SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Magistrale in

INFORMATICA

Algoritmi e Strutture Dati

Prof.ssa Rossella Petreschi

Anno accademico 2010/2011

Il problema dello string-matching

Studente:
Flavio Pietrelli

Indice

Introduzione

Il cosiddetto problema dello *string-matching* o *corrispondenza tra stringhe* trova efficacia in un così grande insieme di applicazioni che non è possibile elencarle tutte. In ambito informatico, le più comuni si trovano nell'elaborazione di testi, in strumenti di ricerca quali *grep* di UNIX, nei motori di ricerca su Internet, nelle biblioteche digitali, nei giornali elettronici, negli elenchi telefonici on-line e anche nelle grandi enciclopedie on-line. Altro importante settore nel quale gli algoritmi di *string-matching* assumono rilevanza, è la bioinformatica; il tema centrale della biologia computazionale è la computazione su sequenze molecolari (stringhe) e costituisce un importante punto di contatto tra biologia e informatica. Considerare il DNA come una stringa mono-dimensionale composta da una ripetizione dei quattro caratteri: adenina A, citosina C, guanina G e timina T, permette di definire i problemi fondamentali della biologia molecolare come problemi computazionali su stringhe di lunghezza finita. Ricostruire lunghe stringhe di DNA da frammenti che si sovrappongono, confrontare due o più stringhe per scoprire similarità, ricercare sottosequenze che occorrono con una certa frequenza nelle sequenze di DNA, costituiscono i problemi che più strettamente legano l'informatica alla biologia.

Lo scopo di questa tesina è quello di approfondire, analizzare ed evidenziare la complessità dei principali algoritmi per il problema dello *string-matching* presenti in letteratura, illustrando altresì implementazioni alternative in grado di migliorarne l'efficienza.

Il testo è diviso nei seguenti capitoli:

- *String-matching*: presentazione e introduzione al problema del confronto tra stringhe, analisi del primo algoritmo risolutivo e descrizione di algoritmi efficienti noti in letteratura.
- *La famiglia di algoritmi Boyer-Moore*: approfondimento sulla famiglia di algoritmi Boyer-Moore, presentazione delle alternative e descrizione delle diverse procedure e similitudini degli uni con gli altri.
- *Conclusioni*: conclusioni e resoconto del lavoro svolto.
- *Appendice: Definizioni fondamentali per le stringhe, notazione e terminologia*: breve introduzione formale al concetto di "stringa".

Capitolo 1

String-matching

Trovare tutte le occorrenze di una stringa all'interno di un testo è un problema che si presenta frequentemente nei programmi di scrittura dei testi. Il testo è tipicamente un documento che si sta scrivendo e la stringa cercata, chiamata anche *pattern*, è una particolare parola fornita dall'utente. Più formalmente [1]:

Sia Σ un *alfabeto* di dimensione finita di *simboli* distinguibili, ad esempio $\Sigma = \{a, b \dots, z\}$ oppure $\Sigma = \{0,1\}$.

Si considera il testo come un array $T[1, n]$ di lunghezza n ed il pattern come un array $P[1, m]$ di lunghezza m, con $m \leq n$. Gli elementi di P e T sono caratteri presi dall'alfabeto Σ e per questo i due array vengono anche chiamati *stringhe* su Σ.

Si dice che il *pattern* P appare *con spostamento* (o *shift*) s nel testo T (o, equivalentemente, il pattern P *occorre a cominciare dalla posizione* $s + 1$ nel testo T) se $0 \leq s \leq n - m$ e $T[s + 1, s + m] = P[1, m]$. Se P appare con spostamento s in T, allora si dice che s è uno *spostamento valido*, altrimenti s è uno *spostamento non valido*.

L'obiettivo del problema della *corrispondenza tra stringhe* è quello di trovare tutti gli spostamenti validi con cui un dato pattern P appare in un testo T; nell'esempio

mostrato in Figura 1.1 il pattern $P = abaa$ compare una sola volta nel testo $T = abcabaabcabac$ con spostamento $s = 3$.

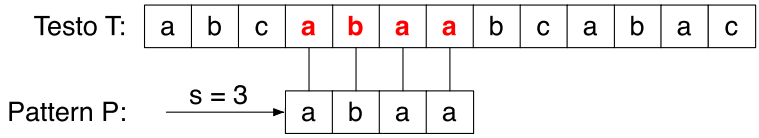


Figura 1.1: Esempio di spostamento valido

In questo primo capitolo saranno introdotti i più importanti algoritmi di *string-matching* presenti in letteratura, ognuno con il rispettivo pseudocodice con l'analisi della complessità; tutti sono inoltre accompagnati da esempi e figure per facilitarne la comprensione nei passaggi più difficili e per fornire al lettore un'idea più chiara del comportamento dell'algoritmo durante la sua esecuzione.

Poiché si farà spesso ricorso a concetti quali *stringa su un alfabeto* Σ, *prefisso*, *suffisso*, *sottostringa*, *concatenazione di stringhe*, ecc, per maggiori dettagli si può far riferimento al capitolo *Appendice* in calce a questo testo.

1.1 Algoritmo ingenuo

L'algoritmo ingenuo [1] trova tutti gli spostamenti validi del pattern all'interno del testo utilizzando un ciclo che controlla la condizione $P[1, m] = T[s + 1, s + m]$ per tutti i possibili $n - m + 1$ spostamenti di s. Il metodo ingenuo allinea il primo carattere del pattern P con il primo carattere del testo T e confronta tutti i caratteri di P in T, scorrendo il pattern da sinistra a destra, finché o si arriva alla fine di P, nel qual caso viene segnalata una occorrenza di P in T , oppure finché vengono incontrati due caratteri diversi tra loro. In entrambi i casi il pattern viene spostato in avanti di una posizione e la procedura ricomincia confrontando i caratteri a partire dal primo carattere di P con il nuovo carattere del testo ad esso allineato. Il procedimento si ripete fintanto che non viene trovata una sottostringa del testo coincidente con il pattern, oppure finché l'ultimo carattere di P ($P[m]$) non sorpassa l'ultimo carattere di T ($T[s + m]$).

La Figura 1.2 mostra un esempio di esecuzione dell'algoritmo ingenuo applicato alle due stringhe $P = abcdabce$ e $T = cabcdabcdabce$. Il colore verde di un carattere del pattern indica un confronto positivo con un carattere uguale del testo (si dice in questo caso che vi è un *match*) mentre il colore rosso indica un confronto con un carattere diverso del testo (in tal caso si dice che vi è un *mismatch*). I caratteri del pattern di colore bianco non vengono confrontati.

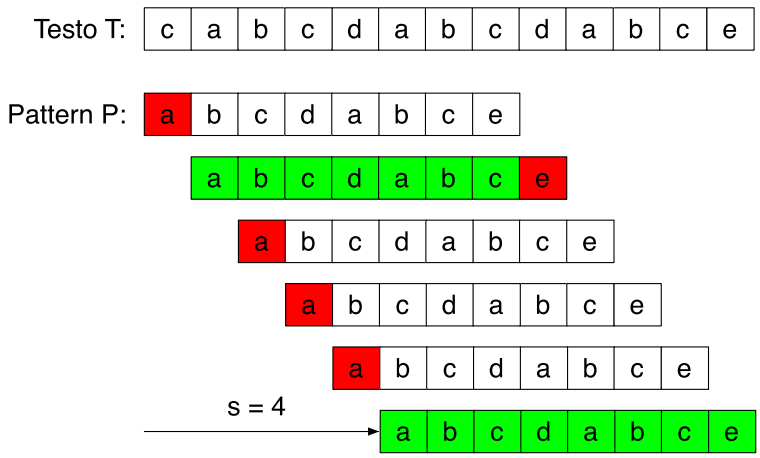


Figura 1.2: Esempio di esecuzione dell'algoritmo ingenuo

Il totale di confronti effettuati dall'algoritmo ingenuo nell'esempio proposto è 20.

Lo pseudocodice dell'algoritmo è il seguente

```
Ingenuo(P,T)
      n ← length(T)
      m ← length(P)
      for s ← 1 to n - m + 1
            j ← 1
            while ((j ≤ m) and (P[j] == T[s + j - 1]))
                  j ← j + 1
            if (j > m)
                  print "Il pattern appare con spostamento" s
```

Per studiare la complessità dell'algoritmo si consideri il caso peggiore in cui $P = a^m$ e $T = a^n$ sono stringhe costituite dallo stesso carattere a ripetuto rispettivamente m ed n volte. Il pattern P occorre ovviamente in ognuna delle prime $n - m + 1$ posizioni di T pertanto, l'algoritmo effettua esattamente $m(n - m + 1)$ confronti per una complessità nel caso peggiore pari a $\Theta(mn)$.

Il metodo ingenuo è certamente semplice da capire e da programmare ma il suo tempo di calcolo $\Theta(mn)$ nel caso peggiore è alquanto insoddisfacente e deve essere migliorato. In letteratura sono presenti diverse varianti dell'algoritmo ingenuo atte a migliorarne l'efficienza e, con alcuni di questi, è possibile portare la complessità nel caso peggiore da $\Theta(mn)$ a $O(m + n)$.

1.1.1 Idee su come migliorare la complessità dell'algoritmo ingenuo

La prima idea per rendere asintoticamente più efficiente l'algoritmo è sicuramente quella di cercare di spostare il pattern P di più posizioni quando si incontra un mismatch tra due caratteri corrispondenti, senza, però, rischiare di perdere qualche occorrenza di P in T. Tale soluzione permette di far scivolare più velocemente il pattern nel testo e quindi di risparmiare un consistente numero di confronti.

Altri metodi per migliorare l'efficienza dell'algoritmo prevedono di risparmiare confronti evitando, dopo aver spostato il pattern, di riesaminare i caratteri di una parte iniziale dello stesso.

Maggiori dettagli su come sviluppare ed implementare le suddette migliorie saranno presenti nel seguito di questo testo, al momento ci si limita a fornire un'idea intuitiva dei possibili perfezionamenti.

Tornando all'esempio di Figura 1.2, il cui pattern era $P = abcdabce$ ed il testo $T = cabcdabcdabce$, l'algoritmo ingenuo richiede 20 confronti, 5 dei quali sono mismatch e 15 sono match. Un primo miglioramento consiste nell'accorgersi che dopo aver effettuato i primi 9 confronti, le 3 successive posizione del pattern non possono essere occorrenze valide di P in T. I match precedenti suggeriscono che i caratteri del testo in tali posizioni ($a\ b\ c$) sono uguali rispettivamente al secondo, terzo e quarto carattere del pattern, che a loro volta sono tutti diversi dal primo (a). Dopo i primi 9 confronti, quindi, l'algoritmo sposta il pattern non di una sola posizione, ma di ben quattro nel modo indicato in Figura 1.3.

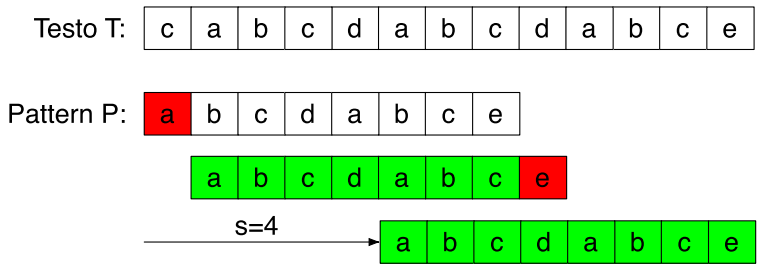


Figura 1.3: Esempio di una prima ottimizzazione dell'algoritmo ingenuo

Il numero di confronti scende a 17.

Un algoritmo ancor più efficiente dovrebbe accorgersi, inoltre, che dopo lo spostamento del pattern di quattro posizioni, i primi tre caratteri del pattern ($a\ b\ c$) sono uguali ai corrispondenti tre caratteri del testo. I precedenti match assicurano che i su indicati tre caratteri di T sono uguali al quinto, sesto e settimo del pattern ($a\ b\ c$) che, a loro volta, sono uguali ai primi tre caratteri di P. Dopo lo spostamento del

pattern di quattro posizioni, l'algoritmo dovrebbe quindi cominciare a confrontare i caratteri di *P* con i corrispondenti caratteri del testo partendo non dal primo ma dal quarto (come indicato in Figura 1.4); con questa ottimizzazione il numero di confronti è ulteriormente ridotto e per l'esempio in considerazione scende a 14.

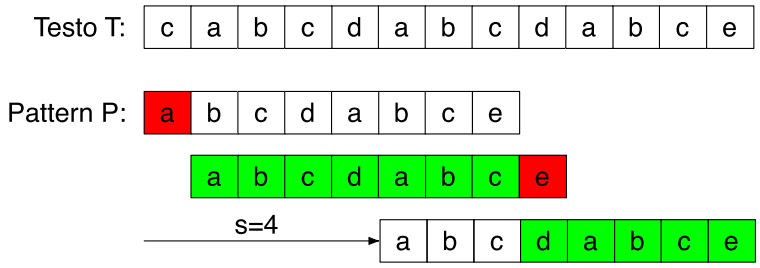


Figura 1.4: Esempio di una seconda ottimizzazione dell'algoritmo ingenuo

Implementazioni efficienti di queste idee sono state sviluppate nei diversi algoritmi trattati in questo testo. La realizzazione di algoritmi efficienti è possibile per mezzo di un processo di analisi preliminare del pattern (eventualmente anche del testo), che prende il nome di *pre-elaborazione* (o *pre-processing*) e consiste di un'analisi preliminare delle stringhe in oggetto, al fine di spendere un tempo "modesto" per apprenderne la struttura interna e sfruttare le informazioni raccolte in una seconda fase di elaborazione che è la ricerca vera e propria del pattern all'interno del testo.

1.2 Algoritmo di Knuth, Morris e Pratt

L'algoritmo di Knuth, Morris e Pratt, pubblicato per la prima volta nel 1977 [2] [3], è il primo algoritmo basato su confronti con complessità lineare nel caso peggiore e come quello ingenuo, allinea il pattern P in posizioni successive del testo T ed effettua vari confronti alla ricerca di eventuali match tra le due stringhe in esame. Terminati i controlli su una sottosequenza di T, il pattern viene spostato verso destra e la procedura reiterata.

A differenza di quello ingenuo, l'algoritmo di Knuth, Morris e Pratt, tiene conto di quelle idee proposte nel *paragrafo 1.1.1* che permettono di spostare il pattern per più di una posizione, ricominciando i confronti non dall'inizio, ma da un carattere successivo. Da sempre considerato uno dei migliori algoritmi di *string-matching*, il suddetto rappresenta il punto di riferimento per lo sviluppo di molti altri algoritmi di corrispondenza tra stringhe presenti attualmente in letteratura.

L'algoritmo si divide in due fasi: la fase di pre-elaborazione del pattern e la fase di ricerca all'interno del testo, ognuna delle quali partecipa al raggiungimento della complessità che, come accennato, è nel caso peggiore lineare nella somma della dimensione del testo e del pattern.

1.2.1 Pre-elaborazione del pattern: la funzione prefisso

Il processo di pre-elaborazione è gestito per mezzo di un algoritmo che prende il nome di *funzione prefisso* e che, in tempo $O(m)$, restituisce un array π utilizzato successivamente per evitare inutili controlli durante la fase di ricerca. Esso rappresenta il cuore dell'algoritmo Knuth, Morris e Pratt, permettendo di ottenere importanti informazioni, utilizzate per ridurre il numero di confronti e mantenere lineare la complessità del medesimo. Per una maggiore chiarezza la descrizione della *funzione prefisso* è divisa in due sottoparagrafi: nel primo è proposta l'idea generale;

nel secondo, invece, è presentato e descritto l'algoritmo che la implementa, il suo pseudocodice e la relativa analisi della complessità.

1.2.1.1 L'idea

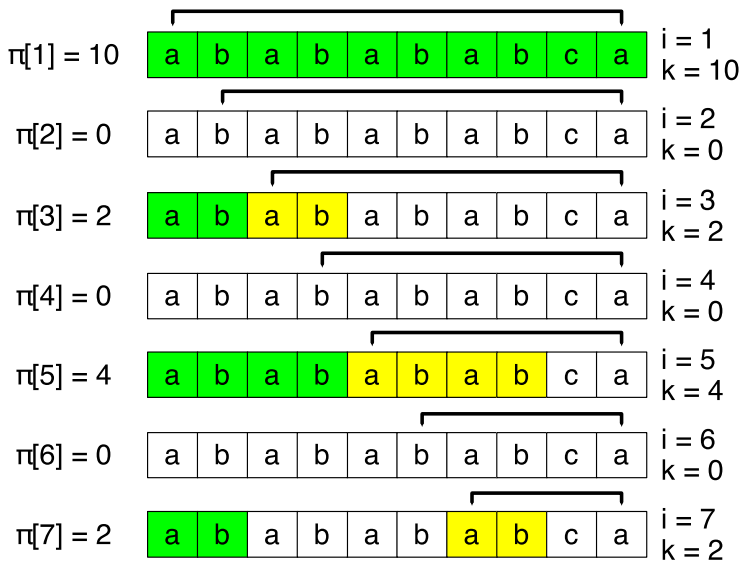
La *funzione prefisso* è utilizzata per analizzare la struttura interna del pattern ed evitare inutili controlli durante la successiva fase di ricerca.

Dato un pattern $P[1, m]$ di lunghezza m, si consideri un array $\pi[1, m]$ di dimensione m e sia $\pi[i]$ il valore la lunghezza del più lungo prefisso di P che occorre in posizione i in P. Formalmente:

$$\pi[i] = \max\{\, k \mid P[1, k] = P[i, i + k - 1] \,\}$$

ossia $\pi[i]$ è la lunghezza del più lungo prefisso comune tra P e il suo suffisso $P[i, m]$.

La Figura 1.5 fornisce il risultato della funzione prefisso per il pattern $P = \{ababababca\}$; si noti che $\pi[1] = m$ per ogni pattern P di lunghezza m poiché il più lungo prefisso comune tra P e $P[1, m] = P$ è tutto P.



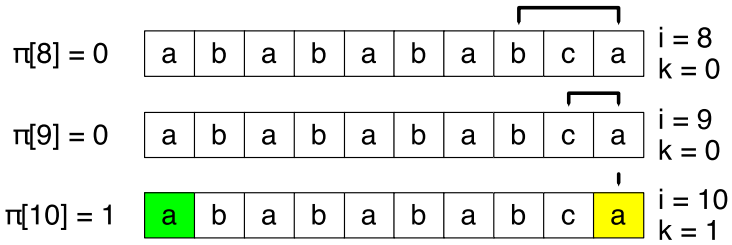


Figura 1.5: Esempio di esecuzione della funzione prefisso

Per definizione $P[i, i + \pi[i] - 1]$ è un'occorrenza in P del prefisso $P[1, \pi[i]]$ il che significa che la stringa $y = P[1, \pi[i]] = P[i, i + \pi[i] - 1]$ è un bordo del prefisso $P[1, i + \pi[i] - 1]$ di P e, di conseguenza, $p = i - 1$ è un periodo di tale prefisso (Figura 1.6).

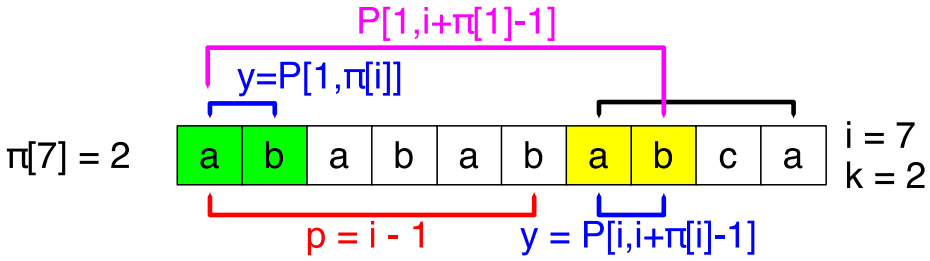


Figura 1.6: Struttura del pattern in funzione di $\pi[i]$

Se $i = 1$ allora $y = P$ e $p = 0$; in questo caso il bordo y e il periodo p si dicono bordo e periodo *degeneri* di P.

Se $i \geq 2$ allora $i + \pi[i] - 1 = m$ e quindi $y = P$, oppure $P[1 + \pi[i]] \neq P[i + \pi[i]]$; in entrambi i casi, la stringa $P[1, i + \pi[i] - 1]$ è il più lungo prefisso di P avente periodo $p = i - 1$.

Le sottostringhe $P[i, i + \pi[i] - 1]$ di P non sono necessariamente disgiunte, ma possono sovrapporsi. Dato un $i \geq 2$ si considerino tutte le sottostringhe $P[j, j + \pi[j] - 1]$ con $2 \leq j \leq i$ e si indichi con r_i il valore massimo dell'estremo destro $j + \pi[j] - 1$ di tali sottostringhe e con $l_i = j$ l'estremo sinistro corrispondente.

$P[l_i, r_i]$ è quindi la sottostringa di P che termina più a destra tra tutte quelle che sono uguali ad un prefisso di P e che iniziano in posizioni compresa tra 2 ed i.

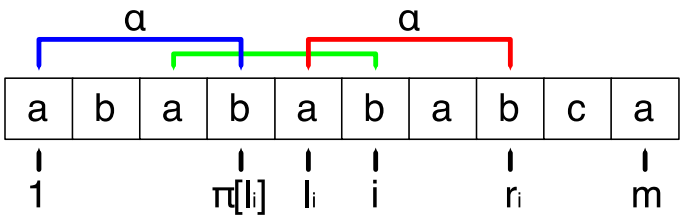


Figura 1.7: Esempio di ricerca della sottostringa $P[l_i, r_i]$

In riferimento all'esempio di Figura 1.5, con pattern $P = \{abababababca\}$, per $i = 6$ una possibile soluzione è quella mostrata in Figura 1.7 dove sono evidenziate le occorrenze dei prefissi di P che iniziano tra le posizioni 2 e 6. L'occorrenza che termina più a destra è evidenziata in rosso; essa inizia nella posizione $l_i = 5$, termina nella posizione $r_i = 8$, ha lunghezza $\pi[l_i] = 4$ ed è uguale al prefisso α di P, indicato in blu, di pari lunghezza.

Per provare il risultato ottenuto è sufficiente appurare che fissata una posizione $i = 6$, $P[l_i, r_i] = P[5,8]$ è la sottostringa di P che termina più a destra tra tutte quelle che sono uguali ad un prefisso di P e che iniziano in posizioni compresa tra 2 ed $i = 6$; infatti:

- Per $j = 2$, $\pi[2] = 0$ e $r_i = 2 + 0 - 1 = 1$, $P[2,1]$ non è una sottostringa valida.
- Per $j = 3$, $\pi[3] = 2$ e $r_i = 3 + 2 - 1 = 4$, $P[3,4]$ è una sottostringa valida (in verde in Figura 1.7).
- Per $j = 4$, $\pi[4] = 0$ e $r_i = 4 + 0 - 1 = 3$, $P[4,3]$ non è una sottostringa valida.
- Per $j = 5$, $\pi[5] = 4$ e $r_i = 5 + 4 - 1 = 8$, $P[5,8]$ è una sottostringa valida (in rosso in Figura 1.7).
- Per $j = 6$, $\pi[6] = 0$ e $r_i = 6 + 0 - 1 = 5$, $P[6,5]$ non è una sottostringa valida.

$P[5,8]$, dove $l_i = 5$ e $r_i = 8$, è quindi la sottostringa di P che termina più a destra tra tutte quelle che sono uguali ad un prefisso di P e che iniziano in posizioni compresa tra 2 e 6.

1.2.1.2 L'algoritmo

Per ragioni tecniche si consideri di aggiungere in coda a P un carattere *sentinella* diverso da tutti quelli del pattern, allora $P = P\$$ e $|P| = m + 1$.

L'algoritmo di pre-elaborazione pone $\pi[1] = m + 1$ e successivamente calcola i valori di $\pi[i]$, r_i ed l_i per $i = 2, \dots, m + 1$. Poiché per eseguire l'i-esima operazione si utilizzano solo i valori di r_{i-1} e l_{i-1} l'algoritmo in realtà usa le due singole variabili r ed l per memorizzare i successivi valori r_i e l_i.

Il valore di $\pi[2]$ fa eccezione dagli altri ed è calcolato direttamente, confrontando da sinistra verso destra i caratteri di $P[2, m + 1]$ con i caratteri di P, finché non viene trovato un mismatch (che deve esserci necessariamente per via del carattere *sentinella*). Il valore di $\pi[2]$ è la lunghezza del massimo prefisso comune ad entrambe le stringhe, ad $r = r_2$ viene assegnato il valore $\pi[2] + 1$ e ad $l = l_2$ viene assegnato il valore 2.

Assumendo di aver calcolato i valori di $\pi[j]$ per ogni $j = 2 \dots i - 1$ ed i valori di $r = r_{i-1}$ e di $l = l_{i-1}$, per calcolare $\pi[i]$, $r = r_i$ e $l = l_i$ possono verificarsi due casi:

Caso 1: Se $i > r$, $\pi[i]$ viene calcolato direttamente confrontando, da sinistra verso destra, i caratteri delle due stringhe $P[i, m + 1]$ e P fino a trovare un mismatch. Il valore di $\pi[i]$ è uguale alla lunghezza k del massimo prefisso comune e si pone $r = i + \pi[i] - 1$ e $l = i$.

Caso 2: Se $i \leq r$, allora il carattere $P[i]$ è contenuto nella sottostringa $\alpha = P[l, r]$ che, per definizione di $\pi[l]$ ed r, è il prefisso $\alpha = P[1, \pi[l]]$ della stringa P. Il carattere $P[i]$ compare quindi anche nella posizione $i' = i - l + 1$ di P e per la stessa ragione la sottostringa $\beta = P[i, r]$ compare anche in posizione i' ossia $\beta = P[i', \pi[l]]$ (Figura 1.8).

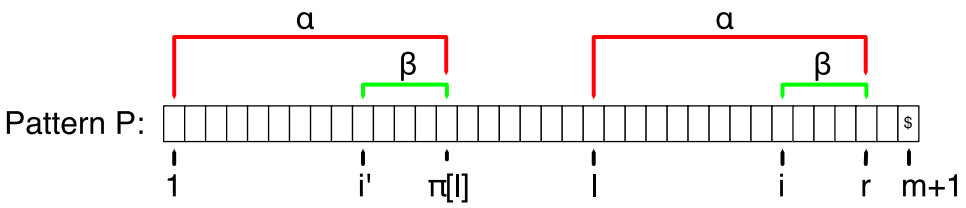


Figura 1.8: Funzione prefisso: caso 2

Poiché $\beta = P[i', \pi[l]]$, in i' occorre un prefisso γ di P di lunghezza $\pi[i']$ che occorre a partire dalle posizioni 1 e i' e, per la parte di lunghezza minore o uguale alla lunghezza di β, anche dalla posizione i (dato che o β contiene γ oppure β è contenuto in γ). Ne consegue che P e il suo suffisso $P[i, m+1]$ (che inizia in posizione i) hanno un prefisso comune di lunghezza uguale al minimo tra $\pi[i']$ e $|\beta| = r - i + 1$; i due casi possono essere considerati separatamente:

Caso 2a: Se $\pi[i'] < |\beta|$ allora $\pi[i] = \pi[i']$ ed r ed l rimangono invariati.

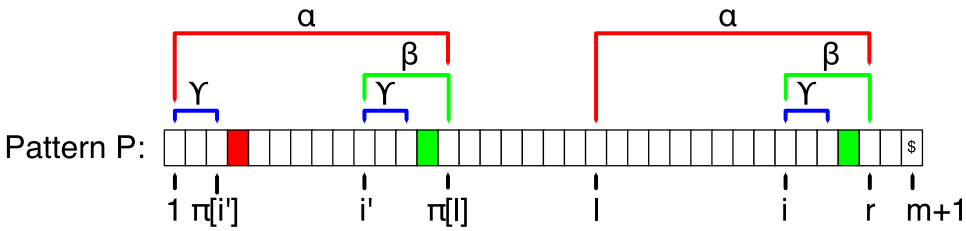


Figura 1.9: Funzione prefisso: caso 2a

Come mostrato in Figura 1.9, se $\pi[i'] < |\beta|$ il prefisso γ di lunghezza $\pi[i']$, oltre a comparire in posizione 1, compare anche nelle posizioni i' ed i e il carattere successivo (in rosso) è diverso dai due caratteri (in verde) successivi alle due occorrenze in posizione i' ed i (uguali tra loro). Segue che $\pi[i] = \pi[i']$.

Caso 2b: Se $\pi[i'] \geq |\beta|$ allora l'intera sottostringa $P[i, r]$ è un prefisso di P e quindi $\pi[i] \geq |\beta| = r - i + 1$. L'algoritmo calcola la lunghezza k del massimo prefisso comune tra P e $P[i, m+1]$ cominciando i confronti a partire dai caratteri in posizione $|\beta| + 1$ e $|\beta| + i = r + 1$, proseguendo fino a trovare un mismatch; una volta trovato, $\pi[i]$ viene posto uguale a k e si pone $r = i + \pi[i] - 1$ e $l = i$.

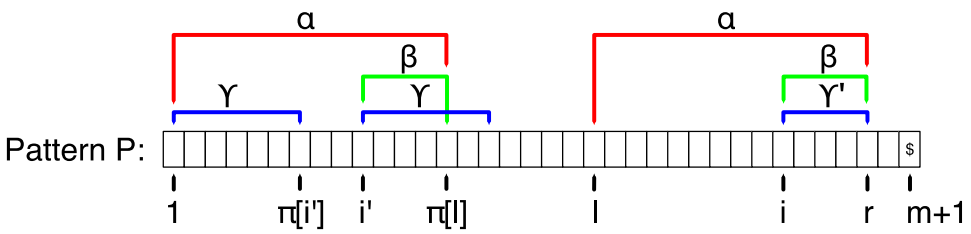


Figura 1.10: Funzione prefisso: caso 2b

Un maggior chiarimento è dato dalla Figura 1.10; se $\pi[i'] \geq |\beta|$ il prefisso γ di lunghezza $\pi[i']$ compare in posizione 1, i' ed i. In quest'ultimo caso compare soltanto la parte $\gamma' = \beta$ contenuta in α. Per calcolare $\pi[i]$ è quindi possibile controllare direttamente i caratteri successivi.

Di seguito lo pseudocodice dell'algoritmo *Funzione_prefisso()*:

```
Funzione_prefisso(P)
      m ← length(P)
      π[1] ← m
      k ← 0
      while P[1+k] == P[2+k]
            k ← k + 1
      π[2] ← k
      l ← 2
      r ← 2 + k - 1
      for i ← 3 to m
            if (r < i)                              //Caso 1
                  k ← 0
                  while (P[1+k] == P[i+k])
                        k ← k + 1
                  π[i] ← k
                  l ← i
                  r ← i + k - 1
            else if (π[i-l+1] < (r-i+1))            //Caso 2a
                  π[i] ← π[i - l + 1]
            else                                    //Caso 2b
                  k ← r - i + 1
                  while (P[1+k] == P[i+k])
                        k ← k + 1
                  π[i] ← k
                  l ← i
                  r ← i + k - 1
      return π
```

L'analisi della complessità permette di dimostrare che la funzione prefisso calcola tutti i valori di $\pi[i]$ in tempo $O(m)$.

Il primo ciclo *while*, quello esterno al *for*, calcola semplicemente $\pi[2]$ e per ciascuno dei successivi valori dell'indice $i = 3, \dots, m$ il calcolo di $\pi[i]$ richiede l'esecuzione, al massimo, di uno solo dei due cicli *while* più interni. In ognuno di questi, il confronto coinvolge i due caratteri $P[1 + k]$ e $P[i + k]$; in caso di esito negativo l'esecuzione viene arrestata, mentre, in caso positivo il controllo si sposta ai due caratteri successivi. Poiché ognuno dei *while* termina non appena viene trovato un mismatch, il numero totale di confronti con esito negativo è al più $m - 1$.

Al termine di entrambi i cicli *while* si pone $r = i + k - 1$; alla successiva iterazione del ciclo *for*, se $r < i$ il successivo ciclo *while* inizia con il carattere destro $P[i]$ per via dell'istruzione $k \leftarrow 0$, altrimenti inizia con il carattere destro $P[r + 1]$ per via dell'istruzione $k = r - i + 1$. In entrambi i casi, durante tutta l'esecuzione dell'algoritmo, il carattere destro non si sposta mai a sinistra e il numero di confronti eseguiti con esito positivo è al più $m - 1$.

Il totale dei confronti è quindi sempre minore o uguale di $2m - 2$ è la complessità dell'algoritmo al più $O(m)$.

La Figura 1.11 mostra i confronti effettuati dall'algoritmo *Funzione_prefisso()* con input la stringa $P = abaaabaabaaabc$. Al solito i match sono indicati in verde e i mismatch in rosso; i colori verde chiaro e rosso chiaro indicano rispettivamente i match e i mismatch che l'algoritmo *Funzione_prefisso()* risparmia rispetto ad una versione ingenua che calcola direttamente i vari $\pi[i]$ per ogni $i = 2, \dots, m$.

È importante notare che la *Funzione_prefisso()* ha al più un solo match per ogni carattere del testo e un solo mismatch per ogni allineamento del pattern nel testo.

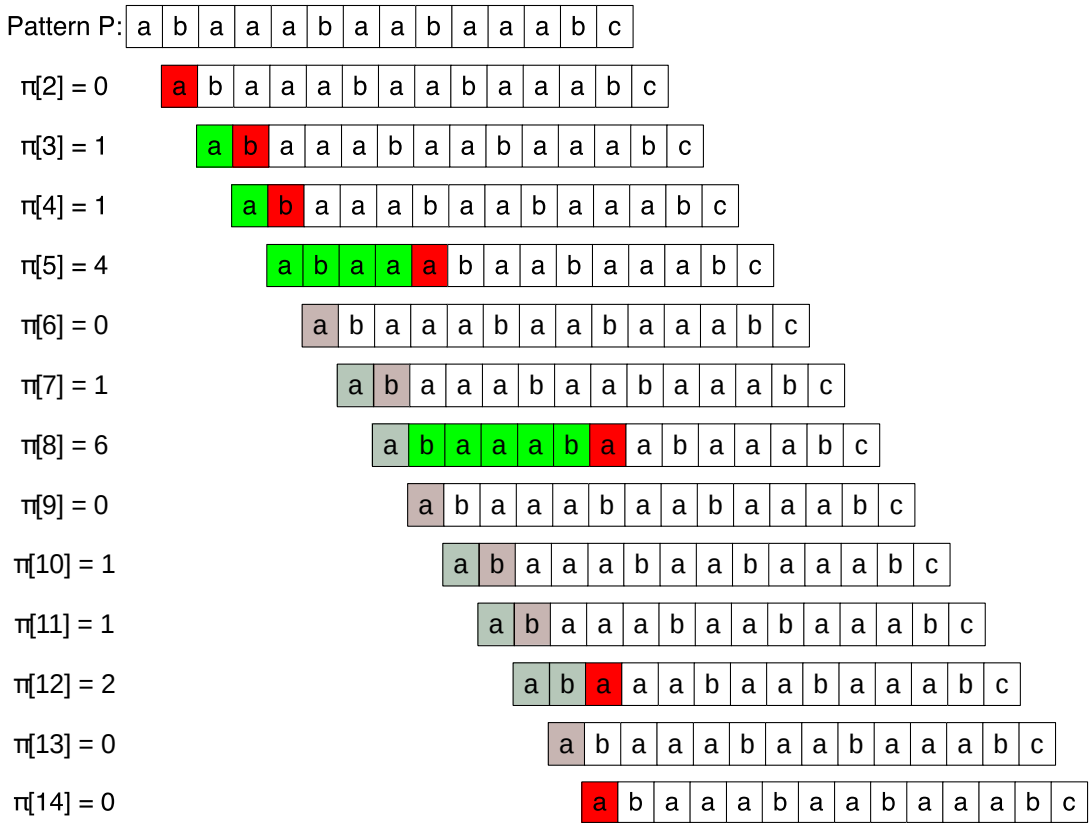


Figura 1.11: Esempio di esecuzione dell'algoritmo funzione prefisso

1.2.2 Ricerca nel testo: descrizione dell'algoritmo di Knuth, Morris e Pratt

Dopo aver presentato la *funzione prefisso*, è ora possibile descrivere come avviene la ricerca della corrispondenza del pattern all'interno del testo.

Per facilitare i controlli sulla dimensione del pattern, si consideri di aggiungere anche in coda a T un carattere *sentinella* diverso anch'esso da tutti quelli del testo e del pattern; a questo punto $T = T\#$ e $|T| = n + 1$.

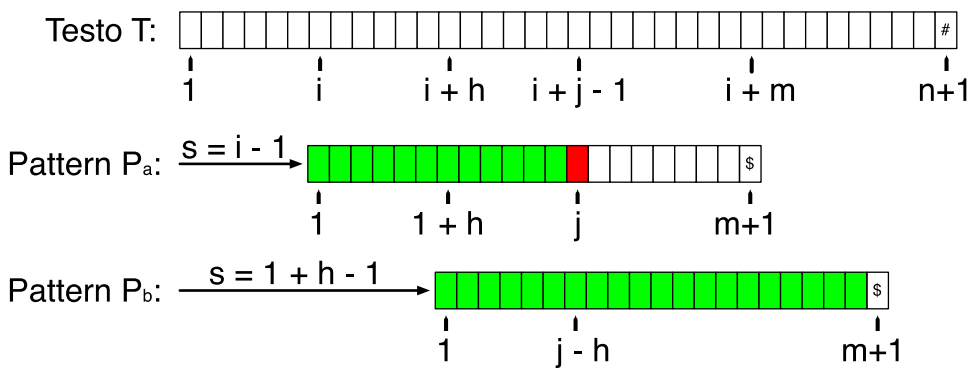


Figura 1.12: Esempio di scorrimento del pattern nell'algoritmo di Knuth, Morris e Pratt

In riferimento all'esempio mostrato in Figura 1.12, dove i match sono colorati in verde e i mismatch in rosso, si supponga di avere il pattern allineato in posizione i del testo e che l'algoritmo abbia trovato il primo mismatch confrontando il carattere $P_a[j]$ con il carattere $T[i+j-1]$; ovviamente $P_a[1, j-1] = T[i, i+j-2]$ e $P_a[j] \neq T[i+j-1]$. La presenza del carattere *sentinella* garantisce di trovare senz'altro un mismatch per qualche $j \leq m+1$.

Tenendo presente che $P_a = P_b$ e che la differenza dei nomi è data per la sola necessità di identificare le due istanze dell'esempio, si consideri ora un'occorrenza $P_b[1, m] = T[i+h, i+h+m-1]$ dove il pattern è stato spostato in avanti di una quantità h compresa tra 1 e $j - 1$. In questo caso $P_b[1, j-h] = T[i+h, i+j-1]$.

Poiché nell'istanza precedente era $P_a[1, j-1] = T[i, i+j-2]$ deve essere $P_b[1, j-h-1] = T[i+h, i+j-2] = P_a[1+h, j-1]$ e siccome $P_a[j] \neq T[i+j-1]$ deve essere anche $P_b[j-h] = T[i+j-1] \neq P_a[j]$.

Dire che $P_b[1, j-h-1] = P_a[1+h, j-1]$ e che $P_b[j-h] \neq P_a[j]$ è come dire che $\pi_{P_a}[i+h] = j-h-1$ ovvero che $j = h + \pi_{P_a}[1+h] + 1$ pertanto, gli spostamenti h compresi tra 1 e $j-1$ per i quali vi può essere un'occorrenza del pattern in posizione $i+h$ del testo, sono soltanto quelli per cui $j = h + \pi_{P_a}[h+1] + 1$. Se questa condizione non è soddisfatta per nessun h, con $1 \leq h \leq j-1$, la prima posizione del testo in cui può esserci un'occorrenza del pattern è $i+j$.

Trovato il primo mismatch confrontando $P_a[j]$ con $T[i+j-1]$, lo spostamento del pattern che si deve effettuare è dato dalla formula

$$s[j] = \min(\{\, j \,\} \cup \{\, h : \; 1 \leq h < j \text{ e } j = 1 + h + \pi_{P_a}[1+h] \,\}\,)$$

Assumendo che al pattern sia stato aggiunto un carattere *sentinella* allora la stessa definizione di $s[j]$ continua a valere anche nel caso in cui il mismatch sia stato trovato in corrispondenza di questo (per $j = m + 1$) ottenendo così

$$s[m+1] = \min(\{j\} \cup \{\, h \mid 1 \leq h < m+1 \text{ e } m+1 = 1 + h + \pi_{P_a}[1+h] \,\}\,) \leq m$$

Dopo aver spostato il pattern di $s[j]$ caratteri, portando i in posizione $i' = i + s[j]$ del testo, possono verificarsi due casi:

- Se $s[j] < j$ allora $P[1, j - s[j] - 1] = T[i + s[j], i + j - 2]$ e si può cominciare a confrontare i caratteri del pattern, con i corrispondenti caratteri del testo, partendo dalla posizione $j' = j - s[j]$ del pattern.
- Se invece $s[j] = j$ il confronto inizia dalla posizione $j' = 1$ del pattern.

Di seguito è riportato lo pseudocodice della fase di ricerca dell'algoritmo di Knuth, Morris e Pratt:

```
Knuth_Morris_Pratt(P,T) // P = P$ e T = T#
      π ← Funzione_prefisso(P)
      for j ← 1 to m + 1
            s[j] ← j
      for h ← m downto 1
            s[1+h+π[1+h]] ← h
      i ← j ← 1
      while (i ≤ n - m + 1)
            while (P[j] == T[i+j-1])
                  j ← j + 1
            if (j > m)
                  print "Il pattern appare con spostamento" i
            i ← i + s[j]
            j ← max(j - s[j], 1)
```

L'analisi della complessità deve ovviamente tener conto del tempo impiegato per lo svolgimento delle fasi di pre-elaborazione e di ricerca.

La procedura di pre-elaborazione è svolta per mezzo dell'algoritmo *Funzione_prefisso()* che come visto impiega un tempo $O(m)$.

Per quanto riguarda la fase di ricerca, invece, il tempo richiesto è proporzionale al numero di confronti tra i caratteri che vengono verificati nel ciclo *while* più interno alla ricerca di un mismatch. Poiché il pattern viene spostato in avanti non appena si incontra un mismatch, il numero di confronti con esito negativo che possono verificarsi sono tanti quante le posizioni del testo a cui il pattern viene allineato. Il numero di queste posizioni è al più $n - m + 1$ e per questo il numero di confronti con esito negativo è $O(n - m + 1)$.

Per i casi positivi, il carattere del testo che viene confrontato con $P[j]$ è quello in posizione $i + j - 1$ ed ogni volta che viene eseguito un confronto con esito positivo si considera il carattere immediatamente seguente. La posizione $i + j - 1$ ha valore 1 all'inizio dell'algoritmo e ad ogni spostamento del pattern rimane invariato oppure aumenta di 1 in funzione di j; poiché la diminuzione di j è minore o uguale all'aumento di i, al termine dell'algoritmo $i + j - 1 \leq n - 1$. Si può concludere che il numero di confronti effettuati con esito positivo è al più $O(n)$.

Tenendo in considerazione entrambe i casi, la complessità totale dell'algoritmo nel caso peggiore è $O(m + n)$.

1.3 Algoritmo di Rabin-Karp

L'algoritmo di Rabin-Karp [4] [1], sviluppato da Michael O. Rabin e Richard M. Karp e presentato ufficialmente nel 1987, si differenzia da tutti gli algoritmi presenti in questo testo per la procedura di pre-processing basata su concetti di teoria dei numeri e non di confronto diretto tra i caratteri delle stringhe interessate.

L'idea alla base dell'algoritmo è quella di utilizzare il concetto matematico di *classi di congruenza* per convertire sia il pattern che il testo, in stringhe numeriche e, tramite il confronto dei loro valori, determinare tutte le possibili occorrenze della stringa di input all'interno del testo.

Per maggiore chiarezza, si assuma di avere un alfabeto $\Sigma = \{0,1,2,\dots,9\}$ e che ogni carattere dell'alfabeto Σ sia un cifra decimale; sotto questa ipotesi ogni stringa su Σ di k caratteri può essere considerata come un numero decimale di lunghezza k e, nel caso generale, è possibile assumere che ogni carattere sia una cifra in notazione base d, dove $d = |\Sigma|$.

Dato un pattern $P[1,m]$ si denoti con p il valore decimale P ($p = \sum_{i=1}^{i=m} P[i] \cdot 10^i$) e analogamente, dato un testo $T[1,n]$ si denoti con t_s il valore decimale della sottostringa $T[s+1, s+m]$ di lunghezza m, per ogni $0 \leq s \leq n-m$. Certamente $t_s = p$ se e solo se $P[1,m] = T[s+1, s+m]$ e s è uno spostamento valido se e solo se $t_s = p$.

Ipotizzando di poter calcolare p in tempo $O(m)$ e tutti i valori t_i in tempo $O(n)$, tutti gli spostamenti validi s potrebbero essere calcolati in tempo $O(n)$ semplicemente confrontando p con ogni t_s. Questo è un risultato eccessivamente ottimistico, ma sfruttando la *regola di Horner*, è possibile dimostrare che lo stesso può essere effettuato in $O(n+m)$.

La *regola di Horner* [5], a volte chiamata anche *algoritmo di Horner*, è una formula che permette la valutazione di un polinomio $P_n(x) = x^n + a_1 x^{n-1} + \cdots + a_{n-1}x + a_n$ di grado n, svolgendo solamente n addizioni ed n moltiplicazioni,

anziché le n addizzioni e $\frac{n(n+1)}{2}$ moltiplicazioni richieste con il metodo di valutazione tradizionale.

Il valore di p può essere calcolato in tempo $O(m)$ nel seguente modo:

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1]) \dots))$$

e lo stesso può essere fatto per calcolare il valore di t_0 a partire da $T[1, m]$. I rimanenti valori $t_1, t_2, \dots, t_{n-m}$, invece, possono essere calcolati in tempo $O(n-m)$ osservando che ogni valore t_{s+1} è calcolabile a partire da t_s utilizzando ricorsivamente la formula:

$$t_{s+1} = (t_s - 10^{m-1}T[s+1]) \cdot 10 + T[s+m+1]$$

è infatti possibile ottenere il t_{s+1}-esimo valore a partire da t_s semplicemente rimuovendo la cifra più significativa con $10^{m-1}T[s+1]$ e aggiungendo a destra la cifra meno significativa con $T[s+m+1]$.

Considerando di calcolare una sola volta 10^{m-1} in tempo $O(m)$, ogni esecuzione dell'equazione sopra descritta richiede un numero costante di operazioni aritmetiche e perciò i vari $p, t_0, t_1, \dots, t_{n-m}$ possono essere calcolati tutti con un tempo complessivo $O(n+m)$.

Poiché i valori di p e t_s potrebbero essere troppo grandi per essere calcolati in modo efficiente con il sistema appena descritto è utile introdurre un concetto di teoria dei numeri, che prende il nome di *classi di congruenza modulo q* [6] così definito:

Dati $a, q \in \mathbb{Z}$ si chiama classe di congruenza a modulo q l'insieme

$$[a]_q = \{x \in \mathbb{Z} \mid x \equiv a \bmod q\}$$

dove per questioni legate alla volontà di far eseguire alla macchina tutti i calcoli con precisione aritmetica, dato un alfabeto di cardinalità d, q viene scelto in modo che il valore dq sia rappresentabile interamente in una parola della macchina.

Utilizzando le classi di congruenza, per un opportuno q è quindi possibile ricalcolare sia il valore p, che tutti i valori t_s, in nuovi valori modulo q pronti per essere tra loro confrontati. La funzione utilizzata per il calcolo dei vari t_s diventa:

$$t_{s+1} = ((t_s - T[s+1] \cdot h) \cdot d + T[s+m+1]) \bmod q$$

dove $h \equiv d^{m-1} \pmod q$ è il valore della cifra 1 nella posizione più significativa di una porzione di testo di m cifre.

La Figura 1.13 mostra un esempio di esecuzione della funzione appena descritta per il calcolo di un generico valore t_s a partire dal precedente valore t_{s-1}:

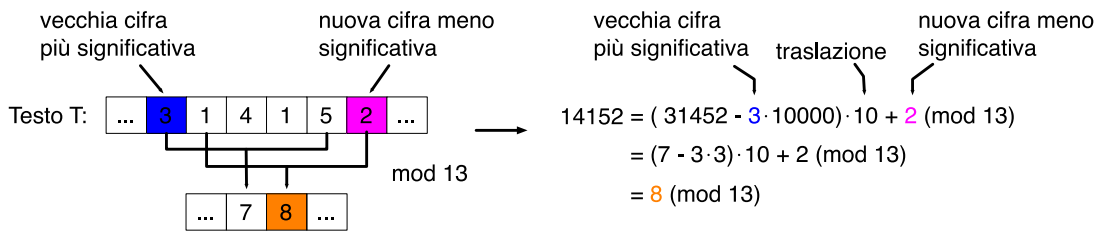


Figura 1.13: Esempio del procedimento di calcolo del valore t_{s+1}

Per utilizzare efficacemente i valori di p e t_s modulo q è necessario fare attenzione ad un dettaglio che non può essere trascurato: se da un lato si può affermare con certezza che se $t_s \not\equiv p \pmod q$, allora sicuramente $t_s \neq p$, lo stesso non si può dire del contrario, $t_s \equiv p \pmod q$ non implica che $t_s = p$. A questo punto, mentre il controllo $t_s \not\equiv p \pmod q$ viene utilizzato dall'algoritmo come veloce controllo euristico per scartare gli spostamenti non validi, per ogni spostamento s tale che $t_s = p$ le due stringhe $P[1,m]$ e $T[s+1, s+m]$ devono essere controllate esplicitamente alla ricerca del match tra tutti i caratteri che le compongono.

L'esempio in Figura 1.14 mostra i confronti effettuati dall'algoritmo di Rabin-Karp su un testo T e un pattern P in cui p e i vari t_s si riferiscono a classi di congruenza modulo 13; è possibile osservare come la sottosequenza indicata in rosso sia un falso positivo, essa ha lo stesso valore modulo 13 di quella in verde, senza però combaciare con il pattern.

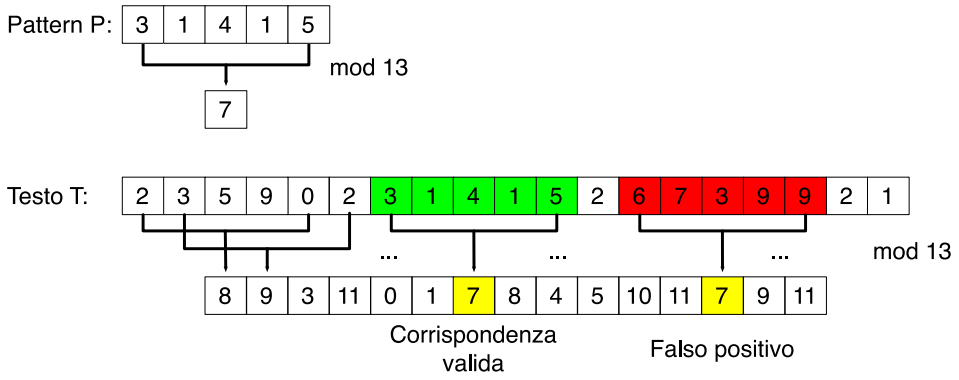


Figura 1.14: Esempio di esecuzione dell'algoritmo Rabin-Karp

Lo pseudocodice dell'algoritmo è il seguente:

```
Rabin_Karp(P,T,d,q)
      n ←length(T)
      m ← length(P)
      h ← d^(m-1) mod q
      p ← 0
      t0 ← 0
      for i ← 0 to m
            p ←(dp + P[i]) mod q
            t0 ← (dt0 + T[i]) mod q
      for s ← 0 to n-m
            if (p == ts)
                  if (P[1,m] == T[s+1,s+m])
                        print "Il pattern appare con
                        spostamento" s
            if (s < n - m)
                  ts+1 ← (ts - T[s+1]·h)·d + T[s+m+1]) mod q
```

L'algoritmo assume che tutti i caratteri siano rappresentati come cifre in base d; le prime due istruzioni semplicemente assegnano alle variabili n ed m rispettivamente

la lunghezza del testo T e del pattern P in tempo costante, mentre con la terza si inizializza il valore h con il valore della cifra più significativa di una sottostringa di T di m cifre. Il calcolo di d^{m-1} impiega tempo $O(m)$.

Il primo ciclo calcola p come il valore di $P[1,m] \bmod q$ e t_0 come il valore di $T[1,m] \bmod q$ in tempo $O(m)$, mentre il secondo si occupa di calcolare tutti i successivi $n-m-1$ valori t_s e segnalare l'eventuale corrispondenza con il pattern P; ogni qualvolta $p = t_s$, per evitare falsi positivi, si verificano esplicitamente tutti i caratteri delle stringhe $P[1,m]$ e $T[s+1, s+m]$. Poiché il valore di d^{m-1} viene calcolato solo una volta all'inizio della procedura, il tempo necessario a calcolare ognuno dei possibili t_s mediante la formula $t_{s+1} = ((t_s - T[s+1] \cdot h) \cdot d + T[s+m+1]) \bmod q$ è costante e impiega quindi tempo $O(n-m)$, mentre il confronto tra le due stringhe nel caso p uguale a t_s, avviene in tempo $O(m)$.

L'algoritmo di Rabin-Karp, come quello ingenuo, verifica esplicitamente ognuno dei possibili spostamenti validi riscontrati. Se $P = a^m$ e $T = a^n$, poiché ognuno degli $n-m+1$ spostamenti è valido, le verifiche richiedono tempo $\Theta((n-m+1) \cdot m)$. La complessità dell'algoritmo è $\Theta((n-m+1) \cdot m)$ nel caso peggiore.

1.4 Algoritmo di Boyer-Moore

L'algoritmo di Boyer-Moore [7], presentato da Bob Boyer e J. Strother Moore nel 1977, è come tutti gli altri diviso in una prima fase iniziale di pre-elaborazione e una seconda fase di ricerca. Le caratteristiche principali di questo algoritmo, che lo distinguono dai precedenti, sono:

- Analisi del pattern da destra verso sinistra (*right-to-left scan*)
- Due fasi di pre-elaborazione del pattern parallele e indipendenti
    - Euristica del buon suffisso (*good-suffix rule*)
    - Euristica del carattere discordante (*bad-character rule*)

Come quello di Knuth, Morris e Pratt, anche l'algoritmo di Boyer-Moore è stato negli anni punto di riferimento per la creazione di nuovi algoritmi o comunque varianti di quelli già esistenti. Nel *Capitolo 2* sono presentati alcuni dei più noti algoritmi della cosiddetta "famiglia Boyer-Moore"; alcuni di essi sono considerati al giorno d'oggi come i più efficienti nella pratica e nella maggior parte dei casi sfruttano la potenza e le caratteristiche delle funzioni euristiche sopra citate.

I paragrafi che seguono, descrivono l'algoritmo di Boyer-Moore originale, nelle due fasi di pre-elaborazione del pattern e ricerca del testo. Prima di iniziare è però utile spendere qualche parola sulla descrizione di come il pattern viene analizzato.

1.4.1 Right-to-left scan

A differenza degli algoritmi visti finora, quello di Boyer-Moore utilizza una tecnica curiosa di verifica del pattern, esplorando i caratteri da destra verso sinistra, partendo dall'elemento in posizione $P[m]$, per poi arrivare a quello in posizione $P[1]$; il confronto viene effettuato con i caratteri del testo che sono nella finestra corrente. Trovata una discordanza (o mismatch) tra un carattere del pattern ed uno del testo alla posizione j, viene restituito il valore $j + 1$ dell'ultima posizione in cui i caratteri del pattern e del testo erano tutti tra loro coincidenti (Figura 1.15).

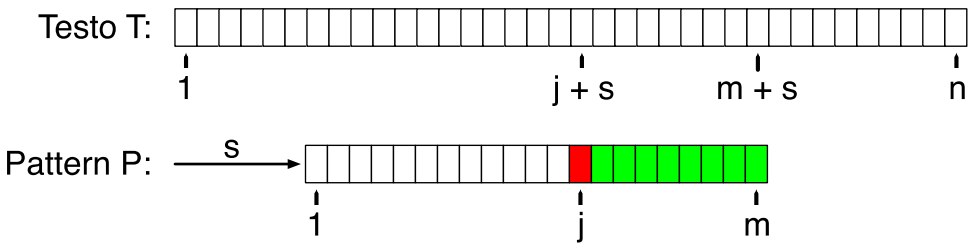


Figura 1.15: Esempio di scorrimento "right-to-left"

Lo spostamento del pattern non è comunque costante, le due funzioni euristiche sopra citate consentono di evitare gran parte del lavoro svolto negli algoritmi precedenti permettendo all'algoritmo di Boyer-Moore di saltare un gran numero di caratteri in una sola volta, utilizzando valori di s variabili e non sempre unitari.

1.4.2 Pre-elaborazione del pattern: le due euristiche

Le due euristiche, conosciute con i nomi di *euristica del carattere discordante* (*bad-character rule*) e *euristica del buon suffisso* (*good-suffix rule*), possono essere viste come operanti in parallelo, ma indipendenti l'una dall'altra. Quando si verifica un mismatch, ogni euristica propone una quantità secondo cui la variabile s può essere incrementata con sicurezza, senza saltare nessuna posizione valida. L'algoritmo Boyer-Moore sceglie la quantità più grande e incrementa lo spostamento s di quel valore.

La sequenza di esempi mostrati in Figura 1.16, Figura 1.17 e Figura 1.18, illustra l'utilizzo delle due euristiche su un testo $T = \{cbrtoctersbadcharacterg \ldots\}$ e pattern $P = \{character\}$:

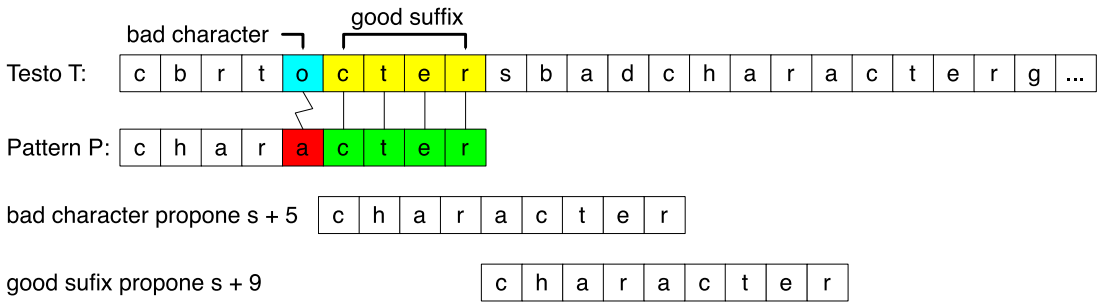


Figura 1.16: Esempio di funzionamento delle euristiche "bad character" e "good suffix" – 1

Il pattern P è inizialmente allineato alla posizione 1 del testo e, come descritto, viene esplorato da destra verso sinistra. Le due euristiche sono evidenziate rispettivamente di colore celeste per la *bad-character rule* e giallo per la *good-suffix rule*. Sebbene un "buon suffisso" sia stato trovato tra gli ultimi quattro caratteri, lo spostamento $s = 0$ non è valido, a confermarlo è la presenza di un carattere discordante in posizione $i = 5$.

L'*euristica del carattere discordante* propone di spostare il pattern a destra, se possibile, di una quantità che garantisca al carattere del testo reo di aver causato mismatch, di combaciare con l'occorrenza più a destra dello stesso carattere nel pattern. Nell'esempio di Figura 1.16 il carattere discordante "*o*" del testo non compare nel pattern; in questo caso P può essere spostato del tutto dopo il carattere discordante del testo.

L'*euristica del buon suffisso*, invece, prevede che se nel pattern è presente un'altra sequenza dei caratteri, uguale a quella che ha determinato il "buon suffisso" (in questo caso "*c t e r*"), il pattern viene spostato a destra della quantità minima atta a garantire che questa nuova sequenza coincida con il "buon suffisso" all'interno del testo. Poiché nell'esempio corrente non è presente nessun'altra occorrenza della sottostringa "*c t e r*", il pattern viene spostato interamente alla destra dell'ultimo carattere indicato in giallo nel testo.

Per gestire lo spostamento, l'algoritmo Boyer-Moore sceglie la quantità più grande tra quelle proposte dalle due euristiche e incrementa lo spostamento s di

quella quantità; nell'esempio corrente di Figura 1.16, l'*euristica del carattere discordante* propone uno spostamento $s = s + 5$, mentre quella *del buon suffisso* uno spostamento $s = s + 9$. Quest'ultima viene prese in considerazione e la situazione diventa quella mostra in Figura 1.17.

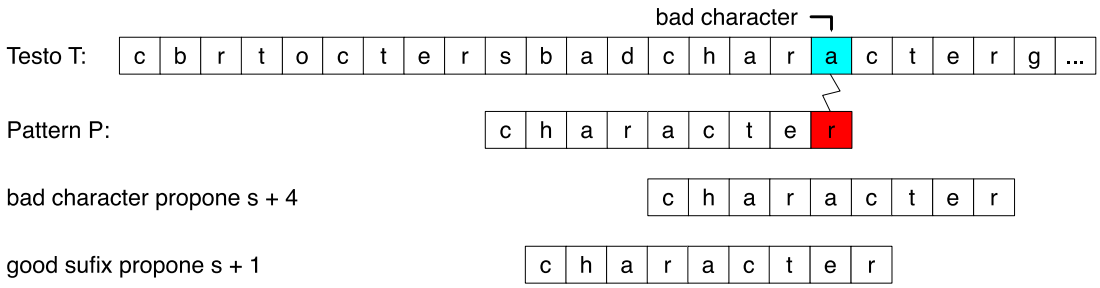


Figura 1.17: Esempio di funzionamento delle euristiche "bad character" e "good suffix" – 2

In questa condizione, poiché il pattern è esplorato da destra verso sinistra, e poiché il primo carattere analizzato è proprio un carattere discordante, l'*euristica del buon suffisso* propone uno spostamento unitario a destra, quindi con $s = s + 1$, mentre l'*euristica del carattere discordante* propone uno spostamento $s = s + 4$ al fine di creare un match tra il carattere discordante "*a*" del testo e la prima occorrenza, a partire da destra, dello stesso carattere all'interno del pattern.

L'*euristica del carattere discordante* vince il confronto e s viene incrementato di una quantità $s = s + 4$. La situazione che si viene a creare è mostrata in Figura 1.18 e consente di concludere che lo spostamento $s = 13$ è uno spostamento valido per P.

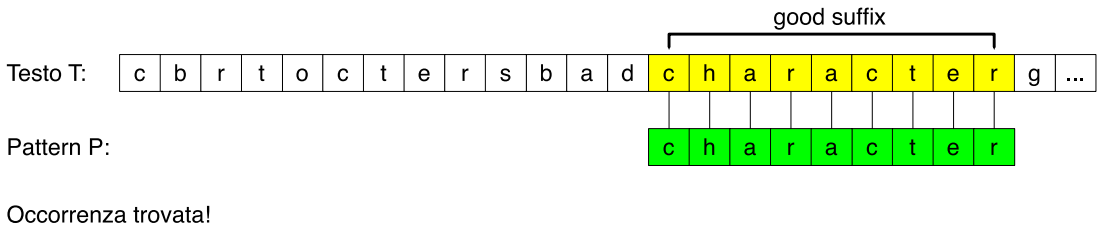


Figura 1.18: Esempio di funzionamento delle euristiche "bad character" e "good suffix" – 3

Nei due paragrafi seguenti sono descritti gli algoritmi alla base delle due euristiche, algoritmi che saranno poi utilizzati dall'algoritmo di Boyer-Moore per la fase di ricerca del pattern all'interno del testo.

1.4.2.3 Euristica del carattere discordante (bad-character rule)

In presenza di un mismatch, l'*euristica del carattere discordante* utilizza le informazioni circa la posizione del carattere discordante del testo all'interno del pattern per proporre un nuovo spostamento. Nel caso migliore, la discordanza si verifica sul primo confronto ($P[m] \neq T[s+m]$); in questo caso, poiché qualunque spostamento più piccolo di $s+m$ permetterebbe l'allineamneto di qualche carattere del pattern con il carattere discordante, lo spostamento proposto dall'euristica coinvolge l'intera lunghezza del pattern.

In generale l'*euristica del carattere discordante* funziona come segue: si supponga di aver appena trovato una discordanza $P[j] \neq T[s+j]$ per qualche j compreso tra 1 e m. Sia allora k l'indice del più grande intervallo $1 \leq k \leq m$ tale che $T[s+j] = P[k]$ se un tale k esiste, altrimenti $k = 0$. Ad ogni iterazione, lo spostamento s può essere incrementato con successo di $j-k$; si considerino i seguenti tre casi:

- $k = 0$: il carattere discordante $T[s+j]$ non compare mai all'interno del pattern; s può essere incrementato di una quantità j senza il pericolo di saltare alcuno spostamento valido (Figura 1.19).

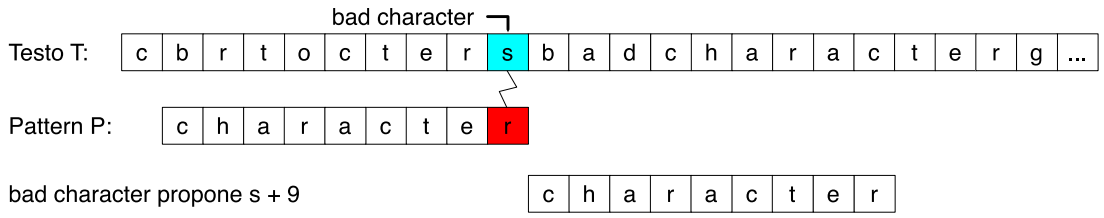


Figura 1.19: Esempio di funzionamento dell'euristica "bad character": caso k = 0

- $k < j$: il carattere discordante $T[s + j]$ compare all'interno del pattern, a sinistra della posizione j, così che $j - k > 0$. Il pattern può essere spostato di $j - k$ caratteri al fine di far corrispondere il carattere discordante del testo con lo stesso carattere all'interno del pattern (Figura 1.20).

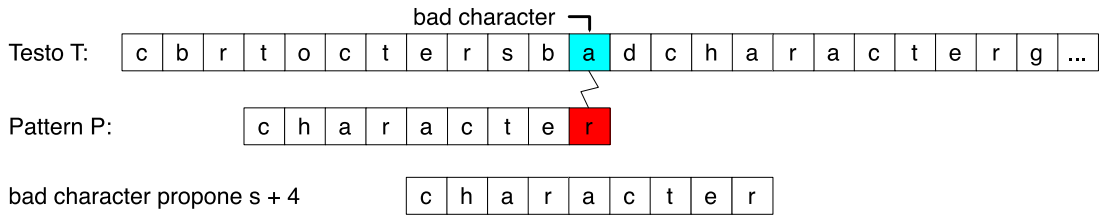


Figura 1.20: Esempio di funzionamento dell'euristica "bad character": caso k < j

- $k > j$: il carattere discordante $T[s + j]$ compare all'interno del pattern, ma a destra della posizione j; in questo caso $j - k < 0$ e la funzione euristica propone un decremento dello spostamento s. Questa proposta viene ignorata dall'algoritmo (Figura 1.21).

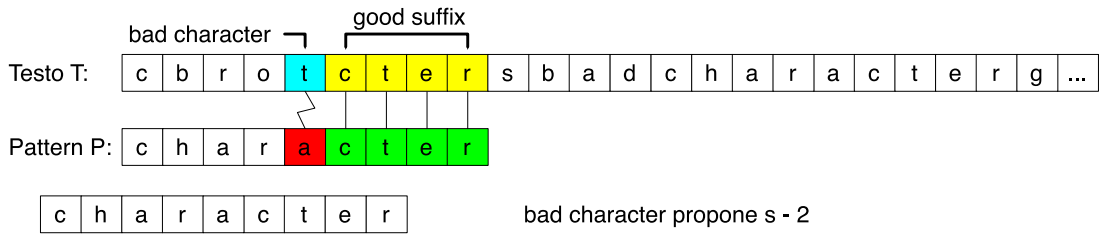


Figura 1.21: Esempio di funzionamento dell'euristica "bad-character": caso k > j

L'*euristica del carattere discordante* è implementata nell'algoritmo di Boyer-Moore, mediante la cosiddetta *funzione dell'ultima occorrenza*. Tale funzione definisce $\lambda[a]$ come l'indice della posizione più a destra, all'interno del pattern, in cui appare il carattere a, per ogni $a \in \Sigma$. Se a non appartiene al pattern, allora $\lambda[a] = 0$.

Lo pseudocodice dell'algoritmo per il calcolo della *funzione dell'ultima occorrenza* è il seguente:

```
Funzione_ultima_occorrenza(P,Σ)
      for ogni carattere a ∈ Σ
            λ[a] ← 0
      for j ← 1 to m
            λ[P[j]] ← j
      return λ
```

Il tempo di esecuzione della procedura *Funzione_ultima_occorrenza()* è dato dalla somma dei tempi dei due cicli *for* ed è quindi $O(|\Sigma| + m)$.

1.4.2.4 Euristica del buon suffisso (good-suffix rule)

I controlli dei caratteri del pattern da destra verso sinistra, permettono di ricavare importanti informazioni circa il numero di caratteri che coincidono con la finestra di testo corrente. Trovato il primo mismatch, si controlla se nel pattern esistono altre sottostringhe che hanno gli stessi caratteri di quelli appena verificati e che fanno match con il testo. Se esistono, l'*euristica del buon suffisso* suggerisce di scorrere il pattern a destra fino ad allineare la sottostringa selezionata con i caratteri del testo "*del buon suffisso*"; la scelta della sottostringa più a destra è dovuta dalla necessità di evitare di saltare eventuali spostamenti validi.

L'*euristica del buon suffisso*, è implementata nell'algoritmo di Boyer-Moore, mediante la cosiddetta *funzione del buon suffisso*. Tale funzione definisce $\gamma[j]$ come la quantità minima di cui lo spostamento s può avanzare in modo che nessun carattere precedentemente etichettato come "buon suffisso" sia discorde dal nuovo allineamento del pattern.

In caso di mismatch in posizione j, $P[j] \neq T[s+j]$ e la *funzione del buon suffisso* calcola l'incremento dello spostamento s mediante la formula

$$\gamma[j] = m - \max\{\, k \mid 0 \leq k < m \text{ e } P[j+1, m] \sim P[1, k] \,\}$$

Il simbolo $\sim$ si legge "simile" e significa che $P[j+1, m] \sqsupset P[1, k]$ oppure $P[1, k] \sqsupset P[j+1, m]$, ovvero che $P[j+1, m]$ è suffiso di $P[1, k]$ oppure $P[1, k]$ è suffisso di $P[j+1, m]$.

Poiché il suffisso $P[j+1, m]$ del pattern, coincide con la sottosequenza $T[s+j+1, s+m]$ del testo, la *funzione del buon suffisso* restituisce il minimo spostamento $s' - s$ tale che $P[k+1, k+m-j] = T[s+j+1, s+m]$ con $s = s' + k - j$.

I casi che possono verificarsi sono tre e sono qui elencati nello stesso ordine in cui vengono verificati:

- Se nel pattern è presente un'ulteriore sottostringa, coincidente con $T[s+j+1, s+m]$, composta cioè dagli stessi caratteri "*del buon suffisso*", ma preceduta da un carattere diverso da quello che ha causato mismatch ($P[j]$), lo spostamento suggerito è uguale al numero minimo di passi necessari per far coincidere la nuova sottostringa di P con il suffisso di T considerato (Figura 1.22)

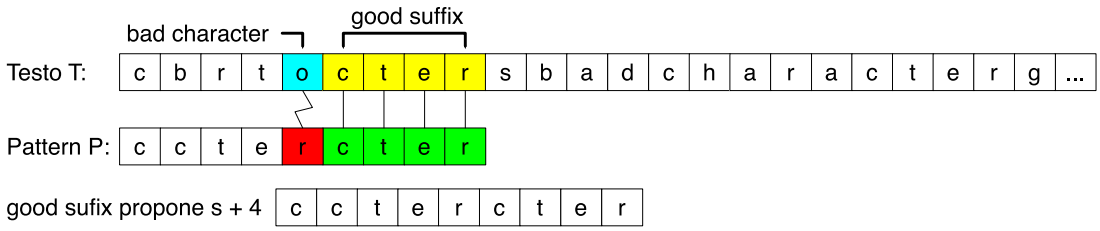


Figura 1.22: Esempio di funzionamento dell'euristica "good-suffix": caso 1

- Se esiste un prefisso v di P che è anche suffisso di $T[s+j+1, s+m]$, la funzione suggerisce uno spostamento $\gamma[j] = m - |v|$ tale da far

coincidere il prefisso v con il suffisso di $T[s+j+1, s+m]$ (Figura 1.23).

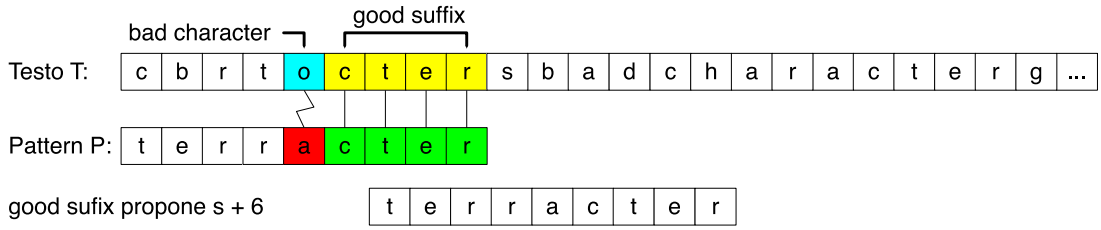


Figura 1.23: Esempio di funzionamento dell'euristica "good-suffix": caso 2

- Se nel pattern P non occorre nessun'altra sottostringa che è anche un suffisso di $T[s+j+1, s+m]$. Lo spostamento proposto è pari ad m e quindi $\gamma[j] = m$ (Figura 1.24).

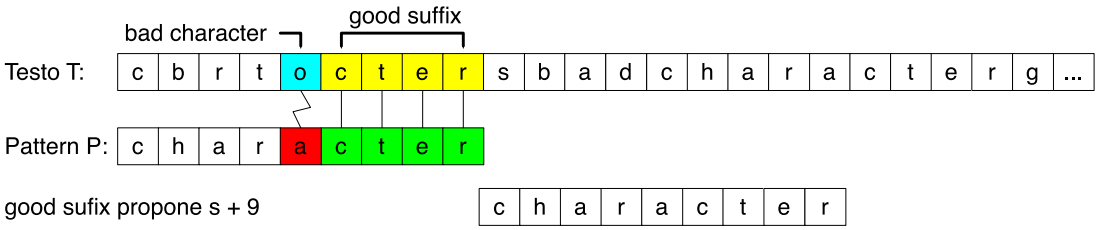


Figura 1.24: Esempio di funzionamento dell'euristica "good-suffix": caso 3

Di seguito lo pseudocodice dell'algoritmo per il calcolo della *funzione del buon suffisso*:

```
Funzione_buon_suffisso(P)
      m ← length(P)
      π ← Funzione_prefisso(P)
      P_rev ← reverse(P)
      π_rev ← Funzione_prefisso(P_rev)
      for j ← 0 to m
            γ[j] ← m - π[m]
      for k ← 1 to m
            j ← m - π_rev[k]
            if (γ[j] > (k - π_rev[k]))
                  γ[j] ← k - π_rev[k]
      return γ
```

La *funzione prefisso*, qui definita come *la lunghezza del più lungo prefisso di P che sia anche suffisso di* $P[1,q]$ secondo la formula

$$\pi[i] = \max\{\, k \mid \mathrm{k} < q \; e \; P[1,k] \sqsupset P[1,q] \,\}$$

viene usata per inizializzare i due array π e π_{rev} rispettivamente utilizzando in input il pattern P e il suo inverso P_{rev}. Il primo ciclo inizializza l'array γ con la differenza tra la lunghezza del pattern P e il valore restituito dalla *funzione prefisso* per $P[1,m]$, mentre il secondo ricerca eventuali spostamenti inferiori a quelli appena trovati e riferiti ai tre casi precedentemente descritti per il calcolo di $\gamma[j]$.

L'intera procedura richiede tempo $O(m)$.

1.4.3 Ricerca nel testo: descrizione dell'algoritmo di Boyer-Moore

La fase di ricerca consiste semplicemente nel far scivolare il pattern all'interno del testo, sfruttando gli spostamenti suggeriti dalle due euristiche e, in caso di match, segnalare lo spostamento valido trovato.

Rispetto a molti altri algoritmi, quello di Boyer-Moore risulta molto veloce nel caso di un alfabeto molto vasto; le due procedure di pre-processing consentono di evitare un gran numero di controlli e far scivolare velocemente il pattern all'interno del testo. A differenza di quello di Knuth, Morris e Pratt, in cui lo spostamento dal pattern non è mai superiore al numero di caratteri confrontati nella posizione attuale, nell'algoritmo di Boyer e Moore lo spostamento può essere maggiore del numero di confronti effettuati in tale posizione. Nella maggior parte dei casi, e soprattutto per pattern lunghi, l'algoritmo di Boyer e Moore richiede un numero totale di confronti che è soltanto una frazione del numero di caratteri del testo.

```
Boyer_Moore(P,T,Σ)
      n ←length(T)
      m ← length(P)
      λ ← Funzione_ultima_occorrenza(P,Σ)
      γ ← Funzione_buon_suffisso(P)
      s ← 0
      while (s ≤ n - m)
            j ←m
            while (j > 0 and P[j] == T[s+j])
                  j ←j - 1
            if (j == 0)
                  print "Il pattern appare con spostamento" s
                  s ←s + γ[0]
            else
                  s ←s + max(γ[j], j - λ[T[s+j]])
```

La complessità della fase di *pre-processig*, dovuta all'*euristica del carattere discordante* e all'*euristica del buon suffisso* è $O(|\Sigma| + m) + O(m) = O(|\Sigma| + m)$, mentre quella della fase di ricerca è $O(n - m + 1)m$.

Il tempo di esecuzione dell'algoritmo Boyer-Moore è pertanto $O((n - m + 1)m + |\Sigma|)$ e quindi questa volta è influenzato anche dalla dimensione dell'alfabeto.

Capitolo 2

La famiglia di algoritmi Boyer-Moore

Quello che segue è un capitolo di approfondimento dedicato alla cosiddetta "famiglia di algoritmi Boyer-Moore", ovvero quegli algoritmi che traggono da questo ispirazione e che utilizzano le funzioni euristiche *del carattere discordante* e *del buon suffisso* per diminuire il numero di confronti e migliorare l'efficienza delle procedure. Per ognuno di essi è descritta brevemente l'idea alla base dell'algoritmo, è presentato lo pseudocodice della fase di ricerca ed eventualmente, se diversa da quelle già viste, anche della fase di pre-elaborazione, è accennata la complessità nel caso peggiore e, per un testo ed un pattern comune a tutti gli algoritmi trattati, è illustrato un esempio di funzionamento in cui sono indicati eventuali match, mismatch e spostamenti effettuati.

Ognuno degli algoritmi di questo capitolo presenta caratteristiche e prestazioni differenti, alcuni sono molto semplici, altri più complessi; alcuni di essi sono anche considerati al giorno d'oggi come i più efficienti nella pratica, nonostante tempi di esecuzione ben più alti di quelli precedentemente incontrati.

2.1 Algoritmo di Horspool

L'algoritmo di Horspool [8] è il primo esempio di variante dell'algoritmo di Boyer-Moore che sfrutta una delle due euristiche per ottimizzare quella che sotto alcuni punti di vista può essere un punto di debolezza dell'algoritmo. L'euristica del carattere discordante, utilizzato nell'algoritmo di Boyer-Moore, non è infatti molto efficiente per piccoli alfabeti, bensì quando l'alfabeto è grande in confronto alla lunghezza del pattern. Horspool propone per questo di utilizzare l'*euristica del carattere discordante* in maniera indipendente, applicando la funzione λ solamente sul carattere più a destra del pattern per calcolare gli spostamenti nell'algoritmo di Boyer-Moore.

La fase di pre-elaborazione di questa implementazione, rimane ovviamente calcolabile in tempo $O(m + |\Sigma|)$, mentre la fase di ricerca ha una complessità quadratica nel caso peggiore.

```
Horspool(P,T)
     n ←length(T)
     m ← length(P)
     k ← 0
     λ ← Funzione_ultima_occorrenza(P,Σ)
     j ← 1
     while (j < n - m)
          c ← T[j + m]
          if ((P[m] == c) and (P[1,m] == T[j+1,j+m]))
               print "Il pattern appare con spostamento" j
          j ← j + λ[c]
```

Esempio di esecuzione:

Testo: $T = \{gcatcgcagagagtataca\}$

Pattern: $P = \{gcagagag\}$

Pre-processing:

$\boldsymbol{\Sigma}$	a	c	g	t
$\boldsymbol{\lambda}$	1	6	2	8

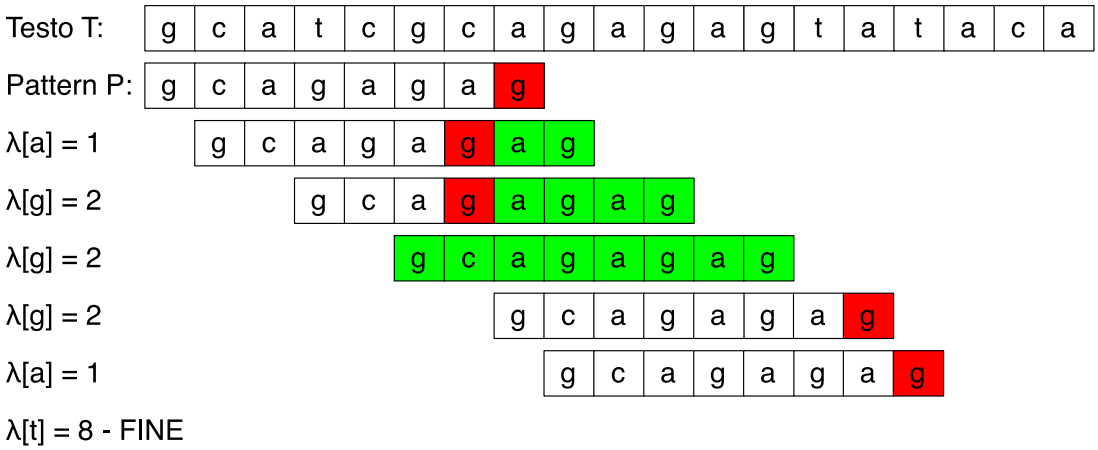


Figura 2.1: Esempio di esecuzione dell'algoritmo di Horspool

2.2 Algoritmo Quick-Search

L'algoritmo Quick-Search sviluppato da D. M. Sunday [9], è una semplificazione dell'algoritmo di Boyer-Moore e anch'esso utilizza i valori dell'*euristica del carattere discordante*.

L'algoritmo si basa sulla seguente osservazione: dopo il confronto tra un generico pattern $P[1, m]$ e la sottostringa $T[s + 1, s + m]$ del testo, il valore dello spostamento viene sempre incrementato di almeno un carattere, ma mai più di m. Il carattere $T[s + m + 1]$ è quindi necessariamente coinvolto nel tentativo successivo. Applicando l'*euristica del carattere discordante* al carattere $T[s + m + 1]$ anziché a quello che ha causato mismatch è quindi possibile ottenere un'incremento dello spostamento maggiore.

Ad ogni passo, lo spostamento s è dato dal valore di $\lambda_{Quick\_Search}[T[s + m + 1]]$, dove l'array λ_{Quick_Search} è così calcolato:

$$\forall a \in \Sigma,\ \ \lambda_{Quick\_Search}[a] = \min(\{\, 1 \leq k \leq m :\ P[m - k] = a\} \cup \{m\})$$

Lo pseudocodice di questa versione modificata del carattere discordante è il seguente:

```
Funzione_ultima_occorrenza_quick_search(P,Σ)
    for k ← 1 to |Σ|
        λ[k] ← m
    for k ← 1 to m
        λ[P[k]] ← m - 1
    return λ
```

La fase di pre-processing ha una complessità temporale pari a $O(m + |\Sigma|)$, mentre quella di ricerca è quadratica nel caso peggiore. Una curiosità dell'algoritmo Quick-

Search è che durante la fase di ricerca i confronti fra i caratteri del pattern e del testo durante ciascun tentativo possono essere fatti in qualsiasi ordine.

```
Quick_Search(P,T)
    n ←length(T)
    m ← length(P)
    λQuick-Search ← Funzione_ultima_occorrenzaquick_search(P,Σ)
    for s ← 1 to n - m
        if (P[s,m] == T[s+1,s+m])
            print "Il pattern appare con spostamento" s
        s ← s + λQuick-Search[T[j+m]]
```

Esempio di esecuzione:

Testo: $T = \{gcatcgcagagagtataca\}$

Pattern: $P = \{gcagagag\}$

Pre-processing:

Σ	a	c	g	t
λ_{Quick_Search}	2	7	1	9

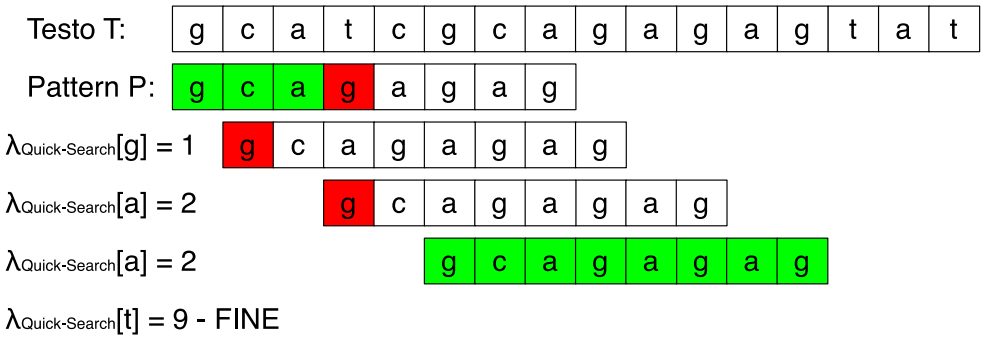


Figura 2.2: Esempio di esecuzione dell'algoritmo Quick-Search

2.3 Algoritmo di Smith

L'algoritmo di Smith [10], nasce da un'osservazione fatta dall'autore circa i valori di spostamento restituiti dall'*euristica del carattere discordante* dell'algoritmo di Boyer-Moore e dell'algoritmo Quick-Search in funzione del carattere considerato. Più precisamente Smith ha osservato che non sempre calcolare lo spostamento sul carattere $T[s+m+1]$ (Algoritmo Quick-Search) fornisce un valore maggiore rispetto allo stesso calcolo sul carattere $T[s+m]$ (Algoritmo di Boyer-Moore).

La fase di pre-processing dell'algoritmo di Smith consiste nel calcolare sia la funzione di spostamento dell'*euristica del carattere discordante* proposta dall'algoritmo di Boyer-Moore, sia quella implementata nell'algoritmo Quick-Search; dei due risultati viene poi scelto quello maggiore.

La complessità dell'algoritmo è $O(m+|\Sigma|)$ per quanto riguarda la fase di pre-elaborazione del pattern e quadratica nel caso peggiore per la fase di ricerca nel testo.

```
Quick_Search(P,T)
    n ←length(T)
    m ← length(P)
    λ ← Funzione_ultima_occorrenza(P,Σ)
    λQuick-Search ← Funzione_ultima_occorrenzaquick_search(P,Σ)
    for j ← 1 to n - m
        if (P[1,m] == T[j+1,j+m])
            print "Il pattern appare con spostamento" j
        j ← j + max( λ[T[j+m-1]], λQuick-Search[T[j+m]] )
```

Esempio di esecuzione:

Testo: $T = \{gcatcgcagagagtataca\}$

Pattern: $P = \{gcagagag\}$

Pre-processing:

Σ	a	c	g	t
λ	1	6	2	8
λ_{Quick_Search}	2	7	1	9

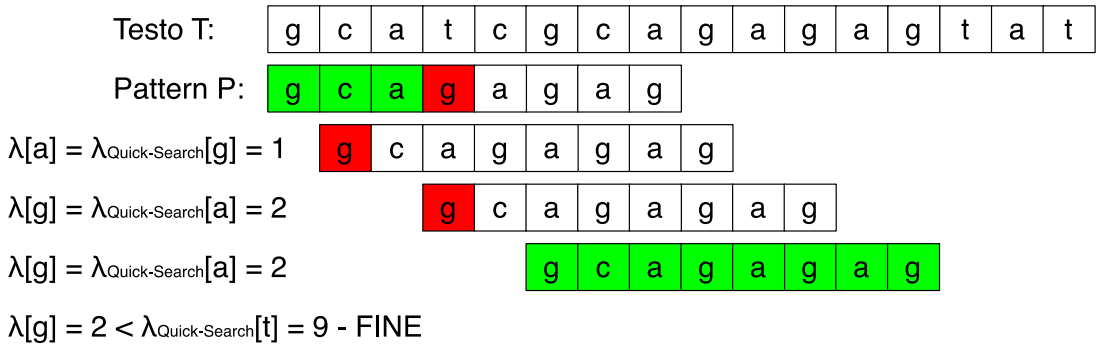


Figura 2.3: Esempio di esecuzione dell'algoritmo di Smith

2.4 Algoritmo Tuned Boyer-Moore

L'algoritmo Tuned Boyer-Moore [11], creato da A. Hume A e D. M. Sunday (lo stesso autore dell'algoritmo Quick-Search), basa la sua forza sull'osservazione che la parte più costosa di un algoritmo di *string-matching* è quella di verifica dei singoli caratteri del pattern all'interno del testo. Per tentare di ridurre il numero di controlli l'algoritmo prevede di effettuare diversi spostamenti successivi prima di verificare l'intero pattern.

Ogni iterazione dell'algoritmo Tuned Boyer-Moore può essere divisa in due fasi:

- Localizzazione dell'ultimo carattere
- Confronto del pattern

La prima fase sfrutta l'*euristica del carattere discordante* per localizzare l'occorrenza dell'ultimo carattere del pattern nel testo (quello in posizione $P[m]$) eseguendo tre spostamenti "ciechi" alla volta, finché necessario. Per consentire quest'operazione il valore di $\lambda[P[m]]$ viene copiato in una variabile *shift* (da utilizzare nel seguito) e successivamente posto uguale a 0.

La seconda fase, invece, verifica semplicemente, da sinistra verso destra, la corrispondenza tra i rimanenti $m - 1$ caratteri del pattern e la sottosequenza di testo considerata. Terminata la fase di matching, si effettua lo spostamento della quantità *shift* precedentemente salvata. Per assicurare la correttezza del calcolo degli spostamenti, in una fase preliminare dell'algoritmo, il testo viene esteso con m copie del carattere $P[m]$ utilizzate come *sentinella*.

È da notare che la scelta di effettuare i tre spostamenti ciechi sopra descritti, è una scelta puramente empirica decisa dall'autore e che permette all'algoritmo di aver un buon comportamento nella pratica, nonostante la sua complessità quadratica nel caso peggiore.

```
Tuned_Boyer_Morre(P,T)
     m ← length(P)
     T ← TP[m]^m
     n ←length(T)
     λ ← Funzione_ultima_occorrenza(P,Σ)
     shift ← λ[P[m]]
      λ[P[m]] ← 0
     for j ← 1 to n
          k ← λ[T[j+m]]
          while ( k ≠ 0)
               for i ← 1 to 3
                    j ← j + k
                    k ← λ[T[j+m]]
          if (P[1,m] == T[j+1,j+m])
               print "Il pattern appare con spostamento" j
          j ← j + shift
```

Esempio di esecuzione:

Testo: $T = \{gcatcgcagagagtataca\}$

Pattern: $P = \{gcagagag\}$

Pre-processing:

Σ	a	c	g	t
λ	1	6	0	8

***shift* = 2**

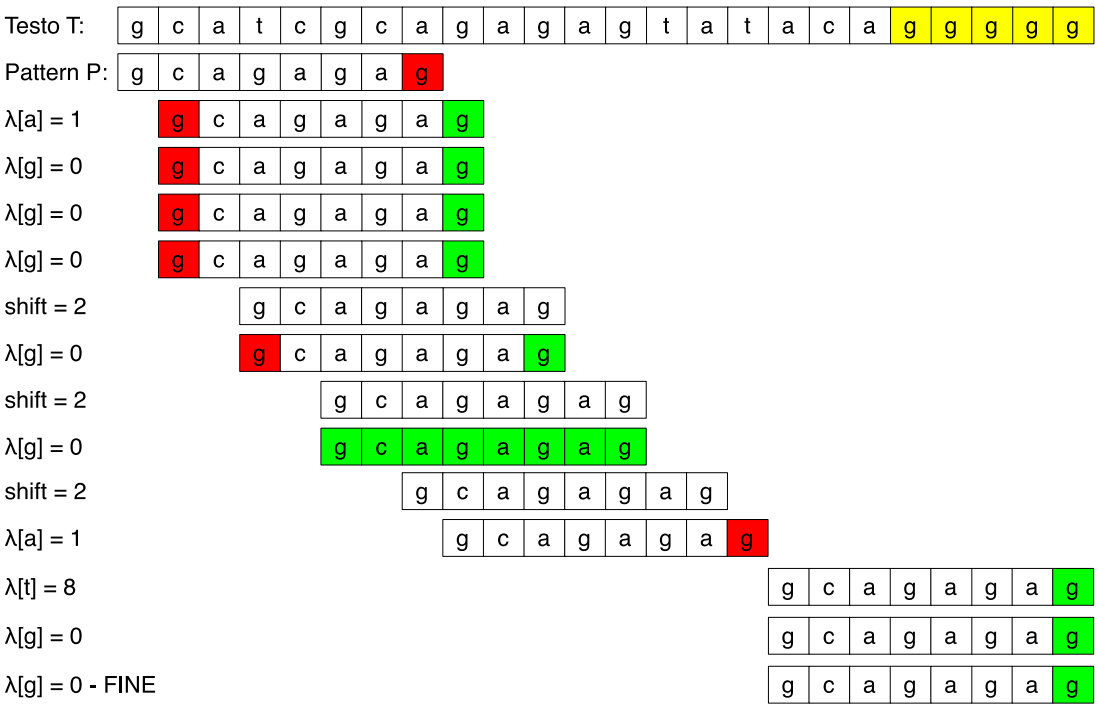


Figura 2.4: Esempio di esecuzione dell'algoritmo Tuned Boyer-Moore

2.5 Algoritmo di Berry-Ravindran

L'algoritmo di Berry-Ravindran [12] è un'estensione di quello del Quick-Search; in questa variante l'*euristica del carattere discordante* non viene applicata al carattere immediatamente successivo all'intervallo considerato, ma alla coppia di caratteri $T[s+m+1]$ e $T[s+m+2]$ successivi. Al termine dello spostamento la sottostringa di testo $T[s+m+1, s+m+2]$ risulta allineata all'occorrenza più a destra in P; su questo è basata la fase di pre-processing dell'algoritmo.

Durante la fase di pre-elaborazione, per ciascuna coppia di caratteri (a, b) con $a, b \in \Sigma$, viene calcolata l'occorrenza più a destra della stringa ab in P. L'array λ diventa in questo algoritmo una matrice $\lambda[a][b]$ così definita:

$$\forall\, a, b \in \Sigma, \qquad \lambda[a][b] = \begin{cases} 1 & \text{se } P[m] = a \\ m - i + 1 & \text{se } P[i]P[i+1] = ab \\ m + 1 & \text{se } P[0] = b \\ m + 2 & \text{altrimenti} \end{cases}$$

```
Funzione_ultima_occorrenza_Berry_Ravidran (P,Σ)
    for ogni carattere a ∈ Σ
        for ogni carattere b ∈ Σ
            λ[a][b] ← m + 2
    for ogni carattere a ∈ Σ
        λ[a][P[0]] ← m + 1
    for i ← 1 to m-1
        λ[P[i]][P[i+1]] ← m - i
    for ogni carattere a ∈ Σ
        λ[P[m]][a] ← 1
    return λ
```

La fase di pre-elaborazione ha complessità $O(m + |\Sigma|^2)$, quella di ricerca $O(mn)$ che nel caso peggiore può arrivare ad essere $O(n^3)$

Sulla base dei valori di λ calcolati, dopo il confronto tra il pattern $P[1, m]$ e la sottostringa di testo $T[j+1, j+m]$, viene eseguito un spostamento di lunghezza

$\lambda[T[j+m], T[j+m+1]]$. Per rispettare le condizioni dell'algoritmo, e applicare la nuova versione dell'euristica del carattere discordante, si assume di aggiungere al testo ulteriori due caratteri speciali, diversi da tutti quelli di Σ, con il solo scopo di permettere anche l'ultimo spostamento previsto.

```
Berry-Ravindran(P,T)
    T ← T##
    n ← length(T)
    m ← length(P)
    λ ← Funzione_ultima_occorrenza_Berry_Ravindran(P,Σ)
    for j ← 1 to n - m
        if ( P[1,m] == T[j+1,j+m])
            print "Il pattern appare con spostamento" j
        j ← j + λ[T[j+m]][T[j+m+1]]
```

Esempio di esecuzione:

Testo: $T = \{gcatcgcagagagtataca\}$

Pattern: $P = \{gcagagag\}$

Pre-processing:

λ	a	c	g	t	#
a	10	10	2	10	10
c	7	10	9	10	10
g	1	1	1	1	1
t	10	10	9	10	10
#	10	10	9	10	10

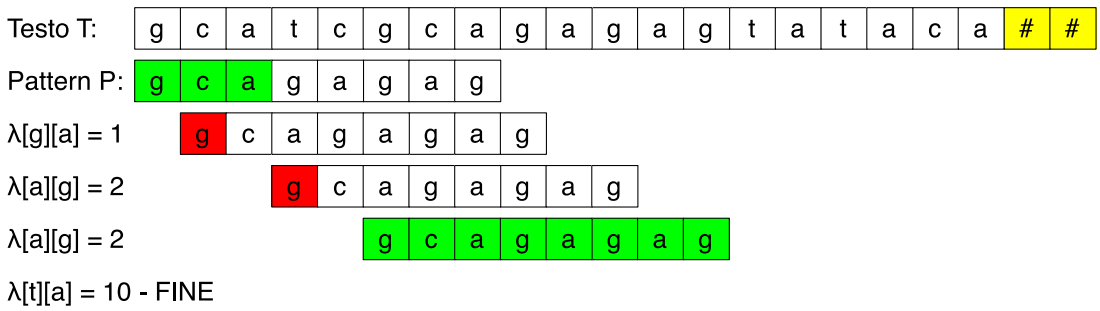


Figura 2.5: Esempio di esecuzione dell'algoritmo di Berry-Ravindran

Capitolo 3

Conclusioni

Lo scopo di questa tesina è stato quello di presentare il problema dello *string-matching* entrando nel dettaglio della definizione del problema e analizzando le principali tecniche algoritmiche presenti in letteratura.

Nel *Capitolo 1* è stato presentato e formalizzato il problema della corrispondenza tra stringhe ed è stato introdotto un algoritmo risolutivo, chiamato algoritmo "ingenuo", utile per avere un primo contatto con l'argomento trattato. La descrizione degli algoritmi di Knuth, Morris e Pratt, di Rabin-Karp e di Boyer-Moore, introdotti nel seguito, è caratterizzata dall'analisi delle scelte implementative e delle tecniche di ottimizzazione utilizzate; il tutto presentato anche tramite esempi ed illustrazioni atte ad aiutarne la lettura e la comprensione.

Nel *Capitolo 2* l'attenzione si è focalizzata sulla "famiglia degli algoritmi Boyer-Moore" ovvero quegli algoritmi che traggono ispirazione dalle scelte implementative di quello ben più noto di Boyer-Moore; per ognuno di questi è stato presentato lo pseudocodice, illustrata la complessità ed eseguito un esempio pratico per mostrarne il funzionamento.

Il problema della corrispondenza tra stringhe rientra in un vastissimo insieme di applicazioni che coprono i campi più disparati non solo dell'informatica, ma anche della biologia, della linguistica e persino della musica [13]. Benché l'importanza pratica del problema dello *string-matching* sia indubbia, ci si può domandare se nonostante la copiosa letteratura disponibile, sia ancora un argomento di qualche

interesse per la ricerca o per la didattica. La risposta è che tale problema non è, allo stato attuale, efficientemente risolto e necessita pertanto di ulteriore approfondimento in quanto le dimensioni delle basi di dati continuano a crescere tanto da mettere in difficoltà anche gli algoritmi più efficienti.

Appendice: Definizioni fondamentali per le stringhe, notazione e terminologia

Sia Σ un *alfabeto* di dimensione finita di *simboli* distinguibili.

Con il simbolo Σ^* si intende l'insieme di tutte le stringhe di lunghezza finita formate utilizzando caratteri dell'alfabeto Σ.

Una *stringa* x su un alfabeto Σ, è una successione finita di caratteri dell'alfabeto. Il numero di caratteri che costituiscono la successione viene detta *lunghezza* della stringa e viene indicata con $|x|$. La stringa di lunghezza zero viene detta *stringa nulla* ed è comunemente indicata con il simbolo ε, pertanto $|\varepsilon| = 0$.

I caratteri della stringa x di lunghezza n sono indicati con gli interi $1 \dots n$, x può quindi essere scritta come $x = x_1 \dots x_n$ oppure in notazione vettoriale $x = x[1] \dots x[n]$.

La *concatenazione* di due stringhe x e y, denotata con xy, ha lunghezza $|x| + |y|$ e consiste dei caratteri di x seguiti dai caratteri di y. Formalmente se $x = x_1 \dots x_n$ e $y = y_1 \dots y_m$ allora $xy = x_1 \dots x_n y_1 \dots y_m$. Naturalmente $x\varepsilon = \varepsilon x = x$ e quindi ε è l'elemento neutro per l'operazione di concatenazione.

Una stringa y è una *sottostringa* di x se esistono due stringhe u e v su Σ tali che $x = uyv$; in questo caso si dice anche che la stringa y *occorre* in posizione $i = |u| + 1$ in x.

Una stringa u è un *prefisso* di una stringa x, e si denota con $u \sqsubset x$, se $x = uy$ per qualche $y \in \Sigma^*$. Analogamente una stringa v è un *suffisso* di una stringa x, e si denota con $v \sqsupset x$, se $x = yv$ per qualche $y \in \Sigma^*$. Si noti che se $u \sqsubset x$, allora $|u| \leq |x|$ e analogamente se $v \sqsupset x$, allora $|v| \leq |x|$. È utile tener presente che $\sqsubset$ e $\sqsupset$ sono relazioni che godono della proprietà transitiva.

Una stringa y che occorre in una stringa x sia come prefisso che come suffisso, si chiama *bordo* di x ed in questo caso: $y = x[1, m] = x[n - m + 1, n]$

Se y è un bordo di x, allora esistono due stringhe z e w tali che $x = zy = yw$. In questo caso le due stringhe z e w hanno la stessa lunghezza $p = |z| = |w| = n - m$. La lunghezza p prende il nome di *periodo* della stringa x.

Bibliografia

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduzione agli algoritmi*, Jackson Libri, 1999.
2. D. E. Knuth, J. H. Morris, V. B Pratt, *Fast pattern matching in strings*, SIAM J. on Computing, pp. 323 - 350, 1977.
3. L. Colussi, *Algoritmi di pattern matching su stringhe*, Università degli studi di Padova, 2011.
4. R. Karp, M. Rabin, *Efficient randomized pattern matching algorithms*, IBM J. of Research and Development, pp. 249 - 260, 1987.
5. W. G. Horner, *A new method of solving numerical equations of all orders, by continuous approximation*, Philosophical Transactions of the Royal Society of London, pp. 308 - 335, 1819.
6. M. Abate, C. de Fabritiis, *Geometria analitica con elementi di algebra lineare*, McGraw-Hill Editrice, 2006.
7. R. S. Boyer, J. S. Moore, *A fast string searching algorithm*. Comm. ACM, pp. 762 - 772, 1977.
8. R. N. Horspool, *Practical fast searching in strings*, Software - Practice & Experience, pp. 501 - 506, 1980.
9. D. M. Sunday, *A very fast substring search algorithm*, Comm. ACM, pp. 132 - 142, 1990.

10. P. D. Smith, *Experiments with a very fast substring search algorithm*, Software - Practice & Experience, pp. 1065 - 1074, 1991.

11. A. Hume A, D. M. Sunday, *Fast string searching*. Software - Practice & Experience, pp. 1221 - 1248, 1991.

12. T. Berry, S. Ravindran, *A fast string matching algorithm and experimental results*, Prague Stringology Club Workshop, pp. 16 - 26,1999.

13. D. Cantone, S. Cristofaro, S. Faro, *Efficient algorithms for the δ-approximate string matching problem in musical sequences*, Prague Stringology Conference, pp. 69 - 82, 2004.

Indice delle illustrazioni