Tecniche Algoritmiche: tecnica greedy (o golosa)

EDUMETER

- Valutate la didattica!
- http://www.unito.it/edumeter

Algoritmi Greedy

Idea: per trovare una soluzione **globalmente ottima**, scegli ripetutamente soluzioni **ottime localmente**

Un esempio

Input: lista di interi che rappresentano il valore di monete disponibili e un intero che rappresenta un resto.
Output: una collezione **di cardinalità minima** di monete la cui somma sia uguale al resto.

Input: monete disponibili: 25, 10, 5 e 1;
resto da formare: 87
Output: 25+25+25+10+1+1

Osservazione:
gli algoritmi greedy non sempre funzionano !

Input: monete disponibili: 1,4 e 6
resto da formare: 9
Greedy output: 6 + 1 + 1 + 1
Output ottimale: 4 + 4 + 1

Le scelte ottime a breve non garantiscono di raggiungere una soluzione ottima finale.

Algoritmi Greedy - Schema generale 1

Le appetibilità degli oggetti sono note fin dall'inizio e non vengono modificate

Greedy1 ($\{a_1, a_2, \ldots a_n\}$)

$S \leftarrow$ insieme vuoto

ordina gli a_i in ordine **non crescente di appetibilità**

for ogni a_i nell'ordine **do**

if a_i può essere aggiunto a S

then $S \leftarrow S \cup \{a_i\}$

return S

A. Horvath Alg&Lab 07/08 (da F. Damiani - Alg. & Sper. 06/07 e M. Zacchi - Alg. & Lab. 03/04)

Una applicazione classica per lo schema 1

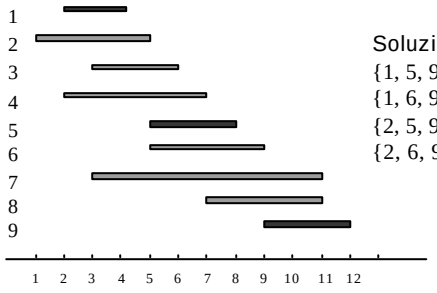
Il problema della selezione di attività

Input: $S = \{1, 2, \ldots, n\}$ insieme di attività che devono usare una risorsa in modo esclusivo. Ogni attività i è caratterizzata da un tempo di inizio e da un tempo di fine: $[s_i, f_i)$ $(s_i \leq f_i)$.

Output: insieme che contiene il massimo numero di attività mutuamente compatibili.

Oggetti: le attività
Appetibilità: tempo di fine

A. Horvath Alg&Lab 07/08 (da F. Damiani - Alg. & Sper. 06/07 e M. Zacchi - Alg. & Lab. 03/04)



A. Horvath Alg&Lab 07/08 (da F. Damiani - Alg. & Sper. 06/07 e M. Zacchi - Alg. & Lab. 03/04)

Greedy-Activity-Selector (n, s, f)

A ← insieme vuoto
"ordina le attività in ordine non decrescente rispetto ai tempi di fine"
A ← {1}
j ← 1
for i ← 2 **to** n **do**
  if $s_i \geq f_j$
    then A ← A È {i}
         j ← i
return A

Complessità: **O(n lgn)**

Ricordiamo che: gli algoritmi greedy non sempre funzionano !

Domanda. L'algoritmo ***Greedy-Activity-Selector*** seleziona sempre un numero massimo di attività?

Greedy-Activity-Selector (n, s, f)

A ← insieme vuoto
"ordina le attività in ordine non decrescente rispetto ai tempi di fine"
A ← {1}
j ← 1
{A e` una soluzione ottima rispetto alle attività esaminati}
for i ← 2 **to** n **do**
  if $s_i \geq f_j$
    then A ← A È {i}
         j ← i
return A
{A e` una soluzione ottima}

Invariante dettagliato durante il ciclo:
esiste un E, insieme massimo di attività compatibili in {1,…,n}, tale che:

- A è un sottoinsieme di E
- l'appetibilità degli elementi di E-A è non superiore a quella degli elementi di A
- j è l'ultima attività aggiunta ad A

Dopo il ciclo:
A e` un insieme massimo di attività compatibili

L'invariante viene reso vero dall'inizializzazione:

- Siano $<i_1, i_2, \ldots, i_k>$ le attività (in ordine crescente di fine attività) di un insieme massimo di attività compatibili E.
- Poichè $f_{1} \leq f_{i_1} \leq s_{i_2}$ anche $<1, i_2, \ldots, i_k>$ è un insieme massimo di attività compatibili.
- L'appetibilita' degli elementi di E-{1} è non superiore a quella degli elementi di {1}, e j = 1 è l'ultima attività scelta.

Supponiamo che

- $A=<1, h_2, \ldots, h_r>$ sia un sottoinsieme di $E=<1, h_2, \ldots, h_r, i_{r+1}, \ldots, i_k>$ (in ordine crescente di fine attività),
- sia $j = h_r$ l'ultima attività scelta.

Allora

- $f_{i_{r+1}} \leq s_{i_{r+2}}$
- se i è l'attività selezionata nel corpo del while, $f_i \leq f_{i_{r+1}}$

Perciò

- $<1, h_2, \ldots, h_r, i>$ è un sottoinsieme di un insieme massimo di attività compatibili: $<1, h_2, \ldots, h_r, i, i_{r+2}, \ldots, i_k>$
- l'appetibilita' degli elementi di E-{1, $h_2, \ldots, h_r$, i} e' non superiore a quella degli elementi di {1, $h_2, \ldots, h_r$, i}
- j = i è l'ultima attività scelta.

Algoritmi Greedy - Schema generale 2

Se le appetibilità degli oggetti possono essere modificate dalle scelte già fatte.

Greedy2 ($\{a_1, a_2, \ldots a_n\}$)
S ← insieme vuoto
valuta le appetibilità degli a_i
while ci sono elementi da scegliere **do**
scegli l'a_i più **appetibile**
if a_i può essere aggiunto a S
then $S \leftarrow S \cup \{a_i\}$
aggiorna le appetibilità degli a_i
return S

Una applicazione classica per lo schema 2

Codici di Huffmann (tecnica efficiente per la compressione dei dati)

Esempio:
- Si consideri un alfabeto di 6 caratteri.
- In un codice a lunghezza fissa servono 3 bit per la loro rappresentazione.
- Quindi un file di dati di 100.000 caratteri richiede 300.000 bit.

Una applicazione classica per lo schema 2

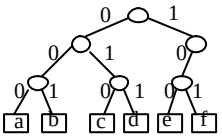
Un codice a lunghezza variabile
- codifica i simboli che occorrono più di frequente con meno bit
- può permettere un risparmio anche del 25%.

Per esempio:
- caratteri: a,b,c,d,e,f
- frequenza: 0.45,0.13,0.12,0.16,0.09,0.05
- codice l. fissa: 001,010,011,100,101
- codice l. variabile: 0,101,100,111,1101,1100

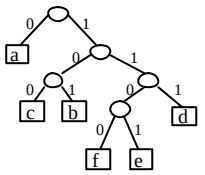
Codici prefissi:
nessuna parola del codice è prefisso di un'altra.

Con codici prefissi è facile la decodifica in quanto non ci sono ambiguità.

Un codice binario può essere rappresentato in modo efficiente con un albero binario in cui le foglie rappresentano i caratteri.



Un codice "ottimo":
- permette la codifica più possibile compressa
- può essere rappresentato con un albero binario in cui
  - ogni nodo interno ha sempre due figli e
  - i caratteri più frequenti compaiono ai livelli alti dell'albero



•Se C è l'alfabeto, l'albero di un codice prefisso ottimo per C ha $|C| - 1$ nodi interni.

• A partire dall'albero è facile calcolare il numero di bit necessari a codificare un file.
• Indichiamo
  • con $f(c)$ la frequenza del carattere "c" nel file
  • con $d_T(c)$ il livello della foglia etichettata "c" nell'albero T.
• Il numero di bit e' allora:

$$B(T) = \sum_{c \in C} f(c)\, d_T(c)$$

L'algoritmo di Huffman

Input: un insieme C di caratteri e una funzione f che indica la frequenza di ogni carattere in un testo.
Output: un codice binario ottimo per la compressione del testo.

L'algoritmo di Huffman

• considerare i caratteri in ordine non decrescente di frequenza
• per ogni carattere creare un albero formato solo da una foglia
• a partire dai due alberi che hanno frequenza minore costruire un nuovo albero che ha come frequenza la somma delle frequenze degli alberi fusi
• ripetere la fusione finché si ottiene un unico albero

Si può individuare in tale descrizione una strategia greedy:

Oggetti: gli alberi
Appetibilità: frequenza

Poiché gli oggetti vengono modificati e con essi le loro appetibilità, applichiamo lo schema greedy, con qualche modifica per l'eventuale aggiornamento delle appetibilità.

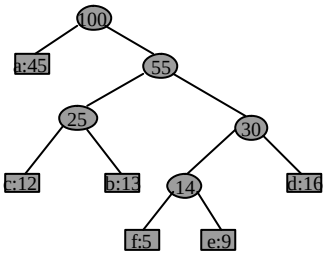
Huffman ($\{c_1, c_2, \ldots c_n\}$)
valuta le appetibilità dei c_i

while l'albero non è unico do

scegli i due alberi più **appetibili**
crea l'albero fusione dei due
aggiorna le appetibilità degli alberi

return l'albero ottenuto

Esempio:
- caratteri: a,b,c,d,e,f
- frequenza: 0.45,0.13,0.12,0.16,0.09,0.05



Implementazione

Consideriamo i seguenti oggetti e metodi dati:
- C: un insieme di |C| caratteri
  - ogni carattere c e' inizialmente rappresentato come una coppia (carattere, frequenza)
  - f[c] denota la frequenza di c

Implementazione

- **Allocate-Node():** un funzione che restituisce un nuovo nodo di una struttura ad albero
  - le cui foglie conterranno coppie (carattere, frequenza) e
  - i cui nodi interni conterranno la somma delle frequenze contenute nelle foglie del sottoalbero corrispondente
  - f[z] denota la frequenza contenuta nel nodo (foglia o nodo interno) z,
  - left[z] e right[z] denotano il figlio sinistro e destro di z rispettivamente

Implementazione

- Q è una coda con priorità che
  - memorizza (radici di) alberi la cui priorità è data dal valore di f
  - sarà utilizzata per identificare i due alberi con la frequenza più piccola, da fondere assieme

Implementazione

```
Huffman (C, f)
  n ← |C|                          {per ottenere un unico albero
  Q ← C                             sono necessarie n-1 fusioni}
  for i ← 1 to n - 1 do
    z ← Allocate-Node ()
    x ← left [z] ← Extract-min (Q)   {due alberi più appetibili e crea
    y ← right[z] ← Extract-min (Q)    l'albero fusione}
    f[z] ← f[x] + f[y]
    Insert (Q, z)                    {aggiorna le appetibilità
  return Extract-min (Q)              degli alberi}
                                     {restituisci l'albero ottenuto}
```


Implementazione

Complessita' in tempo:

O(n lgn) se la coda di priorità è implementata con un heap binario.

$O(n^2)$ se la coda fosse realizzata con una struttura con inserzione e/o estrazione del minimo lineari

Conclusione

Tecnica greedy: varie applicazioni:

- problema del sequenziamento
- distributore automatico di resto
- calcolo dei codici di Huffman
- calcolo del minimo albero ricoprente (algoritmi di Prim, Kruskal, Boruvka)
- cammini minimi a sorgente singola (algoritmo di Dijkstra)