Esercitazione 8: More on Dynamic Programming

Giacomo Paesani

May 7, 2025

Esercizio 1 (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 di peso minimo dalla radice 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 1. 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 esiste cammino di peso 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 di peso 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 di peso minimo con al più $k - 1$ archi. Sia ora v un vertice tale che esiste un cammino di peso 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.

Algorithm 1 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: $\quad Parent \hookleftarrow$ array dei padri
3: $\quad dist \hookleftarrow$ array delle distanze pesate
4: **end global variables**
5: **function Short-Bellman-Ford**(G,w,s)
6: $\quad$ INITIALISE(G,s)
7: $\quad$ **for** $i = 1, \ldots, |V|$ **do**
8: $\qquad Fix =$ `TRUE`
9: $\qquad$ **for** $(u, v) \in E$ **do**
10: $\qquad\quad Fix = Fix \wedge$ RELAX(u,v,w)
11: $\qquad$ **if** Fix==`TRUE` **then**
12: $\qquad\quad$ **return** `TRUE`
13: $\quad$ **return** `FALSE`
14: **function Relax**(u,v,w)
15: $\quad Fix$=`TRUE`
16: $\quad$ **if** $dist[v] > dist[u] + w(u, v)$ **then**
17: $\qquad dist[v] = dist[u] + w(u, v)$
18: $\qquad Parent[v] = u$
19: $\qquad Fix$=`FALSE`
20: $\quad$ **return** Fix
21: **function Initialise**(G,s)
22: $\quad$ **for** $v \in V$ **do**
23: $\qquad dist[v] = +\infty$
24: $\quad dist[s] = 0$

La soluzione proposta è Algoritmo 1 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 modificata 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). $\square$

Esercizio 2 (16.2-2, [1])**.** Il problema dello *zaino* (knapsack problem) è un classico problema di ottimizzazione definito nella seguente maniera. Data una collezione di oggetti $A = \{1, \dots, n\}$, all'oggetto i-esimo vengono associati numeri interi v_i e w_i che rappresentano il valore e il peso, rispettivamente. Allora dato un intero C, il problema richiede un insieme $U \subseteq A$ tale che $\sum_{a_i \in U} w_i \leq C$ e $\sum_{a_i \in U} v_i$ è massima, cioè l'obiettivo è di selezionare il sottoinsieme di oggetti che ha un peso totale che non supera C e che massimizza il valore complessivo.

Fornire lo pseudo-codice di un algoritmo che risolve il problema dello zaino in tempo $\mathcal{O}(n \cdot C)$. Usare un approccio di programmazione dinamica. Come devo modificare l'algoritmo per avere anche gli elementi che compongono una soluzione?

Soluzione 2. Questo stesso esercizio è stato già fatto a lezione, riportiamo comunque la soluzione per chi volesse avere un'ulteriore soluzione. Per risolvere questo esercizio utilizzando la programmazione dinamica, dobbiamo iniziare a pensare di risolvere sotto-problemi partendo dal più semplice. Allora la domanda è: considerando i primi n oggetti qual'è il valore massimo che possiamo ottenere rispettando il limite di peso complessivo C? Ad esempio, per il caso $n = 1$ la risposta è semplice: il valore massimo è v_1 se $w_1 \leq C$ e 0 altrimenti. Per risolvere il problema con input n e C, è necessario prima aver risolto e guardare i risultati per input $(n-1, C)$ e $(n-1, C-w_n)$ cioè distinguere se inserire l'n-esimo oggetto o no.

Allora pensiamo ad una matrice A di dimensioni $n \times C$, dove la coordinata $A[n', C']$ assume come valore la soluzione del sotto-problema con input (n', C') una volta calcolata: la matrice A, memorizzando le soluzioni dei sotto-problemi, aiuta nel far si che ogni sotto-problema deve essere risolto al più una volta e quindi a diminuire il tempo di esecuzione. La soluzione proposta è data dall'Algoritmo 2. L'algoritmo inizia con dei controlli per evitare calcoli superflui: dal valore di $A[n, C]$ si può capire se il sotto-problema è stato già risolto, in tal caso non c'è bisogno di fare altro (Linea 7). I casi base sono costituiti quando non ci sono elementi da aggiungere o il peso a disposizione è nullo e in questi casi la risposta è banalmente 0 (Linea 9). Supponiamo ora che il peso dell'oggetto n-esimo supera il peso a disposizione, cioè se $w_n > C$, allora dobbiamo necessariamente escludere tale oggetto nella soluzione e quindi riportare il risultato del sotto-problema con input $(n-1, C)$ (Linea 11).

Algorithm 2 Algoritmo per risolvere il 0/1-Knapsack Problem in maniera efficiente

Input: interi n e C

Output: il valore massimo di oggetti presi tra i primi n aventi peso complessivo al più C

```
1: global variables
2:     V ↤ array dei valori
3:     W ↤ array dei pesi
4:     A ↤ matrice di soluzione, inizializzata a −∞
5: end global variables
6: function KS(n,C)
7:     if A[n, C] ≠ −∞ then
8:         return A[n, C]
9:     if n == 0 or C == 0 then
10:        r = 0
11:    else if W[n] > C then
12:        r =KS(n − 1,C)
13:    else
14:        t =KS(n − 1,C)
15:        t′ = V[n]+KS(n − 1,C − W[n])
16:        r = max{t, t′}
17:    A[n, C] = r
18:    while n > 0 do
19:        if A[n, C] == V[n] + A[n − 1, C − W[n]] then
20:            n = Add(B)
21:            C = C − W[n]
22:        n = n − 1
23:    return r
```

Consideriamo, finalmente, il caso in cui tutti e tre i controlli precedenti (rappresentato da Linea 13 a 16) sono negativi. Qui si confrontano i risultati corrispondenti a includere o meno l'n-esimo oggetto nella soluzione: se l'n-esimo oggetto non fa parte della soluzione allora si considera la migliore soluzione considerando solo i primi $n-1$ oggetti e la stessa capacità C (Linea 14); se invece l'n-esimo oggetto fa parte della soluzione allora si considera la migliore soluzione considerando solo i primi $n-1$ oggetti e la capacità

C ridotta di w_n, cioè del peso di n (Linea 15).

Si conclude, prendendo il massimo della soluzione tra avere o no l'n-esimo oggetto 16, lo si salva in $A[n, C]$ (Linea 17) e si ritorna tale valore (Linea 23).

Supponiamo ora che vogliamo anche ritornare un insieme di oggetti che costituisce la soluzione al problema originario. Allora è sufficiente analizzare la matrice A, partendo dal valore $A[n, C]$ e capire se è stato ottenuto aggiungendo l'n-esimo oggetto o meno. Questo viene fatto dal ciclo **while** in Linea 18 aggiungendo gli oggetti nel insieme B, aggiornando i valori di n e C, quando necessario. $\square$

Esercizio 3. Sia $X = (X[1], \ldots, X[n])$ una stringa di n elementi e $Y = (Y[1], \ldots, Y[m])$ una stringa di m elementi. Definiamo $d(X, Y)$ come il minimo numero di operazioni necessarie per convertire X in Y, dove le operazioni permesse sono le seguenti:

- $inserire(k, z)$ il simbolo z alla posizione k (e far slittare tutti i successivi di una posizione a destra) in X;
- $rimuovere(k)$ il simbolo in posizione k (e far slittare tutti i successivi di una posizione a sinistra) in X;
- $sostituire(k, z)$ assegnare al simbolo in posizione k il valore z in X.

Ad esempio, se $X = (C, O, L, T)$ e $Y = (C, I, T)$, allora $d(X, Y) = 2$ e prodotta, ad esempio, da $sostituire(2, I)$ e $rimuovere(3)$. Rispondere alle seguenti richieste:

1. dimostrare che la funzione d è una distanza, cioè che $d(X, X) = 0$, $d(X, Y) = d(Y, X)$ e $d(X, Y) \leq d(X, Z) + d(Z, Y)$;
2. fornire in pseudo-codice un algoritmo che, date due stringhe X e Y di lunghezza n e m rispettivamente, calcoli $d(X, Y)$ ed esibisca le operazioni eseguite, in tempo $\mathcal{O}(n \cdot m)$.

Soluzione 3. Questo stesso esercizio è stato già fatto a lezione, riportiamo comunque la soluzione per chi volesse avere un'ulteriore soluzione. Iniziamo dimostrando che d è una distanza e cioè è una funzione che soddisfa le tre proprietà delle distanze: chiaramente per convertire X in X non serve alcuna operazione quindi la distanza è pari a zero. Supponiamo di aver calcolato il minimo numero di operazioni per convertire X in Y e abbiamo anche a disposizione la lista delle operazioni. Allora per convertire Y in X facciamo i seguenti cambi: ogni operazione $inserire(k, z)$ diventa $rimuovere(k)$; ogni operazione $rimuovere(k)$ diventa $inserire(k, Y[k])$; infine ogni operazione $sostituire(k, z)$ diventa $sostituire(k, Y[k])$. Allora il numero di operazioni rimane lo stesso e quindi è dimostrata la simmetria della funzione d. E' allo stesso modo chiaro che il numero minimo di operazioni per convertire X in Y è non superiore al numero minimo di operazioni per convertire X in Y *passando* per Z.

L'idea per risolvere questo esercizio è di risolvere sotto-problemi, partendo dal più semplice, in maniera da poter lavorare efficacemente su quelli più elaborati. Allora la domanda è: considerando i primi n elementi di X, $X_n = \{X[1], \ldots, X[n]\}$ e m elementi di Y, $Y_m = \{Y[1], \ldots, Y[m]\}$, quanto vale $d(X_n, Y_m)$? Infatti, per risolvere il problema con input (n, m) è necessario prima aver risolto e guardare i risultati per input $(n-1, m)$, $(n, m-1)$ e $(n-1, m-1)$ cioè distinguendo tra le diverse operazioni. Allora pensiamo ad una matrice A di dimensioni $n \times m$, dove la coordinata $A[n', m']$ assume come valore la soluzione del sotto-problema con input (n', m') una volta calcolata: la matrice A, memorizzando le soluzioni dei sotto-problemi, aiuta nel far si che ogni sotto-problema deve essere risolto al più una volta e quindi a diminuire il tempo di esecuzione.

La soluzione proposta è data dall'Algoritmo 3. Lo pseudo-codice inizia analizzando alcuni casi base. Se la soluzione con input (n, m) è stata già calcolata, allora è stata già salvata in $A[n, m]$ e quindi non c'è bisogno di calcolarla nuovamente (Linea 7). Se $n = 0$ (o $m = 0$) allora la soluzione al problema con input $(0, m)$ (o $(n, 0)$) è semplicemente m (o n): infatti una delle due parole è vuota e quindi per fare la conversione si usano solo *rimuovere* o *inserire*. Allo stesso modo se $X[n] = Y[m]$ allora il problema si riduce al caso $(n-1, m-1)$ (Linea 13). Supponiamo infine che nessuno dei casi precedenti è applicabile. Analizziamo i vari casi su cui possiamo far si che l'ultimo simbolo di X sia uguale a quello di Y: o sostituiamo il simbolo di X con quello di Y, o inseriamo quello di Y in X o rimuoviamo il simbolo di X. Per ognuno di questi casi vengono analizzate e calcolate le soluzioni corrispondenti.

Supponiamo ora che vogliamo anche ritornare l'insieme di operazioni minimo per convertire la stringa X in Y. Per fare questo è sufficiente guardare la matrice A e procedere da $A[n, m]$. Un esempio viene mostrato in figura.

$$\begin{array}{cccccc} & & C & I & T \\ & 0 & 1 & 2 & 3 \\ C & 1 & 0 & 1 & 2 \\ O & 2 & 1 & 1 & 2 \\ L & 3 & 2 & 2 & 2 \\ T & 4 & 3 & 3 & 2 \end{array}$$

Algorithm 3 Algoritmo per risolvere l'Edit Distance Problem in maniera efficiente

Input: interi n e m

Output: l'edit distance tra X e Y

```
global variables
    X ↤ array di partenza
    Y ↤ array di arrivo
    A ↤ matrice di soluzione, inizializzata a −∞
end global variables
function ED(n,m)
    if A[n, m] ≠ −∞ then
        return A[n, m]
    if n = 0 then
        r = m
    else if m = 0 then
        r = n
    else if X[n] == Y[m] then
        r =ED(n − 1,m − 1)
    else
        t =ED(n − 1,m − 1)
        t' =ED(n,m − 1)
        t'' =ED(n − 1,m)
        r = min{t, t', t''} + 1
    A[n, m] = r
    while n ≠ 0 and m ≠ 0 do
        if X[n] == Y[m] then
            n = n − 1
            m = m − 1
        else if A[n, m] = A[n − 1, m] + 1 then
            rimuovere(n) = Add(B)
            n = n − 1
        else if A[n, m] = A[n, m − 1] + 1 then
            inserire(n + 1, Y[m]) = Add(B)
            m = m − 1
        else
            sostituire(n, Y[m]) = Add(B)
            n = n − 1
            m = m − 1
    return r
```

Esercizio 4. Sia $X = (x_1, \dots, x_n)$ una stringa di n elementi. Definiamo $d_p(X)$ come il minimo numero di cancellazioni di sotto-stringhe palindrome di X per ottenere la stringa vuota. Ad esempio, se $X = (1, 3, 3, 4, 3, 5, 2, 7, 1, 7, 2, 3)$ allora $d_p(X) = 4$ rimuovendo le seguenti stringhe palindrome $(3, 4, 3)$, $(2, 7, 1, 7, 2)$, $(3, 5, 3)$ e infine (1). Fornire in pseudo-codice un algoritmo che, data una stringa X, calcoli $d_p(X)$ ed esibisca le operazioni eseguite, in tempo $\mathcal{O}(n^3)$.

Algorithm 4

Soluzione 4. Input:
Output:

```
global variables
    X ↤ array
    A ↤ matrice di soluzione, inizializzata a +∞
end global variables
function PaDel(i,j)
    if A[i, j] ≠ +∞ then
        return A[i, j]
    if i > j then
        r = i + j
    else if j − i == 1 or j − i == 0 then
        r = j − i
    else
        t = 1+PaDel(i + 1,j)
        t′ = +∞
        k = i + 1
        while k ≤ j do
            if X[i] == X[k] then
                t′ = min{t′,PaDel(i + 1,k − 1)+PaDel(k + 1,j)}
            k = k + 1
        if X[i] == X[i + 1] then
            t′′ = 1+PaDel(i + 2,j)
        else
            t′′ = i + j
        r = min{t, t′, t′′}
    A[i, j] = r
    return r
```

Seguiamo l'idea, già intrapresa negli esercizi precedenti, di risolvere questo problema passando dalla ricostruzione della soluzione a partire dai sotto-problemi più semplici. Indichiamo con $X[i, j]$ la sotto-stringa di X che va dall'indice i all'indice j e con $A[i, j] = d_p[X[i, j]]$ la soluzione del problema con input $X[i, j]$. Distinguiamo i tre seguenti casi:

(1) se $X[i]$ viene cancellato da solo allora si guarda il problema con input $X[i + 1, j]$;

(2) se $X[i]$ viene cancellato come elemento di una sotto-stringa palindroma, allora dobbiamo cercare tutti gli indici $i \leq k \leq j$ tali che $X[i] = X[k]$ (infatti ogni stringa palindroma deve necessariamente iniziare e finire con lo stesso simbolo) e riduciamo il problema alla soluzione dei due sotto-problemi con input $X[i, k - 1]$ e $X[k + 1, j]$;

(3) se $X[i]$ e $X[i + 1]$ sono rappresentati dallo stesso simbolo, li si può cancellare insieme e ridurre al problema con input $X[i + 2, j]$.

La soluzione proposta è data dall'Algoritmo 4. I casi esaminati nelle Linee 6, 8 e 10 sono casi base e vengono risolti rapidamente: se la soluzione per il problema è stata già calcolata, allora non vengono fatte ulteriori operazioni, se il valore di i e j è errato allora si salva un valore che indica l'*incompatibilità* dell'input; se, infine, la stringa $X[i, j]$ ha lunghezza al più uno, allora si salva il valore di tale lunghezza. Consideriamo ora il caso in cui nessuno dei precedenti casi si applica. Allora t (Linea 13) contiene il valore del caso (1), t' (Linea 18) contiene il valore del caso (2) e infine t'' (Linea 21) contiene il valore del caso (3). Una volta calcolate le soluzioni per i tre casi li si confronta e si salva il minimo in r tra di essi (Linea 24); il valore ottenuto viene salvato in $A[i, j]$ e infine ritornato.

Una volta trovata la soluzione e quindi trovato il valore di $A[i, j]$, possiamo procedere a ritroso per capire le operazioni di cancellazione che sono state usate per ottenere la soluzione (questa parte non è scritta nel codice proposto per brevità). Per fare questa ricostruzione, è necessario capire da dove *proviene*, cioè quale caso è stato scelto/applicato e quali sono le sotto-stringhe palindrome che sono cancellate.

Concludiamo la soluzione di questo esercizio con un'analisi del tempo di esecuzione dell'algoritmo proposto. Potenzialmente è necessario calcolare tutti i valori di $A[i', j']$, con $i \leq i' \leq j' \leq j$ (che sono al più $(j - i)^2$) e per ognuno di questi al più $j - i - 2$ casi (2). Allora se poniamo $n = j - i$ allora abbiamo che la complessità totale è di $\mathcal{O}(n^3)$. $\square$

Esercizio 5. Fornire in pseudo-codice un algoritmo che data una sequenza finita di numeri interi X restituisce la lunghezza della più lunga sotto-sequenza alternata Y. Quindi 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 $Y = (3, 8, 2, 6, 0, 9, 5)$. Implementare questo algoritmo in modo che il tempo di esecuzione sia al più $\mathcal{O}(n^2)$ (ma si può fare anche in $\mathcal{O}(n)$). Come deve essere modificato l'algoritmo per far si che restituisca una sotto-sequenza strettamente crescente di lunghezza massima?

Algorithm 5

Soluzione 5. Input:
Output:

```
global variables
end global variables
function AlternateSubSequence(X,n)
    inc = 1
    dec = 1
    i = 1
    while i ≤ n do
        if X[i] > X[i + 1] then
            dec = inc + 1
        else if X[i] < X[i + 1] then
            inc = dec + 1
        i = i + 1
    return max{inc, dec}
```

Questo problema può essere risolto in tempo $\mathcal{O}(n)$. Infatti è sufficiente scorrere il vettore X e salvare delle informazioni nelle variabili *inc* e *dec*: *inc* indica la lunghezza della più lunga sotto-sequenza alternata di $X[i]$ al passo i-esimo dell'algoritmo tale che l'ultimo valore di questa sequenza è maggiore del precedente e *dec* indica la lunghezza della più lunga sotto-sequenza alternata di $X[i]$ al passo i-esimo dell'algoritmo tale che l'ultimo valore di questa sequenza è minore del precedente. Come devo però aggiornare questi valori? Il valore di *inc* deve essere aggiornato se e solo se l'ultimo elemento della sequenza alternata era più piccolo dell'elemento precedente, cioè se sto partendo dalla sotto-sequenza rappresentata da *dec*, e valore di *dec* deve essere aggiornato se e solo se l'ultimo elemento della sequenza alternata era più grande dell'elemento precedente, cioè se sto partendo da la sotto-sequenza rappresentata da *inc*. Lo pseudo-codice è rappresentato nel Algortimo 5. $\square$

References

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