Corso di Algoritmi e Strutture Dati—Modulo 3

Esercizi di ripasso su grafi e programmazione dinamica – 7/4/2020

Moreno Marzolla

Esercizio 1. Disponiamo di n monete aventi tagli rispettivamente $c[1..n]$; si noti che a differenza del problema del resto, $c[i]$ rappresenta il taglio della moneta i-esima, non infinite monete di valore $c[i]$. L'array $c[i]$ può contenere duplicati (possono esistere più monete dello stesso taglio), e si può decidere a scelta che sia ordinato oppure no. Vogliamo determinare il numero massimo di monete, scelte tra le n disponibili, che sono necessarie per erogare un importo R, se questo è possibile.
Ad esempio, se $c = [1, 1, 1, 2, 2, 2, 2, 5]$ e $R = 6$, l'algoritmo deve restituire il valore 4, poiché 4 è il massimo numero di monete che devono essere usate per erogare il resto 6 (1, 1, 2, 2). Nota: non si può assumere che i valori delle monete siano quelli a noi familiari; potrebbero quindi essere presenti monete di taglio arbitrario.

Soluzione. È facile rendersi conto che non è possibile usare un algoritmo greedy "al contrario". Nell'esempio precedente, consumando prima le monete da 1 non sarebbe stato possibile erogare il resto con le rimanenti. Usiamo invece un approccio basato sulla programmazione dinamica. Sia $N[i, r]$ il massimo numero di monete, scelte tra quelle di indici $\{1, \ldots, i\}$, che sono necessarie per erogare un resto r ($i = 1, \ldots, n$, $r = 0, \ldots, R$). Cominciamo dal caso base ($i = 1$): con solo la prima moneta possiamo erogare solo un resto zero (usando zero monete) oppure $c[1]$ (usando una moneta, la prima).

$$N[1,r]=\begin{cases} 0 & \text{se } r=0 \\ 1 & \text{se } r=c[1] \\ -\infty & \text{altrimenti} \end{cases}$$

Nel caso generale, $N[i, r]$ si può calcolare ragionando come segue:

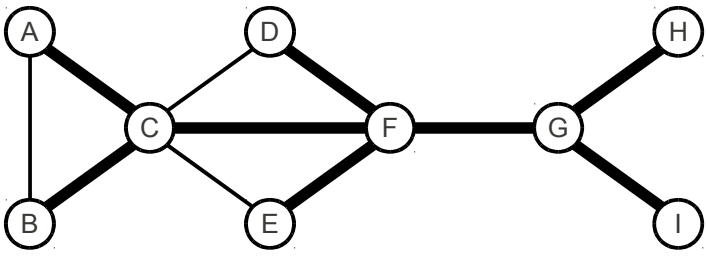
1. Se decidiamo di usare la moneta i-esima, il massimo numero di monete necessarie per erogare il resto r è $(1 + N[i - 1, r - c[i]])$; questo potrebbe risultare $-\infty$ se il resto $(r - c[i])$ non è erogabile con le prime $(i - 1)$ monete. Si noti inoltre che ciò è possibile solo se il resto da erogare è maggiore o uguale al taglio della moneta i-esima.
2. Se decidiamo di non usare la moneta i-esima, il massimo numero di monete necessarie per erogare il resto r è $N[i - 1, r]$; anche qui il risultato potrebbe essere -infinito.

La soluzione migliore sarà il massimo tra le due opzioni precedenti.

$$N[i,r]=\begin{cases} N[i-1,r] & \text{se } r<c[i] \\ \max\{1+N[i-1,r-c[i]], N[i-1,r]\} & \text{altrimenti} \end{cases}$$

Si noti che l'espressione precedente richiede che, se $N[i - 1, r - c[i]]$ è $-\infty$, allora anche $(1 + N[i - 1, r - c[i]])$ sia $-\infty$.

Esercizio 2. Considerare il seguente grafo non orientato:

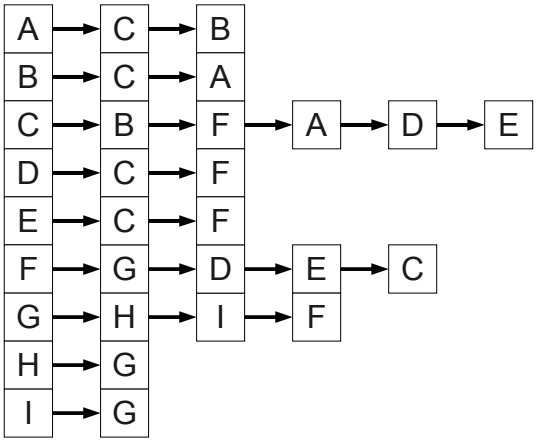


1. Gli archi in grassetto possono rappresentare un albero di visita ottenuto mediante una visita **in profondità** del grafo? In caso affermativo specificare il nodo di inizio della visita, e rappresentare il

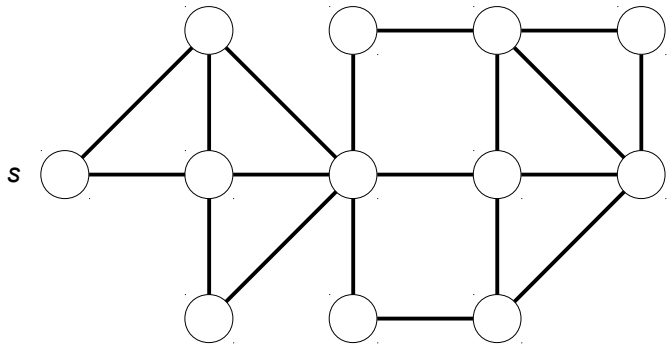
grafo mediante liste di adiacenza in modo tale che l'ordine in cui compaiono gli elementi nelle liste consenta all'algoritmo DFS di produrre esattamente l'albero mostrato.

2. Gli archi in grassetto possono rappresentare un albero di visita ottenuto mediante una visita **in ampiezza** del grafo? In caso affermativo specificare il nodo di inizio della visita, e rappresentare il grafo mediante liste di adiacenza in modo tale che l'ordine in cui compaiono gli elementi nelle liste consenta all'algoritmo BFS di produrre esattamente l'albero mostrato.

Soluzione. La risposta è affermativa in entrambi i casi. Per la visita in profondità si può ad esempio partire da A oppure B. Per la visita in ampiezza si può partire ad esempio da F.
In entrambi i casi (al di là del nodo di partenza, che è differente per la visita DFS e BFS) si può ottenere l'albero di mostrato nel testo utilizzando, ad esempio, la seguente rappresentazione con liste di adiacenza: Naturalmente altre soluzioni corrette erano possibili.



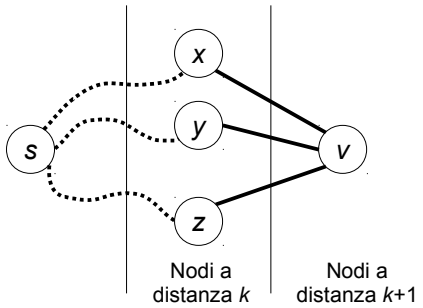
Esercizio 3. Scrivere un algoritmo efficiente per calcolare il numero di cammini minimi distinti che vanno da un nodo sorgente *s* a ciascun nodo *u* in un grafo non orientato $G = (V, E)$ non pesato e connesso. Due cammini si considerano diversi se differiscono per almeno per un arco; si noti che esiste un cammino minimo (il cammino vuoto) dal nodo *s* a se stesso. Applicare l'algoritmo al grafo seguente.



Soluzione. L'algoritmo di visita in ampiezza (BFS) può essere utilizzato per individuare un cammino minimo tra un nodo sorgente *s* e ciascun nodo raggiungibile da *s*; possiamo però estenderlo per calcolare il numero di cammini minimi distinti tra *s* e i nodi da esso raggiungibili. Ricordiamo che l'algoritmo BFS visita i nodi in ordine di distanza non decrescente dalla sorgente, ossia viene prima visitato il nodo *s* (che ha distanza 0 da se stesso), poi i nodi adiacenti a distanza 1, poi quelli a distanza 2 e così via. Supponiamo di aver già calcolato il numero di cammini minimi $c[u]$ per ogni nodo *u* che si trovi a distanza *k* dalla sorgente. Il numero di cammini minimi che portano ad un nodo *v* che si trova a distanza $k + 1$ può essere espresso come:

$$c[v] = \sum_{\{u,v\} \in E,\, d[u]=k} c[u]$$

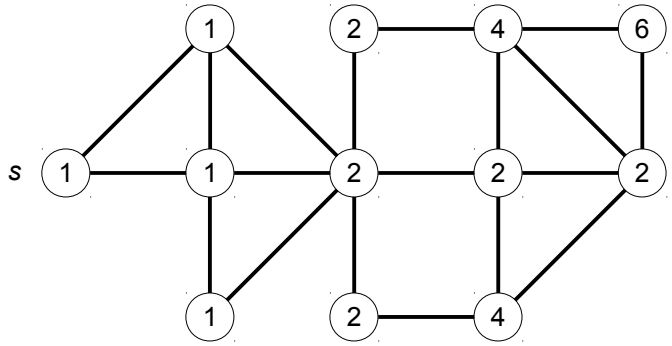
Ad esempio, consideriamo la situazione seguente in cui i nodi x, y, z si trovano a distanza k da s, e sono collegati con un arco al nodo v che si trova a distanza $k + 1$. Supponendo di aver già calcolato i numeri $c[x]$, $c[y]$ e $c[z]$ di cammini minimi distinti da s a (rispettivamente) x, y e z, allora possiamo raggiungere v mediante $c[x]$ cammini distinti che passano per x, $c[y]$ cammini distinti che passano per y e $c[z]$ cammini minimi distinti che passano per z.



È possibile calcolare il valore $c[v]$ per ogni nodo v mano a mano che il grafo viene visitato. Si pone inizialmente $c[v] \leftarrow 0$ per ogni v, ad eccezione della sorgente s per cui si pone $c[s] \leftarrow 1$ in quanto esiste un solo cammino minimo (il cammino vuoto) dalla sorgente a se stessa. Ogni volta che l'algoritmo BFS attraversa l'arco non orientato $\{u, v\}$ che conduce dal nodo u ad un nodo v a distanza $d[v] = d[u] + 1$, si imposta $c[v] \leftarrow c[v] + c[u]$.

```
CONTACAMMINIMINIMI( grafo G = (V, E), nodo s )
    integer d[1..n], c[1..n]
    integer u, v;
    for v ← 1 to n do
        d[v] ← +∞;
        c[v] ← 0;
    endfor
    Queue Q;
    c[s] ← 1;
    d[s] ← 0;
    Q.insert(s);
    while ( not Q.empty() ) do
        u ← Q.dequeue();
        foreach v adiacente a u do
            if ( d[v] = +∞ ) then
                d[v] ← d[u] + 1;
                Q.insert(v);
            endif
            if ( d[v] = d[u] + 1 ) then
                c[v] ← c[v] + c[u];
            endif
        endfor
    endwhile
```

Il costo dell'algoritmo CONTACAMMINIMINIMI è lo stesso di una visita in ampiezza, cioè $O(n + m)$.
Nell'esempio seguente etichettiamo ogni nodo del grafo con il numero di cammini minimi da s.



Esercizio 4. Una remota città sorge su un insieme di n isole, ciascuno identificato univocamente da un intero 1, …, n. Le isole sono collegate da ponti, che possono essere attraversati in entrambe le direzioni. Quindi possiamo rappresentare la città come un grafo non orientato $G = (V, E)$, dove V rappresenta l'insieme delle n isole ed E l'insieme dei ponti. Ogni ponte $\{u, v\}$ è in grado di sostenere un peso minore o uguale a $W[u, v]$. La matrice W è simmetrica (quindi il peso $W[u, v]$ è uguale a $W[v, u]$), e i pesi sono numeri reali positivi. Se non esiste alcun ponte che collega direttamente u e v, poniamo $W[u, v] = W[v, u] = +\infty$
Un camion di peso P si trova sull'isola s (sorgente) e deve raggiungere l'isola d (destinazione); per fare questo può servirsi unicamente dei ponti che siano in grado di sostenere il suo peso. Scrivere un algoritmo che, dati in input la matrice W, il peso P, nonché gli interi s e d, restituisca il numero minimo di ponti che è necessario attraversare per raggiungere d partendo da s, ammesso che ciò sia possibile. Stampare la sequenza di isolotti attraversati.

Soluzione. Utilizziamo un algoritmo di visita in ampiezza modificato opportunamente per evitare di attraversare i ponti che non sosterrebbero il peso P.

```
ATTRAVERSAISOLE( real W[1..n, 1..n], real P, integer s, integer d ) → integer
    integer parent[1..n], dist[1..n];
    Queue Q;
    for v ← 1 to n do
        parent[v] ← -1;
        dist[v] ← +∞;
    endfor
    dist[s] ← 0;
    Q.ENQUEUE(s);
    while ( ! Q.ISEMPTY() ) do
        integer u ← Q.DEQUEUE();
        if ( u = d ) then
            break;                  // esci dal ciclo se si estrae il nodo destinazione dalla coda
        endif
        for v ← 1 to n do
            if ( W[u,v] ≥ P and dist[v] = +∞ ) then
                dist[v] ← dist[u]+1;
                parent[v] ← u;
                Q.ENQUEUE(v);
            endif
        endfor
    endwhile
    if ( dist[d] = +∞ ) then
        print "nessun percorso"
    else
        integer i ← d;
        while ( i ≠ -1 ) do
            print i;
            i ← parent[i];
        endwhile;
    endif
```

```
    return dist[d];
```