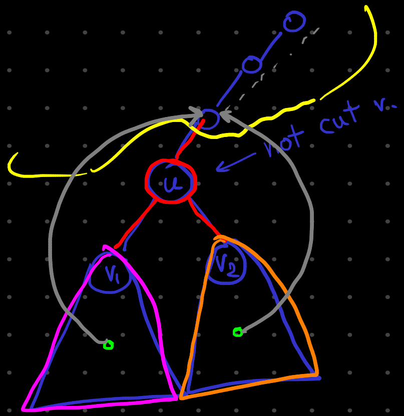
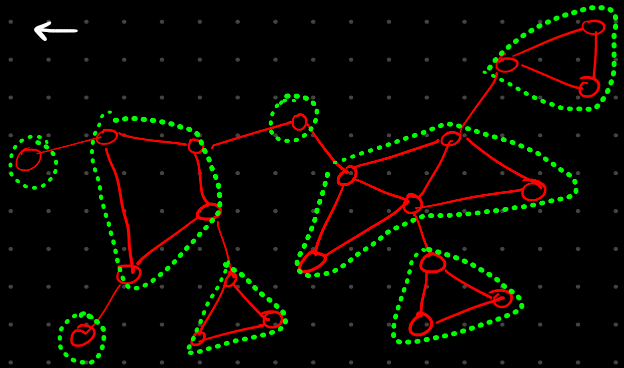
Introduction To Graph Theory

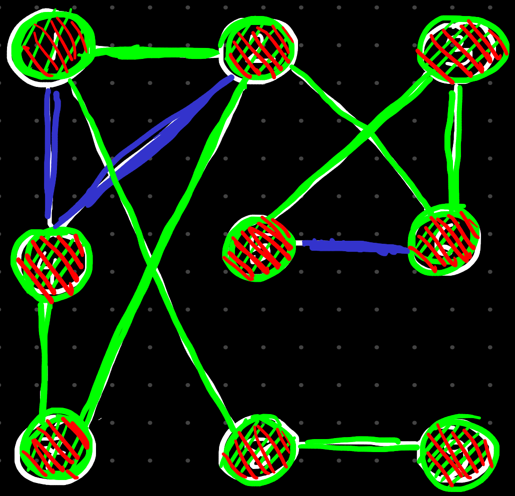
- DFS Tree applications
- Bridge, Articulation Points
- Topological Sorting

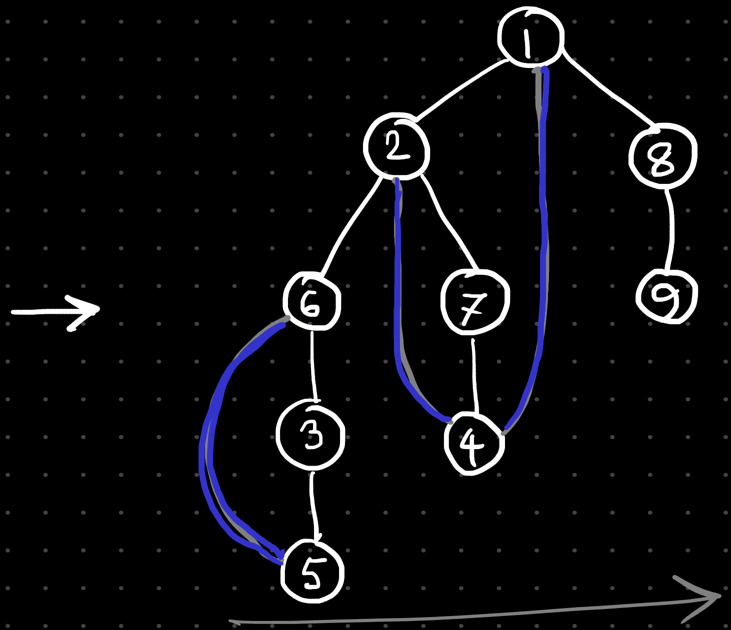


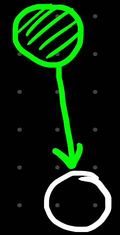


The DFS Tree

Running dfs in an undirected graph will always generate a tree.
(assuming only 1 component)



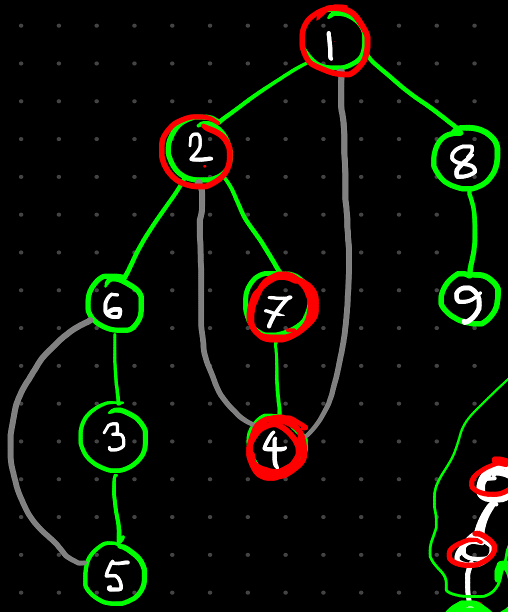




The DFS Tree

Span edge: Edges of the DFS tree

Back edge: Not part of the tree (ignored edges during dfs)

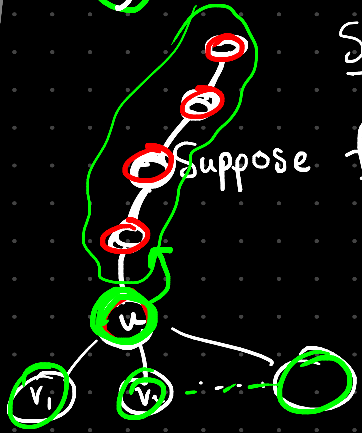


Observation 1:

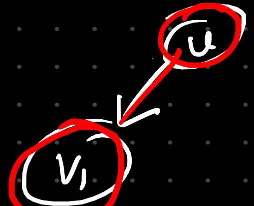
When the dfs reaches u in the graph, only u and it's ancestors are active.

Why??? We can prove by induction

Start : For the root, statement above is TRUE



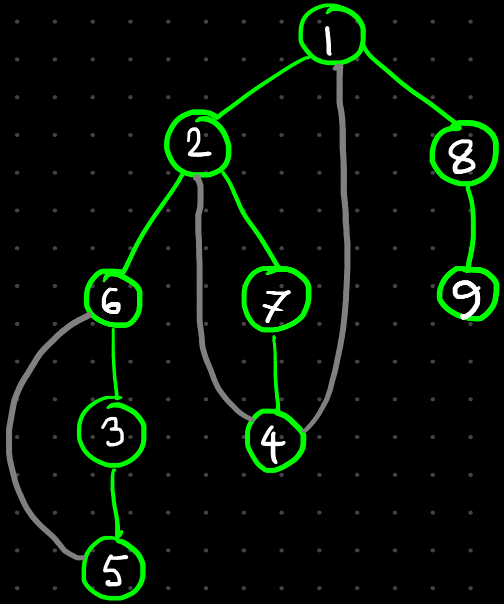
Suppose for node u, the claim is good.



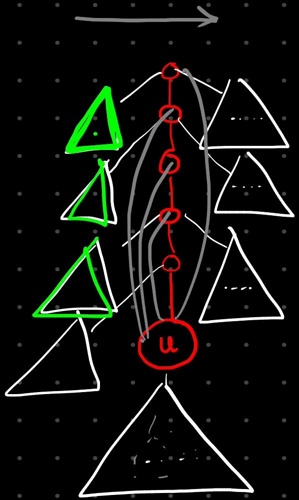
The DFS Tree

Span edge: Edges of the DFS tree

Back edge: Not part of the tree (ignored edges during dfs)



Observation 2:
Back edge from u only goes to
one of it's ancestors in the DFS tree



Why???

u — v

$\boxed{\text{Suppose u is visited first}}$

u ⟷ v

u → v   v visited

⇒ v has its DFS running

This also means that back edges will go to an active node in dfs stack

Fixing cycle printing from last day

0 = unvisited

1 = active 2 = done

```
void dfs (int u) {
    visited[u] = true;
    for (int v : g[u]) {
        if (v == pi[u]) continue;

        if (visited[v] == false) {
            pi[v] = u;
            dfs(v);
        }
        else {
            // Detected cycle
            cout << "Found a cycle between: " << u << " " << v << "\n";

            int cur = u;
            while (cur != v) {
                cout << cur << ", ";
                cur = pi[cur];
            }
            cout << cur << "\n";
        }
    }
}
```

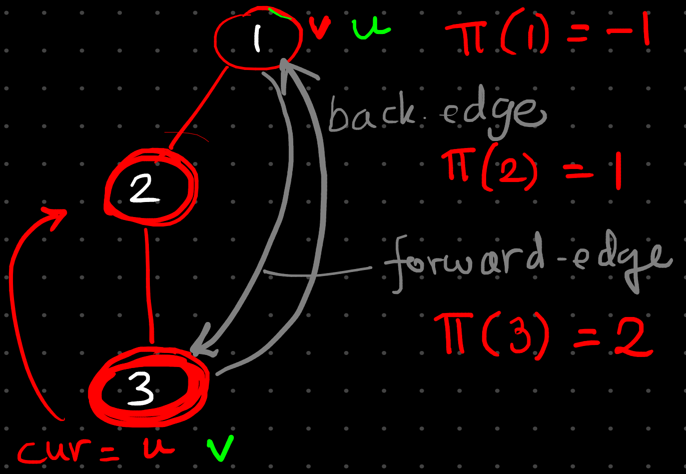
state[u] = 1

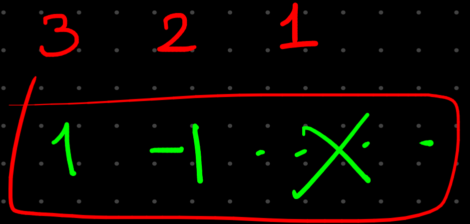
state[v] == 0

if (state[v] == 1) {

else if (state[v] == 2) → forward-edge

state[u] = 2



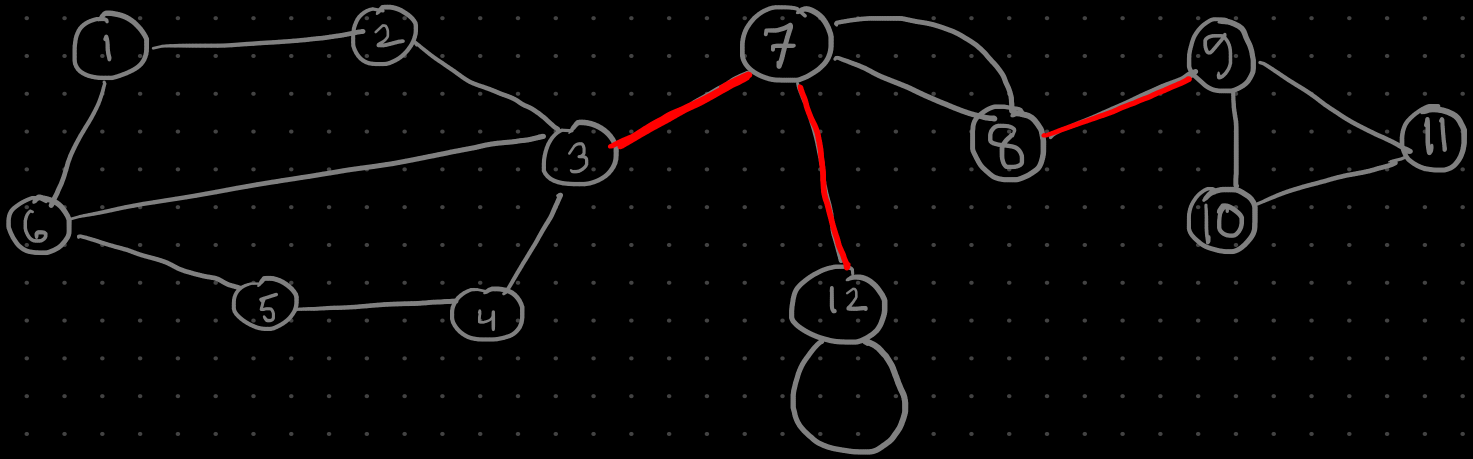


u — v when backedge?

when v is active in DFS!

Bridges in Undirected Graphs

Edges which upon deletion increase the number of connected components.

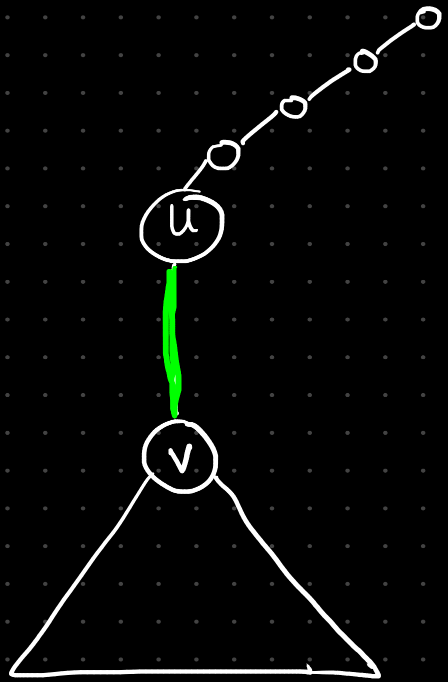


Bruteforce ? For all edge → try removing and then count # components using DFS

$O(E \times (V+E)) \approx O(E^2)$

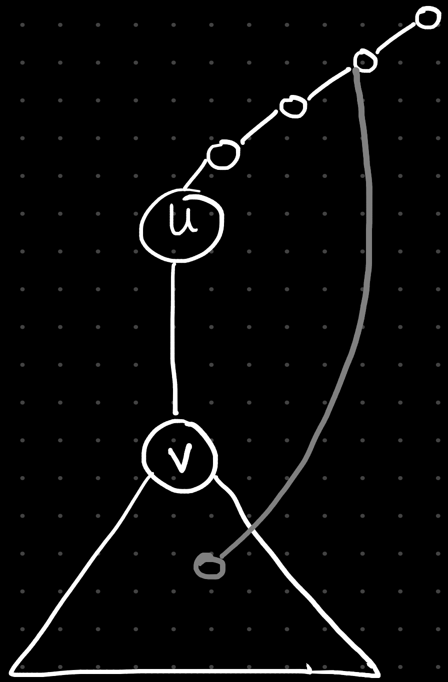
Finding Bridges Quickly

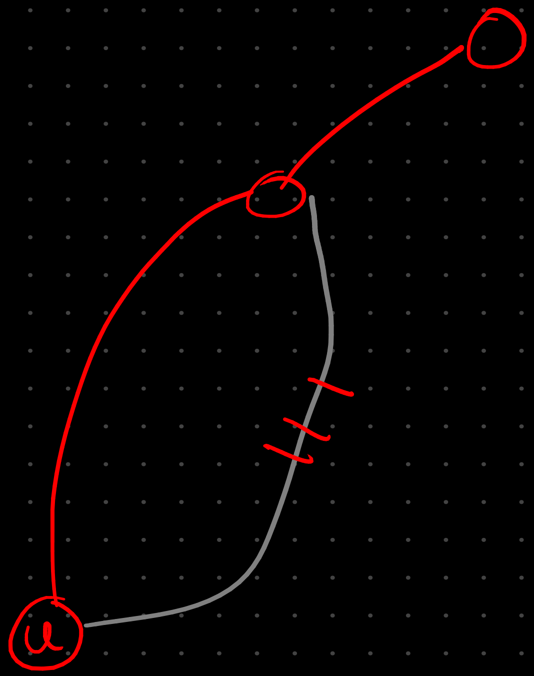
Lemma: A span edge u-v is a bridge iff there exists no back-edge that connects a descendent of u-v with an ancestor of u-v.



Finding Bridges Quickly

Observation: A back-edge is never a bridge.



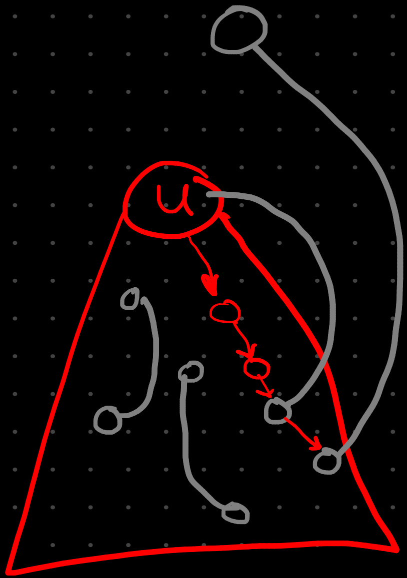


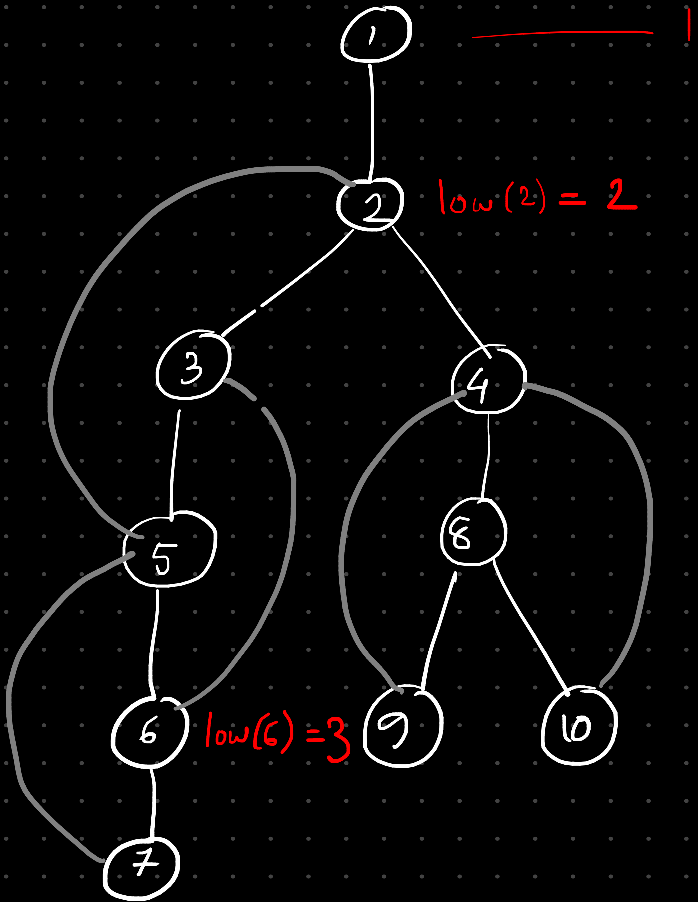
Bridge Finding Algorithm

Define,

$d(u)$ = depth of node u from root in dfs tree.

$low(u)$ = min depth node u that can be reached from u's subtree, using at most 1 back-edge





Bridge Finding Algorithm

Define,

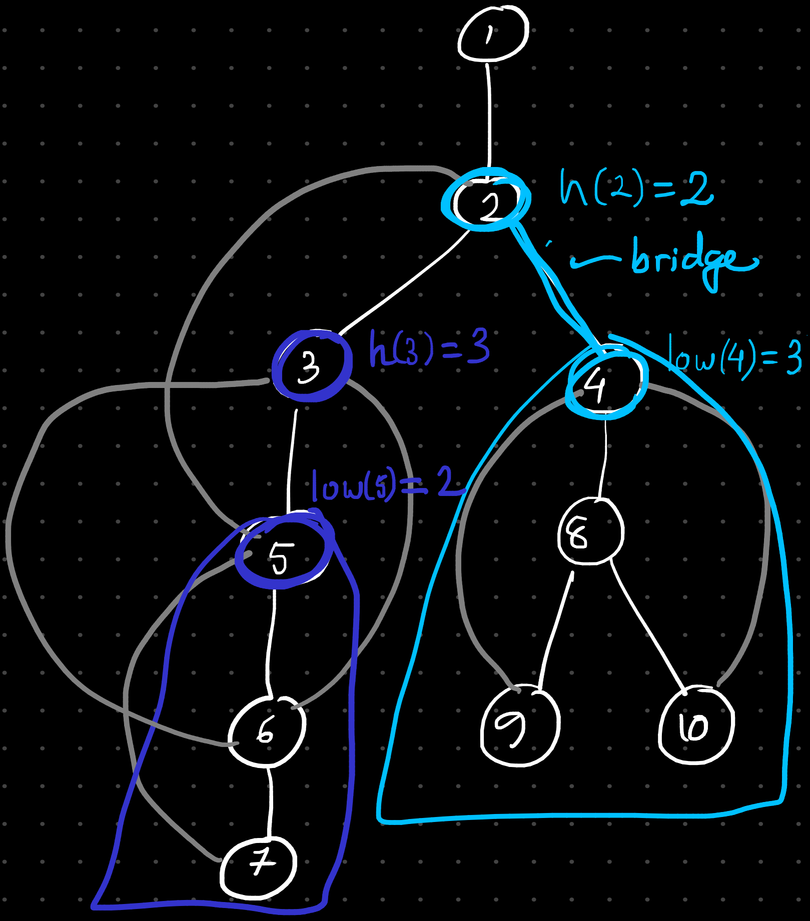
d(u) = depth of node u from root in dfs tree.

low(u) = min depth node u that can be reached from u's subtree, using at most 1 back-edge

An edge u-v is a bridge iff
v can't reach d(u) or higher from its subtree

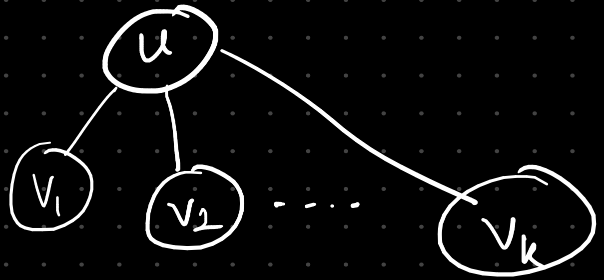
(assuming h(u) < h(v))

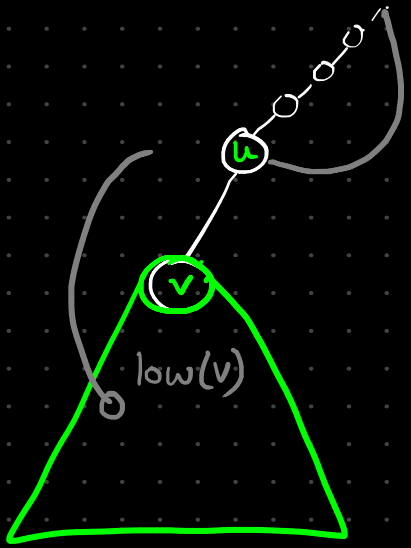
$$\boxed{d(u) < low(v)}$$



How to calculate low(u)?

low(u) = min depth node u that can be reached from u's subtree, using at most 1 back-edge





initially $low(u) = d(u) \quad \forall u.$

$u - v$ (tree edge): $low(u) = \min(low(u), low(v))$

$u - v$ (back edge): $low(u) = \min(low(u), d(v))$

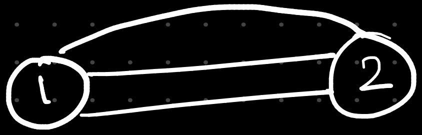
Implementation

f = pi[u]

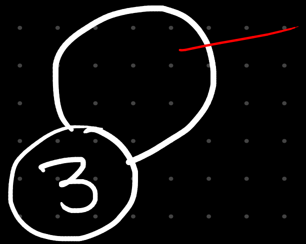
```
int d[N], low[N];
bool vis[N];

void dfs (int u) {
    vis[u] = true;
    d[u] = d[pi[u]] + 1;

    for (int v : g[u]) {
        if (v == f) continue; // ignore edge to parent
        if (vis[v]) low[u] = min(low[u], d[v]); // back-edge
        else {
            dfs(v);
            low[u] = min(low[u], low[v]);

            if (d[u] < low[v]) {
                // This edge is a bridge
            }
        }
    }
}
```

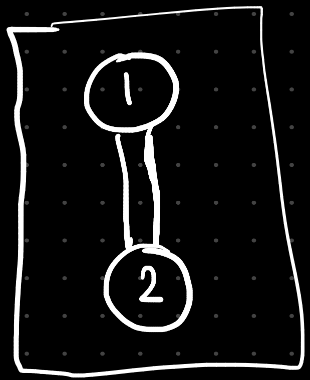
v == u →

Handling Multi-Edges and Self Loops



Multi-edge ??





Fix by making a list of all bridges.

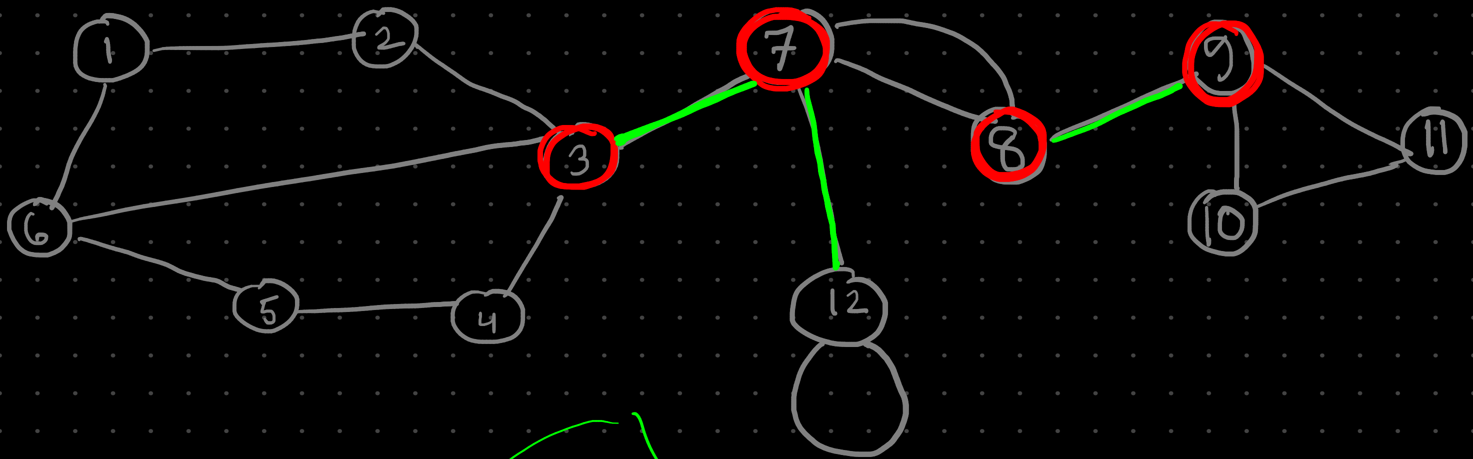
Sort all edges.

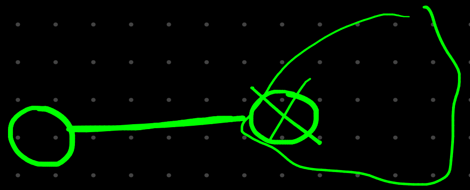
If for any (u,v) you find the same (u,v) adjacent to it,
delete all copies.

Articulation Points in Undirected Graph

cut point / vertex

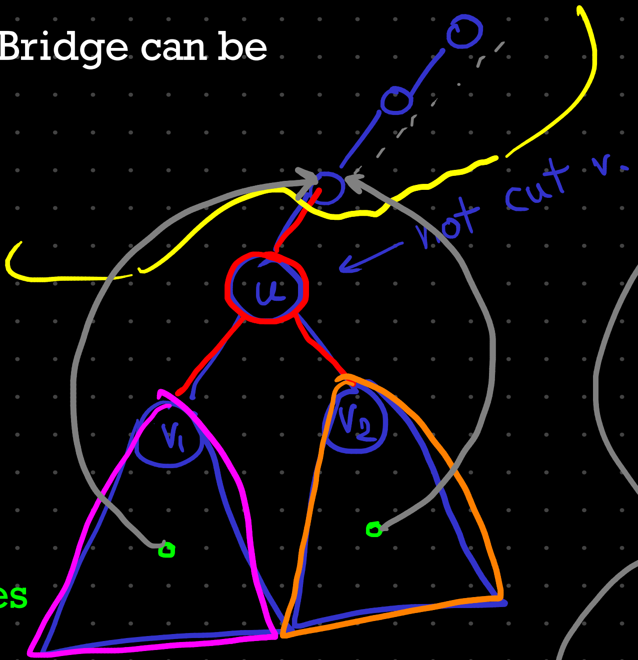
Nodes which upon deletion increase the number of connected components.





Articulation Point Finding Algorithm

It seems most of the ideas of Bridge can be re-used here.



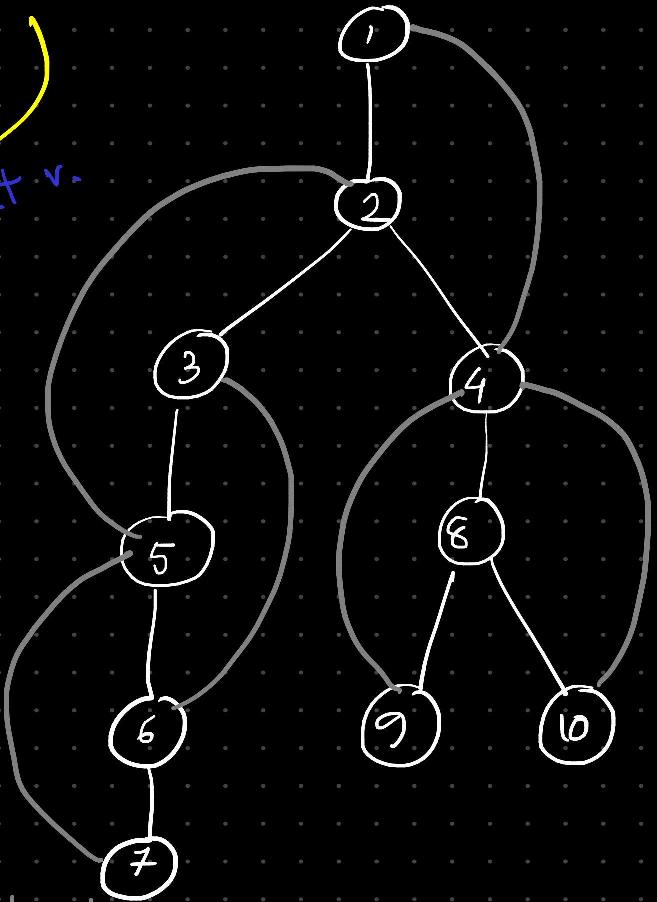
A node u is a cut vertex if

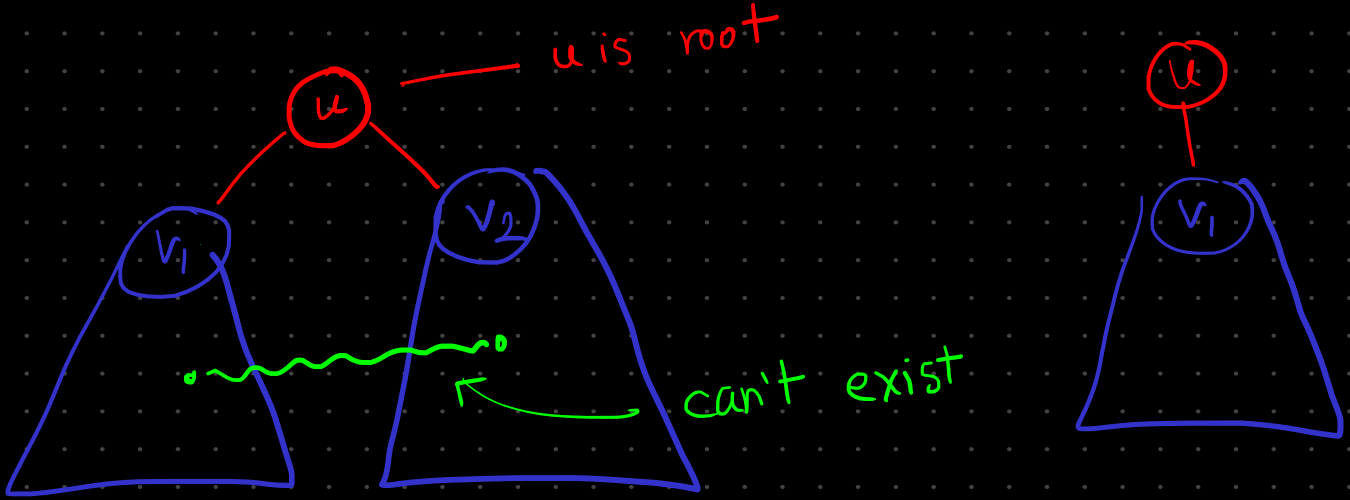
for any children v:
d(u) <= low(v)

only works for non-root nodes

$$\neg \begin{cases} low(v_1) < d(u) \\ low(v_2) < d(u) \end{cases}$$

$\Rightarrow$ d(u) <= low(v) for atleast one v.





If u is root,

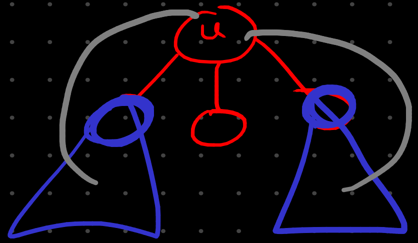
then u is cut point as long as it has >1 children.

Implementation

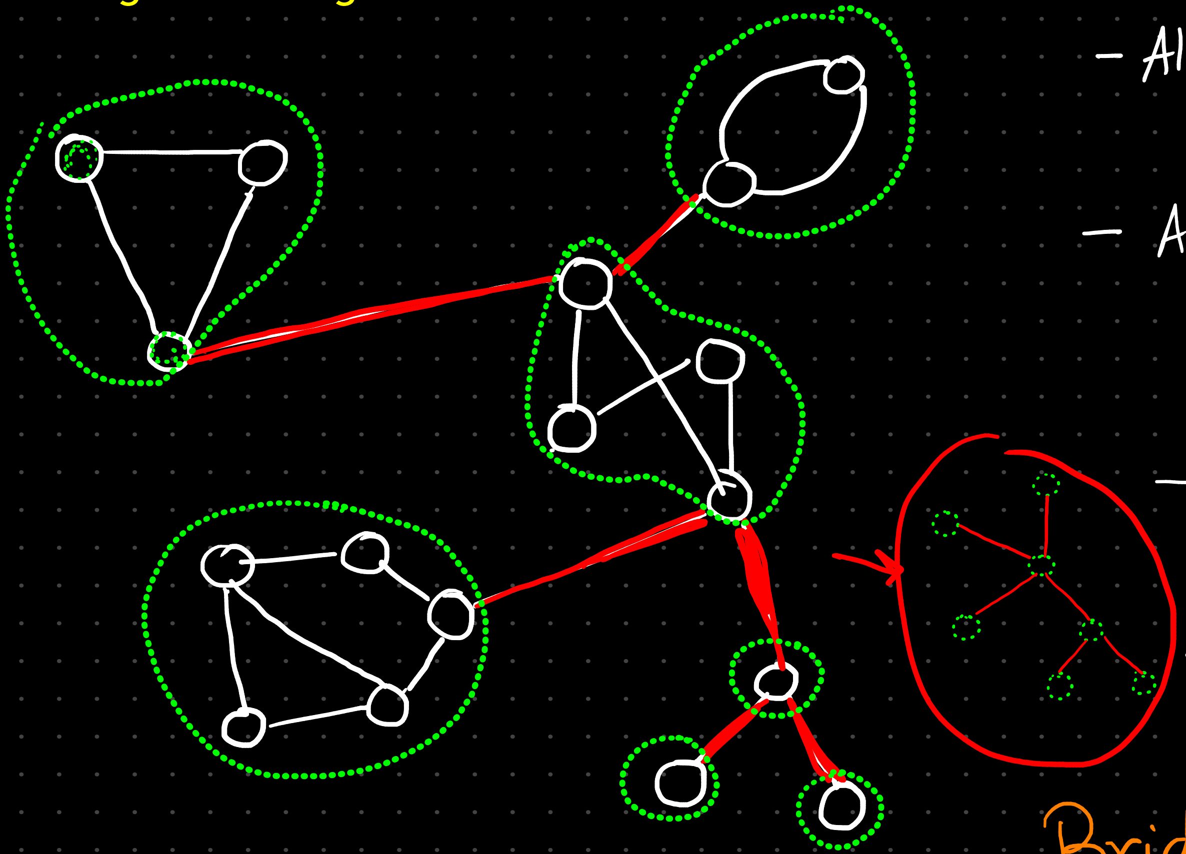
```
int d[N], low[N];
bool vis[N];

void dfs (int u) {
    vis[u] = true;
    d[u] = d[pi[u]] + 1;

    int nchild = 0;
    for (int v : g[u]) {
        if (v == f) continue; // ignore edge to parent
        if (vis[v]) low[u] = min(low[u], d[v]); // back-edge
        else {
            ++nchild;
            dfs(v);
            low[u] = min(low[u], low[v]);
        }

        if (d[u] <= low[v] && pi[u] != 0) {
            // u is a cut point and not a root
        }
    }
    if (pi[u] == 0 && nchild > 1)
        // root is a cut point
}
```

pi[u]

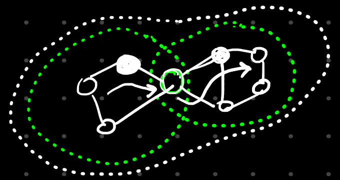
root



More Insight on Bridges



- All nodes belong to some component.
- All components are disjoint.



- The new graph has edges from the original graph.
- New graph is a tree

Bridge Tree

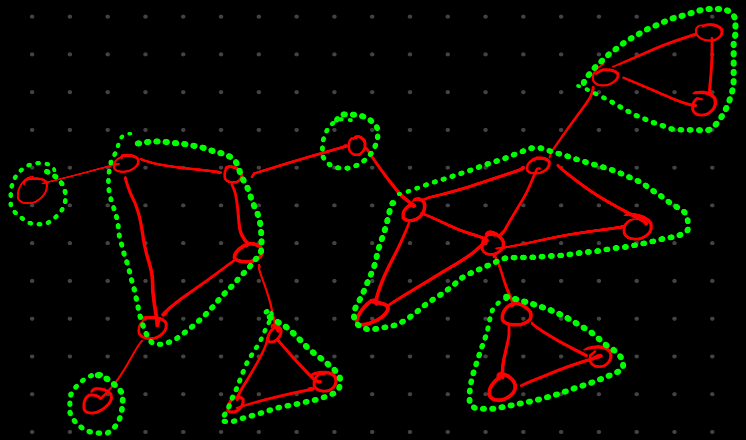
Some Problems

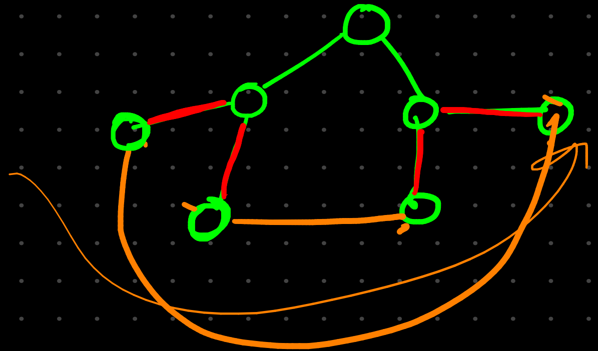
- Find the minimum number of edges to add in a graph so that no bridges exist.

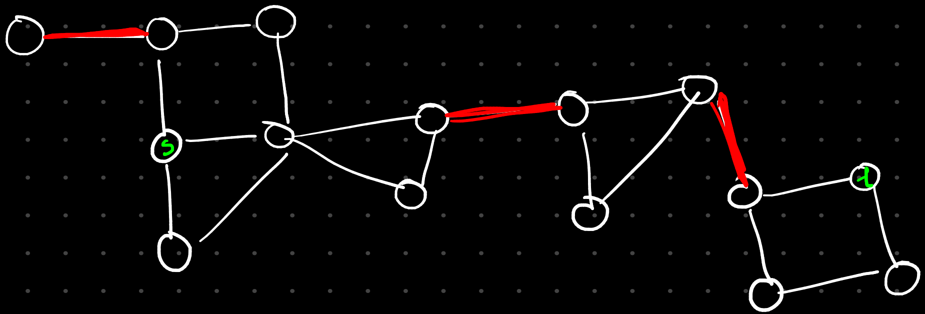
- Find the maximum number of edges you can mark so that there exists a path between two nodes which MUST use all the marked edges. (1000E - CF)

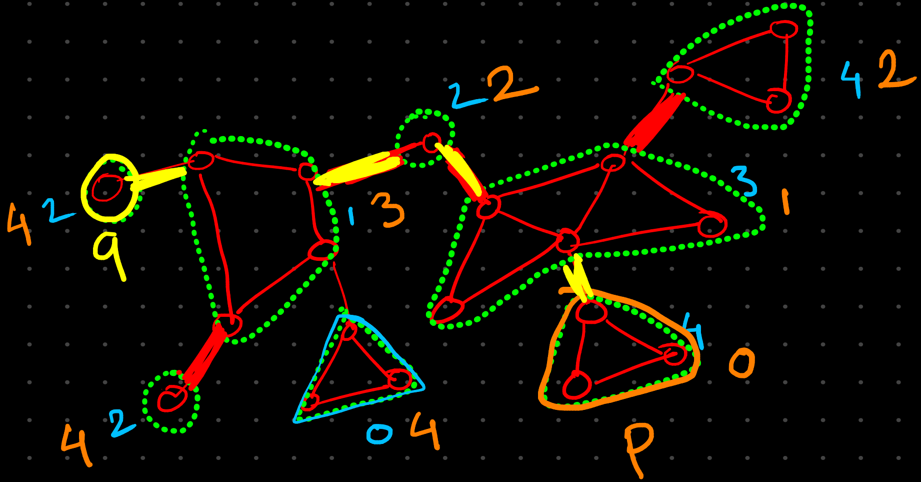
Implementation Practice: LightOJ 1063, 1026

$\left\lceil \frac{l}{2} \right\rceil$









Set $S = \{ \dots \}$

S is valid if s, t exist s.t. all paths from s to t must cross everything from S.

find $\max(|S|)$

Tree diameter

DFS(u)

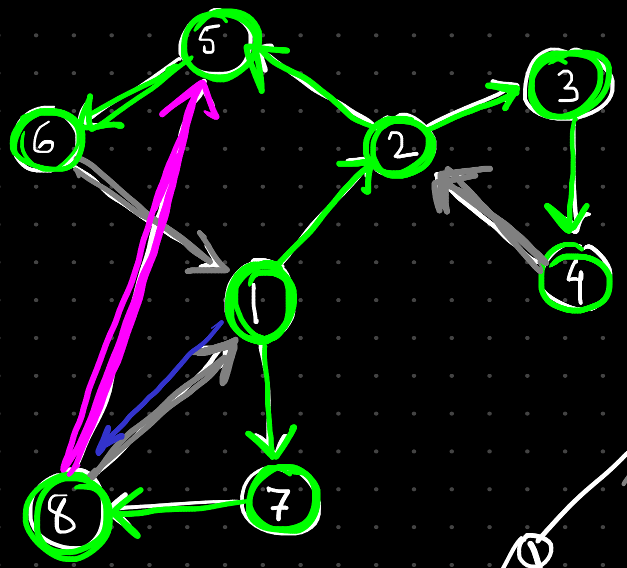
find p with max dist from u.

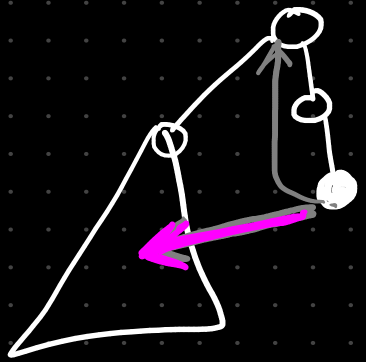
DFS(p)

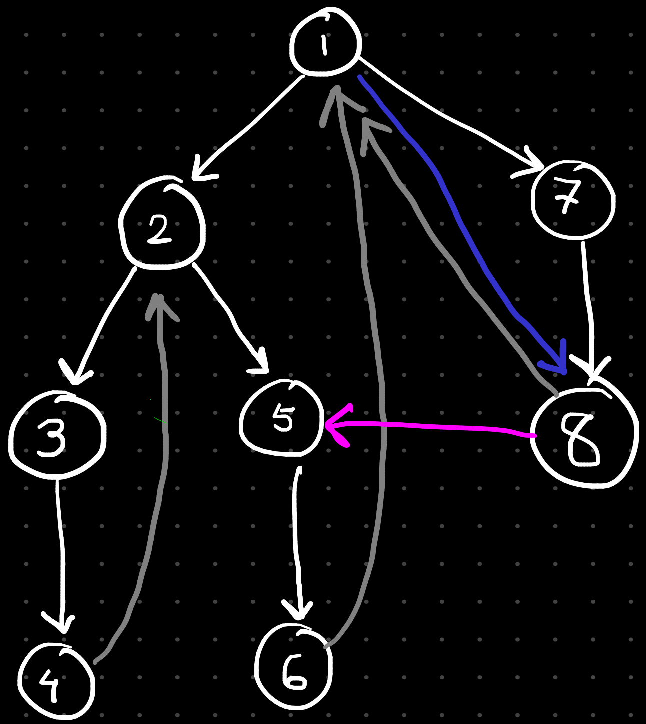
find q with max dist from p.

diameter = dist(p, q)

DFS Tree on Directed Graphs



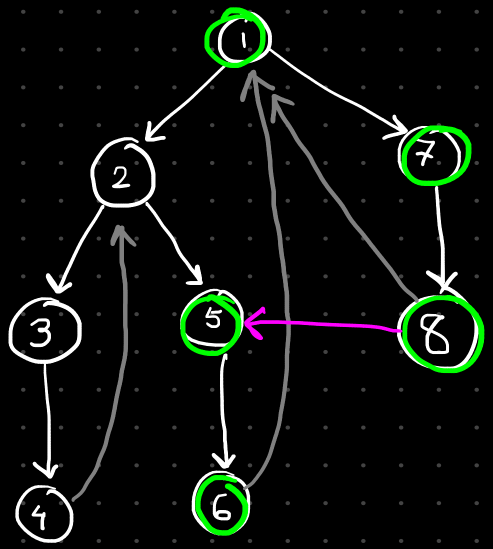




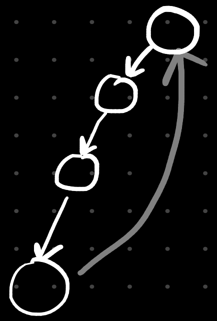
Cycles in Directed Graph

Observe that cross edges can't create cycles (?).

Only back-edges can create cycles.
Thus our previous cycle finding algorithm works here as well.



cross-edge, tree edge
back-edge

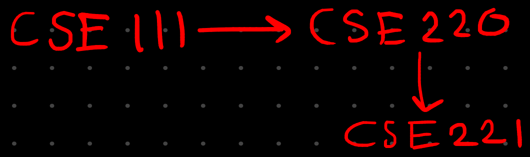


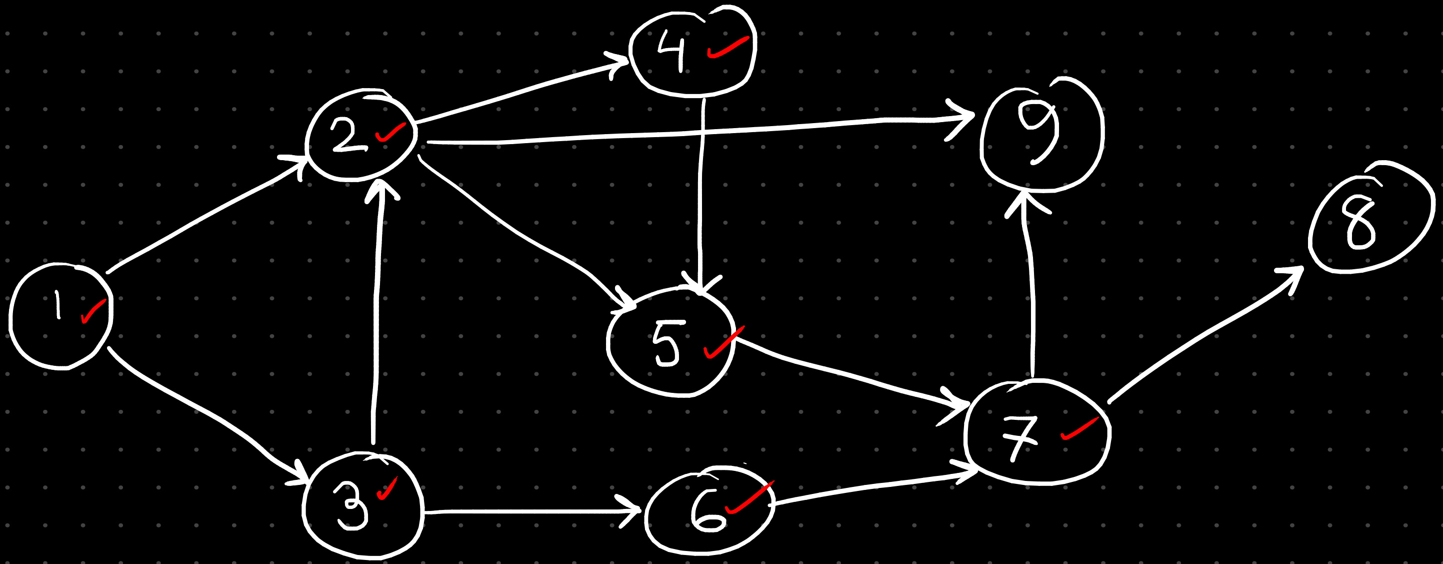
Topological Sort

Given a list of dependencies (a -> b) which represent:

a must be done before b is started

find an ordering that completes all the tasks.



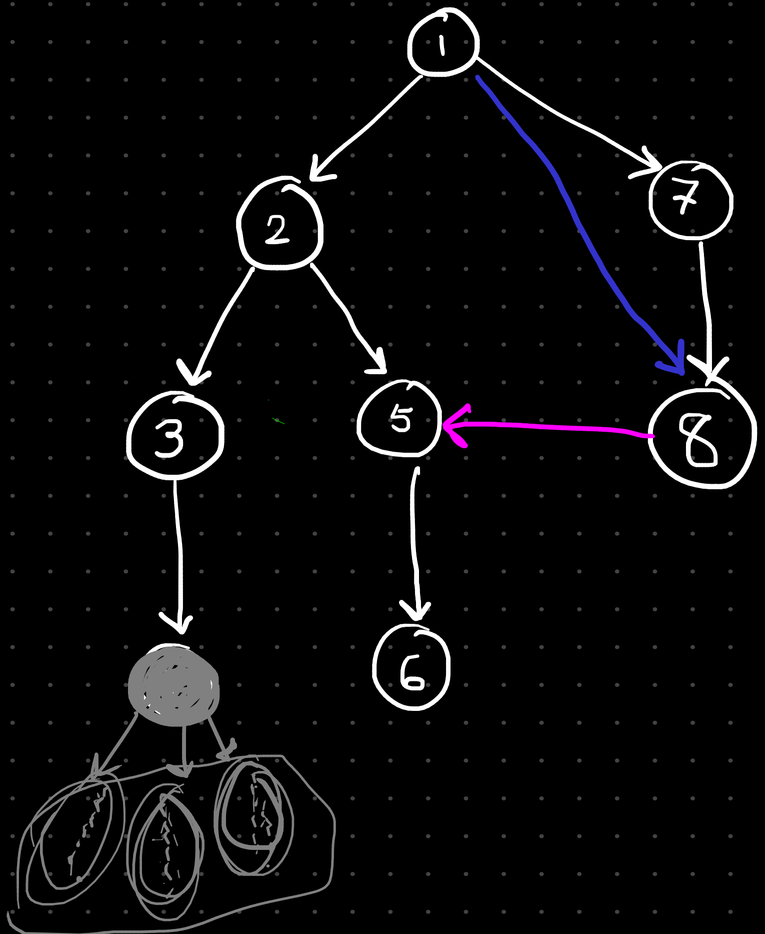


1 → 3 → 2 → 4 → 5 → 6 → 7 → 8 → 9

When is it impossible?

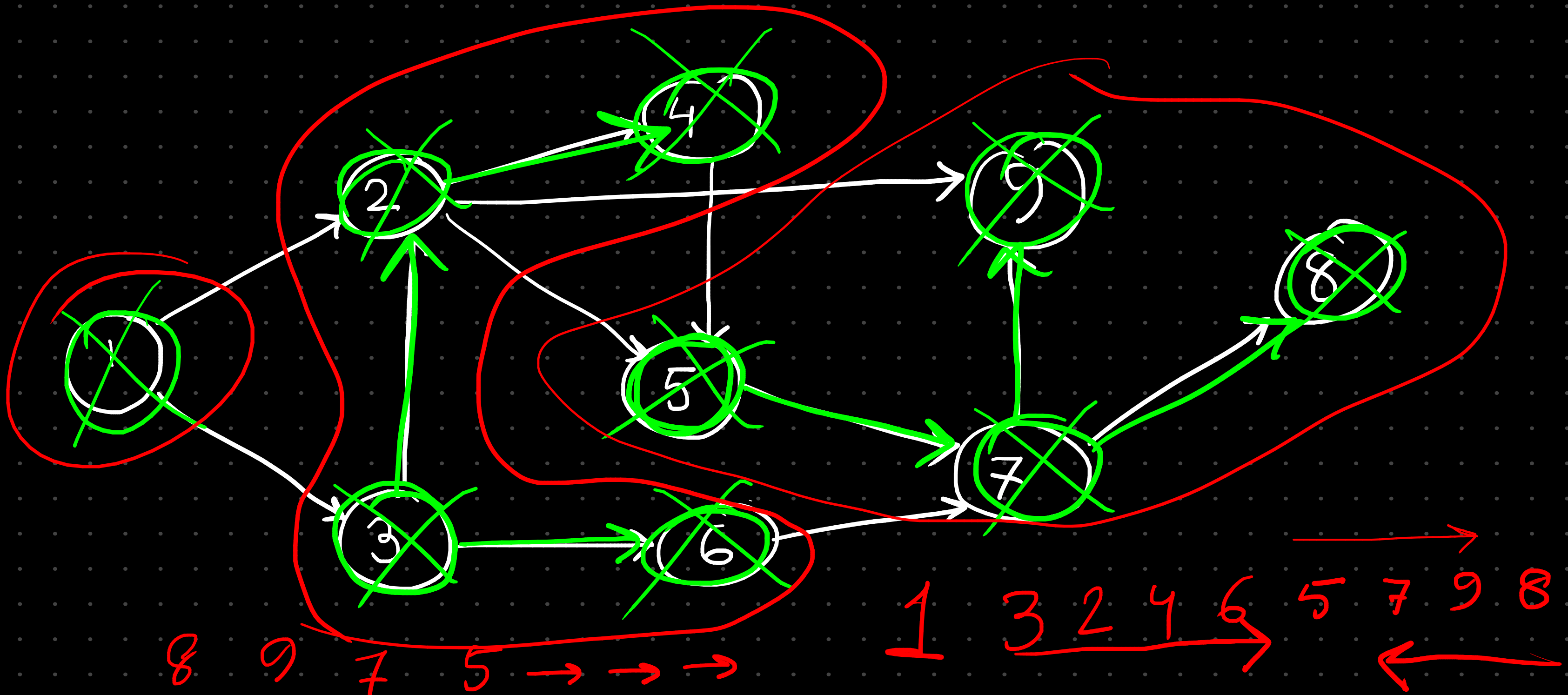
If there is a cycle in the directed graph.

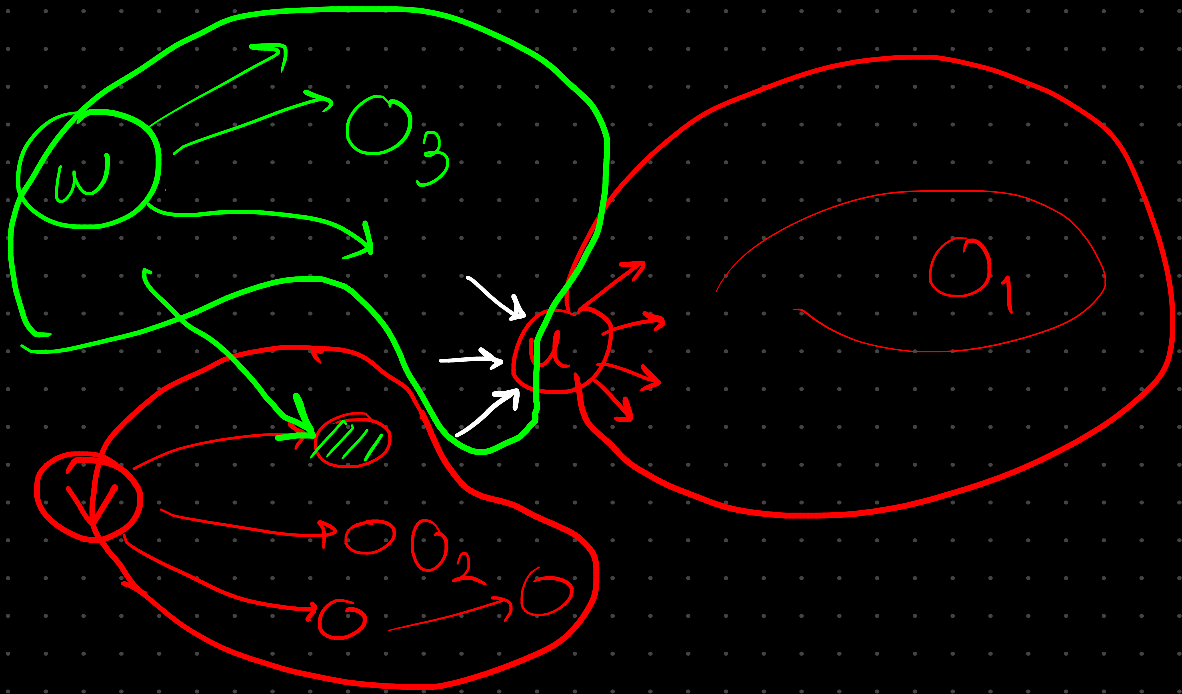
directed acyclic graph (DAG)


1
2
7
3
5
8
6

Topological Sort using DFS

Run a dfs.
Whenever a node finishes, we know all tasks that were supposed to happen after u is done.
So we can do this task u before all of them.





$O_3 \,,\; O_2 \,,\; O_1$

Implementation

```
bool vis[N];
vector<int> order;

void dfs (int u) {
    vis[u] = true;
    for (int v : g[u])
        if (!vis[v])
            dfs(v);
    order.push_back(u);
}

void topsort () {
    for (int i = 1; i <= n; ++i)
        if (!vis[i])
            dfs(i);
    reverse(order.begin(), order.end());
}
```

Problems

- CF510C
- CSES1679, 1680, 1681

s_1 a b
s_2 a b b c
s_3 a c d
s_4 b a a c

$s_1 \le s_2$
$s_1 \le s_3$

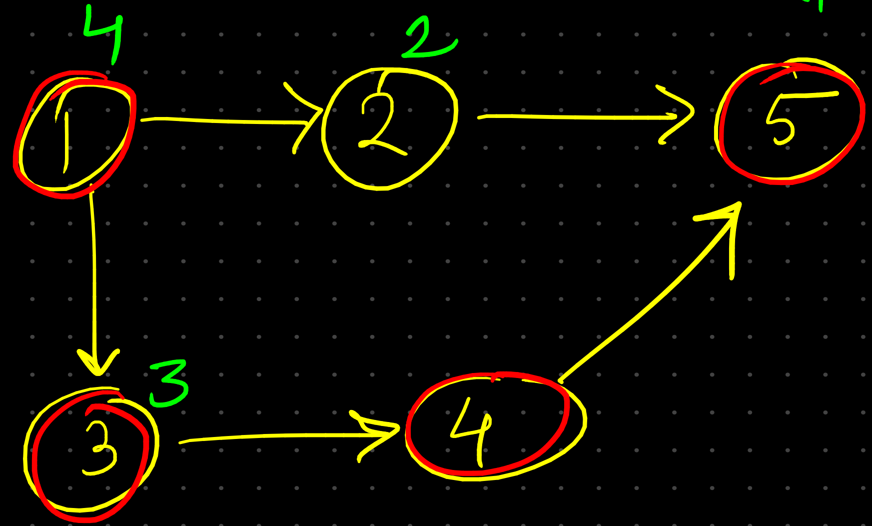
$$\boxed{\begin{matrix} b < c \\ a < b \end{matrix}}$$

$a \to b \to c$

a b c x y z . . . .

$O(n^2 \times |s_i|)$

$dp[u]$ = max dist from u.



$$dp[1] = \max\begin{cases} 1+dp[2] \\ 1+dp[3] \end{cases}$$

1 2 3 4 5

$dp[5] = 1$

$dp[4] = 1+1 = 2$

$dp[2] = 2$

$dp[3] = 3$

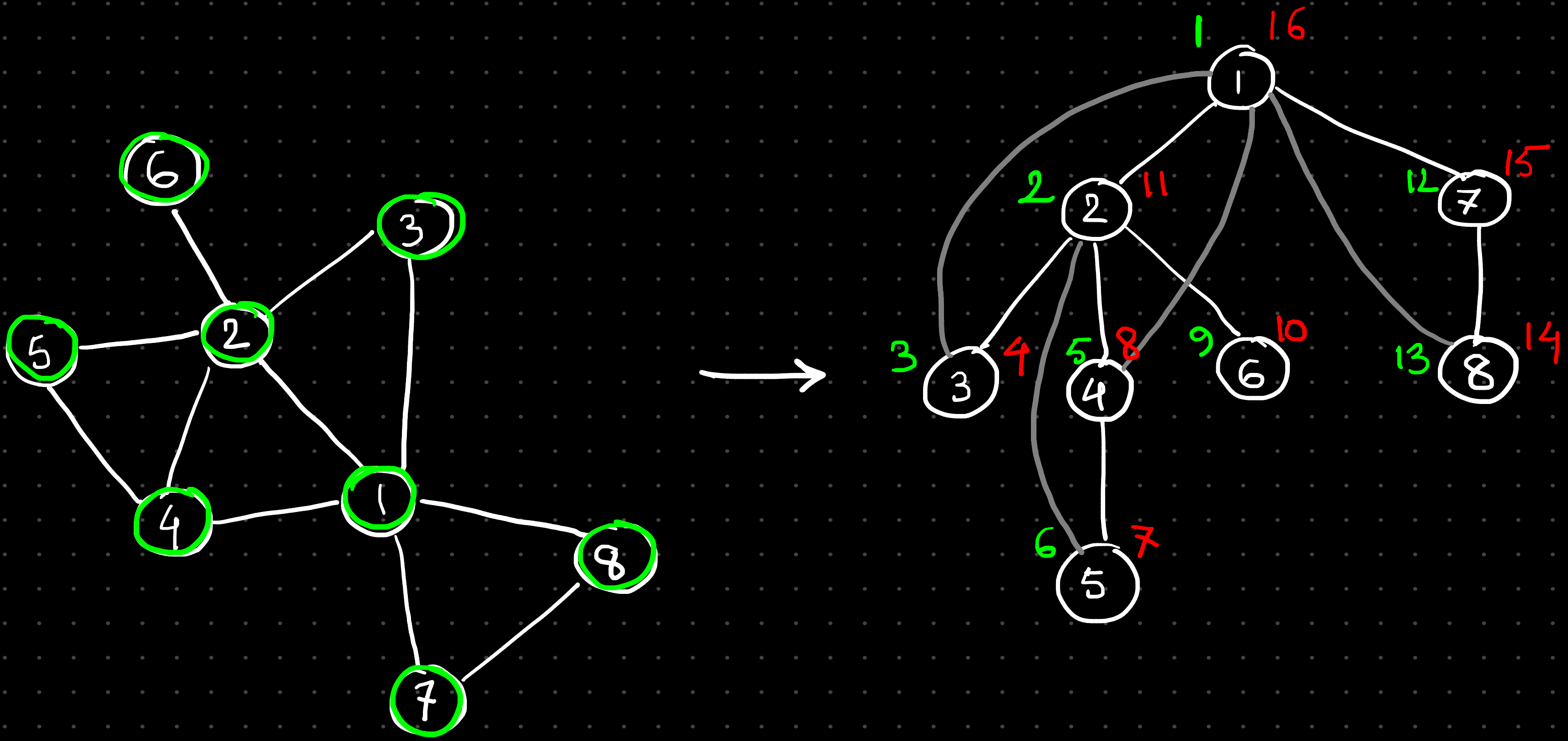
$dp[1] = 1+3 = 4$

DFS Timers

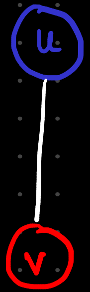
Timers are an easy way to keep track of which edge is which type.
Also has some cool properties.

```
int Timer, in[N], out[N];
int state[N], pi[N];

void dfs (int u) {
    in[u] = Timer++;
    state[u] = 1;
    for (int v : g[u]) {
        if (v == pi[u]) continue;  // remember to ignore immediate edge to parent

        if (state[v] == 0) {
            pi[v] = u;
            dfs(v); // span-edge
        }
        else if (state[v] == 1) {} // back-edge
        else {} // forward-edge , cross-edge
    }
    state[u] = 2;
    out[u] = Timer++;
}
```

DFS Timers

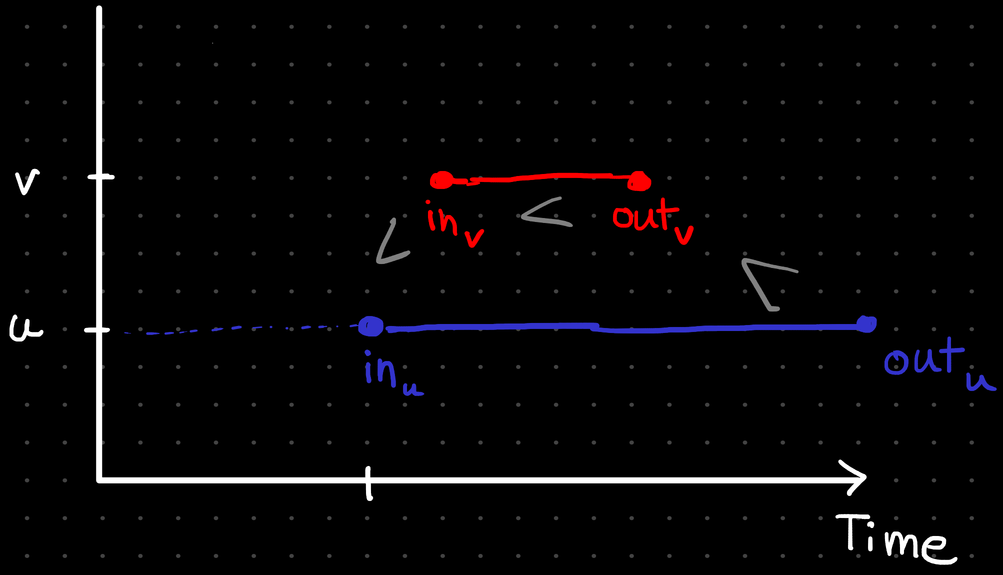


DFS Timers and Edge Types

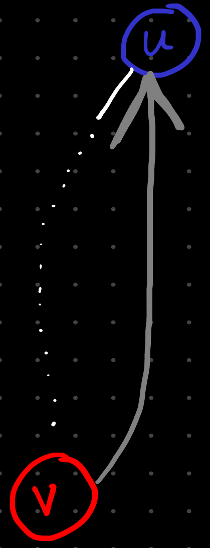


Tree edge

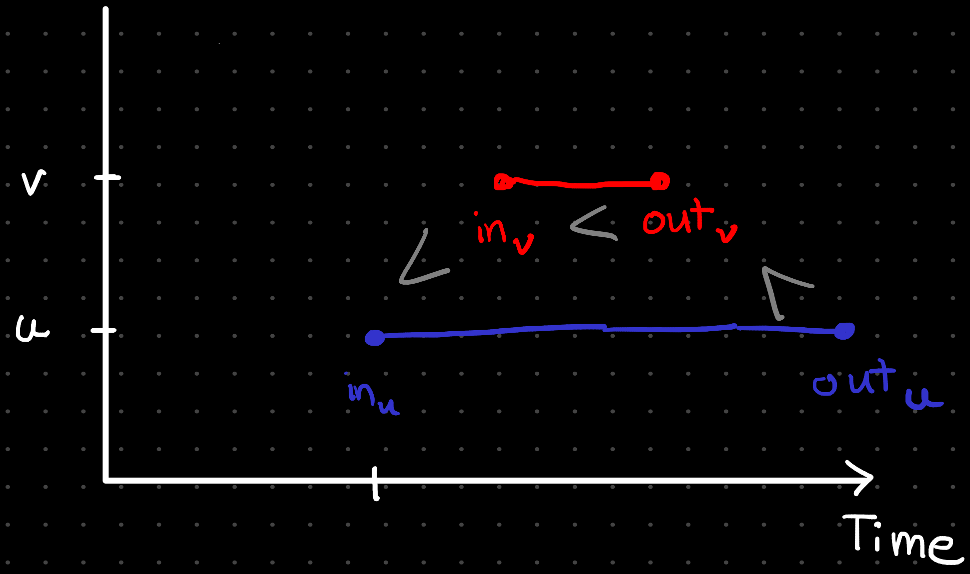
how does (in_u, out_u) and (in_v, out_v) pairs look like?



DFS Timers and Edge Types



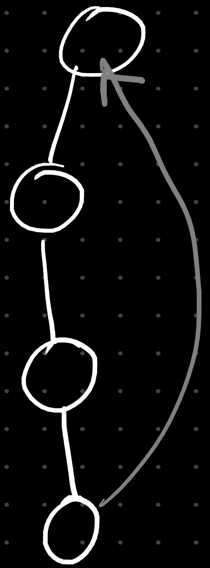
how does (in_u, out_u) and (in_v, out_v) pairs look like?



WAIT!!!

Span edges and Back edges look like the same?

How to differentiate?



$in_u < in_v < out_v < out_u$

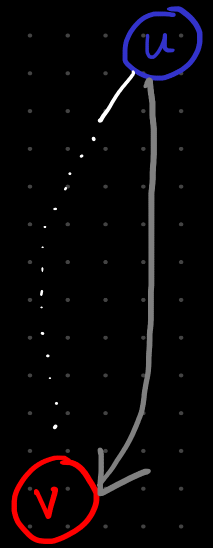
For backedges:

→ $\pi(u) \neq v$ and

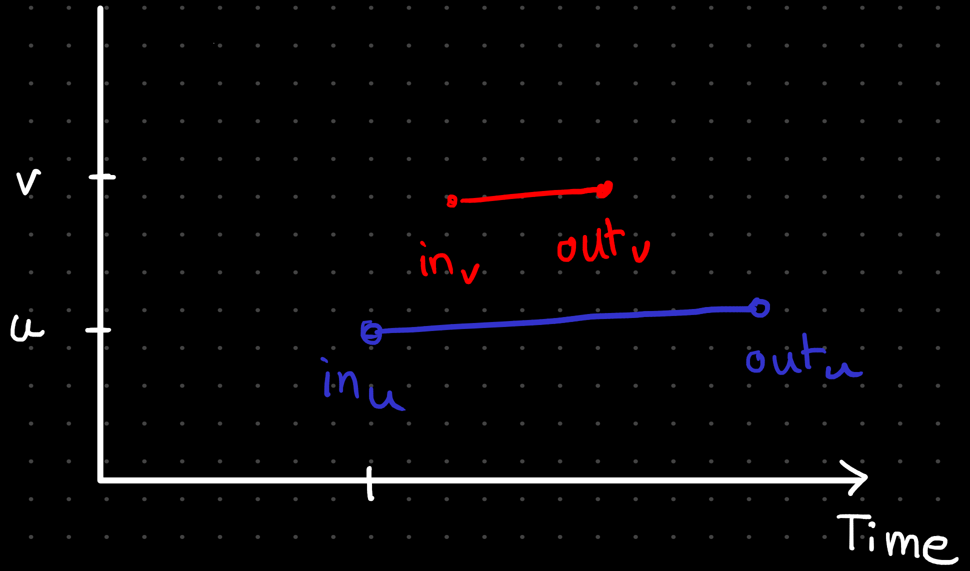
→ $\pi(v) \neq u$

DFS Timers and Edge Types

how does (in_u, out_u) and (in_v, out_v) pairs look like?



Forward Edge $u \rightarrow v$



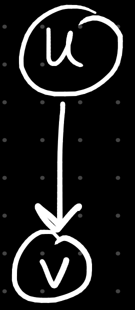
$u \rightarrow v$

TE $\pi(v) = u$

BE $in_v < in_u < out_u < out_v$

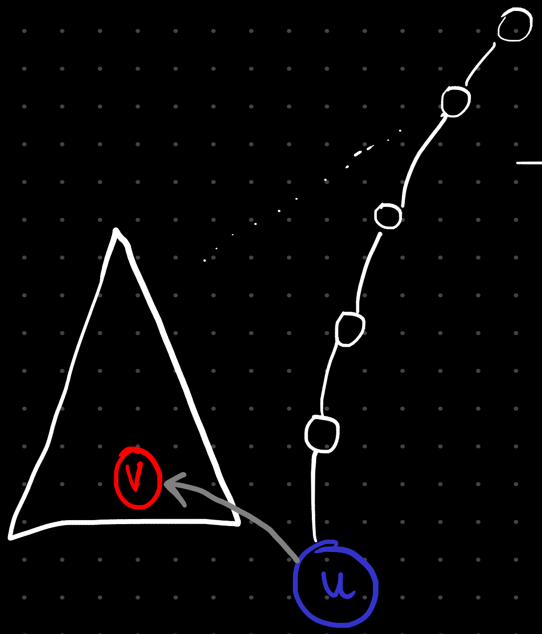
FE $in_u < in_v < out_v < in_v$

CE $out_v < in_u$



DFS Timers and Edge Types

how does (in_u, out_u) and (in_v, out_v) pairs look like?



Cross Edge