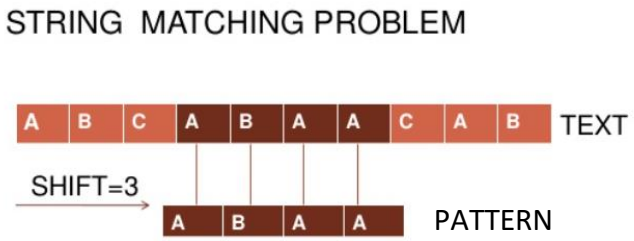
31/05/2021

Algoritmo di Knutt-Morris-Pratt

Progetto di Laboratorio di Algoritmi

Corso di Laurea in Informatica dell'Università di Catania

Elaborato di Kimberly Caziero



INDICE DEI CONTENUTI

Algoritmo di Knutt-Morris-Pratt

Introduzione

L'algoritmo di Knuth-Morris-Pratt (abbreviato come KMP) è estremamente utile nei problemi di pattern-matching su stringhe; le sue applicazioni non si limitano solo all'elaborazione di testi, infatti il suo utilizzo lo rende fondamentale per lo studio delle sequenze ripetute di basi azotate nel DNA, nel campo biologico e della ricerca.

Per comprendere al meglio la qualità del KMP è bene compararlo con altri due metodi di riconoscimento di string matching molto conosciuti: **l'algoritmo di string matching ingenuo** e **l'utilizzo di automi a stati finiti**.

Algoritmo di string matching ingenuo

L'algoritmo di string matching ingenuo, anche detto "naive", è basato su un'idea semplice e la sua implementazione è banale, come vedremo. Questo metodo necessita in input di due stringhe, una sarà quella da ricercare (P) e l'altra quella in cui effettueremo la ricerca di un match con la prima (T).

Pseudocodice

```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4       if P[1..m] == T[s + 1..s + m]
5            print "Pattern occurs with shift" s
```

Funzionamento

Il ciclo for inserito nella riga 3 considera ogni possibile inizio della stringa P in T, entro i limiti della validità. Sarebbero infatti inutili i controlli effettuati su un suffisso di T con una lunghezza minore della lunghezza di P.

Alla riga 4 è implicito un ciclo interno per controllare l'effettiva uguaglianza dei caratteri a seguire al primo, esso inizia quando viene trovato un carattere corrispondente in base al controllo effettuato. Infine, in caso di completa corrispondenza, alla riga 5 verrà stampata la posizione del primo carattere del matching riscontrato, e l'algoritmo continuerà finché i controlli del ciclo esterno avranno senso.

Complessità

È dimostrabile che la complessità di questo algoritmo è pari a Θ((n-m+1) m), ossia **Θ(n²)** se m risulta essere maggiore o uguale a $\lfloor n/2 \rfloor$, dove Θ(n²) è anche il tempo di esecuzione nel caso peggiore. Infine, non avendo svolto alcun lavoro di preelaborazione, il tempo di esecuzione corrisponde al tempo di matching.

L'inefficienza di questo algoritmo è dovuta al fatto che non si conserva alcuna informazione relativa al matching interno, ed eventuali occorrenze possibili per il matching del pattern con sé stesso, oltre che a mancare di una fase di preelaborazione, che aggrava la complessità richiesta.

String matching con automi a stati finiti

Prima di affrontare il problema dello string matching con gli automi a stati finiti è necessario fissare il concetto di automa a stati finiti e il suo funzionamento, così da comprenderne a meglio le sue potenzialità risolutive per questo problema.

Cenni sugli automi a stati finiti

Un **Automa a stati finiti** *M* è una quintupla (*Q*, *q0*, *A*, Σ, δ), dove

- ***Q*** è un determinato set di ***stati***
- ***q0*** ∈ *Q* è lo ***stato di partenza***
- ***F*** ⊆ *Q* è un sottoinsieme di stati detti ***stati di accettazione***
- **Σ** è un determinato ***alfabeto di input***
- **δ** : *Q* x Σ -> *Q* è la ***funzione di transizione*** di *M*.

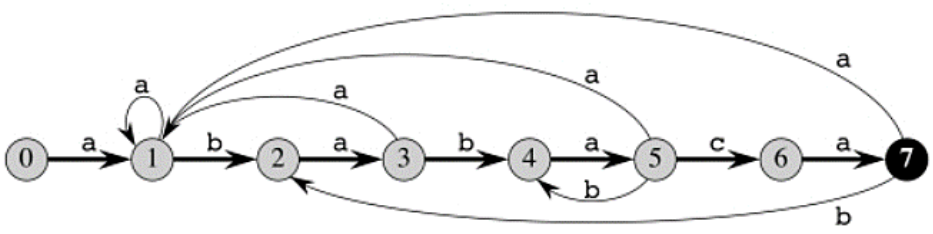
L'automa parte nello stato *q0*. Ogni volta che un carattere (o simbolo) viene letto dall'automa, esso usa δ per determinare in che stato transitare. Non appena *M* entra in uno stato appartenente ad *A*, si dice che l'input è ***accettato***.

Funzionamento

Utilizzando gli automi a stati finiti è possibile ridurre la complessità al fattore lineare, sfruttando la loro caratteristica funzione di transizione **δ**, la quale prende in input lo stato attuale dell'automa $\mathbf{q_i} \in \mathbf{Q}$, l'insieme di tutti gli stati, e il simbolo letto **a**.

Per evitare che ci siano accettazioni errate è importante specificare che lo stato iniziale **q0** è lo stato 0 e che lo stato **m** è l'unico stato finale; questa imposizione è necessaria ai fini del funzionamento applicato al problema dello string matching.

Ciò ci permetterà di fare una preelaborazione sulla stringa P e controllare ogni carattere di T una sola volta. Tuttavia, la creazione di una tabella relativa alla funzione di transizione potrebbe essere dispendiosa nel caso in cui **∑** (alfabeto di input) sia molto esteso.



	input			
state	a	b	c	*P*
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

Dopo aver indicato l'idea generale, scendiamo nei dettagli della funzione da utilizzare. Per costruire l'automa correttamente dovremmo avvalerci dell'uso della funzione che chiameremo **δ**, **funzione suffisso associata a P**. Essa si occuperà di contare **quanti caratteri prefisso di P** sono al contempo **suffisso di T[i]**, la quale è la stringa T considerata fino al suo carattere di indice i, con **1 ≤ i ≤ n**. Quando il valore **δ(q, T[i]) = m** sapremo con certezza che P è contenuta in T.

Pseudocodice e complessità

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
1  m = P.length
2  for q = 0 to m
3      for each character a ∈ Σ
4          k = min(m + 1, q + 2)
5          repeat
6              k = k − 1
7          until P_k ⊐ P_q a
8          δ(q, a) = k
9  return δ
```

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
1  n = T.length
2  q = 0
3  for i = 1 to n
4      q = δ(q, T[i])
5      if q == m
6          print "Pattern occurs with shift" i − m
```

Il tempo di esecuzione di COMPUTE-TRANSITION-FUNCTION è di **O(m³|Σ|)**. È così alto perché i cicli esterni servono per spostarsi nelle righe e nelle colonne della matrice, impiegando un fattore m|Σ| per farlo, mentre il ciclo repeat all'interno può essere eseguito un massimo di m+1 volte al più.

In questo modo otteniamo un **tempo di preelaborazione di O(m|Σ|)** sfruttando qualche miglioria all'algoritmo che calcola la funzione di preelaborazione, e un **tempo di esecuzione del matching di Θ(n).**

Algoritmo di Knutt-Morris-Pratt

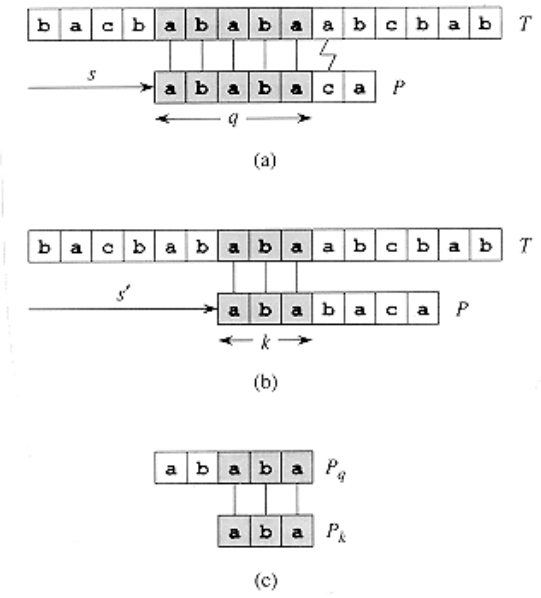
L'algoritmo di KMP sfrutta la preelaborazione, come l'algoritmo basato sugli automi a stati finiti, ma ne riduce la complessità e lo spazio richiesto, mettendosi così un passo avanti a quest'ultimo per complessità.

L'idea è simile, sfruttare la preelaborazione della stringa P per evitare inutili cicli interni e confronti di cui si saprà già il risultato. Anche in questo caso al tempo di esecuzione passeremo una sola volta su ogni carattere di T, ottenendo una complessità lineare su T.

Funzione di preelaborazione

La **funzione prefisso π** per un pattern P ci permette di confrontare P con sé stesso e sapere a priori di quanto spostarci su T. La rappresentazione della funzione prefisso sarà fatta tramite un array che sarà consultato ogni qualvolta ci sia un primo caso di mismatch o di ricerca dell'occorrenza successiva.

Stiamo facendo quel lavoro di "ricordare" i match o i non-match che l'algoritmo ingenuo non fa, senza dover utilizzare memoria aggiuntiva oltre l'array **π di dimensione m**.



Poniamoci nella situazione *(a)*. Proviamo a capire come spostarci nella situazione *(b)* senza rivedere i caratteri già confrontati.

Sappiamo a vista d'occhio che, confrontando il carattere **c con a,** essi non combaceranno. Un occhio attento potrebbe anche notare che il prefisso di P confrontato mostra delle ripetizioni al suo interno, e sfruttandole sappiamo già per certo che gli ultimi tre confronti combaciano con i primi tre caratteri di P.

Possiamo perciò portare avanti l'indice su T e portare indietro il confronto di P, entrambe di **un totale di posizioni q-k**, senza effettuare di nuovo confronti *(c).*

In conclusione, sapendo che $\mathbf{P_q \sqsupset T_{s+q}}$, vogliamo trovare il più lungo **prefisso proprio Pk di Pq** che è al contempo un suffisso di $\mathbf{T_{s+q}}$**.** Sommiamo la differenza q-k delle lunghezze di questi prefissi di P per spostarci da s a s'. Nel caso migliore il prefisso sarà la stringa vuota, così k=0 ci farà spostare saltando tutti gli spostamenti da s+1 ad s+q-1.

Formalizzando, dato un pattern P[1…m], la funzione prefisso per il pattern P è la funzione $\mathbf{\pi:\{1, 2, \dots, m\} \dashrightarrow \{0, 1, \dots, m-1\}}$ tale che $\mathbf{\pi[q]= max \{k : k<q \ \& \ P_k \sqsupset P_q\}}$**.**

Funzione di matching

La funzione di matching di KMP consiste nel salvare le due lunghezze delle stringhe T e P nelle variabili n ed m, esattamente come in precedenza, e calcolare subito dopo **π sulla stringa P**.

Settata la variabile **q=0**, si entra nel ciclo più esterno, il quale scorrerà la stringa T con **l'indice i**. Il ciclo interno verrà eseguito solo quando q>0 e ci sarà un mismatch tra i caratteri da confrontare, e serve per consultare la funzione **π** in merito allo spostamento da effettuare, aggiornando q.

Altrimenti, **q è incrementata in caso di match**, e se **q=m** è stata trovata un'occorrenza.

Pseudocodici e complessità

```
KMP-MATCHER(T, P)
1   n = T.length
2   m = P.length
3   π = COMPUTE-PREFIX-FUNCTION(P)
4   q = 0                                  // number of characters matched
5   for i = 1 to n                         // scan the text from left to right
6       while q > 0 and P[q + 1] ≠ T[i]
7           q = π[q]                       // next character does not match
8       if P[q + 1] == T[i]
9           q = q + 1                      // next character matches
10      if q == m                          // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                       // look for the next match
```

```
COMPUTE-PREFIX-FUNCTION(P)
1  m = P.length
2  let π[1..m] be a new array
3  π[1] = 0
4  k = 0
5  for q = 2 to m
6      while k > 0 and P[k + 1] ≠ P[q]
7          k = π[k]
8      if P[k + 1] == P[q]
9          k = k + 1
10     π[q] = k
11 return π
```

La complessità del KPM-MATCHER, come detto in precedenza, assume ordine di **Θ(n).**

Utilizzando la procedura COMPUTE-PREFIX-FUNCTION rispetto alla corrispettiva per gli automi abbiamo ridotto il limite di tempo di preelaborazione da **O(m|Σ|)** a **Θ(m)**. Il limite di matching totale ed effettivo sarà mantenuto perciò a **Θ(n).**

La correttezza tutti gli algoritmi presentati in questo paragrafo è ricercabile nel libro citato nella bibliografia alle pagine 839, 841-843.

Bibliografia

Ogni immagine presente nel report è stata presa dal libro:

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, L. Colussi – "Introduzione agli algoritmi e strutture dati, terza edizione" – McGrawHill, 2010

con il rispetto delle leggi sul copyright.