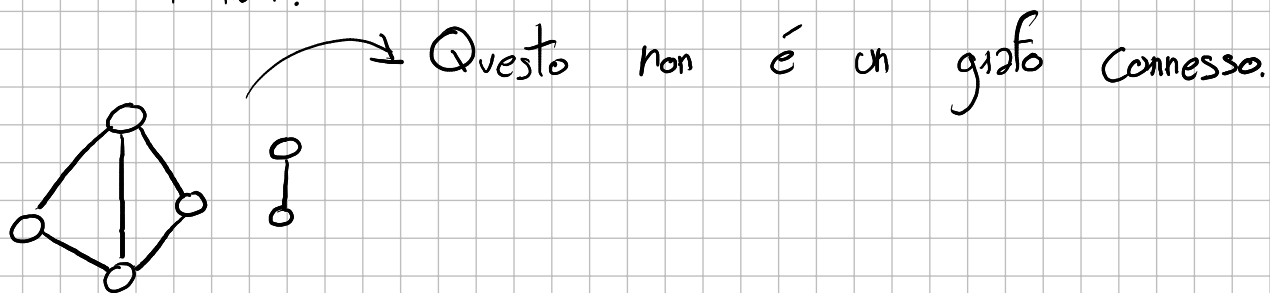


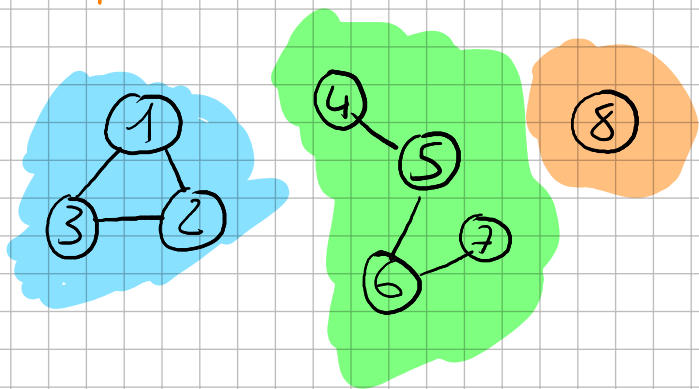
## Grafo connesso:

Un grafo si dice connesso quando da un nodo posso raggiungere tutti i nodi.



Per determinare se un grafo è connesso basta che faccio una visita (DFS o BFS, poco importa) e verifico se sono stati scoperti tutti i nodi.

## Componente connessa (Solo per grafi non orientati)



Questo grafo è composto da 3 componenti connesse.

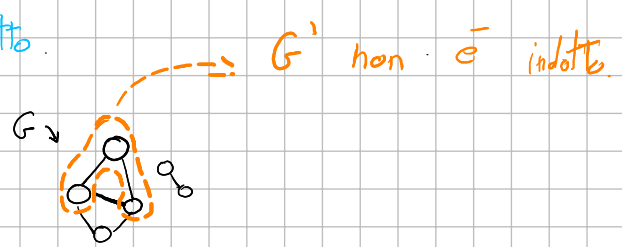
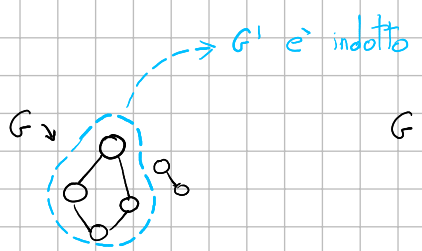
## Definizione formale:

Una componente connessa è un grafo indotto e massimale

Cerchiamo di capire cosa significa:

- grafo indotto: quando in un sottografo non posso più aggiungere ulteriori archi.

Es:



Def. Formale:

dato  $G=(V,E)$ ,  $E \subseteq V \times V$

se  $G'=(V',E')$  con  $G' \subseteq G$ ,

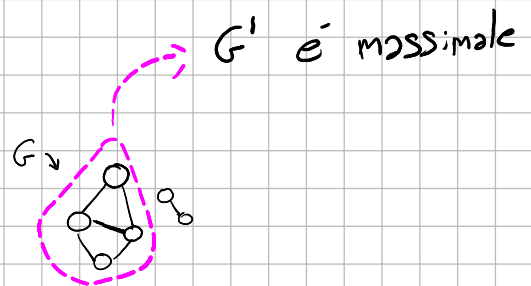
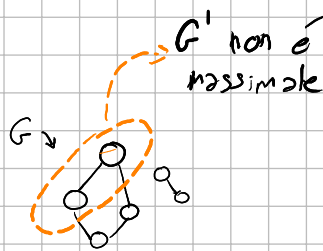
$V' \subseteq V$  e  $E' = E \cap (V' \times V')$

allora ho un sottografo indotto.

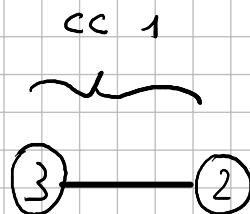
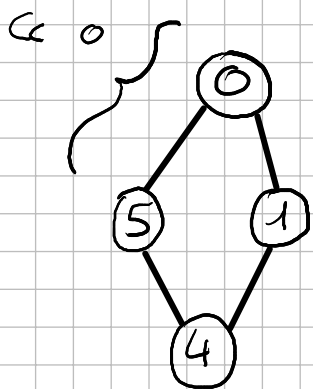
$\Rightarrow$  (dal grafo di partenza)

- grafo massimale: quando non posso aggiungere altri nodi, senza  
non ho un grafo connesso

Es:



Array di component: connesse (CC):



6

cc 2

$cc[i] = \text{num della cc del nodo } i$

$cc[0] = 0$

$cc[4] = 0$

$cc[6] = 2$

$cc[3] = 1$

Implementazione dell' array delle cc.

Puoi usare una qualsiasi visita.

Es con BFS

```
get cc () {
```

```
    count = 0 // var d'istanza
```

```
    For (i = 0 to n)
```

```
        if (!scoperto(i)) {
```

```
            visita BFS (i)
```

```
            count ++;
```

```
        }
```

```
    }
```

```
visita BFS (sorg) {
```

```
    S = S ∪ {sorg}
```

```
    ⋮
```

```
    while coda non vuota {
```

```
        cc[node] = count
```

```
        ⋮
```

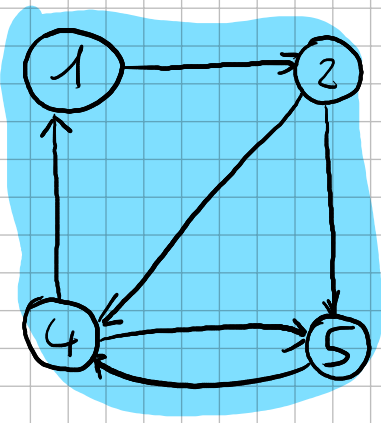
```
    }
```

```
}
```

Tutti i discorsi fatti fino adesso valgono esclusivamente per i grafi non orientati.

---

Component: Fortemente connesse (Solo per graf: orientati)  
aka. SCC



In questo grafo sono  
presenti 3 SCC

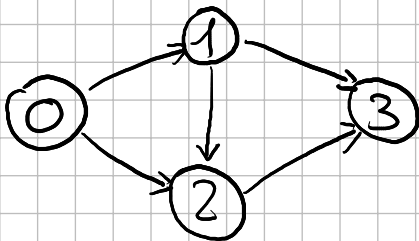
Definizione:

Se si può raggiungere ogni nodo da un altro

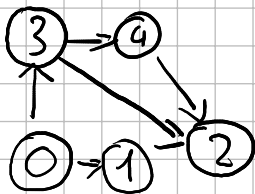
Directed Acyclic graph (DAG):

è un grafo orientato dove non sono presenti cicli.

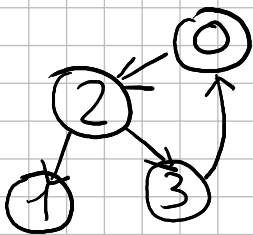
Es:



è un DAG

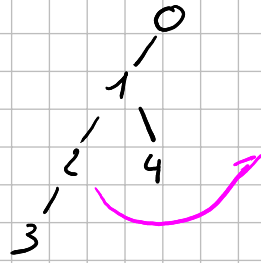
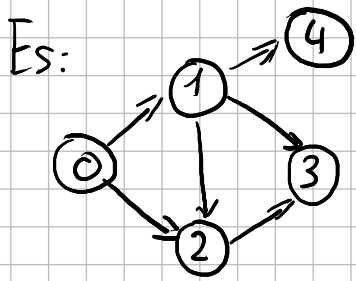


è un DAG



non è un DAG, perché è presente un ciclo.

In un DAG quando la visita DFS di un nodo termina sono automaticamente terminate anche tutte le visite dei suoi discendenti.



quando 2 termino abbiamo esplorato tutti i suoi discendenti.

Stessa cosa vale per qualsiasi altro nodo.

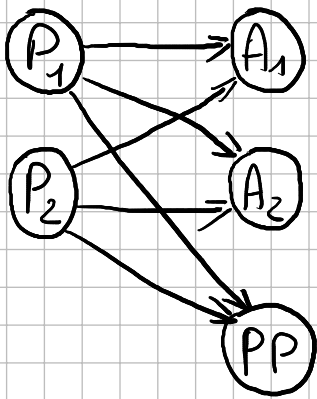
## Ordine Topologico:

### Definizione formale:

L'ordine Topologico è un ordine lineare di tutti i nodi di un DAG

dove se  $G$  contiene l'arco  $(u, v)$   $u$  compare prima di  $v$ .

Es:



nodo dalla quale parte l'ordine

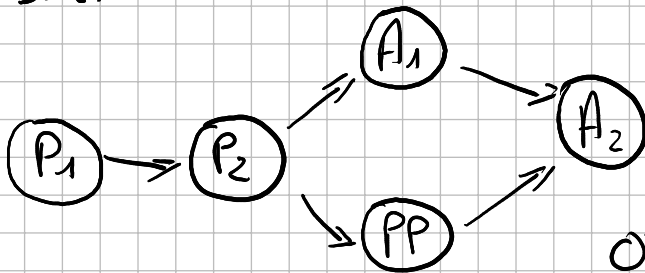
OT(P<sub>2</sub>): P<sub>2</sub>, P<sub>1</sub>, A<sub>2</sub>, PP, A<sub>1</sub>

l'ordine Topologico **deve** partire

da un nodo senza archi entranti.

Oppure parto dal fondo, scegliendo un nodo senza archi uscenti.

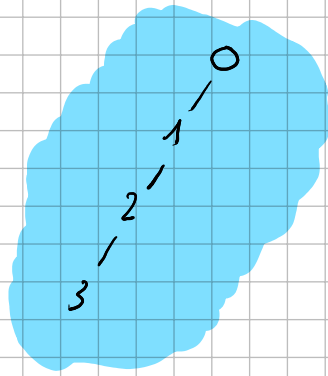
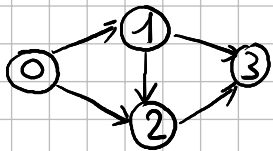
Es 2:



OT(P<sub>1</sub>) = P<sub>1</sub>, P<sub>2</sub>, A<sub>1</sub>, PP, A<sub>2</sub>

Es 3:

Eseguiamo una DFS su questo DAG



Notiamo che l'ordine di fine visita  $T$  è:

3, 2, 1, 0 ovvero l'ordine topologico inverso, infatti abbiamo

$$OT = 0, 1, 2, 3$$

Questo è vero perché:

In un DAG quando la visita DFS di un nodo termina sono automaticamente terminate anche tutte le visite dei suoi discendenti.

Implementazione:

visita DFS (sorg)

$$S = S \cup \{sorg\}$$

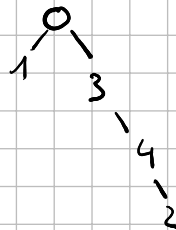
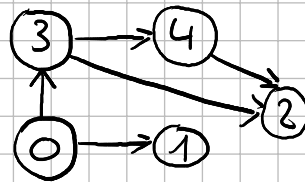
per ogni vicino  $v$  di  $u$

se  $v \notin S$

visita DFS ( $v$ )

$$OT[sorg] = countOT$$

$countOT--$  //  $countOT$  è inizializzato all'ordine del grafo



posizione nell'ordine topologico.

$OT[1] = 4$   $OT[2] = 3$   $OT[4] = 2$   $OT[3] = 1$

$OT[0] = 0$

0, 3, 4, 2, 1

Dimostrazione dell'ordine topologico

Prerequisiti:  $G$  deve essere un DAG

$$G = (V, E)$$

Tesi: l'ordine di fine visita è esattamente l'inverso dell'ordine Topologico.

$$OFV(u) > OFV(v)$$

noi sappiamo che

$$OT(u) = n-1 - OFV(u)$$

$$OT(v) = n-1 - OFV(v)$$

quindi:

$$OT(u) - OT(v) < 0$$

$$n-1 - OFV(u) - (n-1 - OFV(v)) < 0$$

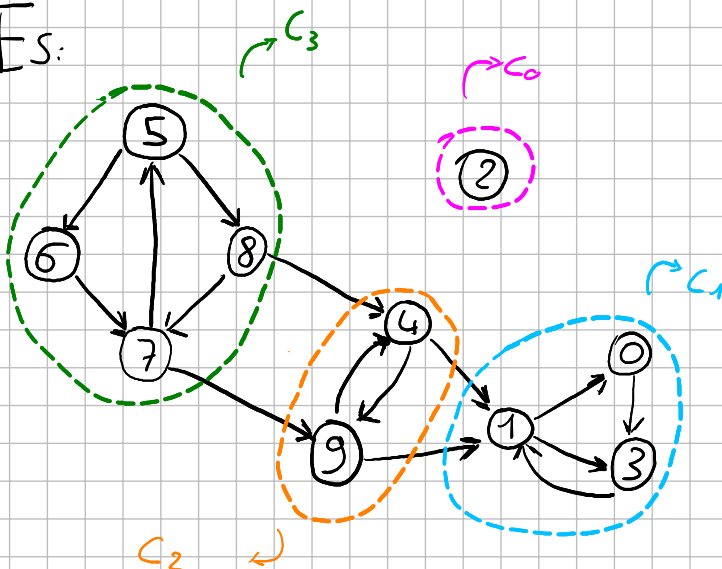
$$\cancel{n-1} - OFV(u) - \cancel{n-1} + OFV(v) < 0$$

$$OFV(v) - OFV(u) < 0$$

## Algoritmo di Kosaraju

Con questo algoritmo sei in grado di trovare tutte le SCC in un grafo orientato, ed assegnare ad ogni nodo la sua SCC.

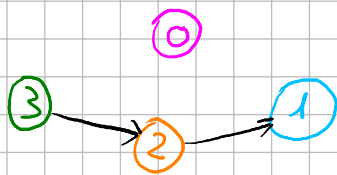
Es:



In questo grafo sono presenti:

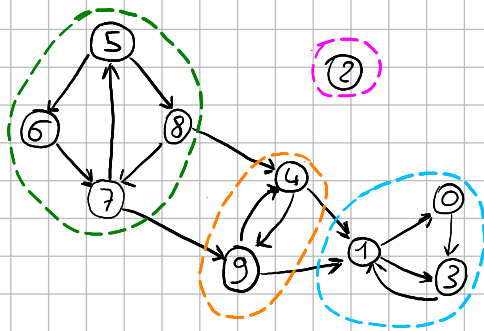
4 SCC, cerchiamo di capire come trovarle.

Creiamo il grafo delle SCC



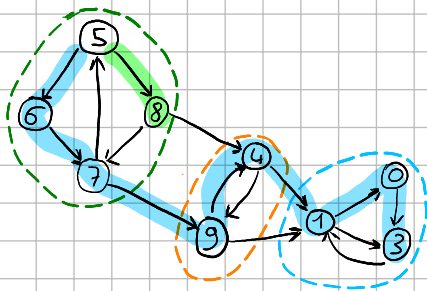
Bene, abbiamo appena costruito un DAG.

Il grafo delle SCC è sempre un DAG.



Indipendentemente da dove partiamo  
la visita completa DFS  
termina su un nodo della SCC 3

Es partendo da SCC 3

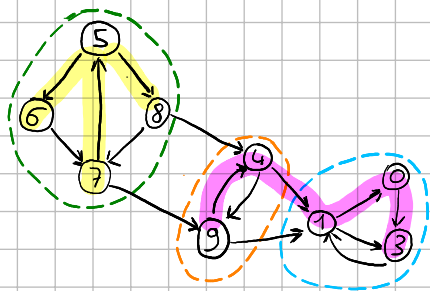


// Non è stato preso il nodo 2 per semplicità  
ma in realtà la visita è completa.

Sorg: 5

OFV: 3, 0, 1, 4, 9, 7, 6, 8, 5 ← Fa parte di SCC 3

Es partendo da SCC 2



Anche lui fa parte di  
↑ SCC 3

OFV: 3, 0, 1, 4, 9, 6, 8, 5, 7

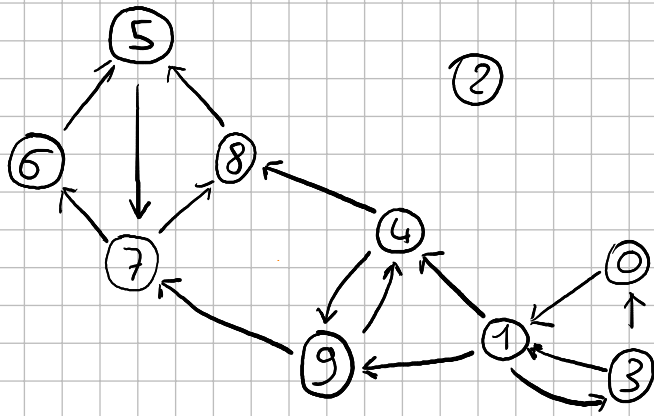
Il OT del grafo delle SCC è: OT = 3, 2, 1, 0

Noi vogliamo che il nostro algoritmo segua l'OT al contrario, in modo da determinare i nodi delle SCC.



Per Farlo non puoi usare OFV (soprattutto al contrario), Sarebbe un  
caso!

Devi trasporre il grafo ( $G^T$ ), ovvero invertire il verso di tutti gli archi.

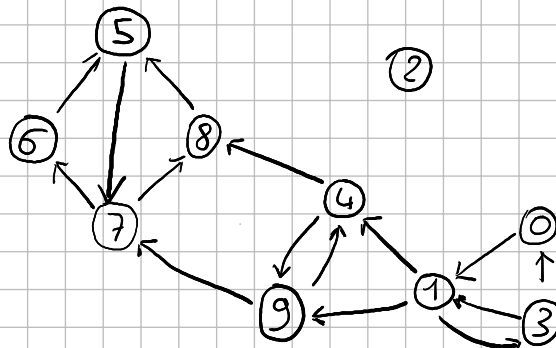


Dopo aver trasposto come ordine di visita usi OFV  
(visto negli esempi di prima), ma inverso.

OFV dell'Es 1 al contrario

5, 8, 6, 7, 9, 4, 1, 0, 3

Visita sul nodo 5, scopro  
la prima SCC



Poi visita sul nodo 9 (ignora 8, 6, 7 perché scoperti dalla visita precedente)  
e scopro la seconda SCC

Poi visita 1 ...

Ed infine visita 2.

Abbiamo scoperto tutte le SCC del grafo!

## Riassunto e Costi:

1) Faccio una visita DFS del grafo e mi salvo OFV

Costo:  $O(n+m)$

2) Creo  $G^T$

Costo:  $O(m)$  // ogni tanto scrive  $n+m$  ogni tanto solo  $m$ , per me  $m$  ha più senso perché lavora solo sui archi.

3) Visito  $G^T$  seguendo l'OFV ottenuto dal punto 1 ma inverse.

Mentre lo visito mi salvo le sc dei nodi

Costo:  $O(n+m)$

Le visite ovunque sono complete!