CSD2183:
Data Structures

Elementary Graph Algorithms

Bingjie Xu

Contact: bingjie.xu@singaporetech.edu.sg

23 February 2026

Before we start…

- Dr. Bingjie Xu, Assistant Professor, ICT.
- Practical (lab) and interactive (app), with deep-dive learning resources (NotebookLM).
- I expect you to be punctual, stay engaged, ask questions, and create respectful online discussion environment.

Information

- The mid-term exam grades will be released by end of week 9.
- Homework 2 will be released in week 9-10.

Overview

	Basics (Week 1 - 6)	**Advanced (Week 8 - 13)**
1	Foundations	Graph Foundations and Traversal
2	Running Times	Minimum Spanning Trees
3	Sorting	Shortest Paths
4	List and Hash Tables	Dynamic Programming
5	Trees	Greedy Algorithms
6	Consultation	Consultation

Cormen, Thomas H., et al. Introduction to algorithms 4th edition. MIT press, 2022.

Learning Objectives

By end of this lecture, you will be able to:

- Name real-world examples of graphs in different domains, e.g., transportation, social networks, knowledge graph.
- Recall important **graph terminologies**, e.g., vertex, edge, directedness, weight, and path.
- Construct **graph representations**, e.g., adjacency-list, adjacency-matrix.
- Implement and analyze **graph traversal algorithms**, e.g., BFS, DFS.

Graph Foundations

- Real-world applications
- Definition
- Representations

Real-World Applications

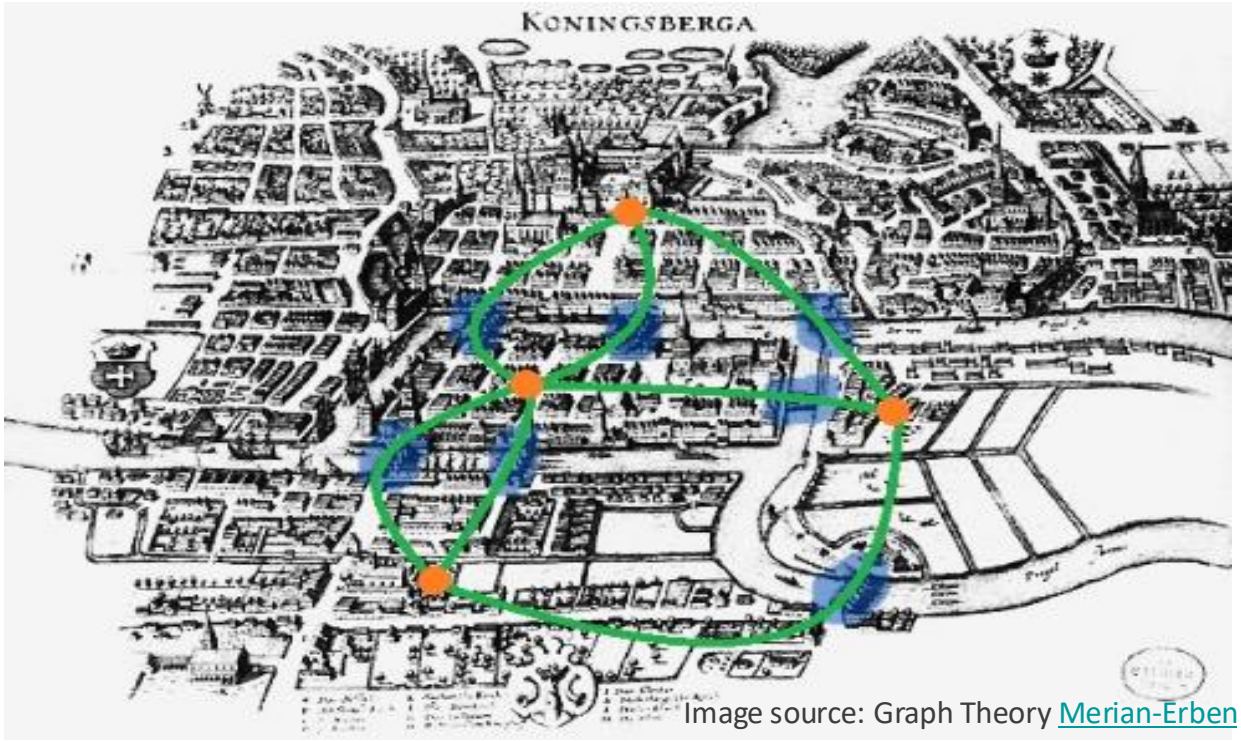


Image source: Graph Theory Merian-Erben

Origin of Graph Theory: 7 Bridges of Königsberg

- Map routes
- Neural networks
- Social networks
- Cube solution states
- …

Real-World Applications

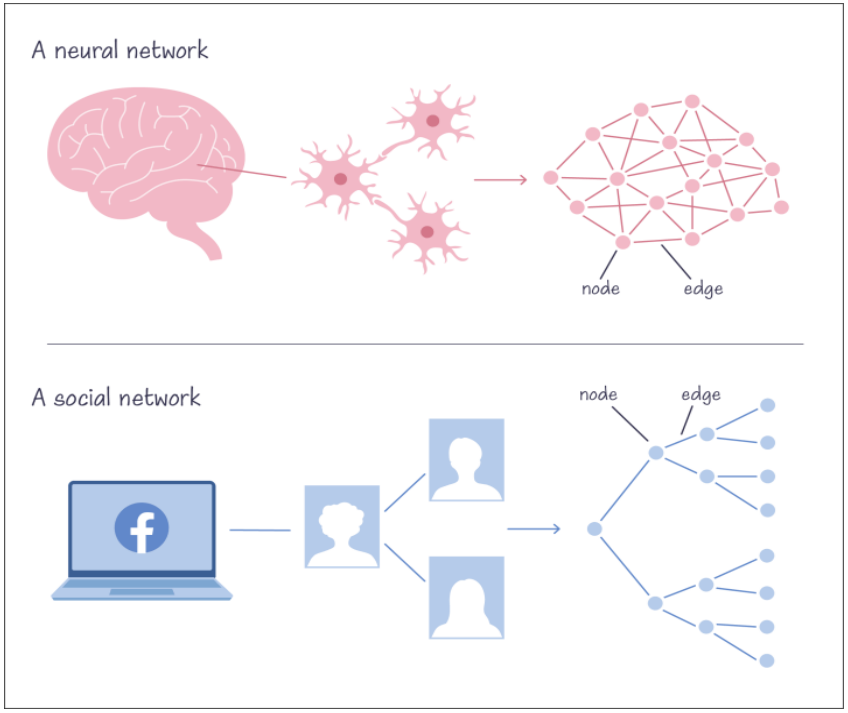
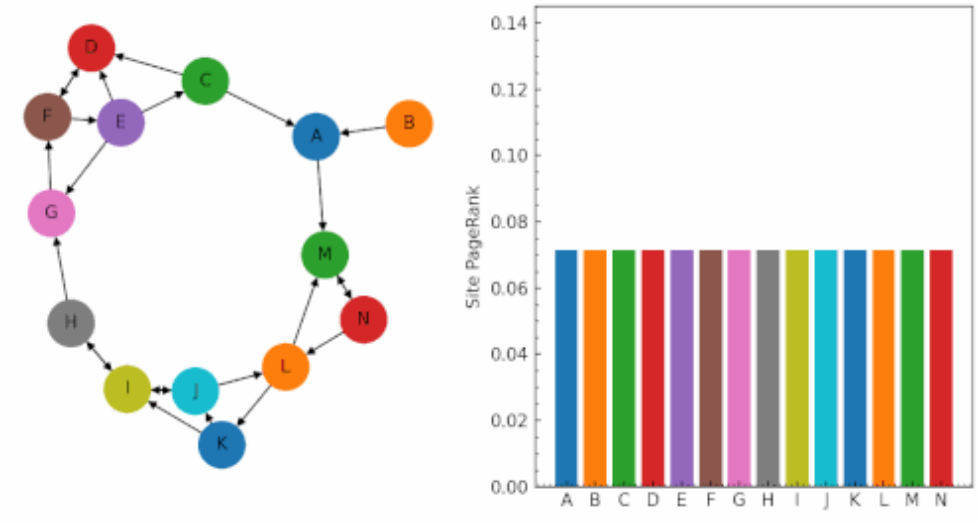


Image source: Graph Theory 101



Google Search Engine PageRank: wiki

Real-World Applications

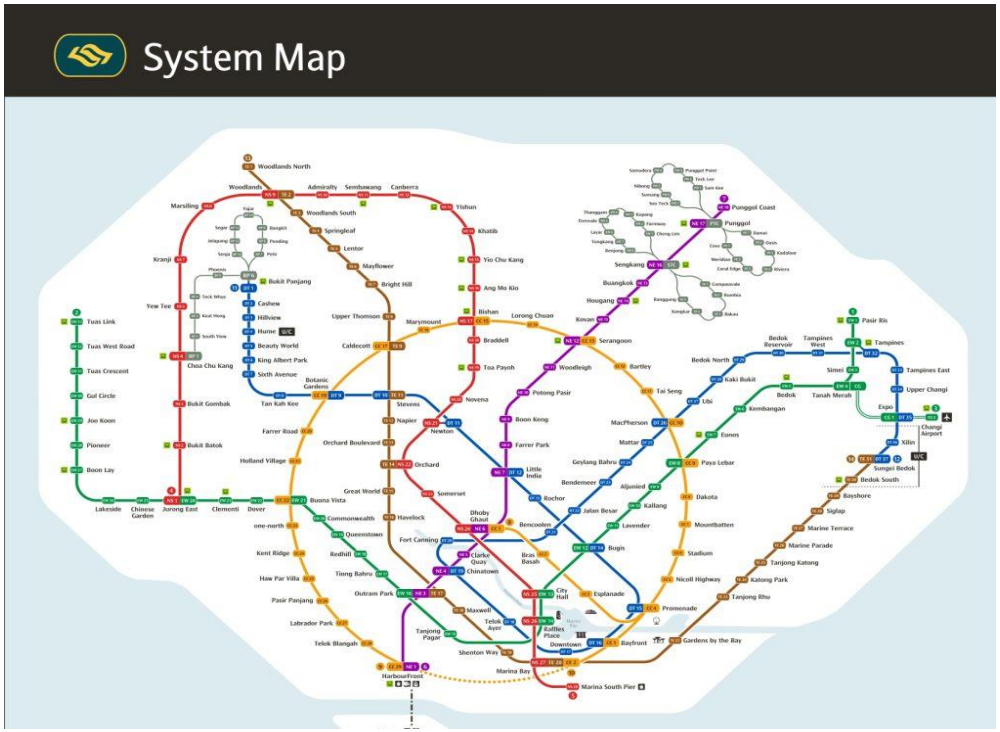


Image source: Land Transport Guru

[Singapore Airlines Route Map]

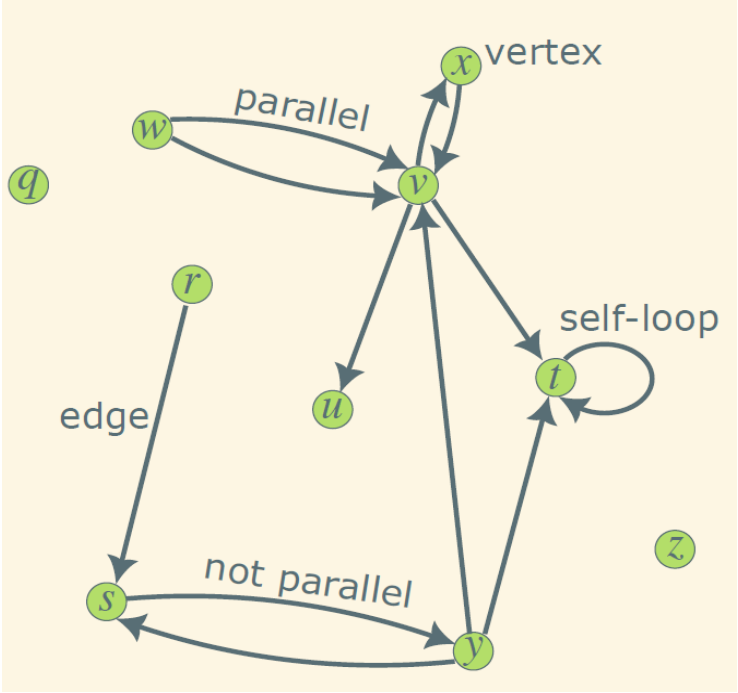
Graph Definition

- A graph $G = (V, E)$ of two sets:

  V : Set of **vertices** (singular: vertex), also called "nodes".

  E : Set of **edges**, also called "links".

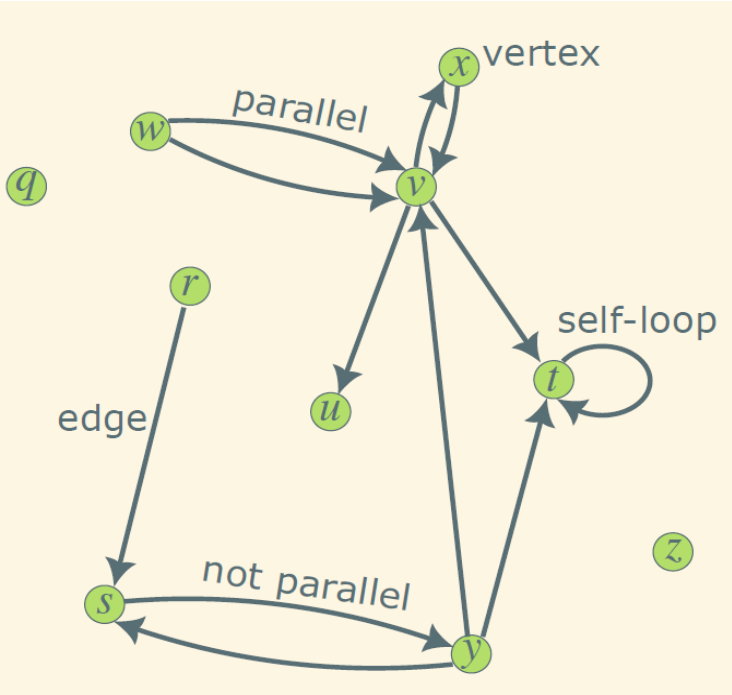
- To emphasize the attributes of G, we sometimes write $G.V$ and $G.E$.



Graph Definition

- In a **directed graph**, also called a "digraph", the edges are ordered pairs of vertices. That is, edge $(r, s) \in E$ does not necessarily imply edge $(s, r) \in E$. In plots, the order is usually indicated by an arrow from r to s.

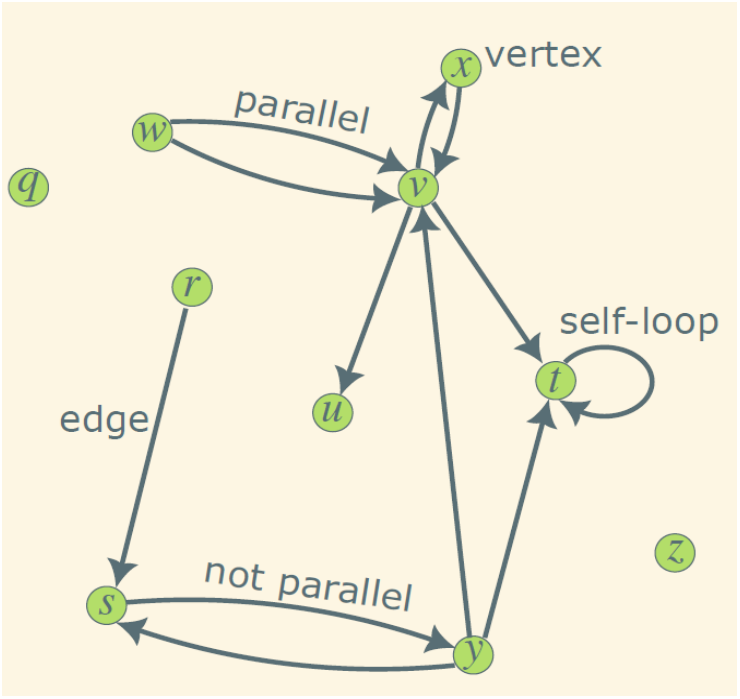
- In an **undirected graph**, the edges are unordered pairs of vertices. That is, $(r, s) = (s, r)$.



Graph Definition

- A **weighted graph** is a graph in which each edge has an associated numerical value, called a **weight**. These weights often represent costs, distances, or capacities depending on the context.

- An **unweighted graph** is a graph in which all edges are treated as equal, i.e., they have no associated weights. Each edge simply represents a connection.

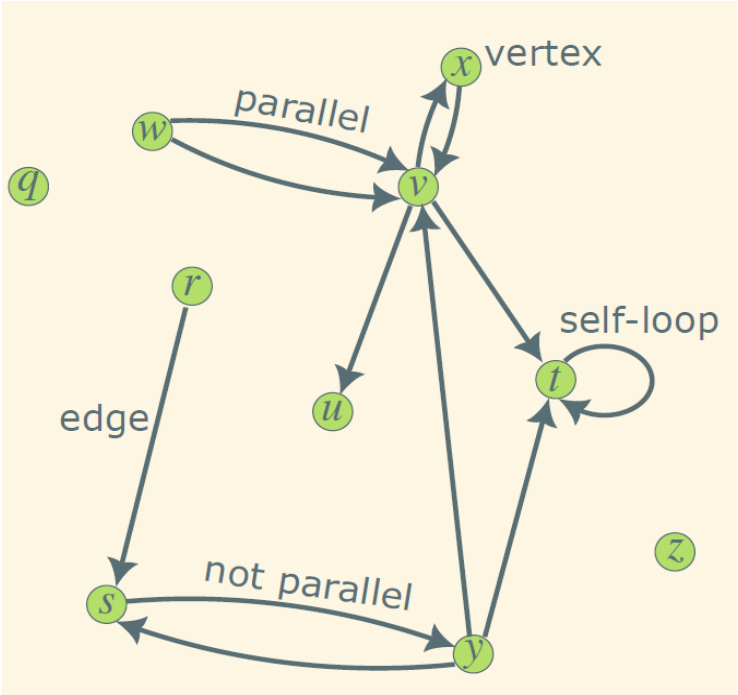


The weights in a bi-directed graph do not need to be the same.

Graph Definition

A graph is **simple** if it does not contain parallel edges or self-loops.

In this course, we only work with simple graphs. For brevity, the adjective “simple” will be omitted but is always implicit unless otherwise stated.

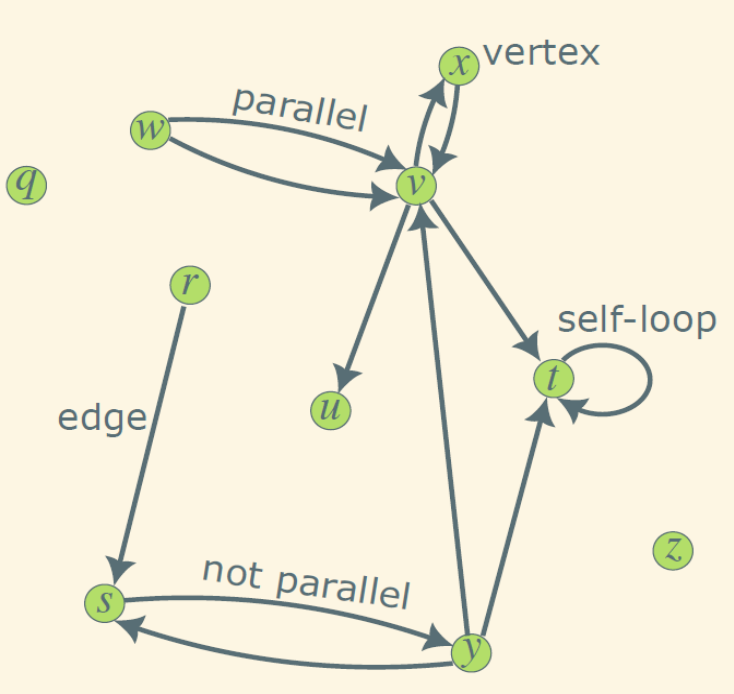


Graph Definition

Let k_v be the number of vertices pointing away from v.

In undirected graphs, k_v is called the **degree** of v.

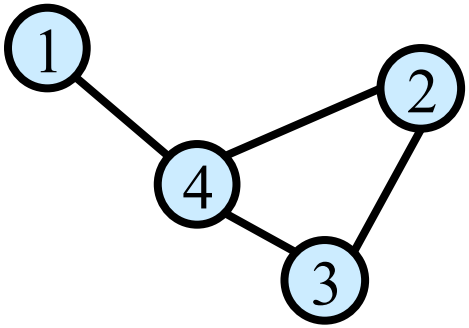
In directed graphs, k_v is called the **out-degree** of v.



Graph Representations

Adjacency List: Array Adj of $|V|$ lists, one per vertex.

- Vertex u's list has all vertices v such that $(u, v) \in E$. Works for both directed and undirected graphs.
- The order of the elements in $Adj[u]$ does not matter.
- If edges have weights, store w in the lists.
- Space: $\Theta(V + E)$
- Time to list all vertices adjacent to u: $\Theta(degree(u))$
- Time to determine whether $(u, v) \in E$: $O(degree(u))$

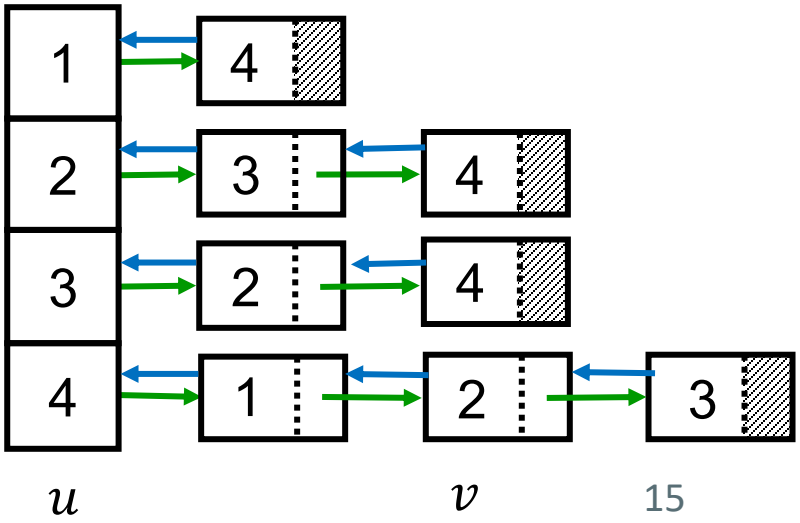


Advantage

- Compact for sparse graphs

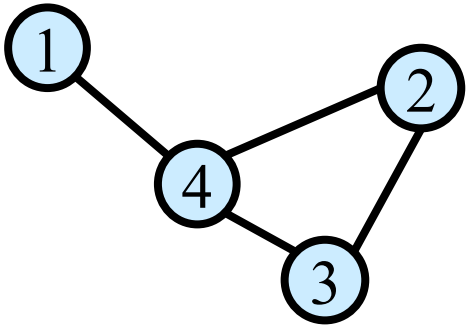
Disadvantage

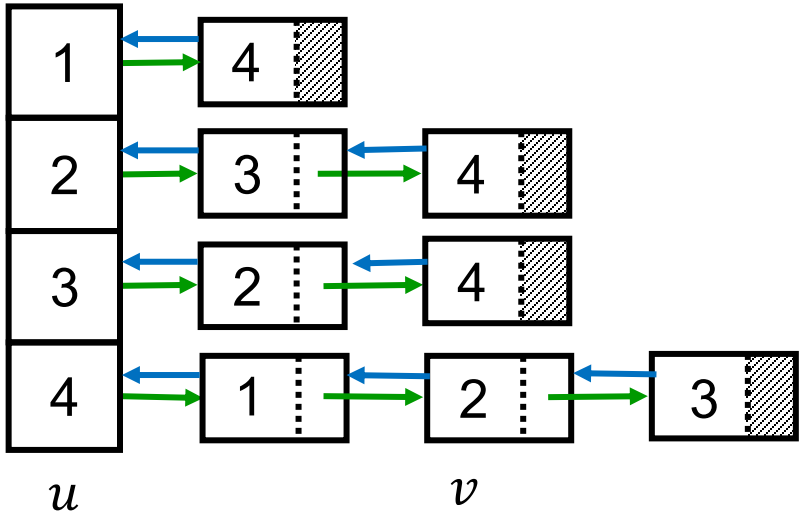
- Bad memory access if graph is dense



Graph Representations

Adjacency List: Array Adj of $|V|$ lists, one per vertex.



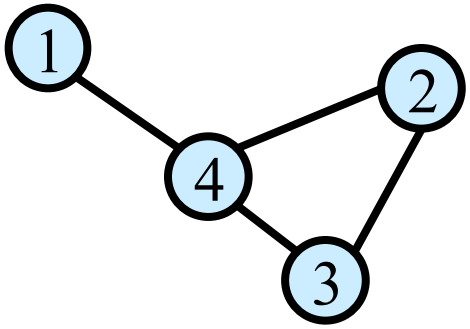



```
// Construct a graph
int n = 4; // Number of nodes
vector<vector<int>> adj(n);

// Add adjacencies
adj[1] = {4};
adj[2] = {3, 4};
adj[3] = {2, 4};
adj[4] = {1, 2, 3};
```

Graph Representations

Adjacency Matrix: $|V| * |V|$ matrix A.

- For all vertex $i, j, A[i, j] = 1$ if $(i, j) \in E$, 0 otherwise.
- Works for both directed and undirected graphs. Can also represent weighted graph, $A[i, j] = w_{ij}$
- Space: $\Theta(V^2)$
- Time to list all vertices adjacent to u: $\Theta(V)$
- Time to determine whether $(u, v) \in E$: $\Theta(1)$



Advantage:

- When graphs are reasonably small

Disadvantage:

- Inefficient for sparse graphs in storage

	1	2	3	4
1	0	0	0	1
2	0	0	1	1
3	0	1	0	1
4	1	1	1	0

Graph Representations

Adjacency Matrix: $|V| * |V|$ matrix A.

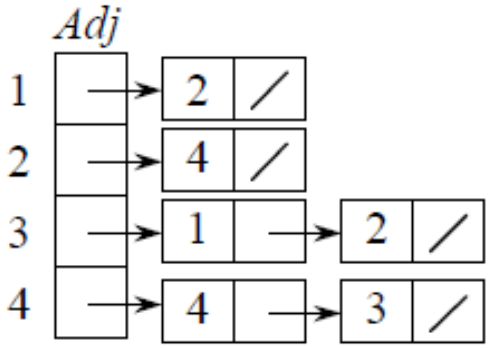
	1	2	3	4
1	0	0	0	1
2	0	0	1	1
3	0	1	0	1
4	1	1	1	0

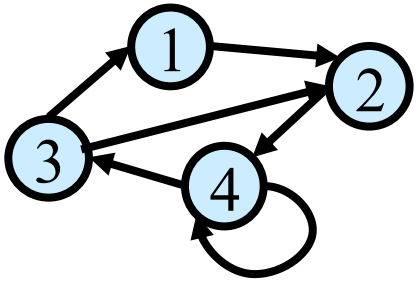
```
int numVertices;
vector<vector<int>> adjMatrix;

adjMatrix = vector<vector<int>>(numVertices, vector<int>(numVertices, 0));

// Add an edge between vertices u and v with weight (default 1)
void addEdge(int u, int v, int weight = 1) {
        adjMatrix[u][v] = weight;
        adjMatrix[v][u] = weight;
    }

// Remove the edge between vertices u and v
void removeEdge(int u, int v) {
        adjMatrix[u][v] = 0;
        adjMatrix[v][u] = 0;
    }
```

Exercise

For a directed graph, same asymptotic space and time as for an undirected graph using Adjacency List.

Can you list the edges (u, v) and draw the directed graph based on below adjacency list?





Graph Traversal Algorithms

- Breadth-first search (BFS)

Graph Traversal Algorithms

- A **path of length k** (k edges) from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $< v_0, v_1, \ldots, v_k >$ of vertices such that all of the following conditions are satisfied:

$$u = v_0$$

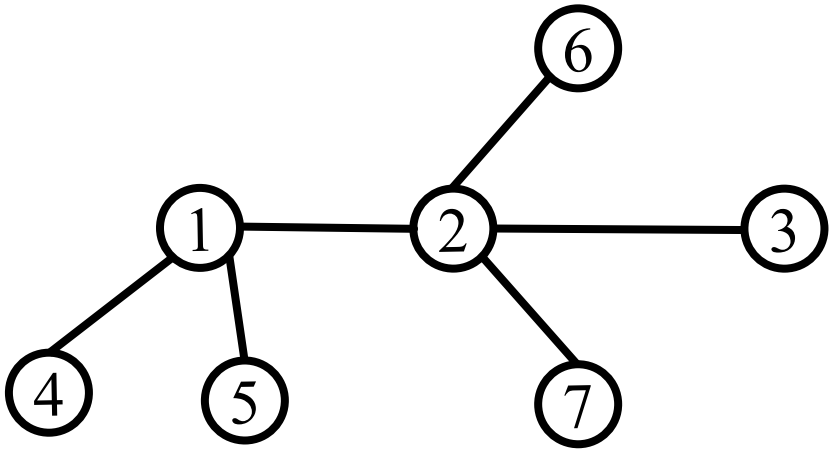
$$u' = v_k$$

$$(v_{i-1}, v_i) \in E, \ \forall i \in \{1, 2, \ldots, k\}$$

- Graph G can be directed and undirected.
- If there is a path p from u to u' , we say that **u' is reachable from u** via p.
- A path is simple if all vertices in the path are distinct.
- **Traversal**: Walk (via edges) from a fixed starting vertex u to all vertices reachable from u.

Graph Traversal Algorithms

Start from a tree (a simplified form of graph)



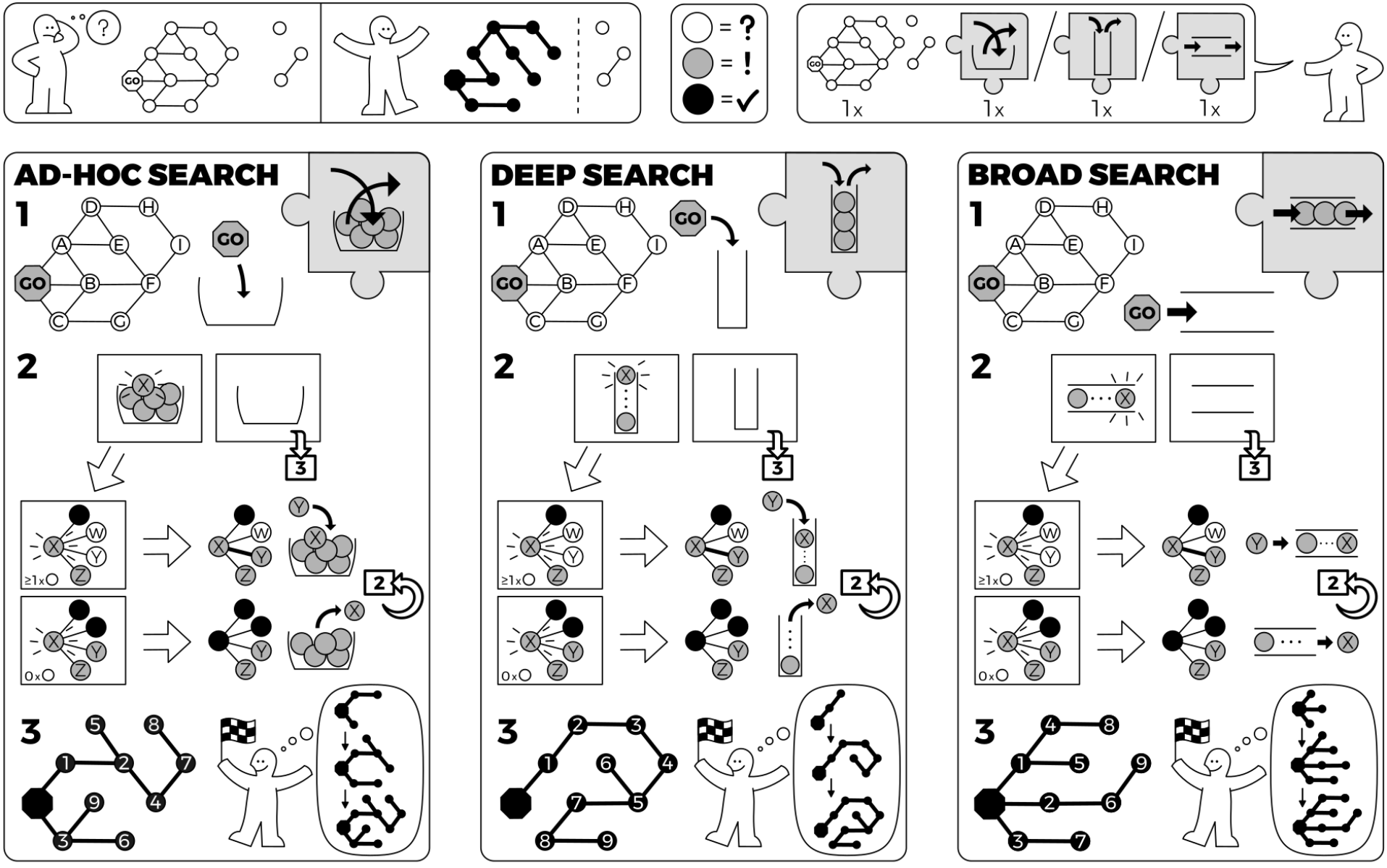
Two important terms:

1. Discovering/visiting a vertex
2. Exploration of vertex

Common rules:

- Start from **any** vertex.
- Order to choose adjacent vertices does **not** matter.
- Works for both **undirected and directed** graphs.

Breadth-first (broad) search order:
1,2,4,5,6,3,7
Depth-first (deep) search order:
1,2,3,6,7,4,5

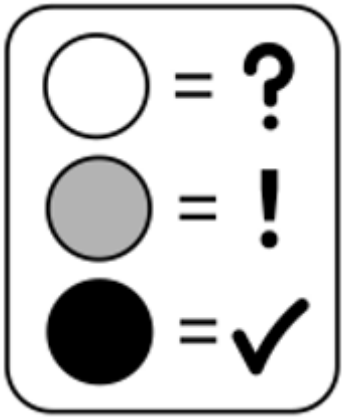

AD-HOC SEARCH
DEEP SEARCH
BROAD SEARCH
GO
= ?
= !
= ✓
1x

Breadth First Search (BFS)

Completely **explore** the vertices in order of their distance from s.

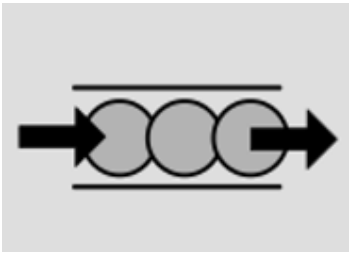
Three states of vertices:

- Undiscovered
- Discovered
- Fully-explored

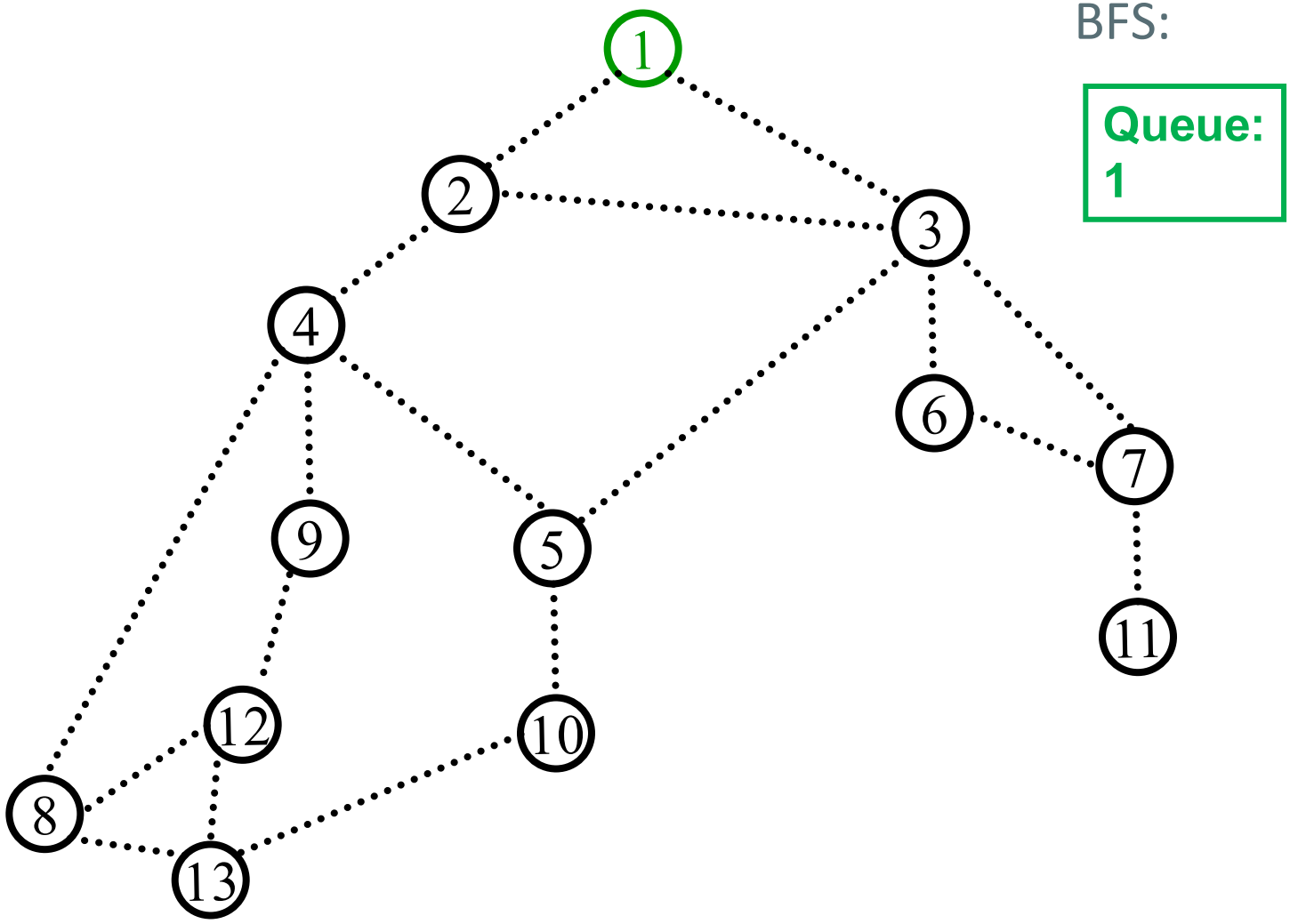


Naturally implemented using a **queue.**

The queue will always have the list of Discovered (visited) vertices.



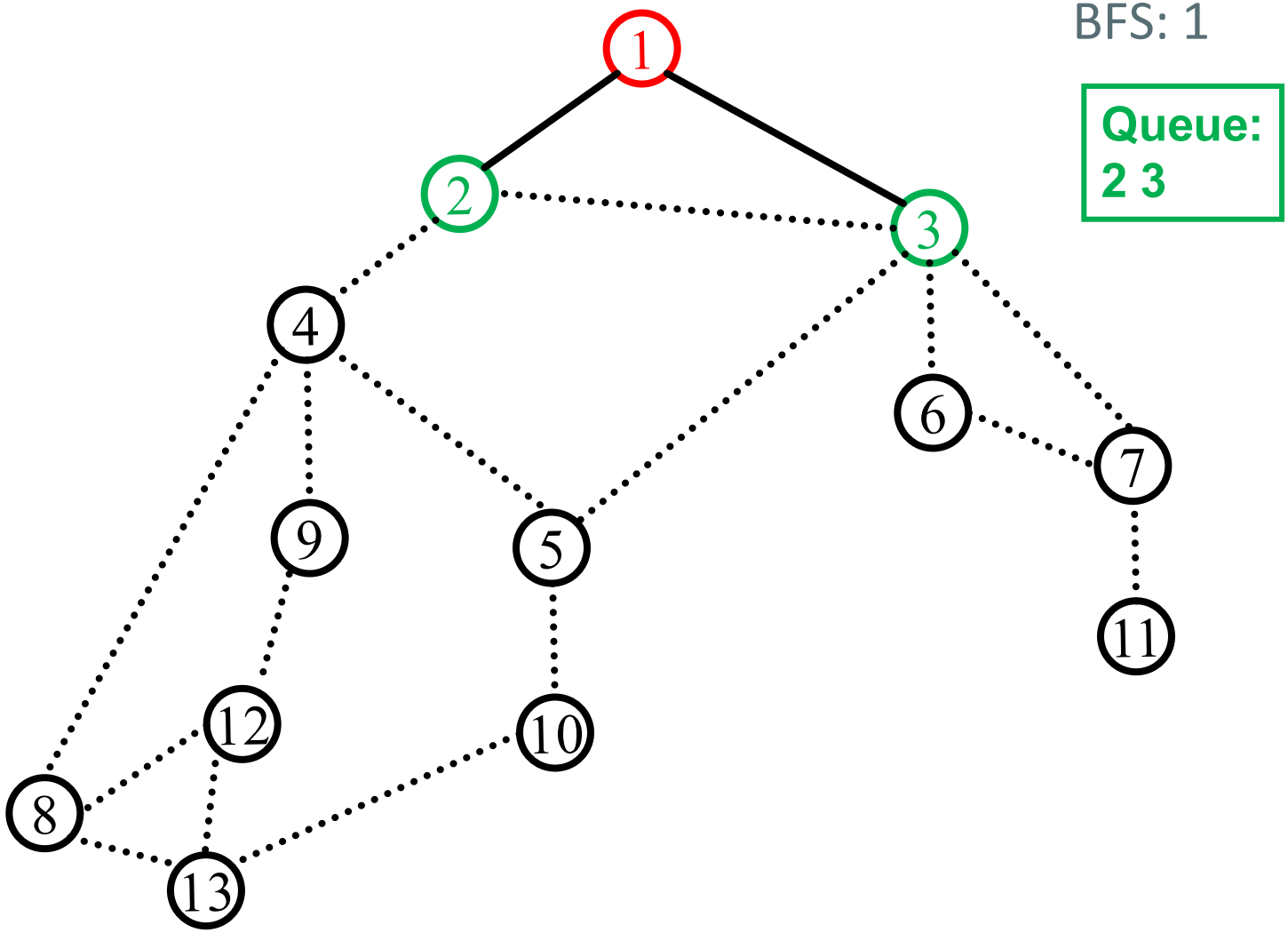
Breadth First Search (BFS)



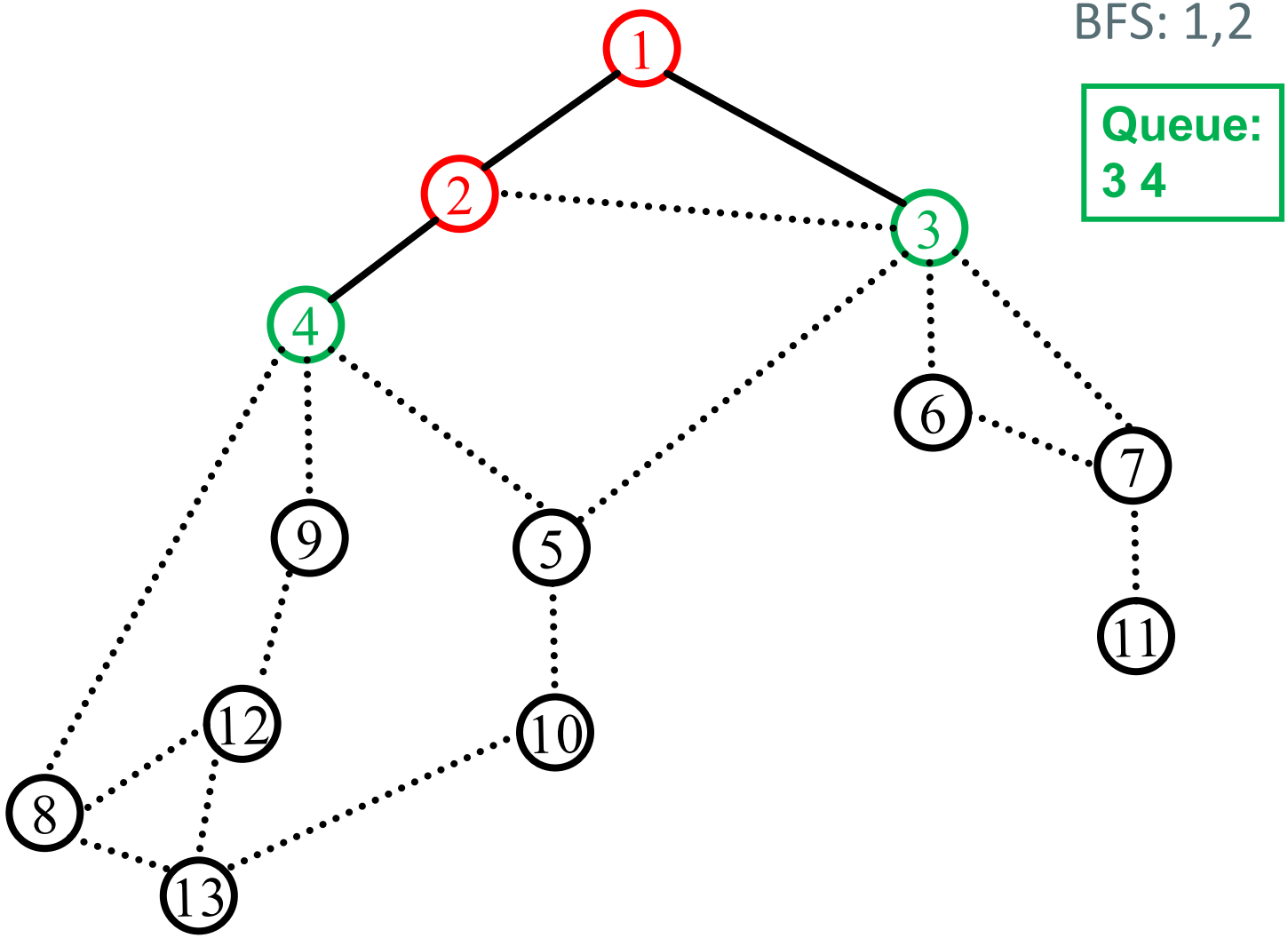
BFS:

Queue:
1

Breadth First Search (BFS)



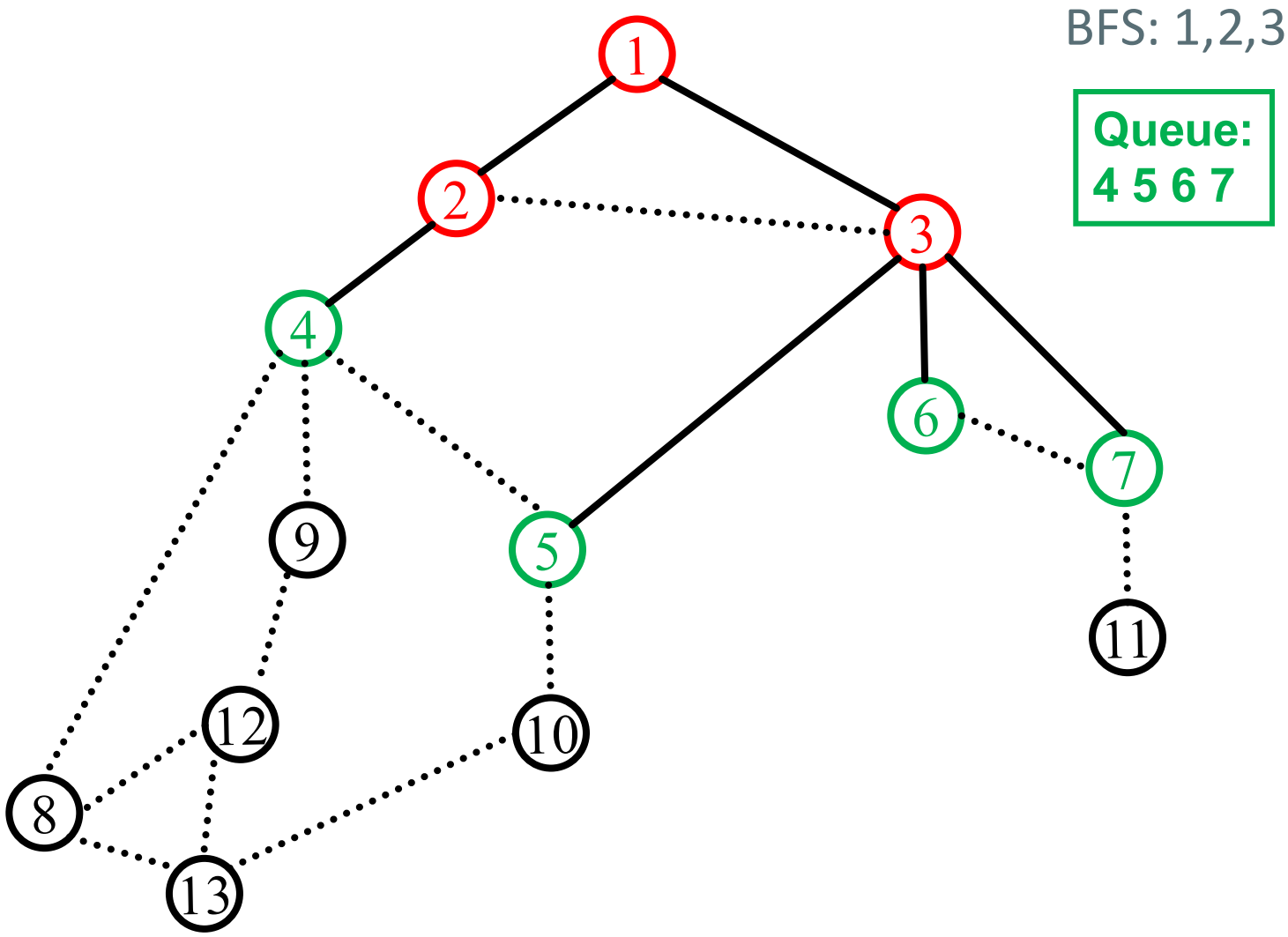
Breadth First Search (BFS)



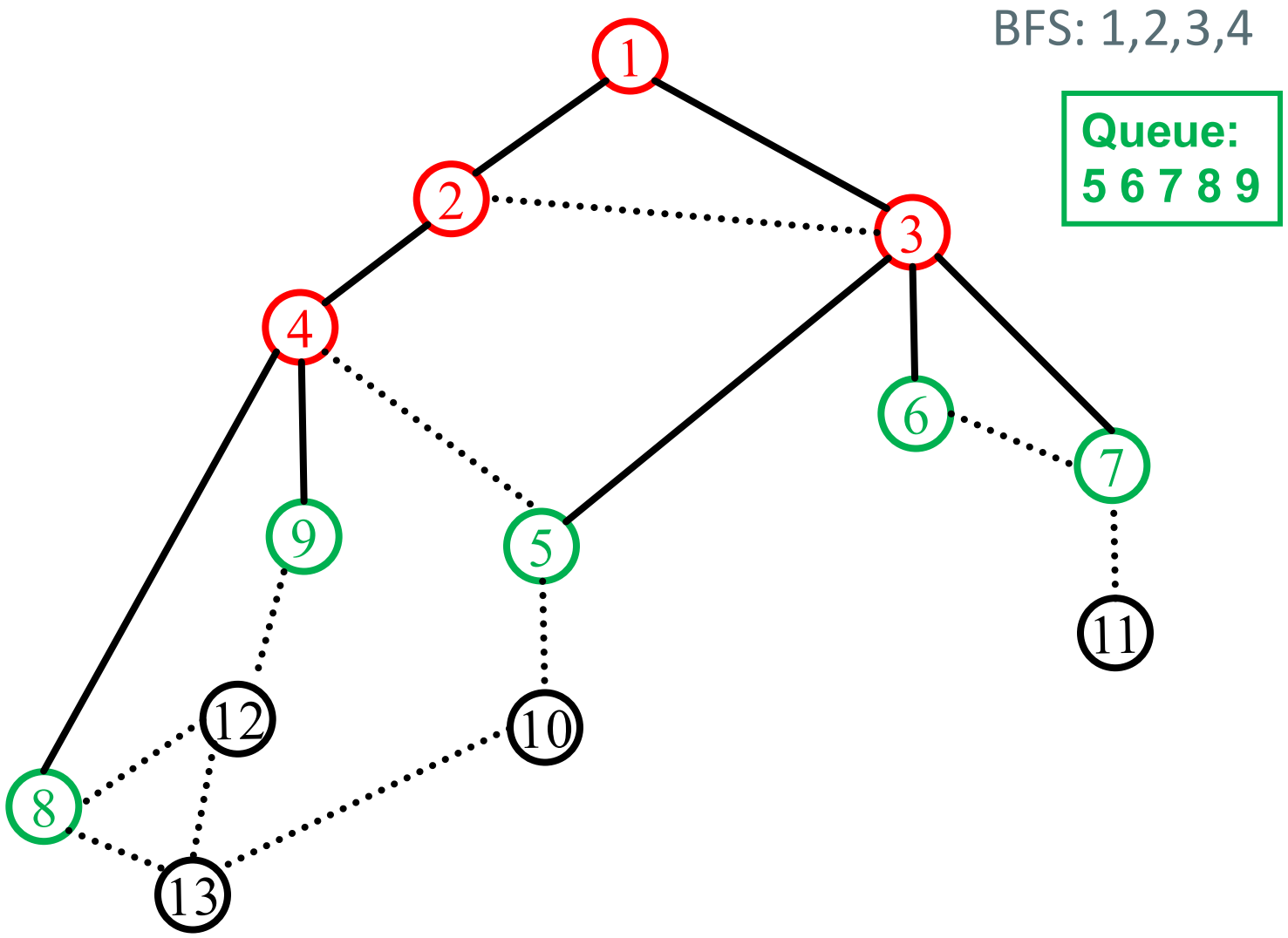
BFS: 1,2

Queue:
3 4

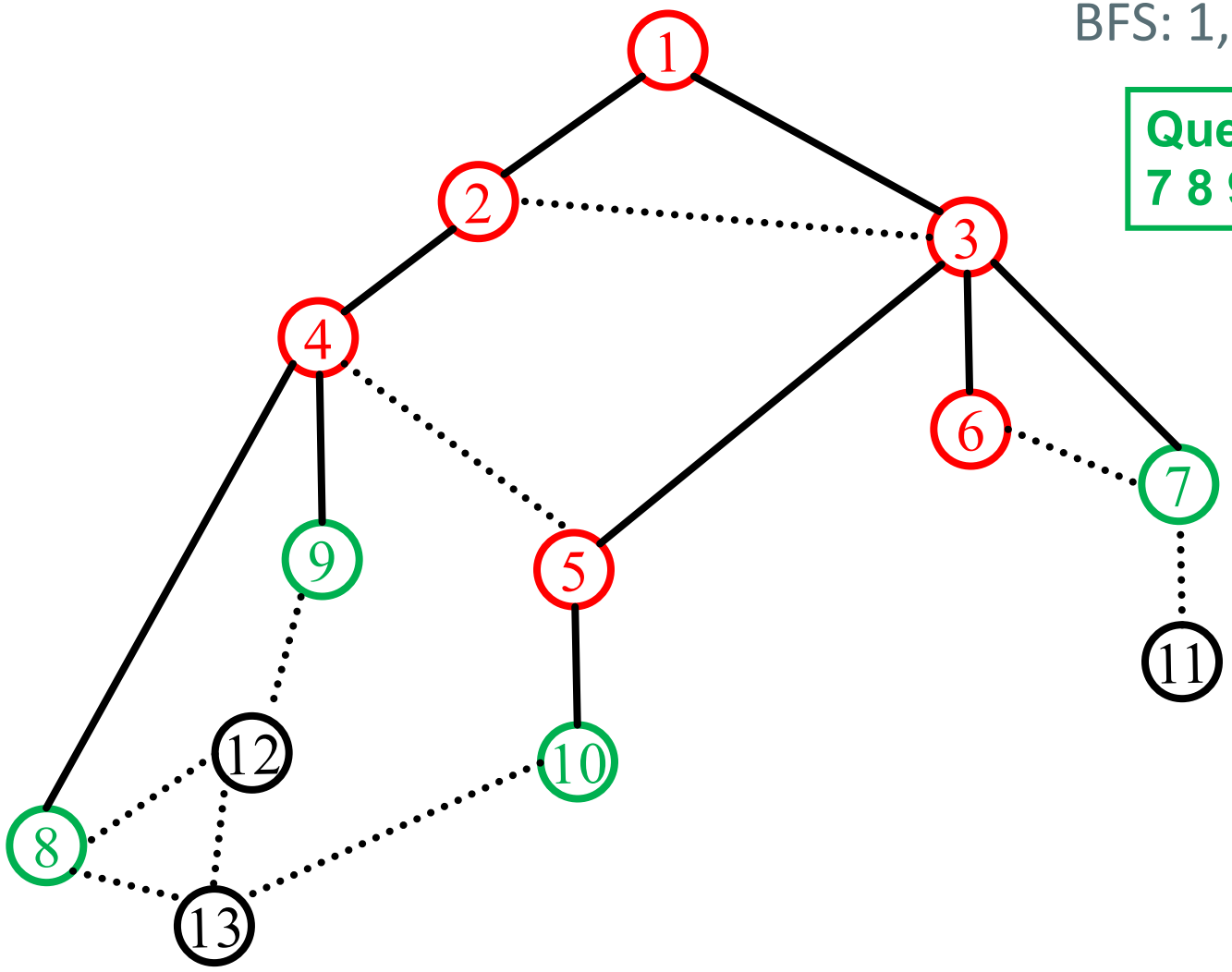
Breadth First Search (BFS)



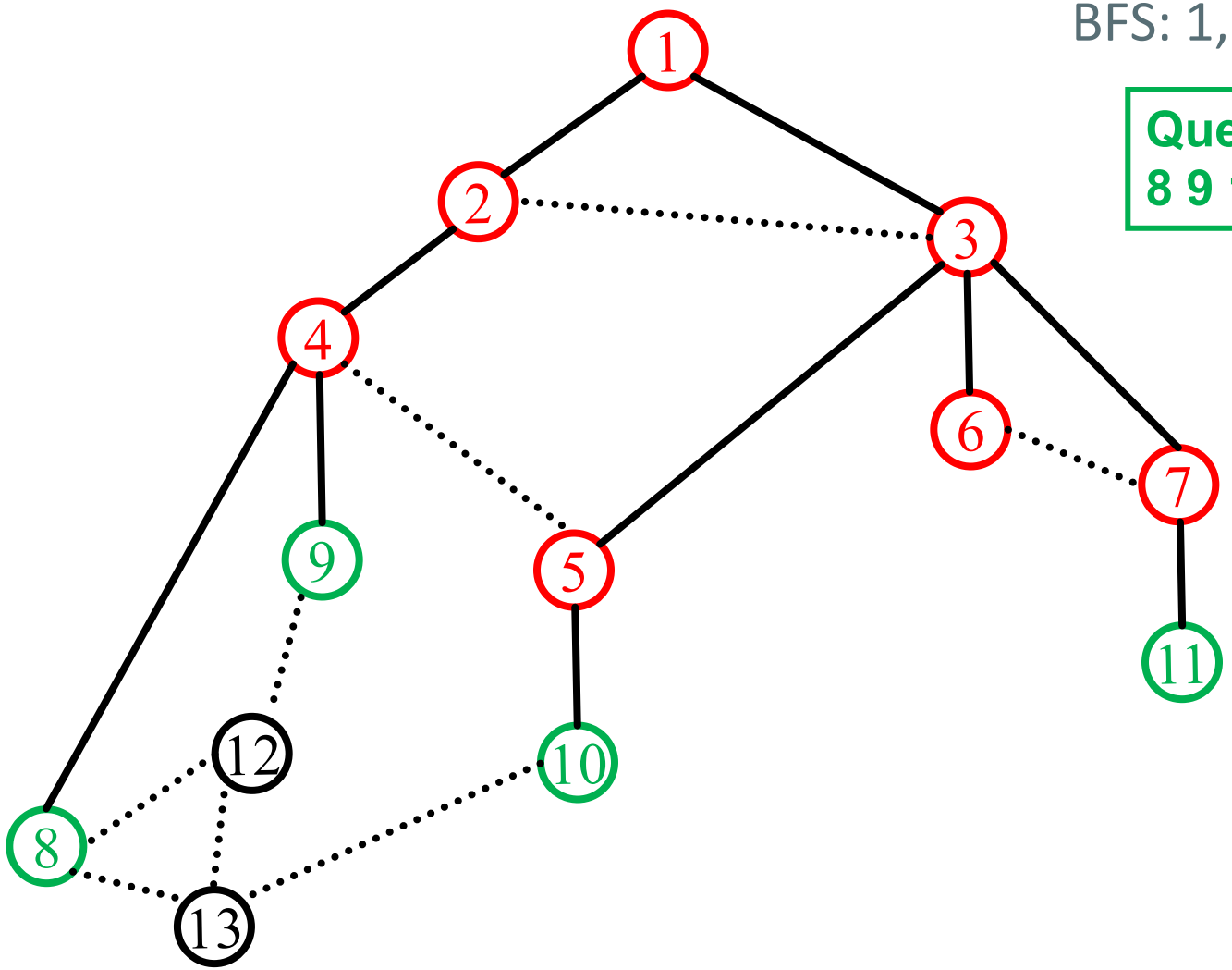
Breadth First Search (BFS)



Breadth First Search (BFS)



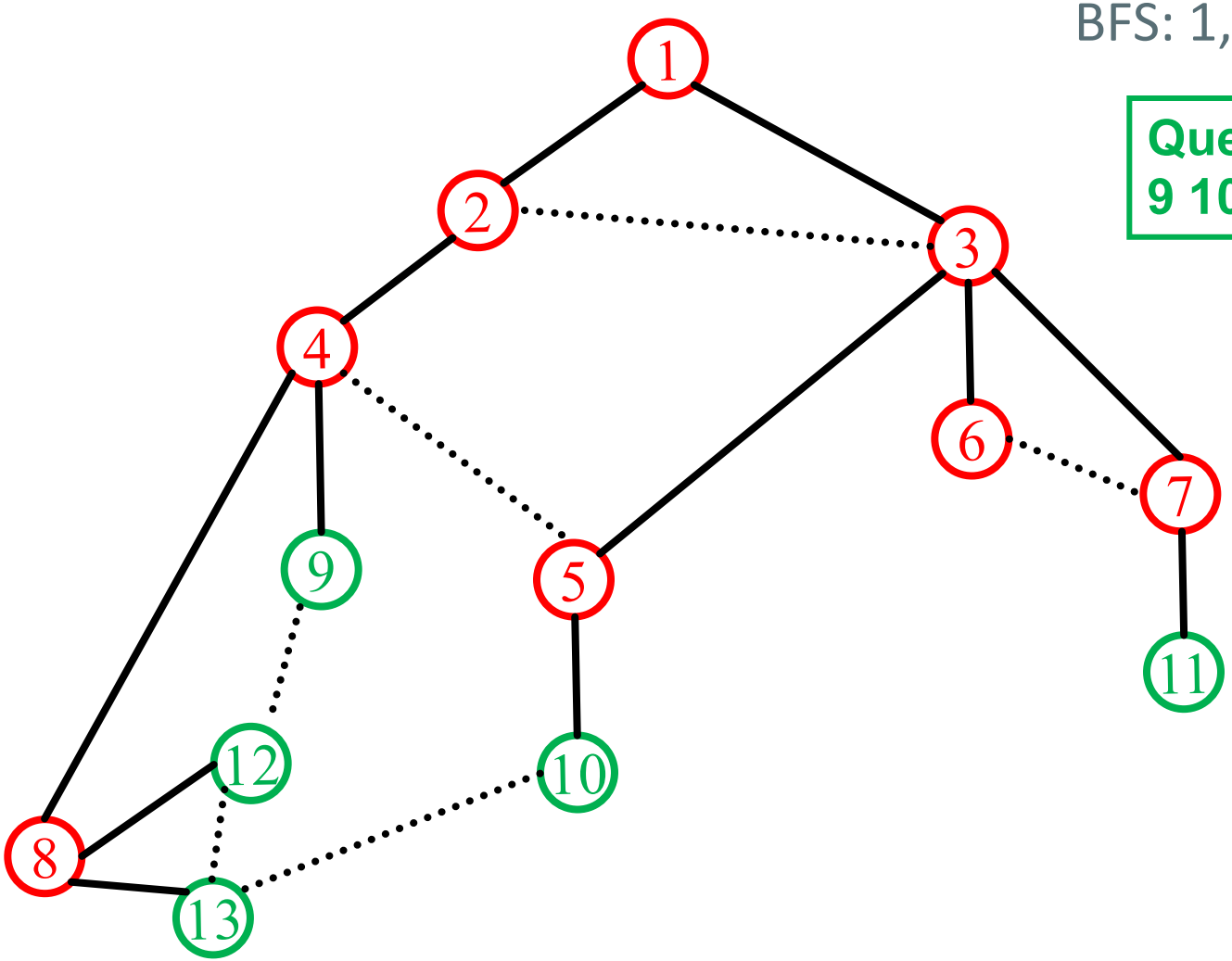
Breadth First Search (BFS)



Breadth First Search (BFS)

BFS: 1,2,3,4,5,6,7,8

Queue:
9 10 11 12 13



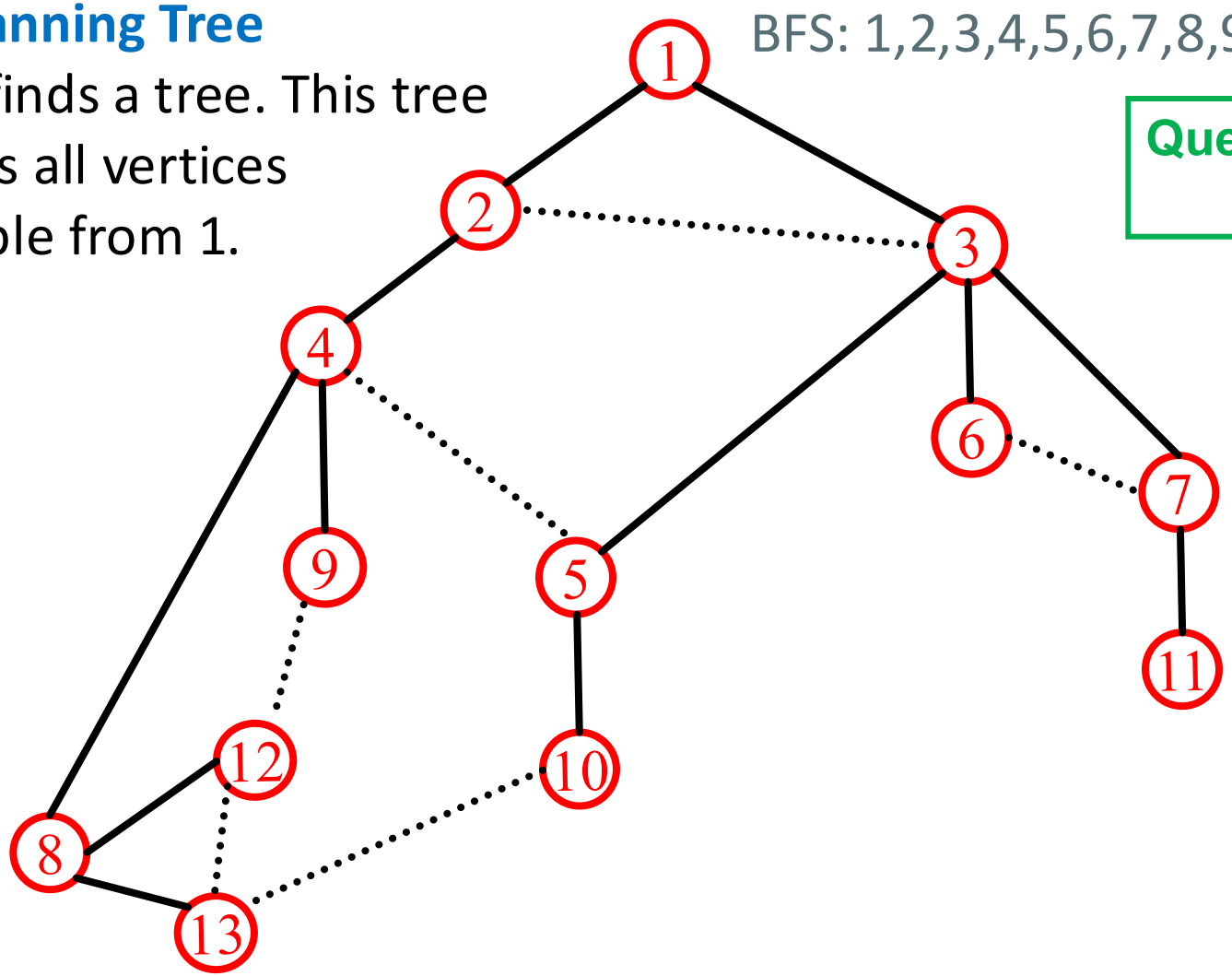
Breadth First Search (BFS)

BFS Spanning Tree

BFS(1) finds a tree. This tree contains all vertices reachable from 1.

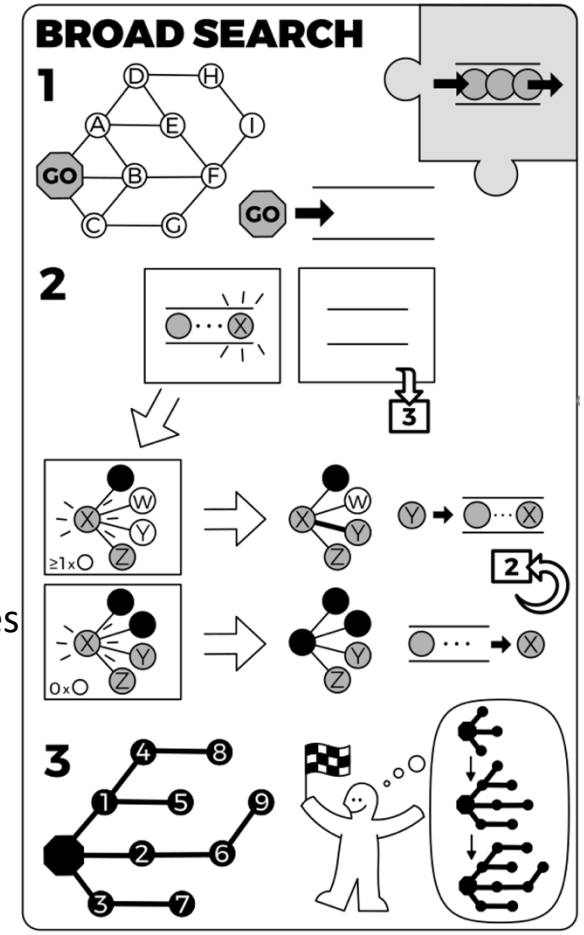
BFS: 1,2,3,4,5,6,7,8,9,10,11,12,13

Queue:



BFS: Implementation

```
BFS(V, E, s)
  for each u ∈ V − {s}
      u.d = ∞                    // initialize "undiscovered"
  s.d = 0
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
      u = DEQUEUE(Q)             // "fully-explored"
      for each v ∈ G.Adj[u]      // if directed graph, v are nodes
          if v.d == ∞                that u has a directed edge to
              v.d = u.d + 1
              ENQUEUE(Q, v)      // "discovered"
```



BFS: Time Complexity Analysis

```
BFS(V, E, s)
  for each u ∈ V − {s}                    // O(V)
      u.d = ∞
  s.d = 0
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
      u = DEQUEUE(Q)
      for each v ∈ G.Adj[u]                // O(E)
          if v.d == ∞
              v.d = u.d + 1
              ENQUEUE(Q, v)                // O(V)
```

$$\sum_{u \in V} degree_u = \begin{cases} 2|E|, & undirected\ graph \\ |E|, & directed\ graph \end{cases}$$

Total time: $O(2V + E)$ -> $O(V + E)$

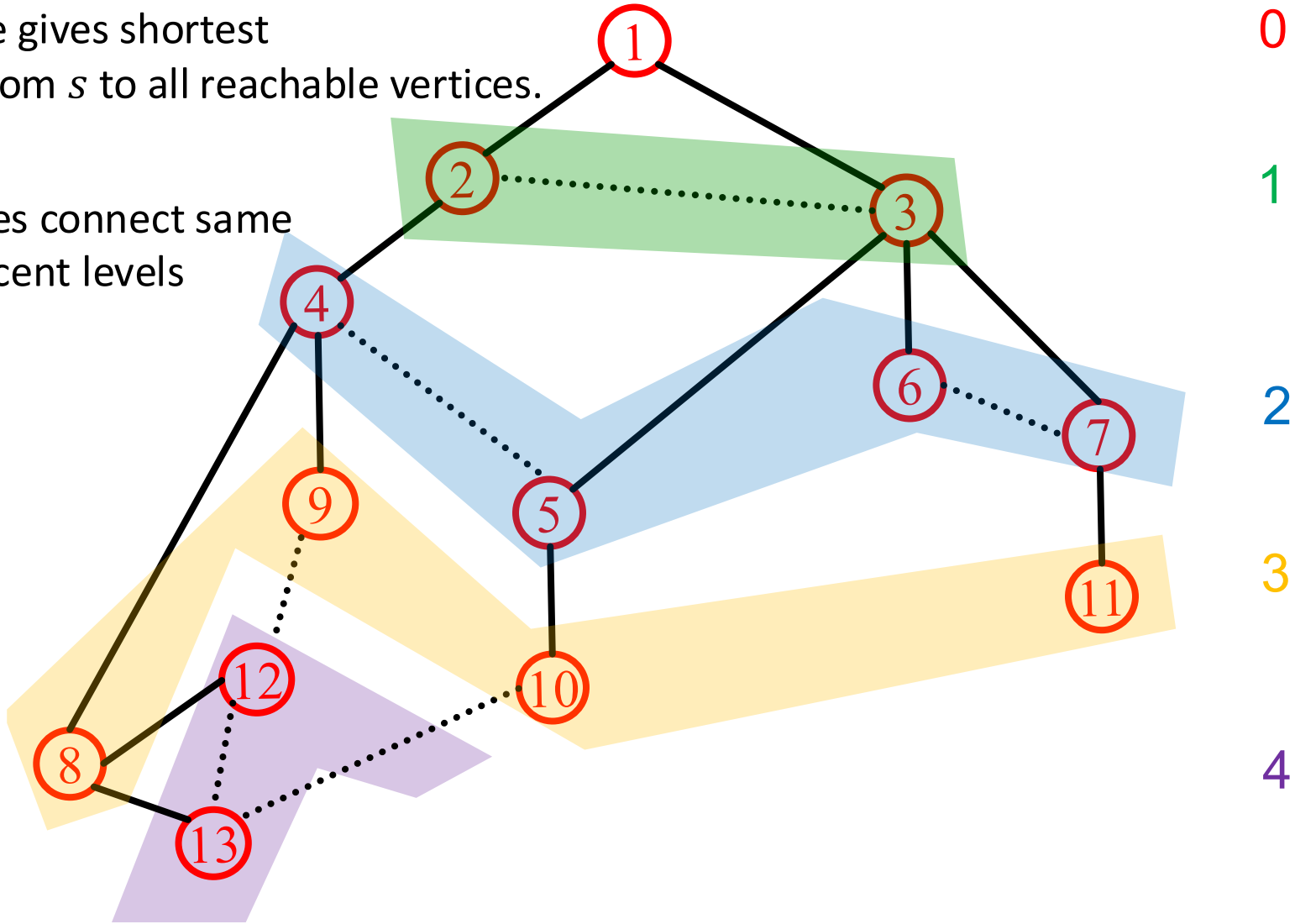
BFS: C++ Implementation

```cpp
void BFS(int start, const vector<vector<int>>& adj, vector<bool>& visited) {
    queue<int> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";

        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}
```

BFS Application: Shortest Paths

Sending a wave out from source vertex s,
BFS tree gives shortest
paths from s to all reachable vertices.

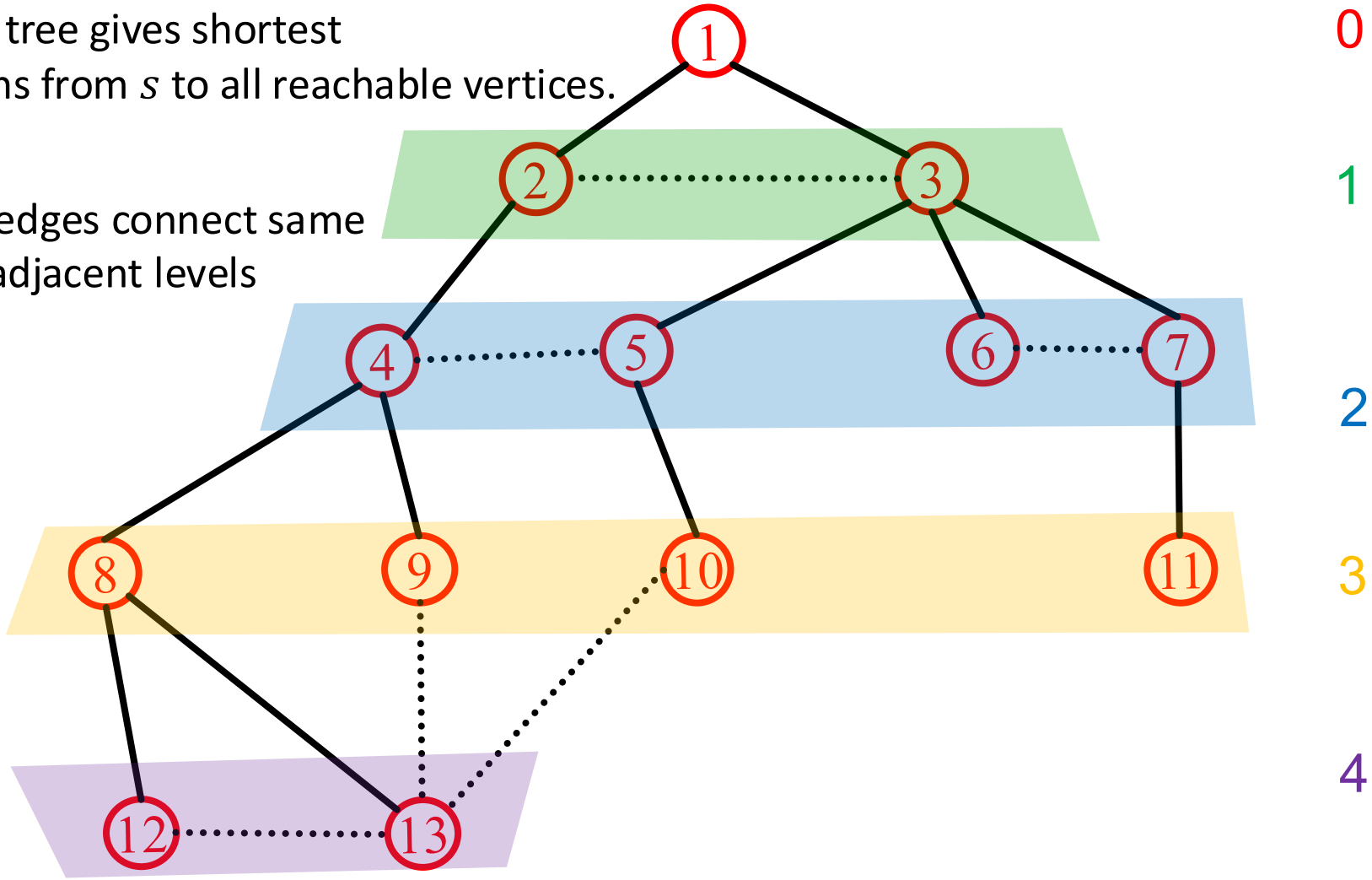
All edges connect same
or adjacent levels



BFS Application: Shortest Paths

Sending a wave out from source vertex s,
BFS tree gives shortest
paths from s to all reachable vertices.

All edges connect same
or adjacent levels



BFS: Properties

Lemma: All vertices at level i of BFS(s) have shortest path distance i to s.

Claim: If $L(v) = i$ then shortest path $\leq i$

Pf: Because there is a path of length i from s to v in the BFS tree

Claim: If shortest path $= i$ then $L(v) \leq i$

Pf: If shortest path $= i$, then say $s \to v_0, v_1, \ldots, v_i \to v$ is the shortest path to v.

By previous claim,

$$L(v_1) \leq L(v_0) + 1$$
$$L(v_2) \leq L(v_1) + 1$$
$$\ldots$$
$$L(v_i) \leq L(v_{i-1}) + 1$$

So, $L(v_i) \leq i$.

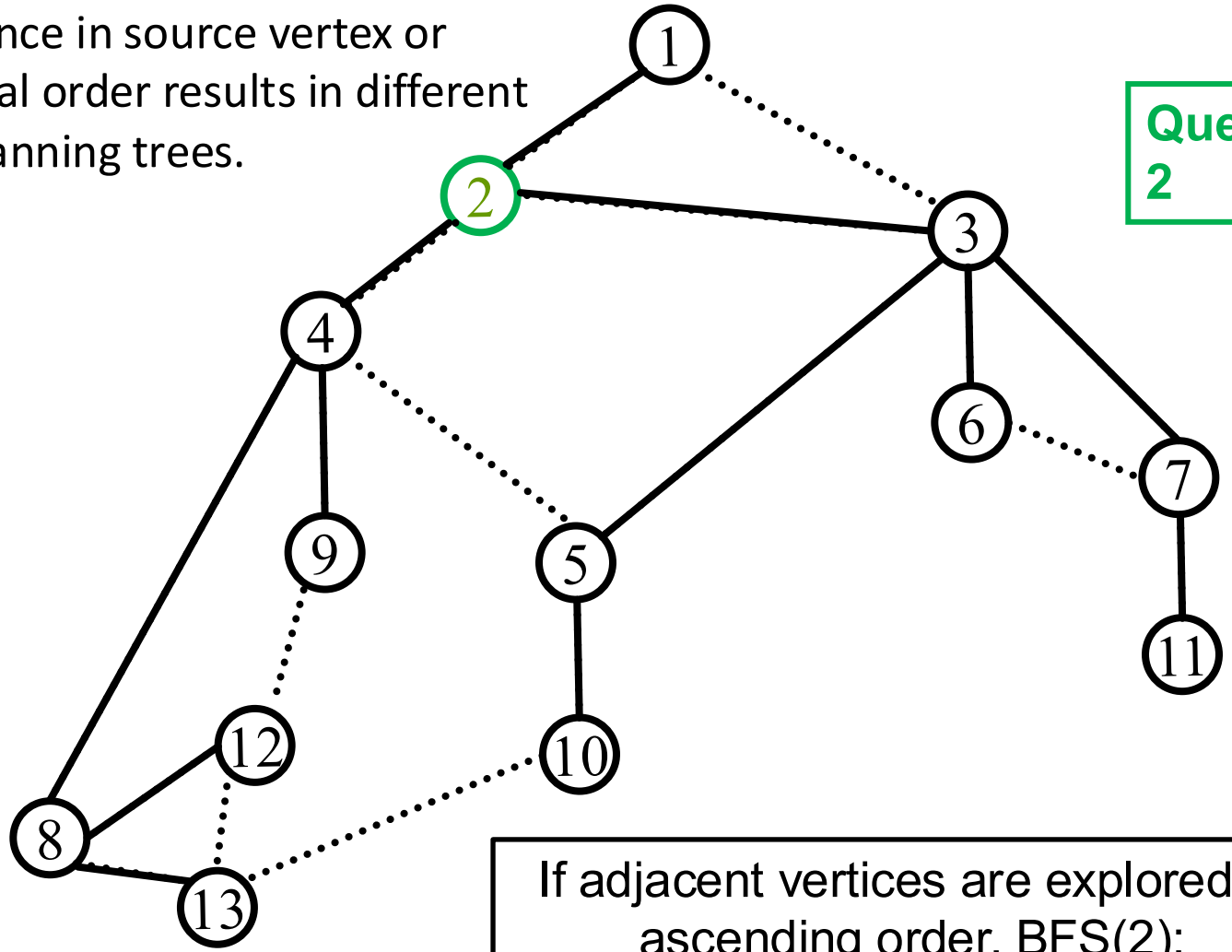
This proves the lemma.

BFS: Wrap-up

- BFS(s) implemented using FIFO queue.
- Edges into the undiscovered vertices define a tree – the "breadth first spanning tree" of G.
- Level i in the tree are exactly all vertices v, s.t., the shortest path (in G) from the root s to v is of length i.
- All nontree edges join vertices on the same or adjacent layers of the tree.
- Applications: reachability, unweighted shortest path, connected component (lab).

Exercise: BFS from Different Source Vertices

Difference in source vertex or traversal order results in different BFS spanning trees.



Queue:
2

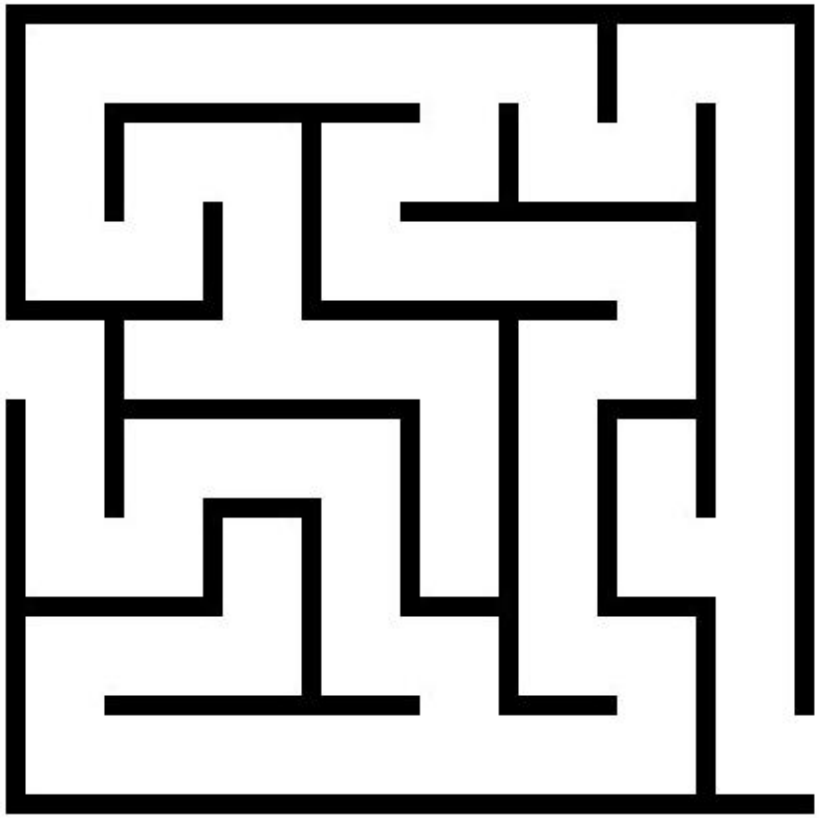
If adjacent vertices are explored in ascending order, BFS(2):
2,1,3,4,5,6,7,8,9,10,11,12,13

Graph Traversal Algorithms

- Depth-first search (DFS)

Depth First Search (DFS)

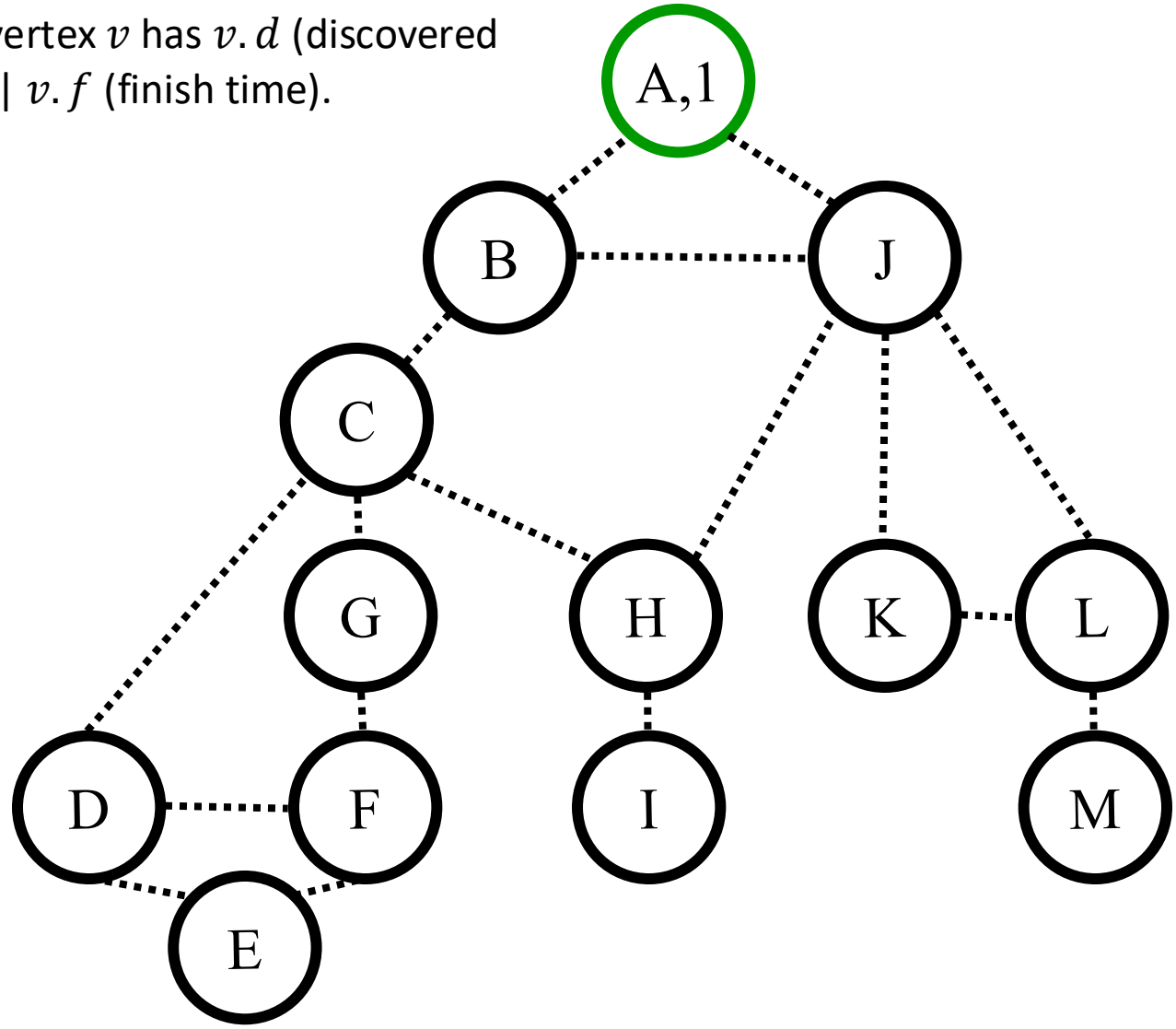
Follow the first path you find as far as you can go; back up to last unexplored edge when you reach a dead end, then go as far you can.



Naturally implemented using **recursive calls or a stack**.

Depth First Search (DFS)

Each vertex v has $v.d$ (discovered time) | $v.f$ (finish time).

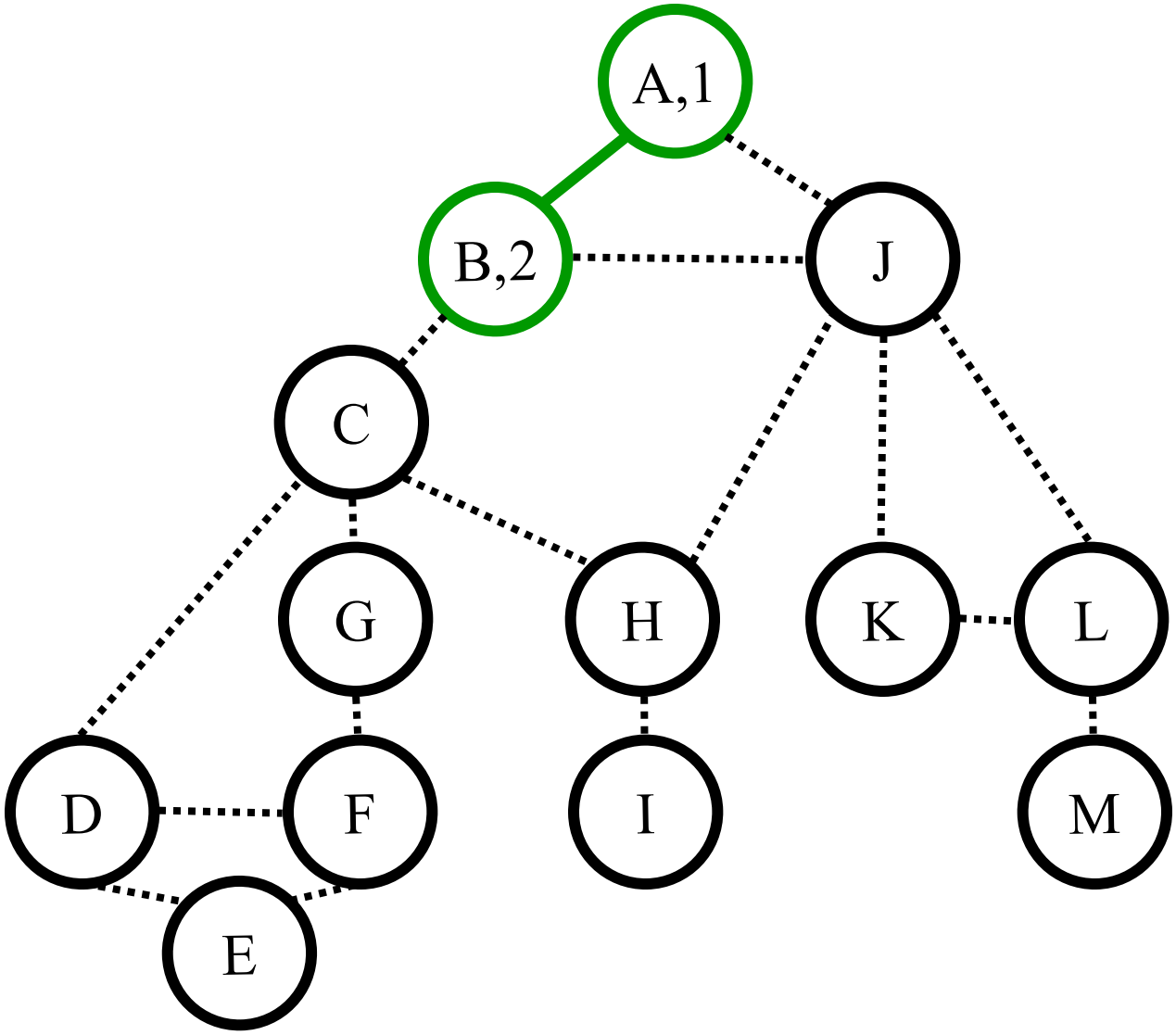


Color code:
Undiscovered
Discovered
fully-explored

Call Stack (Edge list):
A (B,J)

st[] =
{A}

Depth First Search (DFS)



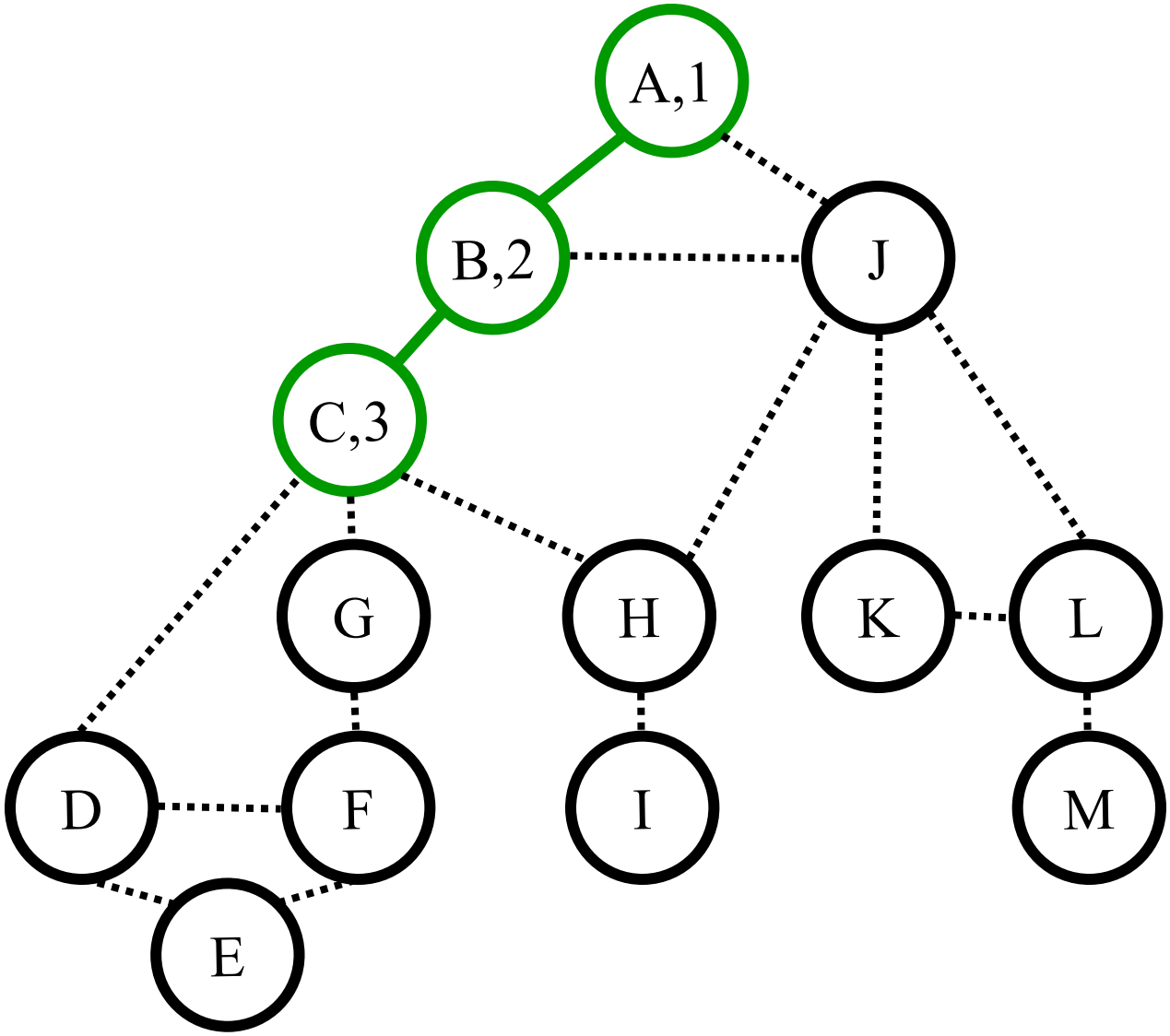
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (A,C,J)

st[] =
{A,B}

Depth First Search (DFS)



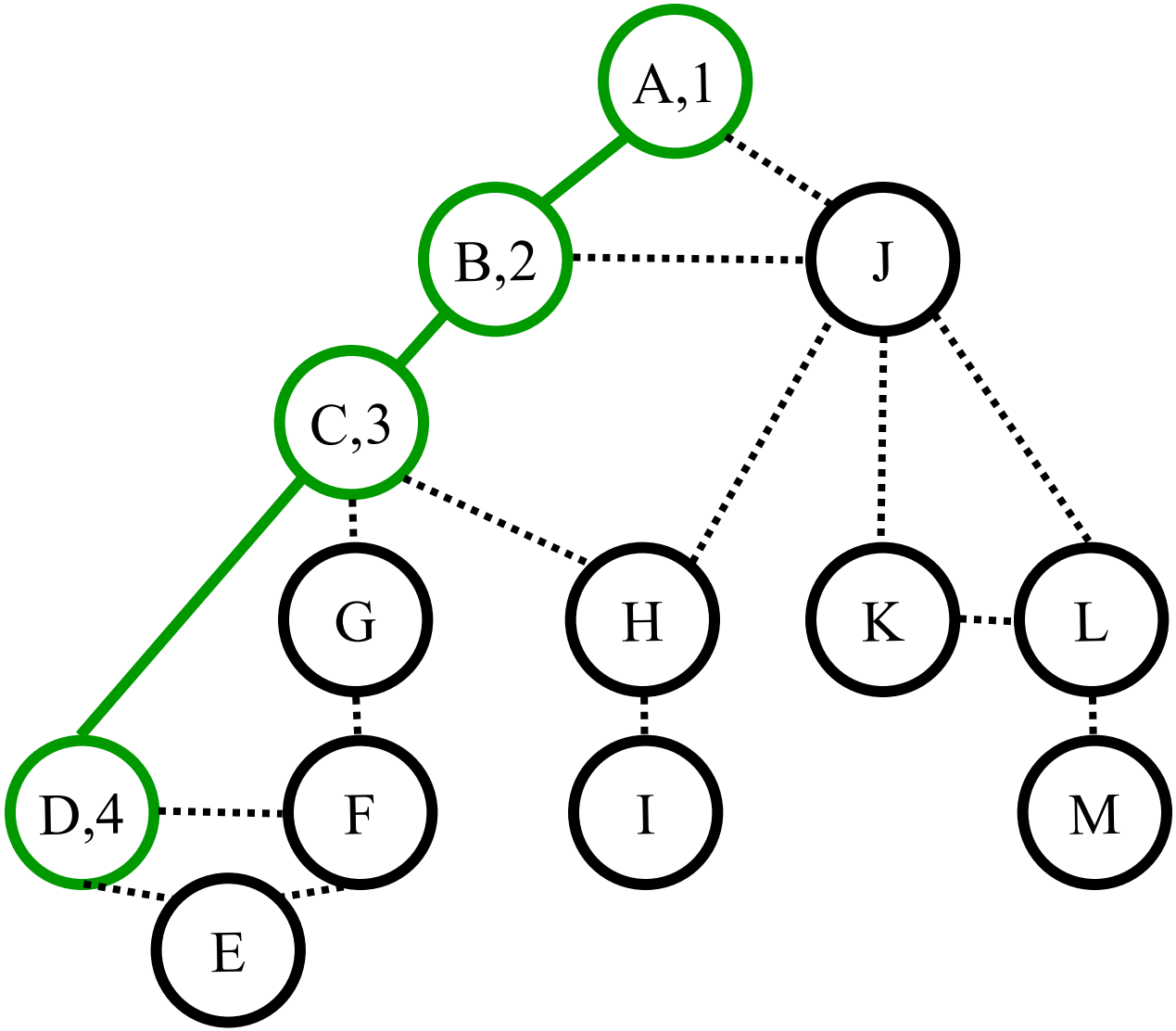
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (B,D,G,H)

st[] =
{A,B,C}

Depth First Search (DFS)



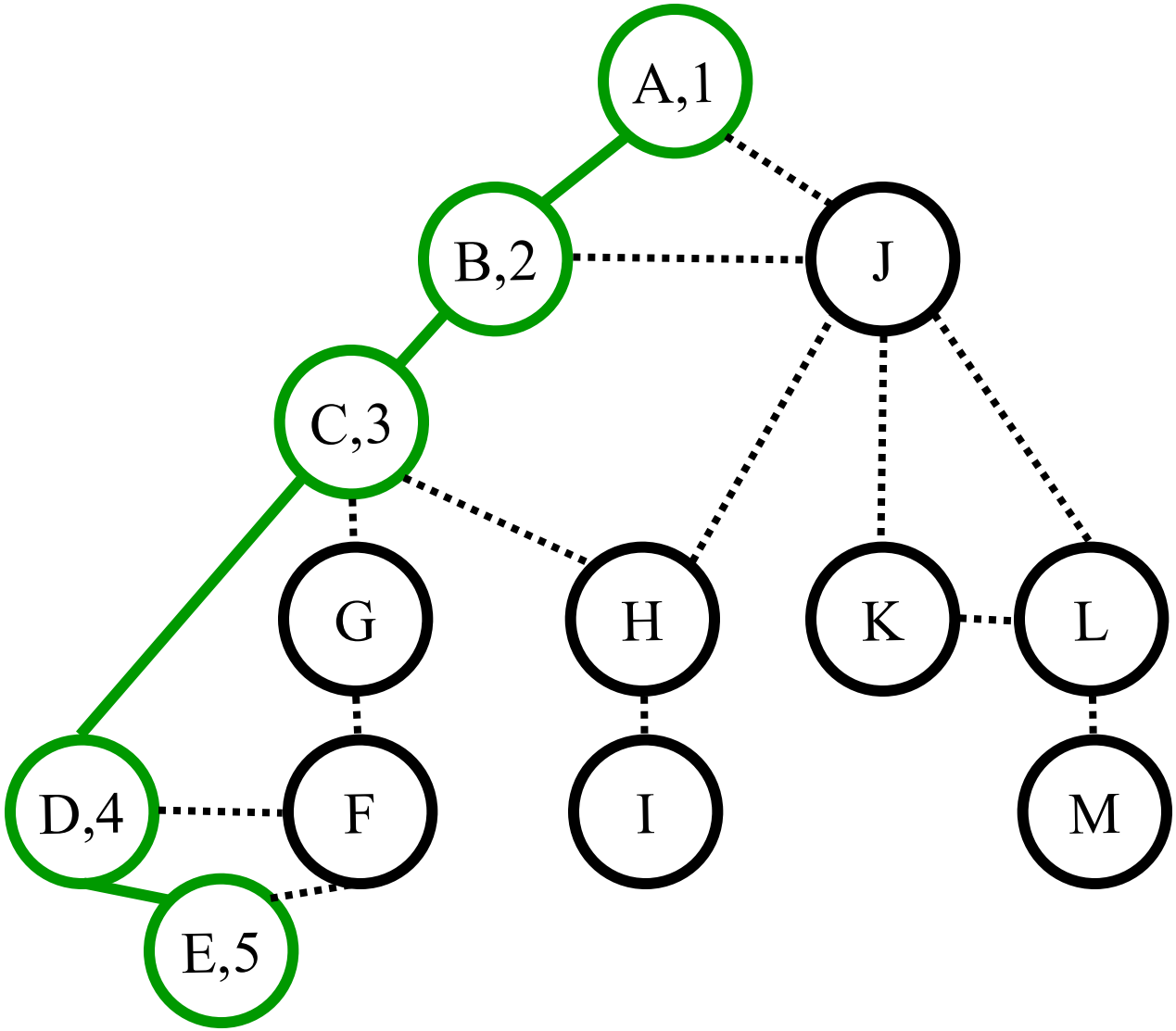
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,G,H)
D (C,E,F)

st[] =
{A,B,C,D}

Depth First Search (DFS)



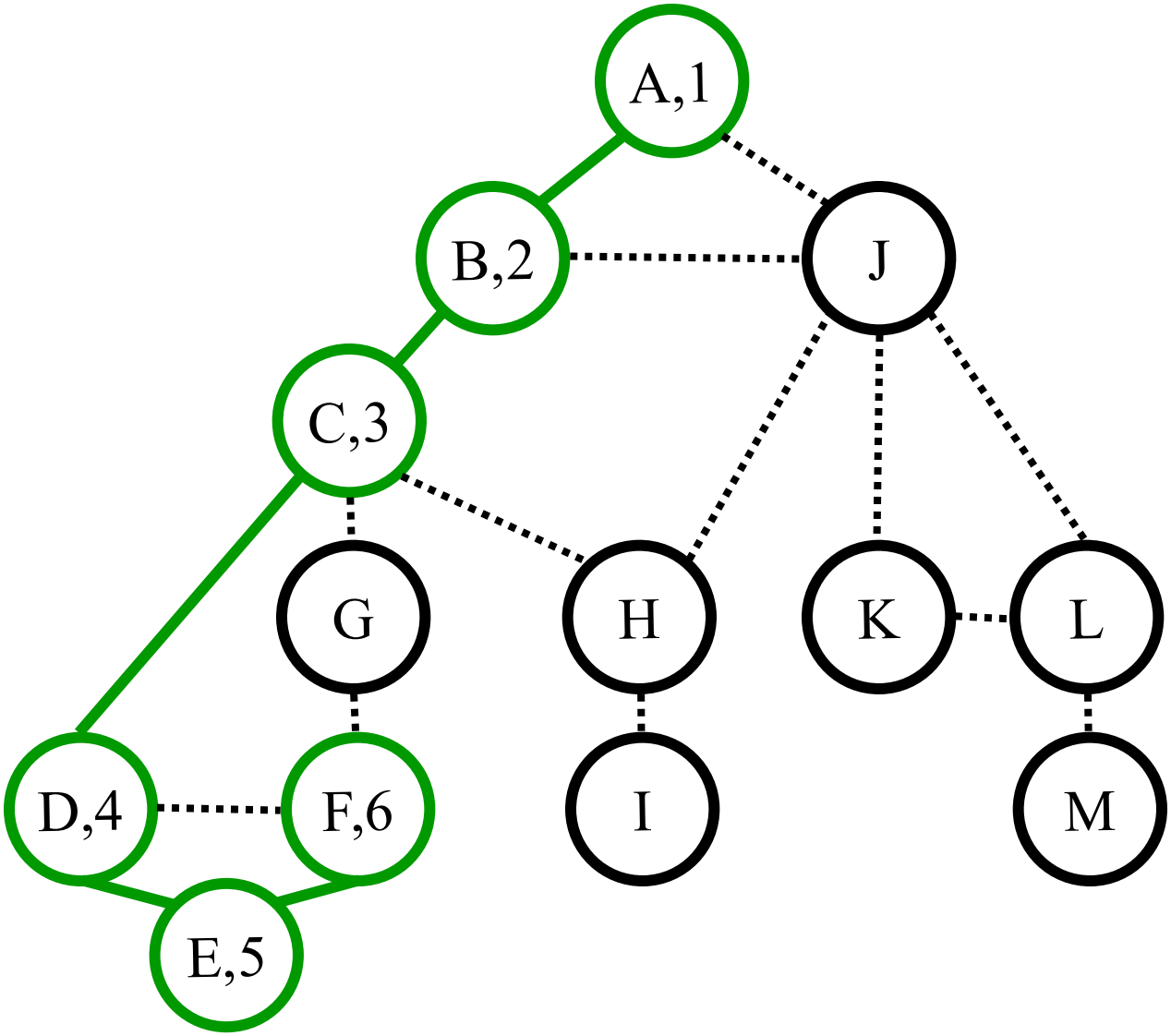
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,G,H)
D (~~C~~,~~E~~,F)
E (D,F)

st[] =
{A,B,C,D,E}

Depth First Search (DFS)



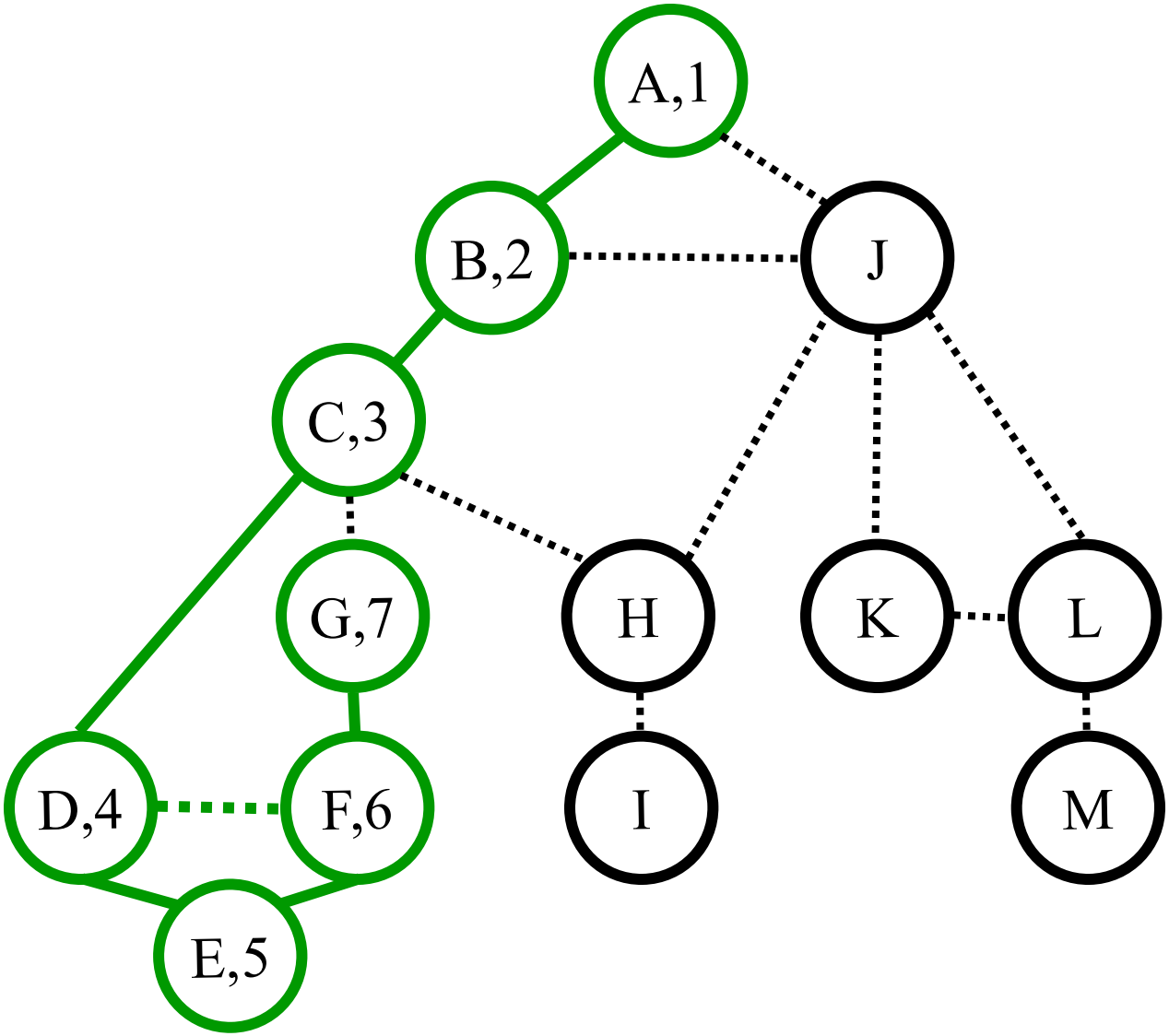
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,G,H)
D (~~C~~,~~E~~,F)
E (~~D~~,~~F~~)
F (D,E,G)

st[] =
{A,B,C,D,E,F}

Depth First Search (DFS)



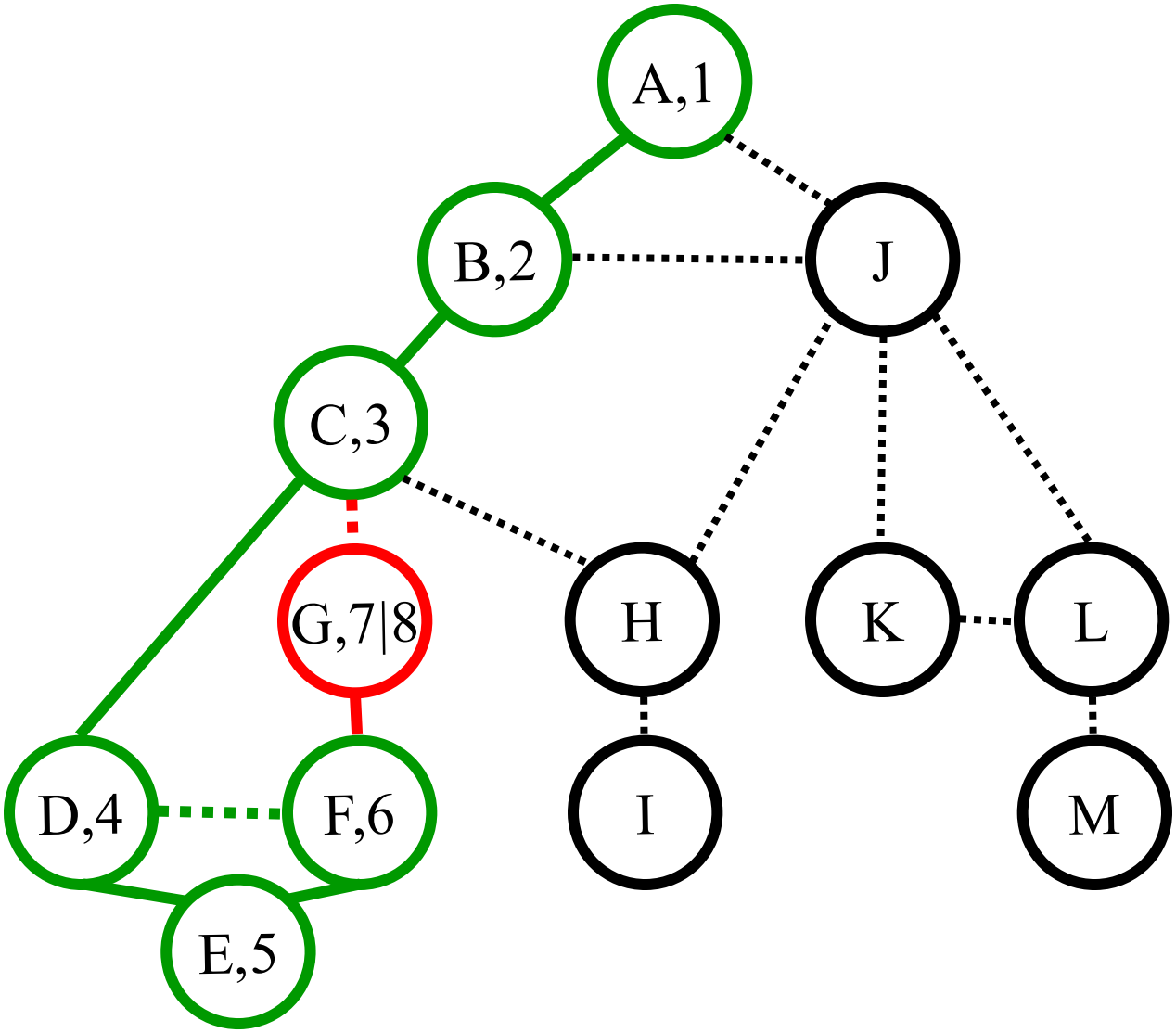
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,G,H)
D (~~C~~,~~E~~,F)
E (~~D~~,~~F~~)
F (~~D~~,~~E~~,~~G~~)
G (C,F)

st[] =
{A,B,C,D,E
,F,G}

Depth First Search (DFS)



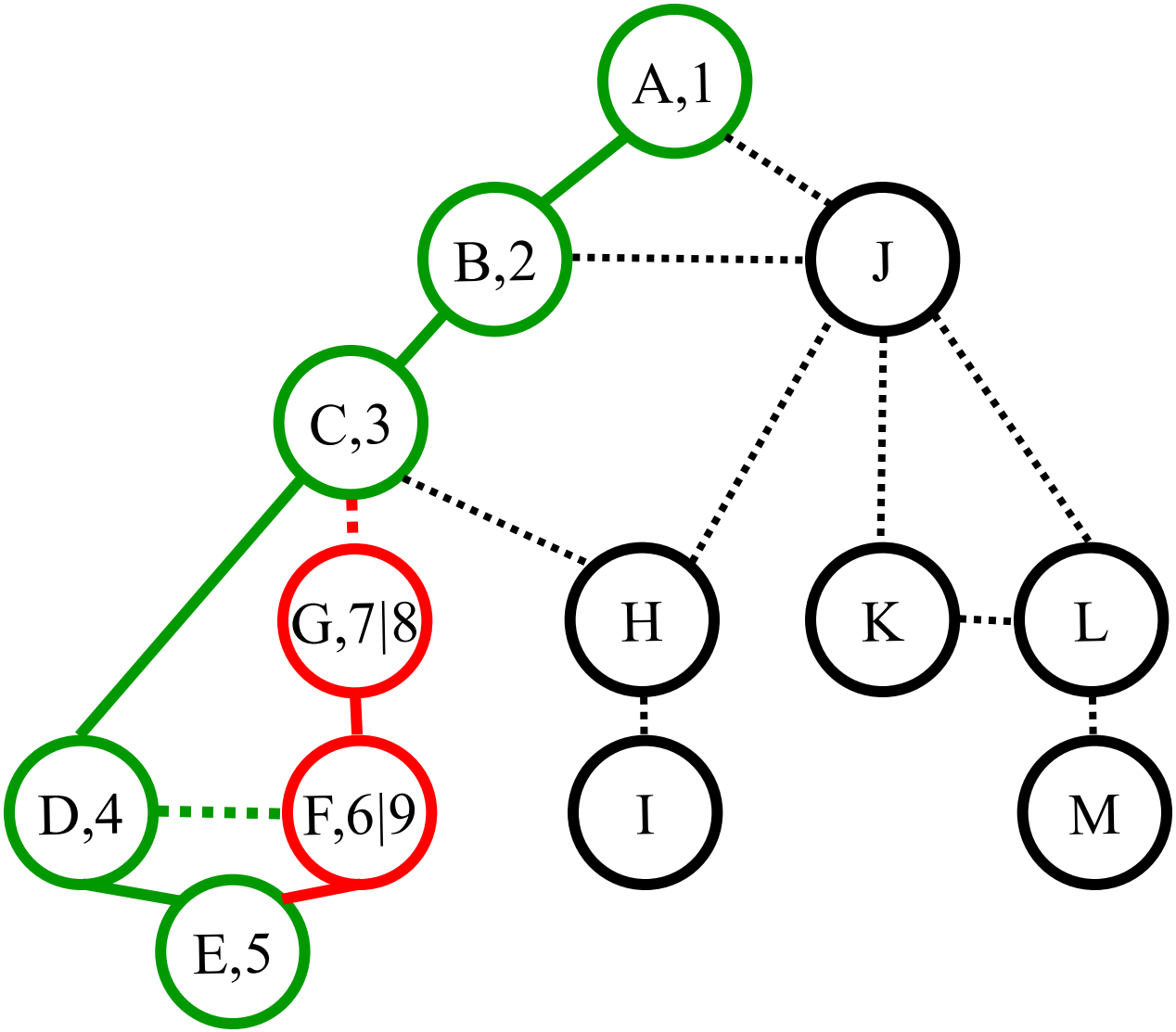
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,G,H)
D (~~C~~,~~E~~,F)
E (~~D~~,~~F~~)
F (~~D~~,~~E~~,~~G~~)

st[] =
{A,B,C,D,E
,F}

Depth First Search (DFS)



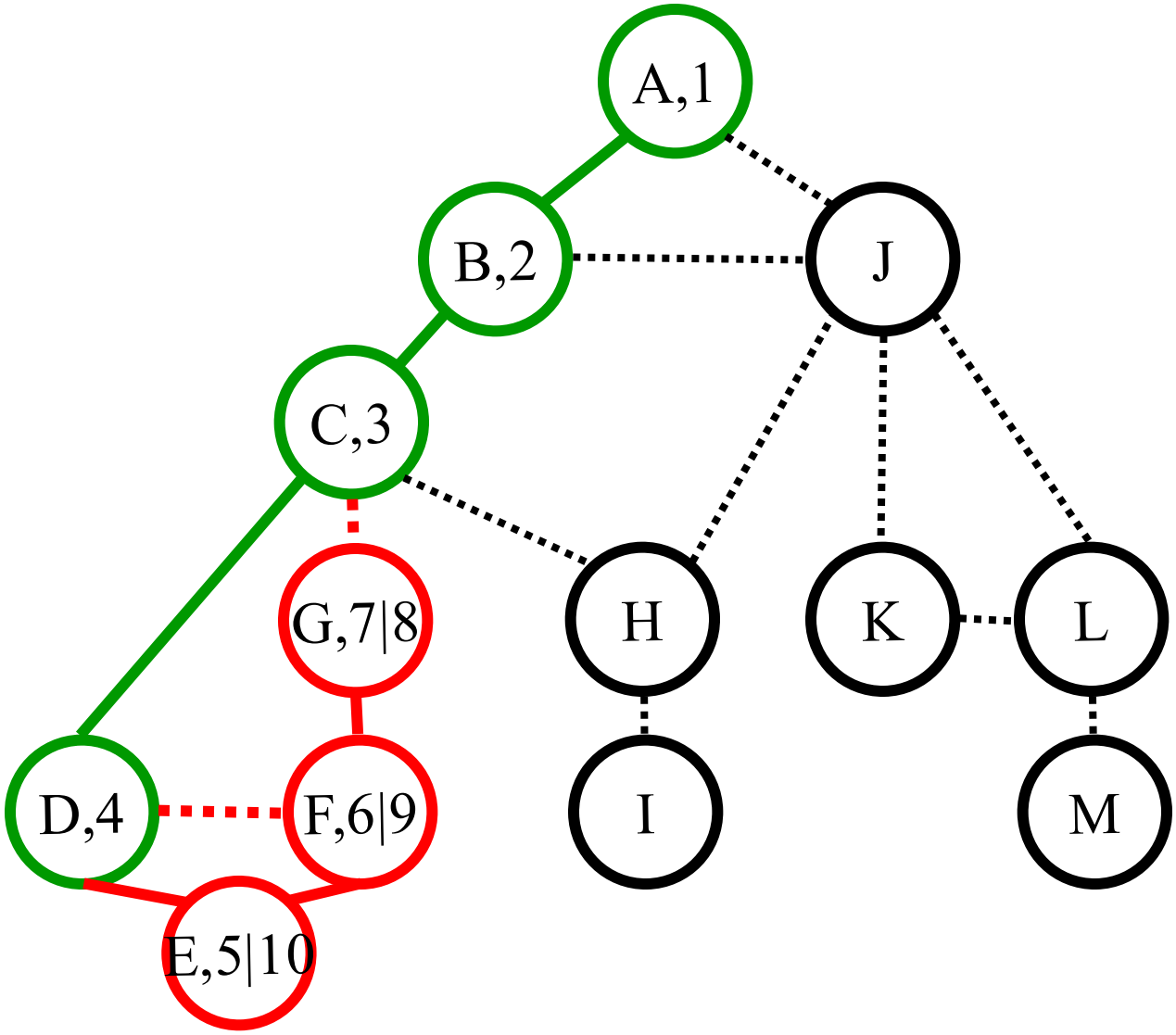
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,G,H)
D (~~C~~,~~E~~,F)
E (~~D~~,~~F~~)

st[] =
{A,B,C,D,E}

Depth First Search (DFS)



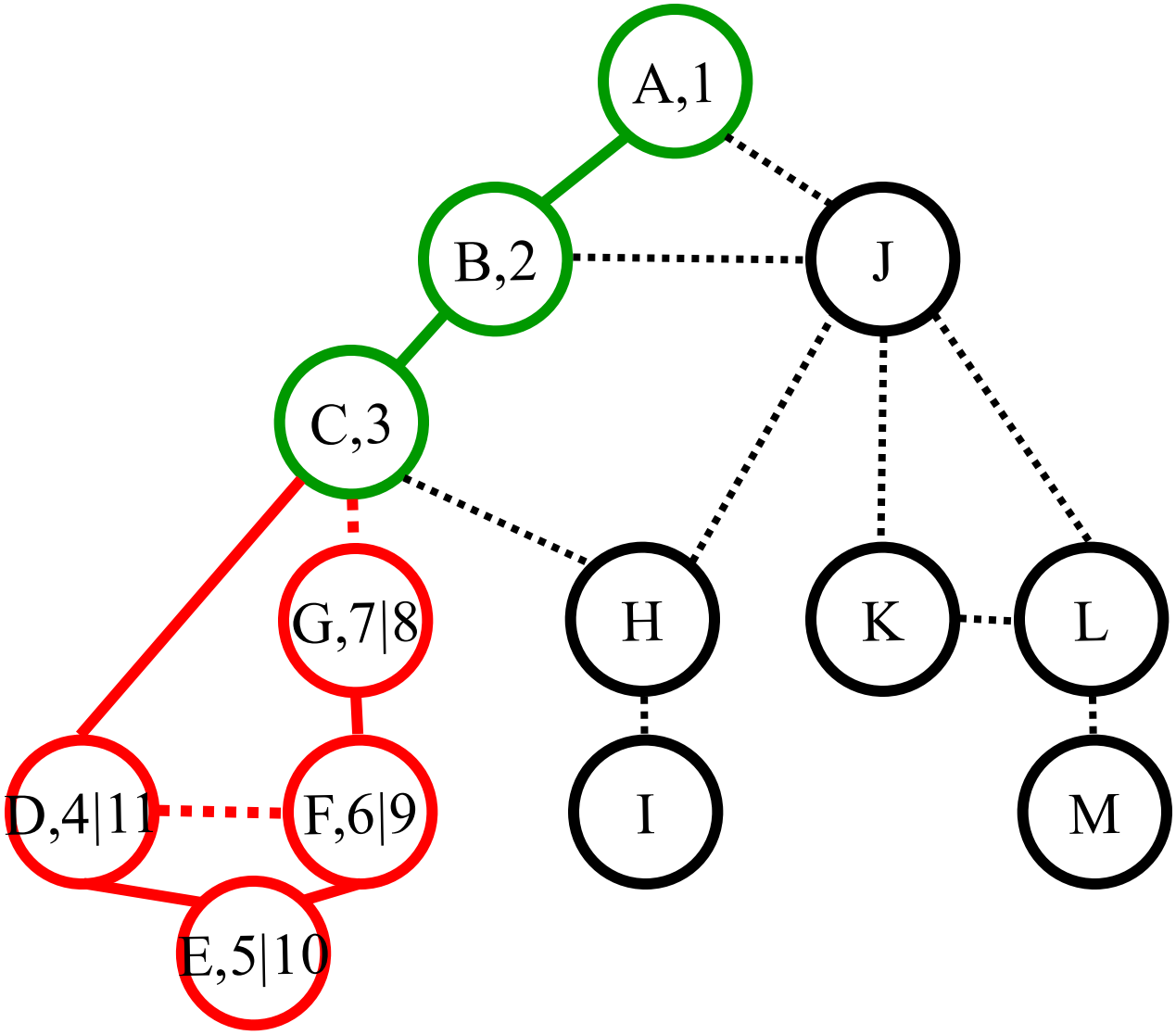
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,G,H)
D (~~C~~,~~E~~,~~F~~)

st[] =
{A,B,C,D}

Depth First Search (DFS)



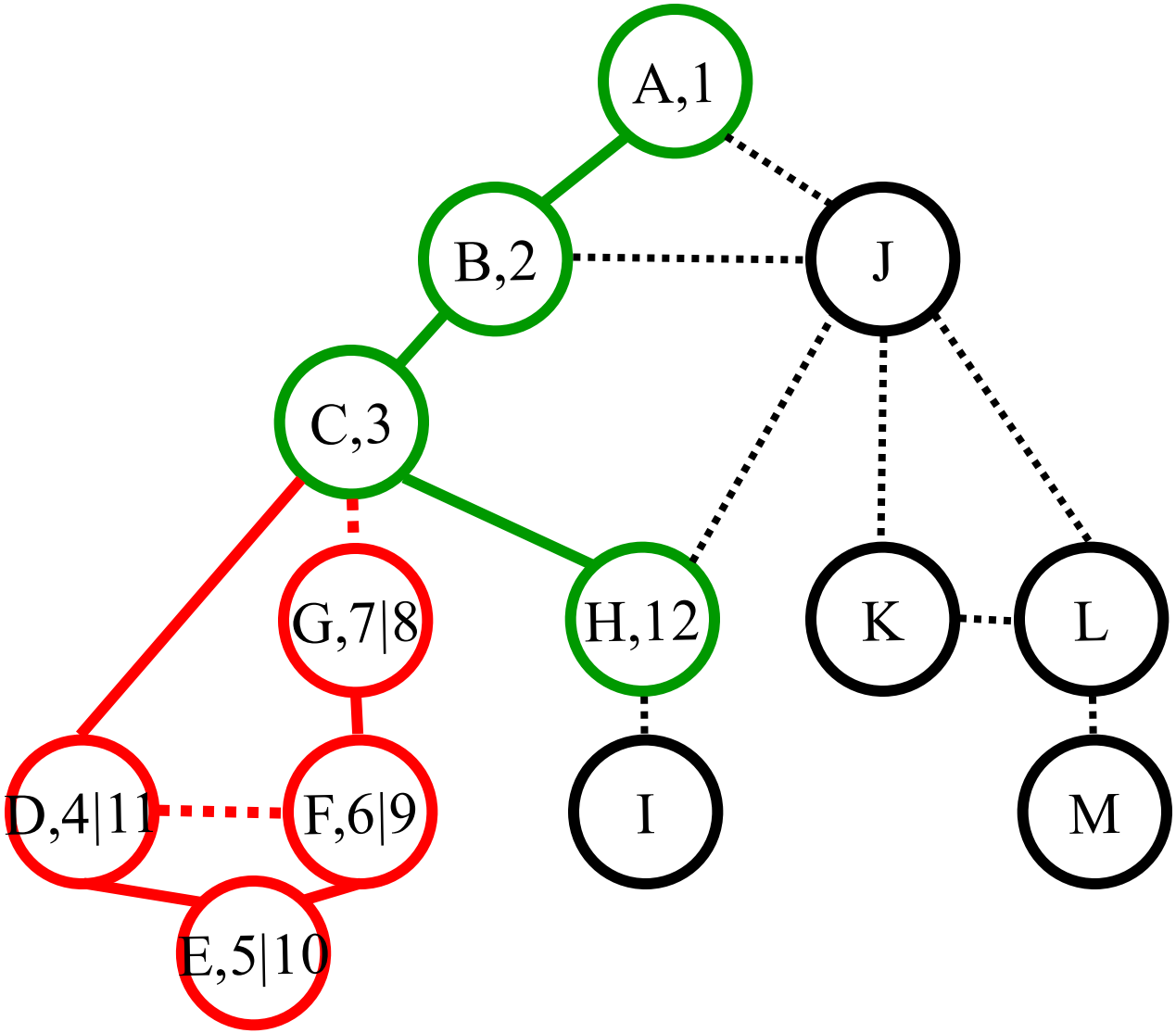
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,G,H)

st[] =
{A,B,C}

Depth First Search (DFS)



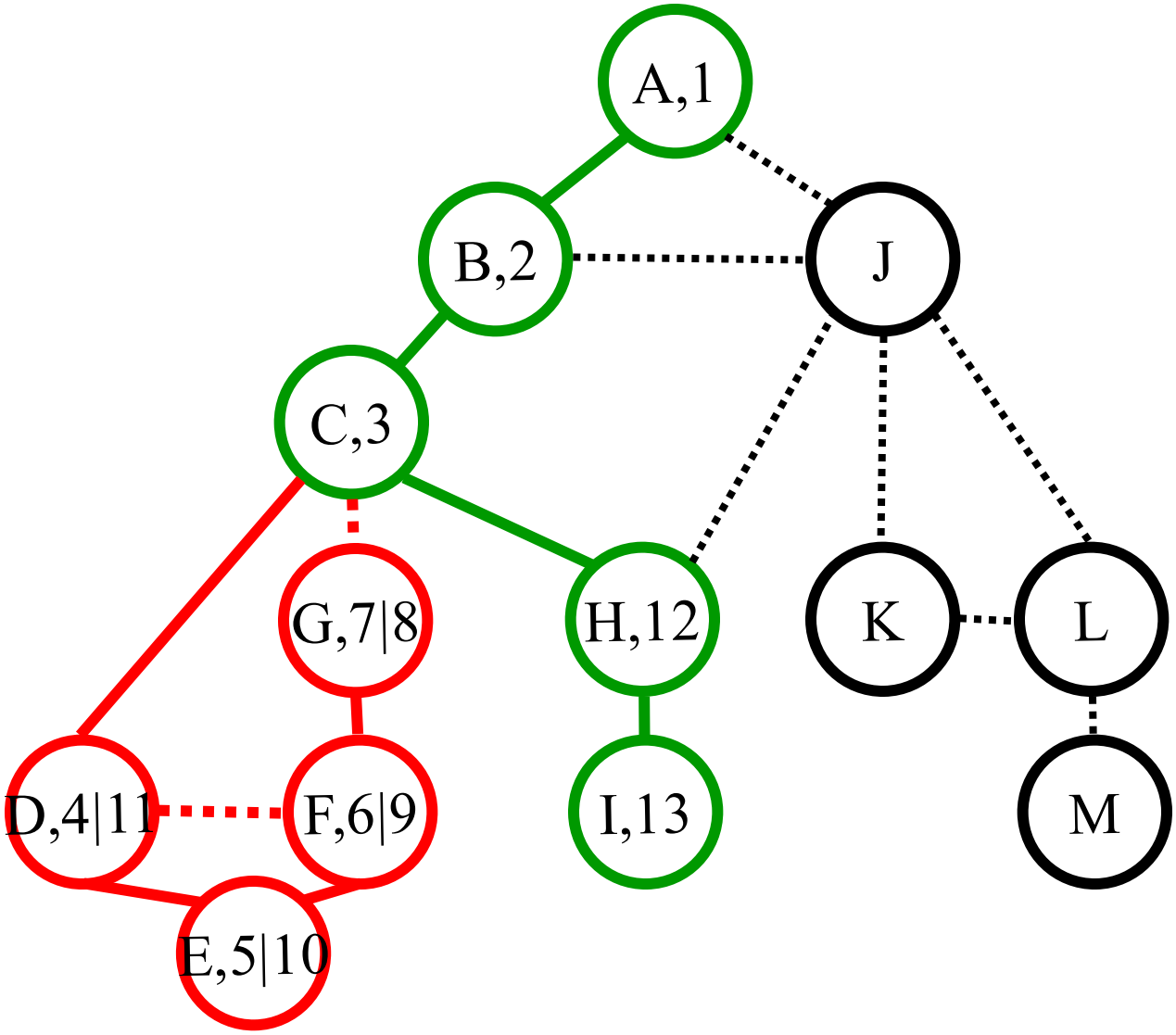
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (C,I,J)

st[] =
{A,B,C,H}

Depth First Search (DFS)



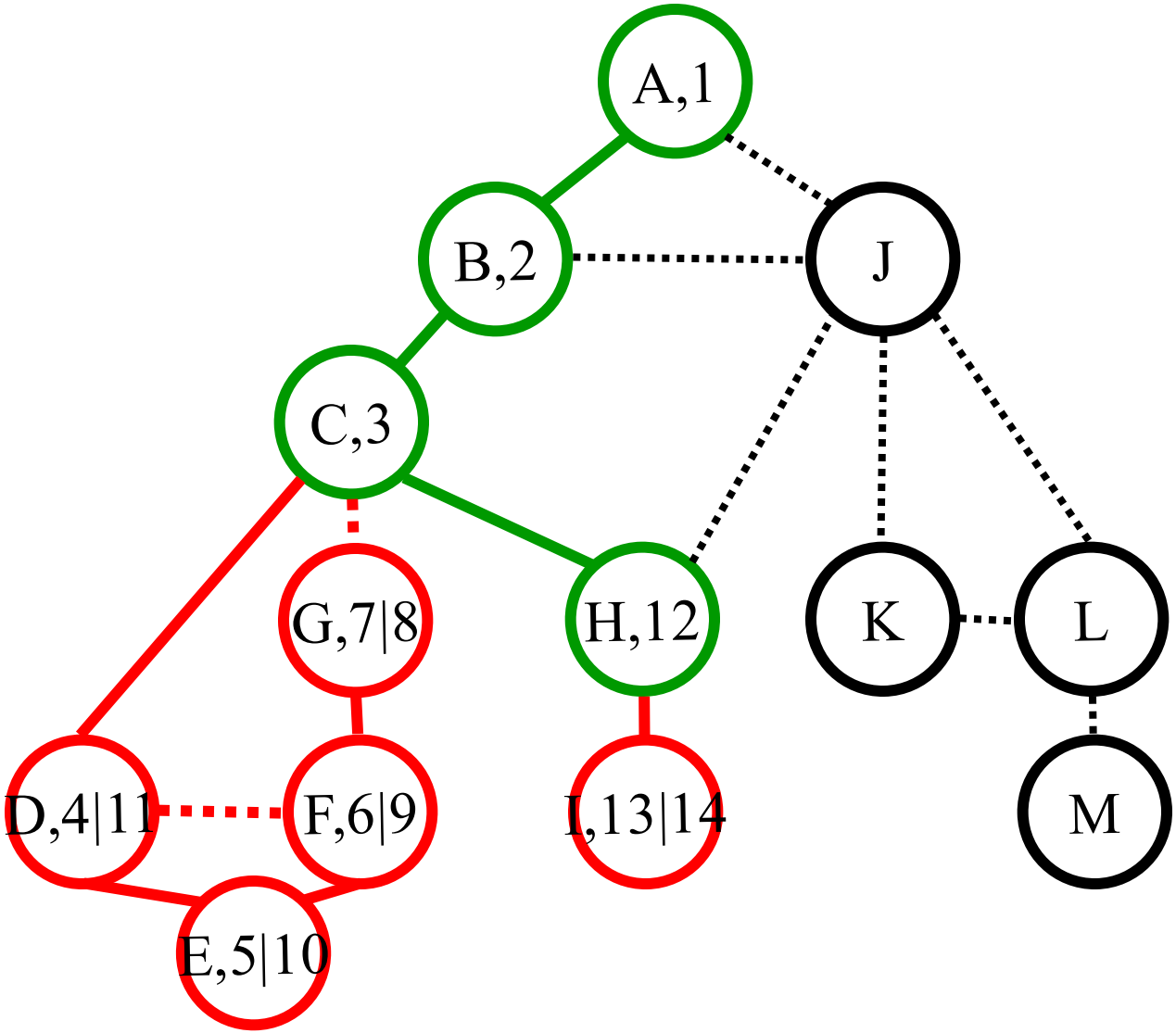
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,J)
I (H)

st[] =
{A,B,C,H,I}

Depth First Search (DFS)



Color code:

Undiscovered

Discovered

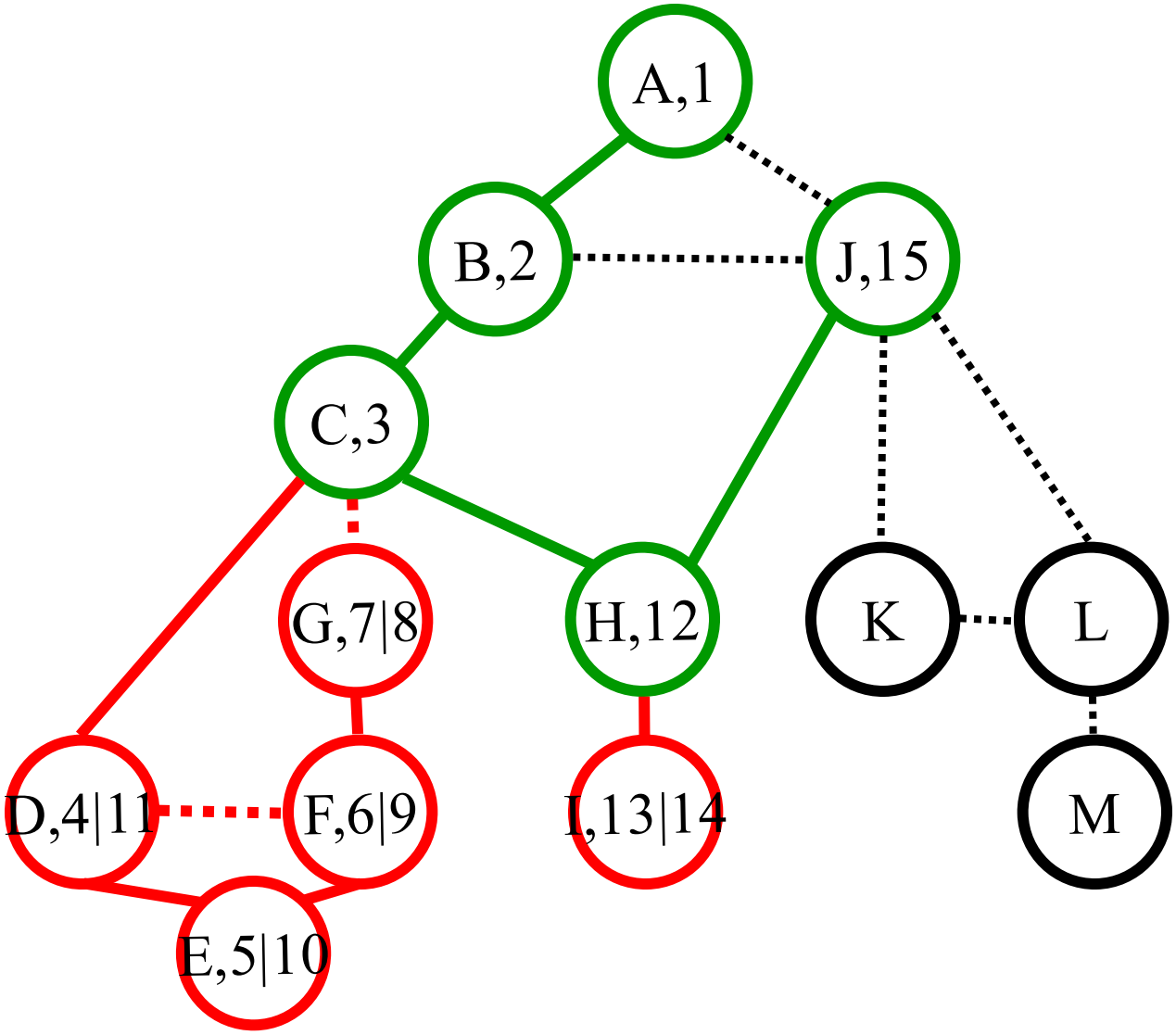
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,J)

st[] =
{A,B,C,H}

Depth First Search (DFS)



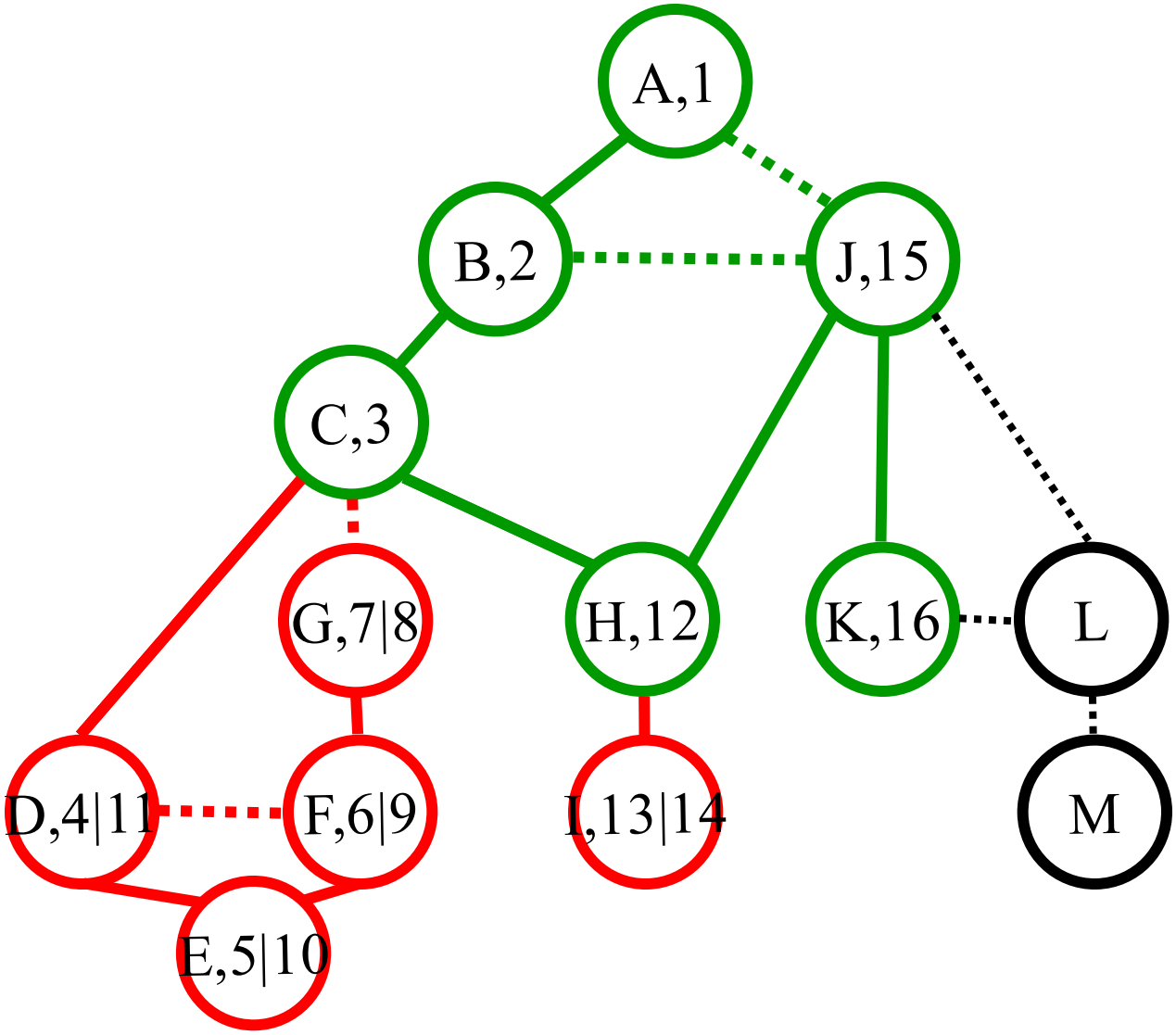
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,~~J~~)
J (A,B,H,K,L)

st[] =
{A,B,C,H,J}

Depth First Search (DFS)



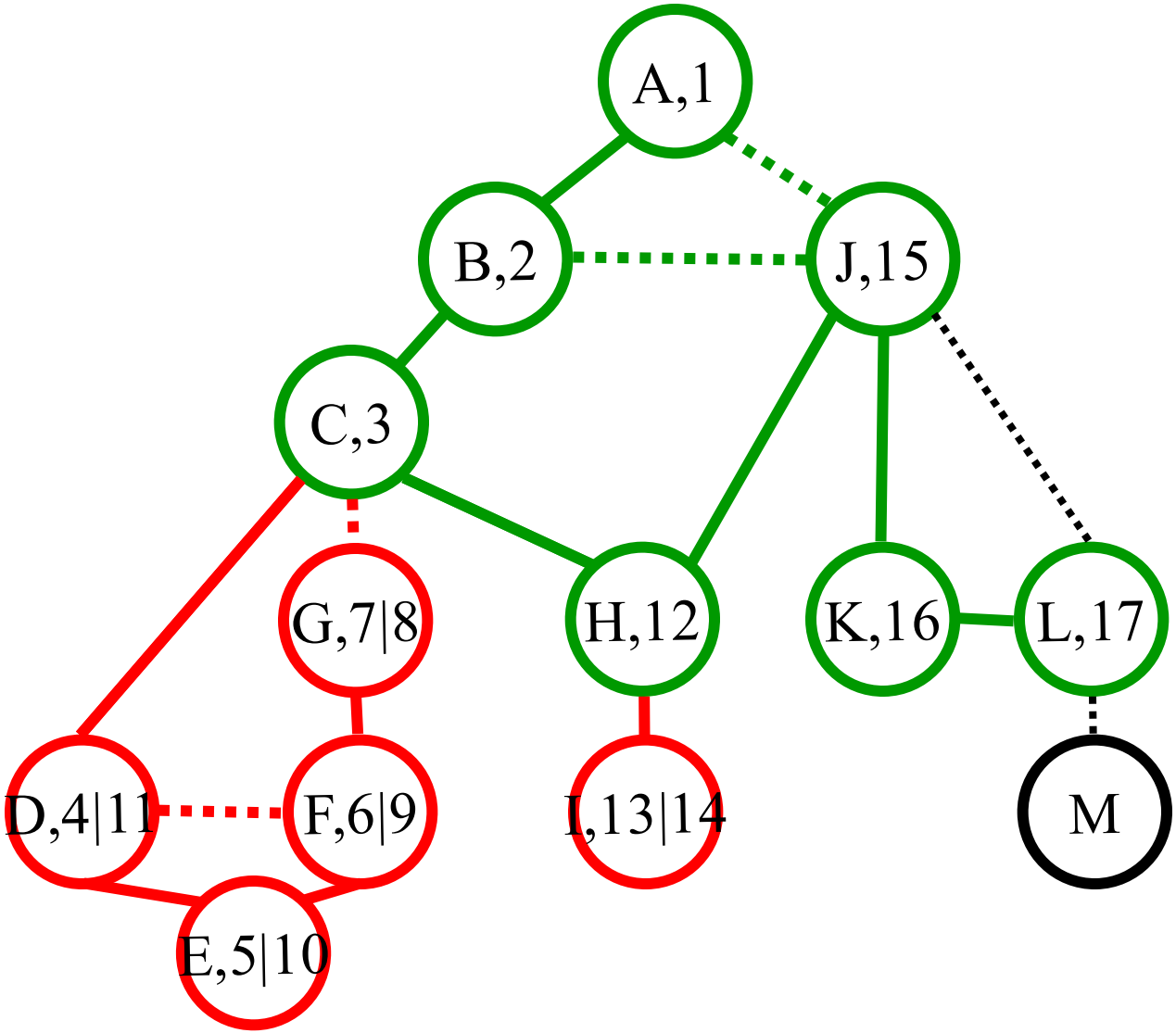
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,~~J~~)
J (~~A~~,~~B~~,~~H~~,K,L)
K (J,L)

st[] =
{A,B,C,H,J,
K}

Depth First Search (DFS)



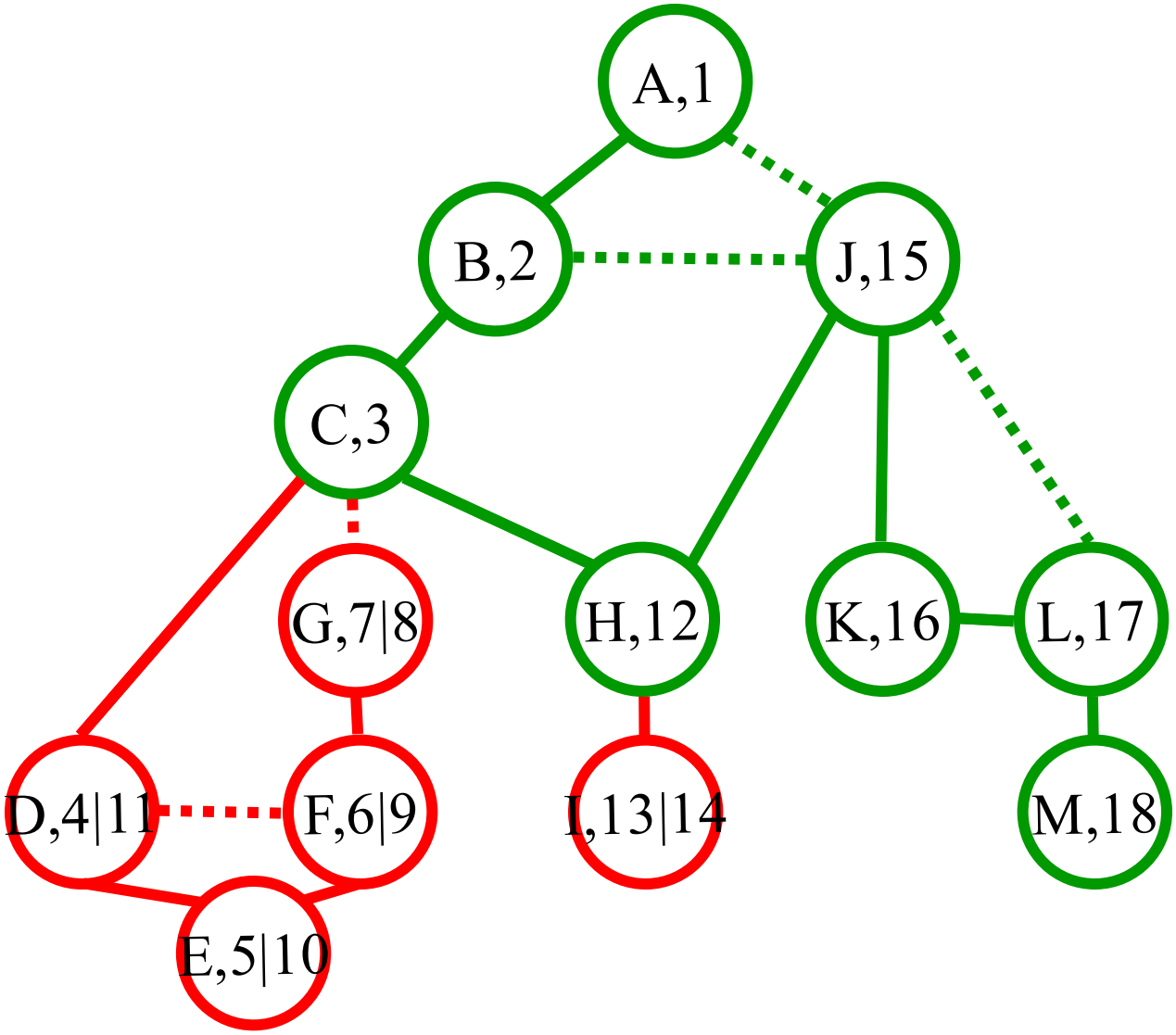
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,~~J~~)
J (~~A~~,~~B~~,~~H~~,K,L)
K (~~J~~,~~L~~)
L (J,K,M)

st[] =
{A,B,C,H,J,
K,L}

Depth First Search (DFS)



Color code:

Undiscovered

Discovered

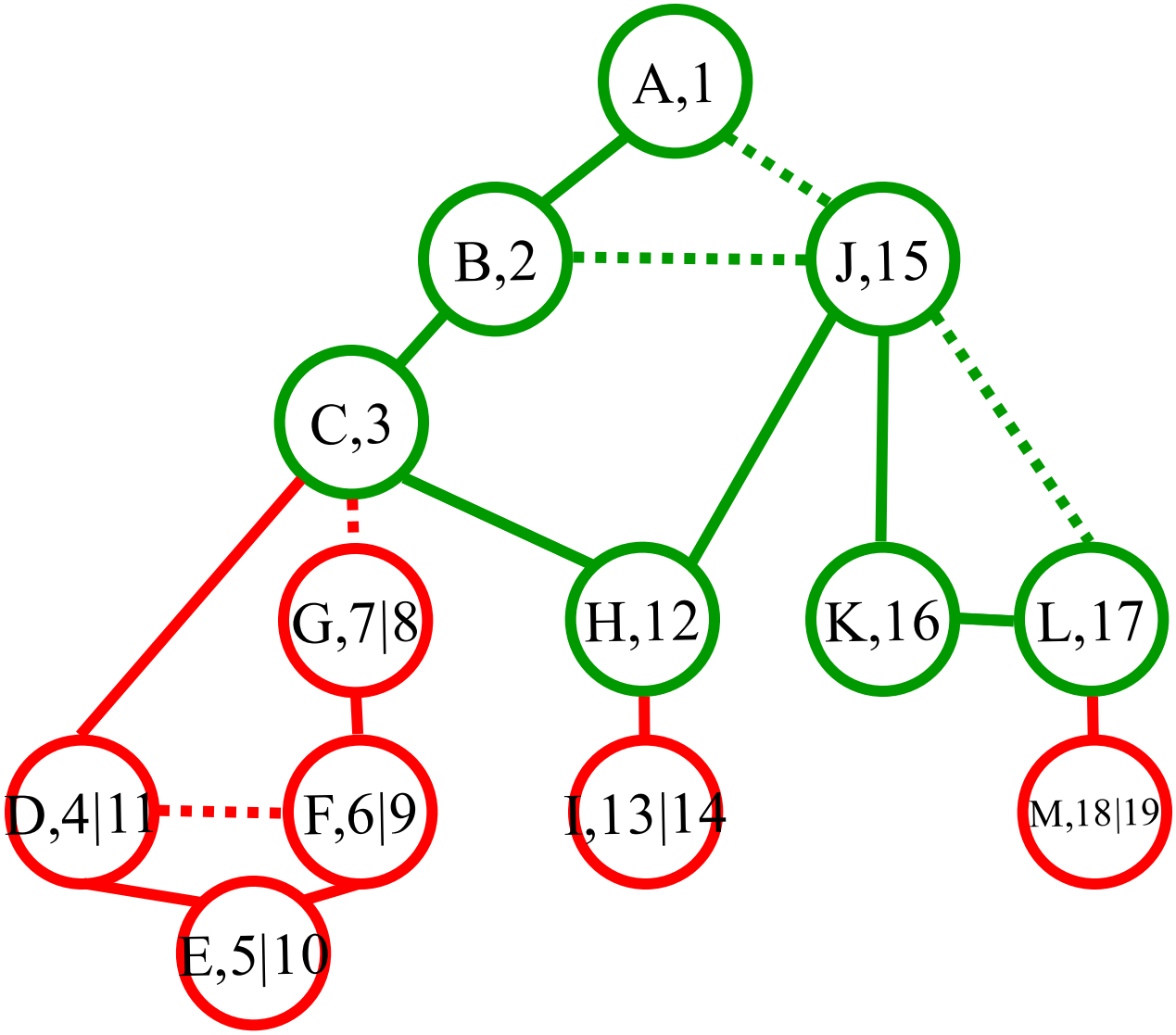
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,~~J~~)
J (~~A~~,~~B~~,~~H~~,K,L)
K (~~J~~,~~L~~)
L (~~J~~,~~K~~,~~M~~)
M(L)

st[] =
{A,B,C,H,J,
K,L,M}

Depth First Search (DFS)



Color code:

Undiscovered

Discovered

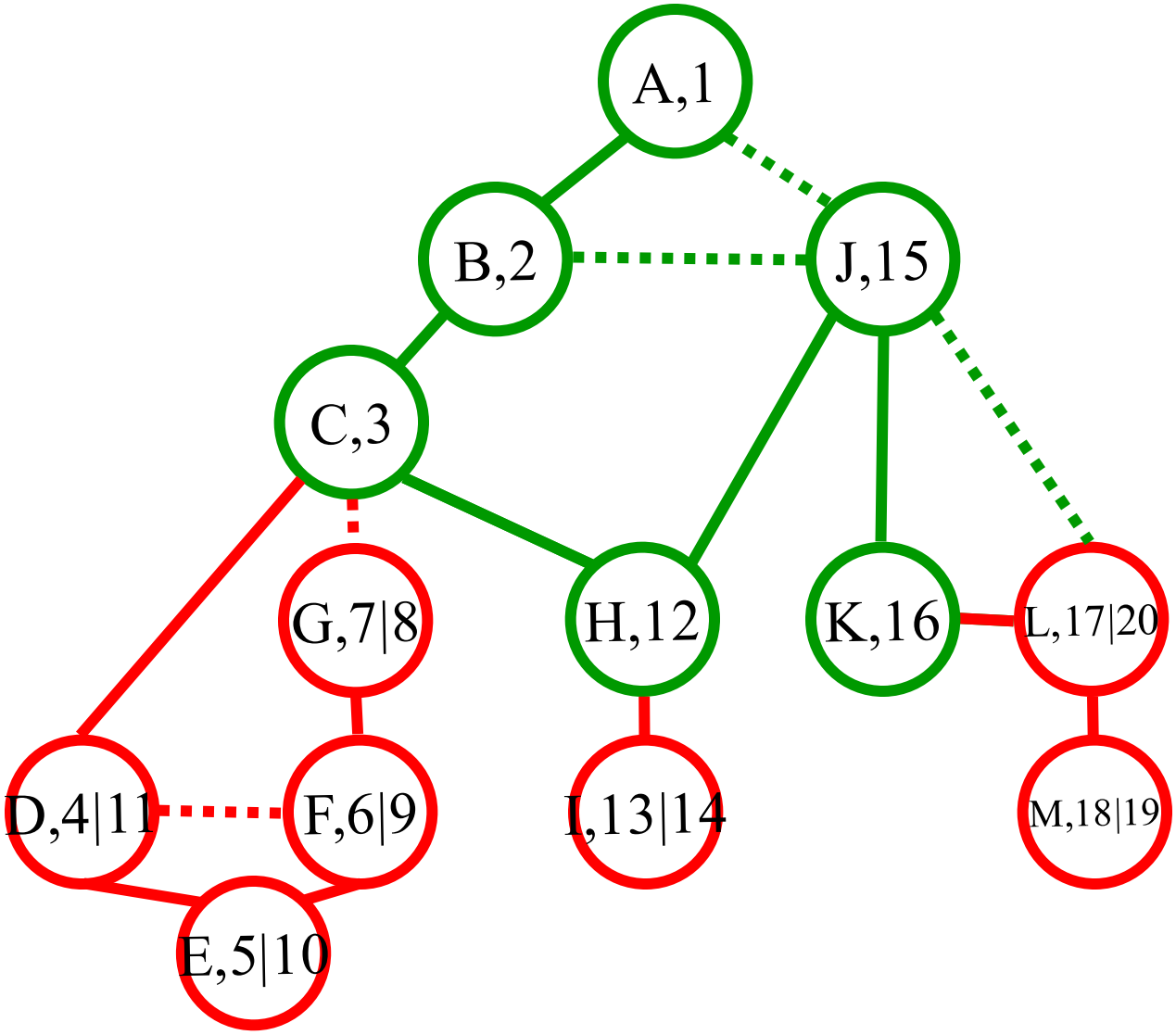
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,~~J~~)
J (~~A~~,~~B~~,~~H~~,K,L)
K (~~J~~,~~L~~)
L (~~J~~,~~K~~,~~M~~)

st[] =
{A,B,C,H,J,
K,L}

Depth First Search (DFS)



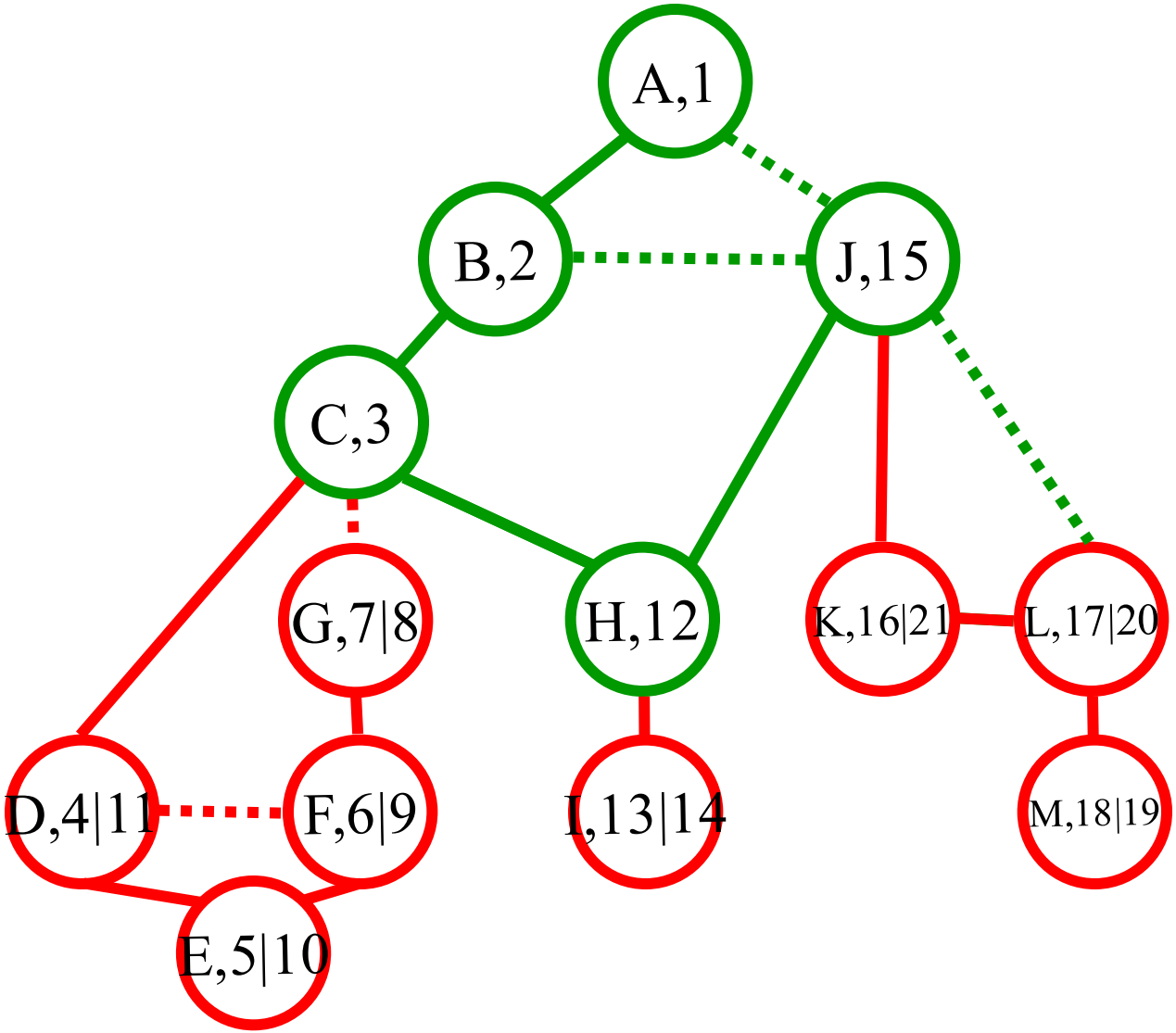
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,~~J~~)
J (~~A~~,~~B~~,~~H~~,K,L)
K (~~J~~,~~L~~)

st[] =
{A,B,C,H,J,
K}

Depth First Search (DFS)



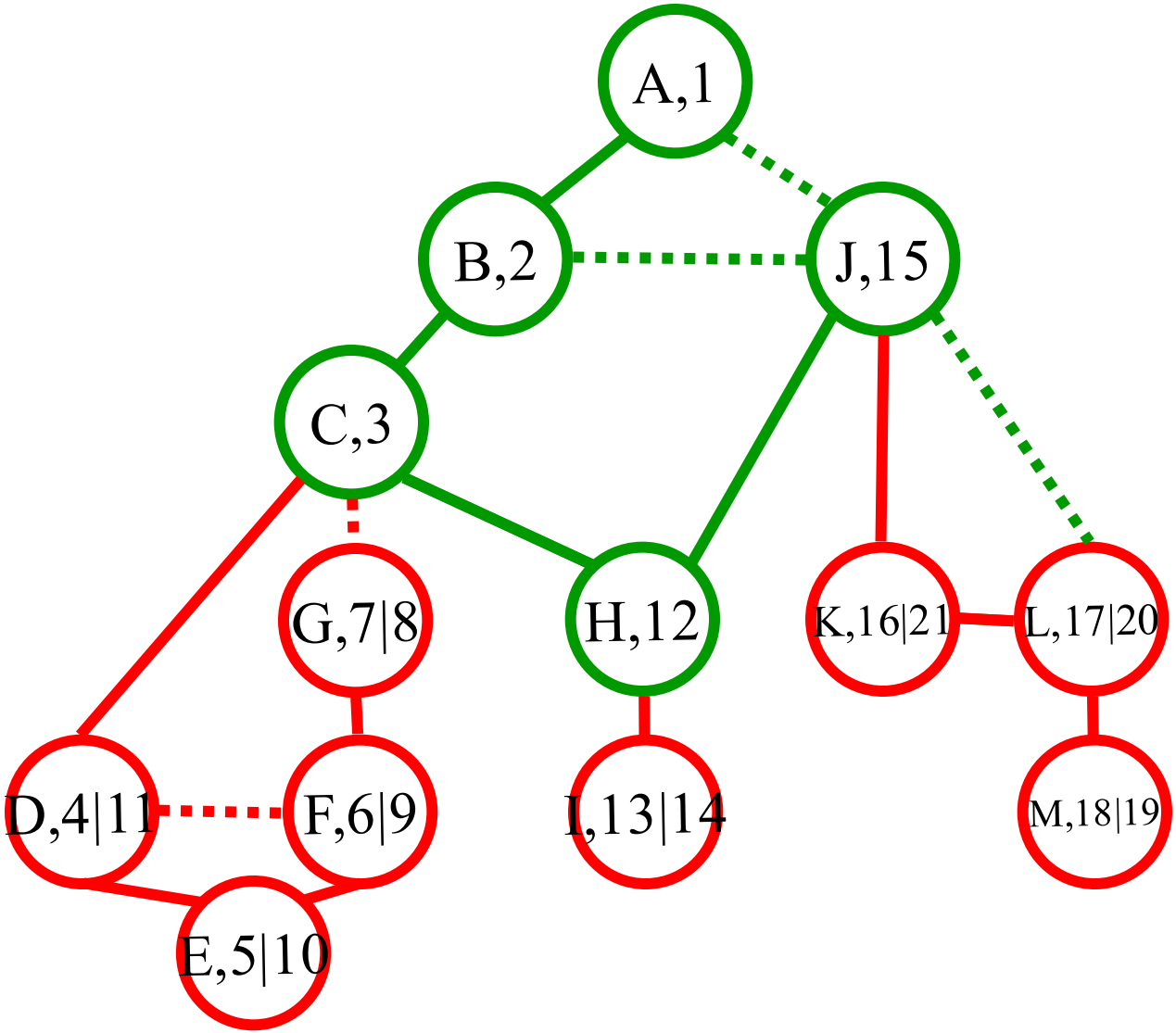
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,~~J~~)
J (~~A~~,~~B~~,~~H~~,K,L)

st[] =
{A,B,C,H,J}

Depth First Search (DFS)



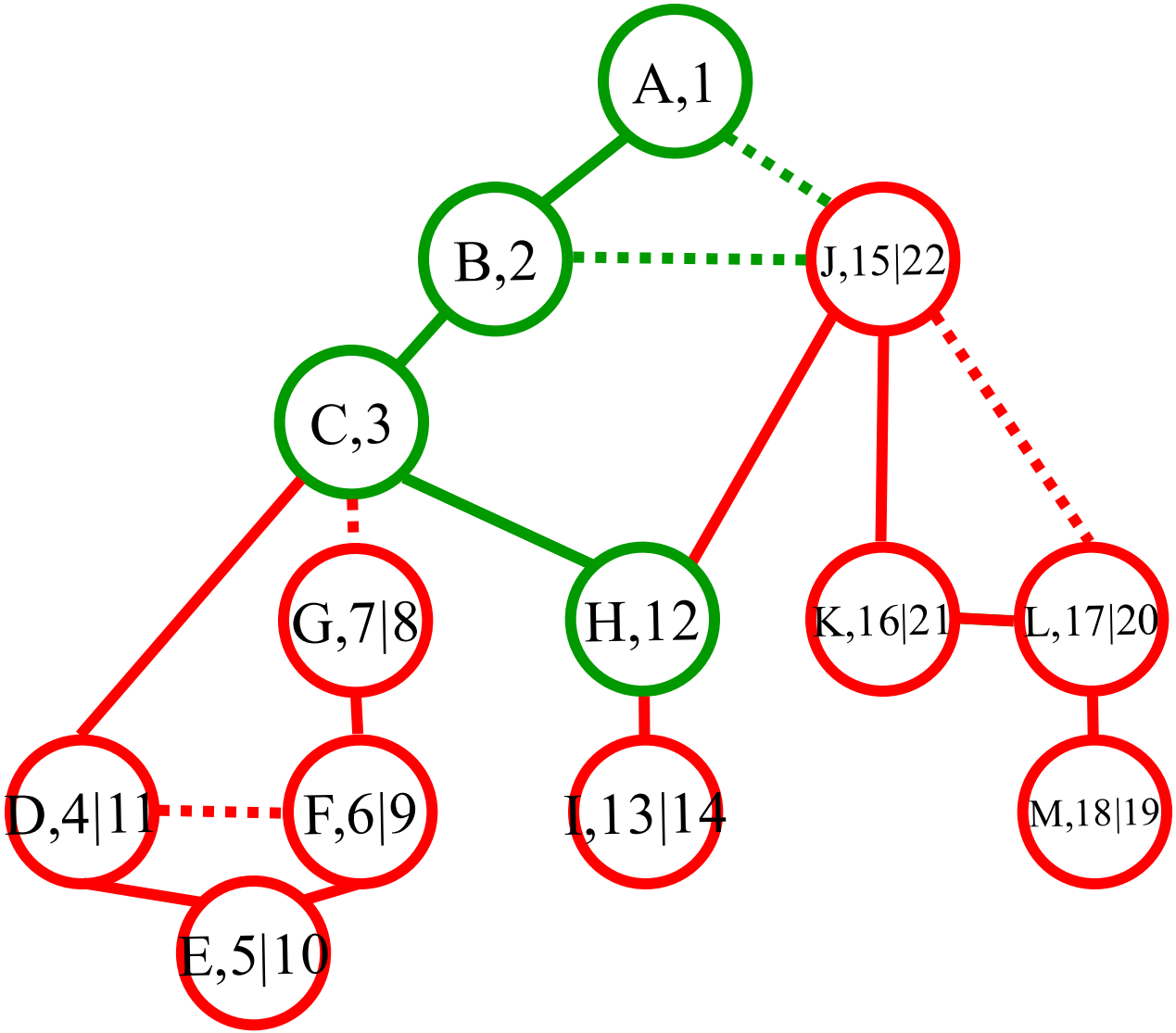
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,~~J~~)
J (~~A~~,~~B~~,~~H~~,~~K~~,~~L~~)

st[] =
{A,B,C,H,J}

Depth First Search (DFS)



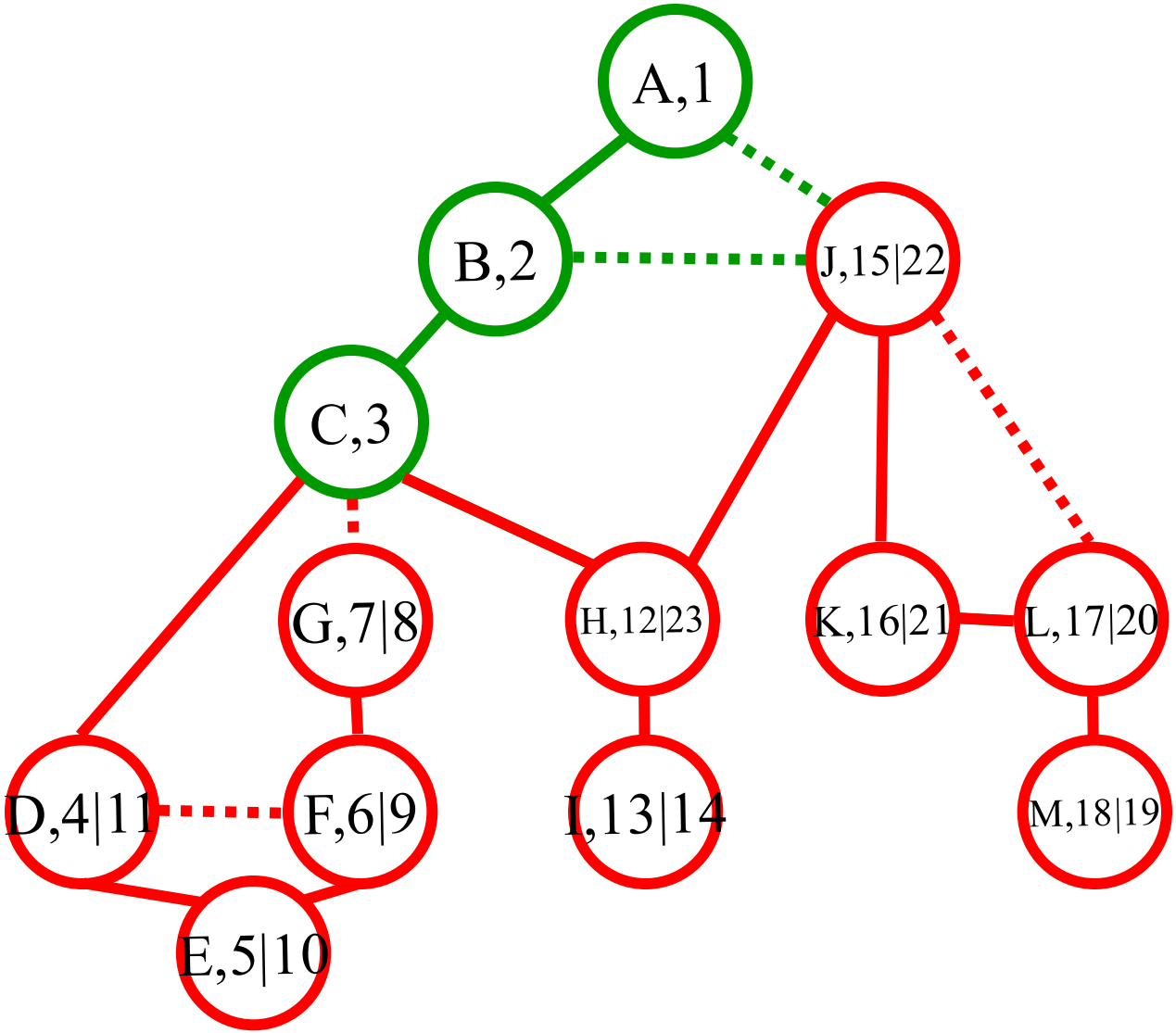
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,~~J~~)

st[] =
{A,B,C,H}

Depth First Search (DFS)



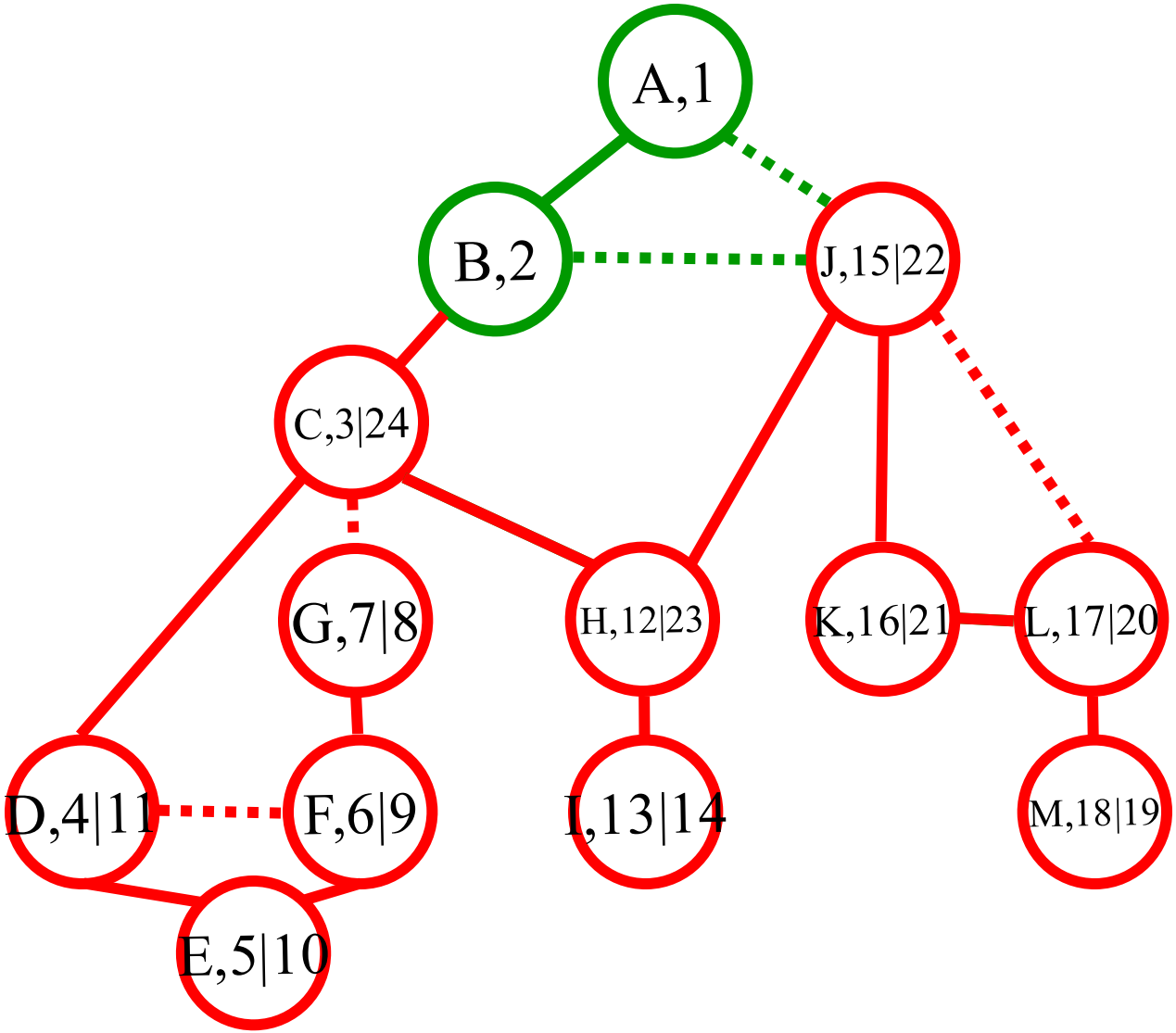
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)

st[] =
{A,B,C}

Depth First Search (DFS)



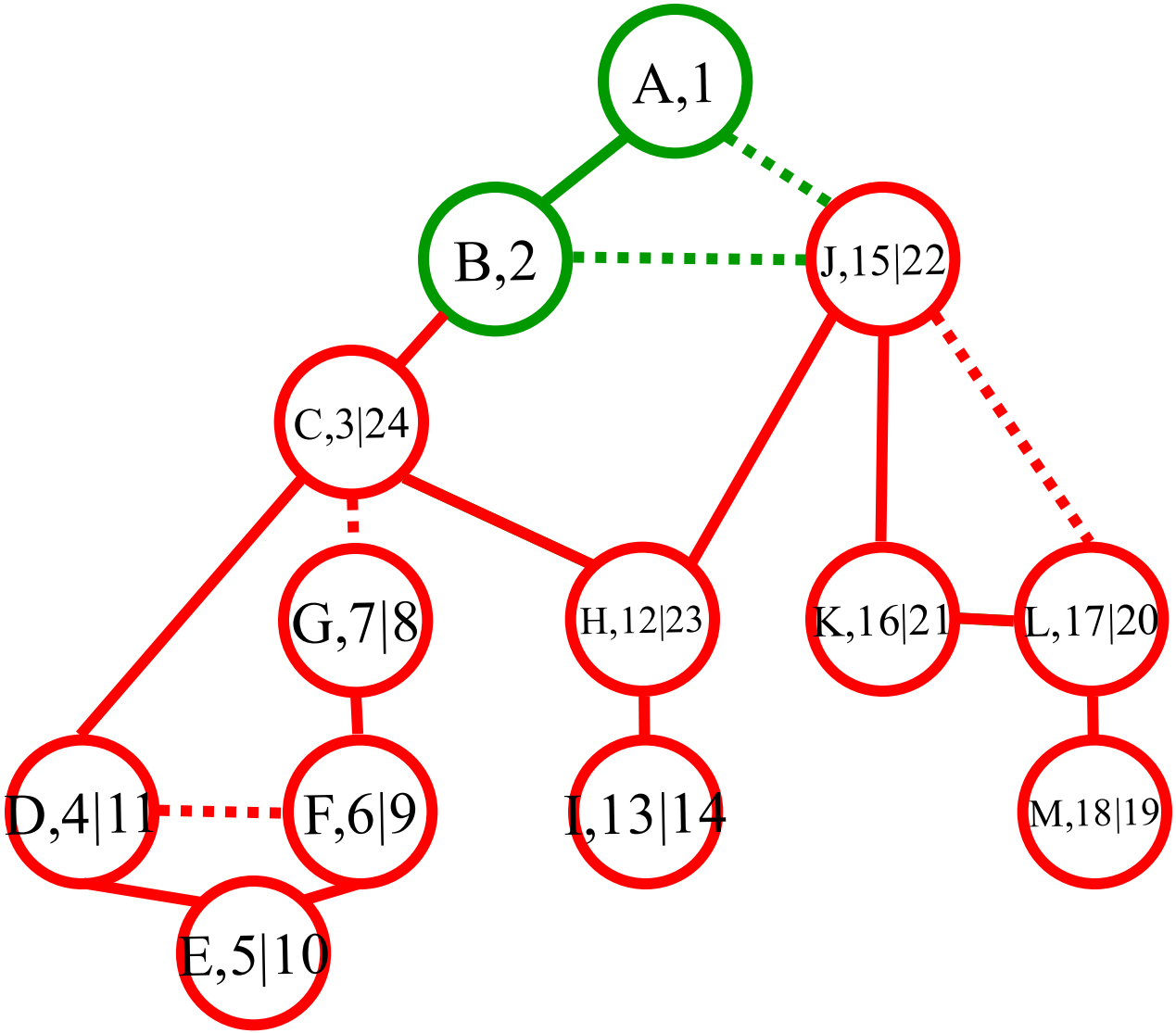
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,J)

st[] =
{A,B}

Depth First Search (DFS)



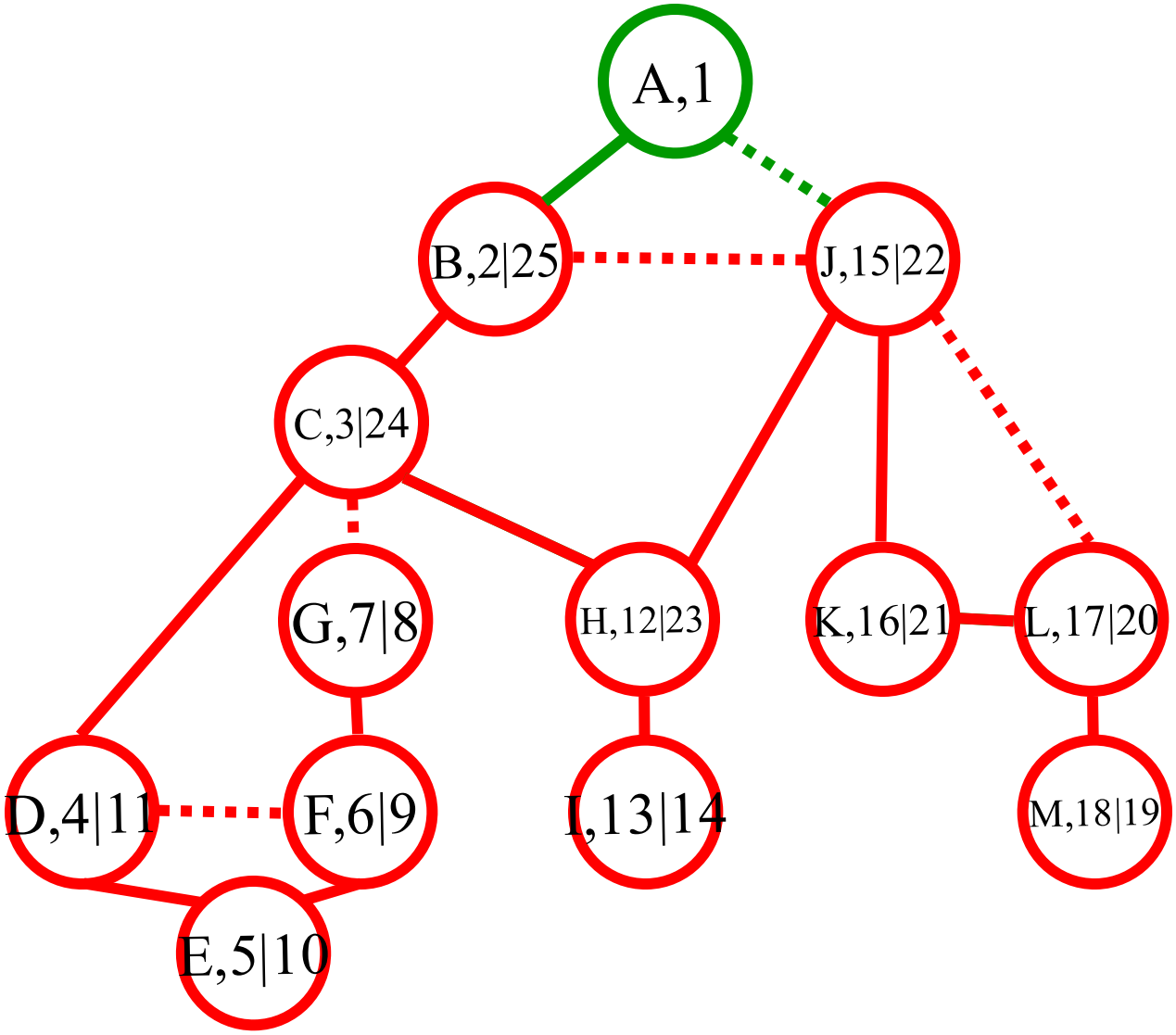
Color code:
Undiscovered
Discovered
fully-explored

Call Stack:
(Edge list)

A (~~B~~,J)
B (~~A~~,~~C~~,~~J~~)

st[] =
{A,B}

Depth First Search (DFS)



Color code:

Undiscovered

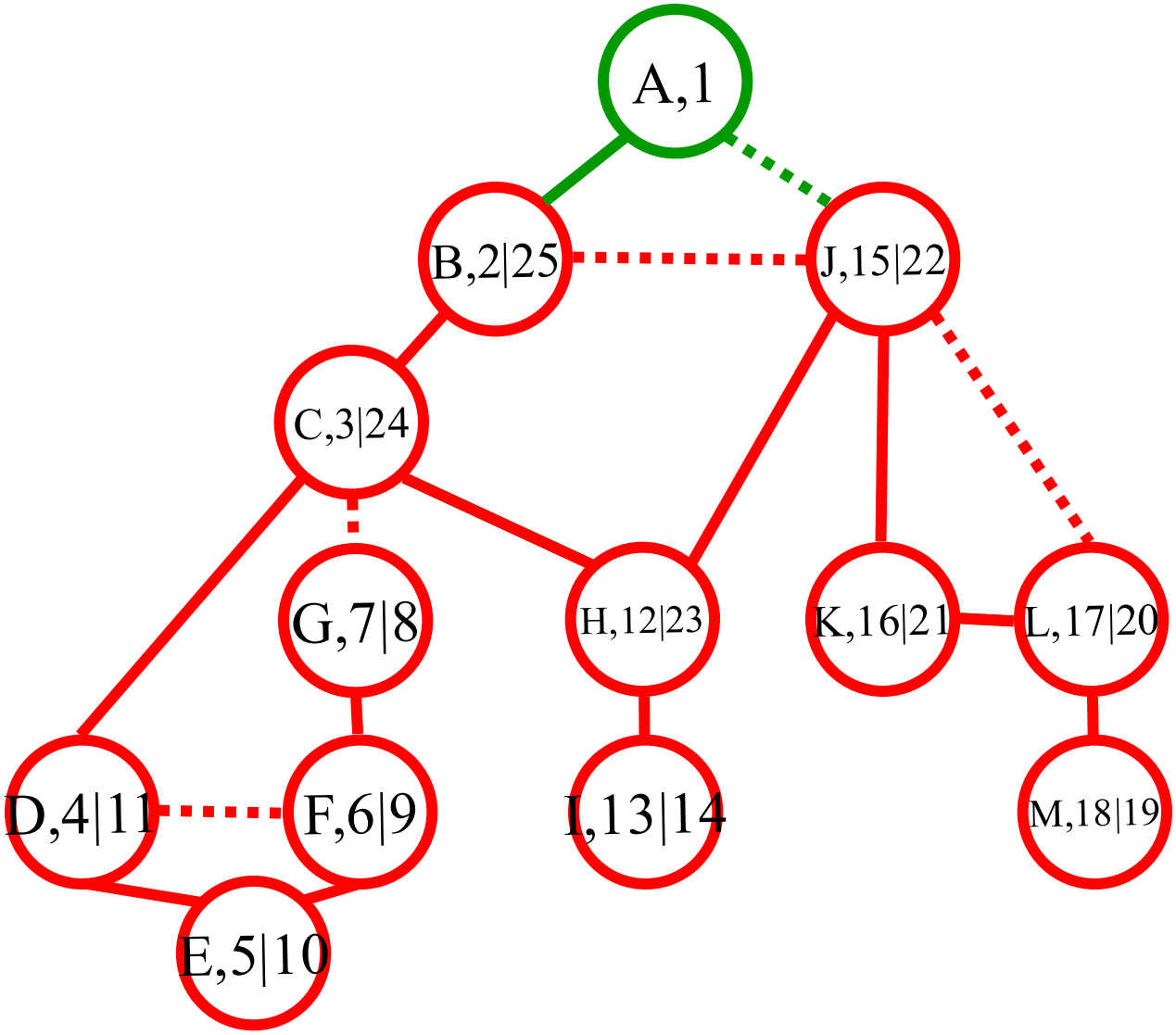
Discovered

fully-explored

Call Stack: (Edge list)
A (~~B~~,J)

st[] = {A}

Depth First Search (DFS)



Color code:

Undiscovered

Discovered

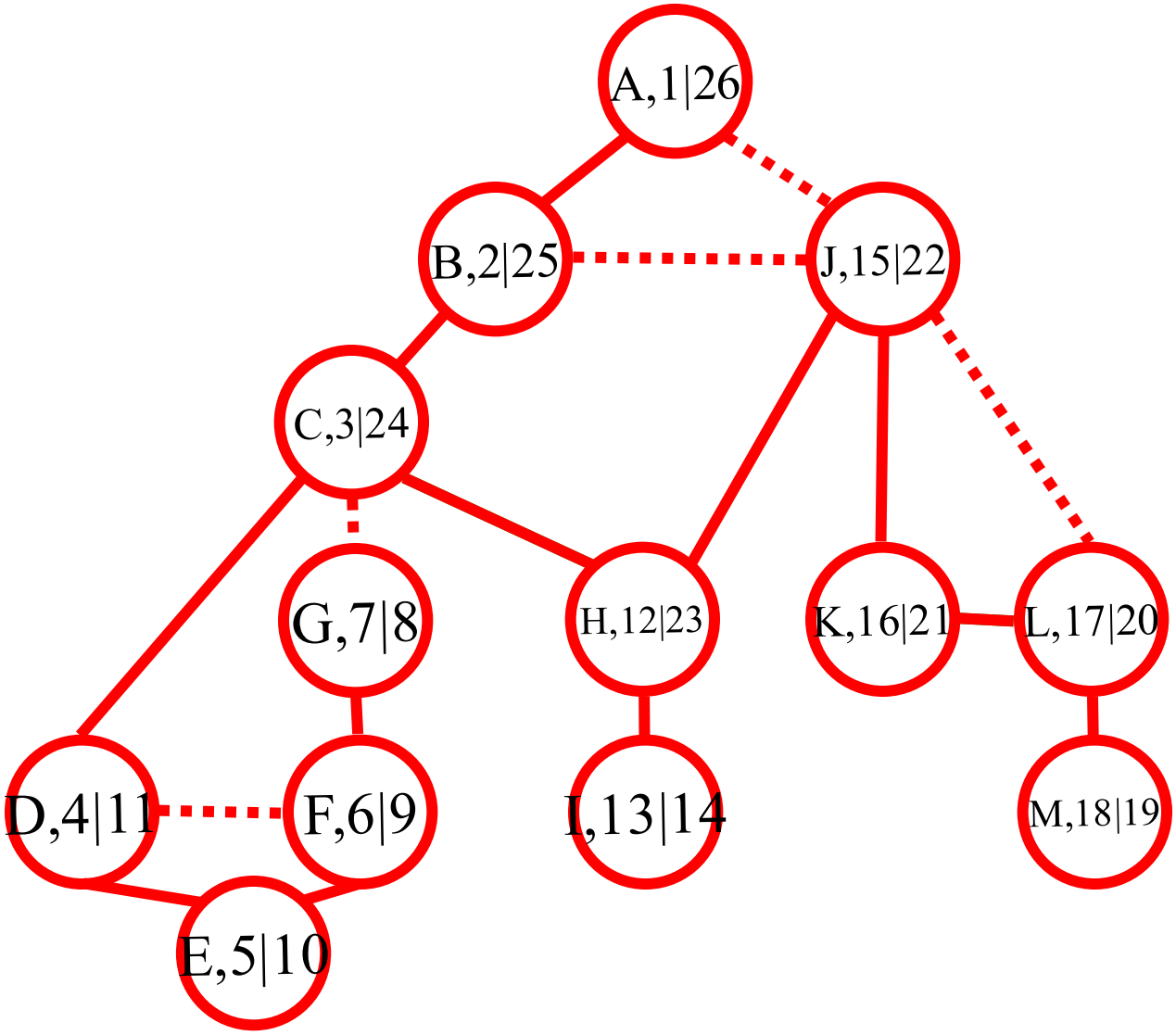
fully-explored

Call Stack:
(Edge list)

A (~~B~~,~~J~~)

st[] = {A}

Depth First Search (DFS)



Color code:

Undiscovered

Discovered

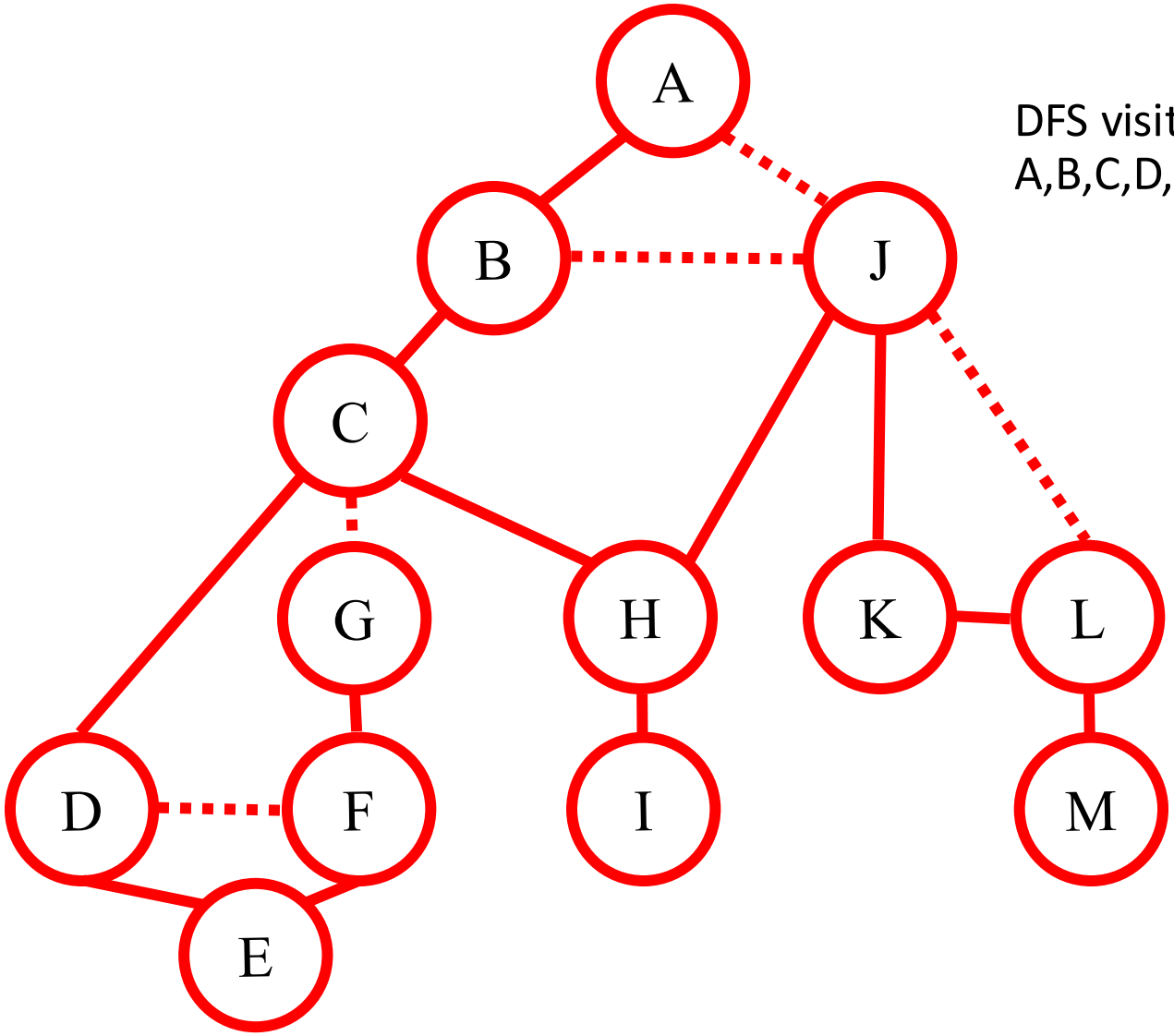
fully-explored

Call Stack:
(Edge list)

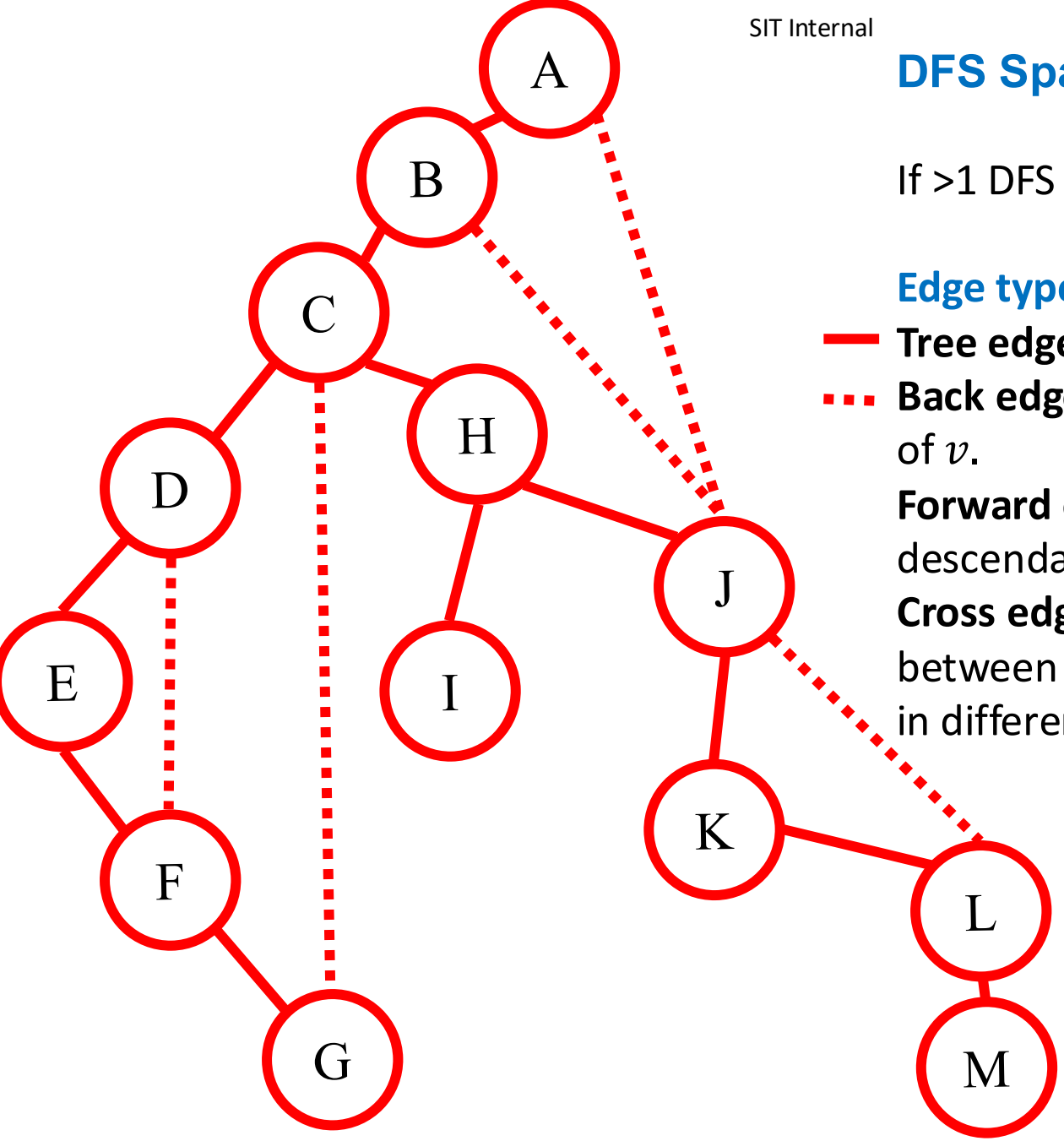
TA-DA!!

st[] = {}

Depth First Search (DFS)



DFS visit order:
A,B,C,D,E,F,G,H,I,J,K,L,M



DFS Spanning Tree

If >1 DFS tree, then **depth-first forest**.

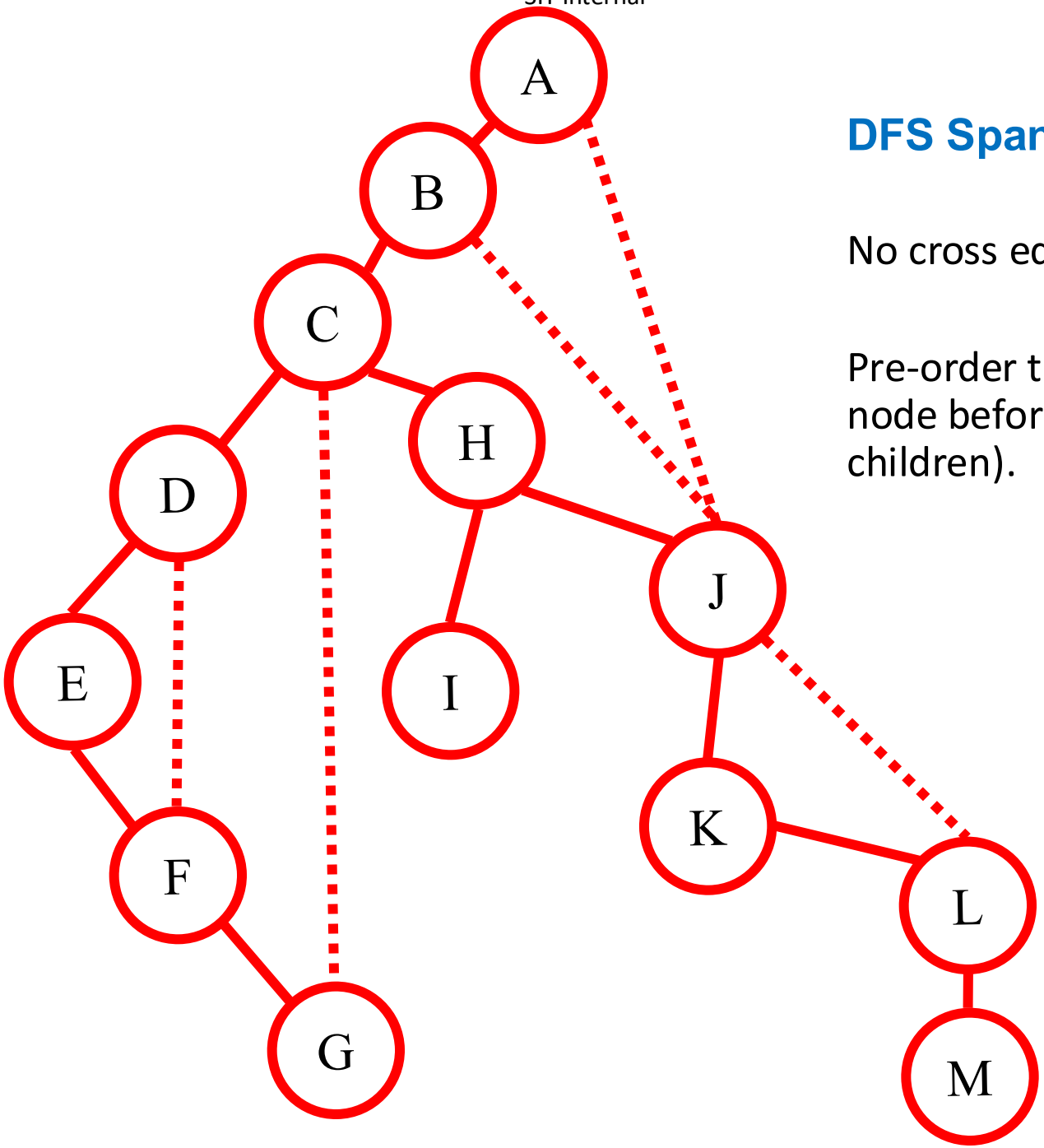
Edge types

Tree edge: found by exploring (u, v).

Back edge: (u, v), u is a descendant of v.

Forward edge: (u, v), v is a descendant of u, but not a tree edge.

Cross edge: Any other edge, go between vertices in same DFS tree or in different DFS trees.



DFS Spanning Tree

No cross edges!

Pre-order tree (visit a node before visiting its children).

DFS: Implementation and Analysis

Initialization: mark all vertices undiscovered, $time = 0$

DFS(u) - recursive

$time = time + 1$

$u.d = time$ **// discover/visit u**

Mark u discovered

for each edge $\{u, x\}$

if (x is undiscovered)

Mark x discovered

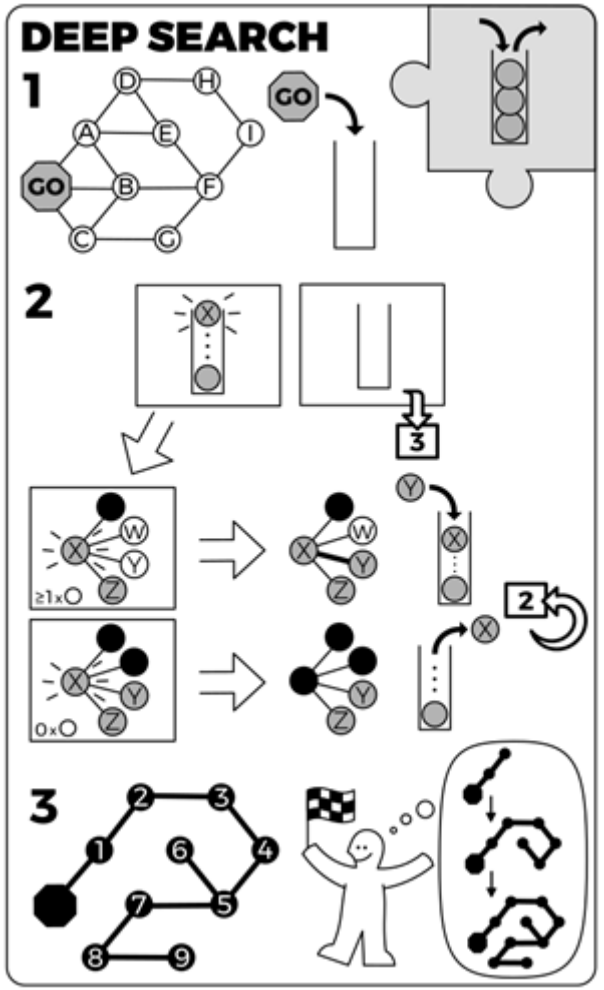
$x.\text{parent} = u$

DFS(x) **// time counts**

Mark u fully-explored

$time = time + 1$

$u.f = time$ **// finish u, fully explored**



Total time: $\Theta(V + E)$, not just big-O because guaranteed to examine every vertex and edge.

DFS: C++ Implementation

```cpp
enum Color { WHITE, GRAY, BLACK };

struct Vertex {
    Color color;
    int discoveryTime;
    int finishTime;
};
```

```cpp
int numVertices;
vector<list<int>> adjList;
vector<Vertex> vertices;
int time;

void DFS_VISIT(int u) {
    time++;
    vertices[u].discoveryTime = time;
    vertices[u].color = GRAY;

    for (int v : adjList[u]) {
        if (vertices[v].color == WHITE) {
            DFS_VISIT(v);
        }
    }

    vertices[u].color = BLACK;
    time++;
    vertices[u].finishTime = time;
}
```

DFS: Wrap-up

Like BFS(s):

- DFS(s) visits x if there is a path in G from s to x.

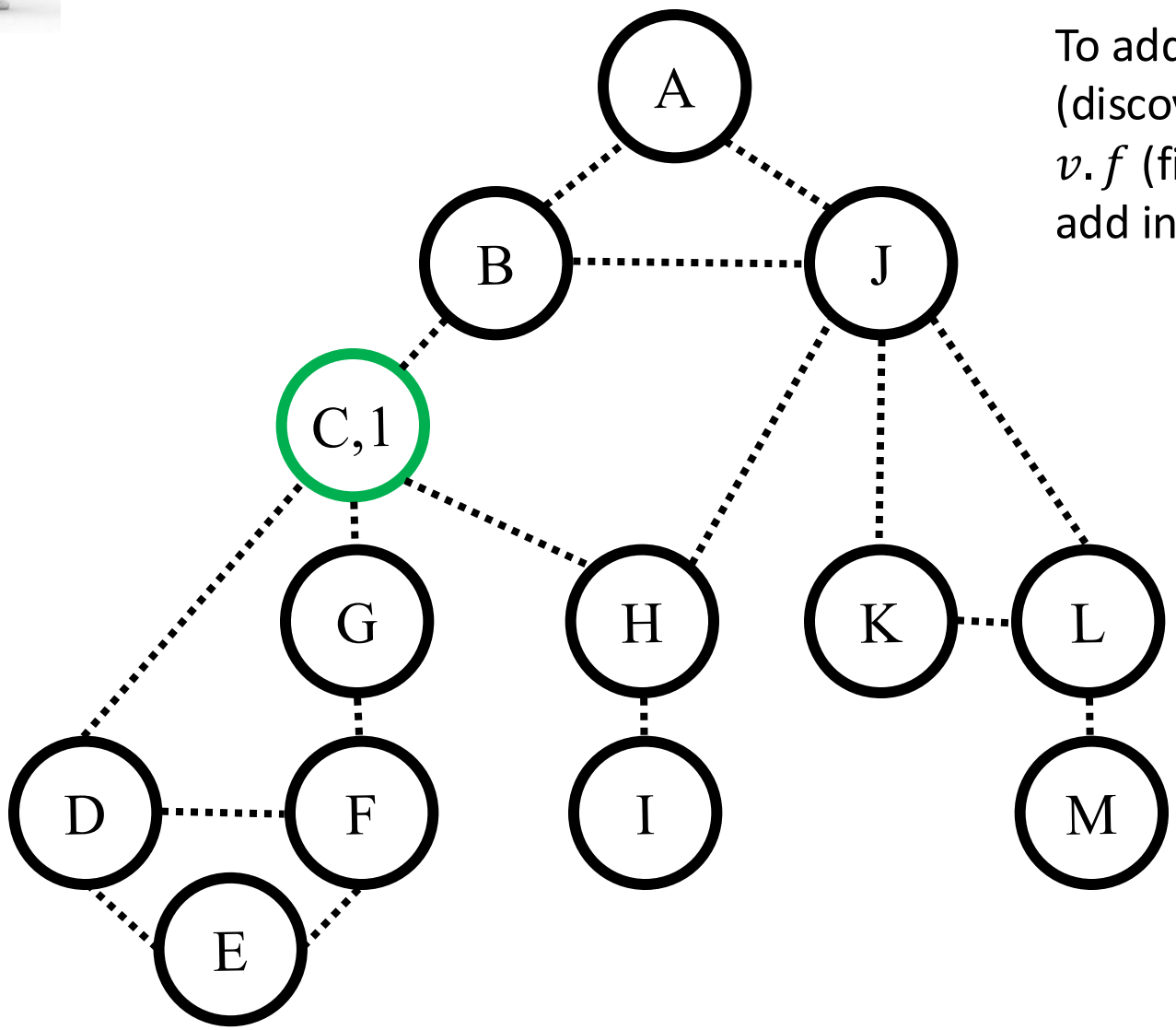
  So, we can also use DFS to find connected components.
- Edges into the-undiscovered vertices define a tree – the "spanning tree" of G.

Unlike the BFS spanning tree:

- The DFS spanning tree is not minimum depth.
- Its levels don't reflect min distance from the root.
- Non-tree edges never join vertices on the same or adjacent levels.

Applications: reachability, topological sort (lab), connected component (lab).

Exercise: DFS from Different Source Vertices



To add in the $v.d$ (discovered time) and $v.f$ (finish time) values; add in edge types.

Summary

- Terminology: vertices, edges, degree, paths, spanning tree…
- Edges and vertices: $|E| = O(V^2)$ in general graphs, $|V| - 1$ for trees
- Representations: adjacency list (for sparse graph), adjacency matrix
- BFS: layers, queue, reachable, shortest paths, all edges go to same or adjacent layer
- DFS: recursion/stack, reachable, all edges ancestor/descendant
- Techniques: discover/visit then explore

Graph Traversal Algorithm	**Type**	**Time Complexity**
Breadth-First Search (BFS)	Iterative	O(V + E)
Depth-First Search (DFS)	Recursive / Iterative	Θ(V + E)