# Esercitazione 2: More Depth-First Search

Giacomo Paesani

March 16, 2024

**Esercizio 1.** In un grafo non diretto e connesso $G = (V, E)$ un vertice $v$ si dice di *articolazione* (*cutvertex* in inglese) se $G - v$, il grafo ottenuto da $G$ rimuovendo $v$ e tutti gli archi ad esso incidenti, non è connesso. Modificare l'algoritmo della ricerca in profondità in maniera da poter ottenere tutti e soli i vertici di articolazione di un grafo connesso; è possibile fare questa modifica in modo che il controllo avvenga in $\Theta(|V| + |E|)$?

Come si può ulteriormente modificare l'algoritmo per ottenere tutti i vertici di articolazione di un grafo non necessariamente connesso (dove in questo caso il criterio di un vertice di articolazione è quello di disconnettere la componente connessa che lo contiene)?

**Soluzione 1.** E' necessario ricordare che nella classificazione degli archi di un grafo non diretto in seguito ad una ricerca in profondità possono essere presenti solo archi dell'albero o all'indietro. Come prima cosa osserviamo che essendo $G$ non diretto e connesso, ogni ricerca in profondità restituisce un singolo albero.

Prima di dare la soluzione dobbiamo fare alcune osservazioni su come individuare i vertici di articolazione durante la ricerca in profondità. Iniziamo considerando la radice $u$ dell'albero della ricerca, il vertice di $G$ da cui inizia la visita: $u$ è un vertice di articolazione di $G$ se e solo se ci sono due archi dell'albero incidenti ad $u$. Sia ora $u$ un vertice che non è la radice. Allora $u$ è un vertice di articolazione di $G$ se e solo se non esiste un arco all'indietro da un discendente di $u$ ad un antenato di $u$.

**Algorithm 1** DFS modificata per ricavare tutti i punti di articolazione di un grafo non diretto.

**Input:** grafo non diretto $G = (V, E)$.
**Output:** l'insieme $A$ dei vertici di articolazione.

```
global variables
    A ↤ coda, inizialmente vuota
    Color ↤ array di |V| elementi
    Parent ↤ array dei padri
    t ↤ array dei tempi di inizio visita
    T ↤ array dei tempi di fine visita
    time ↤ intero che simulta il tempo
end global variables
function DFS(G)
    for u ∈ V do
        Color[u]=BIANCO
    time = 0
    for u ∈ V do
        if Color[u]==BIANCO then
            b=DFS-Visit(G,u)
            Parent[u] = u
    return A
function DFS-Visit(G,u)
    time = time + 1
    t[u] = time
    back= time
    Color[u]=GRAY
    children=0
    for v ∈ Adj[u] do
        if Color[v]==BIANCO then
            Parent[v] = u
            children=children+1
            b=DFS-Visit(G,v)
            if t[u] > 1 and b ≥ t[u] then
                A.enqueue(u)
            back=min{back,b}
        if Color[v]!=BIANCO and v! = Parent[u] then
            back=min{back,t[v]}
    if t[u] == 1 and children≥ 2 then
        A.enqueue(u)
    Color[u]=NERO
    time = time + 1
    T[u] = time
    return back
```

L'Algoritmo 1 è una delle possibile soluzioni: il codice è inspirato all'algoritmo ricorsivo di ricerca in profondità proposto in [1]. Notiamo come tale algoritmo è stato modificato per fornire l'insieme dei punti di articolazione del grafo. In questa versione dell'algoritmo la funzione **DFS-Visit** con input $(G, u)$ ritorna un intero che indica il minimo tempo di inizio visita tra tutti vertici visitati dai discendenti di $u$.

La struttura dati coda $A$, che viene inizializzata come vuota nella linea 2, ha il ruolo di collezionare nel corso dell'algoritmo tutti i vertici di articolazione del grafo in esame. Consideriamo inizialmente la radice $u$ dell'albero; se, una volta terminata la visita, ci sono almeno due archi dell'albero incidenti ad $u$ (e quindi la condizione in linea 34 è soddisfatta) allora $u$ è di articolazione e viene aggiunto alla coda $A$ con la funzione $enqueue(u)$ (linea 35). La variabile *children* è definita per ogni vertice $u$ del grafo (ogni volta che viene chiamata la funzione **DFS-Visit**$(G, u)$) e viene incrementata (linea 27) ogni volta che da $u$ si visita per la prima volta un'altro vertice $v$; notiamo però che tale variabile è utilizzata solo per la radice dell'albero.

Sia ora $u$ un vertice diverso dalla radice dell'albero. Per determinare se $u$ è di articolazione o no, è necessario studiare gli archi all'indietro che originano dai discendenti di $u$. La variabile $b$ (calcolata in linea 28) indica il minimo tempo di visita tra i vertici visitati da $v$ e viene confrontata con il tempo di inzio visita di $u$: se la condizione in linea 29 è soddisfatta, e cioè se ogni arco all'indietro con origine da un discendante di $u$ ha come termine in $u$ o in un suo discendente, allora $u$ è di articolazione e viene aggiunto all'insieme $A$ in linea 30. Concludiamo notando che la variabile *back* ha la funzione di registrare il minimo tempo di visita tra i vertici visitati da $u$: è facile verificare che *back* assume valori minori o uguali di $t[u]$ (in linea 21). Questa variabile viene confrontata, e in caso aggiornata, con $b$ e $t[v]$ (linea 31 e linea 33, rispettivamente) ogni volta che viene visitato un vertice a partire da $u$, sia se questo vertice viene visitato per la prima volta o è già in visita. La seconda condizione in linea 32 è necessaria per verificare che $v$ non sia il padre di $u$.

Si può facilmente notare che l'algoritmo appena descritto è in grado di trovare tutti i vertici di articolazione anche per grafi non connessi, cioè quei vertici la cui rimozione disconnette la componente connessa che li contiene o equivalentemente aumenta il numero di componenti connesse del grafo. Infatti, grazie al **for** nella linea 13, la visita in profondità continua finchè termina la visita di tutti i vertici del grafo. $\square$

**Esercizio 2.** In un grafo non diretto $G$ un cammino *Hamiltoniano* è un cammino $P$ di $G$ che passa per ogni vertice di $G$ esattamente una volta. Provare o confutare la seguente affermazione: $G$ contiene un cammino Hamiltoniano se e solo se può essere prodotto un albero di ricerca di $G$ che è un cammino. Domanda bonus: che succede se invece del cammino Hamiltoniano, si ha un ciclo Hamiltoniano in $G$?

**Soluzione 2.** Iniziamo supponendo che esista un cammino Hamiltoniamo $P$ nel grafo $G$ con $n$ vertici. Possiamo *chiamare* i vertici di $G$ seguendo l'ordine dato dal cammino $P$: cioè $V(G) = \{v_1, \ldots, v_n\}$, dove $v_i$ precede $v_{i+1}$ in $P$, per ogni $i = 1, \ldots, n-1$. Allora si considerano le liste di adiacenza dei vertici in maniera che per il vertice $v_i$, il primo adiacente che si trova è sempre $v_{i+1}$ per ogni $i = 1, \ldots, n-1$. Consideriamo ora la ricerca in profondità partendo dal vertice $v_1$. Per costruzione, l'albero di ricerca di $G$ usa esattamente gli archi di $P$ e quindi tale albero di ricerca è proprio il cammino $P$.

Supponiamo ora che può essere prodotto un albero di ricerca per una ricerca in profondità di $G$ che è un cammino $P$. In particolare il cammino $P$ contiene ogni vertice di $G$ esattamente una volta. Allora, $P$ è un cammino Hamiltoniano di $G$.

Consideriamo invece il caso in cui $G$ ha un ciclo Hamiltoniano $C$. Si osserva facilmente che $C$ contiene un cammino Hamiltoniano $P$ ottenuto rimuovendo un qualsiasi arco da $C$. Allora, per quello che abbiamo mostrato prima si può produrre un albero di ricerca di $G$ che è il cammino $P$.

Il viceversa non è vero. Consideriamo il grafo $P$ non diretto che è composto da un singolo cammino. Una ricerca in profondità di $P$ con vertice iniziale scelto tra i due estremi di $P$ produce nesessariamente il cammino $P$. E' allo stesso modo chiaro che $G$ non ammette alcun ciclo Hamiltoniano.

**Esercizio 3** (22.3-1,[1])**.** Durante una visita in profondità, ad ogni vertice $u$ di un grafo viene associato un colore che varia nel tempo $t$ dell'algoritmo. Se $t < t[u]$ e cioè il vertice $u$ deve ancora essere stato visitato per la prima volta allora $u$ è colorato di BIANCO. Se $t[u] \leq t \leq T[u]$ e cioè il vertice $u$ è ancora in visita allora $u$ è colorato di GRIGIO. Infine, se $t > T[u]$ e cioè il vertice $u$ è stato già completamente visitato allora $u$ è colorato di NERO. Fare una tabella 3x3 con righe e colonne contrassegnate da BIANCO, GRIGIO e NERO. In ogni cella $(i, j)$, indicare se, in un alcun punto di una ricerca in profondità di un grafo diretto, è possibile avere un arco da un vertice di colore $i$ ad un vertice di colore $j$. Per ogni possibile arco, indicare se esso può essere:

- arco dell'albero,
- arco all'indietro,
- arco in avanti o
- arco di attraversamento.

Fare una seconda tabella per una ricerca in profondità di un grafo non diretto.

| | BIANCO | GRIGIO | NERO |
|---|---|---|---|
| BIANCO | | | |
| GRIGIO | | | |
| NERO | | | |

**Soluzione 3.** Per il caso di un grafo diretto avremo:

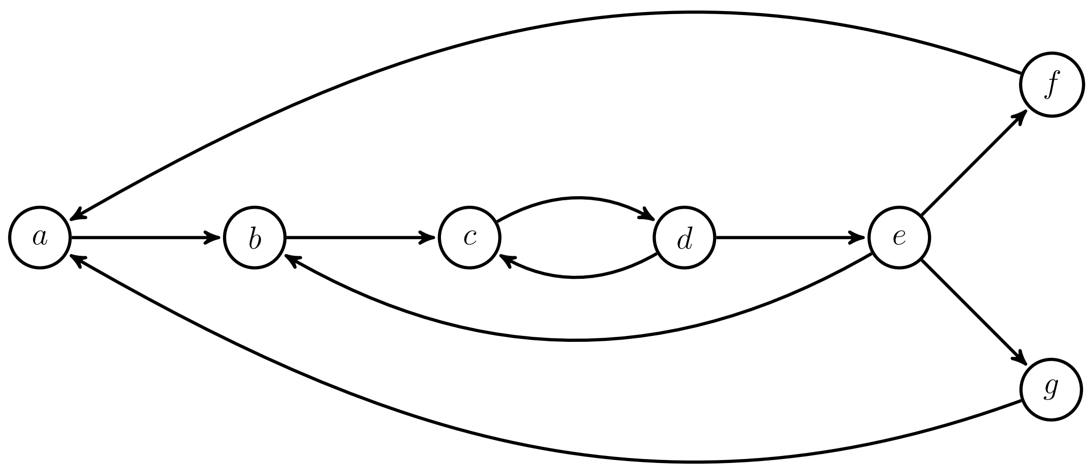
| | BIANCO | GRIGIO | NERO |
|---|---|---|---|
| BIANCO | Alb Die Ava Att | Die Att | Att |
| GRIGIO | Alb Ava | Alb Die Ava | Alb Ava Att |
| NERO | n/a | Die | Alb Die Ava Att |

Invece, per il caso di un grafo non diretto avremo:

| | BIANCO | GRIGIO | NERO |
|---|---|---|---|
| BIANCO | Alb Die | Alb Die | n/a |
| GRIGIO | Alb Die | Alb Die | Alb Die |
| NERO | n/a | Alb Die | Alb Die |

**Esercizio 4** (I. Salvo)**.** Sia $G$ il grafo raffigurato in figura. Determinare il minimo numero di archi che devono essere eliminati da $G$ affinchè $G$ ammetta ordinamenti topologici. Una volta rimosso questo insime minimo di archi, determinare tutti gli ordinamenti topologici di $G$.

**Soluzione 4.** Ricordiamo che un grafo diretto ammette un ordinamento topologico se e solo se è aciclico. Sia $C$ il ciclo di lunghezza due formato dagli archi $(c, d)$ e $(d, c)$. Questo ci permette di trarre due conclusioni. La prima è che è necessario rimuovere almeno un arco da $G$ per renderlo aciclico. Inoltre, ogni sottoinsieme di archi $A$ la cui rimozione rende $G$ senza cicli deve




contenere almeno uno tra $(c, d)$ e $(d, c)$. Supponiamo che $A$ contiene l'arco $(d, c)$; in questo caso è facile notare che esiste almeno un ciclo $C'$ di $G$ che non contiene l'arco $(d, c)$, ad esempio quello formato dagli archi $(b, c)$, $(c, d)$, $(d, e)$ e $(e, b)$. Allora $A$ deve anche contenere almeno uno degli archi di $C'$ e quindi $|A| \geq 2$. E' altrettanto immediato da osservare che se $A = \{(c, d)\}$, allora $G - A$, il grafo ottenuto da $G$ rimuovendo gli archi di $A$, è aciclico. Tutti i possibili ordinamenti topologici sono:

- $[d, e, f, g, a, b, c]$
- $[d, e, g, f, a, b, c]$ □

**Esercizio 5.** Fornire un algoritmo in pseudo-codice che dato un grafo diretto e aciclico $G = (V, E)$, restituisce un ordinamento topologico di $G$. E' possibile implementarlo in modo che il tempo di esecuzione sia $\Theta(|V| + |E|)$? Come dovrebbe essere modificato tale algoritmo per restituire l'elenco di tutti gli ordinamenti topologici di $G$? E' possibile fare questa ulteriore modifica mantenendo lo stesso tempo di esecuzione?

**Soluzione 5.** Prima di dare la soluzione dell'esercizio, è necessario ricordare il seguente fatto: un grafo diretto è aciclico se e solo se esiste un vertice del grafo con grado entrante nullo. L'idea della soluzione proposta è quella di calcolare e aggiornare dinamicamente i gradi entranti dei vertici del grafo: ad ogni passo viene selezionato un vertice di grado entrante nullo nel sottografo che comprende i vertici non ancora ordinati. Questa strategia ci permette di preservare il fatto che ad ogni iterazione la lista che sto creando è un ordinamento topologico *parziale*. La soluzione proposta nell'algoritmo 2 è inspirata all'algoritmo di Kahn [2] per trovare un ordinamento topologico in un grafo diretto e aciclico.

Come prima cosa specifichiamo il ruolo delle variabili globali. L'array $Ind$ di lunghezza $|V|$, uno per ogni vertice di $G$, ha lo scopo di salvare il grado uscente di ogni vertice e di modificarlo dinamicamente (inizializzato in linea 21). La pila $S$ (non è tanto importante che sia una pila) si prefige di accumulare tutti vertici che al momento hanno grado uscente nullo (inizializzato in linea 3). Infine la coda $L$ (qui si che è importante che sia una coda) accumula nel corso dell'algoritmo vertici presenti in $S$ (e che quindi hanno grado uscente nullo in quel momento) diventando alla fine un ordinamento topologico di $G$ (inizializzato in linea 4).

La funzione **ComputeDegr** con input un grafo $G$ (non necessariamente diretto e aciclico) ha il semplice scopo di modificare il vettore $Ind$ affinchè il valore di ogni coordinata sia uguale al grado entrante del vertice corrispondente in $G$. Questa basilare funzione necessita di tempo pari a $\Theta(m)$ e viene chiamata in linea 7 per inizializzare il vettore.

Il ciclo **for** in linea 8 ha l'obbiettivo di scorrere la lista dei vertici del grafo e pone quelli di grado entrante nullo nella pila $S$, cioè $S$ contiene i candidati ad essere i primi elementi nell'ordinamento topologico di $G$. Più in generale, in ogni istante dell'algoritmo, la pila $S$ contiene tutti i candidati ad essere i prossimi elementi nell'ordinamento topologico di $G$.

Supponiamo ora che $S$ è non vuoto e consideriamo l'elemento $u$ in cima ad $S$: come già detto, possiamo aggiungere $u$ in testa alla lista $L$ attraverso l'operazione $append(u)$ 13. Quello che ci resta da fare è aggiornare il valore del grado entrante dei restanti vertici di $G$; in particolare sono modificati (e ridotti di uno) solo i gradi entranti dei vertici adiacenti ad $u$. Se il grado entrante di un vertice $v$ diviene nullo, allora lo si aggiunge alla pila $S$.

**Algorithm 2** Algoritmo per calcolare un ordine topologico di un DAG.

**Input:** grafo diretto e aciclico $G = (V, E)$.
**Output:** un ordine topologico $L$.

1: **global variables**
2:     $Ind \hookleftarrow$ array dei gradi entranti dei nodi
3:     $S \hookleftarrow$ pila, inizialmente vuota
4:     $L \hookleftarrow$ lista, inizialmente vuota
5: **end global variables**
6: **function OrdTop**$(G)$
7:     ComputeDegr$(G)$
8:     **for** $u \in V$ **do**
9:         **if** $Ind[u] == 0$ **then**
10:             $S.push(u)$
11:     **while** $S$ not empty **do**
12:         $u \hookleftarrow S.pop()$
13:         $L.append(u)$
14:         **for** $v \in Adj[u]$ **do**
15:             $Ind[v] = Ind[v] - 1$
16:             **if** $Ind[v] == 0$ **then**
17:                 $S.push(v)$
18:     **return**
19: **function ComputeDegr**$(G)$
20:     **for** $u \in V$ **do**
21:         $Ind[u] = 0$
22:         **for** $v \in Adj[u]$ **do**
23:             $Ind[u] = Ind[u] + 1$
24:     **return**

Ci rimane da dimostrare che il ciclo **while** di linea 11 si ripete esattamente una sola volta per ogni vertice di $G$. Prima mostriamo che per ogni vertice $u \in V$ esiste un passo dell'algoritmo tale che $u$ è un'elemento della pila $S$. Se $u$ non ha archi entranti, allora viene aggiunto a $S$ in linea 9. Siano ora $v_1, \ldots, v_k$, l'insieme dei vertici di $G$ tali che $(v_i, u)$ è un arco di $G$, per $i = 1, \ldots, k$. Analizziamo i seguenti casi: (1) se per ogni $i = 1, \ldots, k$ esiste un passo dell'algoritmo tale che $v_i$ è un'elemento di $S$, allora $Ind[u]$ diventera uguale a zero e quindi $u$ viene aggiunto a $S$ in linea 16. In alternativa (2) se esiste un vertice $v$ diverso da $u$ tale che $(v, u)$ è un arco di $G$ e non c'è alcun passo dell'algoritmo tale che $v$ è un elemento di $S$. In questo secondo caso, la condizione (2) si ripete necessariamente anche a $v$, ad uno dei vertici adiacenti ad $v$ e così via, fino ad una ripetizione di uno di questi vertici: se $z$ è il primo vertice che si ripete in questa sequenza, allora abbiamo creato una sequenza orientata di archi da $z$ fino a $z$ da cui si può ottenere un ciclo di $G$, il che contraddice che $G$ è aciclico. Allora necessariamente si deve applicare il caso (1) e quindi $u$ viene aggiunto a $S$.

Ora sappiamo che ogni vertice $u \in V$ viene aggiunto ad $S$ ad un certo passo dell'algoritmo. Prima che $S$ sia vuoto c'è una iterazione del ciclo **while** di linea 11 dove $u$ è in cima ad $S$: in quella stessa iteazione $u$ viene rimosso da $S$ con l'operazione *pop*() e aggiunto ad $L$ con l'operazione *append*. E' anche facile mostrare che un vertice $u$ che è stato aggiunto a $L$ non può essere inserito nuovamente in $S$ e quindi neanche in $L$ un'ulteriore volta.

**Esercizio 6** (I. Salvo)**.** Descrivere in pseudo-codice un algoritmo che, dato un grafo non diretto $G$, descrivere un algoritmo che ne orienta gli archi in modo da creare un grafo $G'$ diretto e aciclico. Questo algoritmo deve avere tempo di esecuzione $\Theta(n + m)$.

**Soluzione 6.** L'idea per risolvere questo esercizio è quella di, dato un grafo $G = (V, E)$, orientare gli archi di $G$ in maniera che (1) c'è un vertice $z$ con grado entrante nullo e (2) un arco $(u, v)$ di $G'$ è orientato da $u$ a $v$ in $G'$ se e solo se in $G'$ c'è un cammino diretto da $u$ a $v$. Per fare questo, definiamo per ogni vertice $u \in V$ una lista di adiacenza $Arc[u]$ (inizialmente vuota, vedi linea 4) che al terminare dell'algoritmo ha la funzione di essere la lista di adiacenza del vertice $u$ nel grafo $G'$. Nel corso dell'algoritmo, che prende spunto da una classica ricerca in profondità, una volta considerato un arco non orientato $(u, v)$ di $G$ allora in $G'$ è presente esattamente uno tra i seguenti due archi orientati $(u, v)$ o $(v, u)$, dipendentemente dallo stato corrente della ricerca in profondità.

Ricordiamo che in seguito ad una ricerca in profondità di un grafo non diretto, gli archi possono solo essere dell'albero o all'indietro. Consideriamo un arco dell'albero $(u, v)$, cioè un arco in cui la condizione in linea 16 è soddisfatta, allora indirizziamo l'arco da $u$ a $v$ creando un cammino diretto da $u$ a $v$ (formato banalmente da questo arco diretto $(u, v)$) in $G'$ (linea 17). Finalmente consideriamo un arco all'indietro $(u, v)$, cioè un arco in cui la condizione in linea 16 non è soddisfatta, allora indirizziamo l'arco da $v$ a $u$ (linea 20): infatti per definizione già esiste in $G'$ un cammino diretto da $v$ ad $u$ (che è quello univocamente determinato dagli archi dell'albero).

**Algorithm 3** Algoritmo per ottenere un DAG da un grafo.

**Input:** un grafo non diretto $G = (V, E)$.
**Output:** un grafo diretto e aciclico $G'$.

```
global variables
    Color ↤ array di |V| elementi
    for u ∈ G do
        Arc[u] ↤ liste di adiacenza del grafo diretto G', inizialmente vuote
end global variables
function Direct(G)
    for u ∈ V do
        Color[u]=BIANCO
    for u ∈ V do
        if Color[u]==BIANCO then
            DFS-Visit(G,u)
    return
function DFS-Visit(G,u)
    Color[u]=GRAY
    for v ∈ Adj[u] do
        if Color[v]==BIANCO then
            Arc[u].append(v)
            DFS-Visit(G,v)
        else
            Arc[v].append(u)
    Color[u]=NERO
    return
```

# References


[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms. 2022.

[2] Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.