Algoritmi e Strutture di Dati (3ª Ed.)
Scheduling di programmi

Alan Bertossi, Alberto Montresor

La tecnica *greedy* è spesso utile per risolvere problemi di *scheduling*, in cui si hanno dei programmi da eseguire su un processore e si vuole trovarne l'ordine di esecuzione ottimo in base ad un prefissato criterio.

A titolo di esempio, consideriamo il problema dei PROGRAMMI IN RITARDO. Dati n programmi p_1, $\ldots$, p_n, tali che ciascun p_i richiede t_i unità di tempo di esecuzione e deve essere eseguito entro una certa scadenza d_i, trovare un ordine S in cui eseguirli tutti in modo da minimizzare il numero di programmi per i quali la scadenza non è rispettata.

Il problema può essere risolto con un algoritmo *greedy* ideato da Moore (1968). I programmi sono ordinati in una sequenza S per scadenze crescenti. Successivamente, si cerca il primo programma p in ritardo e si elimina il programma p' avente tempo di esecuzione più grande tra quelli che stanno nella sottosequenza iniziale di S che termina con p. Il procedimento è iterato sulla sottosequenza S' ottenuta da S eliminando p', fino ad ottenere una sottosequenza S^* senza programmi in ritardo. La sequenza ottima è ottenuta eseguendo dapprima i programmi della sottosequenza S^*, nell'ordine in cui appaiono in S^*, seguiti dai programmi in ritardo in un ordine qualsiasi.

La scelta *greedy* dell'algoritmo di Moore è basata sul fatto che, se in una sottosequenza deve esserci un programma in ritardo, tanto vale eliminare il programma più lungo per lasciare il maggior tempo disponibile prima delle scadenze dei programmi che seguiranno quello eliminato. La dimostrazione si basa sul *principio di sostituzione*: trasformiamo una qualunque soluzione ottima nella soluzione generata dall'algoritmo *greedy*, senza inficiarne l'ottimalità.

Nel nostro caso, data una sequenza ottima S, essa può essere sempre trasformata nella sequenza ottenuta con le scelte *greedy*, dove i programmi in orario (cioè non in ritardo) sono ordinati in una sottosequenza iniziale per scadenze crescenti, mentre tutti i programmi in ritardo seguono quelli in orario. Infatti, se in S un programma in ritardo precede uno in orario, allora, spostando il programma in ritardo all'ultimo posto di S, il numero di programmi in ritardo non aumenta. Pertanto tutti i programmi in orario possono precedere quelli in ritardo. Consideriamo nella sottosequenza iniziale dei programmi in orario la minima scadenza d_j tale che p_j è preceduto da p_i con $d_i > d_j$ (se ci sono più p_i, allora sono considerati uno alla volta per d_i crescenti). Togliendo p_i e spostandolo immediatamente dopo p_j, i tempi di terminazione dei programmi che seguivano p_i fino a p_j (incluso) diminuiscono, mentre quelli di p_i e dei programmi che lo seguono restano immutati, e quindi il numero dei programmi in ritardo non cambia. Ripetendo il procedimento considerando sempre la minima scadenza d_j, tutti i programmi in orario possono essere ordinati per scadenze crescenti.

Esempio 1 (Programmi in ritardo)**.** La Fig. 1 mostra l'esecuzione dell'algoritmo di Moore per i programmi p_1, $\ldots$, p_4 tali che: $t_1 = 4$, $d_1 = 5$, $t_2 = 3$, $d_2 = 8$, $t_3 = 3$, $d_3 = 9$, $t_4 = 6$, $d_4 = 12$. La sequenza S iniziale è $p_1, \ldots, p_4$. Essendo p_3 il primo programma in ritardo, si elimina p_1, perché $t_1 = 5$ è massimo tra t_1, t_2 e t_3. Dopo aver eliminato p_1, nessun altro programma oltre a p_1 è in ritardo, e la sequenza ottima è p_2, p_3, p_4, p_1. Si noti che l'eliminazione di p_2 o p_3 al posto di p_1 provocherebbe il ritardo di due programmi: quello eliminato e p_4. □

Una realizzazione poco accorta dell'algoritmo porta ad una complessità di $O(n^2)$. Infatti si possono calcolare, per tutti i programmi p nella sequenza S, gli istanti in cui terminano la loro esecuzione, che sono dati dalla somma dei tempi di esecuzione di p e dei programmi che lo precedono. Tale calcolo richiede $O(n)$ tempo per tutti i programmi, effettuando una sola sommatoria. Confrontando poi gli istanti di terminazione dei programmi con le rispettive scadenze, si può determinare il primo programma p in ritardo e quindi eliminare il programma p' più lungo. Anche questa fase richiede $O(n)$ tempo.

©Alan Bertossi, Alberto Montresor. Algoritmi e Strutture di Dati.

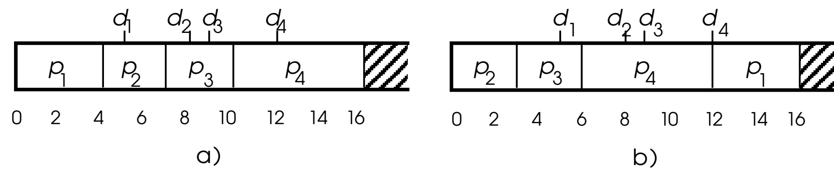

Figura 1: Algoritmo di Moore. a) Ordinamento iniziale; b) soluzione ottima.

Poiché il procedimento di eliminazione di un programma dalla sottosequenza si potrebbe iterare per $O(n)$ volte, la complessità risulterebbe $O(n^2)$.

Utilizzando una coda con priorità Q, realizzata con uno heap [cfr. § **??**], l'algoritmo di Moore può essere realizzato in modo che la sua complessità scenda ad $O(n \log n)$. La procedura moore() utilizza due vettori di interi $d[1 \ldots n]$ e $t[1 \ldots n]$ per le scadenze ed i tempi di esecuzione degli n programmi, rispettivamente, ed un vettore booleano $r[1 \ldots n]$ tale che $r[i]$ = **true** se e solo se il programma i è in ritardo. Dopo aver riordinato i vettori per $d[i]$ crescenti, la procedura considera ciascun programma i, per $i = 1, 2, \ldots, n$, e lo inserisce nella coda con priorità Q, usando una variabile intera T per calcolare la somma dei tempi di esecuzione dei programmi inseriti in Q. Se $T \geq d[i]$, allora il programma i è in ritardo e viene estratto da Q il programma j con priorità (tempo di esecuzione $t[j]$) massima, sottraendo quindi $t[j]$ da T, incrementando k, ed impostando $r[j]$ a **true**. Al termine dell'esecuzione, la variabile k contiene il numero dei programmi in ritardo, mentre il vettore r specifica quali programmi sono o non sono in ritardo. La sequenza ottima è ottenuta eseguendo dapprima i programmi con $r[i]$ = **false** in ordine di indici crescenti, seguiti dai programmi con $r[i]$ = **true** in un ordine qualsiasi.

```
moore(integer[] d, integer[] t, integer n, integer[] r)
    PRIORITYQUEUE Q ← MaxPriorityQueue()
    integer k ← 0
    integer T ← 0
    for integer i ← 1 to n do r[i] ← false
    {ordina d[1...n] e t[1...n] per "scadenze" d[i] crescenti}
    for integer i ← 1 to n do
        Q.insert(i, t[i])
        T ← T + t[i]
        if T ≥ d[i] then
            integer j ← Q.deleteMax()
            T ← T − t[j]
            r[j] ← true
            k ← k + 1
```

L'ordinamento iniziale richiede tempo $O(n \log n)$. Poiché il ciclo **for** è ripetuto n volte, ed al suo interno le operazioni più gravose sono insert() e deleteMax(), che costano $O(\log n)$, il ciclo richiede $O(n \log n)$ tempo. Inoltre, la sequenza ottima può essere ottenuta da r in $O(n)$ tempo tramite una scansione. Pertanto, la complessità della procedura moore() è $O(n \log n)$.

©Alan Bertossi, Alberto Montresor. Algoritmi e Strutture di Dati.