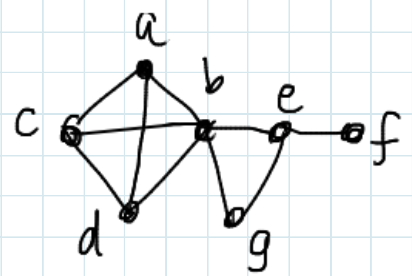
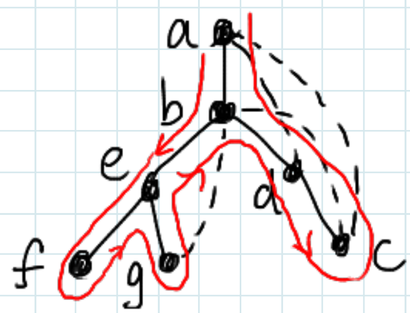
DFS



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!



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):

DFS(v) -- or Explore(v).

mark(v) ← discovered

for each neighbour u of v do

if u is undiscovered then

DFS(u); parent(u) ← v; (u,v) is a tree edge

else (u,v) is a non-tree edge. unless u = parent(v)

end

mark(v) ← 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)

Runtime: $O(n+m)$ (same argument as for BFS)

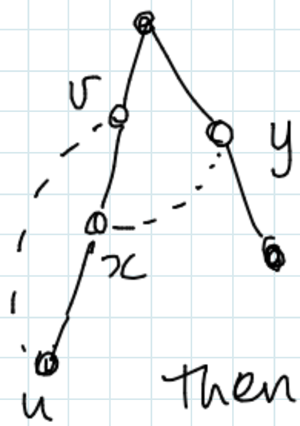
DFS gives rich structure:
- partition into separate trees
- Edge classification
- Vertex order: order of discovery, order of finishing

Lemma DFS from root vertex v_0 discovers all vertices connected to v_0

Proof Suppose there is a path $v_0\ v_1 \cdots\ v_f$
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 <u>ancestor</u> of u

u is a <u>descendant</u> of v

Cannot have edge (x, y):

Suppose x discovered first

Then in 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

```
DFS(v)
    mark(v) ← discovered
    discover(v) ← time; time ← time + 1
    for each neighbour u of v do
        if u is undiscovered then
            DFS(u)
    end
    finish(v) ← time; time ← time + 1
```

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

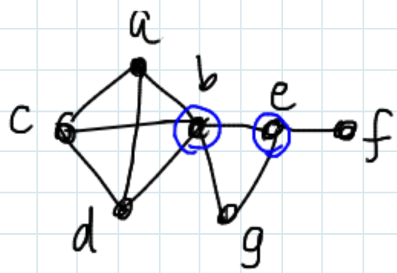
Discover & finish times form a parenthesis system.

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

$\underset{d(v)}{[}\quad \underset{d(u)}{[}\quad \underset{f(u)}{]}\quad \underset{f(v)}{]}$ or $\underset{d(v)}{[}\quad \underset{f(v)}{]}\quad \underset{d(u)}{[}\quad \underset{f(u)}{]}$

because interval $d(v), f(v)$ is time on stack.

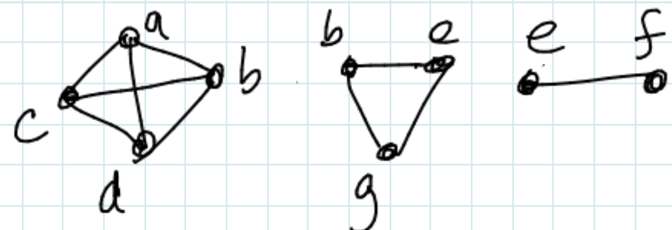
DFS to find 2-connected components



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

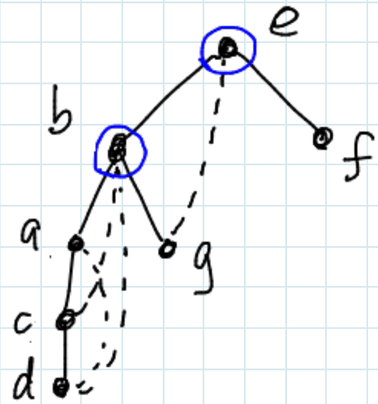
v is a <u>cut vertex</u> if removing v makes G disconnected. Cut vertices are bad in networks.

Biconnected components



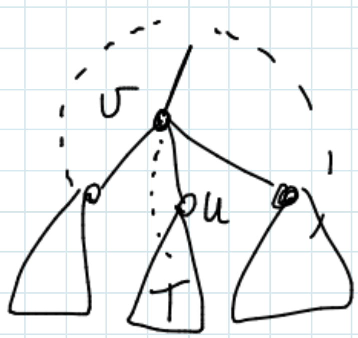
DFS from a shown before

DFS from e



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 ⇐ removing v separates T from rest of graph.

⇒ since removing v disconnects G, some subtree must get disconnected

Define

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

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

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

We can compute low recursively

$$(*)\quad \text{low}(u) = \min \begin{cases} \min\{ d(w) : (u,w) \in E \} \\ \min\{ \text{low}(x) : x \text{ a child of } u \} \end{cases}$$

<u>Algorithm to compute all cut vertices</u>

- can enhance DFS code to compute low
- OR:

run DFS to compute discover times, $d(\cdot)$

for every vertex v in finish time order

$(*)$

for every v

  if v has a child (u) with $\text{low}(u) \geq d(v)$

    then v is a cut node.

Also handle the root.