

Esercizi Capitolo 11 - Strutture di dati e progettazione di algoritmi

Alberto Montresor

19 Agosto, 2014

Alcuni degli esercizi che seguono sono associati alle rispettive soluzioni. Se il vostro lettore PDF lo consente, è possibile saltare alle rispettive soluzioni tramite collegamenti ipertestuali. Altrimenti, fate riferimento ai titoli degli esercizi. Ovviamente, si consiglia di provare a risolvere gli esercizi personalmente, prima di guardare la soluzione.

Per molti di questi esercizi l'ispirazione è stata presa dal web. In alcuni casi non è possibile risalire alla fonte originale. Gli autori originali possono richiedere la rimozione di un esercizio o l'aggiunta di una nota di riconoscimento scrivendo ad `alberto.montresor@unitn.it`.

1 Problemi

1.1 Algoritmo di Dijkstra (Esercizio 11.1 del libro)

Cosa succede se l'algoritmo di Dijkstra è eseguito su un grafo in cui le lunghezze degli archi possono essere negative?

Soluzione: Sezione 2.1

1.2 Algoritmo di Bellman-Ford-Moore (Esercizio 11.2 del libro)

Si utilizzi l'algoritmo di Bellman-Ford-Moore per verificare in tempo $O(mn)$ se un grafo possiede cicli negativi.

Soluzione: Sezione 2.2

1.3 Numero di cammini minimi (Esercizio 11.7 del libro)

Dati un grafo orientato con lunghezze positive sugli archi e due nodi s ed t , possono esserci molti cammini minimi distinti fra essi. Si modifichi l'algoritmo di Dijkstra per calcolare il numero dei cammini minimi distinti da s a t .

Soluzione: Sezione 2.3

1.4 Cammino con arco più corto (Esercizio 11.8 del libro)

Dati un grafo orientato con lunghezze positive sugli archi e due nodi r ed s , si vuole individuare il cammino da r ad s che contiene l'arco più corto, cioè quello con lunghezza minima tra tutti gli archi che possono apparire nel cammino. Si fornisca un algoritmo per determinare il cammino da r ad s contenente l'arco più corto e se ne analizzi la complessità.

Soluzione: Sezione 2.4

1.5 Probabilità di fallimento

Dato un grafo orientato $G = (V, E)$ nel quale ogni arco $(u, v) \in E$ è associato ad un valore reale $r(u, v)$ preso dall'intervallo $[0, 1]$, che rappresenta l'affidabilità del canale di comunicazione dal vertice u al vertice v . Interpretiamo $r(u, v)$ come la probabilità che il canale trasmetta un messaggio e supponiamo che queste probabilità siano indipendenti.

Create un algoritmo efficiente per trovare il cammino più affidabile fra due vertici dati.

Soluzione: Sezione [2.5](#)

2 Soluzioni

2.1 Algoritmo di Dijkstra (Esercizio 11.1 del libro)

Si veda Figura 1(a) per un caso in cui l'algoritmo di Dijkstra fallisce; come è possibile vedere, una volta scelto l'arco (a, c) per raggiungere il nodo c , l'algoritmo di Dijkstra non torna più indietro nonostante sia stato trovato un percorso migliore. Si veda Figura 1(b) per un caso in cui l'algoritmo di Dijkstra ha successo; come è possibile vedere, tutti gli archi negativi escono dalla sorgente.

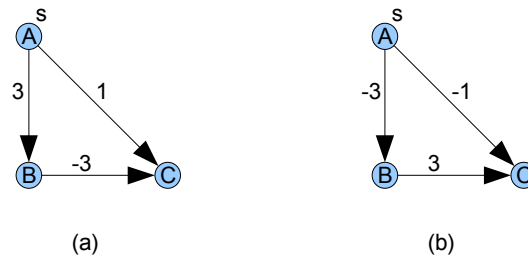


Figura 1: (a) Un grafo con archi negativi in cui l'algoritmo di Dijkstra fallisce. (b) Un grafo con archi negativi in cui l'algoritmo di Dijkstra ha successo.

2.2 Algoritmo di Bellman-Ford-Moore (Esercizio 11.2 del libro)

L'algoritmo di Bellman-Ford-Moore, in presenza di cicli negativi, non termina mai: si consideri ad esempio la Figura 11.3 a pagina 213 del libro: se il peso dell'arco $(4, 3)$ viene sostituito con il valore 2 (al posto del valore 4), il ciclo 2-4-3 ha valore totale -1 ; al passo g), quando viene estratto il nodo 4, è possibile raggiungere il nodo 3 con 2; questo viene re-inserito nella coda e si ricomincia.

Un modo per sfruttare questo fatto consiste nel registrare, per ogni nodo v , la distanza da r a v (misurata in numero di archi) (oltre ovviamente al peso totale del cammino da r a v). Se per un qualunque nodo questo

valore raggiunge n , il grafo contiene un ciclo negativo.

negativeCycle(GRAPH G , NODE r)

```

int[]  $d \leftarrow$  new int[1... $G.n$ ]           %  $d[u]$  è la distanza da  $r$  a  $u$ 
boolean[]  $b \leftarrow$  new boolean[1... $G.n$ ] %  $b[u]$  è true se  $u$  è contenuto in  $S$ 
boolean[]  $L \leftarrow$  new boolean[1... $G.n$ ] %  $L[u]$  è la distanza da  $r$  a  $u$  (in numero di archi)
foreach  $u \in G.V() - \{r\}$  do  $d[u] \leftarrow +\infty$ 

foreach  $u \in G.V() - \{r\}$  do  $b[u] \leftarrow$  false

 $L[r] \leftarrow 0$ ;  $d[r] \leftarrow 0$ ;  $b[r] \leftarrow$  true
QUEUE  $S \leftarrow$  Queue();  $S.enqueue(r)$ 
while not  $S.isEmpty()$  do
     $u \leftarrow S.dequeue()$ 
     $b[u] \leftarrow$  false
    foreach  $v \in G.adj(u)$  do
        if  $d[u] + w(u, v) < d[v]$  then
            if not  $b[v]$  then
                 $S.enqueue(v)$ 
                 $b[v] \leftarrow$  true
             $d[v] \leftarrow d[u] + w(u, v)$ 
             $L[v] \leftarrow L(u) + 1$ 
            if  $L[v] \geq n$  then return true

return false

```

2.3 Numero di cammini minimi (Esercizio 11.7 del libro)

L'algoritmo seguente è basato su programmazione dinamica / memoization. Si crea un vettore che mantiene il numero di cammini con cui ogni nodo può raggiungere t . Inizialmente, tutti i nodi sono inizializzati a -1 , ad indicare che non sono ancora stati calcolati, tranne il nodo t , per cui ovviamente esiste un solo cammino per raggiungere se stesso (il cammino vuoto). Quando viene calcolato il numero di cammini per un nodo u , si effettua la sommatoria di tutti i valori corrispondenti per ognuno dei nodi adiacenti ad u .

int pathcount(GRAPH G , NODE s , NODE t)

```

foreach  $v \in V$  do
     $a[v] \leftarrow -1$ 
 $a[t] \leftarrow 1$ 
return r-pathcount( $G, s$ )

```

int r-pathcount(GRAPH G , NODE u)

```

if  $a[u] < 0$  then
     $a[u] \leftarrow 0$ 
    foreach  $v \in G.adj(u)$  do
         $a[u] \leftarrow a[u] + \text{r-pathcount}(G, v)$ 
return  $a[s]$ 

```

Complessità Ogni nodo viene visitato al più una volta, e tutti gli archi di ogni nodo visitato vengono percorsi. Quindi la complessità è $O(n + m)$.

Sottostruttura ottima Dobbiamo dimostrare che il problema in questione gode della proprietà di sottostruttura ottima. Informalmente, l'idea è la seguente: tutti i vertici adiacenti ad un nodo u hanno un certo numero di cammini per raggiungere t . Se aggiungo un arco a questi percorsi (quello per arrivare ai rispettivi vertici di partenza), i percorsi che ottengo sono tutti distinti. Poiché l'albero è un DAG, non c'è modo di ritornare al nodo di partenza (se ci fossero cicli, il numero di percorsi diventerebbe infinito).

2.4 Cammino con arco più corto (Esercizio 11.8 del libro)

È sufficiente modificare l'algoritmo di Dijkstra nel modo seguente:

La riga: **if** $d[u] + w(u, v) < d[v]$ **then**

diventa **if** $\min(w(u, v), d[u]) < d[v]$ **then**

La riga $d[v] \leftarrow d[u] + w(u, v)$

diventa $d[v] \leftarrow \min(w(u, v), d[u])$

2.5 Probabilità di fallimento

Un metodo molto semplice si basa sulle proprietà dei logaritmi: il prodotto delle affidabilità degli archi di un cammino è uguale all'affidabilità del cammino. Il logaritmo del prodotto di due valori è uguale alla somma del logaritmo dei due valori. L'affidabilità è massima quando il logaritmo è massimo; quindi possiamo definire i nostri pesi come $w(x, y) = -\log(a(x, y))$.