Corso di laurea in Informatica
Progettazione d'algoritmi

I grafi 3

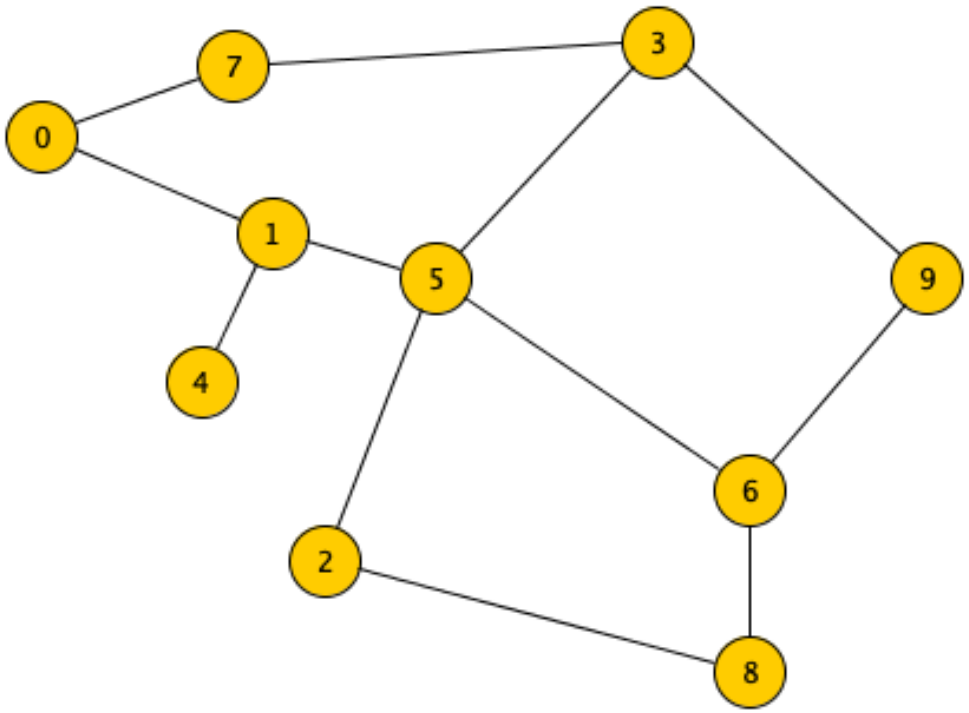
Angelo Monti

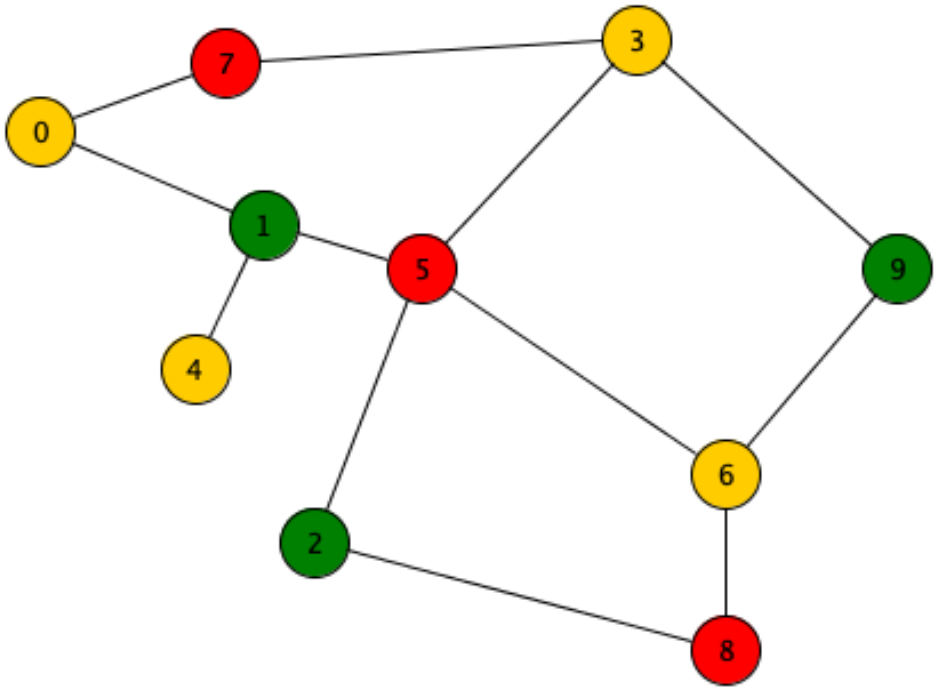
SAPIENZA
UNIVERSITÀ DI ROMA

Colorazione di grafi

Dato un grafo connesso G ed un intero k vogliamo sapere se è possibile colorare i nodi del grafo in modo che nodi adiacenti abbiamo sempre colori distinti.

Esempio di grafo 3-colorabile:





Il teorema dei 4 colori: un grafo planare richiede al più 4 colori per essere colorato.

- Il problema venne posto per la prima volta nel 1852 da uno studente che congetturò che 4 colori sono sempre sufficienti.

- Negli anni successivi molti matematici tentarono invano di dimostrare la congettura.

- La prima dimostrazione fu proposta solo nel 1879.

- Solo nel 1890 si scoprì che la dimostrazione conteneva un sottile errore. Si provò almeno che 5 colori sono sempre sufficienti a colorare una mappa.

- La dimostrazione che 4 colori son sufficienti fu trovata solo nel 1977. Si basa sulla riduzione del numero infinito di mappe possibili a 1.936 configurazioni, per le quali la validità del teorema viene verificata caso per caso con l'ausilio di un calcolatore.

- L'utilizzo di un algoritmo per verificare l'esattezza della congettura scatena grandi polemiche sull'affidabilità di questi metodi. Il fatto che la dimostrazione sia basata sull'analisi di una moltitudine di casi discreti porta alcuni matematici a contestarne la validità: sia per l'impraticabilità di una verifica manuale di tutti i casi possibili, sia per l'impossibilità di avere la certezza che l'algoritmo sia implementato correttamente. Fino ad oggi nell'algoritmo non è stato trovato alcun errore.

- Nel 2000 è stata proposta una nuova dimostrazione del teorema che richiede l'utilizzo della teoria dei gruppi.

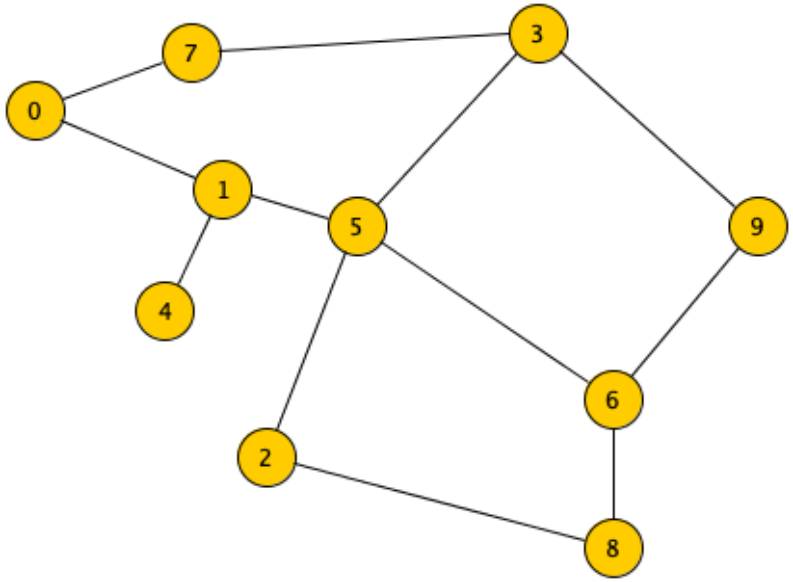
- In generale un grafo può richiedere anche $\Theta(n)$ colori.

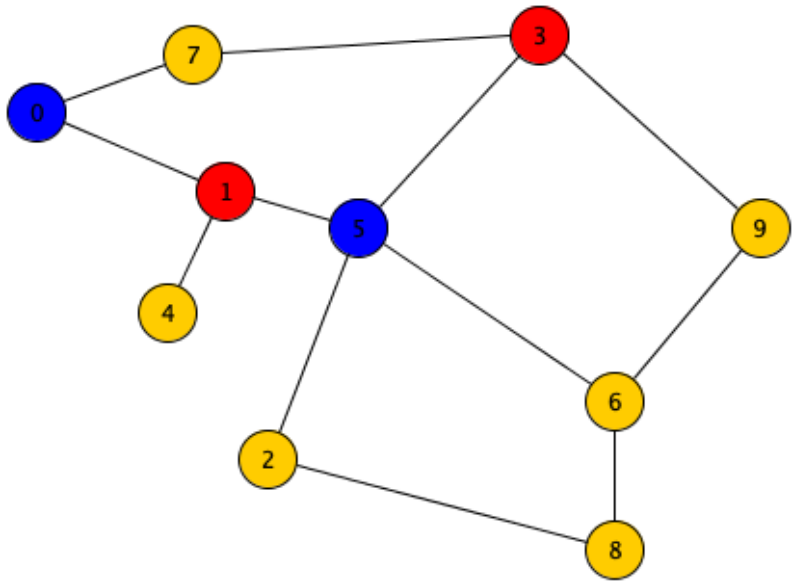
- Non è noto nessun algoritmo polinomiale che, dato un grafo planare G, determini se G è 3-colorabile.

- Non è difficile progettare un algoritmo che determini se un grafo è 2-colorabile:

Un grafo è 2-colorabile se e solo se **non** contiene cicli di lunghezza dispari

Un ciclo di lunghezza dispari rende impossibile la colorazione del grafo con due colori:





Lungo il ciclo i colori devono alternarsi.

L'algoritmo di bi-colorazione che segue prova che un grafo senza cicli dispari può sempre essere 2-colorato:

- Colora il nodo 0 col colore 0
- effettua una visita in profondità del grafo a partire dal nodo 0. Nel corso della visita, a ciascun nodo x che incontri assegna uno dei due colori 0 e 1. Scegli il colore da assegnare in modo che sia diverso dal colore assegnato al nodo padre che ti ha portato a visitare x.

La prova di correttezza: Siano x e y due nodi adiacenti in G, consideriamo i due possibili casi e facciamo vedere che in entrambi i casi i due nodi al termine avranno colori opposti.

1. *l'arco (x, y) viene attraversato durante la visita.* In questo caso banalmente i due nodi hanno colori distinti.
2. *l'arco (x, y) NON viene attraversato durante la visita.* Sia x il nodo visitato prima. Esiste un cammino in G che da x porta a y (quello seguito dalla visita), questo cammino si chiude a formare un ciclo con l'arco (y, x). Il ciclo è di lunghezza pari per ipotesi, quindi il cammino è di lunghezza dispari. Poiché sul cammino i colori si alternano il primo nodo (x) e l'ultimo nodo (y) del cammino avranno colori diversi.

Di seguito l'algoritmo per bi-colorare grafi connessi G senza cicli dispari:

```
def Colora(G):
    '''Su grafi connessi bicolorabili (ovvero senza cicli
    dispari) restituisce una bicolorazione dei nodi'''
    ####
    def DFSr(x, G, Colore, c):
        Colore[x]=c
        for y in G[x]:
            if Colore[y]==-1:
                DFSr(y, G, Colore, 1-c)
    ###
    Colore = [ -1]* len(G)
    DFSr(0, G, Colore, 0)
    return Colore
```

Se il grafo G contiene cicli dispari l'algoritmo produce un assegnamento di colori sbagliato.

Nella versione che segue l'algoritmo produce una bi-colorazione se il grafo G è bi-colorabile, produce una lista vuota in caso contrario.

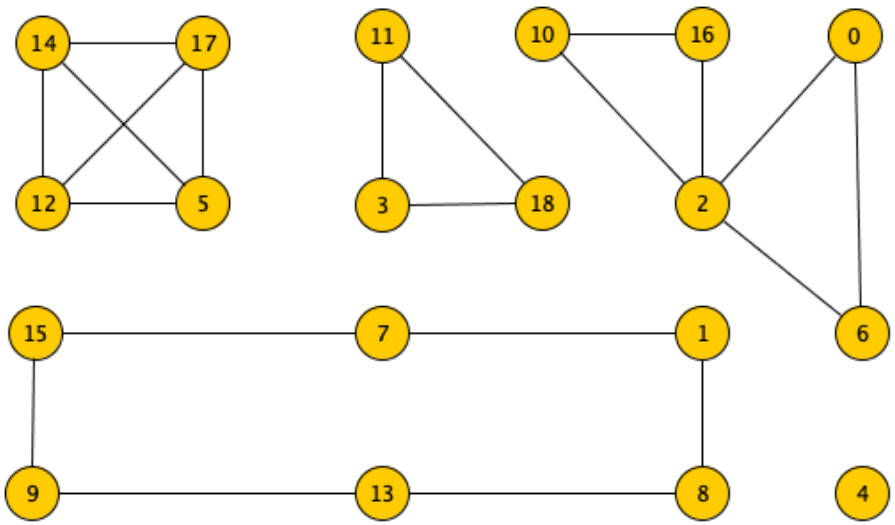
```
def Colora1(G):
    '''Restituisce una bicolorazione di G se il grafo è
    bicolorabile, una lista vuota altrimenti'''
    ####
    def DFSr(x, G, Colore, c):
        Colore[x]=c
        for y in G[x]:
            if Colore[y]==-1:
                if not DFSr(y, G, Colore, 1-c):
                    return False
            elif Colore[y]== Colore[x]:
                return False
        return True
    ###
    Colore = [ -1]* len(G)
    if DFSr(0, G, Colore, 0):
        return Colore
    return []
```

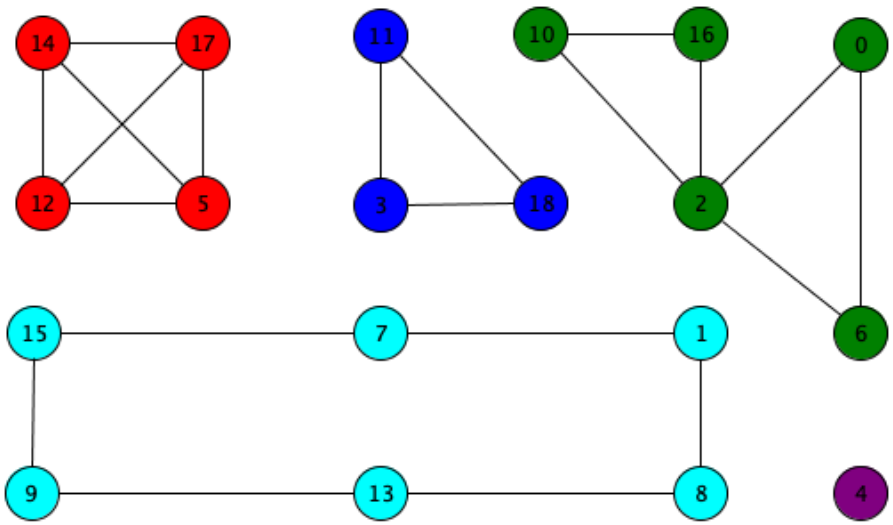
La complessità dell'algoritmo per testare se un grafo è bicolorabile è quella di una semplice visita del grafo connesso da colorare:

$$O(n+m) = O(m)$$

(dove l'ultima uguaglianza dipende dal fatto che in un grafo connesso $m \geq n-1$**)**

Una componente connessa (o semplicemente una componente) di un grafo (indiretto) è un sottografo composto da un insieme massimale di nodi connessi da cammini.
Un qrafo si dice connesso se ha una sola componentennessa.





Vogliamo calcolare il vettore C delle componenti connesse di un grafo G.

Vale a dire il vettore C che ha tanti elementi quanti sono i nodi del grafo e $C[u] = C[v]$ se e solo se u e v sono nella stessa componente connessa.

$C =$

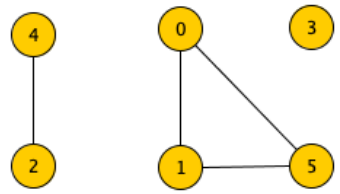
2	1	2	3	4	5	2	1	1	1	2	3	5	1	5	1	2	5	3

```python
def Componenti(G):
    '''restituisce il vettore C delle
    componenti connesse del grafo G'''
    ###
    def DFSr(x, G, C, c):
        C[x] = c
        for y in G[x]:
            if C[y] == 0:
                DFSr(y, G, C, c)
    ###
    C = [0]*len(G)
    c = 0
    for x in range(len(G)):
        if C[x] == 0:
            c += 1
            DFSr(x, G, C, c)
    return C
```

La complessità della procedura è $O(n+m)$

```python
>>> G = [[1, 5], [0, 5], [4], [], [2], [0, 1]]

>>> Componenti(G)
[1, 1, 2, 3, 2, 1]
```

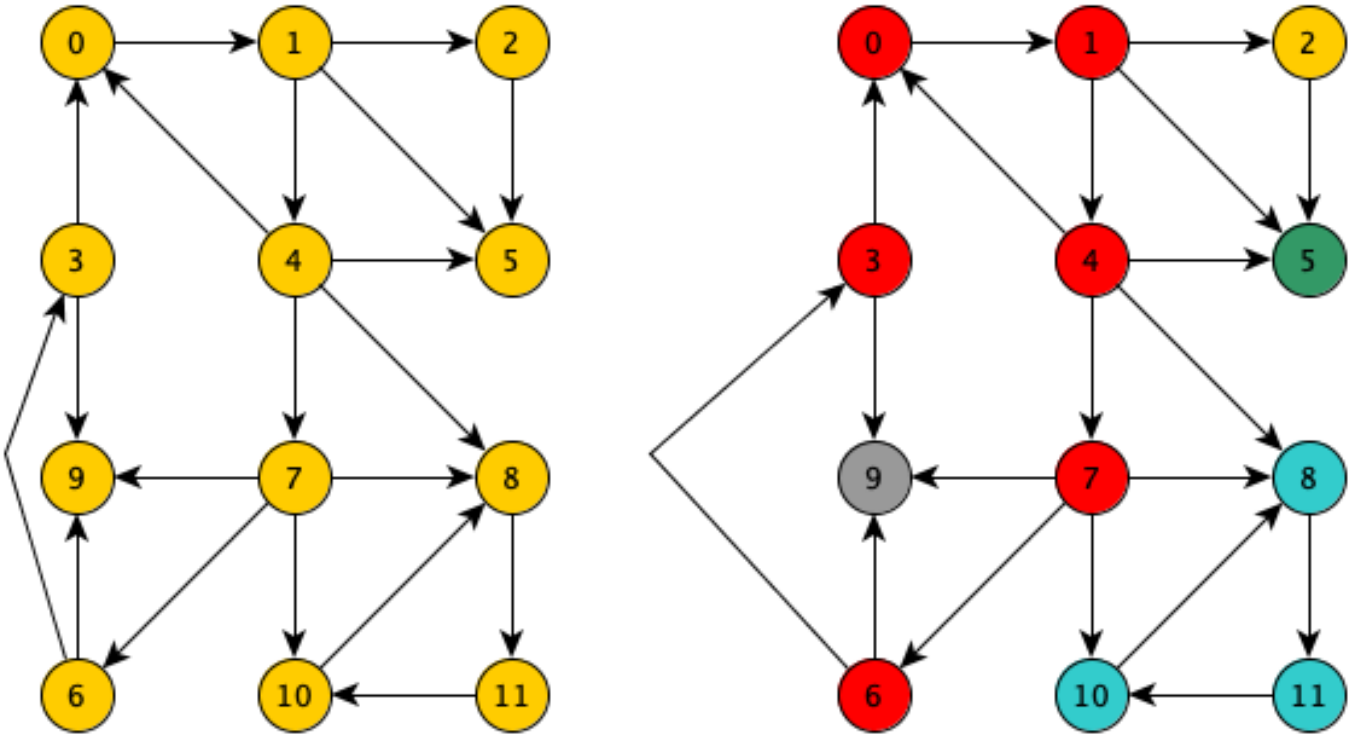


$C =$

1	1	2	3	2	1

Una componente fortemente connessa di un grafo diretto è un sottografo composto da un insieme massimale di nodi connessi da cammini.

Un grafo diretto si dice fortemente connesso se ha una sola componente.



Vogliamo calcolare il vettore C delle componenti fortemente connesse di un grafo G.

Vale a dire il vettore C che ha tanti elementi quanti sono i nodi del grafo e $C[u] = C[v]$ se e solo se u e v sono nella stessa componente fortemente connessa.

$$C = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 2 & 1 & 1 & 3 & 1 & 1 & 4 & 5 & 4 & 4 \\ \hline \end{array}$$

ATTENZIONE: l'algoritmo utilizzato per le componenti connesse non funziona nel caso di componenti fortemente connesse.

Basta pensare al grafo costituito dalla catena:

$$0 - - > 1 - - > 2 - - > 3 - - > 4.$$

La procedura delle componenti connesse produce la soluzione

$$C = \begin{array}{|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 1 \\ \hline \end{array}$$

mentre la soluzione corretta è

$$FC = \begin{array}{|c|c|c|c|c|} \hline 1 & 2 & 3 & 4 & 5 \\ \hline \end{array}$$

Il problema risiede nel fatto che se c'è un cammino che da x mi porta a y questo non significa che x e y sono nella stessa componente fortemente connessa, perché questo sia vero è necessario che ci sia anche un cammino che da y mi porti a x.

Idea per un algoritmo che, dato il grafo diretto G ed un suo nodo u, restituisce i nodi della componente fortemente connessa che contiene u:

1. calcola l'insieme A dei nodi di G raggiungibili da u
2. calcola l'insieme B dei nodi di G che portano a u
3. restituisci l'intersezione dei due insiemi A e B

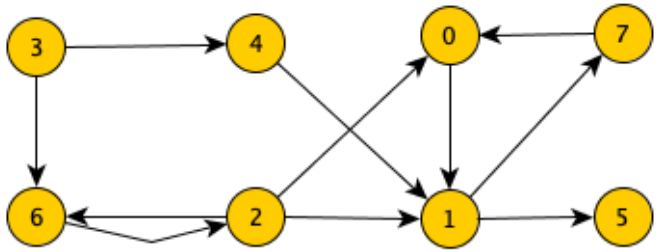
Complessità :

- Il passo 1 richiede $O(n+m)$ (una semplice visita DFS che parte da u)
- Il passo 2 $O(?)$
- Il passo 3 costa $O(n)$

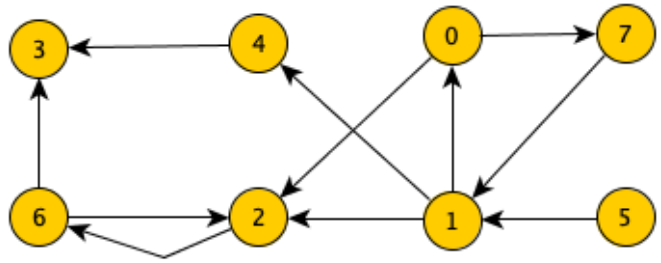
Per eseguire efficientemente il passo 2 si può ricorrere al grafo trasposto di G (G^T).

Dato un grafo diretto G il grafo trasposto di G (denotato con G^T) ha gli stessi nodi di G ma archi con direzione opposta.

G=



G^T=



NOTA: I nodi che in G portano a u sono i nodi che in G^T sono raggiungibili a partire da u.

Il passo 2 dell'algoritmo può essere eseguito in tempo $O(n+m)$ cercando i raggiungibili da u in G^T (ovviamente il grafo G^T andrà prima costruito)

```python
def ComponenteFC(x,G):
    visitati1 = DFS(x,G)
    G1 = Trasposto(G)
    visitati2 = DFS(x,G1)
    componente = []
    for i in range(len(G)):
        if visitati1[i] == visitati2[i] == 1:
            componente.append(i)
    return componente
```

```python
def Trasposto(G):
    GT =[[] for _ in G]
    for i in range(len(G)):
        for v in G[i]:
            GT[v].append(i)
    return GT
```

Idea per algoritmo che, dato il grafo diretto G ed un suo nodo u restituisce i nodi della componente fortemente connessa che contiene u:

1. **calcola l'insieme A dei nodi di G raggiungibili da u**
2. **calcola il grafo $G1$ trasposto di G**
3. **calcola l'insieme B dei nodi di $G1$ raggiungibili da u**
4. **restituisci l'intersezione dei due insiemi A e B**

Complessità :

- **Il passo 1 costa $O(n+m)$ (visita DFS che parte da u in G)**
- **Il passo 2 costa $O(n+m)$**
- **Il passo 3 costa $O(n+m)$ (visita DFS che parte da u in in G^T)**
- **Il passo 4 costa $O(n)$**

L'algoritmo ha dunque complessità $O(n+m)$.

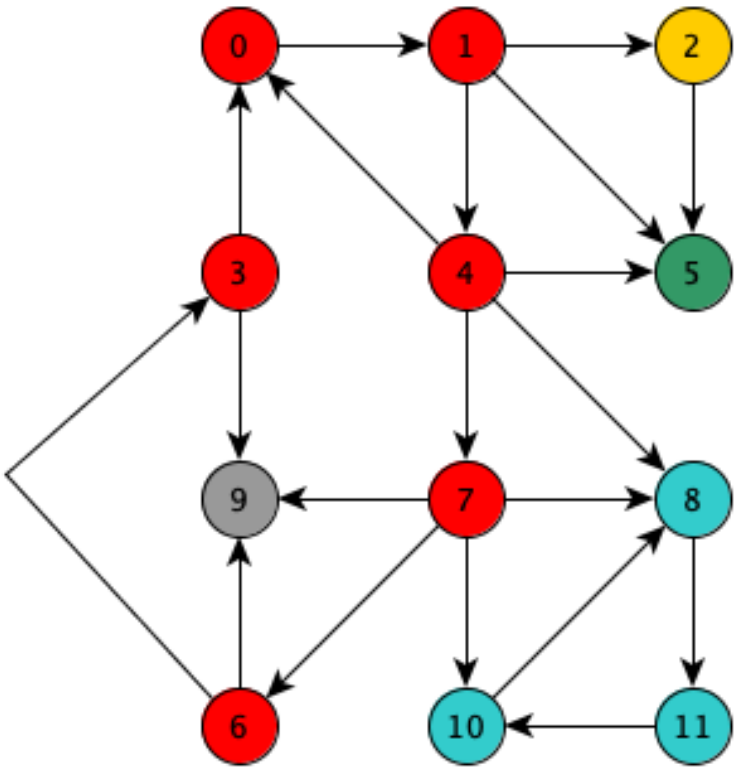
Idea per algoritmo che, dato il grafo diretto G ed un suo nodo u restituisce i nodi della componente fortemente connessa che contiene u:

1. **calcola l'insieme A dei nodi di G raggiungibili da u**
2. **calcola il grafo $G1$ trasposto di G**
3. **calcola l'insieme B dei nodi di $G1$ raggiungibili da u**
4. **restituisci l'intersezione dei due insiemi A e B**

Complessità :

- **Il passo 1 costa $O(n+m)$ (visita DFS che parte da u in G)**
- **Il passo 2 costa $O(n+m)$**
- **Il passo 3 costa $O(n+m)$ (visita DFS che parte da u in in G^T)**
- **Il passo 4 costa $O(n)$**

L'algoritmo ha dunque complessità $O(n+m)$.




```
>>>G=[
      [1], [2, 4, 5], [5], [1, 6],[0, 5, 7, 8],
      [], [3], [6, 8, 9, 10], [11], [],[8], [10]
      ]

>>> componenteFC(0, G)
[0,1,3,4,6,7]

>>> componenteFC(8, G)
[8,10,11]
```

Utilizzando l'algoritmo $ComponenteFC(x, G)$ come subroutine posso ottenere un algoritmo per calcolare il vettore CF delle componenti fortemente connesse:

```
def compFC(G):
    FC = [0]*len(G)
    c = 0
    for i in range(len(G)):
        if FC[i] == 0:
            E = ComponenteFC(i, G)
            c += 1
            for x in E:
                FC[x] = c
    return FC

>>> compFC(G)
[1, 1, 2, 1, 1, 3, 1, 1, 4, 5, 4, 4]
```

La complessità dell'algoritmo è $\Theta(n) \cdot O(n+m) = O(n^2 + nm) = O(n^3)$

```python
def compFC(G):
    FC = [0]*len(G)
    c = 0
    for i in range(len(G)):
        if FC[i] == 0:
            E = ComponenteFC(i, G)
            c += 1
            for x in E:
                FC[x] = c
    return FC
```

Non è difficile provare che al caso pessimo la complessità dell'algoritmo è $\Theta(n^3)$:

considera ad esempio il grafo diretto G avente un arco da u a v per ogni coppia di nodi u, v con $u \leq v$.
Nota che questo grafo ha $\Theta(n^2)$ archi e n componenti fortemente connesse (ciascuna composta da un singolo nodo).

Per il calcolo del vettore CF delle componenti fortemente connesse sono noti diversi algoritmi non banali che lavorano in tempo $O(n+m)$ (ad esempio l'algoritmo di Tarjan o quello di Kosaraju)

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizi da risolvere

1. Di quanto può variare il numero di componenti connesse di un grafo non orientato con l'aggiunta o la rimozione di un arco?

2. Di quanto può variare il numero di componenti fortemente connesse di un grafo orientato con l'aggiunta o la rimozione di un arco?

3. Un grafo si dice *bipartito* se è possibile partizionare i suoi vertici in due insiemi V_1 e V_2 in modo che tutti gli archi abbiano un estremo in V_1 e l'altro in V_2. Progettare un algoritmo che, dato un grafo G, determini in tempo $O(n + m)$ se G è bipartito o meno.

4. Un grafo non orientato si dice euleriano se esiste un ciclo che attraversa ogni arco del grafo esattamente una volta e ritorna al nodo di partenza. Progettare un algoritmo che dato un grafo non orientato determina se questo è euleriano o meno in tempo $O(n + m)$. Nota che in base alla definizione un grafo euleriano non deve essere necessariamente connesso.

- L'aggiunta o la cancellazione di un arco in un grafo non diretto di quanto può far variare il numero di componenti connesse del grafo?

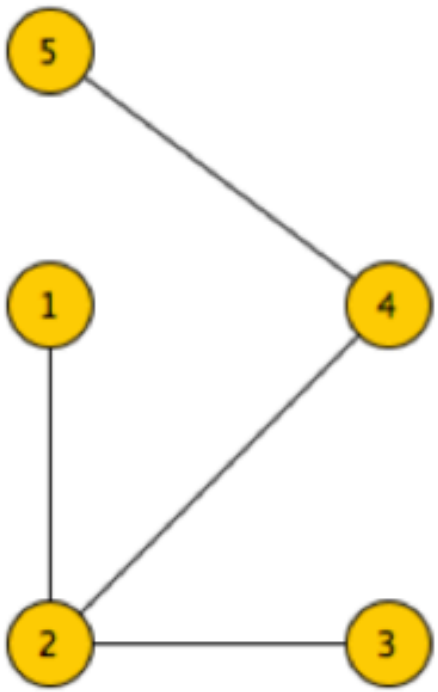
- L'aggiunta o la cancellazione di un arco in un grafo diretto di quanto può far variare il numero di componenti fortemente connesse?

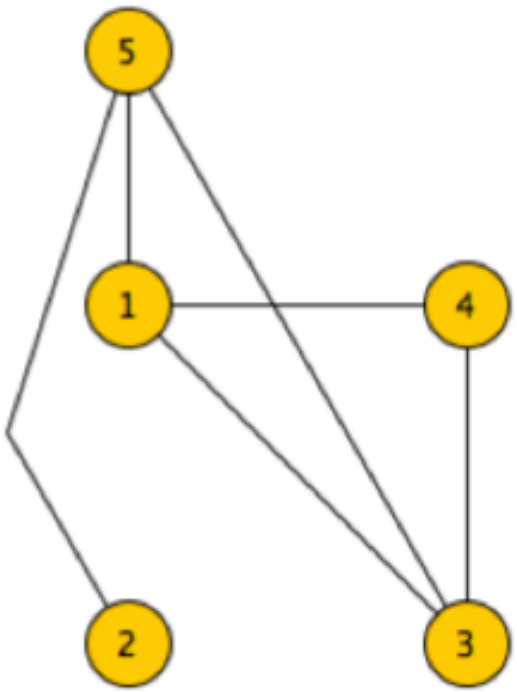
- Un grafo si dice *bipartito* se posso partizionare i suoi vertici in due insiemi V_1 e V_2 tali che tutti gli archi del grafo hanno un estremo in V_1 e l'altro in V_2. Progettare un algoritmo che, dato un grafo G, in tempo $O(n+m)$ determina se G è bipartito o meno.

ESERCIZIO:

Dato un grafo G il suo grafo complementare G^c è un grafo che ha gli stessi nodi di G ma in cui un arco è presente se e solo se manca a G.

Ad esempio, di seguito a sinistra un grafo G e a destra il suo complemento G^c





Dimostrare che, per ogni grafo G, almeno uno dei grafi G e G^c è connesso.

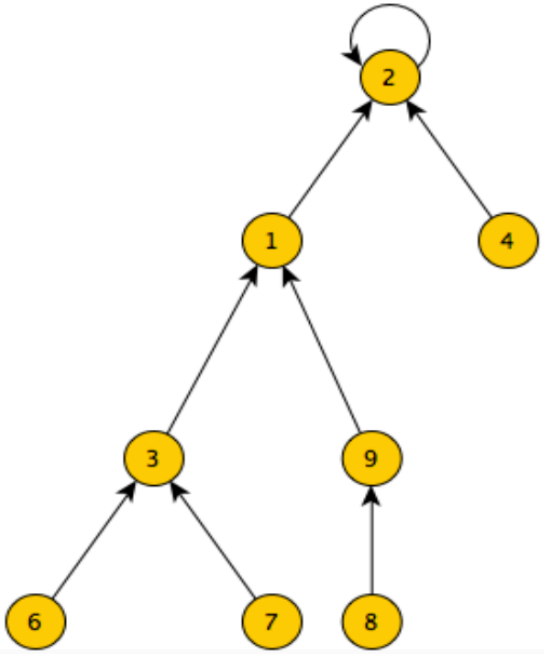
ESERCIZIO:

Dato un albero di n nodi rappresentato tramite il vettore dei padri P (per convenzione il padre del nodo radice è il nodo stesso) e due nodi dell'albero u e v, dare lo pseudocodice di un algoritmo che in tempo $O(n)$ calcola la distanza tra u e v nell'albero.

Ad esempio per

$$P = \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline 2 & 2 & 1 & 2 & 4 & 3 & 3 & 9 & 1 \\ \hline \end{array}$$

$u = 9$ e $v = 4$ deve restituire 3
$u = 3$ e $v = 6$ deve restituire 1.



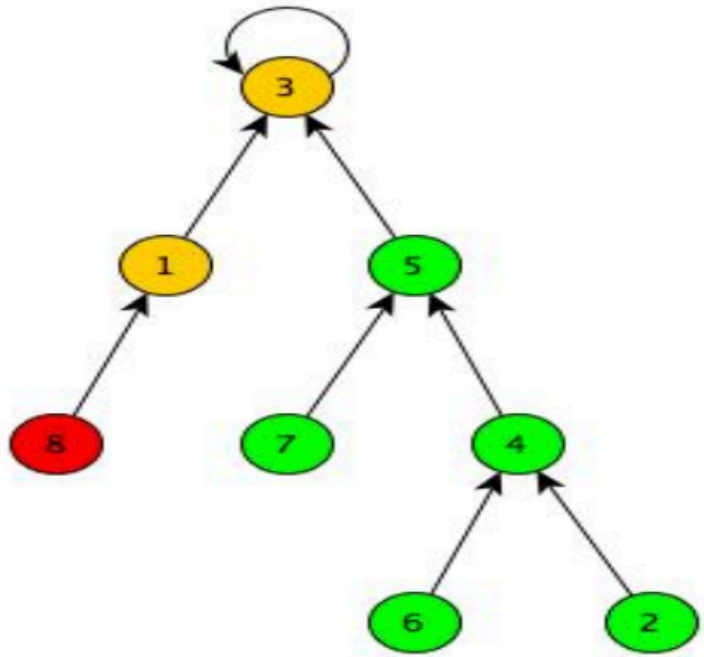
ESERCIZIO:

Dato un albero di n nodi rappresentato tramite il vettore dei padri P (per covenzione il padre del nodo radice è il nodo stesso) e un suo nodo x, dare lo pseudocodice di un algoritmo che in tempo $O(n)$ produce la lista dei nodi di T presenti nel sottoalbero radicato in x.

Ad esempio per

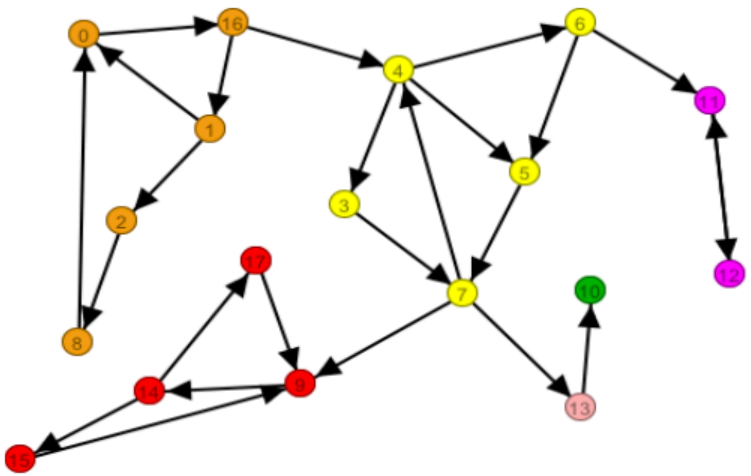
$$P = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 3 & 4 & 3 & 5 & 3 & 4 & 5 & 1 \\ \hline \end{array}$$

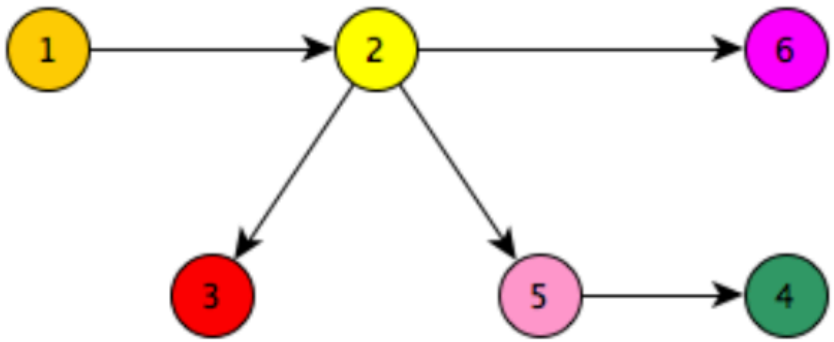
e $x = 5$ deve restituire l'insieme $\{2, 4, 5, 6, 7\}$ mentre per $x = 8$ deve restituire $\{8\}$



ESERCIZIO:

Dato un grafo diretto G, si definisce grafo delle parti il grafo G' che contiene un vertice per ogni componente fortemente connessa di G e tra due suoi nodi a e b c'è un arco che va da a a b se in G è possibile andare da un nodo della componente fortemente connessa corrispondente ad a ad un nodo della componente fortemente connessa corrispondente a b.





- Descrivere un algoritmo che, a partire dal grafo diretto G, costruisce il suo grafo delle parti G' in $O(n+m)$ tempo.

- Dimostrare che il grafo delle parti è sempre un DAG (vale a dire un grafo diretto aciclico).