Corso di laurea in Informatica
Progettazione d'algoritmi

I grafi 6

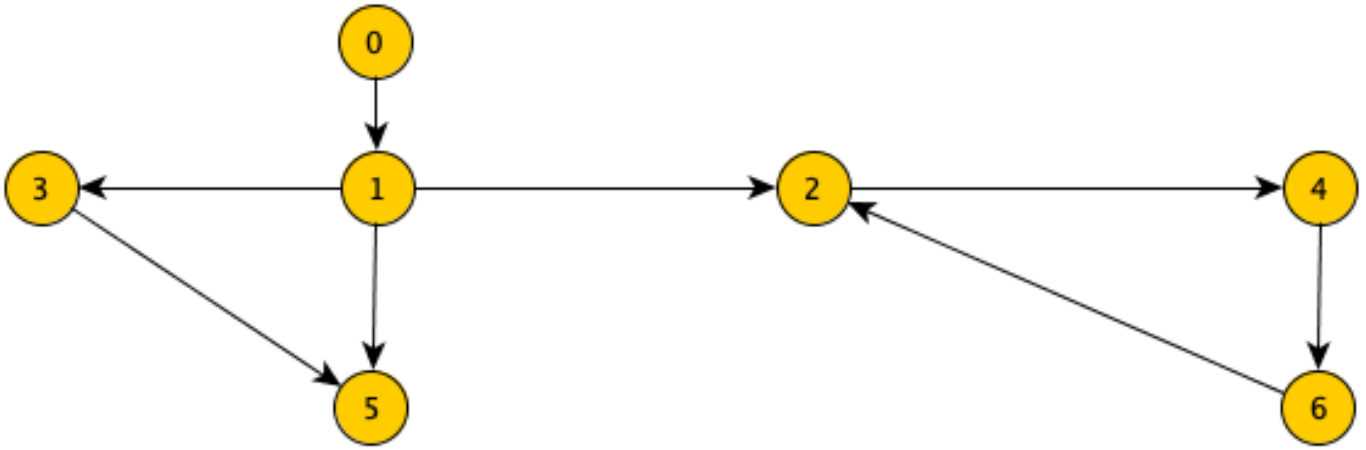
Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

CICLI

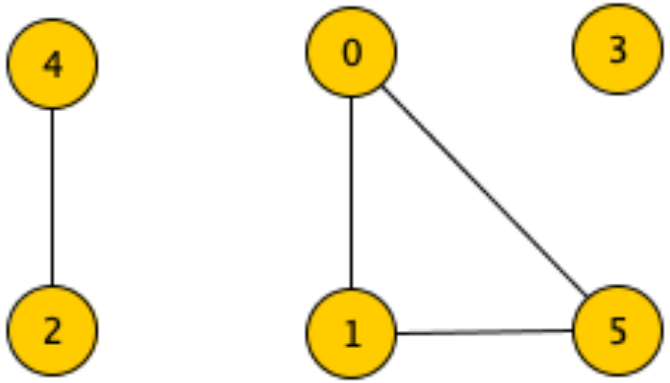
Dato un grafo G (diretto o non diretto) ed un suo nodo u vogliamo sapere se da u è possibile raggiungere un ciclo in G

G=




```
>>> ciclo(G, 1)
True
>>>ciclo(G, 3)
False
```

G1=




```
>>> ciclo(G1, 5)
True
>>>ciclo(G1, 4)
False
```

L'idea di partenza (**sbagliata**) è:
visita il grafo, e se nel corso della visita incontri un nodo già visitato interrompila e restituisci True, se al contrario la visita termina regolarmente restituisci False:

```
def ciclo(u,G):
    visitati = [0] * len(G)
    return DFSr(u, G, visitati)
```

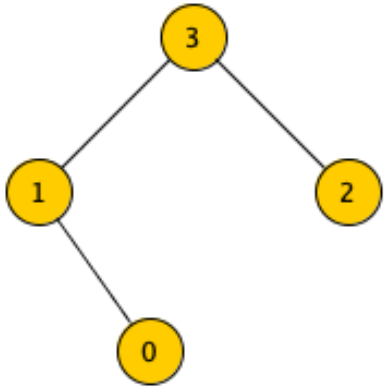
```
def DFSr(u,G,visitati):
    '''restituisce True se nella visita da u si
    incontra nodo già visitato, False altrimenti'''
    visitati[u] = 1
    for v in G[u]:
        if visitati[v]==1:
            return True
        if DFSr(v, G, visitati):
            return True
    return False
```

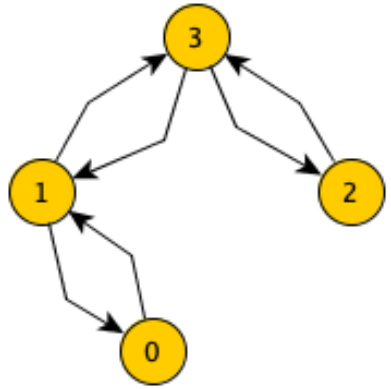
Il codice è scorretto nel caso di grafi non diretti:

```python
def ciclo(u,G):
    visitati = [0] * len(G)
    return DFSr(u, G, visitati)
```

```python
def DFSr(u,G,visitati):
    ''''restituisce True se nella visita da u si
    incontra nodo già visitato, False altrimenti'''
    visitati[u] = 1
    for v in G[u]:
        if visitati[v]==1:
            return True
        if DFSr(v, G, visitati):
            return True
    return False
```

```
G=[
    [1],
    [0, 3],
    [3],
    [1, 2]
  ]
```





Il codice è sbagliato perché (per come è codificato il grafo non diretto) se x ha un arco che lo collega a y nella lista di adiacenza di x è presente y e nella lista di adiacenza di y è presente x. Questo viene interpretato erroneamente come un ciclo di lunghezza 2. (La procedura proposta termina quindi sempre con True).

Per risolvere il problema, durante la visita alla ricerca del ciclo, devo distinguere nella lista di adiacenza di ciascun nodo y che incontro il nodo x che mi ha portato a visitarlo (il padre di y nell'albero DFS).

Versione corretta:

```
def ciclo(u,G):
    visitati = [0] * len(G)
    return DFSr(u, u, G, visitati)

def DFSr(u, padre, G, visitati):
    ''''restituisce True se nella visita da u si
    incontra nodo già visitato diverso dal padre,
    False altrimenti'''
    visitati[u] = 1
    for v in G[u]:
        if visitati[v]==1:
            if v != padre:
                return True
        else:
            if DFSr(v, u, G, visitati):
                return True
    return False
```

```python
def ciclo(u,G):
    visitati = [0] * len(G)
    return DFSr(u, u, G, visitati)
```

```python
def DFSr(u, padre, G, visitati):
    '''restituisce True se nella visita da u si
    incontra nodo già visitato diverso dal padre,
    False altrimenti'''
    visitati[u] = 1
    for v in G[u]:
        if visitati[v]==1:
            if v != padre:
                return True
        else:
            if DFSr(v, u, G, visitati):
                return True
    return False
```

Complessità $O(n)$:

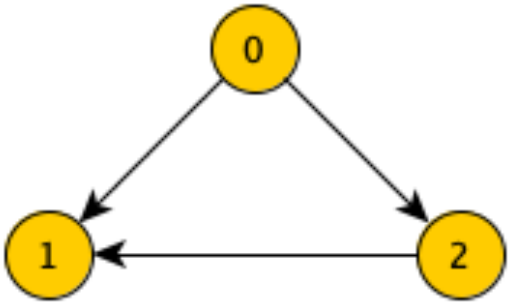
Nota che la complessità $O(n)$ è dovuta al fatto che se il grafo non contiene cicli allora ha al più n-1 archi e quindi $O(n+m) = O(n)$. Se al contrario il grafo contiene cicli se ne scopre uno dopo aver considerato al più n archi e a quel punto la procedura termina la visita.

Il codice è scorretto nel caso di grafi diretti:

```
def ciclo(u,G):
    visitati = [0] * len(G)
    return DFSr(u, G, visitati)
```

```
def DFSr(u,G,visitati):
    ''''restituisce True se nella visita da u si
    incontra nodo già visitato, False altrimenti'''
    visitati[u] = 1
    for v in G[u]:
        if visitati[v]==1:
            return True
        if DFSr(v, G, visitati):
            return True
    return False
```

```
G=[
    [ 1, 2 ],
    [   ],
    [ 1 ]
  ]
```



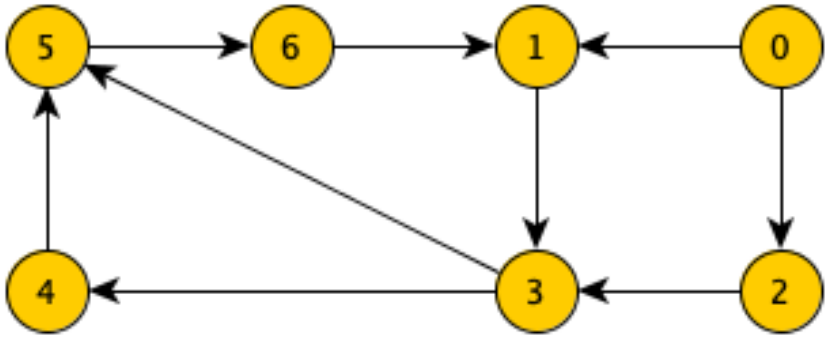

```
>>>ciclo(0,G)
True
```

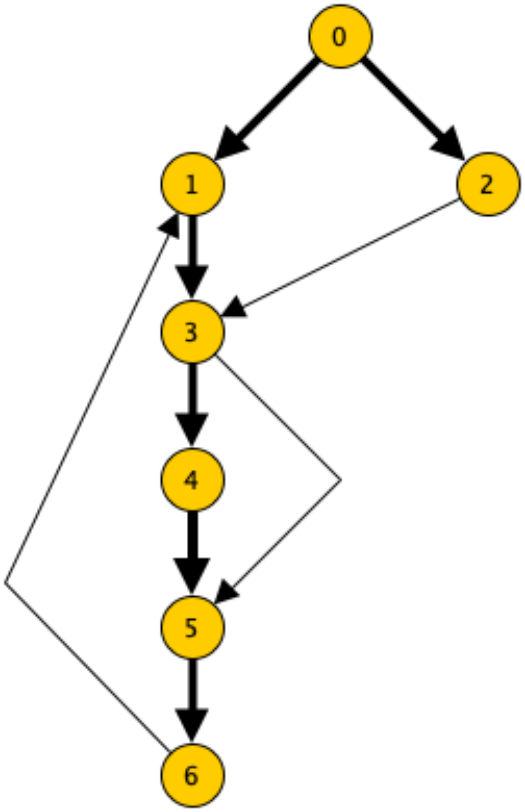
Il codice è sbagliato perché incontrare in un grafo diretto un nodo già visitato non significa necessariamente che si è in presenza di un ciclo (la procedura può terminare con *True* anche in assenza di ciclo)

Durante la visita DFS posso incontrare nodi già visitati in tre modi diversi:

- **archi in avanti** (cioè le frecce dirette da un antenato a un discendente),
- **archi all'indietro** (cioè le frecce dirette da un discendente ad un antenato),
- **archi di attraversamento**.

Solo la presenza di archi all'indietro testimonia la presenza di un ciclo.





6 – 1 è arco all'indietro
2 – 3 è arco di attraversamento
3 – 5 è arco in avanti

Solo la presenza di archi all'indietro testimonia la presenza del ciclo.
Per risolvere il problema, durante la visita DFS alla ricerca del ciclo, devo poter distinguere la scoperta di nodi già visitati grazie ad un arco all'indietro dagli altri.
Posso individuare i visitati da archi all'indietro notando che solo nel caso di archi all'indietro la visita del nodo già visitato ha terminato la sua ricorsione.

IDEA: per il vettore V dei visitati uso tre step:

- In V un nodo vale 0 se il nodo non è stato ancora visitato
- In V un nodo vale 1 se il nodo è stato visitato ma la ricorsione su quel nodo non è ancora finita,
- In V un nodo vale 2 se il nodo è stato visitato e la ricorsione su quel nodo è finita.

In questo modo scopro un ciclo quando trovo un arco diretto verso un nodo già visitato che si trova nello stato 1.

Versione corretta di complessità $O(n+m)$:

```
def DFSr(u, G, visitati):
    '''Restituisce True se la DFS da u trova
    un ciclo (arco all'indietro).'''
    visitati[u] = 1
    for v in G[u]:
        if visitati[v] == 1:
            # Nodo nello stato "in elaborazione" → ciclo trovato
            return True
        if visitati[v] == 0:
            # Se il nodo non è stato visitato, continua la DFS
            if DFSr(v, G, visitati):
                return True
    visitati[u] = 2  # Nodo completamente esplorato
    return False
```

```
def cicloD(u, G):
    visitati = [0]*len(G)
    return DFSr(u, G, visitati)
```

Se voglio sapere se un grafo (diretto o nondiretto) contiene un ciclo o meno, devo visitarlo tutto non importa il punto da cui parto.
Non è difficile modificare le procedure appena viste senza alterarne la complessità $O(n+m)$.

Di seguito la procedura modificata nel caso di grafi diretti:

```
def cicloD(G):
    '''Restituisce True se il grafo diretto G contiene
    un ciclo.'''
    # 0: non visitato, 1: in elaborazione, 2: completato
    visitati = [0] * len(G)
    # Controlla tutti i nodi del grafo
    for u in range(len(G)):
        if visitati[u] == 0:
            # Avvia DFS solo dai nodi non ancora visitati
            if DFSr(u, G, visitati):
                return True
    return False
```

Corso di laurea in Informatica
Introduzione agli Algoritmi

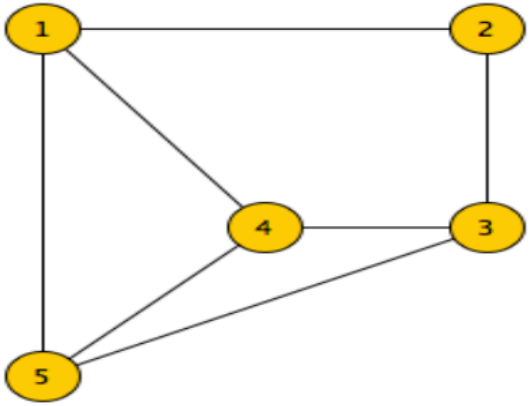
Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZI:

1. Progettare un algoritmo che, dato un grafo diretto G, restituisce in tempo $O(n+m)$ una tripla di interi con nell'ordine: il numero di archi in avanti, il numero di archi all'indietro ed il numero di archi di attraversamento che si incontrano durante una sua visita a partire dal nodo 0.

2. Progettare un algoritmo che, dato un grafo connesso G, restituisce in tempo $O(m)$ un cammino che attraversa tutti gli archi di G una e una sola volta in entrambe le direzioni, (i nodi possono essere toccati anche più volte).



Ad esempio per il grafo in figura che ha 7 archi una possibile soluzione è il seguente cammino di lunghezza 14:

$1-4-5-4-1-5-1-2-3-4-3-5-3-2-1$