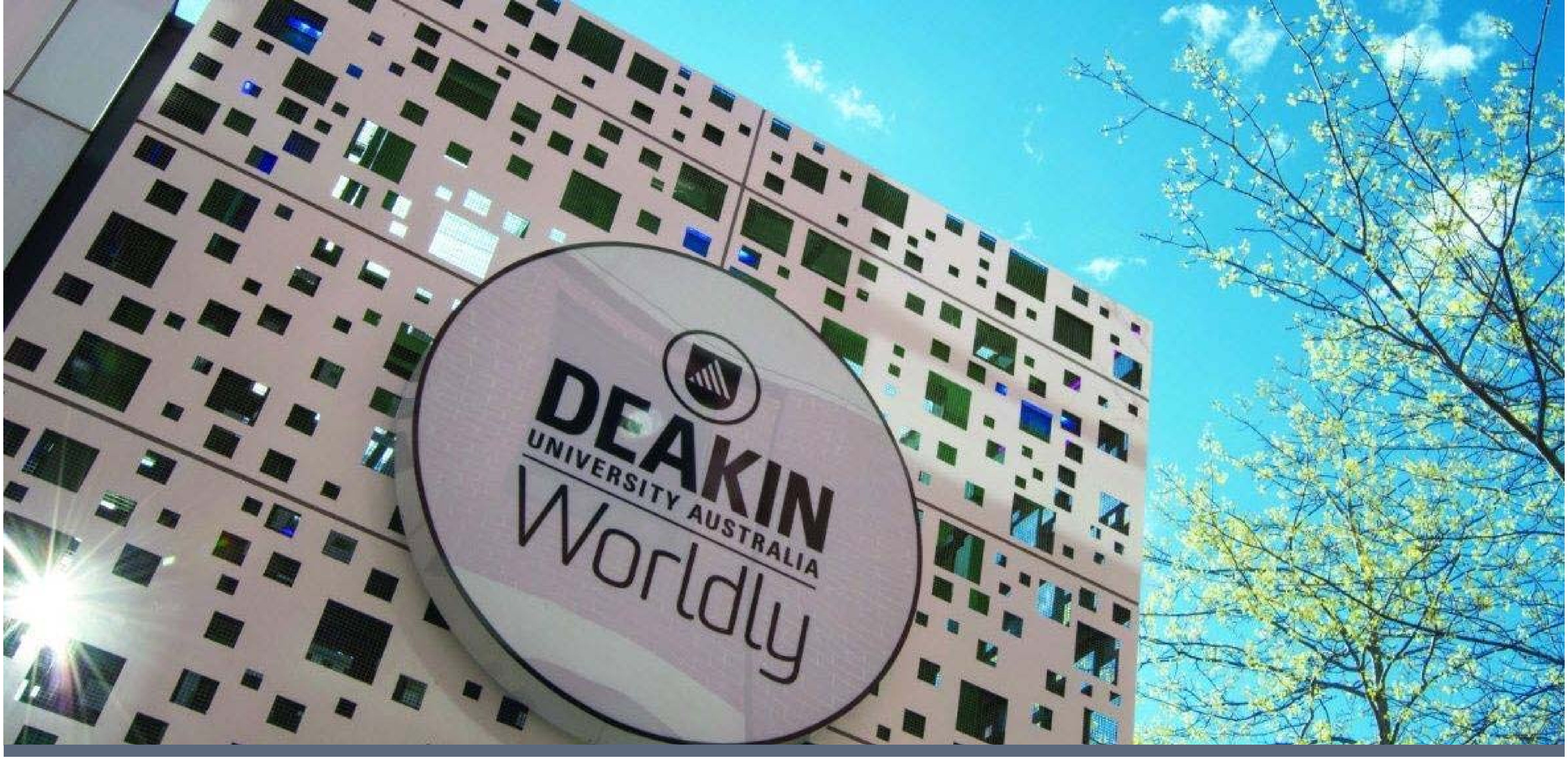


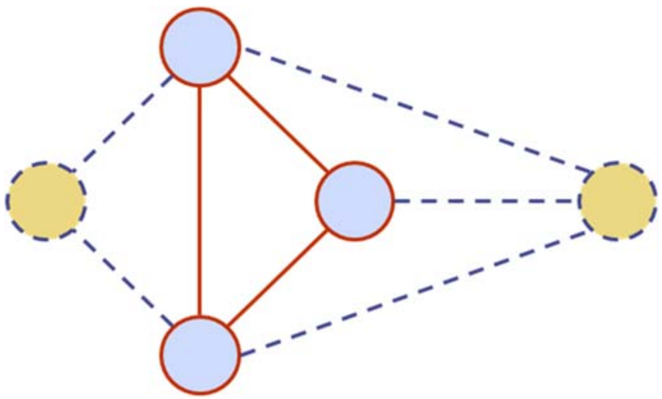
Lecture 8. Graph Traversal.

Depth-First Search and Breadth-First Search.

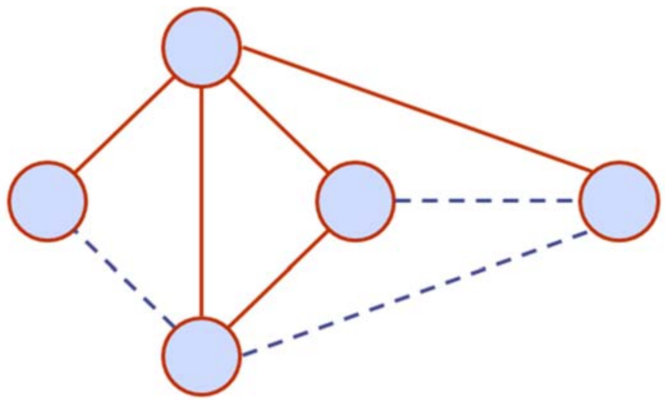
SIT221 Data Structures and Algorithms

Graph: Subgraphs

- A subgraph S of a graph G is a graph such that
  - the vertices of S are a subset of the vertices of G
  - the edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G.



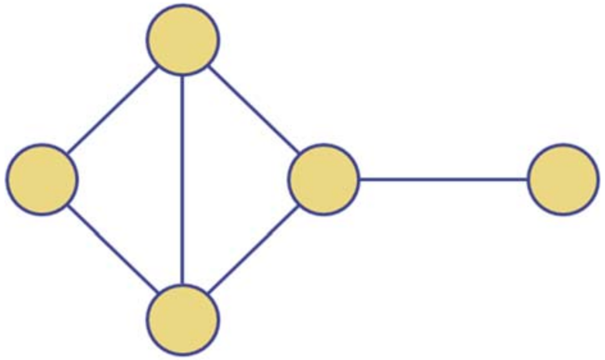
Subgraph



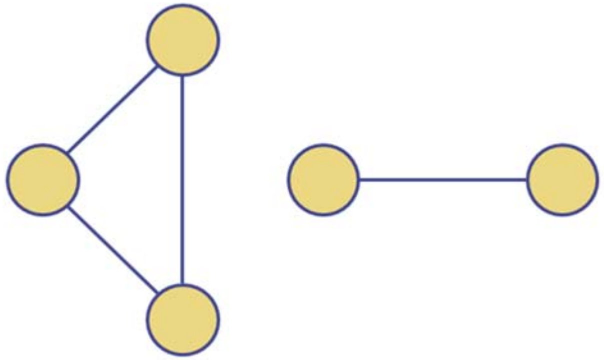
Spanning subgraph

Graph: Connectivity

- A graph is connected if there is a path between every pair of vertices.
- A connected component of a graph G is a maximal connected subgraph of G.



Connected graph

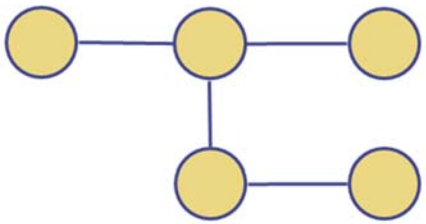


Non connected graph with two connected components

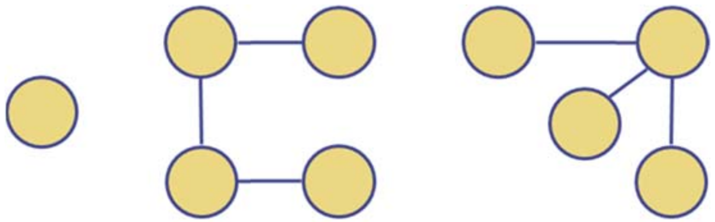
Graph: Trees and Forests

- A tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  (This definition of tree is different from the one of a rooted tree.)
- A forest is an undirected graph without cycles.
- The connected components of a forest are trees.



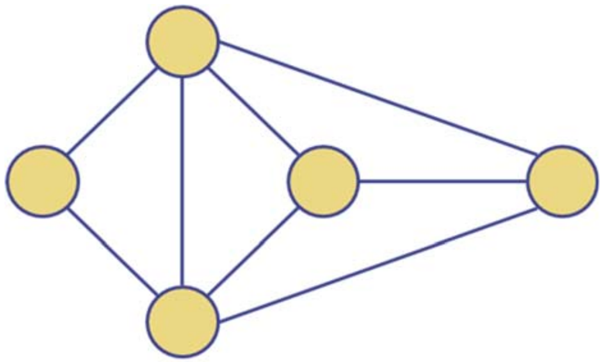
Tree



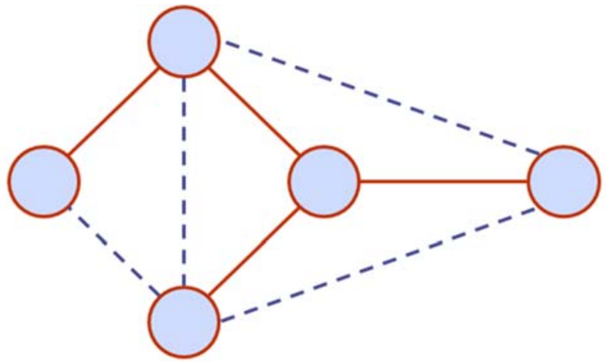
Forest

Graph: Spanning Trees

- A spanning tree of a connected graph is a spanning subgraph that is a tree.
- A spanning tree is not unique unless the graph is a tree.
- Spanning trees have applications to the design of communication networks.



Graph



Spanning tree

Graph Traversal

- A fundamental kind of algorithmic operation that we might wish to perform on a graph is traversing the edges and the vertices of that graph.
- A traversal is a systematic procedure for exploring a graph by examining all of its vertices and edges.
- For example, a web crawler, which is the data collecting part of a search engine, must explore a graph of hypertext documents by examining its vertices, which are the documents, and its edges, which are the hyperlinks between documents.
- A traversal is efficient if it visits all the vertices and edges in **linear time**.

Graph Traversal: Existing Techniques

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

There are two general techniques for traversing a graph:

- Depth-first search (DFS)
- Breadth-First Search (BFS)

Depth-First Search: Idea

Rule: For a given directed graph $G = (V, E)$, whenever you visit a vertex, explore in the next step one of its non-visited neighbours.

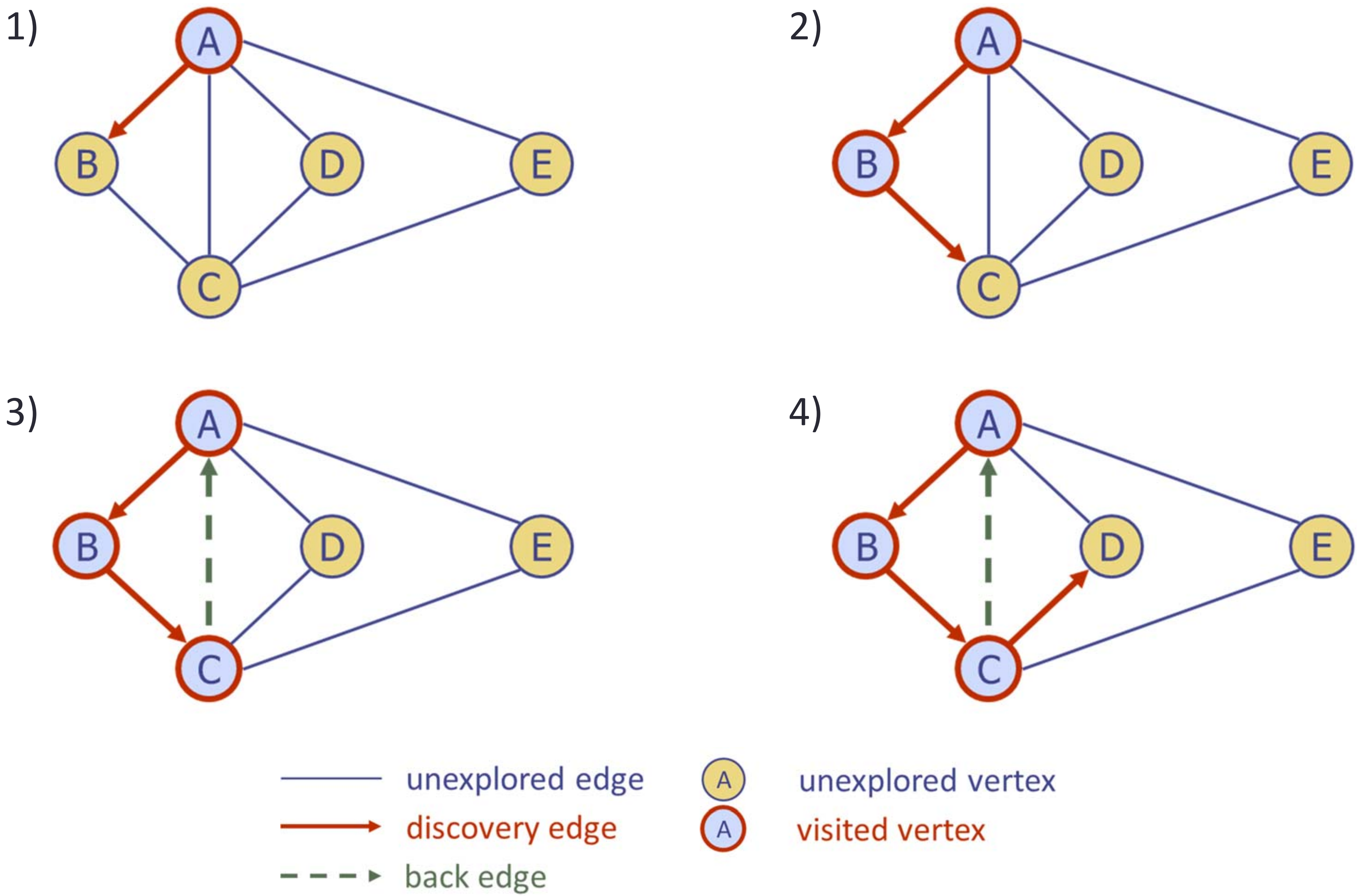
Implementation:

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

A DFS traversal of a graph G

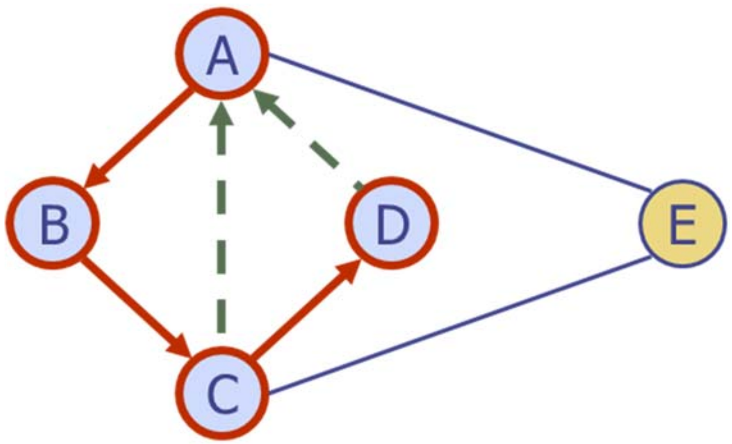
- Visits all the vertices and edges of G
- Determines whether G is connected
- Computes the connected components of G
- Computes a spanning tree of G

Depth-First Search: Example

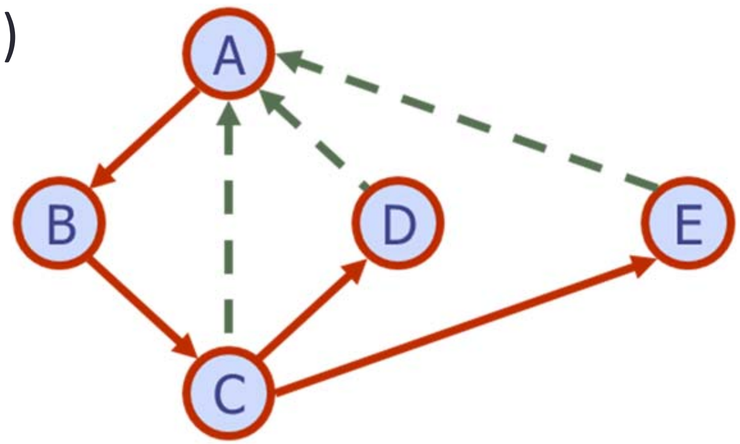


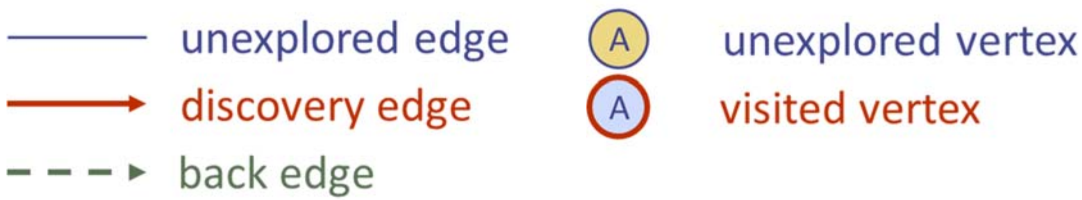
Depth-First Search: Example

5)



6)

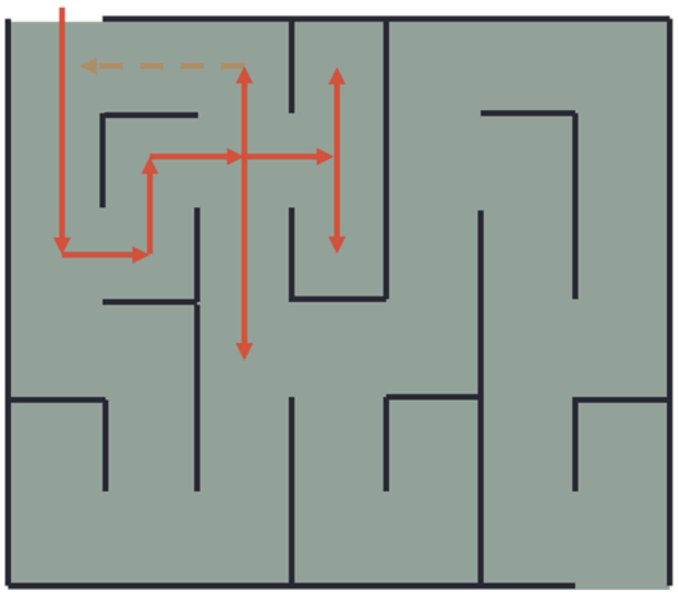




Depth-First Search: Analogy

The DFS algorithm is similar to a classic strategy for exploring a maze:

- We mark each intersection, corner and dead end (vertex) visited.
- We mark each corridor (edge) traversed.
- We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack).



Depth-First Search: Pseudocode

The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

Algorithm DFS(G, v):

Input: A graph G and a vertex v in G

Output: A labeling of the edges in the connected component of v as discovery edges and back edges, and the vertices in the connected component of v as explored

```
Label v as explored
for each edge, e, that is incident to v in G do
    if e is unexplored then
        Let w be the end vertex of e opposite from v
        if w is unexplored then
            Label e as a discovery edge
            DFS(G, w)
        else
            Label e as a back edge
```

Depth-First Search: Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- The method to determine incident edges is called once for each vertex. Note that $\sum_{v \in V} \deg(v) = 2m$ (sum of degrees).
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure.

* n is the number of vertices and m is the number of edges.

Depth-First Search: Properties

Property 1. $\mathrm{DFS}(G, v)$ visits all the vertices and edges in the connected component of v.

Property 2. The discovery edges labeled by $\mathrm{DFS}(G, v)$ form a spanning tree of the connected component of v.

- DFS on a graph with n vertices and m edges takes $\mathrm{O}(n + m)$ time.
- DFS can be further extended to solve other graph problems:
  - Find and report a path between two given vertices
  - Find a cycle in the graph

Depth-First Search: Path Finding

- We can adopt the DFS algorithm to find a path between two given vertices v and z.
- We call $\mathrm{DFS}(G, v)$ with v as the start vertex.
- We use a stack S to keep track of the path between the start vertex and the current vertex.
- As soon as destination vertex z is encountered, we return the path as the contents of the stack.

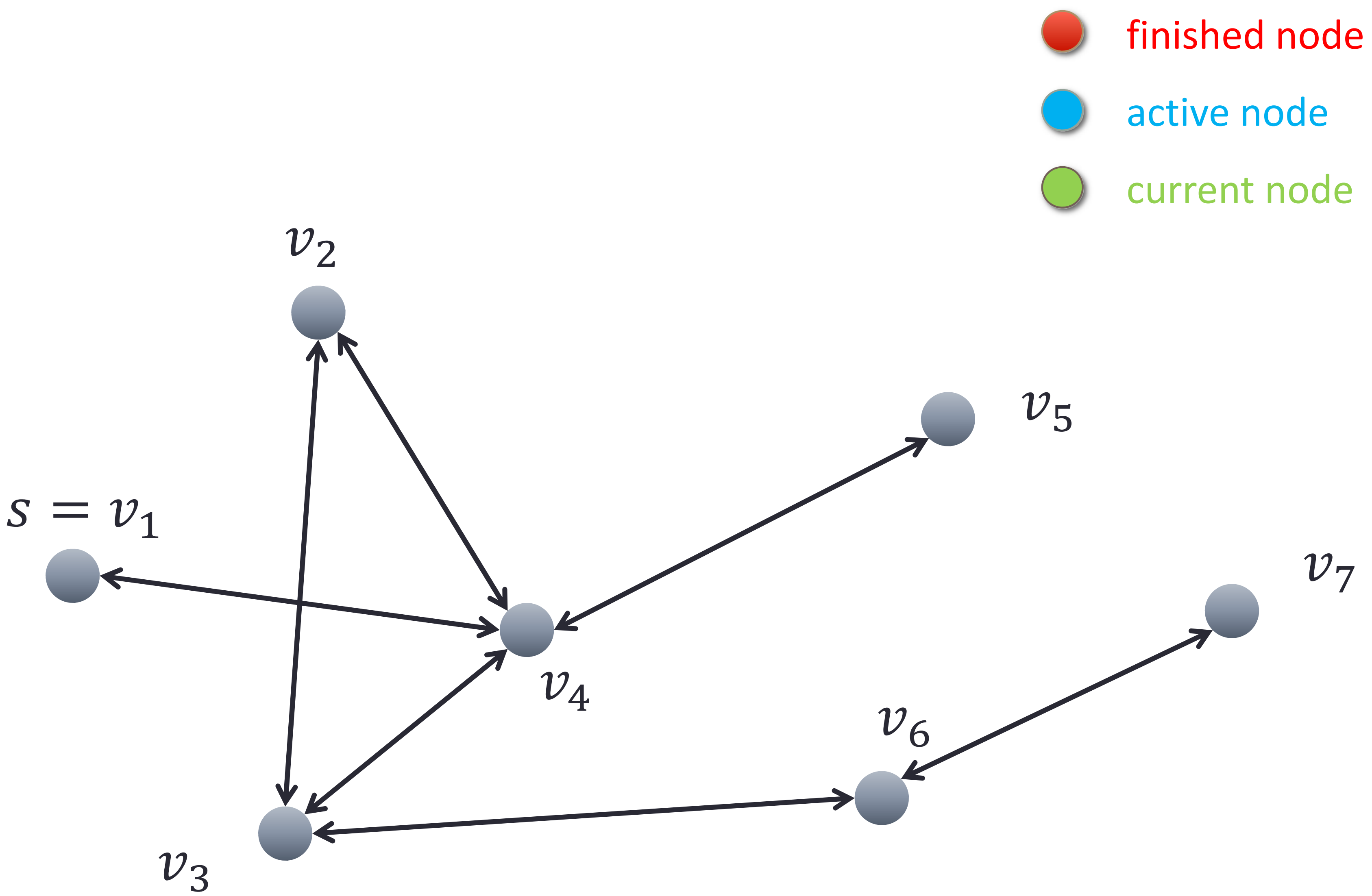
Algorithm pathDFS(G, v, z)

```
setLabel(v, VISITED);
S.push(v);
if ( v == z ) then return S.elements();
foreach ( e in G.incidentEdges(v) )
    if ( getLabel(e) == UNEXPLORED ) then w = opposite(v,e);
        if ( getLabel(w) == UNEXPLORED ) then
            setLabel(e, DISCOVERY);
            S.push(e);
            pathDFS(G, w, z);
            S.pop(e);
        else setLabel(e, BACK);
S.pop(v);
```

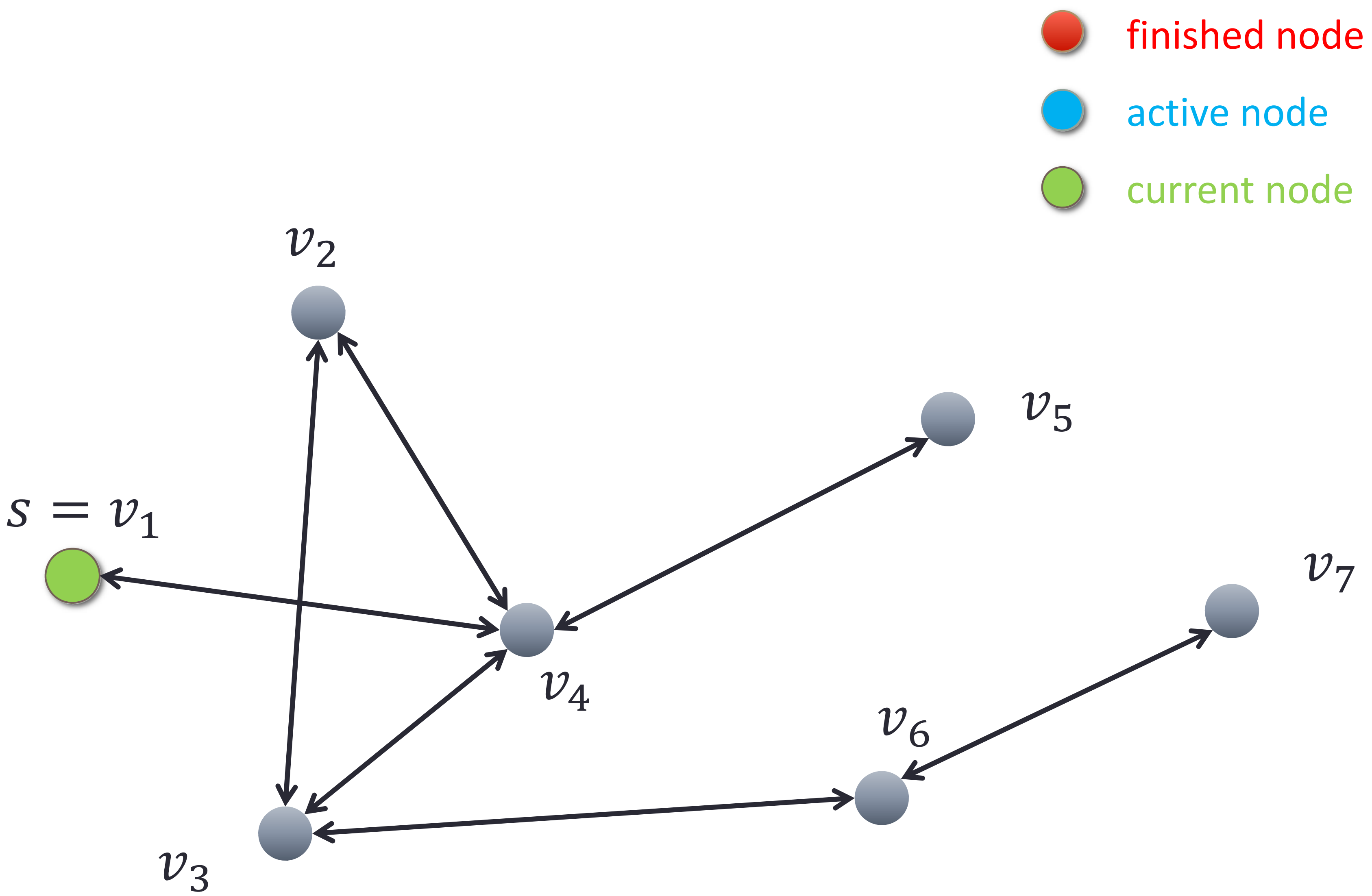
Depth-First Search: Cycle Finding

- We can similarly adopt the DFS algorithm to find a simple cycle.
- We can use a stack S to keep track of the path between the start vertex and the current vertex.
- As soon as a **back edge** (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w.

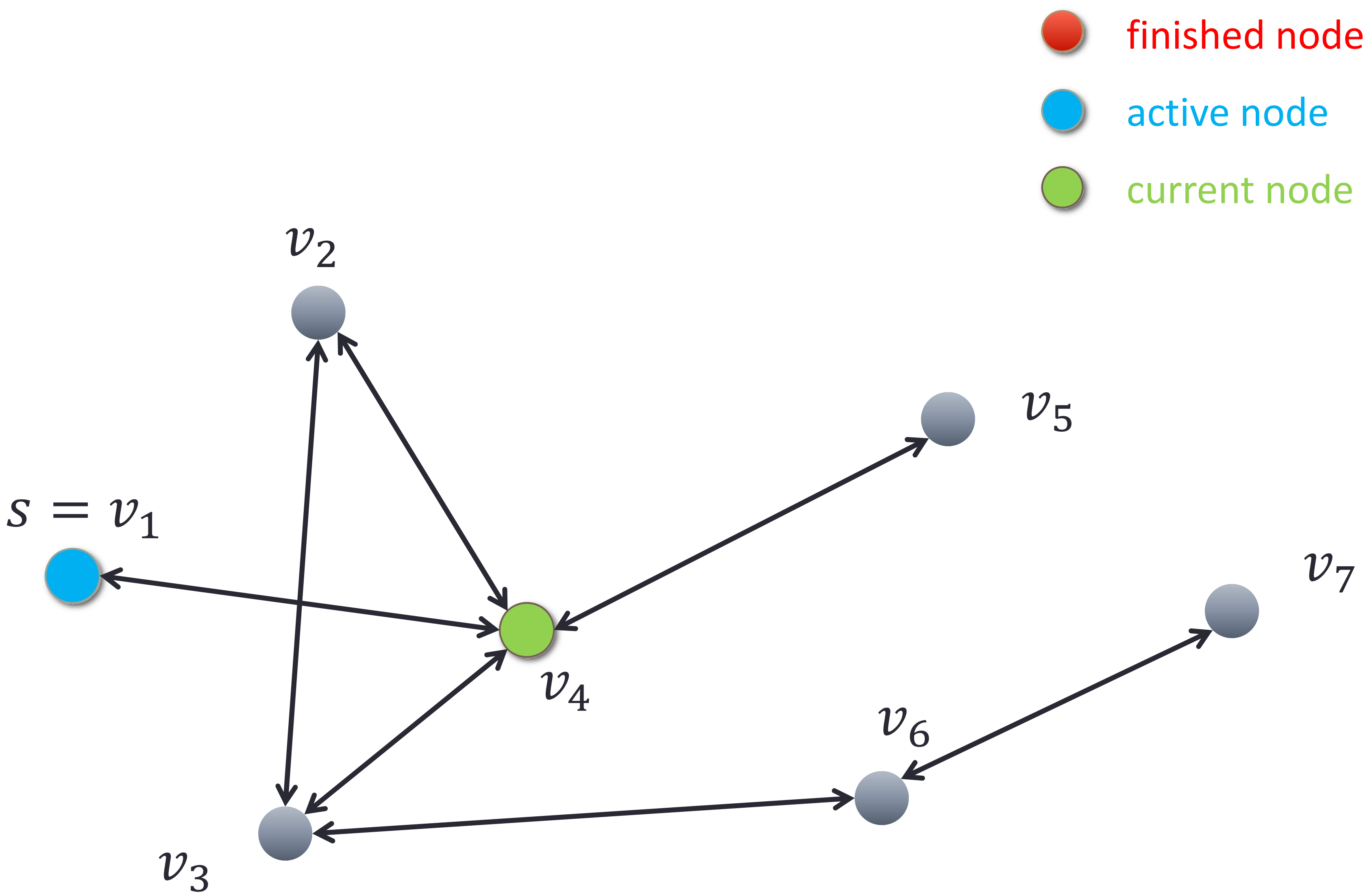
Depth-First Search: DFS Tree



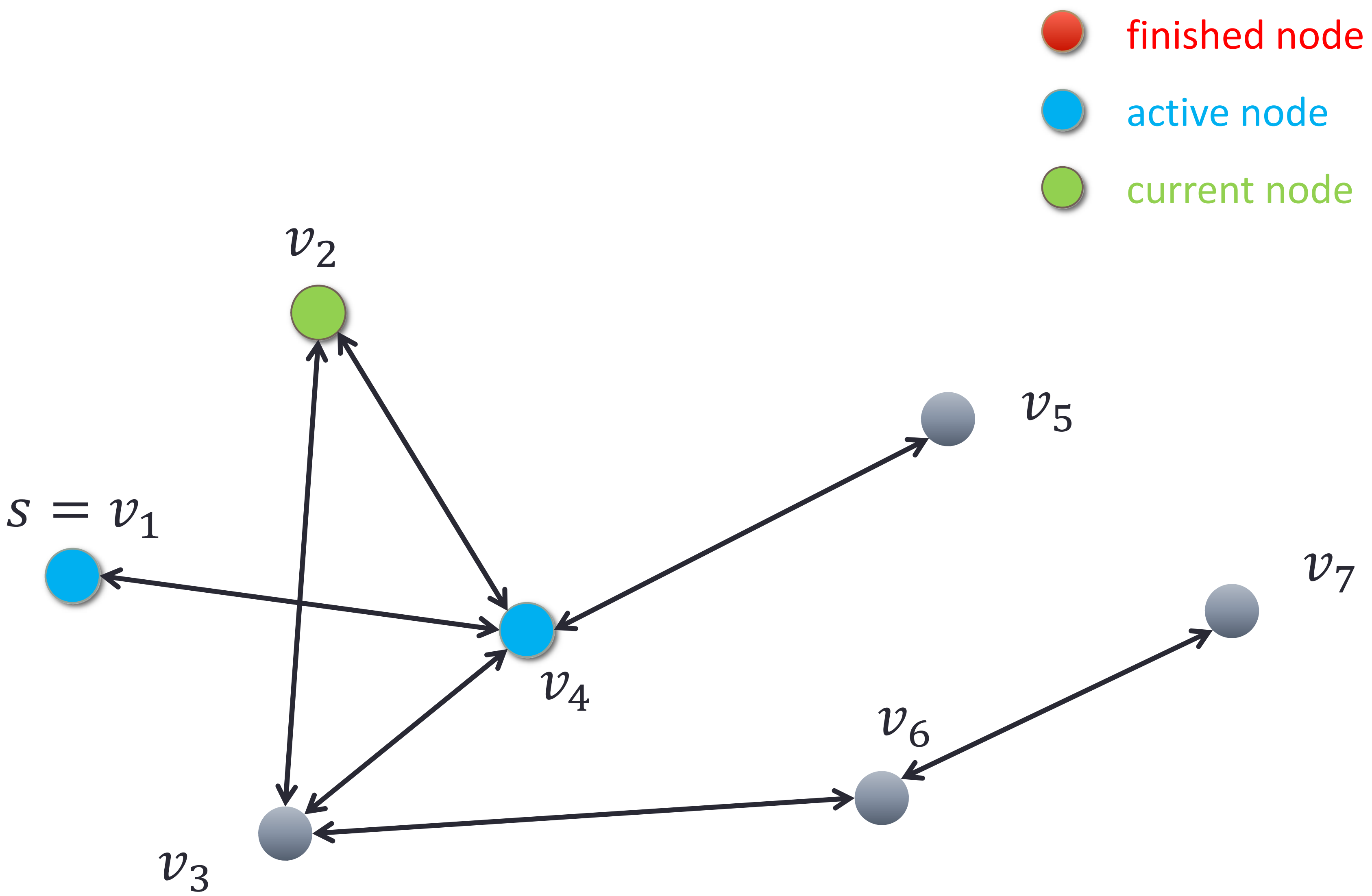
Depth-First Search: DFS Tree



Depth-First Search: DFS Tree



Depth-First Search: DFS Tree

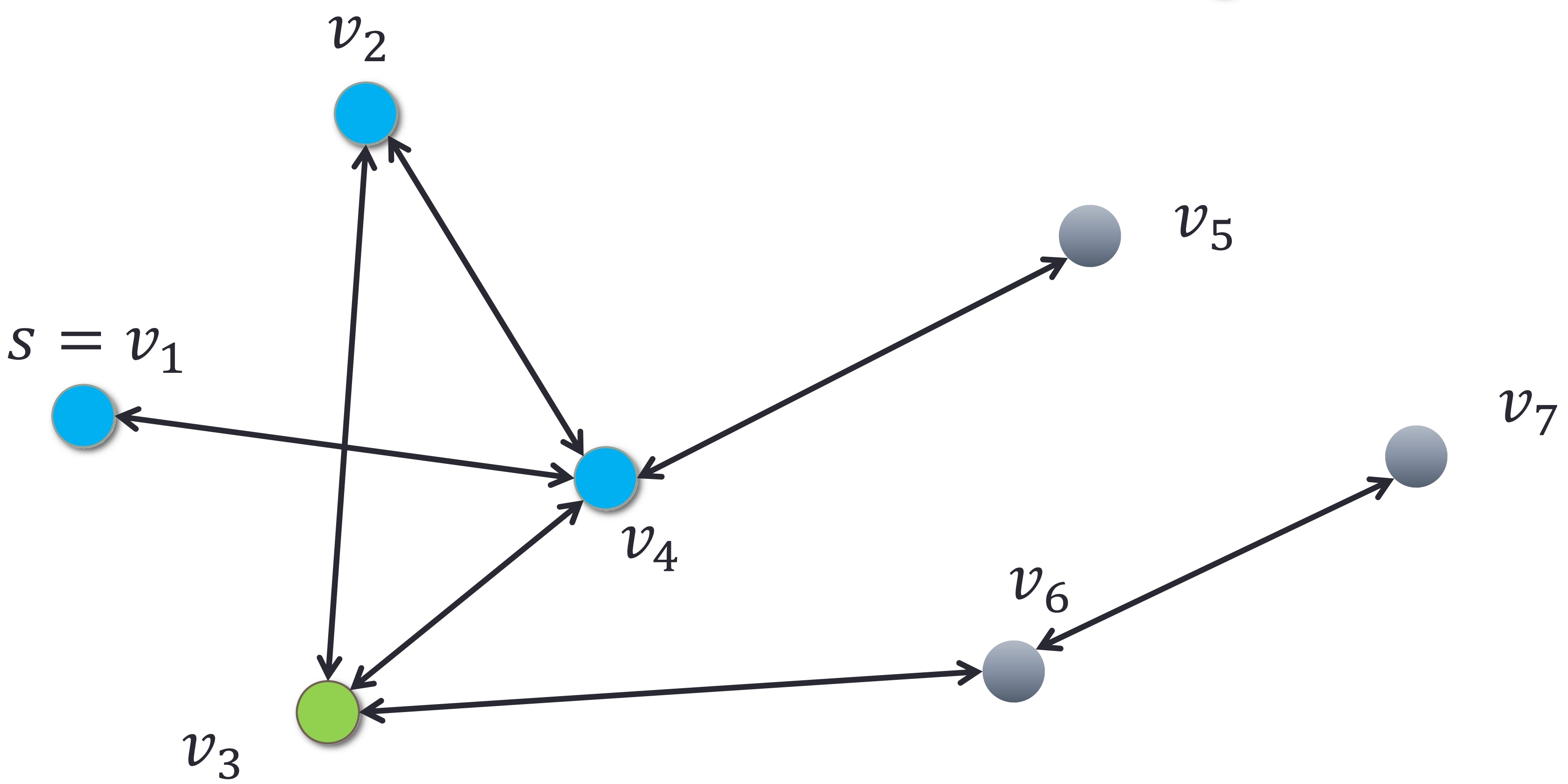


Depth-First Search: DFS Tree

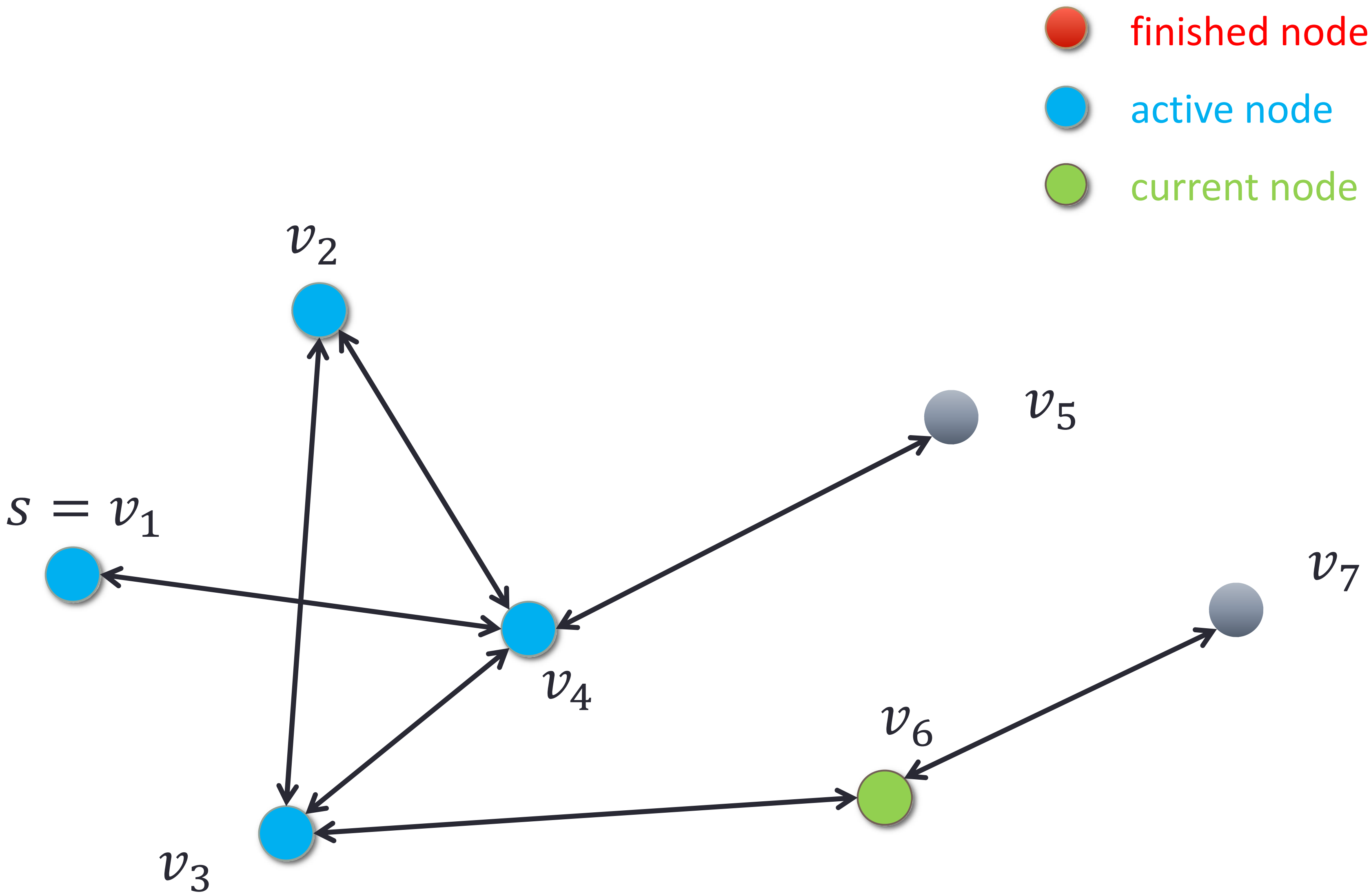
finished node

active node

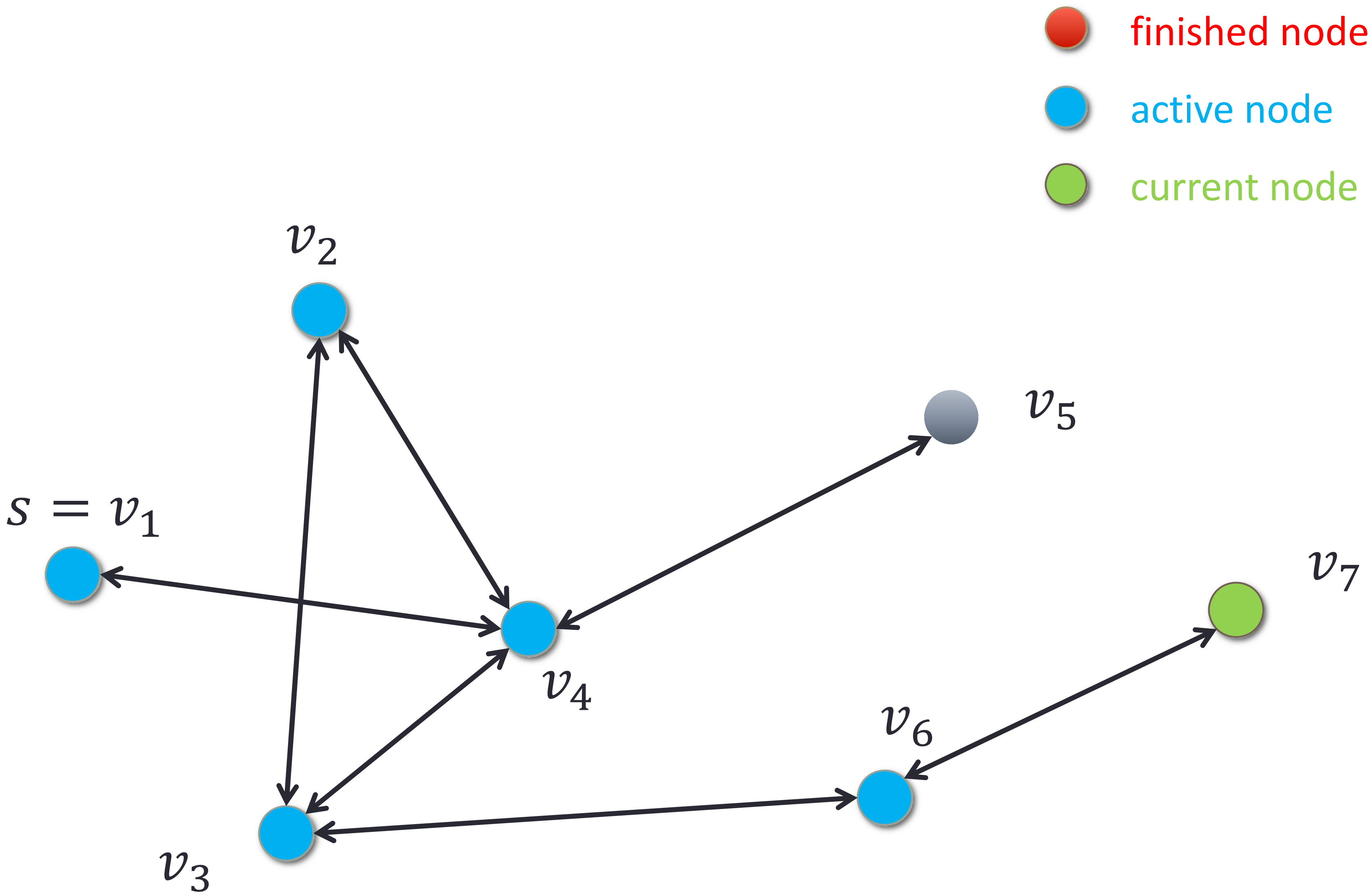
current node



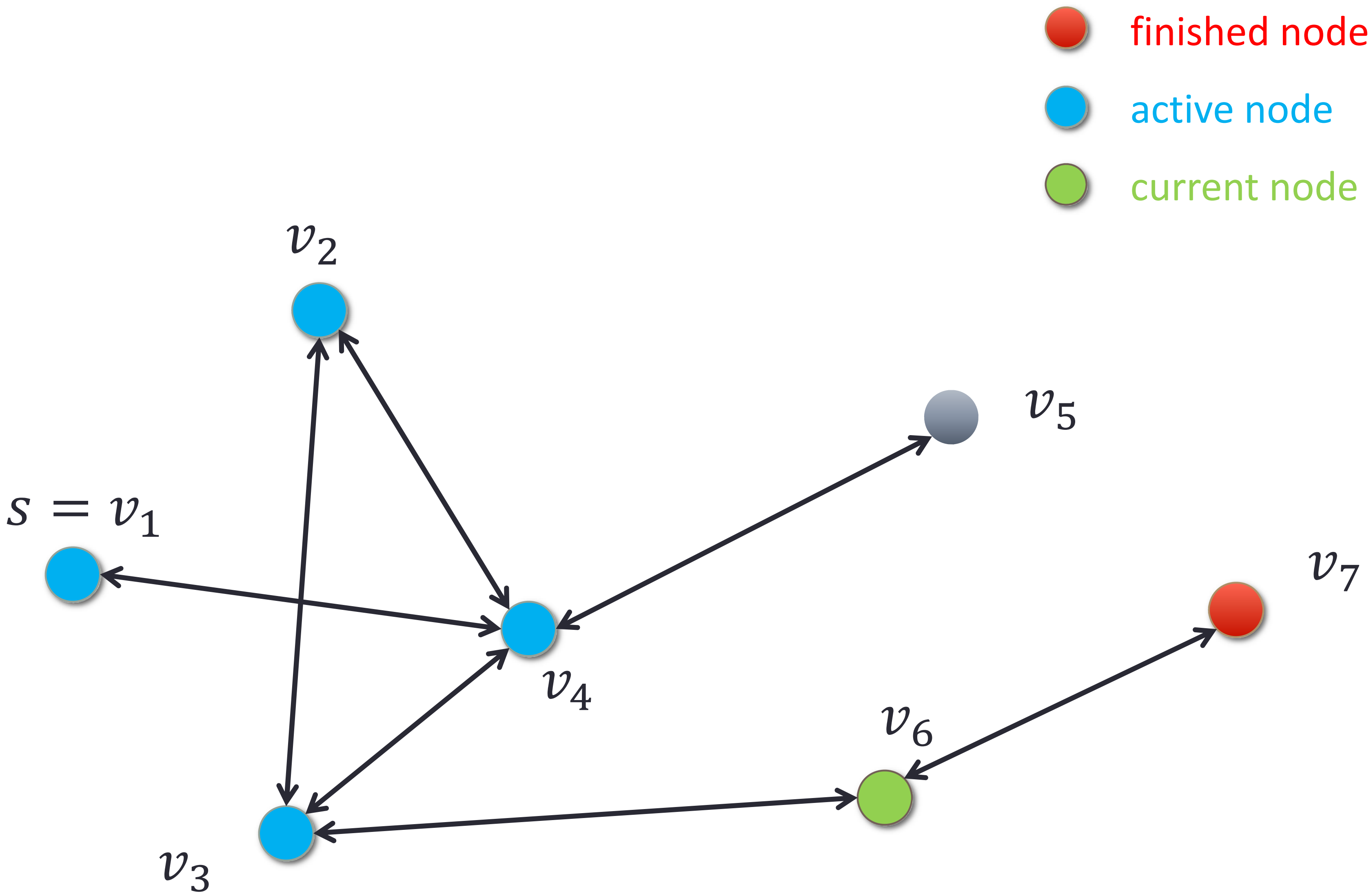
Depth-First Search: DFS Tree



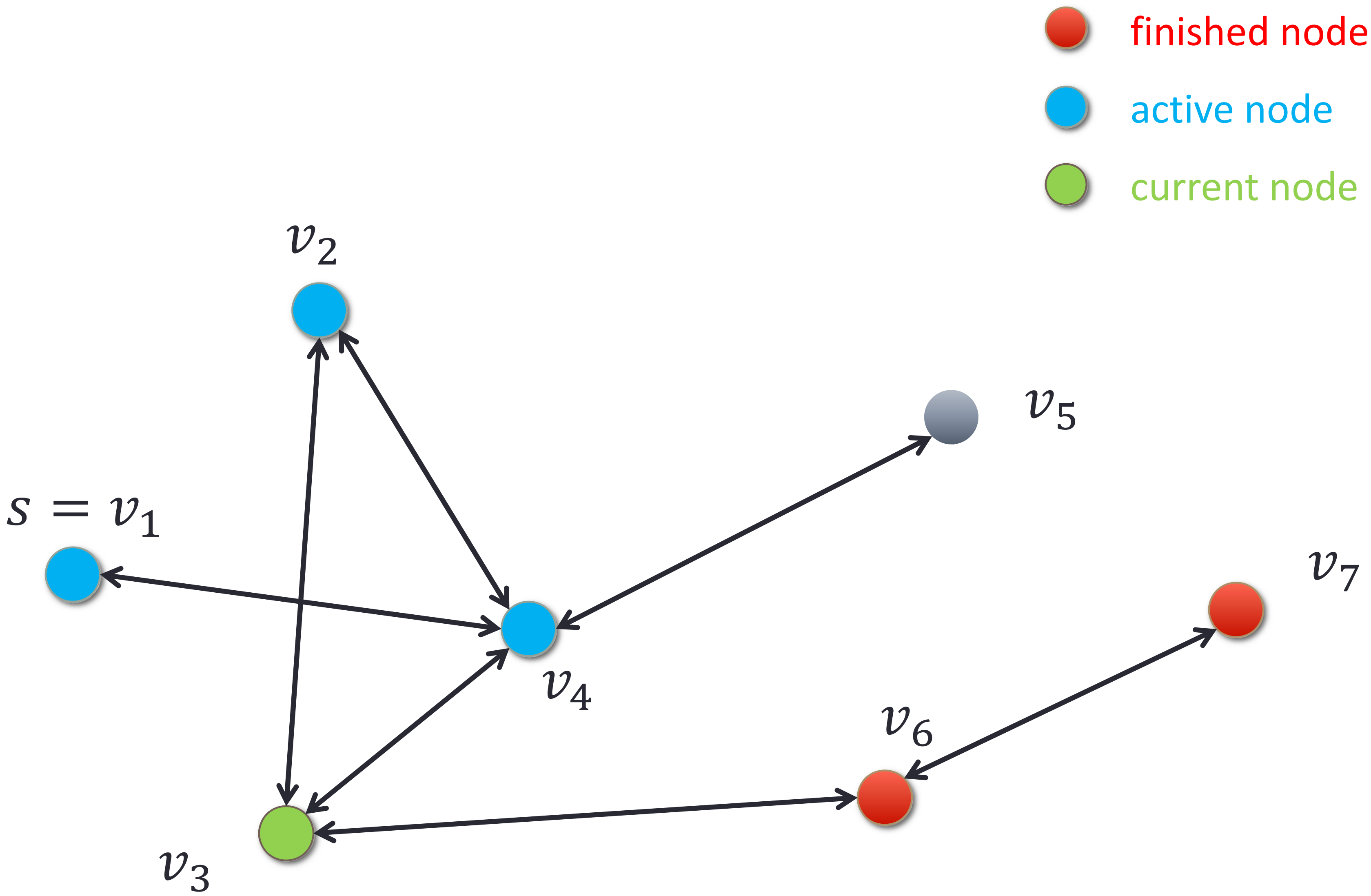
Depth-First Search: DFS Tree



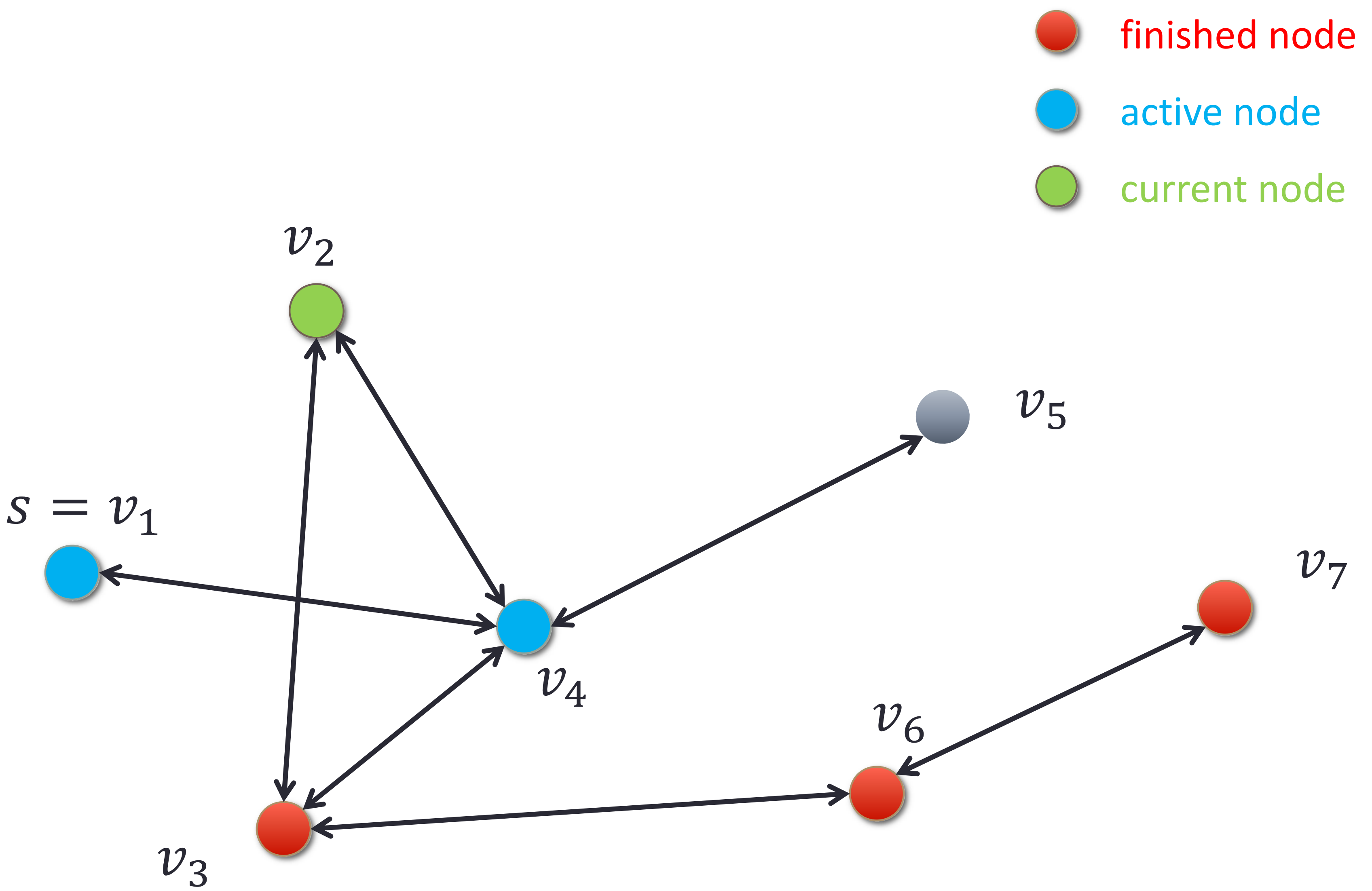
Depth-First Search: DFS Tree



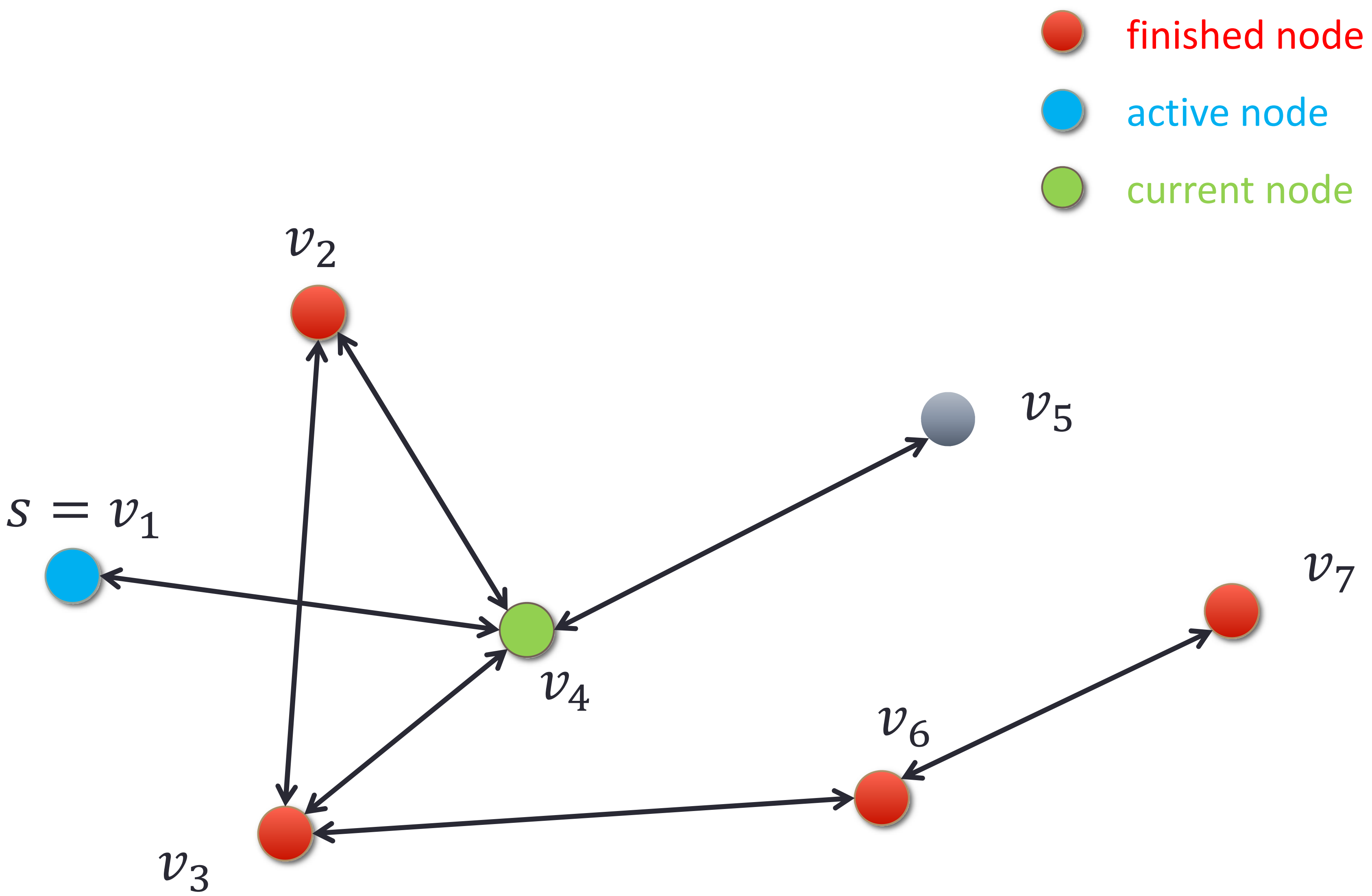
Depth-First Search: DFS Tree



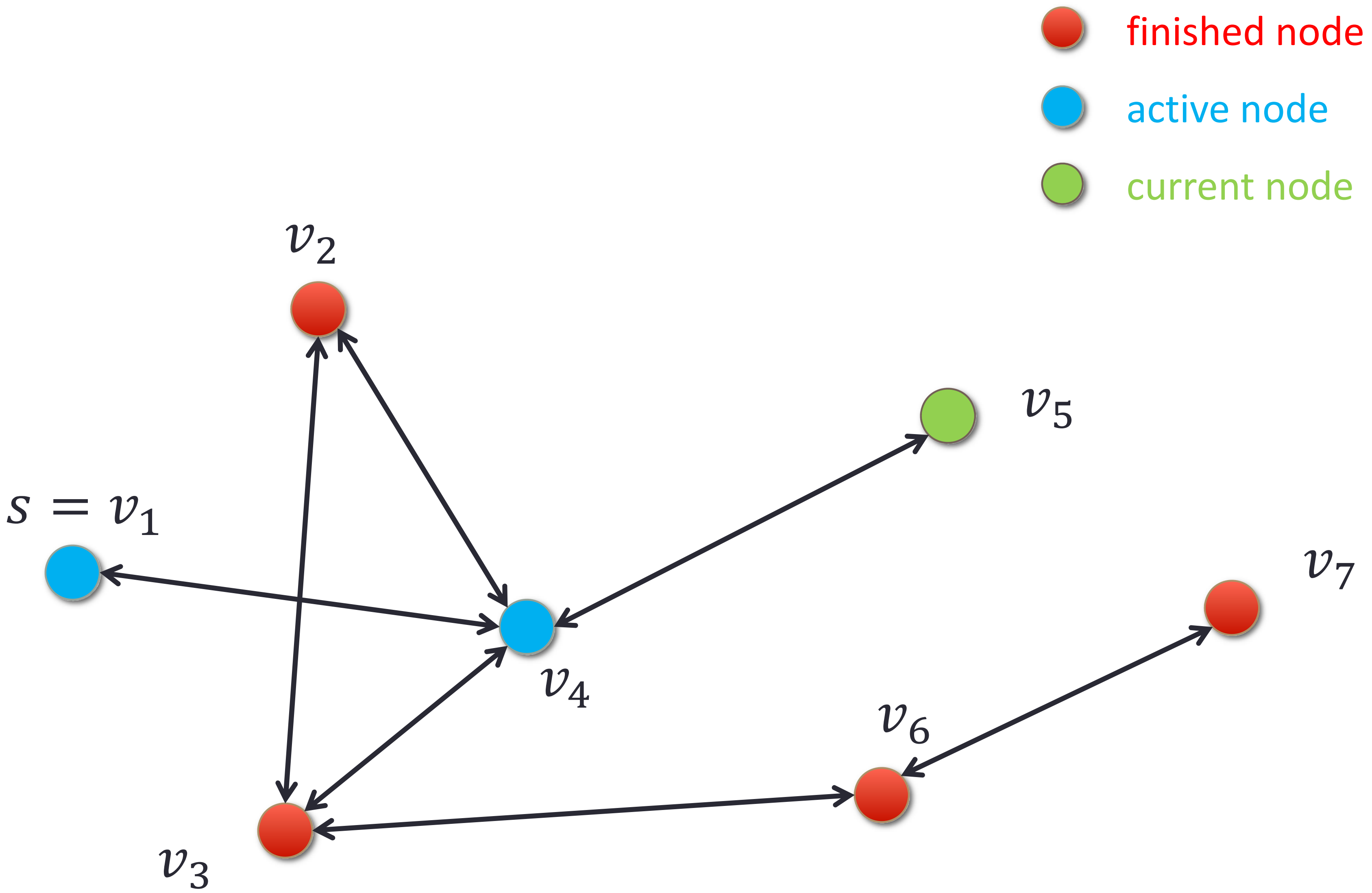
Depth-First Search: DFS Tree



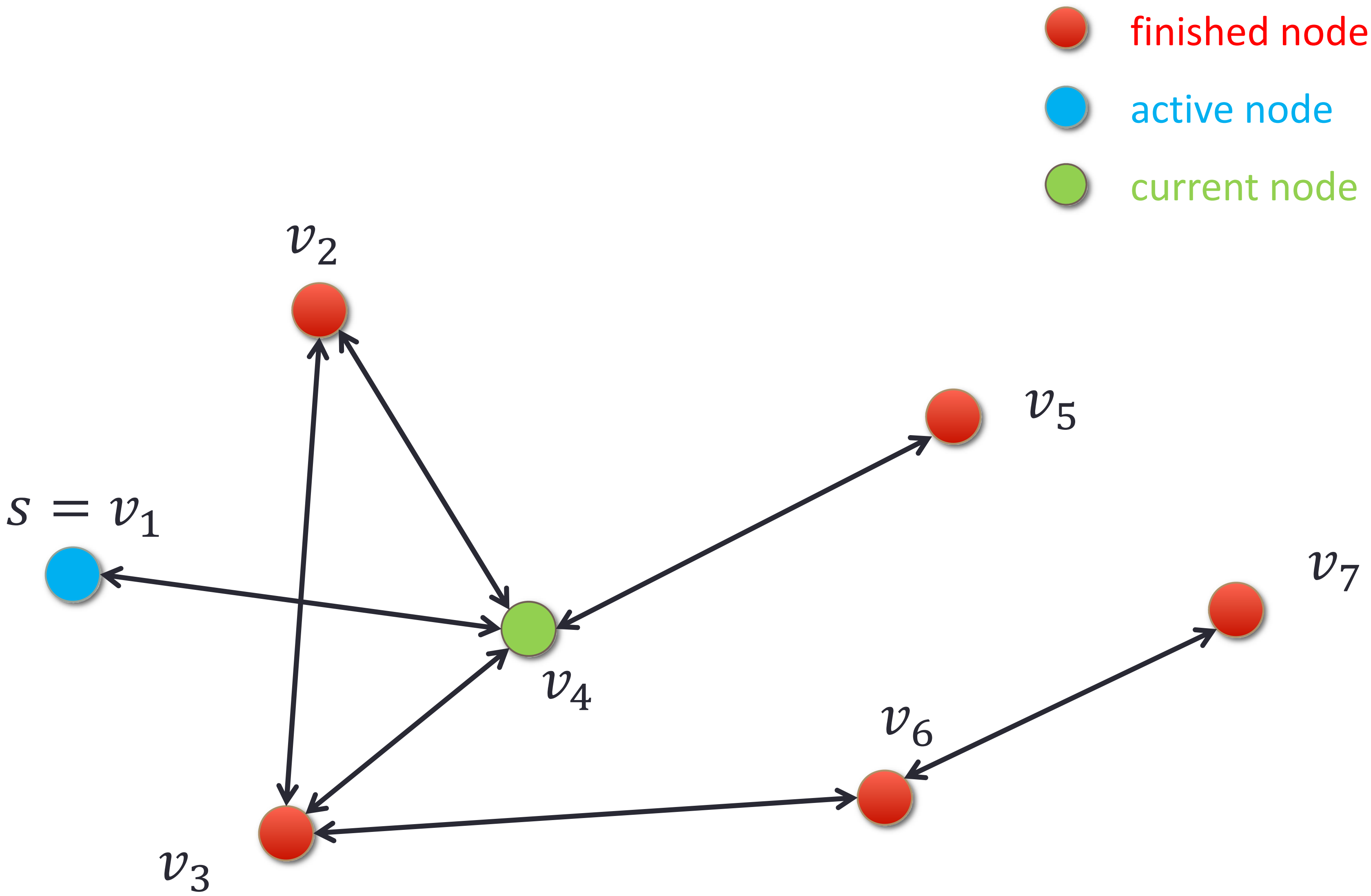
Depth-First Search: DFS Tree



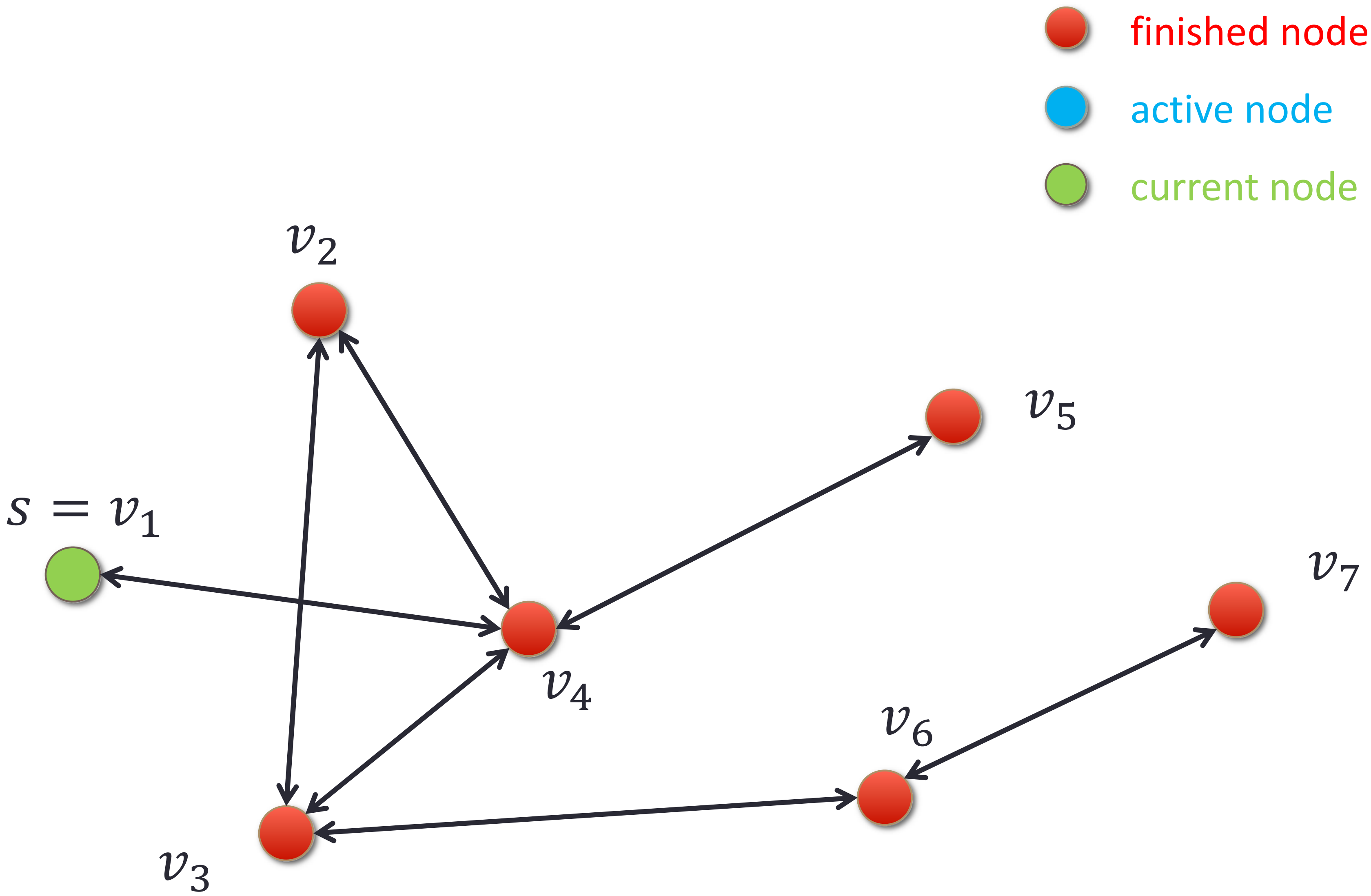
Depth-First Search: DFS Tree



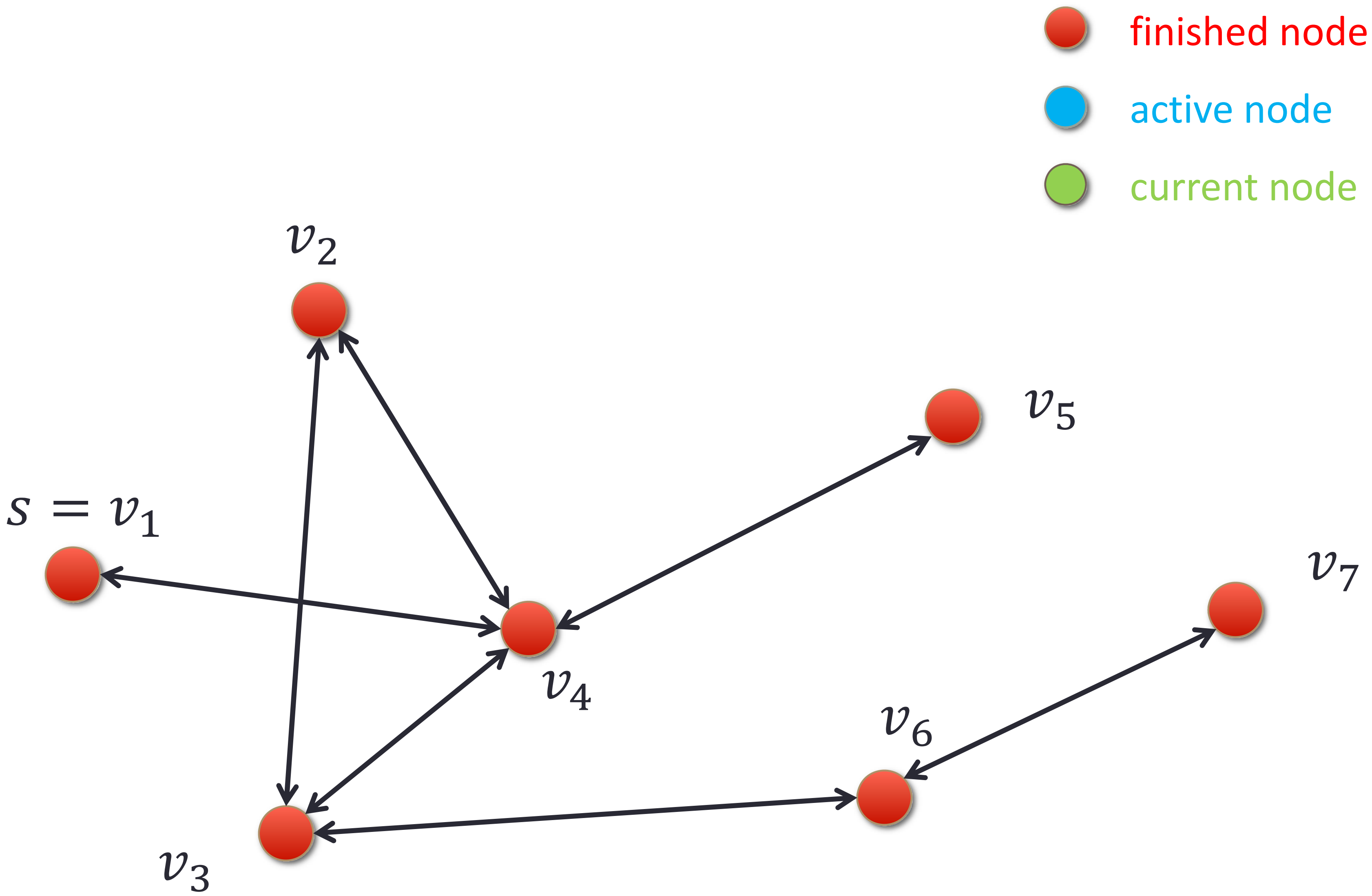
Depth-First Search: DFS Tree



Depth-First Search: DFS Tree



Depth-First Search: DFS Tree

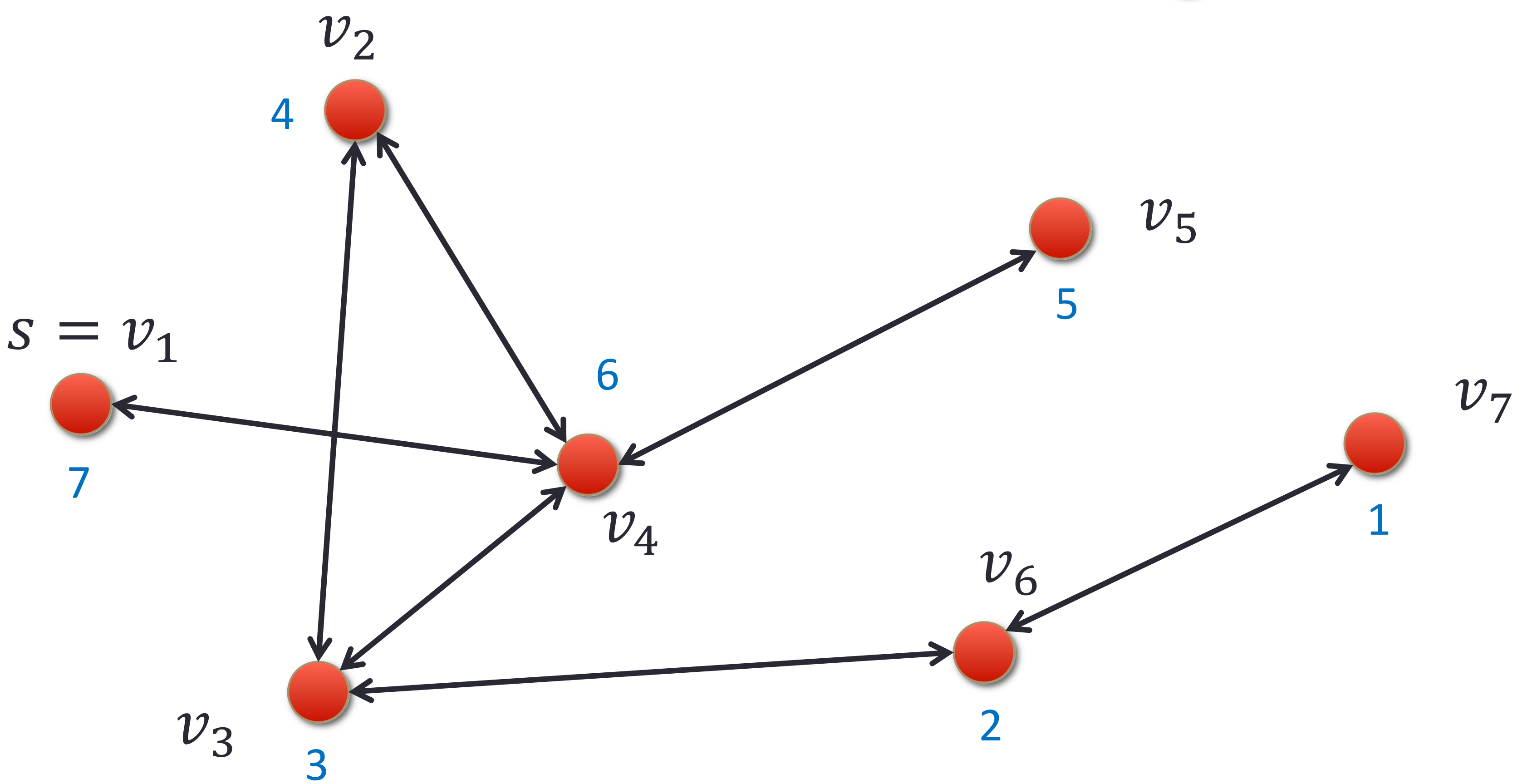


Depth-First Search: DFS Tree

finished node

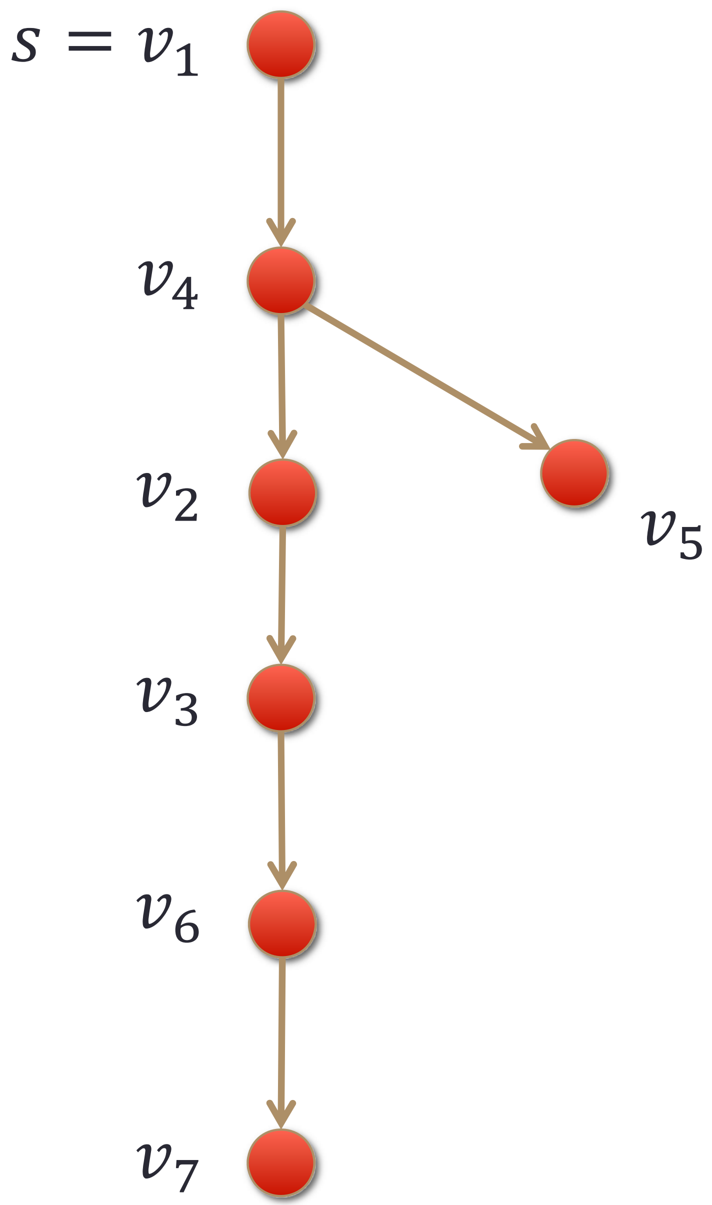
active node

current node



Order in which nodes are finished

Depth-First Search: DFS Tree



Breadth-First Search: Idea

Rule: For a given directed graph $G = (V, E)$, whenever you visit a vertex, explore in iteration i all its non-visited neighbours.

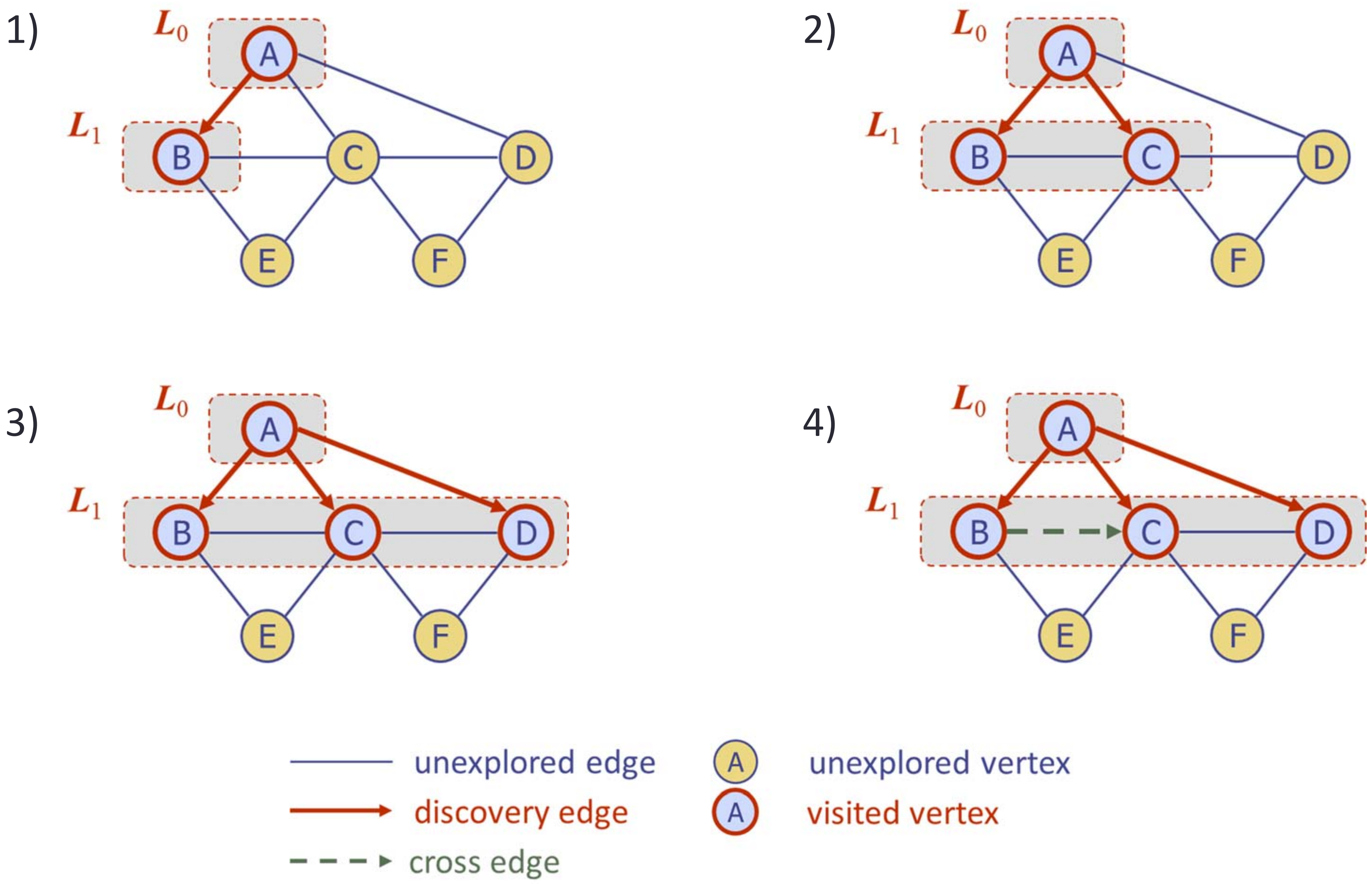
Implementation:

- When visiting a node, mark it as visited and iteratively analyse all its non-visited neighbours forming a joint collection to be examined at once in the next iteration.
- Explore all the nodes in the current collection iteratively. If no non-visited neighbours appear, terminate the algorithm.

A BFS traversal of a graph G

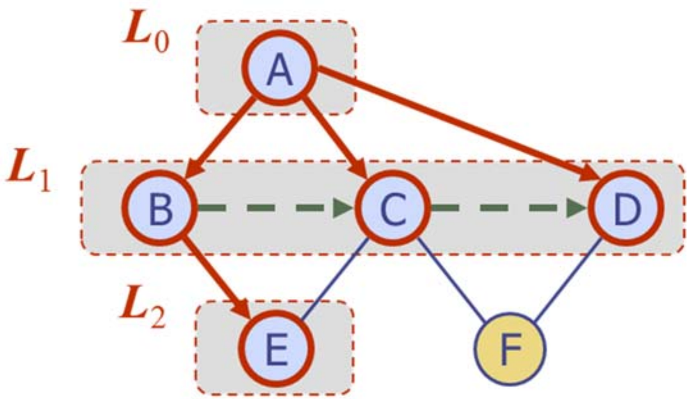
- Visits all the vertices and edges of G
- Determines whether G is connected
- Computes the connected components of G

Breadth-First Search: Example

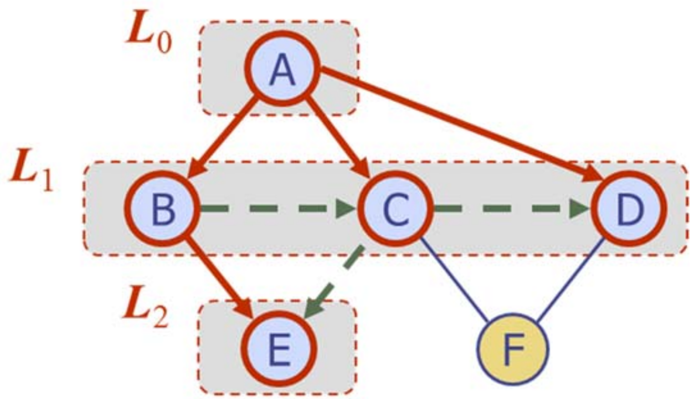


Breadth-First Search: Example

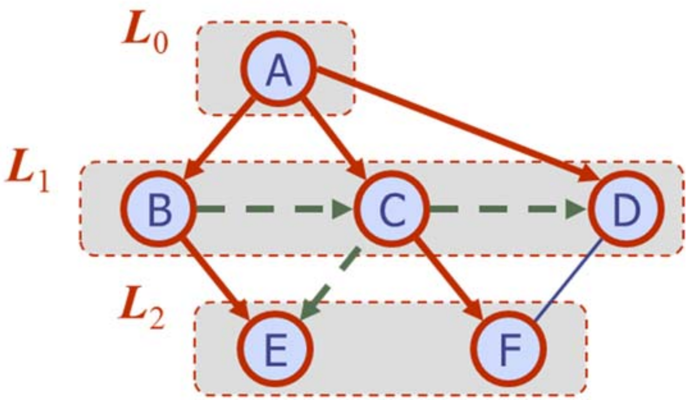
5)



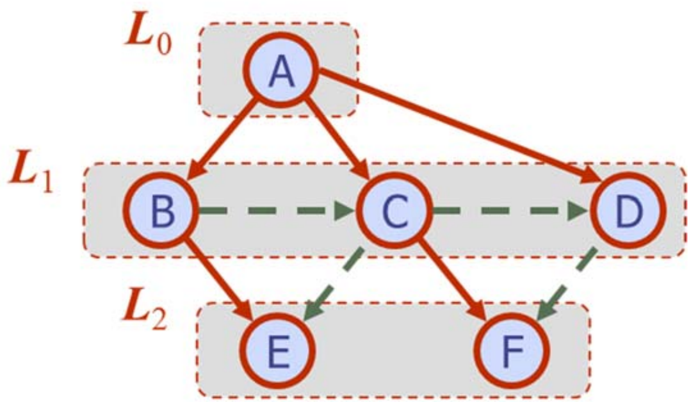
6)

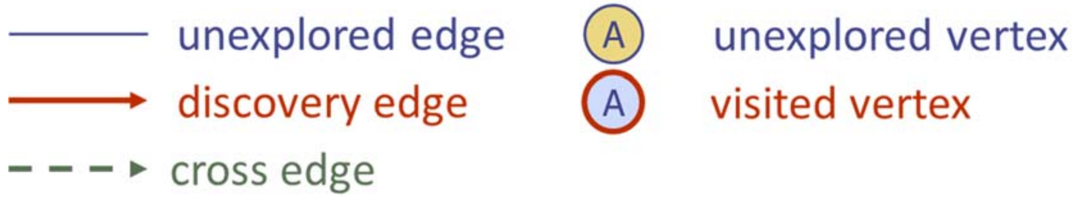


7)



8)



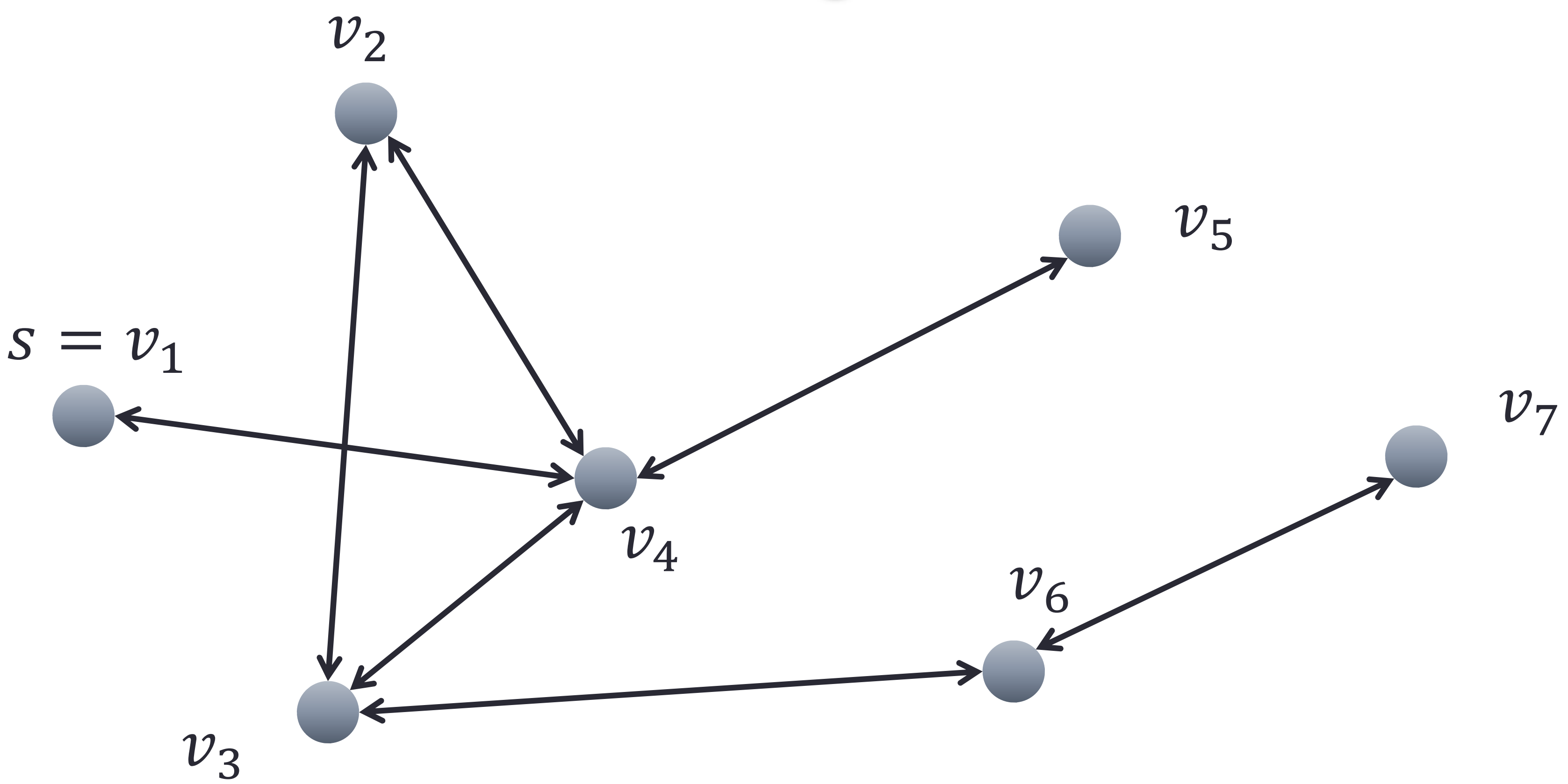


Breadth-First Search: BFS Tree

finished node

next node to be considered (next layer)

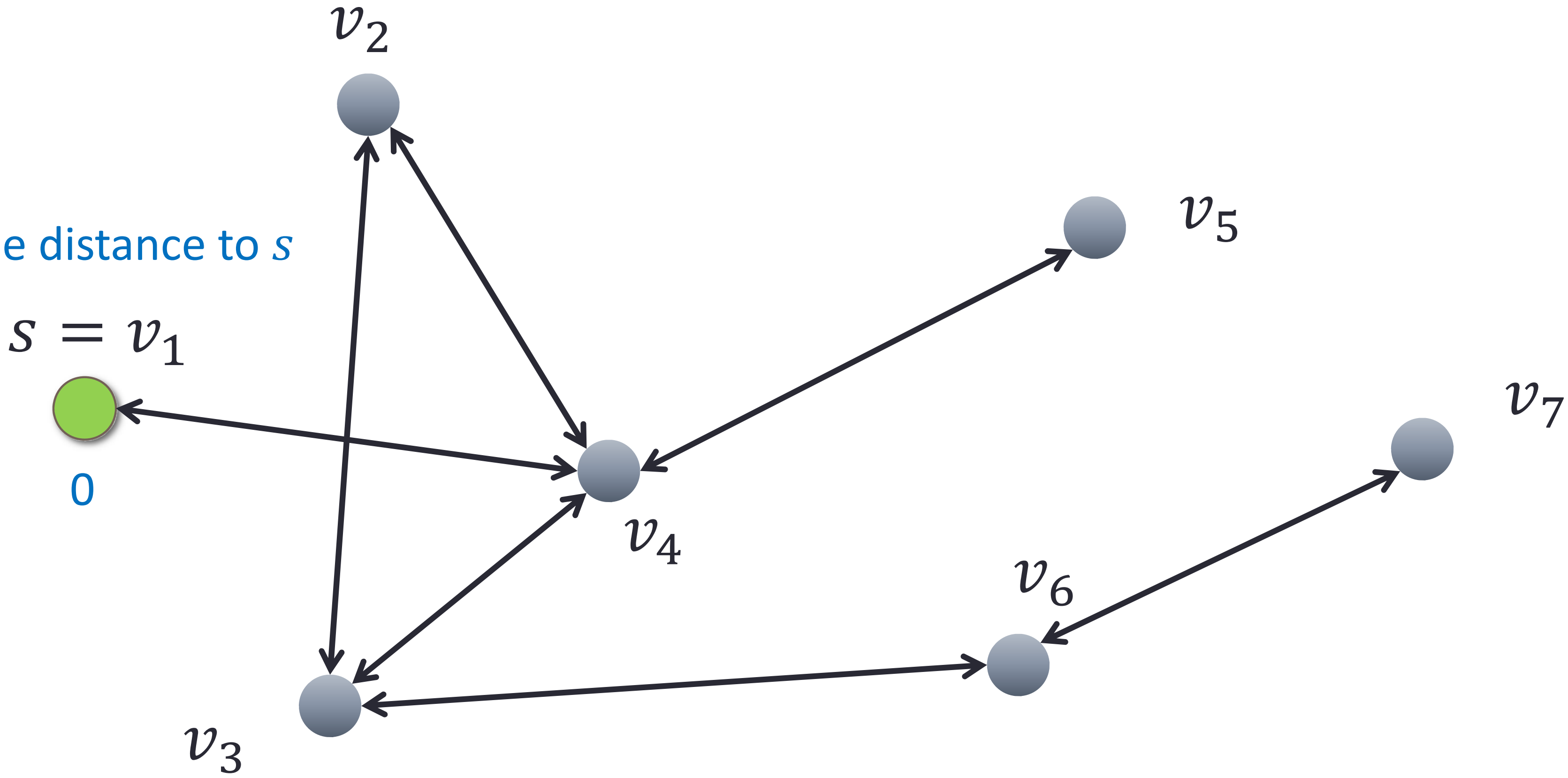
active node (current layer)



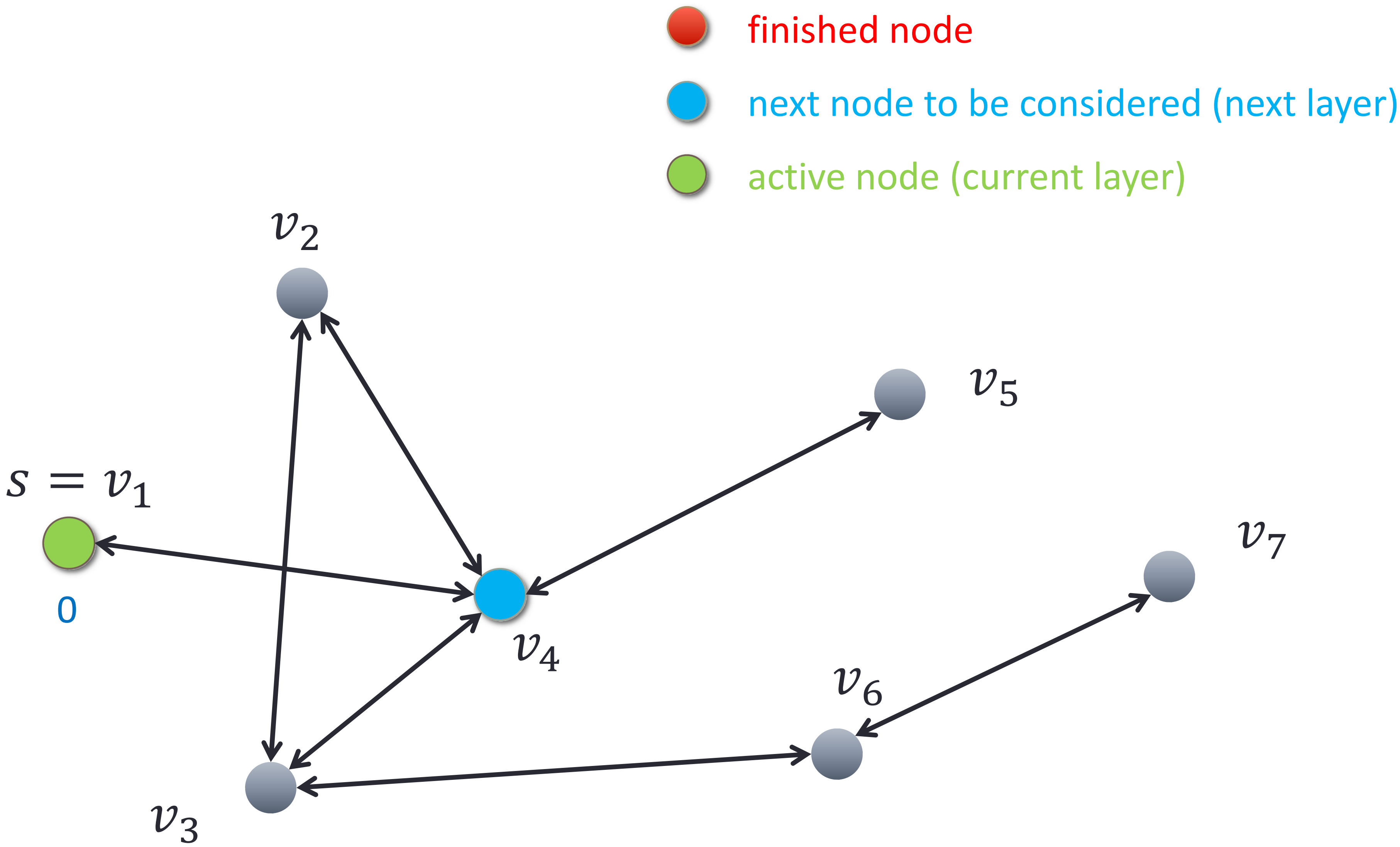
Breadth-First Search: BFS Tree

- finished node
- next node to be considered (next layer)
- active node (current layer)

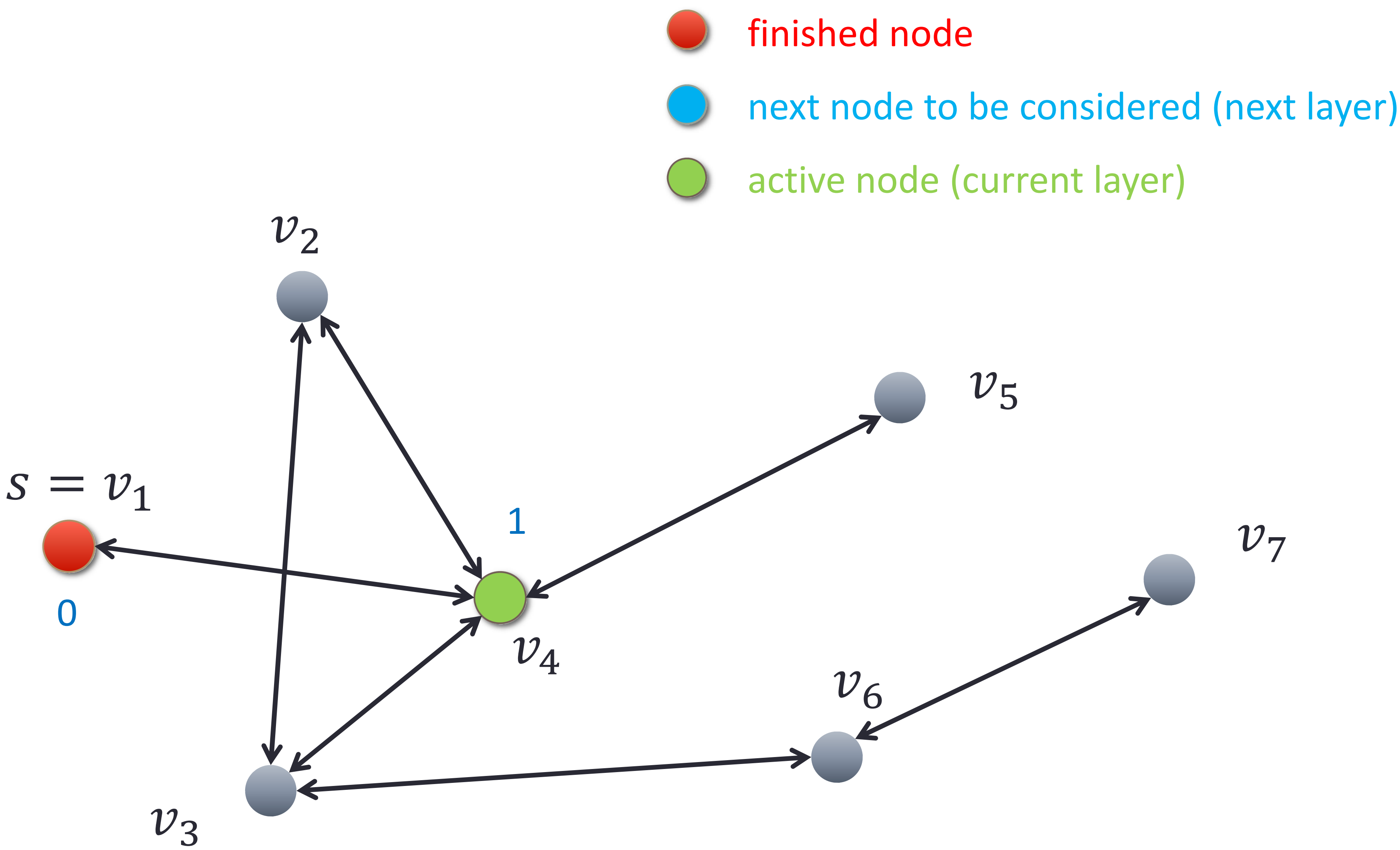
Mark the distance to s



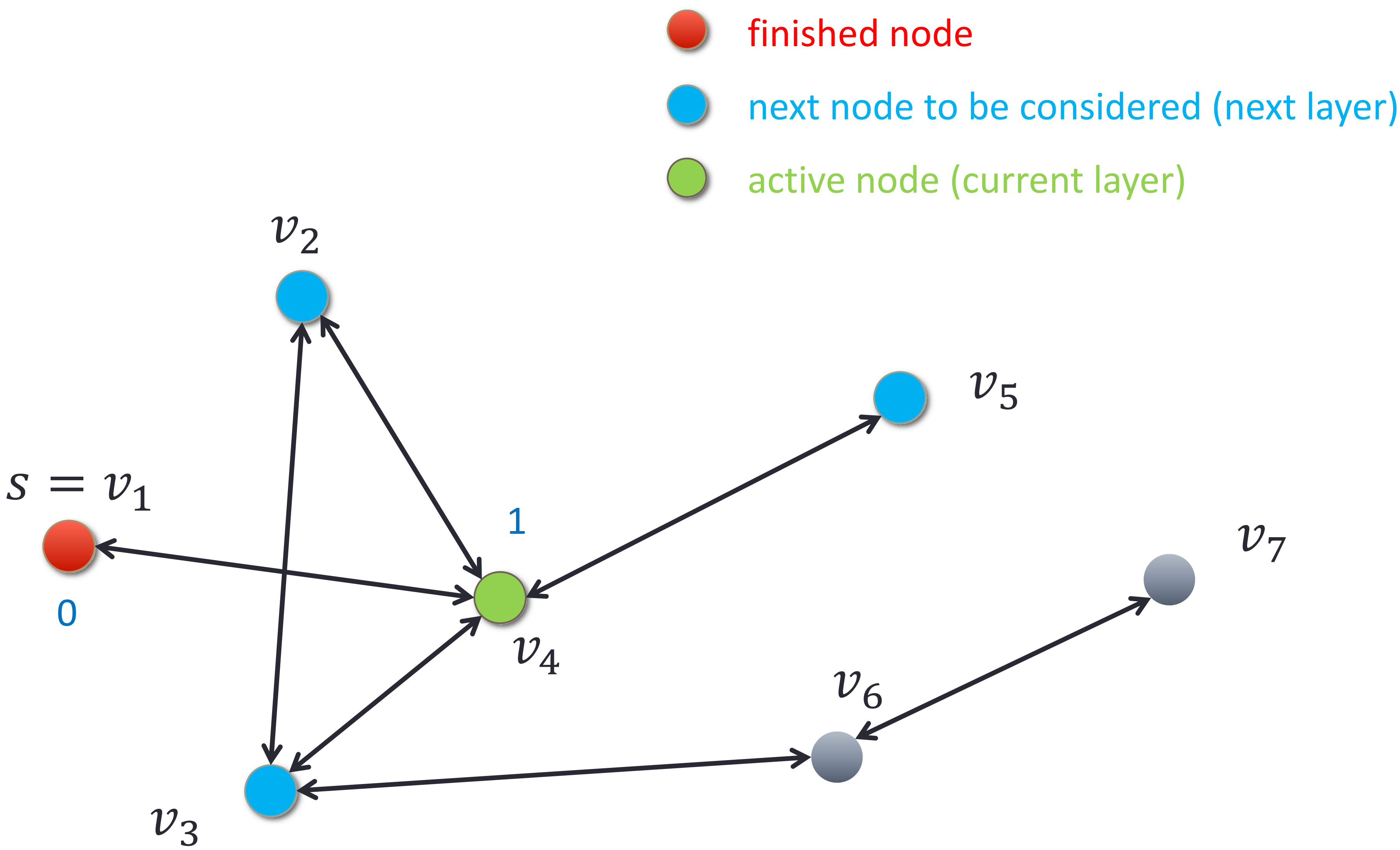
Breadth-First Search: BFS Tree



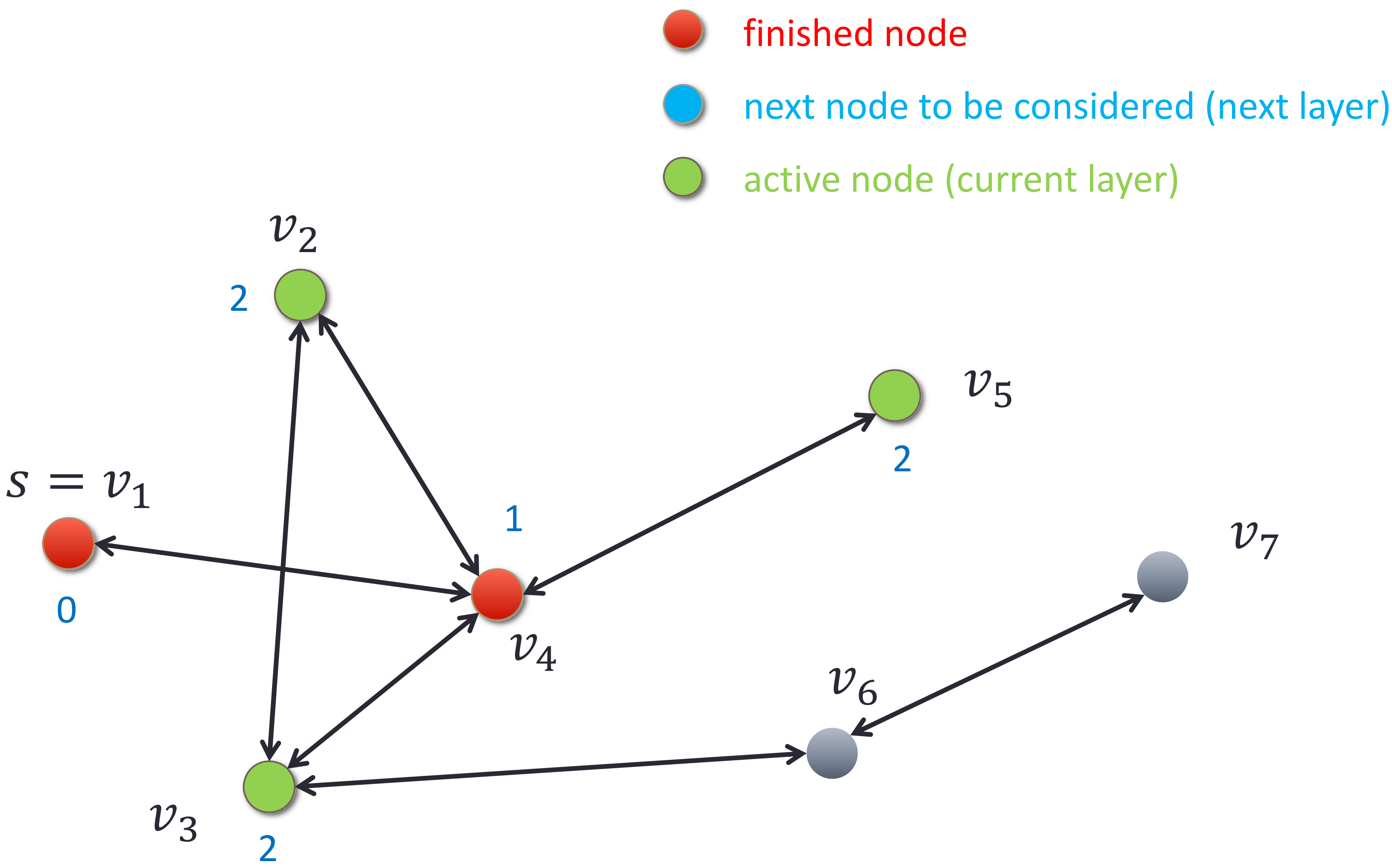
Breadth-First Search: BFS Tree



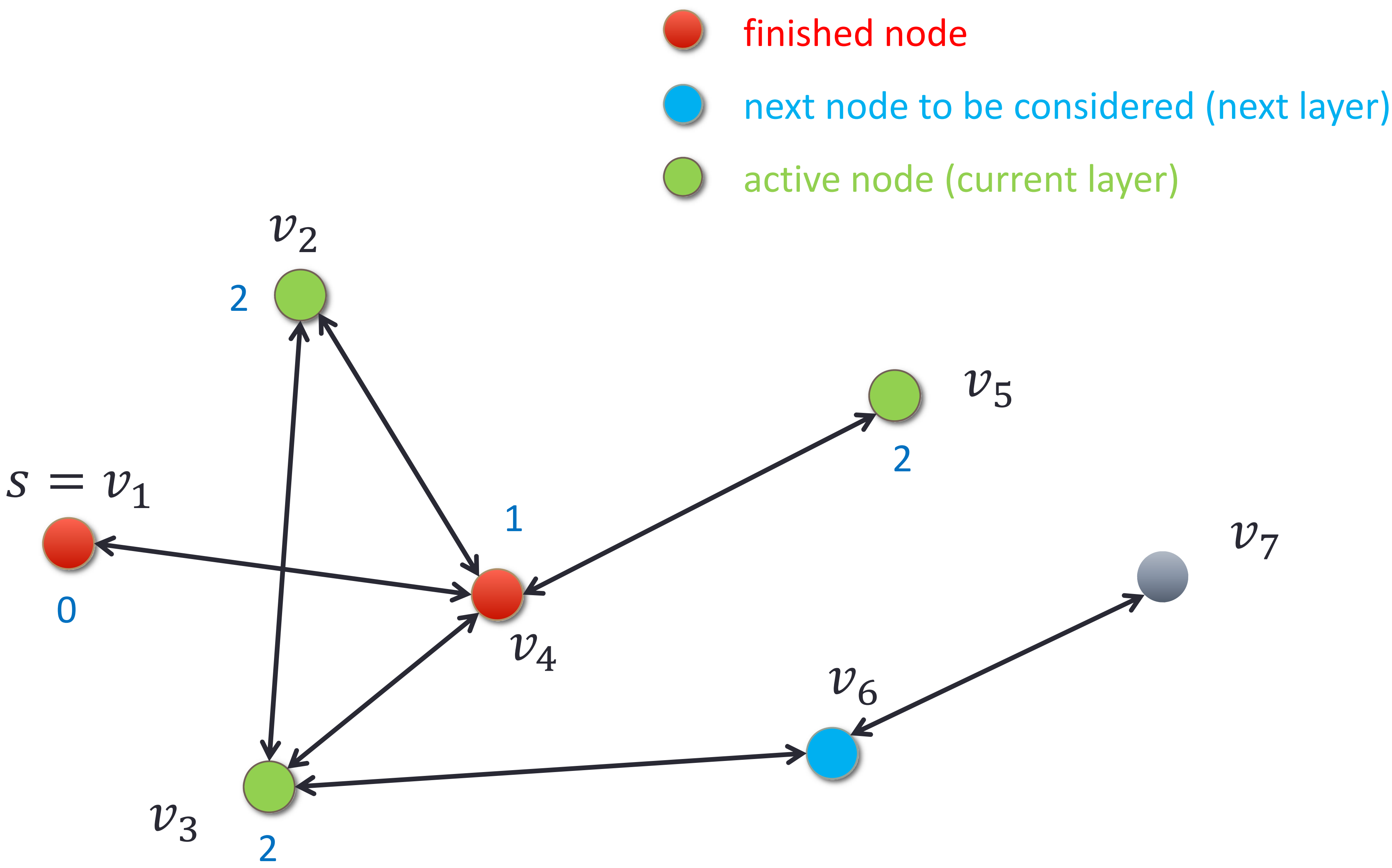
Breadth-First Search: BFS Tree



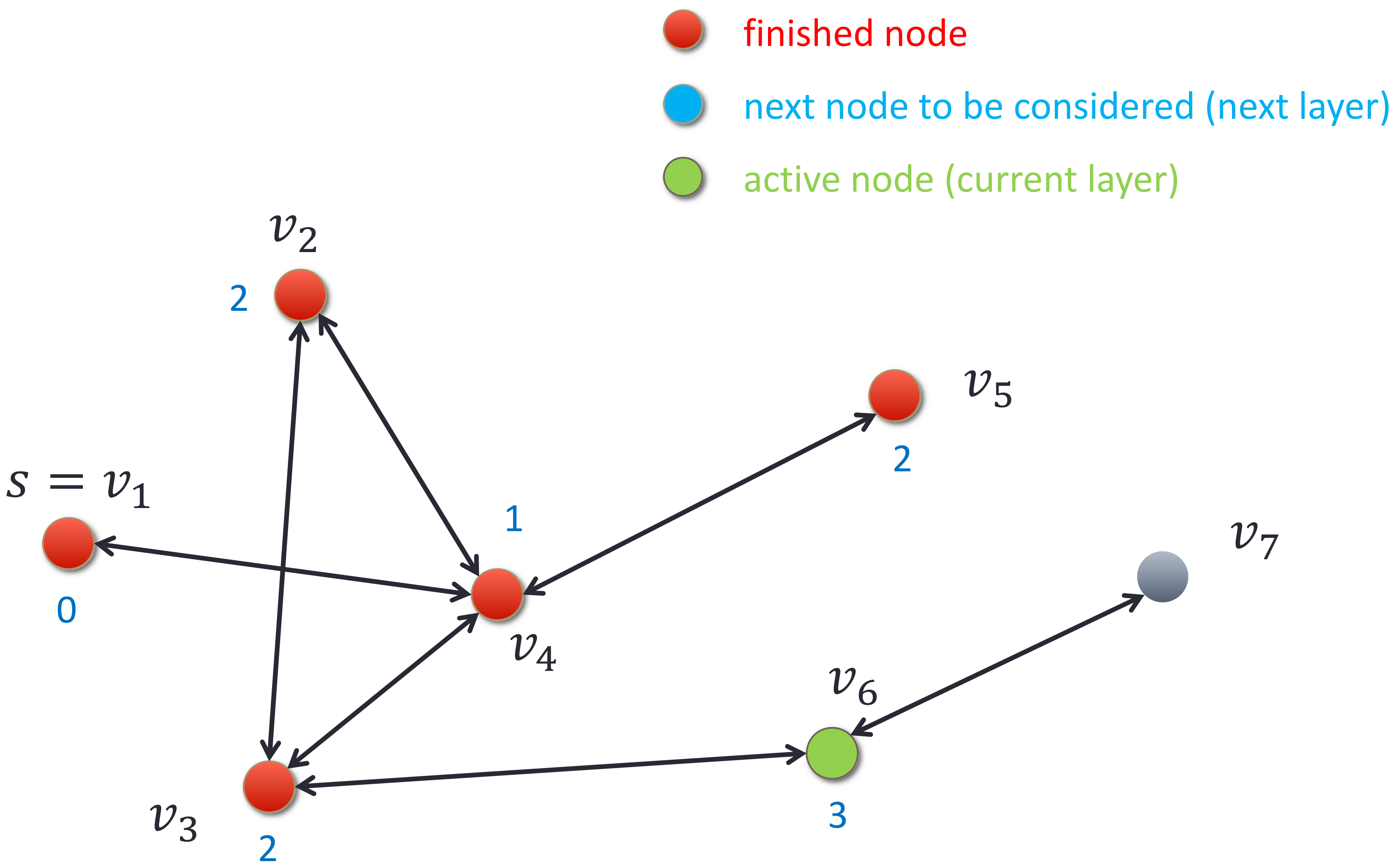
Breadth-First Search: BFS Tree



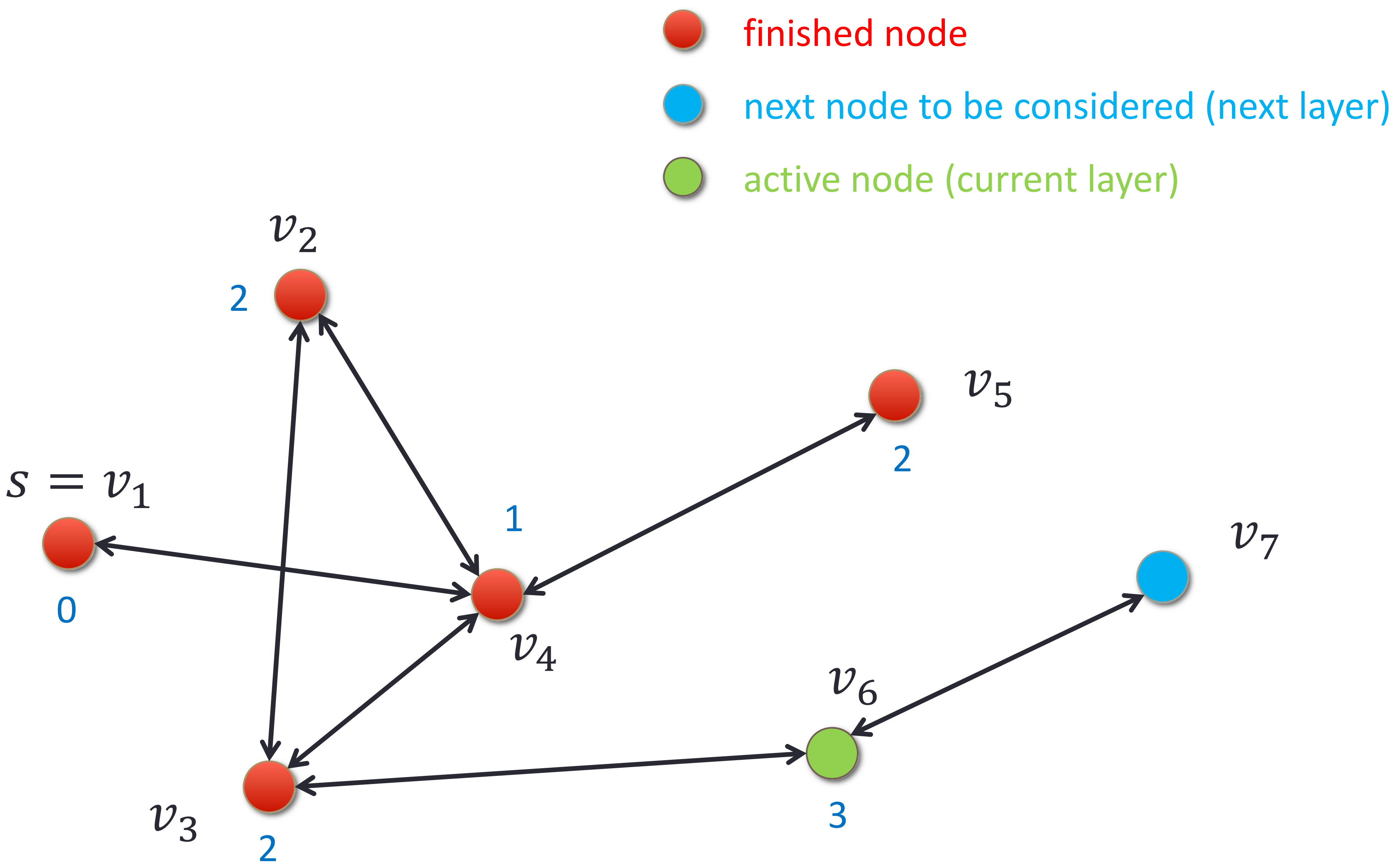
Breadth-First Search: BFS Tree



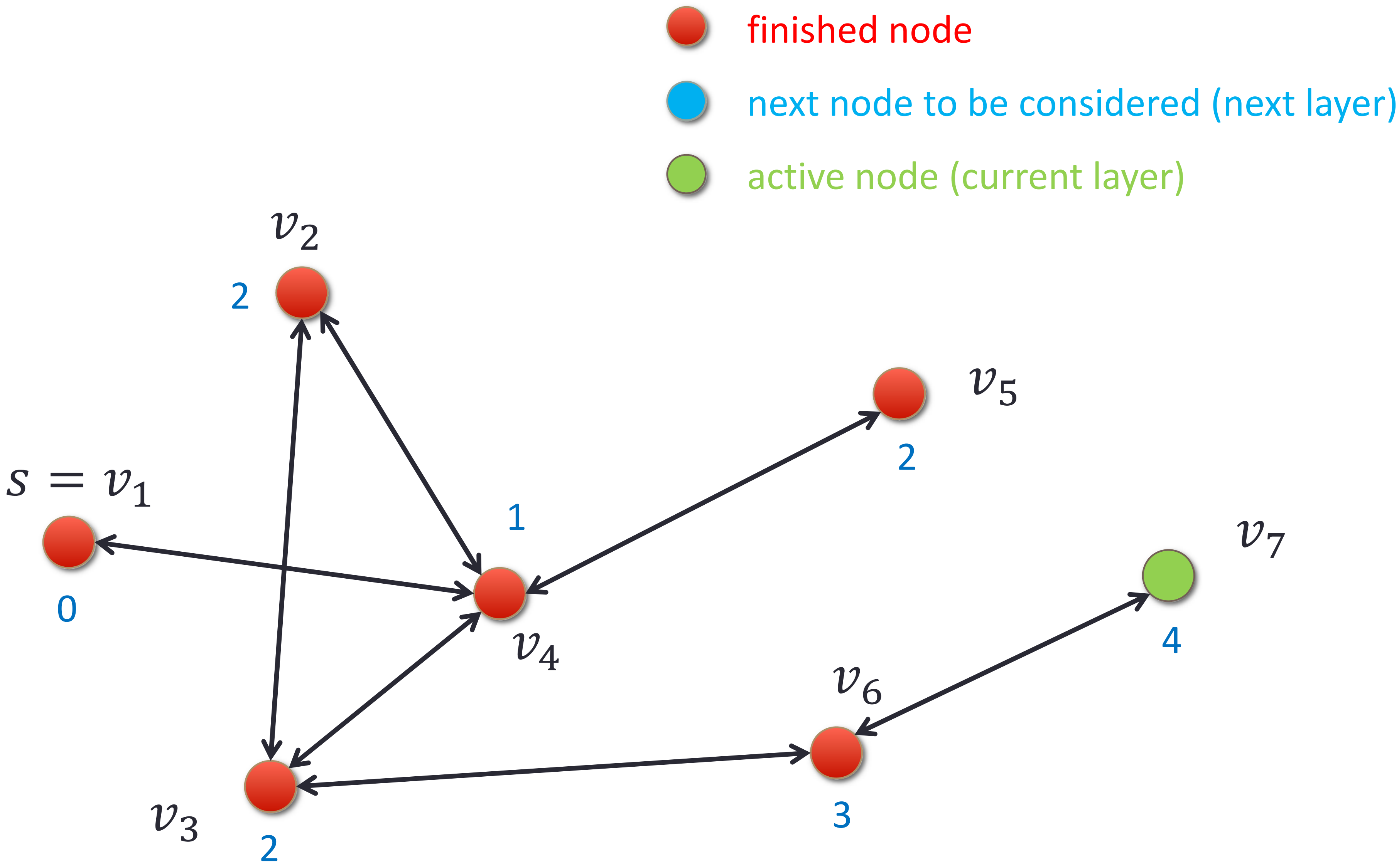
Breadth-First Search: BFS Tree



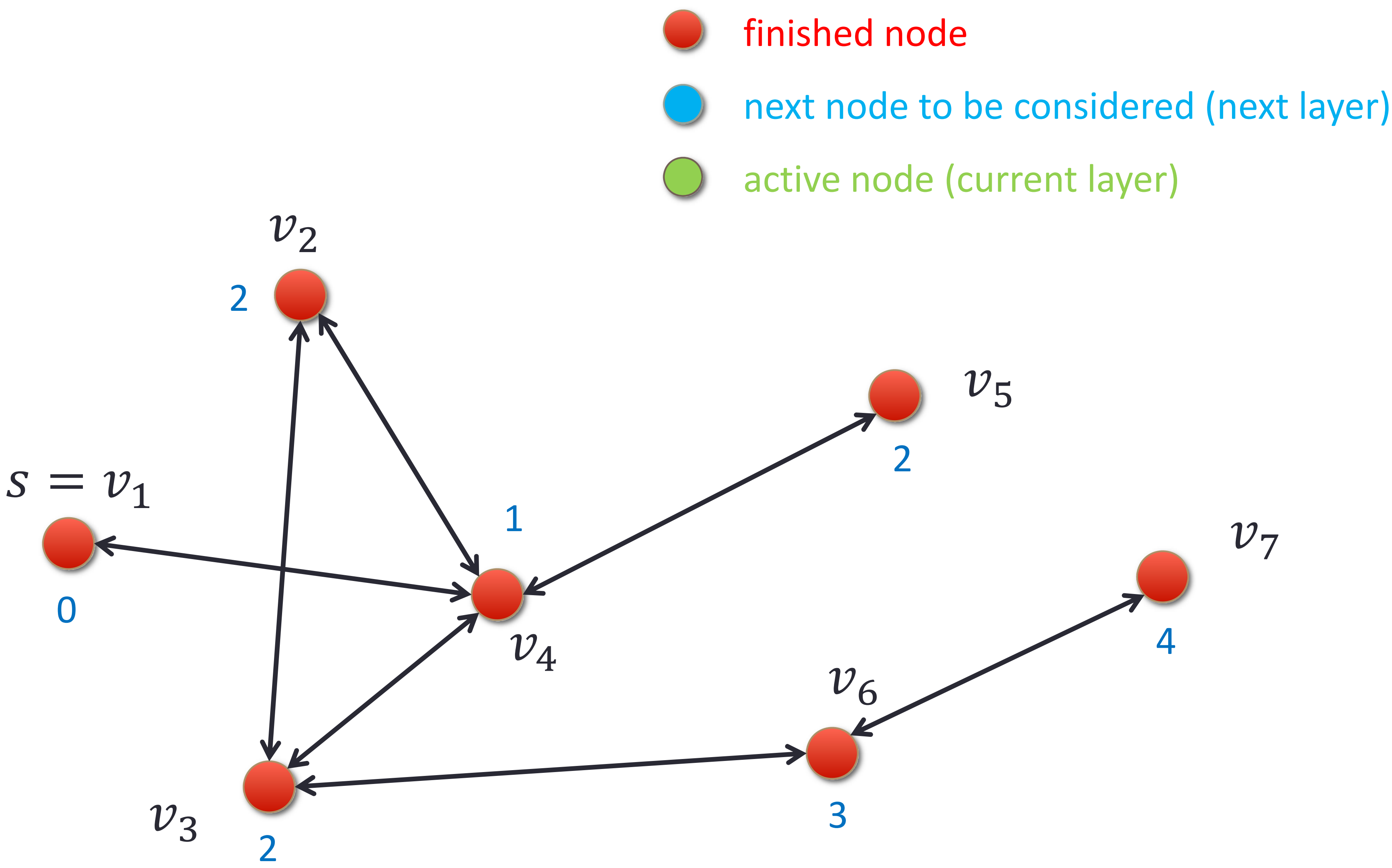
Breadth-First Search: BFS Tree



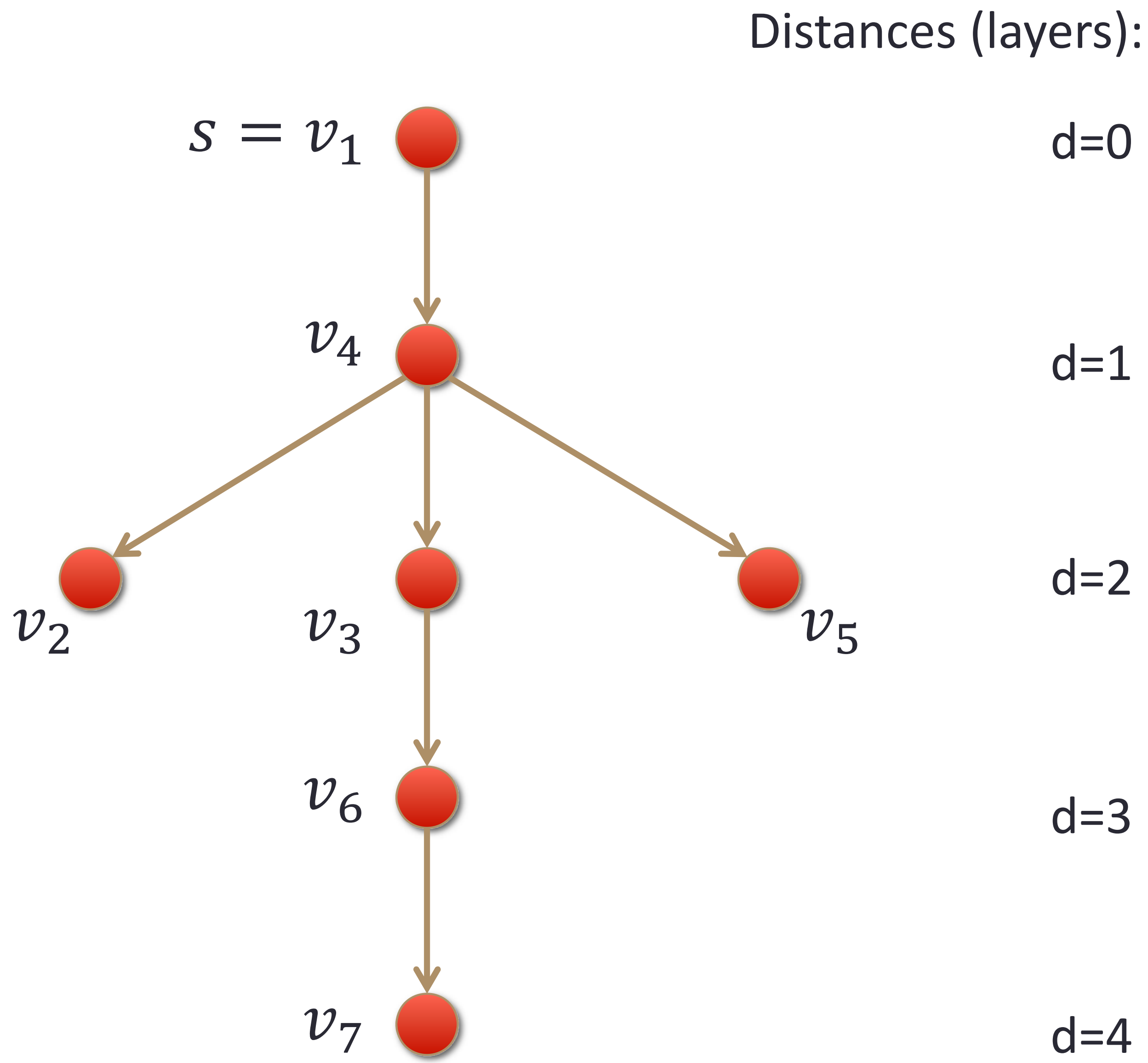
Breadth-First Search: BFS Tree



Breadth-First Search: BFS Tree



Breadth-First Search: BFS Tree



Breadth-First Search: Pseudocode

The algorithm uses "levels" L_i and a mechanism for setting and getting "labels" of vertices and edges.

Algorithm BFS(G, s):
  Input: A graph G and a vertex s of G
  Output: A labeling of the edges in the connected component of s as discovery edges and cross edges

  Create an empty list, L_0
  Mark s as explored and insert s into L_0
  $i \leftarrow 0$
  while L_i is not empty **do**
    create an empty list, L_{i+1}
    for each vertex, v, in L_i **do**
      for each edge, $e = (v, w)$, incident on v in G **do**
        if edge e is unexplored **then**
          if vertex w is unexplored **then**
            Label e as a discovery edge
            Mark w as explored and insert w into L_{i+1}
          else
            Label e as a cross edge
    $i \leftarrow i + 1$

Breadth-First Search: Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time.
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i.
- The method to determine incident edges is called once for each vertex. Note that $\sum_{v \in V} \deg(v) = 2m$ (sum of degrees).
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure.

* n is the number of vertices and m is the number of edges.

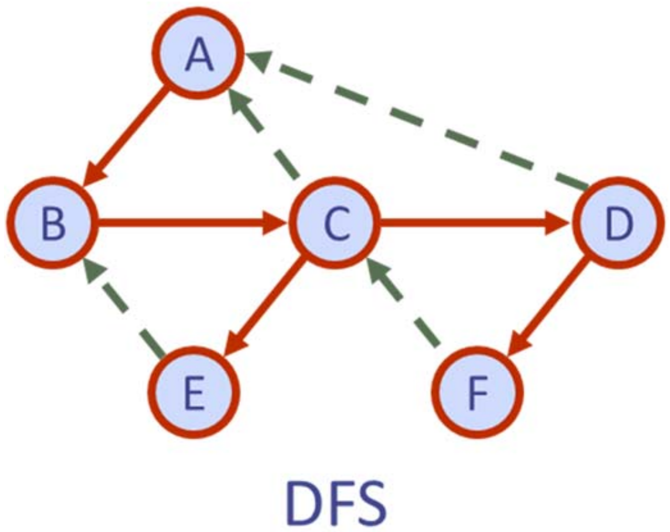
Breadth-First Search: Properties

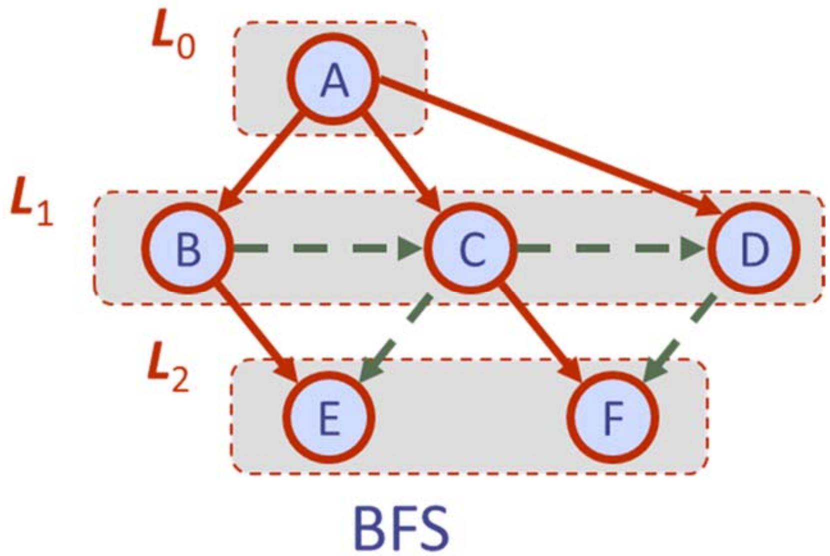
Property 1. $\mathrm{DFS}(G, v)$ visits all the vertices and edges of G_v, the connected component of v.

Property 2. The discovery edges labeled by $\mathrm{DFS}(G, v)$ form a spanning tree T_v of G_v.

- BFS on a graph with n vertices and m edges takes $\mathrm{O}(n + m)$ time.
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

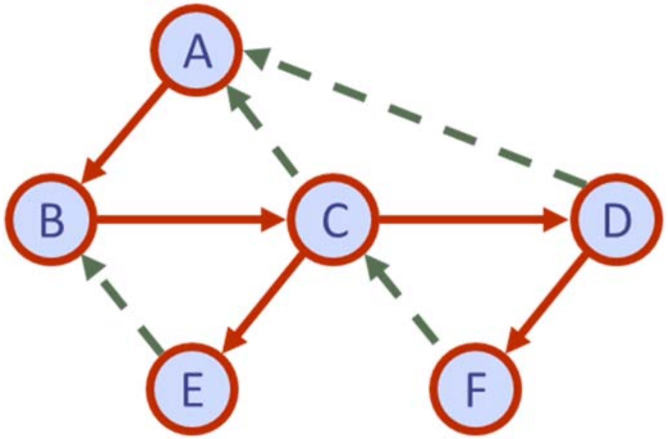
Depth-First Search vs. Breadth-First Search



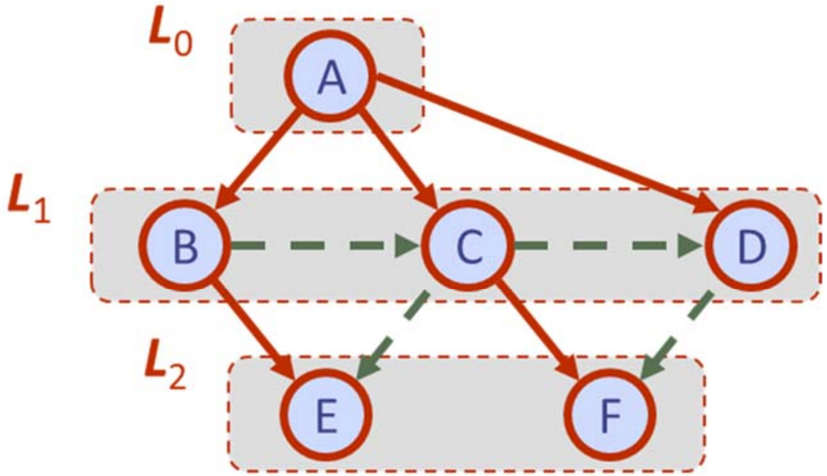


Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√

Depth-First Search vs. Breadth-First Search



DFS



BFS

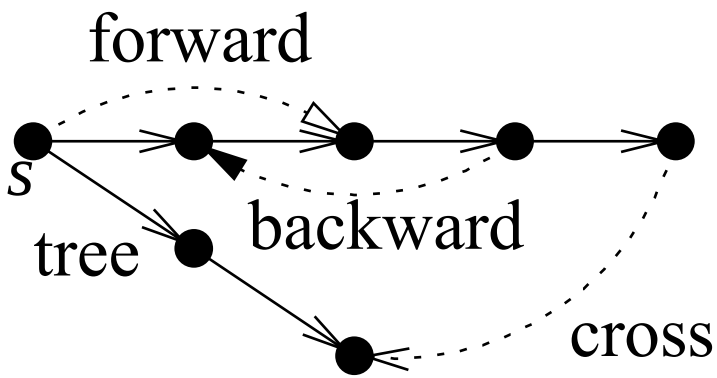
Back edge (v, w):
w is an ancestor of v in the tree of discovery edges

Cross edge (v, w):
w is in the same level as v or in the next level.

Graph Traversal: Types of Edges

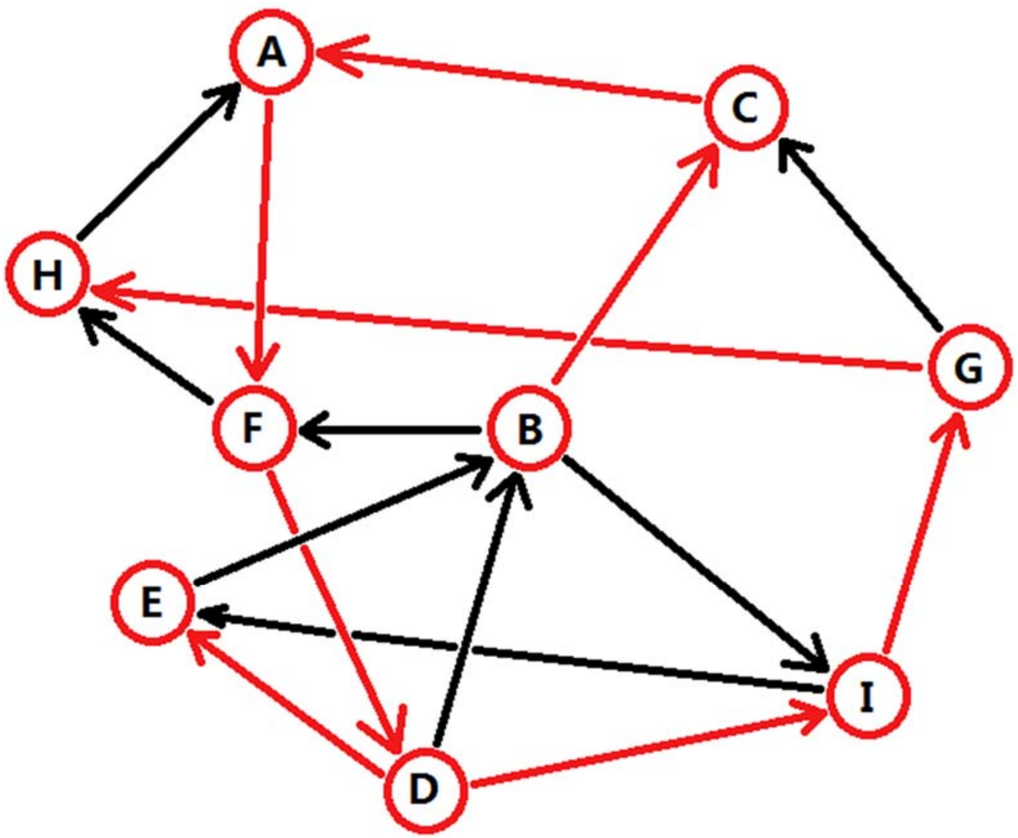
Considering a tree T of a given graph G, we can classify the edges into:

- Tree edges
- Forward edges
- Backward edges
- Cross edges

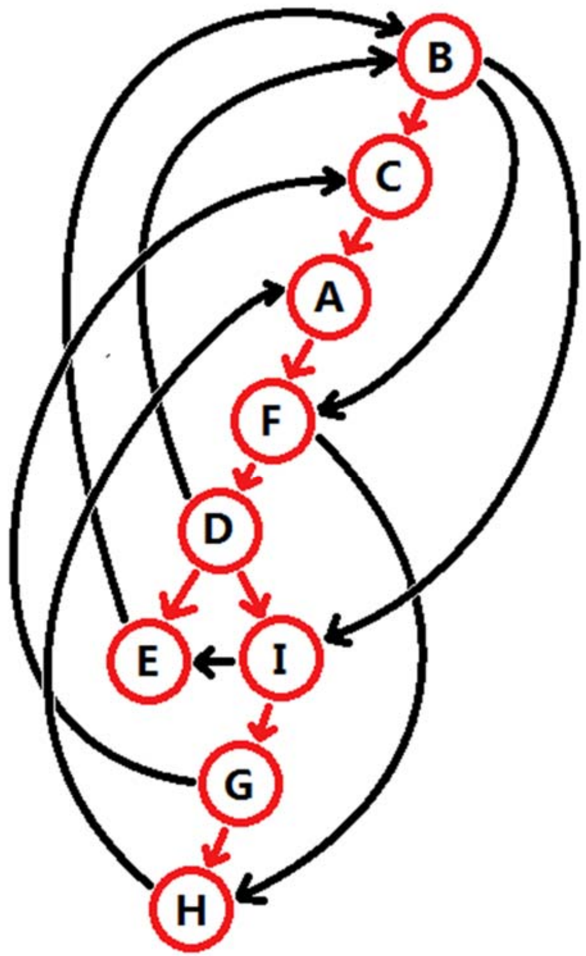


A directed graph does not contain a cycle if and only if the DFS run does not encounter a backward edge.

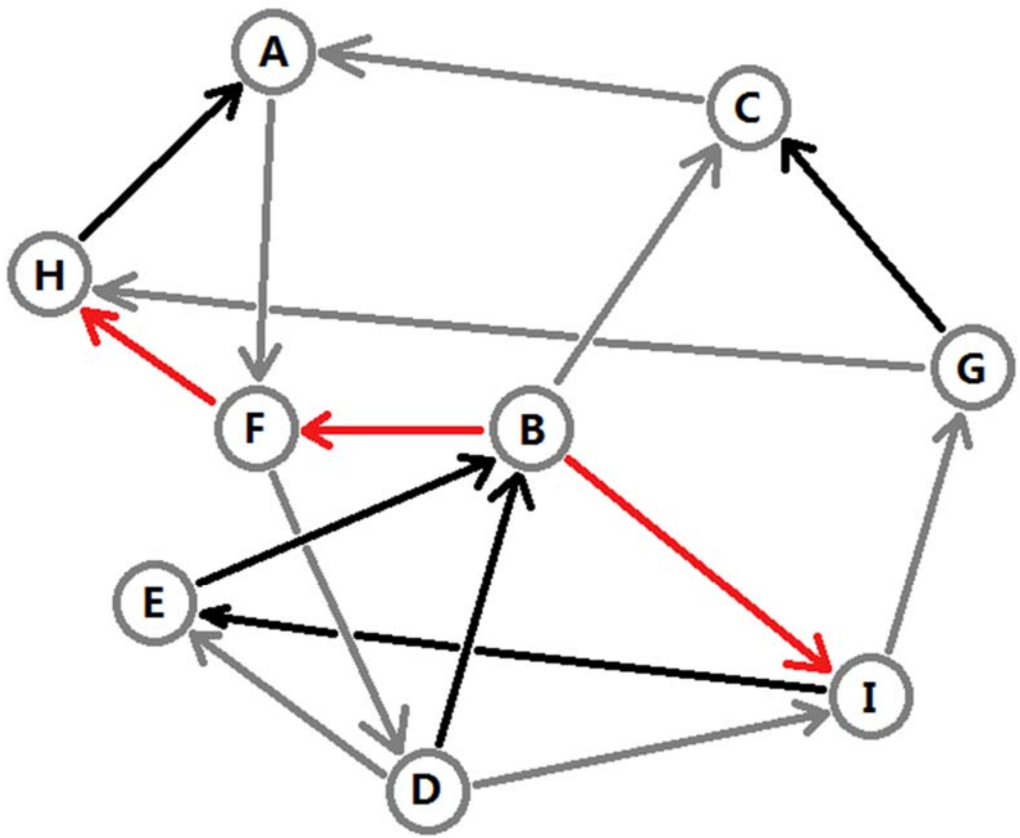
Graph Traversal: Tree Edges



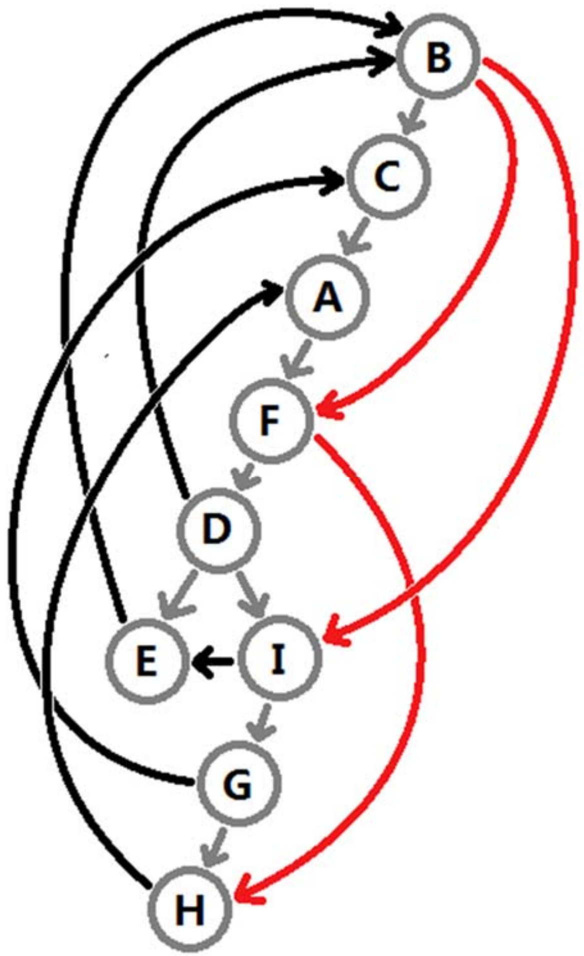
DFS Tree



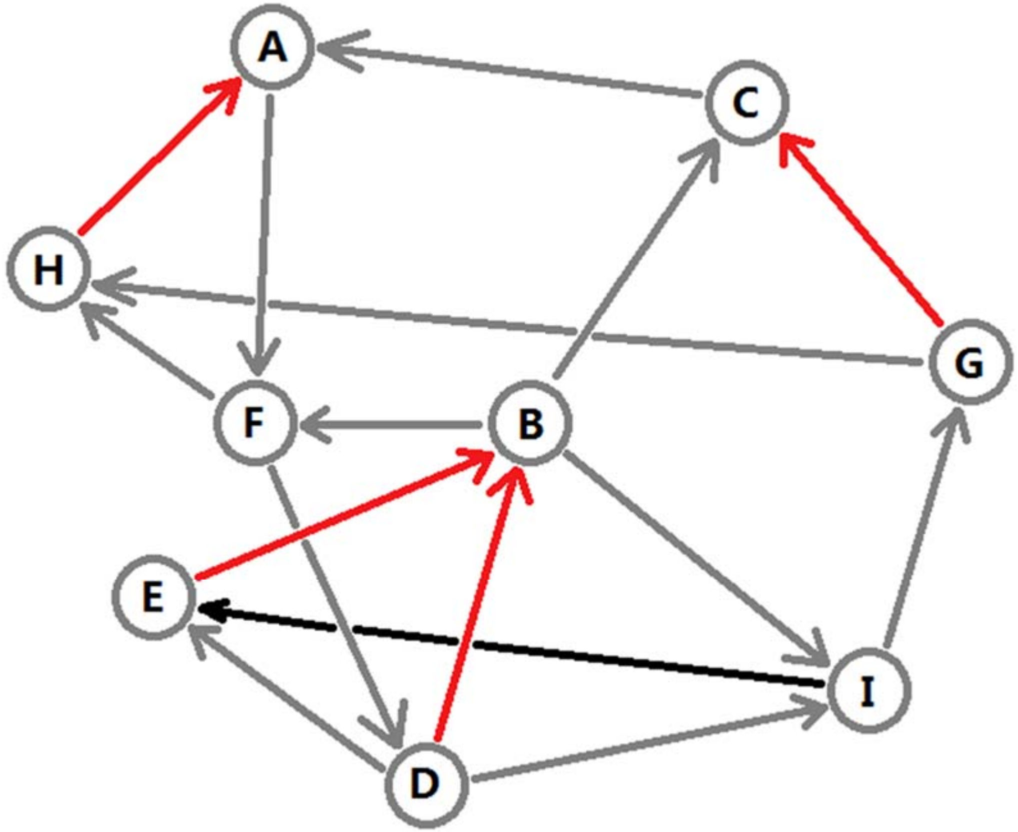
Graph Traversal: Forward Edges



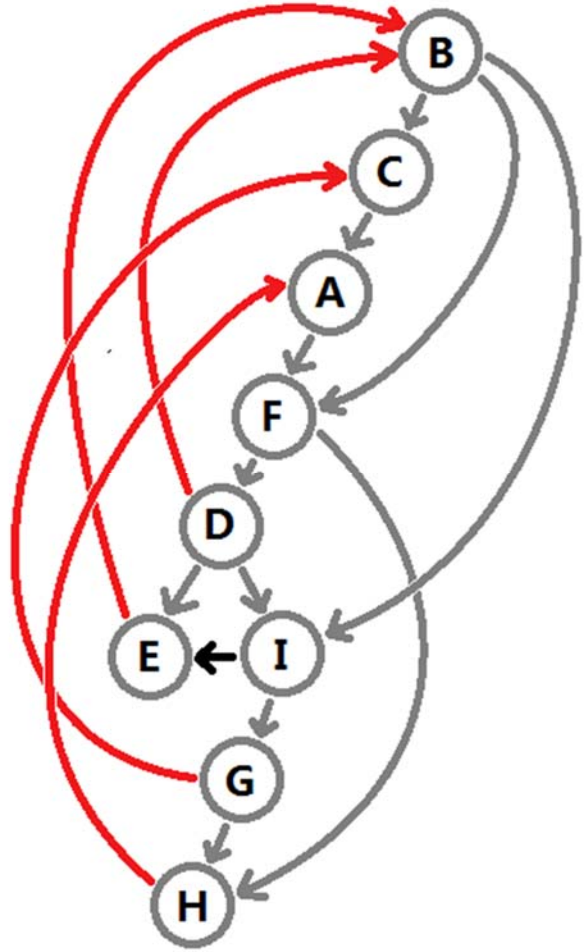
DFS Tree



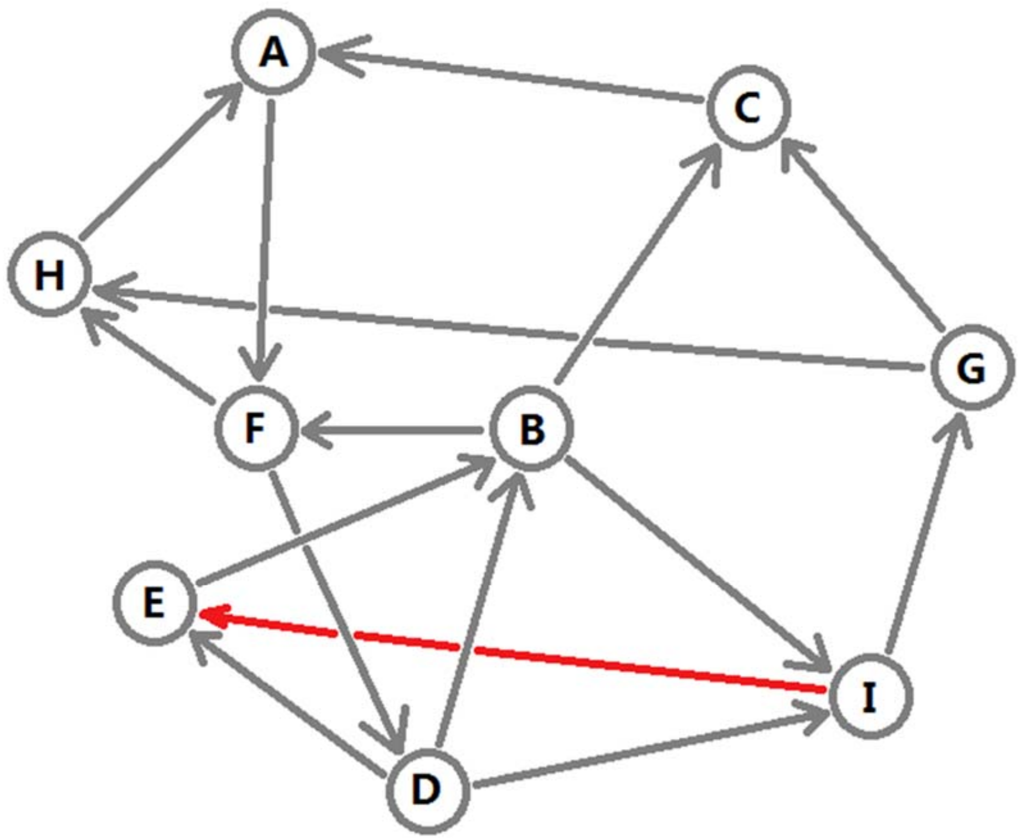
Graph Traversal: Backward Edges



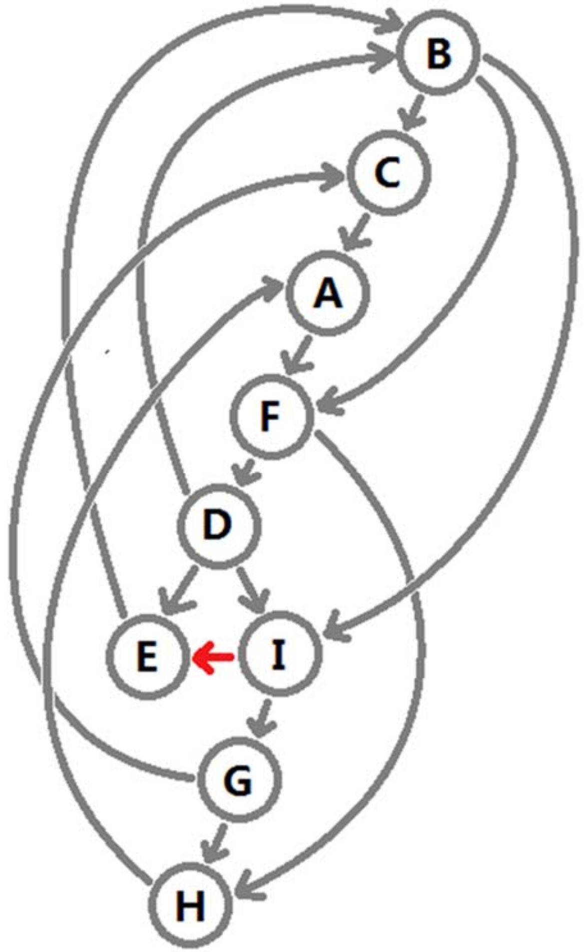
DFS Tree



Graph Traversal: Cross Edges



DFS Tree



Other references and things to do

- Read chapter 14.3 in Data Structures and Algorithms in Java. Michael T. Goodrich, Irvine Roberto Tamassia, and Michael H. Goldwasser, 2014.