Depth first Search

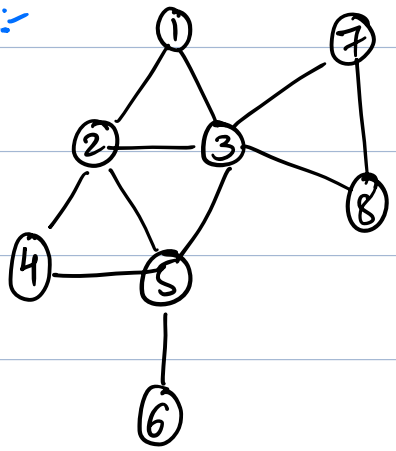
Idea:

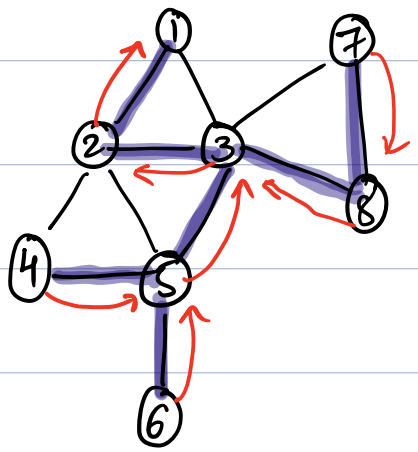
→ To search deeper in the graph whenever Possible.

The Algorithm Starts from a node s and try the first edge leading out of it to a node v. Then it follow the first edge leading out of v and so on untill it reaches a "dead end" (a node for which all neighbors are explored).

Then it backtracks untill it finds a node with unexplored neighbor and resumes from there.

Example:-






```
DFS(u)
    mark u as "explored"
    For each edge uv incident to u
        if v is not marked "explored" then
            Recursively invoke DFS(v)
        Endif
    Endfor
```

For each node u we have

$u.color$ – color of u

$u.\pi$ – predecessor of u

$u.d$ – discovered time

$u.f$ – finishing time

DFS timestamps each vertex. Each vertex v has two timestamps.

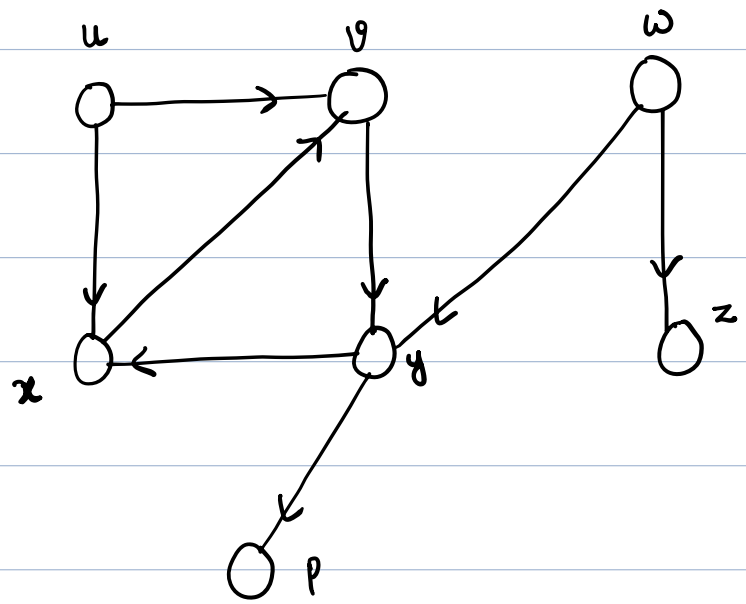
① The first timestamp $v.d$ records when v is first discovered (grayed)

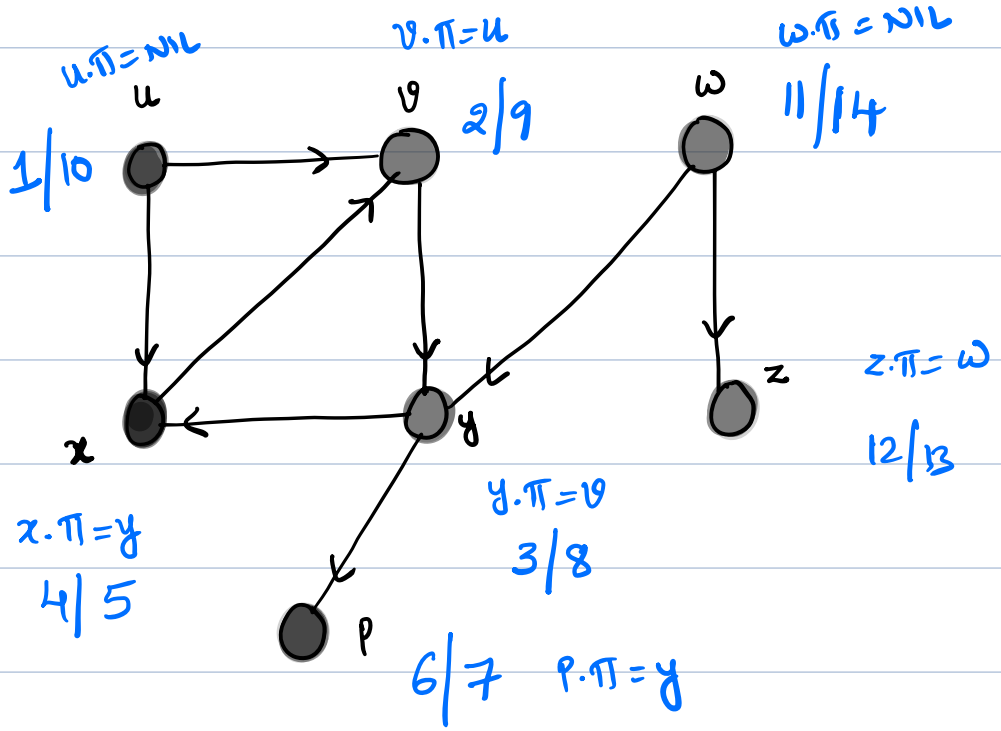
② The second timestamp $v.f$ record when the search finishes examining v's adjacency list.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1          // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]    // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK          // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

Example:


u
v
w
x
y
z
p


u.π = NIL
u
1/10
v.π = u
v
2/9
w.π = NIL
w
11/14
x
x.π = y
4/5
y
y.π = v
3/8
z
z.π = w
12/13
p
6/7
p.π = y

Depth first search tree | forest

$$G_\pi = (V, E_\pi)$$

where $E_\pi = \{ (v.\pi, v) \mid v \in V \text{ and } v.\pi \neq NIL \}$

Classification of edges

DFS can be used to classify the edges of the input graph.

The type of each edge gives some information about the structure of the graph. (use full in some applications)

Based on DFS forest G_π we can define four edge types. (See the next Page)

1. ***Tree edges*** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
3. ***Forward edges*** are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

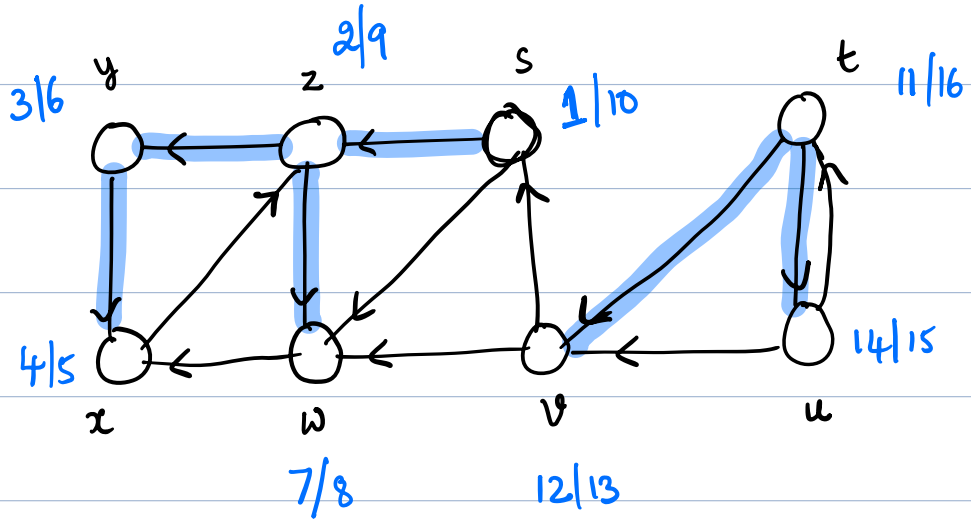
When we first explore an edge uv the color of vertex v tells us something about the edge.

White indicates tree edge

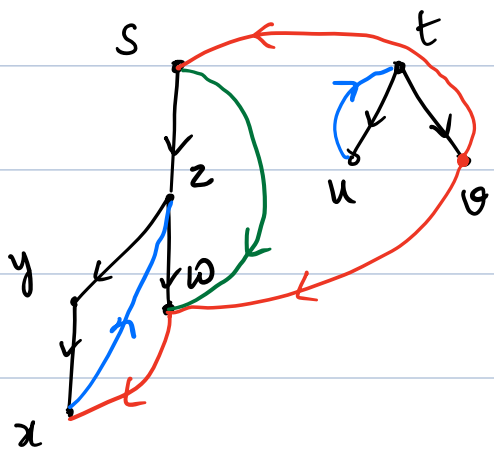
Gray " back edge

Black " forward / Cross edge.

Example:



DFS forest:



Tree edges - black colored

Back edges - blue "

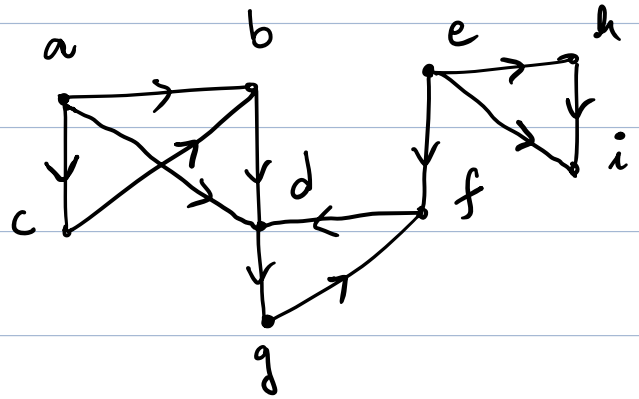
Cross edges - red "

forward edges - green "

$uv \in E(G)$

FORWAD EDGE	–	$u.d < v.d$ & $u.f > v.f$
Back EDGE	–	$u.d > v.d$ and $u.f < v.f$
Tree edge	–	$u.d < v.d$ & $u.f > v.f$
Cross edge	–	$v.d < v.f < u.d < u.f$

Exercise:



(A) Perform a depth first search starting from node 'a' with Preference for visiting lower-character vertices before higher-character vertices.

(B) Write down the discover and finish times for each node

(C) Also classify the edges.

Theorem: In a DFS search of an undirected graph G, every edge of G is either a tree edge or a back edge.

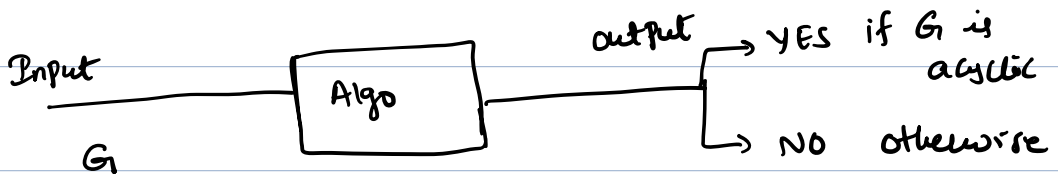
Proof Exercise.

Applications

Q1) How to test whether an connected undirected graph has a cycle?

Q2) How to test whether a directed graph has a cycle?

Check if a directed graph is acyclic.



Lemma: A directed graph is acyclic if and only if a DFS of G yields no back edges.

[Coreman : Lemma 22.11]

Proof Idea:

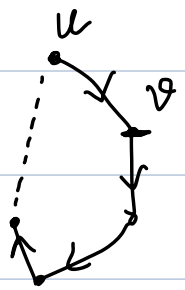
[Forward]: Suppose DFS Produces a back edge uv then v is ancestor of u in DFS forest.

Then the Path from v to u along with the edge uv forms a cycle.

[Backward]: Suppose that G has a cycle C.

Let v be the first vertex to be discovered in C.

and uv be its Preceding edge.



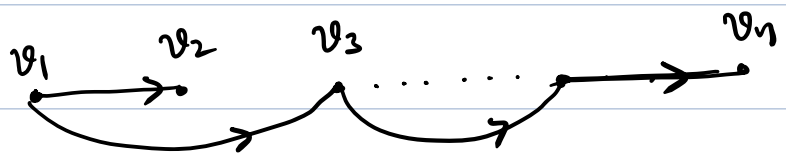
Then at time v.d none of the other vertices of C are not discovered (ie, their color is white)

ie, u becomes a decendent of v in DFS forest.

∴ uv is a backedge.

Topological Sort

A topological sort of a DAG $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge uv, then u appears before v in the ordering.



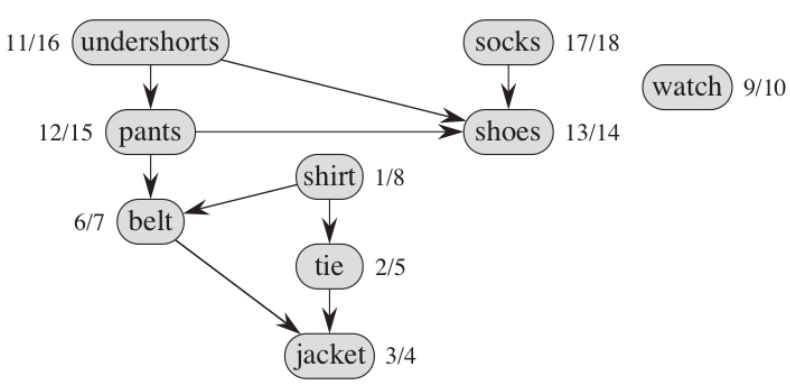
All the edges go from left to right.

Motivation :

Suppose that you need to perform many tasks, but some of them cannot begin until certain others are completed.

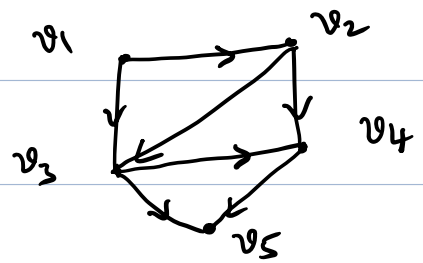
We can model this using directed graphs, where each task is a node and there is an edge from u to v if u is a precondition for v.

ie, before performing a task, all the tasks pointing to it must be completed.

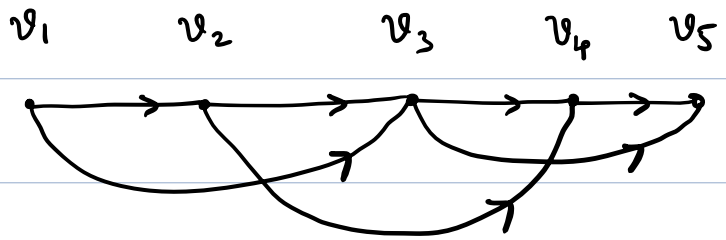


Ref: Coremen

Example: (1)



DAG G



Topological sort of G.

Recap: A directed graph G acyclic if DFS of G yields no back edges.

Algorithm

TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

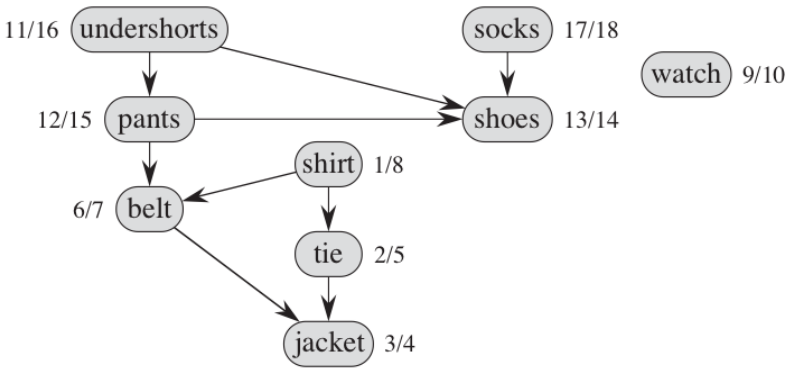
Running time: $O(V+E)$

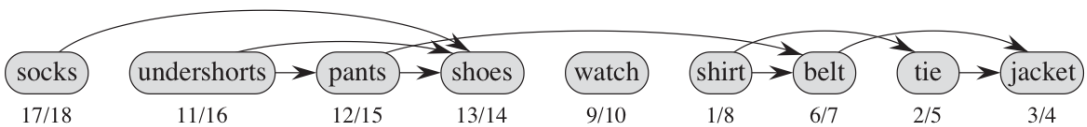
we get the ordering of the vertices of G

$v_1 , v_2 , \ldots v_n$

such that $v_i.f > v_j.f$ if $j > i$

Example:


11/16
undershorts
socks
17/18
watch
9/10
12/15
pants
shoes
13/14
shirt
1/8
6/7
belt
tie
2/5
jacket
3/4


socks
17/18
undershorts
11/16
pants
12/15
shoes
13/14
watch
9/10
shirt
1/8
belt
6/7
tie
2/5
jacket
3/4

Correctness Proof:

We need to show that the algorithm produces a topological sort of the DAG given as its input.

We need to show that for any pair of distinct vertices $u, v \in V$, if $uv \in E(G)$ then $v.f < u.f$.

Consider any edge uv explored by DFS(G)

(I) if v is gray

then v would be an ancestor of u and uv would be a back edge

Contradicting that G is a DAG.

(II) If v is white.

it becomes decendent of u so $v.f < u.f$

(III) If v is black

then it has already been finished.

So $v.f$ has already set, and we are yet to assign a time stamp $u.f$ as we are still exploring from u.

$\therefore \quad v.f < u.f$

Hence for any edge $uv \in E(G)$

we have $v.f < u.f$

Connectivity in Directed Graphs

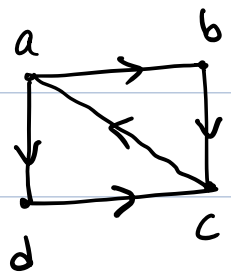
Recap:

A graph is connected if every vertex is reachable from all other vertices.

A directed graph is **Strongly Connected** if every two vertices are reachable from each other.

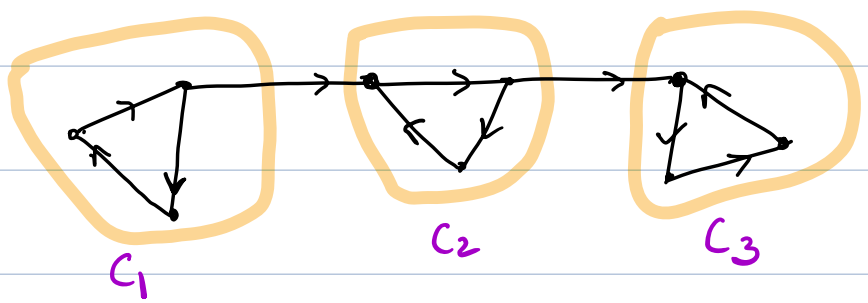
A **Strongly Connected Component** of a directed graph G is a subgraph that is Strongly Connected and is maximal (no additional edges or vertices can be added to subgraph without voilating the Property)

Example ①:



Strongly Connected graph.

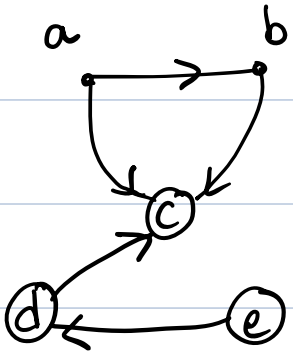
Example ② Graph with three Connected Components.



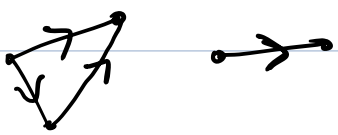
A directed graph is **Weakly Connected** if every two vertices are reachable from each other ignoring the directions of edges.

A **Weakly Connected Component** of a directed graph G is a subgraph that is weakly connected.

Example:



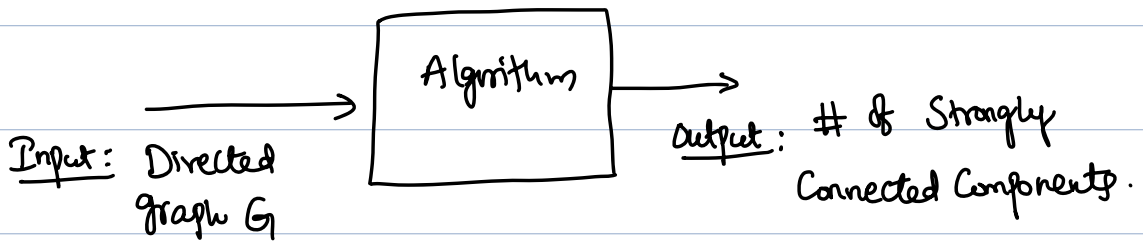
Weakly Connected.



Two weakly Connected Components.

Goal:

Use the DFS to find Strongly Connected Components in a directed graph.



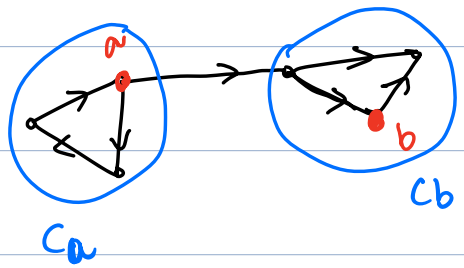
Warmup - Ideas

True | False :

Let G be a directed graph having Strongly Connected components $C_1, C_2, \ldots C_k$.

If we apply DFS from a node v in C_i then it only vistis vertices in C_i (I mean then DFS tree obtained only consists vertices from C_i)

Warmup - Ideas



→ if we start from node b (DFS)
then we explore only C_b

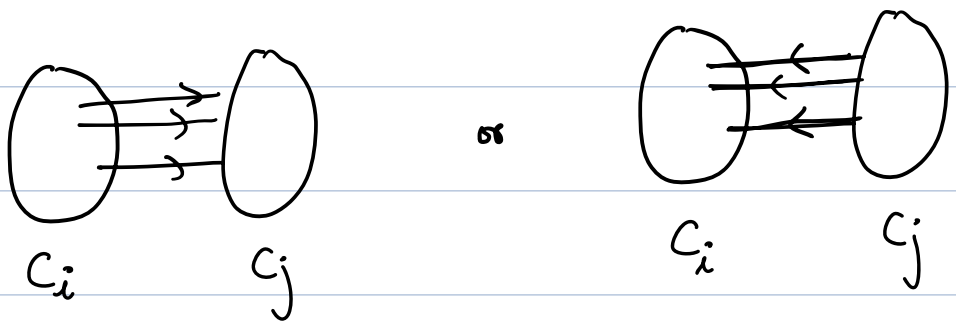
→ but we start (DFS) from a
then we explore $C_a \cup C_b$

Moral: If we call the DFS from the right place then we explore only required SCC.

True / False :

Let G be a directed graph having Strongly Connected components $C_1, C_2, \ldots C_K$. Then between any two distinct SCCs say C_i & C_j, all the edges goes in one direction.

ie,



Transpose of G:

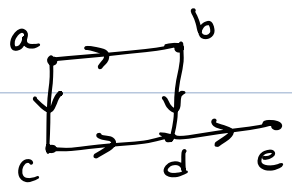
The transpose of a graph G is $G^T = (V, E^T)$

where

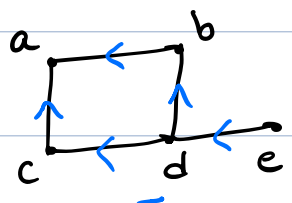
$$E^T = \{ uv \mid vu \in E(G) \}$$

ie, E^T consists of the edges of G with directions reversed.

Ex:



G



G^T

True | False

Both G and G^T have exactly the same SCCs.

ie, if u & v reachable from eachother in G if and only if they are reachable from each other in G^T

Algorithm: [Kosaraju]

STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

Running time: 2 * DFS running time = $O(V+E)$

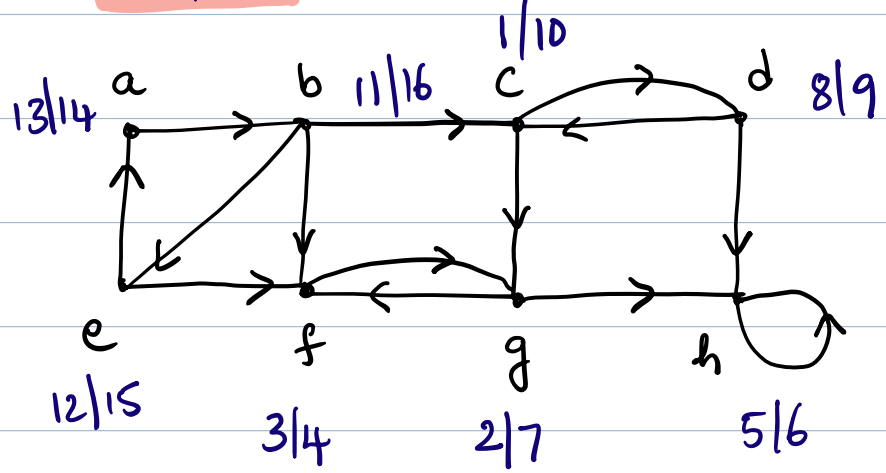
Remark: Because the above algorithm performs two depth-first searches, there is the potential ambiguity when we discuss $u.d$ or $u.f$.

In this section, these values always refer to the discovery and finishing times as computed by the first call of DFS in line 1.

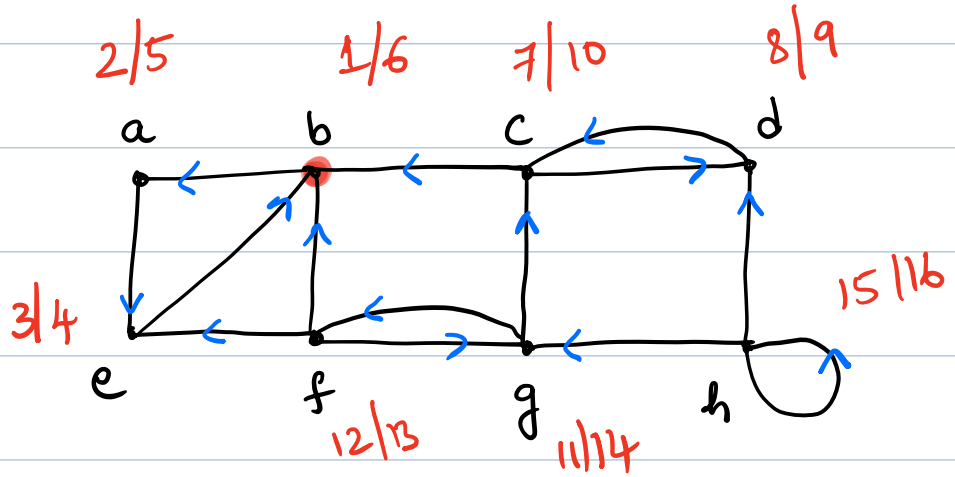
In next we explain the algorithm with an example.

Example

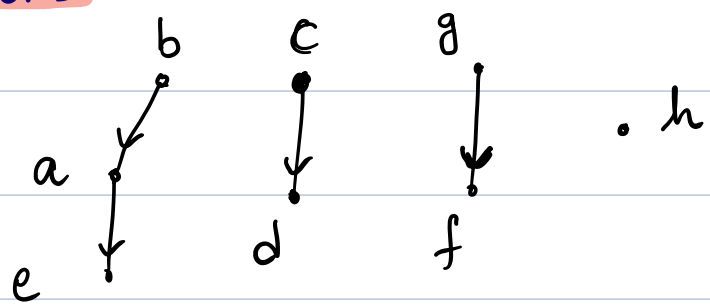
Graph G



Graph G^T



DFS forest



Four Connected Components.

Proof of Correctness:

(Main Idea)

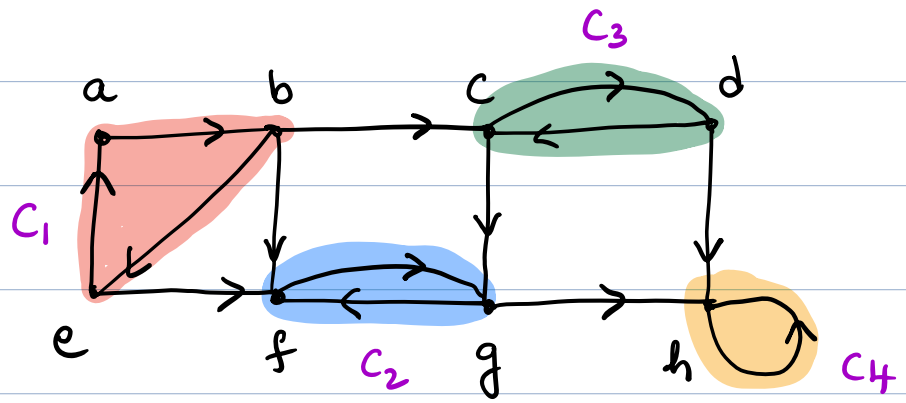
Construct a Component graph $G^{SCC} = (V^{SCC}, E^{SCC})$

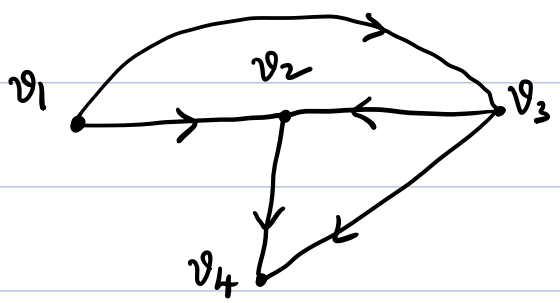
Suppose that G has Strongly Connected Components $C_1, .. C_k$.

The Vertex set V^{SCC} is $\{v_1, v_2, .. v_k\}$

ie, it contains one vertex v_i for each C_i

There is an edge $v_i v_j \in E^{SCC}$ if G Contains a directed edge xy for some $x \in C_i$ and y_i in C_j.



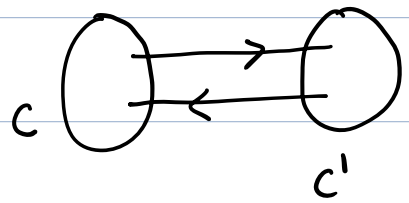


Graph G^{scc}

True/False:

The graph G^{SCC} is acyclic.

Ans: True



then $C \cup C'$ is SCC contradicting our assumption that C & C' are distinct SCC's

Def: $G = (V, E)$

If $U \subseteq V$ then we define

$$d(U) = \min_{u \in U} \{u.d\}$$

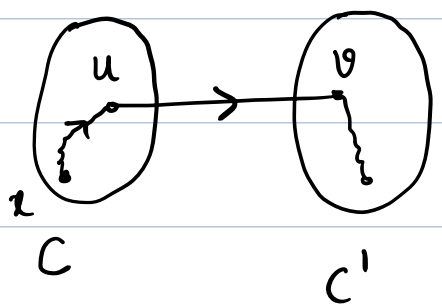
$$f(U) = \max_{u \in U} \{u.f\}$$

That is $d(U)$ and $f(U)$ are the earliest discovery time and latest finishing time respectively of any vertex in U.

Lemma:

Let C and C' be distinct Strongly Connected Components in directed graph $G = (V, E)$. Suppose that there is an edge $uv \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$

Proof idea:

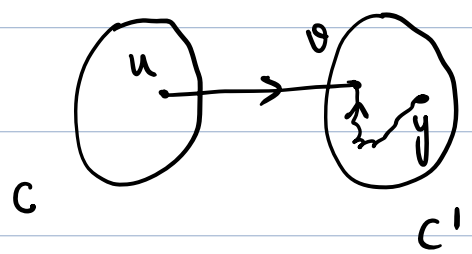


Case 1: $d(C) < d(C')$

Let x be the first vertex discovered in C. At time $x.d$ all vertices in C and C' are white. At that time G contains a Path from x to each vertex in $C \cup C'$.

∴ all vertices in $C \cup C'$ becomes decendents of x in DFS tree. ∴ $f(C) > f(C')$

<u>case 2</u>: $d(C) > d(C')$



let y be the first vertex discovered in C'.

At time $y.d$ all vertices in C' are white and G_1 contains a path from y to each vertex in C' consisting only white vertices.

So all vertices in C' are decendents of y in DFS tree. $y.f = f(C')$.

At time $y.d$ all vertices of C are white and there is no path from C' to C.

Hence, no vertex in C is reachable from y.

$\therefore$ At time $y.f$, all vertices in C are still white.

$\therefore$ $\forall w \in C$, $w.f > y.f$

$\Rightarrow$ $f(C) > f(C')$.

Corollary: Let C and C' be distinct connected components in directed graph $G = (V, E)$.

Suppose that there is an edge $uv \in E^T$,

where $u \in C$ and $v \in C'$ then $f(C) < f(C')$

Summary of the Algorithm

In the 2nd DFS, which is on G^T, we start with the SCC C whose finishing time $f(C)$ is maximum.

The search starts in C at some vertex x and it visits all vertices in C.

By the above corollary G^T contains no edges from C to any other SCC.

∴ The tree rooted at x contains exactly the vertices of C.

Next search selects as a root a vertex from some other SCC C' whose finishing time $f(C')$ is maximum over all components other than C.

and so on.

Cut Vertices & Cut-edges

Articulation Points — bridges

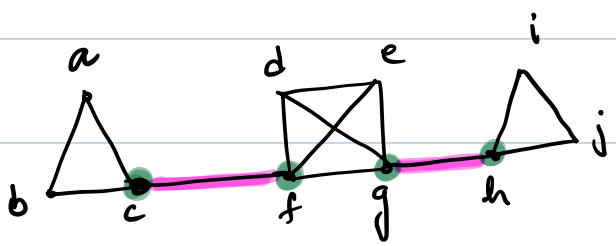
Let $G = (V, E)$ be a connected, undirected graph.

Def:-

A cut vertex of G is a vertex whose removal disconnects G.

A cut edge of G is an edge whose removal disconnects G.

Example:



cut vertices = c, f, g, h

cut edges = cf, gh

Problem:

Design an algorithm to find

(a) Cut vertices in a graph [Homework]

(b) Cut edges in a graph [Now]

Warmup:

Easy but not so efficient algorithm:

(a) Try every vertex of G and then use DFS/BFS

Check the Connectivity of G.

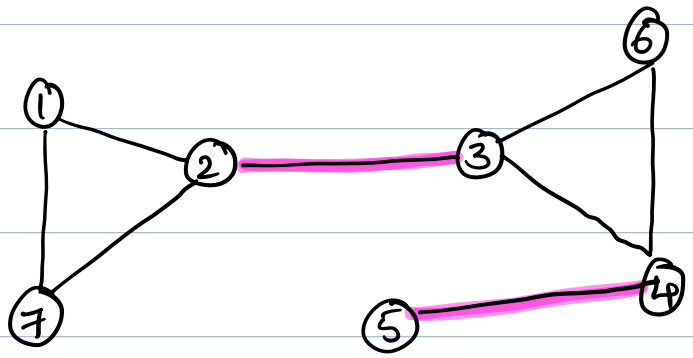
(b) Try every edge of G and then use DFS/BFS

Check the Connectivity of G.

Running time: $O(V(V+E)) = O(V^2 + VE)$

$= O(VE)$ as G is Connected $|E| \geq |V|$

Algorithm to find all cut edges in a graph



Warmup:

Easy but not so efficient algorithm:

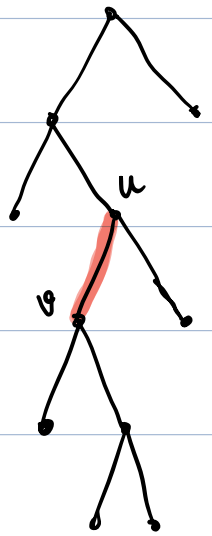
Try every edge of G and then use DFS/BFS

Check the Connectivity of G.

Running time: $O(V(V+E)) = O(V^2 + VE)$

$= O(VE)$ as G is connected $|E| \geq |V|$

Look at the DFS tree:



Q When uv is NOT a cut edge?

Ans: If there is a back edge from some decendent of v (including v) to some ancestor (Proper) of v.

(or)

A tree edge uv with u as v's Parent is a cut-edge if and only if there are no edges in v's subtree that goes to u or higher.

Q Do we need to keep track of all back edges?

Ans: For each node u.

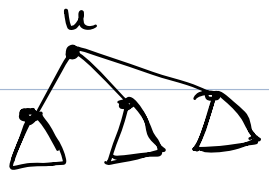
We keep track of the highest node x such that there is a back edge from a descendant of u upto x.

Here the "highest" node is the one with the lowest discover time.

$$h[u] = \min\{ x.d \mid w \text{ descendant of } u \text{ and exists a backedge from } w \text{ up to } x \}$$

We set $h[u] = \infty$ if there is no backedge from any descendent of u.

We can define h recursively as follows.



$$h[u] = \min\Big(\{ h[w] \mid w \text{ is a child of } u \text{ in DFS tree} \} \cup \{ x.d \mid \text{exists back edge from } u \text{ up to } x \} \Big)$$

Pseudocode

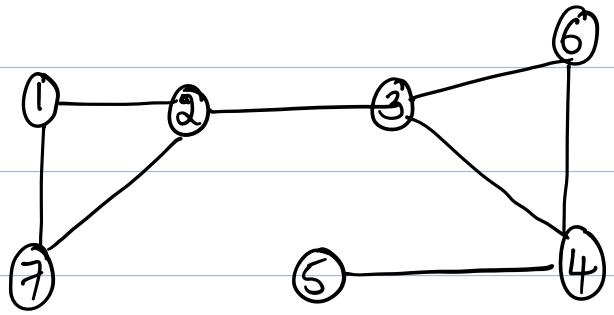
Running time:

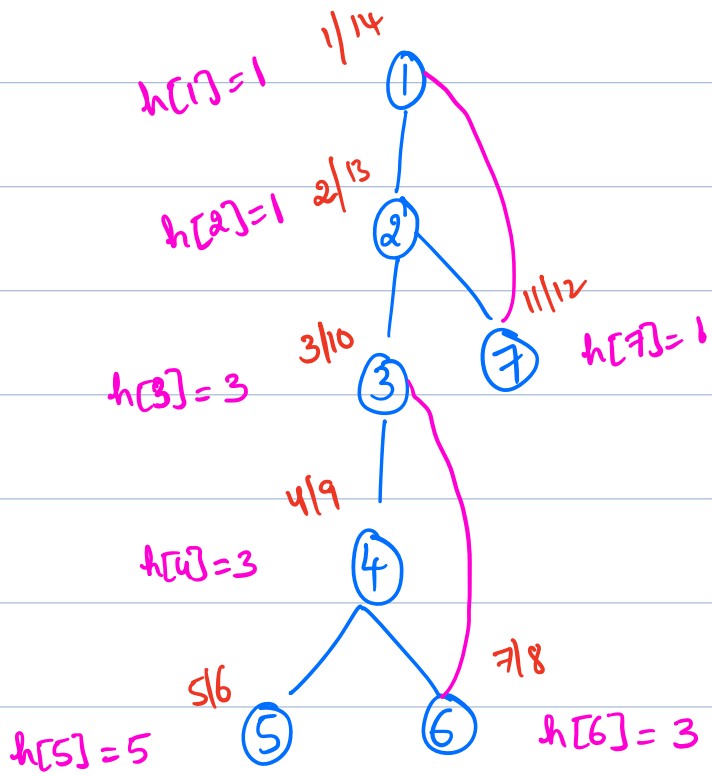
$O(m+n)$

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time , h[u] = u.d
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8      h[u] = min{h[u], h[v]}
9      If h[u] > u.d
10         then uv is a cut edge
11     else
12         h[u] = min{h[u], v.d}
13 u.color = black
14 time = time + 1
15 u.f = time
```

Example





$h[u] = d[u]$

The tree edge from u to w is a bridge

if $h[w] > u.d$

Cut vertices (or) Articulation Points

Recap: A cut vertex is a vertex whose removal disconnects G.

Remark:

It is easy to see that the root of the DFS tree T is a cut vertex of G if and only if it has at least two children in T.

From the above remark, by running DFS once for each node (as a source) we can find all cut vertices of G in $O(|V||E|)$ time.

But we would like to develop an algorithm with running time

Lemma: let v be a nonroot vertex of DFS tree T. v is a cut vertex if and only if v has a child w such that there is no back edge from w or any descendant of w to proper ancestor of v.

Proof: If v has a child w such that neither w nor any of w's descendants have a back edge to a proper ancestor of v.

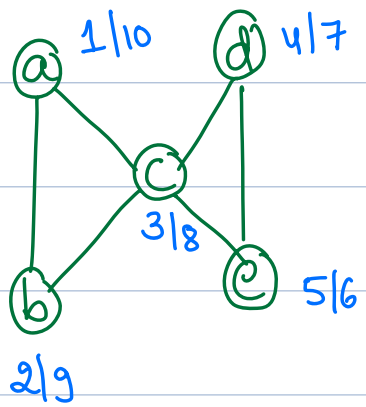
Then root of the DFS tree is not reachable from w, so the graph will be disconnected if v is removed & hence v is a cut vertex.

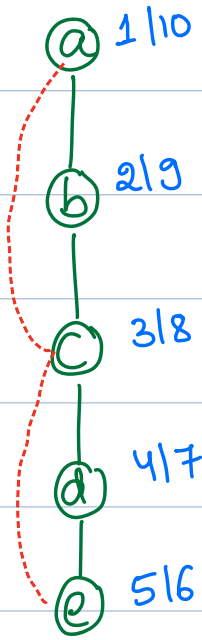
If no such w exists, then all children of u can reach up beyond u and hence removing u will not disconnect G.

For every node u, we store the following information

$$h[u] = \min \begin{cases} u.d, \\ w.d : (v,w) \text{ is a back edge for some descendant } v \text{ of } u \end{cases}$$

Example.





DFS-tree T

u	a	b	c	d	e
h[u]	1	1	1	3	3

To see why $h[d]=3$, observe that node d has a decendant e, such that there is a back edge from e to c.

" A vertex u is a cutvertex if and only if u has a child w with $h[w] \geq u.d$

In the above example

- Vertex b is not a cut vertex, since its only child satisfies $h[c] = 1 < 2 = b.d$

- Vertex c is a cut vertex, since it has a child d with $h[d] = 3 = c.d$