Corso di Perfezionamento

Programmazione Dinamica

Maria Rita Di Berardini[1]

[1]Dipartimento di Matematica e Informatica
Università di Camerino

15 febbraio 2009

Tecniche di Programmazione

Tecniche di progettazione di algoritmi:

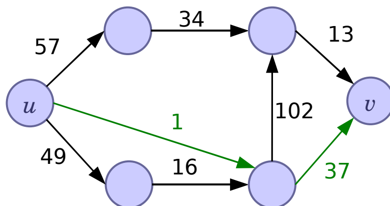
1. Divide et Impera
2. Programmazione Dinamica (PD for short)
3. Algoritmi golosi

Le ultime due tecniche sono più sofisticate rispetto al Divide et Impera, ma consento di risolve in maniera efficiente molti problemi computazionali

Vedremo come queste tecniche di programmazione vengono usate per risolvere **problemi di ottimizzazione**

Problemi di Ottimizzazione: un esempio

Consideriamo il seguente grafo: i nodi rappresentano delle città, gli archi collegamenti tra coppie di città; ogni arco $\langle u, v \rangle$ ha associato un peso che rappresenta la distanza tra u e v



Problema: dati due nodi u e v determinare il cammino minimo tra u e v

Problemi di Ottimizzazione

- Risolvere un problema di ottimizzazione significa determinare nello spazio delle soluzioni ammissibili per un dato problema quella con valore ottimo (massimo o minimo)
- Una soluzione ottima viene costruita effettuando una serie di scelte: ogni scelta determina uno o più sottoproblemi da risolvere
- La programmazione dinamica, come il Divide et Impera, risolve un problema combinando soluzioni di sottoproblemi
- Viene usata anche se i sottoproblemi non sono indipendenti: uno stesso sottoproblema può comparire più volte
- Il concetto chiave: memorizzare le soluzioni dei sottoproblemi per usarle nel caso in cui uno di essi si dovesse ripresentare

Il problema dello zaino 0-1

- Un ladro entra in un magazzino e trova n oggetti
- L'i-esimo oggetto vale v_i euro e pesa w_i Kg, dove v_i e w_i sono dei numeri interi
- Il ladro vuole realizzare il furto di maggior valore compatibile con il peso W del suo zaino
- Ciascun oggetto può essere preso per intero o lasciato; il ladro non può prendere frazioni di oggetti
- Formalmente:

$$\max \sum_{i=1}^{n} x_i \cdot v_i$$

con vincoli

$$\sum_{i=1}^{n} x_i \cdot w_i \leq W$$

$$x_i \in \{0, 1\} \qquad \text{per ogni } i = 1, \ldots, n$$

Il problema dello zaino frazionario

- Un ladro entra in un magazzino e trova n oggetti
- L'i-esimo oggetto vale v_i euro e pesa w_i Kg, dove v_i e w_i sono dei numeri interi
- Il ladro vuole realizzare il furto di maggior valore compatibile con il peso W del suo zaino
- Il ladro può prendere frazioni di oggetti
- Formalmente:

$$\max \sum_{i=1}^{n} x_i \cdot v_i$$

con vincoli

$$\sum_{i=1}^{n} x_i \cdot w_i \leq W$$

$$0 \leq x_i \leq 1 \qquad \text{per ogni } i = 1, \ldots, n$$

Problema dello zaino 0-1

Prima soluzione: la forza bruta

- Consiste nell'enumerate tutte le possibili combinazioni degli n oggetti
- Quante sono?
- Abbiamo due possibili scelte per ogni oggetto (0 o 1), il che significa 2^n combinazioni
- Un algoritmo che si basa su questa scelta ha un costo computazionale $O(2^n)$ – non accettabile

Una soluzione alternativa

- Denotiamo con $Z(n, W)$ il problema originario, che consiste nel realizzare il maggior furto possibile con i primi n oggetti, compatibilmente con il peso W
- Esaminiamo l'n-esimo oggetto e distinguiamo i seguenti casi:

1. $w_n > W$ (non possiamo scegliere questo oggetto): non ci resta che risolvere il sottoproblema $Z(n-1, W)$
2. $w_n \leq W$: prendiamo l'n-esimo oggetto e, poi, risolviamo il sottoproblema $Z(n-1, W-w_n)$
3. $w_n \leq W$: lasciamo l'n-esimo oggetto e, poi, risolviamo il sottoproblema $Z(n-1, W)$

Una soluzione alternativa

- Possiamo risolvere $Z(n, W)$ risolvendo tutti i sottoproblemi della forma

$$Z(i, w) \quad \text{con} \quad i = 0, \dots, n \quad \text{e} \quad w = 0, \dots, W$$

- dove $Z(i, w)$ rappresenta il sottoproblema che consiste nel realizzare il maggior furto possibile con i primi i oggetti, compatibilmente con il peso w
- $Z(0, w)$ non abbiamo altri oggetti da scegliere
- $Z(i, 0)$ non possiamo scegliere altri oggetti perchè la capacità residua dello zaino è zero

Una soluzione alternativa

- Possiamo risolvere $Z(n, W)$ risolvendo tutti i sottoproblemi della forma

$$Z(i, w) \quad \text{con} \quad i = 0, \ldots, n \quad \text{e} \quad w = 0, \ldots, W$$

- dove $Z(i, w)$ rappresenta il sottoproblema che consiste nel realizzare il maggior furto possibile con i primi i oggetti, compatibilmente con il peso w
- $Z(0, w)$ non abbiamo altri oggetti da scegliere
- $Z(i, 0)$ non possiamo scegliere altri oggetti perchè la capacità residua dello zaino è zero
- Questo approccio è corretto?

Proprieta della sottostruttura ottima

Sia $X = \{x_1, \ldots, x_i\}$ una **soluzione ottima** per $Z(i, w)$. Se:

1. $x_i = 0$ — allora X contiene una soluzione $Y = \{x_1, \ldots, x_{i-1}\}$ per il sottoproblema $Z(i-1, w)$. Inoltre:

$$v(X) = \sum_{j=0}^{i} x_j \cdot v_j = \sum_{j=0}^{i-1} x_j \cdot v_j = v(Y)$$

2. $x_i = 1$ — allora X contiene una soluzione $Y = \{x_1, \ldots, x_{i-1}\}$ per il sottoproblema $Z(i-1, w-w_i)$. Inoltre:

$$v(X) = \sum_{j=0}^{i} x_j \cdot v_j = \left(\sum_{j=0}^{i-1} x_j \cdot v_j\right) + v_i = v(Y) + v_i$$

Proprieta della sottostruttura ottima

$Y = \{x_1, \ldots, x_{i-1}\}$ è ottima? Assiamo per assurdo che non lo sia

1. $x_i = 0$

   Esiste una soluzione $Y' = \{x'_1, \ldots, x'_{i-1}\}$ per $Z(i-1, w)$ tale che $V(Y') > v(Y)$.

   Allora $X' = \{x'_1, \ldots, x'_{i-1}, 0\}$ è una soluzione per $Z(i, w)$ e

$$v(X') = v(Y') > v(Y) = V(X)$$

   Assurdo perchè X è ottima

Proprieta della sottostruttura ottima

$Y = \{x_1, \ldots, x_{i-1}\}$ è ottima? Assiamo per assurdo che non lo sia

1. $x_i = 0$
2. $x_i = 1$
   Esiste una soluzione $Y' = \{x'_1, \ldots, x'_{i-1}\}$ per $Z(i-1, w-w_i)$ tale che $V(Y') > v(Y)$.

   Allora $X' = \{x'_1, \ldots, x'_{i-1}, 1\}$ è una soluzione per $Z(i, w)$ e

   $$v(X') = v(Y') + v_i > v(Y) + v_i = V(X)$$

   Assurdo perchè X è ottima

Proprieta della sottostruttura ottima

- In altri termini, una soluzione ottima di $Z(i, w)$ contiene al suo interno soluzioni ottime di sottoproblemi
- Lo Zaino 0-1 soddisfa la proprietà della sottostruttura ottima
- Questa proprietà è fondamentale
- Ci dice che possiamo costruire una soluzione soluzione ottima a partire da soluzioni ottime dei sottoproblemi

Calcolo del valore della soluzione ottima

Cerchiamo di identificare un metodo per il calcolo del valore della soluzione ottima per ciascun sottoproblema $Z(i, w)$

$$V(0, w) = 0 \quad \text{per ogni } w = 0, \ldots, W$$

$$V(i, 0) = 0 \quad \text{per ogni } i = 1, \ldots, n$$

Se $i, w > 0$, allora

$$V(i, w) = \begin{cases} V(i-1, w) & \text{se } w_i > w \\ \max\{V(i-1, w), V(i-1, w-w_i) + v_i\} & \text{se } w_i \leq w \end{cases}$$

Calcolo del valore della soluzione ottima

RecKnapsack01$(\mathbf{w}, \mathbf{v}, W, i)$
  if $i = 0$ or $W = 0$
    then return 0

  if $w[i] > W$
    then return RecKnapsack01$(\mathbf{w}, \mathbf{v}, W, i-1)$

  if $w[i] \leq W$
    then
      $x \leftarrow$ **RecKnapsack01**$(\mathbf{w}, \mathbf{v}, W, i-1)$
      $y \leftarrow$ **RecKnapsack01**$(\mathbf{w}, \mathbf{v}, W - \mathbf{w}[i], i-1)$
      return $\max\{x, y\}$

Calcolo del valore della soluzione ottima

RecKnapsack01$(\mathbf{w}, \mathbf{v}, W, i)$
  if $i = 0$ or $W = 0$
    then return 0

  if $w[i] > W$
    then return RecKnapsack01$(\mathbf{w}, \mathbf{v}, W, i-1)$

  if $w[i] \leq W$
    then
      $x \leftarrow$ **RecKnapsack01**$(\mathbf{w}, \mathbf{v}, W, i-1)$
      $y \leftarrow$ **RecKnapsack01**$(\mathbf{w}, \mathbf{v}, W - \mathbf{w}[i], i-1)$
      return $\max\{x, y\}$

Ha una complessità $O(2^n)$

Identificazione di uno schema Bottom-Up

Possiamo usare una matrice V di dimensione $n+1 \times W+1$ per memorizzare risultati parziali

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

Casi Base: $V[i,0] = V[0,j] = 0$ per ogni $i = 0, \ldots, n$ e $w = 0, \ldots, W$

Identificazione di uno schema Bottom-Up

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0	$\odot$	$\odot$	$\odot$	$\odot$	
3	0				$\odot$	
4	0					

$V[i, w]$ dipende dai valore $V[i-1, w]$e $V[i-1, w-w_i]$

Identificazione di uno schema Bottom-Up

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0	$\odot$	$\odot$	$\odot$	$\odot$	
3	0				$\odot$	
4	0					

$V[i,w]$ dipende dai valore $V[i-1,w]$e $V[i-1,w-w_i]$

Ci basta riempire la matrice riga per riga

Calcolo iterativo del valore di una soluzione ottima

DynamicKnapsack01$(\mathbf{w}, \mathbf{v}, W)$
$n \leftarrow length[\mathbf{v}]$
for $w \leftarrow 0$ **to** W **do** $V[0, w] \leftarrow 0$
for $i \leftarrow 0$ **to** n **do** $V[i, 0] \leftarrow 0$

for $i \leftarrow 1$ **to** n
  do for $w \leftarrow 1$ **to** W
    do if $w \leq \mathbf{w}_i$
      then $V[i, w] \leftarrow \max\{V[i-1, w], V[i-1, w - \mathbf{w}_i] + \mathbf{v}_i\}$
      else $V[i, w] \leftarrow V[i-1, w]$
return V

Calcolo iterativo del valore di una soluzione ottima

DynamicKnapsack01$(\mathbf{w}, \mathbf{v}, W)$
$n \leftarrow length[\mathbf{v}]$
for $w \leftarrow 0$ **to** W **do** $V[0, w] \leftarrow 0$
for $i \leftarrow 0$ **to** n **do** $V[i, 0] \leftarrow 0$

for $i \leftarrow 1$ **to** n
  do for $w \leftarrow 1$ **to** W
    do if $w \leq \mathbf{w}_i$
      then $V[i, w] \leftarrow \max\{V[i-1, w], V[i-1, w - \mathbf{w}_i] + \mathbf{v}_i\}$
      else $V[i, w] \leftarrow V[i-1, w]$
return V

Ha una complessità $O(nW)$

Un esempio

$\mathbf{w} = [2, 3, 4, 5]$, $\mathbf{v} = [3, 4, 5, 6]$ e $W = 5$

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

Un esempio

$\mathbf{w} = [2, 3, 4, 5]$, $\mathbf{v} = [3, 4, 5, 6]$ e $W = 5$

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

Un esempio

$\mathbf{w} = [2, 3, 4, 5]$, $\mathbf{v} = [3, 4, 5, 6]$ e $W = 5$

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

Un esempio

$\mathbf{w} = [2, 3, 4, 5]$, $\mathbf{v} = [3, 4, 5, 6]$ e $W = 5$

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

Un esempio

$\mathbf{w} = [2, 3, 4, 5]$, $\mathbf{v} = [3, 4, 5, 6]$ e $W = 5$

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Costruzione di una soluzione ottima

FindKnapsackItems$(V, \mathbf{w}, \mathbf{v}, W)$

$i \leftarrow length[\mathbf{v}]$
$k \leftarrow W$
while $i > 0$ **and** $k > 0$
  do if $V[i, k] \neq V[i-1, k]$
    then "l'oggetto i è nello zaino"
      $k \leftarrow k - \mathbf{w}_i$
  $i \leftarrow i - 1$

Un esempio

$\mathbf{w} = [2, 3, 4, 5]$, $\mathbf{v} = [3, 4, 5, 6]$ e $W = 5$

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

- $i = 4$ e $k = W = 5$: $V[4, 5] = V[3, 5]$, decrementa i
- $i = 3$ e $k = 5$: $V[3, 5] = V[2, 5]$, decrementa i
- $i = 2$ e $k = 5$: $V[2, 5] \neq V[1, 5]$, inserisce il secondo oggetto nello zaino, decrementa i e $k = k - \mathbf{w}_2 = 5 - 3 = 2$
- $i = 1$ e $k = 2$: $V[1, 2] \neq V[0, 2]$, inserisce il primo oggetto nello zaino, decrementa i e $k = k - \mathbf{w}_1 = 2 - 2 = 0$

Outline

1. Il problema dello zaino
2. Zaino 0-1
   - Sottostruttura ottima
   - Metodo ricorsivo per il calcolo della soluzione ottima
   - Schema Bottom-Up
   - Costruzione della soluzione ottima

Il processo di sviluppo di un algoritmo di PD

È suddiviso in quattro fasi:

1. caratterizzazione della struttura di una soluzione ottima (verifica della proprietà della sottostruttura ottima)
2. definizione di un metodo ricorsivo per il calcolo del valore di una soluzione ottima
3. calcolo del valore di una soluzione ottima secondo uno schema bottom-up (dal basso verso l'alto)
4. costruzione della soluzione ottima dalle informazioni raccolte nella frase precedente