Depth First Search

```python
def DFS(v):
    visited.add(v)
    previsit(v)

    for w in v.adj:
        if w not in visited:
            w.parent = v
            DFS(w)
    postvisit(v)

def DFSAll(G):
    for v in G:
        if v not in visited:
            DFS(v)
```

`pre-order = [I]
def previsit(v):
    preorder.append(v)`
Edge \( (u \rightarrow v) \in G \) can be:

1. Tree edge (green):
   \[ u.pre < v.pre < v.post < u.post \]
   & DFS calls \( v \) from \( u \) directly

2. Forward edges (blue):
   other edges \( w/ \)
   \[ u.pre < v.pre < v.post < u.post \]
   [Note: \( u \rightarrow v \) along tree edges]

3. Backward edges (red):
   \[ v.pre < u.pre < u.post < v.post \]
   [Note: back + tree forms a cycle, \( u \rightarrow v \rightarrow u \) path along tree edges]

4. Cross edges (gray):
   \[ v.pre < u.post < u.pre < u.post \]
   [\( u \) would call \( v \)]
Lemma: \( \forall u, v \)

\[
\text{u.post} < \text{v.pre} < \text{v.post} < \text{u.post}
\]

\[\Rightarrow \] \( u \) is an ancestor of \( v \)

in the DFS Forest

1) Detect cycles

\[\text{[if cycle exists, find it]}\]

Lemma: \( G \) has a cycle \( \Rightarrow \)

\[\exists \] a backward edge

PF: Consider the Reverse Postorder

of \( G \).

\( (u \text{ before } v \Rightarrow u.\text{post} > v.\text{post}) \)

Green, blue, gray all point Forward

no red \( \Rightarrow \) no cycle.

on the other hand

\[\exists \] red edge \( u \rightarrow v \),

when you visit \( u \),

\( v.\text{pre} < \text{current time} < v.\text{post} \)

\[\Rightarrow \] stack contains \( v \) \& \( u \) path

\[\Rightarrow \] that \( + (u \rightarrow v) \) is a cycle.
(2) **topological sort of DAG**

find an order s.t.
all edges go forward

reverse post-order

(3) **implicit topological orderings**

visiting vertices in postorder

⇒ visiting in reverse topological order.

**Example** longest (s -> t) path in DAG.

```python
def longest (v, t):
    if v == t:
        return 0
    if v in memo:
        return memo[v]

    memo[v] = max (longest (u, t) + dist(v->u),
                   for u in u[0])

    return memo[v]
```
Strong Connectivity

U & V are \textbf{Strongly Connected} if:
- U can reach V
- V can reach U.

Form \textbf{Strongly Connected Components}:

1. \textbf{Strong Component Graph} \( SCC(G) \)

which is a DAG

where vertices of \( SCC(G) \)

correspond to strong components of \( G \).

Q: \textbf{How to find Strong Components?}

(a) For fixed \( v \), find its strong component:

- For all \( u \), see if
  - \( v \) reaches \( u \), whether
  - \( u \) reaches \( v \) - 

\( O(V+|E|) \)

- if \( u \) other \( u \), \( E \) time for \( u \) to search.

\( \text{Reach}(v) = \text{set of } u \text{ reachable in } O(|E|) \text{ time} \)

\( \text{reach}^{-1}(v) = \text{set of } u \text{ reachable on reversed graph} \)

\( \text{answer} = \text{reach}(v) \cup \text{reach}^{-1}(v) \)
(b) Find all SCC

- Run (a) on any vertex
  - You don't know component for
  - repeat
    \( O(VE) \)

- Clearly: \( O(VE + V) \)

Note: Each strongly connected

reverse runs \( (b) \) DFS tree) not in component

SCC alg

1) Find some vertex
   in a single component

2) Its component is anything
   it can reach

3) Remove & repeat

\( O(E) \)

Total

Kosaraju:

The last vertex in a postorder
is in a source SCC.

\( \Rightarrow \) last vertex in postorder of
\( Rev(G) \) is in a sink SCC.