Algorithm and Data Structure Analysis (ADSA)

Graphs

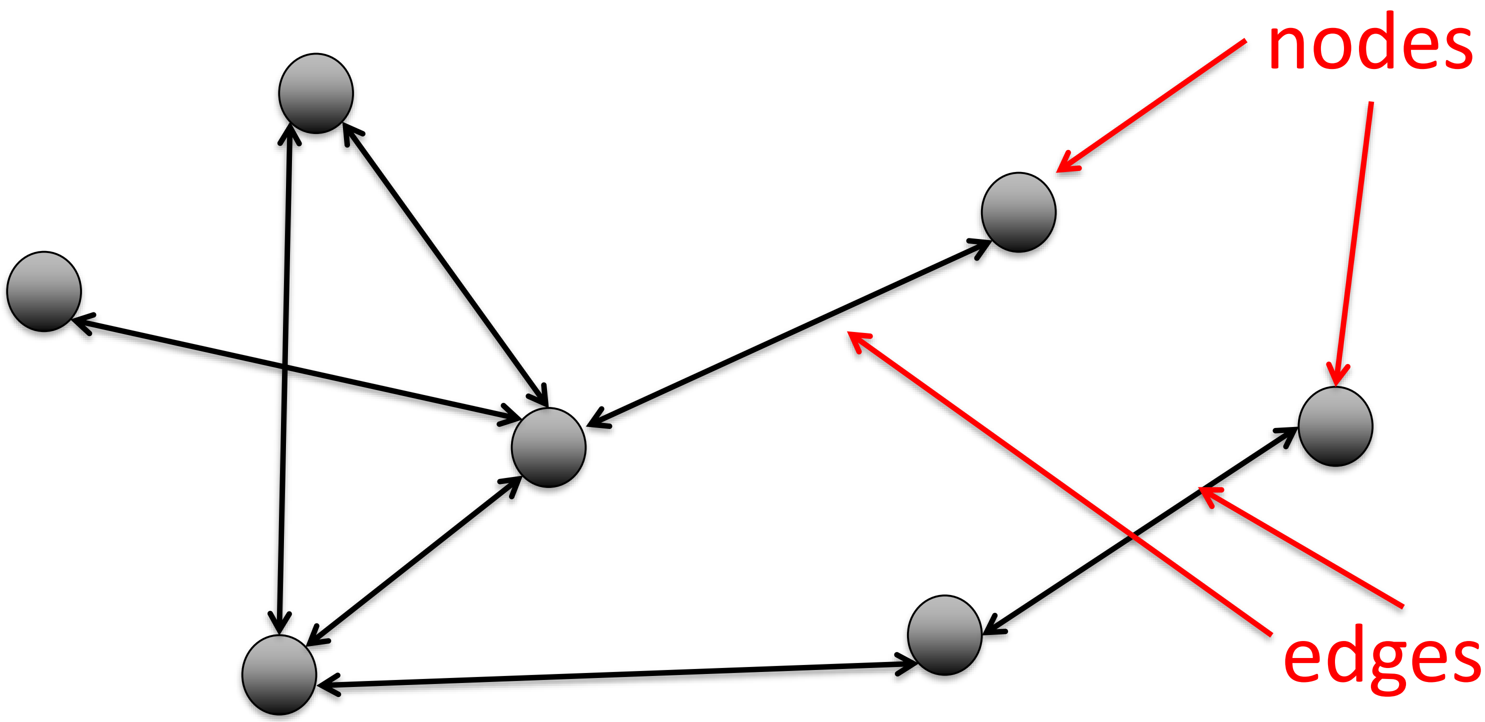
(Book Chapter 2)

Overview

- Graphs
- Basic Algorithms on Graphs

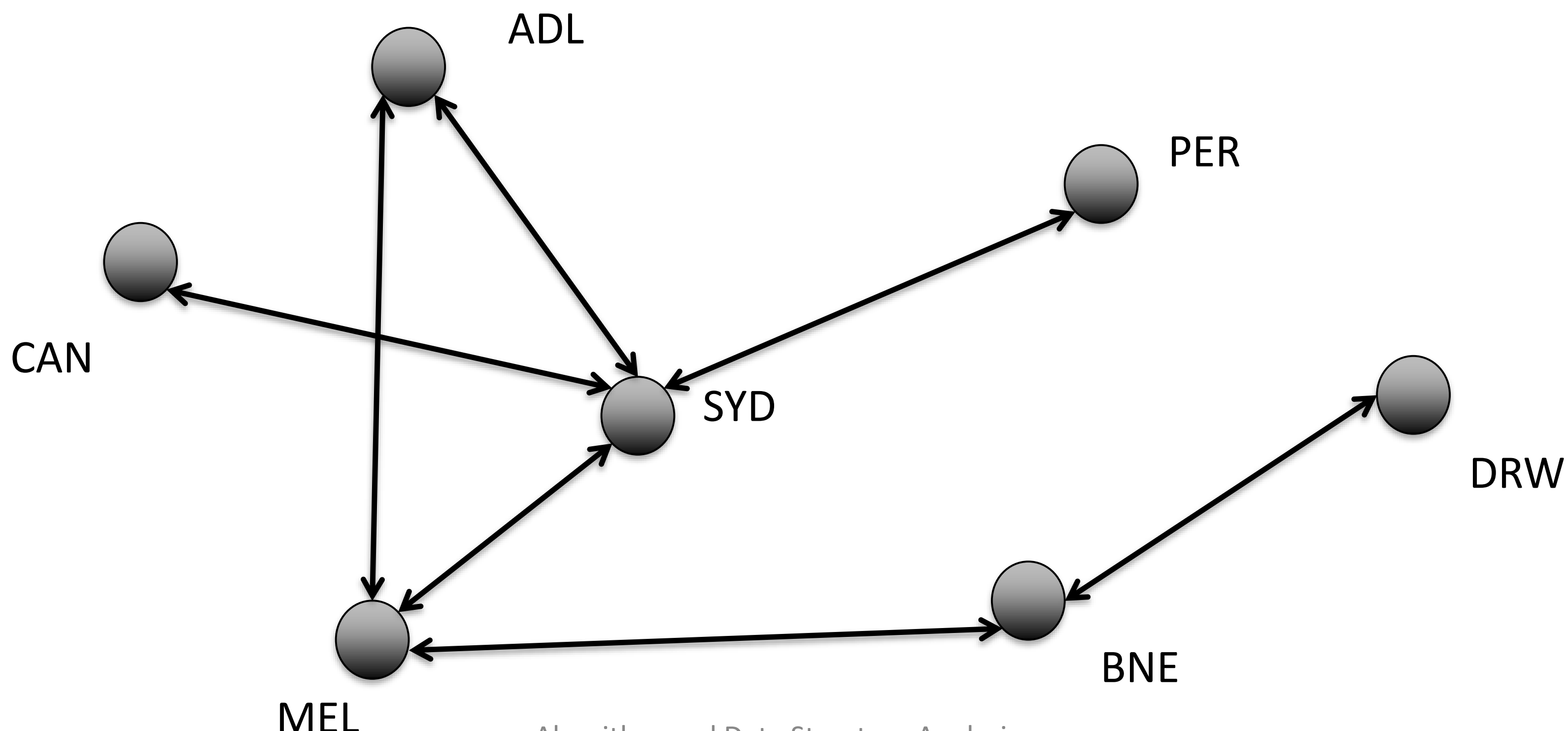
Graphs

- Extremely useful concept in computer science
- Can model many situations



Example

- Nodes are cities
- Edges are flight connections between them.



Mathematical Notation

A directed graph (digraph) G=(V,E) is a pair consisting of a node set (vertex set) V and an edge set (arc set) $E \subseteq V \times V$.

We denote by n =|V| the number of vertices and by m = |E| the number of edges.

Often there are edge weights/costs

$$c : E \rightarrow R$$

Terminology

- An edge e=(u,v) represents a connection from u to v.
- We call u the source and v the target.
- Edge e is incident to u and v.
- Nodes u and v are adjacent.
- Edge (v,v) is called a self-loop.

Terminology

- The number of outgoing edges of a vertex v is called the outdegree of v:

$$outdegree(v) = |\{(v,u) \in E\}|$$

- The number of incoming edges of a vertex v is called the indegree of v:

$$indegree(v) = |\{(u,v) \in E\}|$$

Bidirected graphs

- A bidirected graph G=(V,E) is a digraph where

$$(u,v) \in E \Longrightarrow (v,u) \in E$$

- An undirected graph can be viewed as streamlined representation of a bidirected graph.

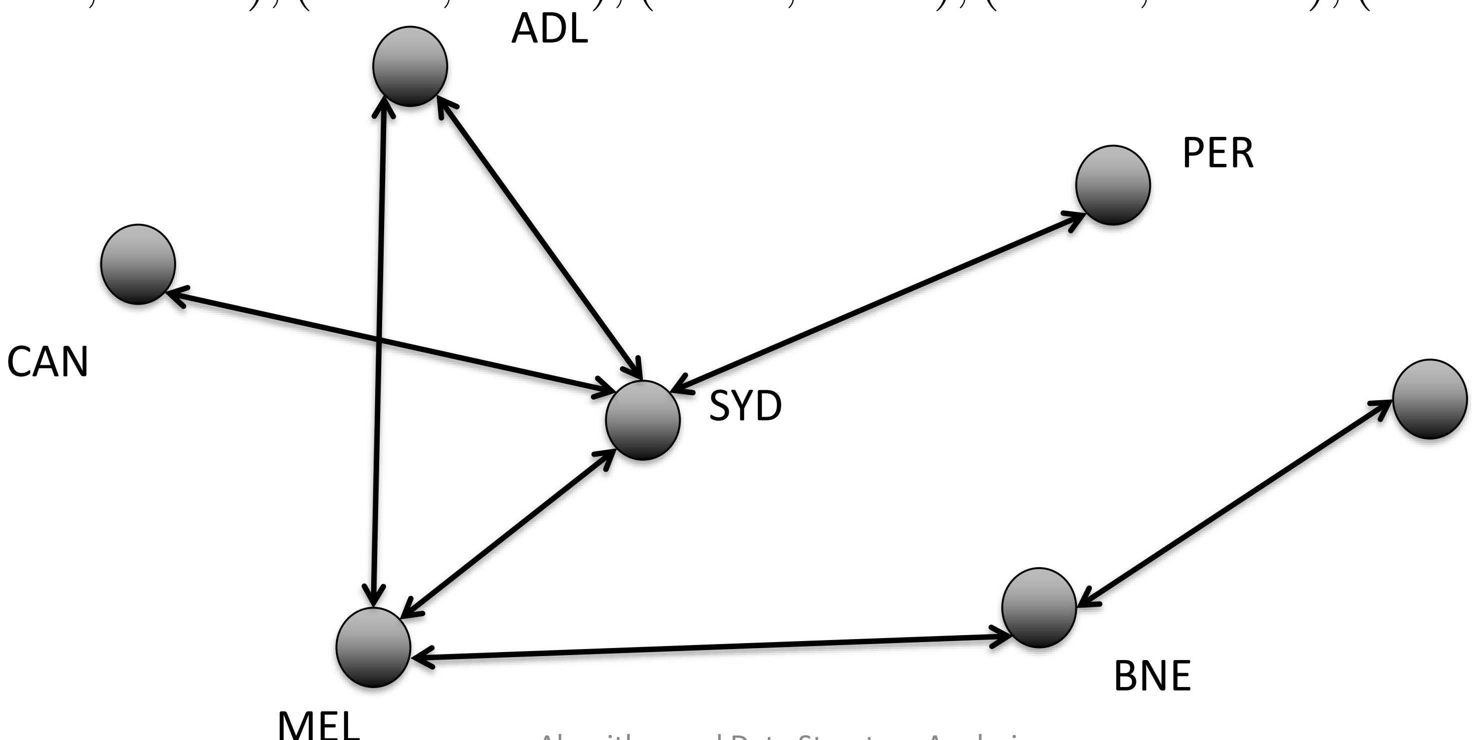
We write

$$(u,v) \quad \text{and} \quad (v,u)$$

as a two-element set $\{u,v\}$

Example: Bidirected graph

$$G_f = (V_f, E_f)$$
$$V_f = \{ADL, BNE, CAN, DRW, MEL, PER, SYD\}$$
$$E_f = \{(ADL, SYD), (SYD, ADL), (ADL, MEL), (MEL, ADL),$$
$$(SYD, CAN), (CAN, SYD), (MEL, SYD), (SYD, MEL), (MEL, BNE),$$
$$(BNE, MEL), (SYD, PER), (PER, SYD), (BNE, DRW), (DRW, BNE)\}$$

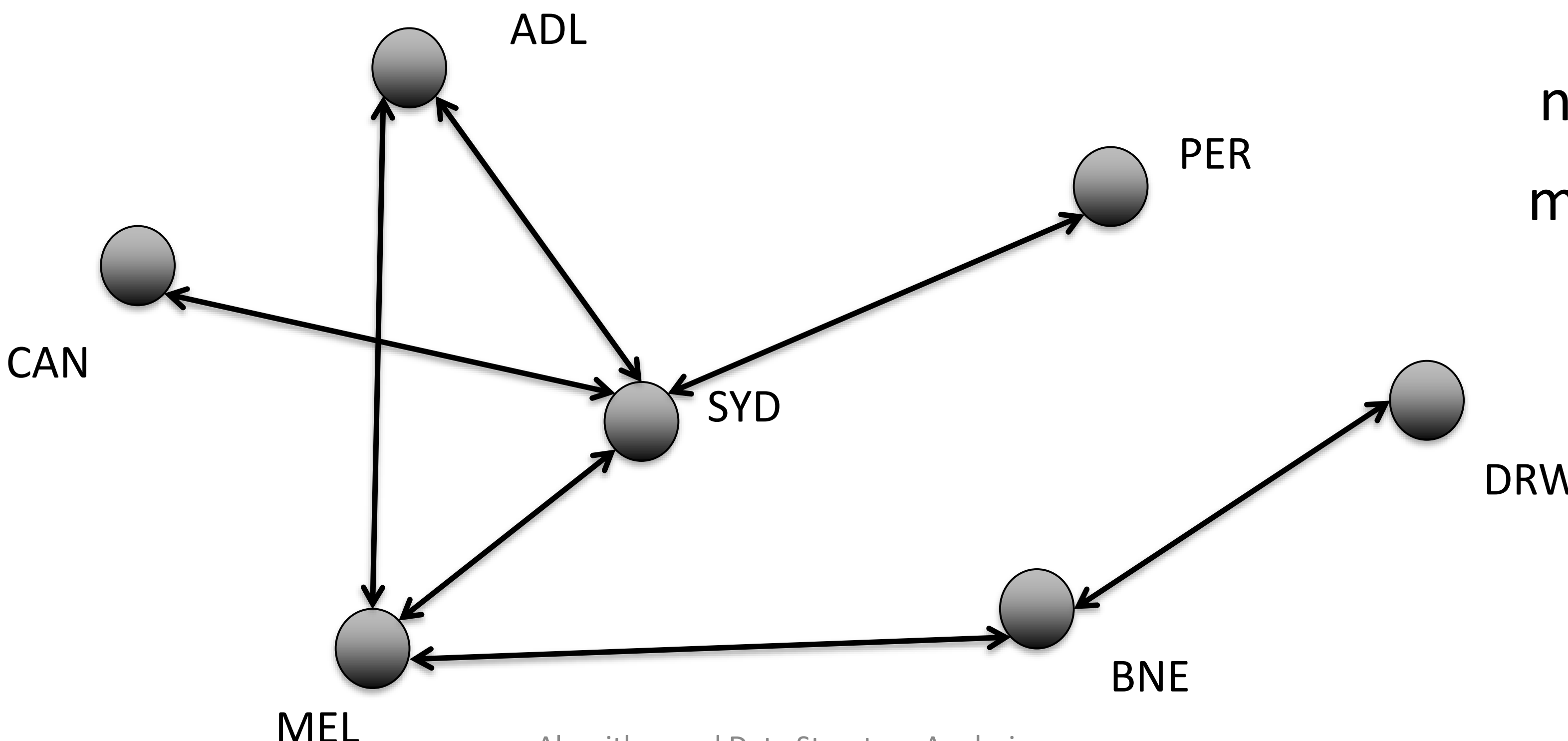


n=7

m=14

Example: Undirected graph

$$G_f = (V_f, E_f)$$
$$V_f = \{ADL, BNE, CAN, DRW, MEL, PER, SYD\}$$
$$E_f = \{\{ADL, SYD\}, \{ADL, MEL\}, \{SYD, CAN\},$$
$$\{MEL, SYD\}, \{MEL, BNE\}, \{SYD, PER\}, \{BNE, DRW\}\}$$



n=7

m=7

Subgraphs

- A graph G'=(V',E') is a subgraph of G=(V,E) if

$$V' \subseteq V \text{ and } E' \subseteq E.$$

- Given a graph G=(V,E) and a subset $V' \subseteq V$ the subgraph induced by V' is defined as

$$G' = (V', E \cap (V' \times V'))$$

Paths

- A path $p = (v_0, \ldots, v_k)$ is a sequence of nodes in which consecutive nodes are connected by an edge of E, i. e.
$$(v_i, v_{i+1}) \in E, 0 \leq i \leq k.$$
- Cycles are paths with a common first and last node, i. e. $v_0 = v_k$

Simple Graph Algorithm

- Given a directed graph G=(V,E).
- Is G acyclic?

Observation:

Node with outdegree zero can not appear in a cycle.

Idea for an algorithm:

- If there is a node v with outdegree zero, delete v (and the incoming edges) to obtain a graph G'
- G is acyclic if and only if G' is acyclic

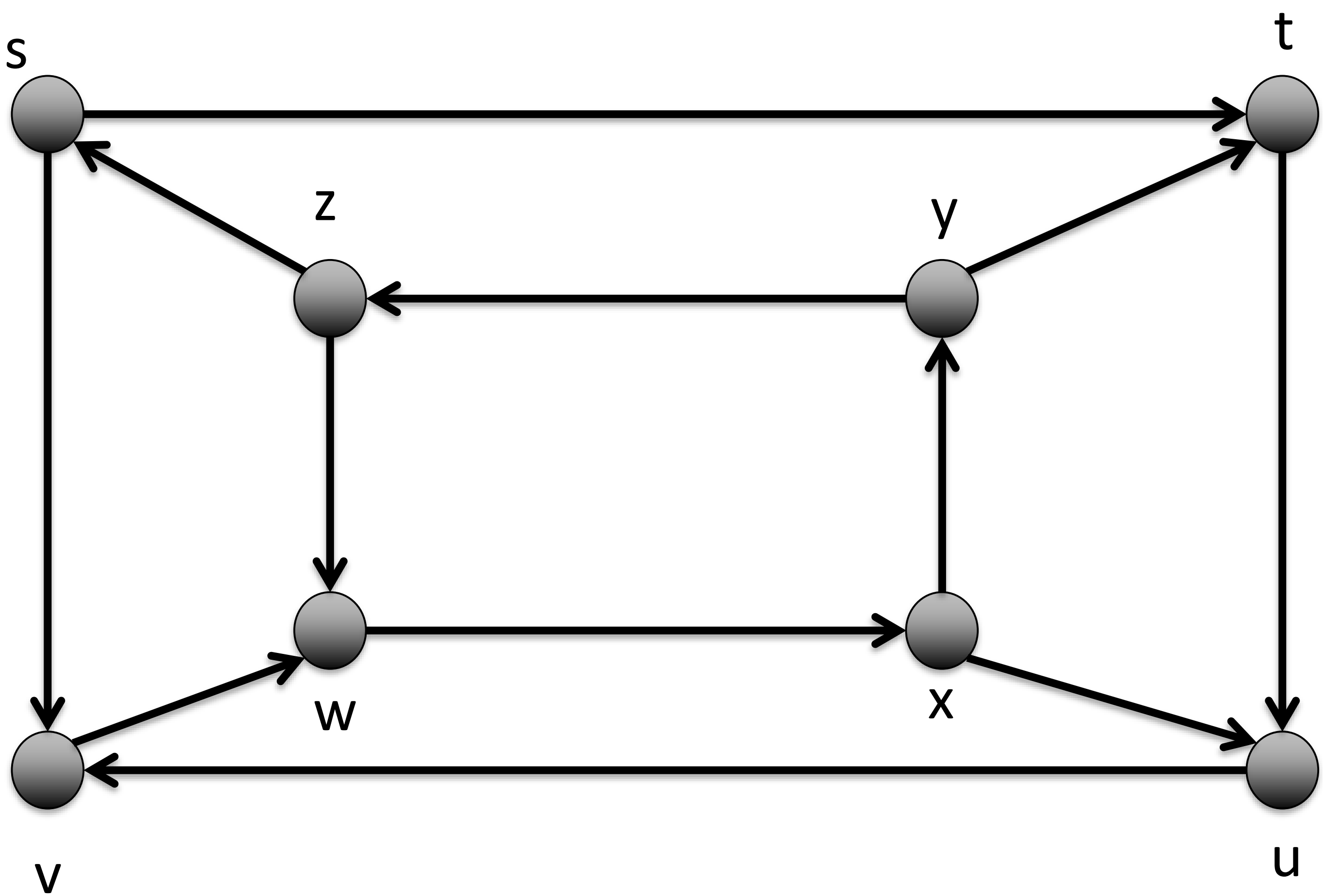
Algorithm

- If there is a node v of outdegree zero delete v and its incoming edges to obtain a graph G'.
- Iterate the transformation.

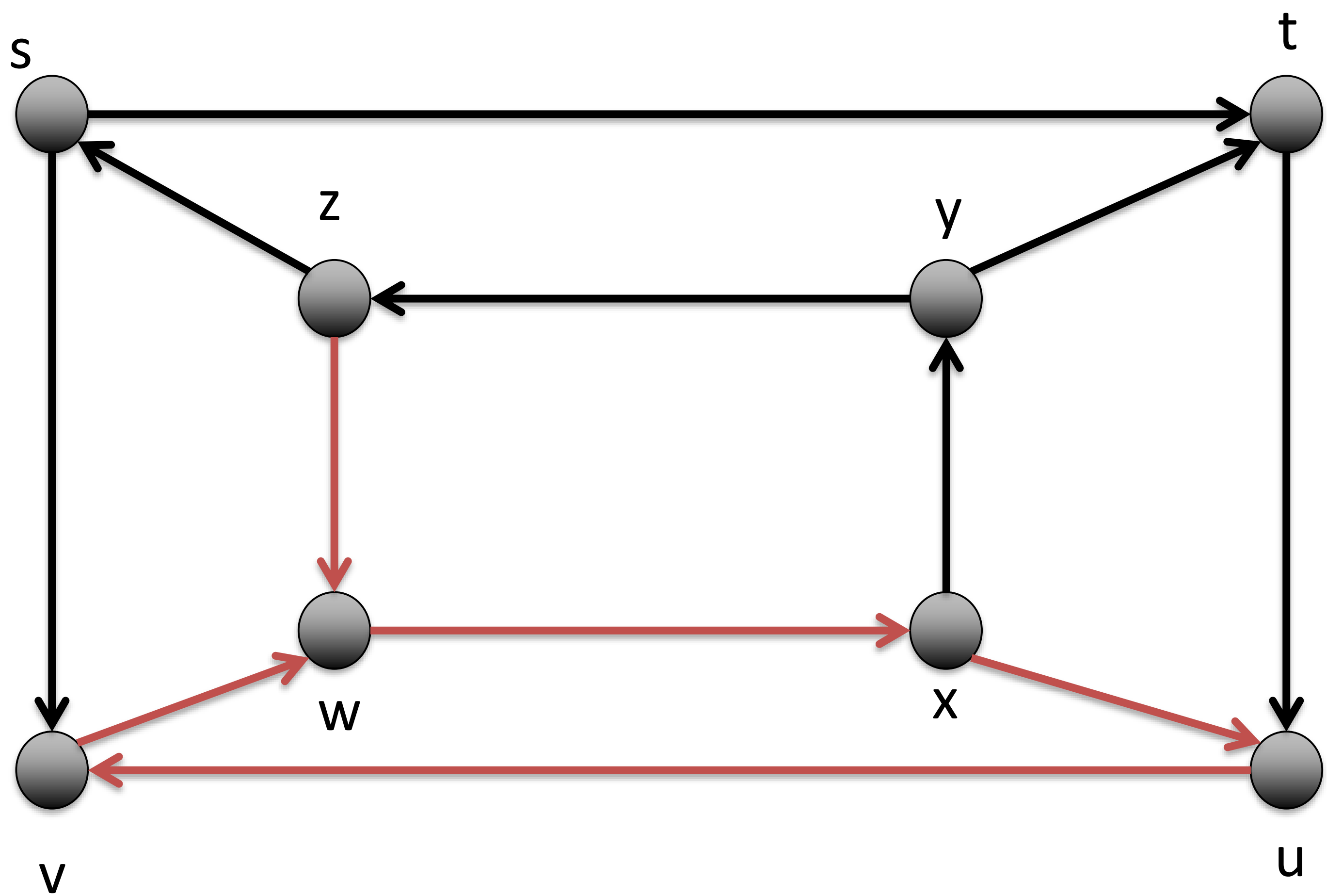
Arrive at a graph G*

- If G* is the empty graph then G is acyclic
- If G* is not the empty graph, we can find a cycle in G* that is also present in G.

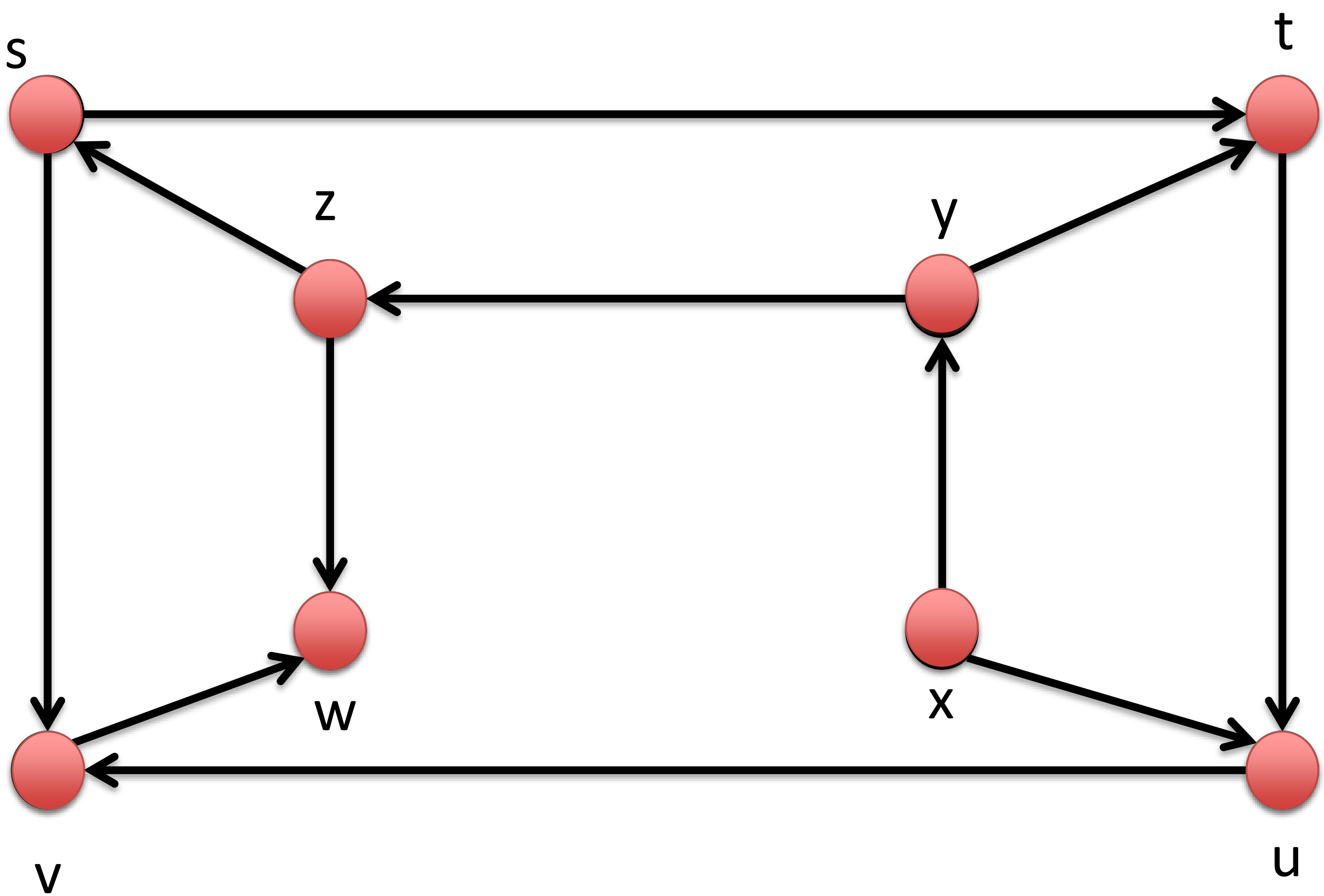
Graph containing a cycle



Graph containing a cycle

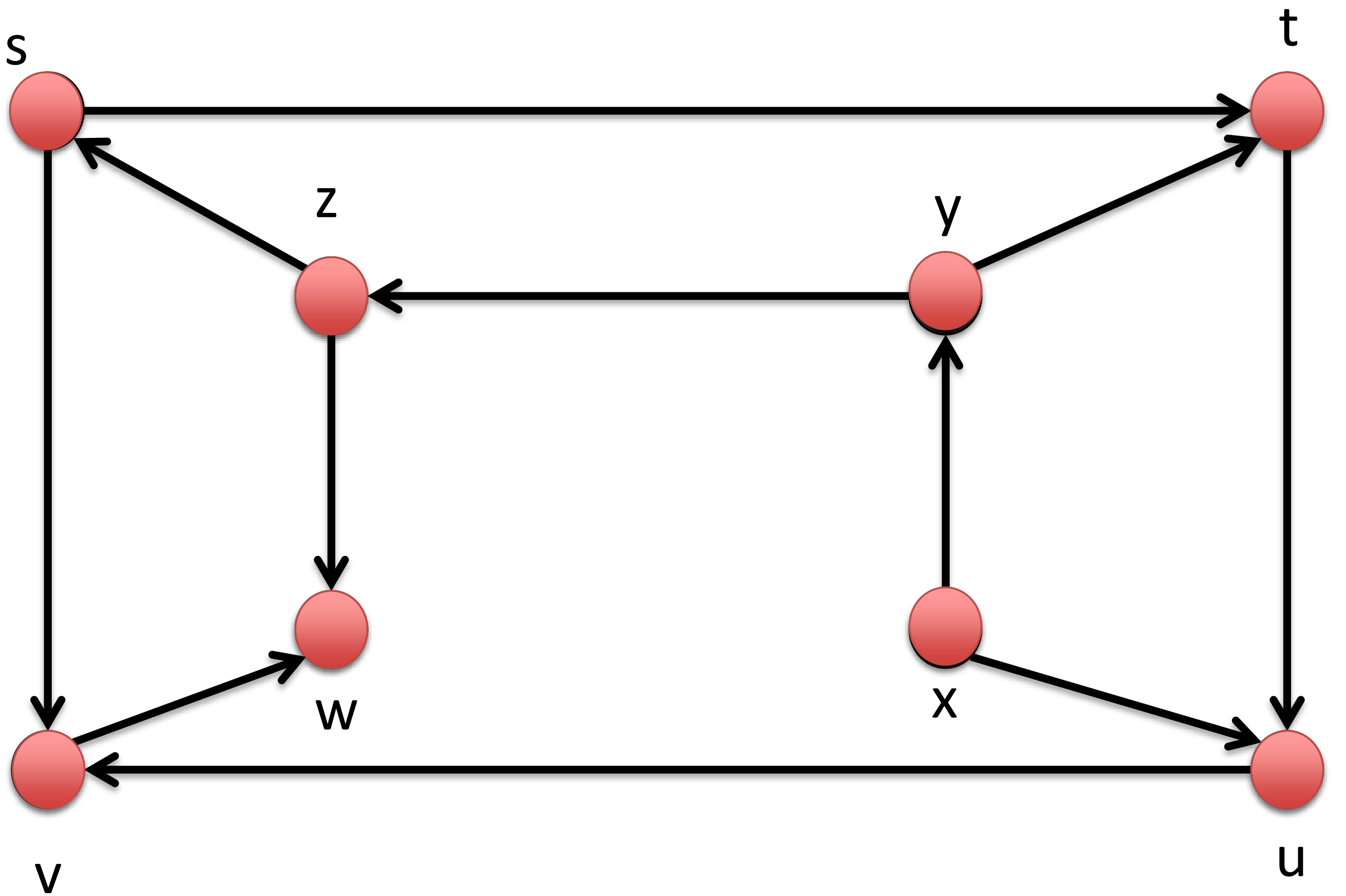


Acyclic Graph



Acyclic Graph

Empty Graph G* implies that G is acyclic



Trees and Forests

- An undirected graph is called a tree if there is exactly one path between any pair of nodes.
- An undirected graph is called a forest if there is at most one path between any pair of nodes.

Note: Each component of a forest is a tree.

Properties of Trees

The following properties of an undirected graph G are equivalent:

1. G is a tree.
2. G is connected and has exactly n-1 edges.
3. G is connected and contains no cycles.

Operations

We want efficiently support the following operations for graphs:

- **Accessing associated information** (get the information stored at nodes and edges)
- **Navigation** (access the edges incident to a node)
- **Edge queries** (ask whether an edge is in the graph, query its reverse edge)
- **Construction, conversion and output** (translate one graph representation into another)
- **Update** (Add and remove nodes and edges)

Unordered Edge Sequences

Simplest choice:

Unordered sequence of edges (e.g. linked list of edges).

Good if you just want to output the edges of the graph.

Problem:

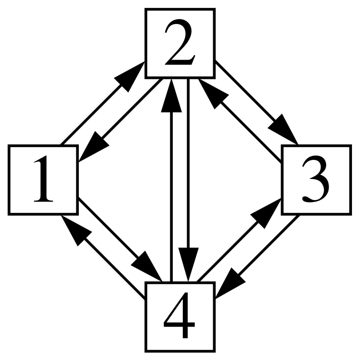
Most interesting operations take time $\Theta(m)$.

Adjacency Arrays (static graphs)

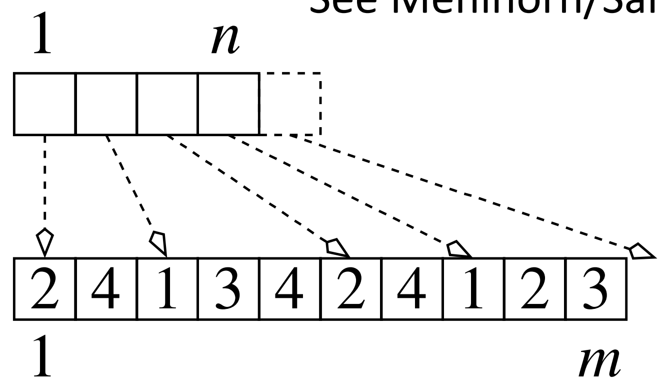
- Assume that the graph is static (i. e. it does not change).
- Then we can store the graph in an array.
- Store the outgoing neighbors of each node in a subarray and concatenate these subarrays into a single edge array E.
- Use an additional array V to store the starting positions of the subarrays.
- Memory consumption: $n+m+\Theta(1)$.

Adjacency Arrays

See Mehlhorn/Sanders (Fig. 8.1)



(Bi)-directed Graph



Adjacency Array

- For any node v, V[v] is the index of the first outgoing edge of v.
- Add dummy entry V[n+1]=m+1
- Outgoing edges of node v are accessible at

  E[V[v]], ..., E[V[v+1]-1]

Question

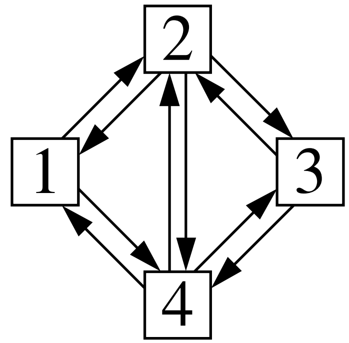
Are there better representations that allow to add or remove edges in constant time?

Two popular choices:

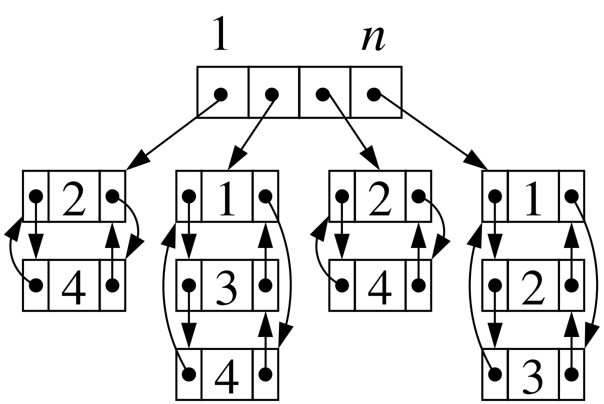
Adjacency Lists

Adjacency Matrices

Adjacency Lists



(Bi)-directed Graph



$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Adjacency List

Adjacency Matrix

 See Mehlhorn/Sanders (Fig. 8.1)

Adjacency Lists

Idea: Use for each node v a double-linked list that stores its outgoing neighbors (alternatively we can also use the incoming neighbors or lists for both).

Advantage:

- Insertion of edges goes in constant time.
- Well suited for sparse graphs (occur often in practice)

Adjacency Matrices

Idea: Represent a graph consisting of n nodes by an n×n matrix A. Set

$$A_{ij} = 1 \text{ if } (i,j) \in E$$
$$A_{ij} = 0 \text{ otherwise}$$

Insertion, removal, edge queries work in constant time.

O(n) to obtain an edge entering or leaving a node.

Disadvantage: Storage requirement n^2 even for sparse graphs.

Graph Traversal

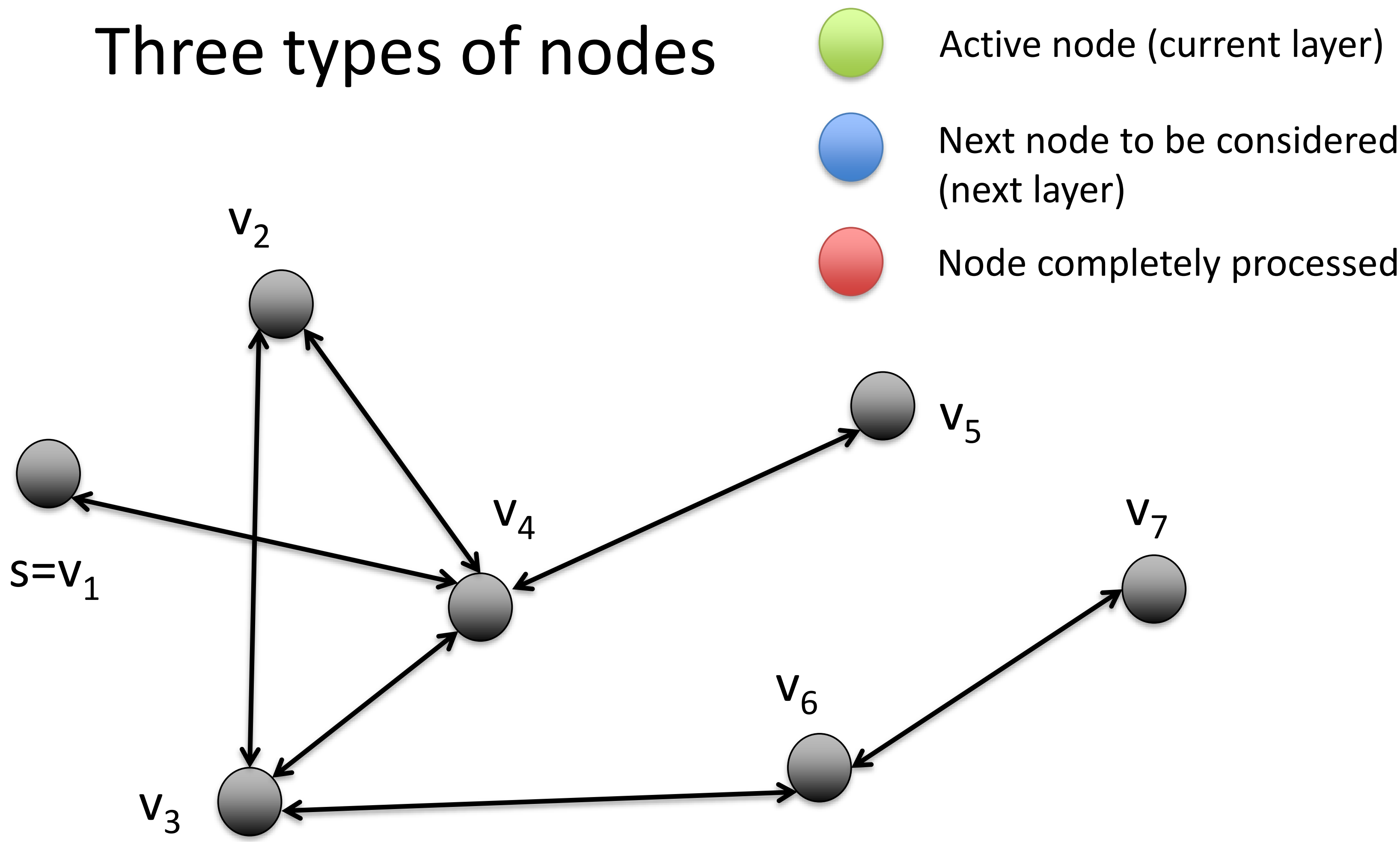
We want to have algorithms that visit very node of a given graph in linear time.

Idea for breadth-first-search: Start at a node s and visit in iteration i all nodes of distance i to s.

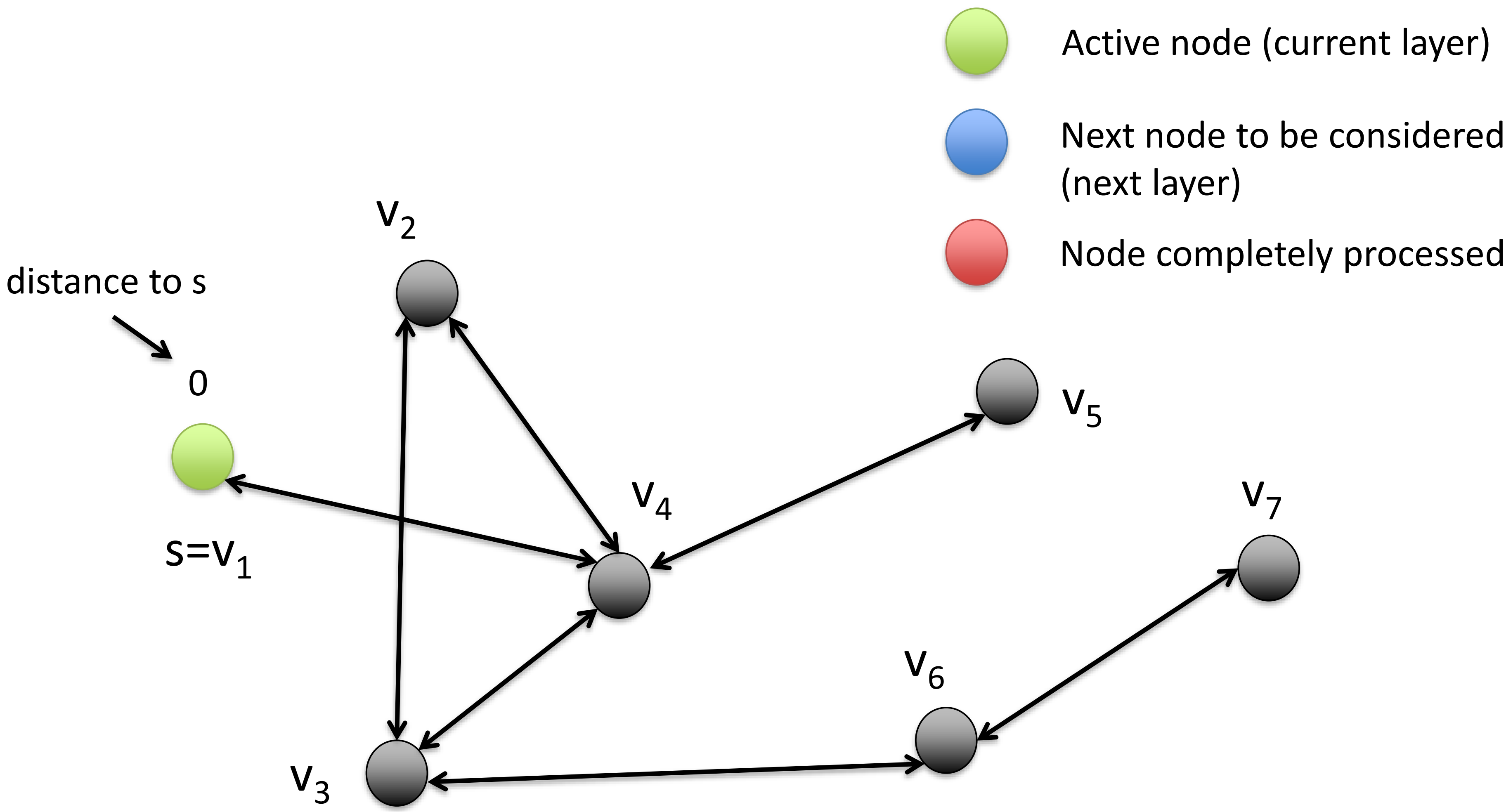
 See Mehlhorn/Sanders, Fig. 8.1)

Breadth-first-search (BFS)

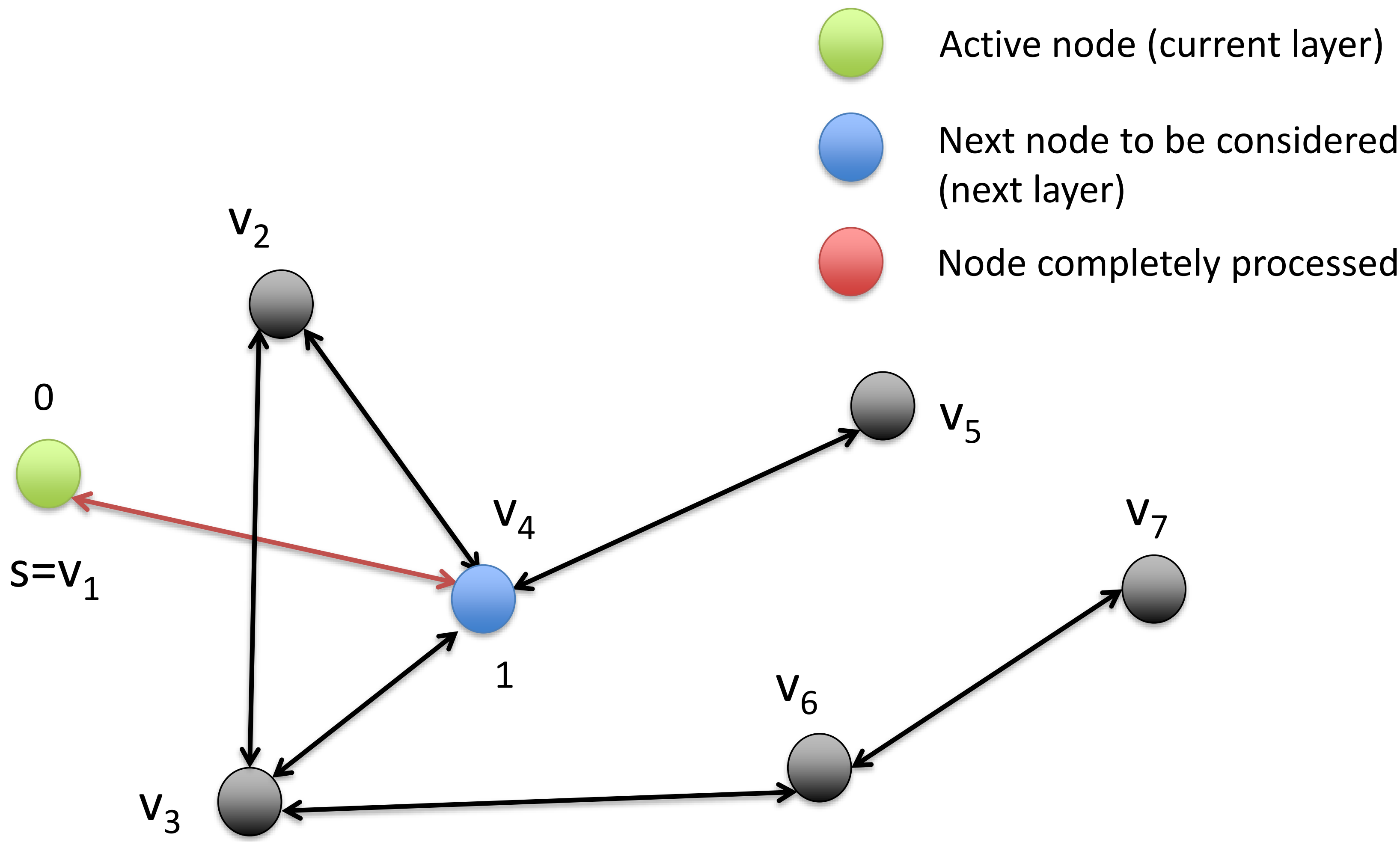
Three types of nodes



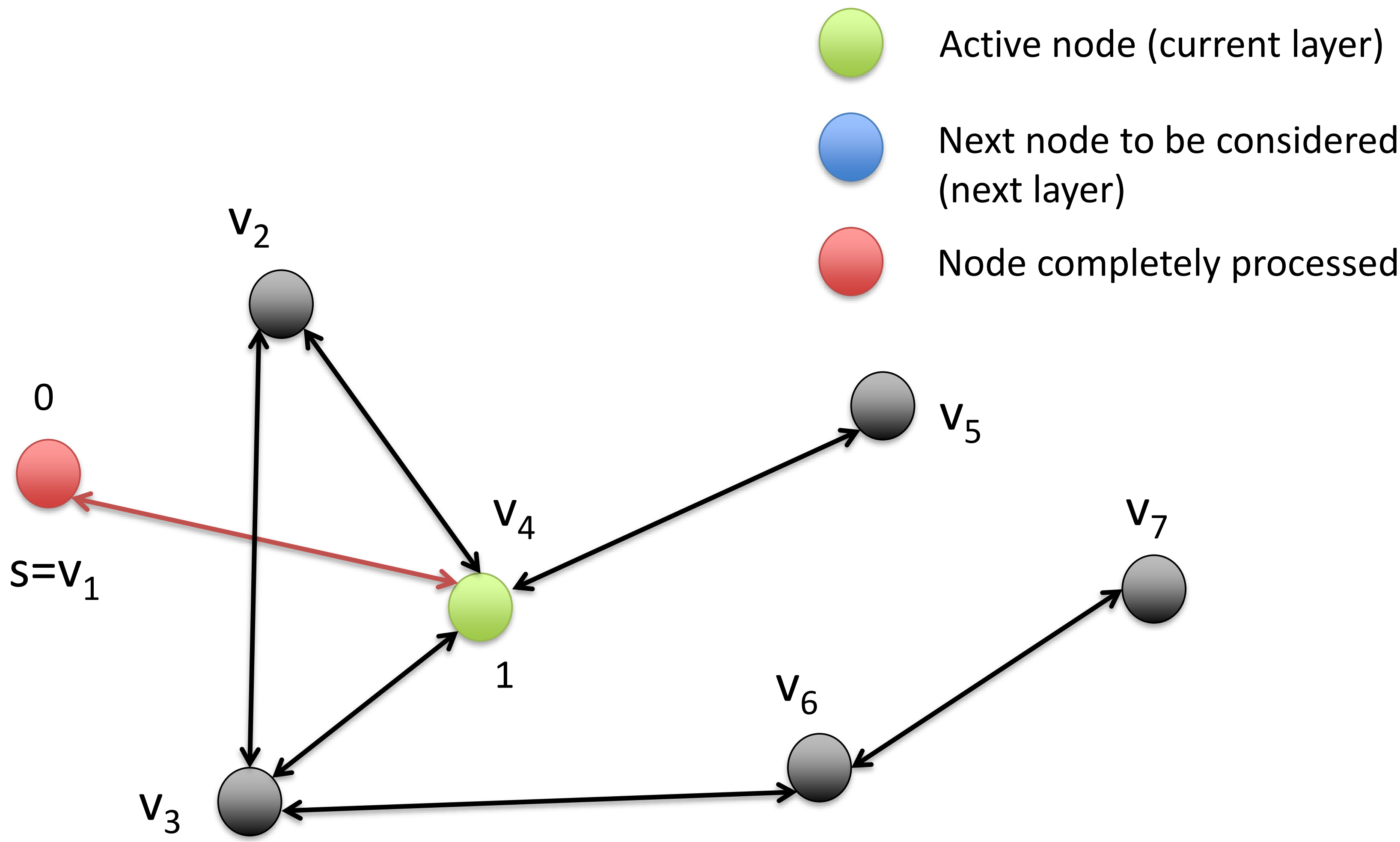
Breadth-first-search (BFS)



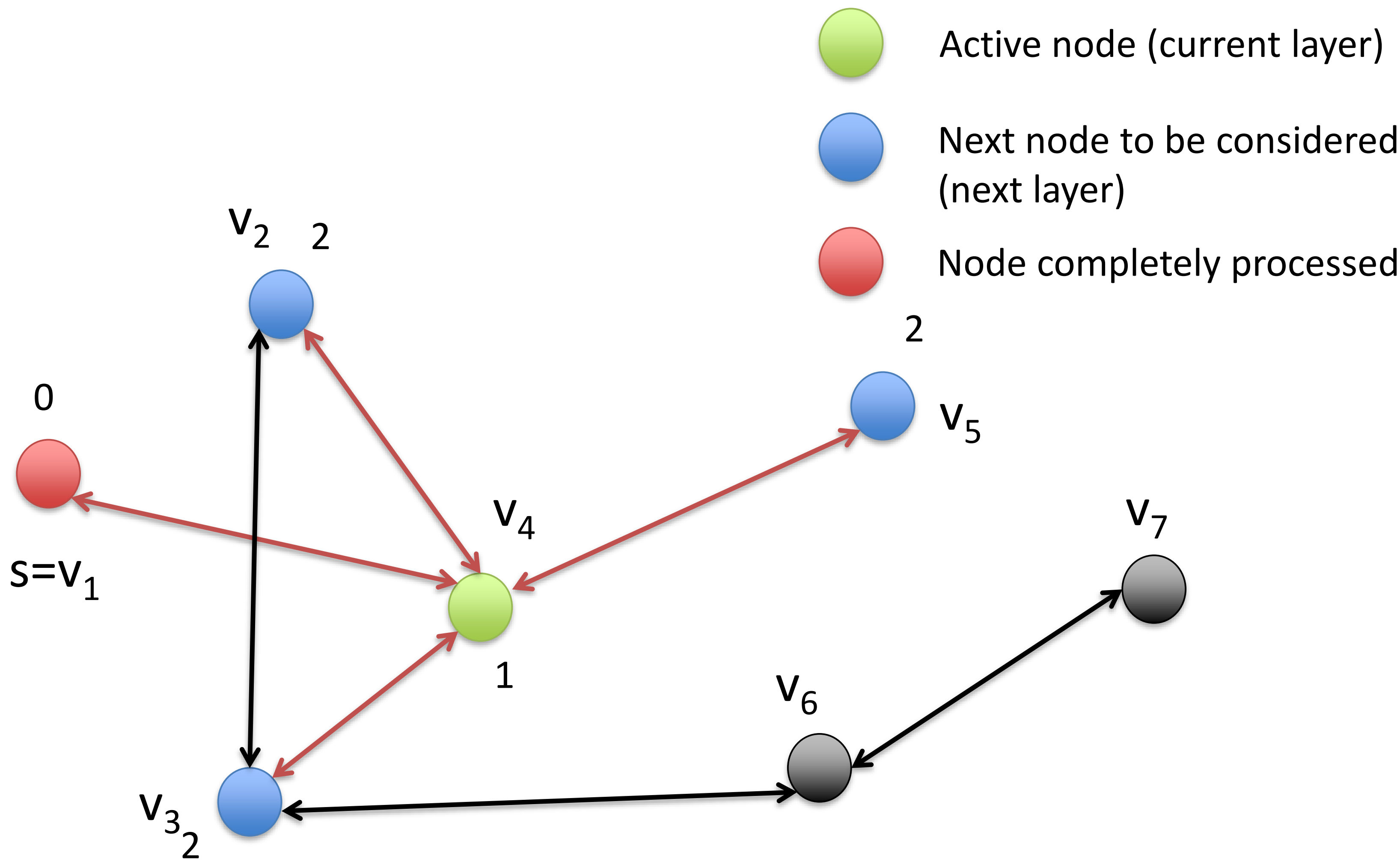
Breadth-first-search (BFS)



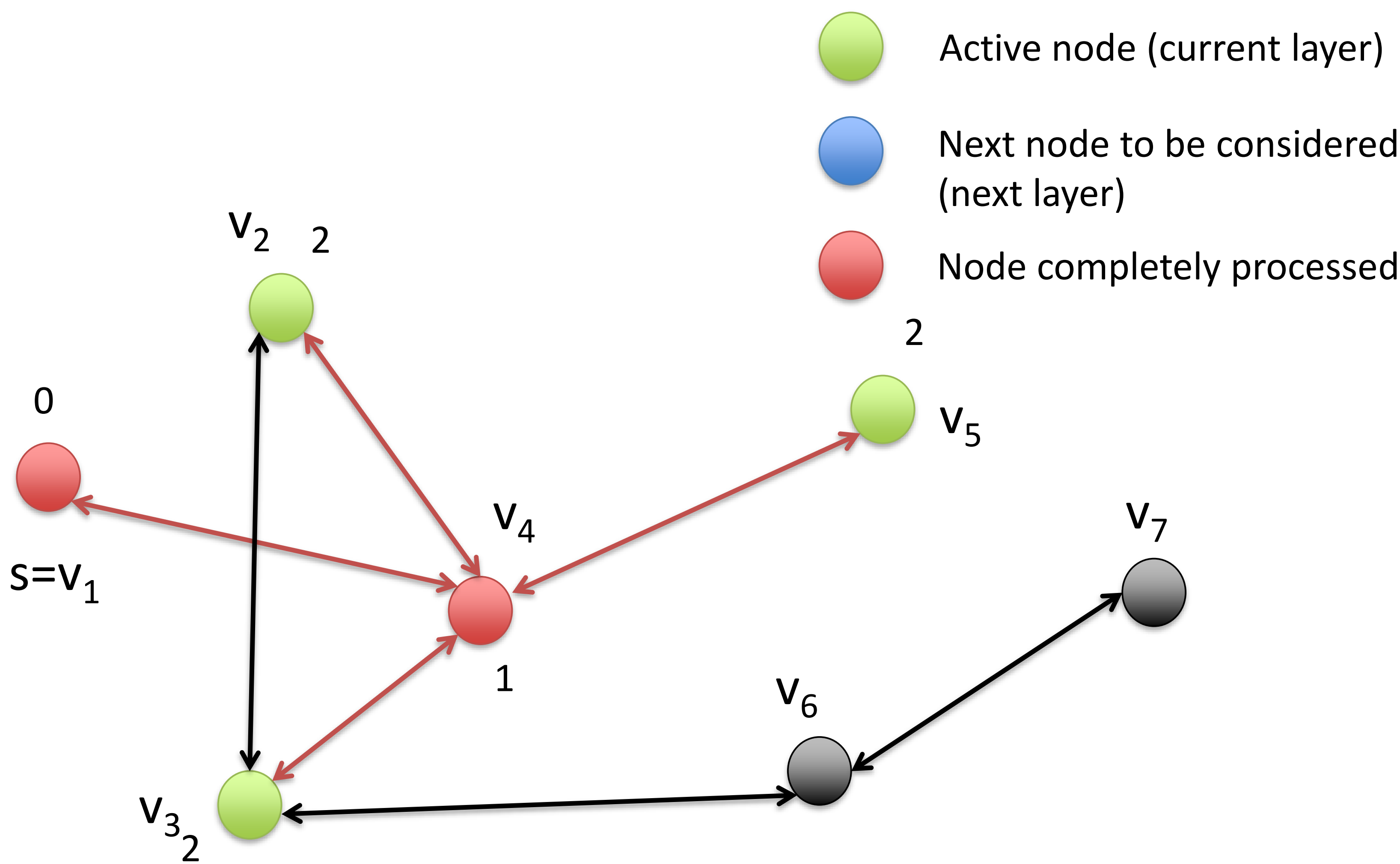
Breadth-first-search (BFS)



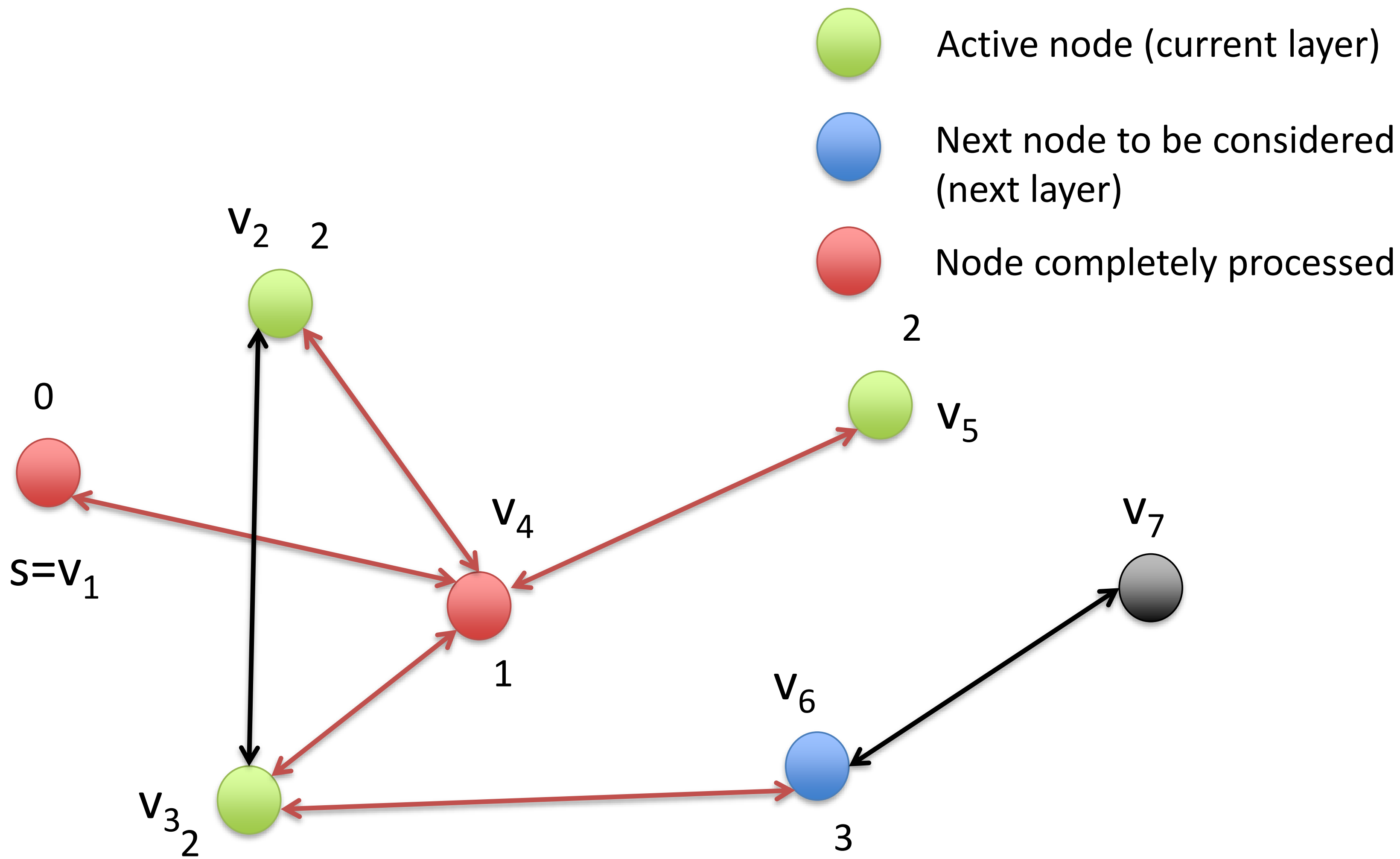
Breadth-first-search (BFS)



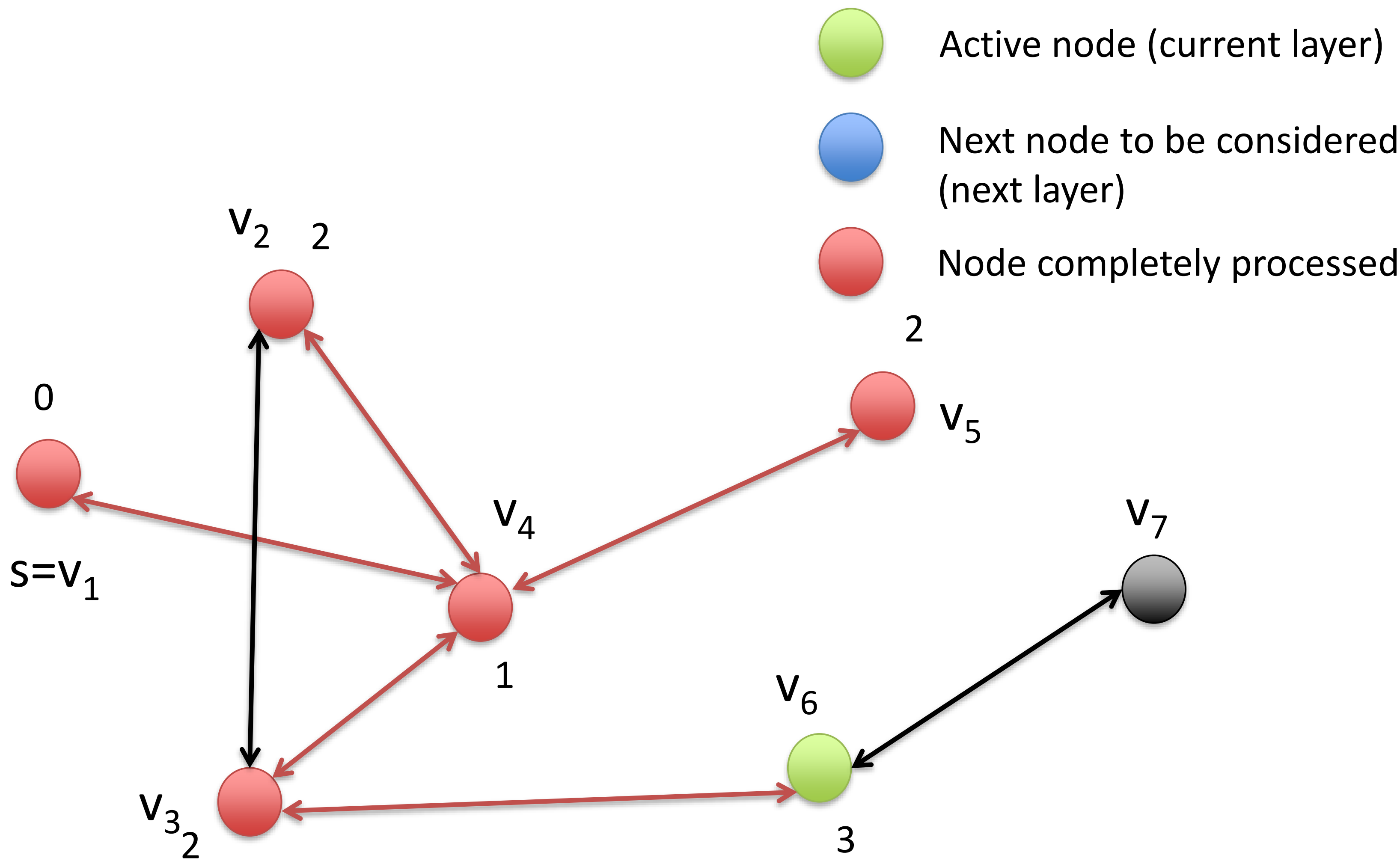
Breadth-first-search (BFS)



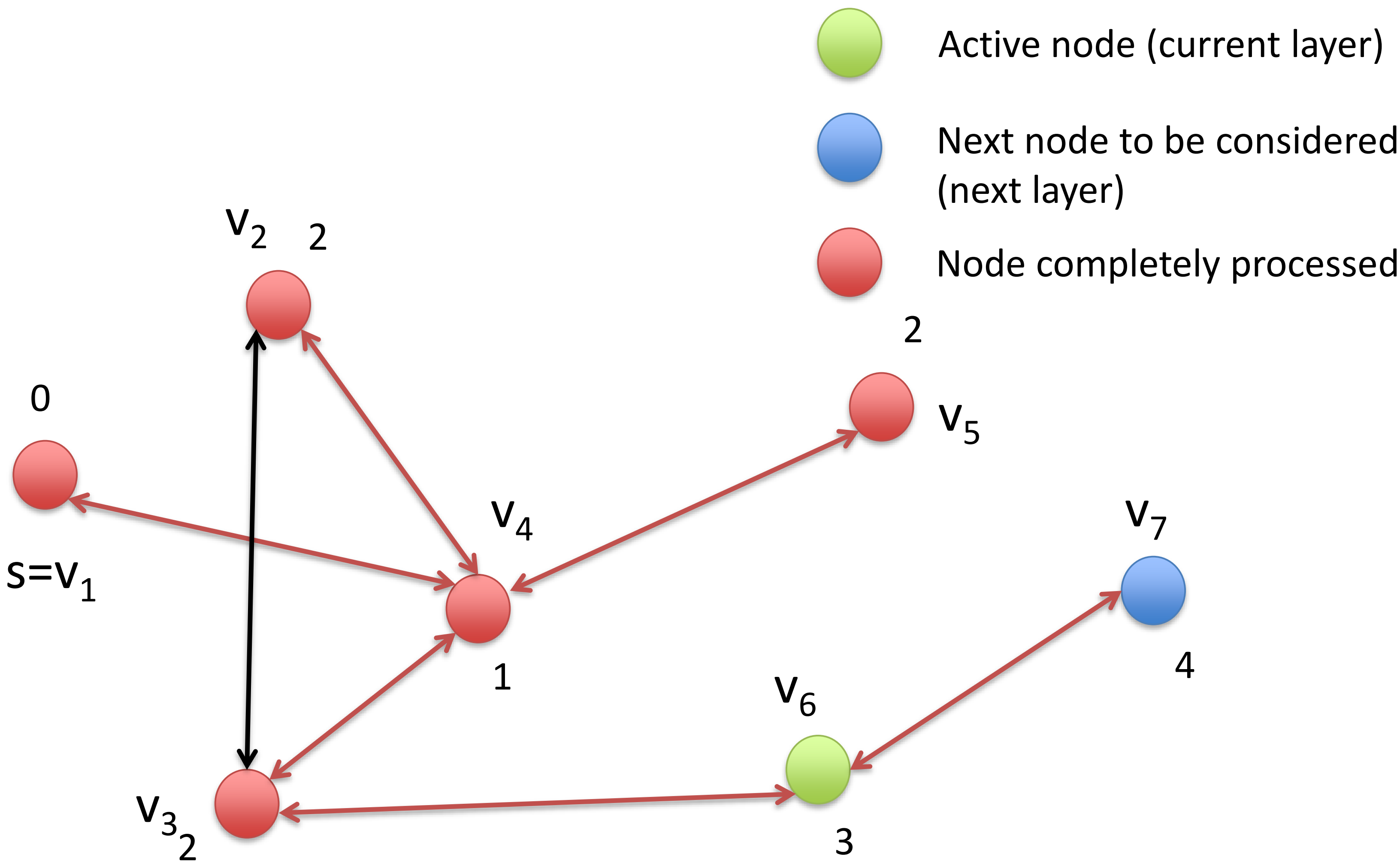
Breadth-first-search (BFS)



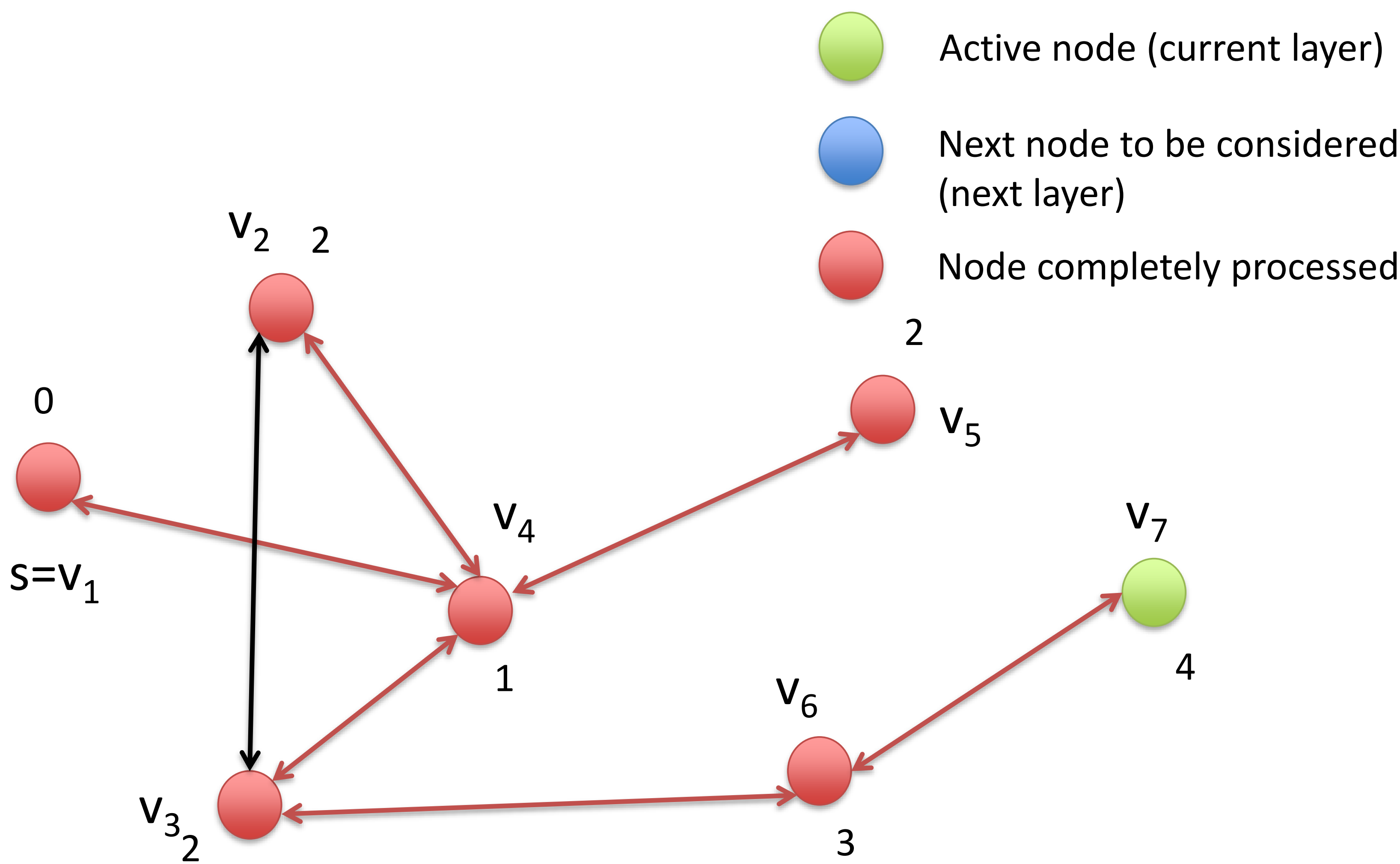
Breadth-first-search (BFS)



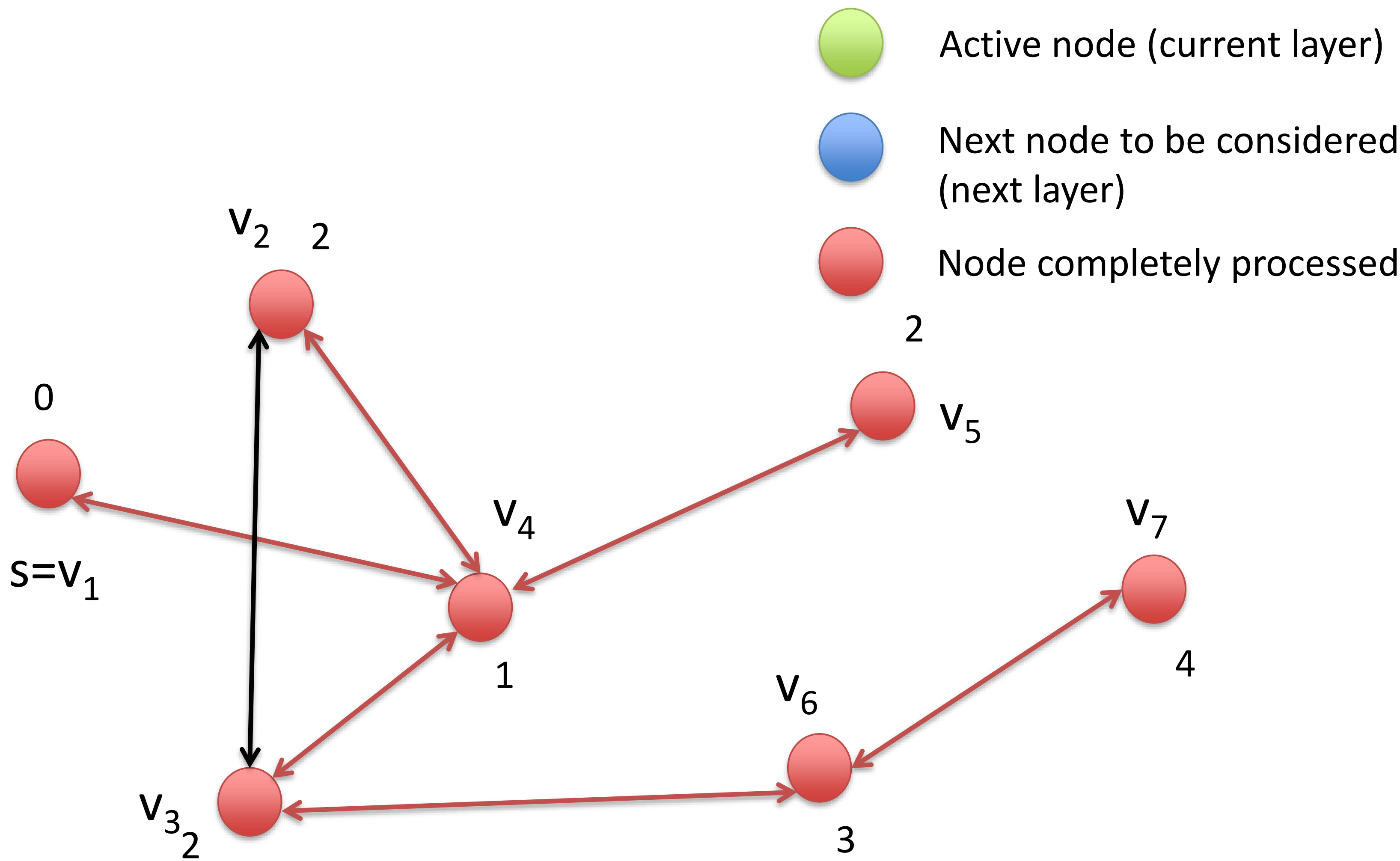
Breadth-first-search (BFS)



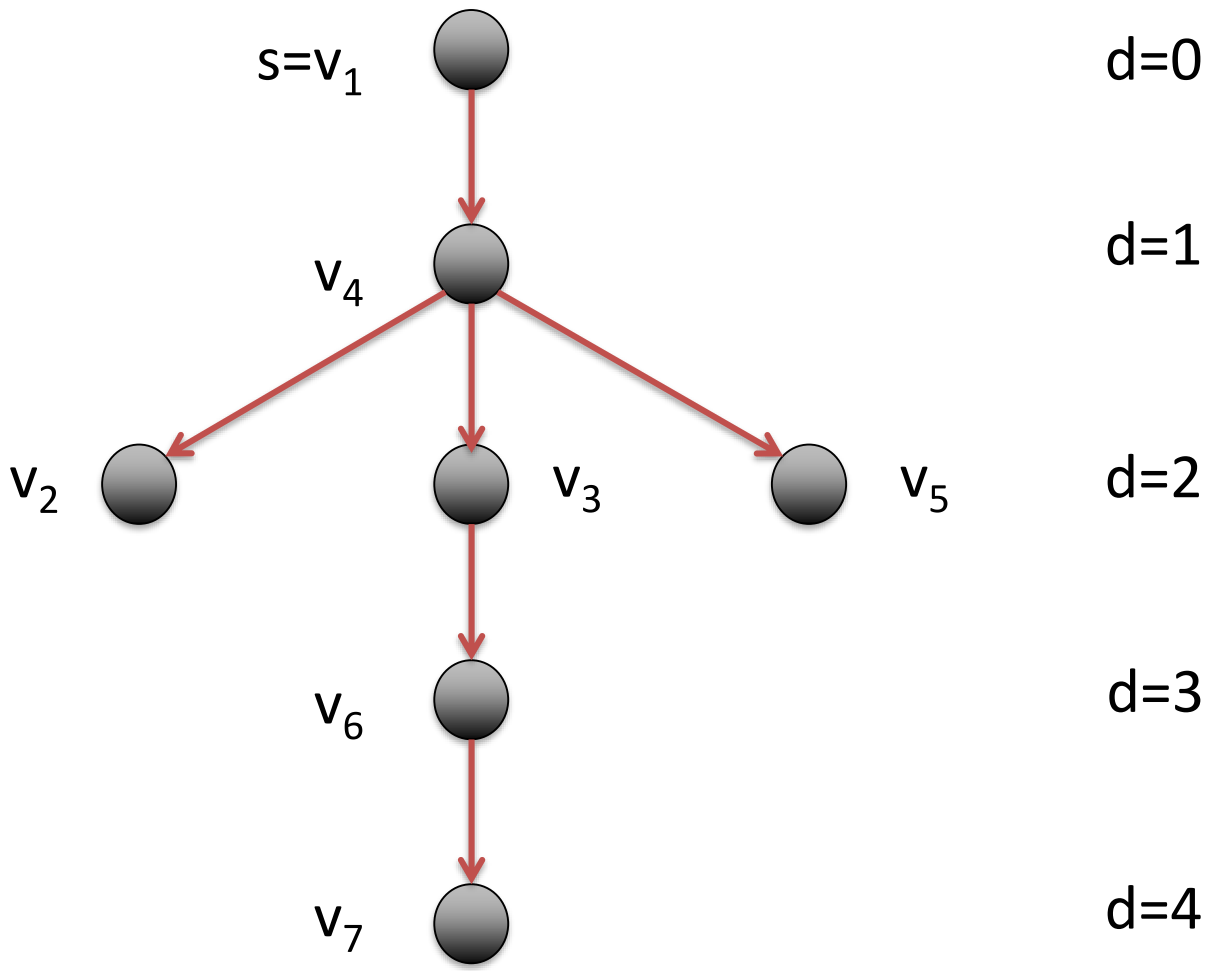
Breadth-first-search (BFS)



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

Function $bfs(s : NodeId) : (NodeArray \textbf{ of } NodeId) \times (NodeArray \textbf{ of } 0..n)$

$d = \langle\infty,\ldots,\infty\rangle \; : NodeArray \textbf{ of } NodeId$ // distance from root

$parent = \langle\bot,\ldots,\bot\rangle \; : NodeArray \textbf{ of } NodeId$

$d[s] := 0$

$parent[s] := s$ // self-loop signals root

$Q = \langle s\rangle \; : Set \textbf{ of } NodeId$ // current layer of BFS tree

$Q' = \langle\rangle \; : Set \textbf{ of } NodeId$ // next layer of BFS tree

for $\ell := 0$ **to** ∞ **while** $Q \neq \langle\rangle$ **do** // explore layer by layer

  invariant Q contains all nodes with distance ℓ from s

  foreach $u \in Q$ **do**

    foreach $(u,v) \in E$ **do** // *scan* edges out of u

      if $parent(v) = \bot$ **then** // found an unexplored node

        $Q' := Q' \cup \{v\}$ // remember for next layer

        $d[v] := \ell + 1$

        $parent(v) := u$ // update BFS tree

  $(Q,Q') := (Q',\langle\rangle)$ // switch to next layer

return $(d, parent)$ // the BFS tree is now $\{(v,w) : w \in V, v = parent(w)\}$

Green nodes

Blue nodes

Green nodes become red, blue nodes become green

See Mehlhorn/Sanders, Fig. 9.2