

Strings

Algoritmo de Knuth-Morris-Pratt

Prof. Edson Alves - UnB/FGA

1. Algoritmo de Morris-Pratt
2. Algoritmo de Knuth-Morris-Pratt

Algoritmo de Morris-Pratt

Motivação

- No algoritmo de contagem de ocorrências de uma substring P em uma string S por busca completa, as comparações feitas entre as substrings $S[i..j]$ e o padrão P são independentes
- Isto resulta em várias comparações sendo feitas, desnecessariamente, mais de uma vez
- Por exemplo, se $S = \text{"xyzab**cd**fgh"}$ e $P = \text{"ab**cd**e"}$, a comparação entre a $S[4..8] = \text{"ab**cd**f"}$ e o P falha apenas no último caractere ($\text{'f'} \neq \text{'e'}$), localizado no índice 8
- Como todos os caracteres de P são distintos, P não pode ocorrer em S a partir dos índices de 5 a 7, mas a busca completa ainda assim realiza tais comparações
- O algoritmo de Morris-Pratt explora justamente as comparações entre caracteres já feitas, movendo o índice de início das comparações entre as substrings e o padrão para a posição mais distante possível

- Um salto seguro s é um inteiro positivo tal que é garantido que o padrão P não pode ocorrer entre as posições i e $i + s$ de S , mas que pode iniciar-se da posição $i + s$ em diante
- Quando o padrão P contém apenas caracteres distintos, é seguro saltar para a posição onde aconteceu a falha
- Contudo, é preciso ter cuidado quando há repetições de caracteres no padrão
- Mais precisamente, para que o salto seja seguro, deve-se identificar a maior borda possível para $P[1..j]$, de modo a aproveitar as comparações bem sucedidas já realizadas
- O salto deve ser feito para a posição onde esta borda se inicia

Conceitos preliminares

- Considere que $S[i..(i + j - 1)] = P[1..j]$ e que $S[i + j] \neq P[j + 1]$
- Assim, o salto seguro $shift(P[1..j])$ de Morris-Pratt para o padrão $P[1..j]$, com $j = 1, 2, \dots, m$, é dado por

$$shift(P[1..j]) = j - |border(P[1..j])|$$

- Lembre-se de que $border(S)$ é a maior substring própria B de S (isto é, $B \neq S$), que é, ao mesmo tempo, sufixo e prefixo de S
- No caso especial de uma string vazia ($S[i..n]$ e P diferem já no primeiro caractere), o salto deve assumir o valor mínimo de 1, de modo que $shift(P[1..0]) = 1$
- Logo, se a comparação entre $S[i..n]$ e P falhou na posição $j + 1$ do padrão, a próxima comparação a ser feita é entre P e $S[(i + s)..n]$, onde $s = shift(P[1..j])$

Exemplo de bordas e de saltos seguros

j	$P[1..j]$	$border(P[1..j])$	$shift(P, j)$
0	""	-1	1
1	"a"	0	1
2	"ab"	0	2
3	"aba"	1	2
4	"abab"	2	2
5	"ababb"	0	5
6	"ababba"	1	5
7	"ababbab"	2	5
8	"ababbaba"	3	5
9	"ababbabab"	4	5
10	"ababbababa"	3	7
11	"ababbababab"	4	7

Pseudocódigo do algoritmo de Morris-Pratt

Algoritmo 1 Algoritmo de Morris-Pratt

Input: Duas strings P e S

Output: O número de ocorrências occ de P em S

```
1: function MORRIS-PRATT( $P, S$ )
2:    $m \leftarrow |P|, n \leftarrow |S|, occ \leftarrow 0, i \leftarrow 1, j \leftarrow 0$ 
3:    $bs \leftarrow \text{BORDERS}(P)$ 
4:   while  $|S[i..n]| \geq m$  do
5:     while  $j < m$  and  $P[j + 1] = S[i + j]$  do
6:        $j \leftarrow j + 1$ 
7:     if  $j = m$  then
8:        $occ \leftarrow occ + 1$ 
9:        $s \leftarrow j - bs[j]$ 
10:       $i \leftarrow i + s$ 
11:       $j \leftarrow \max\{0, bs[j]\}$ 
12:   return  $occ$ 
```

$S = \text{"abaabbabaabaaba"}$

$P = \text{"abaaba"}$

$$i = 1$$

$$j = 0$$

$$b_j = -1$$

$S = \text{"abaabbabaabaaba"}$

$P = \text{"abaaba"}$

$$i = 1$$

$$j = 1$$

$$b_j = 0$$

$S = \text{"ab aabbabaabaaba"}$

$P = \text{"ab aaba"}$

$$i = 1$$

$$j = 2$$

$$b_j = 0$$

$S = \text{"abaabbabaabaaba"}$

$P = \text{"abaaba"}$

$$i = 1$$

$$j = 3$$

$$b_j = 1$$

$S = \text{"abaabbabaabaaba"}$

$P = \text{"abaaba"}$

$$i = 1$$

$$j = 4$$

$$b_j = 1$$

$S = \text{"abaab"}\text{babaabaaba}$

$P = \text{"abaab"}\text{a}$

$$i = 1$$

$$j = 5$$

$$b_j = 2$$

Visualização do algoritmo de Morris-Pratt

$S = \text{"abaab}b\text{abaabaaba"}$

$P = \text{"abaab}a\text{"}$

$$i = 1$$

$$j = 5$$

$$b_j = 2$$

$S = \text{"abaabababaabaaba"}$

$P = \text{"abaaba"}$

$$i = 4$$

$$j = 2$$

$$b_j = 0$$

$$s = 3$$

$S = \text{"abaabababaabaaba"}$

$P = \text{"ababa"}$

$$i = 4$$

$$j = 2$$

$$b_j = 0$$

$S = \text{"abaab} \color{red}{b} \text{abaabaaba"}$

$P = \text{"} \color{red}{a} \text{baaba"}$

$$i = 6$$

$$j = 0$$

$$b_j = -1$$

$S = \text{"abaabbabaabaaba"}$

$P = \text{"abaaba"}$

$$i = 7$$

$$j = 0$$

$$b_j = -1$$

$S = \text{"abaabbabaabaaba"}$

$P = \text{"abaaba"}$

$$i = 7$$

$$j = 1$$

$$b_j = 0$$

Visualização do algoritmo de Morris-Pratt

$S = \text{"abaabbab} \color{blue}{\text{ab}} \text{aabaaba"}$

$P = \text{"} \color{blue}{\text{ab}} \text{aaba"}$

$$i = 7$$

$$j = 2$$

$$b_j = 0$$

Visualização do algoritmo de Morris-Pratt

$S = \text{"abaabbabaabaaba"}$

$P = \text{"abaaba"}$

$$i = 7$$

$$j = 3$$

$$b_j = 1$$

$S = \text{"abaabbabaabaaba"}$

$P = \text{"abaaba"}$

$$i = 7$$

$$j = 4$$

$$b_j = 1$$

Visualização do algoritmo de Morris-Pratt

$S = \text{"abaabbabaaba"}$

$P = \text{"abaaba"}$

$$i = 7$$

$$j = 5$$

$$b_j = 2$$

Visualização do algoritmo de Morris-Pratt

$S = \text{"abaabbabaabaaba"}$

$P = \text{"abaaba"}$

$$i = 7$$

$$occ = 1$$

$$j = 6$$

$$b_j = 3$$

S = "abaabbaba**aba**aba"

P = "abaaba"

$$i = 10 \qquad occ = 1$$

$$j = 3$$

$$b_j = 1$$

$$s = 3$$

Visualização do algoritmo de Morris-Pratt

$S = \text{"abaabbabaabaaba"}$

$P = \text{"abaaba"}$

$$i = 10 \qquad occ = 1$$

$$j = 4$$

$$b_j = 1$$

Visualização do algoritmo de Morris-Pratt

$S = \text{"abaabbabaabaaba"}\text{"}$

$P = \text{"abaaba"}\text{"}$

$$i = 10 \qquad occ = 1$$

$$j = 5$$

$$b_j = 2$$

Visualização do algoritmo de Morris-Pratt

$S = \text{"abaabbabaabaaba"}$

$P = \text{"abaaba"}$

$$i = 10 \qquad occ = 2$$

$$j = 6$$

$$b_j = 3$$

Complexidade do algoritmo de Morris-Pratt

- O algoritmo de Morris-Pratt realiza, no máximo, um número de comparações linear em termos dos tamanhos de S e P , a saber, $2n - m$ comparações
- Isto porque a comparação $P[j + 1] = S[i + j]$ pode falhar, no máximo, $n - m + 1$ vezes, já que o primeiro laço é executado $n - m + 1$ vezes, e pode ter sucesso, no máximo, n vezes, quando S e P são compostas por um mesmo caractere
- Caso a primeira comparação seja bem sucedida, ela não pode falhar no índice 0
- O pior caso, em termos de número de comparações, acontece quando P é formado por apenas duas letras distintas e S é uma repetição de $n - 1$ vezes a primeira letra de P e a última letra é igual a segunda letra de P

Complexidade do algoritmo de Morris-Pratt

- Por exemplo, $P = \text{"ab"}$ e $S = \text{"aaaaaaaaaaaab"}$
- Neste caso, a primeira comparação será bem sucedida $n - 1$ vezes, haverão $n - 2$ falhas (em relação ao último caractere do padrão) e uma última comparação bem sucedida no último caractere
- Daí o máximo de comparações será igual a

$$(n - 1) + (n - 2) + 1 = 2n - 2 = 2n - m$$

- Assim, o algoritmo MP é linear em relação ao tamanho do texto
- Porém, para determinar sua complexidade, falta determinar a complexidade da construção do vetor bs
- Se a construção de bs tem complexidade $O(m)$, o algoritmo de Morris-Pratt tem complexidade $O(n + m)$ no pior caso

Cálculo das bordas de S

- Observe que

$$border(S), border^2(S), \dots, border^k(S),$$

com $border^k(S) = ""$, é uma sequência de strings, decrescente em relação ao tamanho, cujos elementos são todas as bordas de S

- Este fato permite o cálculo de $b_j = |border(P[1..j])|$ para todos os prefixos de P em $O(m)$
- Observe que, $P[j+1] = P[b_j+1]$, então

$$b_{j+1} = 1 + b_j$$

- Isto porque a maior borda de $P[1..j]$ tem tamanho b_j , então se o caractere $P[j+1]$ coincidir com o caractere que sucede o prefixo que forma a borda, a maior borda de $P[1..(j+1)]$ será uma unidade maior do que a maior borda de $P[1..j]$

Cálculo das bordas de S

- Se $P[j+1] \neq P[b_j+1]$, então a borda de $P[j+1]$ deve ser reavaliada em termos da segunda maior borda de $P[1..j]$, isto é,

$$b_{j+1} = 1 + |\text{border}^2(P[1..j])| = 1 + b_j^2$$

se $P[j+1] = P[b_j^2+1]$

- Caso $P[j+1] \neq P[b_j^2+1]$, o raciocínio se repete até atingir a k -ésima borda de $P[i..j]$
- Portanto,

$$b_{j+1} = 1 + \max\{ \text{border}^i(P[i..j]) \mid P[j+1] = P[b_j^i+1], i \in [1..k] \}$$

- O caso base acontece no prefixo vazio, isto é, $P[1..0] = ""$, onde $b_0 = -1$, por conta do termo $+1$ na recorrência anterior

Implementação do algoritmo de Morris-Pratt em C++

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 vector<int> borders(const string& P)
6 {
7     int m = P.size(), t = -1;
8
9     vector<int> bs(m + 1, -1);
10
11     for (int j = 0; j < m; ++j)
12     {
13         while (t > -1 and P[t] != P[j])
14             t = bs[t];
15
16         bs[j + 1] = ++t;
17     }
18
19     return bs;
20 }
```

Implementação do algoritmo de Morris-Pratt em C++

```
22 int MP(const string& S, const string& P)
23 {
24     int n = S.size(), m = P.size(), i = 0, j = 0, occ = 0;
25
26     vector<int> bords = borders(P);
27
28     while (i <= n - m)
29     {
30         while (j < m and P[j] == S[i + j])
31             ++j;
32
33         if (j == m) ++occ;
34
35         int shift = j - bords[j];
36         i += shift;
37         j = max(0, j - shift);
38     }
39
40     return occ;
41 }
```

Algoritmo de Knuth-Morris-Pratt

Diferenças entre os algoritmos MP e KMP

- O algoritmo de Morris-Pratt tem como invariante

$$inv(i, j) = (P[1..j] = S[i + 1, i + j + 1]),$$

o qual permite os saltos seguros e que resulta na complexidade $O(n + m)$ no pior caso

- Contudo, este invariante pode ser melhorada, ao se incorporar a propriedade da diferença, isto é,

$$inv2(i, j) = (P[1..j] = S[i + 1, i + j + 1]) \textbf{ and } (P[j + 1] \neq S[i + j + 1])$$

- Para entender o porquê da melhora, considere o seguinte exemplo: seja $P = \text{"abcabc"}$ e $S = \text{"abcabdabc"}$
- Os 5 primeiros caracteres de ambos coincidem e a diferença ocorre no sexto caractere:
 $P[6] = \text{'c'} \neq S[6] = \text{'d'}$

Diferenças entre os algoritmos MP e KMP

- Como $\text{border}(P[1..5]) = \text{"ab"}$, cujo tamanho é igual a 2, a próxima comparação seria entre $P[3]$ e $S[6]$
- Contudo, esta comparação é idêntica a anterior, pois a borda **"ab"** não é própria, isto é, o próximo caractere (**'c'**) gera uma nova borda **"abc"**
- A contribuição de Knuth para o algoritmo de Morris-Pratt é essa: incorporar a propriedade da diferença e definir as bordas estritas, nas quais o próximo caractere não gera uma nova borda, evitando comparações já realizadas

Bordas estritas

- Uma borda estrita de $S[1..j]$ ($sborder(S[1..j])$) é a maior borda própria $b = [1..k]$ de S tal que $S[j+1] \neq S[k+1]$
- Assim, os coeficientes b_j podem ser redefinidos como

$$b_j = |sborder(S[1..j])|,$$

ou $b_j = -1$, caso não exista tal borda

- Observe que a borda b pode ser uma string vazia
- Esta modificação melhora o tempo de execução de algoritmo, embora sua complexidade assintótica permaneça a mesma: $O(n + m)$
- A implementação é quase idêntica a do algoritmo de Morris-Pratt, residindo a diferença na substituição de $border(P)$ por $sborder(P)$

Exemplo de bordas estritas e de saltos seguros

j	$P[1..j]$	$sborder(P[1..j])$	$shift(P, j)$
0	" "	-1	1
1	"a"	0	1
2	"ab"	-1	3
3	"aba"	0	3
4	"abab"	2	2
5	"ababb"	-1	6
6	"ababba"	0	6
7	"ababbab"	-1	8
8	"ababbaba"	0	8
9	"ababbabab"	4	5
10	"ababbababa"	0	10
11	"ababbababab"	4	7

Implementação do algoritmo KMP em C++

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 vector<int> strong_borders(const string& P)
6 {
7     int m = P.size(), t = -1;
8     vector<int> bs(m + 1, -1);
9
10    for (int j = 1; j <= m; ++j)
11    {
12        while (t > -1 and P[t] != P[j - 1])
13            t = bs[t];
14
15        ++t;
16        bs[j] = (j == m or P[t] != P[j]) ? t : bs[t];
17    }
18
19    return bs;
20 }
```

Implementação do algoritmo KMP em C++

```
22 int KMP(const string& S, const string& P)
23 {
24     int n = S.size(), m = P.size(), i = 0, j = 0, occ = 0;
25
26     vector<int> bs = strong_borders(P);
27
28     while (i <= n - m)
29     {
30         while (j < m and P[j] == S[i + j])
31             ++j;
32
33         if (j == m) ++occ;
34
35         int shift = j - bs[j];
36         i += shift;
37         j = max(0, j - shift);
38     }
39
40     return occ;
41 }
```

1. **CHARRAS**, Christian; **LECROQ**, Thierry. *Handbook of Exact String-Matching Algorithms*¹
2. **CROCHEMORE**, Maxime; **RYTTER**, Wojciech. *Jewels of Stringology: Text Algorithms*, WSPC, 2002.
3. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.

¹[Morris-Pratt Algorithm](#)