Appunti lezione – Capitolo 14
Greedy

Alberto Montresor

19 Novembre, 2015

1 Domanda: dimostrare che $S[i,j] = \emptyset$ con $i \geq j$

Nel problema della selezione delle attività, il sottoinsieme $S[i,j]$ è definito nel modo seguente:

$$S[i,j] = \{\ k \in S \mid b_i \leq a_k < b_k \leq b_j\ \}$$

Se i valori b sono ordinati per tempi crescenti, qual è il contenuto di $S[i,j]$ con $i \geq j$?

Supponiamo per assurdo che esista $k \in S[i,j]$: allora è possibile dimostrare che $b_i < b_j$; infatti:

$$b_i \leq a_k < b_k \leq a_j < b_j$$

Ma poiché i valori b sono ordinati, abbiamo che $b_i \geq b_j$, il che è assurdo.

2 Domanda: Soluzione ottima per "Insieme indipendente di intervalli" contiene primo elemento?

- Dimostriamo innanzitutto la prima parte:
  - sia $S[i,j]$ il problema considerato;
  - sia $A[i,j]$ una soluzione ottima per $S[i,j]$;
  - sia m l'attività di $S[i,j]$ che termina per prima; ovviamente, $i < m < j$;
  - sia $m' \in A[i,j]$ l'attività che termina per prima in $A[i,j]$.

  Se $m' = m$, allora il teorema è banalmente dimostrato. Supponiamo quindi che sia $m' \neq m$.

  Consideriamo la soluzione:

  $$A'[i,j] = A[i,j] - \{m'\} \cup \{m\}$$

  E' possibile affermare che:

  - Tutte le attività in $A'[i,j]$ sono mutualmente compatibili; infatti m' era l'attività che termina per prima in $A[i,j]$; $b_m < b_{m'} \leq b_k$, per qualsiasi attività k in $A[i,j]$, quindi aggiungere l'attività m dopo aver tolto l'attività m' non provoca nessun conflitto
  - $A'[i,j]$ è ottimale perchè $|A[i,j]| = |A'[i,j]|$.

  Quindi $A'[i,j]$ è una soluzione ottimale di $S[i,j]$ che contiene l'attività di $S[i,j]$ che termina per prima.

- Per quanto riguarda la seconda parte; poiché m è la prima attività a terminare, nessun'altra attività può terminare prima dell'inizio di m. Quindi l'insieme $S[i,m]$ è vuoto.

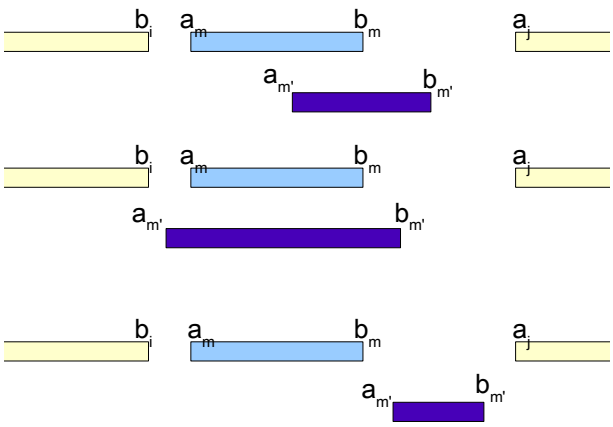


Figura 1: Tre possibile configurazioni delle attività m e m'; in tutti i tre casi, $b_m < b'_m$.

3 Domanda: Descrivere un algoritmo goloso per il problema del resto

Sottostruttura ottima

Sia $t[1 \dots k]$ le pezzature che abbiamo a disposizione, e sia $S(n)$ il problema che vogliamo risolvere (corrispondente a dare il resto per la cifra n). Supponiamo che il *multi-insieme* $A(n)$ sia una soluzione ottima di $S(n)$ e contenga il pezzo $t[j]$; allora $S(n - t[j])$ è un sottoproblema di $S(n)$, la cui soluzione ottima è data da $A(n) - \{t[j]\}$. Se così non fosse, esisterebbe una soluzione B di $S(n - t[j])$ tale che $|B| < |A(n) - \{t[j]\}|$, allora $B \cup \{t[j]\}$ sarebbe una soluzione del problema $S(n)$, e $|B \cup \{t[j]\}| < |A(n)|$.

Definizione ricorsiva

Sia $D[0 \dots n]$ la tabella dei costi utilizzata per risolvere il problema in maniera bottom-up; $D[m]$ contiene il minimo numero di monete per risolvere il problema $S(m)$, con $0 \leq m \leq n$.
La tabella può essere calcolata ricorsivamente in questo modo:

$$D[m] = \begin{cases} 0 & \text{se } m = 0 \\ \min\limits_{1 \leq j \leq k} \{D[m - t[j]] + 1 : t[j] \leq m\} & \text{se } m > 0 \end{cases}$$

In altre parole, utilizzando le soluzioni che abbiamo trovato per i problemi più piccoli di $S(m)$, individuiamo il pezzo che minimizza la soluzione del problema $S(m)$. Ovviamente, il caso base è zero: per dare un resto zero non servono monete.

Algoritmo basato su programmazione dinamica

E' possibile risolvere il problema in maniera bottom-up con un algoritmo di questo tipo:

```
resto(integer[] t, integer k, integer n, integer[] C, integer[] S)
C[0] ← 0
for m ← 1 to n do
    C[m] ← +∞
    for j ← 1 to k do
        if m > t[j] and C[m − t[j]] + 1 < C[m] then
            C[m] ← C[m − t[j]] + 1
            S[m] ← j
```

dove C è il vettore dei costi, S è il vettore delle scelte. Il costo di questo algoritmo è $\Theta(kn)$.
La soluzione finale può venire stampata dalla procedura ricorsiva seguente:

```
stampa-resto(integer[] t, integer n, integer[] S)
while n > 0 do
    print t[S[n]]
    n ← n − t[S[n]]
```

Scelta greedy

E' naturale pensare che la scelta ingorda sia quella di scegliere sempre il pezzo più alto a disposizione. Bisogna però dimostrare che il pezzo più alto faccia sempre parte di una soluzione ottima. Una soluzione potrebbe essere la seguente:

```
resto(integer[] t, integer k, integer n, integer[] A)
for i ← 1 to k do
    A[i] ← ⌊n/t[i]⌋
    n ← n − A[i] · t[i]
```

La soluzione assume che $t[1] > t[2] > \cdots > t[k]$ siano i tagli delle monete; n è il resto da dare; $A[1 \dots k]$ contiene il numero di monete per ogni taglio. Il costo di questo algoritmo è $O(k)$.
Consideriamo quindi l'insieme di monete: $t[1] = 50c$, $t[2] = 10c$, $t[3] = 5c$, $t[1] = 1c$, e dimostriamo la correttezza dell'algoritmo.
Sia A una qualunque soluzione ottima; quindi $\sum_{i=1}^{4} A[i] \cdot t[i] = n$ e $\sum_{i=1}^{4} A[i]$ è minimo.
Sia $n_k = \sum_{i=k+1}^{4} A[i]t[i]$ la somma delle monete di taglio inferiore a $t[k]$.
Se riusciamo a dimostrare che $n_k < t[k]$ per ogni k, allora la soluzione è proprio quella calcolata dall'algoritmo, che quindi restituisce una soluzione ottima.

- $n_4 = 0 < 1 = t[4]$.
- $n_3 = 1 \cdot A[4] < 5 = t[3]$.
  Poiché A è ottima, $A[4] < 5$ (altrimenti si potrebbe sostituire cinque monete da $1c$ con una da $5c$, migliorando la soluzione).
- $n_2 = 5 \cdot A[3] + n_3 \leq 5 + n_3 < 5 + 5 = 10 = t[2]$. Sappiamo che $A[3] \leq 1$ (altrimenti si potrebbe sostituire due monete da $5c$ con una da $10c$, migliorando la soluzione). Dalla disequazione precedente ricaviamo invece $n_3 < 5$.

- $n_1 = 10 \cdot A[2] + n_2 \leq 40 + n_2 < 40 + 10 = 50 = t[1]$.
  Sappiamo che $A[2] \leq 4$ (altrimenti si potrebbe sostituire cinque monete da $10c$ con una da $50c$, migliorando la soluzione). Dalla disequazione precedente, abbiamo anche che $n_2 < 10$.

Questo completa la dimostrazione.
Si noti che una dimostrazione del genere (ma bisogna gestire caso per caso) funziona anche con $1c$, $2c$, $5c$, $10c$, $20c$, $50c$, $100c$, $200c$.

4 Domanda – Monete potenze consecutive

E' possibile applicare la tecnica precedente a questo insieme. La dimostrazione è simile a quella appena illustrata. Una dimostrazione più semplice è pensare che le monete permettono di esprimere qualunque numero in base c.

5 Domanda – Esempio di insieme di monete che non supporta la scelta greedy

Insieme: 12 8 1
Valore: 17

Scelta greedy: 12+1+1+1+1+1
Soluzione ottima: 8+8+1

6 Domanda – Scelta ingorda di SJF (Shortest Job First)

Siano $p_1, p_2, \ldots, p_n$ n job che devono essere eseguiti su una singola macchina, in maniera sequenziale. Ogni job p_i ha una durata $t[i]$. Con il termine "tempo di completamento", intendiamo il tempo necessario per completare il job, inclusa la sua durata e tutto il tempo di attesa prima di essere eseguito. Il tempo di completamento medio è calcolato su tutti i job.
Scrivere un'algoritmo greedy che individui una sequenza di esecuzione il cui tempo di completamento medio sia minimo.

Scelta ingorda

Come possibile scelta, scegliamo il job con durata minore. L'idea è abbastanza intuitiva: mettendo il job più corto per primo, "peserà" di meno nella sommatoria.

Proprietà della scelta ingorda

Dimostriamo che esiste sempre una soluzione ottima che contiene la scelta ingorda, ovvero in cui il job con durata più corta sia scelto per primo.
Supponiamo di avere una soluzione ottima A, rappresentata da una permutazione $i_1, i_2, \ldots, i_n$ degli indici $1, 2, \ldots, n$. Supponiamo che il job in posizione i_1 non sia quello di durata inferiore (altrimenti abbiamo terminato). Il job di durata inferiore è in posizione i_k.
Definiamo con $T_A(h)$ il tempo di completamento dell'h-esimo job della permutazione A:

$$T_A(h) = \sum_{r=1}^{h} t[i_r]$$

Il tempo di completamento totale T_A della permutazione A è dato da:

$$T_A = \sum_{h=1}^{n} T_A(h)$$

Costruiamo una soluzione A', scambiando l'elemento 1 e k della permutazione.

- Tutti i job eseguiti dopo il job k hanno lo stesso tempo di completamento in A e A', perché lo scambio non influisce su di loro:
$$\forall h > k : T_{A'}(h) = T_A(h)$$
- Tutti i job compresi fra 1 e k, estremi esclusi, hanno un tempo di completamento inferiore in A', perchè lo scambio ha ridotto il tempo di completamento per il primo job.
$$\forall h, 1 < h < k : T_{A'}(h) \leq T_A(h)$$
- Il primo e il k-esimo job della permutazione A sono stati scambiati di posto; poichè la durata del k-esimo job è più breve di quella del primo, se ne deduce che quando viene spostato al primo posto, ha un tempo di completamente inferiore; d'altra parte, il tempo di completamento del job in k-esima posizione in entrambe le sequenze è uguale, in quanto comunque devono essere completati tutti i primi k job.
$$\begin{aligned} T_{A'}(1) &\leq T_A(1) \\ T_{A'}(k) &= T_A(k) \end{aligned}$$

Quindi il tempo di completamento totale di A' è inferiore o uguale al tempo di completamento di A; ma poichè A è ottima, questo significa che A e A' hanno lo stesso tempo di completamento.

Sottostruttura ottima

Semplice da verificare per la proprietà *cut-and-paste*.

7 Domanda Problema dello zaino frazionario

Dato un insieme di n oggetti caratterizzati da profitti $p[1], \ldots, p[n]$ e volumi interi $v[1], \ldots, v[n]$, trovare il sottoinsieme di oggetti di valore massimo con peso inferiore o uguale ad una costante C. E' possibile prendere frazioni di oggetti.

Sottostruttura ottima E' possibile dimostrare che il problema ha sottostruttura ottima nello stesso modo in cui è stato provato per lo zaino 0-1. Lasciato per esercizio.

Scelta greedy Come scelta greedy, calcoliamo il "valore specifico" di tutti gli oggetti: $p[i]/v[i]$. Ad ogni passo, prendiamo tutta la quantità possibile dell'oggetto con maggiore valore specifico. Se esauriamo la capacità, abbiamo finito. Altrimenti passiamo all'oggetto successivo.
Dobbiamo però dimostrare che la nostra scelta gode della proprietà greedy.
Supponiamo che l'oggetto con maggiore valore specifico sia l'oggetto 1. Supponiamo esista una soluzione ottima O, che contiene una quantità $x[1]$ dell'oggetto 1 inferiore alla capacità dello zaino e inferiore al peso totale dell'oggetto 1: $x[1] < C$ e $x[1] < v[1]$.
In altre parole, per riempire lo zaino è stato scelto una quantità $\min\{C - x[1], v[1] - x[1]$ di altri oggetti il cui valore specifico è inferiore/uguale a quello dell'oggetto 1. Quindi, se sostituiamo questa quantità con

un'ulteriore frazione dell'oggetto 1, otteniamo una soluzione che massimizza il consumo dell'oggetto 1 e ha valore maggiore o uguale al valore di una soluzione ottima; quindi è ottima anch'essa.
Ecco una versione più compatta rispetto a quella presentata nel libro:

```
zaino(real[] p, real[] v, real C, integer n, real[] x)
  { ordina p e v in modo che p[1]/v[1] ≥ p[2]/v[2] ≥ ··· ≥ p[n]/v[n]}
  integer i ← 1
  while i ≤ n and C > 0 do
      x[i] ← min(C/v[i], 1)
      C ← C − x[i] · v[i]
      i ← i + 1
  for j ← i to n do
      x[j] ← 0
```

8 Scelta ingorda dell'algoritmo di Huffman

Completiamo la dimostrazione dei lucidi con la dimostrazione che il costo degli alberi T' e T'' è inferiore al costo dell'albero T.

$$\begin{aligned} C(T) - C(T') &= \sum_{c\in\Sigma} f[c]d_T(c) - \sum_{c\in\Sigma} f[c]d_{T'}(c) \\ &= (f[x]d_T(x) + f[a]d_T(a)) - (f[x]d_{T'}(x) + f[a]d_{T'}(a)) \\ &= (f[x]d_T(x) + f[a]d_T(a)) - (f[x]d_T(a) + f[a]d_T(x)) \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \\ \geq 0 \end{aligned}$$

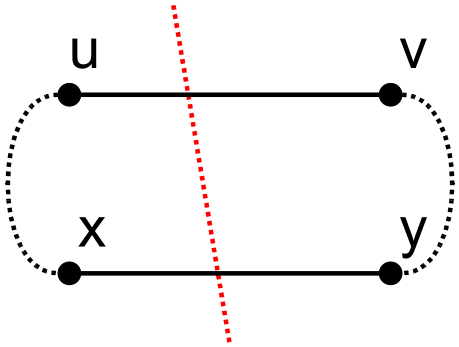
La stessa cosa si può replicare per T'' e T'.

9 Minimo albero di copertura

Teorema 1. Sia $G = (V, E)$ un grafo non orientato e connesso. Sia w una funzione peso a valori reali definita su E. Sia $A \subseteq E$ contenuto in un qualche albero di copertura minimo per G. Sia $(S, V - S)$ un qualunque taglio che rispetta A. Sia (u, v) un arco leggero che attraversa il taglio. Allora l'arco (u, v) è sicuro per A.

Dimostrazione. sia T un albero di copertura minimo che contiene A. Ci sono due casi: l'arco (u, v) appartiene a T, e quindi è sicuro per A. Oppure non appartiene a T, nel qual caso faremo vedere che sostituendo un arco in T con il nuovo arco (u, v) otteniamo un albero T' che è sempre un albero di copertura minimo.
Si considerino i nodi u, v. Questi sono connessi con un qualche cammino sull'albero T (per definizione di albero).
Poiché per ipotesi u, v si trovano in lati opposti di un taglio, esiste almeno un arco (x, y) in T che attraversa il taglio (ne possono esistere anche più di uno).
Consideriamo l'albero $T' = T - \{(x, y)\} \cup \{(u, v)\}$. Con l'eliminazione di (x, y) l'albero si spezza in due componenti che si riuniscono con (u, v).
Dimostriamo che T' é un albero di copertura minimo. Poiché (x, y) attraversa il taglio $(S, V - S)$, abbiamo che $w(u, v) \leq w(x, y)$ ((u, v) è leggero). Quindi:

$$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$$



Ma poiché T é un albero di copertura minimo, $w(T) \leq w(T')$ e quindi il peso dei due archi è uguale. E' facile quindi vedere che $A \cup \{(u, v)\}$ è contenuto in T'; questo perché (x, y) non appartiene ad A. □

Corollario 1. Sia $G = (V, E)$ un grafo non orientato e connesso; sia w una funzione peso a valori reali definita su E; sia $A \subseteq E$ contenuto in un qualche albero di copertura minimo per G; sia C una componente connessa (un albero) nella foresta $G_A = (V, A)$; sia (u, v) un arco leggero che connette C a qualche altra componente in G_A. Allora l'arco (u, v) è sicuro per A.

Dimostrazione. Definiamo un taglio $(C, V - C)$; per definizione, il taglio rispetta A. Poiché (u, v) attraversa il taglio, per il teorema precedente (u, v) è sicuro per A. □