

Esercitazione 3: Applications of Graph Search Algorithms

Giacomo Paesani

March 27, 2025

Esercizio 1. 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 1. Prima di dare la soluzione dell'esercizio, è necessario ricordare il seguente fatto: in un grafo diretto e aciclico esiste un vertice del grafo con grado uscente nullo. L'idea della soluzione proposta è quella di calcolare e aggiornare dinamicamente i gradi uscenti dei vertici del grafo: ad ogni passo viene selezionato un vertice di grado uscente 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 1 è ispirata 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. Il vettore *Out* 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 prefigge di accumulare tutti vertici che al momento hanno grado uscente nullo (inizializzato in linea 3). Infine la coda Q (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 *Out* affinché il valore di ogni coordinata sia uguale al grado uscente 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'obiettivo di scorrere la lista dei vertici del grafo e pone quelli di grado uscente 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 coda Q attraverso l'operazione *enqueue*(u) 13. Quello che ci resta da fare è aggiornare il valore del grado uscente dei restanti vertici di G ; in particolare sono modificati (e ridotti di uno) solo i gradi uscenti dei vertici adiacenti ad u . Se il grado uscente di un vertice v diviene nullo, allora lo si aggiunge alla pila S .

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 uscenti, allora viene aggiunto a S in linea 9. Siano ora v_1, \dots, v_k , l'insieme dei vertici di G tali che (u, v_i) è un arco di G , per $i = 1, \dots, k$. Analizziamo i seguenti casi: (1) se per ogni $i = 1, \dots, k$ esiste un passo dell'algoritmo tale che v_i è un elemento di S , allora $Out[u]$ diventerà 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 (u, v) 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 iterazione u viene rimosso da S con l'operazione *pop*() e aggiunto ad Q con l'operazione *enqueue*. E' anche facile mostrare che un vertice u che è stato aggiunto a Q non può essere inserito nuovamente in S e quindi neanche in Q un'ulteriore volta.

Algorithm 1 Algoritmo per calcolare un ordine topologico di un DAG.

Input: grafo diretto e aciclico $G = (V, E)$.

Output: un ordine topologico Q .

```
1: global variables
2:    $Out \leftarrow$  vettore dei gradi uscenti dei nodi di  $|V|$  elementi
3:    $S \leftarrow$  pila, inizialmente vuota
4:    $Q \leftarrow$  coda, 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:      $u \leftarrow S.pop()$ 
13:      $Q.enqueue(u)$ 
14:     for  $v \in Adj[u]$  do
15:        $Out[v] = Out[v] - 1$ 
16:       if  $Out[v] == 0$  then
17:          $S.push(v)$ 
18:   return  $Q$ 
19: function ComputeDegr( $G$ )
20:   for  $u \in V$  do
21:      $Out[u] = 0$ 
22:     for  $v \in Adj[u]$  do
23:        $Out[u] = Out[u] + 1$ 
24:   return
```

Esercizio 2 (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 2. 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 uscente 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).

Esercizio 3 (22.2-9, [1]). Fornire un algoritmo in pseudo-codice che, dato un grafo non diretto e connesso $G = (V, E)$, trova una passeggiata in G che attraversa tutti gli archi una e una sola volta in ognuna delle due direzioni in tempo $\mathcal{O}(|V| + |E|)$.

Soluzione 3. L'idea è di effettuare una visita qualsiasi del grafo G (che sia una DFS o una BFS) e nel mentre costruire una passeggiata P che segue gli archi di visita. L'Algoritmo 3 è una delle possibili soluzioni: il codice è ispirato all'algoritmo ricorsivo di ricerca in profondità proposto in [1]. La struttura della coda P (inizializzata in Linea 2) ci permette di rappresentare una sequenza dei vertici che forma la soluzione richiesta.

Controlliamo che l'algoritmo proposto ritorni effettivamente una passeggiata di G che attraversa tutti gli archi una e una sola volta in ognuna delle

Algorithm 2 Algoritmo per ottenere un DAG da un grafo.

Input: un grafo non diretto $G = (V, E)$.

Output: un grafo diretto e aciclico G' .

```
1: global variables
2:    $Color \leftarrow$  vettore di  $|V|$  elementi
3:   for  $u \in G$  do
4:      $Arc[u] \leftarrow$  liste di adiacenza del grafo diretto  $G'$ , inizialmente vuote
5: end global variables
6: function Direct( $G$ )
7:   for  $u \in V$  do
8:      $Color[u] = \text{BIANCO}$ 
9:   for  $u \in V$  do
10:    if  $Color[u] == \text{BIANCO}$  then
11:      DFS-VISIT( $G, u$ )
12:   return
13: function DFS-Visit( $G, u$ )
14:    $Color[u] = \text{GRAY}$ 
15:   for  $v \in Adj[u]$  do
16:    if  $Color[v] == \text{BIANCO}$  then
17:       $Arc[u].append(v)$ 
18:      DFS-VISIT( $G, v$ )
19:    else
20:       $Arc[v].append(u)$ 
21:    $Color[u] = \text{NERO}$ 
22:   return
```

due direzioni. Sia uv un arco di G e supponiamo che l'algoritmo sta visitando l'arco uv per la prima volta analizzando la chiamata $\text{DFS-VISIT}(G, u)$. Se l'arco uv risulta un arco dell'albero, allora l'arco da u a v viene percorso subito da P grazie alla Linea 20 e viene in fine percorso da v ad u al termine della chiamata $\text{DFS-VISIT}(G, u)$ grazie alla Linea 25. Se l'arco uv risulta un arco all'indietro allora P percorre prima l'arco da u a v e subito dopo da v ad u grazie alle Linee 23 and 24.

In sintesi, partendo dal vertice che viene scelto come radice della ricerca (aggiunto in testa a P in Linea 11), l'algoritmo riporta i vertici del grafo nell'ordine in questi si presentano nel corso della ricerca. \square

Esercizio 4 (22.4-2, [1]). Fornire un algoritmo in pseudo-codice che dato un grafo diretto e aciclico $G = (V, E)$ e due vertici s e t , restituisce il numero di tutti i cammini da s a t in G .

Soluzione 4. La soluzione proposta descritto nell'Algoritmo 4 è un primo esempio di programmazione dinamica: una diffusa tecnica per risolvere numerosi problemi di informatica teorica. L'idea è quella di trattare le varie chiamate della funzione CONTACAMMINI come una pila, ogni nuova chiamata viene messa in cima e può completata solo quando le chiamate successive sono state già risolte.

Il vettore C viene inizializzato con ogni coordinata uguale a -1 (Linea 6) e una coordinata $C[v]$ rimane uguale a -1 finché la chiamata alla funzione $\text{CONTACAMMINI}(v, t)$ non è stata completata. Per vedere che l'incapsulamento di queste chiamate si risolve, guardiamo prima ad un semplice caso. Osserviamo il comportamento della funzione CONTACAMMINI con input (t, t) : allora per l'assegnazione fatta in Linea 7, possiamo correttamente concludere che c'è un solo cammino da t a t , cioè quello di lunghezza zero. Una volta ottenuta questa informazione, è possibile iniziare a risolvere le chiamate precedenti CONTACAMMINI fino a poter risolvere la chiamata con input (s, t) .

Concludiamo considerando il caso in cui non esiste alcun cammino da s a t in G , allora seguendo ricorsivamente i vertici adiacenti ad s non troviamo mai t ma dei pozzi di G . Questi pozzi non sono contenuti in alcun cammino da s a t e quindi il loro contributo nelle chiamate CONTACAMMINI è nullo. \square

Esercizio 5. Sia un grafo diretto $G = (V, E)$ ed s e t due vertici di G . G si dice s - t -connesso se ogni vertice di G è in almeno un cammino da s a t . Fornire un algoritmo in pseudo-codice che dato un grafo diretto e aciclico $G = (V, E)$ ed s e t due vertici di G , restituisce un sottografo G' di G massimale s - t -connesso.

Algorithm 3 DFS modificata per ricavare una passeggiata che percorre ogni arco di un grafo esattamente una volta in entrambe le direzioni.

Input: grafo non diretto e connesso $G = (V, E)$.

Output: passeggiata P che percorre ogni arco esattamente una volta in ogni direzione.

```
1: global variables
2:    $P \leftarrow$  coda, inizialmente vuota
3:    $Color \leftarrow$  vettore di  $|V|$  elementi
4:    $Parent \leftarrow$  vettore dei padri di  $|V|$  elementi
5: end global variables
6: function DFS-PATH( $G$ )
7:   for  $u \in V$  do
8:      $Color[u] = \text{BIANCO}$ 
9:   for  $u \in V$  do
10:    if  $Color[u] == \text{BIANCO}$  then
11:       $P.enqueue(u)$ 
12:      DFS-VISIT( $G, u$ )
13:       $Parent[u] = u$ 
14:   return  $P$ 
15: function DFS-Visit( $G, u$ )
16:    $Color[u] = \text{GRIGIO}$ 
17:   for  $v \in Adj[u]$  do
18:     if  $Color[v] == \text{BIANCO}$  then
19:        $Parent[v] = u$ 
20:        $P.enqueue(v)$ 
21:       DFS-VISIT( $G, v$ )
22:     if  $Color[v] == \text{GRIGIO}$  and  $v \neq Parent[u]$  then
23:        $P.enqueue(v)$ 
24:        $P.enqueue(u)$ 
25:    $P.enqueue(Parent[u])$ 
26:    $Color[u] = \text{NERO}$ 
27:   return
```

Algorithm 4 Algoritmo che conta il numero di tutti i cammini tra due vertici in un grafo diretto e aciclico

Input: grafo diretto e aciclico $G = (V, E)$ e due vertici s e t

Output: numero di cammini distinti da s a t

```
1: global variables
2:    $C \leftarrow$  vettore di  $|V|$  elementi
3: end global variables
4: function NumeroCammini( $s, t$ )
5:   for  $u \in V$  do
6:      $C[u] = -1$ 
7:    $C[t] = 1$ 
8:   return CONTACAMMINI( $s, t$ )
9: function ContaCammini( $u, t$ )
10:  if  $C[u] \neq -1$  then
11:    return  $C[u]$ 
12:   $k = 0$ 
13:  for  $v \in Adj[u]$  do
14:     $k = k + \text{CONTACAMMINI}(v, t)$ 
15:   $C[u] = k$ 
16:  return  $k$ 
```

Soluzione 5. La soluzione di questo esercizio può essere facilmente ottenuta modificando la soluzione proposta del Esercizio 4. L'Algoritmo 5 adotta nuovamente il paradigma della programmazione dinamica: le chiamate alla funzione VISITACAMMINI sono incapsulate e le chiamate *esterne* non possono essere risolte finché non si ottiene per quelle più *interne*.

Se per un vertice u la soluzione a VISITACAMMINI con input (u, t) è stata già calcolata, allora essa non viene calcolata nuovamente (Linea 14). Chiaramente, la chiamata a VISITACAMMINI con input (t, t) impone $Visited[t] = 1$ (Linea 16), cioè t è incluso in un cammino da t a t . Supponiamo ora che la risposta VISITACAMMINI con input (u, t) non è stata già calcolata e che $u \neq t$. Allora per avere un cammino da u a t ci deve essere almeno un vicino v di u tale che esiste un cammino da v a t . Questa verifica viene fatta nel ciclo **for** di Linea 19: se tale cammino esiste allora $Visited[u]$ viene posto uguale ad 1, e viene posto uguale a 0 altrimenti.

La chiamata iniziale è fatta al Algoritmo MASSIMALE. Dopo aver inizializzato il vettore $Visited$ viene fatta la prima chiamata VISITACAMMINI con input (s, t) . Se in seguito a questa chiamata $Visited[s]$ è uguale ad 1, allora vuol dire che è stato trovato almeno un cammino e quindi la soluzione è valida: i vertici u di G tali $Visited[u] = 1$ sono tutti e soli i vertici del sottografo di G' . Più precisamente, se denotiamo con $V' = \{u \in V \mid Visited[u] = 1\}$ allora $G' = G[V']$.

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.

Algorithm 5 Algoritmo che seleziona il sottografo massimale che contiene tutti i vertici per cui passa almeno un cammino tra due determinati vertici.

Input: grafo diretto e aciclico $G = (V, E)$ e due vertici s e t

Output: tutti i vertici che sono coinvolti in cammini da s a t

```
1: global variables
2:    $Visited \leftarrow$  vettore di  $|V|$  elementi
3: end global variables
4: function Massimale( $s, t$ )
5:   for  $u \in V$  do
6:      $Visited[u] = -1$ 
7:   VISITACAMMINI( $s, t$ )
8:   if  $Visited[s] \neq 1$  then
9:     for  $u \in V$  do
10:       $Visited[u] = 0$ 
11:   return
12: function VisitaCammini( $u, t$ )
13:   if  $Visited[u] \neq -1$  then
14:     return
15:   if  $u == t$  then
16:      $Visited[t] = 1$ 
17:     return
18:    $k = 0$ 
19:   for  $v \in Adj[u]$  do
20:     VISITACAMMINI( $v, t$ )
21:     if  $Visited[v] > k$  then
22:        $k = Visited[v]$ 
23:    $Visited[u] = k$ 
24:   return
```
