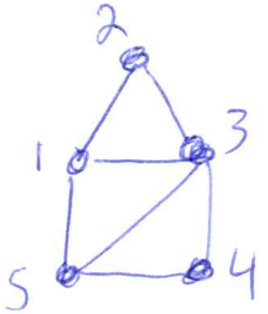
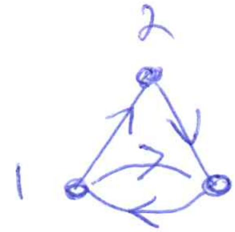
Graphs



$V = \{1,2,3,4,5\}$

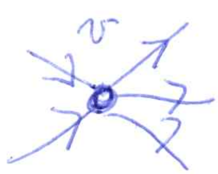
$E = \{(1,2),(1,3),(1,5),\dots\}$



Directed Graph

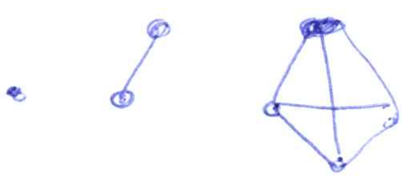
$V = \{1,2,3\}$

$E = \{(1,3),(1,2),(2,3),(3,1)\}$



indegree(v) = 2

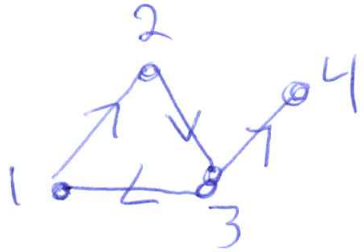
outdegree(v) = 3



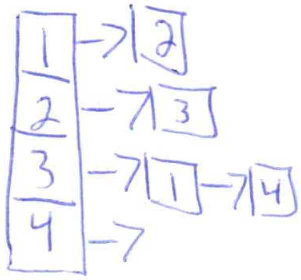
3 connected components

Adjacency Matrix

- if $(i,j) \in E \Rightarrow A[i,j] = A[j,i] = 1$
- undirected graph



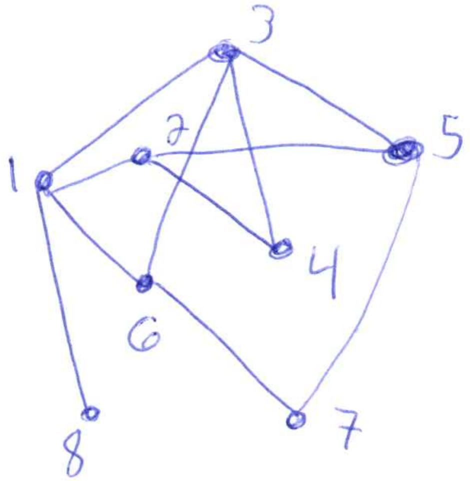
$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 1 & 0 \\
3 & 1 & 0 & 0 & 1 \\
4 & 0 & 0 & 0 & 0
\end{array}
$$



	Adj Matrix	vs Adj Lists
Space:	$O(n^2)$	$O(n+m)$
$(u,v)\in E$?	$O(1)$ $A[u,v]=1$?	$O(1+\deg(u))$ or $O(1+\min\{d(u), d(v)\})$
List v's neighbours	$\Theta(n)$	$\Theta(1+\deg(v))$ *
List all edges	$\Theta(n^2)$	$\Theta(n+m)$ *

* The algs we focus on will typically require * so we'll use adj lists.

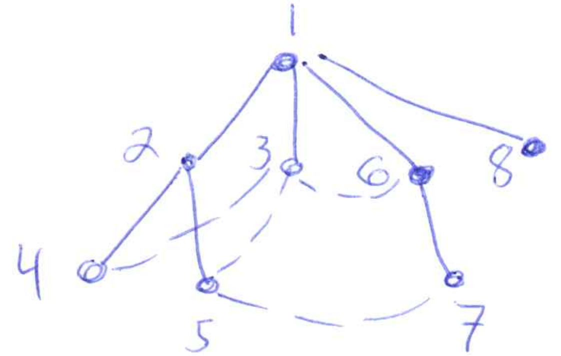
BFS



Order of discovery

$1 \; \underbrace{2, 3, 6, 8}_{\text{1's neighbours}}, \; \underbrace{4, 5}_{\text{2's}}, \; \underbrace{7}_{\text{6's}}$

BFS tree



level 1

level 2

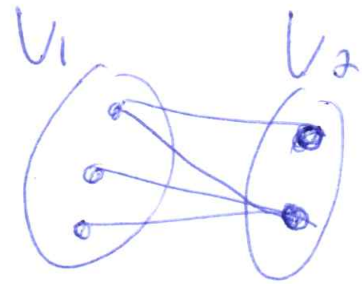
Use a queue to store vertices that have been discovered but must still be explored.

---

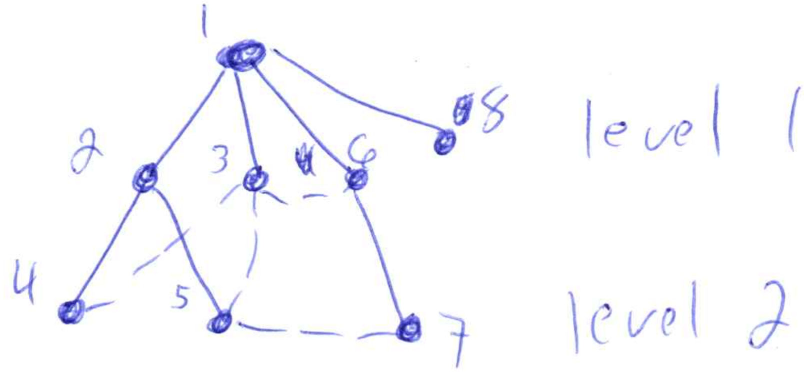
parent pointers

---

bipartite



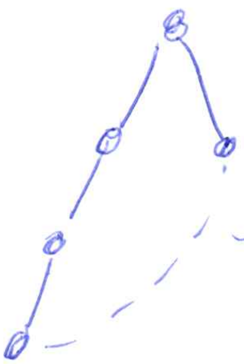
Cycle



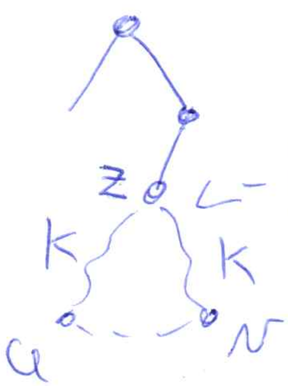
Discover 1 $\underbrace{2\ 3\ 6\ 8}$ $\underbrace{4\ 5}_{2's}$

from 3 check 4, 5, 6 already discovered => cycles.

can we have



~ passes 2 levels? No!

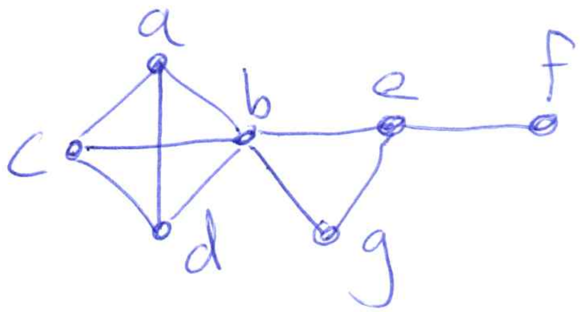


<- least common ancestor

~ on same level

cycle 2k+1 ~ odd.

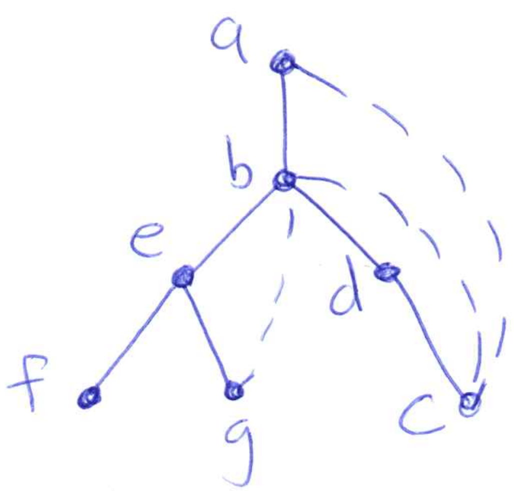
DFS



Adj Lists

a: b, c, d
b: e, g, d, c, a
c: a, b, d
d: a, b, c
e: b, f, g
f: e
g: b, e

DFS tree

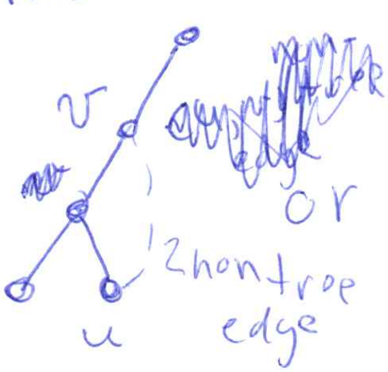


Order of Discovery: a b e f g d c

Finishing: f g e c d b a

- f, g are explored & finished before e is finished.

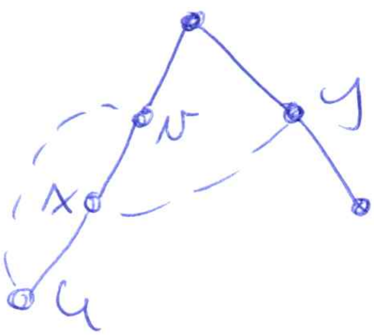
u has already been discovered



encounter u from v's adj list

u = parent(v)

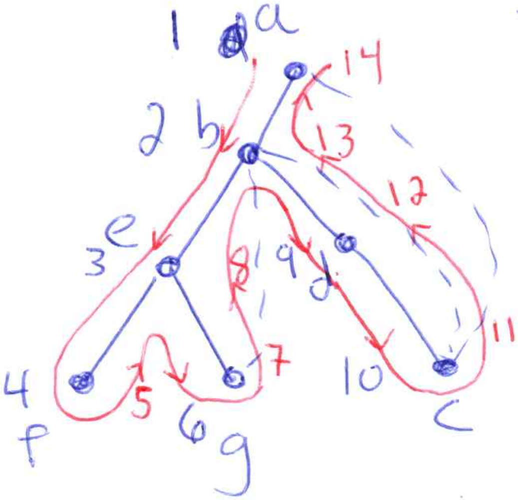
Lemma: All non-tree edges join an ancestor and a descentant.



• v is an ancestor of u & u is a descendant of v ✓

$(x, y) \notin E$

Compute discover & finish times



times

Discovery:	a	b	e	f	g	d	c
	1	2	3	4	6	9	10
Finish:	f	g	e	c	d	b	a
	5	7	8	11	12	13	14

$d()$ - discovery(), $f()$ - finish()

Discovery & finish times form a parenthesis system

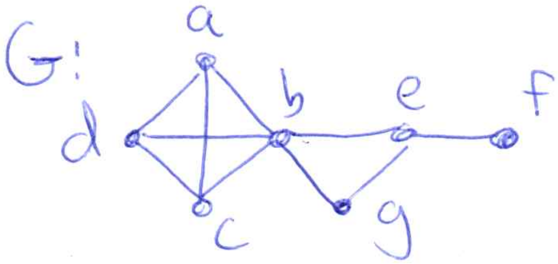
If $d(v) < d(u)$ then

$\underset{d(v)}{[} \; \underset{d(u)}{[} \; \underset{f(u)}{]} \; \underset{f(v)}{]}$ OR $\underset{d(v)}{[} \; \underset{f(v)}{]} \; \underset{d(u)}{[} \; \underset{f(u)}{]}$

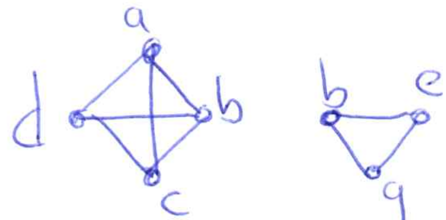
nested   disjoint

DFS to find 2-connected components

G:



Biconnected components of G

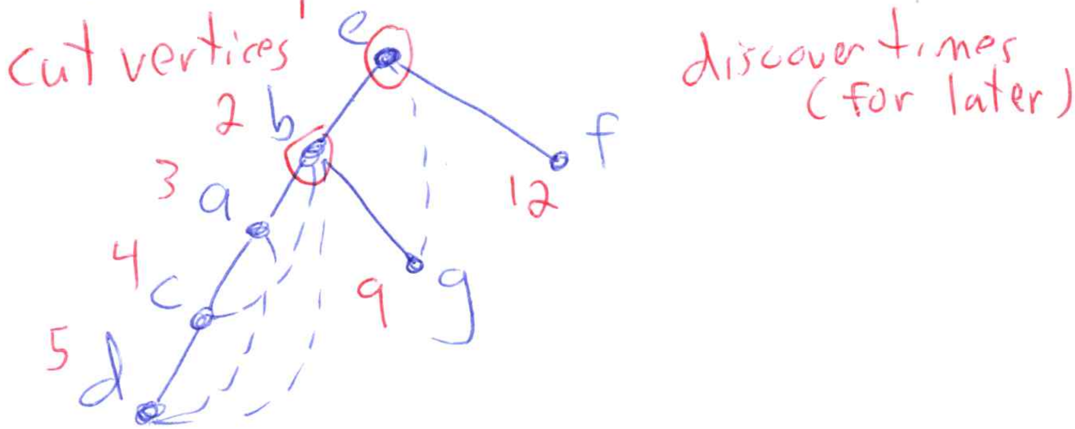


Look at how these components are connected.

- vertices in common

G is connected but removing either b or e would disconnect it.

DFS from e



Cut Vertex Lemma

- if we cut v, T becomes disconnected.

eg $v = b$ above

consider discovery times

Algorithm to find Cut Vertices

• check if discovery(w) < discovery(v)

Let u be the root of a subtree T of v

x a descendant of u

(x, w) is a non-tree edge

*could check all edges

Define: $low(u) = \min\{ d(w) : \uparrow \}$, use *

A non-root vertex v is a cut vertex

<u>iff</u> v has a child u with $low(u) \geq d(v)$

Compute low() recursively

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

Alg to compute all cut vertices

- Enhance DFS to compute low

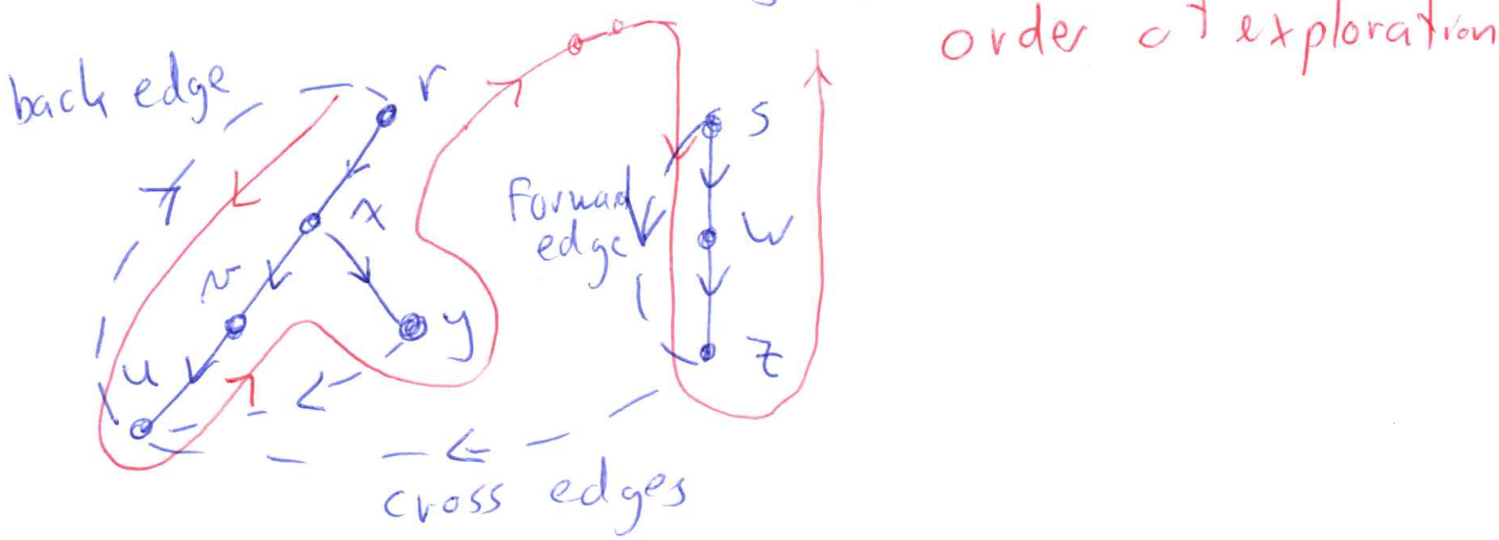
OR
- Run DFS to compute discover times, $d()$

Then, for every vertex ~~w~~ u in finish time order, use (1) to compute $low(u)$

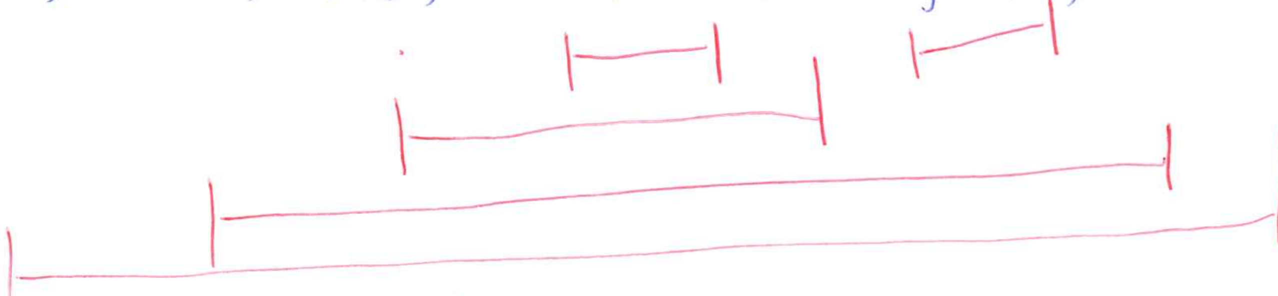
For every non-root v: if v has a child u with $low(u) \geq d(v)$ then v is a cut vertex.

- also handle root.

DFS on directed graphs



d(r) d(x) d(v) d(u) f(u) f(v) d(y) f(y) f(x) f(r) d(s)



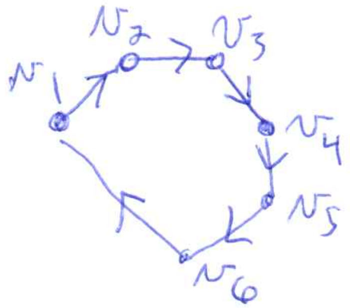
parenthesis system

Detecting cycles in directed graphs

Lemma: A directed graph has a (directed) cycle
<u>iff</u> DFS has a back edge

Proof

$\Leftarrow$  Back edge gives a directed cycle

$\Rightarrow$ 

Suppose there is a directed cycle.

Let v_1 be the first vertex discovered in DFS.

Number vertices of cycle $v_1 \ldots v_K$

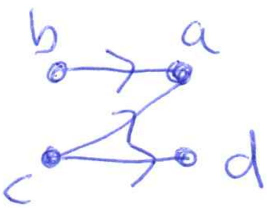
Claim (v_K, v_1) is a back edge.

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
- v_1 is an ancestor of v_K

Topological Sort 

a must be done before b

eg jobscheduling

Ex 

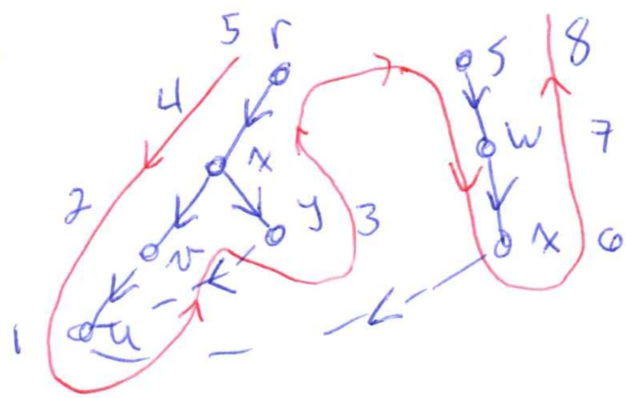
topological sort: b c a d
or c d b a or ...

One solution: find vertex v with no in-edge
Remove v and repeat.

DFS Solution $O(n+m)$

- use reverse finish order

Ex



- no back edge

finish order 1...8

reverse finish order:

s w z r x y v u

This is topological order

Proof

Claim: For every directed edge (u, v),

$\text{finish}(u) > \text{finish}(v)$

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

not possible

- G has no directed cycles, we can't reach u in DFS(v)

So v finished before u is discovered and finished

Finding Strongly connected components in a directed graph

- Don't need to test all pairs u, v

Let s be a vertex

Claim: G is strongly connected <u>iff</u> for all veritices v, there is a path $s \to v$ and a path $v \to s$

Proof $\Rightarrow$ obvious! It really is this time.

$\Leftarrow$ to get from $u \to v$: $u \to s \to v$

To Test for path $s \to v$, $\forall v$, use DFS(s)

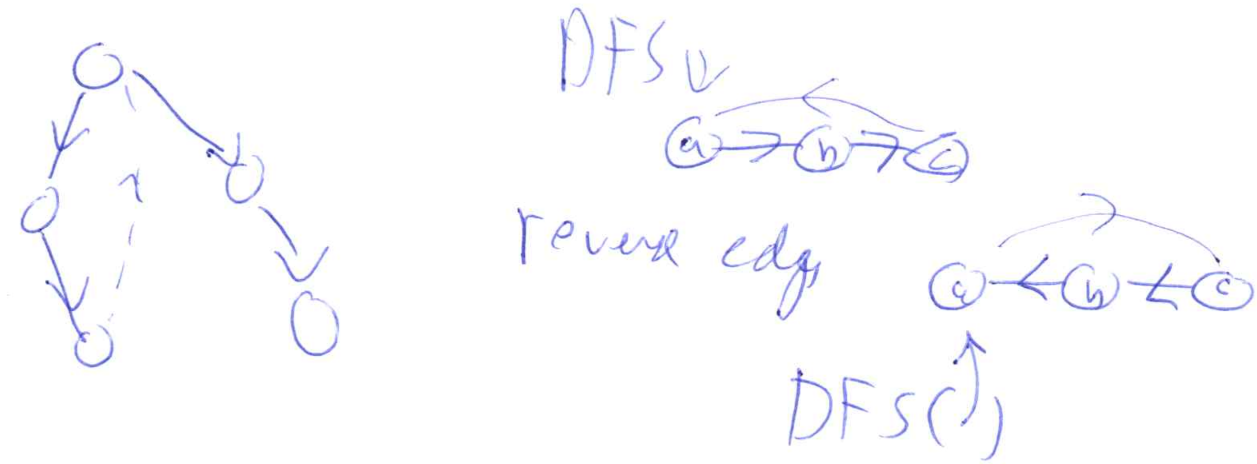
Test for path $v \to s$?

- Reverse edge directions and do DFS(s)

Structure of a digraph

strongly connected components

- Contracting S.C.C. gives an acyclic graph
  - think about it!



History Tarjan 1972

Kosaraju 78/81

Gabow 99

all linear time, all simple but different

Kosaraju: Vertices $1..n$

Run DFS $\Rightarrow$ finish order $f_1, f_2 \dots f_n$

reverse edges of G, call it H

run DFS again with vertex order $f_n \dots f_1$

<u>Lemma</u> Trees in 2nd DFS are exactly the strongly connected components

Runtime $O(n+m)$