Tecniche Algoritmiche/3
Algoritmi *greedy*

Moreno Marzolla
http://www.moreno.marzolla.name/

Original work Copyright © Alberto Montresor, Università di Trento, Italy
(http://www.dit.unitn.it/~montreso/asd/index.shtml)
Modifications Copyright © 2010, 2011 Moreno Marzolla, Università di Bologna, Italy
(http://www.moreno.marzolla.name/teaching/ASD2010/)

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/2.5/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Introduzione

- La programmazione dinamica
  - In maniera bottom-up, valuta tutte le decisioni possibili
  - Evitando però di ripetere sotto-problemi (decisioni) già percorse
- Un algoritmo greedy (ingordo, goloso)
  - Seleziona una sola delle possibili decisioni...
  - ... quella che sembra ottima (ovvero, è localmente ottima)
  - Sperabilmente, si ottiene così un ottimo globale

Introduzione

- Quando applicare la tecnica greedy?
  - Quando è possibile *dimostrare* che esiste una *scelta ingorda*
    - “Fra le molte scelte possibili, ne può essere facilmente individuata una che porta sicuramente alla soluzione ottima”
  - Quando il problema ha sottostruttura ottima
    - “Fatta tale scelta, resta un sottoproblema con la stessa struttura del problema principale”
- Non tutti i problemi hanno una scelta ingorda
- Nota:
  - in alcuni casi, soluzioni non ottime possono essere comunque interessanti

Algoritmo greedy generico

```
algoritmo paradigmaGreedy(insieme di candidati C) → soluzione
   S := ∅
   while ( (not ottimo(S)) and (C ≠ ∅) ) do
      x := seleziona(C)
      C := C - {x}
      if (ammissibile(S ∪ {x})) then
         S := S ∪ {x}
      endif
   endwhile
   if (ottimo(S)) then
      return S
   else
      errore ottimo non trovato
   endif
```

Ritorna *true* sse la soluzione S è ottima

Estrae un candidato dall'insieme C

Ritorna *true* sse la soluzione candidata è una soluzione ammissibile (anche se non necessariamente ottima)

Problema del resto

Problema del resto

- Input
  - Un numero intero positivo n
- Output
  - Il più piccolo numero intero di pezzi per dare un resto di n centesimi utilizzando monete da 50c, 20c, 10c, 5c, 2c e 1c.
- Esempi
  - n = 78, 5 pezzi: 50+20+5+2+1
  - n = 18, 4 pezzi: 10+5+2+1

Algoritmo greedy per il resto

- Insieme dei candidati C
  - Insieme finito delle monete a disposizione nel serbatoio
- Soluzione S
  - Insieme delle monete da restituire
- `ottimo(S)`
  - true sse la somma dei valori di S è esattamente uguale al resto
- `ammissibile(S)`
  - true sse la somma dei valori di S è minore o guale al resto
- `seleziona(C)`
  - scegli la moneta di valore massimo in C

Algoritmo greedy per il resto

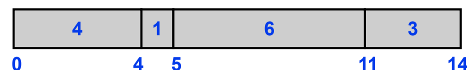
- L'algoritmo greedy sceglie tra le monete disponibili quella di valore massimo che risulti minore o uguale al residuo da erogare
- L'algoritmo fatto in questo modo potrebbe *non* fornire la soluzione ottima in base al sistema monetario in uso
  - Esempio: monete da 1, 3, 4
  - Dobbiamo erogare un resto di 6
  - L'algoritmo greedy produce 4, 1, 1
  - La soluzione che minimizza il numero di pezzi sarebbe 3, 3
  - L'algoritmo greedy potrebbe NON trovare la soluzione; ad esemio, se il serbatoio contiene 4, 3, 3 l'algoritmo greedy sceglie dapprima la moneta 4, e poi non puo' proseguire

Problema di scheduling (Shortest Job First)

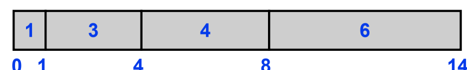
Algoritmo di scheduling—Shortest Job First

4
1
6
3

- Definizione:
  - 1 processore, n job p_1, p_2, ..., p_n
  - Ogni job p_i ha un tempo di esecuzione t[i]
  - Minimizzare il tempo medio di completamento
    - Il libro considera il problema di minimizzare il tempo medio di attesa



(4+5+11+14)/4 = 34/4



(1+4+8+14)/4 = 27/4

Algoritmo greedy di scheduling

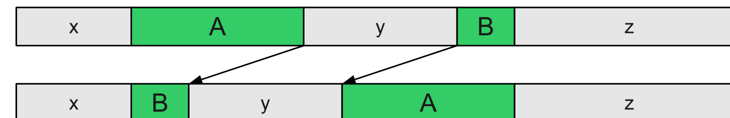
- Siano p_1, p_2, … p_n gli n job che devono essere schedulati
- L'algoritmo greedy esegue n passi
  - ad ogni passo schedula il job più breve tra quelli che rimangono

Algoritmo greedy per lo scheduling Shortest-job-first

- Insieme dei candidati C
  - Job da schedulare, inizialmente { p_1, p_2, … p_n }
- Soluzione S
  - Ordine dei job da schedulare: p_{i1}, p_{i2}, … p_{in}
- `ottimo(S)`
  - True sse l'insieme S contiene tutti i job
- `ammissibile(S)`
  - Restituisce sempre true
- `seleziona(C)`
  - Sceglie il job di durata <u>minima</u> in C

Dimostrazione di ottimalità

- Consideriamo un ordinamento dei job in cui un job "lungo" A viene schedulato prima di uno "corto" B
  - x, y e z sono sequenze di altri job



- Osserviamo:
  - Il tempo di completamento dei job in x non cambia
  - Il tempo di completamento dei job in z non cambia
  - Il tempo di completamento di A nella seconda soluzione è uguale al tempo di completamento di B nella prima soluzione
  - Il tempo di completamento di B <u>si riduce</u>
  - Il tempo di completamento dei job in y <u>si riduce</u>

Problema della compressione (alberi di Huffman)

Problema della compressione

- Rappresentare i dati in modo efficiente
  - Impiegare il numero minore di bit per la rappresentazione
  - Goal: risparmio spazio su disco e tempo di trasferimento
- Una possibile tecnica di compressione: *codifica di caratteri*
  - Tramite *funzione di codifica f: f(c) = x*
    - *c* è un possibile carattere preso da un alfabeto Σ
    - *x* è una rappresentazione binaria
    - *"c è rappresentato da x"*

Codici di Huffman

- Supponiamo di avere un file di *n* caratteri

– caratteri:	**a**	**b**	**c**	**d**	**e**	**f**
– frequenze:	**45%**	**13%**	**12%**	**16%**	**9%**	**5%**

- Codifica tramite ASCII (8 bit per carattere)
  - Dimensione totale: 8n bit
- Codifica basata sull'alfabeto (3 bit per carattere)
  - Codifica: **000 001 010 011 100 101**
  - Dimensione totale: 3n bit
- Possiamo fare di meglio?

Codici di Huffman

- Codifica a lunghezza variabile

– Caratteri:	**a**	**b**	**c**	**d**	**e**	**f**
– Codifica:	**0**	**101**	**100**	**111**	**1101**	**1100**

  - Costo totale: (0.45*1+0.13*3+0.12*3+0.16*3+0.09*4+0.05*4)*n=2.24n
- Codice "a prefisso" (meglio, "senza prefissi"):
  - Nessun codice è prefisso di un altro codice
  - Condizione necessaria per permettere la decodifica
- Esempio: addaabca
  - 0·111·111·0·0·101·100·0

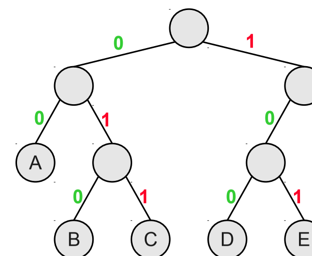
f(a) = 1, f(b)=10, f(c)=101
Come decodificare 101?
come "c" o "ba"?

Codici di Huffman

- Alcune domande
  - È possibile che il testo codificato sia più lungo della rappresentazione con 3 bit?
  - Esistono testi "difficili" per una rappresentazione di questo tipo?
  - Come organizzare un algoritmo per la decodifica?
- Come organizzare un algoritmo per la codifica è il tema dei lucidi seguenti
- Cenni storici
  - David Huffman, "A method for the construction of Minimum-Redundancy Codes", 1952
  - Algoritmo ottimo per costruire codici prefissi

Rappresentazione ad albero per la decodifica

- Rappresentazione come alberi binari
  - Figlio sinistro: 0 Figlio destro: 1
  - Caratteri dell'alfabeto sulle foglie



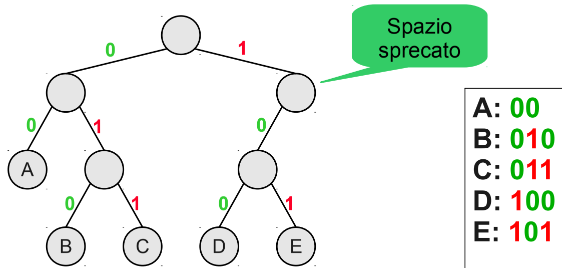
Algoritmo di decodifica:

1. parti dalla radice
2. leggi un bit alla volta percorrendo l'albero:
   0: sinistra
   1: destra
3. stampa il carattere della foglia
4. torna a 1

A: 00
B: 010
C: 011
D: 100
E: 101

Rappresentazione ad albero per la decodifica

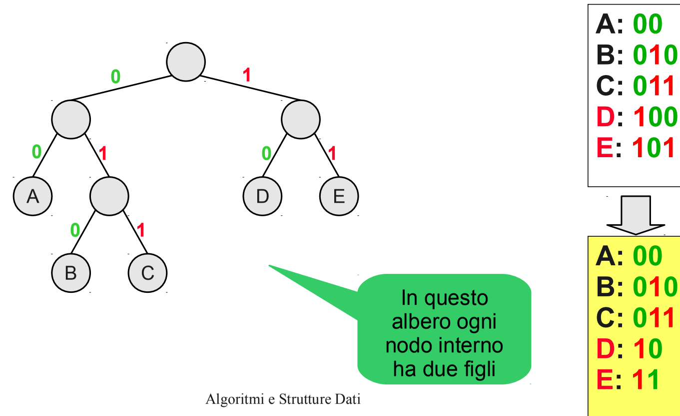
- Non c'è motivo di avere un nodo interno con un solo figlio



A: 00
B: 010
C: 011
D: 100
E: 101

Rappresentazione ad albero per la decodifica

- Non c'è motivo di avere un nodo interno con un solo figlio



A: 00
B: 010
C: 011
D: 100
E: 101

A: 00
B: 010
C: 011
D: 10
E: 11

Definizione formale del problema

- Definizione: codice ottimo
  - Dato un file *F*, un codice *C* è ottimo per *F* se non esiste un altro codice tramite il quale *F* possa essere compresso impiegando un numero inferiore di bit.
- Nota:
  - Il codice ottimo dipende dal particolare file
  - Possono esistere più soluzioni ottime

Definizione formale del problema

- Supponiamo di avere:
  - un file F composto da caratteri nell'alfabeto Σ
  - un albero *T* che rappresenta la codifica
- Quanti bit sono richiesti per codificare il file?
  - Per ogni $c \in \Sigma$, sia $d_T(c)$ la *profondità* della foglia che rappresenta *c*
  - Il codice per *c* richiederà allora $d_T(c)$ bit
- Se *f[c]* è il numero di volte in cui *c* compare in F, allora la dimensione della codifica è

$$C(f,T) = \sum_{c \in \Sigma} f[c] d_T(c)$$

Algoritmo di Huffman

- Principio del codice di Huffman
  - Minimizzare la lunghezza dei caratteri che compaiono più frequentemente
  - Assegnare ai caratteri con la frequenza minore i codici corrispondenti ai percorsi più lunghi all'interno dell'albero
- Un codice è progettato per un file specifico
  - Si ottiene la frequenza di tutti i caratteri
  - Si costruisce il codice
  - Si rappresenta il file tramite il codice
  - Si aggiunge al file una rappresentazione del codice

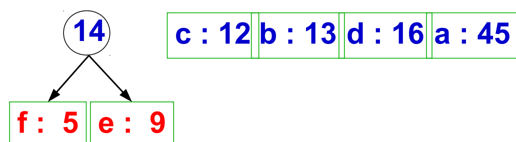
Costruzione del codice

- Passo 1: Costruire una lista ordinata di nodi foglia per ogni carattere, etichettato con la propria frequenza

f : 5	e : 9	c : 12	b : 13	d : 16	a : 45

Costruzione del codice

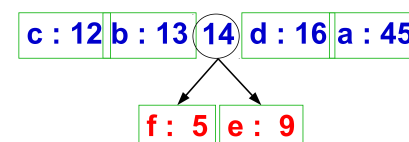
- Passo 2: Rimuovere i due nodi con frequenze minori
- Passo 3: Collegarli ad un nodo padre etichettato con la frequenza combinata (sommata)



c : 12	b : 13	d : 16	a : 45

Costruzione del codice

- Passo 4: Aggiungere il nodo combinato alla lista.



Costruzione del codice

- Ripetere i passi 2-4 fino a quando non resta un solo nodo nella lista

14 d : 16 25 a : 45

f : 5 e : 9 c : 12 b : 13

Costruzione del codice

- Ripetere i passi 2-4 fino a quando non resta un solo nodo nella lista

25 30 a : 45

c : 12 b : 13 14 d : 16

f : 5 e : 9

Costruzione del codice

- Ripetere i passi 2-4 fino a quando non resta un solo nodo nella lista

a : 45 55

25 30

c : 12 b : 13 14 d : 16

f : 5 e : 9

Costruzione del codice

- Al termine si etichettano gli archi dell'albero con bit 0-1

100 0 1

a 55 0 1

25 0 1 30 0 1

c b 14 0 1 d

f e

Algoritmo in pseudo-codice

```
Algoritmo Huffman(f[1..n], c[1..n]) → Tree
    Q := new MinPriorityQueue()
    for i := 1 to n do
        z := new TreeNode(f[i], c[i]);
        Q.insert(z, f[i]);
    endfor
    for i := 1 to n - 1 do
        z1 := Q.findMin(); Q.deleteMin();
        z2 := Q.findMin(); Q.deleteMin();
        z := new TreeNode(z1.f + z2.f, '');
        z.left := z1;
        z.right := z2;
        Q.insert(z,z1+f+z2.f);
    endfor
    return Q.findMin();
```

Costo computazionale: $\Theta(n \log n)$

TreeNode:

f // frequenza (key)
c // carattere
left // figlio sinistro
right // figlio destro

Dimostrazione di correttezza

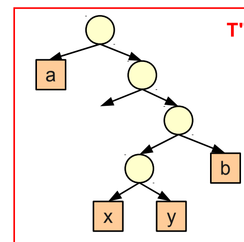
- Teorema: L'output dell'algoritmo Huffman per un dato file è un codice a prefisso ottimo
- Schema della dimostrazione:
  - Proprietà della scelta greedy
    - "Scegliere i due elementi con la frequenza più bassa conduce sempre ad una soluzione ottimale"
  - Sottostruttura ottima
    - "Dato un problema sull'alfabeto Σ, è possibile costruire un sottoproblema con un alfabeto più piccolo"
- (Gli interessati trovano la dimostrazione nelle slide seguenti)

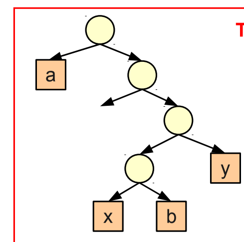
Scelta greedy

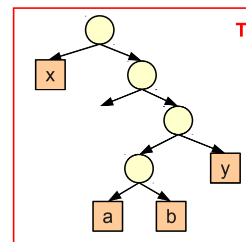
- Sia
  - Σ un alfabeto, f un array di frequenze
  - x, y i due caratteri che hanno frequenza più bassa
- Allora
  - Esiste un codice prefisso ottimo per Σ in cui *x,y* hanno *la stessa profondità massima* e i loro codici *differiscono solo per l'ultimo bit*
- Dimostrazione
  - Al solito, basata sulla trasformazione di una soluzione ottima
  - Supponiamo che esistano due caratteri *a,b* con profondità massima e questi siano diversi da x,y

Scelta greedy

- Assumiamo (senza perdere in generalità):
  - $f[x] \le f[y]$ $f[a] \le f[b]$
- Poiché le frequenze di x e y sono minime:
  - $f[x] \le f[a]$ $f[y] \le f[b]$
- Scambiamo x con a: otteniamo T'
- Scambiamo y con b: otteniamo T"







Scelta greedy

- È possibile verificare che:
  - $C(f,T'') \leq C(f, T') \leq C(f,T)$
- Ma poiché T è ottimo, sappiamo anche che:
  - $C(f,T) \leq C(f,T'')$
- Quindi T'' è anch'esso ottimo

Ricordiamo: C(f,T) = dimensione della codifica

Sottostruttura ottima

- Sia
  - Σ un alfabeto, f un array di frequenze
  - x, y i due caratteri che hanno frequenza più bassa
  - $\Sigma' = \Sigma - \{ x, y \} \cup \{ z \}$
  - $f[z] = f[x] + f[y]$
  - O un albero che rappresenta *un codice a prefisso ottimo per Σ'*
- Allora:
  - *L'albero T (ottenuto da O sostituendo il nodo foglia z con un nodo interno con due figli x,y) rappresenta un codice a prefisso ottimo per Σ*

Sottostruttura ottima

- Esprimiamo la relazione fra il costo di T e O
  - Per ogni $c \in \Sigma - \{ x,y \} \rightarrow f[c] \cdot d_T(c) = f[c] \cdot d_O(c)$
    Quindi tutte queste componenti sono uguali
  - $d_T(x) = d_T(y) = d_O(z)+1$
- Quindi
  - $f[x] \cdot d_T(x) + f[y] \cdot d_T(y) =$
    $(f[x] + f[y])(d_O(z)+1) =$
    $f[z] \cdot d_O(z) + f[x] + f[y]$
- Da cui concludiamo
  - $C(f, T) = C(f, O) + f[x] + f[y]$
    $C(f, O) = C(f, T) - f[x] - f[y]$

Sottostruttura ottima

- Per assurdo: supponiamo T non sia ottimo
  - Allora esiste un albero T' tale che $C(f, T') < C(f, T)$
  - Senza perdere in generalità, sappiamo che T' ha x e y come foglie sorelle
  - Sia T'' l'albero ottenuto da T' sostituendo il padre di x,y con un nodo z tale che $f[z] = f[x] + f[y]$
- Allora
  - $C(f, T'') = C(f, T') - f[x] - f[y]$
    $< C(f, T) - f[x] - f[y]$
    $= C(f, O)$
  - Il che è assurdo, visto che O è ottimo

Algoritmi greedy

- Vantaggi
  - Semplici da programmare
  - Solitamente efficienti
  - Quando è possibile dimostrare la proprietà di scelta greedy, danno la soluzione ottima
  - La soluzione sub-ottima può essere accettabile
- Svantaggi
  - Non sempre applicabili se si vuole la soluzione ottima