P THINK FAST S

Competitive Programming

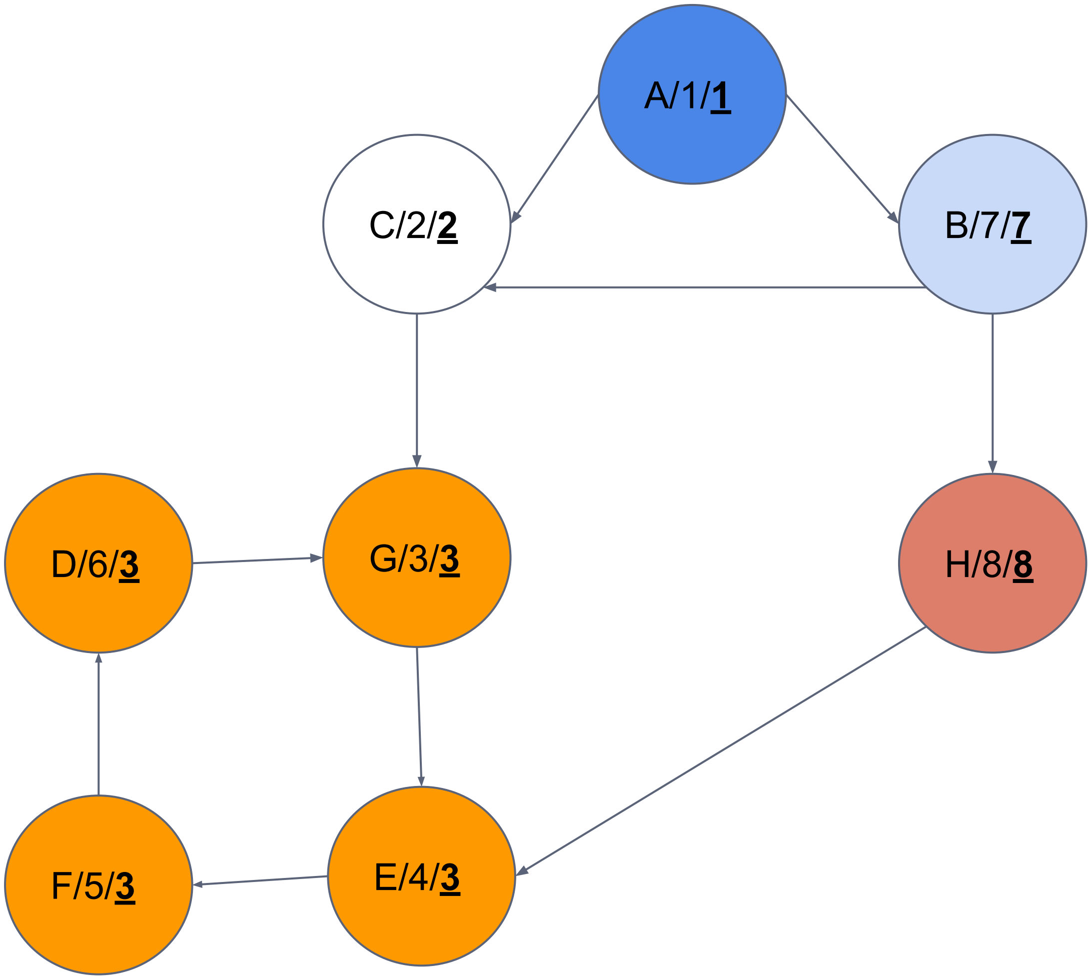
From Problem 2 Solution in O(1)

Graph Theory

Strongly CC using Tarjan - 2

Mostafa Saad Ibrahim
PhD Student @ Simon Fraser University

Recall DFS# and LowestLink



LowestLink # by DFS

- If child node is unvisited?
  - Let ur child find recursively its LowLink #
  - Then simply minimize parent LowLink# with child one
- If child node is visited?
  - Either in current stack (ancestor = cycle) $\Rightarrow$ minimize
  - It is not in stack $\Rightarrow$ Old search tree $\Rightarrow$ Ignore

Normal DFS with DFS#

```
vector< vector<int> > adjList;
vector<int> dfn;
int ndfn;

void tarjan(int node)
{
    dfn[node] = ndfn++;

    rep(i, adjList[node])
    {
        int ch = adjList[node][i];

        if (dfn[ch] == -1)  // Not visited
            tarjan(ch);
    }
}
```

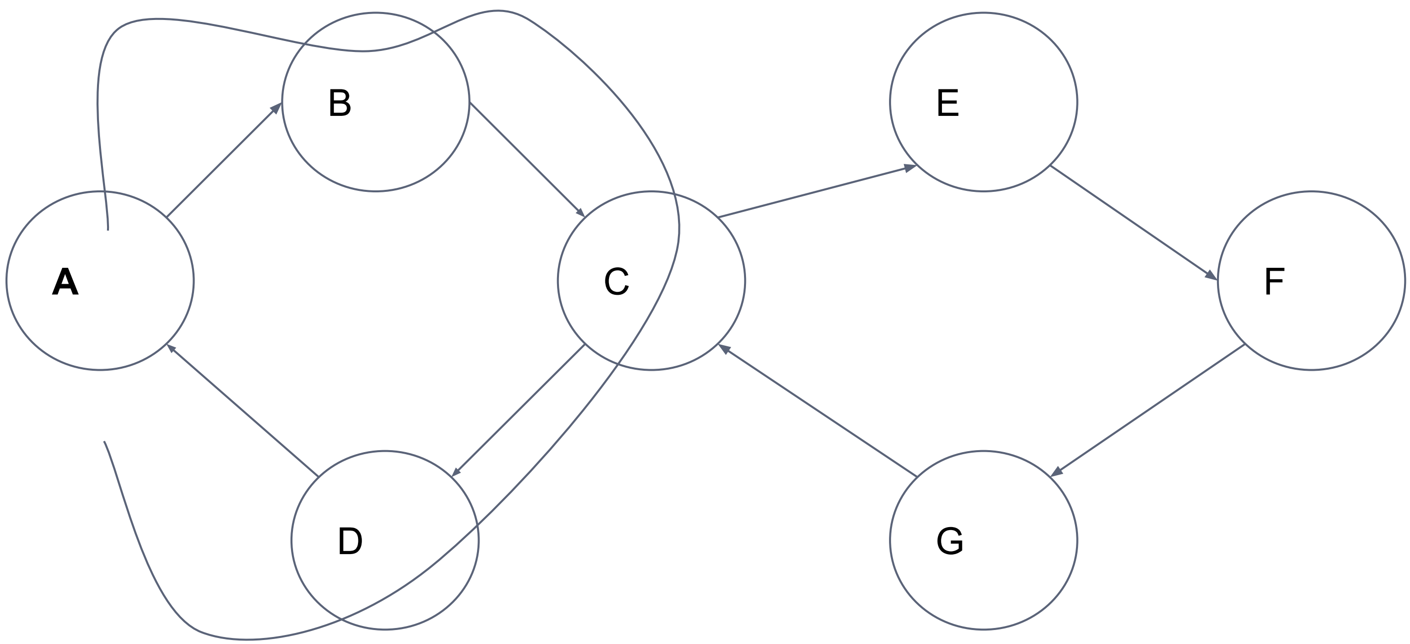
DFS + LowLink

```
vector< vector<int> > adjList;
vector<int> inStack, lowLink, dfn;
stack<int> stk;
int ndfn;

void tarjan(int node) {
    lowLink[node] = dfn[node] = ndfn++;

    rep(i, adjList[node]) {
        int ch = adjList[node][i];
        if (dfn[ch] == -1) {
            tarjan(ch);
            // minimize ancestors of my child
            lowLink[node] = min(lowLink[node], lowLink[ch]);
        } else if (inStack[ch]) // visited + instack = ancestor in cycle
            lowLink[node] = min(lowLink[node], lowLink[ch]);
    }
}
```

Same effect: lowLink[node] = min(lowLink[node], **dfn**[ch]);

Actually, the 2nd way works for **SCC, Bridges and Articulation**

Let’s trace: A, B, C, D, E, F, G

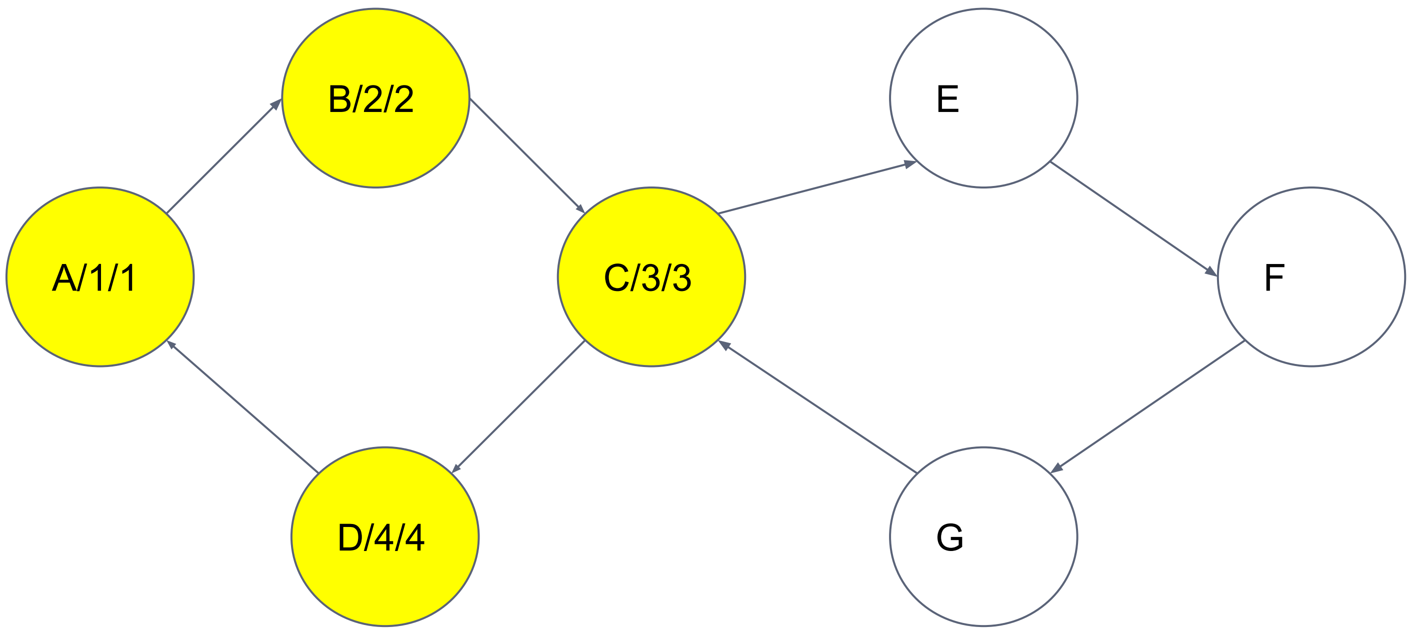


Recall **initialization**:
lowLink[node] = dfn[node] = ndfn++

Recall **unvisited** child case:

```
if (dfn[ch] == -1) {
   tarjan(ch);
   lowLink[node] = min(lowLink[node], lowLink[ch]);
```

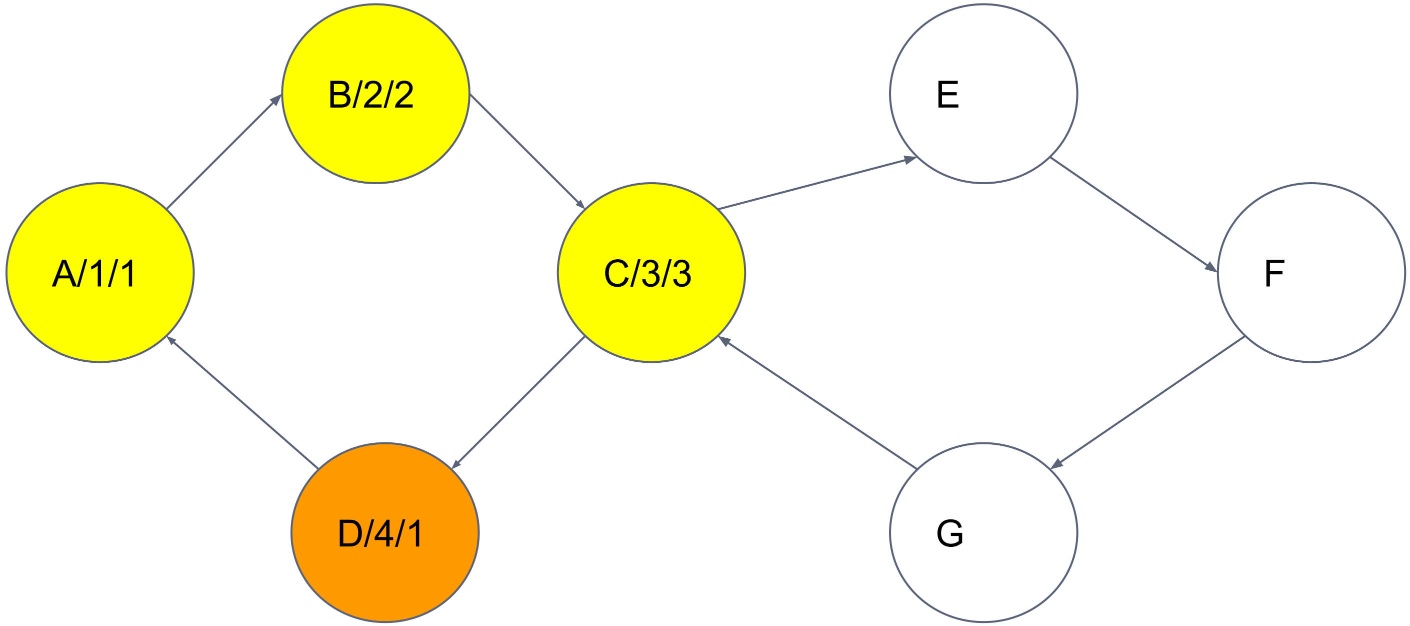
Let’s trace: A, B, C, D, E, F, G



D ⇒ finds A visited = **ancestor**
lowLink[D] = min(lowLink[D], lowLink[A]) = min(4, 1) = 1;

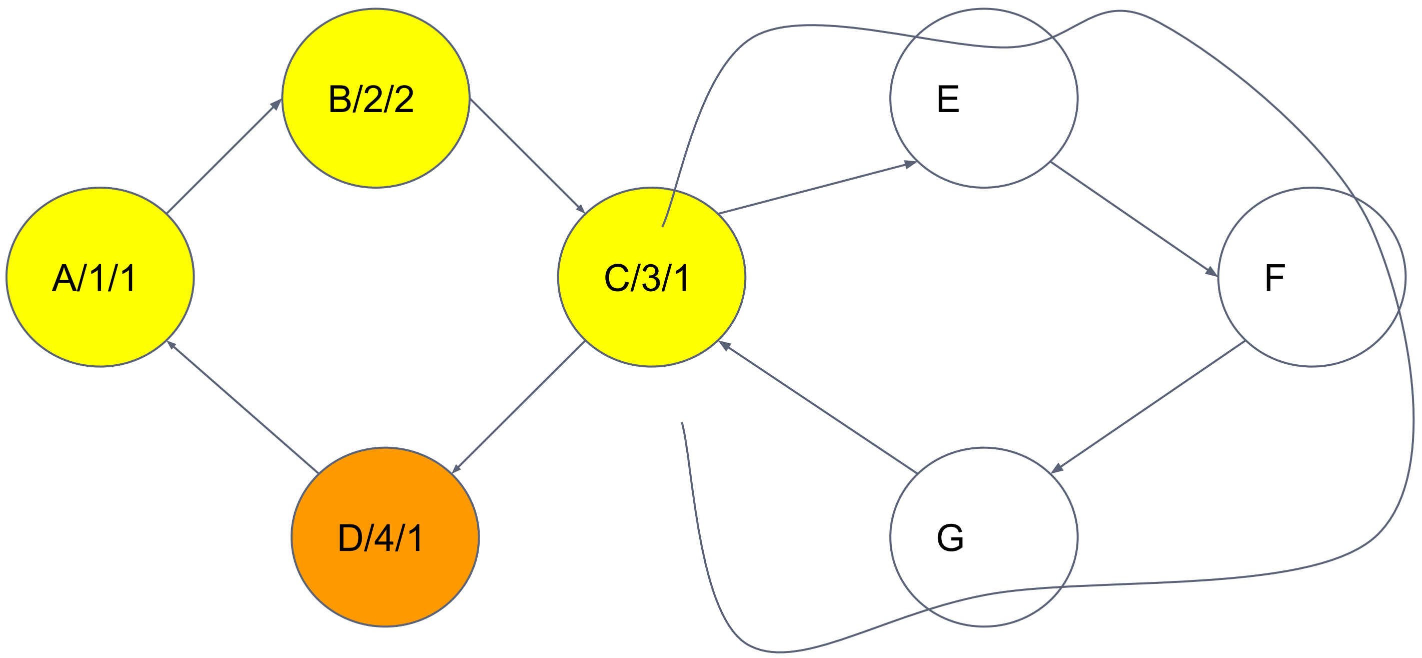
D has no more childs...back to parent (C)

Let’s trace: A, B, C, D, E, F, G



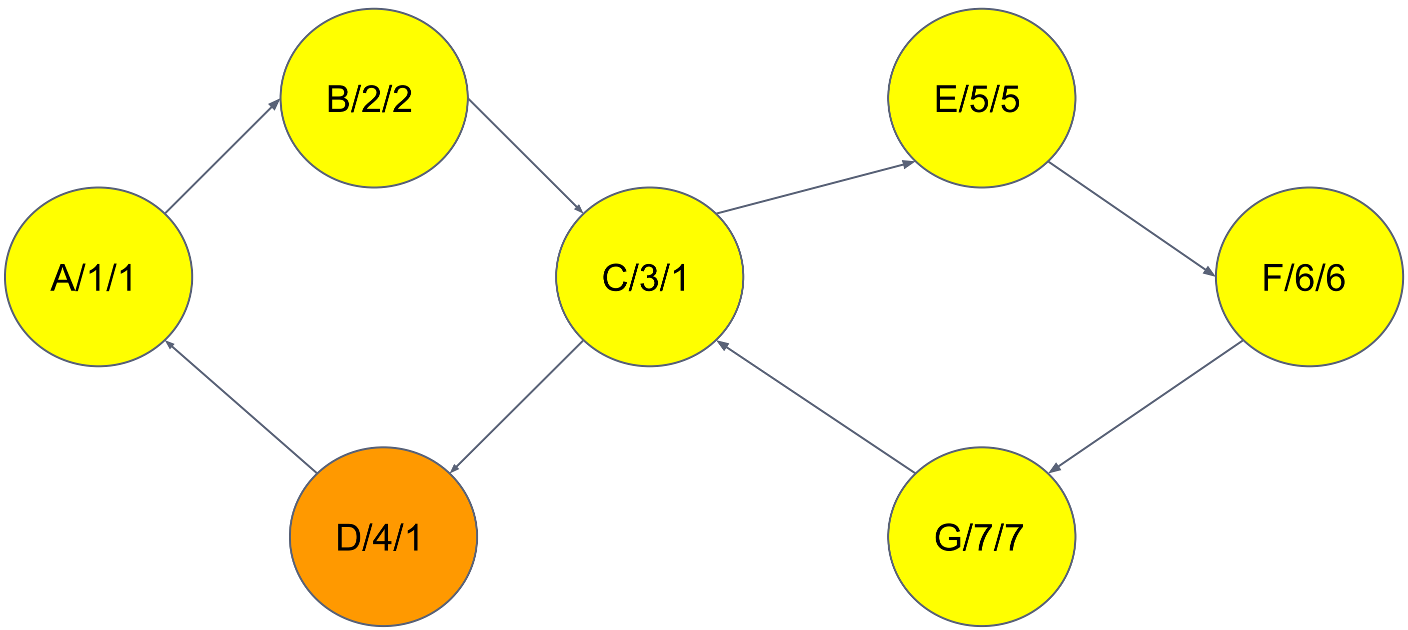
C minimizes on its child D
lowLink[C] = min(lowLink[C], lowLink[D]) = min(3, 1) = 1;

Let’s trace: A, B, C, D, E, F, G



C continues its search to E, F, G….G finds C visited!

Let's trace: A, B, C, D, E, F, G

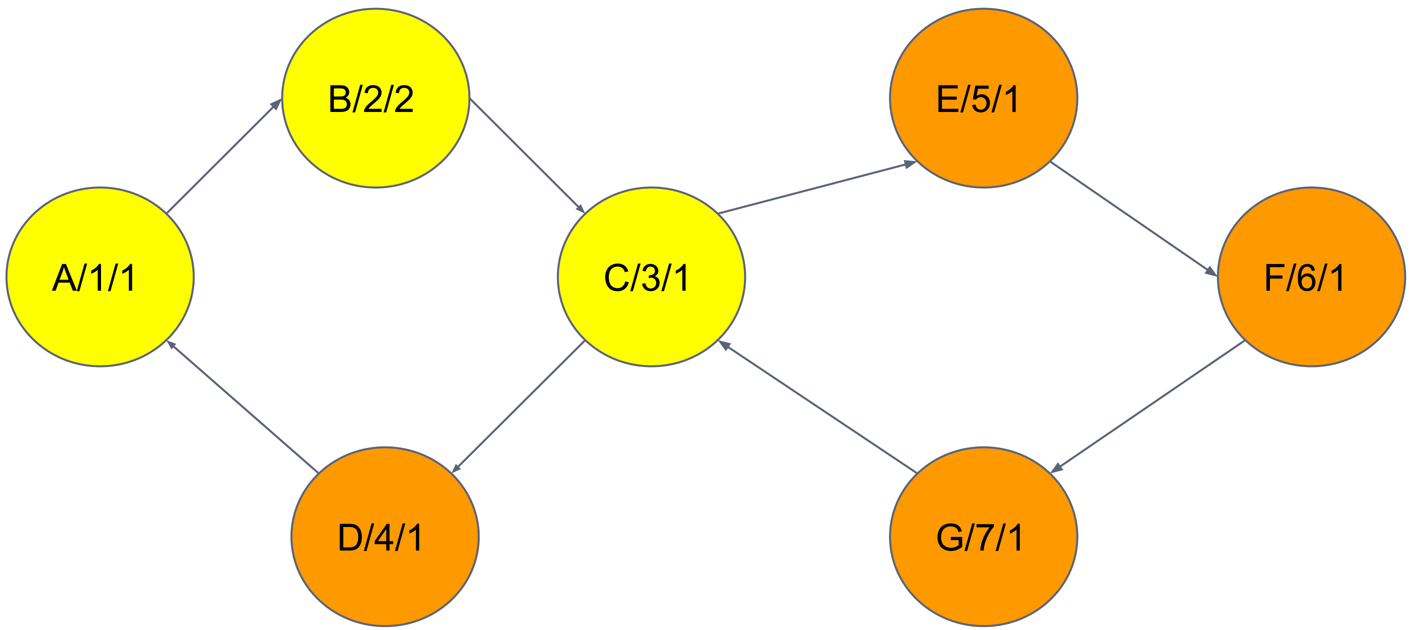


G ⇒ finds C visited = **ancestor**
lowLink[G] = min(lowLink[G], lowLink[C]) = min(7, 1) = 1;

G has no more childs...back to parent (F)

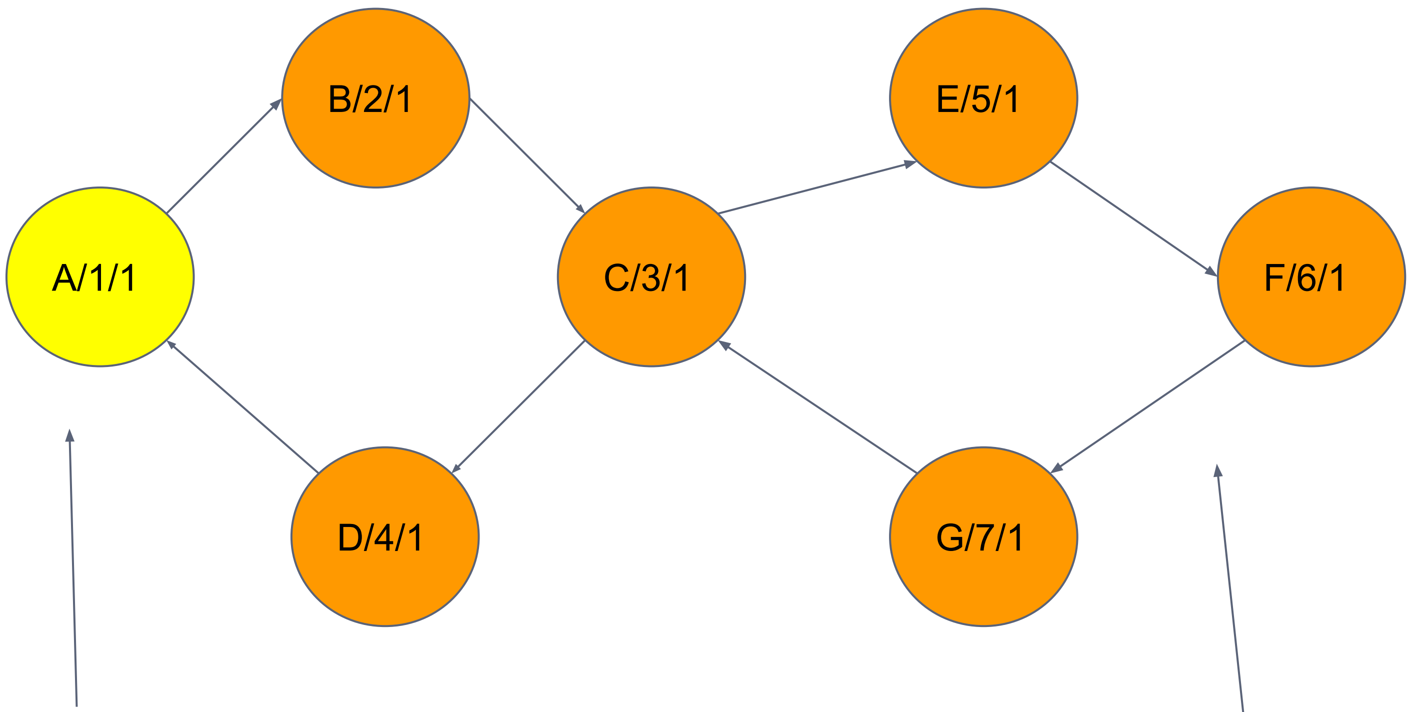
Same for F, E

Let’s trace: A, B, C, D, E, F, G



C minimize on E => no effect
B minimizes on A and ⇒ low[B] = 1

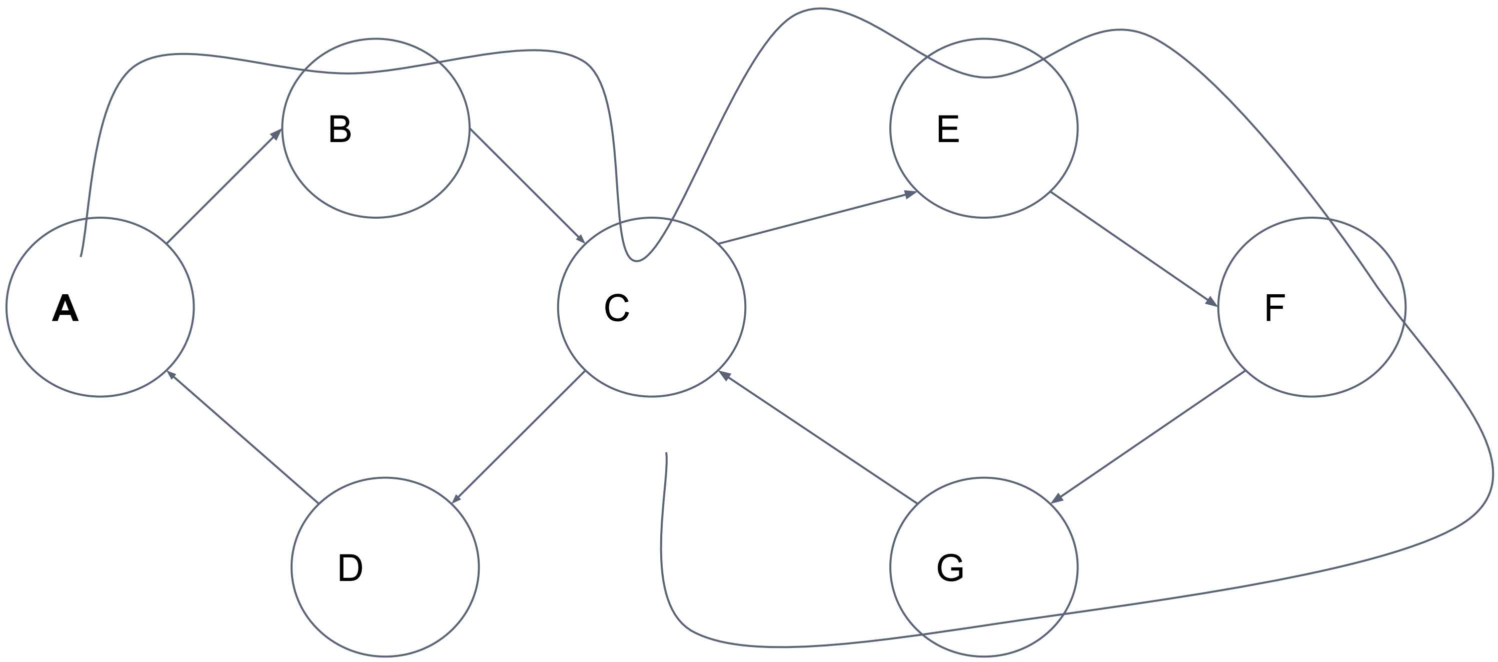
Let’s trace: A, B, C, D, E, F, G



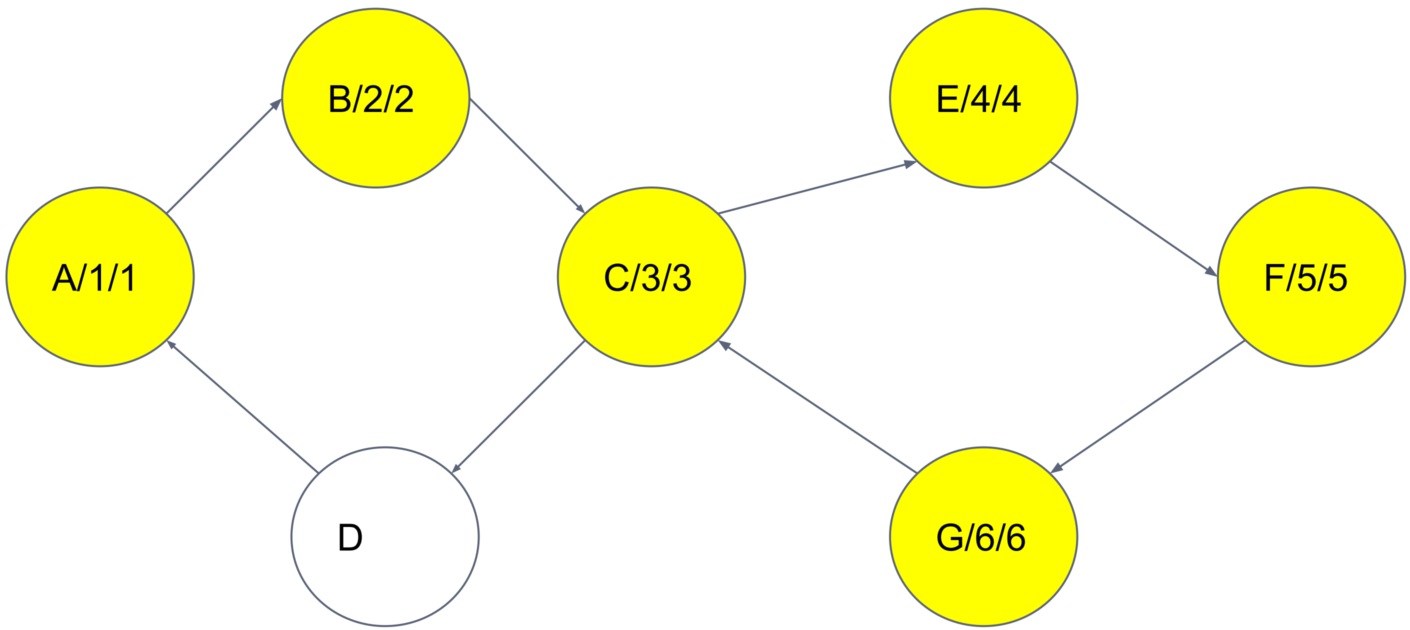
A is the only node with dfn # = LowLink # = **Head of SCC**

All **other** nodes has:
low # < dfs #
Do we guarantee their low # = low scc head ? No

Let’s trace: A, B, C, <u>E, F, G, D</u>



Let’s trace: A, B, C, E, F, G, D

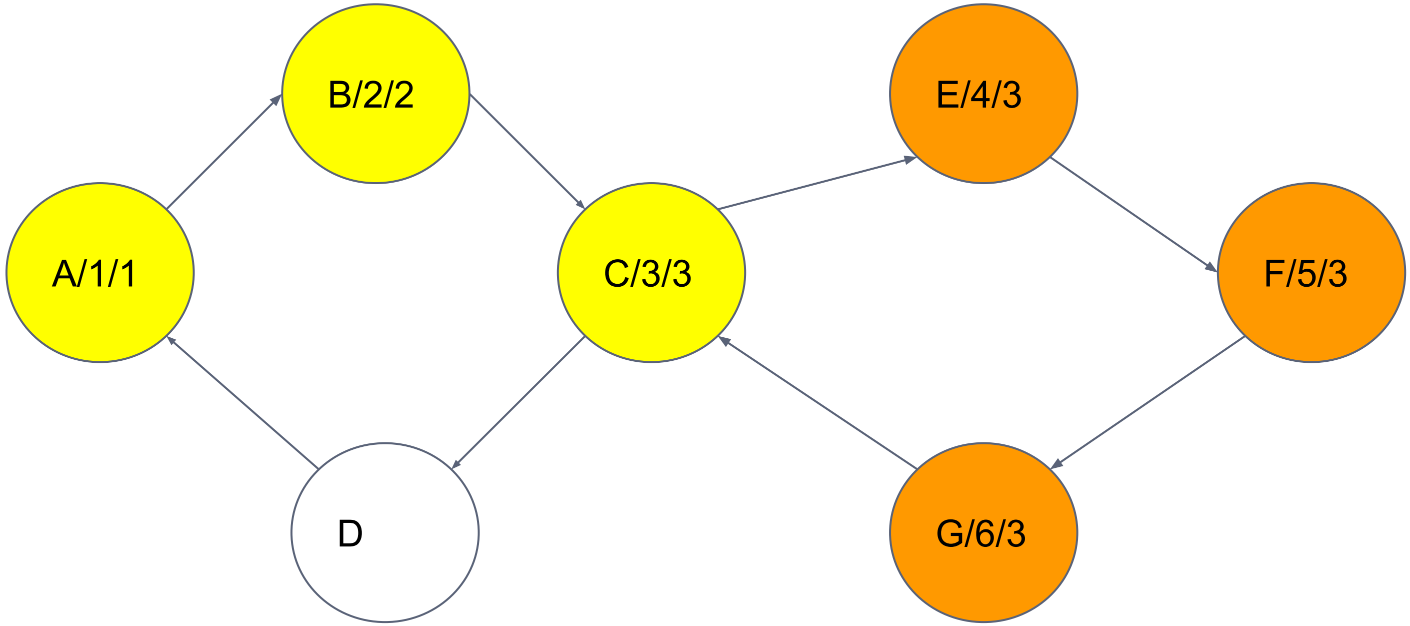


DFS so far: A, B, C, E, F, G

G ⇒ finds C visited = **ancestor**
lowLink[G] = min(lowLink[G], lowLink[C]) = min (6, 3) = 3;

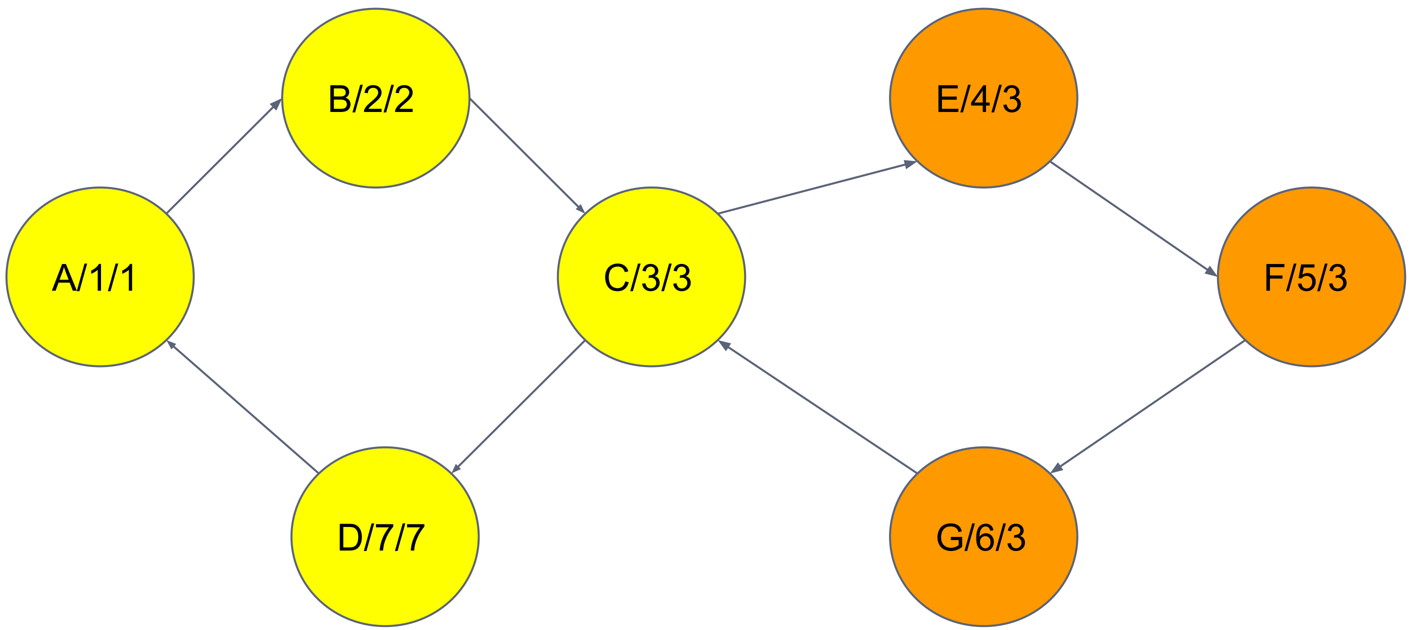
G has no more children...back to parent (F)
Same for F, E

Let’s trace: A, B, C, E, F, G, D



E return to C
C go its unvisited child D

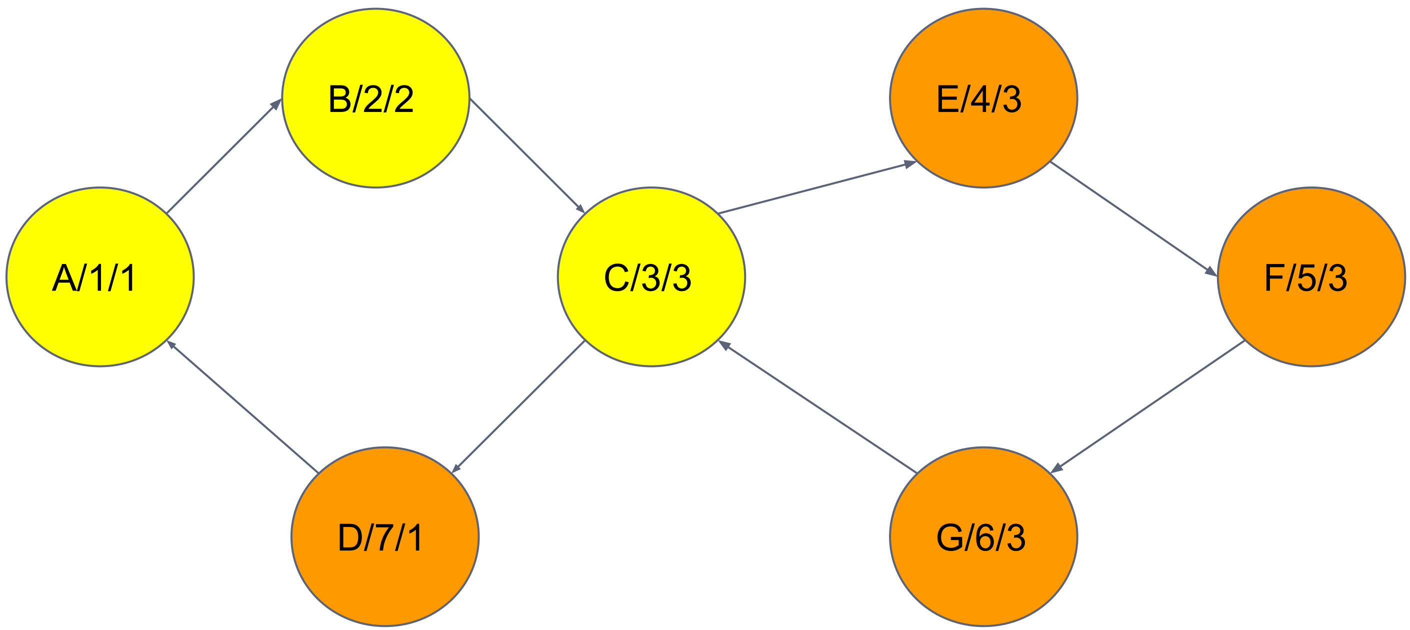
Let’s trace: A, B, C, <u>E, F, G, D</u>



D ⇒ finds A visited = **ancestor**
lowLink[D] = min(lowLink[D], lowLink[A]) = min(7, 1) = 1;

D has no more children...back to parent (C)

Let's trace: A, B, C, E, F, G, D

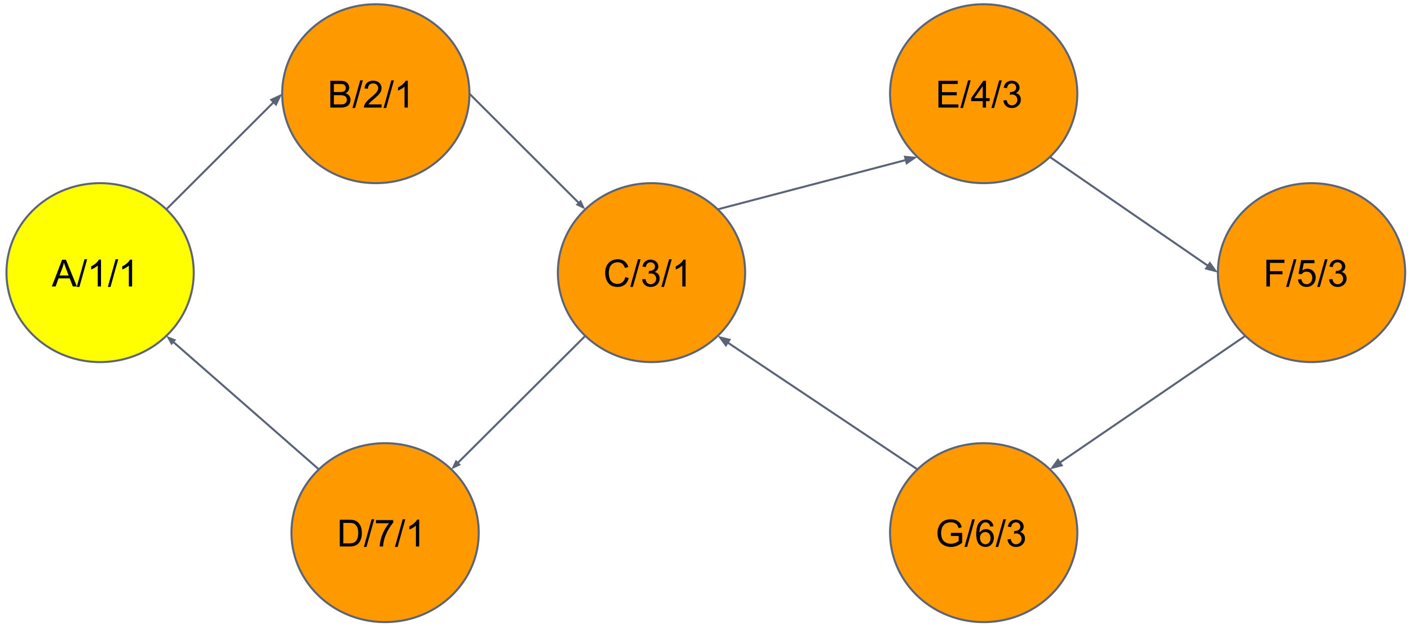


C minimize on D low[C] = min(3, 1) = 1

C has no more unvisited child...return to B

B minimizes on C

Let’s trace: A, B, C, <u>E, F, G, D</u>

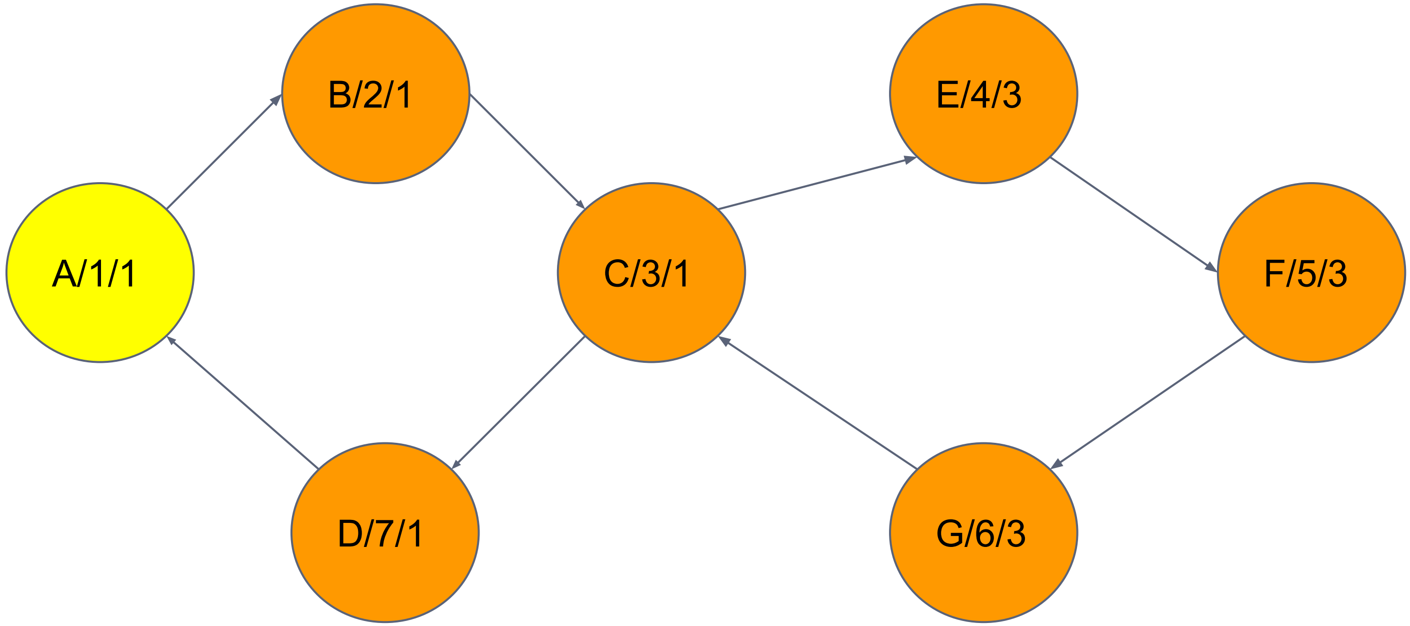


Visited ancestor minimization case

- lowLink[node]
  - Same as dfn # for SCC root
  - Lower than dfn for others
  - **<u>NOT</u>** guaranteed other nodes to have root dfs #
  - then low for a non-root means = the highest reachable

    ancestor within this dfs flow NOT the root ancestor of SCC
- what about:
  - lowLink[node] = min(lowLink[node], **<u>dfn[ch]</u>**);
  - Find the **first** ancestor root of my **internal** cycle
  - That is why it works too. Even it has a better meaning.

Visited ancestor minimization case



using lowLink[node] = min(lowLink[node], **dfn**[ch]);

Both dfs search orders will give **same** low numbers for cycle C, E, F, G, C

More importance when comes to **Articulation Points**

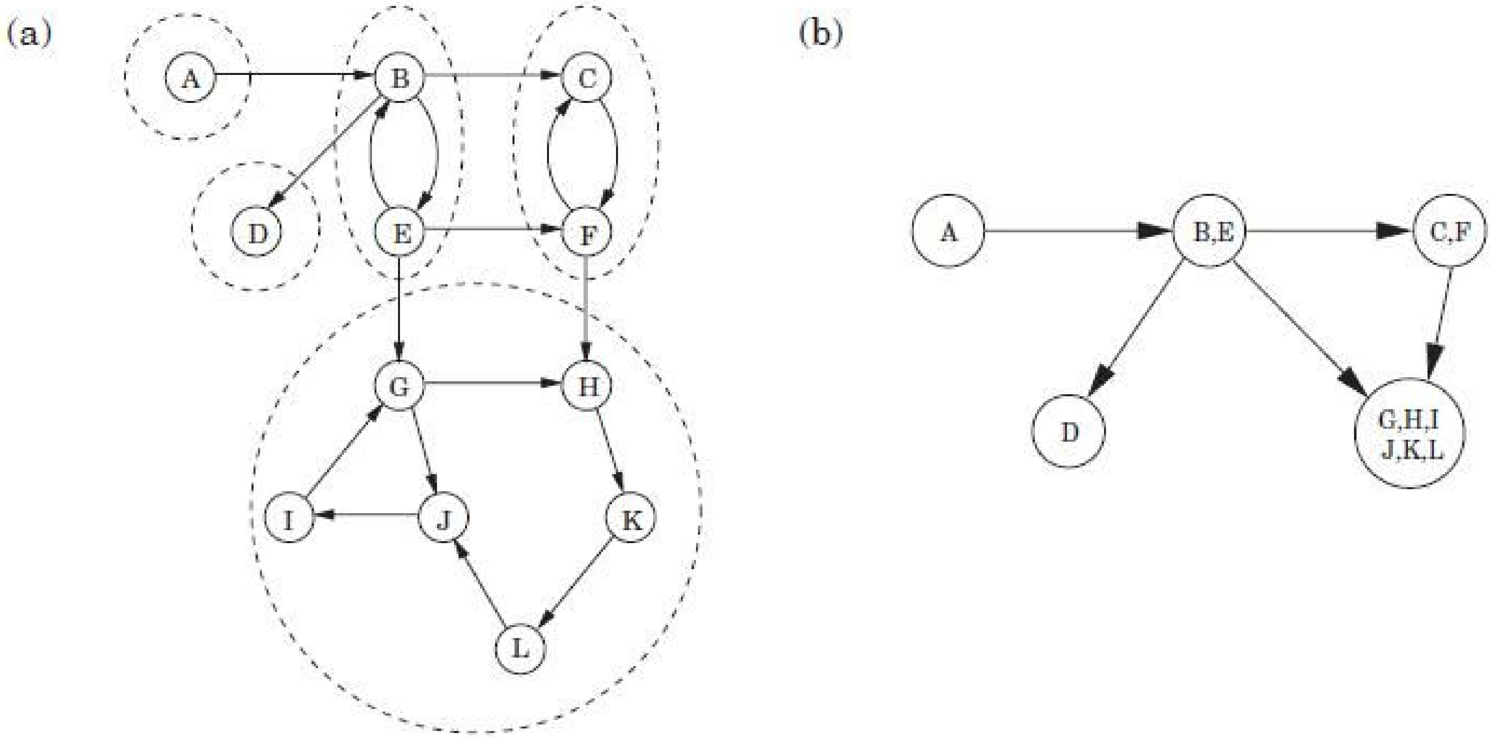
Get SCC

```
vector< vector<int> > adjList, comps;
vector<int> inStack, lowLink, dfn, comp;
stack<int> stk;
int ndfn;

void tarjan(int node) {
    lowLink[node] = dfn[node] = ndfn++, inStack[node] = 1;
    stk.push(node);

    rep(i, adjList[node])
        ...

    // only root has dfs # = low link #
    if (lowLink[node] == dfn[node]) {
        comps.push_back(vector<int> ());    // add new comp
        int x = -1;

        while (x != node) { // go till root
            x = stk.top(), stk.pop(), inStack[x] = 0;

            comps.back().push_back(x);  // add to new comp
            comp[x] = sz(comps) - 1;    // give it sequential ID
        }
    }
}
```

SCCs to Component Graph

- Now we know the components!
- Think each component is a node
- Build a NEW graph
  - # of nodes = # of components
  - Edge between (A, B) if A reaches B

SCCs to Component Graph



Src: https://upload.wikimedia.org/wikipedia/commons/2/21/InterpersonalBridge.jpg

SCCs to Component Graph

```
void computeCompGraph() {
    for (int i = 0; i < sz(adjList); i++)
        for (int j = 0; j < sz(adjList[i]); j++) {
            int k = adjList[i][j];
            if (comp[k] != comp[i])
                dagList[comp[i]].push_back(comp[k]);
        }
}
```

Component Graph..usages?

- What are min edges to add to graph to make it a whole cycle?
  - Get Component Graph
  - Compute Src and Dest nodes .. use simple equation
- Faster Transitive closure
  - Get Component Graph
  - Compute its closure
  - Compute whole closure

Kosaraju's algorithm

- We won’t focus on it…just to have idea
- **DFS 1**: Compute Nodes Topological order
- For each node in Reverse Topological Order
  - **DFS 2**: Just find reachable nodes on transposed graph.
  - These are SCC
- That is all…2 trivial DFS
  - See CLR for description Or See
  - Or even just think about reverse of topological order

    (which equal to reverse of DFS finish nodes) + transpose graph

Kosaraju's algorithm

```
void dfs_topsort(vvi& adj, vector<bool>& used, vi& topsort, int node)
{
    int i;
    used[node] = true;
    for (i=0;i<sz(adj[node]); ++i)
        if (!used[ adj[node][i] ])
            dfs_topsort(adj, used, topsort, adj[node][i]);
    topsort.push_back(node);
}

void dfs_scc(vvi& transpose, vector<bool>& used, vi& scc, int node)
{
    int i;
    used[node] = true;
    for (i=0;i<sz(transpose[node]) ; ++i)
        if (!used[transpose[node][i]])
        {
            scc[transpose[node][i]] = scc[node];
            dfs_scc(transpose, used, scc, transpose[node][i]);
        }
}
```

تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً