Algoritmi per il problema del Pattern Matching

Università degli Studi di Perugia
Corso di Laurea in Ingegneria Informatica
Algoritmi e Strutture Dati

Gian Marco Ferri

Indice

1 Il problema del Pattern Matching

Il problema del Pattern Matching, che consiste nell'individuare l'occorrenza di un certo pattern testuale all'interno di un testo più grande, riveste una grande importanza in moltissime applicazioni. In ambito informatico le più comuni si trovano nell'elaborazione di testi, nei motori di ricerca su Internet, nelle biblioteche digitali, per non parlare della bioinformatica, altro settore in cui il problema del Pattern Matching assume grande importanza nell'elaborazione delle sequenze di DNA.

I documenti di testo sono onnipresenti nell'informatica moderna, poiché vengono utilizzati per comunicare e pubblicare informazioni. Dal punto di vista della progettazione di algoritmi, tali documenti possono essere considerati come semplici stringhe di caratteri. Eseguire operazioni di ricerca ed elaborazione su tali dati, pertanto, richiede di disporre di metodi, ovvero algoritmi, efficienti per gestire tali stringhe.

Le stringhe possono essere elaborate in diversi modi; un'operazione tipica è il taglio di stringhe di grandi dimensioni in stringhe più piccole.

Per descrivere i pezzi che risultano da tale operazione usiamo il termine sottostringa. Una sottostringa di una stringa P lunga m caratteri è una stringa della forma $P[i]P[i+1]P[i+2]...P[j]$ o più semplicemente $P[i...j]$, per qualche $0 \leq$i$\leq$j$\leq$m-1, ovvero è la stringa formata dai caratteri di P che vanno dall'indice i all'indice j inclusi.

Se $i > j$ considereremo $P[i...j]$ una stringa nulla, ovvero una stringa di lunghezza 0. Indicheremo con $P[0...i]$, per $0 \leq$i$\leq$j$\leq$m-1, il prefisso della stringa P e con $P[i...m-1], per 0 \leq$i$\leq$m-1, il suffisso P. Si noti che la stringa nulla è prefisso e suffisso di qualunque altra stringa.

Nell problema del Pattern Matching, a fronte di una stringa T di lunghezza n che rappresenta un testo e una stringa P di lunghezza m che rappresenta il pattern, viene chiesto di verificare se P è una sottostringa di T. La nozione di match (corrispondenza) indica la presenza di una sottostringa di T che inizia in un qualche indice i e che corrisponde a P, carattere per carattere. In formule: $T[i] = P[0]$, $T[i+1] = P[1]$, $T[i+m-1] = P[m-1]$, ovvero $P = T[i...i+m-1]$. L'output di un qualsiasi algoritmo di Pattern Matching può essere o l'indice di T in cui inizia la sottostringa P o un'indicazione che il pattern P non compare in T.

Per rendere l'elaborazione condotta sulle stringhe di caratteri più generale possibile, normalmente non si limita l'analisi a caratteri di T e P provenienti esplicitamente da un set di caratteri ben noto, come quello ASCII o Unicode. Tipicamente viene invece usato il simbolo generale Σ per denotare l'insieme di caratteri dell'alfabeto da cui possono provenire i caratteri di P e T. Questo alfabeto può ovviamente essere un sottoinsieme dei set di caratteri ASCII o Unicode, ma potrebbe anche essere più generale e può anche essere infinito. Tuttavia, poiché la maggior parte degli algoritmi di elaborazione dei documenti sono utilizzati in applicazioni in cui l'insieme di caratteri è finito, di solito si assume che la dimensione dell'alfabeto Σ, indicato con $|\Sigma|$, sia una costante fissata.

2 L'algoritmo Brute Force

L'algoritmo Brute Force è sicuramente l'algoritmo più intuitivo e più semplice tra quelli che risolvono il problema del Pattern Matching. Esso esamina ogni carattere del testo con il carattere iniziale del pattern. Una volta trovato un riscontro si procede con un confronto tra i successivi caratteri del testo con il resto del pattern. Se non si verifica una occorrenza si continua a controllare il testo carattere per carattere nel tentativo di ottenere un altro riscontro. In questo modo bisogna esaminare quasi ogni carattere del testo finché non si trova una corrispondenza del pattern oppure il testo finisce.

L'algoritmo è composto da due cicli annidati. Il ciclo esterno controlla tutti i possibili indici iniziali del pattern nel testo, il ciclo interno controlla ogni carattere del pattern comparandolo con il potenziale carattere corrispondente nel testo.

Algorithm 1: BruteForce(T, P)

Input: String T (testo) di n caratteri e stringa P (pattern) di m caratteri

Output: Indice iniziale della prima sottostringa di T che corrisponde a P, o un'indicazione che P non è una sottostringa di T

```
1 for i ← 0 to n − m do
2     j ← 0
3     while (j < m and T[i + j] = P[j]) do
4         j ← j + 1
5     if j = m then
6         return i
7 return Non esiste una sottostringa di T che corrisponda a P
```

Nel caso peggiore per ogni indice candidato in T l'algoritmo esegue fino a m confronti tra caratteri per scoprire che P non corrisponde a T all'indice corrente. Il ciclo esterno è eseguito al massimo $n - m + 1$ volte, mentre quello interno al massino m volte. Quindi la complessità dell'algoritmo Brute Force nel caso peggiore è $O((n - m + 1)m)$, ovvero $O(nm)$. Quando n e m sono circa uguali l'algoritmo ha una complessità quadratica.

La principale inefficienza dell'algoritmo Brute Force è che quando si testa un possibile piazzamento del pattern vengono effettuati un certo numero di confronti ma non appena si ha una mancata corrispondenza tutta l'informazione ricavata dai confronti precedenti viene persa.

3 L'algoritmo Boyer Moore

A differenza dell'algoritmo Brute Force che esamina ogni carattere del testo per trovare un pattern, l'algoritmo Boyer-Moore può evitare a volte il confronto tra il pattern e una frazione considerevole dei caratteri nel testo. Mentre l'algoritmo Brute Force può funzionare anche con un alfabeto potenzialmente illimitato, l'algoritmo Boyer Moore assume invece che l'alfabeto sia di dimensione fissa e finita.

L'algoritmo Boyer-Moore si basa su due euristiche: l'euristica dello specchio e l'euristica del salto del carattere. La prima afferma che quando si verifica un possibile posizionamento del pattern P nel testo T, si deve iniziare ad eseguire i confronti dalla fine di P e poi spostarsi all'indietro verso l'inizio di P. L'euristica del salto del carattere afferma che quando si verifica un possibile posizionamento del pattern P nel testo T, un mismatch (mancata corrispondenza) del carattere del testo $T[i] = c$ con il corrispondente carattere del pattern $P[j]$ è gestita come segue: se c non è contenuto da nessuna parte in P allora occorre spostare completamente P oltre $T[i]$ (perché $T[i]$ non corrisponde a nessun carattere in P); altrimenti si deve spostare P finché un'occorrenza del carattere c in P non si allinea con $T[i]$.

Le due euristiche permettono di evitare il confronto tra P e interi gruppi di caratteri in T perché se si confronta la fine del pattern con il testo allora possono essere compiuti dei salti all'interno del testo piuttosto che confrontare ogni singolo carattere, evitando così molti confronti inutili.

Allineando il pattern al testo, l'ultimo carattere del pattern viene confrontato con il carattere corrispondente del testo. Se i caratteri non corrispondono non c'è bisogno di effettuare confronti con i caratteri precedenti. Lo scorrimento a salti lungo il testo per effettuare confronti piuttosto che controllare ogni singolo carattere riduce il numero di confronti da compiere, che è la chiave per l'aumento dell'efficienza dell'algoritmo. Il confronto continua finché non viene raggiunto l'inizio di P (significa che si è verificata un'occorrenza) o si ha un mismatch che genera uno spostamento a destra dell'allineamento dettato dall'euristica del salto del carattere. I confronti vengono nuovamente eseguiti col nuovo allineamento e il processo si ripete finché o viene trovata un'occorrenza di P o l'allineamento viene portato oltre la fine di T; ciò significa che non verranno trovate altre corrispondenze.

Per implementare l'euristica del salto del carattere si definisce la funzione $last(c)$ che prende in input un carattere c e produce come output l'indice dell'ultima occorrenza (a destra) di c in P. Se c non è presente in P si pone convenzionalmente $last(c) = -1$. La funzione $last(c)$ specifica quanto si deve spostare il pattern P se nel testo viene trovato un carattere uguale a c che non corrisponde al pattern.

Algorithm 2: BoyerMoore(T, P)

Input: String T (testo) di n caratteri e stringa P (pattern) di m caratteri

Output: Indice iniziale della prima sottostringa di T che corrisponde a P, o un'indicazione che P non è una sottostringa di T

```
1  i ← m − 1
2  j ← m − 1
3  repeat
4      if P[j] = T[i] then
5          if j = 0 then
6              return i
7          else
8              i ← i − 1
9              j ← j − 1
10     else
11         i ← i + m − min(j, 1 + last(T[i]))
12         j ← m − 1
13 until i > n − 1
14 return Non esiste una sottostringa di T che corrisponda a P
```

La correttezza dell'algoritmo Boyer-Moore deriva dal fatto che ogni volta che il metodo fa uno spostamento, non viene saltata nessuna possibile corrispondenza poiché $last(c)$ indica la posizione dell'ultima occorrenza di c in P.

Il calcolo della funzione $last(c)$ richiede un tempo pari a $O(m + |\Sigma|)$, mentre la ricerca effettiva del pattern richiede $O(nm + |\Sigma|)$ nel caso peggiore. Quindi la complessità dell'algoritmo Boyer-Moore nel caso peggiore è $O(nm + |\Sigma|)$.

4 L'algoritmo Knuth Morris Pratt (KMP)

Nello studiare la complessità di caso peggiore dell'algoritmo Brute Force e dell'algoritmo Boyer-Moore su specifiche istanze del problema si può notare una maggiore inefficienza. Si possono eseguire molti confronti mentre viene testato un potenziale posizionamento del pattern nel testo, tuttavia se si scopre un carattere del pattern che non corrisponde nel testo, si buttano via tutte le informazioni ottenute da questi confronti e si ricomincia da capo con il successivo posizionamento del pattern. L'algoritmo Knuth-Morris-Pratt (KMP) evita lo spreco di informazione e così facendo ottiene una migliore complessità nel caso peggiore.

L'idea su cui si basa l'algoritmo Knuth-Morris-Pratt è che quando il confronto fra il carattere del testo e quello del pattern fallisce dopo un certo numero di successi, invece di far arretrare l'indice sul pattern e sul testo, cioè fermare il confronto tra pattern e testo e ricominciare la ricerca ignorando l'esito dei confronti già eseguiti, si possono sfruttare le conoscenze sul pattern evitando i confronti il cui esito è già noto.

La peculiarità dell'algoritmo KMP risiede nel pretrattamento del pattern da cercare, il quale contiene l'indicazione sufficiente a determinare la posizione da cui continuare la ricerca in caso di mismatch. L'elaborazione del pattern avviene mediante una failure function f che indica il corretto spostamento di P tale da poter riutilizzare i confronti eseguiti in precedenza nella misura più ampia possibile.

Con tale metodo si possono non riesaminare i caratteri che erano stati precedentemente verificati durante il confronto tra pattern e testo, ed inoltre il confronto non ricomincia da zero, ovvero azzerando l'indice del pattern, ma prosegue per un certo valore stabilito dalla failure function, anche se nella peggiore delle ipotesi si riparte dall'inizio del pattern.

Nello specifico la failure function f prende in input il pattern e restituisce un vettore contenente, per ciascun valore dell'indice j relativo a P, la lunghezza del prefisso più lungo di P che è suffisso di $P[1...j]$. Per convenzione si pone $f(0) = 0$. La failure function confronta il pattern con se stesso. Ogni volta che due caratteri corrispondono si pone $f(i) = j + 1$. Visto che $i > j$ durante l'esecuzione dell'algoritmo, $f(j-1)$ è sempre definita quando deve essere utilizzata.

Algorithm 3: KMPFailureFunction(P)

Input: Stringa P (pattern) di m caratteri
Output: La failure function f per P che mappa j alla lunghezza del prefisso più lungo di P che è un suffisso di $P[1...j]$

```
1  i ← 1
2  j ← 0
3  f(0) ← 0
4  while i < m do
5      if P[j] = P[i] then
6          f(i) ← j + 1
7          i ← i + 1
8          j ← j + 1
9      else if j > 0 then
10         j ← f(j − 1)
11     else
12         f(i) ← 0
13         i ← i + 1
```

L'importanza della failure function è che codifica sottostringhe ripetute all'interno del pattern stesso. La tabella creata dalla failure function contiene quindi le informazioni necessarie per sapere di quanto traslare il pattern ogni volta che avviene un mismatch in un confronto tra pattern P e testo T.

L'algoritmo KMP elabora in modo incrementale il testo T confrontandolo con il pattern P. Ogni volta che c'è una corrispondenza si incrementano gli indici. D'altro canto se non c'è una corrispondenza e si erano fatti progressi in P, si consulta la failure function per determinare il nuovo indice in P che indica dove si deve continuare a verificare la presenza di P in T. Se invece non c'è una corrispondenza e si è all'inizio di P, si incrementa semplicemente l'indice di T e si mantiene l'indice di P come all'inizio. Si ripete questo processo fino a quando non si trova una corrispondenza di P in T o l'indice di T raggiunge n (la lunghezza del testo T) stando ad indicare che non si è trovato il pattern nel testo.

La parte principale dell'algoritmo KMP è il ciclo *while* che esegue per ogni iterazione un confronto tra un carattere in T e un carattere in P. A seconda del risultato ottenuto, l'algoritmo o passa al carattere successivo in T e in P e consulta la failure function per un nuovo carattere candidato in P, o ricomincia con l'indice successivo in T.

La correttezza dell'algoritmo deriva dalla definizione della failure function. Infatti quest'ultima garantisce che tutti i confronti ignorati sono ridondanti, poiché essi implicherebbero il confronto tra caratteri che già sappiamo corrispondere.

Algorithm 4: KnuthMorrisPratt(T,P)

Input: String T (testo) di n caratteri e stringa P (pattern) di m caratteri

Output: Indice iniziale della prima sottostringa di T che corrisponde a P, o un'indicazione che P non è una sottostringa di T

```
1  f ← KMPFailureFunction(P)
2  i ← 0
3  j ← 0
4  while i < n do
5      if P[j] = T[i] then
6          if j = m − 1 then
7              return i − m + 1
8          i ← i + 1
9          j ← j + 1
10     else if j > 0 then
11         j ← f(j − 1)
12     else
13         i ← i + 1
14 return Non esiste una sottostringa di T che corrisponda a P
```

Escludendo il calcolo della failure function, la complessità dell'algoritmo KMP è chiaramente proporzionale al numero di iterazioni del ciclo *while*. Si definisce $k = i = j$, dove k rappresenta quanto il pattern P è stato spostato rispetto al testo T. Si noti che durante l'esecuzione dell'algoritmo si ha $k \leq n$. Ad ogni iterazione del ciclo si verifica uno dei seguenti tre casi:

- Se $T[i] = P[j]$ si incrementa i di 1 e k non cambia poiché anche j aumenta di 1.
- Se $T[i] \neq P[j]$ e $j > 0$, allora i non cambia e k viene incrementato di almeno 1, poiché in questo caso k varia da $i - j$ a $i - f(j - 1)$
- Se $T[i] \neq P[j]$ e $j = 0$, allora i e k vengono incrementati di 1, poiché j non cambia.

Quindi ad ogni iterazione del ciclo o i o k vengono incrementati di almeno 1; quindi il numero totale di iterazioni del ciclo *while* è al massimo $2n$.

La complessità della failure function è $O(m)$. La sua analisi è analoga a quella dell'algoritmo KMP. Si ha quindi che la complessità dell'algoritmo KMP nel suo complesso è $O(n + m)$ nel caso peggiore.

5 Esempio applicativo

In questa sezione viene riportato l'output del programma basandosi sul testo contenuto in "testo.txt", ovvero il "Lorem ipsum", e un pattern scelto casual-

mente contenuto in "pattern.txt". Di seguito vengono riportati due esempi: una prova con un pattern presente nel testo e una prova con il pattern non presente. I tempi di esecuzione, espressi in millisecondi, potrebbero variare da un'esecuzione ad un'altra.

Pattern presente

Brute Force
Found at position: 92
Number of comparisons: 15
Computation time: 0.0158 ms

Boyer Moore
Found at position: 92
Number of comparisons: 11
Computation time: 0.0308 ms

Knuth Morris Pratt
Found at position: 92
Number of comparisons: 14
Computation time: 0.026 ms

Pattern non presente

Brute Force
Pattern is not matched in the text
Number of comparisons: 24
Computation time: 0.0374 ms

Boyer Moore
Pattern is not matched in the text
Number of comparisons: 5
Computation time: 0.0547 ms

Knuth Morris Pratt
Pattern is not matched in the text
Number of comparisons: 24
Computation time: 0.0372 ms

6 Dati sperimentali

Per confrontare le prestazioni dei tre algoritmi sono state condotte diverse prove sperimentali, ognuna con una diversa configurazione dei dati di input, ovvero stringa di testo e di pattern.

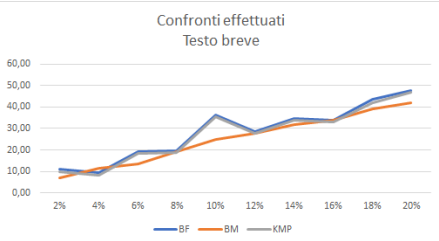
In particolare sono stati selezionati un testo di lunghezza breve (le prime due terzine della Divina Commedia), un testo di lunghezza media (Lorem ipsum da 3500 caratteri) e un testo lungo (Lorem ipsum da 10 mila caratteri). In corrispondenza di ognuno di essi sono stati testati i risultati degli algoritmi, ovvero il tempo impiegato e il numero di confronti effettuati, al variare della lunghezza del pattern da cercare. Quest'ultimo varia dal 2% al 20% (con passo del 2%) della lunghezza del testo in esame.

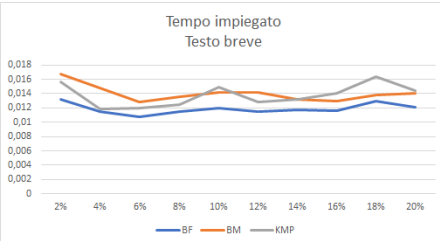
Visto che le prestazioni degli algoritmi dipendono fortemente dalle caratteristiche dei testi e dei pattern, nonché dalle variazioni delle prestazioni del calcolatore utilizzato, sono state generate diverse stringhe di pattern in posizione randomica per ogni lunghezza, per poi generare una media dei numeri dei confronti effettuati e del tempo impiegato dai tre algoritmi.

Si sottolinea che i dati sono stati raccolti utilizzando lo stesso calcolatore e in condizioni di carico simili.

Vengono di seguito discussi i risultati ottenuti per ogni tipologia di testo.

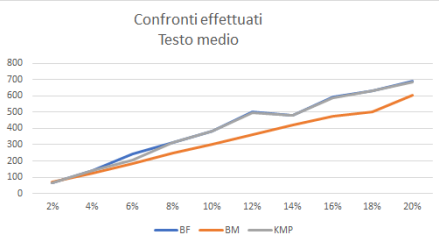
6.1 Testo breve

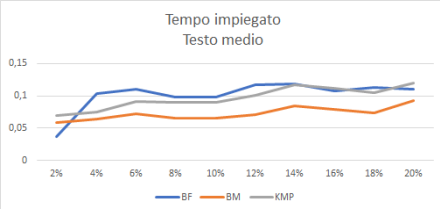




Per i testi di breve lunghezza l'algoritmo Boyer Moore effettua in media meno confronti rispetto a KMP e Brute Force. Con questa tipologia di testo gli algoritmi Boyer Moore e KMP non sono molto efficienti in quanto il tempo di esecuzione è dominato dal tempo di pre processing del pattern, soprattutto nel caso in cui quest'ultimo è di lunghezza considerevole.

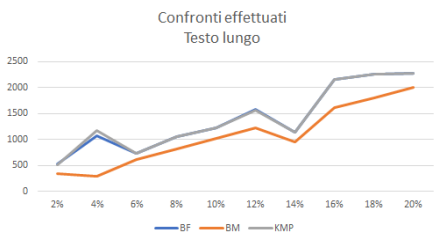
6.2 Testo medio

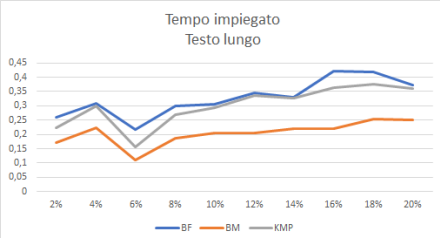




Anche nel caso di ricerca su un testo di media lunghezza, le prestazioni di KMP e Boyer Moore degradano all'aumentare della dimensione del pattern, restando comunque migliori di quelle del Brute Force. Questo fenomeno si osserva soprattutto per pattern in percentuale più lunghi, che portano la fase di pre processing a dominare su quella di ricerca.

6.3 Testo lungo





Anche in questo caso valgono le stesse considerazioni dei casi precedenti. Si può notare come le prestazioni degli algoritmi KMP e Boyer Moore restano comunque migliori di quelle del Brute Force.

6.4 Conclusioni

L'andamento temporale del Brute Force risulta particolarmente dipendente dalla lunghezza del testo in cui si ricerca, risultando più efficiente in testi di breve lunghezza e inefficiente in testi di grande lunghezza. Questo algoritmo è particolarmente efficace quando la lunghezza del pattern è molto limitata, in quanto il numero di confronti in questo caso risulta particolarmente ridotto.

Come previsto, l'algoritmo Boyer Moore e l'algoritmo KMP risultano in media più efficienti di quello Brute Force. In effetti nella maggior parte dei casi e soprattutto per pattern lunghi, Boyer Moore richiede un minor numero totale di confronti, grazie ai salti di carattere che è in grado di effettuare. Al contempo, l'algoritmo KMP, pre processando il pattern tramite la failure function, risulta particolarmente efficiente con un alfabeto più piccolo e con stringhe che hanno prefissi che sono anche suffissi.