Graphs and Graph Algorithms

Slides by Larry Ruzzo

Goals

Graphs: defns, examples, utility, terminology

Representation: input, internal

Traversal: Breadth- & Depth-first search

Three Algorithms:

- Connected components
- Bipartiteness
- Topological sort

Objects & Relationships

The Kevin Bacon Game:

- Obj: Actors
- Rel: Two are related if they've been in a movie together

Exam Scheduling:

- Obj: Classes
- Rel: Two are related if they have students in common

Traveling Salesperson Problem:

- Obj: Cities
- Rel: Two are related if can travel *directly* between them

Graphs

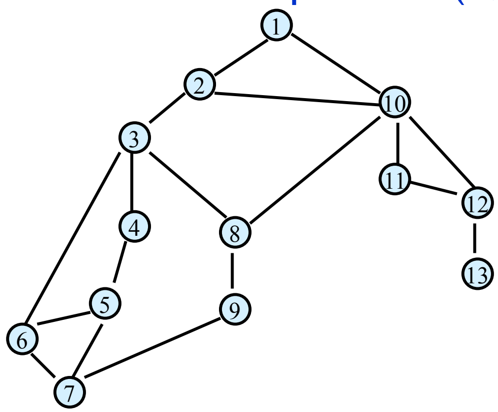
An extremely important formalism for representing (binary) relationships

Objects: "vertices," aka "nodes"

Relationships between pairs: "edges," aka "arcs"

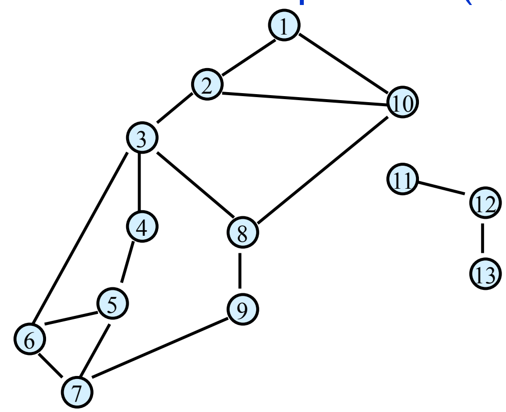
Formally, a graph G = (V, E) is a pair of sets, V the vertices and E the edges

Undirected Graph G = (V,E)



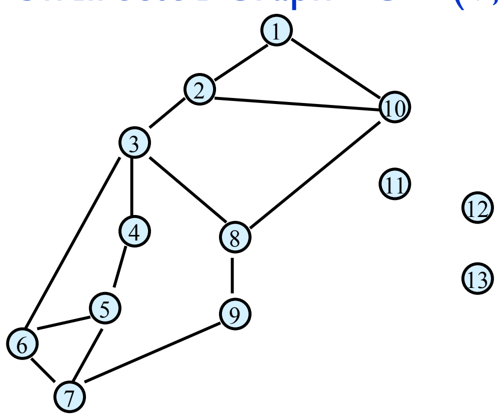
6

Undirected Graph G = (V,E)



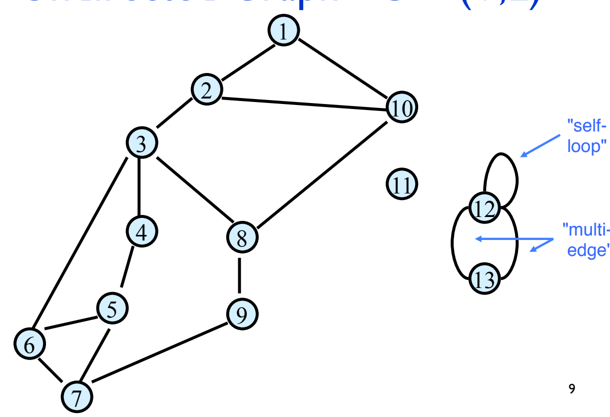
7

Undirected Graph G = (V,E)



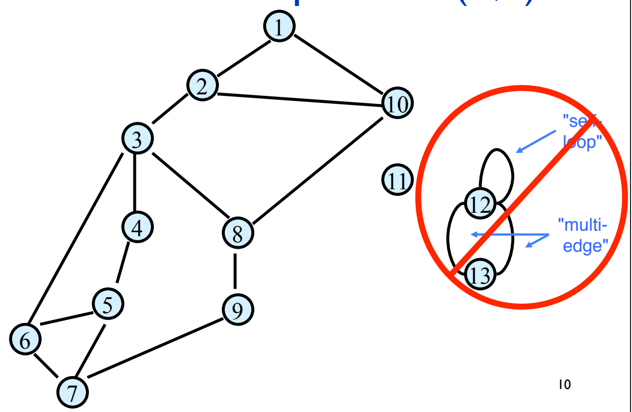
8

Undirected Graph G = (V,E)



9

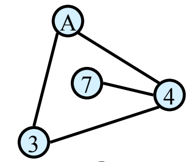
Undirected Graph G = (V,E)

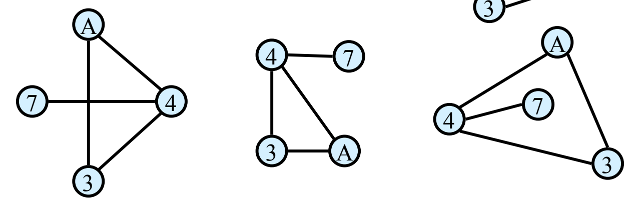


10

Graphs don't live in Flatland

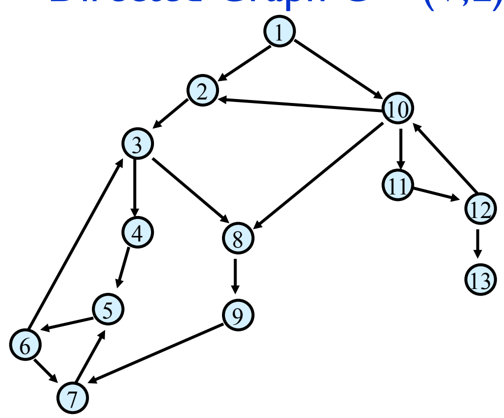
Geometrical drawing is mentally convenient, but mathematically irrelevant: 4 drawings, 1 graph.





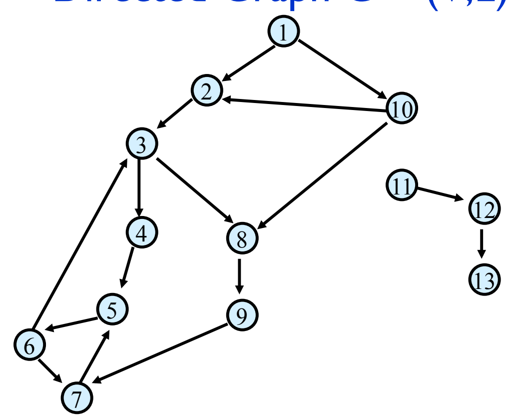
11

Directed Graph G = (V,E)



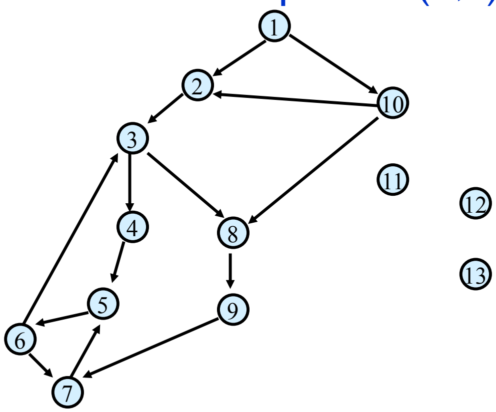
12

Directed Graph G = (V,E)



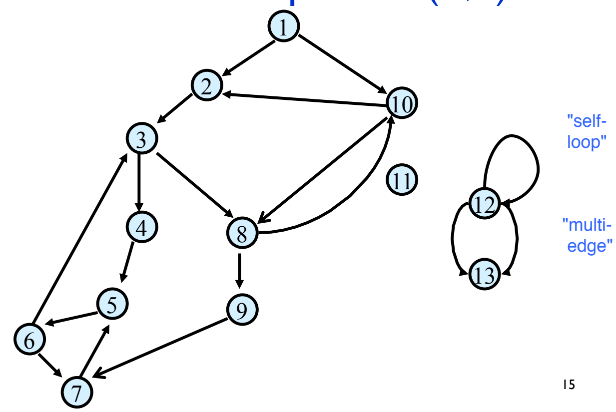
13

Directed Graph G = (V,E)



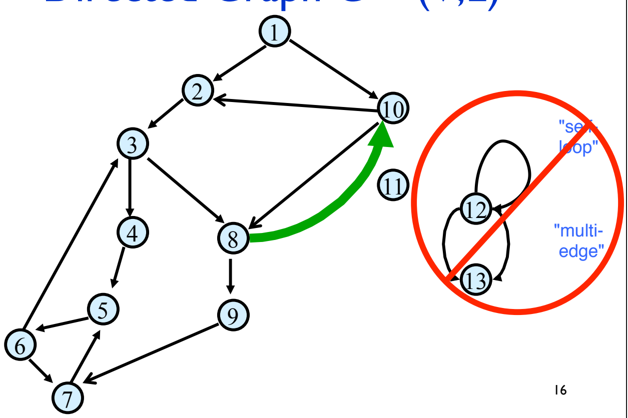
14

Directed Graph G = (V,E)



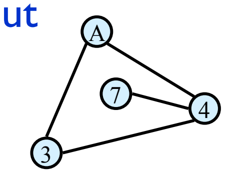
15

Directed Graph G = (V,E)



16

Specifying undirected graphs as input



What are the vertices?

Explicitly list them:
{"A", "7", "3", "4"}

What are the edges?

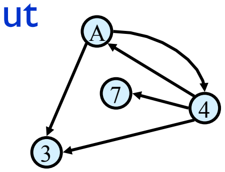
Either, set of edges
{{A,3}, {7,4}, {4,3}, {4,A}}

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

17

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

18

Vertices vs # Edges

Let G be an undirected graph with n vertices and m edges. How are n and m related?

19

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:

$0 \le m \le n(n-1)/2 = O(n^2)$

20

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 ($m \le 3n-6$, for $n \ge 3$)

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!

21

Representing Graph G = (V,E)

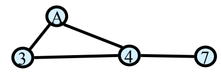
internally, indp of input format

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



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

Advantages:

$O(1)$ test for presence or absence of edges.

Disadvantages: inefficient for sparse graphs, both in storage and access

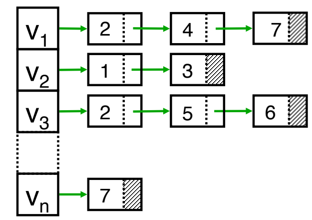
$m \ll n^2$

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

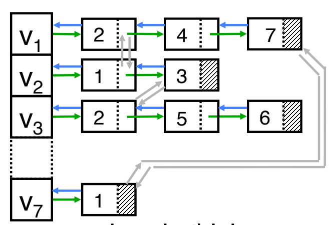
no $O(1)$ edge test

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 *s* to all vertices reachable from *s*

Being *orderly* helps. Two common ways:

Breadth-First Search: order the nodes in successive layers based on distance from s

Depth-First Search: more natural approach for exploring a maze; many efficient algs build on it.

Breadth-First Search

Completely explore the vertices in order of their distance from s

Naturally implemented using a queue

Graph Traversal: Implementation

Learn the basic structure of a graph

"Walk," <u>via edges</u>, from a fixed starting vertex s to all vertices reachable from s

Three states of vertices

undiscovered

discovered

fully-explored

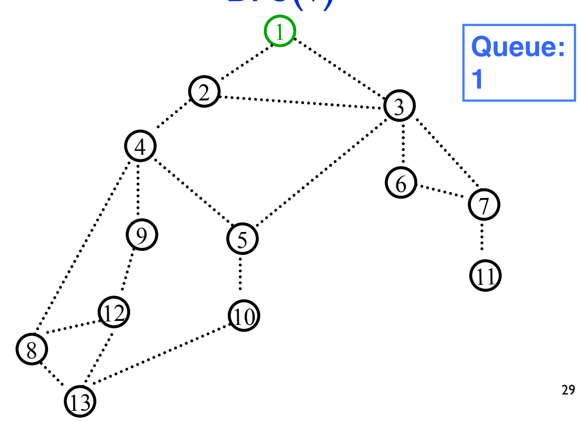
BFS(s) Implementation

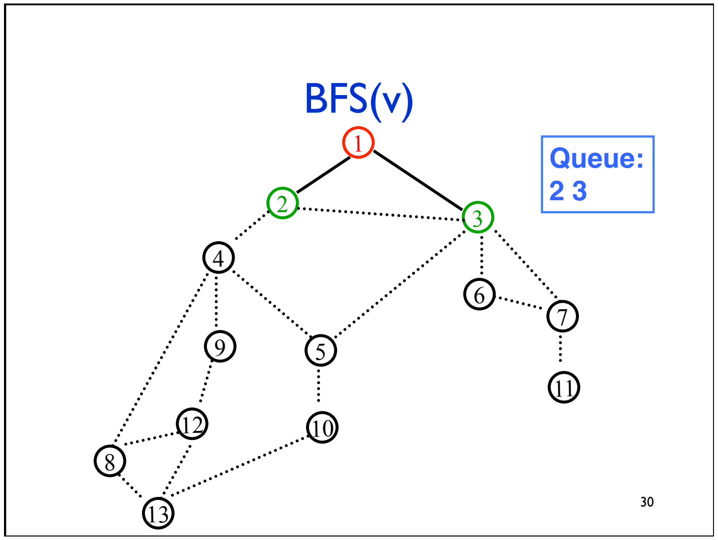
Global initialization: mark all vertices **"undiscovered"**

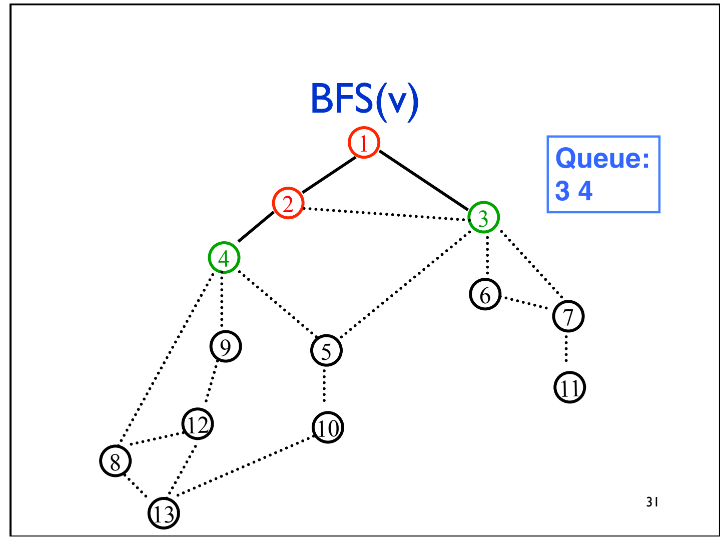
```
BFS(s)
    mark  s "discovered"
    queue = { s }
    while queue not empty
        u = remove_first(queue)
        for each edge {u,x}
            if (x is undiscovered)
                mark x discovered
                append x on queue
        mark u fully explored
```

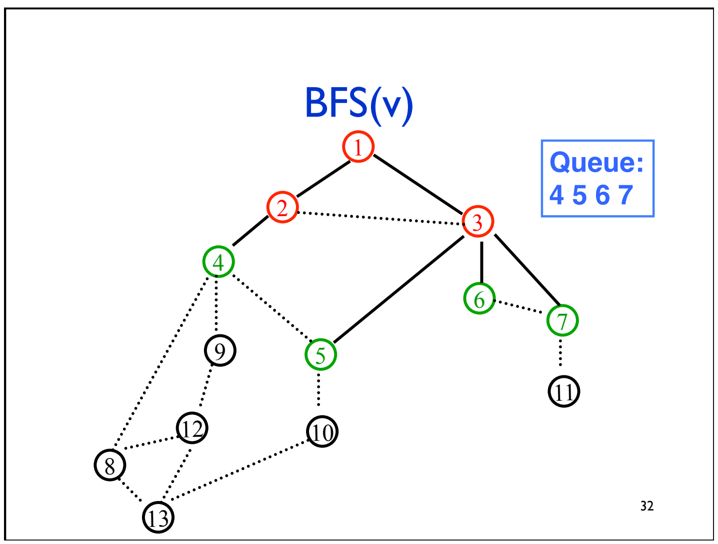
Exercise: modify code to number vertices & compute level numbers

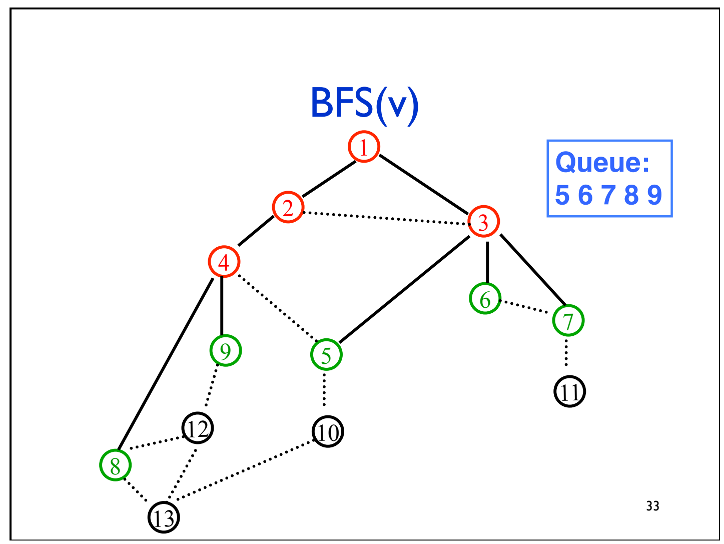
BFS(v)



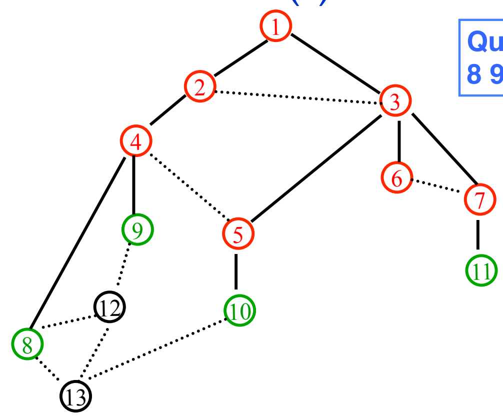

BFS(v)
Queue:
2 3
1
2
3
4
5
6
7
8
9
10
11
12
13
30


BFS(v)
Queue:
3 4
1
2
3
4
5
6
7
8
9
10
11
12
13
31


BFS(v)
Queue:
4 5 6 7
1
2
3
4
5
6
7
8
9
10
11
12
13
32


BFS(v)
Queue:
5 6 7 8 9
1
2
3
4
5
6
7
8
9
10
11
12
13
33

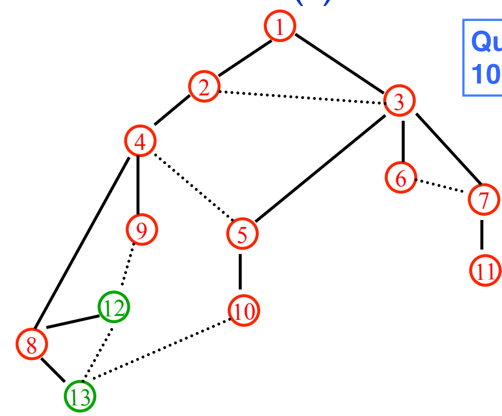
BFS(v)



Queue:
8 9 10 11

34

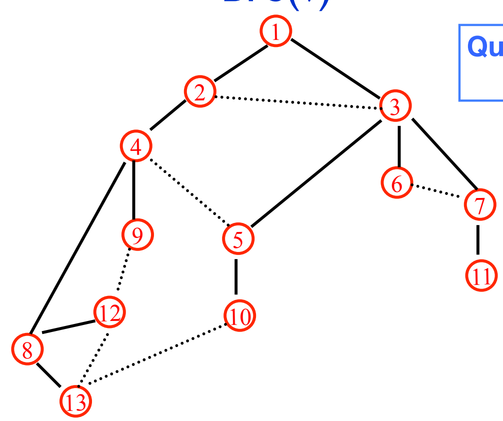
BFS(v)



Queue:
10 11 12 13

35

BFS(v)



Queue:

36

BFS: Analysis, I

```
O(n)  Global initialization: mark all vertices "undiscovered"
 +    BFS(s)
O(1)     mark  s "discovered"
 +       queue = { s }
O(n)     while queue not empty
 x          u = remove_first(queue)
O(n)        for each edge {u,x}
               if (x is undiscovered)
                  mark x discovered
                  append x on queue
 =          mark u fully explored
O(n²)
```

Simple analysis: 2 nested loops. Get worst-case number of iterations of each; multiply.

37

BFS: Analysis, II

Above analysis correct, but pessimistic (can't have $\Omega(n)$ edges incident to each of $\Omega(n)$ distinct "u" vertices if G is sparse). Alt, more global analysis:

Each edge is explored once from each end-point, so *total* runtime of inner loop is O(m).

> Exercise: extend algorithm and analysis to non-connected graphs

Total O(n+m), n = # nodes, m = # 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

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

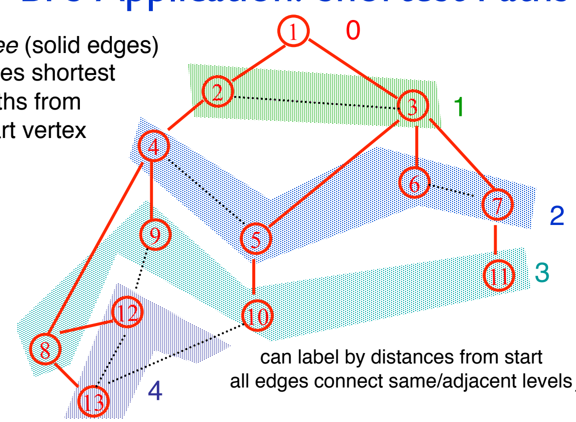
Level i in this tree are exactly those vertices *u* such that the shortest path (in G, not just the tree) from the root v is of length i.

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

> not true of every spanning tree!

BFS Application: Shortest Paths

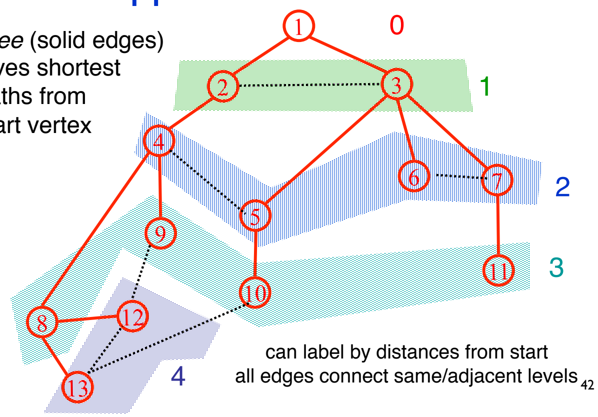
Tree (solid edges) gives shortest paths from start vertex



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

BFS Application: Shortest Paths

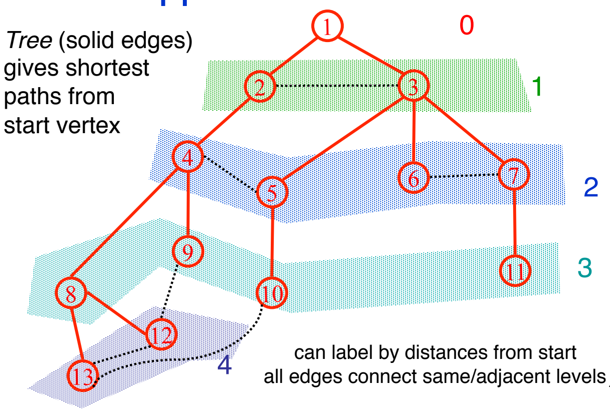
Tree (solid edges) gives shortest paths from start vertex



can label by distances from start
all edges connect same/adjacent levels 42

BFS Application: Shortest Paths

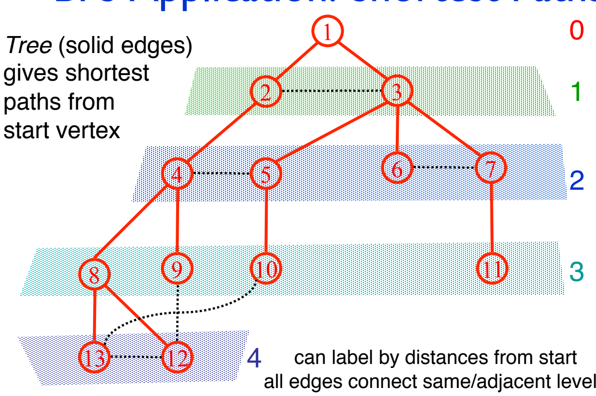
Tree (solid edges) gives shortest paths from start vertex



can label by distances from start
all edges connect same/adjacent levels 43

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 44

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…

45

BFS(s) Implementation

Global initialization: mark all vertices **"undiscovered"**

```
BFS(s)
    mark  s "discovered"
    queue = { s }
    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 fully explored
```

> Exercise: modify code to number vertices & compute level numbers
>
> Label edges as tree edges or non-tree edges

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?

Set up one-time data structure to answer such questions efficiently.

47

Graph Search Application: Connected Components

```
initial state: all v undiscovered
for v = 1 to n do
    if state(v) != fully-explored then
        BFS(v)
    endif
endfor
```

> Exercise: modify code to answer queries

Total cost: O(n+m)

each edge is touched a constant number of times (twice)

works also with DFS

48

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?

Idea: create array A such that

A[u] = smallest numbered vertex that is connected to u. Question reduces to whether A[u]=A[v]?

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

49

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?

Idea: create array A such that

- A[u] = smallest numbered vertex that is connected to u. Question reduces to whether A[u]=A[v]?

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

50

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 (twice)
- works also with DFS

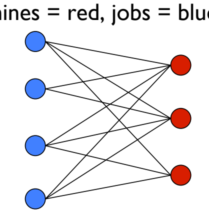
51

3.4 Testing Bipartiteness

Bipartite Graphs

Def. An undirected graph G = (V, E) is *bipartite (2-colorable)* if the nodes can be colored red or blue such that no edge has both ends the same color.

Applications.
- Stable marriage: men = red, women = blue
- Scheduling: machines = red, jobs = blue



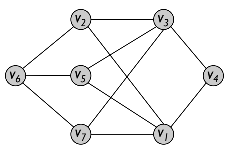
a bipartite graph

"bi-partite" means "two parts." An equivalent definition: G 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/no edge has both ends in the same part.

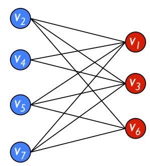
53

Testing Bipartiteness

Testing bipartiteness. Given a graph G, is it bipartite?
- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



a bipartite graph G

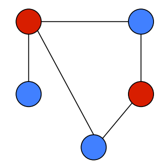


another drawing of G

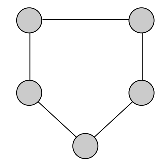
An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

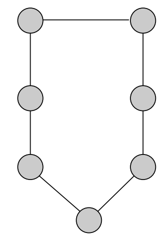
Pf. Impossible to 2-color the odd cycle, let alone G.



bipartite (2-colorable)



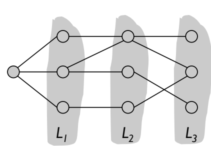
not bipartite (not 2-colorable)



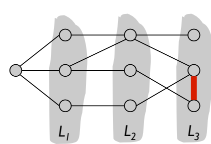
not bipartite (not 2-colorable)

Bipartite Graphs

Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node 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)

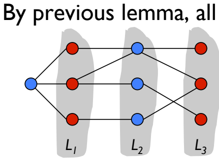


Case (ii)

Bipartite Graphs

Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node 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).

Pf. (i)
- Suppose no edge joins two nodes in the same layer.
- By previous lemma, all edges join nodes on adjacent levels.



Case (i)

Bipartition:
- red = nodes on odd levels,
- blue = nodes on even levels.

Bipartite Graphs

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

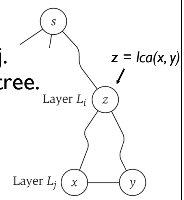
(i) No edge of G joins two nodes of the same layer, and G is bipartite.

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

Pf. (ii)

Suppose (x, y) is an edge & x, y in same level Lj.
Let z = their lowest common ancestor in BFS tree.
Let Li 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.

(x, y) — *path from y to z* — *path from z to x*

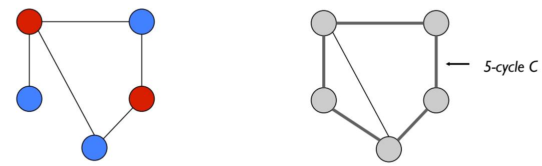


58

Obstruction to Bipartiteness

Cor: A graph G is bipartite iff it contains no odd length cycle.

NB: the proof is algorithmic–it *finds* a coloring or odd cycle.



bipartite (2-colorable)

not bipartite (not 2-colorable)

59

BFS(s) Implementation

Global initialization: mark all vertices **"undiscovered"**

```
BFS(s)
   mark s "discovered"
   queue = { s }
   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 fully explored
```

Exercise: modify code to determine if the graph is bipartite

60

3.6 DAGs and Topological Ordering

Precedence Constraints

Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j.

Applications

- Course prerequisites: course v_i must be taken before v_j
- Compilation: must compile module v_i before v_j
- Computing workflow: output of job v_i is input to job v_j
- Manufacturing or assembly: sand it before you paint it…
- Spreadsheet evaluation order: if A7 is "=A6+A5+A4", evaluate them first

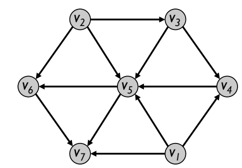
Directed Acyclic Graphs

Def. A DAG is a directed acyclic graph, i.e., one that contains no directed cycles.

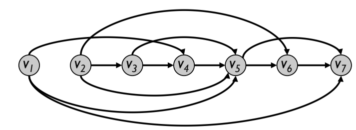
Ex. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.

Def. A topological order of a directed graph G = (V, E) is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.

E.g., $\forall$edge (v_i, v_j), finish v_i before starting v_j



a DAG



a topological ordering of that DAG– all edges left-to-right

Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

Directed Acyclic Graphs

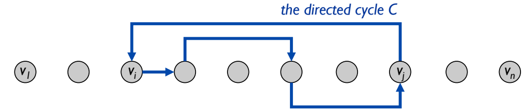
Lemma. If G has a topological order, then G is a DAG.

if all edges go L→R, you can't loop back to close a cycle

Pf. (by contradiction)

Suppose that G has a topological order $v_1, \ldots, v_n$ and that G also has a directed cycle C.
Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.
By our choice of i, we have $i < j$.
On the other hand, since (v_j, v_i) is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction.

the directed cycle C



the supposed topological order: $v_1, \ldots, v_n$

Directed Acyclic Graphs

Lemma.
If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?

66

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

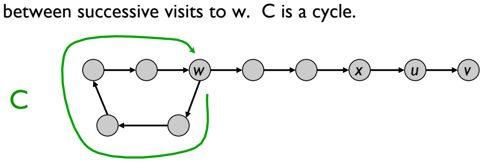
Pf. (by contradiction)
Suppose that G is a DAG and every node has at least one incoming edge

67

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)
Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u.
Then, since u has at least one incoming edge (x, u), we can walk backward to x.
Repeat until we visit a node, say w, twice. ← Why must this happen?
Let C be the sequence of nodes encountered between successive visits to w. C is a cycle.



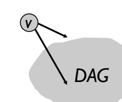
68

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a topological ordering.

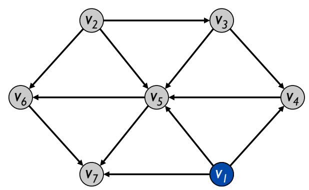
Pf. (by induction on n)
Base case: true if n = 1.
Given DAG on n > 1 nodes, find a node v with no incoming edges.
G - { v } is a DAG, since deleting v cannot create cycles.
By inductive hypothesis, G - { v } has a topological ordering.
Place v first in topological ordering; then append nodes of G - { v } in topological order. This is valid since v has no incoming edges. ▪

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```



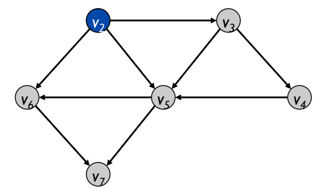
69

Topological Ordering Algorithm: Example



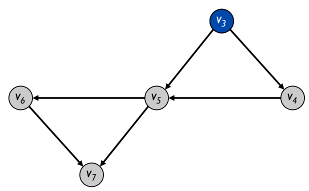
Topological order:

Topological Ordering Algorithm: Example



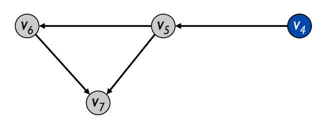
Topological order: v_1

Topological Ordering Algorithm: Example



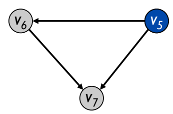
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



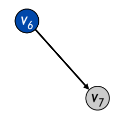
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

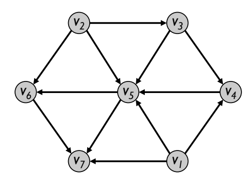


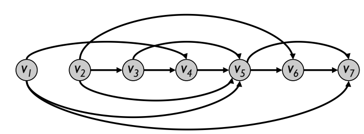
Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$

Topological Ordering Algorithm: Example





Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

Topological Sorting Algorithm

Linear time implementation?

78

Topological Sorting Algorithm

Maintain the following:
  count[w] = (remaining) number of incoming edges to node w
  S = set of (remaining) nodes with no incoming edges

Initialization: (O(m + n))
  count[w] = 0 for all w
  count[w]++ for all edges (v,w)
  S = S ∪ {w} for all w with count[w]==0

Main loop: (O(1) per node, O(1) per edge)
  while S not empty
    remove some v from S
    make v next in topo order
    for all edges from v to some w
    decrement count[w]
    add w to S if count[w] hits 0

Correctness: clear, I hope

Time: O(m + n) (assuming edge-list representation of graph)

79

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

80

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

82

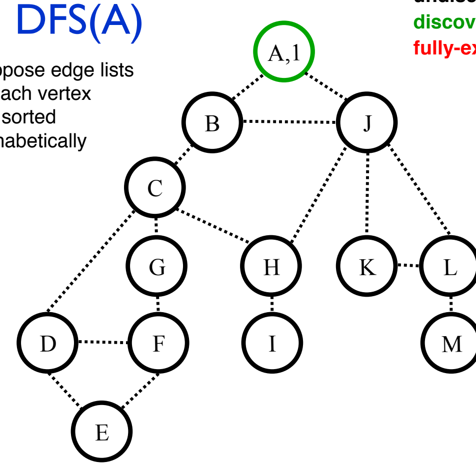
Why fuss about trees (again)?

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

83

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



Color code:
undiscovered
discovered
fully-explored

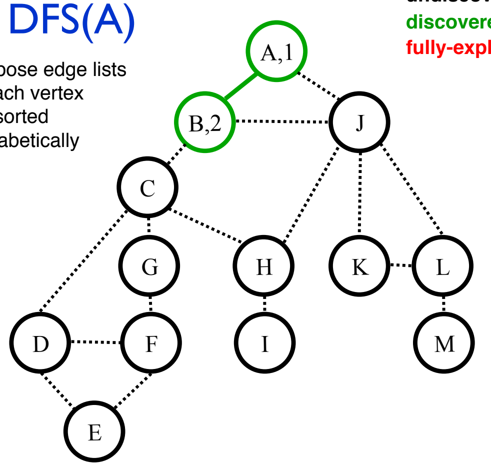
Call Stack (Edge list):

A (B,J)

84

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



Color code:
undiscovered
discovered
fully-explored

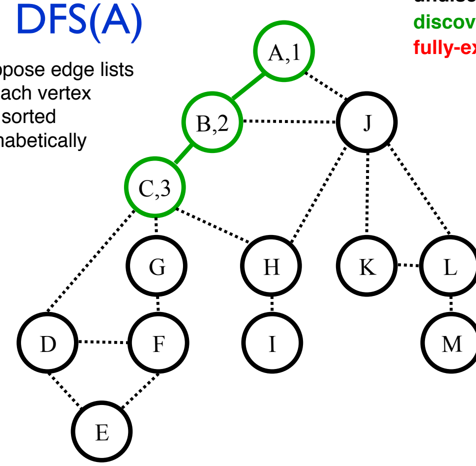
Call Stack: (Edge list)

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

85

DFS(A)

Suppose edge lists at each vertex are sorted alphabetically



Color code:
undiscovered
discovered
fully-explored

Call Stack: (Edge list)

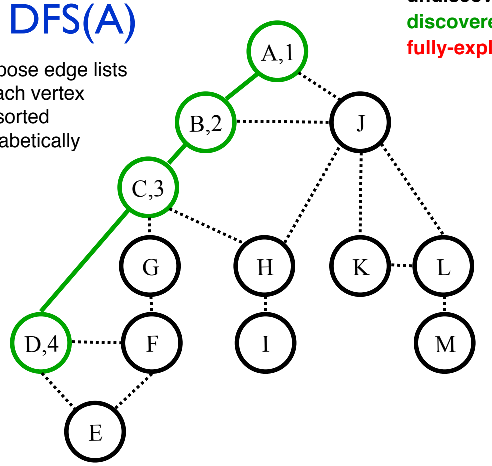
A (B,J)
B (A,C,J)
C (B,D,G,H)

86

DFS(A)

Color code:
undiscovered
discovered
fully-explored

Suppose edge lists at each vertex are sorted alphabetically



Call Stack:
(Edge list)

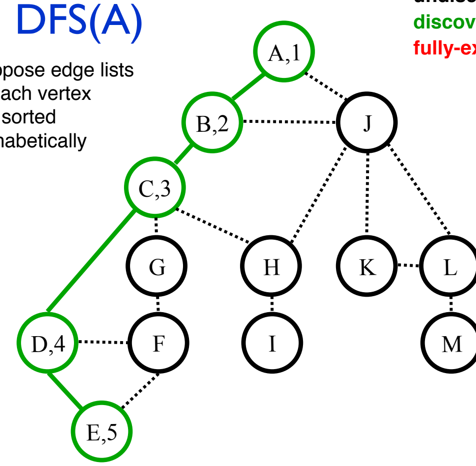
A (B,J)
B (A,C,J)
C (B,D,G,H)
D (C,E,F)

87

DFS(A)

Color code:
undiscovered
discovered
fully-explored

Suppose edge lists at each vertex are sorted alphabetically



Call Stack:
(Edge list)

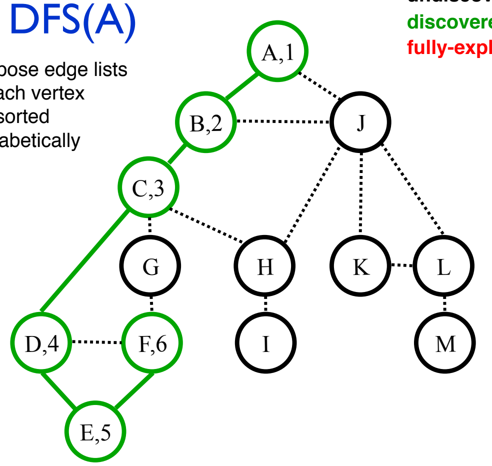
A (B,J)
B (A,C,J)
C (B,D,G,H)
D (C,E,F)
E (D,F)

88

DFS(A)

Color code:
undiscovered
discovered
fully-explored

Suppose edge lists at each vertex are sorted alphabetically



Call Stack:
(Edge list)

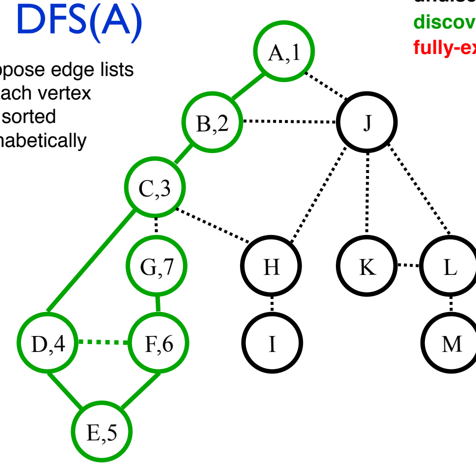
A (B,J)
B (A,C,J)
C (B,D,G,H)
D (C,E,F)
E (D,F)
F (D,E,G)

89

DFS(A)

Color code:
undiscovered
discovered
fully-explored

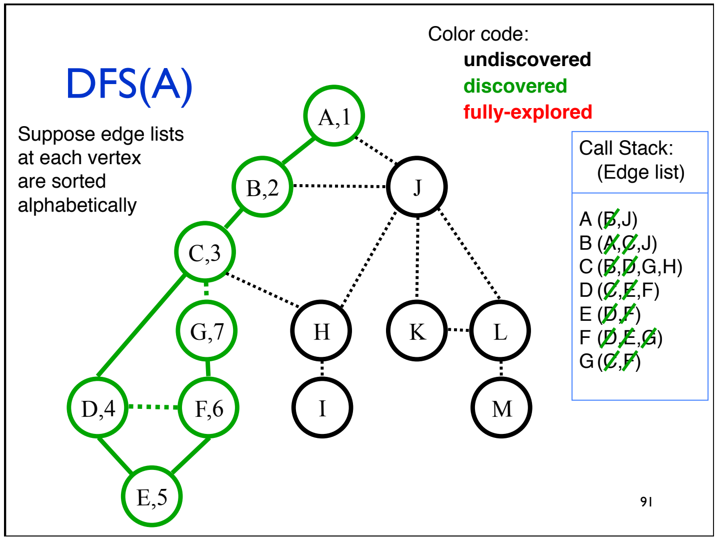
Suppose edge lists at each vertex are sorted alphabetically

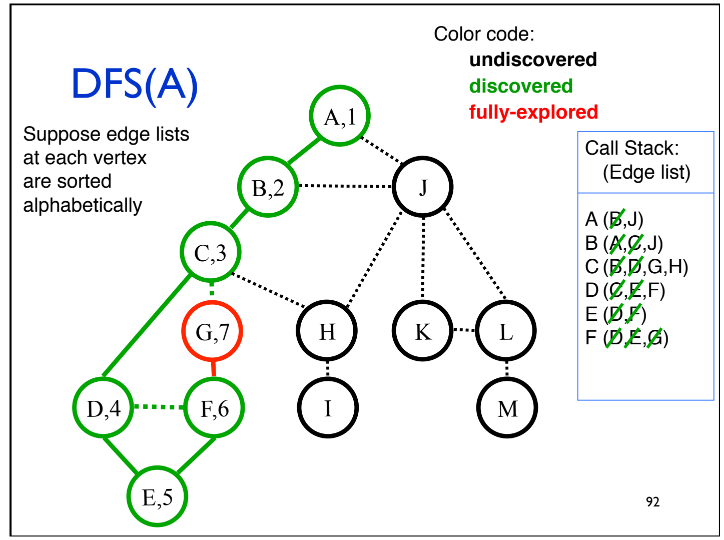


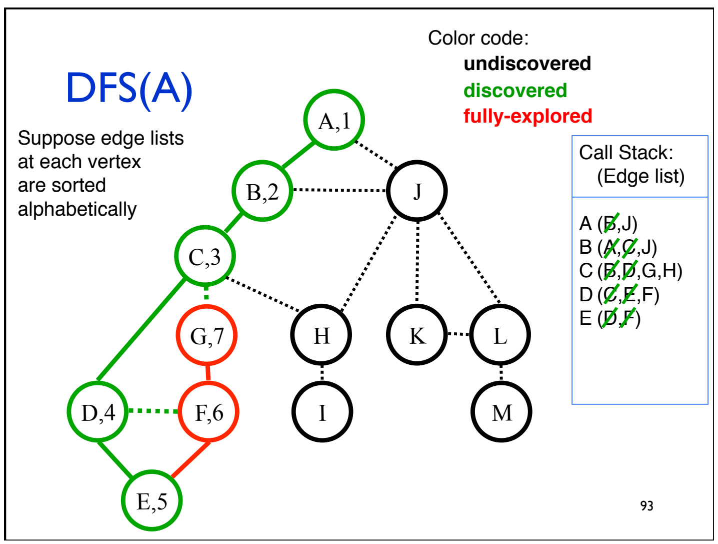
Call Stack:
(Edge list)

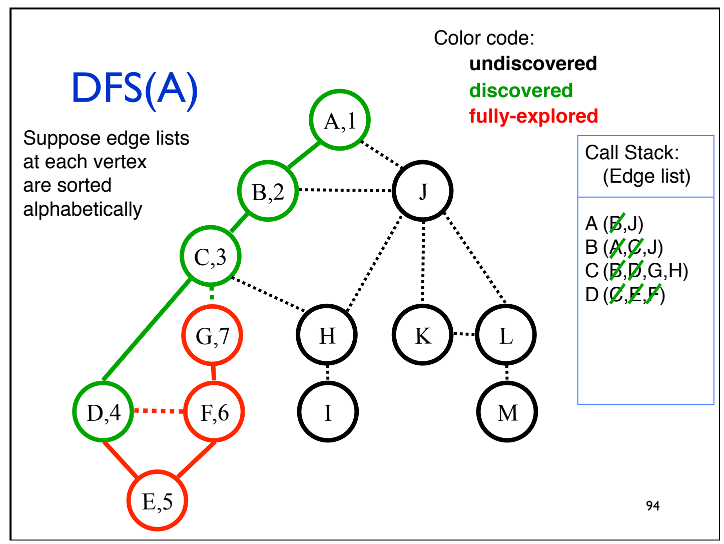
A (B,J)
B (A,C,J)
C (B,D,G,H)
D (C,E,F)
E (D,F)
F (D,E,G)
G (C,F)

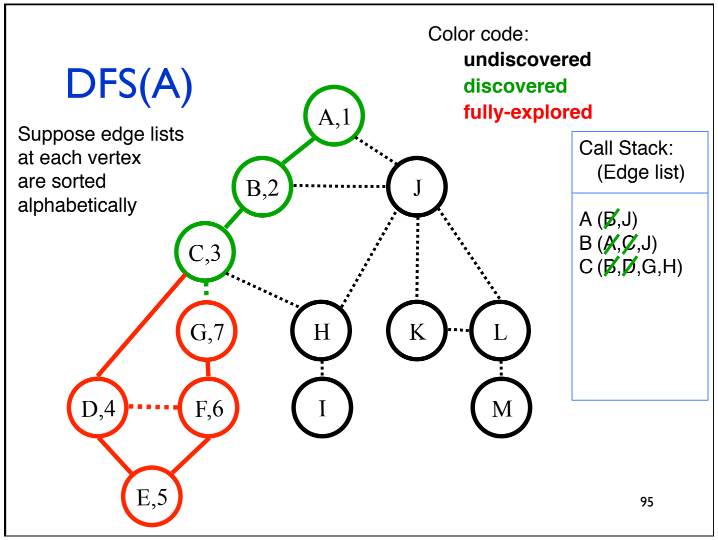
90

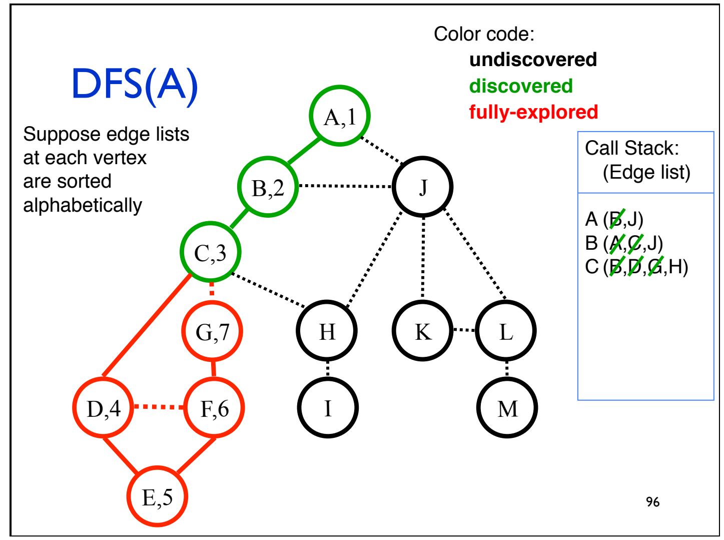

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
C,3
D,4
E,5
F,6
G,7
H
I
J
K
L
M
Call Stack: (Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
D (C,E,F)
E (D,F)
F (D,E,G)
G (C,F)
91

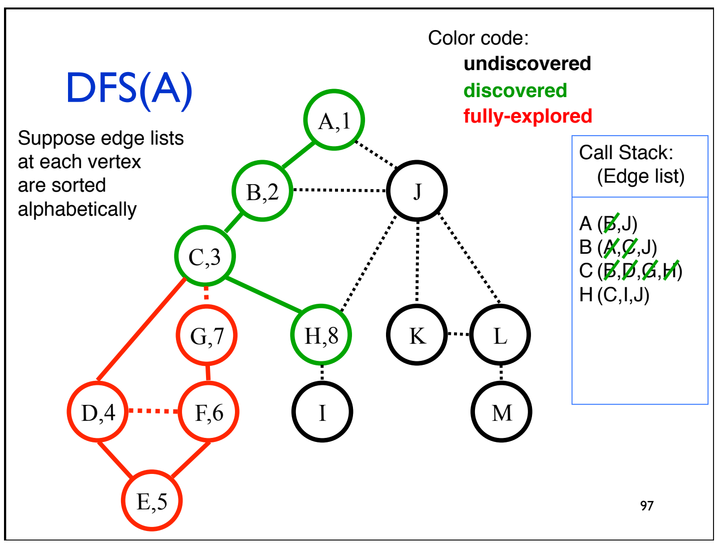

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
C,3
D,4
E,5
F,6
G,7
H
I
J
K
L
M
Call Stack: (Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
D (C,E,F)
E (D,F)
F (D,E,G)
92

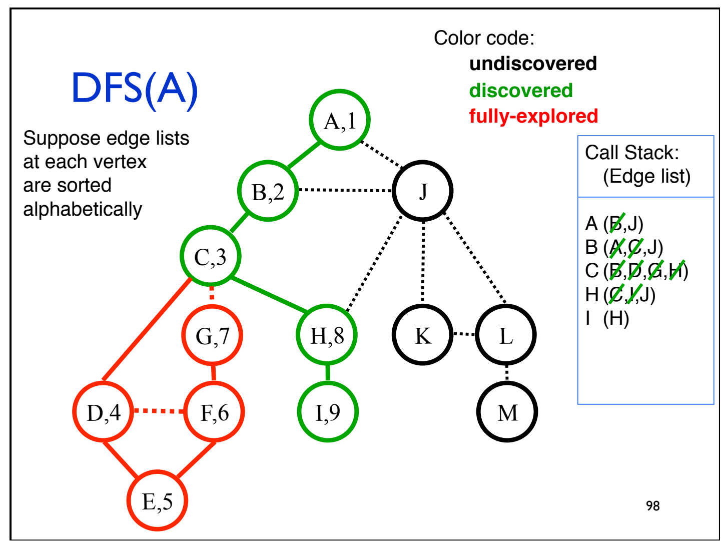

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
C,3
D,4
E,5
F,6
G,7
H
I
J
K
L
M
Call Stack: (Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
D (C,E,F)
E (D,F)
93

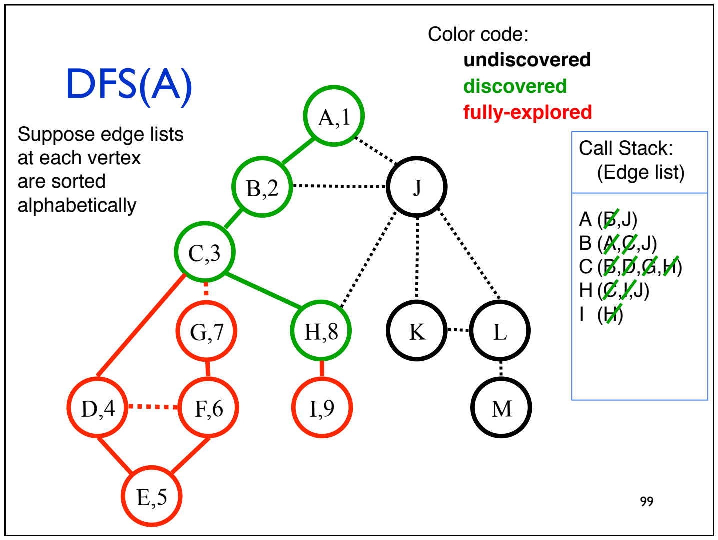

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
C,3
D,4
E,5
F,6
G,7
H
I
J
K
L
M
Call Stack: (Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
D (C,E,F)
94

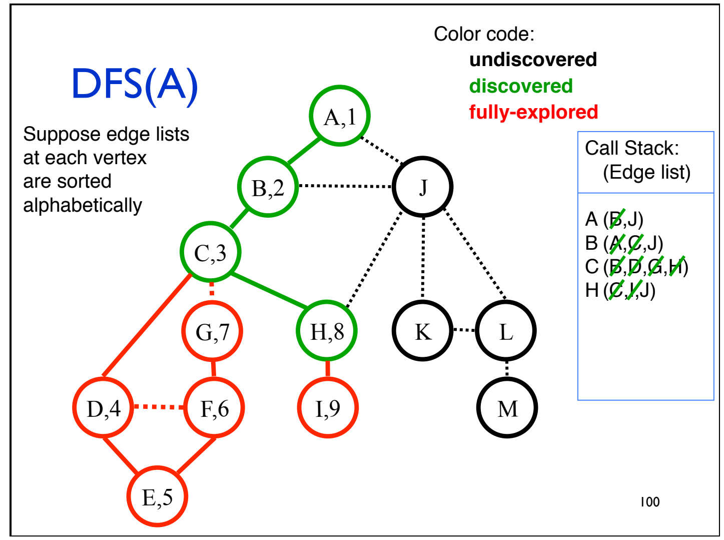

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
C,3
J
G,7
H
K
L
D,4
F,6
I
M
E,5
Call Stack:
(Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
95

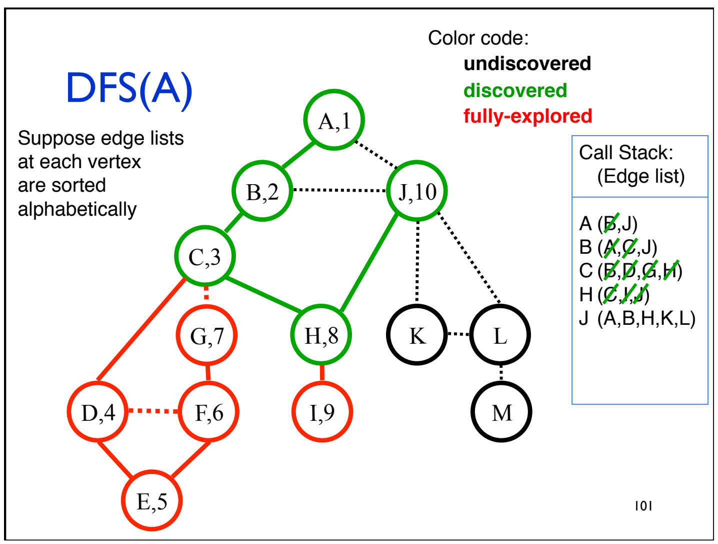

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
C,3
J
G,7
H
K
L
D,4
F,6
I
M
E,5
Call Stack:
(Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
96

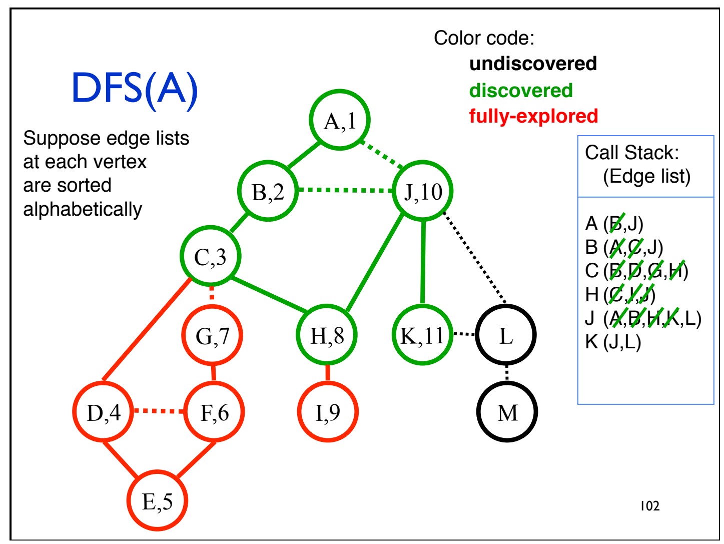

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
C,3
J
G,7
H,8
K
L
D,4
F,6
I
M
E,5
Call Stack:
(Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
H (C,I,J)
97

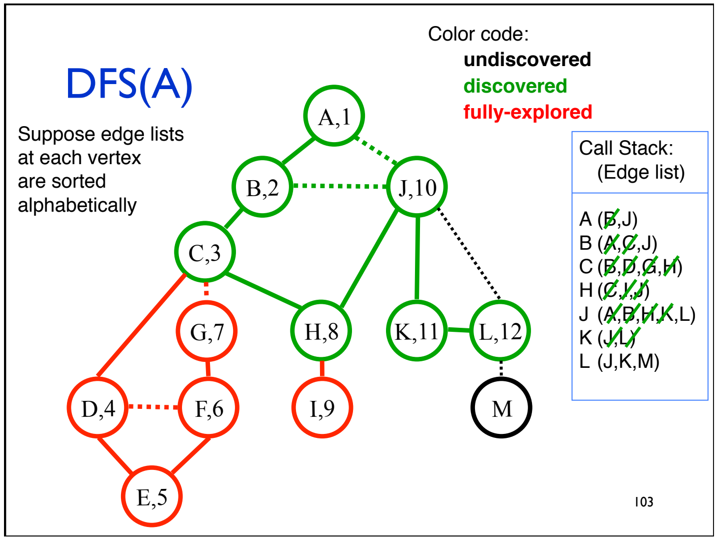

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
C,3
J
G,7
H,8
K
L
D,4
F,6
I,9
M
E,5
Call Stack:
(Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
H (C,I,J)
I (H)
98

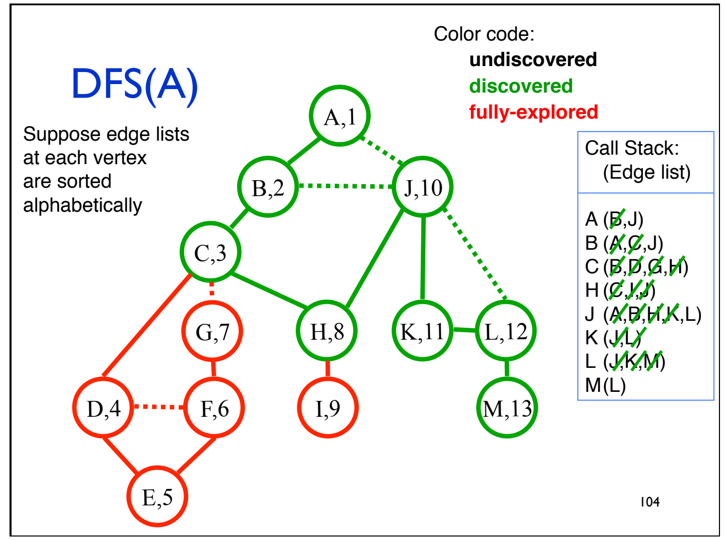

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
J
C,3
G,7
H,8
K
L
D,4
F,6
I,9
M
E,5
Call Stack:
(Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
H (C,I,J)
I (H)
99

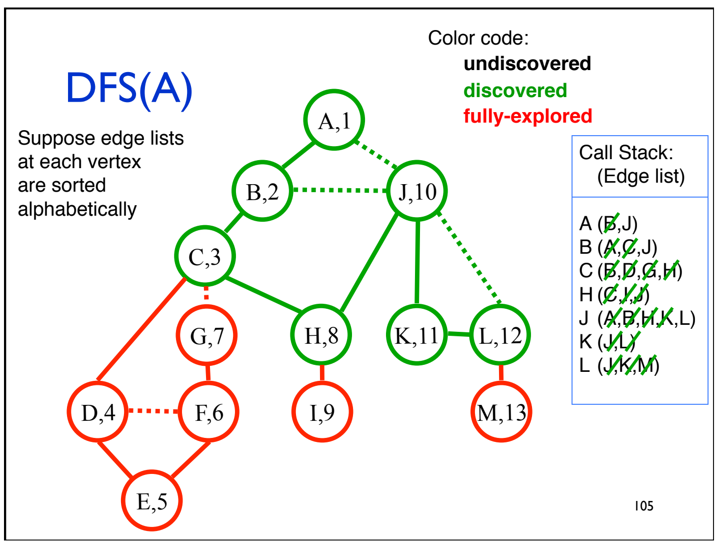

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
J
C,3
G,7
H,8
K
L
D,4
F,6
I,9
M
E,5
Call Stack:
(Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
H (C,I,J)
100

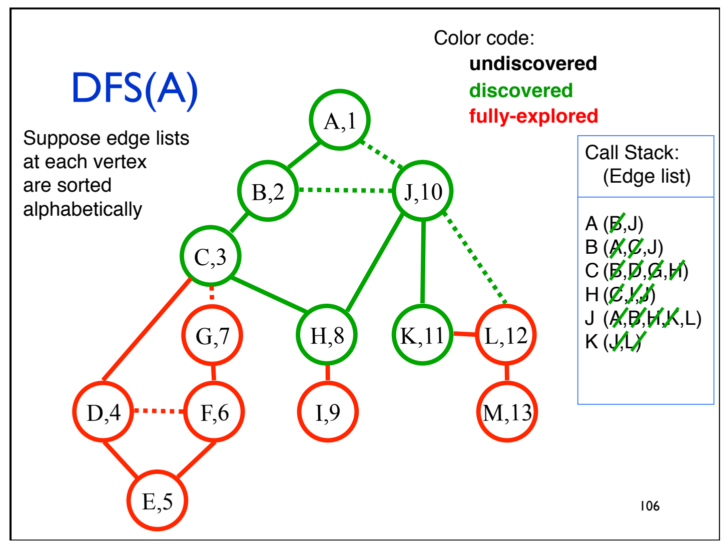

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
J,10
C,3
G,7
H,8
K
L
D,4
F,6
I,9
M
E,5
Call Stack:
(Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
H (C,I,J)
J (A,B,H,K,L)
101

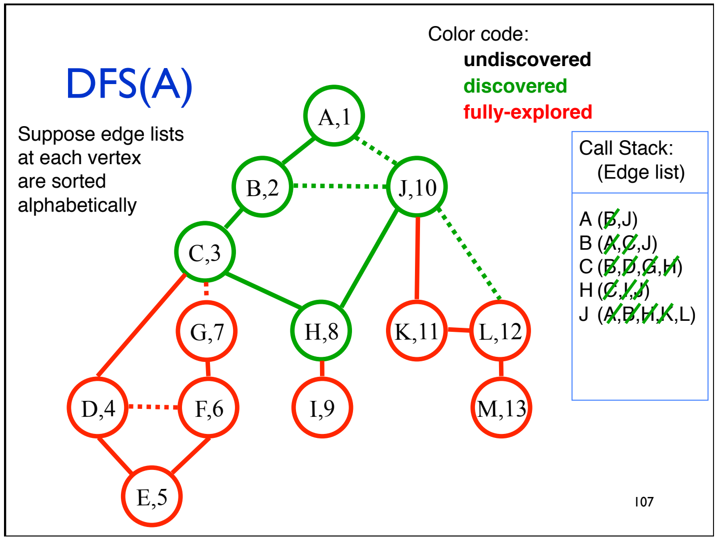

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
J,10
C,3
G,7
H,8
K,11
L
D,4
F,6
I,9
M
E,5
Call Stack:
(Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
H (C,I,J)
J (A,B,H,K,L)
K (J,L)
102

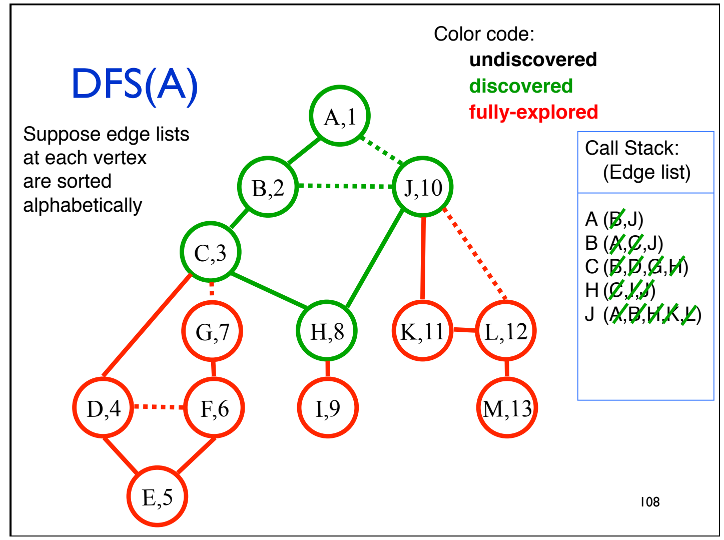

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
J,10
C,3
G,7
H,8
K,11
L,12
D,4
F,6
I,9
M
E,5
Call Stack: (Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
H (C,I,J)
J (A,B,H,K,L)
K (J,L)
L (J,K,M)
103

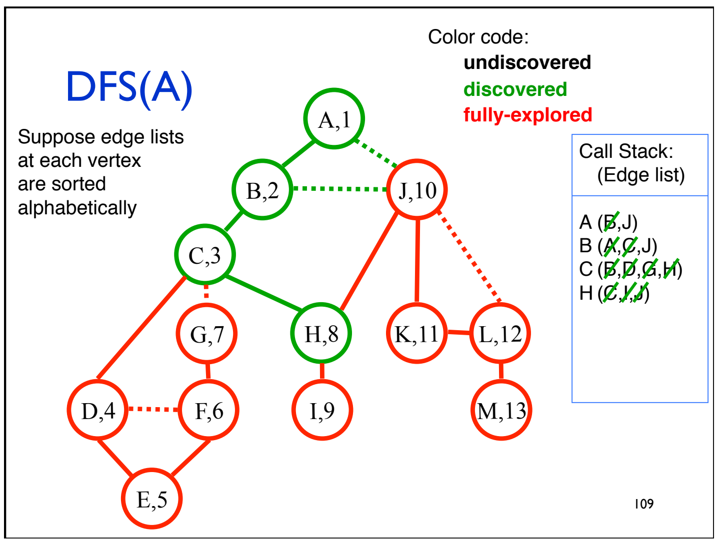

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
J,10
C,3
G,7
H,8
K,11
L,12
D,4
F,6
I,9
M,13
E,5
Call Stack: (Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
H (C,I,J)
J (A,B,H,K,L)
K (J,L)
L (J,K,M)
M (L)
104

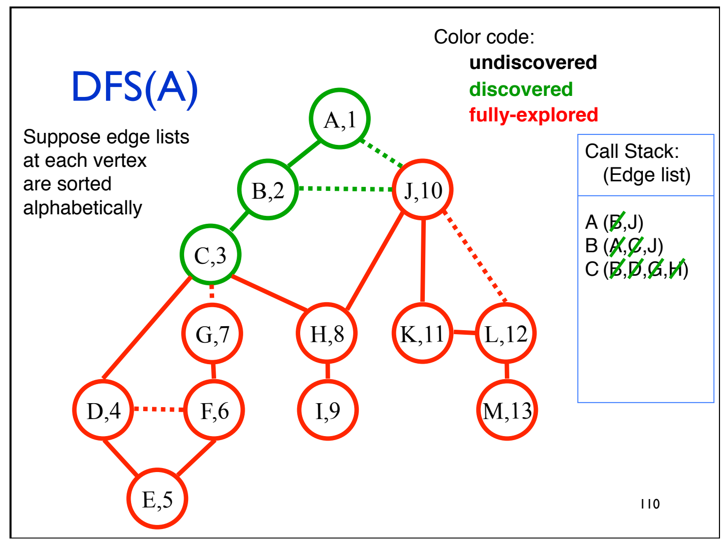

DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
A,1
B,2
J,10
C,3
G,7
H,8
K,11
L,12
D,4
F,6
I,9
M,13
E,5
Call Stack: (Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
H (C,I,J)
J (A,B,H,K,L)
K (J,L)
L (J,K,M)
105


DFS(A)
Suppose edge lists at each vertex are sorted alphabetically
Color code:
undiscovered
discovered
fully-explored
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
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)
106


DFS(A)
Color code:
undiscovered
discovered
fully-explored
Suppose edge lists at each vertex are sorted alphabetically
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
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)
107


DFS(A)
Color code:
undiscovered
discovered
fully-explored
Suppose edge lists at each vertex are sorted alphabetically
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
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)
108

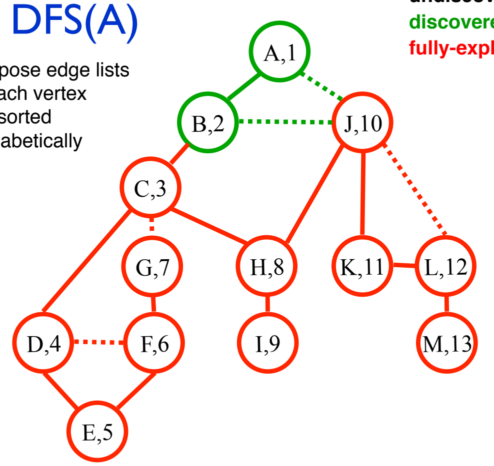

DFS(A)
Color code:
undiscovered
discovered
fully-explored
Suppose edge lists at each vertex are sorted alphabetically
A,1
B,2
J,10
C,3
G,7
H,8
K,11
L,12
D,4
F,6
I,9
M,13
E,5
Call Stack:
(Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
H (C,I,J)
109


DFS(A)
Color code:
undiscovered
discovered
fully-explored
Suppose edge lists at each vertex are sorted alphabetically
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
Call Stack:
(Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
110

DFS(A)

Color code:
undiscovered
discovered
fully-explored

Suppose edge lists at each vertex are sorted alphabetically



Call Stack:
(Edge list)

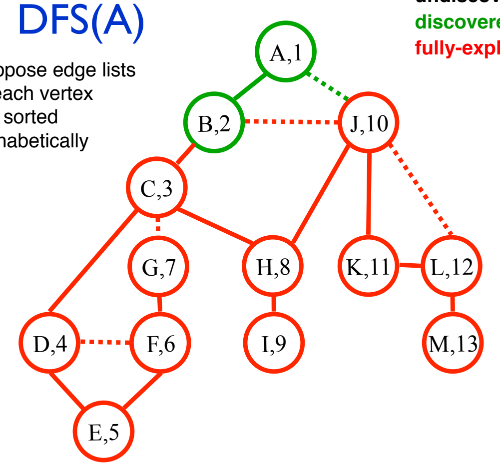
A (B,J)
B (A,C,J)

111

DFS(A)

Color code:
undiscovered
discovered
fully-explored

Suppose edge lists at each vertex are sorted alphabetically



Call Stack:
(Edge list)

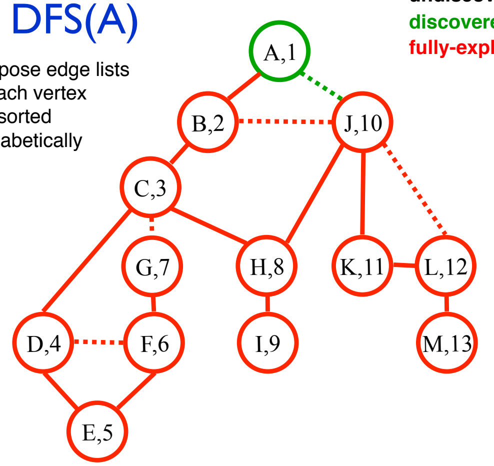
A (B,J)
B (A,C,J)

112

DFS(A)

Color code:
undiscovered
discovered
fully-explored

Suppose edge lists at each vertex are sorted alphabetically



Call Stack:
(Edge list)

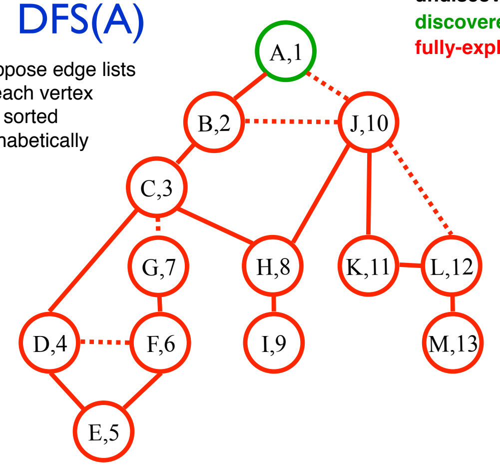
A (B,J)

113

DFS(A)

Color code:
undiscovered
discovered
fully-explored

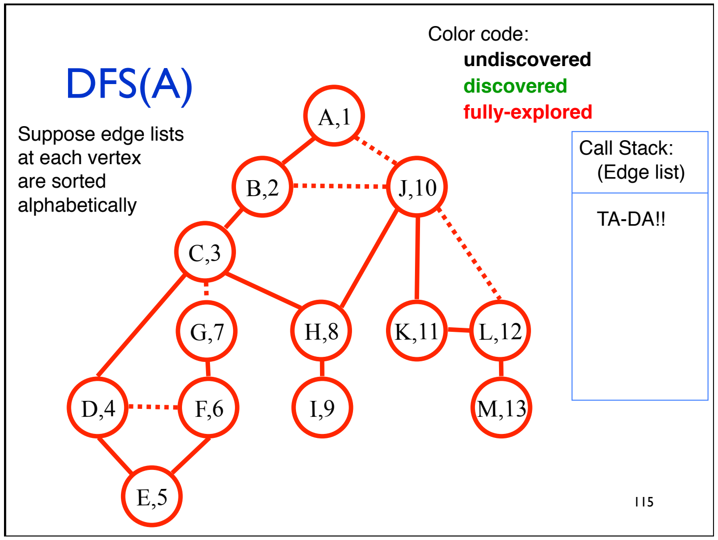
Suppose edge lists at each vertex are sorted alphabetically

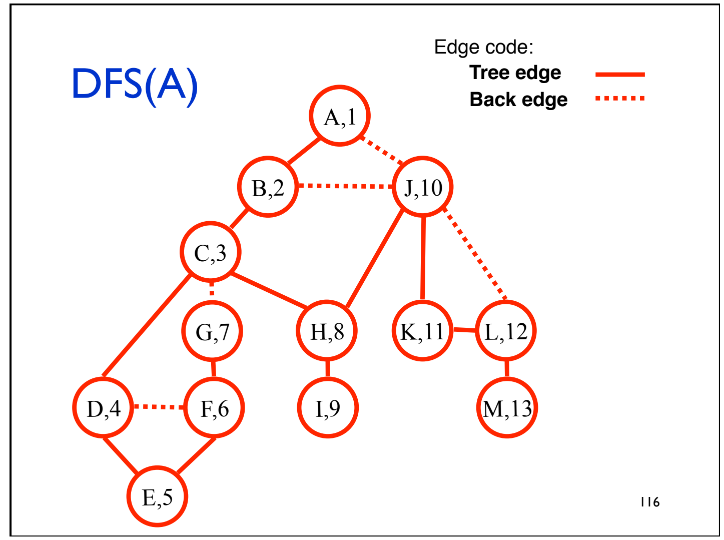


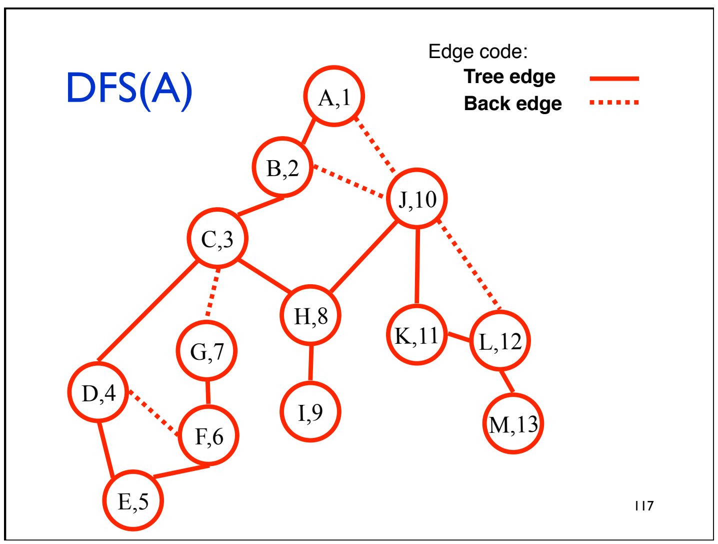
Call Stack:
(Edge list)

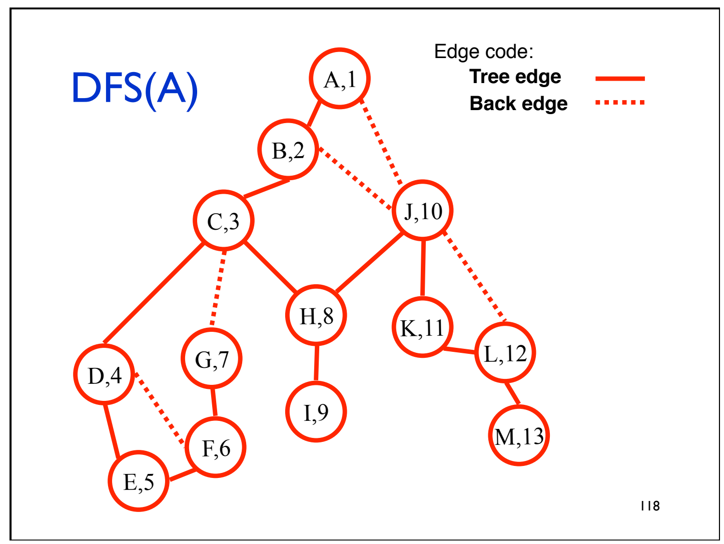
A (B,J)

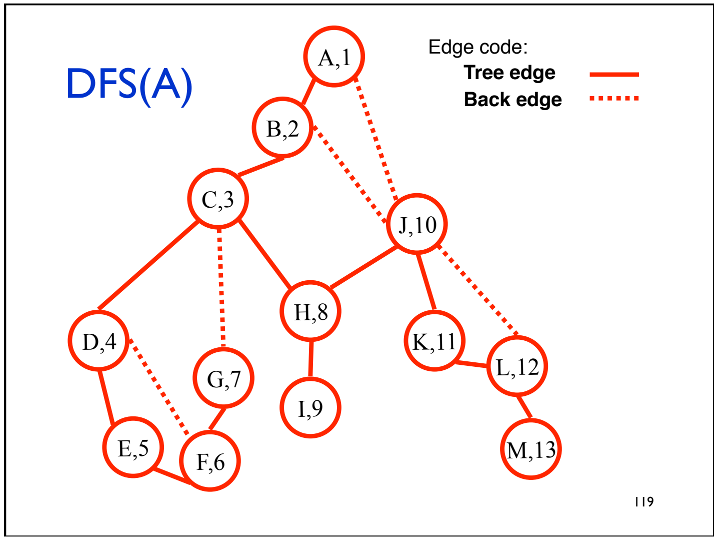
114

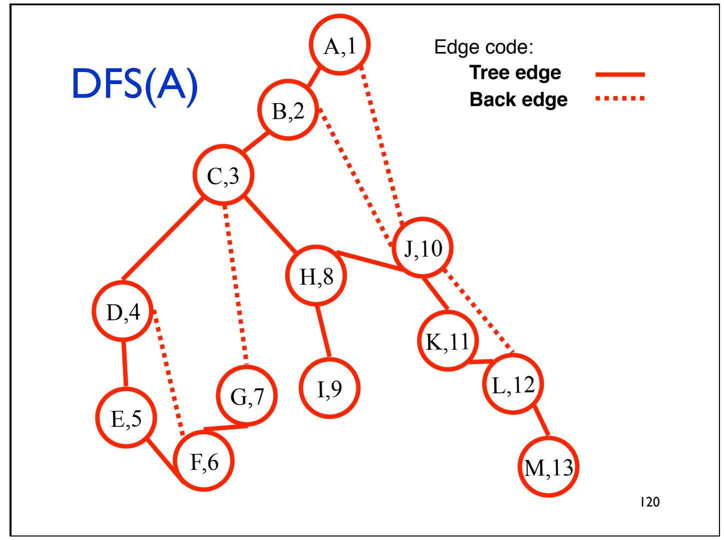

DFS(A)
Color code:
undiscovered
discovered
fully-explored
Suppose edge lists at each vertex are sorted alphabetically
Call Stack:
(Edge list)
TA-DA!!
A,1
B,2
J,10
C,3
G,7
H,8
K,11
L,12
D,4
F,6
I,9
M,13
E,5
115

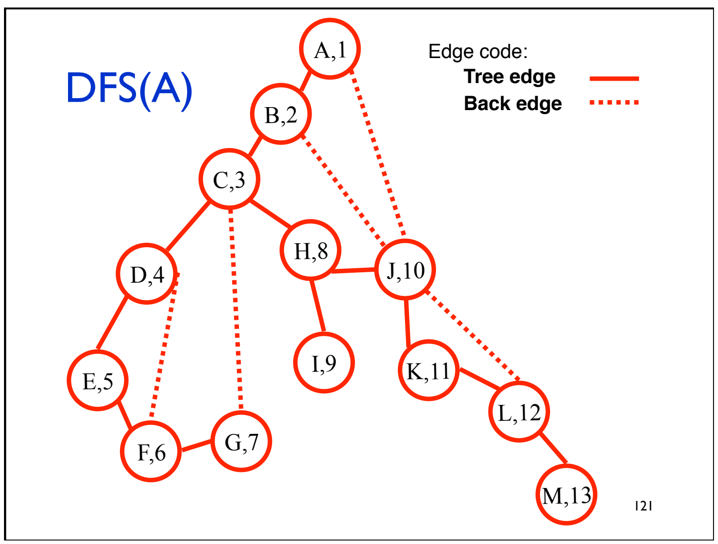

DFS(A)
Edge code:
Tree edge
Back edge
A,1
B,2
J,10
C,3
G,7
H,8
K,11
L,12
D,4
F,6
I,9
M,13
E,5
116

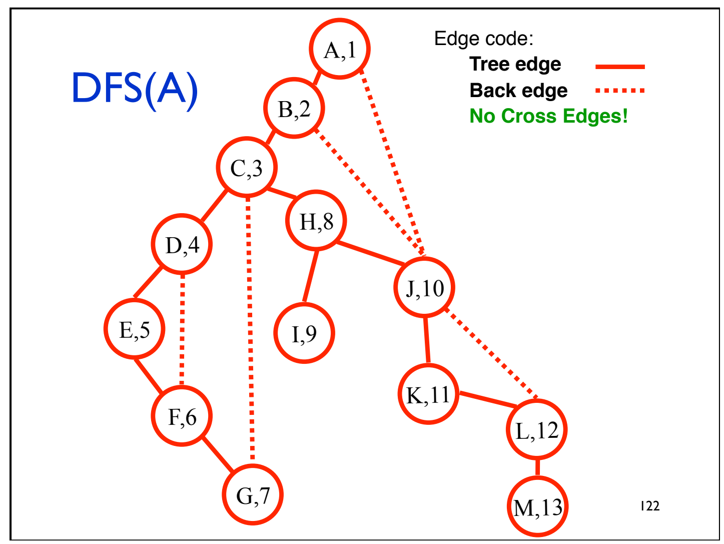

DFS(A)
Edge code:
Tree edge
Back edge
A,1
B,2
J,10
C,3
H,8
K,11
L,12
G,7
D,4
F,6
I,9
M,13
E,5
117


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
118


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
119


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
120


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
121


DFS(A)
Edge code:
Tree edge
Back edge
No Cross Edges!
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
122

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

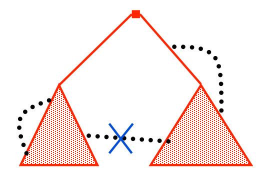
BUT…

123

Non-tree edges

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

No cross edges!



124

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

125

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)            // (x previously  undiscovered)
         DFS(x)
      else …
```

126

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?

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
```


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? Using depth first search

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:

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

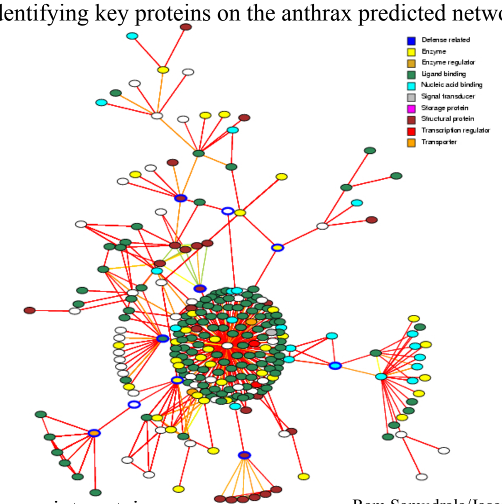
Application: Articulation Points

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

131

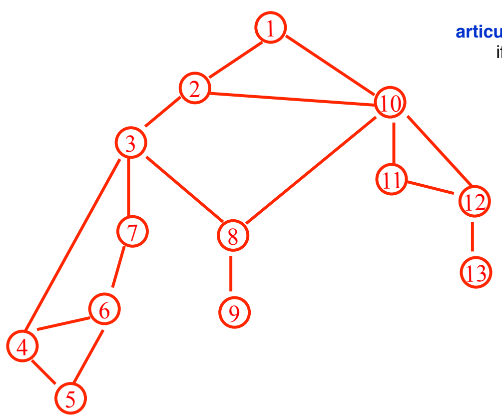
Identifying key proteins on the anthrax predicted network



Articulation point proteins

Ram Samudrala/Jason McDermott

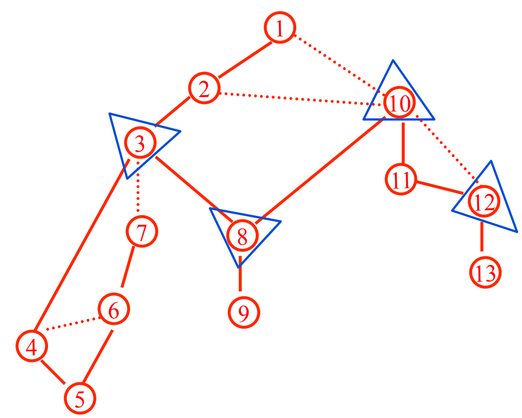
Articulation Points



articulation point iff its removal disconnects the graph

133

Articulation Points



134

Simple Case: Artic. Pts in a tree

Which nodes in a rooted tree are 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?)

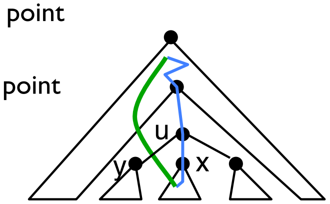
Articulation Points from DFS

Root node is an articulation point iff ….

Leaf is never an articulation point

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

⇕



Articulation Points from DFS

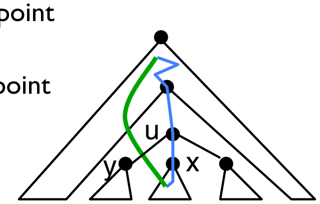
Root node is an articulation point iff it has more than one child

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

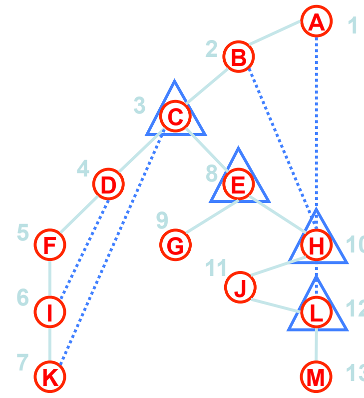
trivial

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

139

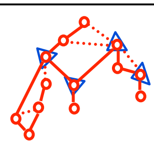
Articulation Points

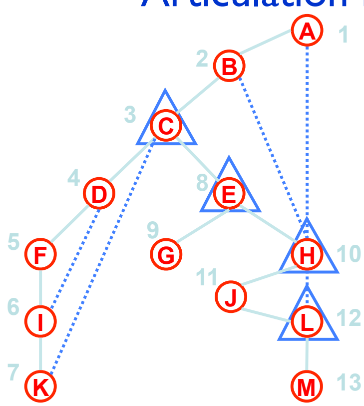


Vertex	DFS #	Low
A	1	1
B	2	1
C	3	1
D	4	3
E	8	1
F	5	3
G	9	9
H	10	1
I	6	3
J	11	10
K	7	3
L	12	10
M	13	13

140

Articulation Point





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

141

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

v articulation point iff…

142

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

v (non-root) articulation point iff some child x of v has LOW(x)) ≥ dfs#(v)

143

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

v (nonroot) articulation point iff some child x of v has LOW(x)) ≥ dfs#(v)

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

144

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?

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

Algorithms – connected components, bipartiteness, topological sort, articulation points

146