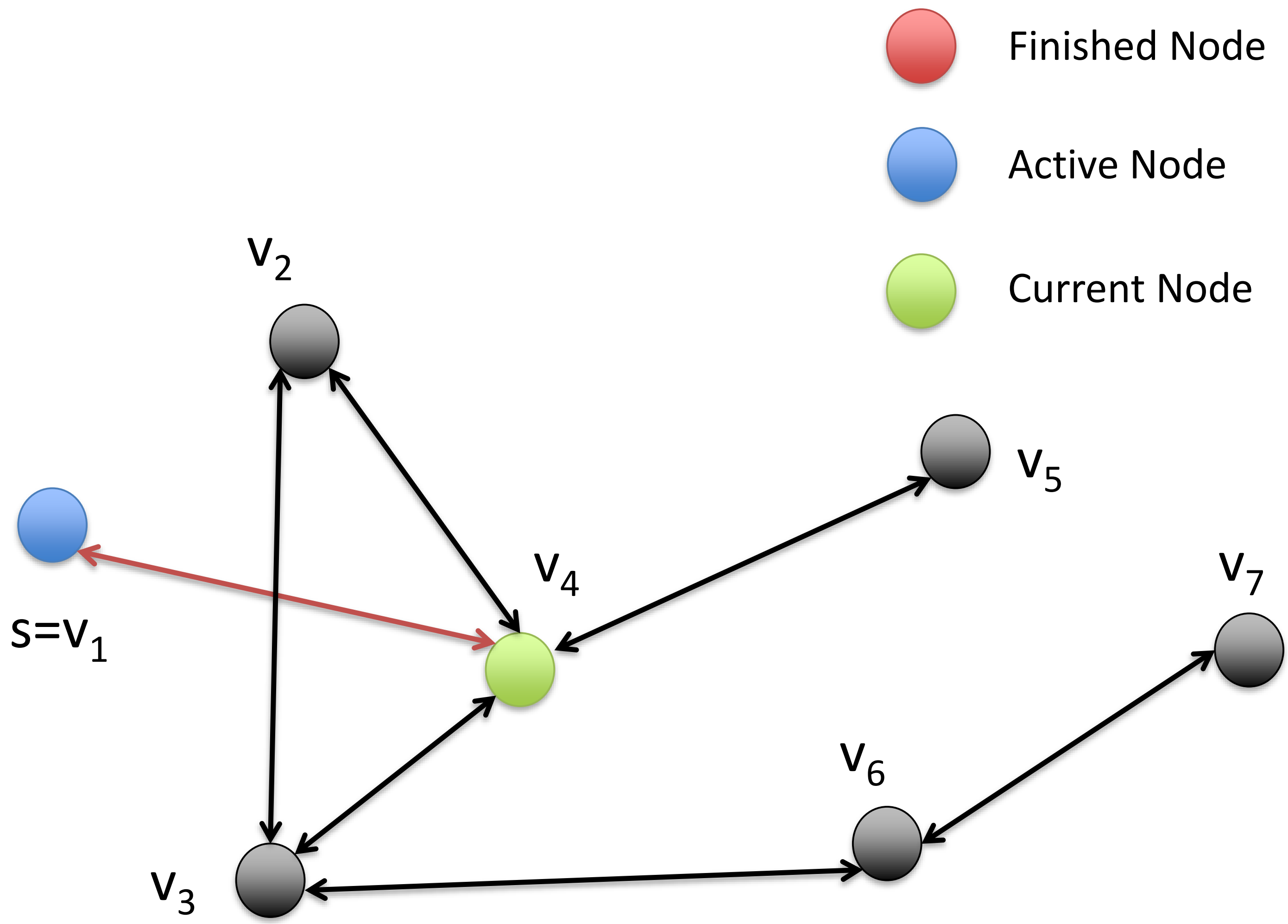
Depth-first-search (DFS)



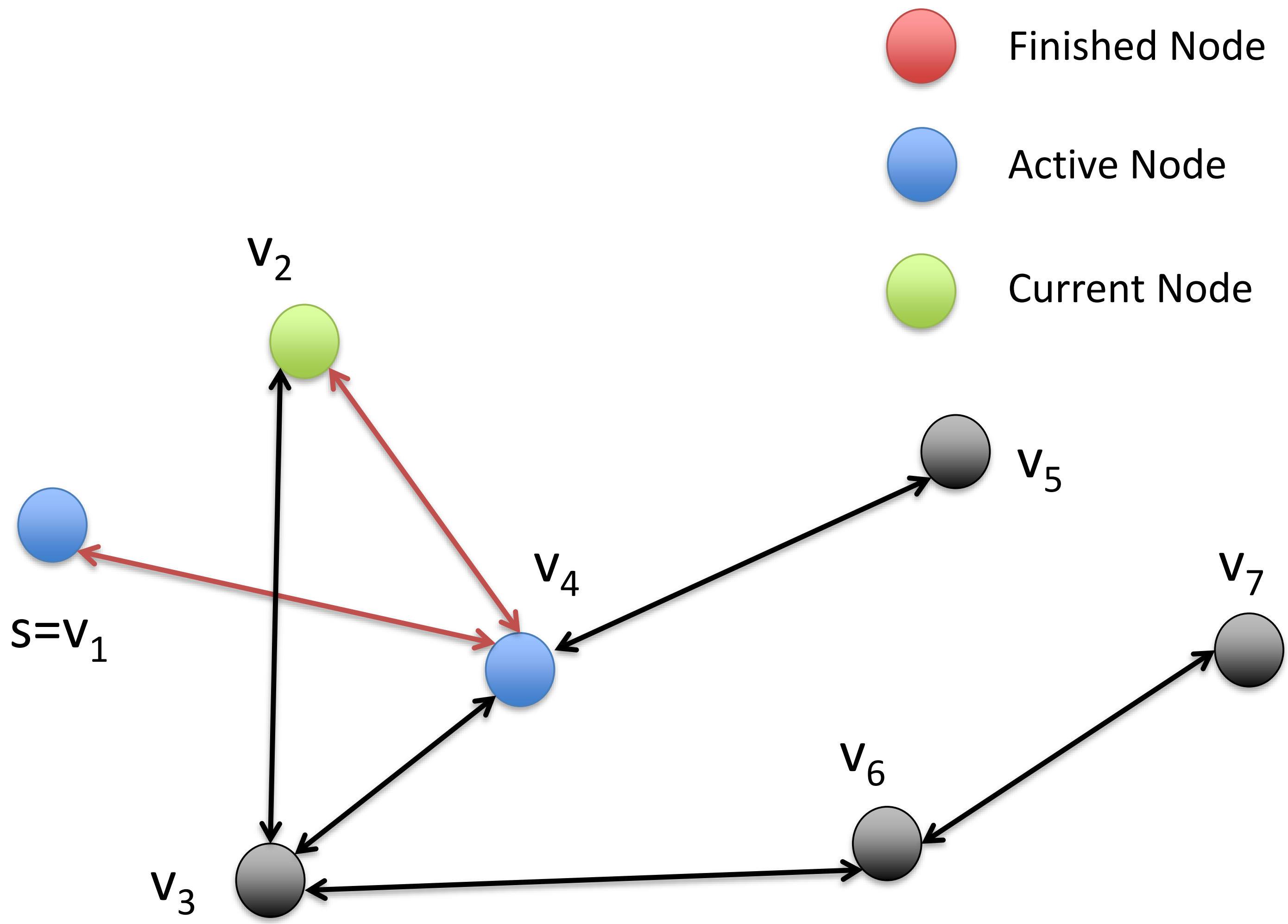
Depth-first-search (DFS)



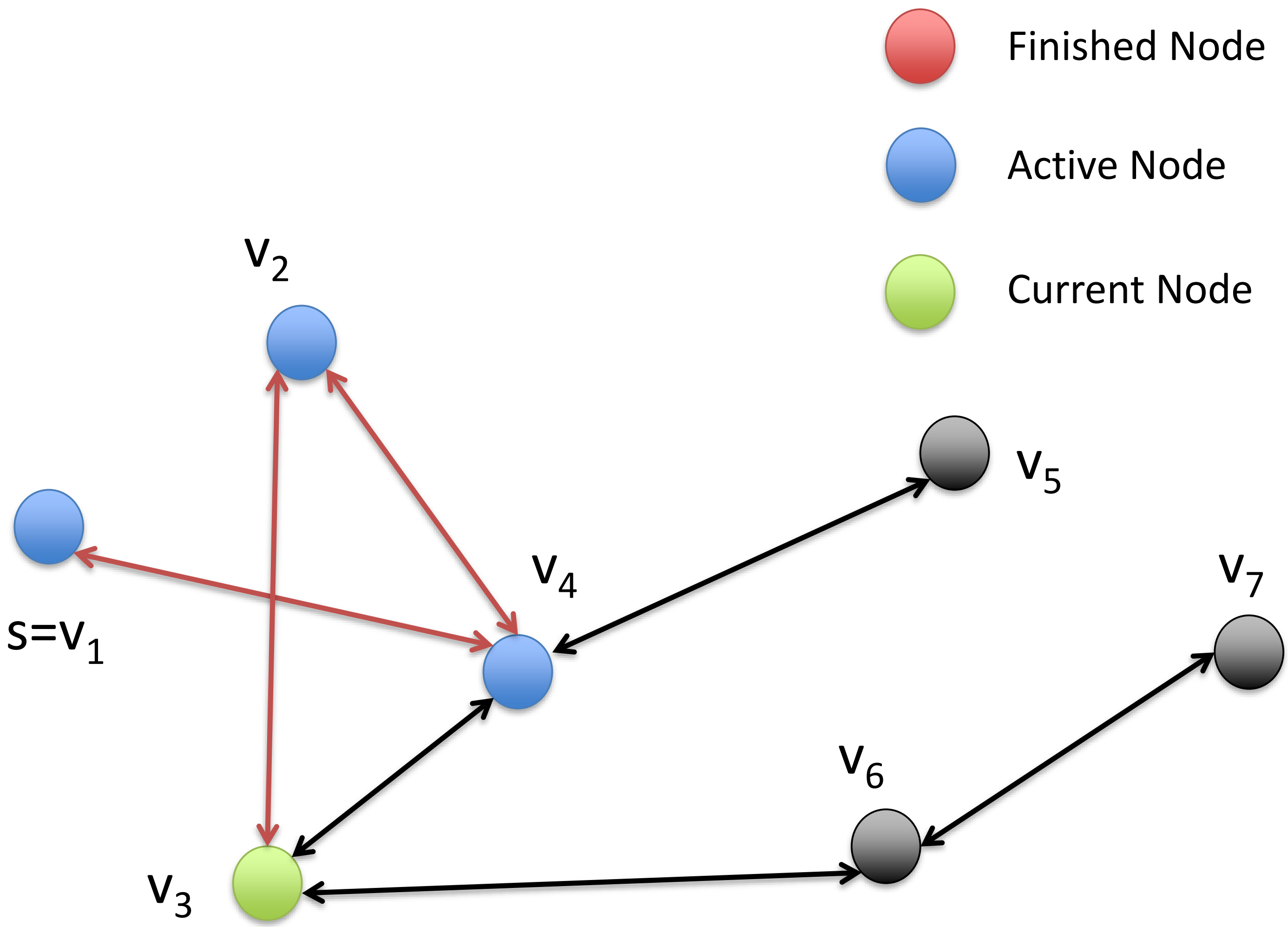
Depth-first-search (DFS)



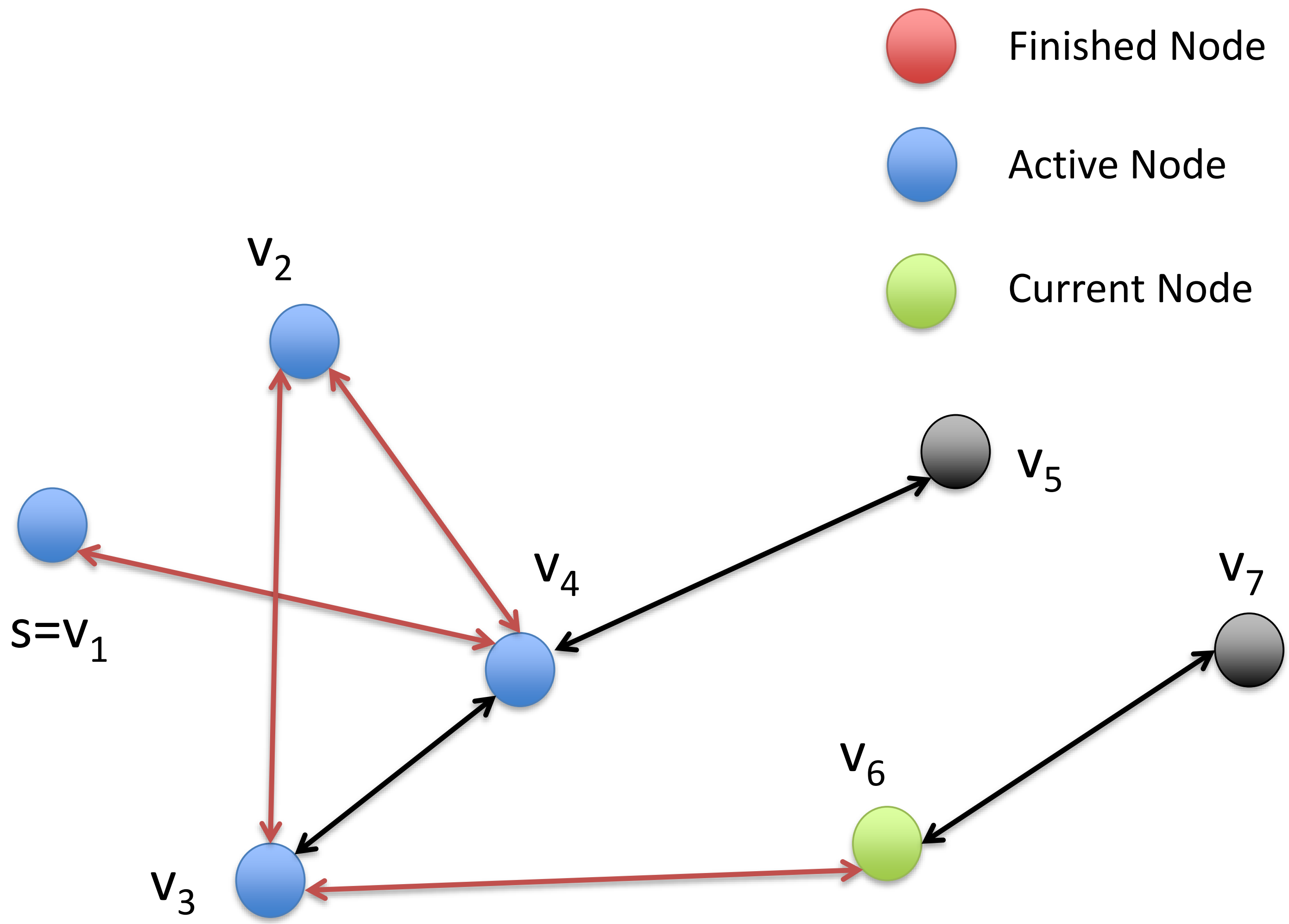
Depth-first-search (DFS)



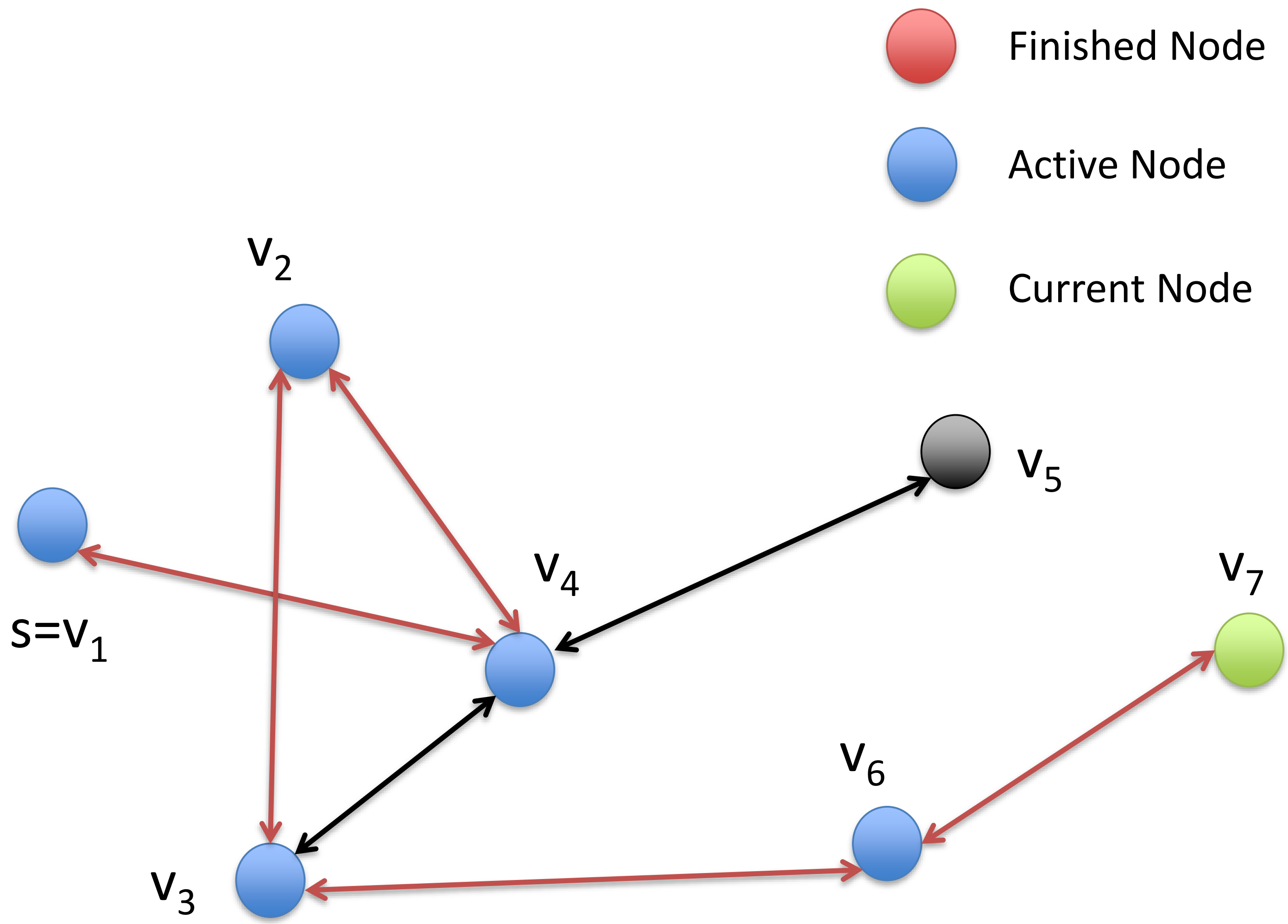
Depth-first-search (DFS)



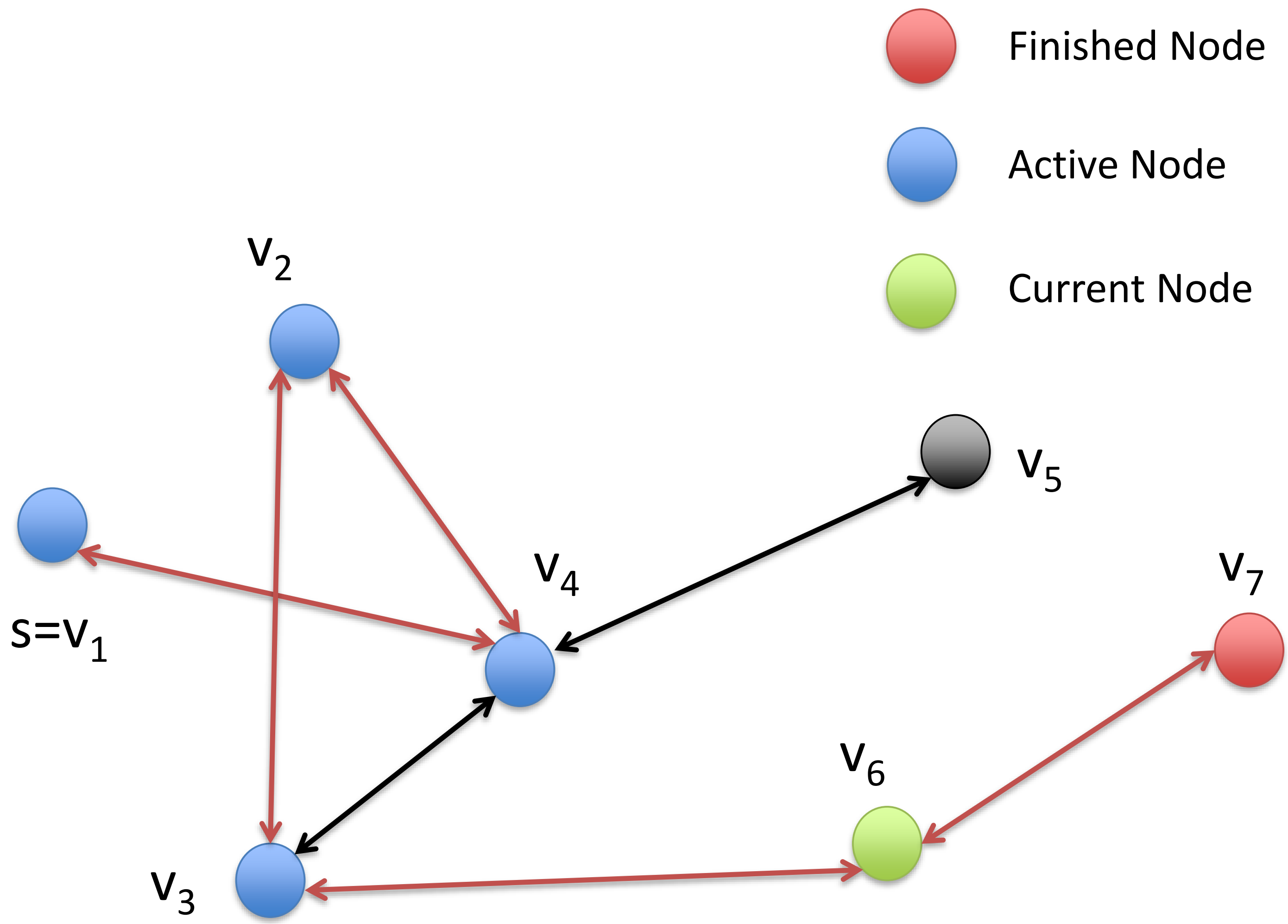
Depth-first-search (DFS)



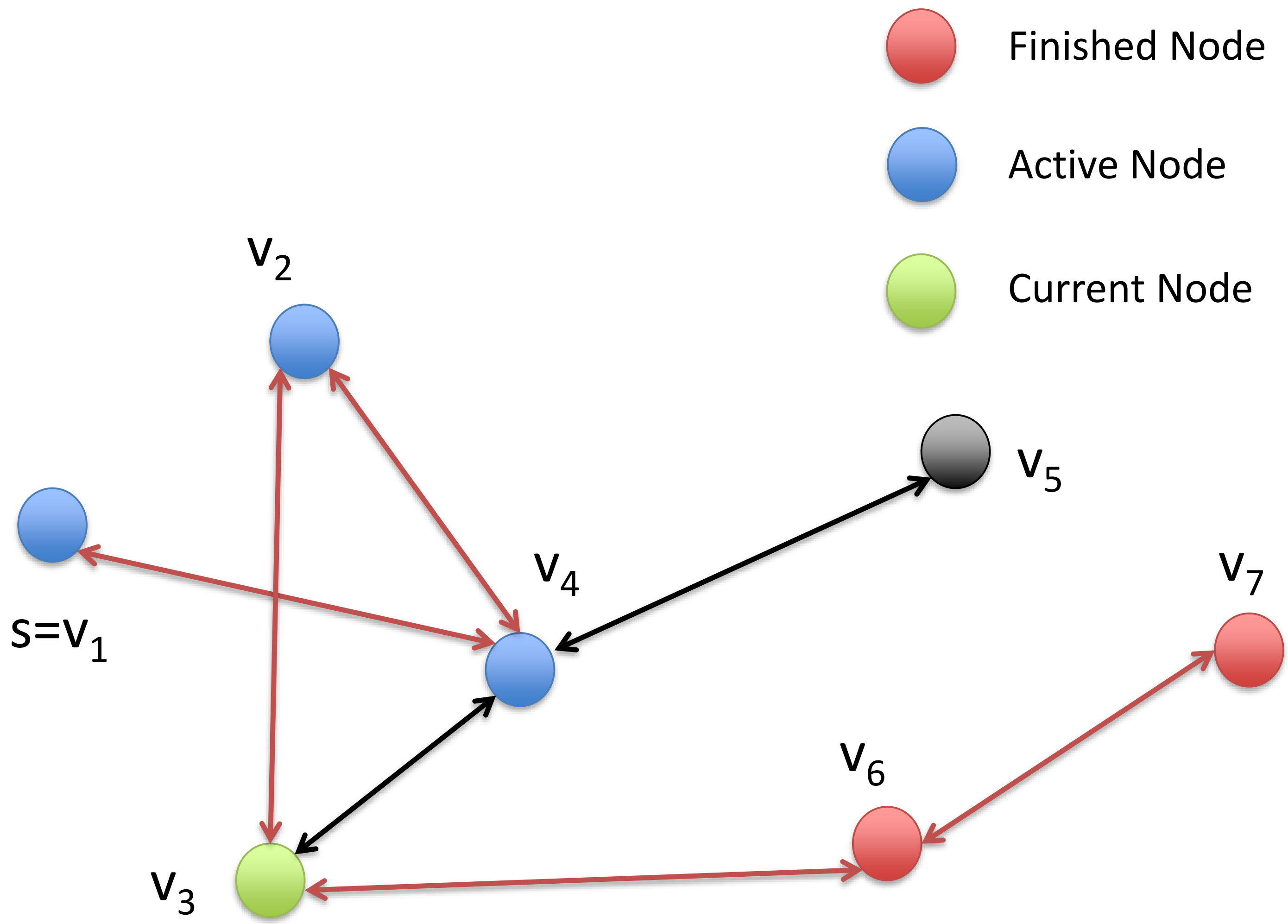
Depth-first-search (DFS)



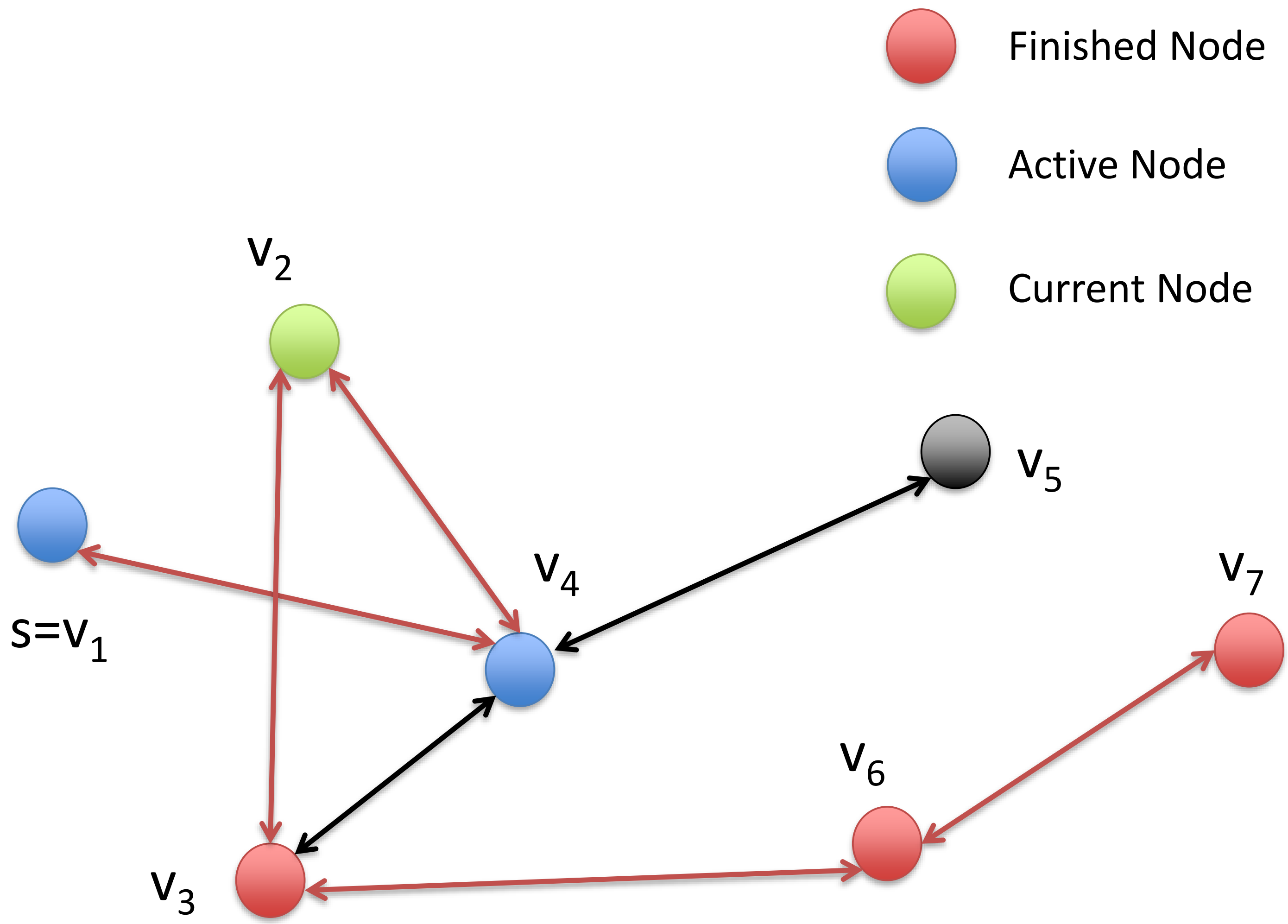
Depth-first-search (DFS)



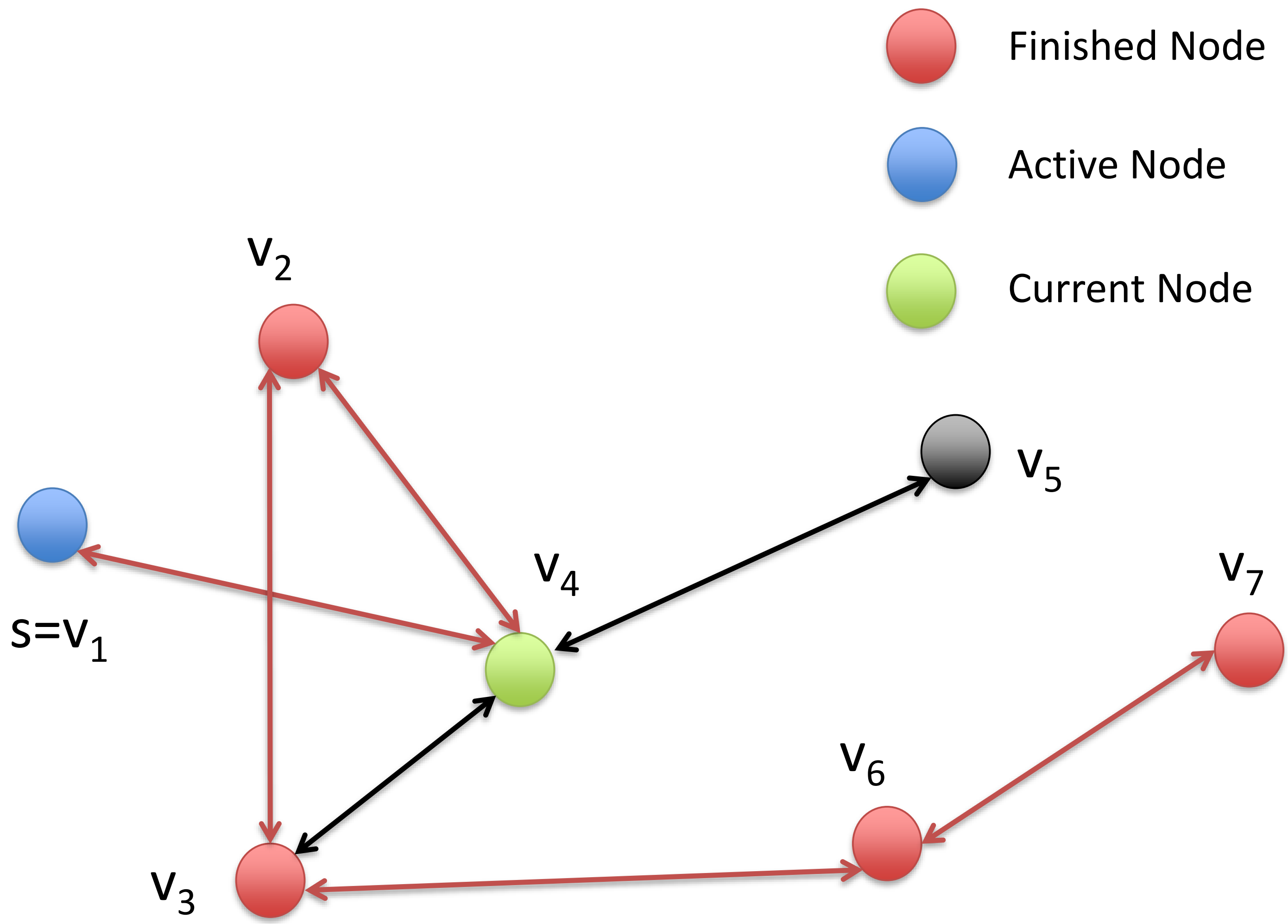
Depth-first-search (DFS)



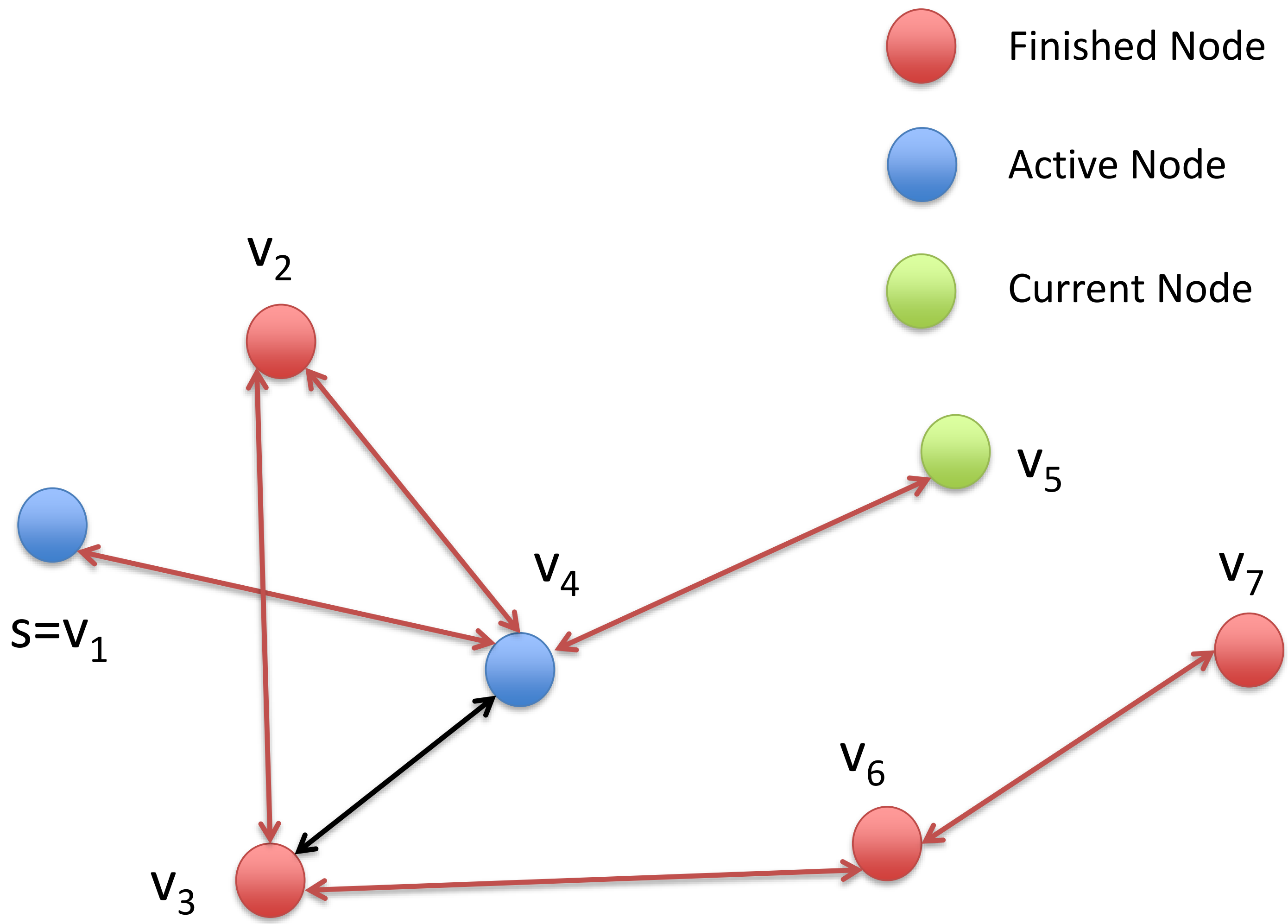
Depth-first-search (BFS)



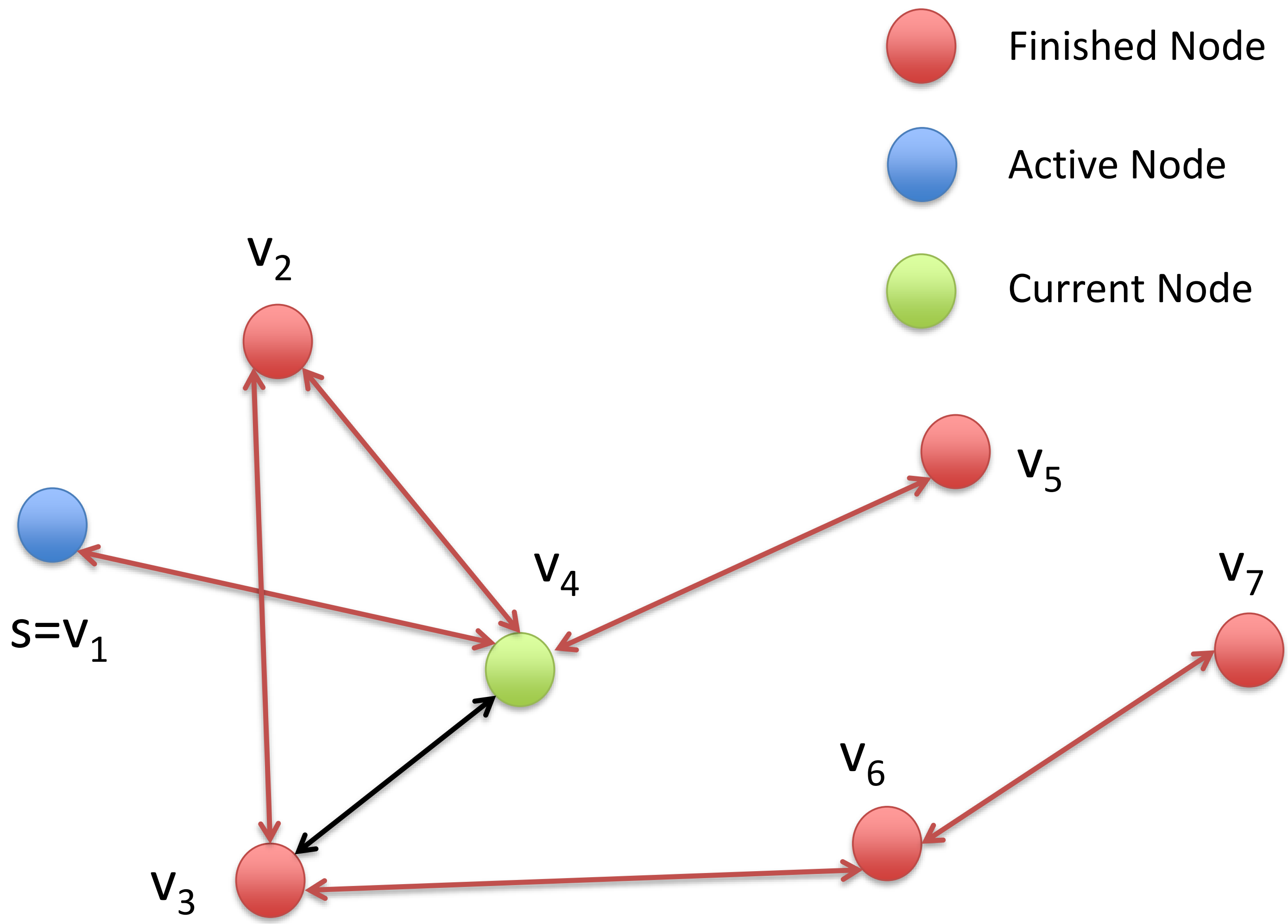
Depth-first-search (DFS)



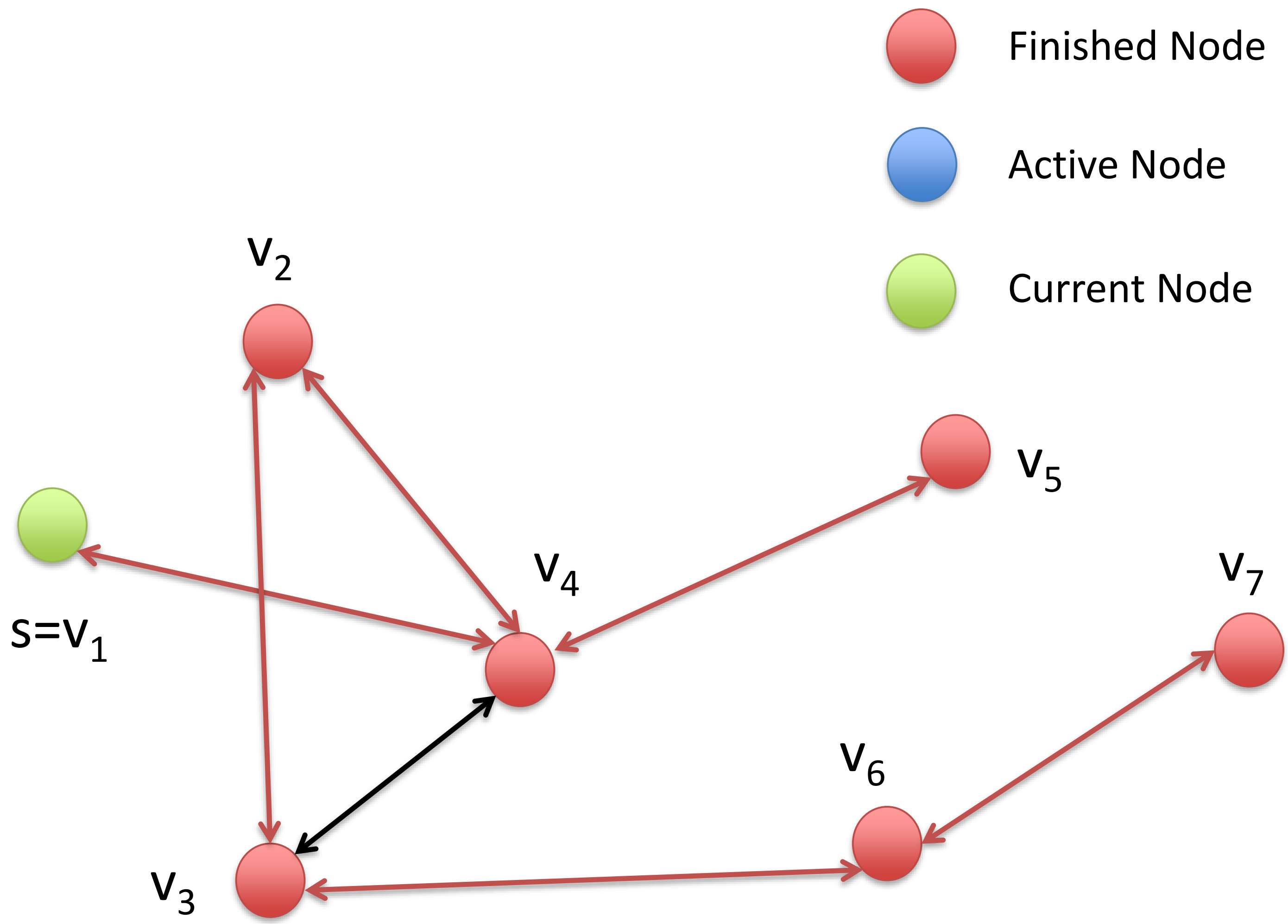
Depth-first-search (DFS)



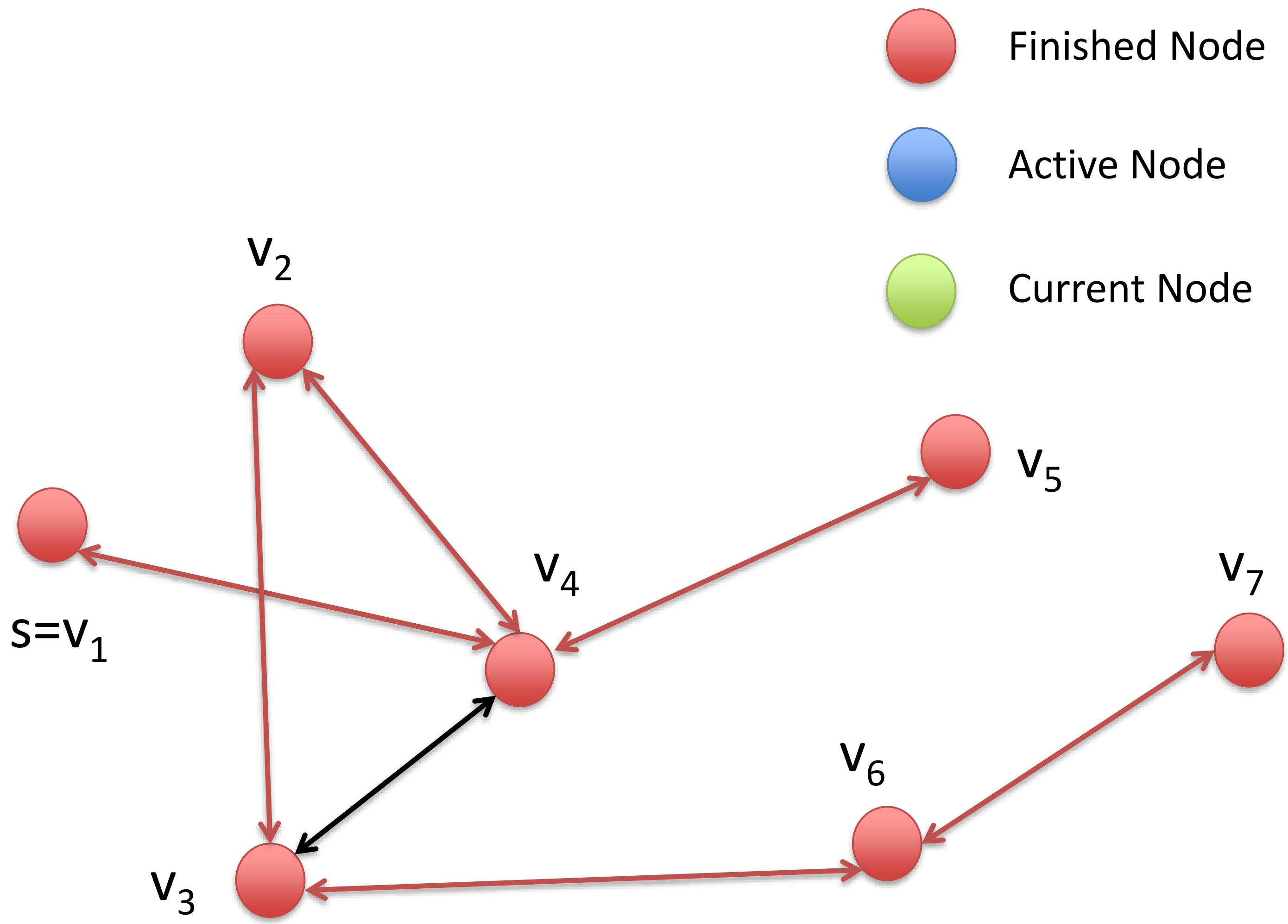
Depth-first-search (DFS)



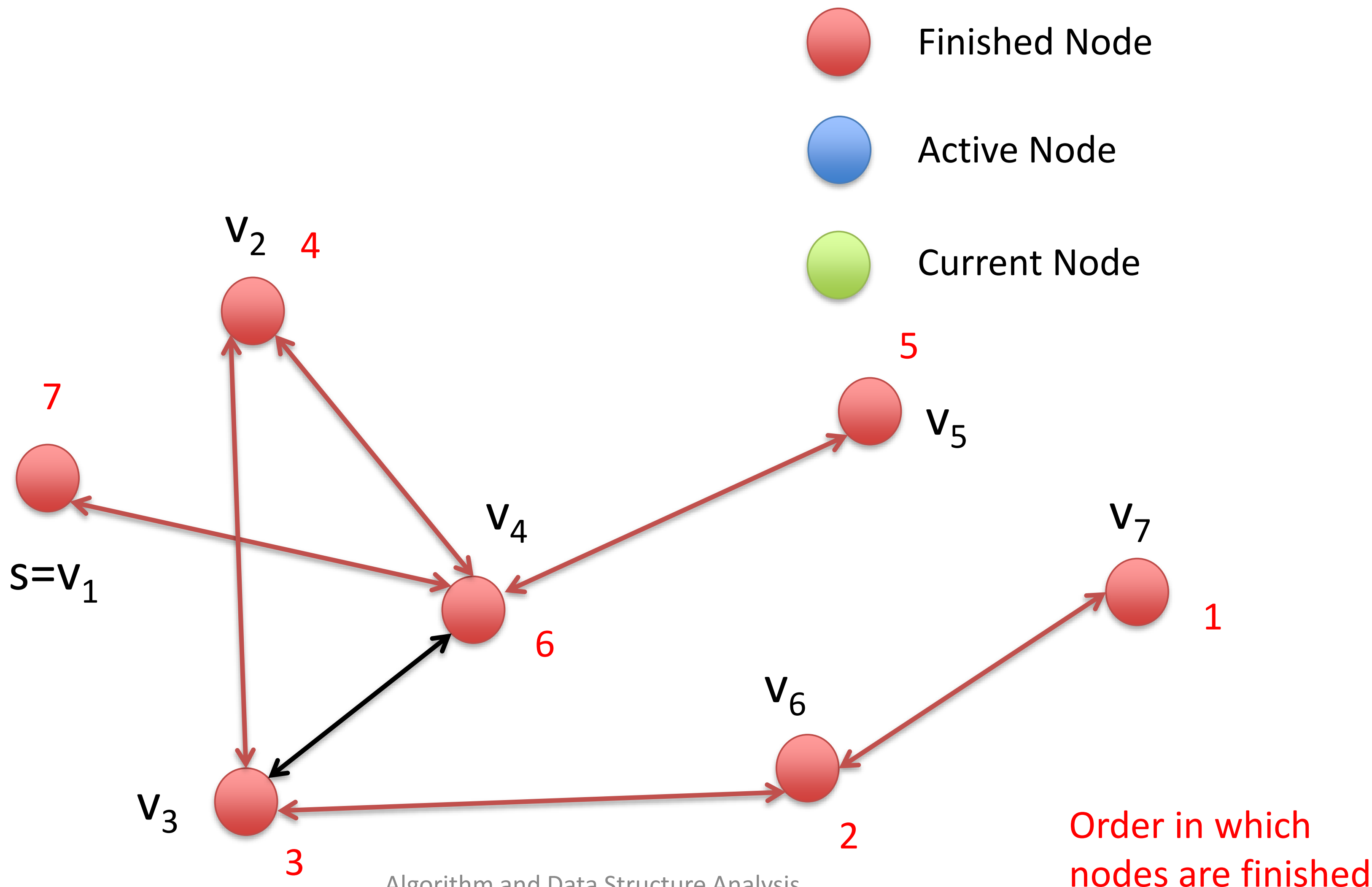
Depth-first-search (DFS)



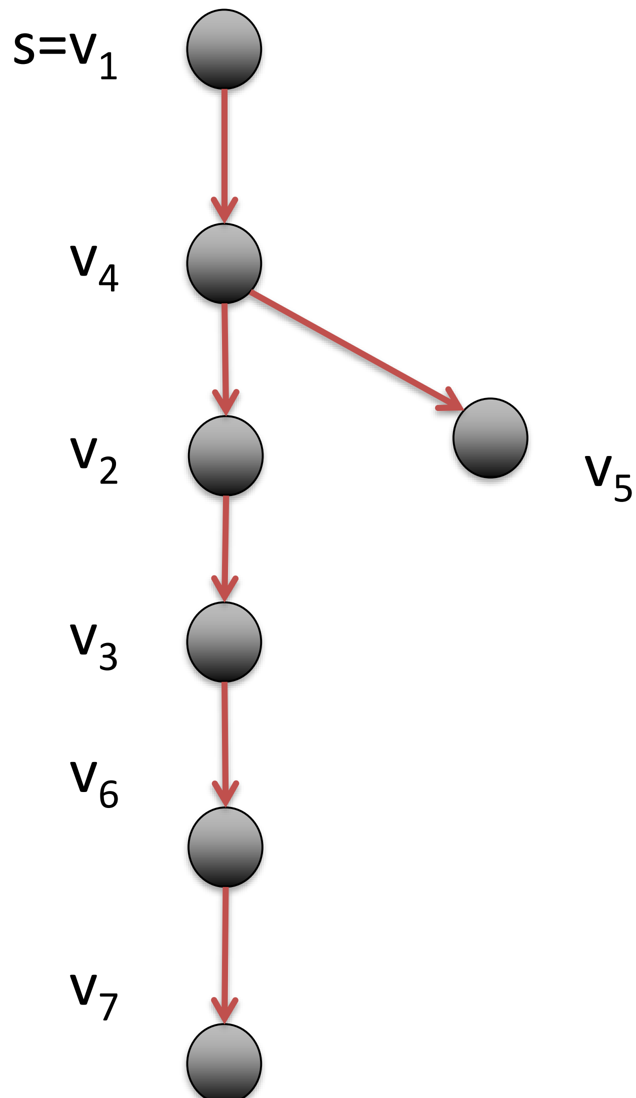
Depth-first-search (DFS)



Depth-first-search (DFS)



Depth-First-Search Tree



Depth-first search of a directed graph $G = (V, E)$

```
unmark all nodes
init
foreach s ∈ V do
    if s is not marked then
        mark s                                      // make s a root and grow
        root(s)                                     // a new DFS tree rooted at it.
        DFS(s, s)

Procedure DFS(u, v : NodeId)                        // Explore v coming from u.
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)   // w was reached before
        else    traverseTreeEdge(v, w)                  // w was not reached before
                mark w
                DFS(v, w)
    backtrack(u, v)                                 // return from v along the incoming edge
```

 See Mehlhorn/Sanders, Fig. 9.4

Runtime DFS

- DFS explores each node and its outgoing edges once.
- Use Adjacency List or an Adjacency Array for representing the graph and remember which edges have already been traversed.
- Runtime: $O(m+n)$

Strongly connected components

Two nodes u and v belong to the same strongly connected component if there is a path from u to v and a path from v to u.

Task: Compute the strongly connected components of a given graph.

Undirected Graphs

Compute strongly connected components of a given undirected graph.

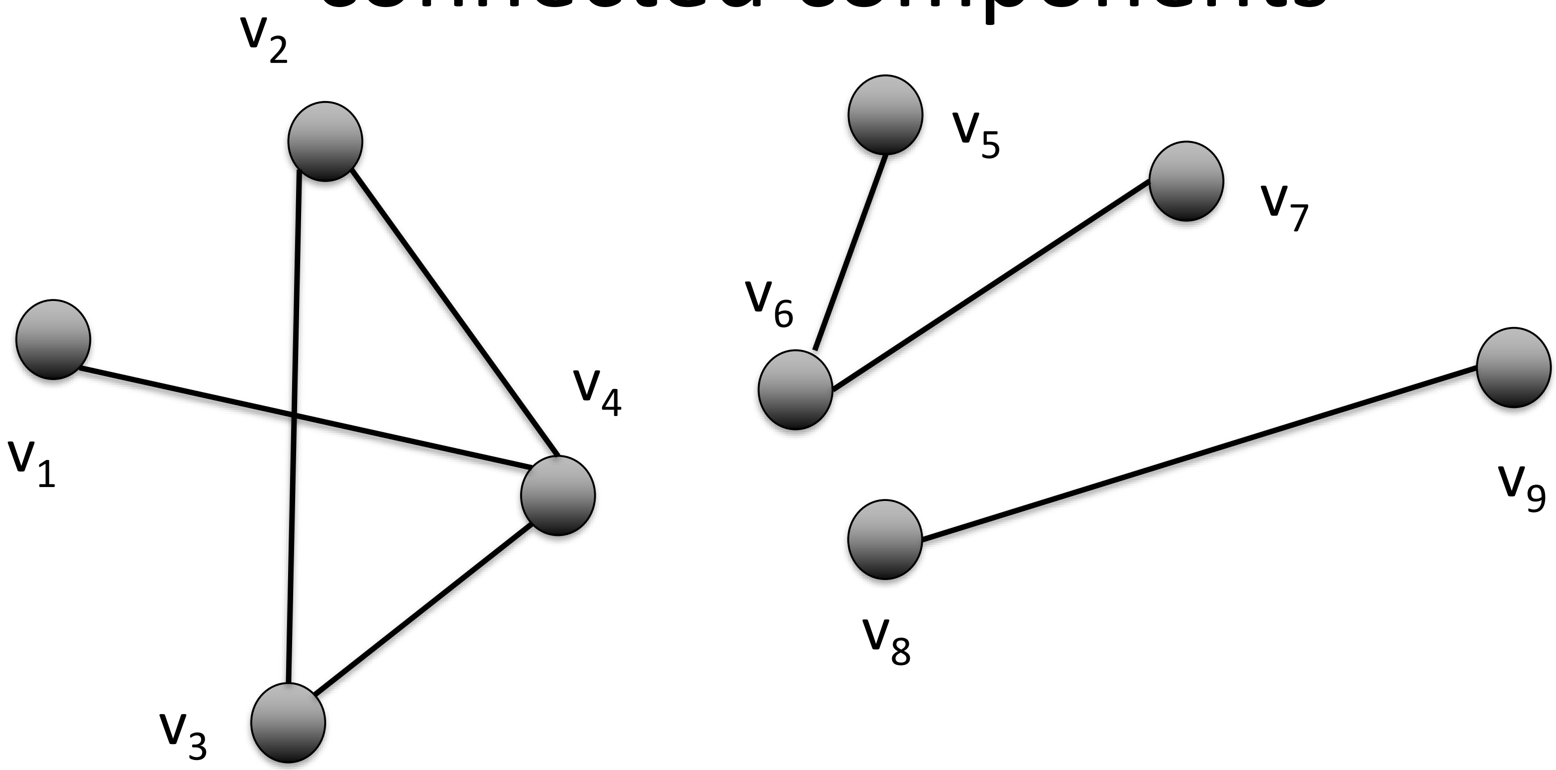
Observation:

- If there is a path from u to v then there is also a path from v to u.

Algorithmic approach:

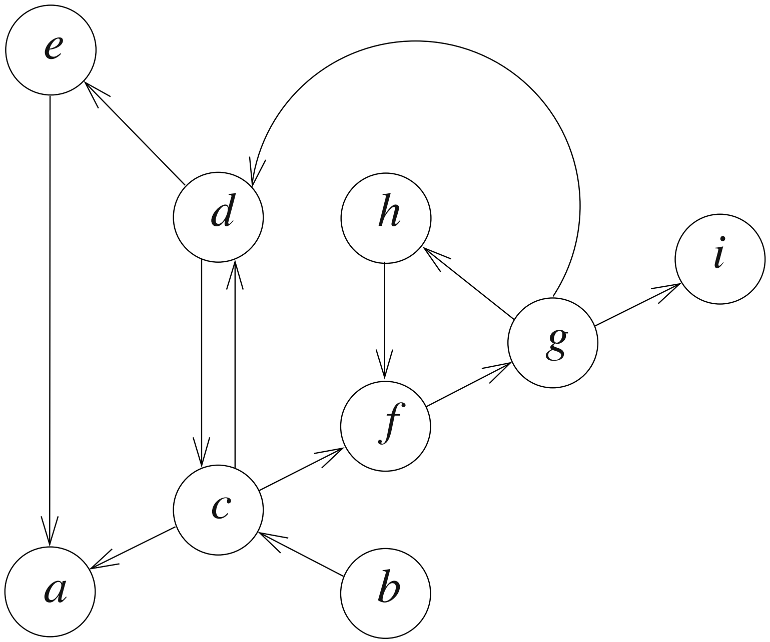
- Use DFS (or BFS) to compute the different connected components of the given undirected graph.
- Runtime O(m+n).

Undirected graph with three strongly connected components

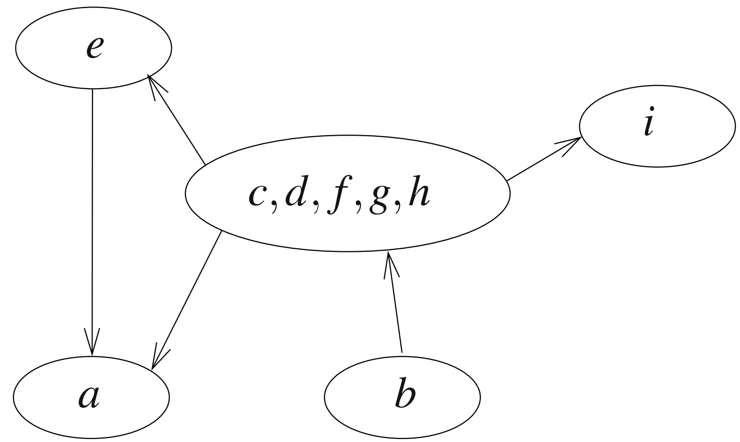


3 strongly connected components: $\{v_1, v_2, v_3, v_4\}$ $\{v_5, v_6, v_7\}$ $\{v_8, v_9\}$

Directed Graphs



Directed graph



Strongly connected components

How to compute the strongly connected components?

See Mehlhorn/Sanders, Fig. 9.5

Algorithm (strongly connected components of directed graph G)

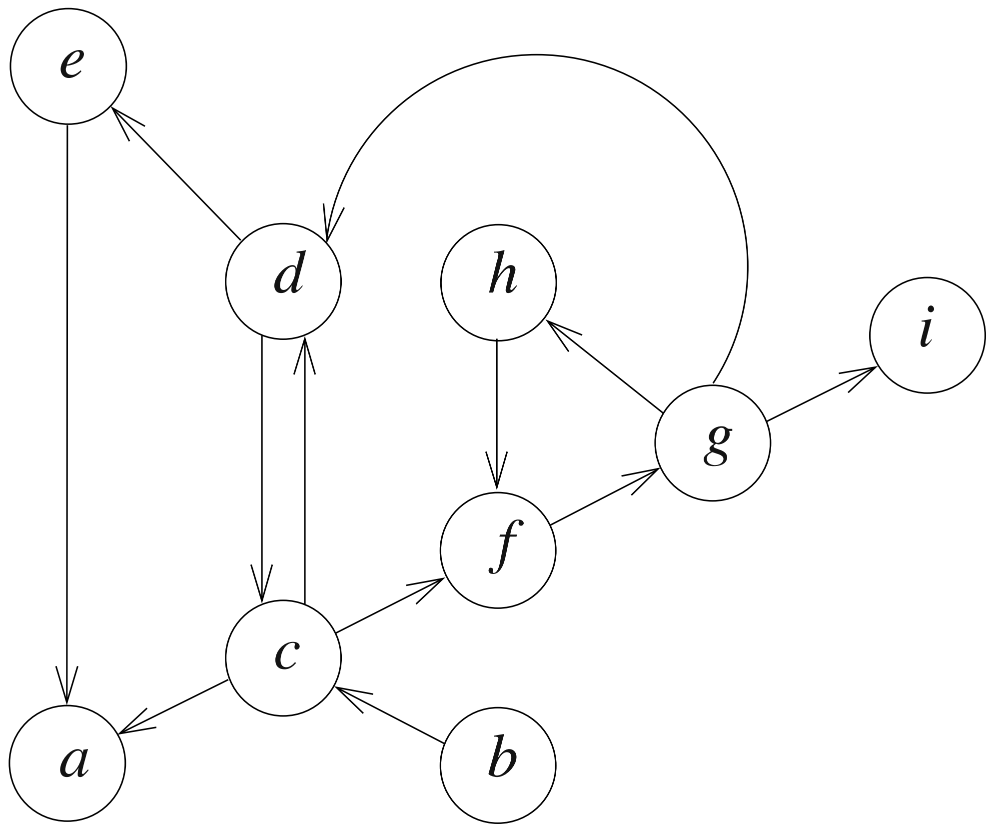
1. Run DFS on the given graph G. Number the nodes according to the termination of their recursive calls.
2. Compute the transpose graph G^T of G. It holds

$$(i, j) \in G^T \text{ if and only if } (j, i) \in G$$

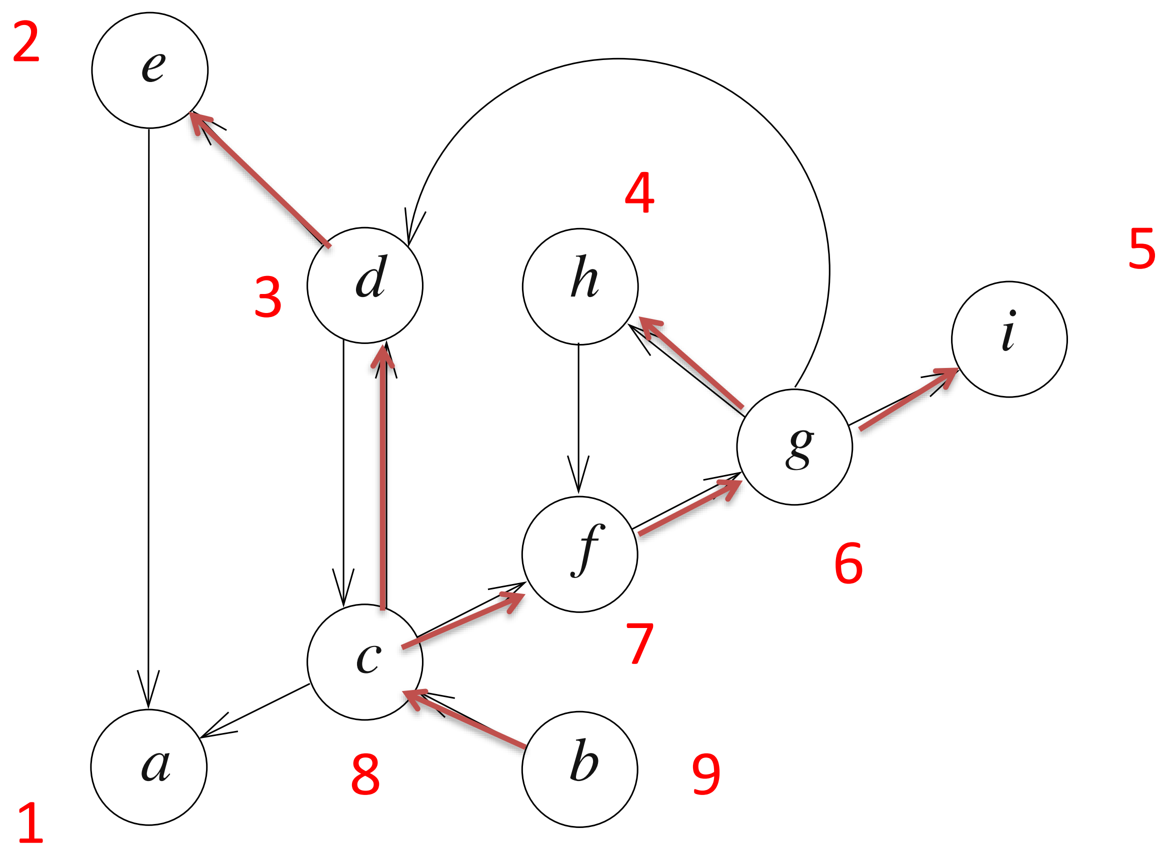
3. Use the numbering of step 1.) to run DFS on G^T. Start with the node that has the highest number. Whenever a tree is completed continue with the unvisited nodes that has the highest number.

4. The single trees computed in step 3) correspond to the node sets of the different strongly connected components.

Directed Graphs

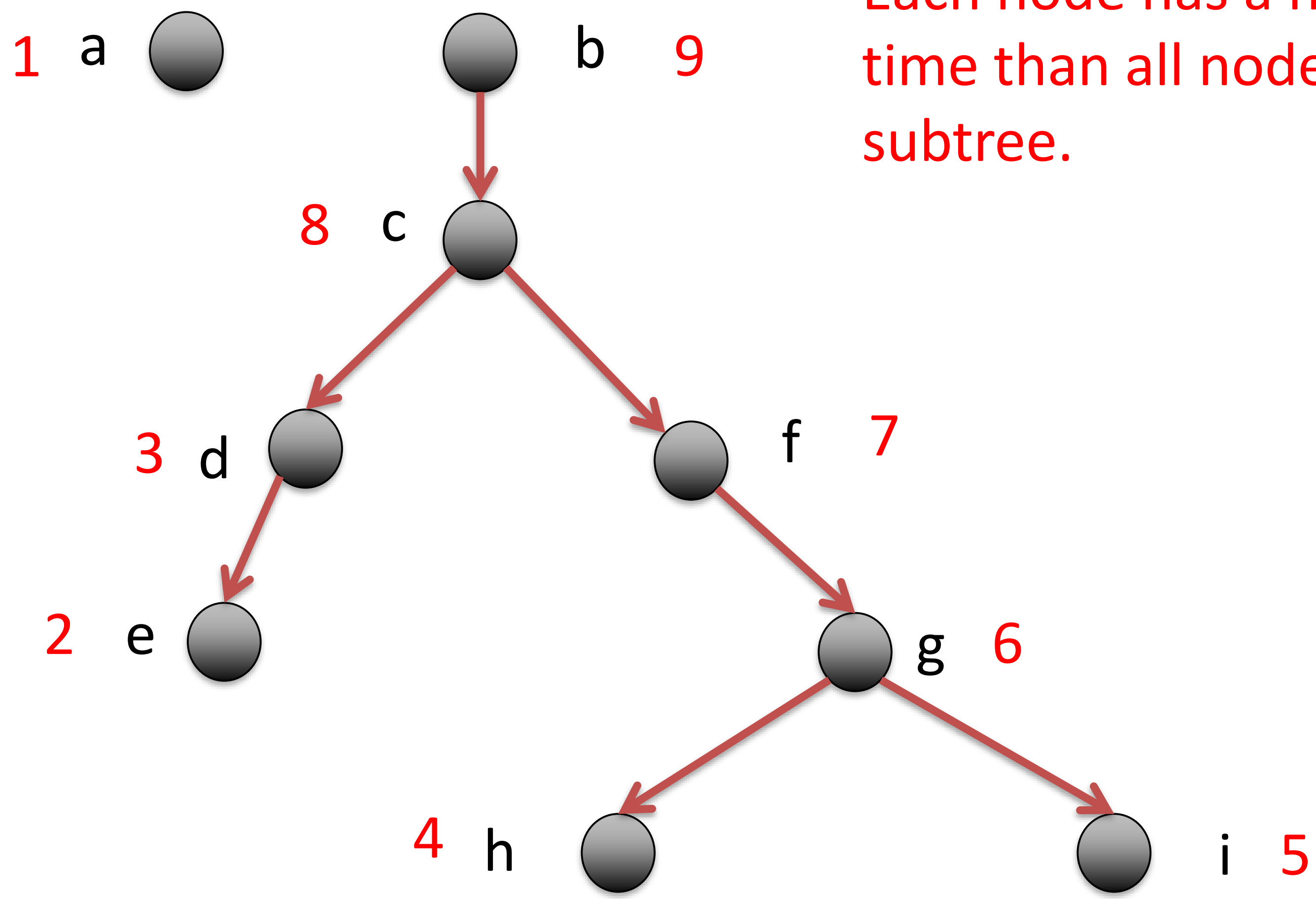


DFS-Tree and numbering



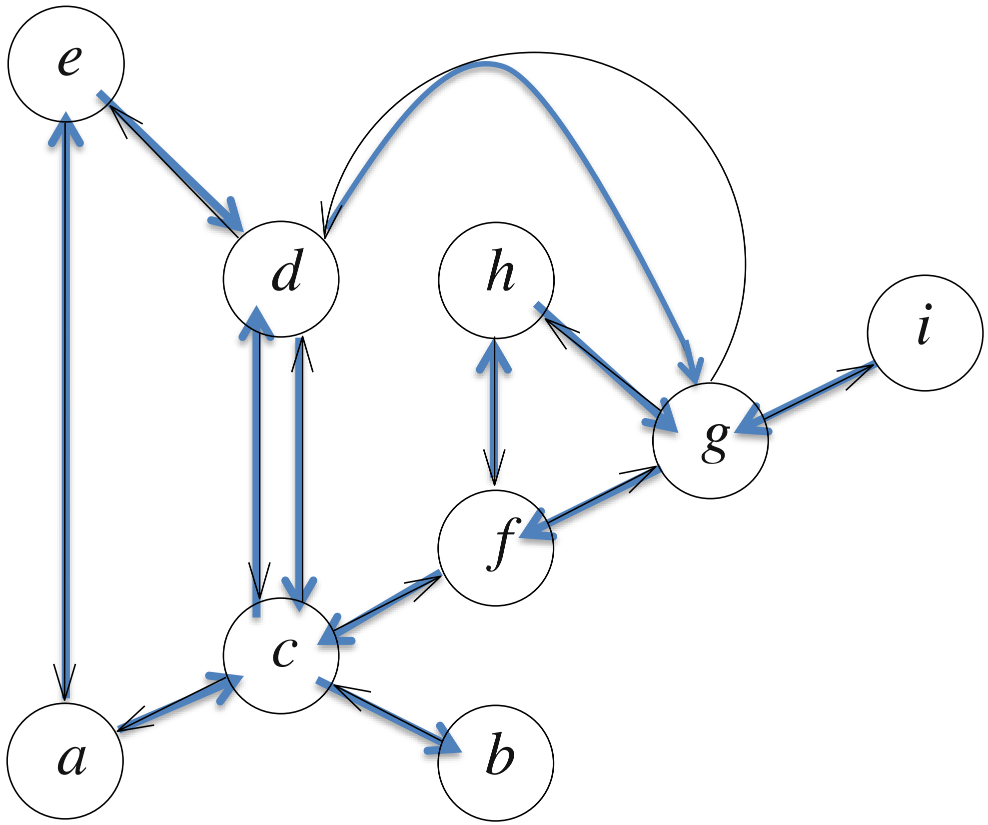
Nodes numbered according to termination of recursive calls

First run: DFS-Tree and finish times



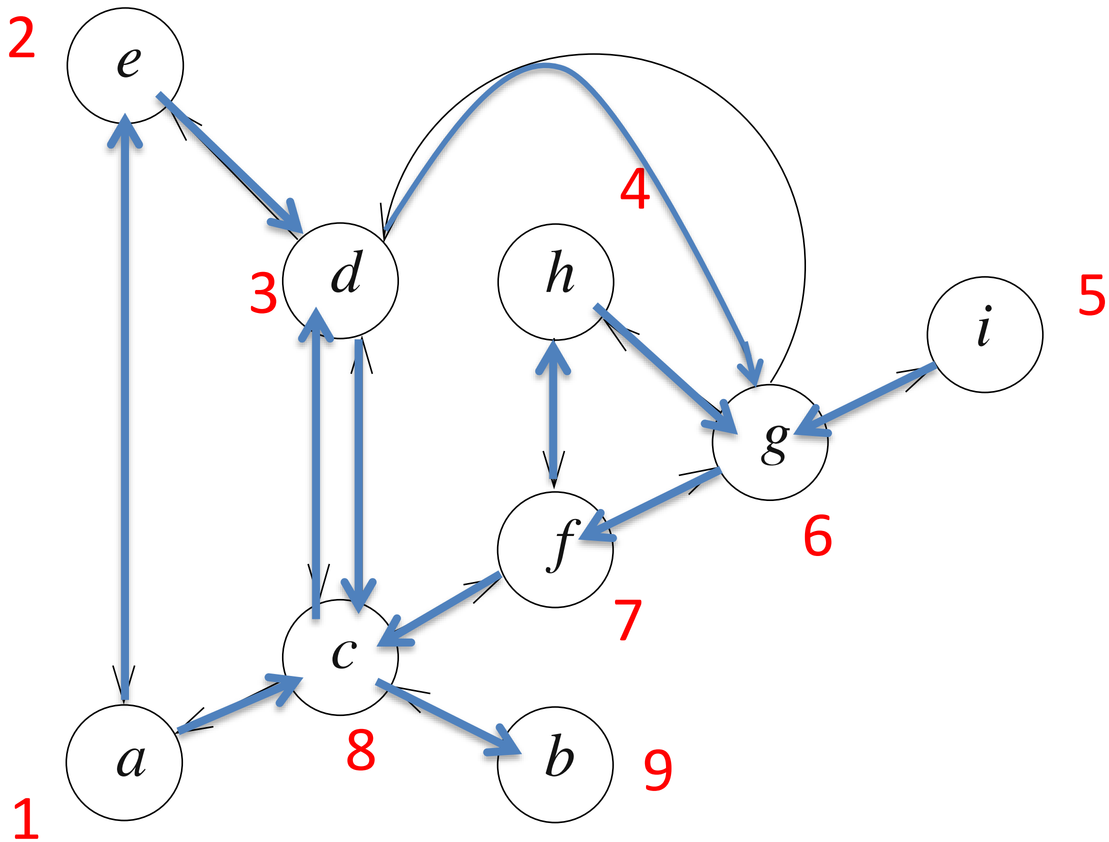
Each node has a higher finish time than all nodes in its subtree.

Directed Graphs

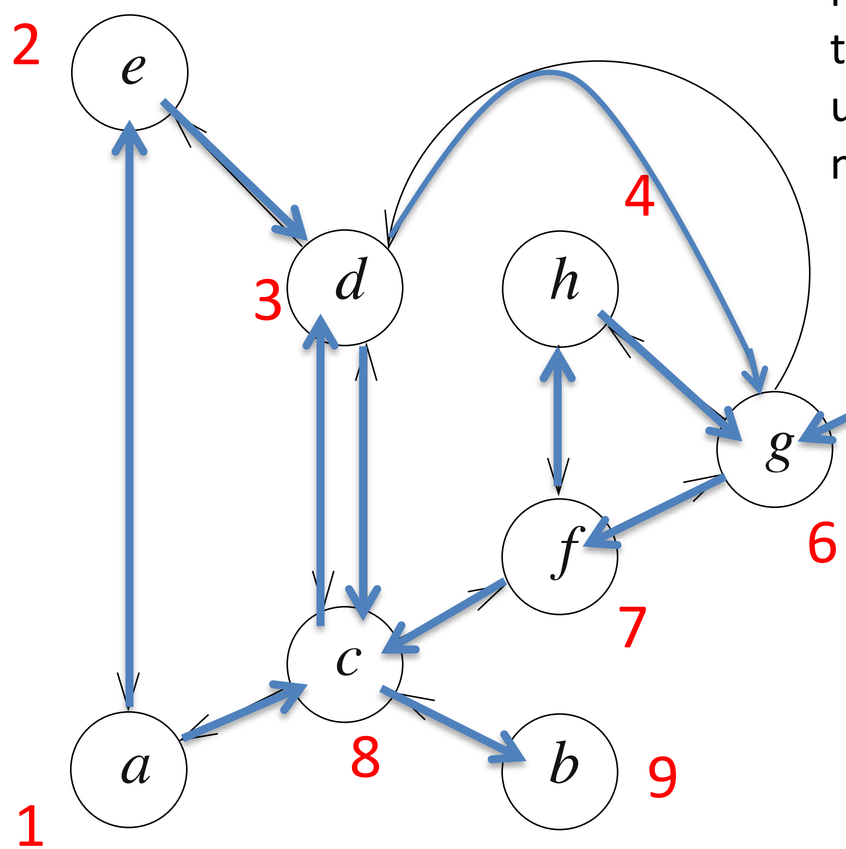


Transposed graph G^T

Directed Graphs



Use the numbering of step 1.) to run DFS on G^T. Start with the node that has the highest number. Whenever a tree is completed continue with the unvisited nodes that has the highest number.



Nodes of different trees can not belong to the same strongly connected component

1. strongly connected component: {b}
2. strongly connected component: {c,d,g,f,h}
3. strongly connected component: {i}
4. strongly connected component: {e}
5. strongly connected component: {a}

Correctness

Consider DFS-Trees obtained in the two runs.

First DFS-run:

- Each root of a (sub)-tree has higher finish time than its children.

Second DFS-run:

- Searches from each root r of a (sub)-tree for a backward path (traveling transposed edges) to its children.
- If a child v is reached then there is a path from v to the root r in that graph.
- This implies that r and v belong to the same strongly connected component.
- Second DFS run can only reach nodes with a smaller numbering.
- Traveling transposed edges implies that no node of another tree of the first DFS run is visited when starting at root r.

Runtime

- Use Adjacency Lists to represent the directed graph.
- We use DFS twice (time O(m+n))
- Have to compute the transpose graph

  (time O(m+n))
- Total runtime: O(m+n)