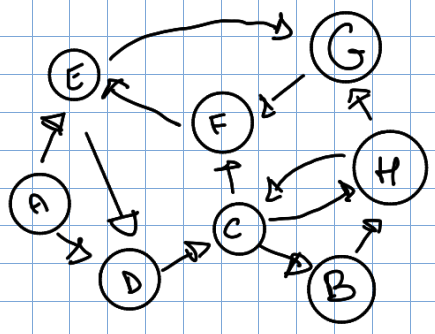
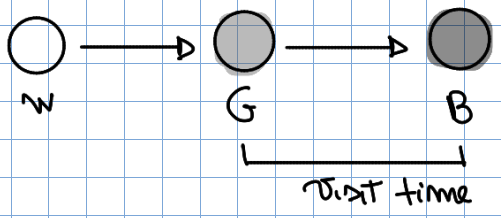
DFS

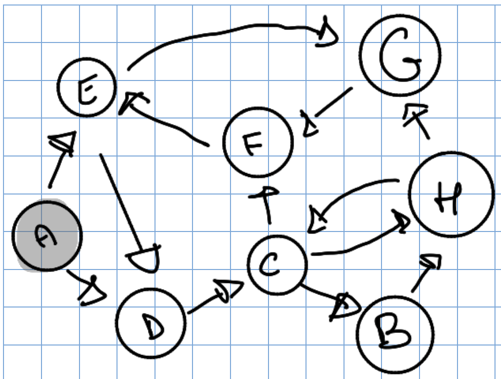
Naturalmente ricorsivo, viene utilizzato per trovare le componenti fortemente connesse

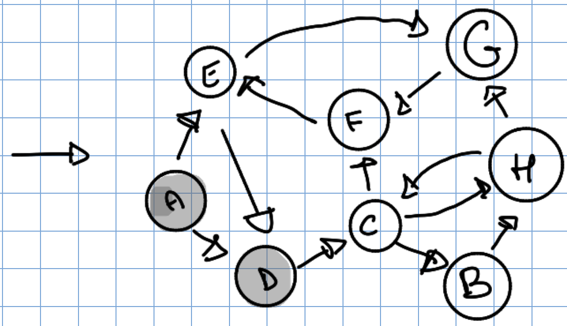


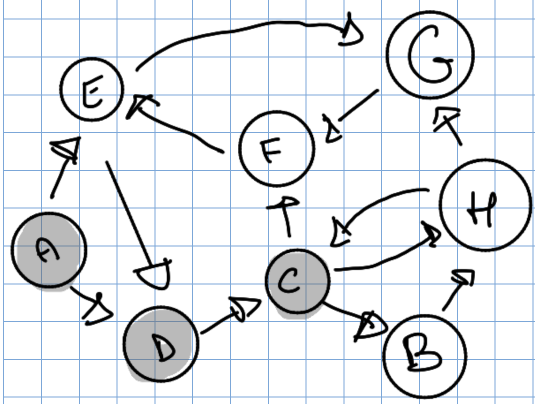


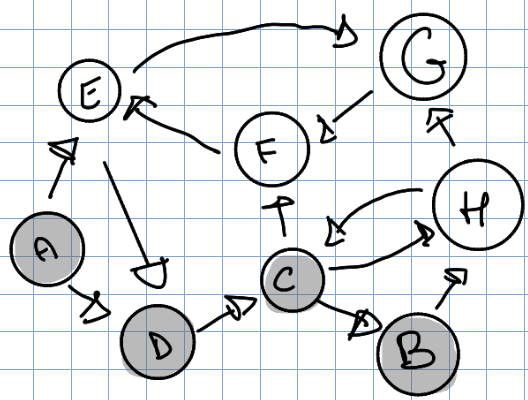
Durante la visita crea un albero di DFS

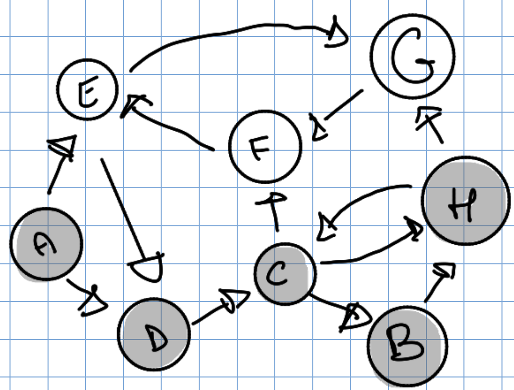
```
DFS(G, r)
  for each v ∈ V DO
      color[v] ← W
      π[v] ← NULL          indica il padre del nodo v nell'albero DFS
  DFS-visit(r) ──────────► DFS-visit(v)
                             color[v] ← G
                             for each u ∈ adj(v) DO:
                                 if color[u] = W then
                                     π[u] ← v
                                     DFS-visit(u)
                             color[v] ← B
```

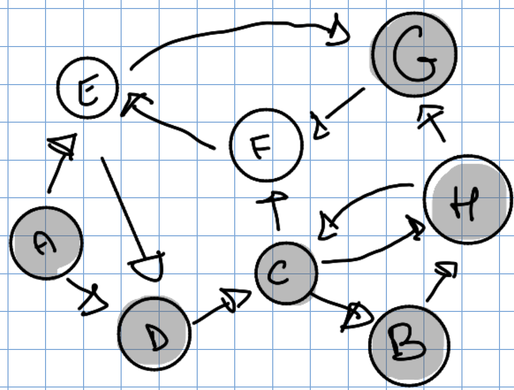

E
G
F
H
A
C
D
B

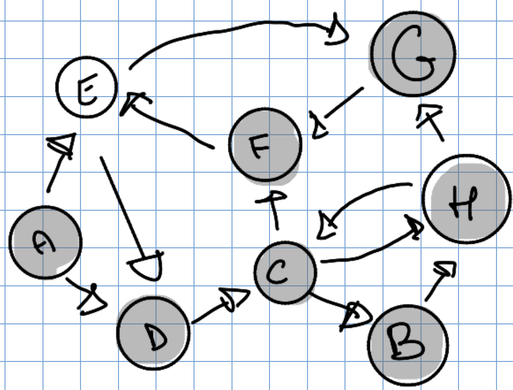

E
G
F
H
A
C
D
B

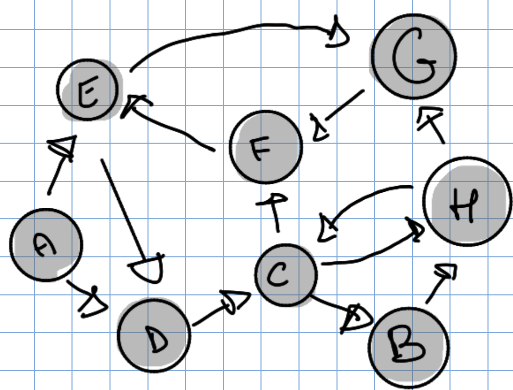

E
G
F
H
A
C
D
B

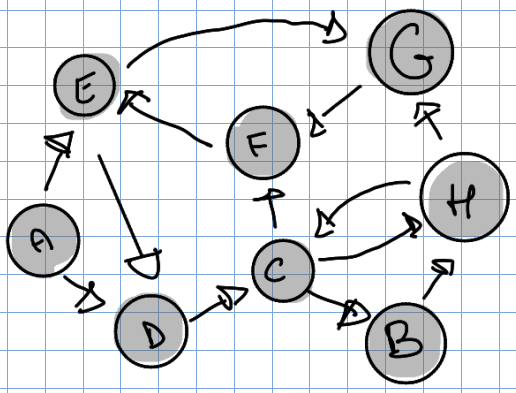
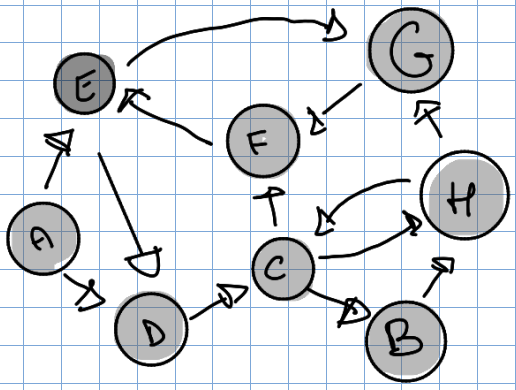
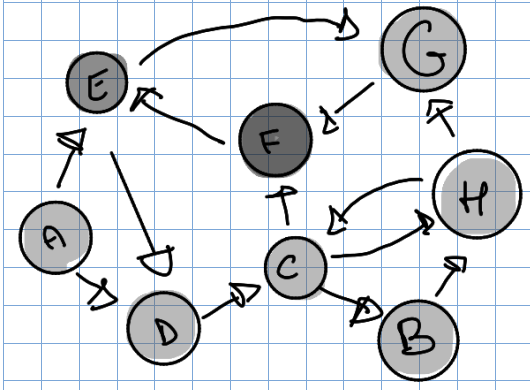
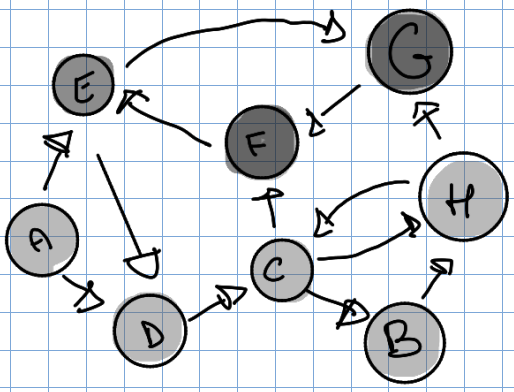
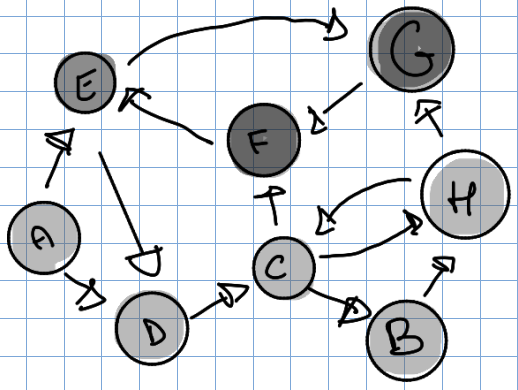
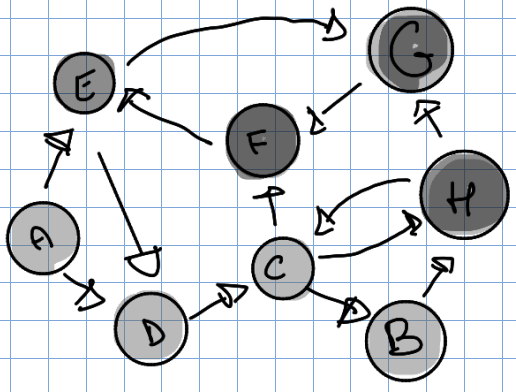
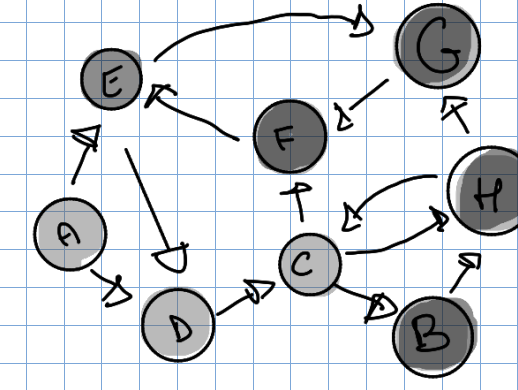
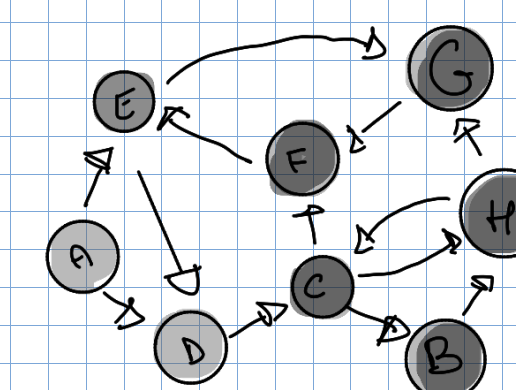

E
G
F
H
A
C
D
B

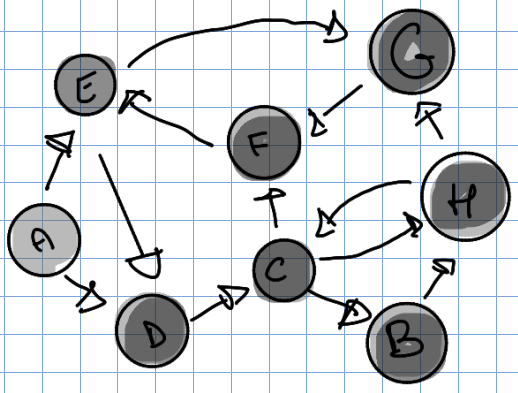

E
G
F
H
A
C
D
B

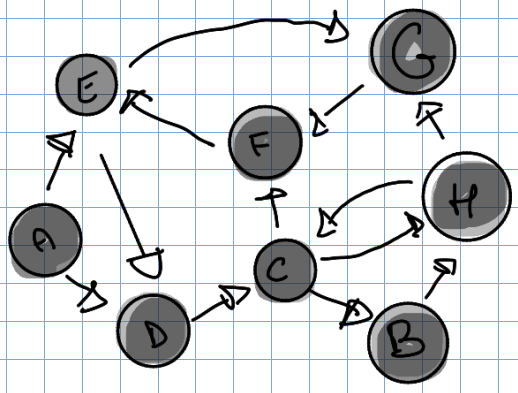

E
G
F
H
A
C
D
B


E
G
F
H
A
C
D
B


E
G
F
H
A
C
D
B





tutto questo crea la seguente albero DFS

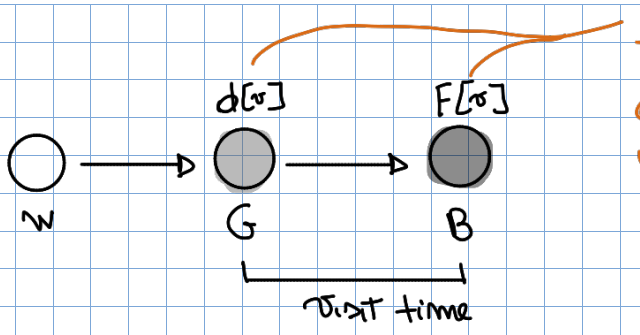
Con queste procedure alcuni nodi non vengono analizzati quindi diventa

```
DFS(G, v)
  for each v ∈ V DO
      color[v] ← W
      π[v] ← NULL
  DFS-visit(v)
```

uno per ogni sorgente

Con questo cambiamento le sorgenti se più di una generano altri alberi DFS - una foreste DFS

```
DFS(G, v)
  for each v ∈ V DO
      color[v] ← W
      π[v] ← NULL
  for each v ∈ V DO
      if color[G] = W then
          DFS-visit[v]
```



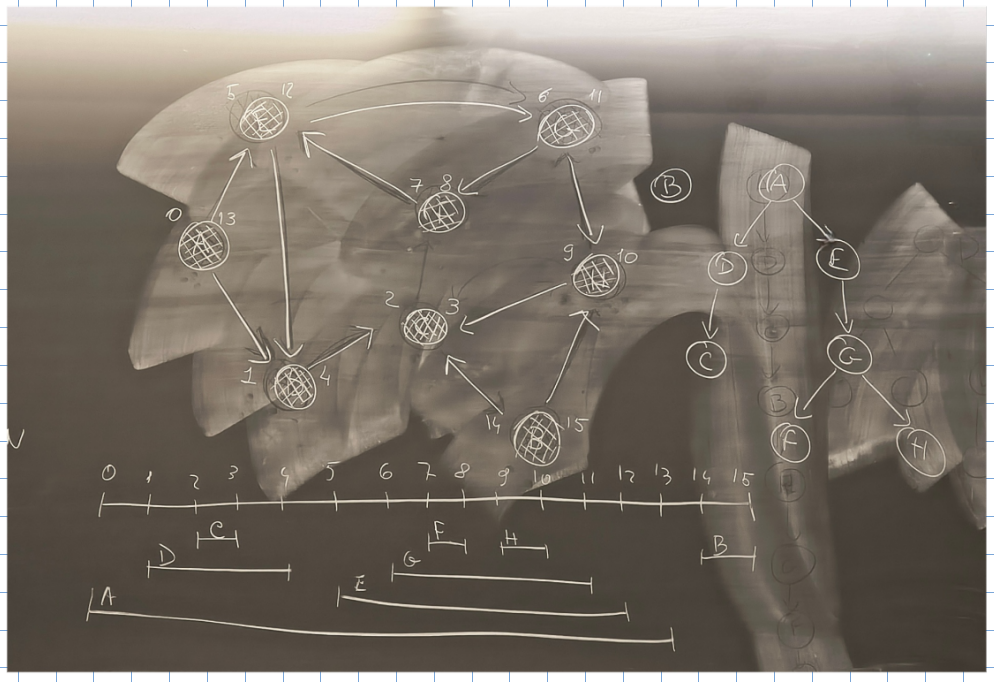
array che tengono traccia di tempo di inizio e fine visita

DFS visit diventa

```
DFS-visit(v)
  d[v] ← t
  t ← t+1
  color[v] ← G
  for each u ∈ adj(v) DO:
      if color[u] = w then
          π[u] ← v
          DFS-visit(u)
      color[v] ← B
  F[v] ← t
  t ← t+1
```

variabile globale che scandisce il tempo

Una visita scandita da tempo diventa:



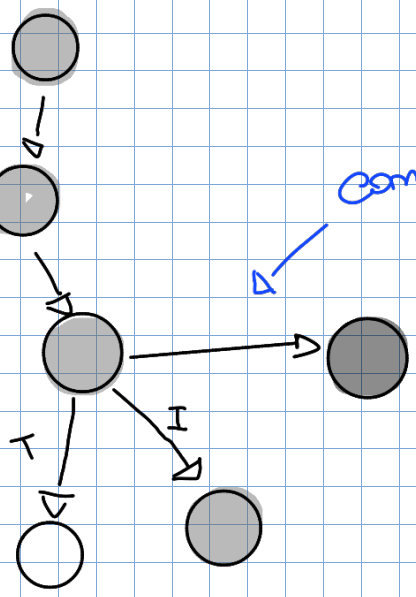
etichettiamo gli archi mentre:

T: archi consecutivi nell'albero DFS

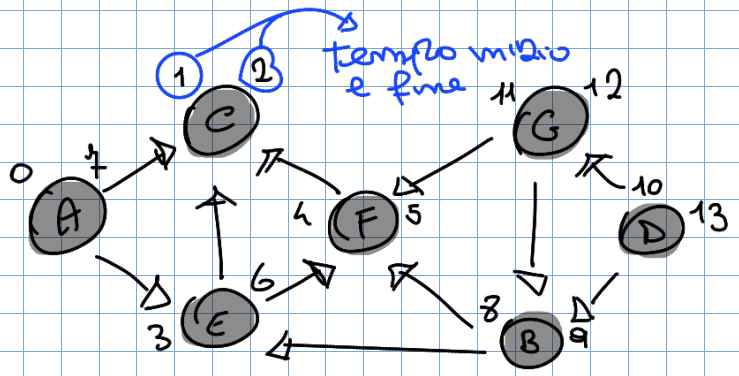
I: archi all'indietro

F: archi in avanti → collega A a C in discendente ma non diretto

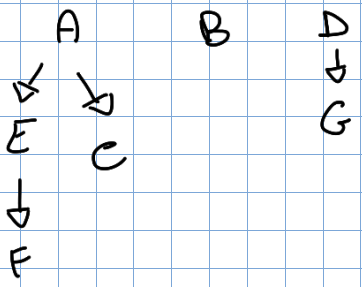
C: archi di attraversamento →

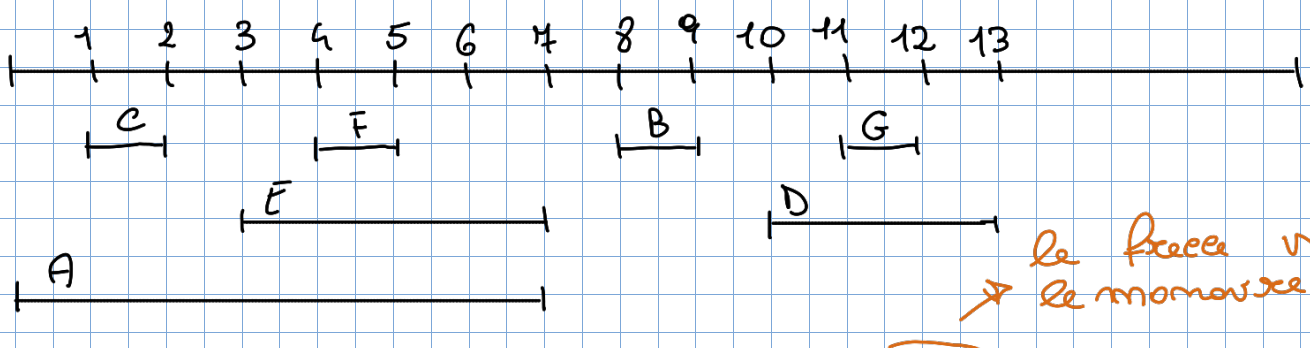


Perché ci servono tutte queste informazioni? Ora lo vediamo fare un ordinamento topologico di un grafo usando la DFS



genera →





le frecce indicano le monovie possibili

ordinamento topologico manuale: A, D, G, B, E, F, C

ordine rispetto al tempo di fine visita:

D G B A E F C

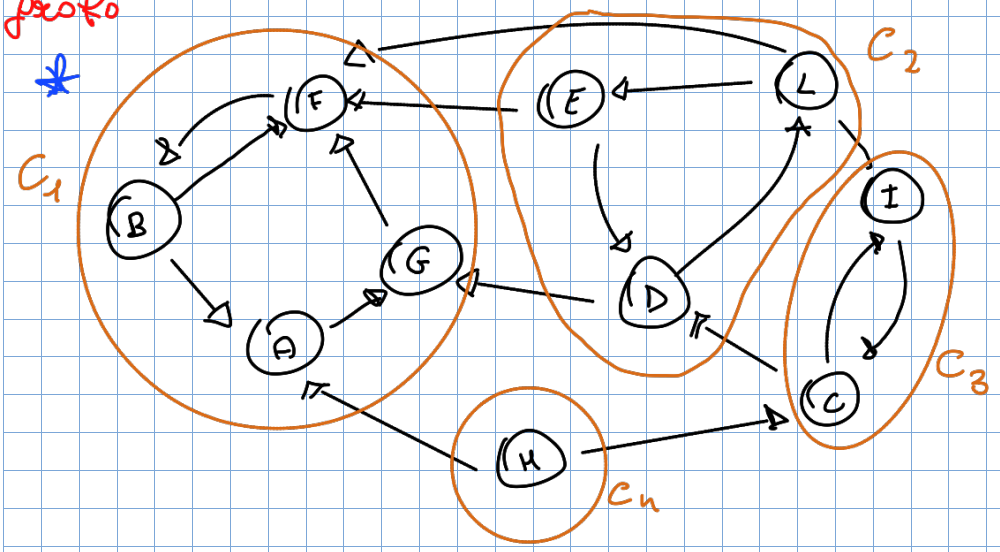
↳ che è un ordinamento valido

quindi possiamo usare la DFS sia per l'ordinamento sia per verificare l'aciclicità

Complessità DFS

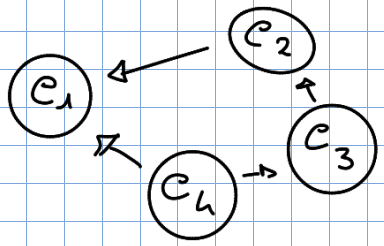
$\sum_{v \in V} O(Adj(v)) = |E|$ che sarebbe $O(V + E)$

Identificare le componenti fortemente connesse di un grafo



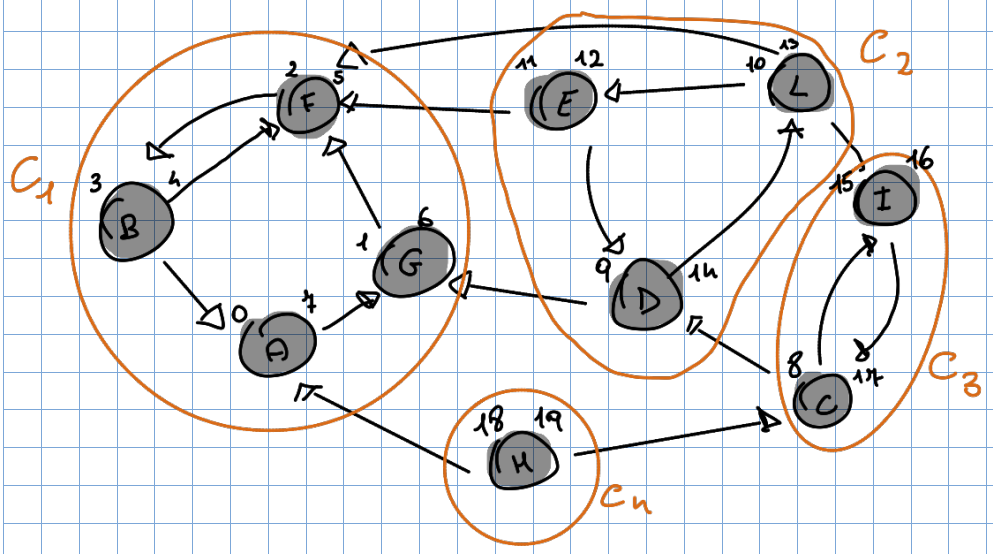
C_1	C_2	C_3	C_4
A G B F	E D L	C I	H

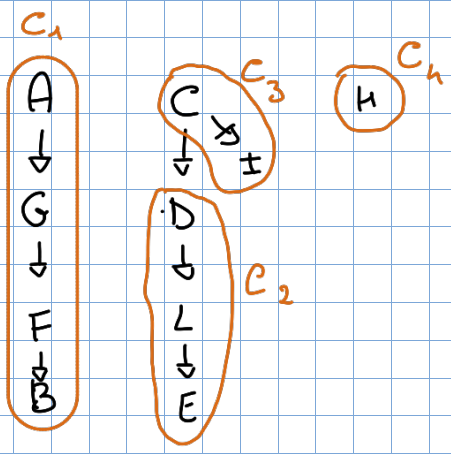
Da questo modo il grafo dei moccamodi:



Se esiamo dei cicli non abbiamo fatto bene le divisioni in comp fort. conn

Una DFS sul grafo * ci dona:





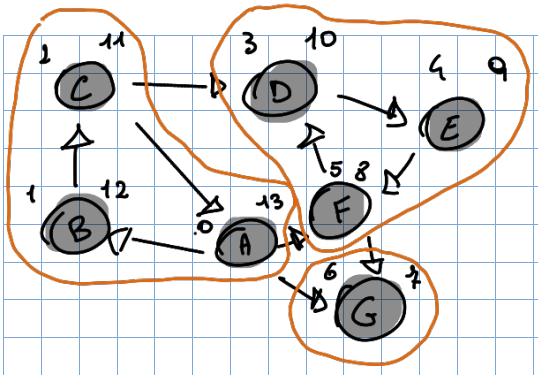
Possiamo fornire il fatto di avere k alberi tanti quante sono le componenti connesse? Sì, devo fare una DFS che simula un ordine topologico delle componenti connesse

archi con direzioni inverse alle originali

ordinamento topologico usando la DFS del grafo trasposto:

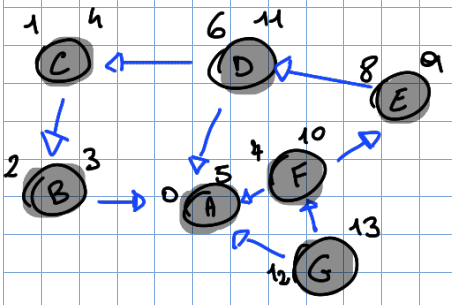
H C I D L E A G F B

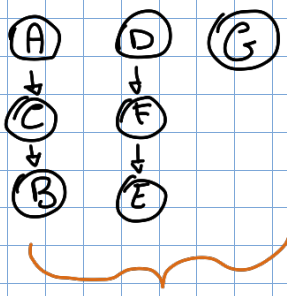
↳ facendo le foreach della DFS trovo tanti alberi quanti sono le comp conn



ordine topologico inverso: A B C D E F G → le DFS iterando su questo ci dà

Grafo trasposto





3 componenti connesse

1°: DFS: $O(V+E)$

2°: Creazione grafo trasposto: $O(V+E)$

3°: DFS $O(V+E)$