Algoritmo di Knuth-Morris-Pratt

L'algoritmo di Rabin-Karp utilizzava l'hashing per risparmiare sui tempi di matching e ci ha permesso di passare da una prestazione pari a $\Theta(NM)$, in cui N è la dimensione del testo e M la dimensione del pattern, ad una nel caso medio pari a $\Theta(N + M)$. Vediamo ora l'algoritmo di Knuth-Morris-Pratt che utilizza una struttura dati chiamata **LSP** o longest suffix-prefix per ottenere prestazioni ancora migliori.

LSP

Questa struttura dati ci permette di rispondere alla domanda: se abbiamo fatto il matching del prefisso s del pattern fino al carattere in posizione i, qual è la lunghezza del suffisso t di s tale che è anche prefisso di s?

Sapendo che $P[1..i]$ è prefisso di $T[k..i]$, voglio trovare il più lungo prefisso di $P[1..i]$ che è suffisso di $T[k..i]$.

Ad esempio se ho il testo `aaaa...` e il pattern `aaab` e inizio a fare matching mi fermerò alla posizione $i = 3$ nel testo e alla posizione $j = 3$ nel pattern. Ma come si può vedere dal testo questa ha 4 `a`, quindi posso shiftare il mio match a sinistra di 1 e riprendere da lì, non dovendo ricontrollare le ultime 3 `a`.

Per sapere però *quanto posso salvare* del matching fallito devo construire una tabella dei suffissi-prefissi più lunghi, che ad ogni lettera del pattern associa l'indice del più lungo suffisso che è prefisso del pattern.

```
LSP <- Array di dimensione pari alla lunghezza M del pattern P
LSP[0] = 0
for i = 1 to M
    j <- LSP[i - 1]
    while j > 0 and P[i] != P[j]
        j = LSP[j - 1]
    if P[i] = P[j]
        j <- j + 1
    LSP[i] = j
return LSP
```

Il caso base, cioè ad indice 0, è 0, ogni altro valore è calcolato a partire dal precedente e a retrocedere se il valore del pattern corrispondente non è uguale al valore del pattern che sto considerando, oppure quando $j = 0$, che significa che non esiste un suffisso valido. Se un carattere è uguale al precedente il suo valore sarà quello del precedente + 1, se invece è diverso dal precedente è necessario trovare il valore del più vicino carattere uguale e inrementarne il valore di 1. Esempio di pattern ed LSP:

```
Pattern ababab      Pattern aaabaaaaab    Pattern abacabab
LSP     001234      LSP     0120123334    LSP     00101232
```

Matching

L'algoritmo in sè è molto semplice e simile a come si costruisce la LSP

```
T <- testo di lunghezza N
P <- pattern di lunghezza M
j <- 0
for i = 0 to N
    while j > 0 and T[i] != P[j]
        j <- LSP[j - 1]
    if T[i] = P[j]
        j <- j + 1
        if j = M
            return i - (j - 1)
return -1
```

Se trovo un match incremento j, che mi indica un matching completo quando è uguale alla dimensione del pattern, altrimento retrocedo nella tabella LSP fino a quando non trovo un carattere uguale, oppure arrivo all'inizio.

Analisi efficienza

Per analizzare il tempo di questo algoritmo analizziamo il tempo di creazione della LSP e il tempo di matching. Il tempo per creare la LSP è $\Theta(m)$, il ciclo `for` viene eseguito $M - 2$ volte.

Per stimare il ciclo `while` dobbiamo analizzare la relazione tra j ed i, queste due variabili partono da 0 e 1 rispettivamente e possiamo dimostrare che $j < i$ durante tutta l'esecuzione perchè j è incrementata al più una volta per iterazione nel `for`, mentre i ad ogni iterazione, da questa ipotesi possiamo quindi affermare che per ogni j abbiamo che LSP$[j] < j$, quindi il numero di esecuzioni complessive del `while` è al più $i - 1$, ma i cresce fino a M, quindi $M - 1$.

Il tempo di esecuzione della procedura per costruire la tabella LSP è $M + O(M - 1) = \Theta(M)$. Per l'algoritmo di matching si utilizza un ragionamento analogo.