**DFS**

Bold search - go as far as you can, when there's nothing new to discover, retrace your steps to find something new.

**DFS tree**

Good for solving a maze!

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

As a recursive program (stack is implicit):

\[
\text{DFS}(v) \quad \text{-- or Explore}(v).
\]

\[
\text{mark}(v) \leftarrow \text{discovered}
\]

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

\[
\text{if } u \text{ is undiscovered then}
\]

\[
\text{DFS}(u); \quad \text{parent}(u) \leftarrow v; \quad (u,v) \text{ is a tree edge}
\]

\[
\text{else } (u,v) \text{ is a non-tree edge.}
\]

\[
\text{end}
\]

\[
\text{mark}(v) \leftarrow \text{finished}
\]

**DFS**

Mark all vertices undiscovered

for all vertices \( u \text{ -- this handles multiple components} \)

\[
\text{if } u \text{ is undiscovered} \quad \text{-- start new tree rooted at } v
\]

\[
\text{DFS}(v).
\]
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)

Runtime: $O(n+m)$ (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 $v_0, v_1, \ldots, v_f$
Look at last vertex discovered $v_i$

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.
$v$ is an ancestor of $u$

$u$ 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 DFS to compute discover & finish times

$\text{DFS}(v)$

\begin{align*}
\text{mark}(v) & \leftarrow \text{discovered} \\
\text{discover}(v) & \leftarrow \text{time} ; \text{time} \leftarrow \text{time} + 1 \\
\text{for each neighbour } u \text{ of } v \text{ do} \\
& \quad \text{if } u \text{ is undiscovered then} \\
& \quad \quad \text{DFS}(u) \\
& \quad \text{end} \\
\text{finish}(v) & \leftarrow \text{time} ; \text{time} \leftarrow \text{time} + 1
\end{align*}

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

Discover & finish times form a parenthesis system.

If $d(v) < d(u)$ then

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

because interval $[d(v), f(v)]$ 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 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 \( u \) with \( \text{low}(u) \geq d(u) \)

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

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

\[ \text{min} \left\{ \text{low}(x) : x \text{ a child of } u \right\} \]

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

\[ \forall u \]
if \( u \) has a child \( u \) with \( \text{low}(u) \geq d(u) \)
then \( u \) is a cut node.

Also handle the root.