Elementary Graph Algorithms

Foundations of Algorithms
Guven

Reading Assignment

- Cormen, Chapter 22

Graph Algorithms Outline

- Graph definitions
- DFS of graphs
- Biconnected components
- DFS of digraphs
- Finding articulation points

Graph Terminology

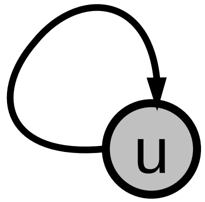
- Graph G = (V,E)
- Vertex set V
- Edge set E = pairs of adjacent vertices
- Incidence
    - pair (u,e) where $u \in V$, $e \in E$ e is incident to u
- Adjacency
    - two distinct incidences (u,e) and (v,f) are adjacent iff u=v, e=f or (u,v)=e or f

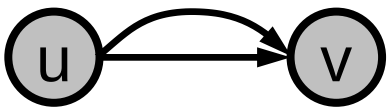
Directed and Undirected Graphs

- G directed
  - edges are ordered pairs (u,v)
  - also called digraph
- G undirected
  - edges are unordered pairs {u,v}

Proper Graphs and Subgraphs

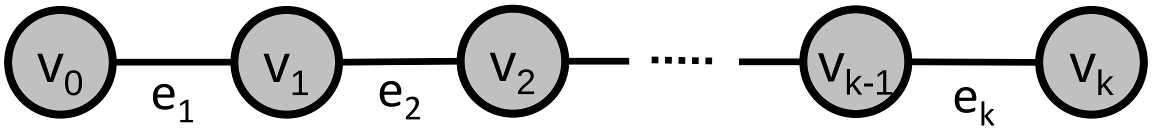
- Proper graph
  - no loops
  - no multi-edges





- Subgraph G' of G
  - G' = (V', E') such that $V' \subset V,\ E' \subset E$

Paths in Graphs



- Path p
- p is a sequence of vertices $v_0, \ldots, v_k$
  where for i=1,…, k, v_{i-1} is adjacent to v_i
- p is a sequence of edges $e_1, \ldots, e_k$
  where for i=2,…, k, edges e_{i-1} and e_i share a vertex

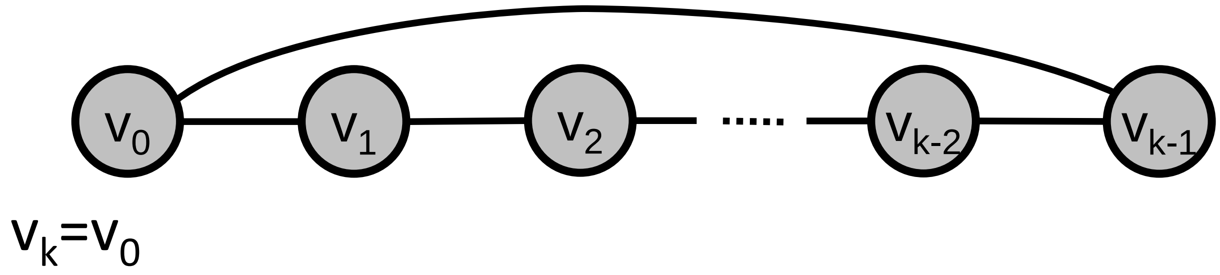
Simple Paths and Cycles

- Simple path

  no edge or vertex is repeated,

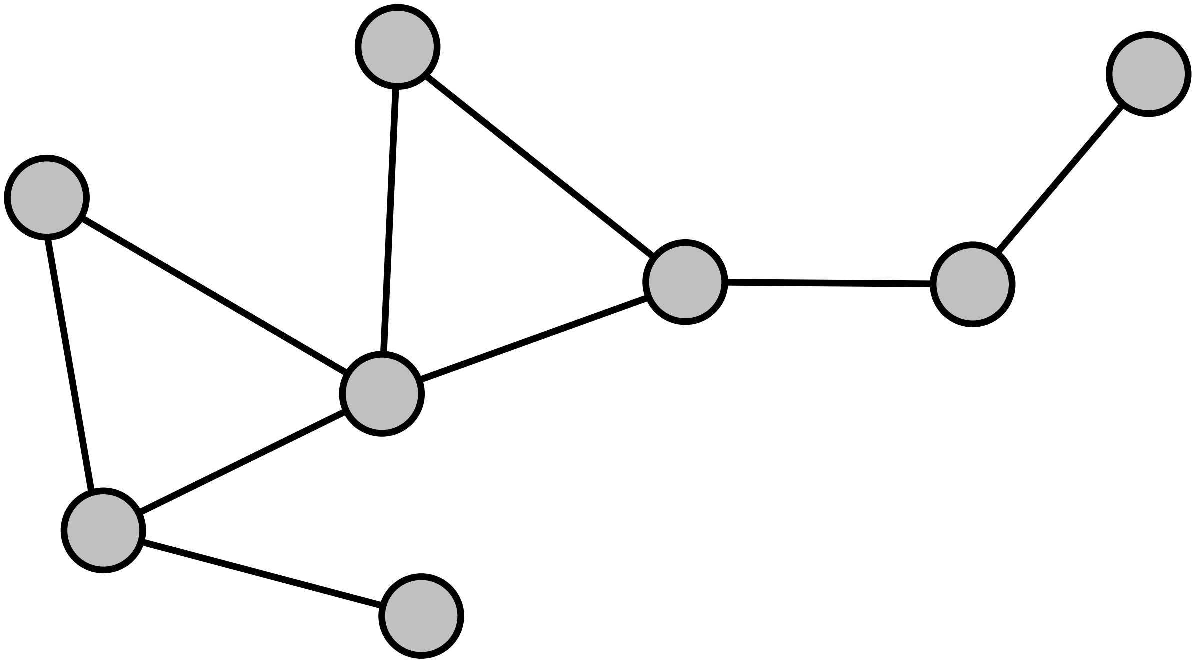
  except possibly $v_0 = v_k$
- Cycle

  a path p with $v_0 = v_k$ where k>1



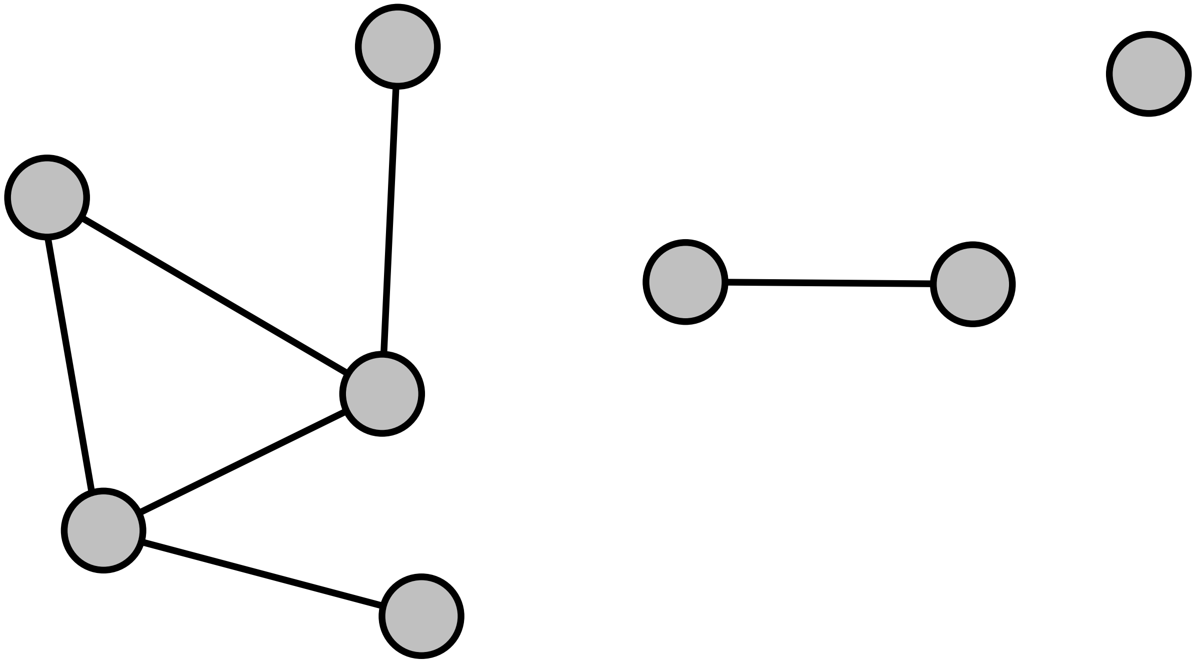
A Connected Undirected Graph

- G is connected if ∃ path between each pair of vertices



Connected Components of an Undirected Graph

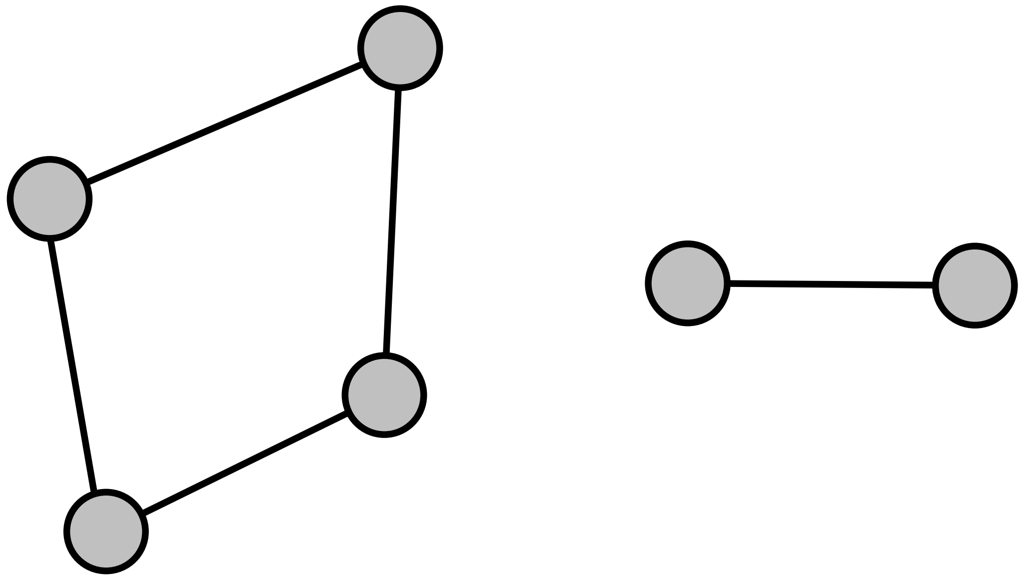
- Or G has ≥ 2 connected components, which are called maximal connected subgraphs



Biconnected Undirected Graphs

- G is biconnected if ∃ two disjoint paths between each pair of vertices

  G can be a single edge



Size of a Graph

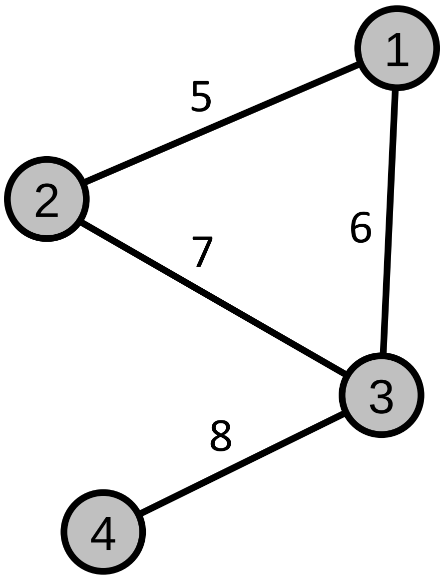
- Graph G = (V,E)

  n = |V| = # vertices

  m = |E| = # edges

  size of G is n+m

- Degree of vertices
  - deg(v) = number of edges for v
  - $\sum_{v \in V} \deg(v) = 2|E|$



Adjacency Matrix

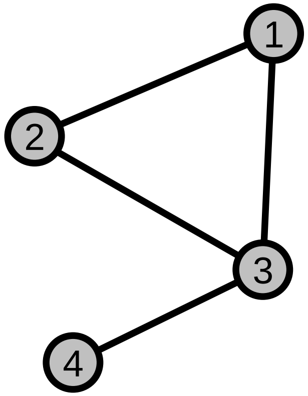
- Adjacency matrix A

  A is n-by-n square matrix

  $$A(i,j) = \begin{cases} 1 & (i,j) \in E \\ 0 & else \end{cases}$$

  space cost = n^2 - n

  - no need to store the diagonal which has to be always zero in a proper graph



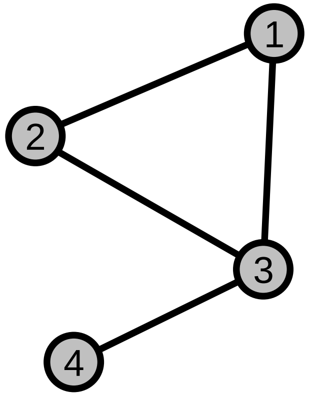
$$\mathrm{A} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}_{4\times 4}$$

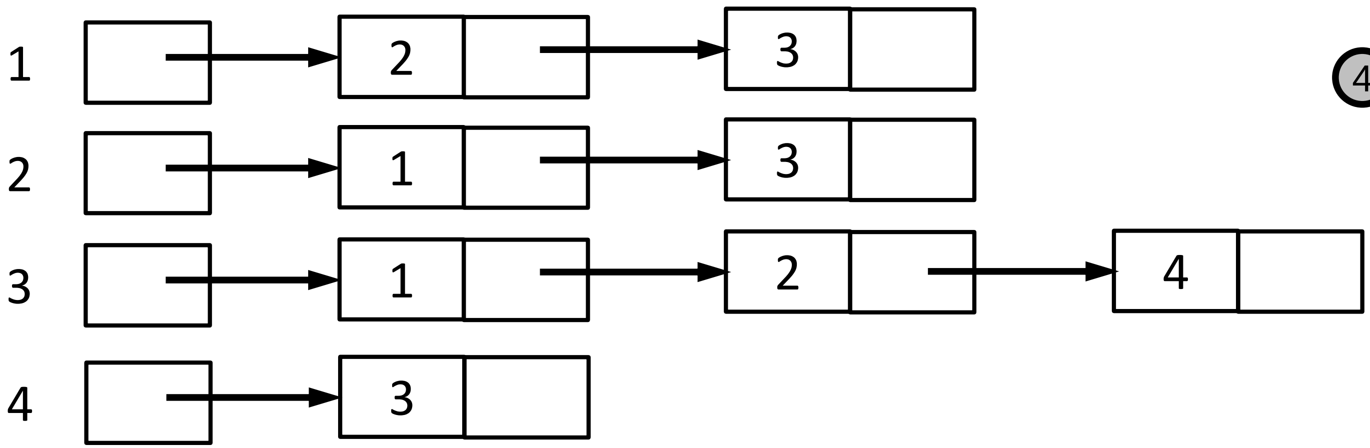
Adjacency List

- Adjacency lists Adj(1), …, Adj(n)

  Adj(v) = list of vertices adjacent to v

  space cost O(n+m)
- Note, the following example uses n+2m space





Definition of an Undirected Tree

- Tree T

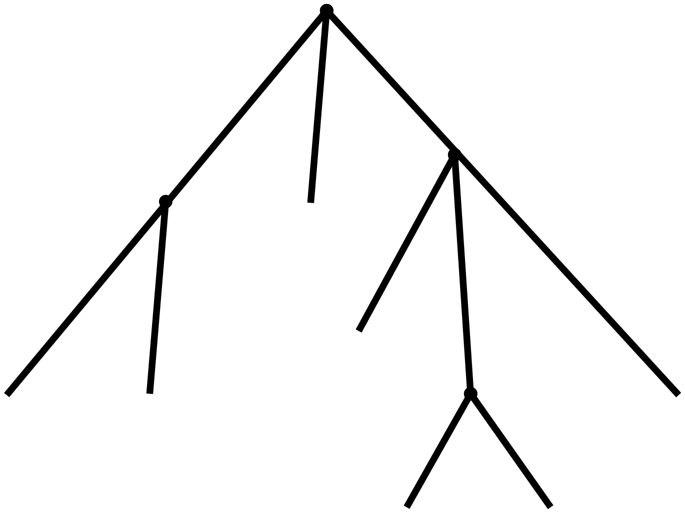
  T is a graph with unique path between every pair of its vertices

  k = # vertices

  k-1 = # edges

- Forest

  set of trees

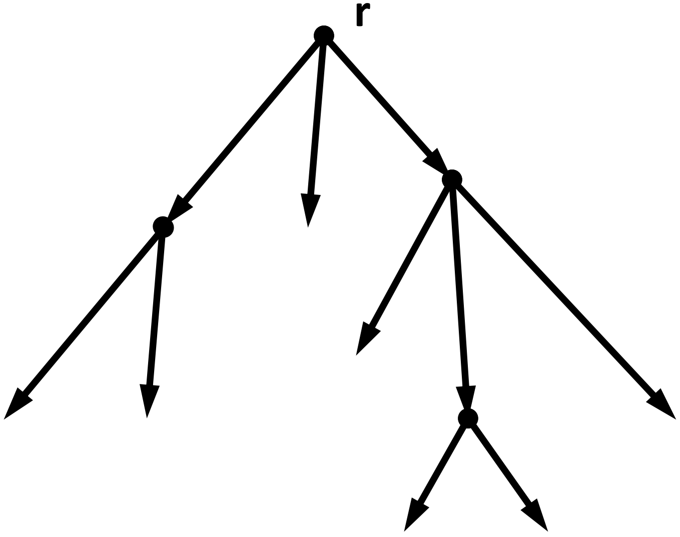


Definition of a Directed Tree

- Directed tree

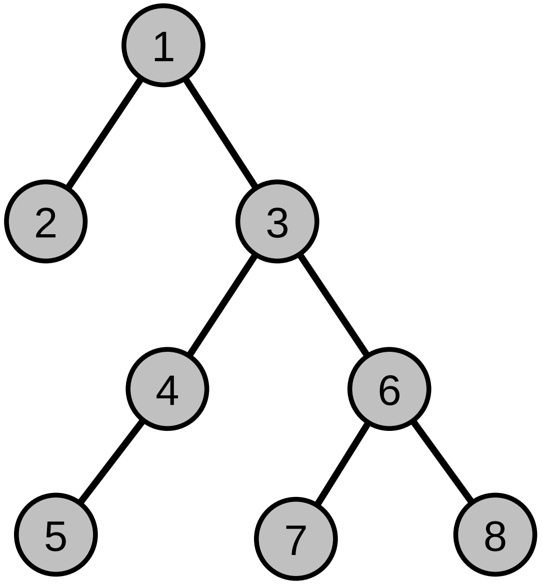
  T is a digraph with distinguished vertex root r such that each vertex is reachable from r by a unique path

  - Family relationships
    - ancestors
    - descendants
    - parent
    - child
    - siblings
    - leaves have no proper descendants



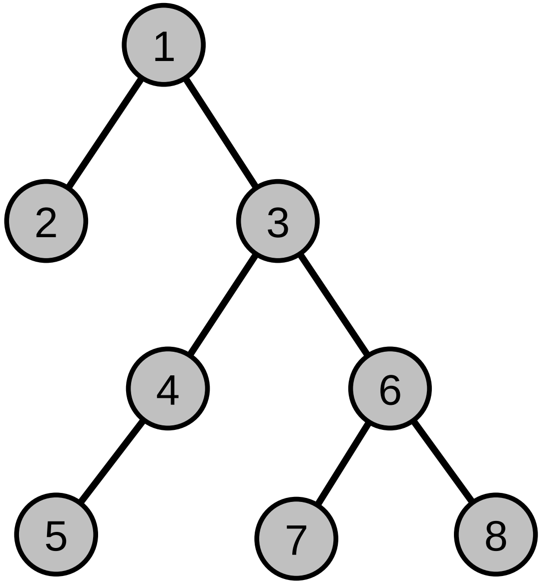
An Ordered Tree

- Ordered Tree is a directed tree with siblings ordered



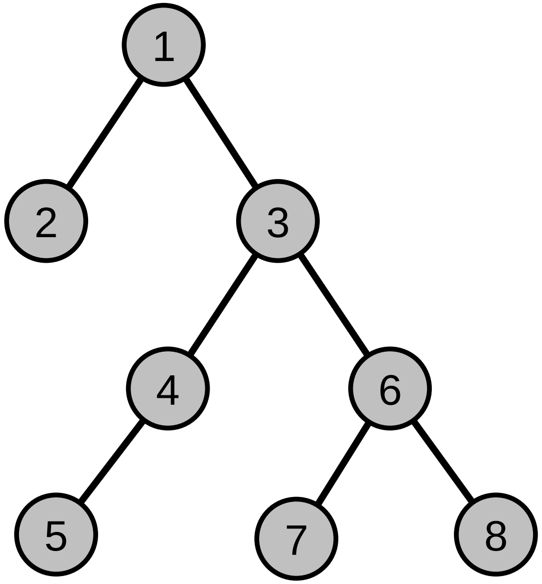
Preorder Recursive Tree Traversal

- Preorder : 1, 2, 3, 4, 5, 6, 7, 8
  1. root (order vertices as pushed on a stack)
  2. preorder left subtree
  3. preorder right subtree



Postorder Recursive Tree Traversal

- Postorder: 2, 5, 4, 7, 8, 6, 3, 1
  1. postorder left subtree
  2. postorder right subtree
  3. root (order vertices as popped off stack)



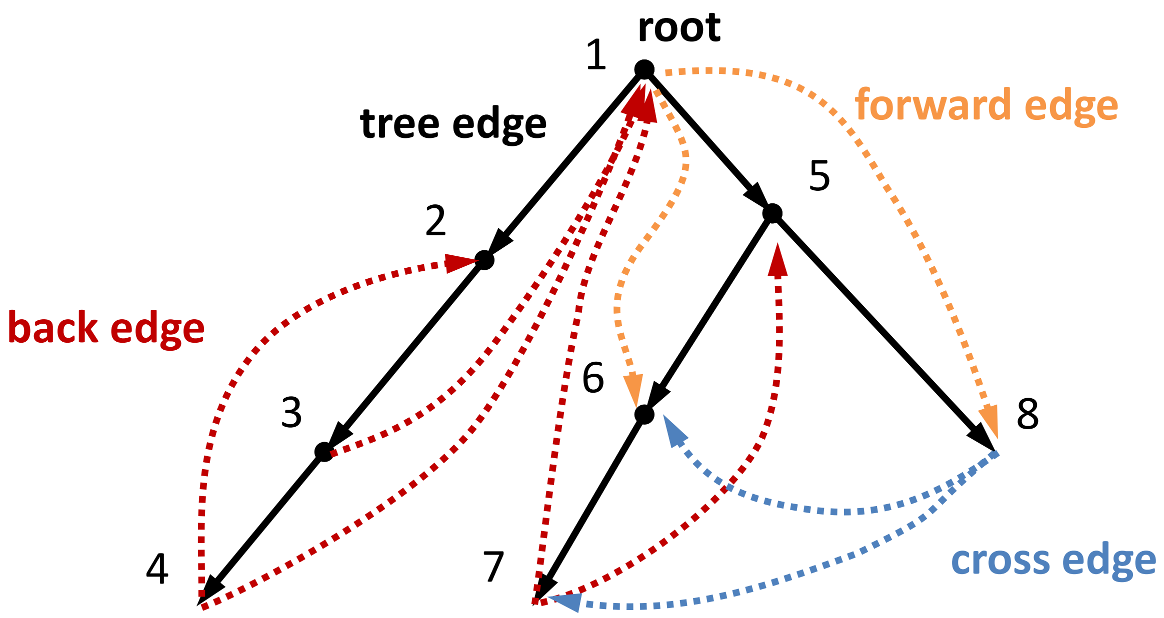
Spanning Tree and Forest of a Graph

- T is a spanning tree of graph G if
  1. T is a directed tree with the same vertex set as G
  2. $e \in T$ is a directed version of an $e \in G$

  Size T = n + (n-1)
- Spanning forest

  forest of spanning trees of connected components of G

Example Spanning Tree of a Graph



Classification of Edges of G with a Spanning Tree T

- An edge (u,v) of T is a tree edge
- An edge (u,v) of **G – T** is a back edge if u is a descendent or ancestor of v
- Else an edge (u,v) of **G – T** is a cross edge
- G = T + back edges + cross edges
  (Note: directions of edges were ignored)

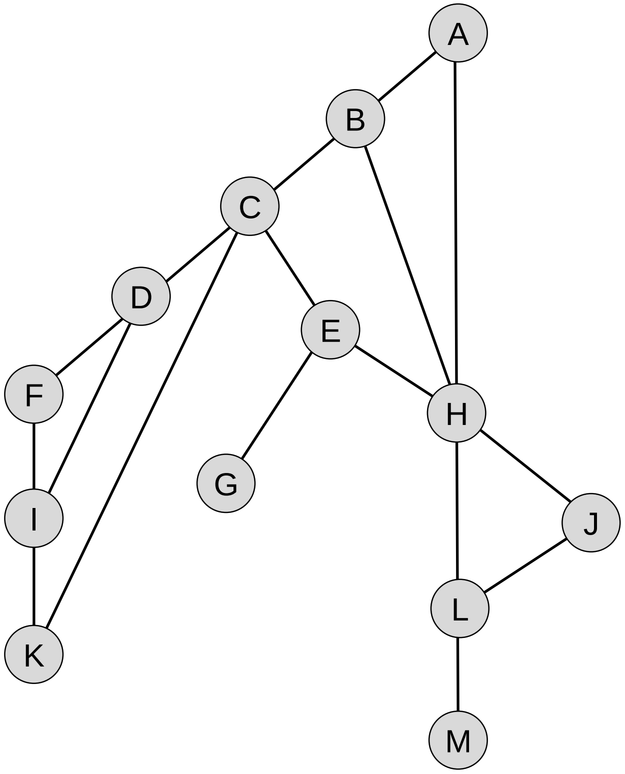
Recursive Depth First Search Algorithm

```
def init():
  # Introduce dfsnum attribute of the vertices V
  set all v.dfsnum = -1  # node is undiscovered
  dfscounter = 1
def dfs(v):
  # starts from root or first vertex
  v.dfsnum = dfscounter++
  for each edge(v,x):
    if x.dfsnum == -1:
      call dfs(x)  # recursive
    else:  # have seen/past this node previously
      process_further()  # process for back edges, etc.
```

Recursive DFS Example

- Adjacency list

```
A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅
```

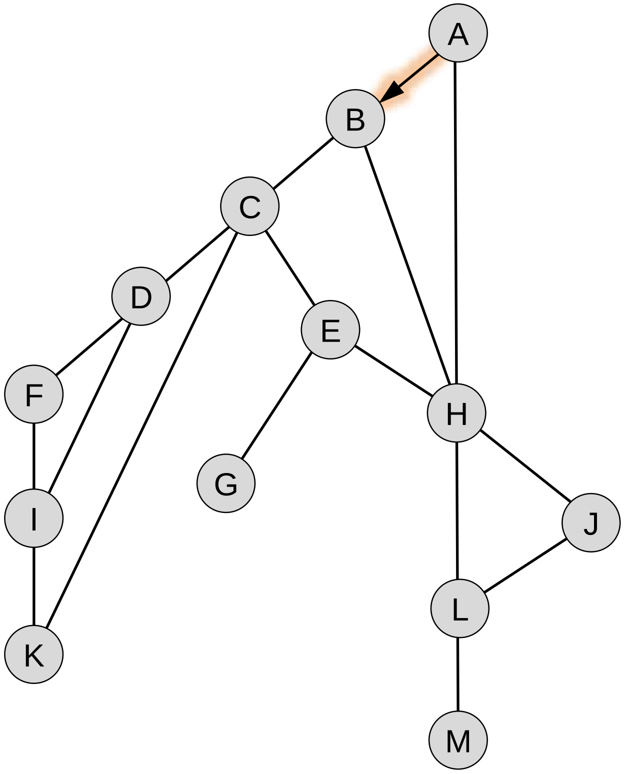


Recursive DFS Example (cont.)

DFS

A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅

Vertex	dfs#
A	1
B	
C	
D	
E	
F	
G	
H	
I	
J	
K	
L	
M	



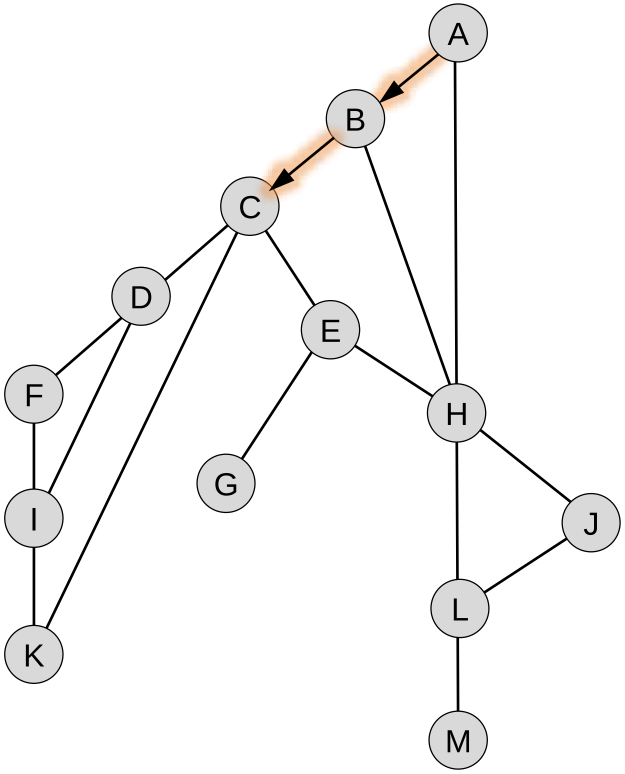
Recursive DFS Example (cont.)

DFS | new v from recursion

seen before, **`dfs#!=-1`**

```
A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅
```

Vertex	dfs#
A	1
B	2
C	
D	
E	
F	
G	
H	
I	
J	
K	
L	
M	



Recursive DFS Example (cont.)

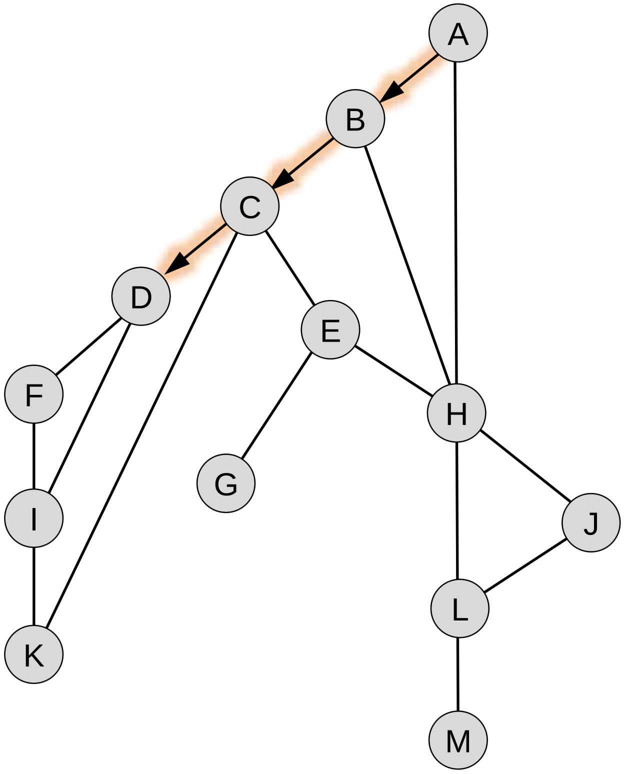
DFS

new v from recursion

seen before, **dfs#!=-1**

```
A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅
```

Vertex	dfs#
A	1
B	2
C	3
D	
E	
F	
G	
H	
I	
J	
K	
L	
M	



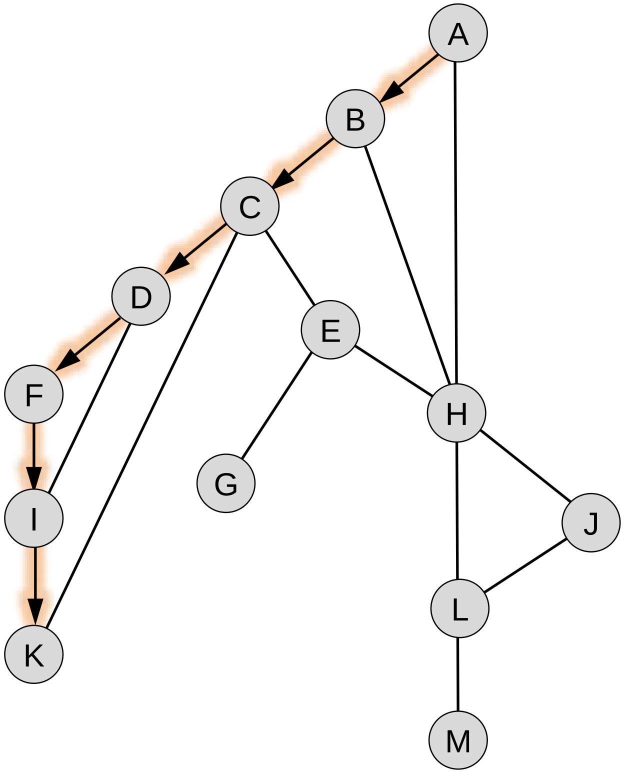
Recursive DFS Example (cont.)

DFS | new v from recursion

seen before, **dfs#!=-1**

A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅

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



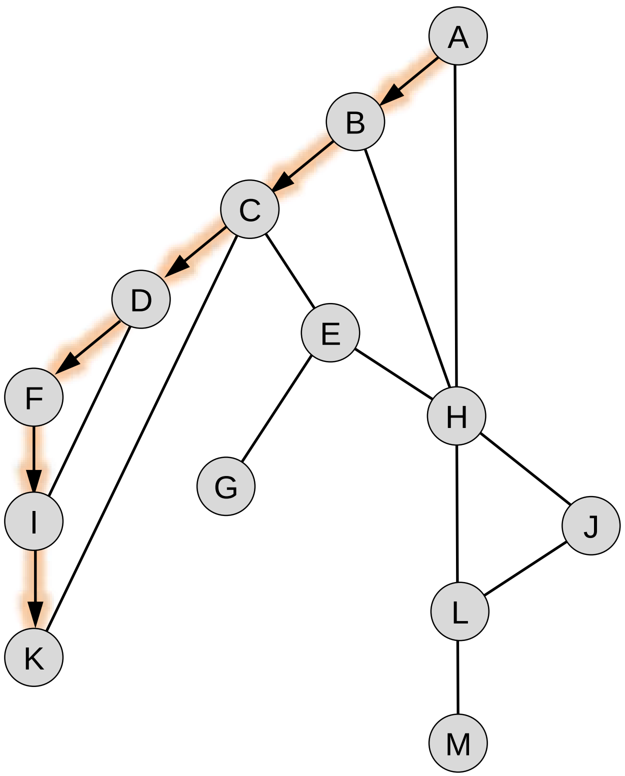
Recursive DFS Example (cont.)

DFS | new v from recursion

seen before, **dfs#!=-1** | back to **for** loop

```
A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅
```

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



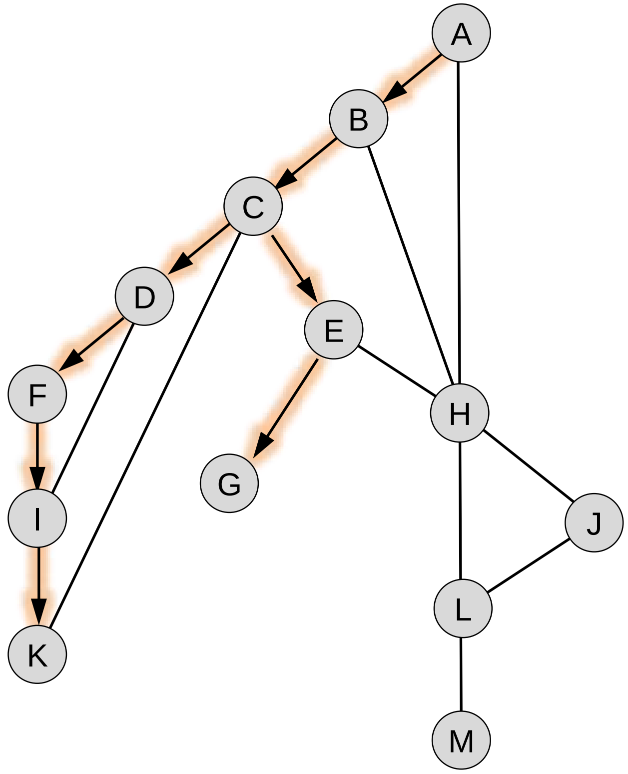
Recursive DFS Example (cont.)

DFS | new v from recursion

seen before, **dfs#!=-1** | back to **for** loop

```
A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅
```

Vertex	dfs#
A	1
B	2
C	3
D	4
E	8
F	5
G	9
H	
I	6
J	
K	7
L	
M	



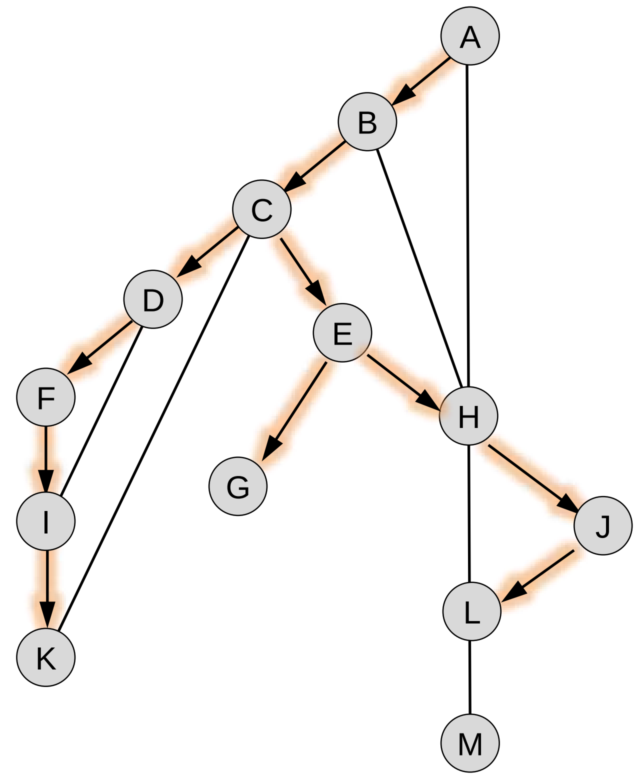
Recursive DFS Example (cont.)

DFS | new v from recursion

seen before, **`dfs#!=-1`** | back to **`for`** loop

A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅

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



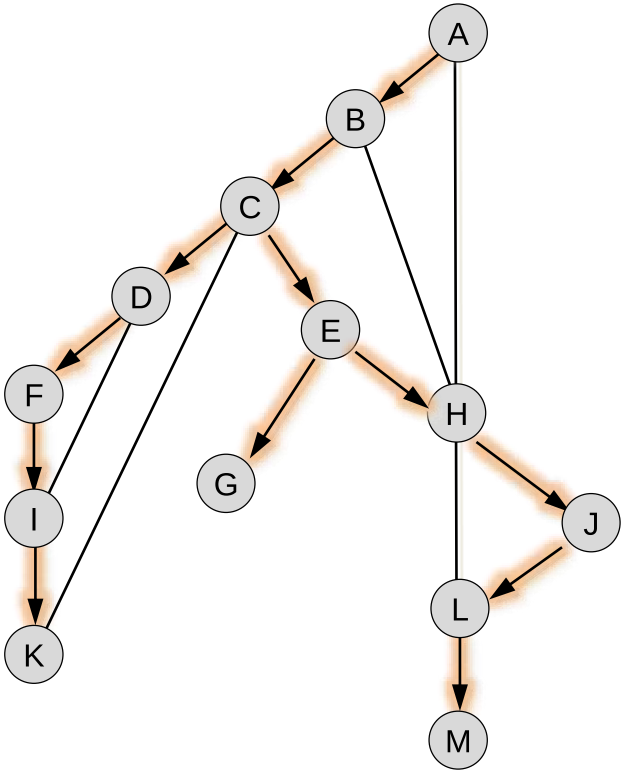
Recursive DFS Example (cont.)

DFS | new v from recursion

seen before, **`dfs#!=-1`** | back to **`for`** loop

```
A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅
```

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



Time Cost of DFS Algorithm

- Input size n = |V|, m = |E|
- **Theorem**
  Depth First Search Algorithm is O(n+m)
- **Proof**
  Each edge and vertex is associated with constant number of operations

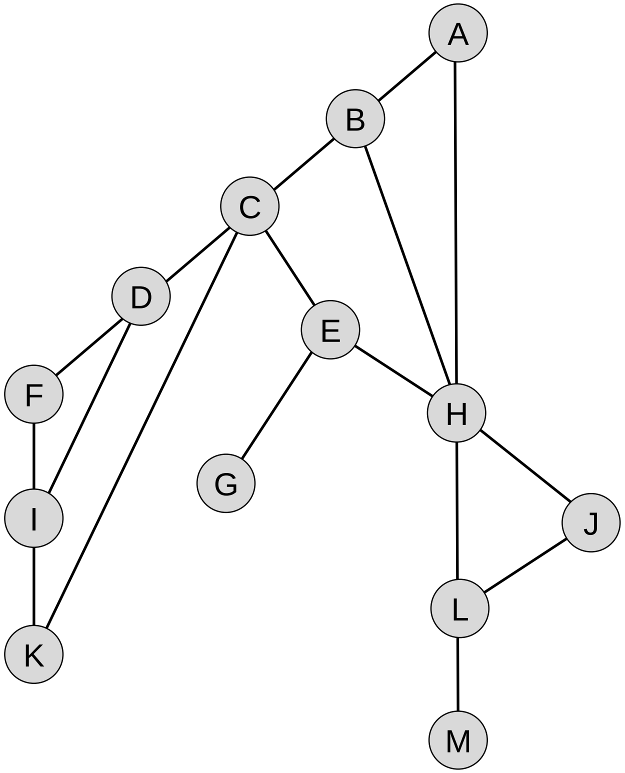
Breadth First Search Algorithm

```
def bfs():
  set all v.bfsnum = -1  # node is undiscovered
  bfscounter = 1
  Q.enqueue(root)  # A queue Q is used for BFS
  root.bfsnum = bfscounter++
  while Q is not empty:
    v = Q.dequeue()  # pick first element from queue
    for each edge(v,x):
      if x.bfsnum == -1:  # x is not visited
        Q.enqueue(x)
        x.bfsnum = bfscounter++
```

BFS Example

- Adjacency list

```
A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅
```



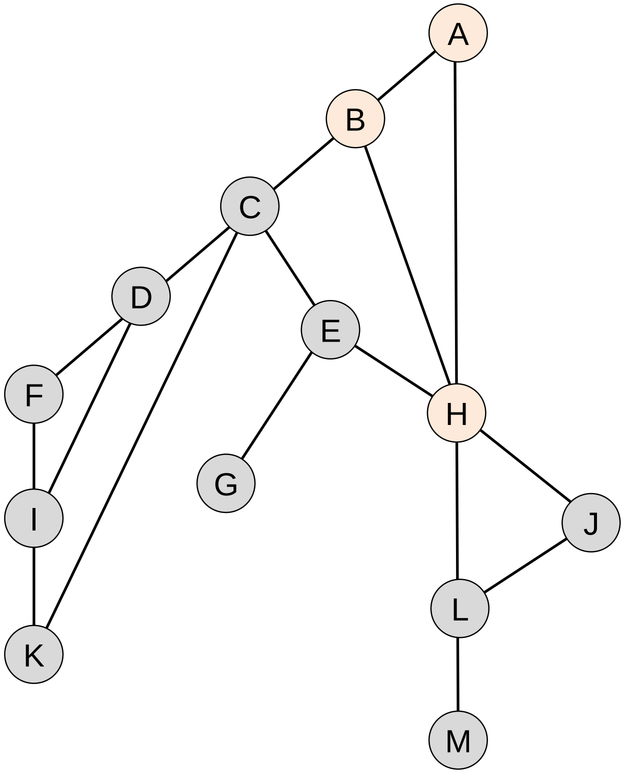
BFS Example (cont.)

BFS

in queue: B, H

A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅

Vertex	bfs#
A	1
B	2
C	
D	
E	
F	
G	
H	3
I	
J	
K	
L	
M	



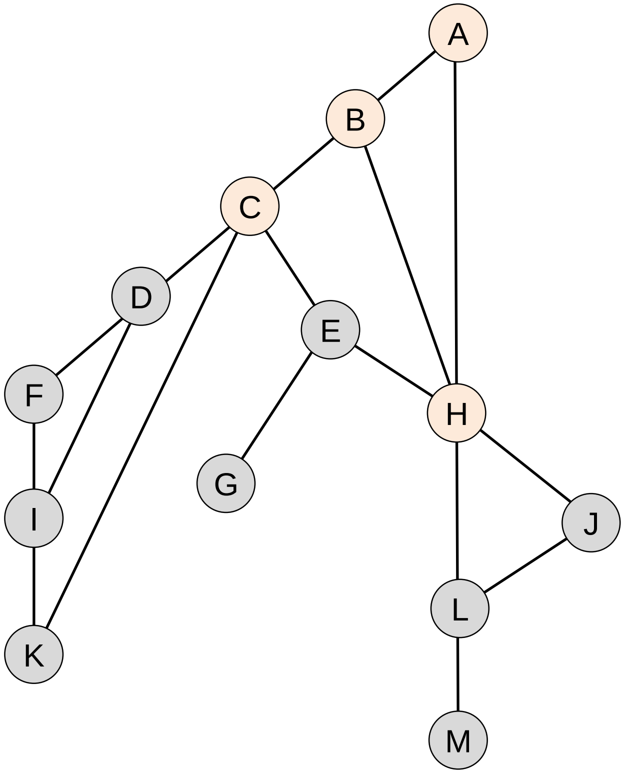
BFS Example (cont.)

BFS | in queue: H, C

seen before, **`bfs#!=-1`**

```
A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅
```

Vertex	bfs#
A	1
B	2
C	4
D	
E	
F	
G	
H	3
I	
J	
K	
L	
M	



BFS Example (cont.)

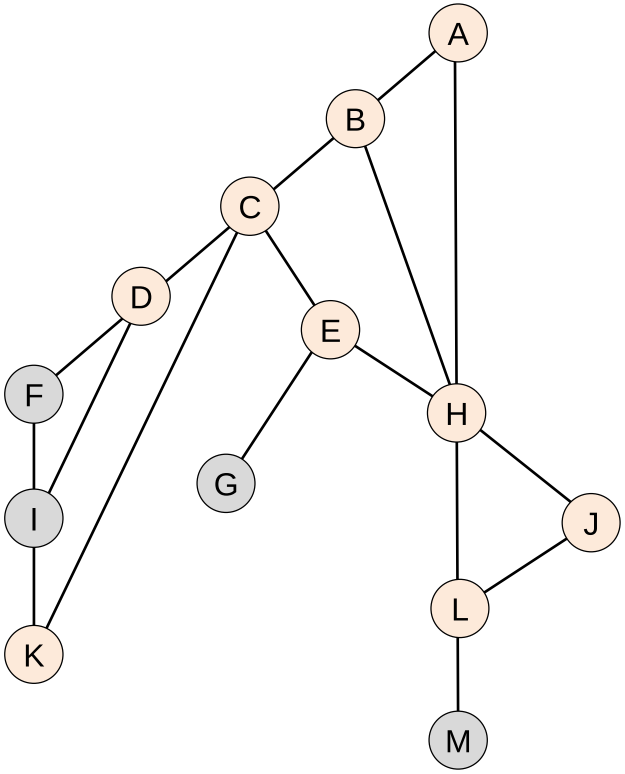
BFS

in queue: E, J, L, D, K

seen before, **`bfs#!=-1`**

```
A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅
```

Vertex	bfs#
A	1
B	2
C	4
D	8
E	5
F	
G	
H	3
I	
J	6
K	9
L	7
M	



BFS Example (cont.)

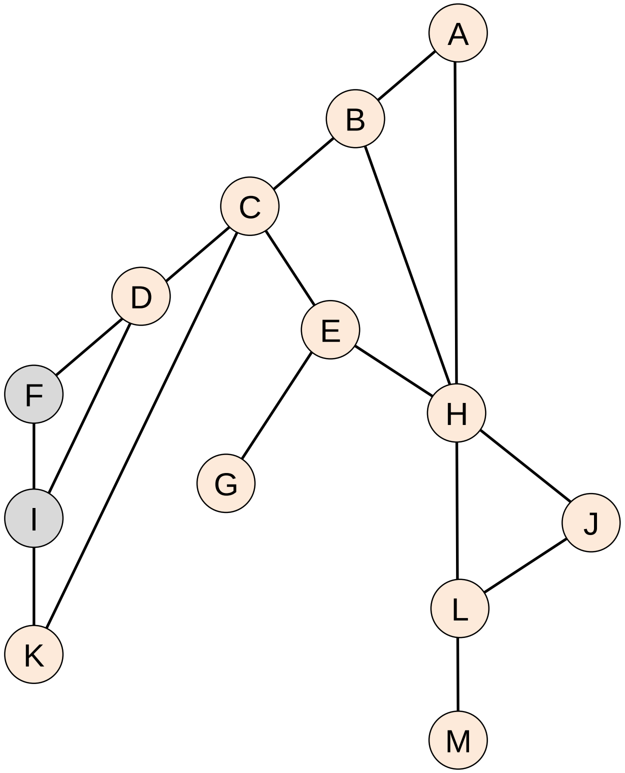
BFS

in queue: D, K, G, M

seen before, **`bfs#!=-1`**

```
A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅
```

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



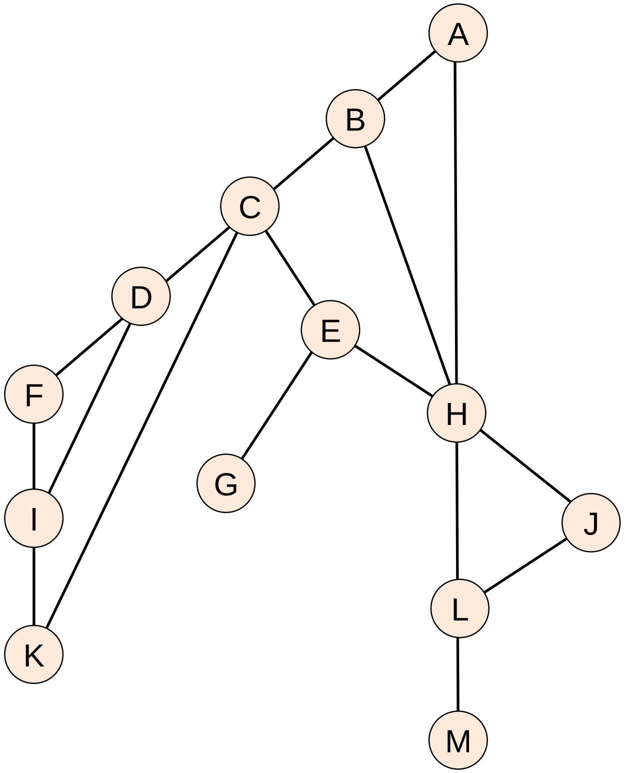
BFS Example (cont.)

BFS | in queue: K, G, M, F, I

seen before, **`bfs#!=-1`**

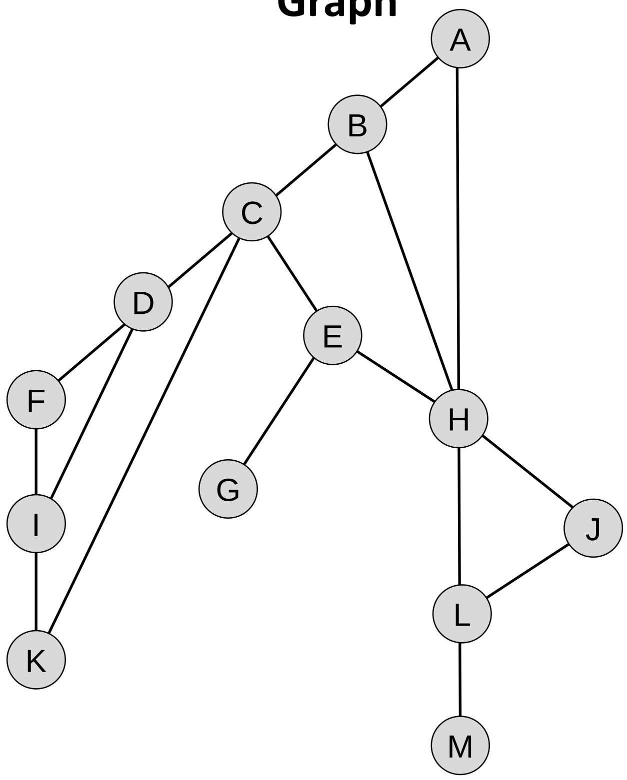
```
A →B →H →∅
B →A →C →H →∅
C →B →D →E →K →∅
D →C →F →I →∅
E →C →G →H →∅
F →D →I →∅
G →E →∅
H →A →B →E →J →L →∅
I →D →F →K →∅
J →H →L →∅
K →C →I →∅
L →H →J →M →∅
M →L →∅
```

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

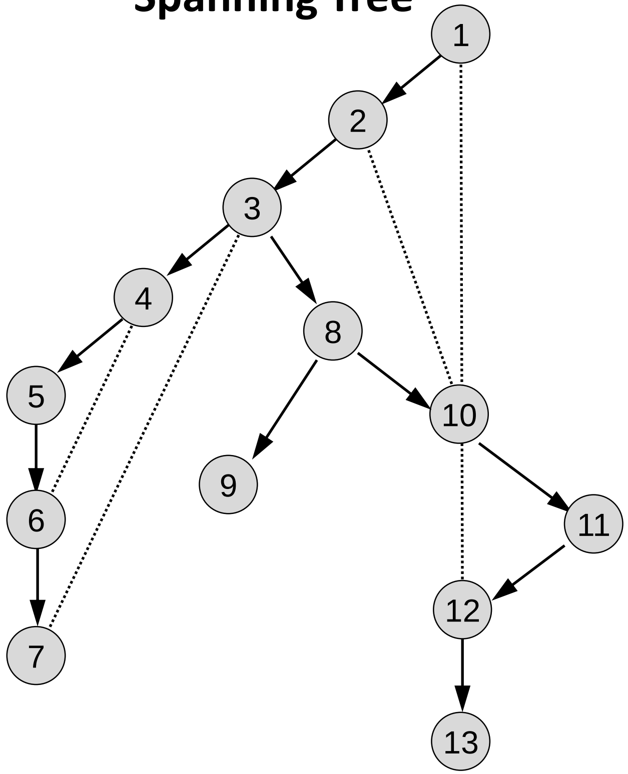


Classification of Edges of a Graph via DFS Spanning Tree

Graph



Spanning Tree



Classification of Edges of a Graph via DFS Spanning Tree (cont.)

- Edge notation induced by DFS Tree T
- $u \rightarrow v$ iff (u,v) is a tree edge of T
- $u \xrightarrow{*} v$ iff u is an ancestor of v
- $u \cdots v$ iff (u,v) is a back edge,
  where $(u,v) \in G - T$ with either $u \xrightarrow{*} v$ or $v \xrightarrow{*} u$
- Every vertex is an ancestor and a descendant of itself

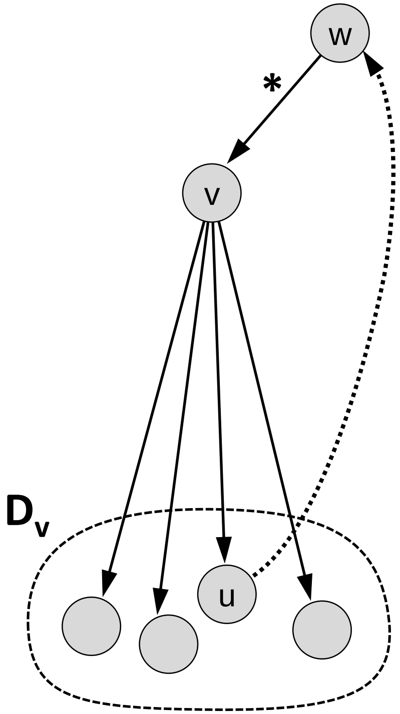
Descendant Vertices via Preordering of a Tree

- Use v.dfs# determined by DFS

  Notation: v.d = v.dfs#
- Let $|D_v|$ = # of descendants of v
- **Lemma**

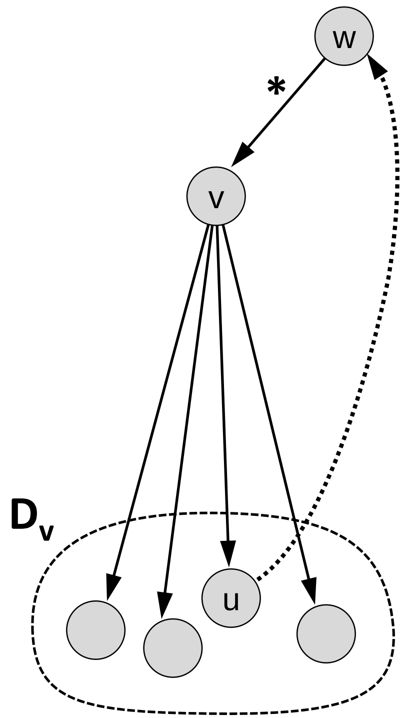
  u is descendant of v iff

  $v.d \leq u.d \leq v.d + |D_v|$



Proper Ancestors via Preordering of a Tree

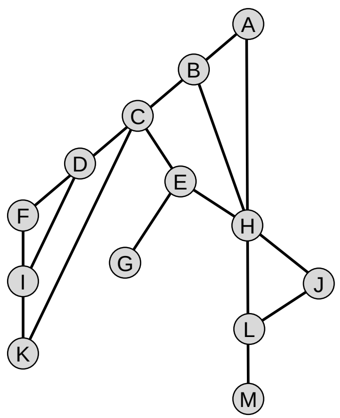
- **Lemma**
  If u is descendant of v
  and (u,w) is a back edge s.t. w.d < v.d
  then w is a proper ancestor of v

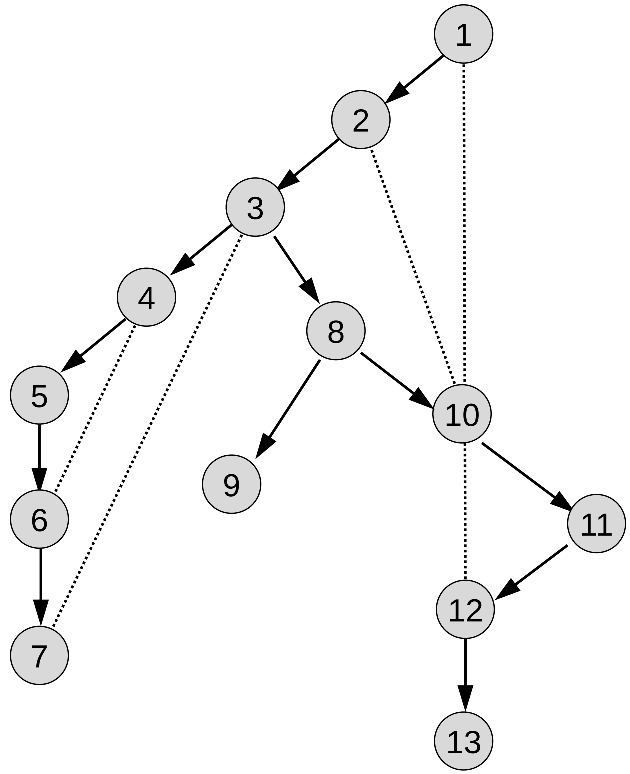


Low Values

- For each vertex v, define
  low(v) = min{v.d, w.d | (u,w) is a back edge for some descendant u of v}
  $\mathrm{low}(v) = \min\{v.d,\ w.d \mid v \xrightarrow{*} \cdots w\}$
- **Lemma**
  $\mathrm{low}(v) = \min\{v.d,\ \{\mathrm{low}(u) \mid v \rightarrow u\},\ \{w.d \mid u \rightarrow \cdots w\}\}$
  (Proof by induction)
- low(v) can be computed during DFS in postorder

Example: Low Values of G





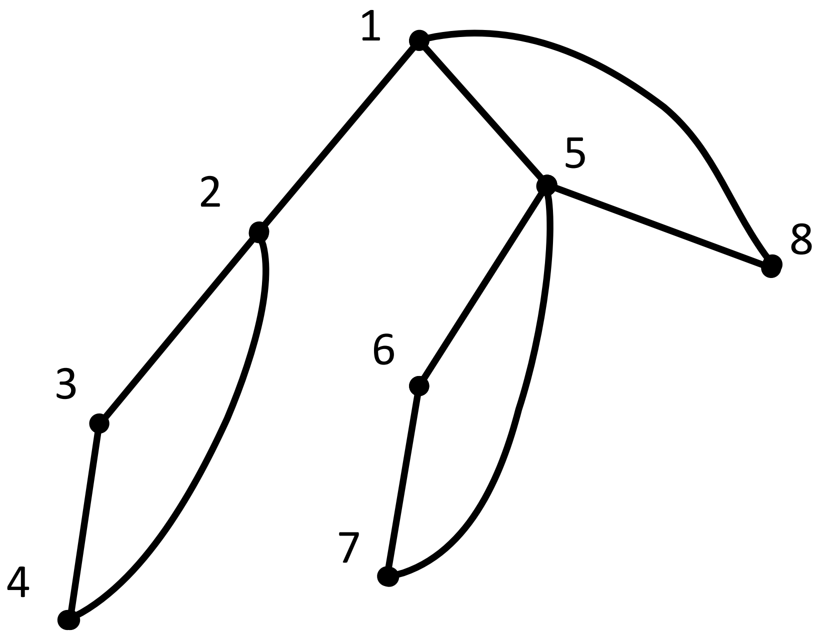
Vertex	dfs#	low
A	1	1
B	2	1
C	3	1
D	4	3
E	8	1
F	5	3
G	9	9
H	10	1
I	6	3
J	11	10
K	7	3
L	12	10
M	13	13

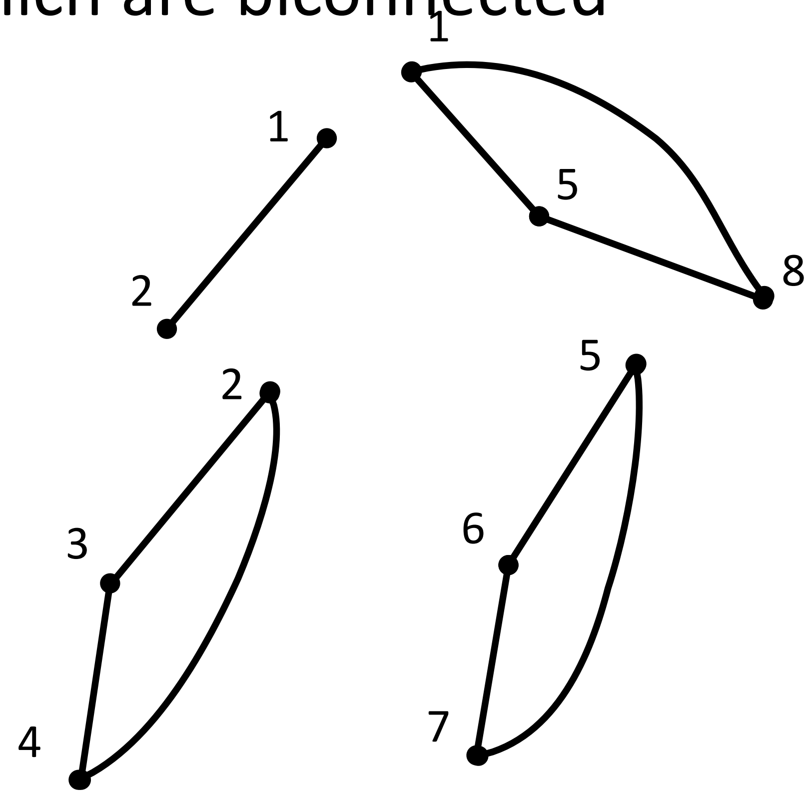
Biconnected Components

- G is biconnected iff either
  1. G is a single edge, or
  2. for each triple of vertices (u, v, w)
     $\exists$ a path p from u to v where w $\notin$ p
     equivalently, $\exists$ two disjoint paths for all edges

Example: Biconnected Components

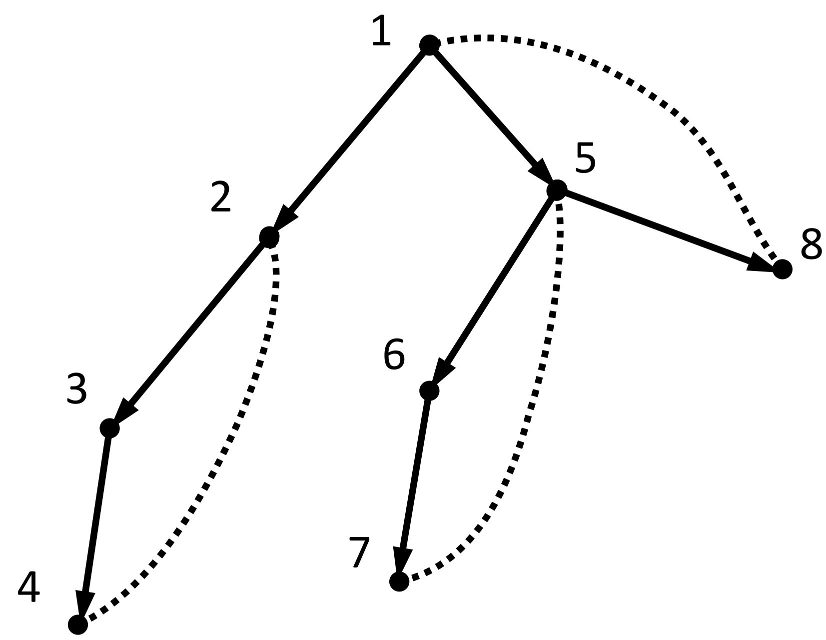
- Maximal subgraphs of G which are biconnected





Biconnected Components Meet at Articulation Points

- The intersection of two biconnected components consists of at most one vertex called an Articulation Point
- Example: 1, 2, 5 are articulation points
- Removing the articulation point disconnects the graph
    - Removing the *bridge* edge disconnects the graph



Discovery of Biconnected Components via Articulation Points

- If we can find articulation points then we can compute biconnected components
- **Algorithm**
  - During DFS, use auxiliary stack to store visited edges
  - Each time we complete the DFS of a tree child of an articulation point, pop all stacked edges currently in stack
  - These popped off edges form a biconnected component

Characterization of an Articulation Point

- **Theorem**

  a is an articulation point iff either

  - 1. **a** is root with ≥ 2 tree children, or
  - 2. **a** is not root, **a** has a tree child v with low(v) ≥ **a.d**

Proof of Characterization of an Articulation Point

- **Proof**

  The conditions are sufficient since any **a**-avoiding path from v remains in the subtree T_v rooted at v, if v is a child of **a**

- To show the condition for necessary, assume **a** is an articulation point

Characterization of an Articulation Point

- Case 1

  If **a** is a root and is an articulation point, then **a** must have ≥ 2 tree edges to two distinct biconnected components

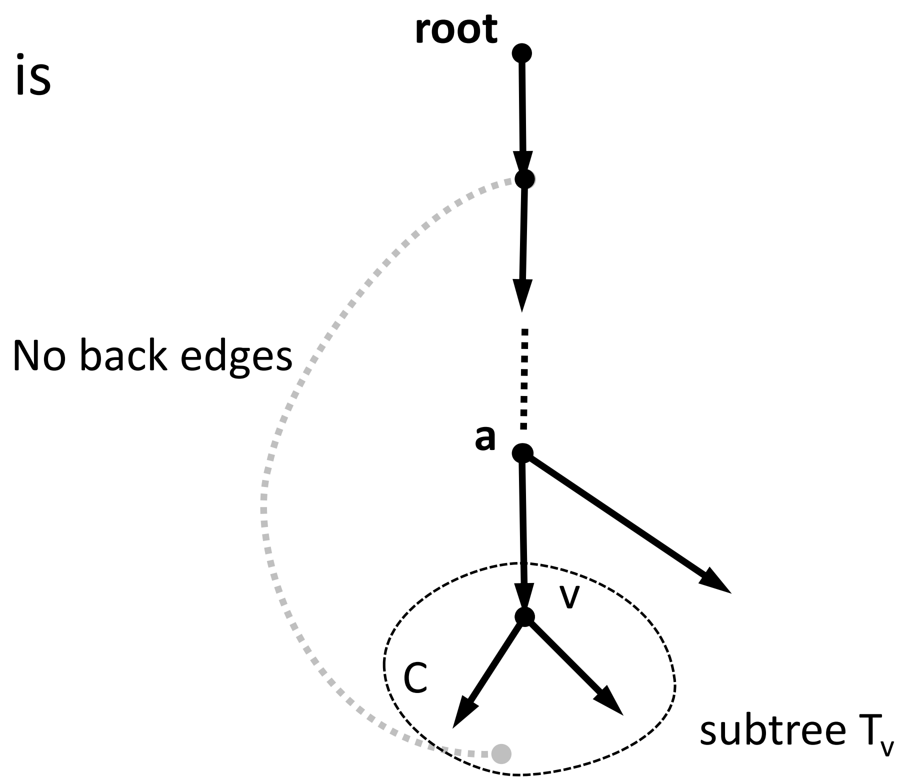
- Case 2

  If **a** is not root, consider graph **G**–{**a**} which must have a connected component C consisting of only descendants of **a**, and with no back edge from C to an ancestor of v. Hence **a** has a tree child v in C and low (v) $\geq$ **a.d**

Proof of Characterization of an Articulation Point (cont.)

- Case 2

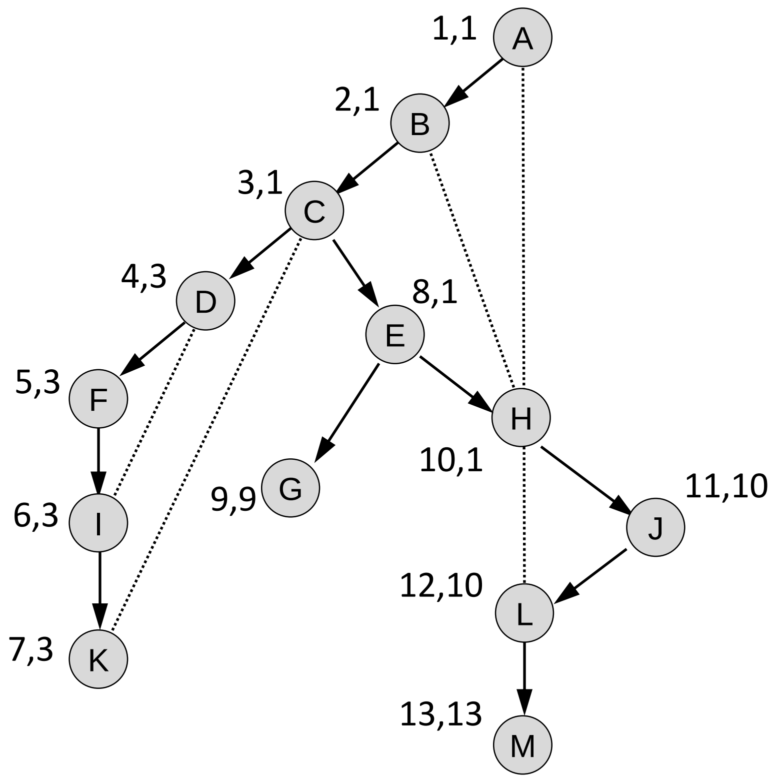
  Articulation point a is not the root



Example: Articulation Points

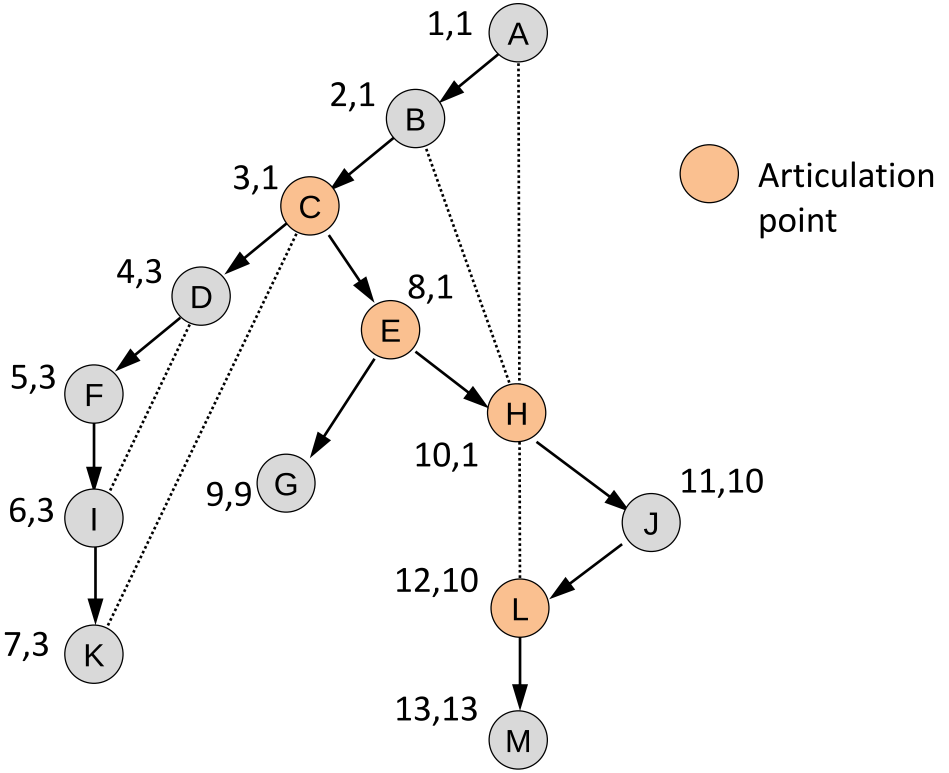
Vertex	dfs#	low
A	1	1
B	2	1
C	3	1
D	4	3
E	8	1
F	5	3
G	9	9
H	10	1*
I	6	3
J	11	10
K	7	3*
L	12	10*
M	13	13

* back edge directly



Example: Articulation Points (cont.)

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



Computing Biconnected Components

- **Theorem**

  The biconnected components of G = (V,E) can be computed in time O(|V|+|E|)
- Proof idea
  - Use characterization of biconnected components via articulation points
  - Identify these articulation points dynamically during DFS
  - Use a secondary stack to store the edges of the biconnected components as they are visited
  - When an articulation point is discovered, pop the edges of this stack off to output a biconnected component