Analisi e Progetto di Algoritmi

Elia Ronchetti
@ulerich

2023/2024

Indice

Capitolo 1

Programmazione Dinamica - DP

La programmazione dinamica (DP - Dynamic Programming) è una tecnica che (come il Divide et Impera), risolve i problemi combinando le soluzioni dei sottoproblemi.
Divide et Impera è ottimo quando i sottoproblemi da risolvere sono indipendenti, mentre DP è efficace quando i sottoproblemi non sono indipendenti e quindi hanno in comune dei sottosottoproblemi e le tecniche di risoluzione top-down risultano quindi inefficienti (chiamate ripetute). La programmazione dinamica si applica tipicamente ai **problemi di ottimizzazione**.

1.1 Problemi di ottimizzazione

Sono problemi dove ci sono molte soluzioni possibile. Ogni soluzione ha un valore e e si vuole trovare una solzuione con il valore ottimo. Ci possono essere più soluzioni che raggiungono il valore ottimo.

1.2 Il processo di sviluppo

Il processo di sviluppo è diviso in 4 fasi:

- Caratterizzare la struttura di una soluzione ottima
- Definire in modo ricorsivo il valore di una soluzione ottima
- Calcolare il valore di una soluzione ottima, di solito con uno schema bottom-up (dal basso verso l'alto, risulta spesso più efficiente rispetto a top-down)

- Costruire una soluzione ottima dalle informazioni calcolate

1.3 Esempio - Fibonacci

Classico esempio è l'esecuzione di Fibonacci. Utilizzando la ricorsione pura si effettuano più volte le stesse chiamate (perchè i sotto-numeri sono gli stessi). Se invece utilizziamo la DP, con un approccio Bottom-Up ci dobbiamo chiedere, ma chi è Fibonacci di n? è Fibonacci di (1)+Fibonacci(2)+ ... + Fibonacci(n). In pratica inizio a calcolare le soluzioni dal sottoproblema più piccolo a salire, così facendo possiamo risparmiare molto tempo, al costo però di un maggiore utilizzo di spazio, dato che ho un Array che deve memorizzare i valori. Si tratta di un compromesso accettabile dato che senza usare Array il tempo di esecuzione sarebbe esponenziale.

1.3.1 Passaggi

A livello pratico dobbiamo:

1. Scomporre il problem in sottoproblemi di dimensione inferiore
2. Formulare la soluzione in maniera ricorsiva - Equazioni di Ricorrenza
3. Usare una strategia bottom-up (non top-down)
4. Memorizzare i risultati in una opportuna struttura dati
5. Individuare il "luogo" che contiene la soluzione del problema (nel caso di Fibonacci l'ultima cella a destra)

DP risulta vantaggiosa quando il numero di chiamate distine è polinomiale (il numero totale di chiamate è esponenziale).

1.4 Osservazioni sui problemi di ottimizzazione

Per ogni istanza del problema esiste un insieme di soluzioni posibili (feasible solutions), più soluzioni perchè le soluzioni ottime possono essere diverse.
Esiste una funzione obiettivo che associa un valore ad ogni soluzione possibile e restituisce come OUTPUT una soluzione possibile (soluzione ottimale) per cui il valore restituito dalla funzione obiettivo è massimo/minimo (valore ottimo).

1.5 LCS - Longest Common Subsequence

Si tratta di un problema che ha come istanza due sequenze di valori e richiede di trovare la più grande sottosequenza comune fra di esse. Si tratta di un problema di ottimizzazione, per questo usare DP è un'ottima idea.

1.5.1 Definizioni di base

Sequenza Successione di elementi topologicamente ordinati, presi da un insieme $\sum$.
Per esempio $X =< 2, 4, 10, 5, 9, 11 >$, più in generale:

- $X =< x_1, x_2, ..., x_m > \rightarrow$ sequenza di $m = |X|$ elementi

Prefisso di lunghezza i Primi i elementi della sequenza:

- $X =< x_1, x_2, ..., x_i > \rightarrow$ prefisso di lunghezza i di X

Dato $X =< 2, 4, 10, 5, 9, 11 >$ per esempio $X_3 =< 2, 4, 10 >$.

i-esimo elemento Indichiamo con $X[i]$ l'i-esimo elemento x_i della sequenza X.

Sottosequenza Una qualsiasi successione di elementi (anche non consecutivi) di una sequenza che però rispettino l'ordine sulla sequenza.
Per esempio data una sequenza $X =< 2, 4, 10, 5, 9, 11 >$

- $Z =< 4, 5, 9 >$ è una sottosequenza di X
- $Z =<>= \epsilon$ è una sottosequenza di X
- $< 9, 5, 4 >$ NON è una sottosequenza di X

Definizione formale di sottosequenza Data $X =< x_1, x_2, ..., x_m >$, una sequenza $Z =< z_1, z_2, ..., z_k >$ $(k \leq m)$ è sottosequenza di X se esiste una successione di k indici interi $i_1 < i_2 < .. < i_k$ tali che $X[i_j] = z_j$ per j compreso tra 1 e k.

Esempio sottosequenza Dato $X =< 2, 4, 10, 5, 9, 11 >$, $Z =< 4, 5, 9 >$ è una sottosequenza di X.

Sottosequenza comune di X e Y è una sottosequenza si di X che di Y.

$$X =< 1, 13, 5, 3, 1, 12, 8, 11, 6, 10, 10 >$$

$$Y =< 1, 5, 5, 2, 3, 1, 12, 8, 8, 10 >$$

$$S =< 5, 3, 1, 8, 10 >$$

S è sottosequenza comune di X e Y.

LCS è la più lunga sottosequenza comune Z di X e Y.

Esempio di LCS

$$X =< 2, 10, 5, 3, 1, 12, 8, 30, 11, 6, 10, 13 >$$

$$Y =< 2, 5, 10, 2, 3, 1, 30, 12, 6, 8, 10 >$$

$< 2, 10, 3, 1, 12, 8, 10 >$ è LCS di X e Y

La LCS è una soluzione ottimale, mentre la sua lunghezza (7) è il valore ottimo.

1.5.2 Istanza del problema

P: date due sequenze $X =< x_1, x_2, ..., x_n > eY =< y_1, y_2, ..., y_n >$, trovare la più lunga sottosequenza comune Z di X e Y.
Abbiamo capito che P è un problema di ottimizzazione di massimo, dove:

- (m,n) → è la dimensione del problema (lunghezza stringhe)
- Soluzioni possibili → tutte le sottosequenze comuni di X e Y
- Funzione obiettivo → lunghezza
- $|Z|$ è il valore ottimo del problema
- Z è una soluzione ottimale

1.6 Procedura LCS

Indichiamo con LCS(A,B) la LCS delle sequenze A e B e di conseguenza $|LCS(A, B)|$ la lunghezza della LCS di A e B.
Procediamo con le seguenti fasi:

1. Individuiamo i sottoproblemi
2. Troviamo le equazioni di ricorrenza
3. Applichiamo una strategia bottom-up con memorizzazione dei risultati

Nota Bene Si deve individuare la sottostruttura ottima del problema. La strategia bottom-up trova l'ottimo (lunghezza di LCS) e in seguito si deve ricostruire una soluzione ottimale (una delle LCS).

1.6.1 Definizione dei sottoproblemi

Sottoproblema di dimensione (i,j). Trovare la LCS dei prefissi X_i e $Y_j \rightarrow LCS(X_i, Y_j)$.

$$i \in \{0, 1, ..., m\}$$

$$j \in \{0, 1, ..., n\}$$

Numero totale sottoproblemi: $(m + 1) \times (n + 1)$
Ricordiamo che $LCS(X_m, Y_n)$ è la soluzione del problema principale.

1.6.2 Equazioni di ricorrenza

Casi base

Tutti i sottoproblemi di dimensione (i,j) tale per cui $i = 0$ oppure $j = 0$.

$$\begin{aligned} i = 0 &\implies LCS(X_0, Y_j) = LCS(\epsilon, Y_j) = \epsilon \\ j = 0 &\implies LCS(X_i, Y_0) = LCS(X_i, \epsilon) = \epsilon \\ i = 0,\ j = 0 &\implies LCS(X_0, Y_0) = LCS(\epsilon, \epsilon) = \epsilon \end{aligned}$$

Passo ricorsivo

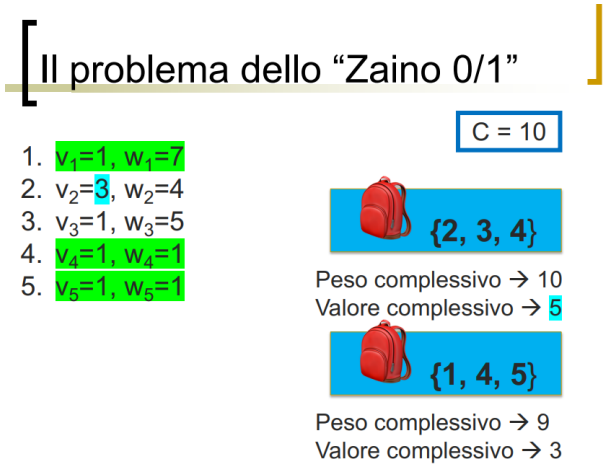
Tutti i sottoproblemi di dimensione (i, j) tale per cui $i > 0$ e $j > 0$.
Introduciamo la sottostruttura ottima del problema:

Capitolo 2

Knapsack Problem 0/1

Questione da risolvere: trovare il subset di oggetti di massimo valore complessivo che non superi la capacità C.

Oggetti Ad ogni oggetto viene associato un peso e un valore, quindi il problema consiste nel inserire nello zaino il massimo valore possibile senza superare il peso massimo.



Capitolo 3

Problema dei cammini minimi - Floyd-Warshall

Come al solito diamo qualche definizione per poter lavorare successivamente in maniera agile.

3.1 Definizioni

3.1.1 Grafo

Un Grafo viene definito come $G = (V, E)$ dove:

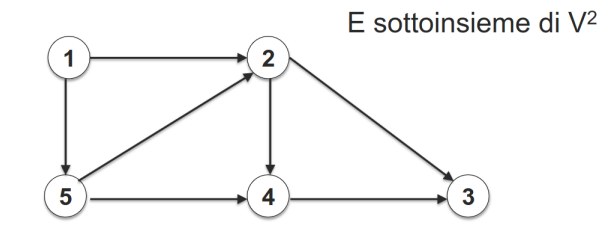
- $V = \{v_1, v_2, v_3, ..., v_n\}$ insieme di vertici
- $E = \{e_1, e_2, e_3, ..., e_m\}$ insieme di archi

Dimensione di G $\rightarrow$ (n,m). Arco $e_k \rightarrow$ relazione R tra due vertici v_i e v_j

R può essere

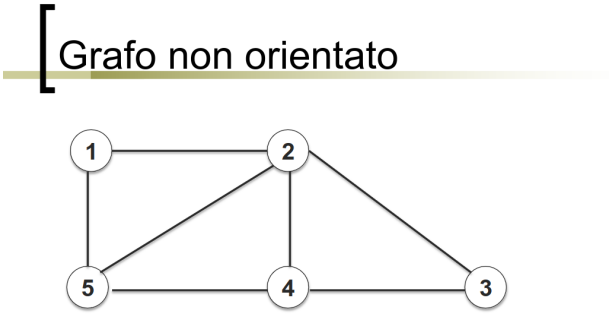
- Simmetrica - Grafo NON Orientato - cioè $v_i \, R \, v_j \Leftrightarrow v_j \, R \, v_i$
- Asimmetrica - Grafo Orientato (o diretto) - cioè $v_i \, R \, v_j \nLeftrightarrow v_j \, R \, v_i$

Un grafo orientato è caratterizzato da un verso di percorrenza degli archi unidirezionale. In questo caso E è sottoinsieme di V^2.



V = {1, 2, 3, 4, 5}

E = {(1,2), (1,5), (2,3), (2,4), (4,3), (5,2), (5,4)}



V = {1, 2, 3, 4, 5}

E = {(1,2), (1,5), (2,3), (2,4), (4,3), (5,2), (5,4)}

3.1.2 Adiacenza

Un vertifica v è adiacente a un vertice u se $(u, v) \in E$.
Per esempio nella rappresentazione del grafo orientato il vertice **1** è adiacante ai vertici **2 e 5**, infatti notiamo che in E è presente $(1, 2), (1, 5)$.

3.1.3 Rappresentazione di un grafo

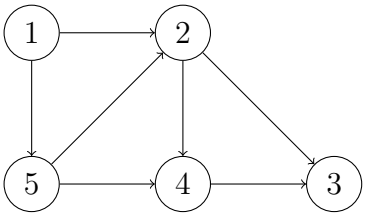
Abbiamo 2 rappresentazioni possibili:

- Liste di adiacenza
- Matrice di adiacenza

1. Le liste di adiacenza utilizzano un vettore L_v di dimensione $|V|$ tale che $V[i]$ è la lista degli adiacenti del vertice v_i. Ogni vertice del grafo avrà un vettore.

2. La matrice di adiacenza è una Matrice M_v di dimensione $n \times n$ tale che $M[i,j] = 1$ se il vertice j è adiacente del vertice i, altrimenti $M[i,j] = 0$. A differenza delle liste in questo caso ho una sola matrice.

3.1.4 Esempio grafo orientato



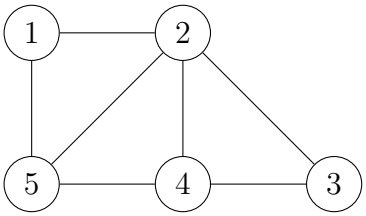
$M =$

	1	2	3	4	5
1	1	0	0	1	0
2	0	0	1	1	0
3	0	0	0	0	0
4	0	1	0	0	0
5	1	0	1	0	0

Dimensione $|V|^2 = n^2$

Numero di celle con 1 $|E|$

3.1.5 Esempio grafo non orientato



$M =$

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Dimensione $|V|^2 = n^2$

Numero di celle con 1 $2|E|$

3.1.6 Liste VS Matrice (memoria)

Liste di adiacenza Sono ottime dal punto di vista dell'occupazione dello spazio nel caso di Grafi sparsi con $|E|$ molto minore di $|V|^2$.

Matrici di adiacenza Risultano migliori nei grafi densi quindi quando ho $|E|$ che si avvicina a $|V|^2$.

3.1.7 Liste VS Matrice (tempo)

(u,v) Intendiamo se i 2 vertici sono collegati. Come tempo intendiamo il tempo per stabilire se (u,v) appartiene ad E e i tempi sono i seguenti:

- Liste di adiacenza $\rightarrow O(|E|) = O(m)$
- Matrice di adiacenza $\rightarrow O(1)$

3.1.8 Cammino in un grafo orientato

Definizione di cammino Sequenza $P =< v_{i_1}, v_{i_2}, ..., v_{i_{k-1}}, v_{i_k} >$ tale che v_{i_k} appartiene a V per $1 \leq j \leq k$ e $(v_{i_j}, v_{i_{j+1}})$ appartiene ad E per $1 \leq j < k$.