Algoritmi e Strutture di Dati (3^a Ed.)
String matching

Alan Bertossi, Alberto Montresor

STRING MATCHING. *Date una stringa P di m caratteri (pattern) e una stringa T di n caratteri, con $m \leq n$, trovare un'occorrenza di P in T, ovvero trovare un'indice k con $1 \leq k \leq n - m + 1$, tale che il j-esimo carattere di P sia uguale al $(k + j - 1)$-esimo carattere di T, per $1 \leq j \leq m$.*

Ovviamente questo è un caso particolare del problema dello *string matching* approssimato, in cui si cercano occorrenze esatte.

Esempio 1 (String matching)**.** Siano T = `10110010101101011011011` con $n = 23$ e P = `10110110` con $m = 8$. È facile vedere che c'è un'occorrenza del pattern P a partire dalla posizione $k = 14$ del testo T: `10110010101101011011011`. □

Un algoritmo ovvio per risolvere il problema consiste nel cercare di riconoscere il pattern a partire dalla prima posizione del testo, confrontando il primo carattere di P col primo di T, il secondo di P col secondo di T, ecc. Se il pattern non è interamente riconosciuto, si ripete il procedimento a partire dalla seconda posizione nel testo, confrontando il primo carattere di P col secondo di T, il secondo di P col terzo di T, ecc. Se anche stavolta il pattern non è completamente riconosciuto, si riprova a partire dalla terza posizione del testo, poi dalla quarta e così via, continuando fino al riconoscimento completo del pattern o all'esaurimento del testo. Realizzando le stringhe P e T con due vettori di caratteri, si ottiene la seguente funzione ricercaBruta(), che utilizza tre indici: k (per avanzare nel testo), j (per scandire $P[1 \ldots m]$), ed i (per scandire la porzione $T[k \ldots k + m - 1]$ quando cerca di riconoscere se $P[1 \ldots m] = T[k \ldots k + m - 1]$). La funzione restituisce la posizione k di T a partire dalla quale si trova la prima occorrenza di P, se c'è una copia di P in T, oppure $i = n + 1$, se non c'è alcuna copia di P in T.

```
integer ricercaBruta(ITEM[] P, ITEM[] T, integer n, integer m)
    integer i, j, k
    i ← j ← k ← 1
    while i ≤ n and j ≤ m do
        if T[i] = P[j] then i ← i + 1; j ← j + 1
                       else k ← k + 1; i ← k; j ← 1
    return iif(j > m, k, i)
```

L'algoritmo può essere interpretato graficamente come se la soluzione di caratteri individuata dal pattern fosse sovrapposta a quella del testo e si cerchi per quale traslazione del pattern sul testo tutti i caratteri del pattern siano uguali a tutti i caratteri del testo.

Esempio 2 (Sovrapposizione e traslazione)**.** Eseguendo la procedura ricercaBruta() sulle stringhe P e T dell'Esempio 1, si ottiene:

T = 10110010101101011011011 $i = 1, j = 1, k = 1$,
P = 10110110 $i = 6, j = 6$,

T = 10110010101101011011011 $i = 2, j = 1, k = 2$,
P = 10110110 $i = 2, j = 1$,

T = 101100101011010110110 11 $i = 3, j = 1, k = 3$,
P = 10110110 $i = 4, j = 2$,
⋮
T = 10110010101101011011011 $i = 14, j = 1, k = 14$,
P = 10110110 $i = 22, j = 8$, EUREKA! □

Se il pattern non è completamente riconosciuto a partire dalla posizione k del testo, la funzione ricercaBruta() effettua un *backtrack* su entrambi gli indici i e j. Nel caso pessimo, l'indice k può assumere $n - m + 1$ valori e, per ogni valore di k, gli indici i e j possono assumerne m. Essendo in generale m più piccolo di n, la quantità $n - m + 1$ è $O(n)$, e quindi nel caso pessimo sono eseguiti $O(mn)$ confronti tra caratteri di P e T.

Esempio 3 (Caso pessimo ricercaBruta()). Se $A = \{0, 1\}$, il caso pessimo si incontra quando sia P che T sono formate da tutti 0 seguiti da un unico 1 finale. □

Vediamo come un *backtrack* meno brutale permetta di progettare un algoritmo di complessità $O(m+n)$.

Algoritmo di Knuth, Morris e Pratt

L'idea di base dell'algoritmo proposto da Knuth, Morris e Pratt nel 1977 è la seguente: dopo aver riconosciuto $j-1$ caratteri del pattern a partire da una certa posizione nel testo ed aver fallito al j-esimo, perché tornare indietro di $j - 2$ posizioni nel testo? In effetti, i $j - 1$ caratteri già riconosciuti fanno parte del pattern stesso e sono noti addirittura prima di iniziare la ricerca nel testo! Perché non trarre vantaggio da questa informazione nota in anticipo?

Esempio 4 (Backtrack inutile). Se T = 10110010101101011011011 e P = 10110110, i primi 5 caratteri di P sono uguali ai primi 5 caratteri di T, ma il sesto è diverso. Al momento del primo backtrack, $i = j = 6$. Ripartire con $i = 2$ e $j = 1$ corrisponde a traslare a destra di una posizione il pattern rispetto al testo. Ma ciò corrisponde anche a traslare a destra di una posizione rispetto a se stessa la sottosequenza 10110 di P che è stata riconosciuta nel testo! Nella sottosequenza 10110, i primi 4 caratteri non coincidono con gli ultimi 4 ed è pertanto inutile ripartire con $i = 2$, perché sicuramente non potrà esserci una copia di P a partire da $T[2]$. Inoltre, neanche i primi 3 caratteri di 10110 coincidono con gli ultimi 3, ed è altrettanto inutile ripartire con $i = 3$, perché non potrà esserci una copia di P neanche a partire da $T[3]$. I primi 2 caratteri, invece, coincidono con gli ultimi 2 (10110 e 10110) e pertanto conviene ripartire direttamente con $i = 4$. Ma poiché i successivi due caratteri di P (cioè 10) coincidono sicuramente con quelli di T, tanto vale ripartire con $i = 6$ e $j = 3$ (anziché 1). Poiché 6 è proprio il vecchio valore di i al momento del backtrack, è inutile effettuare il backtrack sull'indice i, e basta effettuarlo soltanto sull'indice j, portandolo da 6 a 3. □

Si considerino due copie dei primi $j - 1$ caratteri del pattern, e si immagini di disporle orizzontalmente una sotto all'altra, in modo che il primo carattere della copia inferiore stia "sotto" il secondo

carattere della copia superiore. Se tutti i caratteri sovrapposti nelle due copie non sono uguali, si traslino di una posizione a destra tutti i caratteri della copia inferiore. Si arresti tale procedimento di traslazione non appena tutti i caratteri sovrapposti nelle due copie siano identici, oppure quando non ci siano più caratteri sovrapposti. Il nuovo valore di *backtrack* da assegnare all'indice j, che indichiamo con *back*$[j]$, è proprio uguale al numero di caratteri sovrapposti più uno (se non ci sono caratteri sovrapposti, *back*$[j]$ risulta ovviamente uguale ad 1, come nella ricercaBruta()). Per $2 \leq j \leq m$, si ha:

$$\mathit{back}[j] = \max\{h : h \leq j-2 \wedge P[1 \ldots h-1] = P[j-h+1 \ldots j-1]\}.$$

Per $j = 1$ è conveniente porre *back*$[1] = 0$, come sarà chiarito fra breve.

Esempio 5 (Valori di backtrack)**.** I valori di *back*$[j]$ per il pattern P = `10110110` sono i seguenti, accanto ai quali sono evidenziate per chiarezza anche le massime sovrapposizioni tra le due copie dei primi $j-1$ caratteri di P che portano ai valori ottenuti:

back$[1] = 0$ *back*$[2] = 1$
```
1
 1
```
back$[3] = 1$
```
10
 10
```
back$[4] = 2$
```
101
 101
```
back$[5] = 2$
```
1011
 1011
```
back$[6] = 3$
```
10110
  10110
```
back$[7] = 4$
```
1011101
  101101
```
back$[8] = 5$
```
10111011
  1011011
```
□

In altri termini, quando si verifica se c'è una copia di P a partire da $T[k]$ e risulta $P[1 \ldots j-1] = T[k \ldots i-1]$ ma $P[j] \neq T[i]$, con $k \leq i \leq k+m-1$, allora la prossima posizione di T per tentare di riconoscere P è $i - \mathit{back}[j] + 1$. Ma poiché, per definizione di *back*$[j]$, i primi *back*$[j]-1$ caratteri di $P[1 \ldots j-1]$ coincidono con gli ultimi, si ha che $P[1 \ldots \mathit{back}[j]-1] = T[i-\mathit{back}[j]+1 \ldots i-1]$. Pertanto, i non viene modificato e si va a verificare se c'è una copia di $P[\mathit{back}[j] \ldots m]$ a partire da $T[i]$.

L'algoritmo di Knuth, Morris e Pratt, sotto forma di funzione kmp(), è ottenuto da ricercaBruta() apportando alcune modifiche. Innanzitutto è introdotto in fase di inizializzazione il calcolo del vettore *back*, effettuato con un'opportuna procedura computeBack(). È poi eliminato l'indice k, divenuto un inutile doppione dell'indice i dato che non è effettuato più alcun *backtrack* su i. Il *backtrack* sull'indice j viene effettuato con l'assegnamento $j \leftarrow \mathit{back}[j]$, che sostituisce $j \leftarrow 1$, tranne in un caso particolare. Infatti, se j è uguale ad 1 al momento del *backtrack*, allora significa che $P[1] \neq T[i]$ e che all'iterazione successiva occorre nuovamente considerare $P[1]$, ma confrontandolo con $T[i+1]$. Siccome j è divenuto uguale a 0 con l'assegnamento $j \leftarrow \mathit{back}[j]$, essendo *back*$[1] = 0$ per definizione, allora j è subito reimpostato ad 1 ed i è incrementato. Questo è il motivo per il quale è stato definito *back*$[1] = 0$ come caso particolare. La funzione kmp() restituisce un intero con lo stesso significato di ricercaBruta(); in particolare, se il pattern è stato interamente riconosciuto (cioè se $j > m$) allora la posizione di T a partire dalla quale si trova l'occorrenza di P è $i - m$.

Per inizializzare il vettore *back* si riutilizza una lieve variante dello stesso algoritmo, in cui però si confronta il pattern P con se stesso. I valori di *back*$[j]$ sono computati per $j = 1, 2, \ldots, m$ in accordo alla definizione.

back$[1]$ e h sono inizializzati a 0 prima del ciclo, mentre *back*$[1] = 1$, poiché alla prima iterazione $h = 0$. Se $P[j] \neq P[h]$, allora si tenta con un valore più piccolo di h, il cui valore di *backtrack*

back$[h]$ è stato quindi già computato. Quando invece j ed h sono entrambi incrementati, si ha che $P[1 \ldots h-1] = P[j-h-1 \ldots j-1]$ e quindi *back*$[j] = h$.

```
integer kmp(ITEM[] T, ITEM[] P, integer n, integer m)
    integer[] back ← new integer[1...m]
    computeBack(P, back, m)
    integer i ← 1
    integer j ← 1
    while i ≤ n and j ≤ m do
        if j = 0 or T[i] = P[j] then
            i ← i + 1
            j ← j + 1
        else j ← back[j]
    return iif(j > m, i − m, i)
```

```
computeBack(ITEM[] P, integer[] back, integer m)
    back[1] ← 0
    integer j ← 1
    integer h ← 0
    while j ≤ m do
        if h = 0 or P[j] = P[h] then
            j ← j + 1
            h ← h + 1
            back[j] ← h
        else h ← back[h]
```

Esempio 6. Eseguendo computeBack() sulla stringa P dell'Esempio 1, si ha:

$P =$	10110110	$j = 1, h = 0,$	
		$j = 2, h = 1,$	*back*$[2] = 1,$
$P =$	10110110	$j = 2, h = 1,$	
	1	$j = 2, h =$ *back*$[1] = 0,$	
		$j = 3, h = 1,$	*back*$[3] = 1,$
$P =$	10110110	$j = 3, h = 1,$	
	1	$j = 4, h = 2,$	*back*$[4] = 2,$
	10	$j = 4, h =$ *back*$[2] = 1,$	
$P =$	10110110	$j = 4, h = 1,$	
	1	$j = 5, h = 2,$	*back*$[5] = 2,$
	10	$j = 6, h = 3,$	*back*$[6] = 3,$
	101	$j = 7, h = 4,$	*back*$[7] = 4,$
	1011	$j = 8, h = 5,$	*back*$[8] = 5.$ $\square$

La procedura computeBack() può essere ulteriormente migliorata perché computa *back*$[j]$ senza tener conto dell'esito del confronto tra $P[j]$ e $P[h]$. Infatti, se la funzione kmp() effettua un *backtrack*

sull'indice j per un certo valore dell'indice i, allora deve essere $T[i] \neq P[j]$. Ma in tal caso, se $P[j] = P[h]$ allora anche $T[i] \neq P[h]$. Pertanto, poiché $back[j] = h$, la funzione kmp() effettuerà un secondo *backtrack* su j senza modificare i. Ciò può essere evitato assegnando direttamente $back[h]$ a $back[j]$ qualora $P[j] = P[h]$. A tal fine, è sufficiente modificare la computeBack() sostituendo l'assegnamento $back[j] \leftarrow h$ con l'istruzione:

$$\textbf{if } P[j] = P[h] \textbf{ then } back[j] \leftarrow back[h] \textbf{ else } back[j] \leftarrow h$$

Infatti, poiché il vettore *back* è riempito da sinistra verso destra ed $h < j$, il valore $back[h]$ è stato già computato e può essere utilizzato per calcolare $back[j]$. Questa semplice modifica garantisce che per ciascun valore dell'indice i ci sia al più un solo *backtrack* sull'indice j.

Esempio 7 (computeBack() modificata)**.** Si riconsideri il vettore *back* per P = `10110110`. Applicando la precedente modifica si ottiene:

$back[1] = 0$,
$back[2] = 1$, perché $\texttt{0} = P[2] \neq P[1] = \texttt{1}$,
$back[3] = back[1] = 0$, perché $\texttt{1} = P[3] = P[1] = \texttt{1}$,
$back[4] = 2$, perché $\texttt{1} = P[4] \neq P[2] = \texttt{0}$,
$back[5] = back[2] = 1$, perché $\texttt{0} = P[5] = P[2] = \texttt{0}$,
$back[6] = back[3] = 0$, perché $\texttt{1} = P[6] = P[3] = \texttt{1}$,
$back[7] = back[4] = 2$, perché $\texttt{1} = P[7] = P[4] = \texttt{1}$,
$back[8] = back[5] = 1$, perché $\texttt{0} = P[8] = P[5] = \texttt{0}$. □

Durante l'esecuzione della funzione kmp(), l'indice j può essere decrementato al più una volta per ciascun valore dell'indice i. Poiché i può assumere $O(n)$ valori diversi, il numero di confronti tra caratteri di P e di T eseguiti nel ciclo **while** è $O(n)$. D'altronde, per le stesse ragioni, la procedura computeBack() modificata richiede $O(m)$ confronti tra caratteri. Pertanto, la complessità della funzione kmp() è $O(m + n)$.

Esempio 8 (Algoritmo di Knuth, Morris e Pratt)**.** Eseguendo la funzione kmp() sulle stringhe P e T dell'Esempio 1 ed utilizzando il vettore *back* dell'Esempio 7, si ottiene:

T = `10110010101101011011011` $i = 1, j = 1$,
P = `10110110` $i = 6, j = 6, back[6] = 0$,

T = `10110010101101011011011` $i = 7, j = 1$,
P = `  10110110` $i = 10, j = 4, back[4] = 2$,

T = `10110010101101011011011` $i = 10, j = 2$,
P = `    10110110` $i = 15, j = 7, back[7] = 2$,

T = `10110010101101011011011` $i = 15, j = 2$,
P = `      10110110` $i = 22, j = 8$, EUREKA! □