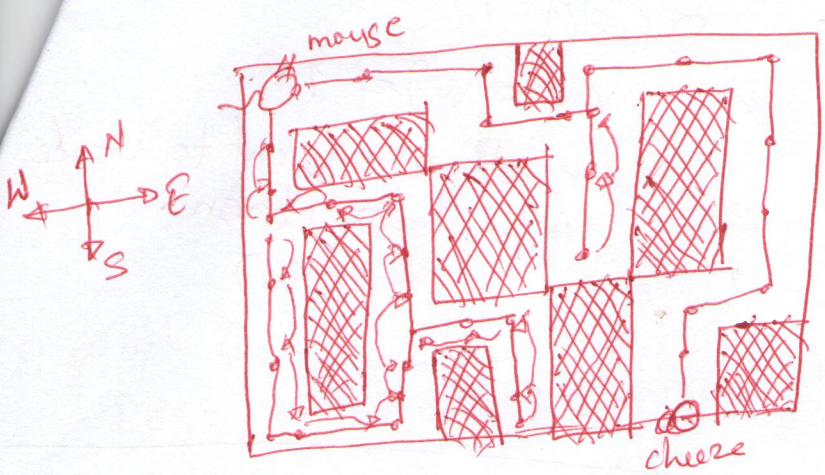
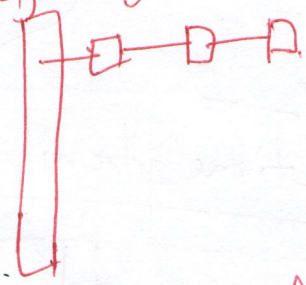
Depth First Search (DFS)



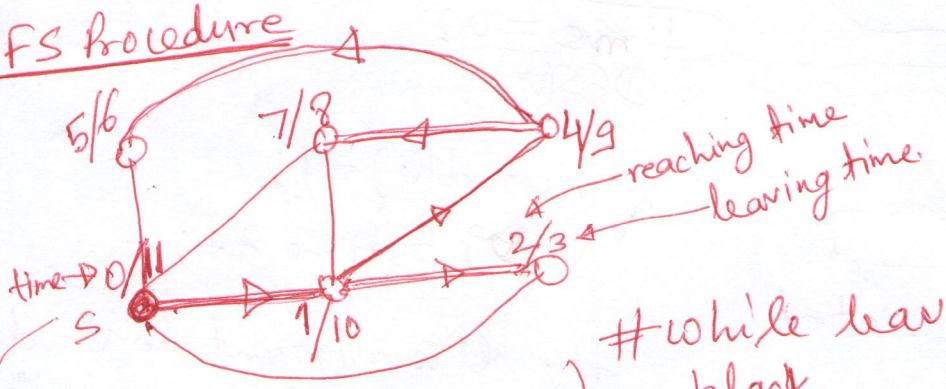
The mouse wants to find the cheeze in the maze.

⇒ DFS way of exploring.

Adjacency List Data Structure.



DFS Procedure



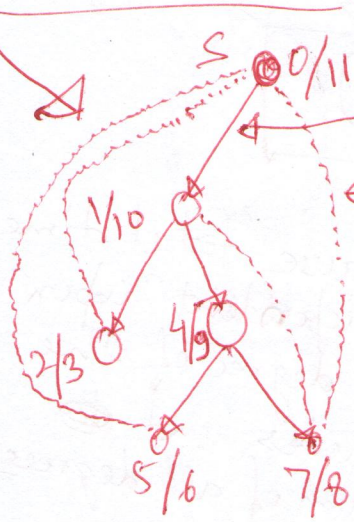
⇒ The no. of edges in DFS Tree is n−1.

↓

form a connected subgraph

while leaving a node, color it black.
Unexplored nodes are white.
Gray nodes are in process and its all children needs to be explored.
If a node is white, then only it will be explored.

⇒ The visited edges form a DFS Tree.

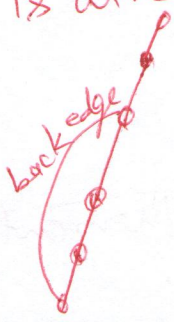


DFS Tree Edges

Other Edges (back edges)

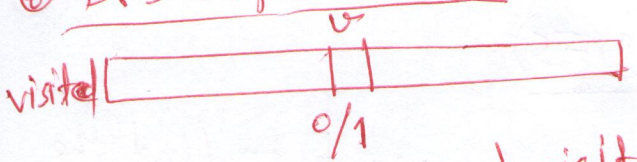
An edge from a node to an ancestor is called the back edges except parent.

going back to a vertex which is already visited.



⇒ DFS will divide all edges into either tree edges or back edges.

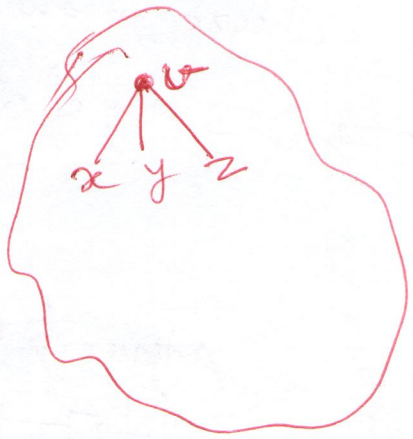
⑧ DFS Implementation



0 means not visited

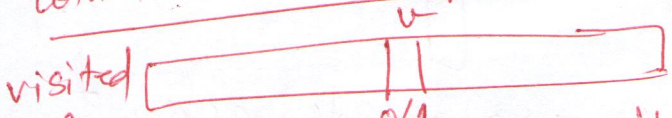
```
DFS(v){
    visited[v] = 1
    for all w adjacent to v do
        if !visited[w] then
            DFS(w)
}
```

```
DFS(v)
    DFS(x)
    :
    DFS(y)
    :
    DFS(z)
    :
Terminate
```




```
time = 0;
DFS(v){
    visited[v] = 1;
    a[v] = time ++;
    for all w adjacent to v do
        if !visited[w] then
            (v,w) is a tree edge
            DFS[w]
    d[v] = time ++;
}
```

with timestamp.



a → arrival time

d → Departure time

0 means not visited

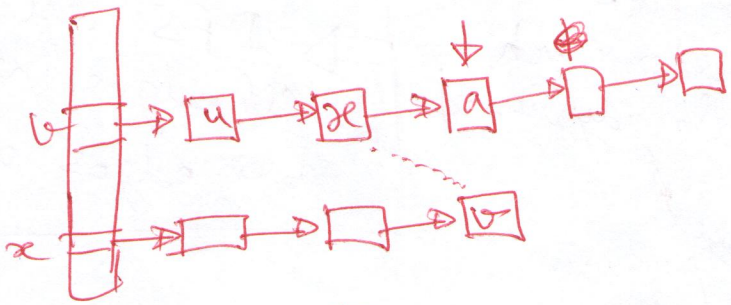
Running Time (DFS)

```
DFS(v){
    visited[v] = 1;
    for all w adjacent to v do
        if !visited[w] then
            DFS(w)
}
```

Assume graph is connected.

Time Complexity: O(m)

Because total time is dependent upon the degree of the nodes and sum of all degrees is equal to 2 time no. of edges



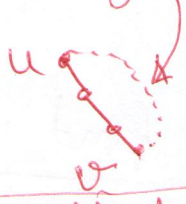
Adjacency list

It can be implemented using stack also.

(u,v) tree edges

$arr[u] < arr[v]$
$dep[u] > dep[v]$

(u,v) back edges

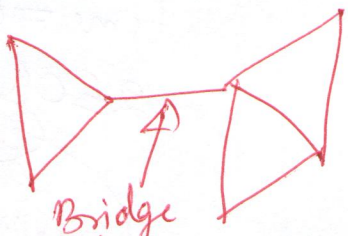


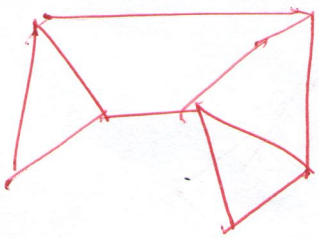
$arr[u] < arr[v]$
$dep(u) > dep[v]$

A graph is 2-Edge Connected ⇒ iff it remains connected after the removal of any one edge.

A graph G is 2-edge connected iff $G - \{e\}$ is connected $\forall e \in E$

$\Downarrow$
(V, E)

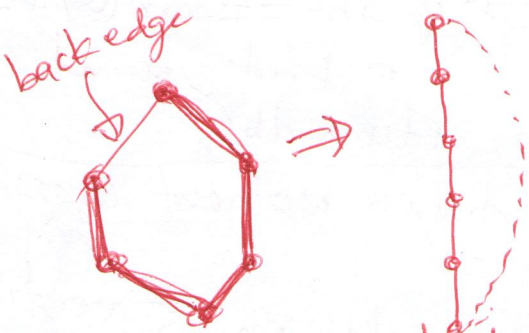




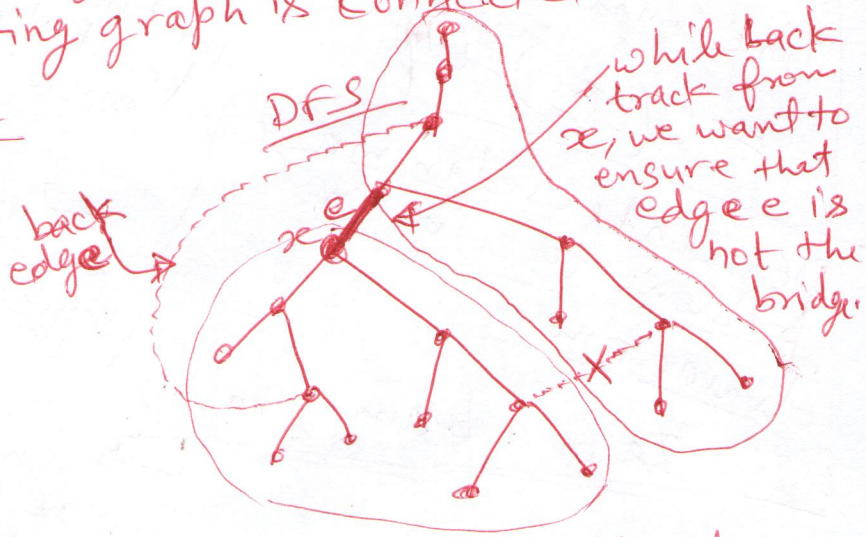
How to check if a given graph is 2-edge connected?

We can do it in $O(m^2)$ by removing every edge & checking if resulting graph is connected.

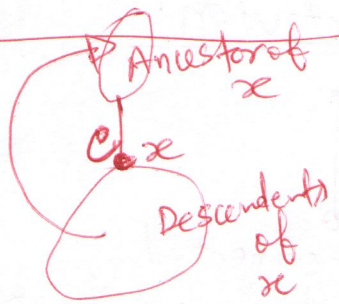
↳ Costly procedure



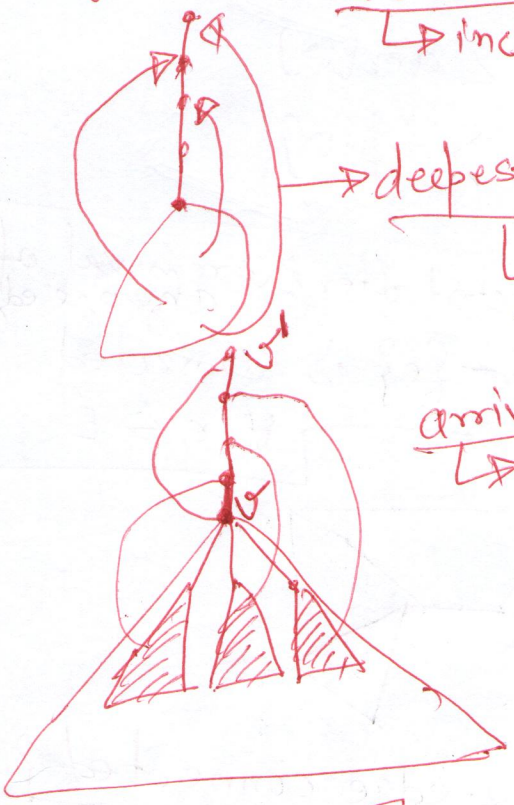
2-edge connected



If there is any back edge from any descendant of x to any ancestor of x, then edge e can not be the bridge edge.



When back tracking from a node v, we need to ensure that there is a back edge from some descendent of v to some ancestor of v.
↳ including node v itself ↳ parent or above.



→ deepest back edge
↳ How to find it?

arrival time of deepest back edge
↳ minimum needs to be traced

deepest back edge

```
time = 0
2EC(v)
dbe = arr[v]
arr(v) = time++
visited(v) = 1
for all w adjacent to v do
    if !visited[w] then
        dbe = min(dbe, 2EC(w))
    else
        dbe = min(dbe, arr[w])
if dbe == arr(v) then
    bridge found.
return dbe;
```

the edge b/w current node and parent should be excluded.

not required for the source vertex.

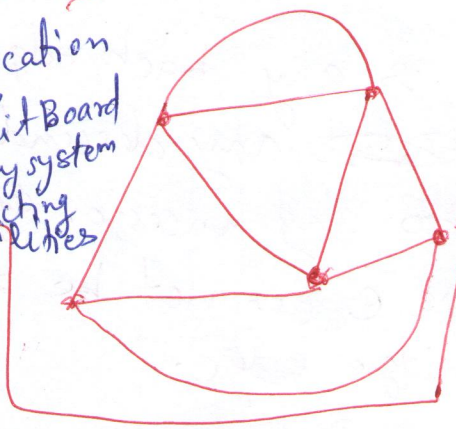
Time Complexity
= O(DFS) = O(m)

DFS can be used in many problems to solve in linear time.

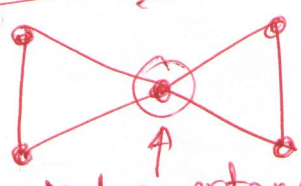
E.g. Is G a planar graph?
↳ Can be drawn without crossing of edges.

⇒ A complete graph on 5 vertices is not a planar graph.

Application
↓
#Circuit Board
#Subway system
#Connecting utilities



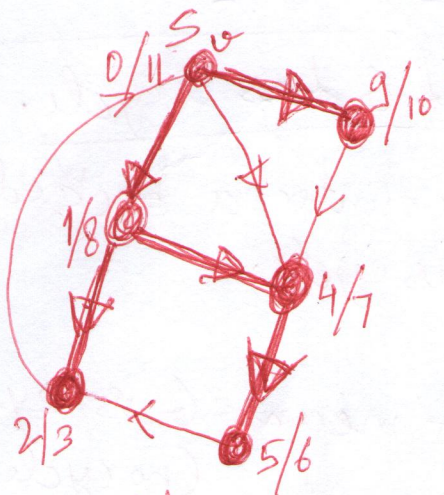
2-Vertex Connectivity.



not 2 vertex connected

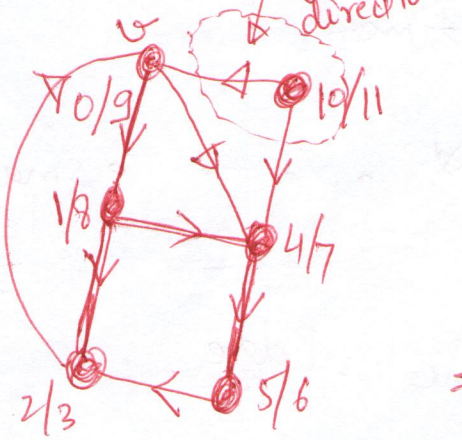
#Can be checked in linear time using DFS.

DFS in Directed Graphs

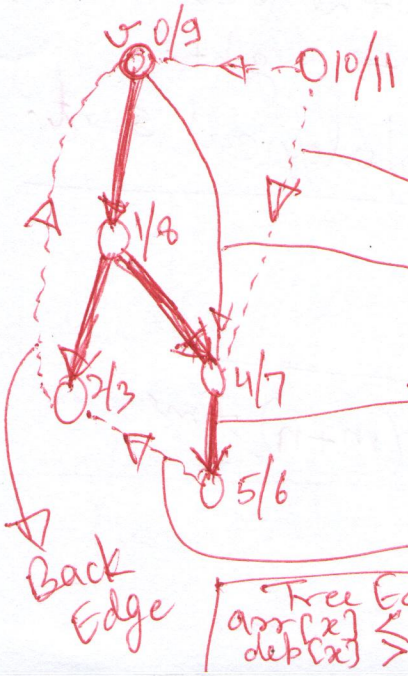


modify

changed the direction

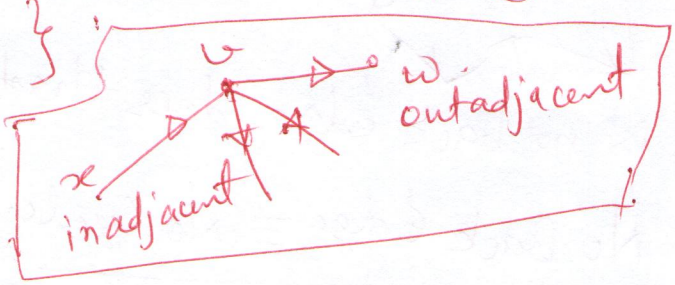


DFS Tree




```
DFS(v){
  visited[v] = 1
  for all w adjacent to v do
    if ! visited[w] then
      DFS(w)
}
```

basically outadjacent



In directed graph, a DFS(v) will visit only those vertices, which can be reached from vertex v.

Thus DFS has to be launched more than once to visit all the nodes.

Similarily, in undirected graph, multiple DFS is needed if graph is not connected.

Basically DFS and BFS should visit all the vertices. If not done in one DFS/BFS, pick the unvisited node and repeat the process until all vertices are visited.

Cross Edge ⟹ No ancestor/descendent Relationship

Forward Edge

Tree Edge

Cross Edge

Tree Edge (x,y)
arr[x] ≤ arr[y]
dep[x] ≥ dep[y]

Forward Edge (x,y)
arr[x] < arr[y]
dep[x] > dep[y]

Back Edge (x,y)
arr[x] > arr[y]
dep[x] < dep[y]

Cross Edge (x,y)
arr[x] > arr[y]
dep[x] > dep[y]

(12)

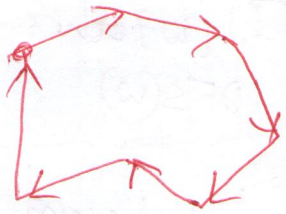
Cross Edge

$(x,y) \Rightarrow arr[y] < dep[y] < arr[x] < dep[x]$

Given a directed graph G, check if G has a cycle.

Do DFS

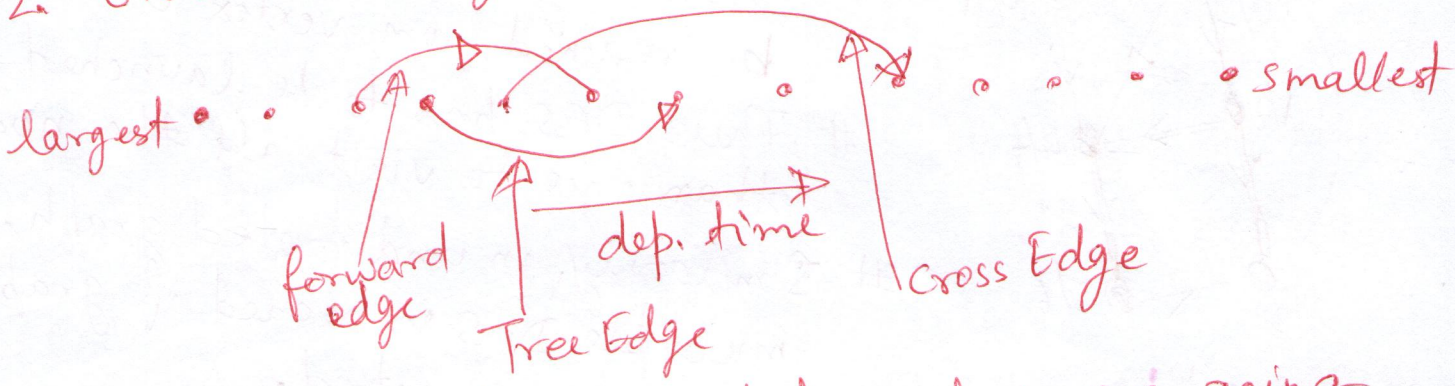
If we encounter a back edge then G has cycle.



⇒ If there is no back edge, does that means G is acylic (no cycle)?

statement : No Back Edge ≡ No Cycle.

1. Do DFS
2. order vertices by their departure times



Because, all forward, cross and tree edges are going from left to right, so cycle can not be possible if back edge is not present.

Given an acyclic graph, we can order the vertices of G so that every edge goes from left to right.

↳ This ordering is also called a topological sort.

→ maximum departure time = 2n-1

It can be computed in O(m+n) time.

A directed acyclic graph is also called a DAG.

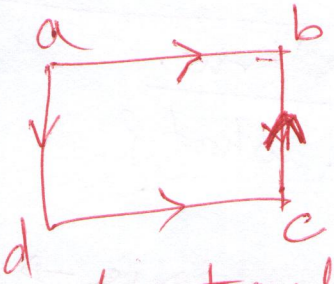
Undirected graphs:

↳ Connected → There is a path b/w every pair of vertices.

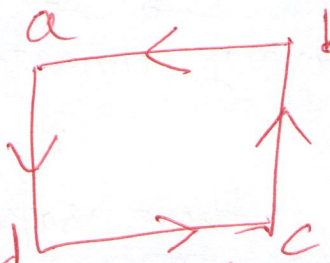
corresponding

Directed graphs:

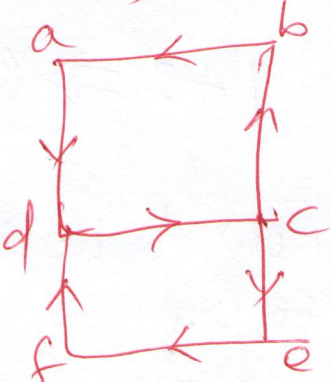
↳ Strongly connected → There is a path b/w every ordered pair of vertices.
(means if a path is b/w (a,b) then there is another path (b,a) also.)



not a strongly connected graph

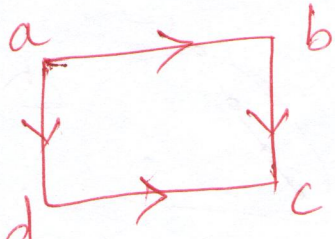


strongly connected graph



strongly connected graph.

Weakly connected Graph

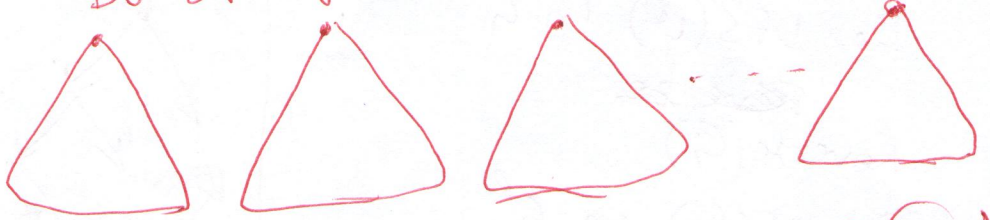


Not a weakly connected graph (there no path from b→d & d→b)

⇒ A graph is weakly connected, if for every pair of vertices (u,v) there is a path from u to v or a path from v to u (or both).

① Given a Graph G, is G strongly connected?

Do DFS for each vertex.



If all these DFS visit all nodes every time, then graph is strongly connected.

⇒ Complexity O(mn)

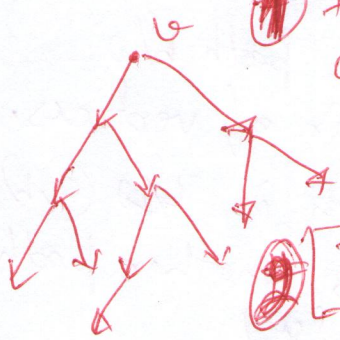
⇒ We want to do in linear time (m+n).

Costly procedure

(14) Applications of DFS in connected Graphs

⇒ Is a given graph G strongly connected?

↳ There should be a path between every pair of ordered vertices.



(1) If DFS(v) visits all vertices in G then ∃ a path from v to every vertex in G.

(2) ∃ (exist) a path from every vertex in G to v.

↳ suppose this is true.

↳ It means what?

1 + 2 ⇒ G is strongly connected.

x, y ⇒ x → v → y (2) (1)

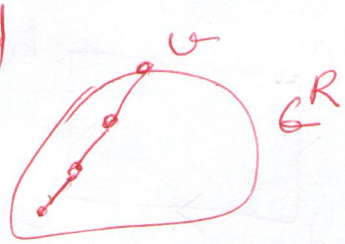
How do we check if there is a path from every vertex to v?

G → (reverse edges) → G^R → Do DFS(v) on G^R

↓

If all vertices are visited

⇐ In G, there is a path from every vertex to v



Complexity = O(DFS)

⇒ Equivalent to DFS in G by considering inadjacent vertices instead of outadjacent vertices.

Procedure

Pick on arbitrary vertex v

Do DFS(v) on G

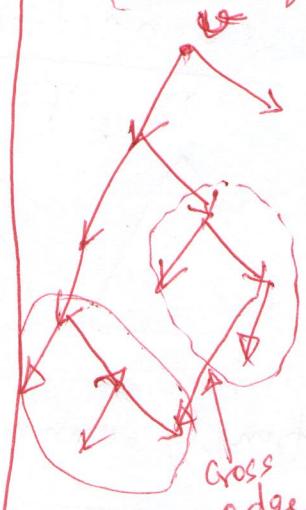
~~Reverse~~

Reverse (G)

Do DFS(v) on G^R

if all vertices are visited in both DFS's then G is strongly connected else G is not strongly connected.

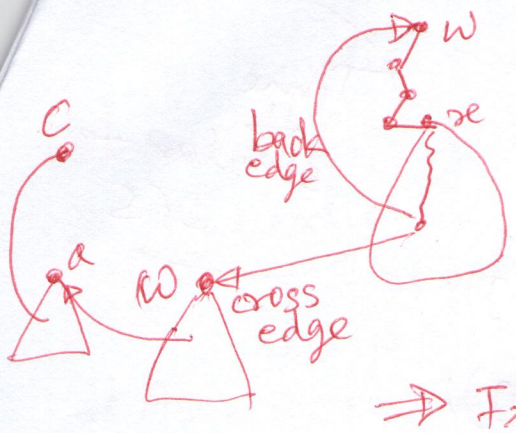
Solving in single DFS



It is better to have the edges outgoing from subtrees.

It is necessary that an edge go out of every subtree.

We want to find a path from every vertex to the root.



$\Rightarrow$ arr[x] > arr[w] > arr[a] > arr[c]

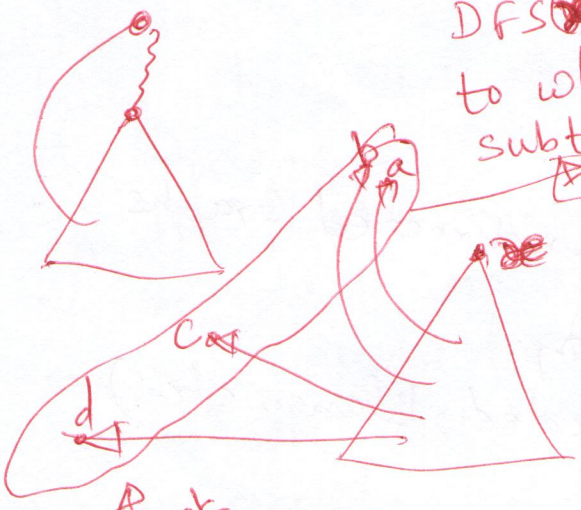
$\Rightarrow$ Thus, by following the combination of back edges and cross edges, If we reach to root v, it means there is a path from x to v.

$\Rightarrow$ It means, it is necessary and sufficient, that an edge go out of every subtree rooted at any vertex x.

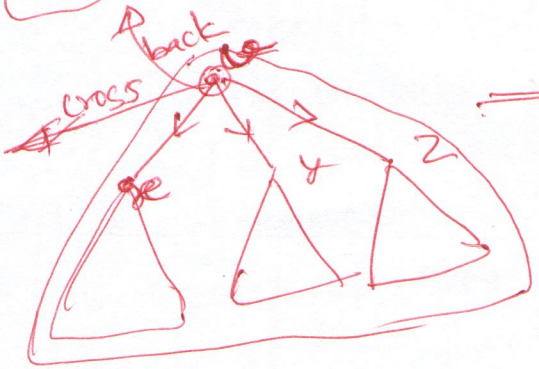
(8) How should we modify DFS so that we can check if there is an edge going out of every subtree?

DFS(x) returns the smallest arrival time to which there is an edge from the subtree rooted at (x).

$\Rightarrow$ smallest arrival time of all theses



DFS(x) returns
min (arr[a], arr[b], arr[c], arr[d])



$\Rightarrow$ DFS(v)

$\Downarrow$

min of all outgoing edge from v, DFS(x), DFS(y) and DFS(z)

If any edge is going out of subtree rooted at v, then arrival time of destination of that edge is less than arrival time of v.

$\Downarrow$

given that DFS(x), DFS(y) and DFS(z) should also go out of subtree rooted at v.

```
SC(v){
    arr[v] = time++
    xeyz = arr[v]
    visited[v] = 1
    for all w outadjacent to v do
        if !visited[w] then
            xeyz = min(xeyz, SC[w]);
        else xeyz = min(xeyz, arr[w]);
    if xeyz = arr[v] then STOP (not strongly connected)
}
```

needs to be checked for root

Time # similar to DFS

Applications of DFS in Directed Graphs
- Strongly connected
- Acyclic
- Topological sort

Applications of DFS in Undirected Graphs
- 2-edge connectivity
- others (vertex connected, planar etc.)

Applications of BFS
- Connected components
- Bipartite