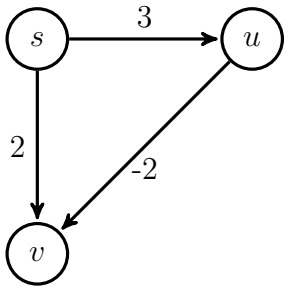
Esercitazione 7: Dynamic Programming

Giacomo Paesani

April 28, 2025

Esercizio 1 (24.4-2, [1])**.** Sia $G = (V, E)$ un grafo orientato con pesi sugli archi, che possono essere anche negativi ma in cui non sono presenti cicli di peso negativo. Dimostrare che l'algoritmo di Dijkstra applicato su grafi di questo tipo non calcola necessariamente i cammini di costo minimo tra la radice e gli altri vertici del grafo.

Soluzione 1. L'algoritmo di Dijkstra funziona correttamente solo per grafi che hanno tutti gli archi di peso positivo. Consideriamo infatti l'esempio dato dal grafo diretto in figura. Infatti, nella seconda iterazione del ciclo **while** viene estratto il vertice v che si pensa come stabilizzato. In pratica l'output è il seguente: $dist[s] = 0$, $dist[u] = 3$ e $dist[v] = 2$ quando il risultato corretto si ottiene notando che $dist[v]$ deve essere necessariamente pari a 1 (passando per u). Questo fenomeno non avviene se G ha tutti gli archi di valore positivo perché ulteriori cammini (come quello passante per u nell'esempio) hanno necessariamente lunghezza non minore. □



Esercizio 2 (24.3-6, [1])**.** Sia $G = (V, E)$ un grafo non diretto che rappresenta un network di comunicazioni. Ad ogni arco uv viene associato un valore

$r(uv)$, che è un numero reale con $0 \leq r(uv) \leq 1$ che rappresenta l'affidabilità dell'arco nel eseguire una comunicazione tra i vertici u e v. Interpretiamo $r(uv)$ come la probabilità che la trasmissione tra u e v abbia successi e supponiamo che tali probabilità sono indipendenti tra loro. Fornire uno pseudocodice che dato G e un vertice s restituisce l'albero dei cammini più affidabili da s.

Soluzione 2. L'idea è quella di modificare l'algoritmo di Dijkstra in maniera che restituisce l'albero dei cammini più affidabili da s. Sia $P = v_0, v_1, \ldots, v_k$ un cammino di G, allora l'affidabilità di P è data da $\mathcal{A}(P) = r(v_0v_1) \cdot r(v_1v_2) \cdot \ldots r(v_{k-1}v_k)$ dato che le probabilità degli eventi elementari sono indipendenti. Allora per ogni vertice $v \in V$ dobbiamo calcolare l'affidabilità massima di un cammino P da s a v.

Algorithm 1 Algoritmo di Dijkstra modificato

Input: $G = (V, E)$ grafo diretto e un vertice s
Output: vettore dei migliori affidabilità possibili da s ad ogni altro vertice v in G

```
1: global variables
2:     Parent ↤ array dei padri
3:     A ↤ array delle affidabilità, inizializzato a 0
4: end global variables
5: function DijkstraSafe(G,s)
6:     A[s] = 1
7:     Parent[s] = s
8:     S = ∅
9:     Q = V
10:    while Q ≠ ∅ do
11:        u = RemoveMax(Q)
12:        S = S ∪ {u}
13:        for v ∈ N(u) do
14:            Relax(u,v)
15:    return A
16: function Relax(u,v)
17:    if A[v] < A[u] · r(uv) then
18:        A[v] = A[u] · r(uv)
19:        u = Parent[v]
20:    return
```

La soluzione proposta è esposta nell'Algoritmo 1: una versione dell'algoritmo di Dijkstra in cui in seguito sono date le modifiche. Ad ogni iterazione del ciclo **while** in Linea 10, selezioniamo il vertice u di Q con $A[u]$ massimo e lo spostiamo in S stabilizzandolo. Infine la funzione di rilassamento RELAX calcola l'affidabilità di un cammino come il prodotto delle affidabilità dei suoi archi. □

Esercizio 3 (15.4-5:6, [1])**.** Fornire in pseudo-codice un algoritmo che data una sequenza finita di numeri interi X restituisce la lunghezza della più lunga sotto-sequenza strettamente crescente Y. Se, ad esempio, abbiamo che la sequenza X è data da $(1, 3, 8, 5, 4, 2, 6, 0, 1, 2, 8, 9, 5)$ allora si ottiene $Y = (1, 3, 4, 6, 8, 9)$. Implementare questo algoritmo in modo che il tempo di esecuzione sia al più $\mathcal{O}(n^2)$ (ma si può fare anche in $\mathcal{O}(nlog(n))$). Come deve essere modificato l'algoritmo per far si che restituisca una sotto-sequenza strettamente crescente di lunghezza massima?

Soluzione 3. L'idea è di usare la tecnica della programmazione dinamica per risolvere questo problema in maniera efficiente. Si costruisce un vettore A che viene continuamente aggiornato con la seguente proprietà: il valore $A[k]$ è il più piccolo valore di X per cui una sotto-sequenza di lunghezza k finisce. In aggiunta costruiamo la pila Y per la soluzione.

Una soluzione è data dall'Algoritmo 2. Per la proprietà data in precedenza, in ogni momento dell'esecuzione il vettore A è sempre in ordine strettamente crescente. La routine principale consiste nel fare una scansione sequenziale di X e per ogni suo elemento x si considera se e dove inserire tale elemento nel vettore A. Se x è più grande di tutti gli elementi già inseriti in A, allora possiamo sostituire con x la coordinata di A che vale $+\infty$ con indice minimo: questo possiamo sempre farlo perché x allunga la soluzione che si aveva precedentemente di 1. Supponiamo altrimenti che esiste un indice ℓ tale che $A[\ell-1] < x < A[\ell]$ allora possiamo porre $A[\ell] = x$: la sotto-sequenza ottenuta ha sempre la stessa lunghezza ma questa sostituzione permette di a più elementi *futuri* di X di essere aggiunti ad A e allungare la sotto-sequenza massima. Se invece esiste un indice ℓ tale che $A[\ell] = x$ allora non eseguiamo alcuna sostituzione. La costruzione del vettore A viene completata nel ciclo **while** di Linea 7 e la lunghezza di una soluzione è data dal massimo indice di A che ha coordinata *finita* e questa viene calcolata in Linea 16.

Supponiamo ora che vogliamo esibire anche la sotto-sequenza che è soluzione dell'esercizio e per far ciò viene costruito un vettore ausiliario B per poter ricostruire la sotto-sequenza desiderata Y. Per come il vettore A è stato

costruito, la soluzione Y ha come ultimo elemento il valore finito maggiore in A. L'idea è che se un elemento $X[i]$ di X viene inserito in A nella posizione ℓ allora il vettore B nella coordinata i contiene l'indice di $X[i]$ in A cioè ℓ. E' utile notare che mentre le coordinate di A cambiano durante l'algoritmo, quelle di B restano le stesse. Allora si scorre il vettore B *da destra a sinistra*, cioè da $B[n]$ a $B[1]$, e si individua una sotto-sequenza decrescente di B di lunghezza massima e i valori corrispondenti di X, cioè quelli aventi gli elementi della sotto-sequenza di B come indici. formano la soluzione richiesta. Trovare la sotto-sequenza ordinata di B di lunghezza massima è semplice perché è quella di lunghezza t con i valori da 1 a t, dove t è l'indice in A del più grande elemento finito di A. Ad esempio: se X è data da $(1, 3, 8, 5, 4, 2, 6, 0, 1, 2, 8, 9, 5)$ allora si ottiene $A = (0, 1, 2, 5, 8, 9)$ e $B = (1, 2, 3, 3, 3, 2, 4, 1, 2, 3, 5, 6, 4)$ e infine $Y = (1, 3, 4, 6, 8, 9)$.

Concludiamo la soluzione con un analisi del tempo di esecuzione. I ciclo **for** (Linea 4) e i cicli **while** (Linee 71618) compiono al più n iterazioni e il costo di ogni iterazione è controllato dalla complessità della ricerca binaria sul vettore ordinato A e quindi $\mathcal{O}(\log(n))$. Quindi la complessità complessiva dell'algoritmo **LIS** è di $\mathcal{O}(n \log(n))$. □

References

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

Algorithm 2 Algoritmo per calcolare la sotto-sequenza strettamente crescente di una sequenza

Input: sequenza X
Output: sottosequenza Y strettamente crescente di X di lunghezza massima

```
1: function LIS(X)
2:     n = |X|
3:     A[0] = −∞
4:     for i = 1, ..., n do
5:         A[i] = +∞
6:     i = 1
7:     while i ≤ n do
8:         ℓ=BinarySearch(X[i],A)
9:         if A[ℓ − 1] < X[i] and X[i] < A[ℓ] then
10:            A[ℓ] = X[i]
11:            B[i] = ℓ
12:        if A[ℓ] = X[i] then
13:            B[i] = −1
14:        i = i + 1
15:    t = 1
16:    while A[t + 1] ≠ +∞ do
17:        t = t + 1
18:    while n > 0 do
19:        if B[n] = t then
20:            X[n] = push(Y)
21:            t = t − 1
22:        n = n − 1
23:    return Y
```