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.

**Biconnected components**

\[
\begin{align*}
B & \quad b & a & e & f \\
c & d & b & a & e & f \\
d & g & b & a & e & f
\end{align*}
\]

**DFS from e**

\[
\begin{align*}
& \quad e \\
b & \quad f \\
g & \quad f \\
d & \quad f
\end{align*}
\]
characterizing cut vertices:

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. \( \Leftarrow \) 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
\[ \text{low}(u) = \min \{ \text{d}(w) : x = u \text{ or } x = \text{a descendant of } u \text{ and } (x,w) \in \text{E} \} \]

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

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 \( \text{low}(u) \geq \text{d}(u) \)

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

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

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

Algorithm to compute all cut vertices

- can enhance DFS code to compute \( \text{low} \)
- OR:

run DFS to compute discover times, \( \text{d}(-) \) for every vertex \( u \) in finish time order

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

for every \( u \)

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

Also handle the root.
Depth First Search on Directed Graphs

order of exploration

forward edge - from vertex to descendant

cross edges

backward edge - from vertex to ancestor

DFS(v)

mark (v) \leftarrow discovered

\begin{align*}
d(v) & \leftarrow \text{time}; \quad \text{time} \leftarrow \text{time} + 1 \quad \text{\# discover time} \\
\text{for each vertex } u \text{ in Adjacency List } (v) \text{ do} & \\
\text{if } u \text{ is undiscovered then} & \\
\quad \text{DFS}(u); \quad (v, u) \text{ is a tree edge} & \\
\text{else} & \\
\quad \text{mark } (v) \leftarrow \text{finished} & \\
\quad f(v) & \leftarrow \text{time}; \quad \text{time} \leftarrow \text{time} + 1 \quad \text{\# finish time}
\end{align*}

\begin{itemize}
\item label back, forward, cross edges
\item if \( u \) not finished then \((v, u)\) is back edge
\item else if \( d(u) > d(v) \) then \((v, u)\) is forward edge
\item else if \( d(u) < d(v) \) then \((v, u)\) is cross edge
\end{itemize}

DFS takes \( O(n + m) \)

Note that 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

\[ \iff \]

back edge gives directed cycle

\[ \Rightarrow \]

Suppose there is a directed cycle. Let \( v_1 \) be first vertex discovered in DFS.

Number vertices of cycle \( v_1, \ldots, v_k \)

Claim \( (v_k, v_i) \) is a back edge.

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

2. Topological sort of directed acyclic graph

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

\[
\begin{array}{cccc}
  & a & b \\
  \downarrow & \downarrow & \downarrow \\
  c & \ & d \\
\end{array}
\]

Topological sort: \(b\ c\ a\ d\) or \(c\ d\ b\ a\) or...

One solution: find vertex \(v\) with no in-edge.

Remove \(v\) and repeat.

EX: Do this in \(O(n + m)\) time.

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

Use reverse of finish order.

Example

(first ex. minus back edge)

Finish order

Reverse finish order: \(s, w, z, r, x, y, v, u\)

This is a topological order.
Proof that this works.

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

If case 1: \(u\) discovered before \(v\)

Then because of edge \((u, v)\), \(v\) is discovered and finished before \(u\) is finished.

Case 2: \(v\) discovered before \(u\)

Because \(G\) has no directed cycle, we can't reach \(u\) in DFS\((v)\). So \(v\) finished before \(u\) is discovered and finished.
3. Finding strongly connected components in a directed graph:

   strongly connected = for all vertices \( u, v \)
   there is a path \( u \rightarrow v \) and a path \( v \rightarrow u \).

   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 \( v \rightarrow s \).

   **pf.** \( \Rightarrow \) clear

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

   To test if there's a path \( s \rightarrow v \) \& \( v \rightarrow s \) — do DFS(\( s \)).

   How can we test if there's a path \( u \rightarrow s \) \& \( v \rightarrow u \)?

   Reverse edge directions and do DFS(\( s \)). Neat!

   More generally, structure of a digraph is

   ![Diagram of strongly connected components]

   Contracting strongly conn. components gives acyclic graph

   (think about the reason)
This was NOT covered in class and will not be tested. It is in the notes only in case you are interested.

How to find strongly connected components.

Idea: Vertices 1...n

- Run DFS (vertex ordering resolves what vertex comes next)
- Let finish order be $f_1, f_2, ..., f_n$
- $G^R = G$ with all edges reversed
- Run DFS on $G^R$ with vertex order $f_n, f_{n-1}, ...$, f

Lemma: Trees in 2nd DFS are exactly the strongly connected components.

For pseudo code, see CLRS.

Run time is $O(n+m)$

Example

Finish order

If first DFS started at 2 2 DFS trees, 2 components.

Last finished vertex still on left. So same 2 trees.
Idea of why this works:

\[ C_1 \xrightarrow{} C_2 \]

If DFS of \( G \) starts at \( v \) in \( C_1 \), then it reaches \( C_1 \cup C_2 \) and \( v \) is last finished.

If DFS of \( G \) starts at \( v \) in \( C_2 \), then it reaches all of \( C_2 \) and new DFS tree starts at \( C_1 \). So last finished is in \( C_1 \).

In \( G^R \):

\[ C_1 \xleftarrow{} C_2 \]

DFS starts in \( C_1 \) — so \( C_2 \) needs a new tree