Recall: DFS to find 2-connected components

This graph is connected but removing one vertex \( b \) or \( e \) disconnects it.

\( v \) is a cut vertex if removing \( v \) makes \( G \) disconnected. Cut vertices are bad in networks.

DFS from \( e \)

Characterizing cut vertices:

**Claim** The root is a cut vertex iff it has \( > 1 \) child.

**Lemma** A non-root \( v \) is a cut vertex iff \( v \) has a subtree \( T \) with no non-tree edge going to a proper ancestor of \( v \).

**Proof** \( \Leftarrow \) removing \( v \) separates \( T \) from rest of graph.

\( \Rightarrow \) since removing \( v \) disconnects \( G \), some subtree must get disconnected
Recall: DFS to find 2-connected components

This graph is connected but removing one vertex $b$ or $e$ disconnects it.

$v$ is a cut vertex if removing $v$ makes $G$ disconnected. Cut vertices are bad in networks.

DFS from $e$

Characterizing cut vertices:

Claim The root is a cut vertex iff it has $> 1$ child.

Lemma A non-root $v$ is a cut vertex iff $v$ has a subtree $T$ with no non-tree edge going to a proper ancestor of $v$.

Proof $\iff$ removing $v$ separates $T$ from rest of graph.
$\Rightarrow$ since removing $v$ disconnects $G$, some subtree must get disconnected
Making the lemma into an algorithm

Define: \( \text{low}(u) = \min \{ d(w) : x \text{ a descendant of } u \text{ and } (x, w) \text{ an edge} \} \)

**Convention:** \( u \) is a descendant of \( u \)

\( \text{low}(u) \) = how high in tree we can get to from \( u \) by going down (0 or more) and then up 1 edge

Note: it does not hurt to look at all edges, not just non-tree edges

Fact: non-root \( v \) is a cut vertex iff \( v \) has a child \( u \) with \( \text{low}(u) \geq d(v) \)

We can compute low recursively

\[
\text{low}(u) = \min \left\{ \min \{ d(w) : (u, w) \in E \} , \min \{ \text{low}(x) : x \text{ a child of } u \} \right\}
\]

(1)

Algorithm to compute all cut vertices

- Enhance DFS code to compute low, OR
- Run DFS to compute discover times \( d(\cdot) \).
  Then, for every vertex \( u \) in finish time order use (1) to compute \( \text{low}(u) \).

For every non-root \( v \): if \( v \) has a child \( u \) with \( \text{low}(u) \geq d(v) \) then \( v \) is a cut vertex.

Also handle the root.
Depth First Search on Directed Graphs

**DFS(v)**

- mark(v) := discovered
- d(v) := time;  time := time + 1
- for u ∈ AdjacencyList(v) do
  - if u is undiscovered then
    - DFS(u);  (v, u) is a tree edge
  - else
    - # label back, forward, cross edges
      - if u not finished then
        - (v, u) is a back edge
      - elif d(u) > d(v) then
        - (v, u) is a forward edge
      - else
        - # d(u) < d(v)
        - (v, u) is a cross edge
    - fi
  - fi
- od
- mark(v) := finished
- f(v) := time;  time := time + 1

DFS takes $O(n + m)$

Note: result depends on vertex ordering.
Applications of DFS

(1) Detecting cycles in directed graphs.

**Lemma** A directed graph has a (directed) cycle iff DFS has a back edge.

**Proof**

\[ \iff \]

\[ \text{back edge gives directed cycle} \]

Suppose there is a directed cycle. Let \( v_1 \) be first vertex discovered in DFS. Number vertices of cycle \( v_1 \cdots v_k \).

**Claim** \((v_k, v_1)\) is a back edge.

**Proof** Because we must discover & explore all \( v_i \) before we finish \( v_1 \), when we test edge \((v_k, v_1)\) we label it a back edge.
Applications of DFS

(2) Topological sort of directed acyclic graph \((\text{acyclic} \equiv \text{no directed cycle})\)

Edge \((a, b)\) means \(a\) must come before \(b\) (e.g., job scheduling).

Find a linear order of vertices satisfying all edges (possible iff no directed cycle).

Example:

\[
\text{topological sort: } b \; c \; a \; d \; \text{or } c \; d \; b \; a \; \text{or } \ldots
\]

One solution: Find vertex \(v\) with no in-edge. Remove \(v\) and repeat.

Solution using DFS: \(O(n + m)\)

use reverse of finish order.

Example

(first example

without back edges)

\[
\begin{array}{c}
\text{finish order} \\
\text{reverse finish order } s, \; w, \; z, \; r, \; x, \; y, \; v, \; u
\end{array}
\]

This is a topological order.

Proof that this works.

Claim For every directed edge \((u, v)\), \(\text{finish}(u) > \text{finish}(v)\)

\[\text{case 1 } u \text{ discovered before } v. \text{ Then because of edge } (u, v), v \text{ is discovered and finished before } u \text{ is finished.}\]

\[\text{case 2 } v \text{ discovered before } u. \text{ Because } G \text{ has no directed cycle, we can't reach } u \text{ in } \text{DFS}(v). \text{ So } v \text{ finished before } u \text{ is discovered and finished.}\]
Applications of DFS

(3) Finding strongly connected components in a directed graph.

Strongly connected \( \equiv \) for all vertices \( u, v \) there is a path \( u \rightarrow v \)

Easy to test if \( G \) is strongly connected because we don’t need to test all pairs \( u, v \).

Here’s how: Let \( s \) be a vertex

Claim \( G \) is strongly connected iff for all vertices \( v \), there is a path \( s \rightarrow v \)

and a path \( v \rightarrow s \).

Proof \( \Rightarrow \) clear

\( \Leftarrow \) to get from \( u \rightarrow v \): \( u \rightarrow s \rightarrow v \)

To test if there’s a path \( s \rightarrow v \ \forall v \) — do \( \text{DFS}(s) \).

How can we test if there’s a path \( v \rightarrow s \ \forall v \)? Reverse edge directions and do \( \text{DFS}(s) \).

Neat!

More generally, the structure of a digraph is

Contracting strongly connected components gives an acyclic graph (think about why).