CS 401: Computer Algorithm I

BFS / DFS

Xiaorui Sun

Graph Traversal

Walk (via edges) from a fixed starting vertex s to all vertices reachable from s.

- Breadth First Search (BFS): Order nodes in successive layers based on distance from s
- Depth First Search (DFS): More natural approach for exploring a maze;

Applications of BFS:

- Finding shortest path for unit-length graphs
- Finding connected components of a graph
- Testing bipartiteness

Properties of BFS

- BFS(s) visits a vertex v if and only if there is a path from s to v
- 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 levels of the tree

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 $A[u] = c$ 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.

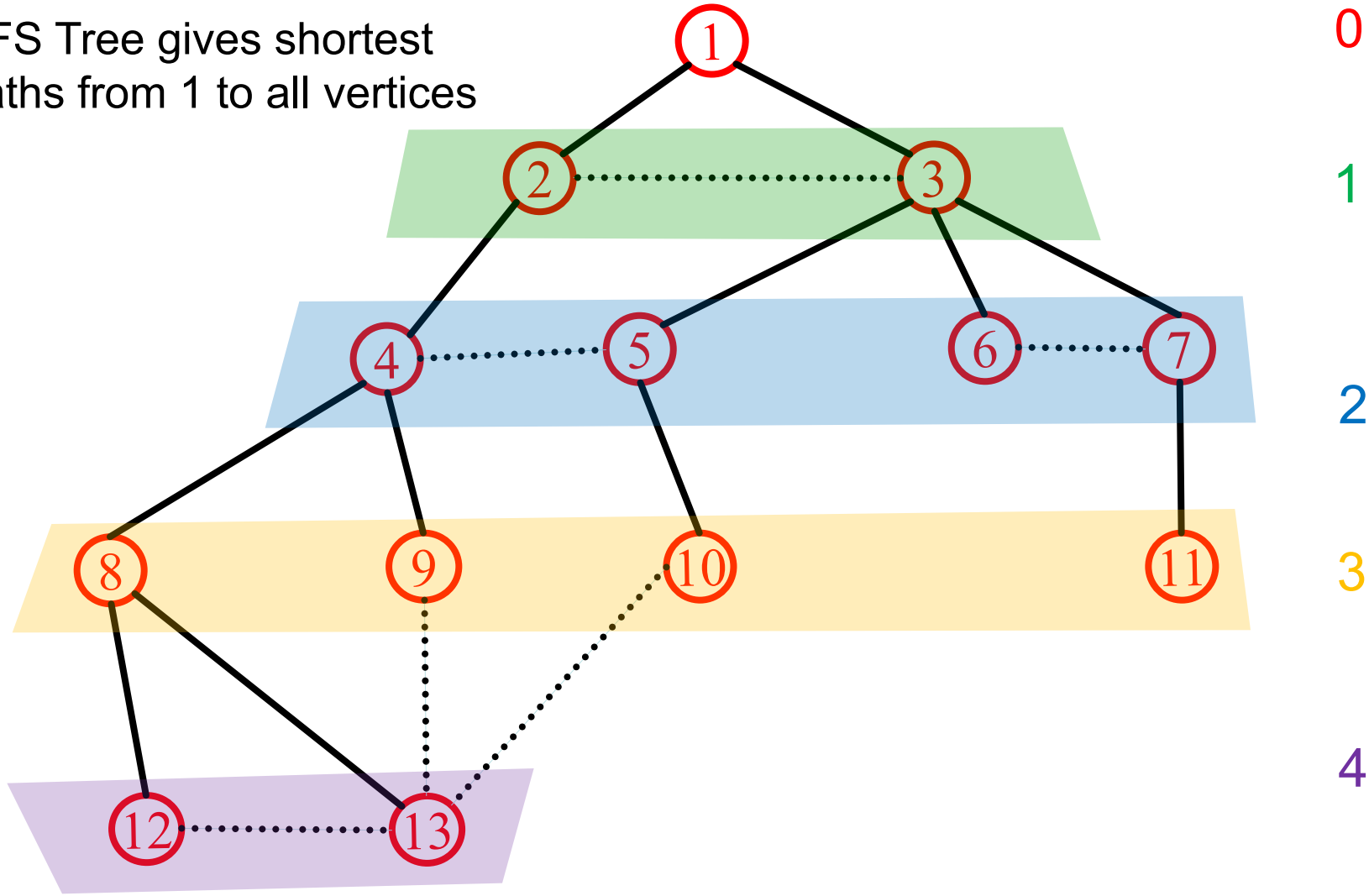
We can use the previous algorithm to detect connected components.

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.

BFS Application: Shortest Paths

BFS Tree gives shortest paths from 1 to all vertices



Bipartite Graphs

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

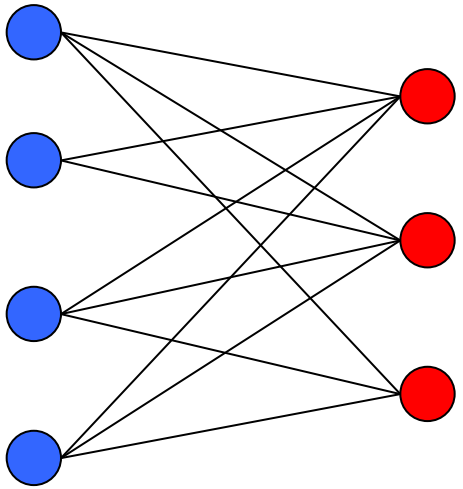
if you can partition the node set into 2 parts (say, blue/red or left/right) so that

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

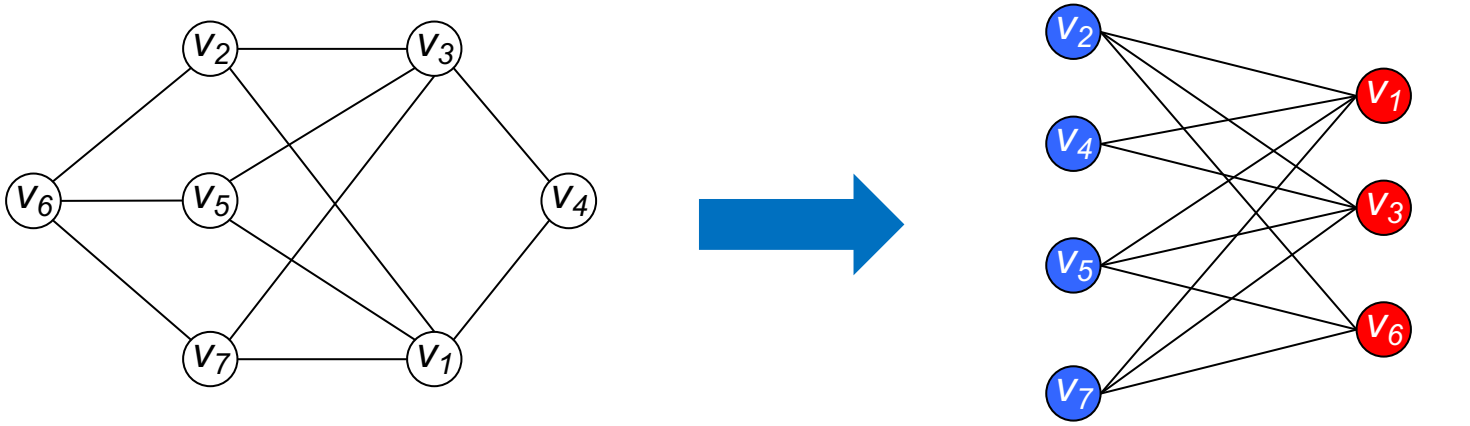
Testing Bipartiteness

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

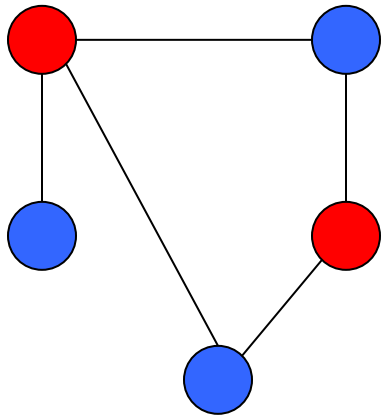
another drawing of G

An Obstruction to Bipartiteness

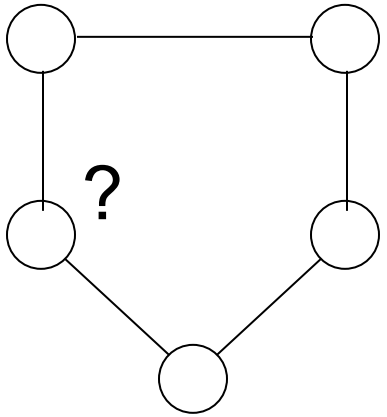
Lemma: If a graph contains an odd length cycle, then the graph is not bipartite.

(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)



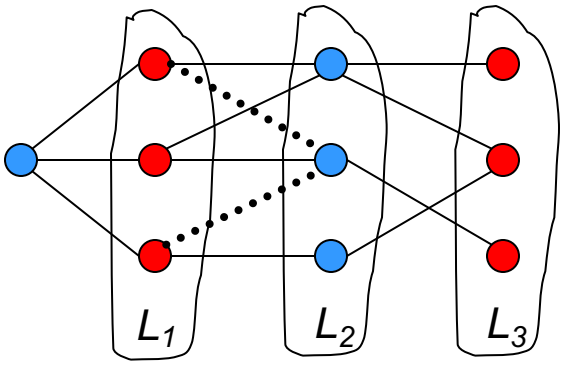
not bipartite
(not 2-colorable)

A Characterization of Bipartite Graphs

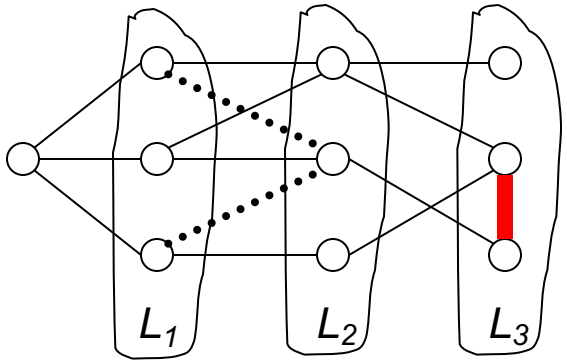
Lemma: Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS(s). Exactly one of the following holds.

(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.

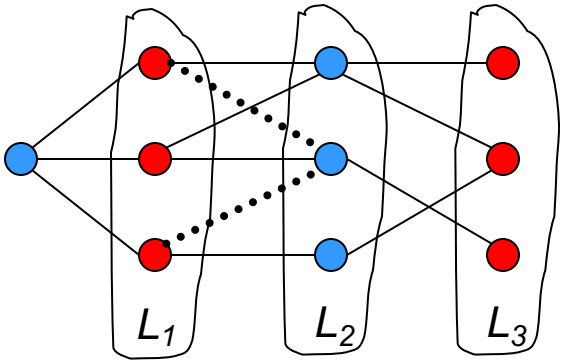
(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(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.

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)

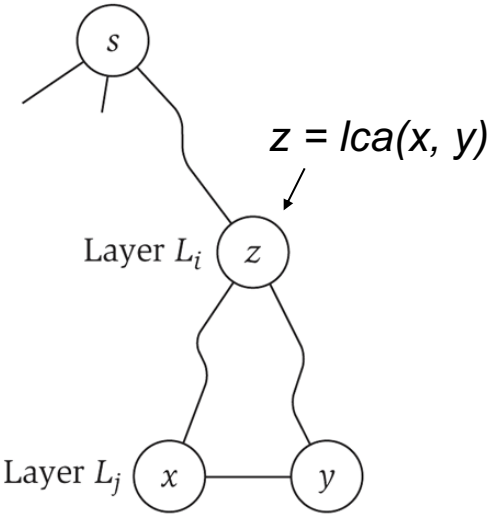
Suppose $\{x, y\}$ is an edge & x, y in same level L_j.

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.

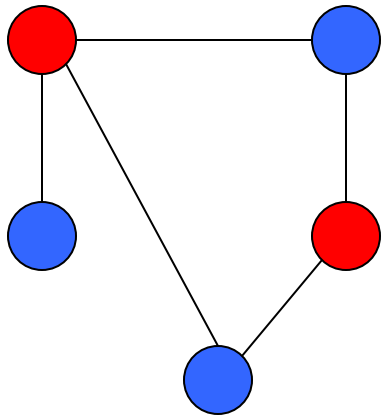
Its length is $1 + (j - i) + (j - i)$, which is odd.



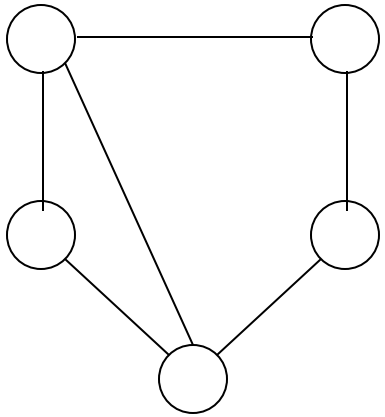
Obstruction to Bipartiteness

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)

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.

Bipartiteness testing algorithm:

- Run BFS with an arbitrary start vertex
- Construct the BFS tree with the start vertex as root
- If each non-tree edge of G connects two vertices at different levels in the rooted BFS tree, then output yes.
- Otherwise, output no.

BFS Summary

Breadth First Search (BFS): Explore vertices according to the order of the discovery of vertices

Property:

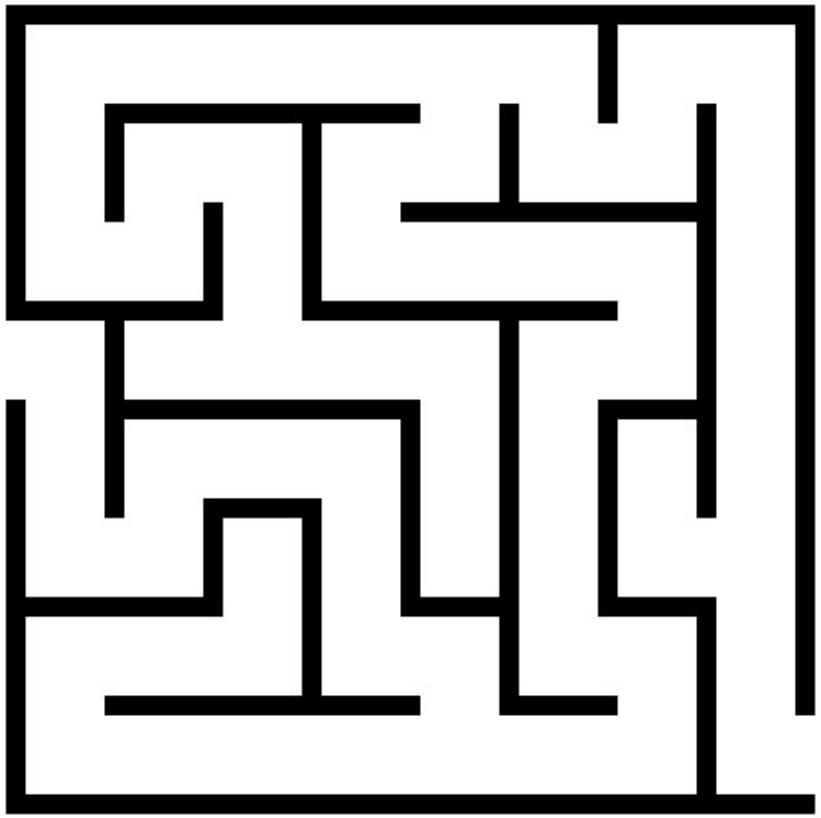
- BFS tree
- Level = distance (length of shortest path) from the initial vertex
- Every edge connect two vertices at the same or adjacent levels

Applications of BFS:

- Finding connected components of a graph
- Finding shortest path for unit-length graphs
- Testing bipartiteness

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)

DFS(x)

Mark v fully-explored

DFS(A)

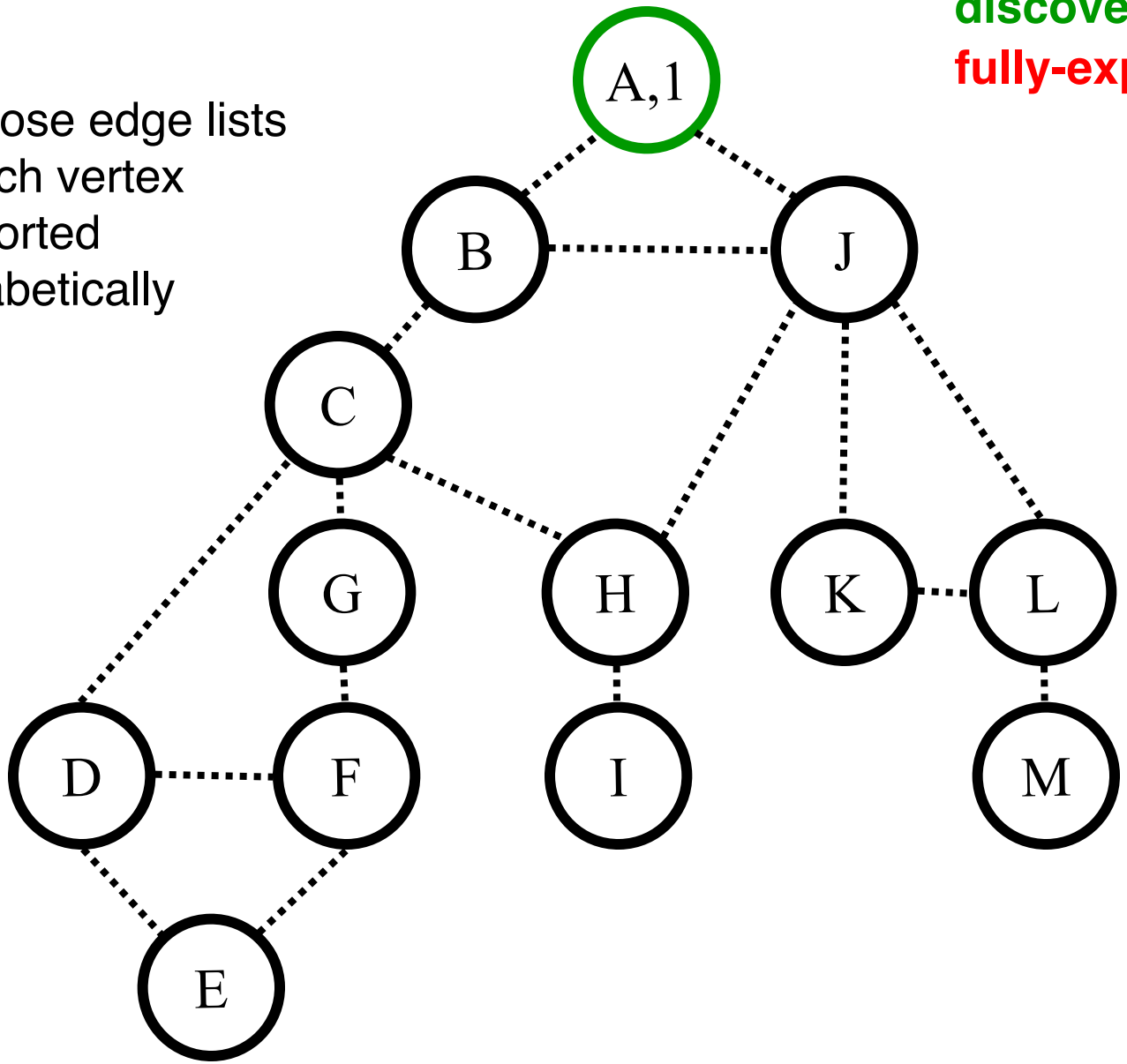
Color code:

undiscovered

discovered

fully-explored

Suppose edge lists at each vertex are sorted alphabetically



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

st[] = {1}

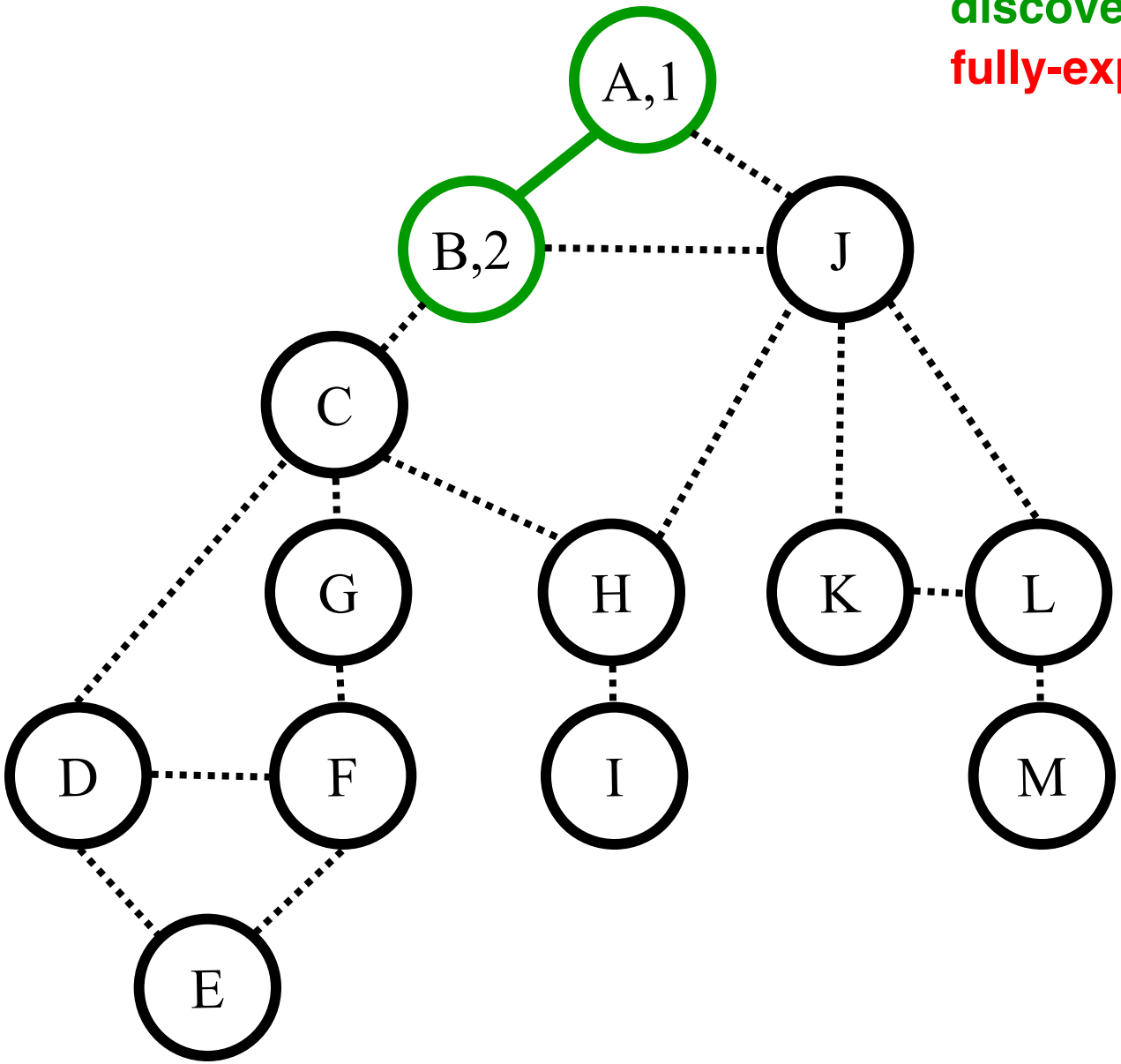
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

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

st[] =
{1,2}

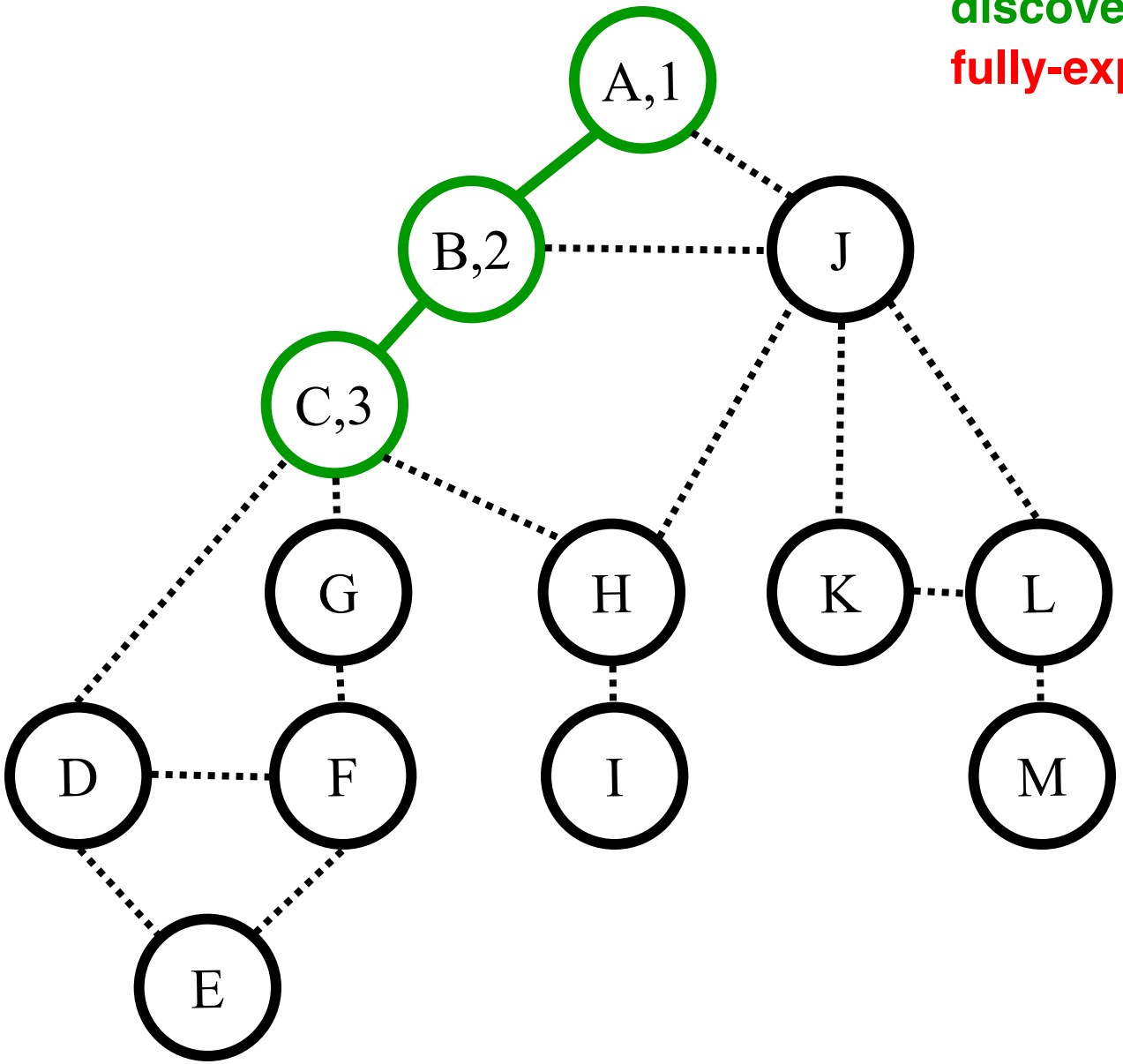
DFS(A)

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}

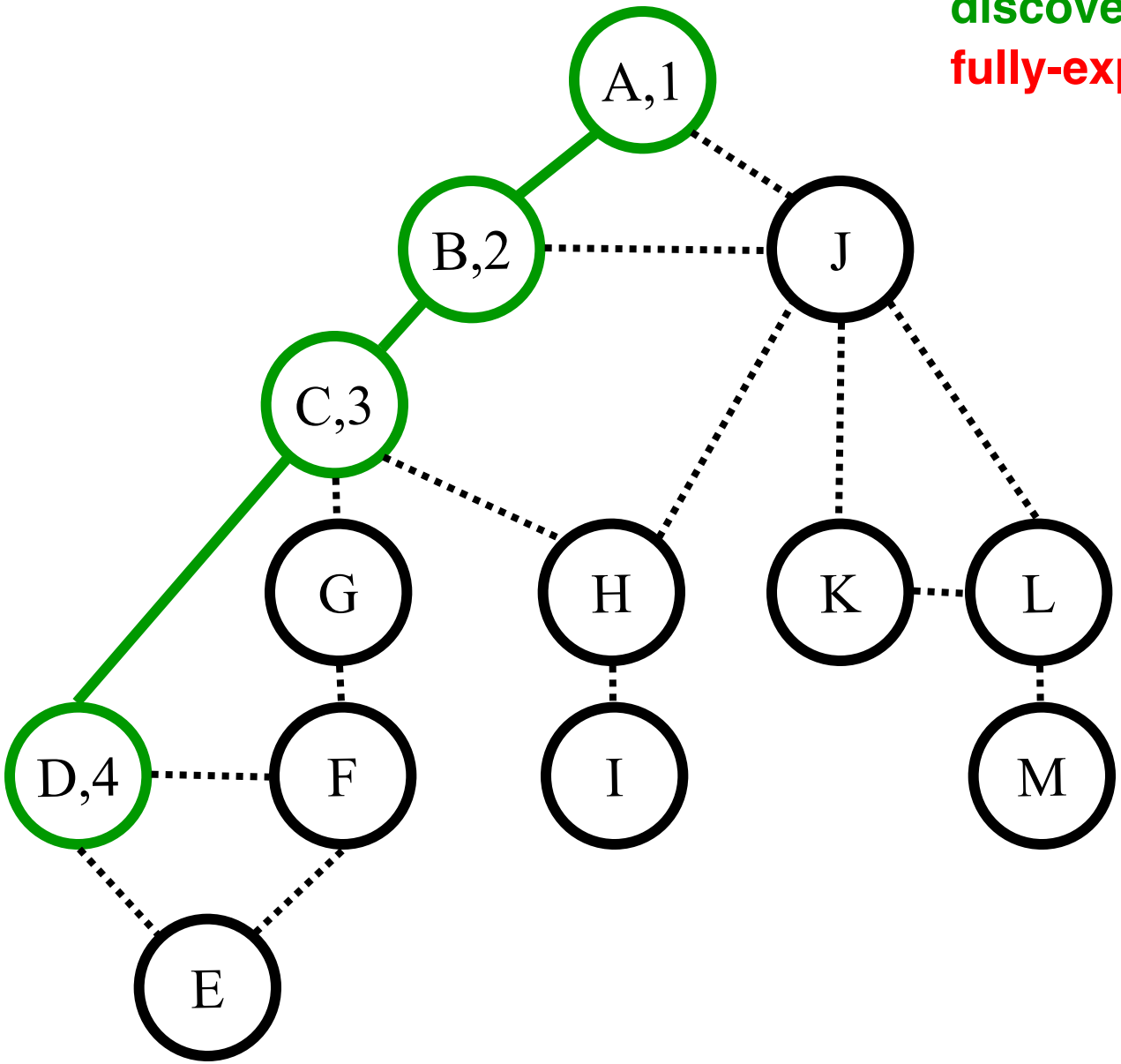
DFS(A)

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}

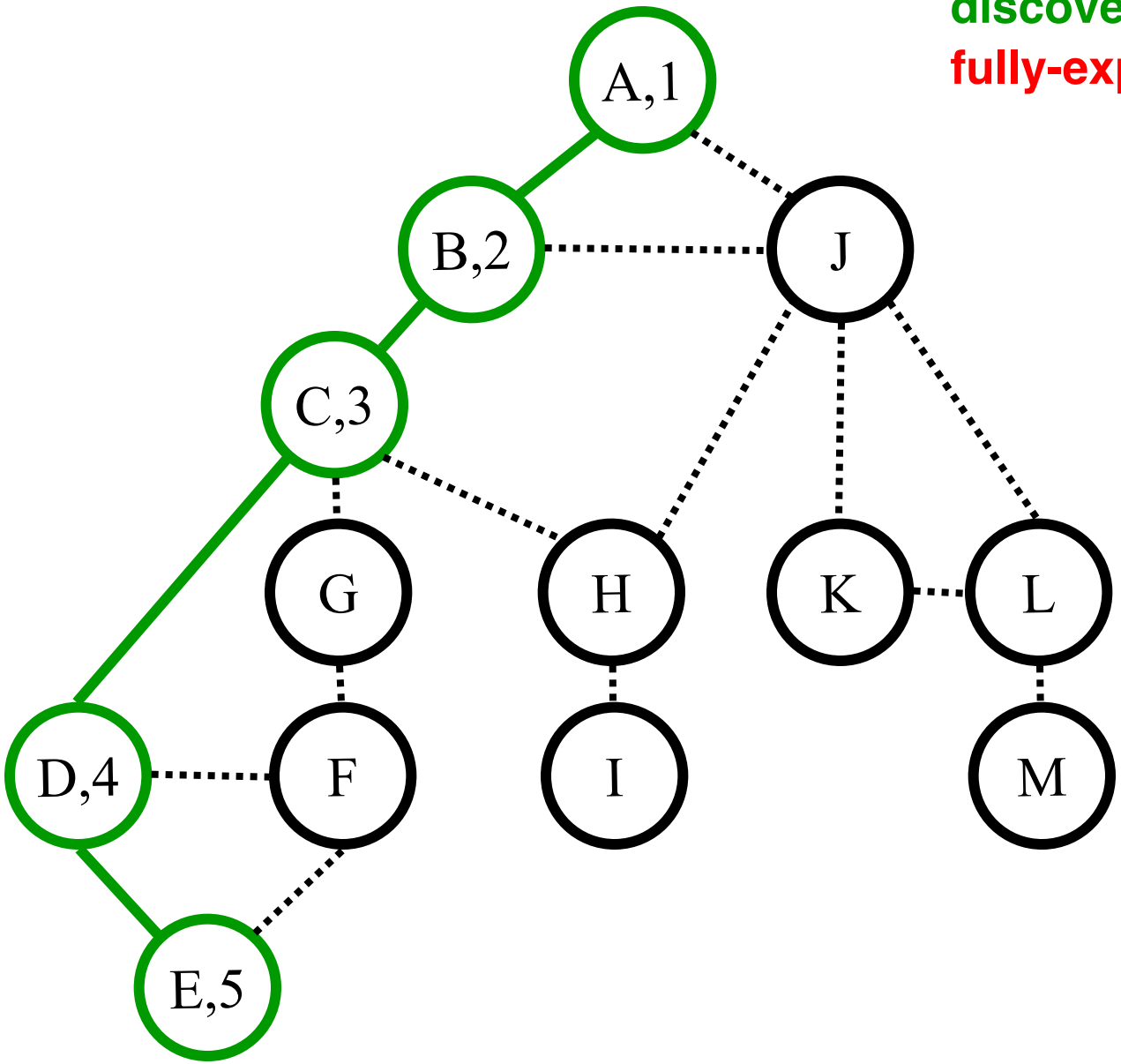
DFS(A)

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}

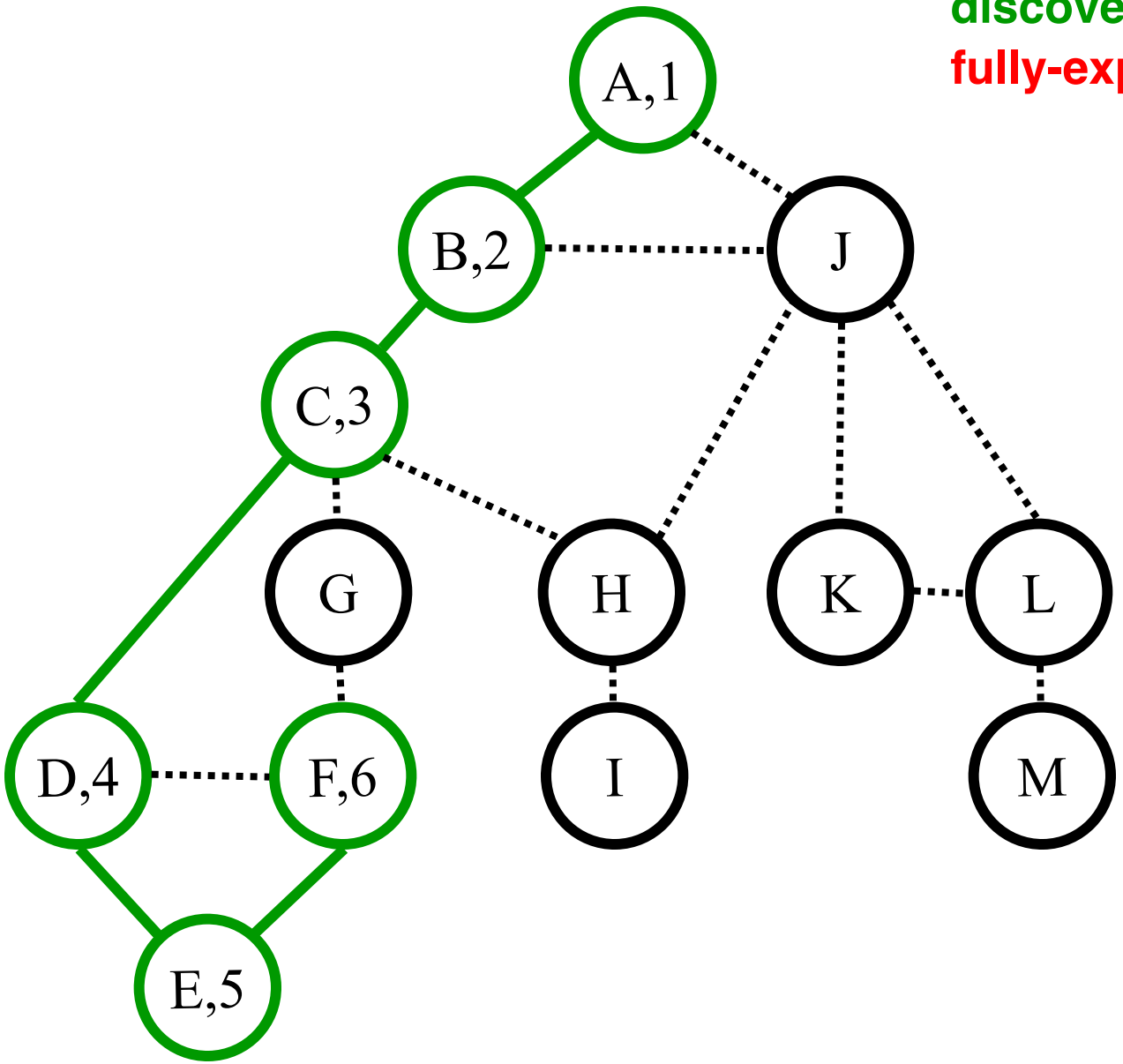
DFS(A)

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}

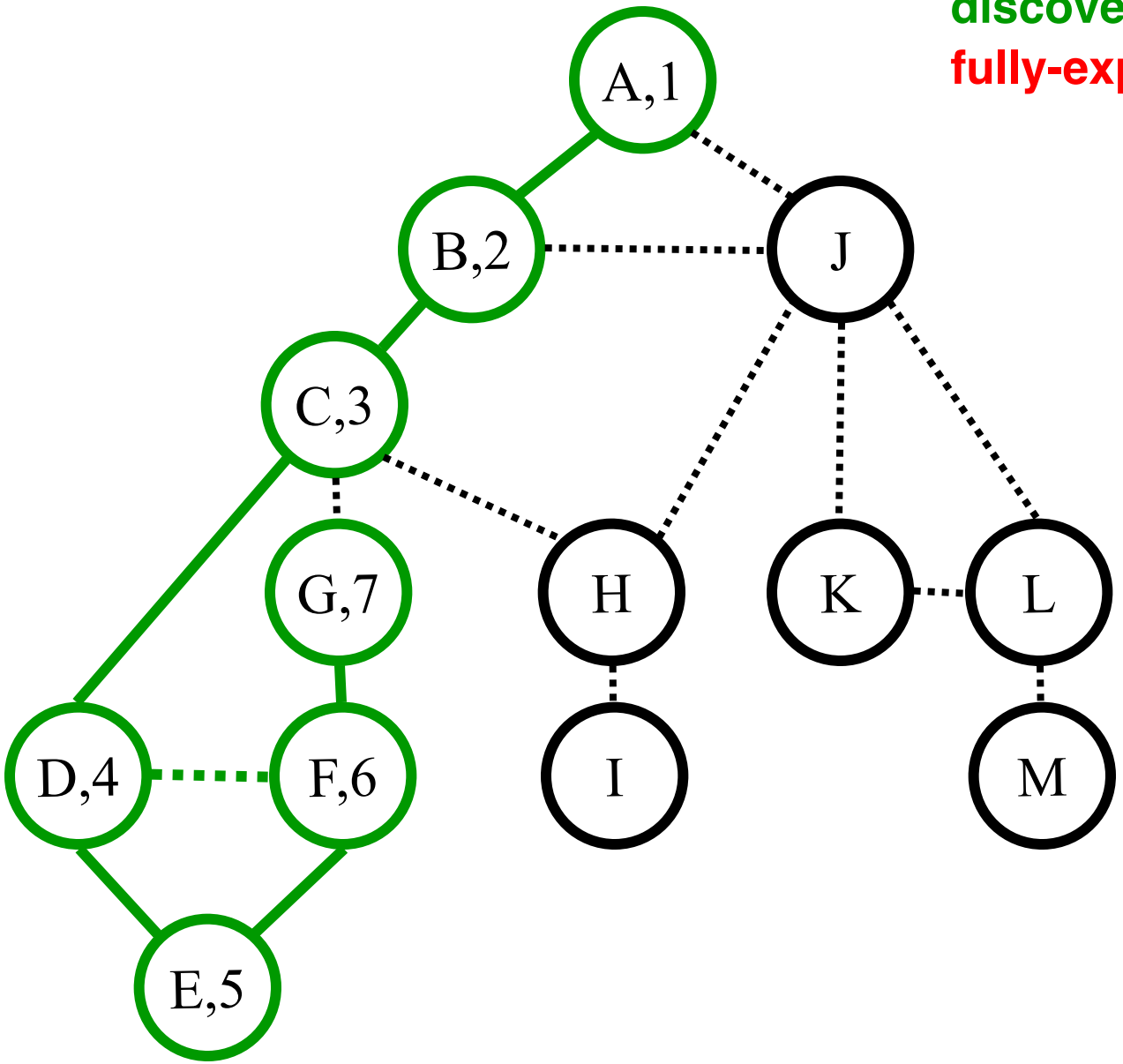
DFS(A)

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}

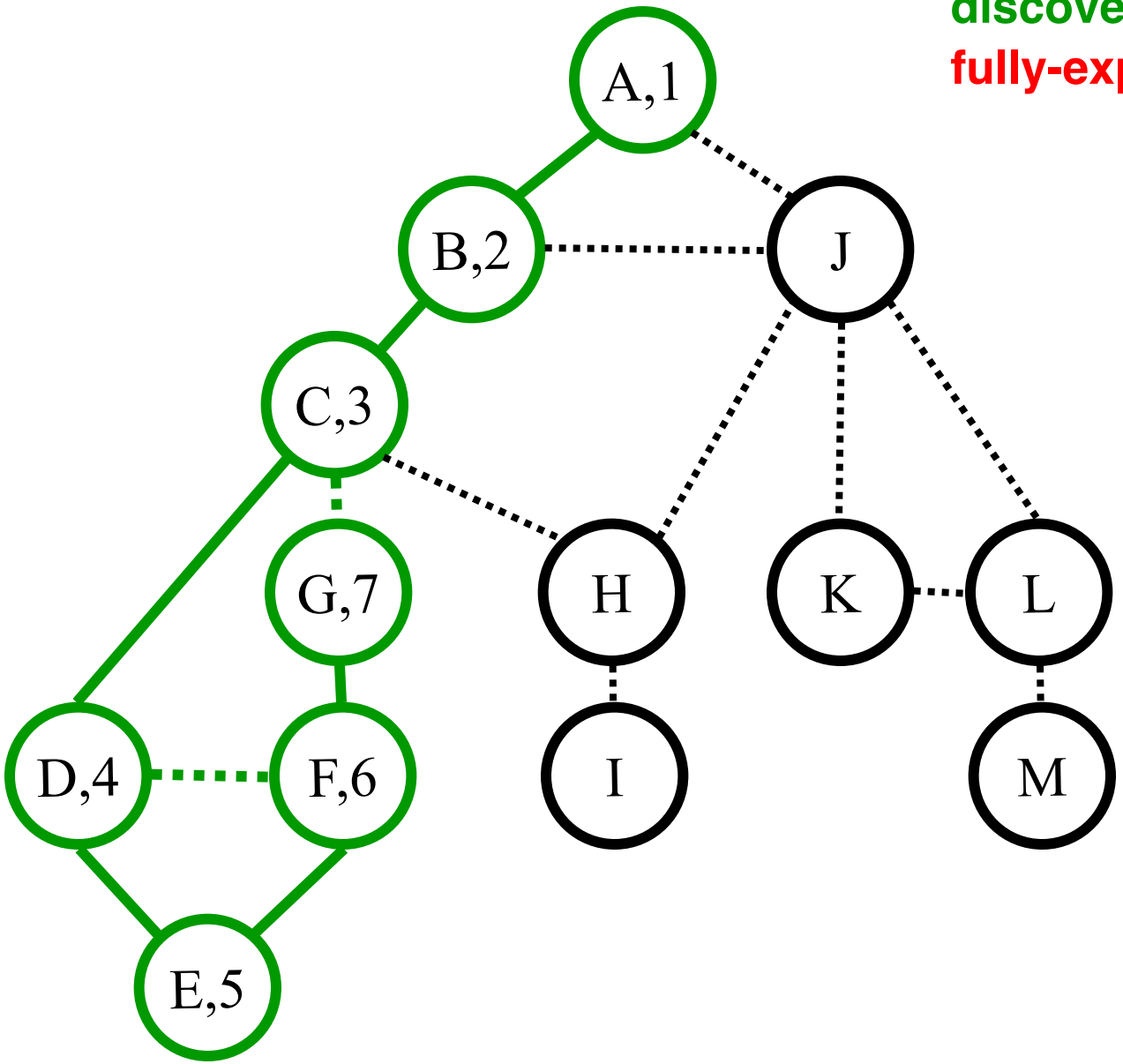
DFS(A)

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}

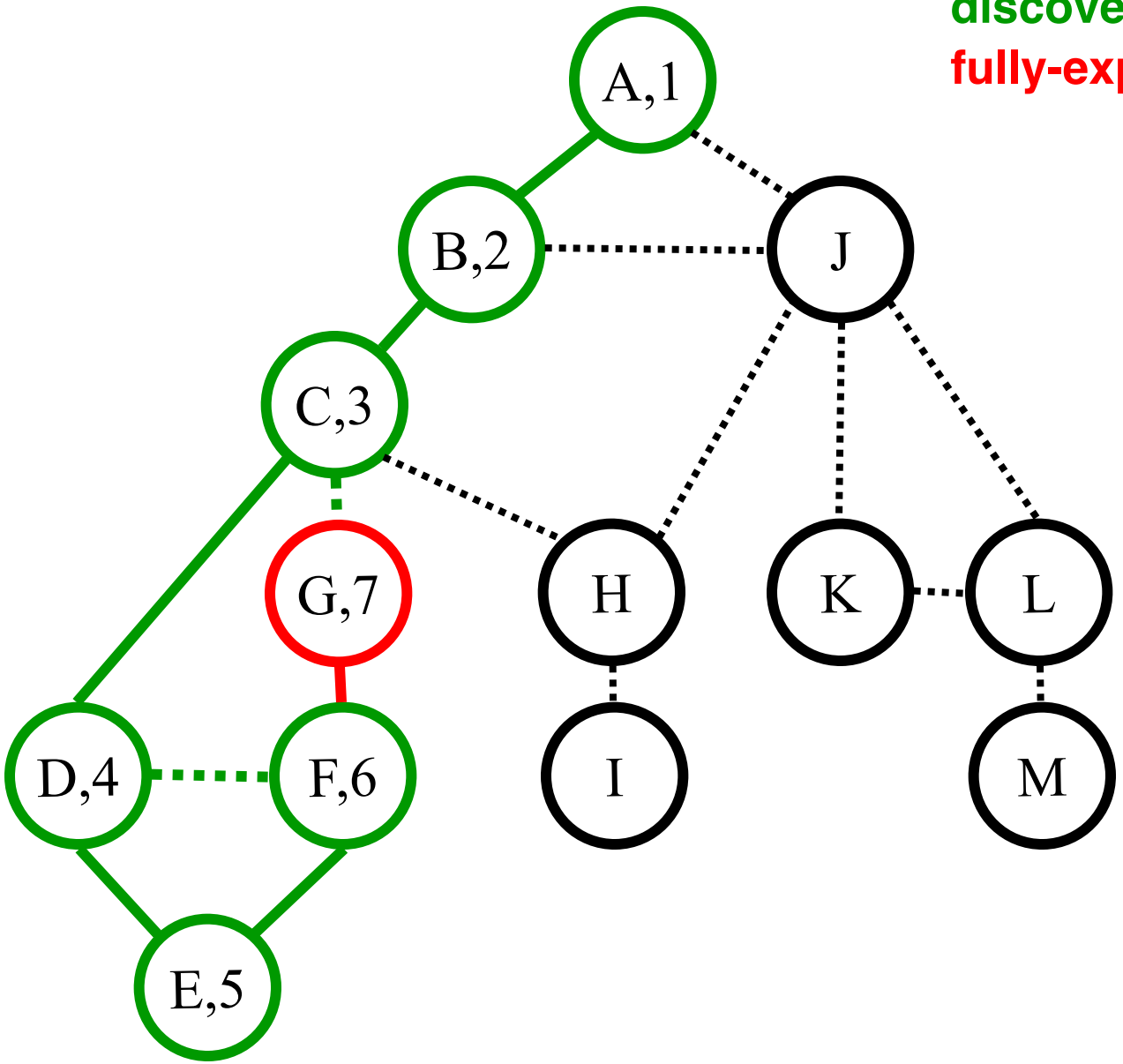
DFS(A)

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}

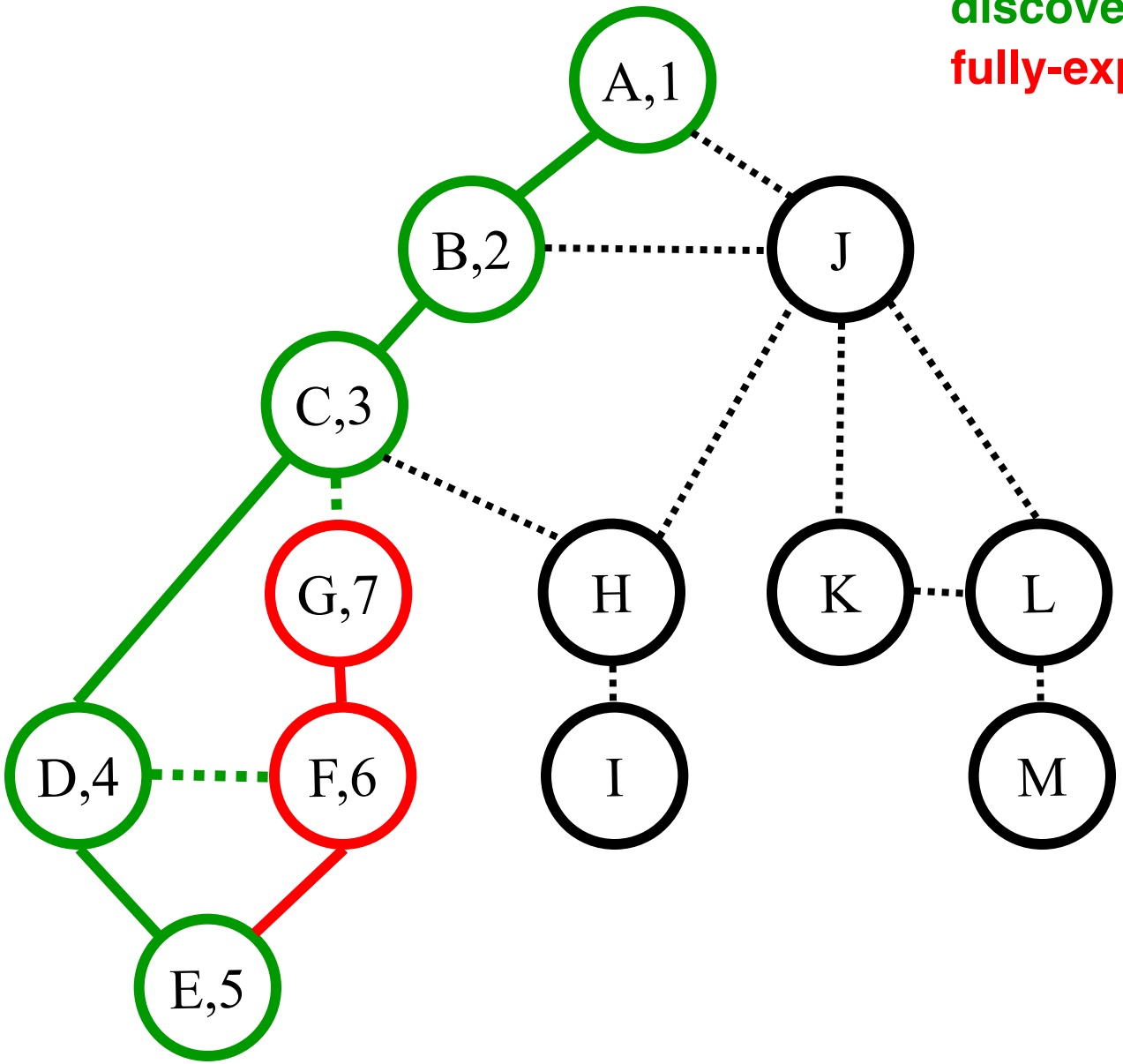
DFS(A)

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}

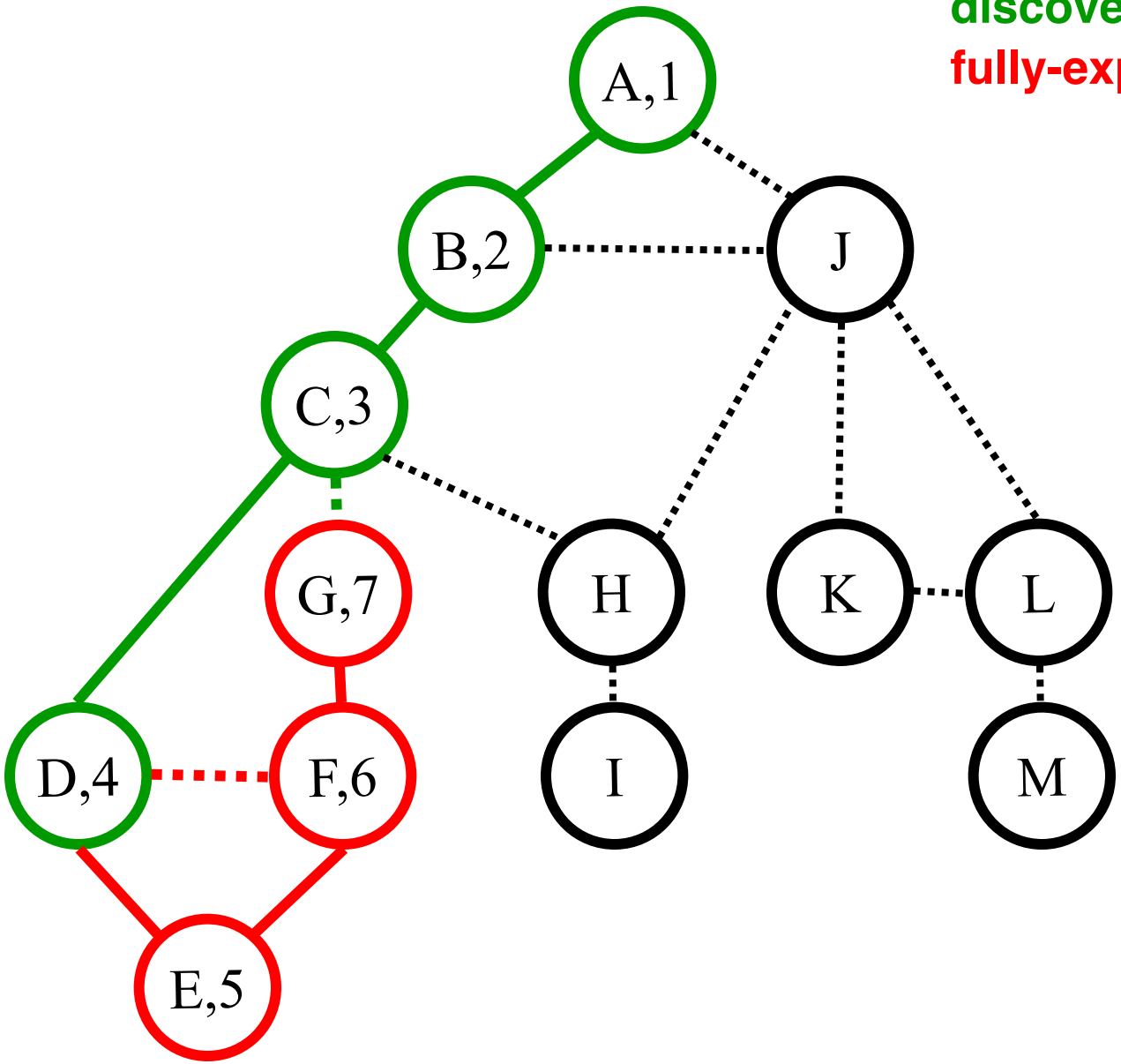
DFS(A)

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}

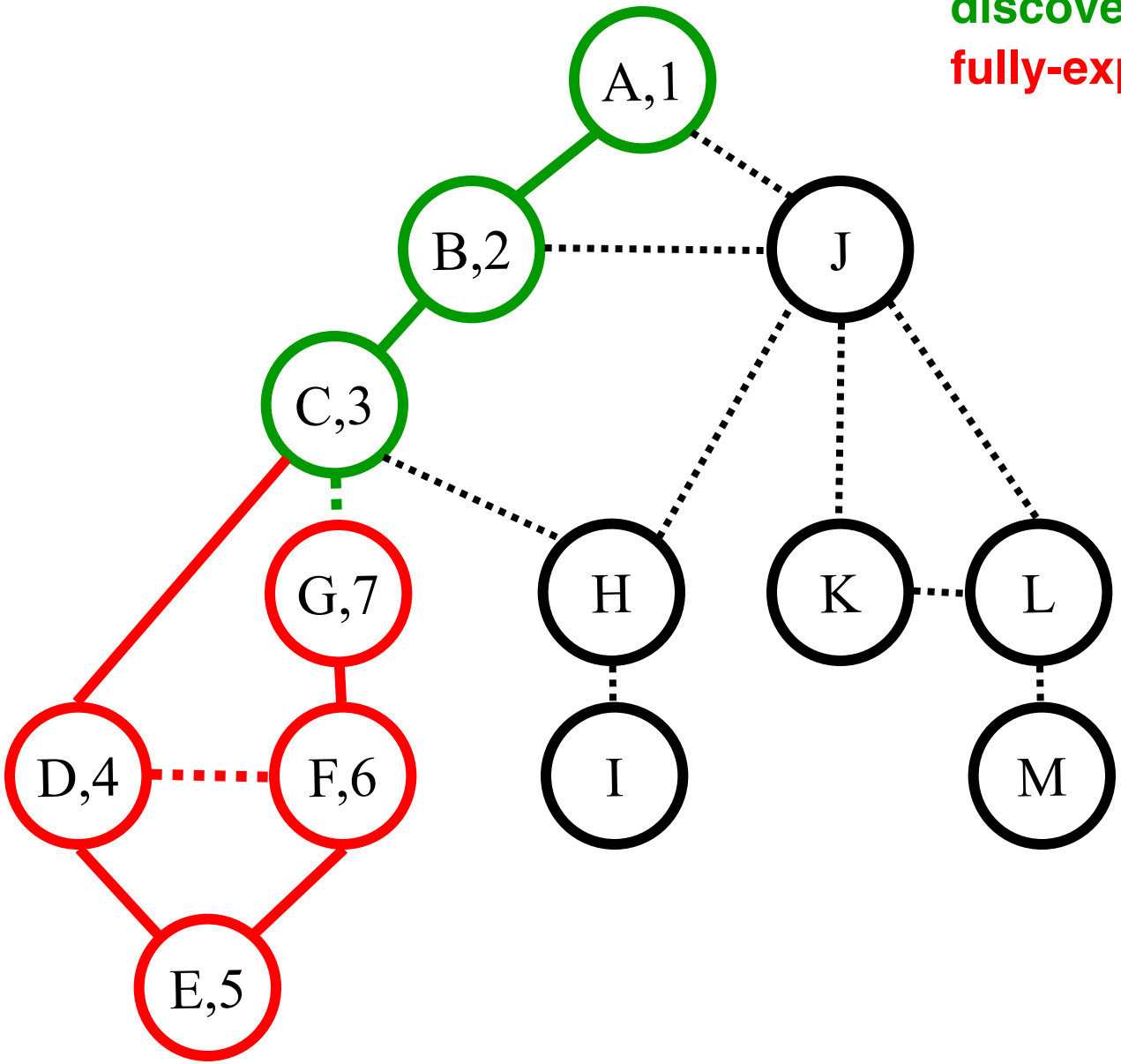
DFS(A)

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}

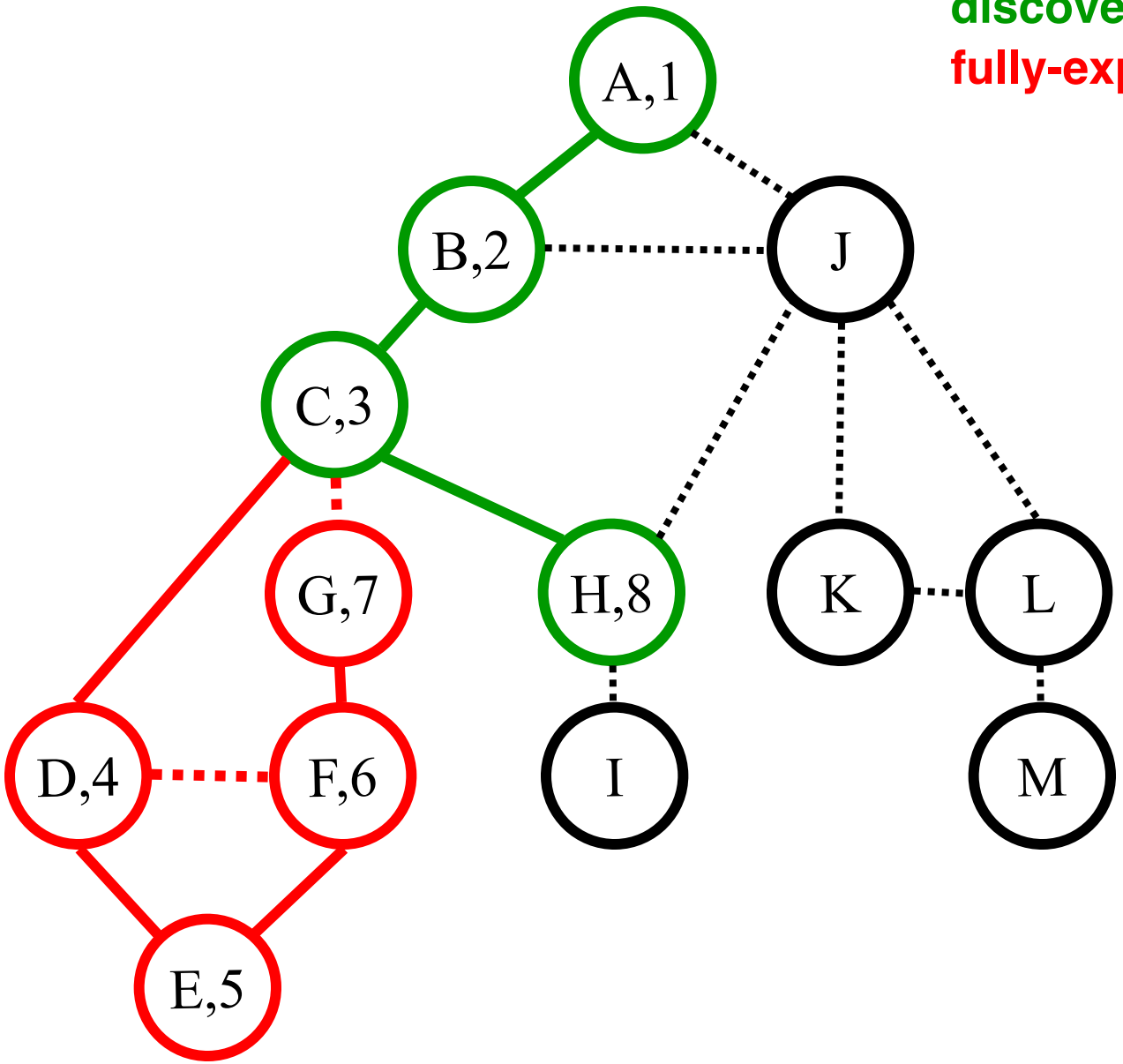
DFS(A)

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}

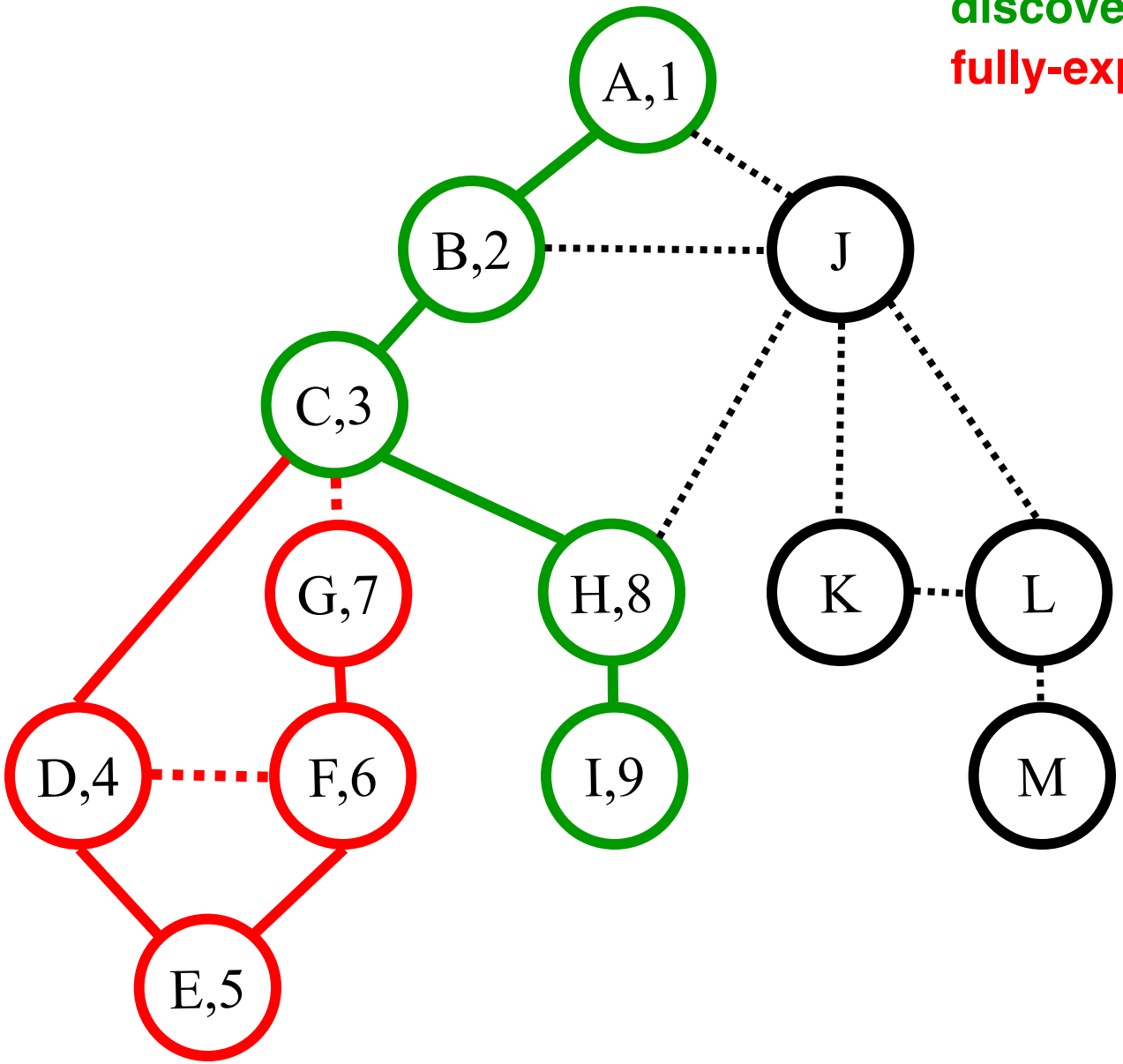
DFS(A)

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}

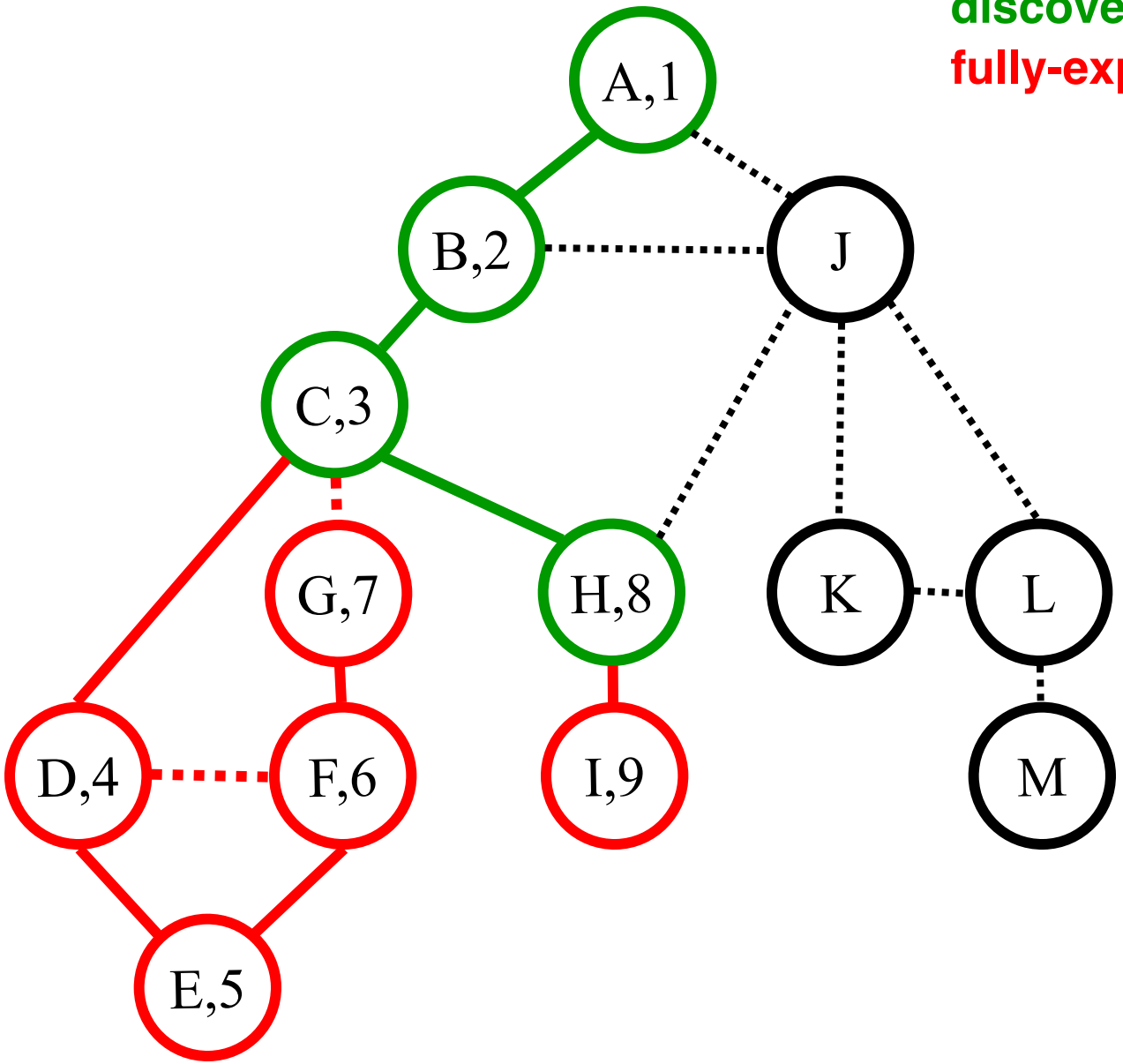
DFS(A)

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}

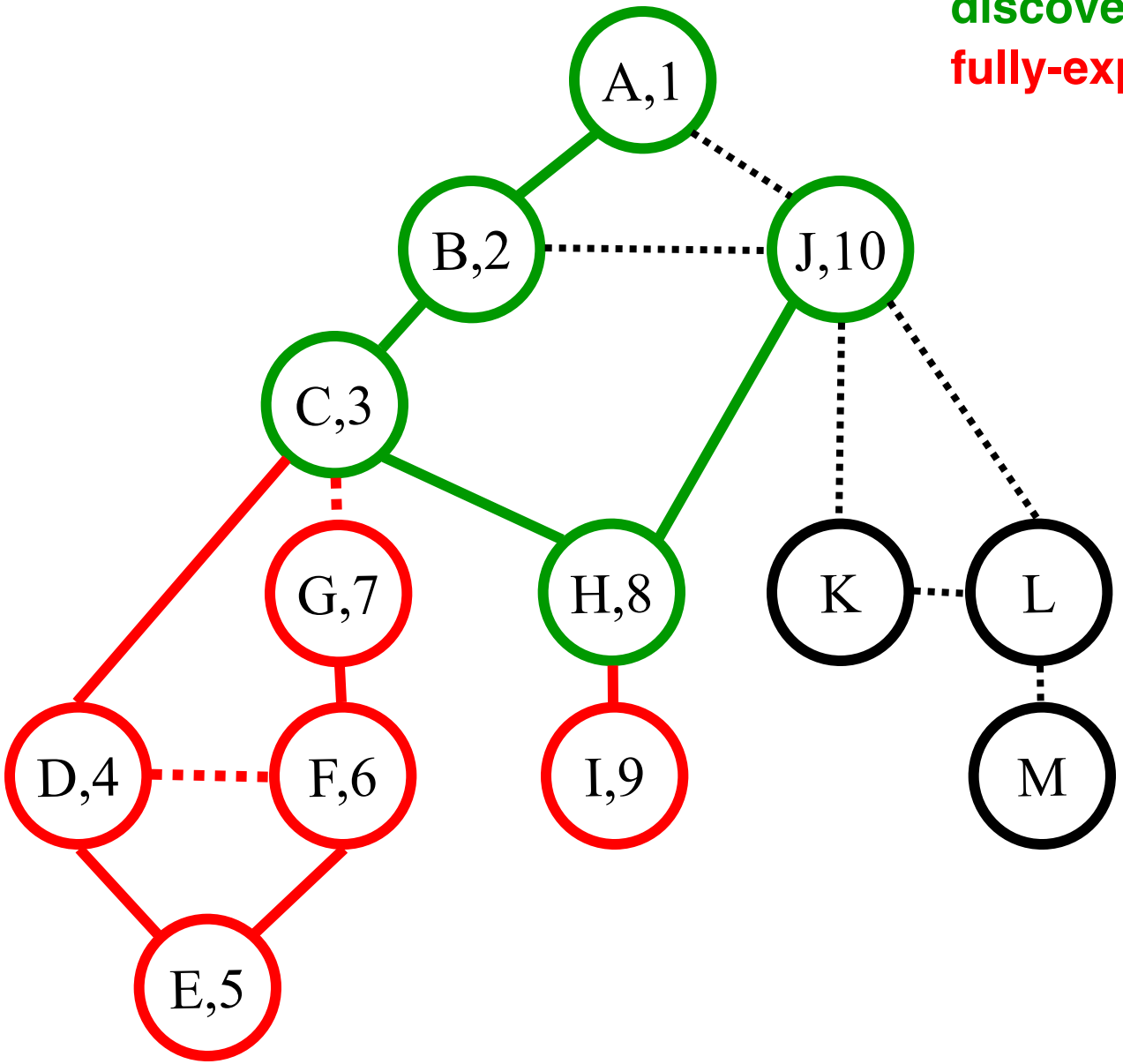
DFS(A)

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}

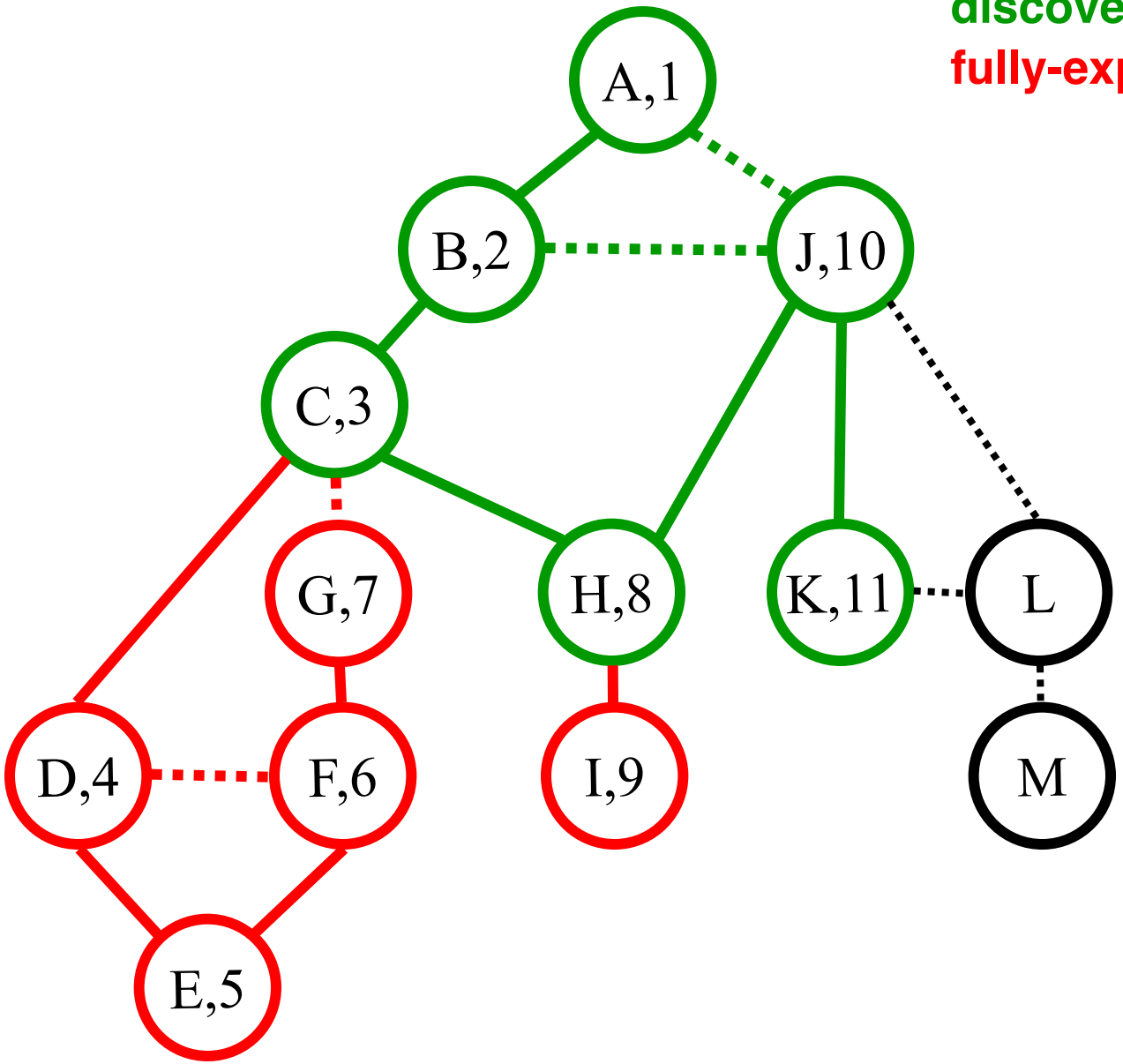
DFS(A)

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~~,~~I~~,K,L)
K (J,L)

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

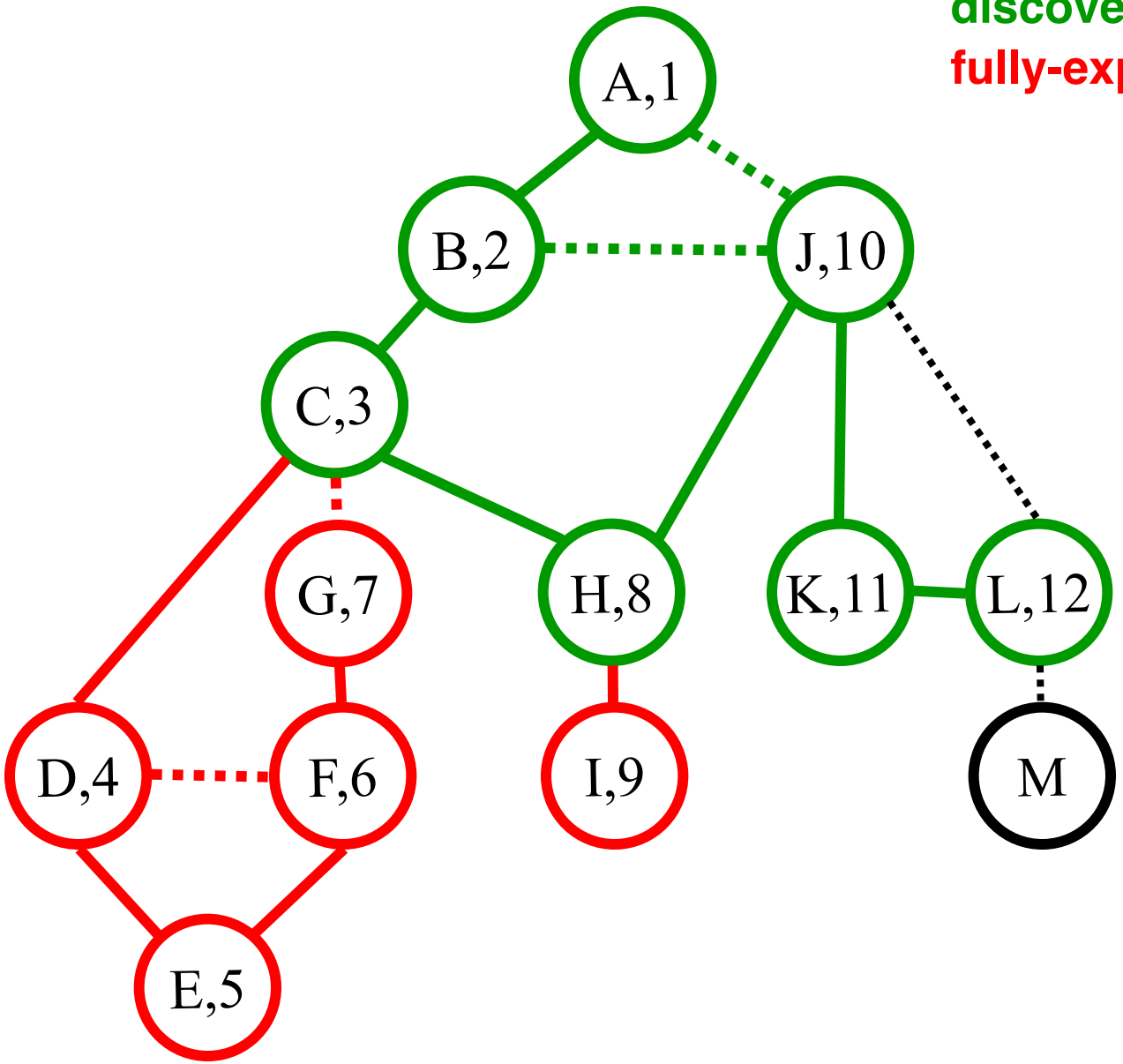
DFS(A)

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}

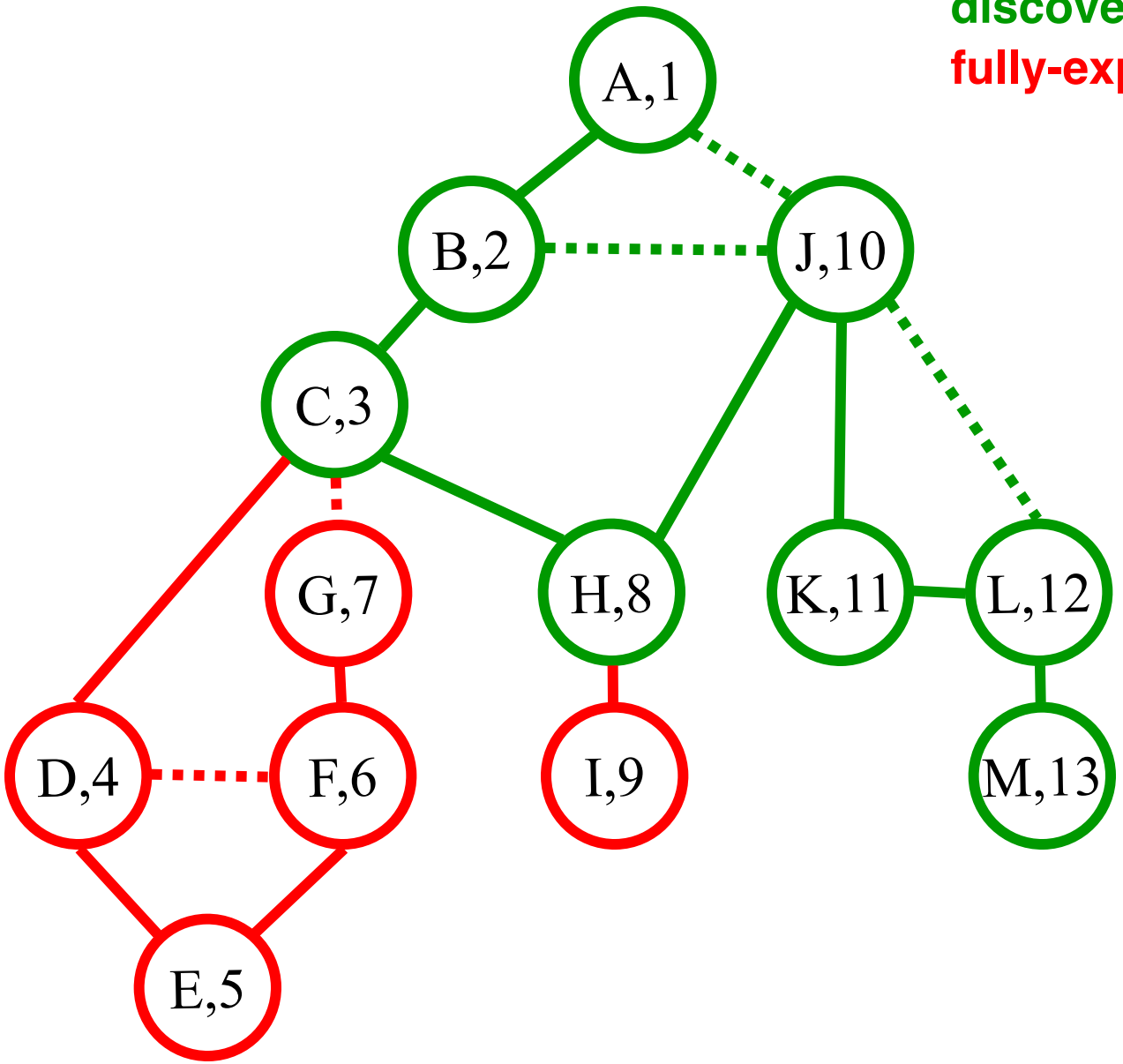
DFS(A)

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}

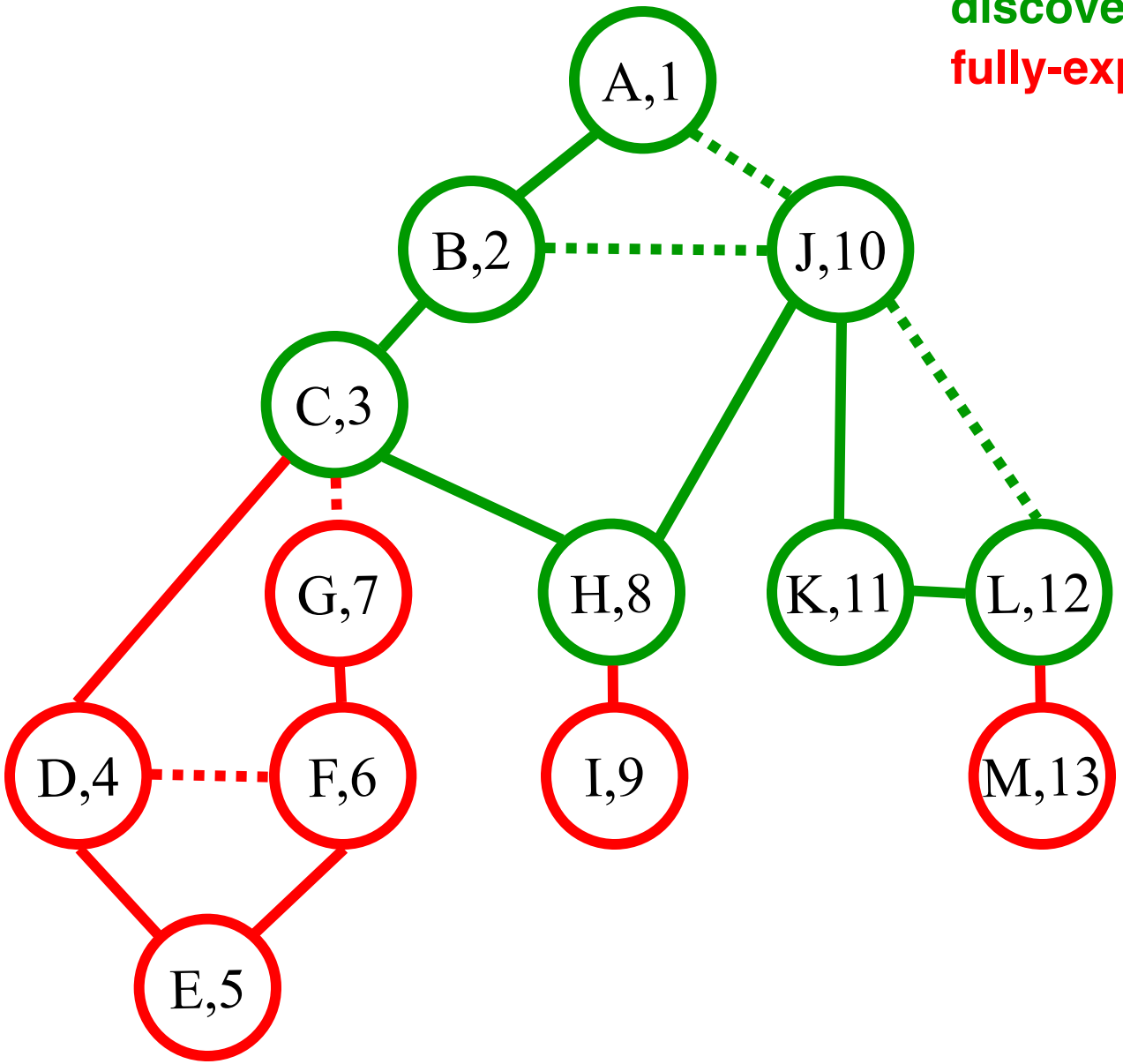
DFS(A)

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}

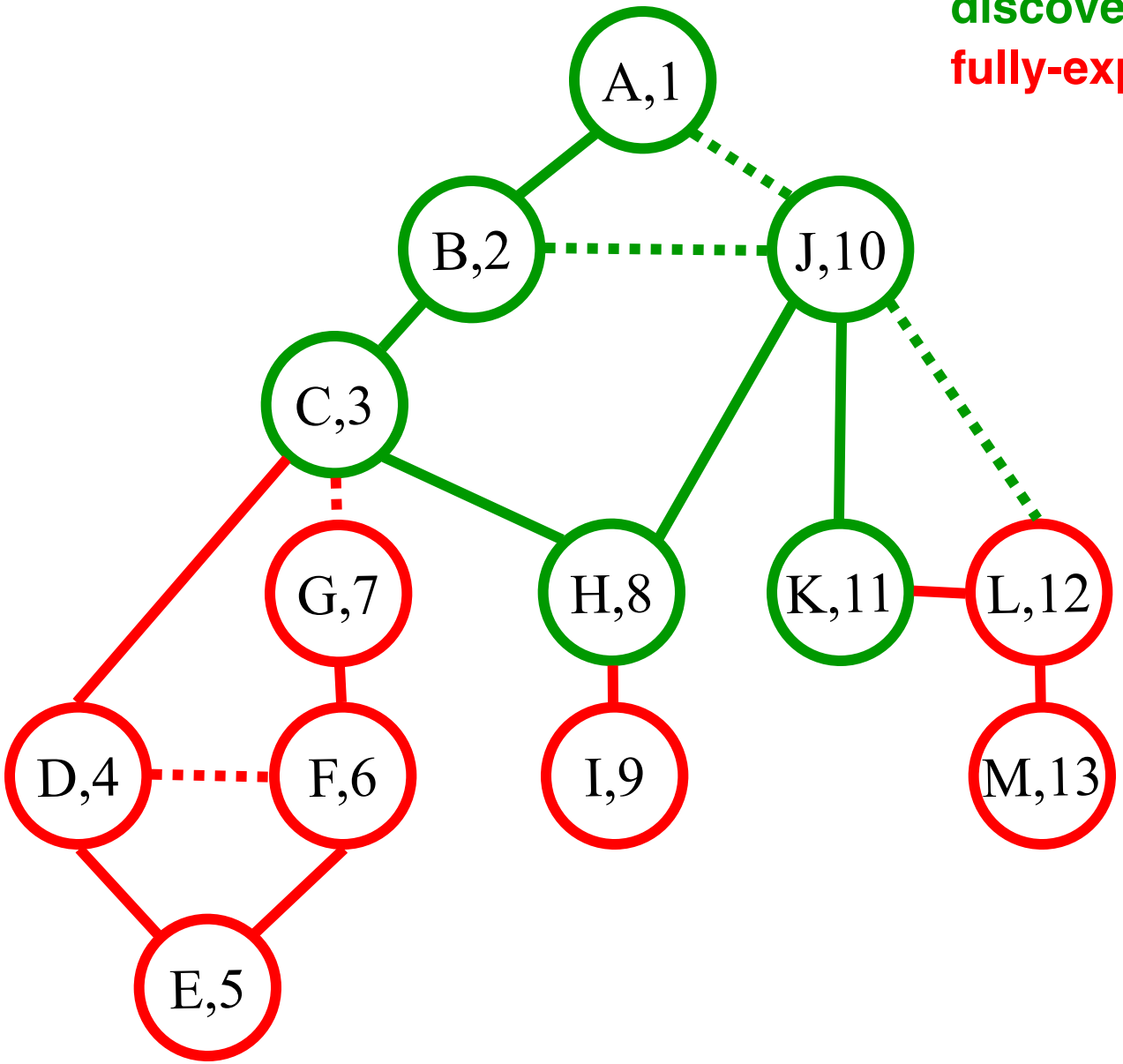
DFS(A)

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}

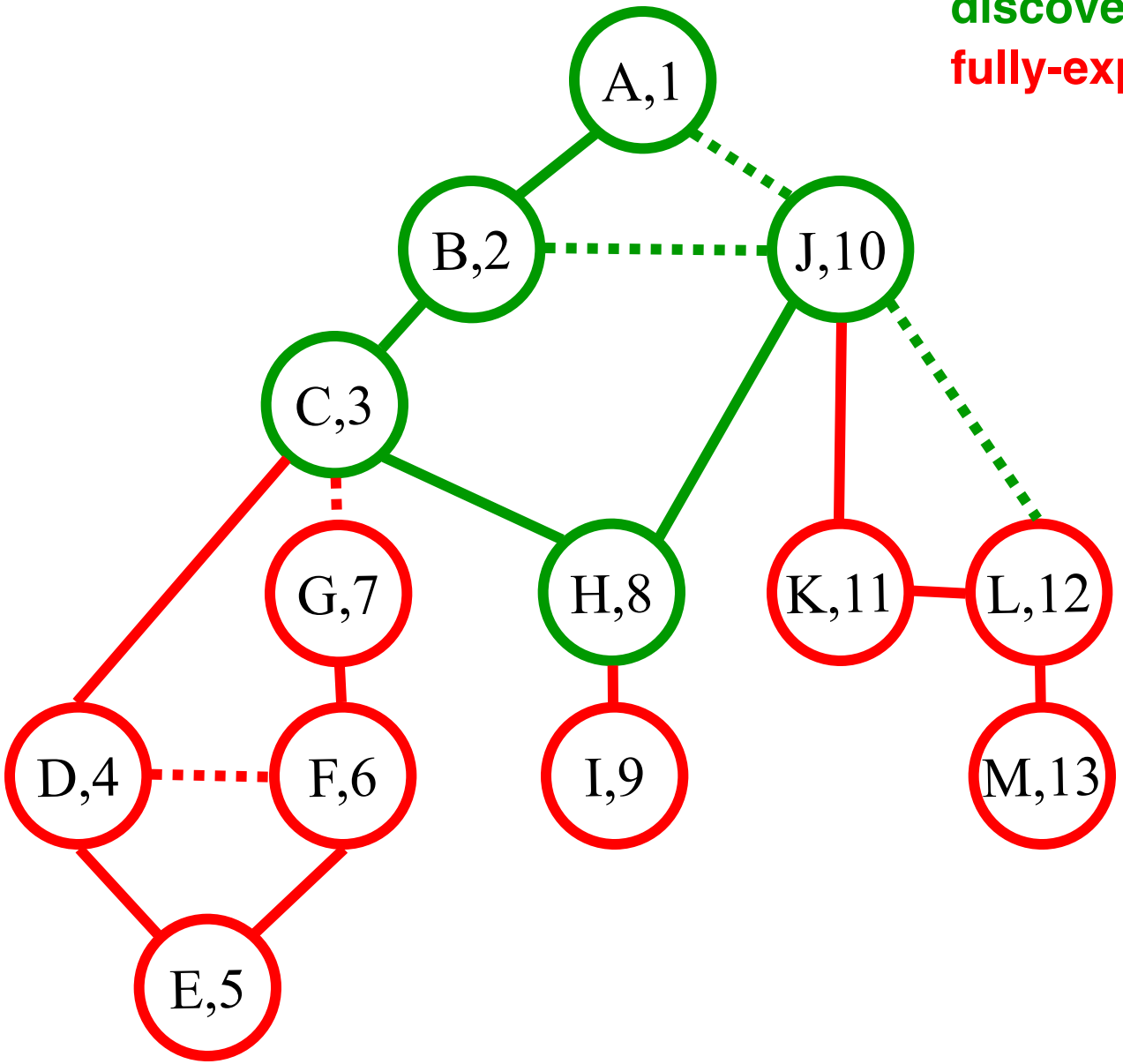
DFS(A)

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}

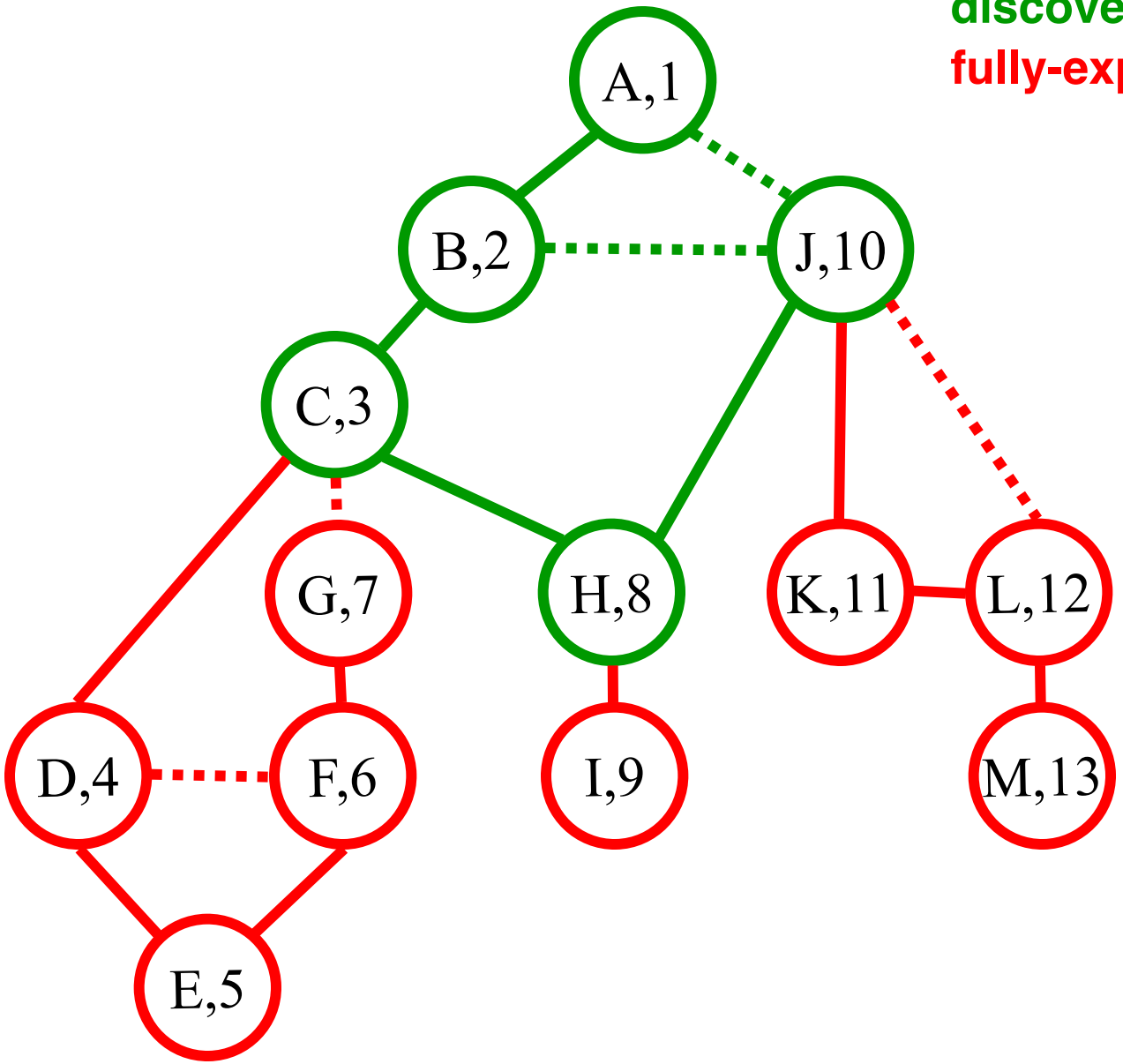
DFS(A)

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}

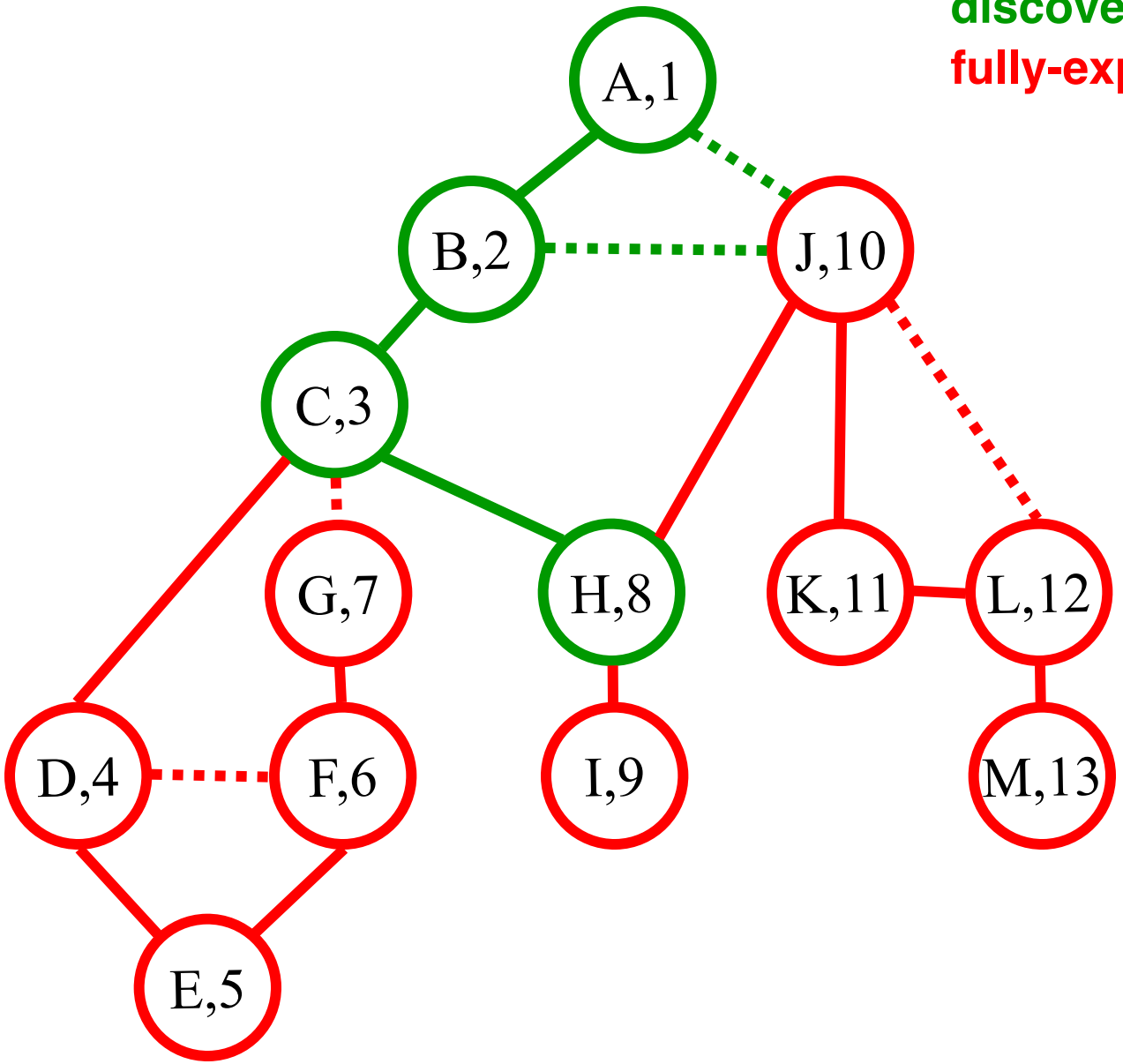
DFS(A)

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}

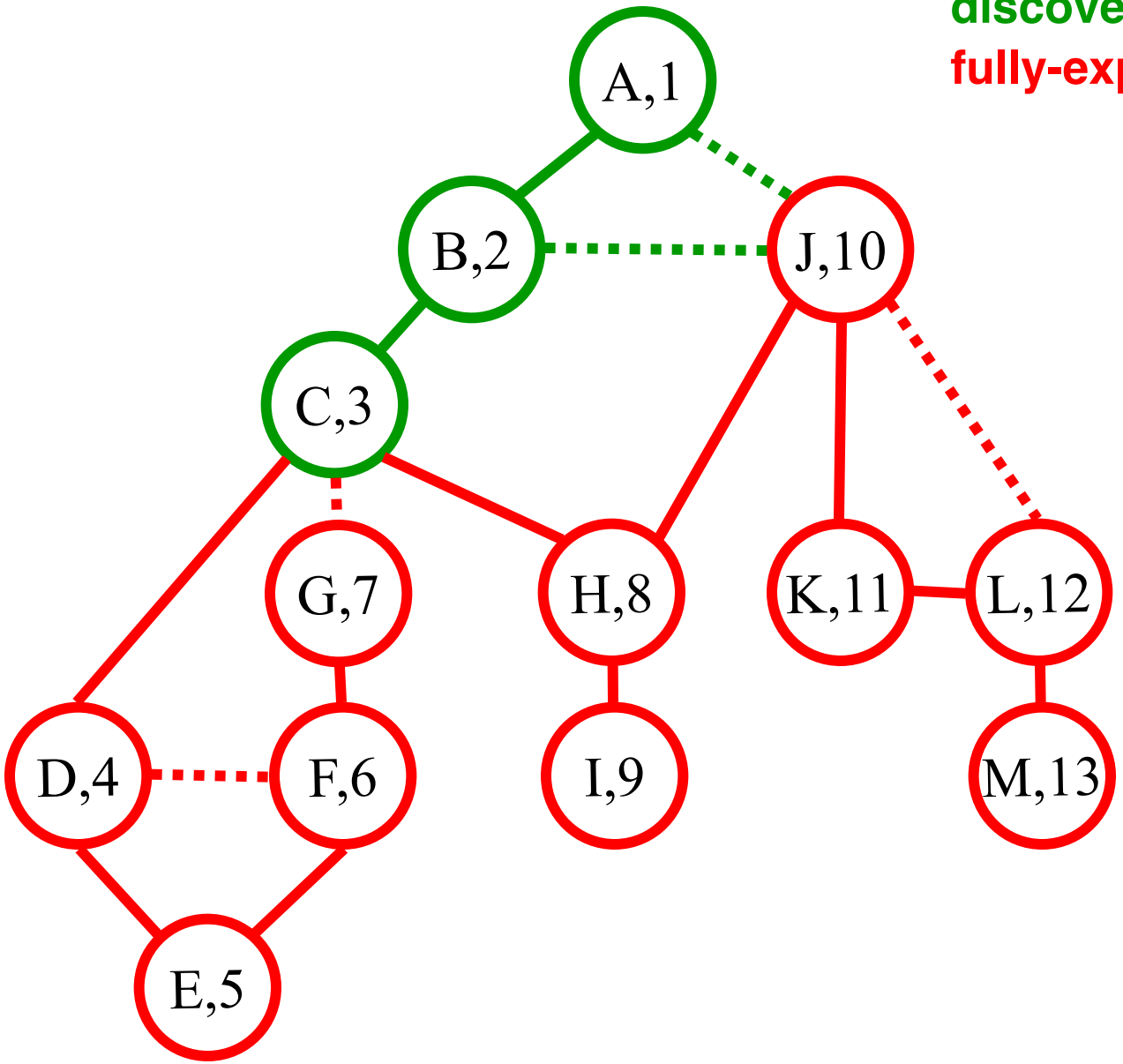
DFS(A)

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}

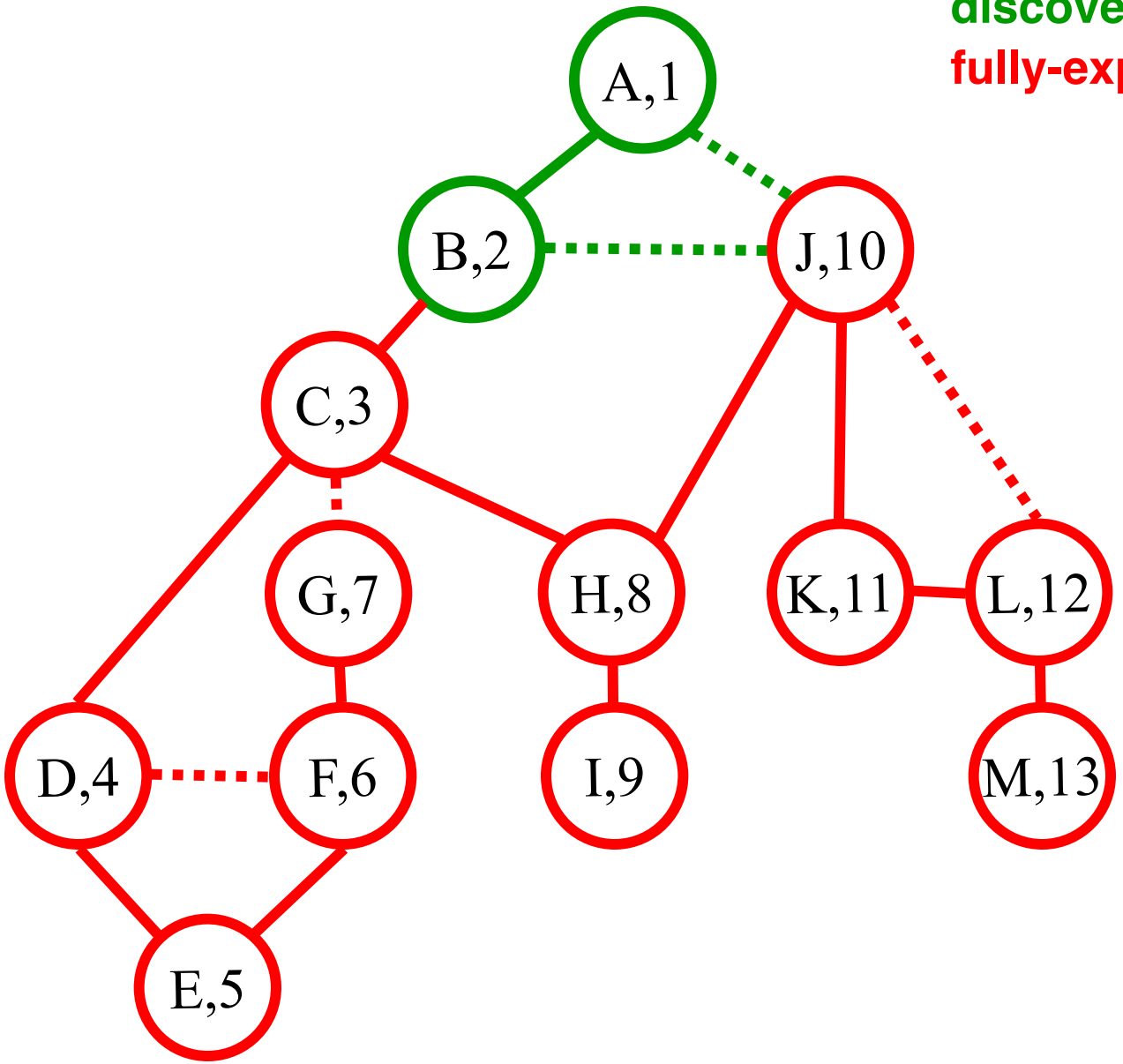
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

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

st[] =
{1,2}

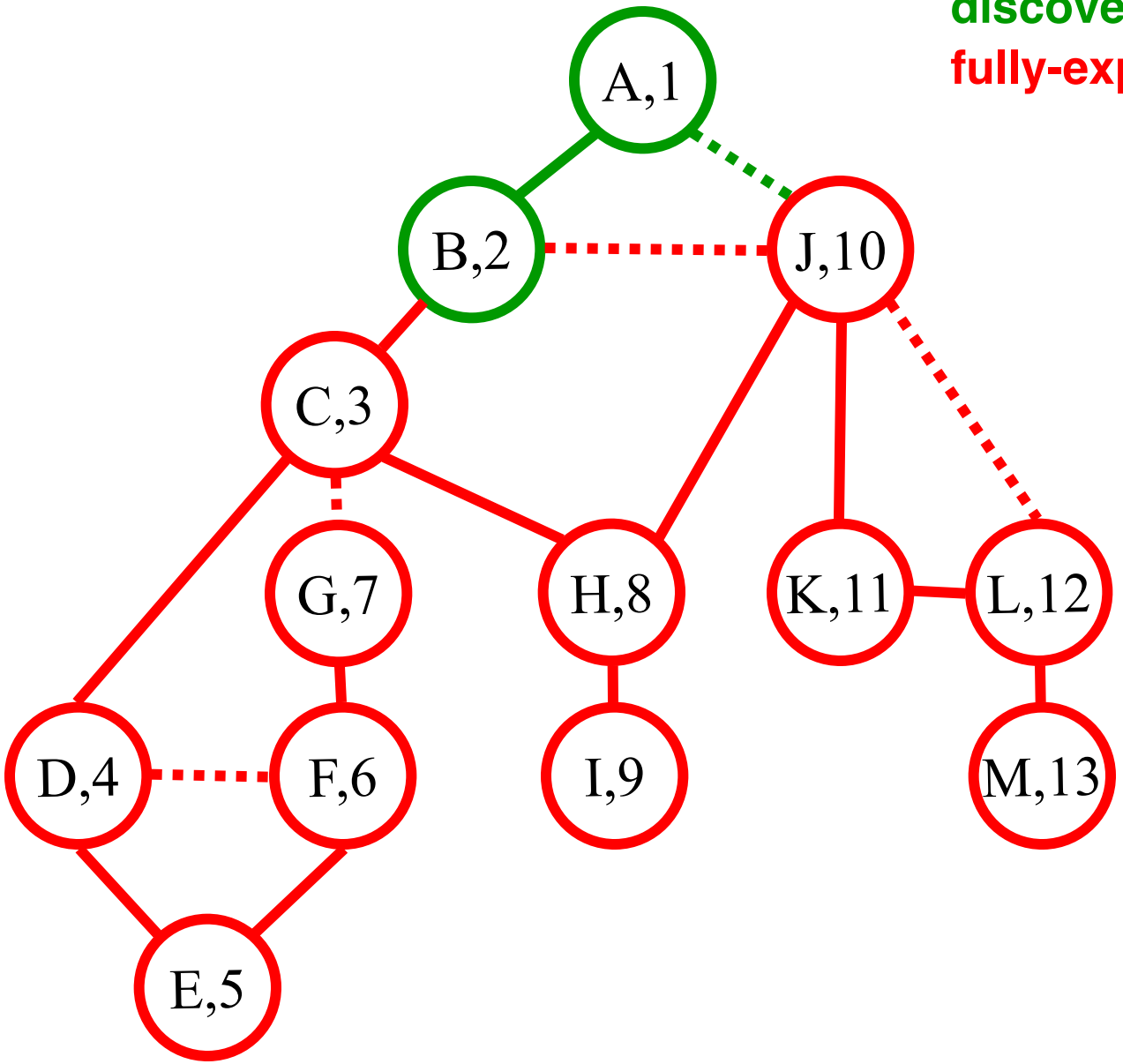
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

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

st[] =
{1,2}

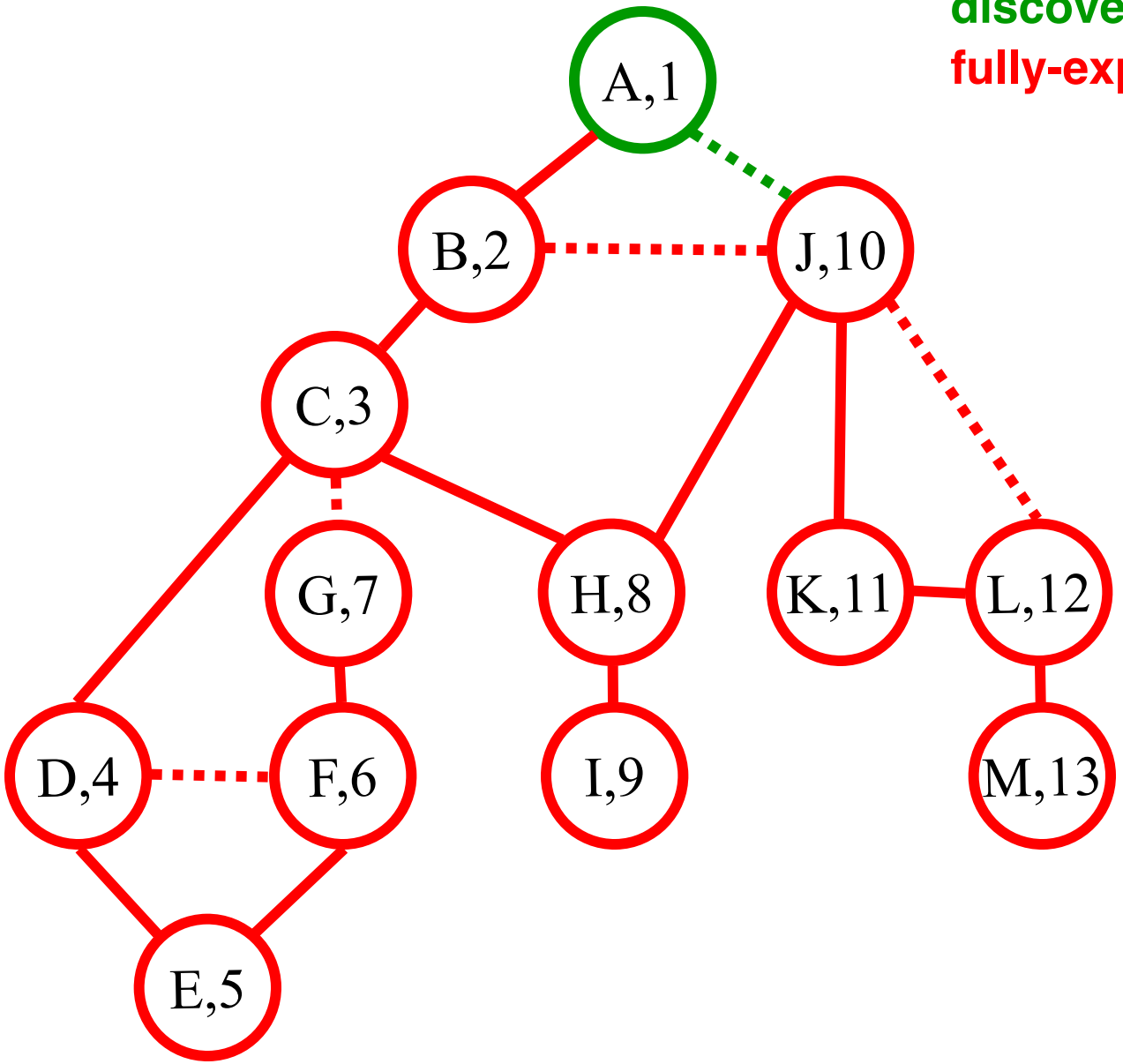
DFS(A)

Color code:

undiscovered

discovered

fully-explored



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

st[] = {1}

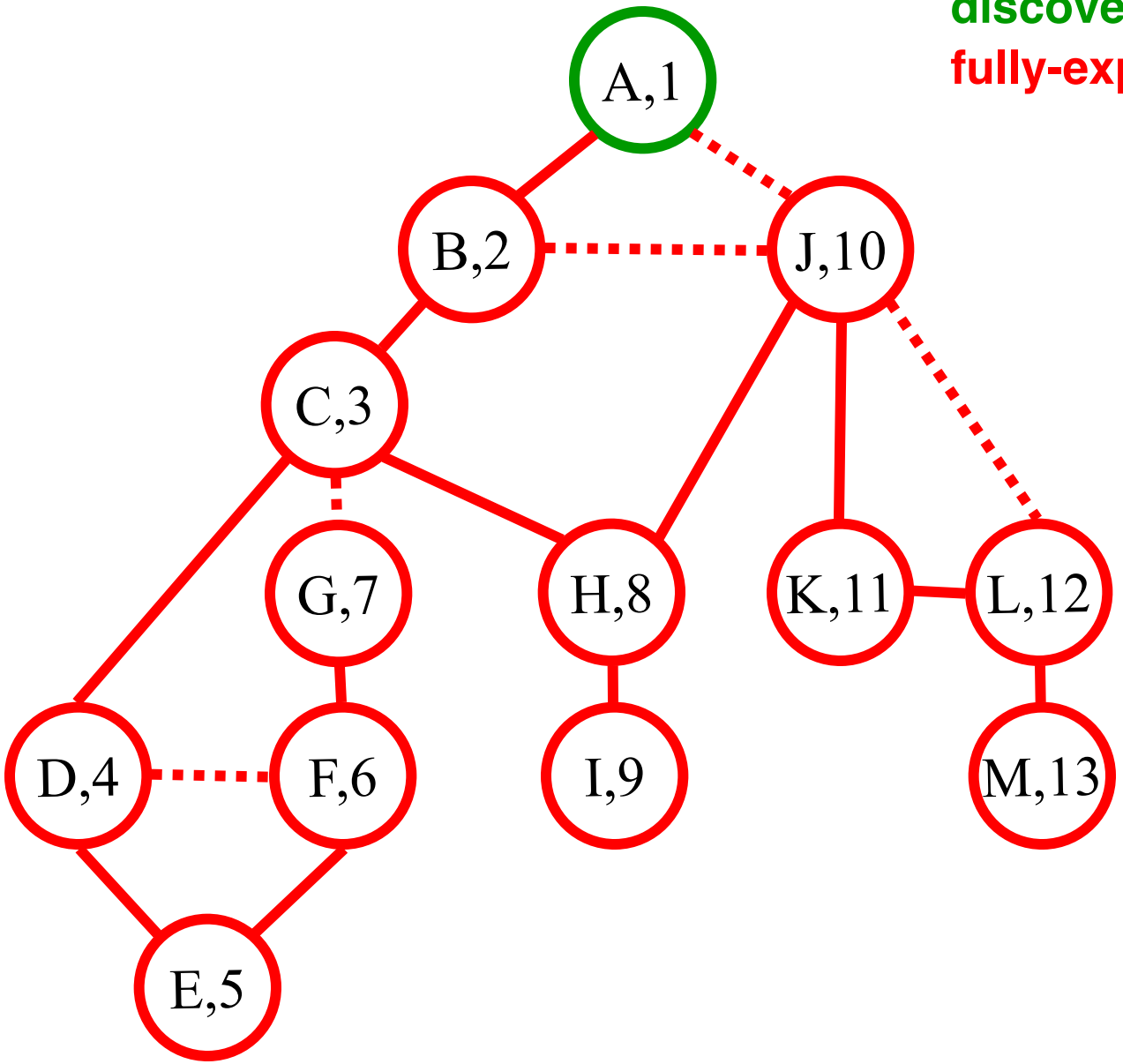
DFS(A)

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

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

st[] =
{1}

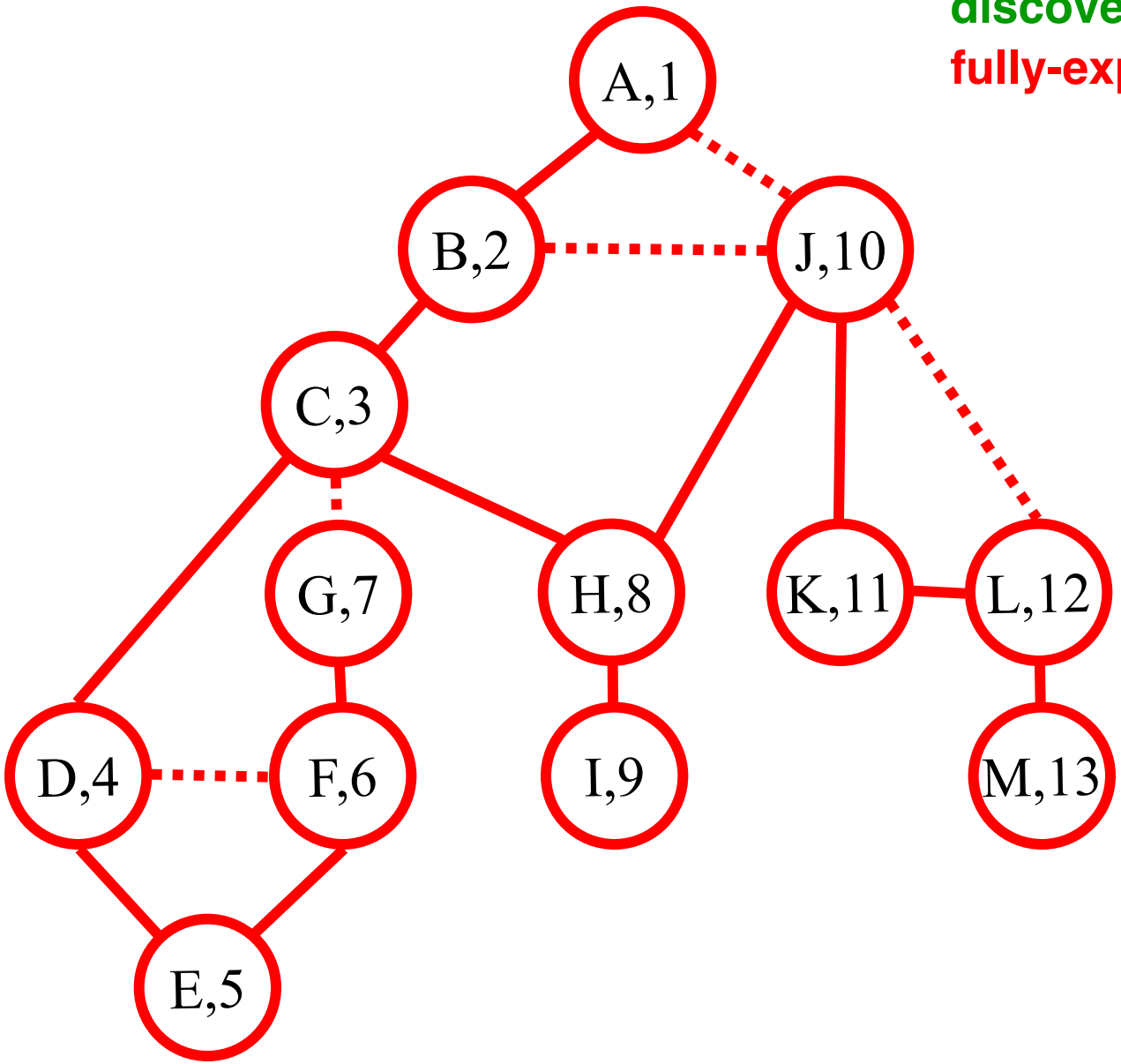
DFS(A)

Color code:

undiscovered

discovered

fully-explored

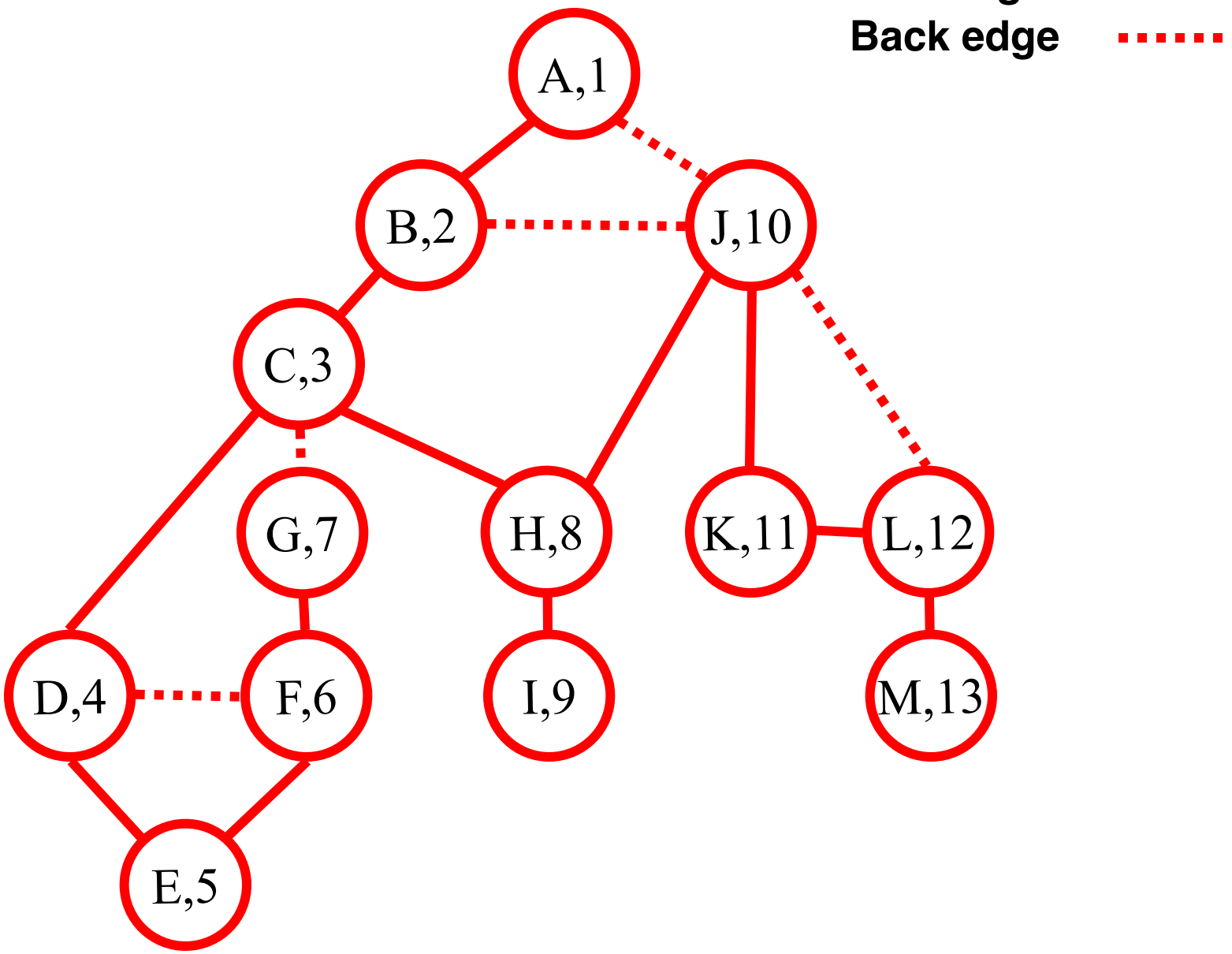


Call Stack:
(Edge list)

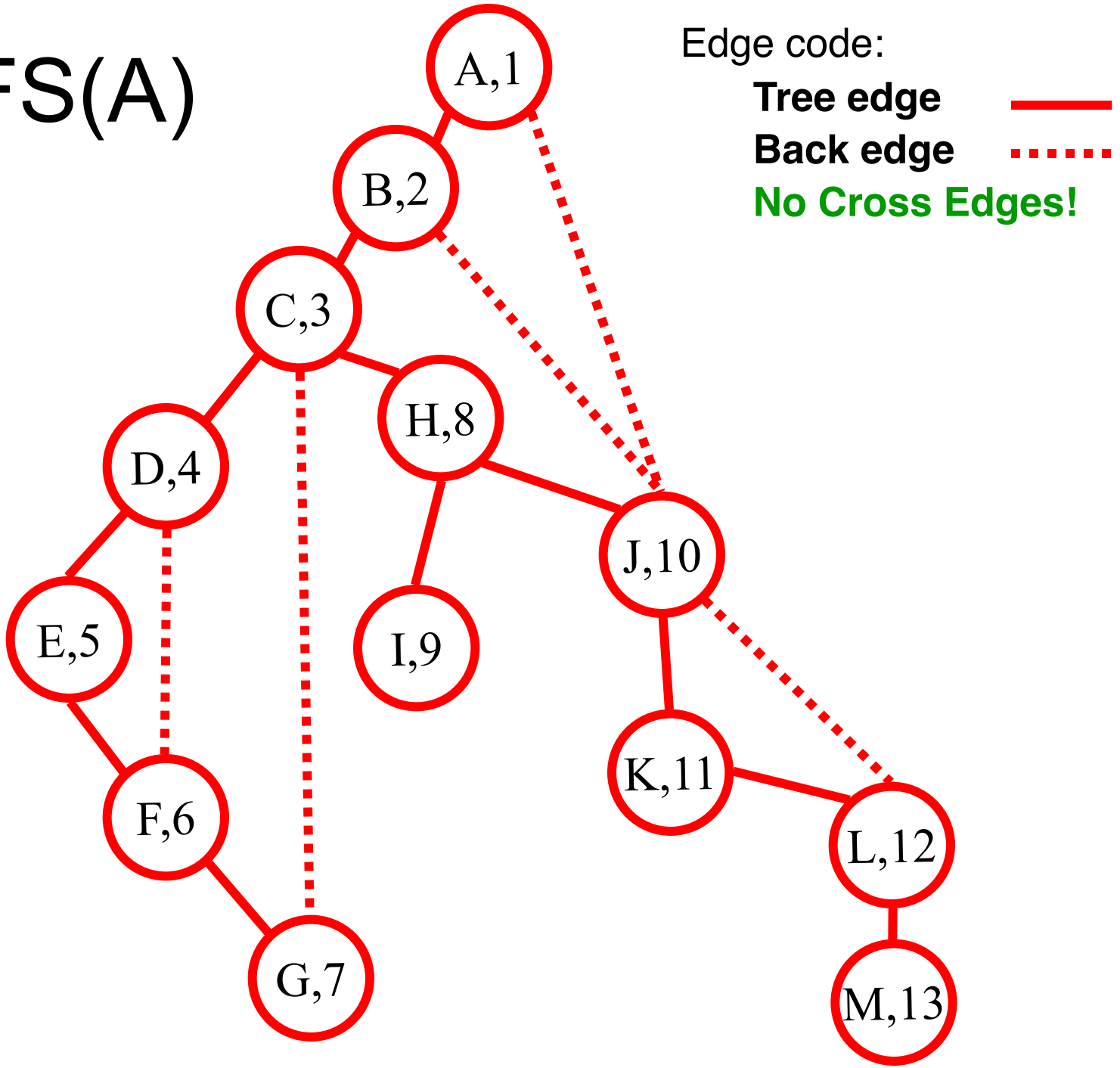
TA-DA!!

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 "DFS tree" of G

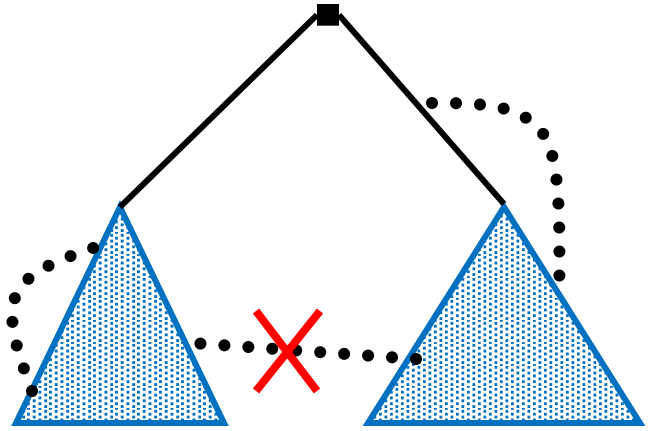
Unlike the BFS tree:

- The DFS 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

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 descendents/ancestors in the DFS tree



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.