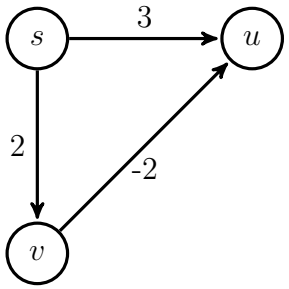
# Esercitazione 7: Dynamic Programming

Giacomo Paesani

May 8, 2024

**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 nono calcola necessariamente i cammini di costo minimo tra la sorgente 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 $(u, v)$ viene associato un

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

**Soluzione 2.** L'idea è quella di modificare l'algoritmo di Dijkstra in maniera che restituisce l'albero dei cammini più sicuri da $s$. Sia $P = v_0, v_1, \ldots, v_k$ un cammino di $G$, allora l'affidabilità di $P$ è data da $\mathcal{A}(P) = r(v_0, v_1) \cdot r(v_1, v_2) \cdot \ldots r(v_{k-1}, v_k)$ 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:** cammino più sicuro da $s$ ad ogni altro vertice $v$ in $G$

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

La soluzione proposta è esposta nell'Algoritmo 1: una versione dell'algoritmo di Dijkstra in cui in seguito sono date le modifiche. Con la funzione INITIALIZE, poniamo $A[v] = 0$ (Linea 16) per tutti i vertici $v \in V$ ad eccezione di $A[s] = 1$ (Linea 17): passare un'informazione da $s$ ad $s$ si può fare con probabilità 1 seguendo il percorso di lunghezza 0. Ad ogni iterazione del ciclo **while** in Linea 9, 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** (24.2-3, [1])**.** Dato un grafo orientato $G = (V, E)$ e con archi pesati da una funzione $w$, senza cicli orientati di peso negativo, e sia $m$ il massimo del numero minimo di archi di un cammino minimo da un vertice $s$ a $v$, per ogni vertice $v \in V$. Fornire uno pseudo-codice modificando l'algoritmo di *Bellman-Ford* in modo che vengono fatte al più $m + 1$ iterazioni, anche se $m$ non è noto a priori.

**Soluzione 3.** L'idea principale di questo esercizio è che dopo $k$ iterazioni di rilassamento sugli archi del grafo $G$, l'algoritmo ha già stabilizzato, cioè ha già trovato il valore finale e corretto di $dist[v]$ per tutti i vertici $v$ tali che il cammino minimo da $s$ a $v$ ha al più $k$ archi. Tale affermazione si può rapidamente dimostrare per induzione su $k$.

Per $k = 0$, questo è banalmente vero: infatti $s$ è l'unico vertice per il cui il cammino minimo che lo collega ad $s$ ha 0 archi e $dist[s]$ è inizializzata a 0 (Linea 24). Supponiamo ora che $k > 0$ e per ipotesi induttiva sappiamo che l'algoritmo ha stabilizzato tutti le distanze relative ai vertici per cui esiste un cammino minimo con al più $k - 1$ archi. Sia ora $v$ un vertice tale che esiste un cammino minimo $P$ da $s$ a $v$ di lunghezza minima con esattamente $k$ archi. Allora esiste un cammino minimo $P'$ da $s$ a $u$ di lunghezza minima con esattamente $k - 1$ archi, dove $u = Parent[v]$ è il padre di $v$ nel albero dei cammini minimi da $s$. Quindi, alla $k$-esima iterazione si crea il cammino $P$ concatenando $P'$ con l'arco $(u, v)$ che realizza un cammino di peso $dist[v]$ corretto di lunghezza $k$.

La soluzione proposta è Algoritmo 2 che è ottenuto modificando l'algoritmo di Bellman-Ford. Viene introdotta la variabile booleana $Fix$ che ha il compito, ad ogni iterazione, di capire se è stata modificato qualche coordinata dell'array $dist$. Infatti, se durante tutta un'iterazione di rilassamento tutti i valori restano invariati (Linea 11) allora si può facilmente dedurre che questi

valori hanno raggiunto un punto fisso e sono quelli finali: l'algoritmo termina e i valori su *dist* sono quelli corretti (Linea 12). Supponiamo quindi che ad ogni iterazione, fino alla $n-1$-esima, almeno una coordinata di *dist* cambia e quindi la variabile $Fix$ assume sempre il valore `FALSE`: la continua alterazione di coordinate dell'array *dist* è causata dall'esistenza di un ciclo di peso negativo raggiungibile da $s$ e quindi l'algoritmo ritorna correttamente `FALSE`(Linea 13). □

**Esercizio 4** (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, $X = (1, 3, 8, 5, 4, 2, 6, 0, 1, 2, 8, 9, 5)$ 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 4.** 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 3. 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 di $A$ 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 $\ell$ 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 $\ell$ 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 vengono costruiti dei vettori ausiliari $C$ e $D$ 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 $\ell$ allora il vettore $B$ nella coordinata $i$ contiene l'indice di $X[i]$ in $A$ cioè $\ell$. 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 crescente 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 di $B$ di lunghezza massima è più facile perché è quella di lunghezza $t$ con i valori da 1 a $t$. □

Esempio: $X = (1, 3, 8, 5, 4, 2, 6, 0, 1, 2, 8, 9, 5)$ 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 $Y = (1, 3, 4, 6, 8, 9)$

# References

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

**Algorithm 2** Algoritmo per *fissare* il numero di iterazioni dell'algoritmo di Bellman-Ford.

**Input:** $G = (V, E)$ grafo diretto e con archi pesati con funzione $w$, vertice $s$
**Output:** `TRUE` se $G$ non ha cicli di peso negativo e `FALSE` altrimenti

```
1: global variables
2:     Parent ↤ array dei padri
3:     dist ↤ array delle distanze pesate
4: end global variables
5: function Short-Bellman-Ford(G,w,s)
6:     Initialise(G,s)
7:     for i = 1, ..., |V| do
8:         Fix = TRUE
9:         for (u, v) ∈ E do
10:            Fix = Fix ∧ Relax(u,v,w)
11:        if Fix==TRUE then
12:            return TRUE
13:    return FALSE
14: function Relax(u,v,w)
15:    Fix=TRUE
16:    if dist[v] > dist[u] + w(u, v) then
17:        dist[v] = dist[u] + w(u, v)
18:        Parent[v] = u
19:        Fix=FALSE
20:    return Fix
21: function Initialise(G,s)
22:    for v ∈ V do
23:        dist[v] = +∞
24:    dist[s] = 0
```

**Algorithm 3** 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
```