CSE 421: Introduction to Algorithms

Breadth First Search

Yin Tat Lee

Properties of BFS

Claim: All nontree edges join vertices on the same or adjacent levels of the tree

Proof: Consider an edge $\{x, y\}$
Say x is first discovered and it is added to level i.
We show y will be at level i or $i + 1$

This is because when vertices incident to x are considered in the loop, if y is still undiscovered, it will be discovered and added to level $i + 1$.

Properties of BFS

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 $s = v_0, v_1, \dots, v_i = v$ is the shortest path to v.

By previous claim,

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

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

This proves the lemma.

Why Trees?

Trees are simpler than graphs

Many statements can be proved on trees by induction

So, computational problems on trees are simpler than general graphs

This is often a good way to approach a graph problem:

- Find a "nice" tree in the graph, i.e., one such that non-tree edges have some simplifying structure
- Solve the problem on the tree
- Use the solution on the tree to find a “good” solution on the graph

CSE 421: Introduction to Algorithms

Application of BFS

Yin Tat Lee

BFS Application: Connected Component

We want to answer the following type questions (fast):
Given vertices u, v is there a path from u to v in G?

Idea: Create an array A such that
For all u in the same connected component, $A[u]$ is same.

Therefore, question reduces to

If $A[u] = A[v]$?

BFS Application: Connected Component

Initial State: All vertices undiscovered, $c = 0$

For v = 1 to n do
  If state(v) != fully-explored then
    Run BFS(v)
    Set $A[u] \leftarrow c$ for each u found in BFS(v)
    $c = c + 1$

Note: We no longer initialize to undiscovered in the BFS subroutine

Total Cost: $O(m + n)$

In every connected component with n_i vertices and m_i edges BFS takes time $O(m_i + n_i)$.

Note: one can use DFS instead of BFS.

Connected Components

Lesson: We can execute any algorithm on disconnected graphs by running it on each connected component.

We can use the previous algorithm to detect connected components.

There is no overhead, because the algorithm runs in time $O(m+n)$.

So, from now on, we can (almost) always assume the input graph is connected.

Cycles in Graphs

Claim: If an n vertices graph G has at least n edges, then it has a cycle.

Proof: If G is connected, then it cannot be a tree. Because every tree has $n-1$ edges. So, it has a cycle.

Suppose G is disconnected. Say connected components of G have $n_1, \dots, n_k$vertices where $n_1 + \cdots + n_k = n$

Since G has $\geq n$ edges, there must be some i such that a component has n_i vertices with at least n_i edges.

Therefore, in that component we do not have a tree, so there is a cycle.

Bipartite Graphs

Definition: An undirected graph $G = (V, E)$ is bipartite

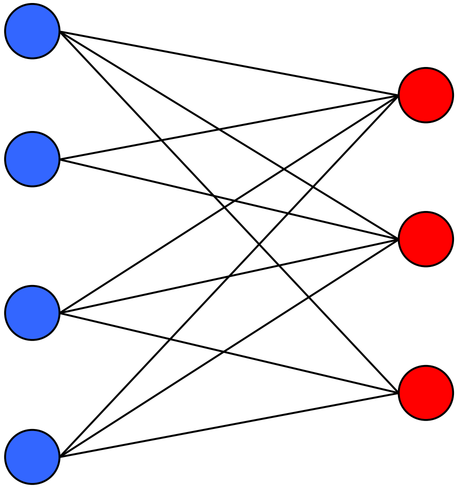
if you can partition the node set into 2 parts (say, blue/red or left/right) so that

all edges join nodes in different parts

i.e., no edge has both ends in the same part.

Application:

- Scheduling: machine=red, jobs=blue
- Stable Matching: men=blue, wom=red



a bipartite graph

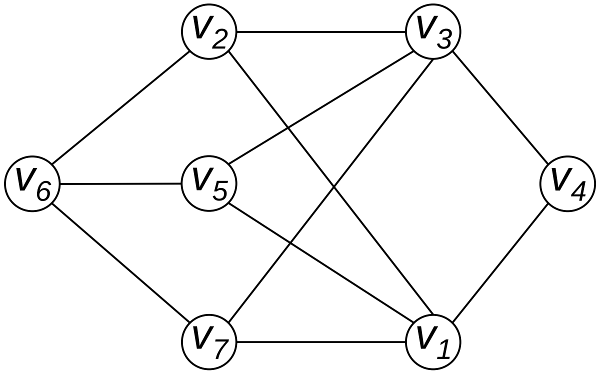
Testing Bipartiteness

Problem: Given a graph G, is it bipartite?

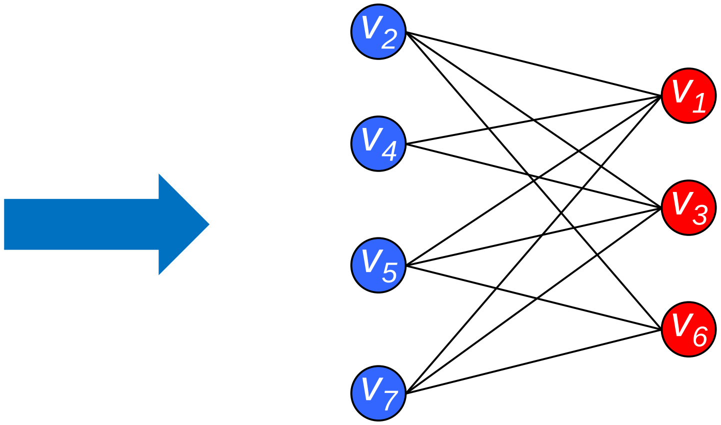
Many graph problems become:

- Easier/Tractable if the underlying graph is bipartite (matching)

Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



a bipartite graph G

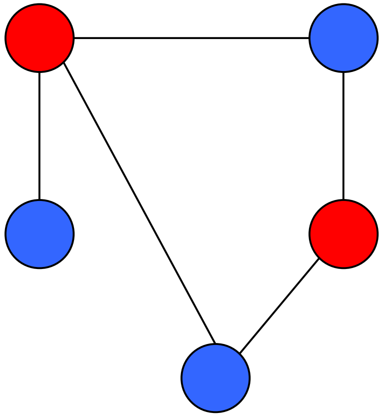


another drawing of G

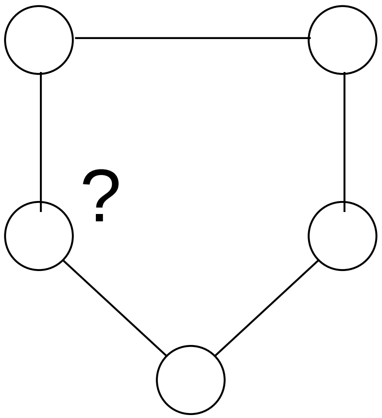
An Obstruction to Bipartiteness

Lemma: If G is bipartite, then it does not contain an odd length cycle.

Proof: We cannot 2-color an odd cycle, let alone G.



bipartite
(2-colorable)



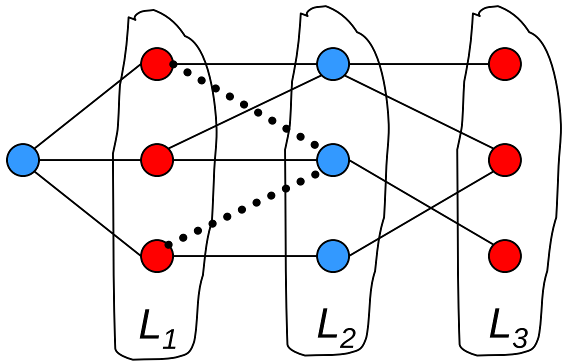
not bipartite
(not 2-colorable)

A Characterization of Bipartite Graphs

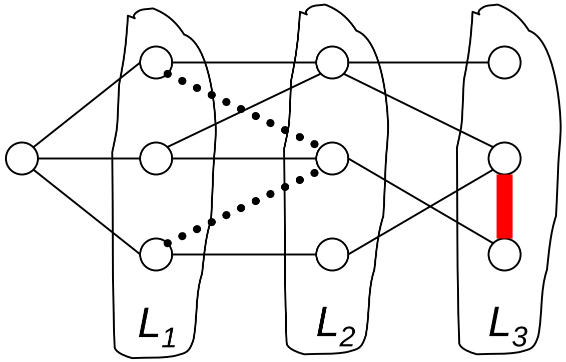
Lemma: Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS(s). Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



Case (i)



Case (ii)

A Characterization of Bipartite Graphs

Lemma: Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS(s). Exactly one of the following holds.

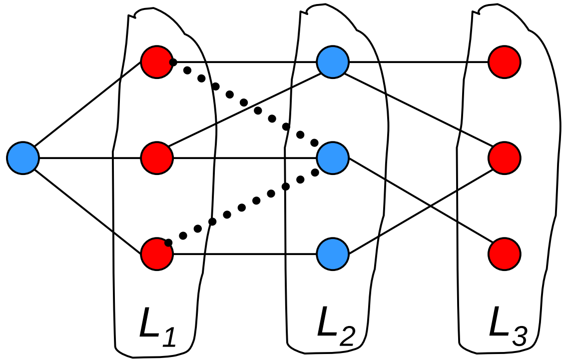
(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Proof. (i)

Suppose no edge joins two nodes in the same layer.

By previous lemma, all edges join nodes on adjacent levels.



Case (i)

Bipartition:

blue = nodes on odd levels,

red = nodes on even levels.

A Characterization of Bipartite Graphs

Lemma: Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS(s). Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Proof. (ii)

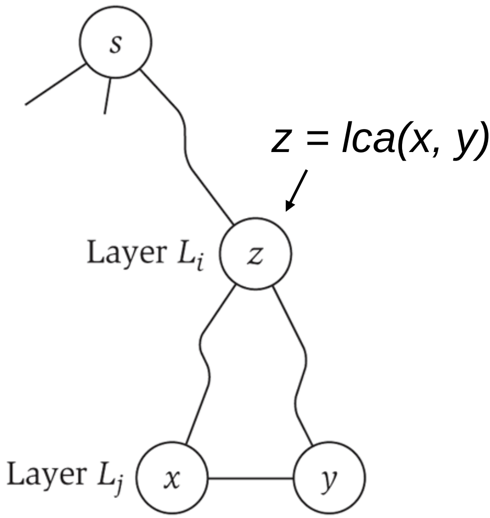
Suppose $\{x, y\}$ is an edge & x, y in same level L_j.

Let $z =$ their lowest common ancestor in BFS tree.

Let L_i be level containing z.

Consider cycle that takes edge from x to y, then tree from y to z, then tree from z to x.

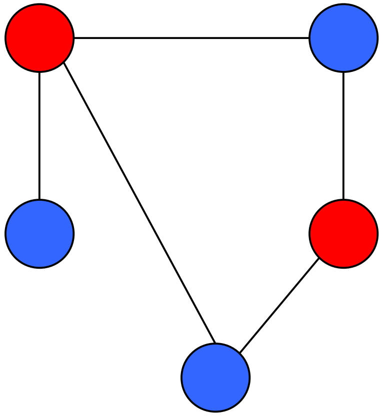
Its length is $1 + (j - i) + (j - i)$, which is odd.



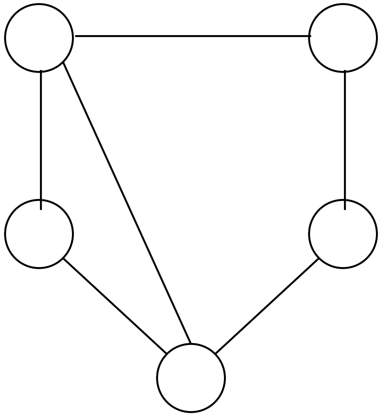
Obstruction to Bipartiteness

Corollary: A graph G is bipartite if and only if it contains no odd length cycles.

Furthermore, one can test bipartiteness using BFS.



bipartite
(2-colorable)



not bipartite
(not 2-colorable)

Summary of last lecture

- BFS(s) implemented using queue.
- Edges into then-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:
  - Shortest Path
  - Connected component
  - Test bipartiteness / 2-coloring

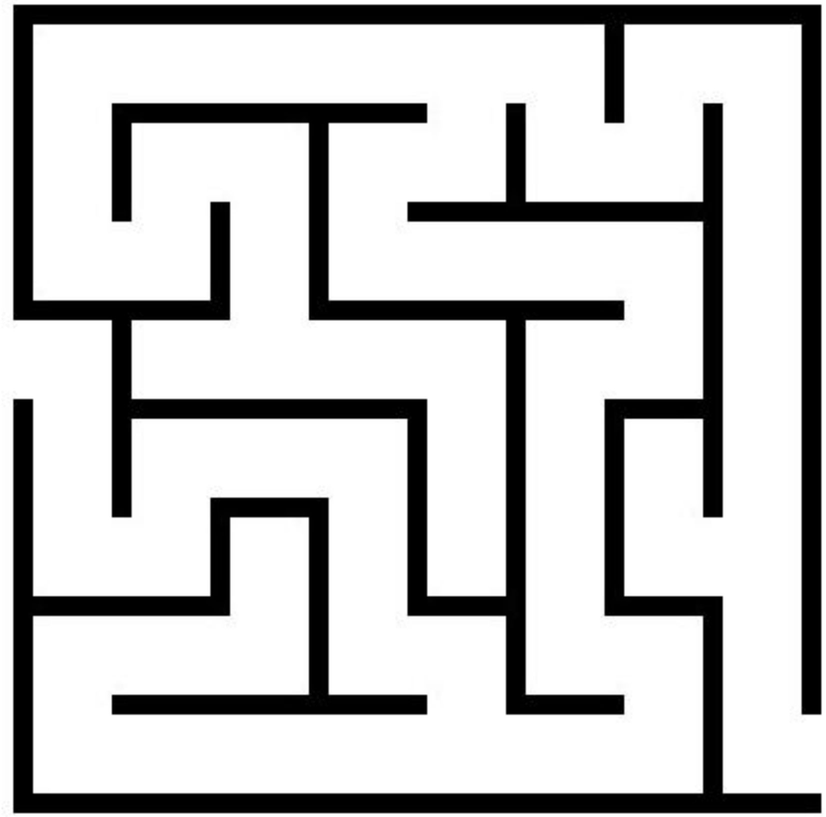
CSE 421

Depth First Search

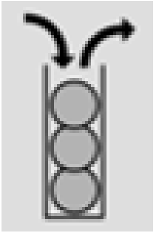
Yin Tat Lee

Depth First Search

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



DFS(s) – Recursive version

Initialization: mark all vertices undiscovered

DFS(v)

  Mark v discovered

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

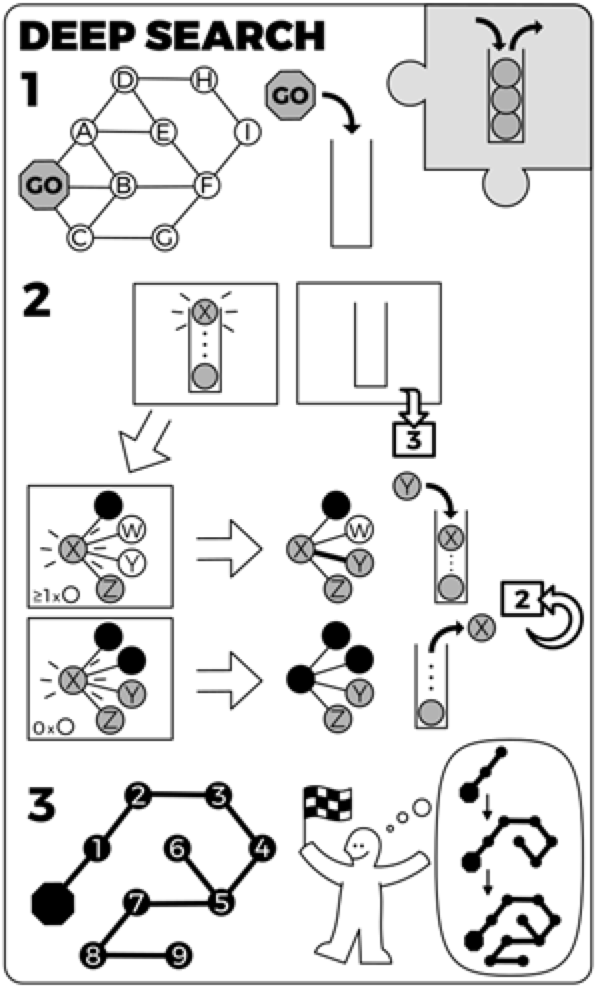
    if (x is undiscovered)

      Mark x discovered

      $x \rightarrow \text{parent} = u$

      DFS(x)

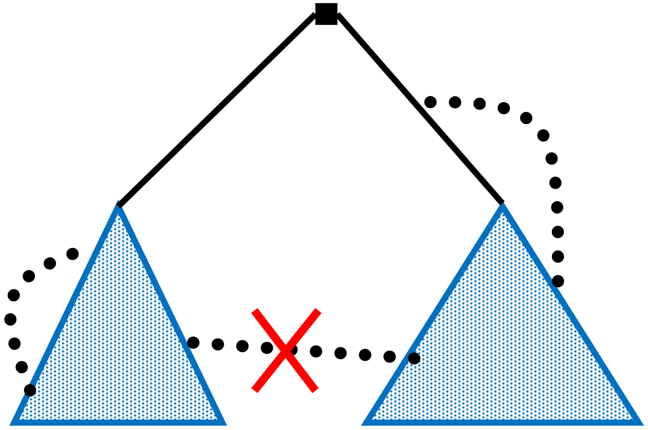
  Mark v fully-discovered



Non-Tree Edges in DFS

BFS tree ≠ DFS tree, but, as with BFS, DFS has found a tree in the graph s.t. non-tree edges are "simple" in some way.

All non-tree edges join a vertex and one of its descendants/ancestors in the DFS tree



DFS(A)

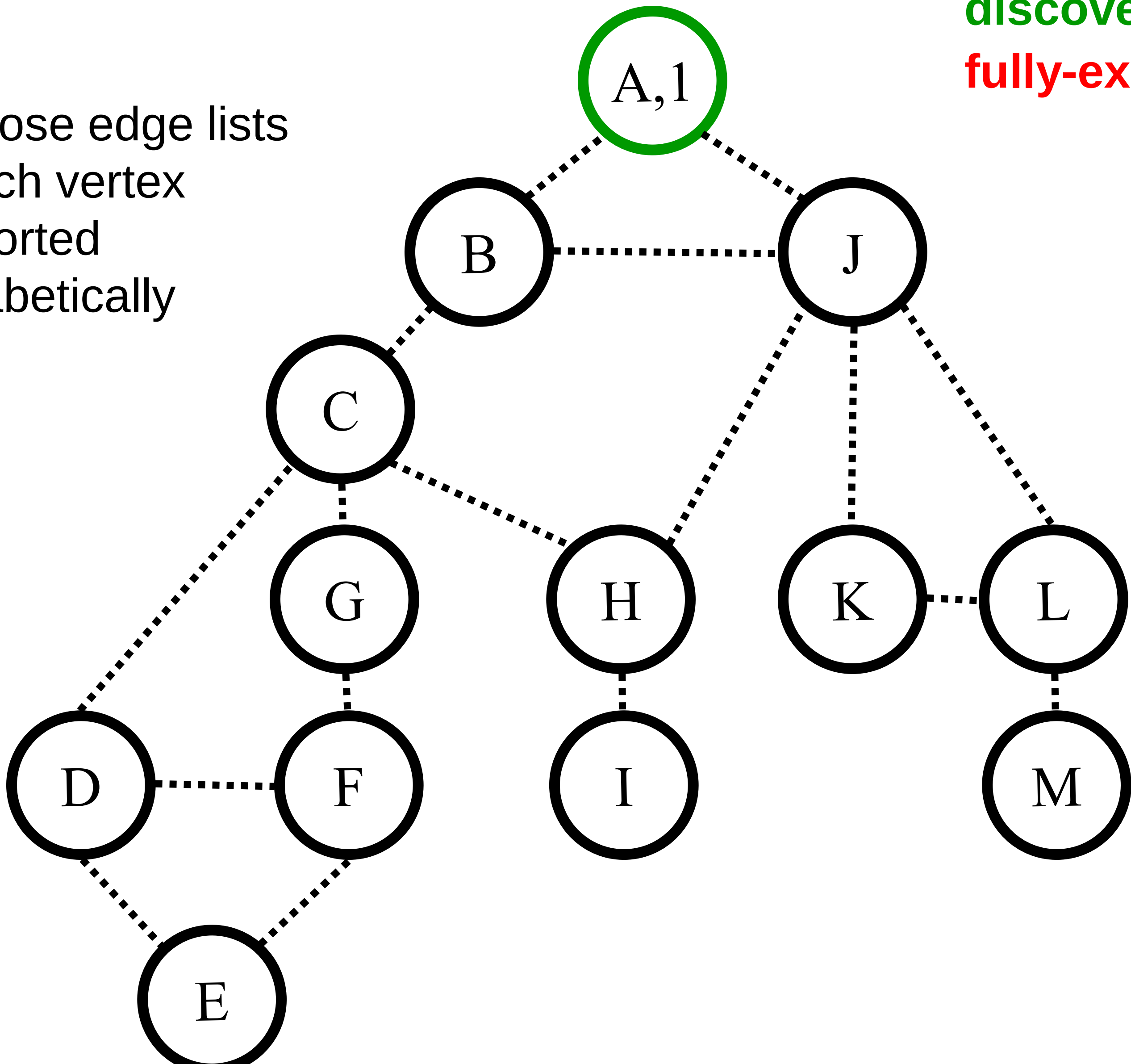
Color code:

undiscovered

discovered

fully-explored

Suppose edge lists at each vertex are sorted alphabetically



Call Stack (Edge list):

A (B,J)

st[] = {1}

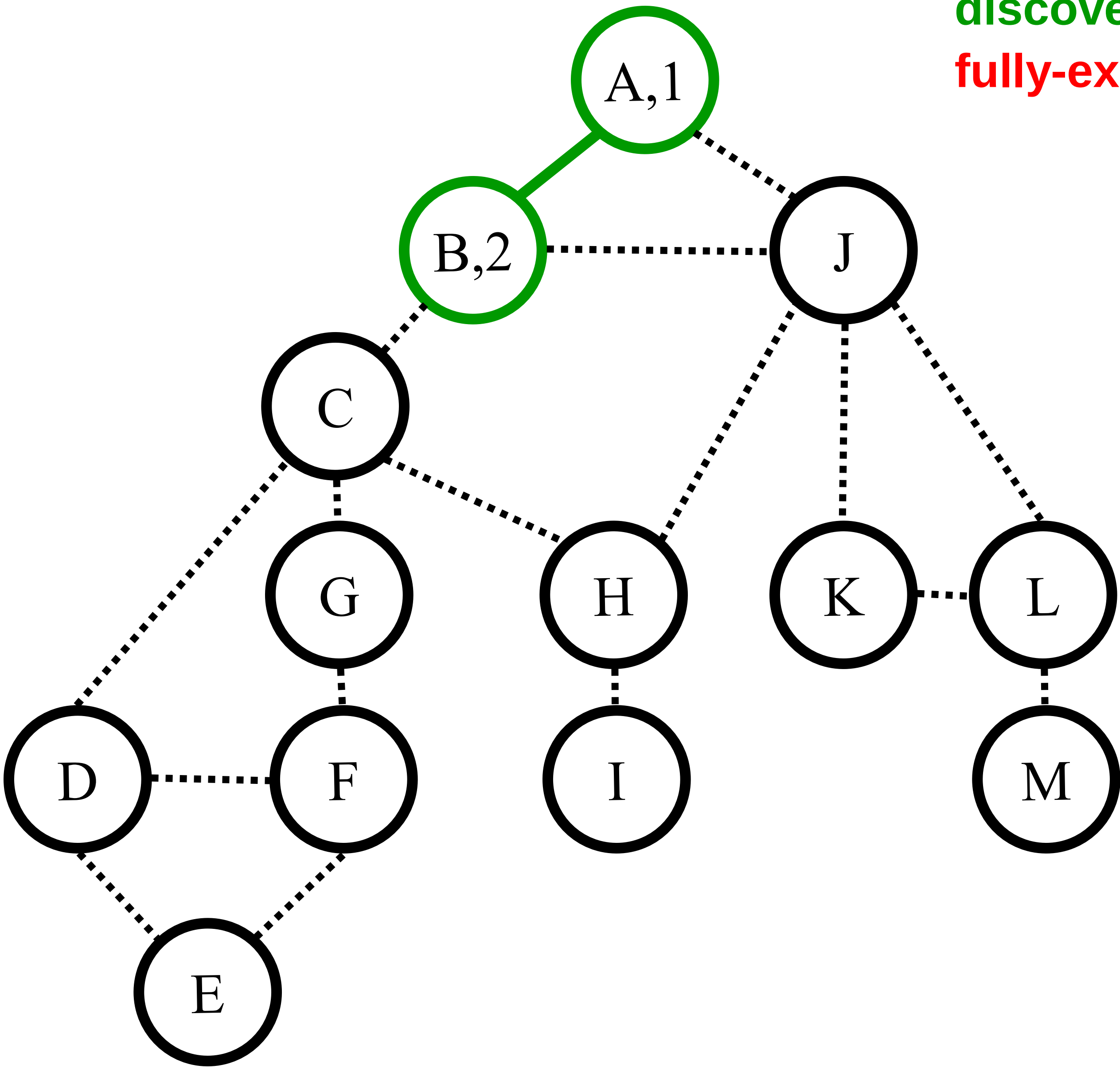
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

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

st[] =
{1,2}

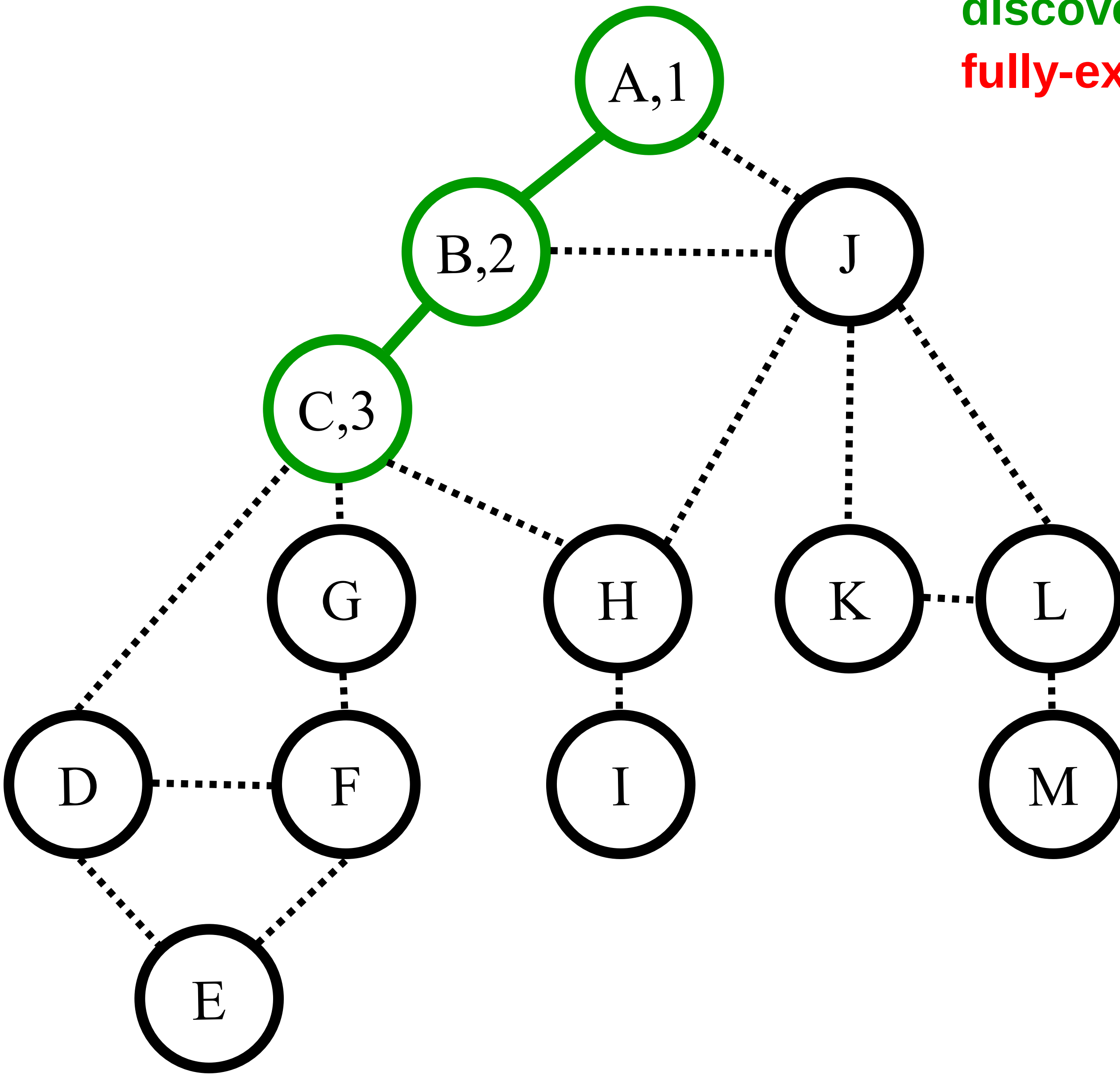
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

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

st[] =
{1,2,3}

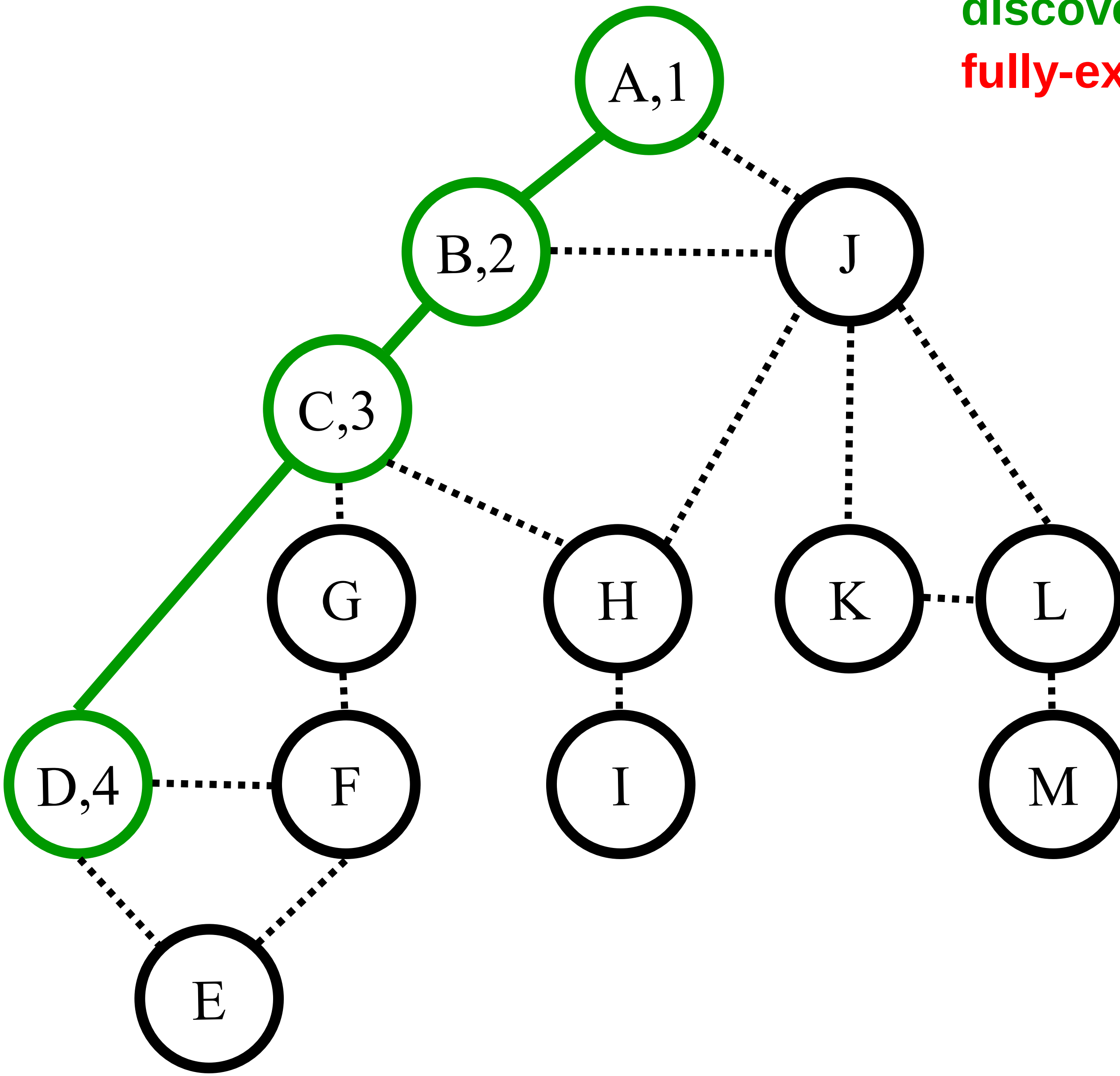
DFS(A)

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[] =
{1,2,3,4}

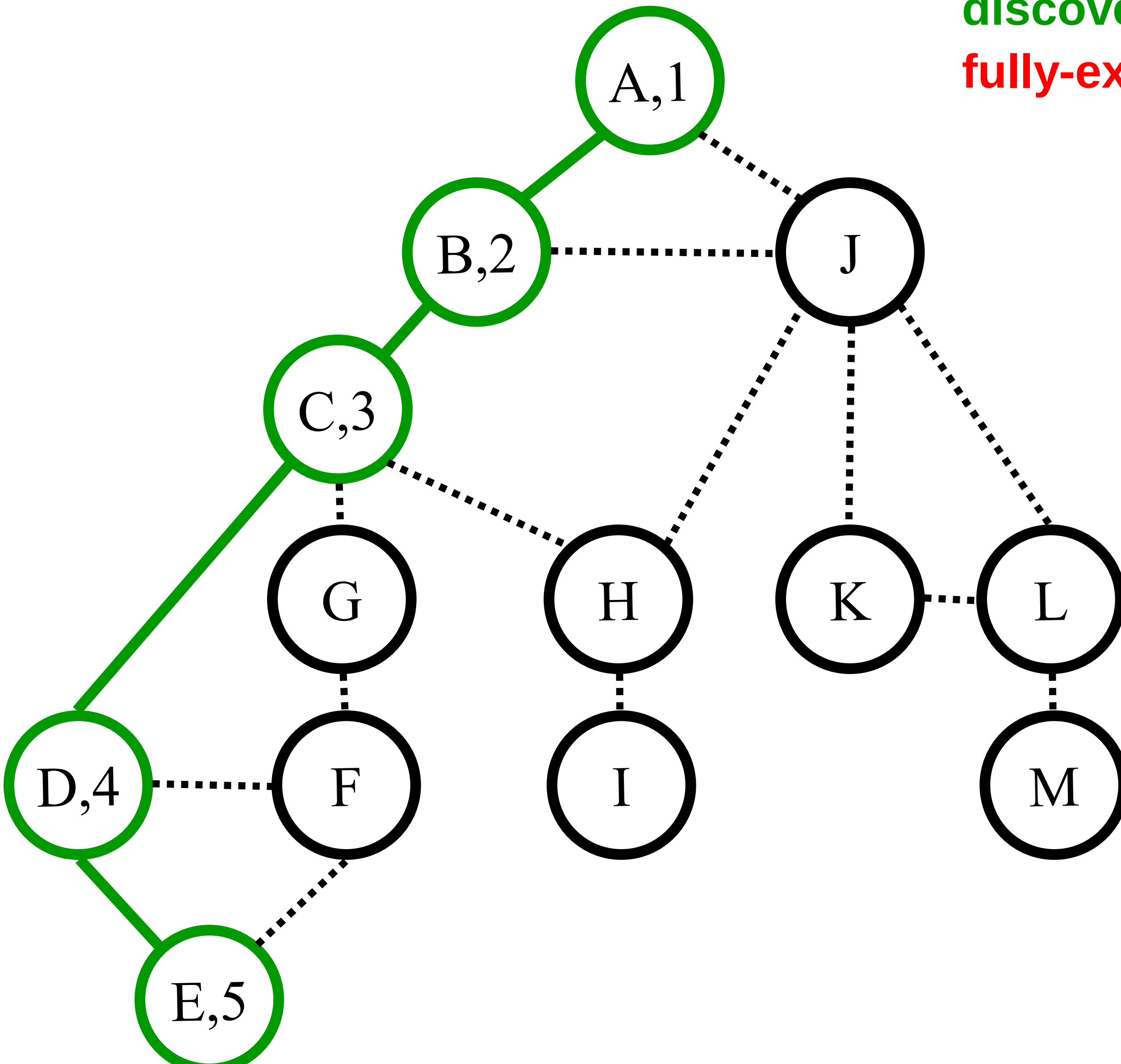
DFS(A)

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[] =
{1,2,3,4,5}

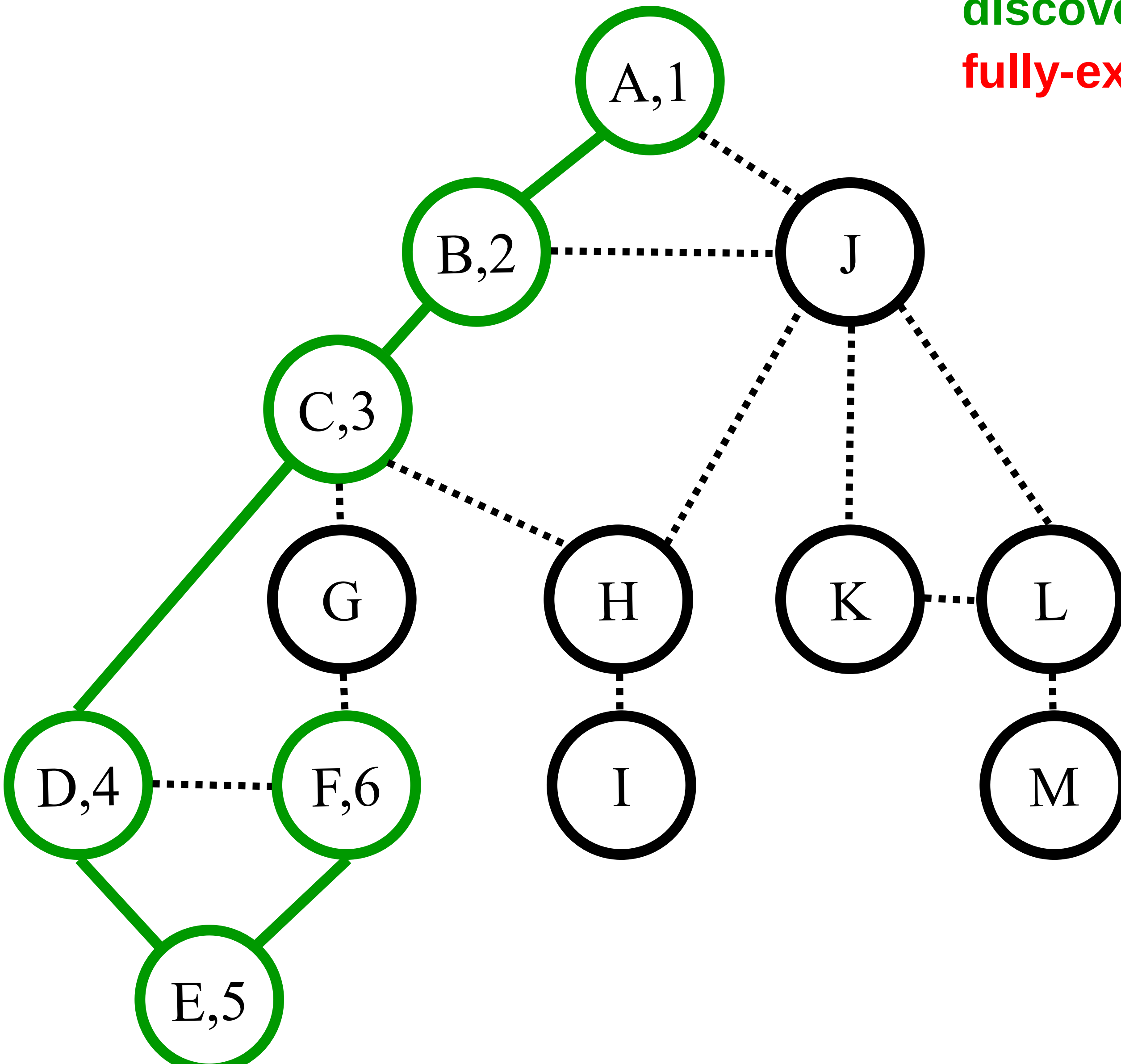
DFS(A)

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[] = {1,2,3,4,5, 6}

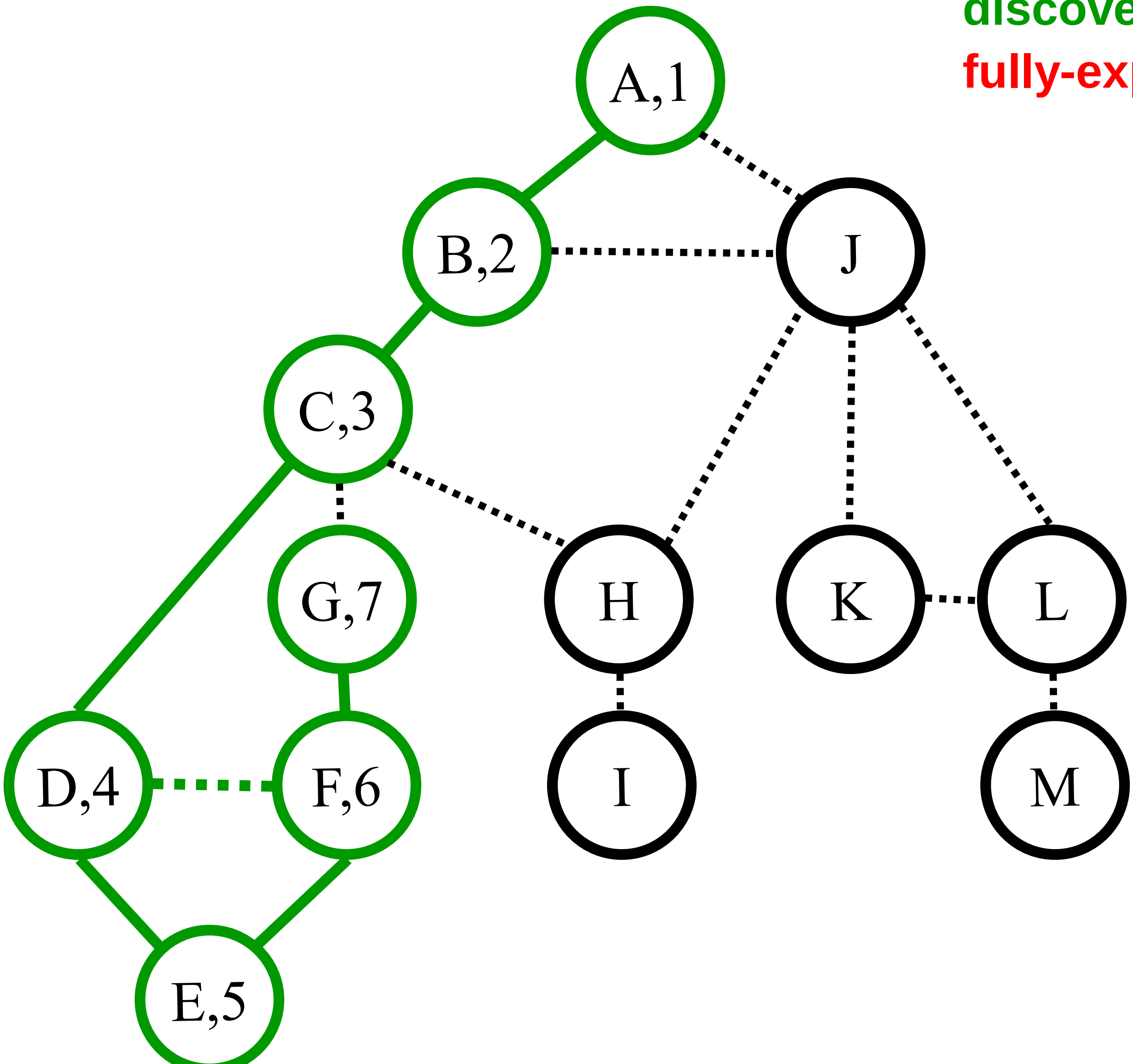
DFS(A)

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[] =
{1,2,3,4,5,
6,7}

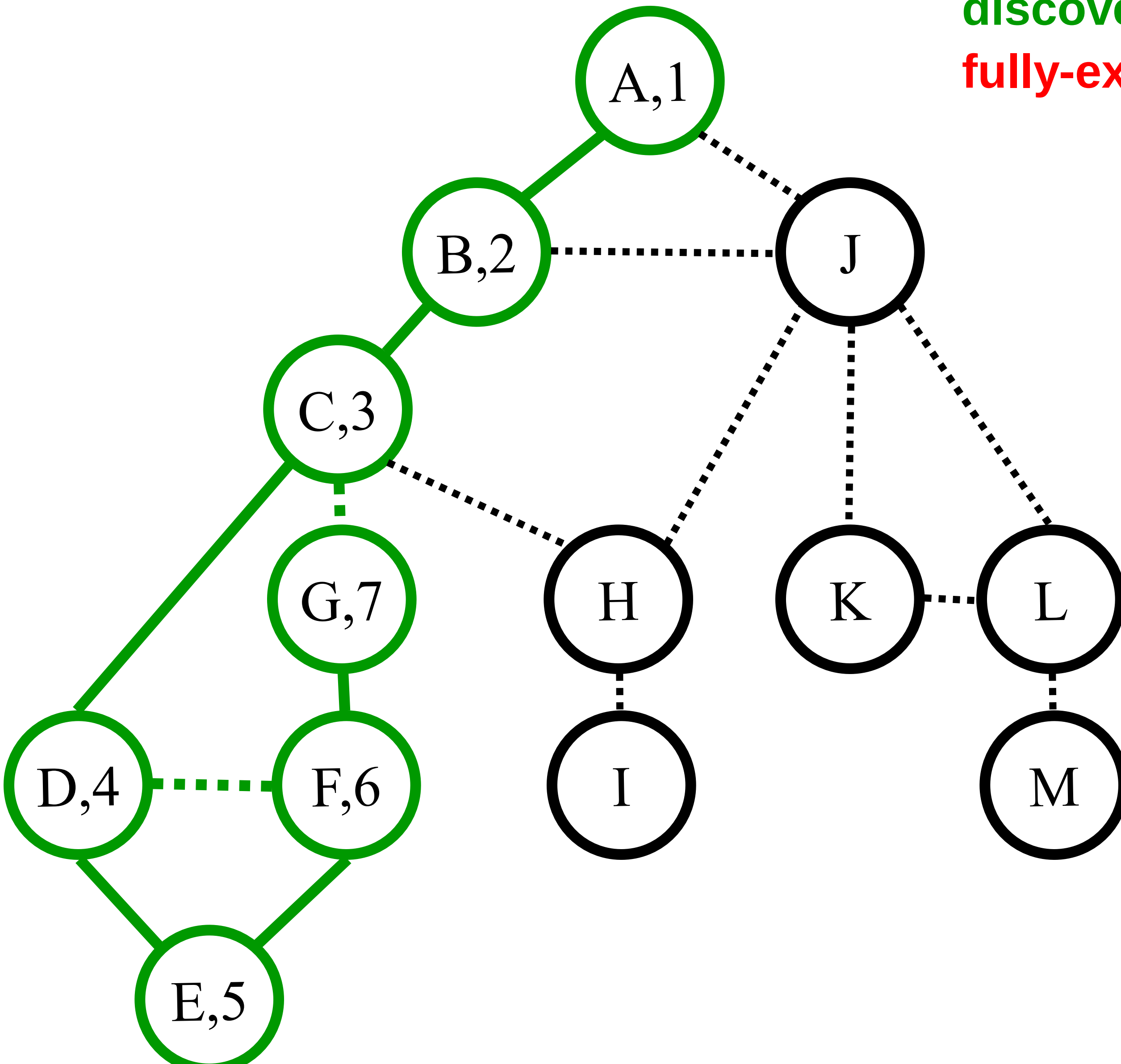
DFS(A)

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[] =
{1,2,3,4,5,
6,7}

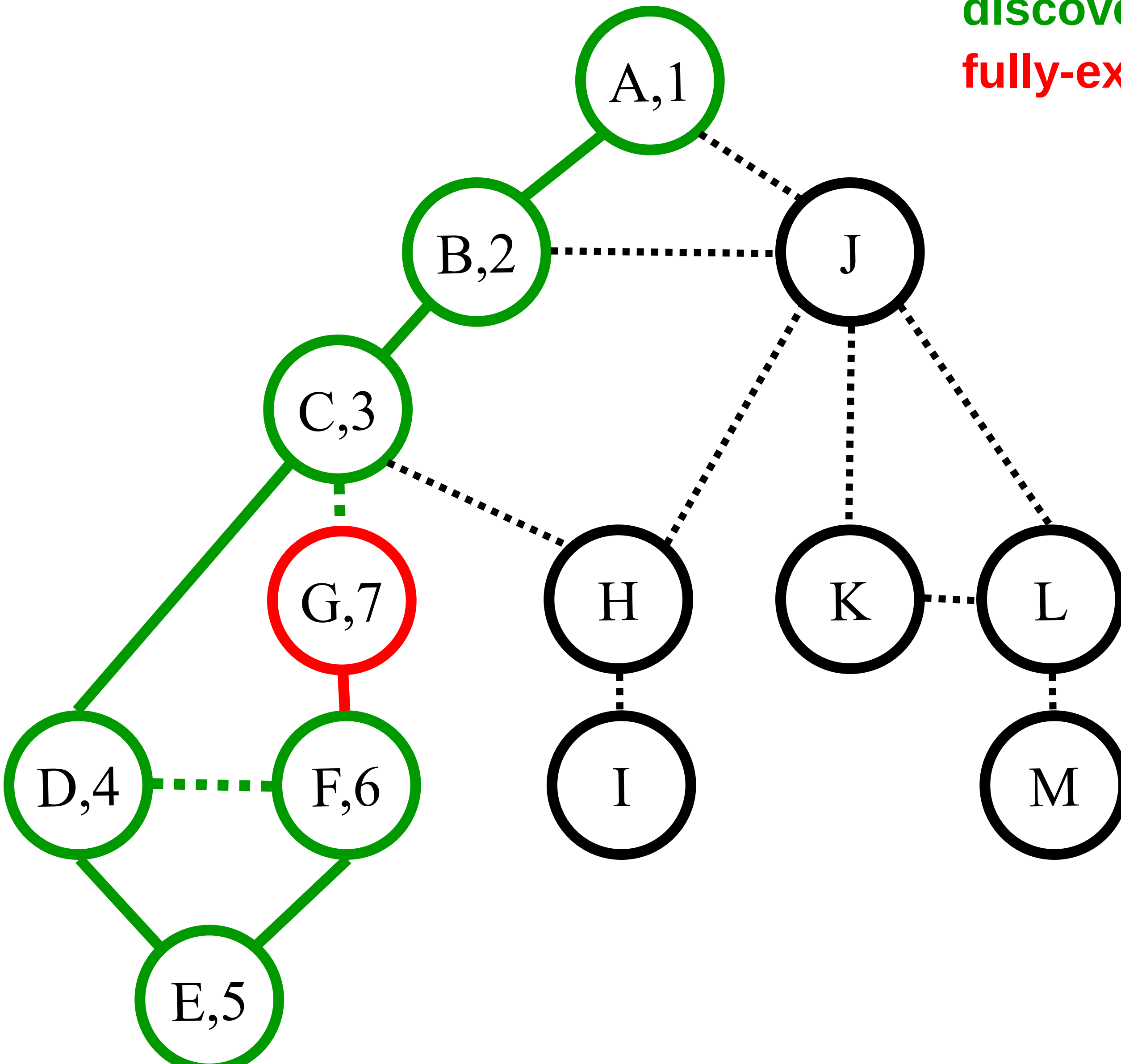
DFS(A)

Color code:

undiscovered

discovered

fully-explored

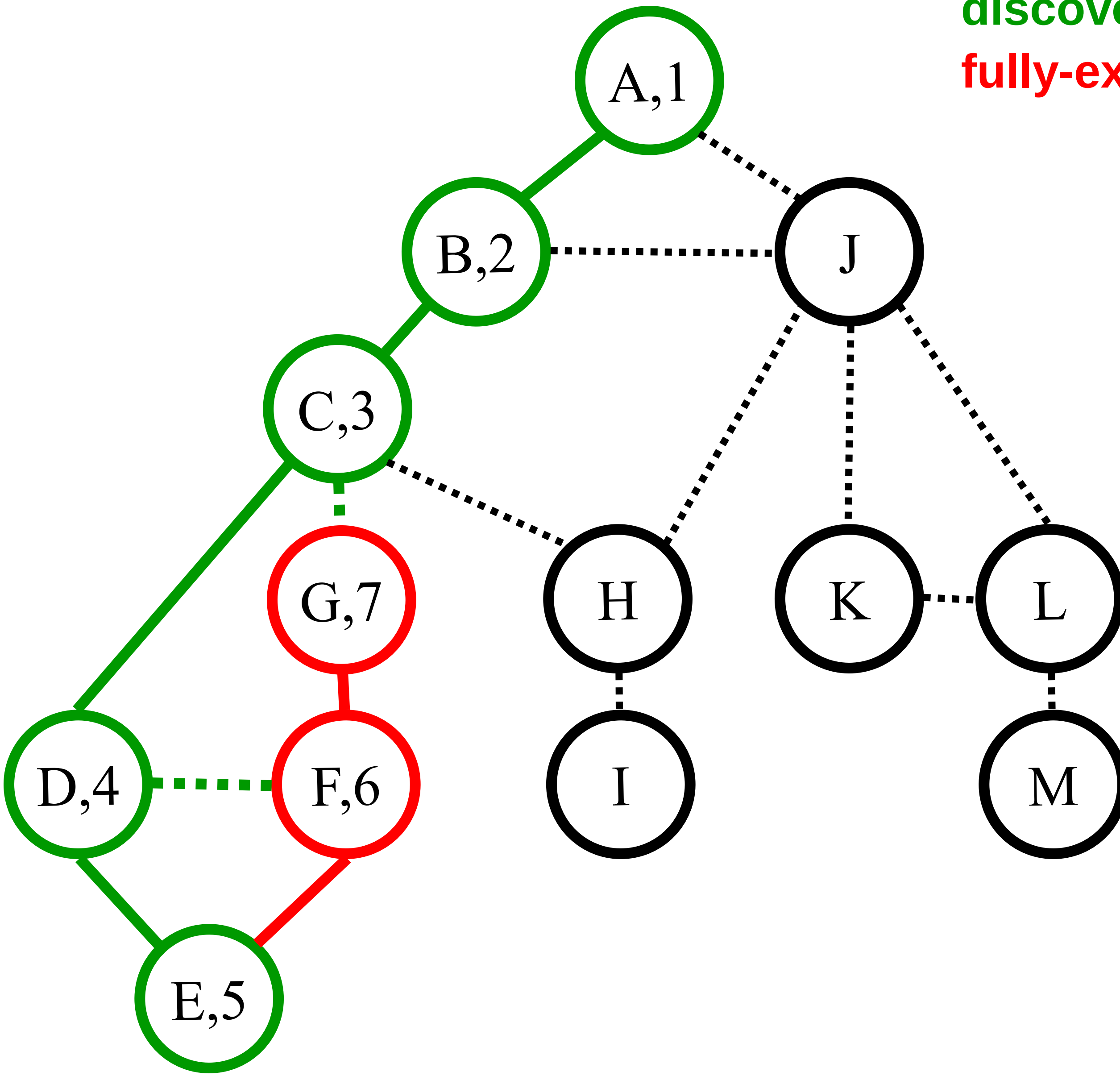


Call Stack:
(Edge list)

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

st[] =
{1,2,3,4,5,
6}

DFS(A)



Color code:

undiscovered

discovered

fully-explored

Call Stack:
(Edge list)

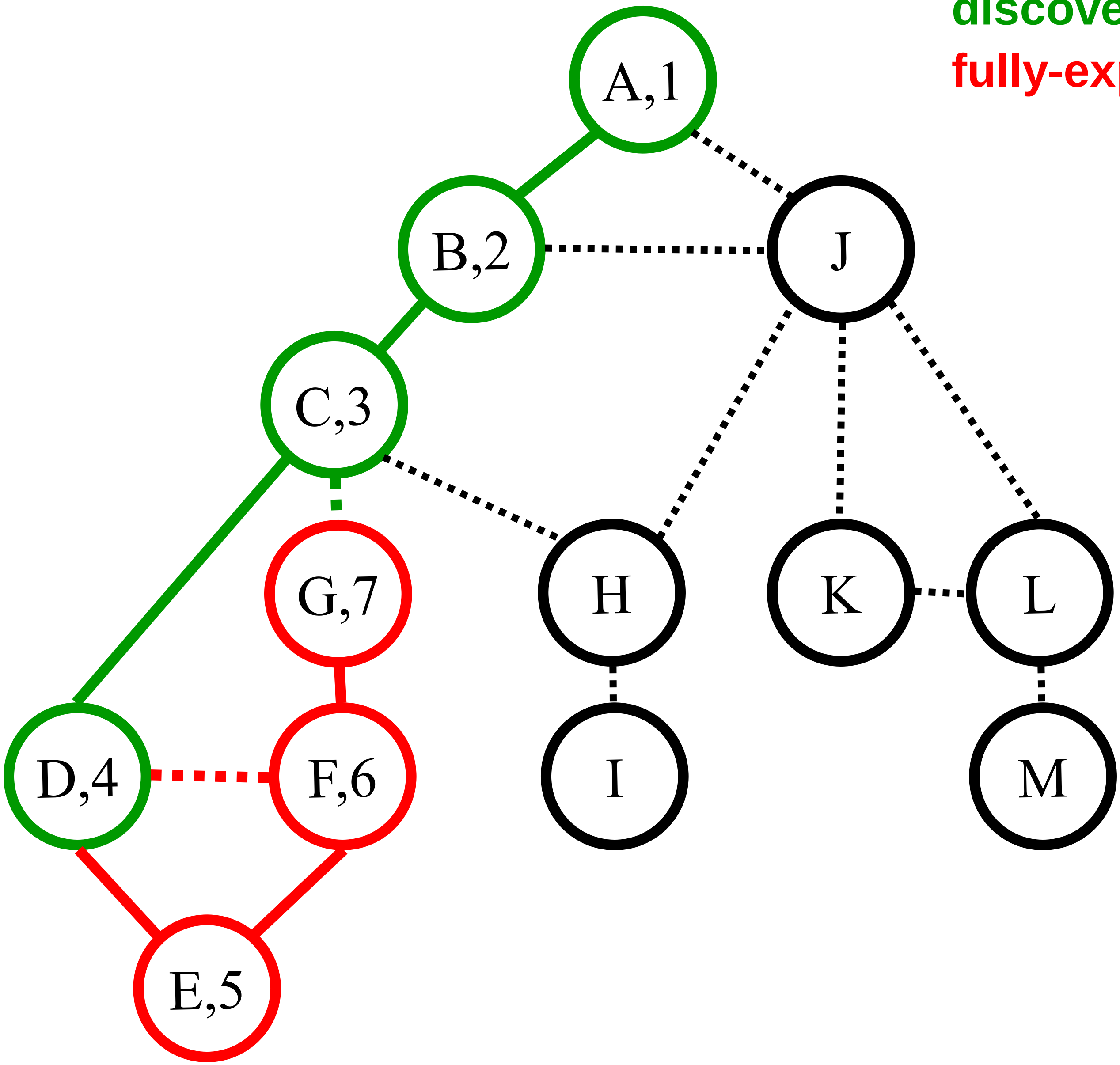
A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,G,H)
D (~~C~~,~~E~~,F)
E (~~D~~,~~F~~)

st[] =
{1,2,3,4,5}

DFS(A)

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[] = {1,2,3,4}

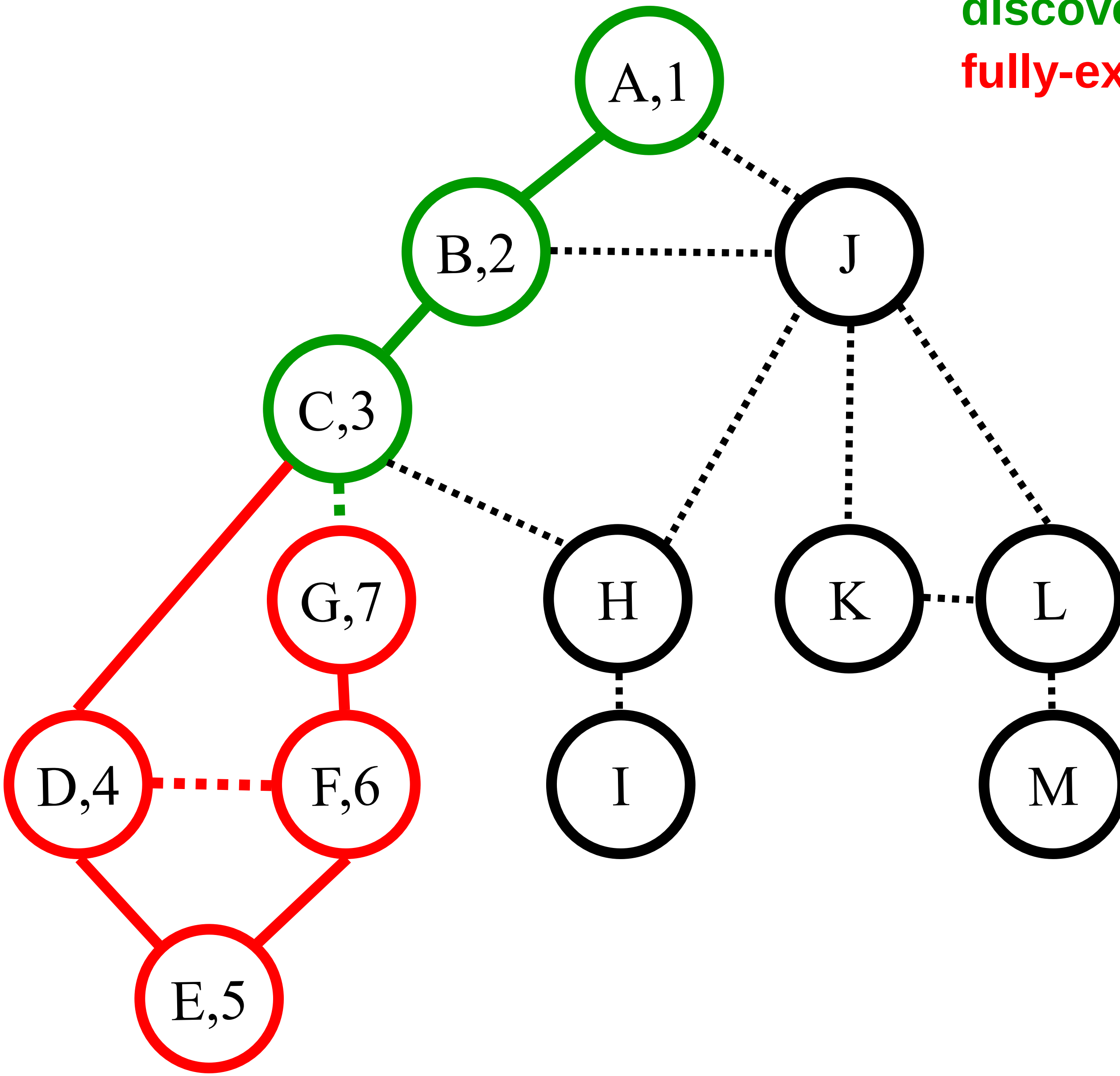
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

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

st[] =
{1,2,3}

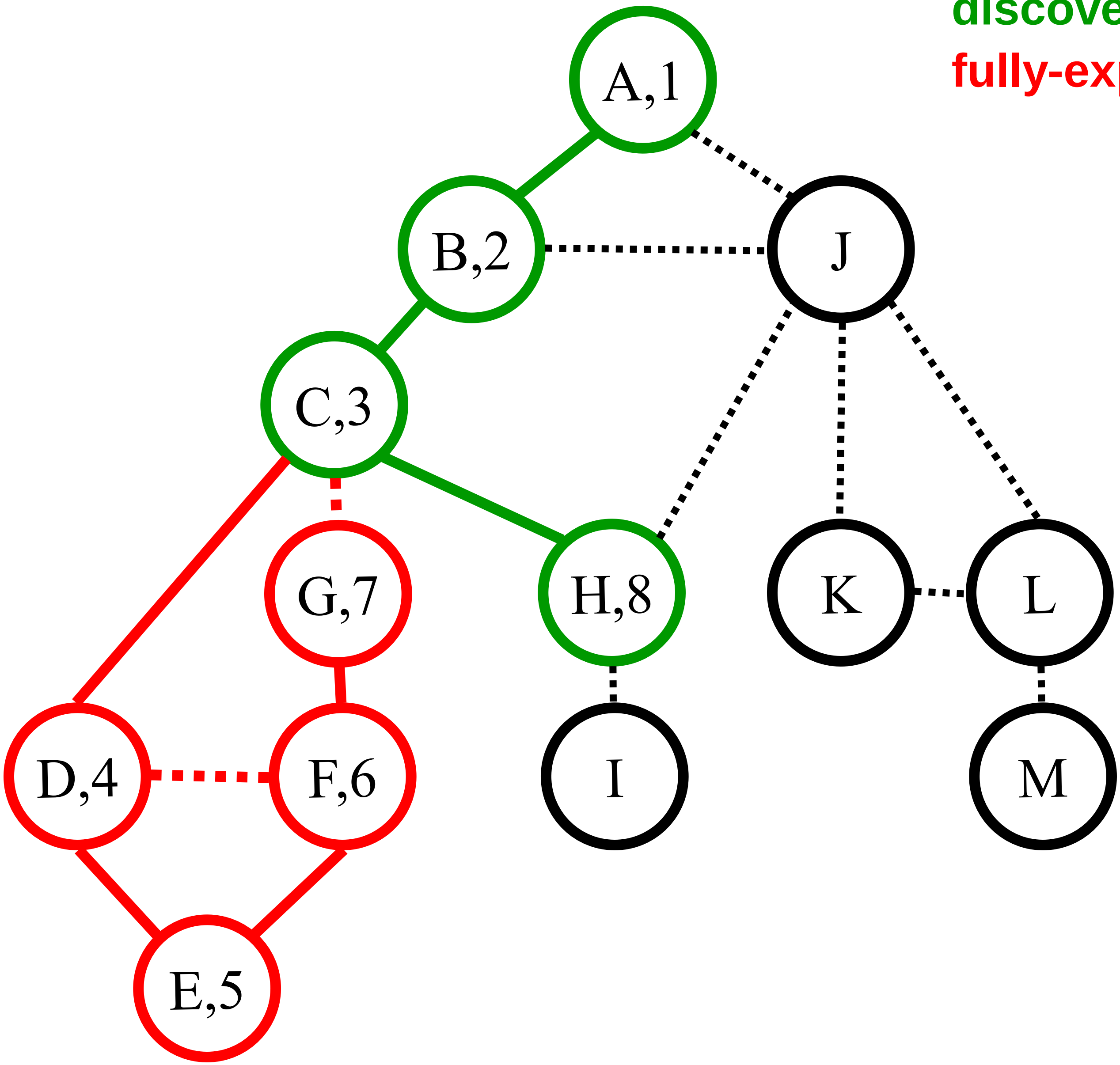
DFS(A)

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[] =
{1,2,3,8}

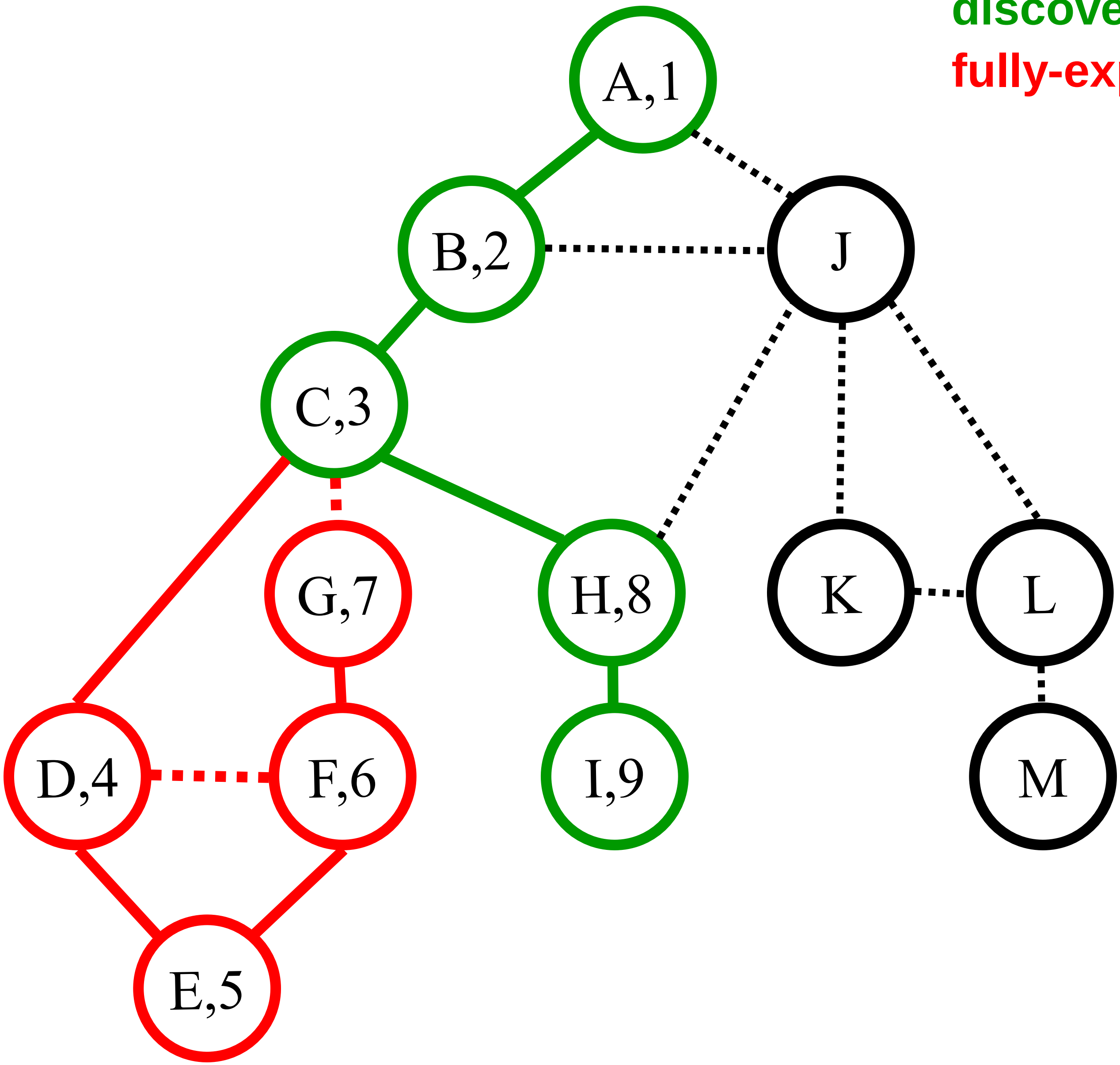
DFS(A)

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[] = {1,2,3,8,9}

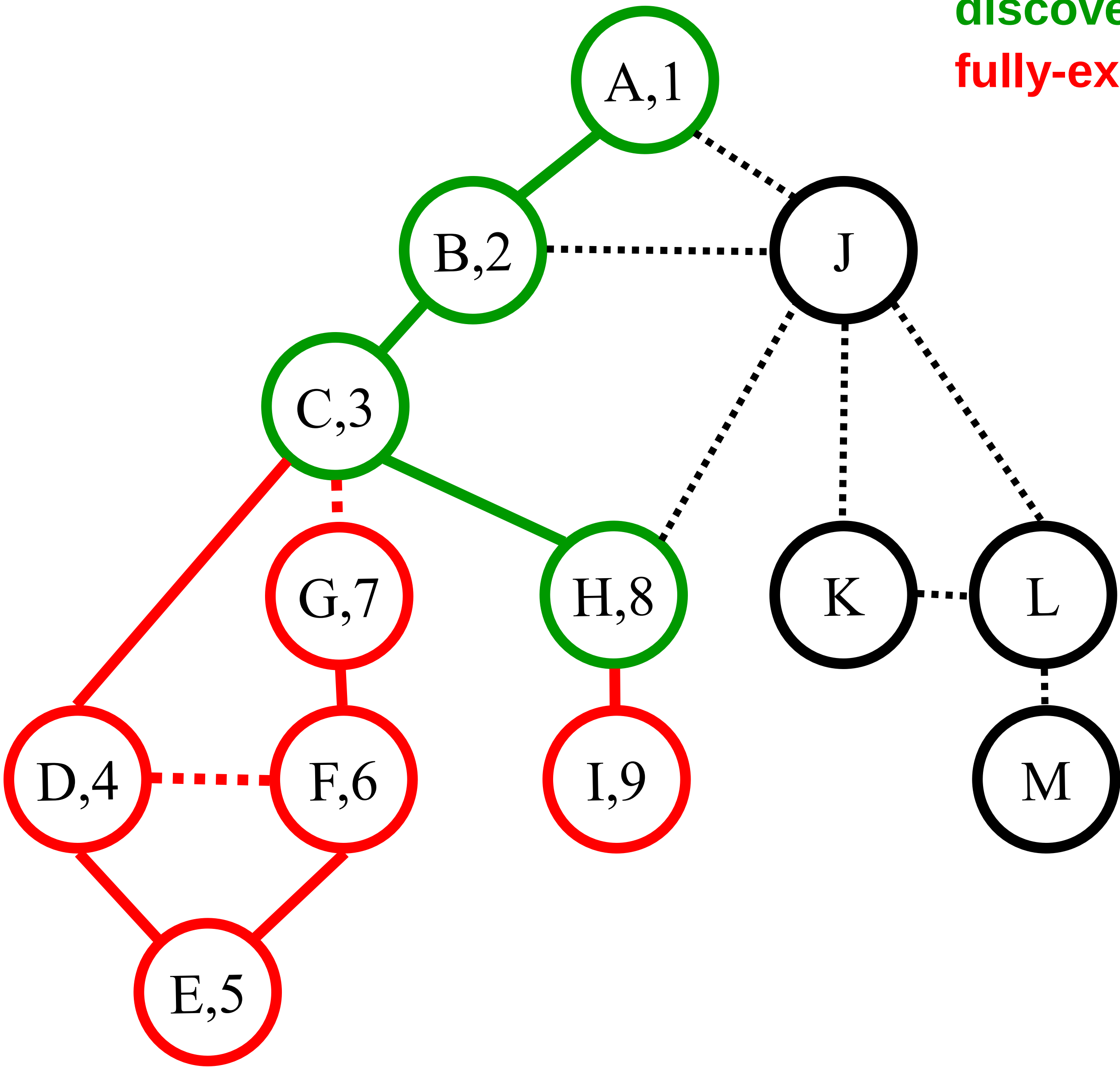
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

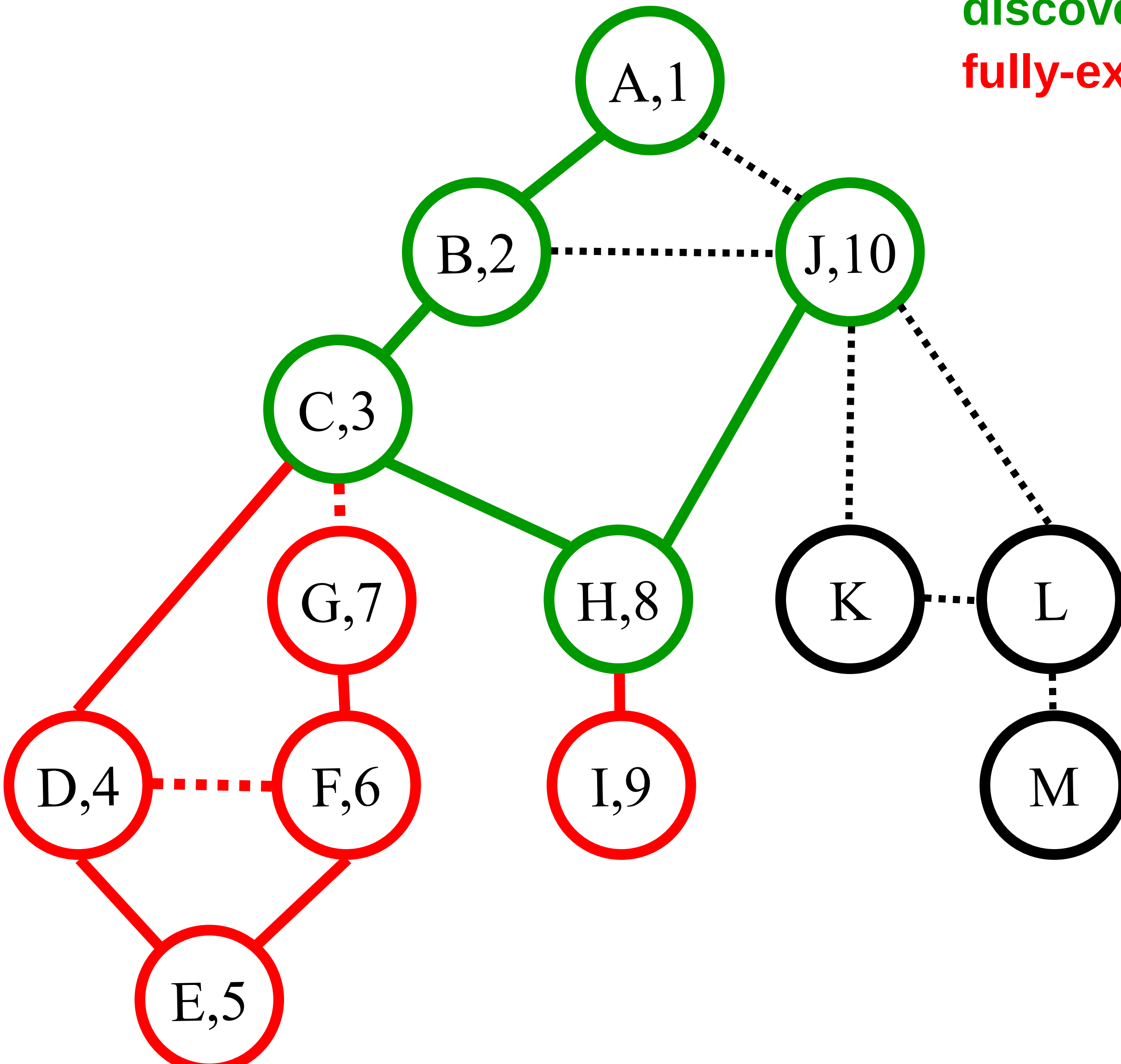
A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,J)

st[] =
{1,2,3,8}

DFS(A)

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[] =
{1,2,3,8,
10}

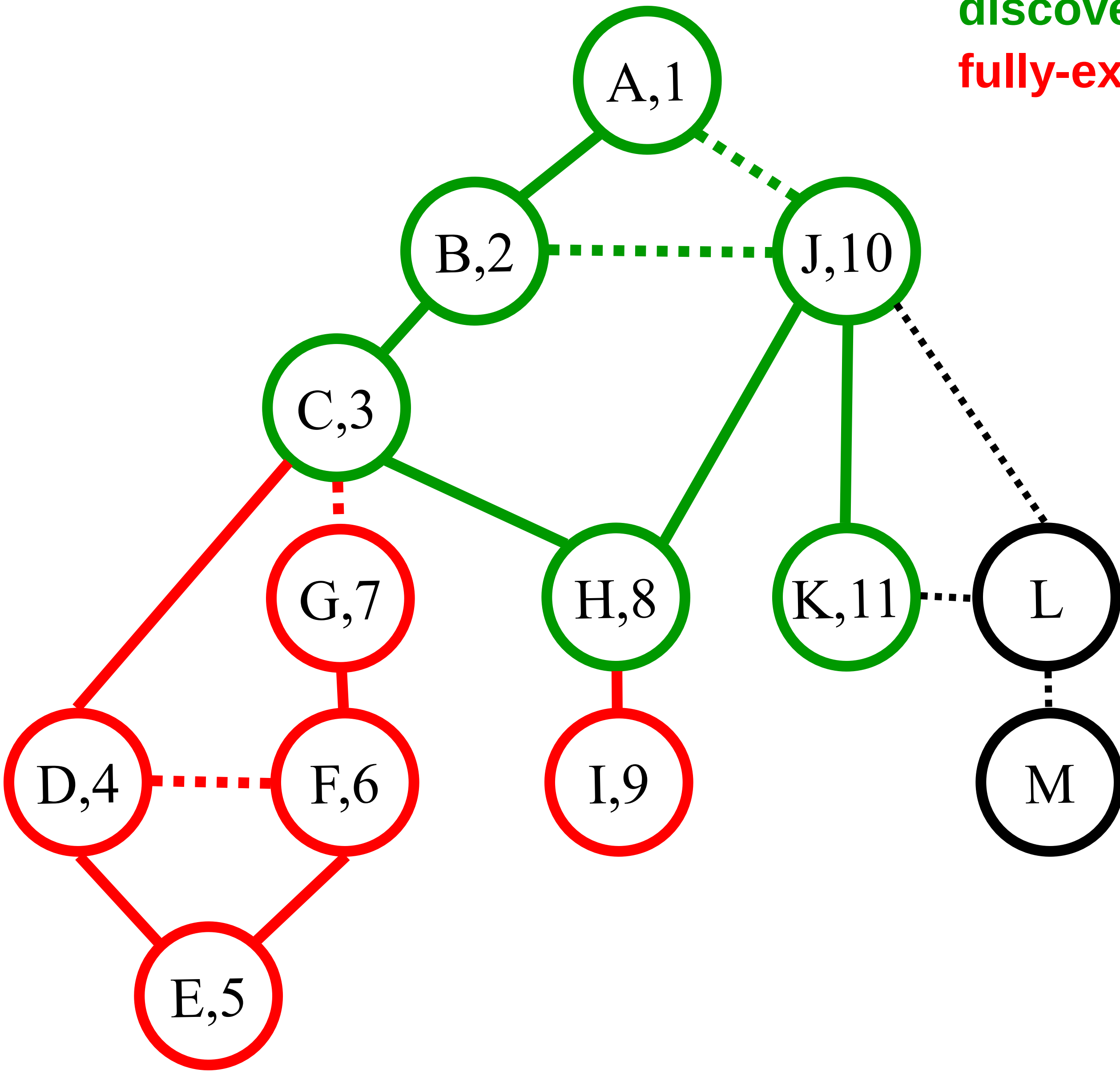
DFS(A)

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[] =
{1,2,3,8,10
,11}

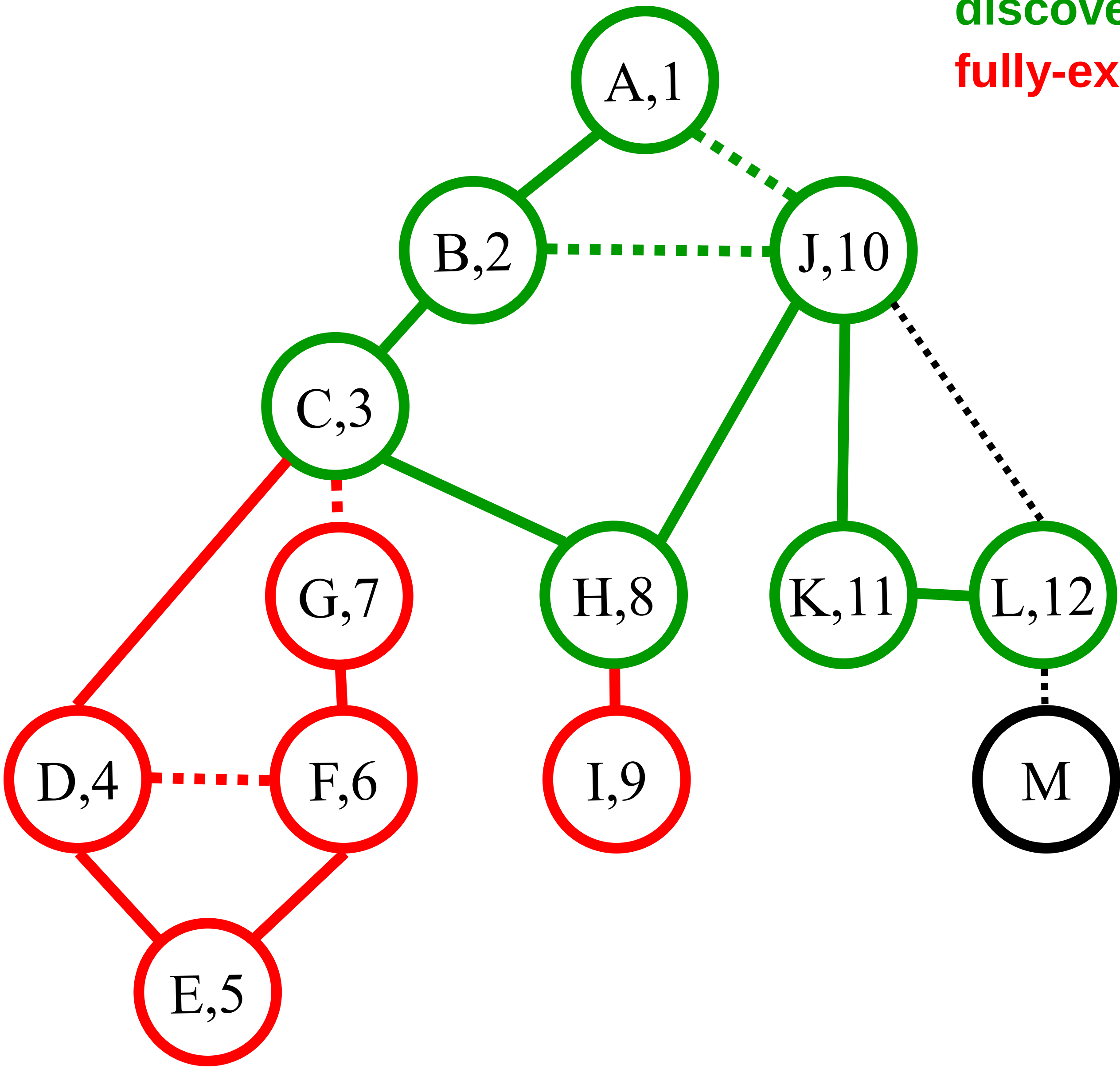
DFS(A)

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[] = {1,2,3,8,10,11,12}

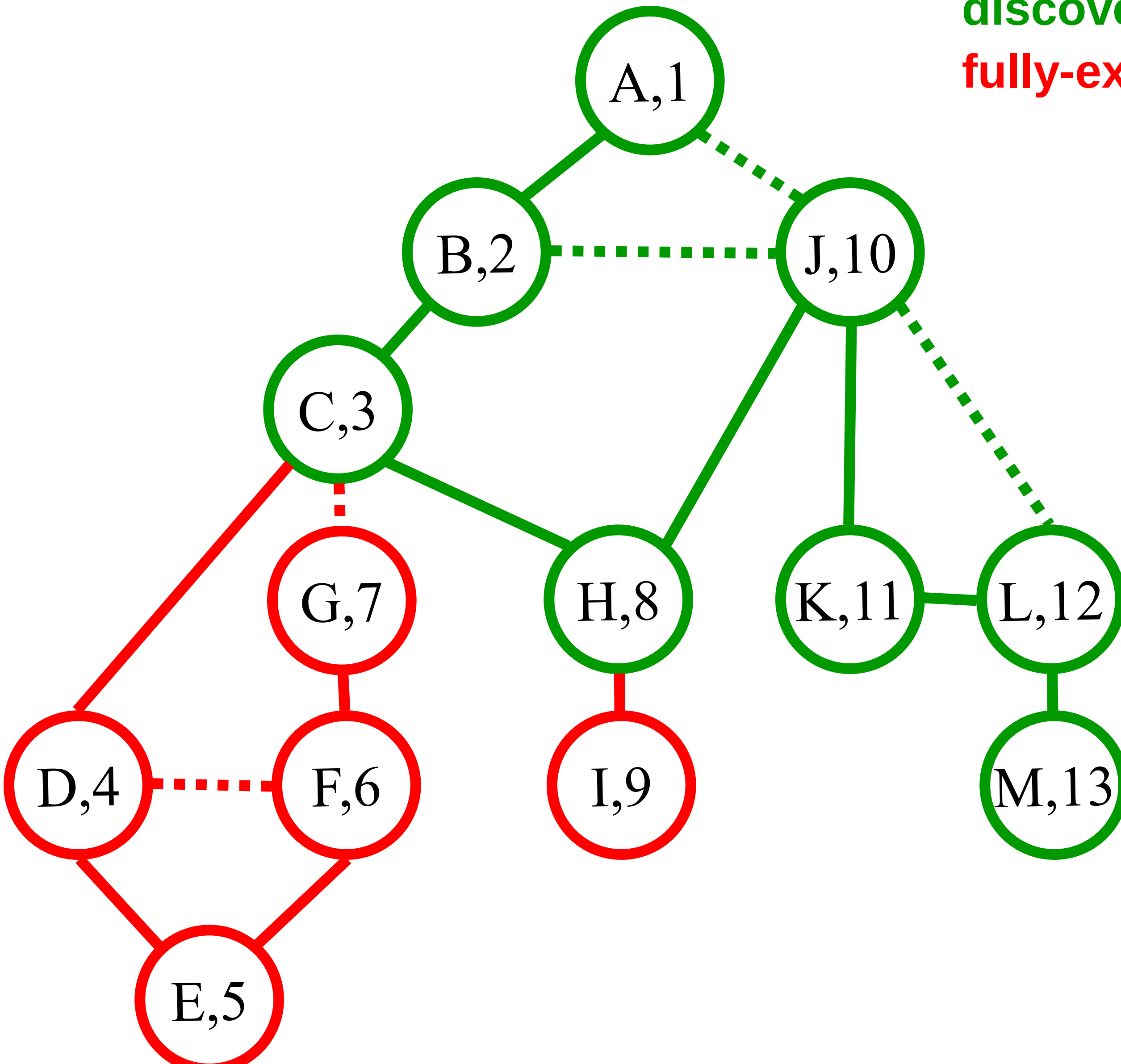
DFS(A)

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[] =
{1,2,3,8,10
,11,12,13}

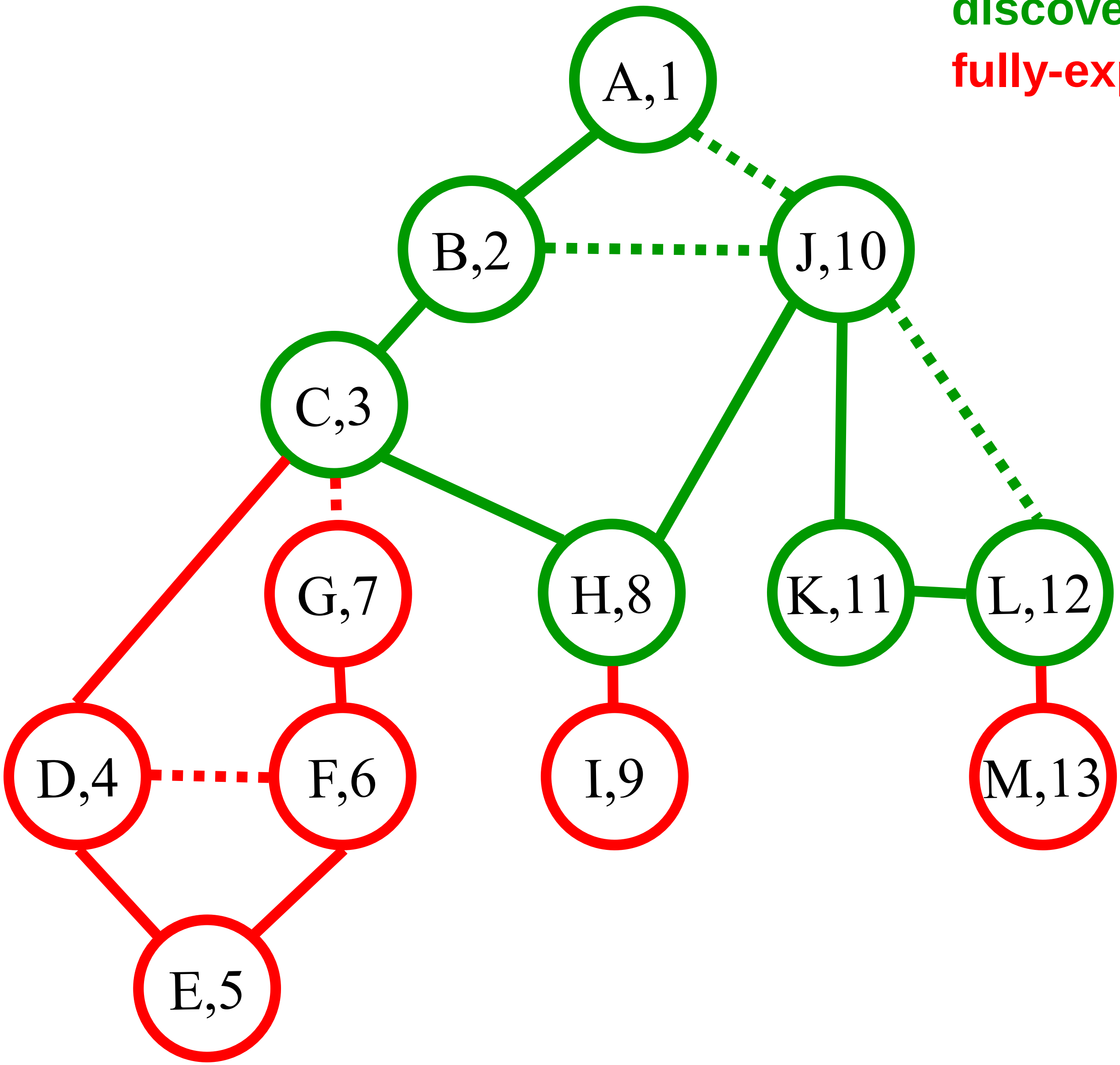
DFS(A)

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[] = {1,2,3,8,10,11,12}

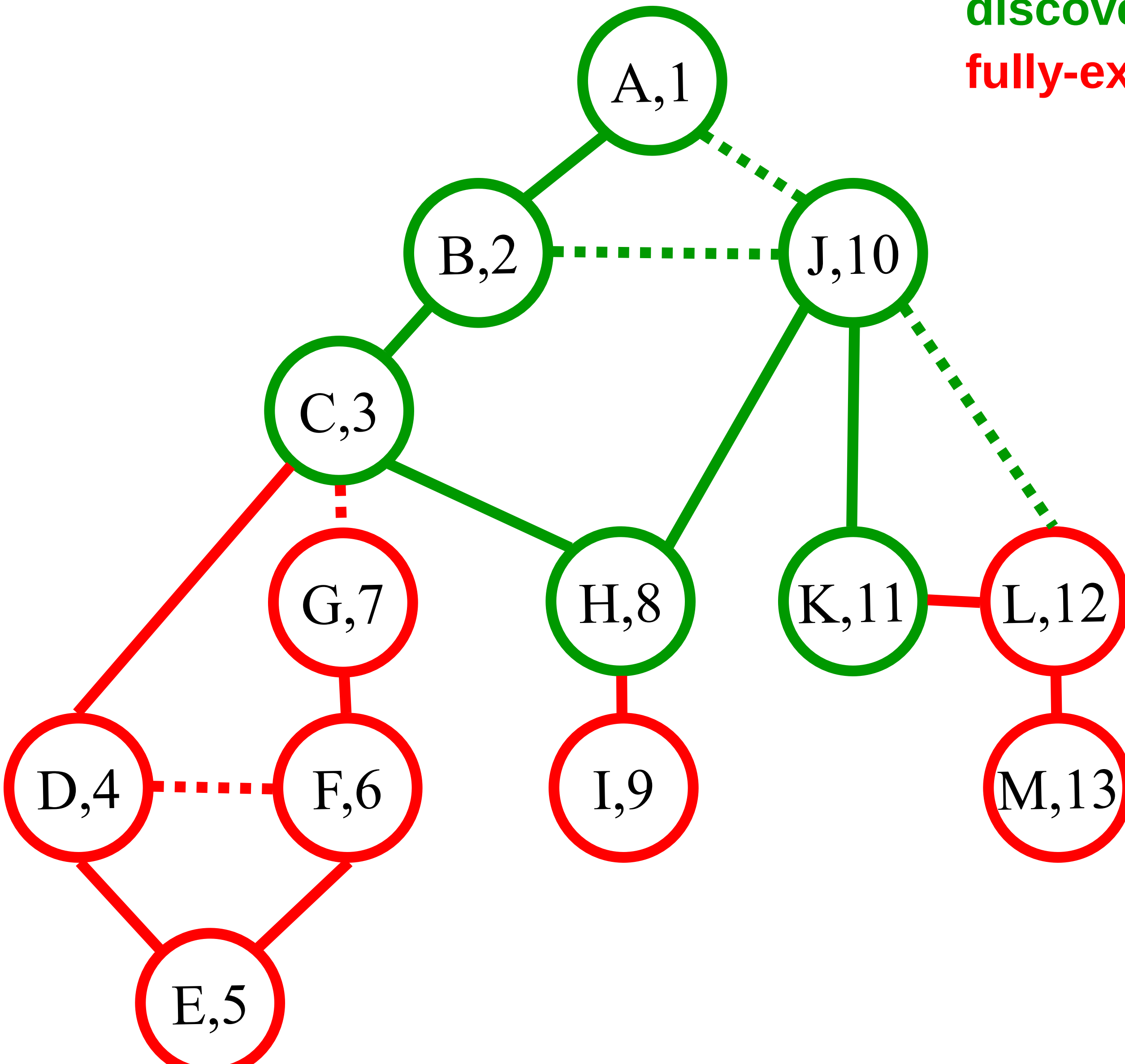
DFS(A)

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[] =
{1,2,3,8,10
,11}

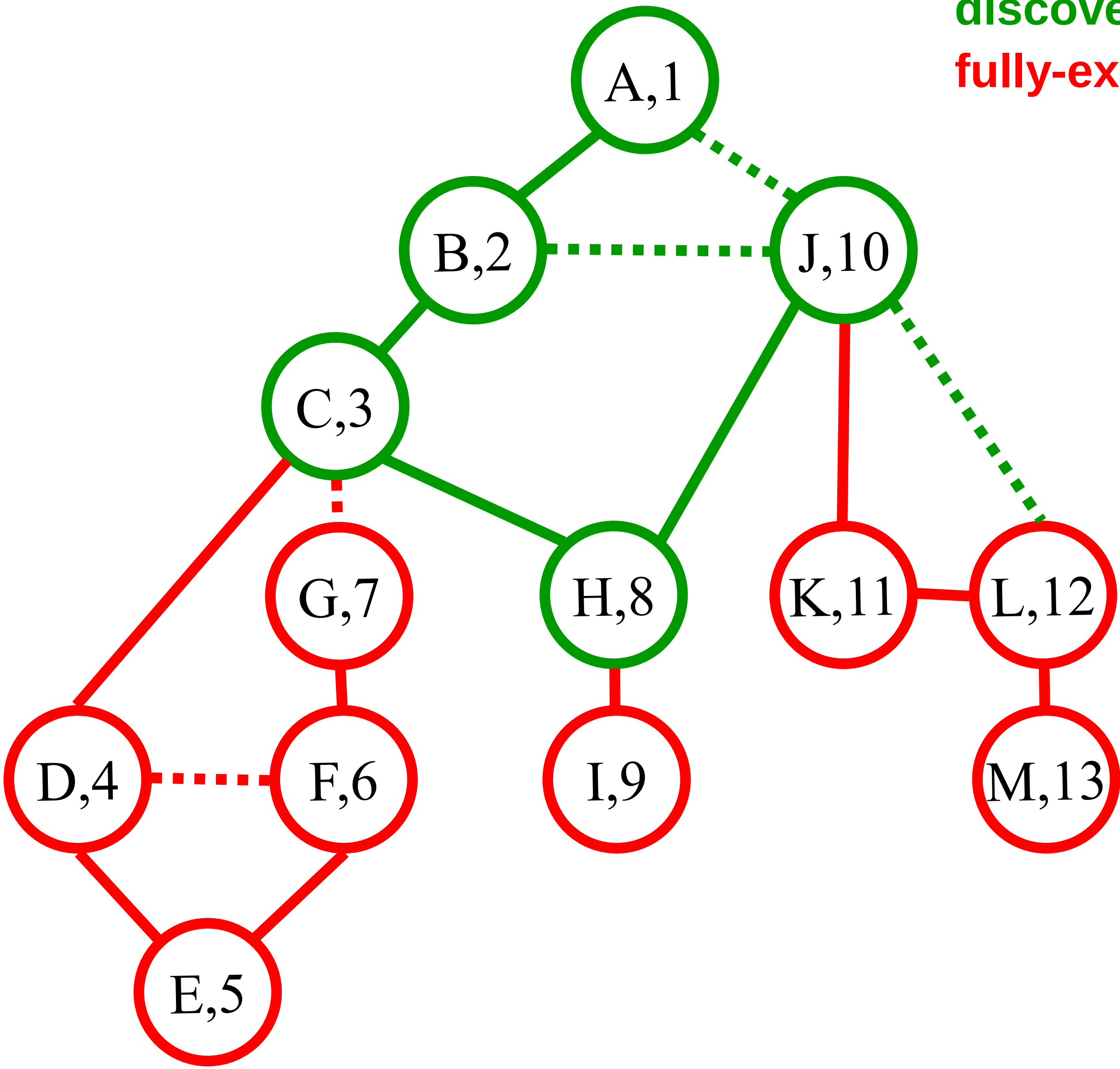
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

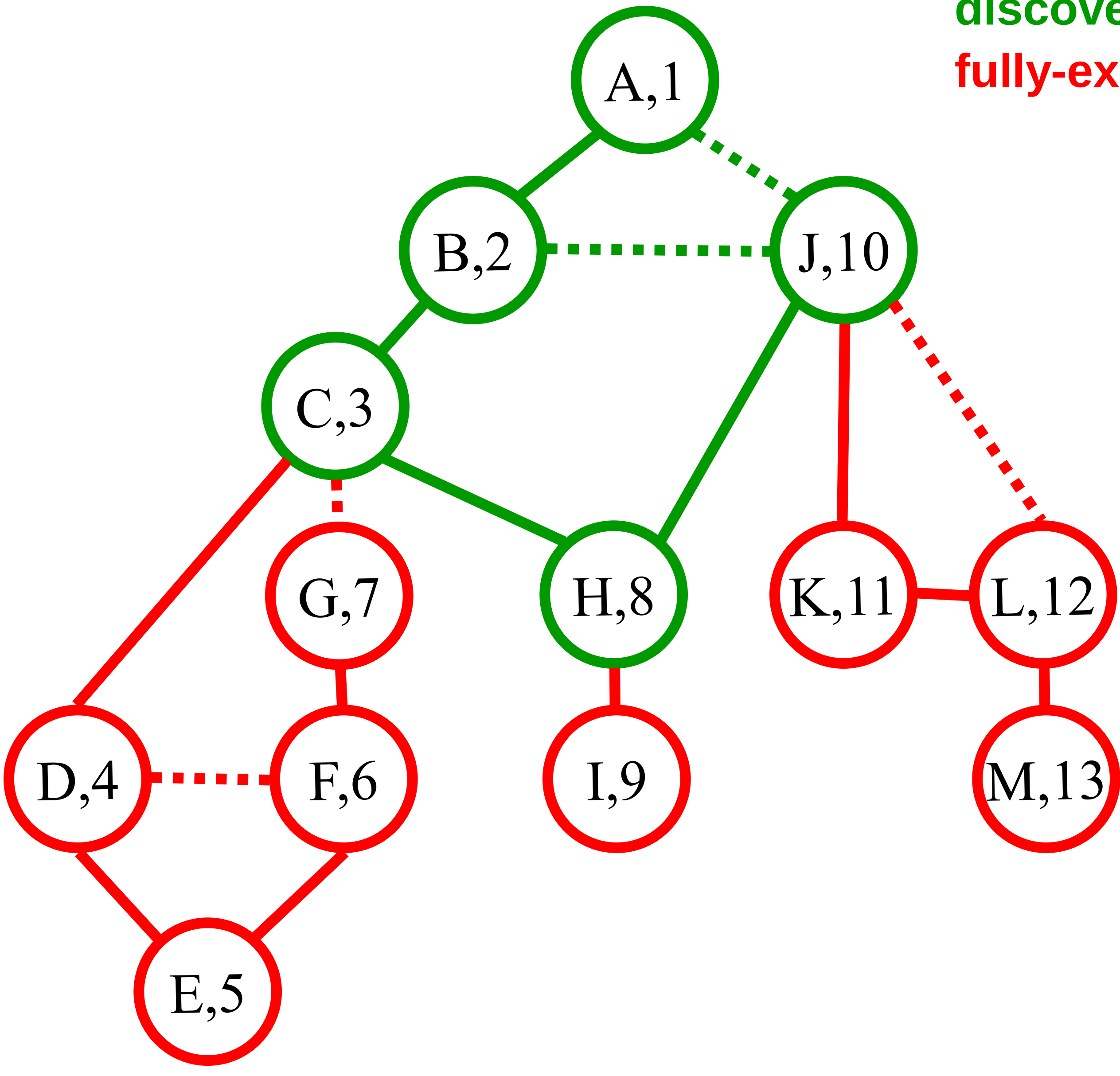
A (B,J)
B (A,C,J)
C (B,D,G,H)
H (C,I,J)
J (A,B,H,K,L)

st[] =
{1,2,3,8,
10}

DFS(A)

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[] =
{1,2,3,8,
10}

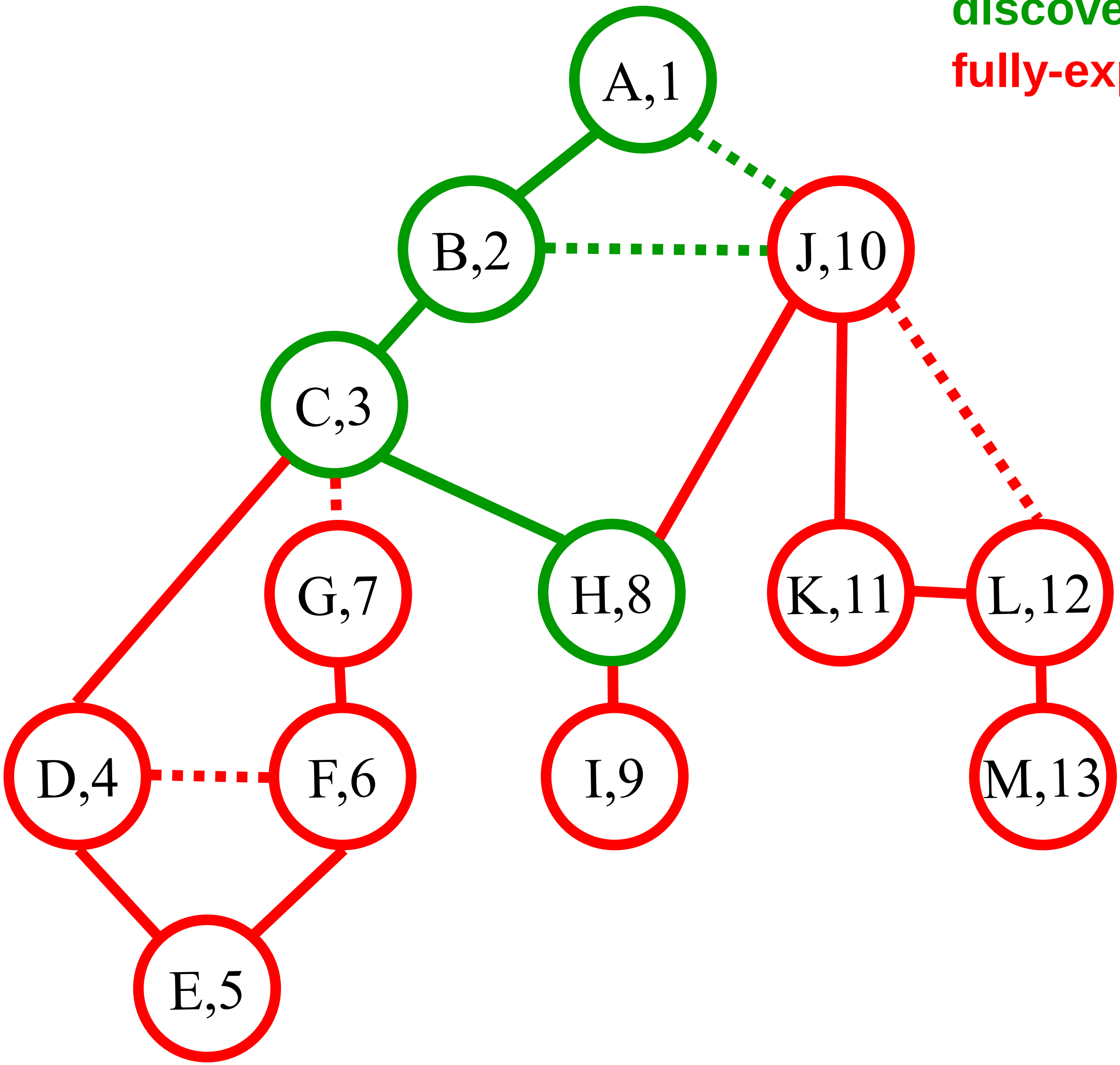
DFS(A)

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[] =
{1,2,3,8}

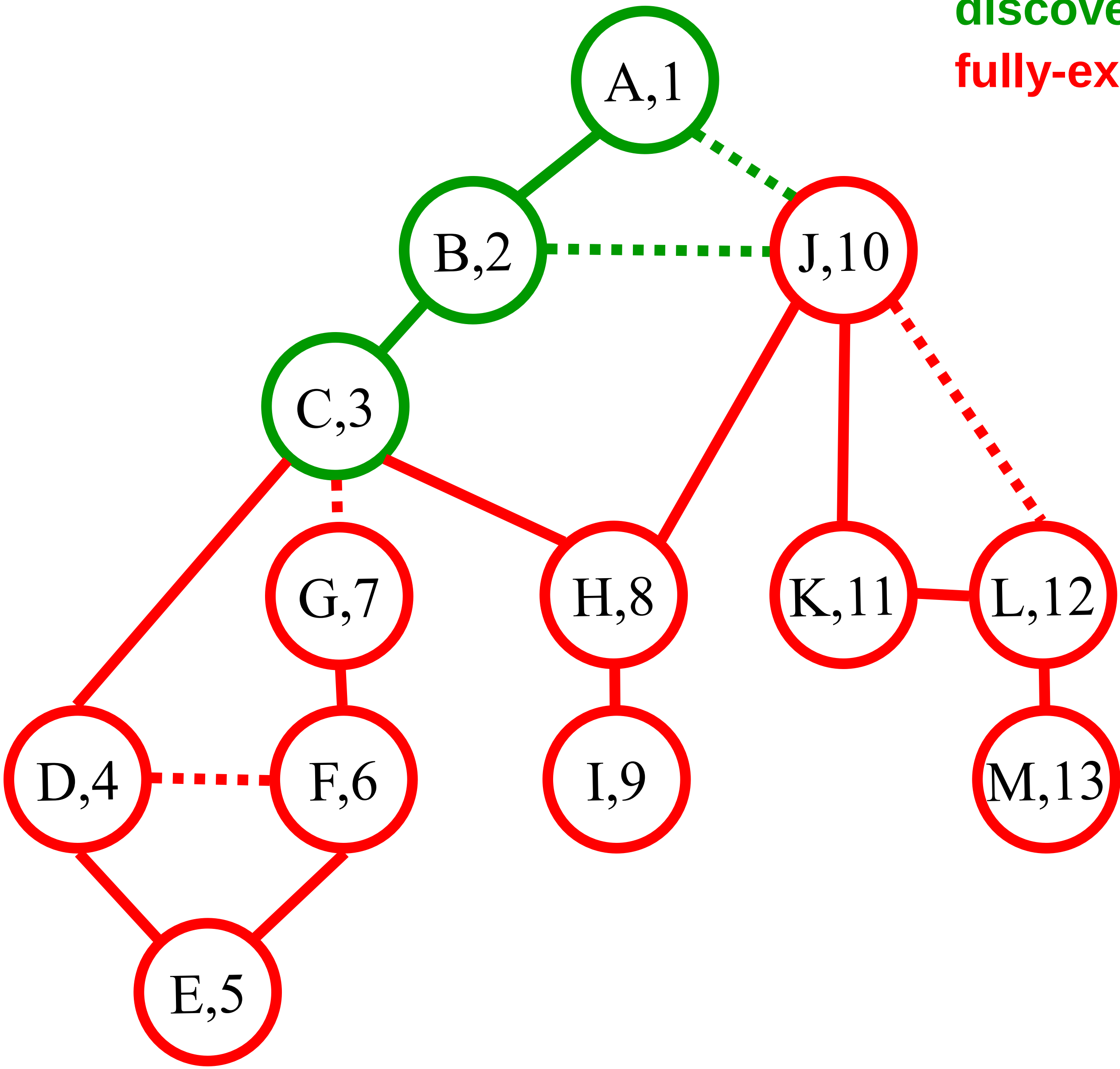
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

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

st[] =
{1,2,3}

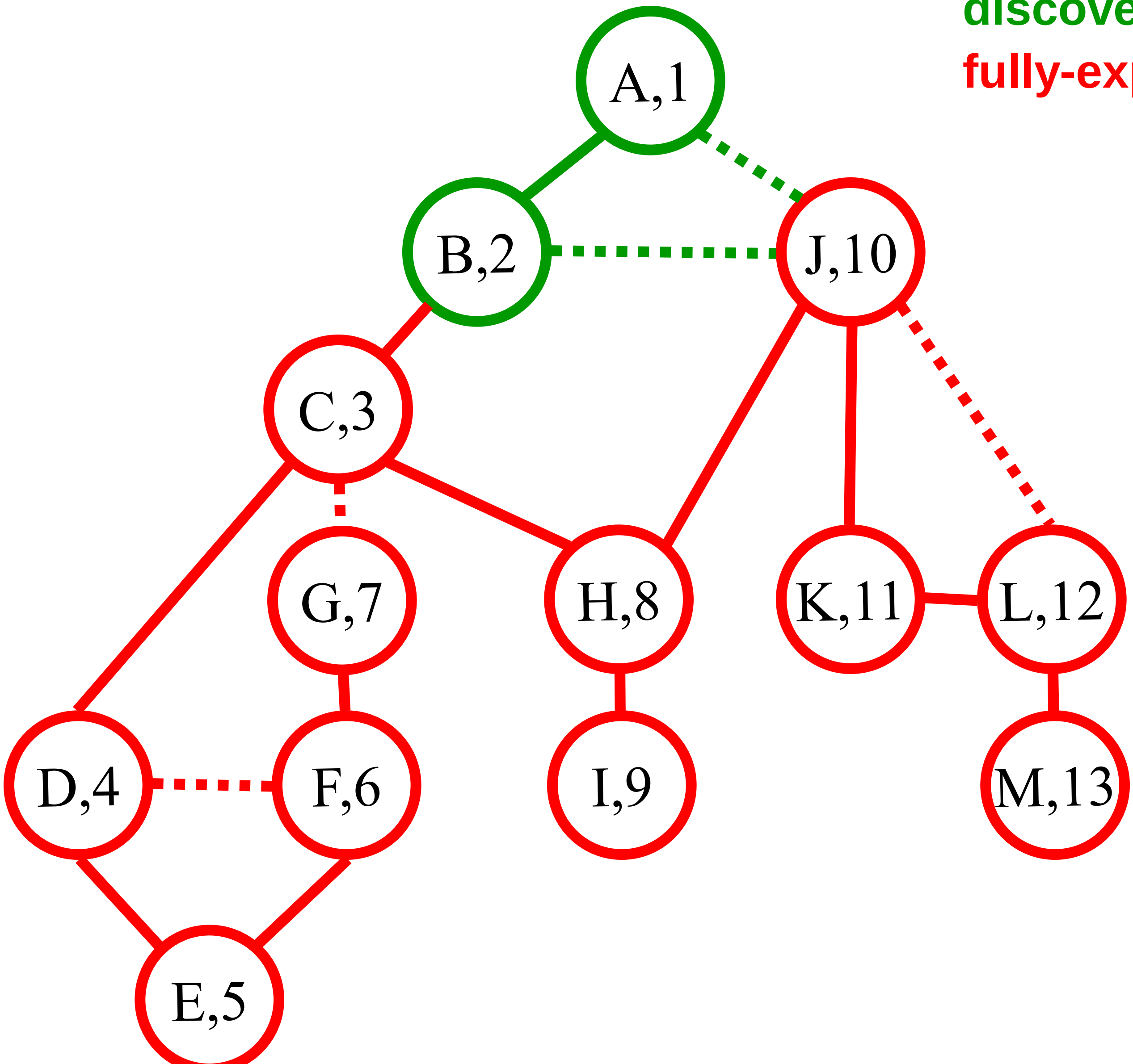
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

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

st[] =
{1,2}

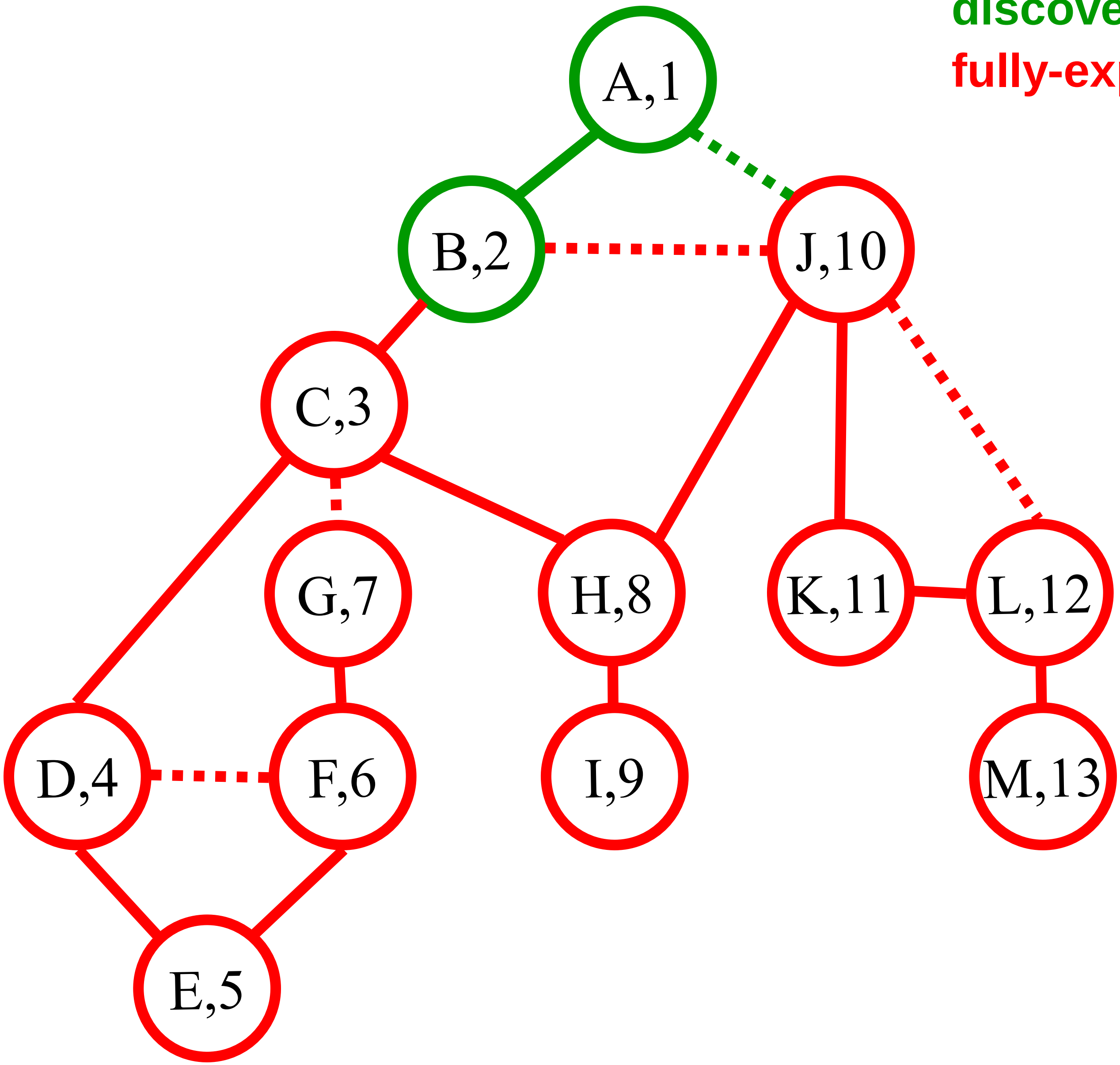
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

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

st[] =
{1,2}

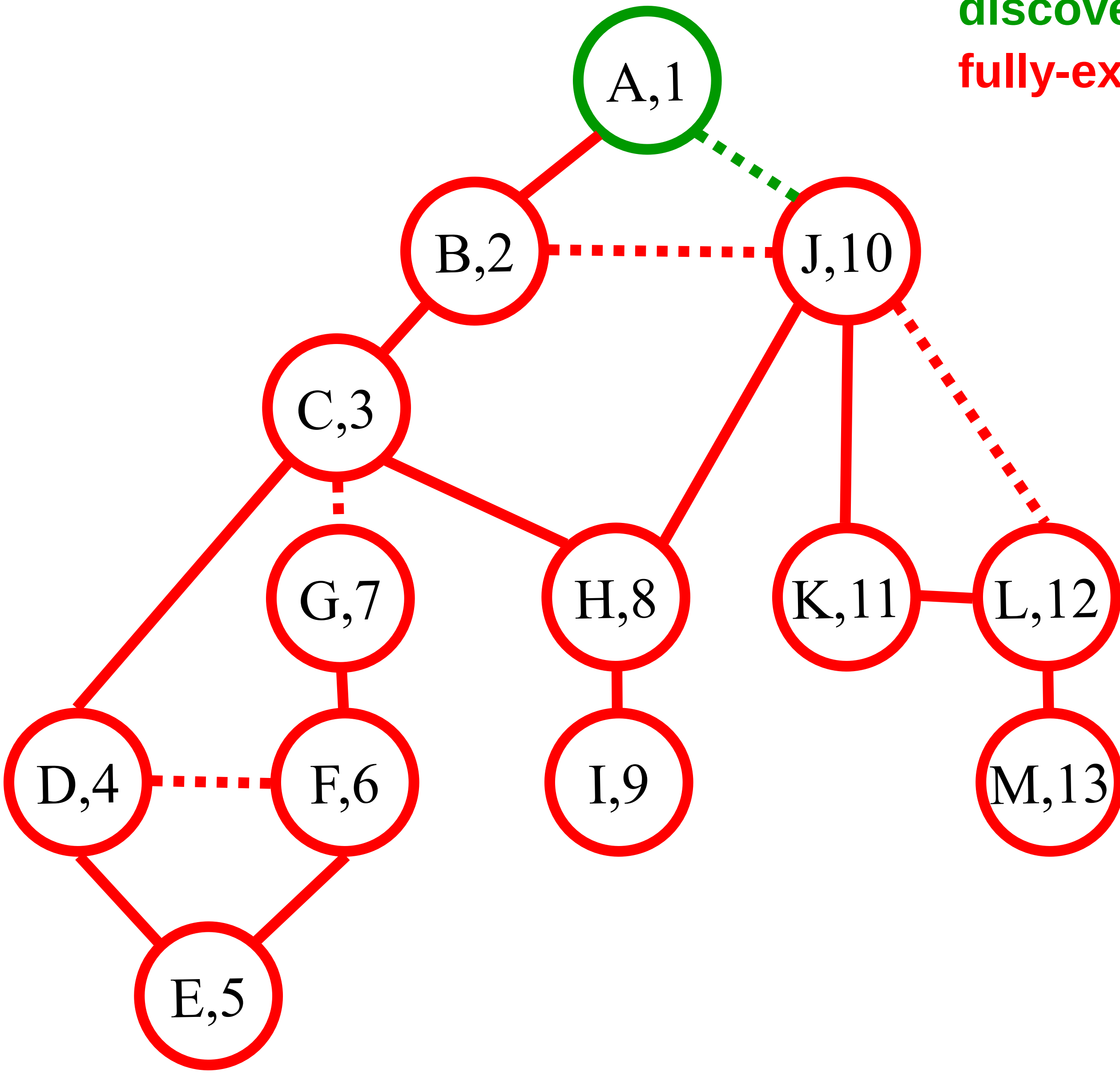
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

A (~~B~~,J)

st[] =
{1}

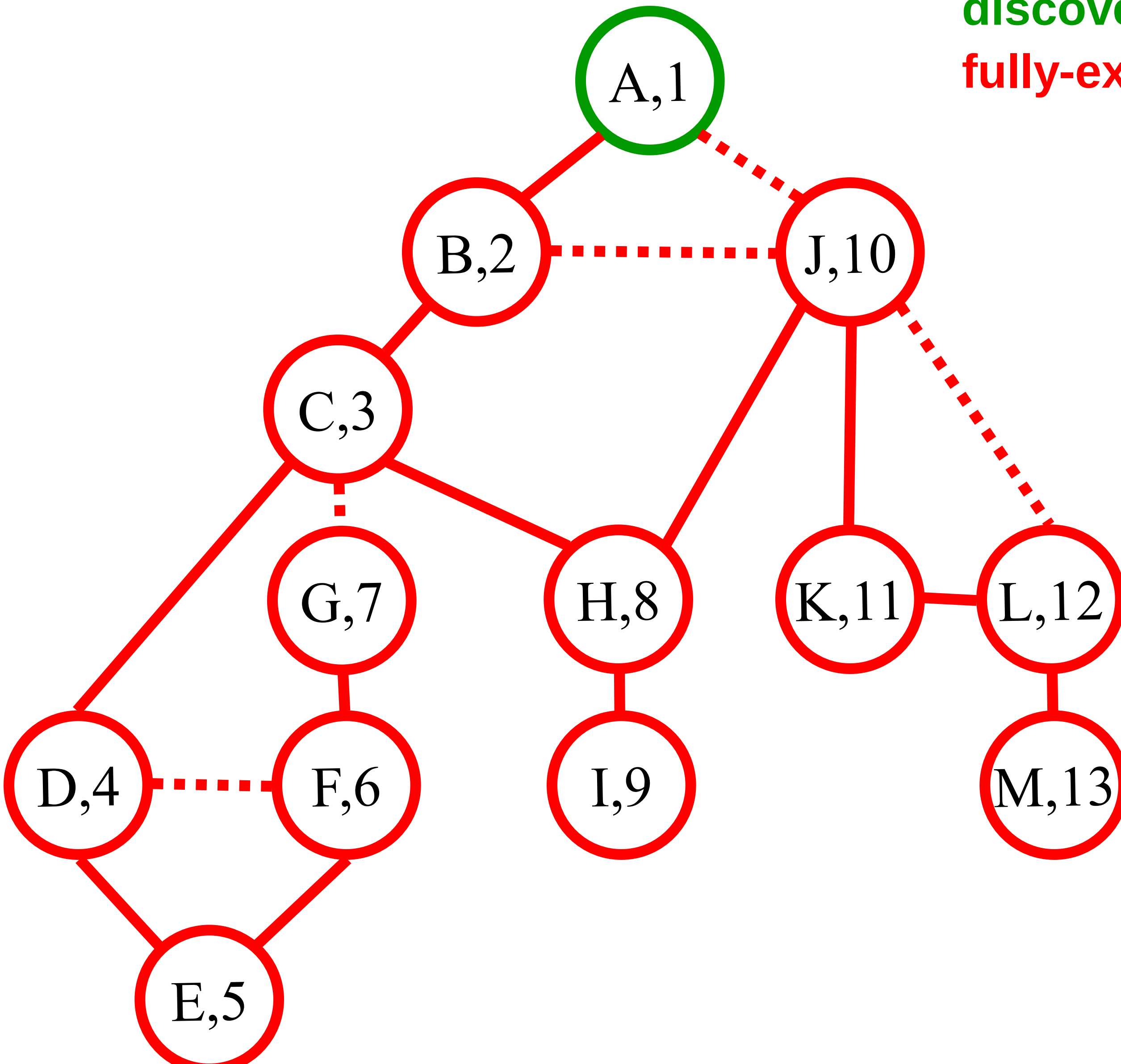
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

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

st[] =
{1}

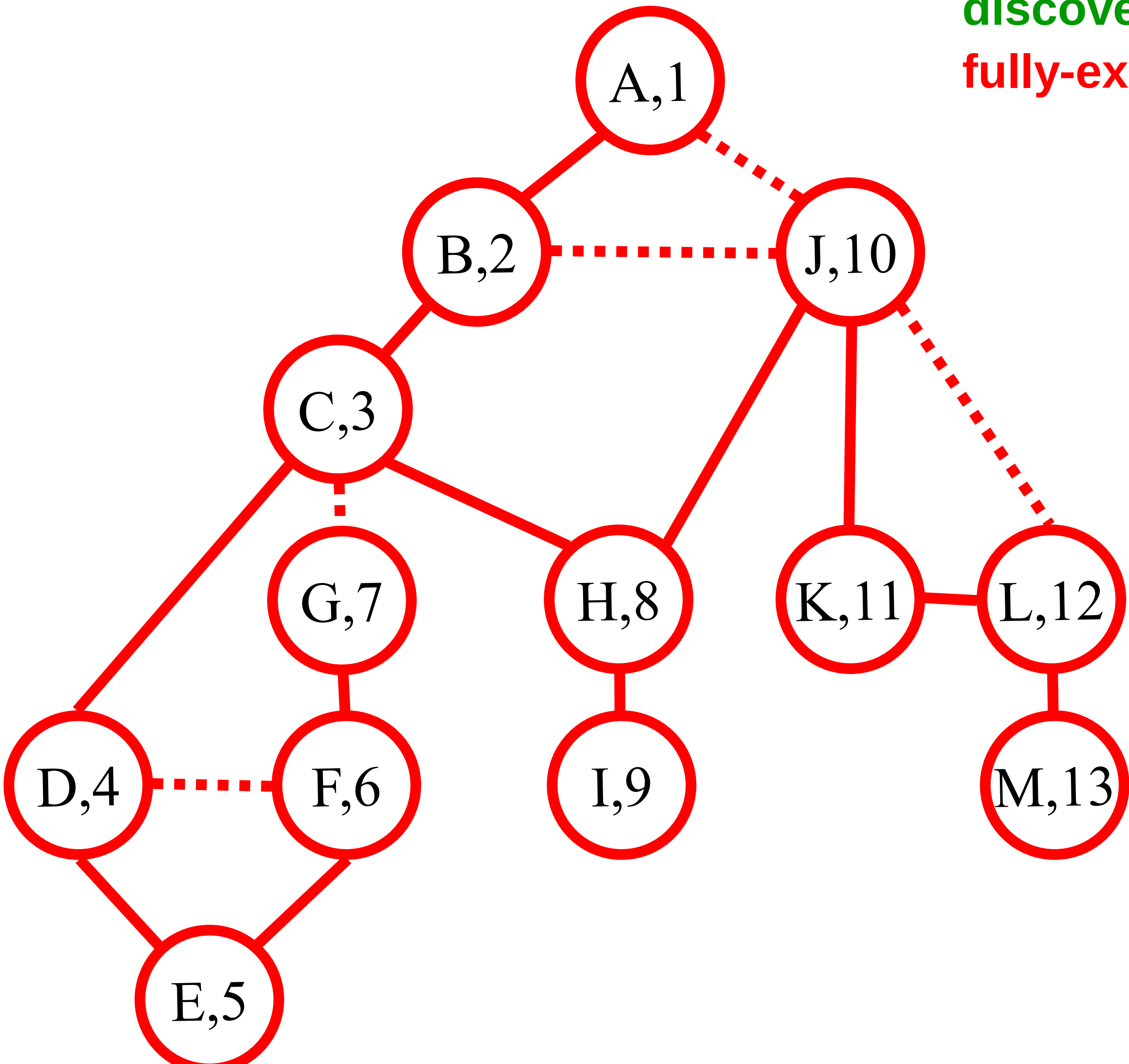
DFS(A)

Color code:

undiscovered

discovered

fully-explored

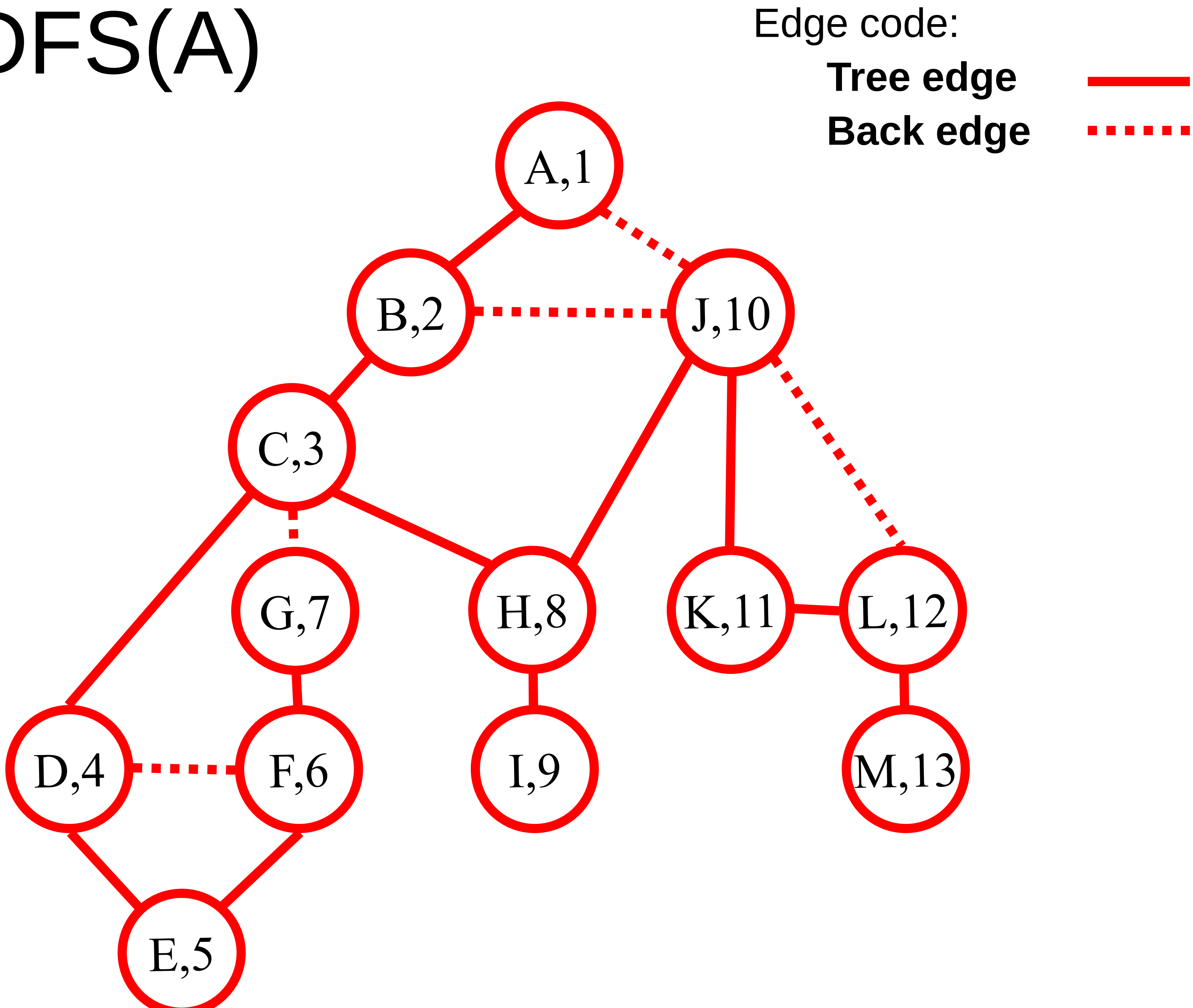


Call Stack:
(Edge list)

TA-DA!!

st[] = {}

DFS(A)


Edge code:
Tree edge
Back edge
A,1
B,2
J,10
C,3
G,7
H,8
K,11
L,12
D,4
F,6
I,9
M,13
E,5

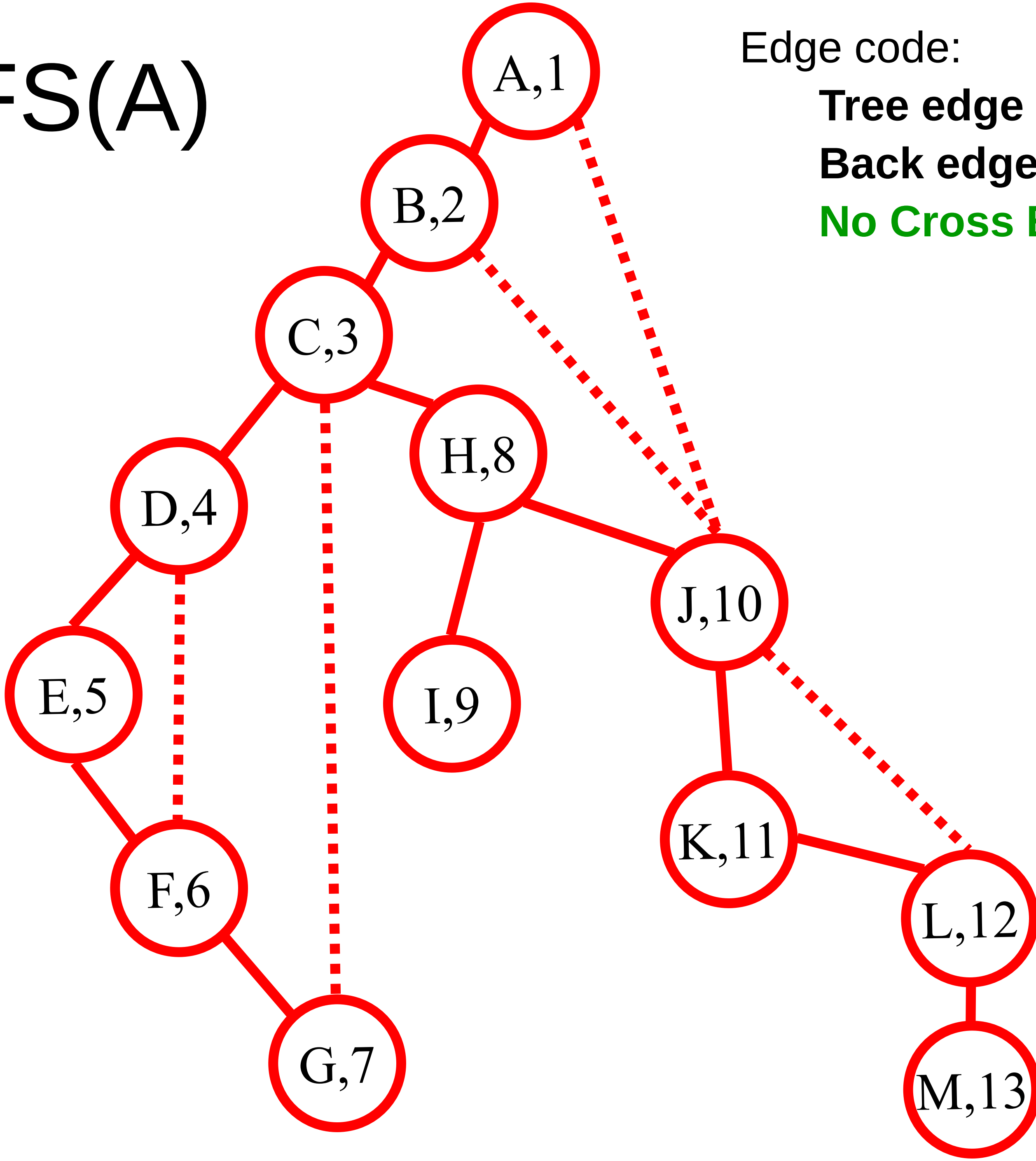
DFS(A)

Edge code:

Tree edge

Back edge

No Cross Edges!



Properties of (undirected) DFS

Like BFS(s):

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

  So, we can use DFS to find connected components

- Edges into then-undiscovered vertices define a ***tree*** – the "depth first spanning tree" of G

Unlike the BFS tree:

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

Non-Tree Edges in DFS

Lemma: For every edge $\{x, y\}$, if $\{x, y\}$ is not in DFS tree, then one of x or y is an ancestor of the other in the tree.

Proof:

Suppose that x is visited first.

Therefore DFS(x) was called before DFS(y)

Since $\{x, y\}$ is not in DFS tree, y was visited when the edge $\{x, y\}$ was examined during DFS(x)

Therefore y was visited during the call to DFS(x) so y is a descendant of x.