Esercitazione 3: Applications of Graph Search Algorithms

Giacomo Paesani

March 25, 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: 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 1 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.

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.

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

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

```
global variables
    Ind ↤ array dei gradi entranti dei nodi
    S ↤ pila, inizialmente vuota
    L ↤ lista, inizialmente vuota
end global variables
function OrdTop(G)
    ComputeDegr(G)
    for u ∈ V do
        if Ind[u] == 0 then
            S.push(u)
    while S not empty do
        u ↤ S.pop()
        L.append(u)
        for v ∈ Adj[u] do
            Ind[v] = Ind[v] − 1
            if Ind[v] == 0 then
                S.push(v)
    return
function ComputeDegr(G)
    for u ∈ V do
        Ind[u] = 0
        for v ∈ Adj[u] do
            Ind[u] = Ind[u] + 1
    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 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).

Esercizio 3 (22.2-9, [1])**.** Fornire una 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 della 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 possibile soluzioni: il codice è inspirato 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 della

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'.

```
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
```

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 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 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.

L'array 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 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.

```
global variables
    P ↤ coda, inizialmente vuota
    Color ↤ array di |V| elementi
    Parent ↤ array dei padri
end global variables
function DFS-PATH(G)
    for u ∈ V do
        Color[u]=BIANCO
    for u ∈ V do
        if Color[u]==BIANCO then
            P.enqueue(u)
            DFS-Visit(G,u)
            Parent[u] = u
    return A
function DFS-Visit(G,u)
    Color[u]=GRIGIO
    for v ∈ Adj[u] do
        if Color[v]==BIANCO then
            Parent[v] = u
            P.enqueue(v)
            DFS-Visit(G,v)
        if Color[v]==GRIGIO and v! = Parent[u] then
            P.enqueue(v)
            P.enqueue(u)
    P.enqueue(Parent[u])
    Color[u]=NERO
    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

```
global variables
    C ↤ array di |V| elementi
end global variables
function NumeroCammini(s, t)
    for u ∈ V do
        C[u] = −1
    C[t] = 1
    return ContaCammini(s,t)
function ContaCammini(u,t)
    if C[u] ≠ −1 then
        return C[u]
    k = 0
    for v ∈ Adj[u] do
        k = k+ContaCammini(v,t)
    C[u] = k
    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 l'array $Visited$ 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

```
global variables
    Visited ↤ array di |V| elementi
end global variables
function Massimale(s, t)
    for u ∈ V do
        Visited[u] = −1
    VisitaCammini(s,t)
    if Visited[s]! = 1 then
        for u ∈ V do
            Visited[u] = 0
    return
function VisitaCammini(u,t)
    if Visited[u]! = −1 then
        return
    if u == t then
        Visited[t] = 1
        return
    k = 0
    for v ∈ Adj[u] do
        VisitaCammini(v,t)
        if Visited[v] > k then
            k = Visited[v]
    Visited[u] = k
    return
```