Applicazioni della visita in profondità – Parte II

Algoritmi 2 – Lezione 5

A. Monti
Sapienza Università di Roma

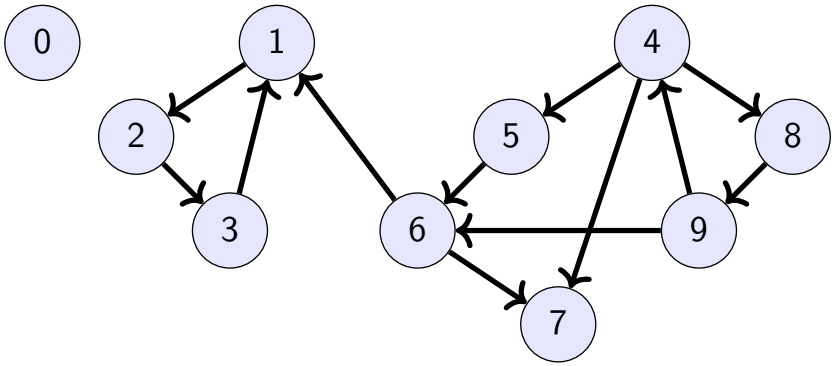
Corso: Algoritmi 2

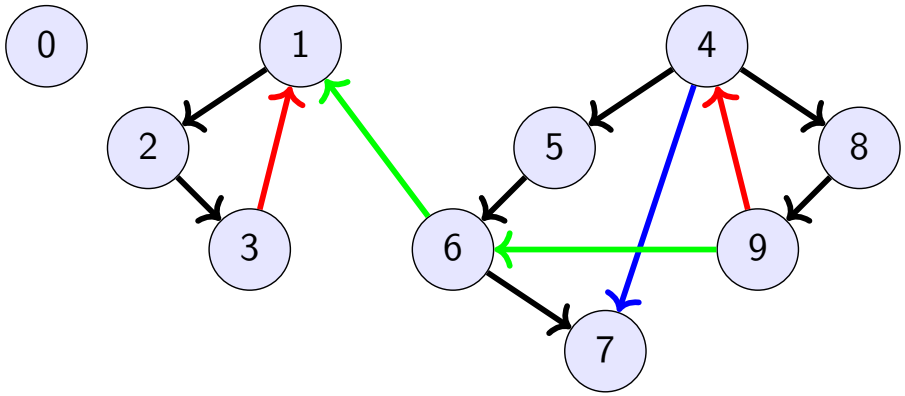
Indice

Classificazione degli Archi dopo una DFS

- Dopo una visita DFS, gli archi in un grafo diretto si classificano in quattro categorie in base alla loro relazione con l'albero di visita.
- Questa classificazione è cruciale per comprendere la struttura del grafo e rilevare proprietà importanti, come la presenza di cicli.
- Un arco diretto (u, v) può essere:
  - **Arco dell' Albero**: fa parte dell'albero di visita DFS.
  - **Arco all'Indietro**: connette un nodo u a un suo antenato v nell'albero DFS.
  - **Arco in Avanti**: connette un nodo u a un suo discendente v nell'albero DFS, ma v non è un figlio diretto di u.
  - **Arco di Attraversamento**: connette un nodo u a un nodo v che non è né un antenato né un discendente di u.

Esempio Visivo





In figura a sinistra un grafo G orientato e a destra la Foresta DFS con archi classificati. Archi all'indietro: rossi; Archi in avanti: blu; Archi di attraversamento: verdi.

Algoritmo lineare per Riconoscere gli Archi (1/3)

L'algoritmo usa due vettori: **Visitati e TempoVisita**.

- **Visitati[u]**:
  - 0: non visitato.
  - 1: in visita (nello stack di ricorsione).
  - 2: visita completata.
- **TempoVisita[u]**: il **tempo di scoperta**, ovvero il momento in cui u viene visitato per la prima volta.

La classificazione di un arco (u, v) avviene come segue:

- **Arco dell'Albero**: se $Visitati[v] == 0$.
- **Arco all'Indietro**: se $Visitati[v] == 1$.
- **Arco in Avanti**: se $Visitati[v] == 2$ **AND** $TempoVisita[u] < TempoVisita[v]$.
- **Arco di Attraversamento**: se $Visitati[v] == 2$ **AND** $TempoVisita[u] > TempoVisita[v]$.

Algoritmo lineare per Riconoscere gli Archi (2/3)

Vedremo ora l'algoritmo che durante la visita DFS del grafo conta i vari tipi di archi che incontra.

L'algoritmo restituirà il vettore *lista* dove:

- *lista*[0] è il numero di archi della foresta
- *lista*[1] è il numero di archi all'indietro
- *lista*[2] è il numero di archi in avanti
- *lista*[3] è il numero di archi di attraversamento

Implementazione: Algoritmo lineare per Riconoscere gli Archi (3/3)

```
def conta_tipologie(G):
    n = len(G)
    Visitati = [0] * n
    TempoVisita = [-1] * n
    tempo = [0]
    Lista = [0, 0, 0, 0]
    for i in range(n):
        if Visitati[i] == 0:
            dfs_util(i, G, Visitati, TempoVisita, tempo, Lista)
    return Lista

def dfs_util(u, G, Visitati, TempoVisita, tempo, Lista):
    Visitati[u] = 1
    TempoVisita[u] = tempo[0]
    tempo[0] += 1
    for v in G[u]:
        if Visitati[v] == 0:
            Lista[0] += 1
            dfs_util(v, G, Visitati, TempoVisita, tempo, Lista)
        elif Visitati[v] == 1:
            Lista[1] += 1
        else:
            if TempoVisita[u] < TempoVisita[v]:
                Lista[2] += 1
            else:
                Lista[3] += 1
    Visitati[u] = 2
```

Rilevazione di cicli in grafi orientati

- Rilevare cicli è un problema fondamentale nell'analisi dei grafi orientati.

  Un esempio pratico è la **pianificazione di compiti** con dipendenze:

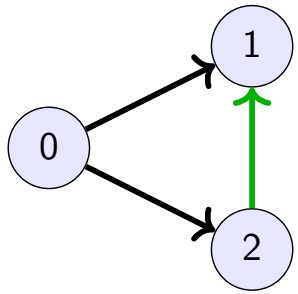
  - In questo contesto si costruisce un **grafo delle dipendenze**: ogni nodo rappresenta un compito da eseguire, mentre un arco diretto (u, v) indica che il compito u deve essere completato prima che il compito v possa iniziare.
  - Se il grafo delle dipendenze contiene un ciclo, si ha un'incoerenza logica: ogni compito nel ciclo dipende (direttamente o indirettamente) da sé stesso. Di conseguenza nessuno di questi compiti può essere iniziato o completato, creando un "circolo vizioso" e un punto morto permanente.

Un grafo diretto senza cicli è chiamato **DAG** (Directed Acyclic Graph).

La Strategia ingenua

- Una strategia ingenua è: *effettua una visita DFS e se nel corso della visita ci si imbatte in un arco che punta a un nodo già visitato, allora c'è un ciclo*.

Questa strategia è **errata**, perché non distingue tra i diversi tipi di archi. In figura abbiamo un grafo orientato senza cicli. La strategia ingenua rileverebbe erroneamente un ciclo.



La Soluzione Corretta

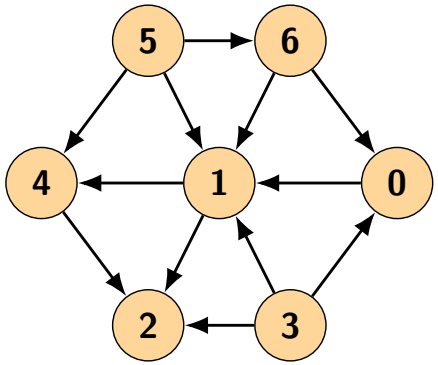
- La vera condizione per la presenza di un ciclo in un grafo orientato, rilevabile tramite DFS, è l'incontro di un **arco all'indietro**.
- Un arco all'indietro (u, v) indica che v è un antenato di u nell'albero DFS e, dato che v è ancora "in visita", esiste un percorso da v a u che chiude un ciclo.

Implementazione: Algoritmoper la Rilevazione dei Cicli

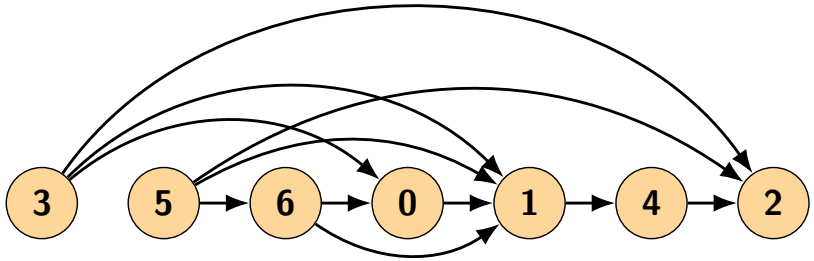
```
def cicliDiretti(G):
    '''
    viene restituito  True se il grafo contiene almeno un ciclo
    '''
    n = len(G)
    Visitati = [0] * n
    for i in range(n):
        if Visitati[i] == 0:
            if dfs_back(i, G, Visitati):
                return True
    return False

def dfs_back(u, G, Visitati):
    Visitati[u] = 1
    for v in G[u]:
        if Visitati[v] == 0:
            if dfs_back(v, G, Visitati):
                return True
        elif Visitati[v] == 1:
            # trovato un arco all'indietro che
            # testimonia la presenza di un ciclo
            return True
    Visitati[u] = 2
    return False
```

Ordinamento Topologico (definizione)

- Un ordinamento topologico è una sequenza lineare dei nodi di un grafo diretto.
- Per ogni arco diretto (u, v), il nodo u deve comparire prima di v nella sequenza.



(a) Un grafo G



(b) Un ordinamento topologico di G

Algoritmo delle Sorgenti

- Un grafo ammette un ordinamento topologico **solo se è un DAG**.
- Un DAG ha sempre almeno una **sorgente**, cioè un nodo privo di archi entranti.

- L'idea dell'algoritmo è costruire l'ordinamento topologico **un nodo alla volta**.
- Ad ogni passo si seleziona una **sorgente** del grafo e la si aggiunge all'ordinamento.
- Il nodo scelto viene quindi rimosso dal grafo insieme a tutti i suoi archi uscenti.
- Poiché la rimozione di nodi non può creare cicli, se il grafo iniziale era un DAG anche il grafo rimanente sarà ancora un DAG e quindi conterrà una nuova sorgente.
- Il procedimento si ripete finché possibile.
- Alla fine possono verificarsi due casi:
  - tutti i nodi sono stati rimossi $\Rightarrow$ è stato costruito un ordinamento topologico;
  - rimangono nodi ma non esistono più sorgenti $\Rightarrow$ il grafo contiene un ciclo.

Implementazione lineare dell'Algoritmo delle Sorgenti

```
def sortTop(G):
    n = len(G)
    GradoEnt = [0] * n
    for i in range(n):
        for j in G[i]:
            GradoEnt[j] += 1
    Sorgenti = [i for i in range(n) if GradoEnt[i] == 0]
    ST = []
    while Sorgenti:
        u = Sorgenti.pop( )
        ST.append(u)
        for v in G[u]:
            GradoEnt[v] -= 1
            if GradoEnt[v] == 0:
                Sorgenti.append(v)
    if len(ST) == n:
        return ST
    return []
```

Complessità dell'Algoritmo delle Sorgenti

- Sia n il numero di nodi e m il numero di archi del grafo.
- Il calcolo iniziale dei gradi entranti visita tutte le liste di adiacenza:

$$O(n+m)$$

- Ogni nodo viene inserito ed estratto dalla lista delle sorgenti al più una volta:

$$O(n)$$

- Ogni arco (u, v) viene esaminato una sola volta quando si rimuove il nodo u:

$$O(m)$$

- Sommando i contributi:

$$O(n+m)$$

- Quindi l'algoritmo calcola un ordinamento topologico in **tempo lineare nella dimensione del grafo**.

Algoritmo alternativo basato sulla DFS

Il calcolo di un ordinamento topologico in un DAG può essere elegantemente risolto anche utilizzando una visita in profondità (DFS).

- Si esegue la DFS del grafo e si aggiungono i nodi a una lista man mano che le loro visite terminano.
- L'ordinamento topologico è l'inverso di questa 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.

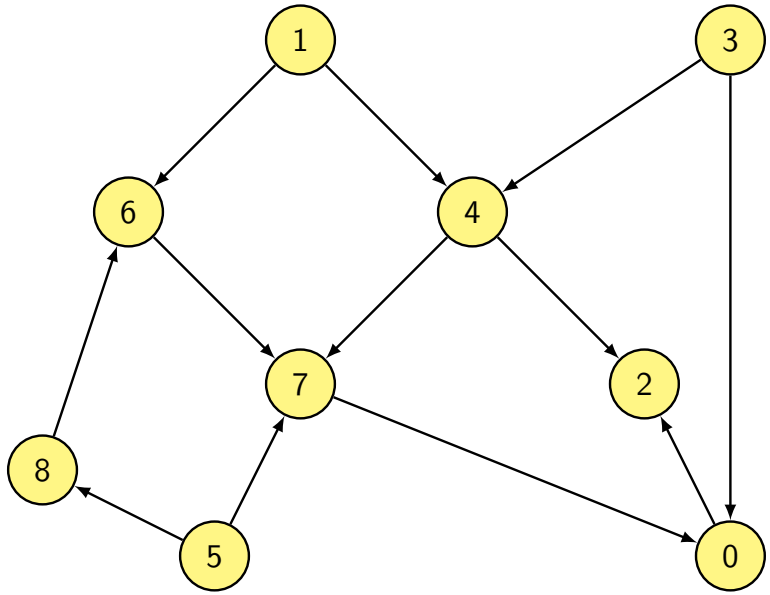
1. l'arco (x, y) viene attraversato durante la visita. In questo caso banalmente la visita di y finisce prima della visita di x e y finisce nella lista prima che ci finisca x.
2. l'arco (x, y) NON viene attraversato durante la visita. Durante la visita di x il nodo y é gia stato visitato e la sua visita 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.

Implementazione lineare dell' Algoritmo alternativo

```
def sortTop1(G):
    visitati = [False] * len(G)
    lista = []
    for u in range(len(G)):
        if visitati[u] == False:
            DFSr(u, G, visitati, lista)
    lista.reverse()
    return lista


def DFSr(u, G, visitati, lista):
    visitati[u] = True
    for v in G[u]:
        if visitati[v] == False:
            DFSr(v, G, visitati, lista)
    lista.append(u)
```

Problema

Problema: Dato un DAG e un nodo x, trovare le sorgenti (vale a dire nodi con grado entrante nullo) da cui è possibile raggiungere x.



Ad esempio per il grafo G in figura:

- per $x = 7$ va restituita la lista $[1, 3, 5]$,
- per $x = 4$ va restituita la lista è $[1, 3]$.

Strategia Inefficiente

- **Approccio ingenuo**: Per ogni nodo sorgente, eseguire una DFS per vedere se x è raggiungibile.
- **Complessità**: Questo approccio ha una complessità di $O(n \cdot (n+m))$, che è troppo lenta.
- Serve una strategia più efficiente, che eviti di ripetere la ricerca per ogni sorgente.

La Soluzione Efficiente e il Grafo Trasposto

- La soluzione efficiente richiede di usare il **grafo trasposto** (G^T).
- Il grafo trasposto ha gli stessi nodi di G ma con la direzione degli archi invertita.
- I nodi che raggiungono x in G sono gli stessi che possono essere raggiunti da x nel grafo trasposto G^T.
- Le sorgenti di G sono i nodi con grado di uscita zero in G^T.
- Il problema si riduce ad effettuare una singola visita a partire da x nel grafo G^T alla ricerca dei pozzi raggiungibili.
- Questo approccio ha una complessità ottimale di $O(n + m)$.

Implementazione dell'Algoritmo Efficiente in $O(n + m)$

```
def trova_sorgenti(G, x):
    n = len(G)
    # Costruisci il grafo trasposto
    GT = Trasposto(G)
    # Esegui la visita di GT a partire  da x e
    # restituisci il vettore caratteristico
    # dei visitati
    visitati = [False] * n
    DFS(x, GT, visitati)
    lista = [ ]
    for u in range(n):
        # Un nodo è sorgente in G se non ha archi
        # uscenti nel grafo trasposto GT
        if GT[u]==[ ] and visitati[u]:
            lista.append(u)
    return lista
```

ESERCIZI

Suggerimento: Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto.
Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

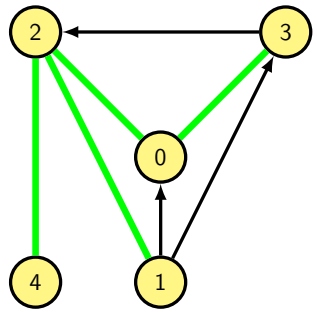
Esercizi

1. Dimostrare che in un DAG è sempre presente almeno un nodo senza archi entranti (sorgente) e un nodo senza archi uscenti (pozzo).

2. Contare quanti sono al massimo gli archi diretti che si possono aggiungere a un DAG di n nodi e m archi senza creare cicli.

3. Modificando opportunamente il codice della ricerca di un ordinamento topologico tramite visita DFS, progettare una procedura che, dato un grafo G, in tempo $O(n + m)$ restituisca un ordinamento topologico se il grafo è aciclico, o una lista vuota in caso contrario.

4 Un grafo si dice **parzialmente orientato** se contiene sia archi orientati sia archi non orientati.
Nella figura che segue è mostrato un esempio di grafo parzialmente orientato, con gli archi non orientati evidenziati in verde.



Dimostrare che per un grafo parzialmente orientato G privo di cicli orientati è sempre possibile assegnare una direzione agli archi non orientati in modo da ottenere un DAG.
La figura che segue mostra un DAG che è possibile ottenere orientando gli archi del grafo parzialmente orientato precedente.