CSE 521: Algorithms

Graphs and Graph Algorithms

Larry Ruzzo

Graphs

An extremely important formalism for representing (binary) relationships

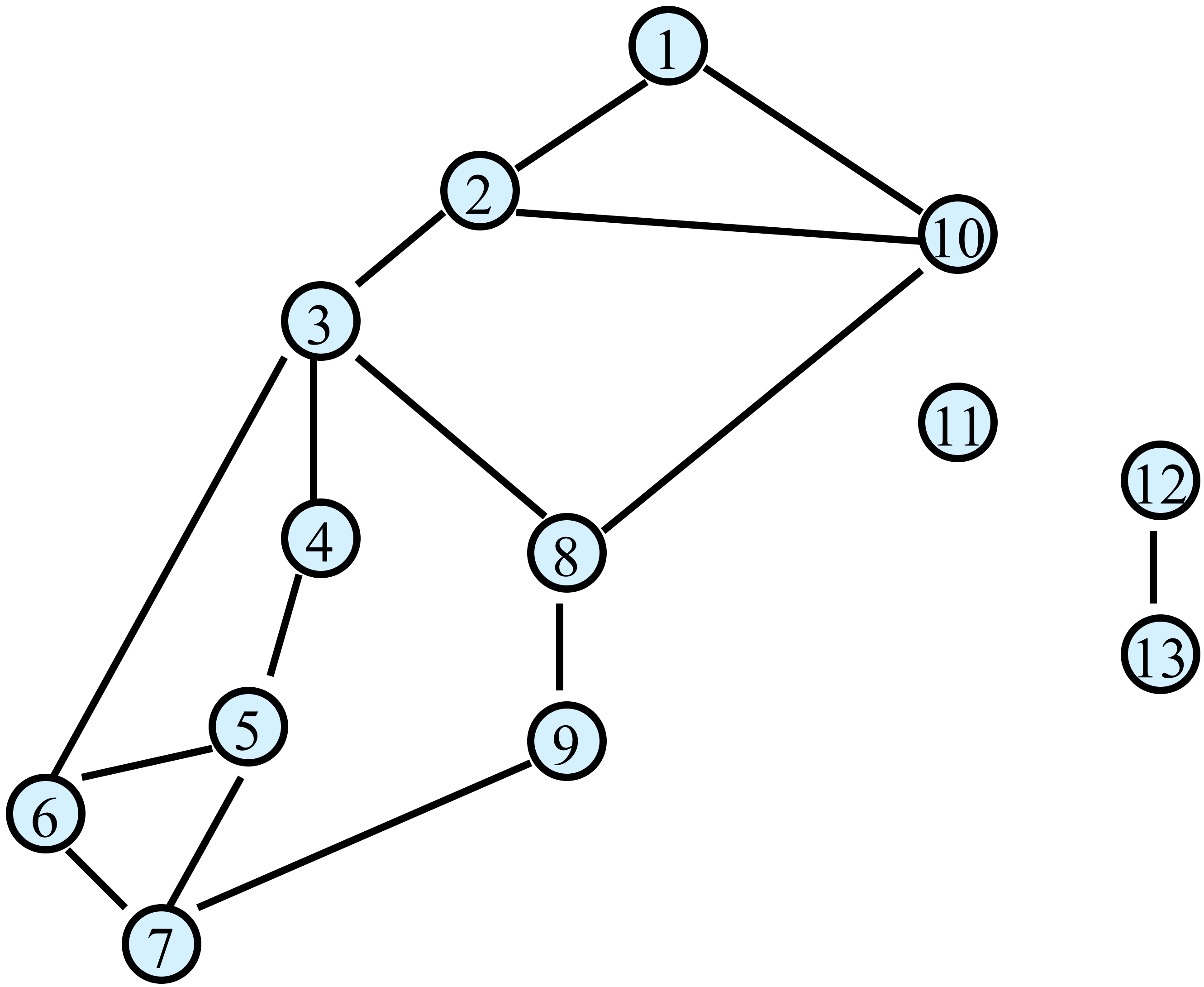
Objects: "vertices," aka "nodes"

Relationships between pairs:

"edges," aka "arcs"

Formally, a graph G = (V, E) is a pair of sets, V the vertices and E the edges

Undirected Graph   G = (V,E)



Graph Traversal

Learn the basic structure of a graph

"Walk," *via edges*, from a fixed starting vertex *s* to all vertices reachable from *s*

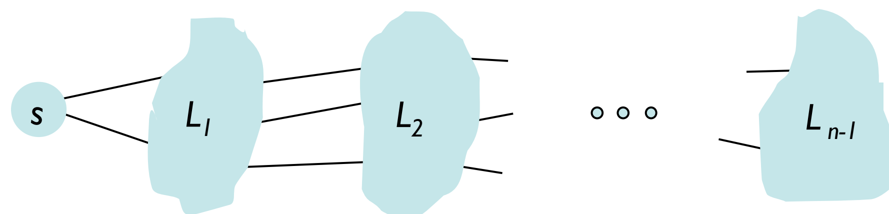
Being *orderly* helps. Two common ways:

- Breadth-First Search
- Depth-First Search

Breadth-First Search

Idea: Explore from s in all possible directions, layer by layer.

BFS algorithm.



$L_0 = \{ s \}$.
L_1 = all neighbors of L_0.
L_2 = all nodes not in L_0 or L_1, and having an edge to a node in L_1.
L_{i+1} = all nodes not in earlier layers, and having an edge to a node in L_i.

Theorem. For each i, L_i consists of all nodes at distance (i.e., min path length) exactly i from s.

Cor: There is a path from s to t iff t appears in some layer.

Properties of (Undirected) BFS(v)

BFS(v) visits x if and only if there is a path in G from v to x.

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

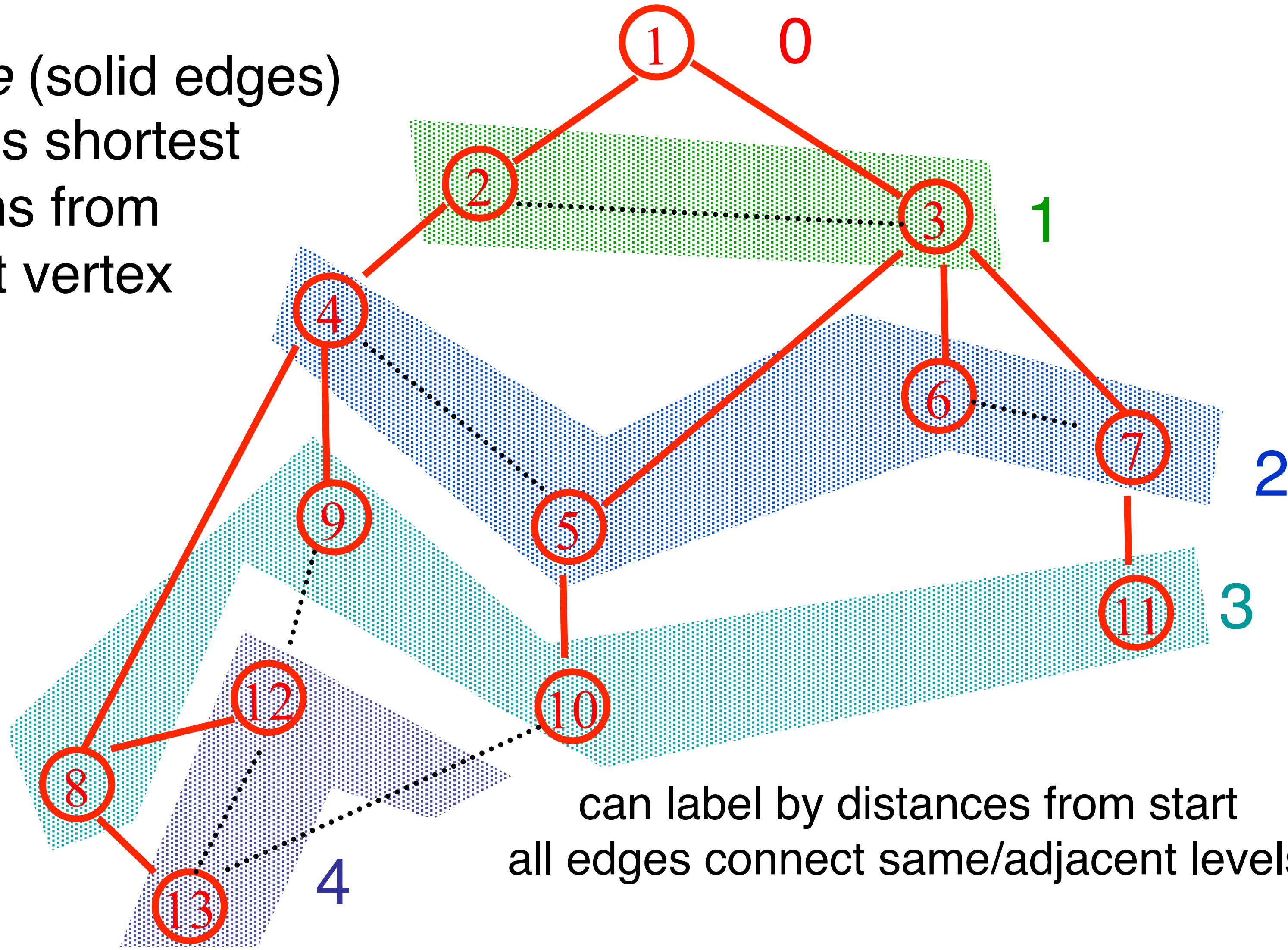
Level i in this tree are exactly those vertices *u* such that the shortest path (in G, not just the tree) from the root v is of length i.

All non-tree edges join vertices on the same or adjacent levels

not true of every spanning tree!

BFS Application: Shortest Paths

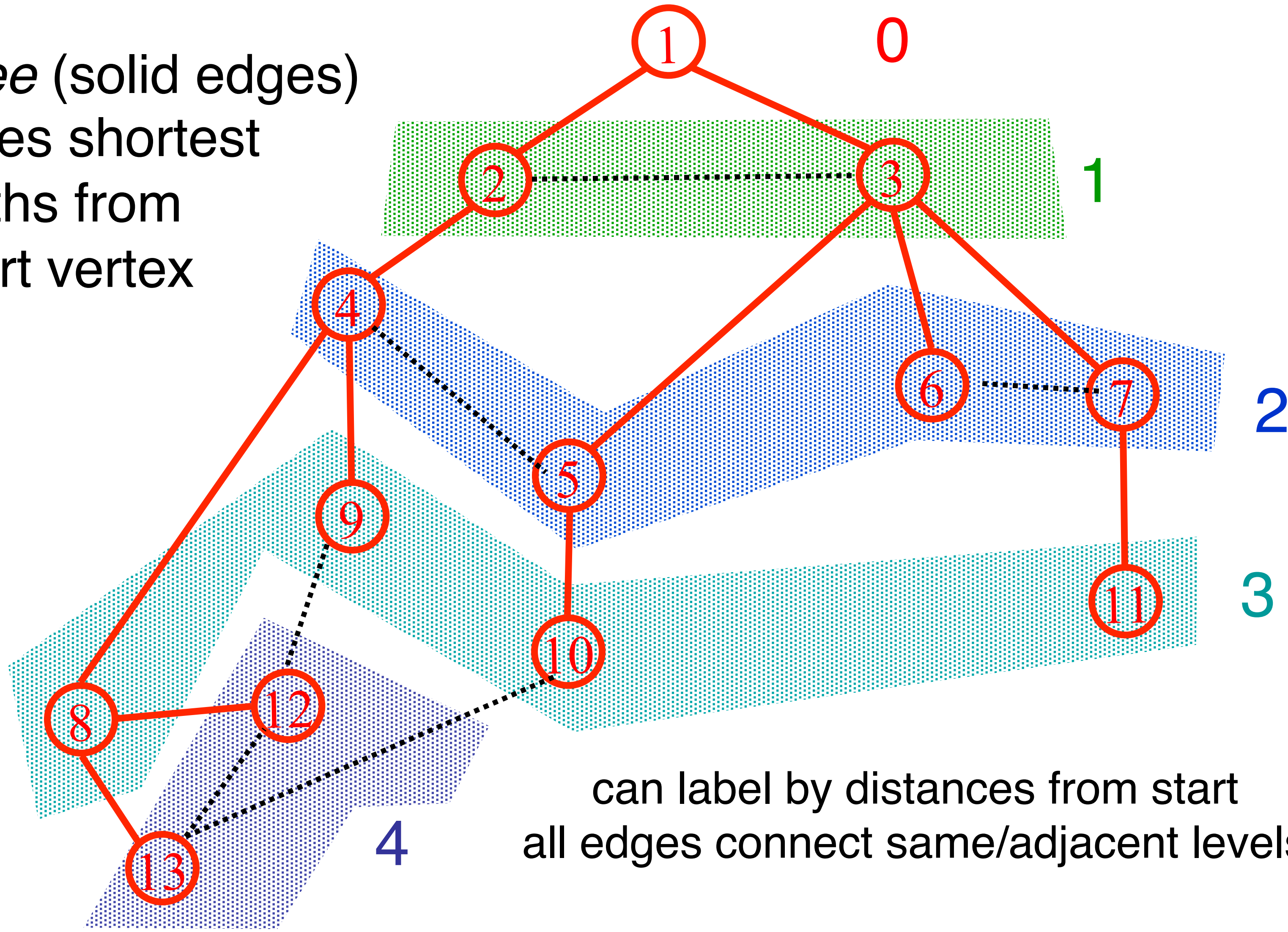
Tree (solid edges) gives shortest paths from start vertex



can label by distances from start
all edges connect same/adjacent levels

BFS Application: Shortest Paths

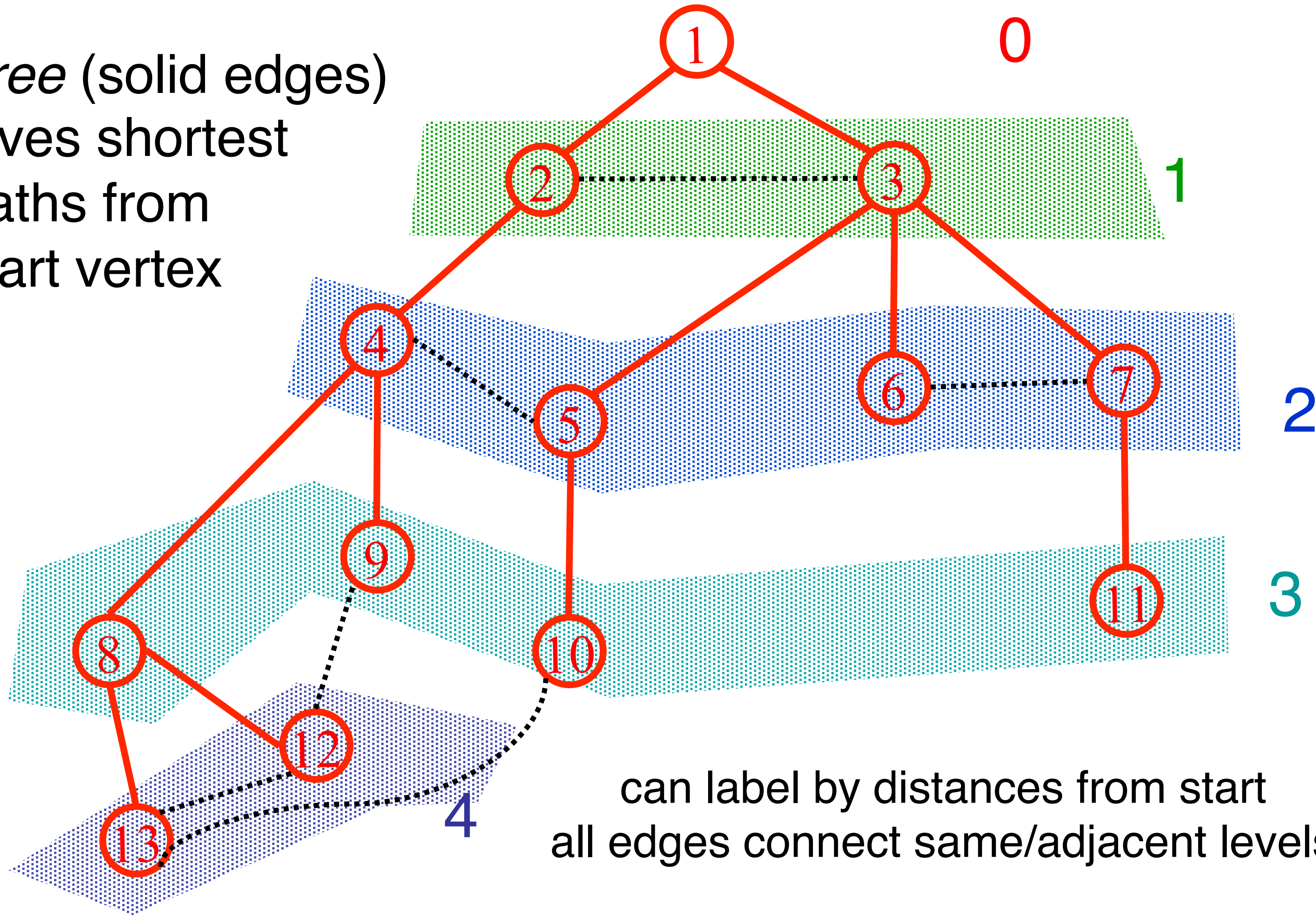
Tree (solid edges) gives shortest paths from start vertex



can label by distances from start
all edges connect same/adjacent levels

BFS Application: Shortest Paths

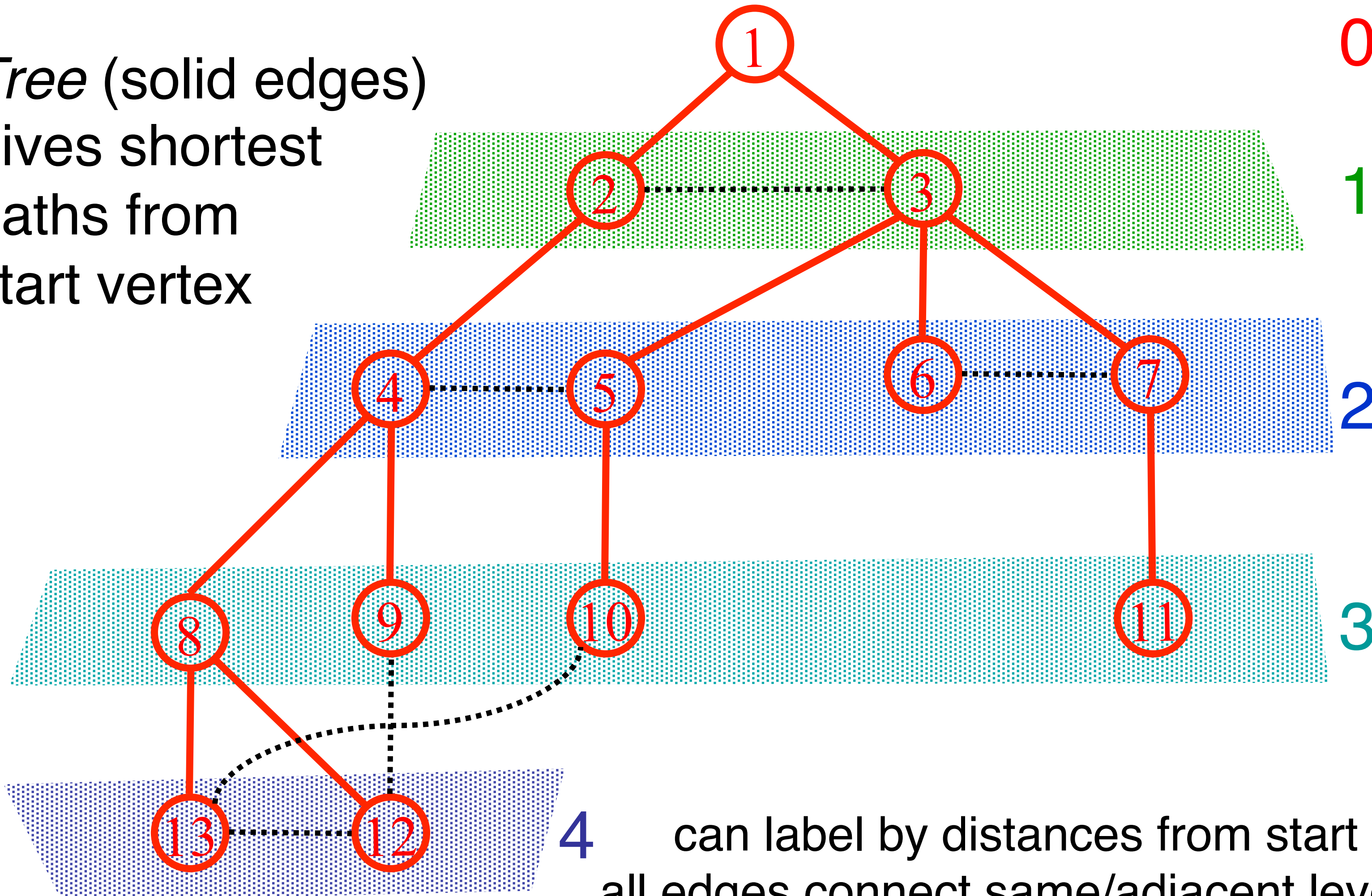
Tree (solid edges) gives shortest paths from start vertex



can label by distances from start
all edges connect same/adjacent levels

BFS Application: Shortest Paths

Tree (solid edges) gives shortest paths from start vertex



can label by distances from start
all edges connect same/adjacent levels

Why fuss about trees?

Trees are simpler than graphs

Ditto for algorithms on trees vs algs on graphs

So, 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

E.g., BFS finds a tree s.t. level-jumps are minimized

DFS (below) finds a different tree, but it also has interesting structure…

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(v) – Recursive version

Global Initialization:

```
    for all nodes v, v.dfs# = -1   // mark v "undiscovered"
    dfscounter = 0

DFS(v)
    v.dfs# = dfscounter++          // v "discovered", number it
    for each edge (v,x)
        if (x.dfs# = -1)           // tree edge (x previously  undiscovered)
            DFS(x)
        else …                     // code for back-, fwd-, parent-
                                   // edges, if needed; mark v
                                   // "completed," if needed
```

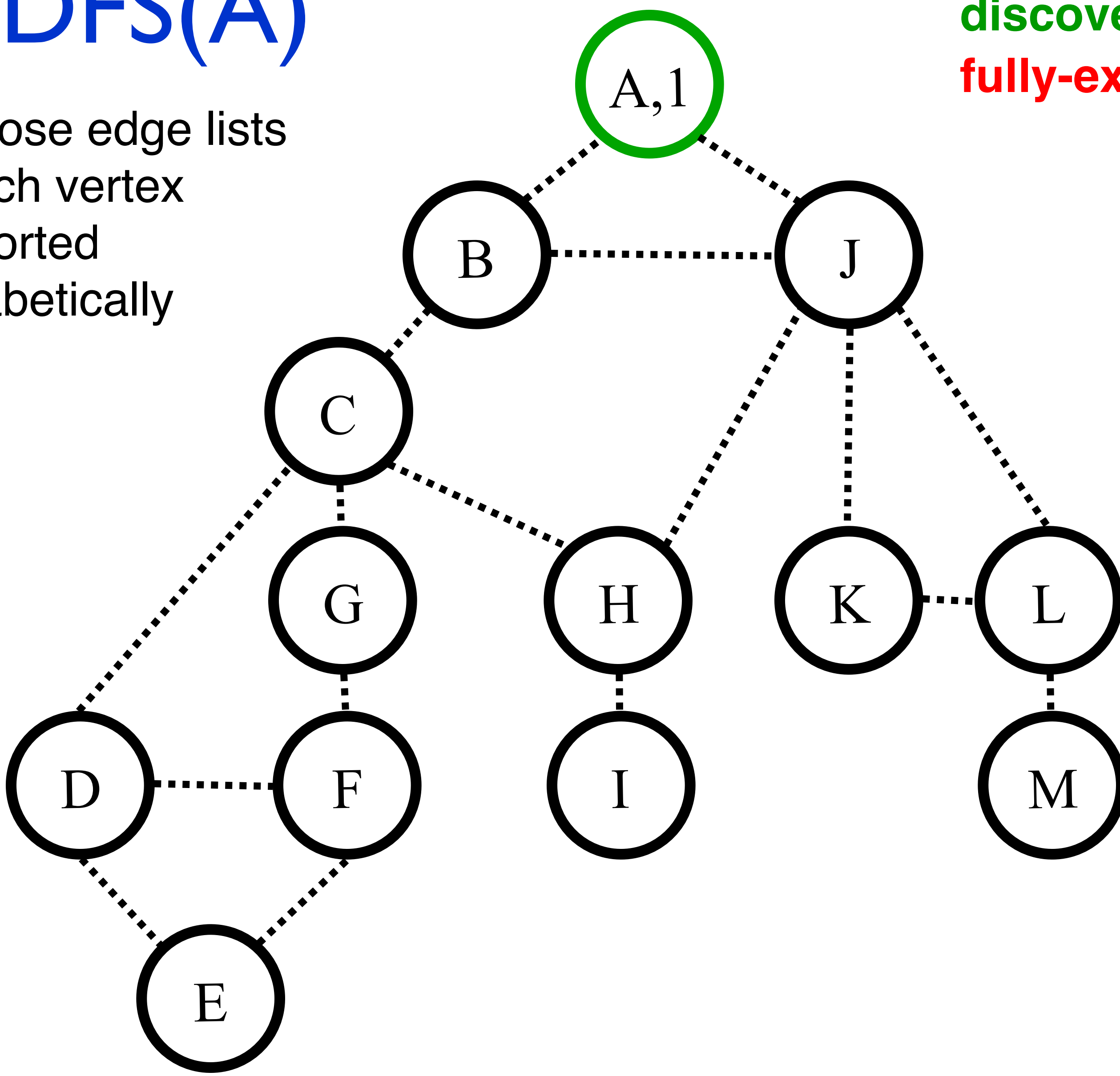
Why fuss about trees (again)?

BFS tree $\neq$ DFS tree, but, as with BFS, DFS has found a tree in the graph s.t. non-tree edges are "simple" – *only descendant/ancestor*

Proof below

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



Color code:

undiscovered

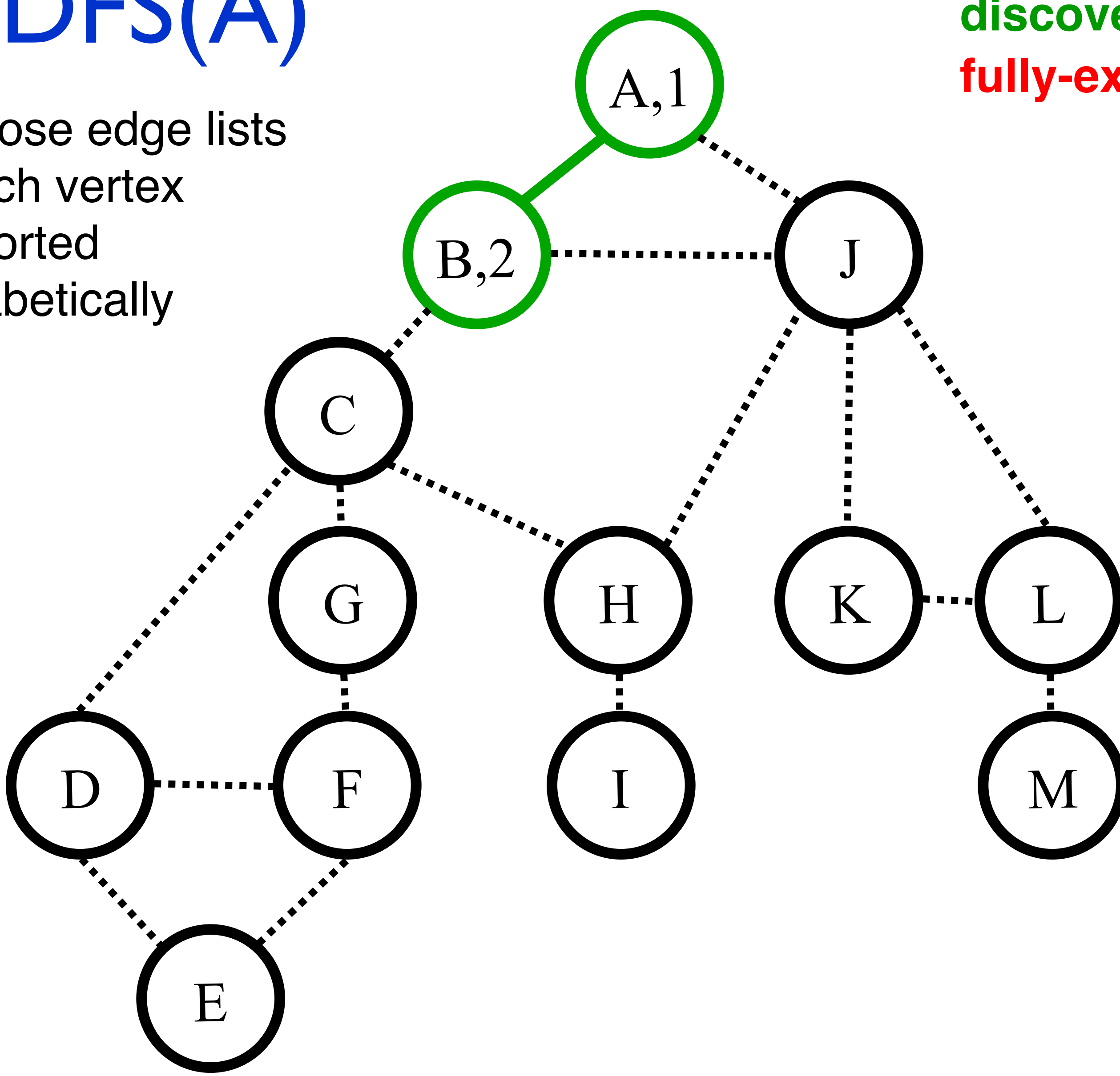
discovered

fully-explored

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

DFS(A)

Suppose edge lists
at each vertex
are sorted
alphabetically

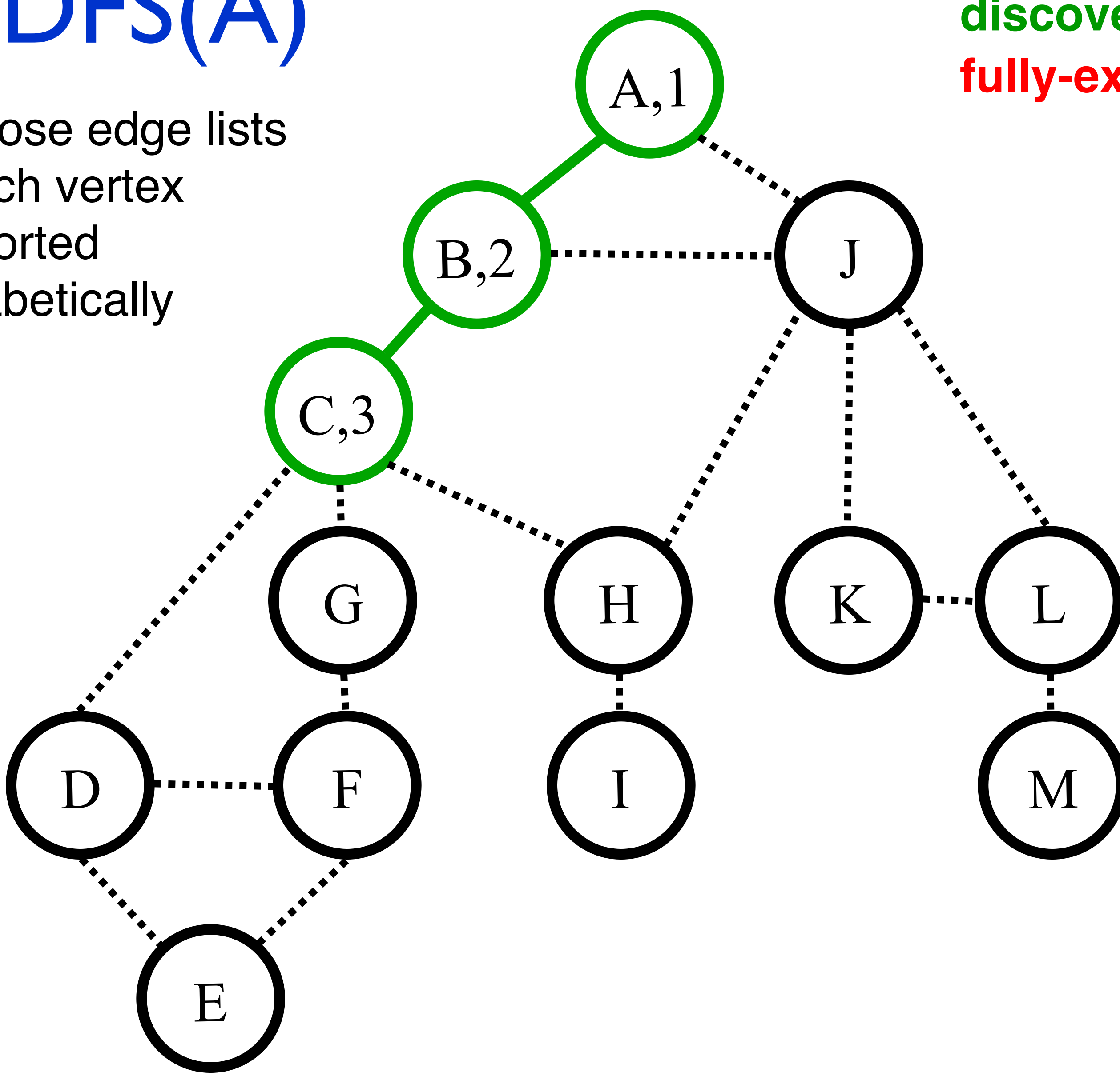


Color code:
undiscovered
discovered
fully-explored

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

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



Color code:

undiscovered

discovered

fully-explored

Call Stack:
(Edge list)

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

DFS(A)

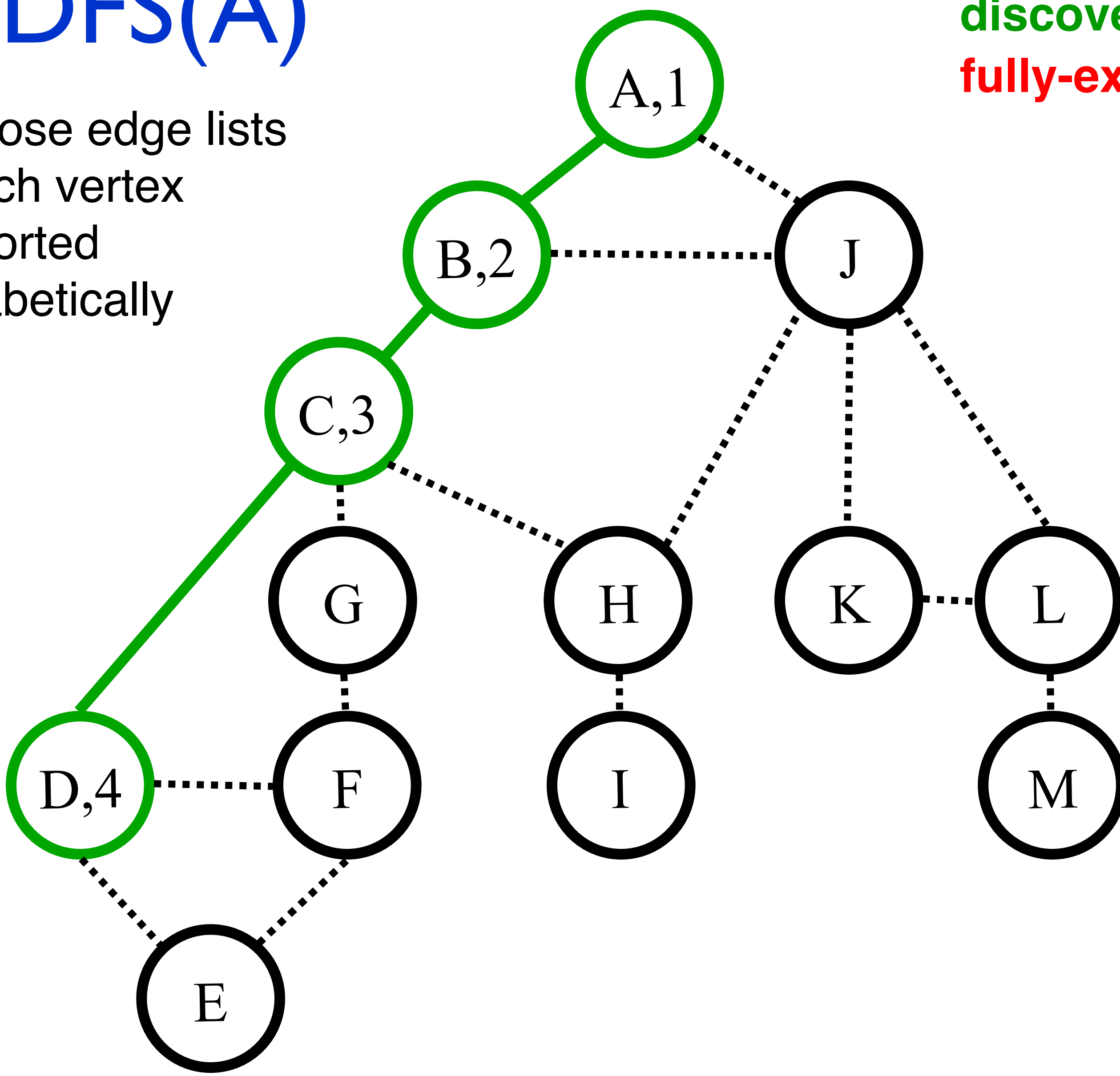
Suppose edge lists at each vertex are sorted alphabetically

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)

DFS(A)

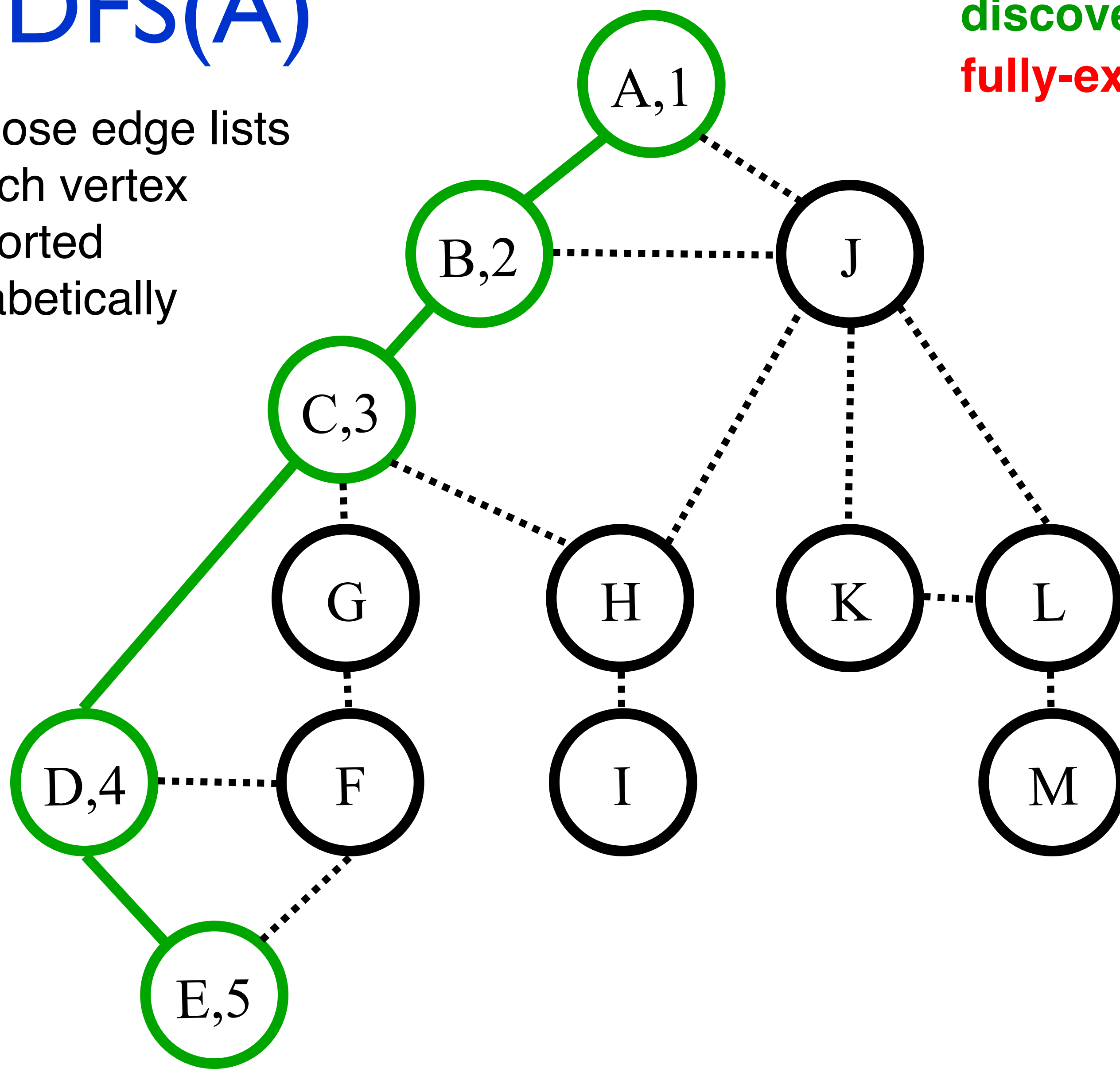
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored



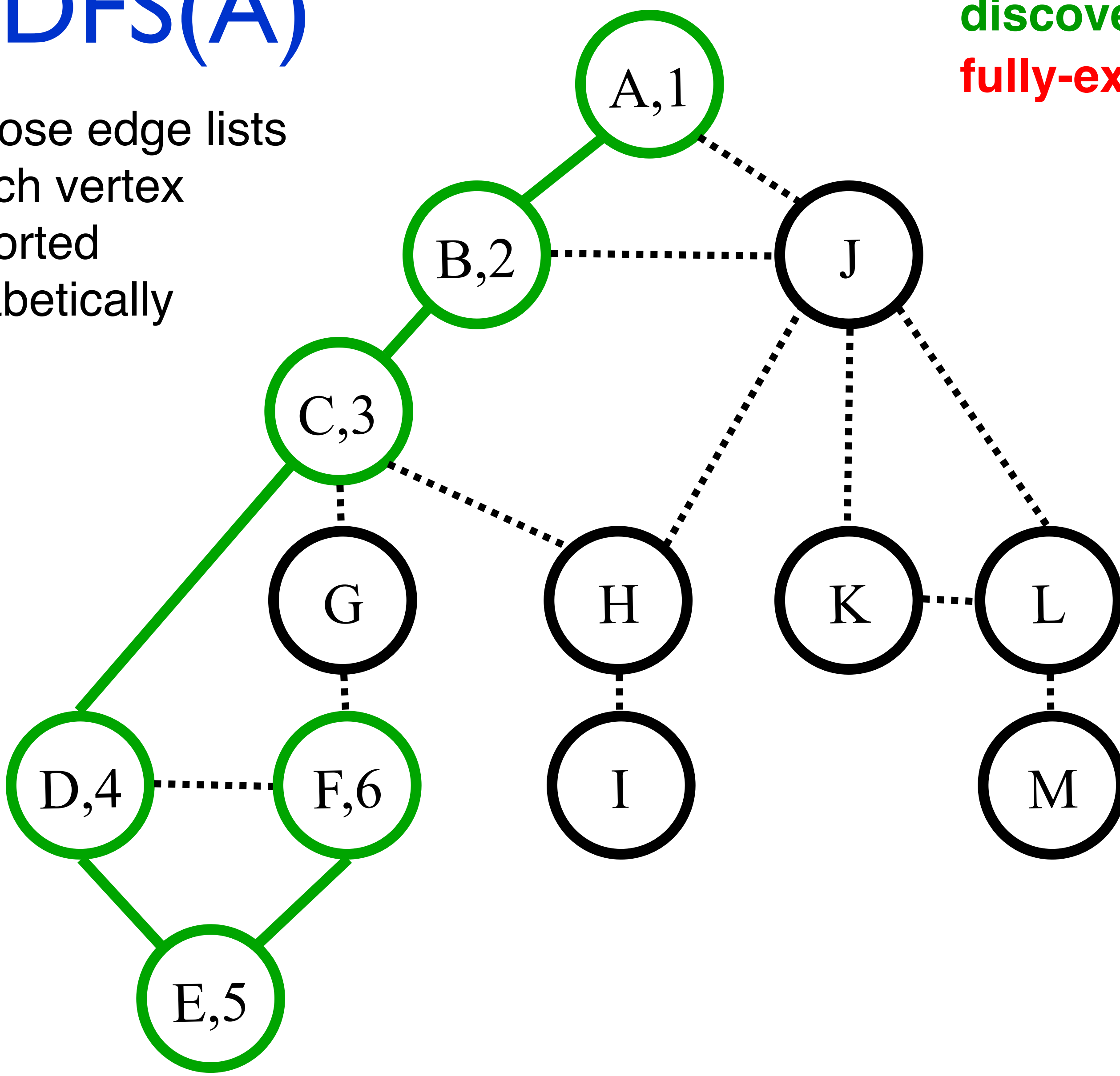
Call Stack:
(Edge list)

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

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically

Color code:
undiscovered
discovered
fully-explored

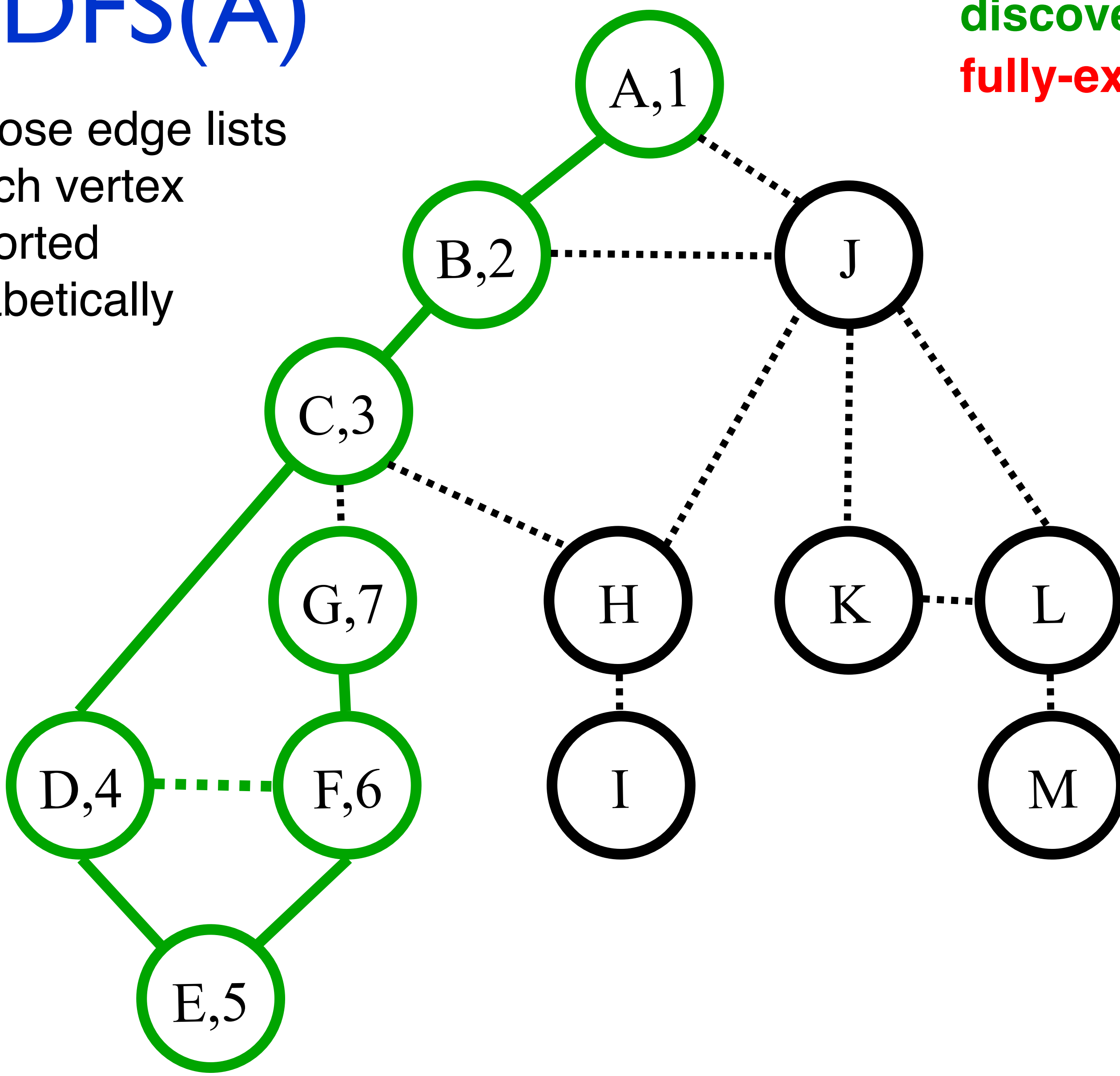


Call Stack:
(Edge list)

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

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically

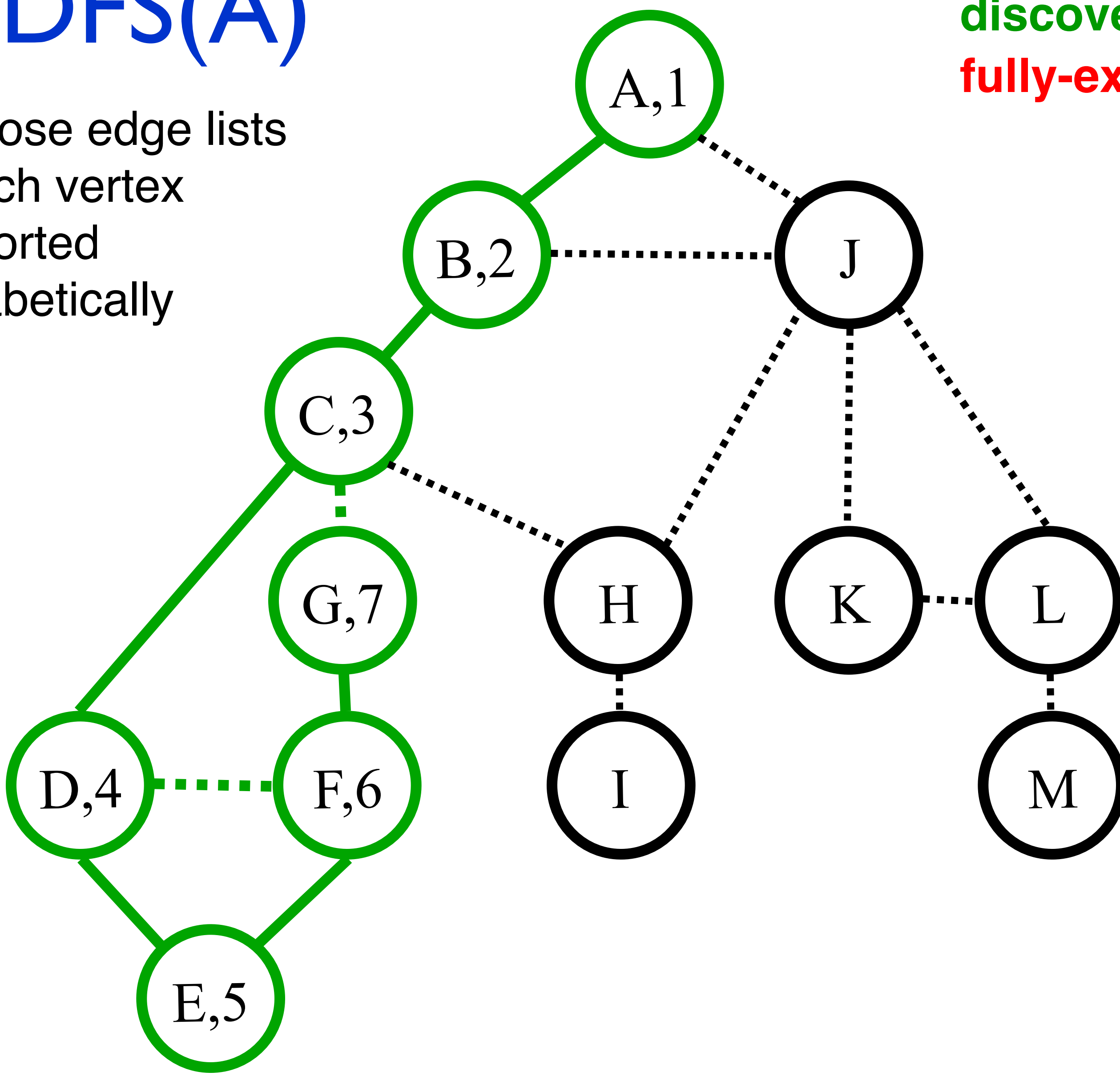


Color code:
undiscovered
discovered
fully-explored

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)

DFS(A)

Suppose edge lists
at each vertex
are sorted
alphabetically



Color code:

undiscovered

discovered

fully-explored

Call Stack:
(Edge list)

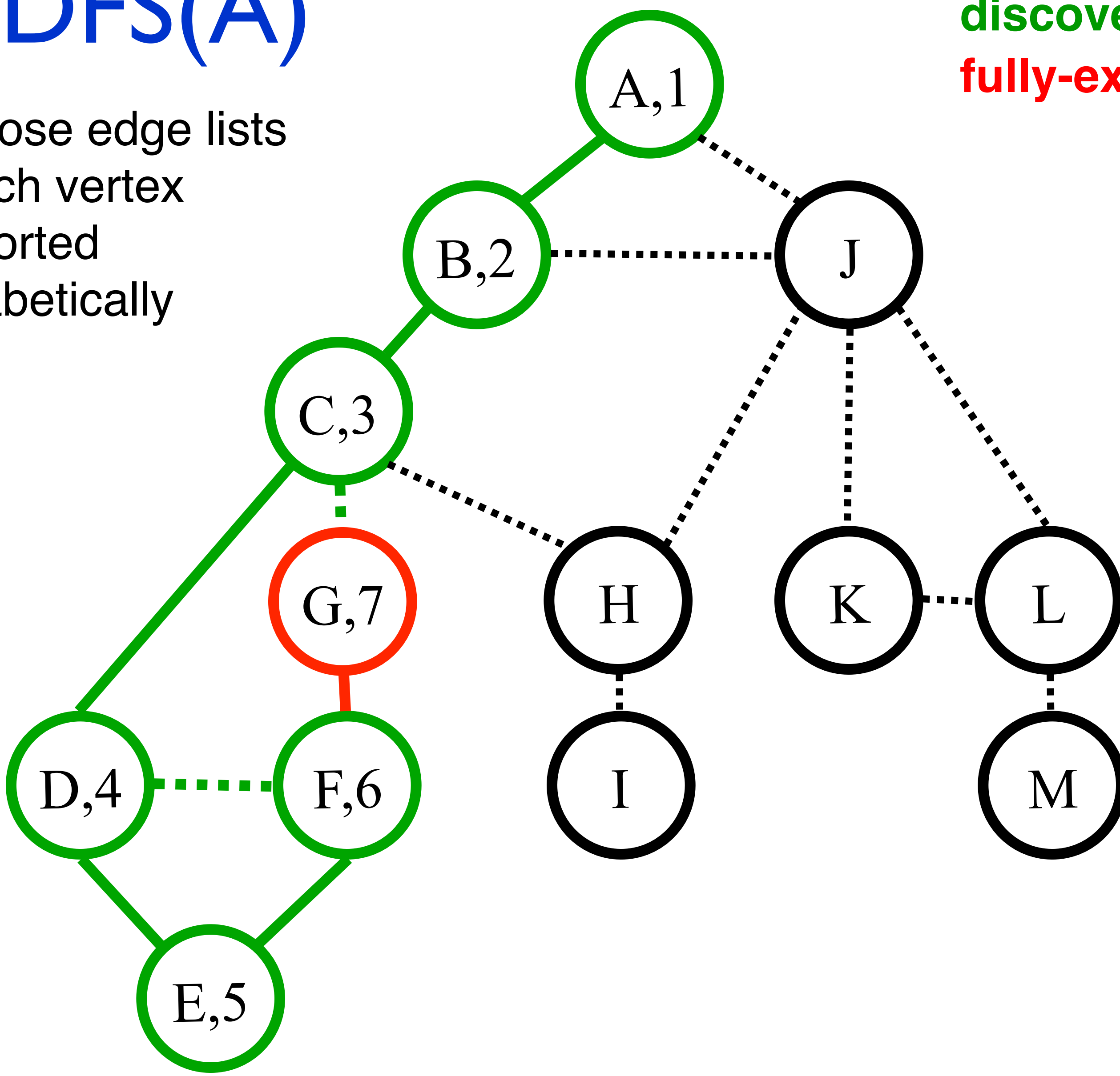
A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,G,H)
D (~~C~~,~~E~~,F)
E (~~D~~,~~F~~)
F (~~D~~,~~E~~,~~G~~)
G (~~C~~,~~F~~)

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically

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~~)

DFS(A)

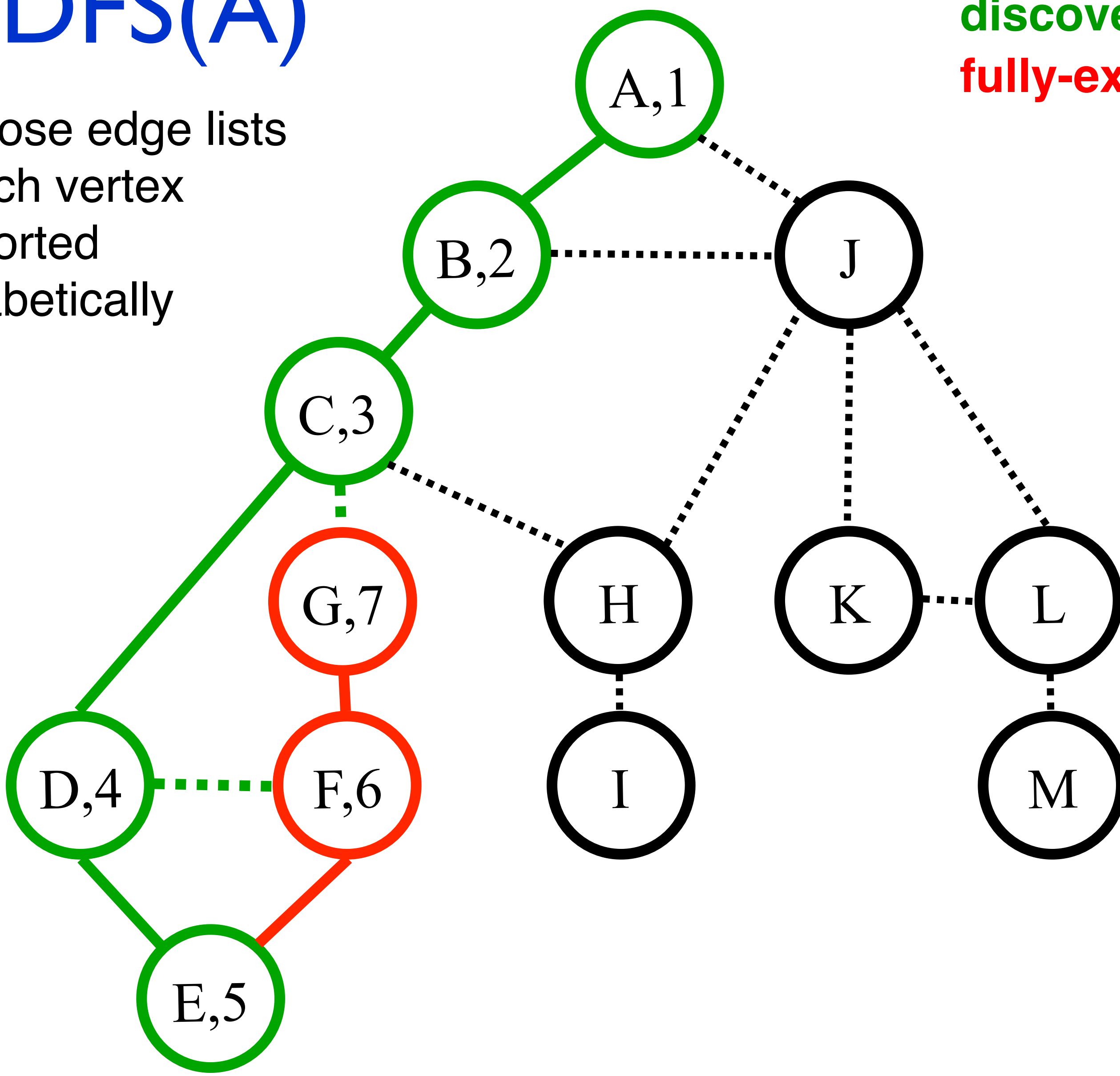
Color code:

undiscovered

discovered

fully-explored

Suppose edge lists at each vertex are sorted alphabetically



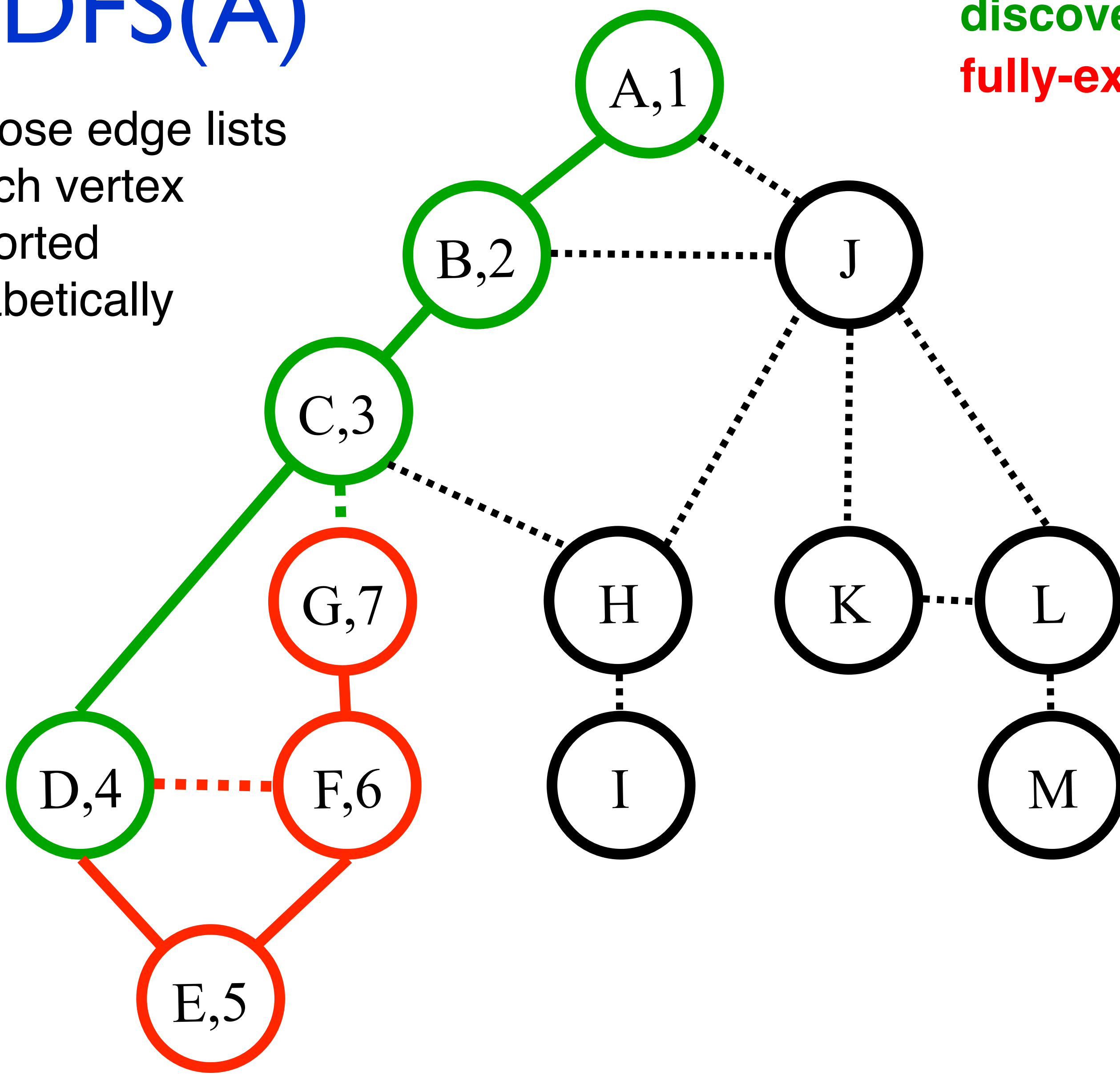
Call Stack:
(Edge list)

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

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically

Color code:
undiscovered
discovered
fully-explored



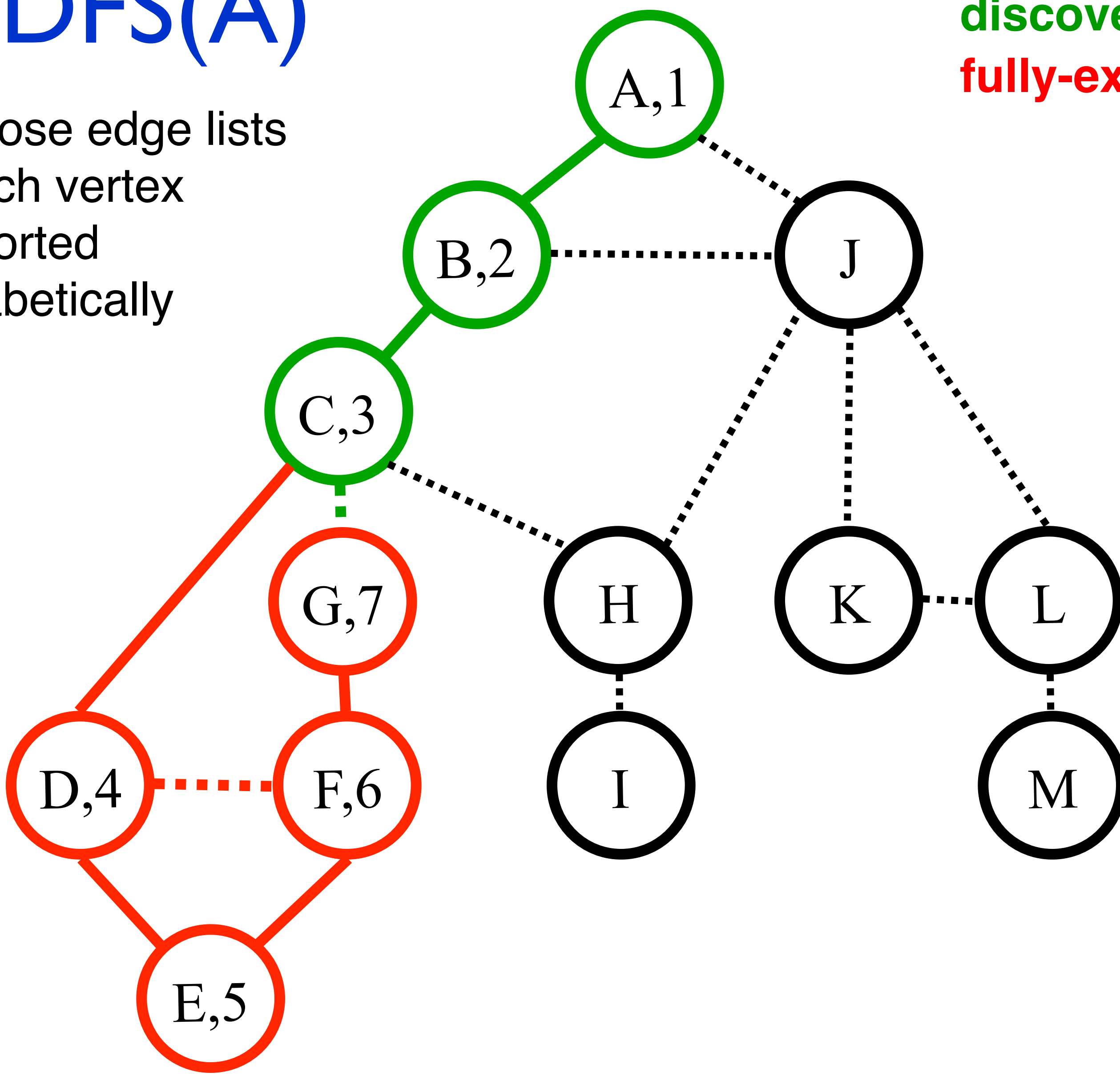
Call Stack:
(Edge list)

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

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically

Color code:
undiscovered
discovered
fully-explored



Call Stack:
(Edge list)

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

DFS(A)

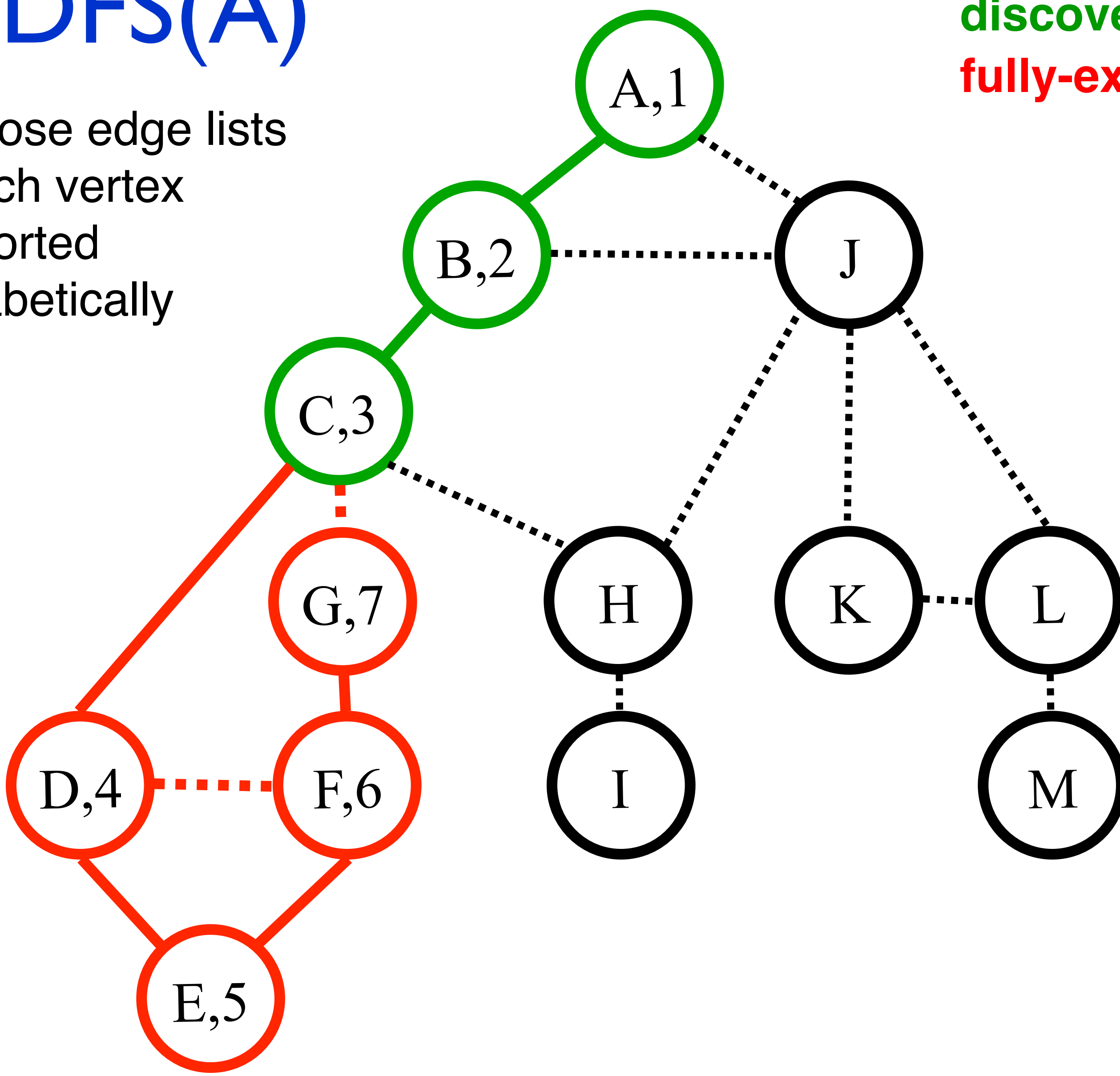
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored

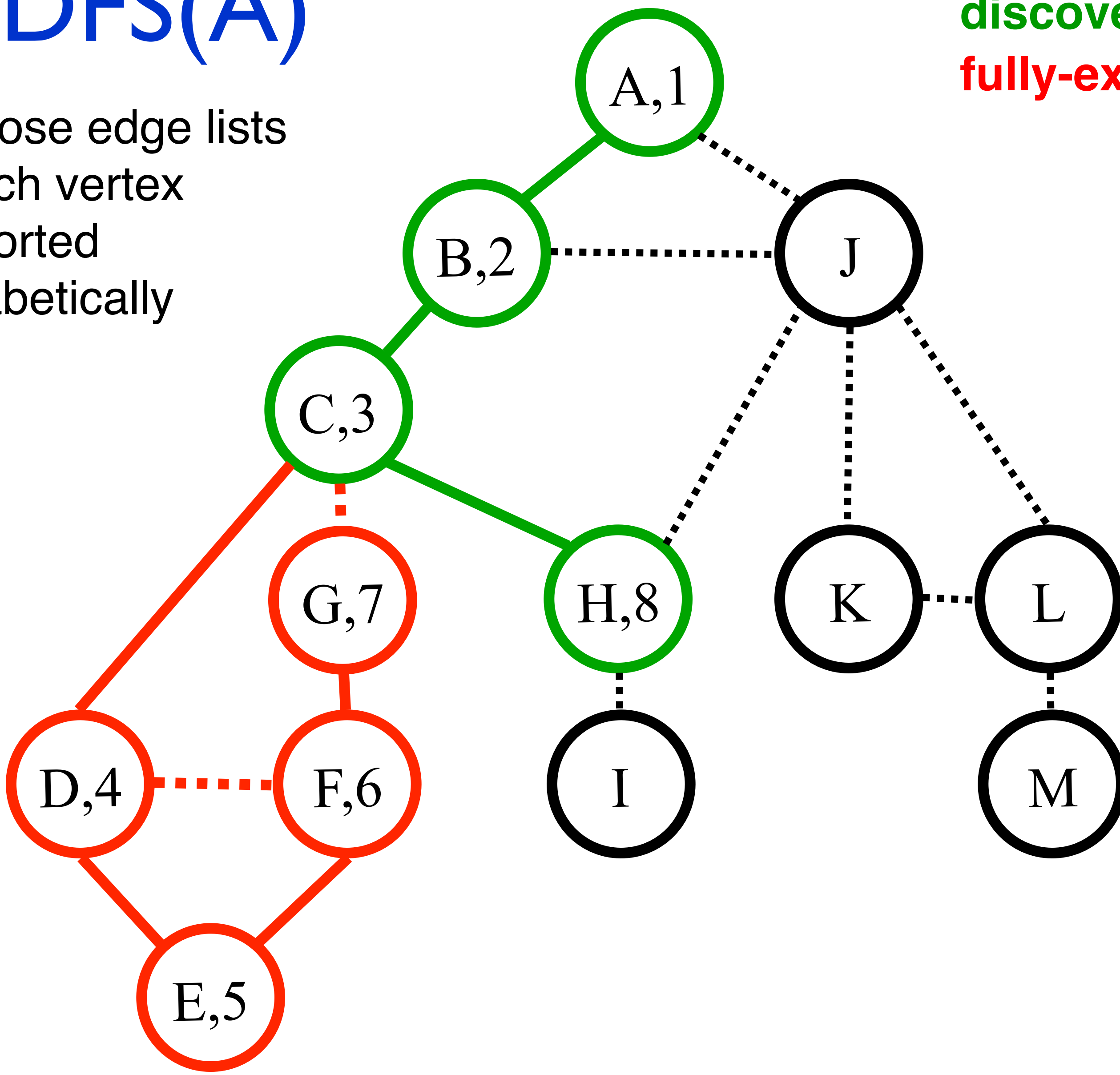


Call Stack:
(Edge list)

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

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



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)

DFS(A)

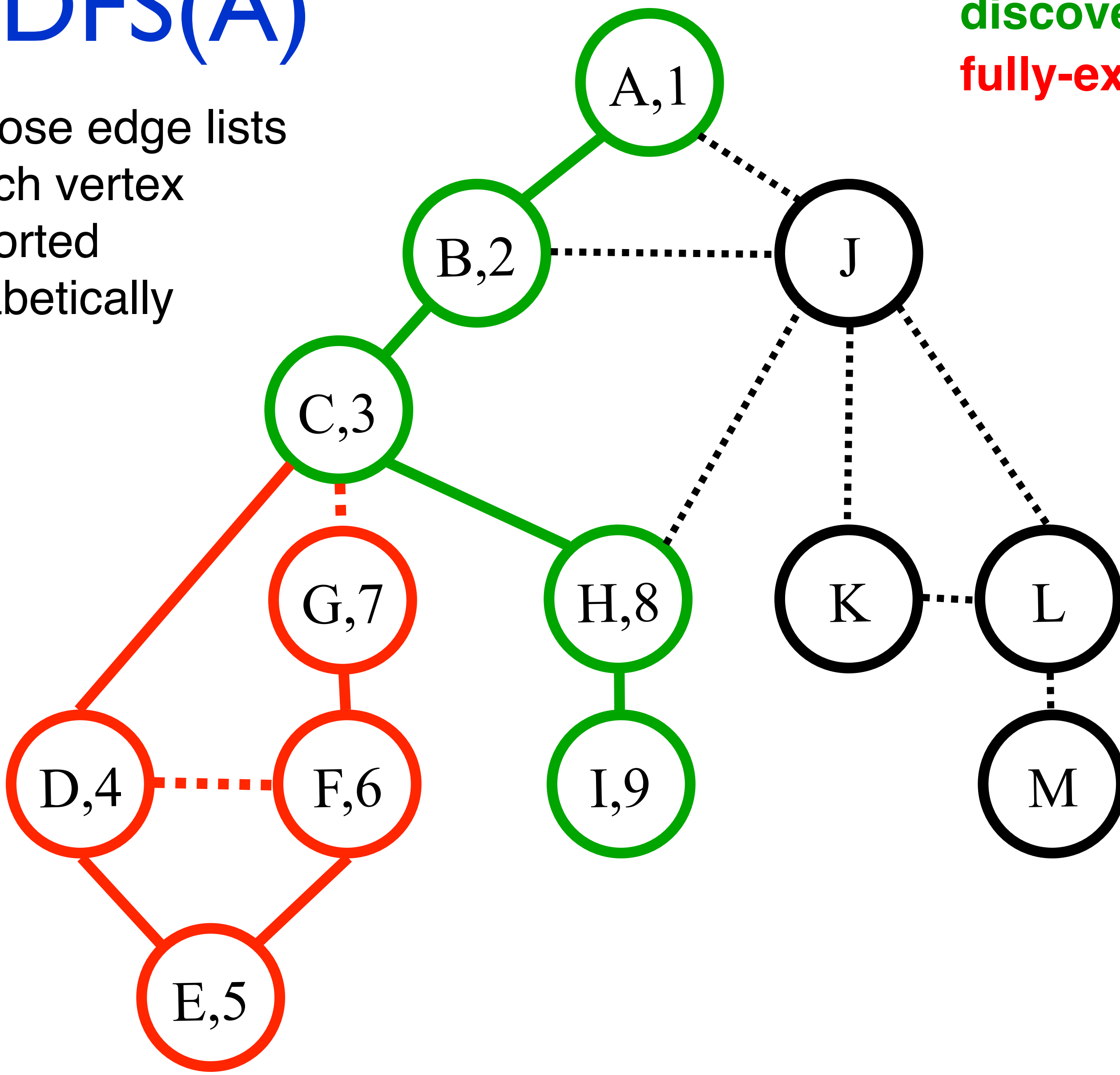
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored

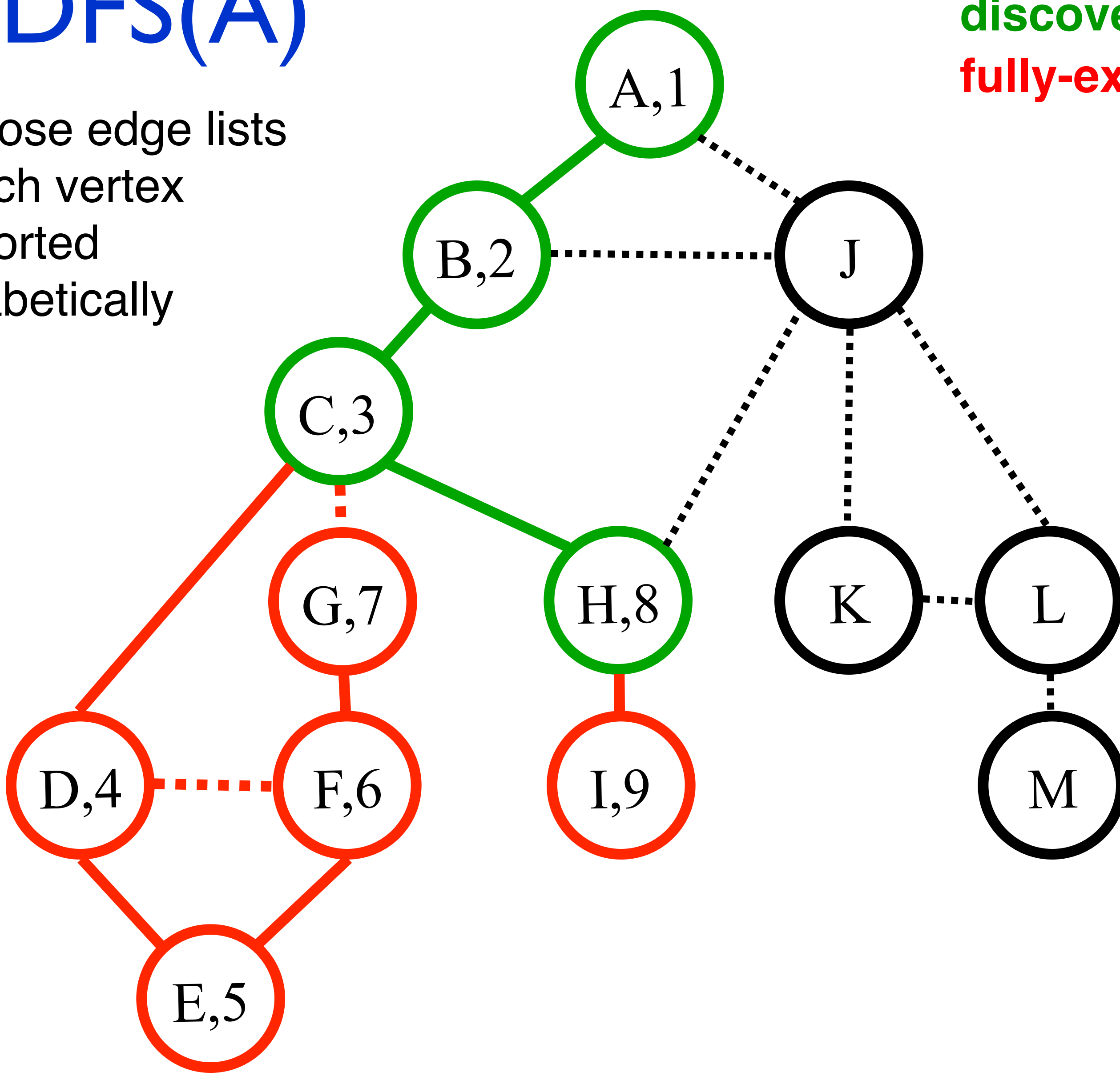


Call Stack:
(Edge list)

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

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



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~~)

DFS(A)

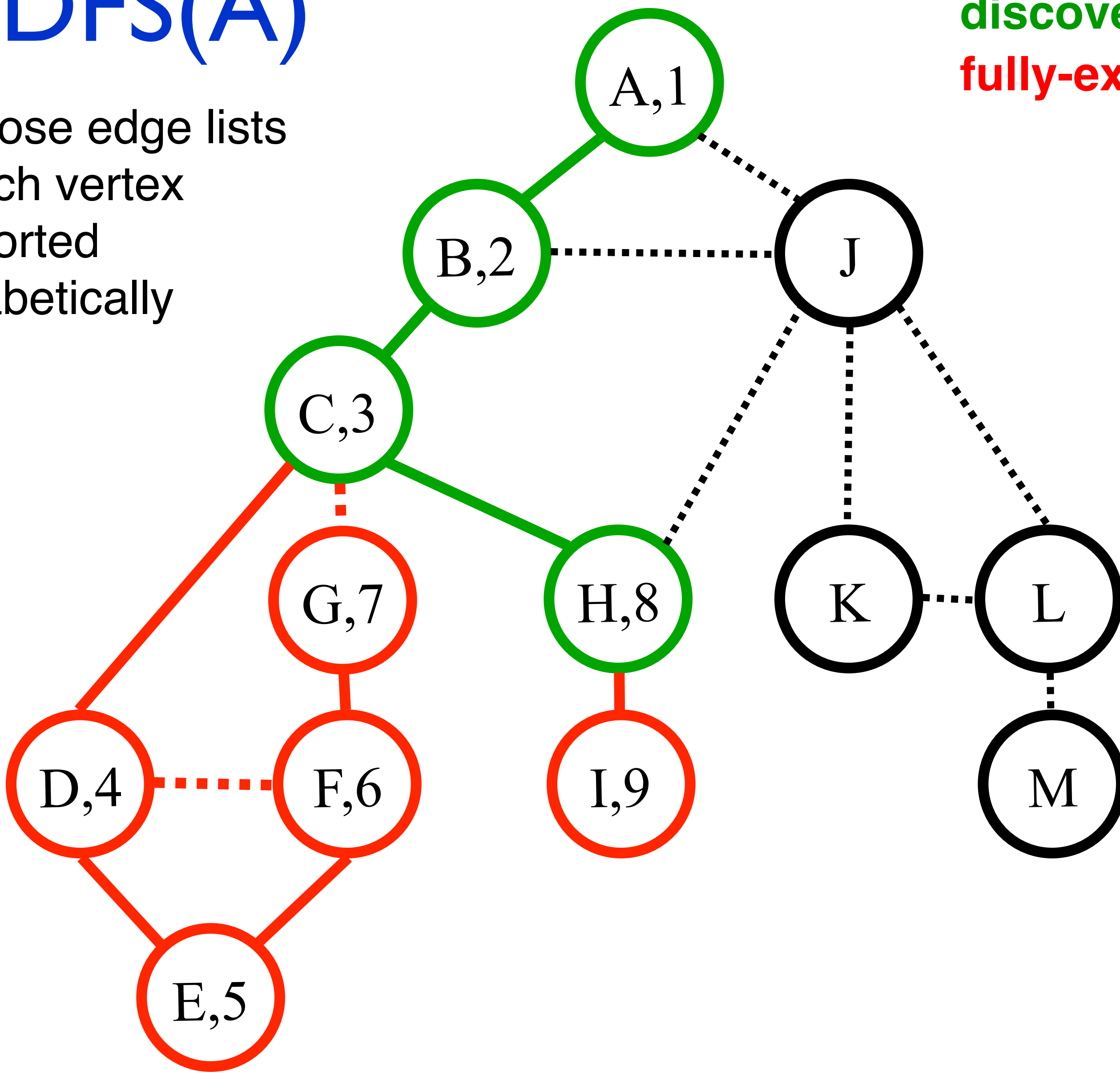
Suppose edge lists at each vertex are sorted alphabetically

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)

DFS(A)

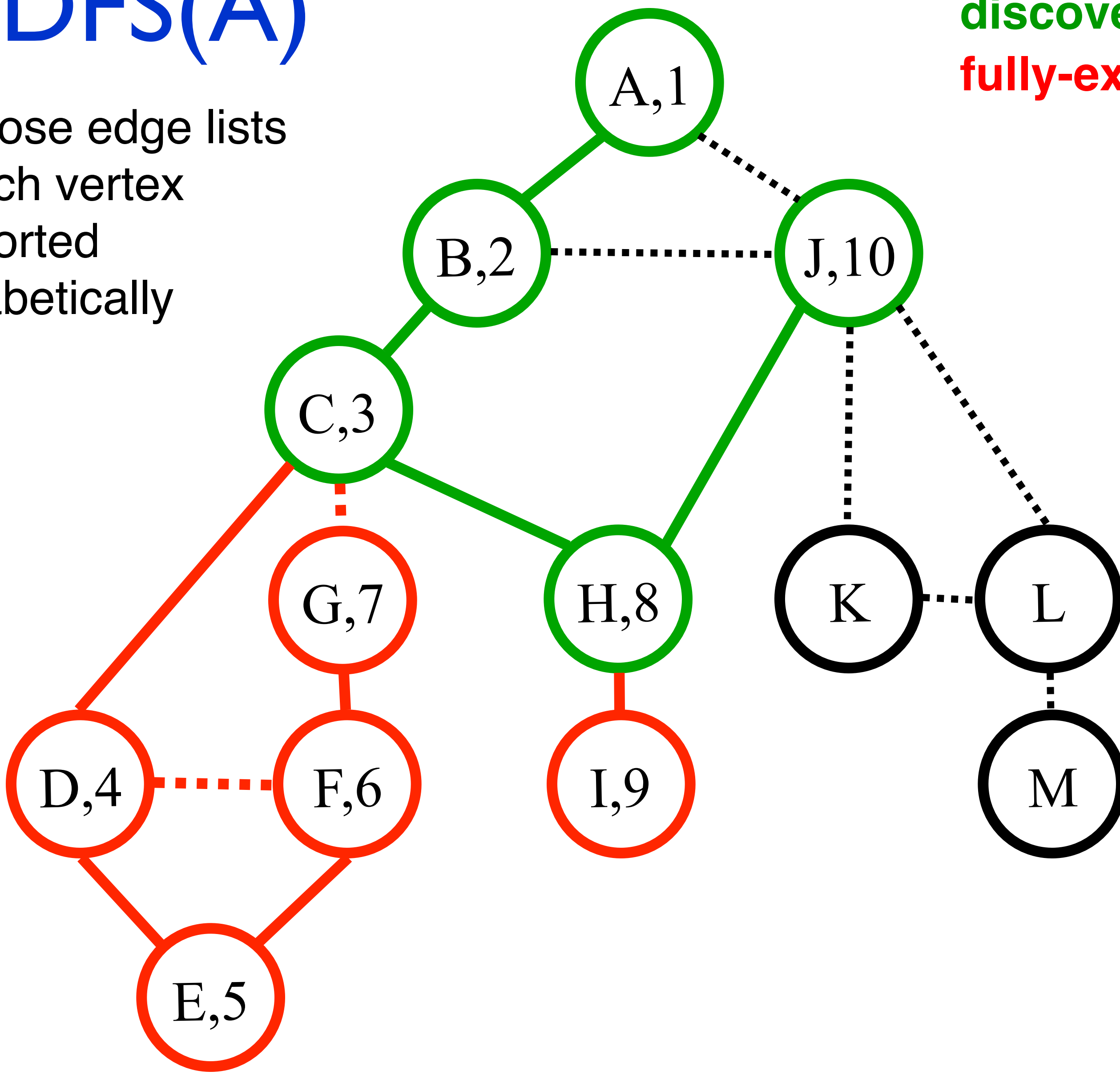
Suppose edge lists at each vertex are sorted alphabetically

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)

DFS(A)

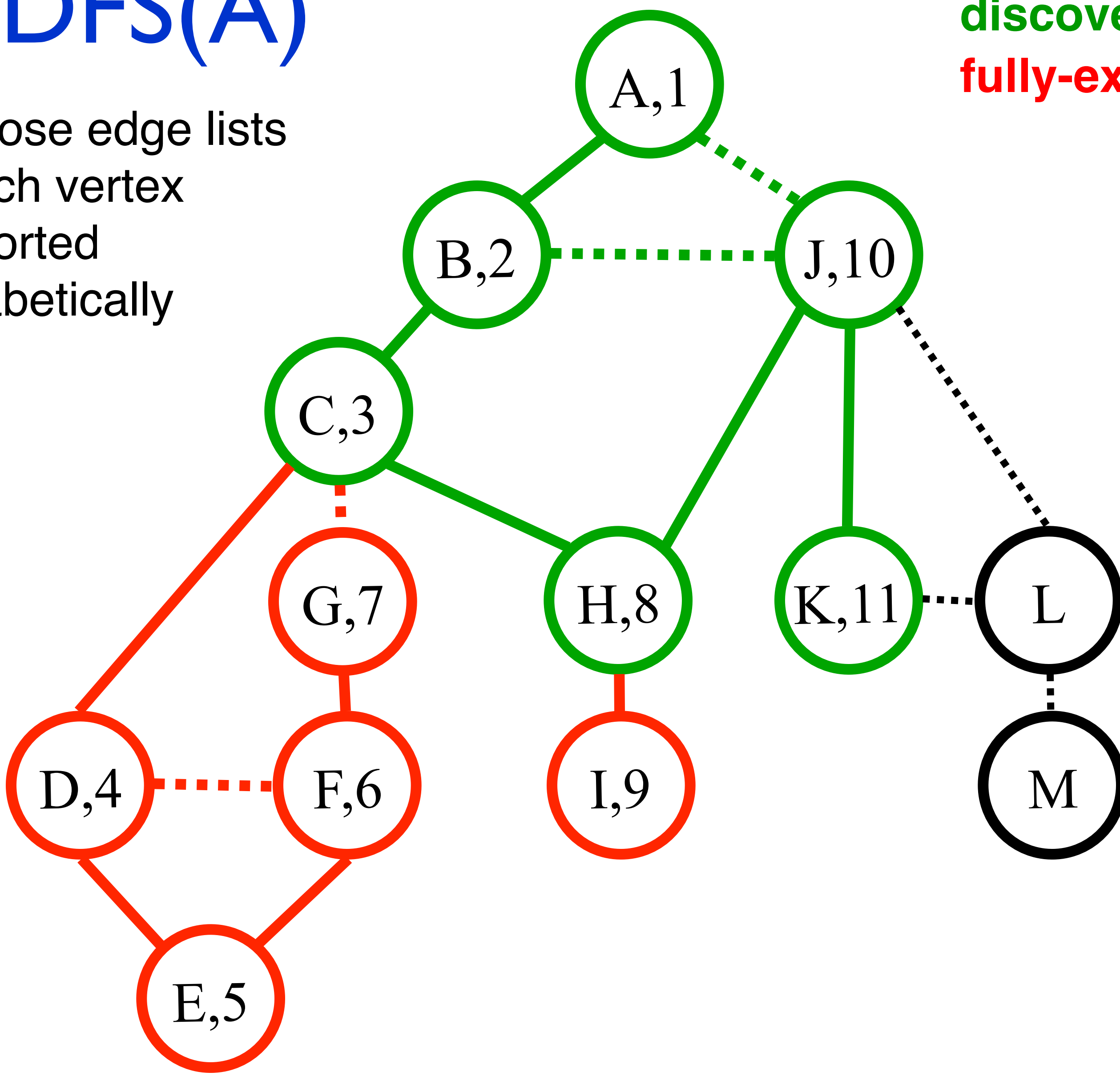
Suppose edge lists at each vertex are sorted alphabetically

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)

DFS(A)

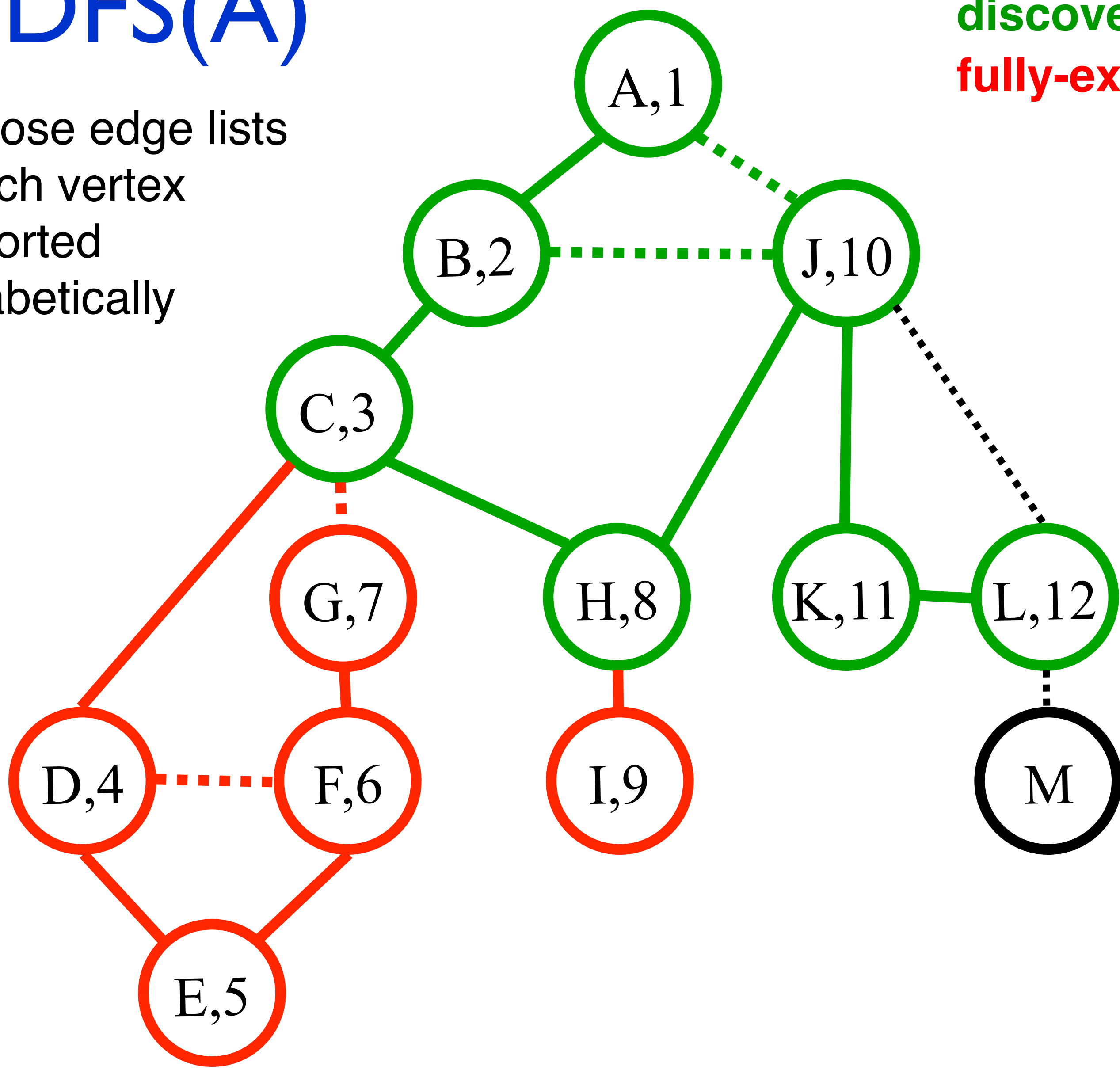
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored



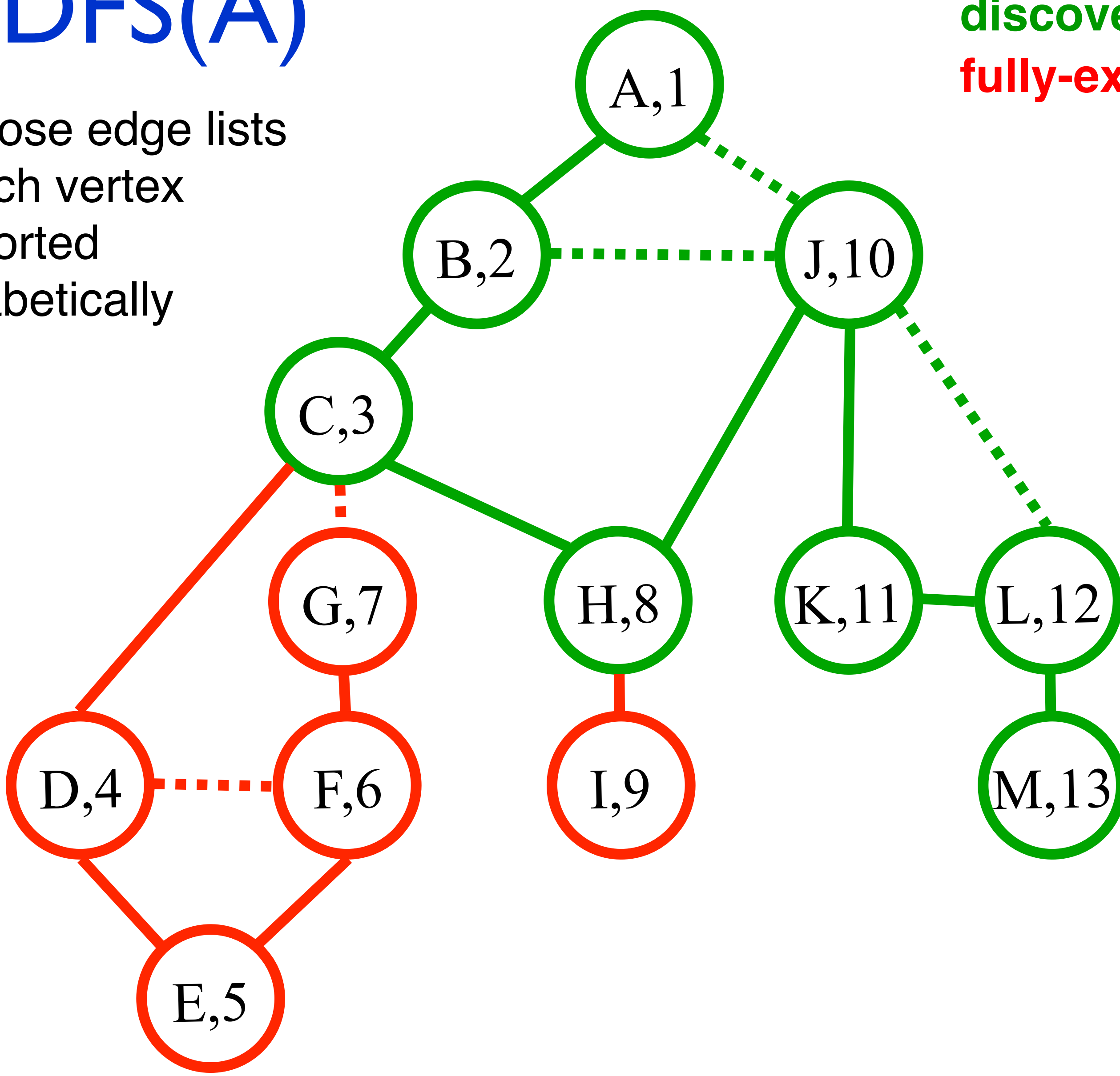
Call Stack:
(Edge list)

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)

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically

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)

DFS(A)

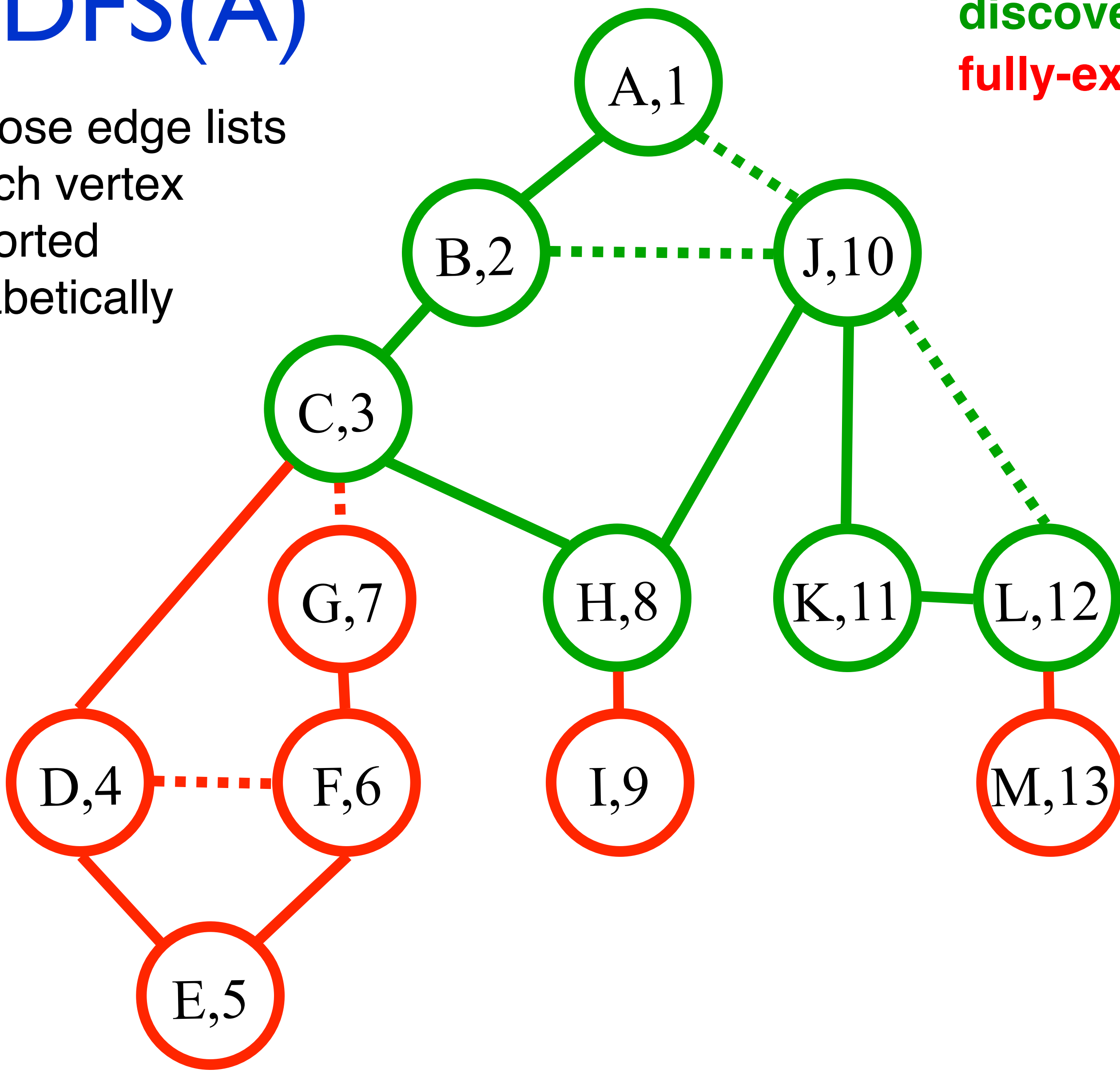
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored

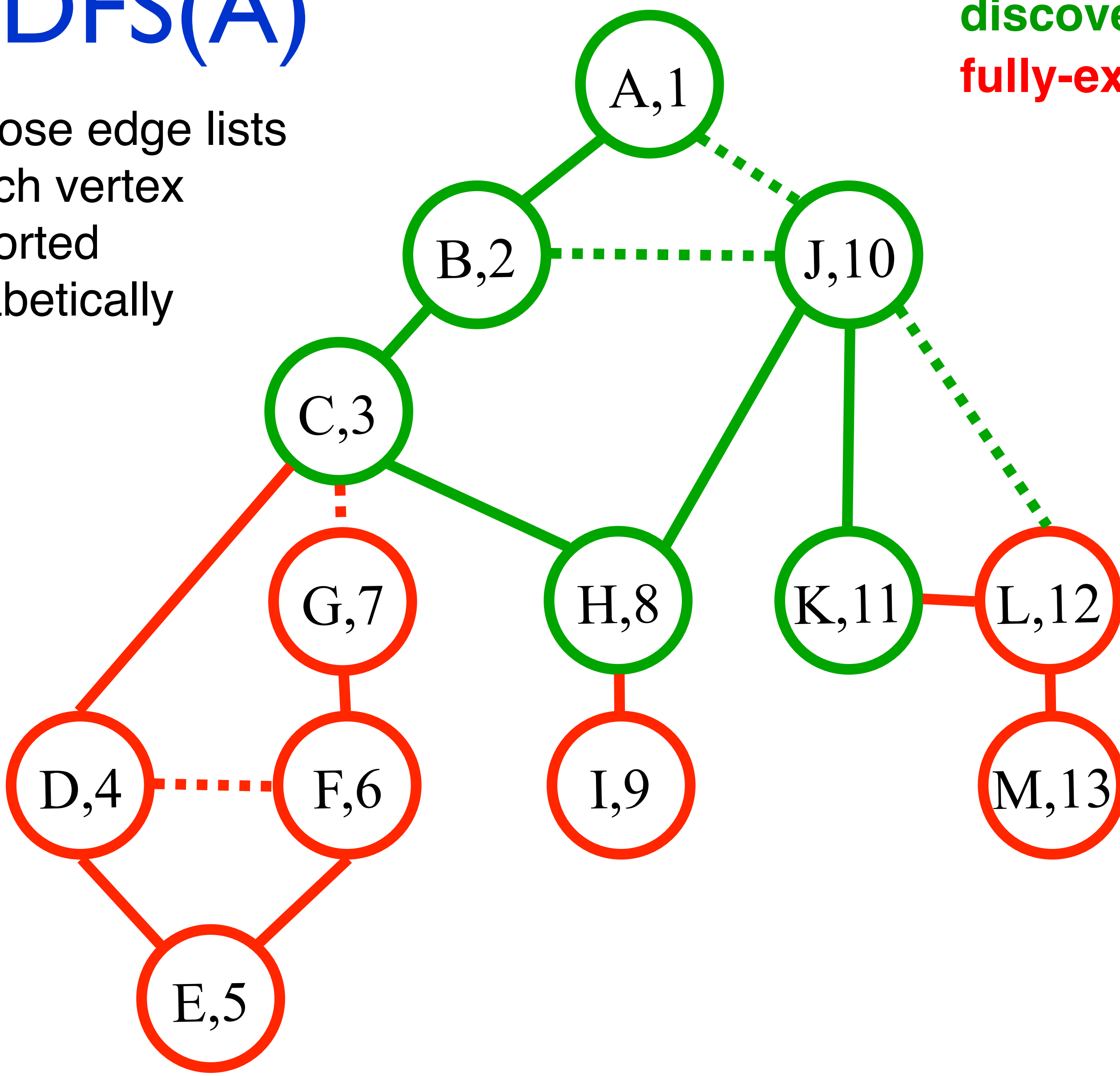


Call Stack:
(Edge list)

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)

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



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)

DFS(A)

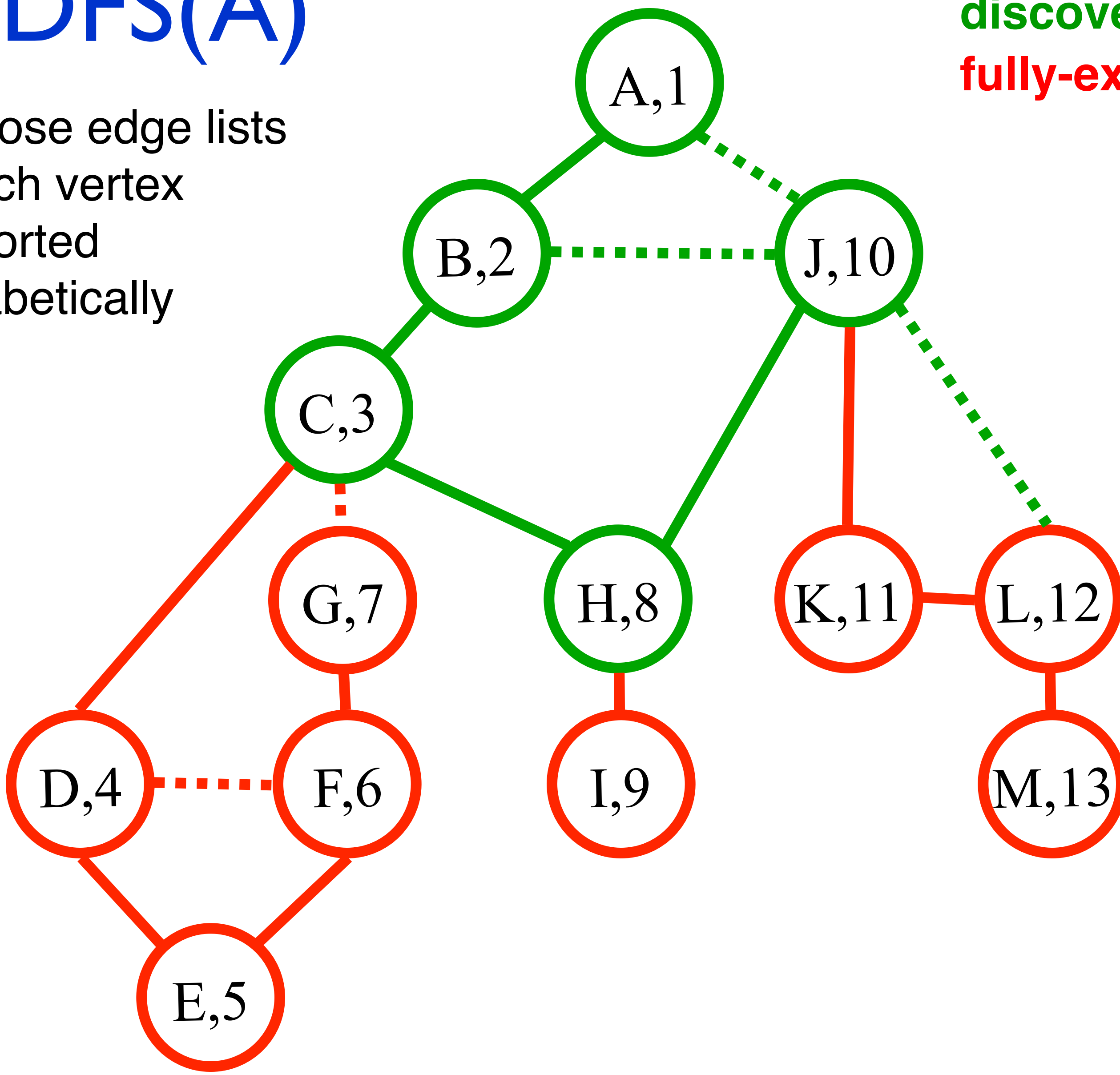
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored



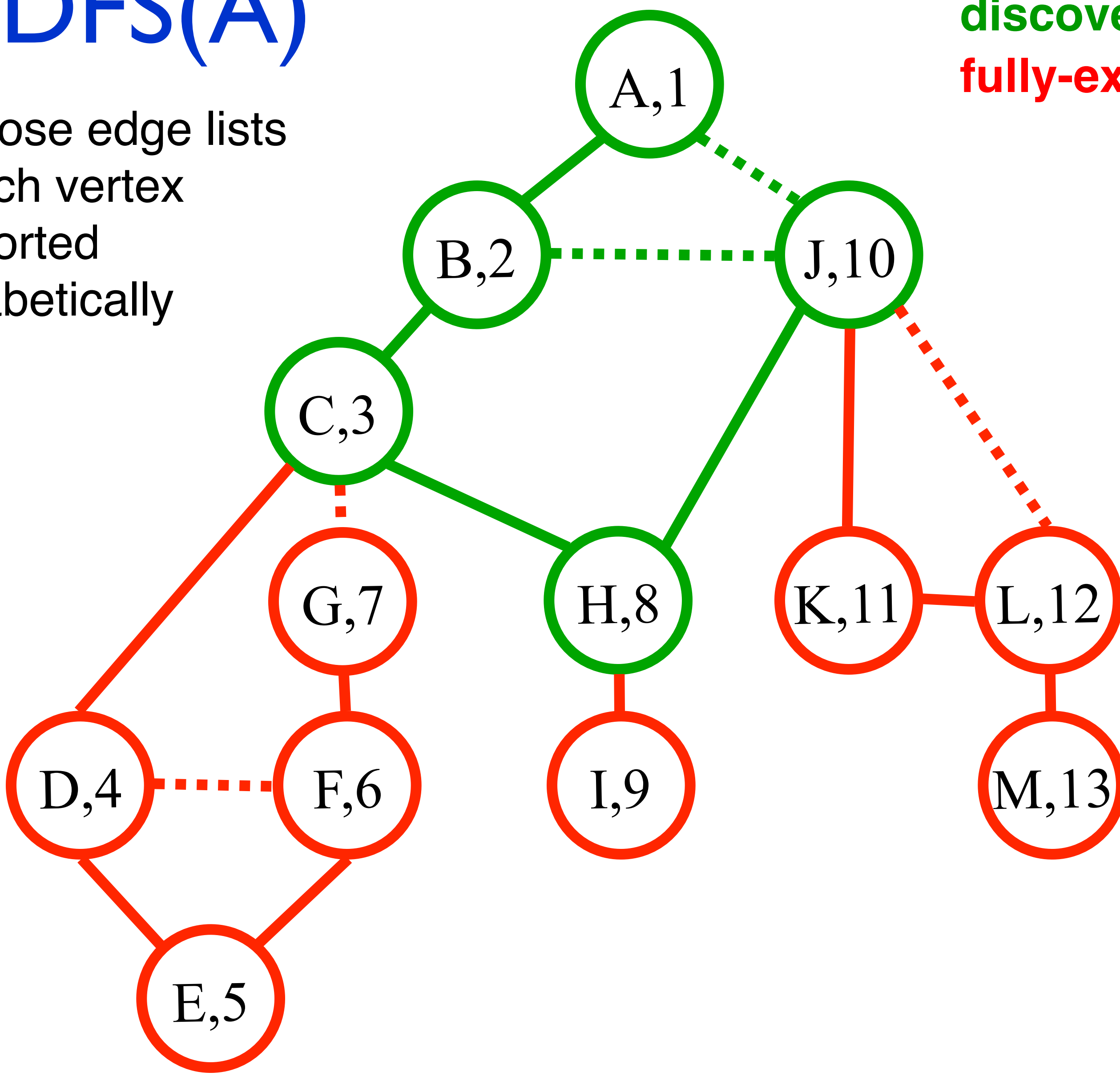
Call Stack:
(Edge list)

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

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically

Color code:
undiscovered
discovered
fully-explored

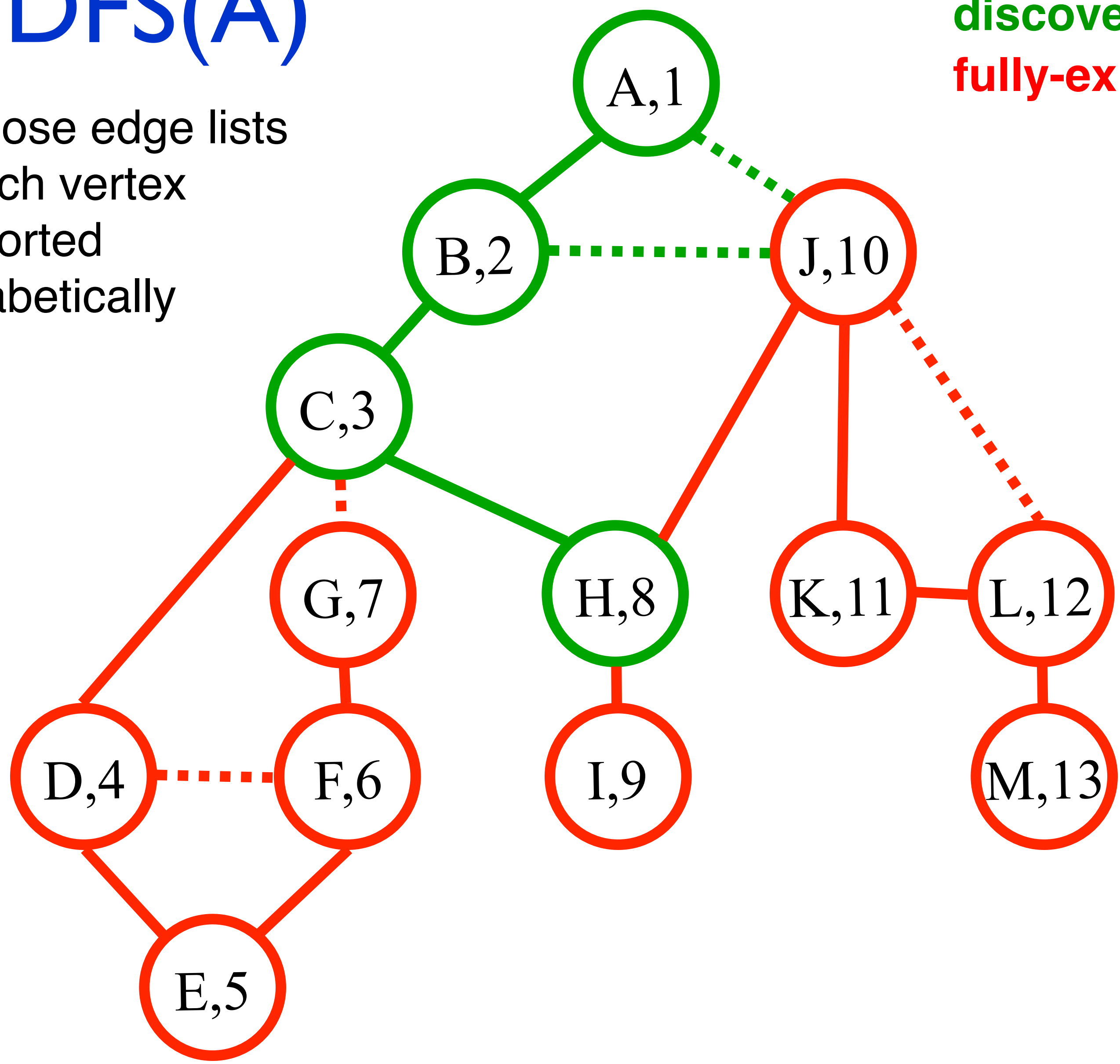


Call Stack:
(Edge list)

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

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



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)

DFS(A)

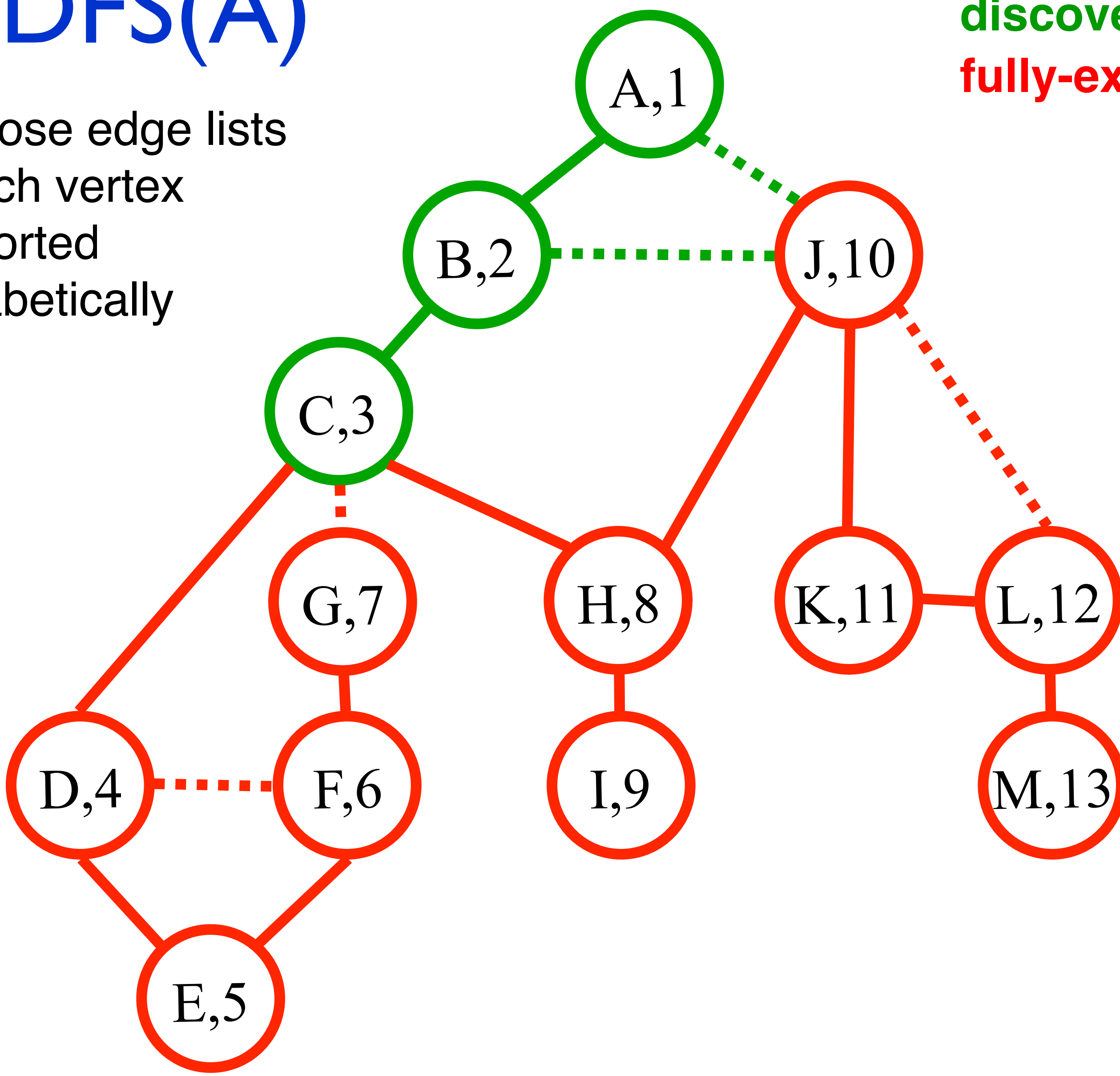
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

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

DFS(A)

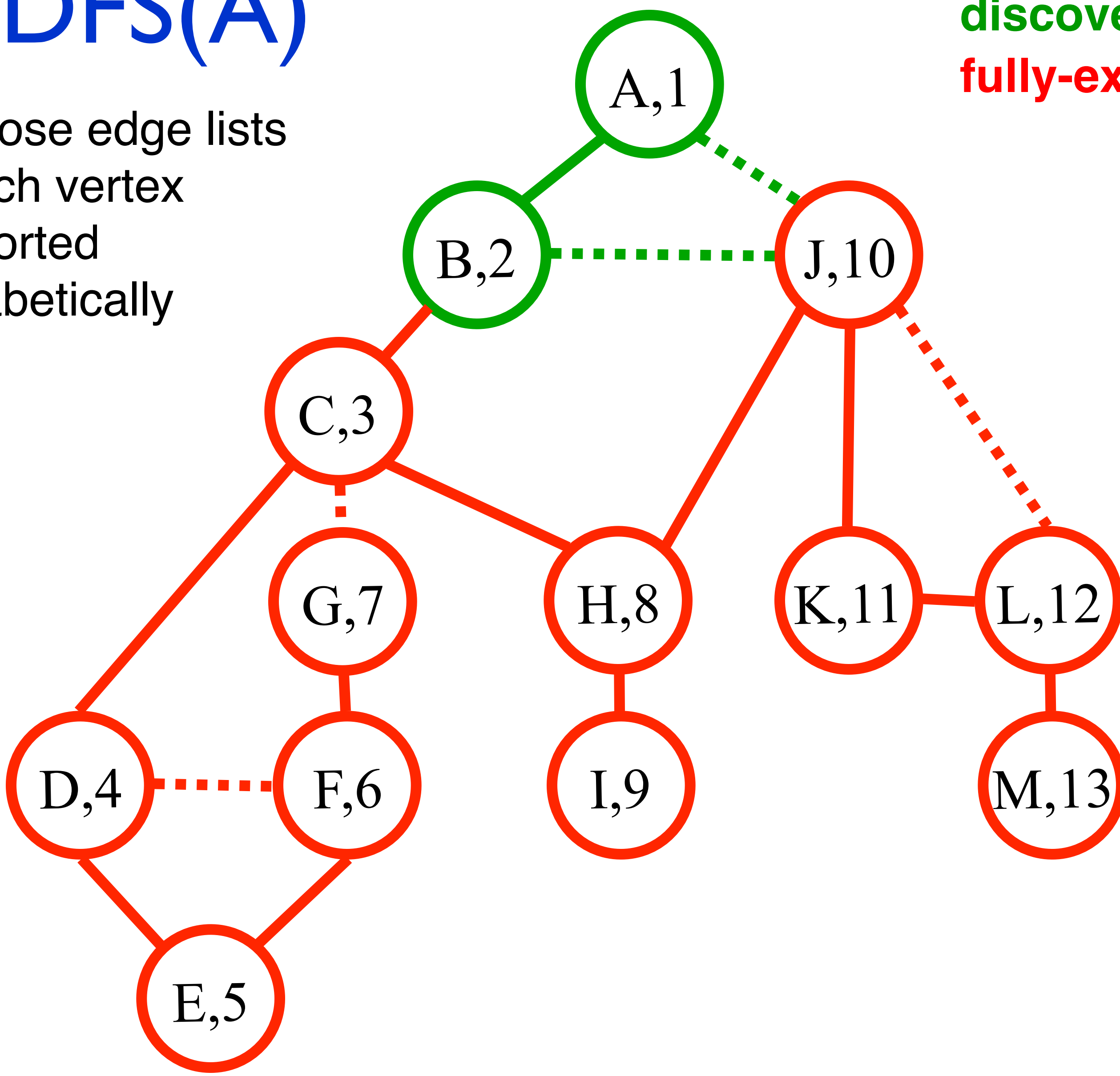
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored

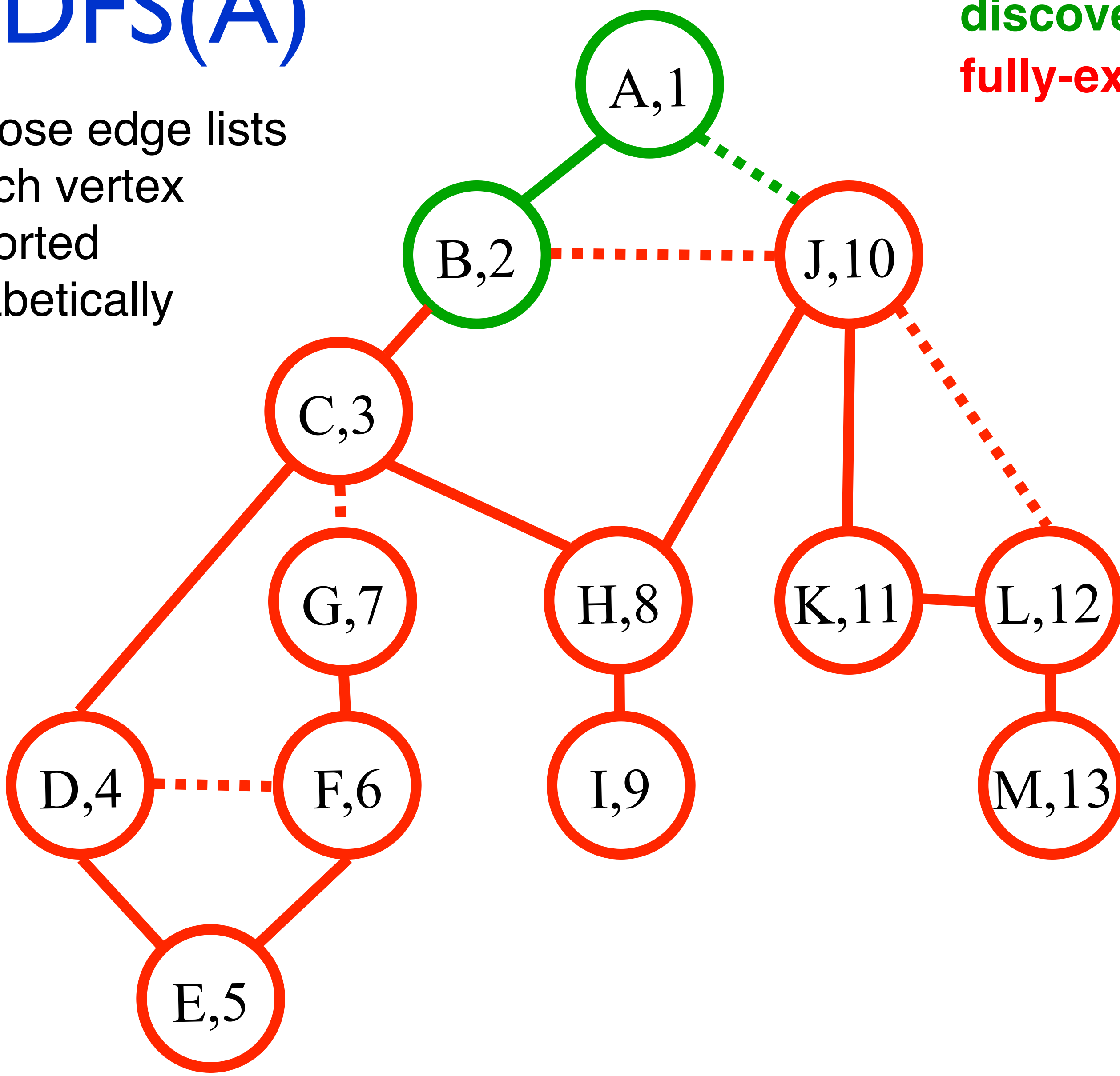


Call Stack:
(Edge list)

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

DFS(A)

Suppose edge lists
at each vertex
are sorted
alphabetically



Color code:

undiscovered

discovered

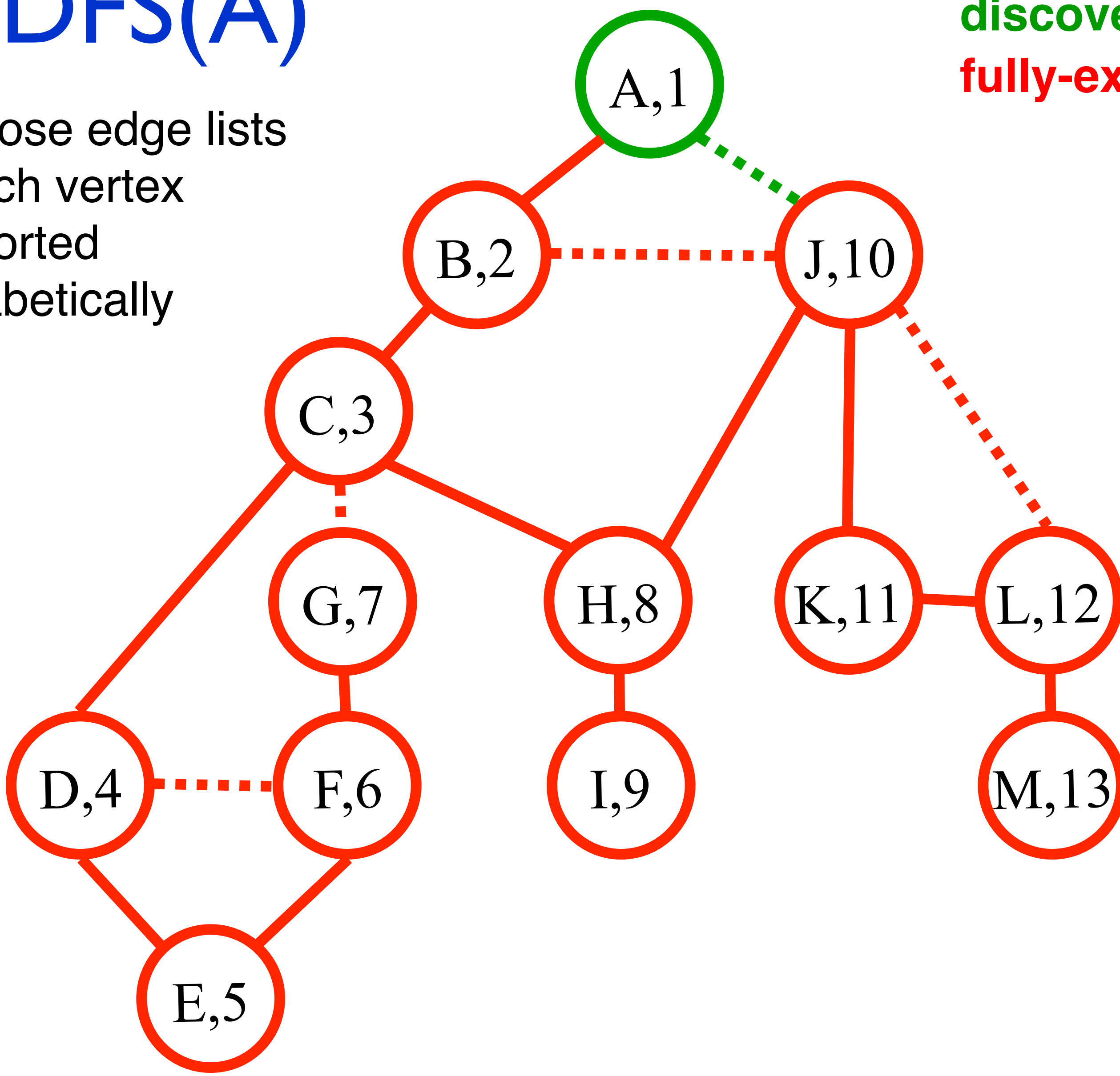
fully-explored

Call Stack:
(Edge list)

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

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



Color code:

undiscovered

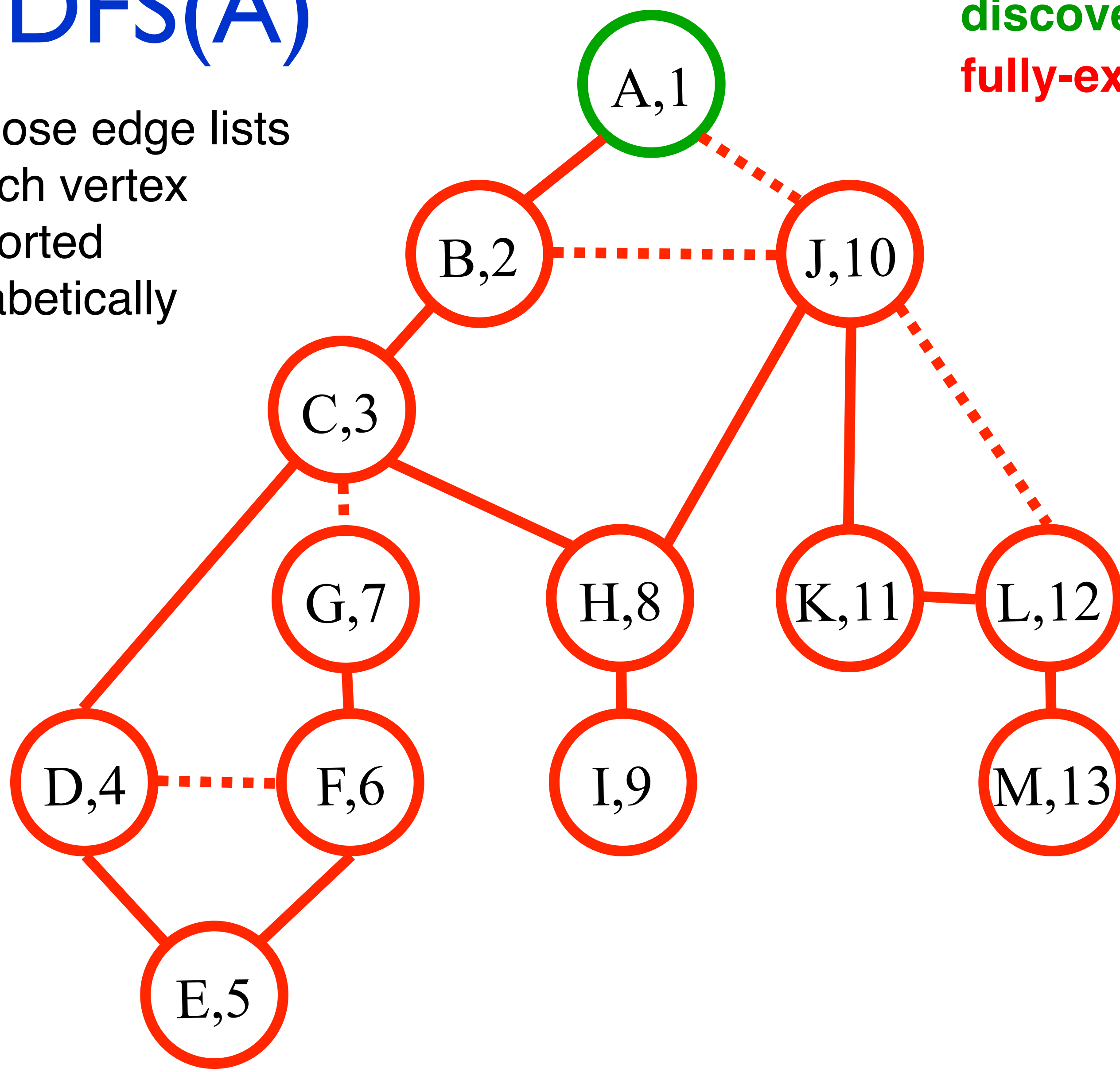
discovered

fully-explored

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

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



Color code:
undiscovered
discovered
fully-explored

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

DFS(A)

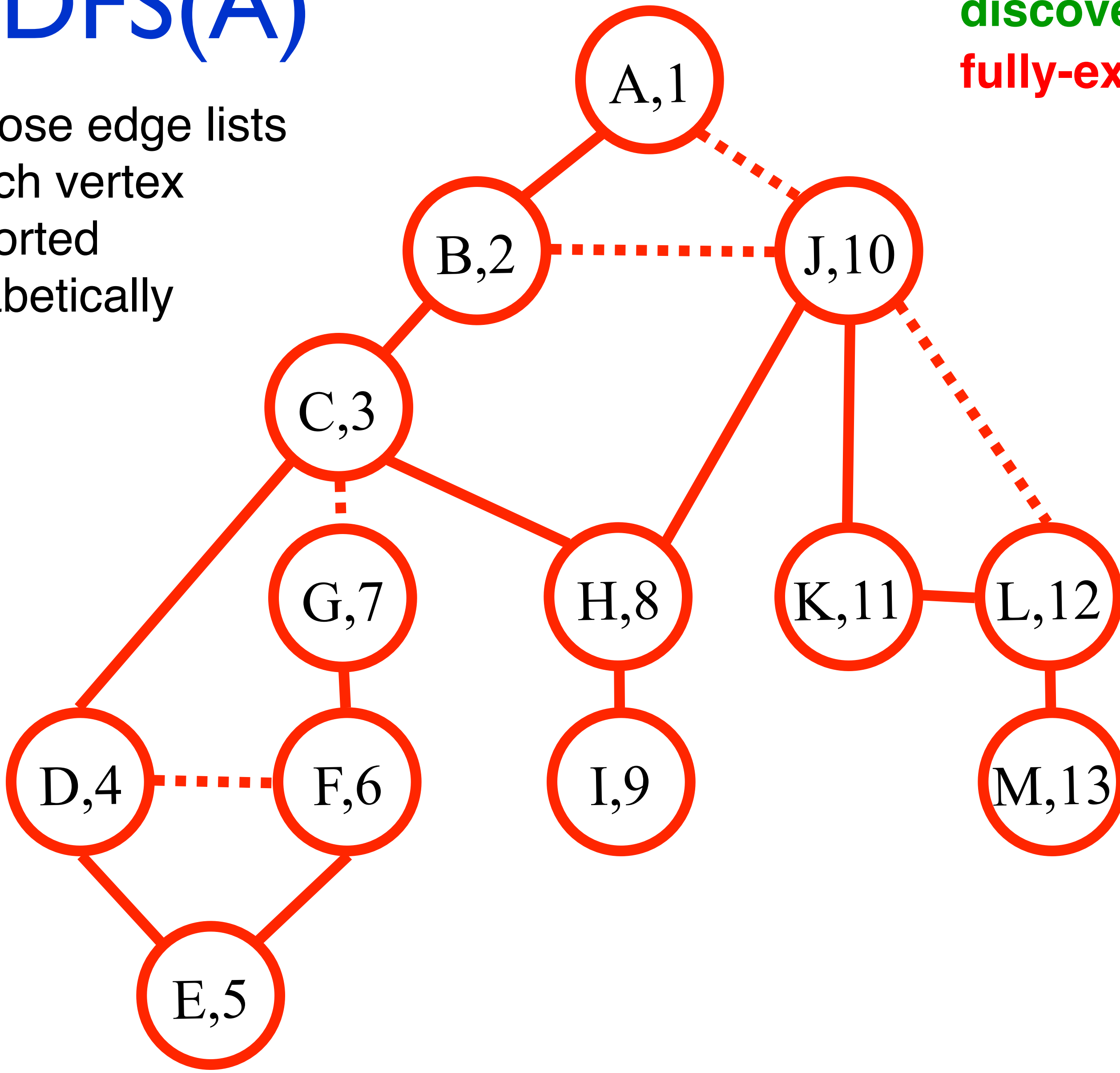
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

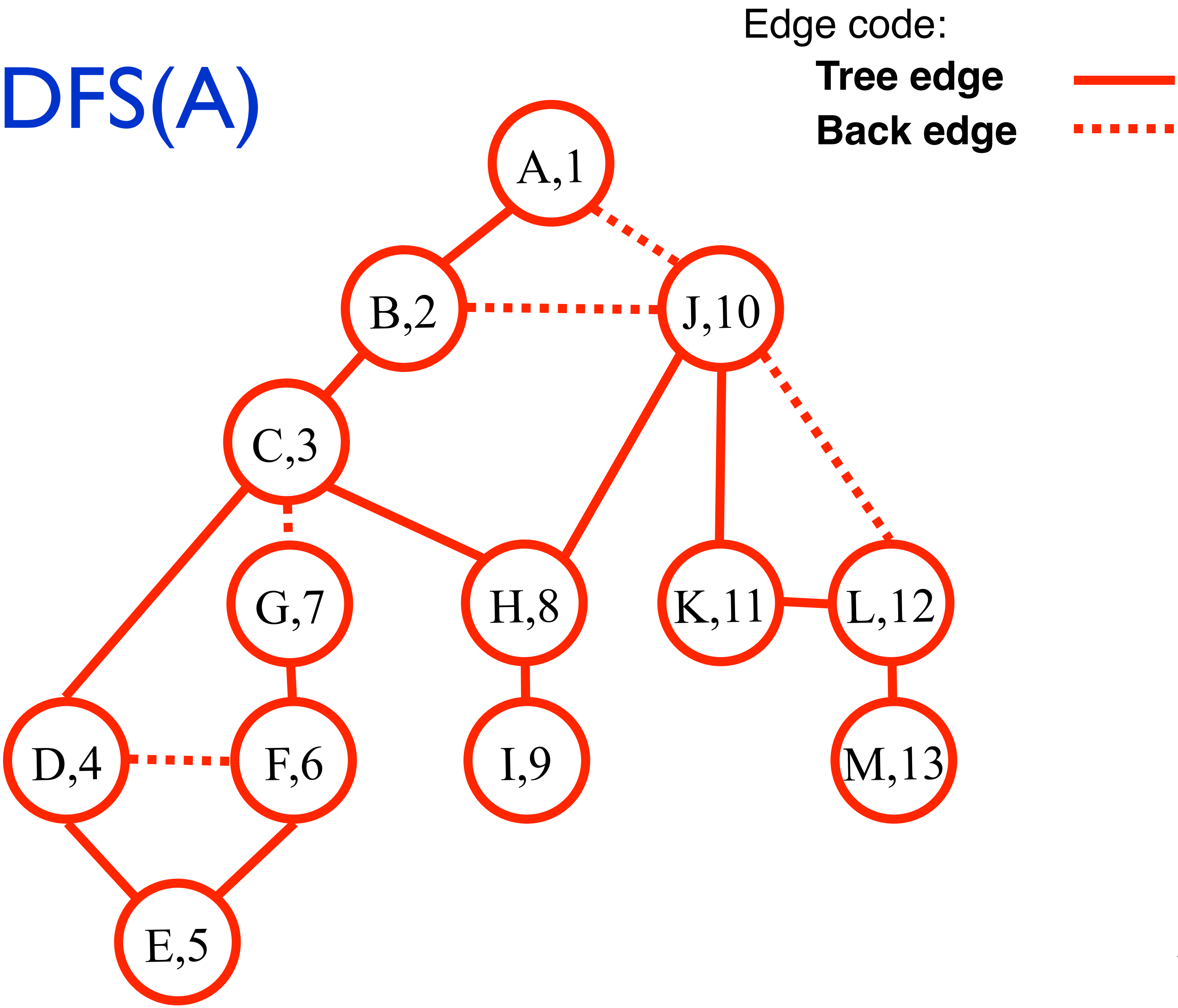
discovered

fully-explored

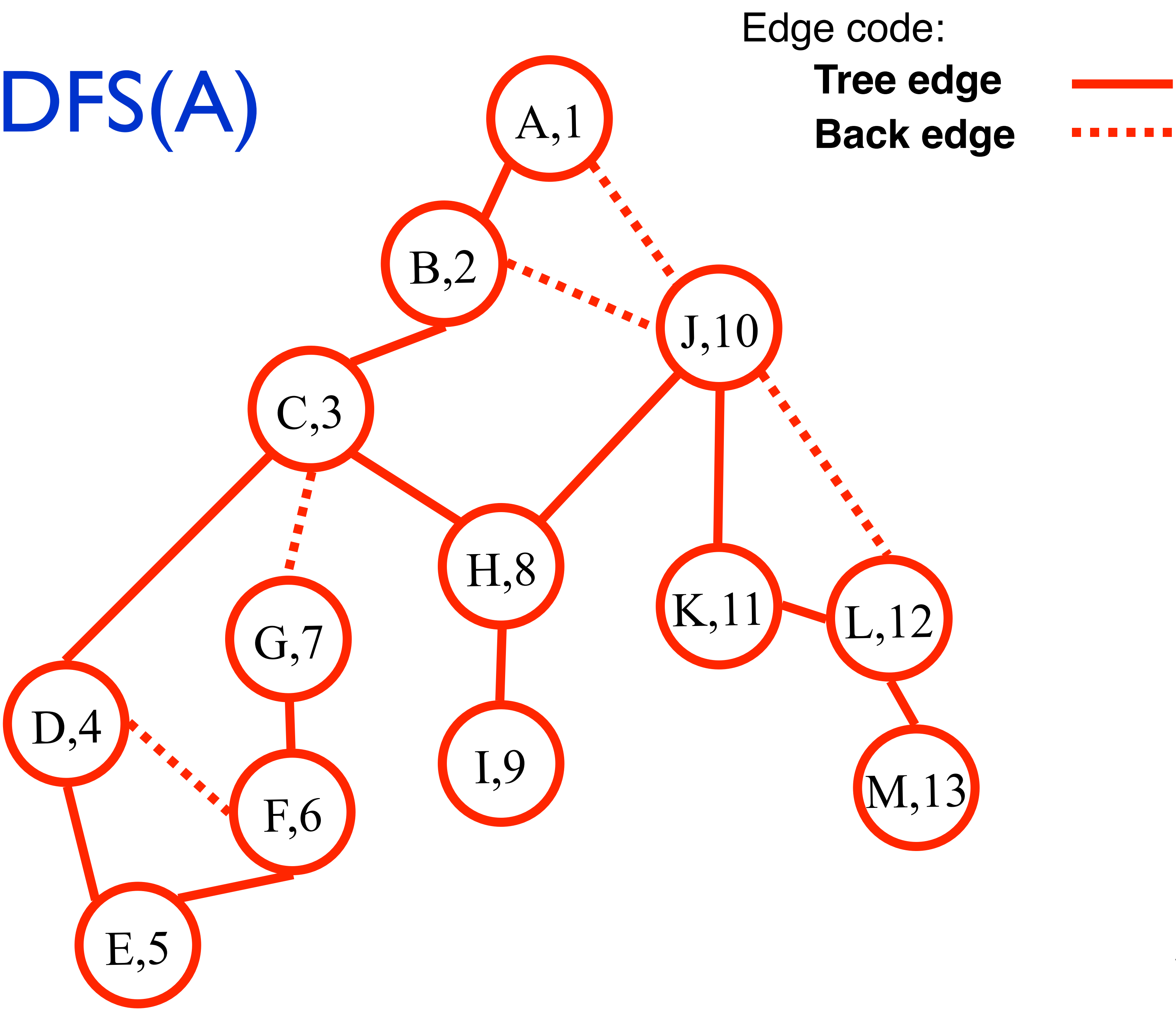


Call Stack: (Edge list)
TA-DA!!

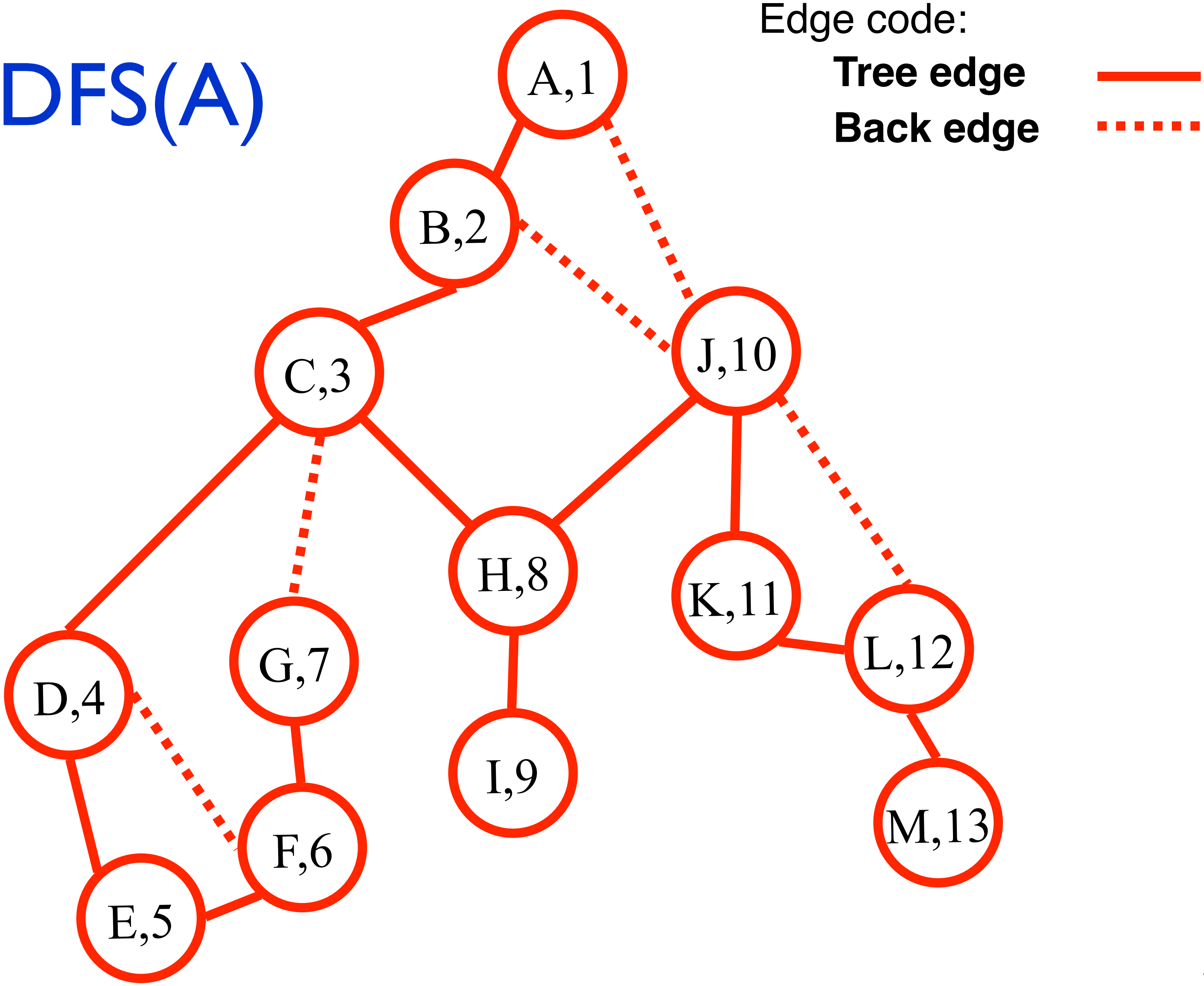
DFS(A)


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)

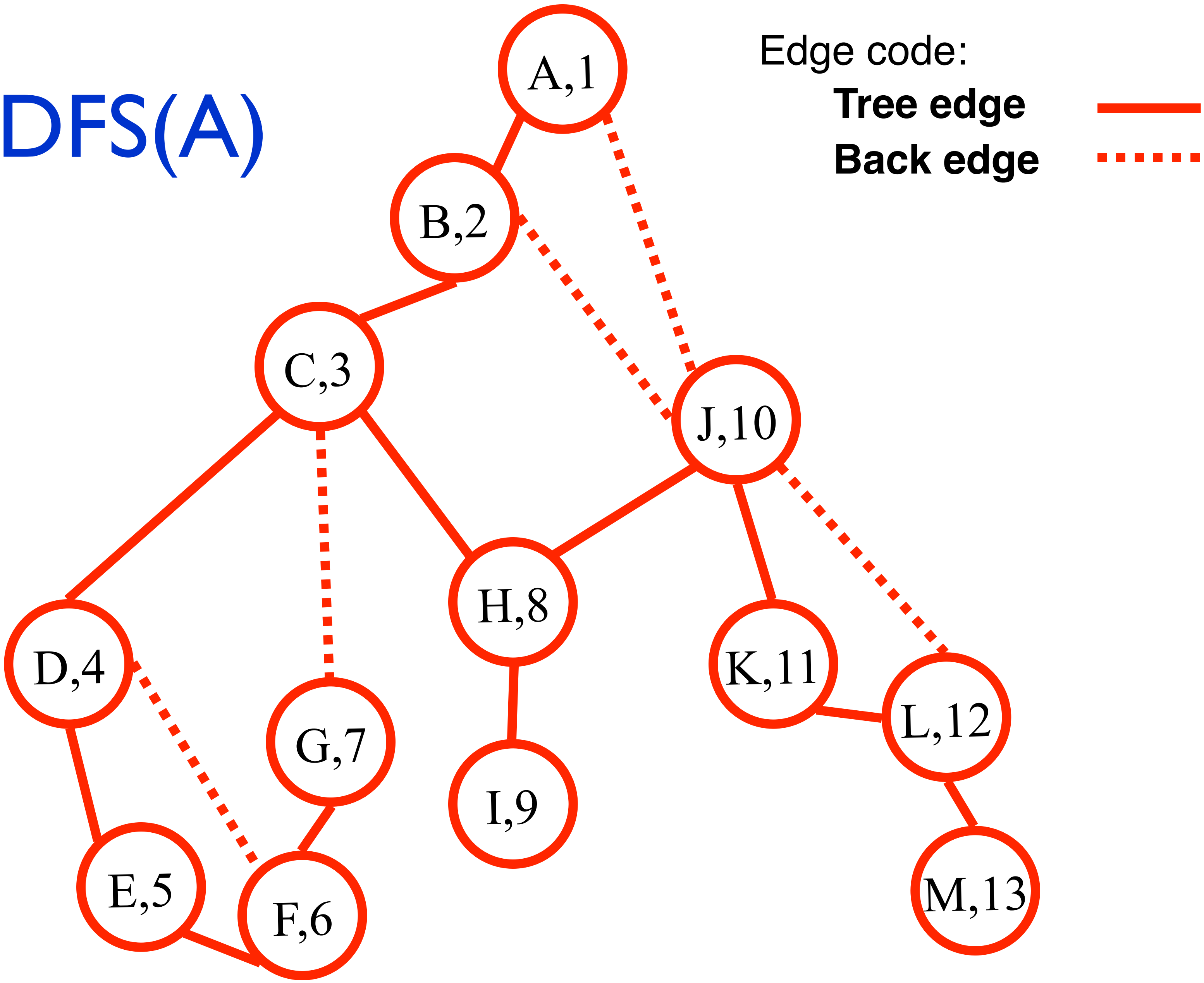

Edge code:
Tree edge
Back edge
A,1
B,2
J,10
C,3
H,8
K,11
L,12
G,7
D,4
I,9
F,6
M,13
E,5

DFS(A)

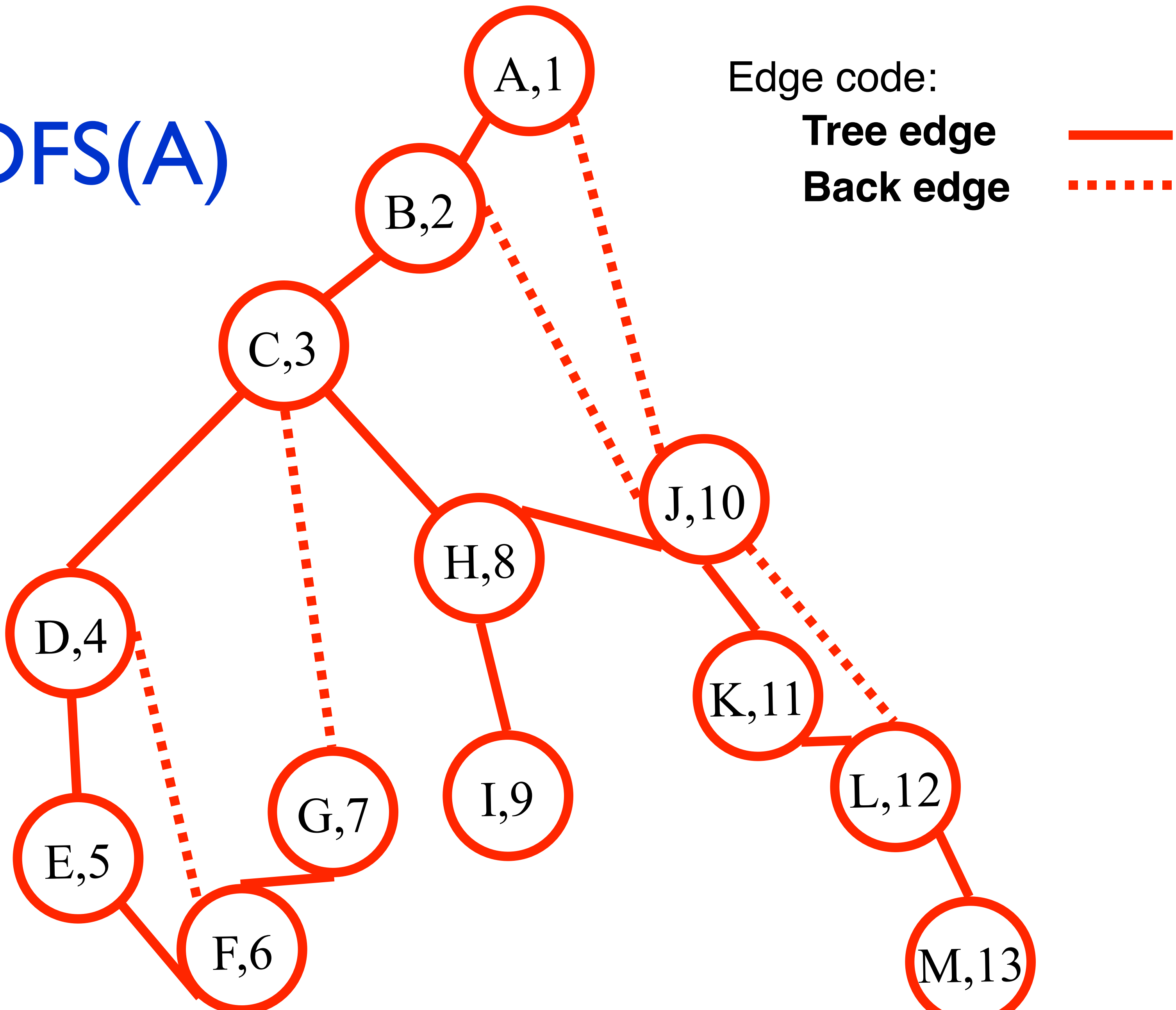


Edge code:
Tree edge
Back edge

DFS(A)

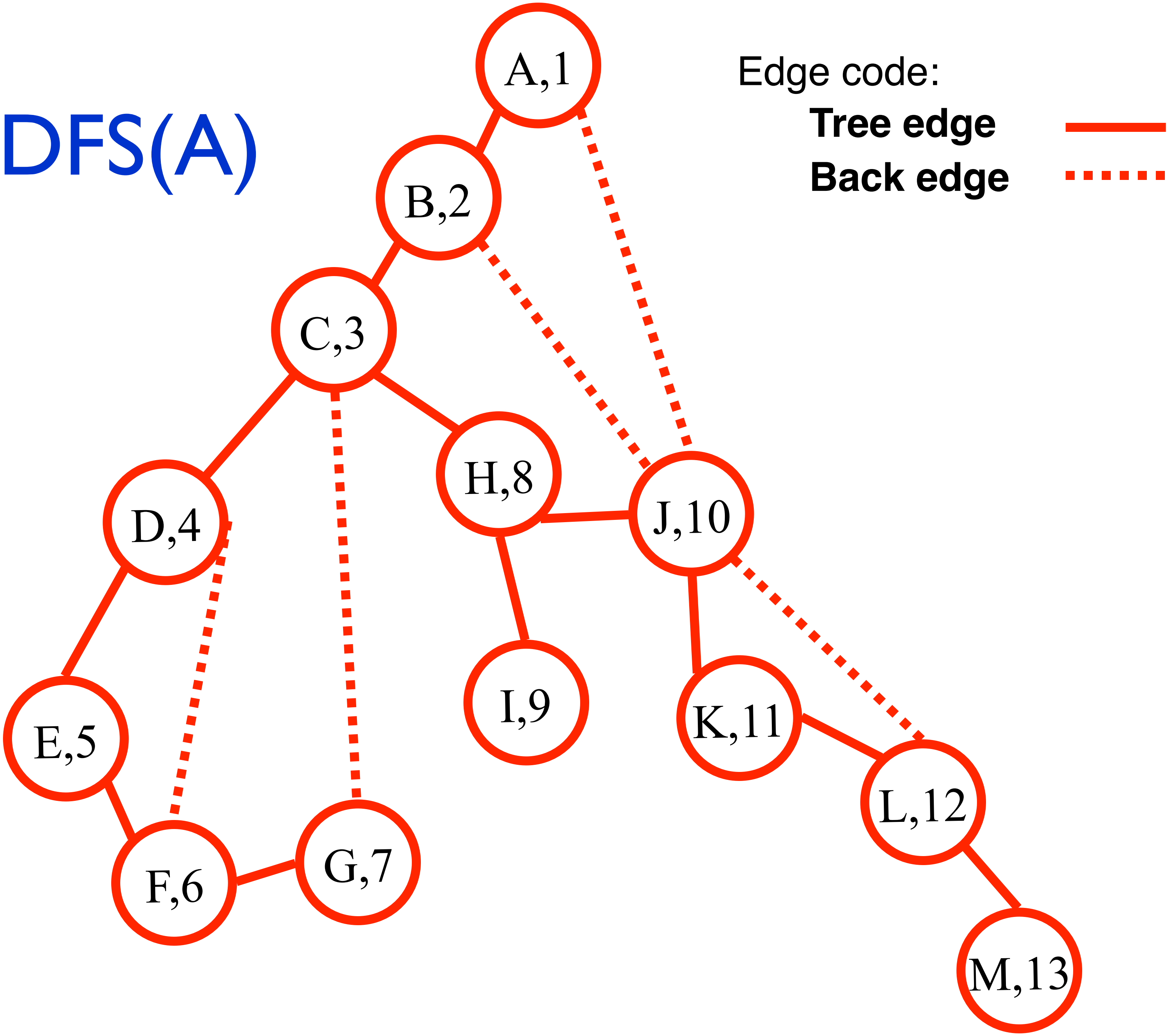

A,1
B,2
C,3
D,4
E,5
F,6
G,7
H,8
I,9
J,10
K,11
L,12
M,13
Edge code:
Tree edge
Back edge

DFS(A)

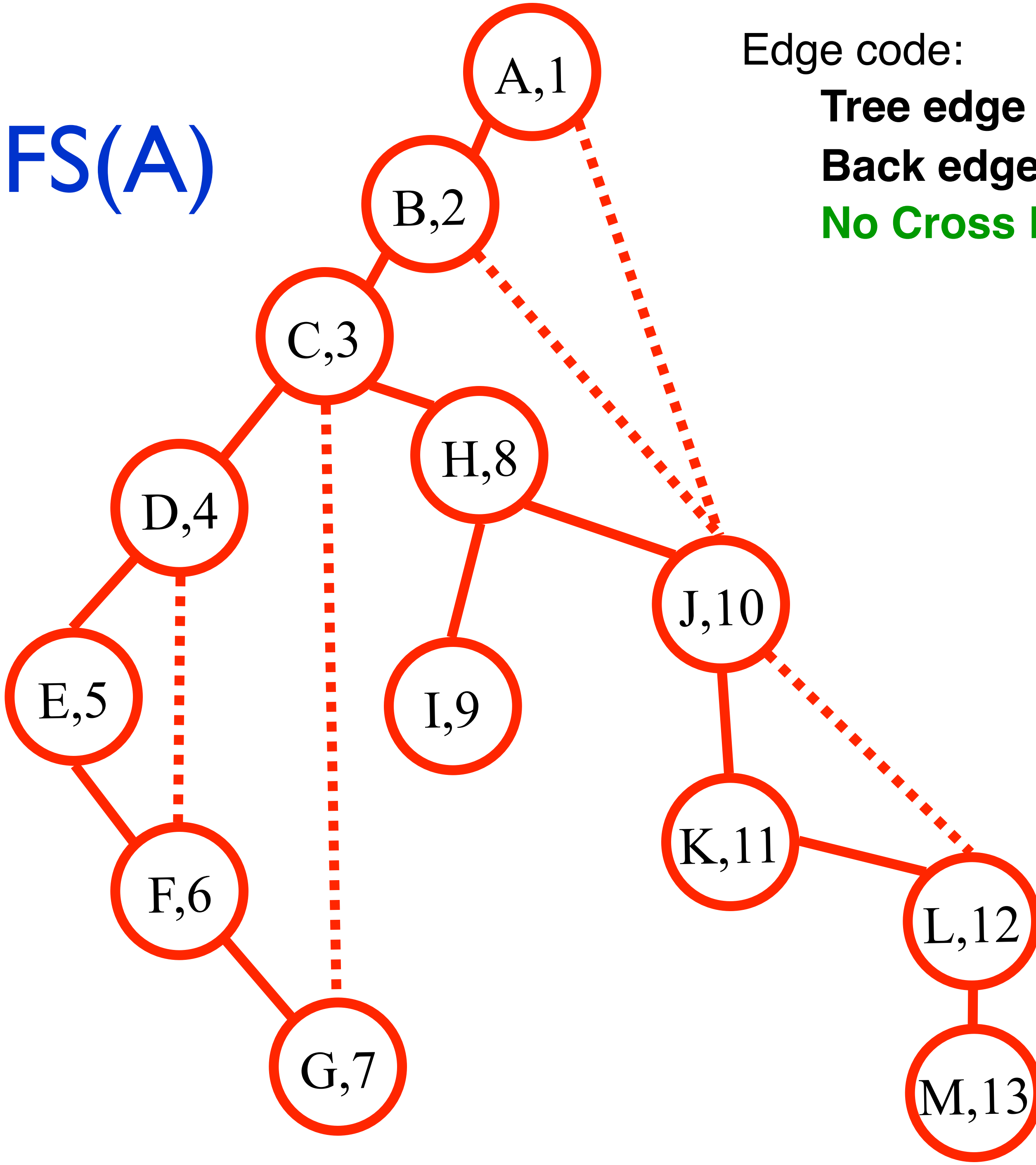


A,1
B,2
C,3
D,4
E,5
F,6
G,7
H,8
I,9
J,10
K,11
L,12
M,13
Edge code:
Tree edge
Back edge

DFS(A)


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

DFS(A)



Edge code:

Tree edge

Back edge

No Cross Edges!

Properties of (Undirected) DFS(v)

Like BFS(v):

- DFS(v) visits x if and only if there is a path in G from v to x (through previously unvisited vertices)
- Edges into then-undiscovered vertices define a ***tree*** – the "depth first spanning tree" of G

Unlike the BFS tree:

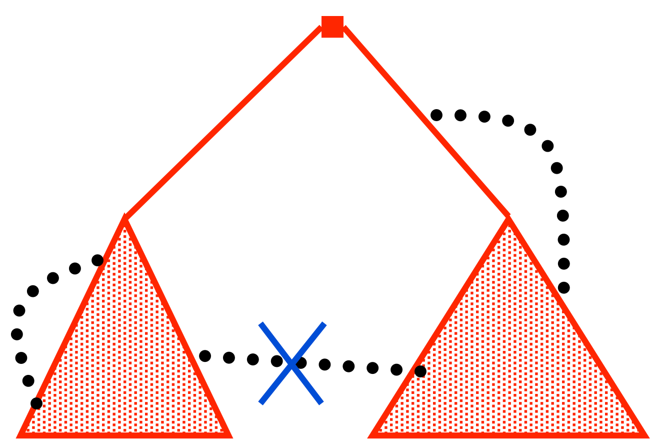
- the DF spanning tree isn't minimum depth
- its levels don't reflect min distance from the root
- non-tree edges never join vertices on the same or adjacent levels

BUT…

Non-tree edges

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

No cross edges!



Why fuss about trees (again)?

As with BFS, DFS has found a tree in the graph s.t. non-tree edges are "simple"--only descendant/ancestor

A simple problem on trees

Given: tree T, a value L(v) defined for every vertex v in T

Goal: find M(v), the min value of L(v) anywhere in the subtree rooted at v (including v itself).

How? Depth first search, using:

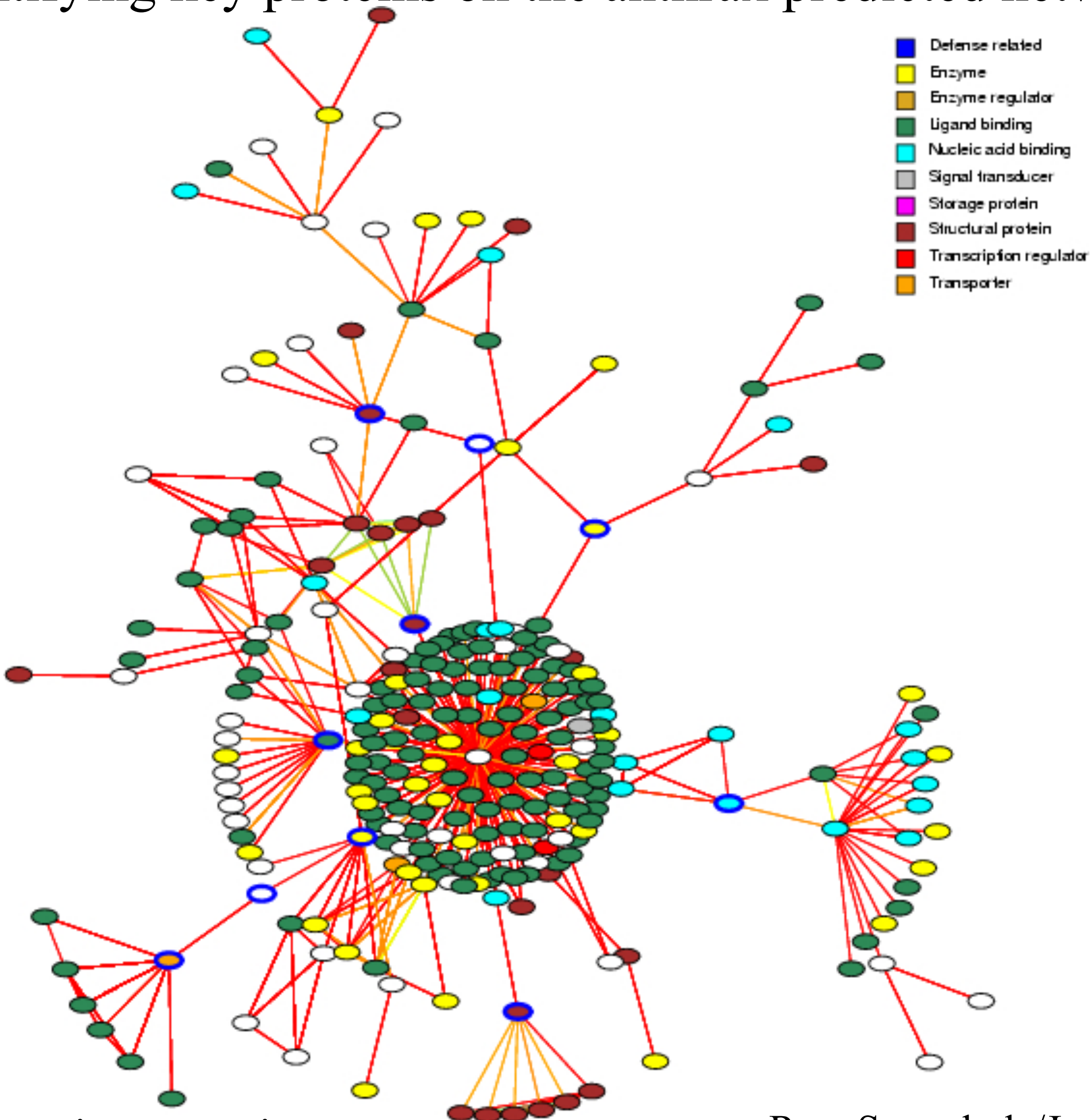
$$M(v) = \begin{cases} L(v) & \text{if } v \text{ is a leaf} \\ \min(L(v),\ \min_{w \text{ a child of v}} M(w)) & \text{otherwise} \end{cases}$$

Application: Articulation Points

A node in an undirected graph is an **articulation point** iff removing it disconnects the graph (or, more generally, increases the number of connected components)

Articulation points represent vulnerabilities in a network – single points whose failure would split the network into 2 or more disconnected components

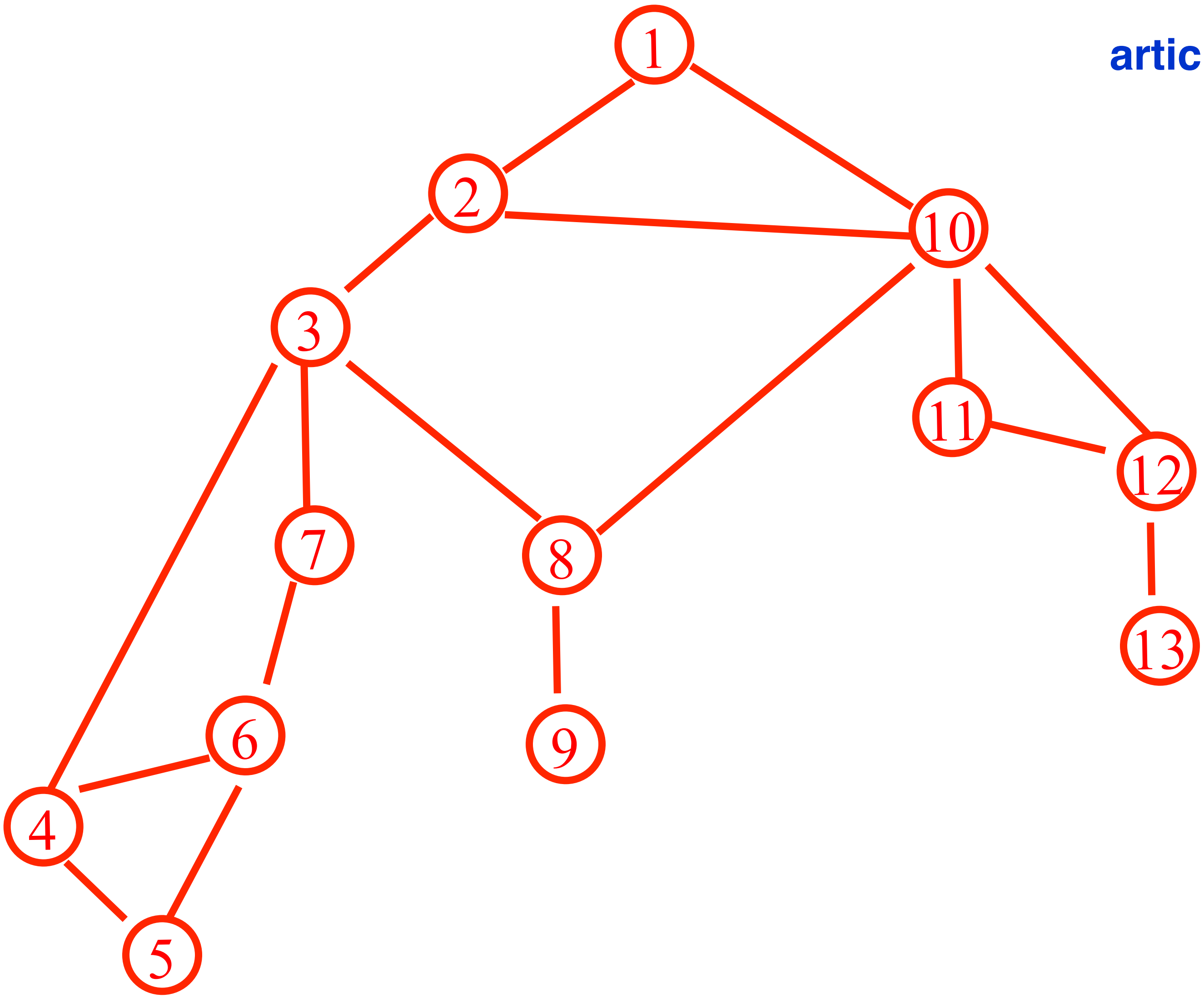
Identifying key proteins on the anthrax predicted network



Articulation point proteins

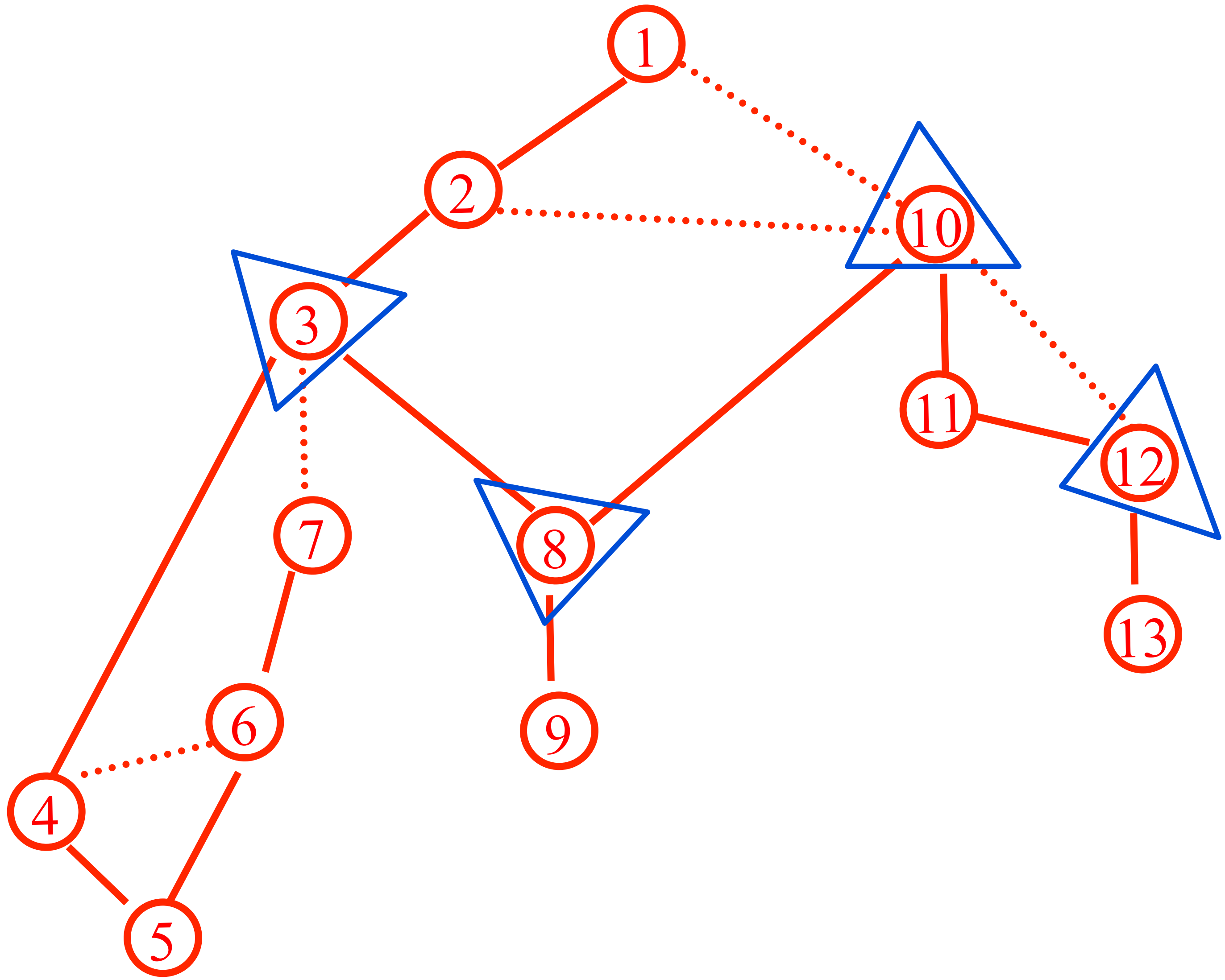
Ram Samudrala/Jason McDermott

Articulation Points



articulation point
iff its removal
disconnects
the graph

Articulation Points



Simple Case: Artic. Pts in a tree

Leaves – never articulation points

Internal nodes – always articulation points

Root – articulation point if and only if two or more children

Non-tree: extra edges remove some articulation points (which ones?)

Articulation Points from DFS

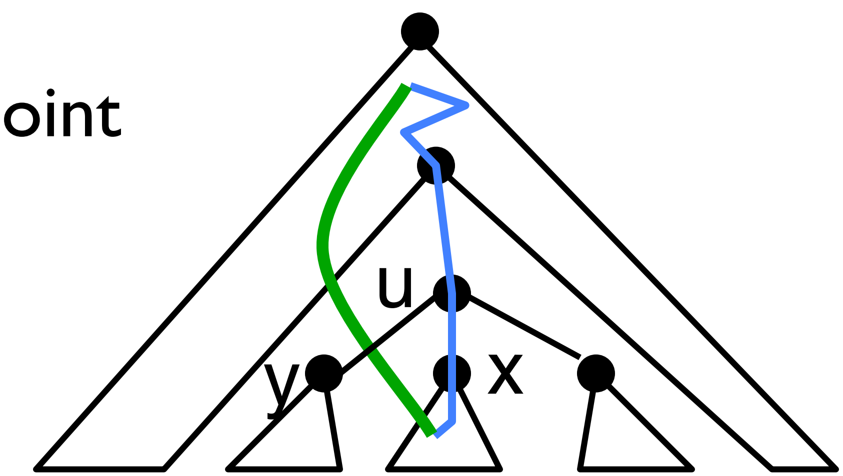
Root node is an articulation point iff it has more than one child

Leaf is never an articulation point

Non-leaf, non-root node u is an articulation point

⇕

∃ some child y of u s.t. no non-tree edge goes above u from y or below



If u's removal does NOT separate x, there must be an exit from x's subtree. How? Via back edge.

Articulation Points: the "LOW" function

trivial

Definition: LOW(v) is the lowest dfs# of any vertex that is either in the dfs subtree rooted at v (including v itself) or directly connected to a vertex in that subtree by a back edge.

critical

Key idea 1: if some child x of v has LOW(x) ≥ dfs#(v) then v is an articulation point (excl. root)

Key idea 2: LOW(v) =

min ({dfs#(v)} ∪ {LOW(w) | w a child of v } ∪ { dfs#(x) | {v,x} is a back edge from v })

DFS(v) for Finding Articulation Points

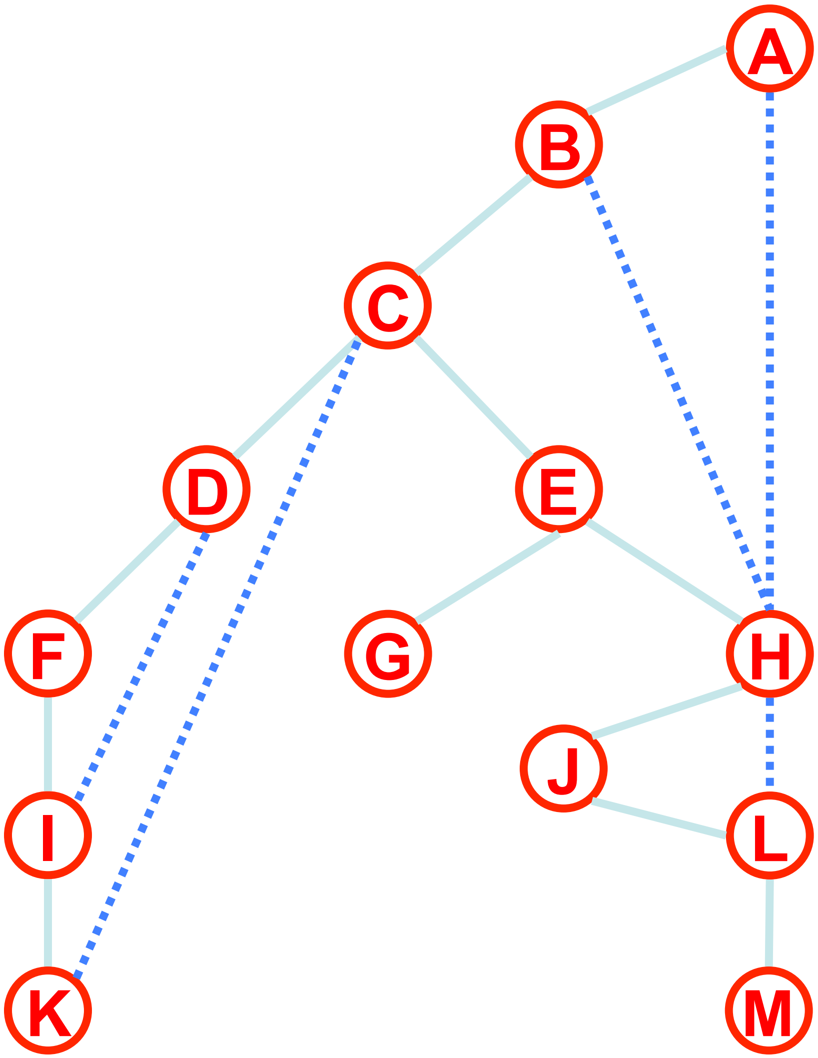
Global initialization: dfscounter = 0; v.dfs# = -1 for all v.

```
DFS(v)
 v.dfs# = dfscounter++
 v.low = v.dfs#                    // initialization
 for each edge {v,x}
      if (x.dfs# == -1)            // x is undiscovered
         DFS(x)
         v.low = min(v.low, x.low)
         if (x.low >= v.dfs#)
            print "v is art. pt., separating x"
      else if (x is not v's parent)
         v.low = min(v.low, x.dfs#)
```

Except for root. Why?

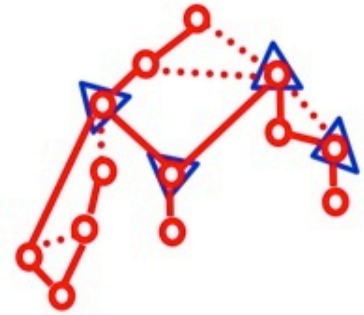
Equiv: "if({v,x} is a back edge)" Why?

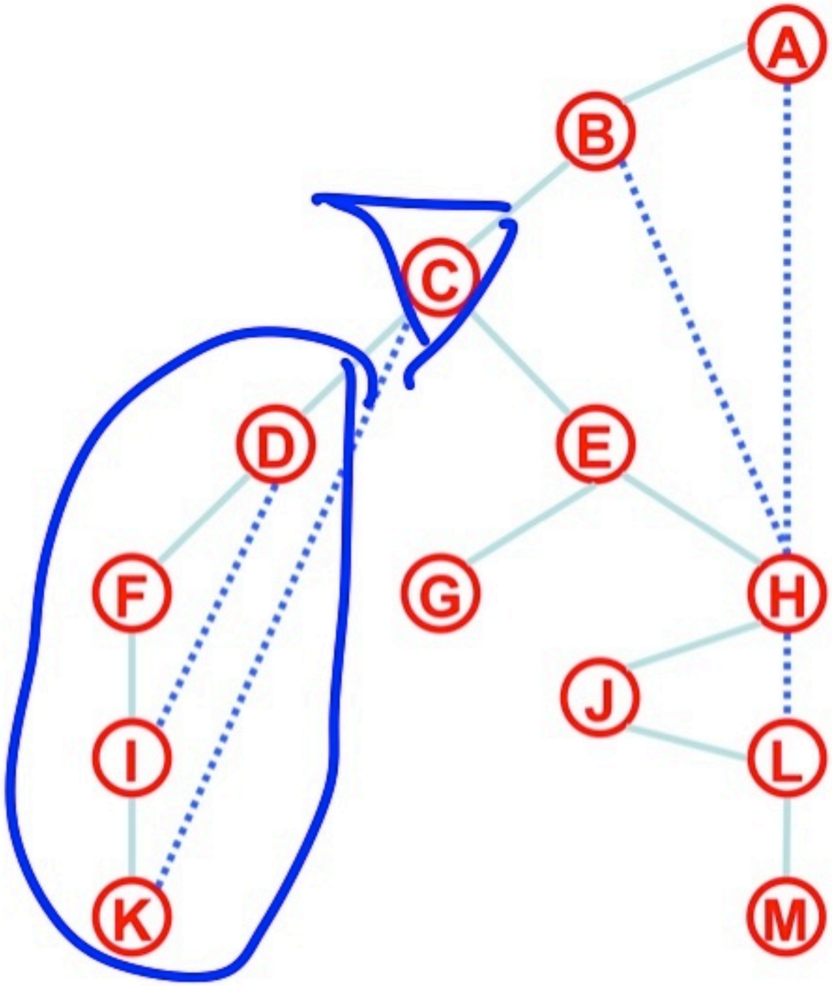
Articulation Point



Vertex	DFS #	Low
A		
B		
C		
D		
E		
F		
G		
H		
I		
J		
K		
L		
M		

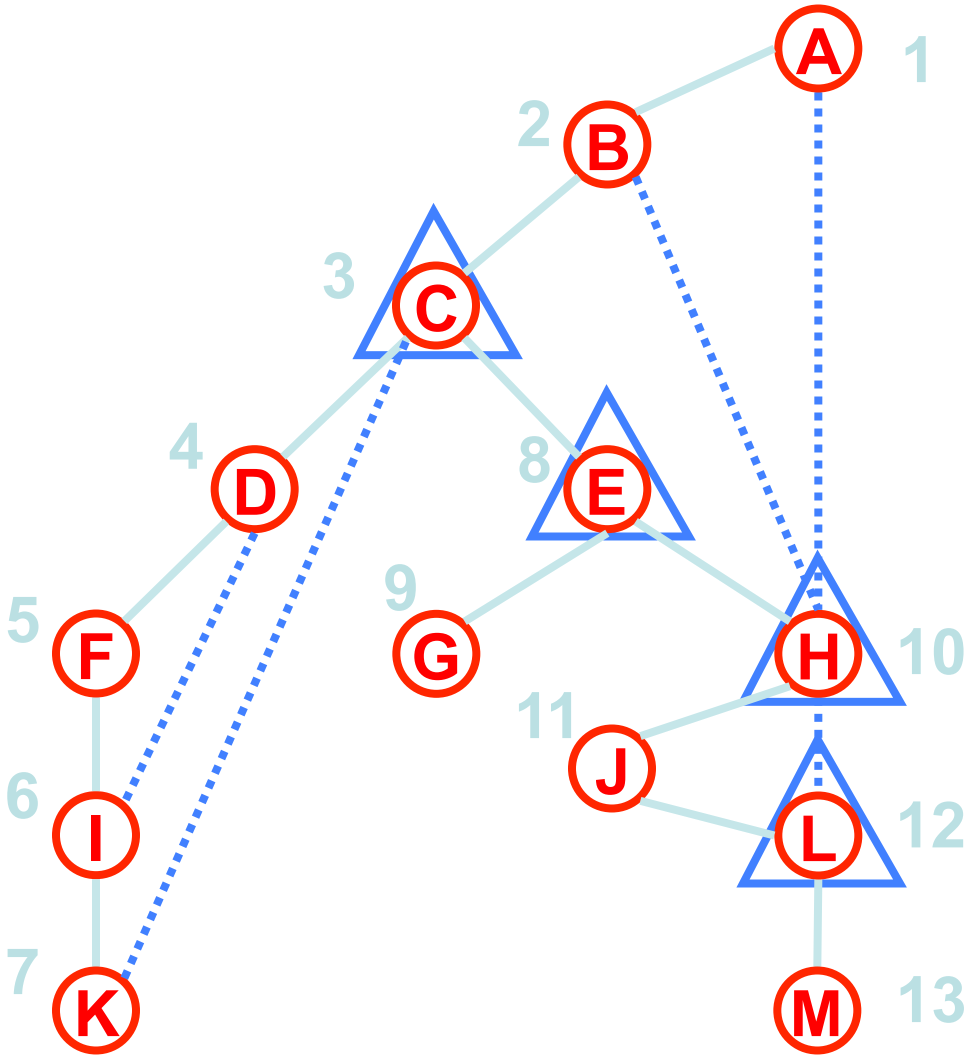
Articulation Point





Vertex	DFS #	Low
A	1	1
B	2	2
C	3	3
D	4	3
E		
F	5	3
G		
H		
I	6	~~4~~ 3
J		
K	7	3
L		
M		

Articulation Points



Vertex	DFS #	Low
A	1	1
B	2	1
C	3	1
D	4	3
E	8	1
F	5	3
G	9	9
H	10	1
I	6	3
J	11	10
K	7	3
L	12	10
M	13	13

Summary

Graphs – abstract relationships among pairs of objects

Terminology – node/vertex/vertices, edges, paths, multi-edges, self-loops, connected

Representation – edge list, adjacency matrix

Nodes vs Edges – $m = O(n^2)$, often less

BFS – Layers, queue, shortest paths, all edges go to same or adjacent layer

DFS – recursion/stack; all edges ancestor/descendant

Algorithm – articulation points