Graph Algorithms

Imran Rashid

University of Washington

Jan 16, 2008

Lecture Outline

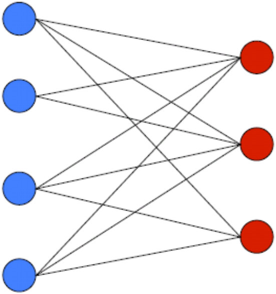
1. BFS
   - Bipartite Graphs
2. DAGs & Topological Ordering
3. DFS

Lecture Outline

1. BFS
    - Bipartite Graphs
2. DAGs & Topological Ordering
3. DFS

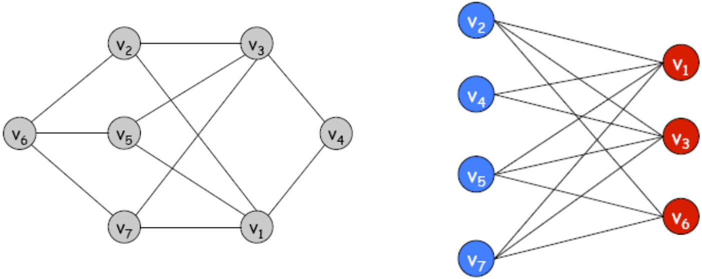
Bipartite Graphs

- Def. An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.
- Applications.
    - Stable marriage: men $=$ red, women $=$ blue
    - Scheduling: machines $=$ red, jobs $=$ blue



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.



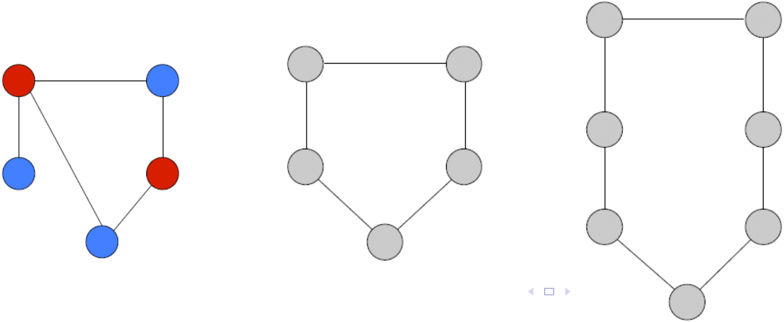
An Obstruction to Bipartiteness

Lemma

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

Proof.

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

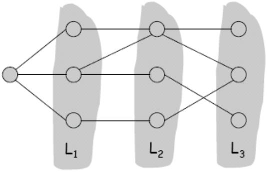


BFS & Bipartite Graphs

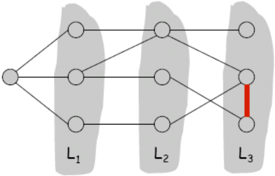
Lemma

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

1. *No edge joins nodes of the same layer, and G is bipartite.*
2. *An edge joins nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).*



case 1



case 2

BFS & Bipartite Graphs

Lemma

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

1. *No edge joins nodes of the same layer, and G is bipartite.*
2. *An edge joins nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).*

Proof.

(1)

- Suppose no edge joins two nodes in the same layer.
- So all edges join nodes on adjacent levels (prop of BFS).
- Bipartition: red $=$ odd levels, blue $=$ even levels.

BFS & Bipartite Graphs

Lemma

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

1. *No edge joins nodes of the same layer, and G is bipartite.*
2. *An edge joins nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).*

Proof.

(2)Suppose (x, y) is an edge & x, y in same level L_j. Let $z =$ their lowest common ancestor in BFS tree. Let L_i be level containing z. Consider cycle that takes edge from x to y, then tree from y to z, then tree from z to x. Its length is $1 + 2(j - i)$, which is odd. ☐

Obstruction to Bipartiteness

Corollary

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

Lecture Outline

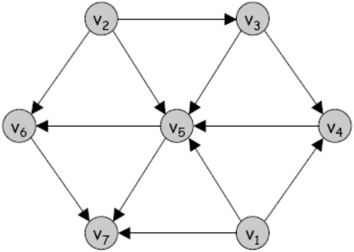
1. BFS
    - Bipartite Graphs
2. DAGs & Topological Ordering
3. DFS

Precedence Constraints

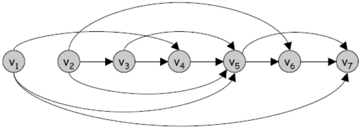
- Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j.
- Applications
  - Course prerequisite graph: course v_i must be taken before v_j
  - Compilation: must compile module v_i before v_j
  - Pipeline of computing jobs: output of job v_i is part of input to job v_j
  - Manufacturing or assembly: sand it before you paint it

Directed Acyclic Graphs

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



a DAG



topological ordering of the DAG

Topological Order $\Rightarrow$ DAG

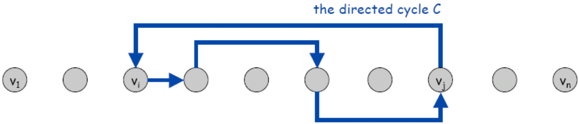
Lemma

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

Proof.

- Suppose that G has a topological order $v_1, \dots, v_n$ 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 $i < j$.
- But, since (v_j, v_i) is an edge and $v_1, \dots, v_n$ is a topological order, we must have $j < i$, a contradiction.

☐



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

DAG $\Rightarrow$ Topological Order ?

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?

DAGs have "source"

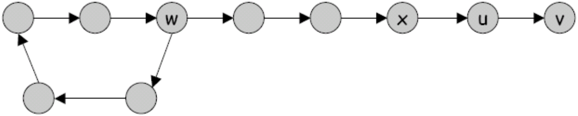
Lemma

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

Proof.

- 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.
- Let C be the sequence of nodes encountered between successive visits to w. C is a cycle.

☐



DAG $\Rightarrow$ Topological ordering

Lemma

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

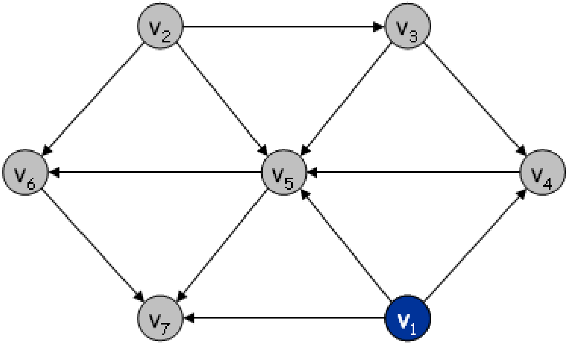
Proof.

By Induction on n, number of nodes

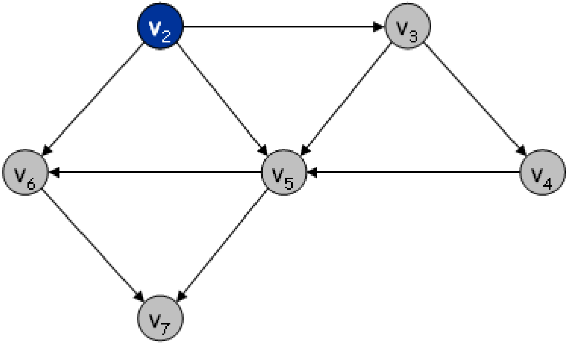
- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node v with no incoming edges.
- $G - \{v\}$ is a DAG, since deleting v cannot create cycles. By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place v first in topological ordering; then append nodes of $G - \{v\}$ in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

☐

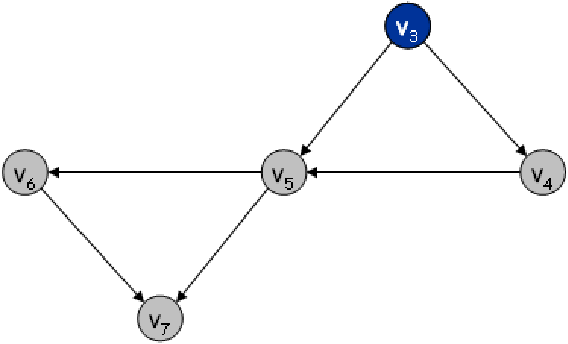
Topological Ordering Algorithm: Example



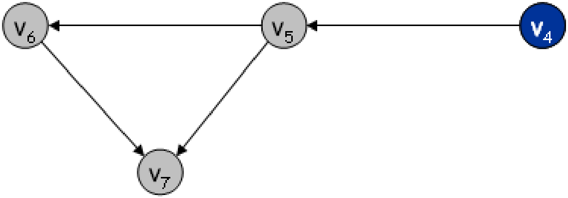
Topological Ordering Algorithm: Example



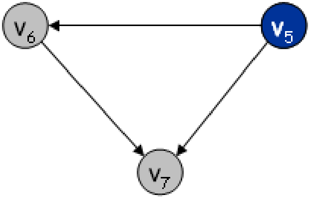
Topological Ordering Algorithm: Example



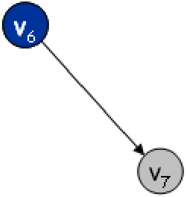
Topological Ordering Algorithm: Example



Topological Ordering Algorithm: Example



Topological Ordering Algorithm: Example



Topological Ordering Algorithm: Example

Topological Sorting Algorithm

```
$count[w] \leftarrow$ (remaining) number of incoming edges to $w$
$S \leftarrow$ set of (remaining) nodes with no incoming edges
while $S$ not empty do
    Remove some $v$ from $S$
    make $v$ next in topological order          ▷ O(1) per node
    for all edges from $v$ to some $w$ do        ▷ O(1) per edge
        decrement $count[w]$
        if $count[w] = 0$ then add $w$ to $S$
        end if
    end for
end while
```

- running time $O(m+n)$

Lecture Outline

1. BFS
    - Bipartite Graphs

2. DAGs & Topological Ordering

3. 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) – Recursive version

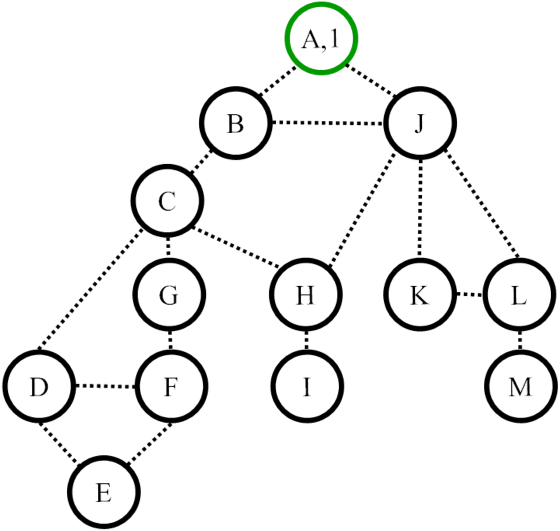
- for all nodes v, $v.dfs\# = -1$ ("undiscovered")
- dfscounter $= 0$

```
DFS($v$):
  $v.dfs\#$ =dfscounter++
  for all edge $(v, x)$ do
    if $x.dfs\# = -1$ then
      DFS($x$)
    else
      (code for back edges, etc.)
    end if
    Mark $v$ "completed"
  end for
```

DFS(v) - explicit stack

```
Initialize all vertices to "undiscovered"
Mark $v$ "discovered"
Push $(v, 1)$ onto stack
while stack not empty do
    $(u, i) = pop(stack)$
    while $i \leq deg(u)$ do
        $x \leftarrow i^{th}$ vertex on $u$'s edge list
        if $x$ undiscovered then
            mark $x$ "discovered"
            $push(u, i + 1)$
            $u \leftarrow x$
            $i \leftarrow 1$
        end if
    end while
```

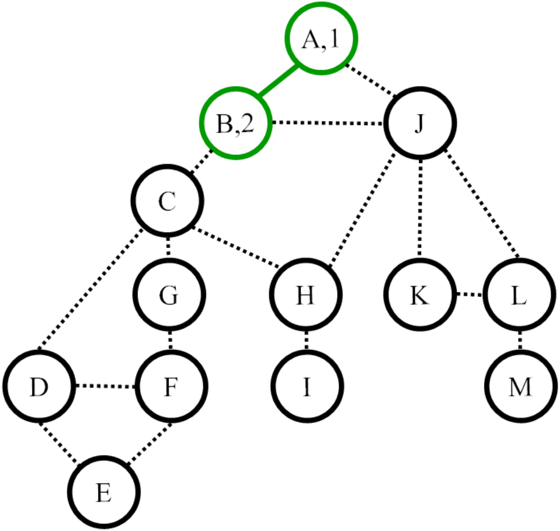
DFS(A) example



Call Stack
(Edge list):

A (B,J)

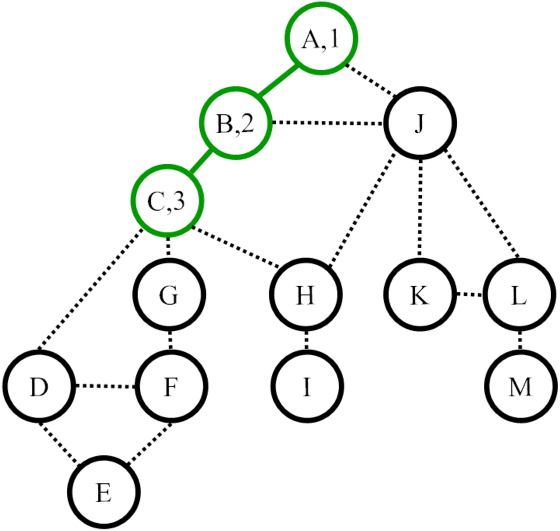
DFS(A) example



Call Stack:
(Edge list)

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

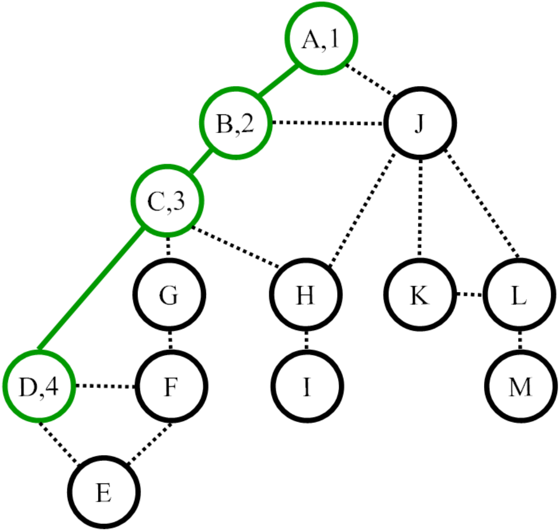
DFS(A) example



Call Stack:
(Edge list)

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

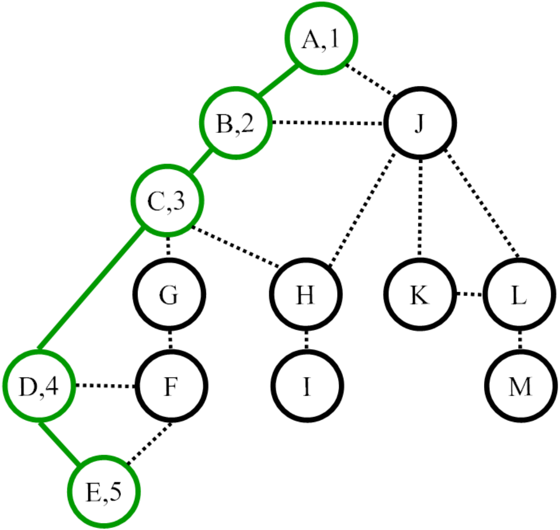
DFS(A) example



Call Stack:
(Edge list)

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

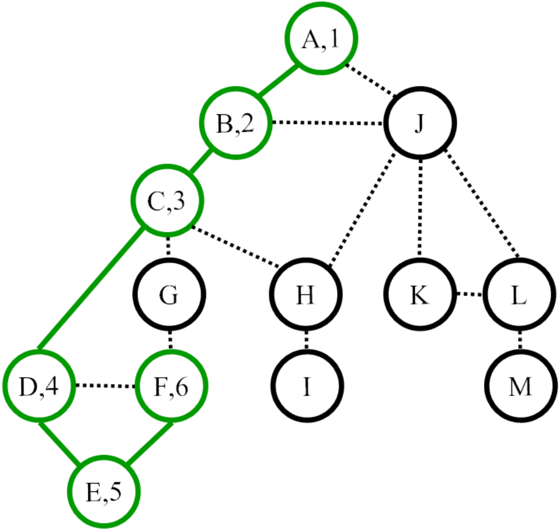
DFS(A) example



Call Stack:
(Edge list)

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

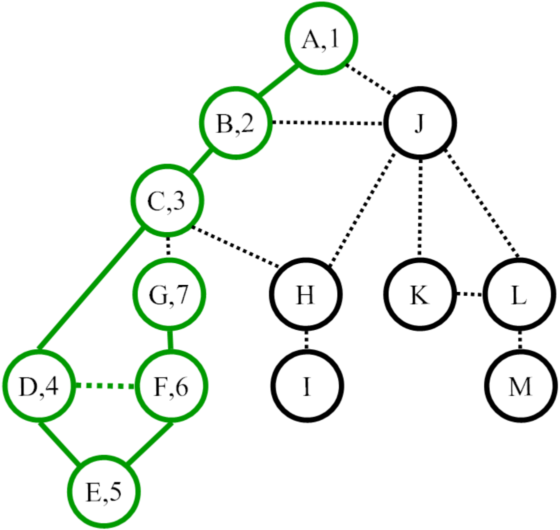
DFS(A) example



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)

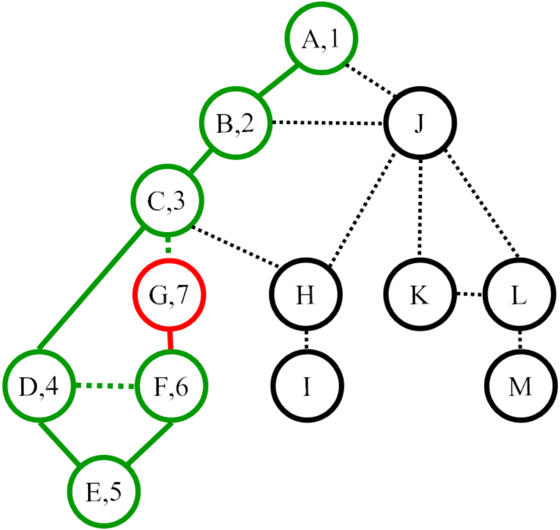
DFS(A) example



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)

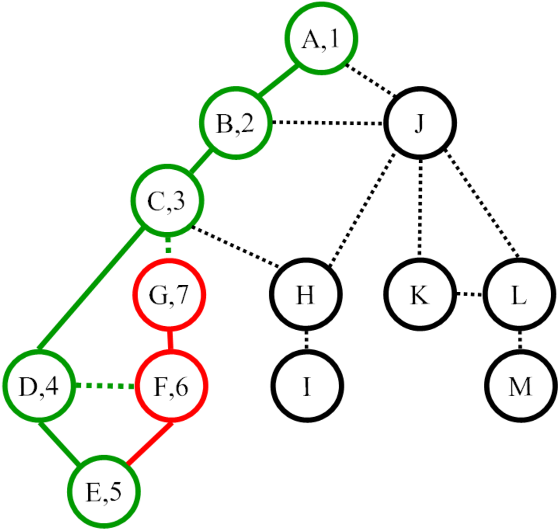
DFS(A) example



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~~)

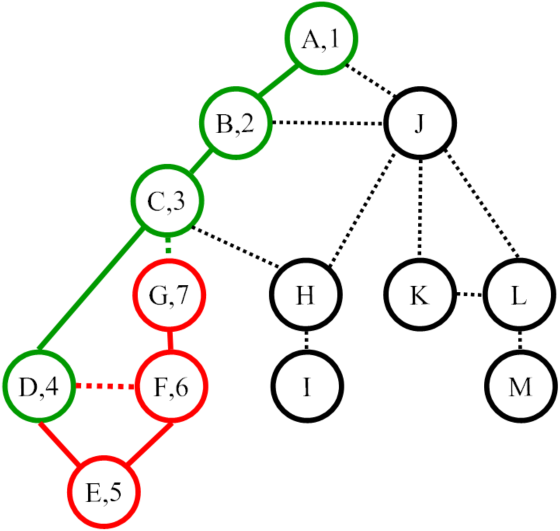
DFS(A) example



Call Stack:
(Edge list)

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

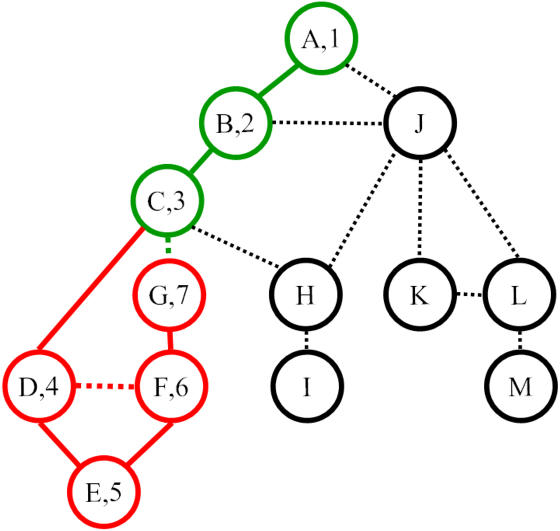
DFS(A) example



Call Stack:
(Edge list)

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

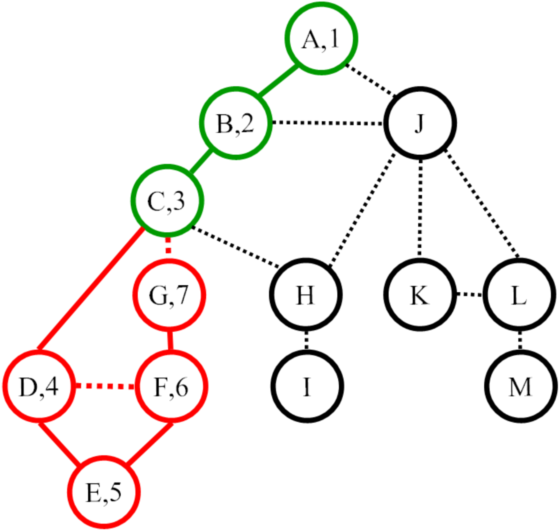
DFS(A) example



Call Stack:
(Edge list)

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

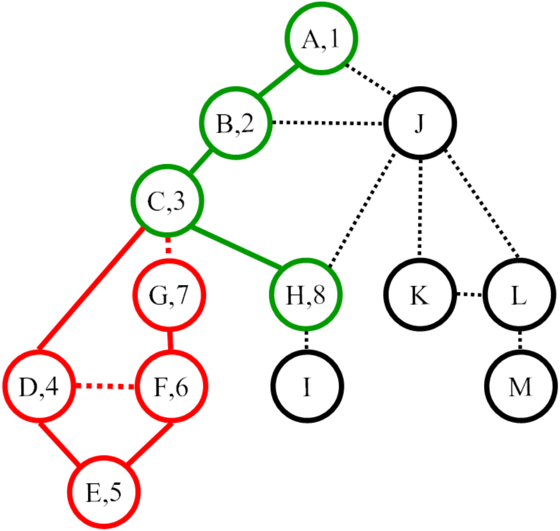
DFS(A) example



Call Stack:
(Edge list)

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

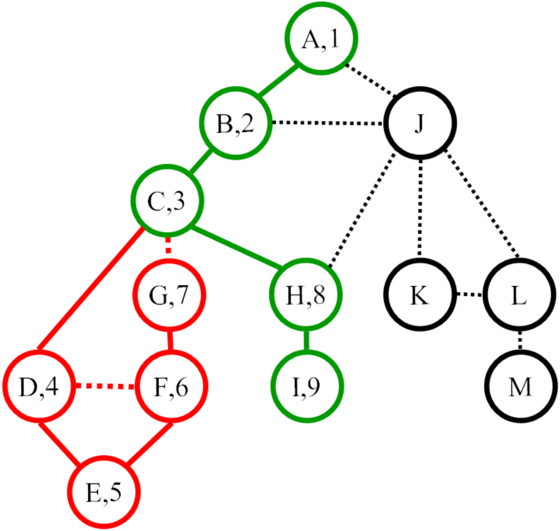
DFS(A) example



Call Stack:
(Edge list)

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

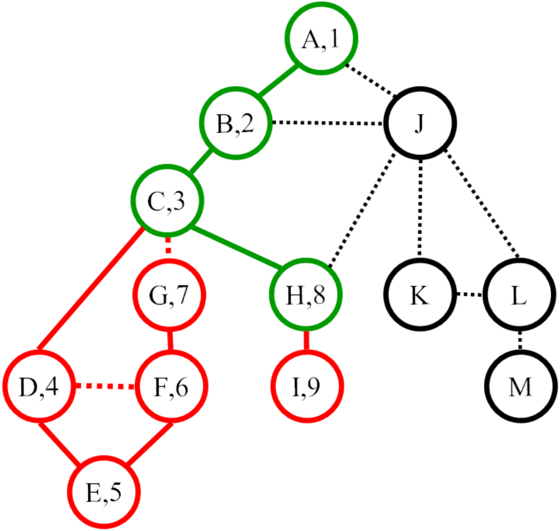
DFS(A) example



Call Stack:
(Edge list)

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

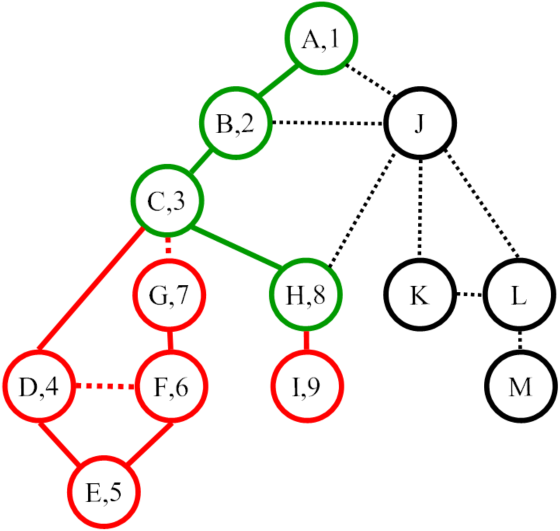
DFS(A) example



Call Stack:
(Edge list)

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

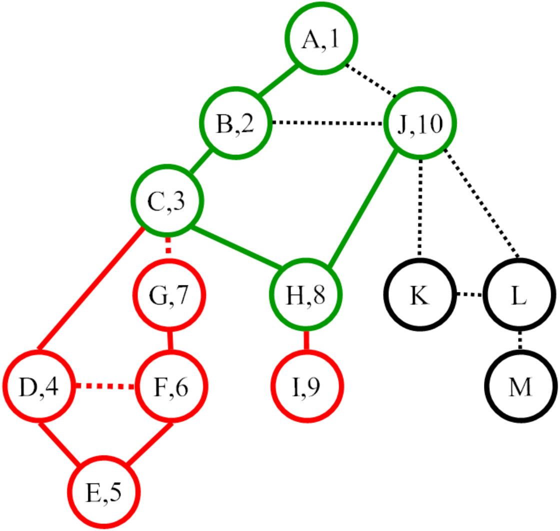
DFS(A) example



Call Stack:
(Edge list)

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

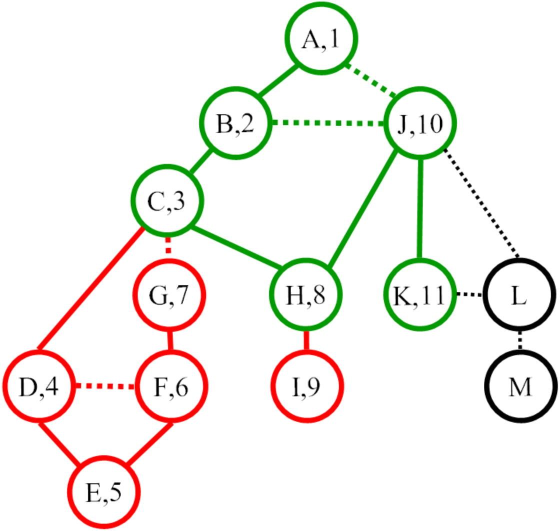
DFS(A) example



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)

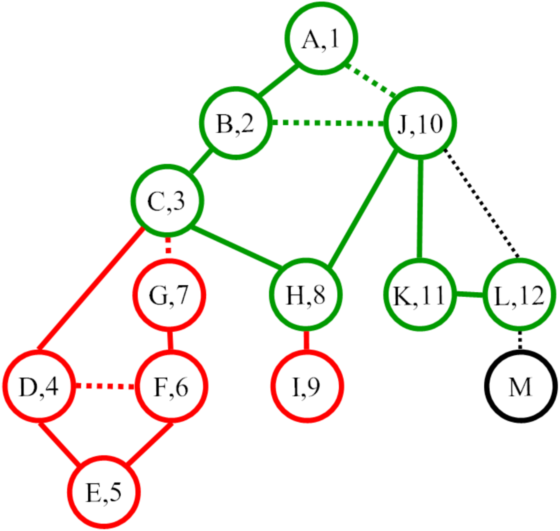
DFS(A) example



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)

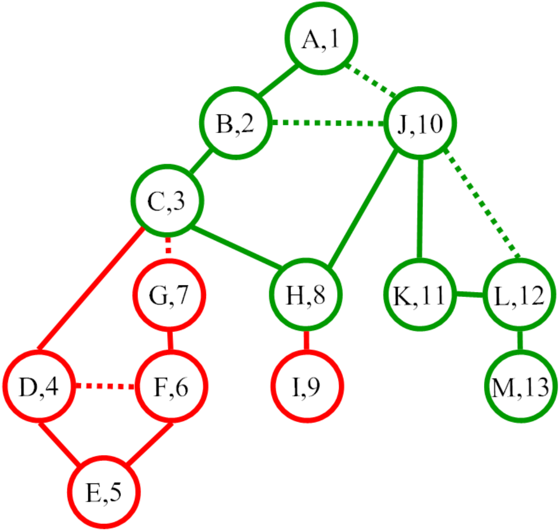
DFS(A) example



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)

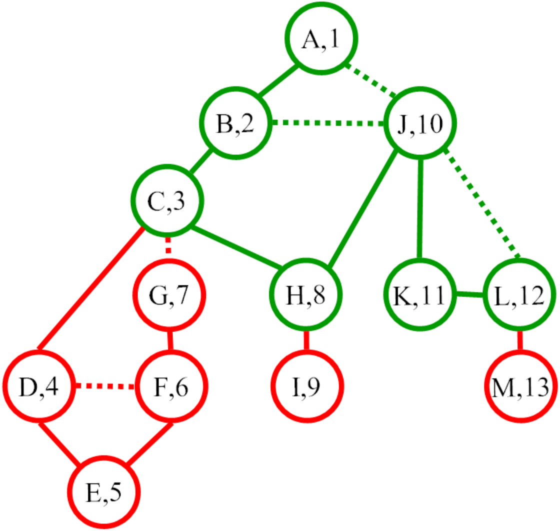
DFS(A) example



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)

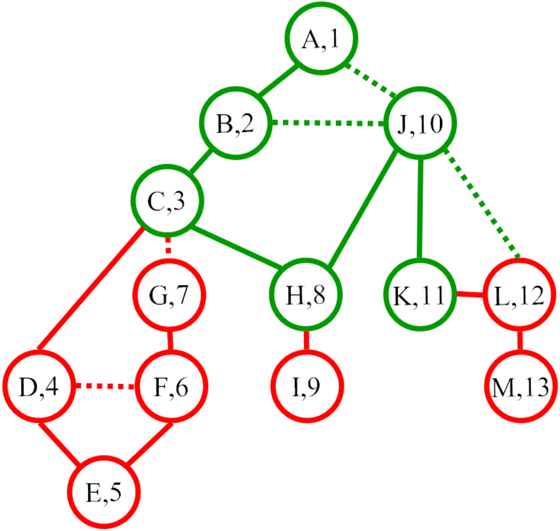
DFS(A) example



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~~)

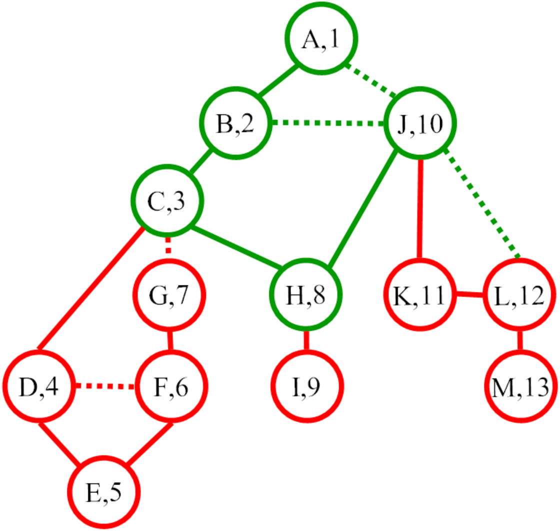
DFS(A) example



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~~)

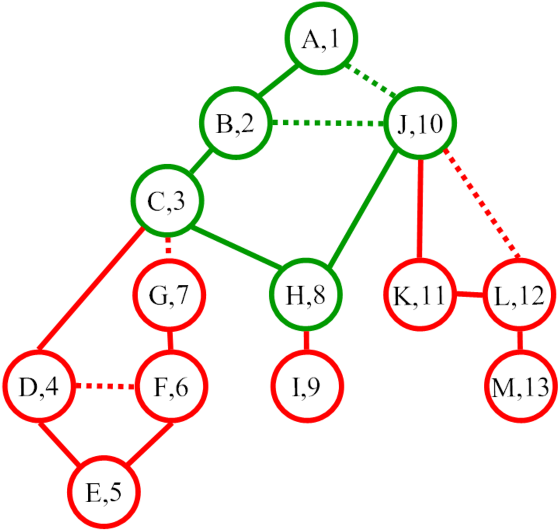
DFS(A) example



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)

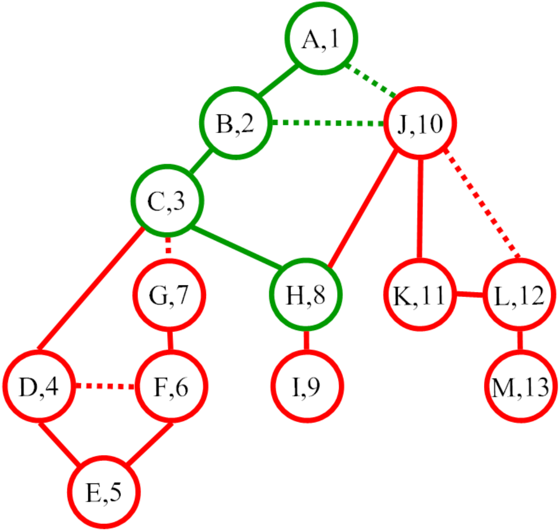
DFS(A) example



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~~)

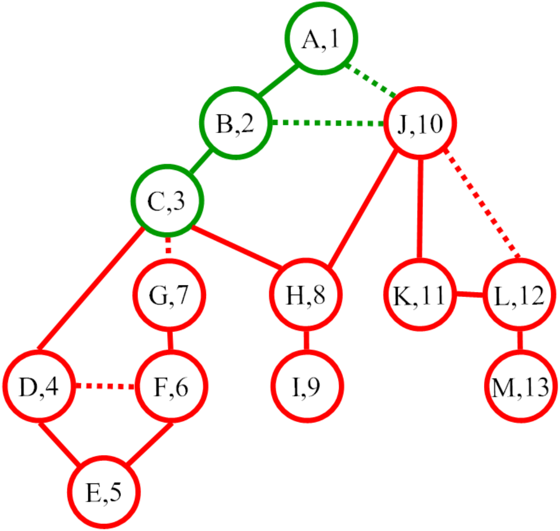
DFS(A) example



Call Stack:
(Edge list)

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

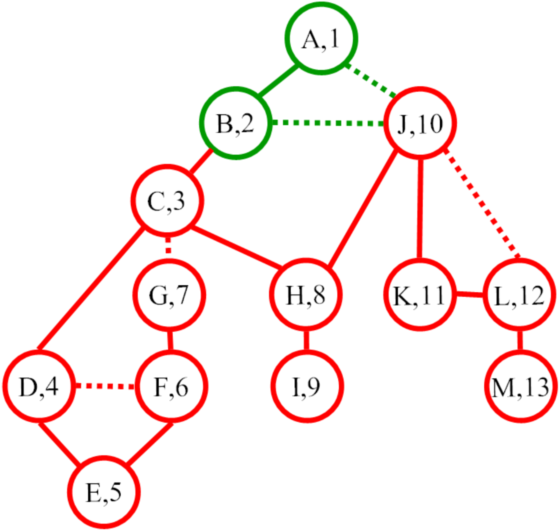
DFS(A) example



Call Stack:
(Edge list)

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

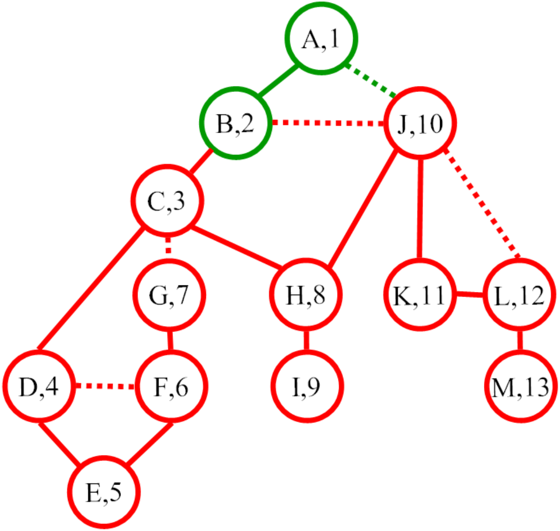
DFS(A) example



Call Stack:
(Edge list)

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

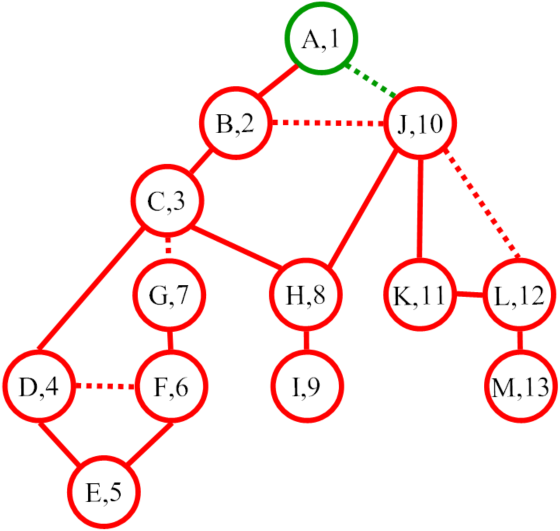
DFS(A) example



Call Stack:
(Edge list)

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

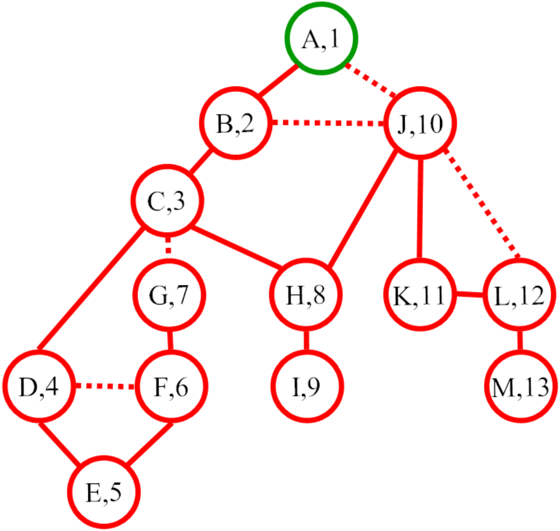
DFS(A) example



Call Stack:
(Edge list)

A (~~B~~,J)

DFS(A) example



Call Stack:
(Edge list)

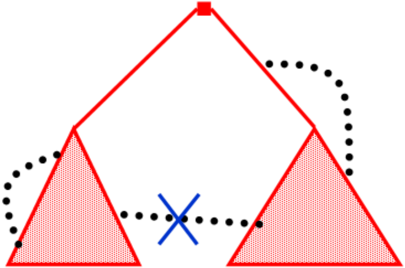
A (~~B~~,~~J~~)

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:

$$M(v) = \begin{cases} L(v) & \text{if } v \text{ is a leaf} \\ min(L(v), min_{w \in children(v)} M(w)) & \text{else} \end{cases}$$