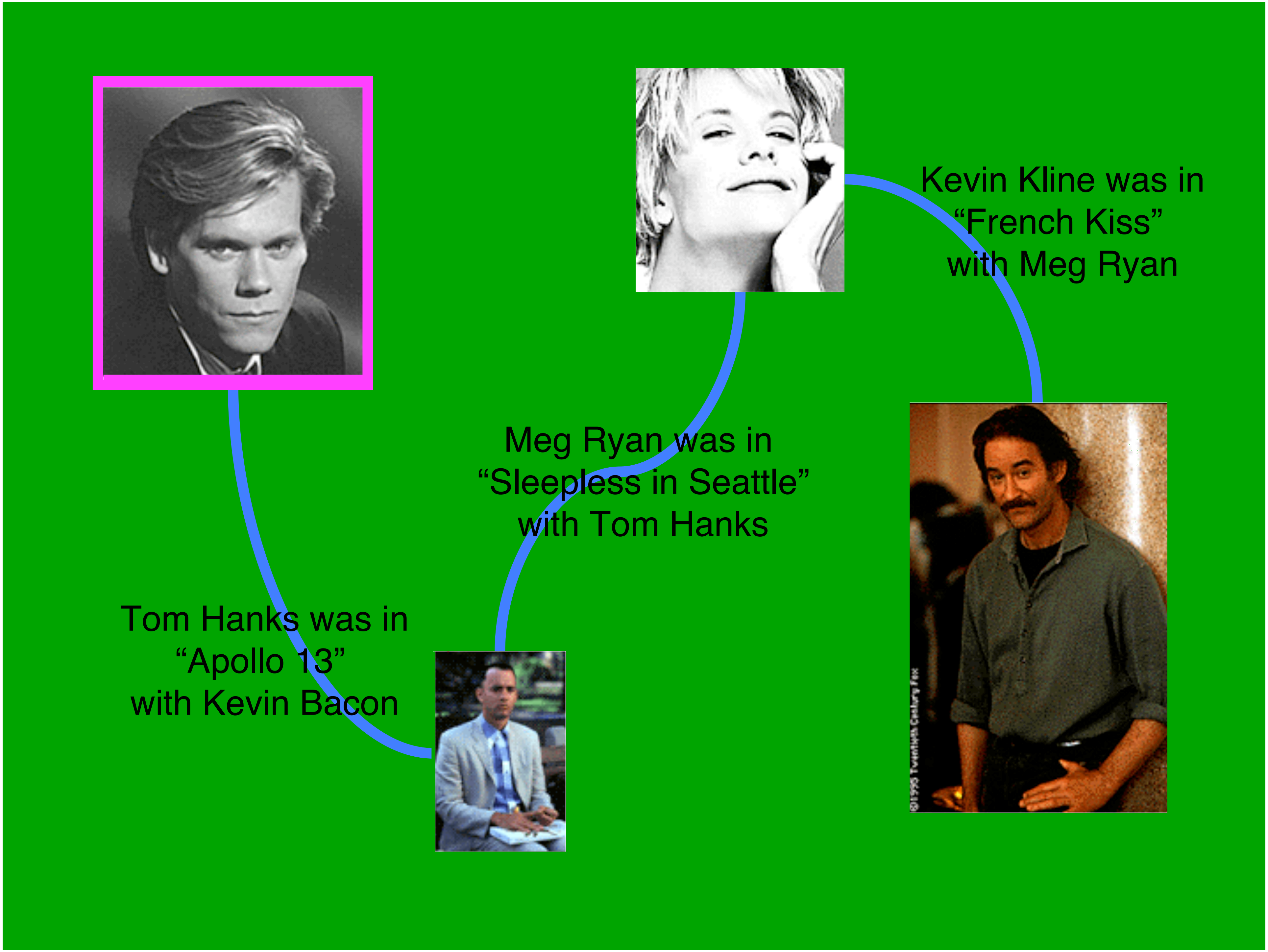
CSE 417: Algorithms and Computational Complexity

Winter 2005

Graphs and Graph Algorithms

Larry Ruzzo


Kevin Kline was in
“French Kiss”
with Meg Ryan
Meg Ryan was in
“Sleepless in Seattle”
with Tom Hanks
Tom Hanks was in
“Apollo 13”
with Kevin Bacon

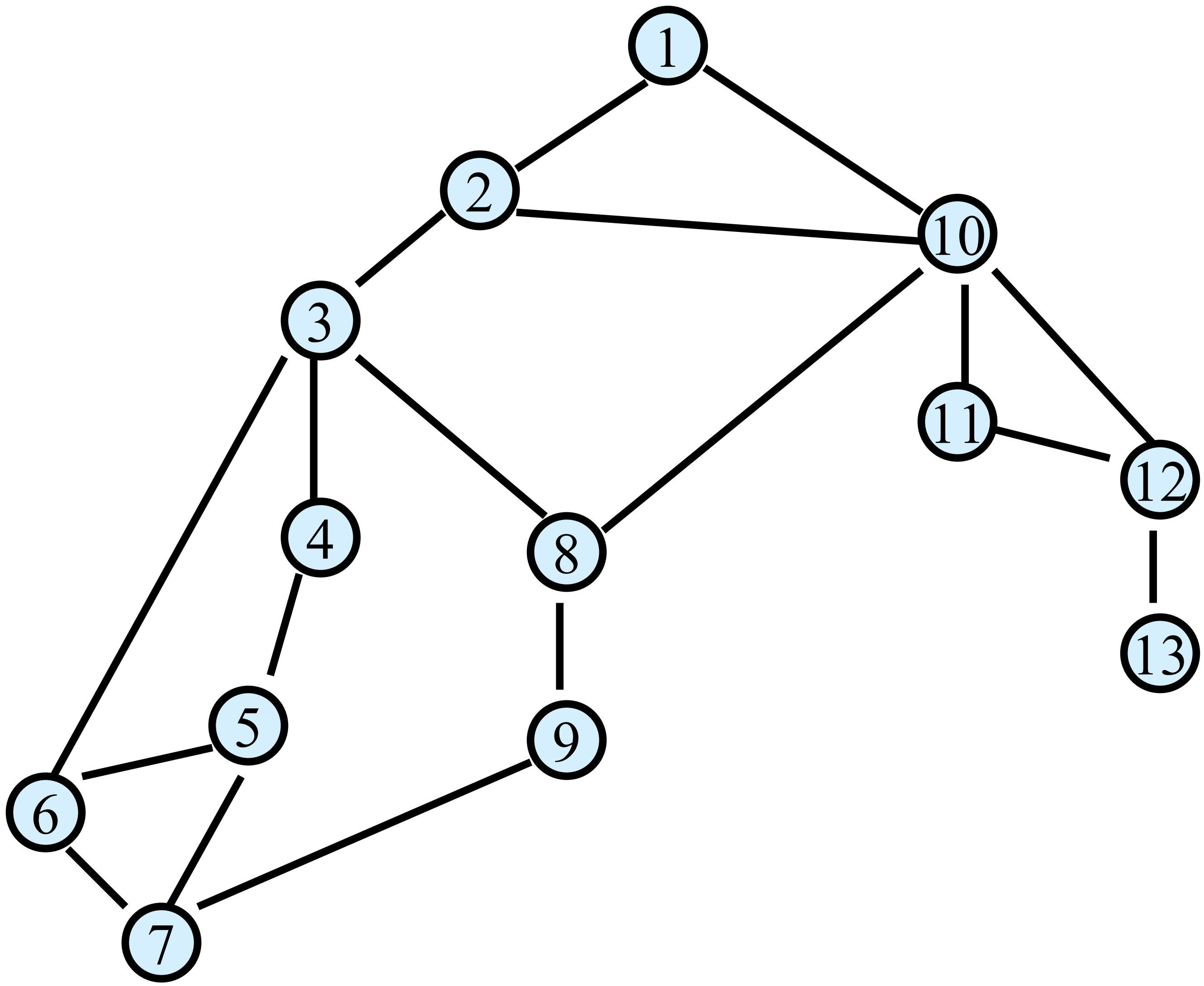
Objects & Relationships

- The Kevin Bacon Game:
  - Actors
  - Two are related if they’ve been in a movie together
- Exam Scheduling:
  - Classes
  - Two are related if they have students in common
- Traveling Salesperson Problem:
  - Cities
  - Two are related if can travel directly between them

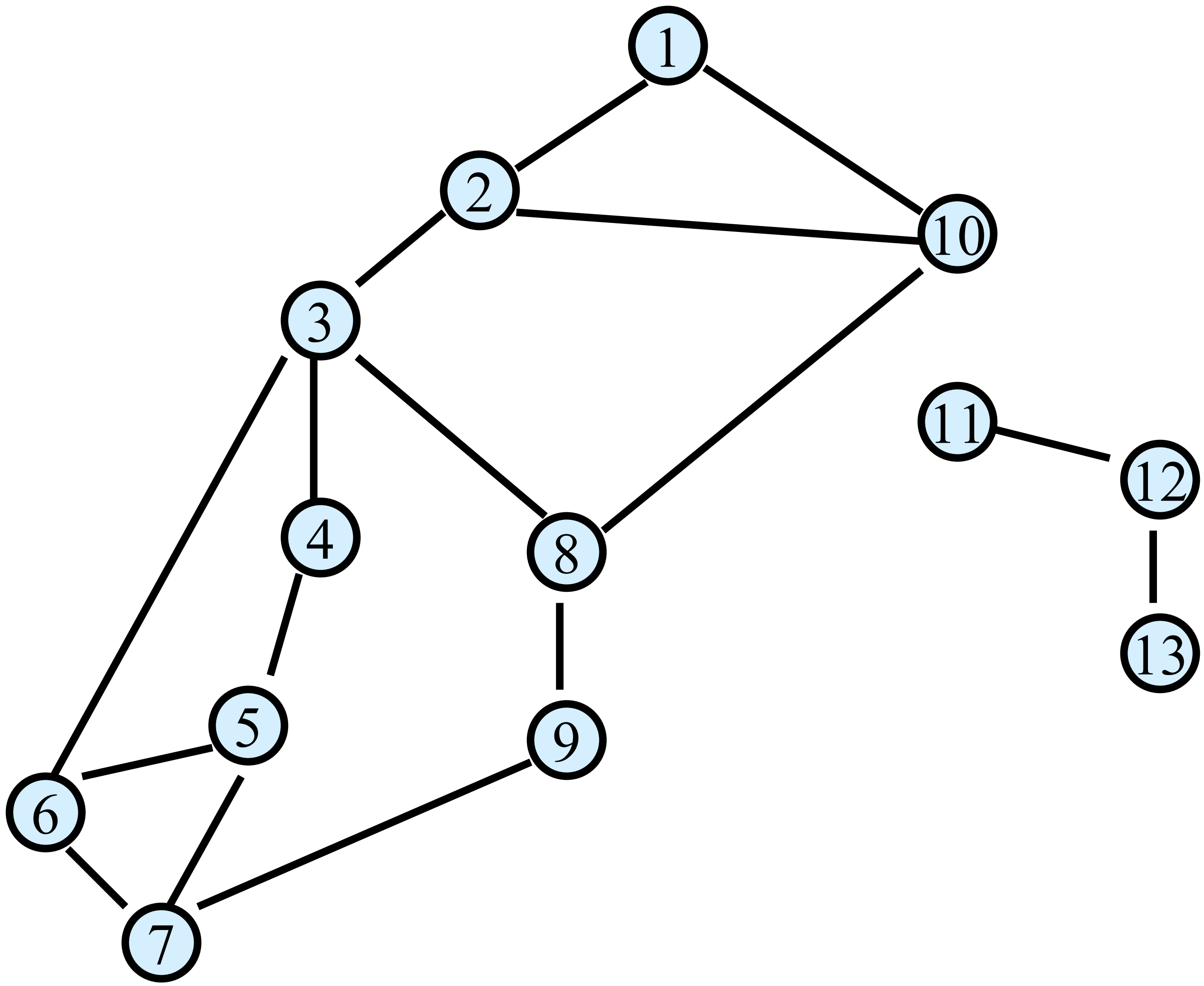
Graphs

- 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

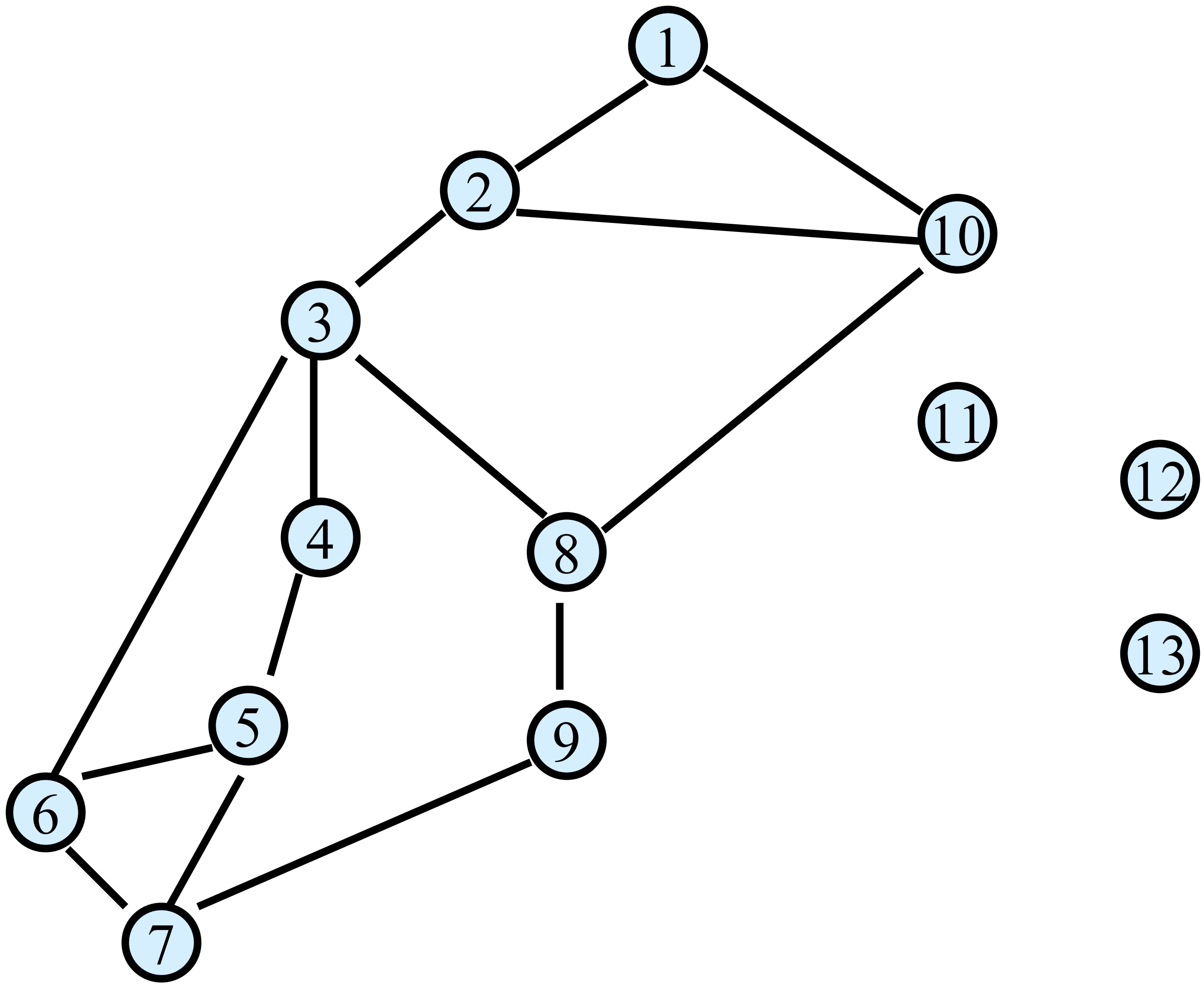
Undirected Graph G = (V,E)



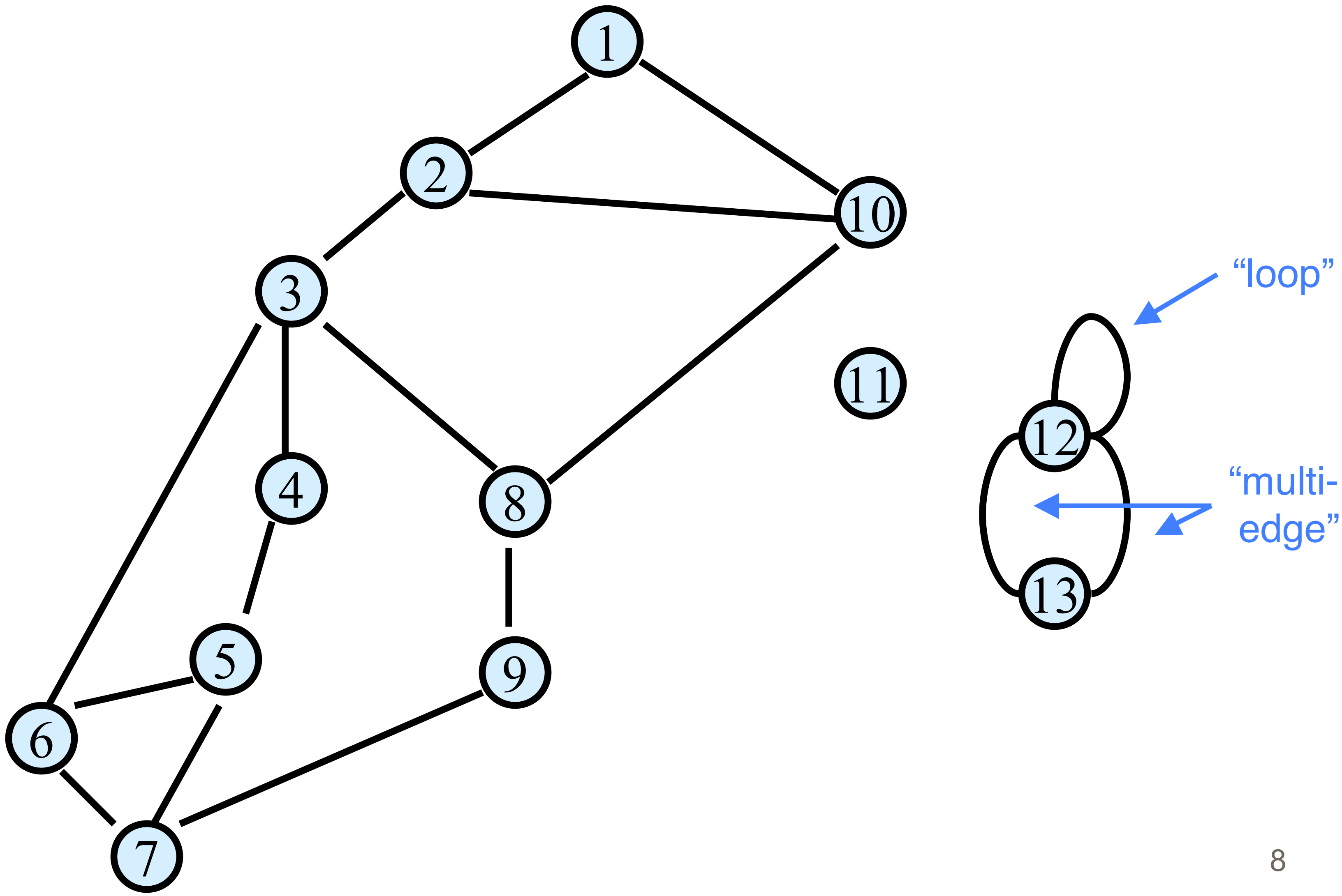
Undirected Graph G = (V,E)



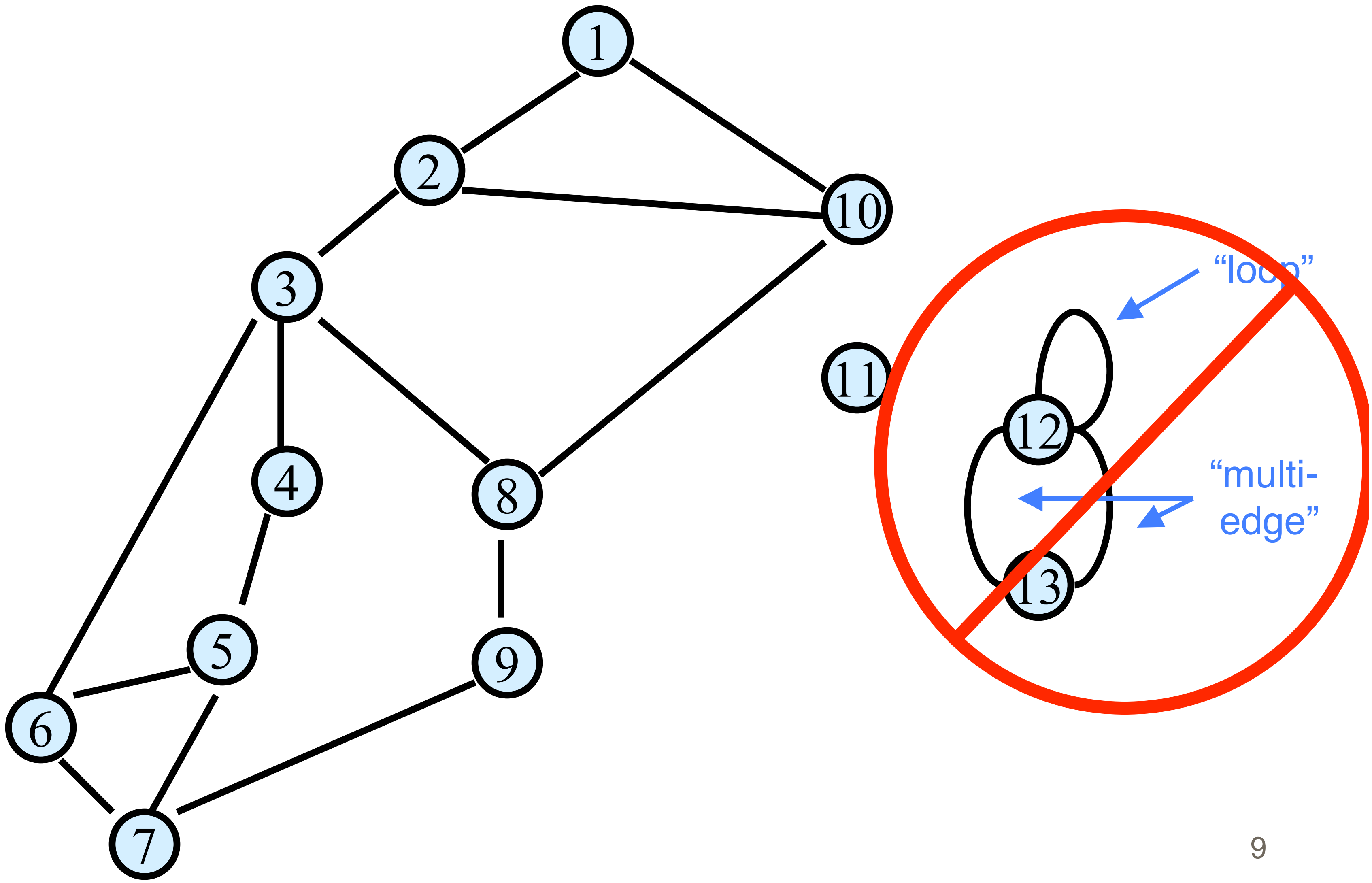
Undirected Graph $G = (V,E)$



Undirected Graph G = (V,E)

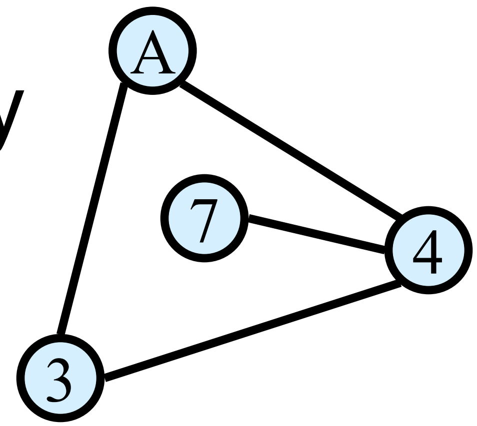


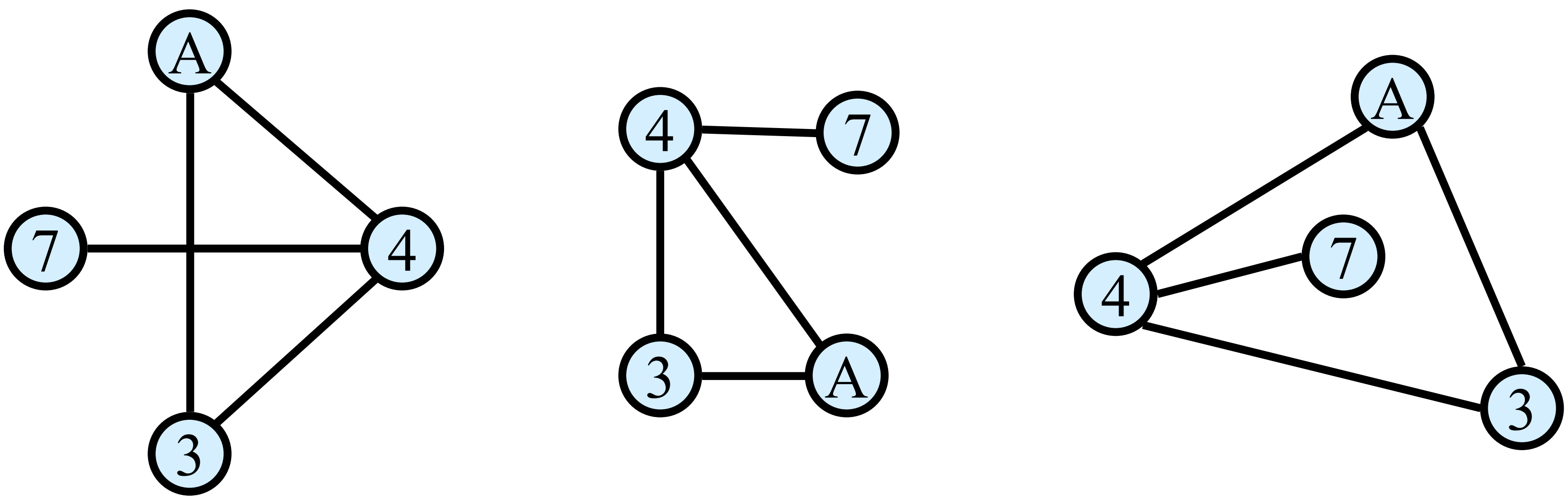
Undirected Graph G = (V,E)



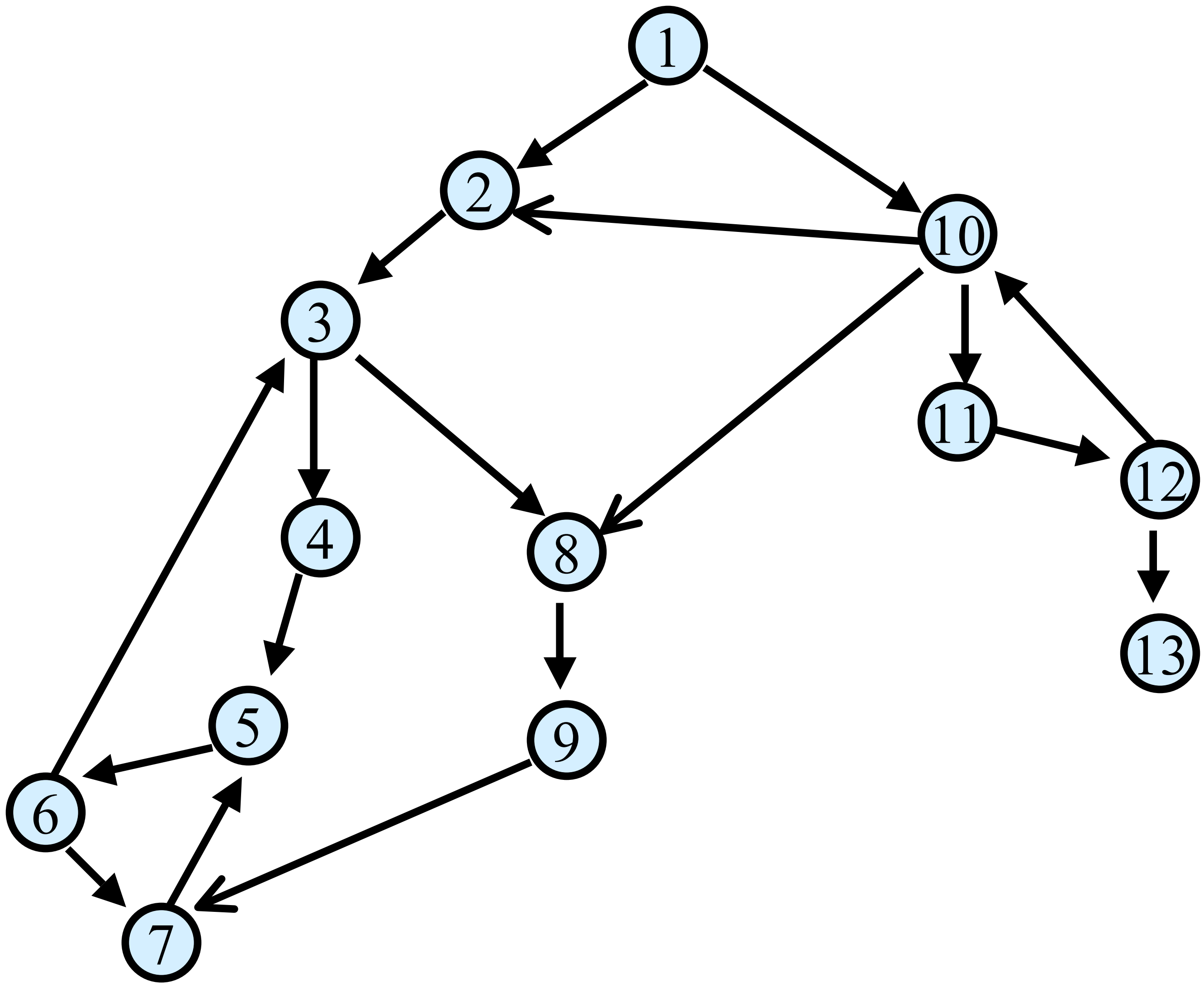
Graphs don't live in Flatland

- Geometrical drawing is mentally convenient, but mathematically irrelevant: 4 drawings, 1 graph.

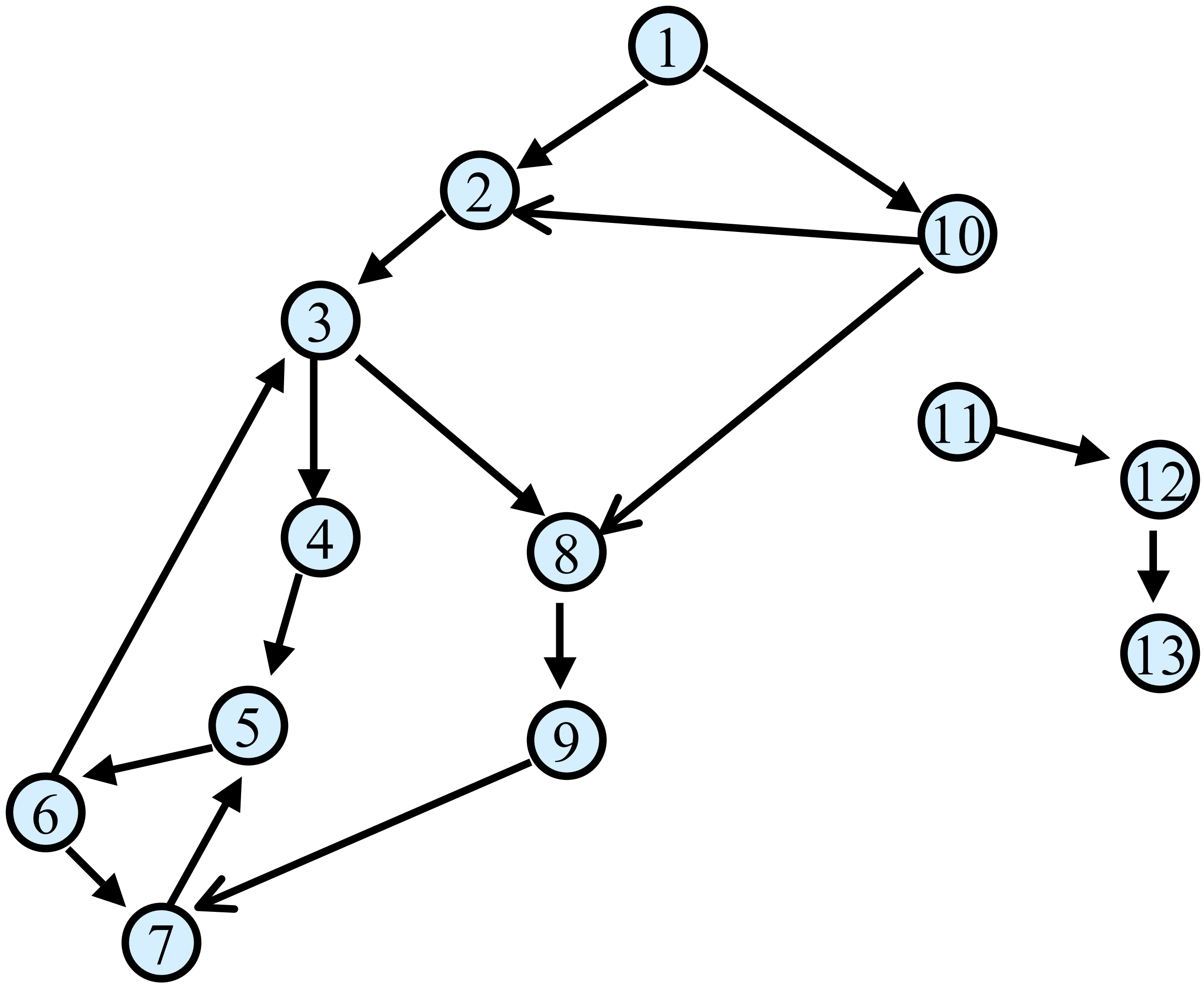




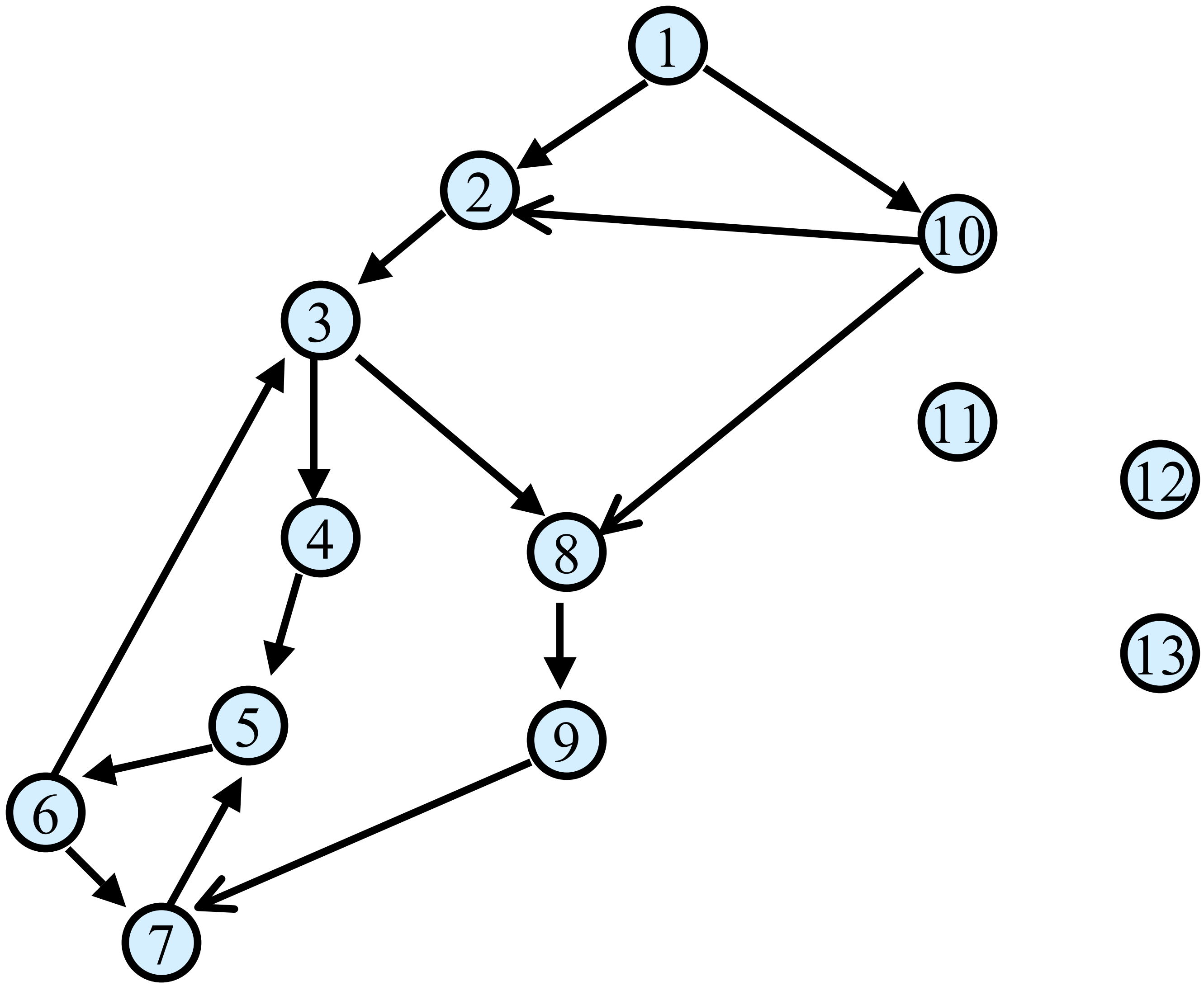
Directed Graph G = (V,E)



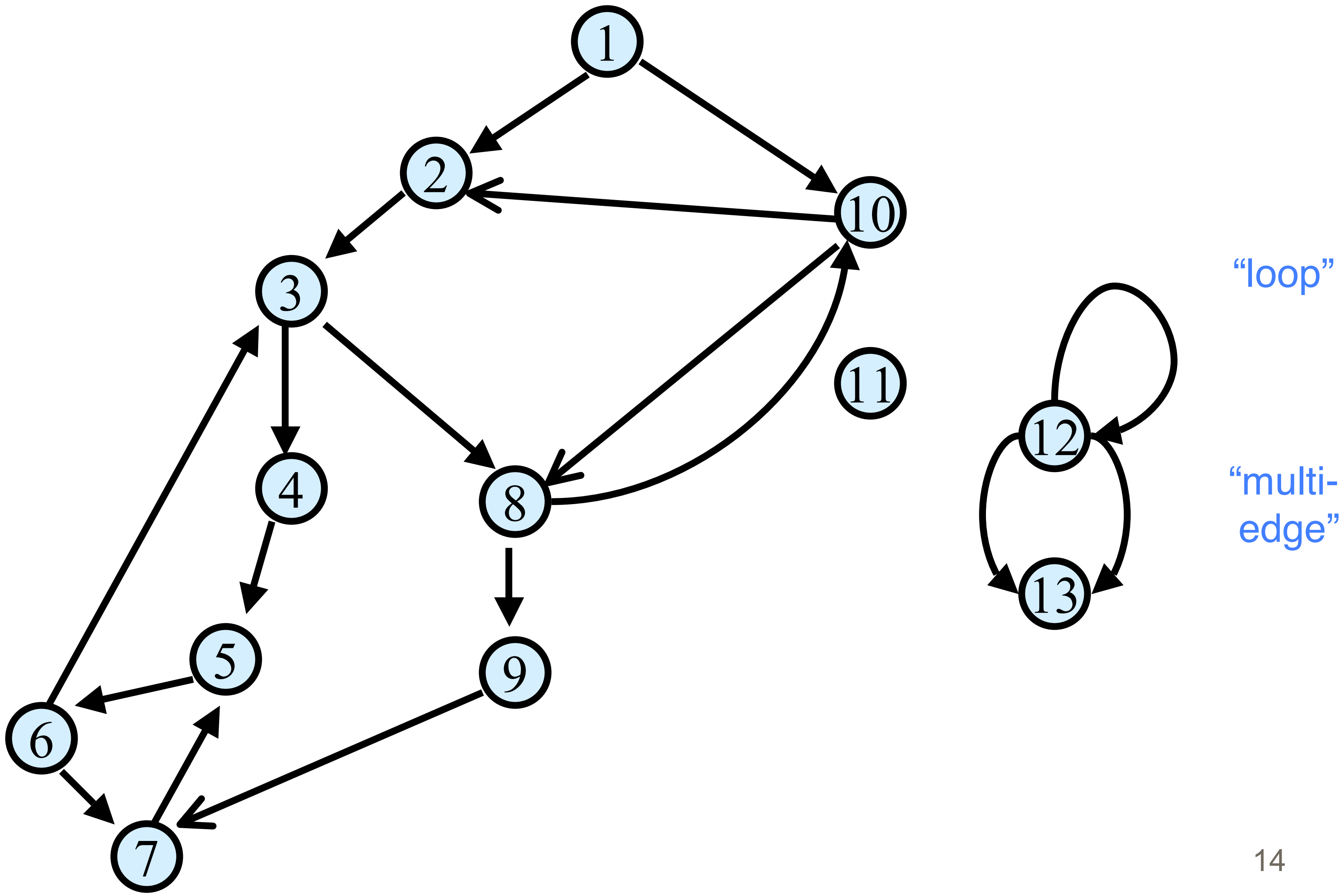
Directed Graph G = (V,E)



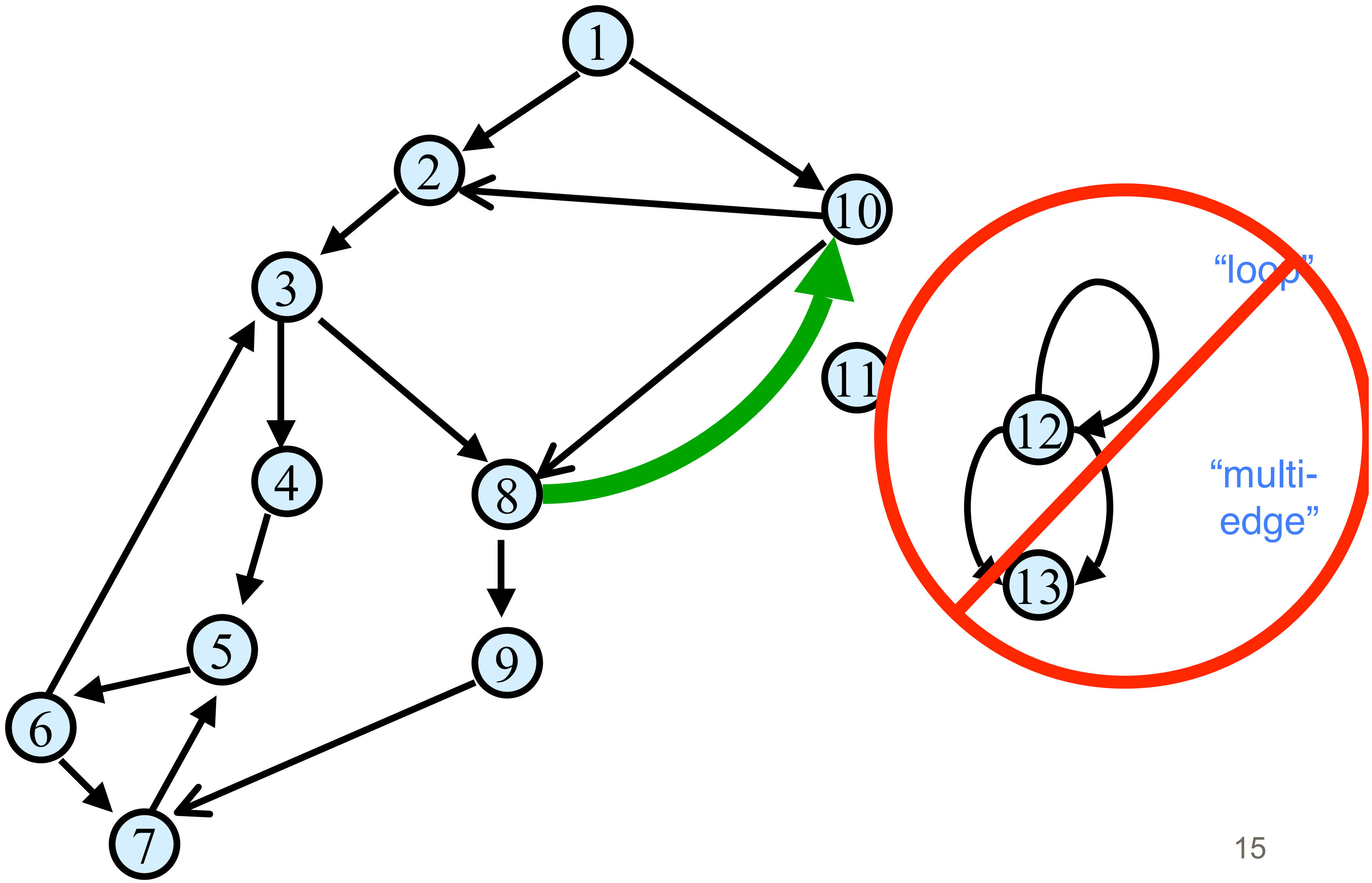
Directed Graph G = (V,E)



Directed Graph G = (V,E)

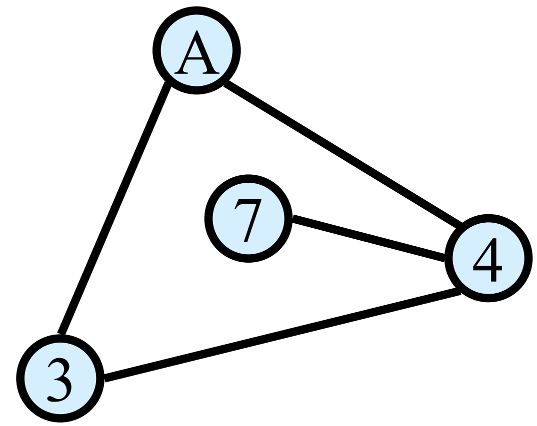


Directed Graph G = (V,E)



Specifying undirected graphs as input

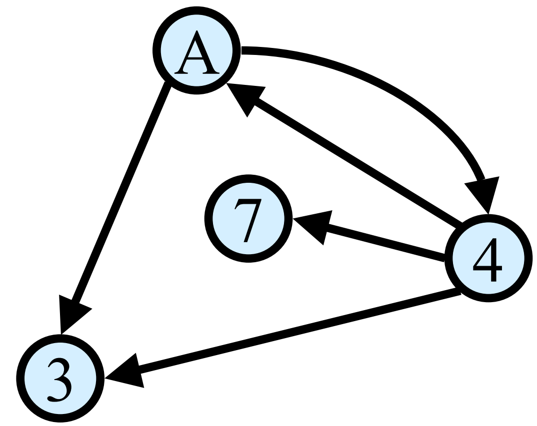
- What are the vertices?
  - Explicitly list them: {"A", "7", "3", "4"}
- What are the edges?
  - Either, set of edges {{A,3}, {7,4}, {4,3}, {4,A}}
  - Or, (symmetric) adjacency matrix:



	A	7	3	4
A	0	0	1	1
7	0	0	0	1
3	1	0	0	1
4	1	1	1	0

Specifying directed graphs as input

- What are the vertices
  - Explicitly list them: {“A”, “7”, “3”, “4”}
- What are the edges
  - Either, set of directed edges: {(A,4), (4,7), (4,3), (4,A), (A,3)}
  - Or, (nonsymmetric) adjacency matrix:



	A	7	3	4
A	0	0	1	1
7	0	0	0	0
3	0	0	0	0
4	1	1	1	0

Vertices vs # Edges

- Let G be an undirected graph with n vertices and m edges
- How are n and m related?
- Since
  - every edge connects two *different* vertices (no loops), and
  - no two edges connect the *same* two vertices (no multi-edges),

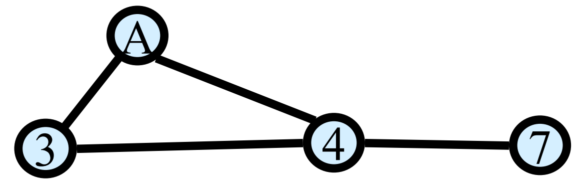
it must be true that: $\boxed{0 \leq m \leq n(n-1)/2 = O(n^2)}$

More Cool Graph Lingo

- A graph is called *sparse* if $m \ll n^2$, otherwise it is *dense*
  - Boundary is somewhat fuzzy; $O(n)$ edges is certainly sparse, $\Omega(n^2)$ edges is dense.
- Sparse graphs are common in practice
  - E.g., all planar graphs are sparse
- Q: which is a better run time, $O(n+m)$ or $O(n^2)$?

A: $O(n+m) = O(n^2)$, but $n+m$ usually way better!

Representing Graph $G = (V,E)$
n vertices, m edges



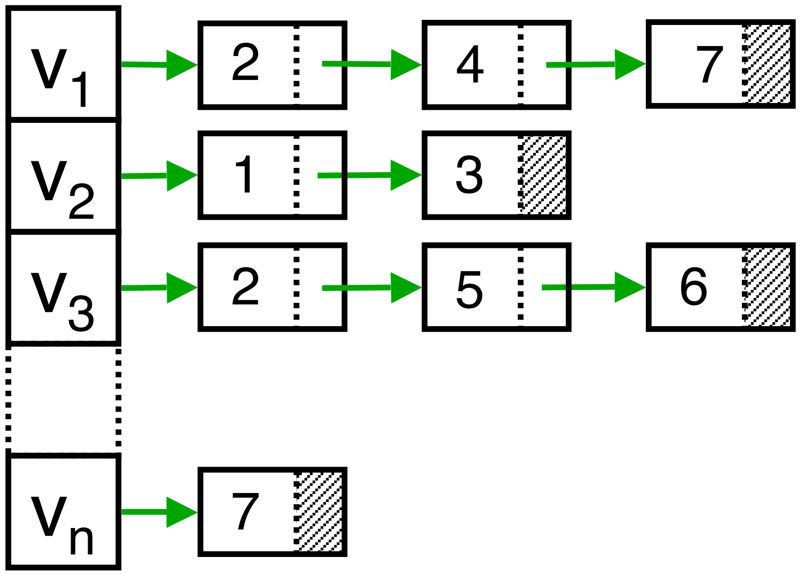
	A	7	3	4
A	0	0	1	1
7	0	0	0	1
3	1	0	0	1
4	1	1	1	0

- Vertex set $V = \{v_1, \ldots, v_n\}$
- Adjacency Matrix A
  - $A[i,j] = 1$ iff $(v_i,v_j) \in E$
  - Space is n^2 bits
- Advantages:
  - $O(1)$ test for presence or absence of edges.
  - compact if in packed binary form for large m
- Disadvantages: inefficient for sparse graphs

$m << n^2$

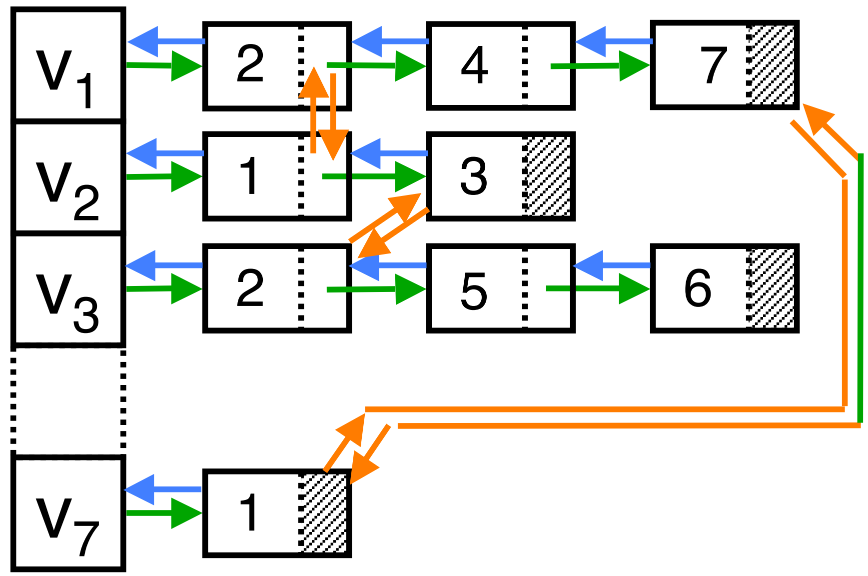
Representing Graph G=(V,E) n vertices, m edges

- Adjacency List:
  - O(n+m) words
- Advantages:
  - Compact for sparse graphs
  - Easily see all edges
- Disadvantages
  - More complex data structure
  - no O(1) edge test



Representing Graph $G=(V,E)$
n vertices, m edges

- Adjacency List:
  - $O(n+m)$ words



- Back- and cross pointers more work to build, but allow easier traversal and deletion of edges, *if needed,* (don't bother if not)

Graph Traversal

- Learn the basic structure of a graph
- "Walk," via edges, from a fixed starting vertex v to all vertices reachable from v

- Three states of vertices
  - **undiscovered**
  - **discovered**
  - **fully-explored**

Breadth-First Search

- Completely explore the vertices in order of their distance from v

- Naturally implemented using a queue

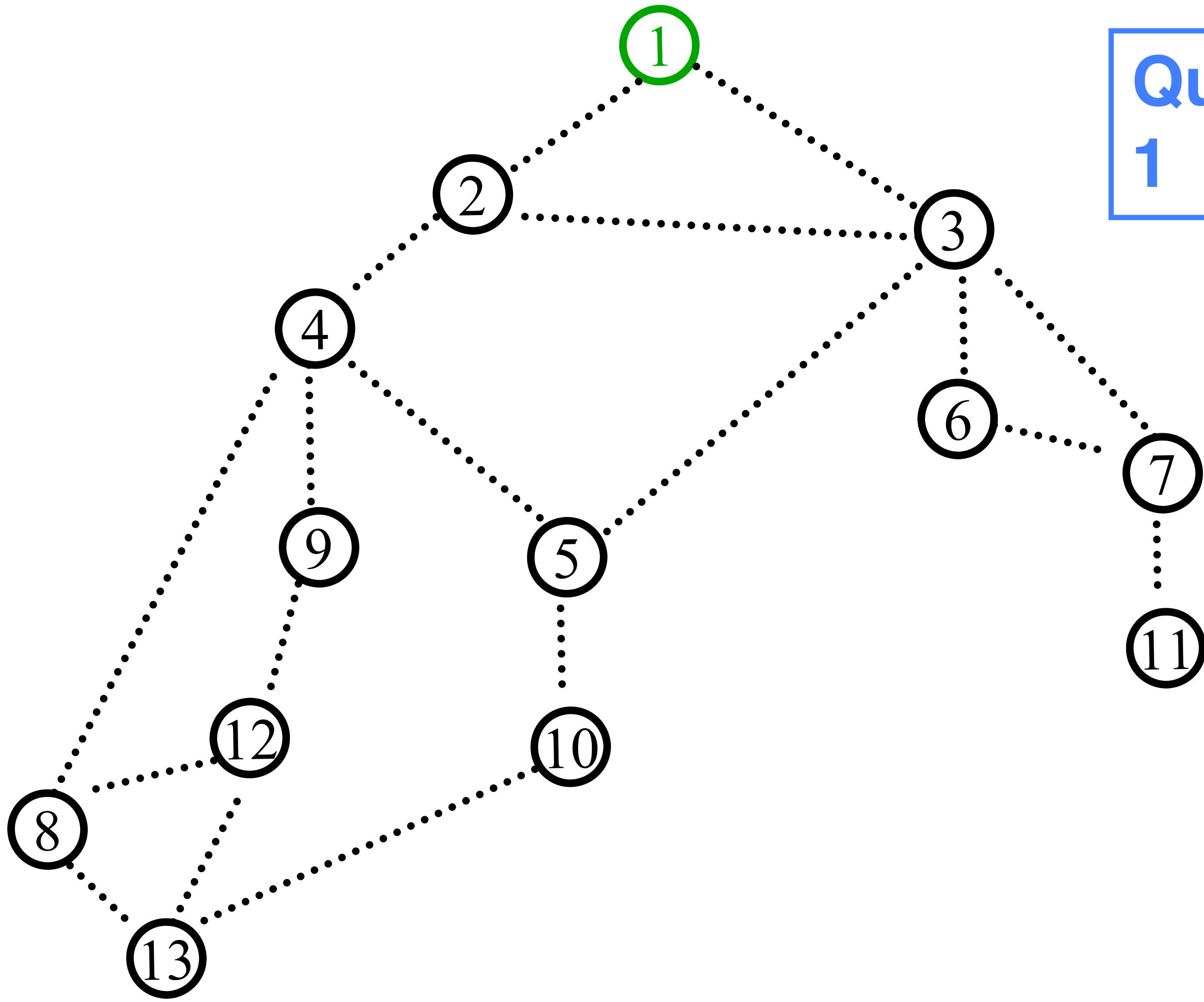
BFS(v)

Global initialization: mark all vertices "undiscovered"

```
BFS(v)
    mark  v "discovered"
    queue = v
    while queue not empty
        u = remove_first(queue)
        for each edge {u,x}
            if (x is undiscovered)
                mark x discovered
                append x on queue
        mark u completed
```

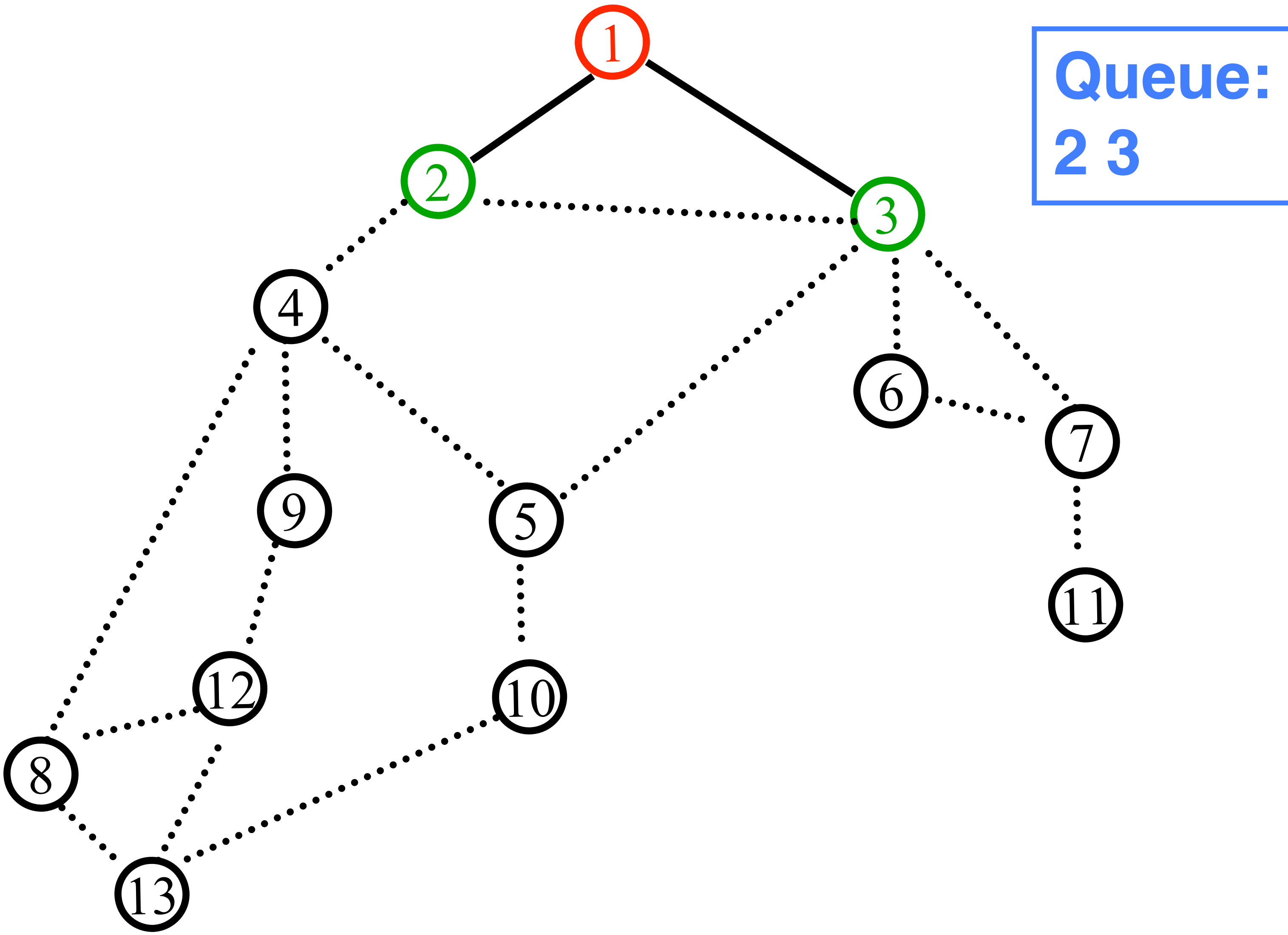
Exercise: modify code to number vertices & compute level numbers

BFS(v)

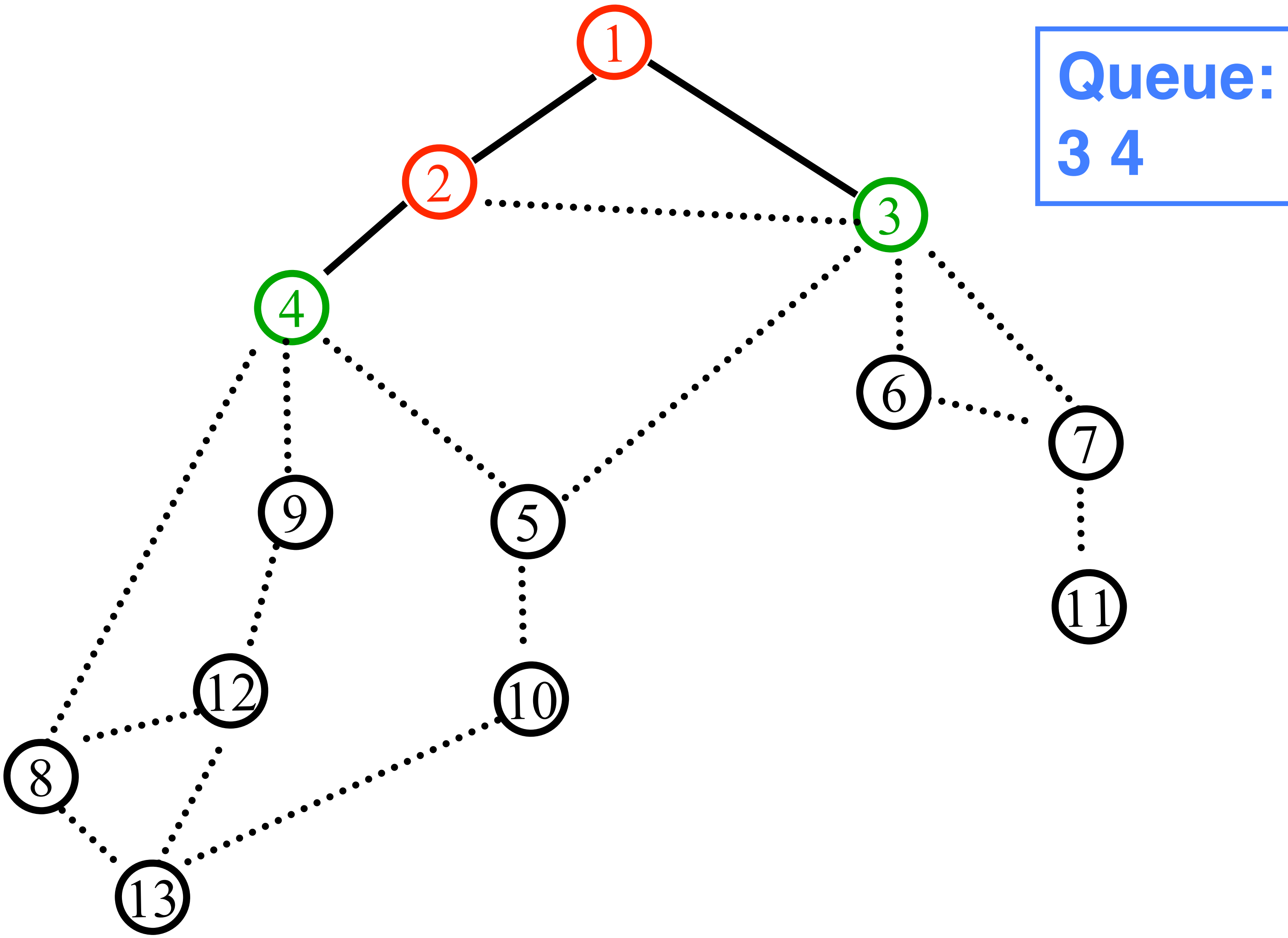


Queue:
1

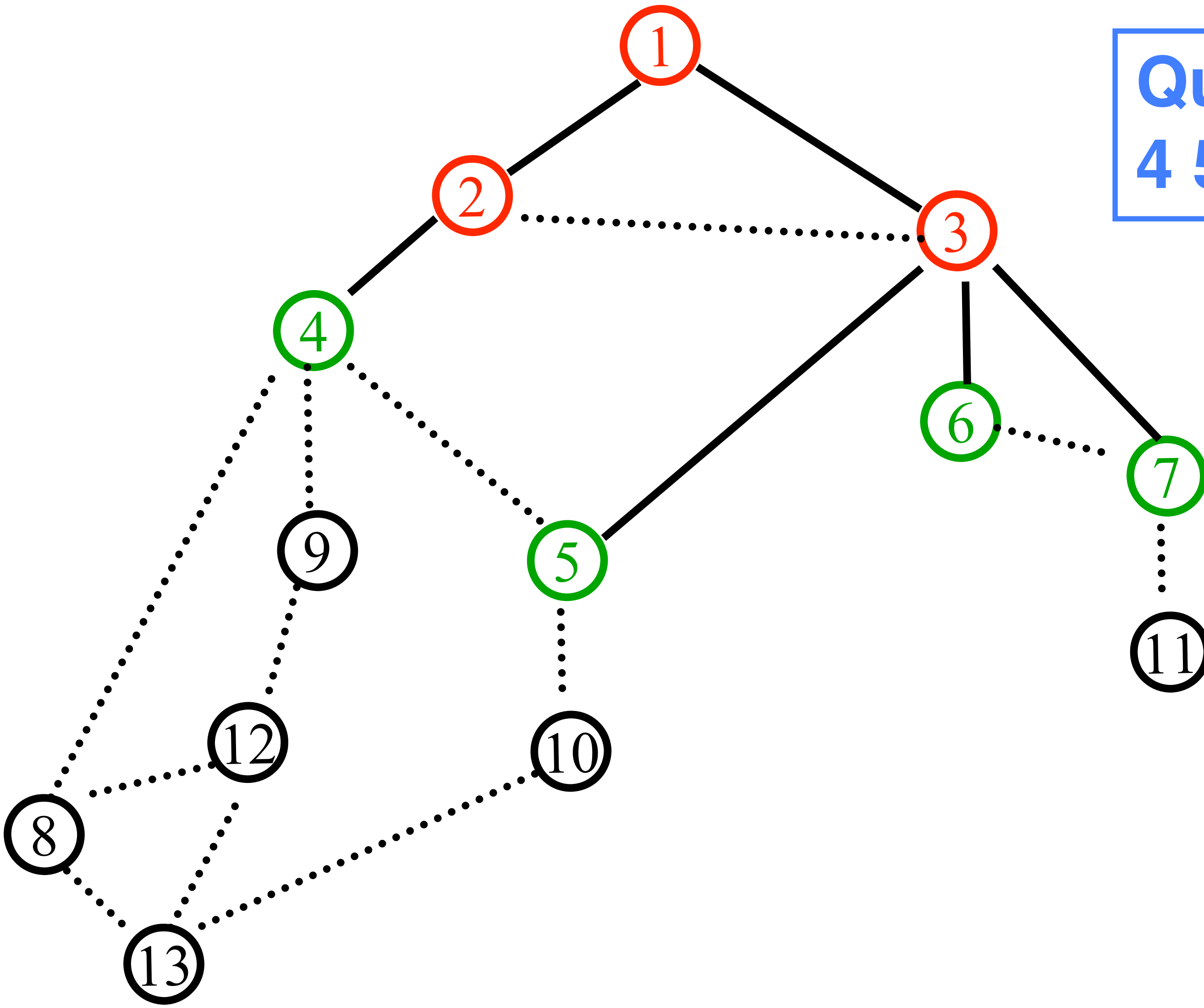
BFS(v)



BFS(v)

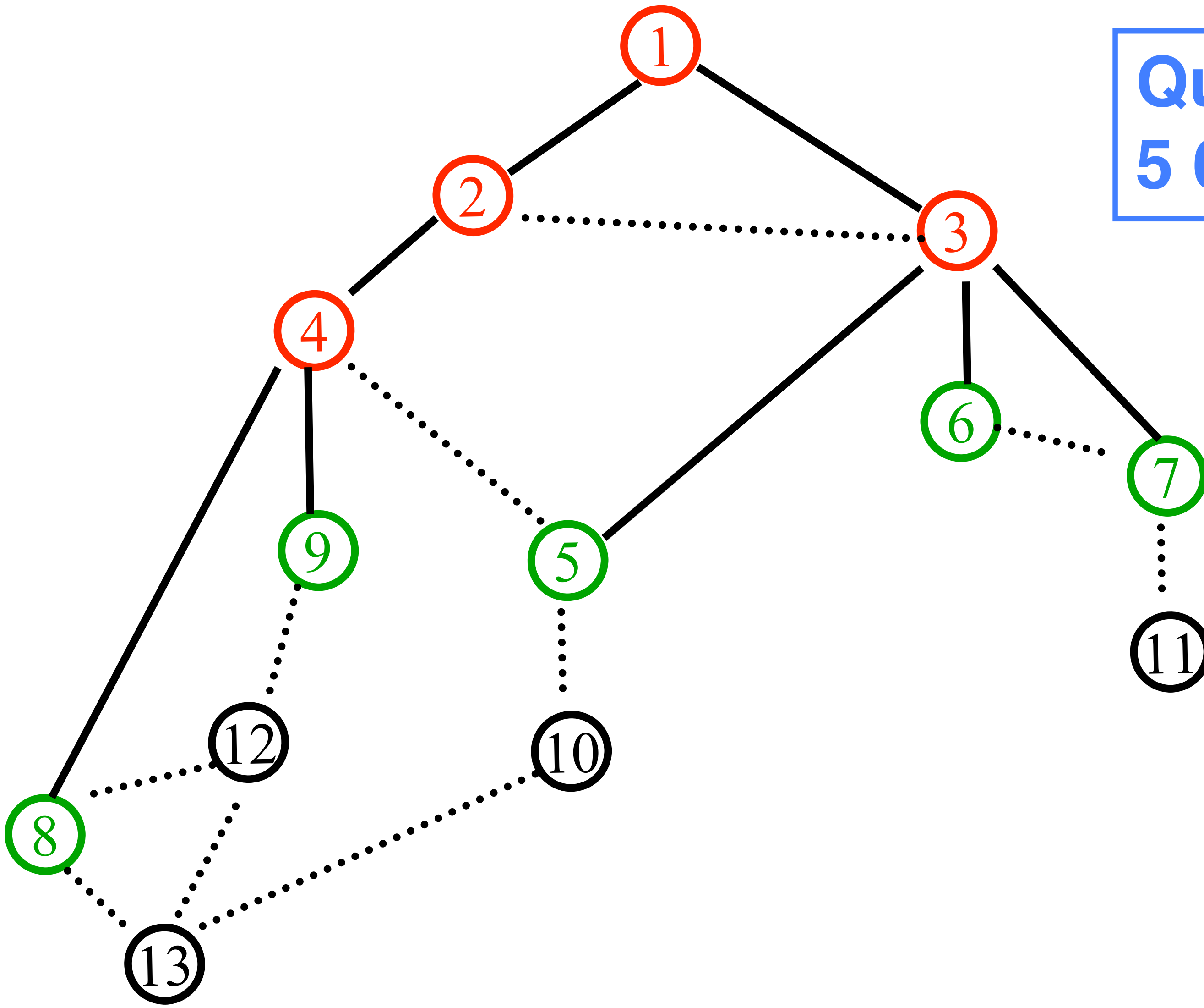


BFS(v)



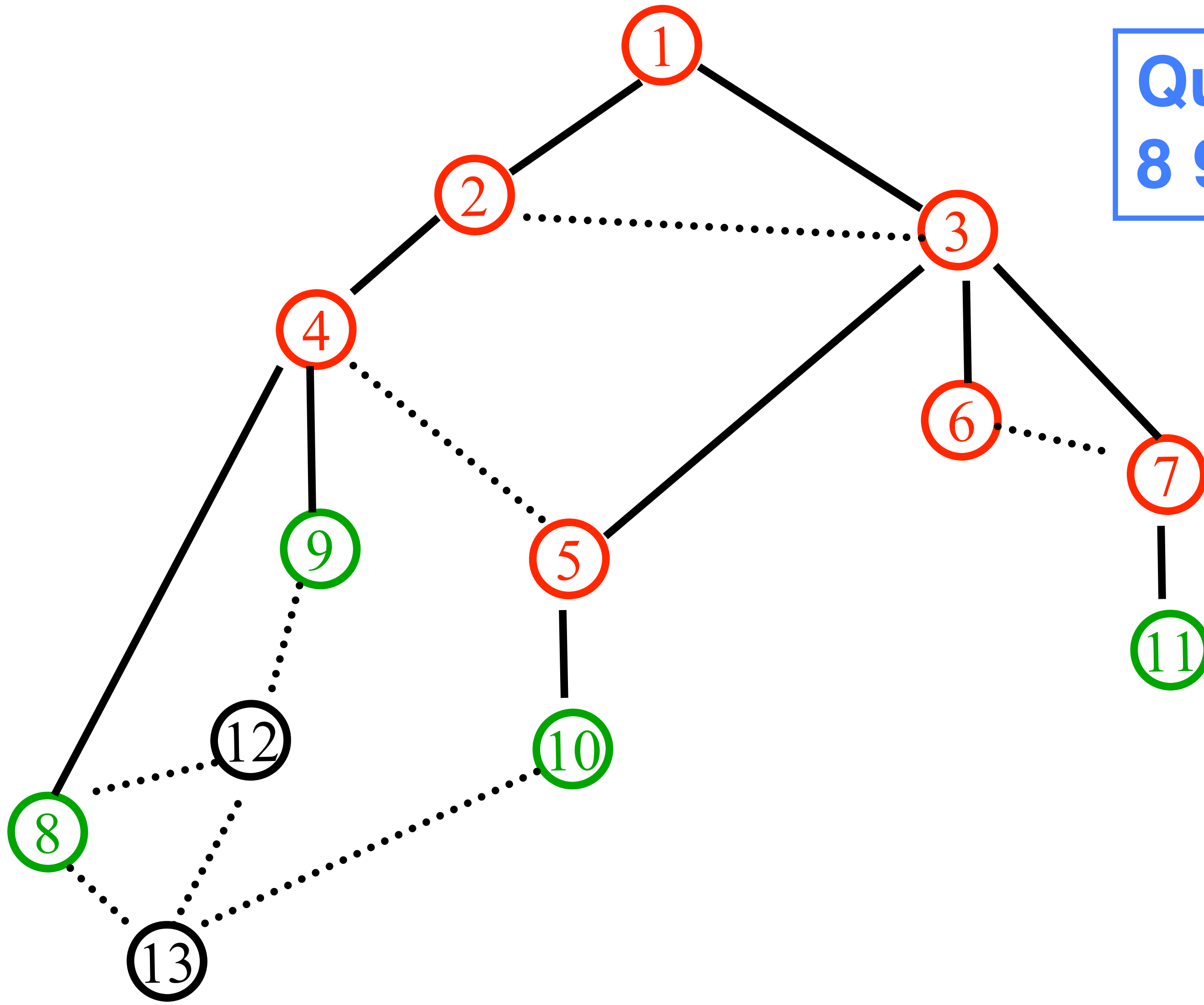
Queue:
4 5 6 7

BFS(v)



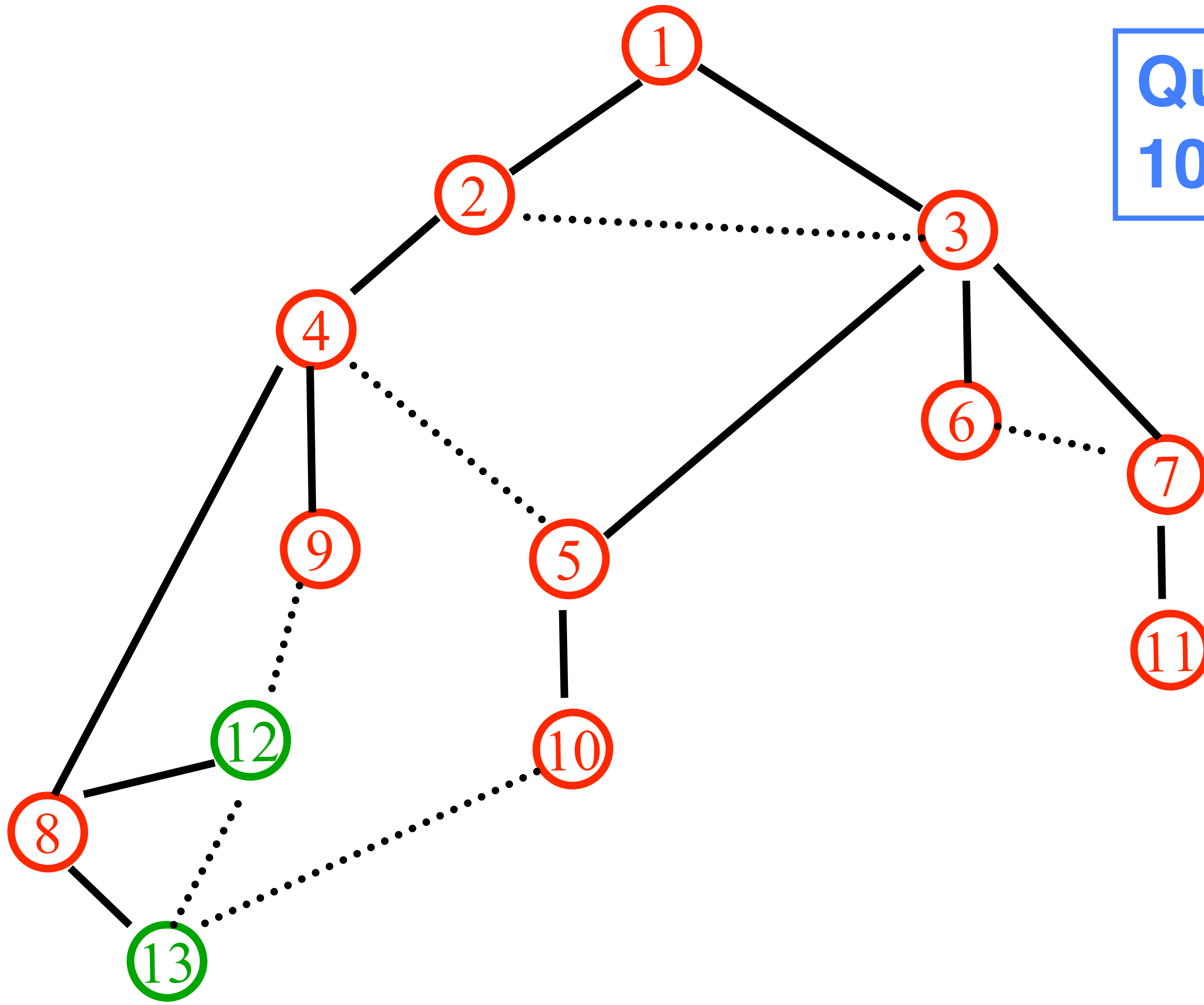
Queue:
5 6 7 8 9

BFS(v)



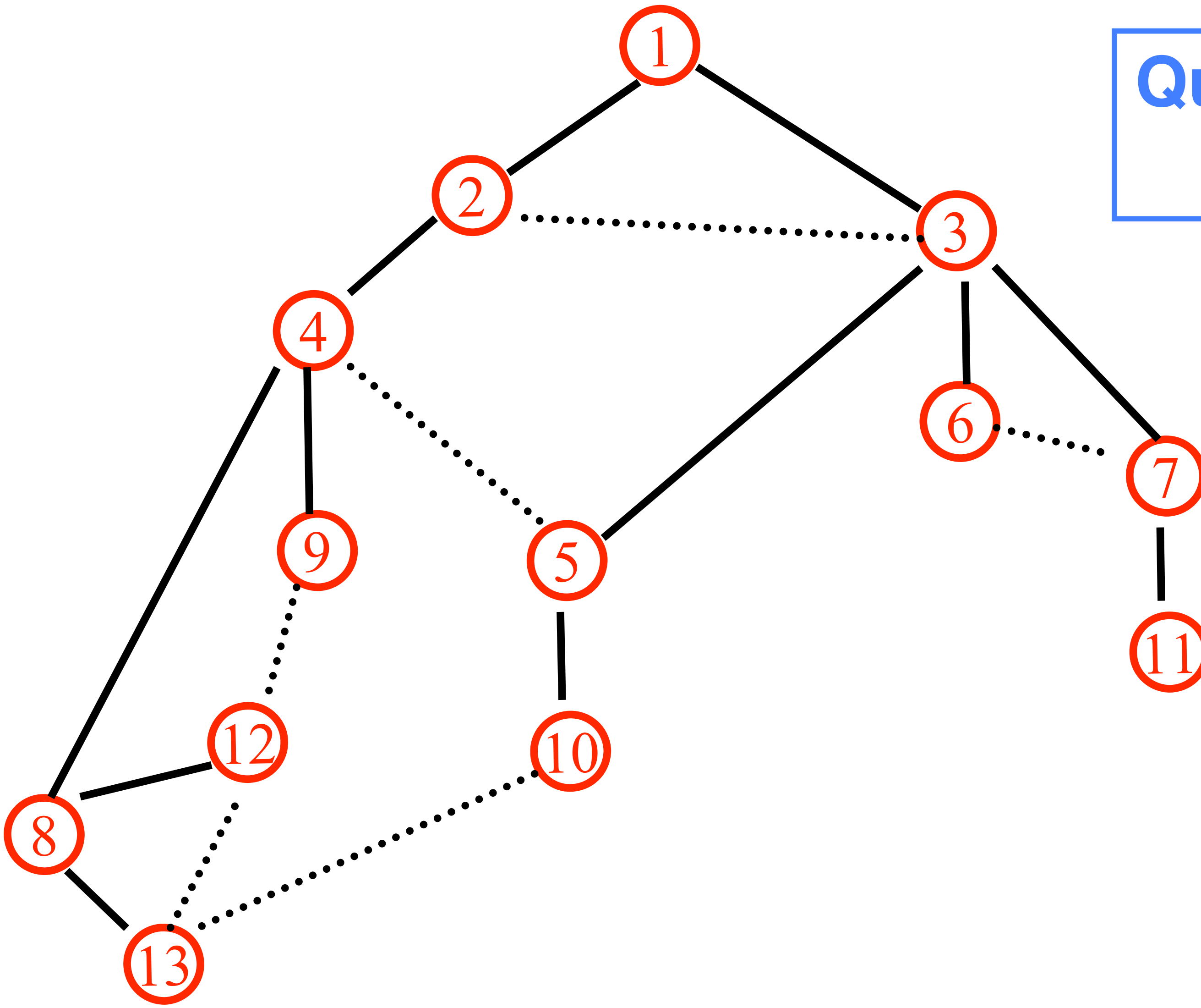
Queue:
8 9 10 11

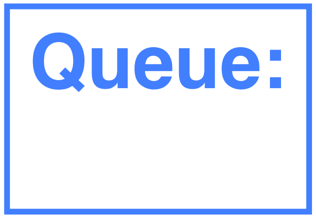
BFS(v)



Queue:
10 11 12 13

BFS(v)





BFS analysis

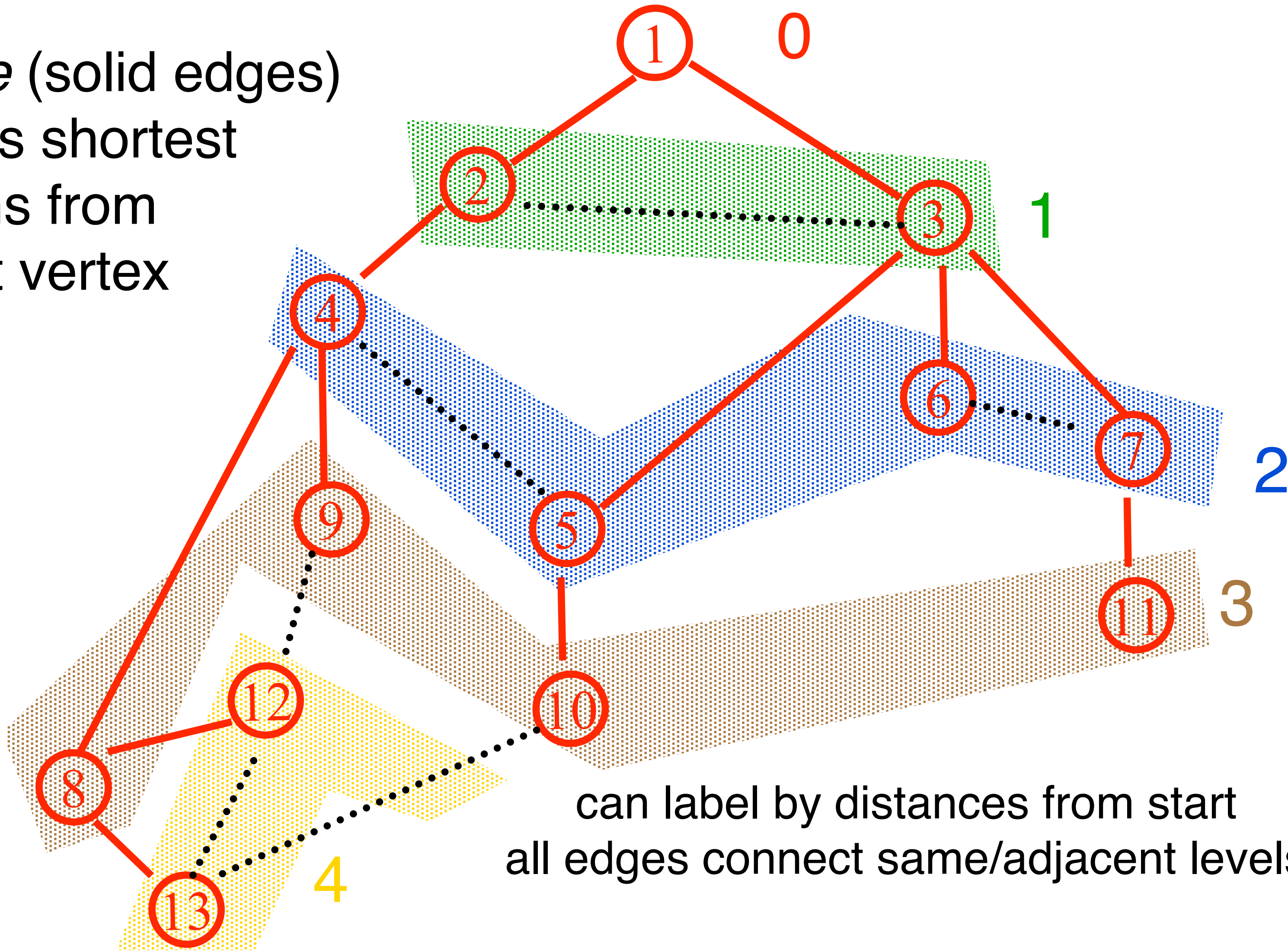
- Each edge is explored once from each end-point (at most)
- Each vertex is discovered by following a different edge
- Total cost O(m) where m=# of edges

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

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

Graph Search Application: Connected Components

- Want to answer questions of the form:
  - given vertices u and v, is there a path from u to v?

Q: Why not create 2-d array Path[u,v]?

- Idea: create array A such that

  A[u] = smallest numbered vertex that is connected to u

- question reduces to whether A[u]=A[v]?

Graph Search Application: Connected Components

- initial state: all v undiscovered
  for v=1 **to** n **do**
    if state(v) != fully-explored **then**
      BFS(v): **setting** A[u] ←v **for each** u **found**
      (and marking u discovered/fully-explored)
    endif
  endfor

- Total cost: O(n+m)
  - each edge is touched a constant number of times
  - works also with DFS

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) - explicit stack

Exercise: modify to compute vertex numbering

Global Initialization: mark all vertices "undiscovered"

```
DFS(v)
    mark  v "discovered"
    push (v,1) onto empty stack
    while stack not empty
        (u,i) = pop(stack)
        for ( ; i ≤ # of neighbors of u; i++)
          x = i^th edge on u's edge list
          if (x is undiscovered)
                mark x "discovered"
                push (u,i+1)      // save info to resume with u's next edge,
                u = x             // after exploring from x,
                i = 1             // (starting with its first edge)
        mark u completed
```

Idea: stack of unfinished vertices, plus pointers into their edge lists to say what work remains to finish.

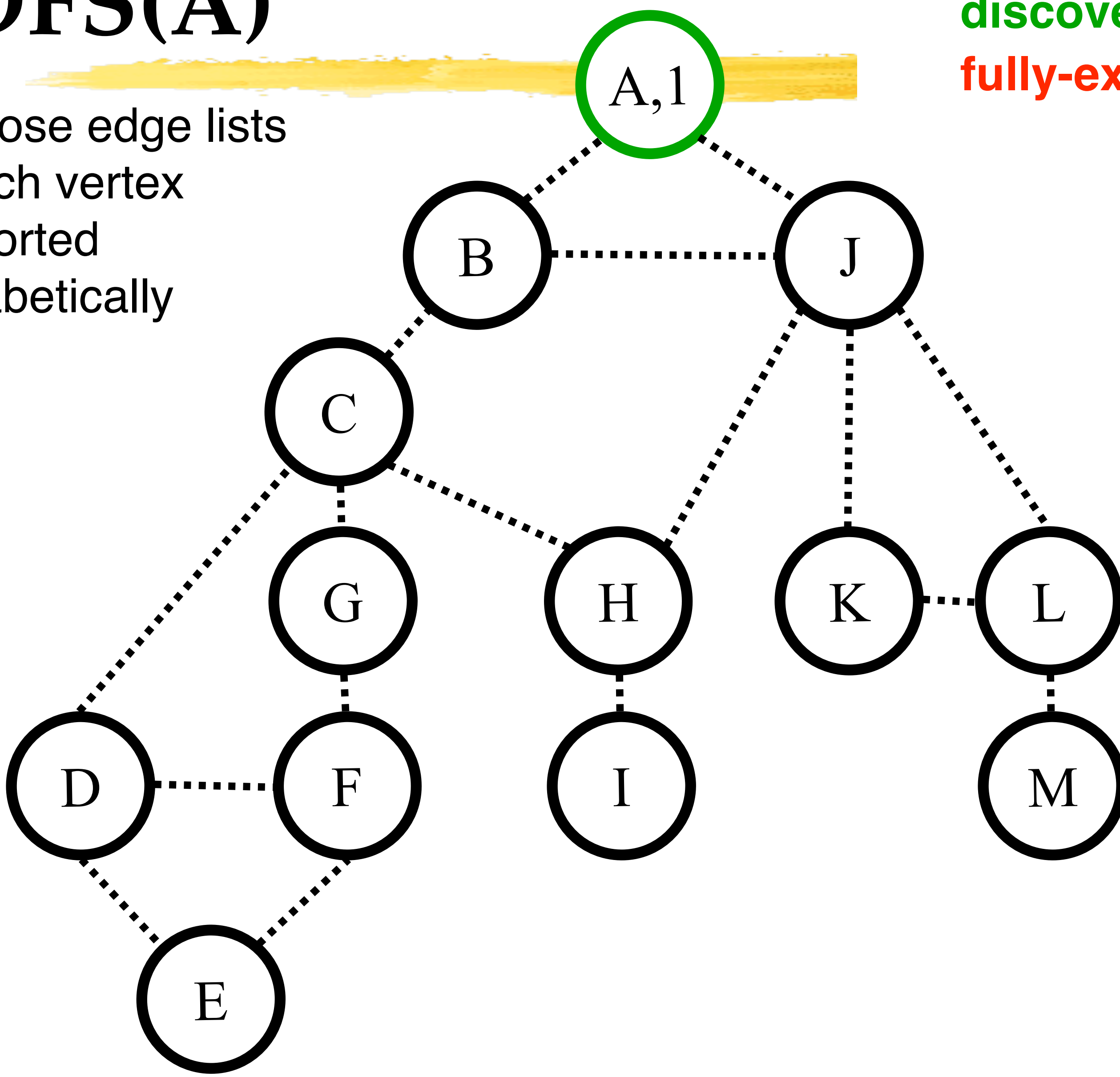
DFS(v) – Recursive version

Global Initialization:
    mark all vertices v "undiscovered” via v.dfs# = -1
    dfscounter = 0

```
DFS(v)
    v.dfs# = dfscounter++    // mark  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
```

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



Color code:

undiscovered

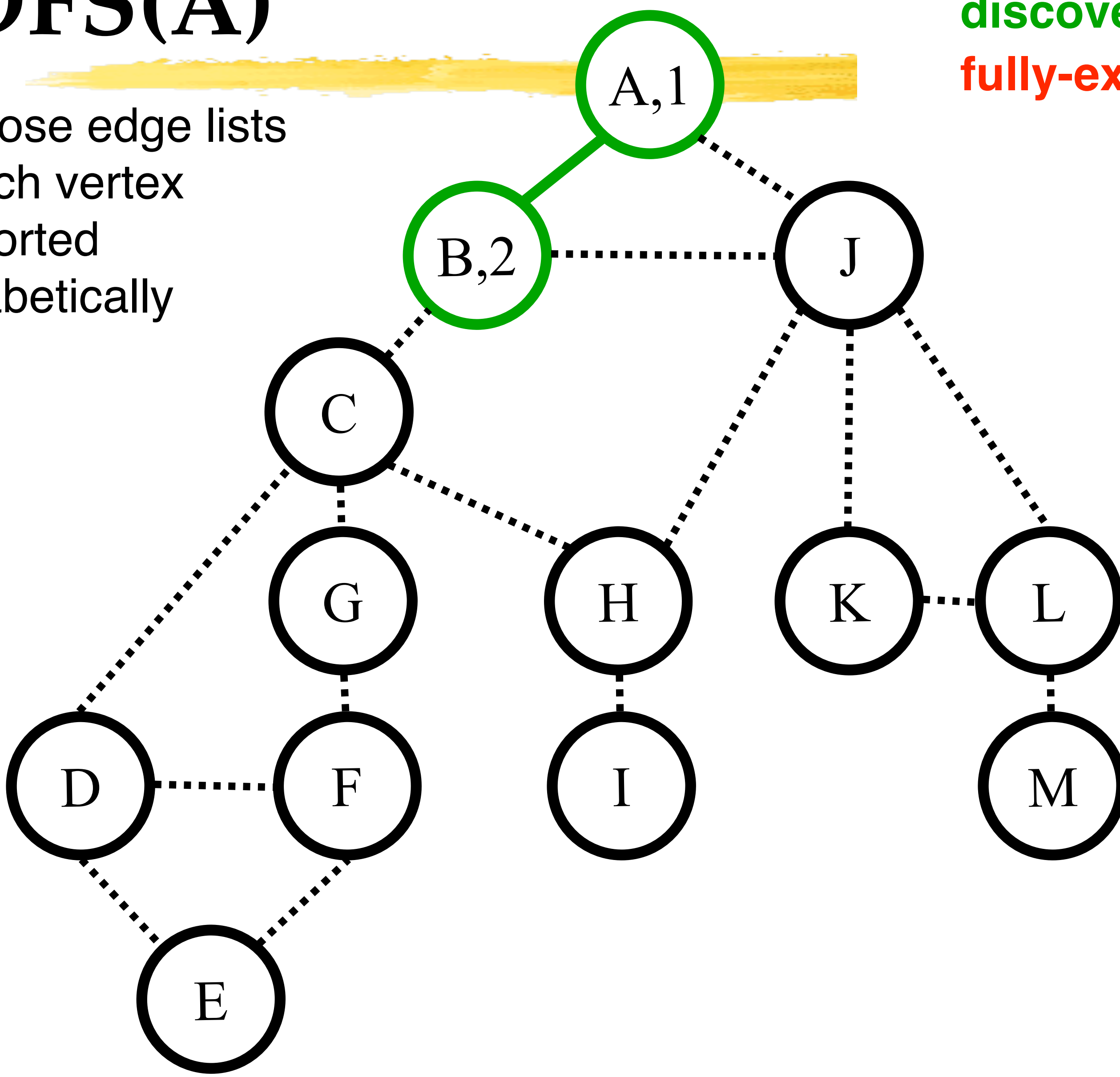
discovered

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)

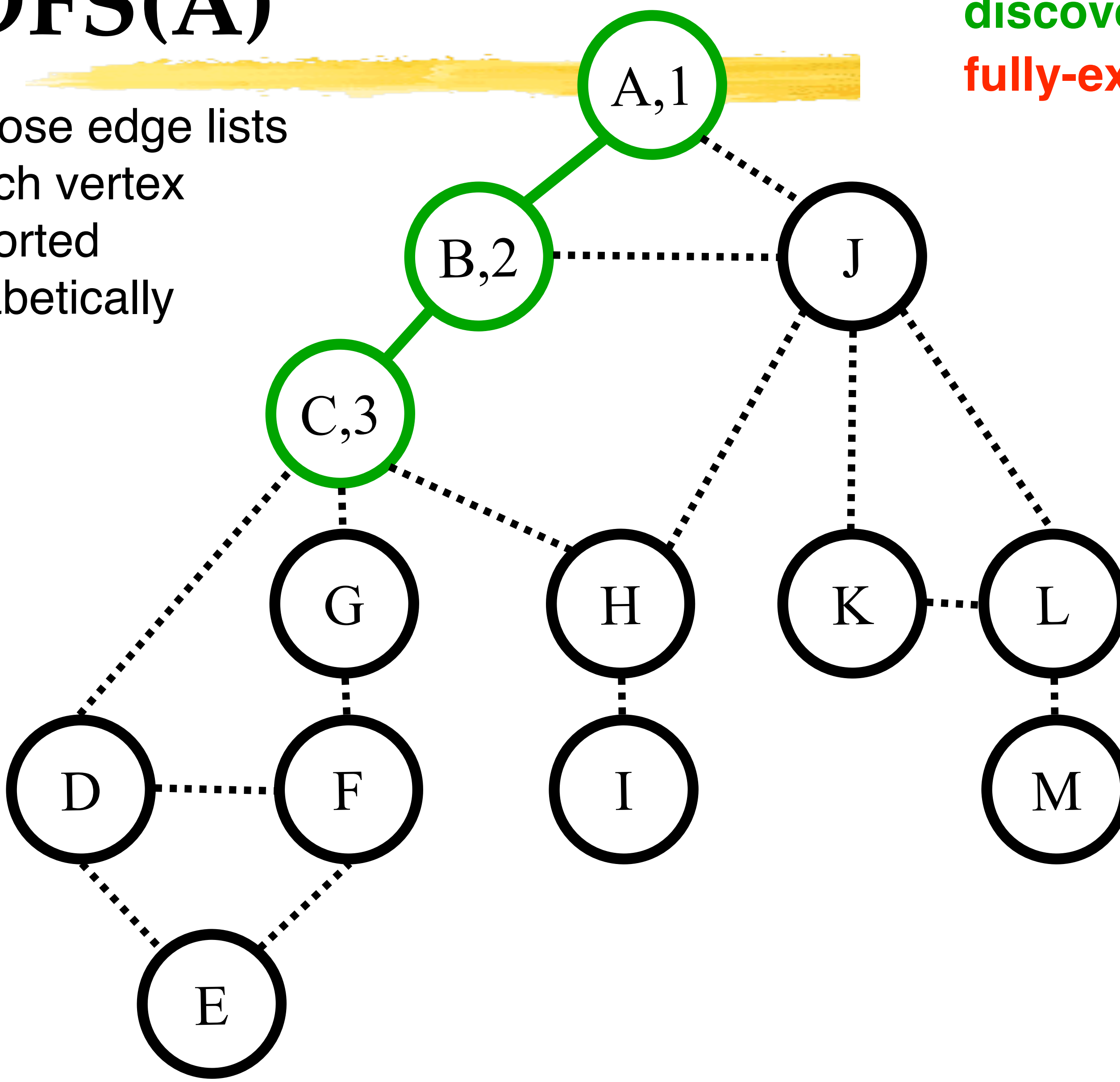
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored

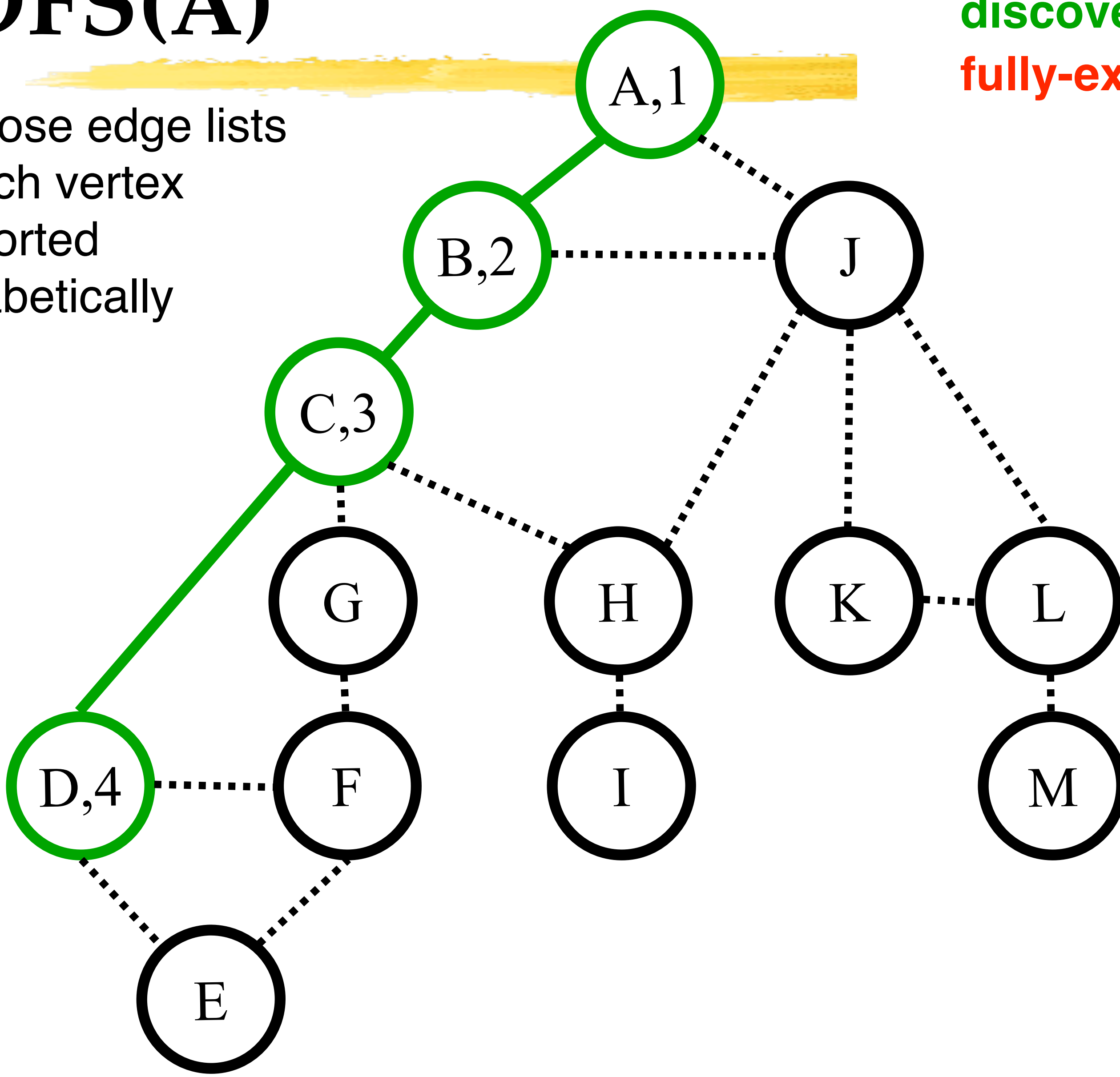


Call Stack:
(Edge list)

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

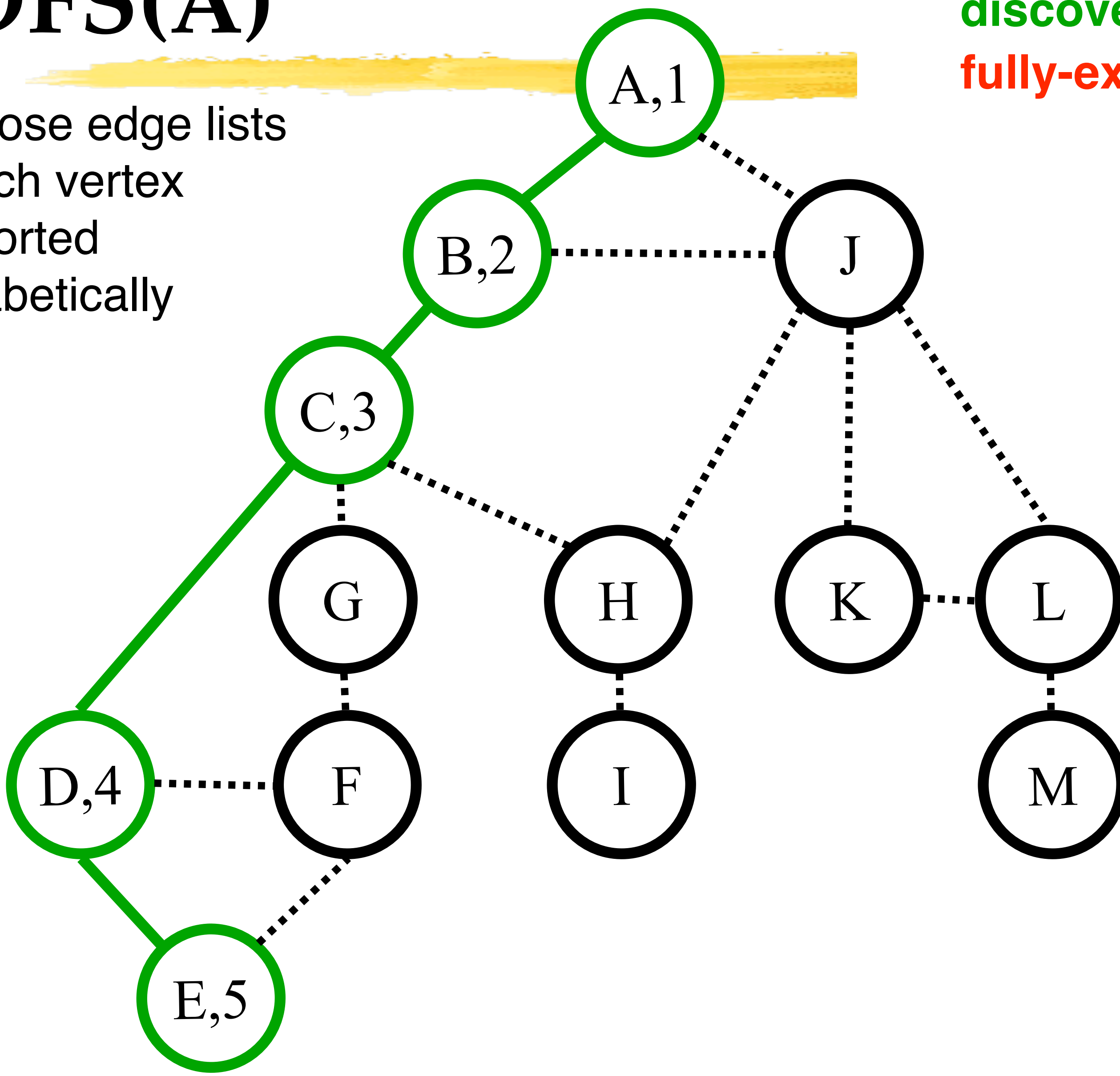
discovered

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)

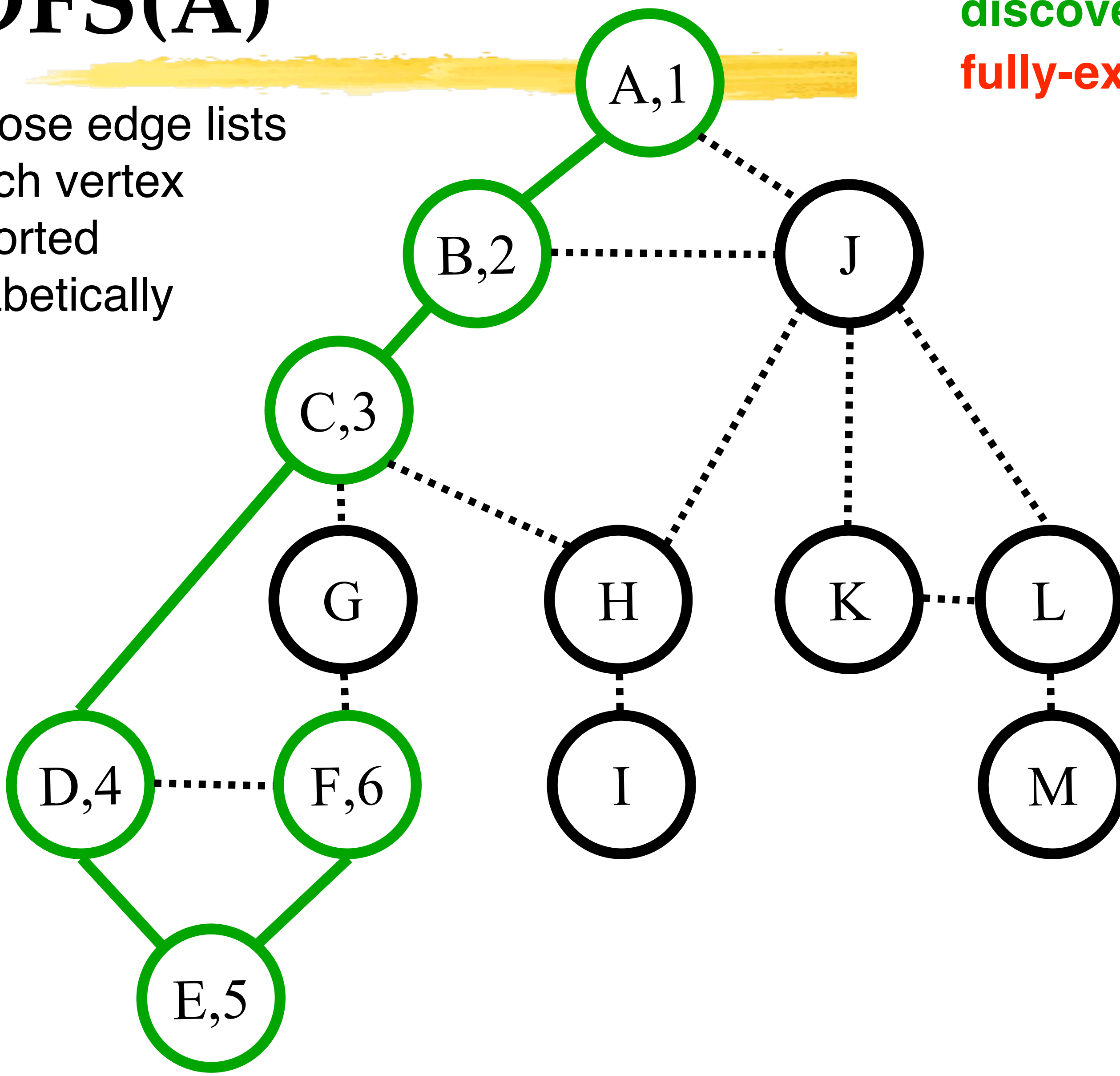
Suppose edge lists at each vertex are sorted alphabetically

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)

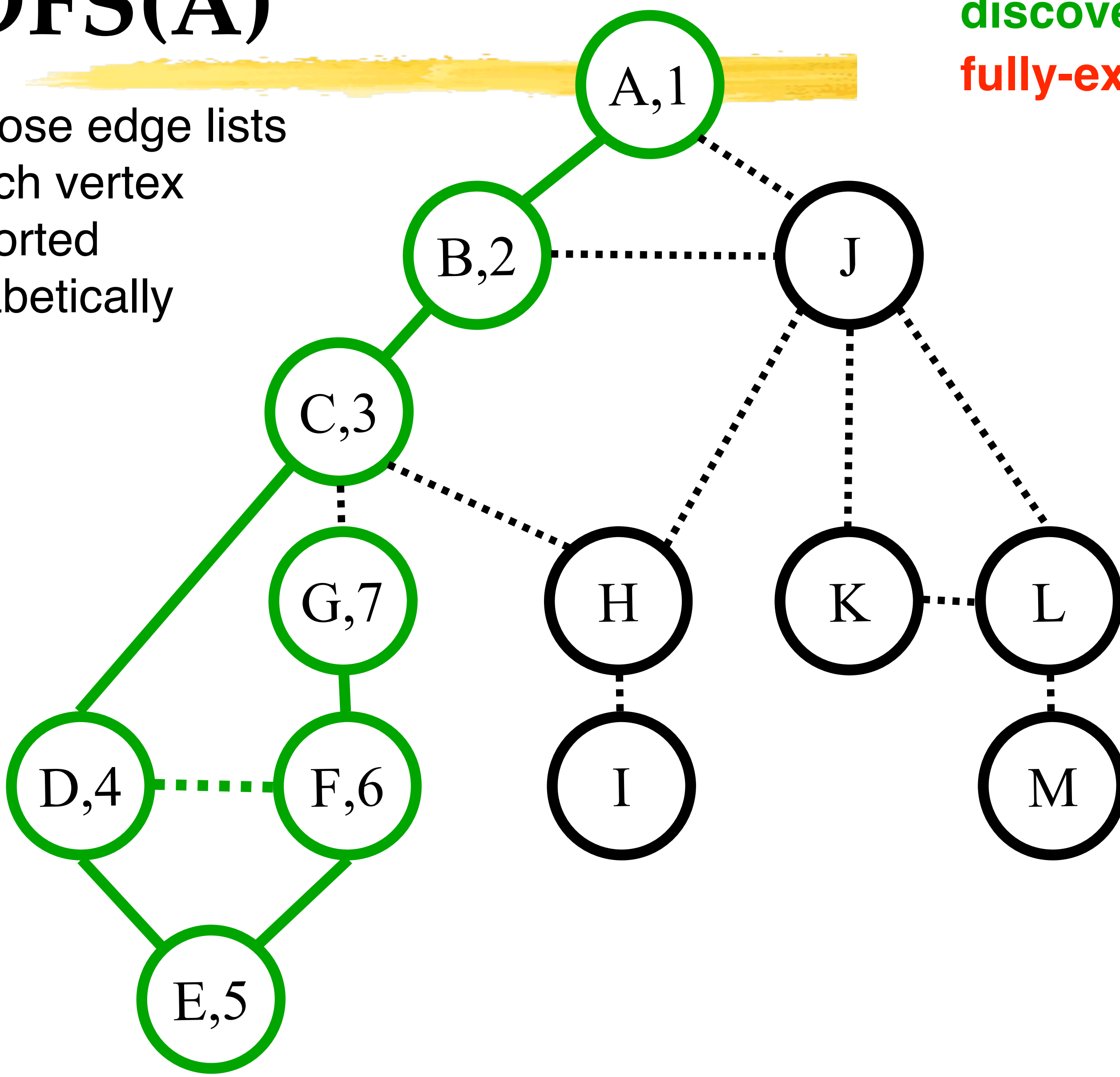
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored

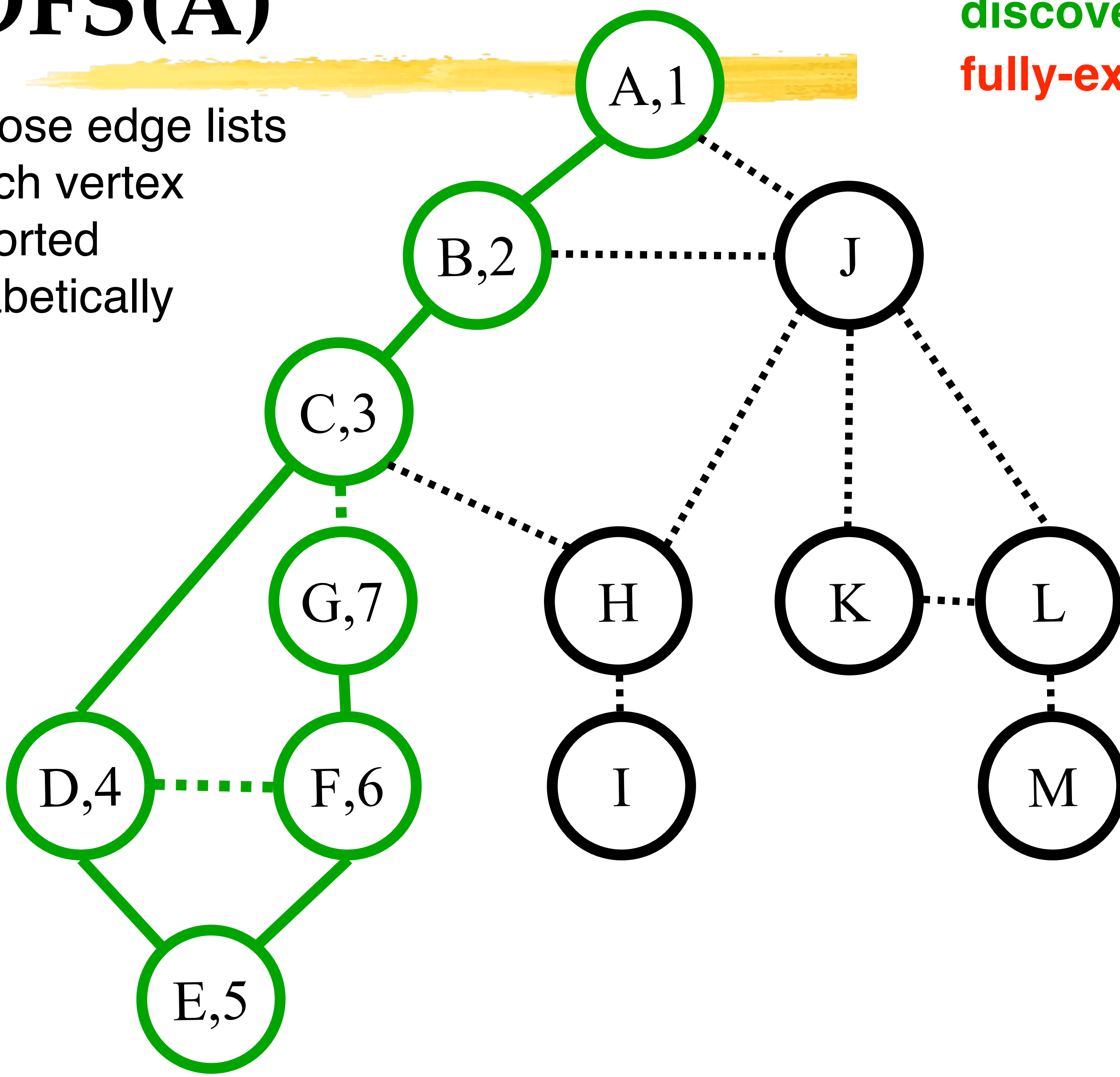


Call Stack:
(Edge list)

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

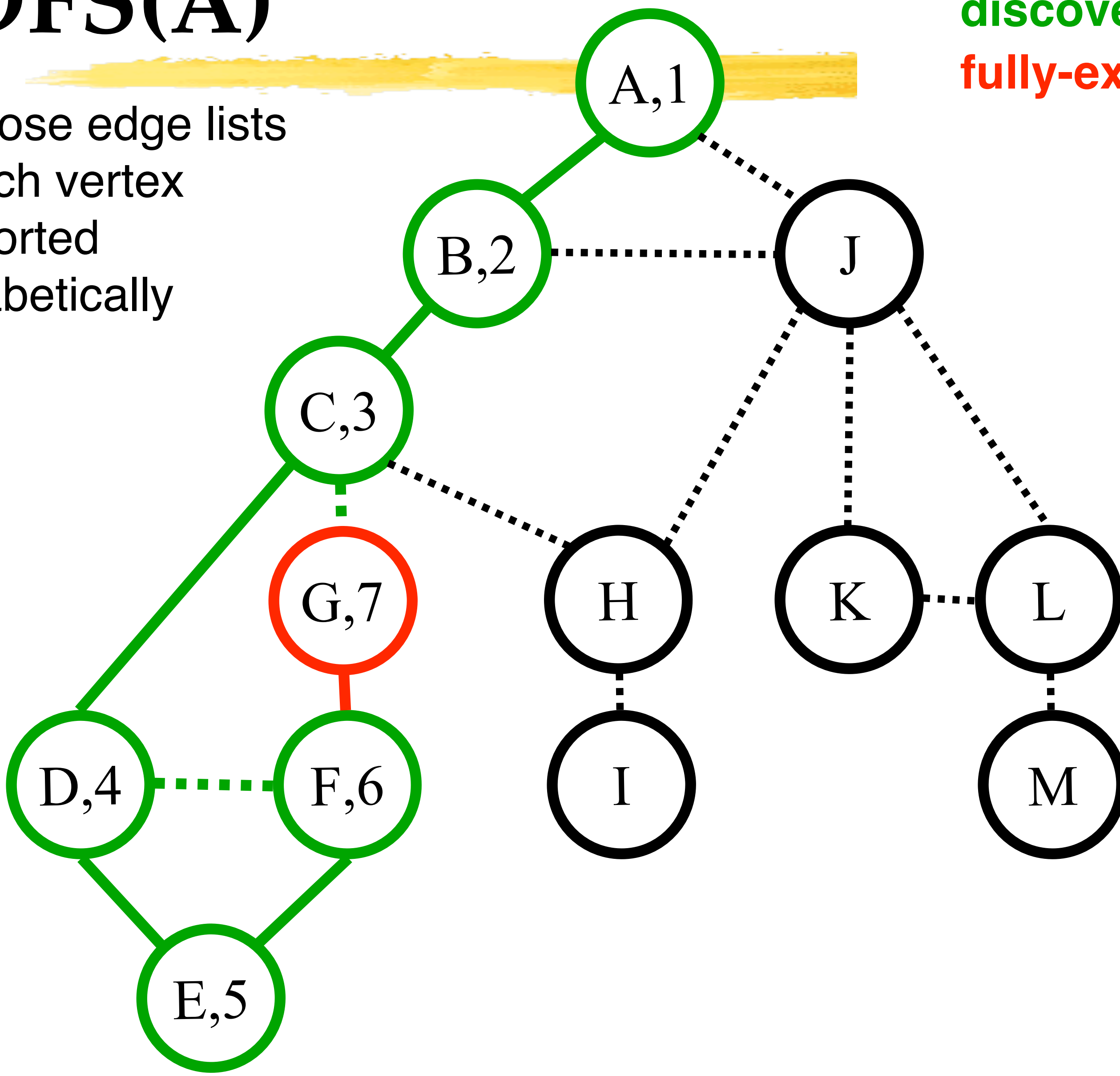
fully-explored

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)

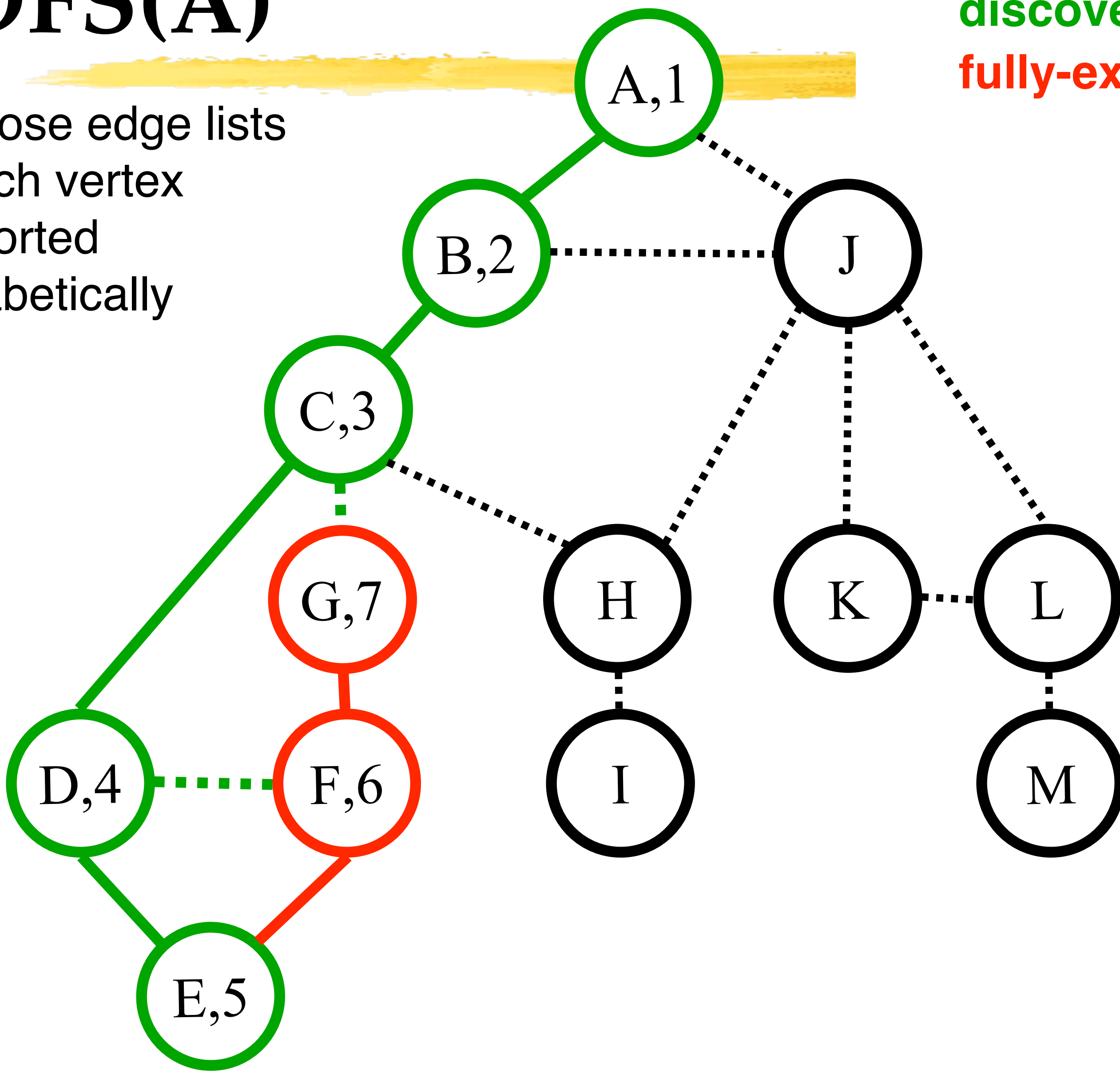
Suppose edge lists at each vertex are sorted alphabetically

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)

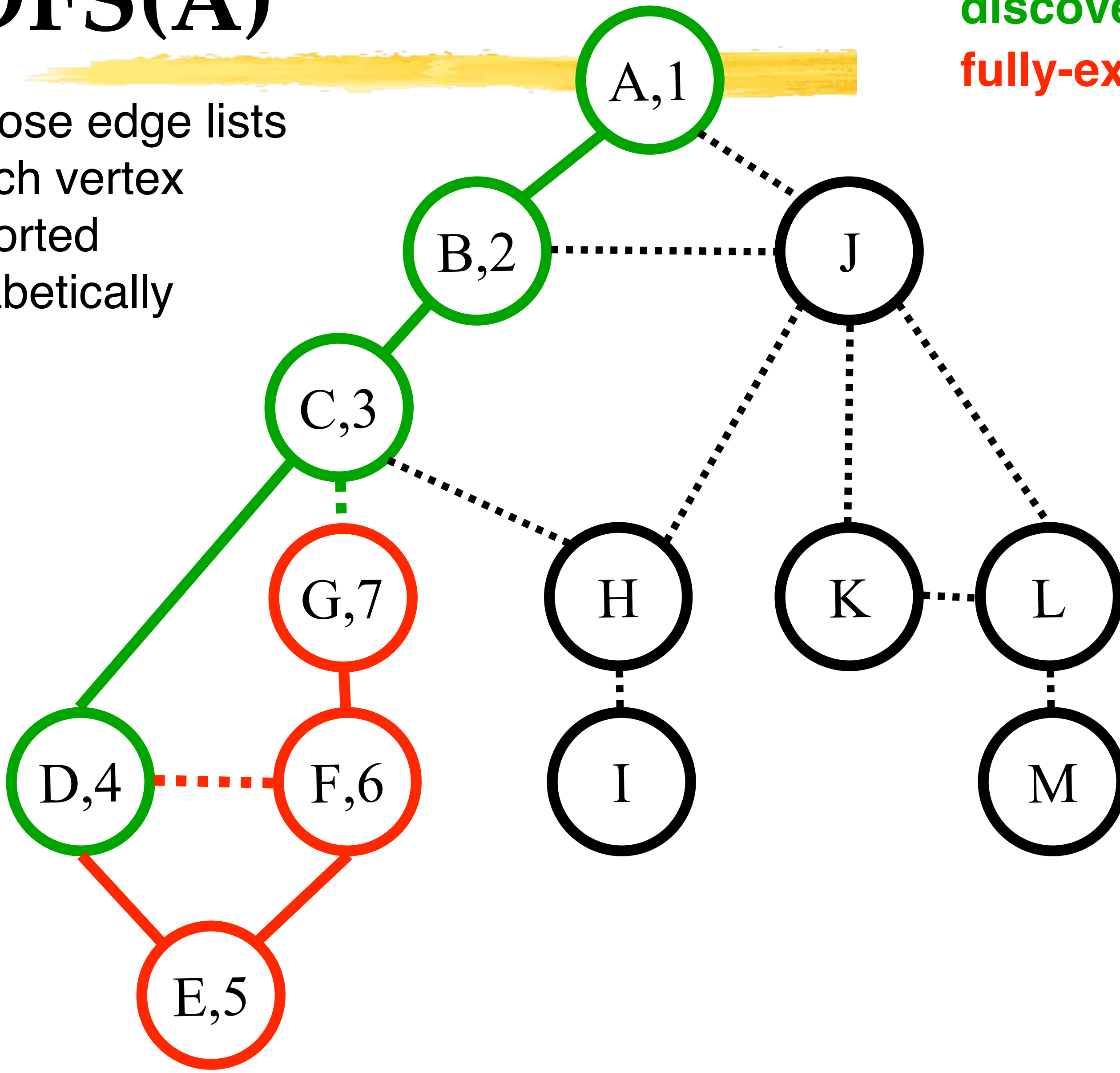
Color code:

undiscovered

discovered

fully-explored

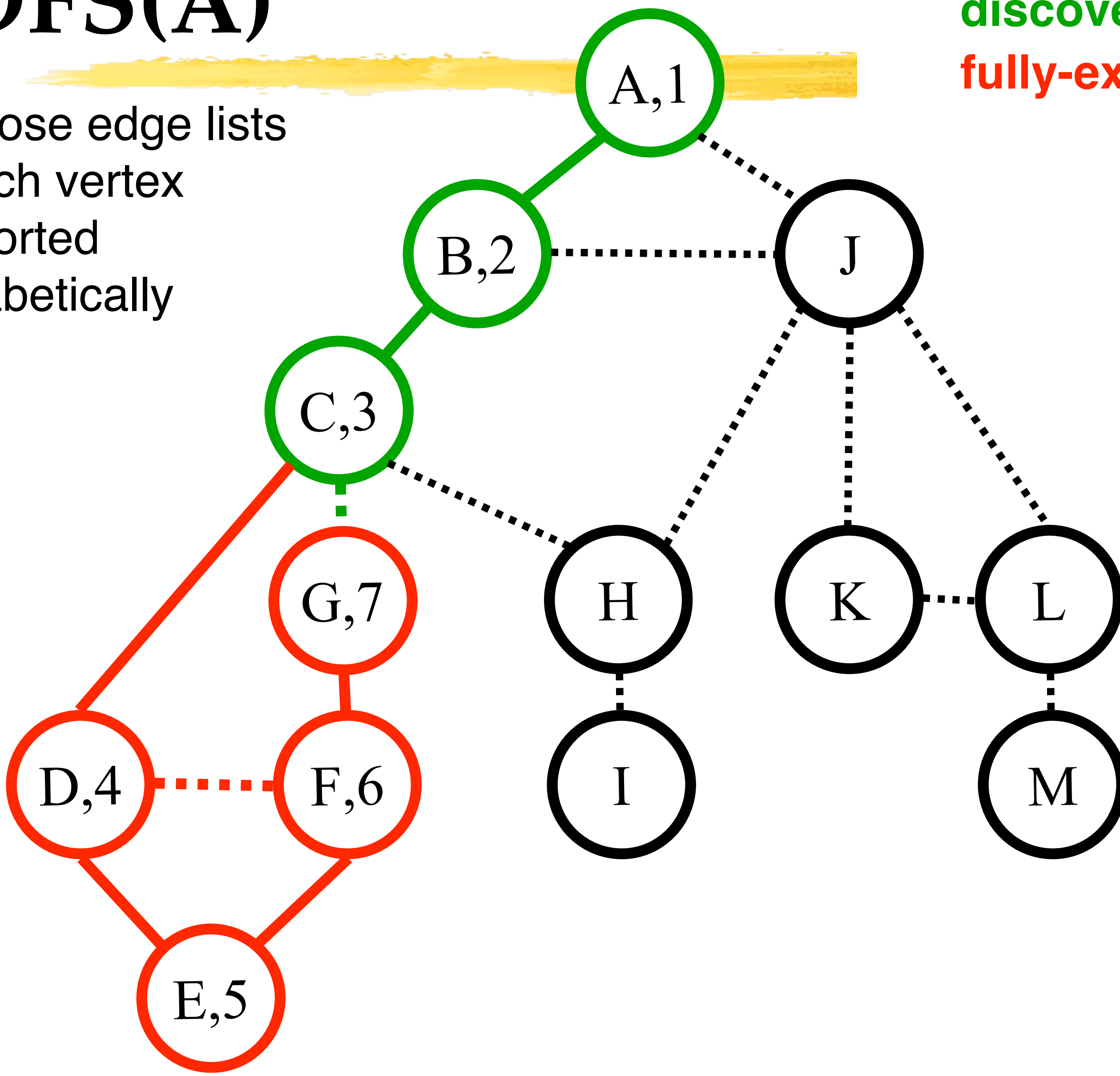
Suppose edge lists at each vertex are sorted alphabetically



Call Stack: (Edge list)
A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,G,H)
D (~~C~~,~~E~~,~~F~~)

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



Color code:

undiscovered

discovered

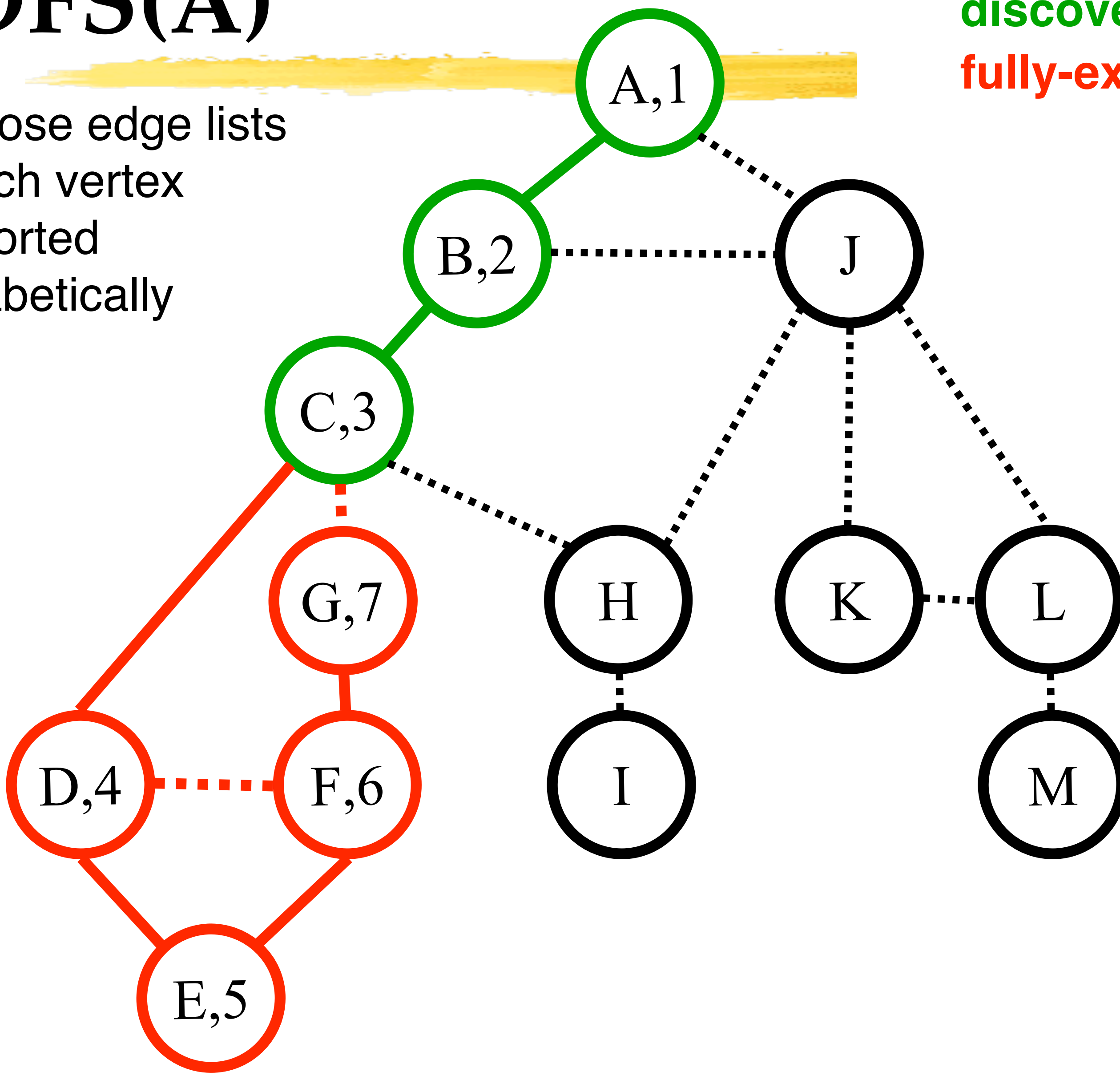
fully-explored

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)

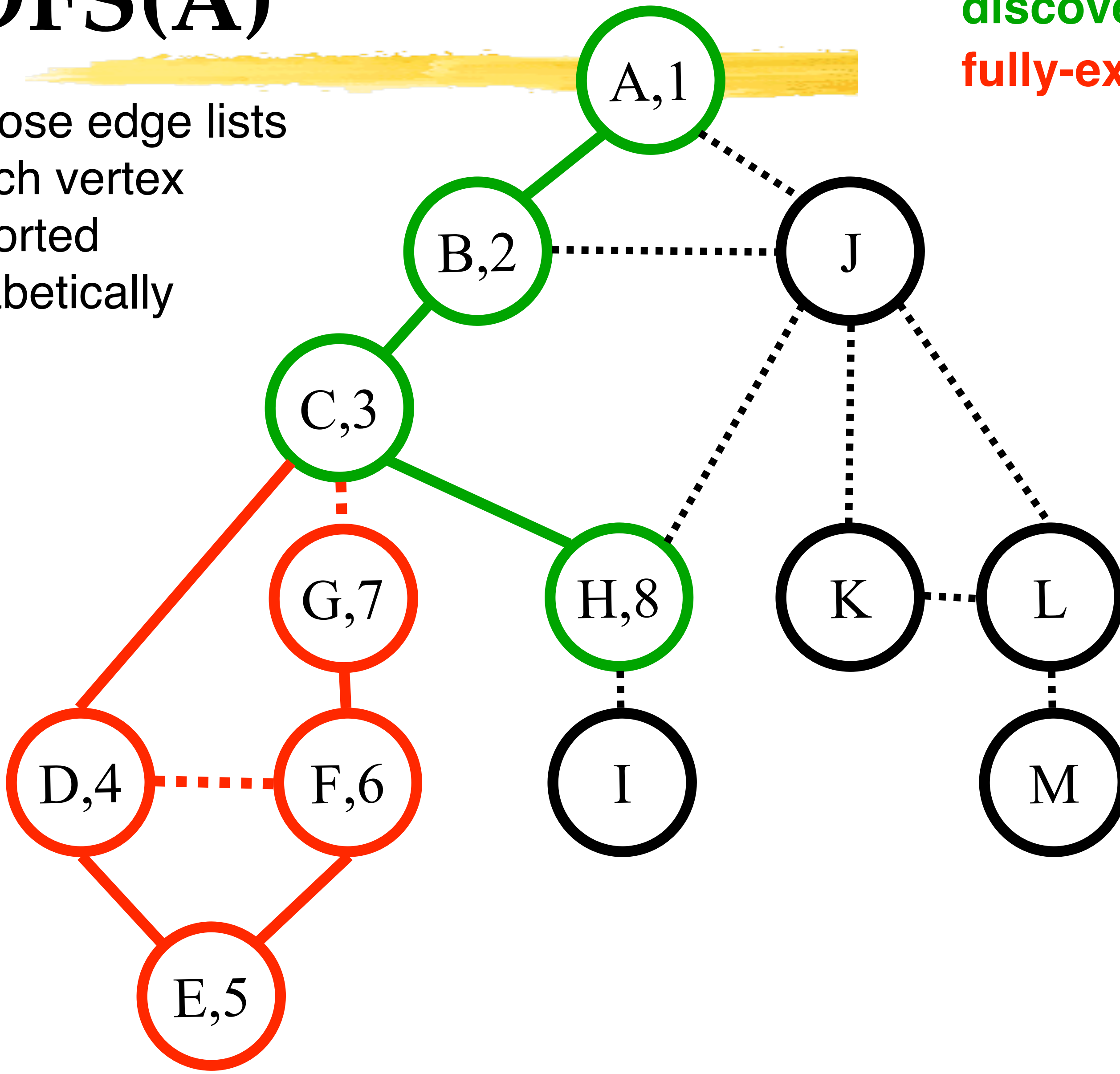
Color code:

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~~)
H (C,I,J)

DFS(A)

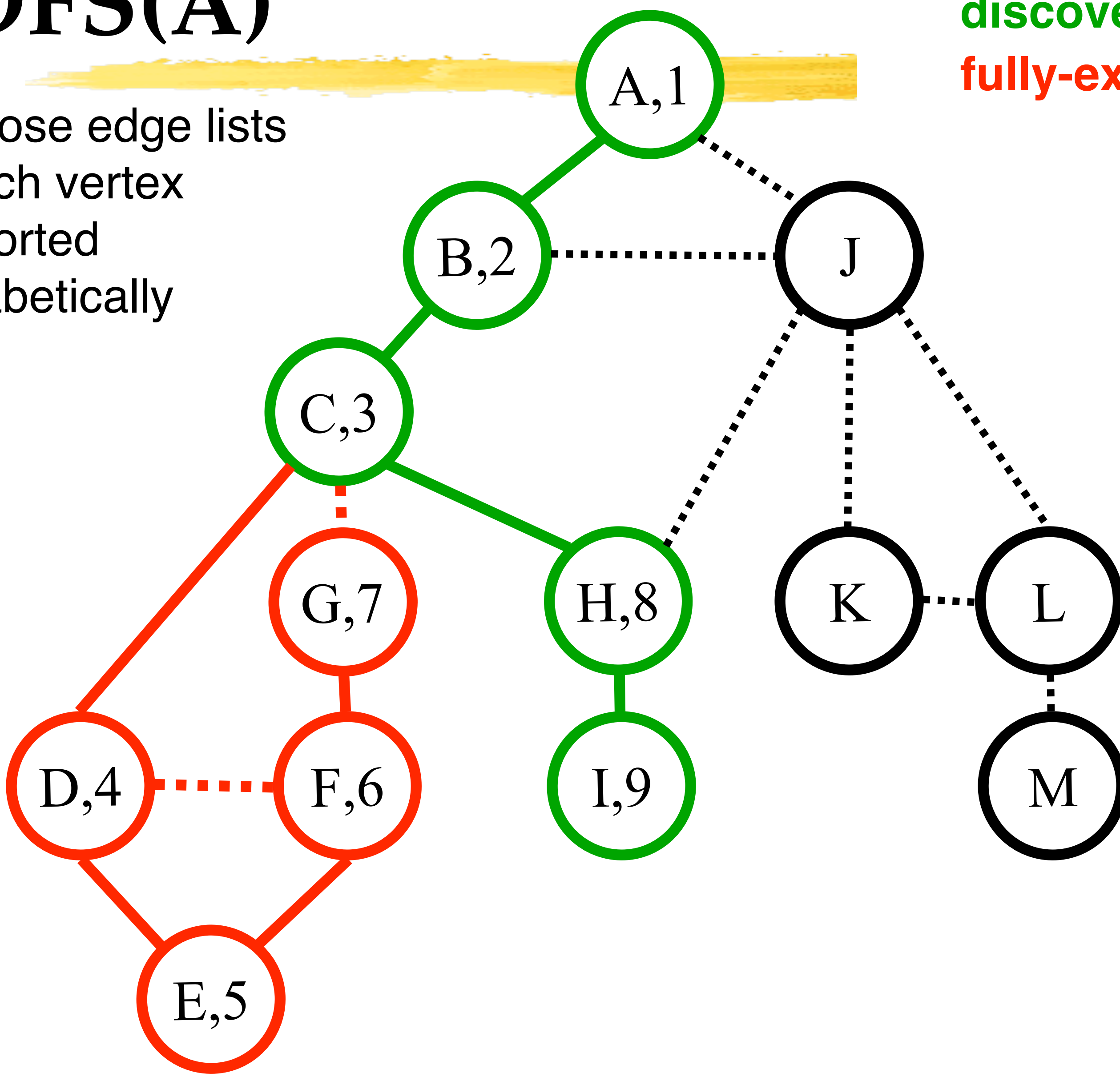
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored

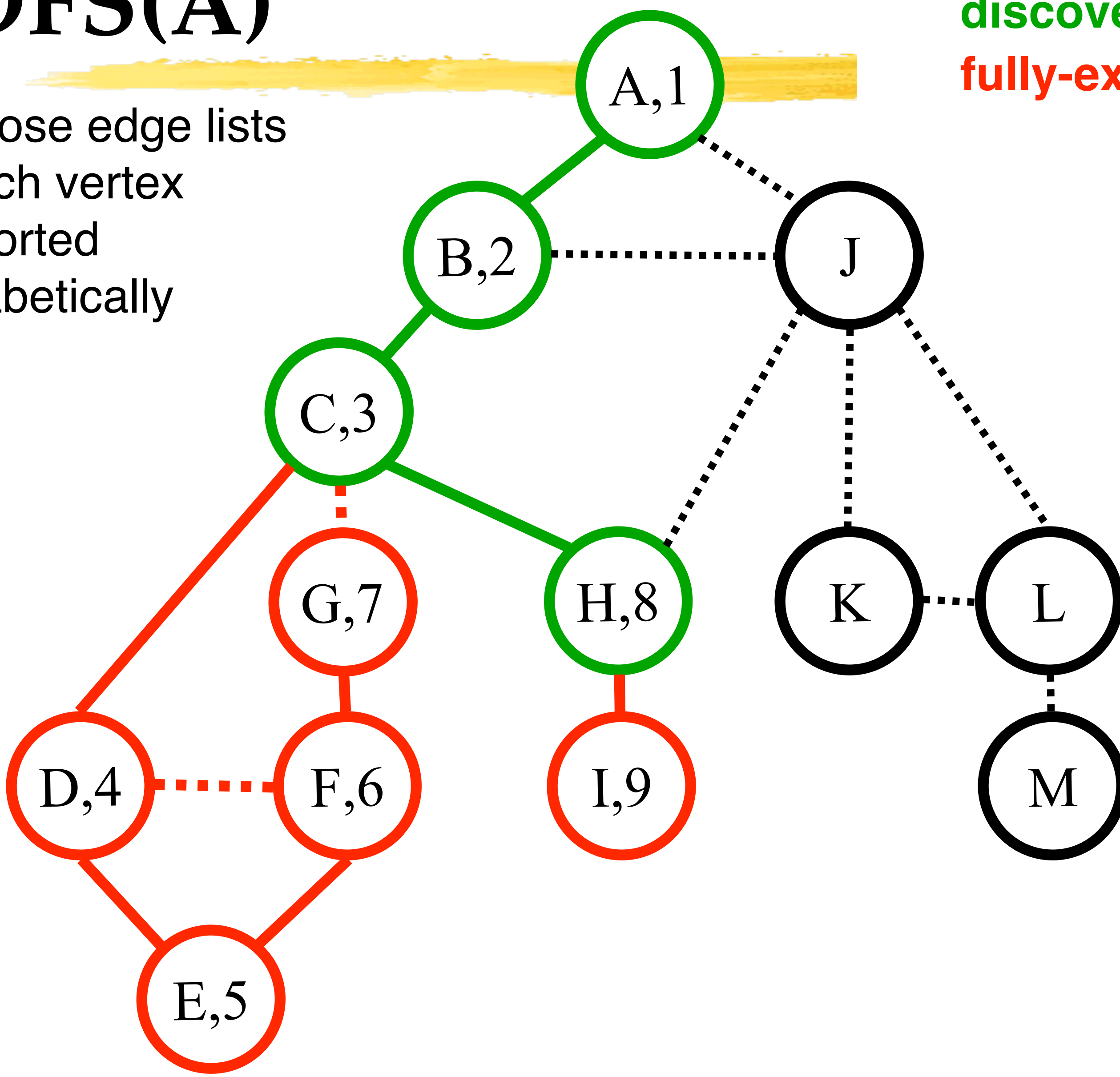


Call Stack:
(Edge list)

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)

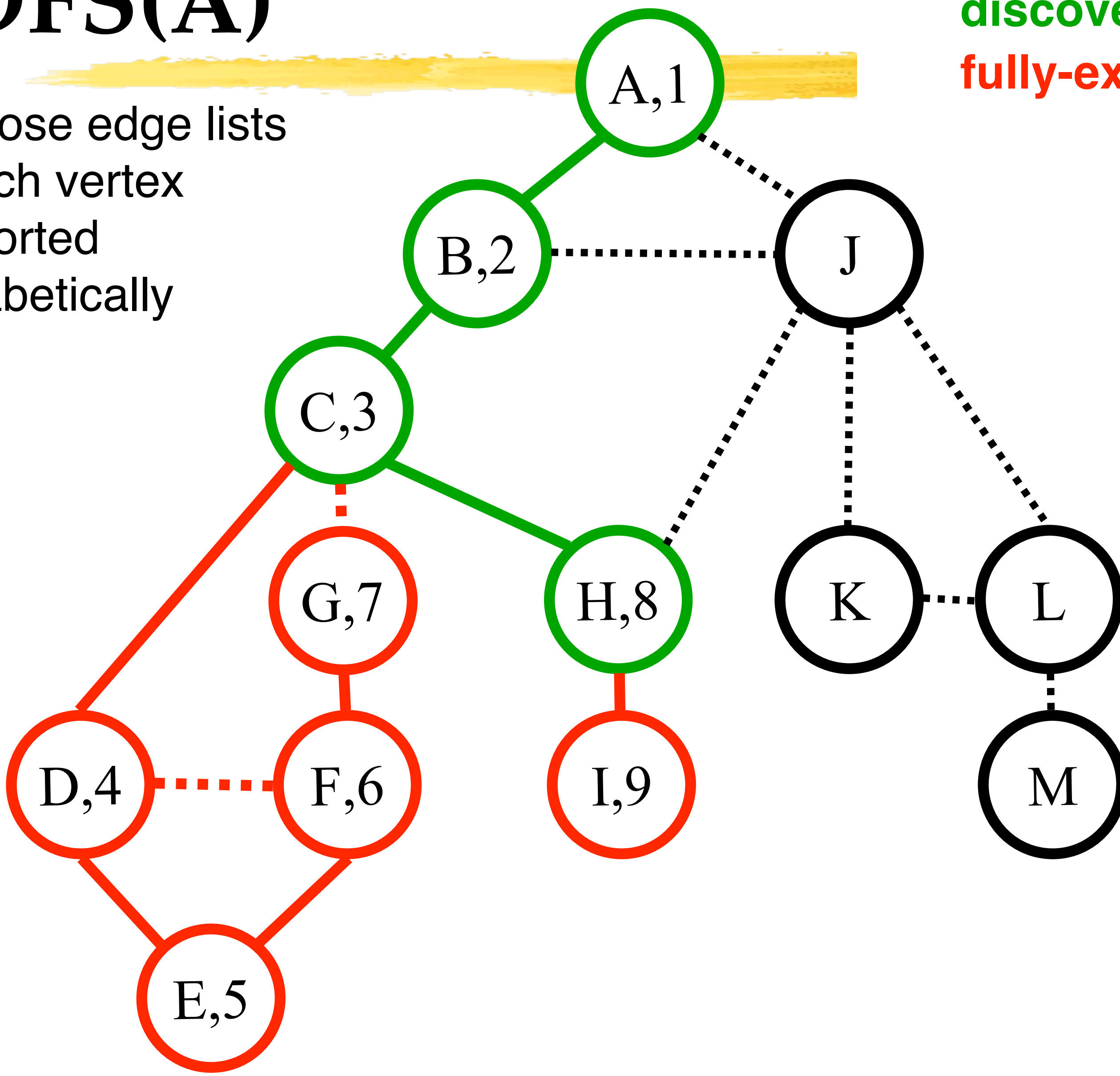
A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,J)
I (~~H~~)

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered
discovered
fully-explored



Call Stack:
(Edge list)

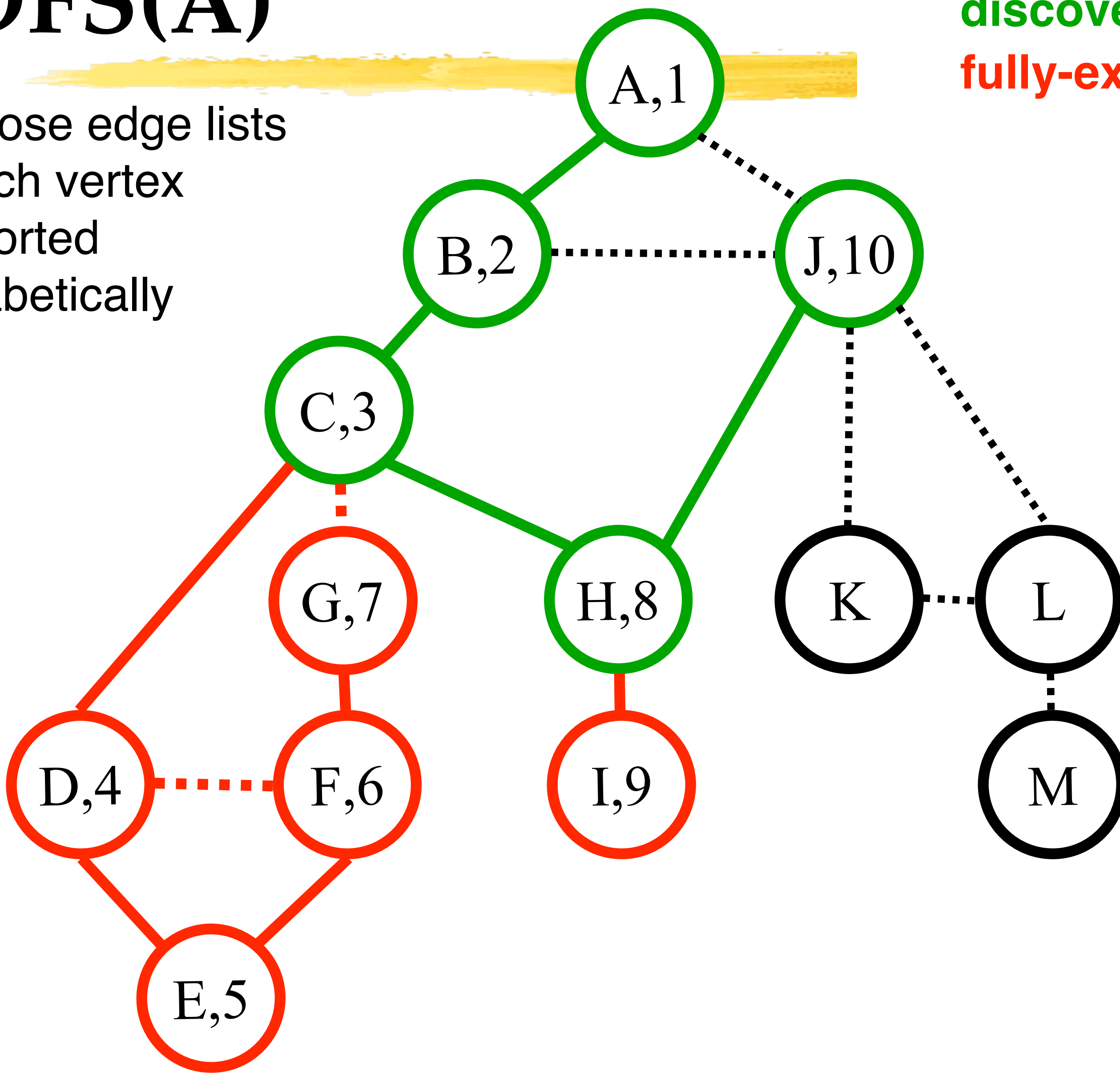
A (~~B~~,J)
B (~~A~~,~~C~~,J)
C (~~B~~,~~D~~,~~G~~,~~H~~)
H (~~C~~,~~I~~,J)

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)

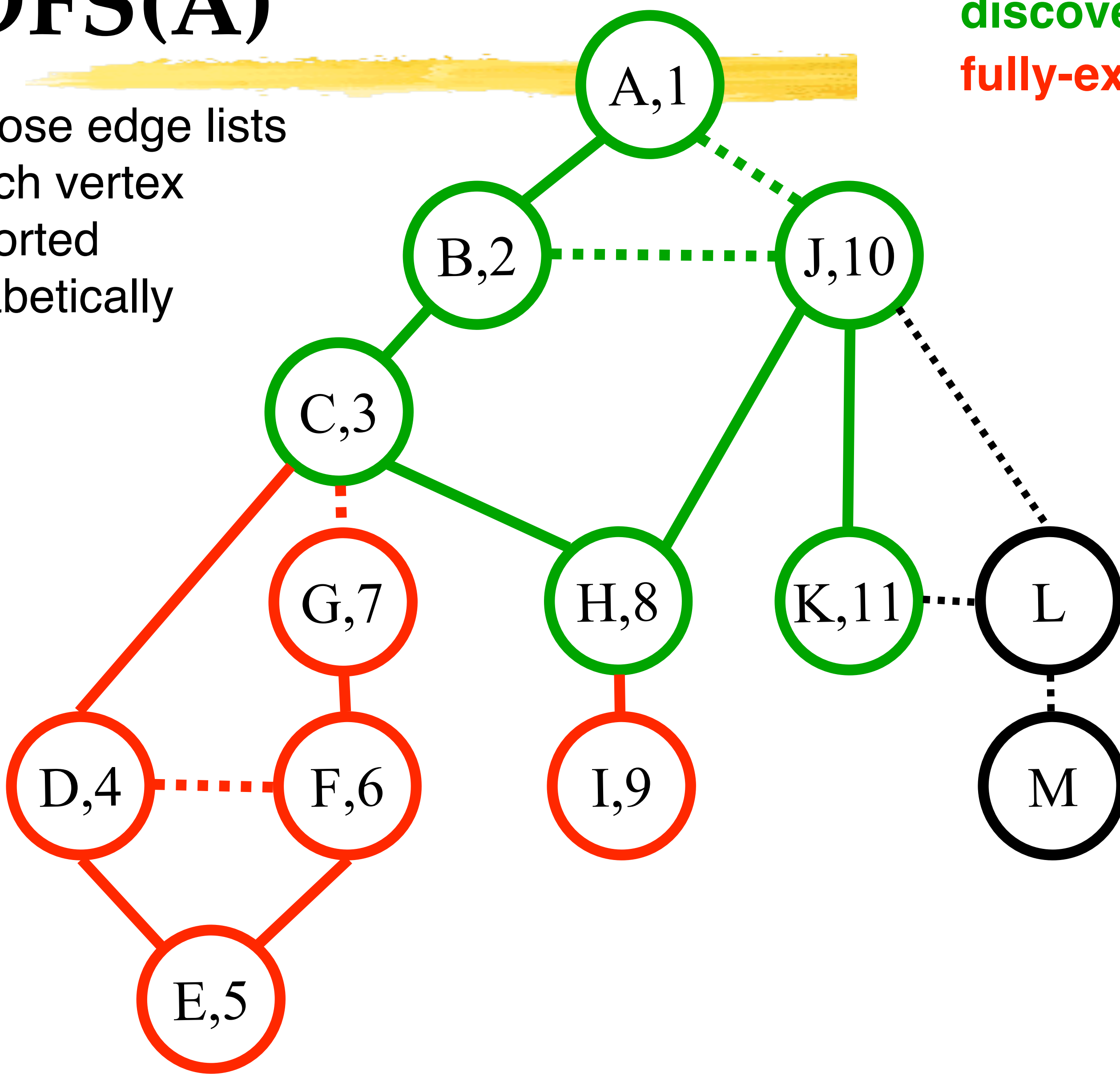
Suppose edge lists at each vertex are sorted alphabetically

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)

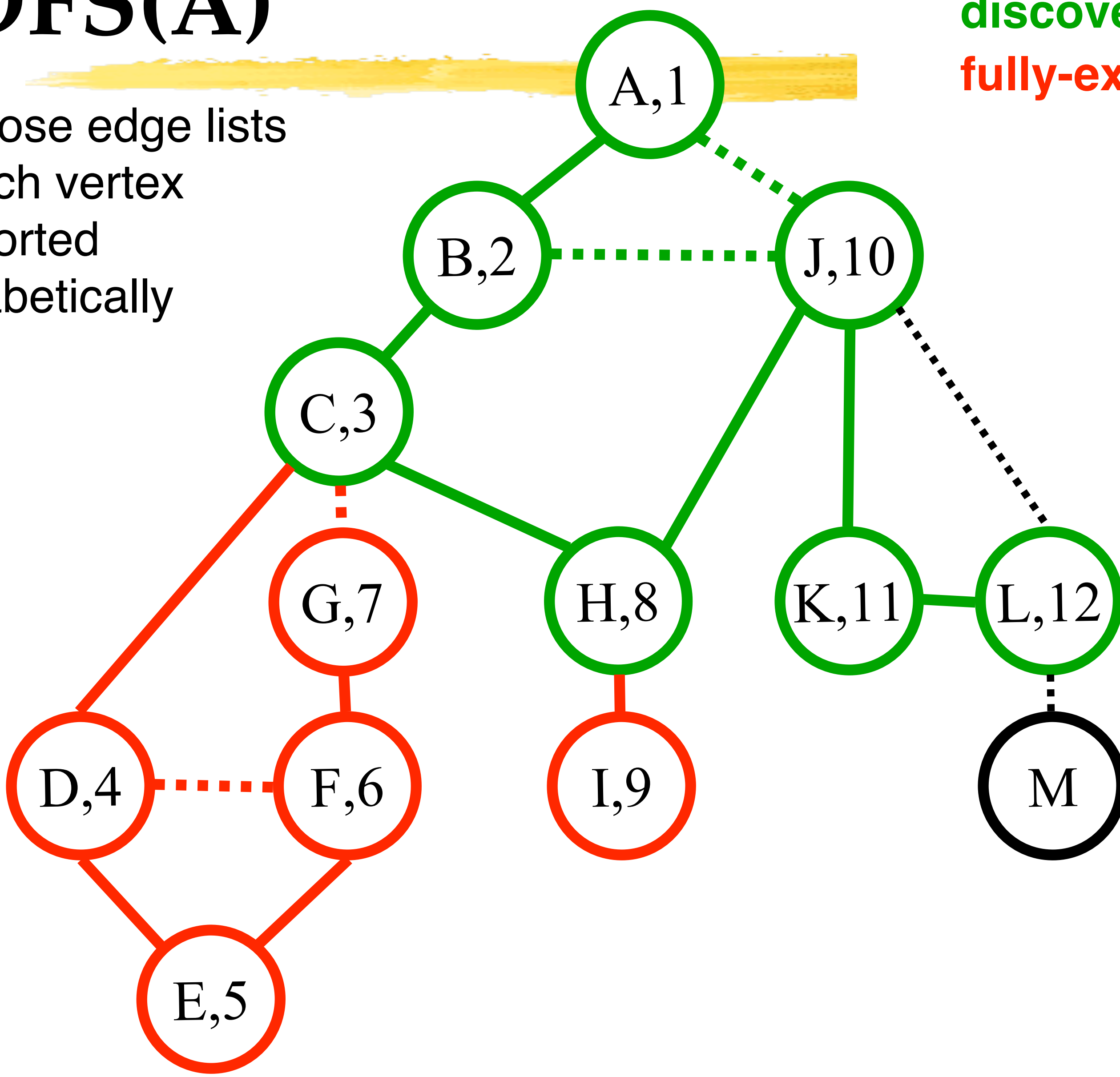
Suppose edge lists at each vertex are sorted alphabetically

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)

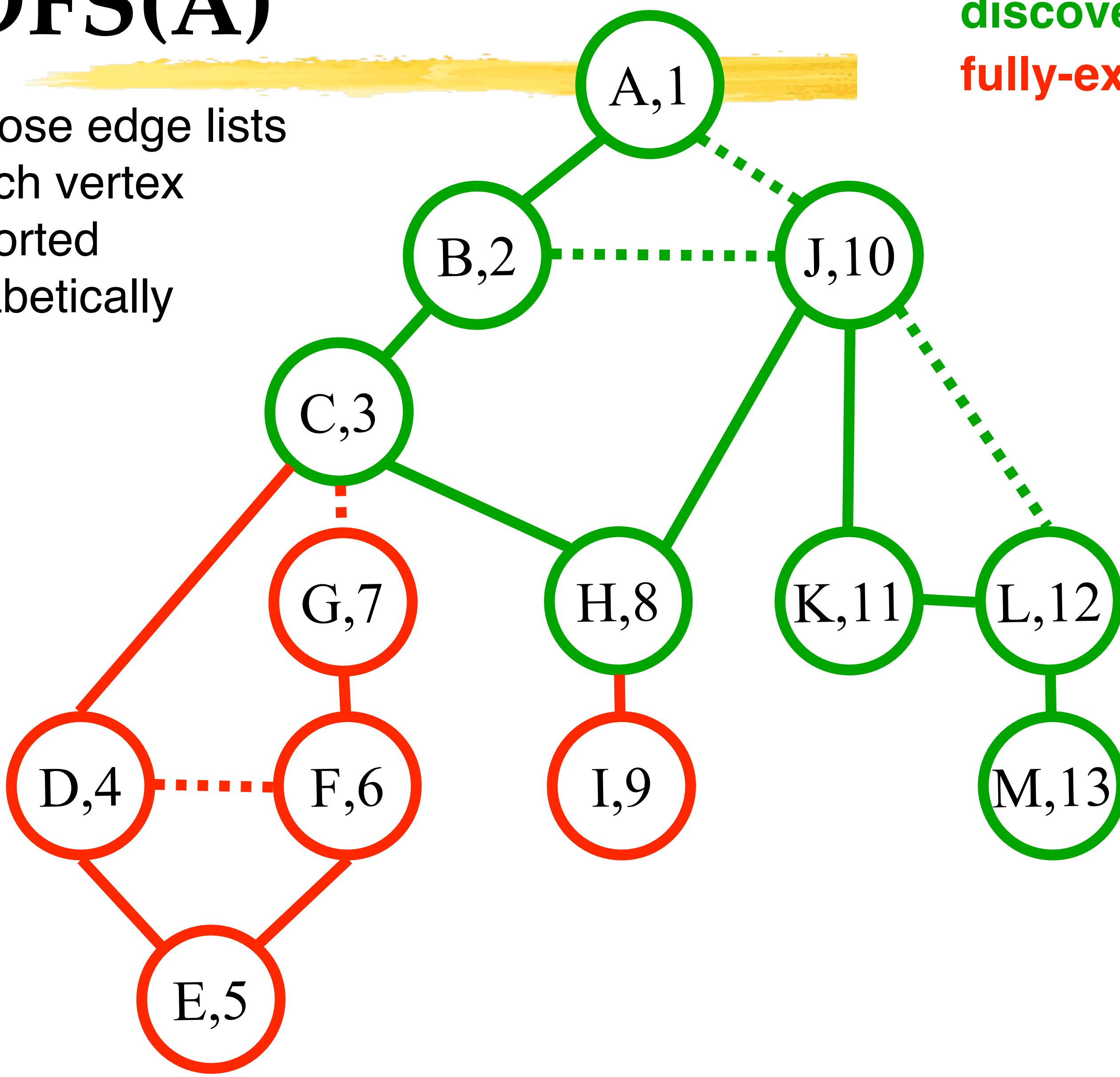
Suppose edge lists at each vertex are sorted alphabetically

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)

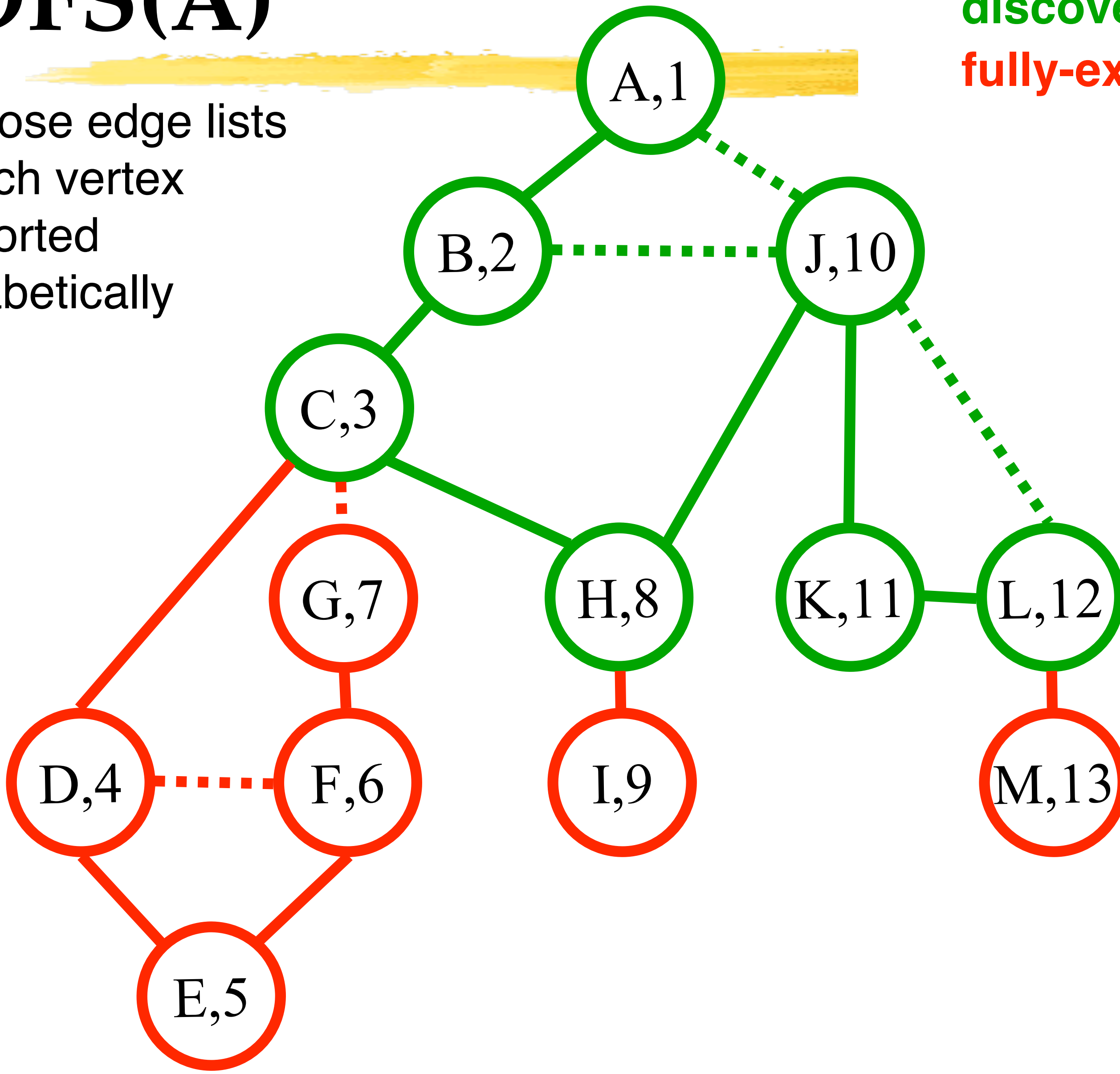
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored

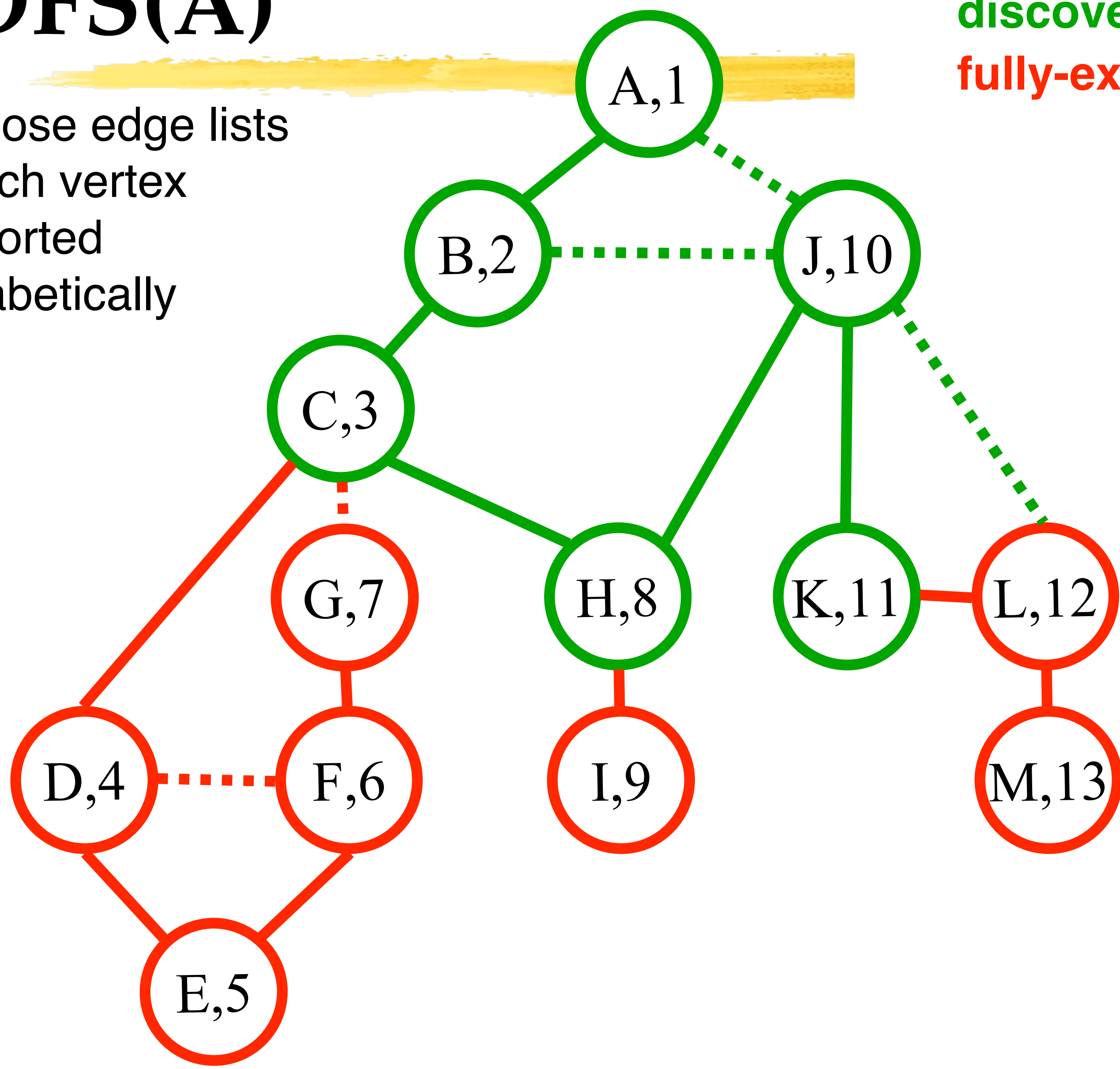


Call Stack:
(Edge list)

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

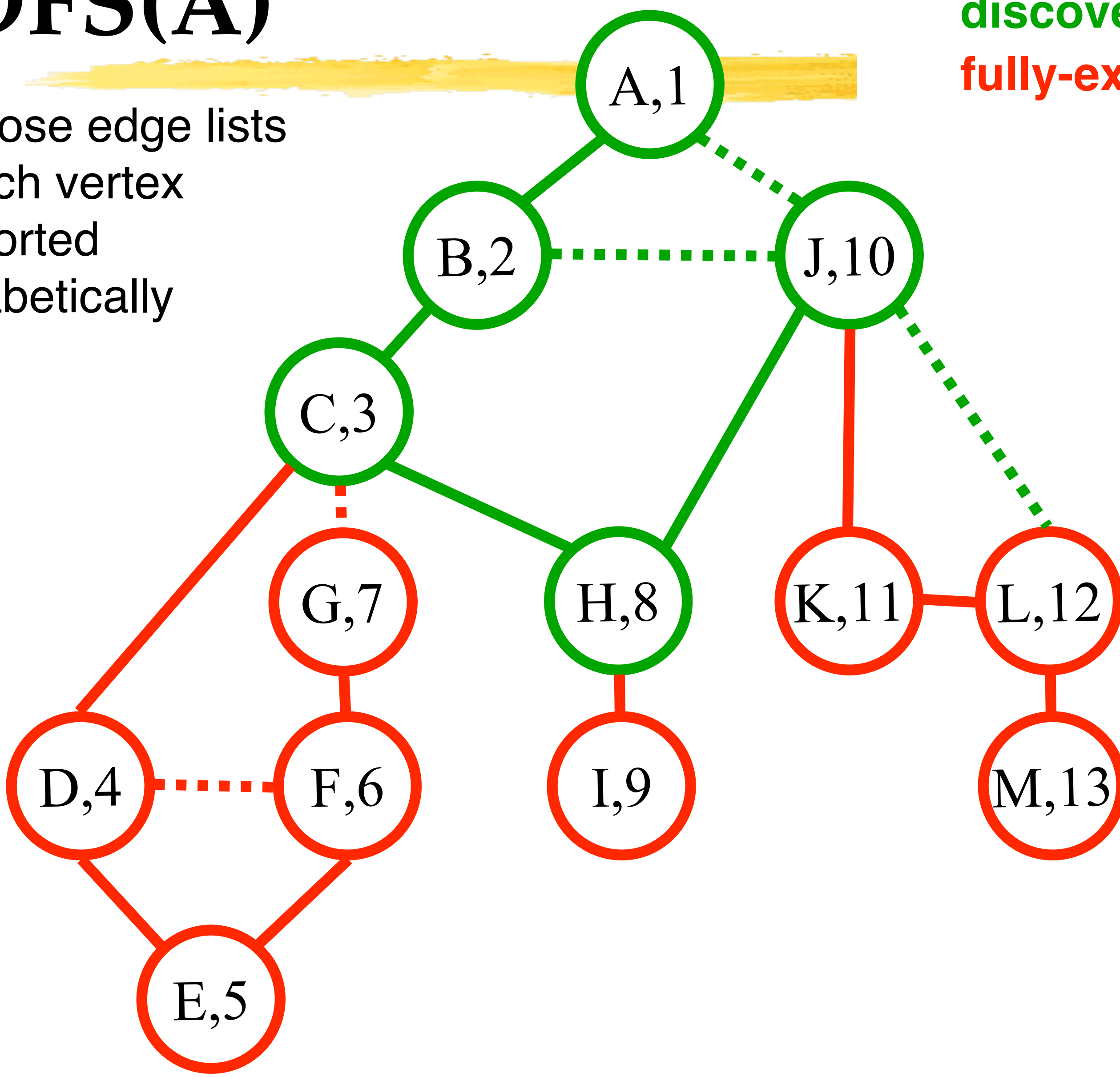
fully-explored

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)

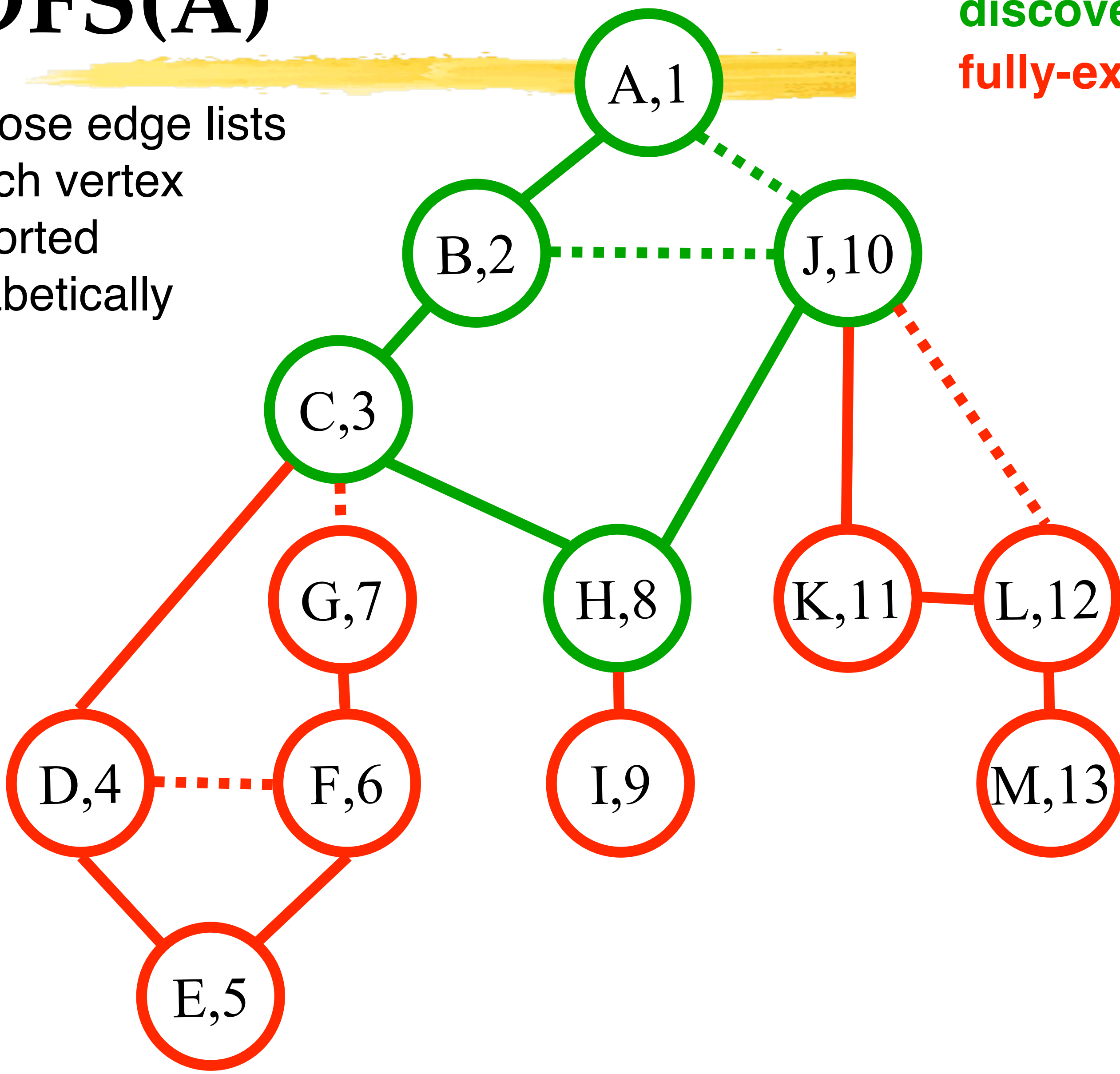
Suppose edge lists at each vertex are sorted alphabetically

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)

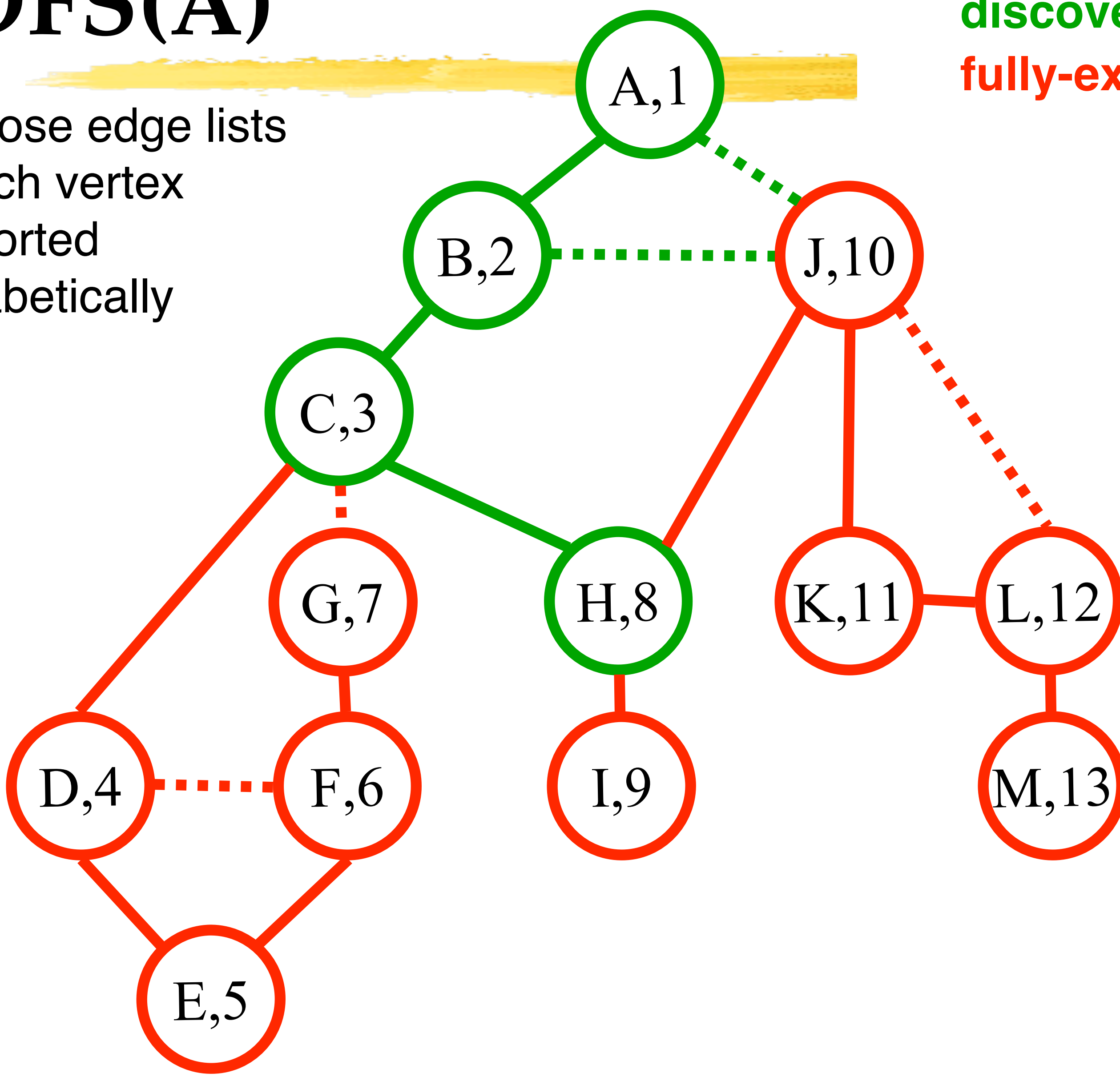
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored

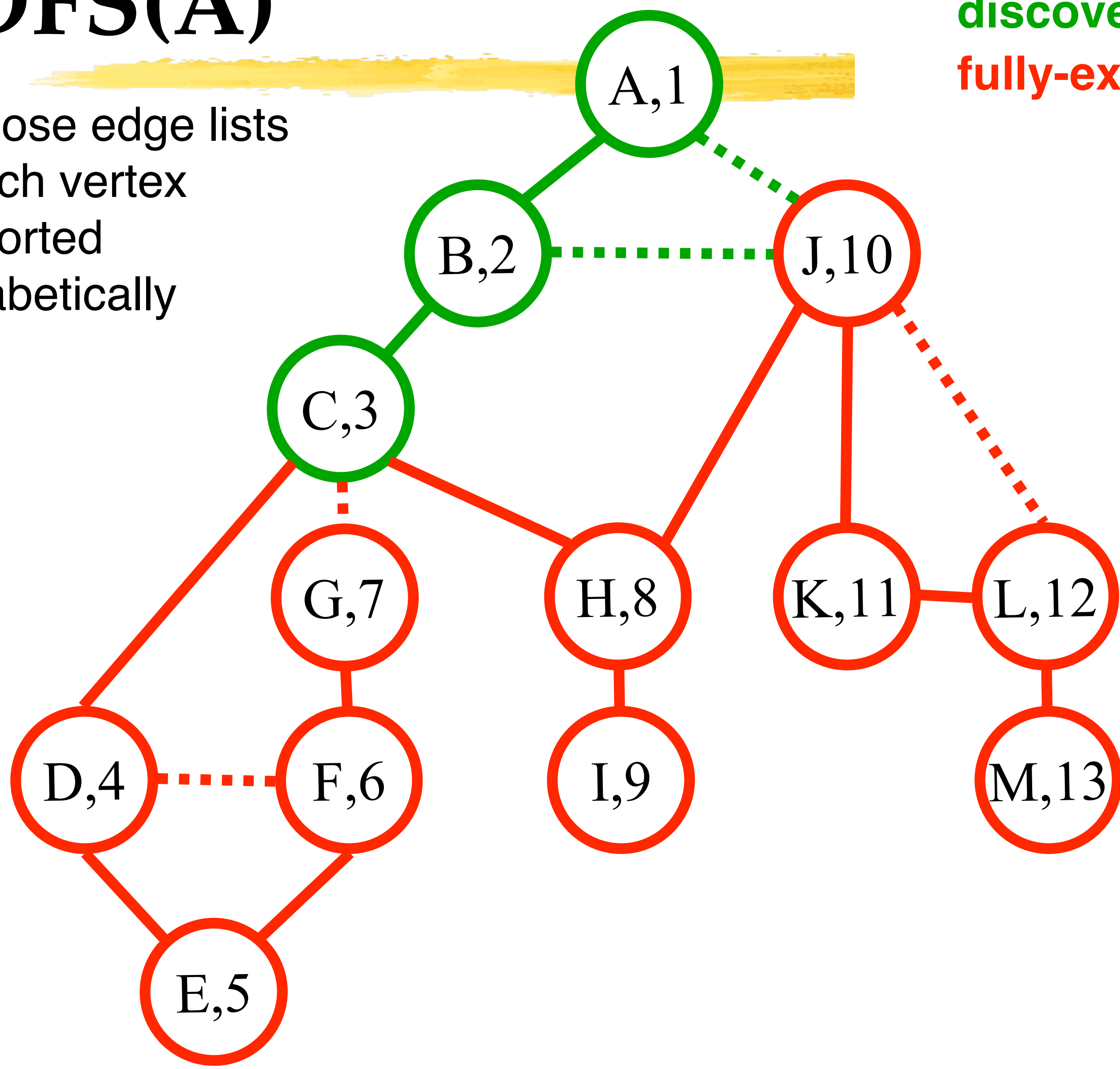


Call Stack:
(Edge list)

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

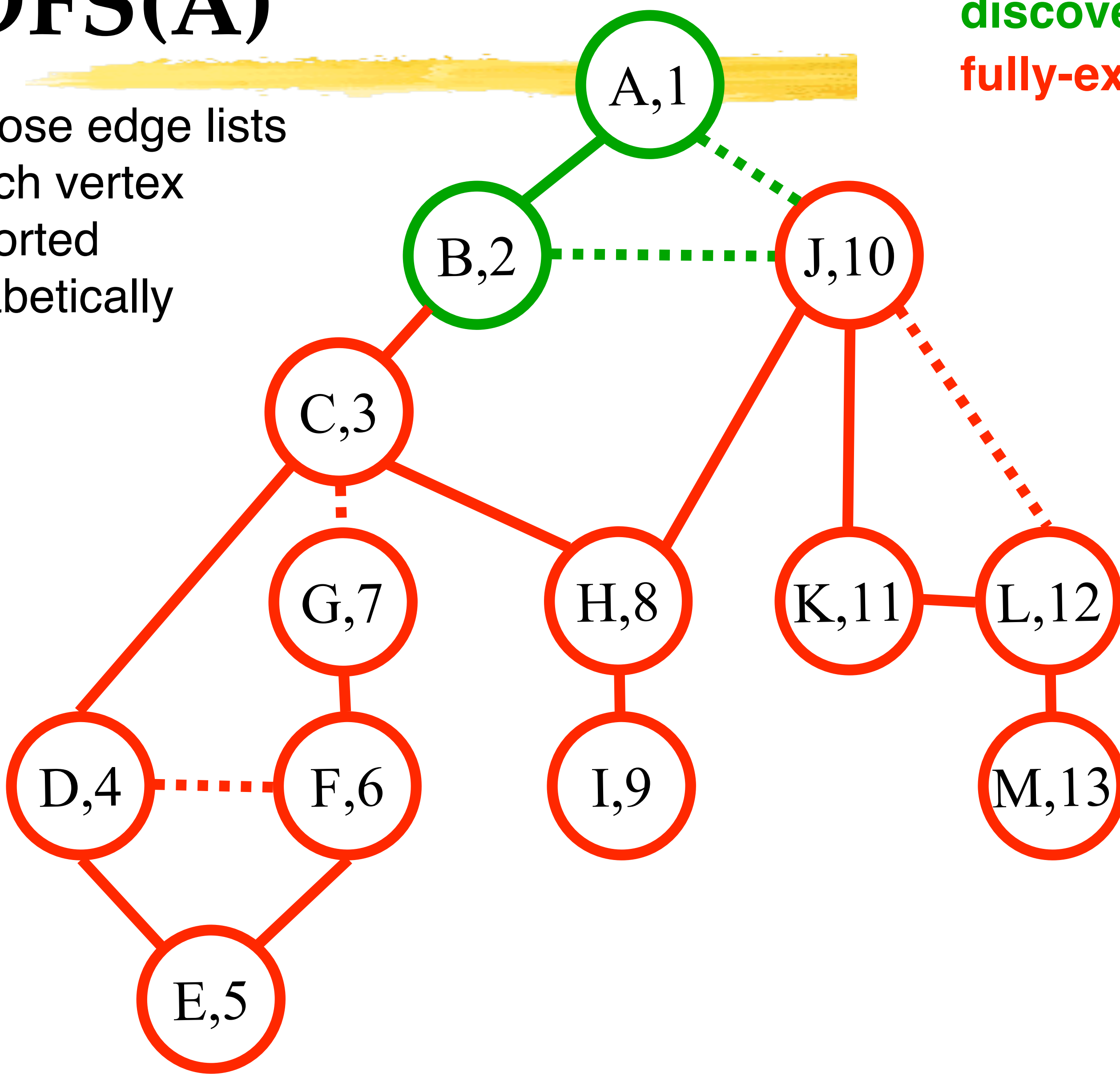
discovered

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)

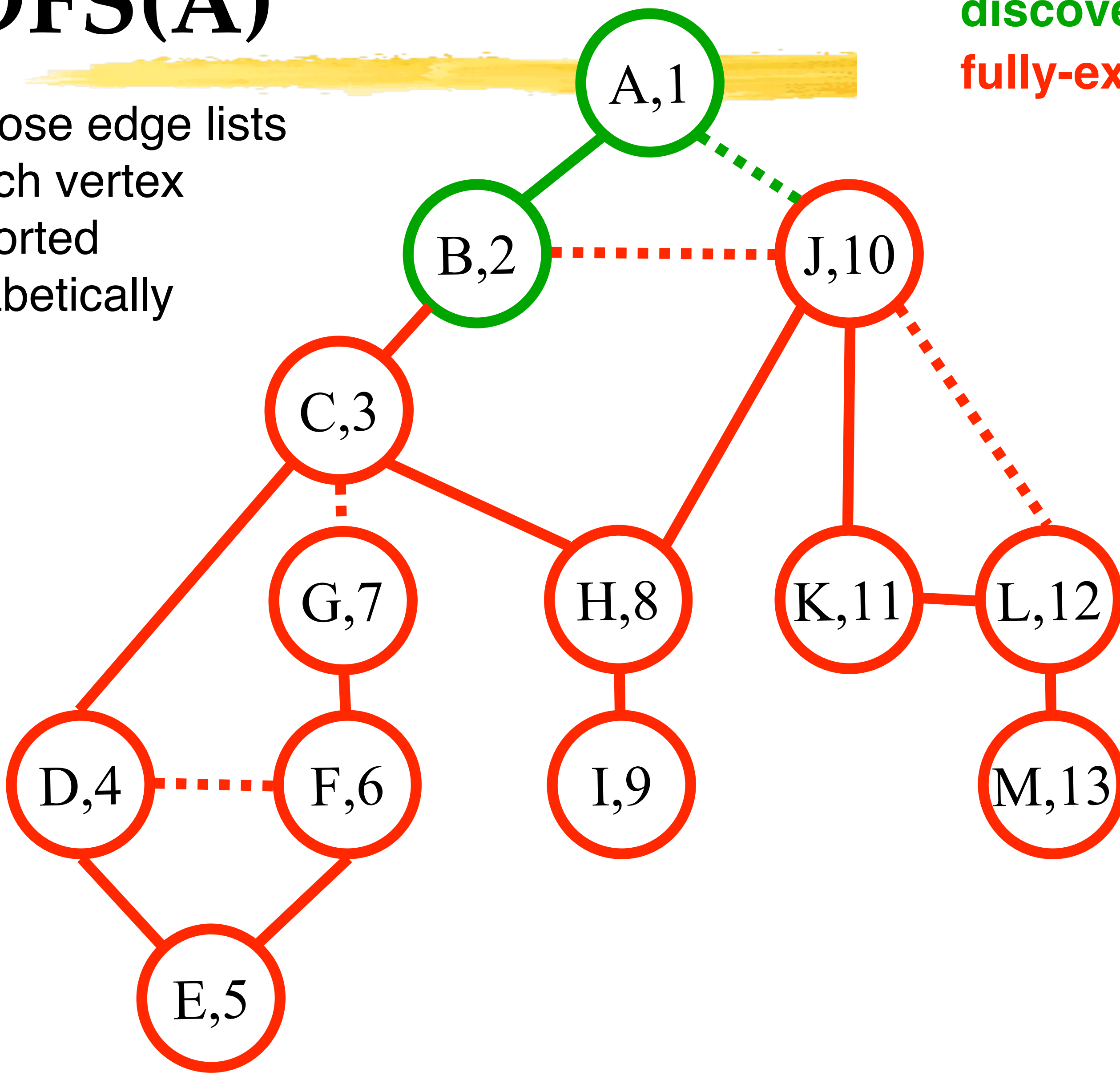
Color code:

undiscovered

discovered

fully-explored

Suppose edge lists at each vertex are sorted alphabetically

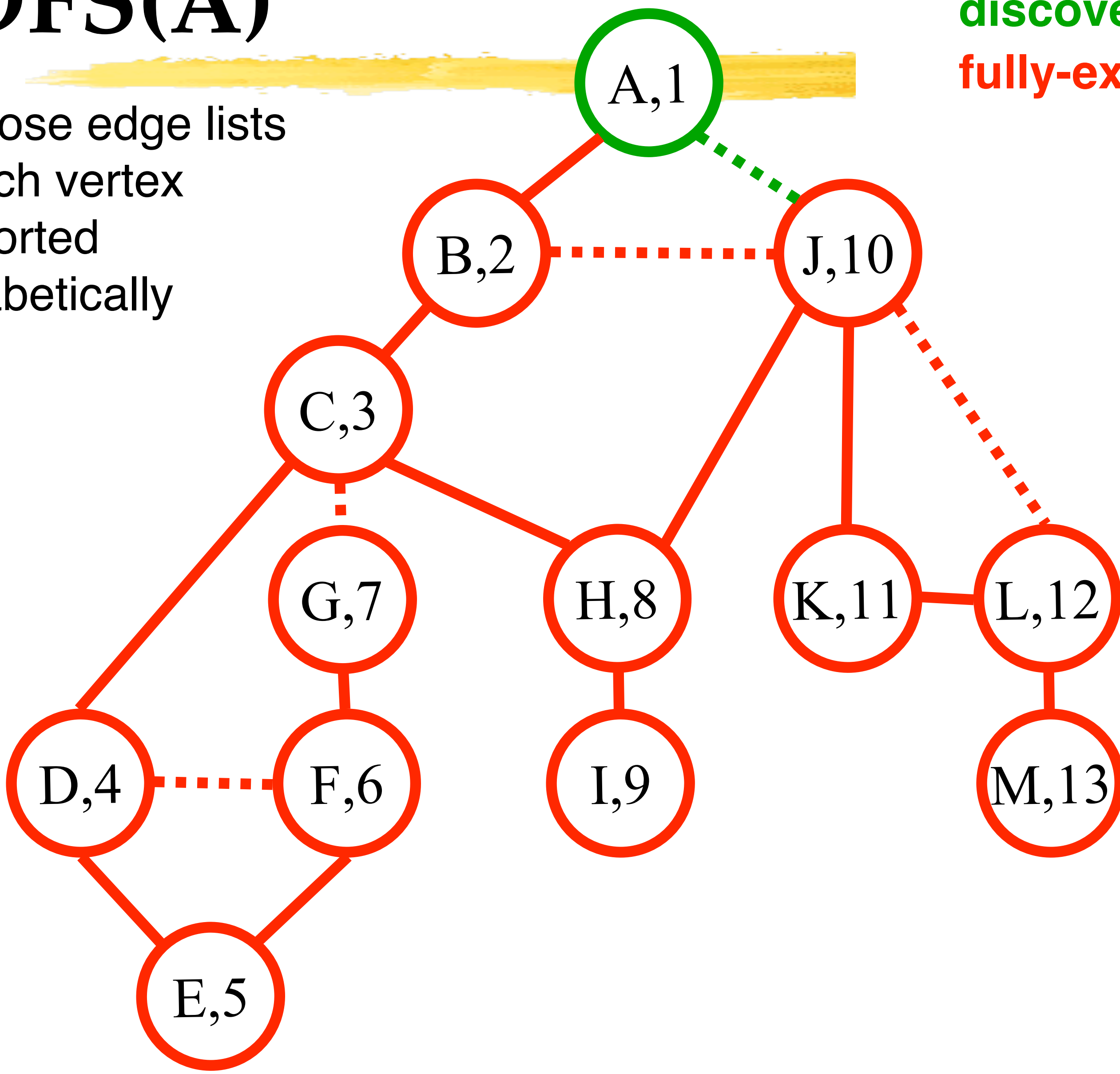


Call Stack:
(Edge list)

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

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)

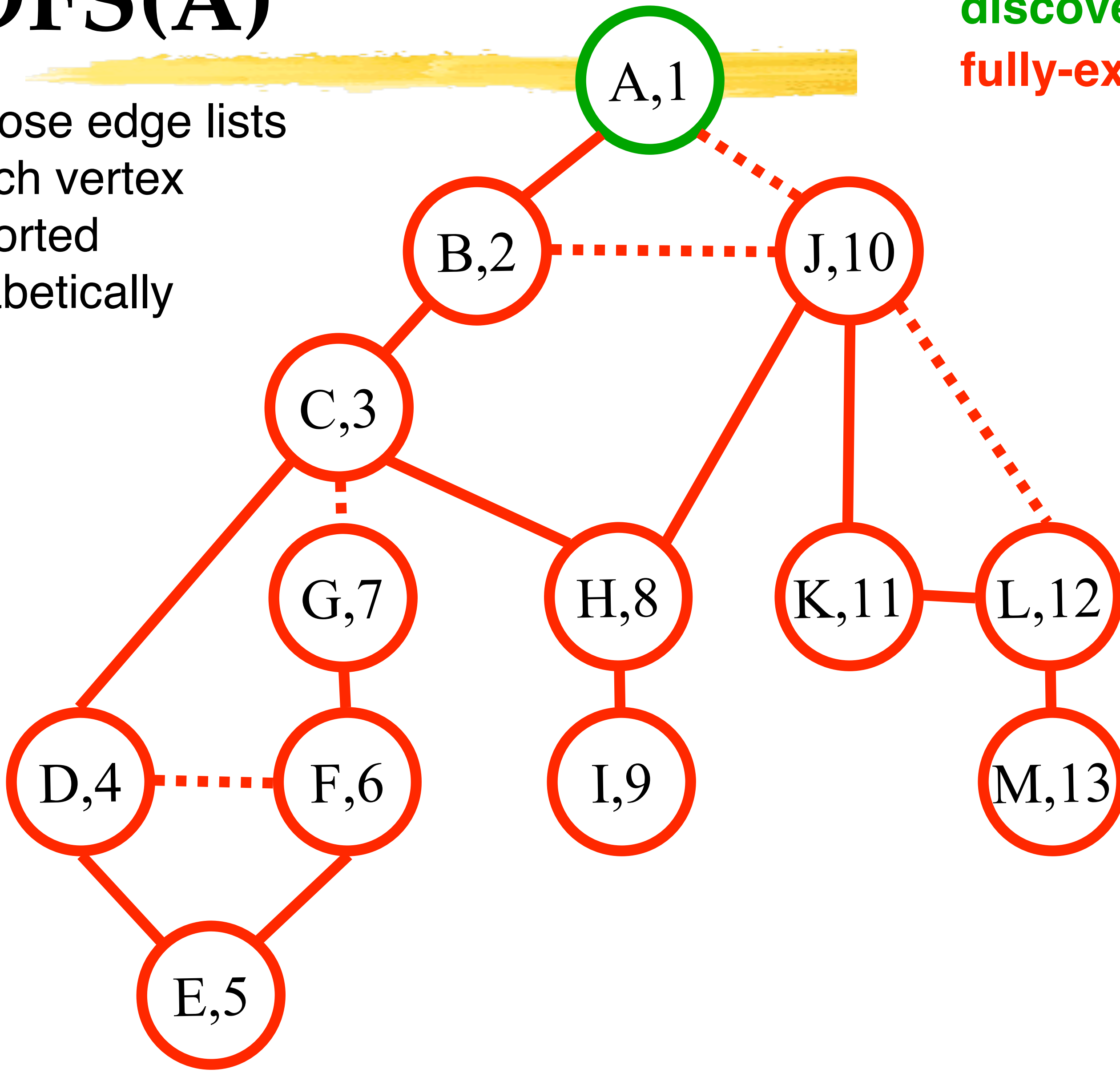
Suppose edge lists at each vertex are sorted alphabetically

Color code:

undiscovered

discovered

fully-explored



Call Stack:
(Edge list)

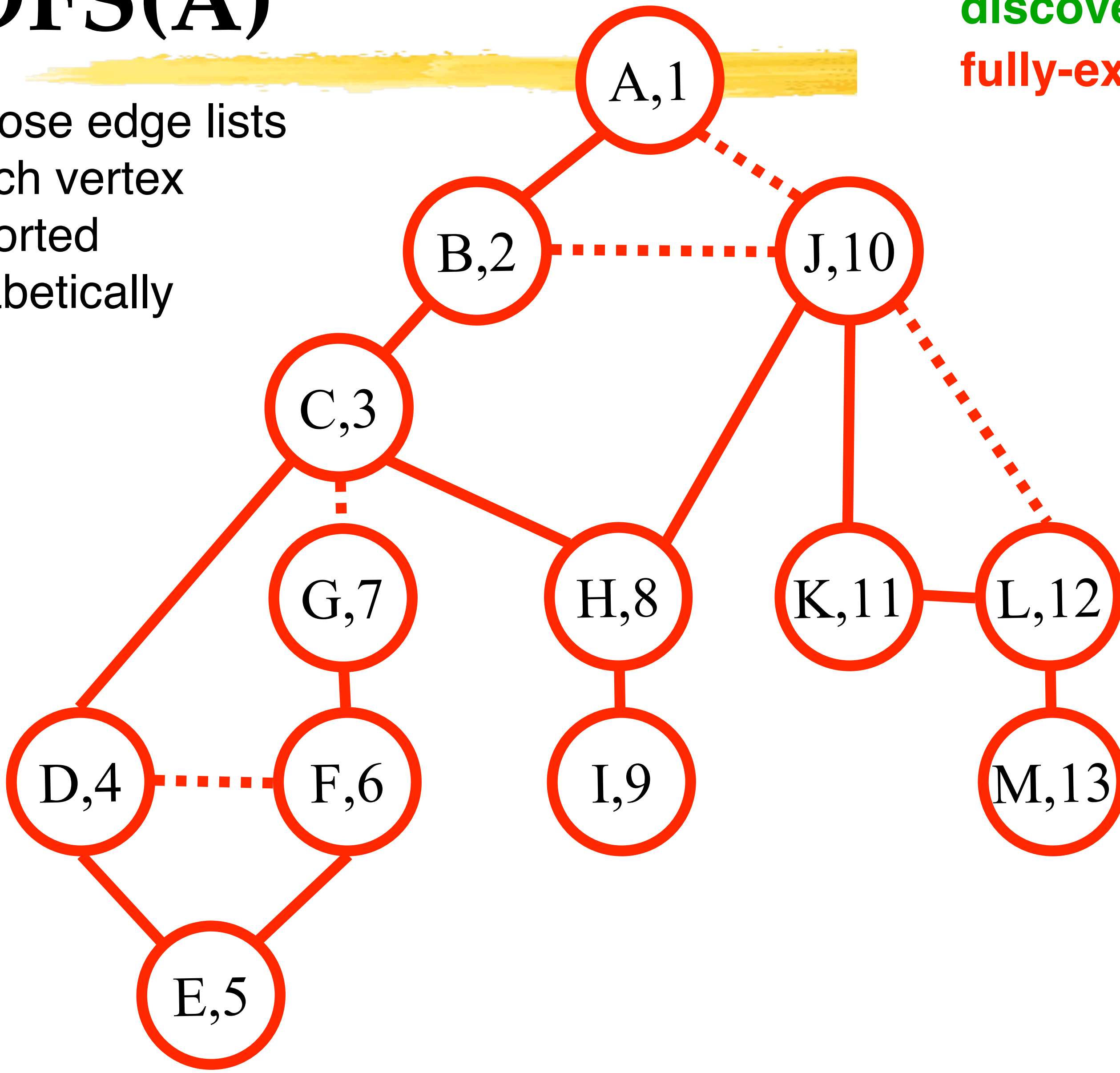
A (~~B~~,~~J~~)

DFS(A)

Color code:

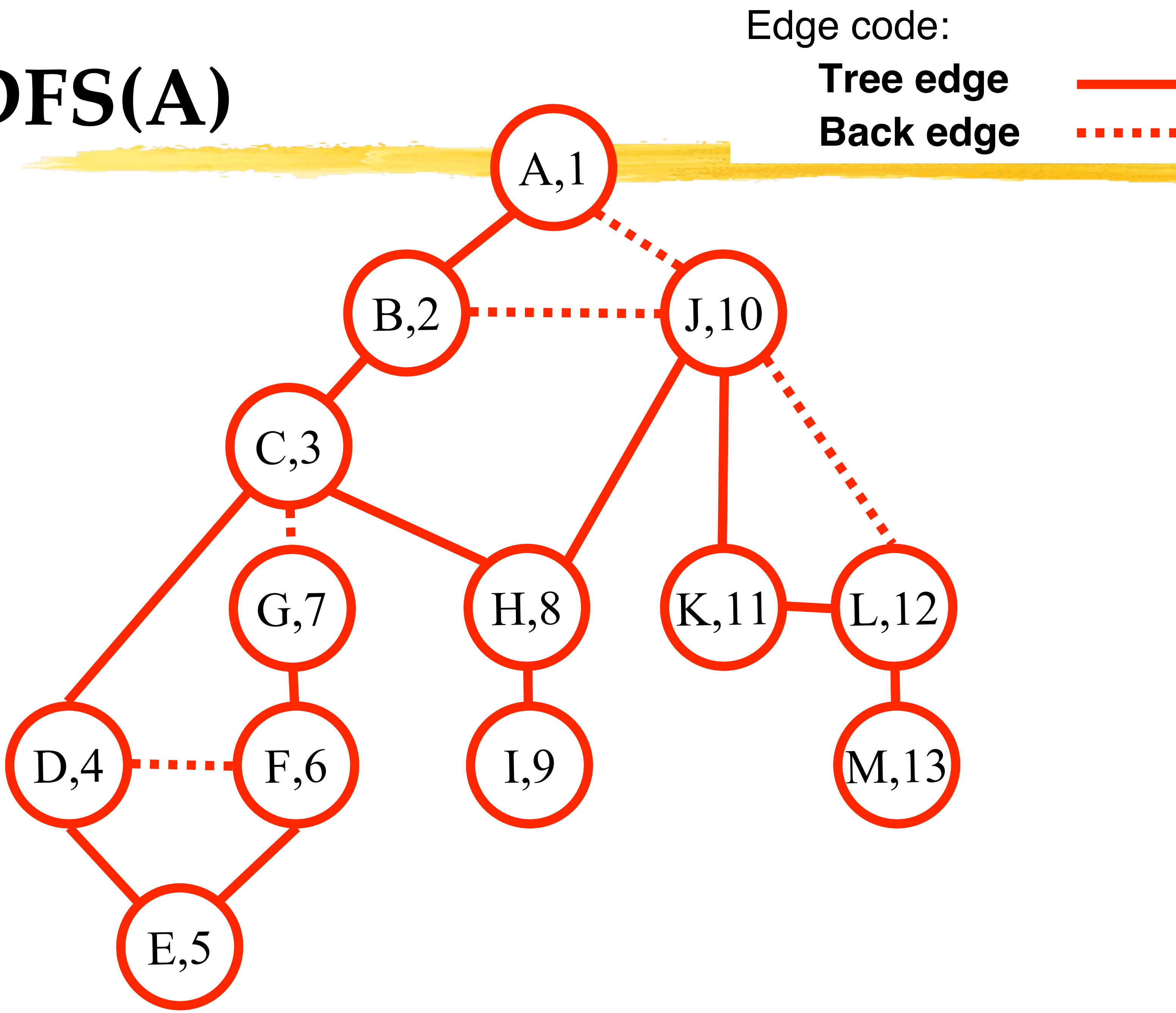
undiscovered
discovered
fully-explored

Suppose edge lists at each vertex are sorted alphabetically

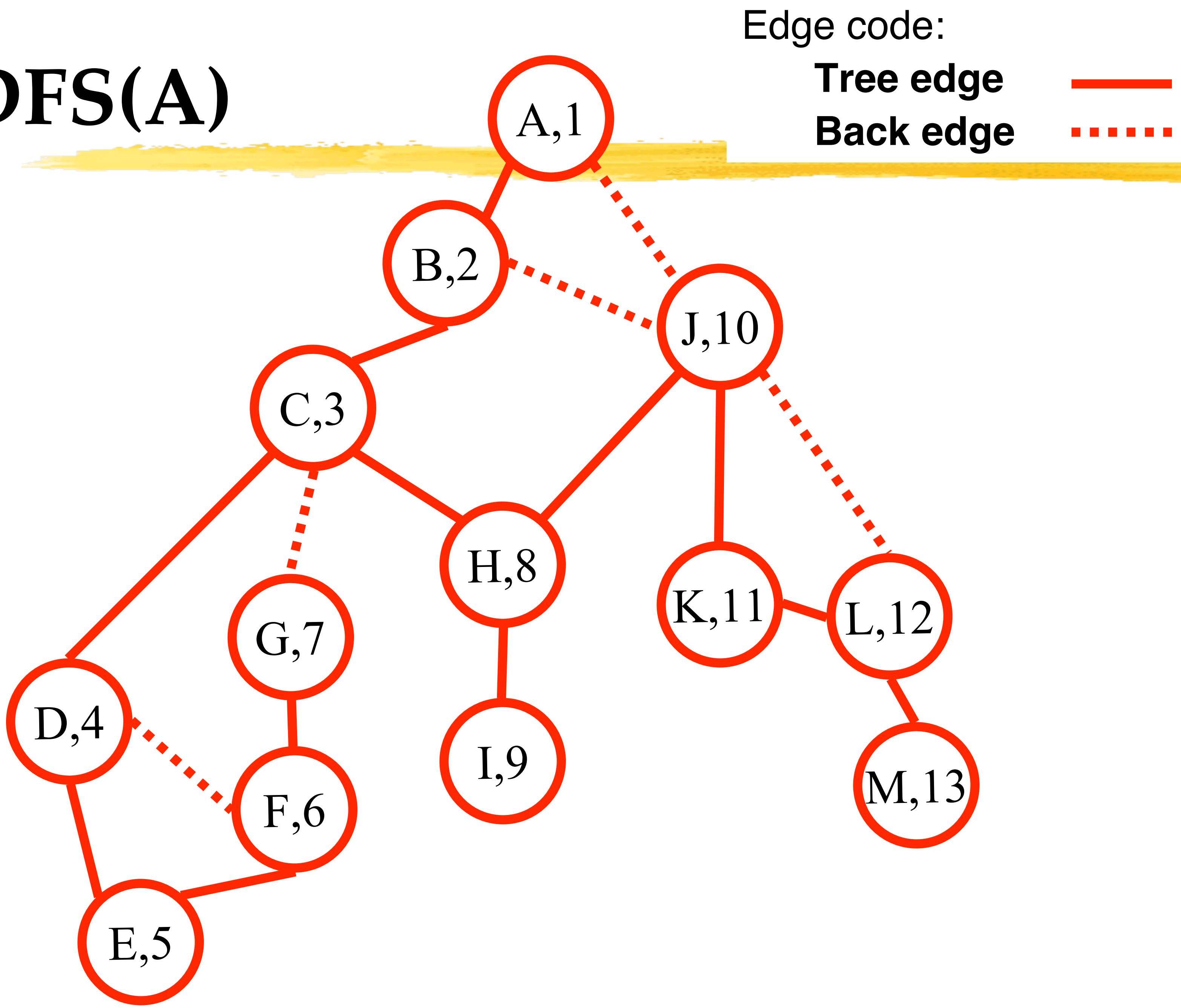


Call Stack: (Edge list)
TA-DA!!

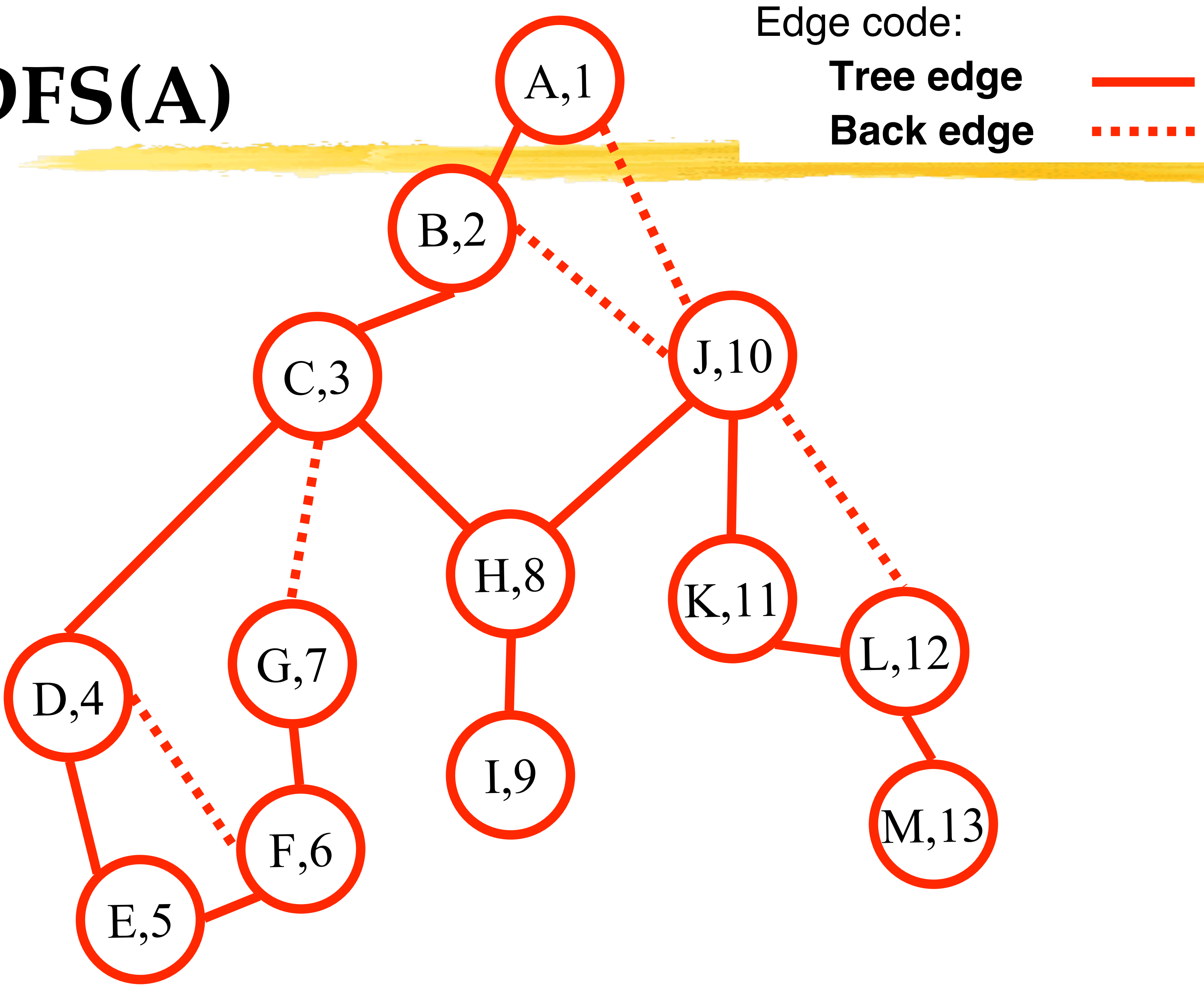
DFS(A)


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)

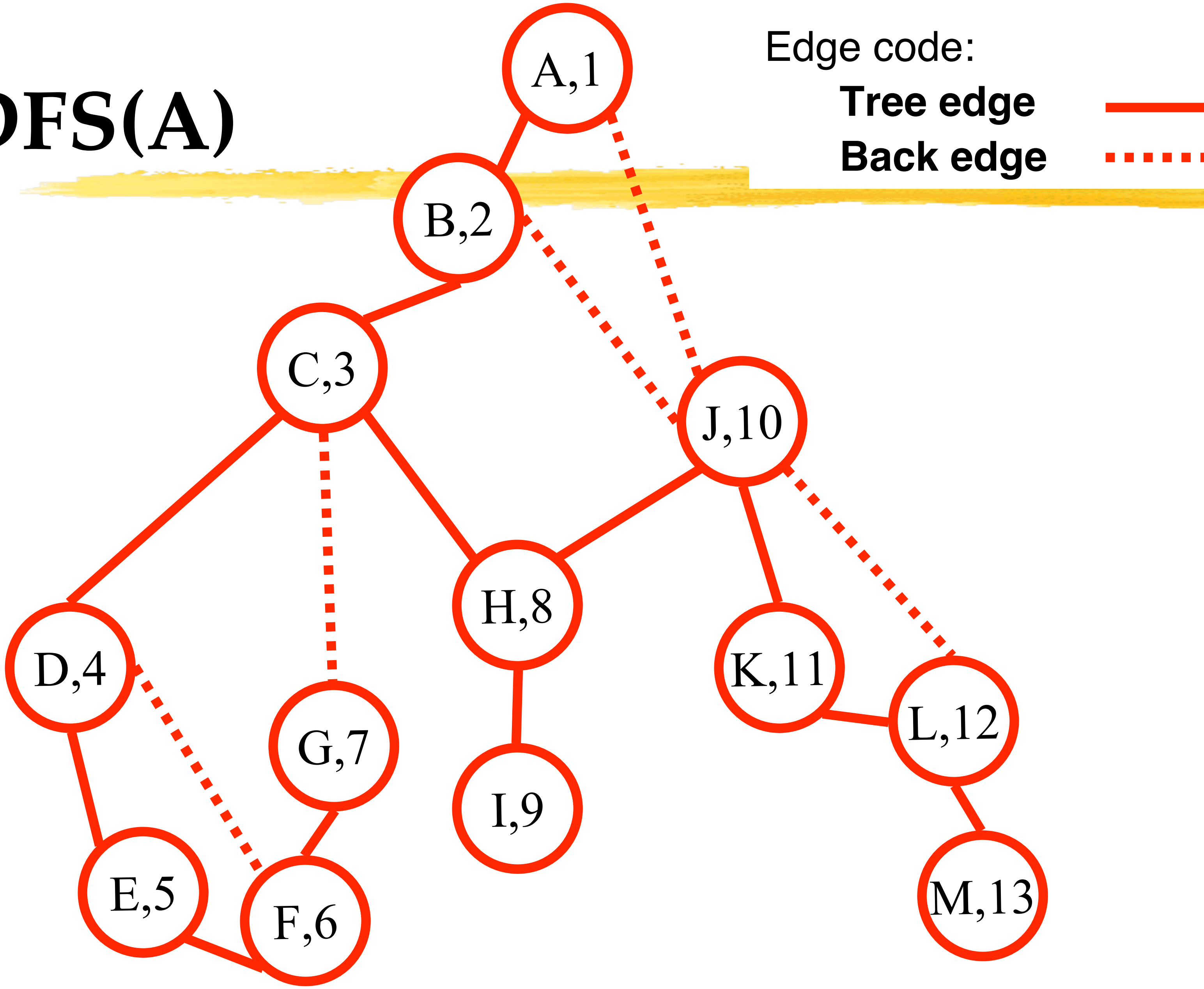

Edge code:
Tree edge
Back edge
A,1
B,2
J,10
C,3
H,8
K,11
L,12
G,7
D,4
F,6
I,9
M,13
E,5

DFS(A)

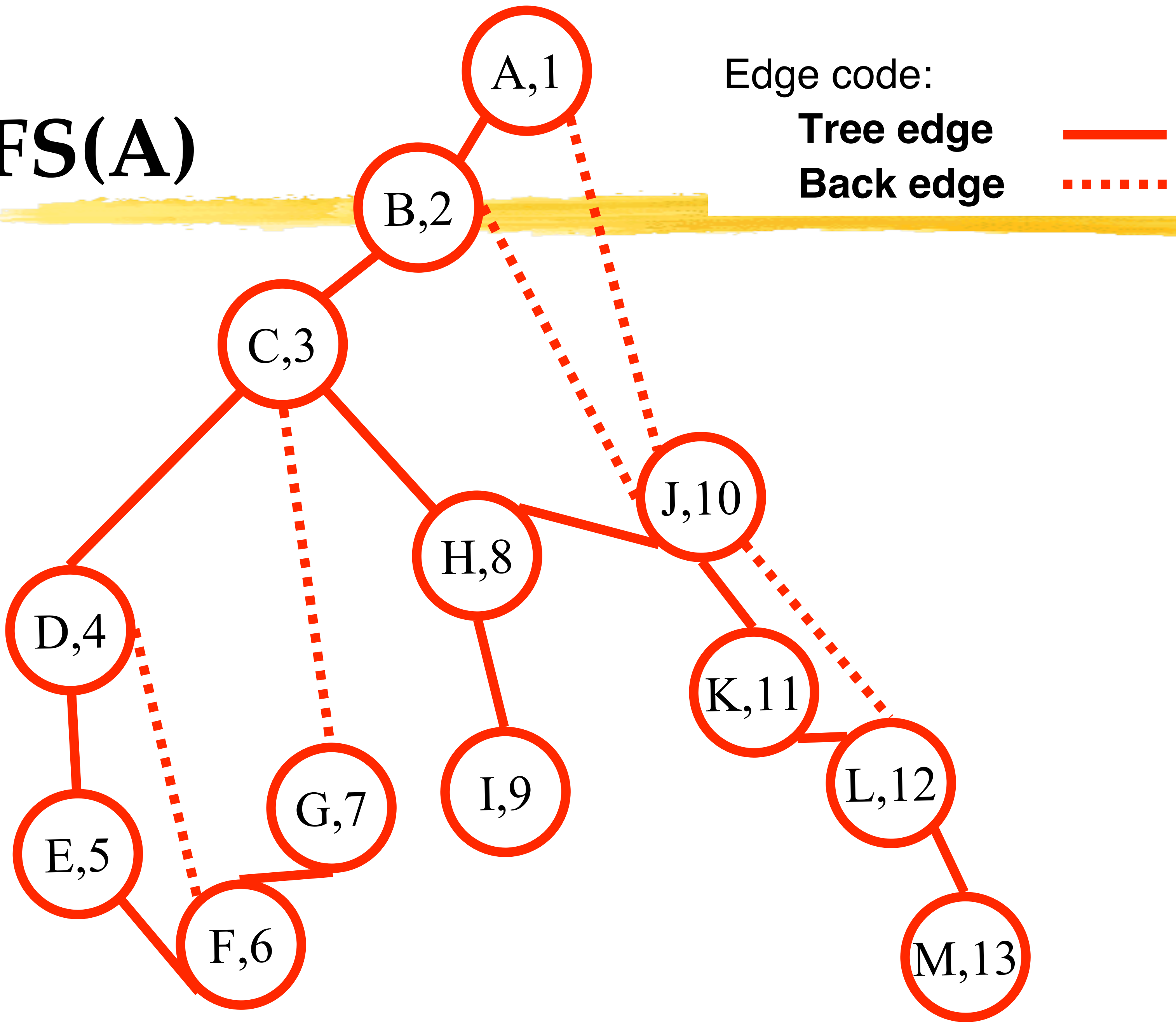


Edge code:
Tree edge
Back edge
A,1
B,2
C,3
J,10
H,8
K,11
L,12
D,4
G,7
I,9
M,13
F,6
E,5

DFS(A)

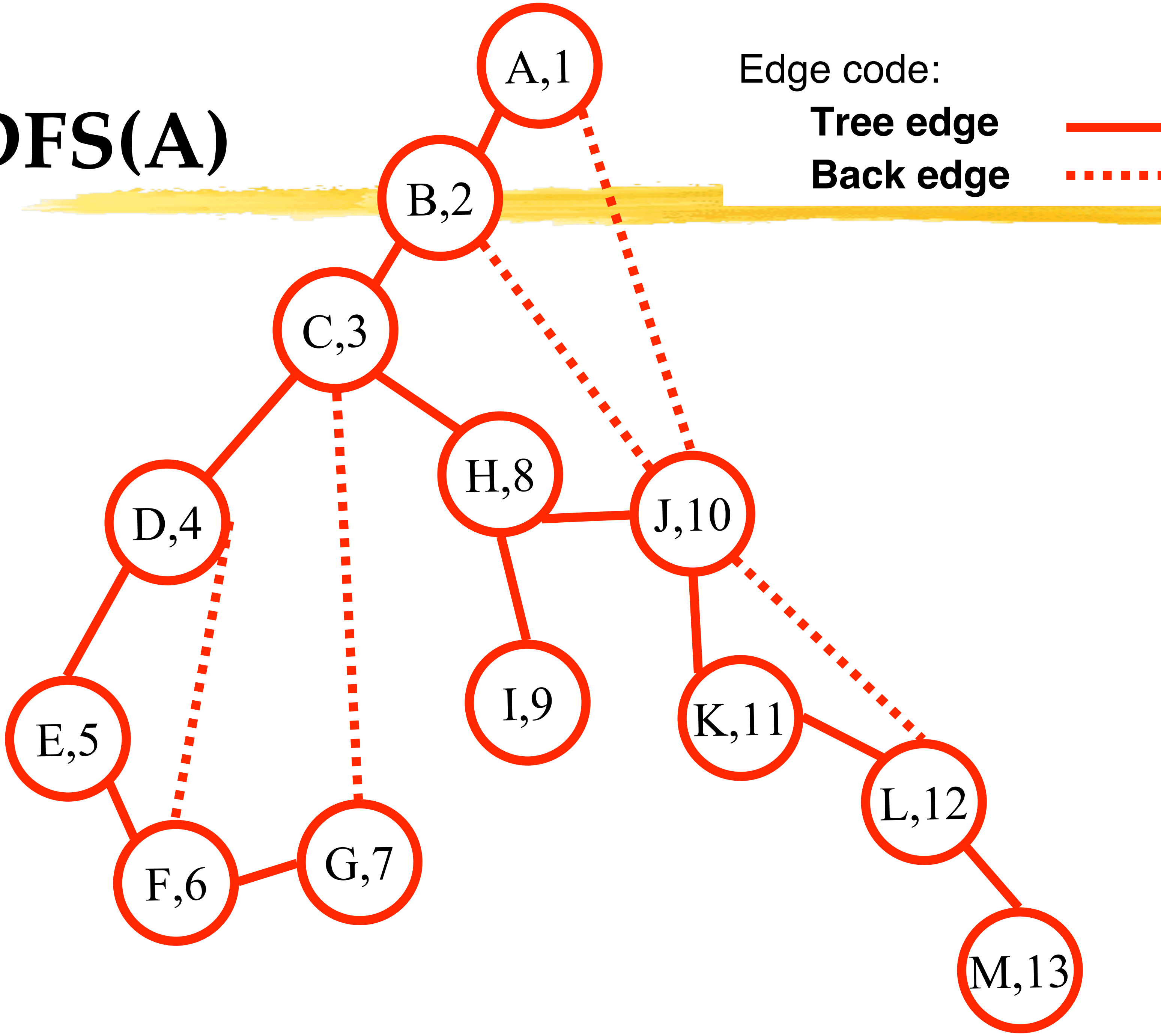

Edge code:
Tree edge
Back edge
A,1
B,2
C,3
D,4
E,5
F,6
G,7
H,8
I,9
J,10
K,11
L,12
M,13

DFS(A)

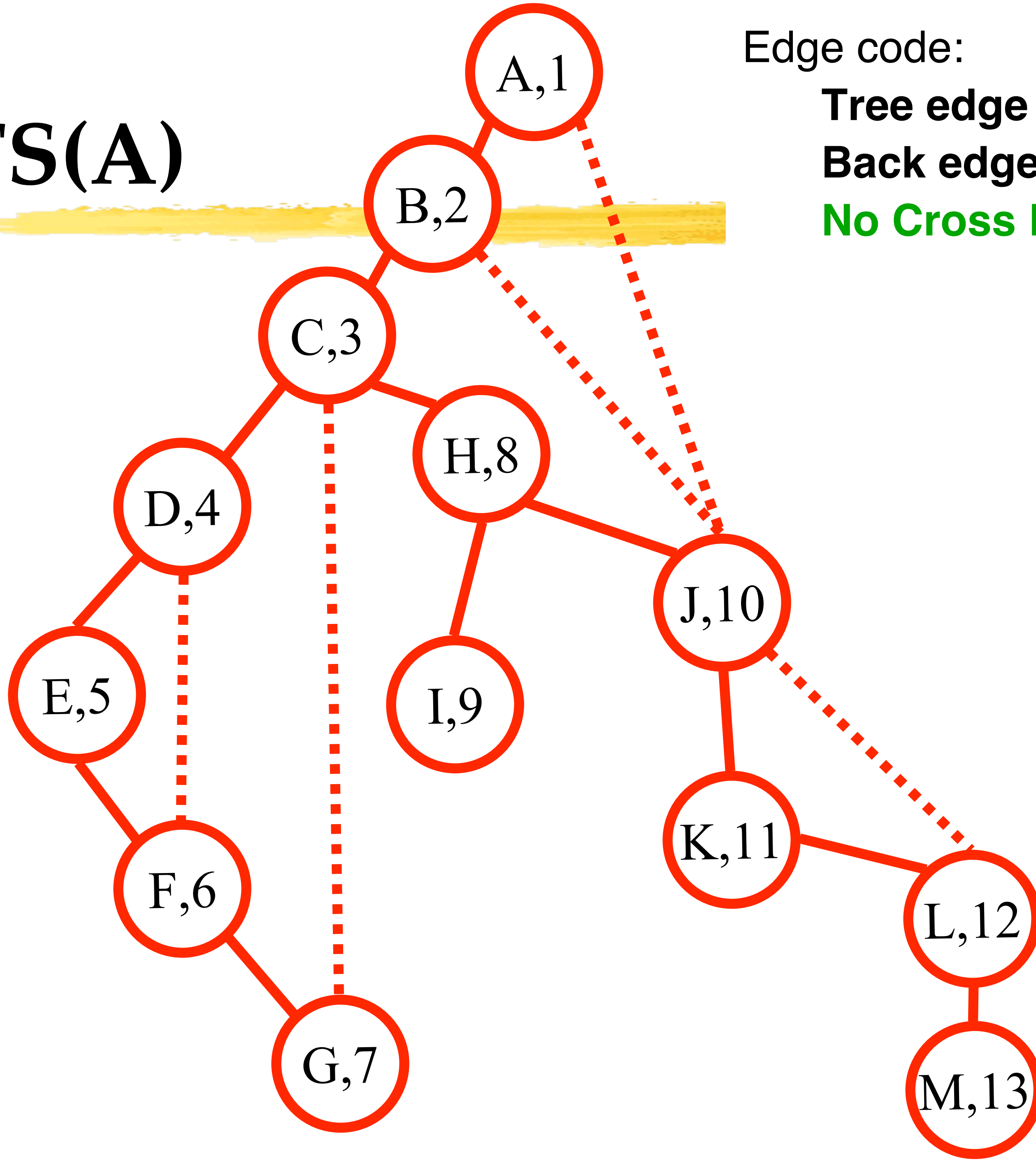


A,1
B,2
C,3
D,4
E,5
F,6
G,7
H,8
I,9
J,10
K,11
L,12
M,13
Edge code:
Tree edge
Back edge

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!

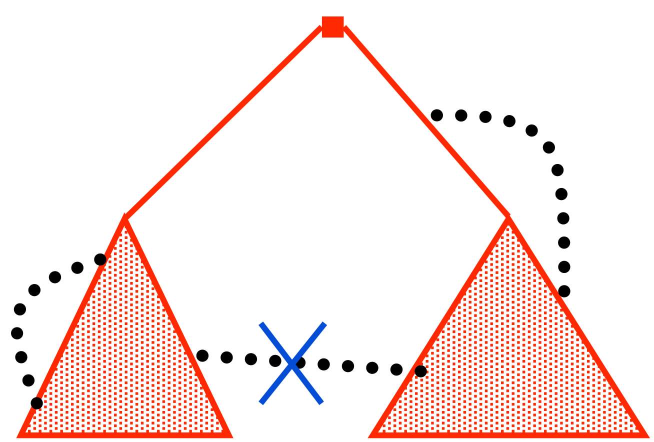
Properties of (Undirected) DFS(v)

- 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:

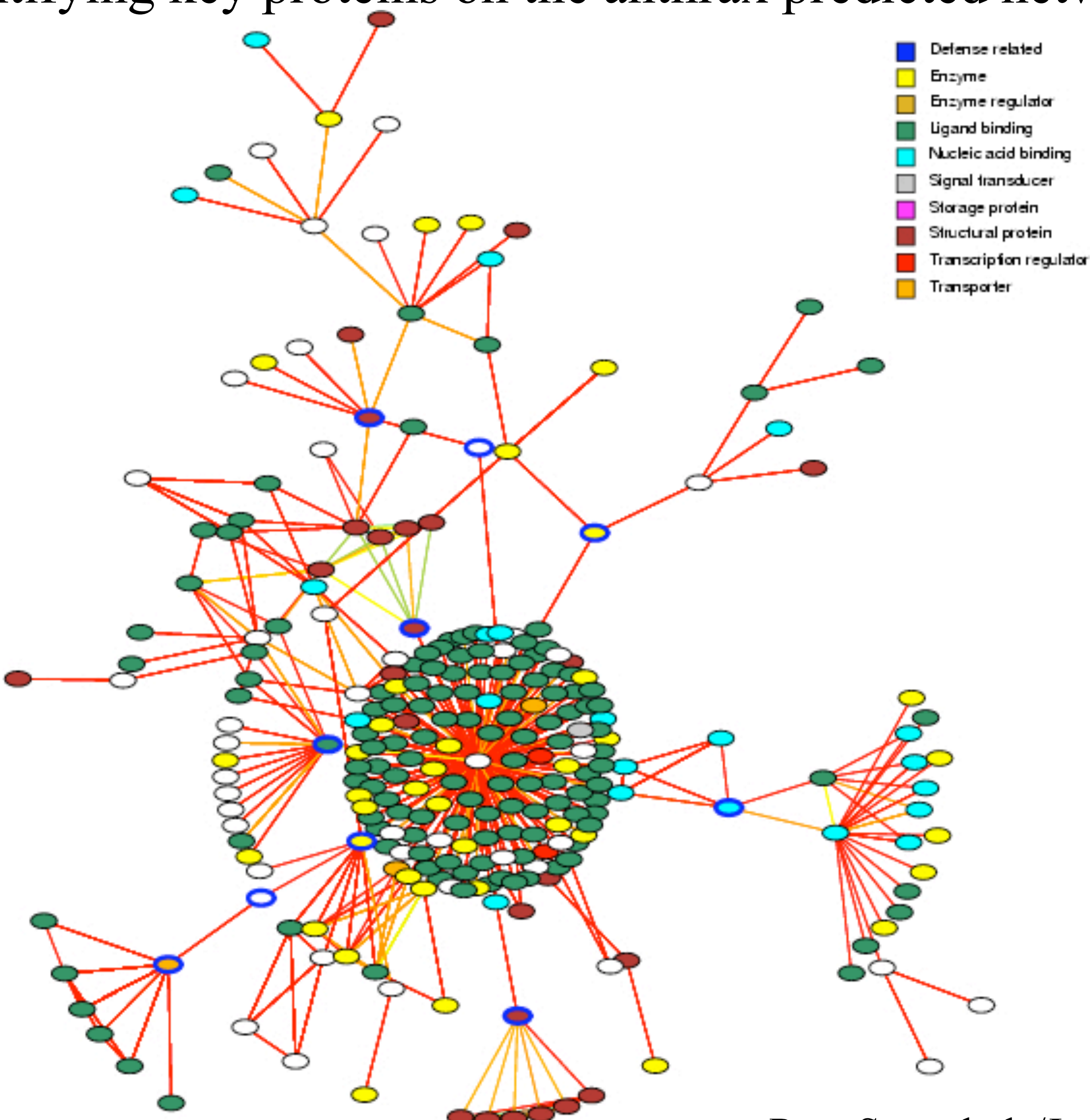
$$M(v) = \begin{cases} L(v) & \text{if } v \text{ is a leaf} \\ \min(L(v),\ \min_{w \text{ a child of v}} M(w)) & \text{otherwise} \end{cases}$$

Application: Articulation Points

- A node in an undirected graph is an **articulation point** iff removing it disconnects the graph

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

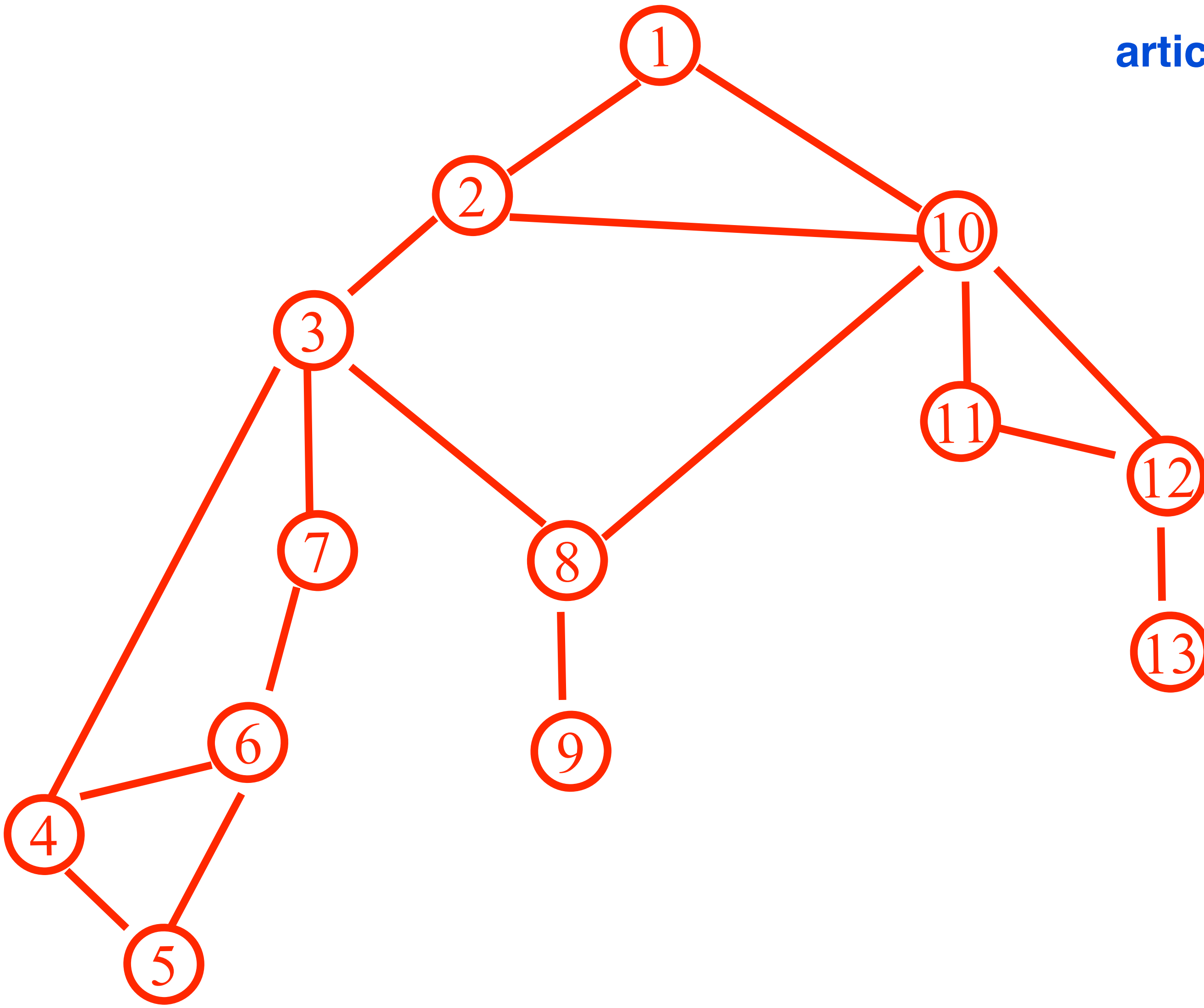
Identifying key proteins on the anthrax predicted network



Articulation point proteins

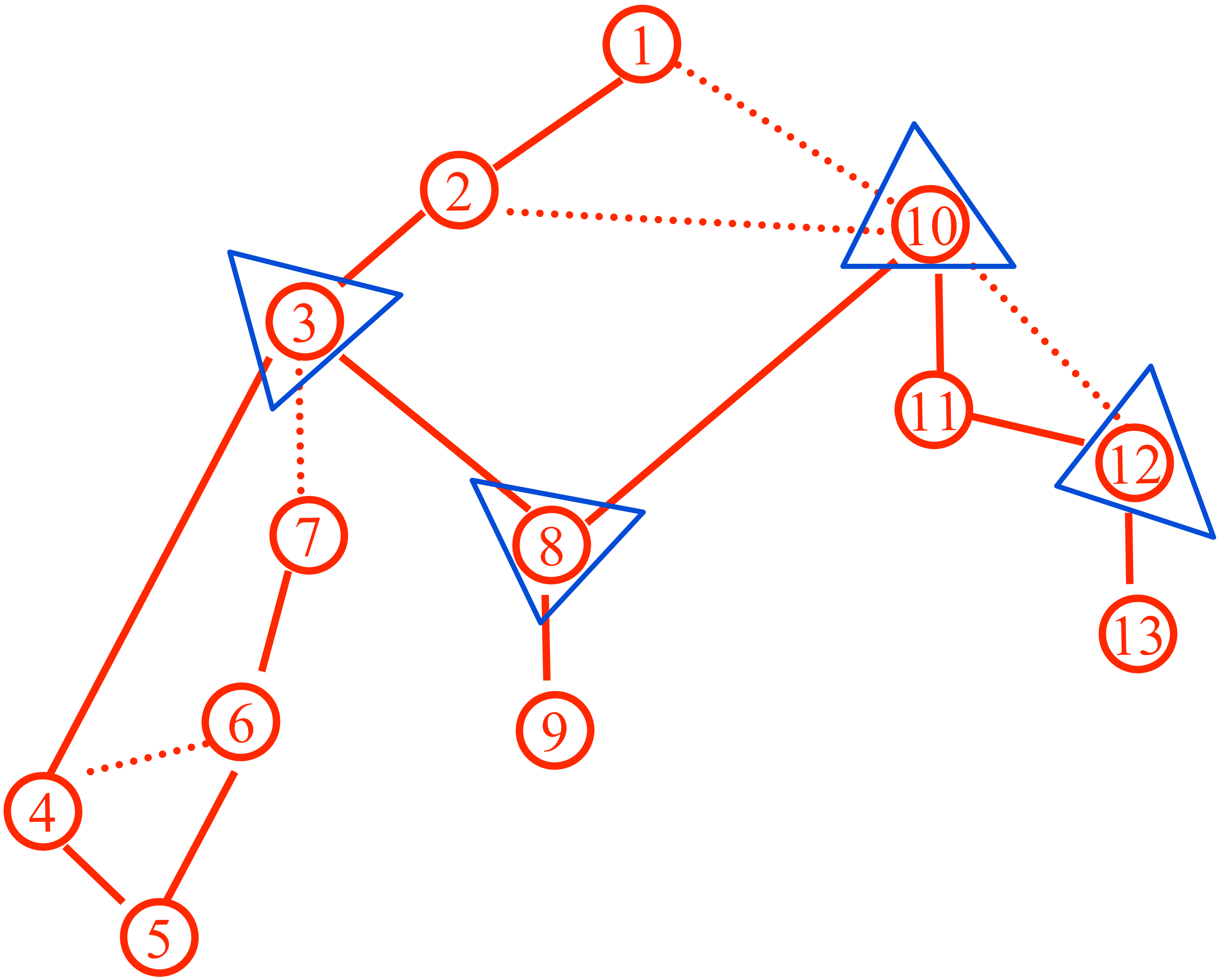
Ram Samudrala/Jason McDermott

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

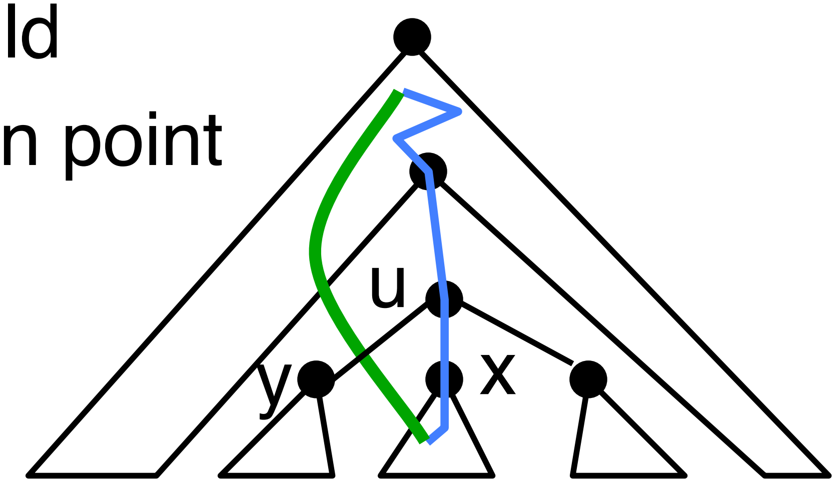
- Non-tree: extra edges remove some articulation points (which ones?)

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 removal of u 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 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 })

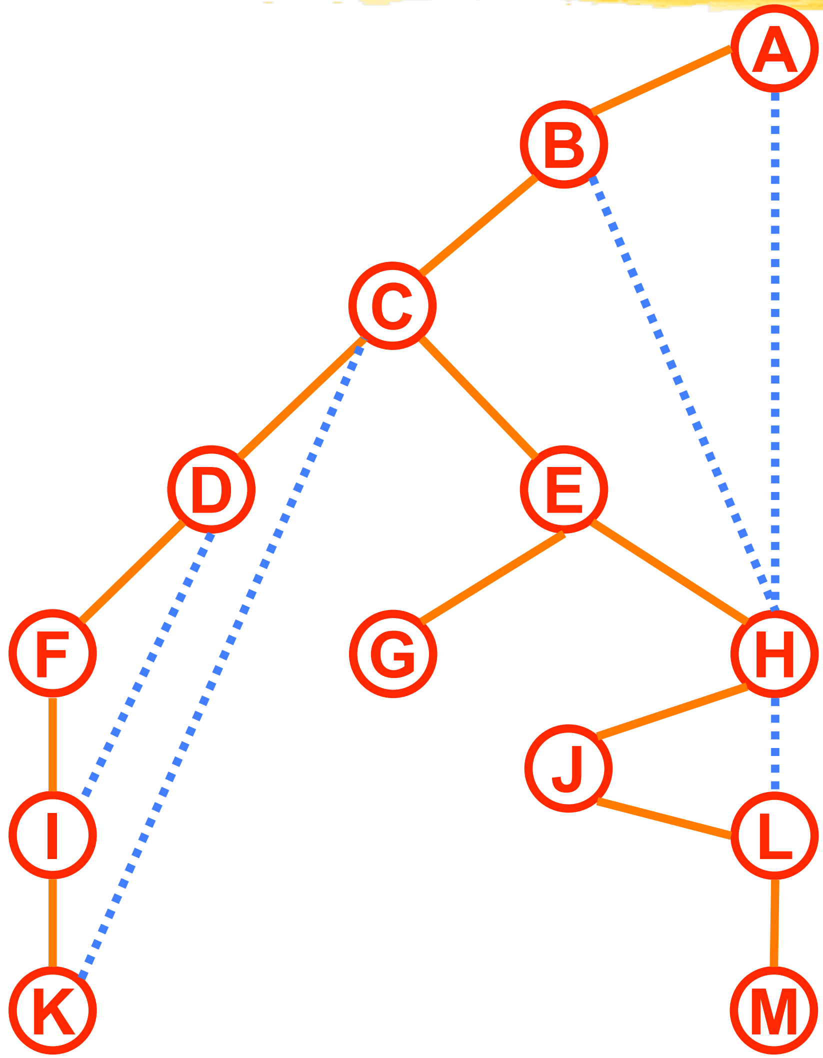
DFS(v) for Finding Articulation Points

```
Global initialization: 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?

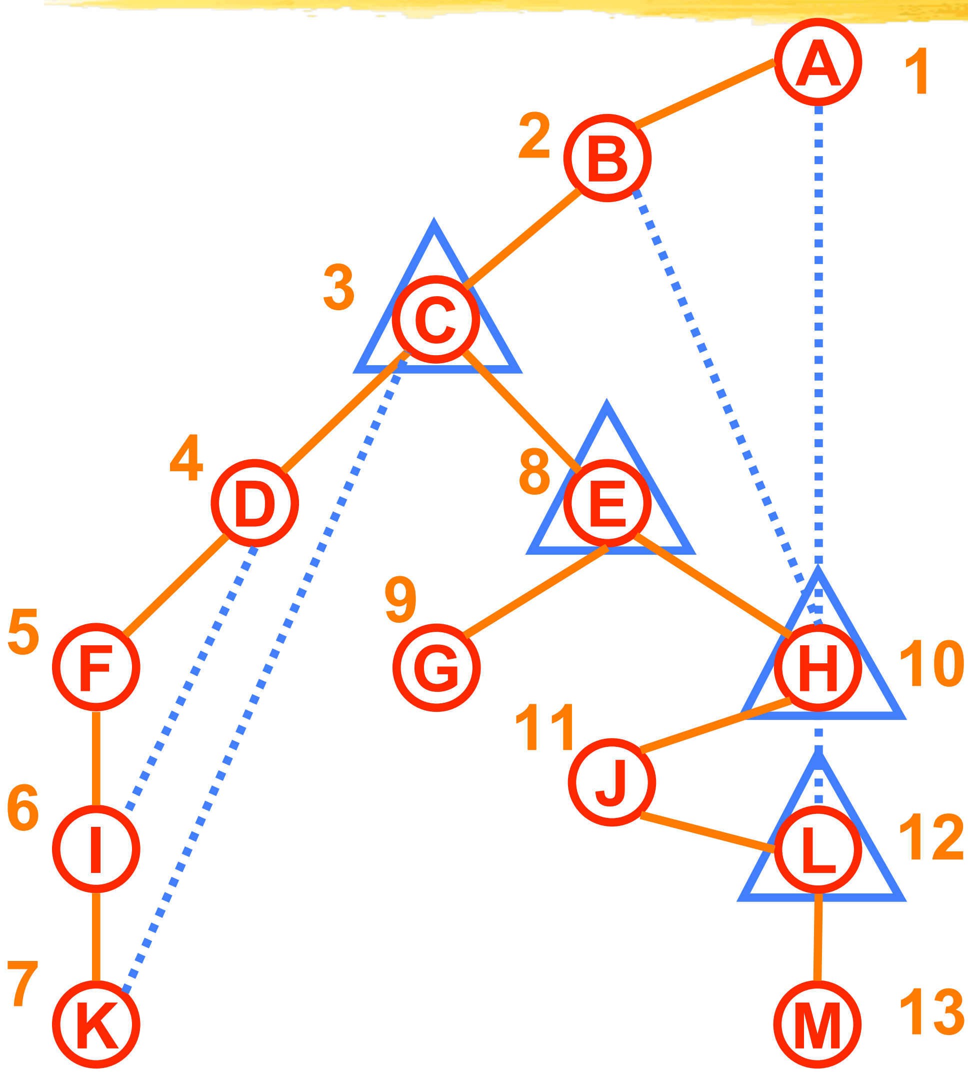
Equiv: "if({v,x} is a back edge)" Why?

Articulation Points



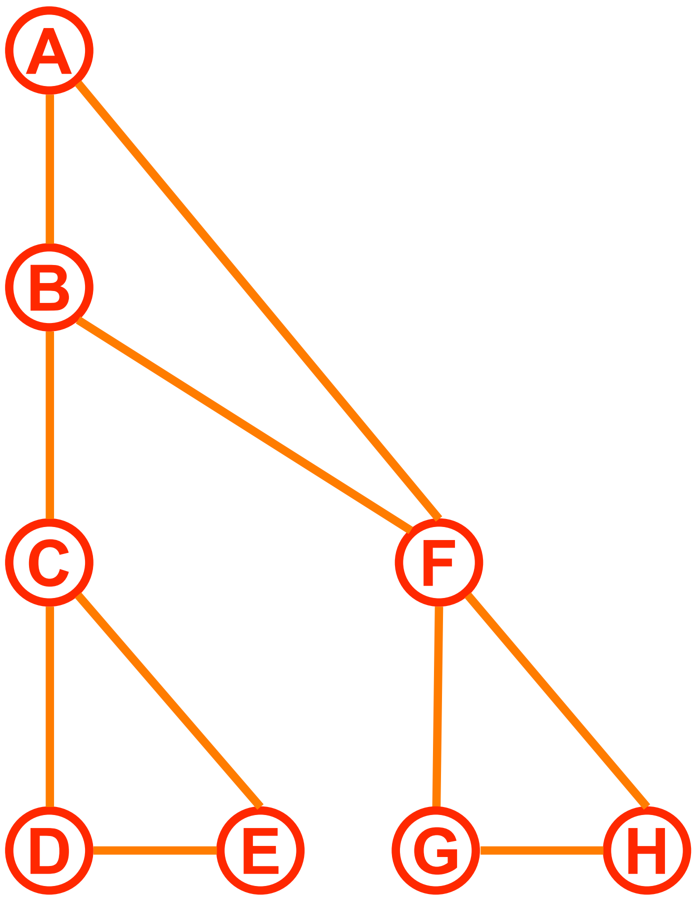
Vertex	DFS #	Low
A		
B		
C		
D		
E		
F		
G		
H		
I		
J		
K		
L		
M		

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

Articulation Points



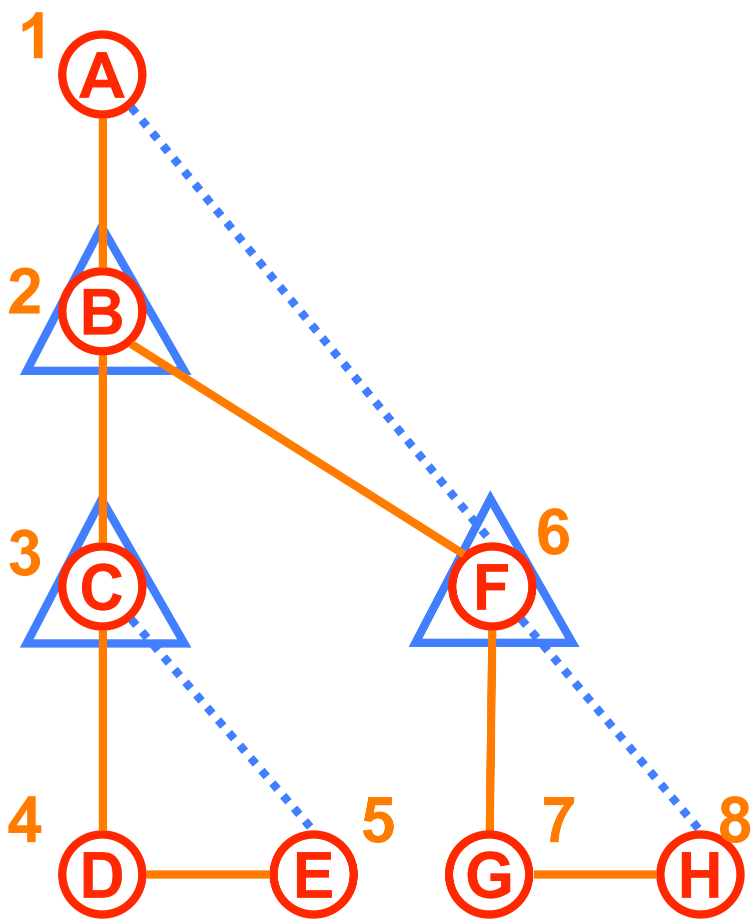
Vertex	DFS #	Low
A		
B		
C		
D		
E		
F		
G		
H		

AP's:

BCC's:

1)
2)
3)
4)

Articulation Points



Vertex	DFS #	Low
A	1	1
B	2	1
C	3	3
D	4	3
E	5	3
F	6	1
G	7	6
H	8	6

AP's: C, B, F

BCC's:

1) C--D, D--E, E--C
2) B--C
3) A--B, B--F, F--A
4) F--G, G--H, H--F