Misof Lecture

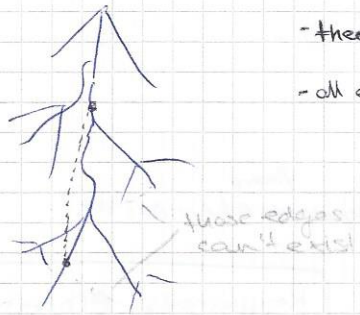
```
DFS(v)
    if visited[v] return
    visited[v] = true;
    FOR EACH (v,w):
        DFS(v);
```

- also works for directed graphs

VISITED[FALSE FOR ALL v in Graph]

Example 1: Number of Components

DFS-Spanning tree:



- there are no vertices between different subtrees.
- all edges are between vertices and their ancestors!

DFS-tree in a directed Graph

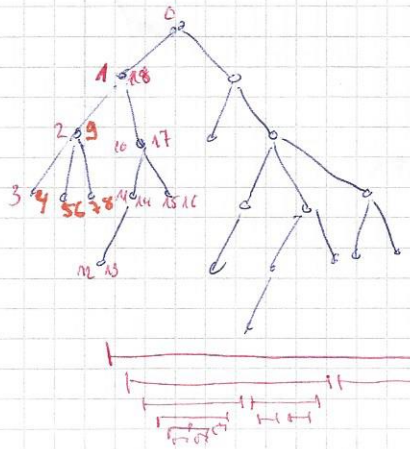
way of constructing a topological order: DFS on directed Graph (only for acyclic Graphs)

~~Example with package manager~~

Circle detection

- visited instead of false/true 3 states: installed, not installed, marked for installation
  circle detected: visit vertic that is marked for installation

DFS on a tree

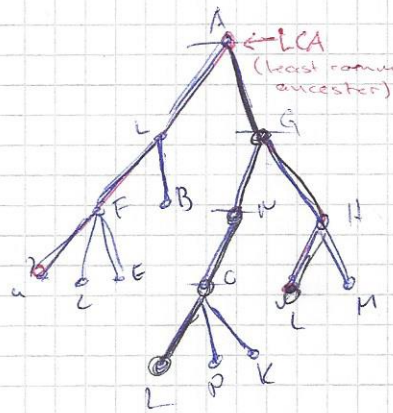


- labeled: "time" of visit (first number"
  : "time of leaving"

usage: it is very easy to tell whether a vertex is an ancestor of a certain other vertic.

- only two possibilities for two edges: either nested or disjoint, no "intersection possible!"

Least common Ancestor



Example. - Change time of Links
- Given two vertices, compute the distance

- length of path (u,w) = ~~long~~ depth(u) - depth(w) - 2·depth[lca(u,w)].

Computing the least common ancestor:

- Version 1: linear time: ask, are we a ancestor of v?
  → going one step ~~beat~~ the time is two slow

- Version 2: We would like to do binary search. But we don't know the path

idea:
ARRAY of parents:
PARENT[ ]

precompute:
UPWARDS[e][v] = if we take 2^e steps upwards from v where are we?

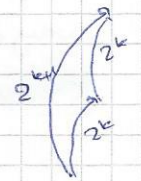
How to compute it:

UPWARDS[0][v] = PARENT[v]

1	1
2	2
3	4
:	
16	2^{16} = ROOT[ ]

$\sim \log_2(n)$ depth

Requirements: $\Theta(n \log n)$ memory
$\Theta(n \log(n))$ time



to compute upwards[k+1][v] = upwards[k][upwards[k][v]]

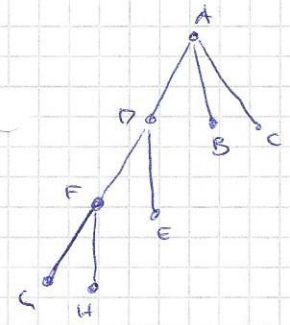
- Version 3: now we truly can do binary search

~~STEP=log~~ STEP = $\lceil \log(n) \rceil$ → at competition we do an actual constant e.g. 70

- each time moving if we are below the lowest common ancester!
(in the end, we are one step below the lowest common ancester).

- Version 4: ("easyier version")

log of what happened throug the DFS:

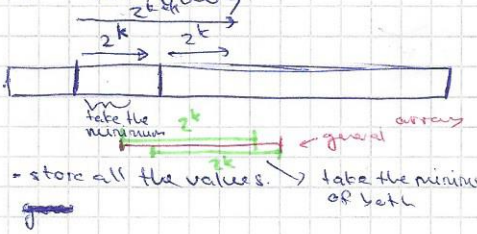


log:	0	1	2	3	4	5	6	7	8	9	10	11	12	13
name:	A	D	F	G	F	H	F	D	E	D	A	B	A	C	A
depth:	0	1	2	3	2	3	2	1	2	1	0	1	0	1	0

Range minimum query
⇒ easy way to do: creating the tree

Version 6:

{ O(n log(n)) for precomputation:
{ O(1) per query:



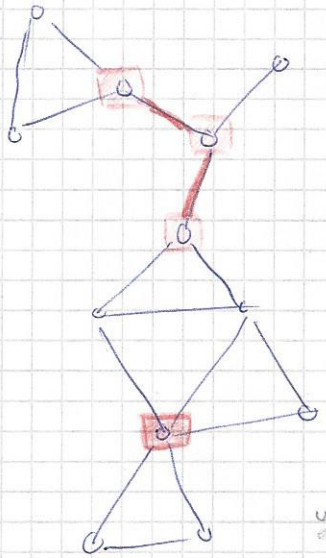
- store all the values. → take the minimum of both

- Misof's favourite solution during a contest.

BRIDGES & ARTICULATION POINTS

Bridge: Edge, if you cut it, the number of components will increase

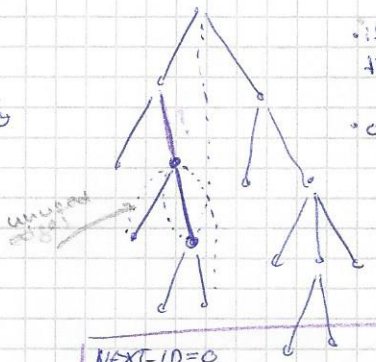
Articulation point/cut vertex: if you remove a vertex, the number of components will increase



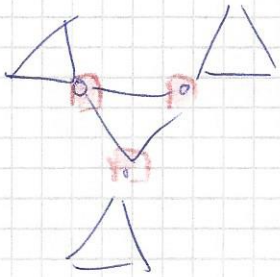
- If you have a bridge ⇒ the endpoints will be cut/vertices (or leaves)
- Reverse idea doesn't work (see example below)

Wish: take a Graph, find all bridges/cut vertices

- we assume our Graph is connected, do the DFS tree



- if all of the unused edges are below ⇒ the vertex is important.
- only edges from the dfs tree can be bridges.



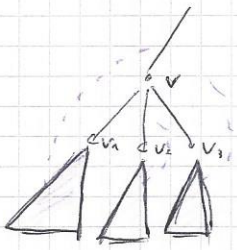
```
NEXT_ID=0
DFS(v, p):          ← parent of v
  ID[v] = NEXT_ID
  ++NEXT_ID
  LOW_LINK[v] = v   ← maximal number
  FOR EACH (v,w) in EDGES:
    IF ID(w) is NONE:
      DFS(w)
      LOWLINK(v) = min(
          (OWLINK(v),
          (OWLINK(w),)      IF w=p CONTINUE;
    ELSE: LOWLINK(v) = min(LOWLINK(v), w)
```

```
NEXT-ID=0
DFS(v, p)
    ID[v] = NEXT-ID;
    ++NEXT-ID;

    LOWLINK[v] = ID[v];

    FOR EACH (v,w) in EDGES:
        IF ID[w] is NONE:
            DFS(w)
            LOWLINK[v] = min(
                    LOWLINK[v],
                    LOWLINK[w])
        IF w=p CONTINUE
        ELSE:
            LOWLINK[v] = min(LOWLINK[v], ID[w])
```

now, if v == LOWLINK[v]
⇒ EDGE from v to its parent is a bridge

- if ID[v] = LOWLINK[v] ⇒ (v,p) is a bridge

Now, if LOWLINK[v_i] ≤ v ⇒ v is a cut-vertex.



if v is no cut-vertice all it's children have to have a high LOWLINK

SPECIAL CASE FOR THE ROOT:

the root is a cut-vertex, if and only if it has only one child in the DFS-tree

Suggestions for further information:

TARJAN'S ALGO FOR SCC (strongly connected components)
KOSARAJU'S