Esercitazione 3: Strongly Connected Components

Giacomo Paesani

March 26, 2024

Esercizio 1 (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 1. 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 1 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 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. □

Algorithm 1 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
```

Esercizio 2 (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 2. La soluzione proposta descritto nell'Algoritmo 2 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).

Algorithm 2 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:     $C \leftarrowtail$ array 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:         $k = k+$ContaCammini(v,t)
15:     $C[u] = k$
16:     **return** k

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 3. 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 $G = (V, E)$ ed s e t due vertici di G, restituisce un sottografo G' di G massimale s-t-connesso.

Soluzione 3. La soluzione di questo esercizio può essere facilmente ottenuta modificando la soluzione proposta del Esercizio 2. L'Algoritmo 3 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 $G' = G[V']$.

Algorithm 3 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
```

Esercizio 4 (22.5-1, [1])**.** Come cambia il numero di componenti fortemente connesse di un grafo diretto se viene aggiunto un nuovo arco?

Soluzione 4. Sia $G = (V, E)$ un grafo diretto e siano u e v due vertici distinti di G. Consideriamo il grafo ottenuto da G aggiungendo l'arco (u, v). Distinguiamo i seguenti casi:

- se u e v appartengono alla stessa componente fortemente connessa di G, allora l'arco (u, v) non modifica la composizione delle componenti fortemente connesse;

- se u e v appartengono a due distinte componenti fortemente connesse di G ed esiste un cammino P da v ad u, allora P concatenato con l'arco (u, v) crea un ciclo: tutte le componenti fortemente connesse che intersecano il cammino P formano un'unica componente fortemente connessa del grafo risultate;

- se u e v appartengono a due distinte componenti fortemente connesse di G e non esiste un cammino da v ad u, allora anche in questo caso l'arco (u, v) non modifica la composizione delle componenti fortemente connesse.

Esercizio 5 (I. Salvo)**.** Sia $G = (V, E)$ un grafo diretto, un vertice $u \in V$ è detto *principale* se per ogni vertice $v \in V$ esiste un cammino diretto da u a v. Fornire un algoritmo in pseudo-codice che, dato un grafo diretto G, determina tutti i vertici principali di G. E' possibile che tale algoritmo abbia complessità $\mathcal{O}(|V| + |E|)$?

Soluzione 5. Per risolvere questo algoritmo, dobbiamo prima ricordare cosa è una componente fortemente connessa: sia $G = (V, E)$ un grafo diretto, allora una componente fortemente connessa di G è un sottoinsieme di vertici $U \subseteq V$ massimale tale che per ogni due vertici $u, v \in U$ allora esiste sempre un cammino da u a v e un cammino da v ad u.

Supponiamo inizialmente che G sia aciclico e che quindi ammette ordinamenti topologici per i suoi vertici. Sia $W \subseteq V$ l'insieme di tutti i vertici di G con grado entrante pari a 0: questi sono tutti i possibili candidati ad essere vertici principali. Ricordiamo che se G è aciclico, allora $|W| \geq 1$. Se l'insieme W contiene almeno due vertici w_1 e w_2 allora G non ammette vertici principali: infatti non esiste alcun cammino da w_1 a w_2 perché w_2 non ha alcun arco entrante e non esiste alcun cammino da w_2 a w_1 perché w_1 non ha alcun arco entrante. Finalmente consideriamo il caso $W = \{w\}$ e allora possiamo concludere che w è l'unico vertice principale di G.

Algorithm 4 Algoritmo per calcolare tutti i vertici principali di un grafo diretto.

Input: grafo diretto $G = (V, E)$.
Output: insieme A che contiene tutti i vertici principali di G.

```
1: function PRINCIPAL(G)
2:     G^SCC =SCC(G)
3:     InDeg ↤ array dei gradi entranti di G^SCC, con ogni coordinata inizializ-
       zata a 0
4:     A ↤ insieme, inizialmente vuoto
5:     a = 0
6:     for u ∈ V(G^SCC) do
7:         for v ∈ Adj[u] do
8:             InDeg[v] = InDeg[v] + 1
9:     for u ∈ V(G^SCC) do
10:        if InDeg[u] == 0 and a! = 0 then
11:            return A
12:        if InDeg[u] == 0 and a == 0 then
13:            a = 1
14:            p = u
15:    for u ∈ p do
16:        A.add(u)
17:    return A
```

Consideriamo ora il caso più generale, cioè quello in cui G contiene cicli. Facciamo ora la seguente affermazione: siano v_1 e v_2 due vertici appartenenti alla stessa componente fortemente connessa di G, allora v_1 è un vertice principale di G se e solo se v_2 lo è. Infatti se da v_1 esiste cammino diretto per ogni altro vertice di G, allora tale cammino esiste anche da v_2 (basta usare il cammino da v_2 a v_1 come prefisso) e viceversa.

Grazie alla precedente affermazione possiamo adottare l'idea implementata nel Algoritmo 4: prima calcoliamo il grafo G^{SCC} delle componenti fortemente connesse di G con la chiamata alla funzione SSC(G) (Linea 2), esso è per sua natura aciclico, e su G^{SCC} possiamo adottare la strategia descritta per i grafi aciclici. Il ciclo **for** in Linea 6 serve a calcolare il grado entrante di ogni vertice di G^{SCC} e salvarlo nell'array $InDeg$. La variabile a controllare il numero di vertici di G^{SCC} con grado entrante nullo, se ce n'è più di uno e quindi a viene aggiornata più di una volta nel corso dell'algoritmo allora non vi sono vertici principali in G^{SCC} e quindi neanche in G (Linea 11). Supponiamo invece che a viene aggiornata una volta sola e quindi esiste un solo vertice u di G^{SCC} con grado entrante nullo. Allora u viene salvato nella variabile p (Linea 14) che rappresenta la componente fortemente connessa principale di G e quindi in A vengono aggiunti tutti i vertici di G in p (Linea 16), che quindi sono tutti e soli i vertici principali di G. □

Esercizio 6 (22.2-8, [1])**.** Sia $G = (V, E)$ un grafo non diretto, allora si definisce il *diametro* di G, $diam(G) = \max_{u,v \in V} d(u, v)$, il massimo della distanza tra due qualsiasi vertici di G. Fornire un algoritmo in pseudo-codice che restituisca il diametro di un grafo G, nel caso in cui G sia un albero. E' possibile ottenere una soluzione con tempo di esecuzione $\mathcal{O}(|V|)$?

Soluzione 6. Per dare una spiegazione convincente della soluzione proposta abbiamo bisogno della seguente affermazione: sia $G = (V, E)$ un albero e u un vertice di G. Se u è un vertice a distanza massima in una visita in profondità radicata in un nodo arbitrario r, allora esiste un altro vertice $v \in V$ tale che $d(u, v) = diam(G)$.

L'Algoritmo 5 è una delle possibile soluzioni: il codice è inspirato all'algoritmo ricorsivo di ricerca in profondità proposto in [1] e l'idea è la seguente. Prima svolgiamo una qualsiasi visita dell'albero (in questo caso usiamo una visita in profondità) scegliendo un qualsiasi vertice r come radice e per ogni vertice viene registrata la distanza da r. In seguito facciamo una seconda visita dell'albero radicata in uno dei vertici a distanza massima da r. Dopo questa ulteriore visita otteniamo il valore corretto del diametro dell'albero in input.

Algorithm 5 Algoritmo per calcolare il diametro di un albero non diretto.

Input: grafo non diretto $G = (V, E)$.
Output: diametro di G.

```
global variables
    Color ↤ array di |V| elementi
    dist ↤ array di |V| elementi
    max ↤ intero che traccia la distanza massima dalla radice
    w ↤ variabile vertice di appoggio
end global variables
function DIAMETRO(G)
    for u ∈ V do
        Color[u]=BIANCO
        dist[u] = 0
    c = 0
    for u ∈ V do
        if Color[u]==BIANCO then
            w = u
            max = 0
            DFS-Dist(G,u)
            c = c + 1
            if c > 1 then
                return +∞
    for u ∈ V do
        dist[u] = 0
    max = 0
    DFS-Dist(G,w)
    return max
function DFS-Dist(G,u)
    Color[u]=GRIGIO
    for v ∈ Adj[u] do
        if Color[v]==BIANCO then
            Color[v]=GRIGIO
            dist[v] = dist[u] + 1
            if dist[v] > max then
                dist[v] = max
                w = v
            DFS-Dist(G,v)
    return
```

Durante la visita in profondità dell'albero G, nel vettore $dist$, inizializzato a 0 (Linee 10 e 21), viene registrata la distanza dalla radice al vertice in analisi (Linea 30): la misura di distanza è corretta perché G è un albero. Il contatore c indica il numero di componente connesse di G: se ce n'è più di una allora il diametro di G è $+\infty$ e viene correttamente riportato 19.

Supponiamo ora che G ha esattamente una componente connessa, la prima chiamata alla funzione **DFS-Dist**(G,u) (Linea 16) serve a trovare il vertice (la foglia) di G a distanza massima dalla radice u. Terminata la chiamata **DFS-Dist**(G,u), la variabile max contiene la distanza massima di un vertice dalla radice e w è un testimone di tale distanza massima, cioè w è un vertice tale che $dist(u, w) = max$. Allora eseguiamo una nuova ricerca in profondità di G con vertice iniziale w (Linea 23). Al termine di della chiamata **DFS-Dist**(G,w), nella variabile max viene registrato il diametro di G e riportato in Linea 24 .

Concludiamo la soluzione a questo esercizio analizzando la complessità della soluzione proposta nel Algoritmo 5. La complessità è dominata dalle due ricerche in profondità in serie e quindi $\mathcal{O}(|V| + |E|)$. Ricordando che se l'input è un albero, avremo che $|E| = |V| - 1$ e quindi la complessità de Algoritmo 5 è $\mathcal{O}(|V|)$. $\square$

References

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