DFS

Depth first search — go as far as you can, when there’s nothing new to discover, retrace your steps to find something new.

**DFS tree**

Order in which vertices are discovered: $a, b, e, f, g, d, c$

Order of finishing: $f, g, e, c, d, b, a$

Use a stack to store vertices that have been discovered but must still be explored.

As a recursive program (stack is implicit):

$$\text{DFS}(v)$$

- or Explore($v$)

1. mark($v$) $\leftarrow$ discovered
2. for each neighbour $u$ of $v$ do
   - if $u$ is undiscovered then
     - mark($u$) $\leftarrow$ discovered
     - DFS($u$), parent($u$) $\leftarrow$ $v$; $(u,v)$ is a tree edge
   - else $(u,v)$ is a non-tree edge, unless $u = \text{parent}(v)$
3. mark($v$) $\leftarrow$ finished

DFS

Mark all vertices undiscovered

for all vertices $v$

- this handles multiple components

  if $v$ is undiscovered

    - start new tree rooted at $v$

    - DFS($v$)
As with BFS, we should store more info as we do this:
Store parent pointers, distinguish tree edges and non-tree edges (see changes above)

Run-time: \(O(n+m)\) (same argument as for BFS)

DFS gives rich structure:
- partition into separate trees
- Edge classification
- Vertex order: order of discovery, order of finishing

**Lemma** DFS from root vertex \(v_0\) discovers all vertices connected to \(v_0\)

**Proof** Suppose there is a path \(v_0 \to v_1 \cdots \to v_f\)
Look at last vertex discovered \(v_f\)
Then we explore all neighbours of \(v_i\) including \(v_{i+1}\)
(more formal by induction)

**EX.** Enhance code to number the connected components and record the component of each vertex

**Lemma** All non-tree edges join ancestor and descendant.
\[ u \text{ is an ancestor of } v \]
\[ u \text{ is a descendant of } v \]

Cannot have edge \((x, y)\).

Suppose \(x\) discovered first.

Then in \(\text{DFS}(x)\) we examine neighbour \(y\).

So \(y\) is discovered before \(x\) finishes and \(y\) appears in subtree of \(x\).

Enhancing \(\text{DFS} \) to compute discover \& finish times:

\[ \text{DFS}(u) \]

\[
\text{mark}(u) \leftarrow \text{discovered}
\]
\[
\text{discover}(u) \leftarrow \text{time} ; \; \text{time} \leftarrow \text{time} + 1
\]

for each neighbour \(u\) of \(v\) do

\[
\text{if } u \text{ is undiscovered then}
\]
\[
\text{DFS}(u)
\]

end

\[
\text{finish}(u) \leftarrow \text{time} ; \; \text{time} \leftarrow \text{time} + 1
\]

Let \(d(u) = \text{discover}(u)\), \(f(u) = \text{finish}(u)\).

Discover \& finish times form a parenthesis system.

If \(d(u) < d(v)\) then
\[
\begin{bmatrix}
\begin{bmatrix}
d(u) & d(u) & f(u) & f(u)
\end{bmatrix}
\end{bmatrix}
\]

or
\[
\begin{bmatrix}
\begin{bmatrix}
d(u) & f(u) & d(u) & f(u)
\end{bmatrix}
\end{bmatrix}
\]

because interval \(d(u), f(u)\) is time on stack.
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.

Biconnected components

DFS from \( e \)

DFS from a shown before

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

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

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

\( \Rightarrow \) since removing \( v \) disconnects \( G \), some subtree must get disconnected
Define

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

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

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

We can compute \( \text{low} \) recursively

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

Algorithm to compute all cut vertices:

- can enhance DFS code to compute \( \text{low} \)
- or:
  
  run DFS to compute discover times, \( d(\cdot) \) for every vertex \( u \) in finish time order

  for every \( u \)
  
  if \( u \) has a child \( (u) \) with \( \text{low}(u) \geq d(v) \)
    
    then \( u \) is a cut node.

Also handle the root.