

Algoritmi Greedy non applicati su grafi.



• Scheduling di processi:

In questo caso abbiamo un problema di ottimizzazione.

$0, 1, \dots, n-1 \rightarrow$ processi

$t_0, t_1, \dots, t_{n-1} \rightarrow$ tempo esecuzione di un processo

Per eseguire k processi uno di seguito all'altro avremo:

$$T_{j,k} = \sum_{s=0}^k T_{j,s} \quad // \text{ogni processo deve aspettare l'esecuzione di quello prima}$$

Per ottenere il tempo di attesa medio si usa:

$$A = \frac{1}{n} \sum_{i=0}^{n-1} T_i$$

Esempio

// prima p_0 poi p_1

	p_0	p_1	
t	7	3	

$\leadsto (p_0, p_1)$ $T_{j0} = 7$ $T_{j1} = T_{j0} + 3 = 10$
// solo se stesso

$$A = \frac{1}{2} (7 + 10) = \frac{17}{2} = 8,5$$

prima p_1 poi p_0

$\leadsto (p_1, p_0)$ $T_{j1} = 3$ $T_{j0} = T_{j1} + 3 = 10$

$$A = \frac{1}{2} (3 + 10) = \frac{13}{2} = 6,5$$

Come puoi notare, per ottenere la A minima dobbiamo ordinare i processi con i tempi in ordine crescente. In questo modo abbiamo l'algoritmo greedy.

Se non fosse greedy

$(J_0, J_1, \dots, J_k, J_{k+1}, \dots, J_{n-1})$ con $t_{J_k} > t_{J_{k+1}}$

Questi due tempi non sono in ordine

Importante per il teorema di scambio

$$A = \frac{1}{n} (T_{J_0}, T_{J_1}, \dots, T_{J_k}, T_{J_{k+1}}, \dots, T_{J_{n-1}})$$

Argomento di scambio

Scambio J_k con J_{k+1} , in modo tale da avere gli argomenti in posizione crescente, e quindi la soluzione ottimale.

Questa operazione (lo scambio) posso farla per qualunque scheduling non greedy. Se invece è greedy lo scambio non ha senso, perché peggioreresti la situazione.

$$A' = \frac{1}{n} (T'_{J_0}, T'_{J_1}, \dots, T'_{J_{k+1}}, T'_{J_k}, \dots, T'_{J_{n-1}})$$

Quindi $A - A' > 0 \rightsquigarrow$ ovvero $A > A'$

puoi semplificare $T_{J_0}, T_{J_1}, T_{J_{n-1}}$ etc con quelli di A'

$$\frac{1}{n} (T_{J_0}, T_{J_1}, \dots, T_{J_k}, T_{J_{k+1}}, \dots, T_{J_{n-1}}) > \frac{1}{n} (T'_{J_0}, T'_{J_1}, \dots, T'_{J_{k+1}}, T'_{J_k}, \dots, T'_{J_{n-1}})$$

$$\text{rimane } T_{J_k} + T_{J_{k+1}} - T'_{J_{k+1}} - T'_{J_k} > 0$$

ma $T_{j,k+1}$ e $T'_{j,k}$ sono semplificabili perché

M sono tutti i t prima di $t_{j,k}$

$$\hookrightarrow T_{j,k} = M + t_{j,k} + t_{j,k+1}$$

$$T'_{j,k+1} = M + t_{j,k+1} + t_{j,k}$$

da notare le t piccole, ovvero i tempi di esecuzione

Quindi il risultato finale è $T_{j,k} - T_{j,k+1} > 0$, ovvero $t_{j,k} - t_{j,k+1} > 0$

• Scheduling di processi con pesi

$0, 1, \dots, n-1$ processi

t_0, t_1, \dots, t_{n-1} tempi di esecuzione

c_0, c_1, \dots, c_{n-1} costo di esecuzione

T_i sarà il Tempo di completamento

Invece C sarà il costo, ovvero il tempo in attesa di tutti i processi.

Quindi abbiamo

$$C = \sum_{i=0}^{n-1} c_i \cdot T_i$$

Quindi adesso non mi conviene eseguire i più costosi prima.

Es: Costi uguali, si possono non considerare ed usare l'approccio visto prima.

p_0 p_1

t 6 3

c 4 4

$$(1, 0) = C = 4 \cdot 3 + 4 \cdot 6 = 48$$

Costi diversi ma tempi uguali. (Mi conviene fare prima i più costosi)

	p_0	p_1	
t	2	2	$(0,1) = C = (4 \cdot 2) + (2 \cdot 4) = 16$
C	4	2	$(1,0) = C' = (2 \cdot 2) + (4 \cdot 4) = 20$

Se avessimo preso i costi in ordine crescente ci sarebbe costato di più!

Tempi e Costi diversi

Devo avere un bilanciamento tra tempo e costo, in particolare avere il tempo che cresce ed il costo che decresce.

Per farlo basta fare una divisione, $\frac{t}{c}$

	p_0	p_1	
t	6	3	$p_0 \frac{6}{5} \quad p_1 \frac{3}{2}$
C	5	2	$(0,1) = C = 5 \cdot 6 + 2 \cdot 9 = 48$ //ordine crescente

Algoritmo dello zaino frazionario

È un rilassamento dell'algoritmo dello zaino, che studierai più avanti.

Descrizione del Problema:

- Hai uno zaino che può contenere un volume massimo C
- Hai n oggetti, ciascuno con un volume vol_i e un valore val_i
- L'obiettivo è massimizzare il valore totale degli oggetti nello zaino senza

Superare il volume massimo C .

All'algoritmo conviene fare $\frac{val}{vol}$, per ottenere il prezzo unitario.

L'algoritmo può prendere frazioni di materiale.

- inizializzazione:
 - per ogni materiale i , $dose(i) = 0$;
 - ordinare i valori $val_0/vol_0, \dots, val_{n-1}/vol_{n-1}$ (ad es. MaxHeap in graphLib)
 - spazioRimasto = capacità
- iterativamente, finché spazioRimasto > 0 e heap non vuoto
 - scegliere il materiale m di valore unitario massimo:
 - se spazioRimasto $\geq vol_m$, prenderlo tutto, cioè
 $dose(m) = 1$ e $quant(m) = vol_m$
 $valTot = valTot + val_m$
 - altrimenti, prenderne la massima frazione possibile,
 $dose(m) = spazioRimasto/vol_m$ e $quant(m) = spazioRimasto$;
 $valTot = valTot + val_m * dose(m)$
oppure $valTot = valTot + val_m * quant(m)/vol_m$
 - aggiornare spazioRimasto

Dimostrazione

L'algoritmo produce un array di questo tipo:

$S_g = (1, 1, 1, 1, d, 0, 0, 0, 0, 0)$ ← dove ogni cella dell'array è la dose che è stata prelevata dall'algoritmo di quel materiale.
↳ Soluzione Greedy
I materiali sono ordinati per $\frac{val}{vol}$

$S' = (1, 1, 1, 1, 1, 1, d, 0, 0, 0)$ ← Questa soluzione non è ammissibile, perché ha superato la capacità massima dello zaino.

$S'' = (1, 1, d, 0, 0, 0, 0, 0, 0, 0)$ ← Questa soluzione è ammissibile, ma non abbiamo

raggiunto la capacità massima dello zaino, quindi è meno efficiente della soluzione Greedy.

$\bar{S} = (1, 1, 0, 0, 1, 1, 0, 0, 0, 0) \leftarrow$ Questa soluzione è ammissibile, anche se il ladro ha scelto un materiale sbagliato.

$(1, 1, 0, 0, 1, 1, 0, 0, 0, 0)$

$(1, 1, 1, 0, 1, 0, 0, 0, 0, 0)$

↓
Possiamo però utilizzare l'argomento di scambio, posando l'oggetto esistente e prendendone un'altro con valore unitario più alto.

↓

↘
A questo punto non otteniamo la soluzione greedy (quindi questa è anche meno efficiente), ma ne otteniamo una ottimale migliorata.

Ovviamente puoi farlo solo se la soluzione non è greedy.

Perché gli algoritmi greedy non si possono usare sempre?

La programmazione Greedy è un approccio che risolve i problemi prendendo la decisione migliore possibile in ogni fase. Questo significa che l'algoritmo fa la scelta che sembra essere la migliore al momento, senza preoccuparsi delle conseguenze future.

Il concetto di "greedy choice property" si riferisce alla capacità di un problema di essere risolto facendo sempre la scelta che sembra essere la migliore in quel momento.

In altre parole, una soluzione globale ottimale può essere raggiunta facendo scelte localmente ottimali.

La "optimal structure" è una proprietà in cui una soluzione ottimale al problema può essere costruita dalle soluzioni ottimali dei suoi sottoproblemi. Questa proprietà è utilizzata per determinare l'utilità degli algoritmi greedy per un problema.

Per quanto riguarda il tuo esempio, l'algoritmo Greedy può essere utilizzato per trovare l'Albero di Copertura Minimo (MST) perché l'MST ha la proprietà della scelta greedy e della struttura ottimale.

In ogni passo, l'algoritmo Greedy sceglie l'arco con il peso minimo che non crea un ciclo, che è la scelta localmente ottimale.

Questa scelta porta alla soluzione globalmente ottimale, che è l'MST.

D'altra parte, il problema dello zaino 0-1 non può essere risolto con un algoritmo Greedy perché non ha la proprietà della scelta greedy.

In altre parole, una scelta che sembra ottimale in un dato momento (ad esempio, scegliere l'oggetto con il rapporto valore/peso più alto) potrebbe non portare a una soluzione ottimale globale.

Per questo motivo, il problema dello zaino 0-1 è solitamente risolto con tecniche come la programmazione dinamica.

Affinché un algoritmo greedy possa funzionare devono sempre avere la greedy choice property e la optimal structure.