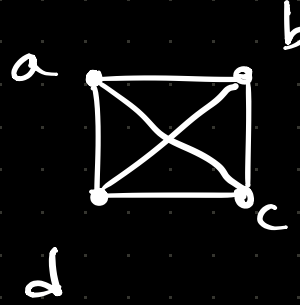
DFS

Depth-First Search

Depth First Search (DFS)

* Systematic exploration of a graph
* Undirected graph
* Directed graph



There's gonna be 2 categories of vertices :

① Visited ② Unvisited

we'll always choose an unvisited vertex at any point of traversal.

* Dead End : No more vertices to explore that are unvisited from the current vertex.

* If you reach a dead end during your traversal, the Control goes back to the previous vertex which called this dead end. This is called Backtracking.

```
DFS (G, u)
    mark u as visited
    For each v ∈ Adj(u)
        if v is NOT visited
            DFS(G, v)
```

} Recursive Backtracking

Terminology

Tree Edge: $(u, v) \in E \leftarrow G = (V, E)$ is a tree edge iff u is visited and v is not visited.

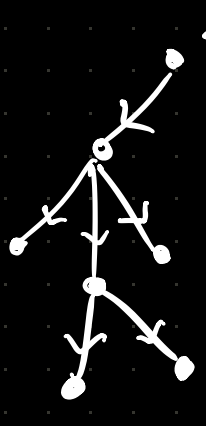
Non-Tree Edge: $(u, v) \in E$ — u visited, v visited

* Always go from visited to unvisited vertex.

DFS Tree: An out-tree

⇓

all the edges in the tree orient away from the root toward leaf.

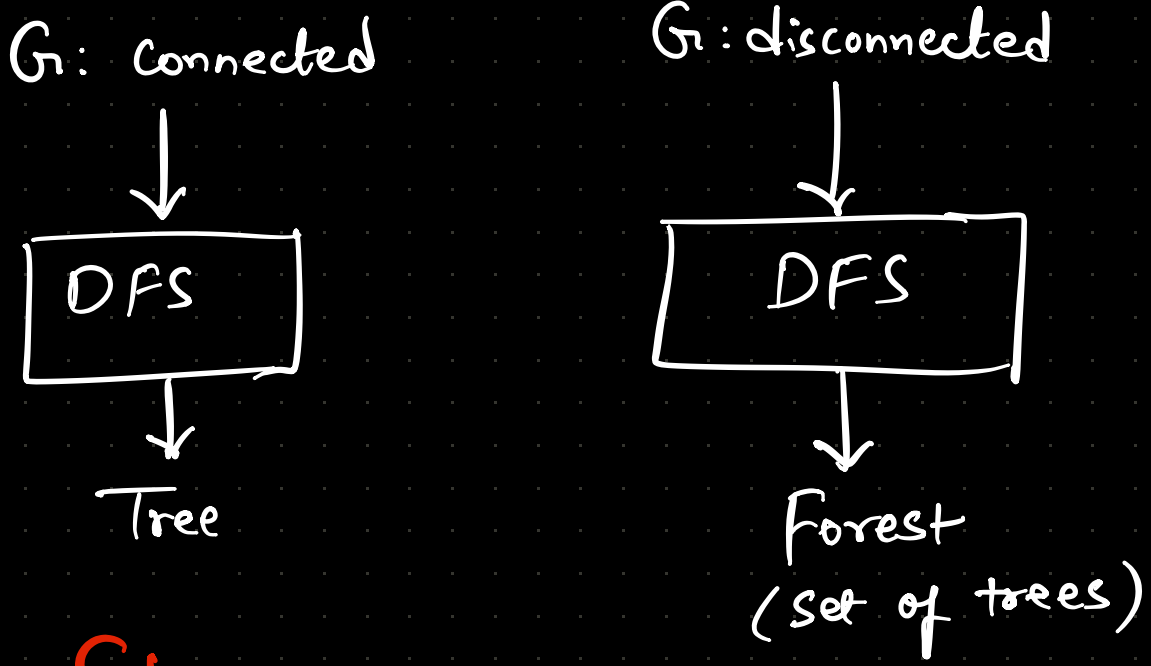


The non-tree edges are oriented toward the root and are called Back edges.

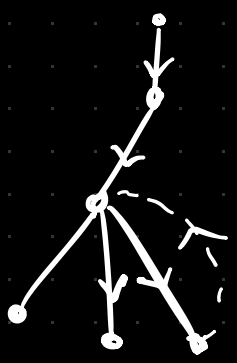
This view of "Back Edges" is important.

* when the input graph G is disconnected, then DFS will produce a forest ⇕ Acyclic Graph

This is called a DFS Forest.



State of a Node in DFS



node :

Visited

unvisited

Live

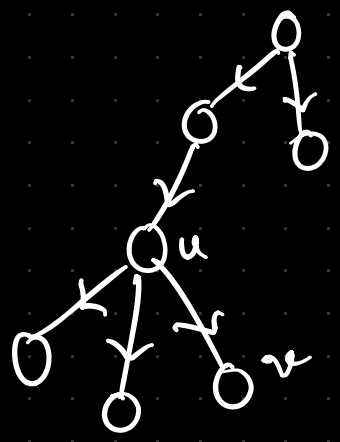
This is when the node's adjacent vertices are being explored.

Coloring The Vertices to know the State of the Node

white : NOT visited

Gray : Visited and Live

Black : Completed the visit & Left for its parent



u is the parent of v

$u = p(v)$ ✓

$v.p = u$ ✓

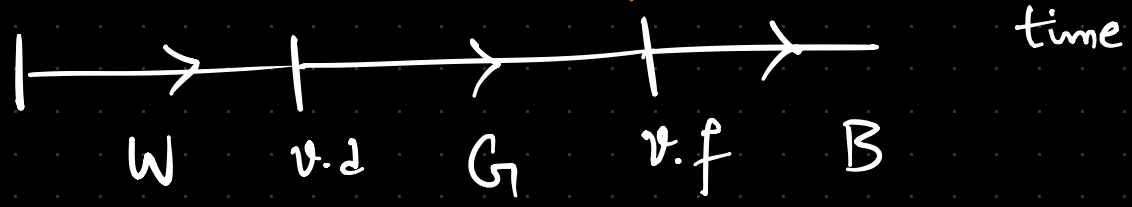
} Notation

∴ $(v.p, v)$ is a Tree Edge.

* we've 4 attributes for a node:
  - → v.color
  - → v.p
  - → v.d : discovery time (first time when you enter)
  - → v.f : finish time

* we are going to keep track of the Order in which the graph was explored (which vertex was visited first). TIME : global variable will tell us at what time the node's visited

a node and at what time the Computation on the current node is finished.



DFS (Undirected Graph)

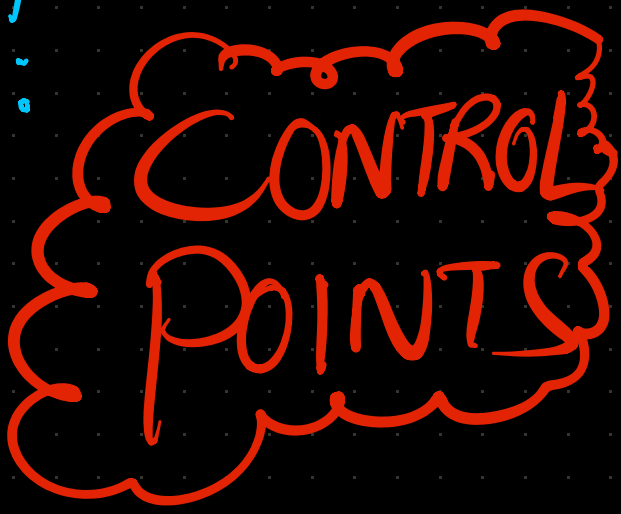
```
DFS (G, u):
    Time = Time + 1;
    u.d = Time;
    u.color = Gray;
    For each v ∈ Adj(u):
        if (v.color == white):
            v.p = u;
            DFS(G, v);
    Time = Time + 1;
    u.f = Time;
    u.color = Black;
```

This is called DFS Skeleton.

We may add more Computational Steps at various "Control points" of the above code

and generate more information about the graph.

Extended view of Computations on the graph with DFS :




```
DFS (G, u) :
    Time = Time + 1 ;
    u.d = Time ;
    // Discovery Time.
    // First time u is visited
    // (1) process at u when u is encountered
    // for the first time
    // (like preorder)
    u.color = Gray ;        // u is active now
    For each v ∈ Adj(u) :
        // (2) Include the computation to be
        // done for every edge here.
        // The current edge is (u, v)
        if (v.color == white)
```

```
    // (3) Include the computations to be done
    // on the tree edges here. The current
    // tree edge is (u, v)
    v.p = u;
    DFS(G, v);
    // (4) Include the computations to be
    // done while backing from v to u.
  else if (v == u.p)
    // (u, v) is the second copy of the
    // tree edge (v, u)
  else
    // (u, v) is a back edge
    // (5) include here the computations
    // to be done for the back edge.
Time = Time + 1;
u.f = Time;
// (6) Include the computations to be done
// while leaving u. (like post order)
u.Color = Black;
end;
```

There are 6 control points in the DFS skeleton.

One can perform additional computational steps and expand the DFS code to obtain more information about the graph "on the go."

One straight application of this approach is Finding the Cut Vertex in an undirected Graph, a canonical example.

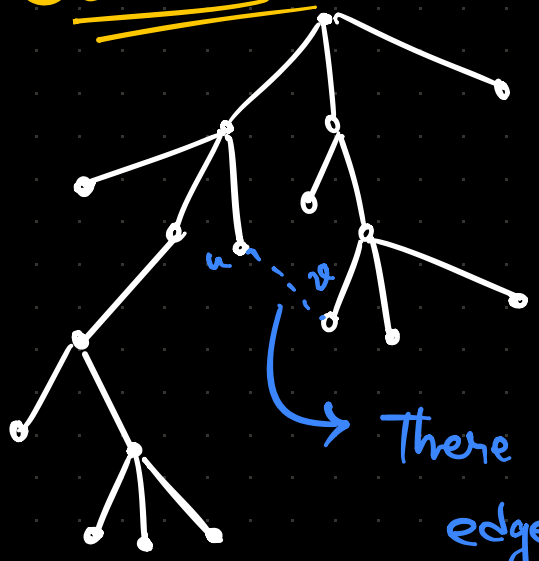
Cut Vertex Search using DFS

Cut Vertex : v is a cut vertex in a graph G iff $\exists$ a pair of vertices x, y $\ni$ every path b/w x & y contains v.

(Computationally Infeasible)

* Lets view G in terms of tree edges and non-tree edges.

Observation :



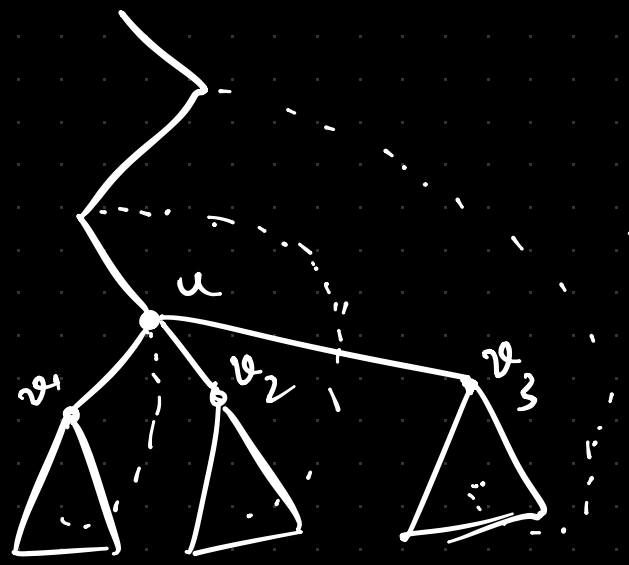
There won't be a non-tree edge like this because while performing DFS from u, we must have included (u, v) as a tree edge.

new attribute for each vertex

$v.cv = 0$ if v is not a cut vertex

$= 1$ if v is a cut vertex

Problem: Label all the vertices in the graph as cut vertex or not.



⇒ Removal of u disconnects v_1 but not v_2 and v_3. //

Descendants of v_1 are max. connected to u only, and thus removing u will disconnect v_1.

* **Low (u)** : = min. { v.d | v is a vertex reachable from u by using zero or more tree edges and zero or one back edge }

v.d → discovery time of the vertex v

Iterative DFS

* Recursion is quite intuitive and meaningful.
* Iterative version of recursion is not so intuitive and straightforward as recursion.

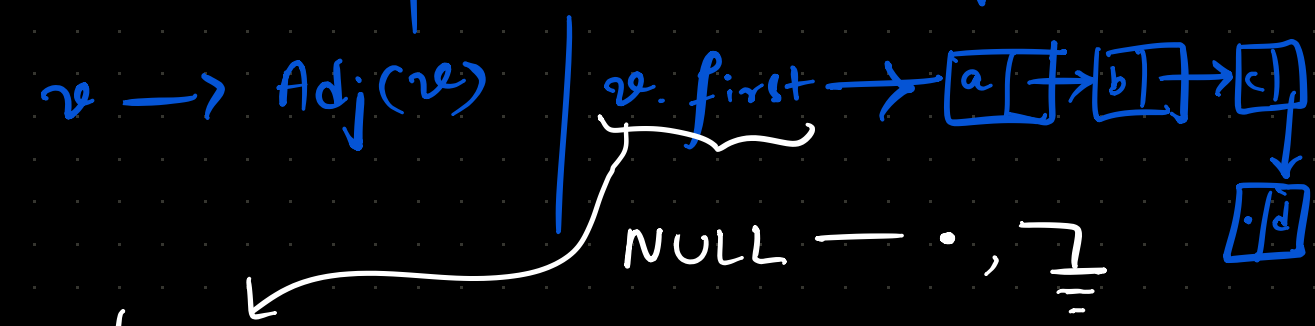
Recursive DFS :

```
DFS (G, u)
    mark u as visited ;
    for all v ∈ adj (u)
        if v is not visited
            DFS (G, v) ;
```

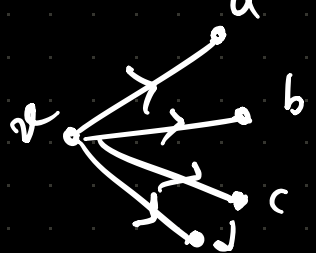
Vanilla DFS

Intuitive
Easy
Simple
Straightforward

* Let G be represented as Adjacency List



head of the linked list
containing the out neighbors of v.

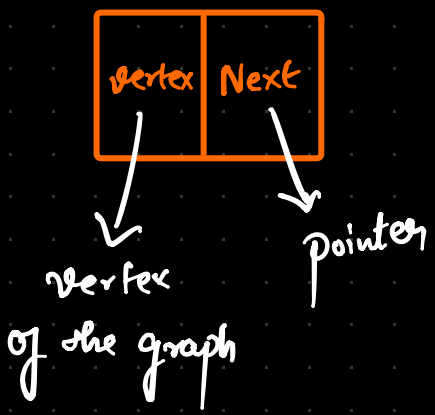


ptr : An independent pointer to indicate a specific position in the adjacency list.

It's more of like a running pointer in the list.

(v, ptr) : A position indicated by the pointer ptr in the adjacency list of v Adj(v).

A Box in the Linked List

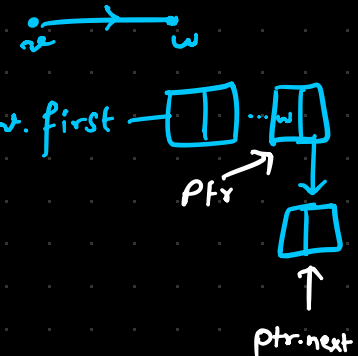


* Any recursive function in Computer maintains a System Stack (implicitly), in which the System stores the references related to all the references of the future computations.

* For this to be done explicitly, one has to maintain a stack for the same purposes.

Non-Recursive DFS (G, u)

```
S = φ;      // S is a stack containing
            // pairs, (v, ptr) that correspond
            // to the position of the neighbor
            // of v to be explored.
Time = Time + 1;
u.d = Time;     // discovery time of
                // the vertex u.
u.color = Gray; // u is active now
push ((u, u.first), S); // push the
                        // position indicator
                        // for u onto the stack
while (S ≠ φ)
    (v, ptr) = pop(S);
    if (ptr != NULL)
        push ((v, ptr.next), S);
        w = ptr.vertex;
        if (w.color == White)
            w.p = v;
            Time = Time+1;
            w.color = Gray;
            push ((w, w.first), S);
```



```
else                    // (v, w) is a non-tree edge
                        // so, do computations related to
                        // the non-tree edge (v, w).

else                    //(ptr == NULL)
                        // => exploration related to v is Complete
    v.color = Black;
    Time = Time + 1;
    v.f = Time;

// (v, ptr) ==(v, NULL) indicates that all
// neighbors of v were visited. Hence visit at
// v may be finished.
```

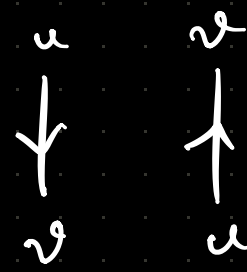
* Keep pushing the positions, for which explorations are still to be done, onto the stack.
* After finishing the visit corresponding to the position (v, ptr), we must continue with (v, ptr.next). That's why we push (v, ptr.next) onto the stack.

DFS (Directed Graph)

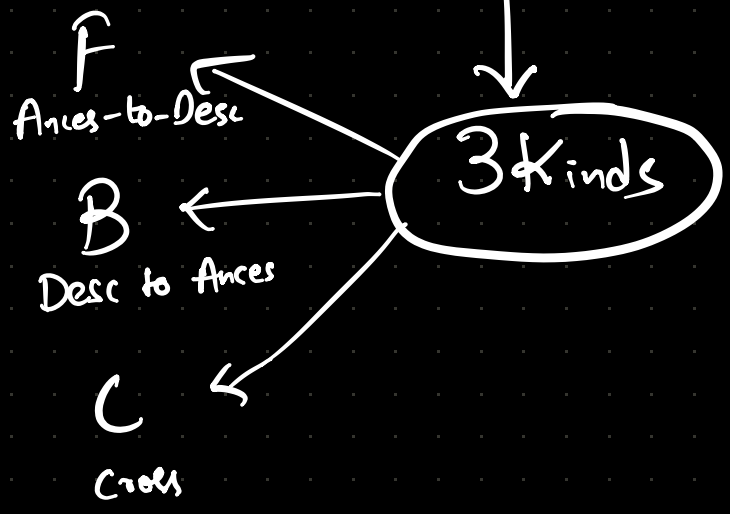
(visited, unvisited) —— Tree Edge

(visited, visited) —— Non-Tree Edge

$e = (u, v) \in E$



F B



* Time, the global variable takes the values 1, 2, ..., 2n → will take values

  1 to 2n Consecutive integers.

* Various vertices are active in various sub-intervals $[u.d, u.f]$ (d: discovery time, f: finish time)

These time intervals have Some significance and also have interesting properties:

① Disjoint Time intervals (or)

② Time intervals containing the other

BUT They never overlap

not possible ✗

possible ✓

$u.d < v.d < v.f < u.f$

disjoint ✓

* One has to keep track of the parent node for every node in directed graph also

$u.p \quad \forall u \in V$ in $G = (V, E)$

```
DFS (G)

    For each u ∈ V
        u.color = white;
        u.p = NULL;                    } Initialization

    Time = 0;

    For each u ∈ V
        if (u.color == white)
            DFS (G, u);
```

This is another sub-routine.

```
DFS (G, u)

    Time = Time + 1;
    u.d = Time;
    u.color = Gray;        // u is active now

    For each v ∈ Adj(u)
        if (v.color == white)
            v.p = u;
```

// (u, v) is a tree edge

DFS (G, v);

else

// (u, v) is a non-tree edge.

// The color of v can be Gray or Black

if (v.color == = Gray)

// (u, v) is a back edge (B)

if (v.f < u.d)

// (u, v) is a cross edge (C)

else

// (u, v) is a forward edge (F)

elaboration of Control point related to non-tree edges