Corso di laurea in Informatica
Progettazione d'algoritmi
Didattica blended

I grafi

Angelo Monti

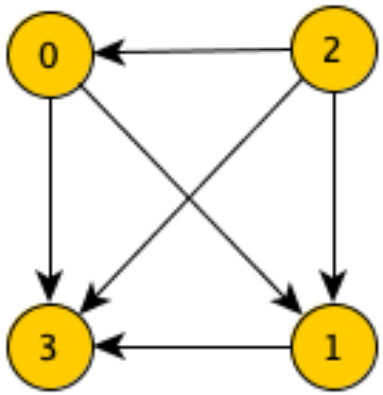
SAPIENZA
UNIVERSITÀ DI ROMA

Problema:

Abbiamo un lavoro da compiere. Il lavoro consiste in un insieme di n compiti. Tra certe coppie di compiti c'è dipendenza: la coppia (a, b) sta ad indicare che l'esecuzione del compito b richiede la terminazione del compito a.
Vogliamo sapere se è possibile completare il lavoro e, nel caso la risposta sia positiva, vogliamo un ordinamento con cui eseguire i compiti in modo da rispettare le dipendenze.

Possiamo rappresentare questa situazione come un grafo diretto G dove i nodi sono i diversi compiti e per ogni dipendenza (a, b) c'è in G un arco diretto dal nodo a al nodo b.

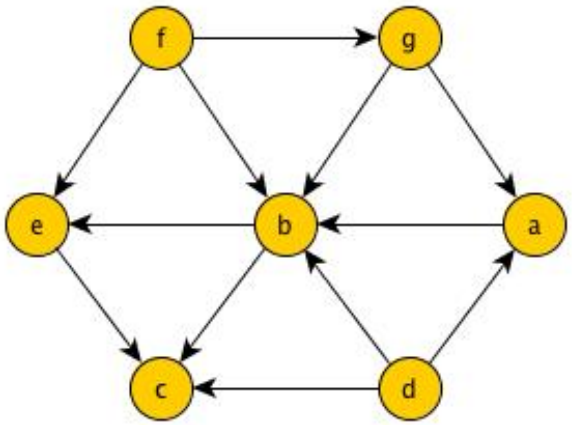
$$\{(0,1), (0,3), (1,3), (2,1), (2,3)\}$$

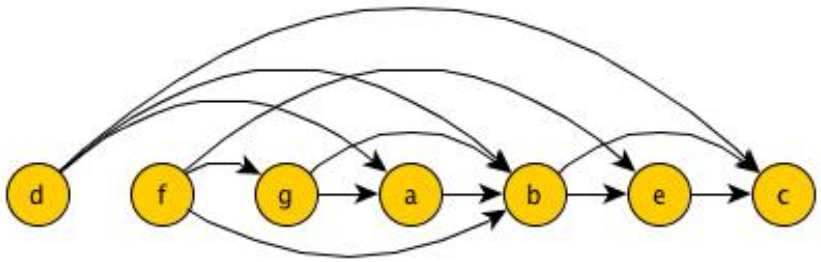


Ordinamento Topologico

Spesso un grafo diretto cattura relazioni di propedeuticità (un arco da a a b indica che a è propedeutico a b).

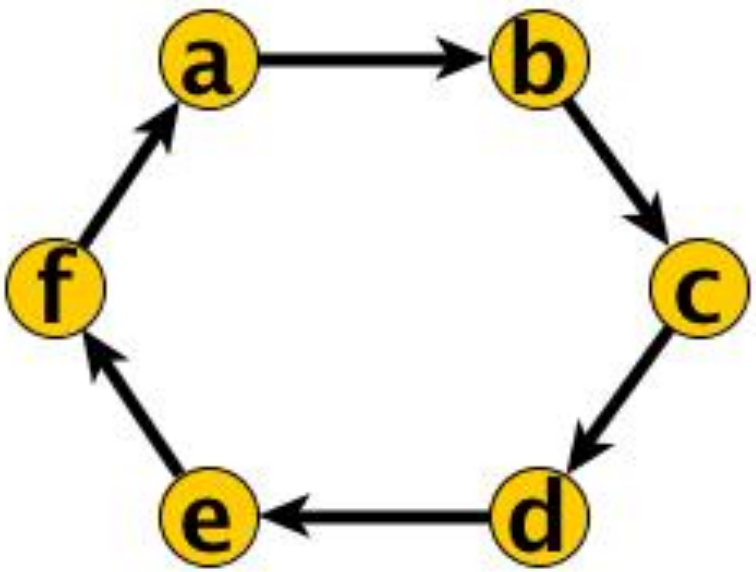
Potrò rispettare tutte le propedeuticità se riesco ad ordinare i nodi del grafo in modo che gli archi (le frecce) vadano tutti da sinistra verso destra. Questa ordinamento è detto ordinamento topologico.





- Un grafo diretto può avere da 0 ad $n!$ ordinamenti topologici
- Un algoritmo esaustivo per il problema (che genera sistematicamente i differenti ordinamenti dei nodi del grafo e per ciascuno di questi testa se il vincolo sugli archi risulta soddisfatto) ha complessità $\Omega(n!)$ ed è quindi improponibile.

Perché G possa avere un ordinamento topologico è necessario che sia un DAG (vale a dire un grafo diretto aciclico).



La presenza di un ciclo nel grafo implica che nessuno dei nodi del ciclo possa comparire nell'ordine giusto. Ognuno di essi richiede di apparire nell'ordinamento alla destra di chi lo precede.

Mostreremo ora che: la condizione che il grafo risultante dalla modellizzazione sia un DAG è anche sufficiente perché l'ordinamento topologico esista.

Un DAG ha sempre un nodo sorgente (un nodo in cui non entrano archi).

Grazie alla proprietà appena richiamata posso costruire l'ordinamento topologico dei nodi del DAG in questo modo:

- inizio la sequenza dei nodi con una sorgente,

- cancello dal DAG quel nodo sorgente e le frecce che partono da lui (che non creeranno problemi nel seguito) ottengo un nuovo DAG,

- itero questo ragionamento finché non ho sistemato in ordine lineare tutti i nodi.

```python
def SortTop(G):
    ''' restituisce un sort topologico ST di G se esiste
        altrimenti restituisce la lista vuota'''
    gradoEnt=[ 0 for _ in G ]
    for u in G:
        for v in G[u]:
            gradoEnt[v]+=1
    sorgenti= { u for u in G if gradoEnt[u]==0}
    ST= [ ]
    while sorgenti:
        u=sorgenti.pop()
        ST.append(u)
        for v in G[u]:
            gradoEnt[v]-=1
            if gradoEnt[v]==0: sorgenti.add(v)
    if len(ST)==len(G): return ST
    return []
```

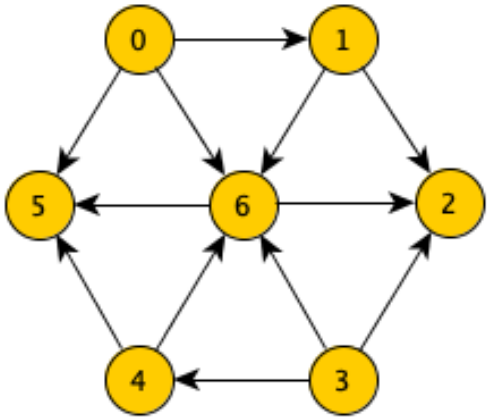
Complessità:

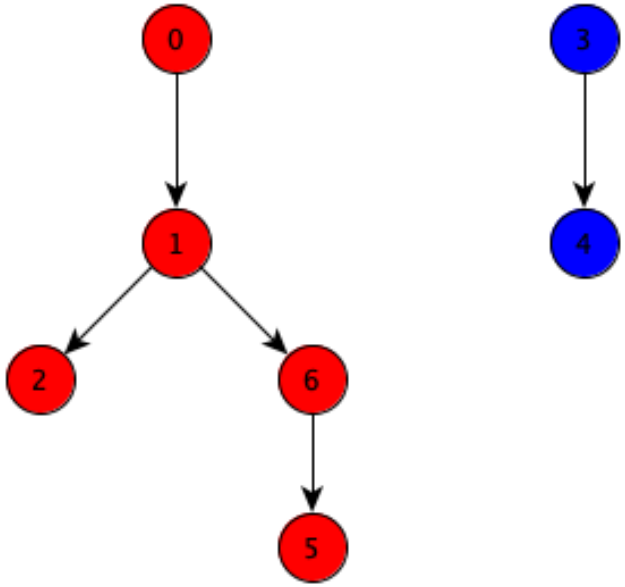
- inizializzare il vettore dei gradi entranti costa $O(n+m)$
- inizializzare l'insieme delle sorgenti costa $O(n)$
- il *while* viene iterato $O(n)$ volte e il costo totale del *for* al termine del while è $O(m)$.

Il costo dell'algoritmo è $O(n+m)$

Algoritmo alternativo basato sulla visita DFS del DAG

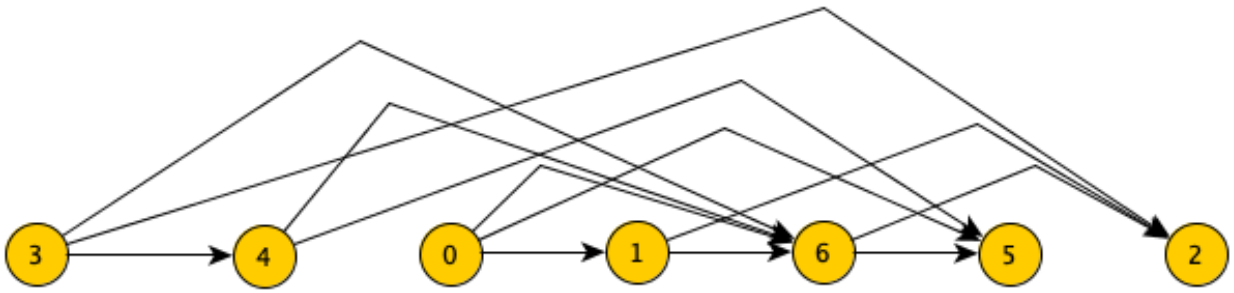
- effettua una visita del DAG a partire dal nodo 0
- man mano che termina la visita dei vari nodi, inseriscili in una lista
- restituisci come ordinamento dei nodi il *reverse* della lista.





lista=[2, 5, 6, 1, 0, 4, 3]

lista=[3, 4, 0, 1, 6, 5, 2]



- **effettua una visita del DAG a partire dal nodo 0**
- **man mano che termina la visita dei vari nodi, inseriscili in una lista**
- **restituisci come ordinamento dei nodi il *reverse* della lista.**

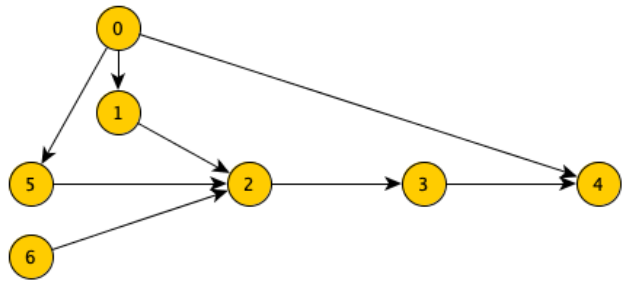
La prova di correttezza: Siano x e y due nodi in G, con arco che va da x a y. Consideriamo i due possibili casi e facciamo vedere che in entrambi i casi nella lista, prima di effettuare il reverse, y precede x.

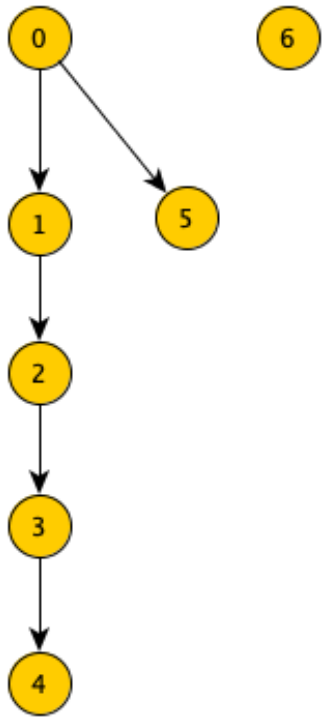
1. *l'arco (x, y) viene attraversato durante la visita.* In questo caso banalmente la visita di y finisce prima della visita di x e y finisce nella lista prima che ci finisca x.

2. *l'arco (x, y) NON viene attraversato durante la visita.* Durante la visita di x il nodo y é gia stato visitato e la sua visita è anche gia terminata (infatti da y non c'è un cammino che porta a x , in caso contrario nel DAG ci sarebbe un ciclo), anche in questo caso y finisce nella lista prima che ci finisca x.

```python
def sortT(G):
    '''restituisce il sort topologico di un grafo
       aciclico G'''
    def DFSr(x,G,visitati,lista):
        visitati[x]=1
        for y in G[x]:
            if visitati[y]==0:
                DFSr(y,G,visitati,lista)
        lista.append(x)
    ####
    visitati=[0 for _ in G]
    lista=[]
    for x in range(len(G)):
        if visitati[x]==0:
            DFSr(x,G,visitati,lista)
    lista.reverse()
    return lista

G={0:[1,4,5],1:[2],2:[3],3:[4],4:[],5:[2],6:[2]}
>>> sortT(G)
[6, 0, 5, 1, 2, 3, 4]
```





Complessità: $O(n+m)+O(n)=O(n+m)$

ordine di fine visita: 4, 3, 2, 1, 5, 0, 6