Programmazione Dinamica

A.A. 2015-2016

Distanza di edit
Scheduling
Zaino

Scheduling

Lo scheduling e` una ricca sorgente di problemi algoritmici motivati dalla pratica.
Consideriamo il problema in cui le richieste di uso di una risorsa siano specificate da una durata e debbano essere soddisfatte entro un particolare intervallo di tempo.

Usiamo la programmazione dinamica, ma con una “svolta”: l’insieme “ovvio” dei sottoproblemi risultera` non bastare e dovremo creare una collezione di sottoproblemi piu` ricca. La difficolta` sta nel capire qual e` questo buon insieme di sottoproblemi.

Scheduling

Come nel problema degli intervalli pesati si hanno due casi: l'ennesima richiesta non appartiene a una soluzione ottima, allora OPT(n) = OPT(n-1); se invece appartiene a una soluzione ottima, vorremmo una ricorsione semplice che dice qual e` il miglior valore possibile tra le soluzioni che contengono l'ultima richiesta 'n'.

Se la richiesta 'n' viene soddisfatta, il tempo a disposizione per le altre richieste e` il tempo totale (W) meno il tempo della richiesta n-esima (w_n): $W - w_n$.

Questa osservazione suggerisce che servono piu` sottoproblemi: per trovare il valore di OPT(n) non serve solo quello di OPT(n-1) ma si deve anche conoscere la miglior soluzione usando un sottoinsieme dei primi n-1 item e un tempo totale $W - w_n$.

Si ha percio` bisogno di risolvere molti sottoproblemi, uno per ogni insieme {1,…,i} di item e ogni possibile valore per i restanti tempi disponibili w.

$$OPT(i, w) = \max_{S} \sum_{j \in S} w_j$$

Dove il massimo e` sui sottinsiemi S di {1,…,i} che soddisfano $\sum_{j \in S} w_j \leq w$.

Se $w < w_i$ allora $OPT(i, w) = OPT(i-1, w)$,
altrimenti $OPT(i, w) = \max (OPT(i-1, w), w_i + OPT(i-1, w-w_i))$

La soluzione al problema originale e` OPT(n, W).

Nota:
Se n non appartiene alla soluzione, $OPT(n, W) = OPT(n-1, W)$
Se n appartiene alla soluzione, $OPT(n, W) = w_n + OPT(n-1, W - w_n)$

Subset-Sum (n, W)

```
M: array [0..n, 0..W]
for w ← 0 to W M[0, w] = 0
for i ← 1 to n
    for w ← 0 to W
    M[i, w] ← max (M(i-1, w), w_i + M(i-1, w-w_i))
return M[n, W]
```

E` facile dimostrare per induzione su n che il valore di M[n, W] e` il valore ottimo per le richieste 1,…,n e tempo disponibile W.

L'algoritmo *Subset-Sum*(n, W) calcola correttamente il valore ottimo e la sua complessita` in tempo e` O(n·W).

Data una tabella dei valori ottimi dei sottoproblemi, l'insieme ottimo S puo` essere trovato in tempo O(n).

Zaino 0/1 (knapsack problem)

*Dato un contenitore (zaino) di "**capacità**" C ed n oggetti O_1, ..., O_n, ciascuno con un "**peso**" ed un "**valore**", scegliere un sottinsieme degli oggetti, da mettere nel contenitore, in modo da massimizzare il "valore" del contenuto.*

Capacità dello zaino, valore e peso degli oggetti sono supposti dati da numeri naturali.

Questo problema è una formulazione alternativa di quello precedente: anche in questo caso per individuare i sottoproblemi dobbiamo considerare non solo gli oggetti, ma anche come si modifica la capacità dello zaino.

Definizione induttiva

- Base. Se non ci sono oggetti o lo zaino ha capacità 0, il massimo valore è 0:

$$Val(0, k) = Val(i, 0) = 0.$$

- Passo i. Se l'i-esimo oggetto non sta nello zaino:

$$Val(i, k) = Val(i-1, k)$$

Se l'i-esimo oggetto sta nello zaino Val(i, k) sarà dato dal massimo tra:

- massimo valore trasportabile con uno zaino di capacità k senza includere l'i-esimo oggetto
- massimo valore trasportabile con uno zaino di capacità k includendo l'i-esimo oggetto e riempiendo in modo ottimale lo zaino di capacità k - p_i

Usando p_i e v_i per denotare rispettivamente il peso e il valore dell' i-esimo oggetto si ottiene:

$$Val(i,k) = \begin{cases} 0 & \text{se } i = 0 \text{ or } k = 0 \\ Val(i-1,k) & \text{se } i, k \neq 0 \text{ and } k < p_i \\ \max \{Val(i-1,k), Val(i-1, k - p_i) + v_i\} & \text{se } i, k \neq 0 \text{ and } k \geq p_i \end{cases}$$

Algoritmo di programmazione dinamica

```
zaino_0-1 (C, P[], V[], n)
    for  i ← 0 to n    MP[i,0] ← 0
    for  k ← 0 to C    MP[0,k] ← 0
    for i ← 1 to n
       for  k ← 1 to C
            if  (k ≥ P[i]  and MP[i-1, k] < MP[i-1, k-P[i]] + V[i])
                    MP[i, k] ← MP[i-1, k-P[i]] + V[i]
            else  MP[i, k] ← MP[i-1, k]
    return MP[n, C]
```

P e V sono array che memorizzano, rispettivamente, pesi e valori degli oggetti.

MP: matrice n+1 x C+1.

Output: MP [n,C] = Val(n,C) = massimo valore trasportabile

Zaino: 0/1

Esempio: $n = 4$ $v_1 = 4$ $v_2 = 7$ $v_3 = 3$ $v_4 = 5$

$C = 8$ $p_1 = 3$ $p_2 = 5$ $p_3 = 2$ $p_4 = 4$

i ↓ k →	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4	4
2	0	0	0	4	4	7	7	7	11
3	0	0	3	4	4	7	7	10	11
4	0	0	3	4	5	7	8	10	11

NB: qiui ci sono due possibili scelte

NB: le frecce rosse indicano da dove provengono i valori più alti

La definizione data funziona ?

È immediato verificare la correttezza dell'algoritmo rispetto alla definizione ricorsiva che abbiamo dato.

Resta però da verificare che quella definizione è corretta, nel senso che "risolve il problema".
Tale correttezza si basa su una proprietà essenziale:

"lo zaino di dimensione k, usando l'i-esimo oggetto, è riempito nel modo ottimo se è riempito nel modo ottimo lo zaino di dimensione k - p_i usando gli oggetti 1, 2,…, i-1"

La complessità dell'algoritmo è $\Theta(n \cdot C)$.

Zaino con ripetizione

Se sono permesse ripetizioni?

Si deve modificare il concetto di sottoproblema: non serve l'informazione sugli oggetti usati.
Il problema si semplifica da un certo punto di vista, ma il numero di scelte aumenta! Non sono più soltanto due.

Esempio:

oggetto	peso	valore
1	6	30
2	3	14
3	4	16
4	2	9

Se la capacità dello zaino è C = 10, la scelta ottima è prendere un oggetto 1 e due oggetti 4 (totale 48).

Zaino con ripetizione

Come esprimere il problema in funzione di sottoproblemi, quali sono i sottoproblemi?

I sottoproblemi dipendono solo dalla capacità dello zaino.

Denotiamo con Val(k) il massimo valore raggiungibile con uno zaino di capacità k; n sia il numero di oggetti diversi.

$$Val(k) = \begin{cases} 0 & \text{se } k = 0 \\ \max \{Val(k-1), \max \{Val(k - p_i) + v_i \mid p_i \leq k\}\} & \text{se } k \neq 0 \\ & 1 \leq i \leq n \end{cases}$$

Convenzione: il massimo su un insieme vuoto è 0.

Zaino con ripetizione

$Val(0) = 0$

$Val(k) = \max \{Val(k-1), \max \{Val(k - p_i) + v_i \mid p_i \leq k\} \}$

```
zaino_r (C, P[], V[], n)
    Val[0] ← 0
    for k ← 1 to C
        m ← Val[k-1]
        for i ← 1 to n
            if (k ≥ P[i] and Val[k – P[i]] + V[i] > m)
                m ← Val[k – P[i]] + V[i]
        Val[k] ← m
    return Val[C]
```

Complessità dell'algoritmo: $\Theta(n \cdot C)$.

Zaino con ripetizione: esempio

Esempio:

oggetto	peso	valore
1	3	7
2	2	4
3	4	5

capacità dello zaino: 7

k	**val**	ins. 1	ins 2	ins 3	**I**
1	0	-	-	-	0
2	4	-	4		2
3	7	7	4	-	1
4	4+4	7	4+4	5	2
5	7+4	7+4	7+4	5	2
6	7+7	7+7	4+4+4	4+5	1
7	4+4+7	4+4+7	7+4+4	4+4+5	1

Il valore massimo trasportabile è di 15.

Zaino con ripetizione: trovare la soluzione

Esercizio: scrivere la funzione per estrarre tutti gli elementi, con la loro molteplicità, che compongo la scelta ottimale.

```
zaino_r (C, P[], V[], n)
    Val[0] ← 0  ;  I[0] ← 0
        for k ← 1 to C
        m ← Val[k-1] ; I[k] ← 0
        for i ← 1 to n
            if (k ≥ P[i] and Val[k – P[i]] + V[i] > m)
                m ← Val[k – P[i]] + V[i] ; I[k] ← i
        Val[k] ← m
    return Val[C], I
```

Zaino oggetti continui

E se gli oggetti fossero continui?

Riempire lo zaino partendo dagli elementi più preziosi → **algoritmo greedy**

```
zaino_c (C, P, V, n)
    peso ← 0  ;  Val ← 0
    " Ordina P e V in modo non crescente
      rispetto a V[i]/P[i]"
    i ← 1
    while  ((i ≤ n)  and  (peso + P[i] ≤ C))
          peso ← peso + P[i]
          Val ← Val  + V[i]
           i ← i + 1
    if ((i ≤ n) and (peso < C))
          Val ← Val + (C - peso) / P[i] * V[i]
    return Val
```